TD3 : Formatted I/O Library

Master M1 MOSIG, Grenoble Universities

Lenka Kuníková Lina Marsso

26/10/2014

1 Introduction

In this project we focus on input in and output from file. Our goal was to implement library usable for formatted I/O on top of the system calls read and write. In other words we try to reimplement few functions from standard library stdio.h, trying to reduce number of system calls needed using the buffer.

In the first part, we talk about few implementation details of our functions, later we show that we succeeded in our attempt to reduce number of system calls and illustrate how time costly this low level calls can be when we compare speed of two programs: one using only system calls and one using our small library my_stdio.h.

2 Implementation

2.1 Struct my_file

Our structure my_file consists of buffer in which we temporarily store information that we read from a file or that user tried to write in. We use two pointers that show us in which part of the buffer are valid data. Of course, we need to remember handler we got from the system call write / read. We also have variables to distinguish between read and write mode, to remember if we already met EOF and one that knows whether buffer was already filled or it is empty.

2.2 My_fopen

In this function we allocate the memory needed for the whole structure and we initialise all variables. Both pointers will point to the beginning of the buffer. According to the specified mode we open the file with appropriate flags.

2.3 My_fclose

In this function we need to close the file and free all allocated memory. If the mode of file is set to write, we also need to write the content of the buffer into the file.

2.4 My_fread

At first, the function should check whether the mode is set to read, if not, it returns -1. Then we need to distinguish between different cases. In case it is the first read operation, the buffer is empty and we need to fill it. In case the amount of data the user requests to read is less than data available, it is the simplest case and we just give requested data to the user. Another case is when we meet EOF before expected. It can be detected thanks to the pointer that points to the end of valid data. In this case we just give user less data than he wanted. It can also happen that user wants more than what is disponsible in the buffer. In this case, we give him all we have, we refill the buffer and give him remaining data. We keep refilling data until we get enough.

2.5 My_fwrite

As in the previous function, we should test whether the mode is set to write. If it is, user data will be stored. If there is enough space in the buffer, we just write all user data in. Once the buffer is full, we write its content in a file using actual system call. We keep writing data in and flushing full buffer until all user data are processed.

2.6 My_feof

This function should say whether we met EOF during previous read. For this purpose we use variable defined in struct my_file, which is updated in function my_fread when EOF is recognized. Here we just return its value.

2.7 My_fprintf

During implementation of this function we needed to deal with list of arguments of variable length. To work with them we use variable of type va_list. The function processes string format in a while cycle. All normal characters are directly written using my_fwrite function. When we reach character '%', we switch to one of three possibilities(%d, %s, %c). When we are dealing with one character (%c), we pop the argument from va_list, and we write its value to the file. When we are dealing with whole string, first we need to discover the length of it using strlen. Then we write appropriate number of characters stored in argument to the file. When we reach %d we need to transform integer value to a char, then we can write it. While cycle ends when we reach \0 and during whole cycle we count number of successfully processed arguments.

2.8 My_fscanf

Function is very similar to previous one but it is a bit more complicated. In case of %s, for example, we do not know the length of string at the beginning. We read characters from file one by one and we look for white characters. When we reach one, we assume it is the end of string. We add \0 to the end and we give it back to user. For %d it is similar. We recognize end of the number when we read first non-digit character.

3 Evaluation

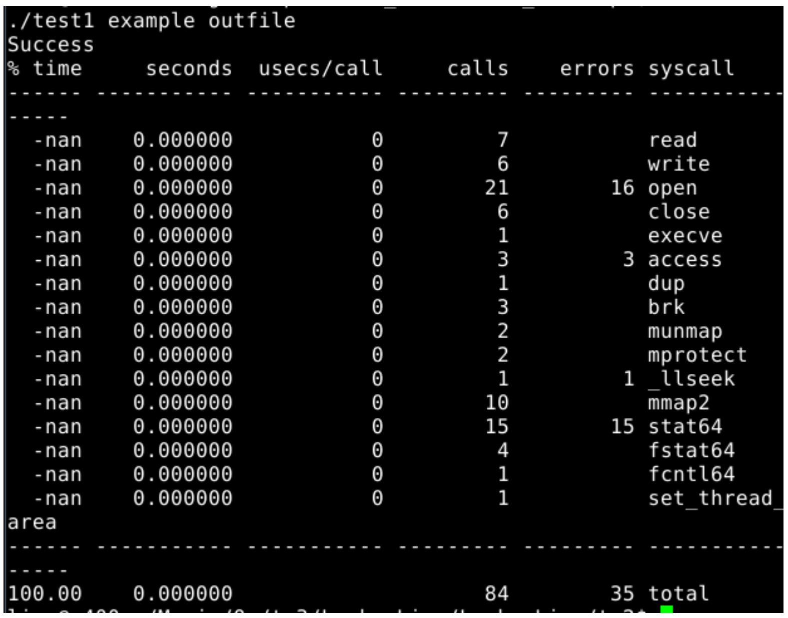
For evaluate our library, we use one test he copie one file in an other file buffered for our library and we write the same test but using the standard open, read and write.

