CSE 5031 Operating Systems
2019/20 Fall Term

Project:	4 – Part 1
Topic:	Multithreaded Programming
Date:	25.11 - 29.11.2019

Objectives:

- to create and manage threads
- to compare multithreaded program performance versus its single threaded version

References:

- **Lawrence Livermore National Laboratory Computing Center Pthreads tutorial portal,** https://computing.llnl.gov/tutorials/pthreads/#Pthreads
- **Linux System Programming 2d ed., Robert Love, O'Reilly 2013** (course web site, or http://pdf-ebooks-for-free.blogspot.com.tr/2015/01/oreilly-linux-system-programming.html
- **The GNU C Library Reference Manual** (course web site, or http://www.gnu.org/software/libc/manual/pdf/libc.pdf)

Section I. Project Definition

I.1 Project Topic

The project aims to implement the prototype of a **multithreaded C** application, and compare its performance versus a **single threaded** version.

The **multithreaded C** prototype will emulate the behavior of an application that contains a mix of independent **I/O intensive** and **computing intensive** code segments. **I/O bound** code is modeled with copying a large file; the **CPU bound** code by a sequence of **for loops**.

Note that, performance results measured in this project are system dependent. Measured times differ, depending on the type and number of processors (i5, i7) and the type of disk units (mechanical. SSD). Therefore the constants defined herein are calibrated with respect to the configurations of the laboratory workstations.

I.2 Implementation Constraints

To observe the performance gains obtained by **multithreaded programming**, the prototype application should be implemented and run with necessary precautions to **eliminate,** if not to **minimize,** the interferences of:

- ✓ the **optimized services** implemented by the underlying **OS**; and
- ✓ the operating platform (the hardware and OS configurations).

The performance of a **C** program running on a **Linux platform** is affected by:

- ✓ the **I/O API** used in the program; and
- ✓ the implementation of the **OS** services (e.g. buffered I/Os and deferred data writes).

The high level **GNU/glibc I/O API** that allows the writing of portable programs at source code level, introduces several abstraction layers hiding the underlying **OS** calls implementation. Therefore, the project should be implemented with low level **GNU/glibc I/O API**, to eliminate the overhead induced by portability abstractions.

The **OS** introduces several **optimizations** independent of the **I/O API** used, e.g. **buffered I/Os** and **deferred data writes**, that alters the performance measurements. Therefore the application should consider the following points in order to minimize their impacts.

a) For the **I/O bound** code, if the **size** of an **input file is small** and the **OS** uses a **very large system I/O buffers** (hundreds of Kbytes), the **OS** can **read** the **entire file** in its buffer at once, and serve program's requests from there; yielding in 1 physical data transfer from disk file to the system buffer, followed by several in memory data copies. This behavior fails obviously to model the **I/O bound** program prototype.

   To offset this **OS** optimization, the tests will use large input files, in the order **several Mbytes**.

b) The **write** operations are not only **buffered** by the **OS**, they are also **deferred. OS** keeps data in its system caches and writes it at its convenience. Therefore, the test program may perform several in memory- data transfers from user data area to system buffers, and the physical data transfer may occur only once. This behavior also fails in modelling the **I/O bound** program prototype.

To offset the above optimizations the test prototype:

- ✓ should **force** the **OS** to transfer data to the disk at each **write** operation; asking for "**synchronous**" **writes** (consult the Operating Modes in GNU C Library Reference Manual for the **O_FSYNC** option);
- ✓ generate enough **I/O** traffic to observe the impact of thread switching on their write requests.

I.3 Implementation Platforms

The project may be developed on the **Oracle Virtualbox VM** platform. **Performance measurements** should be done exclusively on the **laboratory workstations** running under **CentOS 7**:

- ✓ to eliminate the interferences of the **Windows'10** and that of the **virtualization platform** on the behavior of the Guest Host running **CentOS 7**;
- ✓ to avoid the impact of the hardware configuration constraints and the overheads that may be incurred by unknown installation choices of **CentOS 7**.

Section II. Implementing I/O bound Procedure

II.1 Designing the I/O bound Procedure

Define the **I/O bound** test procedure, as a function with **one argument** that complies with the function syntax used for **multithreading**. The argument is the pointer to the path of the file to be copied, as shown here after:

```
void *  iobound ( void *   path )
```

The function should:

- ✓ print its **process ID**, is **thread ID** and the path of the file to be copied;
- ✓ read the file defined by the argument 'path', in a buffer of size **BUFSZ**, until **end-of-file**;
- ✓ write the record to a local file opened with the "**O_FSYNC**" attribute, using the size returned by the read.

You are advised to generate the name of the local file, by extracting it from the argument 'path' using "**basename**" function (refer to the GNU Library Reference Manual page 123). For instance, if the path points to the string "/usr/bin/Xvnc" this function returns a char * to the substring "Xvnc".

