Journal Entries

Nathaniel S Flores

January 4th, 2021

List of Journal Entries

Note: The following are entries in chronological order from when I started to the end of the project.

An Introduction

Need-to-know vocabulary:
CPU: the processor on the computer
OS: Operating system
Kernel: a program that runs at all times and restricts memory, CPU and hardware access for programs to keep them from destroying each other.
Debugger: a program that reads code and prints out errors it detects to the screen. This helps in fixing errors in programs.
Assembler/Compiler: tools that turn code into machine language programs (ones that the computer can execute).
Linker: a tool that combines multiple executables into one
Work completed this week:
Over the week I was looking to solve a few questions that are really important when you are developing low-level stuff:

- How would I use a debugger? Additionally, how would I help myself to find bugs in my code?
- What compiler would I use?
- Where is a good reference for info on the i686 architecture and how it's different than other architectures?

With this in mind I got to work finding answers for some of these. Here's the answers I came up with: I can have the OS print to the screen, but if there's an issue with the screen driver, I can always use a serial port to send data to myself to read later.
GCC would be my compiler of choice because it prints really nice error messages and it's the best for optimization. It also ships with an assembler and linker, so I wouldn't have to get those myself. To get info on the i686 architecture, I went looking for the official Intel docs, then I stopped looking because they cost $600. I'm still looking for more sources and info I could use for my project.
Resources consulted:
-MINIX source code, found at this URL (click here) on Github. I'll whip up a bibliography entry for it soon. Was invaluable in regards to learning about how the PC boots up and how to chainload from GRUB into 32-bit mode.
-Multiboot standard v0.6.96 (Click here to read it) from the Free Software Foundation. This is what a kernel must have for GRUB to boot it, and considering that's what I am using, it's quite important to get it right.
-Would like to say I have consulted the Intel docs, but I can't find the free version, if there even is one. Should I be able to find such a resource, it would likely be the single most useful resource for the project.
Problems encountered:

-The stock GCC that they have online doesn't support building for freestanding environments, so I had to build my own GCC from scratch. That took my computer over 5 hours to build.
-The source code of MINIX contains a lot of sections that aren't applicable to my project and only a small bit of its information may prove useful to me. I really need more sources.
-Boot code on i686 is only allowed to be 512 bytes at maximum. I'll have to be creative on space if I want to boot my OS safely.
Successful moments:
I recently got an i686 PC from 1993 that still runs! It only boots from a CD, so I need to plan out how exactly I will fit my kernel onto a CD and copy it to memory from the CD. Due to this, I will likely need to get a copy of the ISO 9660 standard (the format for how data is written to a CD) and find out how the boot info is read off a CD.
Evaluation of progress:
I would say that this first week was a great start. I haven't written any code yet; likely this week will be mostly reading and getting ready for writing my bootloader. I already have a few ideas for what my kernel would look like internally, and I've been writing scripts for my development environment. Future journal entries will be longer than this because this week was mostly setup for development.

Emacs, Git and Project Management

Okay, so I'm restructuring these. Journal entries are now twice a week and will answer the prompt, and cover some stuff completely unrelated to the prompt, but related to my project. Also, read the need-to-know vocab before you do anything else, or you'll be really confused.
Prompt(s):
How did you come up with the criteria for your project? (list the criteria in this entry)
Why did you choose these criteria?
Are these criteria achievable and measurable? How so?
In what ways and manner are they challenging for you?
Prompt Answer: My criteria is as follows:

- 300 LoC (lines of code)
- Prints text to the screen
- Takes input from the computer
- Must contain well-commented code and lots of documentation
- Must follow modern development practices

I selected these because they would improve my development proccess. The way I develop now is not the way I want to develop at the end of the personal project; for instance, I want to use a VCS over just making a duplicate of a file I want to edit. Most of my projects are developed for myself and never for someone else, so documenting is

important now that I actually have to share it with others. Development practices are done for a reason in the industry- that's to make life easier, and I should make my life easier. As for the 300 LoC, printing text, and takes input goals, these are goals I picked due to their difficulty. The next paragraph will explain why those are difficult goals.

