CSE 5031 Operating Systems
2020/21 Fall Term

Project: 5 – Part 1
Topic: Multithreaded Race Condition
Date: 13 - 21.12.2020

Objectives:

- to develop a multi-thread application with **Pthreads**
- to implement **mutual exclusion** using POSIX **Semaphores**

References:

- **Lawrence Livermore National Laboratory Computing Center Pthreads tutorial portal,** https://computing.llnl.gov/tutorials/pthreads/#Pthreads
- **Linux The Textbook. S.M. Sarwar, R.M/ Koretsky. CRC Press 2019. ISBN 978-1-138-71008-5**
- **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 A. Project Definition

A.1 Problem Statement

The project aims at implementing a **multi-thread race scenario** whereby **concurrent transactions** try to update a **shared data object**, a bank **account**. You are asked to develop a **C** program that:

→ uses **POSIX Pthread API** to create **threads,** and
→ resolves race conditions with **mutual exclusion** primitives provided by **POSIX semaphores**.

Project testbed will host following transaction categories.

✓ **atm** – sketches **withdrawal** or **deposit** actions from/to a **bank account**.
✓ **display** – simulates periodical printing attempts of the bank statement to notify account's owner when the balance has changed.
✓ **salary** – emulates periodic deposits of the **pay checks** to the bank account.
✓ **tax** – emulates generic withdrawals from the bank account for bill, credit card, tax **payments**.

The testbed will run **only 1 instance** for each transaction category, although you may start several `**tax**` transactions with different amount and withdrawal timing.

A.2 Implementation Constraints

i) Shared global variables

As the testbed is limited in transaction types and their instances (one of each) you may use global variables:
✓ to define **mutex semaphores**;
✓ to pass **parameters** e.g. time, amount to threads.

ii) Simulation of processing overheads

Use "**sleep** (seconds)" function to emulate heavy processing and system call related delays. This call will release the CPU(s) that will be dispatched to other threads waiting in the ready queue.

iii) Displaying transaction outputs

Among testbed transactions only "**atm**" requires the use of the keyboard to **read** the deposit or withdrawal amounts entered at random intervals. As such, the thread running "**atm**" should use the **pseudo terminal** "**/dev/pst/0**" - *opened to run the* ***main thread***- and associated with **stdin** and **stdout**/**stderr** **streams**.

Other transactions do not require keyboard inputs; they just display their status or results. However, these transactions **should not write** their output to **stdout**, as these messages will be interleaved with those generated by the input/output streams of the "**atm**" thread!

One feasible solution is to display all these outputs on an alternative **pseudo terminal** e.g**.** "**/dev/pst/1**", thus to use a new **output stream** rather than **stdout** associated with the "**/dev/pst/0**" pseudo file.

To display the outputs of a transaction an alternative **pseudo terminal** you have to perform the following.

→ Open a **pseudo terminal** window using GNOME GUI before running your program and display the **pseudo** terminal identifier using the "**ps**" command. "**TTY**" info e.g. "**pst/1** infers that it has been created in **FSH** as the "**/dev/pst/1**" file. Verify that file name matches the one used in your program,

Example listed here after assumes that you opened only one extra **pseudo terminal,** thus associated "**pts1**" stream with the "**/dev/pst/1**" file. If you open other terminals, modify your program as well.

→ Open in the **main** thread the file "**/dev/pst/1**" in **write-only mode**, and

→ Use stream output functions "**fputs**" or "**fprintf**" referred with a **stream** variable "**pts1**" stream.

```
FILE *pst1;   // global stream variable definition
….
void *display( )
{  ………….
   fprintf (pts1, "\ndisplay> account = %d \n", account);
…….
}
int main(….)
{  ………….
   if ( ( pts1 = fopen ("/dev/pts/1", "w") ) == NULL )
        { printf ("\n error message…..");  return 0; }
…….
   pthread_create ( &displayTID, NULL, display, NULL);
…….
}
```

iv) Terminating asynchronous threads

Correct implementation of a **multithreaded** application requires the **main thread** to wait for the termination of all the **threads** that are started in order to clean up the process context.

The **main thread** can use two primitives "**pthread_cancel**" and "**pthread_kill**" to **terminate** running threads associated with the current process. Which one to use?

✓ "**pthread_kill**" sends an **interrupt signal** to the thread and the OS kernel destroys it regardless of its state. This asynchronous kill request may create **inconsistencies** on resources the thread is using. In other words, the **thread should not die** while holding resources in a way that might cause **deadlock**.

