Lab 4

Humam Amouri & Yonis Said

The purpose of this lab is to implement a read/ write buffer which can operate on text files. The read/ write buffers will be then used to implement a copy program which can duplicate text files.

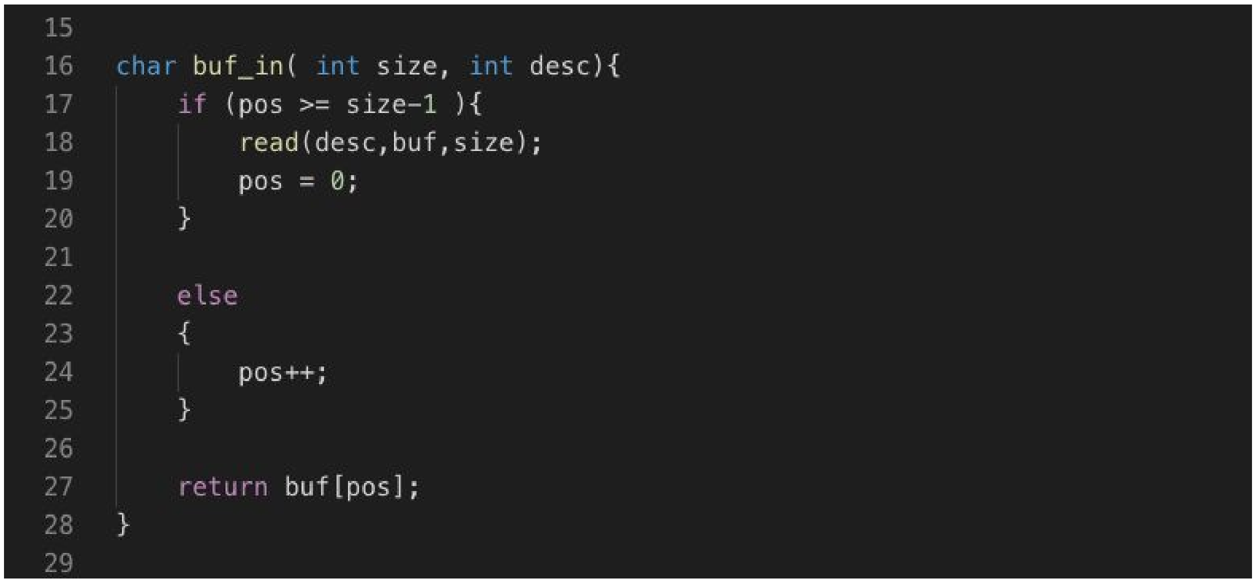
Visual studio code is used to write the code in this lab instead of QEMU because it offers the exact same terminal functionality without the hassle of using the text editor NANO.

All code in this lab is compiled with an “ansi” compile flag. It’s important to note that there are two global variables defined through all the lab the first is “Buf” which has the type char pointer while the other is “size” which is used to acquire the current position in the buffer.

Task 1: Buffered input

The first task is centered around writing a read buffer program using a function called **buf_in**. The program uses a 16-byte character array as a buffer which the function then uses to store the data it reads from the input text file. **Buf_in** should read 16 bytes from the input file and save them in the buffer and on subsequent calls it should return the next byte in the buffer.

The following code snippet shows how the function **buf_in** is implemented:

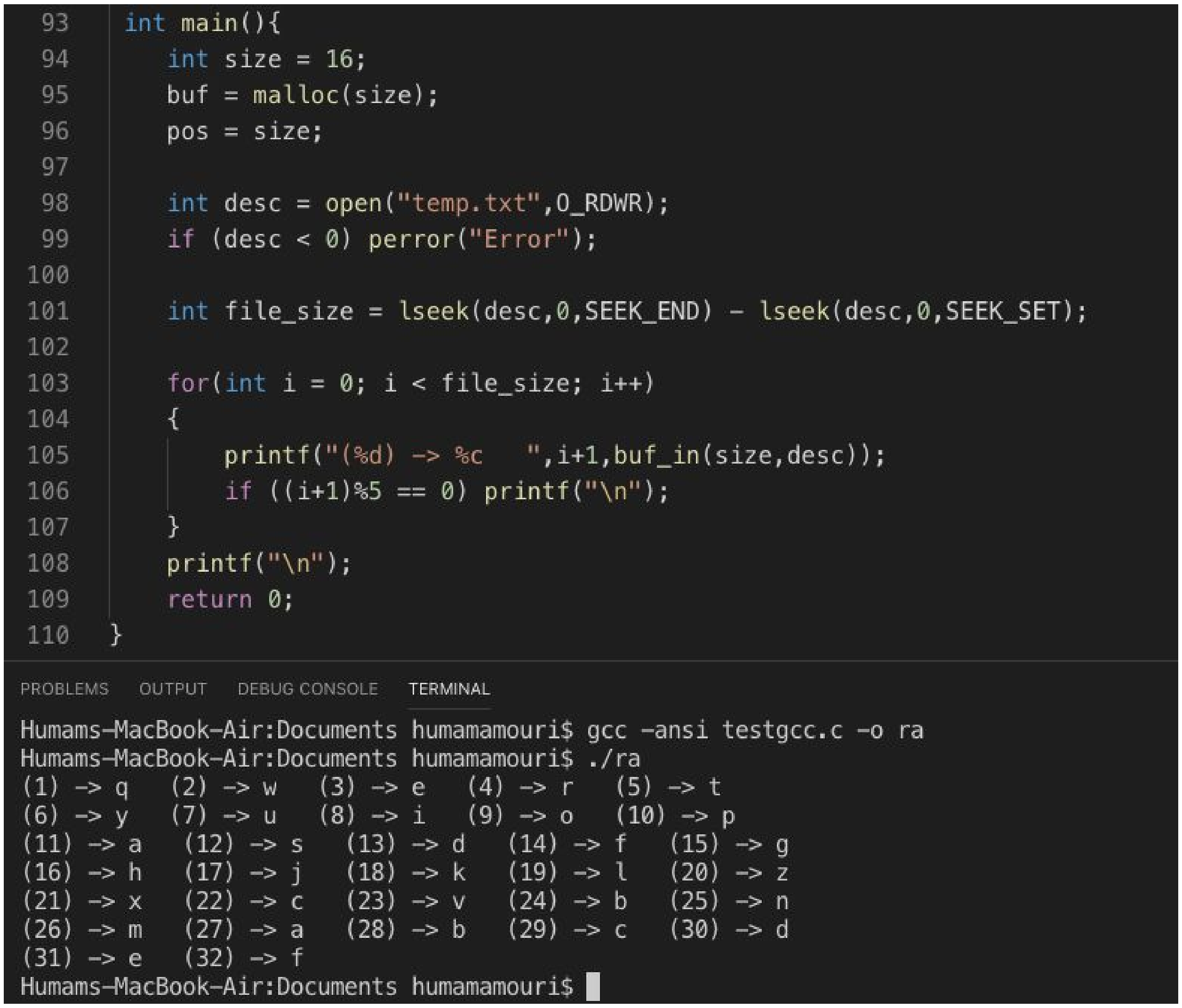


```

char buf_in( int size, int desc){
    if (pos >= size-1 ){
        read(desc,buf,size);
        pos = 0;
    }

    else
    {
        pos++;
    }

    return buf[pos];
}

