silp 1
Q.1 Write a program that demonstrates the use of nice() system call. After a child process is started
using fork(), assign higher priority to the child using nice() system call. [10 marks]

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

int main() {
    pid_t pid;
    int nice_value = 10; // A higher nice value means lower priority

    // Fork a child process
    pid = fork();

    if (pid < 0) {
        // Error occurred
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // This is the child process
        printf("Child Process: PID = %d\n", getpid());

        // Set a higher priority for the child process
        if (nice(-nice_value) == -1) { // Negative value to lower the nice value
            perror("Failed to change nice value in child");
            exit(EXIT_FAILURE);
        }

        printf("Child Process: Nice value changed to lower priority\n");
        // Simulate some workload
        for (int i = 0; i < 5; i++) {
            printf("Child working...\n");
            sleep(1); // Simulate work
        }
        exit(EXIT_SUCCESS);
    } else {
        // This is the parent process
        printf("Parent Process: PID = %d\n", getpid());

        // Wait for the child process to complete
        wait(NULL);
        printf("Parent Process: Child has terminated.\n");
    }

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n=3 as
the number of memory frames.
Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6
Implement FIFO

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

#define MAX_FRAMES 10 // Maximum number of frames

void fifoPageReplacement(int pages[], int n, int frames) {
    int pageFrame[MAX_FRAMES], pageFaults = 0, index = 0;
    int i, j, found;

    // Initialize page frames to -1 (indicating empty)
    for (i = 0; i < frames; i++) {
        pageFrame[i] = -1;
    }

    printf("Reference String: ");
    for (i = 0; i < n; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (i = 0; i < n; i++) {
        found = 0;

        // Check if page is already in any of the frames
        for (j = 0; j < frames; j++) {
            if (pageFrame[j] == pages[i]) {
                found = 1; // Page hit
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFrame[index] = pages[i]; // Replace the page in FIFO order
            index = (index + 1) % frames; // Move to the next frame index
            pageFaults++;

            // Display the current state of the page frames
            printf("Page Fault! Frame State: ");
            for (j = 0; j < frames; j++) {
                if (pageFrame[j] != -1) {
                    printf("%d ", pageFrame[j]);
                } else {
                    printf("- "); // Indicating empty frame
                }
            }
            printf("\n");
        } else {
            // Page hit, no action required
            printf("Page Hit! Frame State: ");
            for (j = 0; j < frames; j++) {
                if (pageFrame[j] != -1) {
                    printf("%d ", pageFrame[j]);
                } else {
                    printf("- "); // Indicating empty frame
                }
            }
            printf("\n");
        }
    }

    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int frames = 3; // Number of memory frames
    int pages[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6}; // Reference
string
    int n = sizeof(pages) / sizeof(pages[0]); // Number of pages in the reference
string

    fifoPageReplacement(pages, n, frames);

    return 0;
}

-----------------------------------------------------------------------------------
-------------------------------
```

slip 2

Q.1 Create a child process using fork(), display parent and child process id. Child process will
display the message "Hello World" and the parent process should display "Hi".
[10 marks]

```
ANS:#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main() {
    pid_t pid;

    // Create a child process
    pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child Process: PID = %d\n", getpid());
        printf("Hello World\n");
    } else {
        // Parent process
        printf("Parent Process: PID = %d\n", getpid());
        printf("Hi\n");
    }

    return 0;
}
```

Q.2 Write the simulation program using SJF (non-preemptive). The arrival time and first CPU
bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units).
The next CPU burst should be generated using random function. The output should give the Gantt
chart, Turnaround Time and Waiting time for each process and average times. [20 marks]

ANS:
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>  // Include the time.h header for time function

#define MAX_PROCESSES 10
#define IO_WAIT_TIME 2

typedef struct {
    int id;
    int arrival_time;
    int cpu_burst;
    int waiting_time;
    int turnaround_time;
} Process;

void generate_next_burst(Process *p) {
    // Simulating next CPU burst with a random value (1 to 10)
    p->cpu_burst = rand() % 10 + 1;
```

```
}

void calculate_times(Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting and turnaround times
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].cpu_burst;
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

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

void sjf_scheduling(Process processes[], int n) {
    int time = 0, completed = 0, current_index;
    bool is_completed[MAX_PROCESSES] = {false};

    printf("\nGantt Chart: ");

    while (completed < n) {
        current_index = -1;
        int min_burst = 9999; // Arbitrarily large value

        // Find the process with the shortest burst time
        for (int i = 0; i < n; i++) {
            if (!is_completed[i] && processes[i].arrival_time <= time) {
                if (processes[i].cpu_burst < min_burst) {
                    min_burst = processes[i].cpu_burst;
                    current_index = i;
                }
            }
        }

        if (current_index != -1) {
            // Update waiting time
            processes[current_index].waiting_time = time -
processes[current_index].arrival_time;
            time += processes[current_index].cpu_burst + IO_WAIT_TIME;
            is_completed[current_index] = true;
            completed++;

            printf("P%d ", processes[current_index].id); // Print process ID in
Gantt chart

            // Generate next CPU burst for the process
            generate_next_burst(&processes[current_index]);
        } else {
            // No process is ready, move time forward
            time++;
        }
    }

    printf("\n\nProcess\tArrival Time\tCPU Burst\tWaiting Time\tTurnaround
Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].arrival_time,
               processes[i].cpu_burst, processes[i].waiting_time,
processes[i].turnaround_time);
    }

    calculate_times(processes, n);
}

int main() {
    Process processes[MAX_PROCESSES];
    int n;

    // Seed for random number generation
    srand(time(NULL));

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

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1; // Process ID starts from 1
        printf("Enter arrival time for process P%d: ", processes[i].id);
        scanf("%d", &processes[i].arrival_time);
        generate_next_burst(&processes[i]); // Initial CPU burst
        processes[i].waiting_time = 0; // Initialize waiting time
    }

    sjf_scheduling(processes, n);

    return 0;
}
```

----

slip 3

Q. 1 Creating a child process using the command exec(). Note down process ids of the parent
and the child processes, check whether the control is given back to the parent after the child
process terminates. [10 marks]

