1. Program for system calls of Unix operating systems (opendir, readdir, closedir)

```
%%writefile os.c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main() {
    // Pointer to DIR type for the directory
    DIR *dir;
    struct dirent *entry;

    // Open the directory (current directory in this case)
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return EXIT_FAILURE;
    }

    // Read and print the contents of the directory
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);  // Print the name of the file or directory
    }

    // Close the directory after reading
    closedir(dir);

    return EXIT_SUCCESS;
}
```

```
Writing os.c
```

```
!gcc os.c -o os
!./os
```

```
.
..
.config
os
os.c
sample_data
```

2.Program for system calls of Unix operating system (fork, getpid, exit)

```
%%writefile os.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid;

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

    if (pid < 0) {
        // Fork failed
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // This block is executed by the child process
        printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
        exit(EXIT_SUCCESS);  // Child process terminates
    } else {
        // This block is executed by the parent process
        printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);
        exit(EXIT_SUCCESS);  // Parent process terminates
    }

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Parent Process: PID = 7348, Child PID = 7349
Child Process: PID = 7349, Parent PID = 1
```

3. CPU Scheduling Algorithms - FCFS

```
%%writefile os.c
#include <stdio.h>

int main() {
    int n = 4;  // Number of processes
    int burst_time[] = {2,3,4,5};  // Burst times for each process
    int waiting_time[n], turnaround_time[n];
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting time for each process
```

```
    waiting_time[0] = 0;  // First process has no waiting time
    for (int i = 1; i < n; i++) {
        waiting_time[i] = burst_time[i - 1] + waiting_time[i - 1];
    }

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
    }

    // Print the results
    printf("\nProcess\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", i + 1, burst_time[i], waiting_time[i], turnaround_tim
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
    }

    // Calculate and print average waiting and turnaround time
    printf("\nAverage Waiting Time = %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time = %.2f\n", (float)total_turnaround_time / n);

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Process Burst Time      Waiting Time    Turnaround Time
1       2               0               2
2       3               2               5
3       4               5               9
4       5               9               14

Average Waiting Time = 4.00
Average Turnaround Time = 7.50
```

4. CPU Scheduling Algorithms - SJF

```
%%writefile os.c
#include <stdio.h>

int main() {
    int n = 3;  // Number of processes
    int burst_time[] = {10, 8, 7};  // Burst times for each process
    int waiting_time[n], turnaround_time[n];
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Sort burst times in ascending order (SJF)
```

```c
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (burst_time[i] > burst_time[j]) {
                int temp = burst_time[i];
                burst_time[i] = burst_time[j];
                burst_time[j] = temp;
            }
        }
    }

    // Calculate waiting time for each process
    waiting_time[0] = 0;
    for (int i = 1; i < n; i++) {
        waiting_time[i] = burst_time[i - 1] + waiting_time[i - 1];
    }

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
    }

    // Print the results
    printf("\nProcess\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", i + 1, burst_time[i], waiting_time[i], turnaround_tim
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
    }

    // Average waiting and turnaround time
    printf("\nAverage Waiting Time = %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time = %.2f\n", (float)total_turnaround_time / n);

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Process Burst Time      Waiting Time    Turnaround Time
1       7               0               7
2       8               7               15
3       10              15              25

Average Waiting Time = 7.33
Average Turnaround Time = 15.67
```

5. CPU Scheduling Algorithms - SRTF

```c
%%writefile os.c
#include <stdio.h>

int main() {
    int n = 4;  // Number of processes
    int at[4] = {0, 1, 2, 3};  // Arrival times
    int bt[4] = {6,3,1,4};  // Burst times
    int rt[4];  // Remaining times
    int i, smallest, time, completed = 0, total_wt = 0, total_tat = 0;
    float avg_wt, avg_tat;

    // Initialize remaining time as burst time
    for (i = 0; i < n; i++) {
        rt[i] = bt[i];
    }

    printf("\nProcess\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n");

    for (time = 0; completed < n; time++) {
        smallest = -1;

        // Find the process with the smallest remaining time that has arrived
        for (i = 0; i < n; i++) {
            if (at[i] <= time && rt[i] > 0 && (smallest == -1 || rt[i] < rt[smallest])) {
                smallest = i;
            }
        }

        if (smallest == -1) {
            continue; // If no process is ready, skip the time unit
        }

        // Execute the selected process
        rt[smallest]--;

        // Process completed
        if (rt[smallest] == 0) {
            completed++;
            int end_time = time + 1;
            int turnaround_time = end_time - at[smallest];
            int waiting_time = turnaround_time - bt[smallest];

            total_tat += turnaround_time;
            total_wt += waiting_time;

            // Print details for the completed process
            printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", smallest + 1, at[smallest], bt[smallest],
        }
    }

    // Calculate and print averages
    avg_wt = (float)total_wt / n;
    avg_tat = (float)total_tat / n;
    printf("\nAverage Turnaround Time = %.2f", avg_tat);
    printf("\nAverage Waiting Time = %.2f\n", avg_wt);
```

```
    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Process Arrival Time    Burst Time      Turnaround Time Waiting Time
P3      2               1               1               0
P2      1               3               4               1
P4      3               4               6               2
P1      0               6               14              8

Average Turnaround Time = 6.25
Average Waiting Time = 2.75
```

