I. Introduction

In this project (Project #3), we implement the high-priority and low-priority tasks using UNIX p-thread. Note that UNIX offers multiple thread-synchronization mechanisms, such as mutex, semaphore ("thread semaphore"), and condition variable, but we use semaphores ("thread semaphores", which are different type of semaphores we used for Project #2) for this project. We will implement an inter-thread message sharing system using threads and thread semaphores (referred as just “semaphores” hereafter).

Different from processes, threads can share data and messages through the global variable space (as long as they are the threads in the same process). Just like out previous project (Project #2), there are two types of threads: the high-priority and low-priority threads. The high-priority threads update the contents in the shared message, while the low-priority threads read the contents in the shared message as shown in the figure below:

II. Program specifications

(a) Functional Requirements

(1) The parent thread (the only thread in the process immediately after the process starts) creates (instantiates) and initializes semaphores (“thread semaphores” (NOT “process semaphore”) and the shared message in the global variable space.

(2) The parent thread creates five child thread (using “pthread_create” system call, which replaces “fork” in Project #2)) in the following order:

- The first child thread created by the parent thread becomes "H2" (the second high-priority thread).
- The second child thread created by the parent thread becomes “H3” (the third high-priority thread).
• The third child thread created by the parent thread becomes “L₁” (the first low-priority thread).
• The fourth child thread created by the parent thread becomes “L₂” (the second low-priority thread).
• The fifth child thread created by the parent thread becomes “L₃” (the third low-priority thread).
• After L₃ is created by the parent thread, the parent thread becomes “H₁” (the first high-priority thread).

(3) Each of the three high-priority threads repeats the following (a), (b), and (c):

(a) sleeps for specific time
(b) updates the shared message after it enters the critical section,
(c) sleeps for some time before the thread leaves the critical section for a number of times specified by NUM_REPEATS (to be declared at the top of the C source code file).

(4) Each of the high-priority thread updates the message, one at a time, while any number of the low-priority threads can read the shared message at a time (as long as no high-priority thread is updating the shared message (i.e., in the critical section)).

(5) Each of the three low-priority thread repeats as long as one of the three high-priority thread is still working. Each of the three low-priority thread waits until all three high-priority threads finishes repeating their (a), (b), and (c) as described above.

(6) After each of the three high-priority threads finishes (has repeated NUM_REPEATS times), the first high-priority thread (the parent thread) proceeds to (7).

(7) The parent thread waits until all the five child threads complete.

(9) The parent thread deleted all the “thread semaphore(s)” before it terminates.

(10) The parent thread terminates.

The following figure shows the required timing (progress) of the six threads.
(b) **Output Requirements:**

**The parent thread:**
1. Print “the parent thread starts …” when the process starts.
2. Print “a semaphore is created …” each time the parent thread creates a semaphore.

**High-priority thread:**
1. Print “Hx starts …” when a high-priority thread starts (when H2 or H3 is created and when the parent thread becomes H1). Each high-priority thread prints this message.
2. Print ”Hx would like to update …” immediately after a high-priority thread stops sleeping (“wakes up”) outside of the critical section.
3. Print ”Hx starts updating …” immediately after a high-priority thread has entered the critical section.
4. Print ”Hx finishes updating …” immediately before a high-priority thread leaves the critical section.

**Low-priority thread:**
1. Print “Lx starts …” when a low-priority thread starts (when L1, L2, or L3 is created).
2. Print ”Lx would like to read …” immediately after a low-priority thread stops sleeping (“wakes up”) outside of the critical section.
3. Print ”Lx starts reading …” immediately after a low-priority thread has entered the critical section.
(4) Print "Lx finishes reading …” immediately before a low-priority thread leaves the critical section

(c) Technical Requirements:

(1) Each of the six tasks (H1, H2, H3, L1, L2, and L3) should be properly implemented by a thread.
(2) No violation of the “mutual exclusion” by any high-priority threads
(3) Concurrent reads by low-priority threads (when no high-priority thread is in the critical section)
(4) No starvation to any thread (especially to the high-priority threads)
(5) No thread deadlocks
(6) The parent thread should wait (“pthread_join”) for all of its five child threads to terminate, before it terminates.
(7) No “spin wait (“busy loops”)” can be used.
(8) While threads (both the high-priority and the low-priority threads) are executing their loop structure, outputs from the “printf” in the following figure are required.
(9) Submitted C source code file should be compiled by gcc compiler on os.cs.siue.edu server without any additional libraries (other than “lpthread”) or without using “makefile”.
(10) Using the random number generator to generate random sleep time (“millisleep(unsigned microsecond)”) is required.
(11) “Clean termination” for any thread (no “crash termination”).

