Checkpoint 1

Paging

1. After initializing paging, the image immediately crashes when booting
    a. The issue is our memory addresses in the page directory were not correctly assigned. To diagnose this issue, we used GDB step and stepi (for assembly) to step through and realized the image crashed as soon as paging was turned on (the appropriate bit in cr0 is set). This implied that the error was with our directory and table entries themselves, so we were able to find the fundamental issue was how we tried to constrain the kernel address to the top 10 bits.
    b. The fix was to simply fix the top bits such that they hold the top bits of the kernel address. Now the memory address it attempts to access will exist. The fix took a significant amount of time because we are not able to step through in GDB after the paging bit is turned on.
    c. ~ 5 hours
2. Our paging structs fail when initializing the page directories and tables
    a. Here our issue was that we tried to union two structs together which resulted in errors. Our reasoning initially was to union two of the structs together so we are able to access different bits of the entry for 4KB and 4MB entries.
    b. The solution was to inside only have one struct for page directory entry and one struct for page table entry. Using this format, we aggregated the important bits of both versions of the page directory entry. For example, we made sure to add a way to access the global bit correctly.
    c. ~ 30 mins
3. The inline assembly in file page.c was not loading the correct page directory address into cr3
    a. The issue is because of how we tried to add page directory address as an argument into the assembly. This is an error with the inline assembly syntax that we need to understand.
    b. We fixed this issue by changing how we actually pass in the directory to the inline assembly code. This consists of using an "r" command to read in the pointer to the page directory, and also stating that the register EAX will be clobbered. Then while using the input in the assembly code, we used %0 to access the first input.
    c. ~ 20 mins
4. The kernel address that shows up on info mem is incorrect
    a. When we ran info mem, the first entry in the page directory was all zeros, whereas the second entry in the page directory was mapped correctly (kernel). This meant that our logic for the kernel was correct, however our table implementation was incorrect.
    b. We looked through the table implementation and we realized that we were shifting the video memory address by 22 bits rather than 12 bits. By shifting the memory address by 22, we are treating the bits as if it was an index to the directory rather than the index to the page. As a result, we created a new macro function to define the index bits to be shifted by 12 to account for the middle 10 bits of the linear address.
    c. ~ 1 hour

IDT

OUTB orientation switched

After writing code for pic and keyboard initialization and handling, I found that nothing was working. I spent 2 whole hours figuring out what was wrong but was unable to. After reading the os-dev document that I referred to again, I found that the arguments for OUTB are switched for our implementation compared to what there is on the internet. After fixing this I was able to properly initialize my pic.

Assembly linkage not working

The first bug I got was that I was improperly doing ask linkage so I never was calling my exception handlers if an exception was thrown. This took me two hours to fix as every time I changed code I got more errors. I found that I did not include the proper header files where I needed them and I also had to set my linkage asm functions to global in the assembly file. I was able to find this fix after using gdb to step through the code for hours.

Our keyboard wasn't typing

Another bug that we had was when we tested the keyboard and reached a segment_not_present exception. To finally fix this we looked back at the initialization of our IDT and I noticed that in the idt function I had set the present bit for idt[0x20] to 1 rather than setting the idt[0x21] present but to 1. Once I spent an hour figuring out what was wrong, I was able to solve this bug.

Released keys appeared when typing

After my keyboard code worked I tried typing but upon releasing every key a random value would be displayed in the console. Since I only handled keys pressed in scan set 1 which are those from 0x0 to 0x59, I had an if statement where I would only call putc to print a character to the console if the byte value read in from port 0x60 was less than 0x59. This solved the issue and took me only 20 min to fix.

Our Rtc was never being called.

