

Assignment No 2:

Name: Samruddhi Devram Khilari

Branch & Year: AIML-B SY

Prn: 12420020

Roll no: 2

Sub: Operating System Lab

1. Q] Solve Dinning philosopher problem using mutex & semaphore.

The screenshot shows a code editor window with the file name "DINNNING_PHILLOSOPHER.c" and the path "~/Desktop/OS". The code is written in C and defines a main function that initializes philosophers, creates threads, and joins them. It uses mutexes to simulate the eating process.

```
27     // Eating
28     printf("Philosopher %d is eating.\n", id);
29     sleep(rand() % 2); // Simulate eating time
30     // Put down right fork
31     pthread_mutex_unlock(&forks[(id + 1) % NUM_PHILOSOPHERS]);
32     printf("Philosopher %d put down right fork.\n", id);
33     // Put down left fork
34     pthread_mutex_unlock(&forks[id]);
35     printf("Philosopher %d put down left fork.\n", id);
36 }
37 }
38 int main() {
39     pthread_t philosophers[NUM_PHILOSOPHERS];
40     int philosopherIds[NUM_PHILOSOPHERS];
41     // Initialize mutexes for each fork
42     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
43         pthread_mutex_init(&forks[i], NULL);
44     }
45     // Create philosopher threads
46     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
47         philosopherIds[i] = i;
48         pthread_create(&philosophers[i], NULL, philosopher, &philosopherIds[i]);
49     }
50     // Join threads (in a real program you might not do this)
51     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
52         pthread_join(philosophers[i], NULL);
53     }
54     // Clean up
55     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
56         pthread_mutex_destroy(&forks[i]);
57     }
58     return 0;
59 }
60
```

The screenshot shows a code editor window with the file name "DINNNING_PHILLOSOPHER.c" and the path "~/Desktop/OS". The code continues from the previous snippet, defining a philosopher function that simulates thinking, picking up forks, and eating.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <unistd.h>
6
7 #define NUM_PHILOSOPHERS 5
8
9 pthread_mutex_t forks[NUM_PHILOSOPHERS]; // Mutexes for each fork
10
11 // Function to simulate the eating process
12 void* philosopher(void* num) {
13     int id = *(int*)num;
14
15     while (1) {
16         printf("Philosopher %d is thinking.\n", id);
17         sleep(rand() % 2); // Simulate thinking time
18
19         // Pick up left fork
20         pthread_mutex_lock(&forks[id]);
21         printf("Philosopher %d picked up left fork.\n", id);
22
23         // Pick up right fork
24         pthread_mutex_lock(&forks[(id + 1) % NUM_PHILOSOPHERS]);
25         printf("Philosopher %d picked up right fork.\n", id);
26
27         // Eating
28         printf("Philosopher %d is eating.\n", id);
29         sleep(rand() % 2); // Simulate eating time
30
31         // Put down right fork
32         pthread_mutex_unlock(&forks[(id + 1) % NUM_PHILOSOPHERS]);
33         printf("Philosopher %d put down right fork.\n", id);
34 }
```

Outputs:

```
vbox@ubuntu:~/Desktop/OS$ gcc DINNNING_PHILLOSOPHER.c
vbox@ubuntu:~/Desktop/OS$ a./.out
bash: a./.out: No such file or directory
vbox@ubuntu:~/Desktop/OS$ ./a.out
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 0 is thinking.
Philosopher 3 picked up left fork.
Philosopher 3 picked up right fork.
Philosopher 3 is eating.
Philosopher 2 picked up left fork.
Philosopher 1 picked up left fork.
Philosopher 0 picked up left fork.
Philosopher 3 put down right fork.
Philosopher 3 put down left fork.
Philosopher 3 is thinking.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 4 picked up left fork.
Philosopher 2 put down right fork.
Philosopher 2 put down left fork.
Philosopher 2 is thinking.
Philosopher 1 picked up right fork.
Philosopher 1 is eating.
Philosopher 3 picked up left fork.
Philosopher 1 put down right fork.
Philosopher 1 put down left fork.
Philosopher 1 is thinking.
Philosopher 2 picked up left fork.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 0 put down right fork.
Philosopher 0 put down left fork.
Philosopher 0 is thinking.
```

