***21K-3278***

***Sohaib Sarosh Shamsi***

***OS - AO3***

**Q1.**

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <pthread.h>**

**int mails = 0;**

**pthread\_mutex\_t mutex;**

**void\* fun1() {**

**for (int i = 0; i < 10000000; i++) {**

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

**// critical section**

**mails++; // can produce race condition : reads value -> +1 -> stores (not atomic)**

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

**}**

**}**

**int main(void) {**

**pthread\_t p1, p2;**

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

**// thread creation**

**if (pthread\_create(&p1, NULL, &fun1, NULL) != 0) {**

**printf("unable to create thread");**

**exit(0);**

**}**

**if (pthread\_create(&p2, NULL, &fun1, NULL) != 0) {**

**printf("unable to create thread");**

**exit(0);**

**}**

**// join them**

**if (pthread\_join(p1, NULL) != 0) {**

**printf("unable to join thread");**

**exit(0);**

**}**

**if (pthread\_join(p2, NULL) != 0) {**

**printf("unable to join thread");**

**exit(0);**

**}**

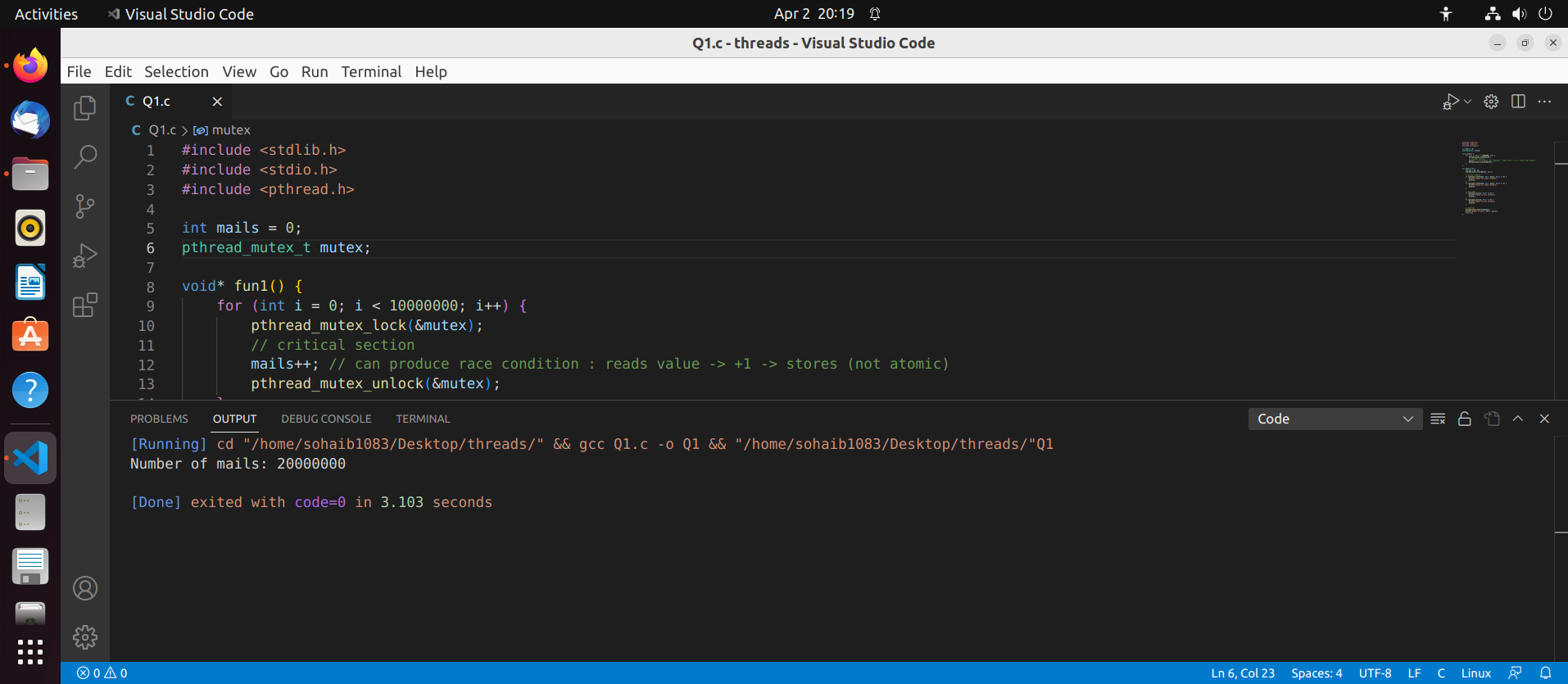
**// optional**

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

**printf("Number of mails: %d\n", mails);**

**return 0;**

**}**

****

**Mutual exclusion:**

If a thread is executing in its critical section, then no other thread can enter in their critical section. Whoever comes first will acquire the lock and other incoming threads will continue to wait (busy waiting) till it unlocks.

**Progress:**

If no thread is executing in its critical section and some thread wishes to enter its critical sections, then it can enter the critical section by acquiring the lock only if it is not executing in its remainder section. Also any thread can enter first since getting the lock is unconditional.

**Bounded waiting:**

The thread won't have to wait for an indefinite time since the lock is acquired on a first come basis. There is no chance for starvation to occur since every thread gets a chance to take CPU’s attention.

