ECE 391, Computer Systems Engineering
MP3 Checkpoint 1 Hints
Fall 2020

General Guidelines

This document is intended to provide some clarity as to what is expected from your submission at demo. If you have any feedback to make the document more clear and concise please let us know on Piazza.

1. During demo we will run your tests and only add tests if we determine that your tests are inadequate. You will lose points for that if we have to add tests.
2. Remember that your tests are running as kernel code.
3. We will be testing different sections independently and will restart your OS multiple times with different tests enabled each time. Ex: We might enable the RTC once, test it, and then the next time enable keyboard interrupts to test your keyboard functionality.
4. Get comfortable with function pointers and writing assembly code in separate .S files as working with inline assembly can get tricky and lead to subtle bugs in your code.
5. Remember to take advantage of all the main functionalities C and x86 Assembly offer you, **code smart!** For example, use structs for the IDT entries, Paging structures etc and use arrays for the keyboard handler!
6. Remember to maintain your **bug log**! While we know that you are capable programmers, we know that everyone has bugs. If you tell us that you had no bugs and hence have none in your bug log, we won't believe you.
7. As always try to use your best style and document code as you write it.

Initialize the IDT

1. For this checkpoint, printing a prompt as to which Exception was raised and freezing by using a while loop is enough as there are no user programs running.
2. You will be implementing System Calls in Checkpoints 3 and 4. For now, you may print that a system call was called when the handler associated with 0x80 is executed.
3. It is highly recommended that you have a common assembly linkage. You will be required to do this at a later checkpoint if you do not implement it now.
4. For entries reserved by Intel you need not fill in anything for the IDT.
5. Once an exception has occurred and you are executing in the handler, no other interrupt must be able to occur.
6. You must have tests to show that you are able to raise **multiple** exceptions.

Initializing the RTC

1. To show us during demo that the RTC interrupts are being received, you should use the `test_interrupts` handler or a function similar to it. If working correctly, the characters on screen should alternate between a set of different characters at every fixed interval.
2. Though not required for this checkpoint or the next, it is highly recommended that you virtualize your RTC by the next checkpoint. By virtualizing the RTC you basically give the illusion to the process that the RTC is set to their frequency when in reality the RTC isn't. Instead you set the RTC to the highest possible frequency and wait until you receive x interrupts to return back to the process such that those x interrupts at the highest frequency is equivalent to 1 interrupt at the frequency the process wants.

Initializing the Keyboard

1. It is sufficient if you show that pressing a key on the keyboard can echo that key on the monitor.
2. We will only test lowercase letters and numbers for this checkpoint and if a single press displays the right character. We will not stress test your keyboard at this checkpoint.
3. You need not care about mapping function keys, special keys such as (but not limited to) alt, ctrl, caps lock, shift, tab, the arrow keys or the numpad for this checkpoint. **As long as pressing these keys does not crash your kernel, you won't lose points for not printing these keys**.
4. Typing to get the right characters on the keyboard is sufficient in terms of testing.

Paging

1. Open the qemu console and use the command `info mem` to see if you have mapped the pages correctly.
2. You can allocate memory for the paging structures in the `x86_desc.S` file, this will place the paging structures in the kernel page. Using structs and arrays will make your job easier.
3. Video memory must only be **one 4KB page**. You may be penalized for allocating more pages than needed for video memory. RTDC and RTCC to see which 4KB page video memory should be at.
4. You must have tests that show that dereferencing locations that shouldn't be accessible aren't accessible. Conversely, you must also have tests that show that dereferencing locations that should be accessible are accessible.
5. Test all edge cases.
6. Data types for pointers are important! Don't overlook them and confuse yourself when a test doesn't work as expected.

ECE 391, Computer Systems Engineering
MP3 Checkpoint 2 Hints

General Guidelines

This document is intended to provide some clarity as to what is expected from your submission at demo. If you have any feedback to make the document more clear and concise please let us know on Piazza.

1. During demo we will run your tests and only add tests if we determine that your tests are inadequate. You will lose points for that if we have to add tests.
2. Remember that your tests are running as kernel code.
3. We will be testing different sections independently and will restart your OS multiple times with different tests enabled each time. Ex: We might will test if you can read small files once and then large files the next time.
4. All your drivers must have the `open, close, read` and `write` functions defined, even if they don't do anything and just return a constant value. Likewise, all these functions must return an integer and follow the interface provided for the corresponding systems calls.
5. Read ahead for the next checkpoint as a lot of the functionality there is dependent on features you implement in this checkpoint.
6. Please look at the hints post on Piazza to find a gif of how test cases should look. Remember that this is not a comprehensive list.
7. Remember to validate your input parameters!
8. Get comfortable with function pointers and writing assembly code in separate .S files as working with inline assembly can get tricky and lead to subtle bugs in your code.
9. Remember to take advantage of all the main functionalities C and x86 Assembly offer you, **code smart!** For example, use structs for the file system structures and use arrays for the terminal!
10. Remember to maintain your **bug log**! While we know that you are capable programmers, we know that everyone has bugs. If you tell us that you had no bugs and hence have none in your bug log, we won't believe you.
11. As always try to use your best style and document code as you write it.

RTC Driver

1. `rtc_open` should reset the frequency to 2Hz.
2. Make sure that `rtc_read` must only return once the RTC interrupt occurs. You might want to use some sort of flag here (you will not need spinlocks. Why?)
3. `rtc_write` must get its input parameter through a buffer and not read the value directly.
4. There is a one line formula that you can use to check if a number is a power of two. You do not need to use a lookup table or switch statement to do so. *Hint: Think binary or in terms of bits.*

5. Your test cases should be able to demonstrate that you are able to change the rtc frequency through all frequencies.

6. Though not required for the MP, it is highly recommended that you virtualize your RTC by the next checkpoint. By virtualizing the RTC you basically give the illusion to the process that the RTC is set to their frequency when in reality the RTC isn't. Instead you set the RTC to the highest possible frequency and wait until you receive x interrupts to return back to the process such that those x interrupts at the highest frequency is equivalent to 1 interrupt at the frequency the process wants.

Terminal Driver

1. All numbers, lower and upper case letters, special characters, the shift, ctrl, alt, capslock, tab, enter and backspace keys should work as intended. You do not need to print anything on hitting a functional key such as shift etc.

2. You need not care about mapping the arrow keys or the numpad. **As long as pressing these keys does not crash your kernel, you won't lose points for not printing these keys**, these keys should not be counted as part of your input buffer either.

3. `terminal_read` only returns when the enter key is pressed and should always add the newline character at the end of the buffer before returning. Remember that the 128 character limit includes the newline character.

4. `terminal_read` should be able to handle buffer overflow (User types more than 127 characters) and situations where the size sent by the user is smaller than 128 characters.

5. If the user fills up one line by typing and hasn't typed 128 characters yet, you should roll over to the next line. Backspace on the new line should go back to the previous line as well.

6. `terminal_write` should write the number of characters passed in the argument. Do not stop writing at a null byte.

7. You may choose to not print the null bytes (recommended).

8. Both `ctrl + l` and `ctrl + L` should clear the screen and reset the buffer. You should not be resetting the read buffer if the user clears the screen before pressing enter while typing in `terminal_read`.

9. Your keyboard test can just be a calling `terminal_read`, followed by `terminal_write` in a while loop.

File System Driver

1. `read_dentry_by_name, read_dentry_by_index` and `read_data` are **NOT** provided for you. The documentation says that these functions are provided by the file system module, you are required to write the file system module

2. You should have tests to demonstrate that you can read small files (`frame0.txt, frame1.txt`), executables (`grep, ls`) and large files (`fish, verylargetextwithverylongname.tx(t)`).

3. `directory_read` should only read one file name at a time.

4. The file names in the dentries need not be null terminated.

5. On reading executables, you should see "ELF" in the beginning and the magic string "0123456789ABCDEFGHIJKLMNOPQRSTU at the very end. If you're printing null bytes you might not be able to see both at the same time. If you're using printf to print you might not be able to see the last magic string and will stop printing at null bytes.

6. Your test for `directory_read` should look like the output from ls, look at the gif for more information.

ECE 391, Computer Systems Engineering
MP3 Checkpoint 3 Hints

General Guidelines

This document is intended to provide some clarity as to what is expected from your submission at demo. If you have any feedback to make the document more clear and concise please let us know on Piazza.

1. It's time to execute user programs! We will no longer be running your tests, instead we will run a few of the user programs given to you. On startup you must be running the `shell` program.
2. This requires you to have the `execute` system call running. If you don't, but have other system calls running you need have tests to demonstrate that they work. If you don't, you will lose all functionality points.
3. **Read the Appendix very carefully!**
4. The doc for this checkpoint might split it up into 5 sections, but parts 6.3.2, 6.3.3, 6.3.4 and, 6.4.5 are all portions you must implement while implementing the system calls as a whole, mostly in `execute` and `halt`.
5. Remember to validate your input parameters!
6. Get comfortable with function pointers and writing assembly code in separate .S files as working with inline assembly can get tricky and lead to subtle bugs in your code(specially in `execute`!).
7. Remember to take advantage of all the main functionalities C and x86 Assembly offer you, **code smart!** Which it may be amusing to write all of the functionality for each system call in assembly, it is hard to do so and to debug. Instead write most of it in C and use either asm volatile or call assembly functions from your C function.
8. Remember to maintain your **bug log**! While we know that you are capable programmers, we know that everyone has bugs. If you tell us that you had no bugs and hence have none in your bug log, we won't believe you.
9. As always try to use your best style and document code as you write it.

System Call Handler

1. Make sure that you have proper assembly linkage for the IDT entry for system calls!
2. You should be verifying that the system call number passed in through `eax` is valid.
3. You **must** use a jump table to execute the right system call here.

Execute

1. If you are having a tough time debugging here, which you most likely will, its always a good idea to take a step back and see what every block of your code is doing at a high level. Making helper functions here is highly encouraged as that helps you test individual portions of what the system call should be doing as a whole.
2. Though you don't have to use the arguments until the next checkpoint, it might be useful to store them somewhere(!?) that you can use later.
3. When reading in the start address of the program make sure you read the bytes in the right order!
4. Usually each user program is given its own page directory, for this MP you do not need to do this. You can use the same page directory and just modify the few entries in it to adapt it for the user program you're running.
5. Make sure that the address of the first instruction for the user program is an address in the user page that you allocated! RTDC to find where this page should be.
6. Though we will be primarily testing using `shell` and `testprint` in this checkpoint, its highly advised that you run as many of the user programs as possible to test if all your system calls work! `shell` uses the `write`, `execute`, and `read` system calls. `testprint` uses the `write` system call. All user programs call the `halt` system call on termination. Other user programs will be using other system calls. Ex: `cat` and `grep` use the `open` and `close` system calls but also require you to have finished the `getargs` system call which isn't required until next week.
7. You must be squashing erroneous user programs. This means that if a user program exits unexpectedly you have to tell `shell`(the parent process) that they did so, so that `shell` can handle the failure properly: in our case, display the correct prompt. You can use the `sigtest` user program to test if you're squashing programs correctly (you will need `getargs` working for this, try "`sigtest 0`").
8. Don't forget to flush the TLB!

Halt

1. Remember that `halt` must return a value to the parent `execute` system call so that we know how the program ended.
2. Halting the base shell should either not let the user halt at all, or after halting must re-spawn a new base shell. This way you must ensure that there is always one program running.

Open, Read, Write & Close

1. Remember to validate the input variables!
2. These system calls, except `open`, will be very small, typically not more than 10 instructions long.
3. The `open` system call must initialize the file descriptor array entry for the file descriptor(the index into the file descriptor array) that it is going return appropriately based on the file type.
4. While setting up the fops table in the fda, you **MUST** use static tables with the function pointers already defined and assign the correct one to the fops table for that particular file descriptor.
5. It is important that you always call the corresponding driver `open`, `close`, `read`, and `write` functions, even if they do not do anything! For the open and close functions for the terminal, you can just call a "`bad_call`" function that always returns -1.
6. You might wonder where you should store the fda. Remember that each process has its own fda. That should point you in the right direction ;)
7. On backspacing while `shell` is accepting input, you should not backspace over the shell prompt that shows up.

ECE 391, Computer Systems Engineering
MP3 Checkpoint 4 Hints

General Guidelines

This document is intended to provide some clarity as to what is expected from your submission at demo. If you have any feedback to make the document more clear and concise please let us know on Piazza.

1. It's time to execute more user programs! You **MUST** be able to execute user programs by this checkpoint. If you are not able to, you will get no functionality points.
2. **Read the Appendix very carefully!**
3. Remember to validate your input parameters!
4. Get comfortable with function pointers and writing assembly code in separate .S files as working with inline assembly can get tricky and lead to subtle bugs in your code(specially in `execute`!).
5. Remember to take advantage of all the main functionalities C and x86 Assembly offer you, **code smart!** For example, use structs for the file system structures and use arrays for the terminal!
6. Remember to maintain your **bug log**! While we know that you are capable programmers, we know that everyone has bugs. If you tell us that you had no bugs and hence have none in your bug log, we won't believe you.
7. As always try to use your best style and document code as you write it.

More General Guidelines!

1. You should be able to support more than one working recursive `shell`(this means that once `shell` is running you should be able to execute another `shell` on top of it and so on). We require you to have at max 3 such shells. It is okay if the last shell is unable to run any programs, but it must be able to exit. All other shells must be able to execute programs.
2. Test all user programs(except `sigtest`). They should all work correctly!
3. Make sure that the running of one user program does not affect the running of the next program. `pingpong` will never stop running and that is okay, you have restart the OS to run other programs.
4. Make sure that you still have function interfaces for assembly functions.