II.2 Testing I/O bound Procedure

Test the correctness of the **I/O bound** procedure using the following steps.

a) Create first a local copy of a small text file (e.g. /etc/passwd), and print its contents to verify that all the records are properly copied

e.g. **iobound (/etc/passwd) ;.**

b) Run the procedure as a **thread**, controlling its termination from the main thread.

c) Run the **I/O bound** procedure using 2 threads; one to copy "/usr/bin/Xvnc", and the other "/usr/bin/Xorg" in the current working directory, and name them as the "Xvnc" and the "Xorg" respectively.

Section III. Measuring Multithreading Performance

The performance criteria of the multithreaded application is the **measure of the time** elapsed between the creation of the first thread and the termination of the last thread, expressed in seconds.

Use the function "**time**" (GNU Library Reference Manual 21.4.1) with a "**NULL**" parameter, to obtain the current calendar time of the system twice: before starting the first **thread** and after the last **thread** has ended.
In the **GNU C Library**, **time_t** is equivalent to **long int**, thus elapsed time can be obtained by a simple substation.

elapsed = start – end;

Section IV. Implementing CPU bound Procedure

IV.1 Defining the I/O bound Procedure

Use the following **C** code to model a **CPU bound** test procedure that can also be used in **multithreading**. Note that you will redefine the loop counters "**imax**" and "**jmax**", as necessary, to maximize the performance gap between **multithreaded** and **single threaded** applications.

```
#define imax 100000
#define jmax 100000
void *cpubound()
{
        int i, j;
        double x,y;
        printf("\n%d> (%u) computing\n", getpid(),  pthread_self());
        for (i=0; i< imax; i++)
                for (j=0; j< jmax; j++)
                        x=(x+5)*(y+1)*(y+2);
        return NULL;
}
```

IV.2 Testing the overall Performance Multithreaded Programming

Rerun the performance test you have defined in the previous section, by including the **CPU bound** procedure in the **thread mix**. Observe and try to explain the changes in measured **elapsed times**.

Section V. Multithreading Performance vs. Single treaded Programming

You can compare the performance of **multithreaded** code versus its **single threaded** version in the same program, as shown here after. Note that, since may also call the "**iobound**()" and the "**cpubound**()" functions **sequentially** , instead of running them in parallel with **threads**.

```
time_t  start, tend;
printf("\n----testing thread performance ----\n");
if ((start= time(NULL)) == -1)   {perror("time -1");return 1;}

        pthread_create(&s1TID, NULL, iobound, (void *) "/usr/bin/Xvnc");
        pthread_create(&s2TID, NULL, iobound, (void *) "/usr/bin/Xorg");
        pthread_create(&s3TID, NULL, cpubound, NULL);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        pthread_join(t3, NULL);

if ((tend= time(NULL)) == -1) {perror("time -1");return 1;}
printf("\nelapsed= %d\n", tend-start);
//------------------------------------------------------------------------
printf("\n----testing single threadrd performance ----\n");
 if ((start= time(NULL)) == -1) {perror("time -1");return 1;}

        iobound ("/usr/bin/Xvnc");
        iobound ("/usr/bin/Xorg");
        cpubound ( );

if ((tend= time(NULL)) == -1) {perror("time -1");return 1;}
printf("\nelapsed= %d\n", tend-start);
```

Section VI. Project IV Part 1 Report

Test your application using the following values first:

- ✓ **BUFSZ** = 1000
- ✓ **imax** = 100000
- ✓ **jmax** = 100000

Experiment with your application by changing the following:

- ➢ **BUFSZ**, reducing buffer size increases the number of I/O operations that have **no impact** on **reads,** as they are mainly buffer-to-buffer data transfers. However, the **write** time should increase since the **thread** has to block each time until data is written to the disk.
- ➢ Adding an additional **CPU bound** and/or **I/O bound thread** to your mix (subsequently to your sequential program).

Submit the **code** that produces the largest performance gain in multithreading. Store the results in a text file by redirecting "**stdout**" to the file "**test.txt**".

Add a comment line consisting of your name and student-id**;** and store the **source code** and the **test.txt** files in the "**Prj4-Part1**" folder, located at the course web site under the tab **CSE5031-X/Assignment**; where "**X**" stands for (A,B,C,D) your laboratory session group you are registered.

Warning

You are encouraged to discuss the implementation procedures and general concepts behind the projects with your fellow students. However, **plagiarism is strictly forbidden**! Submitted report should be the result of **your personal work**!

Be advised that you are **accountable** of your submission not only for this project, but also for the mid-term, and final examinations. Your project grade may be reevaluated retrospectively, had you fail to answer correctly the same or a similar examination questions that you have solved with success in your submissions.