Department of Computer Science

Subject:

OPERATING SYSTEM

Submitted by:

ABDUL REHMAN SUDAIS

Reg number:

23-NTU-CS-1123

Lab :

10

Semester:

5TH

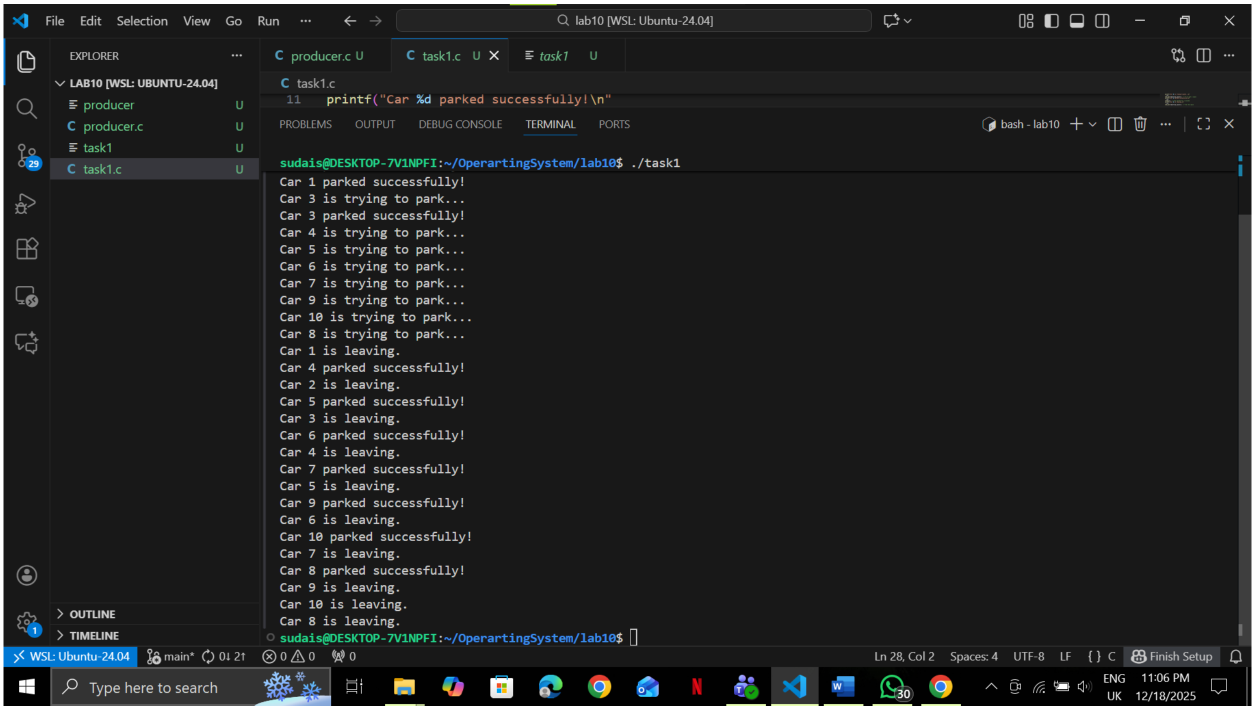
TASK 1:

```
#include <stdio.h>

#include <pthread.h>
```

```c
#include <semaphore.h>
#include <unistd.h>
sem_t parking_spaces;
void* car(void* arg) {
int id = *(int*)arg;
printf("Car %d is trying to park...\n"
, id);
sem_wait(&parking_spaces); // Try to get a space
printf("Car %d parked successfully!\n"
, id);
sleep(2); // Stay parked for 2 seconds
printf("Car %d is leaving.\n"
, id);
sem_post(&parking_spaces); // Free the space
return NULL;
}
int main() {
pthread_t cars[10];
int ids[10];
// Initialize: 3 parking spaces available
sem_init(&parking_spaces, 0, 3);
// Create 10 cars (more than spaces!)
for(int i = 0; i < 10; i++) {
ids[i] = i + 1;
pthread_create(&cars[i], NULL, car, &ids[i]);
}
// Wait for all cars
for(int i = 0; i < 10; i++) {
pthread_join(cars[i], NULL);
}
sem_destroy(&parking_spaces);
return 0;
}
```

- **OUTPUT:**



```c
printf("Car %d parked successfully!\n"
```

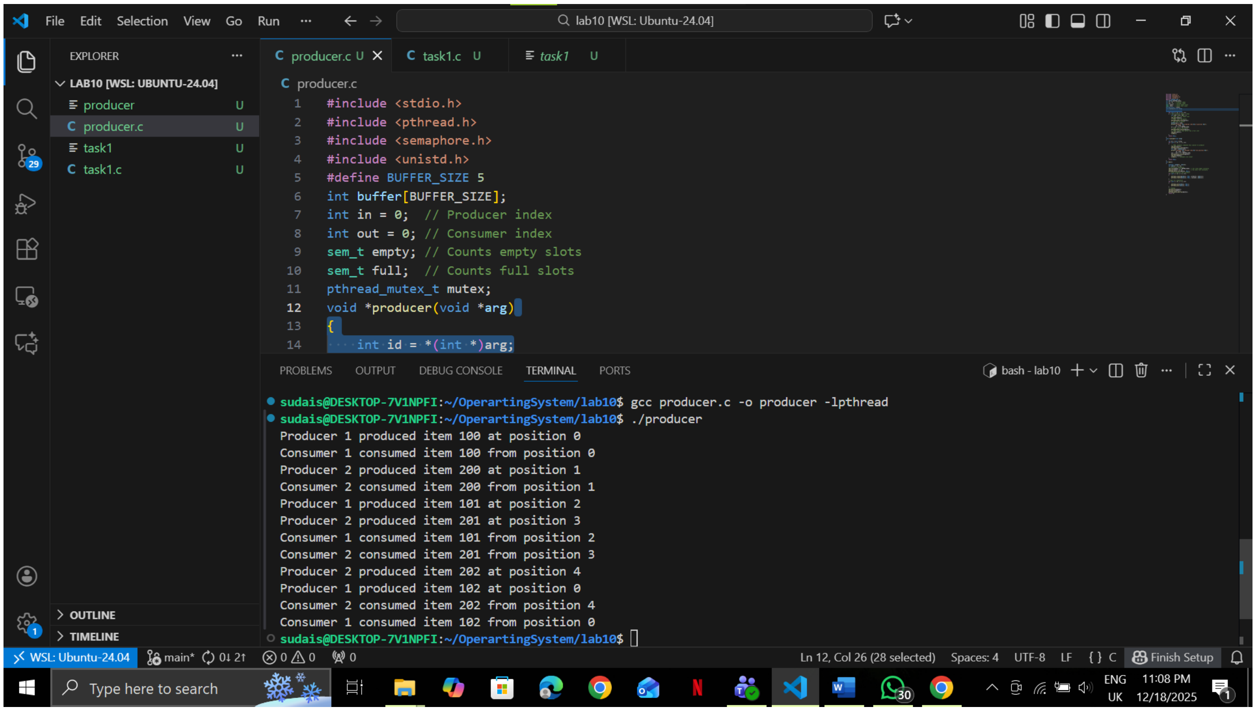
```
sudais@DESKTOP-7V1NPFI:~/OperartingSystem/lab10$ ./task1
Car 1 parked successfully!
Car 3 is trying to park...
Car 3 parked successfully!
Car 4 is trying to park...
Car 5 is trying to park...
Car 6 is trying to park...
Car 7 is trying to park...
Car 9 is trying to park...
Car 10 is trying to park...
Car 8 is trying to park...
Car 1 is leaving.
Car 4 parked successfully!
Car 2 is leaving.
Car 5 parked successfully!
Car 3 is leaving.
Car 6 parked successfully!
Car 4 is leaving.
Car 7 parked successfully!
Car 5 is leaving.
Car 9 parked successfully!
Car 6 is leaving.
Car 10 parked successfully!
Car 7 is leaving.
Car 8 parked successfully!
Car 9 is leaving.
Car 10 is leaving.
Car 8 is leaving.
sudais@DESKTOP-7V1NPFI:~/OperartingSystem/lab10$
```

TASK 2:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0;  // Producer index
int out = 0; // Consumer index
sem_t empty; // Counts empty slots
sem_t full;  // Counts full slots
pthread_mutex_t mutex;
void *producer(void *arg)
{
    int id = *(int *)arg;
    for (int i = 0; i < 3; i++)
    { // Each producer makes 3 items
        int item = id * 100 + i;
        // TODO: Wait for empty slot
        sem_wait(&empty);
        // TODO: Lock the buffer
        pthread_mutex_lock(&mutex);
        // Add item to buffer
        buffer[in] = item;
        printf("Producer %d produced item %d at position %d\n",
               id, item, in);
        in = (in + 1) % BUFFER_SIZE;
        // TODO: Unlock the buffer
        pthread_mutex_unlock(&mutex);
        // TODO: Signal that buffer has a full slot
        sem_post(&full);
        sleep(1);
    }
    return NULL;
}
void *consumer(void *arg)
{
    int id = *(int *)arg;
    for (int i = 0; i < 3; i++)
    {
        // TODO: Students complete this similar to producer
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %d consumed item %d from position %d\n",
               id, item, out);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(2); // Consumers are slower
    }
    return NULL;
}
int main()
{
    pthread_t prod[2], cons[2];
    int ids[2] = {1, 2};
    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE); // All slots empty initially
    sem_init(&full, 0, 0);            // No slots full initially
    pthread_mutex_init(&mutex, NULL);
    // Create producers and consumers
    for (int i = 0; i < 2; i++)
    {
        pthread_create(&prod[i], NULL, producer, &ids[i]);
        pthread_create(&cons[i], NULL, consumer, &ids[i]);
    }
    // Wait for completion
    for (int i = 0; i < 2; i++)
```

```
        {
            pthread_join(prod[i], NULL);
            pthread_join(cons[i], NULL);
        }
        // Cleanup
        sem_destroy(&empty);
        sem_destroy(&full);
        pthread_mutex_destroy(&mutex);
        return 0;
    }
```

- **OUTPUT:**



File Edit Selection View Go Run ···
lab10 [WSL: Ubuntu-24.04]
EXPLORER
LAB10 [WSL: UBUNTU-24.04]
producer U
producer.c U
task1 U
task1.c U
OUTLINE
TIMELINE
producer.c U
task1.c U
task1 U
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash - lab10
WSL: Ubuntu-24.04 main* 0↓ 2↑ 0 0 0
Ln 12, Col 26 (28 selected) Spaces: 4 UTF-8 LF { } C Finish Setup
Type here to search
ENG UK 11:08 PM 12/18/2025

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0;  // Producer index
int out = 0; // Consumer index
sem_t empty; // Counts empty slots
sem_t full;  // Counts full slots
pthread_mutex_t mutex;
void *producer(void *arg)
{
    int id = *(int *)arg;
```

```
sudais@DESKTOP-7V1NPFI:~/OperartingSystem/lab10$ gcc producer.c -o producer -lpthread
sudais@DESKTOP-7V1NPFI:~/OperartingSystem/lab10$ ./producer
Producer 1 produced item 100 at position 0
Consumer 1 consumed item 100 from position 0
Producer 2 produced item 200 at position 1
Consumer 2 consumed item 200 from position 1
Producer 1 produced item 101 at position 2
Producer 2 produced item 201 at position 3
Consumer 1 consumed item 101 from position 2
Consumer 2 consumed item 201 from position 3
Producer 2 produced item 202 at position 4
Producer 1 produced item 102 at position 0
Consumer 2 consumed item 202 from position 4
Consumer 1 consumed item 102 from position 0
sudais@DESKTOP-7V1NPFI:~/OperartingSystem/lab10$ 
```

Remarks: This program correctly implements the Producer–Consumer problem using semaphores and a mutex to ensure synchronization. Semaphores manage buffer availability while the mutex prevents concurrent access, avoiding race conditions and ensuring safe data exchange between threads.

"What happens if producers are faster than consumers?" If producers are faster than consumers, the buffer fills up and producers are blocked until consumers consume items.

"What happens if the belt is empty and a consumer tries to take an item? "If the buffer is empty, consumers are blocked until a producer produces an item.

"What synchronization do we need?" Semaphores and a mutex are needed to synchronize access and prevent race conditions.