OPERATING SYSTEMS

PROJECT -1

TABLE OF CONTENTS

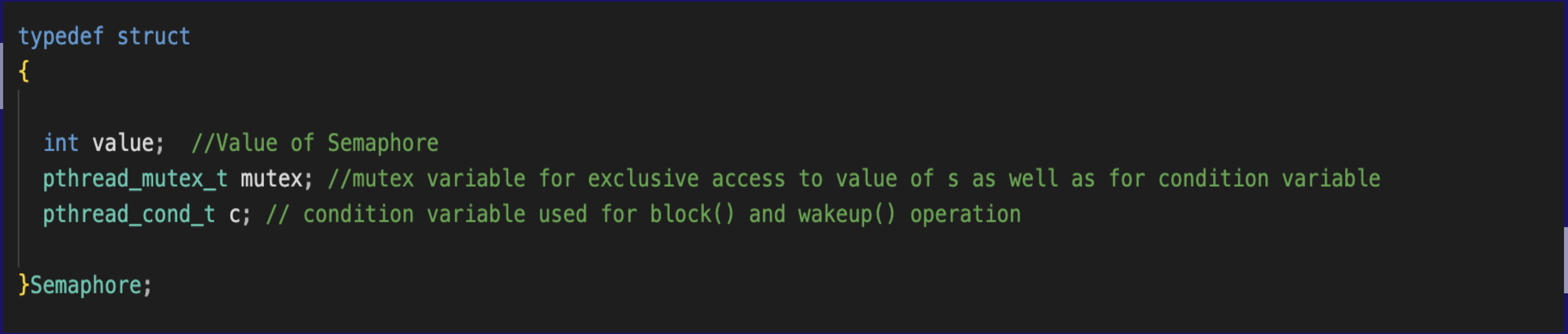
TABLE OF CONTENTS

00

SEMAPHORE IMPLEMENTATION

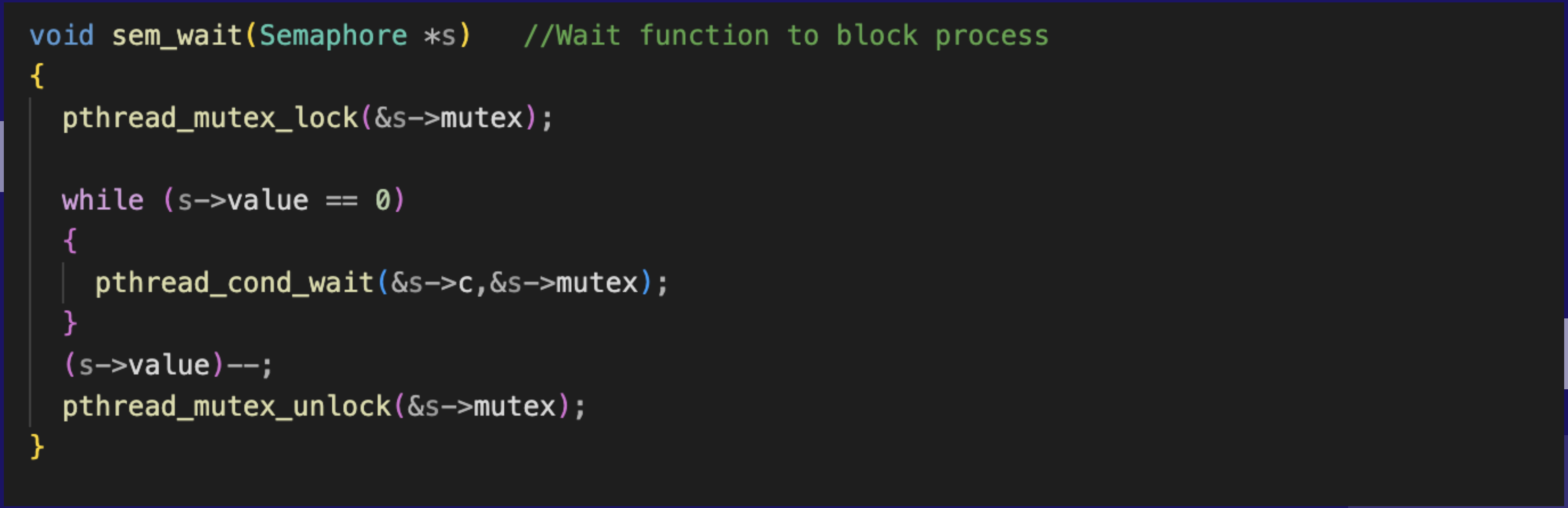
The following is our very own implementation of the Semaphore Struct along with the wait and signal function

Initialization



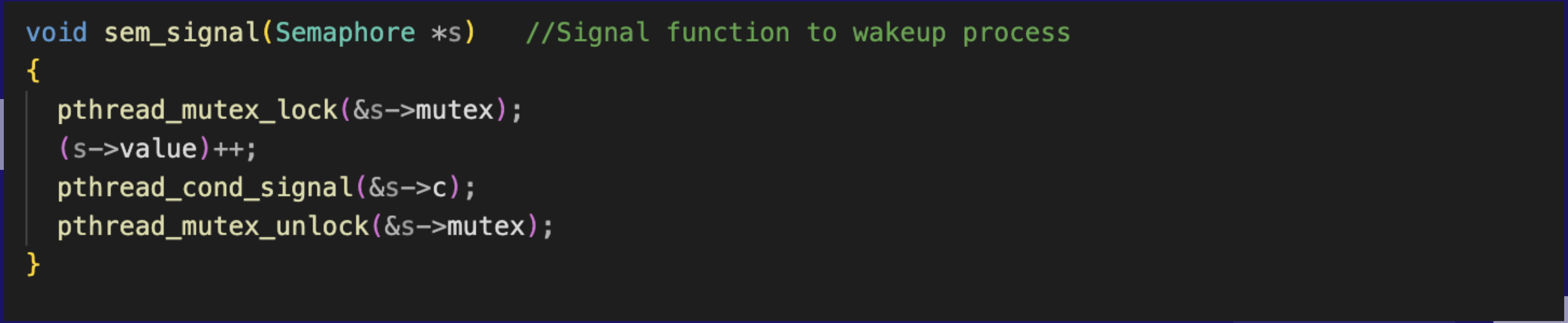
```c
typedef struct
{

  int value;  //Value of Semaphore
  pthread_mutex_t mutex; //mutex variable for exclusive access to value of s as well as for condition variable
  pthread_cond_t c; // condition variable used for block() and wakeup() operation

}Semaphore;
```

Wait Function



```c
void sem_wait(Semaphore *s)   //Wait function to block process
{
  pthread_mutex_lock(&s->mutex);

  while (s->value == 0)
  {
    pthread_cond_wait(&s->c,&s->mutex);
  }
  (s->value)--;
  pthread_mutex_unlock(&s->mutex);
}
```

Signal function



```c
void sem_signal(Semaphore *s)   //Signal function to wakeup process
{
  pthread_mutex_lock(&s->mutex);
  (s->value)++;
  pthread_cond_signal(&s->c);
  pthread_mutex_unlock(&s->mutex);
}
```

01

PRODUCER CONSUMER PROBLEM

Problem is based on synchronization of n producers and m consumers which share a common buffer and try to either produce or consume items using that buffer.

OVERVIEW

The challenge is to ensure that the producers and consumers are synchronized so that the following requirements are met:

1. If the buffer is empty consumer cannot consume an item from that buffer.
2. If the buffer is completely filled then the producer cannot produce an item into that buffer.
3. Producer cannot produce while consumer is consuming and vice versa which ensures mutual exclusion.
4. Producer/Consumer shouldn't wait indefinitely for their turn.
5. There are 3 types of buffer involved in this problem:
   a. Bounded Buffer
   b. Infinite Buffer
   c. Cyclic Buffer

Solution

Variables

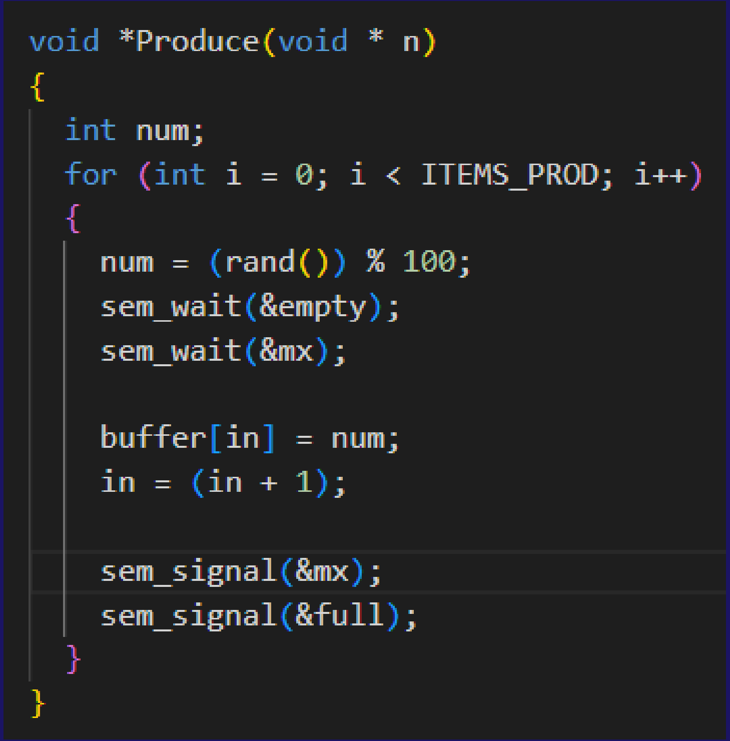
```
in = 0       //buffer index where item will be next produced
out = 0      //buffer index where item will be next consumed
```

Semaphores

```
full = 0     //Counting semaphore that keeps the count of complete buffers
empty = N    //Counting semaphore that keeps the count of empty buffers
mx = 0       //Binary lock to allow the mutual exclusion of shared variables
```

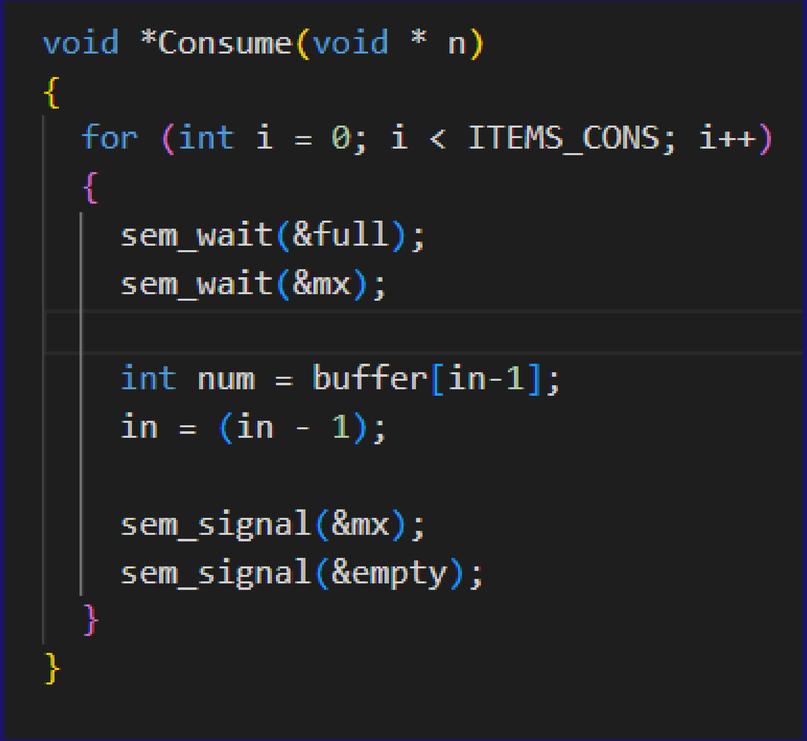
Bounded Buffer

Producer



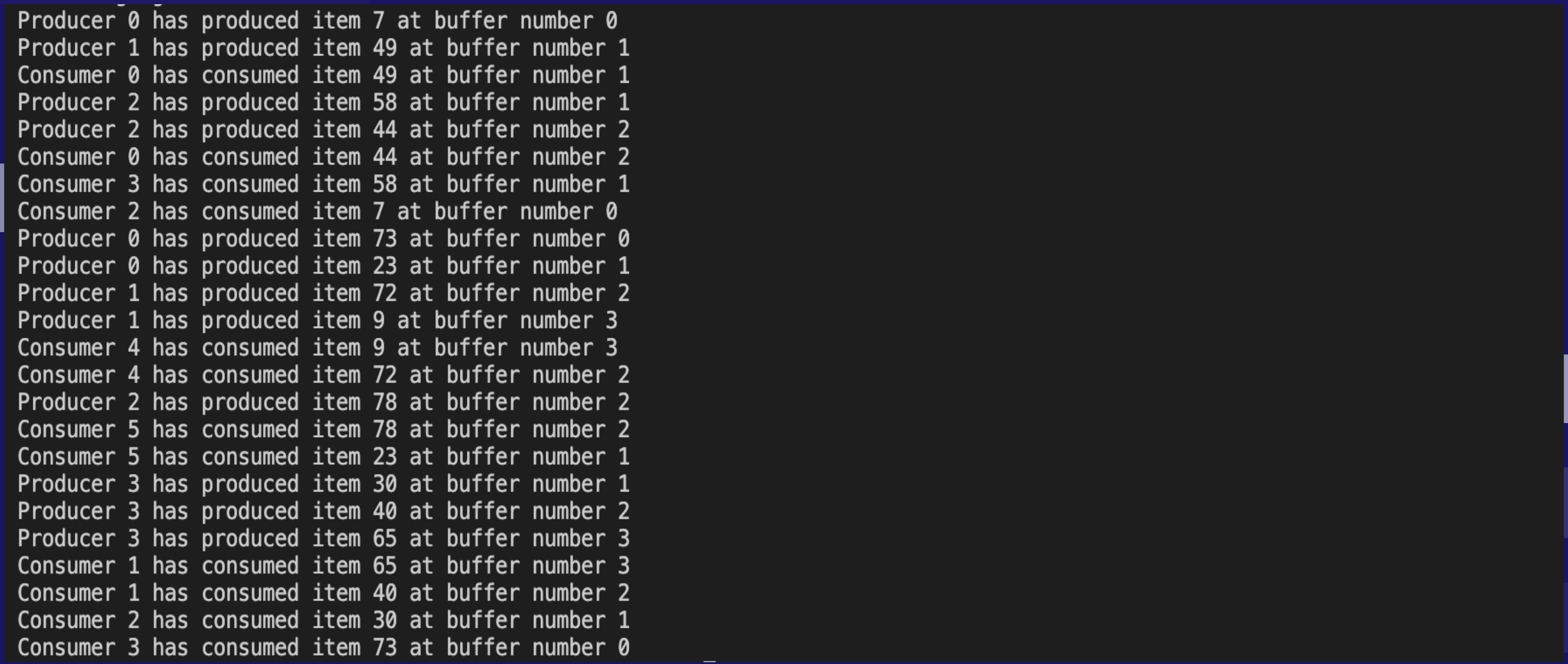
```c
void *Produce(void * n)
{
  int num;
  for (int i = 0; i < ITEMS_PROD; i++)
  {
    num = (rand()) % 100;
    sem_wait(&empty);
    sem_wait(&mx);

    buffer[in] = num;
    in = (in + 1);

    sem_signal(&mx);
    sem_signal(&full);
  }
}
```

