COSC 439

Operating Systems

Fall 2019

**Lab Exercise #3**

*Think left and think right and think low and think high.*

*Oh, the thinks you can think up if only you try.*

* Dr. Seuss

**Name: Fola Alonge / 8**

**Goals:** The intention of this lab is to continue to better understand how an operating system boots up through the implementation of a really basic OS. We will expand upon this OS throughout the semester. This lab is modified from Chapter 3 of *The Little Book About OS Development* (see <https://ordoflammae.github.io/littleosbook/>). In addition, this lab will introduce/practice some basic C programming.

**Environment:** Throughout this lab we will use the Ubuntu operating system, Git/GitLab and a virtualization application (VirtualBox).

**Slack Channel:** All questions, problems and or announcements for this lab should be directed to the course’s Lab 3 discussion forum.

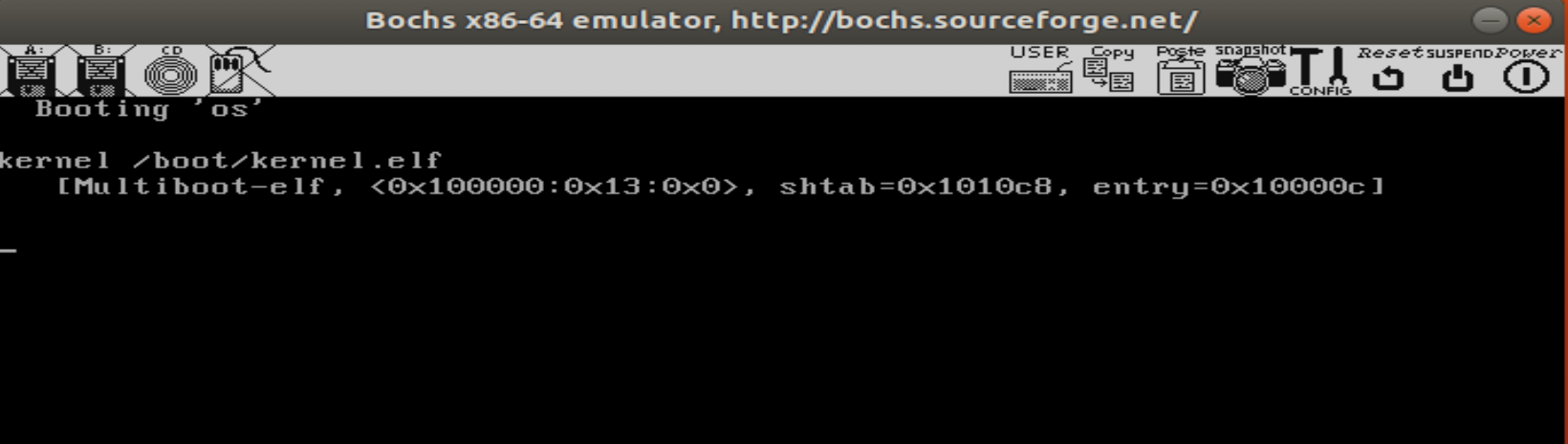
**Submission:** All files, including this document with the required screenshots, should be submitted via Blackboard and GitLab as indicated. ***Lab submissions not following this convention may not be graded.***

**Contents:** Submitted via GitLab with all source code and developed files checked into your GitLab repository. You should commit to a new repository name *cosc439*-*lastname-lab3* similar to the repository created in the previous lab.

**Preparation.** Review the Introduction to Unix, Common Unix Commands and Unix Command Cheat Sheet documents posted to [Blackboard](https://www.youtube.com/watch?v=iA3rIjM3Vds) under Lab 2.

