1. Introduction

In this individual programming project, we will develop a solution for “synchronized swimming pool control 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 (Project_01_Required_Algorithm_Structure.ppt).

The “synchronized swimming pool control problem” consists of the following processes: three bather processes (A_1, A_2, and A_3), two boiler-man processes (B_1 and B_2), and the safeguard process (S). Each bather process represents a bather, who repeats the following activities: enjoys bathing by staying the pool-side for some time and once in a while enters the swimming pool. Each boiler-man process becomes active once in a while to start heating the water in the swimming pool. The safeguard periodically inspects the condition of the swimming pool after kicking out any bather and boiler-man from the pool (as show in the figure below).

![Diagram of the synchronized swimming pool control problem](image.png)

**Figure 1** – visualization of “synchronized swimming pool control problem”
2. Program requirements

Your final program should satisfy the following requirements (please use the following list for your “check list”):

(a) Only one C source code file should be submitted for implementing “synchronized swimming pool control problem”.
(b) Each of the three bathers, two boiler men, and the safeguard should be implemented as a process and concurrently running (multi-tasked).
(c) The first process becomes the safeguard (S) and should create other five processes (B1, B2, A1, A2, and A3 in that order) using fork system call.
(d) Newly created child 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:
   - “S starts inspection”, when the safeguard process enters the critical section
   - “B# starts his water heater …”, when a boiler-man process (# = 1 or 2 (boiler-man number)) enters the critical section
   - “A# is entering the swimming pool”, when a bather process (# = 1, 2, or 3 (bather number)) enters the critical section
(f) When a bather is leaving the swimming pool, each bather outputs the following message:
   - “A# is leaving the swimming pool”
(g) When a boiler-man finishes water-heating, the boiler-man outputs the following message:
   - “B# finishes water heating”
(h) When the safeguard process finishes his/her inspection, the safeguard process outputs the following message:
   - “S finishes inspection”
(i) Each process waits for a random amount of time each time in the swimming pool and outside of the pool.
(j) Boiler men continue to repeat until each of them finishes repeating NUM_REPEAT times. The three bathers terminate after both boiler men finish.
(k) The safeguard process should repeat until all the other five processes finish. The safeguard process should delete the semaphore(s) and the shared memory after all A1, A2, A3, B1, and B2 finish.
(l) To control the activity timing of the bathers, the boiler-men, and the safeguard processes (the timing parameters will be changed when each submission is tested) the following five labels should be declared at the top of your source code and be used in the critical section as specified below (including the following labels in your C source code is required).
• NUM_REPEAT: the number of times each boiler-man does his work (heat up the water)
• BATHER_TIME_01_A: average time interval a bather A1 stays outside of the pool
• BATHER_TIME_02_A: average time interval a bather A2 stays outside of the pool
• BATHER_TIME_03_A: average time interval a bather A3 stays outside of the pool
• BATHER_TIME_01_B: average time interval a bather A1 stays in the pool
• BATHER_TIME_02_B: average time interval a bather A2 stays in the pool
• BATHER_TIME_03_B: average time interval a bather A3 stays in the pool
• BOLIERMAN_TIME_01_A: average time interval a boiler-man B1 waits before next water heat-up
• BOLIERMAN_TIME_02_A: average time interval a boiler-man B2 waits before next water heat-up
• BOLIERMAN_TIME_01_B: average time B1 needs to do heat up the water
• BOLIERMAN_TIME_02_B: average time B2 needs to do heat up the water
• SAFEGURAD_TIME_A: average time interval the safe guard waits before next inspection
• SAFEGURAD_TIME_B: average time the safe guard needs to finish his/her inspection

Note: The above timing parameters as “labels” (should be in “micro-second” order) are posted to the CS314 course home (“required_test_parameters.c”).
(m) Your C source code is required to use the following “required structures” (Figure 2, 3 and 4):

Figure 2 shows the required structure of the critical section for each bather.

```c
long int sleep_time;

while ( .... )
{
    sleep_time = rand (BATHER_TIME_0x_A);
    usleep(sleep_time);

    [ your semaphore(s) here ]

    // the critical section starts here ---------------------
    printf ("A%d is entering the swimming pool \n", my_TID);
    sleep_time = BATHER_TIME_0x_B;
    usleep (sleep_time);
    printf ("A% is leaving the swimming \n", my_TID);

    [ your semaphore(s) here ]

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

Figure 2 - the required structure of the critical section in the bather processes

**Note**: “required” in “the required structure” means:

- The loop structure (the type of the loop structures, such as `while` and `for`) should not be changed.
- The `printf` statements in the structure should not deleted or (logically) skipped
Figure 3 shows the required structure of the critical section for each boiler man.

