CIS 3110: Operating Systems

Assignment 2: System call vs Function Call

Due Saturday, Feb. 16, 2019 @ 11:59pm

1. Objective

The objectives of this assignment is to familiarize you with Unix and C Standard Input/Output (I/O). You will code a program that reads a given file using Unix open()/read() system calls as well as C fopen()/fgetc()/fread() functions. Most importantly, you are required to analyze performance differences between these two I/O options.

2. I/O System calls

Basically there are total 5 types of I/O system calls, including

- create: Used to Create a new empty file.
- open: Used to Open the file for reading, writing or both.
- close: Tells the operating system you are done with a file descriptor and Close the file which is pointed by fd.
- read: Read size bytes from the file specified by fd into the memory location.
- write: Writes the bytes stored in buf to the file specified by fd.

In this assignment, you will use the following Unix system calls and C Standard I/O functions. You can check the specification of each function using man.

- open: is a Unix system call that opens a given file.
- read: is a Unix system call that reads a file into a given buffer.
- close: is a Unix system call that closes a given file.
- fopen: is a C standard I/O function that opens a given file.
- fgetc: is a C standard I/O function that reads one byte from a file.
- fread: is a C standard I/O function that reads a file into a given data structure.
- fclose: is a C standard I/O function that closes a give file.

3. Questions

Write a C progrram, unixio.c, which reads a given file using Unix open()/read() system calls as well as C fopen()/fgetc()/fread() functions and measure how long it takes to finish reading the file.

The following code is a template for measuring time elapsed to read a given file with Unix system call read() as well as that elapsed to read the same file with C Stardard's fread(). The program takes three arguments:

1) the name of a file to be read,
2) the number of bytes to read per read( ) or fread( ), and
3) type of I/O calls used (“1” stands for Unix system calls, and “0” stands for C functions).

```
#include <fcntl.h> // open
#include <unistd.h> // read
#include <sys/types.h> // read
#include <sys/uio.h> // read
#include <stdio.h> // fopen, fread
#include <stdlib.h>
#include <sys/time.h> // gettimeofday

struct timeval start, end; // maintain starting and finishing wall time.

void startTimer( ) { // memorize the starting time
    gettimeofday( &start, NULL );
}

void stopTimer( char *str ) { // checking the finishing time and computes the elapsed time
    gettimeofday( &end, NULL );
    printf("%s's elapsed time\t= %ld\n",str, ( end.tv_sec - start.tv_sec ) * 1000000 +
(end.tv_usec - start.tv_usec));
}

int main( int argc, char *argv[] ) {

    int typeofcalls;
    // validate arguments
    // implementation

    // Parsing the arguments passed to your C program
    // Including the number of bytes to read per read( ) or fread( ), and
    // the type of i/o calls used
    // implementation

   if (typeofcalls == 1) {
        // Use unix I/O system calls to
        // implementation

    } else if (typeofcalls == 0) {
        // Use standard I/O
```

```
        // implementation
    }

    return 0;
}
```

Your program should be robust for bad arguments! If the argument is invalid or bad, your program should print a **usage statement**, and then exit graceful indicating an unsuccessful termination[4]. Further, your program should be robust to handle abnormal situations, for example, reading non-existing files.

There are several parts you must pay attention to about **usage statements** [1]

- The usage message: it always starts with the word "usage", followed by the program name and the names of the arguments. Argument names should be descriptive if possible, telling what the arguments refer to, like "filename" in the example below. Argument names should not contain spaces! Optional arguments are put between square brackets, like "-l" for the Linux command *ls*. Do not use square brackets for non-optional arguments! Always print to stderr, not to stdout, to indicate that the program has been invoked incorrectly.

- The program name: always use argv[0] to refer to the program name rather than writing it out explicitly. This means that if you rename the program (which is common) you won't have to re-write the code.

- Exiting the program: use the exit function, which is defined in the header file <stdlib.h>. Any non-zero argument to exit (e.g. exit(1)) signals an unsuccessful completion of the program (a zero argument to exit (exit(0)) indicates successful completion of the program, but you rarely need to use exit for this). Or, you can simply use EXIT_FAILURE and EXIT_SUCCESS (which are defined in <stdlib.h>) instead of 1 and 0 as arguments to exit.

For example, the following is a code snippet, which prints a usage statement, and then exits the program by indicating an unsuccessful termination

```
fprintf(stderr, "usage: %s filename bytes typeofcallsused\n", argv[0]);
exit(EXIT_FAILURE);
```

If all the command line arguments passed to your program are valid, your program should first print a message describing the current testing case and then complete all required tasks, for example,

(Assume that you are using Unix I/O system calls to read a file by 256 bytes per read)

Using Unix I/O systems calls to read a file by 256 bytes per read

(or Assume that you are using C calls to read a file by 1024 bytes per fread)

Using C functions to read a file by 1024 bytes per fread

Complete the main() function so that the program runs as specified above. Note that when the 2nd argument, (i.e., the number of bytes to read) is 1, you must use fgetc() when using C functions to read the file.

