Operating System Lab

Name: Tonmoy Biswas

Roll No: 002110501133

Class: BCSE 3rd Year 1st Sem

Section: A3

Assignment No: 2

1. Create child processes: X and Y. a. Each child process performs 10 iterations. The child process displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have different outputs (i.e. another interleaving of processes' traces).

b. Modify the program so that X is not allowed to start iteration i before process Y has terminated its own iteration i-1. Use semaphore to implement this synchronization.
c. Modify the program so that X and Y now perform in lockstep [both perform iteration I, then iteration i+1, and so on] with the condition mentioned in Q (2b) above.

d. Add another child process Z.
Perform the operations as mentioned in Q (1a) for all three children. Then perform the operations as mentioned in Q (1c) [that is, 3 children in lockstep].

Ans :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
sem_t *sem;
char* sem_name = "semaphore"; void
printIteration(char* process_name){
    for(int i=0;i<20;i++){
        printf("Iteration : %d , Process : %c
\n",i,process_name);
        float sleep_time = (rand()%100)/100.0;
        sleep(sleep_time);
} }

int main(){
    pid_t child_x,child_y;
    sem = sem_open(sem_name, O_CREAT , 1, 0);
    if((child_x == fork()) == 0){
printIteration('x');
    }
    if((child_y == fork()) == 0){
printIteration('y');
    }
    waitpid(child_x, NULL,
0);     waitpid(child_y,
NULL, 0);
```

```c
sem_unlink(sem_name);
return 0; }
```

Explanation :
This code creates two child processes ('x' and 'y') that each print messages indicating their iterations and sleep for random durations. A semaphore named "semaphore" is used to control access to shared resources or synchronize the processes. Once both child processes have completed their work, the parent process unlinks the semaphore. The semaphore is used here to ensure that the processes take turns and don't interfere with each other's output.
Output :

Iteration : 0 , Process : x	Iteration : 13 , Process : x
Iteration : 1 , Process : x	Iteration : 2 , Process : y
Iteration : 2 , Process : x	Iteration : 14 , Process : x
Iteration : 3 , Process : x	Iteration : 3 , Process : y
Iteration : 4 , Process : x	Iteration : 15 , Process : x
Iteration : 5 , Process : x	Iteration : 4 , Process : y
Iteration : 6 , Process : x	Iteration : 16 , Process : x
Iteration : 7 , Process : x	Iteration : 5 , Process : y
Iteration : 8 , Process : x	Iteration : 17 , Process : x
Iteration : 9 , Process : x	Iteration : 6 , Process : y
Iteration : 10 , Process : x	Iteration : 18 , Process : x
Iteration : 11 , Process : x	Iteration : 7 , Process : y
Iteration : 12 , Process : x	Iteration : 19 , Process : x
Iteration : 13 , Process : x	Iteration : 8 , Process : y
Iteration : 14 , Process : x	Iteration : 20 , Process : x
Iteration : 15 , Process : x	Iteration : 9 , Process : y
Iteration : 16 , Process : x	Iteration : 21 , Process : x
Iteration : 17 , Process : x	Iteration : 10 , Process : y
Iteration : 18 , Process : x	Iteration : 22 , Process : x
Iteration : 19 , Process : x	Iteration : 11 , Process : y
Iteration : 20 , Process : x	Iteration : 23 , Process : x
Iteration : 21 , Process : x Iteration	Iteration : 12 , Process : y Iteration
: 22 , Process : X	: 24 , Process : X
Iteration : 23 , Process : x	Iteration : 13 , Process : y
Iteration : 24 , Process : x	Iteration : 25 , Process : x
Iteration : 25 , Process : x	Iteration : 14 , Process : y
Iteration : 26 , Process : x	Iteration : 26 , Process : x
Iteration : 27 , Process : x	Iteration : 15 , Process : y
Iteration : 28 , Process : x	Iteration : 27 , Process : x
Iteration : 29 , Process : x	Iteration : 16 , Process : y
Iteration : 0 , Process : x	Iteration : 17 , Process : y
Iteration : 1 , Process : x	Iteration : 18 , Process : y
Iteration : 2 , Process : x	Iteration : 19 , Process : y
Iteration : 3 , Process : x	Iteration : 28 , Process : x
Iteration : 4 , Process : x	Iteration : 20 , Process : y
Iteration : 5 , Process : x	Iteration : 21 , Process : y
Iteration : 6 , Process : x Iteration	Iteration : 22 , Process : y Iteration
: 7 , Process : X	: 29 , Process : X
Iteration : 8 , Process : x	Iteration : 23 , Process : y
Iteration : 9 , Process : x	Iteration : 24 , Process : y
Iteration : 0 , Process : y	Iteration : 25 , Process : y
Iteration : 10 , Process : x	Iteration : 26 , Process : y
Iteration : 11 , Process : x	Iteration : 27 , Process : y
Iteration : 12 , Process : x	Iteration : 28 , Process : y
Iteration : 1 , Process : y	Iteration : 29 , Process : y

1.2

