PRACTICAL NO. 4

Problem Statement: Write a C program to implement the following scheduling algorithms.

A. First come first serve
B. Round Robin Scheduling
C. Shortest job first
D. Shortest Job remaining first.

Theory: CPU scheduling is the process that allows one process to use the CPU while others are waiting. The objective of scheduling algorithms is to maximize CPU utilization and system efficiency by determining which processes will execute at a particular time. Several scheduling algorithms exist, each suited to different types of system requirements.

1. **First Come First Serve (FCFS):** The simplest scheduling algorithm, FCFS, follows a non-preemptive approach. The process that arrives first is executed first. It works on the principle of a queue, and processes are executed in the order of their arrival. While this is easy to implement, FCFS suffers from the "convoy effect," where short processes are forced to wait for a long process to complete, leading to poor turnaround times.
2. **Round Robin Scheduling (RR):** Round Robin is a preemptive scheduling algorithm, which ensures that each process is assigned a small time slice or quantum in a cyclic order. After the time slice expires, the next process in the queue is scheduled. This ensures fairness as each process gets an equal share of CPU time. It is widely used in time-sharing systems but can lead to high context-switching overheads.
3. **Shortest Job First (SJF):** The SJF algorithm selects the process with the shortest execution time to execute next. This is an optimal algorithm in terms of minimizing average waiting time for a batch of processes. However, it is non-preemptive, meaning once a process starts, it cannot be interrupted. One drawback of SJF is that it requires knowledge of future process durations, which may not always be available, and it suffers from the problem of "starvation" for longer processes.
4. **Shortest Job Remaining First (SRTF):** Also known as Preemptive SJF, SRTF is a preemptive version of SJF. The CPU is allocated to the process that has the smallest amount of time remaining until completion. If a new process arrives with a shorter burst time than the remaining time of the current process, the CPU switches to the new process. This approach minimizes the average waiting time but increases context switching. Like SJF, SRTF also requires knowledge of process execution times and can lead to starvation for longer processes.

Algorithm:

A. First come first serve
    1. Start.
    2. Sort processes in the order of their arrival time.
    3. Initialize waiting_time of the first process to 0.
    4. For each subsequent process:
        - Set waiting_time[i] = waiting_time[i-1] + burst_time[i-1].

5. Calculate turnaround_time[i] for each process as:
    - turnaround_time[i] = waiting_time[i] + burst_time[i].
6. Calculate average waiting time and turnaround time.
7. Display process details and times.
8. End.

B. Round Robin Scheduling
1. Start.
2. Input the time quantum.
3. Initialize remaining_time for each process as its burst_time.
4. Repeat until all processes are completed:
    - For each process in the list:
        - If remaining_time[i] > 0:
            - If remaining_time[i] > quantum:
                - Reduce remaining_time[i] by the quantum.
                - Increase total time.
    - Else:
        - Set process's waiting_time = current_time - burst_time.
        - Add the remaining time to the total time.
        - Set remaining_time[i] = 0.
5. Calculate turnaround times: turnaround_time[i] = waiting_time[i] + burst_time[i].
6. Calculate average waiting and turnaround times.
7. Display process details and times.
8. End.

C. Shortest job first
1. Start.
2. Sort processes based on their burst times (ascending).
3. Initialize waiting_time for the first process to 0.
4. For each subsequent process:
    - Set waiting_time[i] = waiting_time[i-1] + burst_time[i-1].
5. Calculate turnaround_time[i] for each process:
    - turnaround_time[i] = waiting_time[i] + burst_time[i].
6. Calculate average waiting time and turnaround time.
7. Display process details and times.
8. End.

D. Shortest Job remaining first.
1. Start.
2. Input the number of processes and their burst times.
3. Initialize remaining_time as equal to burst_time for each process.
4. Set the current time to 0 and repeat until all processes are finished:
    - Find the process with the shortest remaining time at the current time.

- Execute this process for one unit of time.
- Decrease the remaining_time of this process by 1.
- If remaining_time[i] == 0, record the completion_time.

5. Calculate waiting time for each process:
   - waiting_time[i] = completion_time[i] - burst_time[i] - arrival_time[i].
6. Calculate turnaround time:
   - turnaround_time[i] = waiting_time[i] + burst_time[i].
7. Calculate average waiting and turnaround times.
8. Display process details and times.
9. End.

Source Code:

A. First come first serve

FCFS.c

```c
#include <stdio.h>

struct Process {
    int pid; // Process ID
    int burst_time;
    int waiting_time;
    int turnaround_time;
};

void calculateTimes(struct Process proc[], int n) {
    proc[0].waiting_time = 0; // Waiting time for first process is 0

    for (int i = 1; i < n; i++) {
        proc[i].waiting_time = proc[i - 1].waiting_time + proc[i - 1].burst_time;
    }

    for (int i = 0; i < n; i++) {
        proc[i].turnaround_time = proc[i].waiting_time + proc[i].burst_time;
    }
}

void printTimes(struct Process proc[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time, proc[i].turnaround_time);
    }

    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}

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

    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].burst_time);
    }

    calculateTimes(proc, n);
    printTimes(proc, n);

    return 0;
}

```

B. Round Robin Scheduling

RRS.C

```c
#include <stdio.h>

struct Process {
    int pid;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
};

void roundRobin(struct Process proc[], int n, int quantum) {
    int time = 0;
    int completed = 0;

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0) {
                if (proc[i].remaining_time > quantum) {
                    time += quantum;
                    proc[i].remaining_time -= quantum;
                } else {
                    time += proc[i].remaining_time;
                    proc[i].waiting_time = time - proc[i].burst_time;
                    proc[i].turnaround_time = time;
                    proc[i].remaining_time = 0;
                    completed++;
                }
            }
        }
    }
}
void printTimes(struct Process proc[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time, proc[i].turnaround_time);
    }

    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
int main() {
    int n, quantum;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].burst_time);
        proc[i].remaining_time = proc[i].burst_time;
    }
    printf("Enter time quantum: ");
    scanf("%d", &quantum);
    roundRobin(proc, n, quantum);
    printTimes(proc, n);
    return 0;
}
```

