VISHNU
UNIVERSAL LEARNING

Estd. 2012

OS Lab Manual
Department of Computer Science

BVRIT Hyderabad

Instructor: Dr. Venkatesh B

Date: April 6, 2025

INDEX

8 Page Replacement Algorithms

- First-In-First-Out (FIFO)
- Least Recently Used (LRU)
- Optimal Page Replacement

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

1 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.

Code

```c
#include <stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int wt[], int at[]) {
int service_time[n];
service_time[0] = at[0];
wt[0] = 0;
for (int i = 1; i < n; i++) {
service_time[i] = service_time[i-1] + bt[i-1];
wt[i] = service_time[i] - at[i];

if (wt[i] < 0) {
wt[i] = 0;
}
}
}
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
for (int i = 0; i < n; i++) {
tat[i] = bt[i] + wt[i];
}
}
void findAvgTime(int processes[], int n, int bt[], int at[]) {
int wt[n], tat[n], total_wt = 0, total_tat = 0;
findWaitingTime(processes, n, bt, wt, at);
findTurnAroundTime(processes, n, bt, wt, tat);
printf("Processes\tBurst Time\tWaiting Time\tTurn-Around Time\n");
for (int i = 0; i < n; i++) {
total_wt += wt[i];
total_tat += tat[i];
printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
}
printf("Average waiting time = %.2f\n", (float)total_wt / (float)n);
printf("Average turn-around time = %.2f\n", (float)total_tat / (float)n);
}
void fcfsScheduling(int processes[], int n, int bt[], int at[]) {
int temp;
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (at[i] > at[j]) {
temp = at[i];
```

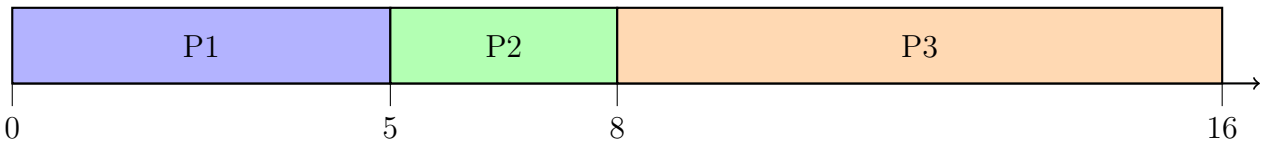
```
at[i] = at[j];
at[j] = temp;
temp = bt[i];
bt[i] = bt[j];
bt[j] = temp;
temp = processes[i];
processes[i] = processes[j];
processes[j] = temp;
}
}
}
findAvgTime(processes, n, bt, at);
}
int main() {
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
int processes[n], burst_time[n], arrival_time[n];
for (int i = 0; i < n; i++) {
processes[i] = i+1;
printf("Enter burst time for process %d: ", i + 1);
scanf("%d", &burst_time[i]);
printf("Enter arrival time for process %d: ", i + 1);
scanf("%d", &arrival_time[i]);
}
fcfsScheduling(processes, n, burst_time, arrival_time);
return 0;
}
```

Output

```
Enter number of processes: 3
Enter Burst Time for P1: 5
Enter Arrival Time for P1: 0
Enter Burst Time for P2: 3
Enter Arrival Time for P2: 1
Enter Burst Time for P3: 8
Enter Arrival Time for P3: 2
```

Process	Arrival	Burst	Waiting	Turnaround	Completion
P1	0	5	0	5	5
P2	1	3	4	7	8
P3	2	8	6	14	16

Gantt Chart



Explanation

FCFS (First-Come, First-Served) is a non-preemptive scheduling algorithm where processes are executed in the order they arrive in the ready queue. It operates like a queue (FIFO), where the first process to arrive gets executed first, regardless of burst time. While simple to implement, it can lead to the convoy effect, where shorter processes get stuck waiting behind longer ones, increasing average waiting time.

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.
3. For each process, input the burst time and arrival time.
4. Sort the processes based on burst time for non-preemptive SJF.
5. For preemptive SJF (SRTF), dynamically choose the process with the shortest remaining burst time.
6. Compute waiting time and turn-around time for each process.
7. Calculate the average waiting and turn-around time.
8. Display the results.
9. End the process.

Procedure:

1. Declare arrays for processes, burst time, waiting time, and turn-around time.
2. Input process details.
3. Sort the processes based on burst time.
4. Compute the waiting time and turn-around time for each process.
5. Calculate and display the average waiting and turn-around time.

Code

```c
#include <stdio.h>
void findWaitingTime(int n, int bt[], int at[], int wt[]) {
    int completed = 0, time = 0, min_bt, shortest;
    int remaining_time[n];
    for (int i = 0; i < n; i++) remaining_time[i] = bt[i];
    while (completed != n) {
        shortest = -1;
        min_bt = 1e9;
        for (int i = 0; i < n; i++) {
            if (at[i] <= time && remaining_time[i] > 0 &&
               remaining_time[i] < min_bt) {
                min_bt = remaining_time[i];
                shortest = i;
            }
        }
        if (shortest == -1) {
```

```
            time++;
            continue;
        }
        remaining_time[shortest]--;
        if (remaining_time[shortest] == 0) {
            completed++;
            wt[shortest] = time + 1 - at[shortest] - bt[shortest
               ];
            if (wt[shortest] < 0) wt[shortest] = 0;
        }
        time++;
    }
}
void findTurnAroundTime(int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) tat[i] = bt[i] + wt[i];
}
void findAvgTime(int n, int bt[], int at[]) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(n, bt, at, wt);
    findTurnAroundTime(n, bt, wt, tat);
    printf("Processes\tBurst Time\tWaiting Time\tTurn-Around Time
       \n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t\t%d\t\t%d\t\t%d\n", i+1, bt[i], wt[i], tat[i
           ]);
    }
    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turn-around time = %.2f\n", (float)total_tat
       / n);
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burst_time[n], arrival_time[n];
    for (int i = 0; i < n; i++) {
        printf("Enter burst time for process %d: ", i+1);
        scanf("%d", &burst_time[i]);
        printf("Enter arrival time for process %d: ", i+1);
        scanf("%d", &arrival_time[i]);
    }
    findAvgTime(n, burst_time, arrival_time);
    return 0;
}
```

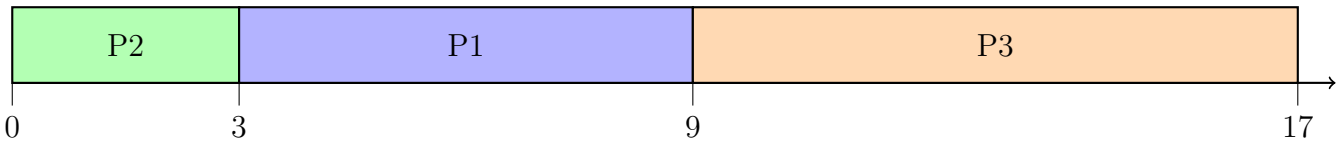
Output

```
Enter number of processes: 3
Enter Burst Time for P1: 6
```

```
Enter Arrival Time for P1: 0
Enter Burst Time for P2: 2
Enter Arrival Time for P2: 1
Enter Burst Time for P3: 8
Enter Arrival Time for P3: 2
```

Process	Arrival	Burst	Waiting	Turnaround	Completion
P1	0	6	3	9	9
P2	1	2	0	2	3
P3	2	8	7	15	17

Gantt Chart




```
Average Waiting Time: 3.33
Average Turnaround Time: 8.67
```

Processes	Burst Time	Waiting Time	Turn-Around Time
2	1	0	1
1	4	1	5
3	6	5	11

Explanation

SJF selects the process with the shortest burst time first. The preemptive version (SRTF) checks for a shorter burst time dynamically. While efficient, it may cause starvation for longer processes.

1 c. Round Robin

AIM: To write a program in C to simulate the Round Robin 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 (time each process will take) and the arrival time (time when the process arrives).
4. Set the time quantum (the maximum time a process can run before being preempted).
5. Maintain a ready queue that processes will enter and exit from.
6. Process the first process in the queue until its burst time has been completed or the time quantum is exhausted.
7. If the process isn't finished, re-queue it to the ready queue with the remaining burst time.
8. If a process finishes execution, calculate the waiting time and turnaround time for each process.
9. Calculate the average waiting time by dividing the total waiting time by the number of processes.
10. Calculate the average turnaround time by dividing the total turnaround time by the number of processes.
11. Display the processes, their burst time, waiting time, and turnaround time.
12. End the process.

Procedure:

1. Declare variables for processes, burst time, waiting time, turn-around time, total time, and time quantum.
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. Implement a Round Robin scheduling algorithm with a ready queue and time quantum.
5. Calculate the waiting time and turnaround time for each process.
6. Display the information for each process, including burst time, waiting time, and turnaround time.

Code

```c
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int tq) {
    int remaining_bt[n], time = 0;
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }

    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0) {
                done = 0;
                if (remaining_bt[i] > tq) {
                    time += tq;
                    remaining_bt[i] -= tq;
                } else {
                    time += remaining_bt[i];
                    wt[i] = time - bt[i];
                    remaining_bt[i] = 0;
                }
            }
        }
        if (done) break;
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findAvgTime(int processes[], int n, int bt[], int tq) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, wt, tq);
    findTurnAroundTime(processes, n, bt, wt, tat);
    printf("Processes\tBurst Time\tWaiting Time\tTurn-Around Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
    }
    printf("Average waiting time = %.2f\n", (float)total_wt / (float)n);
```

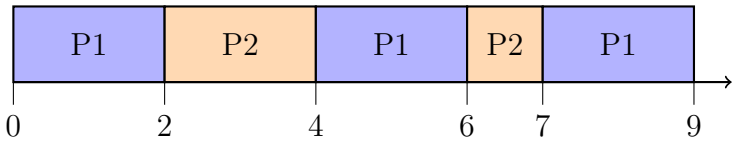
```
    printf("Average turn-around time = %.2f\n", (float)total_tat
       / (float)n);
}

void roundRobinScheduling(int processes[], int n, int bt[], int
   tq) {
    findAvgTime(processes, n, bt, tq);
}

int main() {
    int n, tq;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processes[n], burst_time[n];
    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
    }
    printf("Enter time quantum: ");
    scanf("%d", &tq);

    roundRobinScheduling(processes, n, burst_time, tq);

    return 0;
}
```

Output

```
Enter number of processes: 2
Enter Time Quantum: 2
Enter Burst Time for P1: 5
Enter Arrival Time for P1: 0
Enter Burst Time for P2: 3
Enter Arrival Time for P2: 1
```

Process	Arrival	Burst	Waiting	Turnaround	Completion
P1	0	5	4	9	9
P2	1	3	2	5	6

Gantt Chart



```
Average Waiting Time: 3.00
Average Turnaround Time: 7.00
```

Explanation:

Round Robin (RR) is a preemptive CPU scheduling algorithm where each process gets executed for a fixed time quantum (time slice). The processes are executed in the order they arrive, and if a process doesn't finish within its allocated quantum, it is preempted and added back to the ready queue with the remaining burst time.

