(A)

Implement the Dining Philosophers and Reader Writer Problem of Synchronization (test drive the codes discussed in the class).

Dining Philosophers (codes given in class)

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>

#define THINKING 0
#define HUNGRY 1
#define EATING 1
#define N 5
#define LEFT (ph_num + 1) % N
#define RIGHT (ph_num + 4) % N

//  2 semaphores
// mutex -- binary sem... here it is like the checker to resource
allocations
sem_t mutex;// overall synchronization across philosophers... provides the
fair chance for other threads to use the resource
sem_t S[N]; // the adjacent philosphers are not eating... this is only for
this code

void * philosopher(void *num); // runner in thread
void take_fork(int);
void put_fork(int);
void test(int);
int state[N]; // flag in peterson's... desire to enter critical section
int phil_num[N] = {0, 1, 2, 3, 4};
int main(int argc, char *argv[])
{
   int i;
   pthread_t thread_id[N];
```

```
   // let the sem be shared... (, 0, ) --- shared or non-shared across
threads--- 0--> shared across threads
   sem_init(&mutex, 0, 1); // (,, 1) -- initial start variable
   for ( i = 0; i < N; i++)
   {
       sem_init(&S[i], 0, 0);
   }
   for ( i = 0; i < N; i++)
   {
       pthread_create(&thread_id[i], NULL, philosopher, &phil_num[i]);
       printf("Philosopher %d is ret \n", i + 1);
   }
   for ( i = 0; i < N; i++)
   {
       pthread_join(thread_id[i], NULL);
   }
   return 0;
}
void * philosopher(void *num)// runner in thread
{
   while(1)
   {
       // sleep are used to ensuring the locking functions are working
properly
       int * i = num;
       sleep(1);
       take_fork(*i); // it is taking 2 forks
       sleep(0);
       put_fork(*i);
   }
   pthread_exit(NULL);
}
void take_fork(int ph_num)
{
   // state[ph_num] = HUNGRY; /// try this
   sem_wait(&mutex);  // gain the lock over mutex --- mutex becomes 0 from
1
   state[ph_num] = HUNGRY;
   printf("Philospher %d is hungry\n", ph_num + 1);
   test(ph_num); // test whether the thread can move to the eating state
```

```
   sem_post(&mutex); // release ... to not hold the outer synchronization
of threads
   sem_wait(&S[ph_num]); // test will make 0-->1 here it comes bakc to
1-->0
   sleep(1);   // test if sombody else is allowed to take control
}
void test(int ph_num)
{
   if(state[ph_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
   {
       state[ph_num] = EATING;
       sleep(2);
       printf("Philosopher %d takes the fork %d and %d\n", ph_num + 1,
LEFT + 1, ph_num + 1);
       printf("Philosopher %d is eating\n", ph_num + 1);
       sem_post(&S[ph_num]); // release the lock
   }
}
void put_fork(int ph_num)
{
   // mutex lock in the take fork
// it is as the every other thread would release lock in the take_fork --
means the post of mutex is allowed to happen (it can wait till it gets the
lock)
   sem_wait(&mutex);
   state[ph_num] = THINKING;
   printf("Philosopher %d releases the fork %d and %d\n", ph_num + 1, LEFT
+ 1, ph_num + 1);
   test(LEFT);
   test(RIGHT);
   sem_post(&mutex);
}
```

Output

```
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8$ gcc Nov12.c -lpthread
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8$ ./a.out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philospher 1 is hungry
Philosopher 1 takes the fork 2 and 1
Philosopher 1 is eating
Philospher 2 is hungry
Philospher 3 is hungry
Philospher 4 is hungry
Philospher 5 is hungry
Philosopher 1 releases the fork 2 and 1
Philospher 1 is hungry
^C
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8$ ./a.out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philospher 1 is hungry
Philosopher 1 takes the fork 2 and 1
Philosopher 1 is eating
Philospher 2 is hungry
Philospher 4 is hungry
Philosopher 4 takes the fork 5 and 4
Philosopher 4 is eating
Philospher 3 is hungry
Philospher 5 is hungry
Philosopher 1 releases the fork 2 and 1
Philosopher 4 releases the fork 5 and 4
Philosopher 5 takes the fork 1 and 5
Philosopher 5 is eating
Philospher 1 is hungry
Philospher 4 is hungry
Philosopher 5 releases the fork 1 and 5
Philospher 5 is hungry
```

Note

I am not sure but the program has some problems, it sometimes enters into a deadlock and hence I have attached another solution to the Dining Philosopher in the G-Drive.

Reader Writer Problem

