

Operating System for the Raspberry Pi

A Kernel and simple Shell for the 64-bit Raspberry Pi 3, developed from an historical perspective on the structure of early Operating Systems.

Sam Whitehead - 14325283
psysrw@nottingham.ac.uk
Msci Computer Science

COMP4029: Individual Programming Project
Project Supervisor: Steve Bagley
University of Nottingham

Abstract

In this project I aimed to create an Operating System (OS) for the Raspberry Pi (RPi) 3. I originally intended to follow the design of the Unix system from the beginning, but at the start my work did not seem to be heading in that direction.

Over the course of developing this OS, I learned a lot about which OS components are needed at which stages of development. This project has followed a logical “first discovery” journey into OS development, which shadowed the historic major advances in OS technology.

The OS I have created provides a kernel, with a system call interface, most of the infrastructure for a multiple-process execution model, an implementation of the exFAT Filesystem (FS), and a basic shell which allows users to run several different basic commands.

The final OS is not intended for daily use, but rather it is an example of a typical project for those wanting to learn about OS development. The source code for the project is made freely available online as a learning tool.

Contents

1	Introduction	4
1.1	What is ARM64?	4
1.2	What is an Operating System? (And Unix)	4
1.3	Why the Raspberry Pi?	4
2	Motivation	4
3	Related work	5
3.1	Filesystem code and memory management	5
3.2	Central Processing Unit (CPU) mode initialisation for arm CPUs	5
3.3	Universal Asynchronous Receiver-Transmitter (UART) serial Input / Output (IO)	5
3.4	Other related works	5
4	Description of the work	6
4.1	The goals of the project	6
4.2	Specification of the project	6
4.2.1	Tasks	6
4.2.2	Work Packages	6
5	Methodology	6
5.1	Setting up the build system	6
5.1.1	Cross-compilation terminology and tools	8
5.2	The build process - Make and Makefile	8
5.3	Software tools used to aid development	10
5.3.1	Source code editing	10
5.3.2	Raspberry Pi 3 Emulator	10
5.3.3	Debugging	10
5.3.4	Organising the build process	10
6	Design	11
6.1	Operating System (OS) components block diagram	11
6.2	User Interface and the graphics of the OS	11
7	General implementation details	12
7.1	Programming languages used	12
7.2	Problems encountered with the original implementation details	12
8	The Kernel	12
8.1	Design	12
8.1.1	Bootloader	13
8.1.2	UART driver for keyboard input and initial console output	13
8.1.3	Graphics driver for displaying pixels on the screen	13
8.1.4	System calls	13
8.1.5	SD card driver	14
8.1.6	exFAT filesystem	14
8.2	Implementation	14
8.2.1	Bootloader	15
8.2.2	System Calls	15
8.2.3	CPU setup	15
8.2.4	exFAT Filesystem (FS)	16
8.2.5	Graphics driver	18
8.3	Evaluation	18

9	The Shell	19
9.1	Design	19
9.1.1	Features I expect to complete	19
9.1.2	Nice-to-haves	20
9.1.3	Ideas for next steps	20
9.2	Implementation	20
9.2.1	Getting input from the keyboard	20
9.2.2	Storage of enabled commands	21
9.2.3	Shell commands as functions	21
9.3	Evaluation	21
10	Evaluation and External Aspects	22
10.1	Overall evaluation of the results	22
10.2	Evaluation in relation to the external aspect	22
11	Summary and Reflections	22
11.1	Project management	23
11.1.1	Project management – Re-planning the project	23
11.1.2	Unexpected difficulties in development	23
11.2	Contributions and reflections	23
11.3	Foo	24
	Acronyms	25
	References	25
	Appendices	I
A	Interim: Appendices	I
A	The original project plan	I
B	Work tasks	I
C	Gantt chart	II
D	Work Packages	II
E	Future plans block diagrams	IV
	Acronyms	IV

1 Introduction

TODO: Introduction setting out the aims and objectives of your project, **explaining the overall intention of the project and specific steps that will be taken to achieve that intention.**

First, an introduction to some of the concepts on which this project relies.

1.1 What is ARM64?

- ARM64 (or more commonly Aarch64) is the name given to the 64 bit architecture of arm Central Processing Units (CPUs).

1.2 What is an Operating System? (And Unix)

Operating Systems (OSes) are programs which run directly on top of computer hardware and allow other software to run on the computer. They come in many different varieties, but this project will be based on the “microkernel” design. Unix was an OS developed at Bell Labs of AT&T in the 1970s. It was originally written in assembly language for the specific target machine, the PDP7, but for its 4th version it was rewritten in C, a new language at the time (also created at Bell Labs). Being written in C made the OS very portable, as only the compiler had to be ported for the entire system to run on a new machine. This made Unix very popular, and soon there were many Unix-like OSes. ‘A Unix-like OS is one that behaves in a similar manner to a Unix system’ ([22]). There are many examples of Unix-like OSes, including some very popular ones like Linux, MacOS, and the family of OSes known as the BSDs. The OS I am creating in this project aims to be Unix-like in its behaviour.

1.3 Why the Raspberry Pi?

The Raspberry Pi (RPi) is a cheap Single Board Computer (SBC) powered by an arm System-On-a-Chip (SOC). This project focuses on the RPi 3, the first iteration of the RPi to use a 64-bit CPU. The RPi is ideal for a project like this, as it is complex enough that creating an OS for it is a challenge, but also modern enough that no out-of-date technologies will be necessary to work with it. The RPi board includes a number of different components that would require drivers to function, so there would be plenty of room for additional features if the project moved faster than anticipated.

The RPi is popular among hobbyist OS developers, and its hardware schematics and technical manuals are made available, so there are plenty of resources available online for documentation of technical details. The OS I have developed is intended to be a learning resource for others who wish to discover more about OS design. The source code will be made available under an open source license.

2 Motivation

TODO: Motivation explaining the problem being solved and its importance or need for it.

TODO: explain the problem being solved

My personal motivation for choosing to create an Operating System (OS) for this project stems from my interest in low-level programming. I have an interest in understanding how computers work at the lowest level, and the best way to learn how an OS is implemented and the reasons for design decisions will be to implement my own OS at that low level.

With this project, I wanted to develop a deeper understanding of the inner workings of OSes and how the programs I run on my computer interact with the hardware they run on. I wanted to understand the entire software stack, and one large gap in my knowledge was the internals of the Operating System. It is a black box to most students of Computer Science, and I was determined to fill that gap in my own understanding. I wanted to know how system calls work, and how they relate to the concept of a “secure monitor”. I already knew how filesystems work in theory, but I wanted to experience implementing one to gain a practical appreciation of the topic.

TODO: why is this project “needed” or “important”?

- Understanding common OS choices from an historical perspective by analyzing original OS methods

I believe that this project is needed because although OSes are a common hobby project, there are no OS development projects which have aimed to highlight the motivation behind the choices that have been made by the largest OSes. No project exists which shows clearly the reasons behind why OS development as a field has taken the direction that it has. Why does every major OS use a software interrupt based system call architecture to distinguish between user-level code and privileged kernel operations? What are some of

the reasons for choosing to include hardware drivers in the kernel of an OS instead of running those drivers as user code? These questions, and others like them, have not been answered to a suitable level by other projects.

3 Related work

3.1 Filesystem code and memory management

The project “rpi-boot” [Rw6] contains code for a File Allocation Table (FAT) Filesystem (FS) and also some code for common libc functions. It includes an implementation of the `malloc` function [Rw1], used to allocate memory.

I will use the FS code from this source as inspiration for the FS of my Operating System (OS). The `malloc` implementation will also be used as the basis for my memory management system.

3.2 Central Processing Unit (CPU) mode initialisation for arm CPUs

A project [Rw3] which contains assembly code to switch arm 64-bit architecture CPUs into the correct operation mode for an OS. This code will be very useful when I write the boot code, as I will want to setup the processor into the correct Exception Level (EL) for the kernel.

This project also contains example code for Exception handlers, which is something I will need in order to implement System Calls (subsection 8.2.2).

3.3 Universal Asynchronous Receiver-Transmitter (UART) serial Input / Output (IO)

A set of tutorials on Github [Rw4] contains code which enables debugging via the UART serial pins on the Raspberry Pi (RPi). I will use this code as a reference when I implement a library for the two UART outputs for the RPi.

This project’s later lessons also implement code for the framebuffer and for arm ELs, but I will instead reference documentation and other implementations when I work on these for my OS.

3.4 Other related works

Other related works are different OSes. The Linux kernel [Rw7] is a well known monolithic kernel, and supports numerous different CPU architectures, including Aarch64. I will look at the Linux kernel’s implementation of system calls when I start working on the system call interface for my OS.

Another OS which I will look at is NetBSD [Rw2] and the rest of the BSDs. The source code for NetBSD will be simpler and easier to understand than that of Linux, so I will reference BSD implementations of any OS components which are overly complicated in the Linux kernel.

The RiscOS operating System [Rw5] was the first OS written for arm CPUs. It has been kept updated and a version is available which runs on the RPi. The entire OS is written in arm assembly language, so it will not be particularly useful to me, but I can take a look at its implementation of any components I get particularly stuck on.

References for the Related work

- [Rw1] Doug Lea. *DL malloc, a version of malloc/realloc/free*. Version 2.8.6. URL: <http://gee.cs.oswego.edu/pub/misc/malloc.c> (visited on 2021-11-28).
- [Rw2] *NetBSD*. URL: <https://github.com/NetBSD/src> (visited on 2021-11-28).
- [Rw3] *raspberrypi-os: Learning OS development using Linux kernel and RPi*. URL: <https://github.com/s-matyukevich/raspberrypi-os> (visited on 2021-11-28).
- [Rw4] *raspi3-tutorial: Bare metal RPi 3 tutorials*. URL: <https://github.com/bztsrc/raspi3-tutorial> (visited on 2021-11-28).
- [Rw5] *RiscOS*. Version 5.29. URL: <https://www.riscosopen.org/content/> (visited on 2021-11-28).
- [Rw6] *rpi-boot: A second stage bootloader for the RPi*. URL: <https://github.com/jncronin/rpi-boot> (visited on 2021-11-28).
- [Rw7] *The Linux kernel*. Version v5.15. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/snapshot/linux-5.15.tar.gz> (visited on 2021-11-28).