- **Time Quantum:** It is the maximum time a process can run before being preempted. If the process does not finish within this time, it gets placed at the end of the queue and the CPU scheduler moves to the next process.

- **Context Switching:** When a process is preempted, the operating system saves the state of the process and moves it back to the ready queue.

- **Process Execution:** The first process starts executing. If it does not complete its execution within the given time quantum, it is preempted, and the next process in line gets the CPU for the same amount of time. The processes continue to execute in a round-robin fashion.

- **Waiting Time (WT):** The total time a process spends waiting in the ready queue before its execution begins.

$$\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time}$$

- **Turnaround Time (TAT):** The total time a process takes from arrival to completion.

$$\text{Turnaround Time} = \text{Waiting Time} + \text{Burst Time}$$

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.

Code

```c
#include <stdio.h>

void priorityScheduling(int n, int burst_time[], int priority[],
   int arrival_time[], int waiting_time[], int turnaround_time[],
   int completion_time[]) {
    int i, j;

    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arrival_time[j] > arrival_time[j + 1] ||
                (arrival_time[j] == arrival_time[j + 1] &&
                   priority[j] > priority[j + 1])) {
                int temp;

                temp = burst_time[j];
                burst_time[j] = burst_time[j + 1];
                burst_time[j + 1] = temp;

                temp = priority[j];
                priority[j] = priority[j + 1];
                priority[j + 1] = temp;

                temp = arrival_time[j];
                arrival_time[j] = arrival_time[j + 1];
                arrival_time[j + 1] = temp;
            }
        }
    }

    int current_time = 0;
    for (i = 0; i < n; i++) {
        if (current_time < arrival_time[i]) {
            current_time = arrival_time[i];
        }
        waiting_time[i] = current_time - arrival_time[i];
        completion_time[i] = current_time + burst_time[i];
        turnaround_time[i] = completion_time[i] - arrival_time[i
           ];
        current_time = completion_time[i];
    }
}

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

    int burst_time[num_processes], priority[num_processes],
       arrival_time[num_processes];
```

```c
    int waiting_time[num_processes], turnaround_time[
       num_processes], completion_time[num_processes];

    for (int i = 0; i < num_processes; i++) {
        printf("Enter Burst Time for P%d: ", i + 1);
        scanf("%d", &burst_time[i]);
        printf("Enter Priority for P%d (lower is higher): ", i +
           1);
        scanf("%d", &priority[i]);
        printf("Enter Arrival Time for P%d: ", i + 1);
        scanf("%d", &arrival_time[i]);
    }

    priorityScheduling(num_processes, burst_time, priority,
       arrival_time, waiting_time, turnaround_time,
       completion_time);

    printf("\nProcess\tArrival\tBurst\tPriority\tWaiting\
       tTurnaround\tCompletion\n");
    for (int i = 0; i < num_processes; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
            i + 1, arrival_time[i], burst_time[i], priority[i],
            waiting_time[i], turnaround_time[i], completion_time[
               i]);
    }

    float avg_waiting = 0, avg_turnaround = 0;
    for (int i = 0; i < num_processes; i++) {
        avg_waiting += waiting_time[i];
        avg_turnaround += turnaround_time[i];
    }
    avg_waiting /= num_processes;
    avg_turnaround /= num_processes;

    printf("\nAverage Waiting Time: %.2f\n", avg_waiting);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround);

    return 0;
}
```

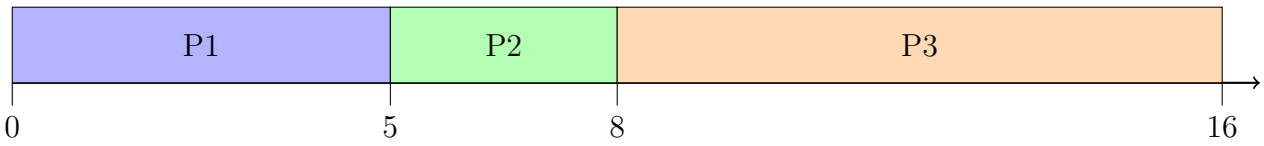
Output

```
Enter number of processes: 3
Enter Burst Time for P1: 5
Enter Priority for P1 (lower is higher): 2
Enter Arrival Time for P1: 0
Enter Burst Time for P2: 3
Enter Priority for P2 (lower is higher): 1
Enter Arrival Time for P2: 1
Enter Burst Time for P3: 8
```

```
Enter Priority for P3 (lower is higher): 3
Enter Arrival Time for P3: 2
```

Process	Arrival	Burst	Priority	Waiting	Turnaround	Completion
P1	0	5	2	0	5	5
P2	1	3	1	4	7	8
P3	2	8	3	6	14	16

Gantt Chart




```
Average Waiting Time: 3.33
Average Turnaround Time: 8.67
```

Explanation

Priority CPU Scheduling is an algorithm where each process is assigned a priority, and the CPU is allocated to the process with the highest priority. If two processes have the same priority, scheduling may follow First-Come, First-Served (FCFS). It can be preemptive (higher priority process interrupts the running process) or non-preemptive (current process runs until completion). Lower priority processes may suffer from starvation, which can be prevented using aging (increasing priority over time). It is commonly used in real-time systems where some tasks require immediate execution.

2 a. Programs on Process Management System Calls

What is a Process Management System Call

A **Process Management System Call** is used to create, control, and manage processes in an operating system. These system calls enable process creation, termination, synchronization, and execution of new programs.

Types of Process Management System Calls

Category	Description and Examples
Process Creation	Creates a new child process using `fork()`.
Process Termination	Terminates a process using `exit()`.
Process Synchronization	Ensures the parent process waits for the child to complete using `wait()` or `waitpid()`.
Process Execution	Loads and executes a new program using `exec()`.

Table 1: Types of Process Management System Calls

Process Management in C using System Calls

This C program demonstrates process creation, execution, termination, and synchronization using low-level system calls in Unix-like operating systems.

Algorithm

1. **Creating a New Process**: The program starts by calling the `fork()` system call. This creates a new child process by duplicating the parent process.
    - If `fork()` is successful, execution continues in both the parent and child processes.
    - If `fork()` fails (returns `-1`), the program prints an error message and exits.
2. **Identifying Process**: The return value of `fork()` determines whether the process is the parent or child.
    - If the return value is `0`, it indicates the child process.
    - If the return value is a positive integer (child PID), it indicates the parent process.
    - If the return value is `-1`, it means process creation failed.
3. **Executing a New Program (Child Process)**: The child process calls an `exec()` family function (e.g., `execl()`, `execvp()`, or `execv()`) to replace its memory space with a new program.

- If `exec()` is successful, the child process will start executing the new program, and no further instructions in the child's original code will execute.
- If `exec()` fails, an error message is printed, and the child process exits with a failure status.

4. **Waiting for Child Process (Parent Process)**: The parent process calls `wait()` or `waitpid()` to pause execution until the child process finishes.
   - The `wait()` function makes the parent wait for any child process to terminate.
   - The `waitpid()` function allows the parent to wait for a specific child process.
   - If no child processes exist, `wait()` returns immediately.
5. **Terminating the Process**: Once the child process completes execution, it calls `exit()` to terminate and return an exit status to the parent.
   - The exit status can be captured by the parent process using `wait()` or `waitpid()`.
   - The parent process resumes execution after the child terminates.
6. **Resuming Parent Execution**: After the child process has terminated, the parent prints a message indicating that it is resuming execution. The program then exits successfully.

Explanation of Key System Calls and Functions

1. `fork()`
   - **Purpose**: Creates a new process by duplicating the current process.
   - **Return Value**:
     - Returns 0 in the child process.
     - Returns child PID in the parent process.
     - Returns `-1` on failure.
2. `exit()`
   - **Purpose**: Terminates the calling process.
   - **Parameters**: Exit status code.
3. `wait()`
   - **Purpose**: Parent waits for any child process to terminate.
   - **Return Value**: Returns the PID of the terminated child.
4. `waitpid()`
   - **Purpose**: Parent waits for a specific child process.
   - **Parameters**: PID of the child process.
5. `exec()`
   - **Purpose**: Replaces the current process with a new program.
   - **Variants**: `execl()`, `execp()`, `execv()`.

Step-by-Step Program Flow

1. **The program starts and creates a child process using `fork()`.**
    - The `fork()` system call is used to create a duplicate of the current process.
    - If `fork()` is successful, two processes exist: the **parent process** and the **child process**.
    - The return value of `fork()` determines the process role:
        - **Return value = 0** → The process is the child.
        - **Return value ¿ 0** → The process is the parent (child's PID is returned).
        - **Return value = -1** → Process creation failed, an error message is displayed, and the program exits.
2. **The parent and child processes execute different code paths.**
    - After `fork()`, both the parent and child execute the same code but take different paths based on the return value.
    - The **child process** proceeds to execute a new program.
    - The **parent process** waits for the child process to finish.
3. **The child process calls `execl()` to execute a new program.**
    - The child replaces its current process image with a new program using the `execl()` system call.
    - Example: The child process executes the 'ls -l' command.
    - If `execl()` is successful:
        - The child process stops executing any further code from the original program.
        - Instead, it starts executing the new program.
    - If `execl()` fails:
        - An error message is displayed using `perror()`.
        - The child process terminates with a failure exit status.
4. **The parent process waits for the child process using `wait()`.**
    - The parent process does not immediately continue execution.
    - Instead, it calls `wait()`, causing it to pause until the child process terminates.
    - This ensures that the child completes execution before the parent resumes.
5. **The child process terminates using `exit()`.**
    - If the `execl()` system call was successful, the child process completes execution and terminates naturally.
    - If an error occurred, the child explicitly terminates using `exit(1)`.
    - The exit status of the child process is passed to the parent.
6. **The parent resumes execution after the child terminates.**

- Once `wait()` detects that the child process has finished, the parent resumes execution.
- The parent may print a message indicating that the child has finished.
- Finally, the parent process completes execution and terminates.

