DAY 1 PROGRAMS

1.Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

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

int main() {
  pid_t pid, parent_pid;

  // Get the process ID of the current process
  pid = getpid();

  // Get the parent process ID of the current process
  parent_pid = getppid();

  // Display the current process and parent PID
  printf("Current Process ID (PID): %d\n", pid);
```

```c
    printf("Parent Process ID (PPID): %d\n", parent_pid);

    // Fork to create a new process
    pid_t fork_pid = fork();

    if (fork_pid == 0) {
        // This block is executed by the child process
        printf("Child Process\n");
        printf("Child Process ID (PID): %d\n", getpid());
        printf("Parent Process ID (PPID) of Child: %d\n",
getppid());
    } else if (fork_pid > 0) {
        // This block is executed by the parent process
        printf("Parent Process\n");
        printf("Parent Process ID (PID): %d\n", getpid());
        printf("Child Process ID (PID) from Parent: %d\n",
fork_pid);
    } else {
        // Handle error if fork fails
        perror("Fork failed");
        return 1;
```

```
    }

    return 0;
}
```

OUTPUT:

```
Current Process ID (PID): 209
Parent Process ID (PPID): 198
Parent Process
Parent Process ID (PID): 209
Child Process ID (PID) from Parent: 210
Child Process
Child Process ID (PID): 210
Parent Process ID (PPID) of Child: 209
```

2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program.

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

```
#include <sys/types.h>
#include <sys/stat.h>

#define BUFFER_SIZE 1024  // Size of the buffer for
reading and writing

int main(int argc, char *argv[]) {
    int src_fd, dest_fd, read_bytes, write_bytes;
    char buffer[BUFFER_SIZE];

    // Check if source and destination file paths are
provided
    if (argc != 3) {
        printf("Usage: %s <source_file>
<destination_file>\n", argv[0]);
        return 1;
    }

    // Open the source file in read-only mode
    src_fd = open(argv[1], O_RDONLY);
    if (src_fd == -1) {
```

```
        perror("Error opening source file");
        return 1;
    }

    // Open (or create) the destination file in write-only
mode, create it if it doesn't exist, and set appropriate
permissions
    dest_fd = open(argv[2], O_WRONLY | O_CREAT |
O_TRUNC, 0644);
    if (dest_fd == -1) {
        perror("Error opening destination file");
        close(src_fd);  // Close the source file before
exiting
        return 1;
    }

    // Read from the source file and write to the
destination file
    while ((read_bytes = read(src_fd, buffer,
BUFFER_SIZE)) > 0) {
        write_bytes = write(dest_fd, buffer, read_bytes);
        if (write_bytes != read_bytes) {
```

```
        perror("Error writing to destination file");
        close(src_fd);
        close(dest_fd);
        return 1;
    }
}

// Check for errors in reading from the source file
if (read_bytes == -1) {
    perror("Error reading from source file");
    close(src_fd);
    close(dest_fd);
    return 1;
}

// Close both the source and destination files
close(src_fd);
close(dest_fd);

printf("File copied successfully.\n");
```

```
    return 0;
}
```

OUTPUT:

```
Usage: /tmp/SwO3Ri8jUX/main.o <source_file> <destination_file>
```

3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices

```
#include <stdio.h>

typedef struct {
    int process_id;  // Process ID
    int burst_time;  // Burst Time
    int waiting_time; // Waiting Time
    int turnaround_time; // Turnaround Time
} Process;
```

```
// Function to calculate waiting time and turnaround
time for each process
void calculate_times(Process processes[], int n) {
    // First process has no waiting time
    processes[0].waiting_time = 0;

    // Calculate waiting time for all processes
    for (int i = 1; i < n; i++) {
        processes[i].waiting_time = processes[i-
1].waiting_time + processes[i-1].burst_time;
    }

    // Calculate turnaround time for all processes
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time =
processes[i].burst_time + processes[i].waiting_time;
    }
}
```

```
// Function to calculate average waiting time and
turnaround time
void calculate_average_times(Process processes[], int
n) {
    int total_waiting_time = 0, total_turnaround_time =
0;

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

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

```
// Function to print the process details
void print_processes(Process processes[], int n) {
    printf("Process\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",
processes[i].process_id, processes[i].burst_time,
processes[i].waiting_time,
processes[i].turnaround_time);
    }
}

int main() {
    int n;

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

    Process processes[n];
```

```
    // Input burst time for each process
    for (int i = 0; i < n; i++) {
        processes[i].process_id = i + 1;  // Process IDs start
from 1
        printf("Enter burst time for Process %d: ",
processes[i].process_id);
        scanf("%d", &processes[i].burst_time);
    }

    // Calculate waiting times and turnaround times
    calculate_times(processes, n);

    // Print process details
    print_processes(processes, n);

    // Calculate and print average times
    calculate_average_times(processes, n);

    return 0;
```

Output

```
Enter number of processes: 3
Enter burst time for Process 1: 5
Enter burst time for Process 2: 3
Enter burst time for Process 3: 2
Process Burst Time  Waiting Time    Turnaround Time
1    5        0        5
2    3        5        8
3    2        8        10
Average Waiting Time: 4.33
Average Turnaround Time: 7.67
```

4. Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

```c
#include <stdio.h>


typedef struct {
    int process_id;    // Process ID
    int burst_time;    // Burst Time
    int waiting_time;  // Waiting Time
    int turnaround_time; // Turnaround Time
} Process;
```

```
// Function to calculate waiting time and turnaround time for each process
void calculate_times(Process processes[], int n) {
    // First process has no waiting time
    processes[0].waiting_time = 0;

    // Calculate waiting time for all processes
    for (int i = 1; i < n; i++) {
        processes[i].waiting_time = processes[i-1].waiting_time + processes[i-1].burst_time;
    }

    // Calculate turnaround time for all processes
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].burst_time + processes[i].waiting_time;
    }
}
```

```
// Function to calculate average waiting time and
turnaround time
void calculate_average_times(Process processes[], int
n) {
    int total_waiting_time = 0, total_turnaround_time =
0;

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

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

```
// Function to print the process details
void print_processes(Process processes[], int n) {
    printf("Process\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",
processes[i].process_id, processes[i].burst_time,
processes[i].waiting_time,
processes[i].turnaround_time);
    }
}

// Function to sort the processes based on their burst
times (Shortest Job First)
void sort_by_burst_time(Process processes[], int n) {
    Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].burst_time >
processes[j].burst_time) {
                // Swap the processes
```

