Operating System for the Raspberry Pi

A Unix-like Kernel, Shell, and Coreutils for the 64-bit Raspberry Pi 3b

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

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

Contents

1 Introduction

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

TODO: expand this

2 Motivation and Background

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.

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a project like this, as it is complex enough that creating an OS for it is a challenge, but also modern enough that no out-of-date technologies will be necessary to work with it.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.

3 Related works

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.

TODO: Expand this.

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 Operating System (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. TODO: refer to system calls implementation/progress here (cross-reference).

TODO: Expand this.

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 reference other implementations when I work on these for my OS.

TODO: Expand this.

- The Linux kernel [Rw7].
- NetBSD [Rw2] (and the rest of the BSDs).

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

- RiscOS [Rw5] – the first operating system written for arm CPUs, kept updated to run on the RPi. Written in assembly language.

TODO: Describe each project and how I will be using them for inspiration / code snippets.

References for the 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).

4 Description of the work

TODO: Modules / work packages.

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 2 on page 9 and then assembled into a Gantt chart in Figure 1.

TODO:

- Talk about the changes since the original set of tasks.

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 1 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 Operating System (OS), especially when the OS is very simple and will not need to run any services.

5 Methodology

TODO

- Mention open source?
- Software development tools used (i.e. vim, CoC, clangd, ctags, etc.)?
- TODO: what is my SW methodology?

6 Design

6.1 Operating System (OS) components

6.1.1 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 [3].

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 processees.	Not started.
A shell	A (POSIX-compliant) shell **or** a port of a simple shell like Dash[3]. 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 1: The work packages of the project and their deliverables.

