

OPERATING SYSTEMS PRACTICAL PROGRAMS

Student Name: _____

Roll Number: _____

Class: _____

Subject: Operating System Lab

Complete Set of 13 Programs

INDEX

1. Zombie Process using fork()
2. Orphan Process using fork()
3. FCFS (First Come First Serve) CPU Scheduling Algorithm
4. SJF (Shortest Job First) Non-Preemptive CPU Scheduling Algorithm
5. Round Robin CPU Scheduling Algorithm
6. Producer-Consumer Problem using Semaphore
7. Banker's Algorithm for Deadlock Avoidance
8. FIFO (First In First Out) Page Replacement Algorithm
9. LRU (Least Recently Used) Page Replacement Algorithm
10. FCFS (First Come First Serve) Disk Scheduling Algorithm
11. SSTF (Shortest Seek Time First) Disk Scheduling Algorithm
12. Deadlock using Multithreading (pthreads)
13. Simple Calculator using Bash Script
14. Deadlock Handling Commands

Program 1: Zombie Process using fork()

Theory:

A zombie process is a process that has completed execution but still has an entry in the process table. This occurs when a child process terminates but its parent hasn't called `wait()` to read its exit status. The child remains in the zombie state until the parent reads its status. Zombie processes don't consume system resources except for a process table entry.

Source Code:

```
/* Zombie Process Demonstration using fork() */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

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

    if(pid > 0) {
        // Parent process
        printf("Parent Process ID: %d\n", getpid());
        printf("Child Process ID: %d\n", pid);
        printf("Parent sleeping for 10 seconds...\n");
        printf("Child will become zombie during this time.\n");
        printf("Use 'ps aux | grep Z' in another terminal to see zombie process\n\n");
        sleep(10); // Parent sleeps, child becomes zombie
        printf("Parent exiting now.\n");
    }
    else if(pid == 0) {
        // Child process
        printf("Child Process ID: %d\n", getpid());
        printf("Child Parent ID: %d\n", getppid());
        printf("Child exiting immediately...\n\n");
        exit(0); // Child terminates, becomes zombie
    }
    else {
        // Fork failed
        printf("Fork failed!\n");
        return 1;
    }

    return 0;
}
```

Sample Input:

No input required

Sample Output:

```
Parent Process ID: 1234
Child Process ID: 1235
Parent sleeping for 10 seconds...
Child will become zombie during this time.
Use 'ps aux | grep Z' in another terminal to see zombie process
```

```
Child Process ID: 1235
Child Parent ID: 1234
Child exiting immediately...
```

```
Parent exiting now.
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit zombie.c
```

3. Compile:

```
gcc zombie.c -o zombie
```

4. Run:

```
./zombie
```

Program 2: Orphan Process using fork()

Theory:

An orphan process is a process whose parent has terminated before it. When a parent process terminates before its child, the child becomes an orphan. The init process (PID 1) or systemd automatically adopts orphan processes. The orphan process continues execution normally under its new parent until it completes.

Source Code:

```
/* Orphan Process Demonstration using fork() */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

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

    if(pid > 0) {
        // Parent process
        printf("Parent Process ID: %d\n", getpid());
        printf("Child Process ID: %d\n", pid);
        printf("Parent exiting immediately...\n\n");
        exit(0); // Parent terminates, child becomes orphan
    }
    else if(pid == 0) {
        // Child process
        printf("Child Process ID: %d\n", getpid());
        printf("Original Parent ID: %d\n", getppid());
        printf("Child sleeping for 5 seconds...\n");
        sleep(5); // Sleep to allow parent to terminate
        printf("Child woke up. New Parent ID (init/systemd): %d\n", getppid());
        printf("Child is now an orphan process.\n");
    }
    else {
        // Fork failed
        printf("Fork failed!\n");
        return 1;
    }

    return 0;
}
```

Sample Input:

No input required

Sample Output:

```
Parent Process ID: 1234  
Child Process ID: 1235  
Parent exiting immediately...
```

```
Child Process ID: 1235  
Original Parent ID: 1234  
Child sleeping for 5 seconds...  
Child woke up. New Parent ID (init/systemd): 1  
Child is now an orphan process.
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit orphan.c
```

3. Compile:

```
gcc orphan.c -o orphan
```

4. Run:

```
./orphan
```

Program 3: FCFS (First Come First Serve) CPU Scheduling Algorithm

Theory:

FCFS is the simplest CPU scheduling algorithm where processes are executed in the order they arrive in the ready queue. It is non-preemptive, meaning once CPU is allocated to a process, it runs until completion. The main advantage is simplicity, but it suffers from the convoy effect where short processes wait for long processes to complete.

