**Slot: L3+L4 Name: Swaranjana Nayak**

**Date: 02/11/2020 Reg. No.: 19BCE0977**

**LAB FAT**

**Problem 1**

**CIGARETTE SMOKER’S PROBLEM**

**Aim:**

To implement Cigarette Smoker’s problem using Semaphore

**Algorithm:**

There are 4 processes - agent, smoker1, smoker2, smoker3.

Note: This program uses UNIX STANDARD library, POSIX THREAD library and SEMAPHORE library. The program has been executed on VMware in the terminal as shown in screenshots.

1. Let there be following semaphores
   1. Agent\_ready: an agent semaphore that represents items on the table.
   2. Smoker\_semaphores: each smoker semaphore represents when a smoker has the items they need
   3. Pusher\_semaphores: while pushing
   4. Pusher\_lock: exclusive access to the items on table
2. Let there be following boolean/flag/information variables
   1. Smoker\_types: strings describing what each smoker type needs.
   2. Items\_on\_table: this list represents item types that are on the table. Respectively for smoker\_types. Each item is the one the smoker has. Paper, then tobacco,then matches.
3. The smoker function handles waiting for the items that they need and and then smokes. Since its given the process goes on forever, just for example, here its done 3 times. It waits on semaphore Smoker\_semaphore[No] for his ingredients, notifies the agent that the ingredients have been taken and the table is free once the needed ingredients are available, then executes for loop to simulate smoking for a while, and then goes back for the next round.
4. Pusher function handles releasing of proper smoker’s semaphores when the right items are on the table.
5. The agent function handles putting items on the table. It waits on the agent\_ready semaphore
6. The main handles agent’s arbitration of putting items on the table.

**Program:**

#include <pthread.h>

#include <semaphore.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

sem\_t agent\_ready;

sem\_t smoker\_semaphors[3];

char\* smoker\_types[3] = { "matches & tobacco", "matches & paper", "tobacco & paper" };

bool items\_on\_table[3] = { false, false, false };

sem\_t pusher\_semaphores[3];

sem\_t pusher\_lock;

void\* smoker(void\* arg)

{

    int smoker\_id = \*(int\*) arg;

    int type\_id   = smoker\_id % 3;

    // Smoke 3 times

    for (int i = 0; i < 3; ++i)

    {

        printf("\033[0;37mSmoker %d \033[0;31m>>\033[0m Waiting for %s\n", smoker\_id, smoker\_types[type\_id]);

        // Wait for the proper combination of items to be on the table

        sem\_wait(&smoker\_semaphors[type\_id]);

        // Make the cigarette before releasing the agent

        printf("\033[0;37mSmoker %d \033[0;32m<<\033[0m Now making the a cigarette\n", smoker\_id);

        usleep(rand() % 50000);

        sem\_post(&agent\_ready);

        // Smoking now

        printf("\033[0;37mSmoker %d \033[0;37m--\033[0m Now smoking\n", smoker\_id);

        usleep(rand() % 50000);

    }

    return NULL;

}

void\* pusher(void\* arg)

{

    int pusher\_id = \*(int\*) arg;

    for (int i = 0; i < 12; ++i)

    {

        // Wait for this pusher to be needed

        sem\_wait(&pusher\_semaphores[pusher\_id]);

        sem\_wait(&pusher\_lock);

        // Check if the other item we need is on the table

        if (items\_on\_table[(pusher\_id + 1) % 3])

        {

            items\_on\_table[(pusher\_id + 1) % 3] = false;

            sem\_post(&smoker\_semaphors[(pusher\_id + 2) % 3]);

        }

        else if (items\_on\_table[(pusher\_id + 2) % 3])

        {

            items\_on\_table[(pusher\_id + 2) % 3] = false;

            sem\_post(&smoker\_semaphors[(pusher\_id + 1) % 3]);

        }

        else

        {

            // The other item's aren't on the table yet

            items\_on\_table[pusher\_id] = true;

        }

        sem\_post(&pusher\_lock);

    }

    return NULL;

}

void\* agent(void\* arg)

{

    int agent\_id = \*(int\*) arg;

    for (int i = 0; i < 6; ++i)

    {

        usleep(rand() % 200000);

        // Wait for a lock on the agent

        sem\_wait(&agent\_ready);

        // Items this agent gives out

        sem\_post(&pusher\_semaphores[agent\_id]);

        sem\_post(&pusher\_semaphores[(agent\_id + 1) % 3]);

        // Print type of items we put on the table

        printf("\033[0;35m==> \033[0;33mAgent %d giving out %s\033[0;0m\n",

            agent\_id, smoker\_types[(agent\_id + 2) % 3]);

    }

    return NULL;

}

int main(int argc, char\* arvg[])

{

    srand(time(NULL));

    // A values of 1 = nothing on the table

    sem\_init(&agent\_ready, 0, 1);

    // Initalize the pusher lock semaphore

    sem\_init(&pusher\_lock, 0, 1);

    // Initialize the semaphores for the smokers and pusher

    for (int i = 0; i < 3; ++i)

    {

        sem\_init(&smoker\_semaphors[i], 0, 0);

        sem\_init(&pusher\_semaphores[i], 0, 0);

    }

    int smoker\_ids[6];

    pthread\_t smoker\_threads[6];

    // Create the 6 smoker threads with IDs

    for (int i = 0; i < 6; ++i)

    {

        smoker\_ids[i] = i;

        if (pthread\_create(&smoker\_threads[i], NULL, smoker, &smoker\_ids[i]) == EAGAIN)

        {

            perror("Insufficient resources to create thread");

            return 0;

        }

    }

    int pusher\_ids[6];

    pthread\_t pusher\_threads[6];

    for (int i = 0; i < 3; ++i)

    {

        pusher\_ids[i] = i;

        if (pthread\_create(&pusher\_threads[i], NULL, pusher, &pusher\_ids[i]) == EAGAIN)

        {

            perror("Insufficient resources to create thread");

            return 0;

        }

    }

    int agent\_ids[6];

    pthread\_t agent\_threads[6];

    for (int i = 0; i < 3; ++i)

    {

        agent\_ids[i] =i;

        if (pthread\_create(&agent\_threads[i], NULL, agent, &agent\_ids[i]) == EAGAIN)

        {

            perror("Insufficient resources to create thread");

            return 0;

        }

    }

    // Make sure all the smokers are done smoking

    for (int i = 0; i < 6; ++i)

    {

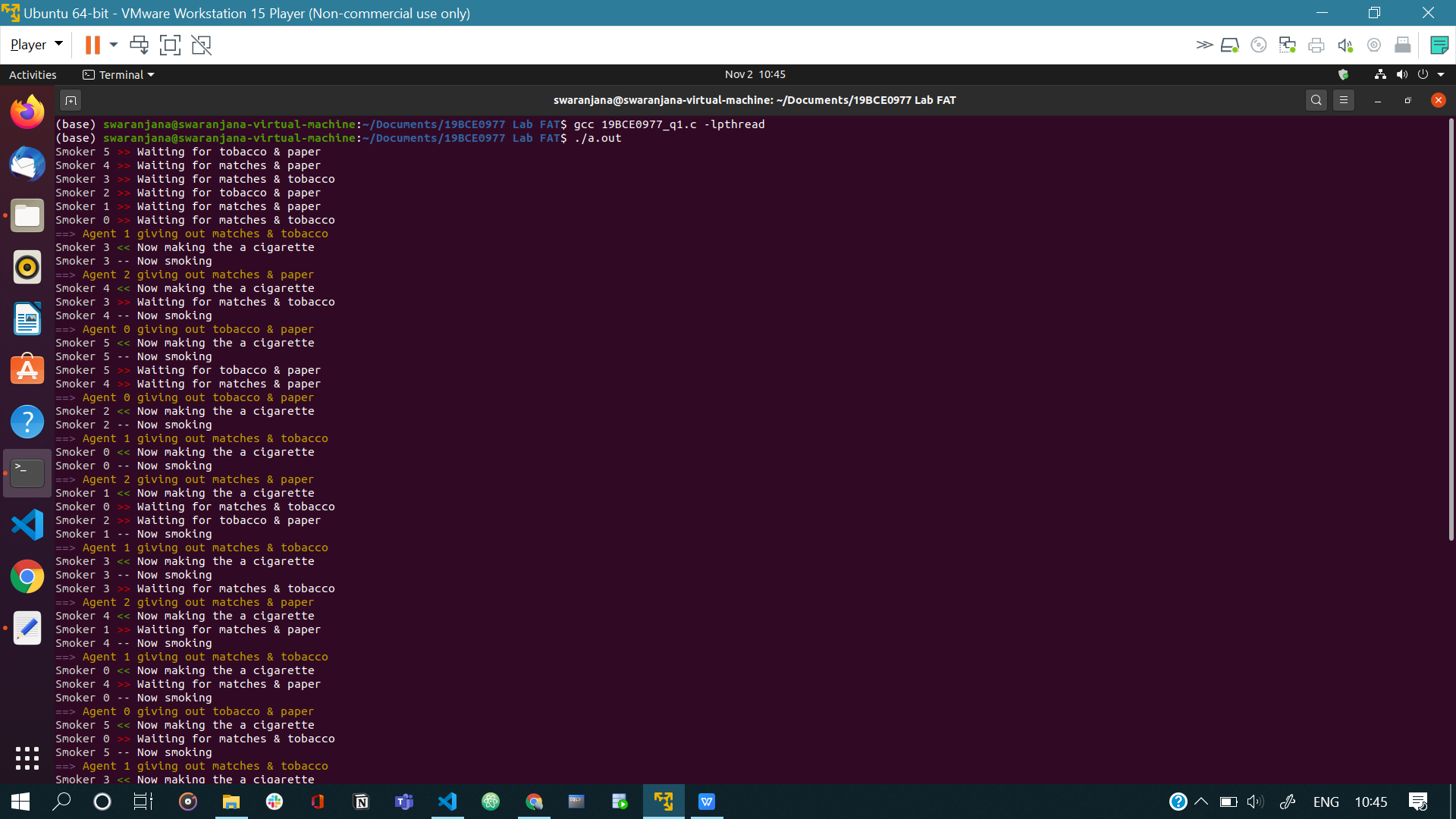
        pthread\_join(smoker\_threads[i], NULL);

