

LAB :- 3

DATE:- 03-11-2022

Process Synchronisation-IPC

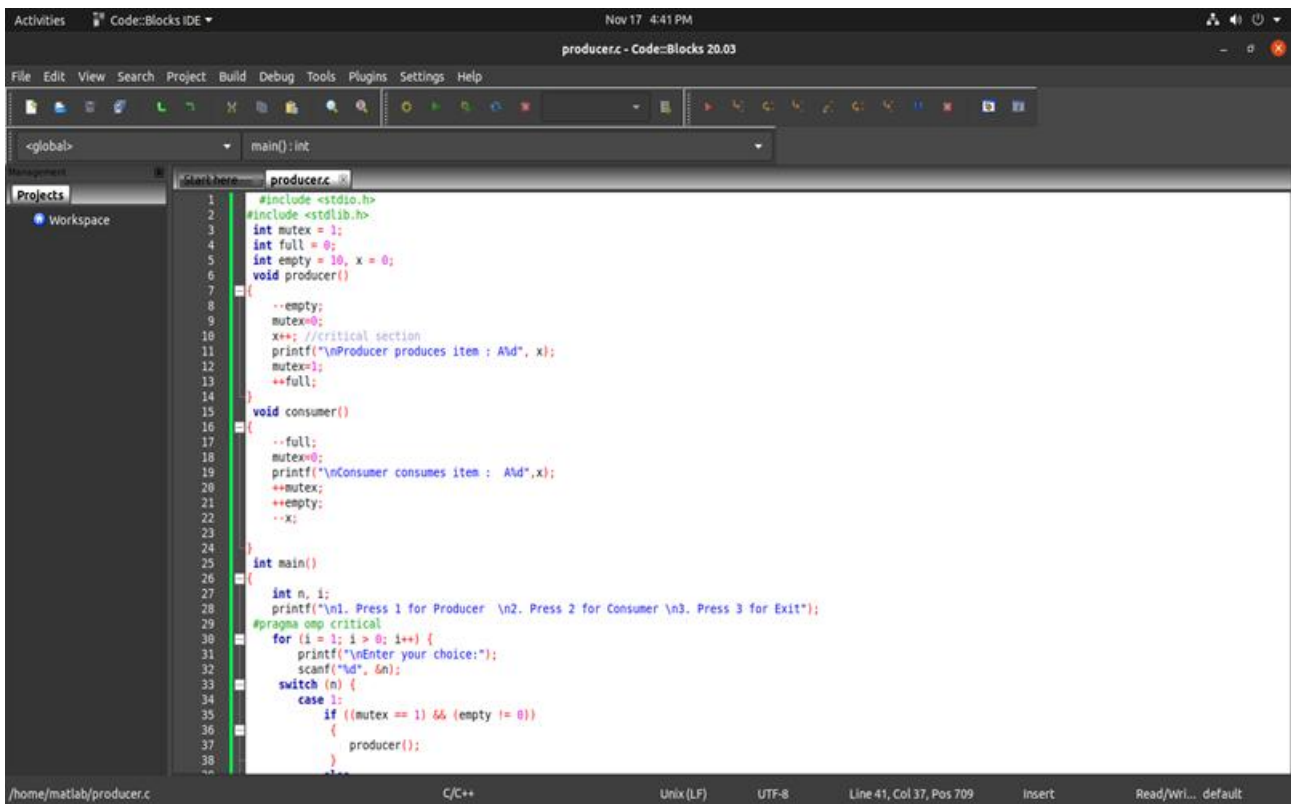
- 1. Implement the deadlock-free solution to producer consumer problem using Semaphore.**
- 2. Implement the deadlock-free solution to Reader writer problem using Semaphore.**
- 3. Implement the deadlock-free solution to Dining philosopher problem using Semaphore.**
- 4. Implement the deadlock-free solution to Reader writer problem using monitors.**