```
            temp = processes[i];
            processes[i] = processes[j];
            processes[j] = temp;
        }
    }
  }
}

int main() {
  int n;

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

  Process processes[n];

  // Input burst time for each process
  for (int i = 0; i < n; i++) {
```

```
        processes[i].process_id = i + 1;  // Process IDs start from 1
        printf("Enter burst time for Process %d: ", processes[i].process_id);
        scanf("%d", &processes[i].burst_time);
    }

    // Sort the processes based on burst time (Shortest Job First)
    sort_by_burst_time(processes, n);

    // Calculate waiting times and turnaround times
    calculate_times(processes, n);

    // Print process details
    print_processes(processes, n);

    // Calculate and print average times
    calculate_average_times(processes, n);

    return 0;
```

```
}
```

Output

```
Enter number of processes: 3
Enter burst time for Process 1: 6
Enter burst time for Process 2: 2
Enter burst time for Process 3: 8
Process Burst Time  Waiting Time    Turnaround Time
2   2       0       2
1   6       2       8
3   8       8       16
Average Waiting Time: 3.33
Average Turnaround Time: 8.67
```

5. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

```
#include <stdio.h>

#include <stdlib.h>


// Define a structure for a process

typedef struct {

    int processID;  // Process ID
```

```
    int burstTime;  // Burst time (time needed to
execute)
    int priority;   // Priority (lower number means higher
priority)
} Process;

// Function to compare processes based on their
priority
int compare(const void* a, const void* b) {
    return ((Process*)a)->priority - ((Process*)b)-
>priority;
}

// Function to execute the processes based on priority
void executeProcesses(Process* processes, int n) {
    printf("Process Execution Order:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d (Burst Time: %d, Priority:
%d)\n",
               processes[i].processID, processes[i].burstTime,
processes[i].priority);
    }
```

```
}

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

    // Create an array of processes
    Process processes[n];

    // Input details of each process
    for (int i = 0; i < n; i++) {
        printf("Enter details for process %d:\n", i + 1);
        processes[i].processID = i + 1;
        printf("  Enter burst time: ");
        scanf("%d", &processes[i].burstTime);
        printf("  Enter priority: ");
        scanf("%d", &processes[i].priority);
    }
```

```
    // Sort processes by priority (higher priority should come first)
    qsort(processes, n, sizeof(Process), compare);

    // Execute and print the processes based on priority
    executeProcesses(processes, n);

    return 0;
```

```
Output
Enter the number of processes: 3
Enter details for process 1:
  Enter burst time: 6
  Enter priority: 4
Enter details for process 2:
  Enter burst time: 1
  Enter priority: 2
Enter details for process 3:
  Enter burst time: 4
  Enter priority: 2
Process Execution Order:
Process 2 (Burst Time: 1, Priority: 2)
Process 3 (Burst Time: 4, Priority: 2)
Process 1 (Burst Time: 6, Priority: 4)
```

6. Construct a C program to implement preemptive priority scheduling algorithm

```
#include <stdio.h>

struct Process {
    int pid;          // Process ID
```

```
    int burst_time;    // Burst time of the process
    int priority;      // Priority of the process
    int remaining_time; // Remaining time for the
process
    int waiting_time;  // Waiting time for the process
    int turnaround_time; // Turnaround time for the
process
};

void findWaitingTime(struct Process proc[], int n) {
    int completed = 0, time = 0, min_priority = -1;
    int last_completed = -1;

    while (completed < n) {
        min_priority = -1;

        // Find the process with the highest priority
(lowest priority number)
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0) {
```

```
            if (min_priority == -1 || proc[i].priority <
proc[min_priority].priority) {
                min_priority = i;
            }
        }
    }

    if (min_priority == -1) {
        break;
    }

    // Execute the process
    proc[min_priority].remaining_time--;
    time++;

    // If the process finishes
    if (proc[min_priority].remaining_time == 0) {
        completed++;
        proc[min_priority].waiting_time = time -
proc[min_priority].burst_time;
```

```
          proc[min_priority].turnaround_time =
proc[min_priority].waiting_time +
proc[min_priority].burst_time;
        }
    }
}

void findTurnAroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].turnaround_time = proc[i].waiting_time +
proc[i].burst_time;
    }
}

void findAvgTime(struct Process proc[], int n) {
    int total_waiting_time = 0, total_turnaround_time =
0;
    printf("PID\tBurst Time\tPriority\tWaiting
Time\tTurnaround Time\n");

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

```
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time +=
proc[i].turnaround_time;

        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid,
proc[i].burst_time, proc[i].priority,
proc[i].waiting_time, proc[i].turnaround_time);
    }

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

int main() {
    int n;

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

```
    struct Process proc[n];

    // Input the process details
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time and priority for Process
%d: ", proc[i].pid);
        scanf("%d %d", &proc[i].burst_time,
&proc[i].priority);
        proc[i].remaining_time = proc[i].burst_time; //
Remaining time initially equals to burst time
    }

    findWaitingTime(proc, n);
    findAvgTime(proc, n);

    return 0;
}
```

```
Output
Enter number of processes: 3
Enter burst time and priority for Process 1: 6 1
Enter burst time and priority for Process 2: 8 4
Enter burst time and priority for Process 3: 7 2
PID Burst Time  Priority    Waiting Time    Turnaround Time
1   6       1       0       6
2   8       4       13      21
3   7       2       6       13

Average Waiting Time: 6.33
Average Turnaround Time: 13.33
```

7. Construct a C program to implement a non-preemptive SJF algorithm.

```
#include <stdio.h>


struct Process {
    int id;           // 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
(Shortest Job First)
void sortProcesses(struct Process processes[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].burst_time >
processes[j].burst_time) {
                // Swap processes[i] and processes[j]
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}

// Function to calculate waiting time and turnaround
time
void calculateTimes(struct Process processes[], int n) {
```

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

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

// Function to calculate average waiting time and
turnaround time
void calculateAverageTimes(struct Process processes[],
int n) {
    int total_waiting_time = 0, total_turnaround_time =
0;
```

```
    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time +=
processes[i].turnaround_time;
    }

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

int main() {
    int n;

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

    struct Process processes[n];
```

```c
    // Input burst times for the processes
    printf("Enter the burst times of %d processes:\n", n);
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1; // Process ID starts from 1
        printf("Burst time for process %d: ",
processes[i].id);
        scanf("%d", &processes[i].burst_time);
    }

    // Sort the processes based on burst time (Shortest
Job First)
    sortProcesses(processes, n);

    // Calculate waiting time and turnaround time for
each process
    calculateTimes(processes, n);

    // Print the process information and times
    printf("\nProcess ID | Burst Time | Waiting Time |
Turnaround Time\n");
```

```
    printf("------------------------------------------------------------
\n");
    for (int i = 0; i < n; i++) {
        printf("    %d      |     %d     |      %d      |      %d\n",
               processes[i].id, processes[i].burst_time,
               processes[i].waiting_time,
processes[i].turnaround_time);
    }

    // Calculate and print the average times
    calculateAverageTimes(processes, n);

    return 0;
```