```

Code snippet 1

Buf_in checks if the buffer is full by comparing the current position to the buffer length. When the position is equal to the buffer length the position is reset to zero and a system call is made to import data into the buffer. The new data in the buffer will be eventually returned every time the function is called by incrementing the actual position. If we assume that we’re reading from a 32-byte file, then this means that we only make two system calls to read from the file and we fetch 30 times from our buffer.

The following code snippet shows how main function looks like and what the output of the function became:



```c
int main(){
   int size = 16;
   buf = malloc(size);
   pos = size;

   int desc = open("temp.txt",O_RDWR);
   if (desc < 0) perror("Error");

   int file_size = lseek(desc,0,SEEK_END) - lseek(desc,0,SEEK_SET);

   for(int i = 0; i < file_size; i++)
   {
       printf("(%d) -> %c   ",i+1,buf_in(size,desc));
       if ((i+1)%5 == 0) printf("\n");
   }
   printf("\n");
   return 0;
}
```

```
Humams-MacBook-Air:Documents humamamouri$ gcc -ansi testgcc.c -o ra
Humams-MacBook-Air:Documents humamamouri$ ./ra
(1) -> q   (2) -> w   (3) -> e   (4) -> r   (5) -> t
(6) -> y   (7) -> u   (8) -> i   (9) -> o   (10) -> p
(11) -> a   (12) -> s   (13) -> d   (14) -> f   (15) -> g
(16) -> h   (17) -> j   (18) -> k   (19) -> l   (20) -> z
(21) -> x   (22) -> c   (23) -> v   (24) -> b   (25) -> n
(26) -> m   (27) -> a   (28) -> b   (29) -> c   (30) -> d
(31) -> e   (32) -> f
Humams-MacBook-Air:Documents humamamouri$
```

Code snippet 2

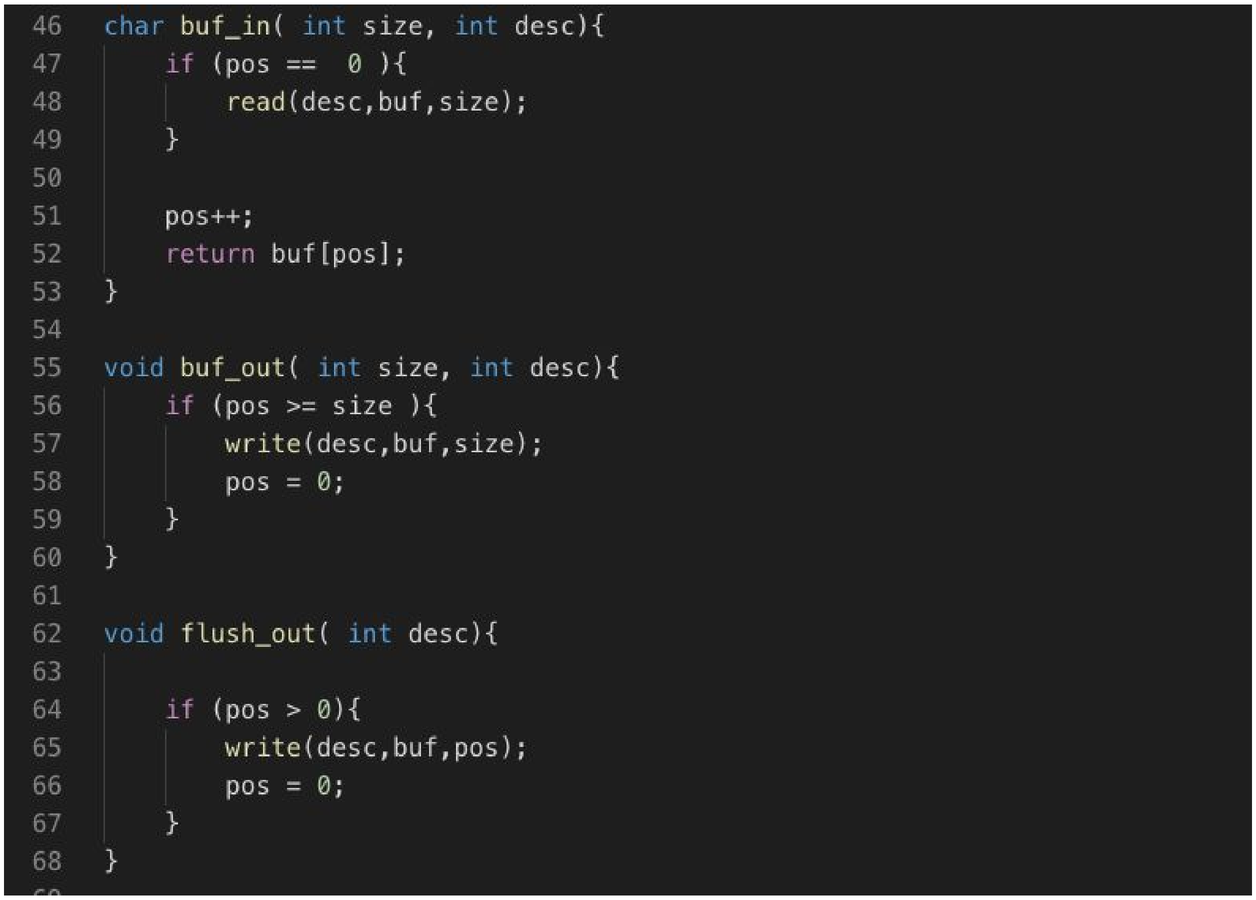
The file size is 32 bytes and **buf_in** is called the same number of times.
Note that even though the buffer is not full when we start, we set pos to 16. This is only used in the beginning to the function to make a system call and to fill the buffer.

