OS from Scratch for Raspberry Pi

Senior Project Report

Ran (Elaine) Ang

B.S. in Computer Science

New York University Shanghai

Supervisor: Olivier Marin

May 16th, 2017

1. Overview

Raspberry Pi is a small single-board computer. As with all other computers, it needs an Operating System on which all other user-space applications could run on. There are numerous existing OSes for Raspberry Pi, of which the most common ones are: Raspbian, RISC OS, LibreELEC, etc. Having customized OSes on a Pi for various purposes is a popular choice—people use it for setting up home media center, arcade cabinet, dedicated server, and even supercomputer cluster [1]. Most of these use a variant of Linux as the OS. However, as excellent as Linux might be, its code base is rather large and thus makes it a less desired example for a beginner to learn basic OS principles. Since the hardware architecture of Raspberry Pi is rather basic, developing an OS on top of it is both educative and doable.

The purpose of this project is to do bare-metal programming that simulates basic OS functionalities of on a Raspberry Pi 2 Model B. The initial reason for choosing this model is that Pi 2 has significant performance increase compared with Pi 1, and has more information available online compared with Pi 3 that just came out last year. This project is meant to explain what a general purpose OS typically does, familiarize myself with what is happening under the hood of a running OS, and help me gain hands-on experience of developing an OS that could run on a piece of real hardware.

Complete source code for this project is on GitHub [2]. Currently, it: 1) boots on Raspberry Pi 2 board, 2) has three 'processes' running, each controlled by an incrementing counter and taking turns outputs a character which represents the content of that 'process', and 3) uses a UART LCD screen for displaying the output information.

The rest of this report is organized as following: Section 2 gives detailed information about this project's hardware and software requirement; Section 3 defines the scope of this project, which, combined with Section 2, outlines what are the parts that I need to write and what are the parts that I could just download; Section 4 briefly summarizes existing relevant work; Section 5 explains implementation details along with a bit of history regarding the initial work plan and refined work process; Section 6 reflects on the overall project by providing critical analysis; Section 7 concludes and gives directions for future improvement.

2. Project Setup

2.1 Hardware

Raspberry Pi 2 Version B (with a SD card) [3]

- Broadcom BCM2836 System on Chip (SOC): A Cortex-A7 CPU, VideoCore IV GPU, and SDRAM. CPU and GPU share the peripheral buses and some interrupt resources.
- 1GB physical memory (shared with GPU)
- 40 GPIO Pins, 4 USB Ports.

Note:

- A System on Chip (SOC) is a microchip with all the necessary electronic circuits and parts for a given system, such as a wearable computer, on a single integrated circuit.
- There is currently no official document for BCM2836 Peripherals. I mainly referenced BCM2835 Peripherals documentation [4] for the usage of GPIO pins. Major changes with BCM2836 that interest us are: 1) it uses I/O Peripherals base address 0x3F000000 (physical address seeing by ARM core) instead of the previous 0x20000000, and 2) its UART clock is about 50 MHz (by experimental deduction).
- Cortex-A7 has the ARMv7 architecture, which uses the 32-bit instruction set. Its complete reference manual could be found online [5]. I mainly reference this manual for ARM specific register accessing, such as during setting up a page table.

Serial Liquid Crystal Display (LCD) Module [6]

- Supports UART interface, runs at 5.0V, and has a baud rate of 9600bps.

2.2 Software

Workstation computer

- Ubuntu 14.04 virtual machine running on VMware.

Note:

- This machine is where I write and compile the code for the target—Raspberry Pi.
- Technically any computer with a major OS (Windows/Mac OS X/Linux) would suffice, as long as the according GNU Tool Chain is chosen properly.

GNU ARM Tool Chain for Cross Compiling

- Version used for this project: 6-2017-q1-update [7], download and configure
- Or install using: *sudo apt-get install gcc-arm-none-eabi gdb-multiarch*

Note:

- This is an essential tool for OS development if not developing on the target machine. It allows compiling executable files for one architecture on a different one. In my setup, for example, it allows me to compile executable files for ARMv7 on my Ubuntu host
- The prefix for GCC when using this tool chain is *arm-none-eabi*

QEMU and GDB

- QEMU version 2.8.95, build from source [8] using configure:
  --target-list=arm-softmmu,arm-linux-user,armeb-linux-user --enable-sdl
- GDB version 7.6.50, installed with *sudo apt-get install gdb-arm-none-eabi*

Note:

- QEMU is a hardware emulator that provides, in this case, a virtual Raspberry Pi board. Latest QEMU has build in support for Raspberry Pi 2 (raspi2). After successful build, check QEMU support machine types using *qemu-system-arm -machine help*
- GDB provides a convenient tool for kernel debugging, allowing us to see the execution status of our kernel. Essentially, in this case, QEMU will act as a remote debugging target for GDB. Detailed setup is in the Makefile under the project's source repo.

