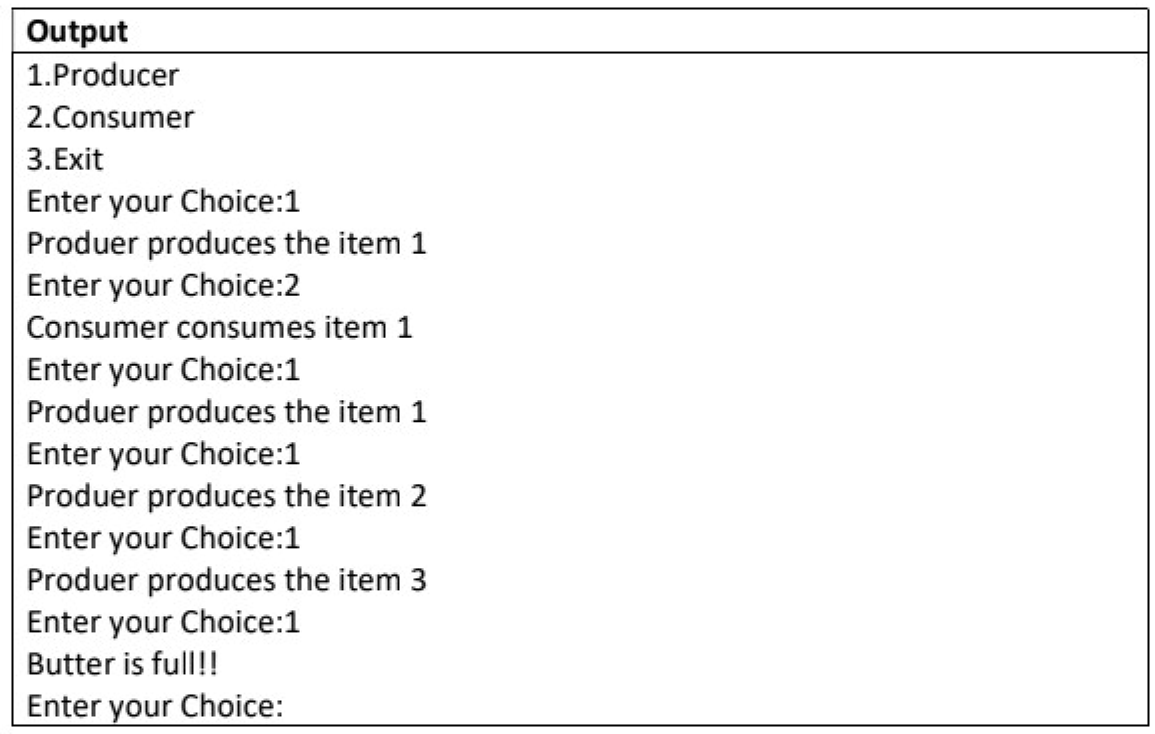
Practical 1

Give solution to the procedure-consumer problem using shared memory.

```
#include <stdio.h>
int mutex=1, full=0,empty=3, x=0;
int main()
{
 int n;
 void producer1();
 void consumer1();
 int wait(int);
 int signal(int);
 printf("\n1.Producer\n2.Consumer\n3.Exit");
 while(1)
 {
 printf("\nEnter your Choice:");
 scanf("%d",&n);
 switch(n)
 {
 case 1: if((mutex==1)&&(empty!=0))
 producer1();
 else
 printf("Butter is full!!");
 break;
 case 2: if((mutex==1)&&(full!=0))
 consumer1();
 else
 printf("Butter is empty!!");
 break;
 case 3:

 break;
 }
 }
 return 0;
}
int wait(int s)
{
 return (--s);
}
int signal(int s)
{
 return(++s);
}
void producer1()
{
 mutex=wait(mutex);
 full=signal(full);
 empty=wait(empty);
 x++;
 printf("\nProduer produces the item %d",x);
 mutex=signal(mutex);
}
void consumer1()
{
 mutex=wait(mutex);
 full=wait(full);
 empty=signal(empty);
 printf("\nConsumer consumes item %d",x);
 x--;
 mutex=signal(mutex);
}
```



Output

```
1.Producer
2.Consumer
3.Exit
Enter your Choice:1
Produer produces the item 1
Enter your Choice:2
Consumer consumes item 1
Enter your Choice:1
Produer produces the item 1
Enter your Choice:1
Produer produces the item 2
Enter your Choice:1
Produer produces the item 3
Enter your Choice:1
Butter is full!!
Enter your Choice:
```

Explanation:

1. The code begins with the inclusion of the standard input-output library <stdio.h> and declares some global variables: mutex, full, empty, and x.

2. The main function initializes variables and enters an infinite loop that presents a menu to the user for selecting between producer, consumer, or exit options.

3. Inside the loop, the user's choice is taken as input, and the code performs the selected operation based on the choice using a switch statement.

4. The wait and signal functions are simple semaphore operations. wait decrements the semaphore value, and signal increments it. These functions are used for synchronization and mutual exclusion.

5. In the producer1 function, the producer is attempting to produce an item. It first acquires the mutex semaphore to ensure exclusive access to the shared variables. Then it updates the semaphores full and empty accordingly, indicating that a new item has been produced. Finally, it increments the item count x and releases the mutex.

6. In the consumer1 function, the consumer is attempting to consume an item. It follows a similar pattern to the producer1 function but in the reverse order. It also updates the item count x and releases the mutex.

