

**SUBMITTED TO:**

|  |  |
| --- | --- |
| **Name of the faculty:** | **Geetika Chatley ma’am** |
| **Course Code:** | **CSE 316** |
| **Course Title:** | **Operating System** |

**SUBMITTED BY:**

|  |  |
| --- | --- |
| **Student Name:** | **Tanna Manohar** |
| **Student ID:** | **11710378** |
| **Email Address:** | **manohartanna12@gmail.com** |
| **GitHub Link:** |  |

**CONTENTS:**

1. Code
2. Description
3. Algorithm
4. Code snippet
5. Description (purpose of use)
6. Description (If you have implemented any additional algorithm to support the solution, explain the need and usage of the same.)
7. Description (Explaining the boundary conditions of the implemented code.)
8. Description (Explain all the test cases applied on the solution of assigned problem.)
9. **Code:**

#include <semaphore.h>

#define MAX\_RESOURCES 5

// available\_resources would be involved in the race condition

int available\_resources = MAX\_RESOURCES;

/\*

decrease available\_resources by count resources

return 0 if sufficient resources available,

otherwise return -1

\*/

int decrease\_count(int count)

{

if (available\_resources < count)

return -1;

else

{

available\_resources -= count;

return 0;

}

}

// increase available\_resources by count

int increase\_count(int count)

{

available\_resources += count;

return 0;

}

int main(void)

{

sem\_t semaphore;

sem\_init(&semaphore, 0, 1);

printf("\n Job %d started\n", sem\_init);

sem\_wait(&semaphore);

printf("\n Job %d waiting\n", sem\_wait);

sem\_post(&semaphore); sem\_post(&semapore);

}

1. **Description:**

The project statement is in a software packages we provide a given numbers or licenses .That indicates number of applications that we run concurrently. When the counting of applications started the count will be decreased ,when the process is completed the count will increase again. If the process or licenses are using then we get denied until the process is completed and we have to increase and decrease the count, the problem statement can be solved using Semaphores or mutex locks. When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

1. **Algorithm**

In this problem statement I used the Semaphore’s. by using semaphore’s we can avoid the race condition, race condition means The situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely when are called race conditions. To see how interprocess communication works in practice, let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name in a special spooler directory. Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and removes their names from the directory. To avoid race condition To avoid race condition we need Mutual Exclusion. Mutual Exclusion is someway of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things. The difficulty above in the printer spooler occurs because process B started using one of the shared variables before process A was finished with it. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. (Rules for avoiding Race Condition)

Solution to Critical section problem:

1. No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)

2. No assumptions may be made about speeds or the number of CPUs.

3. No process running outside its critical region may block other processes.

4. No process should have to wait forever to enter its critical region.

We can use critical section to avoid the race condition.

The proposed algorithm in semaphore’s is:

* + - * Counting Semaphore
      * Binary Semaphore or Mutex

Counting semaphore:

There are the scenarios in which more than one processes need to execute in critical section simultaneously. However, counting semaphore can be used when we need to have more than one process in the critical section at the same time.

The programming code of semaphore implementation is shown below which includes the structure of semaphore and the logic using which the entry and the exit can be performed in the critical section.

* struct Semaphore
* {
* int value; // processes that can enter in the critical section simultaneously.
* queue type L; // L contains set of processes which get blocked
* }
* Down (Semaphore S)
* {
* SS.value = S.value - 1; //semaphore's value will get decreased when a new
* //process enter in the critical section
* if (S.value**<** **0**)
* {
* put\_process(PCB) in L; //if the value is negative then
* //the process will get into the blocked state.
* Sleep();
* }
* else
* return;
* }
* up (Semaphore s)
* {
* SS.value = S.value+1; //semaphore value will get increased when
* //it makes an exit from the critical section.
* if(S.value**<**=0)
* {
* select a process from L; //if the value of semaphore is positive
* //then wake one of the processes in the blocked queue.
* wake-up();
* }
* }
* }

In this mechanism, the entry and exit in the critical section are performed on the basis of the value of counting semaphore. The value of counting semaphore at any point of time indicates the maximum number of processes that can enter in the critical section at the same time.

A process which wants to enter in the critical section first decrease the semaphore value by 1 and then check whether it gets negative or not. If it gets negative then the process is pushed in the list of blocked processes (i.e. q) otherwise it gets enter in the critical section.

When a process exits from the critical section, it increases the counting semaphore by 1 and then checks whether it is negative or zero. If it is negative then that means that at least one process is waiting in the blocked state hence, to ensure bounded waiting, the first process among the list of blocked processes will wake up and gets enter in the critical section.

The processes in the blocked list will get waked in the order in which they slept. If the value of counting semaphore is negative then it states the number of processes in the blocked state while if it is positive then it states the number of slots available in the critical section.

Binary Semaphore or Mutex:

In counting semaphore, Mutual exclusion was not provided because we has the set of processes which required to execute in the critical section simultaneously.

However, Binary Semaphore strictly provides mutual exclusion. Here, instead of having more than 1 slots available in the critical section, we can only have at most 1 process in the critical section. The semaphore can have only two values, 0 or 1.

Let's see the programming implementation of Binary Semaphore.

