OPERATING SYSTEMS LAB

PRACTICAL 7

NAME: VEDANT BHUTADA

ROLL: 69

BATCH: A4

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

Part-7A (Threads)

Program-1: A simple C program to demonstrate use of pthread basic functions and to implement multiple threads with global and static variables

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

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Store the value argument passed to this thread
    int myid = (int)vargp;

    // Let us create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    ++s; ++g;

    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", myid, ++s, ++g);
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 3; i++)
```

```
        pthread_create(&tid, NULL, myThreadFun, (void *)i);


    pthread_exit(NULL);
    return 0;
}
```

```
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gcc -o p7 prac7_1.c -pthread
prac7_1.c: In function ‘myThreadFun’:
prac7_1.c:12:16: warning: cast from pointer to integer of different size [-Wpoin
ter-to-int-cast]
     int myid = (int)vargp;
                ^
prac7_1.c: In function ‘main’:
prac7_1.c:32:49: warning: cast to pointer from integer of different size [-Wint-
to-pointer-cast]
         pthread_create(&tid, NULL, myThreadFun, (void *)i);
                                                 ^
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ ./p7
Thread ID: 0, Static: 2, Global: 2
Thread ID: 2, Static: 4, Global: 4
Thread ID: 1, Static: 6, Global: 6
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$
```

Program 2) To demonstrate thread system calls

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

pthread_t tid[2];

void* doSomeThing(void *arg)
{
unsigned long i = 0;
pthread_t id = pthread_self();

if(pthread_equal(id, tid[0]))
    {
printf("\n First thread processing\n");
    }
else
    {
printf("\n Second thread processing\n");
    }

for(i=0; i<(0xFFFFFFFF);i++);

return NULL;
}

int main(void)
{
int i = 0;
int err;
```

```
while(i < 2)
    {
err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
if (err != 0)
printf("\ncan't create thread :[%s]", strerror(err));
else
printf("\n Thread created successfully\n");

i++;
}

sleep(5);
return 0;
}
```

```
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gcc -o p72 prac7_2.c -lpthread
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ ./p72

 Thread created successfully

 First thread processing

 Thread created successfully

 Second thread processing
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$
```

---

3) Matrix Multiplication using Threads

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

#define SIZE 3 // Size of the matrices

int matrix_A[SIZE][SIZE];
int matrix_B[SIZE][SIZE];
int result_matrix[SIZE][SIZE];

// Structure for passing data to threads
typedef struct {
    int row;
    int col;
} thread_data;

// Function executed by each thread
void *multiply(void *arg) {
    thread_data *data = (thread_data *)arg;
    int row = data->row;
    int col = data->col;

    int sum = 0;
    for (int i = 0; i < SIZE; i++) {
        sum += matrix_A[row][i] * matrix_B[i][col];
    }
    result_matrix[row][col] = sum;
```

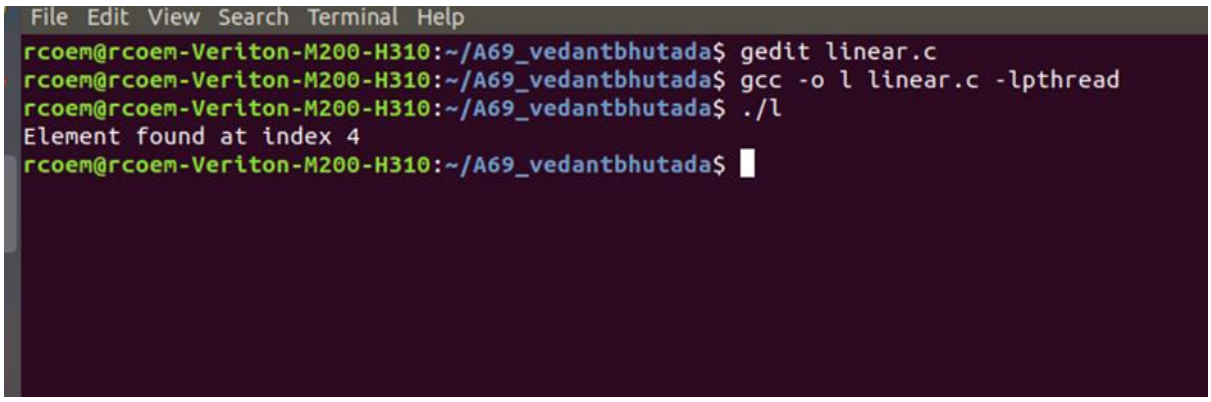
```
    pthread_exit(NULL);
}

int main() {
    // Initialize matrices
    printf("Enter elements of matrix A:\n");
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            scanf("%d", &matrix_A[i][j]);
        }
    }

    printf("Enter elements of matrix B:\n");
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            scanf("%d", &matrix_B[i][j]);
        }
    }

    pthread_t threads[SIZE][SIZE];
    thread_data thread_data_array[SIZE][SIZE];