1. Implement the deadlock-free solution to producer consumer problem using Semaphore.

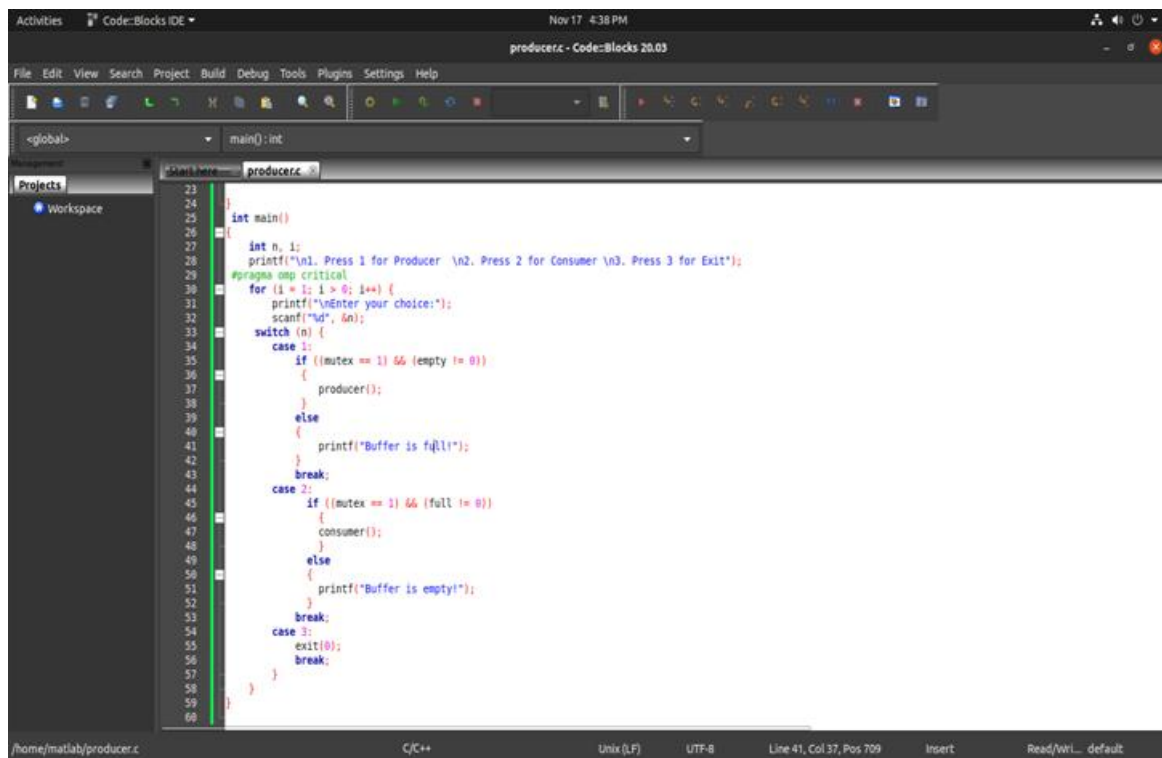
Code:-

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10, x = 0;
void producer()
{
    --empty;
    mutex=0;
    x++; //critical section
    printf("\nProducer produces item : A%d", x);
    mutex=1;
    ++full;
}
void consumer()
{
    --full;
    mutex=0;
    printf("\nConsumer consumes item : A%d",x);
    ++mutex;
    ++empty;
    --x;
}
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3. Press 3 for Exit");
    #pragma omp critical
    for (i = 1; i > 0; i++) {
        printf("\nEnter your choice:");
        scanf("%d", &n);
        switch (n) {
            case 1:
                if ((mutex == 1) && (empty != 0))
                {
                    producer();
                }
                else
                {
                    printf("Buffer is full!");
                }
            }
        }
    }
```

```
        break;
    case 2:
        if ((mutex == 1) && (full != 0))
        {
            consumer();
        }
        else
        {
            printf("Buffer is empty!");
        }
        break;
    case 3:
        exit(0);
        break;
    }
}
```



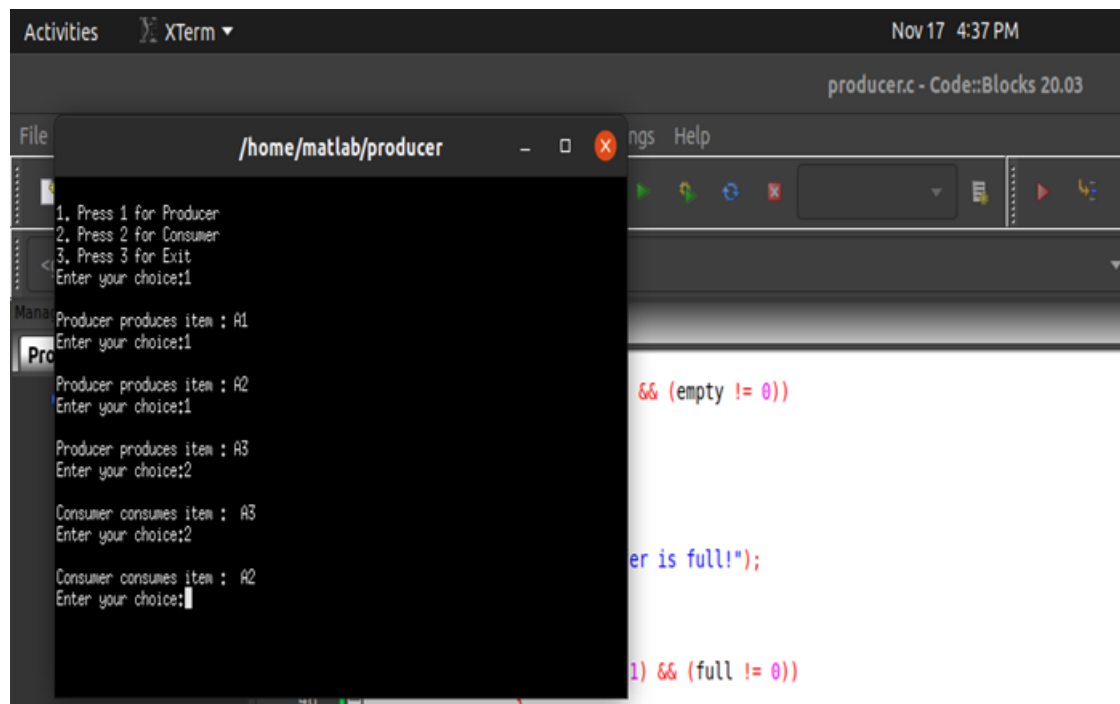
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int mutex = 1;
4  int full = 0;
5  int empty = 10, x = 0;
6  void producer()
7  {
8      //empty;
9      mutex=0;
10     x++; //critical section
11     printf("\nProducer produces item : %d", x);
12     mutex=1;
13     ++full;
14 }
15 void consumer()
16 {
17     //full;
18     mutex=0;
19     printf("\nConsumer consumes item : %d", x);
20     ++mutex;
21     ++empty;
22     --x;
23 }
24
25 int main()
26 {
27     int n, i;
28     printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3. Press 3 for Exit");
29     #pragma omp critical
30     for (i = 1; i > 0; i++) {
31         printf("\nEnter your choice:");
32         scanf("%d", &n);
33         switch (n) {
34             case 1:
35                 if ((mutex == 1) && (empty != 0))
36                 {
37                     producer();
38                 }
39             case 2:
40                 if ((mutex == 1) && (full != 0))
41                 {
42                     consumer();
43                 }
44             case 3:
45                 exit(0);
46                 break;
47             default:
48                 printf("Invalid choice\n");
49         }
50     }
```



