Eric Blasko

Dr. Tong Yu

CSE 461

14 January 2019

Lab 1 – Writing a Simple Shell and Study of XV6

\*\*Note – This lab was completed independently. Also, as a reminder, for labs attendance, I am in the room across the hall as I am the TA for cse460. This was approved by Tong Yu. Each section has code or screenshots, followed by a discussion. If you have any problems reading the screenshots, let me know and I will do type scripts for the other labs

1. **Writing a Simple Shell (Completed)**

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>

#include <string.h>

void read\_command (char cmd[], char \*par[])

{

char line[1024];

int count = 0, i = 0, j = 0;

char \*array[100], \*pch;

//Read line

for(;;)

{

int ch = fgetc(stdin);

line[count++] = (char) ch;

if (ch == '\n')

break;

}

//return if command is 1 or less char

if(count <= 1)

return;

//create tokens

pch = strtok(line, " \n");

//parse the line into words using tokens

while( pch != NULL)

{

array[i++] = strdup(pch);

pch = strtok(NULL, " \n");

}

//first word is the command

strcpy (cmd, array[0]);

//other are parameters

for(int j = 0; j < i; j++)

par[j] = array[j];

par[i] = NULL;

}

//Function to clear screen at start and to print input prompt

void type\_prompt()

{

static int start = 1;

//only done of first start

if (start)

{

start = 0;

printf ("\033[2J"); //clear screen

printf ("\033[H"); //move cursor to home

printf("What is Thy Bidding, My Master?\n");

}

printf("$");

}

//Main prog. Runs shell by creating child through fork to execute command

int main()

{

char cmd[100];

char command[100];

char \*parameters[20];

char \*envp[] = {(char \*)"PATH=/bin", 0 };

//repeat till break

while(1)

{

//clear and display prompt

type\_prompt();

//read input from terminal

read\_command ( command, parameters );

//parent - waits for child

if(fork() != 0 )

wait(NULL);

//child - executes command

else

{

//copy /bin/ to cmd

strcpy(cmd, "/bin/");

//concatenate command to end of cmd

strcat(cmd, command);

//execute cmd with parameters

if(execve(cmd, parameters, envp) <= 1 )

{

//if not a valid command and command was not exit

if(strcmp(command,"exit") != 0)

{

printf("I find your lack of command disturbing...\n");

break;

}

//if command was exit

else

printf("You have failed me for the last time...\n");

}

}

//compare if command was exit. break loop

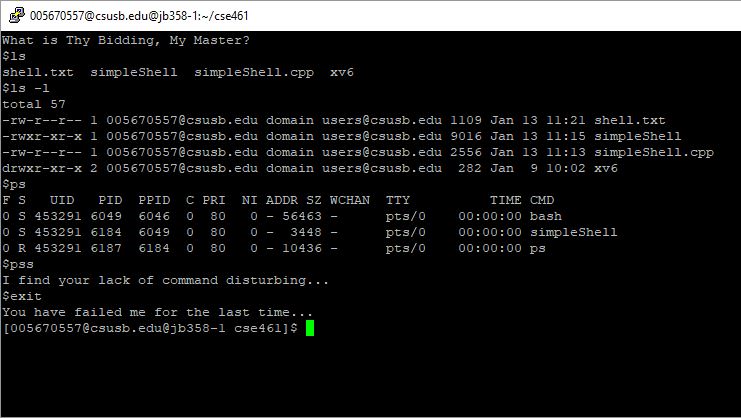
if(strcmp(command,"exit") == 0)

break;

}

return 0;