C. Shortest job first

SJF.c

```c
#include <stdio.h>
struct Process {
    int pid;
    int burst_time;
    int waiting_time;
    int turnaround_time;
};
void sortByBurstTime(struct Process proc[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].burst_time > proc[j].burst_time) {
                struct Process temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
}
void calculateTimes(struct Process proc[], int n) {
    proc[0].waiting_time = 0;

    for (int i = 1; i < n; i++) {
        proc[i].waiting_time = proc[i - 1].waiting_time + proc[i - 1].burst_time;
    }

    for (int i = 0; i < n; i++) {
        proc[i].turnaround_time = proc[i].waiting_time + proc[i].burst_time;
    }
}
void printTimes(struct Process proc[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;
    printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].burst_time);
    }

    sortByBurstTime(proc, n);
    calculateTimes(proc, n);
    printTimes(proc, n);

    return 0;
}
```

D. Shortest Job remaining first.

SJRF.c

```c
#include <stdio.h>
struct Process {
    int pid;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int completed;
};
void SJRF(struct Process proc[], int n) {
    int time = 0, completed = 0, min_burst, shortest;
    while (completed < n) {
        min_burst = 9999;
        shortest = -1;
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0 && proc[i].remaining_time < min_burst) {
                min_burst = proc[i].remaining_time;
                shortest = i;
            }
        }
        if (shortest == -1) break;
        proc[shortest].remaining_time--;
        time++;

        if (proc[shortest].remaining_time == 0) {
            proc[shortest].completed = 1;
            completed++;
            proc[shortest].turnaround_time = time;
            proc[shortest].waiting_time = time - proc[shortest].burst_time;
        }
    }
}
void printTimes(struct Process proc[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;
    printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
        printf("%d\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].burst_time);
        proc[i].remaining_time = proc[i].burst_time;
        proc[i].completed = 0;
    }
    SJRF(proc, n);
    printTimes(proc, n);
    return 0;
}
```

Output:

A. First come first serve

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc FCFS.c -o fcfs
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./fcfs
Enter number of processes: 3
Enter burst time for process 1: 2
Enter burst time for process 2: 7
Enter burst time for process 3: 3
PID     Burst Time      Waiting Time    Turnaround Time
1       2               0               2
2       7               2               9
3       3               9               12
Average Waiting Time: 3.67
Average Turnaround Time: 7.67
```

B. Round Robin Scheduling

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc RRS.c -o rrs
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./rrs
Enter number of processes: 3
Enter burst time for process 1: 10
Enter burst time for process 2: 8
Enter burst time for process 3: 13
Enter time quantum: 3
PID     Burst Time      Waiting Time    Turnaround Time
1       10              17              27
2       8               15              23
3       13              18              31
Average Waiting Time: 16.67
Average Turnaround Time: 27.00
```

C. Shortest job first

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc SJF.c -o sjf
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./sjf
Enter number of processes: 3
Enter burst time for process 1: 6
Enter burst time for process 2: 2
Enter burst time for process 3: 8
PID     Burst Time      Waiting Time    Turnaround Time
2       2               0               2
1       6               2               8
3       8               8               16
Average Waiting Time: 3.33
Average Turnaround Time: 8.67
```

D. Shortest Job remaining first.

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc SJRF.c -o sjrf
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./sjrf
Enter number of processes: 3
Enter burst time for process 1: 6
Enter burst time for process 2: 8
Enter burst time for process 3: 3
PID     Burst Time      Waiting Time    Turnaround Time
1       6               3               9
2       8               9               17
3       3               0               3
Average Waiting Time: 4.00
Average Turnaround Time: 9.67
```

Learning Outcomes:

PRACTICAL NO. 5

Problem Statement: Implementation of the following Memory Allocation Methods for fixed partition.

A. First Fit
B. Worst Fit
C. Best Fit

Theory: In operating systems, memory management is crucial for allocating system resources to processes efficiently. When a process is loaded into memory, it requires a continuous block of memory to execute. However, with multiple processes, determining the optimal way to allocate memory to these processes becomes a challenge. This is where fixed partitioning memory allocation techniques such as First Fit, Best Fit, and Worst Fit come into play. These strategies help in assigning available memory blocks (or partitions) to processes based on different criteria.

1. **Fixed Partitioning:** Fixed partitioning divides the available physical memory into a fixed number of blocks or partitions, each of a fixed size. Each partition can accommodate exactly one process, regardless of the size of the process. Processes are assigned to memory partitions using one of the three main strategies: First Fit, Best Fit, and Worst Fit.
2. **First Fit:** In First Fit, memory is allocated to the first available partition that is large enough to accommodate the process. This is a fast and simple algorithm since it does not search for the best possible partition but simply assigns the first one that fits the process. However, it may lead to external fragmentation as smaller holes in memory may remain unutilized.
3. **Best Fit:** The Best Fit method finds the smallest partition that is large enough to hold the process. This strategy attempts to reduce wasted space in memory by finding the partition that leaves the least unused memory. Though Best Fit minimizes leftover memory in each allocation, it often results in small, unusable gaps between allocated processes, leading to fragmentation over time.
4. **Worst Fit:** Worst Fit allocates the largest available partition to the process. The idea behind this strategy is to leave the largest remaining block of memory available for future allocations. The expectation is that by giving the largest block to the process, fewer but larger holes will remain in memory, potentially preventing fragmentation. However, Worst Fit can lead to inefficient use of memory as large partitions get consumed rapidly, leaving smaller partitions unusable for larger processes.