Source Code:

```
/* FCFS (First Come First Serve) CPU Scheduling Algorithm */
#include <stdio.h>

int main() {
    int n, i;
    float avg_wt = 0, avg_tat = 0;

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

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

    // Input process details
    printf("\nEnter Burst Time for each process:\n");
    for(i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Process P%d: ", i + 1);
        scanf("%d", &bt[i]);
    }

    // Calculate Waiting Time
    wt[0] = 0; // First process has 0 waiting time
    for(i = 1; i < n; i++) {
        wt[i] = wt[i-1] + bt[i-1];
    }

    // Calculate Turnaround Time
    for(i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }

    // Display results
    printf("\n\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time\n");
    printf("-----\n");

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

```

        printf("P%d\t\t%d\t\t%d\t\t%d\n", pid[i], bt[i], wt[i], tat[i]);
        avg_wt += wt[i];
        avg_tat += tat[i];
    }

    // Calculate averages
    avg_wt /= n;
    avg_tat /= n;

    printf("\nAverage Waiting Time: %.2f", avg_wt);
    printf("\nAverage Turnaround Time: %.2f\n", avg_tat);

    return 0;
}

```

Sample Input:

```

Enter number of processes: 4
Enter Burst Time for each process:
Process P1: 5
Process P2: 3
Process P3: 8
Process P4: 6

```

Sample Output:

Process	Burst Time	Waiting Time	Turnaround Time
P1	5	0	5
P2	3	5	8
P3	8	8	16
P4	6	16	22

```

Average Waiting Time: 7.25
Average Turnaround Time: 12.75

```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit fcfs.c
```

3. Compile:

```
gcc fcfs.c -o fcfs
```

4. Run:

```
./fcfs
```


Program 4: SJF (Shortest Job First) Non-Preemptive CPU Scheduling Algorithm

Theory:

SJF scheduling selects the process with the smallest burst time for execution next. It is a non-preemptive algorithm that minimizes average waiting time. However, it can cause starvation for longer processes and requires knowledge of burst times in advance. SJF is optimal in terms of average waiting time but difficult to implement in practice.

Source Code:

```
/* SJF (Shortest Job First) Non-Preemptive CPU Scheduling Algorithm */
#include <stdio.h>

int main() {
    int n, i, j, temp;
    float avg_wt = 0, avg_tat = 0;

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

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

    // Input process details
    printf("\nEnter Burst Time for each process:\n");
    for(i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Process P%d: ", i + 1);
        scanf("%d", &bt[i]);
    }

    // Sort processes by burst time (SJF sorting)
    for(i = 0; i < n-1; i++) {
        for(j = i+1; j < n; j++) {
            if(bt[i] > bt[j]) {
                // Swap burst times
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap process IDs
                temp = pid[i];
                pid[i] = pid[j];
                pid[j] = temp;
            }
        }
    }

    // Calculate Waiting Time
```

```

    wt[0] = 0;
    for(i = 1; i < n; i++) {
        wt[i] = wt[i-1] + bt[i-1];
    }

    // Calculate Turnaround Time
    for(i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }

    // Display results
    printf("\n\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time\n");
    printf("-----\n");

    for(i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", pid[i], bt[i], wt[i], tat[i]);
        avg_wt += wt[i];
        avg_tat += tat[i];
    }

    // Calculate averages
    avg_wt /= n;
    avg_tat /= n;

    printf("\nAverage Waiting Time: %.2f", avg_wt);
    printf("\nAverage Turnaround Time: %.2f\n", avg_tat);

    return 0;
}

```

Sample Input:

```

Enter number of processes: 4
Enter Burst Time for each process:
Process P1: 6
Process P2: 2
Process P3: 8
Process P4: 3

```

Sample Output:

Process	Burst Time	Waiting Time	Turnaround Time
P2	2	0	2
P4	3	2	5
P1	6	5	11
P3	8	11	19

Average Waiting Time: 4.50

Average Turnaround Time: 9.25

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit sjf.c
```

3. Compile:

```
gcc sjf.c -o sjf
```

4. Run:

```
./sjf
```

Program 5: Round Robin CPU Scheduling Algorithm

Theory:

Round Robin is a preemptive CPU scheduling algorithm where each process gets a small unit of CPU time called time quantum. After the time quantum expires, the process is preempted and added to the end of the ready queue. It ensures fair allocation of CPU time to all processes. Performance depends heavily on the time quantum size.

Source Code:

```
/* Round Robin CPU Scheduling Algorithm */
#include <stdio.h>

int main() {
    int n, i, quantum, time = 0;
    float avg_wt = 0, avg_tat = 0;

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

    int pid[n], bt[n], rem_bt[n], wt[n], tat[n];

