Viktor Rask
Amanda Håkansson
DT509G

Lab 4

Computer Architectures for MSc in Engineering

Viktor Rask
Amanda Håkansson
DT509G

Innehåll

Introduction

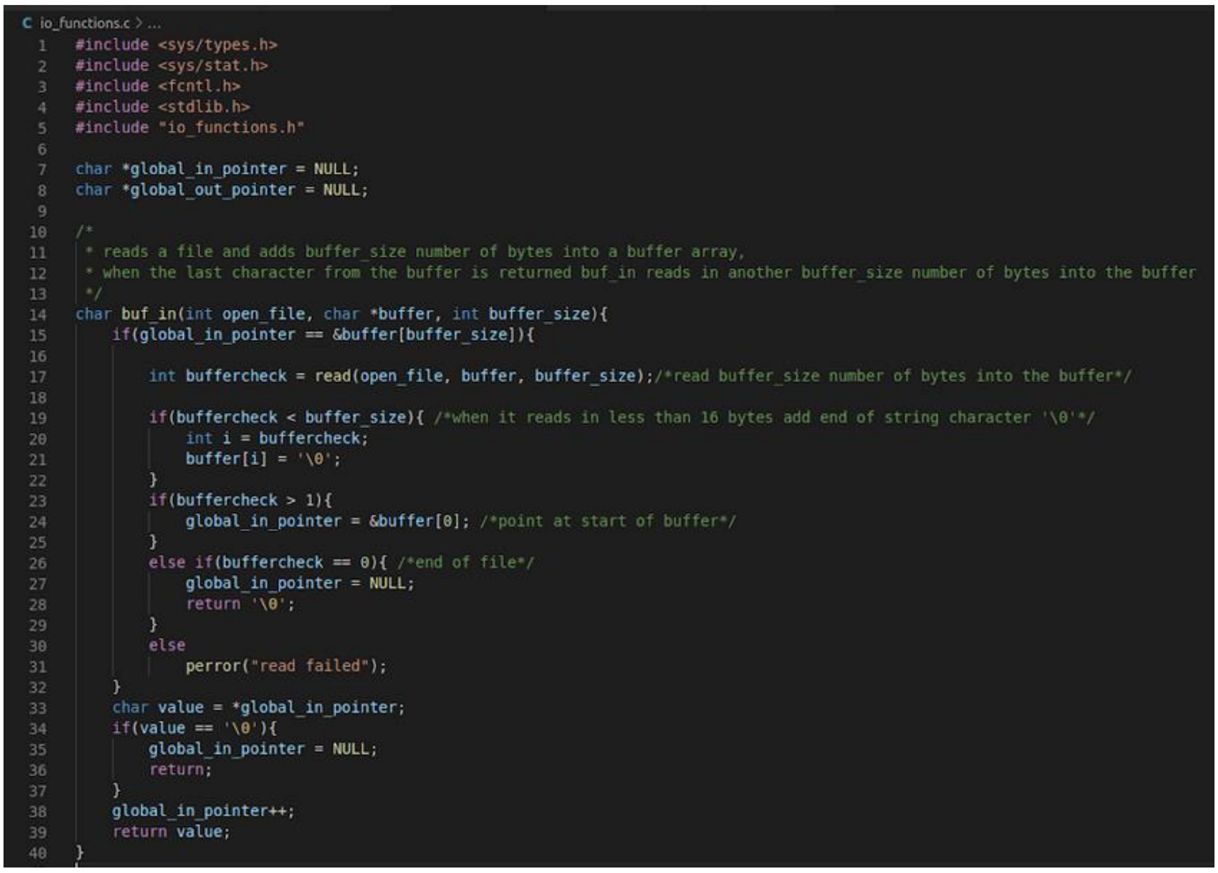
The purpose of this lab was to learn how buffering affects I/O operations. This was done by implementing a function in c for buffered input and one for buffered output. This lab also includes tests to check the time it took for the functions to execute using different buffer sizes, different file sizes and different methods.