ANS:#include <stdio.h>

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

int main() {
    pid_t pid;

    // Create a child process
    pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child Process: PID = %d\n", getpid());

        // Replace child process with a new program
        char *args[] = {"echo", "Hello from Child Process!", NULL};
        execvp(args[0], args);

        // If exec fails
        perror("Exec failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        printf("Parent Process: PID = %d\n", getpid());

        // Wait for the child process to terminate
        wait(NULL);
        printf("Child process has terminated, control returned to Parent
Process.\n");
    }

    return 0;
}
```

Q.2 Write the simulation program using FCFS. The arrival time and first CPU bursts of different
jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst
should be generated using random function. The output should give the Gantt chart,Turnaround
Time and Waiting time for each process and average times. [20 marks

```
ANS:#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

```
#include <time.h>

#define MAX_PROCESSES 10
#define IO_WAIT_TIME 2

typedef struct {
    int id;
    int arrival_time;
    int cpu_burst;
    int waiting_time;
    int turnaround_time;
} Process;

void generate_next_burst(Process *p) {
    // Simulating next CPU burst with a random value (1 to 10)
    p->cpu_burst = rand() % 10 + 1;
}

void calculate_times(Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting and turnaround times
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].cpu_burst;
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

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

void fcfs_scheduling(Process processes[], int n) {
    int time = 0;

    printf("\nGantt Chart: ");

    for (int i = 0; i < n; i++) {
        if (time < processes[i].arrival_time) {
            time = processes[i].arrival_time; // If the CPU is idle
        }

        processes[i].waiting_time = time - processes[i].arrival_time;
        time += processes[i].cpu_burst + IO_WAIT_TIME; // Add CPU burst and I/O
wait time

        printf("P%d ", processes[i].id); // Print process ID in Gantt chart

        // Generate next CPU burst for the process
        generate_next_burst(&processes[i]);
    }

    printf("\n\nProcess\tArrival Time\tCPU Burst\tWaiting Time\tTurnaround
Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].arrival_time,
               processes[i].cpu_burst, processes[i].waiting_time,
processes[i].turnaround_time);
    }

    calculate_times(processes, n);
}

int main() {
    Process processes[MAX_PROCESSES];
    int n;

    // Seed for random number generation
    srand(time(NULL));

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

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1; // Process ID starts from 1
        printf("Enter arrival time for process P%d: ", processes[i].id);
        scanf("%d", &processes[i].arrival_time);
        generate_next_burst(&processes[i]); // Initial CPU burst
        processes[i].waiting_time = 0; // Initialize waiting time
    }

    fcfs_scheduling(processes, n);

    return 0;
}
```

----

slip 4
Q.1 Write a program to illustrate the concept of orphan process (Using fork() and sleep())
[10 marks]
ANS:

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

int main() {
    pid_t pid;

    // Create a child process
    pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
        sleep(5); // Sleep for 5 seconds
        printf("Child Process: I am still running. My new Parent PID = %d\n",
getppid());
    } else {
        // Parent process
        printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);
        sleep(2); // Sleep for 2 seconds (to allow child to become orphan)
        printf("Parent Process: I am terminating now.\n");
        exit(EXIT_SUCCESS);
    }

    return 0;
}
```

Q.2 Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first
CPU burst and priority for different n number of processes should be input to the algorithm.
Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly.
The output should give Gantt chart, turnaround time and waiting time for each process. Also find
the average waiting time and turnaround time..

ANS:
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_PROCESSES 10
#define IO_WAIT_TIME 2

typedef struct {
    int id;
    int arrival_time;
    int cpu_burst;
    int priority;
    int waiting_time;
    int turnaround_time;
} Process;

void generate_next_burst(Process *p) {
    // Simulating next CPU burst with a random value (1 to 10)
    p->cpu_burst = rand() % 10 + 1;
}

void calculate_times(Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting and turnaround times
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].cpu_burst;
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

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

void non_preemptive_priority_scheduling(Process processes[], int n) {
    int time = 0, completed = 0;
    bool is_completed[MAX_PROCESSES] = {false};

    printf("\nGantt Chart: ");

    while (completed < n) {
        int current_index = -1;
        int highest_priority = -1; // Lower number indicates higher priority

        // Find the process with the highest priority (smallest number)
        for (int i = 0; i < n; i++) {
            if (!is_completed[i] && processes[i].arrival_time <= time) {
                if (processes[i].priority < highest_priority || current_index ==
-1) {
                    highest_priority = processes[i].priority;
                    current_index = i;
                }
            }
        }

        if (current_index != -1) {
            // Update waiting time
            processes[current_index].waiting_time = time -
processes[current_index].arrival_time;
            time += processes[current_index].cpu_burst + IO_WAIT_TIME; // Add CPU
burst and I/O wait time
```

```
            is_completed[current_index] = true;
            completed++;

            printf("P%d ", processes[current_index].id); // Print process ID in
Gantt chart

            // Generate next CPU burst for the process
            generate_next_burst(&processes[current_index]);
        } else {
            // No process is ready, move time forward
            time++;
        }
    }

    printf("\n\nProcess\tArrival Time\tCPU Burst\tPriority\tWaiting
Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].arrival_time,
               processes[i].cpu_burst, processes[i].priority,
processes[i].waiting_time, processes[i].turnaround_time);
    }

    calculate_times(processes, n);
}

int main() {
    Process processes[MAX_PROCESSES];
    int n;

    // Seed for random number generation
    srand(time(NULL));

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

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1; // Process ID starts from 1
        printf("Enter arrival time for process P%d: ", processes[i].id);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter CPU burst for process P%d: ", processes[i].id);
        scanf("%d", &processes[i].cpu_burst);
        printf("Enter priority for process P%d (lower number indicates higher
priority): ", processes[i].id);
        scanf("%d", &processes[i].priority);
        processes[i].waiting_time = 0; // Initialize waiting time
    }

    non_preemptive_priority_scheduling(processes, n);
    return 0;
}
```