The screenshot shows the Code-Blocks IDE interface. The main editor window displays a C program titled 'producer.c'. The code implements a producer-consumer problem using a mutex and a buffer. The program starts with a main function that initializes a counter 'i' to 1 and prints instructions for the user. It then enters a loop where it prompts the user to enter a choice (1 for Producer, 2 for Consumer, 3 for Exit). The user's choice is stored in 'n'. A switch statement handles the choices: Case 1 (Producer) checks if the mutex is available and the buffer is not full. If both conditions are met, it calls 'producer()' and increments 'i'. Case 2 (Consumer) checks if the mutex is available and the buffer is not empty. If both conditions are met, it calls 'consumer()' and decrements 'i'. Case 3 (Exit) calls 'exit(0)'. The program uses '#pragma omp critical' to ensure mutual exclusion. The status bar at the bottom indicates the file path as '/home/matlab/producer.c', the language as 'C/C++', and the encoding as 'UTF-8'.

```
23 }
24
25
26
27 int main()
28 {
29     int n, i;
30     printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3. Press 3 for Exit");
31     #pragma omp critical
32     for (i = 1; i > 0; i++) {
33         printf("\nEnter your choice:");
34         scanf("%d", &n);
35         switch (n) {
36             case 1:
37                 if ((mutex == 1) && (empty != 0))
38                 {
39                     producer();
40                 }
41                 else
42                 {
43                     printf("Buffer is full!");
44                 }
45                 break;
46             case 2:
47                 if ((mutex == 1) && (full != 0))
48                 {
49                     consumer();
50                 }
51                 else
52                 {
53                     printf("Buffer is empty!");
54                 }
55                 break;
56             case 3:
57                 exit(0);
58                 break;
59         }
60     }
61 }
```

