Lab Manual for Operating Systems

Lab-12 Process Synchronization

Table of Contents

Lab 12: Process Synchronization

Introduction

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes. Process Synchronization was introduced to handle problems that problem while multiple process executions.

Objective

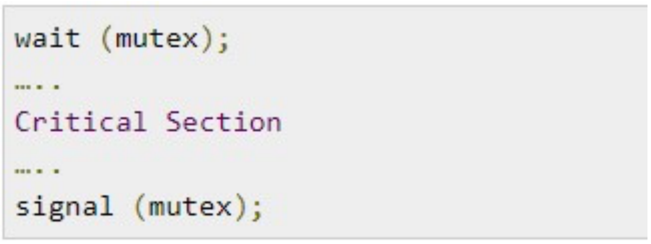
- To be familiar with process synchronization
- To be able to schedule processes
- To be able to implement process synchronization.
- To be able to differentiate between Mutex and Semaphores.
- To be able to perform different tasks with Mutex and Semaphores.

Concept Map

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes. Process Synchronization was introduced to handle problems that problem while multiple process executions. Mutex and Semaphore both provide synchronization services, but they are not the same.

Mutex

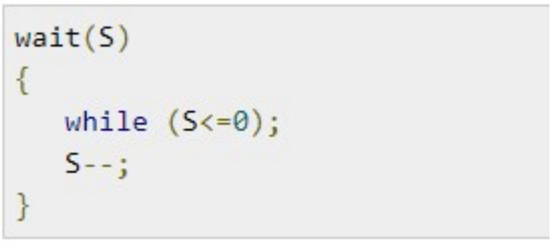
Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section. A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signaling mechanism. A binary semaphore can be used as a Mutex, but a Mutex can never be used as a semaphore.



```
wait (mutex);
…..
Critical Section
…..
signal (mutex);
```

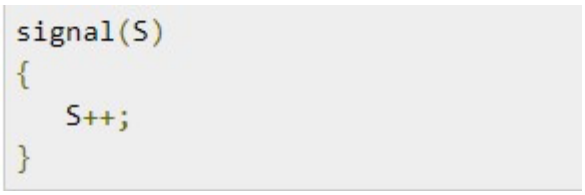
Semaphore

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function. A semaphore uses two atomic operations, wait and signal for process synchronization. The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.



```
wait(S)
{
   while (S<=0);
   S--;
}
```

The signal operation increments the value of its argument S.

