Operating System
COM301P

Programming Assignment
Lab - 7

By :

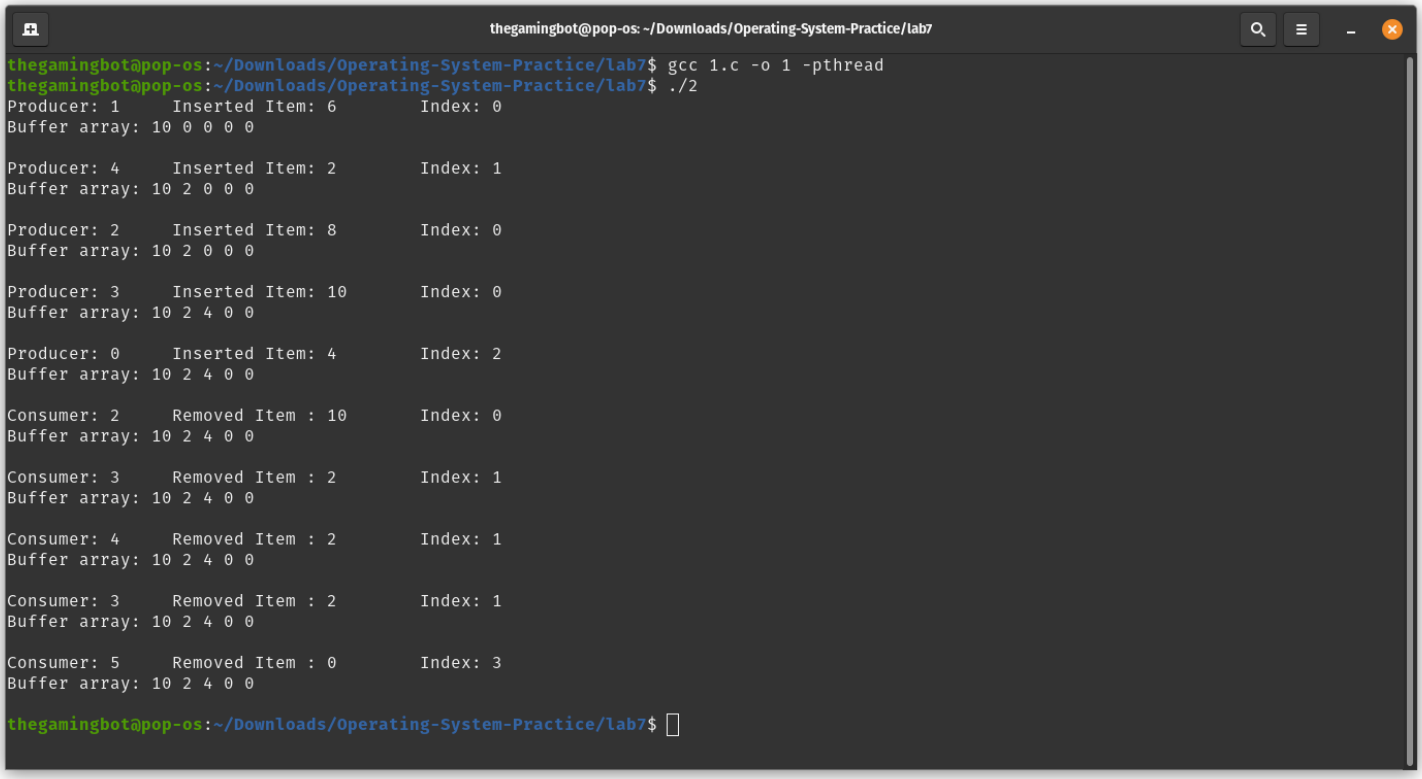
Sai Kaushik S
CED18I044

Question: Simulate the Producer Consumer code discussed in the class.

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>
#include <unistd.h>
// Define the macros
#define BufferSize 5
// Global constants
int in = 0, out = 0;
int buffer[BufferSize];
// Function declarations
void* producer(void *params);
void* consumer(void *params);
void printBuffer();
// Main driver function
int main(){
   // Initialize the mutex and semaphores
   pthread_t ptid[BufferSize], ctid[BufferSize];
   // Create threads for producer and consumer
   for(int i = 0; i < BufferSize; i++)
       pthread_create(&ptid[i], NULL, (void *)producer, (void *)&i);
   for(int i = 0; i < BufferSize; i++)
       pthread_create(&ctid[i], NULL, (void *)consumer, (void *)&i);
   // Wait for the threads to complete
   for(int i = 0; i < BufferSize; i++)
       pthread_join(ptid[i], NULL);
   for(int i = 0; i < BufferSize; i++)
       pthread_join(ctid[i], NULL);
   return 0;
}
// A function for the producer
void* producer(void *params){
   // Get a random number in the range of 1 and 10
   int item = rand() % 10 + 1;
   // Add it to the buffer
   buffer[in] = item;
   printf("Producer: %d\tInserted Item: %d\tIndex: %d\n", *((int *)params), item, in);
   printBuffer();
```

