Contents

1 Write a program using fork() system call to create two child of the same process i.e., Parent P having child process P1 and P2.

Code

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

int main()
{
    pid_t p1, p2;

    printf("Parent process (P) with PID %d is running\n", getpid()
        );

    p1 = fork();

    if (p1 < 0) {
        printf("Failed to create first child\n");
        return 1;
    }

    if (p1 == 0) {
        printf("First child (P1) with PID %d created. Parent PID:
            %d\n", getpid(), getppid());
    } else {
        p2 = fork();

        if (p2 < 0) {
            printf("Failed to create second child\n");
            return 1;
        }

        if (p2 == 0) {
            printf("Second child (P2) with PID %d created. Parent
                PID: %d\n", getpid(), getppid());
        } else {
            printf("Parent has created two children with PIDs %d
                and %d\n", p1, p2);
        }
    }

    return 0;
}
```

2 Write a program using fork() system call to create a hierarchy of 3 process such that P2 is the child of P1 and P1 is the child of P.

Code

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

int main()
{
    pid_t p1;

    printf("Original parent (P) with PID %d starting\n", getpid())
        ;

    p1 = fork();

    if (p1 < 0) {
        printf("Failed to create child process\n");
        return 1;
    }

    if (p1 == 0) {
        printf("First child (P1) with PID %d created. Parent PID:
            %d\n", getpid(), getppid());

        pid_t p2 = fork();

        if (p2 < 0) {
            printf("Failed to create second child process\n");
            return 1;
        }

        if (p2 == 0) {
            printf("Second child (P2) with PID %d created. Parent
                PID: %d\n", getpid(), getppid());
        } else {
            printf("P1 (PID: %d) created child P2 with PID: %d\n",
                getpid(), p2);
        }
    } else {
        printf("Parent (P) with PID %d created child P1 with PID:
            %d\n", getpid(), p1);
    }

    return 0;
}
```

3 Create a parent-child relationship between two processes.

Code

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

int main()
{
    pid_t child_pid;

    printf("Parent (P) is having ID %d\n", getpid());

    child_pid = fork();

    if (child_pid < 0) {
        printf("Fork failed\n");
        return 1;
    }

    if (child_pid == 0) {
        printf("Child is having ID %d\n", getpid());

        printf("My Parent ID is %d\n", getppid());
    } else {
        wait(NULL);

        printf("ID of P's Child is %d\n", child_pid);
    }

    return 0;
}
```

4 Write a program to create two child process. The parent process should wait for both the child to finish.

Code

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

int main()
{
    pid_t child1_pid, child2_pid;

    printf("Parent process starting with PID: %d\n", getpid());

    child1_pid = fork();

    if (child1_pid < 0) {
        printf("First fork failed\n");
        return 1;
    }

    if (child1_pid == 0) {
        printf("Child 1 executing with PID: %d\n", getpid());
        sleep(2);
        printf("Child 1 (PID: %d) finishing\n", getpid());
        return 0;
    }

    child2_pid = fork();

    if (child2_pid < 0) {
        printf("Second fork failed\n");
        return 1;
    }

    if (child2_pid == 0) {
        printf("Child 2 executing with PID: %d\n", getpid());
        sleep(4);
        printf("Child 2 (PID: %d) finishing\n", getpid());
        return 0;
    }

    printf("Parent waiting for child 1 (PID: %d) to finish\n",
        child1_pid);
    wait(NULL);
    printf("Child 1 has finished\n");

    printf("Parent waiting for child 2 (PID: %d) to finish\n",
        child2_pid);
```

```
    wait(NULL);
    printf("Child 2 has finished\n");

    printf("Parent process (PID: %d) exiting\n", getpid());

    return 0;
}
```

Commands

```
gcc program_name.c
./a.out

(or)

gcc program_name.c -o program_name
./program_name

if "./program_name" says "permission denied" then do this

chmod +x program_name
```

5 Create two child process C1 and C2. Make sure that only C2 becomes an orphan process.

Code

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

int main()
{
    pid_t child1_pid, child2_pid;

    printf("Parent process starting with PID: %d\n", getpid());

    child1_pid = fork();

    if (child1_pid < 0) {
        printf("First fork failed\n");
        return 1;
    }

    if (child1_pid == 0) {
        printf("Child C1 executing with PID: %d\n", getpid());
```

```
        printf("Child C1 (PID: %d) finishing\n", getpid());
        return 0;
    }

    child2_pid = fork();

    if (child2_pid < 0) {
        printf("Second fork failed\n");
        return 1;
    }

    if (child2_pid == 0) {
        printf("Child C2 with PID %d starting, parent PID: %d\n",
            getpid(), getppid());
        sleep(5);
        printf("Child C2 with PID %d now has parent PID: %d\n",
            getpid(), getppid());
        printf("If parent PID changed to 1 (or another low number)
            , C2 is now an orphan\n");
        return 0;
    }

    wait(NULL);
    printf("Child C1 (PID: %d) has finished\n", child1_pid);

    printf("Parent (PID: %d) exiting without waiting for C2 (PID:
        %d)\n", getpid(), child2_pid);

    return 0;
}
```

Commands

```
# Compile program_name.c
gcc program_name.c -o program_name

# Run the executable
./program_name

# In another terminal window, you can check if process C2 became
    an orphan:
ps -elf | grep program_name
```

6 Create two child process C1 and C2. Make sure that only C2 becomes a zombie process.

Code

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

int main()
{
    pid_t child1_pid, child2_pid;

    printf("Parent process starting with PID: %d\n", getpid());

    child1_pid = fork();

    if (child1_pid < 0) {
        printf("First fork failed\n");
        return 1;
    }

    if (child1_pid == 0) {
        printf("Child C1 executing with PID: %d\n", getpid());
        printf("Child C1 (PID: %d) finishing\n", getpid());
        return 0;
    }

    child2_pid = fork();

    if (child2_pid < 0) {
        printf("Second fork failed\n");
        return 1;
    }

    if (child2_pid == 0) {
        printf("Child C2 with PID %d starting\n", getpid());
        printf("Child C2 with PID %d finishing quickly to become
            zombie\n", getpid());
        return 0;
    }

    wait(NULL);
    printf("Child C1 (PID: %d) has finished\n", child1_pid);

    printf("Parent sleeping for 10 seconds. During this time, C2 (
        PID: %d) will be a zombie.\n", child2_pid);
    printf("Run 'ps -l' in another terminal to see the zombie
        process (marked with Z)\n");
    sleep(10);

    printf("Parent process (PID: %d) exiting\n", getpid());
```

```
    return 0;
}
```