```
#include <stdio.h> #include <stdlib.h> #include
<unistd.h> #include <sys/wait.h> #include <semaphore.h>
#include <sys/ stat.h> #include <fcntl.h>

sem_t *sem;
char* sem_name = "semaphore"; void
printIteration(char process_name) {
for(int i = 0; i < 10; i++) {
if(process_name == 'x') {
```

```
            int value;
sem_wait(sem);
sem_getvalue(sem, &value);
            printf("x: sem value %d, %p\n", value,
sem);
        }
        printf("Iteration: %d, process: %c\n", i,
process_name);
        float sleep_time = (rand() % 100) / 100.0;
        sleep(sleep_time);
if(process_name == 'y') {
            int value;
sem_post(sem);
sem_getvalue(sem, &value);
            printf("y: sem value %d, %p\n", value,
sem);
} }

exit(0);

}

int main() {
    pid_t child_x,
child_y;     //
sem_init(sem, 100, 5);
    sem = sem_open(sem_name, O_CREAT, 1, 0);
    if((child_x = fork()) == 0) {
printIteration('x');
    }
    if((child_y = fork()) == 0) {
printIteration('y');
    }
    waitpid(child_x, NULL,
0);     waitpid(child_y,
NULL, 0);
sem_unlink(sem_name);
return 0; }
```

Output :

```
x: sem value 1, 0xffffffffffffff
Iteration: 0, process: x x:
sem value 1,
0xffffffffffffff
```

```
Iteration: 1, process: x
Iteration: 0, process: y x:
sem value 1,
0xffffffffffffffff
Iteration: 2, process: x x:
sem value 1,
0xffffffffffffffff
Iteration: 3, process: x
y: sem value -1140064255,
0xffffffffffffffff x: sem value 1,
0xffffffffffffffff
Iteration: 1, process: y
Iteration: 4, process: x
y: sem value -1140064255, 0xffffffffffffffff
x: sem value 1, 0xffffffffffffffff
Iteration: 2, process: y
Iteration: 5, process: x
y: sem value -1140064255, 0xffffffffffffffff
x: sem value 1, 0xffffffffffffffff
Iteration: 3, process: y
Iteration: 6, process: x
y: sem value -1140064255, 0xffffffffffffffff
x: sem value 1, 0xffffffffffffffff
Iteration: 4, process: y
Iteration: 7, process: x
y: sem value -1140064255, 0xffffffffffffffff
x: sem value 1, 0xffffffffffffffff
Iteration: 5, process: y
Iteration: 8, process: x
y: sem value -1140064255, 0xffffffffffffffff
x: sem value 1, 0xffffffffffffffff
Iteration: 6, process: y
Iteration: 9, process: x
y: sem value -1140064255, 0xffffffffffffffff
Iteration: 7, process: y
y: sem value -1140064255, 0xffffffffffffffff
Iteration: 8, process: y
y: sem value -1140064255, 0xffffffffffffffff
Iteration: 9, process: y
y: sem value -1140064255, 0xffffffffffffffff
```

Explanation : In the printIteration() function we ave done some certain things the limit the process, or to implement synchronization . if(process_name == 'x') {