```
    // Increment the buffer
    // Go around if pointer overflows
    in = (in + 1) % BufferSize;
    sleep(2);
}
// A function for the consumer
void* consumer(void *params){
    // Consume the buffer element
    int item = buffer[out];
    printf("Consumer: %d\tRemoved Item : %d\tIndex: %d\n", *((int *)params), item, out);
    printBuffer();
    // Increment the buffer
    // Go around if pointer overflows
    out = (out + 1) % BufferSize;
    sleep(2);
}
// A function to print the buffer
void printBuffer(){
    printf("Buffer array: ");
    for(int i = 0; i < BufferSize; i++)
        printf("%d ", buffer[i]);
    printf("\n\n");
}
```

Output:



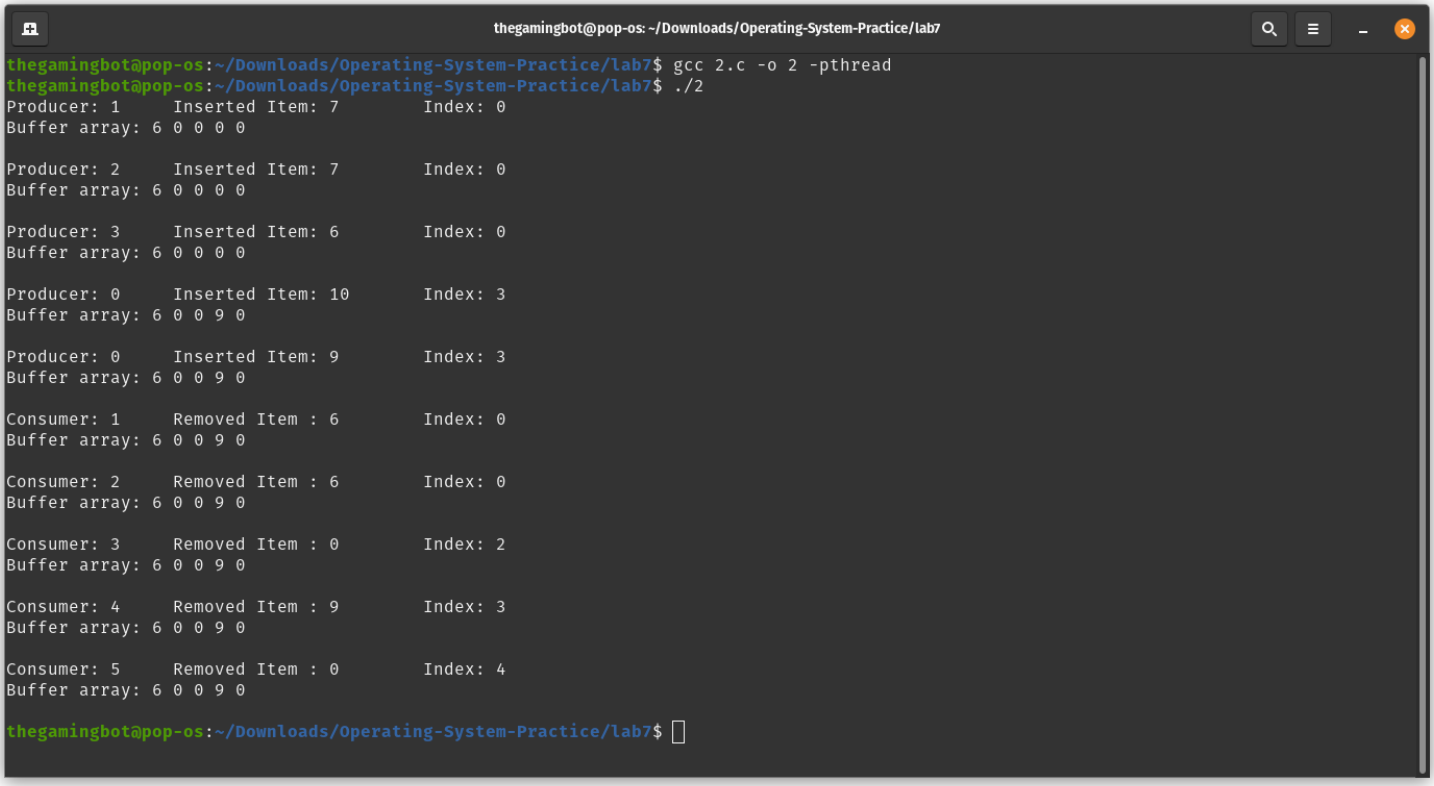
Question: Extend the producer consumer simulation in Q1 to sync access of critical data using Peterson's Algorithm.

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

// Define the macros
#define BufferSize 5
#define Producer 0
#define Consumer 1

// Global constants
int in = 0, out = 0;
int buffer[BufferSize];
int flag[2] = {0, 0};
int turn = 0;

// Function declarations
void* producer(void *params);
void* consumer(void *params);
void printBuffer();

// Main driver function
int main(){
   srand(time(0));
   // Initialize the mutex and semaphores
   pthread_t ptid[BufferSize], ctid[BufferSize];
   // Create threads for producer and consumer
   for(int i = 0; i < BufferSize; i++)
       pthread_create(&ptid[i], NULL, (void *)producer, (void *)&i);
   for(int i = 0; i < BufferSize; i++)
       pthread_create(&ctid[i], NULL, (void *)consumer, (void *)&i);
   // Wait for the threads to complete
   for(int i = 0; i < BufferSize; i++)
       pthread_join(ptid[i], NULL);
   for(int i = 0; i < BufferSize; i++)
```

```
        pthread_join(ctid[i], NULL);
    return 0;
}

// A function for the producer
void* producer(void *params){
    // Entry Section
    flag[Producer] = 1;
    turn = Consumer;
    // Busy Wait
    while((flag[Consumer] ==1) && (turn==Consumer));
    // Critical Section
    // Get a random number in the range of 1 and 10
    int item = rand() % 10 + 1;
    // Add it to the buffer
    buffer[in] = item;
    printf("Producer: %d\tInserted Item: %d\tIndex: %d\n", *((int *)params), item, in);
    printBuffer();
    // Increment the buffer
    // Go around if pointer overflows
    in = (in + 1) % BufferSize;
    // Exit Section
    flag[Producer] = 0;
    sleep(2);
}

// A function for the consumer
void* consumer(void *params){
    // Entry Section
    flag[Consumer] = 1;
    turn = Producer;
    // Busy Wait
    while((flag[Producer]==1) && (turn==Producer));
    // Critical Section
    // Consume the buffer element
    int item = buffer[out];
    printf("Consumer: %d\tRemoved Item : %d\tIndex: %d\n", *((int *)params), item, out);
    printBuffer();
    // Increment the buffer
    // Go around if pointer overflows
    out = (out + 1) % BufferSize;
    // Exit Section
```

```
    flag[Consumer] = 0;
    sleep(2);
}

void printBuffer(){
    printf("Buffer array: ");
    for(int i = 0; i < BufferSize; i++)
        printf("%d ", buffer[i]);
    printf("\n\n");
}
```

Output:



Question:

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

```c
// Include the required libraries
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <stdlib.h>

// Structure for the dictionary node
struct node{
   char* word;
   char* primary;
   char* secondary;
   struct node* next;
};

// Structure for the binary search input
struct data{
   char* value;
   struct node* start;
   struct node* end;
};

// Global variables
struct node* root = NULL, *output = NULL;
int size = 0;
pthread_mutex_t lock;

// Function declaration
struct node* birth();
void sortedInsertion(struct node** head, struct node* new);
void removeDuplicates();
void findSize();
```

```c
void createDictionary();
struct node* middle(struct node* start, struct node* last);
void* binarySearch(void* params);
struct node* search(char* data);
void *producer();
void *consumer();

// Main driver function
int main(){
   // Initialize the mutex
   pthread_mutex_init(&lock, NULL);
   pthread_t ptid, ctid;
   // Create threads for producer and consumer
   pthread_create(&ptid, NULL, producer, NULL);
   pthread_create(&ctid, NULL, consumer, NULL);
   // Wait for the threat execution
   pthread_join(ptid, NULL);
   pthread_join(ctid, NULL);
   // Destroy the mutex
   pthread_mutex_destroy(&lock);
   return 0;
}

// A function to create a new node
struct node* birth(){
   struct node* temp = (struct node *)malloc(sizeof(struct node));
   temp->word = (char*) malloc(sizeof(char) * 40);
   temp->primary = (char*) malloc(sizeof(char) * 1000);
   temp->secondary = (char*) malloc(sizeof(char) * 1000);
   temp->next = NULL;
   return temp;
}

// A function to insert a node in sorted order
void sortedInsertion(struct node** head, struct node* new){
   struct node* temp;
   // Special case for the head
   if (*head == NULL || strcmp((*head)->word, new->word) > 0) {
       new->next = *head;
       *head = new;
   }
   // If not head
```

```
    else {
        temp = *head;
        // Locate the node before the point of insertion
        while (temp->next != NULL && strcmp(temp->next->word, new->word) < 0)
            temp = temp->next;
        // Insert in the next position
        new->next = temp->next;
        temp->next = new;
    }
}

// A function to remove duplicates in a sorted linked list
void removeDuplicates(){
    struct node* current = root;
    struct node* next;
    if (current == NULL)
        return;
    // For each element
    while (current->next != NULL){
        // Compare with next node (as it is sorted)
        if (current->word == current->next->word){
            next = current->next->next;
            free(current->next);
            current->next = next;
        }
        // No match
        else
            // Go to next node
            current = current->next;
    }
}

// A function to find the number of entries for the dictionary
void findSize(){
    FILE *fp=fopen("dict.txt", "r");
    char c;
    // Increment for each \n
    for (c = getc(fp); c != EOF; c = getc(fp))
        if (c == '\n')
            size += 1;
    fclose(fp);
}
```