These memory allocation strategies aim to efficiently manage the memory space in fixed-partition systems but each has its strengths and weaknesses. First Fit is faster but may lead to fragmentation, Best Fit reduces wasted memory at each allocation but may lead to higher fragmentation in the long term, and Worst Fit helps keep large blocks available but may cause inefficient space utilization. Thus, choosing the right allocation strategy depends on the specific requirements of the system and the workload it handles.

Source Code:

A. First Fit

first_fit.c

```c
#include <stdio.h>

void firstFit(int partitions[], int partitionCount, int processes[], int processCount) {
    int allocation[processCount];
    for (int i = 0; i < processCount; i++)
        allocation[i] = -1;

    for (int i = 0; i < processCount; i++) {
        for (int j = 0; j < partitionCount; j++) {
            if (partitions[j] >= processes[i]) {
                allocation[i] = j;
                partitions[j] -= processes[i];
                break;
            }
        }
    }

    printf("\nFirst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tPartition No.\n");
    for (int i = 0; i < processCount; i++) {
        printf("%d\t\t%d\t\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int partitionCount, processCount;
    printf("Enter the number of memory partitions: ");
    scanf("%d", &partitionCount);
    int partitions[partitionCount];
    for (int i = 0; i < partitionCount; i++) {
        printf("Partition %d: ", i + 1);
        scanf("%d", &partitions[i]);
    }

    printf("Enter the number of processes: ");
    scanf("%d", &processCount);
    int processes[processCount];
    for (int i = 0; i < processCount; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processes[i]);
    }

    firstFit(partitions, partitionCount, processes, processCount);
    return 0;
}

```

B. Worst Fit

```c
#include <stdio.h>
#define MAX 10
void worstFit(int partitions[], int m, int processes[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        int worstIndex = -1;
        for (int j = 0; j < m; j++) {
            if (partitions[j] >= processes[i]) {
                if (worstIndex == -1 || partitions[j] > partitions[worstInde
x]) {
                    worstIndex = j;
                }
            }
        }

        if (worstIndex != -1) {
            allocation[i] = worstIndex;
            partitions[worstIndex] -= processes[i];
        }
    }
    printf("\nWorst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tPartition No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
int main() {
    int partitions[MAX], processes[MAX];
    int m, n;
    printf("Enter the number of memory partitions: ");
    scanf("%d", &m);
    printf("Enter the size of each partition:\n");
    for (int i = 0; i < m; i++) {
        printf("Partition %d: ", i + 1);
        scanf("%d", &partitions[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processes[i]);
    }
    worstFit(partitions, m, processes, n);
    return 0;
}
```

C. Best Fit

```c
#include <stdio.h>
#define MAX 10
void bestFit(int partitions[], int m, int processes[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        int bestIndex = -1;
        for (int j = 0; j < m; j++) {
            if (partitions[j] >= processes[i]) {
                if (bestIndex == -1 || partitions[j] < partitions[bestIndex]
) {
                    bestIndex = j;
                }
            }
        }
        if (bestIndex != -1) {
            allocation[i] = bestIndex;
            partitions[bestIndex] -= processes[i];
        }
    }
    printf("\nBest Fit Allocation:\n");
    printf("Process No.\tProcess Size\tPartition No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
int main() {
    int partitions[MAX], processes[MAX];
    int m, n;
    printf("Enter the number of memory partitions: ");
    scanf("%d", &m);
    printf("Enter the size of each partition:\n");
    for (int i = 0; i < m; i++) {
        printf("Partition %d: ", i + 1);
        scanf("%d", &partitions[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processes[i]);
    }
    bestFit(partitions, m, processes, n);
    return 0;
}
```

Output:

A. First Fit

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc first_fit.c -o first-fit
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./first-fit
Enter the number of memory partitions: 5
Partition 1: 100
Partition 2: 500
Partition 3: 200
Partition 4: 300
Partition 5: 600
Enter the number of processes: 4
Process 1: 212
Process 2: 412
Process 3: 112
Process 4: 422

First Fit Allocation:
Process No.     Process Size    Partition No.
1               212             2
2               412             5
3               112             2
4               422             Not Allocated
```

B. Worst Fit

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc worst_fit.c -o worst
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./worst
Enter the number of memory partitions: 3
Partition 1:    100
Partition 2: 200
Partition 3: 300
Enter the number of processes: 4
Process 1: 50
Process 2: 150
Process 3: 250
Process 4: 350

Worst Fit Allocation:
Process No.     Process Size    Partition No.
1               50              3
2               150             3
3               250             Not Allocated
4               350             Not Allocated
```

C. Best Fit

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc best_fit.c -o best
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./best
Enter the number of memory partitions: 3
Partition 1: 100
Partition 2: 200
Partition 3: 300
Enter the number of processes: 4
Process 1: 50
Process 2: 150
Process 3: 250
Process 4: 350

Best Fit Allocation:
Process No.     Process Size    Partition No.
1               50              1
2               150             2
3               250             3
4               350             Not Allocated
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

Learning Outcomes:

PRACTICAL NO. 6

Problem Statement: Write a program to implement reader/writer problems using semaphore.

Theory: The **Reader/Writer problem** is a classic synchronization issue in concurrent computing, where multiple processes either read from or write to a shared resource, such as a file or database. The challenge arises from the need to balance access to the resource between multiple readers and writers without causing conflicts or data corruption. Readers only require access to read the data, and multiple readers can safely read simultaneously without affecting each other or the integrity of the data. Writers, on the other hand, need exclusive access to modify the data. If a writer modifies the data while a reader or another writer is accessing it, inconsistencies can occur, leading to potential data corruption or unpredictable behaviour.