}

```
Output
Enter the number of processes: 3
Enter the burst times of 3 processes:
Burst time for process 1: 6
Burst time for process 2: 8
Burst time for process 3: 7

Process ID | Burst Time | Waiting Time | Turnaround Time
----------------------------------------------------------
    1      |     6      |      0       |      6
    3      |     7      |      6       |      13
    2      |     8      |      13       |      21

Average Waiting Time: 6.33
Average Turnaround Time: 13.33
```

8. Construct a C program to simulate Round Robin scheduling algorithm with C

```
#include <stdio.h>

#define MAX 10  // Maximum number of processes

// Structure to represent a process
struct Process {
```

```c
    int id;        // Process ID
    int bt;        // Burst time (total time required for
execution)
    int wt;        // Waiting time
    int tat;       // Turnaround time
};

// Function to find waiting time of all processes
void findWaitingTime(struct Process proc[], int n, int q)
{
    int rem_bt[n];  // Array to store remaining burst time
for each process
    int t = 0;      // Current time
    for (int i = 0; i < n; i++) {
        rem_bt[i] = proc[i].bt;
    }

    while (1) {
        int done = 1;  // Flag to check if all processes are
done
        for (int i = 0; i < n; i++) {
```

```
            if (rem_bt[i] > 0) {
                done = 0;
                if (rem_bt[i] > q) {
                    t += q;
                    rem_bt[i] -= q;
                } else {
                    t += rem_bt[i];
                    proc[i].wt = t - proc[i].bt;
                    rem_bt[i] = 0;
                }
            }
        }
        if (done == 1)
            break;
    }
}

// Function to find turnaround time of all processes
void findTurnAroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
```

```
        proc[i].tat = proc[i].bt + proc[i].wt;
    }
}

// Function to calculate average waiting time and
turnaround time
void findAverageTime(struct Process proc[], int n) {
    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += proc[i].wt;
        total_tat += proc[i].tat;
    }
    printf("\nAverage waiting time: %.2f", total_wt / n);
    printf("\nAverage turnaround time: %.2f", total_tat /
n);
}

// Main function to drive the Round Robin scheduling
int main() {
    struct Process proc[MAX];
    int n, q;
```

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

    // Taking burst time input for each process
    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &proc[i].bt);
    }

    // Calculating waiting time and turnaround time
    findWaitingTime(proc, n, q);
    findTurnAroundTime(proc, n);

    // Displaying the process information
    printf("\nProcess ID\tBurst Time\tWaiting
Time\tTurnaround Time\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", proc[i].id, 
proc[i].bt, proc[i].wt, proc[i].tat);
    }

    // Calculating and displaying the average times
    findAverageTime(proc, n);

    return 0;
}
```

Output

```
Enter number of processes: 3
Enter the time quantum: 4
Enter burst time for Process 1: 6
Enter burst time for Process 2: 8
Enter burst time for Process 3: 7

Process ID  Burst Time  Waiting Time    Turnaround Time
1       6       8       14
2       8       10      18
3       7       14      21

Average waiting time: 10.67
Average turnaround time: 17.67
```

9. Illustrate the concept of inter-process communication using shared memory with a C program.

WRITER CODE

#include <stdio.h>

Output

```
Writer: Writing to shared memory...
Writer: Message written. Waiting for reader to read...
Writer: Detached from shared memory.
Writer: Shared memory removed.
```

Output

```
Writer: Writing to shared memory...
Writer: Message written. Waiting for reader to read...
Writer: Detached from shared memory.
Writer: Shared memory removed.
```


#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#include <unistd.h>

```
#define SHM_SIZE 1024  // Size of the shared memory
segment

int main() {
   key_t key = 1234;  // Arbitrary key for shared
memory
   int shm_id;
   char *shm_ptr;

   // Create shared memory segment
   shm_id = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
   if (shm_id == -1) {
      perror("shmget failed");
      exit(1);
   }

   // Attach to shared memory
   shm_ptr = (char*) shmat(shm_id, NULL, 0);
   if (shm_ptr == (char*) -1) {
      perror("shmat failed");
      exit(1);
```

```
    }

    // Write a message to shared memory
    printf("Writer: Writing to shared memory...\n");
    strncpy(shm_ptr, "Hello from the server!",
SHM_SIZE);

    // Wait for the reader process to read
    printf("Writer: Message written. Waiting for reader
to read...\n");
    sleep(5);

    // Detach from shared memory
    shmdt(shm_ptr);
    printf("Writer: Detached from shared memory.\n");

    // Cleanup shared memory (this would typically be
done by the server when done)
    shmctl(shm_id, IPC_RMID, NULL);
    printf("Writer: Shared memory removed.\n");
```

```
    return 0;
}
```

Output

```
Writer: Writing to shared memory...
Writer: Message written. Waiting for reader to read...
Writer: Detached from shared memory.
Writer: Shared memory removed.
```

READING CODE

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

#define SHM_SIZE 1024  // Size of the shared memory segment

int main() {
    key_t key = 1234;  // Same key as the server process
```

```
int shm_id;
char *shm_ptr;

// Access the shared memory segment
shm_id = shmget(key, SHM_SIZE, 0666);
if (shm_id == -1) {
    perror("shmget failed");
    exit(1);
}

// Attach to shared memory
shm_ptr = (char*) shmat(shm_id, NULL, 0);
if (shm_ptr == (char*) -1) {
    perror("shmat failed");
    exit(1);
}

// Read from shared memory
printf("Reader: Reading from shared memory...\n");
printf("Reader: Message: %s\n", shm_ptr);
```

```
    // Detach from shared memory
    shmdt(shm_ptr);
    printf("Reader: Detached from shared memory.\n");

    return 0;
}
```

OUTPUT:

```
Reader: Reading from shared memory...
Reader: Message: Hello from the server!
Reader: Detached from shared memory.
```