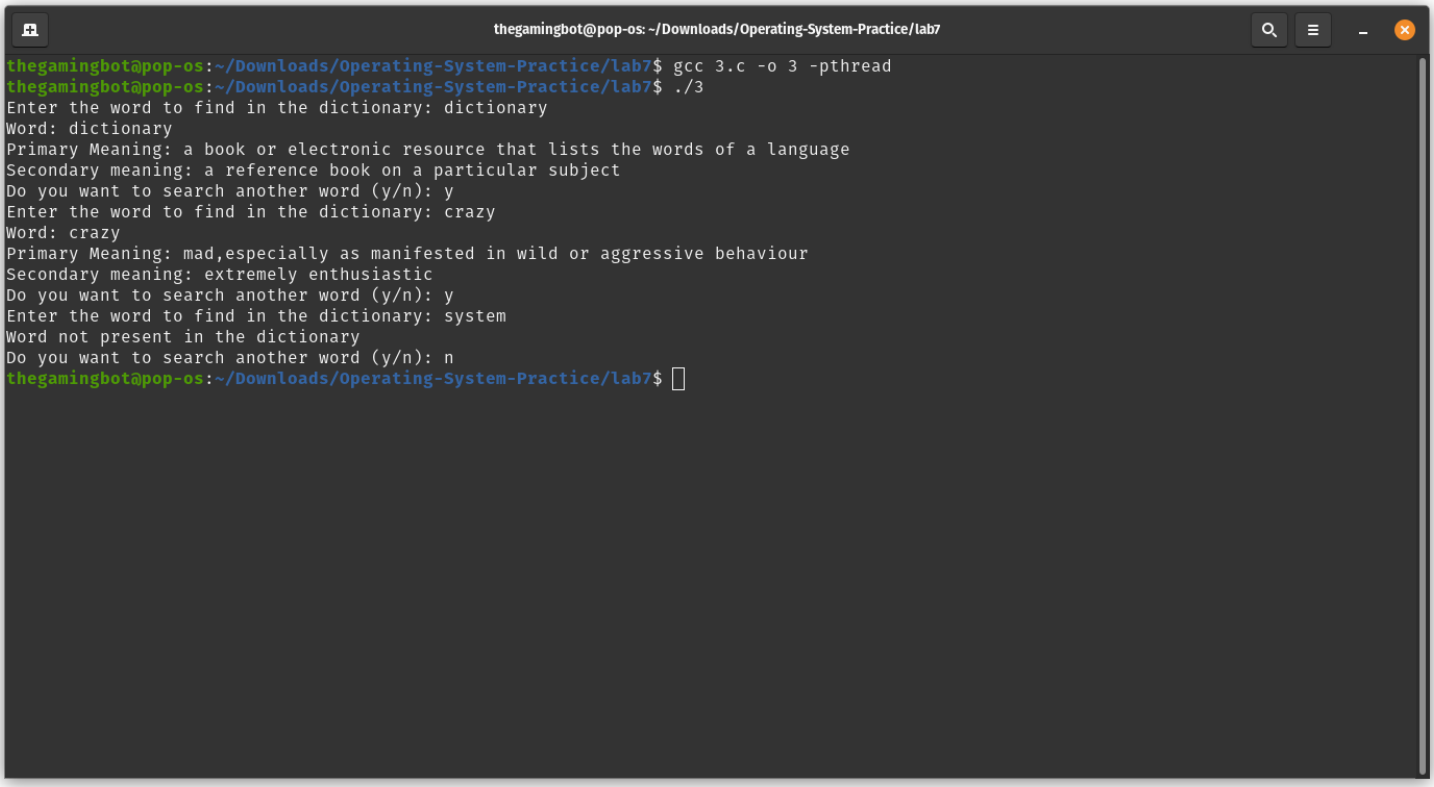
```
// A function to create a dictionary
void createDictionary(){
   // Get the number of lines
   findSize();
   FILE *fp = fopen("dict.txt", "r");
   char c = fgetc(fp);
   // Loop through the lines in the file
   for(int i = 0; i < size; i++){
       // Initialize a new node
       struct node* temp = birth();
       int index = 0;
       // Format: "word/primary_meaning/secondary_meaning/"
       while(c != '/'){
           temp->word[index++] = c;
           c = fgetc(fp);
       }
       index = 0;
       c = fgetc(fp);
       while(c != '/'){
           temp->primary[index++] = c;
           c = fgetc(fp);
       }
       index = 0;
       c = getc(fp);
       while(c != '/'){
           temp->secondary[index++] = c;
           c = fgetc(fp);
       }
       for(int j = 0; j < 2; j++)
           c = fgetc(fp);
       // Insert the node in the linked list
       sortedInsertion(&root, temp);
   }
   // Remove duplicates, if any
   removeDuplicates();
}

// A function to get the middle of a linked list
struct node* middle(struct node* start, struct node* last){
   if (start == NULL)
       return NULL;
```

```
    // Run two pointers, one at double the speed of the other
    struct node* slow = start;
    struct node* fast = start -> next;
    while (fast != last){
        // When fast is null,
        // Slow is the middle node
        fast = fast -> next;
        if (fast != last){
            slow = slow -> next;
            fast = fast -> next;
        }
    }
    return slow;
}

// A binary search function on a linked list
void* binarySearch(void* params){
    // Get the value, start and end
    struct data* temp = (struct data *)params;
    char* value = temp->value;
    struct node* start = temp->start;
    struct node* last = temp->end;
    do{
        // Find middle
        struct node* mid = middle(start, last);
        // If middle is empty
        if (mid == NULL)
            return NULL;
        // If value is present at middle
        if (strcmp(mid->word, value) == 0){
            output = mid;
            return mid;
        }
        // If value is more than mid
        else if (strcmp(mid->word, value) < 0)
            start = mid -> next;
        // If the value is less than mid
        else
            last = mid;
    }while (last == NULL || last != start);
    // value not present
    return NULL;
```

```
}

// A function to search a word
struct node* search(char* data){
   // Get the middle of the linked list
   struct node* mid = middle(root, NULL);
   pthread_t ltid, rtid;
   // Data structure for passing in multithreaded binary search
   struct data lr[2];
   lr[0].value = data;
   lr[0].start = root;
   lr[0].end = mid;
   lr[1].value = data;
   lr[1].start = mid->next;
   lr[1].end = NULL;
   // Create 2 threads for left and right half of the linked list
   pthread_create(&ltid, NULL, binarySearch, &lr[0]);
   pthread_create(&rtid, NULL, binarySearch, &lr[1]);
   pthread_join(ltid, NULL);
   pthread_join(rtid, NULL);
   return output;
}
// A function for the producer (creating a dictionary)
void *producer(){
   // Lock the dictionary
   pthread_mutex_lock(&lock);
   // Create the dictionary
   createDictionary();
   // Unlock the dictionary
   pthread_mutex_unlock(&lock);
   pthread_exit(NULL);
}

// A function for the consumer (binary searching the dictionary)
void *consumer(){
   while(1){
       // Get user search string
       char* data = (char*) malloc(sizeof(char) * 40);
       printf("Enter the word to find in the dictionary: ");
       scanf("%s", data);
       getchar();
       // Lock the dictionary
```

```
        pthread_mutex_lock(&lock);
        // Search the string in the linked list
        struct node* result = search(data);
        // Unlock the dictionary
        pthread_mutex_unlock(&lock);
        if(result)
            printf("Word: %s\nPrimary Meaning: %s\nSecondary meaning: %s\n",
result->word, result->primary, result->secondary);
        else
            printf("Word not present in the dictionary\n");
        char c;
        printf("Do you want to search another word (y/n): ");
        scanf("%c", &c);
        if(c == 'n')
            break;
    }
    pthread_exit(NULL);
}
```

Output:



