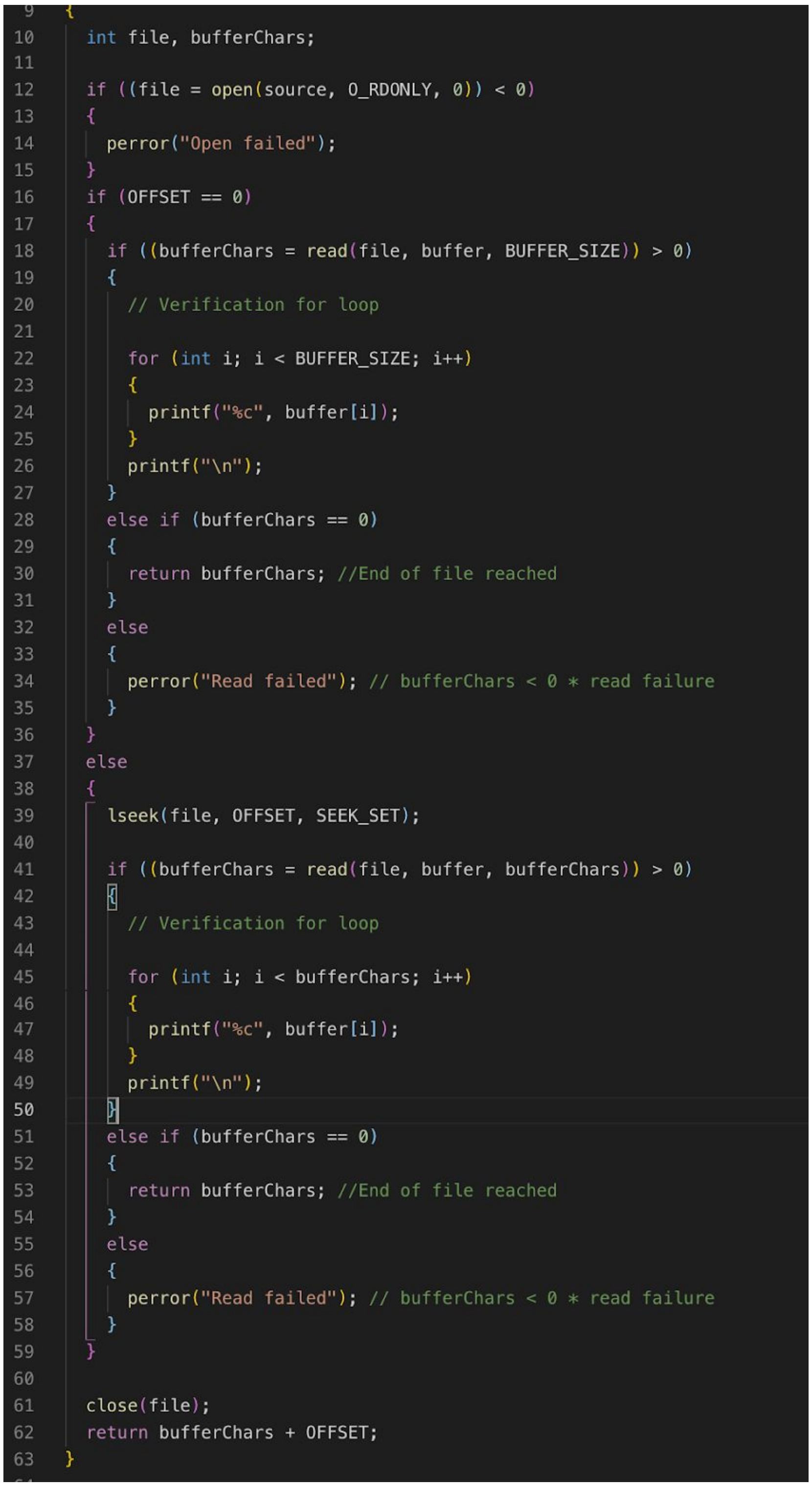
Lab 4: Input-Output Operations and Buffers

