ASSIGNMENT

By

Riya Sharma

2022A1R022

3rd Sem

CSE Department

Model Institute of Engineering & Technology (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade)

Jammu, India

2023

ASSIGNMENT
Group D

Subject Code: Operating System [COM-302]

Due Date: 4/12/23

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	
Faculty Signature: Dr. Mekhla Sharma (Assistant Professor) Email: mekhla.cse@mietjammu.in				

Task 1:

Analyze the differences between mutex locks and semaphores in terms of functionality and use cases for synchronization. Explain situations in which you would choose one over the other and provide specific examples to support your analysis.

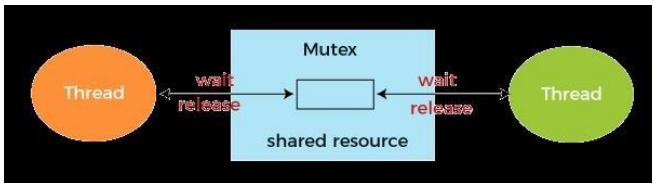
SOL:

Analysis Report: Mutex Locks vs. Semaphores for Synchronization

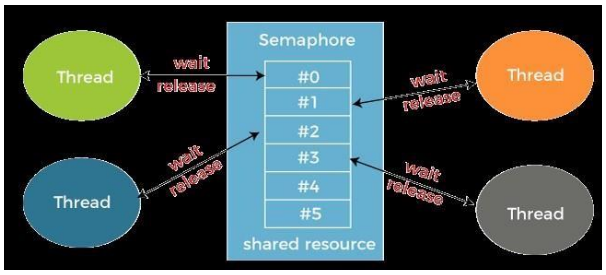
1. <u>Introduction:</u>

As per operating system terminology, mutex and semaphores are kernel resources that provide synchronization services, also called synchronization primitives. Process synchronization plays an important role in maintaining the consistency of shared data. Both the software and hardware solutions are present for handling critical section problems. But hardware solutions for critical section problems are quite difficult to implement. Mutex and semaphore both provide synchronization services, but they are not the same.

Mutex is a mutual exclusion object that synchronizes access to a resource. It is a special type of binary semaphore used for controlling access to the shared resource

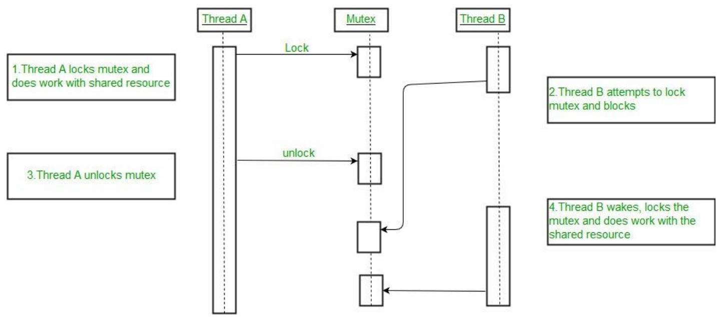


whereas Semaphore is simply a variable that is non-negative and shared between threads. It is distinguished by the operating system in two categories Counting semaphore and Binary semaphore.



2. <u>Functionality:</u>

i. **Mutex:** A mutex, short for "mutual exclusion", is a binary semaphore used to ensure that only one thread can execute a certain section of code. This ensures data integrity and consistency in the application. Mutexes can only be acquired by a single thread at a time, hence preventing any conflicts.

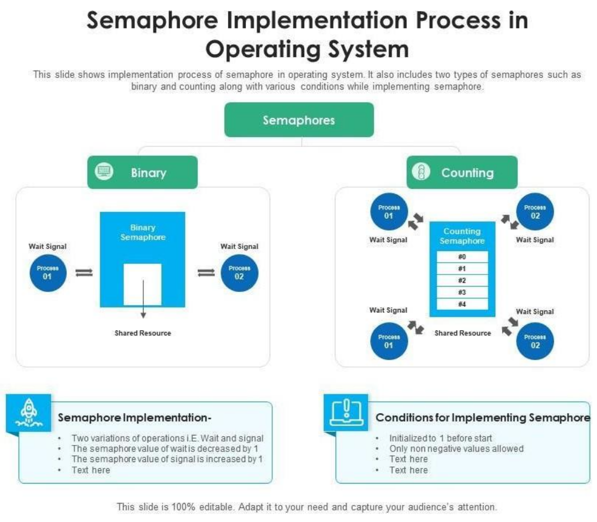

Thread A
Mutex
Thread B
Lock
1.Thread A locks mutex and does work with shared resource
2.Thread B attempts to lock mutex and blocks
unlock
3.Thread A unlocks mutex
4.Thread B wakes, locks the mutex and does work with the shared resource

<u>An example to show how mutexes are used for thread synchronization</u>

ii. **Semaphore**: A semaphore, on the other hand, is a synchronization tool that can control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. It can control the number of threads executing a certain piece of code.

- <u>Binary Semaphore</u>: It is similar to a mutex but can also be released by a different thread. This means that once a thread acquires a binary semaphore, another thread can also release it, allowing other threads to proceed.

- <u>Counting Semaphore</u>: It can handle a larger number of concurrent processes compared to binary semaphores. Counting semaphores use a counter that represents the number of available resources. Acquiring a resource decrements the counter, and releasing a resource increments it.



Semaphore Implementation Process in Operating System

This slide shows implementation process of semaphore in operating system. It also includes two types of semaphores such as binary and counting along with various conditions while implementing semaphore.

Semaphore Implementation-

- Two variations of operations i.E. Wait and signal
- The semaphore value of wait is decreased by 1
- The semaphore value of signal is increased by 1
- Text here

Conditions for Implementing Semaphore

- Initialized to 1 before start
- Only non negative values allowed
- Text here
- Text here

This slide is 100% editable. Adapt it to your need and capture your audience's attention.

