Assignment-8
Synchronization Problems

Subash Mylraj
(CED18I051)

30 November 2020

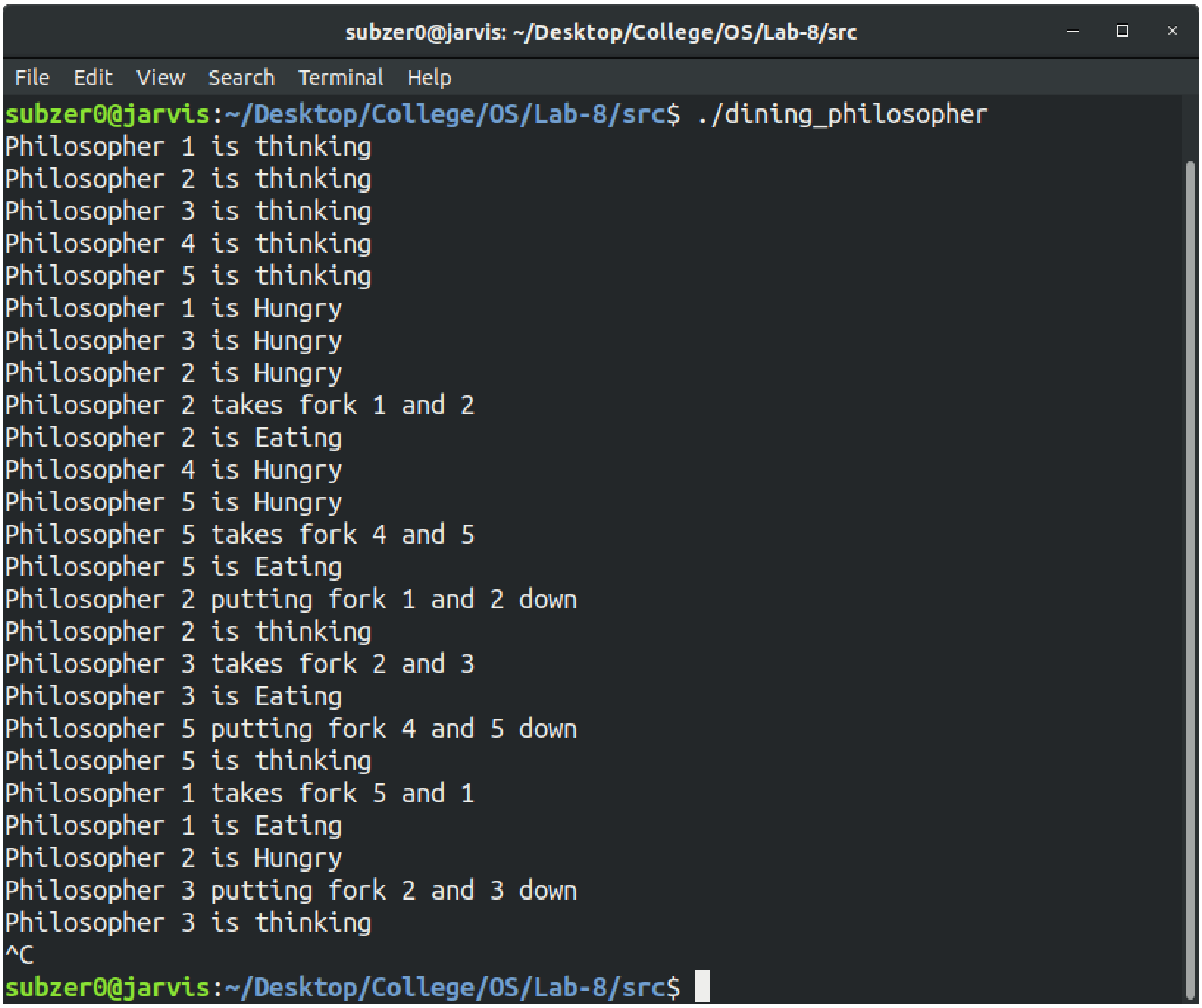
Dining Philosopher problem

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#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);

