Joseph Bentivegna

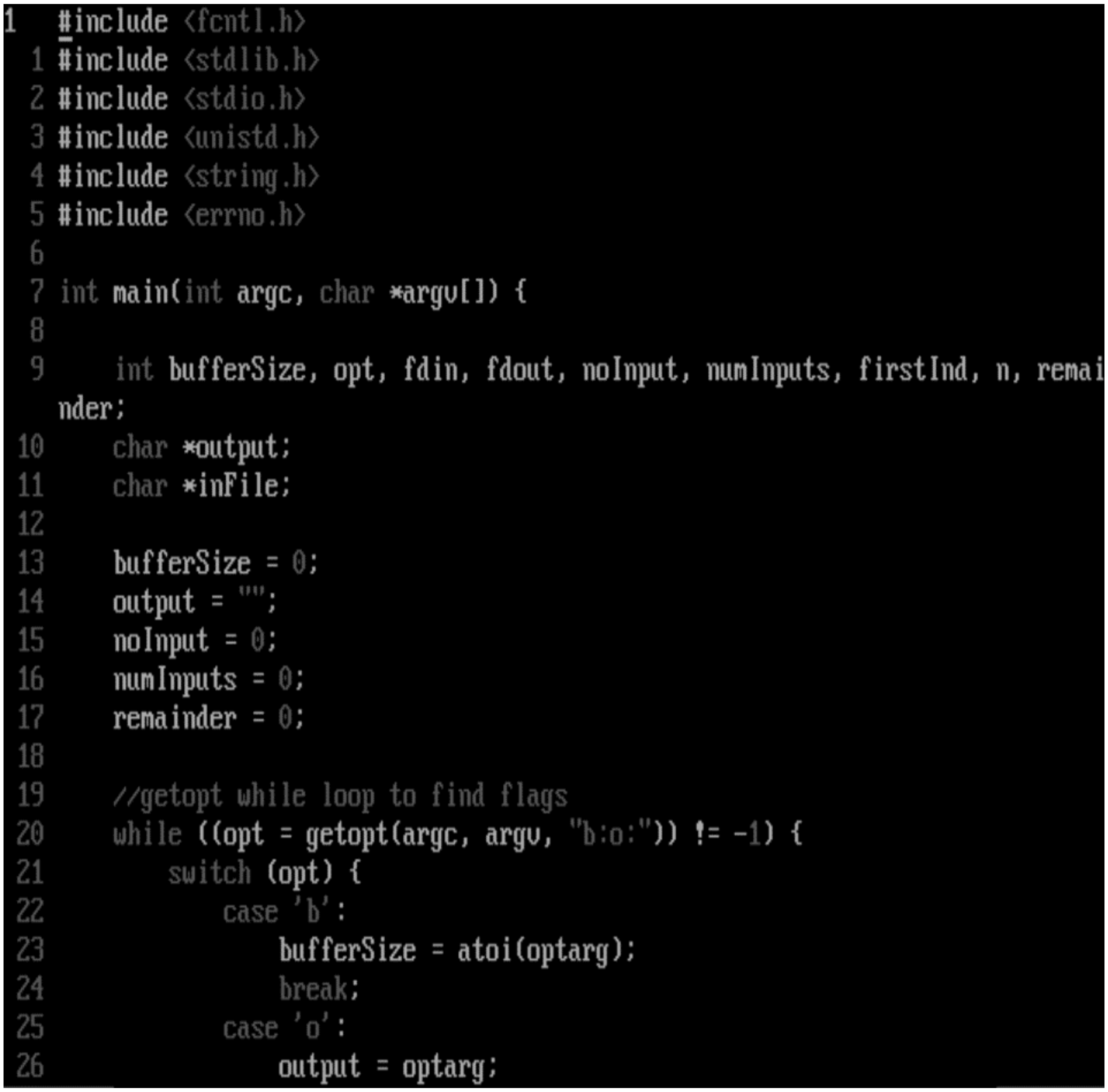
Professor Hakner

ECE 357