Output:-



2. Implement the deadlock-free solution to Reader writer problem using Semaphore.

Code:-

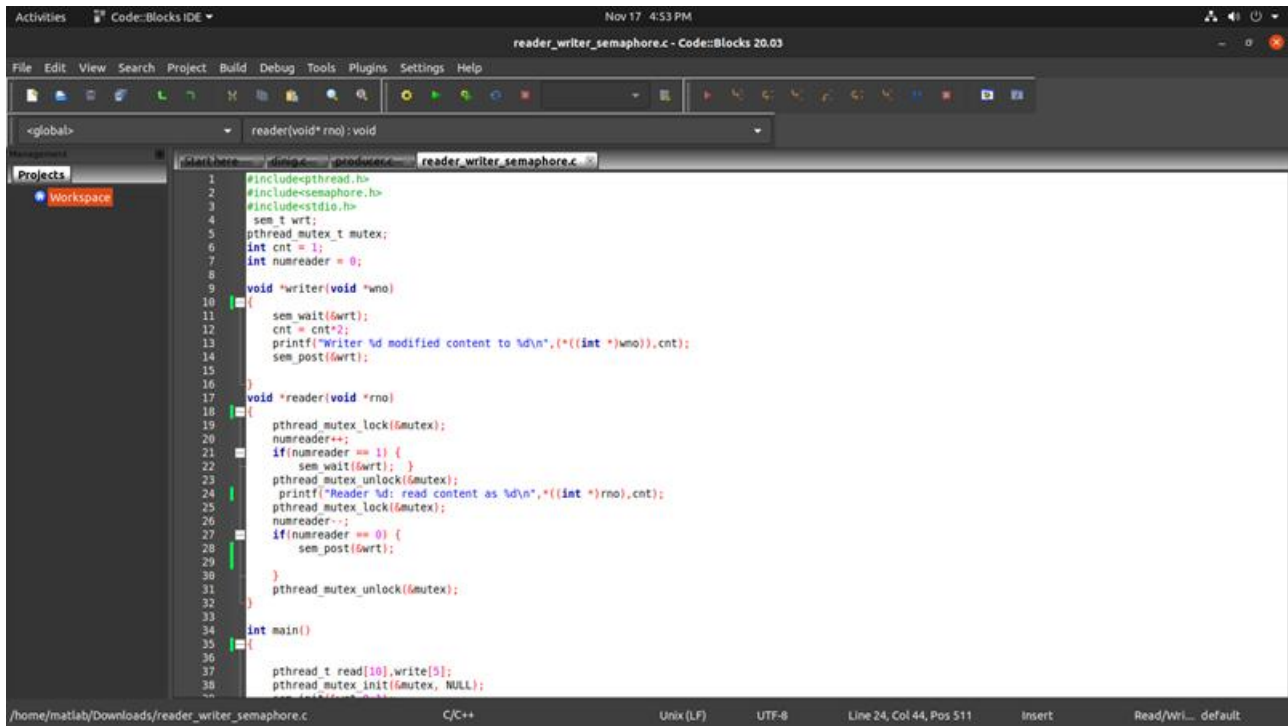
```
#include<pthread.h>
#include<semaphore.h>
#include<stdio.h>
sem_t wrt;
pthread_mutex_t mutex;
int cnt = 1;
int numreader = 0;

void *writer(void *wno)
{
    sem_wait(&wrt);
    cnt = cnt*2;
    printf("Writer %d modified content to %d\n",*((int *)wno),cnt);
    sem_post(&wrt);
}

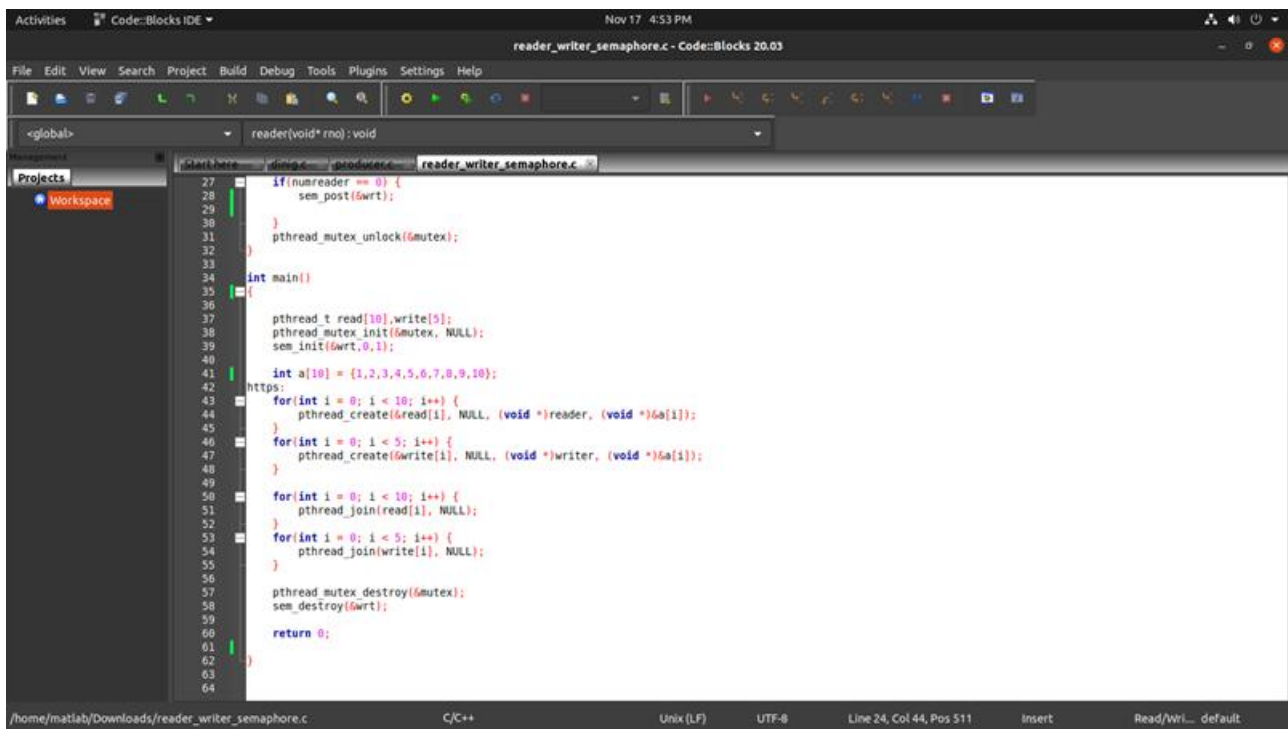
void *reader(void *rno)
{
    pthread_mutex_lock(&mutex);
    numreader++;
    if(numreader == 1) {
        sem_wait(&wrt); }
    pthread_mutex_unlock(&mutex);
    printf("Reader %d: read content as %d\n",*((int *)rno),cnt);
```

```
pthread_mutex_lock(&mutex);
numreader--;
if(numreader == 0) {
    sem_post(&wrt);
}
pthread_mutex_unlock(&mutex);
}
int main()
{
    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);
    int a[10] = {1,2,3,4,5,6,7,8,9,10};
https:
    for(int i = 0; i < 10; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
    }
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
    }
    for(int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

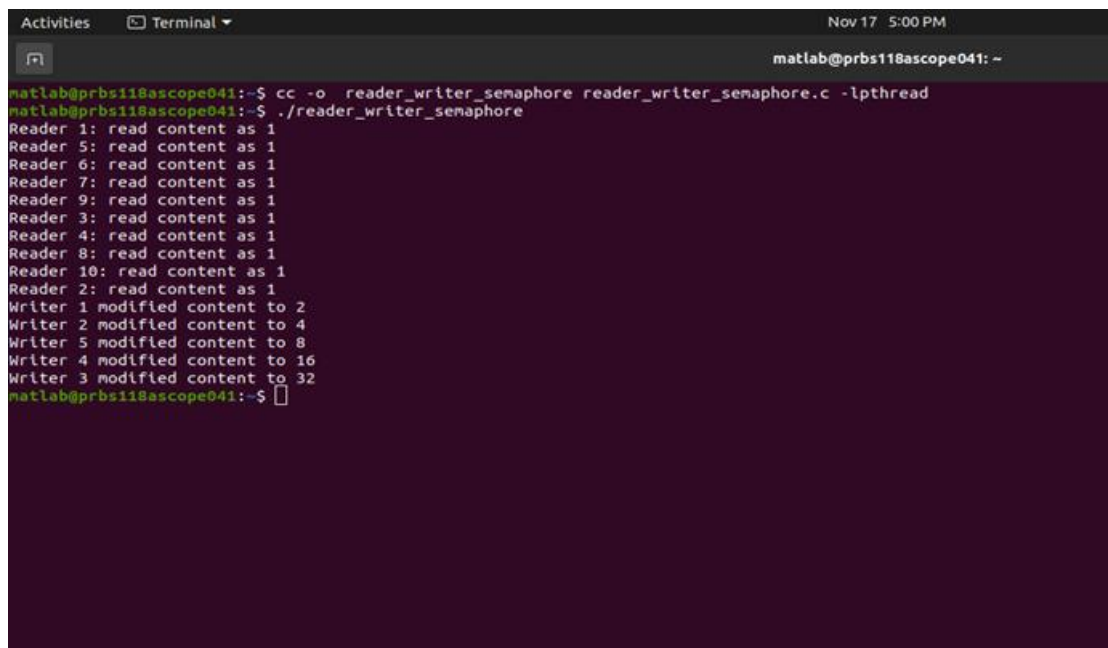


```
1 #include<pthread.h>
2 #include<semaphore.h>
3 #include<stdio.h>
4 sem_t wrt;
5 pthread_mutex_t mutex;
6 int cnt = 1;
7 int numreader = 0;
8
9 void *writer(void *wno)
10 {
11     sem_wait(&wrt);
12     cnt = cnt*2;
13     printf("Writer %d modified content to %d\n",*((int *)wno),cnt);
14     sem_post(&wrt);
15 }
16
17 void *reader(void *rno)
18 {
19     pthread_mutex_lock(&mutex);
20     numreader++;
21     if(numreader == 1) {
22         sem_wait(&wrt);
23     }
24     pthread_mutex_unlock(&mutex);
25     printf("Reader %d: read content as %d\n",*((int *)rno),cnt);
26     pthread_mutex_lock(&mutex);
27     numreader--;
28     if(numreader == 0) {
29         sem_post(&wrt);
30     }
31     pthread_mutex_unlock(&mutex);
32 }
33
34 int main()
35 {
36     pthread_t read[10],write[5];
37     pthread_mutex_init(&mutex, NULL);
38 }
```