    // Create threads for matrix multiplication
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            thread_data_array[i][j].row = i;
            thread_data_array[i][j].col = j;
            pthread_create(&threads[i][j], NULL, multiply,
&thread_data_array[i][j]);
        }
    }

    // Wait for all threads to finish
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            pthread_join(threads[i][j], NULL);
        }
    }

    // Print the result matrix
    printf("\nResultant matrix:\n");
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%d ", result_matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

---

4) Linear search using Multi-threading (use n number of threads)

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

#define THREAD_COUNT 4

typedef struct {
    int* arr;
    int target;
    int start;
    int end;
    int* result;
} ThreadData;

void* linear_search(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int* arr = data->arr;
    int target = data->target;
    int start = data->start;
    int end = data->end;
    int* result = data->result;

    for (int i = start; i < end; i++) {
        if (arr[i] == target) {
            *result = i;
            return NULL;
        }
    }

    return NULL;
}

int parallel_linear_search(int* arr, int n, int target, int
thread_count) {
```

```
    pthread_t threads[THREAD_COUNT];
    ThreadData thread_data[THREAD_COUNT];

    int block_size = n / thread_count;
    int result = -1;

    for (int i = 0; i < thread_count; i++) {
        thread_data[i].arr = arr;
        thread_data[i].target = target;
        thread_data[i].start = i * block_size;
        thread_data[i].end = (i == thread_count - 1) ? n : (i + 1) *
block_size;
        thread_data[i].result = &result;

        pthread_create(&threads[i], NULL, linear_search,
&thread_data[i]);
    }

    for (int i = 0; i < thread_count; i++) {
        pthread_join(threads[i], NULL);
    }

    return result;
}

int main() {
    int arr[] = {5, 12, 8, 3, 9, 6, 1, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 9;
    int result = parallel_linear_search(arr, n, target, THREAD_COUNT);

    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```



5) To find the maximum and minimum element in an array using Multi-threading (for 100 to 200 numbers or more and create 10 or more threads)

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

#define ARRAY_SIZE 100
#define THREAD_COUNT 10

typedef struct {
    int* arr;
    int start;
    int end;
    int min;
    int max;
} ThreadData;

void* find_min_max(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int* arr = data->arr;
    int start = data->start;
    int end = data->end;
    int min = arr[start];
    int max = arr[start];

    for (int i = start + 1; i < end; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    data->min = min;
    data->max = max;

    return NULL;
}

void generate_random_array(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 101; // Generate random numbers between 0 and
100
    }
}

int main() {
    int arr[ARRAY_SIZE];
    generate_random_array(arr, ARRAY_SIZE);

    pthread_t threads[THREAD_COUNT];
    ThreadData thread_data[THREAD_COUNT];

    int block_size = ARRAY_SIZE / THREAD_COUNT;
    int min = arr[0];
    int max = arr[0];
```

```
    for (int i = 0; i < THREAD_COUNT; i++) {
        thread_data[i].arr = arr;
        thread_data[i].start = i * block_size;
        thread_data[i].end = (i == THREAD_COUNT - 1) ? ARRAY_SIZE : (i
+ 1) * block_size;

        pthread_create(&threads[i], NULL, find_min_max,
&thread_data[i]);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_join(threads[i], NULL);

        if (thread_data[i].min < min) {
            min = thread_data[i].min;
        }
        if (thread_data[i].max > max) {
            max = thread_data[i].max;
        }
    }

    printf("Minimum element: %d\n", min);
    printf("Maximum element: %d\n", max);

    return 0;
}
```

```
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gedit maxmin.c
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gcc -o mm maxmin.c -lpthread
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ ./mm
Minimum element: 0
Maximum element: 100
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$
```

---

6) Example without synchronization FOR PRODUCER CONSUMER PROBLEM

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

void *producer(); /* the thread */
void *consumer(); /* the thread */

int main() {

pthread_t ptid, ctid;       //Thread ID
pthread_create(&ptid,NULL,producer,NULL);
pthread_create(&ctid,NULL,consumer,NULL);
pthread_join(ptid, NULL);
pthread_join(ctid, NULL);
     }

//The thread will begin control in this function
void *producer(void *param) {
do{
```

```
printf("I m producer\n");

}while(1);
pthread_exit(0);
}

//The thread will begin control in this function
void *consumer(void *param) {
do{
printf("I m consumer\n");
}while(1);
pthread_exit(0);
}
```

```
I m producer
I m producer
I m producer
I m producer
I m producer
I m producer
I m producer
I m producer
I m producer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
I m consumer
```