      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]) has no effect
      // during takefork
      // used to wake up hungry philosophers
      // during putfork
      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);

      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:



subzer0@jarvis: ~/Desktop/College/OS/Lab-8/src
File Edit View Search Terminal Help

```
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$ ./dining_philosopher
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
^C
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$
```

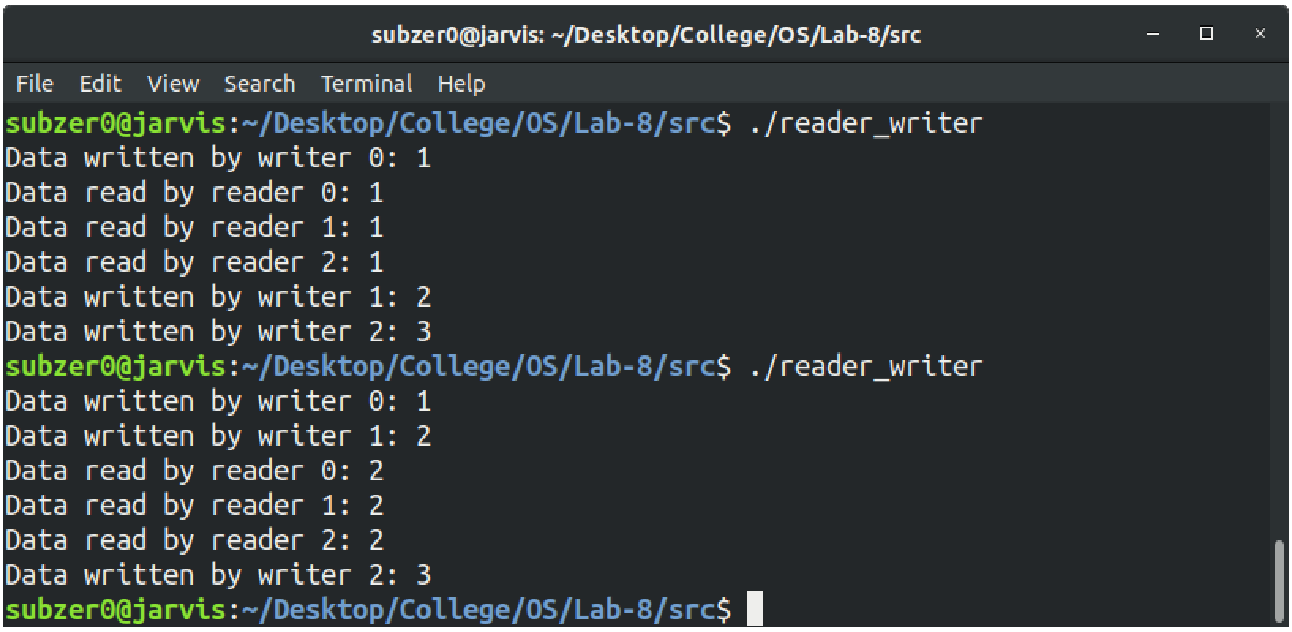
Reader Writer problem

Code:

```
#include<stdio.h>
```

```c
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>

int data = 0, rcount = 0;
sem_t mutex, writeblock;

void *reader(void *arg){

   int f = ((int)arg);
   sem_wait(&mutex);

   rcount += 1;
   if(rcount == 1){
      sem_wait(&writeblock);
   }
   sem_post(&mutex);
   printf("Data read by reader %d: %d\n", f, data);
   sleep(1);
   sem_wait(&mutex);
   rcount -= 1;
   if(rcount == 0){
      sem_post(&writeblock);
   }
   sem_post(&mutex);
}

void *writer(void *arg){

   int f = ((int)arg);
   sem_wait(&writeblock);
   data += 1;
   printf("Data written by writer %d: %d\n", f, data);
   sleep(1);
   sem_post(&writeblock);
}

int main (){

   int i, b;
   pthread_t rtid[5], wtid[5];
   sem_init(&mutex, 0, 1);
   sem_init(&writeblock, 0, 1);

   for(i=0; i<=2; i++){
      pthread_create(&wtid[i], NULL, writer, (void *)i);
      pthread_create(&rtid[i], NULL, reader, (void *)i);
   }

   for(i=0; i<=2; i++){
      pthread_join(wtid[i], NULL);
      pthread_join(rtid[i], NULL);
   }

   return 0;
}
```