✓ "**pthread_cancel**" call is introduced later on to circumvent "**pthread_kill**" mishaps. It sets a **status flag** of the target thread notifying it to "**exit**" when it is possible. Most of system calls check threads status flags to see if they should exit or not. This is a safe way for terminating a thread since it reached a **cancelability state** (refer to the thread-safe notes in the GNU C call definitions).

Alternatively, we will implement in this project an **algorithmic solution** to terminate the threads, instead of **killing** or **cancelling** them. This is a **clean** approach that let **each thread** to control its own resources, including its **execution lifetime**. To that end, we will use:

- ✓ a **flag variable** to set the termination event; and
- ✓ **semaphores** to implement **mutual exclusion** between threads.

The next example shows how you can organize your thread control structure to implement this project.

► Note that, there are several cases in which a **thread** is blocked waiting for an I/O or an event, and cannot check its flags. **Cancelling** or **killing** them is then **the only way** to clean up process' context.

```
int  on = 1;    // global termination flag
....
void *display( )
{  .............
   while ( on >= 0 ) {
.......
   }
   fprintf (pts1, “\ndisplay> terminating \n”);
   return NULL;
}
void *atm( )
{  .............
   while ( ( on = scanf(“%d”, &amount ) ) != EOF) {
                // process amount
   }
   printf (“\natm> terminating \n”);
   return NULL;
}

int main(....)
{  .............
    pthread_create ( &displayTID, NULL, display, NULL);
.......
    pthread_join (displayTID, NULL);
.......
}
```

Section B. Implementing “atm” and “display” Transactions

B.1 Testbed Design

You will first develop the “**atm**” and “**display**” transactions to establish a reliable testbed infrastructure, and then add the rest of the transactions as the platform operate correctly.

Do not forget to open a second **pseudo terminal** before running your program and verify its **file name** with “**ps**”.

B.2 Implementation Guidelines

Organize your program in **three threads** of execution, performing the actions defined here after.

a) **main thread** - *the default thread running main( ) function-* should:

- ✓ initialize the **mutex semaphore(s)**;
- ✓ open the **stream** to access the second pseudo terminal file in write access mode;
- ✓ create the **atm** and **display** threads, and display their **TID**;
- ✓ wait for the termination of both **threads**;
- ✓ terminate.

b) **display thread** should:

- ✓ display its **TID** and starting message at the **second pseudo terminal** window;
- ✓ while the termination flag is off;
  - ▪ sleep for **1 second**;
  - ▪ procure the access to the account;
  - ▪ display the account if it has changed;
  - ▪ vacate the access to the account;
- ✓ display its **TID** and termination message;
- ✓ terminates.

c) **atm thread** should::

- ✓ display its **TID** and starting message at the default pseudo terminal window;
- ✓ display the input request for the amount to be deposit or withdrawn;
- ✓ read from **stdin** (the keyboard) while EOF is not entered
  - ▪ procure the access to the account;
  - ▪ perform a deposit or a withdrawal **if** the account balance allows it;
  - ▪ display the operation message
  - ▪ vacate the access to the account;
- ✓ set the termination flag (if not set by the read operation as implemented in the example);
- ✓ display its **TID** and termination message;
- ✓ terminate.

Section C. Adding "salary" and "tax" Transactions

Once **section B** runs correctly add first the "**salary**" transaction, then "**tax**" as specified here after. .

C.1 Implementation Guidelines

a) **salary thread** should:

- ✓ deposit the account a fixed amount every **10 seconds**;
- ✓ perform similar actions to "**display**" transaction to report and terminate.

b) **tax thread** should:

- ✓ withdraw from the account a fixed amount (e.g. 1/10 of the salary) every 2 **second,** regardless of the balance (means that the account may grow negative as in real life);
- ✓ perform similar actions to "**display**" transaction to report and terminate.

C.2 Report Preparation

Test your application several times with different timings and amounts. Once it performs as specified:

- ✓ add a comment line in each stating your name and student-id ;
- ✓ name the source code as "**Prj 5 – Part 1.c**".

Section D. Project V Part 1 Report Submission

Do not submit a result if your program does not work as specified.

Store your code file "**Prj 5 – Part 1.c**" in the "**Prj5-Part1**" folder, located at the course web site under the tab **CSE5031 - OS Section -X/Assignment**; where "**X**" stands for (1,2,3,4) your laboratory session group.

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.