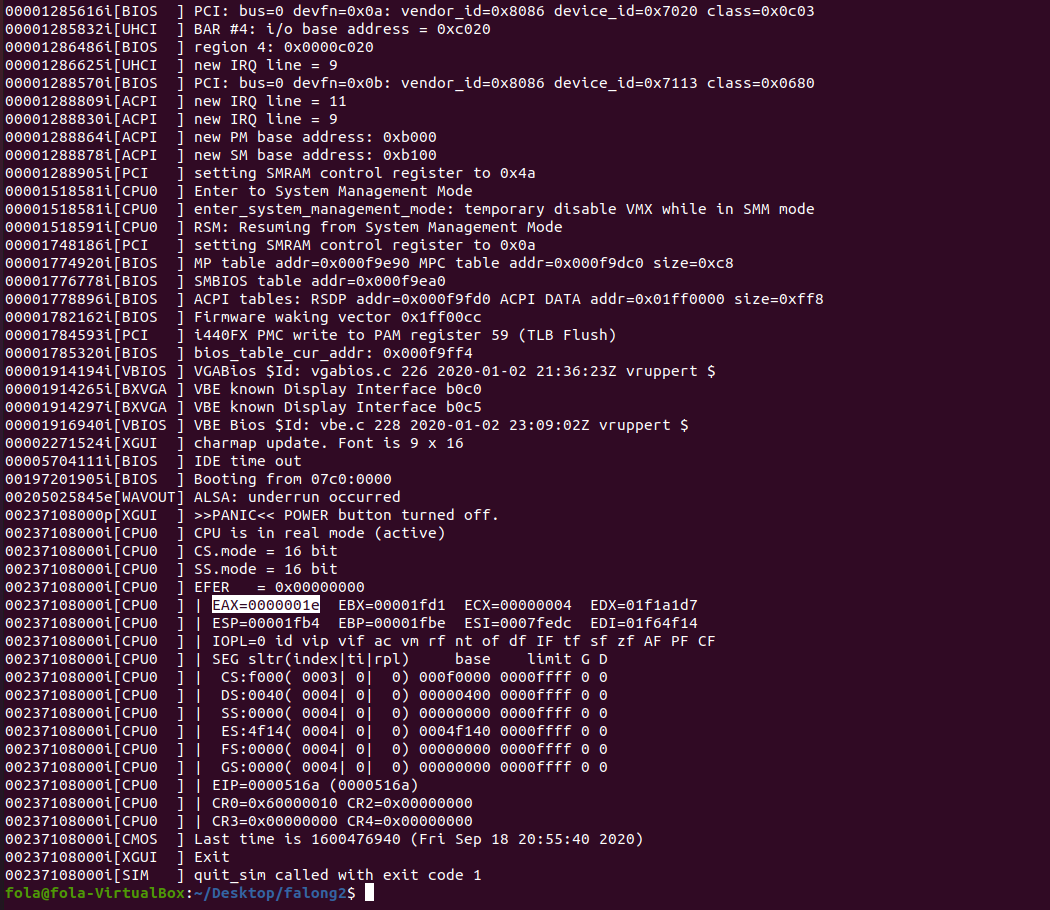
1. **Getting to C & Sum of Three.** (5 points) Read and complete Chapter 3 to improve our basic (*and mostly useless*) OS from Lab 2. Your resulting “operating system” will now be easier to build, can utilize C code and place the result of 1 + 2 + 3 into the EAX register. Begin by using the little operating system created in Lab 2.

When your implemented Chapter 3 OS is complete, you should be able to now boot it with the make run command and it will place the result of sum\_of\_three into the EAX register (viewed in boschlog.txt).



This chapter suffers from some details needed to correctly implement the OS. If you are struggling, see <https://github.com/littleosbook/littleosbook/issues/60>

Once you have successfully implemented this simple OS, you should commit and push your project to GitLab, with the commit comment “Chapter 3” and ***take a screenshot with the result of sum of three and paste it into this document, add the document to your repository, submit a copy of this document to the Blackboard Lab 3 assignment, include in the text a comment with the gitlab project URL.*** Without this, you will not receive credit for this portion of the lab.



