CS 307 OPERATING SYSTEMS PROGRAMMING ASSIGNMENT – 3 REPORT

Prepared by Pelinsu Sarac, 28820

Sabanci University

2023

1. Introduction

This report includes the implementation details of the solution for CS307 Programming Assignment 3. It includes synchronization primitives used, which conditions program complies and with which restrictions, and detailed explanation of threads used.

2. Synchronization Primitives

In this program, 4 synchronization primitives are used that we have seen in the lectures: mutex, conditional variable, semaphore, and barrier. Except the semaphore, rest of the primitives are provided by pthread library. Semaphores used in the program are implemented in the way that we have seen in the lecture slides. This is due to a personal reason, I believe Dijkstra semaphores are more intuitive to me. Only extra thing done for semaphores is the initialization function, which is as follows:

```
void sem_init (sem_t*, int initVal) {
        pthread_mutex_init(&s->lock, NULL);
        pthread_cond_init(&s->c, NULL);
        s->val = initVal;
}
```

Further details about how they were used will be given in next sections.

3. Conditions That Program Complies

The program written for the PA3 complies with the entering and grouping conditions. There is one extra restriction, which is that there can only be one active demo session at a time. Despite this, because “end of demo” line is printed a bit later than all participations, more than 3 participation lines may be seen between two “end of demo” lines.

4. Thread Functions in the Program

Before starting, here are the synchronization primitives used in the thread functions:

- For entering condition:
    - A token counter (enter_token)
    - A mutex (enter_mutex)
    - A condition variable (enter_queue)
- For grouping condition:
    - A binary semaphore (demo_mutex)
    - Two counters for participants waiting for demo (demo_waiting_assistant and demo_waiting_student)
    - Two semaphores representing the waiting students and assistants (demo_assistant_queue and demo_student_queue)
    - A barrier to be sure that all 3 participants participated (demo_barrier)

a. Main Thread

Here is the pseudocode for main thread:

Declare program aim

assistantNum = command line argument 1

studentNum = commandline argument 2

*if assistantNum < 0 or 2*assistantNum != studentNum*

print "main terminates"

return

//Initializations for demo

```
demo_mutex.init(1)

demo_assistant_queue.init(0)

demo_student_queue.init(0)

demo_barrier.init(3)

//Initializations for entering

enter_mutex.init()

enter_queue.init()

totalNum = assistantNum + studentNum

threadList[totalNum]

declare integer i

//Create the threads

for i=0 up to assistantNum

        create(threadList[i], assistantFunc, null)

for i up to totalNum

        create(threadList[i], studentFunc, null)

//Wait for the threads

for j=0 up to totalNum

        join(threadList[j])

destroy demo_barrier

print "main terminates"
```

return

In the main function, first arguments are checked. Checks done are whether assistant number is positive and student number is twice as the assistant number. If these checks are okay, then main thread proceeds with the thread creations. It creates assistantNum many assistant threads and studentNum many student threads. Finally, it waits for all of them to terminate before it terminates itself.

Initializations of synchronization primitives are also done in main.

b. Student Thread

Here is the pseudocode for student thread:

enter_mutex.lock

enter_token -= 1

print "Thread ID: < tid >, Role:Student, I want to enter the classroom."

If enter_token < 0

enter_queue.wait with enter_mutex

print "Thread ID: < tid >, Role:Student, I entered the classroom."

enter_mutex.unlock

Before continuing, let me explain this part. This block of code is for entering condition. Variable enter_token related to the expression "$3*enter_a - enter_s$", as this expression needs to be non-negative according to the inequality given for entering condition, or it can be thought as tokens needed to enter to classroom for a student. Mutex is to protect the atomicity of token manipulations. When a student tries to enter, token decrements by 1, as it can be seen that $enter_s$ has the coefficient -1 in the expression, or one can say that student demands a

token. Then, she checks whether token value is negative, if so she has to wait as the token shouldn't be negative normally as it technically means there is no tokens left but if so, it means there are students waiting. Else, or if the student thread is woken up by some assistant later on, student enters the class officially by printing her entrance line.