Dictionary Initialization Text File

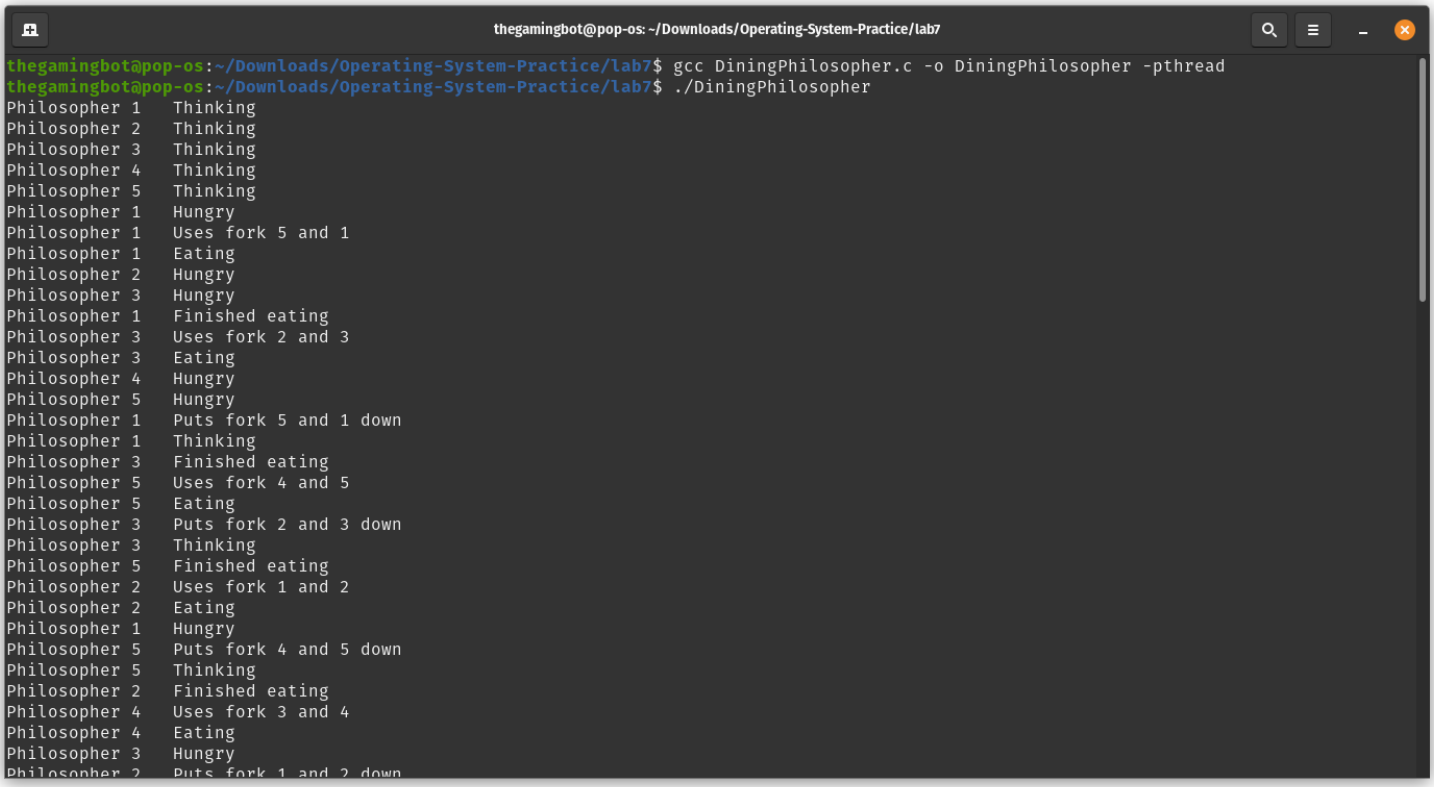
Question: Implement the Dining Philosophers problem

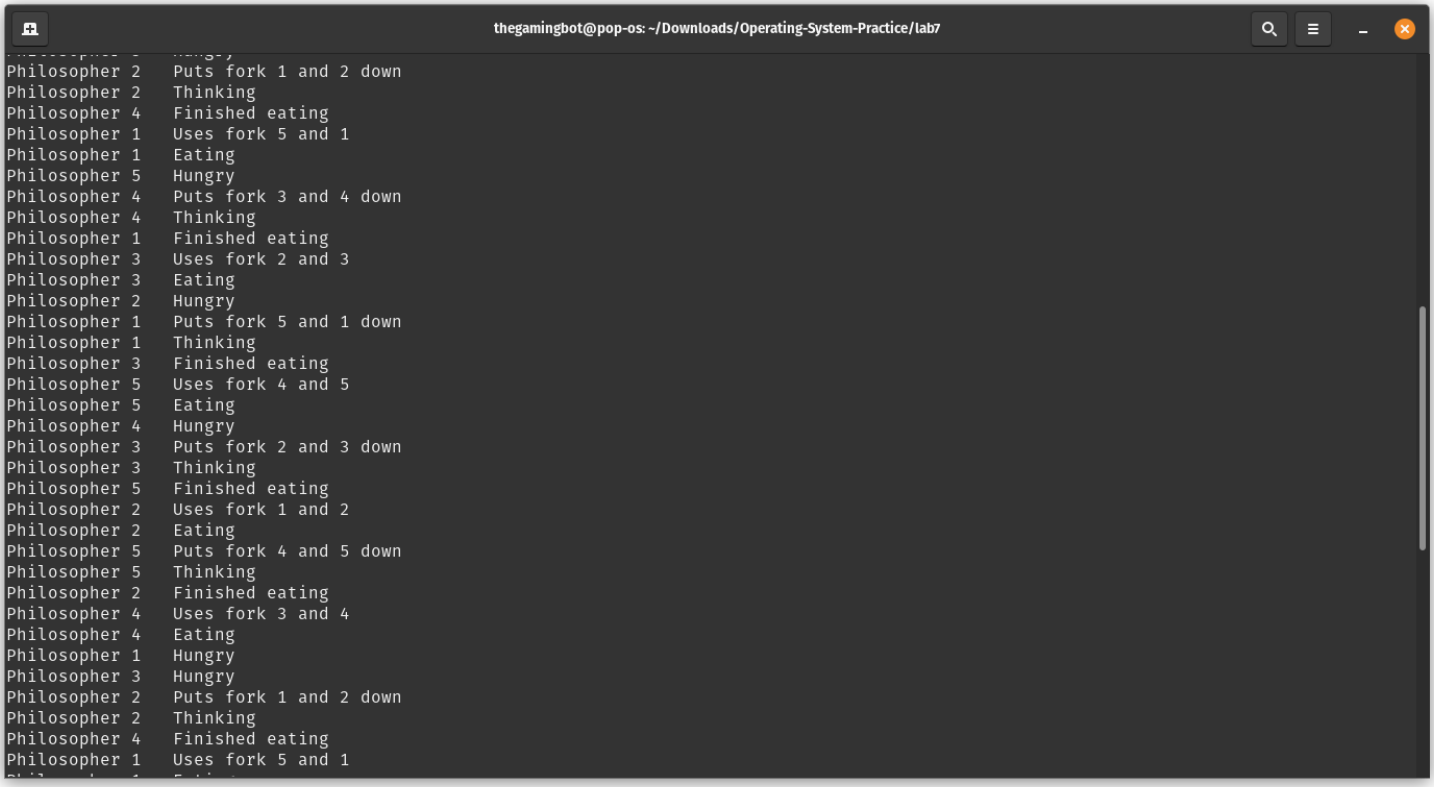
```c
// Include the required libraries
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>
#include<unistd.h>
// Define the macros
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (PhilNo + 4) % N
#define RIGHT (PhilNo + 1) % N
#define ITERATIONS 3
// Global constants
sem_t mutex;
sem_t S[N];
int state[N];
int PhilNoArr[N] = {0, 1, 2, 3, 4};
// Function declarations
void *philosopher(void *num);
void takeFork(int);
void putFork(int);
void test(int);
// Main driver function
int main(){
   pthread_t tid[N];
   // Initialize the semaphores
   sem_init(&mutex, 0, 1);
   for(int i = 0; i < N; i++)
       sem_init(&S[i], 0, 0);
   // Create new threads
   for(int i = 0; i < N; i++){
       pthread_create(&tid[i], NULL, philosopher, &PhilNoArr[i]);
       printf("Philosopher %d\tThinking\n", i+1);
   }
   // Wait for termination of the threads
   for(int i = 0; i < N; i++)
       pthread_join(tid[i], NULL);
   // Destroy the semaphores
   sem_destroy(&mutex);
   for(int i = 0; i < N; i++)
```

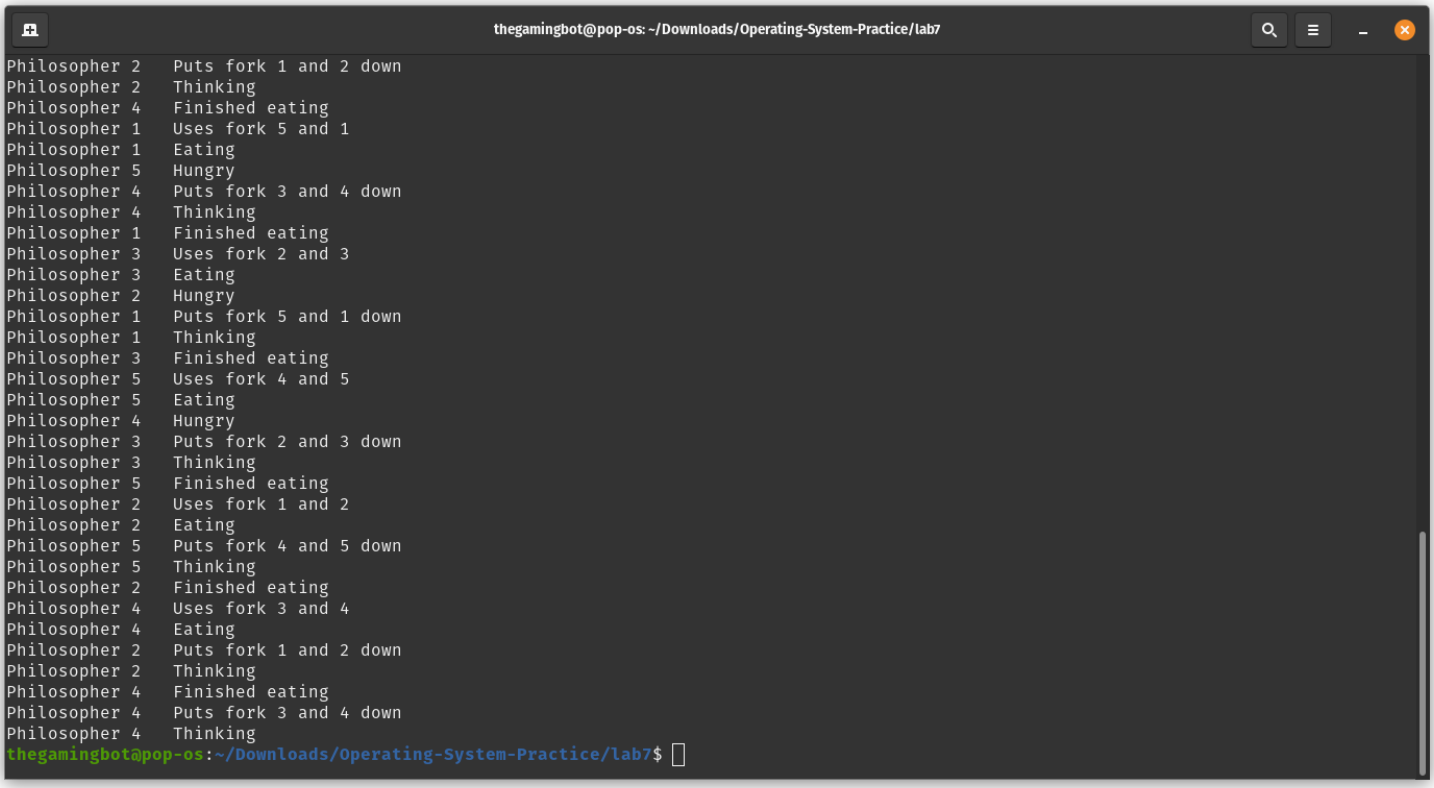
```
        sem_destroy(&S[i]);
    return 0;
}
// A function to run the actions of a philosopher
void *philosopher(void *num){
    for(int j = 0; j < ITERATIONS; j++){
        int *i = num;
        sleep(1);
        // Use the fork
        takeFork(*i);
        printf("Philosopher %d\tFinished eating\n", *i+1);
        // Put the fork back on the table
        putFork(*i);
    }
    pthread_exit(0);
}
// A function to take forks from the table
void takeFork(int PhilNo){
    // Lock the semaphore pointed to mutex to keep the update in the critical section
    sem_wait(&mutex);
    state[PhilNo] = HUNGRY;
    printf("Philosopher %d\tHungry\n", PhilNo + 1);
    // Test the philosopher
    test(PhilNo);
    // Unlocks the semaphore pointed to mutex
    sem_post(&mutex);
    // Lock the semaphore pointed to S[PhilNo]
    sem_wait(&S[PhilNo]);
    sleep(1);
}
// A function to put the fork back on the table
void putFork(int PhilNo){
    // Lock the semaphore pointed to mutex to keep the update in the critical section
    sem_wait(&mutex);
    // Change state to thinking
    state[PhilNo] = THINKING;
    printf("Philosopher %d\tPuts fork %d and %d down\n", PhilNo + 1, LEFT + 1, PhilNo +
1);
    printf("Philosopher %d\tThinking\n", PhilNo + 1);
    // Test his neighbors
    test(LEFT);
    test(RIGHT);
```

```
    // Unlocks the semaphore pointed to mutex
    sem_post(&mutex);
}
// A function to test the state of a given philosopher
void test(int PhilNo){
    // If the current philosopher is hungry and there are forks unused
    if(state[PhilNo] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
        // Change the state of PhilNo
        state[PhilNo] = EATING;
        sleep(2);
        printf("Philosopher %d\tUses fork %d and %d\n", PhilNo + 1, LEFT + 1, PhilNo +
1);
        printf("Philosopher %d\tEating\n", PhilNo + 1);
        // Unlocks the semaphore pointed to S[PhilNo]
        sem_post(&S[PhilNo]);
    }
}
```

Output:




thegamingbot@pop-os: ~/Downloads/Operating-System-Practice/lab7
Philosopher 2 Puts fork 1 and 2 down
Philosopher 2 Thinking
Philosopher 4 Finished eating
Philosopher 1 Uses fork 5 and 1
Philosopher 1 Eating
Philosopher 5 Hungry
Philosopher 4 Puts fork 3 and 4 down
Philosopher 4 Thinking
Philosopher 1 Finished eating
Philosopher 3 Uses fork 2 and 3
Philosopher 3 Eating
Philosopher 2 Hungry
Philosopher 1 Puts fork 5 and 1 down
Philosopher 1 Thinking
Philosopher 3 Finished eating
Philosopher 5 Uses fork 4 and 5
Philosopher 5 Eating
Philosopher 4 Hungry
Philosopher 3 Puts fork 2 and 3 down
Philosopher 3 Thinking
Philosopher 5 Finished eating
Philosopher 2 Uses fork 1 and 2
Philosopher 2 Eating
Philosopher 5 Puts fork 4 and 5 down
Philosopher 5 Thinking
Philosopher 2 Finished eating
Philosopher 4 Uses fork 3 and 4
Philosopher 4 Eating
Philosopher 1 Hungry
Philosopher 3 Hungry
Philosopher 2 Puts fork 1 and 2 down
Philosopher 2 Thinking
Philosopher 4 Finished eating
Philosopher 1 Uses fork 5 and 1