Output:


subzer0@jarvis: ~/Desktop/College/OS/Lab-8/src
File Edit View Search Terminal Help

```
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$ ./reader_writer
Data written by writer 0: 1
Data read by reader 0: 1
Data read by reader 1: 1
Data read by reader 2: 1
Data written by writer 1: 2
Data written by writer 2: 3
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$ ./reader_writer
Data written by writer 0: 1
Data written by writer 1: 2
Data read by reader 0: 2
Data read by reader 1: 2
Data read by reader 2: 2
Data written by writer 2: 3
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$
```

H2O problem

Code:

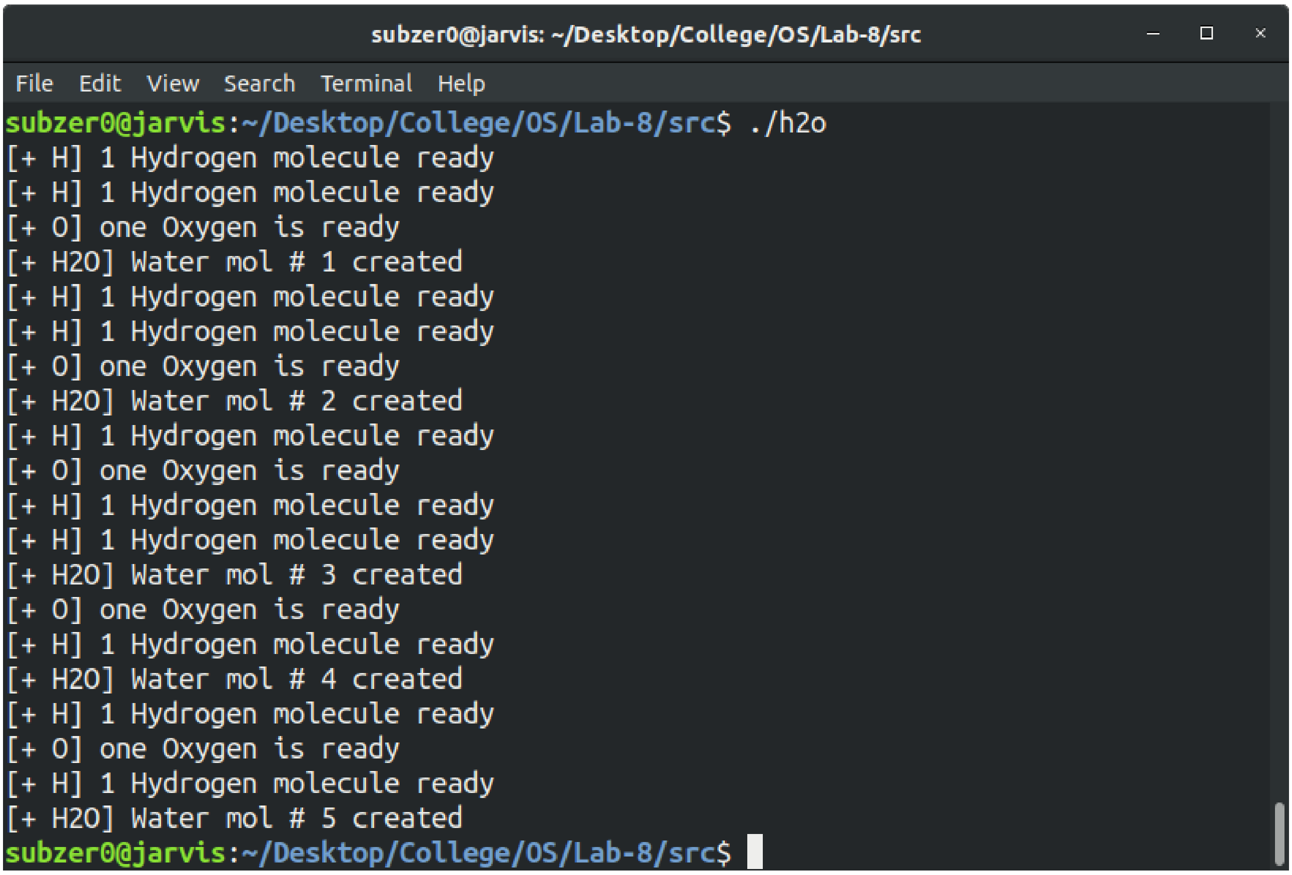
```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#include <unistd.h>

int hydrogen = 0, oxygen = 0, bcount = 0;
sem_t mutex, hydroqueue, oxyqueue, b_mutex, sbarrier;

void barrier_wait(){

    sem_wait(&b_mutex);
    bcount++;
    sem_post(&b_mutex);

    if(bcount == 3){
        sem_post(&sbarrier);
    }

    sem_wait(&sbarrier);
    sem_post(&sbarrier);
}

void bond(){

    static int i = 0;
    i++;

    if(i%3 == 0){
        printf("[+ H2O] Water mol # %d created\n", i/3);
    }
}

void * o_fn(void *arg){

    sem_wait(&mutex);
    oxygen += 1;
    if(hydrogen >= 2){
        sem_post(&hydroqueue);
        sem_post(&hydroqueue); //increase by 2 so twice --> allows 2 H molecules
        hydrogen -= 2;
        sem_post(&oxyqueue);
        oxygen -= 1;
    }
    else{
        sem_post(&mutex);
    }

    sem_wait(&oxyqueue);
    printf("[+ O] one Oxygen is ready\n");
    bond();
    barrier_wait();
