A modular kernel for the Raspberry Pi: Project Specification

Thomas Archbold
1602581
University of Warwick

Background

In most operating systems, many design decisions are made in order to keep things simple for the user, by keeping most of the technical details hidden. In most cases, this is an appropriate approach: needlessly offering more choices for low-level tasks that are usually handled by the operating system, such as CPU scheduling algorithms, would only serve to confuse the average user. It may actually be detrimental to the security and the stability of the system by opening up more opportunities for errors to be introduced. This more insulated approach does mean, however, that the user never really knows what is going on "under the hood", and indeed whether greater performance can be achieved by making *different* fundamental decisions. Furthermore, a number of operating systems exist for the Raspberry Pi, some focusing on ease-of-installation, others on Internet of Things integration, such as the NOOBS [1] Linux distribution or the Windows 10 IoT Core distribution [2]. Yet, none exist to serve as an experimental operating system, designed as a testbed for making and changing these low-level behaviours. This project aims to fill this gap for the operating systems enthusiast, one who wishes to test for themselves the different approaches to CPU scheduling, interprocess communication, and filesystems. It will give the user the ability to alter the fundamental ways in which their machine operates by compiling different modules to handle different tasks, enabling for a more flexible operating system where such things can be tweaked at any point.

Main goal

The goal of this project is to create a modular operating system for the Raspberry Pi 2 Model B that is capable of loading different modules at compilation time to tackle CPU scheduling, interprocess communication, and filesystems in a variety of ways. Specifically, it must have some way to run and switch between multiple processes using a CPU scheduler; to use both shared memory and message passing for interprocess communication; to create, read, update, and delete files and directories using a custom filesystem; and to interface with the SD card for permanent/mass storage. To achieve this, it must implement an interface for compiling different modules, similar to Linux's `insmod`, `rmmod`, and Kbuild system [3, 4]. Furthermore, as executing processes forms a key functional requirement for the project, there must be a convenient way to load programs into memory and begin their execution. A solution to this is to implement a basic shell/command interpreter.

Finally, a key objective of this project will be to get the operating system to work entirely on real hardware, and not solely in an emulated environment. This includes booting from the SD card installed in the Raspberry Pi. As the boot process is handled by the Pi's System on Chip (SoC), so booting will be possible without writing a custom bootloader. On top of booting from it, the operating system must interact with the SD card in conjunction with a filesystem for permanent/mass storage. Finally, it must be capable of taking input from a keyboard connected via USB, and printing output to a physical screen via its HDMI port.

The kernel

Configurable modules

The project must implement the following as modules, which may be configured at compilation time by the user:

- CPU Scheduling:
    - First Come First Served
    - Round Robin
    - Shortest Job First
    - Shortest Remaining Time First
    - Priority Scheduling (preemptive and non-preemptive)
    - Lottery Scheduling
- Interprocess Communication
    - Message passing
    - Shared memory
- Filesystem
    - persistent
    - load-on-request

Stretch goals

Some stretch goals which should be implemented to show understanding of more complex structures would be some more intricate scheduling algorithms, including the following [5, 6, 7, 8]:

- Completely Fair Scheduler
- Multiple Queue Skiplist Scheduler, MuQSS
- Multilevel Queue and Multilevel Feedback Queue
- $\mathcal{O}(n)$ Scheduler
- $\mathcal{O}(1)$ Scheduler

In order to give the operating system more purpose and to increase usability, the collection of relatively simple programs on offer should be extended, including a mix of long running CPU- and I/O-bound programs. This will mean that the relative performance of the schedulers may be seen more easily. While the Not Recently Used (NRU) algorithm will be used for page replacement due to its low overhead and decent performance, other algorithms could be explored and implemented as modules. These may include: First-In-First-Out (to highlight its poor performance), the Clock Page Replacement algorithm, and the Least Recently Used algorithm [9].