10. Illustrate the concept of inter-process communication using message queue with a C program.

CODE FOR SENDER PROCESS

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

```
#include <sys/msg.h>
#include <unistd.h>

// Define message structure
struct message {
    long msg_type;  // Message type, used to
differentiate messages
    char msg_text[100];  // Message text
};

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

    // Generate unique key
    key = ftok("message_queue", 65);
    if (key == -1) {
        perror("ftok failed");
        exit(1);
```

```
    }

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

    // Send messages to the queue
    for (int i = 0; i < 5; i++) {
        msg.msg_type = 1;  // Setting message type
        sprintf(msg.msg_text, "Message %d from sender",
i + 1);

        // Send message to queue
        if (msgsnd(msgid, &msg, sizeof(msg), 0) == -1) {
            perror("msgsnd failed");
            exit(1);
        }
```

```c
        printf("Sent: %s\n", msg.msg_text);
        sleep(1);  // Simulate some delay
    }

    return 0;
}
```

CODE FOR RECIEVER PROCESS

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

// Define message structure
struct message {
    long msg_type;
    char msg_text[100];
};
```

```
int main() {
    key_t key;
    int msgid;
    struct message msg;

    // Generate unique key
    key = ftok("message_queue", 65);
    if (key == -1) {
        perror("ftok failed");
        exit(1);
    }

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

```
    // Receive messages from the queue
    for (int i = 0; i < 5; i++) {
        // Receive message from queue
        if (msgrcv(msgid, &msg, sizeof(msg), 1, 0) == -1) {
            perror("msgrcv failed");
            exit(1);
        }

        printf("Received: %s\n", msg.msg_text);
    }

    // Destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

DAY 2 PROGRAMS

11.Illustrate the concept of multithreading using a C program

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

Output

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Letter: A
Letter: B
Letter: C
Letter: D
Letter: E
Main thread finished executing
```

```
// Function to print numbers from 1 to 5
void* printNumbers(void* arg) {
    for (int i = 1; i <= 5; i++) {
```

```
        printf("Number: %d\n", i);
    }
    return NULL;
}

// Function to print letters from A to E
void* printLetters(void* arg) {
    for (char c = 'A'; c <= 'E'; c++) {
        printf("Letter: %c\n", c);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;  // Declare two
threads

    // Create thread 1 to print numbers
    if (pthread_create(&thread1, NULL, printNumbers,
NULL) != 0) {
        printf("Error creating thread 1\n");
```

```
        return 1;
    }

    // Create thread 2 to print letters
    if (pthread_create(&thread2, NULL, printLetters,
NULL) != 0) {
        printf("Error creating thread 2\n");
        return 1;
    }

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Main thread finished executing\n");

    return 0;
}
```

```
Output
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Letter: A
Letter: B
Letter: C
Letter: D
Letter: E
Main thread finished executing
```

12. Design a C program to simulate the concept of Dining-Philosophers problem

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

#define NUM_PHILOSOPHERS 5
```

```
pthread_mutex_t forks[NUM_PHILOSOPHERS];

void* philosopher(void* num) {
    int philosopher_id = *(int*)num;
    int left_fork = philosopher_id;
    int right_fork = (philosopher_id + 1) %
NUM_PHILOSOPHERS;

    while (1) {
        // Thinking
        printf("Philosopher %d is thinking.\n",
philosopher_id);
        usleep(rand() % 1000);  // Philosopher thinking for
a while

        // Attempt to pick up forks
        pthread_mutex_lock(&forks[left_fork]);
        printf("Philosopher %d picked up left fork %d.\n",
philosopher_id, left_fork);
        pthread_mutex_lock(&forks[right_fork]);
```

```
        printf("Philosopher %d picked up right fork %d.\n",
philosopher_id, right_fork);

        // Eating
        printf("Philosopher %d is eating.\n",
philosopher_id);
        usleep(rand() % 1000);  // Philosopher eating for a
while

        // Put down forks
        pthread_mutex_unlock(&forks[left_fork]);
        printf("Philosopher %d put down left fork %d.\n",
philosopher_id, left_fork);
        pthread_mutex_unlock(&forks[right_fork]);
        printf("Philosopher %d put down right fork %d.\n",
philosopher_id, right_fork);
    }
    return NULL;
}

int main() {
```

```
pthread_t threads[NUM_PHILOSOPHERS];
int philosopher_ids[NUM_PHILOSOPHERS];

// Initialize the mutexes
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
    pthread_mutex_init(&forks[i], NULL);
}

// Create philosopher threads
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
    philosopher_ids[i] = i;
    if (pthread_create(&threads[i], NULL, philosopher,
(void*)&philosopher_ids[i]) != 0) {
        perror("Failed to create thread");
        exit(1);
    }
}

// Join philosopher threads (this never ends, so the
main thread just waits)
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
```

```
        pthread_join(threads[i], NULL);
    }

    // Destroy the mutexes
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_destroy(&forks[i]);
    }

    return 0;
}
```

Output

```
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 0 picked up left fork 0.
Philosopher 0 picked up right fork 1.
Philosopher 0 is eating.
Philosopher 3 picked up left fork 3.
Philosopher 3 picked up right fork 4.
Philosopher 3 is eating.
Philosopher 0 put down left fork 0.
Philosopher 0 put down right fork 1.
Philosopher 0 is thinking.
Philosopher 2 picked up left fork 2.
Philosopher 1 picked up left fork 1.
Philosopher 2 picked up right fork 3.
Philosopher 2 is eating.
Philosopher 3 put down left fork 3.
Philosopher 3 put down right fork 4.
Philosopher 3 is thinking.
Philosopher 4 picked up left fork 4.
Philosopher 4 picked up right fork 0.
```

13. Construct a C program for implementation of the various memory allocation strategies.

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

#define MAX_BLOCKS 10
#define MAX_PROCESSES 5

struct Block {
    int size;          // Size of the memory block
    int allocated;     // 1 if allocated, 0 if free
};

struct Process {
    int size;          // Size of the process
    int allocated;     // 1 if allocated, 0 if not
};

void displayMemory(struct Block blocks[], int
numBlocks) {
```

```
    printf("\nBlock No. | Block Size | Allocation Status\n");
    for (int i = 0; i < numBlocks; i++) {
        printf("    %d     |     %d     |     %s\n",
               i + 1, blocks[i].size,
               blocks[i].allocated ? "Allocated" : "Free");
    }
}

void firstFit(struct Block blocks[], int numBlocks, struct Process processes[], int numProcesses) {
    printf("\n--- First Fit Allocation ---\n");
    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numBlocks; j++) {
            if (blocks[j].allocated == 0 && blocks[j].size >= processes[i].size) {
                blocks[j].allocated = 1;
                processes[i].allocated = 1;
                printf("Process %d of size %d allocated to Block %d of size %d\n",
                       i + 1, processes[i].size, j + 1, blocks[j].size);
```

```
            break;
         }
      }
   }
   displayMemory(blocks, numBlocks);
}

void bestFit(struct Block blocks[], int numBlocks, struct
Process processes[], int numProcesses) {
   printf("\n--- Best Fit Allocation ---\n");
   for (int i = 0; i < numProcesses; i++) {
      int bestIndex = -1;
      for (int j = 0; j < numBlocks; j++) {
         if (blocks[j].allocated == 0 && blocks[j].size >=
processes[i].size) {
            if (bestIndex == -1 || blocks[j].size <
blocks[bestIndex].size) {
               bestIndex = j;
            }
         }
      }
```

```
        if (bestIndex != -1) {
            blocks[bestIndex].allocated = 1;
            processes[i].allocated = 1;
            printf("Process %d of size %d allocated to Block
%d of size %d\n",
                   i + 1, processes[i].size, bestIndex + 1,
blocks[bestIndex].size);
        }
    }
    displayMemory(blocks, numBlocks);
}

void worstFit(struct Block blocks[], int numBlocks,
struct Process processes[], int numProcesses) {
    printf("\n--- Worst Fit Allocation ---\n");
    for (int i = 0; i < numProcesses; i++) {
        int worstIndex = -1;
        for (int j = 0; j < numBlocks; j++) {
            if (blocks[j].allocated == 0 && blocks[j].size >=
processes[i].size) {
```

```
            if (worstIndex == -1 || blocks[j].size >
blocks[worstIndex].size) {
                worstIndex = j;
            }
        }
    }
    if (worstIndex != -1) {
        blocks[worstIndex].allocated = 1;
        processes[i].allocated = 1;
        printf("Process %d of size %d allocated to Block
%d of size %d\n",
               i + 1, processes[i].size, worstIndex + 1,
blocks[worstIndex].size);
    }
  }
  displayMemory(blocks, numBlocks);
}

int main() {
  int numBlocks, numProcesses;
  struct Block blocks[MAX_BLOCKS];
```

```
    struct Process processes[MAX_PROCESSES];

    // Input the number of blocks and processes
    printf("Enter the number of memory blocks: ");
    scanf("%d", &numBlocks);

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

    // Input the sizes of the memory blocks
    printf("Enter the sizes of the %d memory blocks:\n",
numBlocks);
    for (int i = 0; i < numBlocks; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blocks[i].size);
        blocks[i].allocated = 0;  // Mark all blocks as free
initially
    }

    // Input the sizes of the processes
```

```
    printf("Enter the sizes of the %d processes:\n", 
numProcesses);
    for (int i = 0; i < numProcesses; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processes[i].size);
        processes[i].allocated = 0;  // Mark all processes as 
unallocated initially
    }

    // First Fit Allocation
    firstFit(blocks, numBlocks, processes, numProcesses);

    // Reset memory blocks for the next allocation
    for (int i = 0; i < numBlocks; i++) {
        blocks[i].allocated = 0;
    }
    for (int i = 0; i < numProcesses; i++) {
        processes[i].allocated = 0;
    }

    // Best Fit Allocation
```

```
    bestFit(blocks, numBlocks, processes,
numProcesses);

    // Reset memory blocks for the next allocation
    for (int i = 0; i < numBlocks; i++) {
        blocks[i].allocated = 0;
    }
    for (int i = 0; i < numProcesses; i++) {
        processes[i].allocated = 0;
    }

    // Worst Fit Allocation
    worstFit(blocks, numBlocks, processes,
numProcesses);

    return 0;
}
```

Output

```
Enter the sizes of the 5 memory blocks:
Block 1 size: 50
Block 2 size: 100
Block 3 size: 150
Block 4 size: 200
Block 5 size: 300
Enter the sizes of the 3 processes:
Process 1 size: 90
Process 2 size: 80
Process 3 size: 100

--- First Fit Allocation ---
Process 1 of size 90 allocated to Block 2 of size 100
Process 2 of size 80 allocated to Block 3 of size 150
Process 3 of size 100 allocated to Block 4 of size 200

Block No. | Block Size | Allocation Status
    1     |     50     |     Free
    2     |     100     |     Allocated
    3     |     150     |     Allocated
    4     |     200     |     Allocated
    5     |     300     |     Free

--- Best Fit Allocation ---
Process 1 of size 90 allocated to Block 2 of size 100
Process 2 of size 80 allocated to Block 3 of size 150
Process 3 of size 100 allocated to Block 4 of size 200

Block No. | Block Size | Allocation Status
    1     |     50     |     Free
    2     |     100     |     Allocated
    3     |     150     |     Allocated
    4     |     200     |     Allocated
    5     |     300     |     Free

--- Worst Fit Allocation ---
Process 1 of size 90 allocated to Block 5 of size 300
Process 2 of size 80 allocated to Block 4 of size 200
Process 3 of size 100 allocated to Block 3 of size 150

Block No. | Block Size | Allocation Status
    1     |     50     |     Free
    2     |     100     |     Free
    3     |     150     |     Allocated
    4     |     200     |     Allocated
    5     |     300     |     Allocated
```

14. Construct a C program to organise the file using a single level directory.

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

// Function to create a file
void createFile(char *fileName) {
    FILE *file = fopen(fileName, "w");
    if (file == NULL) {
        perror("Error creating file");
    } else {
        printf("File '%s' created successfully.\n", fileName);
        fclose(file);
    }
}
```

```
// Function to list files in the directory
void listFiles(char *dirName) {
    DIR *dir = opendir(dirName);
    struct dirent *entry;

    if (dir == NULL) {
        perror("Error opening directory");
        return;
    }

    printf("Files in directory '%s':\n", dirName);
    while ((entry = readdir(dir)) != NULL) {
        // Skip the '.' and '..' directories
        if (strcmp(entry->d_name, ".") != 0 &&
strcmp(entry->d_name, "..") != 0) {
            printf("%s\n", entry->d_name);
        }
    }

    closedir(dir);
```

```
}

// Function to delete a file
void deleteFile(char *fileName) {
    if (remove(fileName) == 0) {
        printf("File '%s' deleted successfully.\n", fileName);
    } else {
        perror("Error deleting file");
    }
}

int main() {
    char dirName[256];
    char fileName[256];
    int choice;

    // Get the directory name from user
    printf("Enter the directory name to organize: ");
    scanf("%s", dirName);
```

```
    // Check if the directory exists, if not create it
    struct stat st = {0};
    if (stat(dirName, &st) == -1) {
        if (mkdir(dirName, 0700) == 0) {
            printf("Directory '%s' created successfully.\n",
dirName);
        } else {
            perror("Error creating directory");
            return 1;
        }
    }

    while (1) {
        // Menu for file operations
        printf("\nFile Organization Menu:\n");
        printf("1. Create File\n");
        printf("2. List Files\n");
        printf("3. Delete File\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
```

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

        switch (choice) {
            case 1:
                // Create a file
                printf("Enter the file name to create: ");
                scanf("%s", fileName);
                char filePath[512];
                snprintf(filePath, sizeof(filePath), "%s/%s",
dirName, fileName);
                createFile(filePath);
                break;

            case 2:
                // List files
                listFiles(dirName);
                break;

            case 3:
                // Delete a file
```

```
                printf("Enter the file name to delete: ");
                scanf("%s", fileName);
                snprintf(filePath, sizeof(filePath), "%s/%s",
dirName, fileName);
                deleteFile(filePath);
                break;

            case 4:
                // Exit the program
                printf("Exiting the program.\n");
                exit(0);

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
Enter the directory name to organize: test_directory
Directory 'test_directory' created successfully.

File Organization Menu:
1. Create File
2. List Files
3. Delete File
4. Exit
Enter your choice: 1
Enter the file name to create: file1.txt
File 'test_directory/file1.txt' created successfully.

File Organization Menu:
1. Create File
2. List Files
3. Delete File
4. Exit
Enter your choice: 2
Files in directory 'test_directory':
file1.txt

File Organization Menu:
1. Create File
2. List Files
3. Delete File
4. Exit
Enter your choice: 3
Enter the file name to delete: file1.txt
File 'test_directory/file1.txt' deleted successfully.

File Organization Menu:
1. Create File
2. List Files
3. Delete File
4. Exit
Enter your choice: 4
Exiting the program.
```

15.Design a C program to organise the file using a two level directory structure.

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

// Function to create directory if it does not exist
void create_directory(const char *dir_name) {
    struct stat st = {0};
    if (stat(dir_name, &st) == -1) {
        if (mkdir(dir_name, 0700) == -1) {
            perror("Unable to create directory");
            exit(1);
        }
        printf("Directory '%s' created successfully.\n",
dir_name);
```

```
    }
}

// Function to get the file extension
const char *get_file_extension(const char *filename) {
    const char *dot = strrchr(filename, '.');
    if (!dot || dot == filename) return "";
    return dot + 1;
}

// Function to move file to a directory
void move_file(const char *src, const char *dest) {
    char new_file_path[1024];
    snprintf(new_file_path, sizeof(new_file_path),
"%s/%s", dest, strrchr(src, '/') + 1);
    if (rename(src, new_file_path) != 0) {
        perror("Unable to move file");
        exit(1);
    }
    printf("File '%s' moved to '%s'.\n", src,
new_file_path);
```

```
}

// Function to organize files in the current directory
void organize_files() {
    struct dirent *entry;
    DIR *dp = opendir(".");
    if (dp == NULL) {
        perror("Unable to open directory");
        exit(1);
    }

    // Create subdirectories for categorization
    create_directory("Images");
    create_directory("TextFiles");
    create_directory("OtherFiles");

    // Iterate over each file in the current directory
    while ((entry = readdir(dp)) != NULL) {
        // Skip the "." and ".." directories
```

```
        if (strcmp(entry->d_name, ".") == 0 ||
strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        const char *ext = get_file_extension(entry-
>d_name);

        // Categorize the file based on its extension
        if (strcmp(ext, "jpg") == 0 || strcmp(ext, "png") ==
0 || strcmp(ext, "jpeg") == 0) {
            move_file(entry->d_name, "Images");
        } else if (strcmp(ext, "txt") == 0 || strcmp(ext,
"doc") == 0 || strcmp(ext, "pdf") == 0) {
            move_file(entry->d_name, "TextFiles");
        } else {
            move_file(entry->d_name, "OtherFiles");
        }
    }

    closedir(dp);
```

```
}

int main() {
    printf("Organizing files...\n");
    organize_files();
    printf("Files organized successfully!\n");
    return 0;
}
```

