OPERATING SYSTEMS LAB - PRACTICAL 7 - SEMAPHORES

Name - Sakshi Soni
Roll No - 13

AIM -

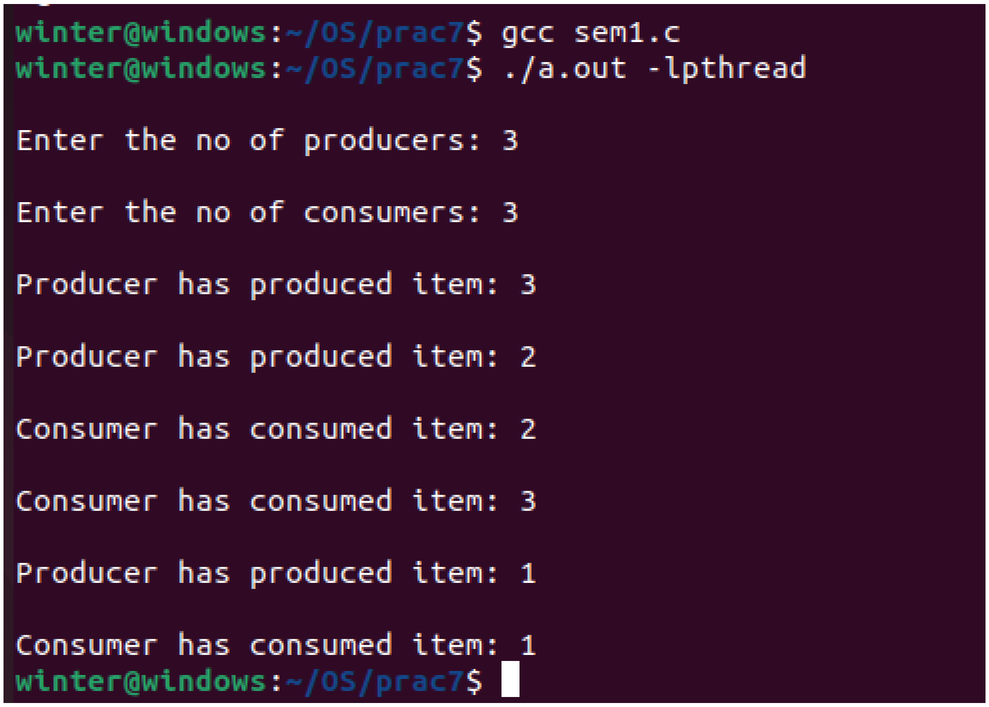
Write C programs to implement threads and semaphores for process synchronization

PROGRAM AND OUTPUT -

Program 1- n Producers and n Consumers with synchronization

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

#define buffersize 10

pthread_mutex_t mutex;
pthread_t tidP[20], tidC[20];
sem_t full,empty;
int counter;
int buffer[buffersize];

void initialize()
{
        pthread_mutex_init(&mutex, NULL);
        sem_init(&full,1,0);
        sem_init(&empty,1,buffersize);
        counter=0;
}

void write(int item)
```

```
{
        buffer[counter++]=item;
}

int read()
{
        return(buffer[--counter]);
}

void * producer (void * param)
{
        int waittime, item, i;
        item=rand()%5;
        waittime=rand()%5;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        printf("\nProducer has produced item: %d\n",item);
        write(item);
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
}

void * consumer (void * param)
{
        Int waittime,item;
        waittime=rand()%5;
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        item=read();
        printf("\nConsumer has consumed item: %d\n",item);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
}


int main()
{
        int n1,n2,i;
        initialize();
        printf("\nEnter the no of producers: ");
        scanf("%d",&n1);
        printf("\nEnter the no of consumers: ");
        scanf("%d",&n2);
        for(i=0;i<n1;i++)
                pthread_create(&tidP[i],NULL,producer,NULL);
        for(i=0;i<n2;i++)
                pthread_create(&tidC[i],NULL,consumer,NULL);
        for(i=0;i<n1;i++)
                pthread_join(tidP[i],NULL);
        for(i=0;i<n2;i++)
                pthread_join(tidC[i],NULL);