----

slip 5
Q.1 Write a program that demonstrates the use of nice () system call. After a child process is
started using fork (), assign higher priority to the child using nice () system call. [10 marks]

```
ANS:#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/wait.h> // Include this header for wait()

int main() {
    pid_t pid;

    // Create a child process
    pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        int nice_value = 10; // Higher nice value means lower priority
        if (nice(nice_value) == -1) {
            perror("Failed to change priority");
            exit(EXIT_FAILURE);
        }
        printf("Child Process: PID = %d, Nice Value = %d\n", getpid(), nice_value);
        // Simulate some work
        sleep(2);
    } else {
        // Parent process
        printf("Parent Process: PID = %d\n", getpid());

        // Wait for the child process to finish
        wait(NULL);
        printf("Parent Process: Child has terminated.\n");
    }

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n as the
number of memory frames. Reference String: 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6
i. Implement FIFO

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

#define MAX_FRAMES 10

void fifo_page_replacement(int reference_string[], int n, int frames) {
    int memory[MAX_FRAMES];
    int page_faults = 0, current_frame = 0;
    int is_page_in_memory;

    // Initialize memory frames to -1 (empty)
    for (int i = 0; i < frames; i++) {
        memory[i] = -1;
    }

    printf("Reference String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", reference_string[i]);
    }
    printf("\n");

    printf("FIFO Page Replacement:\n");

    for (int i = 0; i < n; i++) {
        is_page_in_memory = 0;

        // Check if page is already in memory
        for (int j = 0; j < frames; j++) {
            if (memory[j] == reference_string[i]) {
                is_page_in_memory = 1;
                break;
            }
        }

        // If page is not in memory, replace it
        if (!is_page_in_memory) {
            memory[current_frame] = reference_string[i];
            current_frame = (current_frame + 1) % frames; // Move to the next frame
            page_faults++;
            printf("Page Fault! Added: %d\n", reference_string[i]);
        } else {
            printf("Page Hit: %d\n", reference_string[i]);
        }

        // Print current memory state
        printf("Current Memory State: ");
        for (int j = 0; j < frames; j++) {
            if (memory[j] != -1) {
                printf("%d ", memory[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

    printf("\nTotal Page Faults: %d\n", page_faults);
}

int main() {
    int reference_string[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int n = sizeof(reference_string) / sizeof(reference_string[0]);
    int frames;

    printf("Enter the number of memory frames: ");
    scanf("%d", &frames);
    if (frames > MAX_FRAMES) {
        printf("Maximum frames allowed is %d\n", MAX_FRAMES);
        return 1;
    }

    fifo_page_replacement(reference_string, n, frames);

    return 0;
}
```
----

slip 6

Q.1 Write a program to find the execution time taken for execution of a given set of instructions
(use clock() function) [10 marks]

ANS:
```
#include <stdio.h>
#include <time.h>

void executeInstructions() {
    // Example set of instructions
    int sum = 0;
    for (int i = 0; i < 1000000; i++) {
        sum += i;
    }
    printf("Sum: %d\n", sum);
}

int main() {
    clock_t start, end;
    double cpu_time_used;

    start = clock();  // Start time
    executeInstructions();  // Execute the set of instructions
    end = clock();    // End time

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC; // Calculate time
taken
    printf("Execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n as the
number of memory frames.
Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6
Implement FIFO
ANS:

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

#define MAX_FRAMES 10

void fifoPageReplacement(int pages[], int n, int frames) {
    int pageFrame[MAX_FRAMES], pageFaults = 0;
    int pageIndex = 0;
    int isPageInFrame;

    // Initialize page frames
    for (int i = 0; i < frames; i++) {
        pageFrame[i] = -1; // -1 indicates an empty frame
    }

    printf("Page Frames:\n");

    for (int i = 0; i < n; i++) {
        isPageInFrame = 0;

        // Check if the page is already in one of the frames
        for (int j = 0; j < frames; j++) {
            if (pageFrame[j] == pages[i]) {
                isPageInFrame = 1;
                break;
            }
        }

        // Page fault occurs if the page is not found in any frame
        if (!isPageInFrame) {
            // If there's space in frames, insert the new page
            if (pageIndex < frames) {
                pageFrame[pageIndex++] = pages[i];
            } else {
                // Replace the oldest page (FIFO)
                for (int j = 0; j < frames - 1; j++) {
                    pageFrame[j] = pageFrame[j + 1];
                }
                pageFrame[frames - 1] = pages[i];
            }
            pageFaults++;

            // Print the current page frame status
            for (int j = 0; j < frames; j++) {
                printf("%d ", pageFrame[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", pageFaults);
}

int main() {
    int pages[] = {3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int n = sizeof(pages) / sizeof(pages[0]);
    int frames;

    printf("Enter the number of memory frames: ");
    scanf("%d", &frames);

    if (frames > MAX_FRAMES) {
        printf("Error: Maximum frames allowed is %d\n", MAX_FRAMES);
        return 1;
    }

    fifoPageReplacement(pages, n, frames);

    return 0;
}
```

----

slip 7

.1 Write a program to create a child process using fork().The parent should goto sleep state and

child process should begin its execution. In the child process, use execl() to execute the “ls”
command. [10 marks]

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

int main() {
    pid_t pid = fork(); // Create a child process

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process
        printf("Child process: Executing 'ls'\n");
        execl("/bin/ls", "ls", NULL); // Execute 'ls'

        // If execl returns, it must have failed
        perror("execl failed");
        exit(1);
    } else {
        // Parent process
        printf("Parent process: Going to sleep...\n");
        sleep(5); // Sleep for 5 seconds
        printf("Parent process: Waking up...\n");

        // Wait for the child process to finish
        wait(NULL);
        printf("Parent process: Child process finished.\n");
    }

    return 0;
}
```

Q.2 Write the simulation program using FCFS. The arrival time and first CPU bursts of different
jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU
burst should be generated using random function. The output should give the Gantt chart,
Turnaround Time and Waiting time for each process and average times

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

```
#include <time.h>

#define MAX_PROCESSES 10

typedef struct {
    int id;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnaroundTime;
} Process;

void calculateTimes(Process processes[], int n, int ioTime) {
    int totalTime = 0;

    for (int i = 0; i < n; i++) {
        // If the process arrives after the current time, we need to wait
        if (totalTime < processes[i].arrivalTime) {
            totalTime = processes[i].arrivalTime;
        }