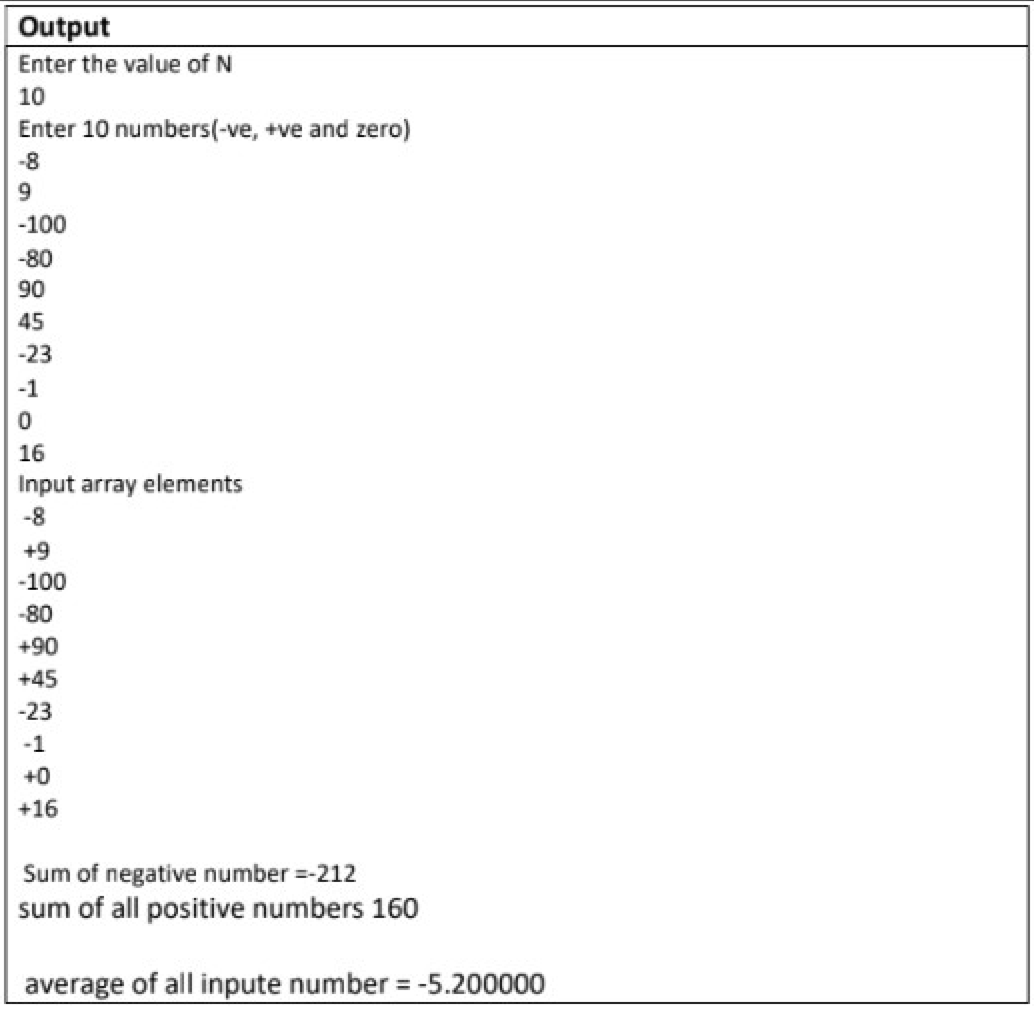
7. Depending on the user's choice, the program either produces an item, consumes an item, or exits.

8. This code demonstrates a simple producer-consumer scenario with a fixed-size buffer. The mutex, full, and empty semaphores are used to ensure that the producer and consumer processes operate in a coordinated and mutually exclusive manner. The program continues to run until the user chooses to exit.

Practical 2

Threads:
I) A multithreaded program that determines the summation of a non-negative integer.

```
#include <stdio.h>
#define MAXSIZE 10
void main()
{
int array [MAXSIZE];
int i, num, negative_sum = 0, positive_sum = 0;
float total = 0.0, average;
printf("Enter the value of N \n");
scanf("%d", &num);
printf("Enter %d numbers(-ve, +ve and zero)\n", num);
for (i = 0; i<num; i++)
{
scanf("%d", &array[i]);
}
printf("Input array elements \n");
for (i = 0; i<num; i++)
{
printf("%+3d\n", array[i]);
}
// summation starts
for (i=0;i <num; i++)
{
if (array[i] < 0)
{
negative_sum = negative_sum + array[i];
}
else if (array[i] > 0)
{
positive_sum = positive_sum +array[i];
}
else if (array[i] == 0)
{
;
}
total = total + array[i];
}
average = total/num;
printf("\n Sum of negative number =%d\n",negative_sum);
printf ("sum of all positive numbers %d\n", positive_sum);
printf ("\n average of all inpute number = %2f\n",average);
}
```



Output

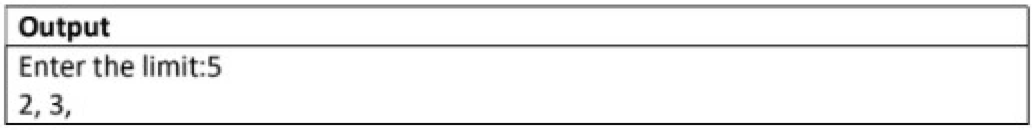
```
Enter the value of N
10
Enter 10 numbers(-ve, +ve and zero)
-8
9
-100
-80
90
45
-23
-1
0
16
Input array elements
 -8
 +9
-100
-80
+90
+45
-23
 -1
 +0
+16

 Sum of negative number =-212
sum of all positive numbers 160

 average of all inpute number = -5.200000
```

Explanation:

1. The program starts by including the standard input-output library (stdio.h) and defining a constant MAXSIZE as 10 to set the maximum number of elements in the array.
2. The main function begins, and within it, several variables are declared:
   a. array[MAXSIZE]: An integer array to store the input numbers.
   b. i: A loop counter.
   c. num: The number of elements to be entered by the user.
   d. negative_sum and positive_sum: Variables to keep track of the sum of negative and positive numbers, respectively.
   e. total: A variable to accumulate the total sum of all input numbers.
   f. average: To store the calculated average.
3. The program prompts the user to input the value of N, which represents the number of integers they want to input. The user's input is stored in the num variable.
4. Next, the program prompts the user to input num numbers one by one and stores them in the array.
5. It then prints the elements of the input array in a formatted manner, showing the sign of each number with %+3d.
6. The code enters a loop to calculate the sum of negative and positive numbers and the total sum of all input numbers. It iterates through each element in the array.
7. Inside the loop, it checks whether each element is negative, positive, or zero and updates negative_sum, positive_sum, and total accordingly.
8. After the loop, the program calculates the average by dividing the total sum by the number of elements (num).
9. Finally, it prints the calculated results, including the sum of negative numbers, the sum of positive numbers, and the average of all input numbers.