In **summary**, mutexes are used for thread synchronization, ensuring that only one thread can execute a certain section of code at a time. On the other hand, semaphores can be used for thread synchronization and inter-process synchronization. Semaphores provide more flexibility than mutexes as they can handle a larger number of concurrent processes. However, both mutexes and semaphores can lead to deadlock situations if not handled properly.

3. <u>Use Cases:</u>

i. **Mutex Locks:**

- **Exclusive Resource Access**: Use mutex locks when only one thread at a time should access a particular resource (e.g. a critical section).

- **Avoiding Deadlocks**: Well-suited for scenarios where deadlock avoidance is critical due to their ownership concept.

ii. **Semaphores:**

- **Resource Pool Management**: Suitable for managing a pool of resources where multiple threads can access resources simultaneously up to a specified limit.
- **Producer-Consumer Problem**: Effective in scenarios involving multiple producers and consumers where a fixed-size buffer needs to be managed.

In **summary**, mutexes are used for ensuring exclusive access to a shared resource by a single thread, while semaphores are used for controlling access to a shared resource by multiple threads. Both can be employed to synchronize access to shared resources in multi-threaded environments, but their use cases and characteristics are different.

4. <u>Choosing Between Mutex Locks and Semaphores</u>:

a) **If Exclusive Access is Needed:**
Example: Managing access to a printer in a network. A mutex lock ensures that only one job is printed at a time.

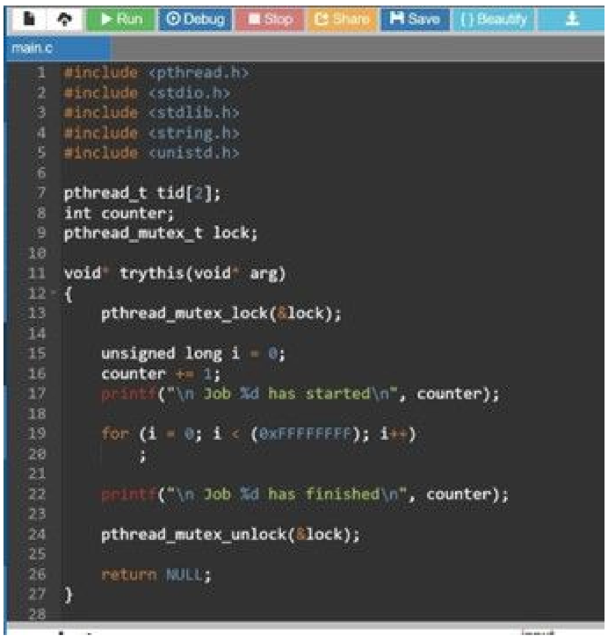
b) **If Resource Pool Management is Required:**
Example: Connection pool in a database system. Semaphores can be used to control the number of simultaneous database
conn
ections.

c) **If Deadlock Avoidance is Critical:**
Example: File system operations. Mutex locks are preferred to avoid potential deadlocks due to their ownership concept.

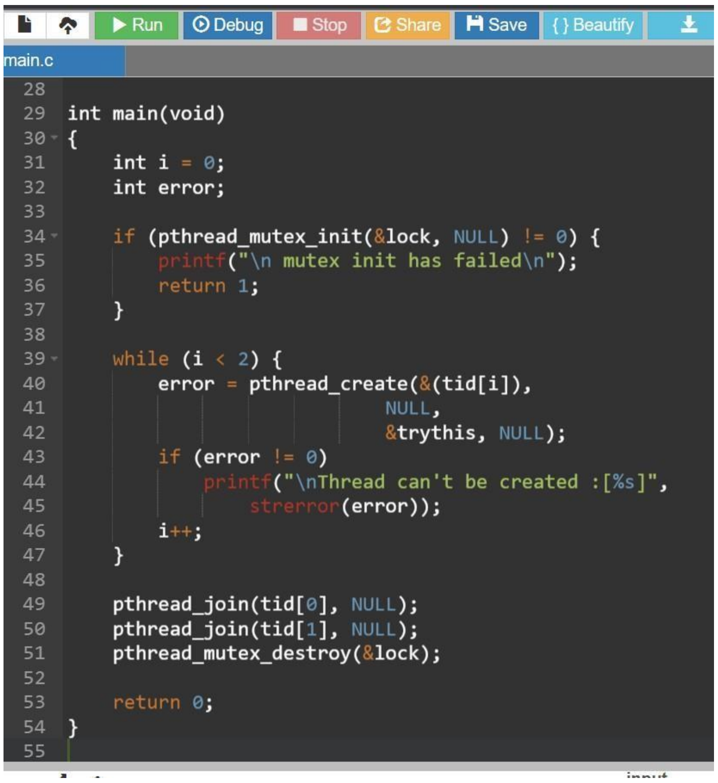
Therefore, to synchronize threads across process boundaries, use mutexes. To synchronize access to limited resources, use a semaphore.