        // Calculate waiting time
        processes[i].waitingTime = totalTime - processes[i].arrivalTime;

        // Calculate turnaround time
        totalTime += processes[i].burstTime + ioTime; // Adding I/O waiting time
        processes[i].turnaroundTime = totalTime - processes[i].arrivalTime;

        // Print Gantt Chart
        printf("| P%d ", processes[i].id);
    }
    printf("|\n");
}

int main() {
    int n, ioTime = 2;
    Process processes[MAX_PROCESSES];

    srand(time(NULL)); // Seed for random number generation

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

    for (int i = 0; i < n; i++) {
        printf("Enter arrival time and first CPU burst time for Process %d: ", i +
1);
        scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);
        processes[i].id = i + 1;

        // Generate the next CPU burst randomly between 1 and 10
        processes[i].burstTime = (rand() % 10) + 1;
    }

    // Sort processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrivalTime > processes[j].arrivalTime) {
                Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    printf("\nGantt Chart:\n");
    calculateTimes(processes, n, ioTime);

    // Calculate averages
    float totalWaitingTime = 0, totalTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

    return 0;
}
```
----

slip 8
Q.1 Write a C program to accept the number of process and resources and find the need matrix
content and display it.
ANS:#include <stdio.h>

```
void calculateNeed(int processes[], int n, int max[][10], int allot[][10], int
need[][10]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < 10; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }
}

void displayMatrix(int matrix[][10], int n, int m, const char* title) {
    printf("%s:\n", title);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int processes[10], n, m;
    int max[10][10], allot[10][10], need[10][10];

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

    printf("Enter the number of resources: ");
    scanf("%d", &m);

    // Input maximum resources needed by each process
    printf("Enter the maximum resource matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("Process P%d: ", i);
        for (int j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    // Input allocated resources for each process
    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("Process P%d: ", i);
        for (int j = 0; j < m; j++) {
            scanf("%d", &allot[i][j]);
        }
    }

    // Calculate the need matrix
    calculateNeed(processes, n, max, allot, need);

    // Display matrices
    displayMatrix(max, n, m, "Maximum Resource Matrix");
    displayMatrix(allot, n, m, "Allocation Matrix");
    displayMatrix(need, n, m, "Need Matrix");

    return 0;
}
```

------OR------

Q.2. Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n =3 as

the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8
Implement OPT

ANS:#include <stdio.h>
```c
#include <stdlib.h>

#define MAX_FRAMES 10
#define MAX_PAGES 20

int findOptimalPage(int pageFrames[], int pages[], int currentIndex, int n, int
frameCount) {
    int furthestIndex = -1, optimalPage = -1;

    for (int i = 0; i < frameCount; i++) {
        int j;
        for (j = currentIndex; j < n; j++) {
            if (pageFrames[i] == pages[j]) {
                if (j > furthestIndex) {
                    furthestIndex = j;
                    optimalPage = i;
                }
                break;
            }
        }
        // If the page is not found in the future references
        if (j == n) {
            return i; // Return this frame index to replace
        }
    }
    return optimalPage; // Return the optimal page to replace
}

void optimalPageReplacement(int pages[], int n, int frameCount) {
    int pageFrames[MAX_FRAMES], pageFaults = 0;

    // Initialize the page frames
    for (int i = 0; i < frameCount; i++) {
        pageFrames[i] = -1; // -1 indicates an empty frame
    }

    printf("Page Reference String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < n; i++) {
        int j;
        int pageFound = 0;
        // Check if the page is already in one of the frames
        for (j = 0; j < frameCount; j++) {
            if (pageFrames[j] == pages[i]) {
                pageFound = 1; // Page hit
                break;
            }
        }

        if (!pageFound) {
            // Page fault occurs
            pageFaults++;

            // If there's an empty frame, add the page
            for (j = 0; j < frameCount; j++) {
                if (pageFrames[j] == -1) {
                    pageFrames[j] = pages[i];
                    pageFound = 1;
                    break;
                }
            }

            // If all frames are full, find the optimal page to replace
            if (!pageFound) {
                int optimalIndex = findOptimalPage(pageFrames, pages, i + 1, n,
frameCount);
                pageFrames[optimalIndex] = pages[i];
            }

            // Print current page frame status
            printf("Page Fault! Frame Status: ");
            for (j = 0; j < frameCount; j++) {
                printf("%d ", pageFrames[j]);
            }
            printf("\n");
        } else {
            // Print frame status if there's no fault
            printf("Page Hit! Frame Status: ");
            for (j = 0; j < frameCount; j++) {
                printf("%d ", pageFrames[j]);
            }
            printf("\n");
        }
    }

    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int pages[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int n = sizeof(pages) / sizeof(pages[0]);
    int frameCount = 3; // Number of frames

    optimalPageReplacement(pages, n, frameCount);

    return 0;
}
```

----

slip 9
Q.1 Write a program to create a child process using fork().The parent should goto sleep state and
child process should begin its execution. In the child process, use execl() to execute the "ls"
command.

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

int main() {
    pid_t pid = fork(); // Create a child process

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process
        printf("Child process: Executing 'ls'\n");
        execl("/bin/ls", "ls", NULL); // Execute 'ls'

        // If execl returns, it must have failed
        perror("execl failed");
        exit(1);
    } else {
        // Parent process
        printf("Parent process: Going to sleep...\n");
        sleep(5); // Sleep for 5 seconds
        printf("Parent process: Waking up...\n");

        // Wait for the child process to finish
        wait(NULL);
        printf("Parent process: Child process finished.\n");
    }
    return 0;
}
```