Consumer



```c
void *Consume(void * n)
{
  for (int i = 0; i < ITEMS_CONS; i++)
  {
    sem_wait(&full);
    sem_wait(&mx);

    int num = buffer[in-1];
    in = (in - 1);

    sem_signal(&mx);
    sem_signal(&empty);
  }
}
```

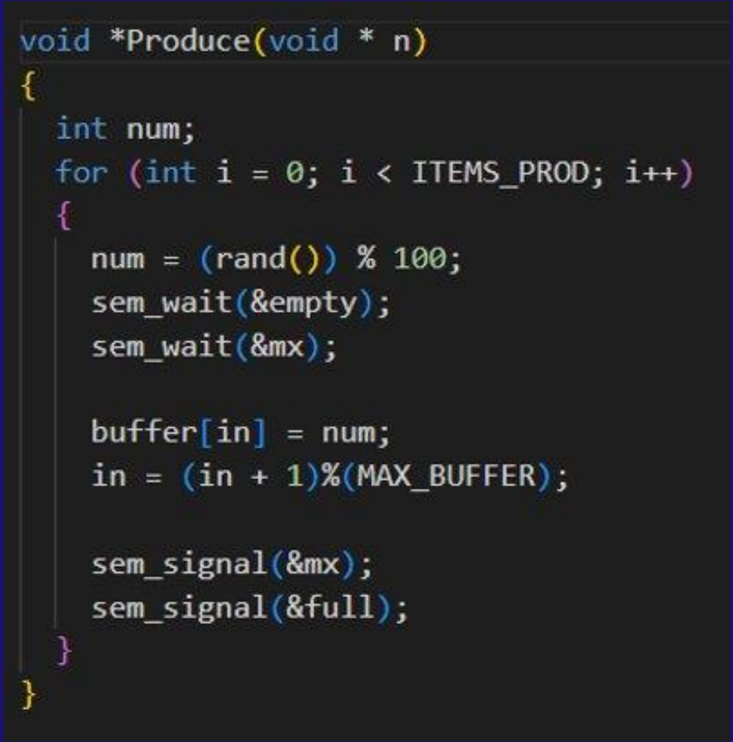
Output



```
Producer 0 has produced item 7 at buffer number 0
Producer 1 has produced item 49 at buffer number 1
Consumer 0 has consumed item 49 at buffer number 1
Producer 2 has produced item 58 at buffer number 1
Producer 2 has produced item 44 at buffer number 2
Consumer 0 has consumed item 44 at buffer number 2
Consumer 3 has consumed item 58 at buffer number 1
Consumer 2 has consumed item 7 at buffer number 0
Producer 0 has produced item 73 at buffer number 0
Producer 0 has produced item 23 at buffer number 1
Producer 1 has produced item 72 at buffer number 2
Producer 1 has produced item 9 at buffer number 3
Consumer 4 has consumed item 9 at buffer number 3
Consumer 4 has consumed item 72 at buffer number 2
Producer 2 has produced item 78 at buffer number 2
Consumer 5 has consumed item 78 at buffer number 2
Consumer 5 has consumed item 23 at buffer number 1
Producer 3 has produced item 30 at buffer number 1
Producer 3 has produced item 40 at buffer number 2
Producer 3 has produced item 65 at buffer number 3
Consumer 1 has consumed item 65 at buffer number 3
Consumer 1 has consumed item 40 at buffer number 2
Consumer 2 has consumed item 30 at buffer number 1
Consumer 3 has consumed item 73 at buffer number 0
```

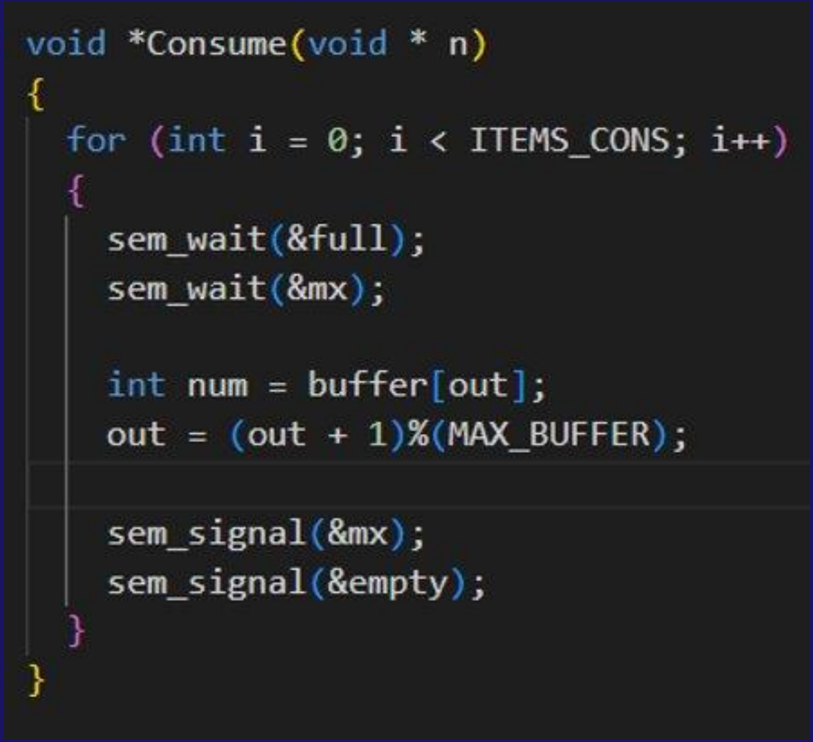
Cyclic Buffer

Producer



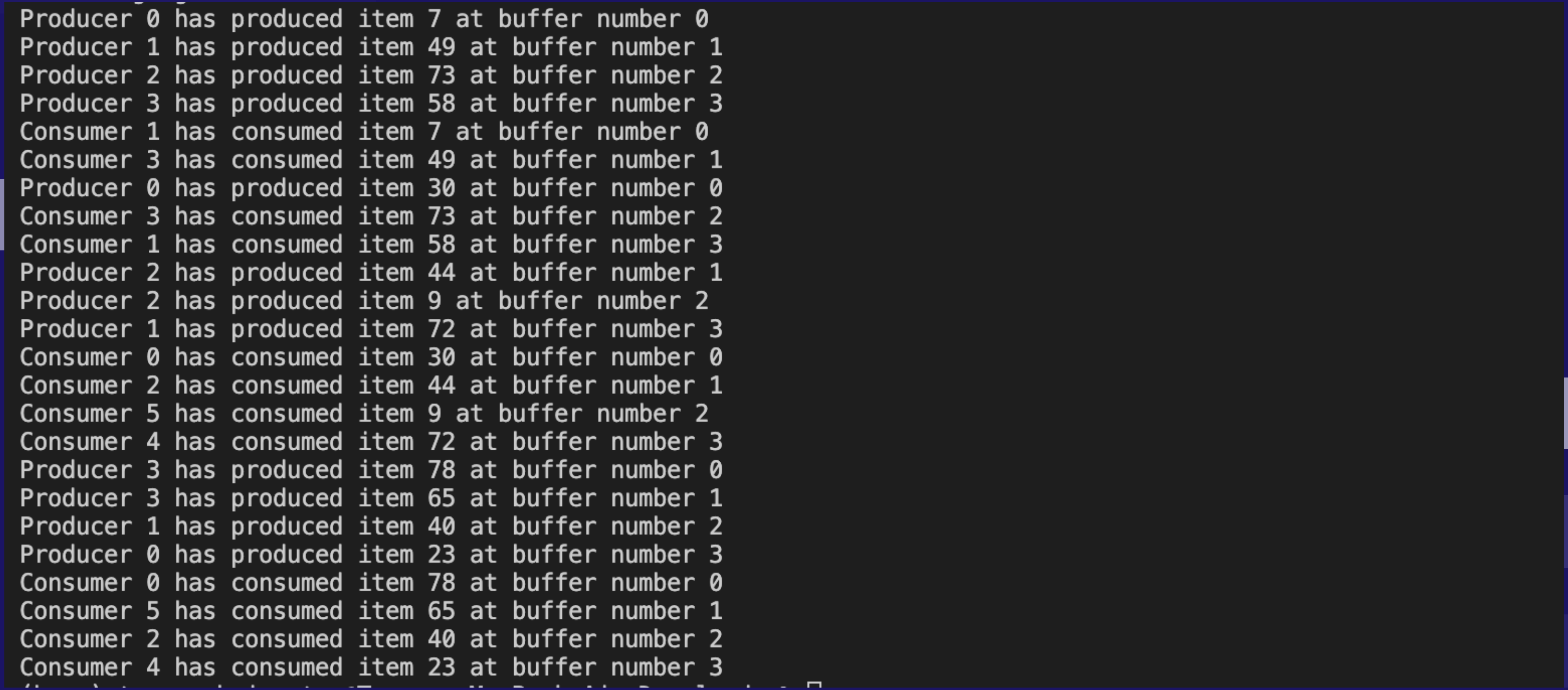
```c
void *Produce(void * n)
{
  int num;
  for (int i = 0; i < ITEMS_PROD; i++)
  {
    num = (rand()) % 100;
    sem_wait(&empty);
    sem_wait(&mx);

    buffer[in] = num;
    in = (in + 1)%(MAX_BUFFER);

    sem_signal(&mx);
    sem_signal(&full);
  }
}
```

Consumer



```c
void *Consume(void * n)
{
  for (int i = 0; i < ITEMS_CONS; i++)
  {
    sem_wait(&full);
    sem_wait(&mx);

    int num = buffer[out];
    out = (out + 1)%(MAX_BUFFER);


    sem_signal(&mx);
    sem_signal(&empty);
  }
}
```

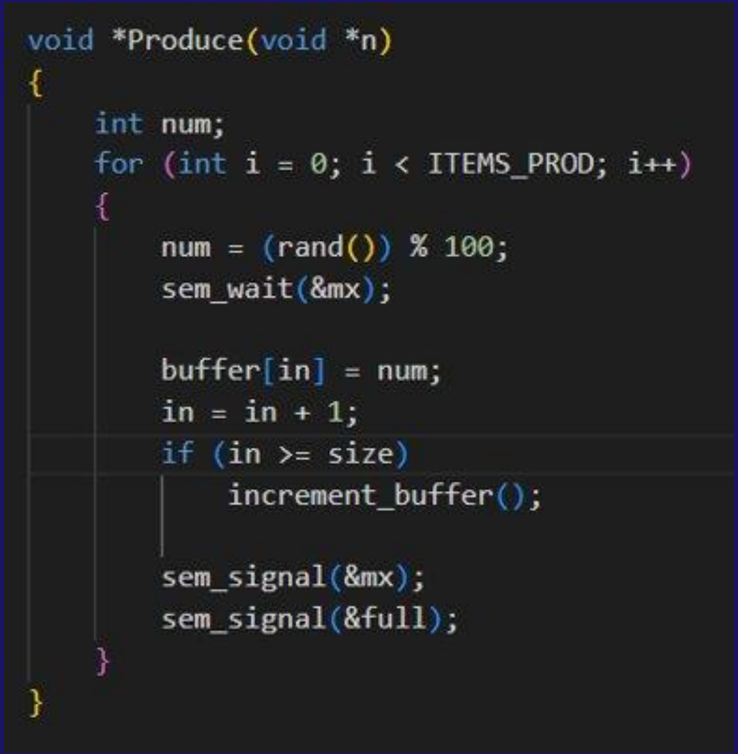
Output



```
Producer 0 has produced item 7 at buffer number 0
Producer 1 has produced item 49 at buffer number 1
Producer 2 has produced item 73 at buffer number 2
Producer 3 has produced item 58 at buffer number 3
Consumer 1 has consumed item 7 at buffer number 0
Consumer 3 has consumed item 49 at buffer number 1
Producer 0 has produced item 30 at buffer number 0
Consumer 3 has consumed item 73 at buffer number 2
Consumer 1 has consumed item 58 at buffer number 3
Producer 2 has produced item 44 at buffer number 1
Producer 2 has produced item 9 at buffer number 2
Producer 1 has produced item 72 at buffer number 3
Consumer 0 has consumed item 30 at buffer number 0
Consumer 2 has consumed item 44 at buffer number 1
Consumer 5 has consumed item 9 at buffer number 2
Consumer 4 has consumed item 72 at buffer number 3
Producer 3 has produced item 78 at buffer number 0
Producer 3 has produced item 65 at buffer number 1
Producer 1 has produced item 40 at buffer number 2
Producer 0 has produced item 23 at buffer number 3
Consumer 0 has consumed item 78 at buffer number 0
Consumer 5 has consumed item 65 at buffer number 1
Consumer 2 has consumed item 40 at buffer number 2
Consumer 4 has consumed item 23 at buffer number 3
```