5. Examples:

Mutex Lock Example:



```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}

```

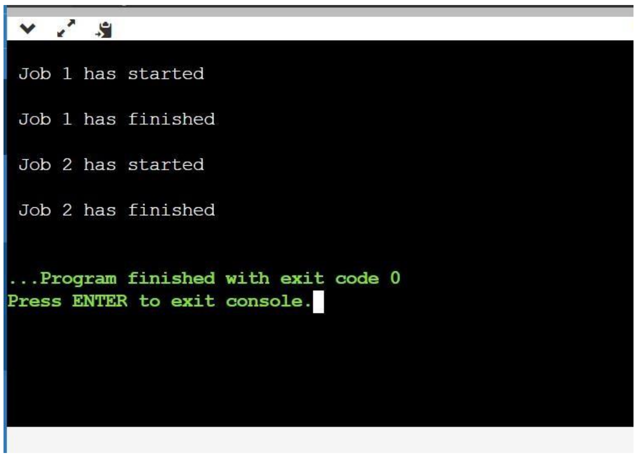


```c

int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}

```

In the above code (written in c language):

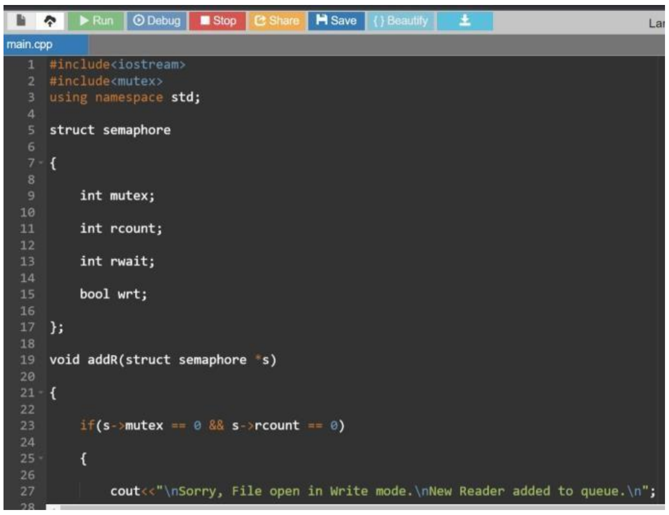
- A mutex is initialized in the beginning of the main function.
- The same mutex is locked in the ‘trythis ()’ function while using the shared resource ‘counter’.
- At the end of the function ‘trythis ()’ the same mutex is unlocked.
- At the end of the main function when both the threads are done, the mutex is destroyed.

Output of the above code:

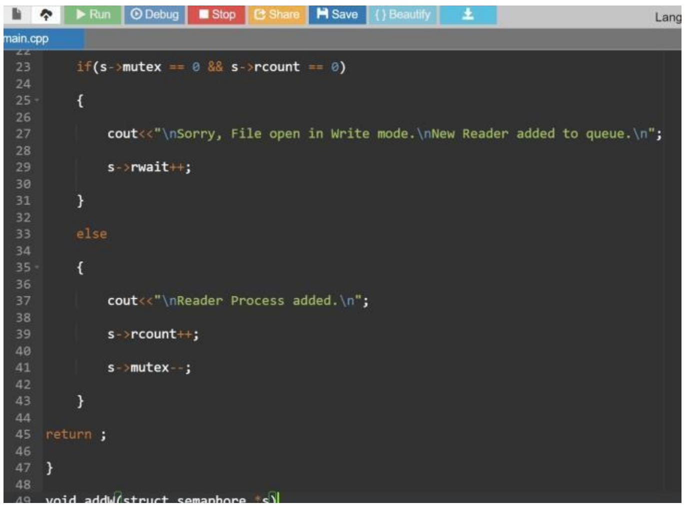


```
Job 1 has started

Job 1 has finished

Job 2 has started

Job 2 has finished


...Program finished with exit code 0
Press ENTER to exit console.
```

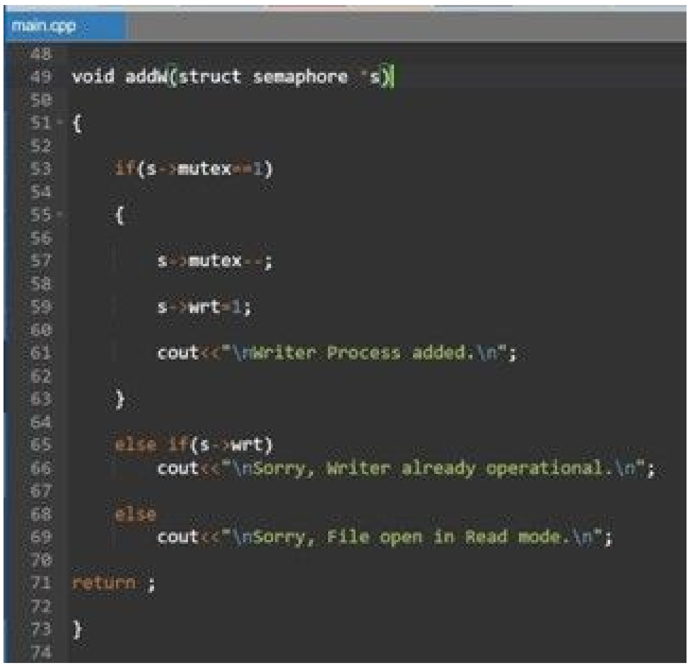
Semaphore Example:



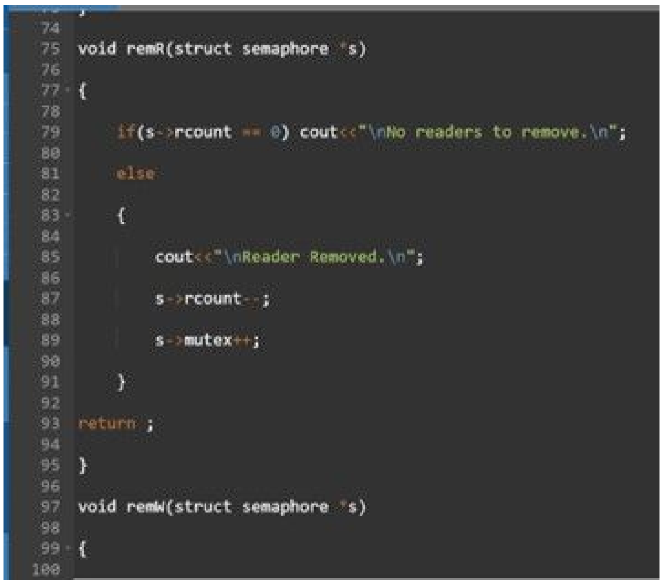
```
#include<iostream>
#include<mutex>
using namespace std;

struct semaphore

{

    int mutex;

    int rcount;

    int rwait;

    bool wrt;

};

void addR(struct semaphore *s)

{

    if(s->mutex == 0 && s->rcount == 0)

    {

        cout<<"\nSorry, File open in Write mode.\nNew Reader added to queue.\n";
```