4 Description of the work

TODO: Description of the work explaining what your project is meant to achieve, how it is meant to function, e.g., perhaps even a functional specification for a software-oriented project.

This project will produce a piece of software. The software will aim to achieve two main goals: to be an Operating System (OS) and to be a useful starting point for anyone wanting to teach themselves OS development.

4.1 The goals of the project

To meet the first goal, the software produced should qualify as an OS. For this, it should have its code organised into a “kernel” structure. There should be drivers for the essential hardware, such as the serial port (for a debug console), the framebuffer (for graphical output), and the eMMC chip, which controls access to SD cards on the Raspberry Pi (RPi). There should also be enough filesystem code to allow for reading and writing of files, and there should be implementations of familiar standard library file I/O wrapper functions, including `fread` and `fwrite`.

For the second goal (being a useful starting point for someone learning about OS development), the OS I produce needs to be simple enough for a competent programmer to get a basic understanding of how all of the components fit together, within only a couple of hours of first studying the source code. As with the first goal, it will need to include all of the required features for a basic OS, and these were described previously. It should also have plenty of room for future development, as I believe that one of the best ways to understand a topic (and to become familiar with an existing project) is to try to develop a new feature involving the topic or extend the project.

4.2 Specification of the project

I specified the components of the project, and these components are described below. The components are split into two parts: the **Tasks** of the OS, and a selection of **Work Packages** I will work on.

4.2.1 Tasks

Tasks are the individual components of the project, separated into the components of the Kernel and Shell, the miscellaneous parts, and the documents I will need to produce for the project. These tasks are enumerated in **Table 1** on page 8 and then assembled into a Gantt chart in **Figure 1**.

Some of the tasks are different to the ones in the original table from the project proposal, which can be found in the Appendix (**Table 3**). Some tasks have changed places, and some have been given extra time. Also, some of the tasks have been completely removed as I decided that I will not have enough time to complete them (primarily: init system and service manager, and multithreading).

4.2.2 Work Packages

A “Work Package” is a collection of tasks with a well-defined deliverable. Not every task will be part of a work package, and some tasks will be work packages on their own. **Table 2** contains all of the work packages for this project. Some changes have been made to the descriptions of the deliverables from the work packages listed in the original table of work packages from the initial project proposal (**Table 4**, in the Appendix). The work package “Init system and service manager” has been removed, as I do not think I will have enough time to create a working init system, and a service manager is not an essential part of an OS, especially when the OS is very simple and will not need to run any services.

5 Methodology

TODO: Methodology describing the (theoretical / experimental / analytical / numerical / software development-based / research-based, etc.) methodologies / techniques / tools / algorithms / technologies / etc. that are relevant to the project topic justifying the choices made for the project, where possible with supporting evidence derived from the existing work.

5.1 Setting up the build system

For this project, I could not use a traditional compilation setup. Normally, a program runs in a hosted environment, which means that it can use a standard library for some basic functions. In the case of C code, this is `libc`. My Operating System (OS) runs on “bare metal”, so it cannot use any of the standard header

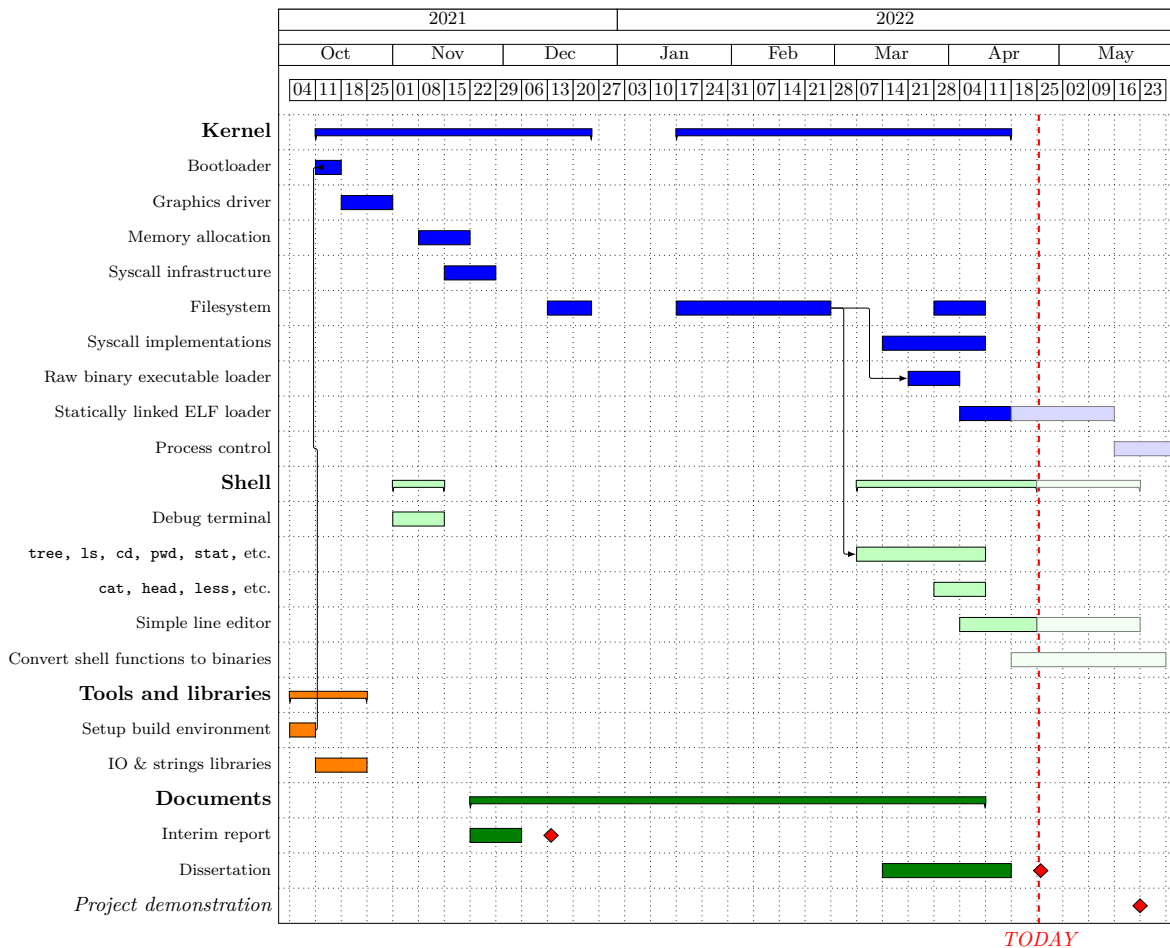


Figure 1: Gantt chart

Task	Duration (weeks)
Kernel	
Bootloader	1
Graphics driver	2
Syscalls	7
Memory management	5
Filesystem	4
Statically linked ELF Loader	2
Process control	8
Shell	
Debug terminal	2
<code>pwd</code> , <code>cd</code> , <code>ls</code> , <code>stat</code> , etc.	1
<code>export</code> , variables, <code>set</code> , etc.	2
<code>if</code> , <code>while</code> , <code>for</code> , <code>case</code> , globs, etc.	3
Command substitution	4
IO pipes and output redirection	1
Tools and libraries	
Setup cross-compiler and build environment	1
IO and strings libraries	2
libc	8
Programs	
<code>cat</code> , <code>head</code> , <code>less</code> , etc.	2
<code>roff</code>	2
<code>man</code>	1
Documents	
Interim report	2
Documentation	4
Dissertation	7

Table 1: The plan of work for the project, including how many weeks I think each component will take.

files provided by `libc`, which means that the entire C standard library is unavailable. I still have access to some header files, including `stddef.h` and `stdint.h`, as these are provided as part of the compiler. There are other header files available in this “freestanding” environment, but these two were the main ones for the first stages of development, since they define types such as `uint32_t` and `size_t`.

In addition to the compiler flags needed for freestanding development, I also needed a completely different compiler. This relates to the **Architecture** of the system running the compiler. The Raspberry Pi I used for this project has an architecture of **Aarch64**, or 64-bit ARM, whereas desktop PCs are usually **x86** or **x86_64** architecture computers. This means I needed a compiler that generates machine code for **Aarch64** based systems. Such a compiler would be called a **Cross-compiler** if it runs on a different architecture compared with the machine code it produces.

5.1.1 Cross-compilation terminology and tools

- **Build:** The architecture of the computer used to compile the program.
- **Host:** The architecture of the computer the program will run on.
- **Target:** The architecture the program builds for (only used when the program is or contains a compiler).

For this project I used the LLVM suite of tools, as it was designed to be a cross-compiler first, so all of the tools (compiler, assembler, binary manipulation tools, etc.) are capable of handling the required Aarch64 host system. The compiler, “clang”, takes a command-line argument which indicates the architecture of the host system (LLVM ignores the difference between host and target, calling them both “target”).

