Annotated Bibliography

Matyukevich, Sergey. “Learning Operating System Development Using Linux Kernel and Raspberry Pi.” *Raspberry-Pi-Os*, 15 May 2017, https://s-matyukevich.github.io/raspberry-pi-os/.

Matyukevich’s guide provides a step by step guide to developing an operating system for the raspberry pi. It includes details like how to download to the pi and how to use system registers.

It has proven to be an excellent guide with a lot of in depth explanation of how the arm architecture is working and what is necessary to interface with it. It currently provides explanations of how to launch the kernel image, how to change your exception level and handle interrupts, and how to manage the page table’s virtual memory.

For my project, this guide served as a perfect resource for collecting a lot of the requirements I’d need to implement before adding each step. It also pointed me to other resources such as the ARM peripherals data sheet. In general I only used this as a guide to explain parts of the code and find links to those other resources. It has been a great place to look up issues as I implement each step by myself.

Broadcom Europe Ltd. “BCM2837-ARM-Peripherals.” 6 Feb. 2012.

BCM2837 ARM Peripherals is an in depth data sheet which I found linked in Matyukevich’s guide to operating systems. It contains an extensive list of all the physical peripherals of the raspberry pi: GPIO (digital pins), their controlling address spaces, and hardware interrupts.

The data sheet has pretty much all the information that a developer could need in great detail. However, because it is in data sheet format, it is difficult resource to comb through. In each

section there is an in depth explanation of the address space including what addresses talk to which peripherals and even what bits to use for performing a certain task.

This data sheet served as my second layer of research. Matyukevich's guide helped to collect relevant information into one place, but it often left out details that I'd end up wanting to know about. I would then go to this data sheet with an understanding of what I was looking for. I mainly used it while setting up the UART communication and as I began to look at the physical memory layout to prepare for page tables.

Timeline of Completion this Quarter

I spent the first week putting together the necessary compilers and assembly code to simply turn on an LED. The next two weeks I spent trying to launch a c program from that assembly code. Here I ran into some issues mainly when it came to the stack. Before I could jump to the c program I had to setup its stack. In the fourth week I made it my goal to start exploring the raspberry pi and understand more of what it's state was. So I made a c program that would blink binary numbers through the LED. This told gave me information like the location of my code, the exact address of the stack pointer, and even that in the 32 bit mode I was operating in, the stack pointer seemed to increase instead of the usual decreasing stack pointer.

The following two weeks I spent getting a more expressive communication with the PI by setting up the UART pins. This is a simple to way communication that sends binary pulses down either of one wire (one wire goes to my computer the other goes to the PI). I used this channel to send text to my computer screen and that served as my display.

In the seventh week, I used my more in depth communication to explore more of the address space. This ended up not being incredibly useful. But I was able to tell that my code was being loaded at address x8000 just as described in my linker script. Apart from that found two chunks of memory that no matter how much research I did, I could not figure out what they were for. At x20000 and x100000 onward, I found the presence of some kind of information. Everywhere else was clear.

At this point I gave up for the time trying to map out the memory space. So I moved on to trying to handle exceptions. It was at this point I started running into issues with system registers. I found that my compiler (which as stated above was currently running in 32 bit mode) would not allow me to use the system registers. The solution I found is to switch to 64 bit mode. But this meant installing a new compiler and rewriting some of the assembly code I was using as the syntax was a little different.

For the final two weeks of school, I spent time setting up the processor and its exception level. This includes disabling caching and virtual memory for the time as well as giving permissions to exception level 1 (which is what my kernel will eventually be running in). I finally was able to switch the processor to run in exception level 1 (instead of the level 3 which it starts in).

Plan for Next Quarter

The desired outcome for next quarter is a single operating system that will support a finite amount of user processes whose source code is already loaded into the kernel. I do not plan to create a driver that talks to the sd card (which serves as the raspberry pi's hard drive). I do not plan to create a program that understands how to load an ELF image of a process. I plan to just have the source code be compiled into the kernel and then setup stack and data spaces for each process.

The project will be a success if I can execute multiple programs at the same time that each are printing information to the screen at the same time. This would also require that I implement preemption and a scheduler to manage processing time between the processes. I also plan to create some level of separation between the processes with a page table. Because I am not loading in their separate images, this may not be able to be done in full, but to have their stacks and global data made virtual is a part of my goals.

To accomplish these goals, I have laid out the following schedule:

Week 1: Allocate a single stack and execute a user process.

Week 2: Have the kernel interrupt that user process.

Week 3: Create a second process and context switch between them.

At this point the two processes are probably going to have kernel level permissions and are writing directly to the UART channel. They will have to be synchronizing between themselves so they don't conflict with each other's messages.

Week 4: Create system calls for the processes to use instead of accessing UART channels and physical memory themselves.

At this point the two processes won't be isolated, but they should not have to consider each other while running.

Week 5: Setup paging and virtual memory to isolate the two processes.

Week 6: Increase the amount of processes that can run to some other fixed amount but larger than 2.

Week 7: Program the kernel to create these processes once asked by the user instead of as a hard programmed part of initialization.

Week 8: Write up the project.

Final Report Outline

1. Introduction
2. Background
    I. GPIO and UART
    II. Exceptions and Exceptions Levels.
    III. Memory Management
3. Description
    I. Explain the UART driver
    II. Explain the code that handles Exception Levels
    III. Explain context switching
    IV. Explanation of the MMU and page tables
4. Evaluation
    I. Overview and Summary of Progress
    II. Various demonstrative tests on my Operating System
5. Conclusion
    I. Future improvements
    II. What I learned
    III. Things I may have done differently