Experiment No 1. Create a child process in Linux using the fork system call.

Code:

```c
#include <stdio.h>
 #include <sys/types.h>
 #include <unistd.h>
 int main()
 {
 // fork() Create a child process
 int pid = fork();
 if (pid > 0) {
 printf("I am Parent process\n");
 printf("ID : %d\n\n", getpid());
 }
 else if (pid == 0) {
 printf("I am Child process\n");
 // getpid() will return process id of child process
 printf("ID: %d\n", getpid());
 }
 else {
 printf("Failed to create child process");
 }
 return 0;
}
```

Experiment No 2. Write shell scripts to Display OS version, release number, kernel version.

Code:

Display OS Version and Release Number:

```bash
#!/bin/bash
# Script to display OS version and release number

os_version=$(lsb_release -d | awk -F'\t' '{print $2}')
release_number=$(lsb_release -r | awk -F'\t' '{print $2}')

echo "Operating System: $os_version"
echo "Release Number: $release_number"
```

Display Kernel Version :

```bash
#!/bin/bash
# Script to display Linux kernel version

kernel_version=$(uname -r)

echo "Kernel Version: $kernel_version"
```

Experiment No 3. Write shell scripts to Display top 10 processes in descending order
Display processes with highest memory usage

Code:

```
#!/bin/bash
# Script to display top 10 processes by CPU usage

echo "Top 10 processes by CPU usage:"
ps -eo pid,ppid,cmd,%cpu --sort=-%cpu | head -n 11

echo -e "\nProcesses with highest memory usage:"
ps -eo pid,ppid,cmd,%mem --sort=-%mem | head -n 11
```

Experinment no 4. To study and implement disk scheduling algorithm FCFS

Code:

```
// FCFS Scheduling Algorithm in C

#include <stdio.h>

// Structure to store process details
struct Process {
    int pid; // Process ID
    int burst_time; // Burst time
    int waiting_time; // Waiting time
    int turnaround_time; // Turnaround time
};

// Function to print the table
void print_table(struct Process p[], int n) {
    printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].burst_time, p[i].waiting_time,
p[i].turnaround_time);
    }
}

// Function to print the Gantt chart
void print_gantt_chart(struct Process p[], int n) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("P%d ", p[i].pid);
    }
    printf("\n");
}

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

    struct Process p[n]; // Array to store process details

    // Input burst times for each process
    printf("Enter burst times for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i;
        printf("Process %d burst time: ", i);
        scanf("%d", &p[i].burst_time);
    }

    // Calculate waiting time and turnaround time
    p[0].waiting_time = 0;
    p[0].turnaround_time = p[0].burst_time;
    for (int i = 1; i < n; i++) {
        p[i].waiting_time = p[i - 1].waiting_time + p[i - 1].burst_time;
        p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
    }

    // Calculate average waiting time and average turnaround time
    float avg_waiting_time = 0;
    float avg_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += p[i].waiting_time;
        avg_turnaround_time += p[i].turnaround_time;
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    // Print the table and Gantt chart
    print_table(p, n);
    print_gantt_chart(p, n);

    printf("\nAverage waiting time = %.2f\n", avg_waiting_time);
    printf("Average turnaround time = %.2f\n", avg_turnaround_time);

    return 0;
}
```

Exp No 5. To study and implement page replacement policy FIFO.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 10
// Function to find the index of the oldest page in the frame
int findOldestPage(int frames[], int n) {
    int oldestIndex = 0;
    for (int i = 1; i < n; i++) {
        if (frames[i] < frames[oldestIndex]) {
            oldestIndex = i;
        }
    }
    return oldestIndex;
}
// Function to simulate FIFO page replacement
void fifoPageReplacement(int pages[], int n, int capacity) {
    int frames[MAX_FRAMES] = {0}; // Initialize frames with zeros
    int page_faults = 0;
    printf("Page Replacement Table:\n");
    printf("Page\tFrames\n");
    for (int i = 0; i < n; i++) {
        int currentPage = pages[i];
        // Check if the page is already in memory
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frames[j] == currentPage) {
                found = 1;
                break;
            }
        }
        if (!found) {
            // Page fault: Replace the oldest page
            int oldestIndex = findOldestPage(frames, capacity);
            frames[oldestIndex] = currentPage;
            page_faults++;
        }
        // Print the current state of frames
        printf("%d\t", currentPage);
        for (int j = 0; j < capacity; j++) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }
    printf("\nTotal page faults using FIFO: %d\n", page_faults);
}
int main() {
```

```
    int n; // Number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int pages[MAX_FRAMES];
    printf("Enter the page reference string:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }
    int capacity;
    printf("Enter the number of page frames: ");
    scanf("%d", &capacity);
    fifoPageReplacement(pages, n, capacity);
    return 0;
}
```

Exp No 6. To study and implement memory allocation strategy Worst fit.

Code:

```
#include <stdio.h>

#define MAX_PARTITIONS 10

// Structure to store partition details
struct Partition {
    int id; // Partition ID
    int size; // Size of the partition
    int allocated; // Flag to check if partition is allocated
};

// Function to find the worst fit partition
int findWorstFit(struct Partition partitions[], int n, int processSize) {
    int worstIndex = -1;
    int worstSize = -1;

    for (int i = 0; i < n; i++) {
        if (!partitions[i].allocated && partitions[i].size >= processSize) {
            if (worstIndex == -1 || partitions[i].size > worstSize) {
                worstIndex = i;
                worstSize = partitions[i].size;
            }
        }
    }

    return worstIndex;
}

// Function to allocate memory using worst fit
void worstFit(struct Partition partitions[], int n, int processSize) {
    int index = findWorstFit(partitions, n, processSize);
```

```
    if (index != -1) {
        partitions[index].allocated = 1;
        printf("Process allocated to Partition %d\n", partitions[index].id);
    } else {
        printf("Process cannot be allocated due to insufficient memory.\n");
    }
}

int main() {
    int n; // Number of partitions
    printf("Enter the number of partitions: ");
    scanf("%d", &n);

    struct Partition partitions[MAX_PARTITIONS];

    // Input partition sizes
    for (int i = 0; i < n; i++) {
        partitions[i].id = i + 1;
        printf("Enter size of Partition %d: ", partitions[i].id);
        scanf("%d", &partitions[i].size);
        partitions[i].allocated = 0;
    }

    int processSize;
    printf("Enter size of the process: ");
    scanf("%d", &processSize);

    worstFit(partitions, n, processSize);

    // Display partition table
    printf("\nPartition Table:\n");
    printf("Partition ID\tSize\tAllocated\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t%s\n", partitions[i].id, partitions[i].size,
               partitions[i].allocated ? "Yes" : "No");
    }

    return 0;
}
```

Exp No 7. To study and implement the process scheduling algorithm SJF.

Code:

```
#include <stdio.h>
// Structure to store process details
struct Process {
    int pid; // Process ID
    int burst_time; // Burst time
```

```
    int waiting_time; // Waiting time
    int turnaround_time; // Turnaround time
};

// Function to sort processes based on burst time (SJF)
void sort_processes(struct Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].burst_time > p[j + 1].burst_time) {
                // Swap processes
                struct Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

// Function to calculate waiting time and turnaround time
void calculate_times(struct Process p[], int n) {
    p[0].waiting_time = 0;
    p[0].turnaround_time = p[0].burst_time;
    for (int i = 1; i < n; i++) {
        p[i].waiting_time = p[i - 1].waiting_time + p[i - 1].burst_time;
        p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
    }
}

// Function to print the table
void print_table(struct Process p[], int n) {
    printf("PID\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].burst_time, p[i].waiting_time,
p[i].turnaround_time);
    }
}

// Function to print the Gantt chart
void print_gantt_chart(struct Process p[], int n) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < n; i++) {
        printf("P%d ", p[i].pid);
    }
    printf("\n");
}

int main() {
    int n; // Number of processes
    printf("Enter the number of processes: ");
```

```
    scanf("%d", &n);

    struct Process p[n]; // Array to store process details

    // Input burst times for each process
    printf("Enter burst times for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i;
        printf("Process %d burst time: ", i);
        scanf("%d", &p[i].burst_time);
    }

    // Sort processes based on burst time (SJF)
    sort_processes(p, n);

    // Calculate waiting time and turnaround time
    calculate_times(p, n);

    // Calculate average waiting time and average turnaround time
    float avg_waiting_time = 0;
    float avg_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += p[i].waiting_time;
        avg_turnaround_time += p[i].turnaround_time;
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    // Print the table and Gantt chart
    print_table(p, n);
    print_gantt_chart(p, n);

    printf("\nAverage waiting time = %.2f\n", avg_waiting_time);
    printf("Average turnaround time = %.2f\n", avg_turnaround_time);

    return 0;
}
```