Raspberry Pi Firmware [9]

Note:

- This is provided by the Raspberry Pi foundation. Files that interests us are the GPU firmware and bootloaders, namely: *bootcode.bin*, and *start.elf*.
- Raspberry Pi boot process:

  1) Raspberry Pi's booting starts from GPU when power on. GPU executes the first stage bootloader that is pre-written in ROM.

  2) The first stage bootloader loads the 2nd stage bootloader (*bootcode.bin*) from SD card, loads it to L2 cache and executes it.

  3) 2nd stage bootloader enables SDRAM, loads the 3rd stage bootloader into RAM, and executes it. On Pi 2 and later, this is combined with step 2) done only by *bootcode.bin*.

  4) 3rd stage bootloader reads the GPU firmware (*start.elf*), which reads system configuration parameters (*config.txt*), loads a kernel image (*kernel.img/kernel7.img*) based on different versions of Pi) that reads kernel parameters (*cmdline.txt*), and release the control to ARM CPU. Before the loading of kernel image, everything runs on GPU.

3. Project Scope

With all aforementioned setup in Section 2 done, I am ready to write my own code that composes a kernel. A minimum kernel contains three types of files:

- A linker script for linking various object files into the final executable kernel
- ARM Assembly that setup hardware environment and branch to some *main* function
- C code for OS routine and necessary header files.

These files will be compiled into a single binary called *kernel7.img*, which could be copied to the SD card along with *bootcode.bin* and *start.elf*, and prepares the Pi for power on and boot (Figure 3.1). A LCD is hardwired with Raspberry Pi's GPIO pin 2 for 5V power, pin 14 for UART RX, and pin 39 for ground for displaying information (Figure 3.2).

It's worth noting that all the C code here is written assuming the absence of Standard C Library (libc). This assumption is intuitive during an OS development, since the lowest level of a libc function is usually a POXIS system call. For example, libc function *printf()* calls *write()* which a standard kernel such as Linux will implement to output to an I/O peripheral. Because I am writing our own kernel, there is no such free-lunch *write()*, and I thus need to implement my own low-level I/O methods based on the type of I/O devices I have. In fact, I need my own implementation for every single function of this sort for our kernel, which does correspond to the basic functionality of an OS: managing hardware resources and providing a higher-level abstraction to whatever code running above it.

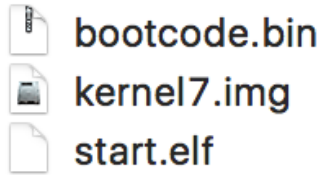

Figure 3.1 Files on SD card that prepares Pi 2 for boot

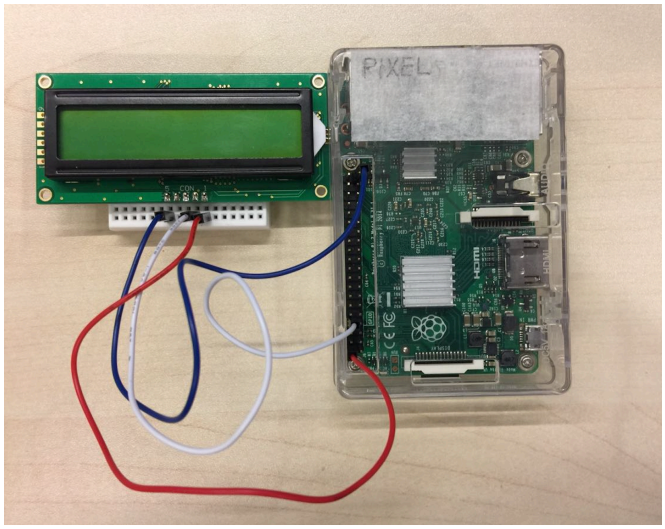

Figure 3.2 LCD with Raspberry Pi board wiring

4. Relevant Work

4.1 Popular Existing OSes for Raspberry Pi

Raspbian

This is the official well-rounded Linux-based OS for Raspberry Pi. It comes with various pre-installed software for general uses, such as Python, Java, Mathematica, etc.