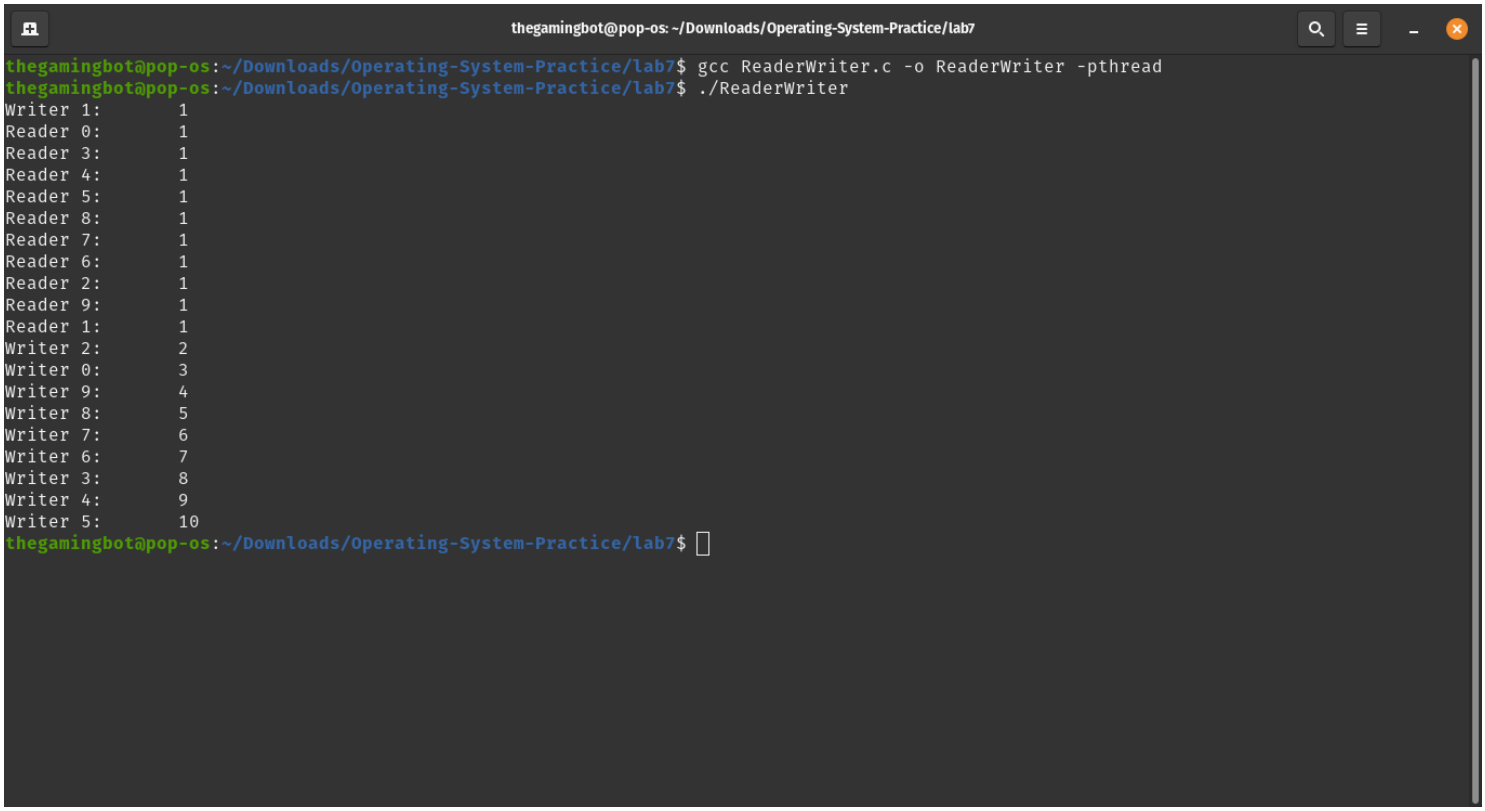

thegamingbot@pop-os: ~/Downloads/Operating-System-Practice/lab7
Philosopher 2 Puts fork 1 and 2 down
Philosopher 2 Thinking
Philosopher 4 Finished eating
Philosopher 1 Uses fork 5 and 1
Philosopher 1 Eating
Philosopher 5 Hungry
Philosopher 4 Puts fork 3 and 4 down
Philosopher 4 Thinking
Philosopher 1 Finished eating
Philosopher 3 Uses fork 2 and 3
Philosopher 3 Eating
Philosopher 2 Hungry
Philosopher 1 Puts fork 5 and 1 down
Philosopher 1 Thinking
Philosopher 3 Finished eating
Philosopher 5 Uses fork 4 and 5
Philosopher 5 Eating
Philosopher 4 Hungry
Philosopher 3 Puts fork 2 and 3 down
Philosopher 3 Thinking
Philosopher 5 Finished eating
Philosopher 2 Uses fork 1 and 2
Philosopher 2 Eating
Philosopher 5 Puts fork 4 and 5 down
Philosopher 5 Thinking
Philosopher 2 Finished eating
Philosopher 4 Uses fork 3 and 4
Philosopher 4 Eating
Philosopher 2 Puts fork 1 and 2 down
Philosopher 2 Thinking
Philosopher 4 Finished eating
Philosopher 4 Puts fork 3 and 4 down
Philosopher 4 Thinking
thegamingbot@pop-os:~/Downloads/Operating-System-Practice/lab7$

Question: Implement the Reader Writer problem

```c
// Include the required libraries
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>

// Define the macros
#define ITERATIONS 10

// Global constants
sem_t mutex, writeblock;
int data = 0, rcount = 0;
int RWIdx[ITERATIONS];

// Function declarations
void *reader(void *params);
void *writer(void *params);

// Main driver function
int main(){
   pthread_t rtid[ITERATIONS], wtid[ITERATIONS];
   // Initialize the semaphores
   sem_init(&mutex, 0, 1);
   sem_init(&writeblock, 0, 1);
   // Creates a new thread
   for(int i = 0; i < ITERATIONS; i++){
       RWIdx[i] = i;
       pthread_create(&wtid[i], NULL, writer, (void *)&RWIdx[i]);
       pthread_create(&rtid[i], NULL, reader, (void *)&RWIdx[i]);
   }
   // Wait for termination of the threads
   for(int i = 0; i < ITERATIONS; i++){
       pthread_join(rtid[i], NULL);
       pthread_join(wtid[i], NULL);
   }
   // Destroy the semaphores
   sem_destroy(&mutex);
   sem_destroy(&writeblock);
   return 0;
}
```

```c
void *reader(void *params){
   int *f = params;
   // Lock the semaphore pointed to mutex
   rcount = rcount + 1;
   // First reader
   if(rcount == 1)
       // Locks the semaphore pointed to writeblock
       sem_wait(&writeblock);
   // Unlock the semaphore pointed to mutex
   sem_post(&mutex);
   printf("Reader %d:\t%d\n", *f, data);
   // Locks the semaphore pointed to mutex
   sem_wait(&mutex);
   rcount = rcount - 1;
   // Final reader
   if(rcount == 0)
       // Unlock the semaphore pointed to by writeblock
       sem_post(&writeblock);
   sleep(1);
   // Unlock the semaphore pointed to mutex
   sem_post(&mutex);
   pthread_exit(0);
}

void *writer(void *params){
   int *f = params;
   // Lock the semaphore pointed to writeblock
   sem_wait(&writeblock);
   // Update the data
   data++;
   printf("Writer %d:\t%d\n", *f, data);
   sleep(1);
   // Unlock the semaphore pointed to writeblock
   sem_post(&writeblock);
   pthread_exit(0);
}
```