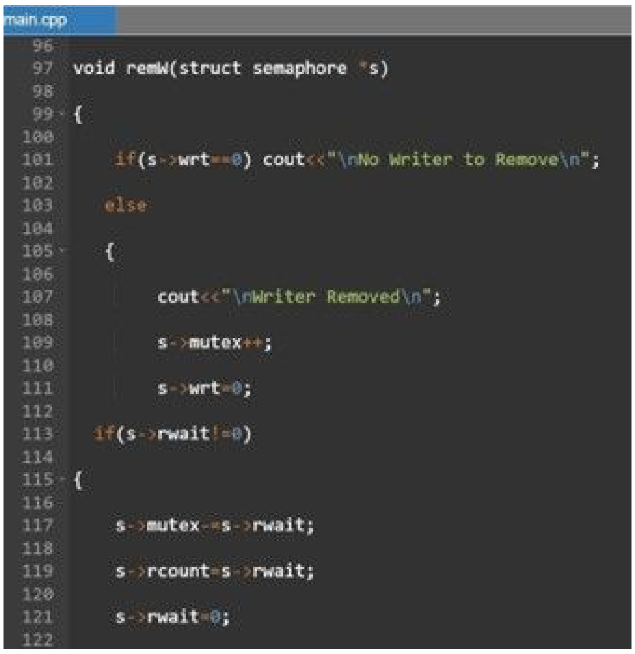
```
    if(s->mutex == 0 && s->rcount == 0)

    {

        cout<<"\nSorry, File open in Write mode.\nNew Reader added to queue.\n";

        s->rwait++;

    }

    else

    {

        cout<<"\nReader Process added.\n";

        s->rcount++;

        s->mutex--;

    }

return ;

}

void addW(struct semaphore *s)
```



```
void addW(struct semaphore *s)

{

    if(s->mutex==1)

    {

        s->mutex--;

        s->wrt=1;

        cout<<"\nWriter Process added.\n";

    }

    else if(s->wrt)
        cout<<"\nSorry, Writer already operational.\n";

    else
        cout<<"\nSorry, File open in Read mode.\n";

return ;

}
```

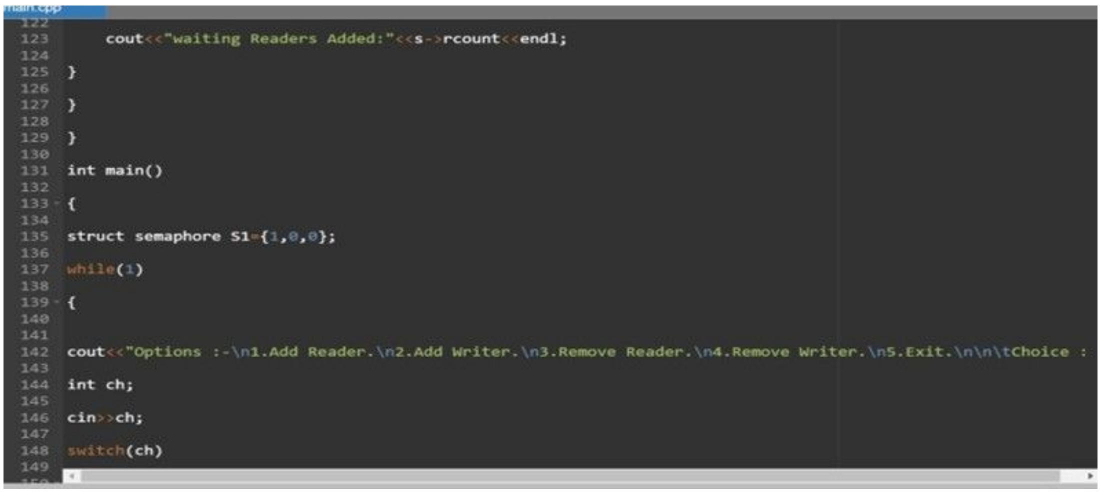


```cpp
void remR(struct semaphore *s)

{

    if(s->rcount == 0) cout<<"\nNo readers to remove.\n";

    else

    {

        cout<<"\nReader Removed.\n";

        s->rcount--;

        s->mutex++;

    }

return ;

}

void remW(struct semaphore *s)

{