        //sleep(5);
        exit(0);
}
```

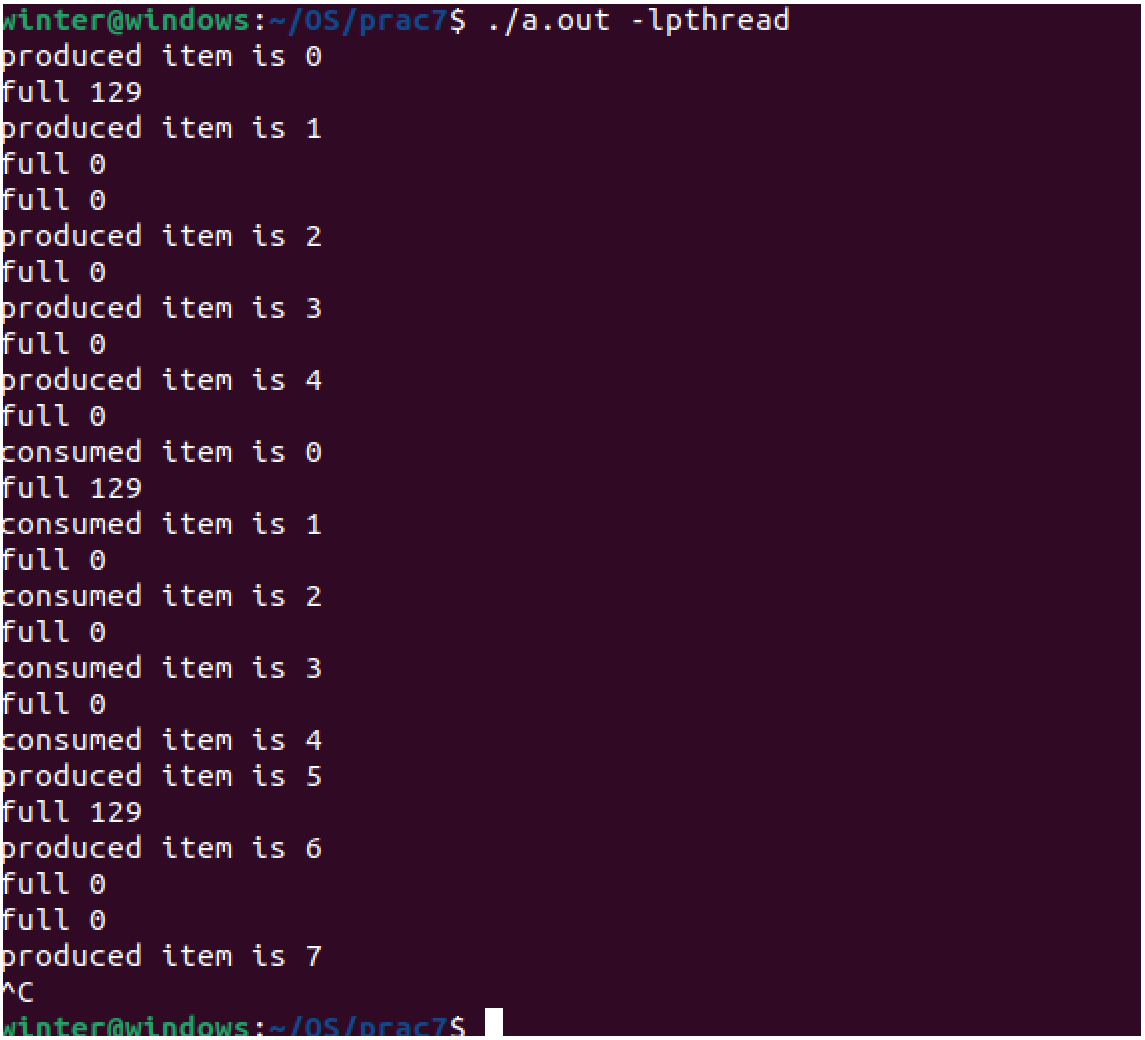


```
winter@windows:~/OS/prac7$ gcc sem1.c
winter@windows:~/OS/prac7$ ./a.out -lpthread

