A moduler kernel for the Raspberry Pi: Project Specification

October 3, 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 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.

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, similar to Linux’s `insmod` and `rmmod` [1], and must do so safely and stably. 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 both of these problems 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 means it must be able to boot from the Pi’s SD card using some bootloader, as well as taking input from a keyboard connected via USB, and printing output to a physical screen via the HDMI port on the Pi.

Loadable modules

The project must implement the following as loadable modules, which may be switched to on-the-fly 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

- Disk Scheduling:
  - 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

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 [2, 3, 4, 5, 6]:

- 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 relatively decent performance, other algorithms could be explored and implemented as modules. These may include: First-In-First-Out (for its poor performance), the Clock Page Replacement algorithm, and the Least Recently Used algorithm [7].

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. It is, however, highly unlikely that this will be achieved given the time-frame of the project, but would make for a nice goal much further into the lifetime of the operating system.

Out-of-scope

There are several features that will not be implemented in the project. These include any form of graphical interfaces, networking, and security, simply due to the complexity they would add and the extra investment required to complete them.

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.

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.

Timetable

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 [8]
- GCC cross compiler for ARM EABI - for cross compiling for the target processor, the Cortex-A7 [9]
- QEMU - for emulating the Pi to allow quicker and safer testing [10]
- Make - used to speed up 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:

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

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] `modprobe://linux.die.net/man/8/insmod`. Page accessed: 2018-10-03.

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

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

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

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

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

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

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

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

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