Testing your program:

For the testing purpose, you first need to create a file with a specific size, for example, using the *truncate* utility. As shown below, you can shrink or extend the size of a file named *filename* to 1M by using the *truncate* command. If the *filename* doesn't exist, it will be created

```
truncate -s 1M filename
```

Learn the syntax of the truncate utility using the man command

```
man truncate
```

```
$gcc -o unixio unixio.c
$./unixio filename 1024 1
```

batch test: create a bash shell script named *t.sh* with following content:

```
#!/bin/bash
buffersize=(1 256 512 1024 2048 2096)
## Start testing
for value in ${buffersize[*]}
do
## Testing Unix I/O system calls
tcommando="./unixio filename $value 1"
eval $tcommando
## Testing C calls
tcommando="./unixio filename $value 0"
eval $tcommando
done
## Testing is done! :-)
```

Note that you need to set up proper permissions on the above testing script before you execute the script, for example,

```
chmod +x t.sh
```

, which allows everyone to execute the script.

Then, enter
./t.sh

Assume you have successfully executed your program and your program passes your tests, please record your execution output when reading a file by 1, 256, 512, 1024, 2048, and 2096 bytes per read and fread, and save your execution output into a file named **a2output.txt** by entering
./t.sh> a2output.txt

The following shows example execution outputs when using the above testing script for reading a file by 1, 512, 1024, 2048, and 2096 bytes per read and fread.

```
Using Unix I/O systems calls to read a file by 1 bytes per read
Unix read's elapsed time            = 426349
Using C functions to read a file by 1 bytes per fread
C fread's elapsed time    = 9812
Using Unix I/O systems calls to read a file by 256 bytes per read
Unix read's elapsed time            = 2002
Using C functions to read a file by 256 bytes per fread
C fread's elapsed time    = 965
Using Unix I/O systems calls to read a file by 512 bytes per read
Unix read's elapsed time            = 1179
Using C functions to read a file by 512 bytes per fread
C fread's elapsed time    = 673
Using Unix I/O systems calls to read a file by 1024 bytes per read
Unix read's elapsed time            = 510
Using C functions to read a file by 1024 bytes per fread
C fread's elapsed time    = 595
Using Unix I/O systems calls to read a file by 2048 bytes per read
Unix read's elapsed time            = 349
Using C functions to read a file by 2048 bytes per fread
C fread's elapsed time    = 742
Using Unix I/O systems calls to read a file by 2096 bytes per read
Unix read's elapsed time            = 511
Using C functions to read a file by 2096 bytes per fread
C fread's elapsed time    = 990
```

Please write a brief (150 words or so) on observations you made about your execution outputs, and save it into a text file called systemcall.txt. Also, the systemcall.txt file should include the system information of the computer used for your test, including computer model, processor, RAM, hard drive, operating system, the version of your VirtualBox. The following is an example of such system information

Model: Dell Latitude 5490
CPU: 7th Gen Intel® Core™ i5-7300U

RAM: 8GB
Harddisk: 256GB SATA Class 20 Solid State Drive
OS: Windows 10 Pro 64bit
VirtualBox version: 5.2.2

Also, you need to test the following abnormal situations to see whether your program is robust to handle these abnormal situations, including

- Invalid number of arguments
- A file argument that does not exist
- Invalid number of bytes (or buffer size) to read per read( ) or fread( ), for example, negative number of bytes
- The type of I/O calls entered is invalid

Submission

- If you have any problems in the development of your programs, contact the teaching assistant (TA) assigned to this course.
- You are encouraged to discuss this project with fellow students. However, you are **not** allowed to share code and your brief with any student.
- If your TA is unable to run/test your program, you should present a demo arranged by your TA's request.
- Please only submit the source code files plus any files required to build your shell program as well as any files requested in the assignment, including
  - the complete source code;
  - the Makefile; and
  - your execution output (a2output.txt)
  - your brief (systemcall.txt)
- How to name your programming assignment projects: For any assignment, zip all the files required to a zip file with name: CIS3110_<assignment_number>_XXX.zip, where <assignment_number> is the assisgnement number you are solving (e.g., a2 for Programming Assignment 2) and XXX is your University of Guelph's email ID (Central Login ID). This naming convention facilitates the tasks of marking for the instructor and course TAs. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction.

  Note: to zip and unzip files in Unix:

  $zip -r filename.zip files
  $unzip filename.zip

Please only submit the .c source code files and Makefile file plus any files requested in an assignment. You are required to develop software in C on the Unix platform. DO NOT SUBMIT ECLIPSE PROJECTS.

References:
[1] Processing command-line arguments.
http://courses.cms.caltech.edu/cs11/material/c/mike/misc/cmdline_args.html
[2] CS 11: How to write usage statements.
http://courses.cms.caltech.edu/cs11/material/general/usage.html
[3] Input-output system calls in C | Create, Open, Close, Read, Write.
https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/
[4] Exit Status.
https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html