Enter the no of producers: 3

Enter the no of consumers: 3

Producer has produced item: 3

Producer has produced item: 2

Consumer has consumed item: 2

Consumer has consumed item: 3

Producer has produced item: 1

Consumer has consumed item: 1
winter@windows:~/OS/prac7$
```

Program 2- 1 Producers and 1 Consumer with synchronization

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
intbuf[5],f,r;
sem_t mutex,full,empty;
```

```
void *produce(void *arg)
{
int i;
for(i=0;i<10;i++)
   {
sem_wait(&empty);
sem_wait(&mutex);
printf("produced item is %d\n",i);
buf[(++r)%5]=i;
sleep(1);
sem_post(&mutex);
sem_post(&full);
printf("full %u\n",full);
   }
}
void *consume(void *arg)
{
Int item,i;
for(i=0;i<10;i++)
      {
sem_wait(&full);
printf("full %u\n",full);
sem_wait(&mutex);
item=buf[(++f)%5];
printf("consumed item is %d\n",item);
sleep(1);
sem_post(&mutex);
sem_post(&empty);
      }
}
main()
{
pthread_t tid1,tid2;
sem_init(&mutex,0,1);
sem_init(&full,0,0);
sem_init(&empty,0,5);
pthread_create(&tid1,NULL,produce,NULL);
pthread_create(&tid2,NULL,consume,NULL);
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
}
```

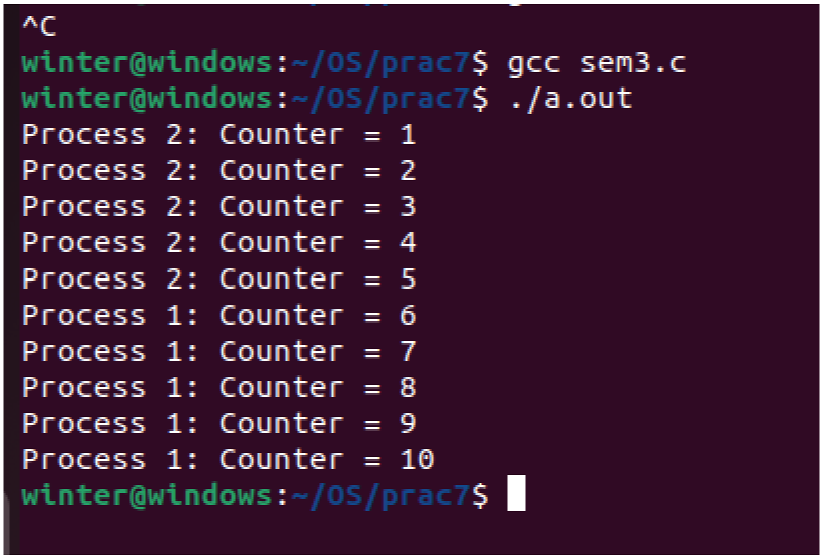


```
winter@windows:~/OS/prac7$ ./a.out -lpthread
produced item is 0
full 129
produced item is 1
full 0
full 0
produced item is 2
full 0
produced item is 3
full 0
produced item is 4
full 0
consumed item is 0
full 129
consumed item is 1
full 0
consumed item is 2
full 0
consumed item is 3
full 0
consumed item is 4
produced item is 5
full 129
produced item is 6
full 0
full 0
produced item is 7
^C
winter@windows:~/OS/prac7$
```

Program 3-
Write a program to create an integer variable using a shared memory concept and increment the variable simultaneously by two processes. Use semaphores to avoid race conditions.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
```

```
#define SHM_SIZE  sizeof(int)  // Size of shared memory segment

int main() {
    int shmid;
    int *counter;

    // Create shared memory segment
    shmid = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory segment
    counter = (int *)shmat(shmid, NULL, 0);
    if (counter == (int *)-1) {
        perror("shmat");
        exit(1);
    }

    // Initialize counter
    *counter = 0;

    // Create semaphore
    sem_t *sem = sem_open("/my_semaphore", O_CREAT, 0666, 1);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }

    // Fork a child process
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {
        // Child process
        for (int i = 0; i < 5; i++) {
            sem_wait(sem);  // Acquire the semaphore
            (*counter)++;  // Increment the counter
            printf("Process 1: Counter = %d\n", *counter);
            sem_post(sem);  // Release the semaphore
        }