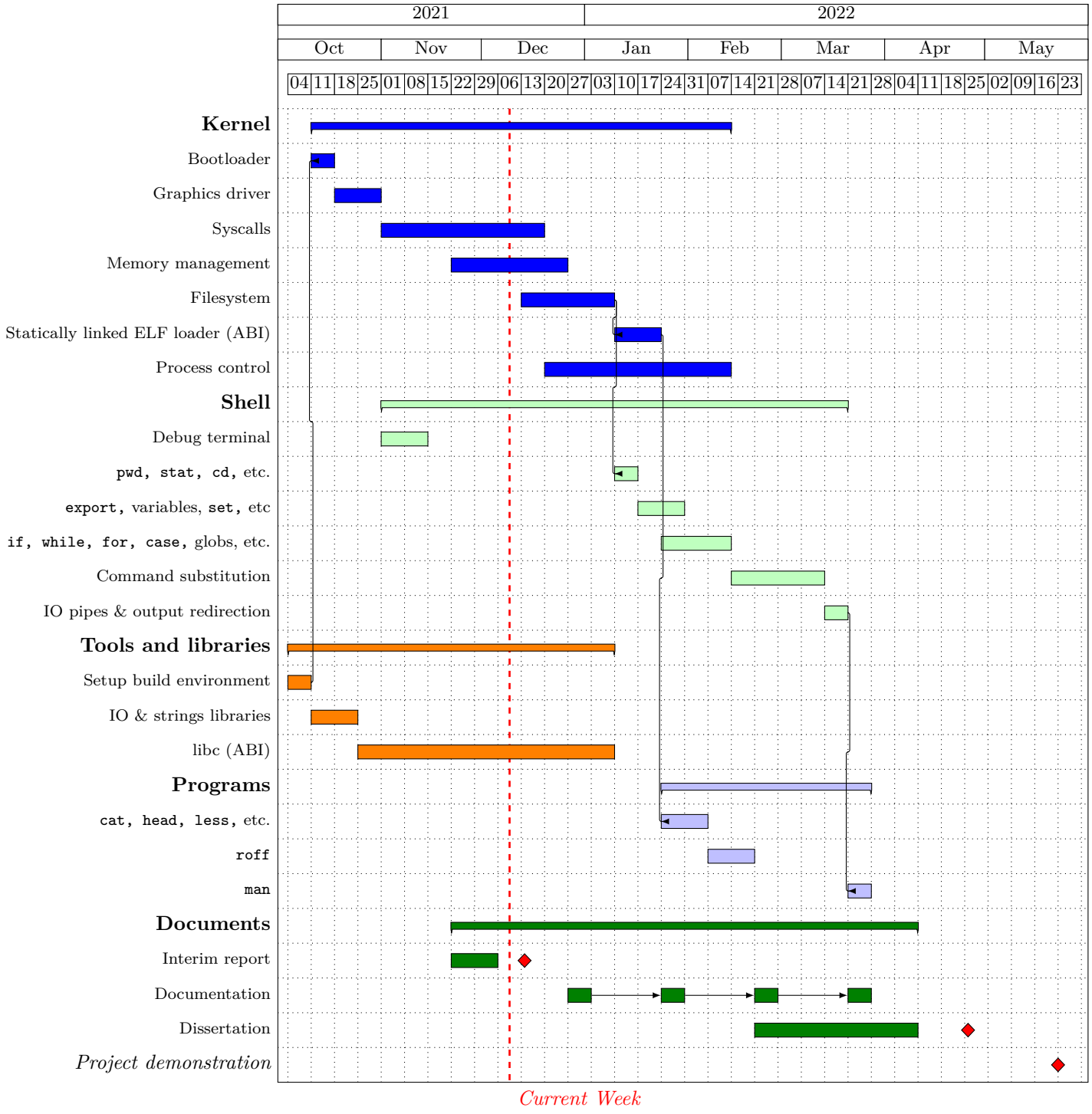

Figure 1: A Gantt chart breaking down the proposed schedule of work for this project.

- 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 [4].
- An init program for starting the OS's essentials and for starting the root* shell.

6.1.2 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).

6.1.3 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).

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 8.5), the filesystem (subsection 8.6), 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.

7 Implementation (if there is any)

TODO: 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.

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 [10]. 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 8.4 for explaining the values placed into various Central Processing Unit (CPU) registers.

7.2 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 [7]

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

†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.

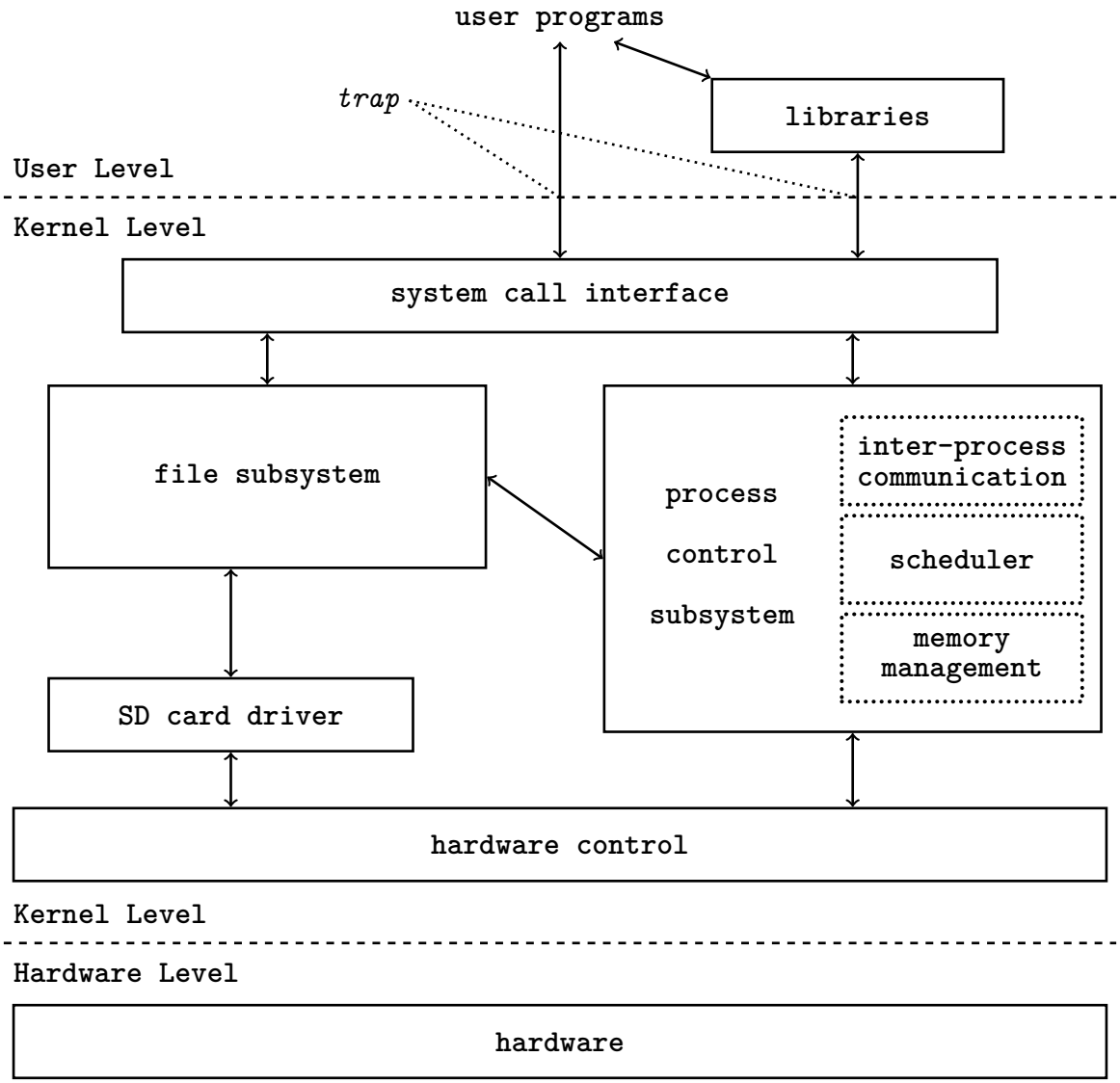

Figure 2: A diagram of how the components of the operating system will interact with each other. Inspired by a diagram from a book on Unix (Figure 2.1 from [2]). The versions of this diagram which are available online are low-resolution, so the version provided here is in a vector format, and this document's source code is available online [9].

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.

7.3 Design changes

TODO: Were there any changes to the design as a result of the implementation?

- Removed tasks
  - Init system and service manager
- Moved some tasks to "future possibilities"
  - Multithreading and multicore scheduling
- Microkernel → something else?

8 Progress

8.1 Project management

TODO: **Project management** covering the tasks as a part of your **work plan as presented in your final proposal** and progress as well as how time and resources are managed. Your interaction with the external sponsor. The need for reviewing the tasks and explanation of/reasons for any adjustment made to the future work plan. Inclusion of a Gantt chart is strongly recommended as a visual indicating the progress of the project.

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. The new plan for the tasks of the project is shown in TODO (table of tasks and time expectations). Table 2 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.

Unexpected difficulties

TODO: Talk about the difficulties and problems I encountered during development.

- Filesystem required SD card driver.
- Memory allocation is a more complex topic than I assumed it would be.
- Programming without `malloc` is difficult. Low-level kernel functions must take pointers as arguments and put results in variables allocated on the stack, as there is no heap. (Also, `malloc` is easiest implemented using a system call `mmap`, but I don't have system calls yet.)

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 (TODO) 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 [6] and used the arm developer documentation [1] 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

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

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.

TODO: Filesystem

TODO: Memory managemant (malloc).

TODO:

- Re-evaluate the parts of the project I think I can complete in the time I have.
- Move the Gantt chart to here?
- Re-plan the organisation of the project modules based on the work I've already done, the current time, and the future.
- Move the planning part to the "Description of the work" section.

8.2 Contributions and reflections

TODO: **Contributions and reflections** providing the details of your achievements and contributions (including the contributions from the external sponsor) up to date as well as a personal reflection on the plan and your experience of the project (a critical appraisal of how the project has been progressing).

8.3 Graphics driver

TODO

8.4 CPU setup

TODO

8.5 System Calls

TODO

8.6 Filesystem (FS)

TODO

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

UART Universal Asynchronous Receiver-Transmitter

References

[1] *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).

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

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

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

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

[6] *`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).

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

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

[9] *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).

[10] 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 plan of work for the project, including how many weeks I think each component will take.