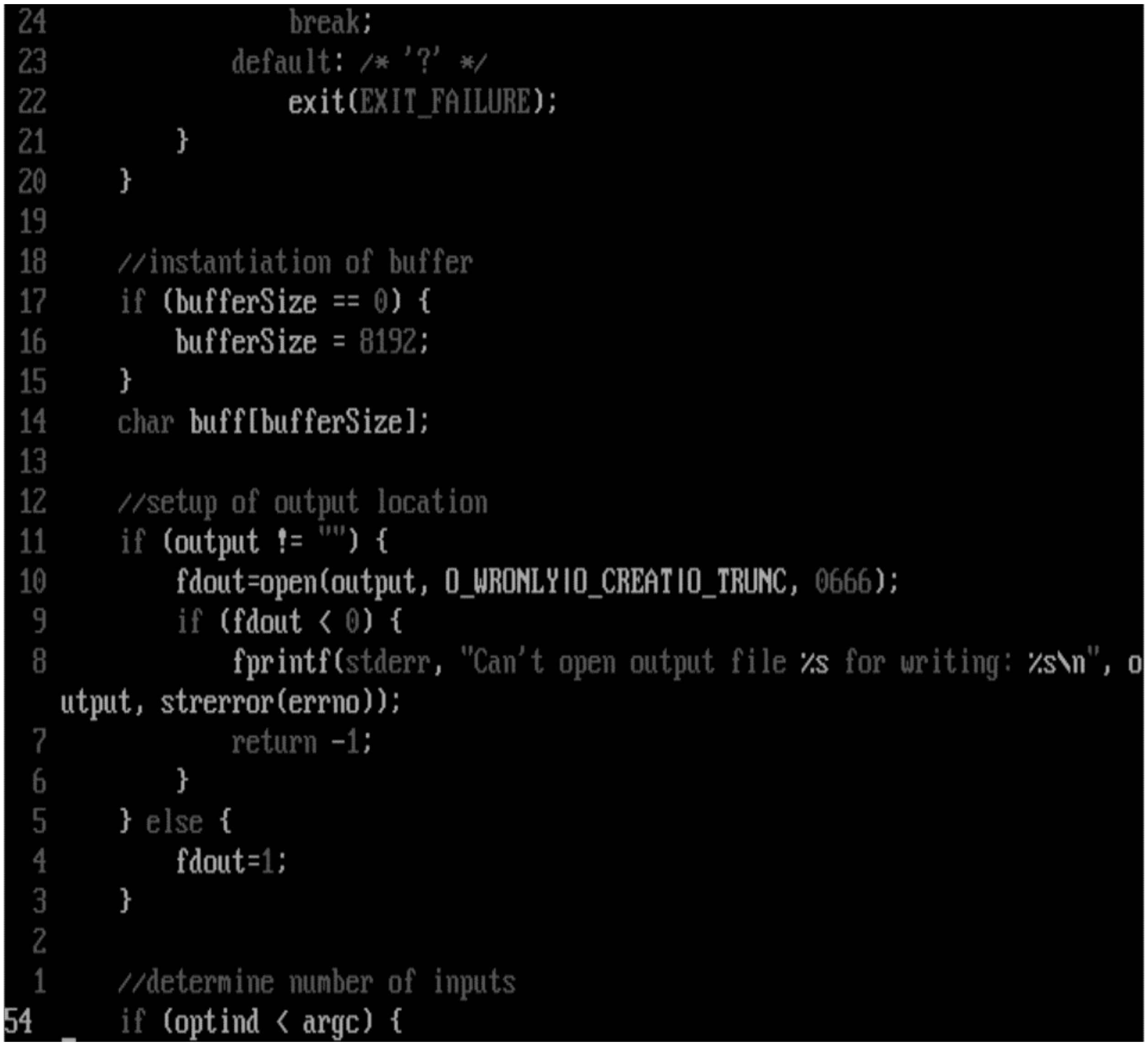
9/23/2017

Problem 1- System Calls, Error Checking