Task 1

The purpose of task 1 was to create a function that opened a file to read with system call C API. The function was supposed to add 16 bytes from the file into a buffer and then return the first character in the buffer. On subsequent calls the function was to return the next array in the buffer, but when the buffer was at the last character meaning it would have gone through the entire buffer the function should once again read in 16 or how many bytes left inside the file into the buffer.

Code

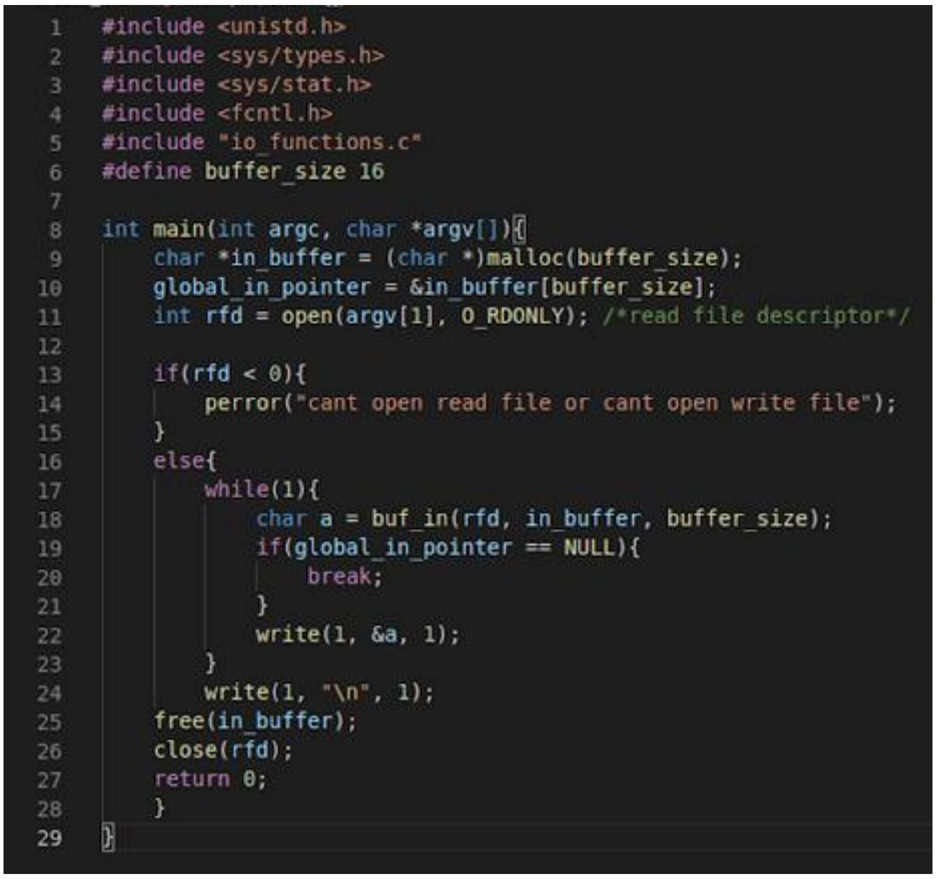
Function buf_in:



```c
C io_functions.c > ...
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include "io_functions.h"

char *global_in_pointer = NULL;
char *global_out_pointer = NULL;

/*
 * reads a file and adds buffer_size number of bytes into a buffer array,
 * when the last character from the buffer is returned buf_in reads in another buffer_size number of bytes into the buffer
 */
char buf_in(int open_file, char *buffer, int buffer_size){
    if(global_in_pointer == &buffer[buffer_size]){

        int buffercheck = read(open_file, buffer, buffer_size);/*read buffer_size number of bytes into the buffer*/

        if(buffercheck < buffer_size){ /*when it reads in less than 16 bytes add end of string character '\0'*/
            int i = buffercheck;
            buffer[i] = '\0';
        }
        if(buffercheck > 1){
            global_in_pointer = &buffer[0]; /*point at start of buffer*/
        }
        else if(buffercheck == 0){ /*end of file*/
            global_in_pointer = NULL;
            return '\0';
        }
        else
            perror("read failed");
    }
    char value = *global_in_pointer;
    if(value == '\0'){
        global_in_pointer = NULL;
        return;
    }
    global_in_pointer++;
    return value;
}
```