Commands

```
# Compile program_name.c
gcc program_name.c -o program_name

# Run the executable
./program_name

# In another terminal window, while program_name is sleeping,
   check for the zombie process:
ps -l | grep defunct

# Or more specifically:
ps aux | grep program_name
# Look for a process with 'Z' in the status column
```

7 FCFS Scheduling (First-Come, First-Served)

Code: FCFS Scheduling Algorithm

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

struct process {
    int pno, at, bt, ct, tat, wt, rt, priority;
};

void print_table(struct process* p[], int n) {
    printf("%-15s%-15s%-15s%-18s%-20s%-15s%-17s%-10s\n",
           "Process No", "Arrival Time", "Burst Time", "Completion
               Time",
           "Turnaround Time", "Waiting Time", "Response Time", "
              Priority");

    for (int i = 0; i < n; i++) {
        printf("%-15d%-15d%-15d%-18d%-20d%-15d%-17d%-10d\n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct,
               p[i]->tat, p[i]->wt, p[i]->rt, p[i]->priority);
    }
}

void fcfs(struct process* p[], int n) {
    int st = 0;
```

```
    float awt = 0, atat = 0;
    char schedule[1000] = "";
    char temp[100];

    if (p[0]->at > 0) {
        sprintf(temp, "stall(0, %d) --> ", p[0]->at);
        strcat(schedule, temp);
        st = p[0]->at;
    }

    for (int i = 0; i < n; i++) {
        if (st < p[i]->at)
            st = p[i]->at;
        p[i]->rt = st - p[i]->at;
        st += p[i]->bt;
        p[i]->ct = st;
        p[i]->tat = p[i]->ct - p[i]->at;
        p[i]->wt = p[i]->tat - p[i]->bt;
        awt += (float)p[i]->wt;
        atat += (float)p[i]->tat;

        sprintf(temp, "P%d(%d, %d) --> ", p[i]->pno, p[i]->ct - p[
           i]->bt, p[i]->ct);
        strcat(schedule, temp);
    }

    printf("\n--------------- FCFS Scheduling ---------------\n")

    print_table(p, n);
    printf("\nSchedule: %s//\n", schedule);
    awt = awt / (float)n;
    atat = atat / (float)n;
    printf("Average Wait Time: %.3f\n", awt);
    printf("Average Turn Around Time: %.3f", atat);
}

void merge(struct process* p[], int low, int mid, int high) {
    int i = low, j = mid + 1, k = 0;
    struct process* temp[high - low + 1];

    while (i <= mid && j <= high) {
        if (p[i]->at <= p[j]->at)
            temp[k++] = p[i++];
        else
            temp[k++] = p[j++];
    }

    while (i <= mid)
        temp[k++] = p[i++];
    while (j <= high)
        temp[k++] = p[j++];

    for (i = low, k = 0; i <= high; i++, k++)
        p[i] = temp[k];
```

```
}

void sort(struct process* p[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        sort(p, low, mid);
        sort(p, mid + 1, high);
        merge(p, low, mid, high);
    }
}

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

    struct process* p[n];
    printf("Enter (arrival time, burst time) for each process:\n")
        ;
    for (int i = 0; i < n; i++) {
        int at, bt;
        scanf("%d%d", &at, &bt);

        p[i] = (struct process*)malloc(sizeof(struct process));
        p[i]->pno = i + 1;
        p[i]->at = at;
        p[i]->bt = bt;
        p[i]->priority = 0;
    }

    sort(p, 0, n - 1);
    fcfs(p, n);

    for (int i = 0; i < n; i++)
        free(p[i]);

    return 0;
}
```

8 SJF Scheduling (Shortest Job First – Non-preemptive)

Code: SJF Scheduling Algorithm (Non-preemptive)

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

```c
struct process {
    int pno, at, bt, ct, tat, wt, rt, priority;
    int done;
};

void print_table(struct process* p[], int n) {
    printf("%-15s%-15s%-15s%-18s%-20s%-15s%-17s%-10s\n",
           "Process No", "Arrival Time", "Burst Time", "Completion
               Time",
           "Turnaround Time", "Waiting Time", "Response Time", "
              Priority");

    for (int i = 0; i < n; i++) {
        printf("%-15d%-15d%-15d%-18d%-20d%-15d%-17d%-10d\n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct,
               p[i]->tat, p[i]->wt, p[i]->rt, p[i]->priority);
    }
}

void sjf(struct process* p[], int n) {
    int completed = 0, current_time = 0;
    float awt = 0, atat = 0;
    char schedule[1000] = "";
    char temp[100];

    for (int i = 0; i < n; i++)
        p[i]->done = 0;

    while (completed < n) {
        int idx = -1;
        int min_bt = 1e9;

        for (int i = 0; i < n; i++) {
            if (!p[i]->done && p[i]->at <= current_time && p[i]->
                bt < min_bt) {
                min_bt = p[i]->bt;
                idx = i;
            }
        }

        if (idx == -1) {
            int next_arrival = 1e9;
            for (int i = 0; i < n; i++)
                if (!p[i]->done && p[i]->at < next_arrival)
                    next_arrival = p[i]->at;

            sprintf(temp, "stall(%d, %d) --> ", current_time,
                next_arrival);
            strcat(schedule, temp);
            current_time = next_arrival;
            continue;
        }

        struct process* curr = p[idx];
```

```
            curr->rt = current_time - curr->at;
            current_time += curr->bt;
            curr->ct = current_time;
            curr->tat = curr->ct - curr->at;
            curr->wt = curr->tat - curr->bt;
            curr->done = 1;
            completed++;

            awt += (float)curr->wt;
            atat += (float)curr->tat;

            sprintf(temp, "P%d(%d, %d) --> ", curr->pno, curr->ct -
                curr->bt, curr->ct);
            strcat(schedule, temp);
        }

        printf("\n---------------- SJF (Non-preemptive) Scheduling
            ----------------\n");

        print_table(p, n);
        printf("\nSchedule: %s//\n", schedule);
        awt = awt / (float)n;
        atat = atat / (float)n;
        printf("Average Wait Time: %.3f\n", awt);
        printf("Average Turn Around Time: %.3f", atat);
    }

    void merge(struct process* p[], int low, int mid, int high) {
        int i = low, j = mid + 1, k = 0;
        struct process* temp[high - low + 1];

        while (i <= mid && j <= high) {
            if (p[i]->at <= p[j]->at)
                temp[k++] = p[i++];
            else
                temp[k++] = p[j++];
        }

        while (i <= mid)
            temp[k++] = p[i++];
        while (j <= high)
            temp[k++] = p[j++];

        for (i = low, k = 0; i <= high; i++, k++)
            p[i] = temp[k];
    }

    void sort(struct process* p[], int low, int high) {
        if (low < high) {
            int mid = (low + high) / 2;
            sort(p, low, mid);
            sort(p, mid + 1, high);
            merge(p, low, mid, high);
        }
```