```c
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>

#define MAX_THREADS 5
```

```
void * reader(void *arg);
void * writer(void * arg);

/* incomplete due to rush hour .. READ SECCTION CONTAINS THINGS NEEDED TO
BE DONE*/
sem_t mutex;
sem_t writeblock;
int data = 0;
int rcount = 0;

int main(int argc, char * argv[])
{
   int i, b;
   pthread_t rtid[MAX_THREADS];
   pthread_t wtid[MAX_THREADS];
   sem_init(&mutex, 0, 1);
   sem_init(&writeblock, 0, 1);

   for (int i = 0; i < MAX_THREADS; i++)
   {
       pthread_create(&rtid[i], NULL, reader, (void *) i);
       pthread_create(&wtid[i], NULL, writer, (void *) i);
   }
   for (int i = 0; i < MAX_THREADS; i++)
   {
       pthread_join(rtid[i], NULL);
       pthread_join(wtid[i], NULL);
   }

   return 0;
}
void * reader(void * arg)
{
   int id = (int) arg;
   sem_wait(&mutex);
   rcount += 1;
   if(rcount == 1)
       sem_wait(&writeblock);
   sem_post(&mutex);
```

```
    printf("Data read by the reader %d is %d\n", id, data);
    sleep(1);
    sem_wait(&mutex);
    rcount -= 1;
    if(rcount == 0)
        sem_post(&writeblock);
    sem_post(&mutex);
}
void * writer(void * arg)
{
    /*get write unlock and then do the change data for showing*/
    int id = (int) arg;
    sem_wait(&writeblock);
    data++;
    printf("Data write by the writer %d is %d\n", id, data);
    sleep(1);
    sem_post(&writeblock);
}
```

Output

