A modular kernel for the Raspberry Pi: Progress Report

November 21, 2018

Thomas Archbold
1602581
University of Warwick

1 Introduction

2 Background

3 Current progress

The project is currently at a point at which the operating system is able to successfully boot in the emulated environment provided by QEMU. Since it has been capable of doing this since `boot.s` was written, early on in Term 1, it is important to note the specific stages of initialisation performed by the kernel at this point, as well as discuss the environment setup that has enabled this point in development to be reached.

3.1 Development environmet

The project is being developed on a machine running Linux kernel version 4.16 onwards. Since the target environment, the Raspberry Pi 2 Model B, is different to that on which it is being developed, a cross-compiler is required to compile code that will run on the target machine, as opposed to the host. In particular, available for download on the ARM developer website [1] is the GNU Embedded Toolchain, which provides tools to target ARM Cortex family of processors, including the GNU Compiler Collection (GCC). Conveniently this suite of tools is available from Arch Linux's package manager, pacman [2], and this is the version of the cross-compiler used in the makefile.

Before writing any code, as the author had little-to-no prior experience in systems programming, research had to be undertaken in order to get acquainted with this environment. In particular, this involved skimming over the various peripherals manuals, technical reference manuals, and programmer's guides for programming on the Raspberry Pi to learn more about its hardware and how to interface with it using ARM assembly. [3] and [4] detail the peripherals on board the Raspberry Pi 2 Model B, the layout of their related registers, and how to read and write them to do meaningful things with the hardware, while [**?**] and [**?**] provided help on ARM assembly's syntax, and how and when to use specific instructions.

Particularly important so far have been the sections of the peripherals manual on GPIO and UART, as until the implementation for the mailbox interface is working, all input and output is done through the serial connection provided by the UART. Since there were issues with getting the Pi to run on real hardware, information about the GPIO peripheral was needed in order to write basic low-level debugging functions, mainly in the form of getting the green ACT LED to blink for various return values of functions.

3.2 boot.s

The first piece of code to be written was `boot.s`, which is responsible for providing the basic setup of the entire system, which includes initialising a minimum C environment. In particular, it sends three of the four cores on the CPU to shutdown (to decrease overall complexity of the system, as discussed in the specification), initialises the stack pointer at address `0x8000`, sets up the BSS segment (where

statically-allocated variables that are not explicitly initialised are stored) and zeroes it out (as required by the C standard), and then loads our C kernel entry point, `kernel_main`, into memory to begin its execution. Note that the Program Counter for the kernel starts at address `0x8000` and grows upwards, so the stack can safely start at `0x8000` without interfering with the kernel (as it grows downwards).

3.3 linker.ld

The code in `linker.ld` is responsible for linking all of the compiled object files into one final executable. There are scripts which do this for user-space programs, but since we are our own user-space, being the kernel, we have to create one for ourselves. The various sections that the script defines are as follows:

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

We also define the entry point of our entire operating system in this script, namely the `_start` routine from `boot.s`. This script also sets the symbols `__start` and `__text_start` to be `0x8000`, which is where the bootloader will put the kernel image - the code from `boot.s` will be put in the first part of this section, `.text.boot`. We use `KEEP` to tell the compiler not to try to optimise the code in `.text.boot`, and `ALIGN` sets the current address to the next available page (the next address divisible by 4096). The `,rodata`, `.data`, and `.bss` sections are then declared in much the same way.

3.4 Makefile

The makefile was written to speed up the build process, and there are only a few features to note. First is that here we specify that we are using the `arm-none-eabi` toolchain, for the compiler to target the Raspberry Pi's architecture as opposed to our own, in particular the Cortex-A7 processor. The `-fpic` compilation flag is currently being used while we are ignoring the Memory Management Unit, and in particular virtual memory, on the Pi, as it creates position-independent code, and will keep separate applications from interfering with each other within this single address space when processes are implemented. The `-ffreestanding` compiler and linker flag, and the `-nostdlib` linker flag, specifies that we are writing code in a freestanding environment, and as such do not expect much of the C standard library to exist, or for program startup to necessarily to be at `main()`. Specifically, we only have access to the following header files: `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, and `<stdnoreturn.h>`. The rest of the standard library must be implemented ourselves.