```
}

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

    struct process* p[n];
    printf("Enter (arrival time, burst time) for each process:\n")
        ;
    for (int i = 0; i < n; i++) {
        int at, bt;
        scanf("%d%d", &at, &bt);

        p[i] = (struct process*)malloc(sizeof(struct process));
        p[i]->pno = i + 1;
        p[i]->at = at;
        p[i]->bt = bt;
        p[i]->priority = 0;
    }

    sort(p, 0, n - 1);
    sjf(p, n);

    for (int i = 0; i < n; i++)
        free(p[i]);

    return 0;
}
```

9 SRTF Scheduling (Shortest Remaining Time First – Preemptive SJF)

Code: SRTF Scheduling Algorithm (Preemptive)

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

struct process {
    int pno, at, bt, ct, tat, wt, rt, priority;
    int remaining_time;
    int started;
    int completed;
};

void print_table(struct process* p[], int n) {
```

```
    printf("%-15s%-15s%-15s%-18s%-20s%-15s%-17s%-10s\n",
           "Process No", "Arrival Time", "Burst Time", "Completion
               Time",
           "Turnaround Time", "Waiting Time", "Response Time", "
              Priority");

    for (int i = 0; i < n; i++) {
        printf("%-15d%-15d%-15d%-18d%-20d%-15d%-17d%-10d\n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct,
               p[i]->tat, p[i]->wt, p[i]->rt, p[i]->priority);
    }
}

void sjf_preemptive(struct process* p[], int n) {
    int completed = 0, current_time = 0, prev = -1;
    float awt = 0, atat = 0;
    char schedule[1000] = "";
    char temp[100];

    for (int i = 0; i < n; i++) {
        p[i]->remaining_time = p[i]->bt;
        p[i]->started = 0;
        p[i]->completed = 0;
    }

    while (completed < n) {
        int idx = -1;
        int min_remaining = 1e9;

        for (int i = 0; i < n; i++) {
            if (!p[i]->completed && p[i]->at <= current_time && p[
               i]->remaining_time < min_remaining && p[i]->
               remaining_time > 0) {
                min_remaining = p[i]->remaining_time;
                idx = i;
            }
        }

        if (idx == -1) {
            current_time++;
            continue;
        }

        if (!p[idx]->started) {
            p[idx]->rt = current_time - p[idx]->at;
            p[idx]->started = 1;
        }

        if (prev != idx) {
            if (prev != -1)
                sprintf(temp, "%d) ", current_time);
            sprintf(temp + strlen(temp), "P%d(%d, ", p[idx]->pno,
               current_time);
            strcat(schedule, temp);
```

```
            }

            p[idx]->remaining_time--;
            current_time++;
            prev = idx;

            if (p[idx]->remaining_time == 0) {
                p[idx]->ct = current_time;
                p[idx]->tat = p[idx]->ct - p[idx]->at;
                p[idx]->wt = p[idx]->tat - p[idx]->bt;
                awt += p[idx]->wt;
                atat += p[idx]->tat;
                p[idx]->completed = 1;

                sprintf(temp, "%d) --> ", current_time);
                strcat(schedule, temp);
                prev = -1;
                completed++;
            }
        }

        printf("\n--------------- SJF (Preemptive) Scheduling
           ---------------\n");
        print_table(p, n);
        printf("\nSchedule: %s//\n", schedule);
        printf("Average Wait Time: %.3f\n", awt / n);
        printf("Average Turn Around Time: %.3f\n", atat / n);
    }

    void merge(struct process* p[], int low, int mid, int high) {
        int i = low, j = mid + 1, k = 0;
        struct process* temp[high - low + 1];

        while (i <= mid && j <= high) {
            if (p[i]->at <= p[j]->at)
                temp[k++] = p[i++];
            else
                temp[k++] = p[j++];
        }

        while (i <= mid)
            temp[k++] = p[i++];
        while (j <= high)
            temp[k++] = p[j++];

        for (i = low, k = 0; i <= high; i++, k++)
            p[i] = temp[k];
    }

    void sort(struct process* p[], int low, int high) {
        if (low < high) {
            int mid = (low + high) / 2;
            sort(p, low, mid);
            sort(p, mid + 1, high);
```

```
            merge(p, low, mid, high);
        }
    }

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

        struct process* p[n];
        printf("Enter (arrival time, burst time) for each process:\n")
            ;
        for (int i = 0; i < n; i++) {
            int at, bt;
            scanf("%d%d", &at, &bt);

            p[i] = (struct process*)malloc(sizeof(struct process));
            p[i]->pno = i + 1;
            p[i]->at = at;
            p[i]->bt = bt;
            p[i]->priority = 0;
        }

        sort(p, 0, n - 1);
        sjf_preemptive(p, n);

        for (int i = 0; i < n; i++)
            free(p[i]);

        return 0;
    }
```

10 RR Scheduling (Round Robin – Preemptive)

Code: RR Scheduling Algorithm (Preemptive)

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

struct process {
    int pno, at, bt, ct, tat, wt, rt, priority;
    int remaining_time;
    int started;
    int completed;
};
```

