Developing a MicroKernel for Raspberry Pi Zero

Kemal Turksonmez and Parker Folkman

December 9, 2019

Introduction

A Microkernel is a minimalist kernel that abstracts away OS services to user space. By abstracting away various OS services such as the file system, CPU scheduler, and memory manager users are able customize these services for their individual applications. This modularity and extensibility is possible because the different OS services run as a server with its own address space in memory. Since these servers are partitioned into different sections in memory space the operating system obtains additional security as one service failing cannot effect other services that live in different sections in memory. These advantages are taken with additional performance cost due to having to context switch multiple times to access hardware. Operating System implementations exist for various Raspberry Pi models 2 and 3 and can be found at [2] [3], but an implementation on the Raspberry Pi Zero was not yet fully documented or attempted. In this paper we outline our efforts to build and implement a functional microkernel for the Raspberry Pi Zero.

Implementation

To first build a functional microkernel for a raspberry pi, we must be able to write it and compile it. Since we will be working on this project from our own computer, which are both x86 systems, we have to use a cross-compiler. A cross-compiler is a compiler that can compile files for one system while on another system. The cross compiler we use in this project is the arm-none-eabi compiler. The arm-none-eabi is a part of the GNU toolchain for embedded ARM processors. The specific processor we are working with is the ARM1176JZ-S, which is an ARM11 processor based on the ARMv6 architecture. For example a compiling a c program using this library will use this single command:

```
arm-none-eabi-gcc -mcpu=arm1176jzf -s
    -fpic -ffreestanding -std=gnu99                    (1)
-c kernel.c -o kernel.o -O2 -Wall -Wextra
```

In this command each individual flag plays a special role. The mcpu flag is used to identify the processor that the program is being compiled for. The fpic flag defines references for functions, variables, or symbols should be by relative address based on the current instruction as opposed to absolute address. The free-standing flag means that gcc should not depend on the standard c library and should not look for a main function as an entry point. The wall flag tells the gcc to show all warnings and the Wextra flag enables some extra warnings not covered by wall. Using this cross-compiler library and a couple extra flags, we can prepare our kernel for our raspberry pi.

To prepare the kernel itself, a boot file was written. Since the main goal was to create a microkernel, the focus was not on understanding the structure of the boot file. The boot file only needs to call the kernel when needed. It was chosen to use the boot file from the [3] tutorial. The next steps are to write the

kernel. To ensure that everything was working, the provided kernel.c file was used and adapted the file to have the right addresses for the raspberry pi zero GPIO headers. The addresses of the GPIO headers are relevant because they are used to enable the use of the MINI UART on the pi. The UART can be used to pass values from memory on the pi to another device using and is especially helpful for communicating with devices when no other communication method is available. In the kernel.c file the GPIO pins were declared and enabled against the UART to send the string "Hello World". Finally, to create a boot-able, functional kernel a linker file was used to link the boot sequence and the kernel together. This combines the files into an executable format. For this portion, the files were linked together into an "Executable and Linking Format", also known as elf format. In the [3] documentation, it was recommend to use an emulator to emulate the kernel for this portion, simply because it's easier than preparing micro-SD for a raspberry pi. The first attempt was therefore to use the QEMU emulator for the raspberry pi zero. However, it was found that there was no support for the raspberry pi zero in the QEMU library.

Several steps were taken to overcome this. First, it was realized that there existed support for the specific processor the raspberry pi zero was using. So we attempted to emulate our kernel by specifying the CPU of the system. This portion was able to run error free, but without receiving any output. Unfortunately nobody offers support for the raspberry pi zero as a machine on QEMU. Without having the raspberry pi zero supported as a machine on QEMU, there was no way to emulate the kernel and receive outputs from the OS. The [2] documentation recommends using the serial output from the raspberry pi as a way to develop the board, so this was explored as a possible IO option. To do this he recommends using the USB to TTL cable sold on the adafruit website. However, this cable was sold out and the GPIO headers were attached on the raspberry pi zero.

In order to find a way to get some IO from the kernel, the possibly tinkering with the OTG USB port provided on the raspberry pi as a means of serial output was investigated as the next option. Documentation on how this might be accomplished was found here [4]. The author of this article discusses how one could modify their raspbian OS to provide serial output over the USB OTG port. However, when this was attempted the output could not be identified as serial output. Whenever it was attempted to use a terminal emulator to the screen the connection would receive a bunch of errors. Additionally the author was taking advantage of the raspbian OS to provide the serial output. Unlike the author, a new OS was being written for this project, and access to raspbian would be lost. Looking at the [3] documentation, it was noticed that there was a small snippet written about printing to a real screen. In this tutorial, the author quickly summarizes the importance of using a frame buffer from the GPU in order to print through the HDMI output. The author goes on to provide code for both the raspberry pi 1 and the raspberry pi 2. However, this documentation is also a build up of all the other code written in the previous docs before this one and there was no way of testing this functionality. Finally, it was decided to use the GPIO headers and a TTL to USB cable to attempt to receive UART output from the raspberry pi zero. Fortunately the raspberry pi kit had a spare set of header pins that could be soldered onto the board. However, there was no way of getting a TTL to USB cable in time to finish the project. Adafruit had sold out of their console cable and there was no way to order one through amazon and have it arrive in time to finish this project. The best option was to make our own cable.

In order to make our own cable we went to Radio shack and picked up a soldering iron, jumper cables, a multi-meter, and a DB9 to USB serial cable. Using these materials we were able to solder the GPIO headers on to the raspberry pi and convert the DB9 serial cable into a TTL. Next the sd-card must have an MBR partition table with a FAT32 boot partition. The sd-card must also contained two files "bootcode.bin" and "start.elf", which can both be found on raspbians github page. A raspbian boot was created and all the other files were removed. The Bootcode.bin and start.elf files must be included due to the way a raspberry pi must boot. Bootcode.bin controls the boot process of the GPU and start.elf handles the transition from the GPU to the kernel. After formatting the sd-card, we could finally cross-compile our kernel and place it in the boot. The last step after creating the elf format was to copy the binary version of that file into a file called kernel.img. So raspberry pi's either run kernel.img, kernel7.img, or kernel8.img. The version

is based on the processor being used and is selected by start.elf. This is why raspbians boot image comes with three different kernels. Since the board for this project was the rapsberry pi zero, a kernel.img file was created. Finally, after doing all of this the raspberry pi was connected to the PC and an emulator was used to interpret the output coming from the raspberry pi. Unfortunately, only received garbage characters were received. The output can be seen in Figure 2.

Unfortunately the cause of this garbage output was ultimately not determined. The driver and the terminal emulator were verified to have the same baud rates. The data sheet for the serial cable was explored to make sure that the cable could handle the right voltage ranges from the raspberry pi. These values were found from raspberry pi programs written by a bare metal programmer [5] for the raspberry pi zero. One of the programs he wrote was specifically for UART output and those programs also produced garbage output. Unfortunately, it is unknown whether the programs were written incorrectly or if there was something wrong with the driver. It is certain that the serial cable received some form of output. Receiving meaningful IO was found to be very challenging for a project like this with limited to no documentation.

Challenges

Our biggest mistake with this project was approaching it as a software project. When planning we only thought about what we would write and how we would go about designing the OS. We never once stopped and considered how we would go about communicating with the OS. This also led to the most important lesson that we learned: building an OS is a hardware project, not a software one. As discussed in our implementation section, all of our challenges was centered around testing. Since we had no I/O there was no way for us to know whether or not our OS was functioning. Since one of the tutorials we were following used an emulator to provide a viable output [1], we thought we could do the same thing. As mentioned, we attempted to overcome our I/O problem by creating our own serial TTL cable. This can be seen in Figure 1. Another interesting challenge we faced was downloading the drivers for the cable. After creating the cable, we first attempted to download the prolific driver on a Macbook. Unfortunately, this download caused the machine to crash which further delayed progress as this machine contained the cross-compiler that was being used to prepare the kernel image.

Conclusion

Overall, we were pretty upset that we were not able to develop our OS due to I/O issues. However, there are so many things that we can and will takeaway from this project. The most important thing being the realization that the OS is more about hardware than software. Alongside this, we learned about the boot procedure of an raspberry pi and all that goes into creating a bootable kernel. We learned about a frame buffer and how one could potentially go about accessing one through the GPU. We learned about cross-compilers and the role they play in OS development. And we also learned about how one goes about receiving serial output from a raspberry pi.

