OPERATING SYSTEM

Student LAB MANUAL

CYCLE – I:

1. **Write a program to simulate the following CPU Scheduling algorithms.**
   a) FCFS

   AIM: To write a program in C to simulate the First-Come, First-Served (FCFS) CPU scheduling algorithm.

   Algorithm:

   1.Start the process.

   2.Input the number of processes from the user.

   3. For each process, input the burst time (the time each process will take) and arrival time (the time when the process arrives).

   4. Sort the processes based on their arrival times. This ensures that we follow the FCFS order.

   5.For the first process, set its waiting time to 0 because it starts immediately.

   6. For each subsequent process, calculate the waiting time by subtracting the arrival time from the service time (time at which the CPU starts executing the process).

   If the result is negative, set the waiting time to 0 (as a process cannot wait a negative amount of time).

   7.For each process, calculate the turn-around time by adding the burst time to the waiting time.

   8.Calculate the average waiting time by dividing the total waiting time by the number of processes.

   9.Calculate the average turn-around time by dividing the total turn-around time by the number of processes.

   10.Display the processes, their burst time, waiting time, and turn-around time.

   11. End the process.

Procedure:

1. Declare variables for processes, burst time, waiting time, turn-around time, and total time.

2. Prompt the user to enter the number of processes.

3. Use a loop to input the burst time and arrival time for each process.

4. Sort the processes based on their arrival times to follow the FCFS order.

5. Initialize the waiting time for the first process to 0.

6. Calculate the waiting time and turn-around time for each process using the formulas provided.

7. Calculate the average waiting time and turn-around time.

8. Display the information for each process, including their burst time, waiting time, and turn-around time.

Expected input:

Enter the number of processes: 3

Enter burst time for process 1: 24

Enter arrival time for process 1: 0

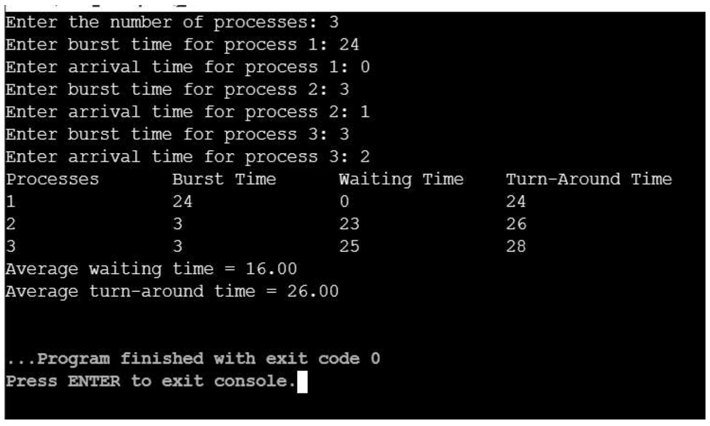
Enter burst time for process 2: 3

Enter arrival time for process 2: 1

Enter burst time for process 3: 3

Enter arrival time for process 3: 2

Output:



```
Enter the number of processes: 3
Enter burst time for process 1: 24
Enter arrival time for process 1: 0
Enter burst time for process 2: 3
Enter arrival time for process 2: 1
Enter burst time for process 3: 3
Enter arrival time for process 3: 2
Processes       Burst Time      Waiting Time    Turn-Around Time
1               24              0               24
2               3               23              26
3               3               25              28
Average waiting time = 16.00
Average turn-around time = 26.00


...Program finished with exit code 0
Press ENTER to exit console.
```

1. b) SJF

Aim: To write a program in C to simulate the Shortest Job First (SJF) CPU scheduling algorithm.

Algorithm:

1.Start the process.

2.Input the number of processes from the user.

3.For each process, input the burst time (the time each process will take) and arrival time (the time when the process arrives).

4.Sort the processes based on their burst times (Shortest Job First). If two processes have the same burst time, sort by arrival time.

5. For the first process, set its waiting time to 0 because it starts immediately.

6. For each subsequent process, calculate the waiting time by subtracting the arrival time from the total time taken by previous processes.

