Formatted I/O Library

Master M1 Mosig, Grenoble University

2014

1 Introduction

In this lab, we have to implement a formatted input/output library on top of the input/output system calls provided by the system: `read` and `write`. The main motivation is to provide users with a higher level interface and to minimize the actual number of system calls issued by a program. The higher level interface lets the user handle the data according to its type and not consider it only as a block of bytes. A smaller number of system calls accelerates a priori the execution and is achieved through bufferization.

2 Buffered Inputs/Outputs

The first exercise focuses on bufferization. We use a simplified interface that includes the following four functions:

- ▷ `MY_FILE *my_fopen(char *name, char *mode);`
  Opens an access to a file, where `name` is the path of the file and mode is either "r" for read or "w" for write. Returns a pointer to a `MY_FILE` structure upon success and `NULL` otherwise.
- ▷ `int my_fclose(MY_FILE *f);`
  Closes the access to a file associated to `f`. Returns 0 upon success and -1 otherwise.
- ▷ `int my_fread(void *p, size_t size, size_t nbelem, MY_FILE *f);`
  Reads at most `nbelem` elements of size `size` from file access `f`, that has to have been opened with mode "r", and places them at the address pointed by `p`. Returns the number of elements actually read, 0 if an end-of-file has been encountered before any element has been read and -1 if an error occured.
- ▷ `int my_fwrite(void *p, size_t taille, size_t nbelem, MY_FILE *f);`
  Writes at most `nbelem` elements of size `size` to file access `f`, that has to have been opened with mode "w", taken at the address pointed by `p`. Returns the number of elements actually written and -1 if an error occured.
- ▷ `int my_feof(MY_FILE *f);`
  Returns 1 if an end-of-file has been encountered during a previous read and 0 otherwise.

The following program illustrates the use of this buffering library. In this example, the file which name is given as first argument is copied to the file which name is given as second argument.

```
#include <stdlib.h>
#include "my_stdio.h"
```

```
int main (int argc, char *argv[])
{
  MY_FILE *f1;
  MY_FILE *f2;
  char c;
  int result;

  // for the sake of simplicity we don't
  // print any error messages
  if (argc != 3)
      exit (-1);

  f1 = my_fopen(argv[1], "r");
  if (f1 == NULL)
      exit (-2);

  f2 = my_fopen(argv[2], "w");
  if (f2 == NULL)
      exit (-3);

  result = my_fread(&c, 1, 1, f1);
  while (result == 1)
    {
      result = my_fwrite(&c, 1, 1, f2);
      if (result == -1)
          exit(-4);
      result = my_fread(&c, 1, 1, f1);
    }
  if (result == -1)
      exit(-5);
  my_fclose(f1);
  my_fclose(f2);
  return 0;
}
```

Question .1. *Write the content of the file* `my_stdio.h` *that contains all the prototypes of our library as well as the definition of the type* `MY_FILE`*. This type can be a structure containing all the fields required for the implementation of the functions of our library or, preferably, a still undefined structure that will be defined in* `my_stdio.c`*.*

Question .2. *Write the content of the file* `my_stdio.c` *that contains the implementation of the functions whose prototypes are in* `my_stdio.h`*, and the structure associated to* `MY_FILE`*.*

Question .3. *Write a* `Makefile` *to compile the previous exemple or an example of your own with your implementation of the library. Test that the behavior of the program is correct. We provide the test in* `test_buffered.c`

Question .4. *Check that your library issues the expected number of system calls by using the command* `strace (man strace)` *and counting the number of calls (*`-c` *option) to* `read` *and* `write`*.*

Question .5. *Write an equivalent program (e.g* `test_buffered_std.c`*) but using the standard* `open, close, read and write` *functions. Compare the number of system calls, as well as the respective execution times of the two programs.*

3 Formatted Inputs/Outputs

Now, we would like to improve our library by adding formatted inputs/outputs functions similar to `printf` and `scanf`. These functions accept a variable number of arguments, thus you will have to use the functions `va_start`, `va_end` and `va_arg` provided by the system. The two new functions you have to implement are the following:

- ▷ `int my_fprintf(MY_FILE *f, char *format, ...);`
  Writes to `f` the arguments given after `format` using the format given in `format`. Returns the number of arguments successfully written or -1 upon error.
- ▷ `int my_fscanf(MY_FILE *f, char *format, ...);`
  Stores at the addresses given after `format` the values read from the access associated to `f` using the format given in `format` as described in the following. Returns the number of arguments successfully read or the value `EOF` if an end-of-file is encountered before any value is read or upon error.

