Lab4

Vide Hopper & Martin Karlsson

Task 1: Buffered input

In this task we were supposed to create a function called *buf_in* which uses a character array of size 16 bytes to implement a read buffer. It should read 16 bytes from a file if the array is empty and return each byte on subsequent calls.

We created a struct which contained the position which increments on each call of *buf_in*. With this attribute we were able to know when we needed to fill the buffer again. The next attribute, *left*, was created so that when we reach the end of the file the program knows if the last read can place the remaining bytes in the buffer. The last attribute is the buffer which is where the read data is stored. The buffer needs to be allocated in memory which the function *alloc* does.

```
#define BSize 1

char* alloc(int length, int size){
        char* array = (char*)calloc(length,size);
        return array;
}

typedef struct {
        int pos;
        int left;
        char* buf;
}BufArr;
```

Code 1, BuffArr

The next step was to create the actual read function *buf_in* that take the character array *buffer* and the file to read from as arguments. The function then proceeds to check if buffer is empty by taking the remainder of the buffers variable pos and the global variable *BSize* which is the same as the buffer size. If the remainder is zero then the function should read from the file. If the statement is true the function clears the buffer and then uses the *read* function. The *read* function takes the file, buffer and *BSize* (the size of how many bytes it should read) as arguments. Our function also checks if the number of bytes read is the same as the *BSize*, if this is not true that means that the file has ended and that we could not fill another buffer and the function returns an empty character. Otherwise the function returns the first byte of the *buffer*. Every other time the function is called and the first if statements is false then it only returns the next byte of the buffer.

```
char buf_in(BufArr *buffer, int file){
        if(buffer->pos % BSize==0){
                memset(buffer->buf, '\0',BSize);
                buffer->left = read(file, buffer->buf, BSize);
                if(buffer->left != BSize)
                        return '\0';
        }
        buffer->pos++;
        return(buffer->buf[(buffer->pos-1) % BSize]);
}
```

Code 2, buf_in

The function is tested with various of buffer sizes to make sure every case worked as expected. In later tasks the function is used for sizes varying from 1 to 64 bytes.

Task 2: Buffered output

The next exercise was to create a *buf_out* and a *buf_flush* function. These functions prints the value of a buffer to a given file. The difference of these are that *buf_out* can be called many times without returning anything. It only writes to the file if the buffer is full. The flush function on the other hand will write what is on the buffer to the file. The difference is the last argument of the write function which determines how many elements are going to be written to the file. In the *buf_out, BSize* is used while on *buf_flush* the structs attribute *left* is used. If last argument of write is larger than the buffer it will try to write '\0' which is known as the end of file symbol. This caused some errors so we made sure not to print it.

```
int buf_out(BufArr *buffer, int file){
        if((buffer->pos-1) % BSize == 0){
                write(file,buffer->buf,BSize);
        }
}

int buf_flush(BufArr *buffer, int file){
        write(file,buffer->buf,buffer->left);
        buffer->pos = 0;
}
```

Code 3, buf_out & buf_flush

This function was tested with different lengths in the buffer. We tested if the buffer was less than *BSize* with *buf_flush* and if it was full with *buf_out*. In the next task the function is used with varying *BSize*.

Task 3: Performance evaluation

This task was about testing the program and compare its performance to the shell command *cp*. To run our program you need to call it with the the names of the read and write files.

The shell command *diff* compares the two text documents and the result we got was that they were identical after the program had been run.

```
pi@raspberrypi:~ $ ./lab4 test.txt write.txt
pi@raspberrypi:~ $ diff test.txt write.txt
pi@raspberrypi:~ $
```

Code 4, diff

The task also specifies that we want to time the read and the write functions that we used in *buf_in* and *buf_out* to get a understanding of how much time the rest of our functions slows down the process.

```
pi@raspberrypi:~ $ ./lab4 test.txt write.txt
0.000296
0.000294
```

Code 5, Timing one byte read and write

After running the program a few times you could clearly see that the times differed for every time we ran it. The next step was to test our functions and calculate an average time for read using *buf_in* and writing using *buf_out* while using the buffer size of 1 byte and do the same for the *read* and *write* functions. We did this by taking the time it takes for a for loop that call our function repeatedly and then divide it by the number of time the for loop ran. We took time with this technique for *buf_in, buf_out, read* and *write*.

```
int i;
int times = 100000;
clock_t start = clock();
int file;
for (i=0;i<times; i++){
        file = open(argv[1], O_RDONLY);
        buf_in(&buffer,file);
        close(file);
}
clock_t end = clock();

clock_t start2 = clock();
int file_2;
for(i=0;i<times;i++){
        file_2 = open(argv[2], O_WRONLY);
        buf_out(&buffer,file_2);
        close(file_2);
}
clock_t end2 = clock();

printf("%f \n", (((double)end - start) / CLOCKS_PER_SEC)/times);
printf("%f \n", (((double)end2 - start2) / CLOCKS_PER_SEC)/times);

start = clock();
char *b;
for (i=0;i<times; i++){
        file = open(argv[1], O_RDONLY);
        read(file,b,BSize);
        close(file);
}
end = clock();

start2 = clock();
file_2;
for(i=0;i<times;i++){
        file_2 = open(argv[2], O_WRONLY);
        write(file_2,b,BSize);
        close(file_2);
}
end2 = clock();

printf("%f \n", (((double)end - start) / CLOCKS_PER_SEC)/times);
printf("%f \n", (((double)end2 - start2) / CLOCKS_PER_SEC)/times);
```

Code 6, Calculating the average time

The loop will run 100000 times which hopefully gives us a reasonable average time for the different functions. The result can be seen in *Code 7*, the first two prints are *buf_in* and *buf_out* and the two following them are *read* and *write*.

```
pi@raspberrypi:~ $ gcc -ansi lab4.c -o lab4
pi@raspberrypi:~ $ ./lab4 test.txt write.txt
0.000202
0.000261
0.000254
0.000226
```

Code 7, Result from average time

We then proceed to time our function while varying our buffer size and file size. The file *text.txt* contains one character, *test2.txt* contains 4KB and *test3.txt* contains 100KB. As can been seen on *Code 8-10* as the buffer sizes increases the times decreases. It's worth notifying that as the size goes from 16 to 32 and 32 to 64, which is multiplying it by 2 every time, the time cuts in half accordingly.

```
pi@raspberrypi:~ $ gcc -ansi lab4.c -o lab4
pi@raspberrypi:~ $ ./lab4 test.txt write.txt
0.000669 READ
0.001347 WRITE
pi@raspberrypi:~ $ ./lab4 test2.txt write2.txt
0.054275 READ
0.054453 WRITE
pi@raspberrypi:~ $ ./lab4 test3.txt write3.txt
1.273716 READ
1.274038 WRITE
pi@raspberrypi:~ $ _
```

Code 8, read and write with 16 byte buffer

```
pi@raspberrypi:~ $ gcc -ansi lab4.c -o lab4
pi@raspberrypi:~ $ ./lab4 test.txt write.txt
0.001090 READ
0.001945 WRITE
pi@raspberrypi:~ $ ./lab4 test2.txt write2.txt
0.026046 READ
0.026196 WRITE
pi@raspberrypi:~ $ ./lab4 test3.txt write3.txt
0.658701 READ
0.659074 WRITE
pi@raspberrypi:~ $ _
```

Code 9, read and write with 32 byte buffer

```
pi@raspberrypi:~ $ gcc -ansi lab4.c -o lab4
pi@raspberrypi:~ $ ./lab4 test.txt write.txt
0.000829 READ
0.001512 WRITE
pi@raspberrypi:~ $ ./lab4 test2.txt write2.txt
0.017037 READ
0.017428 WRITE
pi@raspberrypi:~ $ ./lab4 test3.txt write3.txt
0.333570 READ
0.333826 WRITE
pi@raspberrypi:~ $
```

Code 10, read and write with 64 byte buffer

The last task is to compare the time from our written copy function with the shell command cp while copying a large file. We use the same file test3.txt as before and the largest buffer size since it was the fastest.

```
pi@raspberrypi:~ $ time cp test3.txt write3.txt

real    0m0.152s
user    0m0.050s
sys     0m0.080s
pi@raspberrypi:~ $ time ./lab4 test3.txt write3.txt

real    0m0.351s
user    0m0.040s
sys     0m0.310s
pi@raspberrypi:~ $
```

Code 11, time with our program (64 byte) against cp

We see in *Code 11* that the cp is slightly faster so we tried to change the buffer size to 256.

```
pi@raspberrypi:~ $ time ./lab4 test3.txt write3.txt

real    0m0.166s
user    0m0.030s
sys     0m0.130s
pi@raspberrypi:~ $ > write3.txt
pi@raspberrypi:~ $ time ./lab4 test3.txt write3.txt

real    0m0.159s
user    0m0.060s
sys     0m0.090s
pi@raspberrypi:~ $
```

Code 12, time with our program (256 byte) against cp

Conclusion

In conclusion we have created a 3 main functions, *buf_in* which fills the buffer, *buf_out* with takes a full buffer and writes its contents to a file and *buf_flush* that write the reminding content of a non-full buffer. We then tested and timed our functions and compared it to the shell command *Cp*.

We got a weird result when we used the C time function and that could be because we use our time function within a for loop. The for loop has one if statement and one incrementation every step and those operations could result in taking longer time than expected. Although, since the result are quite similar, our function and the *read* and *write* function should take close to the same time.

In the last exercise we got a result that is close to the cp function, especially when we raised the buffer size.