The architecture of this function is based on the assumption that we don't have an output buffer which will reset the position. In task 2 **buf_in** will be changed to simulate a consecutive write/ read process.

Task 2: Buffered output

Because it’s hard to simulate the difference in functionality between **buf_out** and **flush_out** the function **buf_in** was included in this task to give a better idea of how the output will be.

The second task is in a sense the opposite of task 1. What should be accomplished is a write buffer which is then used to write to a file. When the buffer is full a function called **buf_out** should be called, this function writes all the contents which are stored in the buffer to an output text file. An additional function **buf_flush** is also implemented in case the buffer is not full and the user wants to force-output the actual data in the buffer to a file. Code snippet 3 below shows how **buf_in, buf_out** and **flush_out** are implemented in C code:

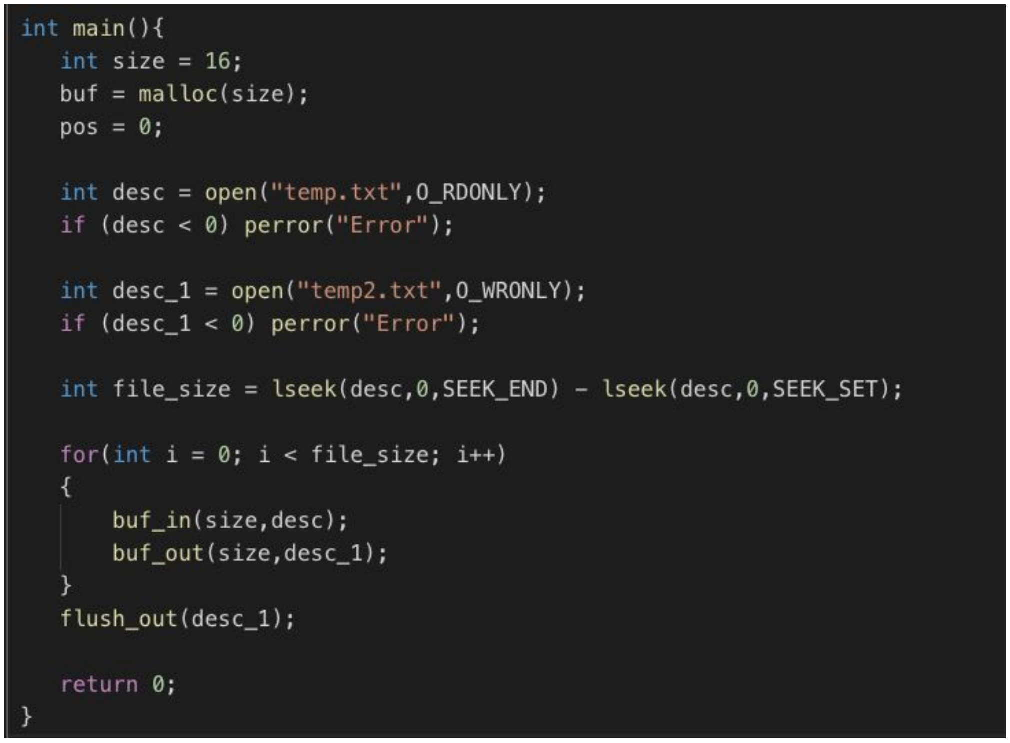


```
char buf_in( int size, int desc){
    if (pos ==  0 ){
        read(desc,buf,size);
    }

    pos++;
    return buf[pos];
}

void buf_out( int size, int desc){
    if (pos >= size ){
        write(desc,buf,size);
        pos = 0;
    }
}

void flush_out( int desc){

    if (pos > 0){
        write(desc,buf,pos);
        pos = 0;
    }
}
```

Code snippet 3

The changes that were made to **buf_in** means in general that **buf_out** will be resetting the actual position assuming that the position starts from zero. This can be seen as if **buf_out** is emptying the buffer after it outputs the buffer’s data. By looking at the code above one can see that the noticeable difference between **buf_out** and **flush_out** is that the first function, **buf_out**, checks if the buffer is full to output while the second function, **flush_out**, outputs regardless of how full the buffer is.

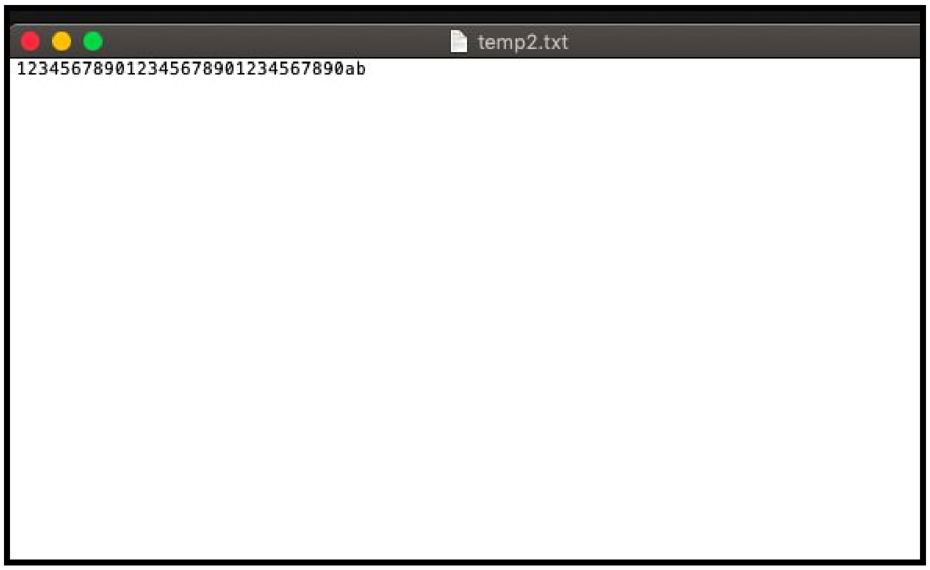
The following code snippet shows how main function looks like:



```
int main(){
   int size = 16;
   buf = malloc(size);
   pos = 0;

   int desc = open("temp.txt",O_RDONLY);
   if (desc < 0) perror("Error");

   int desc_1 = open("temp2.txt",O_WRONLY);
   if (desc_1 < 0) perror("Error");

   int file_size = lseek(desc,0,SEEK_END) - lseek(desc,0,SEEK_SET);

   for(int i = 0; i < file_size; i++)
   {
       buf_in(size,desc);
       buf_out(size,desc_1);
   }
   flush_out(desc_1);

   return 0;
}
```

