OPERATING SYSTEMS PRACTICE (COM301P)

Name: Vinayak Sethi **Roll No:** COE18B061

Assignment 7

(1) Simulate the Producer Consumer code discussed in the class.

Logic:

Producer

```
while (true)
{
// item not produced
while ((in+1) % BS = = out)
; // do nothing as the Buffer is Full
buffer [in] = next-produced-item;
in  = ( in + 1 ) % BS ;
}
```

Consumer

```
while (true)
{
// item not consumed
while (in = = out)
; //do nothing as Buffer is Empty
next-consumed-item = buffer[out];
out  =  (out + 1 ) % BS ;
}
```

```
buffer  - circular array with two pointers / access indices
in  - next free position in the buffer array ;
out  - first full position in the buffer array ;
in == out implies EMPTY; (in+1) % BS = = out implies FULL buffer
states
```

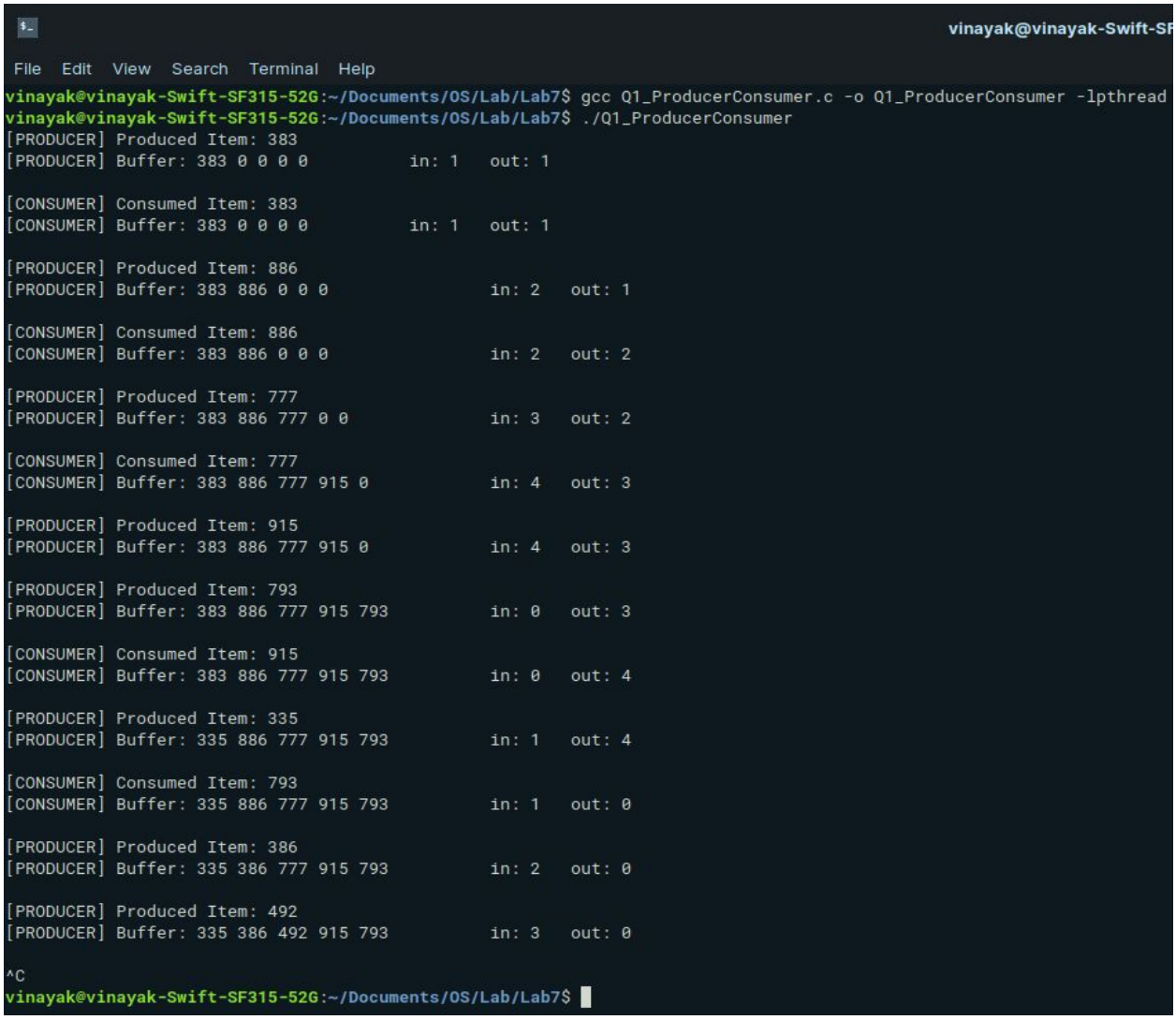
Filename: Q1_ProducerConsumer.c

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

```c
#include<time.h>

#define BS 5
int in = 0;
int out = 0;
int buffer[BS];

void *producer(void *param);
void *consumer(void *param);

int main()
{
   pthread_t pid, cid; //returns the thread id of thread created
   pthread_attr_t attr1, attr2; //to define thread attributes

   //initializes the thread attributes
   pthread_attr_init(&attr1);
   pthread_attr_init(&attr2);

   //creates a new thread
   pthread_create(&pid, &attr1, producer, NULL);
   pthread_create(&cid, &attr2, consumer, NULL);

   //wait for termination of the thread
   pthread_join(pid, NULL);
   pthread_join(cid, NULL);

   return 0;
}

void *producer(void *param)
{
   int next_produced_item;

   while(1)
   {
       next_produced_item = rand() % 1000; //Filling the random numbers
       while((in + 1) % BS == out); //do nothing as buffer is full
       buffer[in] = next_produced_item;
       in = (in + 1) % BS;
       printf("[PRODUCER] Produced Item: %d\n", next_produced_item);
       printf("[PRODUCER] Buffer: %d %d %d %d %d\t\tin: %d\tout: %d\n",
buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out);
```

```
        printf("\n");
        sleep(2);
    }

    pthread_exit(0); //to exit a thread
}

void *consumer(void *param)
{
    int next_consumed_item;

    while(1)
    {
        while(in == out); //do nothing as buffer is empty
        next_consumed_item = buffer[out];
        out = (out + 1) % BS;
        printf("[CONSUMER] Consumed Item: %d\n", next_consumed_item);
        printf("[CONSUMER] Buffer: %d %d %d %d %d\t\tin: %d\tout: %d\n",
buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out);
        printf("\n");
        sleep(3);
    }

    pthread_exit(0); //to exit a thread
}
```

Output:



(2) Extend the producer consumer simulation in Q1 to sync access of critical data using Peterson's algorithm.

Logic:

Peterson's Solution

Peterson's Solution is a classic software-based solution to the critical section problem

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Process Pi structure in Peterson's Solution

Peterson's Solution

```
do {
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);

        critical section

    flag[j] = FALSE;

        remainder section

} while (TRUE);
```