```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4
5  pthread_mutex_t mutex;
6  pthread_t readers[10];
7  pthread_t writers[5];
8  int numreader = 0;
9  int numwriter = 0;
10 int a[10] = {1,2,3,4,5,6,7,8,9,10};
11
12 void reader(void* rno);
13 void writer(void* wno);
14
15 int main()
16 {
17     pthread_mutex_init(&mutex, NULL);
18     sem_t sem;
19     sem_init(&sem, 0, 1);
20
21     for(int i = 0; i < 10; i++) {
22         pthread_create(&readers[i], NULL, (void *)reader, (void *)i);
23     }
24     for(int i = 0; i < 5; i++) {
25         pthread_create(&writers[i], NULL, (void *)writer, (void *)i);
26     }
27
28     for(int i = 0; i < 10; i++) {
29         pthread_join(readers[i], NULL);
30     }
31     for(int i = 0; i < 5; i++) {
32         pthread_join(writers[i], NULL);
33     }
34
35     pthread_mutex_destroy(&mutex);
36     sem_destroy(&sem);
37
38     return 0;
39 }
```

Output:-



```
matlab@prbs118ascope041:~$ cc -o reader_writer_semaphore reader_writer_semaphore.c -lpthread
matlab@prbs118ascope041:~$ ./reader_writer_semaphore
Reader 1: read content as 1
Reader 5: read content as 1
Reader 6: read content as 1
Reader 7: read content as 1
Reader 9: read content as 1
Reader 3: read content as 1
Reader 4: read content as 1
Reader 8: read content as 1
Reader 10: read content as 1
Reader 2: read content as 1
Writer 1 modified content to 2
Writer 2 modified content to 4
Writer 5 modified content to 8
Writer 4 modified content to 16
Writer 3 modified content to 32
matlab@prbs118ascope041:~$
```


3.Implement the deadlock-free solution to Dining philosopher problem using Semaphore.

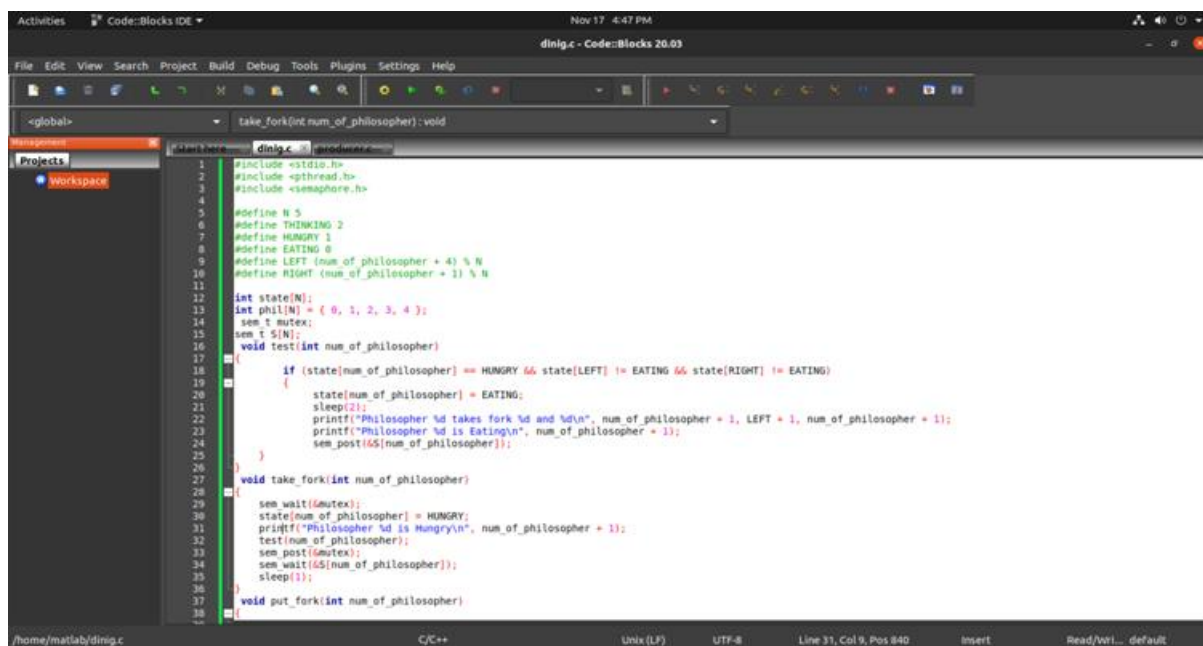
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (num_of_philosopher + 4) % N
#define RIGHT (num_of_philosopher + 1) % N

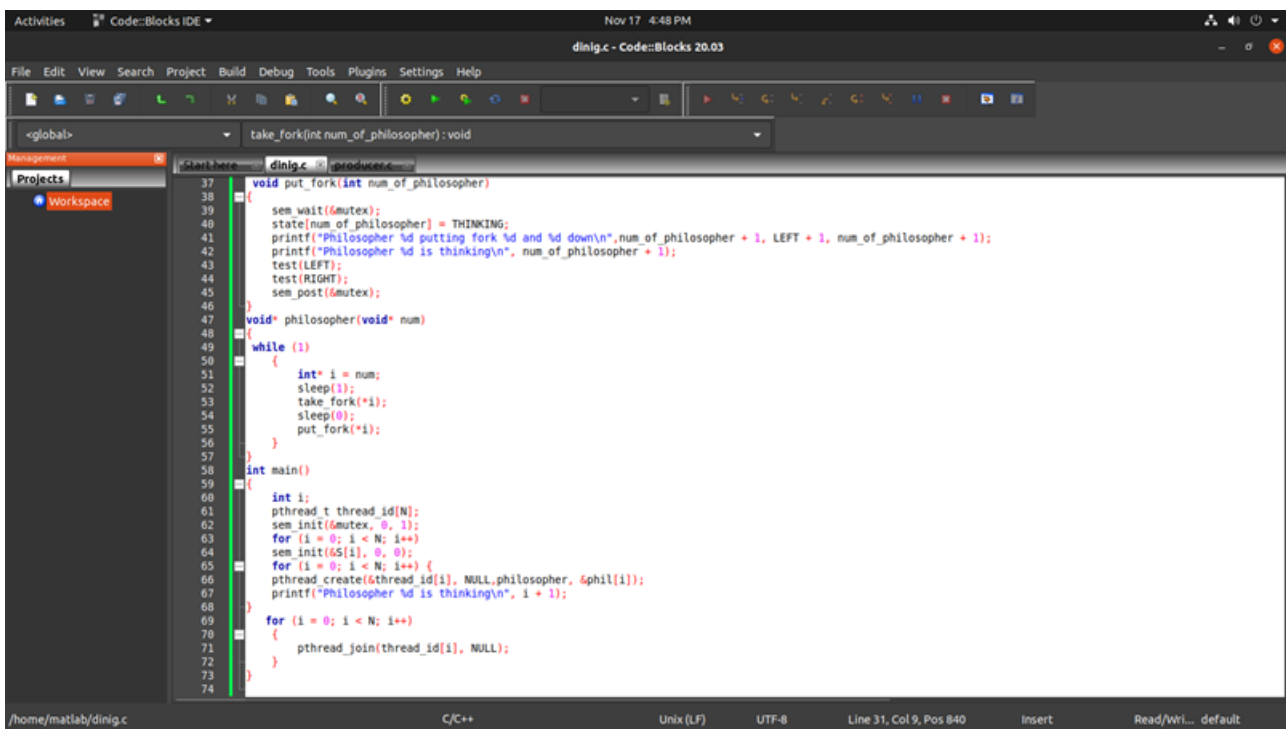
int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];
void test(int num_of_philosopher)
{
    if (state[num_of_philosopher] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[num_of_philosopher] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n", num_of_philosopher + 1, LEFT + 1,
num_of_philosopher + 1);
        printf("Philosopher %d is Eating\n", num_of_philosopher + 1);
        sem_post(&S[num_of_philosopher]);
    }
}
void take_fork(int num_of_philosopher)
{
    sem_wait(&mutex);
    state[num_of_philosopher] = HUNGRY;
    printf("Philosopher %d is Hungry\n", num_of_philosopher + 1);
    test(num_of_philosopher);
    sem_post(&mutex);
    sem_wait(&S[num_of_philosopher]);
    sleep(1);
}
void put_fork(int num_of_philosopher)
{
    sem_wait(&mutex);
    state[num_of_philosopher] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n", num_of_philosopher + 1, LEFT + 1,
num_of_philosopher + 1);
    printf("Philosopher %d is thinking\n", num_of_philosopher + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}
```

```
void* philosopher(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];
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);
    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
    for (i = 0; i < N; i++)
    {
        pthread_join(thread_id[i], NULL);
    }
}
```

