ASSIGNMENT -7
CED18I063

1. Simulate the Producer Consumer code discussed in the class.
We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

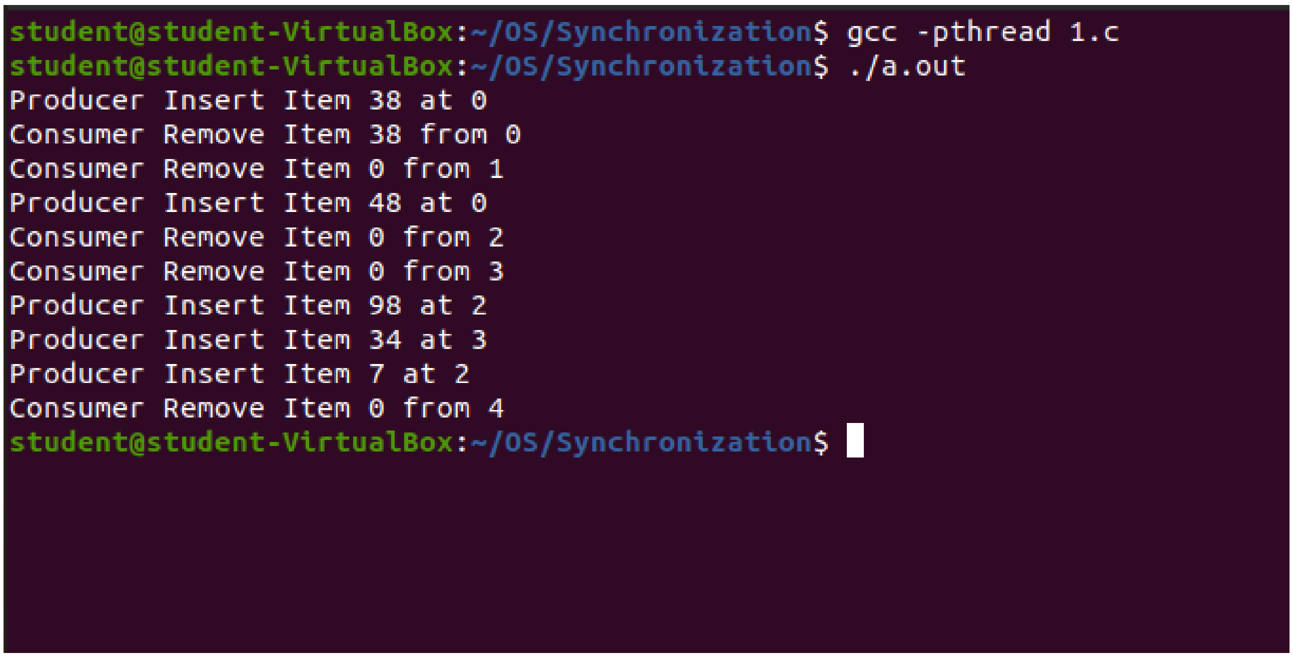
To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

Algo:
```
do{

//produce an item/consume

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

CODE:
```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include<time.h>
#define BufferSize 5 // Size of the buffer

int in = 0;
int out = 0;
int buffer[BufferSize];

void *producer(void *pno)
{
        int item;
        item = rand()%101; // Produce an random item
        buffer[in] = item;
        printf("Producer Insert Item %d at %d\n",buffer[in],in);
        in = (in+1)%BufferSize;

}

void *consumer(void *cno)
```

```
{

        int item = buffer[out];
        printf("Consumer Remove Item %d from %d\n",item, out);
        out = (out+1)%BufferSize;


}

int main()
{
    srand(time(0));
    pthread_t pro[5],con[5];
    for(int i = 0; i < 5; i++) {
        pthread_create(&pro[i], NULL, (void *)producer, NULL);
        pthread_create(&con[i], NULL, (void *)consumer, NULL);

    }

    for(int i = 0; i < 5; i++) {
        pthread_join(pro[i], NULL);
        pthread_join(con[i], NULL);
    }
    return 0;
}
```

SCREENSHOT:



```
student@student-VirtualBox:~/OS/Synchronization$ gcc -pthread 1.c
student@student-VirtualBox:~/OS/Synchronization$ ./a.out
Producer Insert Item 38 at 0
Consumer Remove Item 38 from 0
Consumer Remove Item 0 from 1
Producer Insert Item 48 at 0
Consumer Remove Item 0 from 2
Consumer Remove Item 0 from 3
Producer Insert Item 98 at 2
Producer Insert Item 34 at 3
Producer Insert Item 7 at 2
Consumer Remove Item 0 from 4
student@student-VirtualBox:~/OS/Synchronization$
```

2. Extend the producer consumer simulation in Q1 to sync access of critical data using Petersons algorithm.

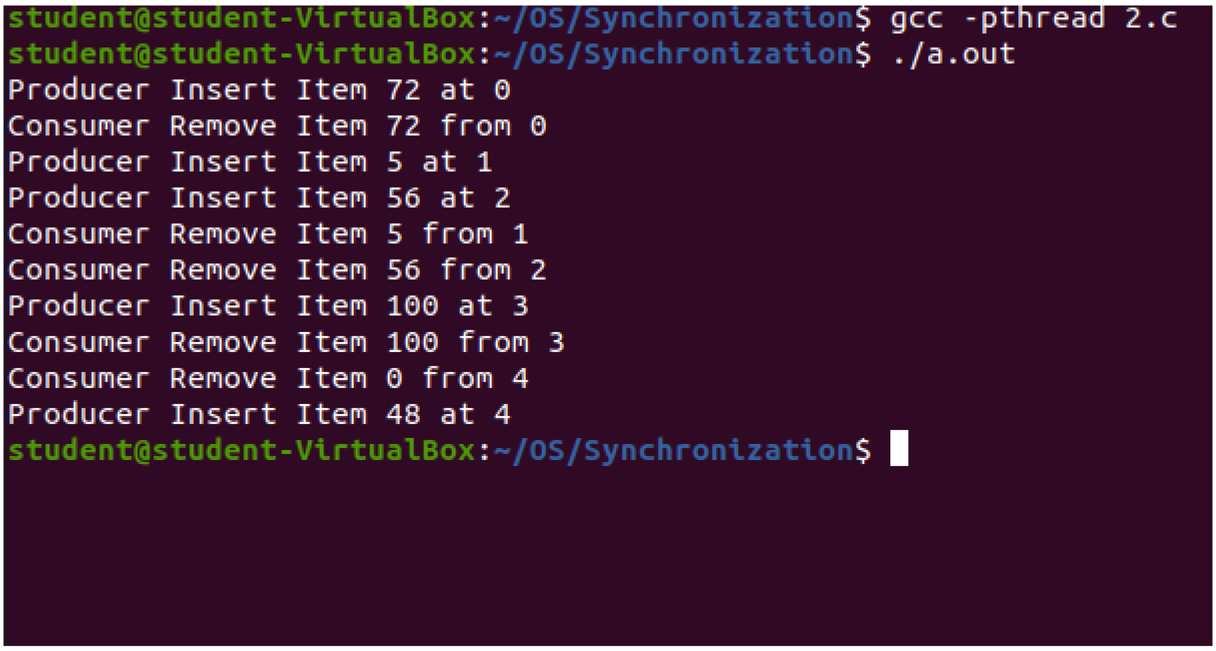
Given 2 processes i and j, we need to write a program that can guarantee mutual exclusion between the two without any additional hardware support. There can be multiple ways to solve this problem, but most of them require additional hardware support. The simplest and the most popular way to do this is by using Peterson's Algorithm for mutual Exclusion. It was developed by Peterson in 1981 though the initial work in this direction was done by Theodorus Jozef Dekker who came up with Dekker's algorithm in 1960, which was later refined by Peterson and came to be known as Peterson's Algorithm.

Basically, Peterson's algorithm provides guaranteed mutual exclusion by using only the shared memory. It uses two ideas in the algorithm:
1. Willingness to acquire lock.
2. Turn to acquire lock.

Explanation:
The idea is that first a thread expresses its desire to acquire a lock and sets flag[self] = 1 and then gives the other thread a chance to acquire the lock. If the thread desires to acquire the lock, then, it gets the lock and passes the chance to the 1st thread. If it does not desire to get the lock then the while loop breaks and the 1st thread gets the chance.

CODE:
```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include<time.h>
#define BufferSize 5 // Size of the buffer
#define prod 0
#define cons 1
int in = 0;
int out = 0;
int buffer[BufferSize];
int flag[2]={0,0};
int turn;
void *producer(void *pno)
{
	int item;
	flag[prod]=1;
	turn = cons;
	while (flag[cons] == 1 && turn == cons);
	item = rand()%101; // Produce an random item
	buffer[in] = item;
	printf("Producer Insert Item %d at %d\n",buffer[in],in);
	in = (in+1)%BufferSize;
	flag[prod]=0;
}