This problem emphasizes the need for controlled access, ensuring that while a writer is modifying the resource, no other writer or reader can interfere with it. Conversely, multiple readers can read the resource concurrently if no writer is present. Therefore, an optimal solution aims to maximize concurrency for readers while maintaining data integrity by restricting writer access. Two primary solutions exist for this problem:

1. **Reader-priority solution:** Gives preference to readers, allowing them to access the shared resource as long as there are no writers waiting. This can cause starvation for writers, as a continuous stream of readers could prevent a writer from gaining access.
2. **Writer-priority solution:** Prioritizes writers, ensuring that as soon as a writer requests access, it is granted once the current readers finish, potentially leading to reader starvation in cases where writers frequently access the resource.

Semaphore-Based Solution

The most common approach to solving the Reader/Writer problem involves using semaphores, a synchronization primitive that helps control access to shared resources. Semaphores can be used to coordinate between readers and writers and ensure that only one writer accesses the shared resource at a time, while multiple readers can access it concurrently when no writer is present. Two semaphores are typically used:

1. **mutex:** Ensures mutual exclusion when modifying the shared reader_count, which tracks how many readers are accessing the resource.
2. **write_lock:** Ensures that writers have exclusive access to the shared resource, blocking both readers and other writers while a write operation is in progress.

In the **reader-priority solution**, the mutex semaphore is used to control access to the reader_count. When the first reader begins reading, it locks the write_lock, preventing any writers from accessing the resource. As long as there are readers, no writers can gain access. Once the last reader finishes, it unlocks the write_lock, allowing waiting writers to proceed.

Writers, in turn, must wait until there are no readers or other writers accessing the resource before they can proceed. The design of the Reader/Writer problem highlights the importance of managing concurrency and access control in multi-threaded systems, where the correct balance between resource utilization and fairness is crucial for preventing deadlock or starvation while ensuring data integrity

Source Code:

reader_prog.c

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

sem_t mutex, writeSemaphore;
int readCount = 0;

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

    // Reader wants to enter
    sem_wait(&mutex);
    readCount++;
    if (readCount == 1) {
        // If this is the first reader, lock the writeSemaphore
        sem_wait(&writeSemaphore);
    }
    sem_post(&mutex);

    // Reading section
    printf("Reader %d is reading\n", readerNum);
    sleep(1); // Simulate reading time

    // Reader wants to leave
    sem_wait(&mutex);
    readCount--;
    if (readCount == 0) {
        // If this is the last reader, release the writeSemaphore
        sem_post(&writeSemaphore);
    }
    sem_post(&mutex);

    printf("Reader %d has finished reading\n", readerNum);
    return NULL;
}

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

    // Writer wants to enter
    sem_wait(&writeSemaphore);

    // Writing section
    printf("Writer %d is writing\n", writerNum);
    sleep(1); // Simulate writing time

    // Writer leaves
    sem_post(&writeSemaphore);
    printf("Writer %d has finished writing\n", writerNum);
    return NULL;
}

int main() {
    int i;
    pthread_t readers[5], writers[3];
    int readerIds[5] = {1, 2, 3, 4, 5};
    int writerIds[3] = {1, 2, 3};

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&writeSemaphore, 0, 1);

    // Create reader threads
    for (i = 0; i < 5; i++) {
        pthread_create(&readers[i], NULL, reader, &readerIds[i]);
    }
```

```
68
69      // Create writer threads
70      for (i = 0; i < 3; i++) {
71          pthread_create(&writers[i], NULL, writer, &writerIds[i]);
72      }
73
74      // Join threads
75      for (i = 0; i < 5; i++) {
76          pthread_join(readers[i], NULL);
77      }
78      for (i = 0; i < 3; i++) {
79          pthread_join(writers[i], NULL);
80      }
81
82      // Destroy semaphores
83      sem_destroy(&mutex);
84      sem_destroy(&writeSemaphore);
85
86      return 0;
87  }
88
```

Output:

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc reader_prog.c -o read
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./read
Reader 1 is reading
Reader 3 is reading
Reader 4 is reading
Reader 5 is reading
Reader 2 is reading
Reader 1 has finished reading
Reader 4 has finished reading
Reader 3 has finished reading
Reader 5 has finished reading
Reader 2 has finished reading
Writer 1 is writing
Writer 1 has finished writing
Writer 2 is writing
Writer 2 has finished writing
Writer 3 is writing
Writer 3 has finished writing
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

Learning Outcomes:

PRACTICAL NO. 7

Problem Statement: Write a program to implement Banker's algorithm for deadlock avoidance.

Theory: The Banker's Algorithm is a deadlock avoidance technique used in operating systems to manage resource allocation among multiple processes. It ensures that the system always remains in a safe state by simulating resource allocation requests and checking whether fulfilling these requests would lead to deadlock. The algorithm is named after the analogy of a bank lending money to customers while ensuring that it can still meet the needs of all customers without running out of funds. Similarly, the operating system acts as a banker, allocating system resources (e.g., memory, CPU time, I/O devices) to processes while preventing deadlock.

In a system, processes request and release resources dynamically, and the goal of the Banker's Algorithm is to avoid unsafe resource allocation that could potentially lead to deadlock. A system is in a safe state if there exists at least one sequence of processes such that each process can complete its execution without any deadlock, even if all processes request the maximum number of resources they need. To achieve this, the Banker's Algorithm uses information about each process's maximum resource demand, the resources already allocated, and the system's currently available resources. Whenever a new resource request is made, the algorithm checks if granting the request would keep the system in a safe state by simulating the allocation and verifying if all processes can still finish.

The algorithm operates in two main steps:

1. **Resource Request Handling:** When a process makes a resource request, the system checks if the request is within the maximum claim of the process and if there are enough available resources to fulfill the request. If the request can be fulfilled, the system proceeds to the next step.
2. **Safety Check:** After temporarily granting the resources to the requesting process, the system checks if this allocation leaves the system in a safe state by calculating a possible safe sequence of process execution. A safe sequence is one where each process can obtain the necessary resources, complete its execution, and release its allocated resources, enabling the next process in the sequence to do the same. If the system remains in a safe state, the request is granted permanently; otherwise, the request is denied to prevent deadlock.

While the Banker's Algorithm is effective in preventing deadlock, it has some limitations. It requires that the system know in advance the maximum resources each process might request, which may not always be possible in real-world scenarios where resource needs can vary dynamically. Moreover, the algorithm introduces some computational overhead, as the safety check has to be performed each time a new resource request is made. Despite these limitations, the Banker's Algorithm provides a theoretical foundation for deadlock avoidance in systems with limited, predictable resource allocation patterns, such as embedded systems or batch processing environments.

Source Code:

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int allocation[10];
    int max[10];
    int need[10];
} Process;
int isSafe(Process processes[], int numProcesses, int numResources, int avai
lable[]) {
    int work[numResources];
    for (int i = 0; i < numResources; i++) {
        work[i] = available[i];
    }
    int finish[numProcesses];
    for (int i = 0; i < numProcesses; i++) {
        finish[i] = 0;
    }
    int safeSequence[numProcesses];
    int sequenceIndex = 0;
    while (sequenceIndex < numProcesses) {
        int found = 0;
        for (int i = 0; i < numProcesses; i++) {
            if (finish[i] == 0) {
                int canAllocate = 1;
                for (int j = 0; j < numResources; j++) {
                    if (processes[i].need[j] > work[j]) {
                        canAllocate = 0;
                        break;
                    }
                }
                if (canAllocate) {
                    for (int j = 0; j < numResources; j++) {
                        work[j] += processes[i].allocation[j];
                    }
                    finish[i] = 1;
                    safeSequence[sequenceIndex++] = i;
                    found = 1;
                }
            }
        }
```

