

# CSE-316 OPERATING SYSTEMS

**SIMULATION PROJECT ASSIGNMENT QUESTION – 16**

**NAME : V.Charan Kumar**

**REGD. : 11709830**

**SECTION : K17TA**

**EMAIL :charankumarv18@gmail.com**

**GITHUB LINK:** <https://github.com/Charanvk/cse-316>

### 

## SUBMITTING TO:

**Amanpreet Kaur ma’am.**

**QUESTION 16**

**Question 16:** Suppose that a system initially has only a single thread (T1), which is running the function f1(). f1() calls pthread\_create to create a new thread, T2, to run function f2() concurrently with T1. These two threads and the functions they execute are illustrated in the two columns of the figure below. As illustrated in the figure, thread T1 calls function fb(), and thread T2 calls functions fa() and fc(). Your task is to synchronize the calls to fa(), fb(), and fc() so that the following synchronization rules are enforced: (1) function fa() finishes before function fb() is called, and (2) function fb() finishes before function fc() is called. Show how to use semaphore(s) to enforce these sychronization rules by adding P() and V() calls in suitable places in functions f1() and f2() in the diagram below. You must also declare (at the top of the diagram) global variables to point to any semphores you use in your solution. Finally, you must include sem create calls in suitable places in f1() and/or f2() to create and initialize the semaphores you need. Be sure that your sem create() calls show the intial semaphore value for each newly created semaphore. Do not use any synchronization primitives or techniques other than sempahores. Keep your solution as

simple as possible.

## Code:

#include<unistd.h>

#include<pthread.h>

#include<semaphore.h>

sem\_t mutex;

pthread\_t t1, t2;

void f2()

{printf("\nF2 called");

}

void f1()

{

printf("\nF1 called");

pthread\_create(&t2, NULL, f2, NULL);

}

void fa()

{

sem\_wait(&mutex);

printf("\nThread for fa Entered");

sleep(2);

printf("\nThread For fa Done");

sem\_post(&mutex);

}

void fb()

{

sem\_wait(&mutex);

printf("\nThread For fb Entered");

sleep(2);

printf("\nFunction fb Thread Done");

sem\_post(&mutex);

}

void fc()

{

sem\_wait(&mutex);

printf("\nThread for fc Entered ");

sleep(2);

printf("\nThread Function fc Done ");

sem\_post(&mutex);

}

int main()

{

sem\_init(&mutex, 5, 1);

pthread\_create(&t1, NULL, f1, NULL);

sleep(4);

pthread\_create(&t2, NULL, fa, NULL);

sleep(4);

pthread\_create(&t1, NULL, fb, NULL);

sleep(3);

pthread\_create(&t2, NULL, fc, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

printf("\nThreads Executed Successfully");

}

## DESCRIPTION

## 1 Explain the problem in terms of operating system concept?

|  |
| --- |
|  |
|  | In this question we should have a Knowledge of threads and Semphore in Operating System for Process Synchronization. |
|  | We are creating two threads in our main Program using pthread\_create function and then using semaphore functions to wait and signal function |
|  |  |
|  | Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment. |
|  | The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only. |
|  |  |
|  | \*\*\*P-and-V-operation-in-OS\*\*\* |
|  | P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation. |
|  | Both operations are atomic and semaphore(s) is always initialized to one. |
|  | A critical section is surrounded by both operations to implement process synchronization. |

**CONSTRAINTS**

The given constraints for this question are :

1. 1.First Constraintis that we have to solve this problem of Process Synchronization using only semaphores and By creating Threads.

|  |
| --- |
|  |
|  | 2.Only Two Threads should be created to call to the Functions that we have created. |
|  | 3. We have two functions f1 and f2 running concurrently with each other through calling of t1 and t2 threads. |
|  | 4.We are synchronizing the calls of functions fa, fb and fc through the synchronization rules given in the question. |
|  | Here Execution will be like fa Entered and done, then fb should be called and done, then fc should be called and done. |

## ****TIME COPLEXITY CODE:****

|  |
| --- |
|  |
|  | #include<stdio.h> |
|  | #include<pthread.h> |
|  | #include<semaphore.h> |
|  | sem\_t mutex; |
|  | pthread\_t t1, t2; |
|  | void f2() |
|  | { |
|  | printf("\nF2 called"); //TimeComplexity will be O(1) |
|  | } |
|  | void f1() |
|  | { |
|  | printf("\nF1 called"); //TimeComplexity will be O(1) |
|  | pthread\_create(&t2, NULL, f2, NULL); |
|  | } |
|  | void fa() |
|  | { |
|  | sem\_wait(&mutex); |
|  | printf("\nThread for fa Entered"); //TimeComplexity will be O(1) |
|  | sleep(2); |
|  | printf("\nThread For fa Done"); //TimeComplexity will be O(1) |
|  | sem\_post(&mutex); |
|  | } |
|  | void fb() |
|  | { |
|  | sem\_wait(&mutex); |
|  | printf("\nThread For fb Entered"); //TimeComplexity will be O(1) |
|  | sleep(2); |
|  | printf("\nFunction fb Thread Done"); //TimeComplexity will be O(1) |
|  | sem\_post(&mutex); |
|  | } |
|  | void fc() |
|  | { |
|  | sem\_wait(&mutex); |
|  | printf("\nThread for fc Entered "); //TimeComplexity will be O(1) |
|  | sleep(2); |
|  | printf("\nThread Function fc Done "); //TimeComplexity will be O(1) |
|  | sem\_post(&mutex); |
|  | } |
|  | int main() |
|  | { |
|  | sem\_init(&mutex, 3, 1); //Here we are passing 3 functions and initial value is 1 |
|  | pthread\_create(&t1, NULL, f1, NULL); //TimeComplexity will be O(1) |
|  | sleep(4); |
|  | pthread\_create(&t2, NULL, fa, NULL); //TimeComplexity will be O(1) |
|  | sleep(4); |
|  | pthread\_create(&t1, NULL, fb, NULL); //TimeComplexity will be O(1) |
|  | sleep(3); |
|  | pthread\_create(&t2, NULL, fc, NULL); //TimeComplexity will be O(1) |
|  | pthread\_join(t1, NULL); |
|  | pthread\_join(t2, NULL); |
|  | printf("\nThreads Executed Successfully"); //TimeComplexity will be O(1) |
|  | } |

**OUTPUT ON THE COMPILER**