    // Input process details
    printf("\nEnter Burst Time for each process:\n");
    for(i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Process P%d: ", i + 1);
        scanf("%d", &bt[i]);
        rem_bt[i] = bt[i]; // Remaining burst time
    }

    printf("\nEnter Time Quantum: ");
    scanf("%d", &quantum);

    // Round Robin scheduling
    int complete = 0;
    while(complete < n) {
        for(i = 0; i < n; i++) {
            if(rem_bt[i] > 0) {
                if(rem_bt[i] > quantum) {
                    time += quantum;
                    rem_bt[i] -= quantum;
                }
                else {
                    time += rem_bt[i];
                    wt[i] = time - bt[i];
                    rem_bt[i] = 0;
                    complete++;
                }
            }
        }
    }
}
```

```

        }
    }
}

// Calculate Turnaround Time
for(i = 0; i < n; i++) {
    tat[i] = wt[i] + bt[i];
}

// Display results
printf("\n\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time\n");
printf("-----\n");

for(i = 0; i < n; i++) {
    printf("P%d\t\t%d\t\t%d\t\t%d\n", pid[i], bt[i], wt[i], tat[i]);
    avg_wt += wt[i];
    avg_tat += tat[i];
}

// Calculate averages
avg_wt /= n;
avg_tat /= n;

printf("\nAverage Waiting Time: %.2f", avg_wt);
printf("\nAverage Turnaround Time: %.2f\n", avg_tat);

return 0;
}

```

Sample Input:

```

Enter number of processes: 4
Enter Burst Time for each process:
Process P1: 5
Process P2: 3
Process P3: 8
Process P4: 6
Enter Time Quantum: 2

```

Sample Output:

Process	Burst Time	Waiting Time	Turnaround Time
P1	5	12	17
P2	3	4	7
P3	8	16	24
P4	6	10	16

Average Waiting Time: 10.50
Average Turnaround Time: 16.00

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit round_robin.c
```

3. Compile:

```
gcc round_robin.c -o round_robin
```

4. Run:

```
./round_robin
```

Program 6: Producer-Consumer Problem using Semaphore

Theory:

The Producer-Consumer problem is a classic synchronization problem where producers generate data and place it in a shared buffer, while consumers remove data from the buffer. Semaphores are used to prevent race conditions: mutex ensures mutual exclusion, empty counts available buffer slots, and full counts filled slots. This prevents buffer overflow and underflow.

Source Code:

```
/* Producer-Consumer Problem using Semaphore */
#include <stdio.h>
#include <stdlib.h>

#define MAX 5 // Maximum buffer size

// Semaphore variables
int mutex = 1; // Mutual exclusion semaphore
int full = 0; // Counts full buffer slots
int empty = MAX; // Counts empty buffer slots
int buffer[MAX]; // Shared buffer
int x = 0; // Buffer index

// Wait operation (P operation / Down operation)
int wait(int s) {
    return (--s);
}

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

// Producer function
void producer() {
    if((mutex == 1) && (empty != 0)) {
        mutex = wait(mutex); // Lock critical section
        empty = wait(empty); // Decrease empty slots

        printf("Enter item to produce: ");
        scanf("%d", &buffer[x]);
        printf("Item produced: %d at position %d\n", buffer[x], x);
        x++;

        full = signal(full); // Increase full slots
        mutex = signal(mutex); // Unlock critical section
    }
    else {
```

```

        printf("Buffer is Full! Cannot produce.\n");
    }
}

// Consumer function
void consumer() {
    if((mutex == 1) && (full != 0)) {
        mutex = wait(mutex); // Lock critical section
        full = wait(full);    // Decrease full slots

        x--;
        printf("Item consumed: %d from position %d\n", buffer[x], x);

        empty = signal(empty); // Increase empty slots
        mutex = signal(mutex); // Unlock critical section
    }
    else {
        printf("Buffer is Empty! Cannot consume.\n");
    }
}

// Display buffer contents
void display() {
    int i;
    if(x == 0) {
        printf("Buffer is empty\n");
    }
    else {
        printf("Buffer items: ");
        for(i = 0; i < x; i++) {
            printf("%d ", buffer[i]);
        }
        printf("\n");
        printf("Full slots: %d, Empty slots: %d\n", full, empty);
    }
}

int main() {
    int choice;

    while(1) {
        printf("\n===== Producer Consumer Problem =====\n");
        printf("1. Produce\n");
        printf("2. Consume\n");
        printf("3. Display Buffer\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                producer();

```

```

        break;

    case 2:
        consumer();
        break;

    case 3:
        display();
        break;

    case 4:
        printf("Exiting...\n");
        exit(0);

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

return 0;
}

```

Sample Input:

```

Enter your choice: 1
Enter item to produce: 10
Enter your choice: 1
Enter item to produce: 20
Enter your choice: 3
Enter your choice: 2
Enter your choice: 4