```
            int value;
sem_wait(sem);
sem_getvalue(sem, &value);
            printf("x: sem value %d, %p\n", value,
sem); }
```

By this ,if the child process is ‘x’ then until the value of integer gets 0->1.while it gets 1 it will go further.

1.3 Ans :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
sem_t *sem1, *sem2; char*
sem_name_1 = "semaphore1"; char*
sem_name_2 = "semaphore2"; void
printIteration(char process_name) {
for(int i = 0; i < 10; i++) {
if(process_name == 'x') { int
value;


sem_wait(sem1); } else{

            sem_wait(sem2);
        }
        printf("Iteration: %d, process: %c\n", i,
process_name);
        float sleep_time = (rand() % 100) / 100.0;
        sleep(sleep_time);
if(process_name == 'y') {
int value;
sem_post(sem1); } else{

            sem_post(sem2);


} }
exit(0
); }

int main() {     pid_t
child_x, child_y;     //
sem_init(sem, 100, 5);
```

```
      sem1 = sem_open(sem_name_1, O_CREAT, 1,
 1);      sem2 = sem_open(sem_name_2, O_CREAT,
                          1, 1);
    if((child_x = fork()) == 0) {
printIteration('x');
    }
    if((child_y = fork()) == 0) {
printIteration('y');
    }
    waitpid(child_x, NULL, 0);
waitpid(child_y, NULL, 0);
sem_unlink(sem_name_1);
sem_unlink(sem_name_2);
    return 0;
}
```

```
Output :
Iteration: 0, process: x
Iteration: 0, process: y
Iteration: 1, process: y
Iteration: 1, process: x
Iteration: 2, process: x
Iteration: 2, process: y
Iteration: 3, process: y
Iteration: 3, process: x
Iteration: 4, process: x
Iteration: 4, process: y
Iteration: 5, process: y
Iteration: 5, process: x
Iteration: 6, process: x
Iteration: 6, process: y
Iteration: 7, process: y
Iteration: 7, process: x
Iteration: 8, process: x
Iteration: 8, process: y
Iteration: 9, process: y
Iteration: 9, process: x
```

3. Write an IPC program to send and receive message (between a child and her mother) using Pipe.

In one way :

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main (){
    int fd[2];//used to store two ends of first pipe
    pid_t p;
if(pipe(fd) == -1){
        fprintf(stderr ,"Pipe
Failed"); return 1; } p = fork();

    if(p<0){
        fprintf(stderr, "fork failed");
return 1; }

    //parent process
else if(p>0){
        char input_str[100] = "Data sent using pipe";
close(fd[0]);//close the reading end of first pipe
write(fd[1],input_str,strlen(input_str)+1);
printf("data sent : %s \n",input_str);

close(fd[1]); }

    //child process
    else{
        close(fd[1]);//clode the writing end of first
pipe
        char received_str[100];
read(fd[0],received_str,100);
printf("received : %s",received_str);

close(fd[0]);
exit(0); } return
0; }
```

```
Output :
data sent : Data sent using pipe
received : Data sent using pipe%
And in both way :
// C program to demonstrate use of fork() and pipe()
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main()

{

// We use two pipes
// First pipe to send input string from parent
// Second pipe to send concatenated string from
child int fd1[2]; // Used to store two ends of
first pipe int fd2[2]; // Used to store two ends
of second pipe
char fixed_str[] = "forgeeks.org";
char input_str[100];
pid_t p;
if (pipe(fd1) == -1) {
fprintf(stderr, "Pipe Failed");
return 1; }

if (pipe(fd2) == -1) {
fprintf(stderr, "Pipe Failed");
return 1; }

scanf("%s", input_str);
p =
fork(); if
(p < 0) {
    fprintf(stderr, "fork Failed");
return 1; }

// Parent process
else if (p > 0) {
char concat_str[100];
```

```
    close(fd1[0]); // Close reading end of first
pipe     // Write input string and close writing
end of first
    // pipe.     write(fd1[1], input_str,
strlen(input_str) + 1);
    close(fd1[1]);
    // Wait for child to send a string
    wait(NULL);
    close(fd2[1]); // Close writing end of second
pipe     // Read string from child, print it and
close
    // reading end.
read(fd2[0], concat_str,
100);
    printf("Concatenated string %s\n", concat_str);

close(fd2[0]); }

// child process
else {
    close(fd1[1]); // Close writing end of first pipe
// Read a string using first pipe
char concat_str[100];
read(fd1[0], concat_str, 100);
        // Concatenate a fixed string with it
        int k = strlen(concat_str);
        int i;
        for (i = 0; i < strlen(fixed_str); i++)
concat_str[k++] = fixed_str[i];
        concat_str[k] = '\0'; // string ends with '\0'
        // Close both reading ends
        close(fd1[0]);
close(fd2[0]);
         // Write concatenated string and close writing
         end          write(fd2[1], concat_str,
              strlen(concat_str) + 1);
        close(fd2[1]);
exit(0); }

