1. Introduction

In this programming project, we will implement an inter-process message sharing system (the one we implemented for Project #2) using threads (instead of processes). In the system, there are two types of threads: the high-priority and low-priority threads. Since threads shared data through global variable(s), the high-priority threads update the content in a global variable (as a character string), while the low-priority threads read the content in the global character string as shown in the figure below:

2. Project Specifications

2.1 Functional Requirements

(1) The main (parent) thread creates and initializes the contents of the global character string (as “no one updates the message yet”) and “pthread mutex semaphores”.

(2) The main (parent) thread creates five child threads in the following order:

- The first child created by the main thread becomes “L1” (the first low-priority thread).
- The second child created by the main thread becomes “L2” (the second low-priority thread).
- The third child created by the main thread becomes “L3” (the third low-priority thread).
- The fourth child created by the main thread becomes “H1” (the first high-priority thread).
- The fifth child created by the main thread becomes “H2” (the second high-priority thread).
- The sixth child created by the main thread becomes “H3” (the third high-priority thread).
- After H3 is created by the main thread, the main thread wait for all the six child threads to terminate without using a spin-wait.

(3) The three low-level threads wait until all the high-priority threads are ready (H1, H2 and H3 are created by the main thread).

(4) Each of the three high-priority threads repeats the following (a), (b), and (c): (a) sleeps for some time, (b) updates the shared message in the global character string after it enters the critical section, and (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).
(5) Each of the high-priority threads updates the message, one at a time, while many 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).

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

(7) The parent thread waits until all the six threads terminate.

2.2 Output Requirements:

The parent thread:
(1) Print “the parent thread starts …” when the parent thread starts
(2) Print “a semaphore is created and initialized …” each time the parent creates a semaphore

High-priority threads:
(1) Print “Hx starts …” when a high-priority thread starts (when H1, H2 or H3 is created). Each high-priority thread prints this message.
(2) Print ”Hx enters the critical section …” immediately after a high-priority thread has entered the critical section
(3) Print ”Hx is leaving the critical section …” immediately before a high-priority thread leaves the critical section

Low-priority threads:
(1) Print “Lx starts …” when a low-priority thread starts (when L1, L2 or L3 is created).
(2) Print ”Lx enters the critical section …” immediately after a low-priority thread has entered the critical section
(3) Print ”Lx is leaving the critical section …” immediately before a low-priority thread leaves the critical section

2.3 Technical Requirements

(1) No violation of the “mutual exclusion” by any high-priority threads
(2) Concurrent reads by low-priority threads
(3) No starvation to any thread (especially to the high-priority threads)
(4) No thread deadlock
(5) No “spin-wait” for preventing thread starvation (just like Project #2)
(6) “Clean termination” for any thread (no “crash termination”)
(7) Submitted C source code file should be compiled by “cc” compiler with pthread library (“-lpthread”) on os.cs.siue.edu server without any additional libraries or without using “makefile”.
(8) While threads (both the high-priority and the low-priority threads) are executing their loop structure, no outputs should be made except those “printf” in the required thread structures provided in Section 2.4 below. “millisleep” for random sleep for each thread should not be altered either.
2.4 Thread Structural Requirements

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

**high-priority thread**

```c
for (i = 0; i < NUM_REPEATS; i++) {
    RANDOM SLEEP
    printf("Hx starts updating ..\n");
    \begin{critical}
        RANDOM SLEEP
    \end{critical}
    printf("Hx finishes updating ..\n");
}
```

**low-priority thread**

```c
while (all_high_priority_threads_complete) {
    RANDOM SLEEP
    printf("Lx starts reading ..\n");
    \begin{critical}
        RANDOM SLEEP
    \end{critical}
    printf("Lx finishes reading ..\n");
}
```

The “base source code file” for Project #2 should be used as the basis for the high-priority and low-priority threads. Any outputs should be made by the “printf” already in the source code files. The structure of the two thread types (i.e., the high-priority and low-priority threads) should not be altered.

The entire system structure is shown in the figure on the next page.
create mutex sem.
initialize mutex sem.
create a thread
create a thread
create a thread
create a thread
create a thread
create a thread
create a thread

H3 thread active
H2 thread active
H1 thread active
L3 thread active
L2 thread active
L1 thread active

H3 terminates
H2 terminates
H1 terminates
L3 terminates
L2 terminates
L1 terminates

wait for all six terminate
terminate
• 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 // 1000 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 // 1000 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 // 1000 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 “pthread mutex semaphores” do NOT need a key.

3. Objectives

This programming is designed for the following objectives

(1) To understand the concept of multi-thread programming using UNIX pthread (user-mode threads)
(2) To understand the concept of inter thread communication
(3) To understand the concept of thread synchronization, using “pthread mutex semaphore”
(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

4. Grading Criteria

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

6. Required submission

Program softcopy should be submitted to Moodle, as “project3_your3digits.c” (one *.c file).

Specify your section number (“001”) in the program header.

7. 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.

8. Early Submissions

The deadlines for the early submissions (“extra-credit” and “free-feedback”) will be announced in the course home.