**OPERATING SYSTEMS**

**ASSIGNMENT 7**

**NAME : T Karthikeyan**

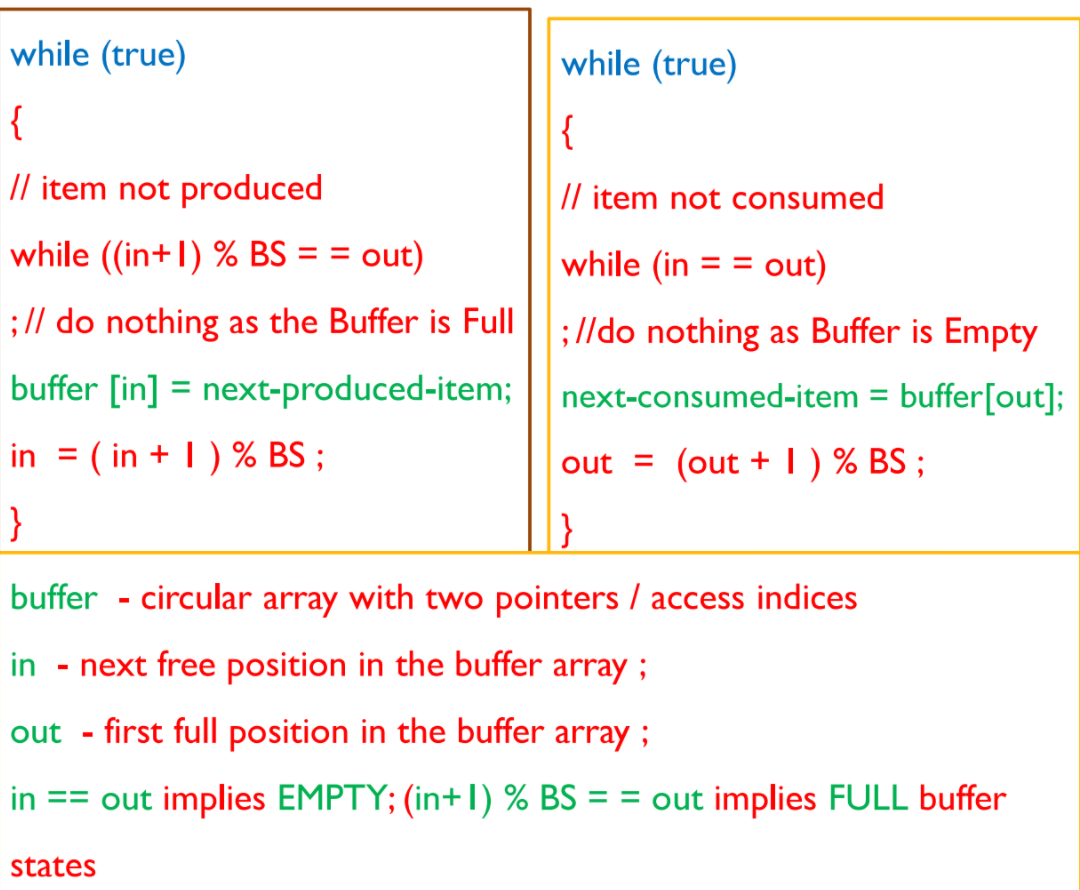
**ROLL : CED18I064**

**All my codes can be found here** [**CLICK HERE**](https://drive.google.com/drive/folders/15f8z17wcV-QzQxj6ZXuyNx5mlu_vCB52?usp=sharing)

**1. Simulate the Producer Consumer code discussed in the class.**

**Logic**

*Producer Consumer*

****

**Code**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<pthread.h>**

**#include<time.h>**

**#define BS 5**

**int in = 0;**

**int out = 0;**

**int buffer[BS];**

**void \*producer(void \*param)**

**{**

**int next\_produced\_item;**

**while(1)**

**{**

**next\_produced\_item = rand()%1000;**

**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 in : %d out : %d\n", buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out); //why not for loop? To make the operation atomic**

**printf("\n");**

**sleep(2);**

**}**

**pthread\_exit(0);**

**}**

**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 in : %d out : %d\n", buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out); //why not for loop? To make the operation atomic**

**printf("\n");**

**sleep(3);**

**}**

**pthread\_exit(0);**

**}**

**int main(int argc, char \*argv[])**

**{**

**int in, out;**

**in = 0;**

**out = 0;**

**srand(time(0));**

**pthread\_t pid, cid;**

**pthread\_attr\_t attr1, attr2;**

**pthread\_attr\_init(&attr1);**

**pthread\_attr\_init(&attr2);**

**pthread\_create(&pid, &attr1, producer, NULL);**

**pthread\_create(&cid, &attr2, consumer, NULL);**

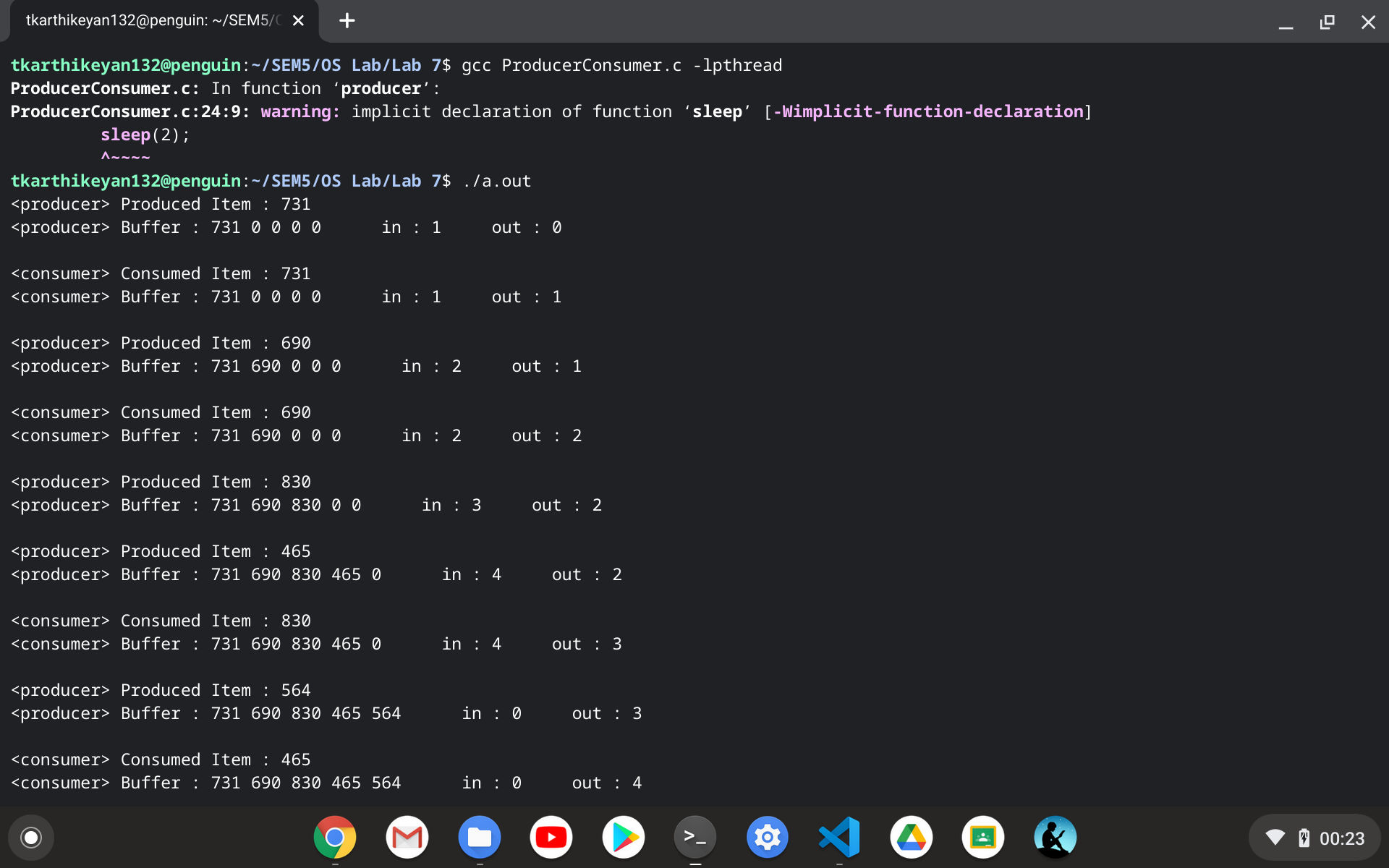
**pthread\_join(pid, NULL);**

**pthread\_join(cid, NULL);**

**return 0;**

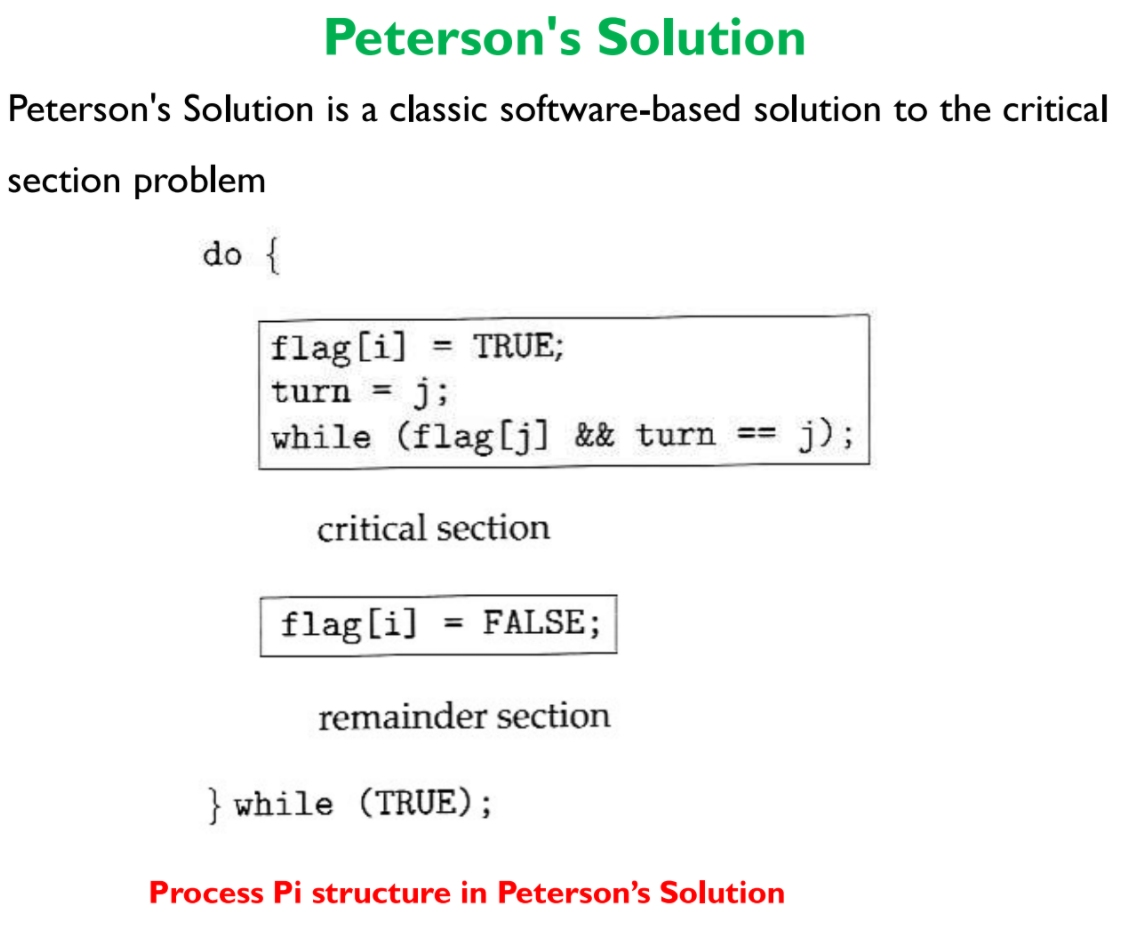
**}**

**Output**

****

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

**Logic**

****

**Code**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<stdbool.h>**

**#include<pthread.h>**

**#include<time.h>**

**#define BS 5**

**int in = 0;**

**int out = 0;**

**int buffer[BS];**

**//0:Producer**

**//1:Consumer**

**//flag[i] = true implies P'i wants to enter critical section**

**bool flag[2] = {false, false};**

**//turn = i implies P'i turn**

**int turn = -1;**

**void \*producer(void \*param)**

**{**

**do**

**{**

**int next\_produced\_item;**

**flag[0] = true;**

**turn = 1;**

**//being humble towards consumer**

**while(flag[1] && turn == 1);**

**//critical section begins**

**next\_produced\_item = rand()%1000;**

**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 in : %d out : %d\n", buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out); //why not for loop? To make the operation atomic**

**printf("\n");**

**sleep(1);**

**//critical section ends**

**flag[0] = false;**

**} while (true);**

**pthread\_exit(0);**

**}**

**void \*consumer(void \*param)**

**{**

**do**

**{**

**int next\_consumed\_item;**

**flag[1] = true;**

**turn = 0;**

**//being humble towards producer**

**while(flag[0] && turn == 0);**

**//critical section begins**

**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 in : %d out : %d\n", buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], in, out); //why not for loop? To make the operation atomic**

**printf("\n");**

**sleep(3);**

**//critical section ends**

**flag[1] = false;**

**} while (true);**

**pthread\_exit(0);**

**}**

**int main(int argc, char \*argv[])**

**{**

**srand(time(0));**

**pthread\_t pid, cid;**

**pthread\_attr\_t attr1, attr2;**

**pthread\_attr\_init(&attr1);**

**pthread\_attr\_init(&attr2);**

**pthread\_create(&pid, &attr1, producer, NULL);**

**pthread\_create(&cid, &attr2, consumer, NULL);**

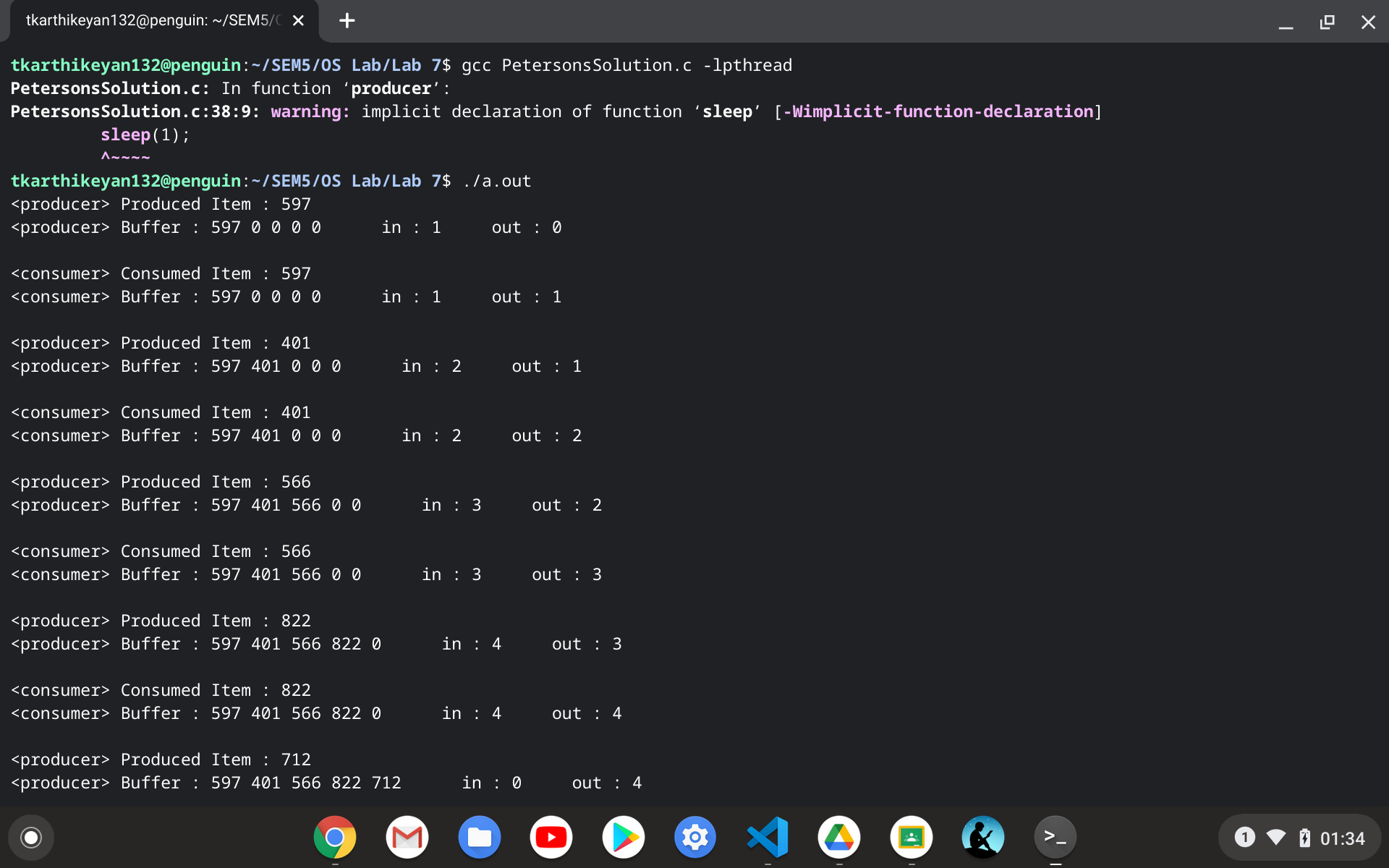
**pthread\_join(pid, NULL);**

**pthread\_join(cid, NULL);**

**return 0;**

**}**

**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.**

**Logic**

*Dictionary is created using peterson's solution for synchronization*

*Linear Search is applied for searching*

**Code**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<stdbool.h>**

**#include<pthread.h>**

**#include<time.h>**

**#include<string.h>**

**#define DICT\_LIM 20**

**int NUM\_WORDS = 0; //number of words in dictionary**

**struct word**

**{**

**char word[25];**

**char meaning1[100];**

**char meaning2[100];**

**};**

**struct word W[DICT\_LIM];**

**//0:Producer**

**//1:Consumer**

**//flag[i] = true implies P'i wants to enter critical section**

**bool flag[2] = {false, false};**

**//turn = i implies P'i turn**

**int turn = -1;**

**void \*producer(void \*param)**

**{**

**do**

**{**

**flag[0] = true;**

**turn = 1;**

**//being humble towards consumer**

**while(flag[1] && turn == 1);**

**//critical section begins**

**if(NUM\_WORDS == DICT\_LIM)**

**printf("Dictionary is full\n");**

**else**

**{**

**char w[25];**

**char mean1[100];**

**char mean2[100];**

**printf("<producer>\n");**

**printf("Enter the word : ");**

**gets(w);**

**printf("Enter the meaning 1 : ");**

**gets(mean1);**

**printf("Enter the meaning 2 : ");**

**gets(mean2);**

**strcpy(W[NUM\_WORDS].word, w);**

**strcpy(W[NUM\_WORDS].meaning1, mean1);**

**strcpy(W[NUM\_WORDS].meaning2, mean2);**

**NUM\_WORDS++;**

**}**

**//critical section ends**

**flag[0] = false;**

**} while (true);**

**pthread\_exit(0);**

**}**

**void \*consumer(void \*param)**

**{**

**do**

**{**

**flag[1] = true;**

**turn = 0;**

**//being humble towards producer**

**while(flag[0] && turn == 0);**

**//critical section begins**

**if(NUM\_WORDS == 0)**

**printf("Dictionary is empty\n");**

**else**

**{**

**char w[25];**

**int flag = -1;**

**printf("<consumer>\n");**

**printf("Enter the word to be searched : ");**

**gets(w);**

**for(int i = 0; i < NUM\_WORDS; i++)**

**{**

**if(strcmp(w,W[i].word) == 0)**

**{**

**printf("primary meaning : %s\n", W[i].meaning1);**

**printf("secondary meaning : %s\n", W[i].meaning2);**

**flag = 0;**

**break;**

**}**

**}**

**if(flag)**

**printf("Requested word not found\n");**

**}**

**//critical section ends**

**flag[1] = false;**

**} while (true);**

**pthread\_exit(0);**

**}**

**int main(int argc, char \*argv[])**

**{**

**pthread\_t pid, cid;**

**pthread\_attr\_t attr1, attr2;**

**pthread\_attr\_init(&attr1);**

**pthread\_attr\_init(&attr2);**

**pthread\_create(&pid, &attr1, producer, NULL);**

**pthread\_create(&cid, &attr2, consumer, NULL);**

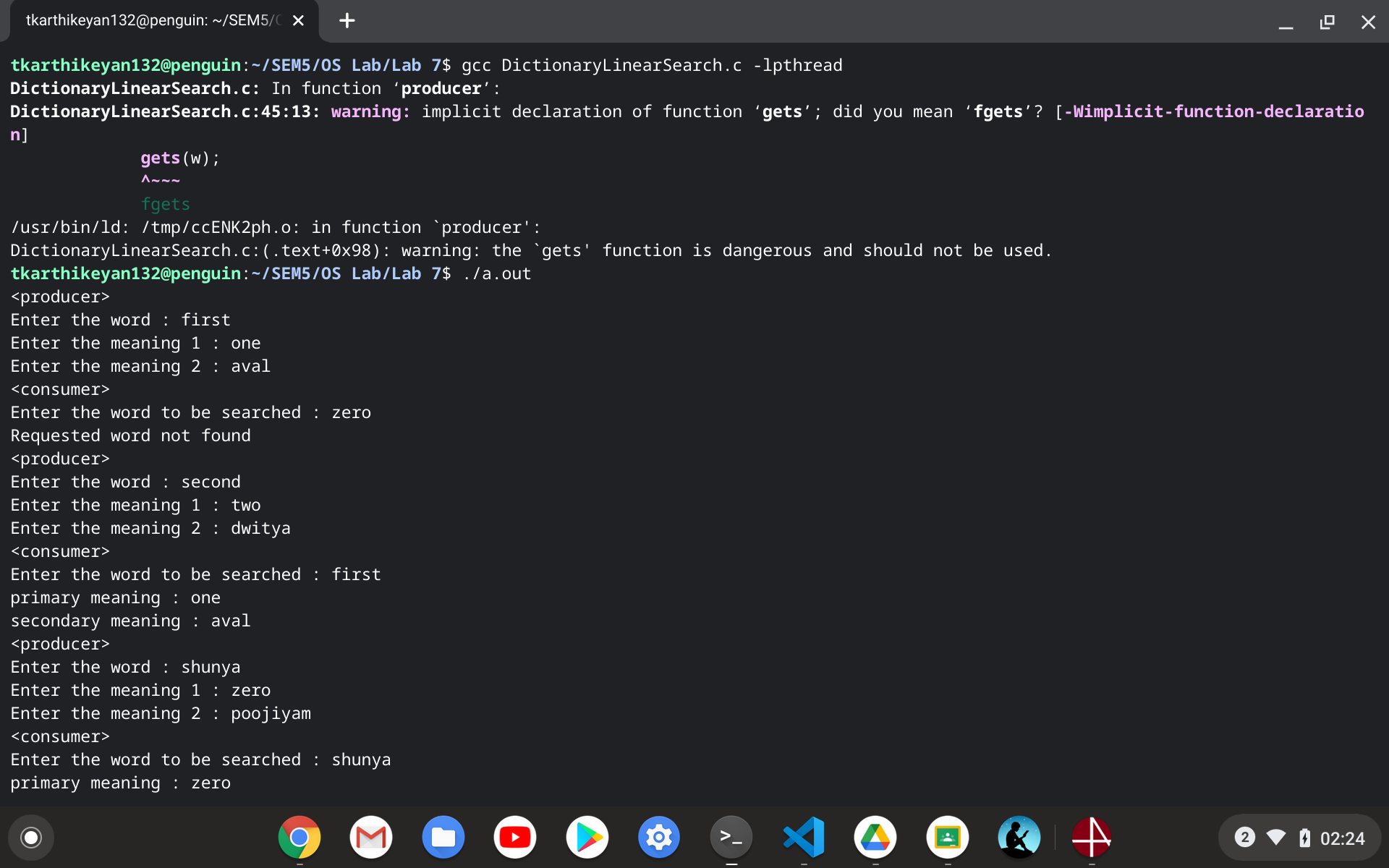
**pthread\_join(pid, NULL);**

**pthread\_join(cid, NULL);**

**return 0;**

**}**

**Output**

****

**Non-Mandatory (Extra credits):**

**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 solution for synchronization*

*Binary Search is applied for searching*

*Binary search is used before inserting in dictionary to avoid duplication*

**Code**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<stdbool.h>**

**#include<pthread.h>**

**#include<time.h>**

**#include<string.h>**

**#define DICT\_LIM 20**

**int NUM\_WORDS = 0; //number of words in dictionary**

**struct word**

**{**

**char word[25];**

**char meaning1[100];**

**char meaning2[100];**

**};**

**struct word W[DICT\_LIM];**

**//0:Producer**

**//1:Consumer**

**//flag[i] = true implies P'i wants to enter critical section**

**bool flag[2] = {false, false};**

**//turn = i implies P'i turn**

**int turn = -1;**

**struct block**

**{**

**char word\_to\_be\_searched[25];**

**int beg;**

**int end;**

**int flag;**

**};**

**struct block\_check**

**{**

**char word\_to\_be\_searched[25];**

**int beg;**

**int end;**

**int flag;**

**};**

**void \*search(void \*param)**

**{**

**struct block \*b = param;**

**for(int i = b->beg; i <= (b->end); i++)**

**{**

**if(strcmp(b->word\_to\_be\_searched,W[i].word) == 0)**

**{**

**printf("primary meaning : %s\n", W[i].meaning1);**

**printf("secondary meaning : %s\n", W[i].meaning2);**

**b->flag = 0;**

**break;**

**}**

**}**

**}**

**void \*search\_check(void \*param)**

**{**

**struct block\_check \*b = param;**

**for(int i = b->beg; i <= (b->end); i++)**

**{**

**if(strcmp(b->word\_to\_be\_searched,W[i].word) == 0)**

**{**

**printf("Word already exists in the dictionary\n");**

**printf("primary meaning : %s\n", W[i].meaning1);**

**printf("secondary meaning : %s\n", W[i].meaning2);**

**b->flag = 0;**

**break;**

**}**

**}**

**}**

**void \*producer(void \*param)**

**{**

**do**

**{**

**flag[0] = true;**

**turn = 1;**

**//being humble towards consumer**

**while(flag[1] && turn == 1);**

**//critical section begins**

**if(NUM\_WORDS == DICT\_LIM)**

**printf("Dictionary is full\n");**

**else**

**{**

**char w[25];**

**char mean1[100];**

**char mean2[100];**

**printf("<producer>\n");**

**printf("Enter the word : ");**

**gets(w);**

**pthread\_t aid, bid;**

**pthread\_attr\_t attra, attrb;**

**pthread\_attr\_init(&attra);**

**pthread\_attr\_init(&attrb);**

**int mid = NUM\_WORDS/2;**

**struct block\_check x, y;**

**strcpy(x.word\_to\_be\_searched, w);**

**strcpy(y.word\_to\_be\_searched, w);**

**x.beg = 0;**

**x.end = mid - 1;**

**y.beg = mid;**

**y.end = NUM\_WORDS - 1;**

**x.flag = 1;**

**y.flag = 1;**

**pthread\_create(&aid, &attra, search\_check, &x);**

**pthread\_create(&bid, &attrb, search\_check, &y);**

**pthread\_join(aid, NULL);**

**pthread\_join(bid, NULL);**

**if(x.flag && y.flag)**

**{**

**printf("Enter the meaning 1 : ");**

**gets(mean1);**

**printf("Enter the meaning 2 : ");**

**gets(mean2);**

**strcpy(W[NUM\_WORDS].word, w);**

**strcpy(W[NUM\_WORDS].meaning1, mean1);**

**strcpy(W[NUM\_WORDS].meaning2, mean2);**

**NUM\_WORDS++;**

**}**

**}**

**//critical section ends**

**flag[0] = false;**

**} while (true);**

**pthread\_exit(0);**

**}**

**void \*consumer(void \*param)**

**{**

**do**

**{**

**flag[1] = true;**

**turn = 0;**

**//being humble towards producer**

**while(flag[0] && turn == 0);**

**//critical section begins**

**if(NUM\_WORDS == 0)**

**printf("Dictionary is empty\n");**

**else**

**{**

**char w[25];**

**printf("<consumer>\n");**

**printf("Enter the word to be searched : ");**

**gets(w);**

**pthread\_t pid, qid;**

**pthread\_attr\_t attrp, attrq;**

**pthread\_attr\_init(&attrp);**

**pthread\_attr\_init(&attrq);**

**int mid = NUM\_WORDS/2;**

**struct block p, c;**

**strcpy(p.word\_to\_be\_searched, w);**

**strcpy(c.word\_to\_be\_searched, w);**

**p.beg = 0;**

**p.end = mid;**

**c.beg = mid + 1;**

**c.end = NUM\_WORDS - 1;**

**p.flag = 1;**

**c.flag = 1;**

**pthread\_create(&pid, &attrp, search, &p);**

**pthread\_create(&qid, &attrq, search, &c);**

**pthread\_join(pid, NULL);**

**pthread\_join(qid, NULL);**

**if(p.flag && c.flag)**

**printf("Requested word not found\n");**

**}**

**//critical section ends**

**flag[1] = false;**

**} while (true);**

**pthread\_exit(0);**

**}**

**int main(int argc, char \*argv[])**

**{**

**pthread\_t pid, cid;**

**pthread\_attr\_t attr1, attr2;**

**pthread\_attr\_init(&attr1);**

**pthread\_attr\_init(&attr2);**

**pthread\_create(&pid, &attr1, producer, NULL);**

**pthread\_create(&cid, &attr2, consumer, NULL);**

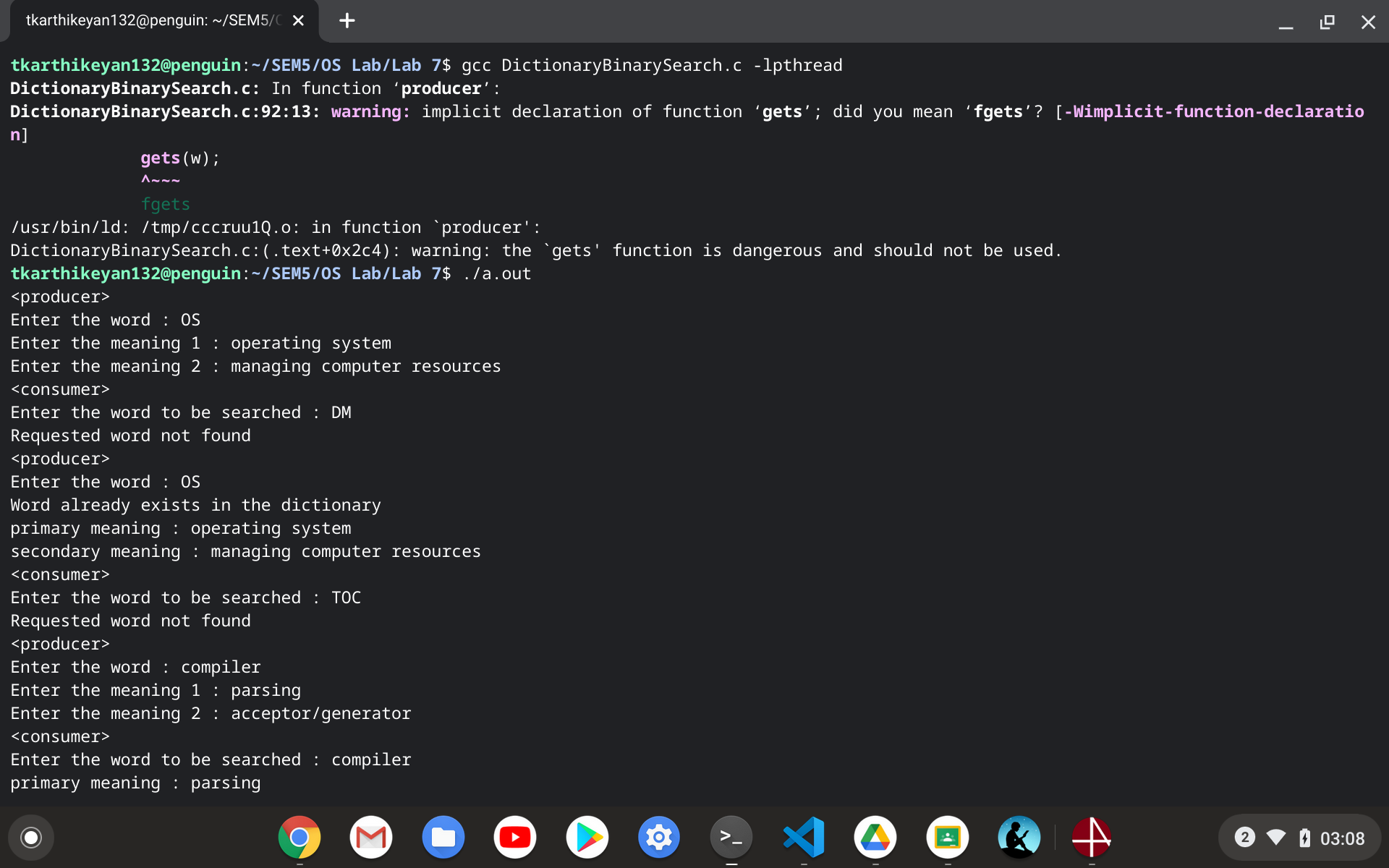
**pthread\_join(pid, NULL);**

**pthread\_join(cid, NULL);**

**return 0;**

**}**

**Output**

****

**5. Trace and understand the working of synchronization algorithms like Dijkstra, Dekker’s algorithm**

**DEKKERs ALGORITHM**

*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.*

**CODE**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<stdbool.h>**

**#include<pthread.h>**

**bool thread1\_wants\_to\_enter = false;**

**bool thread2\_wants\_to\_enter = false;**

**int favoured\_thread = 1;**

**int data = 0;**

**void \*runner1(void \*param);**

**void \*runner2(void \*param);**

**int main()**

**{**

**pthread\_t tid[2];**

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

**pthread\_create(&tid[0], NULL, runner1, &ttid[0]);**

**pthread\_create(&tid[1], NULL, runner2, &ttid[1]);**

**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**

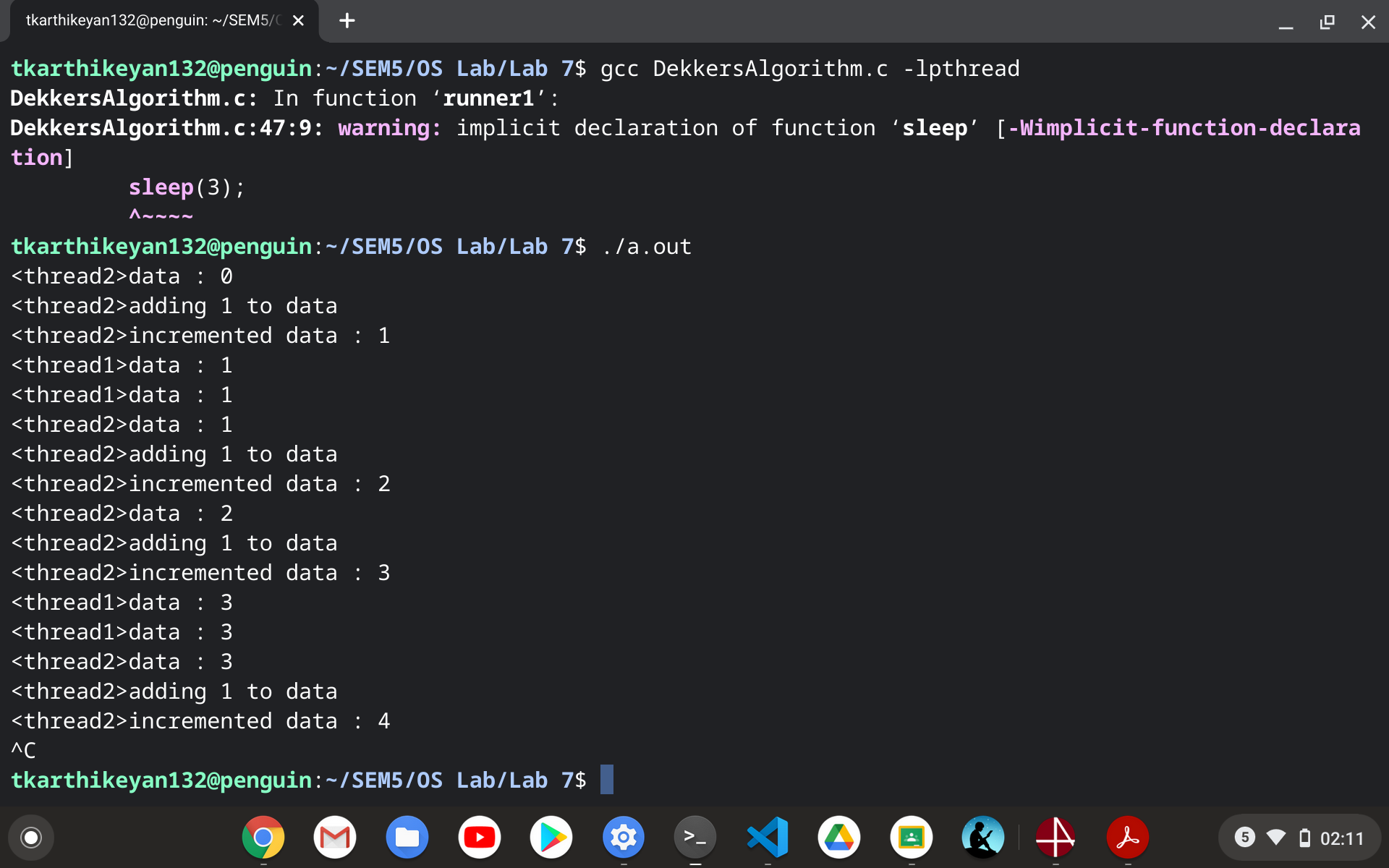
**favoured\_thread = 1;**

**thread2\_wants\_to\_enter = false;**

**} while (1);**

**}**

**OUTPUT**

****

**--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**LATIN MODULE**

1. **Latin Generator**

**CODE**

**#include <pthread.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include <math.h>**

**#include <time.h>**

**struct block**

**{**

**int \*\*arr;**

**int n;**

**int resp;**

**};**

**void \*runner(void \*param)**

**{**

**struct block \*b = param;**

**int temp = b->resp;**

**int index = b->resp - 1;**

**int n = b->n;**

**for(int i = 0; i < b->n; i++)**

**{**

**b->arr[index][i] = temp;**

**temp++;**

**if(temp == (n+1))**

**temp = 1;**

**}**

**pthread\_exit(0);**

**}**

**int main()**

**{**

**int n;**

**printf("Enter the degree of the matrix : ");**

**scanf("%d", &n);**

**int \*\*arr = malloc(sizeof(int \*)\*n);**

**for(int i = 0; i < n; i++)**

**arr[i] = malloc(sizeof(int)\*n);**

**struct block b[n];**

**for(int i = 0; i < n; i++)**

**{**

**b[i].arr = arr;**

**b[i].n = n;**

**b[i].resp = i+1;**

**}**

**pthread\_t tid[n];**

**pthread\_attr\_t attr[n];**

**for(int i = 0; i < n; i++)**

**pthread\_attr\_init(&attr[i]);**

**for(int i = 0; i < n; i++)**

**{**

**pthread\_create(&tid[i], &attr[i], runner, &b[i]);**

**}**

**for(int i = 0; i < n; i++)**

**{**

**pthread\_join(tid[i], NULL);**

**}**

**for(int i = 0; i < n; i++)**

**{**

**for(int j = 0; j < n; j++)**

**printf("%d ",arr[i][j]);**

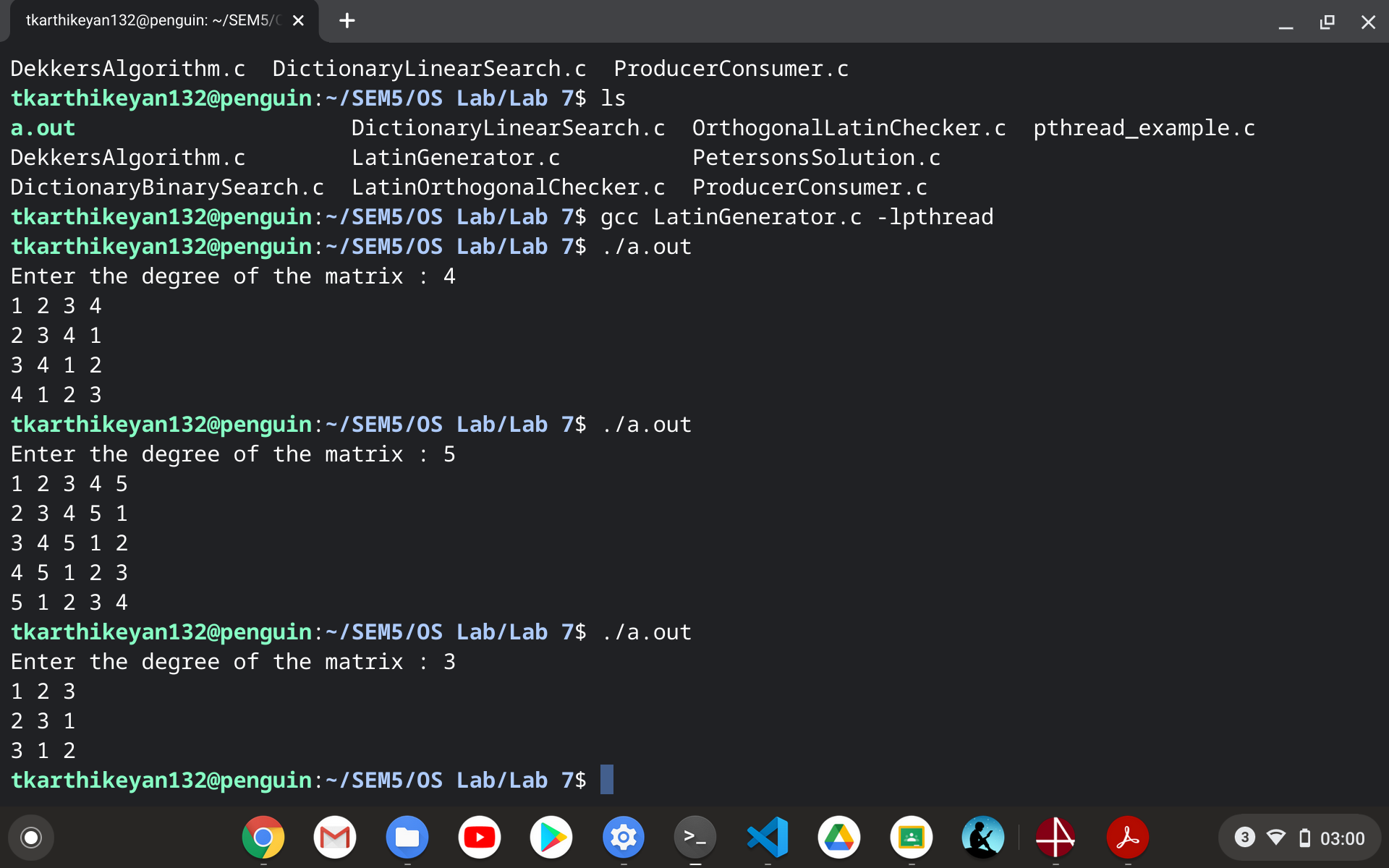
**printf("\n");**

**}**

**return 0;**

**}**

**OUTPUT**

****

1. **Latin Orthogonal Checker**

**CODE**

**#include <pthread.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include <math.h>**

**#include <time.h>**

**struct block**

**{**

**int \*\*orr;**

**int x;**

**int y;**

**};**

**void \*runner(void \*param)**

**{**

**struct block \*b = param;**

**b->orr[b->x][b->y] = 1;**

**}**

**\_Bool isOrthogonal(int \*\*orr, int n) // can be parallelised**

**{**

**for(int i = 0; i < n; i++)**

**{**

**for(int j = 0; j < n; j++)**

**{**

**if(orr[i][j] != 1)**

**return 0;**

**}**

**}**

**return 1;**

**}**

**int main()**

**{**

**int n;**

**printf("Enter the degree of the matrix : ");**

**scanf("%d", &n);**

**int \*\*arr = malloc(sizeof(int \*)\*n);**

**for(int i = 0; i < n; i++)**

**arr[i] = malloc(sizeof(int)\*n);**

**printf("Enter Matrix 1 (row-wise) : ");**

**for(int i = 0; i < n; i++)**

**for(int j = 