Process Pj structure in Peterson's Solution

Filename: Q2_PetersonsSolutionPC.c

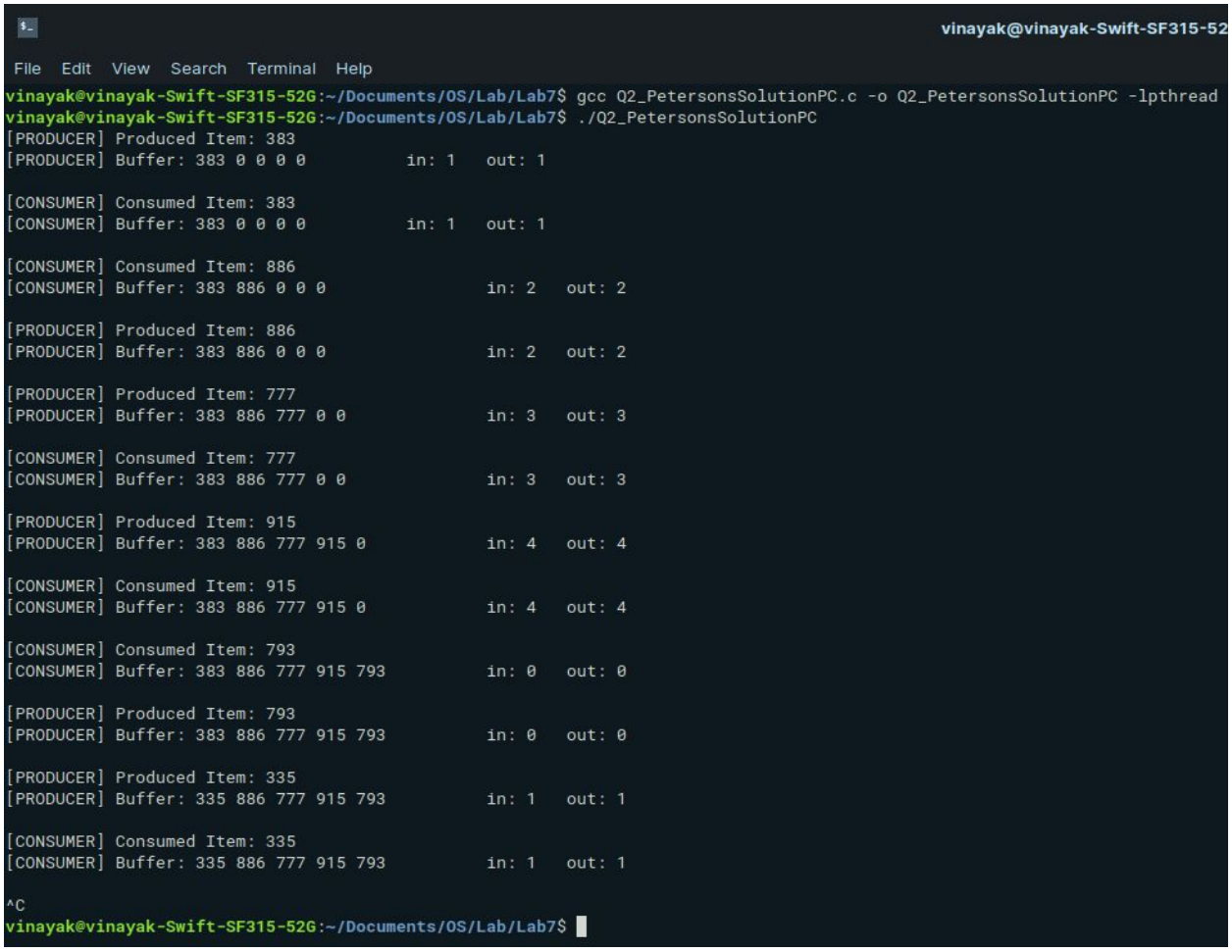
```
#include<stdio.h>
#include<stdlib.h>
```

```c
#include<stdbool.h>
#include<pthread.h>
#include<unistd.h>
#include<time.h>

#define BS 5
int in = 0;
int out = 0;
int buffer[BS];

//0: Producer
//1: Consumer
bool flag[2] = {false, false}; //flag[i] = true means P(i) wants to enter the critical section
int turn = -1; //turn stores which variable is there in critical section

void *producer(void *param);
void *consumer(void *param);

int main()
{
   pthread_t pid, cid; //returns the thread id of thread created
   pthread_attr_t attr1, attr2; //to define thread attributes

   //initializes the thread attributes
   pthread_attr_init(&attr1);
   pthread_attr_init(&attr2);

   //creates a new thread
   pthread_create(&pid, &attr1, producer, NULL);
   pthread_create(&cid, &attr2, consumer, NULL);

   //wait for termination of the thread
   pthread_join(pid, NULL);
   pthread_join(cid, NULL);

   return 0;
}

void *producer(void *param)
{
   do
   {
```

```
        int next_produced_item;
        flag[0] = true;
        turn = 1;

        while(flag[1] && turn == 1); //means consumer is inside Critical
Section

        //Critical Section starts
        next_produced_item = rand() % 1000; //Filling the random numbers
        while((in + 1) % BS == out); //do nothing as buffer is full
        buffer[in] = next_produced_item;
        in = (in + 1) % BS;
        printf("[PRODUCER] Produced Item: %d\n", next_produced_item);
        printf("[PRODUCER] Buffer: %d %d %d %d %d\t\tin: %d\tout: %d\n",
buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out);
        printf("\n");
        sleep(2);

        flag[0] = false; //exit condition from Critical Section
    }while(true);

    pthread_exit(0); //to exit a thread
}

void *consumer(void *param)
{
    do
    {
        int next_consumed_item;
        flag[1] = true;
        turn = 0;

        while(flag[0] && turn == 1); //means producer is inside Critical
Section


        //Critical Section starts
        while(in == out); //do nothing as buffer is empty
        next_consumed_item = buffer[out];
        out = (out + 1) % BS;
        printf("[CONSUMER] Consumed Item: %d\n", next_consumed_item);
        printf("[CONSUMER] Buffer: %d %d %d %d %d\t\tin: %d\tout: %d\n",
buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out);
```

```
        printf("\n");
        sleep(3);

        flag[1] = false; //exit condition from Critical Section
    }while(true);

    pthread_exit(0); //to exit a thread
}
```

Output:



(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.
Note: This can be implemented using either Mutex locks or Peterson's algorithm.

Dictionary used for Q3 and Q4 is:

Filename: Words.txt

Tender;Kind and Loving;Not tough$
Wanting;Not having enough of something;Lacking$
Canvass;To try to persuade people to vote for a particular person or party in an election or to support somebody/something;To find out what people's opinions are about something$
Fleece;The wool coat of a sheep;A piece of clothing like a jacket, made of warm artificial material$
Qualify;To pass the examination that is necessary to do a particular job;To have or give somebody the right to have or do something$
Expansive;(Used about a person) talking a lot in an interesting way;Friendly$
Moment;A very short period of time;Importance or consequence$
Hedge;A row of bushes or trees planted close together at the edge of a garden or field to separate one piece of land from another;To avoid giving a direct answer to a question$
Leave;Go away from;Allow or cause to remain$
Gratuitous;Not necessary, or with no cause;Costing nothing$
Confound;To confuse and very much surprise someone, so that they are unable to explain or deal with a situation;To put to shame$
Scinitillating;Sparkling or shining brightly;Brilliantly and excitingly clever or skilful$
Imbibe;To absorb something, especially information;To drink something, especially alcoholic drinks$
Inundate;To give or send somebody so many things that he/she cannot deal with them all;To cover an area of land with water$
Hefty;Big and strong;Large in amount$
Pelter;A dealer in animal skins or hides;An old, feeble, or inferior horse$
Concretize;Make (an idea or concept) real;Give specific or definite form to$
Manikin;A very short man;A model of the human body used for teaching medical or art students$
Comfort;A state of physical ease and freedom from pain or constraint;The easing or alleviation of a person's feelings of grief or distress$
Jocular;Humorous or amusing;Enjoying making people laugh$

Logic:

Dictionary is created using Peterson's Algorithm, by setting up the flag and turn variable to enter the critical section, and Linear Search is applied for searching the word.