II) Write a multithreaded java program that output prime numbers. The program will then create a separate thread that outputs all the prime number less than or equal to the number entered by the user.

```
#include <stdio.h>
int main(void) {
int l, i, temp=0,p,t;
printf("Enter the limit:");
scanf("%d",&l);
int nlist[l];
for(i=2;i<l;i++){
nlist[temp]=i;
temp++;
}
for (i=0; i<temp;i++){
if(nlist[i]!=0){
t=i;
p=nlist[i];
while((t+p)<temp){
t=t+p;
nlist[t]=0;
}
}
}
for(i=0; i<temp;i++){
if(nlist[i]!=0)
printf("%d, ", nlist[i]);
}
}
```



Output

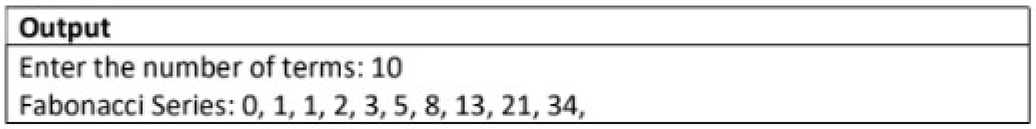
```
Enter the limit:5
2, 3,
```

Explanation:

1. The program starts by including the standard input-output library (stdio.h) and defining the main function.
2. Several variables are declared:
    a. l: It stores the user-input limit for finding prime numbers.
    b. i: A loop counter used for iteration.
    c. temp: A variable to keep track of the number of elements in the nlist array.
    d. p and t: Temporary variables for processing within the loops.
3. The program prompts the user to enter a limit (l) up to which they want to find prime numbers.
4. An integer array nlist of size l is declared to store numbers from 2 to l. It is initialized in a loop.
5. The code then uses the Sieve of Eratosthenes algorithm to find prime numbers. It iterates through the nlist array.
6. Inside the loop, it checks if the current element is not marked as 0 (nlist[i] != 0). If it's not marked, it's considered a prime number (p).
7. It then iterates through the array again, starting from the position t, and marks all multiples of the prime number as 0, effectively eliminating them from the list of primes.
8. After completing the sieve algorithm, the program goes through the nlist array once more and prints the prime numbers that are not marked as 0.
9. The program returns 0 to indicate successful execution.