1,1 Top

```c
        if (!found) {
            return 0; // Not safe
        }
    }
    printf("Safe sequence: ");
    for (int i = 0; i < numProcesses; i++) {
        printf("P%d ", safeSequence[i]);
    }
    printf("\n");
    return 1; // Safe
}
int main() {
    int numProcesses, numResources;
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    printf("Enter the number of resources: ");
    scanf("%d", &numResources);
    Process processes[numProcesses];
    int available[numResources];
    printf("Enter the available resources: ");
    for (int i = 0; i < numResources; i++) {
        scanf("%d", &available[i]);
    }
    printf("Enter the maximum resources for each process:\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process P%d:\n", i);
        for (int j = 0; j < numResources; j++) {
            scanf("%d", &processes[i].max[j]);
        }
    }
    printf("Enter the allocated resources for each process:\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process P%d:\n", i);
        for (int j = 0; j < numResources; j++) {
            scanf("%d", &processes[i].allocation[j]);
            processes[i].need[j] = processes[i].max[j] - processes[i].allocation[j];
        }
    }
    if (isSafe(processes, numProcesses, numResources, available)) {
```

```
        printf("System is safe.\n");
    } else {
        printf("System is not safe.\n");
    }
    return 0;
}
```

Output:

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc bankers_algo.c -o bank
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./bank
Enter the number of processes: 3
Enter the number of resources: 3
Enter the available resources: 3 3 2
Enter the maximum resources for each process:
Process 1: 7 5 3
Process 2: 3 2 2
Process 3: 4 3 3
Enter the allocated resources for each process:
Process 1: 0 1 0
Process 2: 2 0 0
Process 3: 0 0 2
System is not in a safe state. Deadlock may occur.
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

Learning Outcomes:

PRACTICAL NO. 8

Problem Statement: Process Management a) fork() b) execv() c) execlp() d) wait() and e) sleep()

A. Program to implement the fork function using C.
B. Program to implement execv function using C.
C. Program to implement execlp function.
D. Program to implement wait function using C.
E. Program to implement sleep function using C.

Theory: Process management is a crucial part of an operating system, where processes are created, executed, and terminated. In C programming, various system calls and functions provide the capability to manage processes efficiently. Some of the key functions involved in process management include fork(), execv(), execlp(), wait(), and sleep(). These functions allow you to create new processes, replace the current process image with a new one, synchronize processes, and pause execution, respectively.

1. **fork()** is one of the most fundamental system calls in Unix-like operating systems. It creates a new child process by duplicating the parent process. The child process inherits a copy of the parent's memory and variables but has its own unique process ID (PID). The return value of fork() allows distinguishing between the parent and child processes: it returns 0 to the child and the child's PID to the parent. This allows for parallel execution of parent and child processes, enabling multitasking and parallel processing. However, without proper synchronization, issues like race conditions may arise when both processes try to access shared resources.
2. **execv() and execlp()** are part of the exec family of functions, which replace the current process image with a new program. The execv() function takes a file path and an array of arguments, while execlp() accepts the program name and its arguments as a variable argument list and searches for the executable in the system's PATH. These functions are often used after fork() to make the child process execute a different program. For example, in a typical shell, fork() creates a child process, and execv() or execlp() is used to run a command within the child process. If successful, these functions do not return because the current process is completely replaced by the new program.
3. **wait()** is used by a parent process to wait for its child process to finish executing. Without wait(), the parent and child processes would run independently, potentially causing the parent process to finish execution before its child. By using wait(), the parent ensures that it waits until all child processes are done before resuming. This synchronization is critical in process management, as it prevents issues like the parent process terminating and leaving orphaned child processes. Additionally, wait() returns the exit status of the child, allowing the parent to know whether the child completed successfully or encountered an error.
4. **sleep()** allows a process to pause execution for a specific number of seconds. This is useful when you want to delay a process for a certain period or control the timing between operations. The function suspends the calling process and makes it inactive during the

specified sleep duration. sleep() is often used for simulating long-running tasks, testing, or delaying processes in a controlled manner.