and Reporting

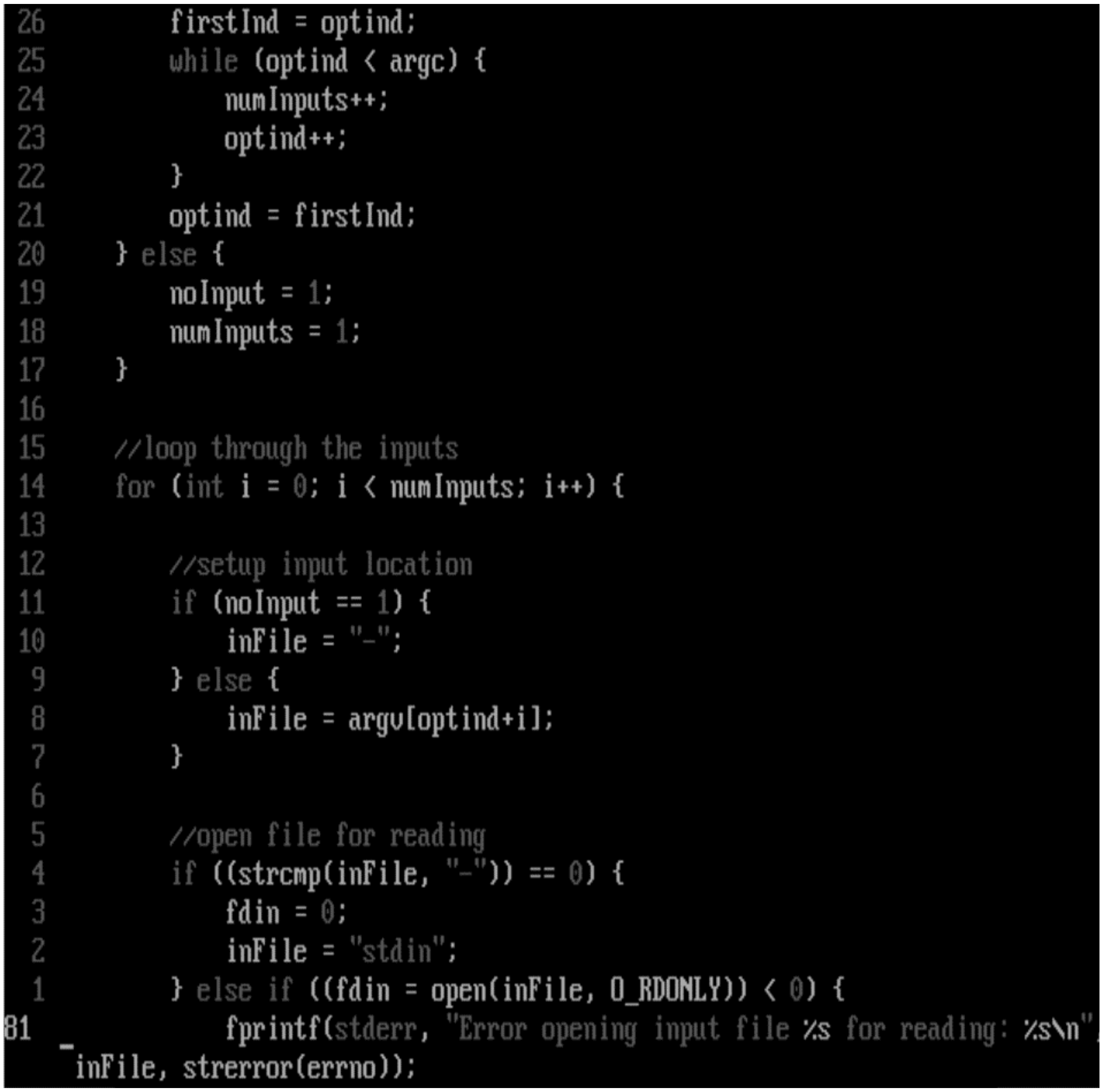
Part 1: Source Code (117 lines)