The function has 3 parameters the open file for reading, the buffer and the buffer size. At the top of the file two global pointers are declared and defined to be used in buf_in, buf_out and buf_flush. The first thing the function does is to check if a global pointer which keeps track of current byte inside the buffer is pointing at the end of the buffer. If the pointer is pointing at the end it reads 16 bytes from the opened file. Then it checks if it read less than 16 bytes, if it did the buffer adds an end of string sign to the end of added bytes to handle if old bytes from previous calls where left inside the buffer. After that it simply checks if it was able to read the file successfully, and if it was successful the global pointer points at the start of the buffer. Lastly the value at the pointer is returned and the pointer is set to point on the next element in the buffer.

Main for task 1:



```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "io_functions.c"
#define buffer_size 16

int main(int argc, char *argv[]){
    char *in_buffer = (char *)malloc(buffer_size);
    global_in_pointer = &in_buffer[buffer_size];
    int rfd = open(argv[1], O_RDONLY); /*read file descriptor*/

    if(rfd < 0){
        perror("cant open read file or cant open write file");
    }
    else{
        while(1){
            char a = buf_in(rfd, in_buffer, buffer_size);
            if(global_in_pointer == NULL){
                break;
            }
            write(1, &a, 1);
        }
        write(1, "\n", 1);
    free(in_buffer);
    close(rfd);
    return 0;
    }
}
```

This main creates a 16 byte buffer and sets a global pointer to point at the end of the buffer. After that it opens a file in read only mode and checks if it successfully opened the file. If the file is opened successfully it will read the file until the global pointer is set to NULL inside the buf_in function which means that there is no more to read from the file. In the end the allocated buffer is freed, and the opened file is closed.

Method of verification

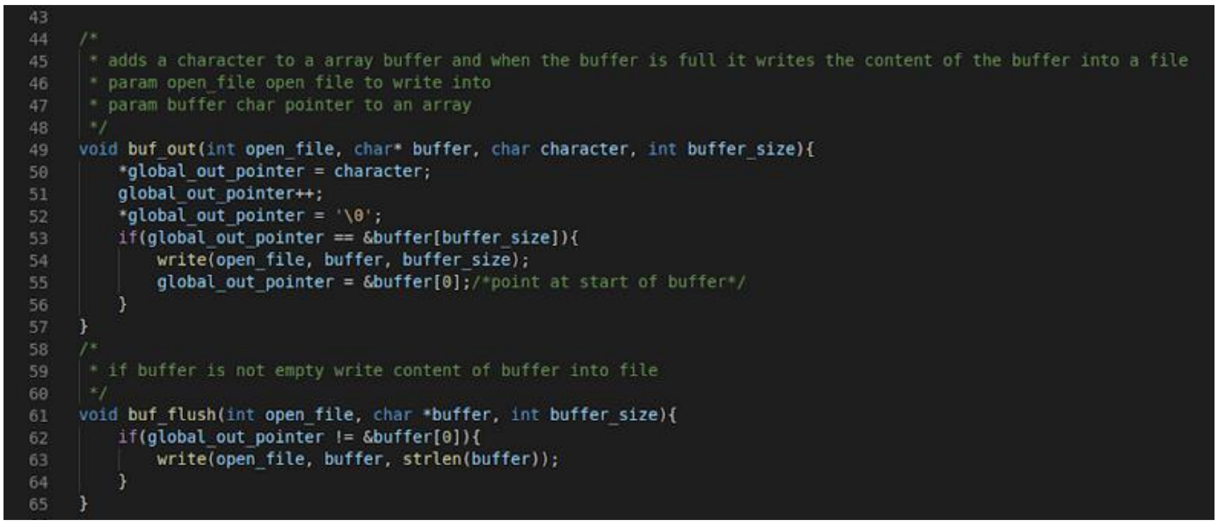
By writing every character in the terminal that was returned from the function buf_in we could compare the text in the terminal with the text inside a file to be sure that the function was working correctly.

