Operating System for the Raspberry Pi

A Kernel and simple Shell for the 64-bit Raspberry Pi 3b, developed from an historical perspective on early Operating System structures.

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

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

Contents

1 Introduction

2 Motivation

3 Related work

4 Description of the work

5 Methodology

6 Design

7 Implementation

8 Evaluation and External Aspects

9 Summary and Reflections

10 Getting Started

10.1 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 runs on “bare metal”, so it cannot use any of the standard header 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.

10.1.1 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”).

10.1.2 The build process - Make and Makefile

10.2 How I will debug the OS

10.2.1 GDB as a remote debugger

10.3 Developing on hardware, or why I won’t be

11 The Kernel

11.1 The early boot process

11.2 Memory allocation

11.3 The filesystem

11.4 System calls

11.5 Process handling and environments

12 The Shell

12.1 Design of the built-in kernel shell

13 Interim: Introduction

I am aiming to produce a Unix-like Operating System (OS) for the Raspberry Pi (RPi) 3 which is capable of being used as a general purpose hobbyist OS.

The OS will provide a kernel, with a system call interface, process scheduler, and Filesystem (FS) abstraction layer, as well as a shell which it will boot into to run programs for the user [*].

14 Interim: Motivation and Background

14.1 Motivation for this project

My 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.

14.2 Background: the Raspberry Pi and Operating Systems

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 Central Processing Unit (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 will require drivers to function, so there is plenty of room for additional features if the project moves 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 develop will 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, and I will create documentation of how each component of the final system works.

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" [†]. 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.

15 Interim: Related works

15.1 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.

15.2 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 19.5).

15.3 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.

[*] The details of the shell are not given here as I do not know them yet. The shell will probably be a port of an already-existing POSIX compliant shell.

[†] Wikipedia, 'Unix-like': `en.wikipedia.org/wiki/Unix-like`

15.4 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 Interim: Related works

[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] *`raspberry-pi-os`: Learning OS development using Linux kernel and RPi.* URL: `https://github.com/s-matyukevich/raspberry-pi-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).

16 Interim: Description of the work

I have split the description of work into two main parts, the Tasks of the Operating System (OS), and a selection of Work Packages I will work on.

16.1 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 10 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).

16.2 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.