If the result is negative, set the waiting time to 0 (as a process cannot wait a negative amount of time).

7. For each process, calculate the turn-around time by adding the burst time to the waiting time.

8. Calculate the average waiting time by dividing the total waiting time by the number of processes.

9. Calculate the average turn-around time by dividing the total turn-around time by the number of processes.

10. Display the processes, their burst time, waiting time, and turn-around time.

11. End the process.

Procedure:

1. Declare variables for processes, burst time, waiting time, turn-around time, and total time.

2. Prompt the user to enter the number of processes.

3. Use a loop to input the burst time and arrival time for each process.

4. Sort the processes based on their burst times. If two processes have the same burst time, sort by arrival time.

5. Initialize the waiting time for the first process to 0.

6. Calculate the waiting time and turn-around time for each process using the formulas provided.

7. Calculate the average waiting time and turn-around time.

8. Display the information for each process, including their burst time, waiting time, and turn-around time.

Expected input:

Enter the number of processes: 3

Enter burst time for process 1: 6

Enter arrival time for process 1: 1

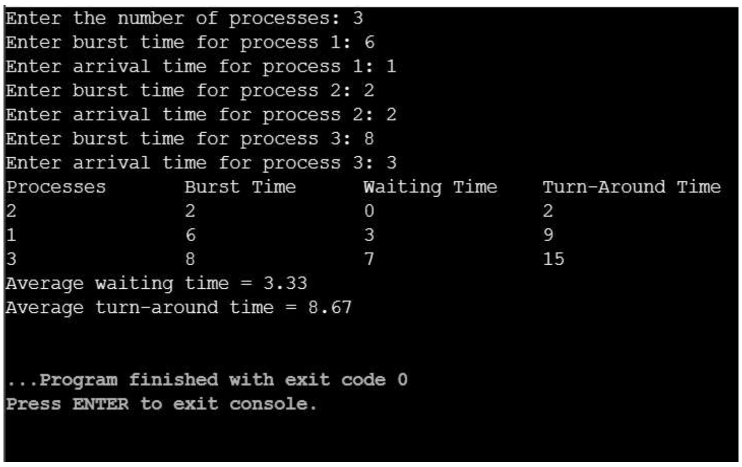
Enter burst time for process 2: 2

Enter arrival time for process 2: 2

Enter burst time for process 3: 8

Enter arrival time for process 3: 3

Output:



```
Enter the number of processes: 3
Enter burst time for process 1: 6
Enter arrival time for process 1: 1
Enter burst time for process 2: 2
Enter arrival time for process 2: 2
Enter burst time for process 3: 8
Enter arrival time for process 3: 3
Processes       Burst Time      Waiting Time    Turn-Around Time
2               2               0               2
1               6               3               9
3               8               7               15
Average waiting time = 3.33
Average turn-around time = 8.67


...Program finished with exit code 0
Press ENTER to exit console.
```

1 . c) Round Robin

Aim: To write a program in C to simulate the Round Robin (RR) CPU scheduling algorithm.

Algorithm:

1.Start the process.

2.Input the number of processes and the time quantum from the user.

3.For each process, input the burst time (the time each process will take) and arrival time (the time when the process arrives).

4.Initialize the waiting time for each process to 0.

5.Initialize the remaining burst time for each process to its burst time.

6.Initialize the current time to 0.

7.While there are unfinished processes, repeat:

For each process, if it has arrived and has remaining burst time:

- If the remaining burst time is greater than the time quantum, subtract the time quantum from the remaining burst time and add the time quantum to the current time.
- Else, add the remaining burst time to the current time, set the remaining burst time to 0, and calculate the waiting time and turn-around time for the process.

8.Calculate the average waiting time by dividing the total waiting time by the number of processes.

9.Calculate the average turn-around time by dividing the total turn-around time by the number of processes.

10.Display the processes, their burst time, waiting time, and turn-around time.

11.End the process.

Procedure:

1.Declare variables for processes, burst time, waiting time, turn-around time, remaining burst time, and total time.

2.Prompt the user to enter the number of processes and the time quantum.