5.2 The build process - Make and Makefile

TODO

Work package	Description of the deliverable	Progress
Debug terminal using the graphics driver	A debug terminal which is printed on the RPi's display. The debug terminal should print important values to the display while the OS is starting up, allowing me to debug the features I am working on easily by verifying that memory addresses are set correctly and that functions are performing their behaviours as I expect them to.	Finished.
A basic kernel which can load (from disk) and jump into a compiled ELF program	A program that can load a statically compiled ELF executable binary into memory and then jump to its entry point. This requires a filesystem library to be implemented, and also includes the ELF loader task.	In progress.
Scheduling algorithm	A scheduling algorithm with support for multiple processes running on the system concurrently. This will include an implementation of <code>fork(3)</code> or a similar function in order to spawn new processes and <code>execve(2)</code> to replace the current program with a new process. It will also require process control infrastructure in the kernel to keep track of the running processes.	Not started.
A shell	A (POSIX-compliant) shell or a port of a simple shell like Dash[9]. If I decide that implementing my own POSIX-compliant shell is too large of a task, then I will port the source code for Dash to run on my OS, implementing the required syscalls to get it working. This shell will be the main way users interact with the OS.	Not started.
Documentation	This package contains several parts. I will create documentation for how each part of the OS works, and a guide for how new programmers can get started developing programs for the system. The documentation will be completed in stages as I develop the OS, and I will go back to keep it up-to-date as I make changes to components and add new features.	Not started.
Multithreading support and multi-core scheduling	The RPi 3 has 4 CPU cores. This work package will enable users to take advantage of the additional processing power of the other 3 cores for their programs. This includes a threading library similar to <code>pthread</code> on Linux systems, and improvements to the scheduler so that it can give different CPU cores to multiple threads owned by the same process. This work package is optional.	Not started.

Table 2: The work packages of the project and their deliverables.

5.3 Software tools used to aid development

5.3.1 Source code editing

For editing the source code of the OS, I use the text editor Vim [23] with several plugins to assist me in writing correct code. I am using the CoC extension [8] for autocompletion, and the clangd language server [6] extension for CoC in order to provide source code linting for C source files.

For source code formatting, I run the program `clang-format` [7] on the C source and header files. I have defined a style guide for the project in a `.clang-format` configuration file, which tells `clang-format` how to format my code. This results in a consistent style throughout the entire project, causing the source code to be easier to read and understand.

5.3.2 Raspberry Pi 3 Emulator

QEMU [1] is an emulator with out-of-the-box support for emulating the host architecture. This means that on an x86 machine, we can emulate another x86 machine. This isn't enough for emulating a Raspberry Pi (RPi), as the RPi uses the Aarch64 architecture. Thankfully, there is a package on most linux distributions, usually called something like `qemu-arch-extra`, which adds support for other architectures. This provides the program `qemu-system-aarch64`, which spins up an emulator for Aarch64-based machines. The RPi 3 is supported as a machine preset, using the command line argument `-machine raspi3`.

The emulator can emulate everything about the RPi except for the first stage bootloader, which is the code that runs at the very start of the boot process on a real RPi and then loads the kernel (`kernel8.img` for 64 bit mode). This bootloader is proprietary and runs on the RPi's VideoCore GPU. I am not trying to develop a first stage bootloader, though, so I do not need to worry about the lack of this functionality. The compiled kernel is passed as an argument to the emulator when we start it (`-kernel kernel8.img`). In the same way, we can give the SD card image to the emulator, with the argument `-drive file=sd.img,if=sd,format=raw`.

5.3.3 Debugging

GDB [11] is a debugger which can be used to debug executable programs. Similar to QEMU only supporting one architecture by default, GDB only supports running a program directly on the current machine. However, it does support remote debugging with a commands once it has started (`target remote URL:PORT`). Like with QEMU, there is an additional package which provides support for additional architectures (`gdb-multiarch`). When we launch QEMU, we can tell it to wait for a remote debugging session to connect before it starts the emulator (using the commandline arguments `-s -S`). Once QEMU is waiting for the debugger, we launch GDB (`gdb-multiarch kernel8.elf` in a terminal) and then connect to the emulator as a remote debugging client (using the GDB command `target remote localhost:1234`).

Now we can debug the OS kernel as if it is a regular program, using breakpoints and disassembling it as we run through the execution. This is how I debug the kernel when a new feature is not running as expected, and it allows me to determine the cause of any issues more easily than a debug shell would have done. Also, the debugger works without needing a working kernel, so I can debug issues which cause the kernel to fail.

5.3.4 Organising the build process

When I first started this project, I was using CMake to automate the build process. CMake is a tool which tries to automatically find the correct dependencies required for compiling a project (the compiler, linker, and also any libraries and included files can all be handled by CMake). Unfortunately, CMake was causing issues with compiler and linker paths, and it was not consistently finding the correct linker for the compilation of this project.

I then rewrote the build system using GNU Make (with a Makefile). This allowed me to hardcode the correct compiler and linker for the build process, and I haven't encountered any more issues with the compilation toolchain.

The top-level Makefile does not contain any rules to invoke the compiler directly, and only handles the linking together of the compiled object files into the final kernel. A separate Makefile is used to compile each source file, and Make invokes itself recursively in each source directory for the compilation.

The source files are kept separate from the object files and compiled kernel, by putting all the results of compilation into a `build/` directory at the top of the project (source files are located in `src/`). The object files and final compiled binaries are also separated, with object files being in `build/obj/` and binaries in `build/bin/`. Keeping the directory structure organised in this way helps maintain an understanding of the overall project structure as the project grows in size, as not keeping the order in this way would result in quickly becoming overwhelmed by the sheer number of files.

6 Design

TODO: shell, kernel get their own sections (with design, implementation, and evaluation), the rest of the components are discussed in a section after that.

FIXME: Write about how this diagram is from a book, the differences from the actual design.

6.1 Operating System (OS) components block diagram

Figure 2 is a diagram of how the different components of the OS will fit together. It shows how user programs running on the OS would be supported by the OS components. The components mentioned include user libraries (which would be included at `/usr/include/` in the Filesystem (FS)), as well as various parts of the kernel, such as system calls (see [subsubsection 8.2.2](#)), the FS ([subsubsection 8.2.4](#)), and the “process control subsystem”. The “process control subsystem” does not currently have any implementation, but it will be made up of the parts of the kernel which control the processes that run on the Central Processing Unit (CPU) at any particular time.

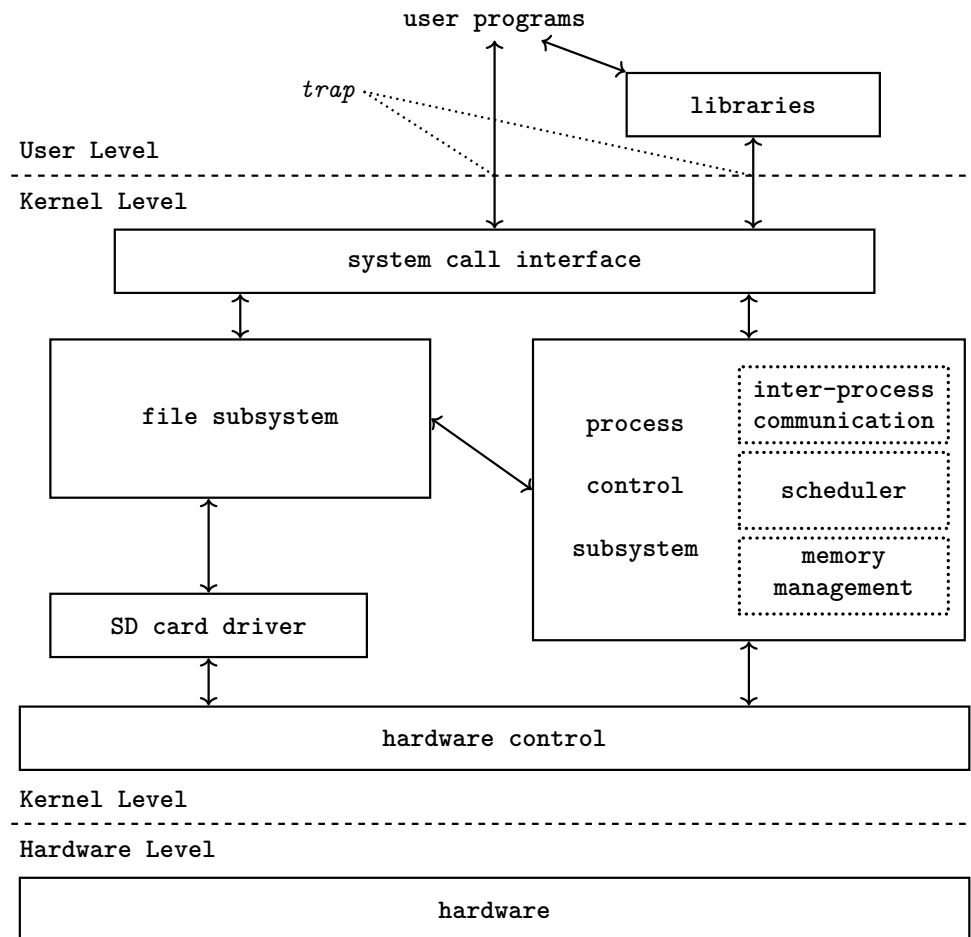


Figure 2: A diagram of how the components of the operating system will interact with each other. Inspired by a diagram from a book on Unix (Figure 2.1 from [4]). The versions of this diagram which are available online are low-resolution, so the version provided here is in a vector format, and this document’s source code is available online [21].

6.2 User Interface and the graphics of the OS

The interface of the OS is the console. It is the text that is displayed to the user from the Raspberry Pi (RPi). I will implement some quality of life features, such as terminal scrollbar, which allows the user to

scroll up and see what was printed to the terminal earlier. This feature reduces the need for a terminal output pager such as `less`, which will not be implemented until very late in the development.

The font I decided to use for the console is Bizcat [5]. It is an 8x16 bitmap font, and I converted it into a C source file as a constant array. The font is easy to read at a screen resolution of 640x480, which gives 80 columns and 30 rows of text.