```

Sample Output:

```

===== Producer Consumer Problem =====

```

1. Produce
2. Consume
3. Display Buffer
4. Exit

```

Enter your choice: 1
Enter item to produce: 10
Item produced: 10 at position 0

```

```

Enter your choice: 3
Buffer items: 10
Full slots: 1, Empty slots: 4

```

```

Enter your choice: 2
Item consumed: 10 from position 0

```

```
Enter your choice: 4  
Exiting...
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit producer_consumer.c
```

3. Compile:

```
gcc producer_consumer.c -o producer_consumer
```

4. Run:

```
./producer_consumer
```

Program 7: Banker's Algorithm for Deadlock Avoidance

Theory:

Banker's Algorithm is a deadlock avoidance algorithm that tests resource allocation for safety before granting requests. It maintains information about available resources, maximum resource needs, and current allocation. The algorithm checks if granting a request leads to a safe state by finding a safe sequence. If no safe sequence exists, the system is in an unsafe state and deadlock may occur.

Source Code:

```
/* Banker's Algorithm for Deadlock Avoidance */
#include <stdio.h>

int main() {
    int n, m, i, j, k;

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

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

    int alloc[n][m], max[n][m], need[n][m];
    int avail[m];

    // Input Allocation Matrix
    printf("\nEnter Allocation Matrix:\n");
    for(i = 0; i < n; i++) {
        printf("Process P%d: ", i);
        for(j = 0; j < m; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }

    // Input Maximum Matrix
    printf("\nEnter Maximum Matrix:\n");
    for(i = 0; i < n; i++) {
        printf("Process P%d: ", i);
        for(j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    // Input Available Resources
    printf("\nEnter Available Resources:\n");
    for(i = 0; i < m; i++) {
        scanf("%d", &avail[i]);
    }
}
```

```

}

// Calculate Need Matrix (Need = Max - Allocation)
for(i = 0; i < n; i++) {
    for(j = 0; j < m; j++) {
        need[i][j] = max[i][j] - alloc[i][j];
    }
}

// Display Need Matrix
printf("\nNeed Matrix:\n");
for(i = 0; i < n; i++) {
    printf("P%d: ", i);
    for(j = 0; j < m; j++) {
        printf("%d ", need[i][j]);
    }
    printf("\n");
}

// Banker's Algorithm for Safe State Check
int finish[n], safeSeq[n], work[m];

// Initialize finish array
for(i = 0; i < n; i++) {
    finish[i] = 0;
}

// Initialize work array with available resources
for(i = 0; i < m; i++) {
    work[i] = avail[i];
}

int count = 0;

// Find safe sequence
while(count < n) {
    int found = 0;

    for(i = 0; i < n; i++) {
        if(finish[i] == 0) { // If process not finished

            // Check if need <= work
            for(j = 0; j < m; j++) {
                if(need[i][j] > work[j])
                    break;
            }

            // If all resources can be allocated
            if(j == m) {
                // Add allocated resources back to work
                for(k = 0; k < m; k++) {

```

```

        work[k] += alloc[i][k];
    }

    safeSeq[count++] = i;
    finish[i] = 1;
    found = 1;
}
}

// If no process found in this iteration
if(found == 0) {
    printf("\nSystem is NOT in SAFE state!\n");
    printf("Deadlock may occur.\n");
    return 0;
}

// System is in safe state
printf("\nSystem is in SAFE state!\n");
printf("Safe sequence is: ");
for(i = 0; i < n; i++) {
    printf("P%d ", safeSeq[i]);
    if(i < n-1) printf("-> ");
}
printf("\n");

return 0;
}

```

Sample Input:

```

Enter number of processes: 5
Enter number of resources: 3
Enter Allocation Matrix:
Process P0: 0 1 0
Process P1: 2 0 0
Process P2: 3 0 2
Process P3: 2 1 1
Process P4: 0 0 2
Enter Maximum Matrix:
Process P0: 7 5 3
Process P1: 3 2 2
Process P2: 9 0 2
Process P3: 2 2 2
Process P4: 4 3 3
Enter Available Resources:
3 3 2

```

Sample Output:

Need Matrix:

P0: 7 4 3

P1: 1 2 2

P2: 6 0 0

P3: 0 1 1

P4: 4 3 1

System is in SAFE state!