7) Example with synchronization : using mutex & condition variable FOR PRODUCER CONSUMER PROBLEM

```
# include <stdio.h>
# include <stdlib.h>
# include <pthread.h>
# define BufferSize 10
```

```
void *Producer();
void *Consumer();

int BufferIndex= -1;
char BUFFER[10];

pthread_cond_t  Buffer_Empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t  Buffer_Full =PTHREAD_COND_INITIALIZER;
pthread_mutex_t mVar=PTHREAD_MUTEX_INITIALIZER;

int main()
{
pthread_t ptid,ctid;


pthread_create(&ptid,NULL,Producer,NULL);
pthread_create(&ctid,NULL,Consumer,NULL);

pthread_join(ptid,NULL);
pthread_join(ctid,NULL);

return 0;
}

void *Producer()
{

    //do
int i;
for(i=0; i<15 ; i++)
    {
pthread_mutex_lock(&mVar);
if(BufferIndex==BufferSize-1)
pthread_cond_wait(&Buffer_Empty,&mVar);

BUFFER[++BufferIndex]='#';
printf("Produce : %d \n",BufferIndex);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Full);

    }//while(1);

}

void *Consumer()
{

    //do
int i;
for(i=0; i<15 ; i++)
    {
pthread_mutex_lock(&mVar);
if(BufferIndex==-1) {
pthread_cond_wait(&Buffer_Full,&mVar);
}
printf("Consume : %d \n",BufferIndex--);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Empty);
```

```
    }//while(1);
}
```

```
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gedit prac7_7.c
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gcc -o p77 prac7_7.c -lpthread
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ ./p77
Produce : 0
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Produce : 5
Produce : 6
Produce : 7
Produce : 8
Produce : 9
Consume : 9
Consume : 8
Consume : 7
Consume : 6
Consume : 5
Consume : 4
Consume : 3
Consume : 2
Consume : 1
Consume : 0
Produce : 0
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Consume : 4
Consume : 3
Consume : 2
Consume : 1
Consume : 0
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$
```

---

8) Readers Writers Problem solved with mutex and pthread.

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

#define NUM_READERS 5
#define NUM_WRITERS 2

int sharedData = 0;
int readersCount = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t readersCV = PTHREAD_COND_INITIALIZER;
pthread_cond_t writersCV = PTHREAD_COND_INITIALIZER;

void *reader(void *arg) {
    int readerId = *((int *)arg);

    pthread_mutex_lock(&mutex);
    while (readersCount == -1)
        pthread_cond_wait(&readersCV, &mutex);

    readersCount++;
    pthread_mutex_unlock(&mutex);
```

```
    // Read data
    printf("Reader %d read data: %d\n", readerId, sharedData);

    pthread_mutex_lock(&mutex);
    readersCount--;

    if (readersCount == 0)
        pthread_cond_signal(&writersCV);

    pthread_mutex_unlock(&mutex);

    return NULL;
}

void *writer(void *arg) {
    int writerId = *((int *)arg);

    pthread_mutex_lock(&mutex);
    while (readersCount != 0)
        pthread_cond_wait(&writersCV, &mutex);

    readersCount = -1;

    // Write data
    sharedData++;
    printf("Writer %d wrote data: %d\n", writerId, sharedData);

    readersCount = 0;

    pthread_cond_broadcast(&readersCV);
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main() {
    pthread_t readers[NUM_READERS];
    pthread_t writers[NUM_WRITERS];
    int i;

    int readerIds[NUM_READERS];
    int writerIds[NUM_WRITERS];

    for (i = 0; i < NUM_READERS; i++) {
        readerIds[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &readerIds[i]);
    }

    for (i = 0; i < NUM_WRITERS; i++) {
        writerIds[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writerIds[i]);
    }

    for (i = 0; i < NUM_READERS; i++)
        pthread_join(readers[i], NULL);

    for (i = 0; i < NUM_WRITERS; i++)
```

```
        pthread_join(writers[i], NULL);

    return 0;
}
```

```
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gedit rw.c
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gcc -o rewr rw.c -lpthread
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ ./rewr
Reader 1 read data: 0
Reader 2 read data: 0
Reader 4 read data: 0
Reader 3 read data: 0
Writer 1 wrote data: 1
Reader 5 read data: 1
Writer 2 wrote data: 2
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$
```

Part-7B (Semaphores)

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