When you write code that runs on raw hardware, you don't have the luxury of having an operating system beneath you. As you may remember from the last journal, an operating system is a program that runs on your computer that abstracts the variations in computer hardware in such a way that a program may run on many different computers. Since I am writing an operating system, that means I need to setup everything myself. For instance, getting the screen to print characters means that I have to manually draw characters on screen, pixel by pixel (because there's no operating system to ask to do it for me!). The criteria I have for my operating system leaves some complex problems yet to be solved, and I think the inherent challenge of developing for raw hardware makes them challenging.

Need-to-know vocabulary:

Git: a VCS tool commonly used by many.

VCS: Version Control System. Bascially, these systems allow you to create a log of all changes to a set of code, and undo or apply changes at will. These also allow you to "branch" a set of code; a branch is a copy of the master branch that you make changes to. You can merge and reverse individual changes and branches. While complicated, VCS tools are extremely powerful and useful for tracking info.

Emacs: A text editor that has an immensely powerful scripting language. I'm using it to help with my development by helping me find errors in my code and to quickly test my code. The changes I'm making to my Emacs will be on my Github page. I will link to my Github in the next entry, I hope.

Work completed this week:

I setup a Github account and created a repository, or code archive, for my Emacs configuration and for my personal project. Everything I do for this personal project will be avalible on ManageBac and the repository. The Github repository will be well-documented and I hope that it will be something anyone can look at and understand. Configuring Emacs so it meets all my needs. While you'll have to look at my repository to get all the details, this was no easy feat and quite the challenge, considering I have never used Emacs for C developement. This will make it easy to test code and spot quick mistakes before they happen. I found a mentor for my project. They specialize in managing the developnment of projects; this is exactly what I was looking for. They will likely be the single greatest assitance to my project, because well-structured code prevents bugs and streamlines development. **Resources consulted:**

I consulted my mentor for advice, but outside of that, I didn't consult any digital or paper resources these week. Later this week I plan to gather more resources for my project (some books, perhaps).

Problems encountered:

Git is not fun to use and took hours to fully configure and install. The repos won't be up until I know how to edit them from the command line and until I have a schedule for my project. Emacs is a nightmare to configure, and fully de-bugging my scripts cost me a lot of valuable time. I likely won't mess with the configuration files unless I absolutely have to.

Successful moments:

Finding a mentor was a really successful moment, because it took me a while to find someone who has the skill to help me improve as a developer. I was extremely nervous I wouldn't find one in time, so that was a great relief for me.
Evaluation of progress:
I am currently quite close to having a development setup ready to go, so then I can quickly program and log all changes I make. Once the environment is actually set up, I'll share my Github repos and get working on the actual OS.

First Call With Mentor & Something Unrelated

Prompt(s): None
Prompt Answer: None
Need-to-know vocabulary:
Kanban cards: A tool one uses to structure development. You create cards, and put them into bins. Each bin represents a level of completion of the card (To-Do, In Progress, Finished). The card must be named descriptively and have at least one commit (a change made to the source code that adds or removes something) attached to it.
Qemu: a virtualization tool used to simulate a computer inside your computer. How does it work? Well, I can't fully explain it here, but here's a simple explanation: Imagine you have an instruction on the CPU. For example, let's say the instruction is MOV EAX, RAX. So we move the value of register EAX to RAX. Now, on a regular computer, these are in hardware. To simulate them in software, we can make two variables in a random address in memory, and then tell the virtual CPU that these are the real values. Put more simply, virtual machines take a little bit of your PC's resources to simulate another PC that can only see the areas allocated to them. There's a lot I didn't cover here for brevity's sake, but hopefully this is enough to understand the concept.
2*32 and 2*64: 2 to the 32nd power and 2 to the 16th power.
Real mode and Protected mode: These are two separate modes in 32-bit CPUs. The "real mode" setting restricts the size of any one variable to 2*16. Software running in "real mode" can also only use the first 216 bytes of memory, or the first megabyte. In "protected mode", any one variable can be up to 2*32. This increases the amount of memory software can use from one megabyte to four gigabytes. Protected mode is far more feature-rich than real mode, but makes it harder to interface with hardware.
Work completed this week:
This half-week has been quite packed with homework, so I haven't been able to get done as much as I would like. I did meet with my mentor and I learned about how I enter protected mode.