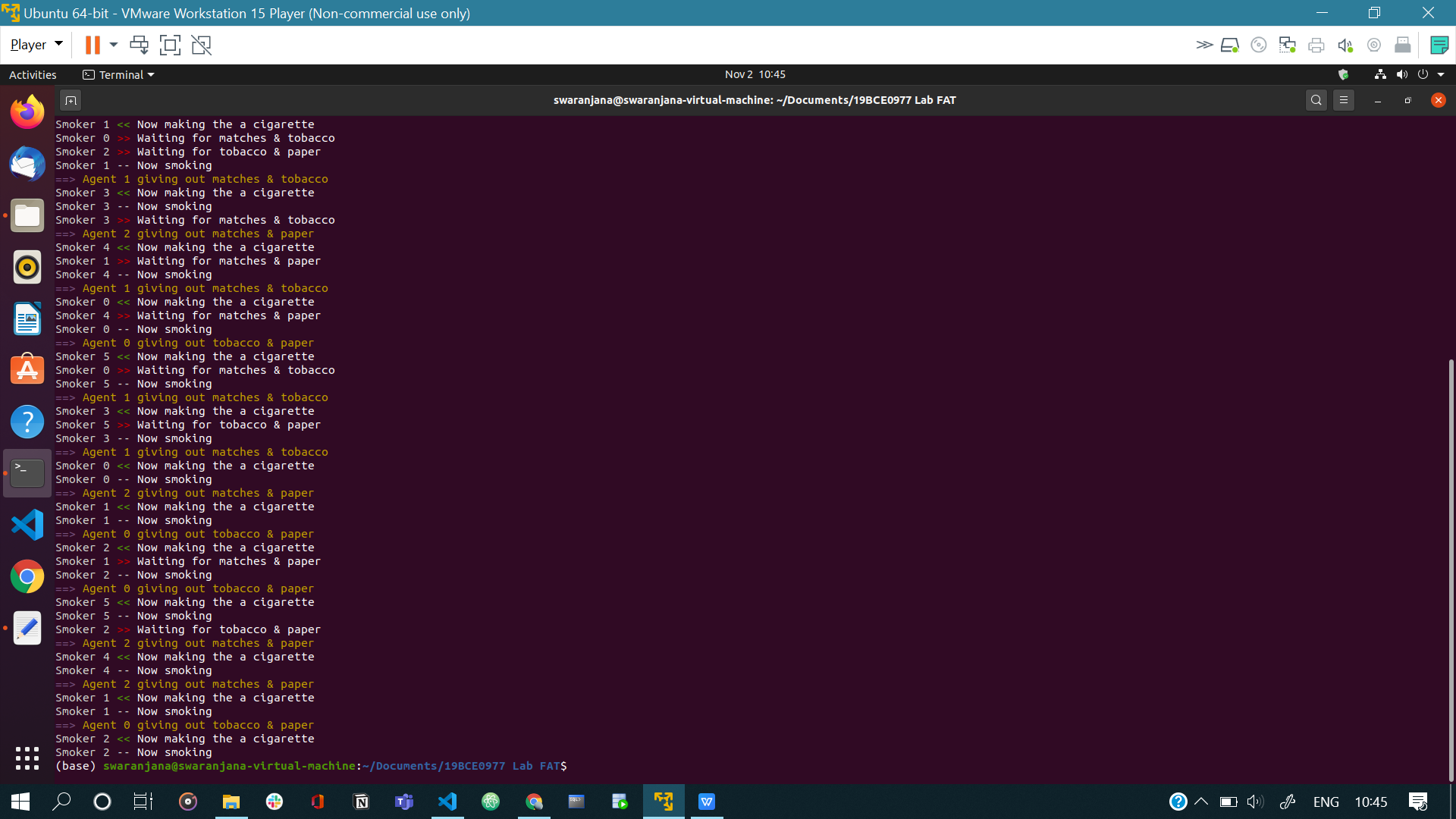
    }

    return 0;

}

**Output:**

****



**Problem 2**

**CSCAN DISK SCHEDULING ALGORITHM**

**Aim:**

To write and execute a C program to implement CSCAN disk scheduling, and test it on given inputs.

**Algorithm:**

1. Let ‘q\_size’ be number of disk positions to be read.
2. Let ‘head’ be position of the disk head.
3. Let queue1 array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival, which come after the initial head position.
4. Let queue2 array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival, which come before the initial head position.
5. Because size of disk movement wasn’t given, the head services only in the left direction from size of the disk to 0.
6. While moving in the right direction do not service any of the tracks.
7. When we reach at the end(right end), reverse the direction.
8. While moving in left direction it services all tracks one by one.
9. While moving in left direction calculate the absolute distance of the track from the head.
10. Increment the total head movement count (‘seek’) with this distance.
11. Currently serviced track position now becomes the new head position.
12. Go to step 6 until we reach at left end of the disk.
13. If we reach at the left end of the disk reverse the direction and go to step 3 until all tracks in request array have not been serviced.

**Program:**

#include <stdlib.h>

#include <stdio.h>

#define UP 4999

#define DOWN 0

int main()

{

  int queue[20], q\_size, head, i, j, seek = 0, diff, max, temp, queue1[20], queue2[20], temp1 = 0, temp2 = 0;

  printf("Input no of disk locations: ");

  scanf("%d", &q\_size);

