Omer Moussa
900171920
Operating Systems Project 2
Report

Purpose:

This report is to illustrate and clarify the solution written in C and the steps of solving the synchronization problem of the same gender bathroom problem given in the project description.

Problem Description:

A unisex bathroom with a finite capacity. No different genders are allowed to coexist in the bathroom. The requirement is to simulate men and women as threads and design a solution to the synchronization problem that is fair for both genders and guarantees that only one gender is allowed to use the entire bathroom at a time.

Solution Details and Pseudo Code:

Like any synchronization problem and inspired by the solution to both reader-writer problem and dining philosophers problem studied in lectures, the the solution has to include semaphore and mutex.

The problem can be thought of as something like a writer-writer problem, where two processes want to write to a common buffer, but they are not allowed to write at the same time. However, here we can have many writers from the same type (many men or women). Therefore, there has to be some sort of synchronization within each gender as well.

So, we should have a mutex to ensure a secure a critical section for either genders; then, we should have a condition on a thread or a semaphore to wait on to block the current running thread if it wants to enter the bathroom but there is a different gender inside or if the bathroom is full.
After that, the condition on the a thread or the semaphore is released and signals everyone that it's is released; this way, if the room is empty now, any gender can come in and if there is a place empty and the same gender is inside, the thread (the person) can come in as well. Otherwise, the person (the thread should not be allowed in) as this contradicts the requirements.

The pseudo code can be realized in one of two ways: either to make one **semaphore for each gender , one for the bathroom** (like the buffer), and one for checking if it's full. Whatever gender acquires the buffer should lock the access to it until it finishes using the bathroom. We also can add counters semaphores, but they are not necessary and can be replaced by keeping a counter for the number of women/men inside. The counter increases when the person enters and decreases when he gets out.

So, this first approach (not the one I implemented) will look something like that for both genders (per thread):

```
Initialize semaphore women, men, bath, full;
Int  counterofmen, counterofwomen;

wait(women or men); // to consider the following section as a critical section.
print("a (man or woman) Enter " );
counterofmen/counterofwomen +=1; //increment the counter
If countermen/women ==capacity
        wait(full);
while counterofmen/counterofwomen equal 1
        wait(bath)// the first one to enter locks the other gender from using it.
signal(full)
signal(women/men);// leave the critical section for another thread (person)
        print("using the bathroom");
delay(some time);

//getting out:
wait(men/women);
counterofmen/counterofwomen -=1; //dec the counter
print("a (man or woman)  Leaves")
while (countermen/women is zero (or any empty case ))
        signal(bath);
signal(men/women);
```

This algorithm needs 4-6 semaphores to run as explained above. A better solution would be to use the **pthread mutex** and the pthread condition with no need to semaphores. This approach needs only 2 synchronization objects which reduces the overhead than using like 4-6 semaphores, makes the program run faster ,and they are native to the threads.

The **pthread condition** is similar to the semaphore in the wait concept, but when there is a wait in a pthread condition variable, it unlocks the mutex until another thread singlas it, then it locks it again and terminates the condition. This **allows us to need only one mutex** instead of 2 for men and women. Also, we can join the two conditions of exceeding the capacity and for finding another gender in the room.

So, this second approach (**the one I implemented**) will be (per thread):

```
Pthread mutex;
Pthread condition notallowed;
Int nmen,nwomen;
define the capacity;

    Void manfunction()
        lock(mutex)
        print("a man  Enter " );
        nmen +=1; //increment the counter
        while (nmen >= capacity or nwomen !=0)
            wait(notallowed, mutex)// either full or another gender inside
        unlock(mutex)
        print("using the bathroom");
        delay(some time);
        lock(mutex)
        nmen--;
        broadcast(notallowed)// tell (signal) all threads condition is
released.
        print("man left")
        unlock(mutex)

Void womanfunction()
        lock(mutex)
        print("a woman  Enter " );
        nwomen +=1; //increment the counter
        while (nwomen >=capacity or nmen !=0)
            wait(notallowed, mutex)// either full or another gender inside
        unlock(mutex)
        print("using the bathroom");
        delay(some time);
        lock(mutex)
```

nmen--;
broadcast(notallowed)// tell (signal) all threads condition is released.
print("Woman left")
unlock(mutex)

The **implementation** is divided into functions (man wants to enter, woman wants to enter, man leaves , woman leaves, manfunction, womanfunction (thread functions)).