}



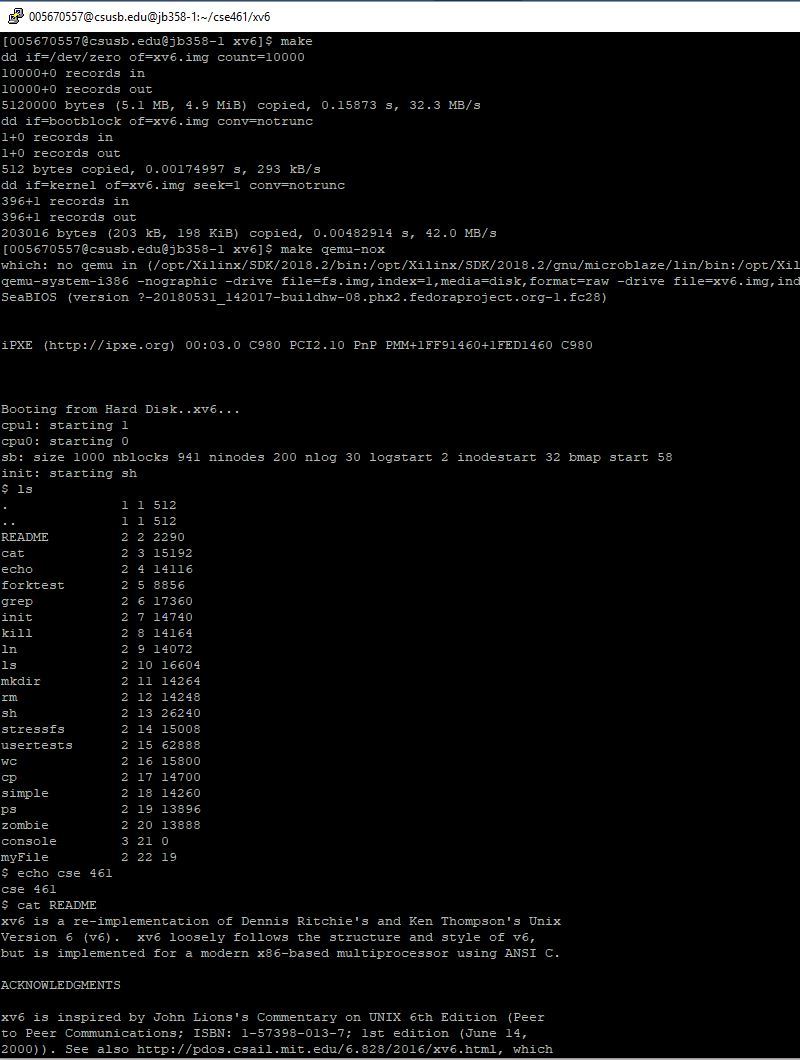
**Discussion:**

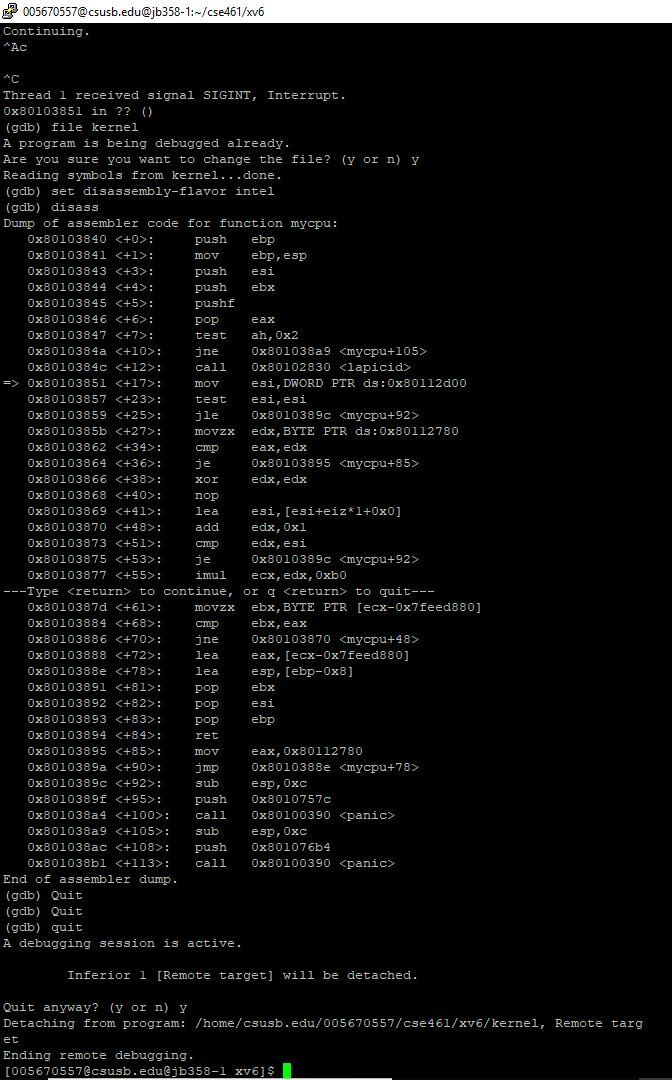
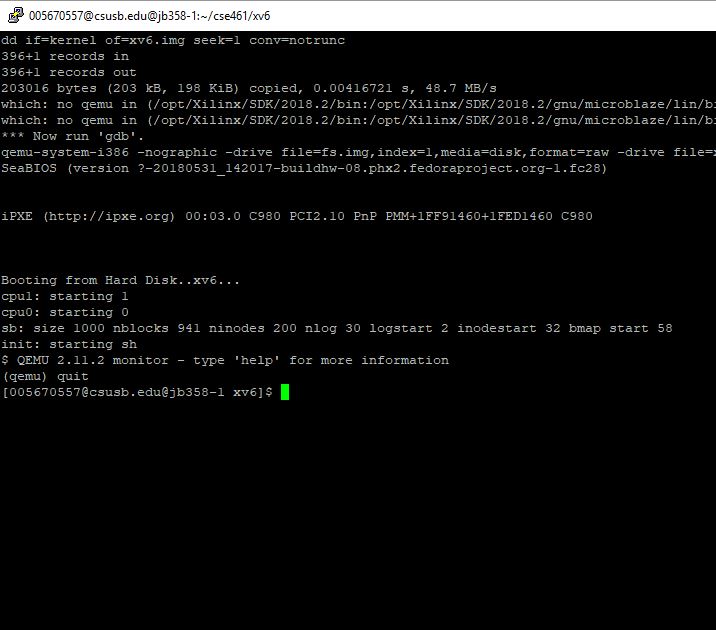
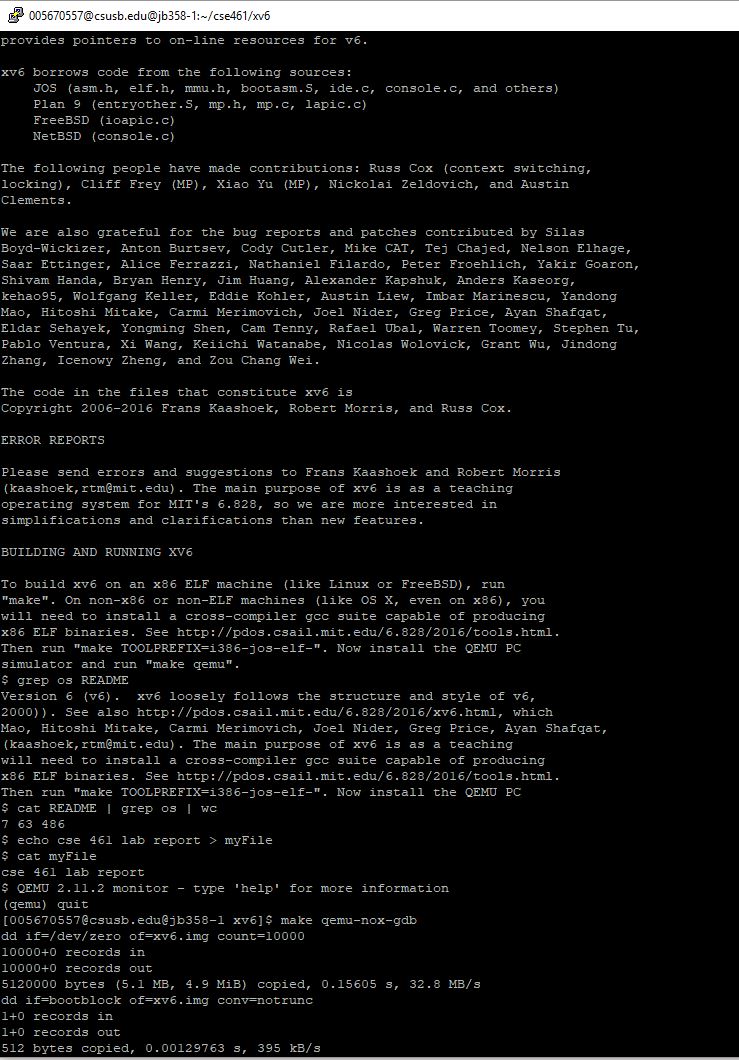
When the program first starts, main calls type\_prompt() to clear the screen and return the cursor to the ‘home’ position. This is done using printf commands with escape sequences. The \033[ is the start of the escape sequence followed by ‘J2’, which means to clear the entire screen. If ‘J0’ or ‘J1’ is used, it will clear the console either before or after the cursor. The ‘H’ escape sequence returns the cursor to the home location. A line prompt is then followed.