In regards to my mentor, I was able to discuss some better development strategies. Eric told me that the best way to plan out development projects is to use kanban cards (see my definition above). With these, I can put the goals on my timeline directly into small steps that I can easily manage. I have also set up a Github account for my use, but until I can make sure nobody but me can change anything, I will not link it here. There might be a really short journal entry in the future that has the link, though.

I also learned a bit about the differences between protected mode and real mode. There are many of them, but you'll have to look above for my full definition. Information like this

helps me begin to plan out a bootloader.
Resources consulted:
Intel programmer's manual. I can't believe that this is now open for free. Here is the text. This used to cost over $600, so now the fact that I can read it is fantastic. This document specifies almost everything I need to know for my project: how to boot, how to print to the screen, the whole instruction set, and more.
Problems encountered:
Git still won't work. I've yet to narrow down the problem, but it likely lies somewhere in the firewall software of my laptop. I recently did something very stupid. While attempting to boot two different Linux distros, I broke the partition table at the start of the disk. This basically means that I had to reinstall everything. Luckily, I lost very little files because I always have backups. Perhaps that's the lesson to learn for today.
Successful moments:
Finding the Intel docs for free was a complete victory. I doubt that I'll need a lot of sources for this project, because most sources that exist are extremely comprehensive and authoritative. I got a working install of Qemu! Took around one hour to build on my laptop.
Evaluation of progress:
I am close to writing a bootloader, but still don't know how to use GNU Linker to make my actual bootable file. That is likely my next step.

Shorter Development Time

I've been told by my supervisor to shorten all journal entries to 350 words. So these are all now shorts.
Prompt(s): None
Prompt Answer: None
Need-to-know vocabulary:
No.
Work completed this week:
I have completed setting up my Github repository. There is no code in it. That will be added later.
Resources consulted:
I talked to my mentor to set up my Github. I also looked at the Soritx repository for info on a bootloader (found here). Sortix runs on the same hardware that I do, so its information is far more useful than that of MINIX for hardware specific problems.
Problems encountered:
Yesterday was ruined by having my entire Github wiped out.
Successful moments:
I was able to restore my Github back to its previous state.
Evaluation of progress:
I will have a schedule in ManageBac soon. I am on schedule to meet my goals.

Nothing Of Note

I have uploaded my schedule for the personal project.
Prompt(s): None
Prompt Answer: None
Need-to-know vocabulary:
No.
Work completed this week:
I have completed a schedule for my project. This is already on my Github.
Resources consulted:
I spoke with my mentor on reasonable goals for my software project.
Problems encountered: Deciding what to prioritize is difficult.
Successful moments:
I've thought quite a bit about what features my operating system should have. I will likely post some of my ideas on Github in the future.
Evaluation of progress:
I am currently behind schedule, but will quickly be ahead of schedule by the end of the calendar week.

All Systems Are Go

Prompt(s): What research have you been doing?
Prompt Answer:
I have been reading the Intel manual, the Multiboot source code, and the Minix source code. There aren't many sources for OS development and research, but the few sources are quite thorough and well-detailed. Most of the resources I am using are either just source code for an OS, or documentation on the CPU's design. Most of the stuff I am reading is documentation and technical literature, so most OS development information is in similar media (PDF files online, with the occasional book).
Need-to-know vocabulary:
System font: A font stored by the VGA card in case the OS doesn't provide their own. VGA cards also come with software to print characters at certain positions on screen.
System call: A function software running under an OS can use to send commands to the OS; this is usually for privileged operations
Work completed this week:
I haven't gotten any work done in Github. Reading the Intel docs is all I have been doing, because I needed to understand how to jump from Real Mode (16-bit mode) to Protected Mode (32-bit mode). I need to be in 32-bit mode per my project requirements.