Safe sequence is: P1 -> P3 -> P4 -> P2 -> P0

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit bankers.c
```

3. Compile:

```
gcc bankers.c -o bankers
```

4. Run:

```
./bankers
```

Program 8: FIFO (First In First Out) Page Replacement Algorithm

Theory:

FIFO is the simplest page replacement algorithm that replaces the oldest page in memory when a page fault occurs. Pages are maintained in a queue, and the page at the front is replaced. While simple to implement, FIFO suffers from Belady's anomaly where increasing frames can increase page faults. It doesn't consider page usage patterns.

Source Code:

```
/* FIFO (First In First Out) Page Replacement Algorithm */
#include <stdio.h>

int main() {
    int n, f, i, j, flag, faults = 0;

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

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

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

    int frames[f];

    // Initialize frames with -1 (empty)
    for(i = 0; i < f; i++) {
        frames[i] = -1;
    }

    int index = 0; // FIFO queue index

    printf("\nFIFO Page Replacement:\n");
    printf("-----\n");

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

        // Check if page already in frame (Page Hit)
        for(j = 0; j < f; j++) {
            if(frames[j] == pages[i]) {
```

```

        flag = 1; // Page hit
        break;
    }
}

// If page not found (Page Fault)
if(flag == 0) {
    frames[index] = pages[i];
    index = (index + 1) % f; // Circular queue
    faults++;
    printf("Page %d -> ", pages[i]);
}
else {
    printf("Page %d -> ", pages[i]);
}

// Display current frame status
for(j = 0; j < f; j++) {
    if(frames[j] != -1)
        printf("%d ", frames[j]);
    else
        printf("- ");
}

if(flag == 0)
    printf(" (Page Fault)");
else
    printf(" (Page Hit)");

printf("\n");
}

printf("-----\n");
printf("Total Page Faults: %d\n", faults);
printf("Page Fault Ratio: %.2f\n", (float)faults/n);
printf("Page Hit Ratio: %.2f\n", (float)(n-faults)/n);

return 0;
}

```

Sample Input:

```

Enter number of pages: 12
Enter page reference string:
1 3 0 3 5 6 3 1 6 3 0 5
Enter number of frames: 3

```

Sample Output:

FIFO Page Replacement:

```
-----  
Page 1 -> 1 - - (Page Fault)  
Page 3 -> 1 3 - (Page Fault)  
Page 0 -> 1 3 0 (Page Fault)  
Page 3 -> 1 3 0 (Page Hit)  
Page 5 -> 5 3 0 (Page Fault)  
Page 6 -> 5 6 0 (Page Fault)  
Page 3 -> 5 6 3 (Page Fault)  
Page 1 -> 1 6 3 (Page Fault)  
Page 6 -> 1 6 3 (Page Hit)  
Page 3 -> 1 6 3 (Page Hit)  
Page 0 -> 1 0 3 (Page Fault)  
Page 5 -> 1 0 5 (Page Fault)  
-----
```

Total Page Faults: 9

Page Fault Ratio: 0.75

Page Hit Ratio: 0.25

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit fifo_page.c
```

3. Compile:

```
gcc fifo_page.c -o fifo_page
```

4. Run:

```
./fifo_page
```

Program 9: LRU (Least Recently Used) Page Replacement Algorithm

Theory:

LRU replaces the page that has not been used for the longest time. It uses the principle of temporal locality - recently used pages are likely to be used again soon. LRU tracks usage time for each page and replaces the least recently used one during a page fault. It performs better than FIFO but requires additional overhead to track page usage history.

Source Code:

```
/* LRU (Least Recently Used) Page Replacement Algorithm */
#include <stdio.h>

int main() {
    int n, f, i, j, k, flag, faults = 0;

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

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

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

    int frames[f], recent[f];

    // Initialize frames with -1 (empty)
    for(i = 0; i < f; i++) {
        frames[i] = -1;
        recent[i] = 0;
    }

    printf("\nLRU Page Replacement:\n");
    printf("-----\n");

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

        // Check if page already in frame (Page Hit)
        for(j = 0; j < f; j++) {
            if(frames[j] == pages[i]) {
                flag = 1; // Page hit
            }
        }
    }
}
```

```

        recent[j] = i; // Update recent usage
        break;
    }
}

// If page not found (Page Fault)
if(flag == 0) {
    int lru_index = 0, min_recent = recent[0];

    // Find LRU page (least recently used)
    for(j = 1; j < f; j++) {
        if(frames[j] == -1) {
            lru_index = j;
            break;
        }
        if(recent[j] < min_recent) {
            min_recent = recent[j];
            lru_index = j;
        }
    }

    frames[lru_index] = pages[i];
    recent[lru_index] = i;
    faults++;
    printf("Page %d -> ", pages[i]);
}
else {
    printf("Page %d -> ", pages[i]);
}

// Display current frame status
for(j = 0; j < f; j++) {
    if(frames[j] != -1)
        printf("%d ", frames[j]);
    else
        printf("- ");
}

if(flag == 0)
    printf(" (Page Fault)");
else
    printf(" (Page Hit)");

printf("\n");
}

printf("-----\n");
printf("Total Page Faults: %d\n", faults);
printf("Page Fault Ratio: %.2f\n", (float) faults/n);
printf("Page Hit Ratio: %.2f\n", (float) (n-faults)/n);

return 0;

```

```
}
```

Sample Input:

```
Enter number of pages: 12
Enter page reference string:
7 0 1 2 0 3 0 4 2 3 0 3
Enter number of frames: 4
```

Sample Output:

LRU Page Replacement:

```
-----
Page 7 -> 7 - - - (Page Fault)
Page 0 -> 7 0 - - (Page Fault)
Page 1 -> 7 0 1 - (Page Fault)
Page 2 -> 7 0 1 2 (Page Fault)
Page 0 -> 7 0 1 2 (Page Hit)
Page 3 -> 3 0 1 2 (Page Fault)
Page 0 -> 3 0 1 2 (Page Hit)
Page 4 -> 3 0 4 2 (Page Fault)
Page 2 -> 3 0 4 2 (Page Hit)
Page 3 -> 3 0 4 2 (Page Hit)
Page 0 -> 3 0 4 2 (Page Hit)
Page 3 -> 3 0 4 2 (Page Hit)
-----
```

```
Total Page Faults: 6
Page Fault Ratio: 0.50
Page Hit Ratio: 0.50
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit lru_page.c
```

3. Compile:

```
gcc lru_page.c -o lru_page
```

4. Run:

```
./lru_page
```

Program 10: FCFS (First Come First Serve) Disk Scheduling Algorithm

Theory:

FCFS disk scheduling services requests in the order they arrive, similar to a queue. The disk arm moves to service each request sequentially without optimization. While simple and fair, it doesn't minimize seek time and can cause high disk arm movement. It works well when the load is light but performs poorly under heavy load.

Source Code:

```
/* FCFS (First Come First Serve) Disk Scheduling Algorithm */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int request[50], n, head, i;
    int total_seek = 0;

    printf("Enter number of disk requests: ");
    scanf("%d", &n);

    printf("Enter disk request queue:\n");
    for(i = 0; i < n; i++) {
        scanf("%d", &request[i]);
    }

    printf("Enter initial head position: ");
    scanf("%d", &head);

    printf("\n==== FCFS Disk Scheduling =====\n");
    printf("Initial head position: %d\n\n", head);

    int initial_head = head; // Save initial position

    // Process requests in FCFS order
    for(i = 0; i < n; i++) {
        int distance = abs(request[i] - head);
        total_seek += distance;

        printf("Move from %d to %d (Distance: %d)\n", head, request[i], distance);
        head = request[i];
    }

    printf("\n===== \n");
    printf("Total Seek Time: %d\n", total_seek);
    printf("Average Seek Time: %.2f\n", (float)total_seek / n);
}
```

```
    return 0;
}
```

Sample Input:

```
Enter number of disk requests: 8
Enter disk request queue:
82 170 43 140 24 16 190 50
Enter initial head position: 50
```

Sample Output:

```
===== FCFS Disk Scheduling =====
Initial head position: 50

Move from 50 to 82 (Distance: 32)
Move from 82 to 170 (Distance: 88)
Move from 170 to 43 (Distance: 127)
Move from 43 to 140 (Distance: 97)
Move from 140 to 24 (Distance: 116)
Move from 24 to 16 (Distance: 8)
Move from 16 to 190 (Distance: 174)
Move from 190 to 50 (Distance: 140)

=====
Total Seek Time: 782
Average Seek Time: 97.75
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit disk_fcfs.c
```

3. Compile:

```
gcc disk_fcfs.c -o disk_fcfs
```

4. Run:

```
./disk_fcfs
```

Program 11: SSTF (Shortest Seek Time First) Disk Scheduling Algorithm

Theory:

SSTF selects the request closest to the current head position, minimizing seek time for each request. It reduces average response time and total seek time compared to FCFS. However, it can cause starvation for requests far from the current position and may not be optimal for all request patterns. It's a greedy algorithm focusing on immediate optimization.

Source Code:

```
/* SSTF (Shortest Seek Time First) Disk Scheduling Algorithm */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int request[50], visited[50];
    int n, head, i, j, min, index;
    int total_seek = 0;

    printf("Enter number of disk requests: ");
    scanf("%d", &n);

    printf("Enter disk request queue:\n");
    for(i = 0; i < n; i++) {
        scanf("%d", &request[i]);
        visited[i] = 0; // Mark as not visited
    }

    printf("Enter initial head position: ");
    scanf("%d", &head);

    printf("\n==== SSTF Disk Scheduling =====\n");
    printf("Initial head position: %d\n\n", head);

    // Process all requests
    for(i = 0; i < n; i++) {
        min = 9999;

        // Find closest request (shortest seek time)
        for(j = 0; j < n; j++) {
            if(visited[j] == 0) {
                int distance = abs(request[j] - head);

                if(distance < min) {
                    min = distance;
                    index = j;
                }
            }
        }

        // Update head position and total seek time
        head = request[index];
        total_seek += min;
        visited[index] = 1;
    }

    printf("Total seek time: %d\n", total_seek);
}
```

```

        }
    }

    // Mark request as visited
    visited[index] = 1;
    total_seek += min;

    printf("Move from %d to %d (Distance: %d)\n", head, request[index], min);
    head = request[index];
}

printf("\n===== \n");
printf("Total Seek Time: %d\n", total_seek);
printf("Average Seek Time: %.2f\n", (float)total_seek / n);

return 0;
}

```

Sample Input:

```

Enter number of disk requests: 8
Enter disk request queue:
82 170 43 140 24 16 190 50
Enter initial head position: 50

```

Sample Output:

```

===== SSTF Disk Scheduling =====
Initial head position: 50

Move from 50 to 43 (Distance: 7)
Move from 43 to 24 (Distance: 19)
Move from 24 to 16 (Distance: 8)
Move from 16 to 82 (Distance: 66)
Move from 82 to 140 (Distance: 58)
Move from 140 to 170 (Distance: 30)
Move from 170 to 190 (Distance: 20)
Move from 190 to 50 (Distance: 140)

=====
Total Seek Time: 348
Average Seek Time: 43.50

```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit disk_sstf.c
```

3. Compile:

```
gcc disk_sstf.c -o disk_sstf
```

4. Run:

```
./disk_sstf
```

Program 12: Deadlock using Multithreading (pthreads)

Theory:

A deadlock occurs when two or more threads are blocked forever, each waiting for a resource held by another. This program demonstrates circular wait using two threads and two resources. Thread 1 locks Resource 1 and waits for Resource 2, while Thread 2 locks Resource 2 and waits for Resource 1, creating a deadlock situation that requires external intervention to resolve.

Source Code:

```
/* Deadlock Demonstration using Multithreading with pthreads */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// Two resources (mutexes)
pthread_mutex_t resource1;
pthread_mutex_t resource2;

// Thread 1 function - locks resource1 then resource2
void* thread1_function(void* arg) {
    printf("Thread 1: Trying to lock Resource 1...\n");
    pthread_mutex_lock(&resource1);
    printf("Thread 1: Locked Resource 1 successfully!\n");

    sleep(1); // Simulate some work

    printf("Thread 1: Now trying to lock Resource 2...\n");
    pthread_mutex_lock(&resource2); // Deadlock occurs here
    printf("Thread 1: Locked Resource 2 successfully!\n");

    // Release locks
    pthread_mutex_unlock(&resource2);
    pthread_mutex_unlock(&resource1);

    printf("Thread 1: Released both resources\n");
    return NULL;
}

// Thread 2 function - locks resource2 then resource1
void* thread2_function(void* arg) {
    printf("Thread 2: Trying to lock Resource 2...\n");
    pthread_mutex_lock(&resource2);
    printf("Thread 2: Locked Resource 2 successfully!\n");

    sleep(1); // Simulate some work
```

```

printf("Thread 2: Now trying to lock Resource 1...\n");
pthread_mutex_lock(&resource1); // Deadlock occurs here
printf("Thread 2: Locked Resource 1 successfully!\n");

// Release locks
pthread_mutex_unlock(&resource1);
pthread_mutex_unlock(&resource2);

printf("Thread 2: Released both resources\n");
return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Initialize mutexes
    pthread_mutex_init(&resource1, NULL);
    pthread_mutex_init(&resource2, NULL);

    printf("==== Deadlock Demonstration =====\n");
    printf("Creating two threads...\n\n");

    // Create threads
    pthread_create(&thread1, NULL, thread1_function, NULL);
    pthread_create(&thread2, NULL, thread2_function, NULL);

    // Wait for threads (will wait forever due to deadlock)
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Cleanup (never reached due to deadlock)
    pthread_mutex_destroy(&resource1);
    pthread_mutex_destroy(&resource2);

    printf("\nBoth threads completed (this won't print due to deadlock)\n");

    return 0;
}

```

Sample Input:

No input required (program will hang in deadlock)

Sample Output:

```

==== Deadlock Demonstration =====
Creating two threads...

```

```
Thread 1: Trying to lock Resource 1...
Thread 1: Locked Resource 1 successfully!
Thread 2: Trying to lock Resource 2...
Thread 2: Locked Resource 2 successfully!
Thread 1: Now trying to lock Resource 2...
Thread 2: Now trying to lock Resource 1...
[Program hangs here in deadlock - use Ctrl+C to terminate]
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit deadlock.c
```

3. Compile with pthread flag:

```
gcc deadlock.c -o deadlock -lpthread
```

4. Run the program:

```
./deadlock
```

5. To kill deadlocked process:

Press Ctrl+C or use kill command (see Deadlock Handling section)

Program 13: Simple Calculator using Bash Script

Theory:

This bash script implements a simple calculator that performs basic arithmetic operations (addition, subtraction, multiplication, division). It uses the bc (basic calculator) command for floating-point arithmetic. The script demonstrates shell programming concepts including input/output, conditional statements using case, and error handling for division by zero.

Source Code:

```
#!/bin/bash
# Simple Calculator using Bash Script

echo "==== Simple Calculator ====="
echo ""

# Input first number
echo "Enter first number:"
read num1

# Input operator
echo "Enter operator (+, -, *, /):"
read op

# Input second number
echo "Enter second number:"
read num2

# Perform calculation based on operator
case $op in
    +)
        result=$(echo "$num1 + $num2" | bc)
        echo ""
        echo "Result: $num1 + $num2 = $result"
        ;;
    -)
        result=$(echo "$num1 - $num2" | bc)
        echo ""
        echo "Result: $num1 - $num2 = $result"
        ;;
    \*)
        result=$(echo "$num1 * $num2" | bc)
        echo ""
        echo "Result: $num1 * $num2 = $result"
        ;;
    /)
        if [ $num2 -eq 0 ]; then
            echo ""
            echo "Error: Division by zero!"
        else
            result=$(echo "$num1 / $num2" | bc)
            echo ""
            echo "Result: $num1 / $num2 = $result"
        fi
        ;;
    *)
        echo "Invalid operator"
        exit 1
esac
```

```

        else
            result=$(echo "scale=2; $num1 / $num2" | bc)
            echo ""
            echo "Result: $num1 / $num2 = $result"
        fi
    ;;
*)
    echo ""
    echo "Error: Invalid operator! Please use +, -, *, or /"
    ;;
esac

echo ""
echo "=====
```

Sample Input:

```

Enter first number: 10
Enter operator (+, -, *, /): +
Enter second number: 5
```

Sample Output:

```

===== Simple Calculator =====

Enter first number:
10
Enter operator (+, -, *, /):
+
Enter second number:
5

Result: 10 + 5 = 15

=====
```

Ubuntu Execution Steps:

1. Open Terminal

2. Create file:

```
gedit calculator.sh
```

3. Give execute permission:

```
chmod +x calculator.sh
```

4. Run the script:

```
./calculator.sh
```


Deadlock Handling Commands

Introduction:

When a program enters deadlock state, it stops responding and needs to be terminated manually. The following commands help identify and kill deadlocked processes in Linux/Ubuntu.

1. Finding Process ID (PID)

Method 1: Using ps command

```
ps aux | grep program_name
```

Example: `ps aux | grep deadlock`

Explanation: Lists all processes and filters for the specified program name. The second column shows the Process ID (PID).

Method 2: Using top command

```
top
```

Explanation: Displays real-time view of running processes. Press 'q' to quit. Shows CPU usage, memory usage, and PID for all processes.

Method 3: Using pgrep command

```
pgrep program_name
```

Example: `pgrep deadlock`

Explanation: Returns only the PID of the matching process name.

2. Killing Deadlocked Process

Method 1: Normal kill command

```
kill PID
```

Example: `kill 1234`

Explanation: Sends SIGTERM signal (signal 15) to the process, requesting graceful termination. The process can catch this signal and clean up before exiting.

Method 2: Forceful kill command

```
kill -9 PID
```

Example: `kill -9 1234`

Explanation: Sends SIGKILL signal (signal 9) for immediate termination. The process cannot catch or ignore this signal. Use when normal kill doesn't work.

Method 3: Using killall command

```
killall program_name
```

Example: `killall deadlock`

Explanation: Kills all processes matching the program name. Useful when multiple instances are running.

3. Quick Reference for Deadlock Program

Step 1: Identify the deadlocked process

```
ps aux | grep deadlock
```

Step 2: Note the PID (second column)

Example output: user 1234 0.0 0.1 ./deadlock

Step 3: Kill the process

```
kill -9 1234
```

4. Alternative: Using Ctrl+C

If the deadlocked program is running in the current terminal, simply press:

```
Ctrl + C
```

This sends an interrupt signal to terminate the foreground process.

Important Notes:

- Always try normal kill before using kill -9
- Be careful with PID numbers - killing wrong process can affect system
- Deadlock program is designed to deadlock - it's expected behavior
- Use these commands to understand process management in Linux