The colorscheme of the console is something that will likely change many times during development, until I find something that I am completely happy with. At the moment, the background is completely black, and the text is completely white, but this would probably put strain on the users' eyes after using the OS for a long time. I will experiment with color schemes once I have implemented a program that can output a text file to the screen.

7 General implementation details

TODO: rename this section ('build system' or 'compilation toolchain'?)

7.1 Programming languages used

From the beginning of this project, I had already decided to write the majority of the code for the Operating System (OS) in C [25]. C is primarily a systems implementation language (making it perfect for writing an OS), and I was already very familiar with it, having used it regularly for the past 5 years.

For some very low-level parts of the OS, I needed to specify the exact order in which I wanted arm instructions to be executed, so ARMv8 assembly was the natural choice, being the closest language to the machine code that gets executed without actually being the machine code. Assembly language allows preprocessor macros, so I did not have to manually align instructions to 4-byte boundaries anytime there was a change, and header file "include"s which allow register values to be abstracted away from the logic and replaced with meaningful names. These features were used extensively in the bootloader and in [subsubsection 8.2.3](#) for explaining the values placed into various Central Processing Unit (CPU) registers.

7.2 Problems encountered with the original implementation details

At the start of development, I was using the GNU cross-compiler for bare-metal arm 64-bit targets, `gcc-aarch64-none-elf`. This was working but the build system I was using at the time (CMake) was easier to configure using clang/llvm as a cross-compiler. Also, the GNU cross-compilers are not available in most Linux distributions' core repositories, so clang/llvm is much easier to install, needing only the package manager of a Linux system.

Later in development, I was having difficulties maintaining the build system which I had started the project with (CMake), as the intended usage is to automatically detect the executables that should be used for compilation (the cross-compiler, linker, etc.). This caused problems because if I modified the status of my build computer, by installing some new packages for example, CMake would detect and start using the wrong linker, causing the build process to fail*.

As a result of the problems I was having with CMake, I decided to use GNU Make (with a `Makefile` instead of a `CMakeLists.txt`, calling the `make` executable) instead. I created a `Makefile` for the project, and this allowed me to hardcode the name of the compiler executable, and all the tools needed to generate the compiled kernel image and SD card image. During the implementation, as the number of source code files grew, I decided to start using sub-make [17] for compilation so that the main `Makefile` could be kept simpler, so I moved the compilation commands into a separate `Makefile` which handles each subdirectory of the main `src/` directory.

8 The Kernel

TODO

8.1 Design

TODO: Design containing a comprehensive description of the design chosen, how it addresses the problem, and **why it is designed the way it is**.

*I later discovered the cause of this. I had re-installed the `gcc-aarch64-none-elf` cross-compiler, which resulted in the GNU arm 64-bit linker appearing before llvm's `ld.lld`. CMake then tried to use `gcc` (**not** the cross-compiler) to invoke the linker, but the arguments for invoking the linker via `gcc` and `clang` are different, so `gcc` was giving back an error, saying that the `-T` argument was missing a parameter, and I was unable to force CMake to call `ld.lld` directly.

8.1.1 Bootloader

On the Raspberry Pi (RPi) 3, the Graphics Processing Unit (GPU) contains a first-stage bootloader which automatically does most of the required setup for booting into an Operating System (OS) kernel. However, the Central Processing Unit (CPU) will not be setup completely, and so I will need to write some code to set register values and verify that everything is as it should be before we jump into the kernel `main` function. I will call this code the ‘second-stage’ bootloader.

8.1.2 UART driver for keyboard input and initial console output

Universal Asynchronous Receiver-Transmitter (UART) is used on the RPi for serial communication, with two different UART ports: a mini-UART and a full PL011 UART [15]. The PL011 UART is faster and more reliable than the mini-UART, but it is disabled by default, enabling it disables bluetooth, and it is more complex to program for. I will use the mini-UART to debug the setup of the PL011 UART, which will be used as the main serial connection once it is enabled and working. I won’t be using bluetooth in this project, but future projects which expand on my code could switch back to the mini-UART if they wanted to use the RPi’s bluetooth chip.

Serial communication is used in this project as a means of interacting with the OS, as I will not be implementing support for USB peripherals. All keyboard input is expected to come through the serial port. I will also use the serial port for debugging as I develop the graphics driver (see [subsection 8.1.3](#)).

For the initial console output, I will implement a `printf` function for formatted printing of characters, integers, hexadecimal numbers, and strings. This will allow me to easily print the values of registers, so that I can make sure that all values are as I expect during the boot process. The `printf` function will be generic, so that I can use the same `printf` function for both of the UARTs and also eventually for the graphical console.

The OS will get keyboard input from the serial port, which will be provided by QEMU, the RPi emulator. QEMU uses VT100 escape sequences [24] for its serial interface, so normal keys on the keyboard (and shifted keys) are passed to the OS as the ASCII value of the characters they represent.

8.1.3 Graphics driver for displaying pixels on the screen

Once I have a working serial terminal, I will implement a graphics driver for interacting with the RPi’s framebuffer. This will include a function for setting individual pixels, a function for setting the entire screen to a colour, and a function to display a bitmap to the screen at a given location. With these functions, I will write functions for displaying characters to the screen, with variables for the current position on the screen. I will convert a bitmap font [5] into a constant array of bitmap values, and the character printing function can use the character as an index into this array to get the bitmap to display to the screen.

The screen blanking function will use a `memset` function to set the entire framebuffer to a given value, as this would be much faster than setting each pixel individually. The bitmap function will use the pixel setting function internally, and will use the bitmap to decide which pixels should be set. Each of these functions is very simple on its own, and will be relatively easy to implement, but once put together they will create the complex behaviour required for displaying characters to the screen.

8.1.4 System calls

TODO: explain that syscalls are split into 2 different components: the mechanism to allow for them and the implementations of the syscall functions themselves.

System call mechanism using software interrupts The kernel will include a system call mechanism, which would eventually be the way for processes (at the user level) to escalate privileges temporarily to request that the kernel perform some action on their behalf. Processes and the user / kernel level split will not yet be implemented, but I will implement support for system calls to prepare for the time when they are needed.

The OS will use software interrupts to trigger a system call, with an exception handler to interpret which system call function needs to be called. On arm CPUs, a software interrupt is capable of changing the Exception Level (EL). In user-mode (once that is implemented) the CPU would be in EL 0, and kernel code will execute in EL 1. The software interrupt will place the CPU into EL 1, entering kernel-mode, and when the system call function returns, the exception handler will run an exception return instruction, placing the CPU back into the EL that it was in when the software interrupt occurred. This is how the system call mechanism will allow the OS to separate user space from kernel space while still allowing user-mode processes to perform operations that require kernel permissions.

Each system call will be assigned a number, and I will copy these from Linux at the start. Most Linux system calls will not ever be implemented, but having some level of equivalence between this system and

Linux system calls will ensure that developer will feel confident that at least some of the knowledge is transferable.

The system call exception handler will use the system call number to determine which function to call. This will be done using a table in memory which maps system call numbers to function pointers. The exception handler will lookup the correct table entry and then call the function at the address it finds.

System call implementations The implementation of the system calls can be separated from the mechanism which enables them to operate. TODO: Expand this

8.1.5 SD card driver

In order to read data from the SD card on the RPi (and write data back to it), the OS will need to implement a driver for the RPi's eMMC chip. The simplest way to do this will be to copy an existing driver for that chip. I will take code from one of the related works [19].

8.1.6 exFAT filesystem

The kernel will include an implementation of the exFAT filesystem. I considered several different filesystems including FAT32, exFAT, and YAFFS [2]. The exFAT filesystem's popularity was the reason I chose it over YAFFS. The FAT32 filesystem seems simpler to implement than exFAT, but I was not able to find a specification for it online, whereas the exFAT official specification is available on Microsoft's documentation website [10]. Having a single source for the filesystem specification gave me more confidence that different tools would work together. Also, the exFAT filesystem is partially supported on Linux systems (FAT32 is not as well supported), and I am developing the OS on Linux.

The file system implementation will need several features in order to work as the main filesystem of this OS. These are listed below, using terminology that will be explained in [subsubsection 8.2.4](#).

- Read the 'superblock' of the SD card
- Interpret the FAT table to follow cluster chains
- Read directory entries
- Search a directory for a file according to its name
- Read the contents of a file
- Write to a file and correctly update its metadata for size and modification time

TODO:

Nice-to-haves

- Virtual filesystem abstraction
- System call implementations for commonly used operations
- Kernel / user mode separation

Future possibilities - move to evaluation section

- Scheduler for multiprocessing (with a process control table / vector in the kernel)
- USB driver for keyboard input (a port of an existing USB driver project)
- Drivers for some other hardware on the RPi (Ethernet / Bluetooth)

8.2 Implementation

TODO: Implementation containing a comprehensive description of the implementation of your software, including the language(s) and platform chosen, problems encountered, any changes made to the design as a result of the implementation, etc.

8.2.1 Bootloader

The second-stage bootloader^{*} is written in arm assembly code. This is because the bootloader performs the necessary setup in order to get the CPU and memory into a state in which code written in C can be run. Once the bootloader has finished this setup, it jumps into the kernel main function (written in C) straight away.

First, the bootloader stops all but one of the CPU cores. The current state of the OS does not support simultaneous multiprocessing with the multicore CPU on the RPi. This first step effectively converts the 4-core CPU into a single-core CPU by putting cores 1, 2, and 3 into infinite ‘do-nothing’ loops, leaving only core 0 to execute code.

Next, the boot code gets the current EL from the CPU. The EL is stored in a special CPU register on arm CPUs, called **CurrentEL**. On the RPi this should always be 2, but I also wrote code to handle the case where it is 3. If the EL is 3 then the boot code sets the needed registers and safely transitions the CPU into EL 2 in the same way as described below.