6. Priority Scheduling (Non-preemptive)

```
%%writefile os.c

#include <stdio.h>

int main() {
  int process[]={1,2,3};
    int burst_time[] = {6, 8, 7};
    int priority[] = {2, 1, 3};  // 1 = Highest priority
    int waiting_time[3], turnaround_time[3];
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Sort based on priority
    for (int i = 0; i < 2; i++) {
        for (int j = i + 1; j < 3; j++) {
            if (priority[i] > priority[j]) {
                // Swap burst time
                int temp = burst_time[i];
                burst_time[i] = burst_time[j];
                burst_time[j] = temp;

                // Swap priority
                temp = priority[i];
                priority[i] = priority[j];
                priority[j] = temp;

                // Swap process
                temp = process[i];
                process[i] = process[j];
                process[j] = temp;

            }
```

```
        }
    }

    // Calculate waiting time
    waiting_time[0] = 0;
    for (int i = 1; i < 3; i++) {
        waiting_time[i] = burst_time[i - 1] + waiting_time[i - 1];
    }

    // Calculate turnaround time
    for (int i = 0; i < 3; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
    }

    // Print results
    printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < 3; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", process[i], burst_time[i], priority[i], waiting
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
    }

    // Averages
    printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / 3);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / 3);

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Process Burst Time      Priority        Waiting Time    Turnaround Time
2       8               1               0               8
1       6               2               8               14
3       7               3               14              21

Average Waiting Time: 7.33
Average Turnaround Time: 14.33
```

7. Round Robin Scheduling UPDATED

```
%%writefile os.c
#include <stdio.h>
```

```c
int main() {
    int n, quantum;

    // Step 1: Take input for number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int bt[n], temp_bt[n], wt[n], tat[n];

    // Step 2: Take input for burst times of processes
    printf("Enter the burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &bt[i]);
        temp_bt[i] = bt[i]; // Copy burst times to temp array
        wt[i] = 0; // Initialize waiting time to 0
    }

    // Step 3: Take input for time quantum
    printf("Enter the time quantum: ");
    scanf("%d", &quantum);

    int t = 0; // Initialize time
    // Step 4: Find waiting time for each process
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (temp_bt[i] > 0) {  // If process i is not finished
                done = 0;
                if (temp_bt[i] > quantum) {
                    t += quantum;
                    temp_bt[i] -= quantum;
                } else {
                    t += temp_bt[i];
                    wt[i] = t - bt[i];
                    temp_bt[i] = 0; // Process is done
                }
            }
        }
        if (done) break; // All processes are done
    }

    // Step 5: Calculate turnaround time
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time
    }

    // Step 6: Calculate average waiting time and average turnaround time
    int total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
```

```
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);

    return 0;
}
```

Overwriting os.c

```
#alternative
%%writefile os.c
#include <stdio.h>

int main() {
    int n = 5, quantum = 1; // Hardcoded number of processes and time quantum
    int bt[] = {10,1,2,1,5}; // Burst times for each process
    int temp_bt[] = {10,1,2,1,5}; // Copy of burst times for processing
    int wt[] = {0, 0, 0, 0,0}; // Waiting times initialized to 0
    int tat[5]; // Turnaround times

    int t = 0; // Initialize time

    // Step 1: Find waiting time for each process
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (temp_bt[i] > 0) {  // If process i is not finished
                done = 0;
                if (temp_bt[i] > quantum) {
                    t += quantum;
                    temp_bt[i] -= quantum;
                } else {
                    t += temp_bt[i];
                    wt[i] = t - bt[i];
                    temp_bt[i] = 0; // Process is done
                }
            }
        }
        if (done) break; // All processes are done
    }

    // Step 2: Calculate turnaround time
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time
    }

    // Step 3: Calculate average waiting time and average turnaround time
    int total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
```

```
    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Average waiting time = 5.40
Average turnaround time = 9.20
```