Together, these functions form the backbone of process management in C, enabling the creation of concurrent processes, execution of new programs, process synchronization, and controlled delays. Understanding and effectively utilizing these functions allows developers to manage system resources efficiently, avoid deadlocks, and ensure proper execution of processes.

Source Code:

(a)

fork.c

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed!\n");
    } else if (pid == 0) {
        printf("Child process: PID = %d\n", getpid());
    } else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
    }

    return 0;
}

```

(b)

execv.c

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

int main() {
    char *args[] = {"/bin/ls", "-l", NULL}; // Executing "ls -l" command

    printf("Before execv call\n");
    execv("/bin/ls", args);
    printf("This line will not be executed if execv is successful\n");

    return 0;
}

```

(c)

execlp.c

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

int main() {
    printf("Before execlp call\n");
    execlp("ls", "ls", "-l", NULL); // Executing "ls -l" command
    printf("This line will not be executed if execlp is successful\n");

    return 0;
}

```

(d)

wait.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed!\n");
    } else if (pid == 0) {
        printf("Child process: PID = %d\n", getpid());
        sleep(2); // Simulate some work in child process
        printf("Child process is done.\n");
    } else {
        printf("Parent waiting for child to complete...\n");
        wait(NULL); // Wait for child to finish
        printf("Child completed. Parent resuming.\n");
    }

    return 0;
}

```

(e)

sleep.c

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

int main() {
    printf("Sleeping for 5 seconds...\n");
    sleep(5); // Sleep for 5 seconds
    printf("Woke up after 5 seconds!\n");

    return 0;
}

```

Output:

(a)

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc fork.c
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc fork.c -o fork
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./fork
Parent process: PID = 22022, Child PID = 22023
Child process: PID = 22023
```

(b)

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc execv.c -o execv
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./execv
Before execv call
total 212
-rwxrwxrwx 1 anshul anshul 16080 Oct 29 18:48 a.out
-rwxrwxrwx 1 anshul anshul 16504 Oct 29 18:42 bank
-rwxrwxrwx 1 anshul anshul  3061 Oct 29 18:40 bankers_algo.c
-rwxrwxrwx 1 anshul anshul 16128 Oct 29 18:25 best
-rwxrwxrwx 1 anshul anshul  1615 Oct 29 18:23 best_fit.c
-rwxrwxrwx 1 anshul anshul   243 Oct 29 18:47 execlp.c
-rwxrwxrwx 1 anshul anshul 16048 Oct 29 18:52 execv
-rwxrwxrwx 1 anshul anshul   278 Oct 29 18:47 execv.c
-rwxrwxrwx 1 anshul anshul 16136 Oct 29 18:18 first-fit
-rwxrwxrwx 1 anshul anshul   100 Oct 24 16:52 first.c
-rwxrwxrwx 1 anshul anshul 40766 Oct 24 16:32 first.exe
-rwxrwxrwx 1 anshul anshul  1464 Oct 29 18:18 first_fit.c
-rwxrwxrwx 1 anshul anshul 16080 Oct 29 18:48 fork
-rwxrwxrwx 1 anshul anshul   326 Oct 29 18:46 fork.c
-rwxrwxrwx 1 anshul anshul 16496 Oct 29 18:29 read
-rwxrwxrwx 1 anshul anshul  1967 Oct 29 18:28 reader_prog.c
-rwxrwxrwx 1 anshul anshul   192 Oct 29 18:51 sleep.c
-rwxrwxrwx 1 anshul anshul 15952 Oct 24 16:53 test
-rwxrwxrwx 1 anshul anshul   536 Oct 29 18:47 wait.c
-rwxrwxrwx 1 anshul anshul 16136 Oct 29 18:24 worst
-rwxrwxrwx 1 anshul anshul  1625 Oct 29 18:23 worst_fit.c
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

(c)

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc execlp.c -o execlp
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./execlp
Before execlp call
total 228
-rwxrwxrwx 1 anshul anshul 16080 Oct 29 18:48 a.out
-rwxrwxrwx 1 anshul anshul 16504 Oct 29 18:42 bank
-rwxrwxrwx 1 anshul anshul  3061 Oct 29 18:40 bankers_algo.c
-rwxrwxrwx 1 anshul anshul 16128 Oct 29 18:25 best
-rwxrwxrwx 1 anshul anshul  1615 Oct 29 18:23 best_fit.c
-rwxrwxrwx 1 anshul anshul 16000 Oct 29 18:53 execlp
-rwxrwxrwx 1 anshul anshul   243 Oct 29 18:47 execlp.c
-rwxrwxrwx 1 anshul anshul 16048 Oct 29 18:52 execv
-rwxrwxrwx 1 anshul anshul   278 Oct 29 18:47 execv.c
-rwxrwxrwx 1 anshul anshul 16136 Oct 29 18:18 first-fit
-rwxrwxrwx 1 anshul anshul   100 Oct 24 16:52 first.c
-rwxrwxrwx 1 anshul anshul 40766 Oct 24 16:32 first.exe
-rwxrwxrwx 1 anshul anshul  1464 Oct 29 18:18 first_fit.c
-rwxrwxrwx 1 anshul anshul 16080 Oct 29 18:48 fork
-rwxrwxrwx 1 anshul anshul   326 Oct 29 18:46 fork.c
-rwxrwxrwx 1 anshul anshul 16496 Oct 29 18:29 read
-rwxrwxrwx 1 anshul anshul  1967 Oct 29 18:28 reader_prog.c
-rwxrwxrwx 1 anshul anshul   192 Oct 29 18:51 sleep.c
-rwxrwxrwx 1 anshul anshul 15952 Oct 24 16:53 test
-rwxrwxrwx 1 anshul anshul   536 Oct 29 18:47 wait.c
-rwxrwxrwx 1 anshul anshul 16136 Oct 29 18:24 worst
-rwxrwxrwx 1 anshul anshul  1625 Oct 29 18:23 worst_fit.c
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