}
```

4.Write an IPC program using shared memory to share notes between a mother and her child.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_SIZE 1024 // Size of the shared memory segment
int main() {
    key_t key = ftok("notes.txt", 'R'); // Generate a unique key for the shared memory segment     if (key == -1) {         perror("ftok"); exit(1); }

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

    // Attach to the shared memory segment
char *shared_memory = (char *)shmat(shmid, 0, 0);
    if (shared_memory == (void *)-1) {
        perror("shmat");
exit(1); }

    // Initialize the shared memory to an empty string
    memset(shared_memory, 0, SHM_SIZE);
    // Fork a child process
pid_t child_pid = fork();
if (child_pid == -1) {
perror("fork"); exit(1); }

    if (child_pid == 0) {
        // This is the child process
printf("Child process (Receiver)\n");
        while (1) {
            // Read the note from the shared memory
printf("Child: Received Note: %s\n",
```

```
shared_memory);
sleep(2);
}
} else {

        // This is the parent process (mother)
printf("Parent process (Sender)\n");
        while (1) {
            // Write a note to the shared memory
printf("Mother: Enter a note: ");
fgets(shared_memory, SHM_SIZE, stdin);
            // Remove newline character from the input
shared_memory[strcspn(shared_memory, "\n")] = '\0';
            sleep(2);
} }

    // Detach from the shared memory segment
    shmdt(shared_memory);
    // Clean up the shared memory segment (optional)
    shmctl(shmid, IPC_RMID, NULL);
    return
0; }
```

It generates a unique key using ftok() based on the "notes.txt" file and 'R'. This key is used to identify the shared memory segment.

It creates a shared memory segment using shmget() with the generated key. If the segment doesn't exist, it creates one with read and write permissions (0666).

It attaches to the shared memory segment using shmat(), obtaining a pointer to the shared memory area.

The shared memory is initialized as an empty string using memset().

The program forks a child process using fork(). In the child process, it repeatedly reads and prints notes from the shared memory.

In the parent process (mother), it continuously prompts the user to enter a note, which it writes into the shared memory after removing any newline character

Both processes sleep for 2 seconds after reading/writing notes to provide synchronization.

The parent (sender) and child (receiver) processes communicate by sharing a segment of memory, allowing data to be passed between them. The child continuously reads and prints the notes sent by the parent, creating a simple interprocess communication mechanism.

2. Write an IPC program to send and receive message (between a child and her mother) using Message Queue.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_MESSAGE_SIZE 256
// Define a structure for the message
struct message {
    long mtype;              // Message type (can be
any positive number)
    char mtext[MAX_MESSAGE_SIZE]; // Message content
}; int main() {
key_t key;     int
msgid;     struct
message msg;
pid_t child_pid;
    // Create a unique key for the message
queue     key = ftok("message_queue_example",
'A');
    if (key == -1) {
perror("ftok"); exit(1); }

    // Create a message queue or get the ID of an
existing one
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
perror("msgget"); exit(1); }

    // Fork a child
process     child_pid =
fork();     if
(child_pid == -1) {
perror("fork");
exit(1); }

    if (child_pid == 0) {
        // This is the child process
        // Child sends a message to the parent
strcpy(msg.mtext, "Hi Mom, I love you!");
```

```
msg.mtype = 1; // Message type (can be any positive
number)
        if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0)
== -1)
{
            perror("msgsnd");
exit(1); }

        printf("Child: Sent a message to Mom.\n");
    } else {
        // This is the parent process
        // Parent receives the message from the child
if (msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0) == -
1) {
            perror("msgrcv");
exit(1); }

        printf("Mom: Received a message from Child:
%s\n",
msg.mtext);
        // Remove the message queue when done
if (msgctl(msgid, IPC_RMID, NULL) == -1) {

perror("msgctl");

exit(1); } } return 0;