Running Raspbian on a Raspberry Pi is a lot like running a Linux on your laptop. The major difference being that: 1) the underline CPU hardware architecture might be different, and 2) the booting process is different as outlined before.

LibreELEC

An open source, Linux-based "just enough OS" for KODI, an award winning open source software media player. OSes of this sort extends what a Linux kernel originally offered, with tailoring for specific purposes, and in this case, providing necessary platforms to turn the Raspberry Pi into a media center. The Videocore GPU is the major reason why a Raspberry Pi board makes such a good media center.

RISC OS

This is an OS specifically designed for ARM microprocessors. It has a Hardware Abstraction Layer that provides a level of dependency from varies ARM hardware platform. It has a kernel that is responsible for the overall control of OS, and system extension modules provide extra functionalities. It was designed to provide high performance even on slow ARM processors, by writing key parts of the OS in ARM assembly language. Such implementation choice makes it a less desirable example for studying kernel design because it does not scale.

Notice:

The above OSes are available for installation using NOOBS with Internet connection. NOOBS (New Out Of Box Software) is an OS installation manager for Raspberry Pi. After downloading, unzip and put all files in NOOBS folder on a formatted SD card, insert into Raspberry Pi. Then boot the Pi and select the OSes for a GUI-Oriented installation.

4.2 Relevant OSes for fun or for education purposes

Xv6 [10]

This is an open source Unix-like OS with less than 9,000 lines of a mix of C and assembly code for x86. It provides the basic interface introduced by AT&T Unix Operating System. Unix is designed to provide a narrow interface whose mechanisms combine well, and has been adopted by most major OSes nowadays including Linux, BSD, Mac OS X and etc. Understanding Xv6 is a good starting point for understanding many other OSes.

JOS [11]

This is an Exokernel [3] style OS skeleton. It was initially designed to run on x86 architecture, which has different booting processes, peripherals, and compiler requirements from ARM. The philosophy of an Exokernel such as JOS is to separate protection from management. More specifically, the kernel only securely exports hardware resources through a low-level interface to untrusted library OSes, which perform other useful functions and policies.

Arunos [12]

This is an open source OS designed for ARM architecture, which also takes references from JOS. The author claimed that it could run on a Raspberry Pi 1 board. The functionality of this OS is rather limited, and does not correspond to the JOS design principle. I plan to study, but not fully adopt this OS. The purpose of my project and his might be approximately the same, but the approach and thus the core code would be different.

JOS-on-ARM [13]

This is an open source project. The author transplanted JOS to ARM and tested it out with QEMU. Based on the compilation flags, this project targets Raspberry Pi 1. The code with this project is not very clean, with lots of remaining code from x86 JOS that is unused.

Mini-ARM-OS [14]

This is an open source project that aims at building a minimal multi-tasking OS kernel for ARM from scratch. The code is not designed for ARM Cortex-M, not to run on Raspberry Pi, but it worth taking a look at how it directly interacts with memory for various purposes.

4.3 Raspberry Pi Bare Metal (NOT AN OS)

Baking Pi [15]

This is a classic Raspberry Pi bare metal course from Cambridge, which is initially written for Raspberry Pi 1 model B. Code for this course are basically all in ARM Assembly. Some of the code does not work directly with Raspberry Pi 2 and requires adjustments as specified in Section 2. This course includes topics such as blinking LED, drawing on the screen, and receiving inputs using keyboard.

ARM-Tutorial-RPI [16]

This is an open source project with detailed explanation of how to get low-level C code cross-compiled and running on Raspberry Pi without an OS. This project explores examples such as coding a blinking LED, C Library Linking, Interrupt Handling, Raspberry Pi VideoCore GPU and Mailboxes. Although these alone won't form a complete OS, they comprises useful explanations of the use of C code for low-level hardware interaction.