.2 Write the program to simulate Round Robin (RR) scheduling. The arrival time and first CPU-
burst for different n number of processes should be input to the algorithm. Also give the time
quantum as input. Assume the fixed IO waiting time (2 units). The next CPU-burst should be
generated randomly. The output should give Gantt chart, turnaround time and waiting time for each
process. Also find the average waiting time and turnaround time.

```
ANS:#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_PROCESSES 10

typedef struct {
    int id;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int waitingTime;
    int turnaroundTime;
} Process;

void calculateTimes(Process processes[], int n, int timeQuantum, int ioTime) {
    int time = 0;
    int completed = 0;

    printf("Gantt Chart:\n");

    while (completed < n) {
        int allDone = 1;

        for (int i = 0; i < n; i++) {
            if (processes[i].remainingTime > 0) {
                allDone = 0; // At least one process is not finished

                if (processes[i].arrivalTime <= time) {
                    if (processes[i].remainingTime > timeQuantum) {
                        time += timeQuantum;
                        processes[i].remainingTime -= timeQuantum;
                        printf("P%d ", processes[i].id);
                        time += ioTime; // Add I/O waiting time
                    } else {
                        time += processes[i].remainingTime;
                        processes[i].waitingTime = time - processes[i].burstTime -
processes[i].arrivalTime;
                        processes[i].turnaroundTime = time -
processes[i].arrivalTime;
                        printf("P%d ", processes[i].id);
                        processes[i].remainingTime = 0; // Process finished
                        completed++;
                    }
                }
            }
        }

        if (allDone) break; // All processes are done
    }

    printf("\n");

    // Calculate average waiting and turnaround times
    float totalWaitingTime = 0, totalTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);
}

int main() {
    Process processes[MAX_PROCESSES];
    int n, timeQuantum, ioTime = 2;

    srand(time(NULL)); // Seed for random number generation

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

    for (int i = 0; i < n; i++) {
        printf("Enter arrival time and initial CPU burst time for Process %d: ", i
+ 1);
        scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);
        processes[i].id = i + 1;
        processes[i].remainingTime = processes[i].burstTime; // Initialize
remaining time
    }

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

    // Simulate and calculate times
    calculateTimes(processes, n, timeQuantum, ioTime);
    return 0;
}
```

----

slip10

Q.1 Write a program to illustrate the concept of orphan process (Using fork() and sleep())
[10 marks]

ANS:
```
#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");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process (PID: %d) is terminating.\n", getpid());
        exit(0);  // Parent exits
    } else {
        // Child process
        printf("Child process (PID: %d) is running.\n", getpid());
        sleep(10);  // Sleep for 10 seconds
        printf("Child process (PID: %d) is still running after parent
termination.\n", getpid());
    }

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and
total number of page faults for the following given page reference string. Give input n=3 as the
number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8
Implement OPT
ANS:
```
#include <stdio.h>
#include <stdlib.h>

#define FRAME_COUNT 3
#define REF_STR_LEN 16
```

```
int findOptimalPage(int pages[], int pageFrame[], int frameCount, int currentIndex,
int totalPages) {
    int farthest = -1, indexToReplace = -1;

    for (int i = 0; i < frameCount; i++) {
        int j;
        for (j = currentIndex; j < totalPages; j++) {
            if (pageFrame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    indexToReplace = i;
                }
                break;
            }
        }
        if (j == totalPages) {
            return i;  // Page not found, replace this page
        }
    }
    return (indexToReplace == -1) ? 0 : indexToReplace;  // Replace the farthest
one
}

int main() {
    int pages[REF_STR_LEN] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15,
19, 8};
    int pageFrame[FRAME_COUNT] = {-1, -1, -1};
    int pageFaults = 0;

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            int replaceIndex = findOptimalPage(pages, pageFrame, FRAME_COUNT, i +
1, REF_STR_LEN);
            pageFrame[replaceIndex] = page;  // Replace the page
            printf("Page %d caused a page fault. Replacing frame %d.\n", page,
replaceIndex);
```

```
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----
slip 11

Q.1 Create a child process using fork(), display parent and child process id. Child process will
display the message "Hello World" and the parent process should display "Hi".
[10 marks]

ANS:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child Process (PID: %d): Hello World\n", getpid());
    } else {
        // Parent process
        printf("Parent Process (PID: %d): Hi\n", getpid());
    }

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n as the

number of memory frames.
Reference String: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1
Implement FIFO
ANS:#include <stdio.h>

```c
#include <stdlib.h>

#define FRAME_COUNT 3
#define REF_STR_LEN 12

int main() {
    int pages[REF_STR_LEN] = {0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1};
    int pageFrame[FRAME_COUNT] = {-1, -1, -1};
    int pageFaults = 0;
    int nextFrame = 0; // To track the next frame to replace

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            pageFrame[nextFrame] = page;  // Replace the page
            printf("Page %d caused a page fault. Replacing frame %d.\n", page,
nextFrame);
            nextFrame = (nextFrame + 1) % FRAME_COUNT; // Move to the next frame
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----
slip 12
Q.1 [10] Write a program to illustrate the concept of orphan process (Using fork() and
sleep()) . [10 marks]

```
ANS:#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");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent Process (PID: %d) is terminating.\n", getpid());
        exit(0);  // Parent exits immediately
    } else {
        // Child process
        sleep(2);  // Sleep to ensure parent terminates first
        printf("Child Process (PID: %d) is running after parent termination.\n",
getpid());
        printf("Child Process says: Hello, I am an orphan now!\n");
    }

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string.
Give input n as the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8
Implement OPT

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

#define FRAME_COUNT 3
```

```
#define REF_STR_LEN 16

int findOptimalPage(int pages[], int pageFrame[], int currentIndex, int totalPages)
{
    int farthest = -1;
    int indexToReplace = -1;

    for (int i = 0; i < FRAME_COUNT; i++) {
        int j;
        for (j = currentIndex; j < totalPages; j++) {
            if (pageFrame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    indexToReplace = i;
                }
                break;
            }
        }
        if (j == totalPages) {
            return i;  // If page not found, replace this one
        }
    }
    return (indexToReplace == -1) ? 0 : indexToReplace;  // Replace the farthest
one
}

int main() {
    int pages[REF_STR_LEN] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15,
19, 8};
    int pageFrame[FRAME_COUNT] = {-1, -1, -1};
    int pageFaults = 0;

