SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 4th Sem

OPERATING SYSTEMS (USA23402J)- Lab Manual

Lab 1: Comparison between various Operating Systems

Title

Comparison between various Operating Systems

Aim

To understand and compare the key features, architectures, and functionalities of different operating systems (e.g., Windows, macOS, Linux, Android, iOS).

Procedure

1. **Research:** Gather information on the history, design philosophy, kernel types (monolithic, microkernel, hybrid), file systems, process management, memory management, security features, and user interfaces of at least three different operating systems.
2. **Categorization:** Classify the operating systems based on their target devices (desktop, mobile, server) and primary use cases.
3. **Comparison Table:** Create a comparative table highlighting the similarities and differences across various parameters.
4. **Analysis:** Analyze the strengths and weaknesses of each operating system in different scenarios.
5. **Conclusion:** Summarize the findings and discuss which operating system might be best suited for specific applications or user needs.

Source Code

(N/A - This lab is primarily theoretical and research-based, not involving direct coding.)

Input

(N/A - This lab does not require specific program input.)

Expected Output

A comprehensive report or presentation detailing the comparison, including a comparative table and analysis of the chosen operating systems.

Lab 2: Booting process in GNU/Linux OS

Title

Booting process in GNU/Linux OS

Aim

To understand the sequence of events and components involved in the booting process of a GNU/Linux operating system.

Procedure

1. **BIOS/UEFI Initialization:** Describe the role of BIOS/UEFI in the initial boot sequence, including POST (Power-On Self-Test) and loading the boot loader.
2. **Boot Loader (GRUB/LILO):** Explain how boot loaders like GRUB or LILO take control, locate the kernel, and load it into memory.
3. **Kernel Initialization:** Detail the steps involved in kernel initialization, including hardware detection, module loading, and setting up the root filesystem.
4. **Init/Systemd:** Explain the role of the `init` process (or `systemd` in modern Linux distributions) as the first user-space process, responsible for starting other services and bringing the system to a usable state.
5. **Runlevels/Targets:** Discuss the concept of runlevels (SysVinit) or targets (systemd) and how they define the system's operational state.
6. **Observation:** (Optional) If possible, observe the boot messages during a Linux system startup to identify key stages.

Source Code

(N/A - This lab is observational and conceptual, not involving direct coding.)

Input

(N/A - This lab does not require specific program input.)

Expected Output

A detailed explanation of the GNU/Linux booting process, potentially including a flowchart or diagram illustrating the sequence.

Lab 3: Multi-thread Programming

Title

Multi-thread Programming

Aim

To implement a program demonstrating multi-threading concepts, including thread creation, synchronization, and communication.

Procedure

1. **Choose a Language:** Select a programming language that supports multi-threading (e.g., C, C++, Java, Python).
2. **Problem Definition:** Define a simple problem that can benefit from multi-threading (e.g., parallel computation, concurrent task execution).
3. **Thread Creation:** Implement code to create multiple threads.
4. **Task Assignment:** Assign different parts of the problem to each thread.
5. **Synchronization (if needed):** If threads share resources, implement synchronization mechanisms (e.g., mutexes, semaphores, locks) to prevent race conditions.
6. **Thread Join:** Ensure the main thread waits for all child threads to complete their execution.
7. **Compile and Run:** Compile and execute the program.

Source Code

```
// Example: C program for simple multi-threading with pthreads
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> // For sleep

// Function to be executed by a thread
void *myThreadFunction(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello from thread #%ld!\n", tid);
    sleep(1); // Simulate some work
    printf("Thread #%ld exiting.\n", tid);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[3]; // Array to hold thread IDs
    int rc;
    long t;

    printf("Main: Creating threads...\n");

    for (t = 0; t < 3; t++) {
        printf("Main: Creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, myThreadFunction, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            return 1;
        }
    }

    printf("Main: All threads created. Waiting for them to finish...\n");
```

```
    // Wait for all threads to complete
    for (t = 0; t < 3; t++) {
        pthread_join(threads[t], NULL);
    }

    printf("Main: All threads finished. Exiting.\n");
    pthread_exit(NULL); // Exit the main thread gracefully
}
```

Input

(N/A - This example program does not require user input.)

Expected Output

```
Main: Creating threads...
Main: Creating thread 0
Main: Creating thread 1
Main: Creating thread 2
Main: All threads created. Waiting for them to finish...
Hello from thread #0!
Hello from thread #1!
Hello from thread #2!
Thread #0 exiting.
Thread #1 exiting.
Thread #2 exiting.
Main: All threads finished. Exiting.
```

(Note: The order of "Hello from thread" and "Thread exiting" messages might vary due to thread scheduling.)

Lab 4: Simulation of FCFS CPU scheduling algorithm

Title

Simulation of FCFS (First-Come, First-Served) CPU scheduling algorithm

Aim

To simulate the First-Come, First-Served (FCFS) CPU scheduling algorithm and calculate performance metrics like average waiting time and average turnaround time.

Procedure

1. **Data Structure:** Define a structure or class to represent a process, including attributes like process ID, arrival time, and burst time.
2. **Input:** Get the number of processes and their arrival and burst times from the user.
3. **Sorting:** (Implicit in FCFS) Processes are scheduled in the order of their arrival. If arrival times are the same, order by process ID.
4. **Calculation:**
   - **Completion Time:** For each process, calculate its completion time (start time + burst time).
   - **Turnaround Time:** Calculate turnaround time (completion time - arrival time).
   - **Waiting Time:** Calculate waiting time (turnaround time - burst time).
5. **Performance Metrics:** Calculate the average waiting time and average turnaround time for all processes.
6. **Output:** Display the results in a clear tabular format.

Source Code

```
// Example: C program for FCFS CPU scheduling simulation
#include <stdio.h>

struct Process {
    int pid; // Process ID
    int arrival_time;
    int burst_time;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

void calculate_times(struct Process p[], int n) {
    int current_time = 0;
    int total_waiting_time = 0;
    int total_turnaround_time = 0;


printf("\nPID\tArrival\tBurst\tStart\tCompletion\tTurnaround\tWaiting\n");
    printf("--------------------------------------------------------------
--------\n");

    for (int i = 0; i < n; i++) {
        // Start time is the maximum of current_time and process's
arrival_time
        p[i].start_time = (current_time > p[i].arrival_time) ? current_time :
p[i].arrival_time;
        p[i].completion_time = p[i].start_time + p[i].burst_time;
        p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
```

```
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;

        // Update current_time for the next process
        current_time = p[i].completion_time;

        total_waiting_time += p[i].waiting_time;
        total_turnaround_time += p[i].turnaround_time;

        printf("%d\t%d\t%d\t%d\t%d\t\t%d\t\t%d\n",
               p[i].pid, p[i].arrival_time, p[i].burst_time, p[i].start_time,
               p[i].completion_time, p[i].turnaround_time,
p[i].waiting_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;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Process %d (Arrival Time Burst Time): ", i + 1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    }

    // FCFS assumes processes are already sorted by arrival time.
    // If not, you would need to sort them here.
    // For simplicity, we assume input is given in arrival order.

    calculate_times(processes, n);

    return 0;
}
```

Input

```
Enter the number of processes: 3
Enter Arrival Time and Burst Time for each process:
Process 1 (Arrival Time Burst Time): 0 5
Process 2 (Arrival Time Burst Time): 1 3
Process 3 (Arrival Time Burst Time): 2 8
```

Expected Output

```
PID     Arrival Burst   Start   Completion      Turnaround      Waiting
--------------------------------------------------------------------------
1       0       5       0       5               5               0
2       1       3       5       8               7               4
3       2       8       8       16              14              6

Average Waiting Time: 3.33
Average Turnaround Time: 8.67
```

Lab 5: Priority CPU scheduling algorithm

Title

Priority CPU scheduling algorithm

Aim

To simulate the Priority CPU scheduling algorithm (non-preemptive) and calculate performance metrics like average waiting time and average turnaround time.

