Problem Statement

Suppose that a certain university somewhere wants to show off how politically correct it is by applying the U.S. Supreme Court's ''Separate but equal is inherently unequal'' doctrine to gender as well as race, ending its long-standing practice of gender-segregated bathrooms on campus. However, as a concession to tradition, it decrees that when a woman is in a bathroom, other women may enter, but no men, and vice versa. A sign with a sliding marker on the door of each bathroom indicates which of three possible states it is currently in:

• Empty

• Women present

• Men present

Using Pthreads, begin by creating n male students and m female students to run as separate threads. Write the following functions: woman wants to enter, man wants to enter, woman leaves, man leaves.

Abstract

This report discusses two solutions for the unisex bathroom problem using two synchronization methods: Mutex Locks with conditional variables and Binary and Counting Semaphores. The logic behind every solution as well as some of the results obtained by some tests will be illustrated below.

Both solutions acquire the same data structure for simulating the bathroom, which is a struct containing all the needed data for the solution. They both have the same skeleton; the difference is the synchronization methods used.

1. Mutex Locks with Conditional Variables

Description:

The locks simulate when a person gets in the bathroom and locks the door behind, the only difference her is that there can be more than one person inside the bathroom at the same time as long as the bathroom has capacity and they are of the same gender. Persons here represents threads and the bathroom represents critical section. A person that finds the bathroom unavailable can wait or come later (someone will let them use the bathroom if available).

Data Structure:

```
typedef struct restroom {
    int turn;
    int men_inside;             // number of men intside the bathroom
    int women_inside;           // number of women intside the bathroom
    int men_waiting;            // number of men waiting outside the bathroom
    int women_waiting;          // number of women waiting outside the bathroom
    pthread_mutex_t lock;        // mutex to provide synchronization
    pthread_cond_t occupied[2]; //condiion var to know when the bathroom if full
}bathroom_t;
```

Functions:

In this solution, seven main functions are implemented (both solutions acquire the same functions, but with different implementations):

- *Man_wants_to_enter()*
- *Man_leavs()*
- *Woman_wants_to_enter()*
- *Woman_leavs()*
- *Men()*
- *Women()*
- *Executioner()*

Man_wants_to_enter():

This function is responsible for updating the monitoring men entrance to the bathroom. It basically locks the mutex of the bathroom struct, as it is the critical section in this case, and start checking whether a man can enter the bathroom or not. To know this, the function checks if the bathroom has women, or there is no empty slot inside for the man to enter, if not, then the man can enter the bathroom. This was the very first solution for this problem, yet it had several problems.

First, it had the famous drawback of mutex locks, which is busy waiting, which hugs the CPU core doing nothing but waiting. To avoid this problem, conditional variables were used. Every gender had a specific conditional variable that marks whether the bathroom is available for use or not. If the previously mentioned condition to enter the bathroom is not met, the ***pthread_cond_wait()*** function is called to temporarily release the lock until it becomes ready and gets back to the original thread.

The other problem with the initial solution was it did not prevent starvation, so a flag was added to the bathroom struct called turn, which was checked to determines which gender's turn is it and if it is the women's turn in this case, men are not allowed to enter and ***pthread_cond_wait()*** is called.

Now, if the bathroom is ready for men, the function decrement 1 from the men waiting queue and increment the number of men inside. At the end, the function unlocks the mutex. The implementation of the loop to check availability is as follows:

```
//while the bathroom is full or there are women inside, wait.
    while(unisex_bathroom->men_inside == CAPACITY || unisex_bathroom->women_inside > 0
            || (unisex_bathroom->turn == women && unisex_bathroom->women_waiting > 0 )){
                //wait for the bathroom to be empty
                pthread_cond_wait(&unisex_bathroom->occupied[men], &unisex_bathroom->lock);
            }
```

The same implementation for the **woman_wants_to_enter()** function.

Man_leavs():

This function is responsible for getting men out of the bathroom after finishing. It also ensures that the program has no deadlocks. The function first locks the mutex to edit the critical section of the code (bathroom data), then it decrements the number of men inside if any. It now checks if the bathroom is empty. If the bathroom is empty, and if there were women waiting outside, the function changes the turn variable to women and ***pthread_cod_broadcast(women)*** is called to indicate that the bathroom is now available for women. If there was no women outside, the turn variable has the value FREE and ***pthread_cod_broadcast()*** is called twice, once for men and once for women to make the bathroom free to use by either, which alongside the condition in the while loop of the ***wants_to_enter*** function prevents deadlocks as the threads are not blocked waiting for another to release the lock. The function then unlocks the mutex. The same logic is implemented for ***woman_leavs().***