8. Producer Consumer Problem Using Semaphores

```
%%writefile os.c
#include <stdio.h>
#include <stdlib.h>

int mutex = 1, full = 0, empty = 3, x = 0; // Shared resources

void producer();
void consumer();
int wait(int);
int signal(int);

int main() {
    int n;
    printf("\n1. PRODUCER\n2. CONSUMER\n3. EXIT\n");

    while (1) {
        printf("\nENTER YOUR CHOICE: ");
        scanf("%d", &n);

        if (n == 1 && mutex == 1 && empty > 0) {
            producer();
        } else if (n == 2 && mutex == 1 && full > 0) {
            consumer();
        } else if (n == 3) {
            exit(0);
        } else {
            printf("Invalid choice or buffer is full/empty\n");
        }
    }

    return 0;
}

// Wait operation
int wait(int s) {
```

```c
    return --s;
}

// Signal operation
int signal(int s) {
    return ++s;
}

// Producer function
void producer() {
    mutex = wait(mutex);  // Lock mutex
    full = signal(full);   // Increase full
    empty = wait(empty);   // Decrease empty
    x++;                   // Produce item
    printf("Produced item %d\n", x);
    mutex = signal(mutex); // Unlock mutex
}

// Consumer function
void consumer() {
    mutex = wait(mutex);  // Lock mutex
    full = wait(full);    // Decrease full
    empty = signal(empty); // Increase empty
    printf("Consumed item %d\n", x);
    x--;                   // Consume item
    mutex = signal(mutex); // Unlock mutex
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
1. PRODUCER
2. CONSUMER
3. EXIT

ENTER YOUR CHOICE: 1
Produced item 1

ENTER YOUR CHOICE: 1
Produced item 2

ENTER YOUR CHOICE: 1
Produced item 3

ENTER YOUR CHOICE: 2
Consumed item 3

ENTER YOUR CHOICE: 2
Consumed item 2

ENTER YOUR CHOICE: 2
Consumed item 1
```

```
ENTER YOUR CHOICE: 2
Invalid choice or buffer is full/empty

ENTER YOUR CHOICE: 3
```

9. IPC using Shared Memory

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

#define SEGSIZE 100

int main(int argc, char *argv[]) {
    int shmid;
    key_t key;
    char *segptr;
    char buff[] = "poooda";

    // Generate a unique key for shared memory segment
    key = ftok(".", 's');

    // Try to get the shared memory segment, create if not exists
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
        // If shared memory already exists, just get it
        if((shmid = shmget(key, SEGSIZE, 0)) == -1) {
            perror("shmget");
            exit(1);
        }
    } else {
        printf("Creating a new shared memory segment \n");
        printf("SHMID: %d\n", shmid);
    }

    // Display all shared memory segments
    system("ipcs -m");

    // Attach the shared memory segment to process's address space
    if((segptr = (char*)shmat(shmid, 0, 0)) == (char*) -1) {
        perror("shmat");
        exit(1);
    }

    // Writing data to shared memory
    printf("Writing data to shared memory...\n");
    strcpy(segptr, buff);
    printf("DONE\n");
```

```
    // Reading data from shared memory
    printf("Reading data from shared memory...\n");
    printf("DATA: %s\n", segptr);
    printf("DONE\n");

    // Removing shared memory segment
    printf("Removing shared memory segment...\n");
    if(shmctl(shmid, IPC_RMID, 0) == -1) {
        printf("Can't remove shared memory segment...\n");
    } else {
        printf("Removed successfully\n");
    }

    return 0;
}
```

Overwriting os.c

```
%%writefile os.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define SEGSIZE 100

int main() {
    int shmid;
    key_t key;
    char *segptr;
    char buff[] = "poooda";

    // Generate or access shared memory segment
    key = ftok(".", 's');
    shmid = shmget(key, SEGSIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Display shared memory segments
    printf("Creating a new shared memory segment \n");
    printf("SHMID: %d\n", shmid);
    system("ipcs -m");