Procedure

1. **Data Structure:** Define a structure or class for a process, including process ID, arrival time, burst time, and priority. (Lower priority number usually means higher priority).
2. **Input:** Get the number of processes and their arrival time, burst time, and priority from the user.
3. **Sorting/Selection:** At each time step, select the process with the highest priority among those that have arrived and are not yet completed.
4. **Calculation:**
   - **Completion Time:** For each process, calculate its completion time.
   - **Turnaround Time:** Calculate turnaround time (completion time - arrival time).
   - **Waiting Time:** Calculate waiting time (turnaround time - burst time).
5. **Performance Metrics:** Calculate the average waiting time and average turnaround time.
6. **Output:** Display the results in a clear tabular format.

Source Code

```
// Example: C program for Non-Preemptive Priority CPU scheduling simulation
#include <stdio.h>
#include <limits.h> // For INT_MAX

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int is_completed; // Flag to track if process is completed
};

void find_avg_time(struct Process p[], int n) {
    int current_time = 0;
    int completed_processes = 0;
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    // Initialize all processes as not completed
    for (int i = 0; i < n; i++) {
        p[i].is_completed = 0;
    }


printf("\nPID\tArrival\tBurst\tPriority\tStart\tCompletion\tTurnaround\tWaiti
ng\n");
```

```
    printf("-----------------------------------------------------------------
-------------------------------\n");

    while (completed_processes < n) {
        int best_priority = INT_MAX;
        int selected_process_index = -1;

        // Find the process with the highest priority that has arrived and is
not completed
        for (int i = 0; i < n; i++) {
            if (p[i].arrival_time <= current_time && p[i].is_completed == 0)
{
                if (p[i].priority < best_priority) {
                    best_priority = p[i].priority;
                    selected_process_index = i;
                }
                // If priorities are same, FCFS among them (based on arrival
time)
                else if (p[i].priority == best_priority) {
                    if (p[i].arrival_time <
p[selected_process_index].arrival_time) {
                        selected_process_index = i;
                    }
                }
            }
        }

        if (selected_process_index == -1) {
            // No process is ready, increment time
            current_time++;
        } else {
            int i = selected_process_index;

            p[i].start_time = current_time;
            p[i].completion_time = p[i].start_time + p[i].burst_time;
            p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
            p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
            p[i].is_completed = 1;

            current_time = p[i].completion_time;
            completed_processes++;

            total_waiting_time += p[i].waiting_time;
            total_turnaround_time += p[i].turnaround_time;

            printf("%d\t%d\t%d\t%d\t\t%d\t%d\t\t%d\t\t%d\n",
                   p[i].pid, p[i].arrival_time, p[i].burst_time,
p[i].priority,
                   p[i].start_time, p[i].completion_time,
p[i].turnaround_time, p[i].waiting_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;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];
```

```
    printf("Enter Arrival Time, Burst Time, and Priority for each 
process:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Process %d (Arrival Time Burst Time Priority): ", i + 1);
        scanf("%d %d %d", &processes[i].arrival_time, 
&processes[i].burst_time, &processes[i].priority);
    }

    find_avg_time(processes, n);

    return 0;
}
```

Input

```
Enter the number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1 (Arrival Time Burst Time Priority): 0 6 2
Process 2 (Arrival Time Burst Time Priority): 1 8 1
Process 3 (Arrival Time Burst Time Priority): 2 7 3
Process 4 (Arrival Time Burst Time Priority): 3 3 0
```

Expected Output

```
PID     Arrival Burst   Priority        Start   Completion      Turnaround
Waiting
------------------------------------------------------------------------------
--------------------
4       3       3       0               3       6               3
0
2       1       8       1               6       14              13
5
1       0       6       2               14      20              20
14
3       2       7       3               20      27              25
18

Average Waiting Time: 9.25
Average Turnaround Time: 15.25
```

Lab 6: Simulation of Round Robin CPU scheduling algorithm

Title

Simulation of Round Robin CPU scheduling algorithm

Aim

To simulate the Round Robin (RR) CPU scheduling algorithm and calculate performance metrics like average waiting time and average turnaround time.

Procedure

1. **Data Structure:** Define a structure or class for a process, including process ID, arrival time, burst time, and remaining burst time.
2. **Input:** Get the number of processes, their arrival times, burst times, and the time quantum from the user.
3. **Ready Queue:** Implement a queue to manage processes ready for execution.
4. **Scheduling Logic:**
   - Maintain a `current_time` variable.
   - At each time step, add processes that have arrived to the ready queue.
   - If the ready queue is not empty, dequeue a process.
   - Execute the process for the time quantum or until its burst time is completed, whichever is shorter.
   - If the process completes, record its completion time, turnaround time, and waiting time.
   - If the process doesn't complete, enqueue it back to the ready queue.
   - Increment `current_time`.
5. **Performance Metrics:** Calculate the average waiting time and average turnaround time.
6. **Output:** Display the results in a clear tabular format.