```
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8$ ./a.out
Data read by the reader 0 is 0
Data read by the reader 1 is 0
Data read by the reader 2 is 0
Data read by the reader 3 is 0
Data read by the reader 4 is 0
Data write by the writer 0 is 1
Data write by the writer 1 is 2
Data write by the writer 2 is 3
Data write by the writer 3 is 4
Data write by the writer 4 is 5
```

(B)

Choose any 2 of the following problems whose details are available in the Downy Book on Semaphores (attached) and implement semaphores based solutions to the same.

(1)Santa Claus Problem
(2) H2O Problem
(3) Baboon Crossing Problem
(4) Dining Hall Problem

(5) Senate Bus Problem

Baboon Crossing Problem

Problem Description

In the baboon crossing problem, a number of baboons are located on the edges of a deep canyon. There are baboons on both sides and, as you have probably guessed, some of the baboons on the left side want to get to the right side, and vice versa. Fortunately, a large rope has been stretched across the abyss allowing baboons to cross by brachiation: hanging from the rope and swinging hand-over-hand to the other side.

The baboon crossing policy must be compatible with the constraints of the situation:

1. If two baboons meet in the middle of the rope, then all activity stops, and no other baboons can cross (deadlock). So, at any given time, all the baboons on the rope must be going the same direction.
2. The rope can hold only a certain number of baboons at a time. A baboon cannot be allowed on the rope if the rope is already at full capacity (limitation to resource).
3. A continuous stream of baboons from one direction could prevent baboons wanting to go the opposite direction from ever being able to cross (unfairness, starvation).

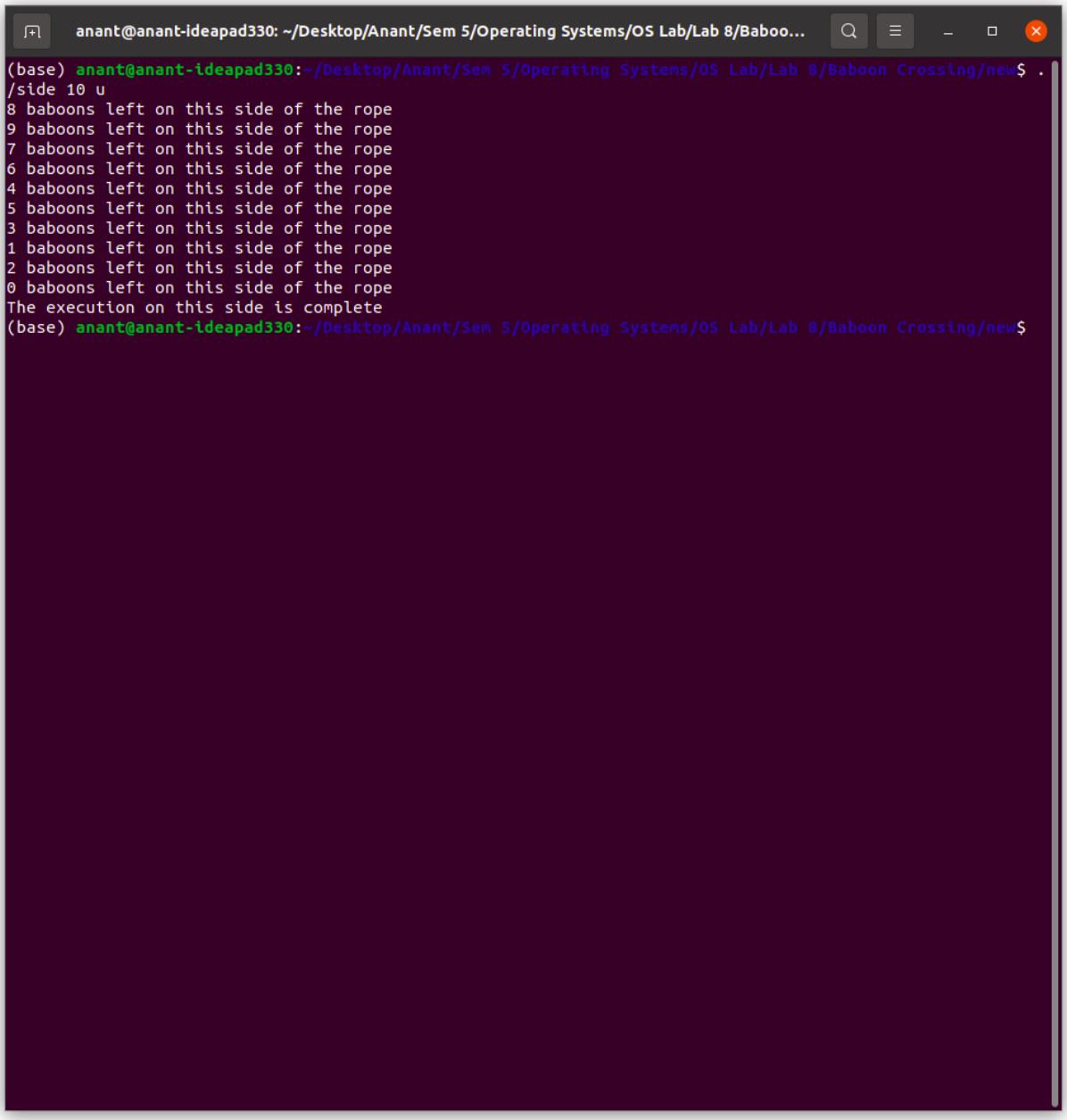
Here I have not taken into account the delay in the process re-entering the queue on the other side, as soon as the process is released from the rope, it enters the queue on the other side.
Inputs:
The inputs are taken from the command line, the input format is
First-side: ./<exec> <no_of_baboons_going_over> <some_arg>
The <some_arg> is used to destroy the semaphores in the end. The frist side is identified by the 3 arguments whereas the second-side will have only 2 arguments as command line input