I've also been working on the VGA driver. To do this, I'm implementing a system call kprint() where I can tell the VGA driver what text to print to the screen. The function can also print newlines, tabs, and spaces. So far, I'm planning on using the system font instead of turning on individual pixels to print characters. While this works, it makes the appearance of the font hardware-dependent.
Resources consulted:
The Multiboot standard has been helpful in understanding how to use GRUB to boot my

OS. I haven't written anything yet. To write the code, I must solve the problem first.
Problems encountered:
The Intel documentation and Multiboot standard are extremely terse.
Successful moments:
Nothing yet.
Evaluation of progress:
I am currently behind schedule, but will be ahead of schedule when I have time over fall break.

2(32) Ought To Be Enough For Anybody

Need-to-know vocabulary:
stdint.h: a program that sets the exact size of an integer (basically, the more data a single variable can represent, the more space it needs).
comn.h: a set of programs that I am creating, they give my OS some general-purpose tools (like being able to read/write sections of memory, enable/disable CPU features, and likely more as I work on it)
Work completed this week:
I haven't gotten any work done in Github. Reading the Intel docs is all I have been doing, because I needed to understand how to jump from Real Mode (16-bit mode) to Protected Mode (32-bit mode). I need to be in 32-bit mode per my project requirements.

I've also been working on the VGA driver. To do this, I'm implementing a system call kprint() where I can tell the VGA driver what text to print to the screen. The function can also print newlines, tabs, and spaces. So far, I'm planning on using the system font instead of turning on individual pixels to print characters. While this works, it makes the appearance of the font hardware-dependent.

I've also been working on creating my own comn.h and stdint.h libraries.
Resources consulted:
The Multiboot standard has been helpful in understanding how to use GRUB to boot my OS. I haven't written anything yet. To write the code, I must solve the problem first.
Problems encountered:
The Intel documentation and Multiboot standard are extremely terse.
Successful moments:
Nothing yet.
Evaluation of progress:
I am currently behind schedule, but will be ahead of schedule when I have time over fall break.

I Have To Rewrite Everything

Need-to-know vocabulary: C standard library: A standard set of functions that most C programs expect to exist. As an OS developer, I have to create my own. This is the hardest part of my project and will likely take the whole time, if not more.
GCC: I've covered this before, but it's software that helps turn C code into raw computer instructions.

Work completed this week:
I finished writing my video driver, but as I went to test-run my shiny new kernel, dozens of errors appeared in my virtual machine software. Over an hour of debugging later, I found my issue. When building my own custom version of GCC, I set it so that it should allow the inclusion of its own version of some standard libraries that are independent of an OS. Sadly, this is not the case; GCC does not provide freestanding libraries. What does this mean? It means that I will need to write my own C standard library.
Resources consulted: GCC documentation, but formally citing it will likely be a challenge.
Problems encountered:
Read the work section- I had a lot of problems.
Successful moments:
This will make my project better designed, as now I know how my components will go together. Additionally, everything should go together more smoothly. I've also heard about what's called "unit testing" in GCC: you can have GCC (the part that builds software) look over the code for bugs.
Evaluation of progress:
I am currently ahead of schedule, but I have yet to put in the sources as I am too busy coding. Hopefully I should be able to get it in before Thursday of this week.

Rewriting Everything

Need-to-know vocabulary:
C Pre-processor: Before code can be compiled in most programming language, the code must be parsed to check for mistakes. This is called a "pass" of the code, and most compilers will tell the programmer where mistakes are so that they can fix them. One thing the preprocessor does is decide what code goes into the final program; it is possible to write code called "preprocessor directives" that tells the processor to enable/disable code under certain conditions.
Work completed this week:
While working on my kernel, I've been updating my Github with new code at the end of the evening after significant changes. While not perfect, I think my OS reflects good design choices. But I've been left with a problem that I'll explain in the "Problems encountered" section.
Resources consulted:
I've been reading the source code of other operating systems and a lot- I truly mean it- of GCC documentation. Should be in my sources list soon, but I need to spend my time working on the code. While sources are important, I will soon be back into the weekly schedule of homework and studying, so I'll have drastically less time to code for long stretches. That's why I intend to put off citations for as long as possible.
Problems encountered:
My problem stems from the following issue: when writing a kernel, you'll be using a lot of the same libraries as you do in any application that runs on your OS- but slightly tweaked. For instance, one function I wrote today, called abort(), quits a program when there is an error. But we can't quit a running OS, so clearly what it should do is that if abort() is

called by the OS, it should print a crash message, and halt the CPU to prevent damage. This feature would be unneeded in a regular application. So then, how do I make it so that some code is only built in for the kernel and some code only for applications? I've yet to find a good solution, but I think it will likely involve C preprocessor directives. Theoretically, I can get the results I want by using them to toggle code on under conditionals, but I'm unsure as to the quality of this solution.