Source Code

```
// Example: C program for Round Robin CPU scheduling simulation
#include <stdio.h>
#include <stdbool.h> // For boolean type

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_burst_time;
    int start_time; // First time process gets CPU
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

void find_avg_time(struct Process p[], int n, int quantum) {
    int current_time = 0;
    int completed_processes = 0;
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    // Array to keep track of whether a process has entered the ready queue
    bool in_ready_queue[n];
    for (int i = 0; i < n; i++) {
        p[i].remaining_burst_time = p[i].burst_time;
```

```
        p[i].start_time = -1; // Initialize start time
        in_ready_queue[i] = false;
    }

    // Simple queue implementation (using an array for simplicity)
    int ready_queue[n];
    int front = 0, rear = -1; // Queue pointers


printf("\nPID\tArrival\tBurst\tRemaining\tStart\tCompletion\tTurnaround\tWait
ing\n");
    printf("-----------------------------------------------------------------
--------------------------------\n");

    while (completed_processes < n) {
        // Add newly arrived processes to the ready queue
        for (int i = 0; i < n; i++) {
            if (p[i].arrival_time <= current_time && !in_ready_queue[i] &&
p[i].remaining_burst_time > 0) {
                rear = (rear + 1) % n; // Circular queue
                ready_queue[rear] = i;
                in_ready_queue[i] = true;
            }
        }

        if (front > rear && completed_processes < n) { // If queue is empty
and not all processes completed
            // No process is ready, increment time to the next arrival
            int next_arrival = -1;
            for(int i = 0; i < n; i++) {
                if(p[i].remaining_burst_time > 0) {
                    if(next_arrival == -1 || p[i].arrival_time <
next_arrival) {
                        next_arrival = p[i].arrival_time;
                    }
                }
            }
            if(next_arrival != -1 && current_time < next_arrival) {
                current_time = next_arrival;
            } else {
                current_time++; // Increment if no next arrival or already
past it
            }
            continue; // Go to next iteration to check for new arrivals
        }

        int current_process_index = ready_queue[front];
        front = (front + 1) % n; // Dequeue

        if (p[current_process_index].start_time == -1) {
            p[current_process_index].start_time = current_time;
        }

        int execution_time = (p[current_process_index].remaining_burst_time <
quantum) ?
                             p[current_process_index].remaining_burst_time :
quantum;

        current_time += execution_time;
        p[current_process_index].remaining_burst_time -= execution_time;

        // Add newly arrived processes during this quantum to the ready queue
        for (int i = 0; i < n; i++) {
            if (p[i].arrival_time <= current_time && !in_ready_queue[i] &&
p[i].remaining_burst_time > 0) {
                rear = (rear + 1) % n;
```

```
                ready_queue[rear] = i;
                in_ready_queue[i] = true;
            }
        }

        if (p[current_process_index].remaining_burst_time == 0) {
            p[current_process_index].completion_time = current_time;
            p[current_process_index].turnaround_time =
p[current_process_index].completion_time -
p[current_process_index].arrival_time;
            p[current_process_index].waiting_time =
p[current_process_index].turnaround_time -
p[current_process_index].burst_time;
            completed_processes++;

            total_waiting_time += p[current_process_index].waiting_time;
            total_turnaround_time +=
p[current_process_index].turnaround_time;

            printf("%d\t%d\t%d\t%d\t\t%d\t%d\t\t%d\t\t%d\n",
                   p[current_process_index].pid,
p[current_process_index].arrival_time, p[current_process_index].burst_time,
                   p[current_process_index].remaining_burst_time,
p[current_process_index].start_time,
                   p[current_process_index].completion_time,
p[current_process_index].turnaround_time,
                   p[current_process_index].waiting_time);
        } else {
            // Process not completed, add back to ready queue
            rear = (rear + 1) % n;
            ready_queue[rear] = current_process_index;
        }
    }

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

    struct Process processes[n];

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Process %d (Arrival Time Burst Time): ", i + 1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    }

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

    find_avg_time(processes, n, quantum);

    return 0;
}
```

Input

```
Enter the number of processes: 3
Enter Arrival Time and Burst Time for each process:
```

```
Process 1 (Arrival Time Burst Time): 0 10
Process 2 (Arrival Time Burst Time): 1 4
Process 3 (Arrival Time Burst Time): 2 6
Enter the time quantum: 3
```

Expected Output

```
PID     Arrival Burst   Remaining       Start   Completion      Turnaround
Waiting
------------------------------------------------------------------------------
--------------------
2       1       4       0               1       7               6
2
3       2       6       0               7       15              13
7
1       0       10      0               0       20              20
10

Average Waiting Time: 6.33
Average Turnaround Time: 13.00
```

Lab 7: Write a procedure for timer interrupt handler

Title

Procedure for Timer Interrupt Handler

Aim

To understand the role and implementation aspects of a timer interrupt handler in an operating system context.

Procedure

1. **Concept of Timer Interrupts:** Explain what a timer interrupt is, its purpose (e.g., time slicing for CPU scheduling, system clock updates), and how it's generated by hardware.
2. **Interrupt Vector Table (IVT)/Interrupt Descriptor Table (IDT):** Describe how the operating system sets up the IVT/IDT to map interrupt numbers to specific interrupt service routines (ISRs).
3. **Timer ISR Flow:** Outline the typical steps involved when a timer interrupt occurs:
   - CPU saves context of the currently running process.
   - CPU jumps to the timer interrupt handler's address.
   - **Handler's Responsibilities:**
     - Acknowledge the interrupt to the Programmable Interrupt Controller (PIC).
     - Update system time/tick count.
     - Decrement process time slice (if used for scheduling).
     - If time slice expires, trigger the scheduler.
     - Perform any other periodic tasks.
   - CPU restores context of the next process (or the interrupted process if not switched).
   - CPU returns from interrupt.
4. **Pseudocode/Conceptual Code:** Provide pseudocode or a conceptual C-like structure for a timer interrupt handler.
5. **Kernel Context:** Discuss how the handler operates in kernel mode and its implications for system stability.

Source Code

```
// Pseudocode for a conceptual Timer Interrupt Handler

// Global variables (part of OS kernel data)
unsigned long system_ticks = 0;
int current_process_timeslice = 0;
const int TIME_QUANTUM = 10; // Example time quantum for scheduling

// Function prototypes (assuming they exist in the kernel)
void save_cpu_context(void *context_ptr);
void restore_cpu_context(void *context_ptr);
void acknowledge_pic_interrupt(int irq_number);
void schedule_next_process(); // Function to invoke the scheduler
void *get_current_process_context();
void *get_next_process_context();

// The Timer Interrupt Service Routine (ISR)
// This function is invoked by the CPU when a timer interrupt (e.g., IRQ0)
occurs.
void timer_interrupt_handler() {
```

```
    // 1. Save the context of the currently executing process/thread
    //    (Registers, program counter, stack pointer, etc.)
    //    This is often done by hardware or assembly stub before calling this
C function.
    //    For conceptual understanding, we assume a function does this.
    // save_cpu_context(get_current_process_context()); // Conceptual

    // 2. Acknowledge the interrupt to the Programmable Interrupt Controller
(PIC)
    //    This tells the PIC that the interrupt has been handled and it can
send more.
    acknowledge_pic_interrupt(0); // Assuming timer uses IRQ0

    // 3. Update system time/tick count
    system_ticks++;

    // 4. Handle CPU scheduling (time slicing)
    current_process_timeslice--;
    if (current_process_timeslice <= 0) {
        // Time slice for the current process has expired
        // Trigger the scheduler to pick the next process
        schedule_next_process();
        // Reset time slice for the new process
        current_process_timeslice = TIME_QUANTUM;
    }

    // 5. Perform other periodic tasks (e.g., updating sleep timers, flushing
buffers)
    //    ... (other kernel tasks) ...

    // 6. Restore the context of the next process to be executed
    //    (If scheduler switched processes, restore the new one; otherwise,
restore the original)
    //    This is also often done by hardware or assembly stub after this C
function returns.
    // restore_cpu_context(get_next_process_context()); // Conceptual
}

// Example of how the timer might be initialized (conceptual)
void initialize_timer() {
    // Set up the timer hardware to generate interrupts periodically
    // (e.g., configure Programmable Interval Timer - PIT)
    // Register the timer_interrupt_handler with the Interrupt Descriptor
Table (IDT)
    // so that the CPU knows where to jump when IRQ0 occurs.
    // current_process_timeslice = TIME_QUANTUM; // Initialize for first
process
}
```

Input

(N/A - This lab is theoretical and conceptual, not requiring direct input.)

Expected Output

A detailed written procedure and conceptual pseudocode outlining the steps and responsibilities of a timer interrupt handler within an operating system.

Lab 8: Classical inter process communication problem (Producer consumer)

Title

Classical Inter-Process Communication Problem: Producer-Consumer

Aim

To implement a solution to the Producer-Consumer problem using inter-process communication (IPC) mechanisms like semaphores and shared memory (or mutexes and condition variables for threads).