Task 2

The objective of task 2 was to implement a function buf_out, that takes a buffer as argument and writes the content of the buffer into a file if it is full, and a function buf_flush which writes everything from the buffer into a file unless it's empty.

Code

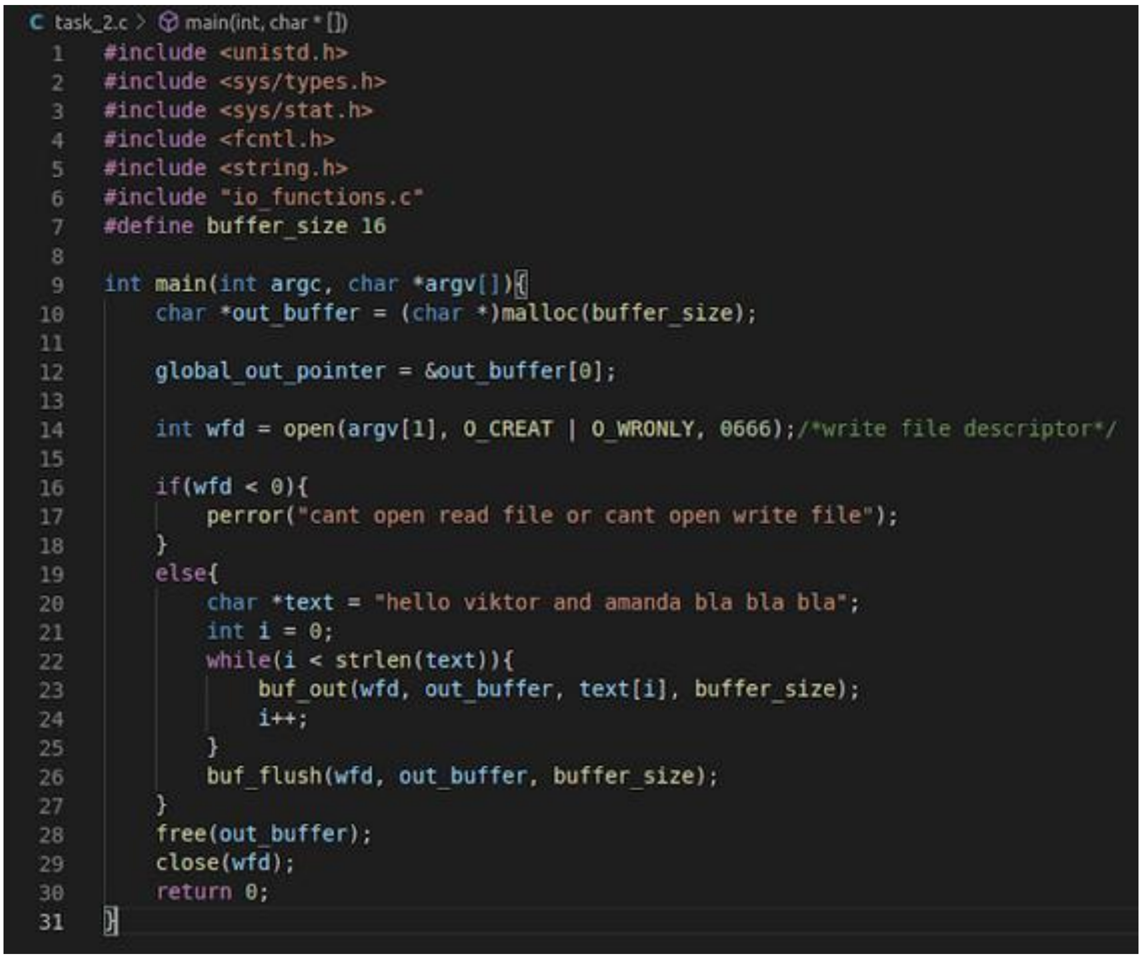
Function buf_out and buf_flush:



```c

/*
 * adds a character to a array buffer and when the buffer is full it writes the content of the buffer into a file
 * param open_file open file to write into
 * param buffer char pointer to an array
 */
void buf_out(int open_file, char* buffer, char character, int buffer_size){
    *global_out_pointer = character;
    global_out_pointer++;
    *global_out_pointer = '\0';
    if(global_out_pointer == &buffer[buffer_size]){
        write(open_file, buffer, buffer_size);
        global_out_pointer = &buffer[0];/*point at start of buffer*/
    }
}
/*
 * if buffer is not empty write content of buffer into file
 */
void buf_flush(int open_file, char *buffer, int buffer_size){
    if(global_out_pointer != &buffer[0]){
        write(open_file, buffer, strlen(buffer));
    }
}
```

