

Task 1

Code

```
● ● ●

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 sem_t parking_spaces;
6 void* car(void* arg) {
7     int id = *(int*)arg;
8     printf("Car %d is trying to park...\n", id);
9     sem_wait(&parking_spaces); // Try to get a space
10    printf("Car %d parked successfully!\n", id);
11    sleep(2); // Stay parked for 2 seconds
12    printf("Car %d is leaving.\n", id);
13    sem_post(&parking_spaces); // Free the space
14    return NULL;
15 }
16 int main() {
17     pthread_t cars[10];
18     int ids[10];
19     // Initialize: 3 parking spaces available
20     sem_init(&parking_spaces, 0, 3);
21     // Create 10 cars (more than spaces!)
22     for(int i = 0; i < 10; i++) {
23         ids[i] = i + 1;
24         pthread_create(&cars[i], NULL, car, &ids[i]);
25     }
26     // Wait for all cars
27     for(int i = 0; i < 10; i++) {
28         pthread_join(cars[i], NULL);
29     }
30     sem_destroy(&parking_spaces);
31     return 0;
32 }
```

Output

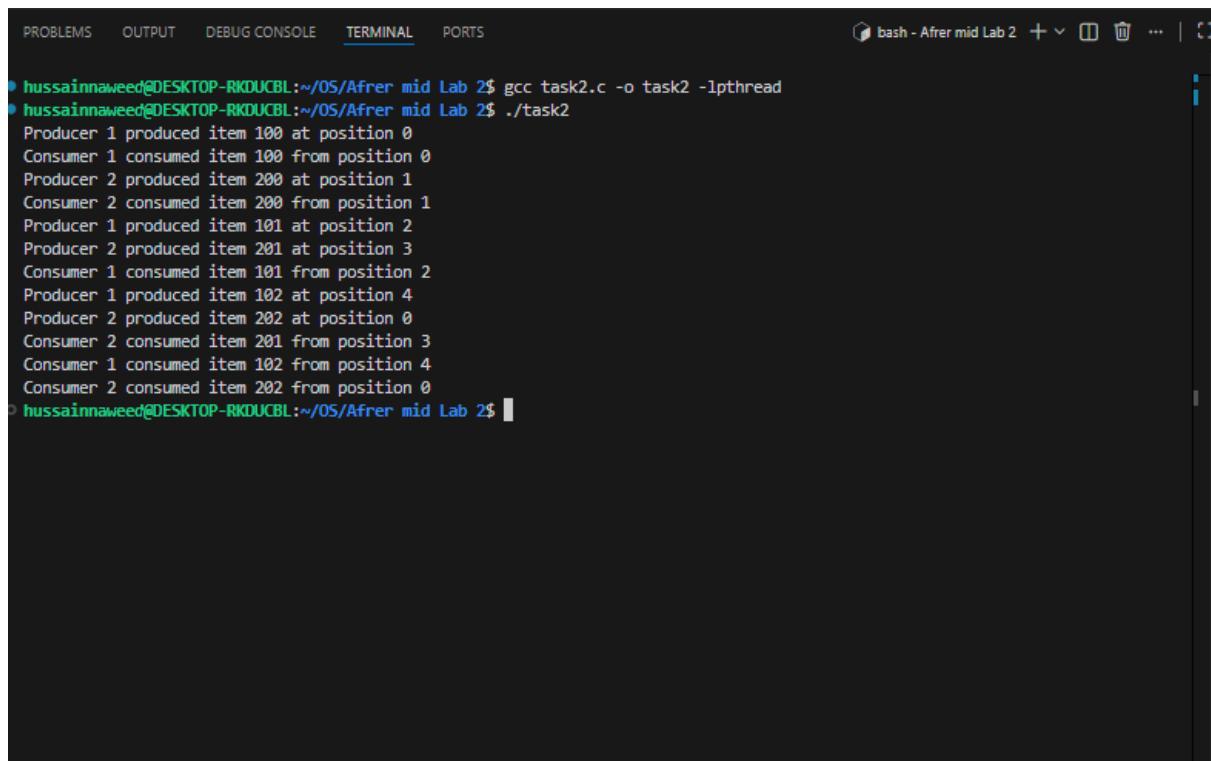
A screenshot of a terminal window titled "bash - After mid Lab 2". The window shows the output of a program named "task1". The output consists of multiple lines of text, each representing a car's attempt to park or leave. The text is color-coded: green for parking attempts, blue for successful parks, and red for leaving. The terminal interface includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. The status bar at the bottom right shows the date and time as "Mon Jun 24".

```
hussainnaweed@DESKTOP-RKOUCLB:~/OS/Afrer mid Lab 2$ ./task1
Car 3 is trying to park...
Car 3 parked successfully!
Car 4 is trying to park...
Car 4 parked successfully!
Car 1 parked successfully!
Car 2 is trying to park...
Car 5 is trying to park...
Car 7 is trying to park...
Car 6 is trying to park...
Car 8 is trying to park...
Car 9 is trying to park...
Car 10 is trying to park...
Car 3 is leaving.
Car 4 is leaving.
Car 2 parked successfully!
Car 1 is leaving.
Car 7 parked successfully!
Car 5 parked successfully!
Car 2 is leaving.
Car 5 is leaving.
Car 7 is leaving.
Car 8 parked successfully!
Car 9 parked successfully!
Car 6 parked successfully!
Car 8 is leaving.
Car 6 is leaving.
Car 10 parked successfully!
Car 9 is leaving.
Car 10 is leaving.
hussainnaweed@DESKTOP-RKOUCLB:~/OS/Afrer mid Lab 2$
```