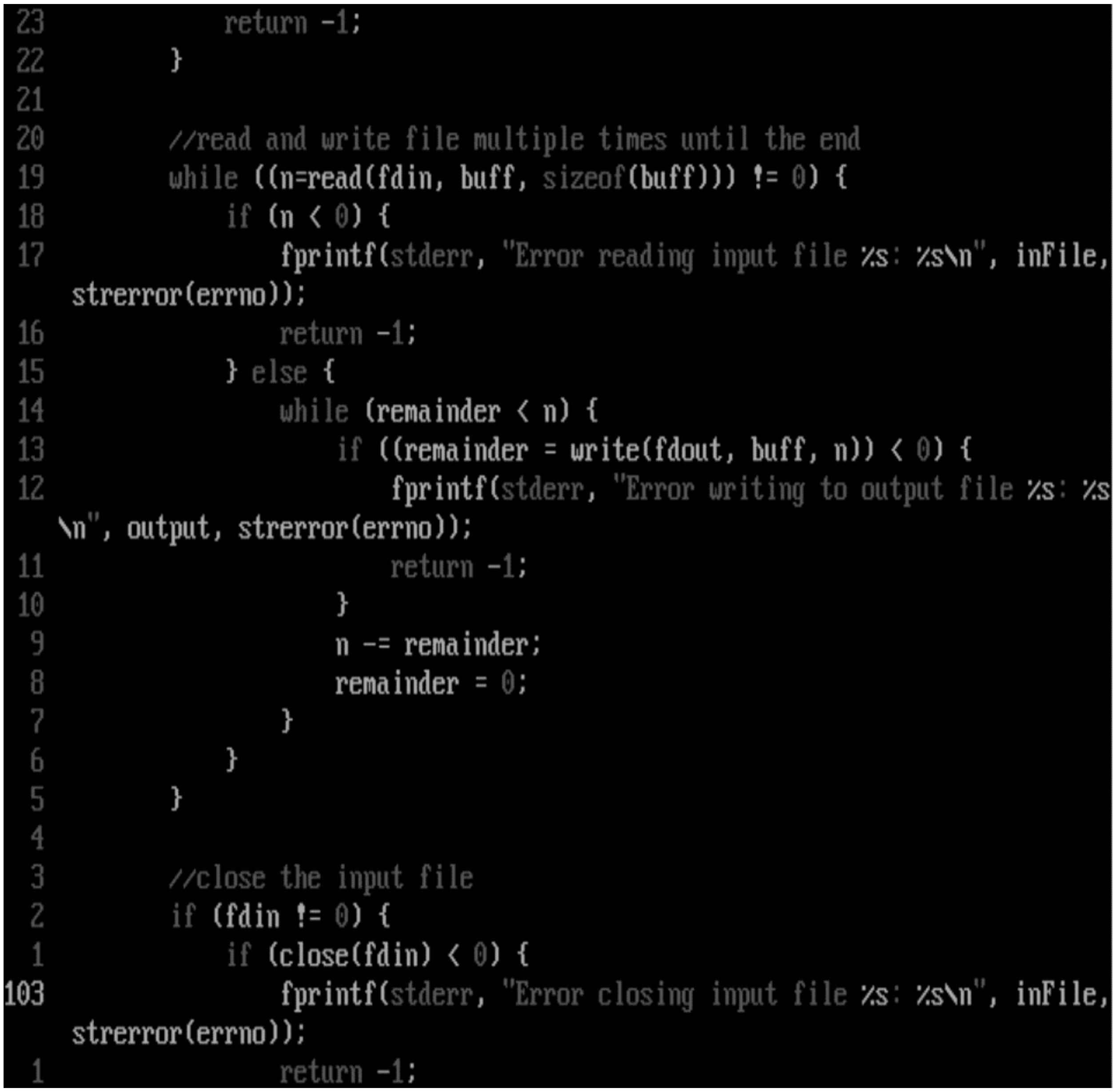
```c
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

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

    int bufferSize, opt, fdin, fdout, noInput, numInputs, firstInd, n, remainder;
    char *output;
    char *inFile;

    bufferSize = 0;
    output = "";
    noInput = 0;
    numInputs = 0;
    remainder = 0;

    //getopt while loop to find flags
    while ((opt = getopt(argc, argv, "b:o:")) != -1) {
        switch (opt) {
            case 'b':
                bufferSize = atoi(optarg);
                break;
            case 'o':
                output = optarg;
```