The function buf_out takes in four arguments: the open file that will be written in, an array that represent the buffer, a character and the size of the buffer. The first thing that is done in the function is to put the new character to the position that the global_ out_pointer is pointing to and then move the pointer to the next position in the buffer and put a '\0' character there to make sure that the content from the previous buffer isn't included. Then it is checked if the global pointer is pointing to the last element in the buffer. If that is true it means that the buffer is full and all elements from the buffer are written into the open file and then the global pointer is set to the start of the buffer and then the function ends. If the buffer isn't full the function ends without writing anything. The second function from task 2 is buf_flush. It takes in three arguments: an open file, an array that represent the buffer and the size of the buffer. This function will write everything from the buffer if it contains anything, the buffer is empty if the global pointer points on the first position of the array. If the buffer is empty buf_flush won't do anything.

Main for task 2:



```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include "io_functions.c"
#define buffer_size 16

int main(int argc, char *argv[]){
    char *out_buffer = (char *)malloc(buffer_size);

    global_out_pointer = &out_buffer[0];

    int wfd = open(argv[1], O_CREAT | O_WRONLY, 0666);/*write file descriptor*/

    if(wfd < 0){
        perror("cant open read file or cant open write file");
    }
    else{
        char *text = "hello viktor and amanda bla bla bla";
        int i = 0;
        while(i < strlen(text)){
            buf_out(wfd, out_buffer, text[i], buffer_size);
            i++;
        }
        buf_flush(wfd, out_buffer, buffer_size);
    }
    free(out_buffer);
    close(wfd);
    return 0;
}
```

This main works similar to the main in task 1. It creates a buffer and makes a global pointer point to the first position of the buffer. Then a file is opened in write mode and a test is implemented to make sure the file is opened. If it's opened the string stored in text is written into the file using the buf_out function and then the buf_flush function.

