**Name-Ujjawal Mandhani**

**Batch-F6**

**Enroll no.-9918103237**

**OSSP Lab – 6**

**Question 1:**To avoid deadlock in dining philosophers’ problem use a possible solution as the odd

numbered philosophers grab the right and then the left. Implement this solution using

pthread mutual exclusion lock.

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**#define N 5**

**#define THINKING 2**

**#define HUNGRY 1**

**#define EATING 0**

**#define LEFT (phnum + 4) % N**

**#define RIGHT (phnum + 1) % N**

**int state[N];**

**int phil[N] = {0, 1, 2, 3, 4};**

**sem\_t mutex;**

**sem\_t S[N];**

**void test(int phnum)**

**{**

**if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)**

**{**

**// state that eating**

**state[phnum] = EATING;**

**sleep(2);**

**printf("Philosopher %d takes fork %d and %d\n",**

**phnum + 1, LEFT + 1, phnum + 1);**

**printf("Philosopher %d is Eating\n", phnum + 1);**

**// sem\_post(&S[phnum]) has no effect**

**// during takefork**

**// used to wake up hungry philosophers**

**// during putfork**

**sem\_post(&S[phnum]);**

**}**

**}**

**// take up chopsticks**

**void take\_fork(int phnum)**

**{**

**sem\_wait(&mutex);**

**// state that hungry**

**state[phnum] = HUNGRY;**

**printf("Philosopher %d is Hungry\n", phnum + 1);**

**// eat if neighbours are not eating**

**test(phnum);**

**sem\_post(&mutex);**

**// if unable to eat wait to be signalled**

**sem\_wait(&S[phnum]);**

**sleep(1);**

**}**

**// put down chopsticks**

**void put\_fork(int phnum)**

**{**

**sem\_wait(&mutex);**

**// state that thinking**

**state[phnum] = THINKING;**