Even though we did not get the exact outcome we were expecting from this project. We are still happy to have learned so much about the OS and all things that go into it. We were even able to boot the kernel. We just couldn't see what it was printing through the serial output. Moving forward, we will definitely continue development of this project. Now that we know what goes into OS design, we will be better prepared for development. Afterwards, the first thing we will design for the OS will be memory allocation on the raspberry pi. Having attempted this project, we can say that OS development is challenging and intimidating at times; but even if nothing goes your way, you can still walk away knowing a lot more than what you started with.

Included Files

In the folder we uploaded, we also included a video, and two separate kernels we used for testing purposes. The video SerialOutput.MOV shows some of the output we were recieving from the raspberry pi zero after booting. The jsandlerTutorial file contains a boot section, kernel, and linker file. This was the original file we used for testing purposes on the raspberry pi kernel and was going to be the base foundation of our kernel. The other folder, dwelchBareMetal includes a bare metal program specifically written for the raspberry pi zero for UART output. We also used this file to test our serial cable. Both folders contain a readme with instructions on how to correctly cross-compile the files.

References

[1] URL `https://wiki.osdev.org/Raspberry_Pi_Bare_Bones`.

[2] S. Matyukevich. URL `https://github.com/s-matyukevich/raspberry-pi-os`.

[3] J. Sandler. URL `https://jsandler18.github.io/tutorial/dev-env.html`.

[4] D. Tomaschik. URL `https://systemoverlord.com/2017/05/21/pi-zero-as-a-serial-gadget.html`.

[5] D. Welch. URL `https://github.com/dwelch67/raspberrypi`.

Appendices

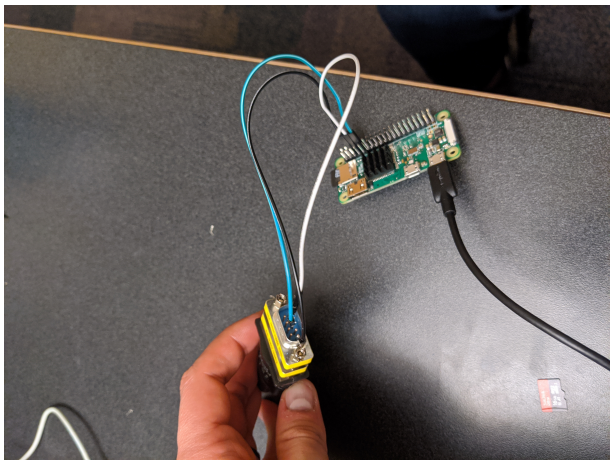

Figure 1: An image of the soldered serial connector cable

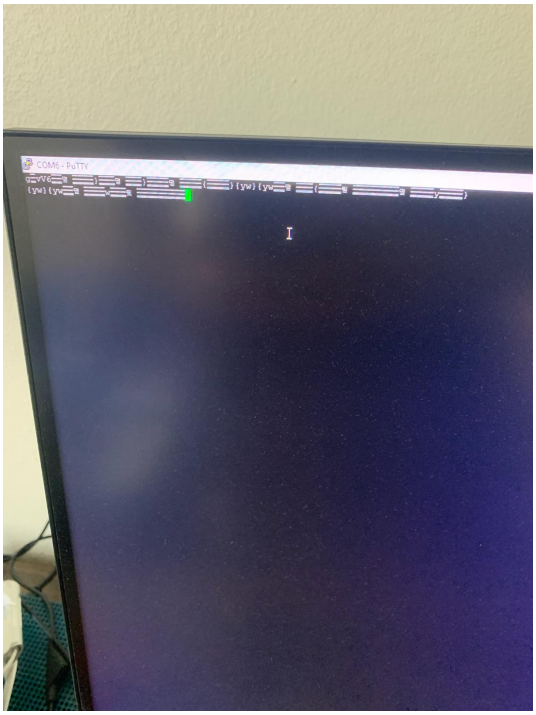

Figure 2: An image of the soldered serial connector cable