```c
long int sleep_time;

for (i = 0; i < NUM_REPEAT; i++)
{
    sleep_time = rand(BOLIERMAN_TIME_0y_A);
    usleep(sleep_time);

    [ your semaphore(s) here ]

    // the critical section starts here --------------------
    printf("B%d starts his water heater ....\n", my_BID);

    sleep_time = BOLIERMAN_TIME_0y_B;
    usleep(sleep_time);

    printf("B%d finishes water heating .\n", my_BID);

    [ your semaphore(s) here ]

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

Figure 3 - the required structure of the critical section for boiler-man processes

Figure 4 shows the required structure of the critical section for the safe guard process

```c
while ( ....)
{
    sleep_time = rand(SAFE_GURAD_TIME_A);
    usleep(sleep_time);

    [ your semaphore(s) here ]

    // the critical section starts here ---------------------
    printf("S starts inspection .\n");

    sleep_time = SAFE_GURAD_TIME_B;
    usleep(sleep_time);
    printf("S finishes inspection .\n");

    [ your semaphore(s) here ]

    // the critical section ends here ---------------------
}
```
**Figure 4** - the required structure of the critical section for the safe guard process

### 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 and the safeguard.
(c) No deadlock should occur (all the processes should be always completed)
(d) Multiple bathers should be able to enter the swimming pool (at least several times in a program execution).
(e) All outputs from your program should be made only by the “printf” specified by Figure 2, 3 and 4.
(f) No busy loop (a.k.a., no “spin wait”) should be used.

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

*Figure 5* – an example of “spin wait”

### 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%

- **For confirming the authorship for submitted source code files, an interview may be conducted.**
6. Guidelines for acceptable activities

- This programming project is an individual programming project.
- Exchanging ideas with your classmates: **Not allowed**
- Exchanging source codes with your classmates: **Not allowed**
- Requesting someone else (but yourself) to write any code for you for this assignment: **Not allowed**. Exams will ask you some questions regarding your code. If you fail to convince Dr. Fujinoki your work, your assignment credit can be cancelled).
- Using the sample source code files posted to the course (CS314, Summer 2023) home by the course instructor (Fujinoki): **Allowed (recommended/required)**.

7. Required submission

Program softcopy, as “P1_your_3_digits.c” (should be submitted by emails) by the project due (11:00 a.m. on June 10th).

8. Late Submission

- Penalty of -10% will be given for every 12 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.

9. Early Submissions

- Extra-credit early submission: TBA
- Free-feedback early submissions: TBA

10. 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 five processes: A₁, A₂, A₃, B₁, and B₂ (**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 five processes
- Let each of the five 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 five processes wait until the first process finishes creates all five processes
- Let all five processes wait for some time (e.g., 3 seconds)
- Let each of the five processes terminate (their order of termination does not matter)
- Let the first process waits for all the five 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

11. Guideline for “seeking helps”

(a) Dr. Fujinoki will not debug your project source code file. Please do not send (attach) your *.c source code file unless you are asked to do so.

(b) Designing your program logic is a part of this project.

(c) When you ask questions/advises, please describe the following details:
   ① The symptom of the problem(s) (what is wrong and when it happens)
   ② Your analysis of the problem(s) (how it is happening)
   ③ What you tried so far

(d) Within 48 hours before the final submission due, no new question will be answered (except for continuing discussions).
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 #1 solution */
/* Your last-three: */
/* Your course section #: */
/* Summer 2023 */
/* You can "copy & paste" this to your source code file */
/* ******************************************************* */
#define NUM_REPEAT 50   // each boiler-man repeats
#define BATHER_TIME_01_A 300000  // 300ms = 0.3 seconds
#define BATHER_TIME_01_B 800000  // 800ms = 0.8 seconds
#define BATHER_TIME_02_A 300000  // 300ms = 0.3 seconds
#define BATHER_TIME_02_B 800000  // 800ms = 0.8 seconds
#define BATHER_TIME_03_A 300000  // 300ms = 0.3 seconds
#define BATHER_TIME_03_B 800000  // 800ms = 0.8 seconds
#define BOILERMAN_TIME_01_A 1200000  // 1200ms = 1.2 seconds
#define BOILERMAN_TIME_01_B 1600000  // 1600ms = 1.6 seconds
#define BOILERMAN_TIME_02_A 1200000  // 1200ms = 1.2 seconds
#define BOILERMAN_TIME_02_B 1600000  // 1600ms = 1.6 seconds
#define SAFEGUARD_TIME_A 1200000  // 1200ms = 1.2 seconds
#define SAFEGUARD_TIME_B 1600000  // 1600ms = 1.6 seconds
#include <.........>
```
APPENDIX-2: the Overall Process Structure

- 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 S

- repeats until all five child complete
- delete shared-memory and semaphores

block until all children created
act as A3
repeats until B1 and B2 completed
terminate itself

block until all children created
act as A2
repeats until B1 and B2 completed
terminate itself

block until all children created
act as A1
repeats until B1 and B2 completed
terminate itself

block until all children created
act as B2
terminate itself

block until all children created
act as B1
terminate itself
APPENDIX-3: the expected internal structure of your P1.c (the one everyone is supposed to submit)

Each of you is expected to submit only one *.c ("P1.c") file, which has the following structure:

```
main

Implementation of B1
Implementation of B2
Implementation of A1
Implementation of A2
Implementation of A3
P1_your_3_digits.c
Implementation of B1
Implementation of S
```

Note: the order of the six processes in the source code can be different
APPENDIX-4: The essential system calls for Project #1 (CS314, Summer 2023)

(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 #1