Output:


thegamingbot@pop-os: ~/Downloads/Operating-System-Practice/lab7
thegamingbot@pop-os:~/Downloads/Operating-System-Practice/lab7$ gcc ReaderWriter.c -o ReaderWriter -pthread
thegamingbot@pop-os:~/Downloads/Operating-System-Practice/lab7$ ./ReaderWriter
Writer 1: 1
Reader 0: 1
Reader 3: 1
Reader 4: 1
Reader 5: 1
Reader 8: 1
Reader 7: 1
Reader 6: 1
Reader 2: 1
Reader 9: 1
Reader 1: 1
Writer 2: 2
Writer 0: 3
Writer 9: 4
Writer 8: 5
Writer 7: 6
Writer 6: 7
Writer 3: 8
Writer 4: 9
Writer 5: 10
thegamingbot@pop-os:~/Downloads/Operating-System-Practice/lab7$

Question: Santa Claus Problem using semaphores

```c
// Include the required libraries
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <stdio.h>
#include <stdbool.h>
#include <semaphore.h>

// Define the macros
#define ELVES 10
#define REINDEER 9
#define ITERATIONS 3

// Global constants
int elves = 0, reindeer = 0;
sem_t santaSem, reindeerSem, elfSem, mutex;

// Function declarations
void *SantaClaus(void *params);
void *Reindeer(void *params);
void *Elf(void *params);

// Main driver function
int main(){
   // Initialize the semaphores
   sem_init(&santaSem, 0, 0);
   sem_init(&reindeerSem, 0, 0);
   sem_init(&elfSem, 0, 1);
   sem_init(&mutex, 0, 1);
   int reindeerArr[REINDEER], elfArr[ELVES];
   pthread_t santa_claus, reindeers[REINDEER], elves[ELVES];
   // Creates a new thread for Santa, Reindeers and Elves
   pthread_create(&santa_claus, NULL, SantaClaus, (void *)0);
   for (int i = 0; i < REINDEER; i++){
       reindeerArr[i] = i;
       pthread_create(&reindeers[i], NULL, Reindeer, (void *)&reindeerArr[i]);
   }
   for (int i = 0; i < ELVES; i++){
       elfArr[i] = i;
```

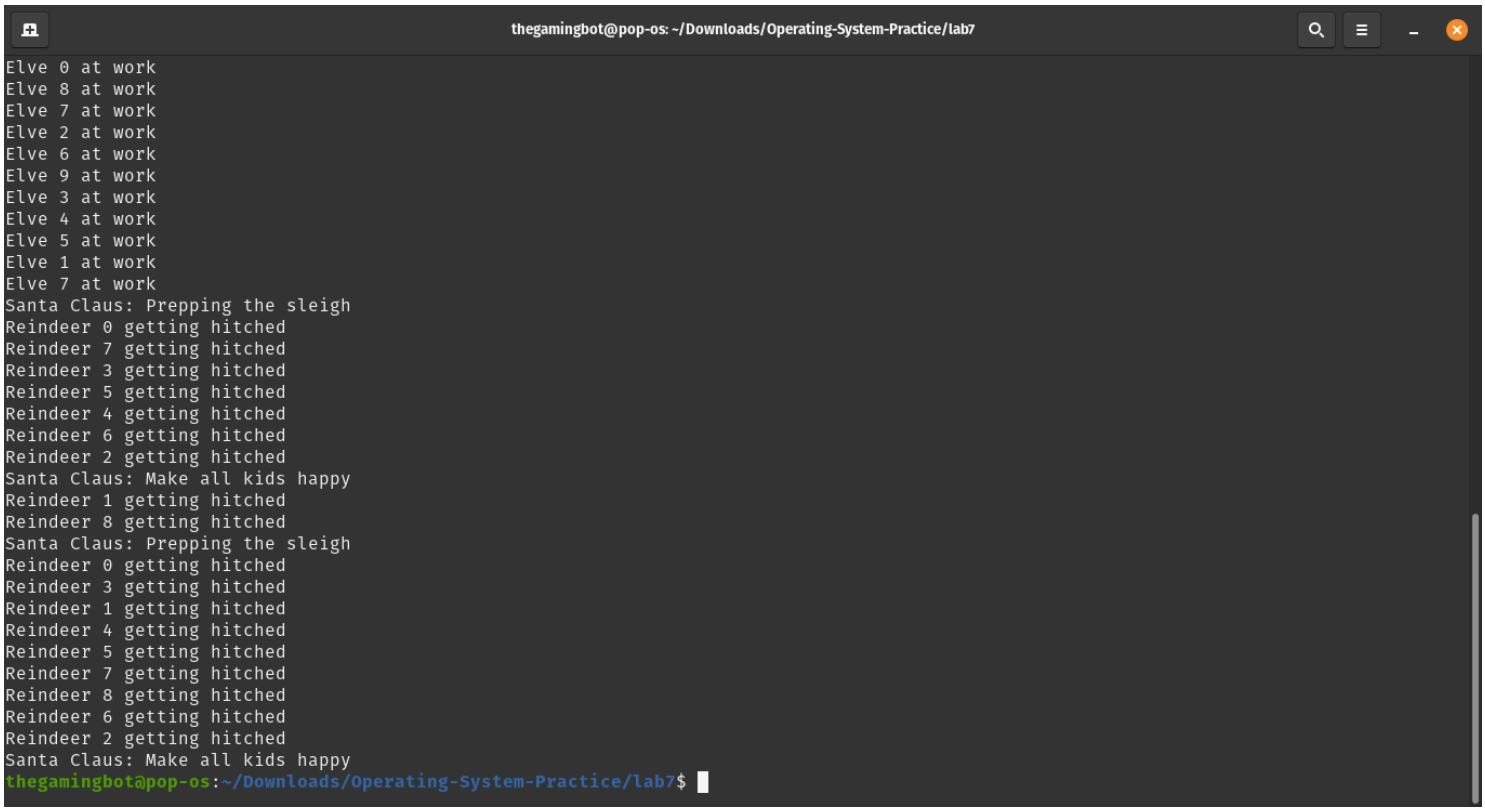
```
        pthread_create(&elves[i], NULL, Elf, (void *)&elfArr[i]);
    }
    // Wait for termination of the threads
    pthread_join(santa_claus, NULL);
    for (int i = 0; i < REINDEER; i++)
        pthread_join(reindeers[i], NULL);
    for (int i = 0; i < ELVES; i++)
        pthread_join(elves[i], NULL);
    // Destroy the semaphores
    sem_destroy(&santaSem);
    sem_destroy(&reindeerSem);
    sem_destroy(&elfSem);
    sem_destroy(&mutex);
}

// A function to manage tasks of Santa Claus
void *SantaClaus(void *params){
    printf("Santa Claus reporting to duty\n");
    // Loop the tasks
    for(int i = 0; i < ITERATIONS; i++){
        // Lock the semaphore pointing at santaSem and mutex
        sem_wait(&santaSem);
        sem_wait(&mutex);
        // Final reindeer
        if (reindeer == REINDEER){
            printf("Santa Claus: Prepping the sleigh\n");
            // Unlock all reindeerSem
            for (int j = 0; j < REINDEER; j++)
                sem_post(&reindeerSem);
            printf("Santa Claus: Make all kids happy\n");
            // Reset reindeer flag
            reindeer = 0;
        }
        // Help every third elf
        else if (elves == 3)
            printf("Santa Claus: Helping elves\n");
        // Unlock the semaphore pointing at mutex
        sem_post(&mutex);
    }
    // Unlock the semaphore pointing at santaSem
    sem_post(&santaSem);
}
```

```c
// A function to manage tasks of Reindeer
void *Reindeer(void *params){
   int id = *((int *)params);
   printf("Reindeer %d reporting to duty\n", id);
   // Loop the tasks
   for(int i = 0; i < ITERATIONS; i++){
       // Lock the semaphore pointing at mutex
       sem_wait(&mutex);
       reindeer++;
       // Final reindeer
       if (reindeer == REINDEER)
           // Unlock the semaphore pointing at santaSem
           sem_post(&santaSem);
       // Unlock the semaphore pointing at mutex
       sem_post(&mutex);
       // Unlock the semaphore pointing at reindeerSem
       sem_wait(&reindeerSem);
       printf("Reindeer %d getting hitched\n", id);
       sleep(20);
   }
}

// A function to manage tasks of Elf
void *Elf(void *params){
   int id = *((int *)params);
   printf("Elf %d reporting to duty\n", id);
   // Loop the tasks
   for(int i = 0; i < ITERATIONS; i++){
       bool need_help = random() % 100 < 10;
       // If help needed
       if (need_help){
           // Lock the semaphore pointing at elfSem and mutex
           sem_wait(&elfSem);
           sem_wait(&mutex);
           elves++;
           // Every third elf
           if (elves == 3)
               // Unlock the semaphore pointing at santaSem
               sem_post(&santaSem);
           // Else
           else
```

```
                // Unlock the semaphore pointing at elfSem
                sem_post(&elfSem);
            // Unlock the semaphore pointing at mutex
            sem_post(&mutex);
            printf("Elf %d will get help from Santa Claus\n", id);
            sleep(10);
            // Lock the semaphore pointing at mutex
            sem_wait(&mutex);
            elves--;
            // If all work is done
            if (elves == 0)
                // Unlock the semaphore pointing at elfSem
                sem_post(&elfSem);
            // Unlock the semaphore pointing at mutex
            sem_post(&mutex);
        }
        printf("Elf %d at work\n", id);
        sleep(2 + random() % 5);
    }
}
```