    // Attach shared memory
    segptr = (char*)shmat(shmid, 0, 0);
    if (segptr == (char*) -1) {
        perror("shmat");
        exit(1);
    }
```

```
    // Write to shared memory
    printf("Writing data to shared memory...\n");
    strcpy(segptr, buff);
    printf("DONE\n");

    // Read from shared memory
    printf("Reading data from shared memory...\n");
    printf("DATA: %s\n", segptr);
    printf("DONE\n");

    // Remove shared memory segment
    printf("Removing shared memory segment...\n");
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
    } else {
        printf("Removed successfully\n");
    }

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os

!./os
```

```
Creating a new shared memory segment
SHMID: 10

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000929 0          root       666        1024       0
0x73370006 10         root       666        100        0

Writing data to shared memory...
DONE
Reading data from shared memory...
DATA: poooda
DONE
Removing shared memory segment...
Removed successfully
```

```
!gcc os.c -o os

!./os
```

```
Creating a new shared memory segment
SHMID: 6
```

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000929 0          root       666        1024       0
0x73370006 6          root       666        100        0

Writing data to shared memory...
DONE
Reading data from shared memory...
DATA: poooda
DONE
Removing shared memory segment...
Removed successfully
```

Start coding or generate with AI.

ipc noterits

9. IPC USING SHARED MEMORY

```
ALGORITHM:
Step 1: Start the process
Step 2: Declare the segment size
Step 3: Create the shared memory
Step 4: Read the data from the shared memory
Step 5: Write the data to the shared memory
Step 6: Edit the data
Step 7: Stop the process.
```

```
%%writefile IPC_sender.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>

int main()
{
    int i;
    void *shared_memory;
    char buff[100];
    int shmid;

    shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
    printf("Key of shared memory is %d\n",shmid);

    shared_memory=shmat(shmid,NULL,0);
    printf("Process attached at %p\n",shared_memory);
```

```
    printf("Enter some data to write to shared memory\n");
    read(0,buff,100);
    strcpy(shared_memory,buff);

    printf("You wrote : %s\n",(char *)shared_memory);
}
```

```
Overwriting IPC_sender.c
```

```
!gcc IPC_sender.c
```

```
!./a.out
```

```
Key of shared memory is 0
Process attached at 0x7e777c282000
Enter some data to write to shared memory
22
You wrote : 22
```

```
%%writefile IPC_Reciever.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

```
Overwriting IPC_Reciever.c
```

```
!gcc IPC_Reciever.c
```

```
!./a.out
```

```
Key of shared memory is 1
Process attached at 0x13b534f91000
Data read from shared memory is : Namaskaram
```

10. Bankers Algorithm For Deadlock Avoidance

```
%%writefile os.c
#include <stdio.h>

#define P 3  // Number of processes
#define R 3  // Number of resources

int main() {
    // Available resources
    int avail[] = {31, 3, 2};

    // Maximum demand of each process
    int max[][R] = {
        {7, 5, 3},  // Process 0
        {3, 2, 2},  // Process 1
        {9, 0, 2}   // Process 2
    };

    // Resources allocated to each process
    int allot[][R] = {
        {0, 1, 0},  // Process 0
        {2, 0, 0},  // Process 1
        {3, 0, 2}   // Process 2
    };

    // Need matrix = Max - Allot
    int need[P][R];
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }

    // To track if a process can finish
    int finish[P] = {0};
    int work[R];

    // Initialize work[] to available resources
    for (int i = 0; i < R; i++) {
        work[i] = avail[i];
    }

    // Checking for safe state
    int safe = 1;
    for (int i = 0; i < P; i++) {
        if (finish[i] == 0) {
            int canFinish = 1;
            // Check if process i can finish
            for (int j = 0; j < R; j++) {
                if (need[i][j] > work[j]) {
                    canFinish = 0;
                    break;
```

```
                }
            }

            if (canFinish) {
                // Process can finish, add its resources to work[]
                for (int j = 0; j < R; j++) {
                    work[j] += allot[i][j];
                }
                finish[i] = 1;  // Mark process as finished
                printf("Process P%d can finish\n", i);
                i = -1;  // Start from the first process again
            }
        }
    }