1. **Introduction (or Review) of C Programming.** The traditional first program of any new programming language you learn is “Hello World!”. Within your Ubuntu virtual machine, create a new directory, named *cpractice*, within the Lab 3 working direction, open a text editor and create the Hello World C program from <https://www.programiz.com/c-programming/examples/print-sentence> and compile it at the terminal using the instructions from <https://www.wikihow.com/Compile-a-C-Program-Using-the-GNU-Compiler-(GCC)>. After completing each practice program, ***push the source and makefile to your Lab 3 repository, add a screenshot of the successful execution of each program to this document, making sure to push the completed document to your Lab 3 repository.*** Without this, you will not receive credit for this portion of the lab.

Every full C program begins inside a function called "main". The main function is always called when the program first executes. To access the standard functions that comes with your compiler, you need to include a header files with the #include directive.

C uses the following standard data types:

* int integer
* short short integer
* long long integer
* float single precision real
* double double precision real
* char character

The printffunction is the standard C way of displaying output on the screen. The general syntax is printf( "format", variables ); The printf function uses its first argument, “format”, to determine how many arguments will follow and of what types they are. If you don’t use enough arguments or if they are of the wrong type, printf will get confused and produce wrong answers. Example formats for printf include:

* %d (print as a decimal integer)
* %6d (print as a decimal integer with a width of at least 6 wide)
* %e (the scientific notation format)
* %f (print as a floating point)
* %4f (print as a floating point with a width of at least 4 wide)
* %.4f (print as a floating point with a precision of four characters after the decimal point)
* %3.2f (print as a floating point at least 3 wide and a precision of 2)
* %c (print as a character)
* %s (print as a string)
* %u (print as an unsigned integer)
* %o (the unsigned octal format)
* %x (the unsigned hexadecimal format)
* \n (escape sequence -- new line)
* \t ( tab)

**Example:**

#include <stdio.h> int main ()

{

/\* variable definition: \*/

int a, b;

int c;

float f;

/\* actual initialization \*/

a = 10;

b = 20;

c = a + b;

printf("value of c : %d \n", c);

f = 70.0/3.0;

printf("value of f : %f \n", f);

return 0;

}

**Program 1.** (1 point) Write, compile and run a program, named prog1.c and saved to the cpractice directory, which displays an integer value in different formats (e.g. integer, float, octal, hexadecimal, etc.).

In C, every memory location has a *value* and a *unique address*. Declaring a variable simply sets up a memory location. For example:

* int alpha; // allocates space in memory for sizeof int, at an address
* alpha = 6; // assigns a value to memory location

Pointers point to locations in memory. The pointer declaration looks like this:

<variable\_type> \*<name>;

For example, you could declare a pointer that stores the address of an integer with the following syntax:

int \*int\_ptr;

float \*ftPtr;

To get the memory address of a variable (i.e., its location in memory), we use the address of operator “&” – it returns the memory address.

int i = 0;

int\* iptr;

iptr = NULL;

iptr = &i; //pointer contains address

The content of the memory location referenced by a pointer is obtained using the ``\*'' operator, this is called *dereferencing* the pointer. Thus, \*iptr refers to the value of x. The following two assignment statements produce the same result.

i = 40;

\*iptr = 40;

For example:

#include <stdio.h>

|  |  |
| --- | --- |
| void main()  { |  |
| int x; | /\* A normal integer\*/ |
| int \*xp;    x=100; | /\* A pointer to an integer |
| xp = &x; | /\* assigns the address of x to xp" \*/ |

printf( " value of x = %d\n", x );

// Note the use of the \* to get the value

printf( " value of x = %d\n", \*xp );

printf(“ address of x =%8x \n”, xp);

}

Another example:

void main()

{

float x, y; /\* x and y are of float type \*/

float \*fp, \*fp2; /\* fp and fp2 are pointers to float \*/

x = 6.5; /\* x now contains the value 6.5 \*/

/\* print contents and address of x \*/

printf("Value of x is %f, address of x %8x\n", x, &x);

fp = &x; /\* fp now points to location of x \*/

/\* print the contents of fp \*/

printf("Value in memory location fp(%8x) is %f\n", \*fp);

/\* change content of memory location \*/

\*fp = 9.2;

printf("New value of x is %f = %f \n", \*fp, x);

/\* perform arithmetic \*/

\*fp = \*fp + 1.5;

printf("Final value of x is %f = %f \n", \*fp, x);

/\* transfer values \*/

y = \*fp;

fp2 = fp;

printf("Transfered value into y = %f and fp2 = %f \n", y, \*fp2);

}

An array name is simply a constant pointer and points to the location of the first element of the array.