        // Detach shared memory segment
        if (shmdt(counter) == -1) {
            perror("shmdt");
            exit(1);
        }

        exit(0);
    } else {
        // Parent process
        for (int i = 0; i < 5; i++) {
            sem_wait(sem);  // Acquire the semaphore
            (*counter)++;  // Increment the counter
            printf("Process 2: Counter = %d\n", *counter);
            sem_post(sem);  // Release the semaphore
        }

        // Wait for the child process to complete
        wait(NULL);

        // Detach shared memory segment
        if (shmdt(counter) == -1) {
            perror("shmdt");
            exit(1);
        }

        // Destroy the semaphore
        sem_close(sem);
        sem_unlink("/my_semaphore");

        // Delete shared memory segment
        if (shmctl(shmid, IPC_RMID, 0) == -1) {
            perror("shmctl");
            exit(1);
        }
    }

    return 0;
}
```



```
^C
winter@windows:~/OS/prac7$ gcc sem3.c
winter@windows:~/OS/prac7$ ./a.out
Process 2: Counter = 1
Process 2: Counter = 2
Process 2: Counter = 3
Process 2: Counter = 4
Process 2: Counter = 5
Process 1: Counter = 6
Process 1: Counter = 7
Process 1: Counter = 8
Process 1: Counter = 9
Process 1: Counter = 10
winter@windows:~/OS/prac7$
```

Program 4-
Producer Consumer with semaphores and shared memory.

problem.h

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <fcntl.h>

#define BUFFER_SIZE 10
#define CONSUMER_SLEEP_SEC 3
#define PRODUCER_SLEEP_SEC 1
#define KEY 1010

// A structure to store BUFER and semaphores for synchronization
typedefstruct
{
    int buff[BUFFER_SIZE];
    sem_tmutex, empty, full;

} MEM;

// Method for shared memory allocation
MEM *memory()
{
    key_t key = KEY;
    int shmid;
    shmid = shmget(key, sizeof(MEM), IPC_CREAT | 0666);
    return (MEM *) shmat(shmid, NULL, 0);
}

voidinit()
{
    // Initialize structure pointer with shared memory
    MEM *M = memory();

    // Initialize semaphores
    sem_init(&M->mutex,1,1);
    sem_init(&M->empty,1,BUFFER_SIZE);
    sem_init(&M->full,1,0);
}
```

producer.c

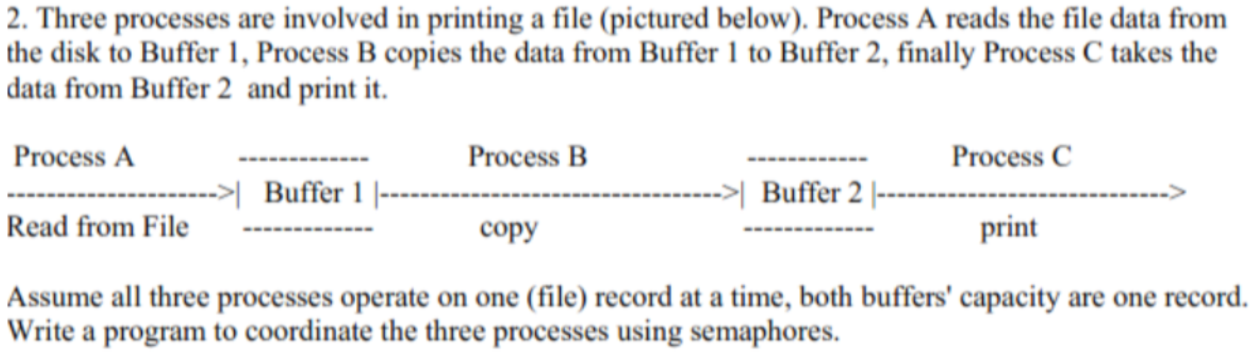
```
#include "problem.h"

void producer()
{
    int i=0,n;
    MEM *S = memory();

    while(1)
    {
        i++;
        sem_wait(&S->empty); // Semaphore down operation
        sem_wait(&S->mutex);
        sem_getvalue(&S->full,&n);
        (S->buff)[n] = i; // Place value to BUFFER
        printf("[PRODUCER] Placed item [%d]\n", i);
        sem_post(&S->mutex);
        sem_post(&S->full); // Semaphore up operation
        sleep(PRODUCER_SLEEP_SEC);

    }
}

main()
{
    init();
    producer();

}
```