3.5 Atags

The first piece of meaningful setup that can be done when the kernel is booted is to get information about the memory available to the system. On the Raspberry Pi, the bootloader creates creates a list of information about the hardware called Atags, places it at address `0x100`, and passes it as the third parameter to `kernel_main` in register 2. Each tag in the list contains a header consisting of two unsigned 32-bit values: the size of the tag (in 32-bit words), and the tag value. Each header is then followed by information specific to that tag, aside from `ATAG_NONE`, which has no associated data, and `ATAG_CORE`, whose data is optional. To access the information in each of the tags when we come across them, we must match the layout of each of the tags [5] by defining appropriate C structs in `atag.h`.

To find the amount of memory on the device, we can skip through the Atags list (using pointer arithmetic and information about the tag's size in its header) until we come across the `ATAG_MEM` tag. Then it is simply a case of return the value in the `size` field of the `atag_mem` struct.

3.6 Organising memory

Throughout the operating system's execution, different processes will require memory to perform computations. Although the only thing doing anything meaningful at the moment is the kernel, which can theoretically use any memory it wants, in order to impose some order for later implementations of processes and user space, we split the available memory up into 4KiB pages. The total number of pages is therefore given by the total amount of memory divided by the page size.

To organise the pages, we give each page a header which stores metadata about the page, including whether the page has been allocated, whether it is a kernel page, and whether this page is part of the heap (used later when dynamically allocating memory). We organise all of the page headers into a linked list directly after the end of the kernel image, using the `__end` variable from the linker script. Each page also stores the virtual address to which it maps; as virtual memory has not yet been implemented, a page's virtual address is simply its index in the page list multiplied by the page size. All that remains is to iterate over the page header list to initialise each page's metadata, and add each page to a linked list of all the free pages.

Naturally the next feature implemented was allocating and freeing pages. Allocating a page is done by simply popping the head of the free page list, setting the appropriate flags, and returning the address of the page. Freeing, meanwhile, is done by passing the address of the page to free, again setting the appropriate flags, and appending this page back to the fre page list.

3.7 Allocating memory

A 1MiB portion of memory located directly after the page headers is reserved for the heap - while the choice of 1MiB is fairly arbitrary, it should be large enough to suffice for all dynamic memory needs, and small enough to not use a significant portion of memory that may be desired by user code. We define the struct `heap_segment` for keeping track of heap allocations, such as the segment size and whether it has been allocated (useful for avoiding external fragmentation later), and initialise the heap by declaring a single heap segment whose size is equal to that of the heap - as more memory is allocated, this initial segment will be split into smaller ones, which in turn may be split further to satisfy requests for differing amounts of memory. Whenever memory is requested, we add the size of the header to the request and 16-byte align it, so as to actually reserve some space for the heap segment metadata.

To allocate a segment, we step through the list of segments in the heap and look for the one best satisfying the number of bytes requested, and not in use. If the current segment is large (i.e. the segemnt size if ≥ 2 times the header size), we split it into two segments and only use one of them. Once a segment satisfying the request is found, we return a pointer to the memory directly after the header. To free an allocation, we set the appropriate flags, and then attempt to merge consecutive free segments, checking both to the left and the right of the current segment.

3.8 Serial output

Initial output was done using the UART on board the Pi, meaning text was sent and received through serial ports. This was mainly due to simplicity in early builds of the system, and while this can be achieved in the final version of the project by using a USB-to-TTL cable, ideally it will use the HDMI port on the Pi instead, which requires interfacing with the Mailbox peripheral (discussed later). This was all done by interacting with the GPIO pins on the Pi, which is done entirely through Memory Mapped I/O (MMIO) - that is, by reading from and writing to predefined memory addresses. A peripheral on the Raspberry Pi is simply an address to and from which you may read and write data, and all may be described by an offset from the Peripheral Base Address; this is `0x3f200000` on the Raspberry Pi 2 Model B. Moreover, a register is a 32-bit chunk of memory that a peripheral may read from or write to. The BCM2835 Peripherals Manual gives the UART base address as `0x7e201000`. Thus, in `gpio.c`, after setting all the required flags for using the UART, we can implement a serial `putc()` by checking that the FIFO is not full and writing our data to the Data Register, and `getc()` by checking that the FIFO is non-empty and reading from it. [1]