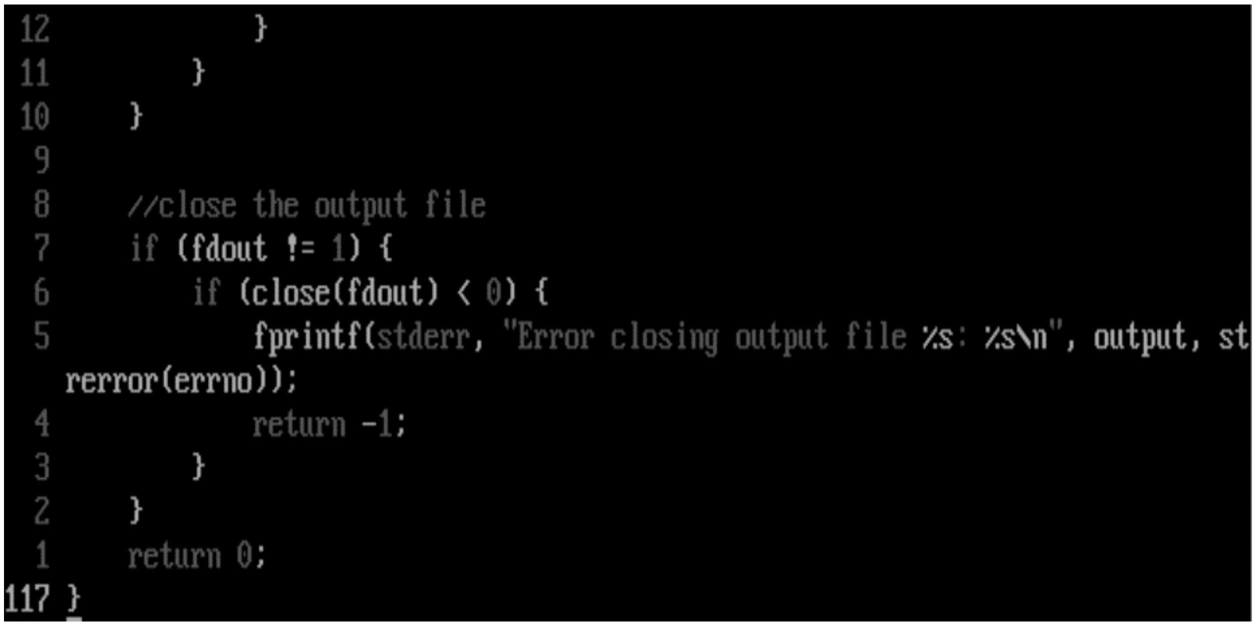
```c
                break;
            default: /* '?' */
                exit(EXIT_FAILURE);
        }
    }

    //instantiation of buffer
    if (bufferSize == 0) {
        bufferSize = 8192;
    }
    char buff[bufferSize];

    //setup of output location
    if (output != "") {
        fdout=open(output, O_WRONLY|O_CREAT|O_TRUNC, 0666);
        if (fdout < 0) {
            fprintf(stderr, "Can't open output file %s for writing: %s\n", output, strerror(errno));
            return -1;
        }
    } else {
        fdout=1;
    }

    //determine number of inputs
    if (optind < argc) {
```



```
        firstInd = optind;
        while (optind < argc) {
            numInputs++;
            optind++;
        }
        optind = firstInd;
    } else {
        noInput = 1;
        numInputs = 1;
    }

    //loop through the inputs
    for (int i = 0; i < numInputs; i++) {

        //setup input location
        if (noInput == 1) {
            inFile = "-";
        } else {
            inFile = argv[optind+i];
        }

        //open file for reading
        if ((strcmp(inFile, "-")) == 0) {
            fdin = 0;
            inFile = "stdin";
        } else if ((fdin = open(inFile, O_RDONLY)) < 0) {
            fprintf(stderr, "Error opening input file %s for reading: %s\n",
 inFile, strerror(errno));
```