The read\_command() function then reads in the users input by breaking up the command using tokens. This allows the first word to be set as the command with the rest being the parameters. If the command is less than one character in length, the function returns and displays an error for a bad command. Once the command and parameters are separated, main calls fork() which creates a parent and child process. The parent waits, until the child process completes executing the command.

I have added additional text such as a beginning prompt, a prompt when a bad command is entered, and when exiting to make my shell feel more unique. When running simpleShell, the above screenshot is the result. Ls list the current files in the directory, ls -l displays inode numbers, file flags and file name, and ps displays a snapshot of the current processes.

1. **Basics of XV6 (Completed)**

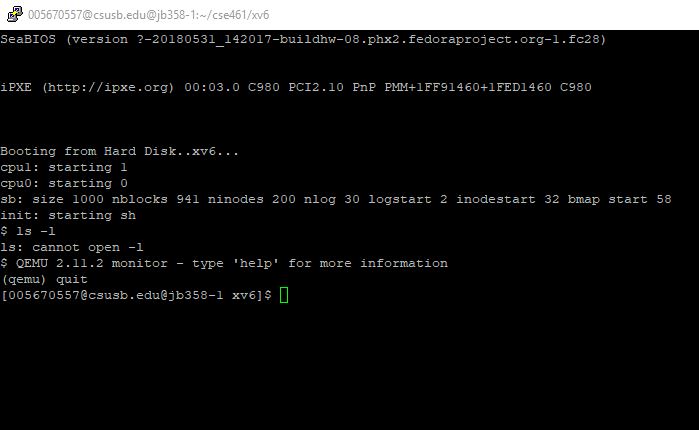
****

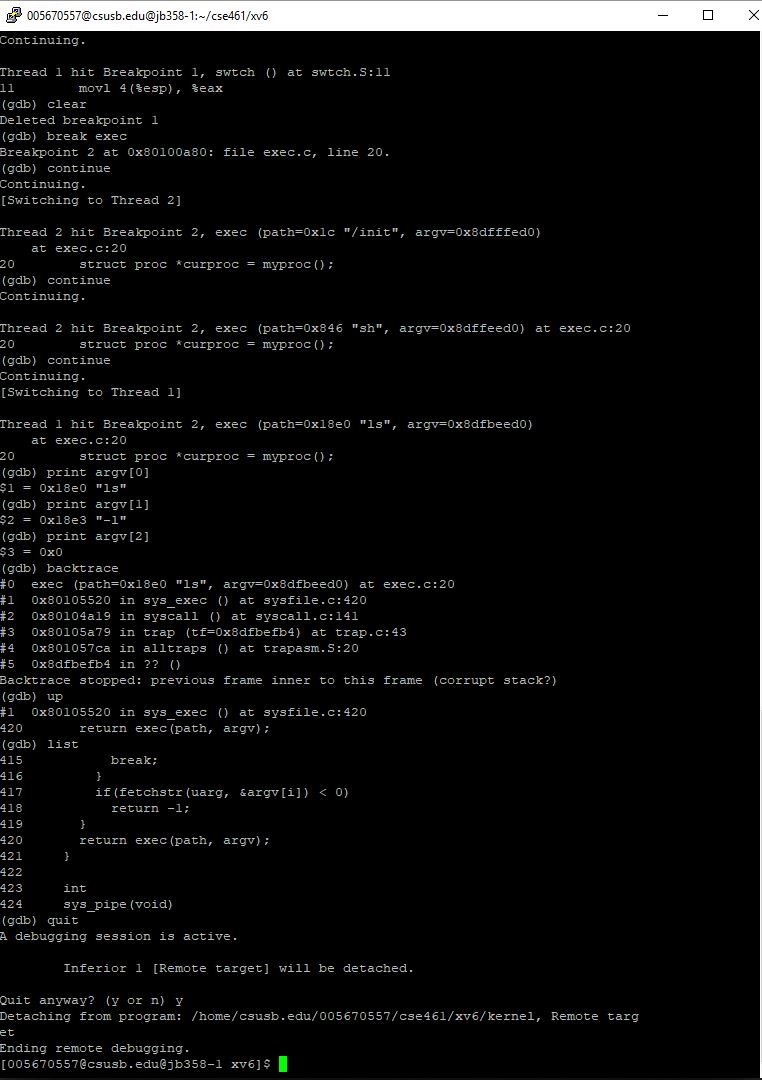
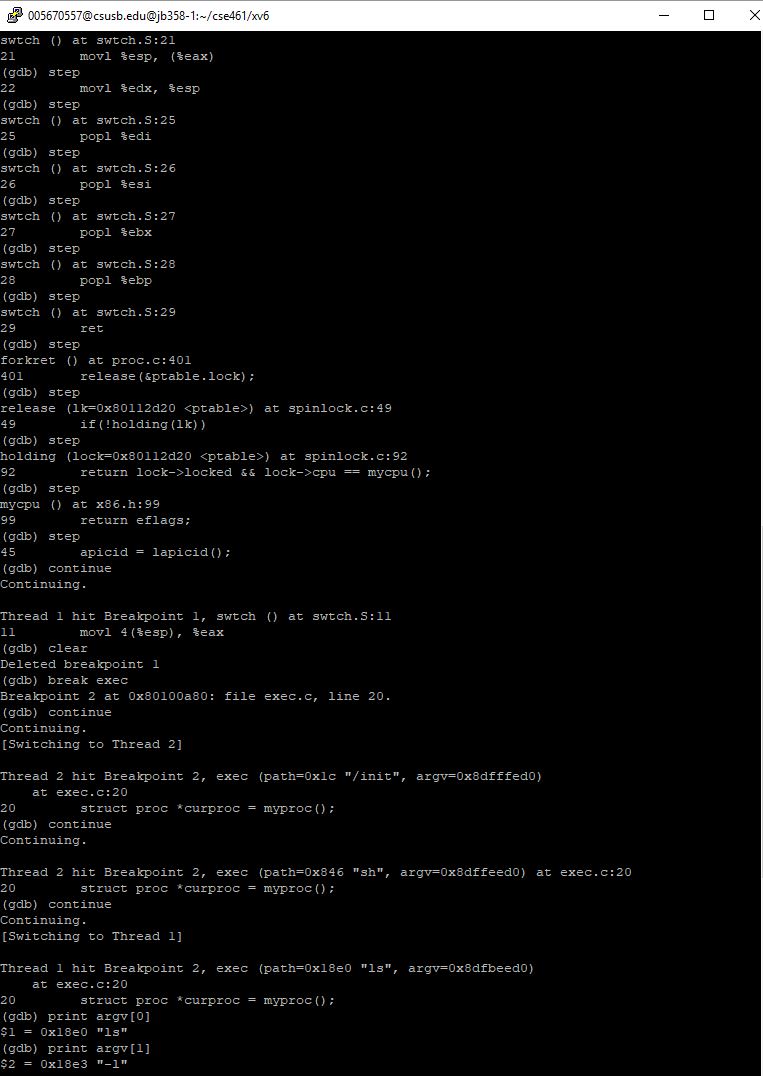
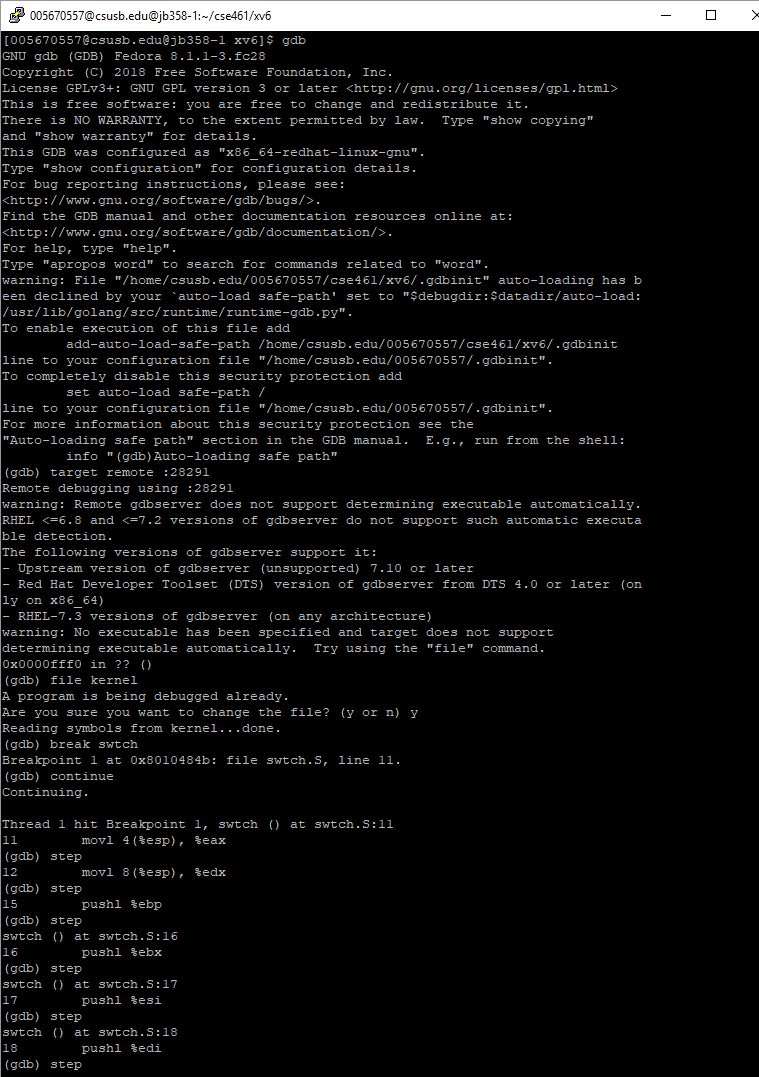
****

**Discussion**

This section of the lab is an introduction on how to use XV6 operating system. Screenshots 1-3 of this section demo commands while using qemu-nox. These are many of the typical commands that you can do through terminal such as ls, cat, grep, echo and file manipulation. I was able to successfully complete all instructions for this section. Screenshots 4-5 demo the qemu-nox-gdb which required using two terminals. I ran into a few problems with the gdb part because the tcp number would sometimes be clipped from the terminal, making it to where I couldn’t connect using the other terminal. This was most likely a problem using putty and required me to maximize the window prior to starting gdb. Expanding the window after still failed to show the clipped number. After a few tries, however, I was able to successfully complete this part as well.

1. **Debugging (Completed)**





**Discussion**

In this section I used gdb to debug the kernel file. As in the previous part, gdb was done on two terminals. The First screen shot is from one terminal and the rest are from the other. After connecting to the target remote, I was able to load the kernel file and set a break point at “swtch”. This caused the program to stop when “swtch” was reached and displayed the line of code at which it stopped. Using step, I could step through each line of code or continue until the break point is reached again. Old break points could be cleared, and new ones could be added while executing the debugger. Once the debugger reaches a point where it waits for user input, the debugger hangs up and the other terminal waits for input. The debugger also accepts commands for checking variable info. An example of this was printing the arguments for ls -l. You can also do back traces and list functions to get more detail of the running program. This is a useful tool which I have used in many IDE’s such as Visual Studio, which helps track how your program is being executed.

1. **Cp.c Copy One File to Two Others (Completed)**

#include "types.h"

#include "stat.h"

#include "user.h"

#include "fcntl.h"

char buf[512];

int

main(int argc, char \*argv[])

{

int fd0, fd1, fd2, i;

if(argc <= 2){

printf(1, "Need 2 Arguments!\n");

exit();

}

if((fd0 = open(argv[1], O\_RDONLY)) < 0)

{

printf(1, "cp: cannot open %s\n", argv[1]);

exit();

}

if((fd1 = open(argv[2], O\_CREATE|O\_RDWR)) < 0)

{

printf(1, "cp: cannot open %s\n", argv[2]);

exit();

}

if(argc > 3)

{

if((fd2 = open(argv[3], O\_CREATE|O\_RDWR)) < 0)

{

printf(1, "cp: cannot open %s\n", argv[3]);

exit();

}

}

while (( i = read ( fd0, buf, sizeof(buf))) > 0 )

{

write(fd1, buf, i);

if(argc > 3)

write(fd2,buf,i);

}

close(fd0);

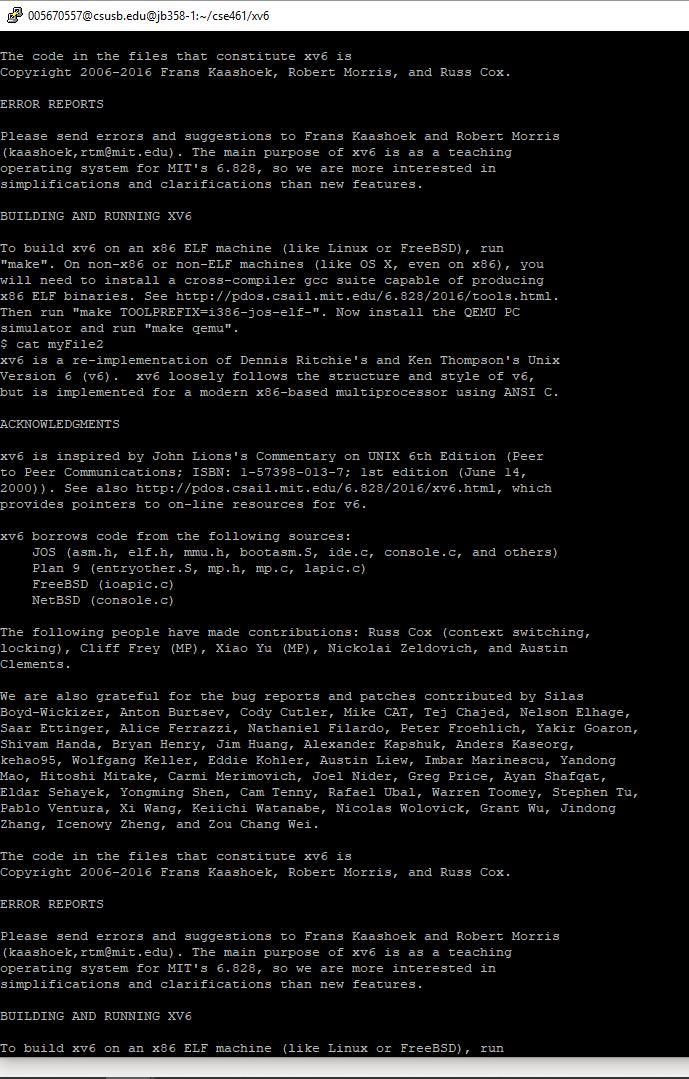
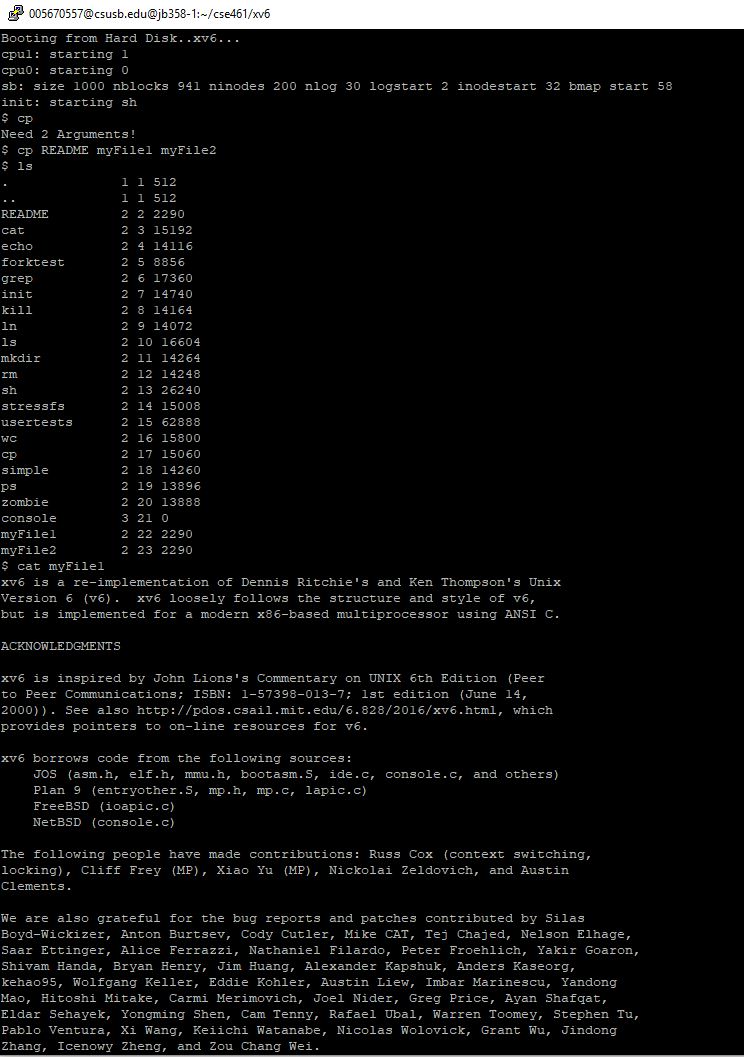
close(fd1);

if(argc > 3)

close(fd2);

exit();

}



**Discussion**

The cp.c file above can copy a file into one or two other files. By default, the command only requires two arguments. However, if a third argument is present, then the file will be copied into both files. This makes the code more dynamic as it can be used for more than one purpose. From the screenshots above, you can see that the README file was successfully copied to both myFile1 and myFile2.

**Self-Evaluation (20/20)**

I was able to complete all sections of this lab successfully. Not only was I able to complete them, but I fully understood how to use them and the concepts behind them. I have done similar projects in the past which included topics of shells and file copy commands, so the overall code parts were not too challenging. I have also had experience using a gdb with many of my school projects, so the debugger sections were not that bad as well. I do plan on continuing to explore XV6 to see if I can add more functions to the program. Overall, I think I did good on this lab.