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.

B The source code

TODO

References

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

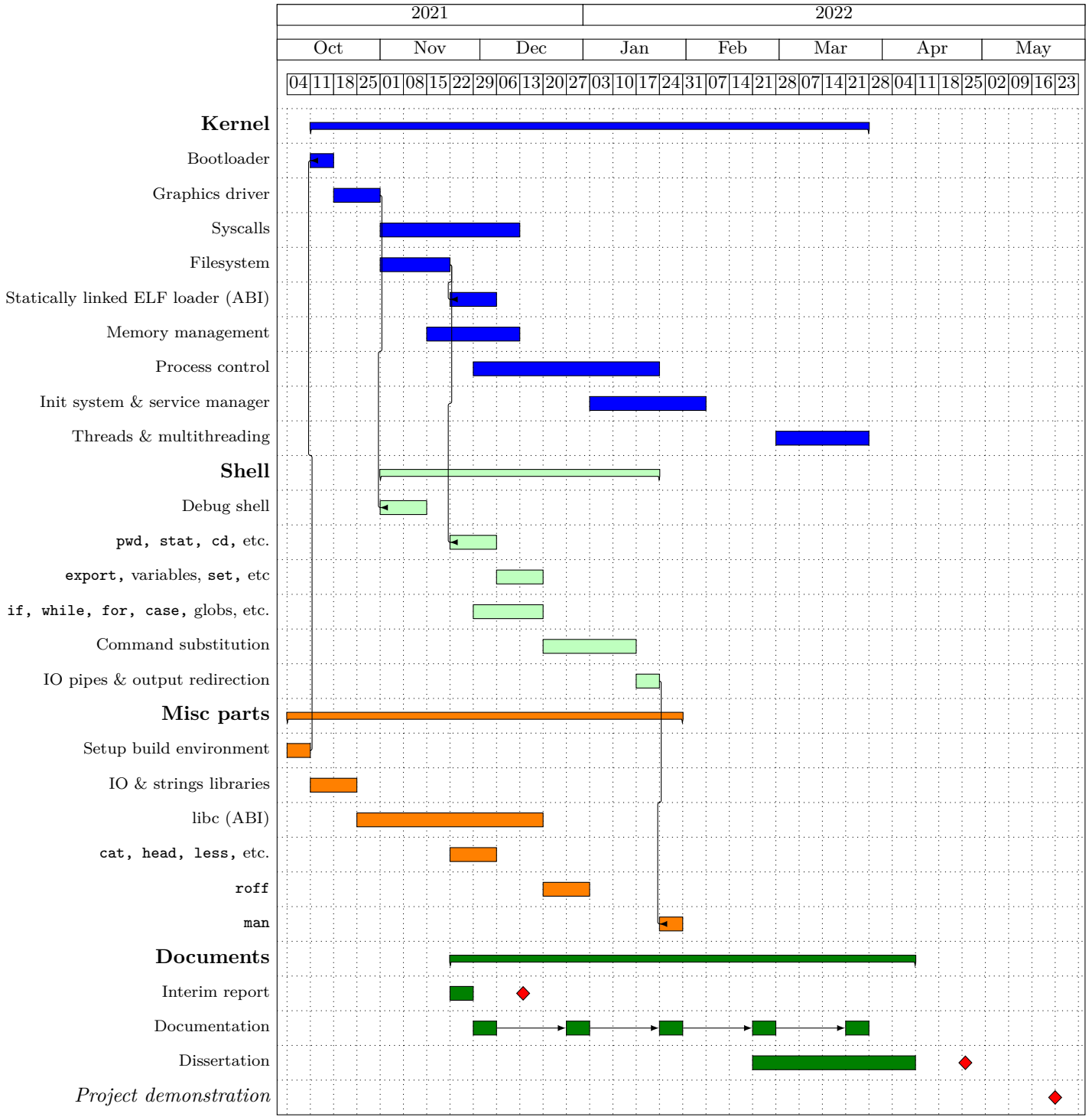

Figure 3: A Gantt chart breaking down the proposed schedule of work for this project.

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[1]. 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.