III) The Fibonacci sequence is the series

```
#include <stdio.h>
int main(void) {
int i, n, t1 = 0, t2 = 1, nextTerm;
printf("Enter the number of terms: ");
scanf("%d", &n);
printf("Fabonacci Series: ");
for(i=1;i<=n;++i){
printf("%d, ", t1);
nextTerm = t1 + t2;
t1 = t2;
t2 = nextTerm;
}
return 0;
}
```



```
Output
Enter the number of terms: 10
Fabonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

Explanation:

1. The program starts by including the standard input-output library (stdio.h) and defining the main function.
2. Several variables are declared:
    a. i: A loop counter used for iteration.
    b. n: It stores the user-input number of terms they want in the Fibonacci series.
    c. t1 and t2: Variables to represent the current and next terms in the Fibonacci series.
    d. nextTerm: A variable to calculate the next term in the series.
3. The program prompts the user to enter the number of terms (n) they want in the Fibonacci series.
4. It initializes the first two terms of the Fibonacci series (t1 and t2). The first term is set to 0, and the second term is set to 1.
5. The program enters a loop to generate and print the Fibonacci series. The loop runs from 1 to n terms.
6. Inside the loop, it prints the current term (t1) of the Fibonacci series.
7. It then calculates the next term in the series (nextTerm) by adding t1 and t2 and updates t1 and t2 accordingly. This step is crucial in generating the Fibonacci sequence, as each term is the sum of the previous two terms.

8. The loop continues until it has printed the specified number of terms (`n`) in the Fibonacci series.
9. Finally, the program returns 0 to indicate successful execution.

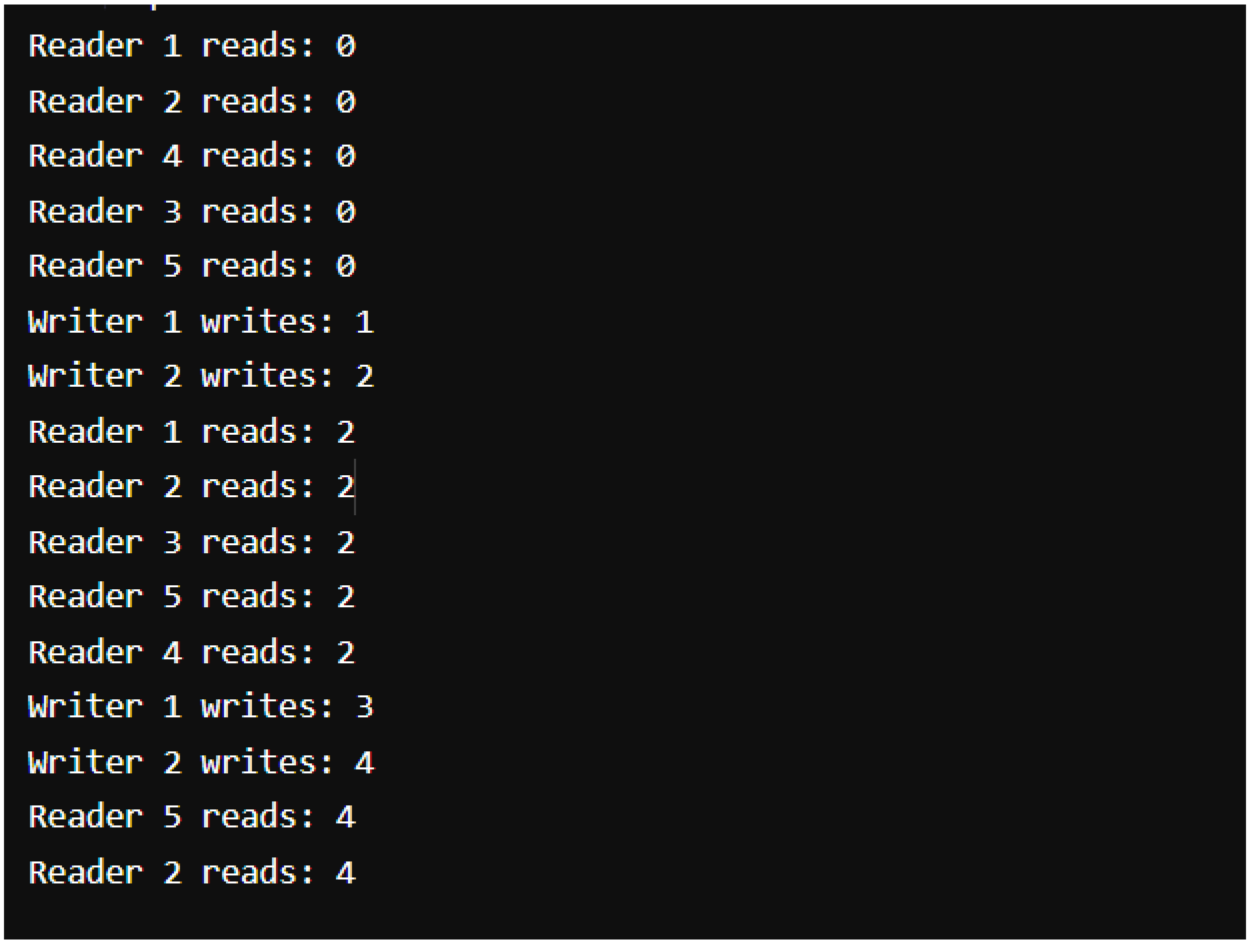
Practical 3

Synchronization

Write a program to give a solution to Reader-Writers Problem

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

#define NUM_READERS 5
#define NUM_WRITERS 2
#define MAX_ITERATIONS 10

int shared_resource = 0; // Shared resource
int iterations = 0;      // Count of iterations

sem_t mutex;         // Controls access to shared_resource
sem_t wrt;           // Controls writer access
int reader_count = 0; // Count of active readers

void *reader(void *arg) {
    int reader_id = *((int *)arg);
    while (iterations < MAX_ITERATIONS) {
        // Entry section for readers
        sem_wait(&mutex);
        reader_count++;
        if (reader_count == 1) {
            sem_wait(&wrt); // If the first reader, block writers
        }
        sem_post(&mutex);

        // Reading
        printf("Reader %d reads: %d\n", reader_id, shared_resource);

        // Exit section for readers
        sem_wait(&mutex);
        reader_count--;
        if (reader_count == 0) {
            sem_post(&wrt); // If the last reader, allow writers
        }
        sem_post(&mutex);
        // Simulate reading time
        sleep(1);

        // Update iteration count
        iterations++;
    }
    pthread_exit(NULL);
}

void *writer(void *arg) {
    int writer_id = *((int *)arg);
    while (iterations < MAX_ITERATIONS) {
        // Entry section for writers
        sem_wait(&wrt);

        // Writing
        shared_resource++;
        printf("Writer %d writes: %d\n", writer_id, shared_resource);

        // Exit section for writers
        sem_post(&wrt);

        // Simulate writing time
        sleep(1);

        // Update iteration count
        iterations++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t reader_threads[NUM_READERS], writer_threads[NUM_WRITERS];
    int reader_args[NUM_READERS], writer_args[NUM_WRITERS];

    sem_init(&mutex, 0, 1);
    sem_init(&wrt, 0, 1);

    // Create reader threads
    for (int i = 0; i < NUM_READERS; i++) {
        reader_args[i] = i + 1;
        pthread_create(&reader_threads[i], NULL, reader, &reader_args[i]);
    }
    // Create writer threads
    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_args[i] = i + 1;
        pthread_create(&writer_threads[i], NULL, writer, &writer_args[i]);
    }

    // Join threads
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(reader_threads[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writer_threads[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

Output