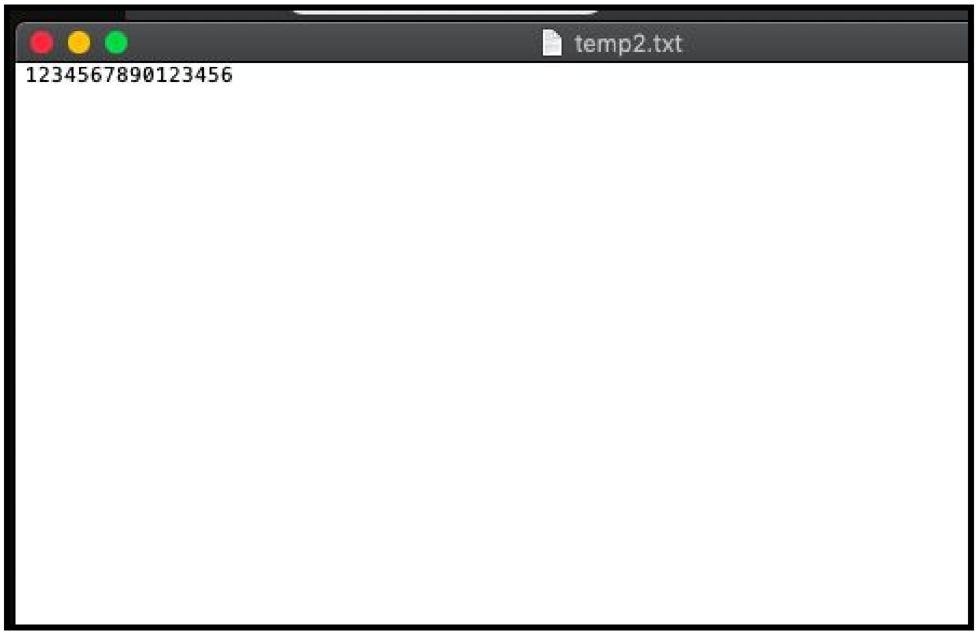
Code snippet 4

Two documents are opened the first one is full while the second is empty. Starting from the position zero we want to read from the first file “temp.txt” and write to the second file “temp2.txt”. Even though the for-loop above loops through the whole file, the function **buf_out** does the actual work only when the buffer is full. The output of this operation can be seen in the snippets below:



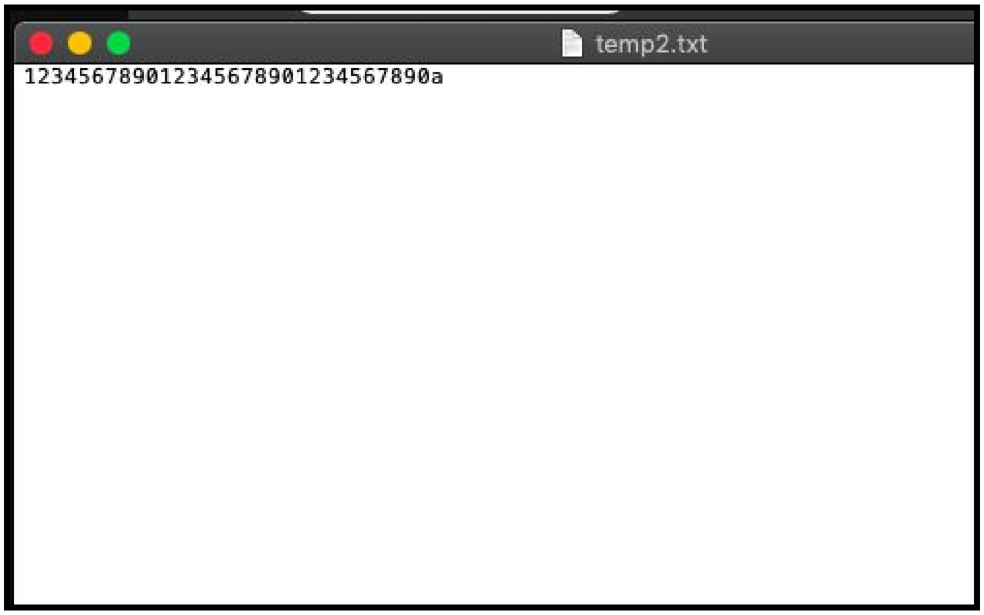
```
123456789012345678901234567890ab
```

Notice that when we the loop in Code snippet 4 is run, 31 times instead of 32 **buf_out** will only write the first 16 bytes because the buffer isn't full (see snippet below for output).



```
1234567890123456
```