#include <stdio.h>

void main()

{

int arr[10]; int i;

for (i=0; I <10; i++) arr[i] = 10\*i;

printf(“ address of arr[0] = %8x \n”, &arr[0]);

printf(“ address of arr[0] = %8x \n”, arr);

int \*temp = arr;

for (i=0; I <10; i++)

printf(“%d \t”, \*temp++);

printf(“\n”);

}

# Pointer arithmetic is possible in C where the pointer is incremented/decremented according to the size of the object it points to. For example:

|  |  |  |
| --- | --- | --- |
| int i,\*ptr = &i; | // ptr contains address 100 | |
| ptr + 1 | // 100 + 1 \* sizeof(int) = 104 | |
| ptr + 3 | // 100 + 3 \* sizeof(int) = 112 | |
| ptr - 2 | // 100 - 2 \* sizeof(int) = 92 | |
|  | |

**Program 2.** (1 point) Write, compile and run a program, named prog2.c and saved to the cpractice directory, which develops code to answer the following questions (provide your answers to the multiple choice questions in comments in the code):

1. Which of the following is the proper declaration of a pointer?

1. int x;
2. int &x;
3. ptr x;
4. int \*x;

2. Which of the following gives the memory address of integer variable a?

1. \*a;
2. a;
3. &a;
4. address(a);

3. Which of the following gives the value stored at the address pointed to by pointer a?

1. a;
2. val(a);
3. \*a;
4. &a;

In C, the scanf() function is used to read character, string, numeric data from stdin.

int scanf(format string, argument);

It reads data and stores them according to the parameter *format* into the locations *pointed* by the additional arguments. The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

int main()

{

int ID;

char name[20];

printf("Enter your ID: ");

scanf("%d", &ID);

printf(“Enter you name: “);

scanf(“%s”, name); //remember that array name is a pointer.

printf("Entered ID:%d\n", ID);

printf(“Entered Name: %s\n”, name);

return(0);

}

Because C is not an object-oriented language, structures provide a way of storing many different values in variables of potentially different types under the same name.

struct struct\_name{

Members

};

Where struct\_name is the name of the entire type of structure and members are the variables within the struct. The syntax for creating a single structure is:

struct struct\_name name\_of\_single\_structure;

To access a variable of the structure, we can use the dot ‘.’ operator.

For example:

struct EmployeeType

{

int id\_number;

char name[50];

int age;

float salary;

};

int main() {

struct EmployeeType emp1, emp2;

emp1.age = 22;

emp1.id\_number = 1;

emp1.salary = 12000.0;

emp1.name =”John Doe”;

emp2.age = 45;

…..

}

We can have array of structures and well as nested structures. If you wish to have a pointer to a structure, to actually access the information stored inside the structure that is pointed to, you use the -> operator in place of the . operator. All points about pointers still apply.

int main()

{

struct EmployeeType emp1;

struct EmployeeType \*emp\_ptr;

emp\_ptr= &emp1;

emp1.name = “Jane Doe”;

emp\_ptr-> age = 25;

printf( "%d\n", );

return 0;

}

**Program 3.** (1 point) Write, compile and run a program, named prog3.c and saved to the cpractice directory, which develops code to answer the following questions (provide your answers to the multiple choice questions in comments in the code):

1. Which of the following accesses a variable in structure b?

1. b->var;
2. b.var;
3. b-var;
4. b>var;

2. Which of the following accesses a variable in a pointer to a structure, \*b?

1. b->var;
2. b.var;
3. b-var;
4. b>var;

3. Which of the following is a properly defined struct?

1. struct {int a;}
2. struct a\_struct {int a;}
3. struct a\_struct int a;
4. struct a\_struct {int a;};

4. Which properly declares a variable of struct foo?

1. struct foo;
2. struct foo var;
3. foo;
4. int foo;

