Max Niia, Mattias Karlsson och Johan Larsson

Lab 4: Input-Output Operations and Buffers

Task 1: Buffered input

Buf_in

The function begins by initializing a buffer, b. This is a struct holding a char array of size 16, the size of the buffer, the number of bytes in the buffer and how many bytes have already been read. This buffer is static so that on subsequent calls the buffer remains and can be read from. On first call to the function the buffer is filled with 0 and number of bytes is set to 0.

Whenever the function is called the first thing to do is check if the buffer is empty, if this is the case then the buffer needs to be filled. This is done by using lseek() and read(). So that the function reads past the first 16 bytes lseek() is used to skip to the next 16 byte block and then read() fills the buffer with the next 16 bytes. Also, the number of elements in the buffer is updated.

If the buffer has bytes that have not been outputted, then output is set to the next byte and the number of elements decreases by one.

For the function to compile with the -ANSI flag we had to make any for-loop into a while loop.

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define COMMENT(X)
#define nrBytes 16

typedef struct buffer_ {
    char curSize; COMMENT(THIS IS NUMBER OF BYTES IN BUFFER)
    char bufSize; COMMENT(THIS IS MAXIMUM SIZE OF BUFFER)
    unsigned int fileP; COMMENT(THIS IS OFFSET FOR FILE)
    char data[nrBytes]; COMMENT(BUFFER)
} buffer;

char buf_in(const char* fileName){
    static buffer b;
    static char cond = 1;
    if(cond){
        b.bufSize = nrBytes;
        b.curSize = 0;
        int i = 0;
        while(i < b.bufSize){
            b.data[i] = 0;
            i++;
        }
        cond = 0;
    }
    char output = 1; COMMENT(INITIALIZE OUTPUT VARIABLE)
    if(b.curSize == 0){
        int fDesc = open(fileName, O_RDONLY); COMMENT(OPEN FILE)
        lseek(fDesc, b.fileP, SEEK_SET); COMMENT(GOES TO OFFSET)
        read(fDesc, b.data, b.bufSize); COMMENT(READS DATA)
        close(fDesc); COMMENT(CLOSE FILE)
        b.curSize = nrBytes; COMMENT(SET CURSIZE TO NR BYTES READ)
        b.fileP += nrBytes; COMMENT(ADVANCE OFFSET)
    }else{
        output = b.data[b.bufSize - b.curSize]; COMMENT(TAKE NEXT VALUE IN BUF
FER)
        b.curSize--; COMMENT(DECREASE CURRENT SIZE)
    }
    return output; COMMENT(RETURN NEXT BYTE OR 0 IF ONLY READ)
}
```

Task 2: Buffered output

Buf_out

Buf_out() takes a char byte to be written into a file. The output-buffer needs to be used by multiple functions so it is initialized outside of the functions. if the buffer is full then first the function writes it into the file using write() and then sets the number of elements to 0. Lastly the byte gets added to the buffer.

```
buffer bOut = {.curSize= 0, .bufSize = nrBytes}; COMMENT(initialize buffer)

void buf_out(const char* destination, char byte){
    if (bOut.curSize == bOut.bufSize){
        int fdesc = open(destination, O_WRONLY | O_APPEND);
        write(fdesc, bOut.data, bOut.bufSize);
        close(fdesc);
        bOut.curSize = 0;
    }
    bOut.data[bOut.curSize] = byte;
    bOut.curSize++;
}
```

Buf_flush

Only takes the file as input and writes into it the entire buffer and sets the number of elements to 0.

```
void buf_flush(const char* destination){
    int fdesc = open(destination, O_WRONLY | O_APPEND);
    write(fdesc, bOut.data, bOut.curSize);
    close(fdesc);
    bOut.curSize = 0;
}
```

Task 3: Performance evaluation

Takes two inputs from the command line, the first one being the source and the second one being the destination. Uses <time.h> to keep track of how much time the functions take. Printstuff copies with buffers, while unprintstuff does not use buffers.

```c
int main(int argc, char** argv){
    if(argc != 3){
        printf("Error too many/few arguments!\n");
        return 1;
    }
    const char* fileName = argv[1];
    const char* outFile = argv[2];
    time_t timer = clock();
    printStuff(fileName, outFile, fileSize);
    printf("Time for buffer: %f\n", (double) (clock() - timer)/CLOCKS_PER_SEC)
;
    timer = clock();
    unprintStuff(fileName, outFile, fileSize);
    printf("Time for unbuffer: %f\n", (double) (clock() - timer)/CLOCKS_PER_SE
C);
    printf("\n");
    buf_flush(outFile);
    return 0;
}

void printStuff(const char* src, const char* dest, int fileSize) {
    int i = 0;
    char copy = 0;
    int offset = fileSize% nrBytes != 0;
    int steps = fileSize + (fileSize / nrBytes) + offset;
    printf("STEPS: %d", steps);
    while(i < steps) {
        copy = buf_in(src);
        if(copy != 1)
            buf_out(dest, copy);
        i++;
    }
    buf_flush(dest);
}

void unprintStuff(const char* src, const char* dest, int fileSize) {
    int i = 0;
    char copy = 0;
    int offset = fileSize% nrBytes != 0;
    int steps = fileSize + (fileSize / nrBytes) + offset;
    printf("STEPS: %d", steps);
    while(i < steps) {
        copy = unbuf_in(src);
        if(copy != 1)
```

```
            unbuf_out(dest, copy);
        i++;
    }
}

char unbuf_in(const char* fileName){
    static int index = 0;
    char output = 0;
    int fDesc = open(fileName, O_RDONLY); COMMENT(OPEN FILE)
    lseek(fDesc, index, SEEK_SET); COMMENT(GOES TO OFFSET)
    read(fDesc, &output, 1); COMMENT(READS DATA)
    close(fDesc); COMMENT(CLOSE FILE)
    index++;
    return output; COMMENT(RETURN NEXT BYTE OR 0 IF ONLY READ)
}

void unbuf_out(const char* destination, char byte){
    int fdesc = open(destination, O_WRONLY | O_APPEND);
    write(fdesc, &byte, 1);
    close(fdesc);
}
```