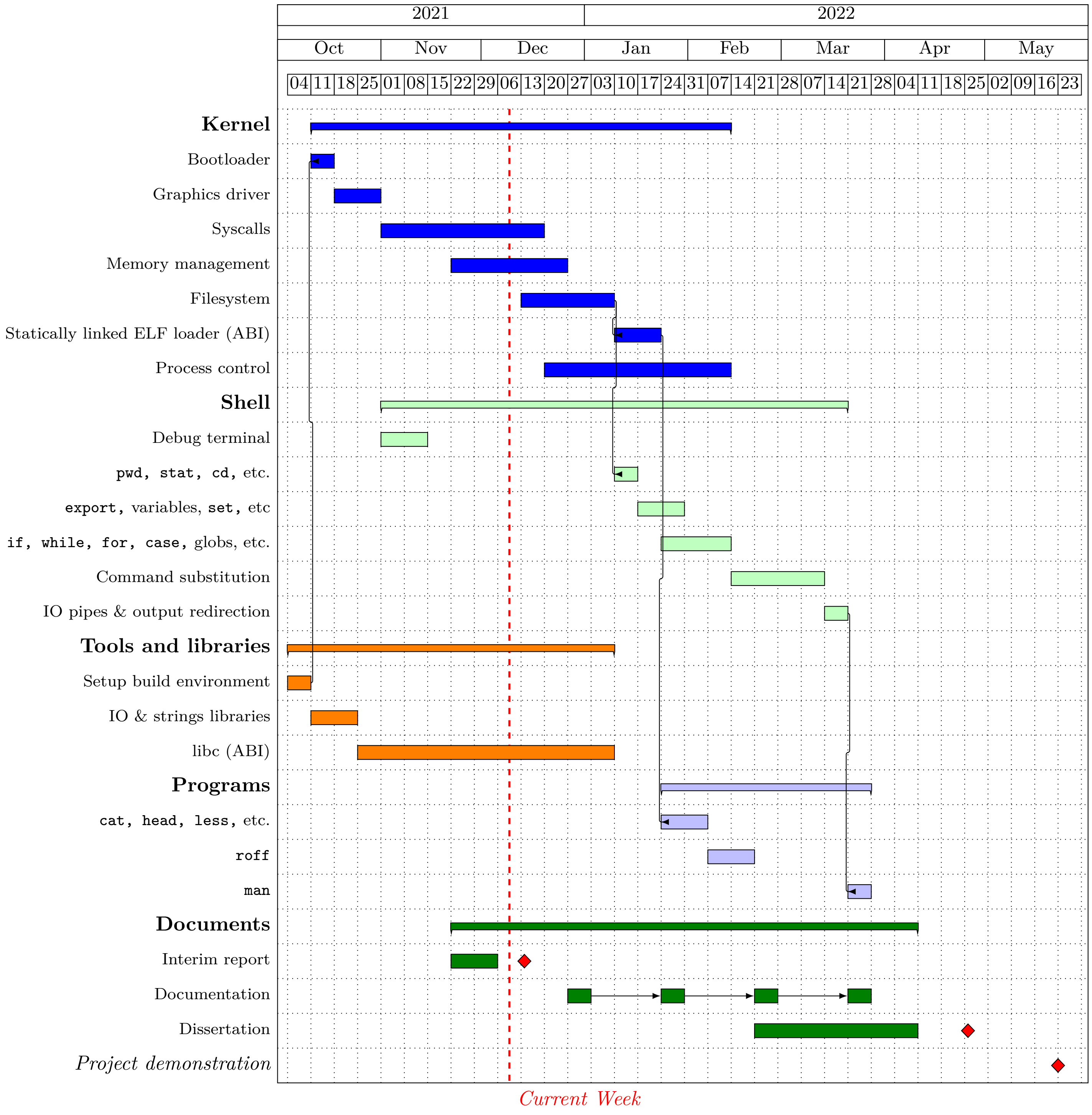

Figure 1: A Gantt chart breaking down the proposed schedule of work for this project.

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
`pwd`, `cd`, `ls`, `stat`, etc.	1
`export`, variables, `set`, etc.	2
`if`, `while`, `for`, `case`, 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	
`cat`, `head`, `less`, etc.	2
`roff`	2
`man`	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.

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 `fork(3)` or a similar function in order to spawn new processes and `execve(2)` 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[8]. 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 `pthread` 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.

17 Interim: Methodology

17.1 Software Development methodology

The methodology I am using for the development of this project borrows attributes from both Cowboy programming and Agile software development.

17.1.1 Cowboy programming

Cowboy programming is a methodology categorised by being very free-form and not having a very well-defined structure. It is often used by novice programmers when they create solo projects, and it is the default when someone doesn't think they are using any kind of methodology.

In cowboy programming, the developer has a lot of control over the project, and they can decide the order features are implemented in, and even add or remove features at will. The lack of planning causes development to follow the developer's natural preferences, often leading to unfinished projects as the developer gets to a stage where the only features left to implement are ones the developer doesn't want to work on.

Cowboy programming is very well-suited to a solo development project, but it comes with a lot of downsides and pitfalls. The best way to avoid these development traps is to not completely rely on this methodology, but to also integrate some aspects of Agile development.

17.1.2 Agile software development

Agile software development is a group of methodologies which all focus on several core ideas. These are: frequent reviews of the product, lots of planning and re-planning, a focus on good communication, and a very well-structured development process. Some of the more popular Agile methodologies include KanBan, SCRUM, Test Driven Development, and Extreme Programming.

Agile development works best in situations with small development teams, and some Agile methodologies involve pair programming, which requires two programmers to work on the same code at the same time. I cannot use pair programming or host team meetings, this being an individual project, but I can still take inspiration from the techniques employed by different Agile methodologies.

I will take inspiration from Extreme Programming (XP) [9] and Test Driven Development (TDD) [12]. I will work from a list of features which are organised by priority, which is an Extreme Programming method. From TDD, I will be writing developer tests and following TDD methods once the Operating System (OS) is mature enough to run tests in the kernel.