Code

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

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

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        printf("Child process (PID: %d) executing new program...\n", getpid());

        // Execute a new program
        execl("/bin/ls", "ls", "-l", NULL);

        // If execl fails
        perror("Exec failed");
        exit(1);
    }
    else {
        // Parent process
        printf("Parent process (PID: %d) waiting for child...\n",
            getpid());

        // Wait for child to terminate
        wait(NULL);

        printf("Child process terminated. Parent resuming execution.\n");
    }

    return 0;
}
```

Explanation

This program demonstrates process creation and management in C using system calls:

- The **parent process** creates a **child process** using ‘fork()‘. - The **child process** executes the ‘ls -l‘ command using ‘execl()‘. - The **parent process** waits for the child to finish using ‘wait()‘. - Once the child exits, the parent resumes execution.

2 b. Programs on System Calls

What is a System call

A **System Call** is a mechanism that allows a user-space program to request services from the operating system (OS) kernel. It acts as an interface between a process and the OS, enabling programs to perform operations like file handling, process control, and device management.

Types of System Calls

System calls are categorized based on their functionality:

Category	Description	Examples
Process Control	Create, execute, and terminate processes.	`fork()`, `exec()`, `exit()`
File Management	Operations on files such as open, read, write, and close.	`open()`, `read()`, `write()`, `close()`
Device Management	Request and release access to hardware devices.	`ioctl()`, `read()`, `write()`
Information Maintenance	Get or set system-related information.	`getpid()`, `stat()`, `uname()`
Communication	Facilitate inter-process communication (IPC).	`pipe()`, `socket()`, `send()`, `recv()`

Table 2: Types of System Calls in OS

File Handling in C using System Calls

This C program demonstrates various file I/O operations using low-level system calls in Unix-like operating systems. It covers file creation, reading and writing, repositioning file pointers, retrieving file metadata, and listing directory contents. Below is a detailed explanation of the algorithm and the system calls used.

Algorithm

1. **Opening a File**: The program first attempts to open a file `"example.txt"` for writing. If the file doesn't exist, it is created, and if it exists, its content is truncated (erased). This is done using the `open()` system call with the flags `O_WRONLY | O_CREAT | O_TRUNC`.

2. **Writing to the File**: After opening the file, the program writes a string `"Hello, this is a test file.
n"` to it using the `write()` system call. The program checks how many bytes were written and handles errors if any occur.

3. **Closing the File**: After writing, the file is closed using the `close()` system call. This ensures that all resources related to the file are freed.

4. **Reopening the File for Reading**: The program then opens the same file again but in read-only mode (`O_RDONLY`) to read its contents.

5. **Reading from the File**: The program reads the contents of the file into a buffer using the `read()` system call and prints the data to the console. It checks for errors, such as end-of-file (EOF) or read failures.

6. **Repositioning the File Pointer**: The program uses `lseek()` to reset the file pointer to the beginning of the file. This demonstrates file pointer manipulation by moving the pointer and then reading the file again.

7. **Reading Again**: After repositioning the file pointer, the program reads the file content again to show that it has returned to the beginning of the file.

8. **Getting File Information**: The program retrieves file metadata, such as the file size, using the `stat()` system call. It prints this information to the console.

9. **Listing Directory Contents**: Finally, the program lists the contents of the current directory using `opendir()` and `readdir()`. It prints the names of all files and directories in the current directory.

Explanation of Key System Calls and Functions

1. `open()`
   - **Purpose**: Opens or creates a file with specific access permissions.
   - **Parameters**:
     - `pathname`: The path of the file.
     - `flags`: Access mode flags (`O_WRONLY` for writing, `O_CREAT` for creating if it doesn't exist, `O_TRUNC` to truncate the file).
   - **Return Value**: Returns the file descriptor (an integer). If it fails, it returns `-1`.

2. `write()`
   - **Purpose**: Writes data to the file.
   - **Parameters**:
     - `fd`: The file descriptor.
     - `buf`: A pointer to the data to write.
     - `count`: The number of bytes to write.
   - **Return Value**: Returns the number of bytes written. If an error occurs, it returns `-1`.

3. `close()`
   - **Purpose**: Closes the file, freeing up resources.
   - **Parameters**:
     - `fd`: The file descriptor to close.
   - **Return Value**: Returns `0` on success or `-1` on error.

4. `read()`

- **Purpose**: Reads data from the file.
- **Parameters**:
  - `fd`: The file descriptor.
  - `buf`: A pointer to the buffer to store the data.
  - `count`: The number of bytes to read.
- **Return Value**: Returns the number of bytes read. A return value of `0` indicates EOF, and `-1` indicates an error.

5. `lseek()`
   - **Purpose**: Moves the file pointer to a specific position.
   - **Parameters**:
     - `fd`: The file descriptor.
     - `offset`: The number of bytes to move the pointer.
     - `whence`: A reference point for the offset (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).
   - **Return Value**: Returns the new file offset (in bytes) from the beginning of the file.
6. `stat()`
   - **Purpose**: Retrieves file information like size, permissions, etc.
   - **Parameters**:
     - `pathname`: The path of the file.
     - `statbuf`: A pointer to a `struct stat` where file metadata is stored.
   - **Return Value**: Returns `0` on success, and `-1` on failure.
7. `opendir()`
   - **Purpose**: Opens a directory for reading.
   - **Parameters**:
     - `name`: The path of the directory to open.
   - **Return Value**: Returns a pointer to a `DIR` structure for reading the directory, or `NULL` if an error occurs.
8. `readdir()`
   - **Purpose**: Reads a directory entry.
   - **Parameters**:
     - `dirp`: The pointer to the `DIR` structure returned by `opendir()`.
   - **Return Value**: Returns a pointer to a `struct dirent` representing the directory entry, or `NULL` when the end of the directory is reached.
9. `closedir()`
   - **Purpose**: Closes the directory stream.
   - **Parameters**:
     - `dirp`: The pointer to the `DIR` structure to close.
   - **Return Value**: Returns `0` on success, or `-1` on failure.

Step-by-Step Program Flow

1. **Opening the File**: The program first opens `"example.txt"` for writing using the `open()` system call with flags `O_WRONLY | O_CREAT | O_TRUNC`. If the file is opened successfully, it proceeds to the next step; otherwise, it exits with an error.

2. **Writing to the File**: Using `write()`, the program writes the string `"Hello, this is a test file.
n"` to the file. It checks the number of bytes written and handles any errors.

3. **Closing the File**: The program then calls `close()` to close the file, ensuring that all changes are saved and resources are freed.

4. **Reopening the File for Reading**: The program opens the file again, but this time in read-only mode (`O_RDONLY`), using `open()`.

5. **Reading the File**: Using `read()`, the program reads the contents of the file into a buffer and prints it to the console. If any errors occur during reading, it handles them appropriately.

6. **Repositioning the File Pointer**: The program uses `lseek()` to reset the file pointer to the beginning of the file, demonstrating how to manipulate the file pointer.

7. **Reading Again**: After repositioning the pointer, the program reads the file again, demonstrating that the pointer was successfully moved.

8. **Getting File Metadata**: The program uses `stat()` to retrieve and print the file's size and other metadata.

9. **Listing Directory Contents**: Finally, the program lists the contents of the current directory by opening the directory with `opendir()`, reading each entry with `readdir()`, and closing the directory with `closedir()`.

Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>

int main() {
    // Open a file for writing
    int file = open("example.txt", O_WRONLY | O_CREAT | O_TRUNC);
    if (file == -1) {
        perror("Open failed");
        return 1;
    }

    // Write to the file
```

```
18      const char *text = "Hello, this is a test file.\n";
19      ssize_t bytesWritten = write(file, text, strlen(text));
20      if (bytesWritten == -1) {
21          perror("Write failed");
22          close(file);
23          return 1;
24      }
25
26      // Close the file
27      close(file);
28
29      // Open the file for reading
30      file = open("example.txt", O_RDONLY);
31      if (file == -1) {
32          perror("Open failed");
33          return 1;
34      }
35
36      // Read from the file
37      char buffer[strlen(text)];
38      ssize_t bytesRead = read(file, buffer, strlen(text));
39      if (bytesRead == -1) {
40          perror("Read failed");
41          close(file);
42          return 1;
43      }
44      buffer[bytesRead] = '\0'; // Null-terminate the string
45      printf("Content of example.txt: %s\n", buffer);
46
47      // Use lseek to move the file pointer to the beginning
48      if (lseek(file, 0, SEEK_SET) == -1) {
49          perror("Lseek failed");
50          close(file);
51          return 1;
52      }
53
54      // Read again from the beginning
55      bytesRead = read(file, buffer, strlen(text));
56      buffer[bytesRead] = '\0';
57      printf("Content after repositioning: %s\n", buffer);
58
59      // Get file info using stat
60      struct stat fileStat;
61      if (stat("example.txt", &fileStat) == -1) {
62          perror("Stat failed");
63          close(file);
64          return 1;
65      }
66      printf("File size: %ld bytes\n", fileStat.st_size);
67
68      // List directory contents
```

```
    DIR *dir = opendir(".");
    if (!dir) {
        perror("Failed to open directory");
        close(file);
        return 1;
    }

    struct dirent *entry;
    printf("Directory contents:\n");
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    closedir(dir);
    close(file);
    return 0;
}
```

Explanation

This program demonstrates file handling in C using low-level system calls in Unix-like operating systems. It shows how to open, write to, and read from files using ‘open()‘, ‘write()‘, and ‘read()‘. The file pointer is manipulated with ‘lseek()‘, and file metadata, such as file size, is retrieved using ‘stat()‘. Additionally, the program lists the contents of the current directory with ‘opendir()‘ and ‘readdir()‘. By using these system calls, the program illustrates basic file and directory operations, providing insight into low-level file management in Unix-like systems.

3.Write a program to simulate Banker's Algorithm for Deadlock Avoidance and Prevention.

Banker's Algorithm

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

Algorithm

1. Initialize the number of processes, resources, and matrices:
   (a) Input the number of processes and resources.
   (b) Input the total number of instances for each resource.
   (c) Input the allocation matrix and the maximum matrix for each process.
2. Calculate the available resources for each resource type by subtracting the sum of allocated resources from the total instances.
3. Calculate the need matrix, which is the difference between the maximum resources required and the currently allocated resources for each process.
4. Implement the Banker's algorithm to check for a safe sequence:
   (a) For each process, check if its needed resources are less than or equal to the available resources.
   (b) If the process can execute, simulate its execution by adding its allocated resources to the available resources and mark the process as finished.
   (c) If all processes can finish, print the safe sequence; otherwise, print a deadlock message.