Changing the Exception Level Once the boot code verifies that it is executing in EL 2, it prepares to change to EL 1, the level that the kernel code will run in. It sets important registers, such as the exception vector table (used by the system call mechanism) and the exception return address (the address of the code to execute once the CPU switches to EL 1). It also ensures that EL 1 is executing in 64-bit mode.

Once all the parameters are set correctly, the bootloader executes an exception return instruction (**eret**). Exceptions are the only way to increase the EL on an arm CPU, and returning from exceptions is the only way to reduce the EL. The exception return instruction changes the CPU into EL 1 and resumes execution at the address of the EL 1 code.

The floating point / vector registers Once I started to implement the function **printf**, which is a *variadic* function[†], I started to experience strange behaviour. I debugged the compiled kernel using remote debugging in GDB, and discovered that upon entering the **printf** function, the executing kernel would jump to an address near 0x0, the beginning of memory. This meant that an exception was occurring and the CPU was going to a position in the *Exception Vector Table*. I determined from the offset into the exception table that the exception was permission related.

Looking at the first instructions in the disassembly of the **printf** function, I saw that the function was accessing registers I did not recognise, labeled **Qn** instead of **Xn**. After some research I determined that the **printf** function was trying to use the 32 floating point registers of the CPU to store the variadic argument list, but the floating point registers were disabled by default.

The solution was to enable the floating point registers after switching to EL 1, by setting some bits in one of the system registers. Setting those bits enabled the floating point registers (also called vector registers) and enabled the use of variadic functions in the C code.

8.2.2 System Calls

The system call interface requires writing an exception vector table. I created an exception handler for the software interrupt instruction I will use to trigger a system call (**svc #0** in Aarch64 ARMv8 assembly). This is incomplete at the moment, as I haven’t yet worked out how I will locate the correct function for a given system call.

I also wrote a basic exception vector table, but I was working from outdated documentation, so I will need to update the table to have the correct padding between the different exception types. I will also need to update the linker script to place the exception vector table at the correct address in the executable, or add code in the boot process to manually set addresses if linking doesn’t correctly set the addresses (the table is in low address space, so it is possible that the bootloader will ignore anything so low in the kernel executable).

8.2.3 CPU setup

When the kernel starts, the CPU will either be in EL 2 or EL 3 (The RPi only supports EL 2 but I have written code to support EL 3 just for completeness). The boot code of the kernel checks the current EL and runs some code to put the CPU into EL 1. The code sets up the system registers correctly to fake the CPU returning from an exception. When the registers are correctly set, the arm instruction **eret** is run, and the CPU “returns” from the “exception handler”, jumping into the kernel code and entering into EL 1. This is done in this way because exceptions are the only way to change EL, and the only way to reduce the EL is to perform an exception return (**eret**).

^{*}The ‘bootloader’ is more accurately described as boot code, since it does not do any loading of the kernel from disk, the traditional task of a bootloader.

[†]A variadic function is one with a variable number of arguments

```

struct exfat_superblock {
    uint64_t partition_offset;
    uint64_t volume_length;
    uint64_t sectorsize; // Bytes per sector
    uint64_t clustersize; // Sectors per cluster
    uint32_t fat_offset;
    uint32_t fat_length;
    uint32_t clusterheap_offset;
    uint32_t cluster_count;
    uint32_t rootdir_start; // First cluster of the root directory
    uint32_t volume_serialnumber;
    uint16_t fs_revision;
    uint8_t fat_cnt; // Number of FAT tables
    uint8_t use_percent;
    // Volume flags
    uint8_t active_fat; // Which fat is active
    bool volume_dirty;
    bool media_failure;
    bool clear_to_zero;
};
int exfat_read_boot_block(struct block_device *, struct exfat_superblock *);

```

Listing 1: The structure used to represent the exFAT superblock information, and the declaration of the function used to read it

```

static uint32_t fat_idx_to_cluster_idx(struct block_device *dev, struct
    exfat_superblock sb, uint32_t fat_idx);

```

Listing 2: The function which is used to get the cluster index of the next cluster in a cluster chain

8.2.4 exFAT Filesystem (FS)

I took an existing driver for interfacing with an SD card (via the RPi’s eMMC controller) and integrated it into the kernel. This code defines a structure called a **struct block_device**. I also copied some FS code from the same related work [19], but most of the code for the FS I wrote by myself.

I first created a function to read the ‘superblock’ of an exFAT system. The superblock is the first block of the disk partition, and contains some critical information for interpreting the information on the rest of the disk. The structure used to store this information and the declaration for the function are both shown in Listing 1. The function is called **exfat_read_boot_block()** because sometimes the superblock is referred to as the ‘boot block’ (although usually only when talking about disks instead of FSes).

There is also a function which takes a cluster index and returns the index of the next cluster in that cluster’s cluster chain. This information is retrieved by reading the File Allocation Table (FAT) section on the volume (see section 4.1 in [10]). The function signature is shown in Listing 2.

The next structure on an exFAT volume that needs to be interpreted is the directory. Directories primarily contain information about files, but can also contain some special entries. These are all stored in ‘directory entries’ (see sections 6 and 7 in [10]). The information for a directory is stored in a **struct exfat_directory_info**, and information describing a single directory entry is stored in a **struct exfat_dirent_info**. These are both defined in Listing 3.

A function in the exFAT driver, **exfat_read_directory_entry()**, is used to read and interpret a single directory entry. Its single argument is a pointer to a **struct exfat_dirent_info**, which is already filled with the information that is needed to identify a single directory entry in a directory. The function fills the rest of the fields.

Another function, **exfat_readdir_fromblock()**, takes a block device (the SD card) and the address of a block which is the beginning of a directory, and reads the entire directory. It stores each file from the directory in a **struct dirent**, partially defined in Listing 4. The **dirent** structure contains a **next** pointer, so the function joins all of the files into a linked list data structure, and returns the first file.

A distinction is made between a directory entry and a file. A directory entry may not be associated with a file, and each file will have at least three directory entries associated with it. The **exfat_dirent_info** structure refers to a single directory entry, whereas the **dirent** structure describes a single file. The **exfat_readdir_fromblock()** function reads directory entries in a loop, ignoring those not associated with


```

struct exfat_directory_info {
    struct block_device *bd;
    struct exfat_superblock super;
    uint32_t start_cluster; // First cluster of the directory
};

struct exfat_dirent_info {
    struct exfat_directory_info *parent;
    uint32_t direntset_idx; // Index into the "Directory"
    // The importance, category, and type code represented in the same way as
    // on disk.
    enum EXFAT_DIRENT_TYPE type;
    bool in_use; // The in_use bit of the entrytype field can change
    union {
        struct exfat_dirent_bitmap_chunk bitmap;
        struct exfat_dirent_upcase_chunk upcase_table;
        struct exfat_dirent_vollab_chunk volume_label;
        struct exfat_dirent_file_chunk file_metadata;
        struct exfat_dirent_volguid_chunk volume_guid;
        struct exfat_dirent_strext_chunk stream_extension;
        struct exfat_dirent_fname_chunk filename;
        struct exfat_dirent_vendor_guid_chunk vendor_guid;
        struct exfat_dirent_vendor_alloc_chunk vendor_alloc;
    } entry_data;
    // If the dirent doesn't point to another cluster chain, then data_length
    // will be 0.
    uint32_t first_cluster;
    uint64_t data_length;
};

```

Listing 3: The structures which describe directories and directory entries on an exFAT volume

```

// Struct representing a file
struct dirent {
    struct dirent *next; // The next file in the directory
    char *name; // Filename
    uint32_t byte_size; // Size of the file
    uint8_t is_dir; // Is this file a directory?
    ...
};

```

Listing 4: The dirent struct, which stores information about files

files. It reads all of the entries for each file, then creates the `struct dirent` and appends it to the linked list. When it reaches the ‘end of directory’ directory entry the function returns.

8.2.5 Graphics driver

For graphics, I created a simple library for manipulation of the framebuffer. In order to print characters to the screen, I converted an 8x16 bitmap font [5] into a constant array definition in a C source file. I then wrote a function to print a character from this array onto the framebuffer, and I added global variables which keep track of the current character position on the screen.

TODO: write about changes here (scrolling at the bottom of the screen - uses the swap rectangle function)

When the console reaches the end of a line, it moves to the start of the next one. At the moment, when it reaches the end of the last line, it moves back to the start of the first line. In the future I will implement scrolling, so that when the console reaches the bottom of the screen, every line will be moved up to make space for the next line.

8.3 Evaluation

TODO: Evaluation and External Aspects explaining how your software/approach was tested (using different datasets or in different environments), statistical evaluation of performance, results of user evaluation questionnaires, etc. You should explicitly address how your project fulfilled (or not) its original intentions with regard to its ‘external aspect’.

Overall, the kernel met the requirements that I set out in subsection 8.1. The bootloader is robust and reliable, starting the kernel main function every time the OS boots.