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                break;
            }
        }
        if (!found) {
            // Page fault occurred
            pageFaults++;
            int replaceIndex = findOptimalPage(pages, pageFrame, i + 1,
REF_STR_LEN);
            pageFrame[replaceIndex] = page;  // Replace the page
            printf("Page %d caused a page fault. Replacing frame %d.\n", page,
replaceIndex);
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----

slip 13
Q.1 Write a program that demonstrates the use of nice() system call. After a child process is
started using fork(), assign higher priority to the child using nice() system call. [10 marks]
ANS:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>

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

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child Process (PID: %d) before nice: %d\n", getpid(), nice(0));
        nice(-10);  // Assign higher priority
        printf("Child Process (PID: %d) after nice: %d\n", getpid(), nice(0));
        // Simulating some work
        for (int i = 0; i < 5; i++) {
            printf("Child Process working...\n");
            sleep(1);
        }
        exit(0);
    } else {
        // Parent process
        printf("Parent Process (PID: %d) before nice: %d\n", getpid(), nice(0));
        // Simulating some work
        for (int i = 0; i < 5; i++) {
            printf("Parent Process working...\n");
            sleep(1);
        }
        exit(0);
    }
}
```

Q.2 Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance. Consider the following snapshot of system, A, B, C and D are the resource type.
a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence.
c) If a request from process P arrives for (0, 4, 2, 0) can it be granted immediately by keeping
the system in safe state. Print a message. [20 marks]

```
ALLOCATION MAX
A B C D A B C D
P0 0 0 1 2 0 0 1 2
P1 1 0 0 0 1 7 5 0
P2 1 3 5 4 2 3 5 6
P3 0 6 3 2 0 6 5 2
P4 0 0 1 4 0 6 5 6
AVAILABLE
A B C D
1 5 2 0
```

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

#define P 4  // Number of processes
#define R 4  // Number of resources

int allocation[P][R] = {
    {0, 0, 1, 2},
    {1, 0, 0, 0},
    {1, 3, 5, 4},
    {0, 6, 3, 2}
};

int max[P][R] = {
    {0, 0, 1, 2},
    {1, 7, 5, 0},
    {2, 3, 5, 6},
    {0, 6, 5, 2}
};

int need[P][R];  // Need matrix
int available[R] = {1, 5, 2, 0};  // Available resources
int finish[P] = {0};  // Finish array
int safeSeq[P];  // Safe sequence

// Function to calculate need matrix
void calculateNeed() {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}

// Function to check if the system is in a safe state
bool isSafe() {
    int work[R];
    for (int i = 0; i < R; i++)
        work[i] = available[i];

    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (!finish[p]) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0; k < R; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return false;
        }
    }
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
```

```
        printf("%d ", safeSeq[i]);
    printf("\n");
    return true;
}

// Function to check request
bool requestResources(int process[], int request[]) {
    for (int i = 0; i < R; i++) {
        if (request[i] > need[process[0]][i]) {
            printf("Error: Process has exceeded its maximum claim.\n");
            return false;
        }
    }

    for (int i = 0; i < R; i++) {
        if (request[i] > available[i]) {
            printf("Process must wait for resources.\n");
            return false;
        }
    }

    for (int i = 0; i < R; i++) {
        available[i] -= request[i];
        allocation[process[0]][i] += request[i];
        need[process[0]][i] -= request[i];
    }

    if (isSafe()) {
        return true;
    } else {
        for (int i = 0; i < R; i++) {
            available[i] += request[i];
            allocation[process[0]][i] -= request[i];
            need[process[0]][i] += request[i];
        }
        printf("Request cannot be granted immediately as it would lead to unsafe 
state.\n");
        return false;
    }
}

int main() {
    calculateNeed();
    printf("Need Matrix:\n");
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++)
            printf("%d ", need[i][j]);
        printf("\n");
    }
    isSafe();

    int request[4] = {0, 4, 2, 0};  // Example request from process P1
    int process[1] = {1};  // Process P1

    if (requestResources(process, request)) {
        printf("Request can be granted immediately.\n");
    } else {
        printf("Request cannot be granted immediately.\n");
    }

    return 0;
}
```
----

slip 14
Q.1 Write a program to find the execution time taken for execution of a given set of instructions
(use clock() function) [10 marks]
ANS:#include <stdio.h>
```
#include <time.h>

void performTask() {
    // Simulate some work
    for (volatile int i = 0; i < 100000000; i++);
}

int main() {
    clock_t start, end;
    double cpu_time_used;

    start = clock();  // Start the clock

    performTask();  // Function whose execution time we want to measure

    end = clock();  // End the clock
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;  // Calculate time in
seconds

    printf("Execution time: %f seconds\n", cpu_time_used);
    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n =3 as
the number of memory frames.
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Implement FIFO

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

#define FRAME_COUNT 3
#define REF_STR_LEN 12

int main() {
    int pages[REF_STR_LEN] = {0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1};
    int pageFrame[FRAME_COUNT] = {-1, -1, -1};
    int pageFaults = 0;
    int pageIndex = 0;  // To keep track of the oldest page for FIFO

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            printf("Page %d caused a page fault. Replacing frame %d (old page
%d).\n", page, pageIndex, pageFrame[pageIndex]);

            // Replace the oldest page in FIFO
            pageFrame[pageIndex] = page;  // Insert the new page
            pageIndex = (pageIndex + 1) % FRAME_COUNT;  // Update index for FIFO
replacement
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----
slip 15
Q.1 Write a program to create a child process using fork().The parent should goto sleep state and
child process should begin its execution. In the child process, use execl() to execute the “ls”
command.
[10 marks]
ANS:#include <stdio.h>

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

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

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child Process (PID: %d) is executing 'ls'.\n", getpid());
        execl("/bin/ls", "ls", NULL);  // Execute the ls command
        // If execl() fails
        perror("execl failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        printf("Parent Process (PID: %d) going to sleep.\n", getpid());
        sleep(5);  // Sleep for 5 seconds
        printf("Parent Process (PID: %d) woke up.\n", getpid());
        exit(0);
    }
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n as the
number of memory frames.
Reference String :7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2
Implement LRU

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

#define FRAME_COUNT 3
#define REF_STR_LEN 13  // Updated to reflect the correct number of pages

int main() {
    int pages[REF_STR_LEN] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int pageFrame[FRAME_COUNT] = {-1, -1, -1};  // Initialize frames to -1
(indicating empty)
    int pageFaults = 0;
    int lastUsed[FRAME_COUNT] = {0}; // Tracks last used time for LRU
    int time = 0; // Time counter

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                lastUsed[j] = time; // Update last used time
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            printf("Page %d caused a page fault.\n", page);

            // Find the LRU page to replace
            int lruIndex = 0;
            for (int j = 1; j < FRAME_COUNT; j++) {
                if (lastUsed[j] < lastUsed[lruIndex]) {
                    lruIndex = j;
                }
            }

            // Replace the least recently used page
            printf("Replacing frame %d (old page %d) with page %d.\n", lruIndex,
pageFrame[lruIndex], page);
            pageFrame[lruIndex] = page;  // Insert the new page
            lastUsed[lruIndex] = time;    // Update last used time for the new page
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");

        time++; // Increment time for the next page reference
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----

slip 16
Q.1 Write a program to find the execution time taken for execution of a given set of instructions
(use clock()
function) [10 marks]
ANS:#include <stdio.h>