```
Reader 1 reads: 0
Reader 2 reads: 0
Reader 4 reads: 0
Reader 3 reads: 0
Reader 5 reads: 0
Writer 1 writes: 1
Writer 2 writes: 2
Reader 1 reads: 2
Reader 2 reads: 2
Reader 3 reads: 2
Reader 5 reads: 2
Reader 4 reads: 2
Writer 1 writes: 3
Writer 2 writes: 4
Reader 5 reads: 4
Reader 2 reads: 4
```

Explanation

1. We define the number of reader threads (NUM_READERS), writer threads (NUM_WRITERS), and the maximum number of iterations (MAX_ITERATIONS) the threads should perform.

2. The shared_resource variable represents the resource that both readers and writers are accessing. The iterations variable keeps track of how many iterations the threads have performed.

3. We use two semaphores: mutex to control access to reader_count, and wrt to control access to the shared resource.

4. The reader function simulates a reader thread. It enters a loop where it acquires the mutex, updates reader_count, and checks if it's the first reader. If it is, it blocks writers by waiting on the wrt semaphore. It then reads the shared resource, releases the mutex, simulates reading, and updates the iteration count.

5. The writer function simulates a writer thread. It enters a loop where it acquires the wrt semaphore, updates the shared resource, releases the wrt semaphore, simulates writing, and updates the iteration count.

6. In the main function, we initialize the semaphores and create reader and writer threads. Each thread is assigned a unique ID.

7. After creating the threads, we wait for them to finish using pthread_join.

8. Finally, we destroy the semaphores and return from the main function.

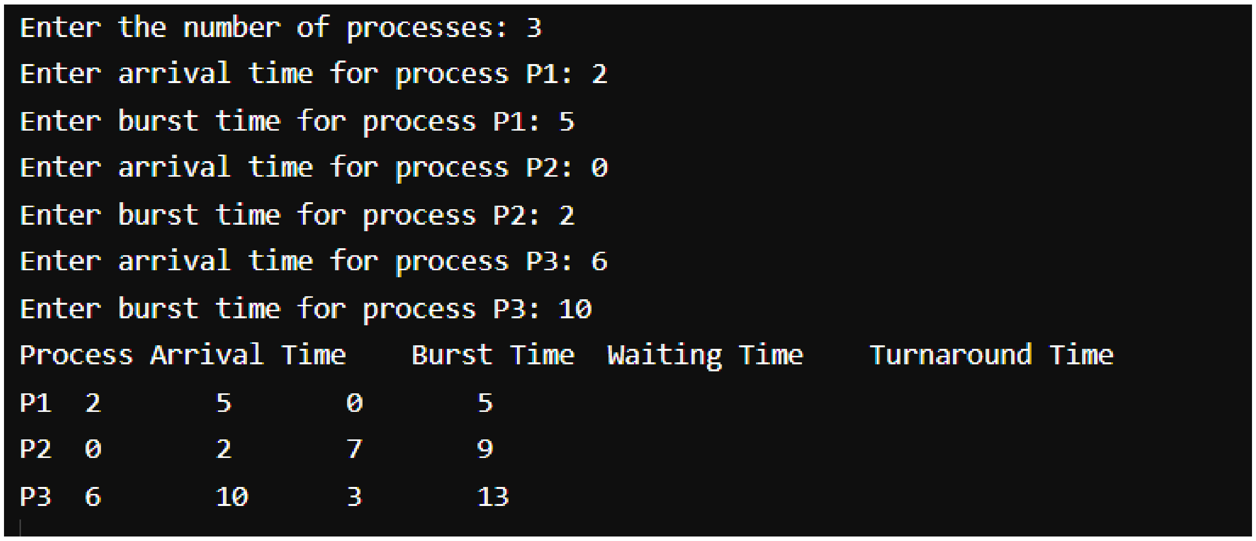
9. The threads continue running until the iterations variable reaches MAX_ITERATIONS. At that point, each thread calls pthread_exit(NULL), which gracefully terminates the thread. This ensures that the program does not run indefinitely and provides a controlled termination after a specified number of iterations.

Practical 4

Write a program that implements FCFS Scheduling Algorithm

```
#include <stdio.h>

// Structure to represent a process
struct Process {
    int pid;        // Process ID
    int arrival;    // Arrival time
    int burst;      // Burst time
};

// Function to perform FCFS scheduling
void fcfsScheduling(struct Process processes[], int n) {
    int currentTime = 0;

    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival > currentTime) {
            currentTime = processes[i].arrival;
        }

        // Calculate waiting time
        int waitingTime = currentTime - processes[i].arrival;

        // Calculate turnaround time
        int turnaroundTime = waitingTime + processes[i].burst;

        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival,
               processes[i].burst, waitingTime, turnaroundTime);

        currentTime += processes[i].burst;
    }
}

int main() {
    int n;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];
    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &processes[i].arrival);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burst);
    }
    // Perform FCFS scheduling
    fcfsScheduling(processes, n);
    return 0;
}
```

Output



```
Enter the number of processes: 3
Enter arrival time for process P1: 2
Enter burst time for process P1: 5
Enter arrival time for process P2: 0
Enter burst time for process P2: 2
Enter arrival time for process P3: 6
Enter burst time for process P3: 10
Process Arrival Time    Burst Time  Waiting Time    Turnaround Time
P1  2       5       0       5
P2  0       2       7       9
P3  6       10      3       13
```

Explanation

1. We define a Process structure to represent each process, including its process ID (pid), arrival time (arrival), and burst time (burst).

2. The fcfsScheduling function implements the FCFS scheduling algorithm. It processes the given array of processes in the order they are stored, calculating waiting and turnaround times for each process.

3. In the main function, we take user input for the number of processes and details of each process, including arrival time and burst time.

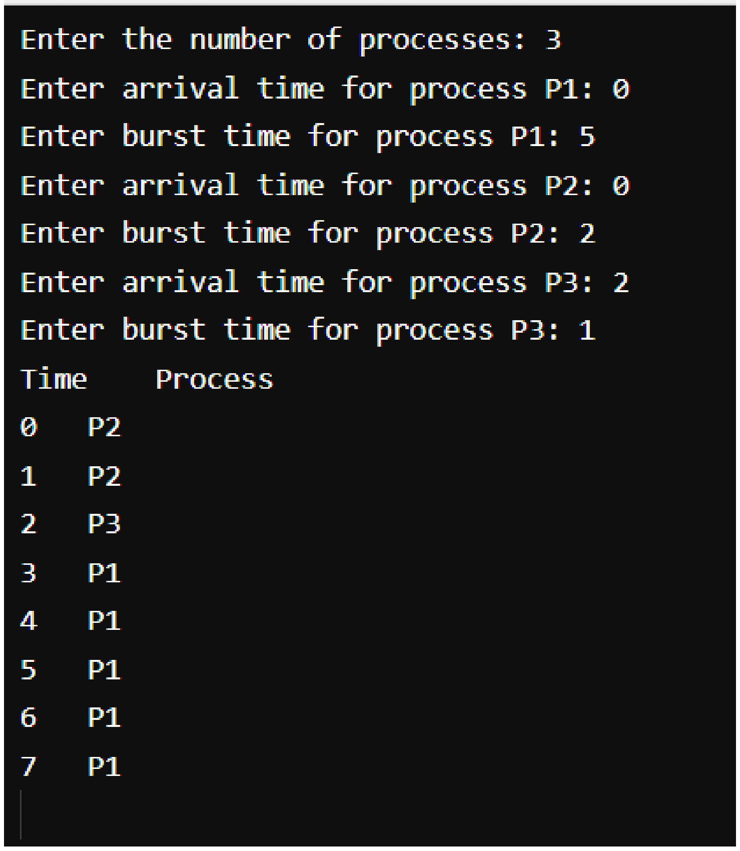
4. After taking input, we call the fcfsScheduling function to perform FCFS scheduling and display the waiting time and turnaround time for each process.

Practical 5

Implementing a preemptive scheduling algorithm requires tracking the state of processes and making decisions to switch between processes based on their priority or remaining execution time. Here's a C program that implements the preemptive SJF scheduling algorithm:

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

// Structure to represent a process
struct Process {
    int pid;         // Process ID
    int arrival;     // Arrival time
    int burst;       // Burst time
    int remaining;   // Remaining burst time
    bool completed;  // Flag to track if the process has completed
};

// Function to perform preemptive SJF scheduling
void preemptiveSJFScheduling(struct Process processes[], int n) {
    int currentTime = 0;
    int completedProcesses = 0;

    printf("Time\tProcess\n");

    while (completedProcesses < n) {
        int shortestRemainingTime = -1;
        int shortestProcessIndex = -1;

        // Find the process with the shortest remaining burst time
        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrival <= currentTime) {
                if (shortestRemainingTime == -1 || processes[i].remaining < shortestRemainingTime) {
                    shortestRemainingTime = processes[i].remaining;
                    shortestProcessIndex = i;
                }
            }
        }

        if (shortestProcessIndex == -1) {
            // No eligible process is available; increment time
            currentTime++;
        } else {
            // Execute the selected process for one time unit
            processes[shortestProcessIndex].remaining--;
            printf("%d\tP%d\n", currentTime, processes[shortestProcessIndex].pid);

            if (processes[shortestProcessIndex].remaining == 0) {
                // Process has completed
                processes[shortestProcessIndex].completed = true;
                completedProcesses++;
            }

            currentTime++;
        }
    }
}

int main() {
    int n;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &processes[i].arrival);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burst);
        processes[i].remaining = processes[i].burst;
        processes[i].completed = false;
    }

    // Perform preemptive SJF scheduling
    preemptiveSJFScheduling(processes, n);

    return 0;
}
```