Good:

- Bootloader is robust, correctly sets EL for kernel
 - TODO
- System call mechanism works. Could run a system call from EL 0 and (if the function exists) the syscall would run in EL 1
 - TODO
- TODO

Bad:

- No syscalls implemented yet
- Test coverage is very limited
- Most functions in `string.h` are missing implementations (TODO: not really a kernel issue?)
- No kernel / user space separation: all code runs in EL 1 (everything is Kernel code)
 - Would require syscall implementations before switching to EL 0 for user mode processing
- Virtual filesystem abstraction is missing: all file code directly calls exFAT-specific functions
 - Should use the `struct fs` member function pointers (`fopen()`, `fread()`, `fwrite()`, `fclose()`)
 - This would be a simple fix - create a `struct fs` from the exFAT filesystem
 - Make functions like `exfat_fopen()`, `exfat_fread()`, etc. in the `struct fs`

Future:

- Implement syscall functions
 - `getcwd`, `getpid`
 - `fstat`, `getdents`
 - `open`, `close`, `read`, `write`
 - `mkdir`, `chdir`, `rmdir`
 - `execve`
- Multi-processing and a simple scheduler
- TODO

9 The Shell

TODO: reword this into an introduction to the shell

- Design of the shell is important
- If it is done well people will immediately feel comfortable using the Operating System (OS)
- A simple but intuitive shell will give a good first impression to the project's code

FIXME: The shell which I will include in the OS will be a port of a POSIX compliant, simple shell, such as Dash [9]. This shell will be the main way that users interact with the system, as it will allow them to execute programs which do anything they need, such as compiling other programs, editing text files, or viewing the contents of files. The shell will run in user space, and any time it needs to run kernel-level functions, it will do so through the system call interface. This is the same way the Bourne Shell ran on Unix, and it is a known good way of allowing the user to control a computer. Anyone who has experience with a shell will immediately feel at home when they see the input prompt and a flashing cursor.

9.1 Design

TODO: write a specification for each component

9.1.1 Features I expect to complete

Commands as functions In the initial implementation, I will develop the shell as a part of the kernel, and all of the commands will be functions built into the final compiled kernel. I will do it this way because it is simple, and my OS will not have a binary file loader at this stage, which would be needed for loading external versions of commands. The downside of using this method is that any new commands or changes to commands would require a recompile and rebuild of the entire kernel, but so far building the kernel from scratch is a relatively quick process.

Commands in the shell will be functions in the shell source code files. The functions for commands will all share the same function signature, which will be the same as the function signature for a `main` function [13]. This will mean minimal changes would be needed to eventually turn each command into its own executable, with the binaries being stored on the filesystem (in a `/bin` directory). This change will be the natural next step from what I create, so I am designing my shell implementation with that in mind from the start.

The shell loop The logic of an interactive shell is quite simple. It consists of an infinite loop:

- Read a line from the keyboard
- Split the line into words by spaces
- Determine which command, if any, the first word matches
- Execute the correct command
- Repeat this process forever

The shell needs to get user input on a line-by-line basis, since each command starts at the beginning of the line and ends at the newline character. In the first iteration, I will store the text input in a fixed-size buffer, meaning that there will be a maximum size for an input line. I will set this maximum size to a reasonable value such that it will not cause any problems, and I will leave this problem to fixed in a future update to the shell's user input functions.

I will split the line by spaces and place the words into a 'vector', passed to the commands as the argument `argv`.

The commands I will create several commands, starting with the simplest and most essential ones. The OS will need a way to shutdown the Raspberry Pi (RPi), so I will implement a `shutdown` command. I will write a `pwd` command to print the current working directory, as this will be a very simple (and also useful) operation. When I write the filesystem, I will create a function for printing the contents of directories (`ls`), and also a recursive version of this, `tree`, which shows an entire directory structure in context.

Once I have implemented enough of the filesystem to get file contents from the SD card, I will create a `cat` command to print files to the screen. Finally, a command for getting file attributes, like `stat` on Linux systems. On Linux systems, `stat` is implemented via a system call (`fstatat`), so this command could be rewritten using that if a future development creates that system call.

9.1.2 Nice-to-haves

Executable file loader One feature that would significantly improve the shell would be the inclusion of a binary file loader. This could either support only raw binary files, or statically compiled Executable and Linkable Format (ELF) files. A file loader would not be possible without first implementing file reading in the filesystem, so this would be a very late feature to start implementing if I put it into the OS.

For loading ELF files, I would implement a basic ELF parser to extract the instructions from the file and put them into memory. The loader would then determine where the entry point is, setup the stack and Central Processing Unit (CPU) registers, then finally jump to the entry point.

The process would be the same for loading raw binary files, except that the entry point would be fixed. There would also be no need to extract the instructions from the ELF structure, just loading the instructions from the filesystem and then jumping into the loaded file contents.

Rewrite commands using system calls Many of the commands I will implement would benefit from using some standard system calls, but I will not have implemented any system calls by the time I create the commands. Rewriting these commands to use the system calls would be a nice-to-have feature. It would also give an incentive to implement the common system calls, as I would be developing system calls I need at the time.

Some commands which would benefit from this would be `ls` (system call `getdents`), `stat` (`fstatat`), and `cat` (which would benefit from the system calls `open` and `read`). The `printf` function could also be rewritten to use the system call `write`, writing to a file descriptor which represents the graphical console.

Replace the shell and all commands with separate executables As a follow up to including a binary file loader in the shell, I could also completely separate the shell and kernel. This would allow me to modify the shell, which will be growing in size, without recompiling the kernel, which will also be growing in size. The individual compile-times of both the shell and the kernel would be effectively cut in half, allowing for much more rapid iteration and development of both.

With the shell separated from the kernel, the kernel would load the shell from the filesystem on boot (using the binary file loader). The commands could also be moved to their own executables, and placed into some common directory, likely `/bin/`. The shell would search this directory for filenames when it is trying to find executables that match input lines. This hardcoded directory would eventually be replaced by an environment variable `PATH`.

Doing this would allow modification of the commands available on the system without recompiling the kernel, restarting the OS, or even restarting the shell.

9.1.3 Ideas for next steps

TODO: Move this to evaluation - rename to just 'Possibilities'?

TODO:

- Support for arrow-key navigation (and other special non-character keys) in the console input line
 - History of previous lines entered into the console with up-arrow to get the previous line
 - Cursor position on the line (cursor is between two characters)
 - Backspace to delete one character back, delete to delete one character forwards
- Line-based text editor (like `ed`)
- Text pager, like `less` or `more`
- Shell scripting language (try to be POSIX compliant) interpreter - or port dash [9]

9.2 Implementation

TODO

9.2.1 Getting input from the keyboard

The shell needs to get input from the keyboard line-by-line, so the terminal can buffer the current line as it is being typed. A buffer of a fixed size is created on the stack when the shell is started. The terminal uses a `getc()` function to get one character from the serial port. When `getc()` returns a newline character, the shell's input line buffer gets tokenised, splitting it by spaces. The function `strtok()` (from `string.h`) is used for this.

The addresses of the strings containing the tokenised words from the line are placed into an array, which is the member 'argv' of a **struct command_args**. The total number of words in the line is placed into that struct's member 'argc'. The member variable 'envp' is set to a pointer to the global **struct process_env** of the kernel. This struct contains all the miscellaneous information that commands may need, such as the current working directory and the return value of the previous command. Once the OS implements the concept of user and kernel space, only a subset of this struct would be passed to commands, and then the commands would need to use system calls (see [subsection 8.2.2](#)) to access that information.

TODO: add a diagram explaining the contents of the **struct process_env**.

9.2.2 Storage of enabled commands

The shell's memory contains a linked list which contains all of the enabled commands. The commands are described by a structure called a **struct shell_command**, with the structure definition shown in [Listing 5](#). The member **cmd_str** is a string containing the word that is used to invoke that command, and **fun** is a function pointer to the function that will be used whenever the first word of an input line is equal to **cmd_str**.

```
struct shell_command {
    int (*fun)(struct command_args);
    const char *cmd_str;
};
```

Listing 5: The definition of the **shell_command** struct

9.2.3 Shell commands as functions

In the current implementation, all of the commands that are available to the shell are functions in the source code of the kernel. The function signature of the command functions is shown in [Listing 6](#), as **command_function()**. The function **add_command()**, also declared in [Listing 6](#), takes a string and a function pointer and creates a **struct shell_command** from them, adding it to the shell's **struct shell_cmdlist** (the linked list). The function **shell_init()** calls **add_command()** once for every shell command, initialising the shell to prepare it for running commands.

```
int command_function(struct command_args);
int add_command(const char *cmd_str, int (*fun)(struct command_args));
int shell_init();
```

Listing 6: The signatures of some important functions for the shell

9.3 Evaluation

TODO

Good:

- Modular system for creating new shell commands: only need to define a function **int fun(struct command_args*)** and decide a string for invoking the command
- The structure passed to the shell functions contains the three arguments that would be passed to the **main()** function of a binary file (**int argv**, **char **argv**, **char *envp[]**). The environment argument is not exactly the same and is actually a pointer to a struct containing all the important information about the kernel process, such as the current working directory and the (currently unused) process ID (PID). The vector of environment variables could be extracted from this struct with some easy pre-processing.

Bad:

- No text editor: line-based editor (based on **ed**)
- TODO

Future:

- Create a binary file loader (Maybe ELF)
- Make the shell a separate binary
- Make each shell command a separate binary
- Implement some shell language features (make the shell an interpreter)

10 Evaluation and External Aspects

TODO: Evaluation and External Aspects explaining how your software/approach was tested (using different datasets or in different environments), statistical evaluation of performance, results of user evaluation questionnaires, etc. You should explicitly address how your project fulfilled (or not) its original intentions with regard to its 'external aspect'.