```



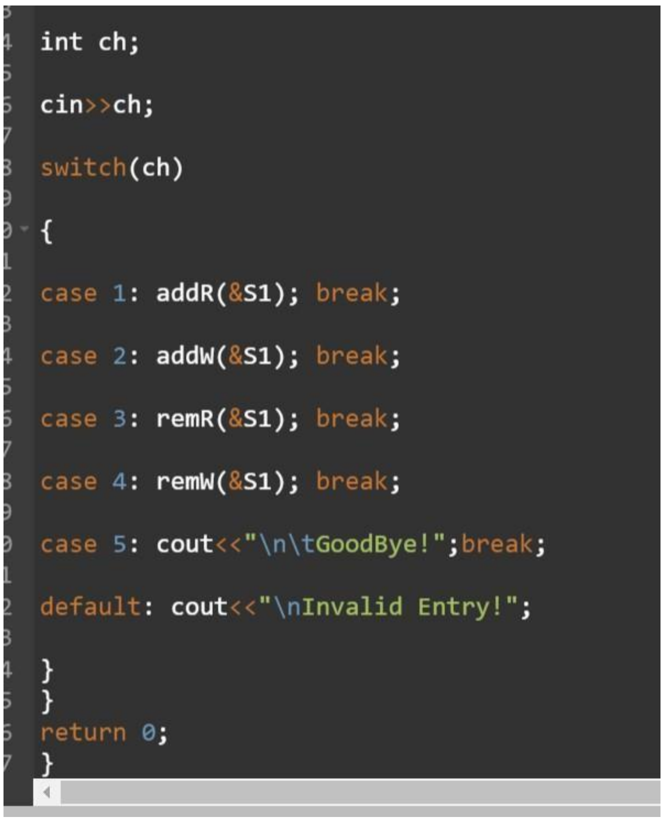
main.cpp

```cpp

void remW(struct semaphore *s)

{

     if(s->wrt==0) cout<<"\nNo Writer to Remove\n";

    else

    {

         cout<<"\nWriter Removed\n";

         s->mutex++;

         s->wrt=0;

  if(s->rwait!=0)

{

    s->mutex-=s->rwait;

    s->rcount=s->rwait;

    s->rwait=0;

```



main.cpp

```cpp

     cout<<"waiting Readers Added:"<<s->rcount<<endl;

}

}

}

int main()

{

struct semaphore S1={1,0,0};

while(1)

{


cout<<"Options :-\n1.Add Reader.\n2.Add Writer.\n3.Remove Reader.\n4.Remove Writer.\n5.Exit.\n\n\tChoice :

int ch;

cin>>ch;

switch(ch)

```



```
int ch;

cin>>ch;

switch(ch)

{

case 1: addR(&S1); break;

case 2: addW(&S1); break;

case 3: remR(&S1); break;

case 4: remW(&S1); break;

case 5: cout<<"\n\tGoodBye!";break;

default: cout<<"\nInvalid Entry!";

}
}
return 0;
}
```

This C++ code implements a simple reader-writer solution using a semaphore structure. The semaphore structure contains fields for a mutex (binary semaphore), reader count (`rcount`), reader wait count (`rwait`), and a Boolean flag for indicating whether a writer is currently operating (`wrt`). The program provides options to add readers, add writers, remove readers, remove writers, or exit the program in an infinite loop.

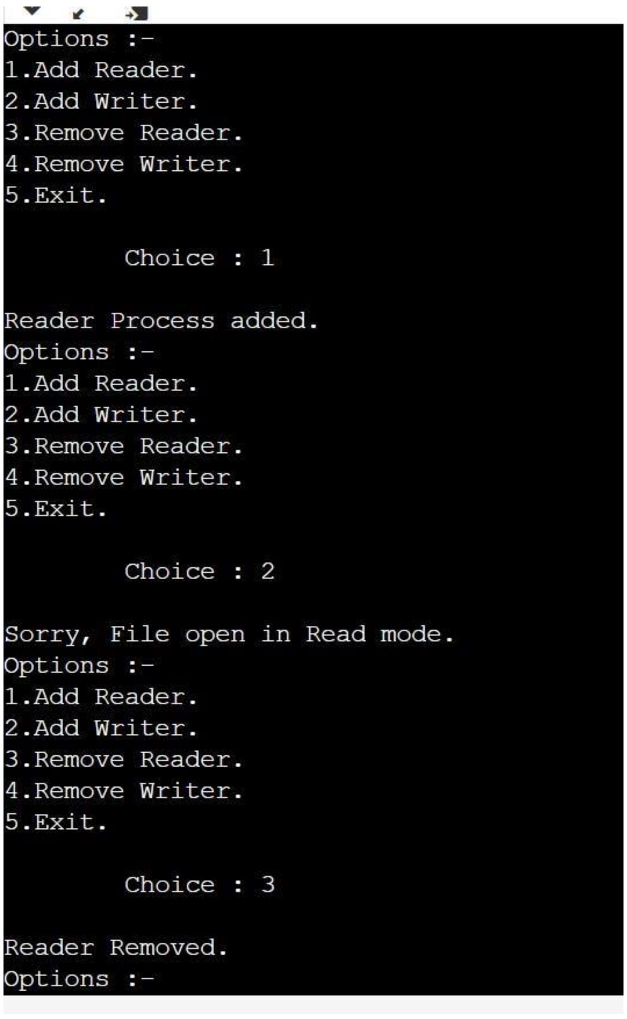
The `addR` function adds a reader to the system if there are no active writers. If a writer is present, the reader is added to the wait queue. The `addW` function adds a writer if there are no other active processes, i.e., no readers or writers. If a writer is already operating, it displays a message, and if readers are present, it indicates that the file is open inread mode.

The `remR` function removes a reader if there are any, and the `remW` function removes a writer if one is present. If there are waiting readers when a writer is removed, they are all added to the system, and the read count is updated accordingly.