void *consumer(void *cno)
{
        flag[cons]=1;
        turn = prod;
        while (flag[prod] == 1 && turn == prod);
        int item = buffer[out];
        printf("Consumer Remove Item %d from %d\n",item, out);
        out = (out+1)%BufferSize;
        flag[cons]=0;

}

int main()
{
    srand(time(0));
    pthread_t pro[5],con[5];
    for(int i = 0; i < 5; i++) {
        pthread_create(&pro[i], NULL, (void *)producer, NULL);
        pthread_create(&con[i], NULL, (void *)consumer, NULL);

    }

    for(int i = 0; i < 5; i++) {
        pthread_join(pro[i], NULL);
        pthread_join(con[i], NULL);
    }
    return 0;
}
```

SCREENSHOT:



```
student@student-VirtualBox:~/OS/Synchronization$ gcc -pthread 2.c
student@student-VirtualBox:~/OS/Synchronization$ ./a.out
Producer Insert Item 72 at 0
Consumer Remove Item 72 from 0
Producer Insert Item 5 at 1
Producer Insert Item 56 at 2
Consumer Remove Item 5 from 1
Consumer Remove Item 56 from 2
Producer Insert Item 100 at 3
Consumer Remove Item 100 from 3
Consumer Remove Item 0 from 4
Producer Insert Item 48 at 4
student@student-VirtualBox:~/OS/Synchronization$
```

3. Dictionary Problem: Let the producer set up a dictionary of at least 20 words with three attributes (Word, Primary meaning, Secondary meaning) and let the consumer search for the word and retrieve its respective primary and secondary meaning.

The dictionary problem is a classical case of readers writers problem. The producer can be imagined as the writer of the dictionary struct and the consumer can be used to look for the search element. The use of semaphores is made prevent the consumer to obstruct when the producer is setting up a dictionary.
Here, readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
Here, we use :- one mutex m and a semaphore w.
1. An integer variable read_count :- used to maintain the number of readers currently accessing the
2. resource. The variable read_count is initialized to 0.
3. A value of 1 is given initially to m and w.
4. Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.

a. Writer Process :

   Writer requests the entry to critical section.
   If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting. It exits the critical section.

b. Reader Process :

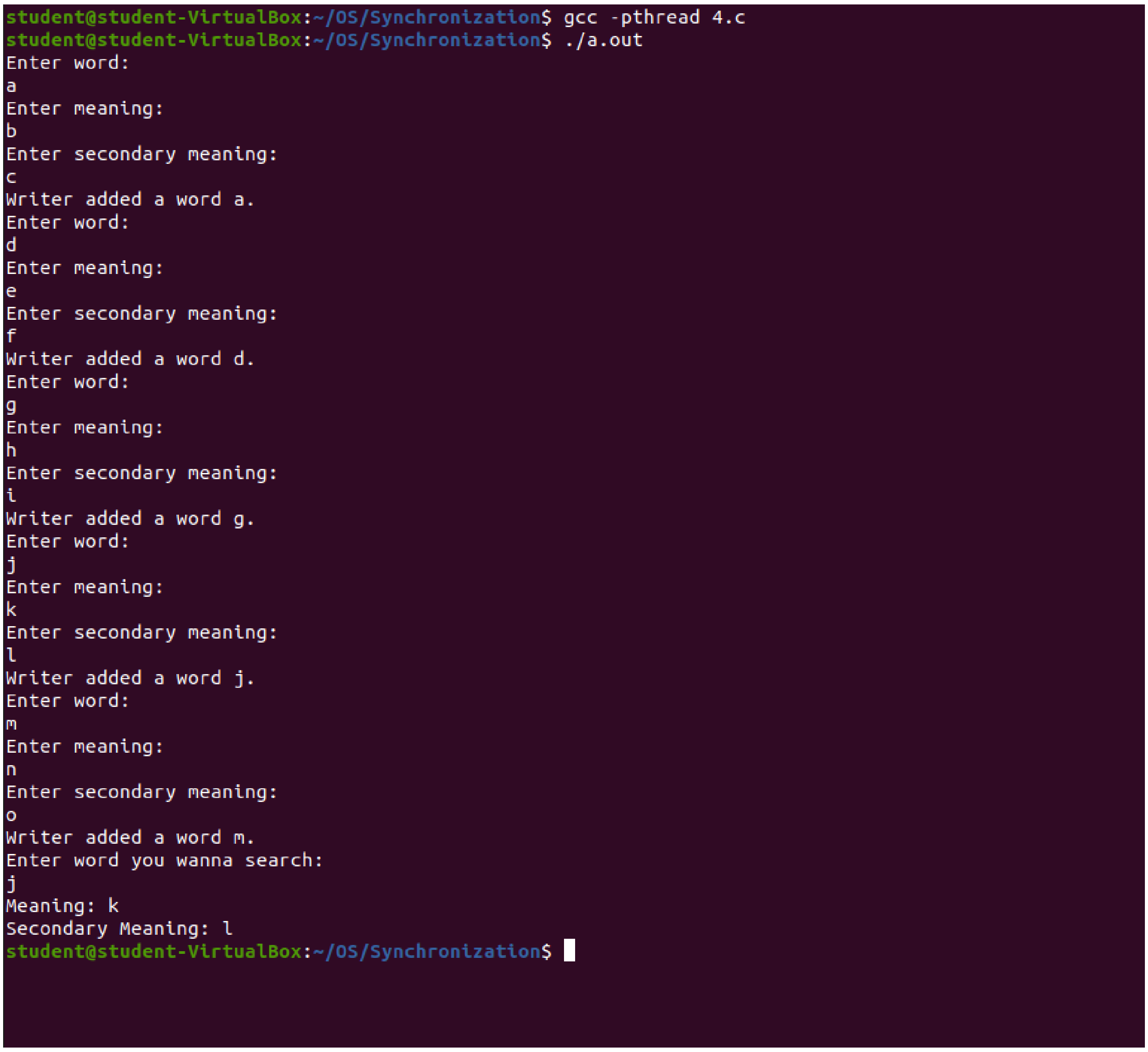
   Reader requests the entry to critical section.
   If allowed:
   i. it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the w semaphore to restrict the entry of writers if any reader is inside.
   ii.It then, signals mutex as any other reader is allowed to enter while others are already reading.
   iii. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore w as now, writer can enter the critical section. If not allowed, it keeps on waiting.

CODE:
```
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
sem_t wrt; char search[100];
pthread_mutex_t mutex;
int numreader = 0;int k=0;int flag;