We evaluate in the begining the time for copy two file with our library, and with the standard open, read and write. We have also one ratio :

$$\left(\frac{My\_stdio\_time\_execution}{Standar\_Read\_Write\_time\_execution}\right) \tag{1}$$

We calculate the time of the execution with a script time.sh. We can see

Library	Time (microsecondes)	Size of file
libmy_sdio	4	8.0K
Standar	36	8.0K

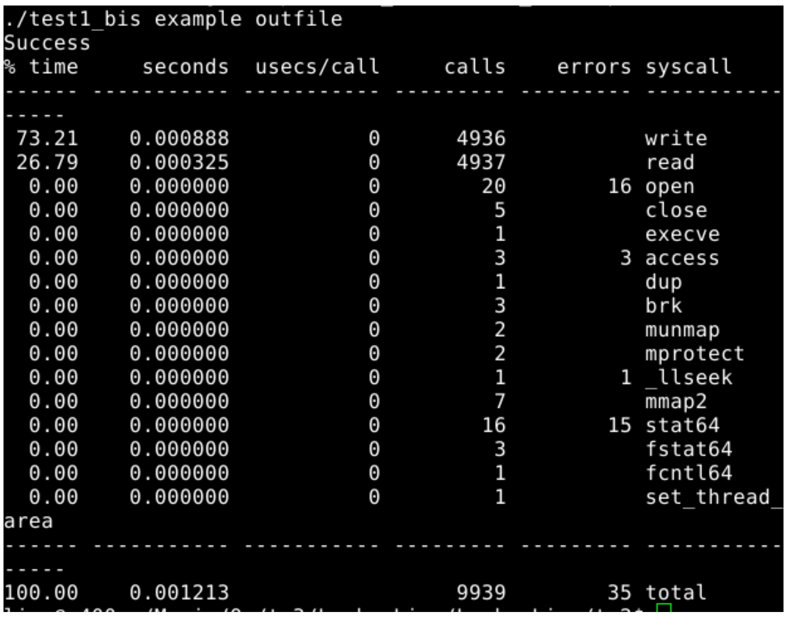


```
./test1 example outfile
Success
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
-----
  -nan    0.000000           0         7           read
  -nan    0.000000           0         6           write
  -nan    0.000000           0        21        16 open
  -nan    0.000000           0         6           close
  -nan    0.000000           0         1           execve
  -nan    0.000000           0         3         3 access
  -nan    0.000000           0         1           dup
  -nan    0.000000           0         3           brk
  -nan    0.000000           0         2           munmap
  -nan    0.000000           0         2           mprotect
  -nan    0.000000           0         1         1 _llseek
  -nan    0.000000           0        10           mmap2
  -nan    0.000000           0        15        15 stat64
  -nan    0.000000           0         4           fstat64
  -nan    0.000000           0         1           fcntl64
  -nan    0.000000           0         1           set_thread_
area
------ ----------- ----------- --------- --------- ----------
-----
100.00    0.000000                    84        35 total
```

Figure 1: Number of system call of our library

3.1 Table with fragmentation level of differents programs

program	worst_fit	best_fit	first_fit
ls	1.83	1.83	2.48
wc	0.96	0.96	0.96
ps	1.37	1.36	1.37
hostname	2.42	2.42	3.23



```
./test1_bis example outfile
Success
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
-----
 73.21    0.000888           0      4936           write
 26.79    0.000325           0      4937           read
  0.00    0.000000           0        20        16 open
  0.00    0.000000           0         5           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           dup
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         2           mprotect
  0.00    0.000000           0         1         1 _llseek
  0.00    0.000000           0         7           mmap2
  0.00    0.000000           0        16        15 stat64
  0.00    0.000000           0         3           fstat64
  0.00    0.000000           0         1           fcntl64
  0.00    0.000000           0         1           set_thread_
area
------ ----------- ----------- --------- --------- ----------
-----
100.00    0.001213                  9939        35 total
```

Figure 2: Number of system call with read and write

Figure 3: Memory requested and memory used with best fit strategy

4 Conclusion

The goal of this project was to get familiar with the concept of memory managment and I think we succeeded. We managed to implement a simple version of memory allocator which allows users to allocate and free memory.

During the implementation we had to deal with several problems. For example we needed to find a way on how to determine positions of free blocks when we do not have linked lists of them, or think about the algorithm that helps us merge contiguous blocks of free space. Moreover, we implemented the three most common algorithms for finding an appropriate free block (first-fit, best-fit and worse-fit). We are now able to describe pros and cons for each of them.

In the second part, we added some other features to our memory allocator. All of them related to security checks. Now we know that implementing functions for free and alloc memory is not enough and we also need to deal with situations where users (on purpose or by chance) do not act the way we expect. The control we are making is not perfect but it helps us to deal with many cases that may happen.

Finnaly, we can see an application of our allocator. We can analyse it's performance with many apllications like ls, hostname... It was a very interesting project.