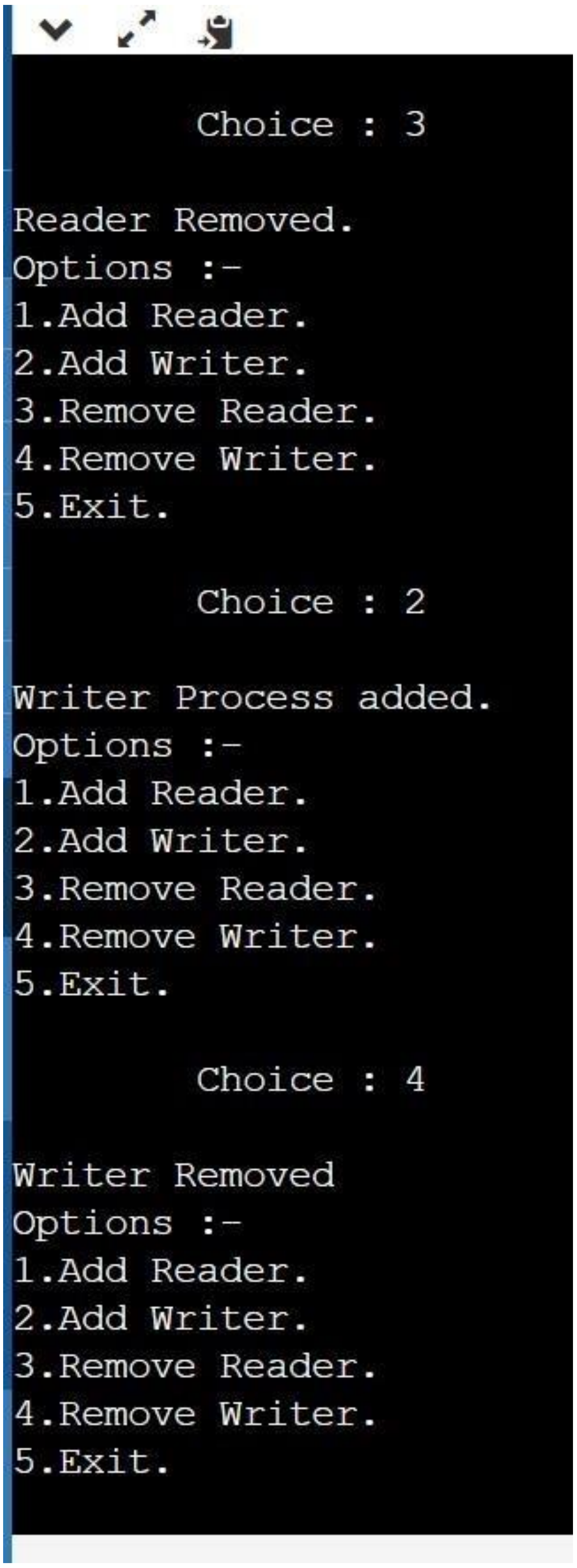
The main function presents a menu for the user to choose options, such as adding readers or writers, removing readers or writers, or exiting the program. The program runs indefinitely until the user chooses to exit.

Overall, this code demonstrates a basic implementation of a reader-writer problem using semaphores to control access to shared resources.

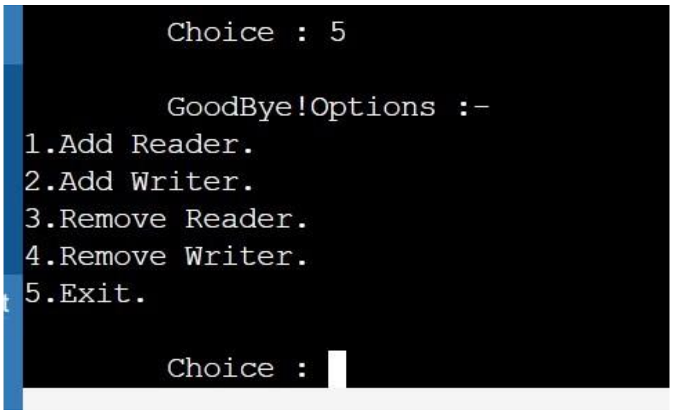
Output of the above code:



```
Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.

        Choice : 1

Reader Process added.
Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.

        Choice : 2

Sorry, File open in Read mode.
Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.

        Choice : 3

Reader Removed.
Options :-
```



```
        Choice : 3

Reader Removed.
Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.

        Choice : 2

Writer Process added.
Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.

        Choice : 4

Writer Removed
Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.
```



```
        Choice : 5

        GoodBye!Options :-
1.Add Reader.
2.Add Writer.
3.Remove Reader.
4.Remove Writer.
5.Exit.

        Choice : 
```

6. Conclusion:
Choosing between mutex locks and semaphores depends on the specific requirements of the concurrent program. Mutex locks are suitable for scenarios requiring exclusive access and deadlock avoidance, while semaphores are preferred for managing resource pools and situations where multiple threads can access resources concurrently. Understanding the nuances of each synchronization mechanism is crucial for designing efficient and correct concurrent programs.

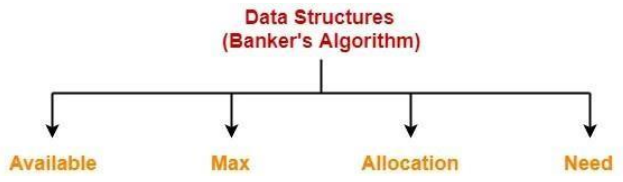
Task 2:

Write a program that implements the Banker's algorithm for deadlock avoidance. Simulate multiple processes making resource requests and releases. Demonstrate how the algorithm ensures safe states and prevents deadlocks. Discuss the advantages and limitations of the Banker's algorithm.

SOL:
The purpose of the Banker’s Algorithm is to prevent deadlock, a situation where two or more processes are blocked, waiting for each other to release resources they need in order to proceed. Deadlock can cause a system to become unresponsive and may require a reboot to recover.