```
Directory 'Images' created successfully.
Directory 'TextFiles' created successfully.
Directory 'OtherFiles' created successfully.
File 'image1.jpg' moved to 'Images/image1.jpg'.
File 'document.txt' moved to 'TextFiles/document.txt'.
File 'script.py' moved to 'OtherFiles/script.py'.
Files organized successfully!
```

16. Develop a C program for implementing random access file for processing the employee details.

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

#define MAX_NAME_LENGTH 100

// Define the Employee structure
typedef struct {
    int id;
    char name[MAX_NAME_LENGTH];
    int age;
    float salary;
} Employee;

// Function to add an employee to the file
void addEmployee(FILE *file) {
    Employee emp;

    printf("Enter employee ID: ");
    scanf("%d", &emp.id);
```

```
    getchar();  // To clear the newline left by scanf

    printf("Enter employee name: ");
    fgets(emp.name, MAX_NAME_LENGTH, stdin);
    emp.name[strcspn(emp.name, "\n")] = 0;  //
Remove trailing newline

    printf("Enter employee age: ");
    scanf("%d", &emp.age);

    printf("Enter employee salary: ");
    scanf("%f", &emp.salary);

    // Move the file pointer to the end to append the
record
    fseek(file, 0, SEEK_END);

    // Write employee to the file
    fwrite(&emp, sizeof(Employee), 1, file);
    printf("Employee added successfully.\n");
}
```

```
}

// Function to retrieve an employee's details by ID
void retrieveEmployee(FILE *file, int id) {
    Employee emp;
    fseek(file, 0, SEEK_SET);

    // Read the employee records one by one
    while (fread(&emp, sizeof(Employee), 1, file)) {
        if (emp.id == id) {
            printf("\nEmployee Details:\n");
            printf("ID: %d\n", emp.id);
            printf("Name: %s\n", emp.name);
            printf("Age: %d\n", emp.age);
            printf("Salary: %.2f\n", emp.salary);
            return;
        }
    }
    printf("Employee with ID %d not found.\n", id);
}
```

```
// Function to modify an employee's details by ID
void modifyEmployee(FILE *file, int id) {
    Employee emp;
    fseek(file, 0, SEEK_SET);

    // Read the employee records one by one
    while (fread(&emp, sizeof(Employee), 1, file)) {
        if (emp.id == id) {
            printf("Employee found. Enter new
details:\n");

            printf("Enter employee name: ");
            getchar();  // To clear the newline
            fgets(emp.name, MAX_NAME_LENGTH, stdin);
            emp.name[strcspn(emp.name, "\n")] = 0;  //
Remove trailing newline

            printf("Enter employee age: ");
            scanf("%d", &emp.age);

            printf("Enter employee salary: ");
```

```
            scanf("%f", &emp.salary);

            // Move the file pointer back to the start of the
record
            long pos = ftell(file) - sizeof(Employee);
            fseek(file, pos, SEEK_SET);

            // Write the modified employee details back to
the file
            fwrite(&emp, sizeof(Employee), 1, file);
            printf("Employee details updated
successfully.\n");
            return;
        }
    }
    printf("Employee with ID %d not found.\n", id);
}

// Function to list all employees in the file
void listEmployees(FILE *file) {
    Employee emp;
```

```
    fseek(file, 0, SEEK_SET);

    printf("\nEmployee List:\n");
    // Read and print each employee record
    while (fread(&emp, sizeof(Employee), 1, file)) {
        printf("ID: %d, Name: %s, Age: %d, Salary:
%.2f\n", emp.id, emp.name, emp.age, emp.salary);
    }
}
int main() {
    FILE *file = fopen("employees.dat", "rb+");

    // If the file does not exist, create it in binary mode
    if (file == NULL) {
        file = fopen("employees.dat", "wb+");
        if (file == NULL) {
            printf("Error opening file.\n");
            return 1;
        }
    }
```

```
int choice, id;

while (1) {
    printf("\nEmployee Database System:\n");
    printf("1. Add Employee\n");
    printf("2. Retrieve Employee\n");
    printf("3. Modify Employee\n");
    printf("4. List All Employees\n");
    printf("5. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            addEmployee(file);
            break;
        case 2:
            printf("Enter employee ID to retrieve: ");
            scanf("%d", &id);
```

```
}
            retrieveEmployee(file, id);
            break;
        case 3:
            printf("Enter employee ID to modify: ");
            scanf("%d", &id);
            modifyEmployee(file, id);
            break;
        case 4:
            listEmployees(file);
            break;
        case 5:
            fclose(file);
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }
}
```

```
1. Add Employee
2. Retrieve Employee
3. Modify Employee
4. List All Employees
5. Exit
Enter your choice: 1
Enter employee ID: 1
Enter employee name: John Doe
Enter employee age: 30
Enter employee salary: 55000
Employee added successfully.

Employee Database System:
1. Add Employee
2. Retrieve Employee
3. Modify Employee
4. List All Employees
5. Exit
Enter your choice: 4

Employee List:
ID: 1, Name: John Doe, Age: 30, Salary: 55000.00

Employee Database System:
1. Add Employee
2. Retrieve Employee
3. Modify Employee
4. List All Employees
5. Exit
Enter your choice: 2
Enter employee ID to retrieve: 1

Employee Details:
ID: 1
Name: John Doe
Age: 30
Salary: 55000.00
```

17. Illustrate the deadlock avoidance concept by simulating Banker's algorithm with c

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

```
#include <stdbool.h>

#define P 5  // Number of processes
#define R 3  // Number of resources

// Function to calculate the Need matrix
void calculateNeed(int need[][R], int max[][R], int
allocation[][R]) {
    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 processes[], int avail[], int max[][R], int
allot[][R]) {
    int need[P][R];
    calculateNeed(need, max, allot);
```

```
    bool finish[P] = {0};  // Track if process is finished
    int safeSeq[P];  // Safe sequence
    int work[R];  // Available resources
    for (int i = 0; i < R; i++) {
        work[i] = avail[i];
    }

    int count = 0;  // Count of processes that can be
completed
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            // Check if process is not finished and its need
is less than or equal to work
            if (!finish[p]) {
                int j;
                for (j = 0; j < R; j++) {
                    if (need[p][j] > work[j]) {
                        break;
                    }
                }
```

```
            if (j == R) {  // Process p can be completed
                for (int k = 0; k < R; k++) {
                    work[k] += allot[p][k];  // Release the
resources of process p
                }
                safeSeq[count++] = p;  // Add process p to
safe sequence
                finish[p] = 1;
                found = true;
                break;
            }
        }
    }

    // If no process was found that can execute, then
the system is in an unsafe state
    if (!found) {
        printf("System is in an unsafe state!\n");
        return false;
    }
}
```

```
    // If all processes can be completed, print the safe
sequence
    printf("System is in a safe state.\nSafe sequence is:
");
    for (int i = 0; i < P; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf("\n");
    return true;
}

int main() {
    int processes[] = {0, 1, 2, 3, 4};  // Process IDs

    // Available instances of resources
    int avail[] = {3, 3, 2};  // Available resources (A, B, C)

    // Maximum demand matrix (Maximum resources
needed by each process)
    int max[][R] = {
```

```
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    // Allocation matrix (Resources currently allocated
to each process)
    int allot[][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    // Check if the system is in a safe state
    isSafe(processes, avail, max, allot);
```

```
    return 0;
}
```

Output:

```
System is in a safe state.
Safe sequence is: P0 P1 P3 P4 P2
```

18. Construct a C program to simulate producer-consumer problem using semaphores.

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

#define MAX_BUFFER_SIZE 5
#define NUM_ITEMS 10

// Buffer to store the items
int buffer[MAX_BUFFER_SIZE];
```

```
// Semaphores
sem_t empty, full, mutex;

// Index variables to keep track of the buffer
int in = 0, out = 0;

// Producer function
void *producer(void *param) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        // Produce an item (for simplicity, let's use the
value of i)
        int item = i;

        // Wait for an empty slot in the buffer
        sem_wait(&empty);

        // Wait for mutual exclusion before accessing the
buffer
        sem_wait(&mutex);
```

```
        // Add the item to the buffer
        buffer[in] = item;
        printf("Producer produced item %d at index
%d\n", item, in);

        // Update the index for the next item
        in = (in + 1) % MAX_BUFFER_SIZE;

        // Release mutual exclusion
        sem_post(&mutex);

        // Signal that the buffer is no longer empty
        sem_post(&full);

        // Sleep for a while to simulate time taken to
produce an item
        sleep(rand() % 2);
    }

    pthread_exit(0);
}
```

```
// Consumer function
void *consumer(void *param) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        // Wait for a full slot in the buffer
        sem_wait(&full);

        // Wait for mutual exclusion before accessing the
buffer
        sem_wait(&mutex);