    sem_post(&mutex);
}

void *h_fn(void *arg){

    sem_wait(&mutex);
    hydrogen += 1;

    if(hydrogen >= 2 && oxygen >= 1){
        sem_post(&hydroqueue);
        sem_post(&hydroqueue);
        hydrogen -= 2;
        sem_post(&oxyqueue);
        oxygen -= 1;
    }
    else{
        sem_post(&mutex);
    }

    sem_wait(&hydroqueue);
    printf("[+ H] 1 Hydrogen molecule ready\n");
    bond();

    barrier_wait();
}

int main(){

    sem_init(&b_mutex, 0, 1);
    sem_init(&sbarrier, 0, 0);
    sem_init(&mutex, 0, 1);
    sem_init(&oxyqueue, 0, 0);
    sem_init(&hydroqueue, 0, 0);

    pthread_t o_thread[10], h_thread[20];

    for(int i = 0; i<5; i++){
        pthread_create(&o_thread[i], NULL, o_fn, NULL);
        pthread_create(&h_thread[i], NULL, h_fn, NULL);
        pthread_create(&h_thread[i+10], NULL, h_fn, NULL);
    }

    for(int i = 0; i<5; i++){
        pthread_join(o_thread[i], NULL);
        pthread_join(h_thread[i], NULL);
        pthread_join(h_thread[i+10], NULL);

    }
    return 0;
}
```

Explanation:

The size of hydroqueue is 2 and that of oxyqueue is 1. As the first two hydrogen threads arrive, they get pushed to the hudrogen queue, and as the first oxygen queue arrives, it gets pushed to the oxygen queue. Once the requirement to bond is achieved (2 x hydrogen and 1 x oxygen), the threads call where they bond after which they are moved to the barrier to make sure that the bonded hydrogen and oxygen are accounted for.

The forthcoming hydrogen and oxygen threads are made to wait until the their respective queues can accmodate them.