    // Check if all processes finished
    for (int i = 0; i < P; i++) {
        if (finish[i] == 0) {
            safe = 0;
            break;
        }
    }

    if (safe) {
        printf("System is in a safe state.\n");
    } else {
        printf("System is in an unsafe state.\n");
    }

    return 0;
}
```

Overwriting os.c

```
%%writefile os.c
#include <stdio.h>

#define P 3  // Number of processes
#define R 3  // Number of resources

int main() {
    int max[][R] = {
        {7, 5, 3},  // Process 0
        {3, 2, 2},  // Process 1
        {9, 0, 2}   // Process 2
    };

    int allot[][R] = {
        {0, 1, 0},  // Process 0
        {2, 0, 0},  // Process 1
        {3, 0, 2}   // Process 2
    };

    int need[P][R], safe[P], available[R] = {3, 3, 2}; // Example available resources
```

```c
    int done[P] = {0};  // Track completed processes
    int count = 0;      // Count of completed processes

    // Calculate the need matrix
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }

    // Banker's Algorithm to find safe sequence
    while (count < P) {
        int found = 0;
        for (int i = 0; i < P; i++) {
            if (!done[i]) {
                int j;
                for (j = 0; j < R; j++) {
                    if (need[i][j] > available[j]) break;
                }
                if (j == R) {  // Process can finish
                    safe[count++] = i;
                    for (int k = 0; k < R; k++) {
                        available[k] += allot[i][k];  // Update available resources
                    }
                    done[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return -1;
        }
    }

    // Output the safe sequence
    printf("Safe sequence is: ");
    for (int i = 0; i < count; i++) {
        printf("P%d ", safe[i]);
    }
    printf("\n");

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
System is not in a safe state.
```

11. Memory Allocation Methods For Fixed Partition – First Fit

```
%%writefile os.c
#include <stdio.h>

#define PARTITIONS 5  // Number of partitions
#define PROCESSES 4   // Number of processes

int main() {
    int blockSize[PARTITIONS] = {100, 500, 200, 300, 600};
    int processSize[PROCESSES] = {212, 417, 112, 426};
    int allocation[PROCESSES];

    // Initialize all allocations as -1 (not allocated)
    for (int i = 0; i < PROCESSES; i++) {
        allocation[i] = -1;
    }

    // Allocate memory using First Fit
    for (int i = 0; i < PROCESSES; i++) {
        for (int j = 0; j < PARTITIONS; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;  // Allocate partition j to process i
                blockSize[j] -= processSize[i];  // Reduce block size
                break;
            }
        }
    }

    // Display results
    printf("\nProcess No. | Process Size | Block No. | Remaining Block Size\n");
    for (int i = 0; i < PROCESSES; i++) {
        if (allocation[i] != -1) {
            printf("%d           | %d             | %d        | %d\n",
                   i + 1, processSize[i], allocation[i] + 1, blockSize[allocation[i]]);
        } else {
            printf("%d           | %d             | Not Allocated\n", i + 1, processSize[i]);
        }
    }

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Process No. | Process Size | Block No. | Remaining Block Size
1           | 212             | 2        | 176
2           | 417             | 5        | 183
```

```
3            | 112            | 2          | 176
4            | 426            | Not Allocated
```

Start coding or generate with AI.

12. Memory Allocation Methods For Fixed Partition – best Fit

```
%%writefile os.c
#include <stdio.h>

int main() {
    int partitionSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int allocation[4];
    int partitions = 5, processes = 4;

    for (int i = 0; i < processes; i++) allocation[i] = -1;

    for (int i = 0; i < processes; i++) {
        int bestIdx = -1;
        for (int j = 0; j < partitions; j++) {
            if (partitionSize[j] >= processSize[i] &&
                (bestIdx == -1 || partitionSize[bestIdx] > partitionSize[j])) {
                bestIdx = j;
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            partitionSize[bestIdx] -= processSize[i];
        }
    }

    for (int i = 0; i < processes; i++) {
        if (allocation[i] != -1) {
            printf("Process %d of size %d -> Partition %d (Remaining size: %d)\n",
                   i + 1, processSize[i], allocation[i] + 1, partitionSize[allocation[i]]);
        } else {
            printf("Process %d of size %d -> Not Allocated\n", i + 1, processSize[i]);
        }
    }

    return 0;
}
```

```
Overwriting os.c
```

```
!gcc os.c -o os
!./os
```

```
Process No. | Process Size | Block No. | Remaining Block Size
1           | 212          | 4         | 88
2           | 417          | 2         | 83
3           | 112          | 3         | 88
4           | 426          | 5         | 174
```

Start coding or generate with AI.