Lab 4: Input-Output Operations and Buffers

Task 1: Buffered Input

To complete the task we had to create a buffered input. We will in detail describe how everything works and describe every little bit of it. The main function where everything we implemented will be used will be described in detail in the last segment.

Open file descriptor

```
if ((file = open(source, O_RDONLY, 0)) < 0)
{
  perror("Open failed");
}
```

When we open the file in the main function, we use the system call function “open”, it takes the source of the file as input (“./text.txt”) and some flags what we want to do with the opened file. The file descriptor is saved into a integer that we call file. If the system call returns -1 the opening of the file failed and we use the function “perror” to output an error.

Implementing the read buffer

```
char buf_in(int file, char buffer[], int BUFFER_SIZE)
{
  int bufferChars;

  if (position == 0)
  {
    if ((bufferChars = read(file, buffer, BUFFER_SIZE)) > 0)
    {
    }
    return buffer[position];
  }
  else
  {
    return buffer[position];
  }
}
```

Our read buffer function buf_in takes in three arguments: a file, a buffer, and buffer size. This is used in the read system call that takes a file, reads buffer_size amount of bits, and stores the bits in the buffer. The task is to first read 16 bits into the buffer, return the first bit and then on subsequent calls iterate through the buffer returning bits one at a time. This is executed using a global variable position, which keeps track of what bit to read on the next function call. So at start the position is 0, the function then read in 16 bits and return the first bit at position 0 in the buffer, the position is later updated in the main function with + 1, one the next buf_in function call the position is 1 and the function then skips reading and returns the bit at position 1 in the buffer. This repeats on subsequent calls until position is 16 in the main function, the position is then reset to 0 and the process repeats from start.

Verification

To verify that buf_in returned the right bits we used printf in the main function after the function had been called, and kept track of our position variable. We also tested different amount of bits.

Task 2: Buffered Output

Open file descriptor

```
if ((file = open(dest, O_WRONLY)) < 0)
{
  perror("Open failed");
}
```

As previously mentioned in task 1 we open the file in the same way as before. However, this time instead of source we use the destination file and what our intentions with the file are. the open function will then return a file descriptor integer and store it in file. the integer will then tell us if the destination file was successfully opened or not.

Implement the write buffer

```
void buf_out(int file2, char *bufferOut, int BUFFER_SIZE, char readByte)
{
  int bufferChars;
  bufferOut[position] = readByte;
  if (position == BUFFER_SIZE - 1)
  {
    if ((bufferChars = write(file2, bufferOut, BUFFER_SIZE)) > 0)
    {
    }
  }
}
```

Our write buffer function buf_out takes in four arguments: a file, a buffer, a buffer size, and the byte read from the buf_in function. Like mentioned in the section above we use our global variable position in this function as well. We first start of by placing the read bit in the buffer at position “position”, the function then checks how many bytes have been placed in the buffer by comparing the position variable with the size of the buffer - 1 (since an array start 0 then the array tail is at buffersize - 1, in this case 15). If the position variable is not the same as buffersize - 1 the rest of the function does not execute, however if position is the same as buffersize - 1, which means the buffer is full, the function writes the buffer onto the file. Position is then later reset in the main function and the process repeats.

Implement the buffer flush

```
void buf_flush(int file2, char buffer[])
{
  int bufferChars;
  if ((bufferChars = write(file2, buffer, position - 1)) > 0)
  {
  }
  else
  {
    perror("No flush needed:");
  }
}
```

Considering that buf_out only writes when the buffer is full, we need a function that writes the remaining bits in the buffer to the file, when there’s no bits left for buf_in and buf_out to process. The buffer flush this this exact job, at the end of the for loop in main, we call buf_flush. Buf_flush takes in 2 arguments: a file and a

buffer. Buf_flush does a write system call which takes a file to write to, a buffer, and a buffersize. In this case we use the position saved from the latest buf_out to know how many bits are left in the buffer, and set the buffersize to position - 1 since we add 1 to position at the end of the for loop in main before the flush function. Buf_flush then writes the remaining bits to the file.

Verification

To verify that everything worked as intended and the bits from file 1 transfered to file 2, we used the command line tool called “diff”, more information about how that worked can be found in the segment called Task 3: Performance evaluation.

Task 3: Performance evaluation

File names from command line

We solved this problem by taking in two arguments in the main function, an integer called argc and an char array called argv, the argv array gets filled with the command line arguments in the 0th position the filename is placed. In the first and second place the first respectively second argument is placed. We verified that our code worked by typing this in the command line “./main file.txt file2.txt” and this print:

```
printf("Source: %s Destination: %s\n", argv[1], argv[2]);
```

The output was this:

```
Andreass-MBP:lab_4 andreas$ ./main file.txt file2.txt
Source: file.txt Destination: file2.txt
```

So grabbing the file names from the command line worked as intended then we opened “file.txt” and “file2.txt” with these lines of code:

```
if ((file = open(argv[1], O_RDONLY, 0)) < 0)
{
  perror("Open failed");
}
if ((file2 = open(argv[2], O_WRONLY, 0)) < 0)
{
  perror("Open failed");
}
```

Check for differences

We checked for differences using the terminal command “diff” we wrote “diff file.txt file2.txt” and we got no error message so we know that the files are the same. If we on purpose

switched a file to not work we got this error message:

```
Andreass-MBP:lab_4 andreas$ diff file.txt file2.txt
1c1
< this is an epic test of epic proportions, there is difference
\ No newline at end of file
---
> this is an epic test of epic proportions cuong
\ No newline at end of file
```

So this basically tells us that our files are the same after being copied using buf_in and buf_out

Main

In the main file is where we then implement the buf_in and buf_out function. And it looks like this:

```c
int main(int argc, char *argv[])
{
  printf("Source: %s Destination: %s\n", argv[1], argv[2]);

  int file, file2, BUFFER_SIZE = 16;
  char buffer[BUFFER_SIZE];
  char bufferOut[BUFFER_SIZE];
  char readByte;

  if ((file = open(argv[1], O_RDONLY, 0)) < 0)
  {
    perror("Open failed");
  }
  if ((file2 = open(argv[2], O_WRONLY, 0)) < 0)
  {
    perror("Open failed");
  }

  //Checking the size of the file
  struct stat st;
  stat(argv[1], &st);
  int filesize = st.st_size;

  for (int i = 1; i <= filesize + 1; i++)
  {
    readByte = buf_in(file, buffer, BUFFER_SIZE);
    buf_out(file2, bufferOut, BUFFER_SIZE, readByte);
    position = position + 1;
    if (position == BUFFER_SIZE)
    {
      position = 0;
    }
  }
  buf_flush(file2, bufferOut);
  close(file);
  close(file2);

  return 0;
}
```

We use an integer array to store our offsets to keep track of what offsets we have used. We use the system call “stat” to get the size of the file in bytes, then we divide it with the buffersize. By doing that we can calculate how many times we have to run the loop to get everything from the first file written to the second one. In the loop we first run buf_in to fill the buffer. And we put the returned offset into the array. In the buf_in we use the previous offset as offset. Then we run buf_out and the offset is the same as in buf_in.