Required Thread Structures

- Your high-priority and low-priority threads should have the following structure:

```c
High-priority Thread
for (i = 0; i < NUM_REPEATS; i++)
{
    printf("Hx starts ..\n");
    RANDOM SLEEP
    printf("Hx would like to update ..\n");
    CRITICAL SECTION
    RANDOM SLEEP
    printf("Hx finishes updating ..\n");
}
printf("Hx terminates ..\n");
```

```c
Low-Priority Thread
while (all_high_priority_thread_active)
{
    RANDOM SLEEP
    printf("Lx would like to read ..\n");
    CRITICAL SECTION
    RANDOM SLEEP
    printf("Lx finishes reading ..\n");
}
printf("Lx terminates ..\n");
```

[this section continues to the next page]
• The followings are the required labels (posted to the CS314 course home – please copy and paste them to the top of your source code file):

// the following labels are required to appear at the top of your source code file
// the TA will change their values for testing your program design
#define NUM_REPEATS 100 // number of loops for the high-priority threads
#define H1_TIME_IN 300000 // 300 ms = 0.300 seconds (inside of the critical section)
#define H1_TIME_OUT 2000000 // 2000 ms = 2.000 seconds (outside of the critical section)
#define H2_TIME_IN 300000 // 300 ms = 0.300 seconds (inside of the critical section)
#define H2_TIME_OUT 2000000 // 2000 ms = 2.000 seconds (outside of the critical section)
#define H3_TIME_IN 300000 // 300 ms = 0.300 seconds (inside of the critical section)
#define H3_TIME_OUT 2000000 // 2000 ms = 2.000 seconds (outside of the critical section)
#define L1_TIME_IN 1000000 // 100 ms = 1.000 seconds (inside of the critical section)
#define L1_TIME_OUT 200000 // 200 ms = 0.200 seconds (outside of the critical section)
#define L2_TIME_IN 1000000 // 100 ms = 1.000 seconds (inside of the critical section)
#define L2_TIME_OUT 200000 // 200 ms = 0.200 seconds (outside of the critical section)
#define L3_TIME_IN 1000000 // 100 ms = 1.000 seconds (inside of the critical section)
#define L3_TIME_OUT 200000 // 200 ms = 0.200 seconds (outside of the critical section)

The required delay module (to be copied to your *.C source code)

/* function "millisleep" -------------------------------- * /
void millisleep(unsigned micro_seconds)
{
    unsigned sleep_time; // sleep time

    /* randomly generate the sleep time ------- */
    sleep_time = rand() % micro_seconds;

    /* start sleeping ------------------------ */
    usleep(sleep_time);
}

III. Testing Procedure

Your submitted *.C source code will be tested in the following ways:

1. Testing by re-compiling and executing your submitted *.C source code file for the functional, output, and technical requirements.
2. The timing parameters shown above will be changed after we receive your submission.
(3) The thread implementation in your *.C source code file will be inspected by tracing your submitted *.C source code file. It is strongly suggested that each of you attach decent in-line comments to avoid penalties by misunderstanding by the grader’s side.

III. Objectives

This project is designed for the following objectives

(1) To understand the concept of threads (UNIX pthread)
(2) To understand the concept of inter thread communication (i.e., global variables)
(3) To understand the concept of thread synchronization
(4) To develop system programming skills to manage race condition and critical section
(5) To develop programming skills to avoid thread starvation and deadlock
(6) To have UNIX system programming experience
(7) To be familiar with UNIX-based operating systems

IV. Grading Criteria

1. Any compile-time error (-80%)
2. Minor run-time error: -5% (for each)
3. Major run-time error (failing to meet any of requirement (a) through (d): depends.
4. Program style (program structure and in-line comments): 10%
V. Guidelines for acceptable activities

1. Exchanging ideas with your classmates: Not allowed
2. Exchanging source codes with your classmates: Not acceptable (absolutely not allowed, no matter how many lines of the codes).
3. Requesting someone else (but yourself) to write any code for you for this assignment: Not acceptable (absolutely not allowed). Exams may ask you some questions regarding your code. If you fail to convince Dr. Fujinoki your work, your assignment credit can be cancelled.