**printf("Philosopher %d putting fork %d and %d down\n",**

**phnum + 1, LEFT + 1, phnum + 1);**

**printf("Philosopher %d is thinking\n", phnum + 1);**

**test(LEFT);**

**test(RIGHT);**

**sem\_post(&mutex);**

**}**

**void \*philospher(void \*num)**

**{**

**while (1)**

**{**

**int \*i = num;**

**sleep(1);**

**take\_fork(\*i);**

**sleep(0);**

**put\_fork(\*i);**

**}**

**}**

**int main()**

**{**

**int i;**

**pthread\_t thread\_id[N];**

**// initialize the semaphores**

**sem\_init(&mutex, 0, 1);**

**for (i = 0; i < N; i++)**

**sem\_init(&S[i], 0, 0);**

**for (i = 0; i < N; i++)**

**{**

**// create philosopher processes**

**pthread\_create(&thread\_id[i], NULL,**

**philospher, &phil[i]);**

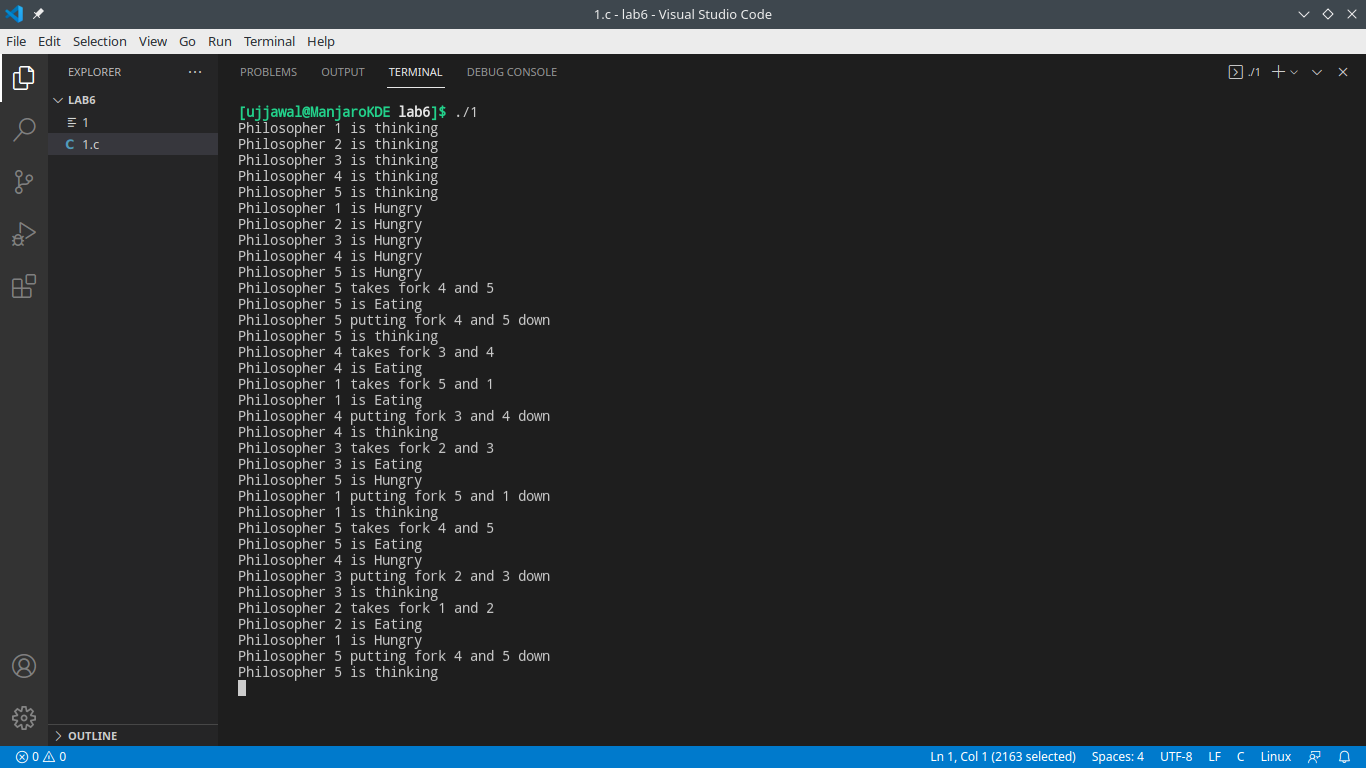
**printf("Philosopher %d is thinking\n", i + 1);**

**}**

**for (i = 0; i < N; i++)**

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

**}**



**Question 2:** Using condition variables to implement a producer-consumer algorithm. Define two

threads: one producer and one consumer. The producer reads characters one by one from

a string stored in a file named “string.txt”, then writes sequentially these characters into a

circular queue. Meanwhile, the consumer reads sequentially from the queue and prints

them in the same order. The diagram illustrates the process. Upon completion of running

the program, “Hello! World.” is printed on the screen. In the program, use #define to

specify the size of the queue. For example, #define QUEUE\_SIZE 5. Make sure to test

your program with different queue sizes, including 1.