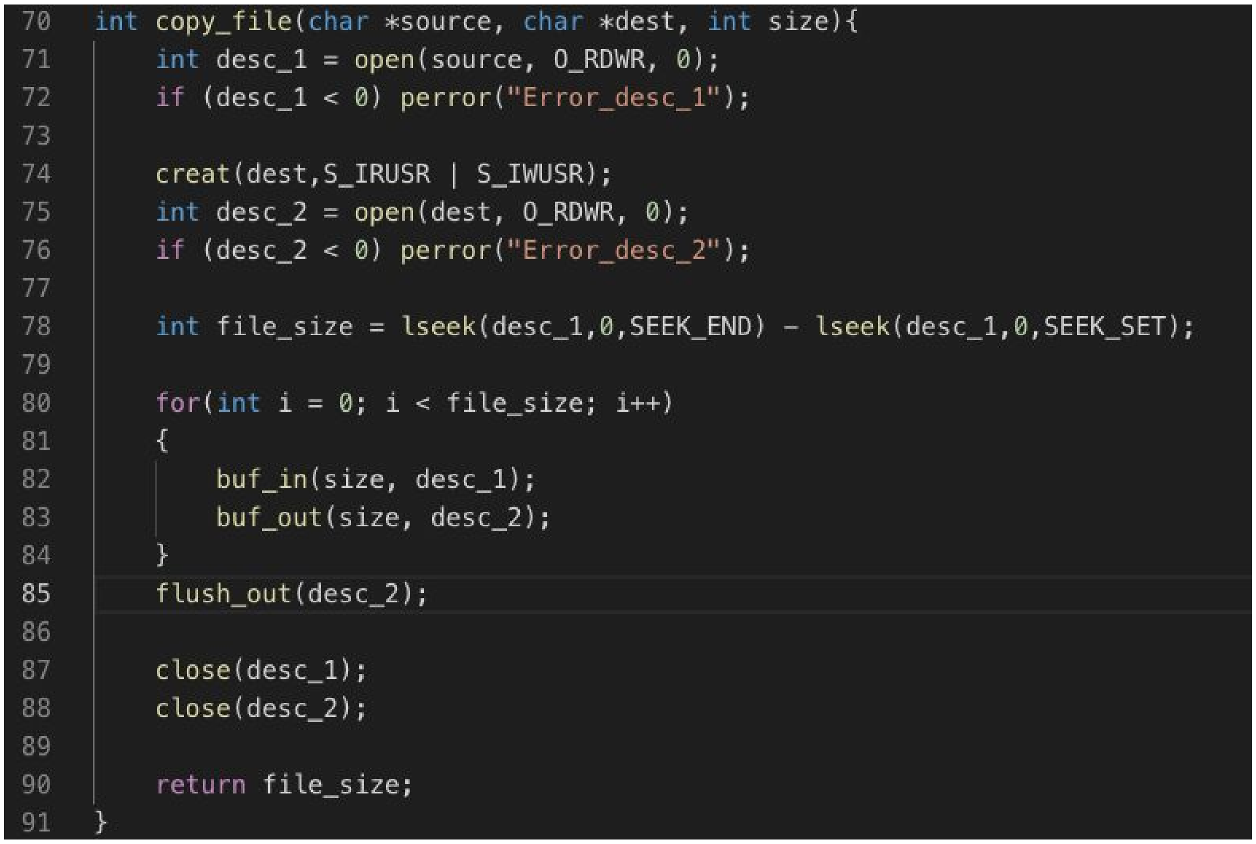
This is when flush_out comes in handy. If we want to force the output out of the buffer we can flush everything out using **flush_out**. In this specific example after **buf_out** wrote the first 16 bytes, we have a 16-byte buffer full with only 15 byes and when **flush_out** is run we get an output of 15 extra characters as below:



```
123456789012345678901234567890a
```

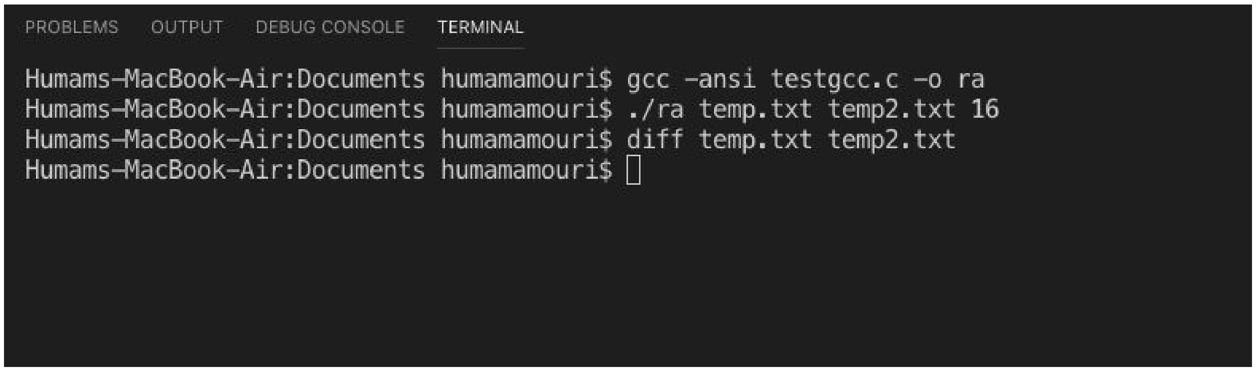
Task 3: Performance evaluation

We start by writing the function for copy a text file to a new text file which is based on the test done in task 2. We first need to open 2 channels for the source file and the destination file. Note that we don't need to create the file ourselves. If we open a channel to a file "dest", it will be automatically created. So what we do is that we use "buf_in" to fill the buffer with ***b*** bytes from the source file and use the "buf_out" function to write it out in the destination file. We loop those two steps enough that we should cover the size of the source file. And just to make sure we have all the bytes we use the function "flush_out" to print out the remaining bytes to the destination file. And we end it with closing the channels to the files. And if we check our current directory the copied file should be there. And we can confirm that it's indeed a copy by using the shell command "diff" (see code snippet 5, 6).



```c
int copy_file(char *source, char *dest, int size){
    int desc_1 = open(source, O_RDWR, 0);
    if (desc_1 < 0) perror("Error_desc_1");

    creat(dest,S_IRUSR | S_IWUSR);
    int desc_2 = open(dest, O_RDWR, 0);
    if (desc_2 < 0) perror("Error_desc_2");

    int file_size = lseek(desc_1,0,SEEK_END) - lseek(desc_1,0,SEEK_SET);

    for(int i = 0; i < file_size; i++)
    {
        buf_in(size, desc_1);
        buf_out(size, desc_2);
    }
    flush_out(desc_2);

    close(desc_1);
    close(desc_2);

    return file_size;
}
```

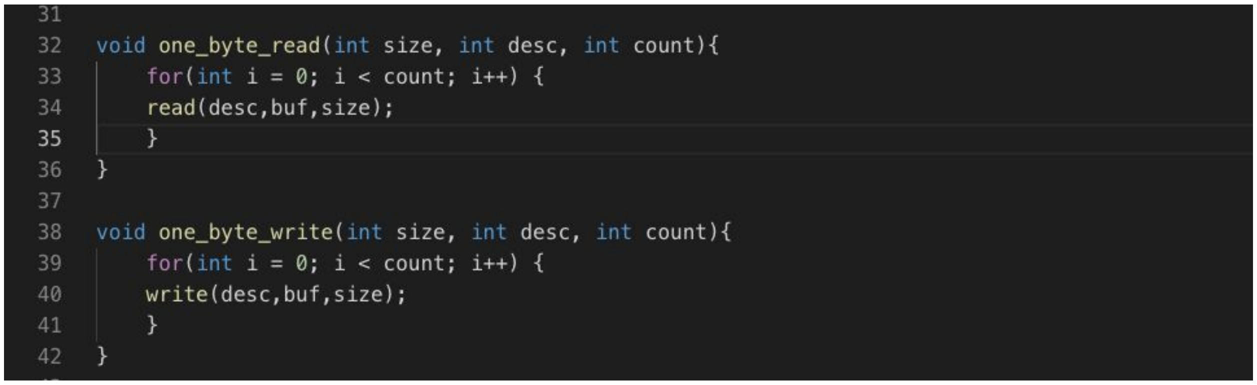
Code snippet 5



```
Humams-MacBook-Air:Documents humamamouri$ gcc -ansi testgcc.c -o ra
Humams-MacBook-Air:Documents humamamouri$ ./ra temp.txt temp2.txt 16
Humams-MacBook-Air:Documents humamamouri$ diff temp.txt temp2.txt
Humams-MacBook-Air:Documents humamamouri$ 
```