10.1 Overall evaluation of the results

TODO:

- Inconsistent code style
 - Started writing functions with return type `ERROR_TYPE`, an alias of `size_t`. Each possible error was assigned a value, and a return value of zero meant no error
 - Once `errno.h` was created, I switched to using `errno` to represent errors
 - Later functions return `int`, with zero meaning success and a non-zero value meaning failure
 - On non-zero, the calling function should check `errno` to interpret what the error was
- Test coverage was very limited
 - I effectively gave up on writing tests after writing tests for the `malloc()` function and for the eMMC driver
- Most of the functions in `string.h` do not have definitions
 - This is not a bad thing (although maybe they shouldn't be in the header file if they are not implemented)
 - Functions are implemented as they are needed - most of the functions are very simple and take less than half an hour to write
 - Eventually I will reach a point where only a couple of the functions in that header file are not implemented, and I will either remove them or implement them

10.2 Evaluation in relation to the external aspect

TODO:

- The Operating System (OS) provides a familiar interface to anyone who has used a Unix-like system in the past
- The code is somewhat easy to understand
 - Some source code is located in places that might not make much sense - The shell should probably have its own top-level directory `src/shell/` instead of `src/kernel/shell/`
 - I commented most of the complex logic and tried to explain it
- There are plenty of possibilities for immediate improvement to the project
 - This provides options for developers who discover the project to start working on an OS development project at whatever level they feel comfortable with
 - They could provide implementations of the functions in `string.h`
 - They could write timer functions - the `usleep()` and similar functions are currently empty stub functions, and accurate timers would be very useful for future development

11 Summary and Reflections

TODO: Summary and Reflections including a discussion of results in a wider context (considering other work).

11.1 Project management

TODO: Project management covering the tasks as a part of your work plan and progress as well as how time and resources are managed. For a COMP4028 project, role of each individual should be explicitly discussed.

11.1.1 Project management – Re-planning the project

The original plan for the project was a little over-ambitious, which I discovered as I was developing the current implementation. I have modified the plan to account for the more restrictive time constraints, and also considering my experience developing the Operating System (OS) so far. Some of the later, more complex tasks have been removed from the plan, as I no longer believe I would be able to complete them, and some tasks have been given more time. [Table 1](#) shows the new plan for the project’s tasks, and [Figure 1](#) is the Gantt chart which describes the dates on which I expect to start and finish each of the tasks.

11.1.2 Unexpected difficulties in development

I have encountered several issues during development which I did not account for in the original project plan. The first is not so much an issue as it is a task not being required in the way that I expected. The **debug shell** was not needed in the way I had planned to implement it. My plan was to prompt the user for a command, and for the shell to support a very limited range of commands, perhaps starting with a command to print the contents of a memory address. This type of debugging has not been needed so far during the development process, and I don’t believe it will be needed, as I can use The GNU Project Debugger (GDB) for debugging memory locations, and GDB doesn’t even require the OS kernel to be in a working state, which would be the most likely scenario for if I would need to read memory.

The “debug” part of the debug shell task was still completed, as the current version of the kernel prints debugging information to the serial port (and also to the Raspberry Pi (RPi) video output once the framebuffer is allocated). This debugging information is changing as I add more features to the OS, and lines which tested code that will not change again (i.e. the `printf` function) are removed.

System calls were far more difficult to implement than I expected. The way that system calls will be implemented is using arm exceptions to execute a function with the processor in a privileged mode of operation. Unfortunately, when I started working on the system call implementation, I did not have a very good understanding of the Central Processing Unit (CPU) mechanisms by which arm processors handle exceptions. The CPU will use a large number of system registers, along with the current Exception Level (EL), to decide where the exception handler can be found, and in which EL the CPU should handle the exception. I did not know any of this at the time that I started to implement the system call mechanism, and I was using documentation for a very old version of the arm architecture. I did not even know which EL the CPU was executing the kernel in, so I worked on code for the **filesystem** for a while.

When I returned to working on system calls, I knew I needed to initialise the CPU state, so I followed a tutorial [\[16\]](#) and used the arm developer documentation [\[3\]](#) to learn enough about arm ELs and exceptions. I wrote code to initialise the CPU into EL 1, which I am going to use for the kernel of my OS. I also set up the processor registers to correctly handle exceptions, so that any system calls from either user space (EL 0) or kernel space (EL 1) will both be handled by the kernel in EL 1.

When I started working on the **filesystem**, I immediately realised something that I forgot to consider for the original project plan. The filesystem would require an SD card driver in order to interact with the RPi storage. I used an existing implementation of a simple RPi SD card driver [\[19\]](#), which only required a small amount of modification to get working. Once I had it integrated into my kernel, I created an SD card image and added it to the emulator, then tested the driver to see if it recognised the SD card. To my surprise, the driver knew when the SD card image was present, and the initialisation function returned the expected value when the image was present on the emulated RPi.

My initial expectation of the difficulty of memory allocation was very inaccurate. It is a much more complicated process than I had assumed it was, and managing the memory that is allocated requires more kernel infrastructure than I expected. As a result of this, memory management was brought forwards in the list of tasks. However, I will need to finish implementing system calls without a `malloc()` function. Programming without `malloc()` is difficult and confusing, as functions must take pointers to data structures as arguments. Results must be placed on the stack, as there is no heap. The easiest way to implement `malloc()` is to use a system call `mmap`, but my OS does not yet have a system call interface, so if it turns out I need memory on the heap, I cannot implement `malloc()` that way.

11.2 Contributions and reflections

TODO: Contributions and reflections providing the details of your achievements and contributions including innovation, creativity and novelty (if there is any) as well as a personal reflection on the plan and your experience of the project (a critical appraisal of how the project went).

The project's external sponsor (who is also my project supervisor), introduced me to several of the related works [14, 18]. He also gave recommendations for some of the areas I was originally unsure of or less knowledgeable about. He suggested that I implement the FAT Filesystem (FS) as it is simple and relatively easy to implement, and he reminded me that on arm based systems, system calls would utilise software interrupts.

I think that the original project plan was too ambitious. I failed to consider that I would have other modules and also extra-curricular obligations, and I underestimated how much of my time would be taken up doing things unrelated to the project. Overall, the project has not progressed as quickly as I had originally expected. This is why I have scaled back the plan for the second half of the year, replacing the shell and libc with ports of already existing versions.

11.3 Foo

My aim for this project was to produce a Unix-like OS for the RPi 3 which would be capable of being used as a general purpose hobbyist OS. The result of the project was not what I originally had hoped to achieve, but the development of the OS provided more education than I expected about the many components required for an OS to function. I believe that this educational experience will prove valuable to others, and so I consider this project to have been a success.

Acronyms

arm Arm Microprocessors Ltd.

CPU Central Processing Unit

EL Exception Level

ELF Executable and Linkable Format

FAT File Allocation Table

FS Filesystem

GDB The GNU Project Debugger

GPU Graphics Processing Unit

IO Input / Output

OS Operating System

RPi Raspberry Pi

SBC Single Board Computer

SOC System-On-a-Chip

UART Universal Asynchronous Receiver-Transmitter

References

- [1] *A generic and open source machine emulator and virtualizer*. URL: <https://www.qemu.org> (visited on 2021-12-12).
- [2] *A Robust Flash File System Since 2002*. URL: <https://yaffs.net/> (visited on 2022-04-21).
- [3] *ARM Cortex-A Series Programmer's Guide for ARMv8-A: System Registers*. Version 1.0. URL: <https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/System-registers> (visited on 2021-12-01).
- [4] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., 1986. ISBN: 978-0132017992.
- [5] *Bizcat - an 8x16 bitmap font*. URL: <https://robey.lag.net/2020/02/09/bizcat-bitmap-font.html> (visited on 2021-12-12).
- [6] *Clangd: teach your editor C++*. URL: <https://clangd.llvm.org/> (visited on 2021-12-12).
- [7] *ClangFormat*. URL: <https://clang.llvm.org/docs/ClangFormat.html> (visited on 2021-12-12).
- [8] *CoC.nvim: the Conquer of Completion*. URL: <https://github.com/neoclide/coc.nvim> (visited on 2021-12-11).
- [9] *Debian Almquist Shell (Dash)*. URL: <https://git.kernel.org/pub/scm/utils/dash/dash.git/> (visited on 2021-10-26).
- [10] *exFAT file system specification*. URL: <https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification> (visited on 2022-04-21).
- [11] *GDB, the GNU Project debugger*. URL: <https://www.sourceware.org/gdb/> (visited on 2021-12-12).
- [12] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Inc., 1984. ISBN: 978-0139376818.
- [13] *Main function - cppreference.com*. URL: https://en.cppreference.com/mwiki/index.php?title=c/language/main_function&oldid=139269 (visited on 2022-04-20).

- [14] *NetBSD*. URL: <https://github.com/NetBSD/src> (visited on 2021-11-28).
- [15] *Primary and Secondary UART*. URL: <https://www.raspberrypi.com/documentation/computers/configuration.html#primary-and-secondary-uart> (visited on 2022-04-19).
- [16] *raspberrypi-os: Learning Operating System (OS) development using Linux kernel and Raspberry Pi (RPi)*. URL: <https://github.com/s-matyukevich/raspberrypi-os> (visited on 2021-11-28).
- [17] *Recursive use of make*. URL: https://www.gnu.org/software/make/manual/html_node/Recursion.html (visited on 2021-12-05).
- [18] *RiscOS*. Version 5.29. URL: <https://www.riscosopen.org/content/> (visited on 2021-11-28).
- [19] *rpi-boot: A second stage bootloader for the RPi*. URL: <https://github.com/jncronin/rpi-boot> (visited on 2021-11-28).
- [20] *The OSDev Wiki*. URL: wiki.osdev.org (visited on 2021-10-20).
- [21] *The source code (in L^AT_EX) of the document you are reading*. URL: <https://github.com/Ytrewq13/YtrewqOS/tree/master/Documentation/dissertation/interim-report> (visited on 2021-12-05).
- [22] *Unix-like*. URL: <https://en.wikipedia.org/w/index.php?title=Unix-like&oldid=1074888381> (visited on 2022-04-15).
- [23] *Vim - the ubiquitous text editor*. URL: <https://www.vim.org> (visited on 2021-12-11).
- [24] *VT100 escape codes*. URL: <https://espterm.github.io/docs/VT100%20escape%20codes.html> (visited on 2022-04-19).
- [25] Brian W.Kernighan and Dennis M.Ritchie. *The C Programming Language*. Second. 1988. ISBN: 978-0-13-110362-7.