Filename: Q3_DictionaryUsingPeterson.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stdbool.h>
#include<pthread.h>
```

```
//dictionary variables
char word[100][100];
char primary[100][1000];
char secondary[100][1000];
char search_word[100];
int size;

//0: Producer
//1: Consumer
bool flag[2] = {false, false}; //flag[i] = true means P(i) wants to enter
the critical section
int turn = -1; //turn stores which variable is there in critical section

void filesize();
void create_dict();
void *producer(void *param);
void *consumer(void *param);
void search();

int main()
{
   filesize();
   pthread_t pid, cid; //returns the thread id of thread created
   pthread_attr_t attr1, attr2; //to define thread attributes

   //initializes the thread attributes
   pthread_attr_init(&attr1);
   pthread_attr_init(&attr2);

   //creates a new thread
   pthread_create(&pid, &attr1, producer, NULL);
   pthread_create(&cid, &attr2, consumer, NULL);

   //wait for termination of the thread
   pthread_join(pid, NULL);
   pthread_join(cid, NULL);

   return 0;
}

void filesize()
{
   FILE *fp = fopen("Words.txt", "r");
```

```
    char c;
    for(c = getc(fp); c != EOF; c = getc(fp))
    {
        if(c == '\n') //whenever encounters a new line
            size += 1;
    }

    fclose(fp);
}

void create_dict()
{
    FILE *fp = fopen("Words.txt", "r");
    char c = fgetc(fp);
    for(int i=0; i<size; i++)
    {
        memset(word[i], 0, sizeof(word[i]));
        memset(primary[i], 0, sizeof(primary[i]));
        memset(secondary[i], 0, sizeof(secondary[i]));

        //stores word
        int index = 0;
        while (c != ';')
        {
            word[i][index++] = c;
            c = fgetc(fp);
        }

        //stores primary meaning
        index = 0;
        c = fgetc(fp);
        while (c != ';')
        {
            primary[i][index++] = c;
            c = fgetc(fp);
        }

        //stores secondary meaning
        index = 0;
        c = getc(fp);
        while (c != '$')
        {
            secondary[i][index++] = c;
```

```
            c = fgetc(fp);
        }

        //for last ($) and EOF
        for (int j = 0; j < 2; j++)
            c = fgetc(fp);
    }

    fclose(fp);
}

void *producer(void *param)
{
    printf("\n[PRODUCER] Setting up the Dictionary...\n");
    flag[0] = true;
    turn = 1;
    while(flag[1] && turn == 1); //means consumer is inside Critical
Section

    //Critical Section starts
    create_dict();

    flag[0] = false; //exit condition from Critical Section

    pthread_exit(0); //to exit a thread
}

void *consumer(void *param)
{
    printf("[CONSUMER] Enter the word to be searched : ");
    scanf("%s", search_word);

    flag[1] = true;
    turn = 0;

    while(flag[0] && turn == 1); //means producer is inside Critical
Section

    //Critical Section starts
    search();

    flag[1] = false; //exit condition from Critical Section
```

```
    pthread_exit(0); //to exit a thread
}

void search()
{
    int flag = 0;
    for(int i=0; i<size; i++)
    {
        if(strcmp(word[i], search_word) == 0)
        {
            printf("\nWord found in the dictionary by the
Consumer\n\n\tWord:\t\t\t%s\n\tPrimary Meaning:\t%s\n\tSecondary
Meaning:\t%s\n\n", word[i], primary[i], secondary[i]);
            flag = 1;
        }
    }
    if(flag == 0)
            printf("\nWord cannot be searched by Consumer as it is not
present in Dictionary\n\n");
}
```

Output:

```
vinayak@vinayak-Swift-SF315-52G: ~/Docume
File  Edit  View  Search  Terminal  Help
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$ gcc Q3_DictionaryUsingPeterson.c -o Q3_DictionaryUsingPeterson -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$ ./Q3_DictionaryUsingPeterson

[PRODUCER] Setting up the Dictionary...
[CONSUMER] Enter the word to be searched : Imbibe

Word found in the dictionary by the Consumer

        Word:                   Imbibe
        Primary Meaning:        To absorb something, especially information
        Secondary Meaning:      To drink something, especially alcoholic drinks

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$ ./Q3_DictionaryUsingPeterson

[PRODUCER] Setting up the Dictionary...
[CONSUMER] Enter the word to be searched : Hello

Word cannot be searched by Consumer as it is not present in Dictionary

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$
```

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

Logic:

Dictionary is created using Peterson's Algorithm, by setting up the flag and turn variable to enter the critical section and whenever a duplicate word is found in the dictionary, the other copies of the word is removed from dictionary maintaining uniqueness, and Binary Search is applied for searching the word, it is implemented in 2 halves, word search in first half is done by one thread and 2nd half search by another thread.

Filename: Q4_EfficientDictionary.c

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

#define SHARED 1

//peterson algo vars
int flag[2];
int turn;

//sem vars for buffer
sem_t empty, full;

struct node
{
   char search_word[1000];
   int index;
   int half;
};

//dictionary variables
char word[100][1000];
char primary[100][1000];
char secondary[100][1000];
int size;

void lock_ini()
{
   flag[0] = 0;
   flag[1] = 0;
```

```
    turn = 0;
}

void lock(int index)
{
    flag[index] = 1;
    turn = 1 - index;
    while (flag[1 - index] == 1 && turn == 1 - index);
}

void unlock(int index)
{
    flag[index] = 0;
}

void swap_word(int i, int j)
{
    char temp[1000] = "";
    strcpy(temp, word[i]);
    strcpy(word[i], word[j]);
    strcpy(word[j], temp);
}

void swap_p(int i, int j)
{
    char temp[1000] = "";
    strcpy(temp, primary[i]);
    strcpy(primary[i], primary[j]);
    strcpy(primary[j], temp);
}

void swap_s(int i, int j)
{
    char temp[1000] = "";
    strcpy(temp, secondary[i]);
    strcpy(secondary[i], secondary[j]);
    strcpy(secondary[j], temp);
}

void bubblesort(int ch, int start, int end)
{
    int i, j;
    for (i = start; i < end - 1; i++)
```

```
    {
        for (j = start; j < end - (i - start) - 1; j++)
        {

            int choice = (ch == 0) ? (strcmp(word[j], word[j + 1]) < 0) :
(strcmp(word[j], word[j + 1]) > 0);
            if (choice == 1)
            {
                swap_word(j, j + 1);
                swap_p(j, j + 1);
                swap_s(j, j + 1);
            }
        }
    }
}

int binarysearch(int start, int end, char key[1000])
{
    while (start <= end)
    {
        int mid = start + (end - start) / 2;
        if (strcmp(word[mid], key) == 0)
            return mid;
        else if (strcmp(word[mid], key) < 0)
            start = mid + 1;
        else
            end = mid - 1;
    }
    return -1;
}

void create_dict()
{
    FILE *fp = fopen("Words.txt", "r");
    char c = fgetc(fp);

    for (int i = 0; i < size; i++)
    {
        int index = 0;
        while (c != ';')
        {
            word[i][index++] = c;
            c = fgetc(fp);
```