5. Implementation

5.1 Initial Work Plan

1) Setup the Ubuntu environment. It would be a virtual machine running on VMware that I could SSH into from my Mac OS X terminal.
2) Get the GNU tool chain, QEMU, and arm-gdb. Configure these tools. Tested out how to use QEMU at command line, and especially how to use it with GDB. Understand the booting process of Raspberry Pi. Test a few existing OSes for booting.
3) Go through the Baking Pi [15] tutorial to study how to write and compile bare metal programs on Raspberry Pi. Also use this series to test if tools in step 1 are correctly setup.
4) Go through examples in the ARM-Tutorial [16], making sure that writing, compiling, and linking a reasonable size C project is possible for Raspberry Pi.
5) Run Arunos [12]. Build Arunos from source and try booting it on Raspberry Pi. If the boot succeeds, study how Arunos sets up necessary environments for the kernel.
6) Study basic structures of Xv6 [10] and JOS [11]. Run JOS with QEMU. Modify the skeleton of JOS according to Arunos, so that it could be compiled and run on ARM architecture. After done with necessary modification, test it out with QEMU and boot it on Raspberry pi.
7) Implement missing components in JOS skeleton. Including: Virtual Memory; Process Structure; Preemptive multitasking; (probably) File System and Network Driver. There are detailed instructions for implementing these components on MIT 6.828 course webpage. Make sure these components work together on the Raspberry Pi hardware.
8) Testing. Add extra fun features if time permitted.

(In fact, this plan is rather ideal, and I present it here only for reference purpose.)

5.2 Actual Work Progress - A Devious Tour

So this is when the ideal world and the real one part…

I successfully went through steps 0-3 in Section 5.1, setting up necessary hardware and software, following the Baking Pi tutorial to get the ACK LED on Pi to blink a Morse Code pattern based on an input binary String, and compiling C source code for running on a Pi. I also tested installing and booting Raspbian and RISC OS.

After going through the Baking Pi tutorial, I had a better sense of ARM Assembly, which is unavoidable during OS development. Apart from setting up the initial entry point, there are situations where I need hardware support for software execution, such as saving register status during context switch, or using atomic instruction during the implementation of a lock.

However, I failed at attempting to run Arunos. It does not boot on the Pi hardware or QEMU machine raspi2. It does, however, run on QEMU with *arch=versatilepb*, but it does not provide the functionalities it claims to have. I contacted its author Hadi regarding this issue, and Hadi explained to me that it used to work at certain point when the Arunos shell is

part of the kernel, but stopped working when he try to separate the shell as a user level process. He did not finish the change at that point and leave the project at a stale point. Frankly, Arunos does not provide a good starting point, as it lacks comments or documentation for the implementation details. I spent sometime digging and modifying its source code but still cannot get it working even with the emulator. Despite this, the setup code for Arunos still worth referencing and studying, as it provides a relatively complete example of how OS handles I/O peripherals, ARM virtual memory, interrupts, etc.

I temporarily set Arunos aside realizing that I am unable to fix it given the semester's time constraint, and went about studying the structure of Xv6. This OS is well documented and gives practical solutions to setting up hardware support for x86 such as paging and interrupt, as well as software abstractions such as processes and stacks. Although the actual code varies drastically on different architectures, these setups for x86 do act as a good analogy for similar steps in ARM, and thus provides me a concrete starting point for further reasoning the Arunos code if time permitted.

While continuing studying Xv6, I branched out looking for a suitable display for showing the internal state of the Pi instead of leaving it running at a "headless" mode. I chose the UART LCD screen because it is the simplest possible I/O device, and I am more familiar with the UART interface after going through available online tutorials for Pi minimum kernel development [17].

By the time this report is written, I had a proof-of-concept OS for Raspberry Pi coded up and tested. It could run with both the emulator machine raspi2 and the actual Raspberry Pi 2 hardware. Details of this prototype will be explained in Section 5.3.

5.3 Current Prototype Details

Features

- LCD shows a welcome message "Hello World …from kernel" upon powering up the Pi.
- Three 'processes' take turns executing, each outputting a character representing its running status and its current content choosing from a different ASCII range, in the format of "pid: content". For process 1 it outputs looping character from a-j, for process 2 from A-J, and for process 3 from 0-9.
- Each process has 4 'clock cycles' that allows it to output 4 characters, after which it saves its current state and yield execution to a different process.
- Each process has pre-defined total running time, after which it finishes running and outputs a message to the screen noting its finishing status.
- After all process finishes running, the screen shows a message "All done", and the kernel enters an infinite spinning loop doing nothing. (Figure 5.3.1)