```
void print_table(struct process* p[], int n) {
    printf("%-15s%-15s%-15s%-18s%-20s%-15s%-17s%-10s\n",
           "Process No", "Arrival Time", "Burst Time", "Completion
               Time",
           "Turnaround Time", "Waiting Time", "Response Time", "
              Priority");

    for (int i = 0; i < n; i++) {
        printf("%-15d%-15d%-15d%-18d%-20d%-15d%-17d%-10d\n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct,
               p[i]->tat, p[i]->wt, p[i]->rt, p[i]->priority);
    }
}

void round_robin(struct process* p[], int n, int tq) {
    int current_time = 0, completed = 0;
    float awt = 0, atat = 0;
    char schedule[1000] = "";
    char temp[100];

    for (int i = 0; i < n; i++) {
        p[i]->remaining_time = p[i]->bt;
        p[i]->started = 0;
        p[i]->completed = 0;
    }

    int queue[1000];
    int front = 0, rear = 0;
    int visited[n];
    memset(visited, 0, sizeof(visited));

    queue[rear++] = 0;
    visited[0] = 1;

    while (completed < n) {
        if (front == rear) {
            current_time++;
            for (int i = 0; i < n; i++) {
                if (!visited[i] && p[i]->at <= current_time) {
                    queue[rear++] = i;
                    visited[i] = 1;
                }
            }
            continue;
        }

        int idx = queue[front++];

        if (!p[idx]->started) {
            p[idx]->rt = current_time - p[idx]->at;
            p[idx]->started = 1;
        }

        int exec_time = (p[idx]->remaining_time >= tq) ? tq : p[
```

```
            idx]->remaining_time;
        sprintf(temp, "P%d(%d, ", p[idx]->pno, current_time);
        strcat(schedule, temp);

        current_time += exec_time;
        p[idx]->remaining_time -= exec_time;

        sprintf(temp, "%d) --> ", current_time);
        strcat(schedule, temp);

        for (int i = 0; i < n; i++) {
            if (!visited[i] && p[i]->at <= current_time) {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }

        if (p[idx]->remaining_time == 0) {
            p[idx]->ct = current_time;
            p[idx]->tat = p[idx]->ct - p[idx]->at;
            p[idx]->wt = p[idx]->tat - p[idx]->bt;
            awt += p[idx]->wt;
            atat += p[idx]->tat;
            p[idx]->completed = 1;
            completed++;
        } else {
            queue[rear++] = idx;
        }
    }

    printf("\n--------------- Round Robin Scheduling (TQ = %d)
       ---------------\n", tq);
    print_table(p, n);
    printf("\nSchedule: %s//\n", schedule);
    printf("Average Wait Time: %.3f\n", awt / n);
    printf("Average Turn Around Time: %.3f\n", atat / n);
}

void merge(struct process* p[], int low, int mid, int high) {
    int i = low, j = mid + 1, k = 0;
    struct process* temp[high - low + 1];

    while (i <= mid && j <= high) {
        if (p[i]->at <= p[j]->at)
            temp[k++] = p[i++];
        else
            temp[k++] = p[j++];
    }

    while (i <= mid)
        temp[k++] = p[i++];
    while (j <= high)
        temp[k++] = p[j++];
```

```
    for (i = low, k = 0; i <= high; i++, k++)
        p[i] = temp[k];
}

void sort(struct process* p[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        sort(p, low, mid);
        sort(p, mid + 1, high);
        merge(p, low, mid, high);
    }
}

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

    struct process* p[n];
    printf("Enter (arrival time, burst time) for each process:\n")
        ;
    for (int i = 0; i < n; i++) {
        int at, bt;
        scanf("%d%d", &at, &bt);

        p[i] = (struct process*)malloc(sizeof(struct process));
        p[i]->pno = i + 1;
        p[i]->at = at;
        p[i]->bt = bt;
        p[i]->priority = 0;
    }

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

    sort(p, 0, n - 1);
    round_robin(p, n, tq);

    for (int i = 0; i < n; i++)
        free(p[i]);

    return 0;
}
```

11 LJF Scheduling (Longest Job First – Non-preemptive)

Code: LJF Scheduling Algorithm (Non-preemptive)

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

struct process {
    int pno, at, bt, ct, tat, wt, rt, priority;
    int done;
};

void print_table(struct process* p[], int n) {
    printf("\n%-15s%-15s%-15s%-18s%-20s%-15s%-17s%-10s\n",
           "Process No", "Arrival Time", "Burst Time", "Completion
               Time",
           "Turnaround Time", "Waiting Time", "Response Time", "
              Priority");

    for (int i = 0; i < n; i++) {
        printf("%-15d%-15d%-15d%-18d%-20d%-15d%-17d%-10d\n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct,
               p[i]->tat, p[i]->wt, p[i]->rt, p[i]->priority);
    }
}

void ljf(struct process* p[], int n) {
    int completed = 0, current_time = 0;
    float awt = 0, atat = 0;
    char schedule[1000] = "";
    char temp[100];

    for (int i = 0; i < n; i++)
        p[i]->done = 0;

    while (completed < n) {
        int idx = -1;
        int max_bt = -1;

        for (int i = 0; i < n; i++) {
            if (!p[i]->done && p[i]->at <= current_time && p[i]->
               bt > max_bt) {
                max_bt = p[i]->bt;
                idx = i;
            }
        }

        if (idx == -1) {
            int next_arrival = 1e9;
            for (int i = 0; i < n; i++)
                if (!p[i]->done && p[i]->at < next_arrival)
                    next_arrival = p[i]->at;
```

```
            sprintf(temp, "stall(%d, %d) --> ", current_time,
                next_arrival);
            strcat(schedule, temp);
            current_time = next_arrival;
            continue;
        }

        struct process* curr = p[idx];
        curr->rt = current_time - curr->at;
        current_time += curr->bt;
        curr->ct = current_time;
        curr->tat = curr->ct - curr->at;
        curr->wt = curr->tat - curr->bt;
        curr->done = 1;
        completed++;

        awt += curr->wt;
        atat += curr->tat;

        sprintf(temp, "P%d(%d, %d) --> ", curr->pno, curr->ct -
            curr->bt, curr->ct);
        strcat(schedule, temp);
    }

    printf("\n--------------- Longest Job First (Non-preemptive)
        Scheduling ---------------\n");
    print_table(p, n);

    printf("\nSchedule: %s//\n", schedule);
    printf("Average Wait Time: %.3f\n", awt / n);
    printf("Average Turn Around Time: %.3f\n", atat / n);
}

void merge(struct process* p[], int low, int mid, int high) {
    int i = low, j = mid + 1, k = 0;
    struct process* temp[high - low + 1];

    while (i <= mid && j <= high) {
        if (p[i]->at <= p[j]->at)
            temp[k++] = p[i++];
        else
            temp[k++] = p[j++];
    }

    while (i <= mid)
        temp[k++] = p[i++];
    while (j <= high)
        temp[k++] = p[j++];

    for (i = low, k = 0; i <= high; i++, k++)
        p[i] = temp[k];
}
```

```c
void sort(struct process* p[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        sort(p, low, mid);
        sort(p, mid + 1, high);
        merge(p, low, mid, high);
    }
}

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

    struct process* p[n];
    printf("Enter (arrival time, burst time) for each process:\n")
        ;
    for (int i = 0; i < n; i++) {
        int at, bt;
        scanf("%d%d", &at, &bt);

        p[i] = (struct process*)malloc(sizeof(struct process));
        p[i]->pno = i + 1;
        p[i]->at = at;
        p[i]->bt = bt;
        p[i]->priority = 0;
    }

    sort(p, 0, n - 1);
    ljf(p, n);

    for (int i = 0; i < n; i++)
        free(p[i]);

    return 0;
}
```