Method of verification

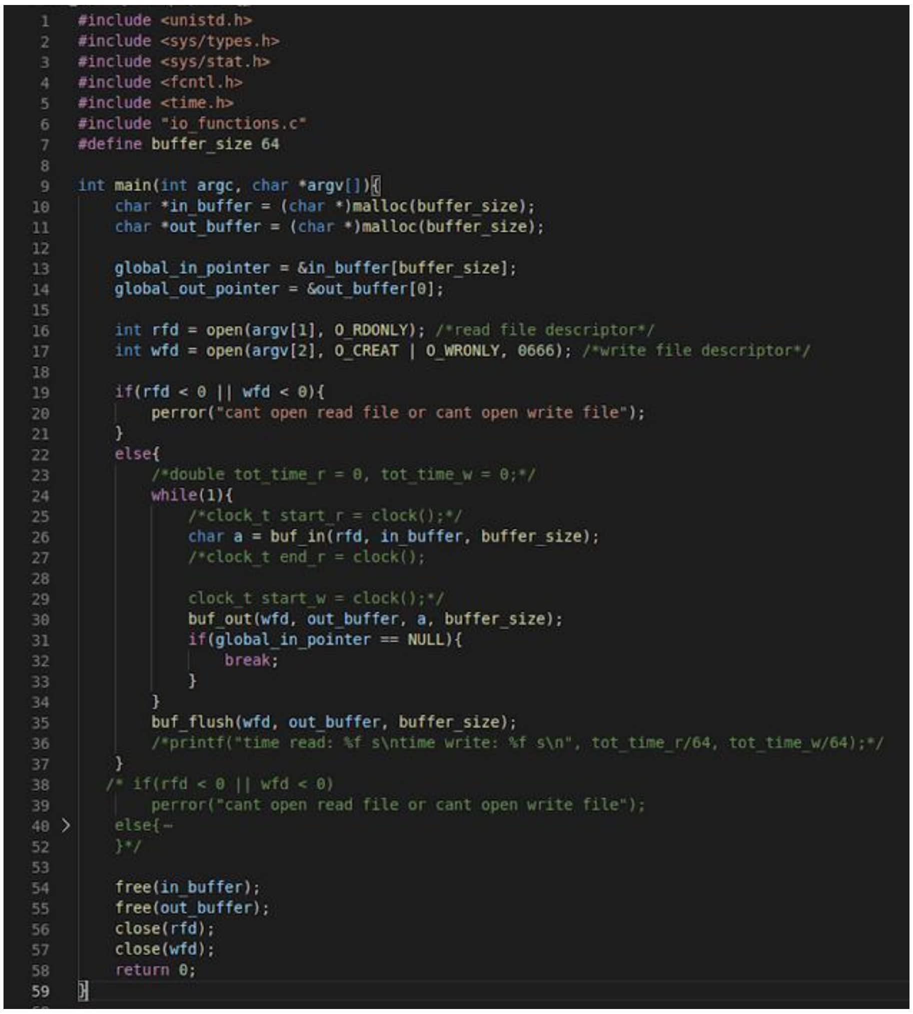
To check if the functions worked, we compared the text in main with the text that the functions wrote into a file. If the text for both places was the same, the function worked as expected. This was tested on multiple sized input strings.

Task 3

This tasks purpose was to combine the previous two tasks to create a program that could copy a file. Furthermore, the next step was to research weather reading 1 byte at a time was faster or slower than using a buffer by comparing time it took to read one byte on different file sizes. The programs time to execute the functions was compared when changing buffer sizes as well. Lastly task 3: s program was compared to the terminals cp command to check which one would be faster.