**// here I used an array to implement the queu**

**#include <pthread.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#define QUEU\_SIZE 5**

**char queue[QUEU\_SIZE];**

**int front = 0, rear = 0; // pointrs to front and rear of the queu**

**int items = 0; // number of items in queu**

**int done = 1; // to show that the input file is finished to read**

**pthread\_mutex\_t mutex;**

**pthread\_cond\_t item\_available; // condition variable to show producer put a word in queu**

**pthread\_cond\_t space\_available; // condition variable to show consumer delete a word from queu**

**void \*p\_thread(); // the producer**

**void \*c\_thread(); // the consumer**

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

**{**

**pthread\_t producer, consumer;**

**//pthread\_init();**

**pthread\_cond\_init(&item\_available, NULL);**

**pthread\_cond\_init(&space\_available, NULL);**

**pthread\_mutex\_init(&mutex, NULL);**

**if (pthread\_create(&producer, NULL, p\_thread, NULL)) // create producer**

**{**

**fprintf(stderr, "Error creating producer thread\n");**

**return 1;**

**}**

**if (pthread\_create(&consumer, NULL, c\_thread, NULL)) // create the consumer**

**{**

**fprintf(stderr, "Error creating consumer thread\n");**

**return 1;**

**}**

**if (pthread\_join(producer, NULL)) // wait for all threads to finish**

**{**

**fprintf(stderr, "Error joining thread\n");**

**return 2;**

**}**

**if (pthread\_join(consumer, NULL))**

**{**

**fprintf(stderr, "Error joining consumer\n");**

**return 2;**

**}**

**return 0;**

**}**

**void \*p\_thread()**

**{**

**FILE \*fp = fopen("message.txt", "r");**

**char c;**

**c = getc(fp);**

**while (c != EOF)**

**{**

**//1**

**sleep(1);**

**pthread\_mutex\_lock(&mutex);**

**//2**

**printf(" front<%d> rear<%d> items <%d>\n", front, rear, items);**

**while (items >= QUEU\_SIZE) // while there is no space in queu to write**

**pthread\_cond\_wait(&space\_available, &mutex);**

**//3**

**printf(" front<%d> rear<%d> items <%d>\n", front, rear, items);**

**// now we cAN write in queue**

**queue[front] = c;**

**front++;**

**if (front == QUEU\_SIZE)**

**front = 0;**

**items++;**

**printf(" character write to queue: <%c>\n", c);**

**printf(" wake up a consumer \n");**

**pthread\_cond\_signal(&item\_available); // wake up a consumer**

**pthread\_mutex\_unlock(&mutex);**

**sleep(1);**

**c = getc(fp);**

**}**

**pthread\_mutex\_lock(&mutex);**

**done = 0; // we should tell the consumer that the file is finished**

**pthread\_cond\_signal(&item\_available);**

**pthread\_mutex\_unlock(&mutex);**

**fclose(fp);**

**//printf ("hello prof. Song Jiang\n") ;**

**//return NULL;**

**pthread\_exit(0);**

**}**

**void \*c\_thread()**

**{**

**FILE \*output = fopen("message\_result.txt", "w");**

**//4**

**while (done != 0)**

**{ // while there is something to read**

**pthread\_mutex\_lock(&mutex);**

**//5**

**printf("front<%d> rear<%d> items <%d>\n", front, rear, items);**

**while (items <= 0 && done != 0) // while there is no character in queu to read we should wait**

**pthread\_cond\_wait(&item\_available, &mutex);**

**//6**

**printf("front<%d> rear<%d> items <%d>\n", front, rear, items);**

**// read the character and write it:**

**char c = queue[rear];**

**rear++;**

**if (rear == QUEU\_SIZE)**

**rear = 0;**

**items--;**

**printf("character read from queue: <%c>\n", c);**

**printf("wake up a producer \n");**

**fprintf(output, "%c", c);**

**pthread\_cond\_signal(&space\_available); // send signal for producer to show that thre is space**

**pthread\_mutex\_unlock(&mutex);**

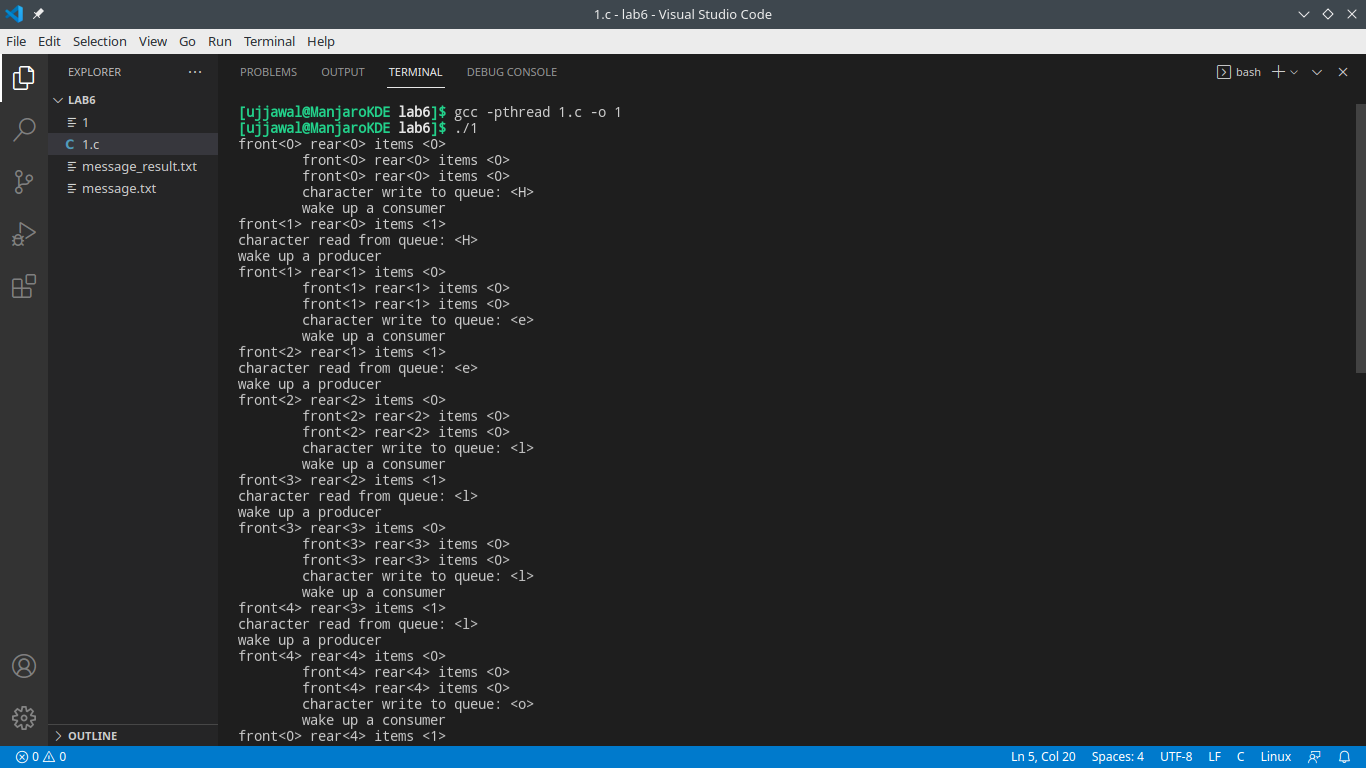
**}**

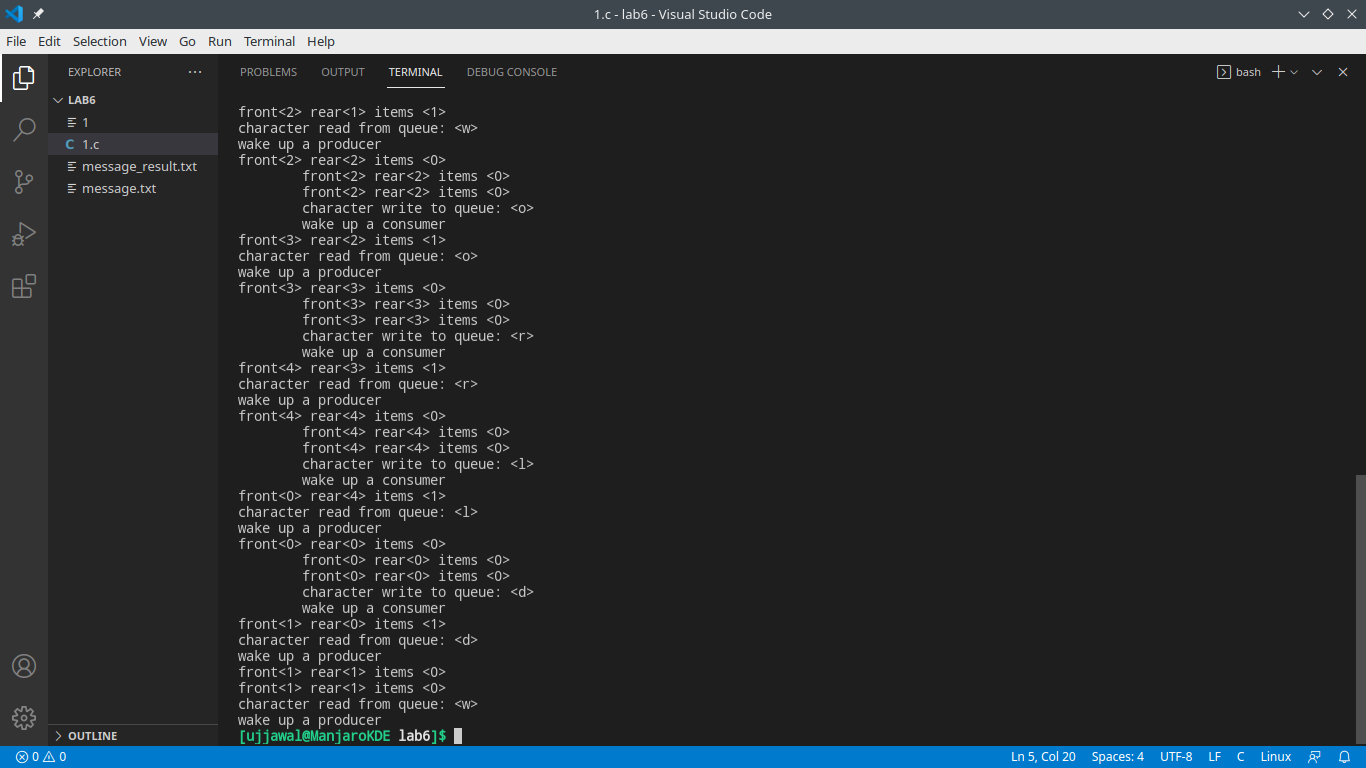
**fclose(output);**

**//return NULL;**

**pthread\_exit(0);**

**}**





**Question 3:**Consider a system with: five processes, P0 P4, three resource types, A, B, and C. Type

A has 10 instances, B has 5 instances, C has 7 instances. At time T0 the following

snapshot of the system is taken.