Further extensions

Beyond these goals, further extensions would focus on increasing the usability of the system, and start to shape it into one which someone might actually use to get things done. One of the simpler ways to achieve this would be to write a text editor. Additionally, implementing networking into the operating system would vastly increase its usability and general usefulness. Such goals are rather far-fetched given the time frame of the project, but would form meaningful projects later in the life of the operating system.

Out-of-scope

Features which will not be implemented in the project include graphical user interfaces and any form of security. Graphics would increase the complexity of the project too much, and provide too little reward, to be considered a worthwhile goal. While security would be easier to implement, for example by following suit of Linux's permissions interface [10], it would again detract attention from features more in line with the project's goals. After all, the operating system produced will only be experimental and designed for use by one user, and as such security will be an unnecessary feature.

Hardware

Compared to the Raspberry Pi 1, the Raspberry Pi 2 Model B has:

- 900 MHz quad-core ARM Cortex-A7 CPU
- 1GB RAM

Like the Pi 1 Model B+, it has:

- 40 GPIO pins
- 4 USB ports
- Full HDMI port
- MicroSD card slot
- 100 Base Ethernet
- VideoCore IV graphics Core
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)

The main reason for choosing to work with the Raspberry Pi was due to its simple boot process, which is detailed below. In particular, as it is handled entirely by its SoC, it means a custom bootloader to load the kernel into memory and transfer control to it will need not be written. The Raspberry Pi 2 Model B in particular uses the BCM2836 processor, whose underlying architecture is identical to the 1's BCM2835 and the 3's BCM2837 chips. The only difference is that the 2 uses the quad-core Cortex-A7 cluster as opposed to the ARM1176JZF-S or the quad-core ARM Cortex-A53 cluster, as used by the 1 and 3 respectively [11, 12]. Therefore, as the choice between specific models of the Pi would make little significant difference to the outcome of the project, it made sense to opt for the one already available to the author at the time the project was conceived, namely the Raspberry Pi 2 Model B.
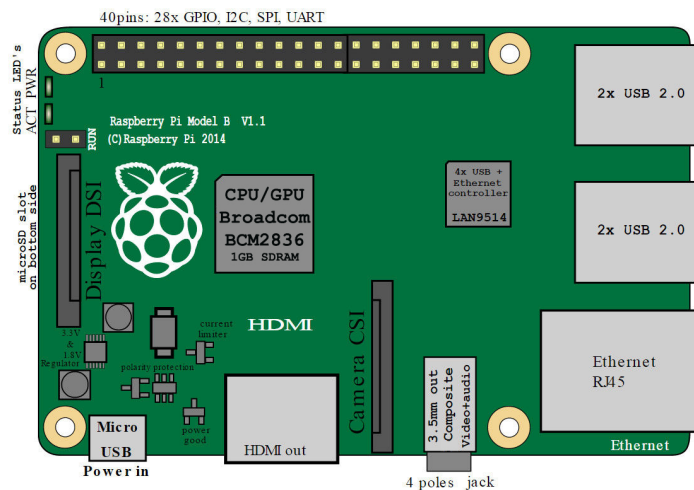
Figure 1: Raspberry Pi 2 Model B