**Q2.**

**#include <pthread.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <semaphore.h>**

**sem\_t semaphore;**

**void\* fun1(void\* args) {**

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

**sleep(1);**

**// Critical section**

**printf("I'm in thread %d\n", \*(int\*)args);**

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

**free(args);**

**}**

**int main(void) {**

**// six threads**

**pthread\_t th[6];**

**sem\_init(&semaphore, 0, 3); // using counting semaphores**

**int i;**

**for (i = 0; i < 6; i++) {**

**int\* a = malloc(sizeof(int));**

**\*a = i;**

**if (pthread\_create(&th[i], NULL, &fun1, a) != 0) {**

**printf("unable to create thread");**

**exit(0);**

**}**

**}**

**for (i = 0; i < 6; i++) {**

**if (pthread\_join(th[i], NULL) != 0) {**

**printf("unable to join thread");**

**exit(0);**

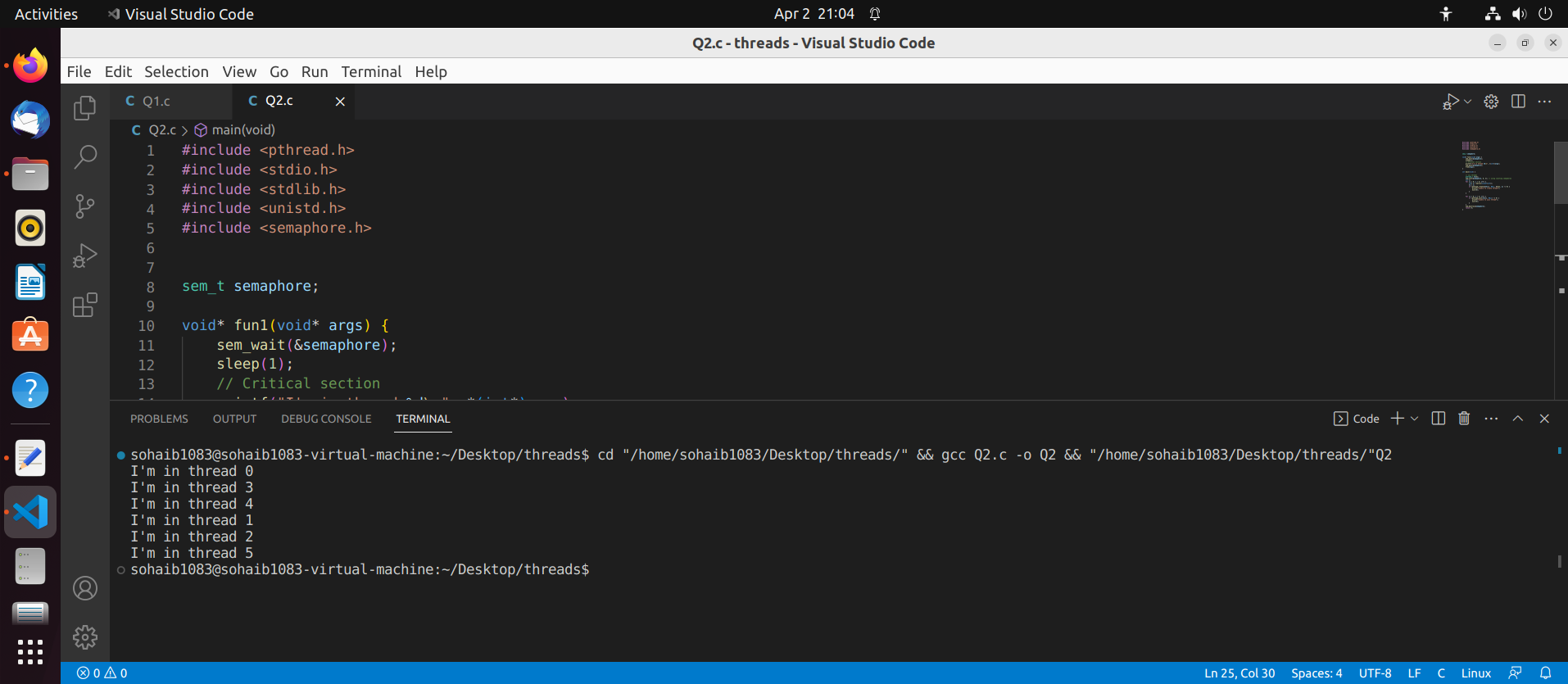
**}**

**}**

**sem\_destroy(&semaphore);**

**return 0;**

**}**

****

**Mutual exclusion:**

If the semaphore value is greater than 1, the process or thread acquires the semaphore. If the semaphore value is 0 then the thread waits until the semaphore is released by the other process or thread that currently holds it.

**Progress:**

When a thread wants to access the shared resource, it acquires the semaphore. If the semaphore count is greater than 0, the process or thread acquires the semaphore and enters the critical section. If the semaphore count is 0, the process or thread is added to the semaphore's queue. When the semaphore is released by the process or thread that currently holds it, the next process or thread in the queue is granted access to the critical section. This ensures that progress is made.

**Bounded waiting:**

If the semaphore count is 0, the process or thread waits for a specific amount of time, after which it gives up waiting and increments the semaphore. This ensures that no thread has to wait indefinitely to access the critical section.