12 Priority Scheduling (Both Preemptive and Non-preemptive)

Code: Priority Scheduling Algorithm (Preemptive and Non-preemptive)

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

struct process {
```

```
    int pno, at, bt, ct, tat, wt, rt, priority;
    int done, remaining_bt, started;
};

int is_higher_better;

void print_table(struct process* p[], int n) {
    printf("Process Number\tArrival Time\tBurst Time\tCompletion
       Time\tTurn Around Time\tWait Time\tResponse Time\tPriority\
       n");
    for (int i = 0; i < n; i++) {
        printf("   %d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\t\t%d\t\t%d\
           n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct, p[i]->tat,
                   p[i]->wt, p[i]->rt, p[i]->priority);
    }
}

int compare_priority(int a, int b) {
    return is_higher_better ? (a > b) : (a < b);
}

void priority_non_preemptive(struct process* p[], int n) {
    int current_time = 0, completed = 0;
    char schedule[1000] = "";
    char temp[100];
    float awt = 0, atat = 0;

    for (int i = 0; i < n; i++) p[i]->done = 0;

    while (completed < n) {
        int idx = -1;

        for (int i = 0; i < n; i++) {
            if (!p[i]->done && p[i]->at <= current_time) {
                if (idx == -1 || compare_priority(p[i]->priority,
                   p[idx]->priority) ||
                   (p[i]->priority == p[idx]->priority && p[i]->at
                       < p[idx]->at)) {
                    idx = i;
                }
            }
        }

        if (idx == -1) {
            current_time++;
            continue;
        }

        struct process* curr = p[idx];
        curr->rt = current_time - curr->at;
        current_time += curr->bt;
        curr->ct = current_time;
        curr->tat = curr->ct - curr->at;
```

```
            curr->wt = curr->tat - curr->bt;
            curr->done = 1;
            awt += curr->wt;
            atat += curr->tat;

            sprintf(temp, "P%d(%d, %d) --> ", curr->pno, curr->ct -
                curr->bt, curr->ct);
            strcat(schedule, temp);
            completed++;
        }

        printf("\n---------------- Priority Scheduling (Non-Preemptive)
            ----------------\n");
        print_table(p, n);
        printf("\nSchedule: %s//\n", schedule);
        printf("Average Wait Time: %.3f\n", awt / n);
        printf("Average Turn Around Time: %.3f\n", atat / n);
    }

    void priority_preemptive(struct process* p[], int n) {
        int current_time = 0, completed = 0, last = -1;
        char schedule[10000] = "";
        char temp[100];
        float awt = 0, atat = 0;

        for (int i = 0; i < n; i++) {
            p[i]->remaining_bt = p[i]->bt;
            p[i]->done = 0;
            p[i]->started = 0;
        }

        while (completed < n) {
            int idx = -1;

            for (int i = 0; i < n; i++) {
                if (p[i]->at <= current_time && !p[i]->done) {
                    if (idx == -1 || compare_priority(p[i]->priority,
                        p[idx]->priority) ||
                        (p[i]->priority == p[idx]->priority && p[i]->at
                            < p[idx]->at)) {
                         idx = i;
                    }
                }
            }

            if (idx == -1) {
                current_time++;
                continue;
            }

            struct process* curr = p[idx];

            if (!curr->started) {
                curr->rt = current_time - curr->at;
```

```
            curr->started = 1;
        }

        if (last != curr->pno) {
            if (last != -1) strcat(schedule, ") --> ");
            sprintf(temp, "P%d(%d", curr->pno, current_time);
            strcat(schedule, temp);
        }

        curr->remaining_bt--;
        current_time++;
        last = curr->pno;

        if (curr->remaining_bt == 0) {
            curr->ct = current_time;
            curr->tat = curr->ct - curr->at;
            curr->wt = curr->tat - curr->bt;
            curr->done = 1;
            completed++;
            awt += curr->wt;
            atat += curr->tat;
        }
    }

    strcat(schedule, ", ");
    sprintf(temp, "%d) --> //", current_time);
    strcat(schedule, temp);

    printf("\n--------------- Priority Scheduling (Preemptive)
       ---------------\n");
    print_table(p, n);
    printf("\nSchedule: %s\n", schedule);
    printf("Average Wait Time: %.3f\n", awt / n);
    printf("Average Turn Around Time: %.3f\n", atat / n);
}

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

    printf("Does a higher number mean a higher priority? (1 for
       Yes, 0 for No): ");
    scanf("%d", &is_higher_better);

    struct process* p[n];
    printf("Enter (arrival time, burst time, priority) for each
       process:\n");
    for (int i = 0; i < n; i++) {
        int at, bt, pr;
        scanf("%d%d%d", &at, &bt, &pr);
        p[i] = (struct process*)malloc(sizeof(struct process));
        p[i]->pno = i + 1;
        p[i]->at = at;
```