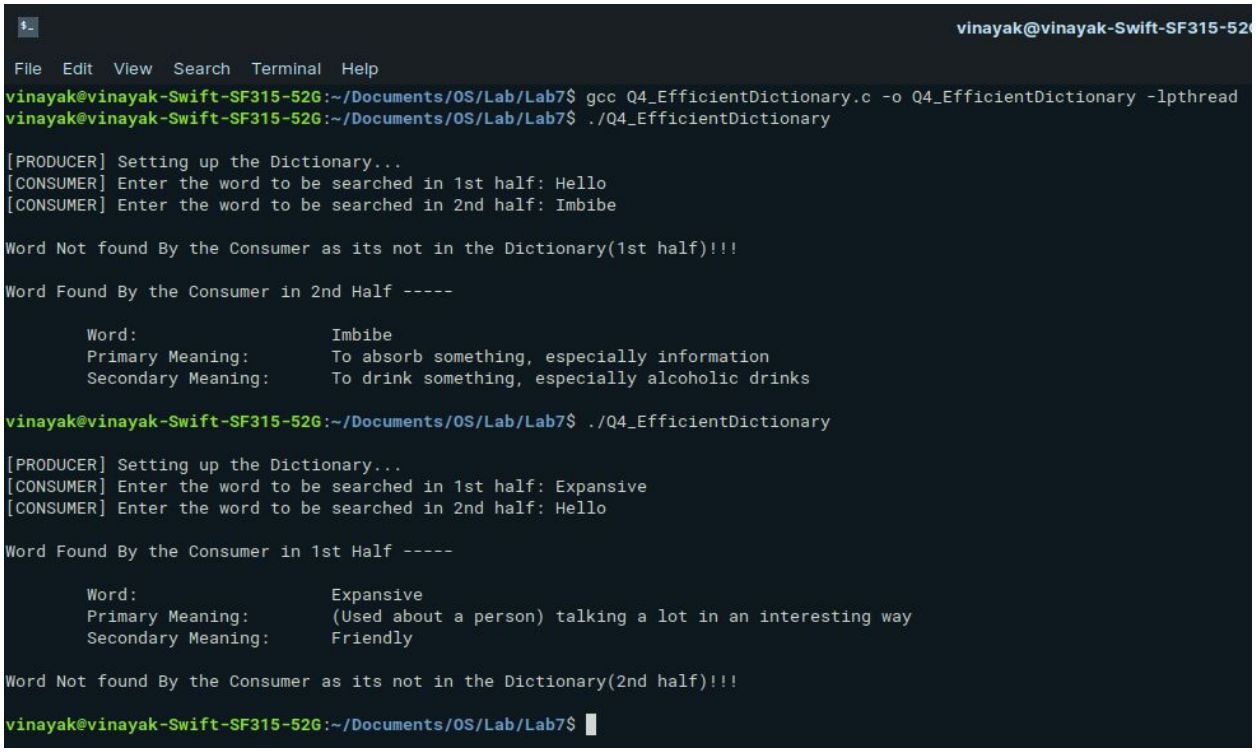
```
        }
        index = 0;
        c = fgetc(fp);
        while (c != ';')
        {
            primary[i][index++] = c;
            c = fgetc(fp);
        }
        index = 0;
        c = getc(fp);
        while (c != '$')
        {
            secondary[i][index++] = c;
            c = fgetc(fp);
        }
        for (int j = 0; j < 2; j++)
            c = fgetc(fp);

        for (int j = 0; j < i; j++)
        {
            //to check for duplicates, if found any remove other copies of
it from Dictionary
            if (strcmp(word[j], word[i]) == 0)
            {
                memset(word[i], 0, sizeof(word[i]));
                memset(primary[i], 0, sizeof(primary[i]));
                memset(secondary[i], 0, sizeof(secondary[i]));
                size -= 1;
                i -= 1;
            }
        }
    }
}

void *producer()
{
    printf("\n[PRODUCER] Setting up the Dictionary...\n");
    sem_wait(&empty);
    //entry-section
    lock(0);
    //cs start
    //assume data to the new producer no
    create_dict();
```

```
    //cs end
    unlock(0);
    //exit section
    sem_post(&full);

    pthread_exit(0); //to exit a thread
}

void *process(void *arg)
{
    struct node *temp = (struct node *)arg;

    if (temp->half == 1)
    {
        bubblesort(1, 0, size / 2);
        temp->index = binarysearch(0, size / 2 - 1, temp->search_word);
    }
    else
    {
        bubblesort(1, size / 2, size);
        temp->index = binarysearch(size / 2, size - 1, temp->search_word);
    }

    pthread_exit(0); //to exit a thread
}

void search()
{
    pthread_t pth[2];

    struct node *keys = (struct node *)malloc(2 * sizeof(struct node));
    //printf("\033[1;31m");
    printf("[CONSUMER] Enter the word to be searched in 1st half: ");
    scanf("%s", keys[0].search_word);

    //printf("\033[1;33m");
    printf("[CONSUMER] Enter the word to be searched in 2nd half: ");
    scanf("%s", keys[1].search_word);
    keys[0].half = 1;
    keys[1].half = 2;

    pthread_create(&pth[0], NULL, process, &keys[0]);
    pthread_create(&pth[1], NULL, process, &keys[1]);
```

```
    for (int i = 0; i < 2; i++)
    {
        pthread_join(pth[i], NULL);
    }
    if (keys[0].index != -1)
    {
        printf("\nWord Found By the Consumer in 1st Half
-----\n\n\tWord:\t\t\t%s\n\tPrimary Meaning:\t%s\n\tSecondary
Meaning:\t%s\n", word[keys[0].index], primary[keys[0].index],
secondary[keys[0].index]);
    }
    else
        printf("\nWord Not found By the Consumer as its not in the
Dictionary(1st half)!!!\n");

    if (keys[1].index != -1)
    {
        printf("\nWord Found By the Consumer in 2nd Half
-----\n\n\tWord:\t\t\t%s\n\tPrimary Meaning:\t%s\n\tSecondary
Meaning:\t%s\n", word[keys[1].index], primary[keys[1].index],
secondary[keys[1].index]);
    }
    else
        printf("\nWord Not found By the Consumer as its not in the
Dictionary(2nd half)!!!\n");
}

void *consumer()
{
    sem_wait(&full);
    //entry-section
    lock(1);
    //cs start
    search();
    //cs end
    unlock(1);
    //exit section
    sem_post(&empty);

    pthread_exit(0); //to exit a thread
}
```

```
void filesize()
{
   FILE *fp = fopen("Words.txt", "r");
   char c;
   for (c = getc(fp); c != EOF; c = getc(fp))
       if(c == '\n') // Increment count if this character is newline
           size += 1;
   fclose(fp);
}

//empty to check whether buffer is empty;default 1
//full to check whether buffer is full;default 0
//lock is mutex lock;default 1

int main()
{
   filesize();

   pthread_t p1, c1; //returns the thread id of thread created

   lock_ini();
   //initializes the unnamed semaphore at the address pointed to
   sem_init(&empty, SHARED, 1);
   sem_init(&full, SHARED, 0);

   //creates a new thread
   pthread_create(&p1, NULL, producer, NULL);
   pthread_create(&c1, NULL, consumer, NULL);

   //wait for termination of the thread
   pthread_join(p1, NULL);
   pthread_join(c1, NULL);

   printf("\n");
   return 0;
}
```

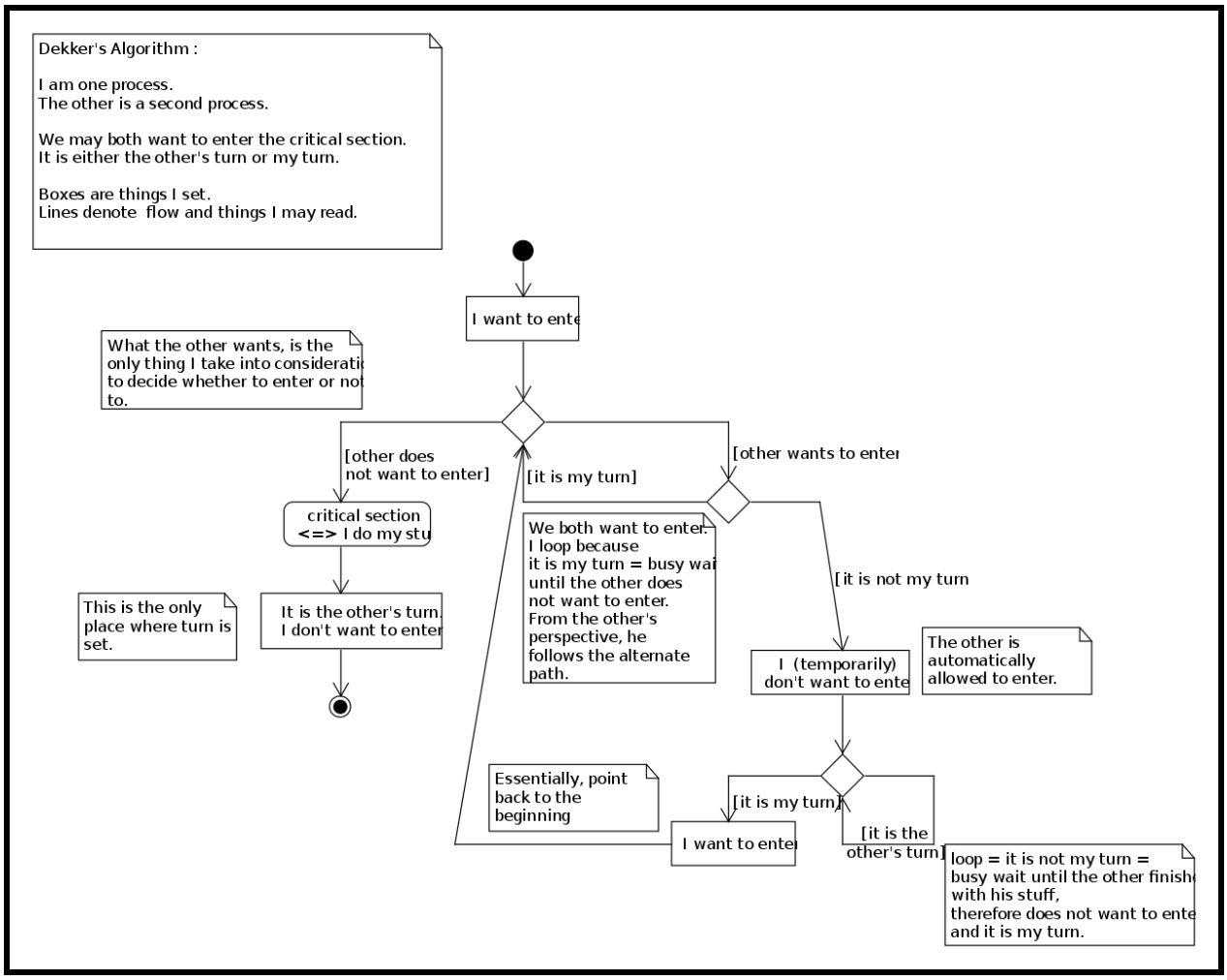
Output:



(5) Trace and understand the working of synchronization algorithms like Dijkstra, Dekker's algorithm.

Logic:

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.



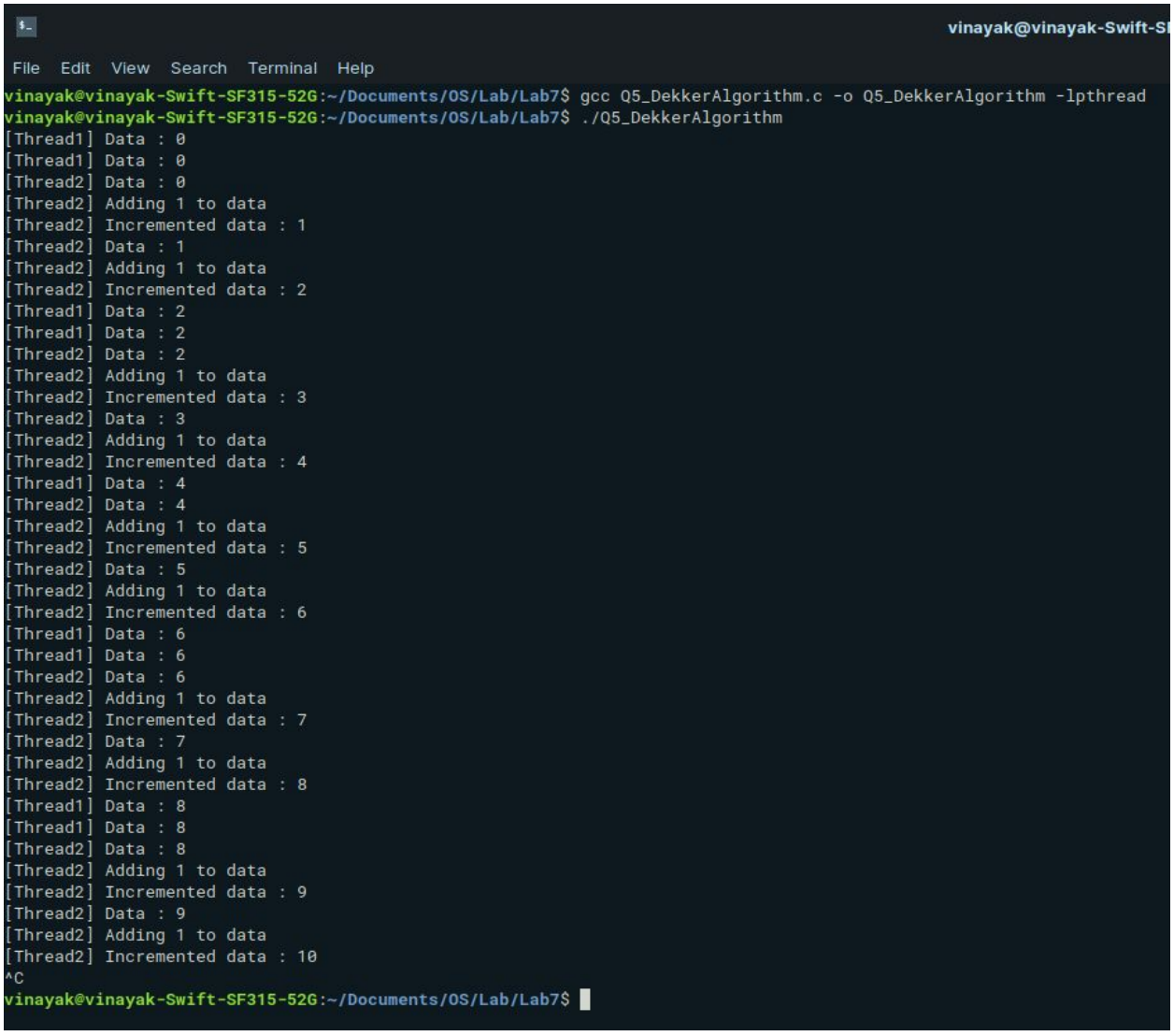
Filename: Q5_DekkerAlgorithm.c

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

// flags to indicate if each thread is in queue to enter its critical
section
bool thread1_wants_to_enter = false;
bool thread2_wants_to_enter = false;

int favoured_thread = 1; // to denote which thread will enter next
int data = 0;

void *runner1(void *param);
void *runner2(void *param);
```

```c
int main()
{
   pthread_t tid[2];

   int ttid[2] = {1,2};

   //creates a new thread
   pthread_create(&tid[0], NULL, runner1, &ttid[0]);
   pthread_create(&tid[1], NULL, runner2, &ttid[1]);

   //wait for termination of the thread
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);

   return 0;
}

void *runner1(void *param)
{
   do
   {
       thread1_wants_to_enter = true;

       while(thread2_wants_to_enter == true)
       {
           if(favoured_thread == 2)
           {
               thread1_wants_to_enter = false;
           }
           while(favoured_thread == 2); // busy waiting
           thread1_wants_to_enter = true;
       }
       //critical section starts
       printf("[Thread1] Data : %d\n", data);
       sleep(3);
       //critical section ends
       favoured_thread = 2;
       thread1_wants_to_enter = false;
   }while(1);
}

void *runner2(void *param)
```

```
{
   do
   {
       thread2_wants_to_enter = true;

       while(thread1_wants_to_enter == true)
       {
           if(favoured_thread == 1)
           {
               thread2_wants_to_enter = false;
           }
           while(favoured_thread == 1); // busy waiting
           thread2_wants_to_enter = true;
       }
       //critical section starts
       printf("[Thread2] Data : %d\n", data);
       printf("[Thread2] Adding 1 to data\n");
       data++;
       printf("[Thread2] Incremented data : %d\n", data);
       sleep(3);
       //critical section ends
       favoured_thread = 1;
       thread2_wants_to_enter = false;
   }while (1);
}
```

Output:



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

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 begins thinking again.

Logic:

The structure of Philosopher *i*:

```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            // eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            // think