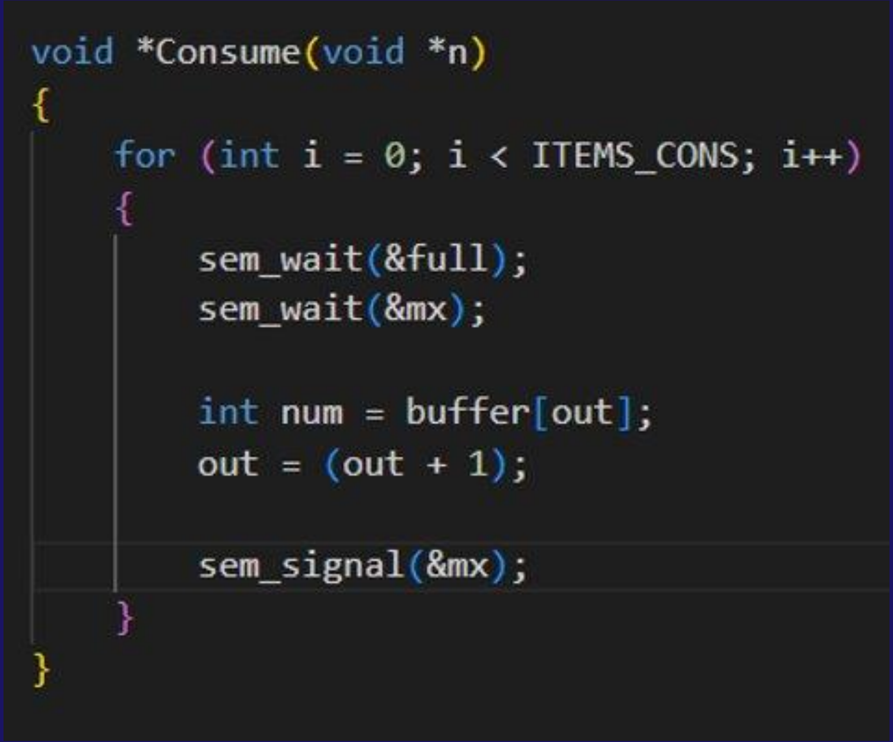
Infinite Buffer

Producer



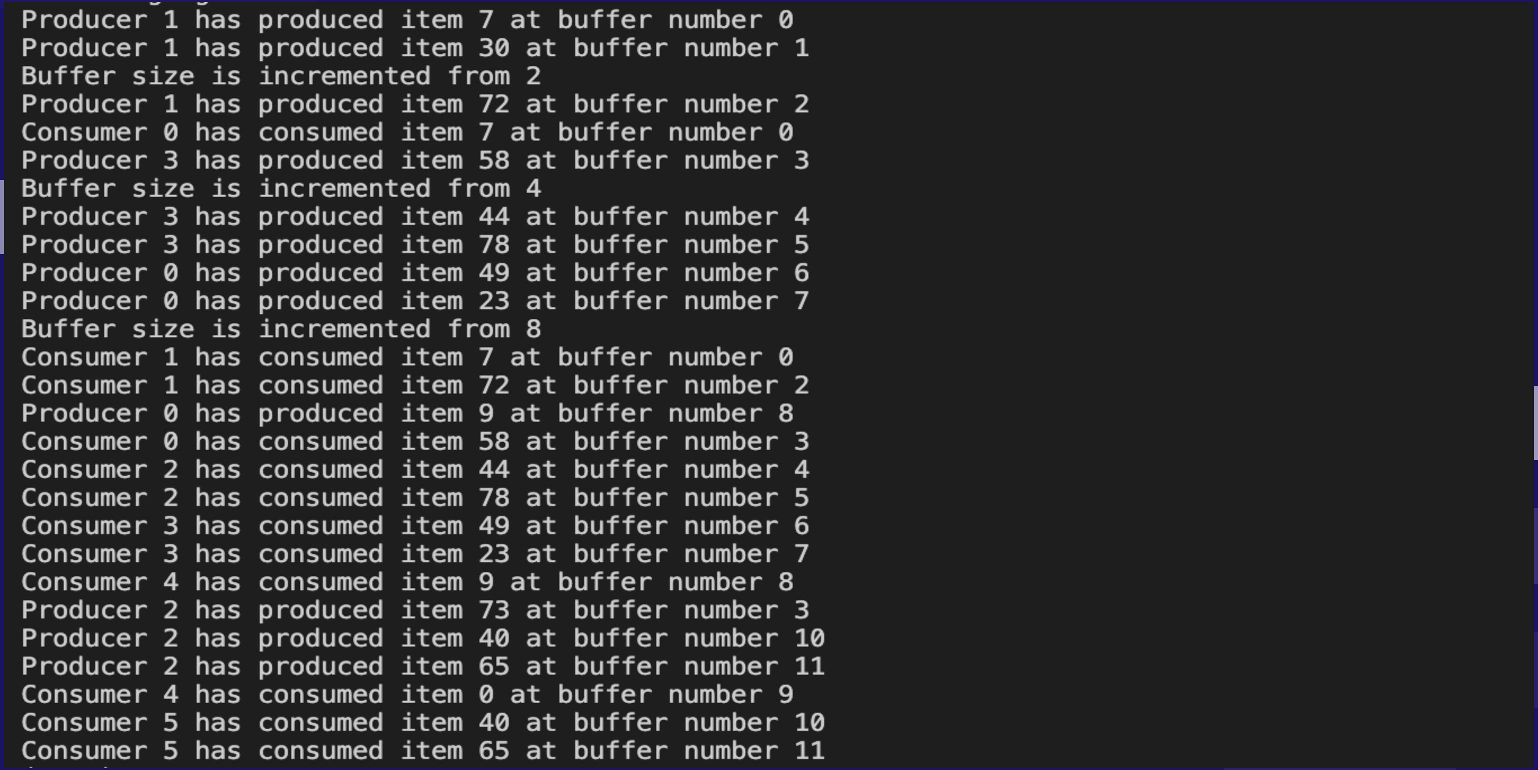
```c
void *Produce(void *n)
{
    int num;
    for (int i = 0; i < ITEMS_PROD; i++)
    {
        num = (rand()) % 100;
        sem_wait(&mx);

        buffer[in] = num;
        in = in + 1;
        if (in >= size)
            increment_buffer();

        sem_signal(&mx);
        sem_signal(&full);
    }
}
```

Consumer



```c
void *Consume(void *n)
{
    for (int i = 0; i < ITEMS_CONS; i++)
    {
        sem_wait(&full);
        sem_wait(&mx);

        int num = buffer[out];
        out = (out + 1);

        sem_signal(&mx);
    }
}
```

Output


```
Producer 1 has produced item 7 at buffer number 0
Producer 1 has produced item 30 at buffer number 1
Buffer size is incremented from 2
Producer 1 has produced item 72 at buffer number 2
Consumer 0 has consumed item 7 at buffer number 0
Producer 3 has produced item 58 at buffer number 3
Buffer size is incremented from 4
Producer 3 has produced item 44 at buffer number 4
Producer 3 has produced item 78 at buffer number 5
Producer 0 has produced item 49 at buffer number 6
Producer 0 has produced item 23 at buffer number 7
Buffer size is incremented from 8
Consumer 1 has consumed item 7 at buffer number 0
Consumer 1 has consumed item 72 at buffer number 2
Producer 0 has produced item 9 at buffer number 8
Consumer 0 has consumed item 58 at buffer number 3
Consumer 2 has consumed item 44 at buffer number 4
Consumer 2 has consumed item 78 at buffer number 5
Consumer 3 has consumed item 49 at buffer number 6
Consumer 3 has consumed item 23 at buffer number 7
Consumer 4 has consumed item 9 at buffer number 8
Producer 2 has produced item 73 at buffer number 3
Producer 2 has produced item 40 at buffer number 10
Producer 2 has produced item 65 at buffer number 11
Consumer 4 has consumed item 0 at buffer number 9
Consumer 5 has consumed item 40 at buffer number 10
Consumer 5 has consumed item 65 at buffer number 11
```

02

Readers Writers Problem

Problem is based on Synchronization of any number of reader and writer processes which operate on a single shared resource.

Problem Overview

The simulation should satisfy the following conditions:

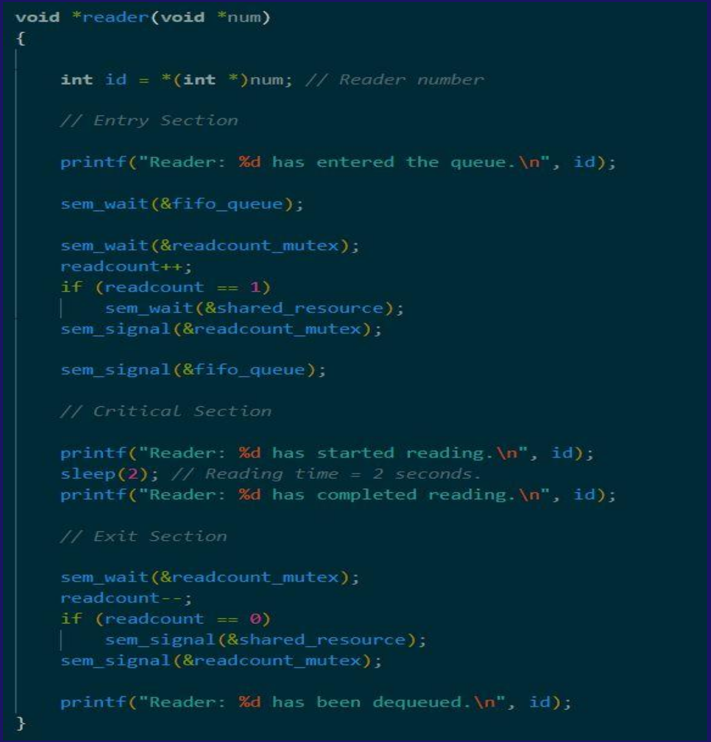
- ❑ Any number of readers should be able to read simultaneously.
- ❑ No two writers can execute simultaneously.
- ❑ Simultaneous reading and writing cannot happen.
- ❑ The system should not be idle at any time if some process is waiting.
- ❑ The system should be deadlock free.
- ❑ The system should be starve free.

Solution

We will be using the following parameters to solve the problem:

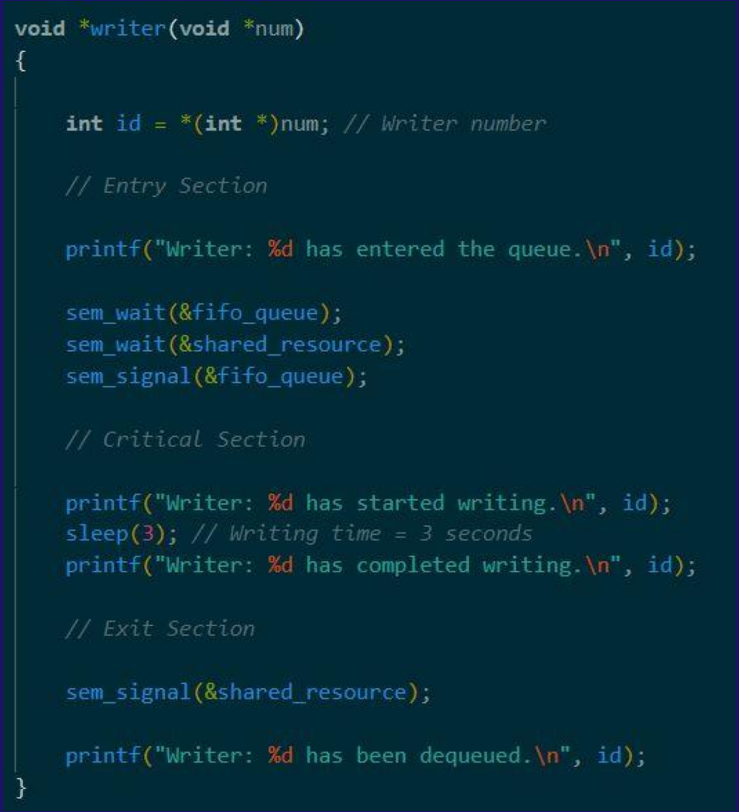
- ❑ fifo_queue: A binary semaphore to make the solution starve free by maintaining a queue of processes to execute them in FIFO order.
- ❑ shared_resource: A binary semaphore to achieve mutual exclusion among readers and writers, such that the given conditions are met.
- ❑ readcount: Number of readers currently reading.
- ❑ readcount_mutex: A mutex lock to avoid simultaneous access of readcount variable.

Reader Process



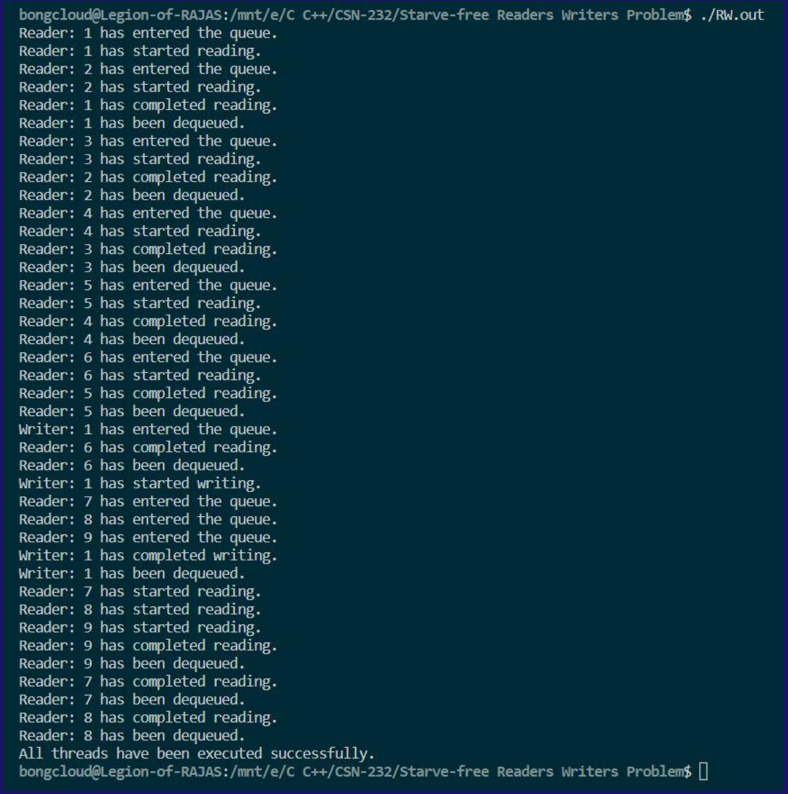
```c
void *reader(void *num)
{

    int id = *(int *)num; // Reader number

    // Entry Section

    printf("Reader: %d has entered the queue.\n", id);

    sem_wait(&fifo_queue);

    sem_wait(&readcount_mutex);
    readcount++;
    if (readcount == 1)
        sem_wait(&shared_resource);
    sem_signal(&readcount_mutex);

    sem_signal(&fifo_queue);

    // Critical Section

    printf("Reader: %d has started reading.\n", id);
    sleep(2); // Reading time = 2 seconds.
    printf("Reader: %d has completed reading.\n", id);

    // Exit Section

    sem_wait(&readcount_mutex);
    readcount--;
    if (readcount == 0)
        sem_signal(&shared_resource);
    sem_signal(&readcount_mutex);

    printf("Reader: %d has been dequeued.\n", id);
}
```