typedef struct
{
	char word[100];
	char primary[1000];
	char secondary[100];
} dict;

dict dictionary[5];
void *writer(void *wno)
```

```
{
   sem_wait(&wrt);
   flag=1;
   printf("Enter word:\n");
   scanf("%s",search);
   int i;
   for(i=0;i<5;i++)//duplicacy check
   {
         if(strcmp(dictionary[i].word,search)==0)
         {
                  printf("Word already present.\n");
                  flag=0;
                  break;
         }
   }
   if(flag==1)
   {
   strcpy(dictionary[k].word,search);
   printf("Enter meaning:\n");
   scanf("%s",dictionary[k].primary);
   printf("Enter secondary meaning:\n");
   scanf("%s",dictionary[k].secondary);
   printf("Writer added a word %s.\n",dictionary[k].word);k++;
         }
   sem_post(&wrt);
}
void *reader(void *rno)
{
   // Reader acquire the lock before modifying numreader
   pthread_mutex_lock(&mutex);
   numreader++;
   if(numreader == 1) {
      sem_wait(&wrt); // If this id the first reader, then it will block the writer
   }
   pthread_mutex_unlock(&mutex);
   // Reading Section
   int i;
   printf("Enter word you wanna search: \n");
   scanf("%s",search);
   int low=0;
         int high=4;
         while(low<=high)
         {
                  int mid=(low+high)/2;
                  if (strcmp(search,dictionary[mid].word)==0)
                  {
                           printf("Meaning: %s\n", dictionary[mid].primary);
                           printf("Secondary Meaning: %s\n", dictionary[mid].secondary);
         exit(0);
                  }
                  else if(strcmp(search,dictionary[mid].word)>0)
                  {
```

```
                                high=high;
                                low=mid+1;
                        }
                        else
                        {
                                low=low;
                                high=mid-1;
                        }
                }
                printf("Word not found\n");
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader--;
    if(numreader == 0)
    {
        sem_post(&wrt); // If this is the last reader, it will wake up the writer.
    }
    pthread_mutex_unlock(&mutex);
}

int main()
{

    pthread_t read,write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, NULL);
    }

        pthread_create(&read, NULL, (void *)reader, NULL);
    pthread_join(read, NULL);

    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;

}
```

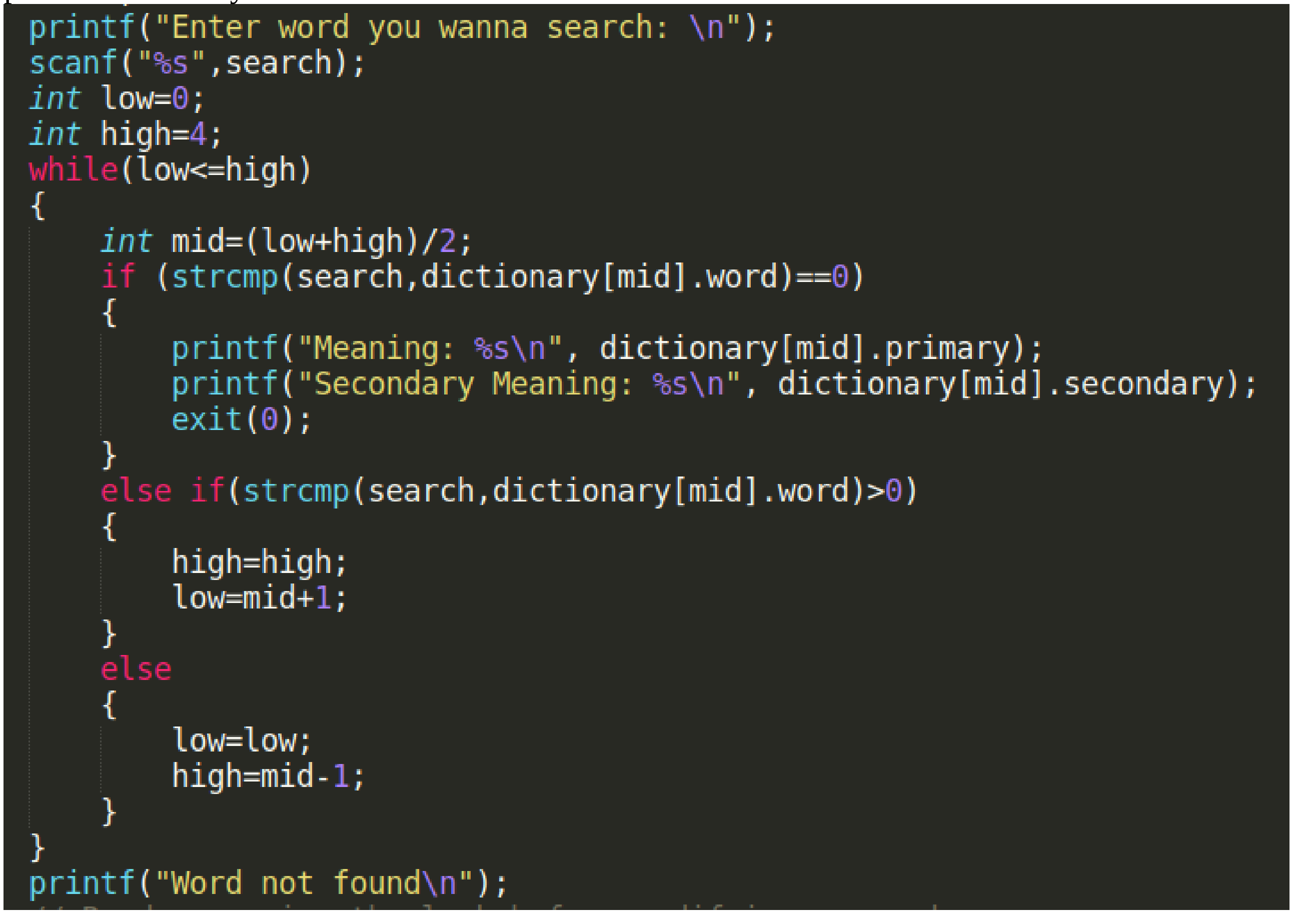
SCREENSHOT:
(Done for 5 words only can be extended for 20 words)



