CS314-001 Operating Systems

Programming Project #2 Description, Spring 2020

Project Due: 9:30 A.M. on April 4th

1. Introduction

In this programming project, we will develop a solution for “boiler men and bathers problem” using UNIX standard semaphores (but not POSIX semaphores) and shared memory. The details of the problem and the project and its requirements will be presented using a PPT presentation.

2. Program requirements

Your final program must satisfy the following requirements:
(a) Only one C/CPP source code file should be submitted.
(b) Three bathers and two boiler men should be concurrently running (multi-tasked).
(c) The first process becomes the first boiler man (B1) and should create other four processes (one is another boiler man, and three are bathers) using fork system call.
(d) Newly created processes must wait for all processes to be created and start running.
(e) When each process enters the critical section, each process outputs the following message:
   • “B# starts his boiler …”, if it is a boiler-man and # = 1 or 2 (boiler-man number)
   • “T# is entering the bathing area”, if it is a bather and # = 1, 2, or 3 (bather number)
(f) When a bather is leaving the bathing area, each bather outputs the following message:
   • “T# is leaving the bathing area”
(g) When a boiler-man is leaving the bathing area, the boiler-man outputs the following message:
   • “B# is leaving the bathing area”
(h) Each process waits for a random amount of time each time in the bathing area or outside of the area.
(i) Boiler men continue to repeat until each of them finishes repeating NUM_REPEAT times. The three bathers terminated after both boiler men repeat NUM_REPEAT times.
(j) The main process should delete semaphore(s) and the shared memory before it terminates itself
(k) Although this project is introduced using two boiler-men and three bathers, submitted work should not be designed based on any assume for any specific number of bathers and boiler-men.
To control the activity timing of the bathers and the boiler-men, the following five labels should be declared at the top of your source code and be used in the critical section as specified below. Figure 1 shows the required structure of the critical section for each bather. Figure 2 shows the required structure of the critical section for each boiler-man.

- **NUM_REPEAT**: the number of times each boiler-man does his work (heat up the water)
- **BATHER_TIME_0x_A**: time interval a bather sleeps out of the pool (for bather x)
- **BATHER_TIME_0x_B**: time interval a bather stays in the pool (for bather x)
- **BOLIERMAN_TIME_0y_A**: time interval a boiler-man does his work (for boiler-man y)
- **BOLIERMAN_TIME_0y_B**: time a needs to do heat up the water (for boiler-many)

```
int sleep_time;

while ( ..... )
{
    sleep_time = rand(BATHER_TIME_01_A);
    usleep(sleep_time);

    [ your semaphore(s) here ]

    // the critical section starts here ---------------
    printf(“T%d is entering the bathing area ..\n”, my_TID);
    sleep_time = BATHER_TIME_01_B;
    usleep (sleep_time);
    printf(“T% is leaving the bathing area ..\n”, my_TID);

    [ your semaphore(s) here ]

    // the critical section ends here ---------------
}
```

**Figure 1** - the required structure of the critical section for bather #1
For bather #2 and #3, their labels should be:

- Bather #2: “BATHER_TIME_02_A” and “BATHER_TIME_02_B”
- Bather #3: “BATHER_TIME_03_A” and “BATHER_TIME_03_B”

```c
int sleep_time;

for (i = 0; i < NUM_REPEAT; i++)
{
    sleep_time = rand (BOLIERMAN_TIME_01_A);
    usleep (sleep_time);

    [ your semaphore(s) here ]

    // the critical section starts here ---------------------
    printf (“B%d starts his boiler .....\n”, my_BID);

    sleep_time = BOLIERMAN_TIME_01_B;
    usleep (sleep_time);

    printf (“B%d is leaving the bathing area ....\n”, my_BID);

    [ your semaphore(s) here ]

    // the critical section ends here ---------------------
}
```

**Figure 2** - the required structure of the critical section for boiler-man #1

**For boiler-man #2:**
- “BOLIERMAN_TIME_02_A” and “BOLIERMAN_TIME_02_B”

### 3. Requirements

Your program must follow the following requirements:

(a) No starvation should occur.
(b) No violation of mutual exclusion should occur for boiler men.
(c) No deadlock should occur (all the processes should be always completed).
(d) Multiple bathers should be able to enter the bathing area (at least several times in a program execution).
(e) No busy loop (a.k.a., “spin loop”) should be used.

```c
while (some_condition != TRUE)
{
    ;
}
```

**Figure 3** – an example of “busy loop”
4. Objectives

This project is designed for the following objectives:

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

5. Grading Criteria

- Any compile-time error (-80%)
- Minor run-time error: -5% (for each)
- A program that satisfies (a) no race condition, (b) no violation for mutual condition, and (c) concurrent bathers: 85% of the credit
- Design and implement “starvation-free” for the two boiler-men: 15% of the credit
- Major run-time error (failing the meet any of requirement (a) through (d): depends.
- Program style (program structure and in-line comments): 10%

6. Guidelines for acceptable activities

- Using external references (web sites, programming reference books, and etc.): Allowed.
- Exchanging ideas with your classmates: Not recommended but acceptable
- Exchanging source codes with your classmates: Not acceptable (absolutely not allowed, no matter how many lines of the codes).
- 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).
- Sample source code files posted to the course (CS314, Spring 2020) home by the course instructor (Fujinoki): Allowed (recommended).