VI. Required submission

- Program softcopy (*.C source code file) should be submitted to Moodle as “project3_you3digits.c” (one *.c file).
- When you are submitting your C source code file to Moodle, please make sure to download it after your submission and test your submitted C source code file by compiling and running it by yourself (make sure the correct submission).

VII. Late Submission

- Penalty of -10% will be given for every 24 hours after the due (i.e., -10% for a submission within the first 24 hours after the due).
- Submission more than 48 hours after the due will not be accepted.

VIII. Early Submissions (for extra credit)

The deadlines for the early submissions (“extra-credit” and “free-feedback”):

11:59:59 p.m. on November 23rd (+4/100 points go to your final exam)

IX. Grading Criteria

- Compile-time errors: If your submitted *.c files do not compile using gcc compiler at os.cs.siue.edu due to any compile-time error: 0 ~ 20% of the credit for this project will be given.

- Run-time error: After your source code files are successfully compiled, if your program does not satisfy any requirement: 5% penalty for each minor problem. For any major problem, penalty depends on each such major problem.
X. Expectations when a question is asked

(a) Please do not attach your *.C source code file when you are asking a question unless Dr. Fujinoki requests it.

(b) Dr. Fujinoki will NOT debug your *.C source code file (Dr. Fujinoki will not debug any general C programming issues – that’s the topics of CS150 and CS240), but not in the scope of CS314.

(c) Identify where (in your source code) the problem exists

(d) Describe the symptom(s) of the problem

(e) Describe how the problem happens (always happen, sometime happen, the condition(s) for the problem to happen, etc.)

(f) Describe what you tried (to understand and/or solve the problem)

(g) No help will be provided on the day before the due (except continuing help that started before the day before the due).

(h) No help will be provided the day before the final submission due (except continuing discussions before the day before the submission due).

XI. Suggestions/Hints

(a) UNIX pthread:

Suggested references will be posted to the course home

XII. Academic Dishonesty

The following conducts (but not limited to them) are considered academic dishonesty, which will result in one of the following sanctions:

(a) For minor cases (e.g., explain a solution/design by non-written format, such as verbally to others): receiving zero credit for this particular project (project #2).

(b) For moderate cases (e.g. showing a solution/design as a memo or sketch for multiple programming issues, or downloading sample programs from the Internet or any other sources after they are modified): receiving zero credit for the entire project grade (not only for project #2).

(c) For major cases (e.g. e-mail source code file or copy/exchange source code files with others, or downloading a whole program source code file from the Internet or any other sources and resubmit it with minor modifications): receiving an F grade for this course.

The bottom line: you are expected to demonstrate of your learning regarding how to handle similar programming projects using only technical references that explain the syntax of the software tools or/and only basic samples that describe the syntax. If you are not sure, contact Dr. Fujinoki, instead of making your own assumptions.
XIII. Sample Screen Snapshots

(a) Immediately after the parent process is started

```
[hujino@vm-10:~/CS314_F20/project_03]$ ./a.out
Thread H2 is created ...
Thread H3 is created ...
Thread L1 is created ...
Thread L2 is created ...
Thread L3 is created ...
the parent thread is wakening up H2 ...
the parent thread is wakening up H3 ...
the parent thread is wakening up L1 ...
the parent thread is wakening up L2 ...
the parent thread is wakening up L3 ...
H1 is the last thread to start ...
H1 would like to update ...
thread H1 starts updating ...
    L1 would like to read ...
    L3 would like to read ...
H3 would like to update ...
thread H1 finishes updating ...
    thread L1 starts reading ...
    thread L3 starts reading ...
        thread L1 reads: I am H1!
        thread L3 reads: I am H1!
H1 would like to update ...
    thread L3 finishes reading ...
H2 would like to update ...
    thread L1 finishes reading ...
thread H3 starts updating ...

thread H3 finishes updating ...

thread H1 starts updating ...
    L2 would like to read ...
thread H1 finishes updating ...
thread H2 starts updating ...

H1 would like to update ...
H3 would like to update ...
thread H2 finishes updating ...

    thread L2 starts reading ...
        thread L2 reads: I am H2!
    thread L2 finishes reading ...
thread H1 starts updating ...
    L1 would like to read ...
H2 would like to update ...
    L3 would like to read ...
```
(b) While all six threads are active (demonstration of concurrent reader accesses)
(c) Immediately before the parent process is terminated