```
        p[i]->bt = bt;
        p[i]->priority = pr;
    }

    struct process* copy1[n], *copy2[n];
    for (int i = 0; i < n; i++) {
        copy1[i] = (struct process*)malloc(sizeof(struct process))
            ;
        copy2[i] = (struct process*)malloc(sizeof(struct process))
            ;
        *copy1[i] = *p[i];
        *copy2[i] = *p[i];
    }

    priority_non_preemptive(copy1, n);
    priority_preemptive(copy2, n);

    for (int i = 0; i < n; i++) {
        free(p[i]);
        free(copy1[i]);
        free(copy2[i]);
    }

    return 0;
}
```

13 HRRN Scheduling (Highest Response Ratio Next - Preemptive)

Code: HRRN Scheduling Algorithm (Preemptive)

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

struct process {
    int pno, at, bt, ct, tat, wt, rt;
    int done;
};

void print_table(struct process* p[], int n) {
    printf("Process Number\tArrival Time\tBurst Time\tCompletion
        Time\tTurn Around Time\tWait Time\tResponse Time\n");
    for (int i = 0; i < n; i++) {
        printf("   %d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\t\t%d\n",
               p[i]->pno, p[i]->at, p[i]->bt, p[i]->ct, p[i]->tat,
                   p[i]->wt, p[i]->rt);
```

```
    }
}

void hrrn_scheduling(struct process* p[], int n) {
    int current_time = 0, completed = 0;
    char schedule[1000] = "";
    char temp[100];
    float awt = 0, atat = 0;

    for (int i = 0; i < n; i++) p[i]->done = 0;

    while (completed < n) {
        int idx = -1;
        float max_hrr = -1.0;

        for (int i = 0; i < n; i++) {
            if (!p[i]->done && p[i]->at <= current_time) {
                float hrr = (float)(current_time - p[i]->at + p[i
                    ]->bt) / p[i]->bt;
                if (hrr > max_hrr) {
                    max_hrr = hrr;
                    idx = i;
                }
            }
        }

        if (idx == -1) {
            current_time++;
            continue;
        }

        struct process* curr = p[idx];
        curr->rt = current_time - curr->at;
        current_time += curr->bt;
        curr->ct = current_time;
        curr->tat = curr->ct - curr->at;
        curr->wt = curr->tat - curr->bt;
        curr->done = 1;
        awt += curr->wt;
        atat += curr->tat;

        sprintf(temp, "P%d(%d, %d) --> ", curr->pno, curr->ct -
            curr->bt, curr->ct);
        strcat(schedule, temp);
        completed++;
    }

    printf("\n--------------- Highest Response Ratio Next (HRRN)
        Scheduling ---------------\n");
    print_table(p, n);
    printf("\nSchedule: %s//\n", schedule);
    printf("Average Wait Time: %.3f\n", awt / n);
    printf("Average Turn Around Time: %.3f\n", atat / n);
}
```

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

    struct process* p[n];
    printf("Enter (arrival time, burst time) for each process:\n")
        ;
    for (int i = 0; i < n; i++) {
        int at, bt;
        scanf("%d%d", &at, &bt);
        p[i] = (struct process*)malloc(sizeof(struct process));
        p[i]->pno = i + 1;
        p[i]->at = at;
        p[i]->bt = bt;
    }

    struct process* copy[n];
    for (int i = 0; i < n; i++) {
        copy[i] = (struct process*)malloc(sizeof(struct process));
        *copy[i] = *p[i];
    }

    hrrn_scheduling(copy, n);

    for (int i = 0; i < n; i++) {
        free(p[i]);
        free(copy[i]);
    }

    return 0;
}
```

Commands

```
gcc program_name.c
./a.out

(or)

gcc program_name.c -o program_name
./program_name

if "./program_name" says "permission denied" then do this

chmod +x program_name
```

14 Write a program to create two Threads T1 and T2. Thread T1 creates a file named Thread.txt while T2 writes "Hello its T2" into the Thread.txt

Code

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

void *thread_T1_function(void *arg);
void *thread_T2_function(void *arg);

pthread_mutex_t file_mutex;

int main() {
    pthread_t T1, T2;

    pthread_mutex_init(&file_mutex, NULL);

    pthread_create(&T1, NULL, thread_T1_function, NULL);
    sleep(1);
    pthread_create(&T2, NULL, thread_T2_function, NULL);

    pthread_join(T1, NULL);
    pthread_join(T2, NULL);

    pthread_mutex_destroy(&file_mutex);

    return 0;
}

void *thread_T1_function(void *arg) {
    pthread_mutex_lock(&file_mutex);
    FILE *file = fopen("Thread.txt", "w");
    if (file == NULL) {
        printf("Error creating file\n");
    } else {
        printf("Thread T1: File created successfully\n");
        fclose(file);
    }
    pthread_mutex_unlock(&file_mutex);
    pthread_exit(NULL);
}

void *thread_T2_function(void *arg) {
    pthread_mutex_lock(&file_mutex);
    FILE *file = fopen("Thread.txt", "w");
    if (file == NULL) {
        printf("Error opening file\n");
    } else {
```

```
        fprintf(file, "Hello its T2");
        printf("Thread T2: Data written to file\n");
        fclose(file);
    }
    pthread_mutex_unlock(&file_mutex);
    pthread_exit(NULL);
}
```

15 Write a program to create a thread T1. The main process passes two numbers to T1. T1 calculatesthe sum of these numbers and returns the sum to the parent process for printing.

Code

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

struct numbers {
    int num1;
    int num2;
};

void *thread_T1_function(void *arg);

int main() {
    pthread_t T1;
    void *result;
    struct numbers nums;

    nums.num1 = 10;
    nums.num2 = 20;

    pthread_create(&T1, NULL, thread_T1_function, &nums);
    pthread_join(T1, &result);

    printf("Sum calculated by thread T1: %d\n", *((int *)result));

    free(result);

    return 0;
}

void *thread_T1_function(void *arg) {
```

```
    struct numbers *nums = (struct numbers *)arg;
    int *sum = malloc(sizeof(int));

    *sum = nums->num1 + nums->num2;

    pthread_exit(sum);
}
```

Commands

```
gcc program_name.c -lpthread
```

16 Write a program to simulate Multilevel Feedback Queue CPU scheduling algorithms to find average turnaround time and average waiting time, where queues 1 and 2 follow round robin with time quantum 4 and 8, respectively and queue 3 follow FCFS.

Code

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

struct process {
    int pno, at, bt, ct, tat, wt, rt;
    int remaining_bt, queue_level, started;
};

int all_done(struct process* p[], int n) {
    for (int i = 0; i < n; i++) {
        if (p[i]->remaining_bt > 0)
            return 0;
    }
    return 1;
}

void multilevel_feedback_queue(struct process* p[], int n) {
    int current_time = 0, completed = 0;
    float awt = 0, atat = 0;
    int quantum1 = 4, quantum2 = 8;
```

```
    while (!all_done(p, n)) {
        int executed = 0;

        for (int i = 0; i < n; i++) {
            if (p[i]->remaining_bt > 0 && p[i]->queue_level == 1
               && p[i]->at <= current_time) {
                if (!p[i]->started) {
                    p[i]->rt = current_time - p[i]->at;
                    p[i]->started = 1;
                }
                int exec_time = (p[i]->remaining_bt > quantum1) ?
                   quantum1 : p[i]->remaining_bt;
                current_time += exec_time;
                p[i]->remaining_bt -= exec_time;

                if (p[i]->remaining_bt == 0) {
                    p[i]->ct = current_time;
                    p[i]->tat = p[i]->ct - p[i]->at;
                    p[i]->wt = p[i]->tat - p[i]->bt;
                    awt += p[i]->wt;
                    atat += p[i]->tat;
                } else {
                    p[i]->queue_level = 2;
                }

                executed = 1;
            }
        }

        for (int i = 0; i < n; i++) {
            if (p[i]->remaining_bt > 0 && p[i]->queue_level == 2
               && p[i]->at <= current_time) {
                if (!p[i]->started) {
                    p[i]->rt = current_time - p[i]->at;
                    p[i]->started = 1;
                }
                int exec_time = (p[i]->remaining_bt > quantum2) ?
                   quantum2 : p[i]->remaining_bt;
                current_time += exec_time;
                p[i]->remaining_bt -= exec_time;

                if (p[i]->remaining_bt == 0) {
                    p[i]->ct = current_time;
                    p[i]->tat = p[i]->ct - p[i]->at;
                    p[i]->wt = p[i]->tat - p[i]->bt;
                    awt += p[i]->wt;
                    atat += p[i]->tat;
                } else {
                    p[i]->queue_level = 3;
                }

                executed = 1;
            }
```

```
        }

        for (int i = 0; i < n; i++) {
            if (p[i]->remaining_bt > 0 && p[i]->queue_level == 3
               && p[i]->at <= current_time) {
                if (!p[i]->started) {
                    p[i]->rt = current_time - p[i]->at;
                    p[i]->started = 1;
                }
                current_time += p[i]->remaining_bt;
                p[i]->remaining_bt = 0;
                p[i]->ct = current_time;
                p[i]->tat = p[i]->ct - p[i]->at;
                p[i]->wt = p[i]->tat - p[i]->bt;
                awt += p[i]->wt;
                atat += p[i]->tat;

                executed = 1;
            }
        }

        if (!executed)
            current_time++;
    }

    printf("\n--- Multilevel Feedback Queue Scheduling ---\n");
    printf("PNo\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i]->pno, p[i]->
           at, p[i]->bt, p[i]->ct, p[i]->tat, p[i]->wt, p[i]->rt);
    }

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

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

    struct process* p[n];
    printf("Enter arrival time and burst time for each process:\n"
       );
    for (int i = 0; i < n; i++) {
        p[i] = (struct process*)malloc(sizeof(struct process));
        printf("P%d: ", i);
        scanf("%d%d", &p[i]->at, &p[i]->bt);
        p[i]->pno = i;
        p[i]->remaining_bt = p[i]->bt;
        p[i]->started = 0;
        p[i]->queue_level = 1;
    }
```

```
    multilevel_feedback_queue(p, n);

    for (int i = 0; i < n; i++)
        free(p[i]);

    return 0;
}
```

17 Write a program to simulate Multilevel Queue CPU scheduling algorithms to find the average turnaround time and average waiting time (Processes P0, P1, P4 in Queue 1 and Processes P2, P3 in Queue 2).

Code

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

struct process {
    int pno, at, bt, ct, tat, wt, rt;
    int remaining_bt, started;
    int queue_no;
};

int all_done(struct process* p[], int n) {
    for (int i = 0; i < n; i++) {
        if (p[i]->remaining_bt > 0)
            return 0;
    }
    return 1;
}

void multilevel_queue(struct process* p[], int n) {
    int current_time = 0;
    float awt = 0, atat = 0;
    int quantum = 4;

    while (!all_done(p, n)) {
        int executed = 0;

        for (int i = 0; i < n; i++) {
            if (p[i]->queue_no == 1 && p[i]->remaining_bt > 0 && p
                [i]->at <= current_time) {
```

```
                if (!p[i]->started) {
                    p[i]->rt = current_time - p[i]->at;
                    p[i]->started = 1;
                }
                int exec_time = (p[i]->remaining_bt > quantum) ?
                   quantum : p[i]->remaining_bt;
                current_time += exec_time;
                p[i]->remaining_bt -= exec_time;

                if (p[i]->remaining_bt == 0) {
                    p[i]->ct = current_time;
                    p[i]->tat = p[i]->ct - p[i]->at;
                    p[i]->wt = p[i]->tat - p[i]->bt;
                    awt += p[i]->wt;
                    atat += p[i]->tat;
                }

                executed = 1;
            }
        }

        for (int i = 0; i < n; i++) {
            if (p[i]->queue_no == 2 && p[i]->remaining_bt > 0 && p
               [i]->at <= current_time) {
                if (!p[i]->started) {
                    p[i]->rt = current_time - p[i]->at;
                    p[i]->started = 1;
                }
                current_time += p[i]->remaining_bt;
                p[i]->remaining_bt = 0;
                p[i]->ct = current_time;
                p[i]->tat = p[i]->ct - p[i]->at;
                p[i]->wt = p[i]->tat - p[i]->bt;
                awt += p[i]->wt;
                atat += p[i]->tat;

                executed = 1;
            }
        }

        if (!executed)
            current_time++;
    }

    printf("\n--- Multilevel Queue Scheduling ---\n");
    printf("PNo\tQueue\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\tQ%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i]->pno, p[
           i]->queue_no, p[i]->at, p[i]->bt, p[i]->ct, p[i]->tat,
           p[i]->wt, p[i]->rt);
    }

    printf("\nAverage Waiting Time: %.2f\n", awt / n);
    printf("Average Turnaround Time: %.2f\n", atat / n);
```

```
}

int main() {
    int n = 5;
    struct process* p[n];

    printf("Enter arrival time and burst time for 5 processes:\n")
        ;
    for (int i = 0; i < n; i++) {
        p[i] = (struct process*)malloc(sizeof(struct process));
        printf("P%d: ", i);
        scanf("%d%d", &p[i]->at, &p[i]->bt);
        p[i]->pno = i;
        p[i]->remaining_bt = p[i]->bt;
        p[i]->started = 0;
        if (i == 0 || i == 1 || i == 4)
            p[i]->queue_no = 1;
        else
            p[i]->queue_no = 2;
    }

    multilevel_queue(p, n);

    for (int i = 0; i < n; i++)
        free(p[i]);

    return 0;
}
```

18 Banker's Algorithm for Deadlock Avoidance

Code

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int main() {
    int n, m, i, j, k;
    int alloc[MAX_PROCESSES][MAX_RESOURCES];
    int max[MAX_PROCESSES][MAX_RESOURCES];
    int avail[MAX_RESOURCES];
    int need[MAX_PROCESSES][MAX_RESOURCES];
    int finish[MAX_PROCESSES] = {0};
    int safeSequence[MAX_PROCESSES];
    int count = 0;
```

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

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

    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);

    printf("Enter maximum matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &max[i][j]);

    printf("Enter available resources:\n");
    for (i = 0; i < m; i++)
        scanf("%d", &avail[i]);

    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];

    while (count < n) {
        bool found = false;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                bool canAllocate = true;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]) {
                        canAllocate = false;
                        break;
                    }
                }
                if (canAllocate) {
                    for (k = 0; k < m; k++)
                        avail[k] += alloc[i][k];
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence: ");
    for (i = 0; i < n; i++)
        printf("P%d ", safeSequence[i]);
```

```
    printf("\n");

    return 0;
}
```

19 Algorithm for Deadlock Detection

Code

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

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

    int allocation[n][m], request[n][m], available[m];
    bool finish[n];

    printf("Enter Allocation Matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &allocation[i][j]);

    printf("Enter Request Matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &request[i][j]);

    printf("Enter Available Resources:\n");
    for (int i = 0; i < m; i++)
        scanf("%d", &available[i]);

    for (int i = 0; i < n; i++)
        finish[i] = false;

    bool deadlock = false;

    while (true) {
        bool found = false;

        for (int i = 0; i < n; i++) {
            if (!finish[i]) {
                bool canFinish = true;
                for (int j = 0; j < m; j++) {
```

```
                    if (request[i][j] > available[j]) {
                        canFinish = false;
                        break;
                    }
                }

                if (canFinish) {
                    for (int j = 0; j < m; j++)
                        available[j] += allocation[i][j];

                    finish[i] = true;
                    found = true;
                }
            }
        }

        if (!found)
            break;
    }

    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            deadlock = true;
            break;
        }
    }

    if (deadlock) {
        printf("System is in a DEADLOCKED state.\n");
        printf("Deadlocked processes: ");
        for (int i = 0; i < n; i++) {
            if (!finish[i])
                printf("P%d ", i);
        }
        printf("\n");
    } else {
        printf("System is NOT in deadlock.\n");
    }

    return 0;
}
```

20 Page Replacement Algorithm - FIFO (First-In First-Out)

Code: FIFO Page Replacement Algorithm

```c
#include <stdio.h>

int main() {
    int frames, pages[100], n, i, j, k = 0, faults = 0, found;
    printf("Enter number of frames: ");
    scanf("%d", &frames);
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page reference string:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &pages[i]);

    int frame[frames];
    for (i = 0; i < frames; i++)
        frame[i] = -1;

    for (i = 0; i < n; i++) {
        found = 0;
        for (j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            frame[k] = pages[i];
            k = (k + 1) % frames;
            faults++;
        }
    }

    printf("Total Page Faults = %d\n", faults);
    return 0;
}
```

21 Page Replacement Algorithm - LRU (Least Recently Used)

Code: LRU Page Replacement Algorithm

```c
#include <stdio.h>

int main() {
    int frames, pages[100], n, i, j, k, faults = 0, time[100],
       counter = 0, pos, min, found;
    printf("Enter number of frames: ");
    scanf("%d", &frames);
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page reference string:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &pages[i]);

    int frame[frames];
    for (i = 0; i < frames; i++) {
        frame[i] = -1;
        time[i] = 0;
    }

    for (i = 0; i < n; i++) {
        found = 0;
        for (j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                time[j] = ++counter;
                break;
            }
        }
        if (!found) {
            int lru = 0, min_time = time[0];
            for (j = 1; j < frames; j++) {
                if (time[j] < min_time) {
                    min_time = time[j];
                    lru = j;
                }
            }
            frame[lru] = pages[i];
            time[lru] = ++counter;
            faults++;
        }
    }

    printf("Total Page Faults = %d\n", faults);
    return 0;
}
```

22 Page Replacement Algorithm - Optimal

Code: Optimal Page Replacement Algorithm

```c
#include <stdio.h>

int search(int key, int frame[], int size) {
    for (int i = 0; i < size; i++)
        if (frame[i] == key)
            return 1;
    return 0;
}

int predict(int pages[], int frame[], int n, int index, int size)
   {
    int res = -1, farthest = index;
    for (int i = 0; i < size; i++) {
        int j;
        for (j = index; j < n; j++) {
            if (frame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    res = i;
                }
                break;
            }
        }
        if (j == n)
            return i;
    }
    return (res == -1) ? 0 : res;
}

int main() {
    int frames, n, pages[100], frame[100], faults = 0, i, j;

    printf("Enter number of frames: ");
    scanf("%d", &frames);
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page reference string:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &pages[i]);

    int count = 0;
    for (i = 0; i < n; i++) {
        if (search(pages[i], frame, count)) continue;
        if (count < frames)
            frame[count++] = pages[i];
        else {
            int pos = predict(pages, frame, n, i + 1, frames);
            frame[pos] = pages[i];
        }
        faults++;
```

```
    }

    printf("Total Page Faults = %d\n", faults);
    return 0;
}
```

Commands

```
gcc program_name.c
./a.out

(or)

gcc program_name.c -o program_name
./program_name

if "./program_name" says "permission denied" then do this

chmod +x program_name
```