Procedure

- Start the program and prompt the user for the number of processes and resources.
- Input the allocation matrix and the maximum matrix.
- Calculate the available resources by subtracting the sum of allocated resources from the total instances of each resource.
- Calculate the need matrix as the difference between the maximum and allocated resources for each process.
- Use the Banker's algorithm to determine whether a safe sequence exists or if the system will encounter a deadlock.
- If a safe sequence exists, print it. If deadlock is detected, print a message indicating the occurrence of deadlock.

Code

```c
#include<stdio.h>

int main() {
    // Inputs for number of processes, number of resources,
       instances of each resource,
    // allocation matrix and maximum matrix
    int n, r;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resources: ");
    scanf("%d", &r);

    int all[n][r], max[n][r], ins[r], ava[r];

    // Input instances of each resource
    printf("Enter the instances of each resource: ");
    for(int i = 0; i < r; i++) {
        scanf("%d", &ins[i]);
    }

    // Input allocation matrix
    printf("Enter the allocation matrix of n x r:\n");
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < r; j++) {
            scanf("%d", &all[i][j]);
        }
    }

    // Input maximum matrix
    printf("Enter the maximum matrix of n x r:\n");
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < r; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    // Calculate available resources
    printf("Available resources are: ");
    int sum = 0;
    for(int i = 0; i < r; i++) {
        sum = 0;
        for(int j = 0; j < n; j++) {
            sum += all[j][i];
        }
        ava[i] = ins[i] - sum;
        printf("%d ", ava[i]);
```

```
    }

    // Calculate need matrix
    int need[n][r];
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < r; j++) {
            need[i][j] = max[i][j] - all[i][j];
        }
    }

    printf("\nPro\tAllo\tMax\tNeed\tAvai\n");
    for(int i = 0; i < n; i++) {
        printf("P%d\t", i); // Print process number
        for(int j = 0; j < r; j++) { // Print Allocation for
           process i
            printf("%d ", all[i][j]);
        }
        printf("\t");
        for(int j = 0; j < r; j++) { // Print Maximum for
           process i
            printf("%d ", max[i][j]);
        }
        printf("\t");
        for(int j = 0; j < r; j++) { // Print Need for
           process i
            printf("%d ", need[i][j]);
        }
        printf("\t");
        if(i == 0) { // Print Available resources
            for(int j = 0; j < r; j++) {
                printf("%d ", ava[j]);
            }
            printf("\t");
        }
        printf("\n");
    }

    // Banker's Algorithm main logic
    int f[n], ans[n], idx = 0;
    for(int i = 0; i < n; i++) {
        f[i] = 0;
    }

    for(int a = 0; a < n; a++) {
        for(int i = 0; i < n; i++) {
            if(f[i] == 0) {
                int flag = 0;
                for(int j = 0; j < r; j++) {
```

```
                    if(ava[j] < need[i][j]) {
                        flag = 1;
                        break;
                    }
                }

                if(flag == 0) {
                    printf("\nBefore the available resources:
                        ");
                    for(int b = 0; b < r; b++) {
                        printf("%d ", ava[b]);
                    }

                    ans[idx++] = i;
                    for(int k = 0; k < r; k++) {
                        ava[k] += all[i][k];
                    }
                    f[i] = 1;
                    printf("\nP%d reached a safe state ", i);
                    printf("\nNow the available resources are
                       : ");
                    for(int c = 0; c < r; c++) {
                        printf("%d ", ava[c]);
                    }
                    printf("\n");
                }
            }
        }
    }

    // Checking for deadlock
    int temp = 0;
    for(int i = 0; i < n; i++) {
        if(f[i] == 0) {
            temp = 1;
            printf("Deadlock occurred\n");
            break;
        }
    }

    if(temp == 0) {
        printf("\nThe safe sequence is: ");
        for(int i = 0; i < n - 1; i++) {
            printf("P%d->", ans[i]);
        }
        printf("P%d", ans[n - 1]);
    }

```

```
    return 0;
}
```

Output

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the instances of each resource: 10 5 7
Enter the allocation matrix of n x r:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the maximum matrix of n x r:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Available resources are: 3 3 2
```

Pid	Allocated	Maximum	Need	Available
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	3 3 2
P2	3 0 2	9 0 2	6 0 0	3 3 2
P3	2 1 1	2 2 2	0 1 1	3 3 2
P4	0 0 2	4 3 3	4 3 1	3 3 2

```
Before the available resources: 3 3 2
P1 reached a safe state
Now the available resources are: 5 3 2

Before the available resources: 5 3 2
P3 reached a safe state
Now the available resources are: 7 4 3

Before the available resources: 7 4 3
P4 reached a safe state
Now the available resources are: 7 4 5

Before the available resources: 7 4 5
P0 reached a safe state
Now the available resources are: 7 5 5
```

```
Before the available resources: 7 5 5
P2 reached a safe state
Now the available resources are: 10 5 7

The safe sequence is: P1->P3->P4->P0->P2
```

Explanation

- The program implements the Banker's Algorithm, which checks if a system can allocate resources to processes without causing a deadlock.
- It calculates the available resources by subtracting the allocated resources from the total instances of each resource.
- It then calculates the need matrix as the difference between the maximum required resources and the allocated resources for each process.
- The algorithm checks whether all processes can finish without causing a deadlock, thus ensuring a safe sequence.
- If a safe sequence exists, it prints it ; otherwise, it reports a deadlock.

4.Write a program to implement the Producer-Consumer problem using semaphores using UNIX/LINUX system calls.

Producer-Consumer Problem

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

Algorithm

1. Initialize the semaphores(mutex, empty and full).
2. Producer function:
   (a) Check if there's space in the buffer: If the buffer is full (i.e., `sem_trywait(&empty)` returns non-zero), print a message and return.
   (b) Acquire mutex to enter the critical section.
   (c) Add a random item to the buffer.
   (d) Update the in index (the position where the item was added).
   (e) Release mutex.
   (f) Signal full to indicate there's one more item in the buffer.
3. Consumer function:
   (a) Check if there's an item to consume: If the buffer is empty (i.e., `sem_trywait(&full)` returns non-zero), print a message and return.
   (b) Acquire mutex to enter the critical section.
   (c) Remove an item from the buffer.
   (d) Update the out index (the position from where the item was consumed).
   (e) Release mutex.
   (f) Signal empty to indicate there's one more empty space in the buffer.

Procedure

- Start the program
- Prompt the user for a valid buffer size.
- Initialize semaphores (mutex, empty, full) with the buffer size and initial conditions.
- Display options for the user: Produce, Consume, or Exit.
- Based on the selection:
  1. Produce: Create the producer thread and wait for it to finish using pthread_join.
  2. Create the consumer thread and wait for it to finish using pthread_join.

3. Exit: Destroy semaphores and exit the program.

- Produce Action:
  1. Check if there is space in the buffer.
  2. Add a random item to the buffer if there is space.
  3. Print a message indicating the item was produced.
- Consume Action:
  1. Check if there is an item to consume.
  2. Consume an item if there's any available in the buffer.
  3. Print a message indicating the item was consumed.

Code

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

#define MAX_BUFFER_SIZE 10

int buffer[MAX_BUFFER_SIZE];
int buffer_size;
int in = 0, out = 0;

sem_t mutex, empty, full;

void* producer(void* arg) {
    int item;
    item = rand() % 100;

    // Check if the buffer is full
    if (sem_trywait(&empty) != 0) {  // Try to acquire an empty slot
        printf("Buffer is full, producer cannot produce.\n");
        return NULL;
    }

    sem_wait(&mutex);  // Enter critical section
    buffer[in] = item;
    in = (in + 1) % buffer_size;
    printf("Producer produced item: %d\n", item);
    sem_post(&mutex);  // Exit critical section
    sem_post(&full);   // Signal that there is one more item in the buffer
```

```
31
32      return NULL;
33  }
34
35  void* consumer(void* arg) {
36      int item;
37
38      // Check if the buffer is empty
39      if (sem_trywait(&full) != 0) {  // Try to acquire a full
            slot
40          printf("Buffer is empty, consumer cannot consume.\n")
                ;
41          return NULL;
42      }
43
44      sem_wait(&mutex);  // Enter critical section
45      item = buffer[out];
46      out = (out + 1) % buffer_size;
47      printf("Consumer consumed item: %d\n", item);
48      sem_post(&mutex);  // Exit critical section
49      sem_post(&empty);  // Signal that there is one more empty
            slot in the buffer
50
51      return NULL;
52  }
53
54  int main() {
55      pthread_t prod_thread, cons_thread;
56      int choice;
57
58      printf("Enter the buffer size: ");
59      scanf("%d", &buffer_size);
60
61      if (buffer_size <= 0 || buffer_size > MAX_BUFFER_SIZE) {
62          printf("Invalid buffer size. Please choose a value
                between 1 and %d.\n", MAX_BUFFER_SIZE);
63          return -1;
64      }
65
66      sem_init(&mutex, 0, 1);   // Mutex for critical section
67      sem_init(&empty, 0, buffer_size);  // Semaphore for empty
            slots
68      sem_init(&full, 0, 0);    // Semaphore for full slots
69
70      while (1) {
71          printf("\nSelect an action:\n");
72          printf("1.Produce\n");
73          printf("2.Consume\n");
```

```
        printf("3.Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                pthread_create(&prod_thread, NULL, producer, NULL);
                pthread_join(prod_thread, NULL);  // Wait for producer thread to finish
                break;
            case 2:
                pthread_create(&cons_thread, NULL, consumer, NULL);
                pthread_join(cons_thread, NULL);  // Wait for consumer thread to finish
                break;
            case 3:
                printf("Exiting program.\n");
                sem_destroy(&mutex);
                sem_destroy(&empty);
                sem_destroy(&full);
                return 0;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Output

```
Enter the buffer size: 5

Select an action:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer produced item: 33

Select an action:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
```

```
Producer produced item: 42

Select an action:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumed item: 33

Select an action:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumed item: 42