A note about the Raspberry Pi's boot process

Methodology

The methodology best suited to the project will be a mix between plan-driven and agile approaches. The early stages of the operating system's development will benefit from the former, as the requirements, such as booting before memory management before writing scheduling algorithms, will abide by a rigid
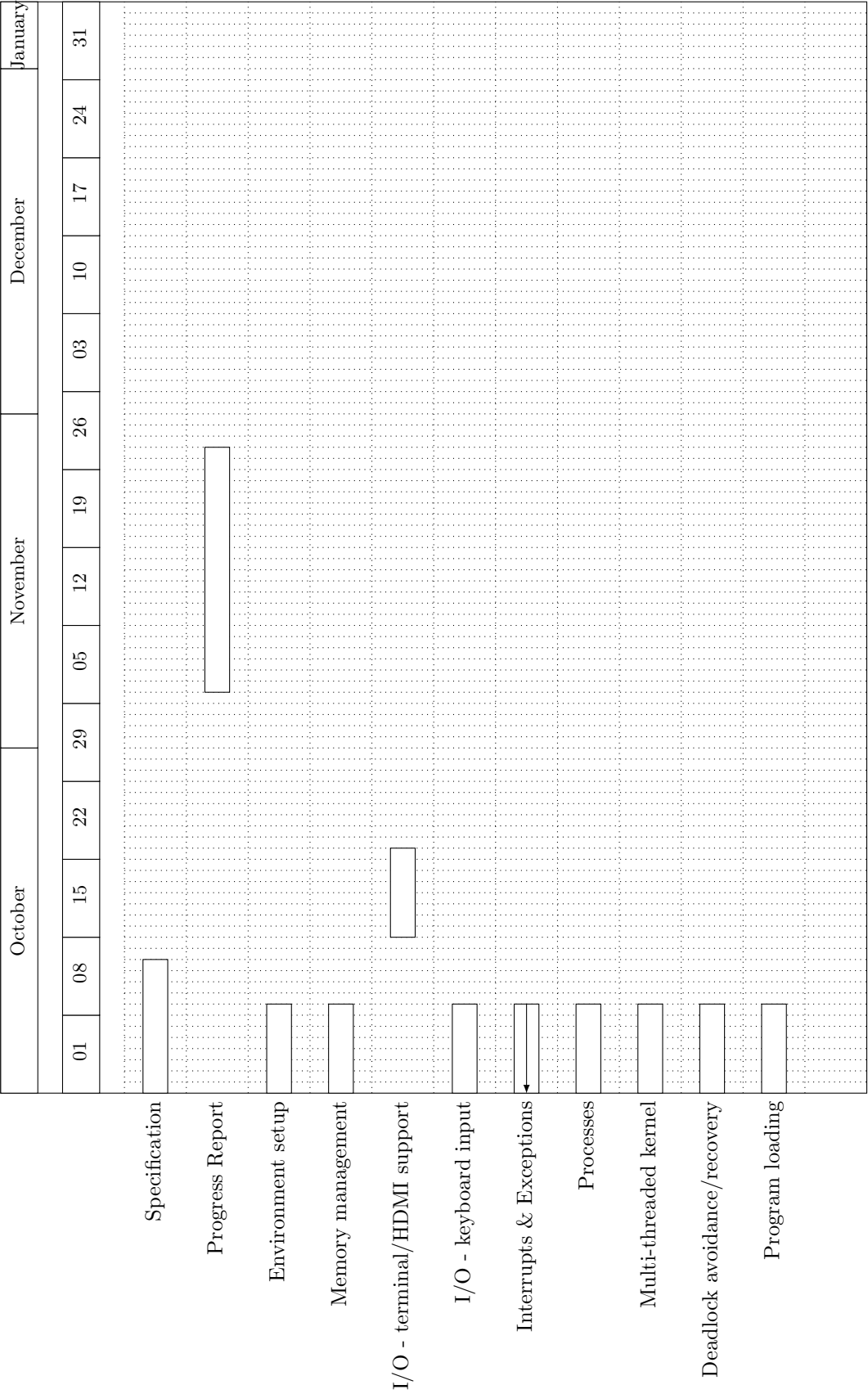
structure. An incremental approach will likely be used as opposed to a waterfall methodology, however, due to its less restrictive nature, and to offer choice when it is appropriate about what to implement next. After the foundations are laid, the project will likely move to a more agile approach, where scrum cycles will be useful both for their flexibility and choice, and their focus on finishing one aspect of the project at a time.

Throughout the project, regular meetings will be taken with the supervisor to discuss progress, current problems, and ideas for solutions when necessary. Organisation will of course be a key aspect to the success of the project, and the project timetable will be updated to reflect the project's progress. The meetings will start at once every fortnight in Term 1, and increase to once weekly in Term 2, simply due to timetabling and course load for other modules.