Procedure

1. **Problem Definition:** Understand the Producer-Consumer problem: a producer generates data and puts it into a buffer, and a consumer takes data from the buffer. The buffer has a finite size.
2. **IPC Mechanisms:** Choose appropriate IPC mechanisms:
   - **For Processes:** Shared memory for the buffer, and semaphores (e.g., `empty`, `full`, `mutex`) for synchronization.
   - **For Threads:** A shared buffer, and mutexes and condition variables for synchronization.
3. **Buffer Implementation:** Create a shared buffer (e.g., a circular array).
4. **Producer Logic:**
   - Generate an item.
   - Wait if the buffer is full (using `empty` semaphore or condition variable).
   - Acquire mutex (or `mutex` semaphore).
   - Add item to buffer.
   - Release mutex.
   - Signal that the buffer is not empty (using `full` semaphore or condition variable).
5. **Consumer Logic:**
   - Wait if the buffer is empty (using `full` semaphore or condition variable).
   - Acquire mutex (or `mutex` semaphore).
   - Remove item from buffer.
   - Release mutex.
   - Signal that the buffer is not full (using `empty` semaphore or condition variable).
6. **Compile and Run:** Compile the producer and consumer programs (or a single program with producer/consumer threads) and observe their synchronized execution.

Source Code

```
// Example: C program for Producer-Consumer problem using pthreads, mutexes,
and condition variables

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h> // For sleep

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;  // Index for producer
int out = 0; // Index for consumer
```

```
int count = 0; // Number of items in buffer

pthread_mutex_t mutex; // Mutex for critical section
pthread_cond_t full;   // Condition variable for consumer (buffer is full)
pthread_cond_t empty;  // Condition variable for producer (buffer is empty)

void *producer(void *param) {
    int item;
    for (int i = 0; i < 10; i++) { // Produce 10 items
        item = rand() % 100; // Generate a random item

        pthread_mutex_lock(&mutex); // Acquire mutex

        while (count == BUFFER_SIZE) { // If buffer is full, wait
            printf("Producer: Buffer is full. Waiting...\n");
            pthread_cond_wait(&empty, &mutex);
        }

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        printf("Producer: Produced item %d. Buffer count: %d\n", item,
count);

        pthread_cond_signal(&full); // Signal consumer that buffer is not
empty
        pthread_mutex_unlock(&mutex); // Release mutex
        sleep(rand() % 2); // Simulate production time
    }
    pthread_exit(0);
}

void *consumer(void *param) {
    int item;
    for (int i = 0; i < 10; i++) { // Consume 10 items
        pthread_mutex_lock(&mutex); // Acquire mutex

        while (count == 0) { // If buffer is empty, wait
            printf("Consumer: Buffer is empty. Waiting...\n");
            pthread_cond_wait(&full, &mutex);
        }

        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        printf("Consumer: Consumed item %d. Buffer count: %d\n", item,
count);

        pthread_cond_signal(&empty); // Signal producer that buffer is not
full
        pthread_mutex_unlock(&mutex); // Release mutex
        sleep(rand() % 3); // Simulate consumption time
    }
    pthread_exit(0);
}

int main() {
    pthread_t prod_tid, cons_tid;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&full, NULL);
    pthread_cond_init(&empty, NULL);

    pthread_create(&prod_tid, NULL, producer, NULL);
    pthread_create(&cons_tid, NULL, consumer, NULL);
```

```
    pthread_join(prod_tid, NULL);
    pthread_join(cons_tid, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&full);
    pthread_cond_destroy(&empty);

    printf("\nProducer-Consumer simulation finished.\n");
    return 0;
}
```

Input

(N/A - This program does not require user input.)

Expected Output

(Output will vary slightly due to thread scheduling and random delays, but will follow this pattern:)

```
Producer: Produced item 84. Buffer count: 1
Consumer: Consumed item 84. Buffer count: 0
Producer: Produced item 75. Buffer count: 1
Producer: Produced item 28. Buffer count: 2
Consumer: Consumed item 75. Buffer count: 1
Producer: Produced item 7. Buffer count: 2
Consumer: Consumed item 28. Buffer count: 1
Producer: Produced item 90. Buffer count: 2
Producer: Produced item 16. Buffer count: 3
Consumer: Consumed item 7. Buffer count: 2
Producer: Produced item 12. Buffer count: 3
Producer: Produced item 34. Buffer count: 4
Producer: Produced item 56. Buffer count: 5
Producer: Buffer is full. Waiting...
Consumer: Consumed item 90. Buffer count: 4
Producer: Produced item 93. Buffer count: 5
Producer: Buffer is full. Waiting...
Consumer: Consumed item 16. Buffer count: 4
Consumer: Consumed item 12. Buffer count: 3
Consumer: Consumed item 34. Buffer count: 2
Consumer: Consumed item 56. Buffer count: 1
Consumer: Consumed item 93. Buffer count: 0

Producer-Consumer simulation finished.
```

Lab 9: Write a procedure to make message passing in inter process communication

Title

Procedure for Message Passing in Inter-Process Communication

Aim

To understand and implement a basic message passing mechanism for inter-process communication (IPC) using system calls like `msgget`, `msgsnd`, and `msgrcv` (in Unix-like systems).

Procedure

1. **Concept of Message Queues:** Explain message queues as a linked list of messages stored within the kernel, allowing processes to send and receive messages.
2. **System Calls:** Describe the purpose of key system calls:
   - `msgget()`: Creates a new message queue or gets an ID of an existing one.
   - `msgsnd()`: Sends a message to a message queue.
   - `msgrcv()`: Receives a message from a message queue.
   - `msgctl()`: Performs control operations on a message queue (e.g., delete).
3. **Message Structure:** Define a structure for the messages to be sent, including a `mtype` (message type) and `mtext` (message text/data).
4. **Sender Process Logic:**
   - Get/create message queue ID using `msgget()`.
   - Prepare the message structure.
   - Send the message using `msgsnd()`.
   - (Optional) Delete the message queue using `msgctl()` after communication is done.
5. **Receiver Process Logic:**
   - Get message queue ID using `msgget()`.
   - Receive the message using `msgrcv()`, specifying the desired message type.
   - Process the received message.
   - (Optional) Delete the message queue using `msgctl()` after communication is done.
6. **Compile and Run:** Compile the sender and receiver programs separately and run them concurrently to observe message exchange.

Source Code

```
// Example: C program for Message Passing (Sender) using System V IPC

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

// Define a structure for the message
struct message_buffer {
    long mtype; // Message type (must be > 0)
    char mtext[100]; // Message data
};
```

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

    // 1. Generate a unique key for the message queue
    // ftok(pathname, proj_id) generates a System V IPC key
    key = ftok("progfile", 65); // "progfile" can be any existing file

    // 2. Create or get the message queue ID
    // msgget(key, msgflg) returns the message queue identifier
    // IPC_CREAT: Create a new queue if it doesn't exist
    // 0666: Read/write permissions for owner, group, others
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget failed");
        exit(EXIT_FAILURE);
    }

    printf("Message Queue ID: %d\n", msgid);

    // 3. Prepare the message to be sent
    message.mtype = 1; // Message type 1
    strcpy(message.mtext, "Hello from Sender!");

    // 4. Send the message to the message queue
    // msgsnd(msqid, msgp, msgsz, msgflg)
    // msqid: Message queue ID
    // msgp: Pointer to the message buffer
    // msgsz: Size of the message text (mtext)
    // msgflg: Flags (e.g., IPC_NOWAIT)
    if (msgsnd(msgid, &message, sizeof(message.mtext), 0) == -1) {
        perror("msgsnd failed");
        exit(EXIT_FAILURE);
    }

    printf("Message sent: %s\n", message.mtext);

