Lab 4: Input-Output Operations and Buffers

Anton Söderlund, Marcus Nyström
2018-10-19

Task 1

We wrote a program which open files to read using a buffer implemented by us. We open the file by creating a file descriptor *infile* using the standard c API open(). We pass the file with the flag O_RDONLY.

The function buf_in takes the in_buffer, file descriptor and a index tracker as arguments. The index tracker keeps track of where the last entry of the buffer is, so that you can return the last character put in the buffer. We tested this function for files of sizes 16,32,64 bytes and 100kB.

Task 2

In this task we use an additional file descriptor *outfile* using open with the flags O_WRONLY | O_CREATE, meaning we can create and write to a given filename. The flag 00700 is used to give us permissions over the newly created file.

Function buf_out takes an out_buffer, an array of chars to be put into the buffer, a file descriptor and an index tracker. The tracker does the same as in task 1, keeps track of where in the buffer the latest character added is located. We have conditions set so when the buffer is full we write the content to the file, and reset the tracker to the beginning of the buffer.

Additionally we added buf_flush which just writes the remaining characters in the buffer. It uses the same out_tracker as the out_buffer so it knows how many characters remain. We tested this function for files of sizes 1,16,32,64 bytes and 100kB.

Task 3

We start with using the command line arguments to create our file descriptors.

```
int infile = open(argv[1],O_RDONLY);
int outfile = open(argv[2],O_WRONLY | O_CREAT,00700);
```

After running our program we got no differences between infile and outfile when using the diff shell command.

```
pi@raspberrypi:~/lab4 $ ./main writeFrom writeTo
real time seconds and micro seconds: s:0 us:595305
pi@raspberrypi:~/lab4 $ diff writeFrom writeTo
pi@raspberrypi:~/lab4 $
```

Figure 1, diff shell command comparison

Time tests with different buffer sizes

```
pi@raspberrypi:~/lab4 $ ./main
usecRead: 25 usecWrite: 26
pi@raspberrypi:~/lab4 $ _
```

Figure 2, 1 byte read and write

```
pi@raspberrypi:~/lab4 $ ./main
real time seconds and micro seconds: s:15 us:868413
pi@raspberrypi:~/lab4 $ _
```

Figure 3, 1 byte buffer with 100kB file

```
pi@raspberrypi:~/lab4 $ ./main
real time seconds and micro seconds: s:1 us:41022
pi@raspberrypi:~/lab4 $ _
```

Figure 4, 16 byte buffer, 100kB file

```
pi@raspberrypi:~/lab4 $ ./main
real time seconds and micro seconds: s:0 us:528085
pi@raspberrypi:~/lab4 $
```

Figure 5, 32 byte buffer, 100kB file

```
pi@raspberrypi:~/lab4 $ ./main
real time seconds and micro seconds: s:0 us:280980
pi@raspberrypi:~/lab4 $ _
```

Figure 6, 64 byte buffer, 100kB file

```
pi@raspberrypi:~/lab4 $ time ./main

real    0m0.326s
user    0m0.090s
sys     0m0.230s
pi@raspberrypi:~/lab4 $ _
```

Figure 7, 64 byte buffer, 100kB file using time shell command

```
pi@raspberrypi:~/lab4 $ time cp test writtenTo

real    0m0.110s
user    0m0.060s
sys     0m0.050s
```

Figure 8, time on the cp shell command

We can see how the time taken is reduced by how large the buffer is (as long as the file is larger than the buffer), because a larger buffer reduces the amount of read and write operations needed. The time shell command is fairly close to our timing using the C API, but the cp command is still significantly faster.

