Challenge problems:-

Lab 1

**Challenge 1:-**

**Enhance the console to allow text to be printed in different colors.**

**Answer:-**

This can be run in the “**challenge1\_color\_mappings**” directory.

We have used the ANSI escape sequences to print text in different colors. We have added a new option in help menu.

setcolor <color>

<color> takes <red, green, yellow, blue, magenta, cyan>.

If the passed option doesn’t match with any of the above, then the color will be set to the default option.

For implementing this functionality, we defined a new function in the kern/monitor.c file.

The function is:-

int mon\_color(int argc, char \*\*argv, struct Trapframe \*tf)

make run-testtime-nox

help

setcolor red

Lab2

Challenge 2

Extend the JOS kernel monitor with commands to:

* Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter 'showmappings 0x3000 0x5000' to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.
* Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!

**Answer: -**

This can be run in the “**challenge1\_color\_mappings**” directory.

Run

make run-testtime-nox

Then on getting the trap in the end, you can do the following.

* We have added “**showmappings**” function in the help menu of the monitor

Syntax is <showmappings startaddr endaddr >

**Example** showmappings 0x8000a00000 0x8000b00000

* We have also added “**dumpmemory**” function in the help menu of the monitor.

Syntax is <dumpmemory startaddr endaddr >

**Example** dumpmemory 0x8000a00000 0x8000b00000

Lab 3

**Challenge 2:-**

**Modify the JOS kernel monitor so that you can 'continue' execution from the current location and so that you can single-step one instruction at a time.**

**Answer:-**

This can be run in the directory “**challenge3\_step\_continue**”

As per the hint, we checked the eflags and found that we need to set the Trap flag (TF) for single step instruction execution. For continue, we need to set the Resume Flag (RF) and unset the (TF) flag.

We also handled the single step case in trap.c for step function.

We have added the “step” and “continue” option to the monitor.

Step will perform single step execution.

Continue will continue the execution.

We can run the “**challengebreakpoint**” program to test this. We have added two breakpoints in this program.

make run-challengebreakpoint-nox

step

continue

We have wrote code in “challengebreakpoint.c” such that we will get two breakpoints exceptions.

If we will press step, then it will step one instruction. If we press continue, then we will get log **“Anshul”** and will get another breakpoint exception. If we again press continue then we will get another log **“Ankit”,** showing that we

are able to step/continue correctly.

Lab 4

Challenge 5

Extend your kernel so that not only page faults, but *all* types of processor exceptions that code running in user space can generate, can be redirected to a user-mode exception handler. Write user-mode test programs to test user-mode handling of various exceptions such as divide-by-zero, general protection fault, and illegal opcode.

**Answer: -**

This can be run in the “**challenge4\_5\_divzero**” directory.

We have changed the environment structure to support the divide by zero exceptions. We have defined a new handler for handling the divide by zero exceptions and also have made system calls just like handling the pagefaults.

We need to run the user program “**divzerofault**” to test this case. On getting the divide by zero exception, the handler for this is called and we are printing the log “In user env: divide by zero occured..exiting” which shows that the handler is getting called properly and the program is exiting gracefully.

make run-divzerofault-nox

Lab 5

Challenge 3

Extend the file system to support write access. Here are a few points you need to consider:

1. Use the block bitmap starting at block 2 to keep track of which disk blocks are free and which are in use. Look at fs/fsformat.c to see how the bitmap is initialized.
2. Make use of the alloc argument in file\_block\_walk. In file\_get\_block, allocate new disk blocks as necessary.
3. In your block cache, use the VM hardware (the PTE\_D "dirty" bit in the uvpt entry) to keep track of whether a cached disk block has been modified, and thus needs to be written back to the disk.
4. Handle O\_CREAT and O\_TRUNC open modes in serve\_open.
5. Handle more file system IPC requests, such as FSREQ\_SET\_SIZE, FSREQ\_WRITE, FSREQ\_FLUSH, FSREQ\_REMOVE and FSREQ\_SYNC, in fs/serv.c. We have defined the argument for these calls for you in inc/fs.h. Also, write the corresponding service routines in fs/fs.c and hook them to client stubs in lib/file.c.
6. For more information about the file system's on-disk structure, read inc/fs.h and fs/fsformat.c. You may also refer to [last year's lab 5 text.](http://pdos.csail.mit.edu/6.828/2011/labs/lab5)

**Answer**

This can be tested in the “**1\_fs\_writable**” directory.

We have implemented the basic writable functionality to the JOS file system.

This can be tested by running the “testfile” testcase which is successfully getting completed.

make run-testfile-nox

Beside it we can run other programs mentioned below.

make run-testfilewrite-nox

make run-testfiledelete-nox

make run-testfile-overwrite-nox