17.1.3 How I have been approaching the development until now

The way that I have been developing the OS so far has been closer to cowboy programming than to any agile software development methodologies. So far, I have not started writing any developer unit tests for TDD, as the kernel has only just reached a stage where I could run tests on the emulator.

I have been loosely following the plan laid out in the Gantt chart from the project proposal (Figure 3 in the Appendix), but I have not been using any other elements of XP. The order in which to develop the tasks has changed as I have been developing them, so I have been exercising the freedom and control over the project that comes with cowboy programming.

17.1.4 How I intend to develop moving forwards

Moving forwards in the development of the OS, I want to more closely follow the intentions of agile development. I will try to use TDD wherever I can, writing developer tests at the start of each task in order to have a clearly defined set of behaviour requirements for every feature.

17.2 Software tools used to aid development

17.2.1 Source code editing

For editing the source code of the OS, I am using the text editor Vim [21] with several plugins to assist me in writing correct code. I am using the CoC extension [7] for autocompletion, and the clangd language server [5] extension for CoC in order to provide source code linting for C source files.

For source code formatting, I run the program `clang-format` [6] 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.

17.2.2 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, ususally 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 kerrnel (`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`.

17.2.3 Debugging

GDB [10] 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.

17.2.4 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.

18 Interim: Design

18.1 Operating System (OS) components

18.1.1 Features I expect to complete

- Microkernel with UART serial output and graphics driver for the Raspberry Pi (RPi)'s HDMI video output. User interaction will use the display and read from the keyboard.
- Shell (hopefully POSIX-compliant) with scripting capabilities. Depending on time constraints, this may be a port of Dash [8].
- Multi-process system allowing multiple processes to time-share the CPU with a process scheduler.
- An implementation of the System V ABI for executable files (statically compiled ELF files).
- Suite of core utilities (like those provided by GNU) - these may be partially or totally ported from the GNU coreutils [11].
- An init program for starting the OS's essentials and for starting the root* shell.

18.1.2 Nice-to-haves

- A service manager for starting services at boot and ensuring that they remain running, restarting them if they fail. This may be bundled with the init system, like with systemd on most Linux distributions, or may be a separate program.
- Runs on hardware (not just a VM/emulator).

*Here, "root" means PID 0, not a root user (although if users are implemented, it would be both).

18.1.3 Future possibilities

- A multi-user system with basic passwords and access control on files (file/directory ownership).
- A threads system with multithreading support (multiple threads owned by one process).

18.1.4 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 subsection 19.5), the FS (subsection 19.6), 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.