Select an action:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Buffer is empty, consumer cannot consume.
```

Explanation

- The program implements the producer-consumer problem using mutual exclusion.
- The producer increases the full slots and decreases the empty slots when producing an item.
- The consumer decreases the full slots and increases the empty slots when consuming an item.
- The program ensures synchronization between producer and consumer.

5.Write a program to illustrate the following IPC mechanisms

5 a. Pipes

Inter-Process Communication (IPC) allows processes to exchange data. One common IPC mechanism is the **pipe**, which facilitates unidirectional communication between processes. This program demonstrates the use of a pipe in C, where the parent process writes data to the pipe, and the child process reads from it.

Pipes in IPC

A **pipe** is one of the simplest IPC mechanisms. It allows unidirectional data transfer between a parent and a child process.

Characteristics of Pipes

- Data flows in **one direction** only.
- Used for communication between **related processes** (parent-child).
- Follows the **First-In-First-Out (FIFO)** principle.
- Cannot be used for unrelated processes unless **named pipes (FIFOs)** are used.

Creating and Using a Pipe in C

A pipe is created using the `pipe()` system call, which returns two file descriptors:

- `fd[0]` – Read end of the pipe.
- `fd[1]` – Write end of the pipe.

Algorithm

The following steps outline the execution flow:

1. Create a pipe using the `pipe()` system call.
2. Create a child process using the `fork()` system call.
3. Check if the fork was successful.
4. If in the child process:
   (a) Close the write end of the pipe.
   (b) Read data from the pipe.
   (c) Close the read end of the pipe.
   (d) Print the received message.
5. If in the parent process:

(a) Close the read end of the pipe.

(b) Write a message into the pipe.

(c) Close the write end of the pipe.

(d) Print a confirmation message.

6. End the program.

Procedure

The step-by-step procedure for executing this program is as follows:

1. Include necessary header files (`stdio.h`, `string.h`, `unistd.h`).
2. Declare an integer array `fd[2]` to store pipe file descriptors.
3. Create a buffer to store the message.
4. Call `pipe(fd)` to create a communication channel.
5. Call `fork()` to create a child process.
6. If `fork()` fails, print an error and terminate.
7. If inside the child process:
   - Close the write end of the pipe.
   - Read data from the pipe.
   - Print the received message.
   - Close the read end.
8. If inside the parent process:
   - Close the read end of the pipe.
   - Write a message into the pipe.
   - Print a confirmation message.
   - Close the write end.
9. Return 0 to indicate successful execution.

Code

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

int main(){
    int fd[2];
    char buffer[100];

```

```
    if(pipe(fd) == -1){
        perror("pipe failed");
        return 1;
    }

    pid_t pid = fork();

    if(pid == -1){
        perror("fork failed");
        return 1;
    }
    else if(pid == 0){
        close(fd[1]);
        read(fd[0], buffer, sizeof(buffer));
        close(fd[0]);
        printf("Child process read from the pipe: %s\n", buffer);
    }
    else{
        close(fd[0]);
        char *msg = "Hello from parent process";
        write(fd[1], msg, strlen(msg) + 1);
        close(fd[1]);
        printf("Parent process wrote to the pipe: %s\n", msg);
    }

    return 0;
}
```

Output

Parent process wrote to the pipe: Hello from parent process
Child process read from the pipe: Hello from parent process

Explanation

This program demonstrates Inter-Process Communication (IPC) using a pipe. It creates a pipe and then forks a child process. The parent process writes a message into the pipe, which the child process reads. The file descriptors are closed appropriately to ensure proper resource management. This example highlights basic IPC and process synchronization in a Unix-based system.

5 b. FIFO

Inter-Process Communication (IPC) is a mechanism that allows processes to exchange data. One of the IPC methods is **FIFO (First-In-First-Out) Pipes**, also known as **Named Pipes**. Unlike regular pipes, FIFOs allow communication between **unrelated processes** because they are identified by a name in the file system.

A FIFO is created using the `mkfifo()` system call, and data written to it by one process can be read by another. This enables **bidirectional communication** between two processes.

Algorithm

The following steps outline the working of the FIFO-based IPC system:

1. Create a named pipe (FIFO) using the `mkfifo()` function.
2. Open the FIFO for reading or writing in each process.
3. In the **reader process**:
   - Open the FIFO in read mode.
   - Read data from the FIFO.
   - Display the received message.
   - Open the FIFO in write mode.
   - Write the response message to the FIFO.
4. In the **writer process**:
   - Open the FIFO in write mode.
   - Write a message to the FIFO.
   - Open the FIFO in read mode.
   - Read and display the response from the FIFO.
5. Repeat the process for continuous communication.

Procedure

The following steps outline the implementation of FIFO-based IPC:

1. Include necessary header files: `stdio.h`, `fcntl.h`, `sys/types.h`, `sys/stat.h`, and `unistd.h`.
2. Declare the FIFO file path as `/tmp/myfifo`.
3. Use `mkfifo()` to create the FIFO file.
4. Implement the **reader process**:
   - Open the FIFO for reading using `open()`.
   - Read data using `read()` and print the received message.

- Open the FIFO for writing using `open()`.
- Write a response message using `write()`.

5. Implement the **writer process**:
   - Open the FIFO for writing using `open()`.
   - Write a message using `write()`.
   - Open the FIFO for reading using `open()`.
   - Read the response message and print it.
6. Repeat the process in a loop for continuous exchange of messages.
7. Close the FIFO using `close()` after each operation.

Code Implementation

Reader Process

```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd1;
    char * myfifo = "/tmp/myfifo";

    mkfifo(myfifo, 0666);

    char str1[80], str2[80];
    while (1)
    {
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        printf("User1: %s\n", str1);
        close(fd1);

        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

Writer Process

```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;

    char * myfifo = "/tmp/myfifo";

    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1)
    {
        fd = open(myfifo, O_WRONLY);

        fgets(arr2, 80, stdin);

        write(fd, arr2, strlen(arr2)+1);
        close(fd);

        fd = open(myfifo, O_RDONLY);

        read(fd, arr1, sizeof(arr1));

        printf("User2: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```

Execution

To run these programs, follow these steps:

1. Compile the programs:

```
    gcc -o reader reader.c
    gcc -o writer writer.c
```

2. Open two separate terminals and run:

    - In Terminal 1: ‘./reader‘
    - In Terminal 2: ‘./writer‘

3. Type a message in one terminal and observe the communication.
4. Press 'Ctrl+C' to exit the programs.

Output

Terminal1 (Writer Program)

Hello from User1
User2: Hi there!
How's it going?
User2: All good, what about you?

Terminal2 (Reader Program)

User1: Hello from User1
Hi there!
User1: How's it going?
All good, what about you?

Explanation

This program demonstrates IPC using **FIFO (Named Pipes)** for communication between two processes.

Key Steps in the Reader Process

- The FIFO is created using `mkfifo()`.
- The process continuously reads messages from the FIFO using `read()`.
- It prints the received message to the console.
- The user enters a response, which is written back to the FIFO.

Key Steps in the Writer Process

- The FIFO is created using `mkfifo()`.
- The user enters a message, which is written to the FIFO using `write()`.
- The process then reads the response from the FIFO and displays it.

Conclusion

The FIFO IPC mechanism allows **unrelated processes** to communicate using a named pipe in the file system. The program demonstrates how a **reader process** and a **writer process** can exchange messages using FIFO. This approach is useful in scenarios requiring **simple and structured communication between processes**.

5 c. Message Queues

Inter-Process Communication (IPC) allows processes to exchange data and synchronize their execution. **Message Queues** provide an efficient IPC mechanism where processes communicate by sending and receiving messages in a queue.

Unlike **Pipes or Shared Memory**, message queues allow asynchronous communication, meaning: - The **sender** can send messages without waiting for the receiver. - The **receiver** can retrieve messages when needed.

In this implementation, we demonstrate IPC using **Message Queues**, where: - The **Sender** process places a message into the queue. - The **Receiver** process retrieves the message from the queue.

Algorithm

The following steps outline the working of the **Message Queue IPC system**:

Sender Process

1. Generate a **unique key** using `ftok()`.
2. Create a **message queue** using `msgget()`.
3. Define a message structure containing:
   - A **message type** (`msg_type`).
   - A **message text** (`msg_text`).
4. Prompt the user to enter a message.
5. Send the message to the queue using `msgsnd()`.
6. Print confirmation of message transmission.

Receiver Process

1. Generate the same **unique key** using `ftok()`.
2. Access the **message queue** using `msgget()`.
3. Define a message structure to store received data.
4. Receive the message from the queue using `msgrcv()`.
5. Print the received message.
6. Delete the message queue using `msgctl()`.

Procedure

The implementation follows these steps:

Sender Process

1. Include necessary headers: `stdio.h`, `sys/ipc.h`, `sys/msg.h`, `string.h`, `stdlib.h`.
2. Define a message structure with:
   - `msg_type` (message type).
   - `msg_text` (message content).
3. Generate a **unique key** using `ftok("progfile", 65)`.
4. Create a **message queue** using `msgget()`.
5. Accept a message from the user.
6. Send the message to the queue using `msgsnd()`.
7. Print confirmation of message transmission.

Receiver Process

1. Include necessary headers.
2. Define a message structure similar to the sender.
3. Generate the **same unique key** using `ftok()`.
4. Access the **message queue** using `msgget()`.
5. Receive the message from the queue using `msgrcv()`.
6. Print the received message.
7. Delete the message queue using `msgctl()`.

Code Implementation

Sender Process

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

#define MSG_SIZE 100

struct msg_buffer {
    long msg_type;
    char msg_text[MSG_SIZE];
};

int main() {
    key_t key;
```

```
    int msgid;
    struct msg_buffer message;

    key = ftok("progfile", 65);

    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget failed");
        exit(1);
    }

    message.msg_type = 1;
    printf("Enter message: ");
    fgets(message.msg_text, MSG_SIZE, stdin);

    if (msgsnd(msgid, &message, sizeof(message.msg_text), 0) ==
       -1) {
        perror("msgsnd failed");
        exit(1);
    }

    printf("Message sent: %s\n", message.msg_text);
    return 0;
}
```

Receiver Process

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

#define MSG_SIZE 100

struct msg_buffer {
    long msg_type;
    char msg_text[MSG_SIZE];
};

int main() {
    key_t key;
    int msgid;
    struct msg_buffer message;

    key = ftok("progfile", 65);

    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget failed");
        exit(1);
    }
```

```
25
26      if (msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0)
           == -1) {
27          perror("msgrcv failed");
28          exit(1);
29      }
30
31      printf("Received message: %s\n", message.msg_text);
32
33      msgctl(msgid, IPC_RMID, NULL);
34
35      return 0;
36  }
```

Execution

To run these programs, follow these steps:

1. Compile both sender and receiver:

```
1       gcc -o sender mq_sender.c
2       gcc -o receiver mq_receiver.c
```

2. Open two terminals:
   - Run ‘./receiver‘ in one terminal.
   - Run ‘./sender‘ in another terminal and enter a message.
3. Type a message in one terminal and observe the communication.
4. Press ‘Ctrl+C‘ to exit the programs.

Output

Terminal1 (Receiver Program)

Received message: Hello from Sender!

Terminal2 (Sender Program)

Enter message: Hello from Sender!
Message sent: Hello from Sender!