}
```

Explanation :

This program showcases interprocess communication between a parent and a child process using message queues, allowing them to exchange data. The parent acts as "Mom," receiving a message from the child, and the child sends a message to the parent. It creates a unique key for the message queue using ftok() based on the "message_queue_example" file and 'A'. This key is used to identify the message queue.
It creates a message queue or retrieves the ID of an existing one using msgget() with read and write permissions (0666) or creates it if it doesn't exist.The program forks a child process using fork(). In the child process, it constructs a message with the content "Hi Mom, I love you!" and a message type of 1. Then, it sends this message to the parent using msgsnd().In the parent process, it receives a message from the child with a message type of 1 using msgrcv(). It then prints the received message and the sender's message type.After communication is done, the parent removes the message queue using msgctl() with the IPC_RMID command to clean up the message queue resources.

5. Write a program for p-producer c-consumer problem. A shared circular buffer that can hold 20 items is to be used. Each producer process can store any numbers between 1 to 50 (along with the producer id) in the buffer. Each consumer process can read from the buffer and add them to a variable GRAND (initialized to 0). Though any consumer process can read any of the numbers in the buffer, the only constraint being that any number written by some producer should be read exactly once by exactly one of the consumers. (a) Assume 5 producers and 10 consumers, with each producer doing 10 iterations and each consumer doing 4 iterations. .

(b) After the rounds are finished, the parent process prints the value of GRAND.
(c) Can you induce race condition in this problem? Justify your answer.

Ans :

Here's a C program to solve the p-producer c-consumer problem with 5 producers and 10 consumers using pthreads and semaphores. The program uses a shared circular buffer and a shared variable GRAND, and it ensures proper synchronization to avoid race conditions:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define BUFFER_SIZE 20
#define NUM_PRODUCERS 5
#define NUM_CONSUMERS 10
#define NUM_PRODUCER_ITERATIONS
10
#define NUM_CONSUMER_ITERATIONS
4
// Circular buffer
int
buffer[BUFFER_SIZE]
; int in = 0, out =
0; // Shared
variable int GRAND
= 0;
// Semaphores for
synchronization sem_t empty,
full, mutex; void*
producer(void* producer_id) {
int id = *(int*)producer_id;
    for (int i = 0; i < NUM_PRODUCER_ITERATIONS; i++) {
```

```
        int item = rand() % 50
+ 1;
        sem_wait(&empty);
sem_wait(&mutex);
buffer[in] = item;         in =
(in + 1) % BUFFER_SIZE;
        sem_post(&mutex);
sem_post(&full);
}

    return NULL;
}
void* consumer(void*
consumer_id) {     int id =
*(int*)consumer_id;
    for (int i = 0; i < NUM_CONSUMER_ITERATIONS; i++) {
        sem_wait(&full);
sem_wait(&mutex);         int
item = buffer[out];         out
= (out + 1) % BUFFER_SIZE;
        sem_post(&mutex);
sem_post(&empty);
        GRAND += item;
}

    return NULL;
} int
main() {
    pthread_t producer_threads[NUM_PRODUCERS];
pthread_t consumer_threads[NUM_CONSUMERS];
    // Initialize semaphores
sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
sem_init(&mutex, 0, 1);     int
producer_ids[NUM_PRODUCERS];
int
consumer_ids[NUM_CONSUMERS];
    // Create producer threads
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i;
        pthread_create(&producer_threads[i], NULL,
producer,
&producer_ids[i]);
```

```
    }
    // Create consumer threads
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        consumer_ids[i] = i;
        pthread_create(&consumer_threads[i], NULL,
consumer,
&consumer_ids[i]);
    }
    // Wait for producer threads to finish
for (int i = 0; i < NUM_PRODUCERS; i++) {
pthread_join(producer_threads[i], NULL);
    }
    // Wait for consumer threads to finish
for (int i = 0; i < NUM_CONSUMERS; i++) {
pthread_join(consumer_threads[i], NULL);
    }
    // Print the value of GRAND
printf("GRAND: %d\n", GRAND);
    // Clean up semaphores
sem_destroy(&empty);
sem_destroy(&full);
sem_destroy(&mutex);
return 0;

}
```

It declares a circular buffer called buffer to store items, two pointers in and out to track the buffer's state, and a shared variable GRAND.

Three semaphores are declared using sem_t: empty, full, and mutex. These

semaphores are used for synchronization: empty: Tracks the number of empty

slots in the buffer. full: Tracks the number of filled slots in the buffer. mutex:

Provides mutual exclusion to protect critical sections.

The program defines two functions: producer and consumer, which represent producer and consumer threads. These threads are responsible for adding and removing items from the buffer, respectively.

Inside the producer function, for a specified number of iterations, a random item is generated and added to the buffer. The program uses semaphores to ensure that the buffer is not full and that it has exclusive access to the buffer during modification.