(d)

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc wait.c -o wait
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./wait
Parent waiting for child to complete...
Child process: PID = 23996
Child process is done.
Child completed. Parent resuming.
```

(e)

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc sleep.c -o sleep
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./sleep
Sleeping for 5 seconds...
Woke up after 5 seconds!
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

Learning Outcomes:

PRACTICAL NO. 9

Problem Statement: Write a program to implement Inter Process Communication (IPC) using Message Queues.

Theory: Inter-Process Communication (IPC) is a critical mechanism used in operating systems to allow processes to communicate and synchronize their activities. One of the most common IPC methods is Message Queues, which provide a way for processes to exchange data in the form of messages. Unlike other IPC methods such as shared memory or pipes, message queues allow asynchronous communication, meaning that processes do not have to be directly connected or active at the same time for the communication to occur. Message queues enable processes to send and receive messages even when they are not synchronized, ensuring that communication can happen in a flexible and decoupled manner.

A message queue is essentially a linked list of messages stored within the kernel. Each message has two components: a message type, which is typically a long integer, and the message body, which is a string or an array of bytes. Processes can send messages to the queue, and other processes can retrieve them. The operating system ensures that the messages are stored and handled safely, preventing any data corruption or loss in case multiple processes access the queue simultaneously. The key advantage of message queues is their ability to store messages independently of the processes that use them. When a process sends a message to a queue, the message is stored in the queue even if the receiving process is not ready to read it immediately.

Components and Workflow

1. **Key Generation:** To use a message queue, the sending and receiving processes need to share a unique identifier or key. This is typically generated using the ftok() function in Linux, which converts a filename and an integer to a unique key. The key is used to identify the queue in the kernel.
2. **Queue Creation:** A process can create or connect to an existing message queue using the msgget() system call. If the message queue does not already exist, it will be created with specified permissions. Otherwise, the process can simply attach to the existing queue.
3. **Sending Messages:** To send a message, the msgsnd() system call is used. The message has a defined structure that includes a message type and the actual message content. The message is placed in the queue, where it remains until another process retrieves it.
4. **Receiving Messages:** A process can receive messages from the queue using the msgrcv() system call. It specifies the message type it wants to receive, allowing it to filter specific messages from the queue. The messages are removed from the queue once they are received, although in some implementations, they may remain for multiple reads depending on the flags set.
5. **Queue Removal:** When a message queue is no longer needed, it should be removed to free up system resources. This is done using the msgctl() system call. While removing the queue, the kernel ensures that any pending messages are flushed and the queue is cleaned up properly.

Source Code:

write.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

#define MAX 100

struct mesg_buffer {
    long mesg_type;
    char mesg_text[MAX];
} message;

int main() {
    key_t key;
    int msgid;

    // Generate unique key
    key = ftok("progfile", 65);

    // Create message queue and return identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data: ");
    fgets(message.mesg_text, MAX, stdin);

    // Send message
    msgsnd(msgid, &message, sizeof(message), 0);

    printf("Data sent is: %s\n", message.mesg_text);
    return 0;
}
```

read.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 100
struct mesg_buffer {
    long mesg_type;
    char mesg_text[MAX];
} message;

int main() {
    key_t key;
    int msgid;

    // Generate unique key
    key = ftok("progfile", 65);

    // Get the message queue identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // Receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // Display the message
    printf("Data received is: %s\n", message.mesg_text);

    // Destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Output:

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc write.c -o write
gcc read.c -o read
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./write
Write Data: MY NAME IS ANSHUL
Data sent is: MY NAME IS ANSHUL

anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./read
Data received is: MY NAME IS ANSHUL

anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

Learning Outcomes:

PRACTICAL NO. 10

Problem Statement: Write a program to implement IPC using pipes.

Theory: Inter-Process Communication (IPC) using pipes is one of the oldest and simplest methods for enabling communication between processes in Unix and Unix-like operating systems. Pipes provide a unidirectional communication channel between processes, allowing one process to send data to another in a producer-consumer fashion. This method is especially useful in scenarios where processes need to share information or synchronize their operations.

Basics of Pipes

A pipe is essentially a conduit for data flow between two endpoints: a read end and a write end. When a pipe is created, the operating system provides two file descriptors, one for reading and one for writing. Data written to the pipe by one process can be read by another, allowing for a smooth exchange of information between processes. Pipes are typically used in a parent-child process relationship where a parent process creates a child process, and both processes can communicate using the pipe. In the simplest form, pipes allow for unidirectional communication, meaning that data flows in only one direction—from the writer to the reader. If bidirectional communication is required, two pipes are generally created: one for sending data from the parent to the child and the other for sending data from the child to the parent.

Creating and Using Pipes

In Unix systems, pipes can be created using the pipe() system call. This system call takes an array of two integers as an argument, where pipefd[0] represents the file descriptor for the read end, and pipefd[1] represents the file descriptor for the write end. Once the pipe is established, the processes involved in communication can use standard I/O functions like write() and read() to exchange data. For example, a parent process can send data to a child process by writing to the write end of the pipe, while the child process can read this data from the read end. When the data transfer is complete, both processes must close the pipe to release system resources.

Process Synchronization and Communication

One common usage of pipes is in the fork() system call, which creates a new child process. After a fork, both the parent and child processes can share the same pipe. The parent typically writes data into the pipe, while the child reads from it. This ensures that the child receives specific information from the parent, such as instructions, configuration data, or results from computations. Pipes provide a level of synchronization between processes. For example, if the reading process attempts to read data from an empty pipe, it will block (wait) until data is available. Similarly, the writing process may block if the pipe's buffer is full, waiting for the reading process to consume some of the data. This behavior ensures that data is transmitted reliably without loss, although it can introduce delays if one process is significantly slower than the other.