Men():

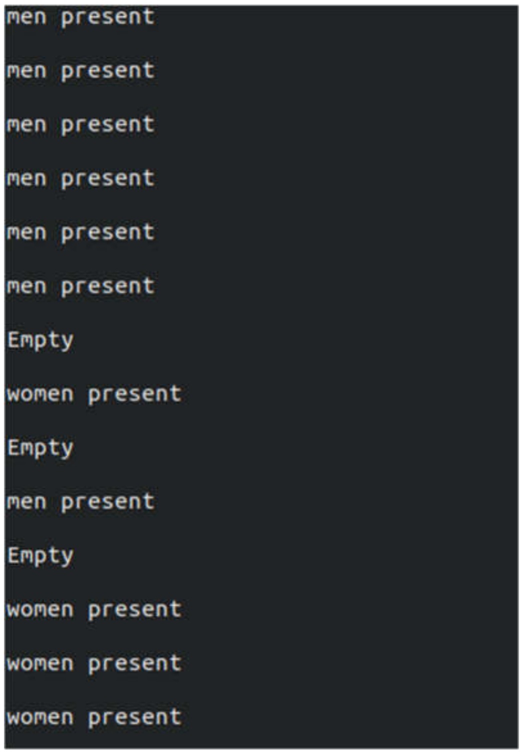
This, and the ***Women()*** function, are the runner of the threads to be pointed to by the pthread_create() function. The implementation of this function is as follows:

```
//men thread function
static void*
men (void *restroom)
{
    bathroom_t *unisex_bathroom = (bathroom_t *) restroom;
    for (int i = 0; i < LOOP ; i++){

        man_wants_to_enter(unisex_bathroom);
        printf("\n\nMen present");
        man_leaves(unisex_bathroom);
    }
    pthread_exit(0);
}
```

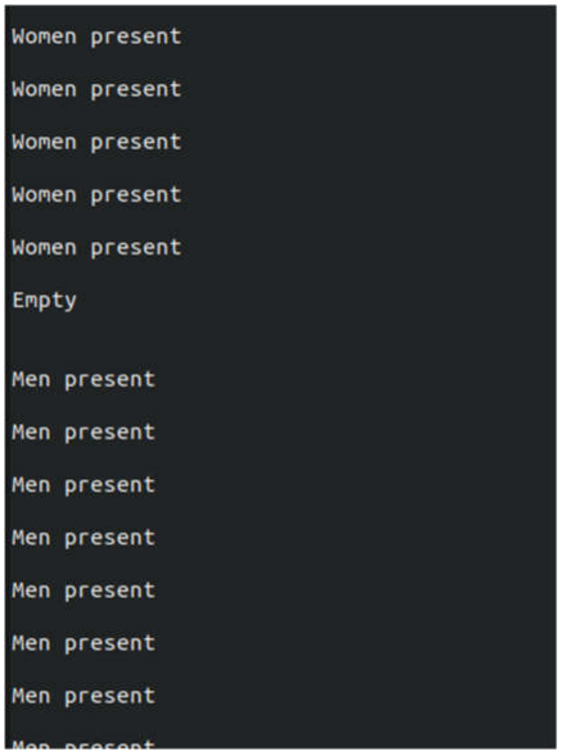
Executioner():

Here I create threads of the number of men added to the number of women and join them to get the results.

Result of Mutex and conditional variable solution:



```
men present

men present

men present

men present

men present

men present

Empty

women present

Empty

men present

Empty

women present

women present

women present
```

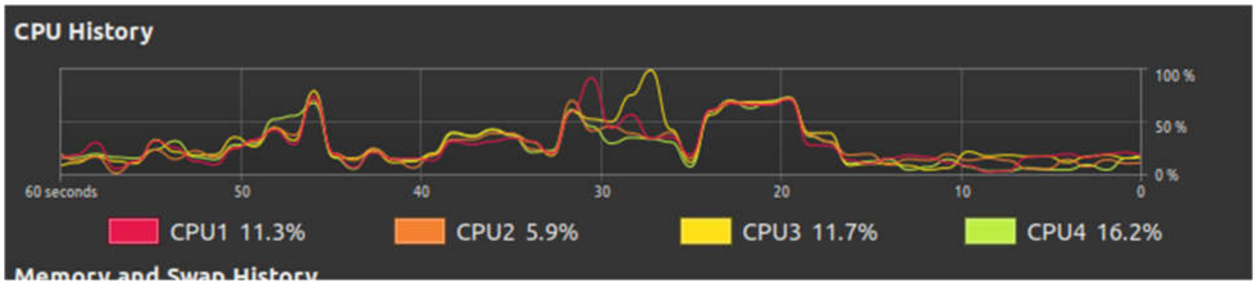


```
Women present

Women present

Women present

Women present

Women present

Empty

Men present

Men present

Men present

Men present

Men present

Men present

Men present
```

These are two runs for the code for different number of threads (men and women). The one on the right is with much more numbers of threads.

CPU Behavior:

The image below shows the effect of running the code on the CPU; all the 4 cores are used in the same time because the bathroom capacity was equal to 4. The effect is shown at the last rise of the graph.



2. Mutex Locks with Conditional Variables

Data Structure:

```
//Create bathroom struct
typedef struct restroom {
    int men_count;      //count of men
    int women_count;    //count of women
    sem_t men;          //men mutex
    sem_t women;        //women mutex
    sem_t n_men;        //men multiplex var (n.men allowed)
    sem_t n_women;      //women multiplex var(n.women allowed)
    sem_t avialable;    //if bathroom is avilable or not
    sem_t turnstile;    //to prevent starvation
}bathroom_t;
```

Description:

In this solution, I used abstraction and modularity, which are the male and female mutex (lightswitches) and the turnstile, which helped me avoid starvation. Semaphores are just generalized mutex.

This solution has the same functions as the Mutex solution, but with different implementation for the following:

- *Man_wants_to_enter()*
- *Woman_wants_to_enter()*
- *Man_leaves()*
- *Woman_leaves()*

Woman_wants_to_enter():

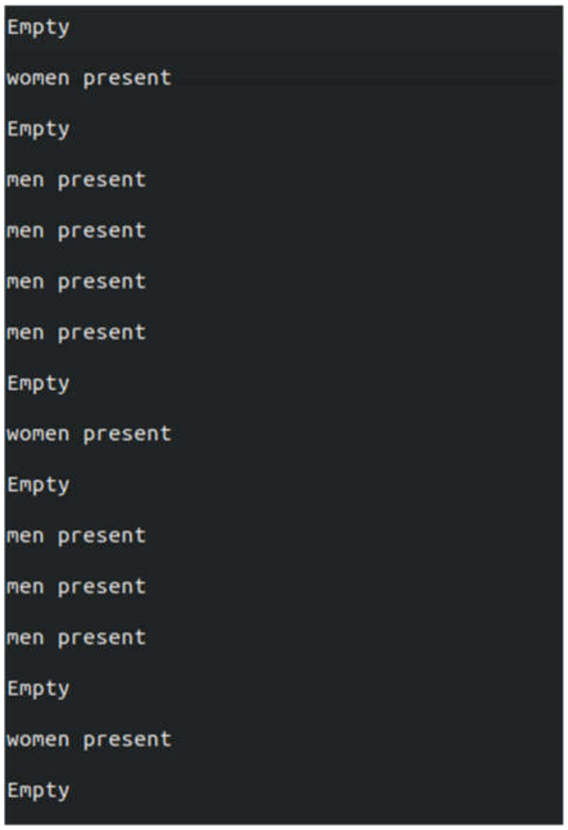
To avoid starvation, a mutex called *turnstile* was used to prevent starvation by locking the women mutex. In other words, if the bathroom is now has women inside, and a man comes to the waiting queue, then a woman comes to the waiting queue after that man; if a woman leaves the bathroom, the *turnstile* mutex prevents the woman outside from knowing that and lets the man in, and hence prevents starvation. So, the function first locks the *turnstile* mutex, then locks the women mutex and increments women count. Then it marks the bathroom unavailable, signals the women and turnstile mutex, and decrement the available slots. The exact same logic is implemented in **men_want_to_enter().**

Woman_leaves():

This function signals the counting mutex of (women multiplex) that was locked in the ***wants_to_enter*** function to mark an available slot inside the bathroom. Then the function locks the women mutex to edit in the critical section. It decrements the women count and checks if the bathroom is empty, if yes it marks the bathroom available, then signals the women mutex. And the same logic for ***man_leaves().*** The rest of functions are the same as the Mutex solution.