3.Use a loop to input the burst time and arrival time for each process.

4.Initialize the waiting time and remaining burst time for each process.

5.Initialize the current time to 0.

6.While there are unfinished processes, repeat:

- For each process, if it has arrived and has remaining burst time:
    - If the remaining burst time is greater than the time quantum, subtract the time quantum from the remaining burst time and add the time quantum to the current time.
    - Else, add the remaining burst time to the current time, set the remaining burst time to 0, and calculate the waiting time and turn-around time for the process.

7. Calculate the average waiting time and turn-around time.

8.Display the information for each process, including their burst time, waiting time, and turn-around time.

Expected input:

Enter the number of processes: 3

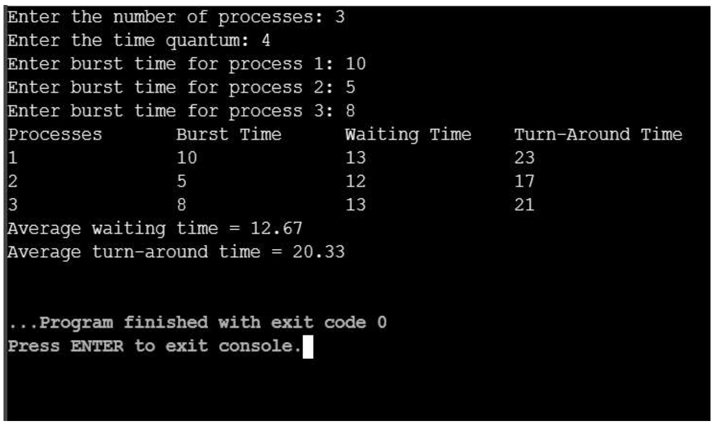
Enter the time quantum: 4

Enter burst time for process 1: 10

Enter burst time for process 2: 5

Enter burst time for process 3: 8

Output:



```
Enter the number of processes: 3
Enter the time quantum: 4
Enter burst time for process 1: 10
Enter burst time for process 2: 5
Enter burst time for process 3: 8
Processes       Burst Time      Waiting Time    Turn-Around Time
1               10              13              23
2               5               12              17
3               8               13              21
Average waiting time = 12.67
Average turn-around time = 20.33


...Program finished with exit code 0
Press ENTER to exit console.
```

1 . d) Priority

Aim: To write a program in C to simulate the Priority CPU scheduling algorithm.

Algorithm:

1. Start the process.

2. Input the number of processes from the user.

3. For each process, input the burst time (the time each process will take), arrival time (the time when the process arrives), and priority (lower number indicates higher priority).

4. Sort the processes based on their priorities. If two processes have the same priority, sort by arrival time.

5. For the first process, set its waiting time to 0 because it starts immediately.

6. For each subsequent process, calculate the waiting time by subtracting the arrival time from the total time taken by previous processes.

If the result is negative, set the waiting time to 0 (as a process cannot wait a negative amount of time).

7. For each process, calculate the turn-around time by adding the burst time to the waiting time.

8. Calculate the average waiting time by dividing the total waiting time by the number of processes.

9. Calculate the average turn-around time by dividing the total turn-around time by the number of processes.

10. Display the processes, their burst time, waiting time, and turn-around time.

11. End the process.

Procedure:

1. Declare variables for processes, burst time, waiting time, turn-around time, priorities, and total time.
2. Prompt the user to enter the number of processes.
3. Use a loop to input the burst time, arrival time, and priority for each process.

4. Sort the processes based on their priorities. If two processes have the same priority, sort by arrival time.
5. Initialize the waiting time for the first process to 0.
6. Calculate the waiting time and turn-around time for each process using the formulas provided.
7. Calculate the average waiting time and turn-around time.
8. Display the information for each process, including their burst time, waiting time, and turn-around time.