3.9 HDMI output

The final goal, however, is to not use any extra adapters to make serial output work - instead, output will be through the HDMI port on the Pi and to do so, familiarisation with the Mailbox peripheral has been important. The Mailbox is a peripheral that facilitates communication between the ARM CPU and the VideoCore GPU [6], and starts at offset `0xb880`. We may get data from the GPU via the read register, pass data to the GPU using the write register, and check if either of these are empty or full using the status register, at offsets `0x00`, `0x20`, and `0x18` respectively. Furthermore, a channel is a number that

[1] From the manual: "Physical addresses range from `0x20000000` to `0x20ffffff` for peripherals. The bus addresses for peripherals are set up to map onto the peripheral bus address range starting at `0x7e000000`. Thus a peripheral listed at `0x7ennnnnn` will be available at physical address `0x20nnnnnn`."

gives meaning to the data being sent to and received from the GPU - for interacting with HDMI, we need the Property channel, channel 8. This provides a means to get and set data about various hardware devices, one of which is the framebuffer.

In order to ask the GPU for a framebuffer, we need first to be able to communicate with it by sending messages and parsing the received response. The messages we send it, for example, are to set the framebuffer's physical and virtual dimensions, and its colour depth (bits per pixel). Once all the parameters are set, we can ask the GPU for a framebuffer, using the `FB_ALLOCATE_BUFFER` tag (defined in `mailbox.h` as `0x00040001`). The returned values are a pointer to this framebuffer and its size.

3.10 Booting on real hardware

This is the first point at which the project has required testing on real hardware, in order to verify that a framebuffer is being correctly requested and supplied by the GPU. This required the kernel to be installed on an SD card and for it to be run on the physical board of the Pi. It is now helpful to detail the unique boot process of Pi, which makes this stage much easier.

3.10.1 The boot process of the Raspberry Pi

The boot process relies on closed-source proprietary code programmed into the SoC processor [7] which cannot be modified. Importantly, the ARM CPU is not the main CPU - it is a coprocessor to the VideoCore GPU. Upon powerup, the ARM CPU is halted and the GPU is run. The firmware then loads the bootloader from ROM to the L2 cache and executes it. This first stage bootloader mounts the FAT32 boot partition on the SD card so that the second stage bootloader may be accessed. This is its only responsibility, to load 'bootcode.bin'. The first stage bootloader is programmed into the SoC itself during manufacture and cannot be reprogrammed by the user. Next, the second stage bootloader ('bootcode.bin') then retrieves the GPU firmware from the SD card, programs the firmware, then starts the GPU. The GPU firmware ('start.elf') is loaded, and allows the GPU to start up the CPU. An additional file 'fixup.dat' is used to configure the SDRAM partition between the GPU and CPU. Here the kernel image is loaded, the CPU is released from reset, and control is transferred to it to execute the kernel.

After the operating system is loaded, the code on the GPU is not unloaded; instead, it runs its own simple operating system, called Video Core Operating System (VCOS). The kernel can then use this to communicate with the services it provides (e.g. providing a framebuffer, as above) using the Mailbox Peripheral and interrupts (the GPU is able to produce ARM interrupts). The GPU is not only in charge of graphical functions - it also controls clocks and audio, for example. In this way the GPU firmware is similar to a normal PC's BIOS (Basic Input/Output System) [8, 9].

4 Project management

4.1 Development

4.2 Testing

5 Next steps

6 Reflection

7 Ethical consent

References

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

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

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

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

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

[6] "Accessing mailboxes." `https://github.com/raspberrypi/firmware/wiki/Accessing-mailboxes`. Page accessed: 2018-11-21.

[7] "Raspberry pi firmware." `https://github.com/raspberrypi/firmware`. Page accessed: 2018-11-21.

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

[9] "Raspberry pi: Level of hackability of raspberry pi." . Page accessed: 2018-11-15.