1. woman_wants_to_enter()
   The function does the first part from the above pseudo code. It locks the mutex to make sure no other threads access this section, increment the variable, or gets into a wrong condition. Then, it checks on the conditional pthread to make sure we are within the capacity and that no other gender is inside.
   Then, it increments some variables as explained in the comments below and then unlocks the mutex.

```
void  woman_wants_to_enter()
{
    sem_wait(&Q); //take a place in the bathroom or  join the waiting queue if no place in bathroom
    pthread_mutex_lock(&mutex); //obtain a lock (mutex) to execute the critical section

    while(num_women >= bathroom_capacity || num_men !=0) // mark as critical section / don't enter if capacity exceeded or there is another gender inside
        pthread_cond_wait(&not_allowed, &mutex);

    num_women++; // increment the num of women in the bathroom
    total_women_in++;// inc the total num of women who have been served so far
    printf( format: "     Woman num %ld Enters, num women in the bathroom : %ld \n", total_women_in, num_women); // print the women ID and the current women in bathrrom
    pthread_mutex_unlock(&mutex); //release the lock.
    sem_post(&Q); // leave the bathroom or  waiting queue.
    //pthread_exit(0);
}
```

2. woman_leaves()
   This function implements the second part of the code above. It locks the mutex to ensure that no one has access to do anything with the variables or wrongly release the condition. It changes some variables, and then signals all the threads that if they were waiting for the pthread condition, the condition is released now.

To elaborate, the wat p**thread_cond_wait, and pthread_cond_broadcast work** is as follows (more about them can be found in the documentation attached in the references):

a. There is some condition (while loop with conditions) that if are true, the thread calls the wait, **unlocks the mutex**, and goes to sleep condition.
b. Another thread **acquires the mutex** and executes its code and it if it executed something that will make the condition of the waiting thread false, it **broadcasts that** or simply signals it.
c. After that, the thread that has been waiting **gains the mutex again** and **rechecks** the condition in the while, if false, it continues the code, it true, it waits again and so on.

So, if **the bathroom was full or had another gender or both**, the waiting thread for that knows when broadcasting, reevaluates the condition and continues that way. This was, it is guaranteed that the two constraints are always met.
The function then releases the mutex and returns.

```
void  woman_leaves()
{

    pthread_mutex_lock(&mutex); //acquire the lock
    total_women_out++; //inc the number of women out
    num_women--; //woman finished

    pthread_cond_broadcast(&not_allowed); //signal all threads

    printf( format: "             -->> Women num %ld left, num women in the bathroom  %ld \n", total_women_out, num_women);

    pthread_mutex_unlock(&mutex); //unlock

    //   pthread_exit(0);
}
```

3. Women Function:

-It is a thread function, so it takes void* as input and returns void*.
-It simply combines the two functions above with some printing.
-It enforces some **delay** for each thread which is **random** to give the thread some time and ensures some fairness (more on that in the following section).
- The usleep function takes an input the num of microseconds to sleep. Here, the delay is random between 0 and about 40 milliseconds.
- the function then returns with NULL (pthread_exit(0))

```
void* women_function(void*arg) //integrates two functions above together (thread function)
{

    // these following lines are for the purpose of calculating a fairness measure at the
        pthread_mutex_lock(&wm);
        num_women_arrived++;
        pthread_mutex_unlock(&wm);

    printf( format: "New Woman arrives\n"); // print arrival to monitor arrival and service
    woman_wants_to_enter(); //enter the bathrrom or request it

    usleep( useconds: rand()%40000); // random wait for the thread to simulate that it takes
    woman_leaves();// leave the bathroom

    pthread_exit( retval: 0);
}
```

4. Men functions are the same logic as women with change in variables.
5. Main function:

    The main function takes the **input from the user** (num of men and women threads).
    It creates the threads, initializes them and launches them.
    Here, it launched one woman thread followed by one man thread to prevent one gender **of being favored** over the other (more on that later.)
    It then uses pthread_join on all threads to make sure that the main function will not proceed **until all threads finish execution** .
    The main also calculates the execution time of the program and other things.

```
pthread_t men_threads[n],women_threads[m];//intialize the threads
pthread_attr_t attr;
pthread_attr_init(&attr); //initialize the threads attributes
// launch threads one after the other to get some fairness
lp(i, n: max(n,m), s: 0){

    if (i<m)
        if(pthread_create(&women_threads[i],&attr,women_function, arg: NULL)!=0);  // launch a woman thread

    if(i<n )
        if(pthread_create(&men_threads[i],&attr,men_function, arg: NULL)!=0);// launch a man thread

}

// use join threads on both man and women threads to wait for all threads to finish before proceeding in the code.
lp(i, n: max(n,m), s: 0){
    if (i<m)
        pthread_join(women_threads[i], thread_return: NULL);
    if(i<n)
        pthread_join(men_threads[i], thread_return: NULL);
}
```

Fairness Guarantees and Measures:

This Implementation ensures fairness for both genders in **Four ways**:

1. **Launching the threads together**:

   By launching one woman thread followed by one man thread, the threads are given almost the same chance to compete for the resource (the bathroom). If women threads were all launched first or the opposite, that will give a huge advantage to the gender launched first, making the other gender starve since they will have to **wait** for most of the threads of the other gender to finish.

2. **Forcing the thread to wait some time:**

   This way, the thread is holding for some random time, which gives a chance for other threads from both genders to arrive and also gives time for the opposite gender t**o compete for the resource.** If there was no delay, the threads from the first gender that arrive will finish extremely fast, and the other gender might not get a chance to execute in the middle.

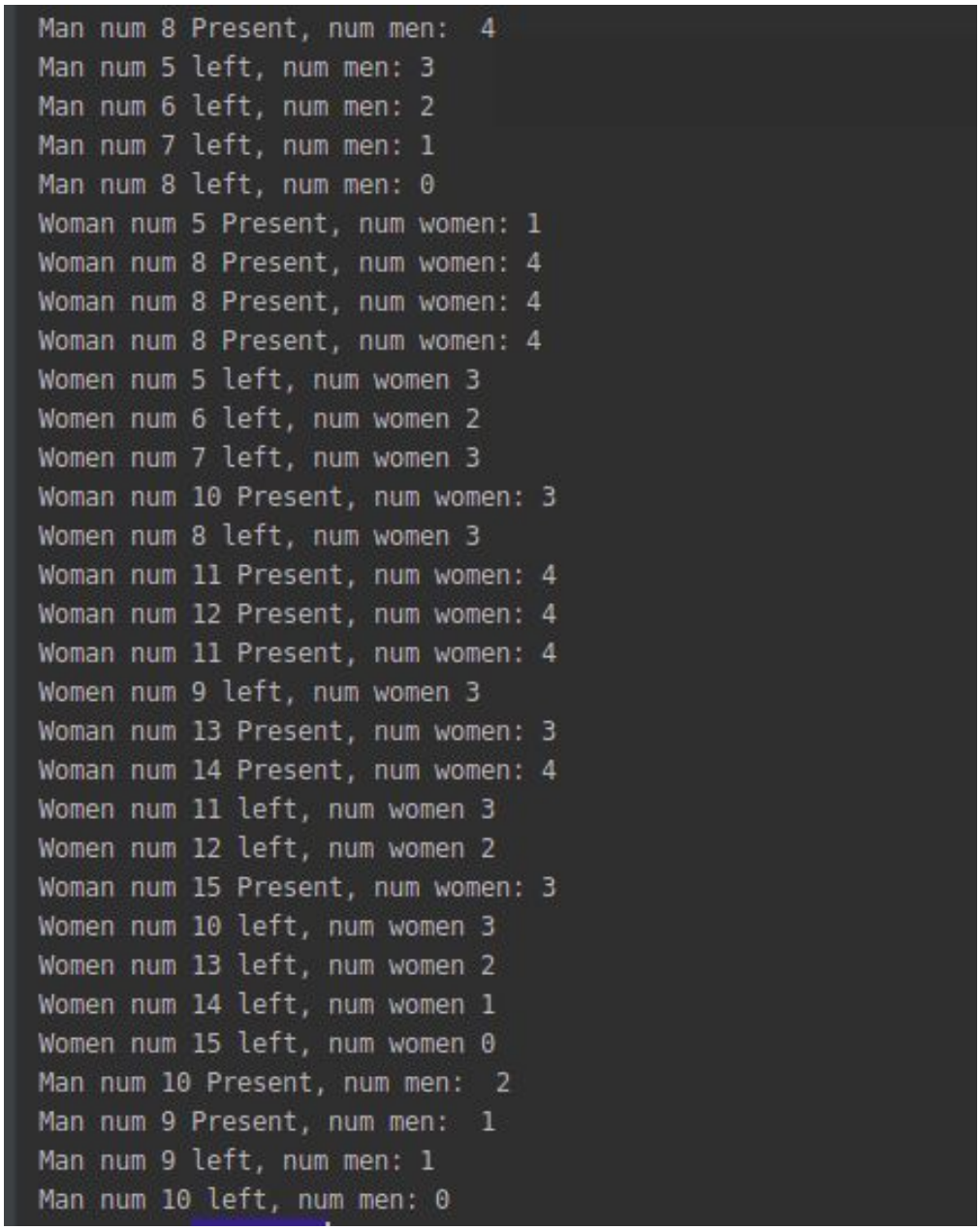
3. **Using Pthread Condition wait and signal:**

   As explained above, the pthread_cond variable is shared among all threads as well as the mutex. Therefore, it can be accessed by both gender threads and hence no gender is favored over the other. To give an example, if there is one man inside the room and a woman **wants to enter.** The woman thread will call the wait, and so when the man finishes, the woman will know and can use the bathroom since the conditional variable is shared between the two.

   Note that, this approach alone is enough to ensure correctness and **some fairness**; however, obviously, the room has to be full with the same gender first then emptied before any woman is allowed. For example, if the sequence for arrival was man woman woman woman man man, the code will serve the **three men first,** waits for them to leave, then allow the women in. This is fair in the sense that both gender get the chance to use the resource interchangeably, but **fairer approach** would be to serve the man first then the three women then the two men (i.e. **gives a priority also based on arrival order**). This fairer approach is implemented using a queue as shown in step the fourth way.

   Below is one example of the output using only the first three approaches together (without the queue). The max capacity of the room is 4; it is noted that as explained,

that they interchange, but if a man gets in, the bathroom has to be full with men then emptied before a woman gets in if it got the mutex before another man of course.



4. **Using a Semaphore Queue:**

   It is known that semaphores can be used as a queue **ordering the processes** that call wait on it and upon signal, the next in queue can obtain the mutex and continue the process explained above.

So, we start by calling wait on that Queue. If there is a place inside the bathroom, it will continue and obtain the mutex lock, if not, the thread will go and wait in the queue of the semaphore. After finishing all of that, it signals (sem_post) the queue to free a space in

the queue. In other words, it is similar to saying that I don't need to wait in the queue; I already got in.

```
sem_wait(&Q); //take a place in the bathroom or  join the wa
pthread_mutex_lock(&mutex); //obtain a lock (mutex) to execu

while(num_women >= bathroom_capacity || num_men !=0) // mark
    pthread_cond_wait(&not_allowed, &mutex);

num_women++; // increment the num of women in the bathroom
total_women_in++;// inc the total num of women who have been
printf( format: "     Woman num %ld Enters, num women in the b
pthread_mutex_unlock(&mutex); //release the lock.
sem_post(&Q); // leave the bathroom or  waiting queue.
```

The semaphore queue is initialized to a size in the main using:

```
sem_init(&Q,0,max(bathroom_capacity/2,1) );
```

This statement determines the size of that queue, with some trial and error, I found that it performs best (most fairness) when it's of size **bathroom capacity/2.**

An example of that fairness is shown below. Despite having a capacity of 4, the code treats the incoming **with all fairness.** The sequence of arrival for example is man man woman man woman ...etc. It served the two men first, then emptied the room and served one women (i.e. it didn't let the third man in because th**e women arrived first**).

```
Man num 1  Enters, num men in the bathroom :  1
Man num 2  Enters, num men in the bathroom :  2
      -->> Man num 1 left, num men in the bathroom : 1
      -->> Man num 2 left, num men in the bathroom : 0
Woman num 3 Enters, num women in the bathroom : 1
        -->> Women num 3 left, num women in the bathroom  0
Man num 3  Enters, num men in the bathroom :  1
      -->> Man num 3 left, num men in the bathroom : 0
Woman num 4 Enters, num women in the bathroom : 1
        -->> Women num 4 left, num women in the bathroom  0
Man num 4  Enters, num men in the bathroom :  1
      -->> Man num 4 left, num men in the bathroom : 0
Woman num 5 Enters, num women in the bathroom : 1
        -->> Women num 5 left, num women in the bathroom  0
Man num 5  Enters, num men in the bathroom :  1
      -->> Man num 5 left, num men in the bathroom : 0
Woman num 6 Enters, num women in the bathroom : 1
        -->> Women num 6 left, num women in the bathroom  0
```

Fairness Measures

To further measure fairness quantitatively, I **came up with an unfairness measur**e that is calculated in the code. The Basic idea is that If many women or men arrived and they are kept for long without being served, that would be unfair. So, I define the unfairness as

Unfairness= **abs(Max number of women waited without being served - the max num of men Waited without being served)**

So, if this **difference is large**, that means that on average there were one gender who arrived but kept waiting while the other was served, which is unfair. The closer this number to zero is, that means that the program is **fairer**. I then **normalize** it by dividing on the total number to make it a number between **(0 and 1).** So, the closer this quantity to zero, the fairer the program is.
The implementation of this is done by (same logic for men women.):

Upon arrival, lock some section (unrelated to the mutex , it's just to count correctly.)
Then increment the counter of the number of arrived.

```
pthread_mutex_lock(&wm);
num_women_arrived++;
pthread_mutex_unlock(&wm);
```

Then update the difference after each new arrival.

```
diff=num_women_arrived-total_women_in;
if (diff > max_num_women_waiting )
        max_num_women_waiting=diff;
```

Then calculate the unfairness and normalize it :

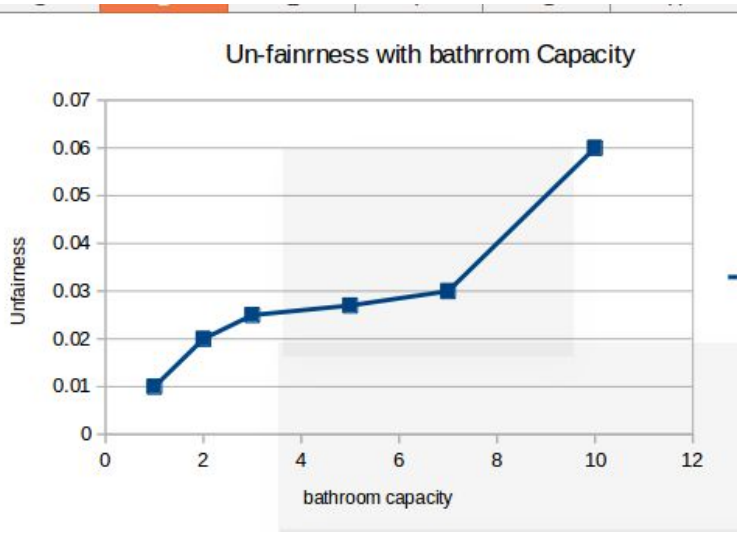
```
printf("Un-Fairness Measure: %f\n", fabs(max_num_men_waiting / (float)n -
max_num_women_waiting/(float)m) );
```

I then do some **numerical tests** on this measure to get some plots. I plot it against, the max capacity of the bathroom, the total number of threads and the difference between the threads (data and plots are also attached in the project folder.).

First

The graph between the bathroom capacity and unfairness is shown below. It is an **increasing trend**, which makes sense because the more the capacity is, the more number of one gender can be allowed in and the more the other gender has to wait before being served, leading to more unfainess.

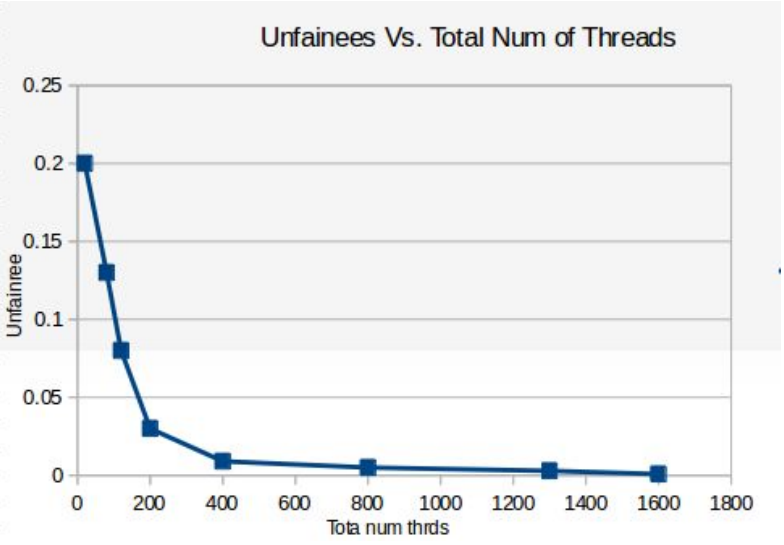
ath Cap	Unfainress
1	0.01
2	0.02
3	0.025
5	0.027
7	0.03
10	0.06



Second

The graph between the total num of threads (num of men threads + num women threads.) and unfairness is shown below. Here, the num of threads for both genders is **fairly close** to each other., the bathroom capacity is constant at 4 It is a **decreasing trend**, which makes sense because the more the number of threads is, the more the chance of threads to **compete with each other**, not giving the chance for one gender to dominate over the other.

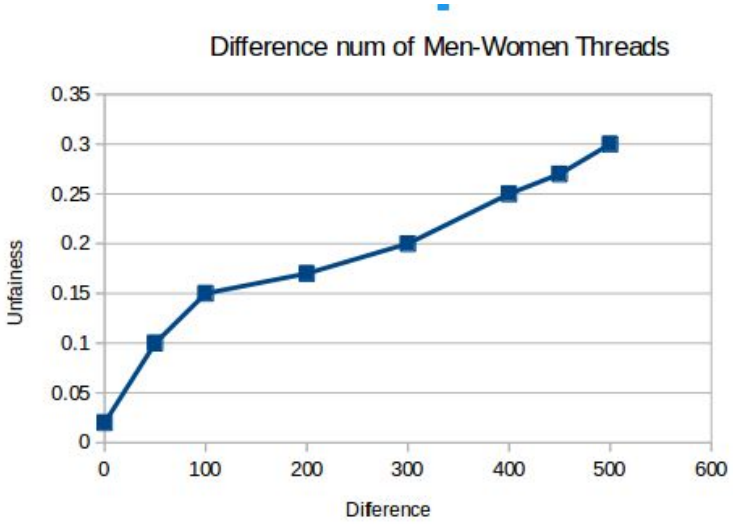
Total Num Thrds	Unfainress
20	0.2
80	0.13
120	0.08
200	0.03
400	0.009
800	0.005
1300	0.0029
1600	0.0008



Third

The graph between the **difference in num of threads** (abs(num of men threads - num women threads).) and unfairness is shown below. Here, I assume that we start by an equal num of threads (i.e. 100 for men and 100 for women). Then, I increase the difference between them (e.g. make one 120 and the other 160) and so on. The graph shows an **increasing trend**, which is valid because if the difference between the num of threads is big, this means that the chance of one needs being fulfilled over the others is larger (for example if the difference is 100, it means there is a **big chance** 100 threads of one gender will be kept waiting until the other two genders are served equally first.)

Difference	Unfainress
0	0.02
50	0.1
100	0.15
200	0.17
300	0.2
400	0.25
450	0.27
500	0.3



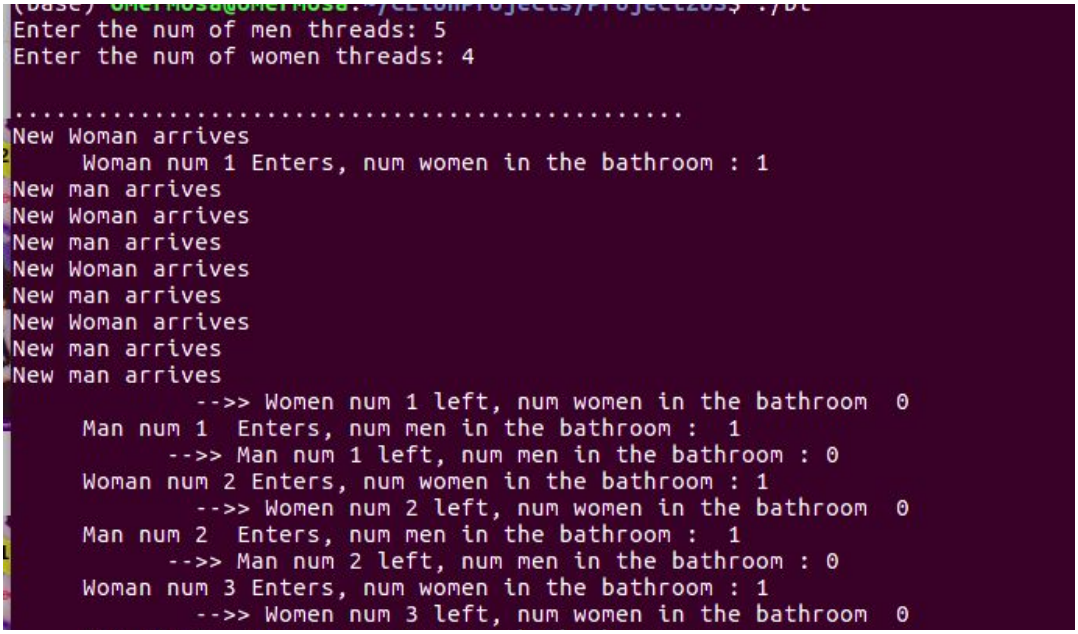
The conclusion here is that the fairest situation happens when threads are of equal number, bathroom capacity is one, and their number is big. Of course a capacity of one is unrealistic. So, the rule is to lower the capacity, keep the numbers close and high.

Tracing a small example:

I trace a small example here to further emphasize how the code works. In this example, The bathroom capacity is 1 and the semaphore queue capacity is 1.

1. Men and women arrive in the order shown.
2. The first women (women 1) enters its function. This call the **wait on the queue**. The queue has no one waiting, so it passes it and enters the mutex. It locks the mutex, increment the num_women and prints that woman 1 entered.

3. The following man tries to enter and acquire the mutex, but it can't since woman 1 **didn't signal** it yet. So, it waits on it and becomes first in the queue or waiting.
4. More men and women arrive; they also **join the waiting queue** in order. So, now, the waiting queue order is [man woman man woman man woman man man].
5. After that woman 1 leaves the bathroom, which means she already **signaled** the queue.
6. The first one in the queue is man 1, who can now **acquire the mutex** and enters the bathroom.
7. This process keeps on repeating for all in the queue, which the **fairest thing possible**.



Deadlocks Impossibility:

The deadlocks cannot happen in this implementation. For them to happen, there has to be a cycle of using the shared resources. We have only three shared objects: the mutex, the queue, and the condition.

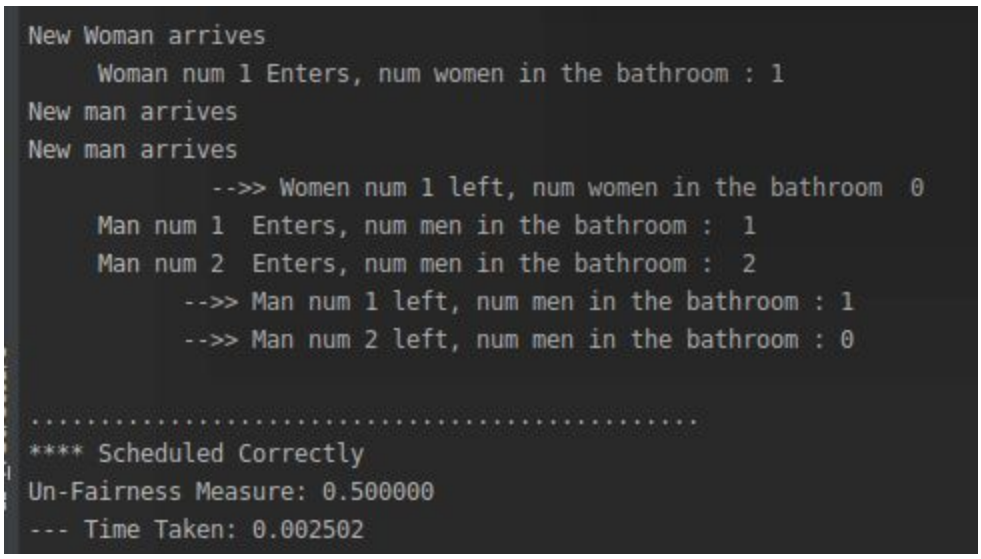
Obviously, since the queue is the first line called in the function, it cannot be held without release because: first, it's a semaphore, so it can be **waited on as many** as possible as long as it's **signalled back**. Therefore, as long as its following section (the mutex, the increment of variables and the pthread condition) executes, it cannot be held without release or cause a deadlock.