Appendices

A Interim: Appendices

A The original project plan

This appendix contains all of the tables, figures, and diagrams from the original project proposal, laying out my initial beliefs about what the project entailed, and how much work each part of the project would take.

The diagrams are copied directly from the project proposal, so tenses may now be incorrect (e.g. “I think” instead of the now more correct “I thought”), but this is done intentionally as this section is included only as a reference to the content of the previous document.

B Work tasks

Tasks are the individual components of the project, separated into the components of the Kernel and Shell, the miscellaneous parts, and the documents I will need to produce for the project. These tasks are enumerated in [Table 3](#) and then assembled into a Gantt chart in [Figure 3](#).

Task	Duration (weeks)
Kernel	
Bootloader	1
Graphics driver	2
Syscalls	6
Filesystem	3
Statically linked ELF Loader	2
Memory management	4
Process control	8
Init system and service manager	5
Threads and multithreading	4
Shell	
Debug shell	2
<code>pwd</code> , <code>cd</code> , <code>ls</code> , <code>stat</code> , etc.	2
<code>export</code> , variables, <code>set</code> , etc.	2
<code>if</code> , <code>while</code> , <code>for</code> , <code>case</code> , globs, etc.	3
Command substitution	4
IO pipes and output redirection	1
Misc parts	
Setup cross-compiler and build environment	1
IO and strings libraries	2
<code>libc</code>	8
<code>cat</code> , <code>head</code> , <code>less</code> , etc.	2
<code>roff</code>	1
<code>man</code>	1
Documents	
Interim report	1
Documentation	5
Dissertation	7

Table 3: The original plan of work for the project, including how many weeks I thought each component would take.

C Gantt chart

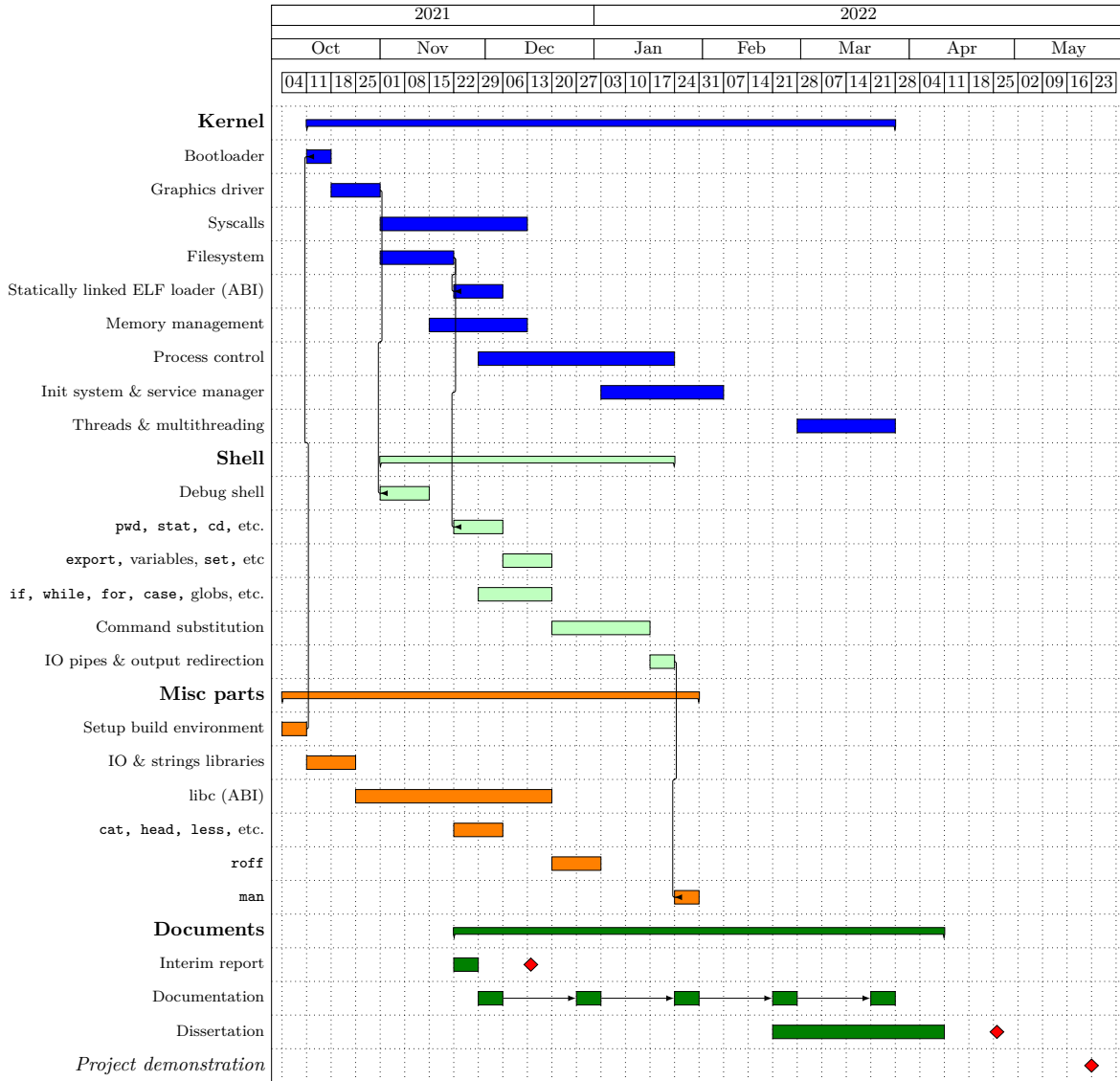


Figure 3: A Gantt chart breaking down the proposed schedule of work for this project.

D Work Packages

A “Work Package” is a collection of tasks with a well-defined deliverable. Not every task will be part of a work package, and some tasks will be work packages on their own. Table 4 contains all of the work packages for this project.

Work package	Description of the deliverable
Debug shell using the graphics driver	A debug shell which is printed on the Raspberry Pi (RPi)'s display and which uses keyboard input. The debug shell should accept several commands for debugging the processor status (including, but not limited to, printing the contents of memory, writing to memory, and jumping to a given memory address).
A basic kernel which can load (from disk) and jump into a compiled Executable and Linkable Format (ELF) program.	A program that can load a statically compiled ELF executable binary into memory and then jump to its entry point. This requires a filesystem library to be implemented, and also includes the ELF loader task.
Scheduling algorithm	A scheduling algorithm with support for multiple processes running on the system concurrently. This will include an implementation of <code>fork(3)</code> or a similar function in order to spawn new processes and <code>execve(2)</code> to replace the current program with a new process.
Init system and service manager	An init system which is run at boot and starts all necessary parts of the Operating System (OS) and then spawns the root shell. The service manager ensures that all desired daemons and processes are started at the correct times and remain running, restarting them if they die.
A shell	A (POSIX-compliant) shell or a port of a simple shell like Dash[0]. If I decide that implementing my own POSIX-compliant shell is too large of a task, then I will port the source code for Dash to run on my OS, implementing the required syscalls to get it working. This shell will be the main way users interact with the OS.
Documentation	This package contains several parts. I will create documentation for how each part of the OS works, and a guide for how new programmers can get started developing programs for the system. The documentation will be completed in stages as I develop the OS, and I will go back to keep it up-to-date as I make changes to components and add new components.
Multithreading support and multi-core scheduling	The RPi 3 has 4 Central Processing Unit (CPU) cores. This work package will enable users to take advantage of the additional processing power of the other 3 cores for their programs. This includes a threading library similar to <code>pthread</code> on Linux systems, and improvements to the scheduler so that it can give different CPU cores to multiple threads owned by the same process. This work package is optional.

Table 4: The work packages of the project and their deliverables.

E Future plans block diagrams

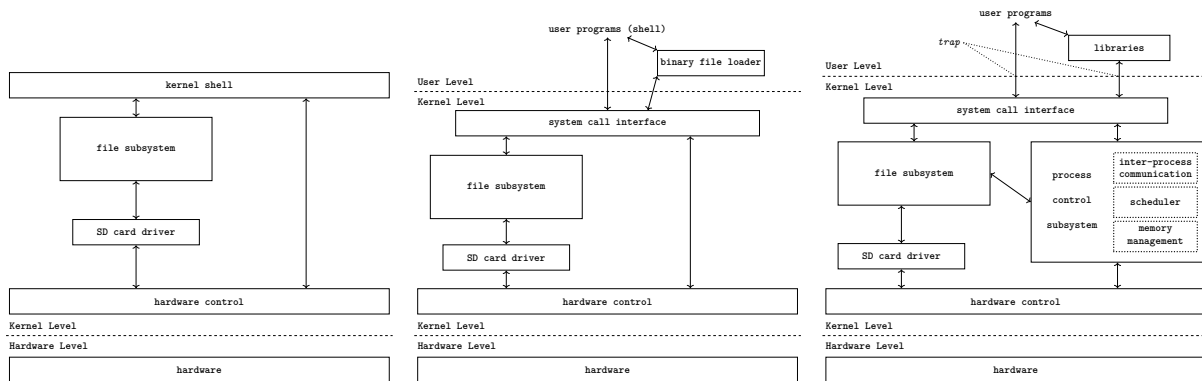


Figure 4: My current estimation for how future development would work on this project. On the left is the current state of the Operating System. On the right is the diagram of how the Unix system was structured.

Acronyms

arm Arm Microprocessors Ltd.

CPU Central Processing Unit

EL Exception Level

ELF Executable and Linkable Format

FAT File Allocation Table

FS Filesystem

GDB The GNU Project Debugger

GPU Graphics Processing Unit

IO Input / Output

OS Operating System

RPi Raspberry Pi

SBC Single Board Computer

SOC System-On-a-Chip

UART Universal Asynchronous Receiver-Transmitter