Arash Mostafaee
Foad Ghafourian

**Lab 4: Input-Output Operations and Buffers**

The main focus of this lab is to learn how buffering affects I/O operations. The reasoning behind buffering is to reduce the number of system calls, accumulate data in a buffer and transfer a large amount of data each time a system call is made. Since this lab is directly taken from the course textbook, we will use the information from chapter 17, “A Programmer’s View Of Devices, I/O, And Buffering” which circulates around the lab assignment.

**Task 1**

To start off with the first task we implement a read buffer which operates on a text file. Figure 1.1 and figure 1.2 is just for showing that we were successful with trying to open a file descriptor for reading.

```
int main(){
        int fd = open("textfile.txt", O_RDONLY);
        char buff[10];
        read(fd, buff, 10);
        printf("buff: %s\n", buff);

        close(fd);
}
```

Figure 1.1

```
pi@raspberrypi:~ $ gcc -ansi lab4.c -o file
pi@raspberrypi:~ $ ./file
buff: 1234567890
pi@raspberrypi:~ $
```

Figure 1.2

The next step is to create a function “buf_in” which uses a character array of size 16 bytes as a read buffer and storing the read data from our text file. In addition, the function should return the next byte in the buffer on subsequent calls.

When we assume that the underlying device allows transfer of more than one byte of data, then buffering can reduce the number of system calls. This is done by allocating a buffer then making a system call to fill the buffer and furthermore satisfying subsequent requests from the buffer. Figure 17.11 from the course textbook lists the steps required for buffering on input.

**Setup(N)**

1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to indicate that the buffer is empty.

**Input(N)**

1. If the buffer is empty, make a system call to fill the entire buffer, and set pointer p to the start of the buffer.
2. Extract a byte, D, from the position in the buffer given by pointer p, move p to the next byte, and return D to the caller.

**Terminate**

1. If the buffer was dynamically allocated, deallocate it.

**Figure 17.11** The steps required to achieve buffered input.

Figure 1.3

We set the position to zero in order to make a system call to fill the buffer. Then we extract a byte from the position and iterate to the next byte, finally returning the extracted byte. Since we are to test that our function which uses a character array of 16 bytes will work on files of 32 bytes, then we will extract up to 30 times from our read buffer since we are making two system calls to fill the buffer.

```
char *buff;
int pos;

char buf_in(int size, int fd){
        if(pos >= size-1){
                buff = malloc(size);
                read(fd, buff, size);
                pos = 0;
        }
        else{
                pos++;
        }
        return buff[pos];
}
```

Figure 1.4

Arash Mostafaee
Foad Ghafourian

We set the position equal to the size so that the if statement is executed in order to make a system call to fill the buffer. Then we use the same method to open a file descriptor for reading as done in the first step of this task. In addition, we use lseek in order to measure the number of bytes of the file.

```
int main(){
        int size = 16;
        pos = size;

        int fd = open("textfile.txt", O_RDONLY);

        int fsize = lseek(fd, 0, SEEK_END) - lseek(fd, 0, SEEK_SET);
        if(lseek(fd, 0, SEEK_SET) == -1){
                printf("lseek error");
        }
        int i = 1;
        for (i; i<fsize; i++){
                printf("(%d - %c) ", i, buf_in(size, fd));

        }
        free(buff);
}
```

Figure 1.5

```
pi@raspberrypi:~ $ ./file
(1 - a) (2 - b) (3 - c) (4 - d) (5 - e) (6 - f) (7 - g) (8 - h) (9 - i) (10 - j)
 (11 - k) (12 - l) (13 - m) (14 - n) (15 - o) (16 - p) (17 - q) (18 - r) (19 - s
) (20 - t) (21 - u) (22 - v) (23 - w) (24 - x) (25 - y) (26 - z) (27 - A) (28 -
B) (29 - C) (30 - D) (31 - E) (32 - F) pi@raspberrypi:~ $
```