```
#include <time.h>

void performTask() {
    // Simulate some work with a busy loop
    for (volatile long i = 0; i < 100000000; i++);
}

int main() {
    clock_t start, end;
    double cpu_time_used;

    start = clock();  // Start the clock

    performTask();  // Function whose execution time we want to measure

    end = clock();  // End the clock
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;  // Calculate time in
seconds

    printf("Execution time: %f seconds\n", cpu_time_used);
    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page

scheduling
and total number of page faults for the following given page reference string. Give input n =3 as
the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8
Implement OPT
ANS:#include <stdio.h>

```
#include <stdlib.h>

#define FRAME_COUNT 3
#define REF_STR_LEN 16

int findOptimalPage(int pages[], int currentIndex, int frames[], int n) {
    int farthest = -1;
    int pageToReplace = -1;

    for (int i = 0; i < n; i++) {
        int j;
        for (j = currentIndex; j < REF_STR_LEN; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    pageToReplace = i; // Index of the page to replace
                }
                break;
            }
        }
        // If the page is not found in the future reference string
        if (j == REF_STR_LEN) {
            return i; // Replace this page immediately
        }
    }
    return pageToReplace;
}

int main() {
    int pages[REF_STR_LEN] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15,
19, 8};
    int frames[FRAME_COUNT] = {-1, -1, -1};  // Page frames initialized to -1
    int pageFaults = 0;

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;
        // Check if the page is already in the frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            printf("Page %d caused a page fault.\n", page);

            // Find which page to replace
            int pageToReplace = findOptimalPage(pages, i + 1, frames, FRAME_COUNT);
            printf("Replacing page %d with page %d.\n", frames[pageToReplace],
page);
            frames[pageToReplace] = page;  // Replace the page
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----
slip 17
.1 Write the program to calculate minimum number of resources needed to avoid deadlock. [10 marks]
ANS:#include <stdio.h>

```
int main() {
    int totalProcesses, totalResources;

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

    printf("Enter the number of resources: ");
    scanf("%d", &totalResources);
    int minResourcesNeeded = totalProcesses + totalResources - 1;

    printf("Minimum number of resources needed to avoid deadlock: %d\n",
minResourcesNeeded);
    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n=3 as
the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8
Implement OPT
ANS:

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

#define FRAME_COUNT 3
#define REF_STR_LEN 16

int findOptimalPage(int pages[], int currentIndex, int frames[], int n) {
    int farthest = -1;
    int pageToReplace = -1;

    for (int i = 0; i < n; i++) {
        int j;
        for (j = currentIndex; j < REF_STR_LEN; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    pageToReplace = i; // Index of the page to replace
                }
                break;
            }
        }
        // If the page is not found in the future reference string
        if (j == REF_STR_LEN) {
            return i; // Replace this page immediately
        }
    }
    return pageToReplace;
}

int main() {
    int pages[REF_STR_LEN] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15,
19, 8};
    int frames[FRAME_COUNT] = {-1, -1, -1};  // Page frames initialized to -1
    int pageFaults = 0;
    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in the frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            printf("Page %d caused a page fault.\n", page);

            // Find which page to replace
            int pageToReplace = findOptimalPage(pages, i + 1, frames, FRAME_COUNT);
            printf("Replacing page %d with page %d.\n", frames[pageToReplace],
page);
            frames[pageToReplace] = page;  // Replace the page
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----

slip 18
Q. 1 Write a C program to accept the number of process and resources and find the need matrix
content and display it. [10 marks]

```
ANS:#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int main() {
    int numProcesses, numResources;
    int allocation[MAX_PROCESSES][MAX_RESOURCES];
    int max[MAX_PROCESSES][MAX_RESOURCES];
    int need[MAX_PROCESSES][MAX_RESOURCES];

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

    // Input Allocation Matrix
    printf("Enter the Allocation Matrix:\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < numResources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    // Input Max Matrix
    printf("Enter the Max Matrix:\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < numResources; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    // Calculate Need Matrix
    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numResources; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    // Display Need Matrix
    printf("Need Matrix:\n");
    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numResources; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page scheduling
and total number of page faults for the following given page reference string. Give input n as
the number of memory frames.
Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8
Implement OPT
ANS:#include <stdio.h>

```
#include <stdlib.h>

#define FRAME_COUNT 3
#define REF_STR_LEN 16

int findOptimalPage(int pages[], int currentIndex, int frames[], int n) {
    int farthest = -1;
    int pageToReplace = -1;

    for (int i = 0; i < n; i++) {
        int j;
        for (j = currentIndex; j < REF_STR_LEN; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    pageToReplace = i; // Index of the page to replace
                }
                break;
            }
        }
        // If the page is not found in the future reference string
        if (j == REF_STR_LEN) {
            return i; // Replace this page immediately
        }
    }
    return pageToReplace;
}

int main() {
    int pages[REF_STR_LEN] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15,
19, 8};
    int frames[FRAME_COUNT] = {-1, -1, -1};  // Page frames initialized to -1
    int pageFaults = 0;