8. Required submission

Program softcopy, as “P2_your_3_digits.cpp” (should be e-mailed to the instructor) by the project due (9:30AM on April 4th).

9. 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.
10. Early Submissions

- Extra-credit early submission: 11:59:59 PM, March 26th
- Free-feedback early submissions: 11:59:59 March 31st

11. Suggested progresses

**PHASE-1:** Test the sample code for UNIX POSIX semaphores and shared memory (1 hour)
- Download it (“with_semaphore.cpp”) from the CS314 home
- Compile it and run its binary executable
- Make sure you successfully/correctly create a semaphore and shared-memory by “ipcs” command
- Identify the essential system calls for semaphore and shared-memory controls by studying “with_semaphore.cpp”.

**PHASE-2:** Test the sample code for fork system call (1 hour)
- Download it (“fork_test.cpp”) from the CS314 home
- Compile and run its binary executable
- Understand the basic program structure for “fork” system call

**PHASE-3:** Create four processes: B2, T1, T2, and T3 (1 to 4 hours)
- Let each of them sleep as soon as each of them is created
- Apply “ps –a” to make sure you create exactly four processes
- Let each of the four processes to terminate

**PHASE-4:** Using a semaphore, synchronize the five processes (1 to 2 days)
- While the first process is creating a process at time, let each of the four processes wait until the first process finishes creates all four processes
- Let all five processes wait for some time (e.g., 3 seconds)
- Let each of the four processes terminate (their order of termination does not matter)
- Let the first process waits for all the four processes terminate, then let the first process terminate.

**PHASE-5:** Implement the boiler-man and bathers’ logics to what you have correctly implemented for (d) (1 to 3 days) using semaphores and shared memory
APPENDIX-1:

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

```c
/* ********************************************************** *
 * CS314 Project #2 solution *
 * Your last-three: *
 * Your course section #: *
 * Spring 2020 *
 * You can "copy & paste" this to your source code file *
 * ********************************************************** */
#define NUM_REPEAT 50   // each boiler-man repeats
#define BATHER_TIME_01_A 300   // 300ms = 0.3 seconds
#define BATHER_TIME_01_B 800   // 800ms = 0.8 seconds
#define BATHER_TIME_02_A 300   // 300ms = 0.3 seconds
#define BATHER_TIME_02_B 800   // 800ms = 0.8 seconds
#define BATHER_TIME_03_A 300   // 300ms = 0.3 seconds
#define BATHER_TIME_03_B 800   // 800ms = 0.8 seconds
#define BOLIERMAN_TIME_01_A 1200   // 1200ms = 1.2 seconds
#define BOLIERMAN_TIME_01_B 1600   // 1600ms = 1.6 seconds
#define BOLIERMAN_TIME_02_A 1200   // 1200ms = 1.2 seconds
#define BOLIERMAN_TIME_02_B 1600   // 1600ms = 1.6 seconds
#include <.........>
```
APPENDIX-2: the Overall Process Structure

start
create shared-memory
create semaphores
create a child process
create a child process
create a child process
create a child process
create a child process

act as B1
act as B2
act as T1
act as T2
act as T3

block until all children created
block until all children created
block until all children created
block until all children created

block until B1 and B2 completed
block until B1 and B2 completed
block until B1 and B2 completed

terminate itself
terminate itself
terminate itself
terminate itself

block until all four child complete
delete shared-memory and semaphores

start
create shared-memory
create semaphores
create a child process
create a child process
create a child process
create a child process
create a child process

act as B1
act as B2
act as T1
act as T2
act as T3

block until all children created
block until all children created
block until all children created
block until all children created

block until B1 and B2 completed
block until B1 and B2 completed
block until B1 and B2 completed

terminate itself
terminate itself
terminate itself
terminate itself

block until all four child complete
delete shared-memory and semaphores
APPENDIX-3: the expected internal structure of your P2.cpp (the one everyone is supposed to submit)

Each of you is expected to submit only one *.cpp (“P2.cpp”) file, which has the following structure:

```
main
    This one should also as B1
    Implementation of B2
    Implementation of T1
    Implementation of T2
    Implementation of T3

P2_your_3_digits.cpp
```
APPENDIX-4: The essential system calls for Project #2 (CS314-002, Spring 2020)

(a) Fork system call

- Structure
- Variables (which variables are inherited and which variables will not be inherited between the parent and the child process)
- Risk in calling “fork” in a loop

(b) Shared memory

1. Define the contents of a shared memory (as C “struct”)
2. Obtain the size of the shared memory
3. Create the shared memory (“shmget”)
4. Attach the shared memory (“shmat”) – “shmat” returns a pointer
5. Initialize the contents in the shared memory (using the pointer)

Using the shared memory

6. Detach the shared memory (“shmdt”)
7. Delete the shared memory (“shmctl”)

(c) Semaphores

1. Prepare “semaphore operation array” (“struct sembuf operations[1]”; – for manipulating a semaphore)
2. Prepare “semaphore control data structure” (“union semun” – for initializing a semaphore)
3. Set the initial value of a semaphore (argument.val = 1;)
4. Create a semaphore (“semget”)
5. Initialize the new semaphore (“semctl”)

6. Use the semaphore by using “semaphore operation array” (“semop”)
   - for “wait” operation
   - for “signal” operation

7. Delete a semaphore (“semctl”)

CS314 Operating System, Programming Project #2