A moduler kernel for the Raspberry Pi: Project Specification

October 2, 2018

Thomas Archbold
1602581
University of Warwick

1 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 and disk 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, but none exist to serve as a testbed for these low-level decisions. 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, disk scheduling, interprocess communication, and filesystems. It will give the user the ability to dynamically change the fundamental ways in which their machine operates by loading different modules to handle different tasks, without the need to reboot, enabling for a more flexible operating system where such things can be tweaked at any point.

2 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 dynamically loading modules to tackle CPU scheduling, disk 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; interact with a hard disk drive and a disk scheduler for permanent/mass storage; and be able to create, read, update, and delete files and directories using a custom filesystem. To achieve this, it must implement an interface for loading/removing modules and must do so safely and stably.

2.1 Core aims

The operating system must be able to:

- boot successfully from either GRUB or a custom bootloader
- take keyboard input via the Pi's USB port
- send output to a screen using the Pi's HDMI or serial port
- implement a small shell/comand interpreter
- exhibit some degree of multiprogramming
- interface with a hard drive for mass storage
- read from and write to a custom filesystem

- implement an interface for dynamically switching modules for various tasks
- run stably throughout its lifetime
- shutdown safely

2.1.1 Loadable modules

The following will be implemented as loadable modules, which may be switched to on-the-fly:

- CPU Scheduling [1]:
  - First Come First Served
  - Round Robin
  - Shortest Job First
  - Shortest Remaining Time First
  - Priority Scheduling (preemptive and non-preemptive)
  - Lottery Scheduling
- Disk Scheduling [2]:
  - First Come First Served
  - Shortest Seek Time First
  - SCAN and C-SCAN (elevator algorithm)
  - LOOK and C-LOOK
- Interprocess Communication
  - Message passing
  - Shared memory
- Filesystem
  - persistent
  - load-on-request

2.2 Stretch goals

Some stretch goals which would not be entirely necessary for the success of the project, but should be implemented to show understanding of more complex structures, would be primarily some more intricate scheduling algorithms, including the following:

- Multilevel Queue and Multilevel Feedback Queue
- Completely Fair Scheduler
- $O(n)$ Scheduler
- $O(1)$ Scheduler
- Staircase Deadline Scheduler
- Multiple Queue Skiplist Scheduler, MuQSS (a reimplementation of Con Kolivas' Brain Fuck Scheduler)

In order to give the operating system more purpose and to increase usability, a simple text editor would be useful and should be implemented. Looking ahead much further into areas for development, it would be useful to implement a C compiler, so that the system is fully functional and on par with modern operating systems.

2.3 Further extensions

Beyond this, the project could implement more or all of the complex CPU schedulers from the list above. Any further extensions would aim to increase the operating system's usability, and bring it more in line to what we expect from an operating system. As the system will be built with modularity as a key focus, this should aid in the development of additional functionality, and leave the option open for features such as networking and security. The timespan for these additions is likely to extend past the project deadline, however.

3 Methodology

The methodology best suited to the project will be a mix between a plan-driven and agile approach; the basic requirements of the system will not change over the course of the project, and furthermore there will be a rigid structure with regards to dependencies that the project is likely to abide by (for example, the system will have to boot before implementing memory management before implementing scheduling algorithms). Therefore, the early stages of the project will benefit from a plan driven approach, most likely an Incremental one to allow for some choice in what to implement, as opposed to the more restrictive structure of a Waterfall methodology. After the foundations have been implemented successfully, the project is likely to open up and take a more agile approach; Scrum cycles are likely to be useful dedicating a large portion of concentration implementing one feature, or fixing specific bugs, at one time, in an incremental manner.

Throughout the project, weekly meetings will be held with the supervisor in order to discuss any current problems and talk through approaches to solutions (especially for the more complex ones), the overall progress of the project, as well as the direction in which it is headed. It would also be at this time that progress is compared with the timetable, and any notes and adjustments are made dynamically in order to fully stay on top of the work.

4 Testing

The project will be tested in an incremental manner. Especially to begin with, it is vital that some systems operate correctly before moving on and developing other areas. As the project progresses and its complexity increases, unit tests will be written to systematically cover all, or at least most, likely paths of execution, and to account for each of these. The most fundamental requirement to fulfill while testing the solution will be stability, that is to say, whether the system is able to safely switch between different modules and continue operation. Of course, the solution must also be correct: the user must be able to switch dynamically between the different modules, and the system must react accordingly. There must be a way to verify that the system is indeed operating in the way that is expected from the user, and again, unit tests and verification software must be produced to ensure this.

5 Timetable

6 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 [9]
- GCC cross compiler for ARM EABI - for cross compiling for the target processor, the Cortex-A7 [10]
- QEMU - for emulating the Pi to allow quicker and safer testing [11]
- Make - used to speed up the build process

7 Resources

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

- Cortex-A7 MPCore Technical Reference Manual
- ARM Cortex-A Series Programmer's Guide
- Broadcom BCM2835 ARM Peripherals Manual

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

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, ch. 12: Mass Storage Structure. Wiley, 9 ed., 2014.

[3] . Page accessed: 2018-08-30.

[4] . Page accessed: 2018-08-30.

[5] . Page accessed: 2018-08-30.

[6] . Page accessed: 2018-08-30.

[7] . Page accessed: 2018-08-30.

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

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

[10] "Why do i need a cross compiler?." `https://wiki.osdev.org/Why_do_I_need_a_Cross_Compiler%3F`. Page accessed: 2018-08-30.

[11] `https://www.qemu.org/`. Page accessed: 2018-08-30.