Second-side: ./<exec> <no_of_baboons going_over>
The distinction between first and second side is necessary as the first argument unlinks any semaphore by the name used, whereas the second only connects to this semaphore created.
The movement between the 2 side now is symmetric, the side cross over to each other
Procedure Used:

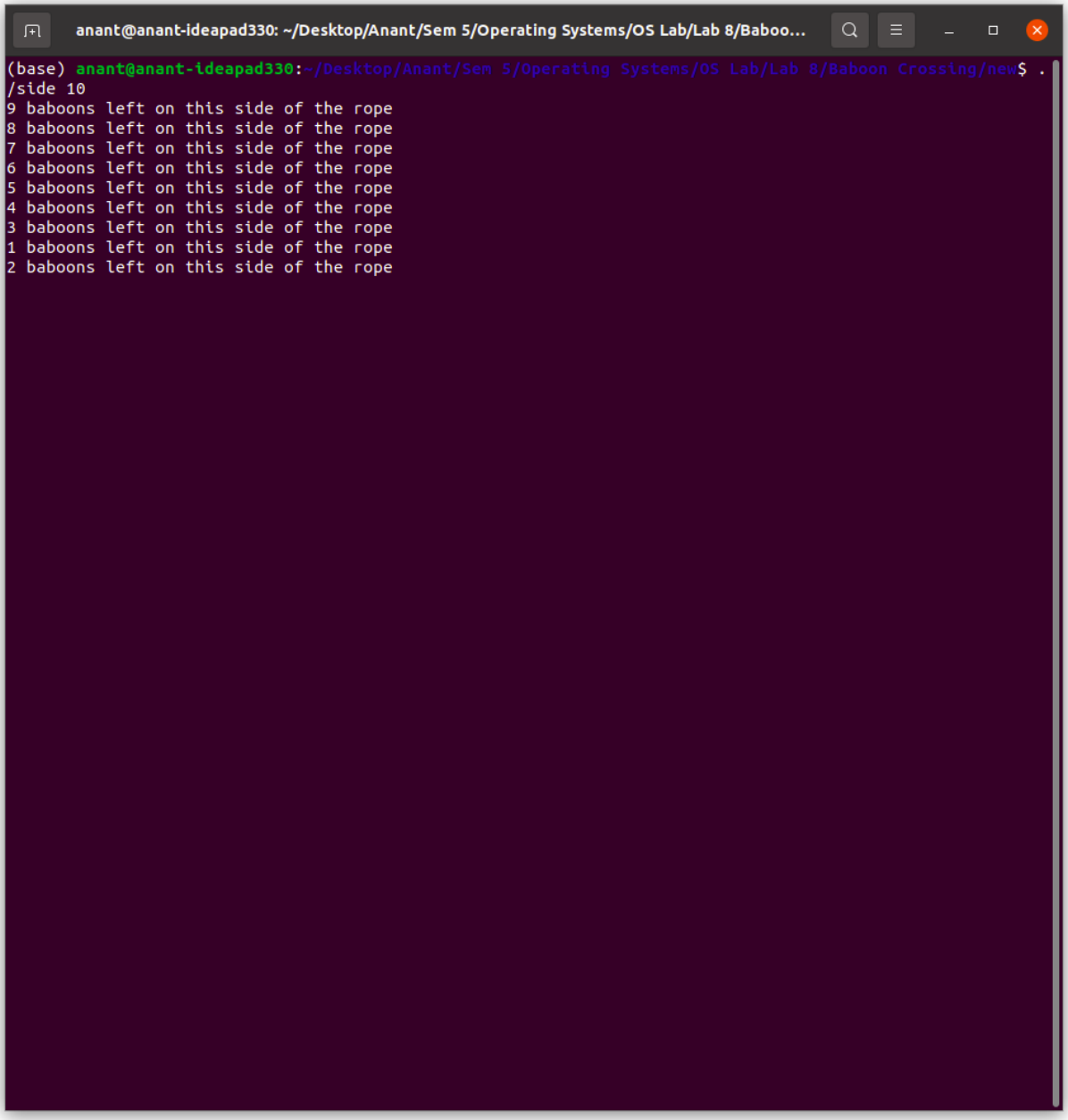
1. The threads created in the control() function are equivalent to the baboons, the baboons, the threads are created when there is a baboon left on ‘this’ side of the rope
2. The crossover is simply making the thread wait for some time and decreasing the baboon count on ‘this’ side.
3. After joining the threads, to give the opportunity to switch over to let baboons from ‘other’ side over to ‘this’ side, the control sleeps.
4. This process repeats itself until all the baboons are on the ‘other’ side for both the executing processes.

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <semaphore.h>

#define DIR "/direction"
#define ROPE_SEMA "/rope_sem"
#define ROPE_LIMIT 3

void control();
void * crossover(void *);

sem_t * direction;
sem_t * rope_sem;
int b_count;
pthread_t baboon_on_rope[ROPE_LIMIT];
int main(int argc, char * argv[])
{
   b_count = atoi(argv[1]);
   if(argc == 3)
   {
       sem_unlink(DIR);
       sem_unlink(ROPE_SEMA);
   }

   direction = sem_open(DIR, O_CREAT, 0660, 1);
```

```
    rope_sem = sem_open(ROPE_SEMA, O_CREAT, 0660, ROPE_LIMIT);
    control();
    printf("The execution on this side is complete\n");
    return 0;
}
void control()
{
    while (b_count > 0)
    {
        sem_wait(direction);
        for (int i = 0; i < ROPE_LIMIT & b_count > 0; i++)
        {
            pthread_create(&baboon_on_rope[i], NULL, crossover, NULL);
        }
        for (int i = 0; i < ROPE_LIMIT; i++)
        {
            pthread_join(baboon_on_rope[i], NULL);
        }
        sem_post(direction);
        sleep(1);
    }

}
void * crossover(void * param)
{
    sem_wait(rope_sem);
    if(b_count <= 0)
    {
        return NULL;
    }
    printf("%d baboons left on this side of the rope\n", --b_count);
    sleep(1);
    sem_post(rope_sem);
    pthread_exit(NULL);
}
```