    // Optional: Delete the message queue after sending
    // msgctl(msqid, cmd, buf)
    // cmd: IPC_RMID to remove the queue
    // if (msgctl(msgid, IPC_RMID, NULL) == -1) {
    //     perror("msgctl (IPC_RMID) failed");
    //     exit(EXIT_FAILURE);
    // }
    // printf("Message queue removed.\n");

    return 0;
}

// Example: C program for Message Passing (Receiver) using System V IPC

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

// Define a structure for the message (must be same as sender)
struct message_buffer {
    long mtype;
    char mtext[100];
};
```

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

    // 1. Generate the same unique key as the sender
    key = ftok("progfile", 65);

    // 2. Get the message queue ID (it must already exist)
    msgid = msgget(key, 0666); // No IPC_CREAT here, just get existing
    if (msgid == -1) {
        perror("msgget failed (queue might not exist or permissions are
wrong)");
        exit(EXIT_FAILURE);
    }

    printf("Message Queue ID: %d\n", msgid);

    // 3. Receive the message from the message queue
    // msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)
    // msgtyp: 0 for first message, >0 for specific type, <0 for first
message <= abs(msgtyp)
    if (msgrcv(msgid, &message, sizeof(message.mtext), 1, 0) == -1) { //
Receive message of type 1
        perror("msgrcv failed");
        exit(EXIT_FAILURE);
    }

    printf("Message received: %s\n", message.mtext);

    // Optional: Delete the message queue after receiving
    // if (msgctl(msgid, IPC_RMID, NULL) == -1) {
    //     perror("msgctl (IPC_RMID) failed");
    //     exit(EXIT_FAILURE);
    // }
    // printf("Message queue removed.\n");

    return 0;
}
```

Input

(Run the sender program first, then the receiver program. No direct input is given to the programs themselves.)

Expected Output

For Sender Program:

```
Message Queue ID: <some_integer_id>
Message sent: Hello from Sender!
```

For Receiver Program:

```
Message Queue ID: <same_integer_id>
Message received: Hello from Sender!
```

Lab 10: Program to implement Bankers Algorithm

Title

Program to implement Banker's Algorithm

Aim

To implement the Banker's Algorithm for deadlock avoidance and determine if a system is in a safe state.

Procedure

1. **Algorithm Concept:** Understand the Banker's Algorithm, which checks for a safe state by simulating the allocation of resources to processes.
2. **Data Structures:** Define arrays/matrices to store:
   - `Available`: Number of available instances of each resource type.
   - `Max`: Maximum demand of each resource type for each process.
   - `Allocation`: Number of resources of each type currently allocated to each process.
   - `Need`: Remaining resources needed by each process (`Need = Max - Allocation`).
3. **Input:** Get the number of processes, number of resource types, and the initial values for `Available`, `Max`, and `Allocation` matrices.
4. **Safety Algorithm Implementation:**
   - Initialize `Work = Available` and `Finish[i] = false` for all processes `i`.
   - Find a process `i` such that `Finish[i] == false` and `Need[i] <= Work`.
   - If such a process is found:
     - `Work = Work + Allocation[i]`
     - `Finish[i] = true`
     - Repeat step 2.
   - If no such process is found, check if `Finish[i] == true` for all `i`. If yes, the system is in a safe state; otherwise, it's in an unsafe state.
5. **Output:** Display whether the system is in a safe state and, if so, print a safe sequence of processes.

Source Code

```
// Example: C program to implement Banker's Algorithm

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int n_processes; // Number of processes
int n_resources; // Number of resource types

int available[MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

void calculate_need() {
    for (int i = 0; i < n_processes; i++) {
        for (int j = 0; j < n_resources; j++) {
```

```
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

bool is_safe() {
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES];
    int safe_sequence[MAX_PROCESSES];
    int count = 0;

    // Initialize work and finish
    for (int j = 0; j < n_resources; j++) {
        work[j] = available[j];
    }
    for (int i = 0; i < n_processes; i++) {
        finish[i] = false;
    }

    int iter_count = 0; // To prevent infinite loops in case of no safe
sequence

    // Find a process that can be executed
    while (count < n_processes && iter_count < n_processes * n_processes) {
// Added iter_count for safety
        bool found_process = false;
        for (int p = 0; p < n_processes; p++) {
            if (!finish[p]) {
                bool can_execute = true;
                for (int r = 0; r < n_resources; r++) {
                    if (need[p][r] > work[r]) {
                        can_execute = false;
                        break;
                    }
                }

                if (can_execute) {
                    // Execute process p
                    for (int r = 0; r < n_resources; r++) {
                        work[r] += allocation[p][r];
                    }
                    finish[p] = true;
                    safe_sequence[count++] = p;
                    found_process = true;
                    break; // Found one, restart search for next
                }
            }
        }

        if (!found_process) {
            // No process found that can be executed
            break;
        }
        iter_count++;
    }

    if (count == n_processes) {
        printf("\nSystem is in a SAFE STATE.\n");
        printf("Safe Sequence: < ");
        for (int i = 0; i < n_processes; i++) {
            printf("P%d ", safe_sequence[i]);
        }
        printf(">\n");
        return true;
    } else {
        printf("\nSystem is in an UNSAFE STATE.\n");
```

```
        return false;
    }
}

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

    printf("\nEnter Available resources:\n");
    for (int j = 0; j < n_resources; j++) {
        printf("Resource %d: ", j);
        scanf("%d", &available[j]);
    }

    printf("\nEnter Max matrix:\n");
    for (int i = 0; i < n_processes; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < n_resources; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    printf("\nEnter Allocation matrix:\n");
    for (int i = 0; i < n_processes; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < n_resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    calculate_need();

    printf("\nNeed Matrix:\n");
    for (int i = 0; i < n_processes; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < n_resources; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    is_safe();

    return 0;
}
```

Input

```
Enter the number of processes: 5
Enter the number of resource types: 3

Enter Available resources:
Resource 0: 3
Resource 1: 3
Resource 2: 2

Enter Max matrix:
Process 0: 7 5 3
Process 1: 3 2 2
Process 2: 9 0 2
Process 3: 2 2 2
Process 4: 4 3 3

Enter Allocation matrix:
```

```
Process 0: 0 1 0
Process 1: 2 0 0
Process 2: 3 0 2
Process 3: 2 1 1
Process 4: 0 0 2
```

Expected 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 a SAFE STATE.
Safe Sequence: < P1 P3 P4 P0 P2 >
```

Lab 11: Program to implement memory allocation with pages

Title

Program to implement Memory Allocation with Pages (Paging Simulation)

Aim

To simulate a basic memory allocation scheme using paging, demonstrating how logical addresses are translated to physical addresses.

Procedure

1. **Paging Concept:** Understand paging: dividing logical memory into fixed-size blocks called pages and physical memory into fixed-size blocks called frames.
2. **Page Table:** Implement a page table data structure, which maps page numbers to frame numbers.
3. **Input:**
   - Page size.
   - Number of pages in logical address space.
   - Mapping of logical pages to physical frames (the page table entries).
   - A logical address to translate.
4. **Address Translation Logic:**
   - Given a logical address, calculate the page number and offset within the page.
   - Use the page table to find the corresponding frame number for the calculated page number.
   - Calculate the physical address: `Physical Address = (Frame Number * Page Size) + Offset`.
5. **Output:** Display the page number, offset, frame number, and the resulting physical address. Handle cases where the logical address is invalid (e.g., page not found, address out of bounds).

Source Code