It is also good to note that, entering code block both in student and assistant threads are inspired by semaphore functions.

demo_mutex.wait

demo_waiting_student += 1

if demo_waiting_student >= 2 and demo_waiting_assistant >= 1

demo_student_queue.post

demo_student_queue.post

demo_waiting_student -= 2

demo_assistant_queue.post

demo_waiting_assistant -= 1

else

demo_mutex.post

demo_student_queue.wait

participate

demo_barrier.wait

student leaves the classroom

This part of the code is adapted from the H2O problem solution given in the Little Book of Semaphores (Downey, 2016, p. 143).

What it does is that, student first increments the waiting student for demo count as she had officially entered the class before. Then she checks whether there are enough participants for a demo: one more student and an assistant. If so, it decreases the waiting participant counts and wakes them up. Reason that student wakes up presumably two students is that she will be waiting for the semaphore after the if-else. It is like, throwing a ball over the net and run to the other side to catch it. After that, technically all woken up participants will participate (a similar section is also present in assistant code) and wait at the barrier for others to finish. One can wonder why student does not release the mutex. This will be clearer after the assistant code.

If there are not enough participants, student releases the mutex and waits. She will participate after being woken up. Finally, student leaves the class after demo.

c. Assistant Thread

Here is the pseudocode for assistant thread:

enter_mutex.lock

enter_token += 3

print "Thread ID: < tid >, Role:Assistant, I entered the classroom."

enter_queue.post

enter_queue.post

enter_queue.post

enter_mutex.unlock

This is the entering part of the assistant code. What it does is increment the token by 3 and wake up any sleeping/waiting student threads. Reason for the 3 is again the expression. Here, it means that there happens to be a capacity/token increase enough for 3 students, as each student occupy a single capacity/token by decrementing token count by 1. Thus, it also means that more students can enter now. Assistant hence wakes up at most 3 students to let them in.

To sum up the token idea, each assistant provides 3 tokens while entering and each student uses one token while entering. If there are already tokens, students can directly get into class as there is still capacity. If not, token count reduces to a negative number. Intuitively, it may look wrong that the token can be negative but while token is negative, no one is actually entering the class and token count represents the waiters outside. Until a new assistant come, they will be waiting and inequality will be preserved.

```
demo_mutex.wait
demo_waiting_assistant += 1
if demo_waiting_students >= 2
        demo_student_queue.post
        demo_student_queue.post
        demo_waiting_student -= 2
        demo_assistant_queue.post
        demo_waiting_assistant -= 1
else
        demo_mutex.post
```

demo_assistant_queue.wait

participate

demo_barrier.wait

demo_mutex.post

This part of the code is adapted from the H2O problem solution given in the Little Book of Semaphores (Downey, 2016, p. 143).

What does the assistant do here is pretty similar to what student does. First it checks whether there are enough participants for a demo. If so, she wakes them up and everyone participates, then wait for each other at barrier. Here comes the reason why student thread doesn't release the mutex but the assistant does.

There are two cases for a demo session to start: a student comes as last participant or an assistant. For the waiters to be woken up, they don't have the mutex at the time of waiting. One that has the demo mutex is always the latecomer, whether she is a student or an assistant, and the mutex remains in the "group" until the end of demo session. At the end of demo, we know that each participant has reached the barrier, and there is still a mutex to be released. Regardless of who got the mutex, we give the role to release the mutex to the assistant. Reason for that is, we know that there is only one assistant in a demo group, thus we can guarantee that the mutex was released only once, like it was acquired only once (Downey, 2016).

An analogy for it would be like this: a student or an assistant gets in to a separate cabinet in the classroom for private demo. Student/assistant enters the cabinet, closes the door and checks whether there are enough participants. They see that there aren't, so they open the door again for others to come in. Last one to enter the cabinet, a student or an assistant, will

be closing the door for demo to begin. When demo ends, assistant opens the door for everyone to go out, so that no two students try to “open the door twice”. Here, door corresponds to the demo mutex.

To sum up, each participant checks participant counts when they enter. If there are enough participants, demo starts at that point with an acquired mutex. This mutex and barriers at the end of participations allows a single demo at a time with one assistant and two students only and forces them to finish their participations before demo is over.

References

Downey, A. (2016). *The little book of semaphores*. Green Tea Press.