Output



```
Enter the number of processes: 3
Enter arrival time for process P1: 0
Enter burst time for process P1: 5
Enter arrival time for process P2: 0
Enter burst time for process P2: 2
Enter arrival time for process P3: 2
Enter burst time for process P3: 1
Time    Process
0   P2
1   P2
2   P3
3   P1
4   P1
5   P1
6   P1
7   P1
```

In this program:

1. We define a Process structure to represent each process, including its process ID (pid), arrival time (arrival), burst time (burst), remaining burst time (remaining), and a flag (completed) to track if the process has completed.

2. The preemptiveSJFScheduling function implements the preemptive SJF scheduling algorithm. It tracks the state of processes, continuously selects the process with the shortest remaining burst time among the eligible processes, and executes it for one time unit.

3. In the main function, we take user input for the number of processes and details of each process, including arrival time and burst time.

4. After taking input, we call the preemptiveSJFScheduling function to perform preemptive SJF scheduling and display the timeline of process execution.

5. Compile and run the program, and it will perform preemptive SJF scheduling for the given set of processes.

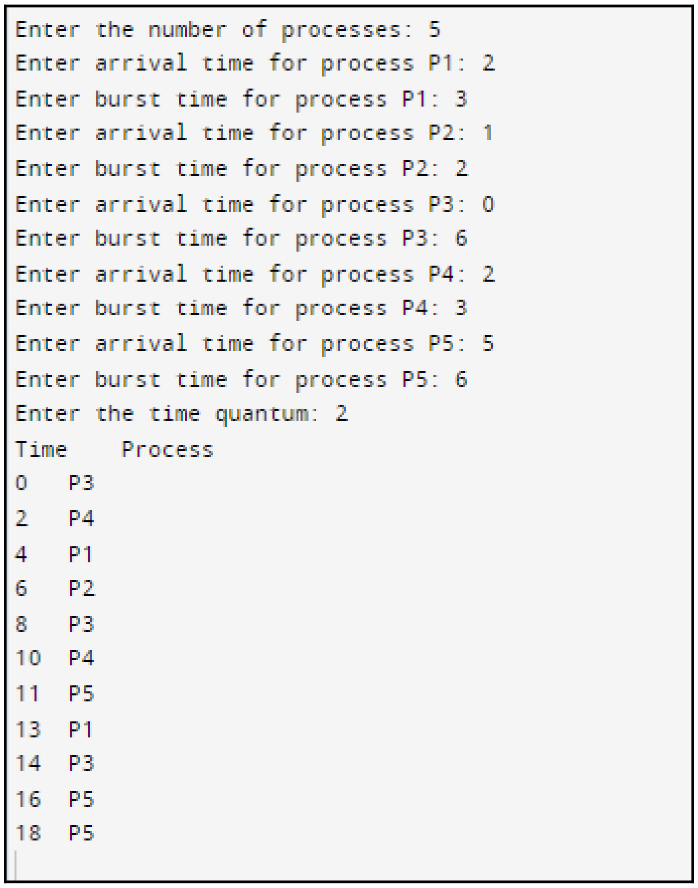
Practical 6

C program that implements the Round-Robin (RR) scheduling algorithm with preemptive behavior. This program takes user input for the number of processes, their arrival times, burst times, and the time quantum.

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

// Structure to represent a process
struct Process {
    int pid;        // Process ID
    int arrival;    // Arrival time
    int burst;      // Burst time
    int remaining;  // Remaining burst time
    bool completed; // Flag to track if the process has completed
};

// Function to perform preemptive Round-Robin scheduling
void preemptiveRoundRobinScheduling(struct Process processes[], int n, int time_quantum) {
    int currentTime = 0;
    int completedProcesses = 0;

    printf("Time\tProcess\n");

    while (completedProcesses < n) {
        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrival <= currentTime) {
                int execution_time = (processes[i].remaining < time_quantum) ?
processes[i].remaining : time_quantum;

                processes[i].remaining -= execution_time;
                printf("%d\tP%d\n", currentTime, processes[i].pid);

                if (processes[i].remaining == 0) {
                    processes[i].completed = true;
                    completedProcesses++;
                }

                currentTime += execution_time;
            }
        }
    }
}
```

```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &processes[i].arrival);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burst);
        processes[i].remaining = processes[i].burst;
        processes[i].completed = false;
    }

    int time_quantum;
    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);