Output


anant@anant-ideapad330: ~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8/Baboo...
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8/Baboon Crossing/new$ ./side 10 u
8 baboons left on this side of the rope
9 baboons left on this side of the rope
7 baboons left on this side of the rope
6 baboons left on this side of the rope
4 baboons left on this side of the rope
5 baboons left on this side of the rope
3 baboons left on this side of the rope
1 baboons left on this side of the rope
2 baboons left on this side of the rope
0 baboons left on this side of the rope
The execution on this side is complete
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8/Baboon Crossing/new$



More:

The above solution is the simplest deduction of the baboon crossing problem. Here, the baboons are all identical, there is no baboon coming back to ‘this’ side.

H2O Problem

Problem Description

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.
As each thread passes the barrier, it should invoke a bond. You must guarantee that all the threads from one molecule invoke a bond before any of the threads from the next molecule do.
In other words:
• If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
• If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.`

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

unsigned int H2O;         //how many H2O to produce
unsigned int PROD_H;       //given H
unsigned int PROD_O;       //given O

unsigned int order = 0;
unsigned int len = 0;

pthread_mutex_t mutex_hy = PTHREAD_MUTEX_INITIALIZER; //mutex on H
pthread_mutex_t mutex_oxy = PTHREAD_MUTEX_INITIALIZER;  //mutex on O
pthread_cond_t cond_hy = PTHREAD_COND_INITIALIZER; //conditional on H
pthread_cond_t cond_oxy = PTHREAD_COND_INITIALIZER;    //conditional on O

void *prod_hy(void* p)
{
   unsigned int buf = 0;
   unsigned int produced = 0;

   unsigned int seed;
```

```
    unsigned int random_wait;
    seed = time(NULL) ^ getpid() ^ pthread_self(); // just for some random
output
    random_wait = rand_r(&seed) % 1000000; // random wait in micro sec, max
1 sec
;

    while(produced < (2 * H2O) / PROD_H)
    {
        printf("H \t start H production\n");
        //produce random wait in microsec
        usleep(random_wait);
        buf++;
        produced++;

        // add atom produced to water .. lock the on H
        pthread_mutex_lock(&mutex_hy);
        order++;
        printf("H \t thread has H %d waits for another\n", produced);
        pthread_cond_wait(&cond_hy, &mutex_hy);
        printf("H \t produce water\n");
        buf--;
        pthread_mutex_unlock(&mutex_hy);
    }
    pthread_exit(0);
}

void *prod_oxy(void* p)
{
    unsigned int buf = 0;
    unsigned int produced = 0;

    unsigned int seed;
    unsigned int random_wait;
    seed = time(NULL) ^ getpid() ^ pthread_self(); // just for some random
output
    random_wait = rand_r(&seed) % 1000000; // random wait in micro sec, max
1 sec
    while(produced < H2O/PROD_O)
    {
```