```
// Example: C program for Paging Simulation

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

#define MAX_PAGES 10
#define MAX_FRAMES 10

// Page Table Entry: maps logical page to physical frame
struct PageTableEntry {
    int page_number;
    int frame_number;
    int valid_bit; // 1 if valid, 0 if invalid/not in memory
};

int main() {
    int page_size;
    int num_logical_pages;
    int num_physical_frames;
    struct PageTableEntry page_table[MAX_PAGES];

    printf("Enter page size (e.g., 4096 bytes): ");
    scanf("%d", &page_size);
```

```
    printf("Enter number of logical pages: ");
    scanf("%d", &num_logical_pages);

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

    // Initialize page table (assume no pages are mapped initially)
    for (int i = 0; i < num_logical_pages; i++) {
        page_table[i].page_number = i;
        page_table[i].frame_number = -1; // -1 indicates not mapped
        page_table[i].valid_bit = 0;
    }

    printf("\nEnter page to frame mappings (Logical Page Number Physical
Frame Number, -1 -1 to stop):\n");
    int lp, pf;
    while (1) {
        printf("Mapping (LP PF): ");
        scanf("%d %d", &lp, &pf);
        if (lp == -1 && pf == -1) {
            break;
        }
        if (lp >= 0 && lp < num_logical_pages && pf >= 0 && pf <
num_physical_frames) {
            page_table[lp].frame_number = pf;
            page_table[lp].valid_bit = 1;
        } else {
            printf("Invalid page or frame number. Please re-enter.\n");
        }
    }

    printf("\n--- Page Table ---\n");
    printf("Page No.\tFrame No.\tValid Bit\n");
    for (int i = 0; i < num_logical_pages; i++) {
        printf("%d\t\t%d\t\t%d\n", page_table[i].page_number,
page_table[i].frame_number, page_table[i].valid_bit);
    }

    int logical_address;
    printf("\nEnter a logical address to translate (-1 to exit): ");
    while (scanf("%d", &logical_address) == 1 && logical_address != -1) {
        if (logical_address < 0 || logical_address >= (num_logical_pages *
page_size)) {
            printf("Error: Logical address out of bounds.\n");
        } else {
            int page_number = logical_address / page_size;
            int offset = logical_address % page_size;

            if (page_number >= num_logical_pages ||
page_table[page_number].valid_bit == 0) {
                printf("Error: Page %d is not in memory or invalid.\n",
page_number);
            } else {
                int frame_number = page_table[page_number].frame_number;
                int physical_address = (frame_number * page_size) + offset;

                printf("Logical Address: %d\n", logical_address);
                printf("  Page Number: %d\n", page_number);
                printf("  Offset: %d\n", offset);
                printf("  Frame Number: %d\n", frame_number);
                printf("Physical Address: %d\n", physical_address);
            }
        }
        printf("\nEnter a logical address to translate (-1 to exit): ");
    }
```

```
    return 0;
}
```

Input

```
Enter page size (e.g., 4096 bytes): 100
Enter number of logical pages: 5
Enter number of physical frames: 3

Enter page to frame mappings (Logical Page Number Physical Frame Number, -1 -
1 to stop):
Mapping (LP PF): 0 2
Mapping (LP PF): 1 0
Mapping (LP PF): 3 1
Mapping (LP PF): -1 -1

Enter a logical address to translate (-1 to exit): 50
Enter a logical address to translate (-1 to exit): 120
Enter a logical address to translate (-1 to exit): 380
Enter a logical address to translate (-1 to exit): 450
Enter a logical address to translate (-1 to exit): 250
Enter a logical address to translate (-1 to exit): -1
```

Expected Output

```
--- Page Table ---
Page No.        Frame No.       Valid Bit
0               2               1
1               0               1
2               -1              0
3               1               1
4               -1              0

Enter a logical address to translate (-1 to exit): 50
Logical Address: 50
  Page Number: 0
  Offset: 50
  Frame Number: 2
Physical Address: 250

Enter a logical address to translate (-1 to exit): 120
Logical Address: 120
  Page Number: 1
  Offset: 20
  Frame Number: 0
Physical Address: 20

Enter a logical address to translate (-1 to exit): 380
Logical Address: 380
  Page Number: 3
  Offset: 80
  Frame Number: 1
Physical Address: 180

Enter a logical address to translate (-1 to exit): 450
Error: Page 4 is not in memory or invalid.

Enter a logical address to translate (-1 to exit): 250
Error: Page 2 is not in memory or invalid.

Enter a logical address to translate (-1 to exit): -1
```

Lab 12: Simulation of FIFO page replacement algorithm

Title

Simulation of FIFO (First-In, First-Out) Page Replacement Algorithm

Aim

To simulate the First-In, First-Out (FIFO) page replacement algorithm and calculate the number of page faults for a given page reference string and a fixed number of frames.

Procedure

1. **FIFO Concept:** Understand FIFO: when a page fault occurs and the memory is full, the oldest page (the one that has been in memory the longest) is replaced.
2. **Data Structures:**
   - An array or list to represent the main memory frames.
   - A queue to keep track of the order in which pages entered the frames (for FIFO replacement).
3. **Input:**
   - Number of available frames in physical memory.
   - A page reference string (sequence of page numbers).
4. **Simulation Logic:**
   - Iterate through the page reference string.
   - For each page:
     - Check if the page is already in a frame (page hit). If yes, continue.
     - If not (page fault):
       - Increment page fault count.
       - If frames are available, add the page to an empty frame and enqueue it.
       - If frames are full, dequeue the oldest page from the queue, replace it in the corresponding frame with the new page, and enqueue the new page.
5. **Output:** Display the state of frames at each step, indicating page hits/faults, and the total number of page faults.

Source Code

```
// Example: C program for FIFO Page Replacement Algorithm Simulation

#include <stdio.h>
#include <stdbool.h>
#include <string.h> // For memset

#define MAX_PAGES 20
#define MAX_FRAMES 10

int main() {
    int page_reference_string[MAX_PAGES];
    int num_pages;
    int num_frames;
    int frames[MAX_FRAMES]; // Represents physical memory frames
    int page_faults = 0;
    int front = 0; // For FIFO queue logic (index of oldest page in frames)
    int rear = -1; // For FIFO queue logic (index where new page would be
added)
    int current_frame_count = 0; // How many frames are currently occupied
```

```
    printf("Enter the number of frames: ");
    scanf("%d", &num_frames);

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &num_pages);

    printf("Enter the page reference string (space separated): ");
    for (int i = 0; i < num_pages; i++) {
        scanf("%d", &page_reference_string[i]);
    }

    // Initialize frames with -1 (empty)
    memset(frames, -1, sizeof(frames));

    printf("\nPage Reference String\tFrames\t\t\tHit/Fault\n");
    printf("--------------------------------------------------------------
\n");

    for (int i = 0; i < num_pages; i++) {
        int current_page = page_reference_string[i];
        bool is_hit = false;