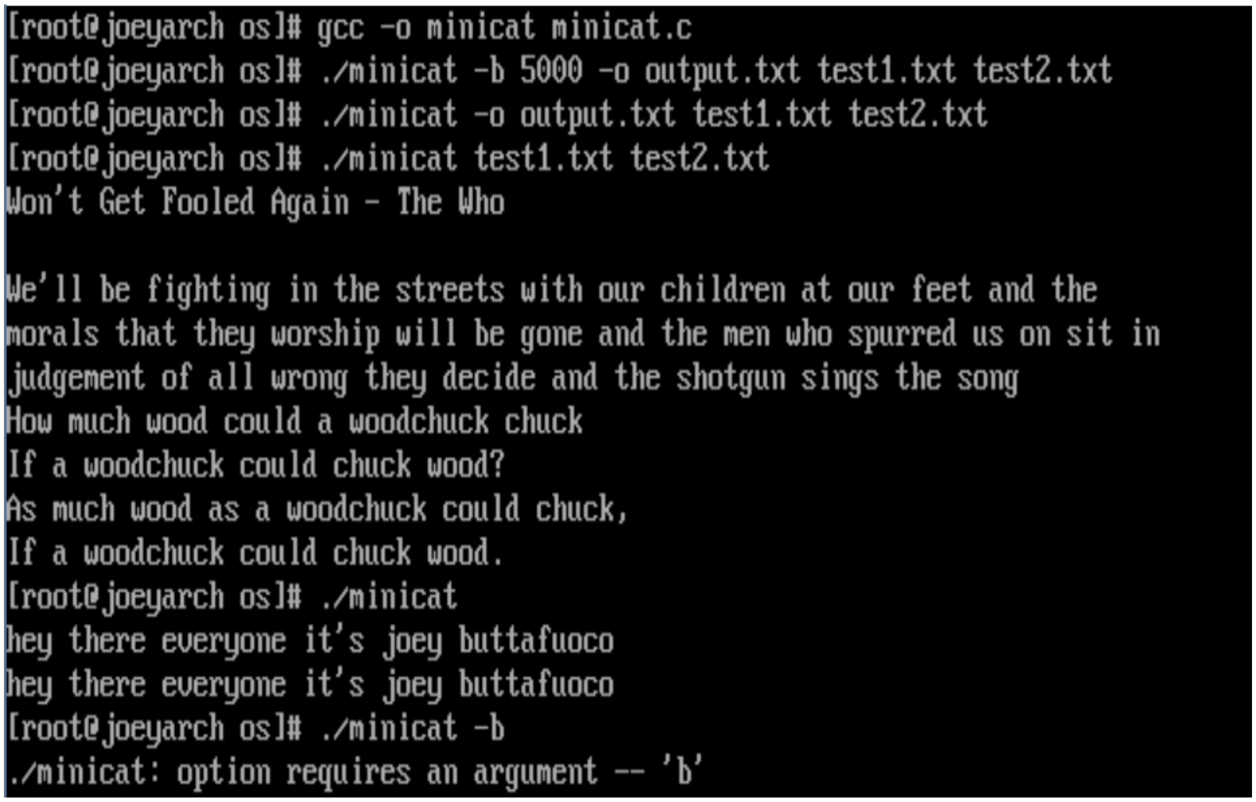
```c
            return -1;
        }

        //read and write file multiple times until the end
        while ((n=read(fdin, buff, sizeof(buff))) != 0) {
            if (n < 0) {
                fprintf(stderr, "Error reading input file %s: %s\n", inFile, strerror(errno));
                return -1;
            } else {
                while (remainder < n) {
                    if ((remainder = write(fdout, buff, n)) < 0) {
                        fprintf(stderr, "Error writing to output file %s: %s\n", output, strerror(errno));
                        return -1;
                    }
                    n -= remainder;
                    remainder = 0;
                }
            }
        }

        //close the input file
        if (fdin != 0) {
            if (close(fdin) < 0) {
                fprintf(stderr, "Error closing input file %s: %s\n", inFile, strerror(errno));
                return -1;
```



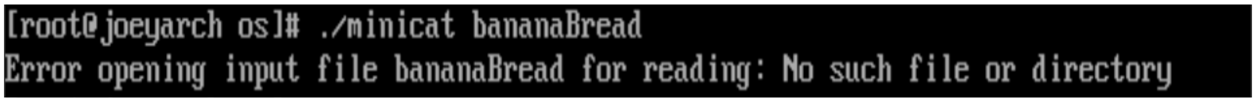
```
                }
            }
        }

        //close the output file
        if (fdout != 1) {
            if (close(fdout) < 0) {
                fprintf(stderr, "Error closing output file %s: %s\n", output, st
rerror(errno));
                return -1;
            }
        }
        return 0;
}
```

Part 2: Sample Run and Error Handling

The program was written in the VIM text editor on the ArchLinux operating system as a guest in a VirtualBox on a Windows 10 host machine. The images below show successful compiling and running of the minicat program and accurate file concatination. Error messages are thrown when the parameters of the program are not met. Other error messages, such as those for read and write, are not shown due to the difficulty in generating them, but are nonetheless still checked for.



```
[root@joeyarch os]# gcc -o minicat minicat.c
[root@joeyarch os]# ./minicat -b 5000 -o output.txt test1.txt test2.txt
[root@joeyarch os]# ./minicat -o output.txt test1.txt test2.txt
[root@joeyarch os]# ./minicat test1.txt test2.txt
Won’t Get Fooled Again - The Who

We’ll be fighting in the streets with our children at our feet and the
morals that they worship will be gone and the men who spurred us on sit in
judgement of all wrong they decide and the shotgun sings the song
How much wood could a woodchuck chuck
If a woodchuck could chuck wood?
As much wood as a woodchuck could chuck,
If a woodchuck could chuck wood.
[root@joeyarch os]# ./minicat
hey there everyone it’s joey buttafuoco
hey there everyone it’s joey buttafuoco
[root@joeyarch os]# ./minicat -b
./minicat: option requires an argument -- ’b’
```



```
[root@joeyarch os]# ./minicat bananaBread
Error opening input file bananaBread for reading: No such file or directory
```