The data structure used are:

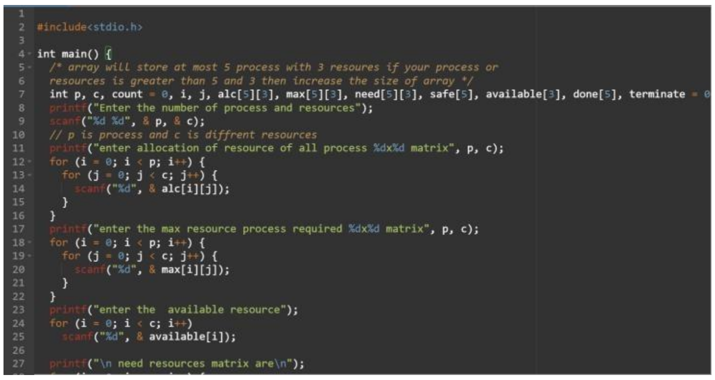
- Available vector
- Max Matrix
- Allocation Matrix
- Need Matrix



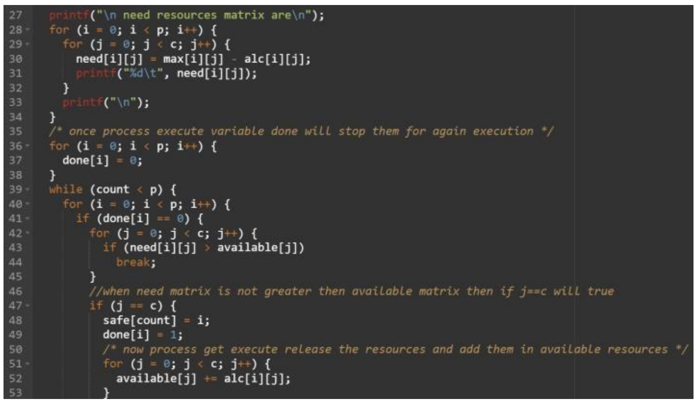
Banker algorithm program in C for deadlock avoidance:

The banker algorithm is developed by Edsger Dijkstra and used for deadlock avoidance by executing processes according to the resources they need. this algorithm is basically testing whether the safe state exists or not.

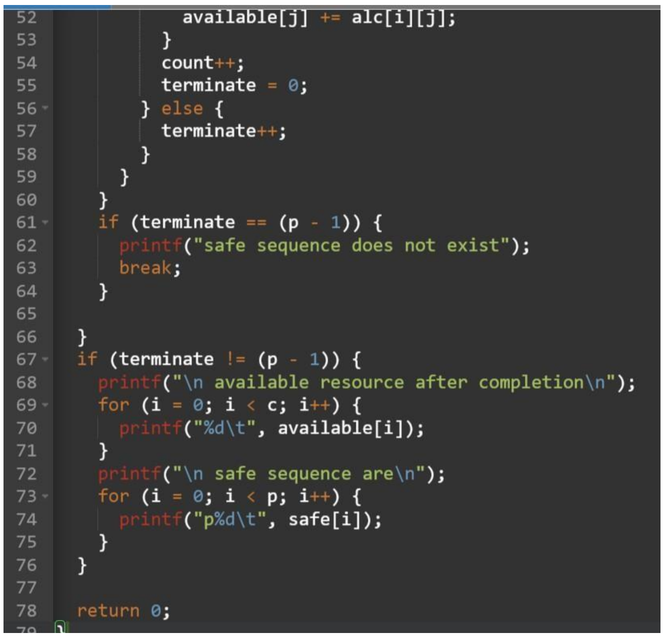
C program for bankers’ algorithm



```c

#include<stdio.h>

int main() {
  /* array will store at most 5 process with 3 resoures if your process or
  resources is greater than 5 and 3 then increase the size of array */
  int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3], done[5], terminate = 0
  printf("Enter the number of process and resources");
  scanf("%d %d", & p, & c);
  // p is process and c is diffrent resources
  printf("enter allocation of resource of all process %dx%d matrix", p, c);
  for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
      scanf("%d", & alc[i][j]);
    }
  }
  printf("enter the max resource process required %dx%d matrix", p, c);
  for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
      scanf("%d", & max[i][j]);
    }
  }
  printf("enter the  available resource");
  for (i = 0; i < c; i++)
    scanf("%d", & available[i]);

  printf("\n need resources matrix are\n");
```



```c
  printf("\n need resources matrix are\n");
  for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
      need[i][j] = max[i][j] - alc[i][j];
      printf("%d\t", need[i][j]);
    }
    printf("\n");
  }
  /* once process execute variable done will stop them for again execution */
  for (i = 0; i < p; i++) {
    done[i] = 0;
  }
  while (count < p) {
    for (i = 0; i < p; i++) {
      if (done[i] == 0) {
        for (j = 0; j < c; j++) {
          if (need[i][j] > available[j])
            break;
        }
        //when need matrix is not greater then available matrix then if j==c will true
        if (j == c) {
          safe[count] = i;
          done[i] = 1;
          /* now process get execute release the resources and add them in available resources */
          for (j = 0; j < c; j++) {
            available[j] += alc[i][j];
          }
```



```c
            available[j] += alc[i][j];
          }
          count++;
          terminate = 0;
        } else {
          terminate++;
        }
      }
    }
    if (terminate == (p - 1)) {
      printf("safe sequence does not exist");
      break;
    }

  }
  if (terminate != (p - 1)) {
    printf("\n available resource after completion\n");
    for (i = 0; i < c; i++) {
      printf("%d\t", available[i]);
    }
    printf("\n safe sequence are\n");
    for (i = 0; i < p; i++) {
      printf("p%d\t", safe[i]);
    }
  }

  return 0;
}
```

The provided C program simulates the Banker's algorithm for deadlock avoidance. Let's break down how the program works and how it demonstrates the Banker's algorithm:

1. User Input:
- The user is prompted to input the number of processes (`p`) and the number of different types of resources (`c`).
- The allocation matrix (`alc`), maximum matrix (`max`), and available resources vector (`available`) are inputted.

2. Initialization:
   - The program initializes arrays to store information about the allocation, maximum claim, need, and availability of resources.
   - It also initializes arrays to keep track of whether a process is done and a variable (`terminate`) to check for termination.