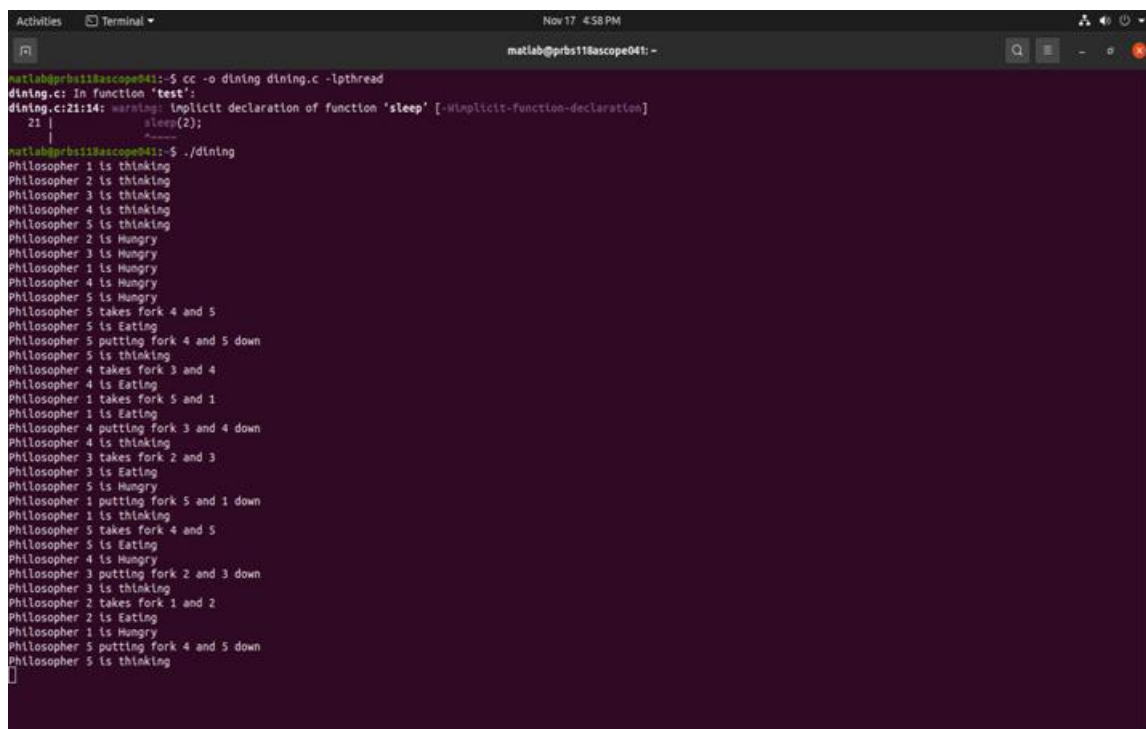


```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4
5 #define N 5
6 #define THINKING 2
7 #define HUNGRY 1
8 #define EATING 0
9 #define LEFT (num_of_philosopher + 4) % N
10 #define RIGHT (num_of_philosopher + 1) % N
11
12 int state[N];
13 int phil[N] = { 0, 1, 2, 3, 4 };
14 sem_t mutex;
15 sem_t S[N];
16
17 void test(int num_of_philosopher)
18 {
19     if (state[num_of_philosopher] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
20     {
21         state[num_of_philosopher] = EATING;
22         sleep(2);
23         printf("Philosopher %d takes fork %d and %d\n", num_of_philosopher + 1, LEFT + 1, num_of_philosopher + 1);
24         printf("Philosopher %d is Eating\n", num_of_philosopher + 1);
25         sem_post(&S[num_of_philosopher]);
26     }
27 }
28
29 void take_fork(int num_of_philosopher)
30 {
31     sem_wait(&mutex);
32     state[num_of_philosopher] = HUNGRY;
33     printf("Philosopher %d is Hungry\n", num_of_philosopher + 1);
34     test(num_of_philosopher);
35     sem_post(&mutex);
36     sem_wait(&S[num_of_philosopher]);
37     sleep(1);
38 }
39
40 void put_fork(int num_of_philosopher)
41 {
42     state[num_of_philosopher] = THINKING;
43 }
```



```
37 void put_fork(int num_of_philosopher)
38 {
39     sem_wait(&mutex);
40     state[num_of_philosopher] = THINKING;
41     printf("Philosopher %d putting fork %d and %d down\n", num_of_philosopher + 1, LEFT + 1, num_of_philosopher + 1);
42     printf("Philosopher %d is thinking\n", num_of_philosopher + 1);
43     test(LEFT);
44     test(RIGHT);
45     sem_post(&mutex);
46 }
47
48 void* philosopher(void* num)
49 {
50     while (1)
51     {
52         int* i = num;
53         sleep(1);
54         take_fork(*i);
55         sleep(0);
56         put_fork(*i);
57     }
58 }
59
60 int main()
61 {
62     int i;
63     pthread_t thread_id[N];
64     sem_init(&mutex, 0, 1);
65     for (i = 0; i < N; i++)
66     {
67         sem_init(&S[i], 0, 0);
68         for (i = 0; i < N; i++) {
69             pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
70             printf("Philosopher %d is thinking\n", i + 1);
71         }
72     }
73     for (i = 0; i < N; i++)
74     {
75         pthread_join(thread_id[i], NULL);
76     }
77 }
```