} while (TRUE);
```

There are three states of philosopher : THINKING, HUNGRY and EATING. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher.

Filename: Q6_DiningPhilosophers.c

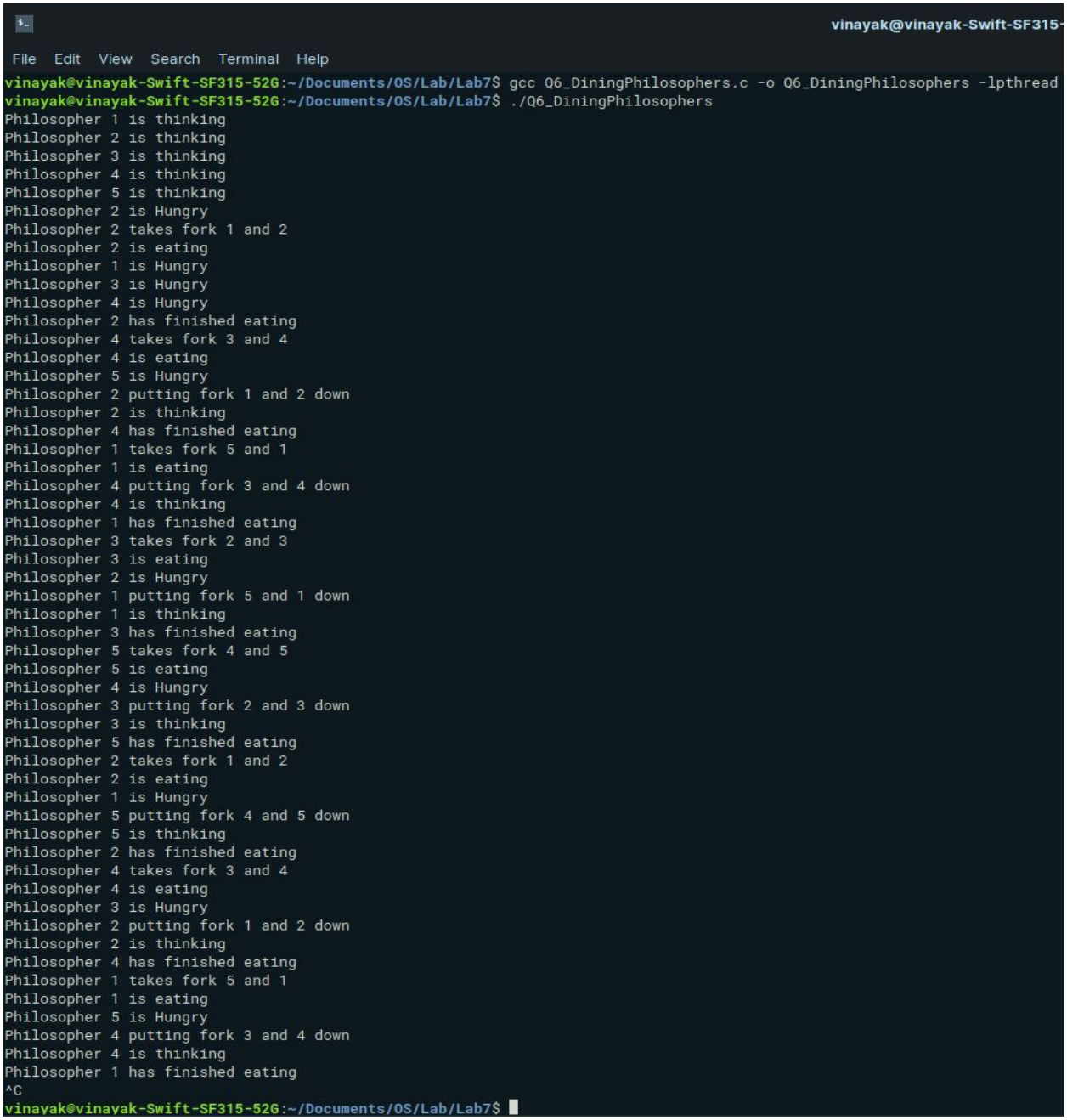
```c
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (ph_num+4)%N
#define RIGHT (ph_num+1)%N

sem_t mutex;
sem_t S[N];

void *philosopher(void *num);
void take_fork(int);
void put_fork(int);
```

```
void test(int);

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

int main()
{
   int i;
   pthread_t thread_id[N]; //returns the thread id of thread created

   //initializes the unnamed semaphore at the address pointed to
   sem_init(&mutex, 0, 1);
   for(int i = 0; i < N; i++)
       sem_init(&S[i], 0, 0);

   //creates a new thread
   for(int i = 0; i < N; i++)
   {
       pthread_create(&thread_id[i], NULL, philosopher, &phil_num[i]);
       printf("Philosopher %d is thinking\n", i+1);
   }

   //wait for termination of the thread
   for(int i = 0; i < N; i++)
       pthread_join(thread_id[i], NULL);

   //destroys the unnamed semaphore at the address pointed to
   sem_destroy(&mutex);
   for(int i = 0; i < N; i++)
       sem_destroy(&S[i]);

   return 0;
}

void *philosopher(void *num)
{
   while(1)
   {
       int *i = num;
       sleep(1);
       take_fork(*i);
       printf("Philosopher %d has finished eating\n", *i+1);
       put_fork(*i);
```

```
    }

    pthread_exit(0); //to exit a thread
}

void take_fork(int ph_num)
{
    sem_wait(&mutex); // so that state update remains critical section
    state[ph_num] = HUNGRY;
    printf("Philosopher %d is Hungry\n", ph_num + 1);
    test(ph_num);
    sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
    sem_wait(&S[ph_num]); //decrements(locks) the semaphore pointed to by
S[ph_num]
    sleep(1);
}

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 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]); //increments(unlocks) the semaphore pointed
to by S[ph_num]
    }
}

void put_fork(int ph_num)
{
    sem_wait(&mutex); //decrements(locks) the semaphore pointed to by mutex
    state[ph_num] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n", ph_num + 1, LEFT
+ 1, ph_num + 1);
    printf("Philosopher %d is thinking\n", ph_num + 1);
    test(LEFT);
    test(RIGHT);
```

```
   sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
}
```

Output:



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

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem.**

Logic:

Writers Section

```
do {
    // writer requests for critical section
    wait(wrt);

    // performs the write

    // leaves the critical section
    signal(wrt);

} while(true);
```

Reader Section

```
do {

   // Reader wants to enter the critical section
   wait(mutex);

   // The number of readers has now increased by 1
   readcnt++;

   // there is atleast one reader in the critical section
   // this ensure no writer can enter if there is even one reader
   // thus we give preference to readers here
   if (readcnt==1)
      wait(wrt);

   // other readers can enter while this current reader is inside
   // the critical section
   signal(mutex);

   // current reader performs reading here
   wait(mutex);   // a reader wants to leave

   readcnt--;

   // that is, no reader is left in the critical section,
   if (readcnt == 0)
       signal(wrt);         // writers can enter

   signal(mutex); // reader leaves

} while(true);
```

Filename: Q7_ReaderWriter.c

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

sem_t mutex, writeblock;

int data = 0, rcount = 0;

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

int main()
{
   int i;
   pthread_t rtid[3], wtid[3]; //returns the thread id of thread created

   //initializes the unnamed semaphore at the address pointed to
   sem_init(&mutex, 0, 1);
   sem_init(&writeblock, 0, 1);

   int twid[3] = {0,1,2};
   int trid[3] = {0,1,2};

   //creates a new thread
   for(int i = 0; i <= 2; i++)
   {
       pthread_create(&wtid[i], NULL, writer, &twid[i]);
       pthread_create(&rtid[i], NULL, reader, &trid[i]);
   }

   //wait for termination of the thread
   for(int i = 0; i <= 2; i++)
   {
       pthread_join(wtid[i], NULL);
       pthread_join(rtid[i], NULL);
   }

   //destroys the unnamed semaphore at the address pointed to
   sem_destroy(&mutex);
   sem_destroy(&writeblock);
```

```
   return 0;
}

void *reader(void *arg)
{
   int *f = arg;

   sem_wait(&mutex); //decrements(locks) the semaphore pointed to by mutex
   rcount = rcount + 1;

   if(rcount == 1) // first reader
       sem_wait(&writeblock); //decrements(locks) the semaphore pointed to
by writeblock

   sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex

   printf("Data read by the reader %d is %d\n", *f, data);

   sem_wait(&mutex); //decrements(locks) the semaphore pointed to by mutex
   rcount = rcount - 1;

   if(rcount == 0) // last reader
       sem_post(&writeblock); //increments(unlocks) the semaphore pointed
to by writeblock
   sleep(3);

   sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex

   pthread_exit(0); //to exit a thread
}

void *writer(void *arg)
{
   int *f = arg;

   sem_wait(&writeblock); //decrements(locks) the semaphore pointed to by
writeblock

   data++;
```

```
   printf("Data written by the writer %d is %d\n", *f, data);
   sleep(3);

   sem_post(&writeblock); //increments(unlocks) the semaphore pointed to
by writeblock

   pthread_exit(0); //to exit a thread
}
```

Output:

(8) Implement semaphores based solutions for the Santa Claus Problem whose details are available in the Downy Book on Semaphores.