```
#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);
	sem_destroy(&full);
	sem_destroy(&empty);
	exit(0);
}
```

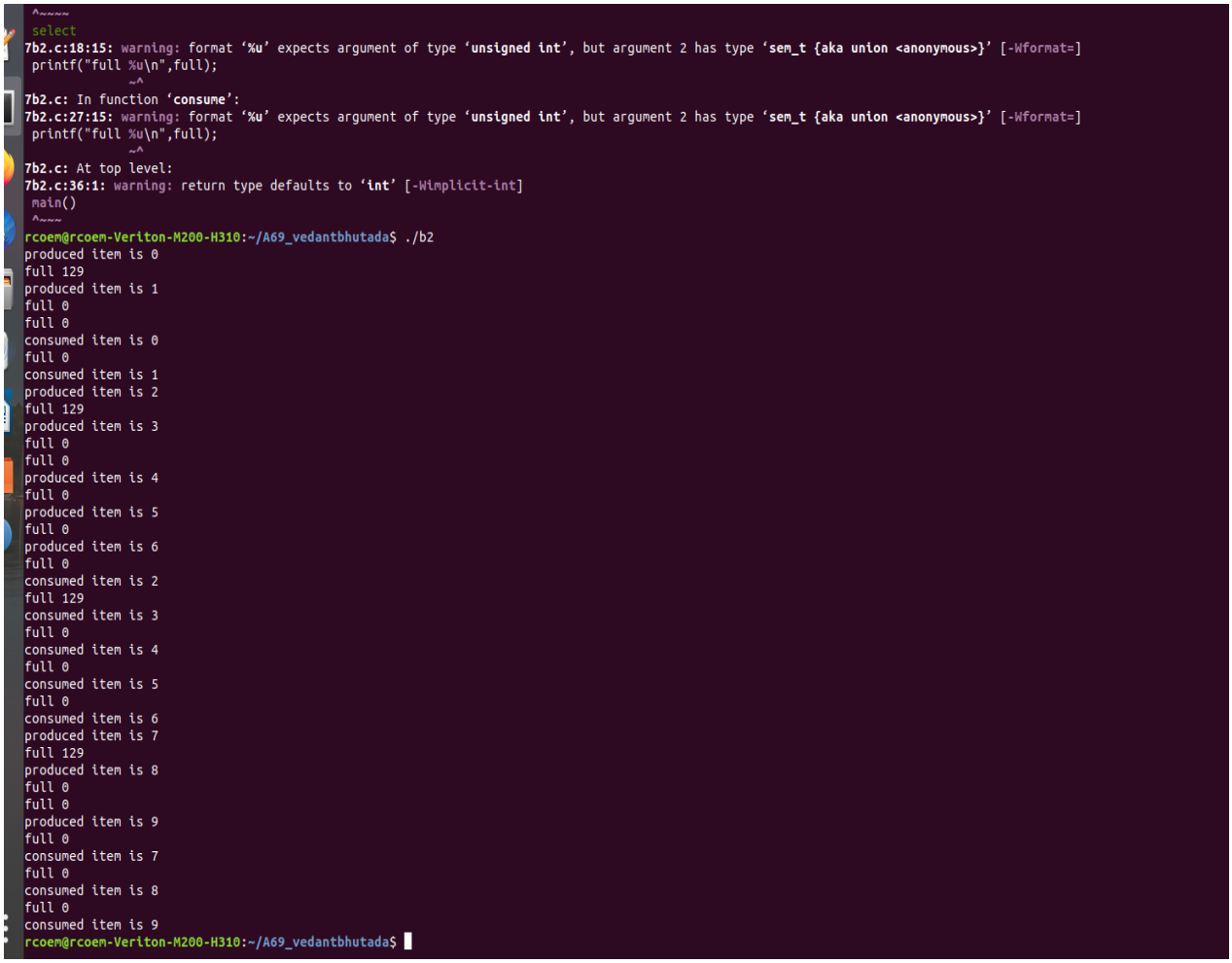
```
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ gcc -o b1 7b1.c -lpthread
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$ ./b1

Enter the no of producers: 2

Enter the no of consumers: 2

Producer has produced item: 3

Producer has produced item: 2

Consumer has consumed item: 2

Consumer has consumed item: 3
rcoem@rcoem-Veriton-M200-H310:~/A69_vedantbhutada$
```

---

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

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.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("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);
sem_destroy(&mutex);
sem_destroy(&full);
sem_destroy(&empty);
}
```



3) AIM: 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.

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

#define KEY 1234

int main() {
    int shmid, semid;
    key_t key = KEY;
    int *shared_var;

    // Create shared memory segment
    shmid = shmget(key, sizeof(int), IPC_CREAT | 0666);
    if (shmid < 0) {
        perror("shmget");
        exit(1);
    }

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

    // Create semaphore
    semid = semget(key, 1, IPC_CREAT | 0666);
    if (semid < 0) {
        perror("semget");
        exit(1);
    }

    // Initialize semaphore
    if (semctl(semid, 0, SETVAL, 1) < 0) {
        perror("semctl");
        exit(1);
    }

    int pid = fork();
    if (pid == 0) {
        // Child process
        for (int i = 0; i < 10; i++) {
            struct sembuf sem_op;
            sem_op.sem_num = 0;
            sem_op.sem_op = -1; // Wait
            sem_op.sem_flg = SEM_UNDO;
            semop(semid, &sem_op, 1);
```

```
            (*shared_var)++;
            printf("Child Process: Incremented value = %d\n",
*shared_var);

            sem_op.sem_op = 1; // Signal
            semop(semid, &sem_op, 1);

            sleep(1);
        }
    } else if (pid > 0) {
        // Parent process
        for (int i = 0; i < 10; i++) {
            struct sembuf sem_op;
            sem_op.sem_num = 0;
            sem_op.sem_op = -1; // Wait
            sem_op.sem_flg = SEM_UNDO;
            semop(semid, &sem_op, 1);

            (*shared_var)++;
            printf("Parent Process: Incremented value = %d\n",
*shared_var);

            sem_op.sem_op = 1; // Signal
            semop(semid, &sem_op, 1);

            sleep(1);
        }
    } else {
        perror("fork");
        exit(1);
    }

    // Detach shared memory segment
    if (shmdt(shared_var) < 0) {
        perror("shmdt");
        exit(1);
    }

    // Remove shared memory segment
    if (shmctl(shmid, IPC_RMID, NULL) < 0) {
        perror("shmctl");
        exit(1);
    }

    // Remove semaphore
    if (semctl(semid, 0, IPC_RMID) < 0) {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

```
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gedit 7b_3.c
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gcc 7b_3.c
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ ./a.out
Parent Process: Incremented value = 1
Child Process: Incremented value = 2
Parent Process: Incremented value = 3
Child Process: Incremented value = 4
Parent Process: Incremented value = 5
Child Process: Incremented value = 6
Parent Process: Incremented value = 7
Child Process: Incremented value = 8
Parent Process: Incremented value = 9
Child Process: Incremented value = 10
Parent Process: Incremented value = 11
Child Process: Incremented value = 12
Parent Process: Incremented value = 13
Child Process: Incremented value = 14
Parent Process: Incremented value = 15
Child Process: Incremented value = 16
Parent Process: Incremented value = 17
Child Process: Incremented value = 18
Parent Process: Incremented value = 19
Child Process: Incremented value = 20
```

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>
#include<unistd.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
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);

	}
}

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

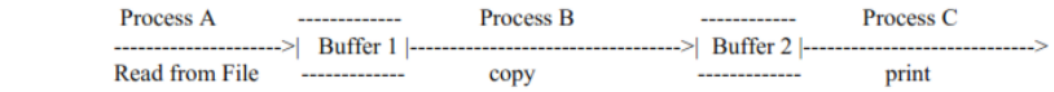
```
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gcc -o p producer.c -lpthread
producer.c:23:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   23 | main()
      | ^~~~
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ ./p
[PRODUCER] Placed item [1]
[PRODUCER] Placed item [2]
[PRODUCER] Placed 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]
^C
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$
```

```
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gcc problem.h
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gcc -o c consumer.c -lpthread
consumer.c:21:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   21 | main()
      | ^~~~
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ ./c
^C
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gcc -o c consumer.c -lpthread
consumer.c:21:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   21 | main()
      | ^~~~
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ ./c
^C
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ gcc -o c consumer.c -lpthread
consumer.c:21:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   21 | main()
      | ^~~~
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$ ./c
[CONSUMER] Removed item [3]
[CONSUMER] Removed item [6]
[CONSUMER] Removed item [8]
[CONSUMER] Removed item [11]
[CONSUMER] Removed item [14]
[CONSUMER] Removed item [15]
[CONSUMER] Removed item [13]
[CONSUMER] Removed item [12]
[CONSUMER] Removed item [10]
[CONSUMER] Removed item [9]
[CONSUMER] Removed item [7]
[CONSUMER] Removed item [5]
[CONSUMER] Removed item [4]
^C
rcoem@rcoem-Vostro-3669:~/A69_VEDANTBHUTADA$
```

---

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.

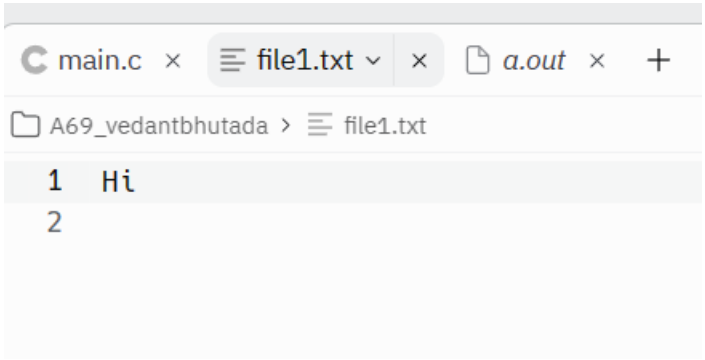


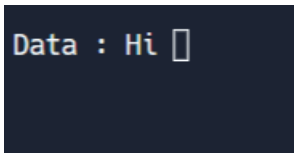
Assume all three processes operate on one (file) record at a time, both buffers' capacity are one record. Write a program to coordinate the three processes using semaphores.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<semaphore.h>
#include<pthread.h>
FILE *fp;
int buffer_1, buffer_2;
sem_t sem1, sem2, sem3;
void *ProcessA(void *args) {
   int data;
   while (fscanf(fp, "%d", &data) != EOF) {
```

```
        sem_wait(&sem1);
        buffer_1 = data;        sem_post(&sem2);
    }

    pthread_exit(NULL);
}
void *ProcessB(void *args) {
 int data;     while (1) {
        sem_wait(&sem2);
     data = buffer_1;
  if (data == -1) {
    buffer_2=data;
          sem_post(&sem1);
           sem_post(&sem3);
          break;
}else{
           buffer_2 = data;
          sem_post(&sem1);
         sem_post(&sem3);
        }
    }
    pthread_exit(NULL);
}
void *ProcessC(void *args) {
     while (1) {
      sem_wait(&sem3);
     if (buffer_2 == -1) {
        break;
        }       else {
           printf("Data: %d\n", buffer_2);
          sem_post(&sem1);
        }
    }
    pthread_exit(NULL);
}
int main() {
    pthread_t threadA, threadB, threadC;
    fp = fopen("file1.txt", "r");     if (fp == NULL) {
      printf("File does not exist\n");
      exit(1);
    }
    sem_init(&sem1, 0, 1);
   sem_init(&sem2, 0, 0);
  sem_init(&sem3, 0, 0);
    pthread_create(&threadA, NULL, ProcessA, NULL);
    pthread_create(&threadB, NULL, ProcessB, NULL);
   pthread_create(&threadC, NULL, ProcessC, NULL);
```

```
  sleep(3);

   pthread_join(threadA, NULL);
  pthread_join(threadB, NULL);
   pthread_join(threadC, NULL);
    sem_destroy(&sem1);
 sem_destroy(&sem2);
  sem_destroy(&sem3);
   fclose(fp);
  return 0;
}
```





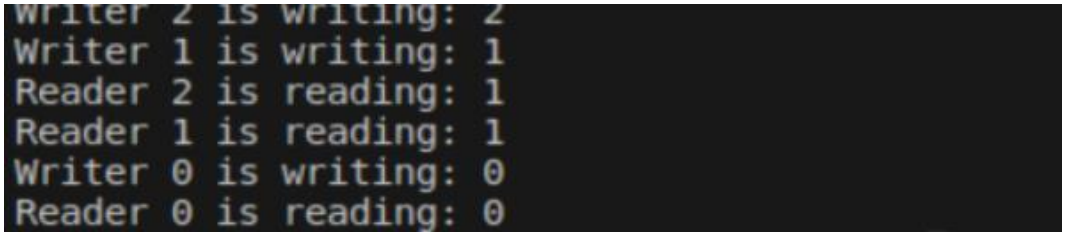
---

6) Readers Writers Problem solved with semaphores and shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/shm.h>

#define SHARED_MEMORY_KEY 12345

#define NUM_READERS 3 #define NUM_WRITERS 3
typedef struct {
  int readers_count;
   int shared_data;
 pthread_mutex_t mutex;
  pthread_cond_t readers_cond;
   pthread_cond_t writers_cond;
} SharedMemory;
SharedMemory* shared_memory;
void* reader(void* arg) {
```

```
  int reader_id = (int)arg;

      pthread_mutex_lock(&shared_memory->mutex);
      // Wait if there is an active writer
      while (shared_memory->readers_count == -1)
         pthread_cond_wait(&shared_memory->readers_cond, &shared_memory>mutex);
      // Increment readers count
      shared_memory->readers_count++;
      pthread_mutex_unlock(&shared_memory->mutex);
      // Read operation
      printf("Reader %d is reading: %d\n", reader_id, shared_memory->shared_data);
      pthread_mutex_lock(&shared_memory->mutex);
      // Decrement readers count and signal waiting writers if it's the last reader
shared_memory->readers_count--;
    if (shared_memory->readers_count == 0)
         pthread_cond_signal(&shared_memory->writers_cond);
      pthread_mutex_unlock(&shared_memory->mutex);

  return NULL;
}
void* writer(void* arg) {    int writer_id = (int)arg;

      pthread_mutex_lock(&shared_memory->mutex);
      // Wait if there is an active writer or if there are active readers
   while (shared_memory->readers_count != 0)
         pthread_cond_wait(&shared_memory->writers_cond, &shared_memory>mutex);
      // Set readers_count to -1 to block subsequent readers
  shared_memory->readers_count = -1;
      // Write operation
      shared_memory->shared_data = writer_id;
      printf("Writer %d is writing: %d\n", writer_id, shared_memory->shared_data);       //
Reset readers count and signal waiting readers and writers
  shared_memory->readers_count = 0;
      pthread_cond_broadcast(&shared_memory->readers_cond);
pthread_cond_signal(&shared_memory->writers_cond);
      pthread_mutex_unlock(&shared_memory->mutex);

  return NULL;
}
int main() {
  int shm_id = shmget(SHARED_MEMORY_KEY, sizeof(SharedMemory), IPC_CREAT |
0666);
  if (shm_id == -1) {
   perror("shmget failed");
     exit(1);
  }
  shared_memory = (SharedMemory*)shmat(shm_id, NULL, 0);
```

```
    if (shared_memory == (void*)-1) {
       perror("shmat failed");
     exit(1);
    }
    shared_memory->readers_count = 0;
 shared_memory->shared_data = 0;
    pthread_mutex_init(&shared_memory->mutex, NULL);
pthread_cond_init(&shared_memory->readers_cond, NULL);
pthread_cond_init(&shared_memory->writers_cond, NULL);
    pthread_t readers[NUM_READERS];
    pthread_t writers[NUM_WRITERS];
    int reader_ids[NUM_READERS];
   int writer_ids[NUM_WRITERS];
    // Create multiple reader threads
   for (int i = 0; i < NUM_READERS; i++) {
  reader_ids[i] = i;
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
    }
    // Create multiple writer threads
for (int i = 0; i < NUM_WRITERS; i++) {
 writer_ids[i] = i;
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }
    // Wait for threads to finish (not necessary for an infinite loop)
 for (int i = 0; i < NUM_READERS; i++) {
  pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
      pthread_join(writers[i], NULL);
    }
    pthread_mutex_destroy(&shared_memory->mutex);
pthread_cond_destroy(&shared_memory->readers_cond);
pthread_cond_destroy(&shared_memory->writers_cond);
    if (shmdt(shared_memory) == -1) {
    perror("shmdt failed");
   exit(1);
    }
    if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
     perror("shmctl failed");
  exit(1);
    }
    return 0;
}
```



---

7) Readers Writers Problem solved with semaphores and pthread.

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
sem_t mutex,wrt;
int readercount = 0;


void reader(void param)
{
   for(int i=0;i<5;i++){
            sem_wait(&mutex);
            readercount++;
            if(readercount==1)
              sem_wait(&wrt);
            sem_post(&mutex);
            printf("\n%d reader is inside",readercount);
            sem_wait(&mutex);
            readercount--;
            if(readercount==0)
            {
                  sem_post(&wrt);
            }
            sem_post(&mutex);
            printf("\n%d Reader is leaving",readercount+1);
            sleep(3);
   }
}

void writer(void param)
{
   for(int i=0;i<5;i++){
            printf("\nWriter is trying to enter");
            sem_wait(&wrt);
            printf("\nWriter has entered");
            sem_post(&wrt);
            printf("\nWriter is leaving\n");
```

```
            sleep(2);
    }
}


int main()
{
    pthread_t tid1,tid2;
    sem_init(&mutex,0,1);
    sem_init(&wrt,0,1);
    pthread_create(&tid1,NULL,reader,NULL);
    pthread_create(&tid2,NULL,writer,NULL);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    sem_destroy(&mutex);
    sem_destroy(&wrt);
    return 0;
}
```

```
~/DeficientCumbersomeProcessor/A69_vedantbhutada$ gcc main.c
~/DeficientCumbersomeProcessor/A69_vedantbhutada$ ./a.out

1 reader is inside
1 Reader is leaving
Writer is trying to enter
Writer has entered
Writer is leaving

Writer is trying to enter
Writer has entered
Writer is leaving

1 reader is inside
1 Reader is leaving
Writer is trying to enter
Writer has entered
Writer is leaving

1 reader is inside
1 Reader is leaving
Writer is trying to enter
Writer has entered
Writer is leaving