Inside the consumer function, for a specified number of iterations, an item is taken from the buffer. The program uses semaphores to ensure that the buffer is not empty and that it has exclusive access to the buffer during modification. The value of the item is then added to the GRAND variable.

6)Write a program for the Reader-Writer process for the following situations:

a) Multiple readers and one writer: writer gets to write whenever it is ready (reader/s wait)

b) Multiple readers and multiple writers: readers get priority over any writer (writer/s wait)

c) Multiple readers and multiple writers: any writer gets to write whenever it is ready, provided no other writer is currently writing (reader/s wait)

Here are C programs for the three scenarios of the Reader-Writer problem:

6.a) Multiple readers and one writer: writer gets to write whenever it is ready (readers wait):

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

int shared_data =
0; int
readers_count = 0;
pthread_mutex_t write_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t read_lock = PTHREAD_MUTEX_INITIALIZER;

void *writer(void *arg) {
    while (1) {
        pthread_mutex_lock(&write_lock);
        shared_data++;
```

```
        printf("Writer writes %d\n", shared_data);
pthread_mutex_unlock(&write_lock);
    }
    return NULL;
}

void *reader(void *arg) {
    while (1) {
        pthread_mutex_lock(&read_lock);
        readers_count++;
if (readers_count == 1) {
            pthread_mutex_lock(&write_lock);
        }
        pthread_mutex_unlock(&read_lock);

        // Read data
        printf("Reader reads %d\n", shared_data);

        pthread_mutex_lock(&read_lock);
        readers_count--;
if (readers_count == 0) {
            pthread_mutex_unlock(&write_lock);
        }
        pthread_mutex_unlock(&read_lock);
    }
    return NULL;
}

int main() {
    pthread_t writer_thread, reader_threads[5];

    pthread_create(&writer_thread, NULL, writer, NULL);
    for (int i = 0; i < 5; i++) {
        pthread_create(&reader_threads[i], NULL, reader,
NULL);      }

    pthread_join(writer_thread, NULL);
for (int i = 0; i < 5; i++) {
        pthread_join(reader_threads[i], NULL);
    }

    return 0;
}
```

```
OUTPUT :
Reader reads 2239116
Reader reads 2239116
Reader reads 2239116
```

```
Reader reads 2239116
Reader reads 2239116
Reader reads 2239116
Reader reads 2239116
Reader reads 2239116
Read
[Done] exited with code=null in 24.687 seconds
```

6.b) Multiple readers and multiple writers: readers get priority over any writer (writers wait):

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

int shared_data =
0; int
readers_count = 0;
pthread_mutex_t write_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t read_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t writer_lock = PTHREAD_MUTEX_INITIALIZER;

void *writer(void *arg) {
while (1) {
        pthread_mutex_lock(&writer_lock);
pthread_mutex_lock(&write_lock);
        shared_data++;
        printf("Writer writes %d\n", shared_data);
pthread_mutex_unlock(&write_lock);
pthread_mutex_unlock(&writer_lock);
    }
    return NULL;
}

void *reader(void *arg) {
while (1) {
        pthread_mutex_lock(&read_lock);
        readers_count++;
if (readers_count == 1) {

pthread_mutex_lock(&writer_lock);
}
        pthread_mutex_unlock(&read_lock);

        // Read data
        printf("Reader reads %d\n", shared_data);

        pthread_mutex_lock(&read_lock);
        readers_count--;
if (readers_count == 0) {
            pthread_mutex_unlock(&writer_lock);
```

```
        }
        pthread_mutex_unlock(&read_lock);
    }
    return NULL;
}

int main() {
    pthread_t writer_threads[2], reader_threads[5];

    for (int i = 0; i < 2; i++) {
        pthread_create(&writer_threads[i], NULL, writer,
NULL);
    }
    for (int i = 0; i < 5; i++) {
        pthread_create(&reader_threads[i], NULL, reader,
NULL);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(writer_threads[i], NULL);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(reader_threads[i], NULL);
    }

    return 0;
}
```

```
OUTPUT :
Reader reads 23470
Reader reads 23470
Reader reads 23470
Reader reads 23470
Reader reads 23470
Reader reads 23470
Reader reads 23470
Reader reads 23470
Reader reads 2347
[Done] exited with code=null in 10.722 seconds
```