This problem is from William Stallings's Operating Systems [11], but he attributes it to John Trono of St. Michael's College in Vermont.

Stand Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not

want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Here are some addition specifications:
• After the ninth reindeer arrives, Santa must invoke prepareSleigh, and then all nine reindeer must invoke getHitched.
• After the third elf arrives, Santa must invoke helpElves. Concurrently, all three elves should invoke getHelp.
• All three elves must invoke getHelp before any additional elves enter (increment the elf counter).

Logic and Pseudocode:

Listing 5.12: Santa problem hint

```
1 elves = 0
2 reindeer = 0
3 santaSem = Semaphore(0)
4 reindeerSem = Semaphore(0)
5 elfTex = Semaphore(1)
6 mutex = Semaphore(1)
```

elves and reindeer are counters, both protected by mutex. Elves and reindeer get mutex to modify the counters; Santa gets it to check them. Santa waits on santaSem until either an elf or a reindeer signals him. The reindeer wait on reindeerSem until Santa signals them to enter the paddock and get hitched. The elves use elfTex to prevent additional elves from entering while three elves are being helped.

Listing 5.13: Santa problem solution (Santa)

```
1 santaSem.wait()
2 mutex.wait()
3     if reindeer == 9:
4         prepareSleigh()
5         reindeerSem.signal(9)
6     else if elves == 3:
7         helpElves()
8 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.

Listing 5.14: Santa problem solution (reindeer)

```
1  mutex.wait()
2      reindeer += 1
3      if reindeer == 9:
4          santaSem.signal()
5  mutex.signal()
6
7  reindeerSem.wait()
8  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.

Listing 5.15: Santa problem solution (elves)

```
1  elfTex.wait()
2  mutex.wait()
3      elves += 1
4      if elves == 3:
5          santaSem.signal()
6      else
7          elfTex.signal()
8  mutex.signal()
9
10 getHelp()
11
12 mutex.wait()
13     elves -= 1
14     if elves == 0:
15        elfTex.signal()
16 mutex.signal()
```

The first two elves release elfTex at the same time they release the mutex, but the last elf holds elfTex, barring other elves from entering until all three elves have invoked getHelp.
The last elf to leave releases elfTex, allowing the next batch of elves to enter.

Filename: Q8_SantaClausProblem.c

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

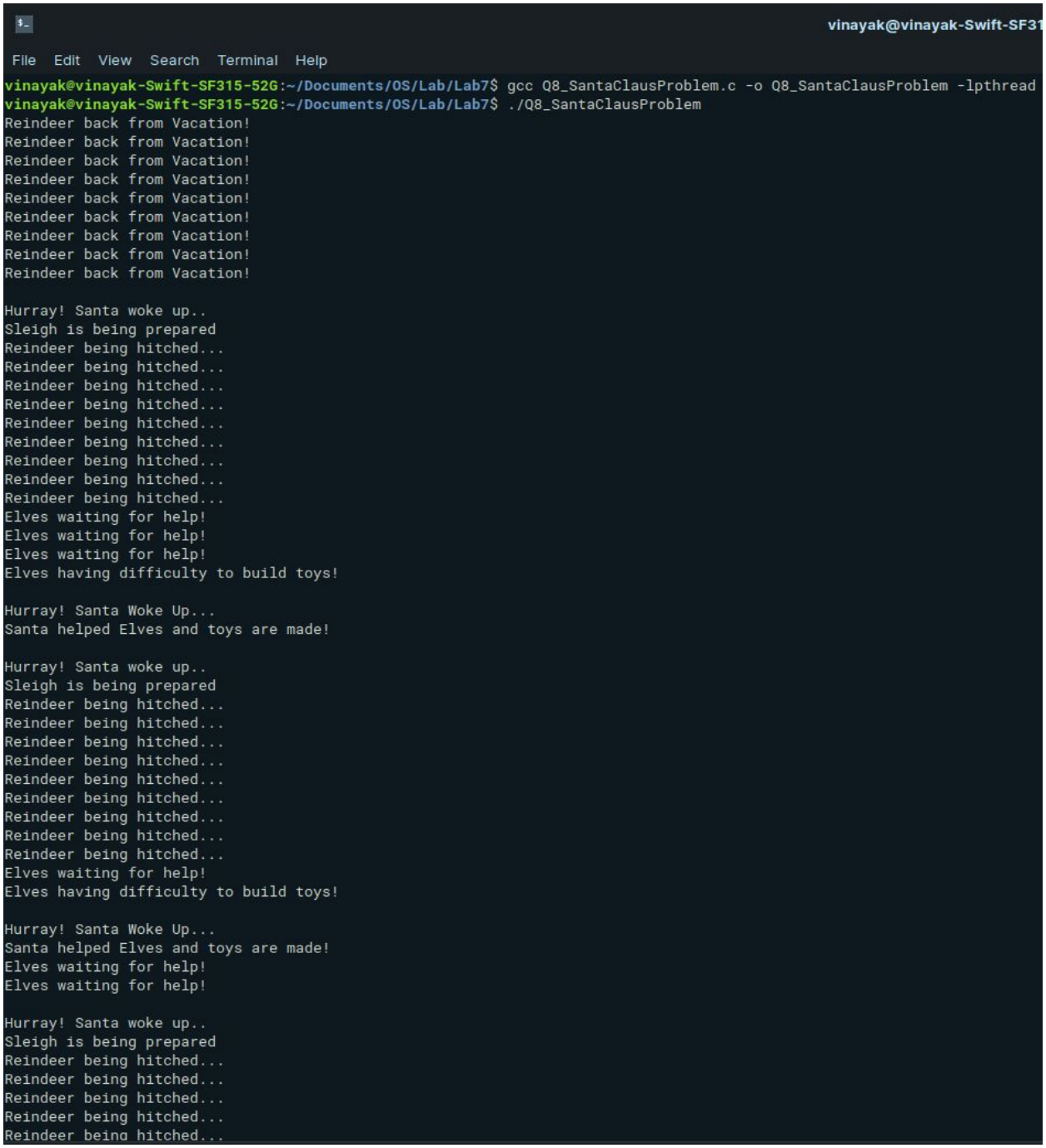
```c
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>

#define SHARE 1 //means semaphore is shared between processes

int elves_count = 0;
int reindeer_count = 0;
sem_t santa_sem, reindeer_sem, elves_sem, mutex;

void *santa(void *arg);
void *reindeer(void *arg);
void *elves(void *arg);

int main()
{
   pthread_t santathread, elfthread[3], reindeerthread[9]; //returns the
thread id of thread created

   //initializes the unnamed semaphore at the address pointed to
   sem_init(&santa_sem, SHARE, 0);
   sem_init(&reindeer_sem, SHARE, 0);
   sem_init(&elves_sem, SHARE, 1);
   sem_init(&mutex, SHARE, 1);

   //creates a new thread
   pthread_create(&santathread, NULL, santa, NULL);
   for(int i=0; i<9; i++)
       pthread_create(&reindeerthread[i], NULL, reindeer, NULL);
   for(int i=0; i<3; i++)
       pthread_create(&elfthread[i], NULL, elves, NULL);

   //wait for termination of the thread
   pthread_join(santathread, NULL);
   for(int i=0; i<9; i++)
       pthread_join(reindeerthread[i], NULL);
   for(int i=0; i<3; i++)
       pthread_join(elfthread[i], NULL);

   //destroys the unnamed semaphore at the address pointed to
   sem_destroy(&santa_sem);
   sem_destroy(&reindeer_sem);
```

```
    sem_destroy(&elves_sem);
    sem_destroy(&mutex);

    printf("\n");
    return 0;
}

void *santa(void *arg)
{
    while(1)
    {
        sem_wait(&santa_sem); //decrements(locks) the semaphore pointed to
by santa_sem
        sem_wait(&mutex); //decrements(locks) the semaphore pointed to by
mutex

        if(reindeer_count == 9)
        {
            printf("\nHurray! Santa woke up..\n");
            printf("Sleigh is being prepared\n");

            reindeer_count = 0;

            for(int i=0; i<9; i++)
                sem_post(&reindeer_sem); //increments(unlocks) the
semaphore pointed to by reindeer_sem
        }
        else if(elves_count == 3)
        {
            sleep(1);
            printf("Elves having difficulty to build toys!\n");
            printf("\nHurray! Santa Woke Up...\n");
            printf("Santa helped Elves and toys are made!\n");
        }

        sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
    }

    pthread_exit(0); //to exit a thread
}

void *reindeer(void *arg)
```

```
{
   printf("Reindeer back from Vacation!\n");
   while(1)
   {
       sem_wait(&mutex); //decrements(locks) the semaphore pointed to by
mutex
       reindeer_count += 1;
       if(reindeer_count == 9)
           sem_post(&santa_sem); //increments(unlocks) the semaphore
pointed to by santa_sem
       sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex

       sem_wait(&reindeer_sem); //decrements(locks) the semaphore pointed
to by reindeer_sem
       printf("Reindeer being hitched...\n");
       sleep(1);
   }

   pthread_exit(0); //to exit a thread
}

void *elves(void *arg)
{
   while(1)
   {
       sem_wait(&elves_sem); //decrements(locks) the semaphore pointed to
by elves_sem
       sem_wait(&mutex); //decrements(locks) the semaphore pointed to by
mutex

       elves_count += 1;

       if(elves_count == 3)
           sem_post(&santa_sem); //increments(unlocks) the semaphore
pointed to by santa_sem
       else
           sem_post(&elves_sem); //increments(unlocks) the semaphore
pointed to by elves_sem

       sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
```

```
        sleep(1);
        printf("Elves waiting for help!\n");
        sleep(1);

        sem_wait(&mutex); //decrements(locks) the semaphore pointed to by
mutex
        elves_count -= 1;
        if(elves_count == 0)
            sem_post(&elves_sem); //increments(unlocks) the semaphore
pointed to by elves_sem
        sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
    }

    pthread_exit(0); //to exit a thread
}
```