Explanation

This program demonstrates **Inter-Process Communication (IPC) using Message Queues**.

Sender Process

- Creates a message queue using `msgget()`.
- Accepts user input.
- Sends the message using `msgsnd()`.
- Prints confirmation.

Receiver Process

- Retrieves the message queue using `msgget()`.
- Receives the message using `msgrcv()`.
- Prints the received message.
- Deletes the message queue using `msgctl()`.

Conclusion

The Message Queue IPC Mechanism allows asynchronous communication between processes. It is suitable for: - Scenarios where sender and receiver do not need to run simultaneously. - Multi-process environments where multiple receivers can read from the queue.

Message Queues provide a structured and efficient IPC mechanism compared to pipes or shared memory.

5 d. Shared Memory

Inter-Process Communication (IPC) allows processes to exchange data and synchronize their execution. **Shared Memory IPC** is one of the fastest methods for communication between processes because it allows them to share a segment of memory.

In this program, we implement a **Producer-Consumer model** using shared memory. - The Producer writes a message into shared memory. - The Consumer reads the message and signals the producer that it has read the data.

This model ensures efficient communication between processes.

Algorithm

The following steps outline the Producer-Consumer shared memory IPC system:

Producer Process

1. Generate a unique key using the `ftok()` function.
2. Create a shared memory segment using `shmget()`.
3. Attach the shared memory to the process using `shmat()`.
4. Continuously:
   - Accept input from the user.
   - Store the message in shared memory.
   - Set the `available` flag to indicate data is ready.
   - Wait for the consumer to reset the flag.
5. Detach from the shared memory using `shmdt()`.

Consumer Process

1. Generate the same key using `ftok()`.
2. Access the shared memory segment using `shmget()`.
3. Attach the shared memory to the process using `shmat()`.
4. Continuously:
   - Check if new data is available.
   - Read and print the message.
   - Reset the `available` flag.
5. Detach from the shared memory using `shmdt()`.
6. Remove the shared memory segment using `shmctl()`.

Procedure

The implementation involves the following steps:

Producer Process

1. Include the required headers: `stdio.h`, `stdlib.h`, `sys/ipc.h`, `sys/shm.h`, `unistd.h`, and `string.h`.
2. Define a **shared memory size** (`SHM_SIZE`) and a structure to store the message and an availability flag.
3. Generate a unique key using `ftok()`.
4. Create a shared memory segment using `shmget()`.
5. Attach the shared memory to the process using `shmat()`.
6. Continuously:
   - Read a message from the user.
   - Store it in the shared memory.
   - Set the `available` flag to **1**.
   - Wait for the consumer to reset the flag.
7. Detach from the shared memory using `shmdt()`.

Consumer Process

1. Include the same header files as the producer.
2. Generate the same unique key using `ftok()`.
3. Access the shared memory segment using `shmget()`.
4. Attach the shared memory to the process using `shmat()`.
5. Continuously:
   - Wait until the `available` flag is **1**.
   - Read and print the message.
   - Reset the `available` flag to **0**.
6. Detach from the shared memory using `shmdt()`.
7. Remove the shared memory segment using `shmctl()`.

Code Implementation

Producer Process

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

```c

#define SHM_SIZE 1024

struct shared_data {
    char message[SHM_SIZE];
    int available;
};

int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, sizeof(struct shared_data), 0666 |
       IPC_CREAT);
    struct shared_data* data = (struct shared_data*)shmat(shmid,
       NULL, 0);

    while (1) {
        printf("Enter message to produce: ");
        fgets(data->message, SHM_SIZE, stdin);
        data->available = 1;
        while (data->available) {
            usleep(100000);
        }
    }
    shmdt(data);
    return 0;
}
```

Consumer Process

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

#define SHM_SIZE 1024

struct shared_data {
    char message[SHM_SIZE];
    int available;
};

int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, sizeof(struct shared_data), 0666 |
       IPC_CREAT);
    struct shared_data* data = (struct shared_data*)shmat(shmid,
       NULL, 0);

    while (1) {
        while (!data->available) {
```

```
            usleep(100000);
        }
        printf("Consumed: %s", data->message);
        data->available = 0;
    }
    shmdt(data);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

Execution

To run these programs, follow these steps:

1. Compile both writer and reader:

```
    gcc -o writer shm_writer.c
    gcc -o reader shm_reader.c
```

2. Open two terminals:
   - Run ‘./writer‘ in one terminal and enter data.
   - Run ‘./reader‘ in another terminal to read the shared memory.
3. Type a message in one terminal and observe the communication.
4. Press ‘Ctrl+C‘ to exit the programs.

Output

Terminal1(Producer Program)

Enter message to produce: Hello from Producer!
Enter message to produce: How are you?
Enter message to produce: Goodbye!

Terminal2(Consumer Program)

Consumed: Hello from Producer!
Consumed: How are you?
Consumed: Goodbye!

Explanation

This program demonstrates **Inter-Process Communication (IPC) using Shared Memory**.

Producer Process

- Creates a shared memory segment.
- Accepts input from the user.
- Stores the message in shared memory.
- Sets the `available` flag to **1**.
- Waits for the consumer to reset the flag.

Consumer Process

- Reads the message from shared memory when `available` flag is **1**.
- Prints the received message.
- Resets the `available` flag to **0**.

Conclusion

The Shared Memory IPC Mechanism allows fast and efficient communication between processes. It enables a producer to write data into shared memory while the consumer reads it. Unlike pipes or message queues, shared memory provides direct access, making it one of the fastest IPC methods.

6. Write a program to simulate the following memory management techniques

6 a. Paging

Paging is a memory management technique used in operating systems to handle memory allocation efficiently. Instead of using contiguous memory blocks, the process's address space is divided into fixed-size **pages**, and the physical memory is divided into equal-sized **frames**. A **page table** is maintained by the operating system to map logical pages to physical frames.

Why is Paging Needed?

- Avoids external fragmentation by using fixed-size pages.
- Allows processes to use non-contiguous memory, improving memory utilization.
- Supports virtual memory, enabling efficient memory access and management.

Algorithm

1. **Start**
2. Define **PAGE_SIZE** and total number of pages in memory.
3. Create a **page table** mapping logical pages to physical frames.
4. Take **logical address** as input.
5. Compute **Page Number**:

$$\text{Page Number} = \frac{\text{Logical Address}}{\text{PAGE\_SIZE}}$$

6. Compute **Offset**:

$$\text{Offset} = \text{Logical Address}\%\text{PAGE\_SIZE}$$

7. Check if **Page Number is valid**:
   - If valid, continue.
   - If invalid, display an **error** and exit.
8. Get **Frame Number** from the **page table** using the **Page Number**.
9. Compute the **Physical Address**:

$$\text{Physical Address} = (\text{Frame Number} \times \text{PAGE\_SIZE}) + \text{Offset}$$

10. Display the **Physical Address**.
11. **End**

Procedure

1. **Step 1: Divide Memory** The logical address space is divided into fixed-size **pages**, and the physical memory is divided into **frames** of the same size.
2. **Step 2: Create a Page Table** The operating system maintains a **page table** that maps logical pages to physical frames.
3. **Step 3: User Provides a Logical Address** A process generates a **logical address**, which consists of:
   - **Page Number** (to identify the page).
   - **Offset** (to locate the exact byte within the page).
4. **Step 4: Extract Page Number and Offset** The page number and offset are calculated as follows:

$$\text{Page Number} = \frac{\text{Logical Address}}{\text{Page Size}}$$

$$\text{Offset} = \text{Logical Address} \mod \text{Page Size}$$

5. **Step 5: Look Up the Page Table** The **page number** is used to find the corresponding **frame number** in the page table.
6. **Step 6: Check for Page Fault**
   - If the page is in memory, proceed to the next step.
   - If the page is missing (**page fault**), the OS loads it from secondary storage.
7. **Step 7: Compute the Physical Address** The physical address is calculated using the formula:

$$\text{Physical Address} = (\text{Frame Number} \times \text{Page Size}) + \text{Offset}$$

8. **Step 8: Access Memory** The process retrieves data from the computed **physical address**.
9. **Step 9: Return Result** The translated **physical address** is returned and used by the CPU.
10. **Step 10: Repeat for Next Memory Access** The process continues generating new logical addresses as needed.

Code Implementation

```c
// Paging simulation in C
#include <stdio.h>

#define PAGE_SIZE 4  // Each page has 4-byte size
#define TOTAL_PAGES 4  // Assume we have 4 pages

```

```c
int main() {
    // Page table mapping logical pages to physical frames
    int pageTable[TOTAL_PAGES] = {3, 1, 2, 0};

    int logicalAddress, pageNumber, offset, physicalAddress;

    // Taking logical address input
    printf("Enter logical address (Page Number * PAGE_SIZE +
       Offset): ");
    scanf("%d", &logicalAddress);

    // Calculate page number and offset
    pageNumber = logicalAddress / PAGE_SIZE;
    offset = logicalAddress % PAGE_SIZE;

    // Check if the page number is valid
    if (pageNumber >= TOTAL_PAGES) {
        printf("Error: Invalid Page Number!\n");
        return 1;
    }

    // Get frame number from the page table
    int frameNumber = pageTable[pageNumber];

    // Calculate physical address
    physicalAddress = (frameNumber * PAGE_SIZE) + offset;

    // Output results
    printf("Logical Address: %d\n", logicalAddress);
    printf("Page Number: %d, Offset: %d\n", pageNumber, offset);
    printf("Frame Number: %d\n", frameNumber);
    printf("Physical Address: %d\n", physicalAddress);

    return 0;
}
```

Explanation

Paging is a crucial memory management technique that enables efficient memory allocation and avoids external fragmentation. Here's how it works:

- Logical memory is divided into **pages**, while physical memory is divided into **frames**.
- The **page table** keeps track of where each logical page is stored in physical memory.
- When a process accesses memory, the OS looks up the page table and translates the logical address into a physical address.
- If a requested page is not in memory, a **page fault** occurs, and the OS loads the missing page from secondary storage.

- This technique eliminates external fragmentation but may cause **internal fragmentation** if a page is not fully used.

Input:

```
Enter logical address (Page Number * PAGE_SIZE + Offset): 9
```

Processing:

- Logical Address = 9
- Page Number = 9 / 4 = 2
- Offset = 9
- Frame Number (from page table) = 2
- Physical Address = (2 * 4) + 1 = 9

Output:

```
Logical Address: 9
Page Number: 2, Offset: 1
Frame Number: 2
Physical Address: 9
```

Conclusion

Paging is a widely used memory management technique in modern operating systems. It eliminates external fragmentation, allows efficient memory allocation, and enables virtual memory. However, it requires careful management of the page table and handling of page faults to ensure optimal performance.