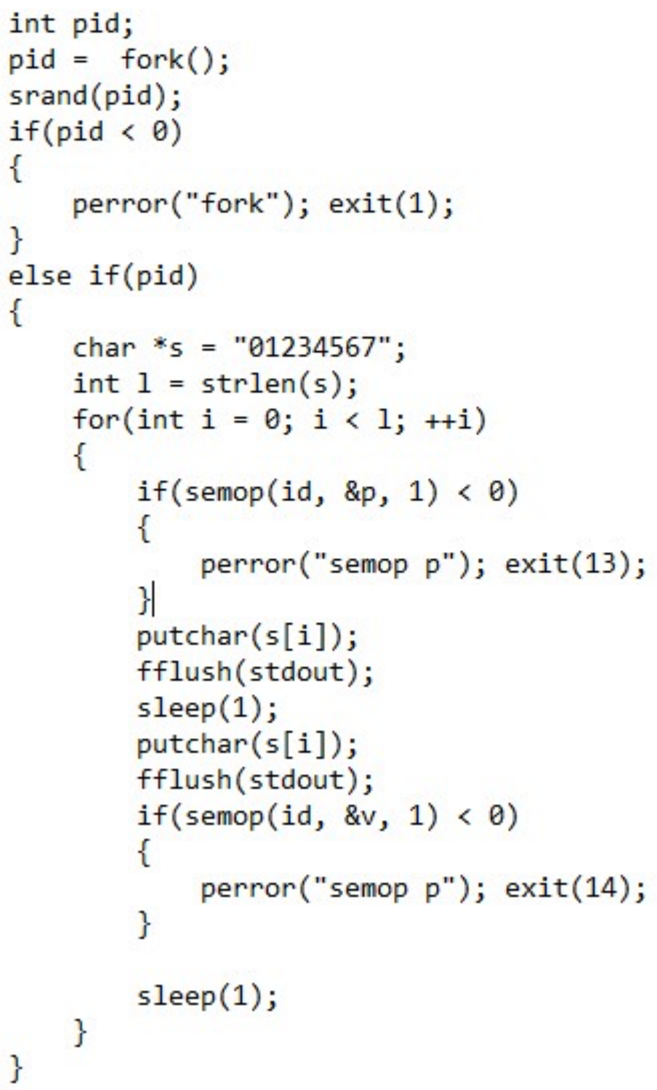


```
signal(S)
{
   S++;
}
```

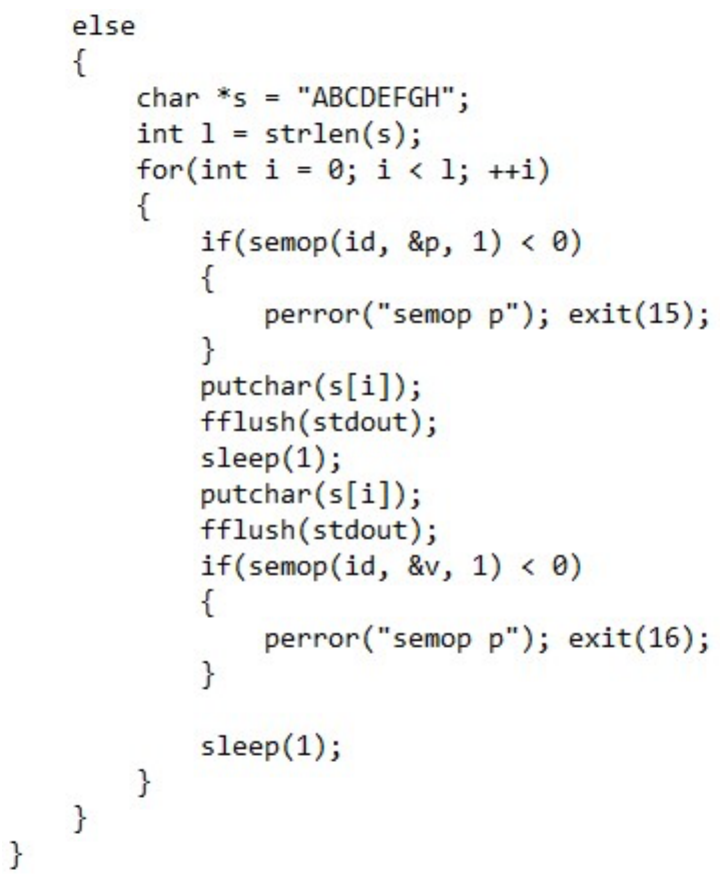
There are mainly two types of semaphores i.e., counting semaphores and binary semaphores. Counting Semaphores are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. The binary semaphores are like counting semaphores, but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.

Example code:

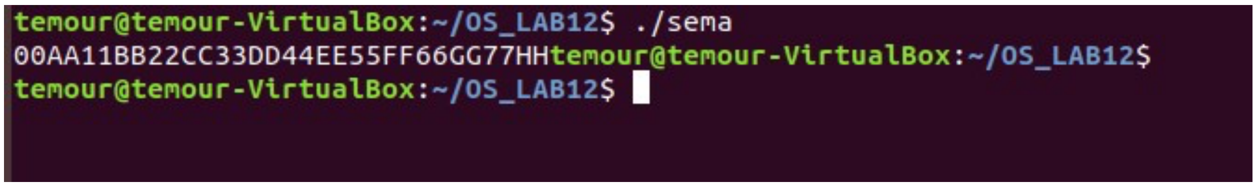
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};

struct sembuf p = { 0, -1, SEM_UNDO};
struct sembuf v = { 0, +1, SEM_UNDO};

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget"); exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {
        perror("semctl"); exit(12);
    }
```



```c
int pid;
pid =  fork();
srand(pid);
if(pid < 0)
{
    perror("fork"); exit(1);
}
else if(pid)
{
    char *s = "01234567";
    int l = strlen(s);
    for(int i = 0; i < l; ++i)
    {
        if(semop(id, &p, 1) < 0)
        {
            perror("semop p"); exit(13);
        }
        putchar(s[i]);
        fflush(stdout);
        sleep(1);
        putchar(s[i]);
        fflush(stdout);
        if(semop(id, &v, 1) < 0)
        {
            perror("semop p"); exit(14);
        }

        sleep(1);
    }
}
```



```
        else
        {
            char *s = "ABCDEFGH";
            int l = strlen(s);
            for(int i = 0; i < l; ++i)
            {
                if(semop(id, &p, 1) < 0)
                {
                    perror("semop p"); exit(15);
                }
                putchar(s[i]);
                fflush(stdout);
                sleep(1);
                putchar(s[i]);
                fflush(stdout);
                if(semop(id, &v, 1) < 0)
                {
                    perror("semop p"); exit(16);
                }

                sleep(1);
            }
        }
    }
```



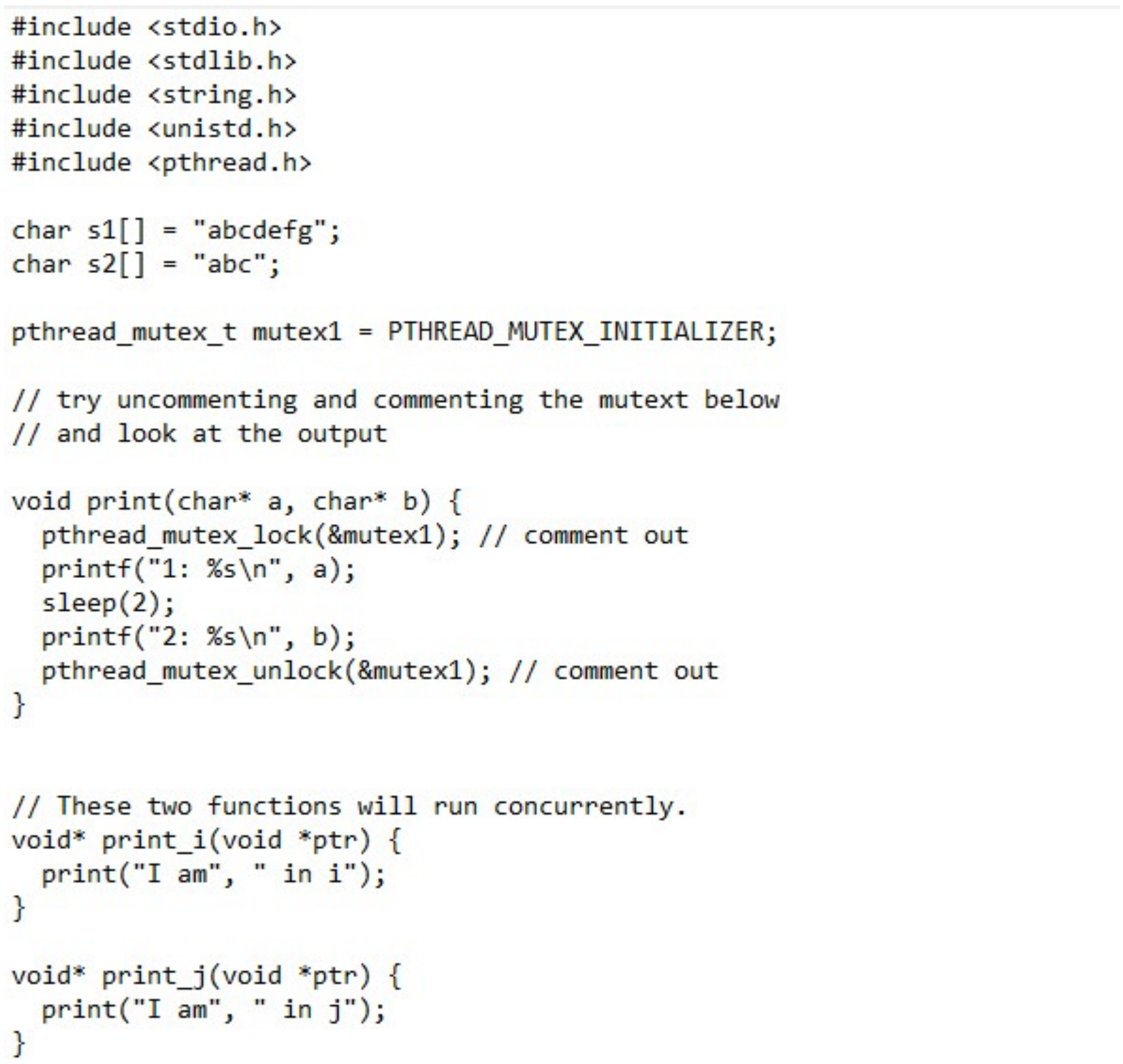
```
temour@temour-VirtualBox:~/OS_LAB12$ ./sema
00AA11BB22CC33DD44EE55FF66GG77HHtemour@temour-VirtualBox:~/OS_LAB12$
temour@temour-VirtualBox:~/OS_LAB12$ 
```

Walkthrough Tasks

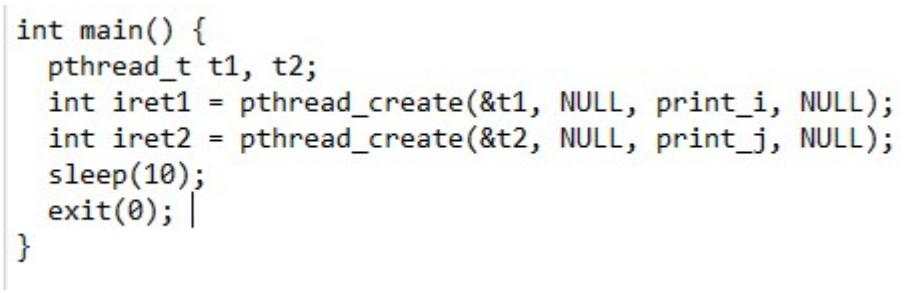
In this section you are given an example task which you have to implement by following the steps stated in this portion.

Task:

You have to implement mutex lock in your multithread processing program. You have to create worker thread and then apply mutex lock so that one thread can complete its processing before the second thread continues. Here one thread will wait for other thread to complete the working in which critical portion of code is involved and then it continues to proceed it's working.

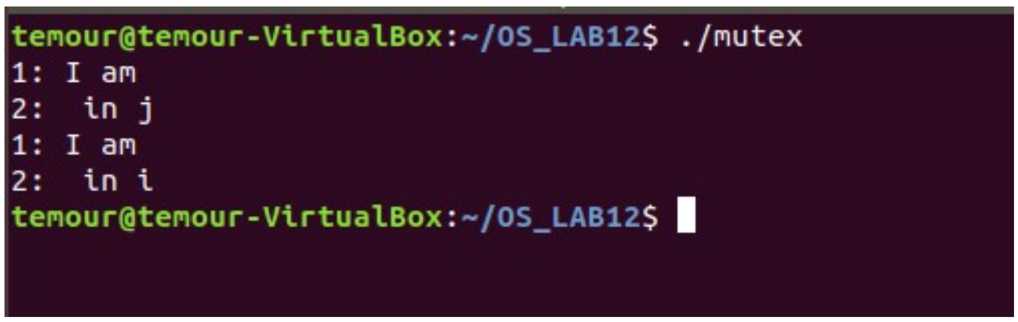


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

char s1[] = "abcdefg";
char s2[] = "abc";

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// try uncommenting and commenting the mutext below
// and look at the output

void print(char* a, char* b) {
  pthread_mutex_lock(&mutex1); // comment out
  printf("1: %s\n", a);
  sleep(2);
  printf("2: %s\n", b);
  pthread_mutex_unlock(&mutex1); // comment out
}


// These two functions will run concurrently.
void* print_i(void *ptr) {
  print("I am", " in i");
}

void* print_j(void *ptr) {
  print("I am", " in j");
}
```



```c
int main() {
  pthread_t t1, t2;
  int iret1 = pthread_create(&t1, NULL, print_i, NULL);
  int iret2 = pthread_create(&t2, NULL, print_j, NULL);
  sleep(10);
  exit(0);
}
```

Output



```
temour@temour-VirtualBox:~/OS_LAB12$ ./mutex
1: I am
2:  in j
1: I am
2:  in i
temour@temour-VirtualBox:~/OS_LAB12$ 
```

Practice Tasks

This section will provide more practice exercises which you need to finish during the lab. You need to finish the tasks in the required time. When you finish them, put these tasks in your GitHub Account in the name of `Lab12`.

Practice Task 1 [Expected time =15mins]

Implement the above-mentioned semaphore techniques code and take screenshots of output. Submit code file along with output screenshot.

Evaluation Task (Unseen) **[Expected time = 30mins]**

The lab instructor will give you unseen task depending upon the progress of the class.

Evaluation criteria

The evaluation criteria for this lab will be based on the completion of the following tasks. Each task is assigned the marks percentage which will be evaluated by the instructor in the lab whether the student has finished the complete/partial task(s).

Table 3: Evaluation of the Lab

Sr. No.	**Task No**	**Description**	**Marks**
1	1	Problem Modeling	20
2	2	Procedures and Tools	10
3	3	Practice tasks and Testing	35
4	4	Evaluation Tasks (Unseen)	20
5		Comments	5
6		Good Programming Practices	10

Further Reading

This section provides the references to further polish your skills.

Slides

Chapter_6 Slides already been shared.