Writer Process



```c
void *writer(void *num)
{

    int id = *(int *)num; // Writer number

    // Entry Section

    printf("Writer: %d has entered the queue.\n", id);

    sem_wait(&fifo_queue);
    sem_wait(&shared_resource);
    sem_signal(&fifo_queue);

    // Critical Section

    printf("Writer: %d has started writing.\n", id);
    sleep(3); // Writing time = 3 seconds
    printf("Writer: %d has completed writing.\n", id);

    // Exit Section

    sem_signal(&shared_resource);

    printf("Writer: %d has been dequeued.\n", id);
}
```

Output Format



```
bongcloud@Legion-of-RAJAS:/mnt/e/C C++/CSN-232/Starve-free Readers Writers Problem$ ./RW.out
Reader: 1 has entered the queue.
Reader: 1 has started reading.
Reader: 2 has entered the queue.
Reader: 2 has started reading.
Reader: 1 has completed reading.
Reader: 1 has been dequeued.
Reader: 3 has entered the queue.
Reader: 3 has started reading.
Reader: 2 has completed reading.
Reader: 2 has been dequeued.
Reader: 4 has entered the queue.
Reader: 4 has started reading.
Reader: 3 has completed reading.
Reader: 3 has been dequeued.
Reader: 5 has entered the queue.
Reader: 5 has started reading.
Reader: 4 has completed reading.
Reader: 4 has been dequeued.
Reader: 6 has entered the queue.
Reader: 6 has started reading.
Reader: 5 has completed reading.
Reader: 5 has been dequeued.
Writer: 1 has entered the queue.
Reader: 6 has completed reading.
Reader: 6 has been dequeued.
Writer: 1 has started writing.
Reader: 7 has entered the queue.
Reader: 8 has entered the queue.
Reader: 9 has entered the queue.
Writer: 1 has completed writing.
Writer: 1 has been dequeued.
Reader: 7 has started reading.
Reader: 8 has started reading.
Reader: 9 has started reading.
Reader: 9 has completed reading.
Reader: 9 has been dequeued.
Reader: 7 has completed reading.
Reader: 7 has been dequeued.
Reader: 8 has completed reading.
Reader: 8 has been dequeued.
All threads have been executed successfully.
bongcloud@Legion-of-RAJAS:/mnt/e/C C++/CSN-232/Starve-free Readers Writers Problem$
```

03

Dining Philosophers

Problem is based on synchronization between number of philosophers sitting around a circular table and sharing chopsticks

Problem Overview

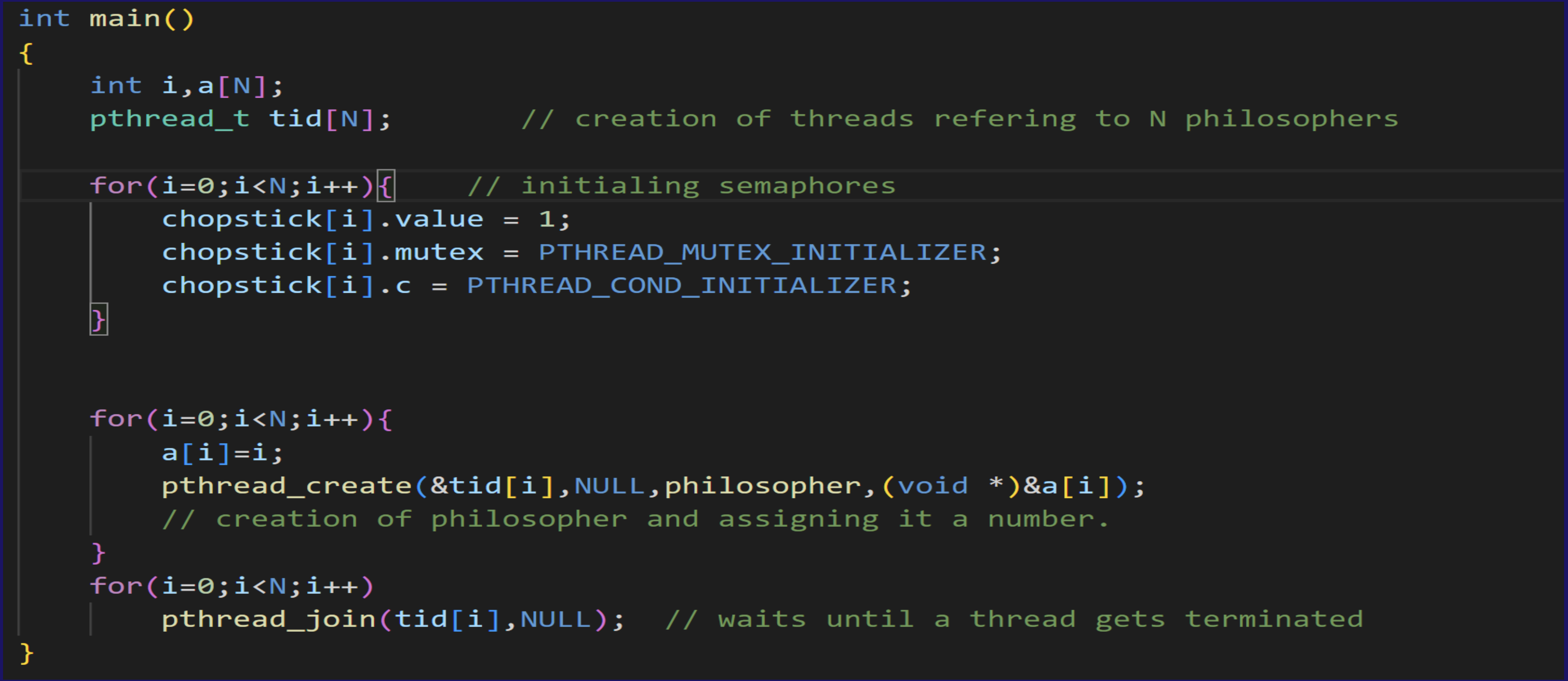
The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

Variables and Semaphores Used:

1. N = 5 // number of philosophers
2. THINKING = 2
3. HUNGRY = 1
4. EATING = 0
5. state[N] // state[I] - represents current state of philosophers
6. int phil // identifies a philosopher thread
7. int a[N] // philosopher number
8. chopsticks[N] = array of binary semaphores (indicates if a chopstick is picked up or put down)

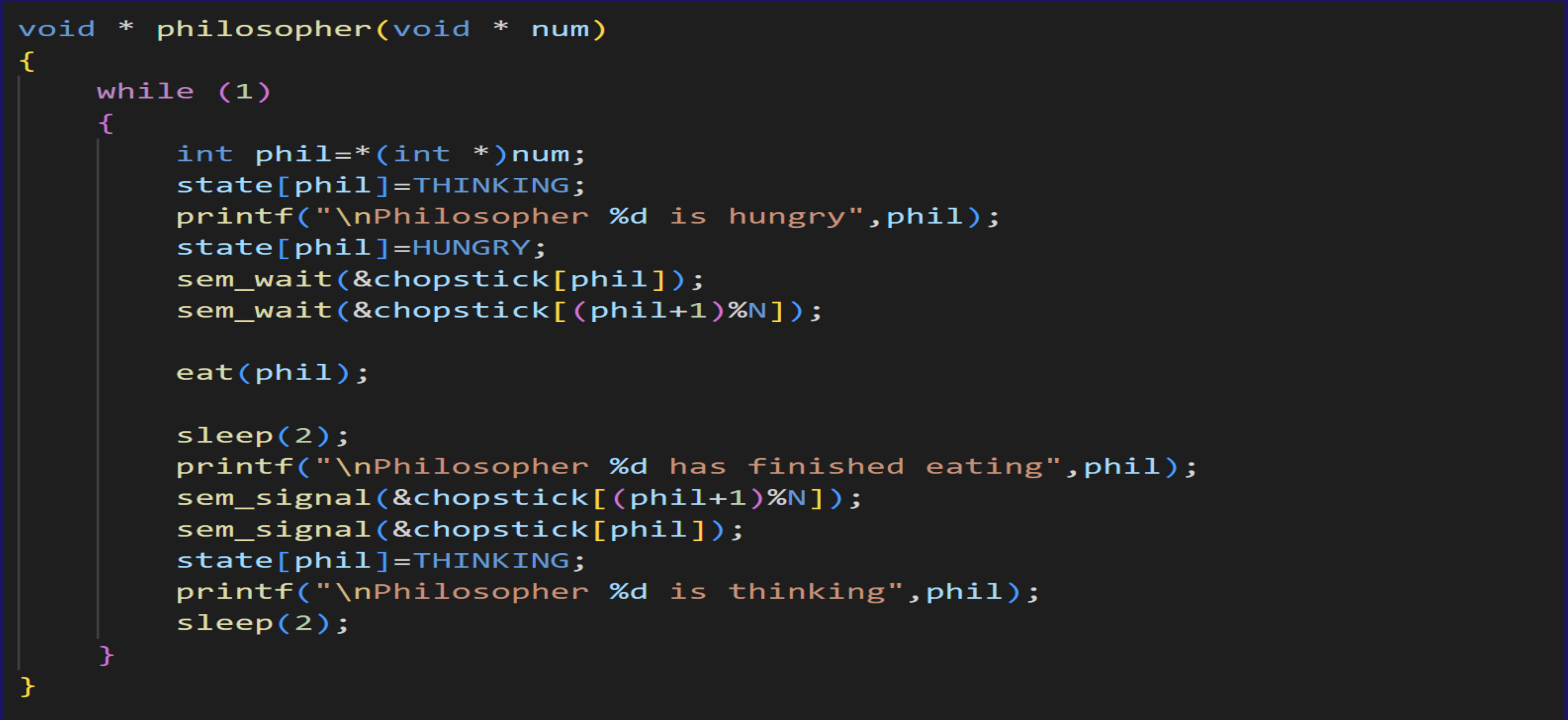
Solution:

int main()



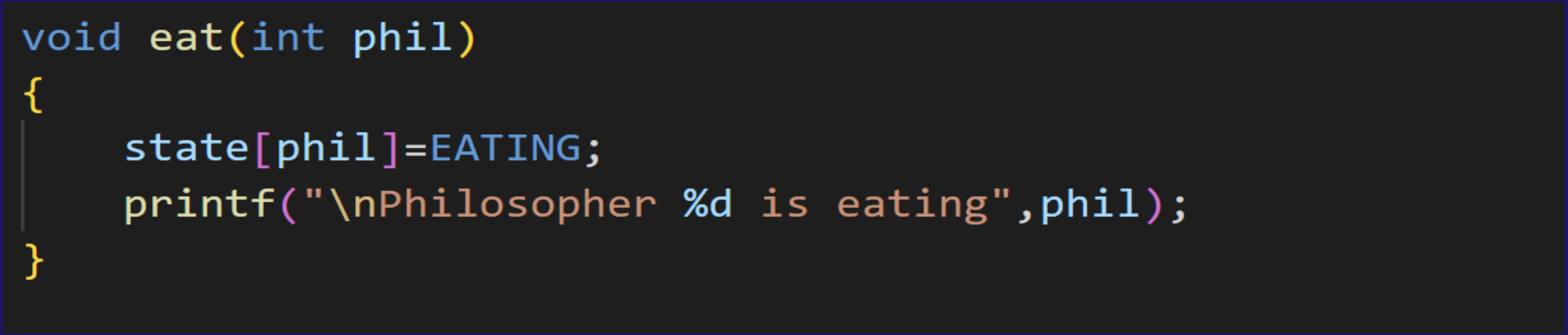
```c
int main()
{
    int i,a[N];
    pthread_t tid[N];        // creation of threads refering to N philosophers

    for(i=0;i<N;i++){     // initialing semaphores
        chopstick[i].value = 1;
        chopstick[i].mutex = PTHREAD_MUTEX_INITIALIZER;
        chopstick[i].c = PTHREAD_COND_INITIALIZER;
    }


    for(i=0;i<N;i++){
        a[i]=i;
        pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
        // creation of philosopher and assigning it a number.
    }
    for(i=0;i<N;i++)
        pthread_join(tid[i],NULL);  // waits until a thread gets terminated
}
```

void * philosopher()



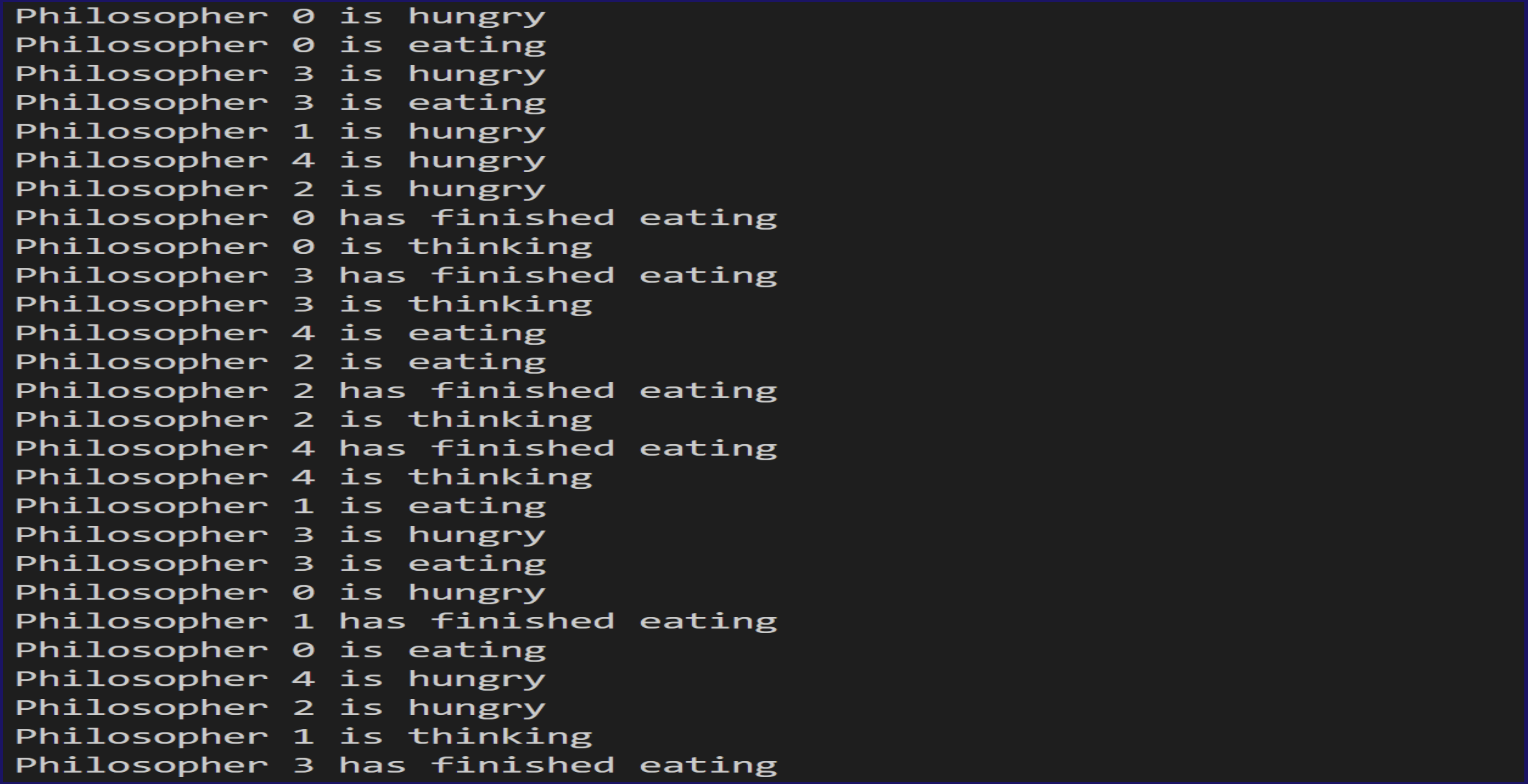
```
void * philosopher(void * num)
{
    while (1)
    {
        int phil=*(int *)num;
        state[phil]=THINKING;
        printf("\nPhilosopher %d is hungry",phil);
        state[phil]=HUNGRY;
        sem_wait(&chopstick[phil]);
        sem_wait(&chopstick[(phil+1)%N]);

        eat(phil);

        sleep(2);
        printf("\nPhilosopher %d has finished eating",phil);
        sem_signal(&chopstick[(phil+1)%N]);
        sem_signal(&chopstick[phil]);
        state[phil]=THINKING;
        printf("\nPhilosopher %d is thinking",phil);
        sleep(2);
    }
}
```

void eat()



```
void eat(int phil)
{
    state[phil]=EATING;
    printf("\nPhilosopher %d is eating",phil);
}
```