A modular kernel for the Raspberry Pi: Project Specification

12 October, 2018

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, with Linux's NOOBS [1] distribution, others on Internet of Things integration, such as 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), 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

The kernel will be built using the cross-compiler from GCC for `arm-none-eabi`, which provides a toolchain to target the System V Application Binary Interface (ABI). As a result, programs and frag-

ments of programs on disk, and by extension the kernel itself, will be in the Executable and Linkable Format (ELF) after compilation and linking. The kernel will use just a single core of the four available to the BCM2836, but will support multithreading, both at the kernel and user levels, with appropriate interfaces being written in both cases.

The memory available to the operating system will be organised into pages, and furthermore it will use a dynamic memory allocator, similar to the C standard library's `malloc()` and `free()`, to further split the available memory into segments. Processes will need to be loaded into and out of memory, and as such will need an appropriate representation as a Process Control Block (PCB), and will need to be stored in a Process Table to facilitate context switching. On top of this, the kernel must also be able to handle interrupts and exceptions to safely halt processes and bring them out of memory. At this point, CPU scheduling will need to be tackled, and a long-term and multiple short-term schedulers implemented. As QEMU does not simulate a system timer, the move to working on real hardware will coincide with the introduction of multitasking, involving taking input via USB keyboard and printing output via HDMI.

With the possibility of multiple running processes, synchronisation will need to be tackled, most likely using semaphores, and the issues of deadlock avoidance, detection, and correction will need to be considered. Furthermore, solutions for interprocess communication will then be developed, most likely starting with shared memory due to its simplicity. Message passing will follow as a configurable module. Beyond this, a filesystem can then be implemented and development can move to focus more on user space, including a command interpreter, actually accessing mass storage, and implementing `fork()` and `execute()`. At this time the notion of syscalls and operating system traps will also need to be developed. As an operating system needs to be written in a freestanding (as opposed to hosted) environment, a standard library will be continually developed over the course of the project.

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, details of which can be found here [11]. 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 [12, 13]. 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.

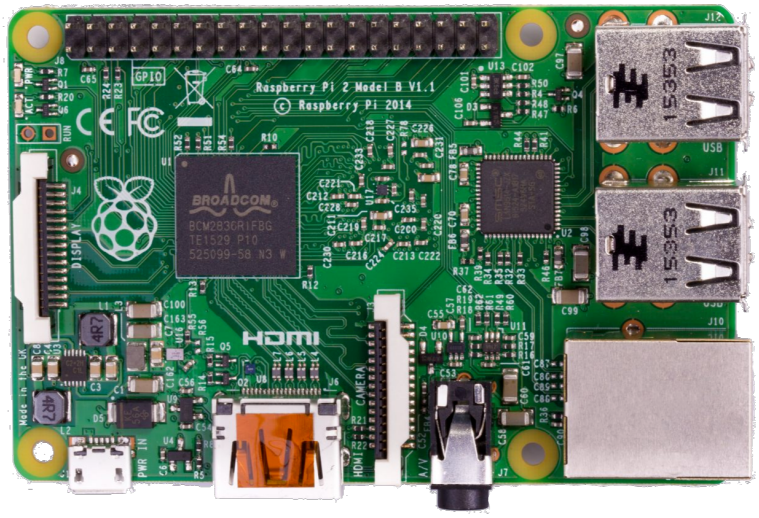


Figure 1: Raspberry Pi 2 Model B [14]

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

See Appendix.

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 [15]
- 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 instruction 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 [16], Stanford's Pintos [17], and [18].

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.

[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.

[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

[11] "Raspberry pi boot process." https://www.raspberrypi.org/forums/viewtopic.php?f=63&t=6685. Page accessed: 2018-10-11.

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

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

[14] "Raspberry pi 2 model b." https://www.raspberrypi.org/products/raspberry-pi-2-model-b/. Page accessed: 2018-10-11.