6 b. Segmentation

Segmentation is a memory management technique that divides a process's address space into variable-sized segments based on logical divisions such as code, stack, and data. Unlike paging, which uses fixed-size blocks, segmentation allows for dynamic memory allocation and better logical organization of memory.

Why is Segmentation Needed?

- Provides logical grouping of related data (e.g., code, stack, heap).
- Allows variable-sized memory allocation, reducing internal fragmentation.
- Enables sharing of code segments between processes.
- Supports dynamic memory management, making memory allocation more efficient.

Algorithm for Segmentation

The following algorithm describes how segmentation translates a logical address into a physical address.

1. Divide the process into logical segments such as code, data, and stack.
2. Assign a segment table that stores the base address and length of each segment.
3. When a program generates a logical address, extract:
   (a) Segment number
   (b) Offset (displacement within the segment)
4. Look up the segment table to find the segment's base address.
5. Check if the offset is within the segment limit:
   (a) If valid, compute the physical address:

$$\text{Physical Address} = \text{Base Address} + \text{Offset}$$

   (b) If invalid, generate a segmentation fault.
6. Access memory using the calculated physical address.
7. Return data to the CPU.
8. Repeat for the next memory request.

Procedure for Segmentation

The procedure to implement segmentation is as follows.

1. Divide the process into logical segments such as code, data, and stack.
2. Assign each segment a base address and a length.
3. Store this information in the segment table.
4. When a process generates a logical address, extract:
   (a) Segment number (identifies the segment).
   (b) Offset (identifies location within the segment).
5. Look up the segment table for the segment's base address.
6. Validate the offset:
   (a) If offset is less than the segment size, compute the physical address.
   (b) If offset exceeds the segment size, generate a segmentation fault.
7. Access memory at the calculated physical address.
8. Return the requested data to the CPU.

Code Implementation

```c
// Segmentation Simulation in C
#include <stdio.h>

#define TOTAL_SEGMENTS 3  // Number of segments

// Structure to store segment table entries
struct Segment {
    int base;   // Base address of the segment
    int limit;  // Size (length) of the segment
};

int main() {
    struct Segment segmentTable[TOTAL_SEGMENTS] = {
        {100, 500},  // Segment 0 (Code)
        {700, 300},  // Segment 1 (Data)
        {1200, 200}  // Segment 2 (Stack)
    };

    int segmentNumber, offset, physicalAddress;

    // Taking logical address input
    printf("Enter Segment Number and Offset: ");
    scanf("%d %d", &segmentNumber, &offset);

```

```
    // Check if the segment number is valid
    if (segmentNumber >= TOTAL_SEGMENTS || segmentNumber < 0) {
        printf("Error: Invalid Segment Number!\n");
        return 1;
    }

    // Check if the offset is within segment limit
    if (offset >= segmentTable[segmentNumber].limit) {
        printf("Error: Segmentation Fault! Offset out of bounds.\
           n");
        return 1;
    }

    // Calculate the Physical Address
    physicalAddress = segmentTable[segmentNumber].base + offset;

    // Display Results
    printf("Segment Number: %d, Offset: %d\n", segmentNumber,
       offset);
    printf("Base Address: %d, Limit: %d\n", segmentTable[
       segmentNumber].base, segmentTable[segmentNumber].limit);
    printf("Physical Address: %d\n", physicalAddress);

    return 0;
}
```

Sample Input and Output

Input:

```
Enter Segment Number and Offset: 1 50
```

Processing:

- Segment Number = 1
- Offset = 50
- Base Address (from table) = 700
- Limit = 300
- Offset is valid since 50 ¡ 300
- Physical Address = 700 + 50 = 750

Output:

```
Segment Number: 1, Offset: 50
Base Address: 700, Limit: 300
Physical Address: 750
```

Example 2: Invalid Logical Address (Segmentation Fault)

Input:

```
Enter Segment Number and Offset: 2 250
```

Processing:

- Segment Number = 2
- Offset = 250
- Base Address = 1200, Limit = 200
- Offset is invalid since 250 ¿ 200

Output:

```
Error: Segmentation Fault! Offset out of bounds.
```

Conclusion

Segmentation provides a logical way to divide memory into variable-sized segments, reducing internal fragmentation and improving memory efficiency. However, it introduces external fragmentation and requires complex memory management.

7. Write a program to simulate Contiguous Memory Allocation techniques

7 a. First-Fit

AIM: To write a program in C to simulate the First-Fit contiguous memory allocation technique.
Algorithm:

1. Start the program.
2. Prompt the user to input the size of memory and the number of memory blocks.
3. Prompt the user to input the size of each memory block.
4. Prompt the user to input the number of processes.
5. For each process:
   (a) Prompt the user to input the size of the process.
   (b) Allocate the process to the first memory block that is large enough to accommodate it.
   (c) Update the remaining size of the memory block after allocation.
6. Display the allocation of memory blocks to each process.
7. End the program.

Procedure:

1. Declare variables for memory size, number of memory blocks, memory block sizes, number of processes, and process sizes.
2. Input the size of memory and the number of memory blocks from the user.
3. Input the size of each memory block from the user.
4. Input the number of processes and the size of each process from the user.
5. Allocate each process to the first block of memory that is large enough to accommodate it, and update the remaining size of the memory block after allocation.
6. Display the allocation of memory blocks to each process.

Code

```c
#include <stdio.h>
int main() {
    int memory_size, num_blocks, num_processes;
    int block_size[10], process_size[10], allocation[10];
    printf("Enter the size of memory: ");
    scanf("%d", &memory_size);
    printf("Enter the number of memory blocks: ");
    scanf("%d", &num_blocks);
    for (int i = 0; i < num_blocks; i++) {
        printf("Enter the size of memory block %d: ", i + 1);
        scanf("%d", &block_size[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the size of process %d: ", i + 1);
        scanf("%d", &process_size[i]);
        allocation[i] = -1;
    }
    for (int i = 0; i < num_processes; i++) {
        for (int j = 0; j < num_blocks; j++) {
            if (block_size[j] >= process_size[i]) {
                allocation[i] = j;
                block_size[j] -= process_size[i];
                break;
            }
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < num_processes; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\n", i + 1, process_size[i],
               allocation[i] + 1);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1,
               process_size[i]);
        }
    }
    return 0;
}
```

Output

```
Enter the size of memory: 1000
Enter the number of memory blocks: 5
Enter the size of memory block 1: 200
Enter the size of memory block 2: 300
Enter the size of memory block 3: 100
Enter the size of memory block 4: 150
```

```
Enter the size of memory block 5: 250
Enter the number of processes: 4
Enter the size of process 1: 120
Enter the size of process 2: 200
Enter the size of process 3: 350
Enter the size of process 4: 70
```

Process No.	Process Size	Block No.
1	120	1
2	200	2
3	350	Not Allocated
4	70	1

Explanation

The First Fit memory allocation algorithm assigns a process to the first available memory block that is large enough. It scans memory blocks sequentially and stops as soon as it finds a suitable block. This method is fast but can lead to external fragmentation due to leftover unused spaces. If no suitable block is found, the process remains unallocated.

7 b. Best-Fit

AIM: To write a program in C to simulate the Best-Fit contiguous memory allocation technique.

Algorithm:

1. Start the program.
2. Prompt the user to input the size of memory and the number of memory blocks.
3. Prompt the user to input the size of each memory block.
4. Prompt the user to input the number of processes.
5. For each process:
   (a) Prompt the user to input the size of the process.
   (b) Allocate the process to the best fitting block of memory (smallest block that can accommodate the process).
   (c) Update the remaining size of the memory block after allocation.
6. Display the allocation of memory blocks to each process.
7. End the program.

Procedure:

1. Declare variables for memory size, number of memory blocks, memory block sizes, number of processes, and process sizes.
2. Input the size of memory and the number of memory blocks from the user.
3. Input the size of each memory block from the user.
4. Input the number of processes and the size of each process from the user.
5. Allocate each process to the best fitting block of memory and update the remaining size of the memory block after allocation.
6. Display the allocation of memory blocks to each process.

Code

```c
#include <stdio.h>
int main() {
    int memory_size, num_blocks, num_processes;
    int block_size[10], process_size[10], allocation[10];
    printf("Enter the size of memory: ");
    scanf("%d", &memory_size);
    printf("Enter the number of memory blocks: ");
    scanf("%d", &num_blocks);
    for (int i = 0; i < num_blocks; i++) {
        printf("Enter the size of memory block %d: ", i + 1);
```

```
        scanf("%d", &block_size[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the size of process %d: ", i + 1);
        scanf("%d", &process_size[i]);
        allocation[i] = -1;
    }
    for (int i = 0; i < num_processes; i++) {
        int best_index = -1;
        for (int j = 0; j < num_blocks; j++) {
            if (block_size[j] >= process_size[i]) {
                if (best_index == -1 || block_size[j] <
                   block_size[best_index]) {
                    best_index = j;
                }
            }
        }
        if (best_index != -1) {
            allocation[i] = best_index;
            block_size[best_index] -= process_size[i];
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < num_processes; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\n", i + 1, process_size[i],
               allocation[i] + 1);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1,
               process_size[i]);
        }
    }
    return 0;
}
```

Output

```
Enter the size of memory: 1000
Enter the number of memory blocks: 5
Enter the size of memory block 1: 200
Enter the size of memory block 2: 300
Enter the size of memory block 3: 100
Enter the size of memory block 4: 150
Enter the size of memory block 5: 250
Enter the number of processes: 4
Enter the size of process 1: 120
Enter the size of process 2: 200
Enter the size of process 3: 350
```

```
Enter the size of process 4: 70
```

Process No.	Process Size	Block No.
1	120	4
2	200	1
3	350	Not Allocated
4	70	3

Explanation

The Best Fit memory allocation algorithm selects the smallest available block that is large enough to accommodate a process, aiming to minimize wasted space. However, this can lead to many small fragmented blocks, making future allocations difficult. Since it scans all blocks before selecting the best one, it is slower than First Fit. Best Fit is preferred when reducing external fragmentation is more important than allocation speed.

7 c. Worst-Fit

AIM: To write a program in C to simulate the Worst-Fit contiguous memory allocation technique.

Algorithm:

1. Start the program.
2. Prompt the user to input the size of memory and the number of memory blocks.
3. Prompt the user to input the size of each memory block.
4. Prompt the user to input the number of processes.
5. For each process:
   (a) Prompt the user to input the size of the process.
   (b) Allocate the process to the worst fitting block of memory (largest block that can accommodate the process).
   (c) Update the remaining size of the memory block after allocation.
6. Display the allocation of memory blocks to each process.
7. End the program.

Procedure:

1. Declare variables for memory size, number of memory blocks, memory block sizes, number of processes, and process sizes.
2. Input the size of memory and the number of memory blocks from the user.
3. Input the size of each memory block from the user.
4. Input the number of processes and the size of each process from the user.
5. Allocate each process to the worst fitting block of memory and update the remaining size of the memory block after allocation.
6. Display the allocation of memory blocks to each process.

Code

```c
#include <stdio.h>
int main() {
    int memory_size, num_blocks, num_processes;
    int block_size[10], process_size[10], allocation[10];
    printf("Enter the size of memory: ");
    scanf("%d", &memory_size);
    printf("Enter the number of memory blocks: ");
    scanf("%d", &num_blocks);
    for (int i = 0; i < num_blocks; i++) {
        printf("Enter the size of memory block %d: ", i + 1);
```

```
        scanf("%d", &block_size[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    for (int i = 0; i < num_processes; i++) {
        printf("Enter the size of process %d: ", i + 1);
        scanf("%d", &process_size[i]);
        allocation[i] = -1;
    }
    for (int i = 0; i < num_processes; i++) {
        int worst_index = -1;
        for (int j = 0; j < num_blocks; j++) {
            if (block_size[j] >= process_size[i]) {
                if (worst_index == -1 || block_size[j] >
                   block_size[worst_index]) {
                    worst_index = j;
                }
            }
        }
        if (worst_index != -1) {
            allocation[i] = worst_index;
            block_size[worst_index] -= process_size[i];
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < num_processes; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\n", i + 1, process_size[i],
               allocation[i] + 1);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i + 1,
               process_size[i]);
        }
    }
    return 0;
}
```

Output

```
Enter the size of memory: 1000
Enter the number of memory blocks: 5
Enter the size of memory block 1: 200
Enter the size of memory block 2: 300
Enter the size of memory block 3: 100
Enter the size of memory block 4: 150
Enter the size of memory block 5: 250
Enter the number of processes: 4
Enter the size of process 1: 120
Enter the size of process 2: 200
Enter the size of process 3: 350
```

```
Enter the size of process 4: 70
```

Process No.	Process Size	Block No.
1	120	2
2	200	5
3	350	Not Allocated
4	70	1

Explanation

The Worst Fit memory allocation algorithm assigns a process to the largest available block, leaving bigger free spaces for future allocations. While this approach reduces the chances of small fragmented blocks, it can lead to internal fragmentation due to excessive unused space. Since it always picks the largest block, it requires scanning all blocks, making it slower than First Fit. Worst Fit is useful when large processes are expected later, but it can be inefficient in memory utilization.

8.Write a program to stimulate Page Replacement Algorithms

8 a. First-In-First-Out (FCFS)

AIM: To write a program in C to simulate the FCFS Page Replacement Algorithm.
Algorithm:

1. Start the program.
2. Prompt the user to enter the number of pages and the frame capacity.
3. Maintain a queue to store pages.
4. For each page request:
   (a) If the page is already in memory, count it as a hit.
   (b) If memory is full, remove the oldest page (first in queue).
   (c) Insert the new page at the end of the queue.
   (d) Count page faults and misses.
5. Display the total page faults, hits, and misses.
6. End the program.

Procedure:

1. Declare variables for the number of pages, frame capacity, page faults, hits, and misses.
2. Input the page reference string and frame capacity.
3. Process each page request using the FIFO method.
4. Track the pages in memory and replace them as per the FIFO policy.
5. Print the total number of page faults, hits, and misses.

Code

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

void FCFS(int pages[], int n, int capacity) {
    int queue[capacity], front = 0, count = 0, page_faults = 0,
       hits = 0;
    bool in_memory[100] = {false}; // Track if a page is in
       memory

    for (int i = 0; i < n; i++) {
        if (in_memory[pages[i]]) { // Page hit
```

```
            hits++;
        } else { // Page fault
            if (count < capacity) {
                queue[count++] = pages[i];
            } else {
                in_memory[queue[front]] = false; // Remove oldest
                    page
                queue[front] = pages[i];
                front = (front + 1) % capacity;
            }
            in_memory[pages[i]] = true;
            page_faults++;
        }
    }
    int misses = page_faults;
    printf("Total Page Faults: %d\n", page_faults);
    printf("Total Hits: %d\n", hits);
    printf("Total Misses: %d\n", misses);
}

int main() {
    int pages[] = {1, 3, 0, 3, 5, 6, 3, 5, 1, 0};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 3;

    FCFS(pages, n, capacity);
    return 0;
}
```

Output

```
Total Page Faults: 6
Total Hits: 4
Total Misses: 6
```

Explanation

The FCFS Page Replacement Algorithm replaces the oldest page in memory when a new page needs to be loaded. It follows a queue-based approach, where the first inserted page is the first to be removed. One major disadvantage is Belady's Anomaly, which means increasing the number of frames may increase page faults instead of reducing them. Although simple, FCFS is not optimal for performance, as it does not consider future page requests.

8 b. Least Recently Used (LRU)

AIM: To write a program in C to simulate the LRU Page Replacement Algorithm.
Algorithm:

1. Start the program.
2. Input the number of pages and frame capacity.
3. Maintain an array to store page access times.
4. For each page request:
   (a) If the page is already in memory, update its usage time and count it as a hit.
   (b) If memory is full, replace the least recently used page.
   (c) Insert the new page and update access times.
   (d) Count page faults and misses.
5. Display the total page faults, hits, and misses.
6. End the program.

Procedure:

1. Declare variables for page requests, frame size, and an array for tracking recent usage.
2. Input the total number of pages and frame capacity.
3. For each page request, check if it is already in memory.
4. If the page is not found:
   - If the memory is full, remove the least recently used page.
   - Insert the new page and update the tracking array.
5. Print the total number of page faults, hits, and misses.

Code

```c
#include <stdio.h>

void LRU(int pages[], int n, int capacity) {
    int frames[capacity], index[capacity], count = 0, page_faults
        = 0, hits = 0;

    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < count; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                index[j] = i;
```

```
                hits++;
                break;
            }
        }

        if (!found) {
            int replace = 0;
            if (count < capacity)
                replace = count++;
            else {
                int min = index[0], min_index = 0;
                for (int j = 1; j < capacity; j++)
                    if (index[j] < min) {
                        min = index[j];
                        min_index = j;
                    }
                replace = min_index;
            }
            frames[replace] = pages[i];
            index[replace] = i;
            page_faults++;
        }
    }
    int misses = page_faults;
    printf("Total Page Faults: %d\n", page_faults);
    printf("Total Hits: %d\n", hits);
    printf("Total Misses: %d\n", misses);
}

int main() {
    int pages[] = {2, 3, 1, 3, 5, 1, 2, 4};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 3;

    LRU(pages, n, capacity);
    return 0;
}
```

Expected Output

```
Total Page Faults: 6
Total Hits: 2
Total Misses: 6
```

Explanation

The Least Recently Used (LRU) Page Replacement Algorithm replaces the page that has not been used for the longest time in the past. It uses a tracking array to determine the least recently used page when a new page needs to be loaded into memory. When a page request is made, if the page is already in memory, it is counted as a hit, and its last

usage index is updated. If the page is not in memory, it results in a miss, and if there is space available, it is simply added. If memory is full, the page that was least recently used (determined using the stored indices) is replaced. The LRU algorithm is widely used in modern operating systems because it offers better performance than FIFO by making intelligent replacement decisions based on past usage patterns. However, it requires additional storage to track the usage history of pages, which makes its implementation more complex than simpler algorithms like FIFO.

8 c. Optimal Page Replacement

AIM: To write a program in C to simulate the Optimal Page Replacement Algorithm.
Algorithm:

1. Start the program.
2. Input the number of pages and frame capacity.
3. Maintain an array to store future page requests.
4. For each page request:
   (a) If the page is already in memory, count it as a hit.
   (b) If memory is full, replace the page that will not be used for the longest future time.
   (c) Insert the new page and count page faults and misses.
5. Display the total page faults, hits, and misses.
6. End the program.

Procedure:

1. Declare arrays for tracking pages and their future use.
2. Input the number of pages and frame capacity.
3. For each page request, check if it is already in memory.
4. If the page is not found:
   - If memory is full, replace the page that will not be used for the longest future time.
   - Insert the new page.
5. Print the total number of page faults, hits, and misses.

Code

```c
#include <stdio.h>

int predict(int pages[], int n, int frame[], int frameSize, int index) {
    int farthest = index, replaceIndex = -1;
    for (int i = 0; i < frameSize; i++) {
        int j;
        for (j = index; j < n; j++) {
            if (frame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    replaceIndex = i;
                }
```

```
                break;
            }
        }
        if (j == n)
            return i;
    }
    return (replaceIndex == -1) ? 0 : replaceIndex;
}

void optimalPageReplacement(int pages[], int n, int frameSize) {
    int frame[frameSize], count = 0, page_faults = 0, hits = 0,
       misses = 0;

    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < count; j++)
            if (frame[j] == pages[i]) {
                found = 1;
                hits++;
                break;
            }

        if (!found) {
            if (count < frameSize)
                frame[count++] = pages[i];
            else {
                int replaceIndex = predict(pages, n, frame,
                   frameSize, i + 1);
                frame[replaceIndex] = pages[i];
            }
            page_faults++;
            misses++;
        }
    }
    printf("Total Page Faults: %d\n", page_faults);
    printf("Total Hits: %d\n", hits);
    printf("Total Misses: %d\n", misses);
}

int main() {
    int pages[] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2};
    int n = sizeof(pages) / sizeof(pages[0]);
    int frameSize = 3;

    optimalPageReplacement(pages, n, frameSize);
    return 0;
}
```

Expected Output

```
Total Page Faults: 5
Total Hits: 4
Total Misses: 5
```

Explanation

The Optimal Page Replacement Algorithm replaces the page that will be used farthest in the future. Unlike LRU or FIFO, it makes perfect page replacement decisions by looking ahead at upcoming page requests. This results in the minimum possible page faults. When a page is found in memory, it is counted as a hit, and no page replacement occurs. When the page is not in memory, it is counted as a miss, and the algorithm replaces the page that will not be used for the longest future time. This method guarantees the minimum number of page faults, but it is impractical in real-world systems since it requires knowledge of future page requests, which is typically unavailable. Nonetheless, it serves as the optimal solution for theoretical scenarios.