0; j < n; j++)**

**scanf("%d", &arr[i][j]);**

**int \*\*brr = malloc(sizeof(int \*)\*n);**

**for(int i = 0; i < n; i++)**

**brr[i] = malloc(sizeof(int)\*n);**

**printf("Enter Matrix 2 (row-wise) : ");**

**for(int i = 0; i < n; i++)**

**for(int j = 0; j < n; j++)**

**scanf("%d", &brr[i][j]);**

**int \*\*orr = malloc(sizeof(int \*)\*n);**

**for(int i = 0; i < n; i++)**

**orr[i] = malloc(sizeof(int)\*n);**

**for(int i = 0; i < n; i++)**

**for(int j = 0; j < n; j++)**

**orr[i][j] = -1;**

**int \*hash1 = malloc(sizeof(int)\*10000);**

**int \*hash2 = malloc(sizeof(int)\*10000);**

**int index1 = 0, index2 = 0;**

**for(int i = 0; i < n; i++)**

**{**

**hash1[arr[0][i]] = index1;**

**hash2[brr[0][i]] = index2;**

**index1++;**

**index2++;**

**}**

**int \*\*H1 = malloc(sizeof(int \*)\*n);**

**for(int i = 0; i < n; i++)**

**H1[i] = malloc(sizeof(int)\*n);**

**int \*\*H2 = malloc(sizeof(int \*)\*n);**

**for(int i = 0; i < n; i++)**

**H2[i] = malloc(sizeof(int)\*n);**

**for(int i = 0; i < n; i++)**

**for(int j = 0; j < n; j++)**

**{**

**H1[i][j] = hash1[arr[i][j]];**

**H2[i][j] = hash2[brr[i][j]];**

**}**

**for(int i = 0; i < n; i++)**

**{**

**for(int j = 0; j < n; j++)**

**printf("%d ", H1[i][j]);**

**printf("\n");**

**}**

**for(int i = 0; i < n; i++)**

**{**

**for(int j = 0; j < n; j++)**

**printf("%d ", H2[i][j]);**

**printf("\n");**

**}**

**//orthogonal verification**

**int NUM\_THREAD\_MUL = n\*n;**

**struct block b[NUM\_THREAD\_MUL];**

**for(int i = 0; i < NUM\_THREAD\_MUL; i++)**

**{**

**int x = i/n;**

**int y = i%n;**

**b[i].orr = orr;**

**b[i].x = H1[x][y];**

**b[i].y = H2[x][y];**

**}**

**pthread\_t tid[NUM\_THREAD\_MUL];**

**pthread\_attr\_t attr[NUM\_THREAD\_MUL];**

**for(int i = 0; i < NUM\_THREAD\_MUL; i++)**

**pthread\_attr\_init(&attr[i]);**

**for(int i = 0; i < NUM\_THREAD\_MUL; i++)**

**{**

**pthread\_create(&tid[i], &attr[i], runner, &b[i]);**

**}**

**for(int i = 0; i < NUM\_THREAD\_MUL; i++)**

**{**

**pthread\_join(tid[i], NULL);**

**}**

**for(int i = 0; i < n; i++)**

**{**

**for(int j = 0; j < n; j++)**

**printf("%d ", orr[i][j]);**

**printf("\n");**

**}**

**if(isOrthogonal(orr,n))**

**printf("Matrix is orthogonal\n");**

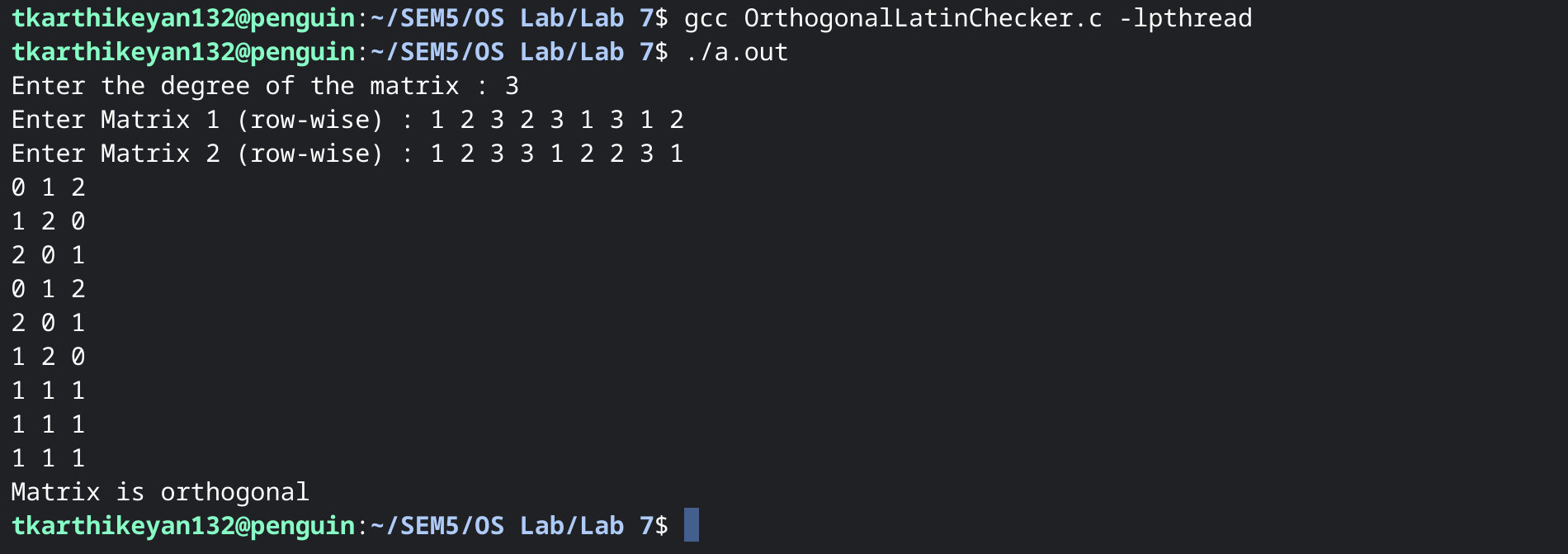
**else**

**printf("Matrix is not orthogonal\n");**

**return 0;**

**}**

**OUTPUT**

****