Successful moments:

I've rewritten everything and generally had an all-round productive week. By tomorrow, I should solve this "kernel vs app code" problem and be ready to implement it. This will make sure that I'm ready to go compile and test my code. Overall, I did not expect to move this fast, but this should buy me time for the (likely) difficult time I will have writing a keyboard driver and shell. While I likely won't exceed the goals I've set, I am confident that I will meet them for sure. This has been a great relief for me, because when I started this project I was worried that I wouldn't meet my goals.

Evaluation of progress:

I am currently ahead of schedule, but I do not have sources in yet. That should be done before Monday afternoon.

Rewritten Everything & A Solution

Need-to-know vocabulary:

GNU Make: A program for auto-building projects, especially ones with complex operands (like mine!). When used on code, it needs a file called a "Makefile" telling it what to do in the directory.

Linker vs Compiler: A compiler converts code into what is called "object files", or binary files that can be executed by the processor. By itself, the processor can't understand how to execute them. To solve this, a tool called a linker can put object files together. It also appends crti.S and crtn.S before and after the file. These two are powerful tools that almost all software is built with.

Ctri.S and Ctrn.S: My implementation of some computer magic. These are special files that are appended by the linker to every binary it creates. They always go first and last respectively, and provide functionality that apps need at the start and end of execution. Also known by the name of "Global Constructors".

Work completed this week:

I have finished re-writing all previously existing code, and I have now started to write new code. The fix to my long-standing code re-usage issue was fixed: when make is called, I can tell it to build code with certain options. This means that I can control exactly what lines of my code will go into the kernel, and what lines will go into everything else. Hopefully, I should have it all implemented by the end of the evening today.

Resources consulted:

I've been looking at some Makefiles of other projects.

Problems encountered:

I haven't built my project yet. It's possible that I may encounter issues with my first builds. I am also missing some features I need for kernel printing and software printing to work.

Successful moments:

As of right now, I have finally re-written all code used for my project. With the rewrite, I have fixed a couple of bugs from earlier:

- The formatting for comments was wrong. I have since corrected it for consistency's sake.
- All code was fixed so that there was minimal memory usage. Additionally, I have taken advantage of special features of the compiler to enhance speed.
- Additionally, I now have all libraries I need for cross-compilation.

Evaluation of progress:
I am currently ahead of schedule, but I do not have sources in yet. That should be done before Monday afternoon.

Hitting The Books, Fixing Documents + Where Have I Been?

Need-to-know vocabulary:
Special Character: a character that doesn't really exist, but tells other programs to do something special with the text (like add a newline or tab where the newline is).
Newline Character: a special character that represents the press of the enter key. Most keyboards and OS represent this key with 0x0A or something unused otherwise.
OOP: Object-oriented programming. Normally, programs are like "to-do" lists: they contain a list of things for the program to do (this is called imperative programming). In OOP, certain things can be called objects. What is useful about this is that you can group objects into groups of things called classes- a class is a group of objects with similar properties. Objects can inherit properties from their classes, and you can put classes inside of classes. That's where the fun starts, and this explanation ends (I don't want to get too in-depth).
Work completed this week:
Over the last couple of weeks I've possibly read more source code than I have any right to really force myself to do so. My issue that I've been working on for a couple of weeks is laughably small compared to previous (and future) hurdles, but I'd say quite possibly my hardest. The problem is as follows: ***HOW AND WHERE DO I ADD NEWLINES TO MY OS?***
While it seems simple (just increment the screen by one line when you see a line), this is actually an infuriatingly hard problem. You see, I have to manually type however many spaces are needed until the text wraps to the next column. I also have to make sure that my OS can do this every time the newline character is passed, and I have to make sure that if we're at the end of the screen, the cursor should wrap around to the top so the VGA card doesn't fail. This is a lot to track, and I've spent the past couple of weeks coming up with a perfect solution. On Halloween night I cracked the problem- that's for another section.
Resources consulted:
I spoke to my mentor on this problem a number of times, and he helped me out by suggesting how to do it. Also did a lot of rereading of some SORTIX and MINIX source code, and some of the Multiboot Standard to debug global constructors (see my last entry for more on that).
Problems encountered:

I ran into issues on Kubuntu so now I'm running Gentoo; I don't plan to change distributions for a while, so this should be the new normal until I find or make a distro that's better than this. GCC updated so now I have to rebuild my toolchain. That took an evening to fix.

Successful moments:

My solution to the newline problem was to add it in front of the term_putchar function; so when the terminal gets a new character, it should add one new line. Despite being less than 10 lines, this was a bug so infuriating to find I had to take days off from working on it so I wouldn't go mad from the irritation. It's also worth noting that my OS should "just work" on most Linux distros in the future upcoming patch, but that's to be seen. Recently I was able to speak to a few people that know some stuff about MacOS, so I may have working builds on MacOS as soon as I actually figure out how to use its command line.

Evaluation of progress:

See my GitHub here for more into. Go into the docs folder, then read the "progress.pdf" file. Depending on my speed, it *should* be up by the end of the day.

Really Cumbersome Bug

Note: I haven't done some of the prompt responses, nor have I provided nearly as many personal project journal entries as I should've. Next week over Thanksgiving break I will be attempting to finish the rest of the programming side of my project so I can start confirming my hypothesis.

Need-to-know vocabulary:

printf() and scanf(): These are two functions that read and write data to the virtual text terminal in my operating system. When you type text into your computer, how does the computer know where to send the data? It has programs use this library to get input from the user, and the operating system decides how to give programs input to the ones that need it. The scanf() reads data and the printf() shows it to the user. Since my OS is really simple, it will only use text for these, and not more advanced forms of input and output.

Work completed this week:

I've been working on online schoolwork but I've been getting up extra early in the mornings to work on my project and prepare for the next week of development work ahead. This has involved reading an absurd amount of documentation on PS/2 keyboard drivers from MINIX and SORTIX. I've been looking at their documentation because while Intel has a lot of info on this particular device, their documentation does not properly convey what one must do to get it working. If I fix this up I can write a driver and connect it to scanf(). After that point I will have met my project requirements and I'm technically done, but would like to see what else I can construct with the powers of my OS. Should I decide to do more work into December, I'll write about it here.

Resources consulted:

MINIX and SORTIX code.

Problems encountered:

Still fixing decimal printing in my own OS. The issue appears to be with the fact that my conversion function is not working properly, but I'm not completely sure. I plan to go and

investigate the issue. If the conversion system works, I should be able to print numbers in base 10, 8, 16, 32, and 20- all useful for debugging, since the system uses numbers with different memory (memory addresses are in base 8 and some devices only use base 20 for data internally). When I fix my conversion function I'll talk about it further, and show off it working in printf().

Successful moments:

Nothing yet. I've pinned down the source of the bug but this doesn't mean I've fixed it yet. It appears to be in the algorithm doing some cursed mathematics that results in a processor triple fault interrupt, but this needs further analysis. To do this, I've started rewriting my number conversion library for Linux so I can test the algorithm independently of my printing function.

Evaluation of progress:

See my GitHub here for more into. Go into the docs folder, then read the "progress.pdf" file.

I plan on being done with all code by the end of the month. While meeting my own specifications, I doubt that my OS will be finished enough in this deadline to have much user interaction beyond just reading one key at a time.

Thanksgiving Break, Part 1

Need-to-know vocabulary:

Bases: In mathematics, a base is the relative unit by which all other numbers are put in reference to. For instance, most math today is done in base 10 (decimal), so when you multiply something, it shifts in the decimal place whenever it is multiplied by 10. Other bases are the same principle, but with different numbers sometimes better suited to fit their environment. For instance, computer memory is often referred to in units of 8 (8 bits make up 1 byte), so as a result memory addresses are written in base 16.