The content of format is similar to the one used with classical functions such as `printf` and `scanf` and you can refer to the manpages of these functions is some details are missing in this text. Three data types will be handled by our library:

- ▷ character: %c
- ▷ string: %s
- ▷ integer: %d

The following example illustrates the use of these two functions. It writes to the file given as second argument the name of the input file given as first argument, then, for each character contained in the input file, the character itself and its ASCII code.

```
#include <stdlib.h>
#include "my_stdio.h"

int main (int argc, char *argv[])
{
  MY_FILE *f1, *f2;
  char c;

  // for the sake of simplicity we don't
  // print any error messages
  if (argc != 3)
      exit (-1);

  f1 = my_fopen(argv[1], "r");
  if (f1 == NULL)
      exit (-2);
```

```
  f2 = my_fopen(argv[2], "w");
  if (f2 == NULL)
      exit (-3);

  my_fprintf(f2, "Input file: %s\n", argv[1]);
  my_fscanf(f1, "%c", &c);
  while (!feof(f1))
    {
  my_fprintf(f2, "Character %c read, its ASCII code is %d\n", c, c);
      my_fscanf(f1, "%c", &c);
    }

  fclose(f1);
  fclose(f2);
  return 0;
}
```

Question .6. *Add the two functions to your library, possibly using* `my_fread` *and* `my_write` *to implement them. Test your implementation using the given example or one of your own.*

In the last two questions, the goal is to package your implementation as a library. In the previous lab, you have already generated a shared dynamic library. More details about the types of libraries and their creation are given in the appendix.

Question .7. *Change your* `Makefile` *to generate the static library associated with your implementation and link it with the programs you have used as tests. Pay attention to the sizes of your executable files.*

Question .8. *Change your* `Makefile` *to generate the dynamic library associated with your implementation and link it with the programs you have used as tests. Make sure that the execution is possible. What are the sizes of your executables now?*

4 APPENDIX: Library Generation and Usage

Static Library

A static library is associated with a program during the link phase of the compilation process. More precisely, the code of the functions contained in the library is included in the final executable.

A static library is usually stored in a file with a `.a` extension. Its advantage is that it gathers several object files into a single library file, which is simpler to use. It can be installed in standard locations in the system and the compiler knows how to find it when linking. The drawback of such a library is that its code is included in the final executable. If many programs use this library, this can result in a waste of storage space. It can be generated and used as follows:

- ▷ generation of object files:
  `gcc -c my_stdio.c`
- ▷ generation of the library file from object files:
  `ar q libmy_stdio.a my_stdio.o`
- ▷ compilation of a program that uses this library:
  `gcc -o main_program main_program.c -L. -lmy_stdio`

Notice the `-L` option which tells to `gcc` that it should look for libraries in the current directory and the `-l` which let the programmer give a library name to link with the program.

Dynamic Shared Library

A shared library, as opposed to a static library, contains code that is not included in the program during the link process. Instead, the compiler includes code to look for the library at the beginning of the program's execution. The obvious advantage is that the library code is now stored into a single location, the library itself, and there is no more wasted space due to its copies. The less obvious advantage is that, if another program using the same library is in execution, the system can share the memory in which the library has been loaded between several processes, saving memory and load time. Furthermore, by linking the library to the program at runtime, the program naturally takes advantage of possible bugfixes and improvements present in revisions of the library.

As a constraint, a dynamic library used by some program has to be installed on the system otherwise the program execution is not possible. The command `ldd` prints the libraries required for the execution of a given program. By default, these libraries are searched in standard locations by the system. It is possible to change this behavior by storing in the `LD_LIBRARY_PATH` environment variable the list of directories in which shared libraries are located (using the same format as the `PATH` variable). More precisely, dynamic libraries can be generated and used as follows:

- ▷ generation of object files:
  `gcc -c my_stdio.c`
- ▷ generation of the library file from object files:
  `gcc -shared -o libmy_stdio.so my_stdio.o`
- ▷ compilation of a program that uses this library:
  `gcc -o main_program main_program.c -L. -lmy_stdio`
- ▷ execution without any special care:
  `./main_program`
  this prints some error messages...

  ```
  main_program: error while loading shared libraries:
  libmy_stdio.so: cannot open shared object file: No such file or directory
  ```

  we can try to check which libraries are required using `ldd main_program`. This prints:

  ```
  linux-gate.so.1 =>  (0xffffe000)
  libmy_stdio.so.1 => not found
  libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e63000)
  /lib/ld-linux.so.2 (0xb7fc6000)
  ```

- ▷ execution after setting `LD_LIBRARY_PATH`:
  `export LD_LIBRARY_PATH=.`
  `ldd test_format`
  Now prints:

  ```
  linux-gate.so.1 =>  (0xffffe000)
  libmy_stdio.so.1 => ./libstdes.so.1 (0xb7faf000)
  libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e4f000)
  /lib/ld-linux.so.2 (0xb7fcd000)
  ```

  and the execution is possible.

The same effect can be obtained using the `LD_PRELOAD` option.