Writer is trying to enter
Writer has entered
Writer is leaving

1 reader is inside
1 Reader is leaving
1 reader is inside
1 Reader is leaving~/DeficientCumbersomeProcessor/A69_vedantbhutada
$
```

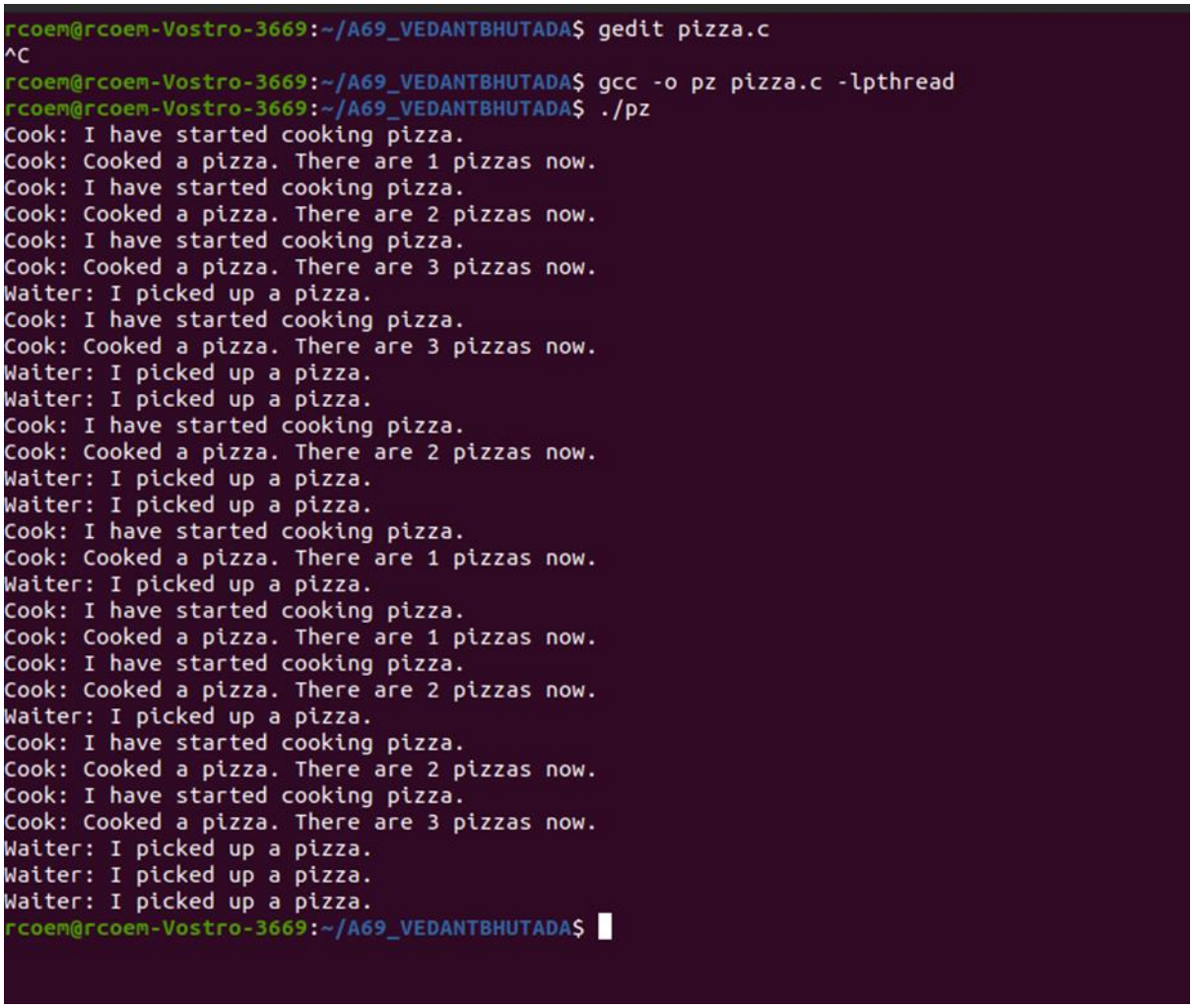
---

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_PIZZA 10

#define MAX_PIZZAS 3

sem_t fill;    // Semaphore to track available pizzas on the
shelf
sem_t avail;   // Semaphore to track empty slots on the shelf
sem_t mutex;   // Semaphore for mutual exclusion

int pizzas_on_shelf = 0;
int pizzas_cooked = 0;

void *cook(void *arg) {
        sem_wait(&avail);
        sem_wait(&mutex);
        printf("Cook: I have started cooking pizza.\n");
        pizzas_on_shelf++;
        printf("Cook: Cooked a pizza. There are %d pizzas
now.\n", pizzas_on_shelf);

        sem_post(&mutex);
        sem_post(&fill);
        pizzas_cooked++;

    pthread_exit(NULL);
}

void *waiter(void *arg) {
        sem_wait(&fill);
        sem_wait(&mutex);

        printf("Waiter: I picked up a pizza.\n");
        pizzas_on_shelf--;

        sem_post(&mutex);
        sem_post(&avail);

    pthread_exit(NULL);
}

int main() {
    pthread_t cook_thread, waiter_thread;
```

```
    // Initialize semaphores
    sem_init(&fill, 0, 0);
    sem_init(&avail, 0, MAX_PIZZAS);
    sem_init(&mutex, 0, 1);

    // Create cook and waiter threads
    for (int i = 0; i < NUM_PIZZA; i++){
    pthread_create(&cook_thread, NULL, cook, NULL);
    }
  for (int i = 0; i < NUM_PIZZA; i++){
    pthread_create(&waiter_thread, NULL, waiter, NULL);
  }
    // Wait for the threads to finish
  for (int i = 0; i < NUM_PIZZA; i++){
    pthread_join(cook_thread, NULL);
  }
  for (int i = 0; i < NUM_PIZZA; i++){
    pthread_join(waiter_thread, NULL);
  }
    // Destroy semaphores
    sem_destroy(&fill);
    sem_destroy(&avail);
    sem_destroy(&mutex);

    return 0;
}
```



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