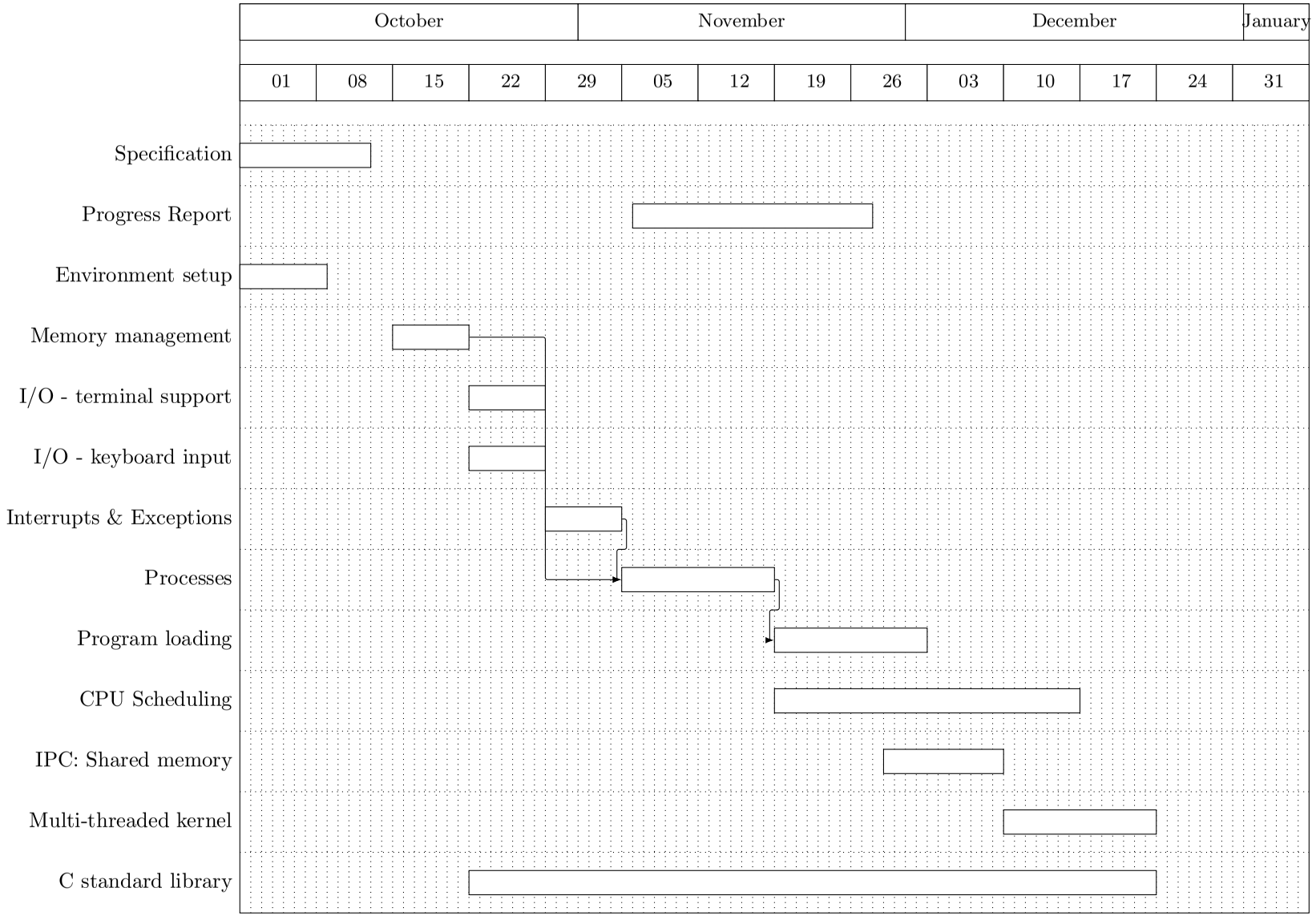
[15] "C: Things c can't do." https://wiki.osdev.org/C#Things_C_can.27t_do. Page accessed: 2018-08-30.