Output:-



```
matlab@prbs118ascope041:~$ cc -o dining dining.c -lpthread
dining.c: In function 'test':
dining.c:21:14: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   21 |         sleep(2);
      |         ^~~~~~
matlab@prbs118ascope041:~$ ./dining
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 1 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
```

4. Reader writer problem using monitors.

Code:-

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#include <bits/stdc++.h>
using namespace std;
// Define the data we need to create the monitor
struct monitor_DataType {
    sem_t OKtoRead;
    sem_t OKtoWrite;
    int readerCount;
    int isBusyWriting;
    // The read-queue
    int readRequested;
};
struct monitor_DataType monitor_data;

// Function that will block until write can start
void monitor_StartWrite() {
    if(monitor_data.isBusyWriting || monitor_data.readerCount != 0){
        sem_wait(&(monitor_data.OKtoWrite));
    }
    monitor_data.isBusyWriting++;    // Using 1 as true
}

// Function to signal reading is complete
void monitor_EndWrite() {
    monitor_data.isBusyWriting--;
    if(monitor_data.readRequested){
        sem_post(&(monitor_data.OKtoRead));
    } else {
        sem_post(&(monitor_data.OKtoWrite));
    }
}

// Function that will block until read can start
void monitor_StartRead() {
    if(monitor_data.isBusyWriting){
        monitor_data.readRequested++;
        sem_wait(&(monitor_data.OKtoRead));
        monitor_data.readRequested--;
    }
    monitor_data.readerCount++;
    sem_post(&(monitor_data.OKtoRead));
}

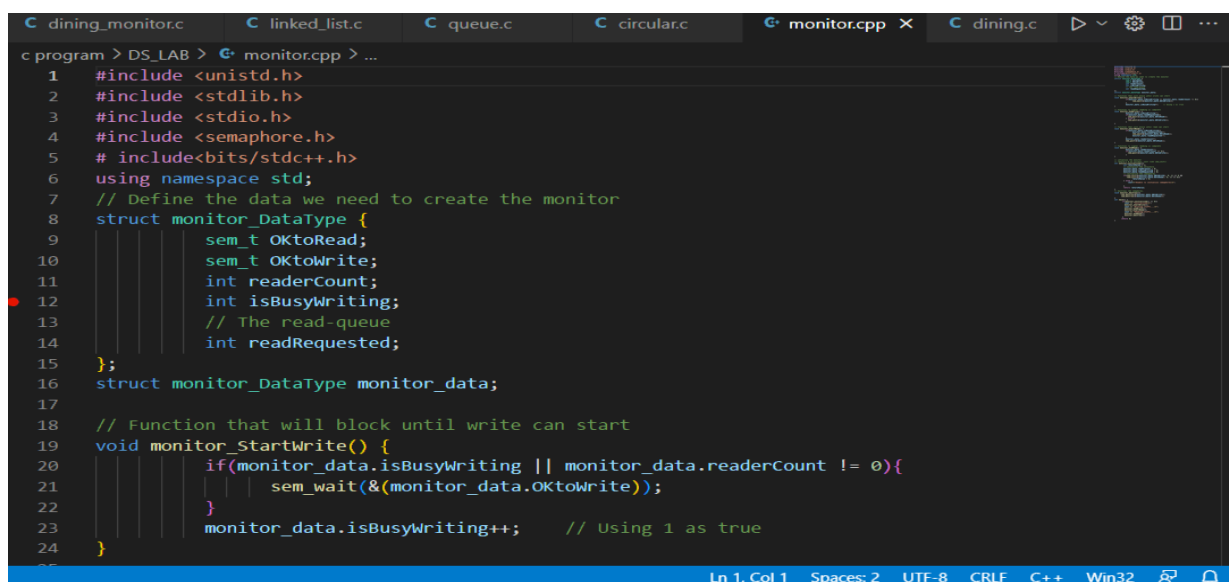
// Function to signal reading is complete
void monitor_EndRead() {
    monitor_data.readerCount--;
    if(monitor_data.readerCount == 0){
        sem_post(&(monitor_data.OKtoWrite));
    }
}
```