Output:



subzer0@jarvis: ~/Desktop/College/OS/Lab-8/src
File Edit View Search Terminal Help

```
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$ ./h2o
[+ H] 1 Hydrogen molecule ready
[+ H] 1 Hydrogen molecule ready
[+ O] one Oxygen is ready
[+ H2O] Water mol # 1 created
[+ H] 1 Hydrogen molecule ready
[+ H] 1 Hydrogen molecule ready
[+ O] one Oxygen is ready
[+ H2O] Water mol # 2 created
[+ H] 1 Hydrogen molecule ready
[+ O] one Oxygen is ready
[+ H] 1 Hydrogen molecule ready
[+ H] 1 Hydrogen molecule ready
[+ H2O] Water mol # 3 created
[+ O] one Oxygen is ready
[+ H] 1 Hydrogen molecule ready
[+ H2O] Water mol # 4 created
[+ H] 1 Hydrogen molecule ready
[+ O] one Oxygen is ready
[+ H] 1 Hydrogen molecule ready
[+ H2O] Water mol # 5 created
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$
```

Senate problem

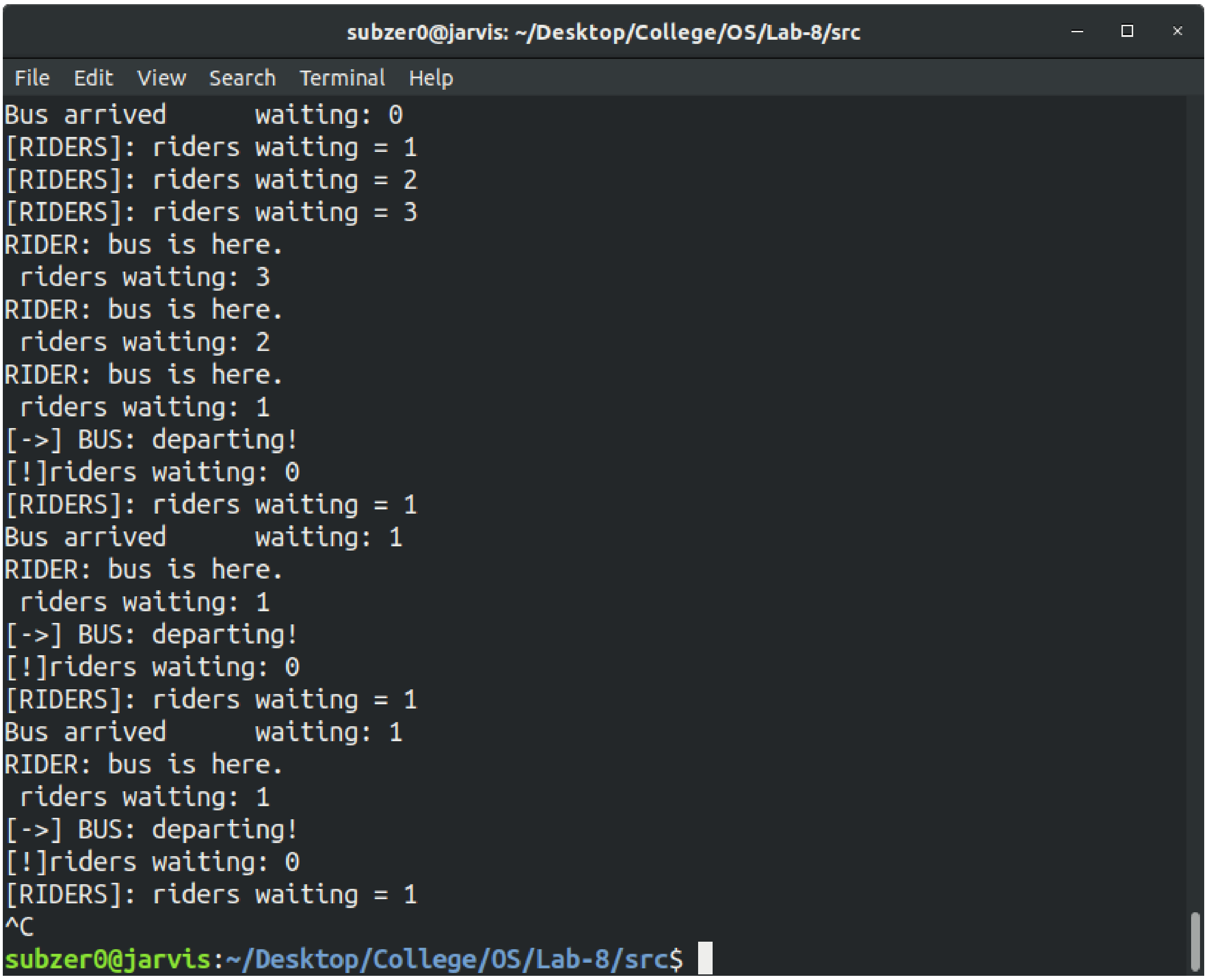
Code:

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

#define MAX_RIDERS 10

pthread_mutex_t mutex;
sem_t riders_waiting; //multiplex
sem_t bus_arrival; //bus
sem_t bus_depart;  //all_aboard
int waiting = 0;

void * rider(){
   while(1){

      sem_wait(&riders_waiting);
      pthread_mutex_lock(&mutex);
      waiting = waiting + 1;
      printf("[RIDERS]: riders waiting = %d \n", waiting);
      sleep(1);
      pthread_mutex_unlock(&mutex);
      sem_wait(&bus_arrival);
      sem_post(&riders_waiting);

      printf("RIDER: bus is here. \n riders waiting: %d \n", waiting--);
      sleep(1);
      if(waiting == 0){
         sem_post(&bus_depart);
      }else{
         sem_post(&bus_arrival);
      }
   }
}

void * bus(){
   while(1){
      sem_wait(&riders_waiting);
      printf("Bus arrived \t waiting: %d\n", waiting);
      pthread_mutex_lock(&mutex);
      if(waiting > 0){
         sem_post(&bus_arrival);
         sem_wait(&bus_depart);
      }
      printf("[->] BUS: departing! \n[!]riders waiting: %d \n", waiting);
      pthread_mutex_unlock(&mutex);
      sleep(1);
   }
}


void main(int argc, char * argv[]){

   pthread_t riders [MAX_RIDERS];
   pthread_t b1;

   pthread_mutex_init(&mutex, NULL);
   sem_init(&riders_waiting, 0, 50);
   sem_init(&bus_arrival, 0, 0);
   sem_init(&bus_depart, 0, 0);

   int t_id;
   pthread_create(&b1, NULL, bus, NULL);
   for(t_id=0; t_id< MAX_RIDERS ;t_id++){
      pthread_create(&riders[t_id], NULL, rider, NULL);
   }

   for(t_id=0; t_id< MAX_RIDERS ;t_id++){
      pthread_join(riders[t_id], NULL);
   }

   pthread_join(b1, NULL);
}
```

Explanation:

We set the maximum number of riders by the *MAX_RIDERS* constant. The global variable waiting is incremented as riders arrive at the station. Once the bus arrives, waiting is decreased for each rider that is at the station. A mutex lock *riders_waiting* is acquired by the bus when it has arrived, hence not allowing more riders to enter the station. *bus_arrival* and *bus_depart* are semaphores that are used to communicate between the riders and the bus to know whether all the riders at the station have entered or not.

Output:



subzer0@jarvis: ~/Desktop/College/OS/Lab-8/src
File Edit View Search Terminal Help

```
Bus arrived      waiting: 0
[RIDERS]: riders waiting = 1
[RIDERS]: riders waiting = 2
[RIDERS]: riders waiting = 3
RIDER: bus is here.
 riders waiting: 3
RIDER: bus is here.
 riders waiting: 2
RIDER: bus is here.
 riders waiting: 1
[->] BUS: departing!
[!]riders waiting: 0
[RIDERS]: riders waiting = 1
Bus arrived      waiting: 1
RIDER: bus is here.
 riders waiting: 1
[->] BUS: departing!
[!]riders waiting: 0
[RIDERS]: riders waiting = 1
Bus arrived      waiting: 1
RIDER: bus is here.
 riders waiting: 1
[->] BUS: departing!
[!]riders waiting: 0
[RIDERS]: riders waiting = 1
^C
subzer0@jarvis:~/Desktop/College/OS/Lab-8/src$
```