        // Check if page is already in frames (page hit)
        for (int j = 0; j < num_frames; j++) {
            if (frames[j] == current_page) {
                is_hit = true;
                break;
            }
        }

        printf("%d\t\t\t", current_page);

        if (is_hit) {
            printf(" (Hit)\n");
        } else { // Page Fault
            page_faults++;
            printf(" (Fault)\n");

            if (current_frame_count < num_frames) {
                // Frames are not full, add to an empty frame
                frames[rear + 1] = current_page;
                rear = (rear + 1); // Update rear for next insertion
                current_frame_count++;
            } else {
                // Frames are full, replace the oldest page (FIFO)
                frames[front] = current_page; // Replace the page at 'front'
                front = (front + 1) % num_frames; // Move front to next
oldest
                rear = (rear + 1) % num_frames; // Move rear to the newly
inserted position
            }
        }

        // Print current state of frames
        printf("\t\t\t");
        for (int j = 0; j < num_frames; j++) {
            if (frames[j] == -1) {
                printf(" - ");
            } else {
                printf(" %d ", frames[j]);
            }
        }
        printf("\n");
    }
```

```
    printf("\nTotal Page Faults: %d\n", page_faults);

    return 0;
}
```

Input

```
Enter the number of frames: 3
Enter the number of pages in the reference string: 12
Enter the page reference string (space separated): 0 1 2 3 0 1 4 0 1 2 3 4
```

Expected Output

```
Page Reference String     Frames                    Hit/Fault
----------------------------------------------------------------
0                        (Fault)
                          0  -  -
1                        (Fault)
                          0  1  -
2                        (Fault)
                          0  1  2
3                        (Fault)
                          3  1  2
0                        (Fault)
                          3  0  2
1                        (Fault)
                          3  0  1
4                        (Fault)
                          4  0  1
0                        (Hit)
                          4  0  1
1                        (Hit)
                          4  0  1
2                        (Fault)
                          4  2  1
3                        (Fault)
                          3  2  1
4                        (Fault)
                          3  2  4

Total Page Faults: 10
```

Lab 13: multiple partition (dynamic Memory allocation method)

Title

Multiple Partition (Dynamic Memory Allocation Method) Simulation

Aim

To simulate dynamic memory allocation using multiple partitions, specifically focusing on the First Fit, Best Fit, and Worst Fit algorithms.

Procedure

1. **Dynamic Allocation Concept:** Understand how dynamic partitioning allows memory to be divided into variable-sized partitions as needed by processes.
2. **Algorithms:**
   - **First Fit:** Allocate the first hole that is big enough.
   - **Best Fit:** Allocate the smallest hole that is big enough.
   - **Worst Fit:** Allocate the largest hole that is big enough.
3. **Data Structures:** Represent memory as a list of holes (free blocks) and allocated blocks. Each block should have a starting address and size.
4. **Input:**
   - Initial memory partitions (free blocks).
   - Sequence of process requests (process ID, size).
   - Choice of allocation algorithm (First Fit, Best Fit, Worst Fit).
5. **Allocation Logic (for each algorithm):**
   - When a process requests memory:
     - Search for a suitable hole according to the chosen algorithm.
     - If a hole is found:
       - Allocate the requested size from the hole.
       - If the hole is larger than needed, split it, leaving a smaller free hole.
       - Mark the allocated block.
     - If no suitable hole is found, the process cannot be allocated.
6. **Deallocation Logic:** When a process finishes, its memory block becomes a free hole. Implement merging of adjacent free holes to prevent fragmentation.
7. **Output:** Display the memory state (allocated and free blocks) after each allocation/deallocation, and indicate if a process could not be allocated.

Source Code

```
// Example: C program for Dynamic Memory Allocation Simulation (First Fit)

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

#define MAX_BLOCKS 10

// Structure to represent a memory block (either free or allocated)
struct MemoryBlock {
    int id; // -1 for free block, process ID for allocated block
    int start_address;
    int size;
};
```

```
// Global array to simulate memory blocks
struct MemoryBlock memory[MAX_BLOCKS];
int num_memory_blocks = 0; // Current number of blocks in memory array

// Function to display current memory state
void display_memory() {
    printf("\n--- Current Memory State ---\n");
    printf("ID\tStart\tSize\tStatus\n");
    printf("------------------------------------\n");
    for (int i = 0; i < num_memory_blocks; i++) {
        printf("%d\t%d\t%d\t%s\n",
               memory[i].id,
               memory[i].start_address,
               memory[i].size,
               (memory[i].id == -1) ? "FREE" : "ALLOCATED (P)");
    }
    printf("------------------------------------\n");
}

// Function to merge adjacent free blocks (to reduce fragmentation)
void merge_free_blocks() {
    // A more robust merge would sort by start_address first.
    // For simplicity, this assumes blocks are generally ordered.
    for (int i = 0; i < num_memory_blocks - 1; i++) {
        if (memory[i].id == -1 && memory[i+1].id == -1) {
            // Merge memory[i+1] into memory[i]
            memory[i].size += memory[i+1].size;
            // Shift subsequent blocks to fill the gap
            for (int j = i + 1; j < num_memory_blocks - 1; j++) {
                memory[j] = memory[j+1];
            }
            num_memory_blocks--;
            i--; // Recheck the current block as it might merge again
        }
    }
}

// First Fit Allocation
void first_fit(int process_id, int request_size) {
    int allocated = -1;
    for (int i = 0; i < num_memory_blocks; i++) {
        if (memory[i].id == -1 && memory[i].size >= request_size) {
            // Found a suitable free block
            if (memory[i].size == request_size) {
                // Exact fit
                memory[i].id = process_id;
            } else {
                // Split the block
                // Create a new allocated block
                for (int j = num_memory_blocks; j > i + 1; j--) {
                    memory[j] = memory[j-1];
                }
                memory[i+1].id = -1; // New free block
                memory[i+1].start_address = memory[i].start_address +
request_size;
                memory[i+1].size = memory[i].size - request_size;

                // Update the original block to be allocated
                memory[i].id = process_id;
                memory[i].size = request_size;
                num_memory_blocks++;
            }
            allocated = i;
            printf("Process P%d allocated %d units using First Fit.\n",
process_id, request_size);
            break;
```

```c
        }
    }

    if (allocated == -1) {
        printf("Process P%d (size %d) could not be allocated (No suitable
block found).\n", process_id, request_size);
    }
}

// Deallocation
void deallocate(int process_id) {
    bool found = false;
    for (int i = 0; i < num_memory_blocks; i++) {
        if (memory[i].id == process_id) {
            memory[i].id = -1; // Mark as free
            printf("Process P%d deallocated.\n", process_id);
            found = true;
            merge_free_blocks(); // Attempt to merge adjacent free blocks
            break;
        }
    }
    if (!found) {
        printf("Process P%d not found for deallocation.\n", process_id);
    }
}

int main() {
    int total_memory_size;
    printf("Enter total memory size: ");
    scanf("%d", &total_memory_size);

    // Initially, all memory is one large free block
    memory[0].id = -1;
    memory[0].start_address = 0;
    memory[0].size = total_memory_size;
    num_memory_blocks = 1;

    display_memory();

    int choice;
    int pid, size;

    while (1) {
        printf("\n1. Allocate (First Fit)\n");
        printf("2. Deallocate\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter Process ID and size to allocate: ");
                scanf("%d %d", &pid, &size);
                first_fit(pid, size);
                display_memory();
                break;
            case 2:
                printf("Enter Process ID to deallocate: ");
                scanf("%d", &pid);
                deallocate(pid);
                display_memory();
                break;
            case 3:
                printf("Exiting.\n");
                exit(0);
            default:
```

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

    return 0;
}
```