Code snippet 6

What is requested next in task 3 is to write a program which can measure the performance of reading/ writing one byte and the average performance of this process. To implement an easy solution for both problems two test functions, **one_byte_read** and **one_byte_write**, are written (see code snippet 7). The main characteristic of these two functions is to be unbuffered thus resulting in many system calls and inefficiency .

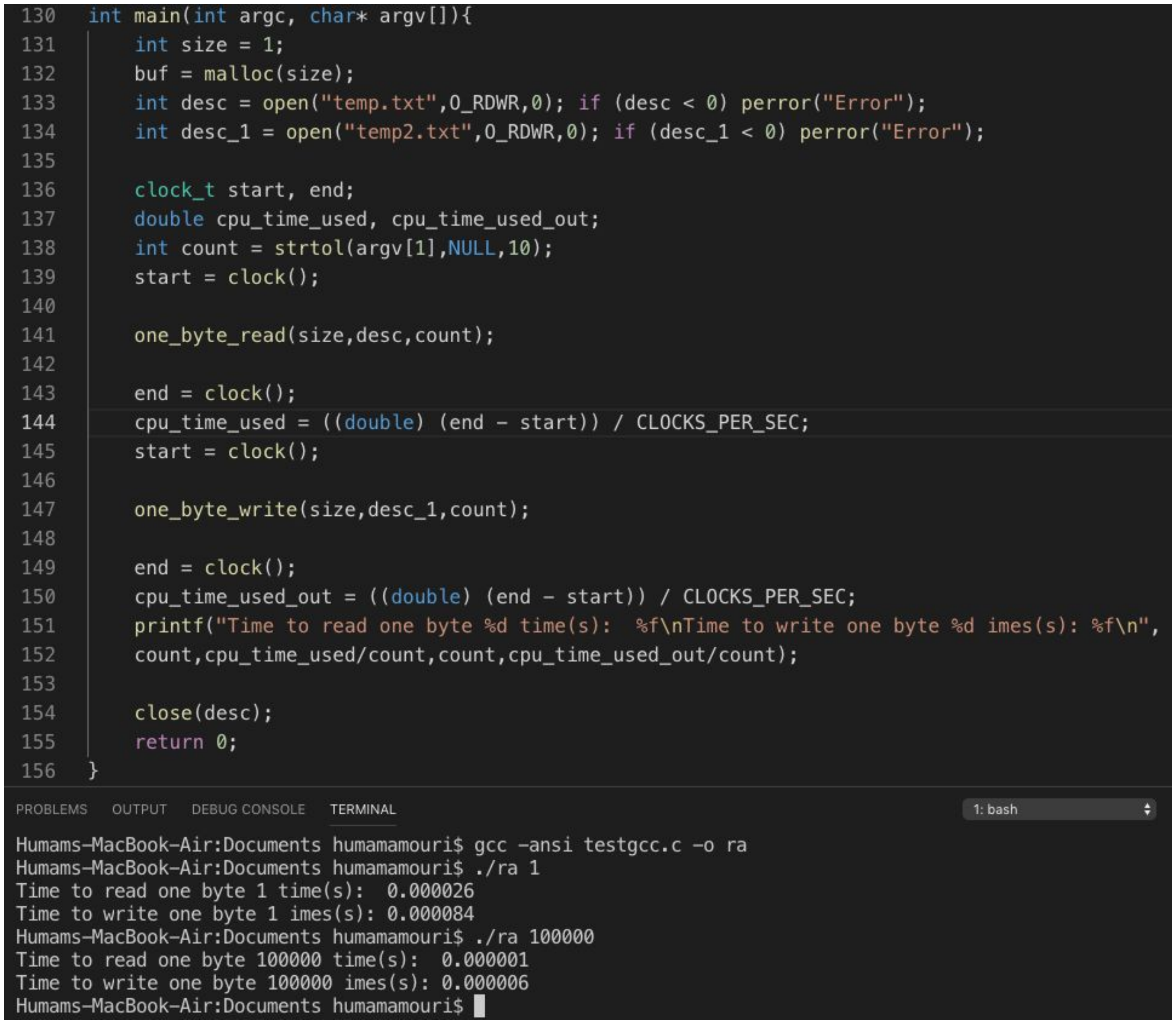


```c
void one_byte_read(int size, int desc, int count){
    for(int i = 0; i < count; i++) {
    read(desc,buf,size);
    }
}

void one_byte_write(int size, int desc, int count){
    for(int i = 0; i < count; i++) {
    write(desc,buf,size);
    }
}
```

Code snippet 7

To find the time the structure “clock_t” is used and the buffer size is set to 1. So after that we only need to use a few lines of code to find the time it takes to read and write 1 byte. We start by initializing “start” and “end” variables with the “clock_t” data type. We define the “start” variable as the current time. We run the two test functions, **one_byte_read** and **one_byte_write**, and define the “end” variable as the time right after the two function calls. The performance of the measured function would be the time difference between the “start” and the “end” variables. For calculating the average times “count” is set to 100 000 , this means that the final value is divided with 100 000 too.