Pointer: a number, usually in base 16, that refers to the exact memory address of the object. When data is needed by a function, there are two ways to give it this data: either give it the raw data, or the pointer to the data. Pointers can contain anything, including nothing and not existing, so they must be used with caution. Type casting: in the C language, you can't compare types that aren't the same (I can't compare a number to an array, for instance). However, types can be converted (you could take numbers out of the array and then read those individually). This is useful for when you need to use data with a certain function that only accepts certain types.

Work completed this week:

So far I've added many more tools to printf(), including several new ways to print integers and, of course, integer printing. A lot of the new features are older features reused in more intelligent ways, and I'm glad to finally have these all in. It will make debugging a lot easier, now that I can actually figure out where the problem is in memory thanks to pointer printing. Also of note: I've added new tools for building tools needed for my project and executing them. This was done to make it easier to install and run my OS in a virtual machine or real hardware. The project directory is getting cluttered with shell scripts, and I'd like to hopefully find some way to clean that up a bit.

Resources consulted:

MINIX and SORTIX code, referenced in ManageBac! Yes, it's now in there, since I finally updated it a while ago. This information was useful to figure out the cast for my pointer printing.

Problems encountered:

The conversion system stopped working, so I had to re-design it from scratch to be easier and more performant. Decimal printing randomly started and stopped working, seemingly at random. Random numbers were appearing onscreen, and newlines were being inserted where they should not be. Puzzled, I examined this further and concluded that there was something truly wrong with the magic va_list wizardry I had been doing, and rewrote it to be simpler. Instead of messing with pointer magic, it now creates a scratch buffer for holding data so data isn't overwritten by other functions using the same value. In retrospect, many different parts of my program using the same variable was a horrendous choice, and I'm shocked I did this. I'd eventually like to re-write it further so it becomes a case statement and not this long if-then tree, but that's for later. It now works, and I can print base 10, base 8, and base 16. Hypothetically I could add even more, but I have yet to think of a need for any more than that.

Successful moments:

Decimal printing, a problem plaguing me for the entire project, has now been fixed for good. I also brushed up on how keyboards work. Sadly keyboards are far more complicated than I imagined, but at least I understand what exactly I'm getting myself into. BIOS data tables was another thing I've learned about; when my OS boots I want it to check the BIOS table for information on where the video address is, and print it to the screen along with other data there. On real hardware, the video address is different than where it is in the VM, so I need to check to see if it exists.

Evaluation of progress:

See my GitHub here for more into. Go into the docs folder, then read the "progress.pdf" file.

TL;DR: With decimal printing done, I am now free to work on keyboards. Hopefully I'll have something to show by the end of the week, but no guarantees until it works in a VM.

Thanksgiving Break, Part 2

Prompt:

Reflect on the progress of your project. Focus on progress, social skills, and self-management skills/time management skills.

How is your final product coming?

How are you sticking to your plan?

(time management) Organization –Managing time and tasks effectively?

Celebrate success, admit...

Prompt answer:

I have managed my time well and have fully stuck to my plan. Due to the challenges of coding and debugging, my OS has had several setbacks. On raw hardware, bugs cause an instant execution failure with no explanation on why. This means that writing anything takes a long time, as all bugs must be pruned in a very careful matter. Thus, it is impossible for me to accurately predict how long anything will take. I would like to have keyboard

input by the end of November, but there may be unforeseen consequences, like extra parts of the standard I must comply with. Overall I think that my final product is nearing its completion, and I am proud of the work I have gotten done and will complete further. When keyboard input is added and parseable, writing code that uses it will be extremely easy.
Need-to-know vocabulary:
Toolchain: a set of tools required to build a set of software. Generally this includes dependencies (prerequisite tools and libraries needed by the program), and tools needed to make the final executable/installation (compiler, assembler, linker). Keymap: a table that compares the key number pressed to a letter or number

Work completed this week:
Today I've been working on the toolchain and having it be installed in a few simple steps. My OS has a relatively complicated toolchain, so I'm looking to make it easy to install in a couple of commands. Initially I started with the design of setting up a bunch of scripts, but have since learned that the "make" program I am already using to build my OS has the same functionality. Perhaps I'll rewrite it so it runs as a make script, but it needs to actually work first.