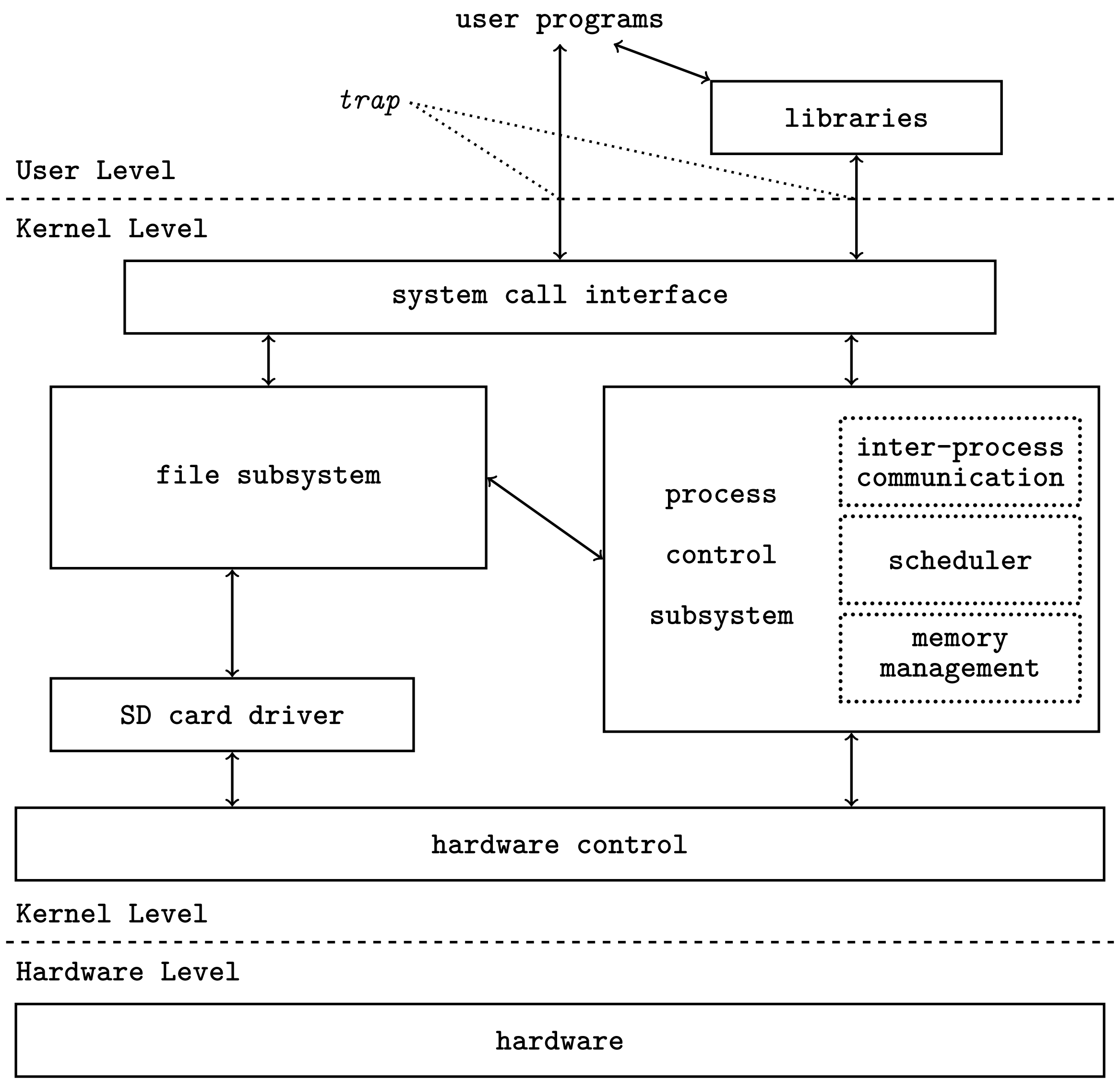

Figure 2: A diagram of how the components of the operating system will interact with each other. Inspired by a diagram from a book on Unix (Figure 2.1 from [3]). The versions of this diagram which are available online are low-resolution, so the version provided here is in a vector format, and this document's source code is available online [20].

18.2 Design of the shell and user interface

18.2.1 Shell

The shell which I will include in the OS will be a port of a POSIX compliant, simple shell, such as Dash [8]. 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.

18.2.2 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 RPi. I will implement some quality of life features, such as terminal scrollback, 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 [4]. 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.

19 Interim: Implementation

19.1 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 [22]. 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 subsection 19.4 for explaining the values placed into various Central Processing Unit (CPU) registers.

19.2 Problems ecountered 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 [16] 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.

19.3 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 [4] 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.

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.

*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.ldd`. 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.ldd` directly.

19.4 CPU setup

When the kernel starts, the CPU will either be in Exception Level (EL) 2 or EL 3 (The Raspberry Pi (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`).

19.5 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).

19.6 Filesystem (FS)

I haven't made much progress on the FS so far, as I was busy learning what I needed to know about ELs when I planned to implement the FS. Also, when I started working on the FS, I discovered I had overlooked the fact that I would need an SD card driver to be able to access storage from the RPi.

I took an existing driver for interfacing with an SD card (via the RPi's eMMC controller) and integrated it into the kernel. I also copied some FS code from the same related work [18], but I haven't finished integrating anything related to a FS into the OS. The virtual FS is unfinished, and so is the FAT FS. This is one of the next components that I will focus my efforts on.

20 Interim: Progress

20.1 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.

20.2 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 [15] and used the arm developer documentation [2] 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 [18], 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.