* StructBsemaphore
* {
* enum Value(0,1); //value is enumerated data type which can only have two values 0 or 1.
* Queue type L;
* }
* /\* L contains all PCBs corresponding to process
* Blocked while processing down operation unsuccessfully.
* \*/
* Down (Bsemaphore S)
* {
* if (s.value == 1) // if a slot is available in the
* //critical section then let the process enter in the queue.
* {
* S.value = 0; // initialize the value to 0 so that no other process can read it as 1.
* }
* else
* {
* put the process (PCB) in S.L; //if no slot is available
* //then let the process wait in the blocked queue.
* sleep();
* }
* }
* Up (Bsemaphore S)
* {
* if (S.L is empty) //an empty blocked processes list implies that no process
* //has ever tried to get enter in the critical section.
* {
* S.Value =1;
* }
* else
* {
* Select a process from S.L;
* Wakeup(); // if it is not empty then wake the first process of the blocked queue.
* }
* }

1. **Calculating the complexity of implemented algorithm.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function Name** | **NLOC** | **Complexity** | **Token #** |  |
| decrease\_count | 10 | 2 | 27 |  |
| increase\_count | 5 | 1 | 14 |  |
| main | 9 | 1 | 51 |  |

Here, the decrease count complexity is 2 and NLOC is 10 the tokens are 27.

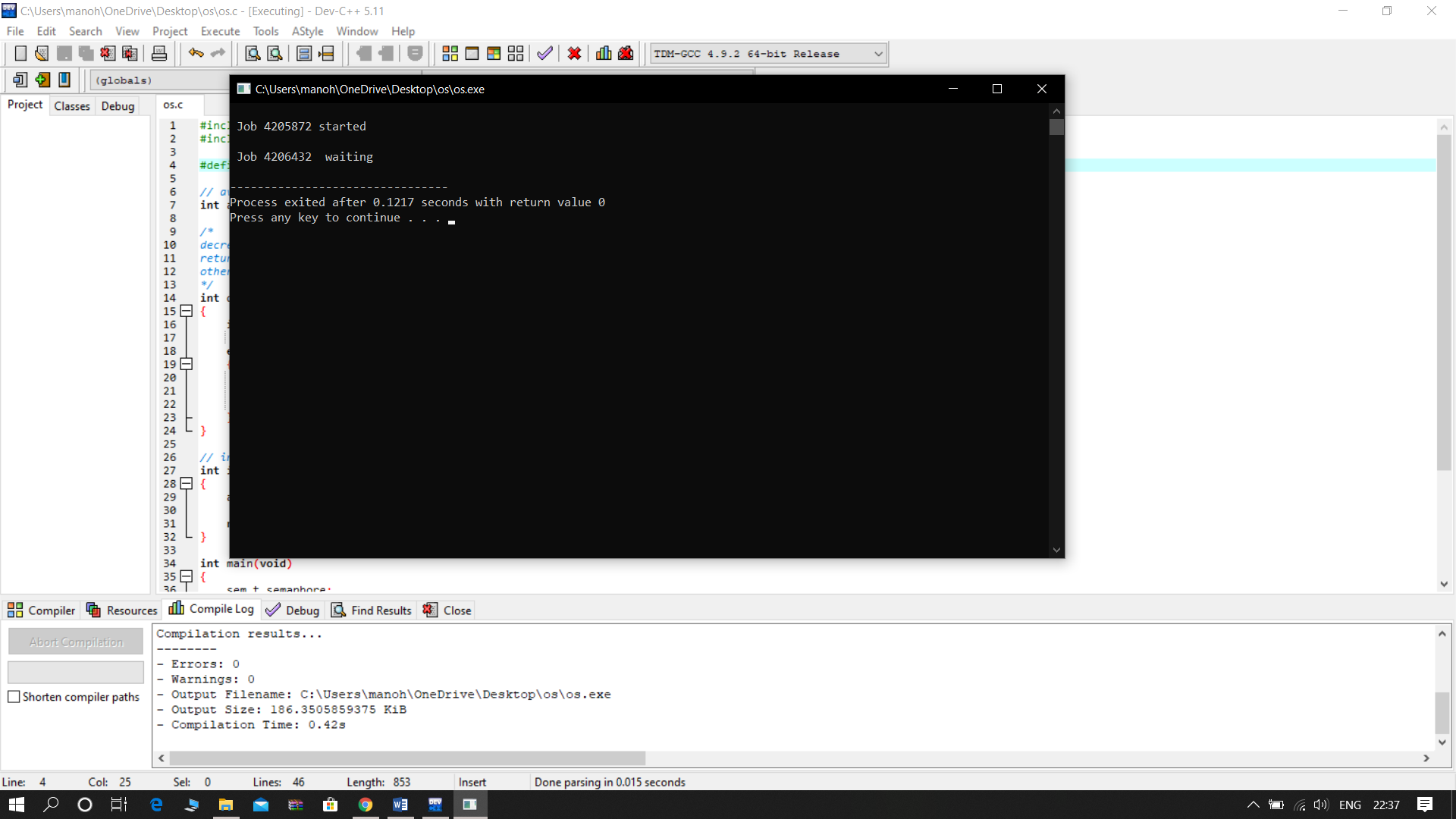
The increase count complexity is 1 and NLOC is 5 and the tokens are 14.

The total main function the complexity is 1 and NLOC is 9 and tokens are 51.

1. **Boundry conditions of the implemented code:**

* There are never any race conditions there's zero chance of deadlock; otherwise a subset of threads are forever starved Race conditions can generally be solved with mutexes.
* We use them to mark the boundaries of critical regions and limit the number of threads present within them to be at most one.
* The maximum value we have to take is 5only.
* In this we are setting particular value.
* In this it provides a queue of threads waiting for a resource.
* In the code I implemented only the functions.
* In the code the complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
* Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
* By using semaphore’s may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.
* By using the code there is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.

1. **Code snippet:**

****