As for the keyboard, the first step is getting a reliable ACK (All Checks oKay) signal, and then I'll be able to verify that it is connected before any further data can be sent. Sending data without the keyboard being plugged in results in a triple fault, or an instant computer crash. I've been working on the mechanics of

Resources consulted:
PS/2 standard by IBM (will be on ManageBac by the end of the evening), MINIX pckbd.c
Problems encountered:
Different distros have different packages. Testing the same script on all the distros is hard, because I have to wait around 5-10 minutes for GCC and Binutils to finish compiling. This means that I have to wait 5-10 minutes to see if there's even a bug in my program. Running it on an extremely beefy computer helps, but not as much as it should. The builds are single-threaded, so I can only ever really do four at a time.

Shell scripting in the way that I'm doing is new and unfamiliar. Making sure my options work well on all systems is also a challenge, since not all systems support the same shell scripting extensions. **Successful moments:**
None yet. Still working on it. Maybe I'll post another update today if I finish it. **Evaluation of progress:**
See my GitHub here for more into. Go into the docs folder, then read the "progress.pdf" file.
TL;DR: With decimal printing done, I am now free to work on keyboards. Hopefully I'll have something to show by the end of the week, but no guarantees until it works in a VM.

Thanksgiving Break, Part 3

Prompt:
Reflect on the progress of your project in terms of your goal, global context and criteria.
How is your product coming along? What is the statues in relation to your goal?
How is your product's process reflecting your chosen global context?
What criteria have you be successful with completing & what criteria are you still working

on?
What adjustments have you had to make?

Prompt answer:
This is the same prompt as last time. I am nervous that despite several days of testing my PS/2 driver in my local testing version, it is not ready yet. I currently do not get the response I need from the keyboard firmware, and without it I'm stranded like a canoe without a paddle. This has proven to be one of, if not the hardest part of the project thus far.

Need-to-know vocabulary:
Intel VT-m: a hardware emulator for running virtualized OSes, meaning that each instruction is "virtualized" and run with the assistance of special hardware to speed up virtualization. This tends to be more like real hardware than the software emulator, but also slightly flawed and not exact to the standard (ironically also like real hardware) QEMU: a software emulator, meaning that each instruction is "simulated" and is not run with the assistance of special hardware to speed it up

Work completed this week:
I've gotten my OS to build and run in VirtualBox. This proved to be very difficult, but is needed for my criteria. I need this to run in as many virtual machines and real hardware as much as possible. In QEMU my OS runs fine, but in VirtualBox it ran into issues. After some further analysis this proved to be due to GRUB interfering, but now it works fine; this was not an issue with my OS specifically. I have uploaded a screenshot along with this- I did not know ManageBac had these powers- but now I will upload it.

I have also been working on the same toolchain build scripts for a while now; it is not done yet, but I'm getting there. By tomorrow I would like to have GCC and Binutils autobuild with the selected options.

Resources consulted:
VGA documentation in the Intel standard. **Problems encountered:**
VirtualBox and by extension Intel write the standard and do something else, and this has proven to writing a bunch of hacks. Writing a clean OS is proving to be a nearly impossible task, as progress becomes agonizingly slow as I fix bugs and polish existing code to a shine.
Successful moments:
It works now!
Evaluation of progress:
See my GitHub here for more into. Go into the docs folder, then read the "progress.pdf" file.
TL;DR: With decimal printing done, I am now free to work on keyboards. Hopefully I'll have something to show by the end of the week, but no guarantees until it works in a VM.

Continued Work

This isn't really much of an entry, rather just a note that I am still working on some code for the project. At the advice of my supervisor, I am working on the script for my final presentation. When that is done, I will code for the rest of winter break so I can fix minor bugs and issues in my OS. I will lock the repository (no further changes) sometime before January the 8th. I also plan to have at least two more project journal entries by that point.

As of the time of writing (12/13/20), the semester ends in exactly four days. When that time arrives, I hopefully will have my script approved so I can release my final commits for the project. Furthermore, I have one more meeting with my supervisor before then. The final patch will only make minor fixes, nothing too big I hope.