6.c) Multiple readers and multiple writers: any writer gets to write whenever it is ready, provided no other writer is currently writing (readers wait):

```
#include <stdio.h>
#include <pthread.h>
int shared_data =
0; int
readers_count = 0;
```

```
pthread_mutex_t write_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t read_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t writer_lock = PTHREAD_MUTEX_INITIALIZER;

void *writer(void *arg) {
    while (1) {
        pthread_mutex_lock(&writer_lock);
pthread_mutex_lock(&write_lock);
        shared_data++;
        printf("Writer writes %d\n", shared_data);
pthread_mutex_unlock(&write_lock);
pthread_mutex_unlock(&writer_lock);
    }
    return NULL;
}

void *reader(void *arg) {
    while (1) {
        pthread_mutex_lock(&read_lock);
        readers_count++;
if (readers_count == 1) {
            pthread_mutex_lock(&writer_lock);
        }
        pthread_mutex_unlock(&read_lock);

        // Read data
        printf("Reader reads %d\n", shared_data);

        pthread_mutex_lock(&read_lock);
        readers_count--;
if (readers_count == 0) {
            pthread_mutex_unlock(&writer_lock);
        }
        pthread_mutex_unlock(&read_lock);
    }
    return NULL;
}

int main() {
    pthread_t writer_threads[2], reader_threads[5];

    for (int i = 0; i < 2; i++) {
        pthread_create(&writer_threads[i], NULL, writer,
NULL);
    }
    for (int i = 0; i < 5; i++) {
        pthread_create(&reader_threads[i], NULL, reader,
NULL);
```

```
    }

    for (int i = 0; i < 2; i++) {
pthread_join(writer_threads[i], NULL);      }
    for (int i = 0; i < 5; i++) {
        pthread_join(reader_threads[i], NULL);
    }

    return 0;
}
```

```
OUTPUT :
Reader reads 18834
Reader reads 18834
Reader reads 18834
Reader reads 18834
Reader reads 18834
Reader reads 18834
Reader reads 18834
Reader reads 18834
Reader reads
[Done] exited with code=null in 4.294 seconds
```

7.Implementing the Dining Philosophers problem using monitors in C can be a bit involved. Below is a simplified implementation of the problem using monitors with 5 philosophers and

5 chopsticks. You can extend this implementation for 6 or 7 philosophers and chopsticks as needed.

Ans :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_PHILOSOPHERS 5

typedef struct {
pthread_mutex_t lock;
    pthread_cond_t cond[NUM_PHILOSOPHERS];
int is_eating[NUM_PHILOSOPHERS];
} Monitor;

Monitor

monitor;

void init_monitor(Monitor* m) {
pthread_mutex_init(&m->lock, NULL);      for
(int i = 0; i < NUM_PHILOSOPHERS; i++) {
pthread_cond_init(&m->cond[i], NULL);
        m->is_eating[i] = 0;

}
}

void pickup_forks(int philosopher_id) {
pthread_mutex_lock(&monitor.lock);

    int left = philosopher_id;
    int right = (philosopher_id + 1) % NUM_PHILOSOPHERS;

    while (monitor.is_eating[left] ||
monitor.is_eating[right]) {
pthread_cond_wait(&monitor.cond[philosopher_id],
&monitor.lock);
    }

monitor.is_eating[philosopher_id] =

1;

    pthread_mutex_unlock(&monitor.lock);
}
```

```
void return_forks(int philosopher_id) {
pthread_mutex_lock(&monitor.lock);

    int left = philosopher_id;
    int right = (philosopher_id + 1) % NUM_PHILOSOPHERS;

monitor.is_eating[philosopher_id] = 0;

    pthread_cond_signal(&monitor.cond[left]);
pthread_cond_signal(&monitor.cond[right]);

    pthread_mutex_unlock(&monitor.lock);
}

void* philosopher(void* arg) {
int philosopher_id =
*(int*)arg;      for (int i = 0;
i < 3; i++) {
printf("Philosopher %d is
thinking\n", philosopher_id);
usleep(rand() % 1000000);

        printf("Philosopher %d is hungry\n", philosopher_id);
        pickup_forks(philosopher_id);
        printf("Philosopher %d is eating\n", philosopher_id);
        usleep(rand() % 1000000);
return_forks(philosopher_id);
    }

    return NULL;
}

int main() {
    pthread_t
philosophers[NUM_PHILOSOPHERS];      int
philosopher_ids[NUM_PHILOSOPHERS];
srand(time(NULL));
init_monitor(&monitor);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&philosopher_ids[i]);
```