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");
    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in the frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            printf("Page %d caused a page fault.\n", page);

            // Find which page to replace
            int pageToReplace = findOptimalPage(pages, i + 1, frames, FRAME_COUNT);
            printf("Replacing page %d with page %d.\n", frames[pageToReplace],
page);
            frames[pageToReplace] = page;  // Replace the page
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```

----
slip 19
Q.1 Write a program to create a child process using fork().The parent should goto sleep state and
child process should begin its execution. In the child process, use execl() to execute the “ls”
command. [10 marks]
ANS:#include <stdio.h>
```
#include <stdlib.h>
#include <unistd.h>
```

```c
int main() {
    pid_t pid = fork();  // Create a child process

    if (pid < 0) {
        // Error occurred
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID: %d) executing 'ls'\n", getpid());
        execl("/bin/ls", "ls", NULL);  // Execute the 'ls' command
        perror("execl failed");  // execl only returns on error
        return 1;
    } else {
        // Parent process
        printf("Parent process (PID: %d) going to sleep\n", getpid());
        sleep(5);  // Sleep for 5 seconds
        printf("Parent process (PID: %d) awake\n", getpid());
    }

    return 0;
}
```

Q.2 Write the program to simulate Non-preemptive Priority scheduling. The arrival time and
first CPU burst and priority for different n number of processes should be input to the algorithm.
Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly.
The output should give Gantt chart, turnaround time and waiting time for each process. Also find
the average waiting time and turnaround time
ANS:

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

typedef struct {
    int pid;
    int arrivalTime;
    int burstTime;
    int priority;
    int waitingTime;
    int turnaroundTime;
} Process;

void calculateTimes(Process proc[], int n) {
    int totalWaitingTime = 0, totalTurnaroundTime = 0;

    // Calculate waiting time and turnaround time
    for (int i = 0; i < n; i++) {
        proc[i].turnaroundTime = proc[i].burstTime + proc[i].waitingTime;
        totalWaitingTime += proc[i].waitingTime;
        totalTurnaroundTime += proc[i].turnaroundTime;
    }

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

int main() {
    srand(time(0));  // Seed for random number generation
    int n;

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

    Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter arrival time, burst time and priority for Process %d: ", i +
1);
        scanf("%d %d %d", &proc[i].arrivalTime, &proc[i].burstTime,
&proc[i].priority);
        // Simulate next CPU burst
        proc[i].burstTime += (rand() % 5); // Random burst time addition
        proc[i].waitingTime = 0;  // Initialize waiting time
    }

    // Sort processes by arrival time and priority
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].arrivalTime > proc[j].arrivalTime ||
                (proc[i].arrivalTime == proc[j].arrivalTime && proc[i].priority >
proc[j].priority)) {
                Process temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }

    // Calculate waiting times
    for (int i = 0; i < n; i++) {
        if (i > 0) {
            proc[i].waitingTime = proc[i - 1].waitingTime + proc[i - 1].burstTime +
2; // IO wait time of 2 units
        }
    }
    // Print Gantt chart
    printf("\nGantt Chart:\n| ");
    for (int i = 0; i < n; i++) {
        printf("P%d | ", proc[i].pid);
    }
    printf("\n");

    // Calculate average waiting and turnaround times
    calculateTimes(proc, n);

    return 0;
}
```

----
slip 20
Q.1 Write a program to create a child process using fork().The parent should goto sleep state and
child process should begin its execution. In the child process, use execl() to execute the “ls”
command. [10 marks]
ANS:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();  // Create a child process

    if (pid < 0) {
        // Error occurred
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID: %d) executing 'ls'\n", getpid());
        execl("/bin/ls", "ls", NULL);  // Execute the 'ls' command
        perror("execl failed");  // execl only returns on error
        return 1;
    } else {
        // Parent process
        printf("Parent process (PID: %d) going to sleep\n", getpid());
        sleep(5);  // Sleep for 5 seconds
        printf("Parent process (PID: %d) awake\n", getpid());
    }

    return 0;
}
```

Q.2 Write the simulation program to implement demand paging and show the page
scheduling

and total number of page faults for the following given page reference string. Give input n=3 as
the number of memory frames.
Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2
i. Implement LRU
ANS:#include <stdio.h>

```
#include <stdlib.h>

#define FRAME_COUNT 3
#define REF_STR_LEN 13

int main() {
    int pages[REF_STR_LEN] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int pageFrame[FRAME_COUNT] = {-1, -1, -1}; // Initialize frames to -1
    int pageFaults = 0;
    int lastUsed[FRAME_COUNT] = {0}; // Tracks last used time for LRU
    int time = 0; // Time counter

    printf("Reference String: ");
    for (int i = 0; i < REF_STR_LEN; i++) {
        printf("%d ", pages[i]);
    }
    printf("\n");

    for (int i = 0; i < REF_STR_LEN; i++) {
        int page = pages[i];
        int found = 0;

        // Check if the page is already in a frame
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                lastUsed[j] = time; // Update last used time
                break;
            }
        }

        if (!found) {
            // Page fault occurred
            pageFaults++;
            printf("Page %d caused a page fault.\n", page);

            // Find the LRU page to replace
            int lruIndex = 0;
            for (int j = 1; j < FRAME_COUNT; j++) {
                if (lastUsed[j] < lastUsed[lruIndex]) {
                    lruIndex = j;
                }
            }
            // Replace the least recently used page
            printf("Replacing frame %d (old page %d) with page %d.\n", lruIndex,
pageFrame[lruIndex], page);
            pageFrame[lruIndex] = page;  // Insert the new page
            lastUsed[lruIndex] = time;    // Update last used time for the new page
        } else {
            printf("Page %d hit in frame.\n", page);
        }

        // Print current page frames
        printf("Current page frames: ");
        for (int j = 0; j < FRAME_COUNT; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");

        time++; // Increment time for the next page reference
    }

    printf("Total number of page faults: %d\n", pageFaults);
    return 0;
}
```