TD3 : Formatted I/O Library

Master M1 MOSIG, Grenoble Universities

Lenka Kuníková Lina Marsso

26/10/2014

1 Introduction

In this project we have to implement a I/O library. And evaluate their performances in different applications.

Firstly in this report we give answers of questions in the subject. Then we present you our implementation. Then we speak about the safety checks implementation. Finnaly, we analyse the performance and conclude this project.

2 Subject questions

2.1 Question 1

3 Implementation

3.1 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

4 Safety checks

5 Evaluation

In the begining the library was static. It's is more fast for access, but the library take many place in memory, and we conclude it's more efficient with a dynamic library.

For evaluate our dynamic 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.

5.1 Evaluation of the execution time

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 for example in table 5.1, the ratio is 4/36. Our library is also 9 time more fast.

5.2 Table with time execution of two differents test

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

5.3 Evaluation on the number of system call

A system call is very expensive. It's why we implemented this library, when we try ti reduce the number of the system call thanks to a buffer. Now we would like check if the number of system call of all library is really different then we use standar sytem functions.

With two same test, test1.c and test1_bis.c. One test use our dynamic library, and test1_bis.c use sytems functions. This test have the same goal, copie text from a file in a other file.

With the program strace we explore the number of system call for test1 (Figure 1) and the test1_bis (Figure 2). We can see for example with our library we have 6 calls of write whereas if use directely system function we have 4936 calls of write.

```
./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

5.4 Table with number of system call by test1 and test1_bis

Test	System function	Number
Test1	read	7
Test1_bis	read	4937
Test1	write	6
Test1_bis	write	4936

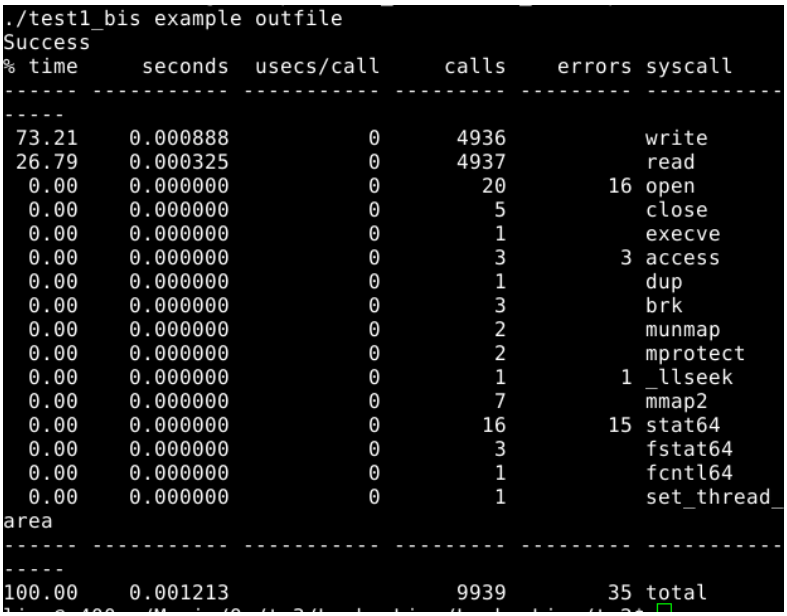

Figure 2: Number of system call with read and write

6 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.