        // Consume the item from the buffer
        int item = buffer[out];
        printf("Consumer consumed item %d from index
%d\n", item, out);

        // Update the index for the next item
        out = (out + 1) % MAX_BUFFER_SIZE;

        // Release mutual exclusion
        sem_post(&mutex);
```

```
        // Signal that the buffer is no longer full
        sem_post(&empty);

        // Sleep for a while to simulate time taken to
consume an item
        sleep(rand() % 2);
    }

    pthread_exit(0);
}

int main() {
    // Initialize the semaphores
    sem_init(&empty, 0, MAX_BUFFER_SIZE);  // empty
slots in the buffer
    sem_init(&full, 0, 0);                // full slots in the
buffer
    sem_init(&mutex, 0, 1);               // mutual
exclusion for buffer access
```

```
    // Create threads for producer and consumer
    pthread_t prod, cons;

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for both threads to complete
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Destroy the semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

Output

```
Producer produced item 0 at index 0
Consumer consumed item 0 from index 0
Producer produced item 1 at index 1
Consumer consumed item 1 from index 1
Producer produced item 2 at index 2
Consumer consumed item 2 from index 2
Producer produced item 3 at index 3
Consumer consumed item 3 from index 3
Producer produced item 4 at index 4
Consumer consumed item 4 from index 4
Producer produced item 5 at index 0
Consumer consumed item 5 from index 0
Producer produced item 6 at index 1
Producer produced item 7 at index 2
Consumer consumed item 6 from index 1
Producer produced item 8 at index 3
Producer produced item 9 at index 4
Consumer consumed item 7 from index 2
Consumer consumed item 8 from index 3
Consumer consumed item 9 from index 4
```

19. Design a C program to implement process synchronization using mutex locks.

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

```
#include <pthread.h>

// Define the mutex lock
pthread_mutex_t lock;

// Shared resource (Global variable)
int sharedResource = 0;

// Function to be executed by each thread
void* increment(void* arg) {
    for (int i = 0; i < 5; i++) {
        // Lock the mutex before accessing the shared
resource
        pthread_mutex_lock(&lock);

        // Critical section: accessing and modifying
shared resource
        sharedResource++;
        printf("Incremented Shared Resource: %d\n",
sharedResource);
```

```
        // Unlock the mutex after accessing the shared
resource
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

void* decrement(void* arg) {
    for (int i = 0; i < 5; i++) {
        // Lock the mutex before accessing the shared
resource
        pthread_mutex_lock(&lock);

        // Critical section: accessing and modifying
shared resource
        sharedResource--;
        printf("Decremented Shared Resource: %d\n",
sharedResource);

        // Unlock the mutex after accessing the shared
resource
```

```
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    // Initialize the mutex
    pthread_mutex_init(&lock, NULL);

    // Create two threads
    pthread_t thread1, thread2;

    // Create threads that will run the increment and
decrement functions
    pthread_create(&thread1, NULL, increment, NULL);
    pthread_create(&thread2, NULL, decrement,
NULL);

    // Wait for both threads to finish execution
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
```

```
    // Destroy the mutex after it is no longer needed
    pthread_mutex_destroy(&lock);

    // Final value of the shared resource
    printf("Final Shared Resource Value: %d\n",
sharedResource);

    return 0;
```

Output

```
Incremented Shared Resource: 1
Incremented Shared Resource: 2
Incremented Shared Resource: 3
Incremented Shared Resource: 4
Incremented Shared Resource: 5
Decremented Shared Resource: 4
Decremented Shared Resource: 3
Decremented Shared Resource: 2
Decremented Shared Resource: 1
Decremented Shared Resource: 0
Final Shared Resource Value: 0
```

20. Construct a C program to simulate Reader-Writer problem using Semaphores.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX_READERS 5 // Max number of readers
#define MAX_WRITERS 2 // Max number of writers

// Shared resource
int data = 0;

// Semaphores
sem_t mutex; // To control access to the reader count
sem_t write_sem; // To allow only one writer at a time
int read_count = 0; // Keeps track of the number of active readers

// Function for Reader Thread
```

```
void* reader(void* arg) {
    int id = *((int*)arg);

    while(1) {
        // Entry section: A reader enters the critical
section
        sem_wait(&mutex); // Lock the mutex to update
the read_count
        read_count++;
        if (read_count == 1) {
            sem_wait(&write_sem); // Block writers when
the first reader enters
        }
        sem_post(&mutex); // Unlock the mutex

        // Critical section: reading the shared resource
        printf("Reader %d: read data = %d\n", id, data);

        // Exit section: A reader exits the critical section
        sem_wait(&mutex); // Lock mutex to update the
read_count
```

```
        read_count--;
        if (read_count == 0) {
            sem_post(&write_sem); // Allow writers when
the last reader leaves
        }
        sem_post(&mutex); // Unlock the mutex

        sleep(1); // Simulate some reading time
    }

    return NULL;
}

// Function for Writer Thread
void* writer(void* arg) {
    int id = *((int*)arg);

    while(1) {
        // Entry section: A writer waits to write
        sem_wait(&write_sem); // Block other writers
and readers
```

```c
        // Critical section: writing the shared resource
        data++;
        printf("Writer %d: updated data = %d\n", id,
data);

        // Exit section: A writer exits the critical section
        sem_post(&write_sem); // Allow other writers
and readers

        sleep(2); // Simulate some writing time
    }

    return NULL;
}

int main() {
    pthread_t readers[MAX_READERS],
writers[MAX_WRITERS];
    int reader_ids[MAX_READERS],
writer_ids[MAX_WRITERS];
```

```
    // Initialize semaphores
    sem_init(&mutex, 0, 1); // Mutex for read count
(binary semaphore)
    sem_init(&write_sem, 0, 1); // Semaphore for
writer (binary semaphore)

    // Create reader threads
    for (int i = 0; i < MAX_READERS; i++) {
        reader_ids[i] = i + 1; // Assign reader ids
        pthread_create(&readers[i], NULL, reader,
&reader_ids[i]);
    }

    // Create writer threads
    for (int i = 0; i < MAX_WRITERS; i++) {
        writer_ids[i] = i + 1; // Assign writer ids
        pthread_create(&writers[i], NULL, writer,
&writer_ids[i]);
    }
```

```
    // Join reader threads
    for (int i = 0; i < MAX_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    // Join writer threads
    for (int i = 0; i < MAX_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&write_sem);

    return 0;
}
```

Output:

```
Reader 1: read data = 0
Reader 2: read data = 0
Writer 1: updated data = 1
Reader 3: read data = 1
...
```