```
}

// initialize the monitor
// return's 0 on success, just like sem_init()
int monitor_Initialized(){
    int returnValue = 1;
    // Initialize the structure
    monitor_data.readerCount = 0;
    monitor_data.isBusyWriting = 0;
    monitor_data.readRequested = 0;
    // initialize the semaphores
    if(sem_init(&(monitor_data.OKtoWrite), 0, 1) == 0 &&
        sem_init(&(monitor_data.OKtoRead), 0, 1) == 0){
        returnValue = 0;
    } else {
        cout<<"Unable to initialize semaphores\n";
    }
    return returnValue;
}

// Destroys the semphores.
void monitor_Destroy(){
    sem_destroy(&(monitor_data.OKtoWrite));
    sem_destroy(&(monitor_data.OKtoRead));
}

int main() {
    if(monitor_Initialized() == 0){
        cout << "Initialized\n";
        monitor_StartWrite();
        cout << "Writing stuffs...\n";
        monitor_EndWrite();
        monitor_StartRead();
        cout << "Reading stuffs...\n";
        monitor_EndRead();
        monitor_Destroy();
    }
    return 0;
}
```



```
C dining_monitor.c C linked_list.c C queue.c C circular.c monitor.cpp X dining.c
c program > DS_LAB > monitor.cpp > ...
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <semaphore.h>
5 #include <bits/stdc++.h>
6 using namespace std;
7 // Define the data we need to create the monitor
8 struct monitor_DataType {
9     sem_t OKtoRead;
10    sem_t OKtoWrite;
11    int readerCount;
12    int isBusyWriting;
13    // The read-queue
14    int readRequested;
15 };
16 struct monitor_DataType monitor_data;
17
18 // Function that will block until write can start
19 void monitor_StartWrite() {
20     if(monitor_data.isBusyWriting || monitor_data.readerCount != 0){
21         sem_wait(&(monitor_data.OKtoWrite));
22     }
23     monitor_data.isBusyWriting++; // Using 1 as true
24 }
```

```
24 }
25
26 // Function to signal reading is complete
27 void monitor_EndWrite() {
28     monitor_data.isBusyWriting--;
29     if(monitor_data.readRequested){
30         sem_post(&(monitor_data.OKtoRead));
31     } else {
32         sem_post(&(monitor_data.OKtoWrite));
33     }
34 }
35
36 // Function that will block until read can start
37 void monitor_StartRead() {
38     if(monitor_data.isBusyWriting){
39         monitor_data.readRequested++;
40         sem_wait(&(monitor_data.OKtoRead));
41         monitor_data.readRequested--;
42     }
43     monitor_data.readerCount++;
44     sem_post(&(monitor_data.OKtoRead));
45 }
46
```

```
c program > DS_LAB > monitor.cpp > monitor_EndRead()
48 void monitor_EndRead() {
49     monitor_data.readerCount--;
50     if(monitor_data.readerCount == 0){
51         sem_post(&(monitor_data.OKtoWrite));
52     }
53 }
54
55 // initialize the monitor
56 // return's 0 on success, just like sem_init()
57 int monitor_Initialized(){
58     int returnValue = 1;
59     // Initialize the structure
60     monitor_data.readerCount = 0;
61     monitor_data.isBusyWriting = 0;
62     monitor_data.readRequested = 0;
63     // initialize the semaphores
64     if(sem_init(&(monitor_data.OKtoWrite), 0, 1) == 0 &&
65        sem_init(&(monitor_data.OKtoRead), 0, 1) == 0){
66         returnValue = 0;
67     } else {
68         cout<<"Unable to initialize semaphores\n";
69     }
70 }
71 return returnValue;
72
```

```
c:\program > DS_LAB > monitor.cpp > ...
69
70     }
71     return returnValue;
72 }
73 // Destroys the semphores.
74 void monitor_Destroy(){
75     sem_destroy(&(monitor_data.OKtoWrite));
76     sem_destroy(&(monitor_data.OKtoRead));
77 }
78 int main() {
79     if(monitor_Initialized() == 0){
80         cout << "Initialized\n";
81         monitor_StartWrite();
82         cout << "Writing stuffs...\n";
83         monitor_EndWrite();
84         monitor_StartRead();
85         cout << "Reading stuffs...\n";
86         monitor_EndRead();
87         monitor_Destroy();
88     }
89     return 0;
90 }
91
```

Output:-

```
student@sh-4.4-desktop:~$ gcc rw1.c -lpthread
student@sh-4.4-desktop:~$ ./a.out
-----
reader-0 is reading
updated value : 5

writer-0 is writing
-----
reader-1 is reading
updated value : 10
-----
reader-2 is reading
updated value : 10
-----
reader-3 is reading
updated value : 10

writer-3 is writing
-----
reader-4 is reading
```