For the mutex, It is shared between the two genders, so if there is no situation in which the **lock of the mutex is not unlocked** due to **conflict** in resources, there will be no deadlocks. The same goes for the **pthread condition.**

The one conflict happens in the code when a **man enters the critical section and locks the mutex and there is a woman inside the bathroom**. The thread then has to call pthread wait until the bathroom is empty. However, for the woman to leave the bathroom, she has to acquire the mutex, but the man already locked it. This **would have caused** a deadlock if we are using a regular semaphore for waiting. However, the **pthread_cond_wait** released the mutex so that the one causing the condition can go and **release** the condition causing it to wait. Hence, **there is no deadlock here**, the women will acquire the released mutex and leave the bathroom. Then, the mutex will be returned to the man, who will make sure that the woman left by reevaluating the condition, and then **continue execution** in its critical section.

Output:

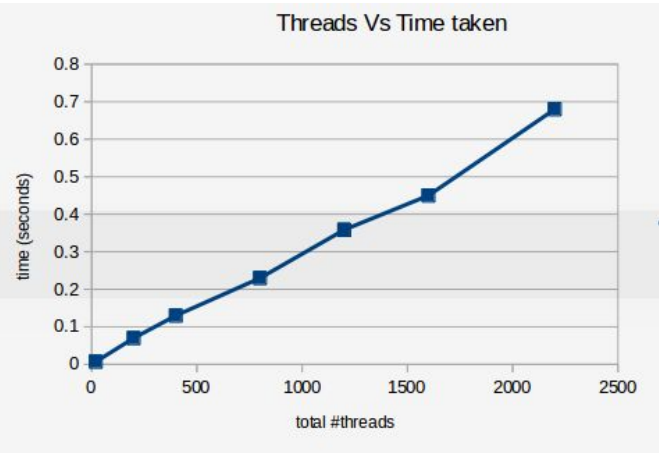
The console outputs as shown: when a new man/woman arrives. When a man /women enters (with their Id and order and the num of them currently in the bathroom). And when a man/woman leave (also with their ID and num left remaining in the bathroom).
It outputs the unfairness, the time taken by the program, and a flag marking if the scheduling was done correctly (it checks that by comparing the number that should have been scheduler vs. the number that actually scheduled.)



Performance and Features:

1. The time taken by the program is not large. Below is a measure of time taken vs the number of threads. The time almost **grows linearly** with the number of threads, which is reasonable given the overhead of synchronization object, and it scheduler 2200 threads in only **0.68 seconds**.

Total Num Thrds	Time
20	0.007
200	0.07
400	0.13
800	0.23
1200	0.3586
1600	0.45
2200	0.68



2. **Low overhead** due to the use of only one mutex, one pthread conditional variable and one semaphore
3. **Very fair** due to the integration of many things as detailed above.

4. The num of threads can be controlled via **user input** (flexible)
5. Doesn't have a lot of **dependencies** on external libraries (the native pthreads only.)

How to run:

1. Either running the executable attached from the terminal using **./bathromm_synch_sol (**after being in the directory of it of course**)**
2. Using only gcc compiler:
   gcc bathroom\problem.c -o bathromm_synch_sol -lpthread

3. From a regular IDE (like Clion):
   Add this line to the CmakeLists.txt `SET(CMAKE_C_FLAGS -pthread)`
   Then, run the program from the IDE.

How to run:

[1] Operating System Concepts Book 10th edition.

[2]https://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_cond_broadcast.html

[3]https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_cond_wait.html

[4]https://github.com/prateekparallel/InterThreadCommunication/blob/master/ThreadControl/ThreadControl.c

[5]https://www.youtube.com/watch?v=rMpOfbaP2PQ
[6]https://www.youtube.com/watch?v=1ks-oMotUjc&list=PLzCAxcRKUAgkc65DlRo0gr0B8sqlE6WZY&index=2

[7]http://man7.org/linux/man-pages/man3/sem_post.3.html

[8]https://stackoverflow.com/questions/33991918/link-to-pthread-library-using-cmake-in-clion

[9] http://shivammitra.com/reader-writer-problem-in-c/#