```
        printf("O \t start producing Oxygen\n");
        //produce random wait in microsec
        usleep(random_wait);
        buf++;
        produced++;
        // add atom produced to water .. lock the on H
        pthread_mutex_lock(&mutex_oxy);
        len++;
        printf("O \t thread has produced O %d is waiting\n", produced);
        pthread_cond_wait(&cond_oxy, &mutex_oxy);
        printf("O \t produced\n");
        buf--;
        pthread_mutex_unlock(&mutex_oxy);
    }
    pthread_exit(0);
}

void *water_production(void* p)
{

    unsigned int h2o_made = 0;

    while(h2o_made < H2O)
    {
        // critical section
        pthread_mutex_lock(&mutex_hy);
        pthread_mutex_lock(&mutex_oxy);
        if(order >= 2 && len >= 1)
        {
            // wait for enough water atoms to produce water
            h2o_made++;
            order-=2;
            len--;
            printf("H2O \t number of water molecules produced %d\n\n",
h2o_made);
            pthread_cond_signal(&cond_hy);
            pthread_cond_signal(&cond_hy);    //2 H taken
            pthread_cond_signal(&cond_oxy);     //1 O taken
        }
        pthread_mutex_unlock(&mutex_oxy);
```

```
        pthread_mutex_unlock(&mutex_hy);
    }
    usleep(500000);
    printf("H2O \t formed %d water molecule\n", h2o_made);
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf("1 arguments : [num H2O]\n");
        return 0;
    }
    H2O = atoi(argv[1]);
    PROD_H = H2O * 2;
    PROD_O = H2O;

    pthread_t hyderogen[PROD_H];
    pthread_t oxygen[PROD_O];
    pthread_t water;

    for(int i = 0; i < PROD_H; i++)
    {
        pthread_create(&hyderogen[i], NULL, prod_hy, NULL);
    }
    for(int i = 0; i < PROD_O; i++)
    {
        pthread_create(&oxygen[i], NULL, prod_oxy, NULL);
    }
    pthread_create(&water, NULL, water_production, NULL);

    for(int i = 0; i < PROD_H; i++)
    {
        pthread_join(hyderogen[i], NULL);
    }
    for(int i = 0; i < PROD_O; i++)
    {
        pthread_join(oxygen[i], NULL);
    }
```

```
    pthread_join(water, NULL);

    pthread_cond_destroy(&cond_hy);
    pthread_cond_destroy(&cond_oxy);
    return 0;
}
```

Here instead of semaphore solution for this doesn't work properly for me, I put a simple mutex solution for the above problem, the procedure goes:

1. Thread creation of multiple threads for Hydrogen, Oxygen and Water each.
2. The thread created for the Hydrogen made the molecules and so did the Oxygen threads. Until then the water thread waits for the 2 H molecules and 1 O molecule.
3. After receiving the signal to from 2 H and 1 O the water molecule is created,
4. The procedure is repeated until the threads create the required number of water molecules.

Output

```
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8/H2O$ ./a.out 3
H         start H production
H         start H production
H         start H production
H         start H production
H         start H production
H         start H production
O         start producing Oxygen
O         start producing Oxygen
O         start producing Oxygen
H         thread has H 1 waits for another
O         thread has produced O 1 is waiting
H         thread has H 1 waits for another
H2O       number of water molecules produced 1

H         produce water
O         produced
H         produce water
O         thread has produced O 1 is waiting
H         thread has H 1 waits for another
O         thread has produced O 1 is waiting
H         thread has H 1 waits for another
H2O       number of water molecules produced 2

H         produce water
O         produced
H         produce water
H         thread has H 1 waits for another
H         thread has H 1 waits for another
H2O       number of water molecules produced 3

H         produce water
H         produce water
O         produced
H2O       formed 3 water molecule
(base) anant@anant-ideapad330:~/Desktop/Anant/Sem 5/Operating Systems/OS Lab/Lab 8/H2O$
```