Source Code:

ipc_pipe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main() {
    int pipefd[2];
    pid_t pid;
    char write_msg[] = "Hello, this is a message from the parent process!";
    char read_msg[100];
    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("Pipe failed");
        return 1;}
    // Create a child process
    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        return 1;}
    if (pid > 0) { // Parent process
        close(pipefd[0]); // Close the read end of the pipe

        // Write the message to the pipe
        write(pipefd[1], write_msg, strlen(write_msg) + 1);
        printf("Parent sent message: %s\n", write_msg);
        close(pipefd[1]); // Close the write end of the pipe
    } else { // Child process
        close(pipefd[1]); // Close the write end of the pipe
        // Read the message from the pipe
        read(pipefd[0], read_msg, sizeof(read_msg));
        printf("Child received message: %s\n", read_msg);
        close(pipefd[0]); // Close the read end of the pipe
    }
    return 0;
}
```

Output:

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc ipc_pipe.c -o ipc
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./ipc
Parent sent message: Hello, this is a message from the parent process!
Child received message: Hello, this is a message from the parent process!
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$
```

Learning Outcomes:

PRACTICAL NO. 11

Problem Statement: Write a program using Pthread, where main thread calculates number of lines in a file and child calculates number of words.

Theory: Multithreading is a popular technique used in modern computing for improving the performance and efficiency of programs by executing multiple tasks simultaneously. The pthread library in C provides the means to create and manage threads in UNIX-based operating systems. Threads are lightweight processes that share the same memory space, allowing parallel execution of tasks. This makes it possible to divide the work between multiple threads, improving execution time, especially for tasks that can be executed concurrently. In the context of file processing, such as counting the number of lines and words in a file, multithreading can be an effective way to distribute these two tasks to different threads.

In C, the pthread library provides an API for creating and managing threads. It allows developers to spawn new threads from the main thread, execute tasks concurrently, and then synchronize them to ensure proper execution flow. In this context, the problem of counting the number of lines and words in a file can be divided into two tasks, where one thread counts the lines and the other counts the words. The pthread_create function is used to create a new thread, and the pthread_join function ensures synchronization by making the main thread wait for the child thread to complete its task.

In the case of counting lines and words in a file, the main thread can be tasked with counting the lines while a child thread concurrently counts the words. When we count lines in a file, we simply look for newline characters ('\n'), as each newline represents the end of one line. On the other hand, counting words requires scanning through the text for transitions between whitespace characters (spaces, newlines, or tabs) and non-whitespace characters, where each such transition signals the beginning of a new word.

The pthread_create function spawns a new thread and assigns it a specific task, defined in a function that the thread executes. In our case, the function to count the number of words in the file is executed by the child thread. The parent thread (the main thread) remains responsible for counting the lines. The pthread_join function is crucial here for ensuring that the main thread waits for the child thread to finish its execution. This avoids the scenario where the program terminates before both tasks are completed.

Thread Synchronization and Communication:

One of the primary challenges in multithreading is synchronization, ensuring that all threads execute correctly without race conditions or memory access issues. In our program, synchronization is relatively simple, as both the main thread and the child thread operate on the same file but perform different tasks independently. However, the pthread_join function is necessary to ensure that the main thread waits for the child thread to finish its work before printing the results and terminating the program. Although our simple example does not involve shared memory or complex synchronization, threads in larger programs often need to coordinate access to shared resources. This is commonly done using mutexes (mutual exclusion locks) or condition variables to ensure that threads do not interfere with each other's operations on shared data.

Source Code:

pthread.c

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

typedef struct {
    char *filename;
    int line_count;
    int word_count;
} FileData;

// Function for child thread to count words
void *count_words(void *arg) {
    FileData *file_data = (FileData *)arg;
    FILE *file = fopen(file_data->filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        pthread_exit(NULL);
    }

    int words = 0;
    char ch, prev = ' ';
    while ((ch = fgetc(file)) != EOF) {
        if (isspace(ch) && !isspace(prev)) {
            words++;
        }
        prev = ch;
    }
    if (!isspace(prev)) words++;  // count the last word if it doesn't end with space

    file_data->word_count = words;
    fclose(file);
    pthread_exit(NULL);
}

// Function for main thread to count lines
void count_lines(FileData *file_data) {
    FILE *file = fopen(file_data->filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(1);
    }

    int lines = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        if (ch == '\n') {
            lines++;
        }
    }
    file_data->line_count = lines;
    fclose(file);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    FileData file_data;
    file_data.filename = argv[1];
    file_data.line_count = 0;
    file_data.word_count = 0;

    // Create a child thread for word counting
    pthread_t thread;
    if (pthread_create(&thread, NULL, count_words, &file_data) != 0) {
        perror("Error creating thread");
        return 1;
    }

    // Count lines in the main thread
    count_lines(&file_data);

    // Wait for the child thread to finish
    pthread_join(thread, NULL);

    // Output results
    printf("Child thread: Number of words = %d\n", file_data.word_count);
    printf("Main thread: Number of lines = %d\n", file_data.line_count);

    return 0;
}
```

Output:

```
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ gcc pthread.c -o pthread
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ ./pthread os.txt
Child thread: Number of words = 108
Main thread: Number of lines = 2
anshul@LAPTOP-0GQIM1QM:/mnt/d/anshuuu$ cat os.txt
Multithreading is a popular technique used in modern computing for improving the performance and efficiency of
programs by executing multiple tasks simultaneously. The pthread library in C provides the means to create and
manage threads in UNIX-based operating systems. Threads are lightweight processes that share the same memory sp
ace, allowing parallel execution of tasks. This makes it possible to divide the work between multiple threads,
improving execution time, especially for tasks that can be executed concurrently. In the context of file proces
sing, such as counting the number of lines and words in a file, multithreading can be an effective way to distr
ibute these two tasks to different threads.
```

Learning Outcomes: