CS314-001 Operating Systems

Programming Project #2 Description, Spring 2023

Project Due: 9:30 A.M. on April 3rd

1. Introduction

In this programming project, we will develop “a multi-process BBS” using UNIX standard semaphores (but not POSIX semaphores) and shared memory. The multi-process BBS (Bulletin Board System) is a software system for controlling (regulating) the ways the shared message is accessed (read and write). The BBS has a memory buffer, where a message is posted. There are two types of processes who access the posted message in the BBS. The first group is readers, who just read the posted message, but they never modify or write any message. The other is writers, who post a new message to the BBS (i.e., overwrite the existing message by a new one).

2. Program specifications

Your final program should satisfy the following requirements:

(a) The shared message should be implemented as a character string (up to 255 characters) in a shared memory.

```c
char posted_msg[256];
// (possible) another item
// (possible) another item
// .......

shared memory
```

(b) The content of the posted message should be initialized as “no one has posted a message” at the beginning of your program (before child processes are created).

(c) The first process (which is called “the parent process”) becomes the first reader (R1) and should create other five processes (R2, R3, R4, W1, and W2) using `fork` system call. Processes that are created by the parent process are called “child processes”.

(d) Each new process should wait until all five child processes are created by the parent process before they start doing their job (i.e., accessing the posted message).

(e) Four readers (R1, R2, R3, and R4) and two writers (W1 and W2) should be concurrently running (multi-tasked).
(f) Up to three (not four) readers (any three in R1, R2, R3, and R4) can read the posted message at the same time, while only one writer can post a new message at a time. When a writer is posting (updating) the posted message, no other process can access the posted message.

(g) When a process (a reader or a writer) tries to access the posted message (i.e., when a process tries to enter the critical section), each process should display: “R# (or W#) likes to read (or post) the message” (see Figure 1 and 2 for an example).

(h) Right after each process enters the critical section (for accessing the posted message), each process outputs (displays) the following message:
- “R# starts reading the message: <the contents of the message>”, if it is a reader and # = 1, 2, 3, or 4.
- “W# starts posting a new message: <the contents of the new message>”, if it is a writer and # = 1 or 2.

(i) Right before a reader leaves its critical section, each reader outputs (displays) the following message:
- “R# finishes reading the message”

(j) Right before a writer leaves its critical section, the writer outputs (displays) the following message:
- “W# finishes posting a new message”

(k) Writers continue to repeat until each of them finishes repeating NUM_REPEAT times. Each writer process terminates after that.

(l) The four readers stop reading the posted message after both writers terminate. After that, the three child readers (R2, R3, and R4) terminate themselves.

(m) The parent process should wait until all the five child processes to terminate. After that, the parent process delete semaphore(s) and the shared memory before it terminates itself.

(n) To control the activity timing of the readers and the writers, the following five labels should be declared at the top of your source code (the TA will change their values for testing your program). Figure 1 shows the required structure of the critical section for each reader. Figure 2 shows the required structure of the critical section for each writer.
- NUM_REPEAT: the number of times each writer updates a new message.
- READER_TIME_01: the time each reader sleeps before it reads the posted message (before a reader starts trying to enter the critical section).
- READER_TIME_02: the time each reader sleeps after the reader finishes reading and after displaying the posted message (but before a reader leaves the critical section).
- WRITER_TIME_01: the time each writer sleeps right before it updates a new message (before a writer starts trying to enter the critical section).
- WRITER_TIME_02: the time each writer sleeps after it finishes updating the posted message (but before a writer leaves the critical section).

Note 1: “the required structure” means: (i) that part must be executed for each iteration of the loops (no skipping, no bypassing, or no altering those statements is allowed), (ii) you add your own codes on top of those C statements.

Note 2: The required structure for readers and writers are shown in the next page.
To prevent starvation against writers, no more than two readers can enter the critical section after a writer starts trying to enter its critical section (any number of readers can read the posted message as many times as they like as long as no writer tries to post a message).

The compiler we should use is “gcc” (instead of “cc” compiler).

```c
int sleep_time;

while ( .... )
{
    sleep_time = rand(READER_TIME_01);
    msleep(sleep_time);
    printf(“Rn likes to read the posted message\n ”);

    // the critical section starts here -------------------
    printf(“Rn starts reading the message: ”);
    printf(“%s\n”, <the pointer to the shared memory>);    
    sleep_time = READER_TIME_02; // no “rand” – fixed time
    msleep(sleep_time);
    printf(“Rn finishes reading the posted message ..\n”);
    // the critical section ends here -------------------
}
```

**Figure 1** - the required structure for each reader

```c
int sleep_time;
char my_message[MSG_SIZE];
strcpy(my_message, “Hello, I am Wx ....”);

for (i = 0; i < NUM_REPEAT; i++)
{
    sleep_time = rand(WRITER_TIME_01);
    msleep(sleep_time);
    printf(“Wn likes to post its message\n”);

    // the critical section starts here -------------------
    printf(“Wn starts posting a new message;”);
    printf(“%s\n”, my_message);
    strcpy(<the pointer to the shared memory>, my_message);
    sleep_time = WRITER_TIME_02; // no “rand” – fixed time
    msleep(sleep_time);
    printf(“Wn finishes posting a new message ..\n”);
    // the critical section ends here -------------------
}
```

**Figure 2** - the required structure for each writer
3. Requirements

Your program should satisfy requirements (a) through (p), especially the following requirements:

(1) No starvation should occur.
(2) No violation of mutual exclusion should occur for any process.
(3) No deadlock should occur (all the processes should be always completed)
(4) Multiple readers should be able to read the posted message (at least several times in a program execution).
(5) No spin wait to wait for a condition to be satisfied should be used.
(6) The required program structures (as presented in Figure 1 and 2), as well as the five labels for “sleep time” should be used (as required). The function of “msleep” will be posted to the CS314 course home.

4. Objectives

This programming is designed for the following objectives

(1) To understand the concept of process
(2) To understand the concept of inter process communication (IPC)
(3) To understand the concept of process synchronization
(4) To develop system programming skills to manage race condition and critical section
(5) To develop programming skills to avoid process starvation and deadlock
(6) To have UNIX system programming experience
(7) To be familiar with UNIX-based operating systems

5. Grading Criteria

1. Any compile-time error (-80%)
2. Minor run-time error: -5% (for each)
3. A program that satisfies (2) no violation for mutual exclusion (no race condition), (3) no process deadlock, (4) concurrent readers, and (5) no spin wait: 85% of the credit
4. Design and implement “starvation-free” for the two writers (1): 15% of the credit
5. Major run-time error (failing to meet any of the requirement (a) through (p)): depends.
6. Program style (program structure and in-line comments) – including the required “program header” shown in APPENDIX: 10%
6. Guidelines for acceptable activities

1. Using external references (web sites, programming reference books, and etc.): **Allowed** (but only for the semaphore system calls. The source of the information (references) must be provided to Fujinoki).

2. Exchanging ideas with your classmates: **Not acceptable**

3. Exchanging source codes with your classmates: **Not acceptable** (absolutely not allowed, no matter how many lines of the codes).

4. 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.

5. Sample source code files posted to the course (CS314, Spring 2023) home by the course instructor (Fujinoki): **Allowed (recommended)**.

8. Required submission

   Program softcopy (should be e-mailed to the instructor) by the project due (9:30 AM on April 3rd).

9. Late Submission

   - Penalty of -10% will be given for every 12 hours after the due (i.e., -10% for a submission within the first 12 hours after the due).
   - Submission more than 48 hours after the due will not be accepted.

10. Early submissions

   - Extra-Credit: TBA
   - Free Feedback: TBA
APPENDIX:

The five constant labels for defining time interval should be placed at the beginning of your source code file as shown below:

```c
/* ********************************************************** *
* CS314-001 Project #2 solution                          *
*                                                             *
* Your last-three:                                          *
* Your course section #:                                    *
*                                                             *
* Spring 2023                                              *
*                                                             *
* ********************************************************** */
#define NUM_REPEAT      50   // each boiler-man repeats
#define READER_TIME_01  300   // 300ms = 0.3 seconds
#define READER_TIME_02  800   // 800ms = 0.8 seconds
#define WRITER_TIME_01  1200  // 1200ms = 1.2 seconds
#define WRITE_TIME_02   1600  // 1600ms = 1.6 seconds
#include <........>
```