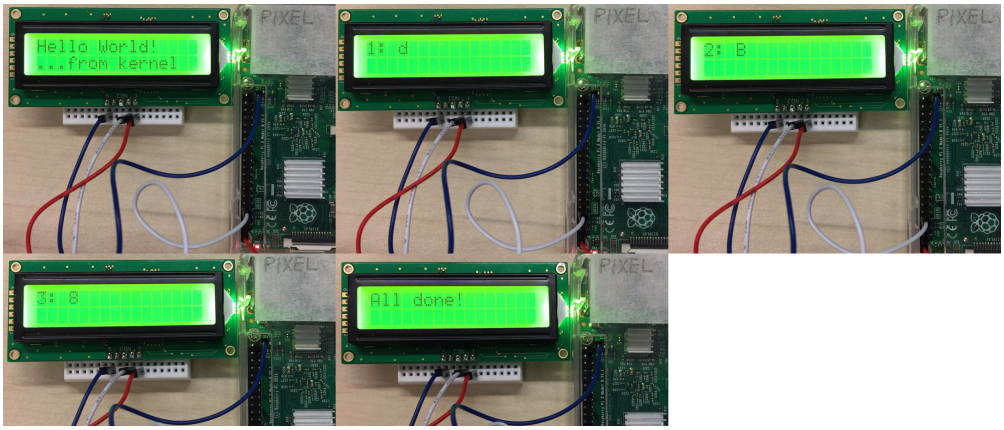

Figure 5.3.1 Visualization for various state of execution

Design Explanation

- For communication via UART, the kernel does necessary initialization of the SOC UART environment and reads from or writes to physical address 0x3F201000, which is the MMIO address for UART0 data register. Detailed initialization process could be found in the *kernel/uart.c* source code and comments. Roughly speaking, the important steps are: 1) it sets up GPIO pin 14 for transmit information from Pi to another UART device, 2) sets up the correct baud rate for communication using an Integer (IBRD) and a Fraction (FBRD) register, and 3) enabling FIFO buffer as well as 8-bit data transmission. Usages of these registers are documented in the BCM2835 Peripherals [4].
- Communicating with the LCD is done via sending Hex command through UART interface. A complete list of command could be found in the LCD specs [6].
- A process here is a C struct type comprises of the following fields: process id (pid), overall running time (exp), current running time (cur), its content character (ctnt), its initial value for restoring the looping character (init_c), a trivial counter for this simulation purpose (cc), a byte determine if the process has finished (fin), and a byte determine if the process should be running at the current moment (run). Note that I use a byte for 'fin' and 'run' only for simplicity. They could only use a bit if for optimization purpose.
- Scheduling here is done in a Round-Robin fashion. Each process gets to run for 4 'clock cycles', controlled by an incrementing counter. Thus, each process prints exactly 4 characters before yielding execution to a different one waiting to be executed. The simulation here totally ignores the division between IO and CPU, and it is by no means what is done in an actual OS scheduler, but it gives the basic idea alone with some visualization for time slice of a single CPU in the presence of multiple processes.
- There is no dynamic memory allocation happening here. All variables are statically allocated and referenced either by their variable name, or by an explicit pointer. Since there is no virtual memory setting up beforehand, all the addresses are physical addresses visible to the ARM core.

6. Reflection

Do thorough research before choosing a topic

I chose Raspberry Pi as the hardware base for this project without doing much background research, and this turned out to be a really bad idea. While it might be true that comparing two popular architectures and modifying kernel code written for one to make it run on another is quite fun, the lacking of proper documentation for Raspberry Pi hobby OS development could leads to enormous trouble, especially for a beginner. ARM might has a simpler instruction set, but there are simply more examples and online tutorials for hobby OS development on x86. The fact that Raspberry Pi has a close-source bootloader and stoppes release official documentation for its SOC does pull up an alert: they might have shifted the direction of their product to providing better compatibility on a system like Raspbian rather than for open-source OS development hobbyist. I would have done a better job at researching and chosen an x86 board such as Arduino101 had I have a chance to start this project all over again. That is not to say transplanting an x86 kernel to ARM is not fun, it just requires more background knowledge from the developer and is not beginner-friendly.