The output from code snippet 8 shows how the average value is much faster than the individual value:

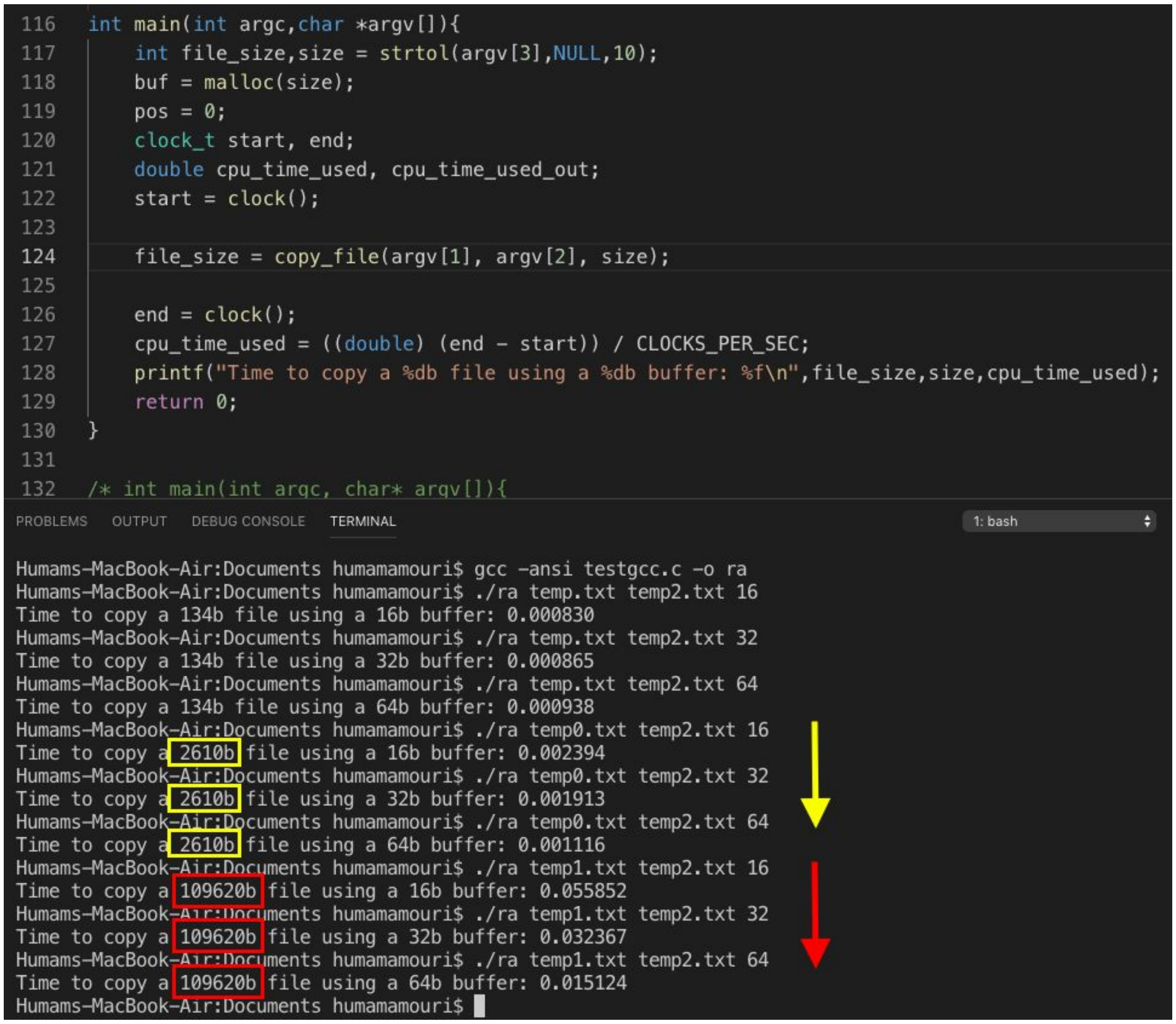


```c
int main(int argc, char* argv[]){
    int size = 1;
    buf = malloc(size);
    int desc = open("temp.txt",O_RDWR,0); if (desc < 0) perror("Error");
    int desc_1 = open("temp2.txt",O_RDWR,0); if (desc_1 < 0) perror("Error");

    clock_t start, end;
    double cpu_time_used, cpu_time_used_out;
    int count = strtol(argv[1],NULL,10);
    start = clock();

    one_byte_read(size,desc,count);

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    start = clock();

    one_byte_write(size,desc_1,count);

    end = clock();
    cpu_time_used_out = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time to read one byte %d time(s):  %f\nTime to write one byte %d imes(s): %f\n",
    count,cpu_time_used/count,count,cpu_time_used_out/count);

    close(desc);
    return 0;
}
```

```
Humams-MacBook-Air:Documents humamamouri$ gcc -ansi testgcc.c -o ra
Humams-MacBook-Air:Documents humamamouri$ ./ra 1
Time to read one byte 1 time(s):  0.000026
Time to write one byte 1 imes(s): 0.000084
Humams-MacBook-Air:Documents humamamouri$ ./ra 100000
Time to read one byte 100000 time(s):  0.000001
Time to write one byte 100000 imes(s): 0.000006
Humams-MacBook-Air:Documents humamamouri$
```

Code snippet 8

In the same way the function **copy_file**'s performance is measured. But this time the size of both the buffer and text file are varied. Three different buffer sizes are chosen: 16, 32, and 64. And three randomly generated file sizes are used: 134 bytes, ~2,6 kilo bytes, and ~110 kilo bytes. Code snippet 9 below shows that for small file sizes (134 bytes) the performance difference caused by increasing the buffer size is not so noticable. While in larger files marked in yellow and red below, the benefits of using a bigger buffer are extremely clear, especially when looking at the largest file in red.