Figure 1.6

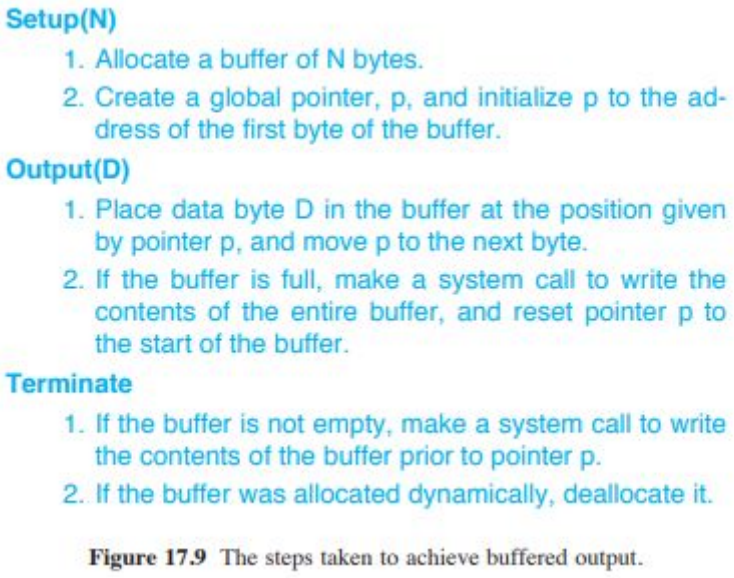
```
 GNU nano 2.2.6            File: textfile.txt

abcdefghijklmnopqrstuvwxyzABCDEF
```

Figure 1.6, the text file

Arash Mostafaee
Foad Ghafourian

**Task 2**

With the second task of this lab being to implement a write buffer instead of a read buffer, we still could use the buffer from the previous task to write the contents from an input text file to an output text file (reading from the first file and write to the second file). Hence, when the buffer has been initialized as in task 1, our output function buf_in is called to transfer data to the second text file which is empty. Once again we took use of the course text book following the steps on how to implement the output function.



**Setup(N)**

1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to the address of the first byte of the buffer.

**Output(D)**

1. Place data byte D in the buffer at the position given by pointer p, and move p to the next byte.
2. If the buffer is full, make a system call to write the contents of the entire buffer, and reset pointer p to the start of the buffer.

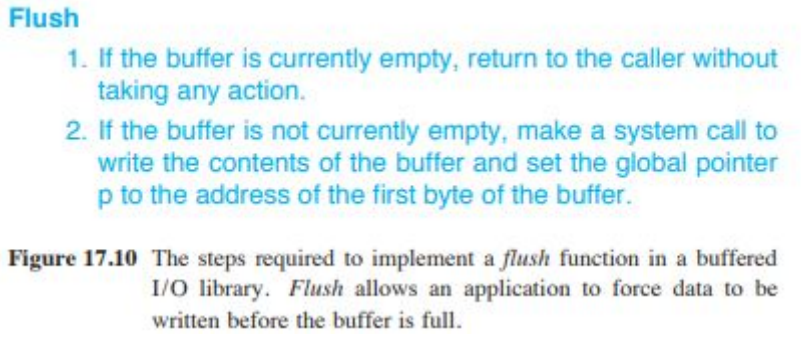
**Terminate**

1. If the buffer is not empty, make a system call to write the contents of the buffer prior to pointer p.
2. If the buffer was allocated dynamically, deallocate it.

**Figure 17.9** The steps taken to achieve buffered output.

Figure 2.1

We were also tasked to implement a flush function to force data to be sent even though the buffer could not be full. The only time the call of the flush function will have no effect is if a buffer is empty.



**Flush**

1. If the buffer is currently empty, return to the caller without taking any action.
2. If the buffer is not currently empty, make a system call to write the contents of the buffer and set the global pointer p to the address of the first byte of the buffer.

**Figure 17.10** The steps required to implement a *flush* function in a buffered I/O library. *Flush* allows an application to force data to be written before the buffer is full.

Figure 2.2

Arash Mostafaee
Foad Ghafourian

```
void buf_out(int size, int fd){
        if(pos >= size){
                write(fd, buff, size);
                pos = 0;
        }
}

void buf_flush(int fd){
        if(pos > 0){
                write(fd, buff, pos);
                pos = 0;
        }
}
```

Figure 2.3

The modification in the main function should be to also open a second file with “O_WRONLY” (open for writing only) and to call the output function in the loop.

However, we were not successful in our implementation as our second text file remained empty. The cause of this is believed to lie with the input function although without certainty. If its the case, we are not entirely sure how to customize buf_in in order to achieve a read and write process of the buffer even though the implementation should be pretty straightforward.

**Task 3**

Due to being unsuccessful in task 2 with implementing a write buffer, we were not able to start with the last task of the lab. However, if we were successful then our approach to the first step of lab 3 would be to have the input function to fill the buffer from the source file and then the output function should write the content to the destination file.