3. Calculating Need Matrix:
- The program calculates the need matrix, which represents the remaining resources that each process needs to complete its execution.

4. Main Algorithm Loop:
- The program enters a loop until all processes are done (`count` equals the number of processes).
- Within the loop, it iterates through each process (`i`) to check if it can be executed.

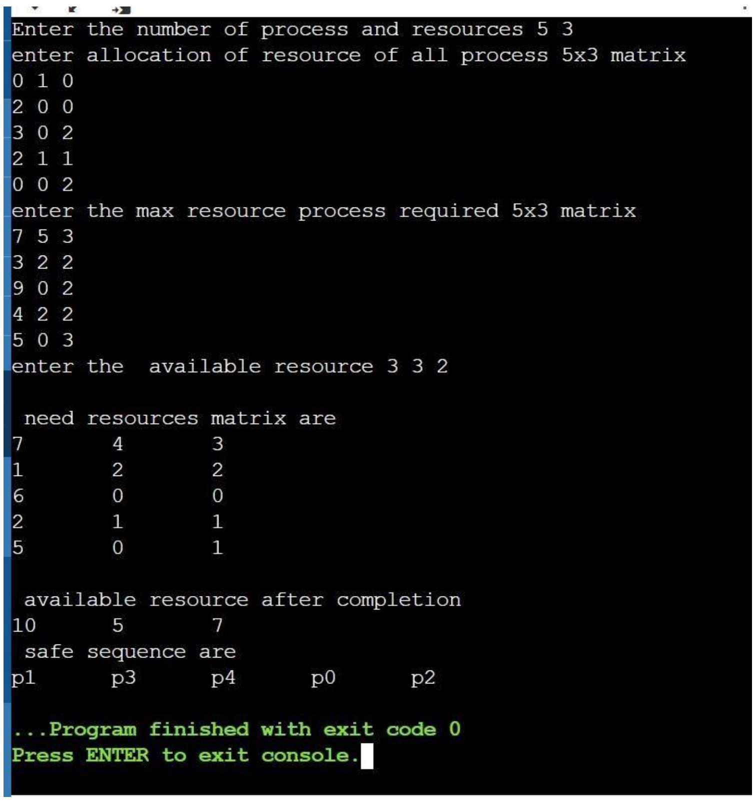
5. Checking Safety:
- For each process, it checks if the need for each resource is less than or equal to the available resources.
- If true for all resources, the process is considered safe, and its resources are released (`available` resources are increased).
- The process is marked as done, and the loop continues.

6. Termination Check:
- If a process cannot be executed (i.e., all processes are in a deadlock situation), the program terminates with a message indicating that a safe sequence does not exist.

7. Output:
- If the program successfully finds a safe sequence, it outputs the available resources after completion and the safe sequence of processes.

In short, this program simulates multiple processes making resource requests and releases. The Banker's algorithm ensures that the system remains in a safe state by carefully considering the resource allocation and need matrices. If a safe sequence exists, the algorithm allows processes to execute and releases resources accordingly, preventing deadlocks.

Output of the above bankers' algorithm program:



```
Enter the number of process and resources 5 3
enter allocation of resource of all process 5x3 matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
enter the max resource process required 5x3 matrix
7 5 3
3 2 2
9 0 2
4 2 2
5 0 3
enter the  available resource 3 3 2

 need resources matrix are
7       4       3
1       2       2
6       0       0
2       1       1
5       0       1

 available resource after completion
10      5       7
 safe sequence are
p1      p3      p4      p0      p2

...Program finished with exit code 0
Press ENTER to exit console.
```

Advantages of Bankers Algorithm:

- Here are some advantages of this algorithm:

1. **Safety**: This algorithm ensures that a system never gets into an unsafe state, meaning it can avoid potential deadlocks. By considering all possible execution scenarios, the Banker's Algorithm can prevent a system from ending up in a state where it cannot fulfill the next request, which could potentially cause the system to crash.

2. **Deadlock detection:** This algorithm can also detect deadlocks before they occur. By comparing the number of resources a process needs to complete its execution with the number of resources currently available, the Banker's Algorithm can identify whether a system can fulfil a request and avoid entering a deadlocked state.

3. **Efficient resource allocation**: This algorithm takes into account all the processes that are currently executing and can be executed in the future. It allocates resources in such a way that it can meet the needs of all these processes while maintaining safety and avoiding deadlocks.

4. **Easy implementation:** The Banker's Algorithm is relatively simple to implement. It requires a centralized resource management system and a tracking mechanism for resource availability. Additionally, the algorithm does not require real-time updates to the resource availability.

   In conclusion, the Banker's Algorithm is a reliable and efficient resource management technique that offers significant advantages in terms of system safety, deadlock detection, and resource allocation.

Limitations of Bankers Algorithm:

While the Banker's Algorithm is an effective resource management technique, it also has some limitations:

1. **Overhead**: Implementing the Banker's Algorithm can be complex and may introduce a significant overhead, particularly for large systems with many processes.

2. **Non-pre-emptive nature**: The Banker's Algorithm assumes that a process will release all the resources it is currently holding before it can request more. This may not always be practical in real-world scenarios, where a process may need to release some resources but not all, while it is still executing.

3. **Limited applicability**: The Banker's Algorithm is specifically designed for deadlock prevention in a resource- constrained environment. It may not be suitable for applications that do not require resource management or deadlock prevention, such as certain parallel computing scenarios.

4. **Uniform priority scheduling assumption:** The Banker's Algorithm assumes that all processes will be executed in a uniform priority order. This assumption may not always hold true in real-world systems, where processes may have different priority levels or may be executed in a non-uniform order.

   Despite these limitations, the Banker's Algorithm remains a valuable resource management technique and can be successfully employed in appropriate systems and applications.