  printf("Enter initial head position: ");

  scanf("%d", &head);

  printf("Enter disk positions to be read: ");

  for (i = 0; i < q\_size; i++)

  {

    scanf("%d", &temp);

    if (temp >= head)

    {

      queue1[temp1] = temp;

      temp1++;

    }

    else

    {

      queue2[temp2] = temp;

      temp2++;

    }

  }

  //sort both arrays

  for (i = 0; i < temp1 - 1; i++)

  {

    for (j = i + 1; j < temp1; j++)

    {

      if (queue1[i] > queue1[j])

      {

        temp = queue1[i];

        queue1[i] = queue1[j];

        queue1[j] = temp;

      }

    }

  }

  for (i = 0; i < temp2 - 1; i++)

  {

    for (j = i + 1; j < temp2; j++)

    {

      if (queue2[i] > queue2[j])

      {

        temp = queue2[i];

        queue2[i] = queue2[j];

        queue2[j] = temp;

      }

    }

  }

  //calculate closest edge

  if (abs(head - DOWN) >= abs(head - UP))

  {

    for (i = 1, j = 0; j < temp1; i++, j++)

    {

      queue[i] = queue1[j];

    }

    queue[i] = UP;

    queue[i + 1] = 0;

    for (i = temp1 + 3, j = 0; j < temp2; i++, j++)

    {

      queue[i] = queue2[j];

    }

  }

  else

  {

    for (i = 1, j = temp2 - 1; j >= 0; i++, j--)

    {

      queue[i] = queue2[j];

    }

    queue[i] = DOWN;

    queue[i + 1] = UP;

    for (i = temp2 + 3, j = temp1 - 1; j >= 0; i++, j--)

    {

      queue[i] = queue1[j];

    }

  }

  queue[0] = head;

  for (j = 0; j <= q\_size + 1; j++)

  {

    diff = abs(queue[j + 1] - queue[j]);

    seek += diff;

    printf("Disk head moves from %d to %d with head movements %d\n", queue[j], queue[j + 1], diff);

  }

  printf("\nTotal head movements are %d\n", seek);

  return 0;

}

**Output:**