Testing

The project will be tested incrementally. In its early stages, progress will simply not be able to be made until some systems operate correctly, so thorough manual testing of such areas will be vital as there will simply not be the platform to write dedicated unit tests. As it progresses, and unit tests become more viable, they will be written to cover most likely paths of execution to identify shortcomings of the system, and dealt with accordingly. Since the project's aim is to create a configurable operating system, manual testing will again need to be undertaken in order to verify whether it works under the various combinations of modules. This will test both the correctness and the stability of the system rigorously, two of the most important goals of any piece of software.

Timetable

Gantt Chart

Task	October					November				December				January
	01	08	15	22	29	05	12	19	26	03	10	17	24	31
Specification														
Progress Report														
Environment setup														
Memory management														
I/O - terminal/HDMI support														
I/O - keyboard input														
Interrupts & Exceptions														
Processes														
Multi-threaded kernel														
Deadlock avoidance/recovery														
Program loading														

	January				February				March				April			
	07	14	21	28	04	11	18	25	04	11	18	25	01	08	15	22	29
Presentation																	
Final Report																	
User mode																	
Syscalls																	
Fork & execute																	
Shell																	
Threading																	
Mass storage																	
Filesystem																	

Technologies

The following technologies will be used by the project:

- Git - version control

- Github - to access the project from multiple sources, as well as to back it up

- C - the language in which most of the operating system will be implemented

- ARM assembly - used when C is unavailable/inappropriate [13]

- GCC cross compiler for ARM EABI - for cross compiling for the target processor, the Cortex-A7

- QEMU - for emulating the Pi to allow quicker and safer testing [1]

- Make - automate the build process

Resources

The following documentation will be used throughout for reference to the architecture of the Cortex-A7 processor and its intruction set, and the peripherals on the Pi:

- Cortex-A7 MPCore Technical Reference Manual

- ARM Cortex-A Series Programmer's Guide

- Broadcom BCM2835 ARM Peripherals Manual

Additional guidance will be taken from the MINIX book [14], Stanford's Pintos [15], and [16].

Legal, social, ethical, and professional considerations

All software used to build the project is available to use under the GNU Public License. Throughout the project's development, some testing will be required from people other than the creator, to gain informal feedback especially with regards to usability; these people are likely to be friends and colleagues, hence the social, ethical, and professional issues are insignificant.

References

[1] "Noobs." `https://www.raspberrypi.org/downloads/noobs/`. Page accessed: 2018-10-08.

[2] "An overview of windows 10 iot core." `https://docs.microsoft.com/en-us/windows/iot-core/windows-iot-core`. Page accessed: 2018-10-08.

[3] "insmod(8) - linux man page." `modprobe://linux.die.net/man/8/insmod`. Page accessed: 2018-10-03.

[4] "Kbuild: the linux kernel build system." `https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system`. Page accessed: 2018-10-09.

[5] "Cfs scheduler." `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`. Page accessed: 2018-10-03.

[6] "Kernel patch homepage of con kolivas." `www.users.on.net/~ckolivas/kernel/`. Page accessed: 2018-10-03.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, ch. 5: Process Scheduling. Wiley, 9 ed., 2014.

---

[1]QEMU does not simulate a system timer (at least for the Raspberry Pi 2), so some testing will eventually need to be done on real hardware

[8] "Scheduling: Linux." `https://en.wikipedia.org/wiki/Scheduling_(computing)`. Page accessed: 2018-08-30.

[9] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*, ch. 4: Memory Management. Pearson, 3 ed., 2009.

[10] "Ownership and permissions." `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Step_by_Step_Guide/s1-navigating-ownership.html`. Page accessed: 2018-10-09.

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

[12] "Bcm2837." `https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md`. Page accessed: 2018-10-08.

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

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

[15] "Pintos." `https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html`. Page accessed: 2018-10-09.

[16] "Building an operating system for the raspberry pi." `https://jsandler18.github.io/`. Page accessed: 2018-10-09.