Expected input:

Enter the number of processes: 3

Enter burst time for process 1: 10

Enter arrival time for process 1: 0

Enter priority for process 1: 2

Enter burst time for process 2: 5

Enter arrival time for process 2: 1

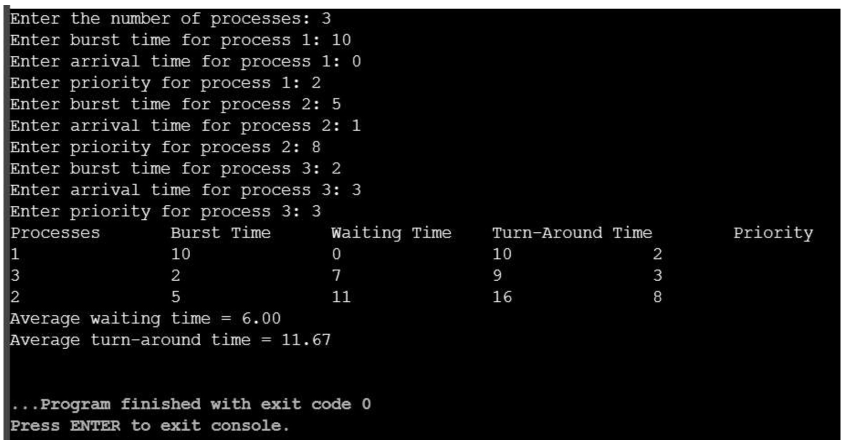
Enter priority for process 2: 1

Enter burst time for process 3: 8

Enter arrival time for process 3: 2

Enter priority for process 3: 3

Output:



```
Enter the number of processes: 3
Enter burst time for process 1: 10
Enter arrival time for process 1: 0
Enter priority for process 1: 2
Enter burst time for process 2: 5
Enter arrival time for process 2: 1
Enter priority for process 2: 8
Enter burst time for process 3: 2
Enter arrival time for process 3: 3
Enter priority for process 3: 3
Processes       Burst Time      Waiting Time    Turn-Around Time        Priority
1               10              0               10              2
3               2               7               9               3
2               5               11              16              8
Average waiting time = 6.00
Average turn-around time = 11.67


...Program finished with exit code 0
Press ENTER to exit console.
```

2 . a) Process Management

Aim: To implement process management system calls: fork(), exit(), wait(), waitpid(), and exec() in C.

Algorithm:

1. Start: Initialize the program.
2. Fork: Create a child process using fork().
3. Exec: In the child process, use execvp() to execute a new program.
4. Wait: The parent process calls wait() or waitpid() to wait for the child.
5. Exit: The child process terminates using exit().
6. Waitpid: The parent uses waitpid() to retrieve the child's exit status.
7. Display: Show process IDs and exit status.
8. End: Terminate both processes.

Procedure:

Declare variables (pid, status).

Call fork() to create a child process.

Child Process:

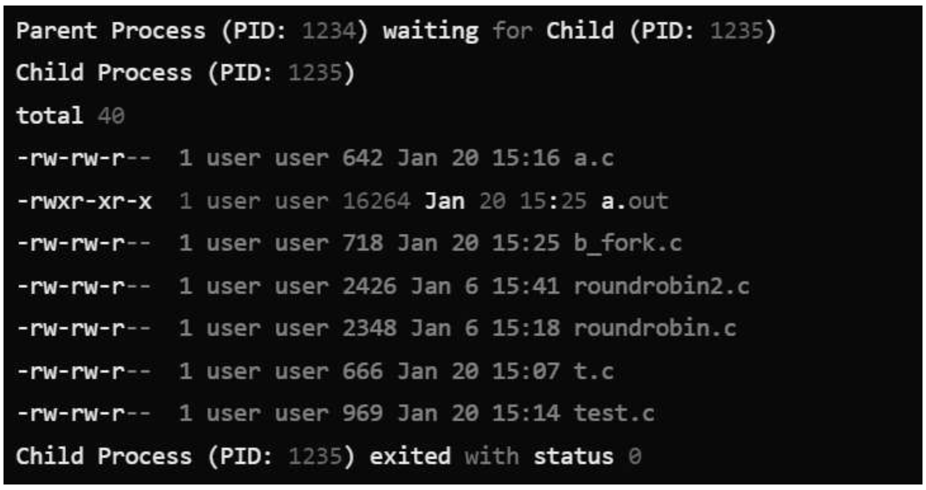
- Display child's PID using getpid().
- Call execvp() to execute a new program (e.g., ls -l).
- If execvp() fails, print an error and call exit().

Parent Process:

- Display parent's PID using getpid().
- Use waitpid() to wait for the child and get the exit status.
- Print the child's exit status using WEXITSTATUS().

Exit: Both processes terminate.

Output:



```
Parent Process (PID: 1234) waiting for Child (PID: 1235)
Child Process (PID: 1235)
total 40
-rw-rw-r--  1 user user 642 Jan 20 15:16 a.c
-rwxr-xr-x  1 user user 16264 Jan 20 15:25 a.out
-rw-rw-r--  1 user user 718 Jan 20 15:25 b_fork.c
-rw-rw-r--  1 user user 2426 Jan 6 15:41 roundrobin2.c
-rw-rw-r--  1 user user 2348 Jan 6 15:18 roundrobin.c
-rw-rw-r--  1 user user 666 Jan 20 15:07 t.c
-rw-rw-r--  1 user user 969 Jan 20 15:14 test.c
Child Process (PID: 1235) exited with status 0
```

2 . b) File and Directory Operations

Aim: To implement file and directory operations using system calls (open, read, write, close, lseek, stat, opendir, readdir, closedir).

Algorithm:

File Operations

1. Open/Create a file using open().
2. Write data using write().
3. Move file pointer using lseek().
4. Read data using read().
5. Close file using close().

File Status Retrieval

6. Use stat() to get file metadata.
7. Print file size, permissions, and last modified time.

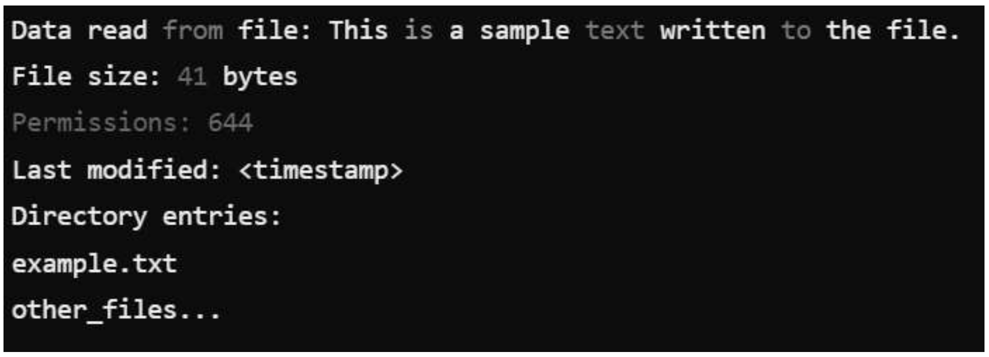
Directory Operations

8. Open the current directory using opendir().
9. Read and print directory contents using readdir().
10. Close the directory using closedir().

Procedure:

1. Open/Create a file with read/write permissions (0644).
2. Write a sample text to the file.
3. Move the file pointer to the beginning.
4. Read and print the data from the file.
5. Close the file.
6. Retrieve and print file metadata using stat().
7. Open the current directory, read entries, and print filenames.
8. Close the directory.
9. Handle errors using perror() at each step.

Output:



```
Data read from file: This is a sample text written to the file.
File size: 41 bytes
Permissions: 644
Last modified: <timestamp>
Directory entries:
example.txt
other_files...
```

3. Banker's Algorithm for Deadlock Avoidance

Aim: Write a program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention

Algorithm:

1. Input the number of processes and resources.
2. Input Max Need, Allocated Resources, and Available Resources.
3. Compute Need matrix:
    a. Need[i][j] = Max[i][j] - Allocation[i][j]
4. Apply Safety Algorithm:
    a. Find a process where Need ≤ Available.
    b. If found, allocate resources temporarily, update Available, and mark it as finished.
    c. Repeat until all processes are finished or no further progress is possible.
5. If all processes complete, print Safe Sequence. Else, print Deadlock.
6. Allow user to request additional resources:
    a. If request ≤ Need and Available, grant the request.
    b. Otherwise, deny the request.

Procedure:

1. Declare variables for processes, resources, maximum need, allocation, available resources, and need matrix.
2. Input the number of processes and resources.
3. Use a loop to input the Maximum, Allocation, and Available resources.
4. Calculate the Need matrix.
5. Use the safety algorithm to check for a safe sequence.
6. Display the result (safe sequence or deadlock message).
7. Allow the user to request resources and decide if they can be granted.

Expected Input:

Enter number of processes: 3

Enter number of resources: 3

Enter Maximum resource matrix:

Process 1: 7 5 3

Process 2: 3 2 2

Process 3: 9 0 2

Enter Allocation matrix:

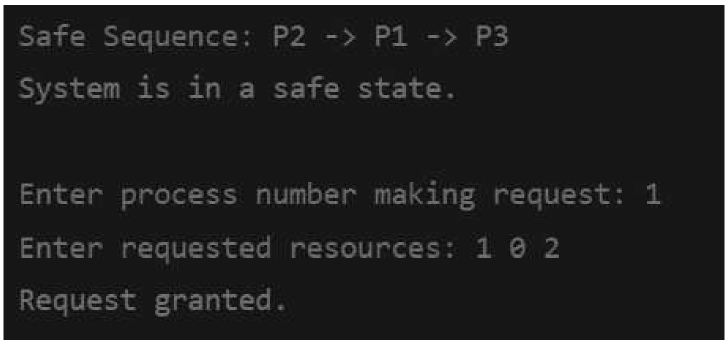
Process 1: 0 1 0

Process 2: 2 0 0

Process 3: 3 0 2

Enter Available resources: 3 3 2

Expected Output:



```
Safe Sequence: P2 -> P1 -> P3
System is in a safe state.

Enter process number making request: 1
Enter requested resources: 1 0 2
Request granted.
```

4. Producer-Consumer Problem using semaphores

Aim: Write a program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls

Algorithm:

1. Start the process.
2. Initialize semaphores:
   a. mutex = 1 (Controls mutual exclusion).
   b. full = 0 (Counts filled slots).
   c. empty = N (Counts empty slots, where N is buffer size).
3. The Producer process:
   a. Wait if the buffer is full.
   b. Acquire mutex (enter critical section).
   c. Produce an item and add it to the buffer.
   d. Release mutex and signal full.
4. The Consumer process:
   a. Wait if the buffer is empty.
   b. Acquire mutex (enter critical section).
   c. Consume an item from the buffer.
   d. Release mutex and signal empty.
5. Repeat steps 3 and 4 indefinitely or for a fixed number of times.
6. End the process.

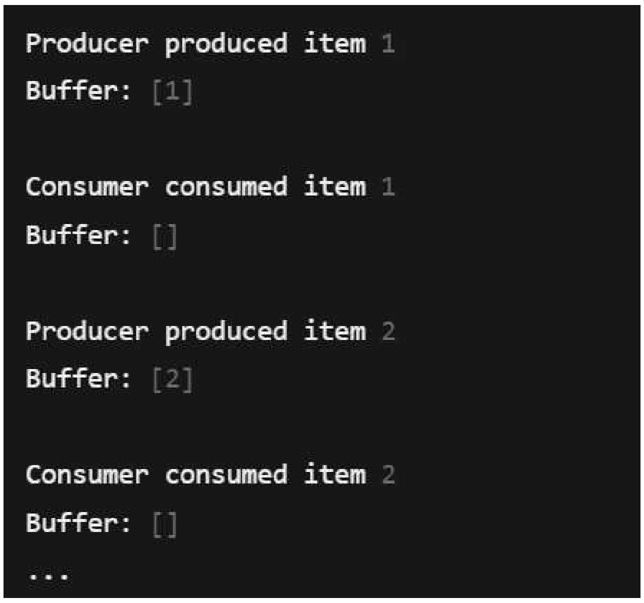
Procedure:

1. Declare buffer, semaphores, and counters for producer and consumer.
2. Initialize semaphores using sem_init().
3. Create producer and consumer threads using pthread_create().
4. Use sem_wait() and sem_post() to handle synchronization.
5. Display buffer status after each operation.
6. Use pthread_join() to wait for threads to finish execution.
7. End the process.

Expected Input:

Enter number of producer-consumer cycles: 5

Expected Output:



```
Producer produced item 1
Buffer: [1]

Consumer consumed item 1
Buffer: []

Producer produced item 2
Buffer: [2]

Consumer consumed item 2
Buffer: []
...
```

5. (a) Inter-Process Communication (IPC) using Pipes

Aim: Write a C program to illustrate IPC using Pipes for communication between a parent and child process.

Algorithm:

1. Start the process.
2. Create a pipe using the pipe() system call.
3. Fork a child process using fork().
4. Parent Process:
   a. Writes data into the pipe using write().
5. Child Process:
   a. Reads data from the pipe using read() and displays it.
6. Close the pipe and end the process.

Procedure:

1. Declare an array for the pipe (pipe_fd[2]).
2. Create the pipe using pipe(pipe_fd).
3. Use fork() to create a child process.
4. Inside the parent:
   a. Close the read end of the pipe.
   b. Write data into the pipe using write().
5. Inside the child:
   a. Close the write end of the pipe.
   b. Read data using read() and print it.

Expected Input:

```
Enter a message: Hello IPC
```

Expected Output:

```
Child received message: Hello IPC
```

5. (b) Inter-Process Communication (IPC) using FIFOs

Aim: Write a C program to illustrate IPC using FIFOs (Named Pipes) for communication between two processes.

Algorithm:

1. Start the process.
2. Create a FIFO (named pipe) using mkfifo().
3. Process 1 (Writer):
    a. Open the FIFO for writing using open().
    b. Write data into the FIFO using write().
4. Process 2 (Reader):
    a. Open the FIFO for reading using open().
    b. Read data from the FIFO using read() and display it.
5. Close the FIFO and remove it using unlink().
6. End the process.

Procedure:

1. Create a FIFO using mkfifo("fifo_file", 0666).
2. In the writer process:
    a. Open the FIFO in write mode and send data.
3. In the reader process:
    a. Open the FIFO in read mode and receive data.
4. Display the received message.
5. Close and remove the FIFO after communication.

Expected Input:

Enter a message: IPC with FIFO

Expected Output:

Received message: IPC with FIFO

5(c) Inter-Process Communication (IPC) using Message Queues

Aim: Write a C program to illustrate IPC using Message Queues for communication between two processes.

Algorithm:

1. Start the process.
2. Create a message queue using msgget().
3. Sender Process:
   a. Send a message to the queue using msgsnd().
4. Receiver Process:
   a. Receive the message using msgrcv() and display it.
5. Delete the message queue using msgctl().
6. End the process.

Procedure:

1. Create a message queue using msgget().
2. In the sender process:
   a. Fill the message structure and send it using msgsnd().
3. In the receiver process:
   a. Receive the message using msgrcv() and print it.
4. Remove the message queue after use.

Expected Input:

Enter a message: Hello Message Queue

Expected Output:

Received message: Hello Message Queue

5(d) Inter-Process Communication (IPC) using Shared Memory

Aim: Write a C program to illustrate IPC using Shared Memory for communication between two processes.

Algorithm:

1. Start the process.
2. Create a shared memory segment using shmget().
3. Writer Process:
    a. Attach the shared memory using shmat().
    b. Write data into the shared memory.
4. Reader Process:
    a. Attach the shared memory using shmat().
    b. Read the data and display it.
5. Detach and remove the shared memory segment using shmdt() and shmctl().
6. End the process.

Procedure:

1. Create shared memory using shmget().
2. Attach the memory in the writer process and write data.
3. Attach the memory in the reader process and read data.
4. Display the received message.
5. Remove the shared memory after communication.

Expected Input:

Enter a message: Shared Memory IPC

Expected Output:

Received message: Shared Memory IPC