```
vbox@ubuntu:~/Desktop/OS$ ./a.out
Philosopher 3 put down left fork.
Philosopher 3 is thinking.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 4 picked up left fork.
Philosopher 4 picked up right fork.
Philosopher 4 is eating.
Philosopher 4 put down right fork.
Philosopher 4 put down left fork.
Philosopher 4 is thinking.
Philosopher 2 put down right fork.
Philosopher 2 put down left fork.
Philosopher 2 is thinking.
Philosopher 3 picked up left fork.
Philosopher 3 picked up right fork.
Philosopher 3 is eating.
Philosopher 1 picked up left fork.
Philosopher 1 picked up right fork.
Philosopher 1 is eating.
Philosopher 3 put down right fork.
Philosopher 3 put down left fork.
Philosopher 3 is thinking.
Philosopher 0 picked up left fork.
Philosopher 2 picked up left fork.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 1 put down right fork.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 1 put down left fork.
Philosopher 1 is thinking.
Philosopher 0 put down right fork.
Philosopher 0 put down left fork.
Philosopher 0 is thinking.
```

2. Q] Solve Producer consumer problem using mutex & semaphore.

```
Open ▾  [+]
PRODUCER_CONSUMER.c
~/Desktop/OS

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <unistd.h>
6
7 #define BUFFER_SIZE 5 // Size of the buffer
8 #define NUM_PRODUCERS 2
9 #define NUM_CONSUMERS 2
10
11 int buffer[BUFFER_SIZE]; // Shared buffer
12 int in = 0; // Index for the next item to produce
13 int out = 0; // Index for the next item to consume
14
15 sem_t empty; // Semaphore to count empty slots
16 sem_t full; // Semaphore to count full slots
17 pthread_mutex_t mutex; // Mutex for critical section
18
19 // Producer function
20 void *producer(void *arg) {
21     int id = *(int *)arg;
22     while (1) {
23         // Produce an item
24         int item = rand() % 100; // Random item
25         sem_wait(&empty); // Wait for an empty slot
26         pthread_mutex_lock(&mutex); // Lock the critical section
27
28         // Add the item to the buffer
29         buffer[in] = item;
30         printf("Producer %d: Produced %d at index %d\n", id, item, in);
31         in = (in + 1) % BUFFER_SIZE; // Update the index
32
33         pthread_mutex_unlock(&mutex); // Unlock the critical section
34         sem_post(&full); // Signal that a new item has been produced

```

```
Open ▾  [+]
*PRODUCER_CONSUMER.c
~/Desktop/OS

50         sleep(1); // Simulate time taken to consume
51     }
52 }
53 int main() {
54     pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
55     int producerIds[NUM_PRODUCERS], consumerIds[NUM_CONSUMERS];
56     // Initialize semaphores and mutex
57     sem_init(&empty, 0, BUFFER_SIZE); // Initially, all slots are empty
58     sem_init(&full, 0, 0); // Initially, no slots are full
59     pthread_mutex_init(&mutex, NULL);
60     // Create producer threads
61     for (int i = 0; i < NUM_PRODUCERS; i++) {
62         producerIds[i] = i + 1;
63         pthread_create(&producers[i], NULL, producer, &producerIds[i]);
64     }
65     // Create consumer threads
66     for (int i = 0; i < NUM_CONSUMERS; i++) {
67         consumerIds[i] = i + 1;
68         pthread_create(&consumers[i], NULL, consumer, &consumerIds[i]);
69     }
70     // Join threads (in a real program you might not do this)
71     for (int i = 0; i < NUM_PRODUCERS; i++) {
72         pthread_join(producers[i], NULL);
73     }
74     for (int i = 0; i < NUM_CONSUMERS; i++) {
75         pthread_join(consumers[i], NULL);
76     }
77     // Clean up
78     sem_destroy(&empty);
79     sem_destroy(&full);
80     pthread_mutex_destroy(&mutex);
81     return 0;
82 }
83
```

Outputs:

```
vbox@ubuntu: ~/Desktop/OS
```

```
Producer 1: Produced 35 at index 0
Consumer 2: Consumed 35 from index 0
Producer 2: Produced 86 at index 1
Consumer 1: Consumed 86 from index 1
Producer 1: Produced 92 at index 2
Consumer 2: Consumed 92 from index 2
Producer 2: Produced 49 at index 3
Consumer 1: Consumed 49 from index 3
Producer 1: Produced 21 at index 4
Consumer 2: Consumed 21 from index 4
Producer 2: Produced 62 at index 0
Consumer 1: Consumed 62 from index 0
Producer 1: Produced 27 at index 1
Consumer 2: Consumed 27 from index 1
Producer 2: Produced 90 at index 2
Consumer 1: Consumed 90 from index 2
Producer 1: Produced 59 at index 3
Consumer 2: Consumed 59 from index 3
Producer 2: Produced 63 at index 4
Consumer 1: Consumed 63 from index 4
Producer 1: Produced 26 at index 0
Consumer 2: Consumed 26 from index 0
Producer 2: Produced 40 at index 1
Consumer 1: Consumed 40 from index 1
Producer 1: Produced 26 at index 2
Consumer 2: Consumed 26 from index 2
Producer 1: Produced 72 at index 3
Consumer 1: Consumed 72 from index 3
Producer 2: Produced 36 at index 4
Consumer 2: Consumed 36 from index 4
Producer 1: Produced 11 at index 0
Consumer 1: Consumed 11 from index 0
Producer 2: Produced 68 at index 1
Consumer 2: Consumed 68 from index 1
[]
```

```
vt
```

```
Producer 2: Produced 33 at index 1
Consumer 2: Consumed 33 from index 1
Producer 1: Produced 33 at index 2
Consumer 1: Consumed 33 from index 2
Producer 2: Produced 48 at index 3
Consumer 2: Consumed 48 from index 3
Producer 1: Produced 90 at index 4
Consumer 1: Consumed 90 from index 4
Producer 2: Produced 54 at index 0
Consumer 2: Consumed 54 from index 0
Producer 2: Produced 67 at index 1
Producer 1: Produced 46 at index 2
Consumer 1: Consumed 67 from index 1
Consumer 2: Consumed 46 from index 2
Producer 2: Produced 68 at index 3
Producer 1: Produced 29 at index 4
Consumer 1: Consumed 68 from index 3
Consumer 2: Consumed 29 from index 4
Producer 2: Produced 0 at index 0
Producer 1: Produced 46 at index 1
Consumer 1: Consumed 0 from index 0
Consumer 2: Consumed 46 from index 1
Producer 2: Produced 88 at index 2
Producer 1: Produced 97 at index 3
Consumer 1: Consumed 88 from index 2
Consumer 2: Consumed 97 from index 3
Producer 1: Produced 49 at index 4
Consumer 1: Consumed 49 from index 4
Producer 2: Produced 90 at index 0
Consumer 2: Consumed 90 from index 0
Producer 1: Produced 3 at index 1
Producer 2: Produced 33 at index 2
Consumer 1: Consumed 3 from index 1
Consumer 2: Consumed 33 from index 2
[]
```

3. Q] Solve Reader writer problem using mutex & semaphore.

Open READER_WRITER.c
~/Desktop/OS

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <unistd.h>
6
7 #define NUM_READERS 5
8 #define NUM_WRITERS 2
9
10 sem_t readSemaphore;
11 pthread_mutex_t writeMutex;
12 int readCount = 0; // Count of current readers
13 int sharedResource = 0; // Shared resource
14
15 // Reader function
16 void *reader(void *arg) {
17     int id = *(int *)arg;
18     while (1) {
19         // Start reading
20         sem_wait(&readSemaphore);
21         pthread_mutex_lock(&writeMutex);
22         readCount++;
23         if (readCount == 1) {
24             // First reader locks the resource
25             sem_wait(&writeMutex);
26         }
27         pthread_mutex_unlock(&writeMutex);
28         sem_post(&readSemaphore);
29         // Reading
30         printf("Reader %d: Read value %d\n", id, sharedResource);
31         sleep(1); // Simulate reading time
32
33         // Done reading
34         pthread_mutex_lock(&writeMutex);

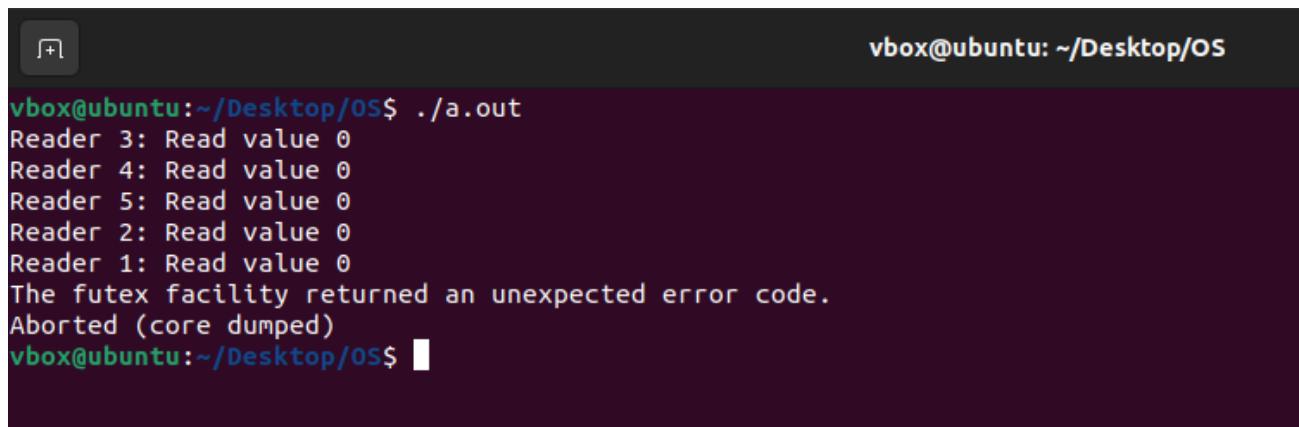
```

