**Lab 2: ( <https://github.com/Its-Masoom/su21-lab-starter> )**

**Exercise 1:**

Action Item:

Below is my code for exercise3 make file

CC=gcc

CFLAGS=-Wall -std=c99

all: bit\_ops

bit\_ops: bit\_ops.o test\_bit\_ops.o

$(CC) -o bit\_ops bit\_ops.o test\_bit\_ops.o

bit\_ops.o: bit\_ops.c bit\_ops.h

$(CC) $(CFLAGS) -c bit\_ops.c

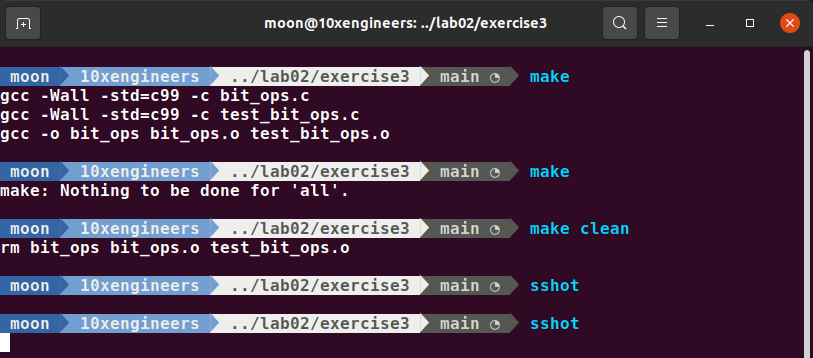
test\_bit\_ops.o: test\_bit\_ops.c bit\_ops.h

$(CC) $(CFLAGS) -c test\_bit\_ops.c

clean:

rm bit\_ops bit\_ops.o test\_bit\_ops.o

Here is the terminal output



Below is my code for exercise4 make file

CC=gcc

CFLAGS=-Wall -std=c99

all: vector

vector: vector.o test\_vector.o

$(CC) -o vector vector.o test\_vector.o

vector.o: vector.c vector.h

$(CC) $(CFLAGS) -c vector.c

test\_vector.o: test\_vector.c vector.h

$(CC) $(CFLAGS) -c test\_vector.c

clean:

rm vector vector.o test\_vector.o

Here is the terminal output

****

**Exercise 2: Valgrind**

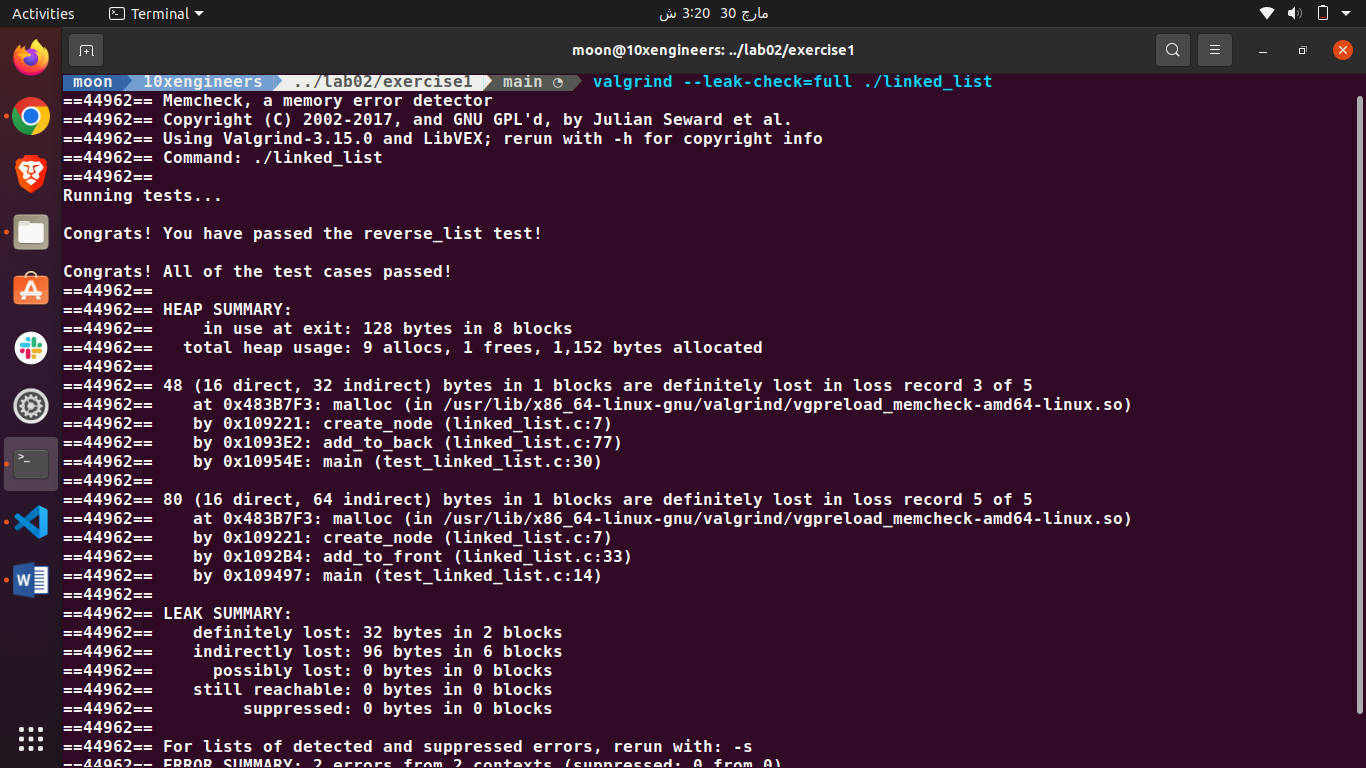
**Using Valgrind to find segfaults**

I have installed Valgrind. Run it according to the instructions given in the manual. Copy the solution from lab1. Again running it with valgrind gives me no error



**Using Valgrind to detect memory leaks**

I have removed two free statements as mentioned in the manual to check memory leaks and then run valgrind with –leak-check=full flag to see the information about the memory leaks



**Exercise 3: Bit Operations**

Below are the codes for the functions as required in the manual. I have explained each line of code with a comment. So you can understand what is going on in the program.

**get\_bit function:**

/\* Returns the Nth bit of X. Assumes 0 <= N <= 31. \*/

unsigned get\_bit(unsigned x, unsigned n) {

/\* YOUR CODE HERE \*/

// x >> n will right shift x by a factor of n so now we have our required value at 0th bit

unsigned shift = x >> n;

// Taking & with 1 give us the value at 0th bit

unsigned req\_bit = shift & 1;

return req\_bit;

}

**set\_bit function:**

/\* Set the nth bit of the value of x to v. Assumes 0 <= N <= 31, and V is 0 or 1 \*/

void set\_bit(unsigned \*x, unsigned n, unsigned v) {

/\* YOUR CODE HERE \*/

// Firstly there should be zero at the nth bit

// So we will create mask on the required bit if n = 2 mask will be 100

unsigned mask = 1 << n;

// Taking not(~) because we want the index value of required bit to be 0 and all other indexes value 1

// if n = 2 we will have n\_mask = 011

unsigned n\_mask = ~mask;

// if n\_mask = 011

// if \*x = 1001110

// unset\_bit = 1001010

unsigned unset\_bit = n\_mask & \*x;

// Shifting v by the value of n so if v = 1 then set\_bit = 100 and if v = 0 then set\_bit = 000

unsigned set\_bit = v << n;

// Taking or with unset\_bit

//

\*x = unset\_bit | set\_bit;

}

**flip\_bit function:**

/\* Flips the Nth bit in X. Assumes 0 <= N <= 31.\*/

void flip\_bit(unsigned \*x, unsigned n) {

/\* YOUR CODE HERE \*/

// We will create mask on the required bit if n = 2 mask will be 100

unsigned mask = 1 << n;

// Taking xor with mask will flip the bit

// if mask = 100

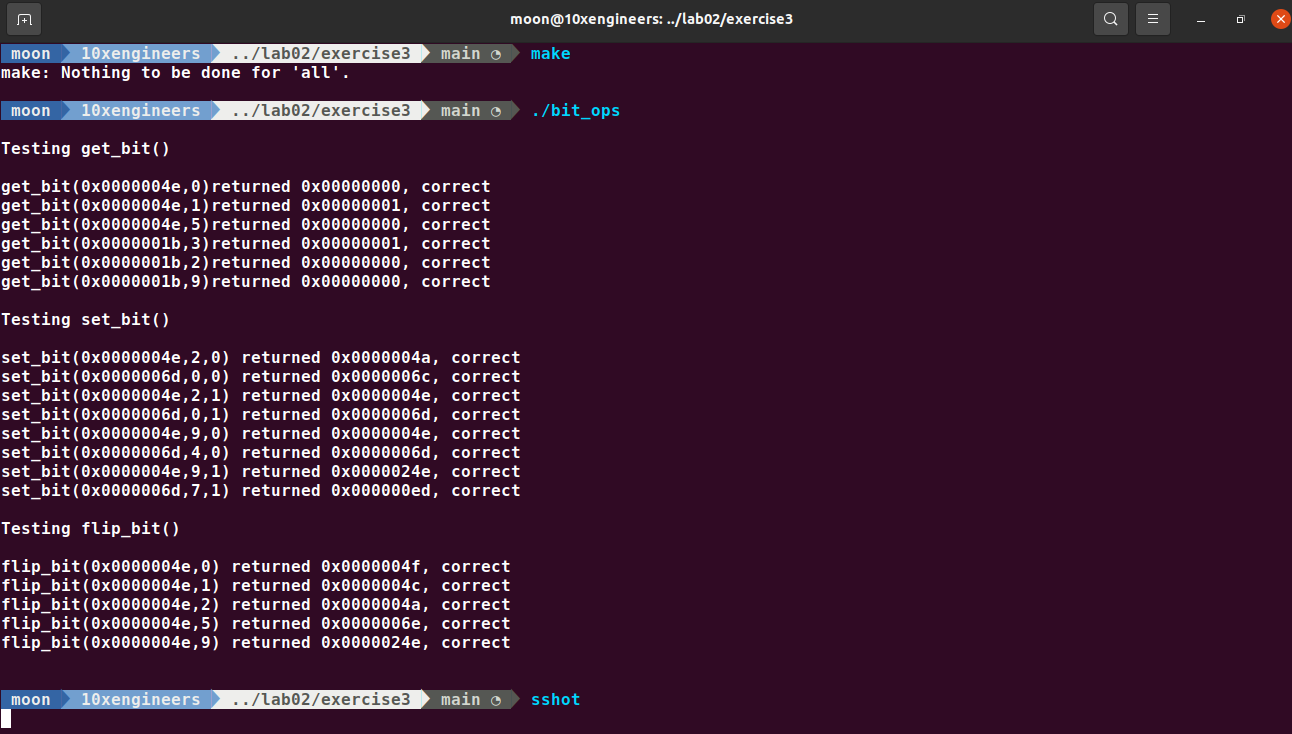
// \*x = 1001110

// xor = 1001010

\*x = (\*x)^mask;

}

**Output of above three functions:**

****

**Exercise 4: Memory Management**

**Action Item**

**1):** For checkoff, be prepared to explain why bad\_vector\_new() and also\_bad\_vector\_new() are bad.

**bad\_vector\_new** is bad because this creates struct v and pointer to this struct v (i.e retval)

Now the problem is that when at the end when function return retval, the variable v is destroyed. Because it was created in the stack and the variables at the stack are destroyed when the function returns. So v is destroyed (in other words the struct is destroyed) and retval is pointing to v. There is no v so now retval becomes a dangling pointer that is pointing to anywhere not known causing the undefined behavior of the program which can lead to crashes.

**also\_bad\_vector\_new** is bad because it is not using pointers and return vector\_t object by value. So it will create a copy of v when it returns. When v is very large, it will certainly slow down the program. Also if you want to change the value of v struct after it is returned, then any modification made to this copy will not effect the original struct that was created inside the function that is the disadvantage of pass by value.

**2):** Fill in the functions vector\_new(), vector\_get(), vector\_delete(), and vector\_set() in vector.c so that our test code test\_vector.c runs without any memory management errors.

I have fill all the functions. You can find them in my github repo [here](https://github.com/Its-Masoom/su21-lab-starter/blob/main/lab02/exercise4/vector.c)

**3):** Test your implementation of vector\_new(), vector\_get(), vector\_delete(), and vector\_set() for both correctness and memory management.