Appendix 1: The program

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <time.h>

/*
* The size of the buffers used
*/
#define BUF_SIZE 16

/*
* buf_in returns a char from in_buffer, if buffer is empty reads BUF_SIZE
* bytes from infile.
*/
char buf_in(char* in_buffer, int infile, int *in_tracker);

/*
* buf_out stores inChars into out_buffer, if out_buffer fills up
* out_buffer is written to outfile, repeated until all chars in inChars
* has been used.
*/
void buf_out(char* out_buffer,char *inChars,int outfile, int* out_tracker);

/*
* buf_flush writes remaining chars in out_buffer to outfile.
*/
void buf_flush(char *out_buffer,int outfile, int* out_tracker);

int main(int argc, char* argv[]) {
    char* in_buffer  = malloc(sizeof(char)*BUF_SIZE);
    char* out_buffer = malloc(sizeof(char)*BUF_SIZE);
    /*
    * Open infile in read-only.
    * Open outfile in write-only, if it doesn't exist create with rwx
    * permissions.
    */
    int infile = open(argv[1],O_RDONLY);
    int outfile = open(argv[2],O_WRONLY | O_CREAT,00700);

    /*
    * out_tracker tracks number of bytes (chars) in out_buffer.
    * in_tracker tracks the next byte (char) to return from in_buffer.
    */
    int *out_tracker = malloc(sizeof(int));
    *out_tracker = 0;
    int* in_tracker = malloc(sizeof(int));
    *in_tracker = -1;
```

```
    /*
    * charOut[] is used to hold the char returned from buf_in, it is in
    * array form because buf_out takes an array as input to allow it to
    * take a large number of chars at once.
    */
    char charOut[] = "a";
    /*
    * Time API used to time the execution.
    */
    struct timeval startTime,endTime;
    gettimeofday(&startTime,NULL);

    while (charOut[0] != '\0') {
        charOut[0] = buf_in(in_buffer,infile,in_tracker);
        buf_out(out_buffer,charOut,outfile,out_tracker);
    }
    buf_flush(out_buffer,outfile,out_tracker);
    gettimeofday(&endTime,NULL);
    /*
    * Time computed.
    */
    int usec_time = endTime.tv_usec - startTime.tv_usec;
    int sec_time = endTime.tv_sec - startTime.tv_sec;
    if(usec_time < 0){
      sec_time = sec_time - 1;
      usec_time = usec_time + 1000000;
    }
    printf("real time seconds and micro seconds: s:%d us:%d\n",
sec_time,usec_time);
    close(infile);
    close(outfile);
}


void buf_flush(char* out_buffer, int outfile, int *out_tracker){
    write(outfile, out_buffer, *out_tracker);
    *out_tracker = 0;
}

void buf_out(char* out_buffer, char* inChars, int outfile, int* out_tracker)
{
    int i;
    int j = 0;
    /*
    * Counts number of chars
    */
    while(inChars[j] != '\0'){
        j = j + 1;
    }
```

```
    /*
    * Runs until all chars have been put into the buffer
    */
    for (i = 0; i < j; i++) {
        /*
        * Writes content of buffer to file when buffer is full
        */
        if(*out_tracker == BUF_SIZE) {
            write(outfile, out_buffer, BUF_SIZE);
            *out_tracker = 0;
        }
        out_buffer[*out_tracker] = inChars[i];
        *out_tracker = *out_tracker + 1;
    }
}

char buf_in(char* in_buffer, int infile, int* in_tracker) {
    /*
    * If in_tracker is -1 the in_buffer is empty and we read from infile
    * and returns the first char.
    * else the next char is returned and if the end of the buffer is
    * reached in_tracker is updated to -1.
    */
    if(*in_tracker == -1) {
        int bytes = read(infile,in_buffer,BUF_SIZE);
        /*
        * If bytes is less than BUF_SIZE we reached the end of the file.
        */
        if(bytes < BUF_SIZE) {
            in_buffer[bytes] = '\0';
        }
        *in_tracker = *in_tracker + 2;
        /*
        * Handles the special case of BUF_SIZE == 1
        */
        if(*in_tracker == BUF_SIZE){
            *in_tracker = -1;
        }
        return in_buffer[0];
    }else{
        int tmp = *in_tracker;
        if(*in_tracker == BUF_SIZE-1){
            *in_tracker = -1;
        }
        else{
            *in_tracker = *in_tracker + 1;
        }
        return in_buffer[tmp];
    }
}
```