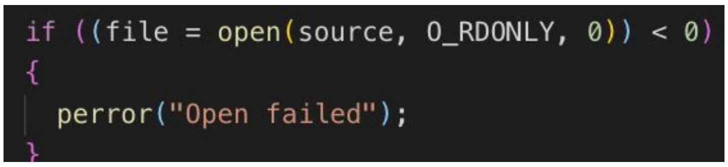
Task 1: Buffered Input



```c
{
  int file, bufferChars;

  if ((file = open(source, O_RDONLY, 0)) < 0)
  {
    perror("Open failed");
  }
  if (OFFSET == 0)
  {
    if ((bufferChars = read(file, buffer, BUFFER_SIZE)) > 0)
    {
      // Verification for loop

      for (int i; i < BUFFER_SIZE; i++)
      {
        printf("%c", buffer[i]);
      }
      printf("\n");
    }
    else if (bufferChars == 0)
    {
      return bufferChars; //End of file reached
    }
    else
    {
      perror("Read failed"); // bufferChars < 0 * read failure
    }
  }
  else
  {
    lseek(file, OFFSET, SEEK_SET);

    if ((bufferChars = read(file, buffer, bufferChars)) > 0)
    {
      // Verification for loop

      for (int i; i < bufferChars; i++)
      {
        printf("%c", buffer[i]);
      }
      printf("\n");
    }
    else if (bufferChars == 0)
    {
      return bufferChars; //End of file reached
    }
    else
    {
      perror("Read failed"); // bufferChars < 0 * read failure
    }
  }

  close(file);
  return bufferChars + OFFSET;
}
```

To complete the task we had to create a buffered input. To the left here you can see the whole function “buf_in”, 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.

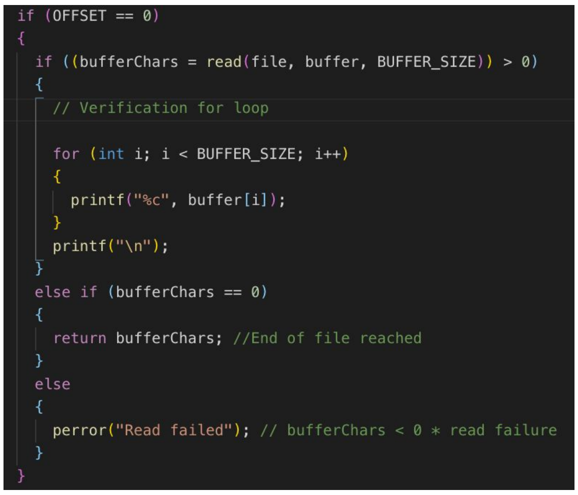
Open file descriptor



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

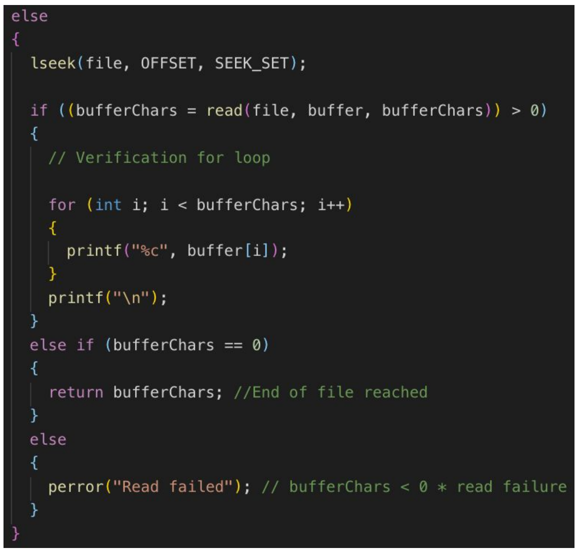
This is where we open the file, we use the system call function “open”, it takes the source of the file as input (“./text.txt”) and some flags what we want to do with the opened file. The file descriptor is saved into a integer that we call file. If the system call returns -1 the opening of the file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

Implementing the read buffer



```
if (OFFSET == 0)
{
  if ((bufferChars = read(file, buffer, BUFFER_SIZE)) > 0)
  {
    // Verification for loop

    for (int i; i < BUFFER_SIZE; i++)
    {
      printf("%c", buffer[i]);
    }
    printf("\n");
  }
  else if (bufferChars == 0)
  {
    return bufferChars; //End of file reached
  }
  else
  {
    perror("Read failed"); // bufferChars < 0 * read failure
  }
}
```

Image 3



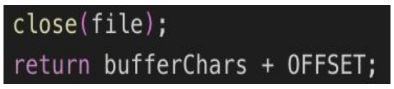
```
else
{
  lseek(file, OFFSET, SEEK_SET);

  if ((bufferChars = read(file, buffer, bufferChars)) > 0)
  {
    // Verification for loop

    for (int i; i < bufferChars; i++)
    {
      printf("%c", buffer[i]);
    }
    printf("\n");
  }
  else if (bufferChars == 0)
  {
    return bufferChars; //End of file reached
  }
  else
  {
    perror("Read failed"); // bufferChars < 0 * read failure
  }
}
```

Image 4

With this if and else statement we implement the read buffer. The if statement checks if the OFFSET is 0, if the OFFSET is 0 that means that this is the first time we are reading text into the buffer and we don’t have to change where the “read” system calls writes on to the buffer. “read” returns an integer with how many bytes the system call read. If the integer is less than 0 it means that something went wrong and we use “perror” again. If the integer was 0 then it means that 0 bytes was written and we have reached the end of the file.

If the offset is something else than 0 we go into to the else statement. (image 4) First in the else statement we use the system call “lseek”, what that does is that it repositions the read file offset so when we then in the if statement call “read” it puts the read bytes from the right position in the file. This means that we don’t read the same 16 bytes from the file, but we “progress” through the file. Read more about the for loops under the verification segment of task 1.

Lastly in the buf_in function we do this:



```
close(file);
return bufferChars + OFFSET;
```

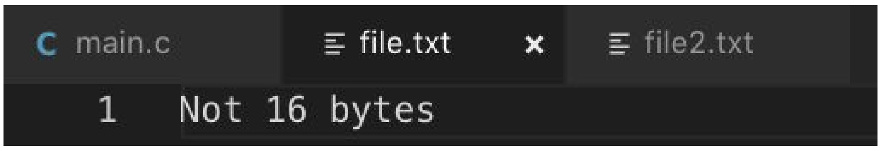
We use the system call “close” to close the file and then we return the amount of characters buffered plus the offset. This way we get the right offset next time we run the buf_in to read everything from the file not just the first 16 bytes.

Empty buffer and subsequent calls

As described in the last section we use an offset to be able to run subsequent calls.

Verification

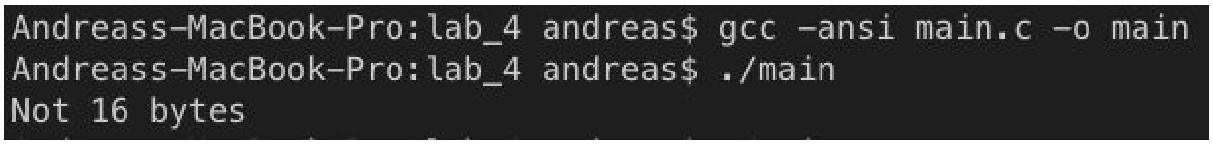
To verify that our function does what it is supposed to do we use the for loops in image 3 and 4.
This is how 'file.txt' looks it has 12 bytes:



```
C main.c        file.txt  ×        file2.txt
1   Not 16 bytes
```

Image 6

and the for loops output with the -ansi compile flag:

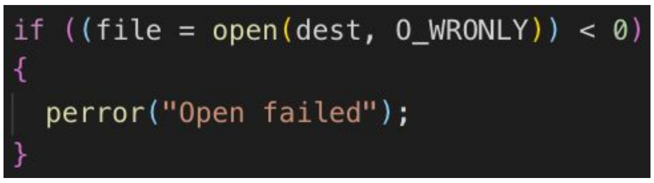


```
Andreass-MacBook-Pro:lab_4 andreas$ gcc -ansi main.c -o main
Andreass-MacBook-Pro:lab_4 andreas$ ./main
Not 16 bytes
```

Image 7

Task 2: Buffered Output

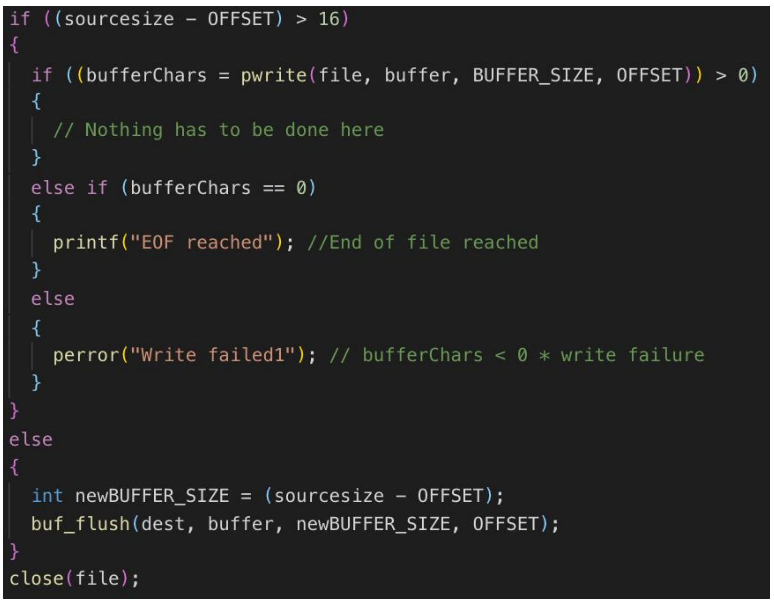
Open file descriptor



```
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



```
if ((sourcesize - OFFSET) > 16)
{
  if ((bufferChars = pwrite(file, buffer, BUFFER_SIZE, OFFSET)) > 0)
  {
    // Nothing has to be done here
  }
  else if (bufferChars == 0)
  {
    printf("EOF reached"); //End of file reached
  }
  else
  {
    perror("Write failed1"); // bufferChars < 0 * write failure
  }
}
else
{
  int newBUFFER_SIZE = (sourcesize - OFFSET);
  buf_flush(dest, buffer, newBUFFER_SIZE, OFFSET);
}
close(file);
```

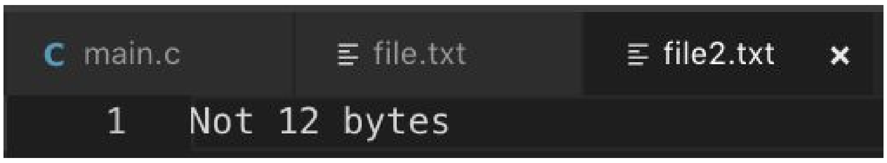
For the write buffer there's two scenarios. By subtracting the amount of bytes that have been read from the bytesize of the file, the function knows how many bytes there's left to read from the source file. The function then either uses the pwrite system call directly if there's more than 16 bytes left or calls the buffer flush function which are mentioned in the next section. The way pwrite works are straightforward, it takes in 4 arguments which are: the destination file, the buffer from read buffer, the buffer size and lastly an offset to where in the destination file it will start writing. Pwrite will then try writing what's from the buffer to the offset in the destinationfile, it will then return a file descriptor integer to bufferChars which displays the outcome of the system call. As mentioned for read buffer the write buffer functions starts by opening the destination file and ends by closing the destination file.

Implement the buffer flush

The buffer flush forces the contents in the buffer to be written in the destination file disregarding if the buffer is full or not. The buffer flush takes in four arguments: destination file, the buffer from read buffer, the buffer size of how many bytes are left and lastly the offset to know where to write in the destination file.

Verification

As seen in image 6 the file that is being written from says "Not 12 bytes", the file that we write too is called 'file2.txt', the content of that file after the "./main" command has been written in the terminal is:

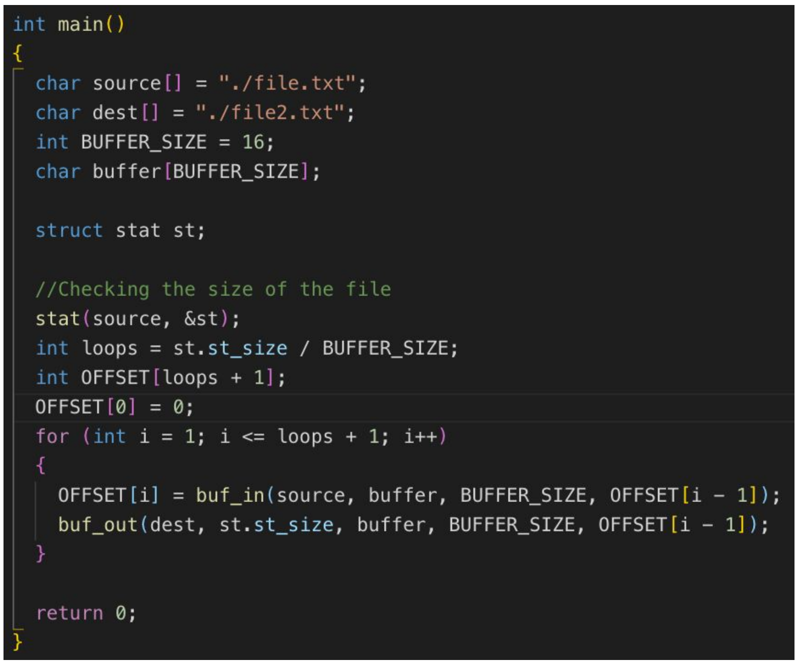


```
Not 12 bytes
```

So we can clearly see that the buffer is working as intended.

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()
{
  char source[] = "./file.txt";
  char dest[] = "./file2.txt";
  int BUFFER_SIZE = 16;
  char buffer[BUFFER_SIZE];

  struct stat st;

  //Checking the size of the file
  stat(source, &st);
  int loops = st.st_size / BUFFER_SIZE;
  int OFFSET[loops + 1];
  OFFSET[0] = 0;
  for (int i = 1; i <= loops + 1; i++)
  {
    OFFSET[i] = buf_in(source, buffer, BUFFER_SIZE, OFFSET[i - 1]);
    buf_out(dest, st.st_size, buffer, BUFFER_SIZE, OFFSET[i - 1]);
  }

  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.