Code

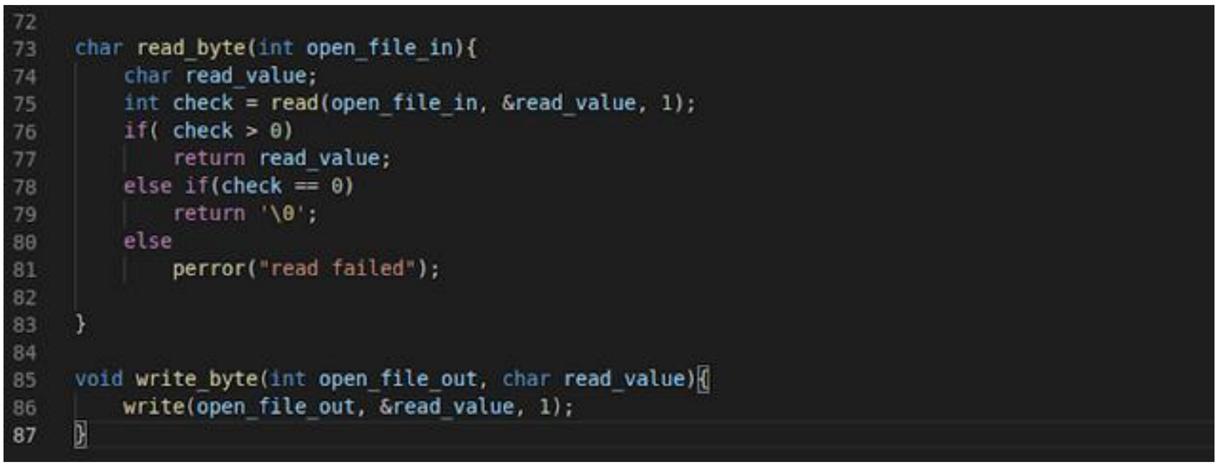
Task 3 main copy program:



```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include "io_functions.c"
#define buffer_size 64

int main(int argc, char *argv[]){
    char *in_buffer = (char *)malloc(buffer_size);
    char *out_buffer = (char *)malloc(buffer_size);

    global_in_pointer = &in_buffer[buffer_size];
    global_out_pointer = &out_buffer[0];

    int rfd = open(argv[1], O_RDONLY); /*read file descriptor*/
    int wfd = open(argv[2], O_CREAT | O_WRONLY, 0666); /*write file descriptor*/

    if(rfd < 0 || wfd < 0){
        perror("cant open read file or cant open write file");
    }
    else{
        /*double tot_time_r = 0, tot_time_w = 0;*/
        while(1){
            /*clock_t start_r = clock();*/
            char a = buf_in(rfd, in_buffer, buffer_size);
            /*clock_t end_r = clock();

            clock_t start_w = clock();*/
            buf_out(wfd, out_buffer, a, buffer_size);
            if(global_in_pointer == NULL){
                break;
            }
        }
        buf_flush(wfd, out_buffer, buffer_size);
        /*printf("time read: %f s\ntime write: %f s\n", tot_time_r/64, tot_time_w/64);*/
    }
   /* if(rfd < 0 || wfd < 0)
        perror("cant open read file or cant open write file");
    else{…
    }*/

    free(in_buffer);
    free(out_buffer);
    close(rfd);
    close(wfd);
    return 0;
}
```

This code is very similar to the previous mains only difference is that the functions buf_in, buf_out and buf_flush is used together to read from one file and write into another. The program stops when the global_in_pointer is a NULL pointer since that means there is nothing more to read from the file. Then in the end the buffers are freed, and the files closed.

Read_byte and write_byte functions:



```
char read_byte(int open_file_in){
    char read_value;
    int check = read(open_file_in, &read_value, 1);
    if( check > 0)
        return read_value;
    else if(check == 0)
        return '\0';
    else
        perror("read failed");

}

void write_byte(int open_file_out, char read_value){
    write(open_file_out, &read_value, 1);
}
```

Functions used when comparing if reading and writing one byte at a time was slower or faster than using a buffer.

Method of verification

The main method used to check if the program could successfully copy a file was the terminal command “diff”. When the diff command was performed there were no difference between the copied file and the original.

Time measures and differences:

Reading and writing single byte Buffered vs non buffered(all values are not exact since it gave slightly different answers each time, the values in the tables is the most common value given):

File of size 16 bytes

buffer size	no buffer	16	32	64
read	0.000008 s	0.000003 s	0.000003 s	0.000003 s
write	0.000017 s	0.000005 s	0.000003 s	0.000002 s

File of size 33 bytes

buffer size	no buffer	16	32	64
read	0.000008 s	0.000006 s	0.000005 s	0.000005 s
write	0.000017 s	0.000010 s	0.000008 s	0.000005 s

Time difference from copying large file (100 kb) with different buffer sizes and the terminal command cp(all values are not exact since it gave slightly different answers each time, the vaues in the tables is the most common value given:

buffer size	terminal cp	16	32	64
time	0.004 s	0.027 s	0.017 s	0.011 s