20.3 Contributions and reflections

The project's external sponsor (who is also my project supervisor), introduced me to several of the related works [14, 17]. 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.

Now that I have reached a stage where I can more easily utilise techniques from Agile software development (subsubsection 17.1.2), I will be able to more successfully meet the deadlines I have set for the rest of the project. I believe I have finished the most difficult part of developing the OS, as I have learned a lot about arm CPUs and OS development best practices. The rest of the project should be easier, as the most complex systems are already implemented, but I will continue to put in as much effort as I can, so that I can stay ahead of the new plan.

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
IO	Input / Output
OS	Operating System
RPi	Raspberry Pi
SBC	Single Board Computer
SOC	System-On-a-Chip
TDD	Test Driven Development
UART	Universal Asynchronous Receiver-Transmitter
XP	Extreme Programming

References

[1] *A generic and open source machine emulator and virtualizer.* URL: https://www.qemu.org (visited on 2021-12-12).

[2] *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).

[3] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., 1986. ISBN: 978-0132017992.

[4] *Bizcat - an 8x16 bitmap font.* URL: https://robey.lag.net/2020/02/09/bizcat-bitmap-font.html (visited on 2021-12-12).

[5] *Clangd: teach your editor C++.* URL: https://clangd.llvm.org/ (visited on 2021-12-12).

[6] *ClangFormat.* URL: https://clang.llvm.org/docs/ClangFormat.html (visited on 2021-12-12).

[7] *CoC.nvim: the Conquer of Completion.* URL: https://github.com/neoclide/coc.nvim (visited on 2021-12-11).

[8] *Debian Almquist Shell (Dash).* URL: https://git.kernel.org/pub/scm/utils/dash/dash.git/ (visited on 2021-10-26).

[9] *Extreme Programming: A gentle introduction.* URL: http://www.extremeprogramming.org (visited on 2021-12-11).

[10] *GDB, the GNU Project debugger.* URL: https://www.sourceware.org/gdb/ (visited on 2021-12-12).

[11] *GNU Coreutils.* URL: https://www.gnu.org/software/coreutils/ (visited on 2021-10-26).

[12] *Introduction to Test Driven Development (TDD).* URL: http://agiledata.org/essays/tdd.html (visited on 2021-12-11).

[13] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Inc., 1984. ISBN: 978-0139376818.

[14] *NetBSD.* URL: https://github.com/NetBSD/src (visited on 2021-11-28).

[15] *`raspberry-pi-os`: Learning Operating System (OS) development using Linux kernel and Raspberry Pi (RPi).* URL: https://github.com/s-matyukevich/raspberry-pi-os (visited on 2021-11-28).

[16] *Recursive use of `make`.* URL: https://www.gnu.org/software/make/manual/html_node/Recursion.html (visited on 2021-12-05).

[17] *RiscOS.* Version 5.29. URL: https://www.riscosopen.org/content/ (visited on 2021-11-28).

[18] *`rpi-boot`: A second stage bootloader for the RPi.* URL: https://github.com/jncronin/rpi-boot (visited on 2021-11-28).

[19] *The OSDev Wiki.* URL: wiki.osdev.org (visited on 2021-10-20).

[20] *The source code (in LaTeX) of the document you are reading.* URL: https://github.com/Ytrewq13/YtrewqOS/tree/master/Documentation/dissertation/interim-report (visited on 2021-12-05).

[21] *Vim - the ubiquitous text editor.* URL: https://www.vim.org (visited on 2021-12-11).

[22] Brian W.Kernighan and Dennis M.Ritchie. *The C Programming Language.* Second. 1988. ISBN: 978-0-13-110362-7.

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
`pwd`, `cd`, `ls`, `stat`, etc.	2
`export`, variables, `set`, etc.	2
`if`, `while`, `for`, `case`, 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
libc	8
`cat`, `head`, `less`, etc.	2
`roff`	1
`man`	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.

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.

A.1 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.

A.2 Gantt chart

A.3 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.

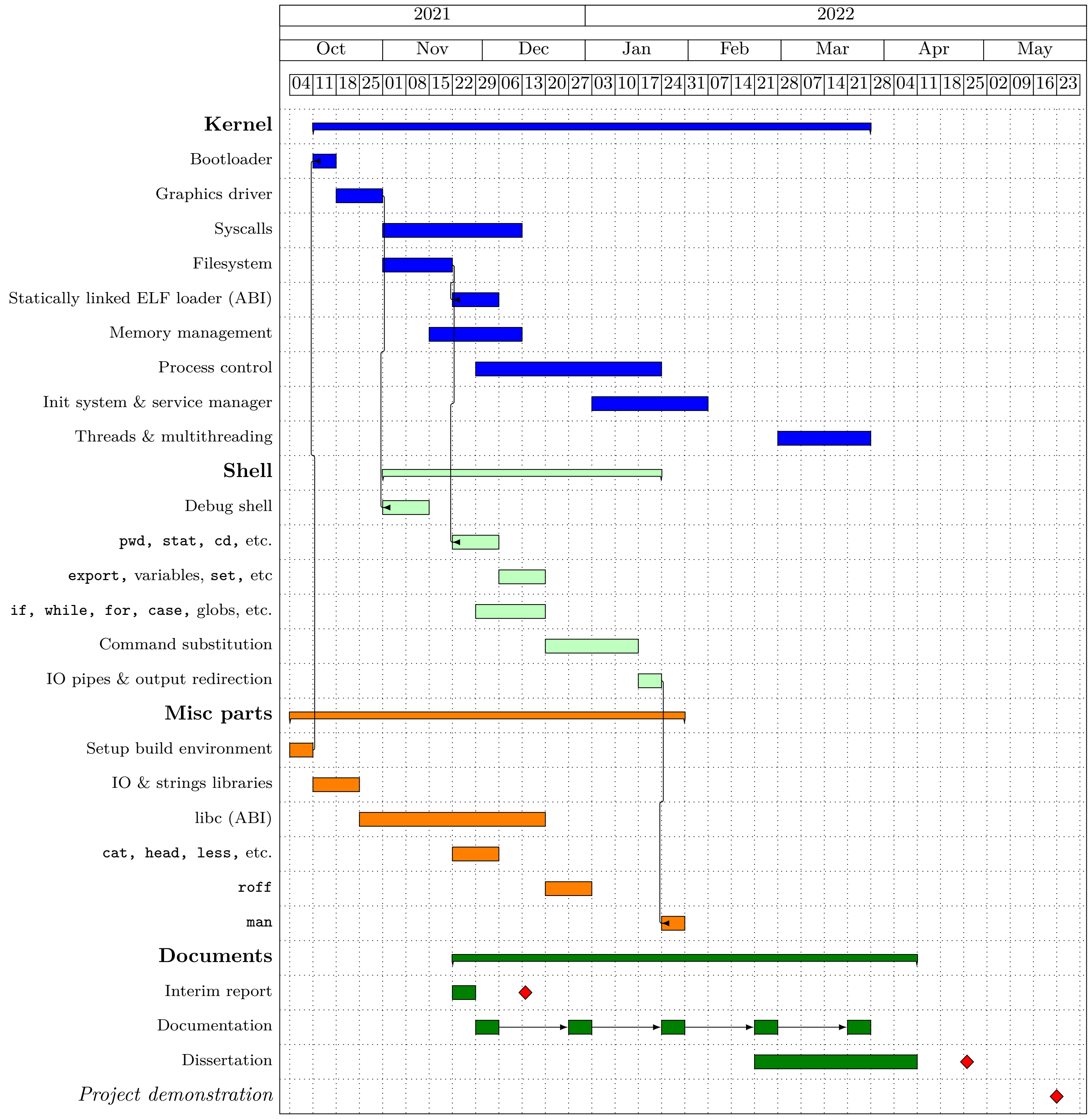

Figure 3: A Gantt chart breaking down the proposed schedule of work for this project.

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 `fork(3)` or a similar function in order to spawn new processes and `execve(2)` 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 `pthread` 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.

A Gantt chart

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
IO	Input / Output
OS	Operating System
RPi	Raspberry Pi
SBC	Single Board Computer
SOC	System-On-a-Chip
TDD	Test Driven Development
UART	Universal Asynchronous Receiver-Transmitter
XP	Extreme Programming

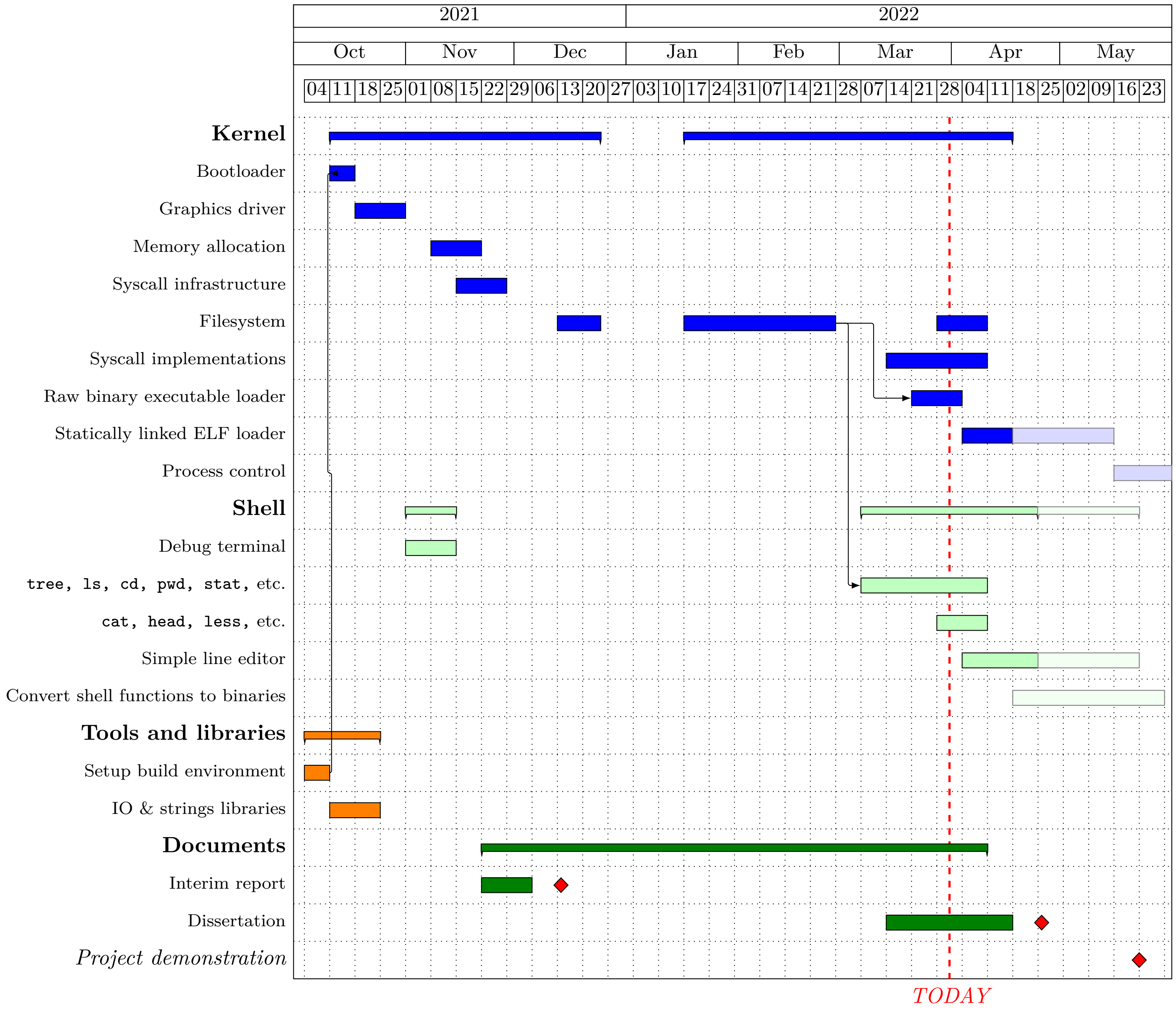

Figure 4: A Gantt chart breaking down the proposed schedule of work for this project.