High-level simulation is good for understanding the logic not the real world

I used to take a full OS course where we do high-level simulation of various OS algorithms [18]. While it does a decent job introducing basic OS logics, the hiding of interaction with real hardware and actual kernel implementation details do not provide me with a solid understanding of how an OS does what it claims to do. Concepts such as linking, paging, system calls, isolation of processes all seem rather abstract to me even though I know there are actual code implementing them somewhere underneath. Not having a concrete understanding of these implementations makes it nearly impossible for me to further reasoning a design choice of an OS or any computer system in general. This is why this project exists: I want to start from the simplest possible design of a real-world computer system, gradually build on top of it and eventually have a fully functioning OS build on my own, in the hope that such process could provide me with a thorough understanding of how various system stacks interacts with each other. At the same time, it may allow me to quickly understand other systems even with different hardware architecture, spot design flaws of existing systems and thus build better ones in the future.

Holding this thought and taking a look at my current design. These implemented features outlined in Section 5.3, though give a flavor of OS scheduling and simple process management, stills, in my opinion, should be classified as bare-metal programming and do not reflect what a general purpose OS does. For example, I used a self-defined counter variable for simulating the running time of a process. This is essentially incorrect since the incrementing rate of that counter depends on the execution states of the current running process. I could have done a better job using a system timer that incrementing on its own,

and I did have that counter working using Assembly in the Baking Pi LED example, but it somehow fails with the C code I wrote, and I am still trying to figure it out.

In the attempt of doing more than just simulation, I also tried setting up virtual memory and static page tables for translating virtual addresses to physical ones leveraging the ARM MMU. I took references from Arunos and found out that in order to enable paging on ARM, there are three steps: 1) Set domain access control by writing a permission bit to register DACR; 2) set translation table base by writing the base address of a correctly formatted page or section table to register TTBR0; 3) set the lowest bit in the system control register (SCTLR) which enables the virtual to physical address translating. After these steps, all kernel operations could only see virtual address, and the MMU hardware does the translation for every memory access. I am still coding this up and have not finished by the time this report is written.

Future Directions

I do want to finish my initial work plan from Section 5.1 and have the full JOS transplanted and boot on a Raspberry Pi in the future. It's worth noting that JOS has a different virtual address layout compared with Arunos since JOS adopts the Exokernel design. A rough work plan for near-term future work would be: 1) finishing memory management; 2) setting up interrupt and exception handling; 3) implements context switch and fork; 4) adopting an SD card driver and developing a file system; 5) have a shell.

Another interesting piece to look at is the Mailbox service that provides a communication between the ARM and VideoCore firmware running on GPU for actual graphics displaying.

7. Acknowledgement

I want to thank Prof. Olivier Marin for proposing this fascinating topic and good-naturedly bearing with me throughout this semester. Thanks to Prof. Scott Fitzgerald for providing plenty of useful information regarding hardware. Thanks to Prof. Romain Corcolle for being patient and helping me setup UART LCD correctly. Thanks to Mr. Hadi Moshayedi for his Arunos and being responsive to my emails.

8. Reference

[1] Steps to make Raspberry pi Supercomputer:
http://www.southampton.ac.uk/~sjc/raspberrypi/pi_supercomputer_southampton.htm

[2] Project source: https://github.com/ElaineAng/eos_rpi/tree/master/minimal-kernel

[3] Raspberry Pi 2 Model B Specs:
https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[4] BCM2835-ARM Peripherals:
https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

[5] ARMv7 Architecture Reference Manual:
http://liris.cnrs.fr/~mmrissa/lib/exe/fetch.php?media=armv7-a-r-manual.pdf

[6] Serial UART LCD Reference:
http://rl-display.com/ajax/Download.ashx?type=1&ID=90071ff9-1957-4330-9e1d-170db48ef253

[7] GNU ARM Embedded Toolchain:
https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

[8] QEMU source repo: https://github.com/qemu/qemu

[9] Raspberry Pi firmware: https://github.com/raspberrypi/firmware/tree/master/boot

[10] Xv6: https://pdos.csail.mit.edu/6.828/2016/xv6.html

[11] JOS on the MIT Git repo: https://pdos.csail.mit.edu/6.828/2016/jos.git/

[12] Arunos Source code: https://github.com/pykello/arunos

[13] JOS on ARM: https://github.com/GrayKing/JOS-ARM

[14] Mini ARM OS: https://github.com/embedded2015/mini-arm-os

[15] Baking Pi: http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html

[16] ARM Tutorial for Raspberry Pi: https://github.com/BrianSidebotham/arm-tutorial-rpi

[17] Raspberry Pi Bare Bones: http://wiki.osdev.org/Raspberry_Pi_Bare_Bones

[18] NYU CS202 Fall 2015: https://github.com/ElaineAng/CS202