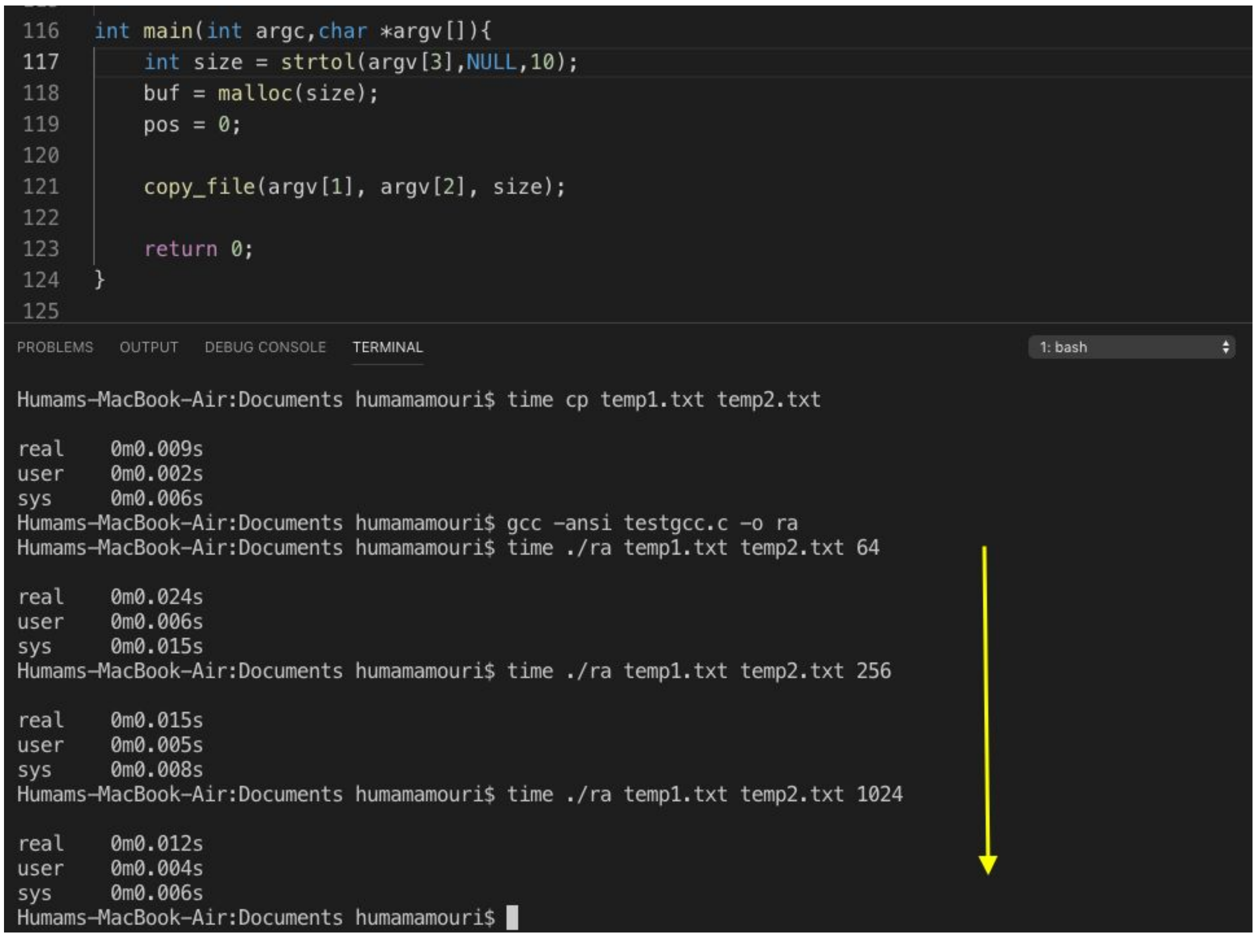


```c
int main(int argc,char *argv[]){
    int file_size,size = strtol(argv[3],NULL,10);
    buf = malloc(size);
    pos = 0;
    clock_t start, end;
    double cpu_time_used, cpu_time_used_out;
    start = clock();

    file_size = copy_file(argv[1], argv[2], size);

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time to copy a %db file using a %db buffer: %f\n",file_size,size,cpu_time_used);
    return 0;
}

/* int main(int argc, char* argv[]){
```

```
Humams-MacBook-Air:Documents humamamouri$ gcc -ansi testgcc.c -o ra
Humams-MacBook-Air:Documents humamamouri$ ./ra temp.txt temp2.txt 16
Time to copy a 134b file using a 16b buffer: 0.000830
Humams-MacBook-Air:Documents humamamouri$ ./ra temp.txt temp2.txt 32
Time to copy a 134b file using a 32b buffer: 0.000865
Humams-MacBook-Air:Documents humamamouri$ ./ra temp.txt temp2.txt 64
Time to copy a 134b file using a 64b buffer: 0.000938
Humams-MacBook-Air:Documents humamamouri$ ./ra temp0.txt temp2.txt 16
Time to copy a 2610b file using a 16b buffer: 0.002394
Humams-MacBook-Air:Documents humamamouri$ ./ra temp0.txt temp2.txt 32
Time to copy a 2610b file using a 32b buffer: 0.001913
Humams-MacBook-Air:Documents humamamouri$ ./ra temp0.txt temp2.txt 64
Time to copy a 2610b file using a 64b buffer: 0.001116
Humams-MacBook-Air:Documents humamamouri$ ./ra temp1.txt temp2.txt 16
Time to copy a 109620b file using a 16b buffer: 0.055852
Humams-MacBook-Air:Documents humamamouri$ ./ra temp1.txt temp2.txt 32
Time to copy a 109620b file using a 32b buffer: 0.032367
Humams-MacBook-Air:Documents humamamouri$ ./ra temp1.txt temp2.txt 64
Time to copy a 109620b file using a 64b buffer: 0.015124
Humams-MacBook-Air:Documents humamamouri$
```

Code snippet 9

The final requirement in task 3 was to compare the performance of the copy program with the built-in shell command “cp”. What can be noticed by observing the output in code snippet 10 is that the command’s performance is fast. But testing the copy program written in this program with larger buffers, one can notice that the speed converges to match cp’s. The reason why “cp” is this fast is because of its buffer size, which is relatively enormous compared to the buffer sizes used in code snippet 10 below.



```
int main(int argc,char *argv[]){
    int size = strtol(argv[3],NULL,10);
    buf = malloc(size);
    pos = 0;

    copy_file(argv[1], argv[2], size);

    return 0;
}
```

```
Humams-MacBook-Air:Documents humamamouri$ time cp temp1.txt temp2.txt

real	0m0.009s
user	0m0.002s
sys	0m0.006s
Humams-MacBook-Air:Documents humamamouri$ gcc -ansi testgcc.c -o ra
Humams-MacBook-Air:Documents humamamouri$ time ./ra temp1.txt temp2.txt 64

real	0m0.024s
user	0m0.006s
sys	0m0.015s
Humams-MacBook-Air:Documents humamamouri$ time ./ra temp1.txt temp2.txt 256

real	0m0.015s
user	0m0.005s
sys	0m0.008s
Humams-MacBook-Air:Documents humamamouri$ time ./ra temp1.txt temp2.txt 1024

real	0m0.012s
user	0m0.004s
sys	0m0.006s
Humams-MacBook-Air:Documents humamamouri$ 
```

Code snippet 10