Consumer.c

```
#include "problem.h"

void consumer()
{
    int n;
    MEM *S = memory();

    while(1)
    {
        sem_wait(&S->full); // Semaphore down operation
        sem_wait(&S->mutex); // Semaphore for mutual exclusion
        sem_getvalue(&S->full,&n); // Assign value of semphore full, to integer n
        printf("[CONSUMER] Removed item [%d]\n", (S->buff)[n]);
        sem_post(&S->mutex); // Mutex up operation
        sem_post(&S->empty); // Semaphore up operation
        sleep(CONSUMER_SLEEP_SEC);

    }
}

main()
```

```
{
        consumer();
}
```

Program 5-
Implement the C program for the processes given below using semaphores and system calls required.



2. Three processes are involved in printing a file (pictured below). Process A reads the file data from the disk to Buffer 1, Process B copies the data from Buffer 1 to Buffer 2, finally Process C takes the data from Buffer 2 and print it.

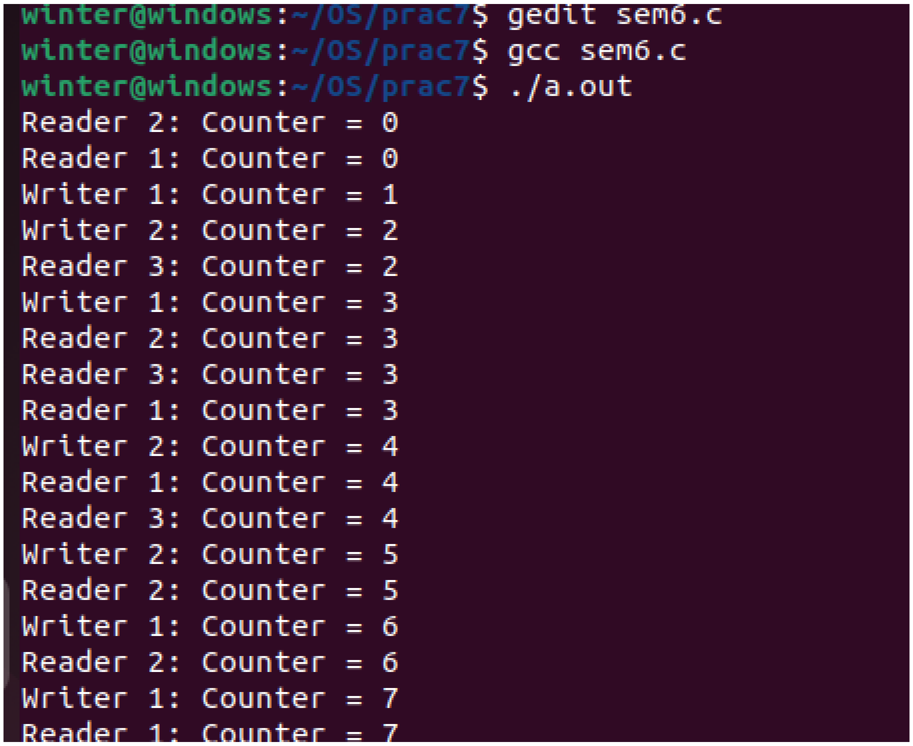
```
 Process A            -------------          Process B                ------------        Process C
--------------------->|  Buffer 1 |---------------------------------->|  Buffer 2 |----------------------------->
Read from File        -------------           copy                   -------------          print
```

Assume all three processes operate on one (file) record at a time, both buffers' capacity are one record. Write a program to coordinate the three processes using semaphores.

Program 6- Readers Writers with semaphores and shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

#define SHM_SIZE  sizeof(int)  // Size of shared memory segment

// Shared data structure
typedef struct {
   int counter;
   int readers_count;
   sem_t mutex;
   sem_t wrt;
} SharedData;
```

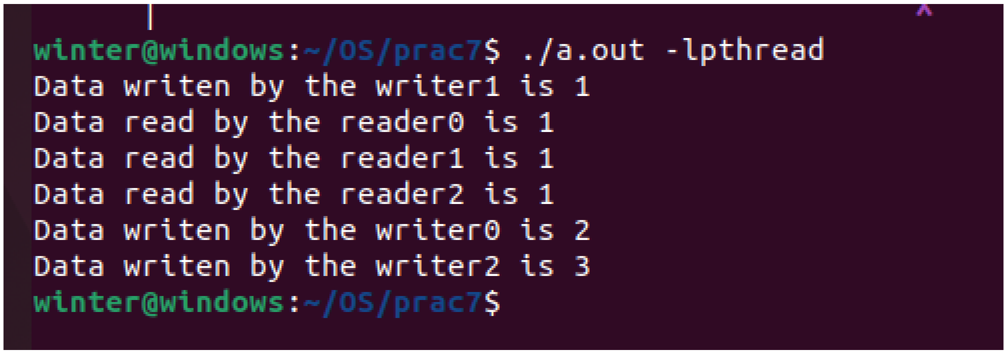
```
void reader(int id, SharedData* data) {
    while (1) {
        // Reader entry section
        sem_wait(&data->mutex);
        data->readers_count++;
        if (data->readers_count == 1) {
            sem_wait(&data->wrt);
        }
        sem_post(&data->mutex);

        // Critical section (reading)
        printf("Reader %d: Counter = %d\n", id, data->counter);

        // Reader exit section
        sem_wait(&data->mutex);
        data->readers_count--;
        if (data->readers_count == 0) {
            sem_post(&data->wrt);
        }
        sem_post(&data->mutex);

        // Random delay before next read
        usleep(rand() % 1000000);
    }
}

void writer(int id, SharedData* data) {
    while (1) {
        // Writer entry section
        sem_wait(&data->wrt);

        // Critical section (writing)
        data->counter++;
        printf("Writer %d: Counter = %d\n", id, data->counter);

        // Writer exit section
        sem_post(&data->wrt);

        // Random delay before next write
        usleep(rand() % 1000000);
    }
}

int main() {
    int shmid;
    SharedData* sharedData;

    // Create shared memory segment
    shmid = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory segment
    sharedData = (SharedData*)shmat(shmid, NULL, 0);
    if (sharedData == (SharedData*)-1) {
        perror("shmat");
        exit(1);
    }

    // Initialize shared data
    sharedData->counter = 0;
    sharedData->readers_count = 0;
    sem_init(&sharedData->mutex, 1, 1);
    sem_init(&sharedData->wrt, 1, 1);

    // Fork multiple reader and writer processes
    int numReaders = 3;
    int numWriters = 2;

    for (int i = 0; i < numReaders + numWriters; i++) {
        pid_t pid = fork();

        if (pid == -1) {
            perror("fork");
            exit(1);
        }
        if (pid == 0) {
            // Child process (reader or writer)
            if (i < numReaders) {
                reader(i + 1, sharedData);
            } else {
                writer(i - numReaders + 1, sharedData);
            }

            // Detach shared memory segment
            if (shmdt(sharedData) == -1) {
                perror("shmdt");
                exit(1);
            }

            exit(0);
        }
    }

    // Wait for all child processes to complete
    for (int i = 0; i < numReaders + numWriters; i++) {
        wait(NULL);
    }

    // Destroy semaphores
    sem_destroy(&sharedData->mutex);
    sem_destroy(&sharedData->wrt);

    // Delete shared memory segment
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        exit(1);
    }

    return 0;
}
```



```
winter@windows:~/OS/prac7$ gedit sem6.c
winter@windows:~/OS/prac7$ gcc sem6.c
winter@windows:~/OS/prac7$ ./a.out
Reader 2: Counter = 0
Reader 1: Counter = 0
Writer 1: Counter = 1
Writer 2: Counter = 2
Reader 3: Counter = 2
Writer 1: Counter = 3
Reader 2: Counter = 3
Reader 3: Counter = 3
Reader 1: Counter = 3
Writer 2: Counter = 4
Reader 1: Counter = 4
Reader 3: Counter = 4
Writer 2: Counter = 5
Reader 2: Counter = 5
Writer 1: Counter = 6
Reader 2: Counter = 6
Writer 1: Counter = 7
Reader 1: Counter = 7
```

Program 7- Readers Writers with semaphores and pthread.

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

 sem_tmutex,writeblock;
 int data = 0,rcount = 0;

 void *reader(void *arg)
 {
   int f;
   f = ((int)arg);
   sem_wait(&mutex);
   rcount = rcount + 1;
   if(rcount==1)
    sem_wait(&writeblock);
   sem_post(&mutex);
   printf("Data read by the reader%d is %d\n",f,data);
   sleep(1);
   sem_wait(&mutex);
 rcount = rcount - 1;
 if(rcount==0)
  sem_post(&writeblock);
 sem_post(&mutex);
}

void *writer(void *arg)
{
 int f;
 f = ((int) arg);
 sem_wait(&writeblock);
 data++;
 printf("Data writen by the writer%d is %d\n",f,data);
 sleep(1);
 sem_post(&writeblock);
}

main()
{
 inti,b;
 pthread_trtid[5],wtid[5];
 sem_init(&mutex,0,1);
 sem_init(&writeblock,0,1);
 for(i=0;i<=2;i++)
 {
  pthread_create(&wtid[i],NULL,writer,(void *)i);
  pthread_create(&rtid[i],NULL,reader,(void *)i);
 }
 for(i=0;i<=2;i++)
 {
       pthread_join(wtid[i],NULL);
       pthread_join(rtid[i],NULL);
 }
}
```



```
winter@windows:~/OS/prac7$ ./a.out -lpthread
Data writen by the writer1 is 1
Data read by the reader0 is 1
Data read by the reader1 is 1
Data read by the reader2 is 1
Data writen by the writer0 is 2
Data writen by the writer2 is 3
winter@windows:~/OS/prac7$
```

Program 8-

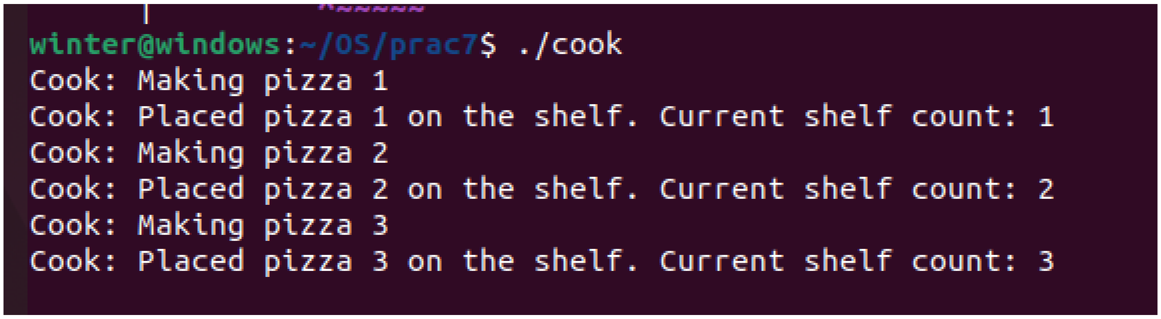
The cook cooks pizza and puts that pizza on the shelf. The waiter picks pizza from the shelf and serves it to customers. The shelf can hold three pizzas at most at the same time. When the shelf is full, cook and wait until pick up; when there is no pizza on the shelf, the waiter waits until made.

Cook.c

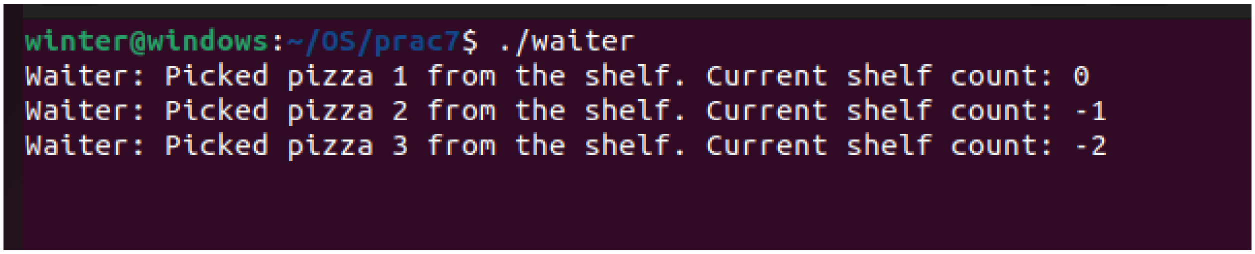
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>

#define SHM_SIZE  sizeof(int)  // Size of shared memory segment
#define MAX_PIZZAS 3  // Maximum number of pizzas on the shelf

int main() {
    int shmid;
    int *shelf;

    // Create shared memory segment
    shmid = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory segment
    shelf = (int *)shmat(shmid, NULL, 0);
    if (shelf == (int *)-1) {
        perror("shmat");
        exit(1);
    }

    // Initialize shared memory
    *shelf = 0;

    // Open semaphores
    sem_t *mutex = sem_open("/mutex", O_CREAT, 0666, 1);
    if (mutex == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }

    sem_t *fill = sem_open("/fill", O_CREAT, 0666, 0);
    if (fill == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }

    // Cook pizzas
    int pizzaCount = 1;
    while (1) {
        sem_wait(mutex);  // Acquire mutex semaphore

        if (*shelf < MAX_PIZZAS) {
            printf("Cook: Making pizza %d\n", pizzaCount);

            // Place pizza on the shelf
            (*shelf)++;
            printf("Cook: Placed pizza %d on the shelf. Current shelf count: %d\n",
pizzaCount, *shelf);

            pizzaCount++;

            sem_post(fill);  // Post fill semaphore
        }

        sem_post(mutex);  // Release mutex semaphore
        // Random delay before cooking the next pizza
        usleep(rand() % 2000000);
    }

    // Detach shared memory segment
    if (shmdt(shelf) == -1) {
        perror("shmdt");
        exit(1);
    }

    // Close and unlink semaphores
    sem_close(mutex);
    sem_close(fill);
    sem_unlink("/mutex");
    sem_unlink("/fill");

    // Delete shared memory segment
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        exit(1);
    }

    return 0;
}
```

Waiter.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>

#define SHM_SIZE  sizeof(int)  // Size of shared memory segment
#define MAX_PIZZAS 3  // Maximum number of pizzas on the shelf

int main() {
    int shmid;
    int *shelf;
    // Create shared memory segment
    shmid = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach shared memory segment
    shelf = (int *)shmat(shmid, NULL, 0);
    if (shelf == (int *)-1) {
        perror("shmat");
        exit(1);
    }

    // Open semaphores
    sem_t *mutex = sem_open("/mutex", O_CREAT, 0666, 1);
    if (mutex == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }

    sem_t *fill = sem_open("/fill", O_CREAT, 0666, 0);
    if (fill == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }

    // Serve pizzas
    int pizzaCount = 1;
    while (1) {
        sem_wait(fill);  // Wait for pizza to be available on the shelf

        sem_wait(mutex);  // Acquire mutex semaphore

        printf("Waiter: Picked pizza %d from the shelf. Current shelf count: %d\n",
pizzaCount, *shelf);

        // Serve the pizza
        (*shelf)--;
        pizzaCount++;

        sem_post(mutex);  // Release mutex semaphore

        // Random delay before serving the next pizza
        usleep(rand() % 2000000);
    }

    // Detach shared memory segment
    if (shmdt(shelf) == -1) {
        perror("shmdt");
        exit(1);
    }

    // Close and unlink semaphores
    sem_close(mutex);
    sem_close(fill);
    sem_unlink("/mutex");
    sem_unlink("/fill");

    // Delete shared memory segment
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        exit(1);
    }

    return 0;
}
```



```
winter@windows:~/OS/prac7$ ./cook
Cook: Making pizza 1
Cook: Placed pizza 1 on the shelf. Current shelf count: 1
Cook: Making pizza 2
Cook: Placed pizza 2 on the shelf. Current shelf count: 2
Cook: Making pizza 3
Cook: Placed pizza 3 on the shelf. Current shelf count: 3
```



```
winter@windows:~/OS/prac7$ ./waiter
Waiter: Picked pizza 1 from the shelf. Current shelf count: 0
Waiter: Picked pizza 2 from the shelf. Current shelf count: -1
Waiter: Picked pizza 3 from the shelf. Current shelf count: -2
```

RESULT -

Linux C programs to demonstrate the concept of threads and semaphores for process synchronization have been implemented.