```
student@student-VirtualBox:~/OS/Synchronization$ gcc -pthread 4.c
student@student-VirtualBox:~/OS/Synchronization$ ./a.out
Enter word:
a
Enter meaning:
b
Enter secondary meaning:
c
Writer added a word a.
Enter word:
d
Enter meaning:
e
Enter secondary meaning:
f
Writer added a word d.
Enter word:
g
Enter meaning:
h
Enter secondary meaning:
i
Writer added a word g.
Enter word:
j
Enter meaning:
k
Enter secondary meaning:
l
Writer added a word j.
Enter word:
m
Enter meaning:
n
Enter secondary meaning:
o
Writer added a word m.
Enter word you wanna search:
j
Meaning: k
Secondary Meaning: l
student@student-VirtualBox:~/OS/Synchronization$ 
```

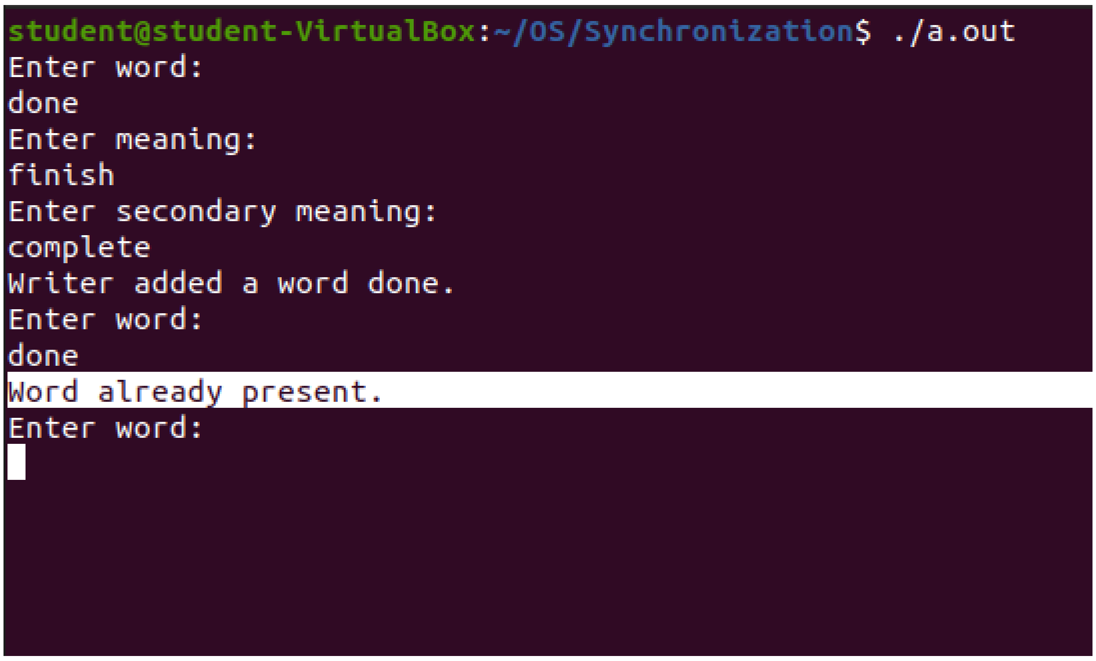
4. Extend Q3 to avoid duplication of dictionary entries and implement an efficient binary search on the consumer side in a multithreaded fashion.

CODE: Same as the previous question. Since it was an extension of the previous question I directly extended the logic to remove duplication and binary search mechanism in the previous question to promote efficiency.



```
printf("Enter word you wanna search: \n");
scanf("%s",search);
int low=0;
int high=4;
while(low<=high)
{
    int mid=(low+high)/2;
    if (strcmp(search,dictionary[mid].word)==0)
    {
        printf("Meaning: %s\n", dictionary[mid].primary);
        printf("Secondary Meaning: %s\n", dictionary[mid].secondary);
        exit(0);
    }
    else if(strcmp(search,dictionary[mid].word)>0)
    {
        high=high;
        low=mid+1;
    }
    else
    {
        low=low;
        high=mid-1;
    }
}
printf("Word not found\n");
```

OUTPUT:



```
student@student-VirtualBox:~/OS/Synchronization$ ./a.out
Enter word:
done
Enter meaning:
finish
Enter secondary meaning:
complete
Writer added a word done.
Enter word:
done
Word already present.
Enter word:
```

5. Dekker’s Algorithm

To obtain such a mutual exclusion, bounded waiting, and progress there have been several algorithms implemented, one of which is Dekker’s Algorithm. To understand the algorithm let’s understand the solution to the critical section problem first.
A process is generally represented as :

```
do {
   //entry section
      critical section
   //exit section
      remainder section
} while (TRUE);
```

Dekker’s algorithm was the first provably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

Although there are many versions of Dekker’s Solution, the final or 5th version is the one that satisfies all of the above conditions and is the most efficient of them all.

Dekker’s Algorithm : Final and completed Solution – -Idea is to use favoured thread notion to determine entry to the critical section. Favoured thread alternates between the thread providing mutual exclusion and avoiding deadlock, indefinite postponement or lockstep synchronization.

ALGO/Trace

```
Main()
{

   // to denote which thread will enter next
   int favouredthread = 1;

   // flags to indicate if each thread is in
   // queue to enter its critical section
   boolean thread1wantstoenter = false;
   boolean thread2wantstoenter = false;

   startThreads();
}

Thread1()
{
   do {

      thread1wantstoenter = true;

      // entry section
      // wait until thread2 wants to enter
      // its critical section
      while (thread2wantstoenter == true) {
          // if 2nd thread is more favored
          if (favaouredthread == 2) {

              // gives access to other thread
              thread1wantstoenter = false;

              // wait until this thread is favored
              while (favouredthread == 2)
                  ;

              thread1wantstoenter = true;
          }
      }

      // critical section

      // favor the 2nd thread
      favouredthread = 2;

      // exit section
      // indicate thread1 has completed
      // its critical section
      thread1wantstoenter = false;

      // remainder section

   } while (completed == false)
}

Thread2()
{

   do {

      thread2wantstoenter = true;

      // entry section
      // wait until thread1 wants to enter
      // its critical section
      while (thread1wantstoenter == true) {

          // if 1st thread is more favored
          if (favaouredthread == 1) {

              // gives access to other thread
              thread2wantstoenter = false;

              // wait until this thread is favored
              while (favouredthread == 1)
                  ;
              thread2wantstoenter = true;
          }
      }

      // critical section

      // favour the 1st thread
      favouredthread = 1;

      // exit section
      // indicate thread2 has completed
      // its critical section
      thread2wantstoenter = false;

      // remainder section

   } while (completed == false)
}
```
This version guarantees a complete solution to the critical solution problem.

Additional Problems:

1. Dining Philosopher's Problem

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available.Otherwise a philosopher puts down their chopstick and begin thinking again.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below −

```
semaphore chopstick [5];
```
Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

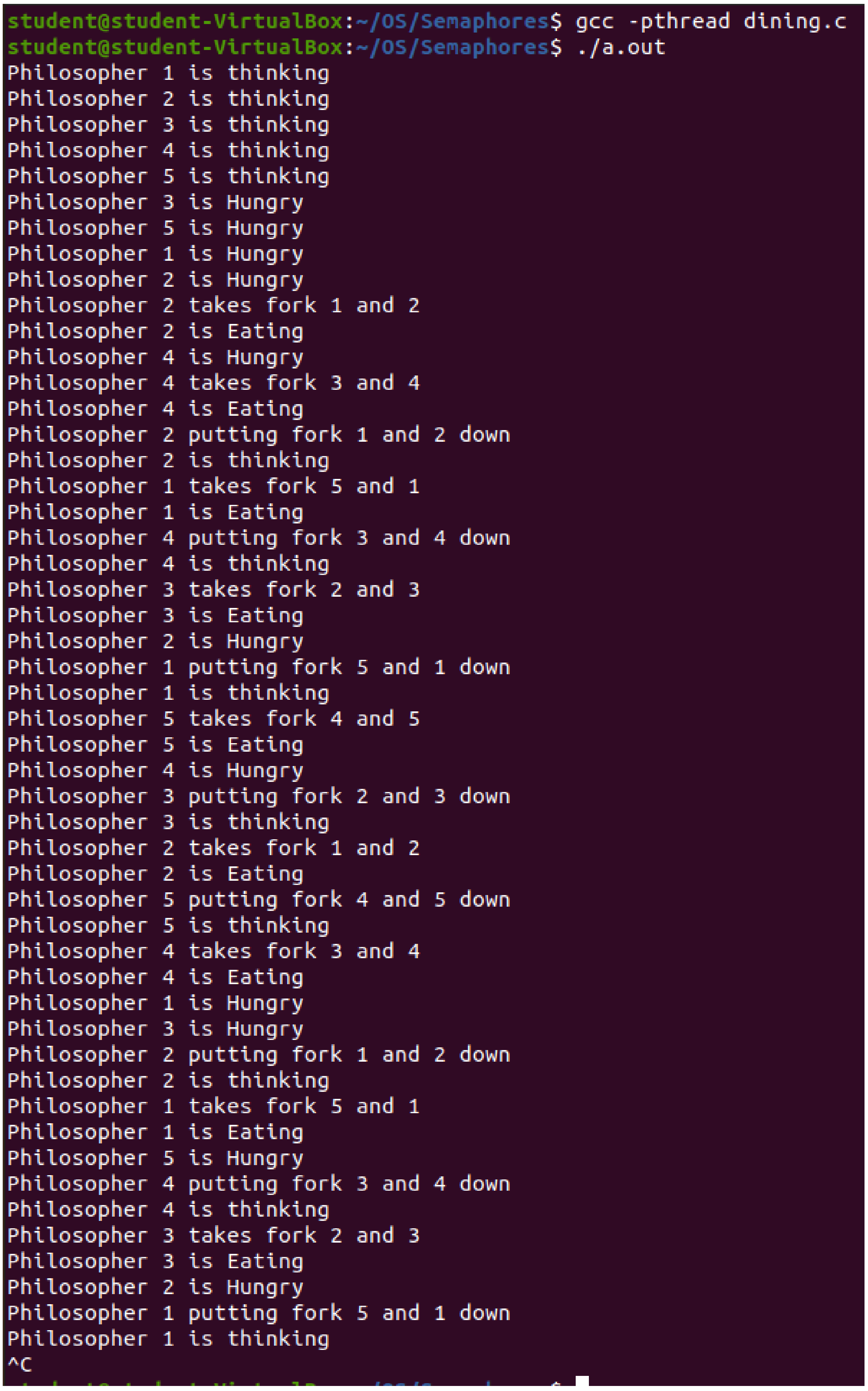
The structure of a random philosopher i is given as follows −

```
do {
   wait( chopstick[i] );
   wait( chopstick[ (i+1) % 5] );
   . .
   . EATING THE RICE
   .
   signal( chopstick[i] );
   signal( chopstick[ (i+1) % 5] );
   .
   . THINKING
   .
} while(1);
```

CODE:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define N 5 //no. of philosophers
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
        if (state[phnum] == HUNGRY
                && state[LEFT] != EATING
                && state[RIGHT] != EATING) {
                // state that eating
                state[phnum] = EATING;
                sleep(2); //eating time
                printf("Philosopher %d takes fork %d and %d\n",
                                        phnum + 1, LEFT + 1, phnum + 1);
                printf("Philosopher %d is Eating\n", phnum + 1);
                sem_post(&S[phnum]);
        }
}

// take up chopsticks
void take_fork(int phnum)
{

        sem_wait(&mutex);
        // state that hungry
        state[phnum] = HUNGRY;
        printf("Philosopher %d is Hungry\n", phnum + 1);
        // eat if neighbours are not eating
        test(phnum);
        sem_post(&mutex);
        // if unable to eat wait to be signalled
        sem_wait(&S[phnum]);
        sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

        sem_wait(&mutex);
        // state that thinking
        state[phnum] = THINKING;
        printf("Philosopher %d putting fork %d and %d down\n",
                phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is thinking\n", phnum + 1);
        test(LEFT);
        test(RIGHT);
        sem_post(&mutex);
}

void* philospher(void* num)
{

        while (1) {

                int* i = num;
                sleep(1);
                take_fork(*i);
                sleep(0); //immediately drops fork after eating
                put_fork(*i);
        }
}

int main()
{

        int i;
        pthread_t thread_id[N];
        // initialize the semaphores
        sem_init(&mutex, 0, 1);
        for (i = 0; i < N; i++)
                sem_init(&S[i], 0, 0);

        for (i = 0; i < N; i++) {
                // create philosopher processes
                pthread_create(&thread_id[i], NULL,
                                        philospher, &phil[i]);
                printf("Philosopher %d is thinking\n", i + 1);
        }

        for (i = 0; i < N; i++)
                pthread_join(thread_id[i], NULL); }
```

OUTPUT



```
student@student-VirtualBox:~/OS/Semaphores$ gcc -pthread dining.c
student@student-VirtualBox:~/OS/Semaphores$ ./a.out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 5 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
^C
```

2. Readers Writers Problem:
The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows −

Reader Process
The code that defines the reader process is given below −

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);
.
. READ THE OBJECT
.
wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

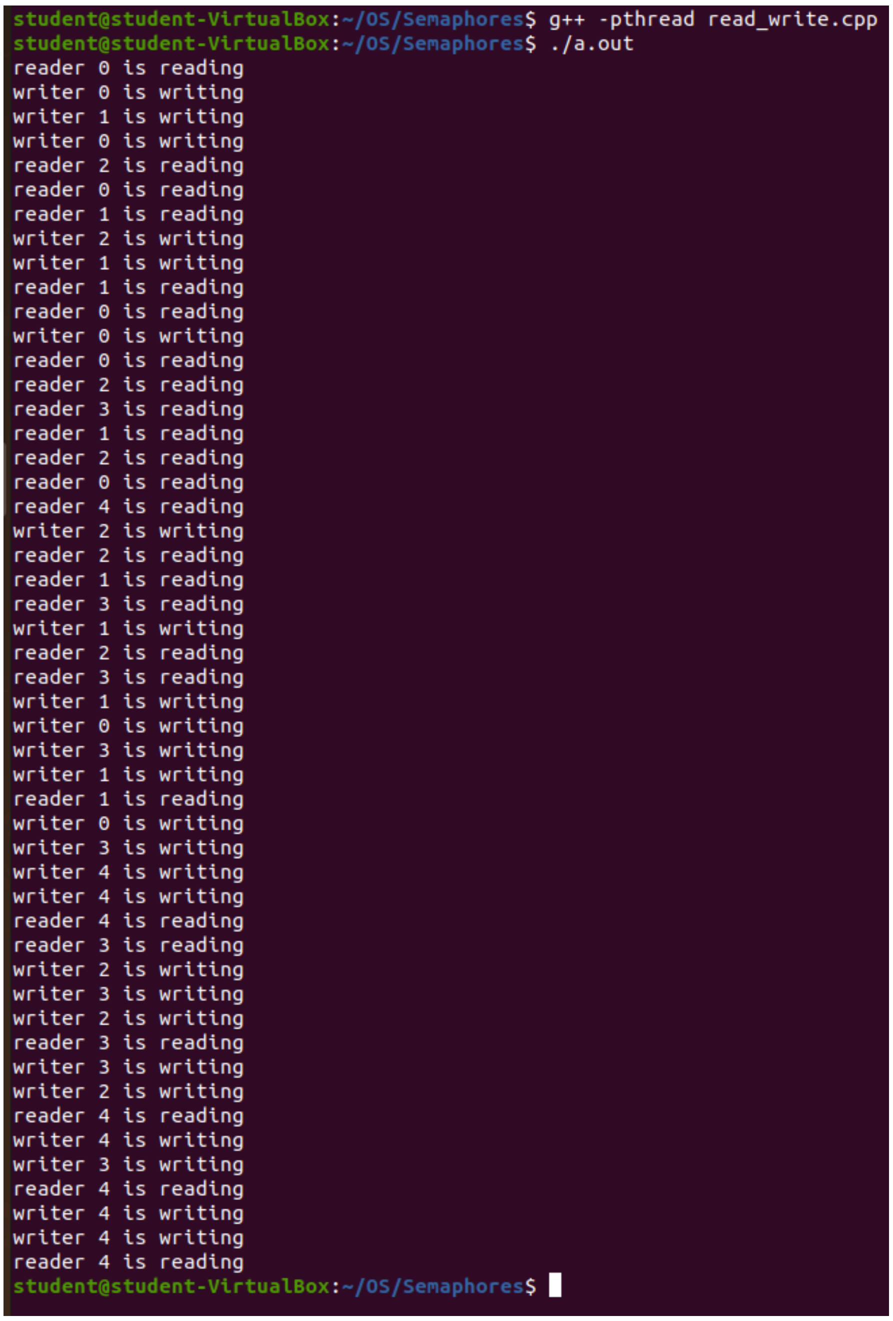
In the above code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

Code:

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
using namespace std;
class monitor {
private:

   int rcnt;
   int wcnt;
   int waitr;
   int waitw;
   pthread_cond_t canread;
```

```
    pthread_cond_t canwrite;
    pthread_mutex_t condlock;

public:
    monitor()
    {
        rcnt = 0;
        wcnt = 0;
        waitr = 0;
        waitw = 0;

        pthread_cond_init(&canread, NULL);
        pthread_cond_init(&canwrite, NULL);
        pthread_mutex_init(&condlock, NULL);
    }

    void beginread(int i)
    {
        pthread_mutex_lock(&condlock);


        if (wcnt == 1 || waitw > 0) {
            waitr++;
            pthread_cond_wait(&canread, &condlock);
            waitr--;
        }
        rcnt++;
        cout << "reader " << i << " is reading\n";
        pthread_mutex_unlock(&condlock);
        pthread_cond_broadcast(&canread);
    }

    void endread(int i)
    {
        pthread_mutex_lock(&condlock);

        if (--rcnt == 0)
            pthread_cond_signal(&canwrite);

        pthread_mutex_unlock(&condlock);
    }

    void beginwrite(int i)
    {
        pthread_mutex_lock(&condlock);
        if (wcnt == 1 || rcnt > 0) {
            ++waitw;
            pthread_cond_wait(&canwrite, &condlock);
            --waitw;
        }
        wcnt = 1;
        cout << "writer " << i << " is writing\n";
        pthread_mutex_unlock(&condlock);
    }

    void endwrite(int i)
    {
        pthread_mutex_lock(&condlock);
        wcnt = 0;

        // if any readers are waiting, threads are unblocked
        if (waitr > 0)
            pthread_cond_signal(&canread);
        else
            pthread_cond_signal(&canwrite);
        pthread_mutex_unlock(&condlock);
    }

}

M;  //object

void* reader(void* id)
{
    int c = 0;
    int i = *(int*)id;
    while (c < 5) {
        usleep(1);
        M.beginread(i);
        M.endread(i);
        c++;
    }
    pthread_exit(0);
}

void* writer(void* id)
{
    int c = 0;
    int i = *(int*)id;
    while (c < 5) {
        usleep(1);
        M.beginwrite(i);
        M.endwrite(i);
        c++;
    }
    pthread_exit(0);
}

int main()
{
    pthread_t r[5], w[5];
    int id[5];
    for (int i = 0; i < 5; i++) {
        id[i] = i;
        pthread_create(&r[i], NULL, &reader, &id[i]);
        pthread_create(&w[i], NULL, &writer, &id[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(r[i], NULL);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(w[i], NULL);
    }
}
```

OUTPUT:



```
student@student-VirtualBox:~/OS/Semaphores$ g++ -pthread read_write.cpp
student@student-VirtualBox:~/OS/Semaphores$ ./a.out
reader 0 is reading
writer 0 is writing
writer 1 is writing
writer 0 is writing
reader 2 is reading
reader 0 is reading
reader 1 is reading
writer 2 is writing
writer 1 is writing
reader 1 is reading
reader 0 is reading
writer 0 is writing
reader 0 is reading
reader 2 is reading
reader 3 is reading
reader 1 is reading
reader 2 is reading
reader 0 is reading
reader 4 is reading
writer 2 is writing
reader 2 is reading
reader 1 is reading
reader 3 is reading
writer 1 is writing
reader 2 is reading
reader 3 is reading
writer 1 is writing
writer 0 is writing
writer 3 is writing
writer 1 is writing
reader 1 is reading
writer 0 is writing
writer 3 is writing
writer 4 is writing
writer 4 is writing
reader 4 is reading
reader 3 is reading
writer 2 is writing
writer 3 is writing
writer 2 is writing
reader 3 is reading
writer 3 is writing
writer 2 is writing
reader 4 is reading
writer 4 is writing
writer 3 is writing
reader 4 is reading
writer 4 is writing
writer 4 is writing
reader 4 is reading
student@student-VirtualBox:~/OS/Semaphores$ 
```

3. Santa Claus Problem:

Santa's code is pretty straightforward. Remember that it runs in a loop.

```
santaSem.wait()
 mutex.wait()
 if reindeer == 9:
 prepareSleigh()
 reindeerSem.signal(9)
 else if elves == 3:
 helpElves()
 mutex.signal()
```

When Santa wakes up, he checks which of the two conditions holds and either deals with the reindeer or the waiting elves. If there are nine reindeer waiting, Santa invokes prepareSleigh, then signals reindeerSem nine times, allowing the reindeer to invoke getHitched. If there are elves waiting, Santa just invokes helpElves. There is no need for the elves to wait for Santa; once they signal santaSem, they can invoke getHelp immediately.
Here is the code for reindeer:

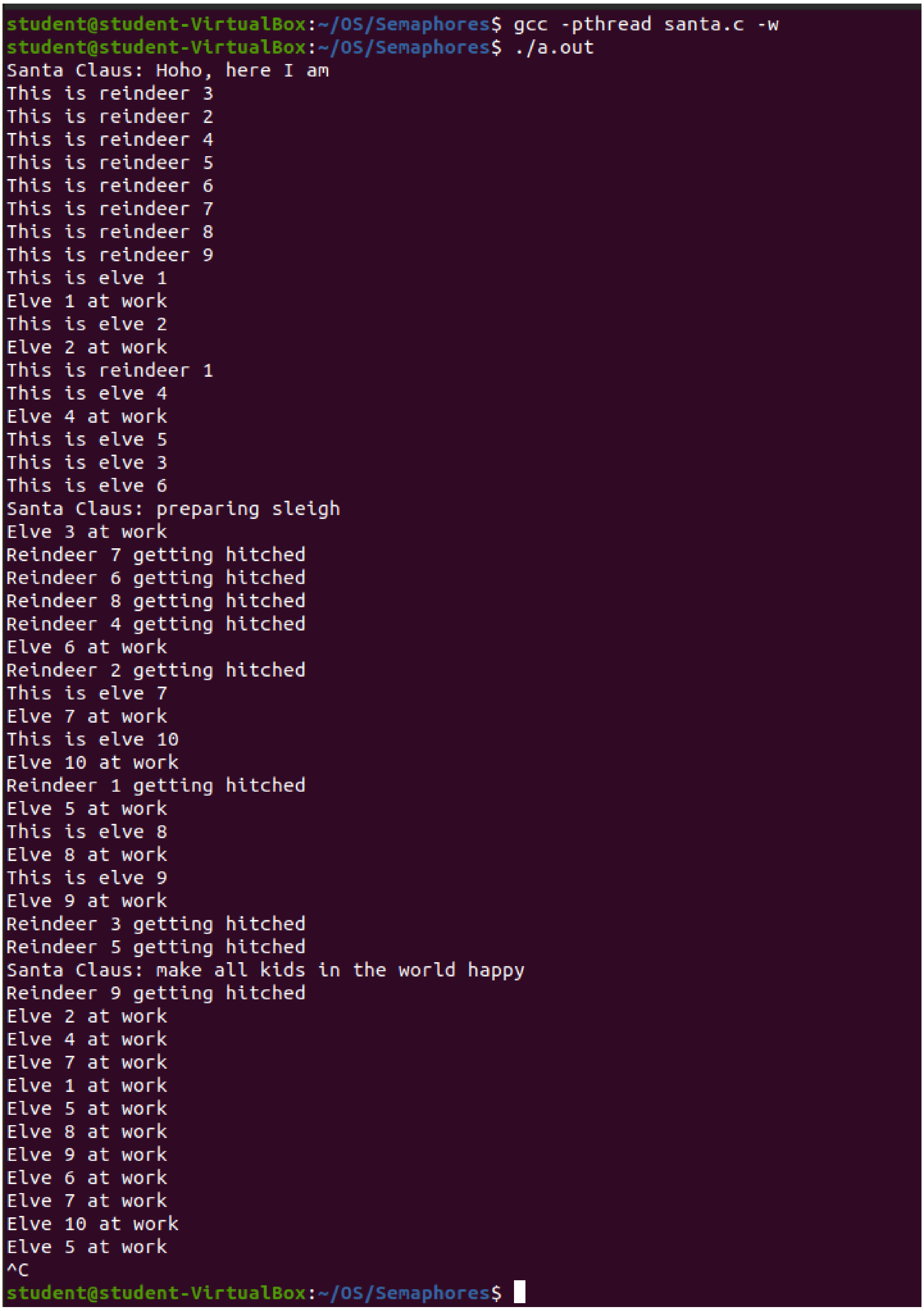
```
mutex.wait()
reindeer += 1
if reindeer == 9:
santaSem.signal()
mutex.signal()
reindeerSem.wait()
getHitched()
```

The ninth reindeer signals Santa and then joins the other reindeer waiting on reindeerSem. When Santa signals, the reindeer all execute getHitched. The elf code is similar, except that when the third elf arrives it has to bar subsequent arrivals until the first three have executed getHelp.

CODE:

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

pthread_t *CreateThread(void *(*f)(void *), void *a)
{
	pthread_t *t = malloc(sizeof(pthread_t));
	assert(t != NULL);
	int ret = pthread_create(t, NULL, f, a);
	assert(ret == 0);
	return t;
}

static const int N_ELVES = 10;
static const int N_REINDEER = 9;
static int elves;
static int reindeer;
static sem_t santaSem;
static sem_t reindeerSem;
static sem_t elfTex;
static sem_t mutex;

void *SantaClaus(void *arg)
{
	printf("Santa Claus: Hoho, here I am\n");
	while (true)
	{
		sem_wait(&santaSem);
		sem_wait(&mutex);
		if (reindeer == N_REINDEER)
		{
			printf("Santa Claus: preparing sleigh\n");
			for (int r = 0; r < N_REINDEER; r++)
				sem_post(&reindeerSem);
			printf("Santa Claus: make all kids in the world happy\n");
			reindeer = 0;
		}
		else if (elves == 3)
		{
			printf("Santa Claus: helping elves\n");
		}
		sem_post(&mutex);
	}
	return arg;
}

void *Reindeer(void *arg)
{
	int id = (int)arg;
	printf("This is reindeer %d\n", id);
	while (true)
	{
		sem_wait(&mutex);
		reindeer++;
		if (reindeer == N_REINDEER)
			sem_post(&santaSem);
		sem_post(&mutex);
		sem_wait(&reindeerSem);
		printf("Reindeer %d getting hitched\n", id);
		sleep(20);
	}
	return arg;
}

void *Elve(void *arg)
{
	int id = (int)arg;
	printf("This is elve %d\n", id);
	while (true)
	{
		bool need_help = random() % 100 < 10;
		if (need_help)
		{
			sem_wait(&elfTex);
			sem_wait(&mutex);
			elves++;
			if (elves == 3)
				sem_post(&santaSem);
			else
				sem_post(&elfTex);
			sem_post(&mutex);

			printf("Elve %d will get help from Santa Claus\n", id);
			sleep(10);

			sem_wait(&mutex);
			elves--;
			if (elves == 0)
				sem_post(&elfTex);
			sem_post(&mutex);
		}
		// Do some work
		printf("Elve %d at work\n", id);
		sleep(2 + random() % 5);
	}
	return arg;
}

int main(int ac, char **av)
{
	elves = 0;
	reindeer = 0;
	sem_init(&santaSem, 0, 0);
	sem_init(&reindeerSem, 0, 0);
	sem_init(&elfTex, 0, 1);
	sem_init(&mutex, 0, 1);

	pthread_t *santa_claus = CreateThread(SantaClaus, 0);

	pthread_t *reindeers[N_REINDEER];
	for (int r = 0; r < N_REINDEER; r++)
		reindeers[r] = CreateThread(Reindeer, (void *)r + 1);

	pthread_t *elves[N_ELVES];
	for (int e = 0; e < N_ELVES; e++)
		elves[e] = CreateThread(Elve, (void *)e + 1);

	int ret = pthread_join(*santa_claus, NULL);
	assert(ret == 0); }
```

OUTPUT:



```
student@student-VirtualBox:~/OS/Semaphores$ gcc -pthread santa.c -w
student@student-VirtualBox:~/OS/Semaphores$ ./a.out
Santa Claus: Hoho, here I am
This is reindeer 3
This is reindeer 2
This is reindeer 4
This is reindeer 5
This is reindeer 6
This is reindeer 7
This is reindeer 8
This is reindeer 9
This is elve 1
Elve 1 at work
This is elve 2
Elve 2 at work
This is reindeer 1
This is elve 4
Elve 4 at work
This is elve 5
This is elve 3
This is elve 6
Santa Claus: preparing sleigh
Elve 3 at work
Reindeer 7 getting hitched
Reindeer 6 getting hitched
Reindeer 8 getting hitched
Reindeer 4 getting hitched
Elve 6 at work
Reindeer 2 getting hitched
This is elve 7
Elve 7 at work
This is elve 10
Elve 10 at work
Reindeer 1 getting hitched
Elve 5 at work
This is elve 8
Elve 8 at work
This is elve 9
Elve 9 at work
Reindeer 3 getting hitched
Reindeer 5 getting hitched
Santa Claus: make all kids in the world happy
Reindeer 9 getting hitched
Elve 2 at work
Elve 4 at work
Elve 7 at work
Elve 1 at work
Elve 5 at work
Elve 8 at work
Elve 9 at work
Elve 6 at work
Elve 7 at work
Elve 10 at work
Elve 5 at work
^C
student@student-VirtualBox:~/OS/Semaphores$ 
```

4. Building H_2O Problem

Initially hydroQueue and oxyQueue are locked. When an oxygen thread arrives it signals hydroQueue twice, allowing two hydrogens to proceed. Then the oxygen thread waits for the hydrogen threads to arrive.

Oxygen code

```
mutex.wait()
oxygen += 1
if hydrogen >= 2:
hydroQueue.signal(2)
hydrogen -= 2
oxyQueue.signal()
oxygen -= 1
else:
mutex.signal()
oxyQueue.wait()
bond()
barrier.wait()
mutex.signal()
```

As each oxygen thread enters, it gets the mutex and checks the scoreboard. If there are at least two hydrogen threads waiting, it signals two of them and itself and then bonds. If not, it releases the mutex and waits. After bonding, threads wait at the barrier until all three threads have bonded, and then the oxygen thread releases the mutex. Since there is only one oxygen thread in each set of three, we are guaranteed to signal mutex once. The code for hydrogen is similar:

Hydrogen code

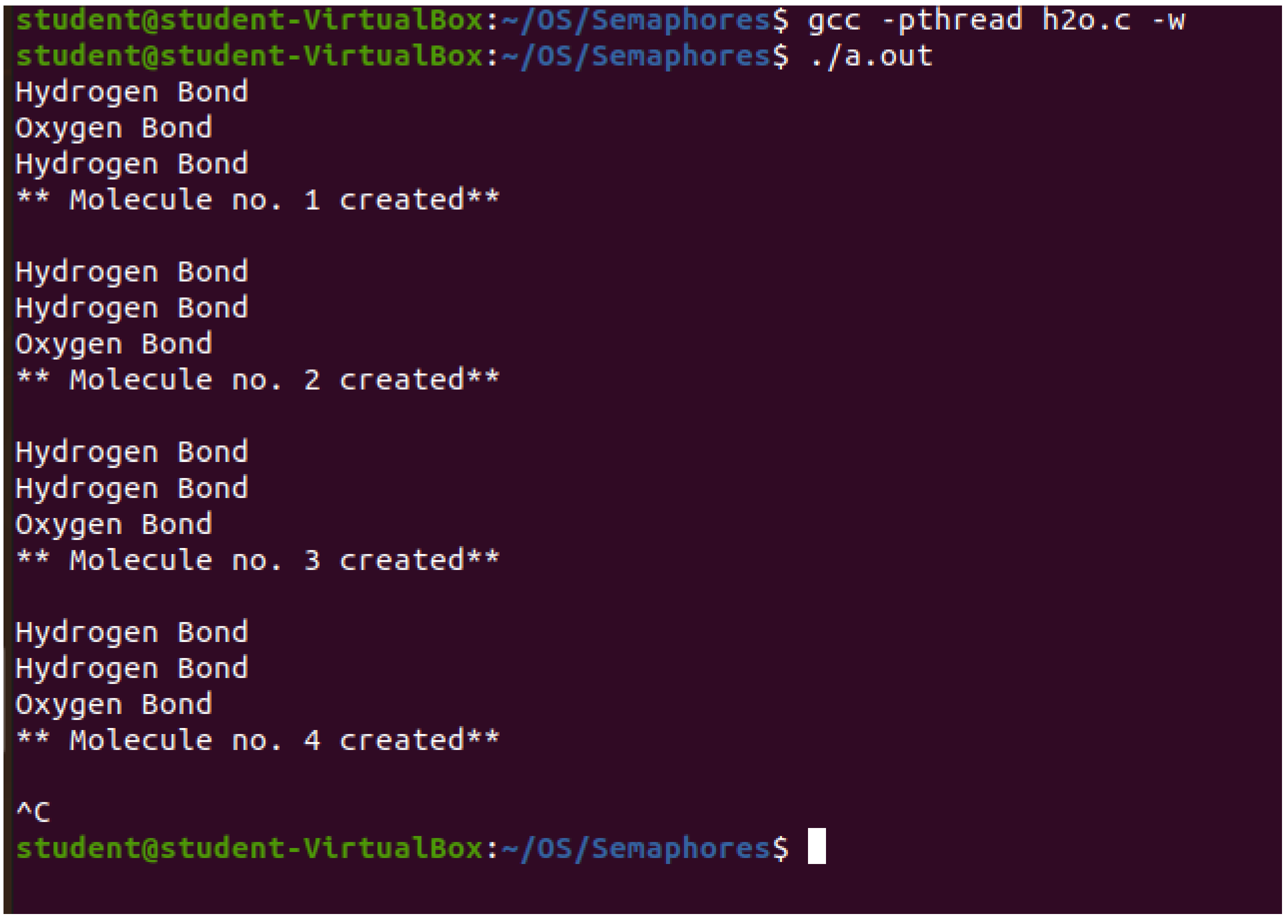
```
mutex.wait()
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
hydroQueue.signal(2)
hydrogen -= 2
oxyQueue.signal()
oxygen -= 1
else:
mutex.signal()
hydroQueue.wait()
bond()
barrier.wait()
```

CODE:

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
sem_t smutex,oxyQueue,hydroQueue;

int oxygen=0,hydregen=0;

pthread_t oxyThread,hydroThread1,hydroThread2;

int bond(){
   static int i=0;
   i++;
```

```
    if((i%3)==0)
        printf("** Molecule no. %d created**\n\n",i/3);
    sleep(2);
    return(0);
}

void* oxyFn(void* arg){

    while(1){
        sem_wait(&smutex);

        oxygen+=1;

        if(hydregen>=2){
            sem_post(&hydroQueue);
            sem_post(&hydroQueue);
            hydregen-=2;
            sem_post(&oxyQueue);
            oxygen-=1;
        }
        else {
            sem_post(&smutex);
        }

        sem_wait(&oxyQueue);
        printf("Oxygen Bond\n");
        bond();

        sleep(3);
        sem_post(&smutex);
    }
}

void* hydroFn(void* arg){
    while(1){
        sem_wait(&smutex);

        hydregen+=1;

        if(hydregen>=2 && oxygen>=1){
            sem_post(&hydroQueue);
            sem_post(&hydroQueue);
            hydregen-=2;
            sem_post(&oxyQueue);
            oxygen-=1;
        }
        else{
            sem_post(&smutex);
        }

        sem_wait(&hydroQueue);
        printf("Hydrogen Bond\n");
        bond();
        sleep(3);
        }

}

int main(){

    if(sem_init(&smutex,0,1)==-1){
        perror("error initilalizing semaphore\n");
    }
    if(sem_init(&oxyQueue,0,0)==-1){
        perror("error initilalizing semaphore\n");
    }
    if(sem_init(&hydroQueue,0,0)==-1){
        perror("error initilalizing semaphore\n");
    }
    sleep(2);
    pthread_create(&oxyThread,0,oxyFn, NULL);
    pthread_create(&hydroThread1,0,hydroFn, NULL);
    pthread_create(&hydroThread2,0,hydroFn, NULL);

    for(;;);

}
```

OUTPUT:



```
student@student-VirtualBox:~/OS/Semaphores$ gcc -pthread h2o.c -w
student@student-VirtualBox:~/OS/Semaphores$ ./a.out
Hydrogen Bond
Oxygen Bond
Hydrogen Bond
** Molecule no. 1 created**

Hydrogen Bond
Hydrogen Bond
Oxygen Bond
** Molecule no. 2 created**

Hydrogen Bond
Hydrogen Bond
Oxygen Bond
** Molecule no. 3 created**

Hydrogen Bond
Hydrogen Bond
Oxygen Bond
** Molecule no. 4 created**

^C
student@student-VirtualBox:~/OS/Semaphores$ 
```