Ex. No. 6	Mutual Exclusion-Semaphore and Reader Writer Solution	Date :

Semaphore

Semaphore is used to implement process synchronization. This is to protect critical region shared among multiples processes.

SYSTEM V SEMAPHORE SYSTEM CALLS

Include the following header files for System V semaphore

```
<sys/ipc.h>,  <sys/sem.h>, <sys/types.h>
```

To create a semaphore array,

```
int semget(key_t key, int nsems, int semflg)
```

key ➔ semaphore id
nsems ➔ no. of semaphores in the semaphore array
semflg ➔ `IPC_CREATE|0664` : to create a new semaphore
`IPC_EXCL|IPC_CREAT|0664` : to create new semaphore and the call fails if the semaphore already exists

To perform operations on the semaphore sets viz., allocating resources, waiting for the resources or freeing the resources,

```
int semop(int semid, struct sembuf *semops, size_t nsemops)
```

semid ➔ semaphore id returned by semget()
nsemops ➔ the number of operations in that array
semops ➔ The pointer to an array of operations to be performed on the semaphore set. The structure is as follows

```
struct sembuf {
        unsigned short sem_num;   /* Semaphore set num */
        short sem_op;             /* Semaphore operation */
        short sem_flg;  /*Operation flags, IPC_NOWAIT, SEM_UNDO */
};
```

Element, sem_op, in the above structure, indicates the operation that needs to be performed –
If sem_op is –ve, allocate or obtain resources. Blocks the calling process until enough resources have been freed by other processes, so that this process can allocate.
If sem_op is zero, the calling process waits or sleeps until semaphore value reaches 0.
If sem_op is +ve, release resources.

To perform control operation on semaphore,

```
int semctl(int semid, int semnum, int cmd,...);
```

semid ➔ identifier of the semaphore returned by semget()
semnum ➔ semaphore number
cmd ➔ the command to perform on the semaphore. Ex. GETVAL, SETVAL
semun ➔ value depends on the cmd. For few cases, this is not applicable.

Q1. Execute and write the output of the following program for *mutual exclusion* using system V semaphore

```
#include<sys/ipc.h>
#include<sys/sem.h>
int main()
{
 int pid,semid,val;
 struct sembuf sop;

 semid=semget((key_t)6,1,IPC_CREAT|0666);

 pid=fork();

 sop.sem_num=0;
 sop.sem_op=0;
 sop.sem_flg=0;

 if (pid!=0)
 {
   sleep(1);
   printf("The Parent waits for WAIT signal\n");
   semop(semid,&sop,1);
   printf("The Parent WAKED UP & doing her job\n");
   sleep(10);
   printf("Parent Over\n");
 }
 else
 {
   printf("The Child sets WAIT signal & doing her job\n");
   semctl(semid,0,SETVAL,1);
   sleep(10);
   printf("The Child sets WAKE signal & finished her job\n");
   semctl(semid,0,SETVAL,0);
   printf("Child Over\n");
 }
 return 0;
}
```

Output :

POSIX SEMAPHORE

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to include `semaphore.h` and compile the code by linking with `-lpthread - lrt`

To lock a semaphore or wait

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore

```
int sem_post(sem_t *sem);
```

To initialize a semaphore

```
sem_init(sem_t *sem, int pshared, unsigned int
value);
```

Where,

sem : Specifies the semaphore to be initialized.

pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes/threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.

value : Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore

```
sem_destroy(sem_t *mutex);
```

Q2. Program creates two threads: one to increment the value of a shared variable and second to decrement the value of the shared variable. Both the threads make use of semaphore variable so that only one of the threads is executing in its critical section. Execute and write the output.

```
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
 void *fun2();
 int shared=1; //shared variable
 sem_t s; //semaphore variable
 int main()
 {
 sem_init(&s,0,1); //initialize semaphore variable - 1st argument is
//address of variable, 2nd is number of processes sharing semaphore,
//3rd argument is the initial value of semaphore variable
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 sleep(1);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
```

```
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last
//updated value of shared variable
 }
void *fun1()
{
    int x;
    sem_wait(&s); //executes wait operation on s
    x=shared;//thread1 reads value of shared variable
    printf("Thread1 reads the value as %d\n",x);
    x++;  //thread1 increments its value
    printf("Local updation by Thread1: %d\n",x);
    sleep(1); //thread1 is preempted by thread 2
    shared=x; //thread one updates the value of shared variable
    printf("Value of shared variable updated by Thread1 is:
%d\n",shared);
    sem_post(&s);
}
void *fun2()
{
    int y;
    sem_wait(&s);
    y=shared;//thread2 reads value of shared variable
    printf("Thread2 reads the value as %d\n",y);
    y--;  //thread2 increments its value
    printf("Local updation by Thread2: %d\n",y);
    sleep(1); //thread2 is preempted by thread 1
    shared=y; //thread2 updates the value of shared variable
    printf("Value of shared variable updated by Thread2 is:
%d\n",shared);
    sem_post(&s);
}
```

The final value of the variable *shared* will be 1. When any one of the threads executes the wait operation the value of "s" becomes zero. Hence the other thread (even if it preempts the running thread) is not able to successfully execute the wait operation on "*s*". Thus, not able to read the inconsistent value of the shared variable. This ensures that only one of the threads is running in its critical section at any given time.

Output:

Reader Writer Solution

In a reader-writer problem, multiple readers can read data from a shared resource, while only one writer can write data to the resource. The challenge is to ensure that the readers do not read data while the writer is writing, and the writer does not write data while the readers are reading.

For example, consider a scenario where a database is being accessed by multiple users. The users can read data from the database, but only one user can write data to the database at a time. The challenge is to ensure that the users do not read data while the database is being written to, and the writer does not write data while the users are reading.

Q3.

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

/*
This program provides a possible solution for first readers writers
problem using mutex and semaphore.
I have used 10 readers and 5 producers to demonstrate the solution.
You can always play with these values.
*/

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 cnt to %d\n",(*((int *)wno)),cnt);
    sem_post(&wrt);

}
void *reader(void *rno)
{
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader++;
    if(numreader == 1) {
        sem_wait(&wrt); // If this id the first reader, then it will
block the writer
    }
    pthread_mutex_unlock(&mutex);
    // Reading Section
    printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);

    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
```

```
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt); // If this is the last reader, it will wake
up the writer.
    }
    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}; //Just used for numbering
the producer and consumer

    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;

}
```

Output:

Verified by

Faculty In-charge Sign :	**Date :**