    // Perform preemptive Round-Robin scheduling
    preemptiveRoundRobinScheduling(processes, n, time_quantum);

    return 0;
}
```

Output



```
Enter the number of processes: 5
Enter arrival time for process P1: 2
Enter burst time for process P1: 3
Enter arrival time for process P2: 1
Enter burst time for process P2: 2
Enter arrival time for process P3: 0
Enter burst time for process P3: 6
Enter arrival time for process P4: 2
Enter burst time for process P4: 3
Enter arrival time for process P5: 5
Enter burst time for process P5: 6
Enter the time quantum: 2
Time    Process
0   P3
2   P4
4   P1
6   P2
8   P3
10  P4
11  P5
13  P1
14  P3
16  P5
18  P5
```

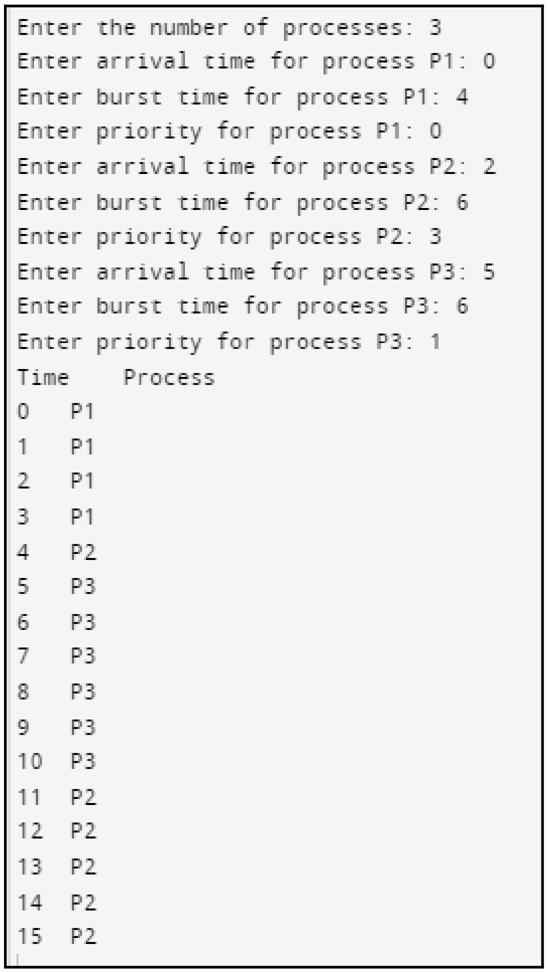
This program is quite similar to the preemptive SJF scheduling program you provided, with adjustments made to implement the Round-Robin scheduling algorithm. It takes user input for process details and the time quantum, then simulates the execution of processes using the RR scheduling algorithm and displays the timeline of process execution.

Practical 7

C program that implements the Priority Scheduling algorithm with a preemptive approach. This program takes user input for the number of processes, their arrival times, burst times, and priorities.

```c
#include <stdio.h>
#include <stdbool.h>

// Structure to represent a process
struct Process {
    int pid;        // Process ID
    int arrival;    // Arrival time
    int burst;      // Burst time
    int priority;   // Priority
    int remaining;  // Remaining burst time
    bool completed; // Flag to track if the process has completed
};

// Function to perform preemptive Priority scheduling
void preemptivePriorityScheduling(struct Process processes[], int n) {
    int currentTime = 0;
    int completedProcesses = 0;

    printf("Time\tProcess\n");

    while (completedProcesses < n) {
        int highestPriority = 1000;  // Highest priority
        int highestPriorityIndex = -1;

        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrival <= currentTime) {
                if (processes[i].priority < highestPriority) {
                    highestPriority = processes[i].priority;
                    highestPriorityIndex = i;
                }
            }
        }

        if (highestPriorityIndex == -1) {
            // No eligible process is available; increment time
            currentTime++;
        } else {
            // Execute the selected process for one time unit
            processes[highestPriorityIndex].remaining--;
            printf("%d\tP%d\n", currentTime, processes[highestPriorityIndex].pid);

            if (processes[highestPriorityIndex].remaining == 0) {
                // Process has completed
                processes[highestPriorityIndex].completed = true;
                completedProcesses++;
            }

            currentTime++;
        }
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &processes[i].arrival);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burst);
        printf("Enter priority for process P%d: ", i + 1);
        scanf("%d", &processes[i].priority);
        processes[i].remaining = processes[i].burst;
        processes[i].completed = false;
    }

    // Perform preemptive Priority scheduling
    preemptivePriorityScheduling(processes, n);