Output:


thegamingbot@pop-os: ~/Downloads/Operating-System-Practice/lab7
Reindeer 5 getting hitched
Reindeer 0 getting hitched
Reindeer 7 getting hitched
Reindeer 6 getting hitched
Santa Claus: Make all kids happy
Reindeer 4 getting hitched
Reindeer 8 getting hitched
Reindeer 1 getting hitched
Reindeer 2 getting hitched
Elve 2 at work
Elve 3 at work
Elve 0 at work
Elve 5 at work
Elve 8 at work
Elve 9 at work
Elve 1 at work
Elve 4 at work
Elve 6 at work
Elve 0 at work
Elve 8 at work
Elve 7 at work
Elve 2 at work
Elve 6 at work
Elve 9 at work
Elve 3 at work
Elve 4 at work
Elve 5 at work
Elve 1 at work
Elve 7 at work
Santa Claus: Prepping the sleigh
Reindeer 0 getting hitched
Reindeer 7 getting hitched
Reindeer 3 getting hitched
Reindeer 5 getting hitched


thegamingbot@pop-os: ~/Downloads/Operating-System-Practice/lab7
Elve 0 at work
Elve 8 at work
Elve 7 at work
Elve 2 at work
Elve 6 at work
Elve 9 at work
Elve 3 at work
Elve 4 at work
Elve 5 at work
Elve 1 at work
Elve 7 at work
Santa Claus: Prepping the sleigh
Reindeer 0 getting hitched
Reindeer 7 getting hitched
Reindeer 3 getting hitched
Reindeer 5 getting hitched
Reindeer 4 getting hitched
Reindeer 6 getting hitched
Reindeer 2 getting hitched
Santa Claus: Make all kids happy
Reindeer 1 getting hitched
Reindeer 8 getting hitched
Santa Claus: Prepping the sleigh
Reindeer 0 getting hitched
Reindeer 3 getting hitched
Reindeer 1 getting hitched
Reindeer 4 getting hitched
Reindeer 5 getting hitched
Reindeer 7 getting hitched
Reindeer 8 getting hitched
Reindeer 6 getting hitched
Reindeer 2 getting hitched
Santa Claus: Make all kids happy
thegamingbot@pop-os:~/Downloads/Operating-System-Practice/lab7$

Question: H_2O problem using semaphores

```c
// Include the required libraries
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

// Define the macros
#define HYDROGEN 10
#define OXYGEN 5

// Global constants
int oxygenNo = 0, hydrogenNo = 0, count = 0, flag = 0;
sem_t mutex, hydrogenSem, oxygenSem, barrier;

// Function declarations
void bond();
void *Hydrogen(void* params);
void *Oxygen(void* params);

// Main driver function
int main(){
   pthread_t htid[HYDROGEN], otid[OXYGEN];
   // Initialize the semaphores
   sem_init(&mutex, 0, 1);
   sem_init(&hydrogenSem, 0, 0);
   sem_init(&oxygenSem, 0, 0);
   sem_init(&barrier, 0, 3);
   int H[HYDROGEN], O[OXYGEN];
   // Creates new threads
   for (int i = 0; i < HYDROGEN; i++){
       H[i] = i;
       pthread_create(&htid[i], NULL, Hydrogen, (void *)&H[i]);
   }
   for (int i = 0; i < OXYGEN; i++){
       O[i] = i;
       pthread_create(&otid[i], NULL, Oxygen, (void *)&O[i]);
   }
   // Wait for termination of the threads
   for (int i = 0; i < HYDROGEN; i++)
```

```
        pthread_join(htid[i], NULL);
    for (int i = 0; i < OXYGEN; i++)
        pthread_join(otid[i], NULL);
    // Destroy the semaphores
    sem_destroy(&mutex);
    sem_destroy(&hydrogenSem);
    sem_destroy(&oxygenSem);
    sem_destroy(&barrier);
    return 0;
}

// A function to bond the atoms
void bond(){
    printf("H2O molecule bonded\n");
    count += 1;
    // If the molecule count is equal to oxygen count
    if (count == OXYGEN){
        // Exit the program
        printf("\nMaximum Generated Water Molecules: %d\n\n", count);
        exit(1);
    }
    sleep(1);
}

// A function to generate H atoms
void *Hydrogen(void* params){
    printf("H atom generated: %d\n", *((int *)params));
    while(1){
        // Lock the semaphore pointing at mutex
        sem_wait(&mutex);
        hydrogenNo += 1;
        // If there are more than 2 hydrogen and one oxygen
        if (hydrogenNo >= 2 && oxygenNo >= 1){
        // If there are more than 2 hydrogen and one oxygen
            sem_post(&hydrogenSem);
            // Decrement H atoms
            hydrogenNo -= 2;
            // Unlock the semaphore pointing at oxygenSem
            sem_post(&oxygenSem);
            // Decrement O atoms
            oxygenNo -= 1;
        }
```

```
        // Else
        else
            // Unlock the semaphore pointing at the mutex
            sem_post(&mutex);
        // Lock the semaphore pointing at the hydrogenSem
        sem_wait(&hydrogenSem);
        // Bond the atoms
        bond();
        // Lock the semaphore pointing at the barrier
        sem_wait(&barrier);
    }
    // Exit the thread
    pthread_exit(NULL);
}

// A function to generate O atoms
void *Oxygen(void* params){
    printf("Oxygen Molecule Generated: %d\n", *((int *)params));
    while (1){
        // Lock the semaphore pointing at mutex
        sem_wait(&mutex);
        oxygenNo += 1;
        // If there are more than 2 hydrogen and one oxygen
        if (hydrogenNo >= 2){
        // If there are more than 2 hydrogen and one oxygen
            sem_post(&hydrogenSem);
            // Decrement H atoms
            hydrogenNo -= 2;
            // Unlock the semaphore pointing at oxygenSem
            sem_post(&oxygenSem);
            // Decrement O atoms
            oxygenNo -= 1;
        }
        // Else
        else
            // Unlock the semaphore pointing at the mutex
            sem_post(&mutex);
        // Lock the semaphore pointing at the oxygenSem
        sem_wait(&oxygenSem);
        // Bond the atoms
        bond();
        // Lock the semaphore pointing at the barrier
```

```
        sem_wait(&barrier);
        // Unlock the semaphore pointing at the mutex
        sem_post(&mutex);
    }
    // Exit the thread
    pthread_exit(NULL);
}
```

Output: