Erik Hammar & Markus Beronius

Input-output operations and buffers

The task at hand was to create create a character buffer with size b = 16 bytes, and functions to read and write b bytes at a time between two different files. The buffer was implemented to mimic a bus between two memory locations. Part 1 was to code the function which reads b bytes from a file and stores them in the buffer and we called the function *buf_in*. Part 2 was to implement two functions. One which writes the b bytes in the buffer into a separate file, called *buf_out*. The other function was to flush the remaining bytes in the buffer, for example if the buffer was not filled fully, into the file, called *buf_flush*. Following are code snippets and relevant descriptions of the integral parts of the functions.

buf_in

buf_in stores bytes into a buffer if the buffer is empty, and then returns each byte, one at the time, until all bytes have been returned. *pos* determines what byte to return and is also the flag that tells the program when all bytes in the buffer has been returned.

@input *file*: Source file to read
buffer[]: char array to store bytes in after being read
bufferSize: int determining how many bytes *read* should read

@return A single byte from *buffer*.

Line 11: Compares *pos* to -1. If true it means that the file is being read for the first time and that *buffer* is empty.

Line 12: *read* reads from *file* and stores the bytes it reads into *buffer*. *bufferSize* determines how many bytes for *read* to read.

Line 19: *pos* == 15 means that each byte in *buffer* has been returned and the program is ready to read in the next 16 bytes from the file.

Line 29: Increment *pos*. This is why *pos* initial value is -1, so that the program can increment *pos* before returning the byte.

```
1  #include <fcntl.h>
2  #include <sys/stat.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int pos = -1;
7
8  char buf_in(int file, char buffer[], int bufferSize) {
9      int bufferChars = 0;
10
11     if (pos == -1) {
12         if ((bufferChars = read(file, buffer, bufferSize)) > 0) {
13         }
14         else {
15             perror("Read failed");
16             return;
17         }
18     }
19     else if (pos == 15) {
20         pos = -1;
21         if ((bufferChars = read(file, buffer, bufferSize)) > 0) {
22         }
23         else {
24             perror("Read failed");
25             return;
26         }
27     }
28
29     pos++;
30     printf("%c", buffer[pos]);
31     return buffer[pos];
32 }
```

buf_flush

buf_flush writes the last bytes to the file, and is only called at the end of the program to write the last bytes that may not fill the buffer completely.
@input *file:* Destination file to write the buffer in
buffer: char array where byte to be written is stored

buf_out

buf_out writes the contents of *buffer* to a file
@input *dest*: Source file to written to
**buffer*: char array where bytes to be written are stored
bufferSize: int determining how many bytes *write* should write.

Line 39: Should only write if the buffer is completely filled (with new bytes compared to the last time it wrote), i.e. if pos ==15.

```
34  void buf_flush(int file, char *buffer) {
35      write(file, buffer, pos);
36  }
37
38  void buf_out(int dest, char *buffer, int bufferSize) {
39      if (pos == 15) {
40          write(dest, buffer, bufferSize);
41      }
42  }
```

main

In main we declare all the variables we need in the functions. *source* is the file that *buf_in* reads from, and similarly *dest* is the file that *buf_out* and *buf_flush* writes to. We create a pointer so that we can use the buffer in all functions. *struct stat* is used to get the number of bytes from the file to read, so that the loop runs an adequate number of times.

```
44  int main() {
45      char source[] = "./file.txt";
46      char dest[] = "./file2.txt";
47      int bufferSize = 16;
48      char *buffer;
49      buffer = malloc(bufferSize*sizeof(char));
50      char returnValues[4];
51      int file1, file2;
52
53      struct stat st;
54
55      stat(source, &st);
56      int loop = st.st_size;
57
58      if ((file1 = open(source, O_RDONLY, 0)) < 0) {
59          perror("Open failed");
60          return;
61      }
62      if ((file2 = open(dest, O_WRONLY, 0)) < 0) {
63          perror("Open failed");
64          return;
65      }
66
67      int i;
68      for (i = 0; i <= loop; i++) {
69          returnValues[i] = buf_in(file1, buffer, bufferSize);
70          buf_out(file2, buffer, bufferSize);
71      }
72      buf_flush(file2, buffer);
73      close(file1);
74      close(file2);
75      return 0;
76  }
```

```
C:\Users\Erik Hammar\Documents\Datorteknik>FC file.txt file2.txt
Comparing files file.txt and FILE2.TXT
FC: no differences encountered
```

Task 3

For this task we used the functions *buf_in* and *buf_out* from the previous tasks. Since the shell command *diff* is a linux function, we found the shell command *FC* for windows. We used *FC* on file.txt and file2.txt where file.txt is the file from which the buffer reads in the function *buf_in* and file2.txt is the file to which the buffer writes in the function *buf_out*. When we compare the files after we have run *buf_in* and *buf_out* there are no differences between the files. This ensures us that the files are identical.
In the code, we included the library *ctime* to be able to time the read and write functions. Here we create the variables *start* and *end* and place them around the code that we want to time. When running the functions and reading/writing only 1 byte the CPU is too fast to show

the amount of time elapsed, so we had to do it to while reading the entire text file. Here we can see that it still takes a very small amount of time. The file size was 3.7kB.

Line 69: The *clock_t* variable start initiates the timer.

Line 75: The *clock_t* variable end ends the timer.

Line 76, 77: Displays the values of start and end as well as calculates the time passed.

```
69      start = clock();
70      int i;
71      for (i = 0; i <= loop; i++) {
72          buf_in(file1, buffer, bufferSize);
73          buf_out(file2, buffer, bufferSize);
74      }
75      end = clock();
76      printf("\n%d %d\n", start, end);
77      printf("Time: %f", (double)((end - start))/(CLOCKS_PER_SEC*loop));
```

```
46 77
Time: 0.000008
```