Input

```
Enter total memory size: 1000

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 101 200

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 102 150

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 103 300

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 2
Enter Process ID to deallocate: 102

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 104 100

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 3
```

Expected Output

```
Enter total memory size: 1000

--- Current Memory State ---
ID      Start   Size    Status
------------------------------------
-1      0       1000    FREE
------------------------------------

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 101 200
Process P101 allocated 200 units using First Fit.

--- Current Memory State ---
ID      Start   Size    Status
```

```
------------------------------------
101     0       200     ALLOCATED (P)
-1      200     800     FREE
------------------------------------

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 102 150
Process P102 allocated 150 units using First Fit.

--- Current Memory State ---
ID      Start   Size    Status
------------------------------------
101     0       200     ALLOCATED (P)
102     200     150     ALLOCATED (P)
-1      350     650     FREE
------------------------------------

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 103 300
Process P103 allocated 300 units using First Fit.

--- Current Memory State ---
ID      Start   Size    Status
------------------------------------
101     0       200     ALLOCATED (P)
102     200     150     ALLOCATED (P)
103     350     300     ALLOCATED (P)
-1      650     350     FREE
------------------------------------

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 2
Enter Process ID to deallocate: 102
Process P102 deallocated.

--- Current Memory State ---
ID      Start   Size    Status
------------------------------------
101     0       200     ALLOCATED (P)
-1      200     150     FREE
103     350     300     ALLOCATED (P)
-1      650     350     FREE
------------------------------------

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 1
Enter Process ID and size to allocate: 104 100
Process P104 allocated 100 units using First Fit.

--- Current Memory State ---
ID      Start   Size    Status
------------------------------------
101     0       200     ALLOCATED (P)
104     200     100     ALLOCATED (P)
-1      300     50      FREE
103     350     300     ALLOCATED (P)
```

```
-1      650     350     FREE
------------------------------------

1. Allocate (First Fit)
2. Deallocate
3. Exit
Enter your choice: 3
Exiting.
```

Lab 14: Simulation of FIFO page replacement algorithm Paging

Title

Simulation of FIFO (First-In, First-Out) Page Replacement Algorithm with Paging Context

Aim

This lab is a duplicate of Lab 12. It aims to simulate the First-In, First-Out (FIFO) page replacement algorithm and calculate the number of page faults for a given page reference string and a fixed number of frames, reinforcing the concepts from Lab 12 within the broader context of paging.

Procedure

(Refer to Lab 12 Procedure)

Source Code

(Refer to Lab 12 Source Code)

Input

(Refer to Lab 12 Input)

Expected Output

(Refer to Lab 12 Expected Output)

Lab 15: Simulation of optimal page replacement algorithm

Title

Simulation of Optimal Page Replacement Algorithm

Aim

To simulate the Optimal (OPT) page replacement algorithm and calculate the number of page faults for a given page reference string and a fixed number of frames.

Procedure

1. **Optimal Concept:** Understand Optimal: when a page fault occurs and the memory is full, the page that will not be used for the longest period of time in the future is replaced. This is a theoretical algorithm as it requires future knowledge.
2. **Data Structures:**
   - An array or list to represent the main memory frames.
3. **Input:**
   - Number of available frames in physical memory.
   - A page reference string (sequence of page numbers).
4. **Simulation Logic:**

    - Iterate through the page reference string.
    - For each page:
        - Check if the page is already in a frame (page hit). If yes, continue.
        - If not (page fault):
            - Increment page fault count.
            - If frames are available, add the page to an empty frame.
            - If frames are full, determine which page in the current frames will not be used for the longest time in the *future* of the reference string. Replace that page with the new page.
5. **Output:** Display the state of frames at each step, indicating page hits/faults, and the total number of page faults.

Source Code

```
// Example: C program for Optimal Page Replacement Algorithm Simulation

#include <stdio.h>
#include <stdbool.h>
#include <limits.h> // For INT_MAX

#define MAX_PAGES 20
#define MAX_FRAMES 10

// Function to find the optimal page to replace
int find_optimal_page(int page_reference_string[], int num_pages, int
frames[], int num_frames, int current_index) {
    int farthest_index = current_index;
    int page_to_replace = -1;

    for (int i = 0; i < num_frames; i++) {
        int j;
        int found_future_use = -1; // Index of next use in future

        // Find when frames[i] will be used next
        for (j = current_index + 1; j < num_pages; j++) {
            if (frames[i] == page_reference_string[j]) {
                found_future_use = j;
                break;
            }
        }

        if (found_future_use == -1) {
            // This page will not be used again, so it's the best to replace
            return i;
        } else {
            // This page will be used, compare its next use with others
            if (found_future_use > farthest_index) {
                farthest_index = found_future_use;
                page_to_replace = i;
            }
        }
    }
    return page_to_replace;
}

int main() {
    int page_reference_string[MAX_PAGES];
    int num_pages;
    int num_frames;
    int frames[MAX_FRAMES]; // Represents physical memory frames
    int page_faults = 0;
    int current_frame_count = 0; // How many frames are currently occupied

    printf("Enter the number of frames: ");
```

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

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &num_pages);

    printf("Enter the page reference string (space separated): ");
    for (int i = 0; i < num_pages; i++) {
        scanf("%d", &page_reference_string[i]);
    }

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

    printf("\nPage Reference String\tFrames\t\t\tHit/Fault\n");
    printf("------------------------------------------------------------
\n");

    for (int i = 0; i < num_pages; i++) {
        int current_page = page_reference_string[i];
        bool is_hit = false;

        // Check if page is already in frames (page hit)
        for (int j = 0; j < num_frames; j++) {
            if (frames[j] == current_page) {
                is_hit = true;
                break;
            }
        }

        printf("%d\t\t\t", current_page);

        if (is_hit) {
            printf(" (Hit)\n");
        } else { // Page Fault
            page_faults++;
            printf(" (Fault)\n");

            if (current_frame_count < num_frames) {
                // Frames are not full, add to an empty frame
                frames[current_frame_count] = current_page;
                current_frame_count++;
            } else {
                // Frames are full, replace the optimal page
                int replace_index = find_optimal_page(page_reference_string,
num_pages, frames, num_frames, i);
                frames[replace_index] = current_page;
            }
        }

        // Print current state of frames
        printf("\t\t\t");
        for (int j = 0; j < num_frames; j++) {
            if (frames[j] == -1) {
                printf(" - ");
            } else {
                printf(" %d ", frames[j]);
            }
        }
        printf("\n");
    }

    printf("\nTotal Page Faults: %d\n", page_faults);

    return 0;
```

```
}
```

Input

```
Enter the number of frames: 3
Enter the number of pages in the reference string: 12
Enter the page reference string (space separated): 0 1 2 3 0 1 4 0 1 2 3 4
```

Expected Output

```
Page Reference String     Frames                    Hit/Fault
----------------------------------------------------------------
0                        (Fault)
                          0   -   -
1                        (Fault)
                          0   1   -
2                        (Fault)
                          0   1   2
3                        (Fault)
                          0   1   3
0                        (Hit)
                          0   1   3
1                        (Hit)
                          0   1   3
4                        (Fault)
                          0   4   3
0                        (Hit)
                          0   4   3
1                        (Fault)
                          0   4   1
2                        (Fault)
                          2   4   1
3                        (Fault)
                          2   3   1
4                        (Hit)
                          2   3   4

Total Page Faults: 8
```