Output



```
Philosopher 0 is hungry
Philosopher 0 is eating
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 1 is hungry
Philosopher 4 is hungry
Philosopher 2 is hungry
Philosopher 0 has finished eating
Philosopher 0 is thinking
Philosopher 3 has finished eating
Philosopher 3 is thinking
Philosopher 4 is eating
Philosopher 2 is eating
Philosopher 2 has finished eating
Philosopher 2 is thinking
Philosopher 4 has finished eating
Philosopher 4 is thinking
Philosopher 1 is eating
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 0 is hungry
Philosopher 1 has finished eating
Philosopher 0 is eating
Philosopher 4 is hungry
Philosopher 2 is hungry
Philosopher 1 is thinking
Philosopher 3 has finished eating
```

04

SLEEPING BARBERS

The analogy is based on a hypothetical barber shop. There is a barber shop which has multiple barbers and n chairs for waiting for customers if there are any to sit on the chairs.

OVERVIEW

The challenge is to ensure that the barbers and customers are synchronized so that the following requirements are met:

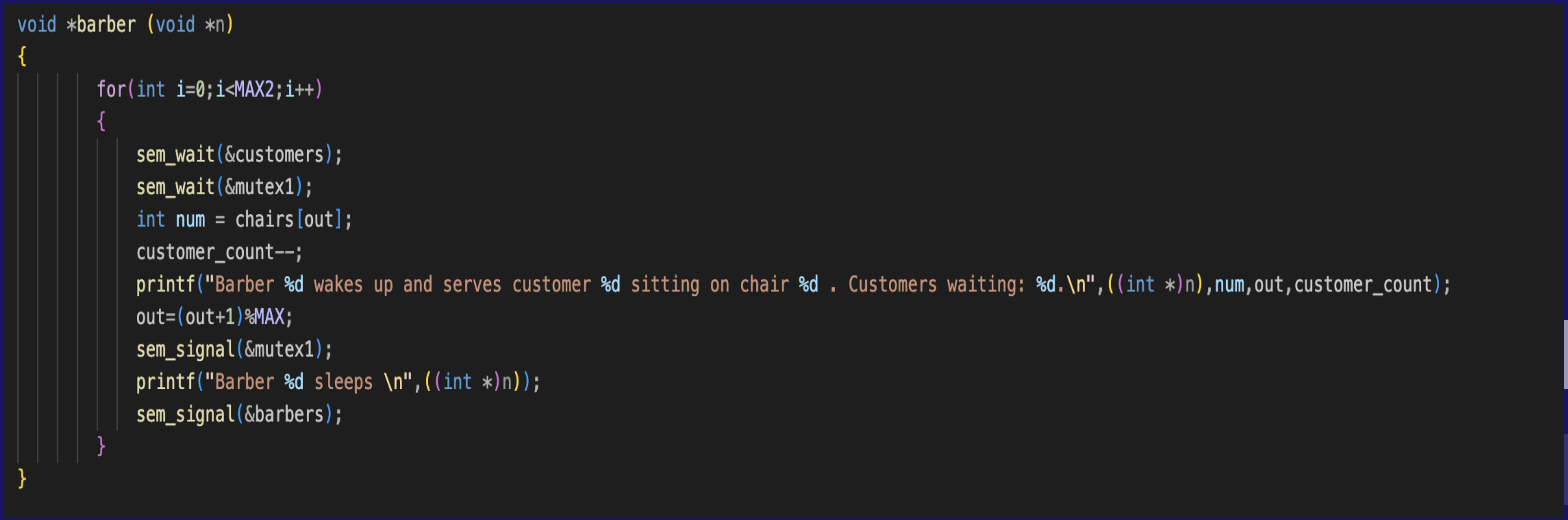
1. If there are no customers in the waiting room, the barber sleeps.
2. When a customer arrives, they should take a seat in the waiting room if there are chairs available or leave if there are no chairs.
3. If the waiting room is empty, the barber should not call a customer.
4. If there are customers in the waiting room, the barber should call the next customer and start cutting their hair.
5. Multiple barbers should be able to operate simultaneously without conflict or confusion.

Solution

This problem can be made deadlock and starvation free by using the following parameters

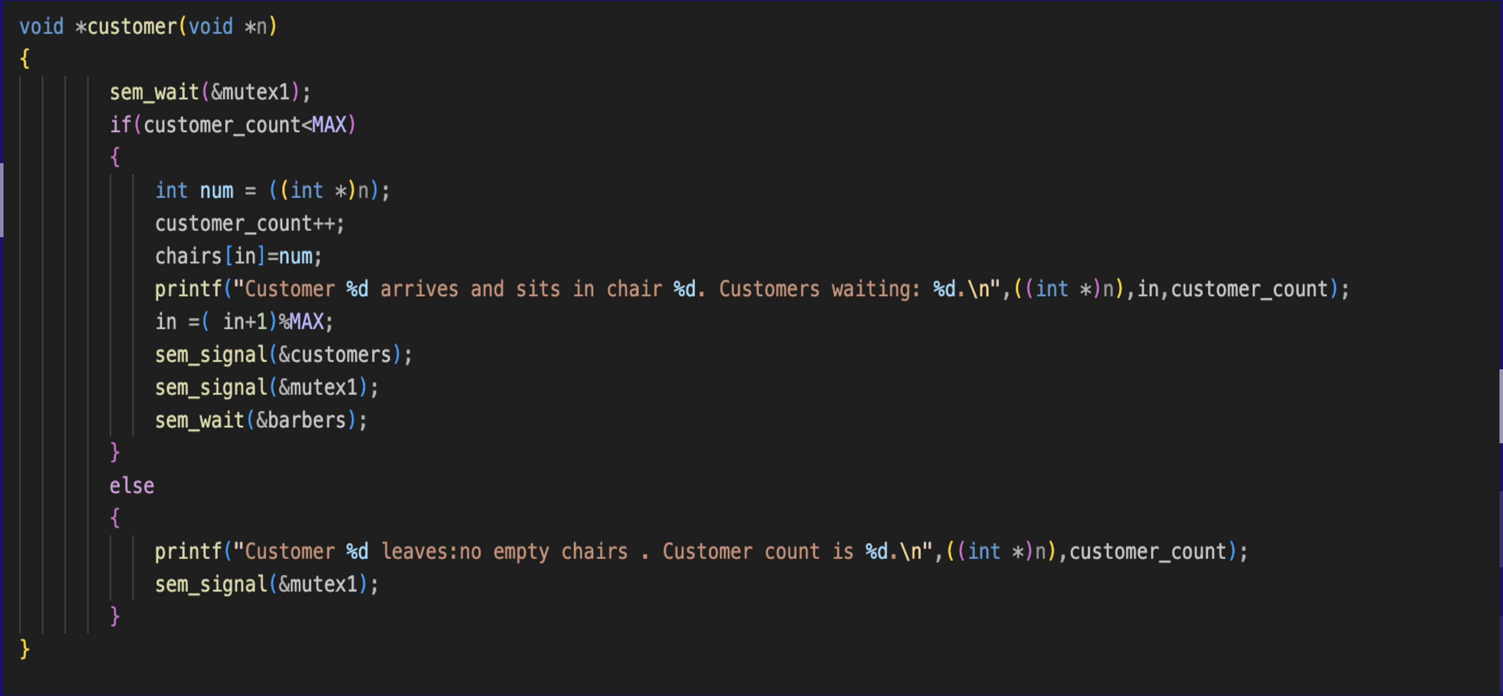
1. Barbers:- Counting semaphore that keeps the count of sleeping barbers initialized to the total number of barbers.
2. Customers:- Counting semaphore that keeps the count of current customers initialized to zero
3. Mutex1:- Binary lock to allow the mutual exclusion of shared variables
4. Customer_count:- keeps count of total customers inside the shop

Barber function



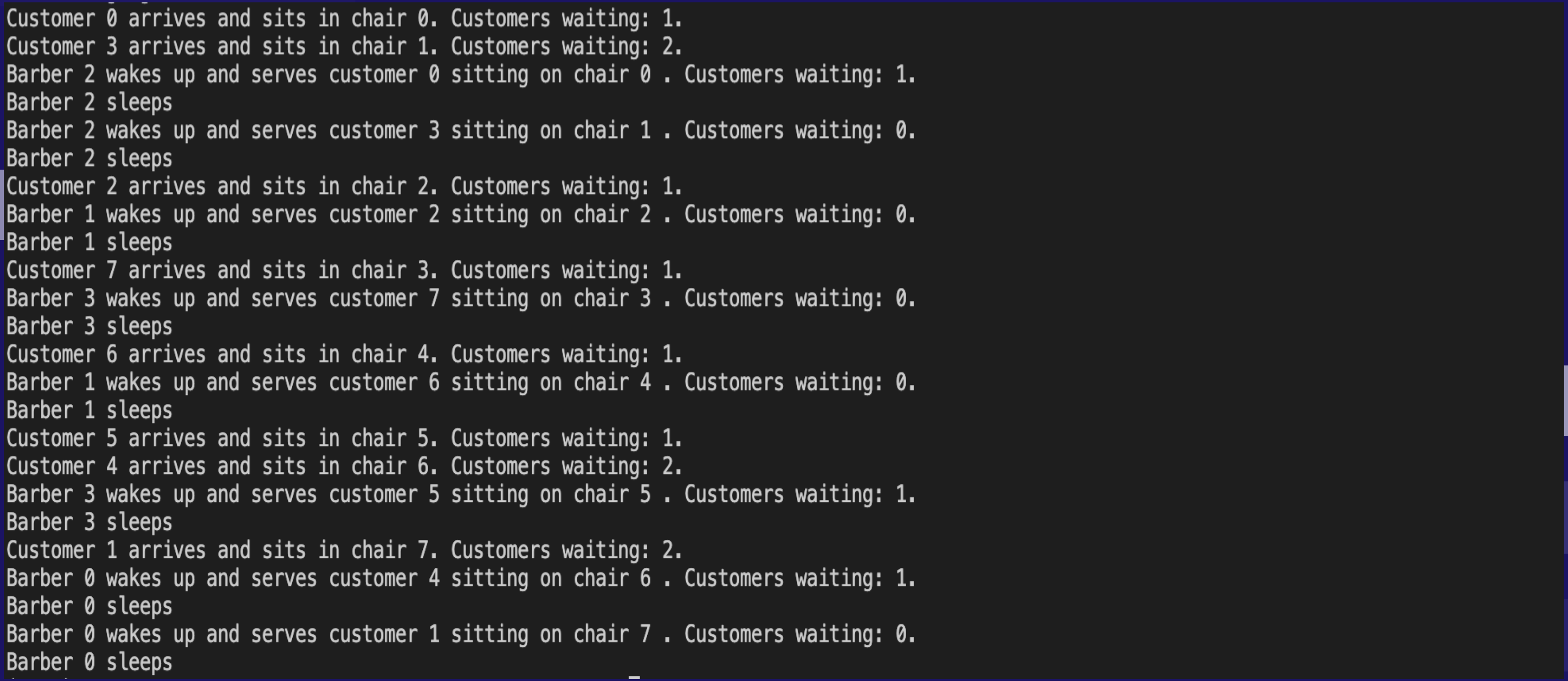
```c
void *barber (void *n)
{
        for(int i=0;i<MAX2;i++)
        {
            sem_wait(&customers);
            sem_wait(&mutex1);
            int num = chairs[out];
            customer_count--;
            printf("Barber %d wakes up and serves customer %d sitting on chair %d . Customers waiting: %d.\n",((int *)n),num,out,customer_count);
            out=(out+1)%MAX;
            sem_signal(&mutex1);
            printf("Barber %d sleeps \n",((int *)n));
            sem_signal(&barbers);
        }
}
```

Customer function



```
void *customer(void *n)
{
        sem_wait(&mutex1);
        if(customer_count<MAX)
        {
            int num = ((int *)n);
            customer_count++;
            chairs[in]=num;
            printf("Customer %d arrives and sits in chair %d. Customers waiting: %d.\n",((int *)n),in,customer_count);
            in =( in+1)%MAX;
            sem_signal(&customers);
            sem_signal(&mutex1);
            sem_wait(&barbers);
        }
        else
        {
            printf("Customer %d leaves:no empty chairs . Customer count is %d.\n",((int *)n),customer_count);
            sem_signal(&mutex1);
        }
}
```

Output



```
Customer 0 arrives and sits in chair 0. Customers waiting: 1.
Customer 3 arrives and sits in chair 1. Customers waiting: 2.
Barber 2 wakes up and serves customer 0 sitting on chair 0 . Customers waiting: 1.
Barber 2 sleeps
Barber 2 wakes up and serves customer 3 sitting on chair 1 . Customers waiting: 0.
Barber 2 sleeps
Customer 2 arrives and sits in chair 2. Customers waiting: 1.
Barber 1 wakes up and serves customer 2 sitting on chair 2 . Customers waiting: 0.
Barber 1 sleeps
Customer 7 arrives and sits in chair 3. Customers waiting: 1.
Barber 3 wakes up and serves customer 7 sitting on chair 3 . Customers waiting: 0.
Barber 3 sleeps
Customer 6 arrives and sits in chair 4. Customers waiting: 1.
Barber 1 wakes up and serves customer 6 sitting on chair 4 . Customers waiting: 0.
Barber 1 sleeps
Customer 5 arrives and sits in chair 5. Customers waiting: 1.
Customer 4 arrives and sits in chair 6. Customers waiting: 2.
Barber 3 wakes up and serves customer 5 sitting on chair 5 . Customers waiting: 1.
Barber 3 sleeps
Customer 1 arrives and sits in chair 7. Customers waiting: 2.
Barber 0 wakes up and serves customer 4 sitting on chair 6 . Customers waiting: 1.
Barber 0 sleeps
Barber 0 wakes up and serves customer 1 sitting on chair 7 . Customers waiting: 0.
Barber 0 sleeps
```

05

CIGARETTE SMOKER PROBLEM

*smoking is injurious to health

The problem is often used to illustrate the issues related to concurrent execution of multiple processes/threads and the need for synchronization mechanisms.

OVERVIEW

The cigarette smokers problem is a classic synchronization problem in Operating Systems.

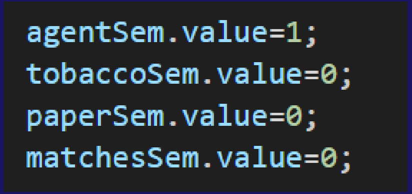
1. The problem involves an agent and three smokers who share a table.
2. The agent randomly places two different ingredients on the table.
3. Each smoker has an infinite supply of one of the remaining ingredients.
4. The smoker with the missing ingredient must wait until the agent places the required ingredients on the table.
5. Once the ingredients are placed, the smoker picks them up, makes a cigarette, and smokes it.
6. The problem is to design a solution that ensures only one smoker can smoke at a time, and that the agent must wait until the previous smoker has finished before placing new ingredients on the table.

Solution

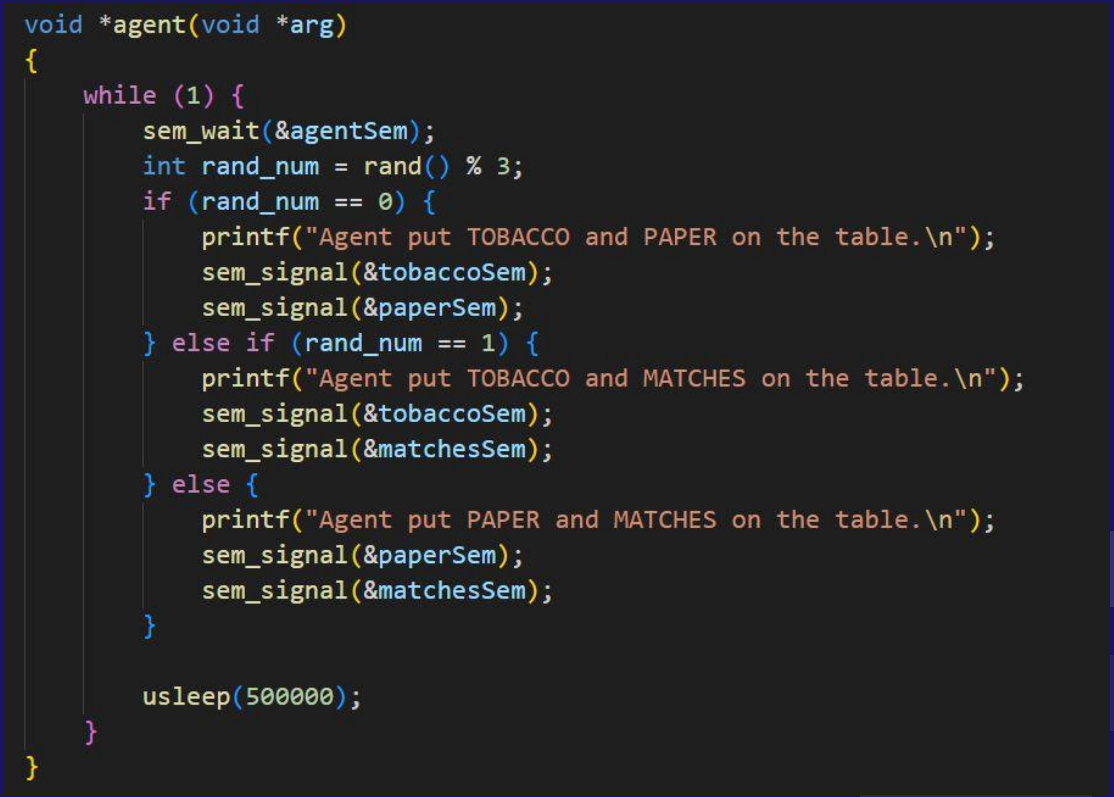
This problem can be made deadlock and starvation free by using the following parameters

1. Use semaphores to represent the different ingredients and a mutex to represent the smoking process.
2. The agent randomly chooses two ingredients and signals the corresponding semaphores.
3. Each smoker waits for the appropriate semaphore to be signalled and acquires the mutex to ensure that only one smoker is smoking at a time.
4. The smoker makes a cigarette and signals the smoking semaphore to allow the next smoker to smoke.
5. The smoker signals the appropriate agent semaphore to allow the agent to place new ingredients on the table.
6. The agent waits on a mutex to ensure that it does not place new ingredients on the table until the previous smoker has finished smoking.

Agent Function

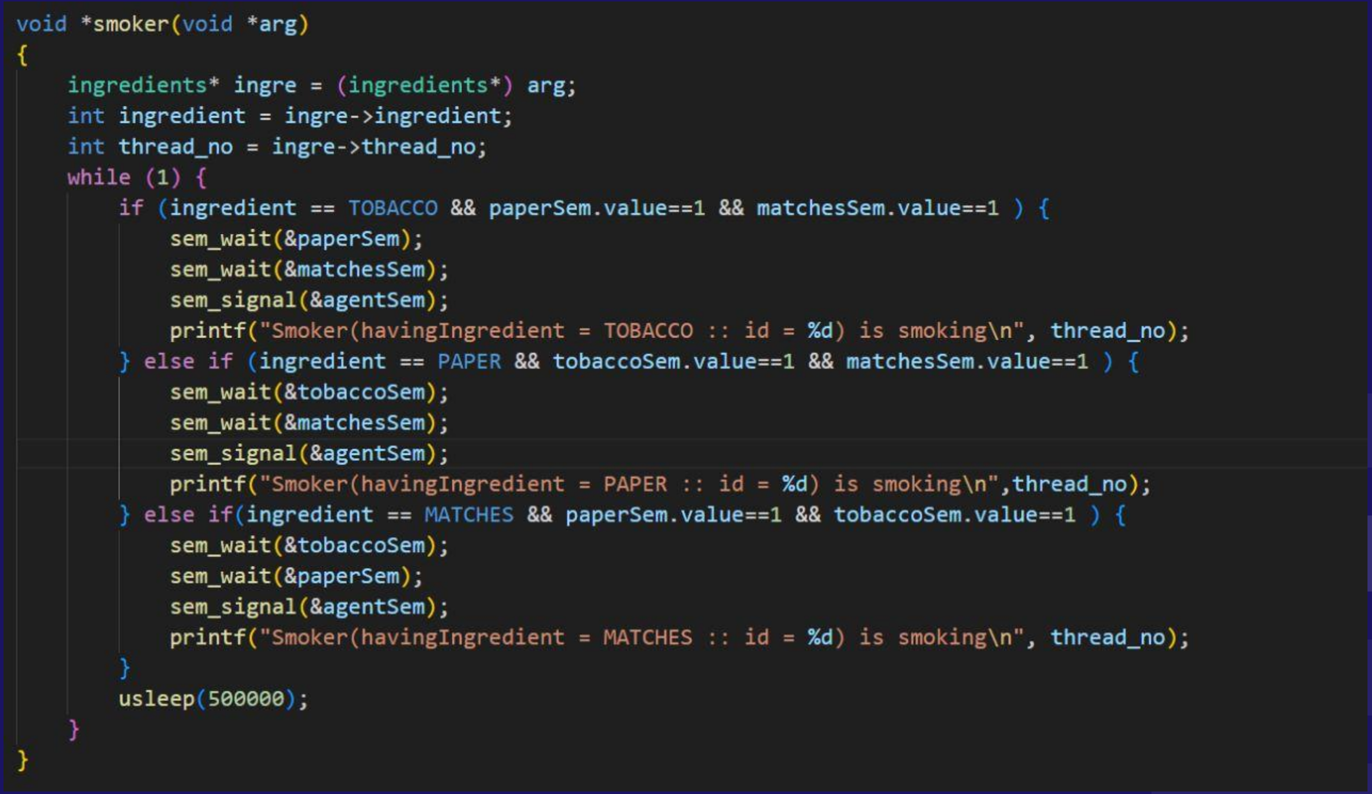


```c
agentSem.value=1;
tobaccoSem.value=0;
paperSem.value=0;
matchesSem.value=0;
```



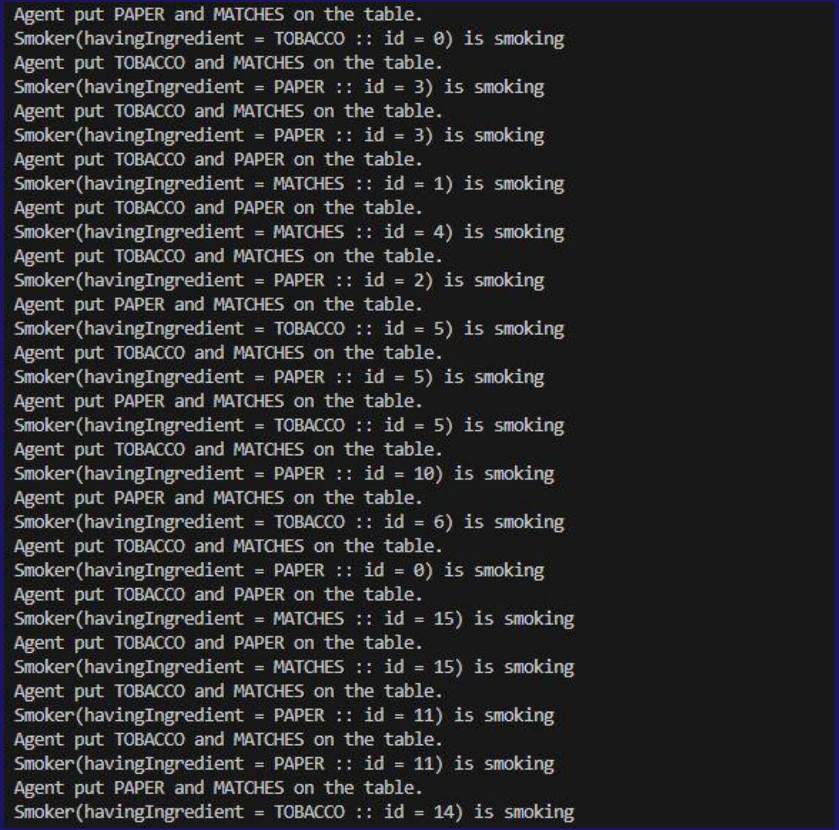
```c
void *agent(void *arg)
{
    while (1) {
        sem_wait(&agentSem);
        int rand_num = rand() % 3;
        if (rand_num == 0) {
            printf("Agent put TOBACCO and PAPER on the table.\n");
            sem_signal(&tobaccoSem);
            sem_signal(&paperSem);
        } else if (rand_num == 1) {
            printf("Agent put TOBACCO and MATCHES on the table.\n");
            sem_signal(&tobaccoSem);
            sem_signal(&matchesSem);
        } else {
            printf("Agent put PAPER and MATCHES on the table.\n");
            sem_signal(&paperSem);
            sem_signal(&matchesSem);
        }

        usleep(500000);
    }
}
```

Smoker Function



```c
void *smoker(void *arg)
{
    ingredients* ingre = (ingredients*) arg;
    int ingredient = ingre->ingredient;
    int thread_no = ingre->thread_no;
    while (1) {
        if (ingredient == TOBACCO && paperSem.value==1 && matchesSem.value==1 ) {
            sem_wait(&paperSem);
            sem_wait(&matchesSem);
            sem_signal(&agentSem);
            printf("Smoker(havingIngredient = TOBACCO :: id = %d) is smoking\n", thread_no);
        } else if (ingredient == PAPER && tobaccoSem.value==1 && matchesSem.value==1 ) {
            sem_wait(&tobaccoSem);
            sem_wait(&matchesSem);
            sem_signal(&agentSem);
            printf("Smoker(havingIngredient = PAPER :: id = %d) is smoking\n",thread_no);
        } else if(ingredient == MATCHES && paperSem.value==1 && tobaccoSem.value==1 ) {
            sem_wait(&tobaccoSem);
            sem_wait(&paperSem);
            sem_signal(&agentSem);
            printf("Smoker(havingIngredient = MATCHES :: id = %d) is smoking\n", thread_no);
        }
        usleep(500000);
    }
}
```

Output



```
Agent put PAPER and MATCHES on the table.
Smoker(havingIngredient = TOBACCO :: id = 0) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 3) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 3) is smoking
Agent put TOBACCO and PAPER on the table.
Smoker(havingIngredient = MATCHES :: id = 1) is smoking
Agent put TOBACCO and PAPER on the table.
Smoker(havingIngredient = MATCHES :: id = 4) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 2) is smoking
Agent put PAPER and MATCHES on the table.
Smoker(havingIngredient = TOBACCO :: id = 5) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 5) is smoking
Agent put PAPER and MATCHES on the table.
Smoker(havingIngredient = TOBACCO :: id = 5) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 10) is smoking
Agent put PAPER and MATCHES on the table.
Smoker(havingIngredient = TOBACCO :: id = 6) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 0) is smoking
Agent put TOBACCO and PAPER on the table.
Smoker(havingIngredient = MATCHES :: id = 15) is smoking
Agent put TOBACCO and PAPER on the table.
Smoker(havingIngredient = MATCHES :: id = 15) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 11) is smoking
Agent put TOBACCO and MATCHES on the table.
Smoker(havingIngredient = PAPER :: id = 11) is smoking
Agent put PAPER and MATCHES on the table.
Smoker(havingIngredient = TOBACCO :: id = 14) is smoking
```

06

River Crossing

Problem Statement:

There is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe. As each thread boards the boat it should invoke a function called board. You must guarantee that all four threads from each boatload invoke board before any of the threads from the next boatload do. After all four threads have invoked board, exactly one of them should call a function named rowBoat, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does. Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.

Variables and Semaphores Used:

Here are the global variables used in the solution:

```
barrier = Semaphore (4)
mutex = Semaphore (1)
int hackers = 0
int serfs = 0
hackerQueue = Semaphore (0)
serfQueue = Semaphore (0)
```

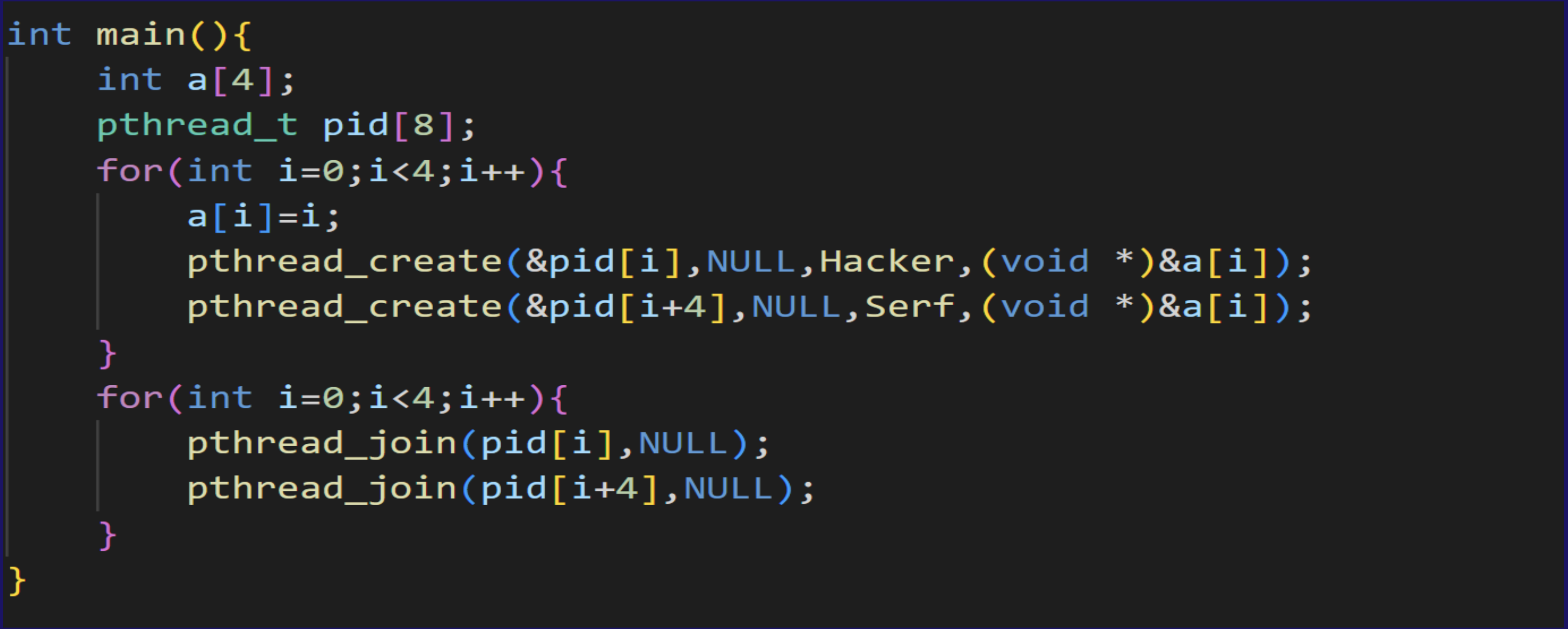
The local variable used in Hackers() and Serfs() functions:

```
local bool isCaptain = False
```

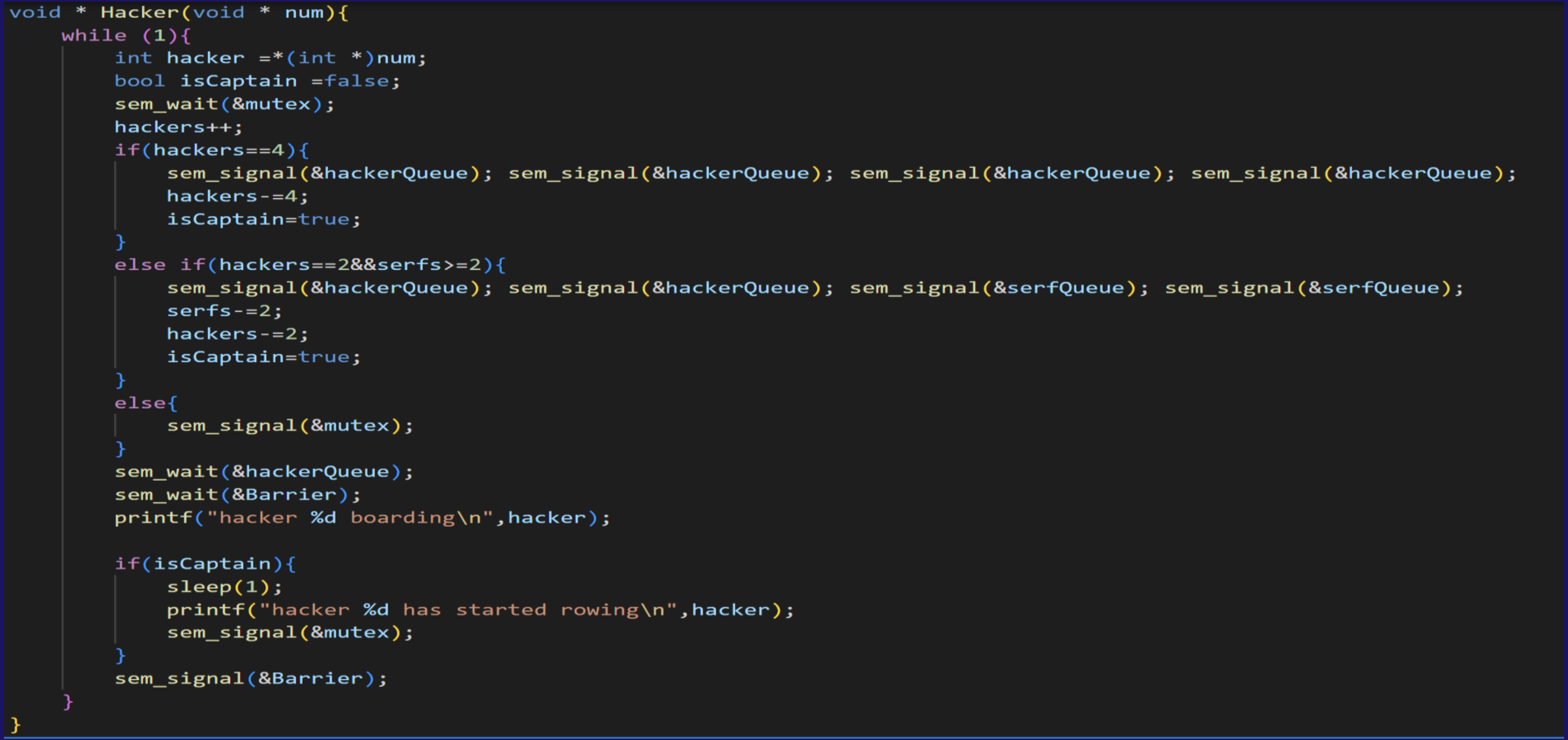
hackers and serfs count the number of hackers and serfs waiting to board. Since they are both protected by mutex, we can check the condition of both variables without worrying about an untimely update.
hackerQueue and serfQueue allow us to control the number of hackers and serfs that pass.
The barrier makes sure that all four threads have invoked board before the captain invokes rowBoat.
isCaptain is a local variable that indicates which thread should invoke row

Solution:

Int main() -

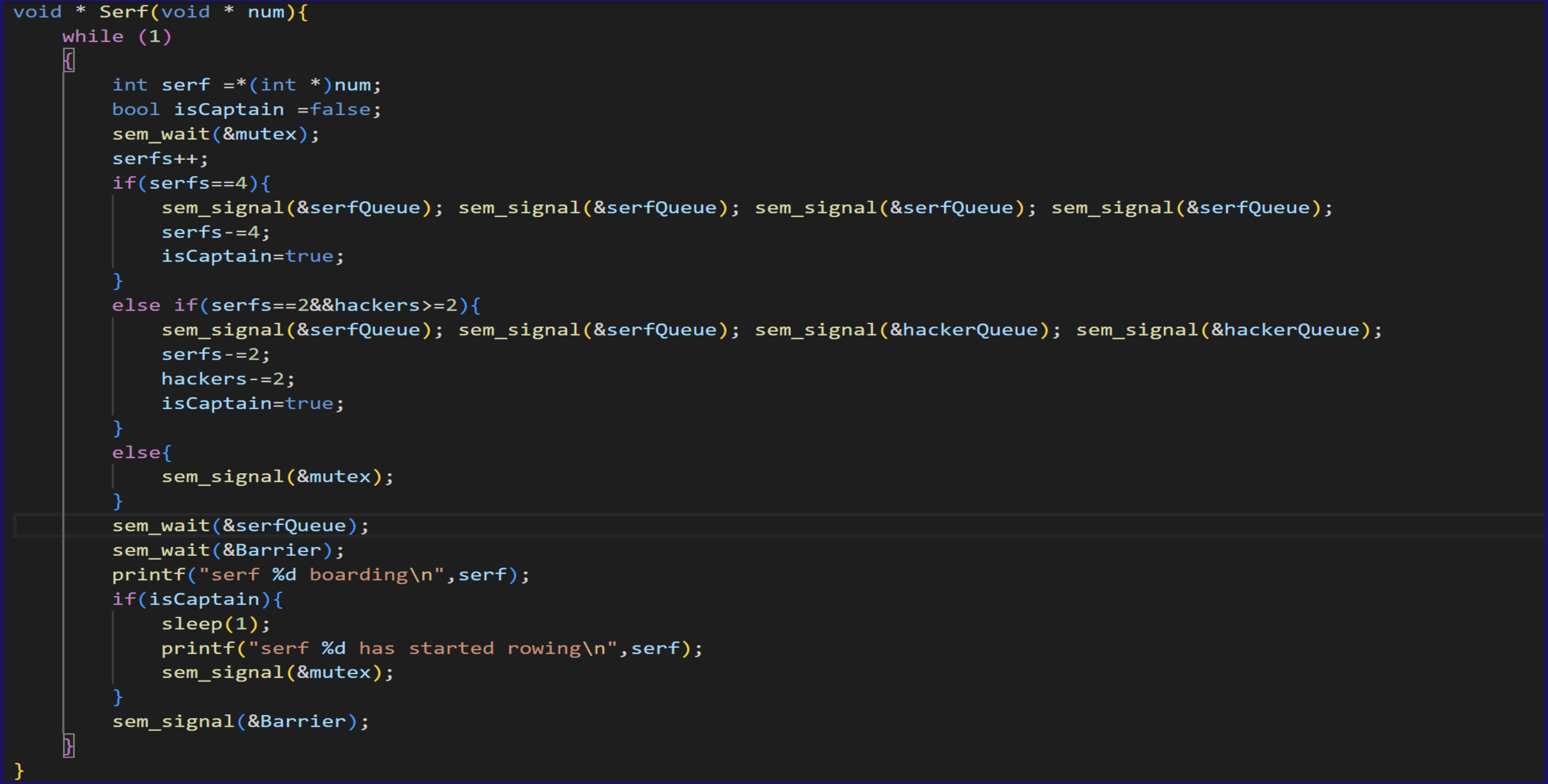

```
int main(){
    int a[4];
    pthread_t pid[8];
    for(int i=0;i<4;i++){
        a[i]=i;
        pthread_create(&pid[i],NULL,Hacker,(void *)&a[i]);
        pthread_create(&pid[i+4],NULL,Serf,(void *)&a[i]);
    }
    for(int i=0;i<4;i++){
        pthread_join(pid[i],NULL);
        pthread_join(pid[i+4],NULL);
    }
}
```

void* Hacker()



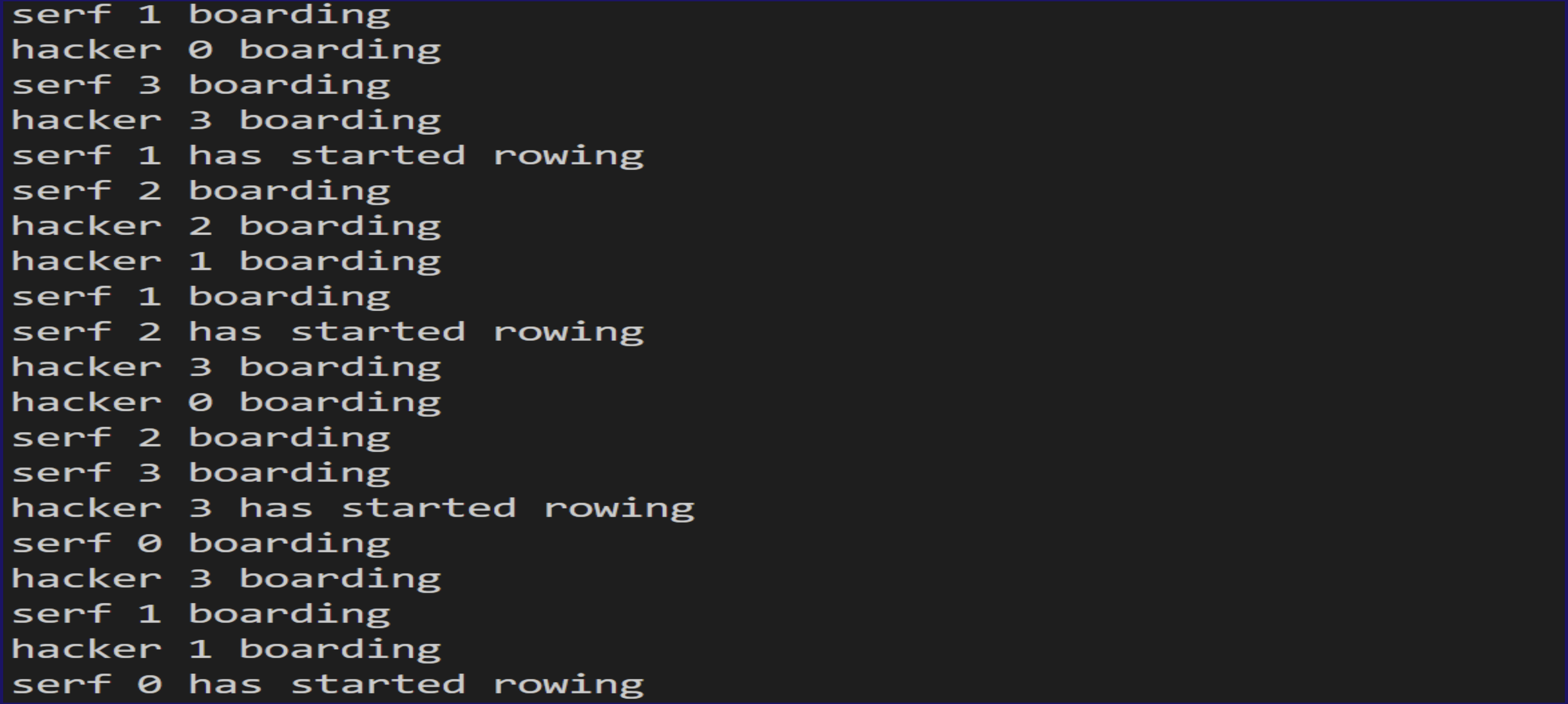
```c
void * Hacker(void * num){
    while (1){
        int hacker =*(int *)num;
        bool isCaptain =false;
        sem_wait(&mutex);
        hackers++;
        if(hackers==4){
            sem_signal(&hackerQueue); sem_signal(&hackerQueue); sem_signal(&hackerQueue); sem_signal(&hackerQueue);
            hackers-=4;
            isCaptain=true;
        }
        else if(hackers==2&&serfs>=2){
            sem_signal(&hackerQueue); sem_signal(&hackerQueue); sem_signal(&serfQueue); sem_signal(&serfQueue);
            serfs-=2;
            hackers-=2;
            isCaptain=true;
        }
        else{
            sem_signal(&mutex);
        }
        sem_wait(&hackerQueue);
        sem_wait(&Barrier);
        printf("hacker %d boarding\n",hacker);

        if(isCaptain){
            sleep(1);
            printf("hacker %d has started rowing\n",hacker);
            sem_signal(&mutex);
        }
        sem_signal(&Barrier);
    }
}
```