Write a Linux C program to check weather system is in safe state or not. Also

demonstrate the unsafe sequence by modifying any resource allocation.

**// Banker's Algorithm**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**// P0, P1, P2, P3, P4 are the Process names here**

**int n, m, i, j, k;**

**n = 5; // Number of processes**

**m = 3; // Number of resources**

**int alloc[5][3] = {{0, 1, 0}, // P0 // Allocation Matrix**

**{2, 0, 0}, // P1**

**{3, 0, 2}, // P2**

**{2, 1, 1}, // P3**

**{0, 0, 2}}; // P4**

**int max[5][3] = {{7, 5, 3}, // P0 // MAX Matrix**

**{3, 2, 2}, // P1**

**{9, 0, 2}, // P2**

**{2, 2, 2}, // P3**

**{4, 3, 3}}; // P4**

**int avail[3] = {3, 3, 2}; // Available Resources**

**int f[n], ans[n], ind = 0;**

**for (k = 0; k < n; k++)**

**{**

**f[k] = 0;**

**}**

**int need[n][m];**

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

**{**

**for (j = 0; j < m; j++)**

**need[i][j] = max[i][j] - alloc[i][j];**

**}**

**int y = 0;**

**for (k = 0; k < 5; k++)**

**{**

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

**{**

**if (f[i] == 0)**

**{**

**int flag = 0;**

**for (j = 0; j < m; j++)**

**{**

**if (need[i][j] > avail[j])**

**{**

**flag = 1;**

**break;**

**}**

**}**

**if (flag == 0)**

**{**

**ans[ind++] = i;**

**for (y = 0; y < m; y++)**

**avail[y] += alloc[i][y];**

**f[i] = 1;**

**}**

**}**

**}**

**}**

**cout << "Following is the SAFE Sequence" << endl;**

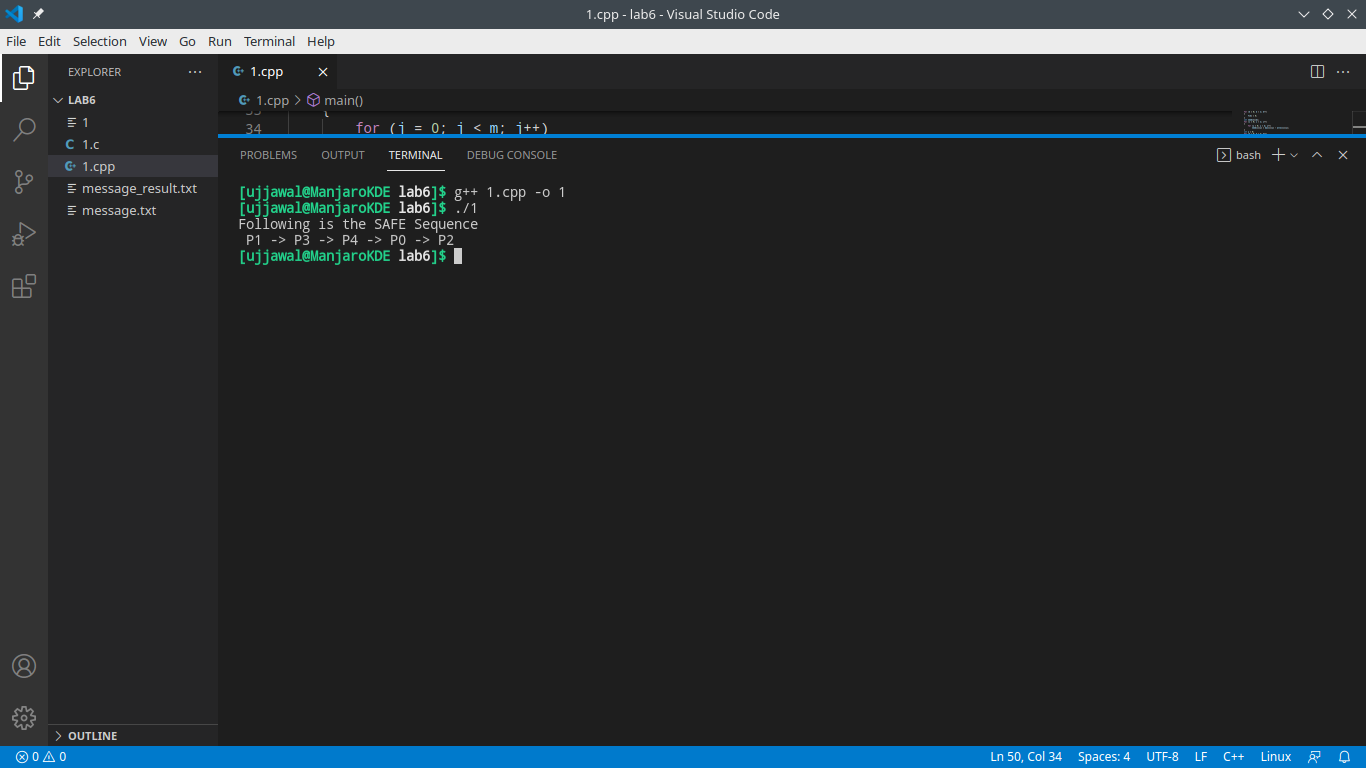
**for (i = 0; i < n - 1; i++)**

**cout << " P" << ans[i] << " ->";**

**cout << " P" << ans[n - 1] << endl;**

**return (0);**

**}**



**Question 4:**Write a Linux C program to demonstrate that deadlock and starvation are opposite

problems. Solution to deadlock problem cause more starvation and solution to the

starvation problem can cause deadlock.

**#include <unistd.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <stdio.h>**

**#define N 5**

**//a change here**

**#define HOLDRIGHT 4**

**#define HOLDLEFT 3**

**#define THINKING 2**

**#define HUNGRY 1**

**#define EATING 0**

**#define LEFT (phnum + 4) % N**

**#define RIGHT (phnum + 1) % N**

**int state[N];**

**int phil[N] = {0, 1, 2, 3, 4};**

**sem\_t mutex;**

**sem\_t S[N];**

**//a change here**

**void testright(int phnum)**

**{**

**if (state[phnum] == HOLDLEFT && phnum % 2 == 0 && state[RIGHT] != EATING && state[RIGHT] != HOLDLEFT)**

**{**

**// state that eating**

**state[phnum] = EATING;**

**sleep(2);**

**printf("Philosopher %d now holds fork %d and %d\n",**

**phnum + 1, LEFT + 1, phnum + 1);**

**printf("Philosopher %d is Eating\n", phnum + 1);**

**// sem\_post(&S[phnum]) has no effect**

**// during takefork**

**// used to wake up hungry philosophers**

**// during putfork**

**sem\_post(&S[phnum]);**

**}**

**//a change here**

**else**

**{**

**printf("Philosopher %d Cant Eat\n", phnum + 1);**

**state[phnum] = HUNGRY;**

**sleep(2);**

**sem\_post(&S[phnum]);**

**}**

**}**

**//a change here**

**void testleft(int phnum)**

**{**

**if (state[phnum] == HUNGRY && state[LEFT] != HOLDRIGHT && state[LEFT] != EATING)**

**{**

**// state that eating**

**state[phnum] = HOLDLEFT;**

**printf("Philosopher %d is holding left fork\n", phnum + 1);**

**testright(phnum);**

**}**

**}**

**// take up chopsticks**