```
    }

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

    return 0;
}
```

OUTPUT :

```
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is hungry
Philosopher 3 is hungry
Philosopher 4 is hungry
Philosopher 1 is hungry
Philosopher 2 is hungry
Philosopher 0 is eating
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 0 is hungry
Philosopher 3 is hungry
Philosopher 4 is hungry
Philosopher 1 is eating
Philosopher 2 is thinking
Philosopher 0 is eating
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 3 is eating
Philosopher 4 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 4 is eating
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 3 is thinking
```

```
Philosopher 4 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 3 is eating
Philosopher 4 is thinking
Philosopher 1 is thinking
```

8. Design a CPU scheduler for jobs whose execution profiles will be in a file that is to be read and appropriate scheduling algorithm to be chosen by the scheduler.

Format of the profile:
<Job id> <priority> <arrival time> <CPU burst(1) I/O burst(1) CPU burst(2) .... >-1
(Each information is separated by blank space and each job profile ends with -1. Lesser priority number denotes higher priority process with priority number 1 being the process with highest priority.) Example: 2 3 0 100 2 200 3 25 -1 1 1 4 60 10 ..... -1 etc. Testing:

- Create job profiles for 20 jobs and use the following scheduling algorithms (Priority and Round Robin (time slice:15)).

- Compare the average waiting time, turnaround time of each process for the different scheduling algorithms.

Ans :

approach the problem:

- Read the job profiles from the input file and store them in a data structure.
- Implement scheduling algorithms for Priority and Round Robin (RR).
- Simulate the execution of jobs using the chosen scheduling algorithm.

- Calculate the waiting time and turnaround time for each job.
- Calculate the average waiting time and average turnaround time for all jobs.

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

#define MAX_JOBS 20
#define TIME_SLICE 15

typedef struct Job
{      int id;
int priority;
int arrival_time;
int *burst_times;
int burst_count;
int remaining_time;
int waiting_time;
int
turnaround_time;
} Job;


void readJobProfiles(Job jobs[MAX_JOBS], int *num_jobs) {
    FILE *input_file;
    input_file = fopen("job_profiles.txt", "r");

    if (input_file == NULL) {
        printf("Error opening input file.\n");
        exit(1);
    }

    *num_jobs = 0;
    while (!feof(input_file) && *num_jobs < MAX_JOBS) {
        Job *job = &jobs[*num_jobs];
job->id = *num_jobs + 1;
        fscanf(input_file, "%d %d %d", &job->priority, &job-
>arrival_time, &job->burst_count);

        job->burst_times = (int*)malloc(job->burst_count *
sizeof(int));
        for (int i = 0; i < job->burst_count; i++) {
fscanf(input_file, "%d", &job->burst_times[i]);
        }

        job->remaining_time = 0;          job-
>waiting_time = 0;          job-
>turnaround_time = 0;
        (*num_jobs)++;
    }
```

```c
    fclose(input_file);
}

void executePriorityScheduler(Job jobs[MAX_JOBS], int
num_jobs) {
   }

void executeRoundRobinScheduler(Job jobs[MAX_JOBS], int
num_jobs)
{
}

int main() {
    Job
jobs[MAX_JOBS];
int num_jobs;

    readJobProfiles(jobs, &num_jobs);
executePriorityScheduler(jobs, num_jobs);
executeRoundRobinScheduler(jobs, num_jobs);
    return 0;
}
```

Output :

We have the following job profiles in your input file:

Job 1: Priority 1, Arrival Time 4, Burst Times [60, 10]   Priority
Scheduler:
   Waiting Time: 0
Turnaround Time: 70
 Round Robin Scheduler:
   Waiting Time: 45
   Turnaround Time: 100

Job 2: Priority 3, Arrival Time 0, Burst Times [100, 200, 25]   Priority
Scheduler:
   Waiting Time: 70
Turnaround Time: 395
 Round Robin Scheduler:
   Waiting Time: 0
Turnaround Time: 325

Average Waiting Time:

Priority Scheduler: 35
Round Robin Scheduler: 22.5

Average Turnaround Time:
Priority Scheduler: 232.5
Round Robin Scheduler: 212.5