Our keyboard would function properly but after we wrote our code for Rtc we found out that Rtc should be initialized and sending interrupts to be handled but since the test_interrupts function was never called we realized that the handler was never being called. We also used GDB to step through the code and set a break point at our linkage function for Rtc and it never reached either. After taking a step back, realized that we would call enable irq() but this would never reach the secondary pic because we never added enable_irq(2). After doing this, we now got a segment_not_present exception thrown. This was the **second bug** connected to the first one here. After taking a look, we realized that it was a very silly mistake in being that I called enable_irc(1) instead of 8, fixing the error. This but overall took 3 hours to solve.

Checkpoint 2

FileSystem Driver

- Reading the longer files like fish and the large text file were resulting in page faults. The issue was that the read_data() function was not accurately decrementing the dataRemaining variable. As a result, the dataRemaining always stayed constant and the function tried to read more than it could in a block. We had to step through this function and print variables several times to narrow down the issue, and this took about an hour to fix.
- Reading longer files also resulted in another bug, which also resulted in page faults. Once again we used gdb to see what the issue was and stepped through, printing and mapping out the logic. We noticed that an issue occurred in the last line of the read_data function, which determined the next data block to be read. Essentially what was happening was that we had a variable that stored our block_idx, which would index into the data_blocks array to determine the index of the actual data block. When passing the block_idx into the data_block array we passed it such that it was data_block[block_idx++], which would mean data_block[block_idx] would be initialized first, before incrementing block_idx. We fixed it so that it was ++block_idx and we had no page faults. This took about an hour to fix.
- The first character on top of the screen was not printing at all under any context. The actual issue was just a type difference when writing to the console. The use of uint32_t as opposed to int resulted in a difference in signedness that caused bugs when using this value. This took about 5 minutes to fix as it was just changing the type of a variable.
- Trying to open a nonexistent file name did not return -1. We expect trying to open a file that isn't there should result in the function not passing and returning -1 but it was not because the open_file() function never checked if the read_dentry_by_name() function worked or not. To fix this, we held the return value of this function in a variable and checked if it is -1, then returned -1 if it is. This took about 10 minutes to fix.

Terminal Driver

- The arrow keys, page up/down, insert/delete, all f123456789etc. Would print a garbage value to the screen. After 30 minutes I had realized that my bounds check for printing was a silly mistake. Instead of checking if the scancode was less than 58, I did 0x58 which is a lot higher, allowing too many keys to print.
- When I pressed backspace and screen_x was 0, I could return to the previous line; however, if I clicked enter to go to a new line I could still return to the old line. I also page faulted if I backspaced at the top left of the screen. After an hour I implemented a flag that was raised if typing reached screen_x = 79 an wrapped around to the next line. With this flag, I could only return to the previous line if backspace was pressed and the overflow flag was high.
- Although visually on the screen I could handle backspace when I printed my buffer through terminal_write, it would not remove the characters that I had backspaced. To fix this, I added a condition to decrement the buffer pointer and replace that character in the buffer to a space character. This still did not work. After deleting this logic completely my code would somehow still return to the line unconditionally. This is because I would decrement screenx if backspace was pressed and if it became negative it would go back to the previous line. This took me 2 hours to fux but once I added the condition (if screenx >0) it worked.

- When I tried clearing the screen with ctrl+l and hit enter, I noticed that the buffer would have the character l in it. It took me some time to figure this out but I had realized that I had to clear my buffer upon every enter pressed and clear. Once I iterated through the buffer and cleared it this problem was solved. The fix was an hour long

RTC Driver

- There were few bugs for this section since we did not virtualize the rtc and were able to reuse code from the prior checkpoint
- One bug we had was that we could not properly decipher the frequency rate by right shifting by (rate-1). This took me an hour to fix and to remedy this we added a logarithm function that figures out what power of two the frequency was, subtracted it from the max, and masked it to figure out what the rate we have to use in terms of 0x__. This then worked
- In rtc_write we could not change the frequency. This was a two hour fix but very simple. I got helped by Dongming for this bug. We would call our change rtc frequency function with the buf argument directly but since buf is a void pointer we have to cast it to an int pointer and then also get the value inside of the argument instead of passing in the argument directly. Once I fixed this I could properly test my rtc.

Checkpoint 3

- We first had a bug with our jumptable logic, in the fact that the code compiled, but upon calling execute("shell"), we would get an exception error. This was due to the faulty pointer logic in our implementation of the jump table. We had multiple statements that included type casting and assigning functions to the jump table struct, however the program was not recognizing the specific function, which was some sort of syntax issue. We then started from scratch and understood that we can typecast the functions as pointers in a struct and then initialize these points to the specific functions we defined before. When we wrote the jumptable in this way, we ran the programs and they were perfect. However, the pingpong program was not working as expected. We read through some documentation and understood some code to realize that the program needed the use of rtc and so we included rtc files in our jumptable as well and in return, the pingpong program worked as expected. This issue took 3 hours to fix.
- Once we got executed working, we realized that we were not able to type in the terminal console. We looked through the code and saw that we had code so that the program spins nicely so we do not chew up cycles - we thought that the program kept halting and thus, we were not able to type. However this was not the case. Looking through, we looked through code and thought the read functions we were declaring were incorrect. We changed the file_read function to terminal_read function since the operation we are performing relates to stdout. However, this still does not remove the typing issue. Looking through further, we looked in the IDT handler to see if any of the code was not performing accurately. After lots of gdb and research, we changed the reserved3 bit from 0 to 1. This allowed us to type for system calls, where before it was not allowing us. Now when we see the ECE391OS> we were able to type. This issue took 8 hours to fix.
- Now that we were able to type, anything we typed would only return the first three characters which would be populated in the buffer. Debugging this took lots of time, as it required multiple gdb debugging processes and understanding of the system call linkage code. We set breakpoints to step

into the code to pinpoint where exactly the issue was occurring. We typed in “hello” into the terminal and then stepped through the code to see if the terminal function was recognizing the hello in its buffer, which it was. Each and every character of the buffer was correctly being populated. We pressed enter, and stepped through and the buffer still contained all of the characters. Stepping through again until the assembly linkage code, we saw that after the assembly linkage code the buffer populated with only three characters instead. So, the issue was in the assembly code itself. Reading through, pushing al and fl actually overwrote the EAX register, which was the return value from 5 (5 characters in hello) to 3, which represented the jumptable number. To fix this we pushed the registers manually so that EAX is not getting overwritten. After this, all characters we would write would be printed back to the console. This issue took 5 hours to fix.

- Along with the buffer issue, we encountered another buffer size issue. Reading through the code, we can make sure we understood how the system call functions were working. We looked through and realized the processor was issuing a variable to set equal to the ece391_read and then null terminals the buffer at the index which represents the return value of ece391_read. Knowing this, we checked to see our terminal_read function to see if we were returning the actual value. We realized that we were just returning the nbytes argument we were just passing in, which worked previously but not for this case. To change this, we implemented a new variable and returned that variable instead, along with resetting the buffer index to 0 after we are done writing to a buffer. After testing, this implementation worked flawlessly and now whenever we typed onto the console, it would repeat back accurately. This issue took about 2 hours to fix.

Checkpoint 4

- This checkpoint essentially involved tidying up the logic we had previously, where test cases would measure the accuracy of our code. A notable issue was that we were not able to cat the verylargetextwithverylargename.txt file. Once we encountered this issue, we had to look at our buffer logic when extracting inputs from the terminal. We looked to see where the filename was being used, and it was being used for read_dentry_by_name and in read_dentry_by_index. In the boot block, the filename can only store until .tx and not the last t, however we needed to make sure that it reads the file in both cases. To do this, we realized we need to restrict the filename buffer if it was longer than needed, and also cap it off so that it reaches .tx for max file name size. Through using GDB, we were able to pinpoint where exactly the issue was occurring and included an extra check, which involved checking if the file name was 32 or less and if it is more than 32, cap it off at 32, before calling the read_dentry_by_index function. We implemented the restricting logic in *execute* before passing it to the other functions. After implementing this, we were able to see the whole contents of the file. We also made sure to include a null terminating character at the end. This took about 4 hours to fix.
- Another one of our bugs was such that upon pressing enter, the terminal would crash immediately. We were very confused since all the logic previously worked, just not now. We set a breakpoint at execute and using GDB to trace the pathway upon pressing enter, we encountered that the issue was in the temrinal_write function. We saw that there was a check that when a newline was signaled, the write function would break out of the for loop to indicate that the current argument is done getting typed and needs to be read now. However, upon removing this, anytime we pressed enter the function would not break out of the for loop and now we were able to continue writing other words to the terminal. This issue took about 1 hour to fix.

- Another issue was that the program would not be able to read the whole file fully. We thought we handled this in the previous checkpoints, but upon looking through the code and debugging, we did not consider the edge cases that we had to fix. Using GDB, we were able to pinpoint that the issue was in our read_data function. Previously, after each read we would increment and go to the next data block. However looking through more carefully now, we included a check that first makes sure if we need to go to the next data block or not. And if we do then it initializes the offsets to 0 for the next block. After reading some data, we also realized that when reading, we were not incrementing the offset into the block, which would signify that we have always read with the wrong offset previously. Taking this into account, we now incremented the offset after each memcpy into the buffer. Compiling this and running the program again, fixed the exception issue after trying to cat a text file. This issue took about 5 hours to fix.

Checkpoint 5

- One of our issues was that we were unable to run the ls command multiple times after one another. We assumed this to be a terminal specific issue, however it was not. The first ls would work as expected, however every ls command after would output a directory entry read failed after printing the verylargetextwithverylongname.txt file. We set up multiple breakpoints and attempted to step through the whole ls program, but we were unable to pinpoint the exact root of the issue. We stepped through the previous printing of the verylargetextwithverylongname.txt file and it seemed to be correct, but when we stepped through the next read function, our file descriptor had a value of 2562, which was nowhere close to between 0 and 8. We thought we were not handling the fd properly, but stepping through the code rendered that assumption invalid. We eventually came to figure out that it was the buffer that was the only issue causing the fd to become a different number. We were not handling for when the buffer size would be greater than 32 for directory read, and once we implemented the logic, we were able to perform multiple ls commands. This issue took about 7 hours to fix.
- Another issue was that we were not able to switch between terminals properly. We have already implemented the process index logic so we had to just focus on the terminal switching logic. We looked at the discussion slides to determine how exactly to determine the switch between the terminals. We initially mapped the physical address for the video memory location, which was incorrect based on our understanding. After going to office hours, we eventually understood that we had to actually change the video memory itself. As a result, we implemented a new method where our paging scheme was the same as before for checkpoint 4. Now, we include a memcpy between the video memory and the video page address for both before and after switching the terminal. This allowed us to restore the current video memory terminal and also change the video memory to the current terminal we are at. As a result, if we were to switch back to a previous terminal we would have already had the memory stored in the virtual memory page. This issue took about 5 hours to fix.
- Another bug we had was that any time we would type and switch to another terminal, the cursor would be in the location of the previous terminal in the newly switched terminal. To fix this, we had to implement an update cursor logic algorithm which we would call after we switched the terminal. To accomplish this, we had to declare a global array which stores the x and y locations for each terminal. We would update these arrays in the same way we updated our video memory before and after switching the terminal, this allowed us to store the previous location of the cursor for any of the terminals we implemented. This issue took about 1 hour to fix.

- We also had trouble with the process index logic in the sense that we had to handle scenarios where all of our processes would have to add up to 6 processes maximum. At first, we handled the terminals in such a way that the first three processes were already taken up. Then we implemented it so that only when you switch to the next terminal, then a new process gets allocated. We then implemented a function that allows us to determine the next available process so that we can determine it and then allocate that specific page in physical memory when we call for a new process in the terminal. This allowed us to only have six processes. This issue took about 1 hour to fix.