    return 0;
}
```

Output



```
Enter the number of processes: 3
Enter arrival time for process P1: 0
Enter burst time for process P1: 4
Enter priority for process P1: 0
Enter arrival time for process P2: 2
Enter burst time for process P2: 6
Enter priority for process P2: 3
Enter arrival time for process P3: 5
Enter burst time for process P3: 6
Enter priority for process P3: 1
Time    Process
0   P1
1   P1
2   P1
3   P1
4   P2
5   P3
6   P3
7   P3
8   P3
9   P3
10  P3
11  P2
12  P2
13  P2
14  P2
15  P2
```

This program allows you to enter the number of processes, their arrival times, burst times, and priorities. It then simulates the execution of processes based on the preemptive Priority Scheduling algorithm and displays the timeline of process execution.

Explanation of the code:

1. We start by defining a structure struct Process to represent a process. This structure contains fields for the process ID (pid), arrival time (arrival), burst time (burst), priority (priority), remaining burst time (remaining), and a flag to track if the process has completed (completed).

2. The preemptivePriorityScheduling function is used to perform the preemptive Priority scheduling. This function takes an array of processes and the number of processes as its parameters.

3. We initialize currentTime to 0 to keep track of the current time and completedProcesses to 0 to keep track of the number of completed processes.

4. Inside the scheduling loop, we find the process with the highest priority among those that have arrived and are not completed. We also keep track of the index of this process.

5. If there's no eligible process available at the current time, we increment the currentTime. This is because no processes have arrived or all eligible processes have been completed.

6. If we find an eligible process with the highest priority, we execute this process for one time unit by decrementing its remaining burst time. We also print the process's ID and the current time.

7. If the process's remaining burst time becomes 0, it means the process has completed, so we mark it as completed and increment completedProcesses.

8. The loop continues until all processes have completed.

9. In the main function, we take user input for the number of processes, their arrival times, burst times, and priorities.

10. After inputting process details, we call the preemptivePriorityScheduling function to perform the scheduling and display the timeline of process execution.

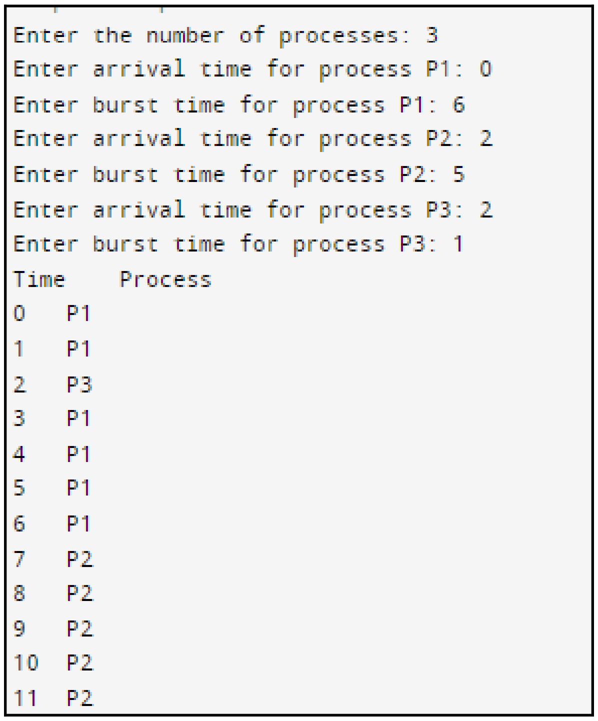
11. This code simulates the Priority Scheduling algorithm with a preemptive approach and allows the user to input the necessary process details.

Practical 8

C program that implements the Shortest Remaining Time First (SRTF) scheduling algorithm, which is also known as Shortest Job First (SJF) with preemption. This program takes user input for the number of processes, their arrival times, burst times, and then simulates the SRTF scheduling algorithm

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

// Structure to represent a process
struct Process {
    int pid;        // Process ID
    int arrival;    // Arrival time
    int burst;      // Burst time
    int remaining;  // Remaining burst time
    bool completed; // Flag to track if the process has completed
};

// Function to perform Shortest Remaining Time First (SRTF) scheduling
void SRTFScheduling(struct Process processes[], int n) {
    int currentTime = 0;
    int completedProcesses = 0;

    printf("Time\tProcess\n");

    while (completedProcesses < n) {
        int shortestRemainingTime = 1000;  // Initialize with a large value
        int shortestProcessIndex = -1;

        for (int i = 0; i < n; i++) {
            if (!processes[i].completed && processes[i].arrival <= currentTime) {
                if (processes[i].remaining < shortestRemainingTime) {
                    shortestRemainingTime = processes[i].remaining;
                    shortestProcessIndex = i;
                }
            }
        }

        if (shortestProcessIndex == -1) {
            // No eligible process is available; increment time
            currentTime++;
        } else {
            // Execute the selected process for one time unit
            processes[shortestProcessIndex].remaining--;
            printf("%d\tP%d\n", currentTime, processes[shortestProcessIndex].pid);

            if (processes[shortestProcessIndex].remaining == 0) {
                // Process has completed
                processes[shortestProcessIndex].completed = true;
                completedProcesses++;
            }

            currentTime++;
        }
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", i + 1);
        scanf("%d", &processes[i].arrival);
        printf("Enter burst time for process P%d: ", i + 1);
        scanf("%d", &processes[i].burst);
        processes[i].remaining = processes[i].burst;
        processes[i].completed = false;
    }

    // Perform SRTF scheduling
    SRTFScheduling(processes, n);

    return 0;
}
```

Output



```
Enter the number of processes: 3
Enter arrival time for process P1: 0
Enter burst time for process P1: 6
Enter arrival time for process P2: 2
Enter burst time for process P2: 5
Enter arrival time for process P3: 2
Enter burst time for process P3: 1
Time    Process
0   P1
1   P1
2   P3
3   P1
4   P1
5   P1
6   P1
7   P2
8   P2
9   P2
10  P2
11  P2
```

Explanation of the code:

1. The code structure is similar to the previous code examples, with the main difference being the SRTF (Shortest Remaining Time First) scheduling logic.

2. We define a structure struct Process to represent a process, including its process ID (pid), arrival time (arrival), burst time (burst), remaining burst time (remaining), and a flag to track if the process has completed (completed).

3. The SRTFScheduling function performs the Shortest Remaining Time First scheduling. This function takes an array of processes and the number of processes as parameters.

4. We initialize currentTime to 0 to track the current time and completedProcesses to 0 to keep track of the number of completed processes.

5. Inside the scheduling loop, we search for the process with the shortest remaining burst time among those that have arrived and are not completed. We also keep track of the index of this process.

6. If there are no eligible processes available at the current time, we increment the currentTime. This is because no processes have arrived or all eligible processes have been completed.

7. If we find an eligible process with the shortest remaining burst time, we execute this process for one time unit by decrementing its remaining burst time. We also print the process's ID and the current time.

8. If the process's remaining burst time becomes 0, it means the process has completed, so we mark it as completed and increment completedProcesses.

9. The loop continues until all processes have completed.

10. In the main function, we take user input for the number of processes, their arrival times, burst times, and execute the SRTF scheduling by calling the SRTFScheduling function.

11. This code simulates the Shortest Remaining Time First (SRTF) scheduling algorithm, where the process with the shortest remaining burst time is selected for execution at each step