Name:Mohammad Abubakr Khanooni
Cse a roll no 27
Practical 7B operating system

Aim:
Write C programs to implement threads and semaphores for process synchronisation.

Prac 7B-semaphores:

1.C PROGRAM FOR PRODUCER CONSUMER PROBLEM with synchronisation using semaphores for n producer and n consumer)

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.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 intialize() {
  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;
  intialize();
  printf("Enter the no of Producers : \n");
  scanf("%d", &n1);
  printf("\nEnter the no of Consumers : \n");
  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);
}
```

o/p:

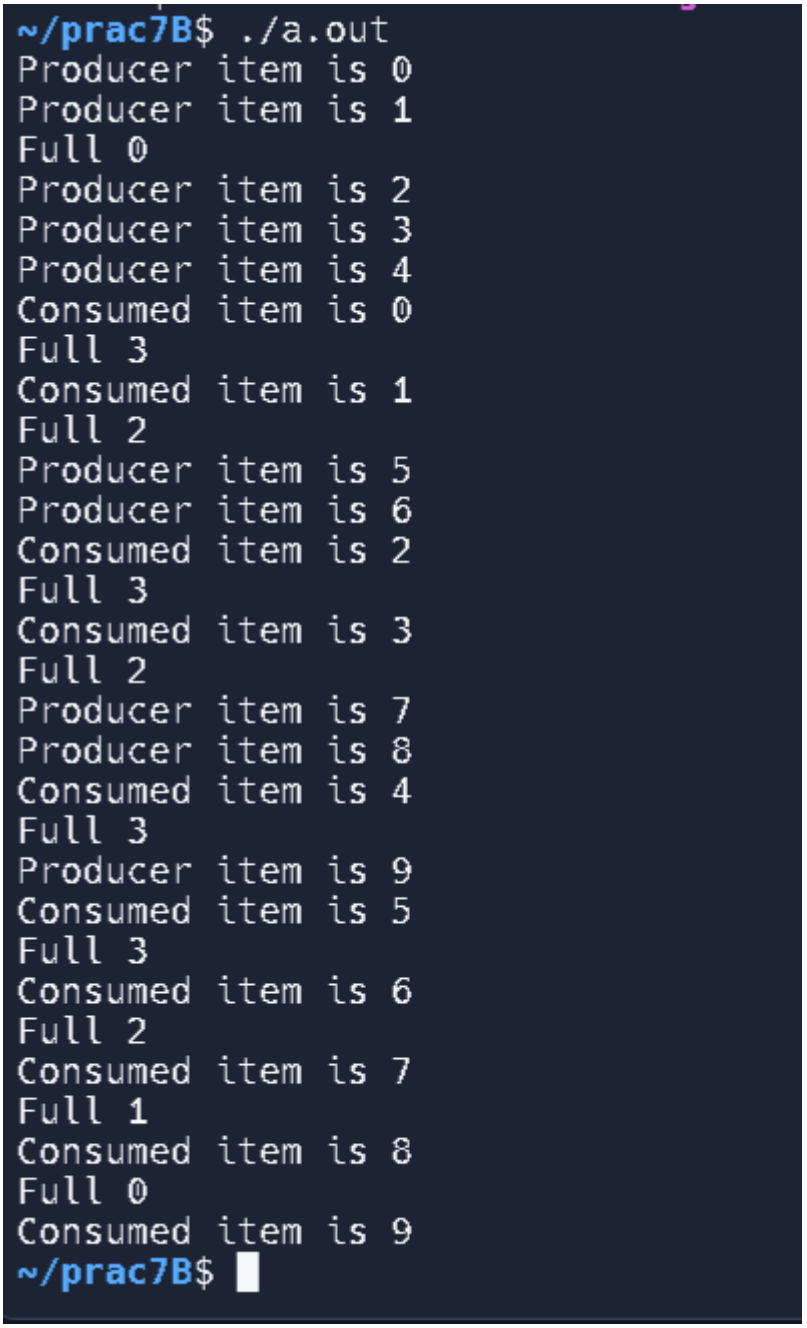
```
~/prac7B$ ./a.out
Enter the no of Producers :
3

Enter the no of Consumers :
3

Producer has produced item : 3

Producer has produced item : 2

Consumer has consumed item : 2

Producer has produced item : 0

Consumer has consumed item : 0

Consumer has consumed item : 3
~/prac7B$
```

2.PRODUCER-CONSUMER PROBLEM – using SEMAPHORE (for one producer and one consumer)

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
int buf[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("Producer item is %d\n", i);
    buf[(++r) % 5] = i;
    sleep(1);
    sem_post(&mutex);
    sem_post(&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);
  }
}
int main() {
  pthread_t tid1, tid2;
  sem_init(&mutex, 0, 1);
  sem_init(&empty, 0, 5);
  sem_init(&full, 0, 0);
  pthread_create(&tid1, NULL, produce, NULL);
  pthread_create(&tid2, NULL, consume, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

o/p:



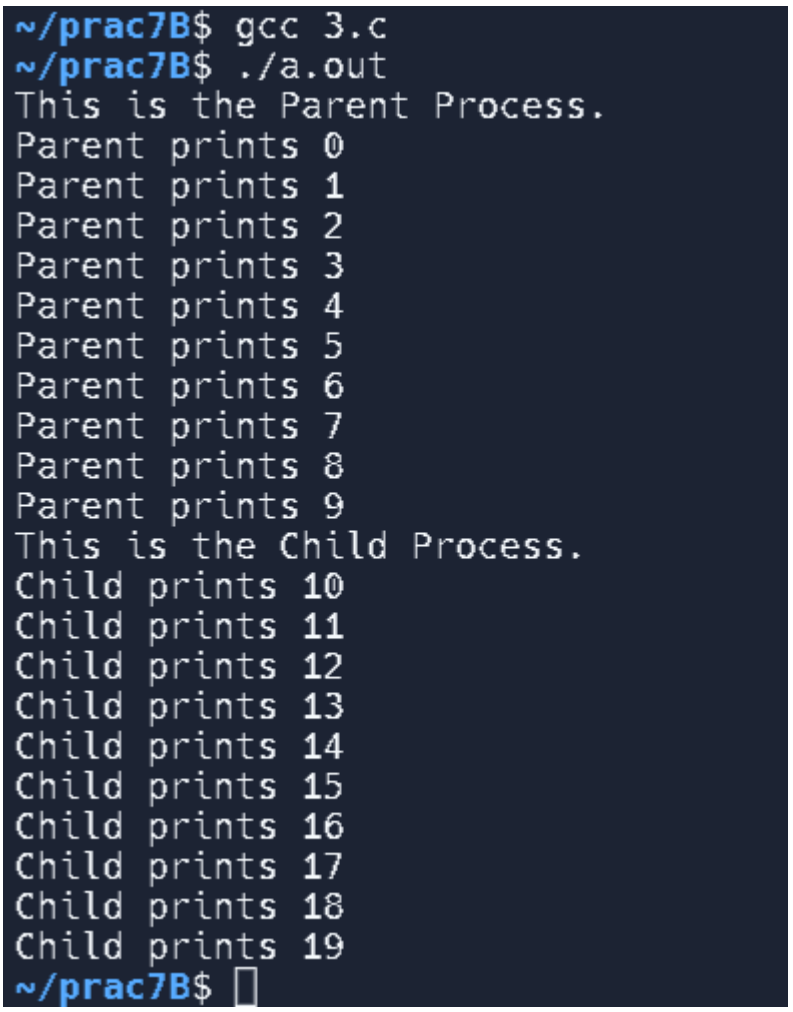
3.Write a program to create an integer variable using shared memory concept and increment the variable simultaneously by two processes. Use semaphores to avoid race conditions.

Code:

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

```
#include <unistd.h>
int *a;
sem_t mutex;
int main() {
  sem_init(&mutex, 0, 1);
  int shmid;
  shmid = shmget(IPC_PRIVATE, 2 * sizeof(int), 0777 | IPC_CREAT);
  a = (int *)shmat(shmid, NULL, 0);
  *a = 0;
  pid_t pid;
  pid = fork();
  if (pid == -1) {
    printf("Error !!\n");
  } else if (pid == 0) {
    // Child Process
    printf("This is the Child Process.\n");
    for (int i = 0; i < 10; i++) {
      sem_wait(&mutex);
      printf("Child prints %d\n", (*a)++);
      sem_post(&mutex);
    }
  } else if (pid > 0) {
    // Parent Process
    printf("This is the Parent Process.\n");
    for (int i = 0; i < 10; i++) {
      sem_wait(&mutex);
      printf("Parent prints %d\n", (*a)++);
      sem_post(&mutex);
    }
  }
}
```

o/p:



4.Producer - Consumer problem solved with semaphores and shared memory.

Problem.h:

```
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <fcntl.h>
#include <sys/shm.h>
#define BUFFER_SIZE 10
#define CONSUMER_SLEEP_SEC 3
#define PRODUCER_SLEEP_SEC 1
#define KEY 1010
// A structure to store BUFFER and semaphores for synchronization
typedef struct
{
int buff[BUFFER_SIZE];
sem_t mutex, 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);
}
void init()
{
// 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);
  }
}
int main() {
  init();
  producer();
  return 0;
}
```

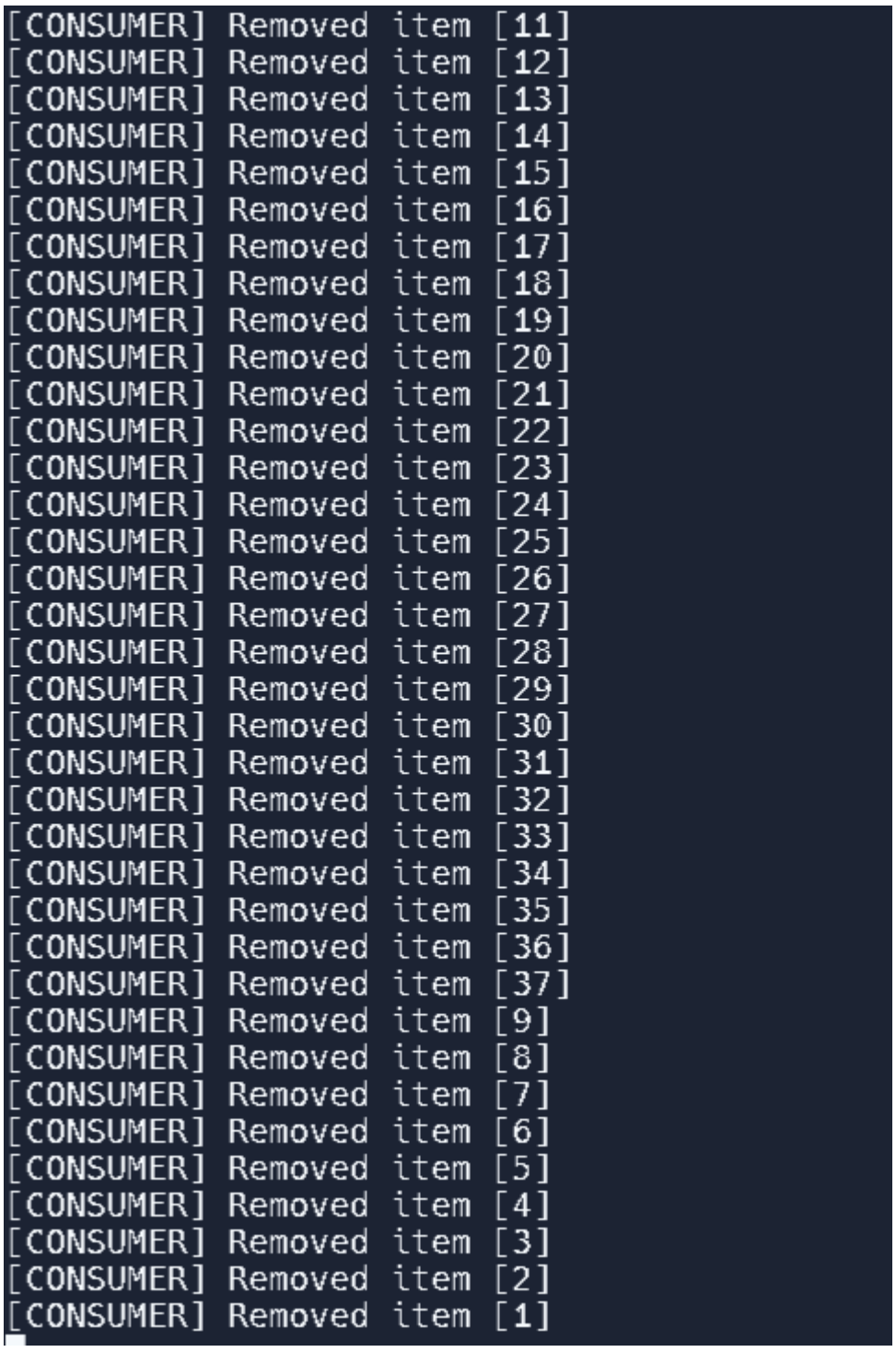
Consumer.c

```
#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 semaphore fullinteger 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);
  }
}
int main() {
  consumer();
  return 0;
}
```

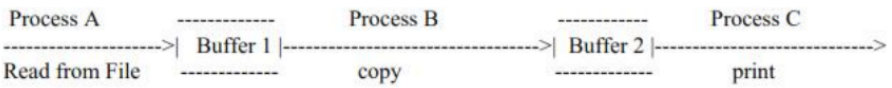
o/p:

```
~/prac7B$ ./a.out
[PRODUCER] Placed item [1]
[PRODUCER] Placed item [2]
[PRODUCER] Placed item [3]
[PRODUCER] Placed item [4]
[PRODUCER] Placed item [5]
[PRODUCER] Placed item [6]
[PRODUCER] Placed item [7]
[PRODUCER] Placed item [8]
[PRODUCER] Placed item [9]
[PRODUCER] Placed item [10]
[PRODUCER] Placed item [11]
[PRODUCER] Placed item [12]
[PRODUCER] Placed item [13]
[PRODUCER] Placed item [14]
[PRODUCER] Placed item [15]
[PRODUCER] Placed item [16]
[PRODUCER] Placed item [17]
[PRODUCER] Placed item [18]
[PRODUCER] Placed item [19]
[PRODUCER] Placed item [20]
[PRODUCER] Placed item [21]
[PRODUCER] Placed item [22]
[PRODUCER] Placed item [23]
[PRODUCER] Placed item [24]
[PRODUCER] Placed item [25]
[PRODUCER] Placed item [26]
[PRODUCER] Placed item [27]
[PRODUCER] Placed item [28]
[PRODUCER] Placed item [29]
[PRODUCER] Placed item [30]
[PRODUCER] Placed item [31]
[PRODUCER] Placed item [32]
```



5.Implement 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.



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.

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

sem_t empty1, empty2, full1, full2;

void *PA(void *args) {
  int *a = (int *)args;
  int val = 32;
  for (int i = 0; i < 3; i++) {
    sem_wait(&empty1);
    sem_wait(&empty2);
    printf("Process A is writing into buffer 1: %d\n", val * i);
    *a = val * i;
    sem_post(&full1);
    sem_post(&empty2);
  }
}

void *PB(void *args) {
  int *a = (int *)args;
  int copy;
  for (int i = 0; i < 3; i++) {
    sem_wait(&full1);
    copy = *a;
    sem_post(&empty1);
    printf("Process B is copying contents from buffer 1 to buffer 2: %d\n", copy);
    *a = copy;
    sem_post(&full2);
  }
}
```

```
void *PC(void *args) {
  int *a = (int *)args;
  for (int i = 0; i < 3; i++) {
    sem_wait(&full2);
    printf("Process C takes data from buffer 2: %d\n", *a);
    sem_post(&empty2);
  }
}

int main() {
  key_t key;
  key = 5678;
  pthread_t TA, TB, TC;
  int *a; // shared variable
  int shmid = shmget(key, sizeof(int), IPC_CREAT | 0666);
  a = (int *)shmat(shmid, NULL, 0);
  sem_init(&empty1, 0, 1);
  sem_init(&empty2, 0, 1);
  sem_init(&full1, 0, 0);
  sem_init(&full2, 0, 0);
  pthread_create(&TA, NULL, PA, a);
  pthread_create(&TB, NULL, PB, a);
  pthread_create(&TC, NULL, PC, a);
  pthread_join(TA, NULL);
  pthread_join(TB, NULL);
  pthread_join(TC, NULL);
  shmdt(a);
  shmctl(shmid, IPC_RMID, NULL);
  return 0;
}
```

o/p:

```
~/prac7B$ gcc 5.c
~/prac7B$ ./a.out
Process A is writing into buffer 1: 0
Process B is copying contents from buffer 1 to buffer 2: 0
Process A is writing into buffer 1: 32
Process B is copying contents from buffer 1 to buffer 2: 32
Process A is writing into buffer 1: 64
Process B is copying contents from buffer 1 to buffer 2: 64
Process C takes data from buffer 2: 64
Process C takes data from buffer 2: 64
Process C takes data from buffer 2: 64
```

6.Readers Writers Problem solved with semaphores and shared memory
Code:

```
#include <semaphore.h>
#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>
#define readers 5
#define writers 2
typedef struct {
  int sharedData;
  sem_t rwMutex;
  sem_t mutex;
  int readersCount;
} SharedMemory;
void readerProcess(SharedMemory *sharedMemory, int readerId) {
  while (1) {
    sem_wait(&(sharedMemory->mutex));
    sharedMemory->readersCount++;
    if (sharedMemory->readersCount == 1) {
      sem_wait(&(sharedMemory->rwMutex));
    }
    sem_post(&(sharedMemory->mutex));
    printf("Reader %d reads shared data: %d\n", readerId,
          sharedMemory - > sharedData);
    sem_wait(&(sharedMemory->mutex));
    sharedMemory->readersCount--;
    if (sharedMemory->readersCount == 0) {
      sem_post(&(sharedMemory->rwMutex));
    }
    sem_post(&(sharedMemory->mutex));
    usleep(rand() % 1000000);
  }
}
void writerProcess(SharedMemory *sharedMemory, int writerId) {
  while (1) {
    sem_wait(&(sharedMemory->rwMutex));
    sharedMemory->sharedData = writerId;
    printf("Writer %d writes shared data: %d\n", writerId,
          sharedMemory - > sharedData);
    sem_post(&(sharedMemory->rwMutex));
    usleep(rand() % 1000000);
  }
}
int main() {
  int shmid;
  SharedMemory *sharedMemory;
```

```
  shmid = shmget(IPC_PRIVATE, sizeof(SharedMemory), IPC_CREAT | 0666);
  sharedMemory = (SharedMemory *)shmat(shmid, NULL, 0);
  sharedMemory->sharedData = 0;
  sem_init(&(sharedMemory->rwMutex), 1, 1);
  sem_init(&(sharedMemory->mutex), 1, 1);
  sharedMemory->readersCount = 0;
  for (int i = 0; i < readers; i++) {
    pid_t pid = fork();
    if (pid < 0) {
      perror("fork error");
      exit(1);
    } else if (pid == 0) {
      readerProcess(sharedMemory, i + 1);
      exit(0);
    }
  }
  for (int i = 0; i < writers; i++) {
    pid_t pid = fork();
    if (pid < 0) {
      perror("fork error");
      exit(1);
    } else if (pid == 0) {
      writerProcess(sharedMemory, i + 1);
      exit(0);
    }
  }
  for (int i = 0; i < readers + writers; i++) {
    wait(NULL);
  }
  shmdt(sharedMemory);
  shmctl(shmid, IPC_RMID, NULL);
  return 0;
}
```

o/p:

```
~/prac7B$ gcc 6.c
~/prac7B$ ./a.out
Reader 1 reads shared data: 0
Reader 2 reads shared data: 0
Reader 3 reads shared data: 0
Reader 4 reads shared data: 0
Writer 1 writes shared data: 1
Reader 5 reads shared data: 1
Writer 2 writes shared data: 2
Reader 1 reads shared data: 2
Reader 2 reads shared data: 2
Reader 3 reads shared data: 2
Reader 4 reads shared data: 2
Writer 1 writes shared data: 1
Reader 5 reads shared data: 1
Writer 2 writes shared data: 2
Reader 1 reads shared data: 2
Reader 2 reads shared data: 2
Reader 3 reads shared data: 2
Reader 4 reads shared data: 2
Writer 1 writes shared data: 1
Reader 5 reads shared data: 1
Writer 2 writes shared data: 2
Reader 1 reads shared data: 2
Reader 2 reads shared data: 2
Reader 3 reads shared data: 2
Reader 4 reads shared data: 2
Writer 1 writes shared data: 1
Reader 5 reads shared data: 1
Writer 2 writes shared data: 2
Reader 1 reads shared data: 2
Reader 2 reads shared data: 2
```

Infinite loop

7.Readers Writers Problem solved with semaphores and pthread.

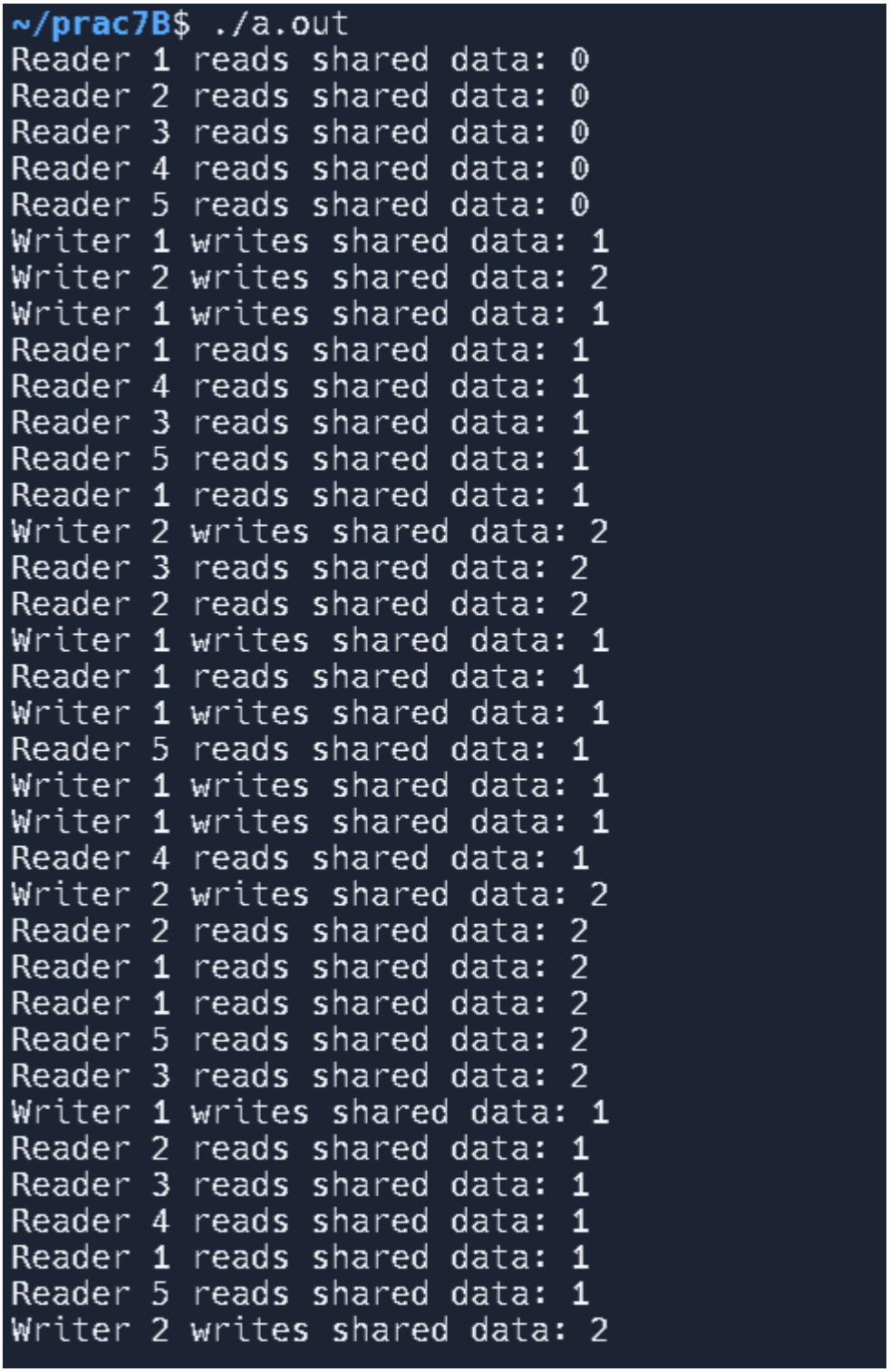
Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define readers 5
#define writers 2
int sharedData = 0;
sem_t rwMutex, mutex;
```

```
int readersCount = 0;
void *reader(void *arg) {
  int readerId = *(int *)arg;
  while (1) {
    sem_wait(&mutex);
    readersCount++;
    if (readersCount == 1) {
      sem_wait(&rwMutex);
    }
    sem_post(&mutex);
    printf("Reader %d reads shared data: %d\n", readerId, sharedData);
    sem_wait(&mutex);
    readersCount--;
    if (readersCount == 0) {
      sem_post(&rwMutex);
    }
    sem_post(&mutex);
    usleep(rand() % 1000000);
  }
  pthread_exit(NULL);
}
void *writer(void *arg) {
  int writerId = *(int *)arg;
  while (1) {
    sem_wait(&rwMutex);
    sharedData = writerId;
    printf("Writer %d writes shared data: %d\n", writerId, sharedData);
    sem_post(&rwMutex);
    usleep(rand() % 1000000);
  }
  pthread_exit(NULL);
}
int main() {
  pthread_t readerThreads[readers];
  pthread_t writerThreads[writers];
  int readerIds[readers];
  int writerIds[writers];
  sem_init(&rwMutex, 0, 1);
  sem_init(&mutex, 0, 1);
  for (int i = 0; i < readers; i++) {
    readerIds[i] = i + 1;
    pthread_create(&readerThreads[i], NULL, reader, &readerIds[i]);
  }
  for (int i = 0; i < writers; i++) {
    writerIds[i] = i + 1;
    pthread_create(&writerThreads[i], NULL, writer, &writerIds[i]);
  }
  for (int i = 0; i < readers; i++) {
```

```
    pthread_join(readerThreads[i], NULL);
  }
  for (int i = 0; i < writers; i++) {
    pthread_join(writerThreads[i], NULL);
  }
  sem_destroy(&rwMutex);
  sem_destroy(&mutex);
  return 0;
}
```

o/p:



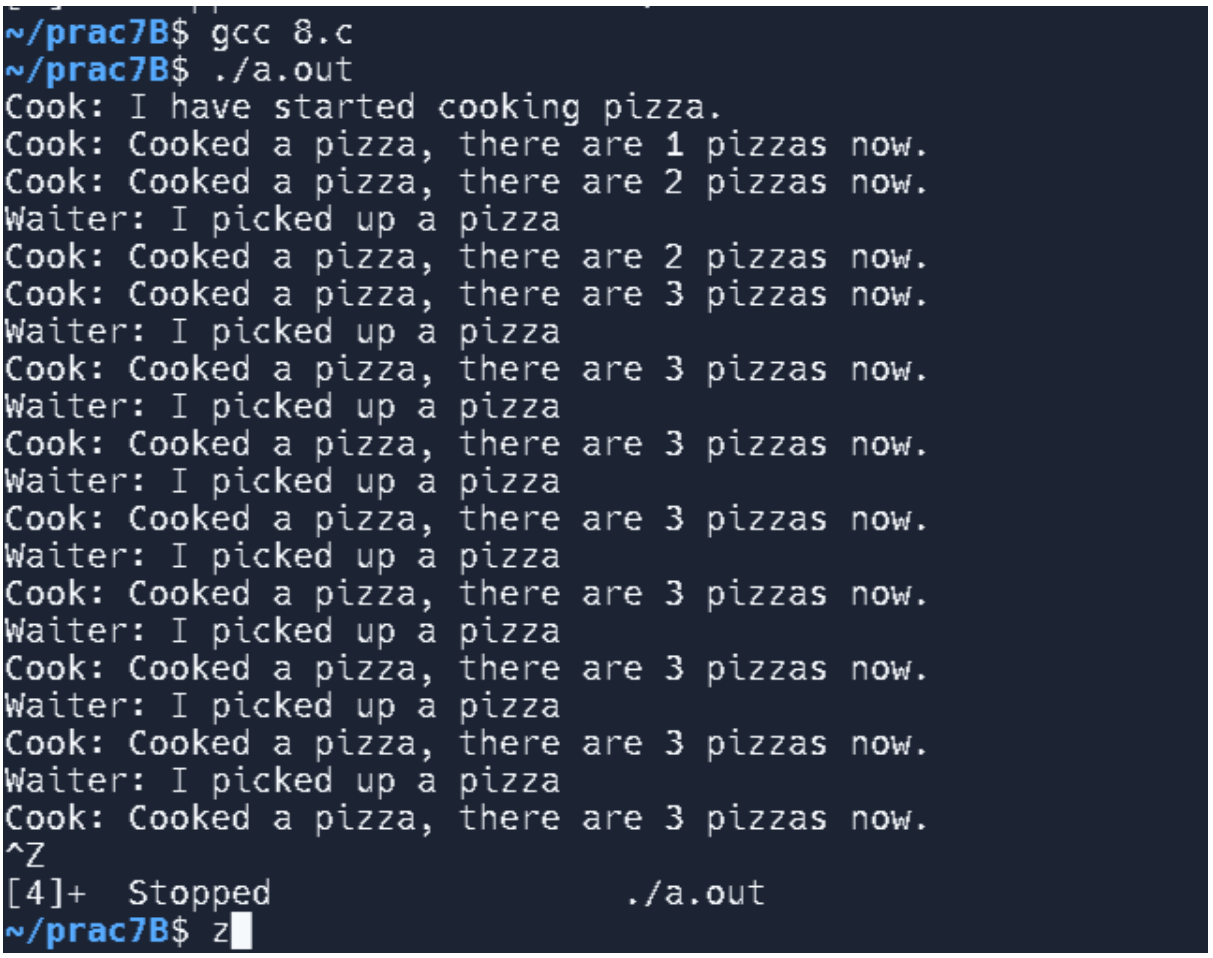
8.Cook cooks pizza and puts that pizza onto shelf. Waiter picks pizza from the shelf and serves it to customers. The shelf can hold three pizza at most at the same time. When the shelf is full, cook wait until picked up; when there is no pizza on the shelf, waiter waits until made. Hint: We use three semaphores to synchronize cook and waiter. cook.c is a producer program. After cooking one pizza and placing it on shelf, it posts semaphore fill and makes shelf increase by 1. waiter.c is a consumer program. After picking a pizza from the shelf, it posts semaphore avail and makes shelf decrease by 1. The value of shelf is the current

number of pizza and processes should access it exclusively. Semaphore mutex is response for the mutual exclusion. Note that shared memory is used in these two programs.

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAX_PIZZA 3
int shelf = 0;
sem_t mutex;
void *cook(void *arg) {
  while (1) {
    if (shelf < MAX_PIZZA) {
      sem_wait(&mutex);
      shelf++;
      printf("Cook: Cooked a pizza, there are %d pizzas now.\n", shelf);
      sem_post(&mutex);
    }
    sleep(1);
  }
}
void *waiter(void *arg) {
  while (1) {
    if (shelf > 0) {
      sem_wait(&mutex);
      printf("Waiter: I picked up a pizza\n");
      shelf--;
      sem_post(&mutex);
    }
    sleep(2);
  }
}
int main() {
  pthread_t Tcook, Twaiter;
  sem_init(&mutex, 0, 1);
  printf("Cook: I have started cooking pizza.\n");
  pthread_create(&Tcook, NULL, cook, NULL);
  pthread_create(&Twaiter, NULL, waiter, NULL);
  pthread_join(Tcook, NULL);
  pthread_join(Twaiter, NULL);
  return 0;
}
```

o/p:



Result:”
Linux C programs to demonstrate the concept of threads and semaphores for process synchronization has been implemented