**void take\_fork(int phnum)**

**{**

**sem\_wait(&mutex);**

**// state that hungry**

**state[phnum] = HUNGRY;**

**printf("Philosopher %d is Hungry\n", phnum + 1);**

**// eat if neighbours are not eating**

**testleft(phnum);**

**sem\_post(&mutex);**

**// if unable to eat wait to be signalled**

**sem\_wait(&S[phnum]);**

**sleep(1);**

**}**

**// put down chopsticks**

**void put\_fork(int phnum)**

**{**

**sem\_wait(&mutex);**

**// state that thinking**

**state[phnum] = THINKING;**

**printf("Philosopher %d putting fork down\n",**

**phnum + 1);**

**printf("Philosopher %d is thinking\n", phnum + 1);**

**testleft(LEFT);**

**testleft(RIGHT);**

**sem\_post(&mutex);**

**}**

**void \*philospher(void \*num)**

**{**

**while (1)**

**{**

**int \*i = num;**

**sleep(1);**

**take\_fork(\*i);**

**sleep(0);**

**put\_fork(\*i);**

**}**

**}**

**int main()**

**{**

**int i;**

**pthread\_t thread\_id[N];**

**// initialize the semaphores**

**sem\_init(&mutex, 0, 1);**

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

**sleep(1);**

**for (i = 0; i < N; i++)**

**sem\_init(&S[i], 0, 0);**

**for (i = 0; i < N; i++)**

**{**

**// create philosopher processes**

**pthread\_create(&thread\_id[i], NULL,**

**philospher, &phil[i]);**

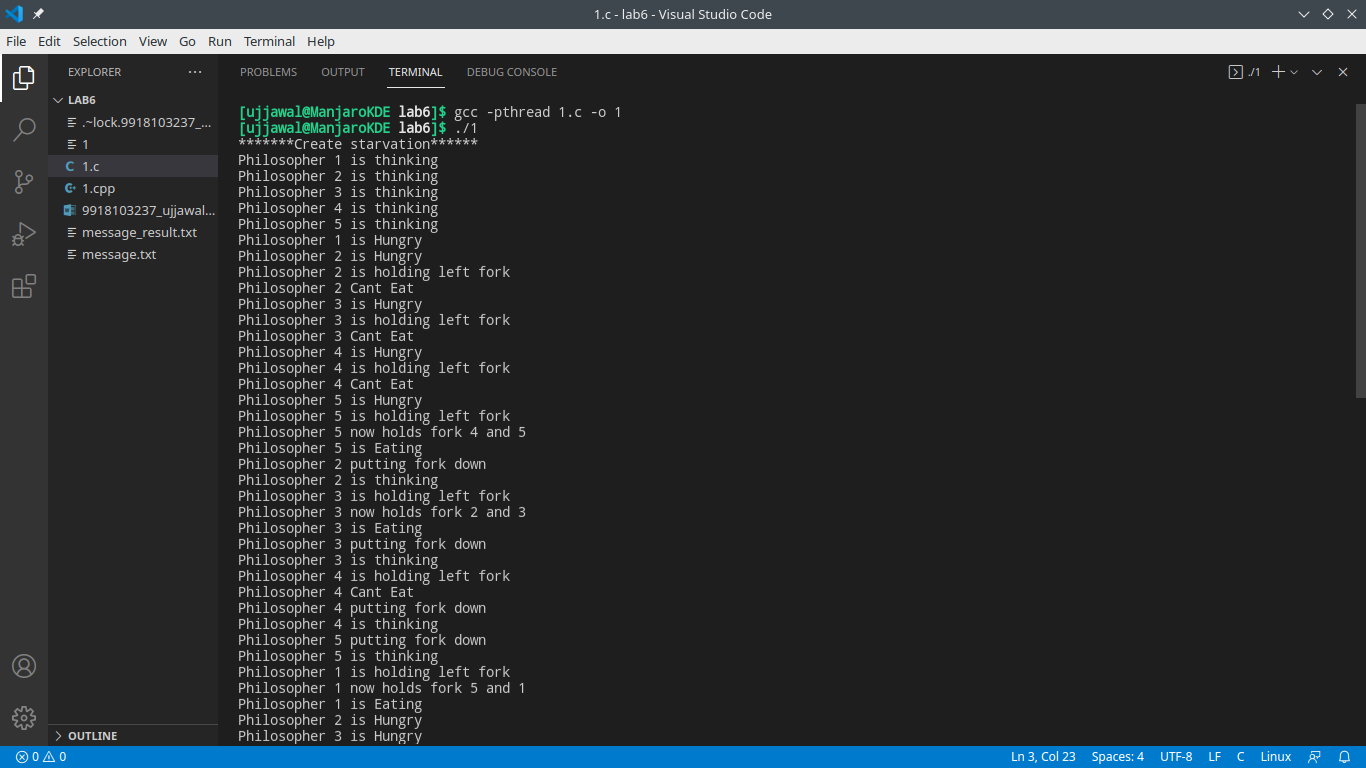
**printf("Philosopher %d is thinking\n", i + 1);**

**}**

**for (i = 0; i < N; i++)**

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

**}**

****