void * Serf()



```
void * Serf(void * num){
    while (1)
    {
        int serf =*(int *)num;
        bool isCaptain =false;
        sem_wait(&mutex);
        serfs++;
        if(serfs==4){
            sem_signal(&serfQueue); sem_signal(&serfQueue); sem_signal(&serfQueue); sem_signal(&serfQueue);
            serfs-=4;
            isCaptain=true;
        }
        else if(serfs==2&&hackers>=2){
            sem_signal(&serfQueue); sem_signal(&serfQueue); sem_signal(&hackerQueue); sem_signal(&hackerQueue);
            serfs-=2;
            hackers-=2;
            isCaptain=true;
        }
        else{
            sem_signal(&mutex);
        }
        sem_wait(&serfQueue);
        sem_wait(&Barrier);
        printf("serf %d boarding\n",serf);
        if(isCaptain){
            sleep(1);
            printf("serf %d has started rowing\n",serf);
            sem_signal(&mutex);
        }
        sem_signal(&Barrier);
    }
}
```

Output-



```
serf 1 boarding
hacker 0 boarding
serf 3 boarding
hacker 3 boarding
serf 1 has started rowing
serf 2 boarding
hacker 2 boarding
hacker 1 boarding
serf 1 boarding
serf 2 has started rowing
hacker 3 boarding
hacker 0 boarding
serf 2 boarding
serf 3 boarding
hacker 3 has started rowing
serf 0 boarding
hacker 3 boarding
serf 1 boarding
hacker 1 boarding
serf 0 has started rowing
```

07

H2O Problem

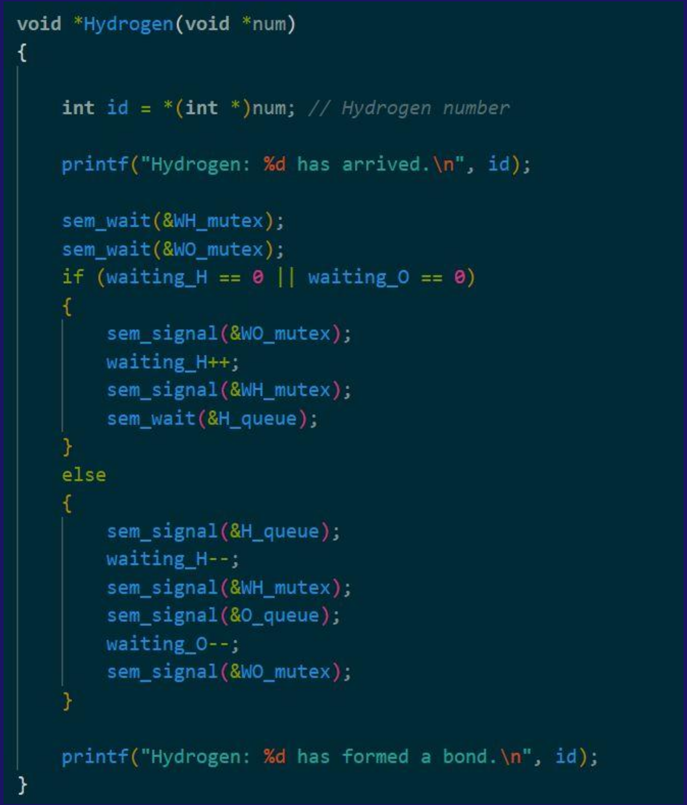
Hydrogen and oxygen atoms keep arriving randomly to form water molecules. Whenever there are two free hydrogen atoms and one free oxygen atom, we pair them up to form a water molecule. Simulate this situation.

Solution

We will use the following parameters to solve this problem:

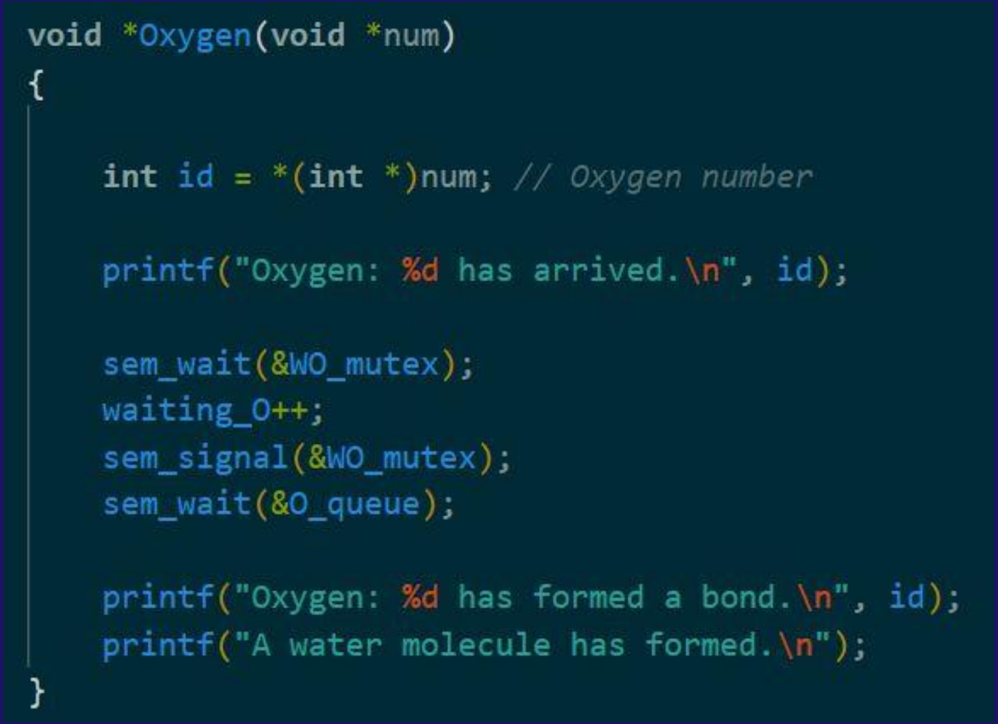
- H_queue: A binary semaphore to keep track of waiting Hydrogen atoms by maintaining a queue of them.
- O_queue: A binary semaphore to keep track of waiting Oxygen atoms by maintaining a queue of them.
- waiting_H: Number of waiting Hydrogen atoms.
- waiting_O: Number of waiting Oxygen atoms.
- WH_mutex: A mutex lock to avoid simultaneous access of waiting_H variable.
- WO_mutex: A mutex lock to avoid simultaneous access of waiting_O variable.

Hydrogen Process



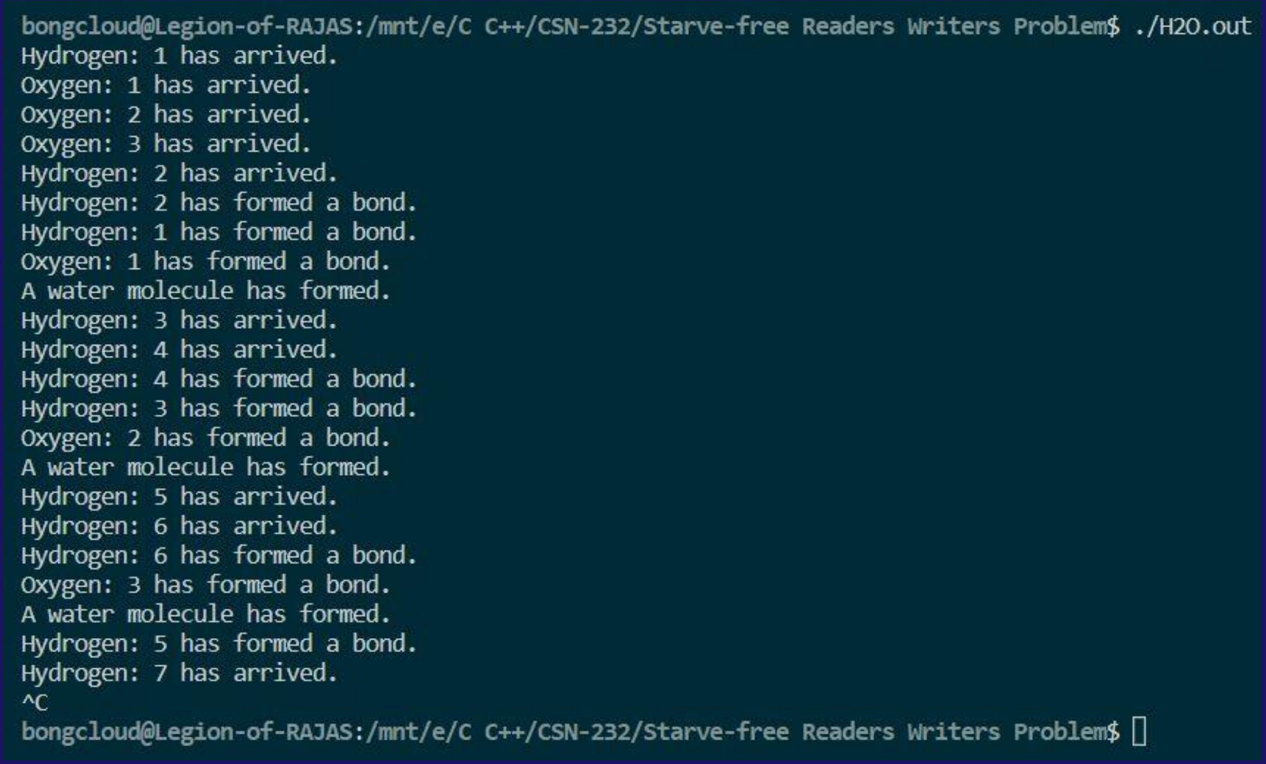
```c
void *Hydrogen(void *num)
{

    int id = *(int *)num; // Hydrogen number

    printf("Hydrogen: %d has arrived.\n", id);

    sem_wait(&WH_mutex);
    sem_wait(&WO_mutex);
    if (waiting_H == 0 || waiting_O == 0)
    {
        sem_signal(&WO_mutex);
        waiting_H++;
        sem_signal(&WH_mutex);
        sem_wait(&H_queue);
    }
    else
    {
        sem_signal(&H_queue);
        waiting_H--;
        sem_signal(&WH_mutex);
        sem_signal(&O_queue);
        waiting_O--;
        sem_signal(&WO_mutex);
    }

    printf("Hydrogen: %d has formed a bond.\n", id);
}
```

Oxygen Process



```c
void *Oxygen(void *num)
{

    int id = *(int *)num; // Oxygen number

    printf("Oxygen: %d has arrived.\n", id);

    sem_wait(&WO_mutex);
    waiting_O++;
    sem_signal(&WO_mutex);
    sem_wait(&O_queue);

    printf("Oxygen: %d has formed a bond.\n", id);
    printf("A water molecule has formed.\n");
}
```

Output Format



```
bongcloud@Legion-of-RAJAS:/mnt/e/C C++/CSN-232/Starve-free Readers Writers Problem$ ./H2O.out
Hydrogen: 1 has arrived.
Oxygen: 1 has arrived.
Oxygen: 2 has arrived.
Oxygen: 3 has arrived.
Hydrogen: 2 has arrived.
Hydrogen: 2 has formed a bond.
Hydrogen: 1 has formed a bond.
Oxygen: 1 has formed a bond.
A water molecule has formed.
Hydrogen: 3 has arrived.
Hydrogen: 4 has arrived.
Hydrogen: 4 has formed a bond.
Hydrogen: 3 has formed a bond.
Oxygen: 2 has formed a bond.
A water molecule has formed.
Hydrogen: 5 has arrived.
Hydrogen: 6 has arrived.
Hydrogen: 6 has formed a bond.
Oxygen: 3 has formed a bond.
A water molecule has formed.
Hydrogen: 5 has formed a bond.
Hydrogen: 7 has arrived.
^C
bongcloud@Legion-of-RAJAS:/mnt/e/C C++/CSN-232/Starve-free Readers Writers Problem$ 
```

Meet The Team

Ashutosh Pise	- Readers Writers, H2O Problem
Ashutosh Kumar	- Cigarette Smoker Problem
Kirtan Patel	- Dining Philosopher's, River Crossing Problem
Mudit Gupta	- Readers Writers, H2O Problem
Naqiyah Kagzi	- Semaphore implementation, Producer Consumer Problem
Rishi Kejriwal	- Dining Philosopher's, River Crossing Problem
Rohan Kalra	- Cigarette Smoker Problem
Sahil Safy	- Semaphore implementation, Producer Consumer Problem
Shreya Shivkumar	- The Sleeping Barber Problem
Tanmay Shrivastav	- The Sleeping Barber Problem

Thank You