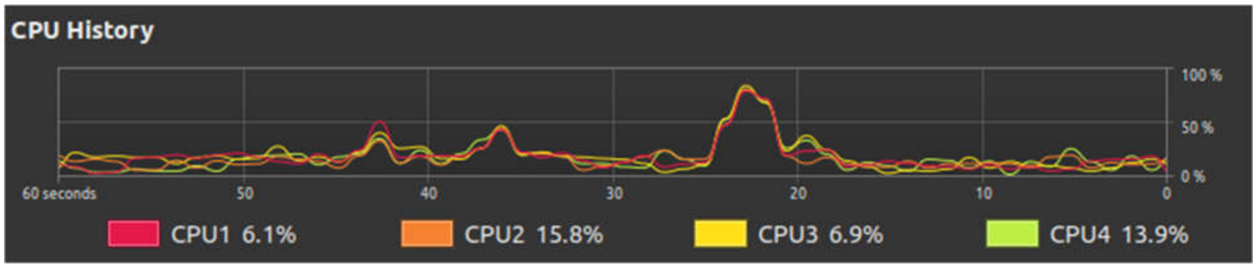
Results of the Semaphore solution:

Below is a snapshot from running the solution:



```
Empty

women present

Empty

men present

men present

men present

men present

Empty

women present

Empty

men present

men present

men present

Empty

women present

Empty
```

CPU behavior:



This graph shows the CPU behavior when running the program with 2000 threads (The highest rise). As with the mutex solution, the four cores are used at the same time because the capacity was 4. It may also be noticed that it is sharper than of the Mutex's solution, yes, it is, and this will be discussed later.

Proof of deadlock free solutions

Because in this scenario, the only resource is the bathroom, and if a thread is using it (holding the resource), it will not be waiting for another resource at the same time to release the current one, which negates Hold and Wait condition of deadlock, then one of the four main reasons for deadlock is proven to not hold, and hence these four conditions has to happed simultaneously for a deadlock to occur, then the solutions are deadlock free. A non-formal proof is that both solutions were tested for 10000 threads looping 100 times, and the execution was successful.

Performance of the solutions

Experiment:

A test has been done by running both programs with different number of threads and record the time of execution respectively to each number of threads. The recorded values are average values of running the programs for 5 times/thread value, to account for any scheduler effect, each thread looping 100 times. In the table below are the recorded results:

Mutex Locks		**Semaphores**	
Number of threads	**Time(s)**	**Number of threads**	**Time(s)**
10	0.026	10	0.04
20	0.05	20	0.097
30	0.081	30	0.118
40	0.083	40	0.129
50	0.053	50	0.144
100	0.169	100	0.173
200	0.223	200	0.308
300	313	300	0.434
1000	1.091	1000	1.191
2000	2.677	2000	2.349
5000	14.98	5000	6.296
10000	56.527	10000	12.897

Observation:

The results show that mutex are slightly faster with small number of threads, but with the number getting bigger we can see that semaphores are much faster relatively. These results make sense as mutex by nature are more expensive because of the protection protocols associated with it.

Conclusion

Mutex is a locking mechanism to implement mutual exclusion solution to the problem to synchronize the access to the bathroom. It makes sure that only one thread can acquire the mutex and enter its critical section, yet in this problem, it was required that more than one thread can use the same resource at the same time. To provide a solution for this problem using mutex, conditional variables was used. It is another method of synchronization, which was used to overcome the busy wait problem with mutex. If the bathroom is not ready to be used by the current person, it temporarily releases the CPU for other processing until the bathroom is ready to be used. On the other hand, Semaphores are signaling mechanisms that allow signaling by other threads. There are two types of semaphores, binary and counting, and they both were used in the solution. Counting semaphores were used to represent the capacity of the bathroom, and binary semaphores were used to indicate locks. Binary semaphores and mutex can be used Mutex, but not the opposite, yet they are not the same and it is advised to be treated differently because of the difference between locking and signaling mechanisms. While mutex locks can only be unlocked by the accruing thread, semaphores can be signaled by other threads.

References

[1] G. Castagna, “Concurrency ,” Index of /~gc/slides. [Online]. Available: https://www.irif.fr/~gc/slides/concurrency.pdf. [Accessed: 09-May-2020].

[2] “Introduction of Deadlock in Operating System,” GeeksforGeeks, 30-Sep-2019. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/. [Accessed: 09-May-2020].

[3] “Mutex vs Semaphore,” GeeksforGeeks, 10-Jul-2018. [Online]. Available: https://www.geeksforgeeks.org/mutex-vs-semaphore/. [Accessed: 09-May-2020].

[4] Silberschatz, Operating system concepts, 10th ed. Hoboken, NJ: John Wiley & Sons, Inc, 2019.

[5] “Welcome to The Open Group Publications Server,” The Open Group Publications Catalog. [Online]. Available: https://pubs.opengroup.org/. [Accessed: 09-May-2020].