Open READER_WRITER.c
~/Desktop/OS

```

56         sem_post(&writeMutex);
57         sleep(1); // Simulate idle time before next write
58     }
59 }
60 int main() {
61     pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
62     int readerIds[NUM_READERS], writerIds[NUM_WRITERS];
63     // Initialize semaphores and mutex
64     sem_init(&readSemaphore, 0, 1);
65     sem_init(&writeMutex, 0, 1);
66     pthread_mutex_init(&writeMutex, NULL);
67     // Create reader threads
68     for (int i = 0; i < NUM_READERS; i++) {
69         readerIds[i] = i + 1;
70         pthread_create(&readers[i], NULL, reader, &readerIds[i]);
71     }
72     // Create writer threads
73     for (int i = 0; i < NUM_WRITERS; i++) {
74         writerIds[i] = i + 1;
75         pthread_create(&writers[i], NULL, writer, &writerIds[i]);
76     }
77     // Join threads (in a real program you might not do this)
78     for (int i = 0; i < NUM_READERS; i++) {
79         pthread_join(readers[i], NULL);
80     }
81     for (int i = 0; i < NUM_WRITERS; i++) {
82         pthread_join(writers[i], NULL);
83     }
84     // Clean up
85     sem_destroy(&readSemaphore);
86     sem_destroy(&writeMutex);
87     pthread_mutex_destroy(&writeMutex);
88     return 0;
89 }
```

Outputs:



The screenshot shows a terminal window with a dark background and light-colored text. In the top right corner, it says "vbox@ubuntu: ~/Desktop/OS". On the far left, there is a small icon with a plus sign. The terminal output is as follows:

```
vbox@ubuntu:~/Desktop/OS$ ./a.out
Reader 3: Read value 0
Reader 4: Read value 0
Reader 5: Read value 0
Reader 2: Read value 0
Reader 1: Read value 0
The futex facility returned an unexpected error code.
Aborted (core dumped)
vbox@ubuntu:~/Desktop/OS$
```