Part 3: Experimental Buffer Testing

In the case of files that are larger than the buffer size, it is necessary for the program to cycle through multiple read and write system calls in order to properly write all the data to the output file. Large buffers are able to accomplish this in single pass but at the cost of more memory. The program was run on an HP OMEN Notebook PC 15 which has an Intel Core i7-4710HQ CPU @ 2.5GHz, NVDIA GeForce GTX 970M, 8GB DDR3 RAM and a 256 GB SSD.

In order to adequately test the buffer speed, a large text file needed to be created. I created the file using: cat -> largeFile.txt. Then I input two lines: “hello” and “world”. Finally I created a loop which concatinated the file with itself many times until I was left with a 12MB file.

Buffer sizes were sampled between 1 byte and 262,144 bytes (2^{18}) in powers of 2 resulting in 18 measurements. Time was measured using the “time” command: “time ./minicat -b [buffer size] -o output.txt largeFile.txt”

The result of the test shows that increasing the buffer gives diminishing marginal returns on performance speed. After approximately 2^{13} bytes, the throughput levels off and shows that the hardware is now the limitation to the system. Thus, any further increase in buffer size will not show better results in the speed of the system.

Figure 1: Trendline showing speed as buffer size increases

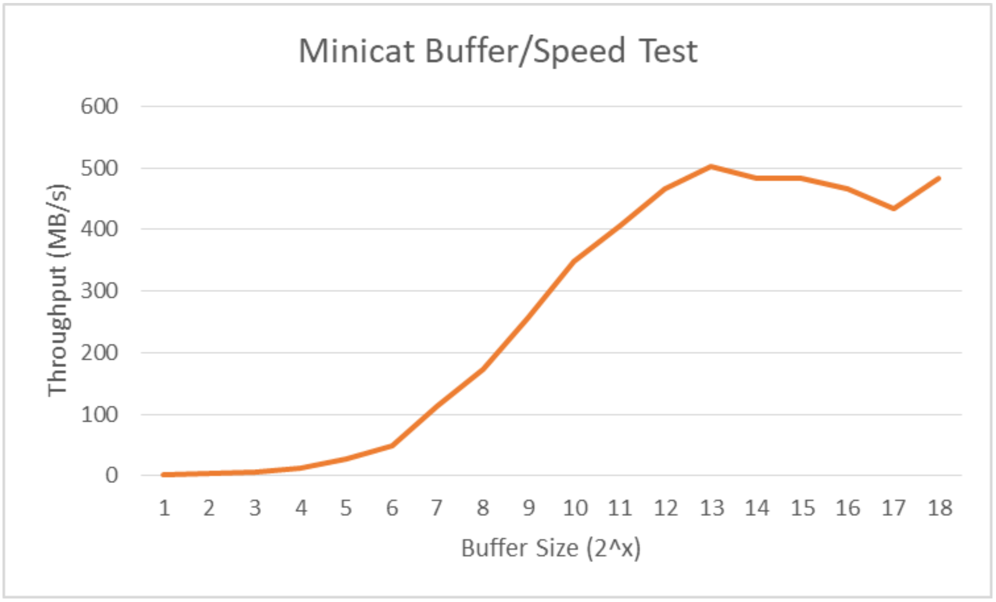


Figure 2: Raw Buffer Data

2^x	Buffer Size (bytes)	Real Time Elapsed (s)	Throughput (MB/s)
1	2	7.375	1.706
2	4	3.691	3.409
3	8	1.997	6.301
4	16	0.994	12.659
5	32	0.461	27.295
6	64	0.259	48.583
7	128	0.111	113.360
8	256	0.073	172.369
9	512	0.049	256.794
10	1024	0.036	349.525
11	2048	0.031	405.900
12	4096	0.027	466.034
13	8192	0.022	571.951
14	16384	0.026	483.958
15	32768	0.026	483.958
16	65536	0.027	466.034
17	131072	0.029	433.894
18	262144	0.026	483.958