Output:



(9) Implement semaphores based solutions for the Dining Hall Problem whose details are available in the Downy Book on Semaphores.

This problem was written by Jon Pollack during my Synchronization class at Olin College.
Students in the dining hall invoke dine and then leave. After invoking dine and before invoking leave a student is considered “ready to leave”. The synchronization constraint that applies to students is that, in order to

maintain the illusion of social suave, a student may never sit at a table alone. A student is considered to be sitting alone if everyone else who has invoked dine invokes leave before she has finished dine.

Logic and Pseudocode:

Listing 7.27: Dining Hall problem hint

```
1  eating = 0
2  readyToLeave = 0
3  mutex = Semaphore(1)
4  okToLeave = Semaphore(0)
```

eating and readyToLeave are counters protected by mutex, so this is the usual scoreboard pattern. If a student is ready to leave, but another student would be left alone at the table, she waits on okToLeave until another student changes the situation and signals.

Listing 7.28: Dining Hall problem solution

```
1  getFood()
2
3  mutex.wait()
4  eating++
5  if eating == 2 and readyToLeave == 1:
6      okToLeave.signal()
7      readyToLeave--
8  mutex.signal()
9
10 dine()
11
12 mutex.wait()
13 eating--
14 readyToLeave++
15
16 if eating == 1 and readyToLeave == 1:
17     mutex.signal()
18     okToLeave.wait()
19 elif eating == 0 and readyToLeave == 2:
20     okToLeave.signal()
21     readyToLeave -= 2
22     mutex.signal()
23 else:
24     readyToLeave--
25     mutex.signal()
26
27 leave()
```

When a student is checking in, if she sees one student eating and one waiting to leave, she lets the waiter off the hook and decrements readyToLeave for him.

After dining, the student checks three cases:
• If there is only one student left eating, the departing student has to give up the mutex and wait.
• If the departing student finds that someone is waiting for her, she signals him and updates the counter for both of them.
• Otherwise, she just decrements readyToLeave and leaves.

Filename: Q9_DiningHall.c

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

#define SHARE 0 //means semaphore is shared between the threads of a
process
#define DINE_CAPACITY 5

int eating = 0;
int readyToLeave = 0;
sem_t mutex, okToLeave;

void *dininghall(void *arg);

int main()
{
   pthread_t tid[DINE_CAPACITY]; //returns the thread id of thread created

   //initializes the unnamed semaphore at the address pointed to
   sem_init(&mutex, SHARE, 1);
   sem_init(&okToLeave, SHARE, 0);

   int temp[DINE_CAPACITY];
   for(int i=0; i<DINE_CAPACITY; i++)
       temp[i] = i+1;

   //creates a new thread
   for(int i=0; i<DINE_CAPACITY; i++)
       pthread_create(&tid[i], NULL, dininghall, &temp[i]);

   //wait for termination of the thread
```

```
    for(int i=0; i<DINE_CAPACITY; i++)
        pthread_join(tid[i], NULL);

    //destroys the unnamed semaphore at the address pointed to
    sem_destroy(&mutex);
    sem_destroy(&okToLeave);

    return 0;
}

void *dininghall(void *arg)
{
    int *param = arg;
    printf("Diner %d entered the hall and ready to eat\n", *param);

    sem_wait(&mutex); //decrements(locks) the semaphore pointed to by mutex
    eating ++;
    printf("Diner %d is eating\n", *param);
    if(eating == 2 && readyToLeave == 1)
    {
        sem_post(&okToLeave); //increments(unlocks) the semaphore pointed
to by okToLeave
        readyToLeave --;
    }
    sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
    sleep(3);

    sem_wait(&mutex); //decrements(locks) the semaphore pointed to by mutex
    eating--;
    readyToLeave++;
    printf("Diner %d is ready to leave\n", *param);

    if(eating == 1 && readyToLeave == 1)
    {
        sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
        sem_wait(&okToLeave); //increments(unlocks) the semaphore pointed
to by okToLeave
    }
    else if(eating == 0 && readyToLeave == 2)
    {
```

```
        sem_post(&okToLeave); //increments(unlocks) the semaphore pointed
to by okToLeave
        readyToLeave -= 2;
        sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
    }
    else
    {
        readyToLeave--;
        sem_post(&mutex); //increments(unlocks) the semaphore pointed to by
mutex
    }

    printf("Diner %d has left the hall\n", *param);

    pthread_exit(0); //to exit a thread
}
```

Output:

```
vinayak@vina
File  Edit  View  Search  Terminal  Help
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$ gcc Q9_DiningHall.c -o Q9_DiningHall -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$ ./Q9_DiningHall
Diner 1 entered the hall and ready to eat
Diner 1 is eating
Diner 2 entered the hall and ready to eat
Diner 2 is eating
Diner 3 entered the hall and ready to eat
Diner 3 is eating
Diner 4 entered the hall and ready to eat
Diner 4 is eating
Diner 5 entered the hall and ready to eat
Diner 5 is eating
Diner 1 is ready to leave
Diner 1 has left the hall
Diner 2 is ready to leave
Diner 2 has left the hall
Diner 4 is ready to leave
Diner 4 has left the hall
Diner 3 is ready to leave
Diner 5 is ready to leave
Diner 5 has left the hall
Diner 3 has left the hall
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab7$
```