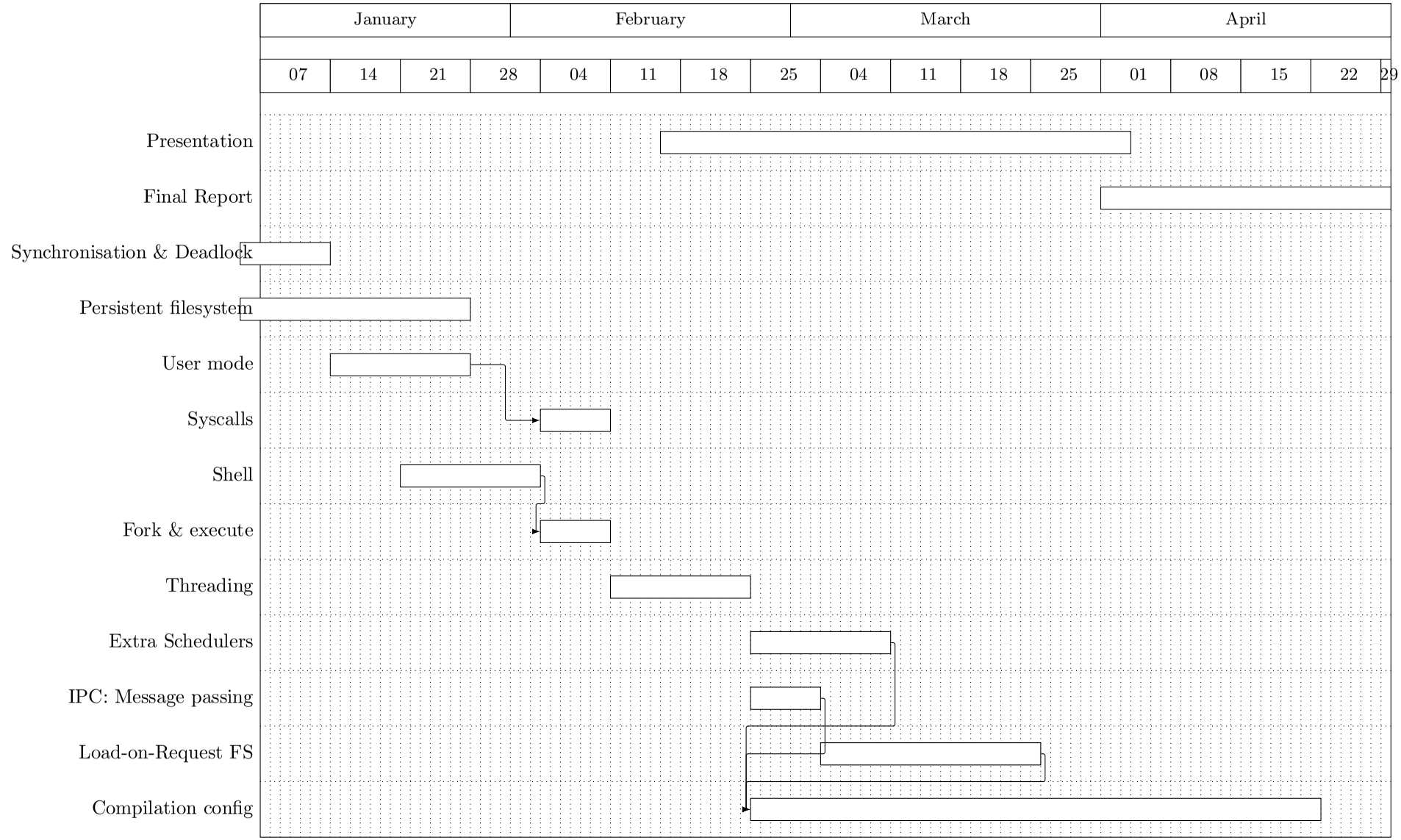
[16] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Pearson, 3 ed.

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

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

Appendix


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 support
I/O - keyboard input
Interrupts & Exceptions
Processes
Program loading
CPU Scheduling
IPC: Shared memory
Multi-threaded kernel
C standard library


January
February
March
April
07
14
21
28
04
11
18
25
04
11
18
25
01
08
15
22
29
Presentation
Final Report
Synchronisation & Deadlock
Persistent filesystem
User mode
Syscalls
Shell
Fork & execute
Threading
Extra Schedulers
IPC: Message passing
Load-on-Request FS
Compilation config