Task2

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 #define BUFFER_SIZE 5
6 int buffer[BUFFER_SIZE];
7 int in = 0; // Producer index
8 int out = 0; // Consumer index
9 sem_t empty; // Counts empty slots
10 sem_t full; // Counts full slots
11 pthread_mutex_t mutex;
12 void* producer(void* arg) {
13     int id = *(int*)arg;
14     for(int i = 0; i < 3; i++) { // Each producer makes 3 items
15         int item = id * 100 + i;
16         // TODO: Wait for empty slot
17         sem_wait(&empty);
18         // TODO: Lock the buffer
19         pthread_mutex_lock(&mutex);
20         // Add item to buffer
21         buffer[in] = item;
22         printf("Producer %d produced item %d at position %d\n",
23                id, item, in);
24         in = (in + 1) % BUFFER_SIZE;
25         // TODO: Unlock the buffer
26         pthread_mutex_unlock(&mutex);
27         // TODO: Signal that buffer has a full slot
28         sem_post(&full);
29         sleep(1);
30     }
31     return NULL;
32 }
33 void* consumer(void* arg) {
34     int id = *(int*)arg;
35     for(int i = 0; i < 3; i++) {
36         // TODO: Students complete this similar to producer
37         sem_wait(&full);
38         pthread_mutex_lock(&mutex);
39         int item = buffer[out];
40         printf("Consumer %d consumed item %d from position %d\n",
41                id, item, out);
42         out = (out + 1) % BUFFER_SIZE;
43         pthread_mutex_unlock(&mutex);
44         sem_post(&empty);
45         sleep(2); // Consumers are slower
46     }
47     return NULL;
48 }
49 int main() {
50     pthread_t prod[2], cons[2];
51     int ids[2] = {1, 2};
52     // Initialize semaphores
53     sem_init(&empty, 0, BUFFER_SIZE); // All slots empty initially
54     sem_init(&full, 0, 0);
55     pthread_mutex_init(&mutex, NULL);
56     // No slots full initially
57     // Create producers and consumers
58     for(int i = 0; i < 2; i++) {
59         pthread_create(&prod[i], NULL, producer, &ids[i]);
60         pthread_create(&cons[i], NULL, consumer, &ids[i]);
61     }
62     // Wait for completion
63     for(int i = 0; i < 2; i++) {
64         pthread_join(prod[i], NULL);
65         pthread_join(cons[i], NULL);
66     }
67     // Cleanup
68     sem_destroy(&empty);
69     sem_destroy(&full);
70     pthread_mutex_destroy(&mutex);
71     return 0;
72 }
```

Output



The screenshot shows a terminal window with the following output:

```
● hussainnaweed@DESKTOP-RKOUCLB:~/OS/Afrer mid Lab 2$ gcc task2.c -o task2 -lpthread
● hussainnaweed@DESKTOP-RKOUCLB:~/OS/Afrer mid Lab 2$ ./task2
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
Producer 1 produced item 102 at position 4
Producer 2 produced item 202 at position 0
Consumer 2 consumed item 201 from position 3
Consumer 1 consumed item 102 from position 4
Consumer 2 consumed item 202 from position 0
● hussainnaweed@DESKTOP-RKOUCLB:~/OS/Afrer mid Lab 2$
```

Demonstration

- 1. This program uses two semaphores along with one mutex.**

- 2. The semaphores manage the producer and consumer operations, while the mutex handles locking and unlocking the shared buffer.**

3. The semaphores are named full and empty— empty keeps track of available slots, and full tracks how many slots are already filled.

4. `sem_wait()` makes a thread pause until an empty slot becomes available.

5. `sem_post()` signals that a slot has been occupied or freed.

6. All threads are allowed to finish their work, and afterward, the semaphores and mutex are properly destroyed.

7. The output values may look like this:

- **$\text{id} * 100 + i$ where $i = 0, 1, 2$ and $\text{id} = 1, 2$**
- **101**
- **102**
- **103**

- 201

- 202

- 203

<u>CONSUMER</u>	<u>PRODUCER</u>
1. wait (full) 2. mutex locked 3. critical section 4. unlock. 5. wait(empty)	1. wait (empty) 2. mutex locked. 3. critical section 4. unlock. 5. wait(full)