

SAURAV

**2022UCS1713** **OS PRACTICAL FILE**

# EXPERIMENT:1

**AIM:** Linux shell commands.

**COMMANDS:**

Here are some commonly used Linux shell commands along with their descriptions:

1. ls: List files and directories in the current directory.
   * Example: `ls -l` (list with long format including permissions, owner, size, etc.)
2. cd: Change directory.

- Example: `cd /path/to/directory` (change to a specific directory)

1. pwd: Print the current working directory.

- Example: `pwd` (prints the full path of the current directory)

1. mkdir: Create a new directory.

- Example: `mkdir my\_directory` (creates a directory named "my\_directory")

1. touch: Create an empty file or update the timestamp of an existing

file.

- Example: `touch new\_file.txt` (creates an empty file named "new\_file.txt")

1. cp: Copy files or directories.

- Example: `cp file1.txt file2.txt` (copies "file1.txt" to "file2.txt")

1. mv: Move or rename files or directories.
   * Example: `mv old\_file.txt new\_file.txt` (renames "old\_file.txt" to "new\_file.txt")
2. rm: Remove files or directories.

- Example: `rm file1.txt` (removes "file1.txt")

1. rmdir: Remove empty directories.

- Example: `rmdir my\_directory` (removes the directory "my\_directory")

1. cat: Display the contents of a file.

- Example: `cat file.txt` (displays the contents of "file.txt")

1. head: Display the beginning of a file (by default, the first 10 lines).
   * Example: `head file.txt` (displays the first 10 lines of "file.txt")
2. tail: Display the end of a file (by default, the last 10 lines).

- Example: `tail file.txt` (displays the last 10 lines of "file.txt")

1. grep: Search for a pattern in files.

- Example: `grep "pattern" file.txt` (searches for "pattern" in "file.txt")

1. ps: Display information about active processes.

- Example: `ps aux` (displays detailed information about all processes)

1. kill: Terminate processes by ID or name.

- Example: `kill PID` (terminates the process with the specified PID)

# EXPERIMENT-2

**AIM:**Write C programs using fork(), getpid(), getppid() and exec() system calls.

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

The programs use the following system calls and concepts:

* fork(): Used to create a new process (child process) from an existing process (parent process).
* getpid(): Retrieves the process ID (PID) of the current process.
* getppid(): Retrieves the parent process ID (PPID) of the current process.
* exec(): Executes a new program in the context of the current

process.

## Code:

**Program 1: Using `fork()` and `getpid()`**

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

int main() { pid\_t pid;

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

if (pid < 0) {

fprintf(stderr, "Fork failed\n");

return 1;

} else if (pid == 0) { printf("Child process:\n");

printf("PID: %d\n", getpid()); // Get child process ID printf("Parent PID: %d\n", getppid()); // Get parent process ID

} else {

printf("Parent process:\n");

printf("PID: %d\n", getpid()); // Get parent process ID printf("Child PID: %d\n", pid); // Get child process ID

}

return 0;

}

**Program 2: Using `fork()` and `exec()`**

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

int main() { pid\_t pid;

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

if (pid < 0) {

fprintf(stderr, "Fork failed\n"); return 1;

} else if (pid == 0) { printf("Child process:\n");

printf("PID: %d\n", getpid()); // Get child process ID printf("Parent PID: %d\n", getppid()); // Get parent process ID

// Execute a command using exec()

char \*args[] = {"ls", "-l", NULL}; // Command and arguments execvp(args[0], args); // Execute ls -l command

printf("This will not be printed if execvp() succeeds.\n"); // This line is not executed if execvp() succeeds

} else {

printf("Parent process:\n");

printf("PID: %d\n", getpid()); // Get parent process ID printf("Child PID: %d\n", pid); // Get child process ID

wait(NULL); // Wait for the child process to finish printf("Child process finished.\n");

}

return 0;

}

## Output:

***Program 1 Output:***

Parent process:

PID: <parent\_pid> Child PID: <child\_pid> Child process:

PID: <child\_pid>

Parent PID: <parent\_pid>

```

***Program 2 Output:***

Parent process:

PID: <parent\_pid>

Child PID: <child\_pid> Child process:

PID: <child\_pid>

Parent PID: <parent\_pid>

<output\_of\_ls\_-l\_command> Child process finished.

```

## Result:

* Program 1 demonstrates the creation of a child process using `fork()` and displays the PID and PPID of both parent and child processes.
* Program 2 demonstrates the creation of a child process using `fork()` and executing the `ls -l` command in the child process using

`exec()`. It also waits for the child process to finish using `wait()` in the parent process.

**EXPERIMENT-3**

**AIM**:Write a C program to represent a family of processes as a tree.

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

In this program, we will use the `fork()` system call to create child processes. Each process will print its process ID (PID) and its parent's process ID (PPID), thereby representing a tree structure where each process has one parent and possibly multiple children.

Code:

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

void createFamily(int depth, int maxDepth) { if (depth >= maxDepth) {

return;

}

pid\_t pid = fork(); if (pid < 0) {

fprintf(stderr, "Fork failed\n"); return;

} else if (pid == 0) {

// Child process

printf("Child: PID = %d, PPID = %d\n", getpid(), getppid());

createFamily(depth + 1, maxDepth); // Recursive call to create child processes

return;

} else {

// Parent process

printf("Parent: PID = %d, Child PID = %d\n", getpid(), pid);

// No need to wait for child processes here return;

}

}

int main() {

int maxDepth = 3; // Maximum depth of the process tree createFamily(0, maxDepth);

return 0;

}

## Output:

The output will show the process IDs (PID) and parent process IDs (PPID) of each process, forming a tree-like structure.

**Parent: PID = 30555, Child PID = 30556 Child: PID = 30556, PPID = 30555 Parent: PID = 30556, Child PID = 30557 Child: PID = 30557, PPID = 30556**

## Result:

* The program demonstrates the creation of a family of processes as a tree, where each process has one parent and multiple children, up to a specified depth.
* Running the program will display the process IDs and parent process

IDs, showing the tree structure of the processes.

# EXPERIMENT-4

**AIM:**Write a program to simulate FCFS CPU scheduling algorithm. Here's a C program to simulate the First-Come, First-Served (FCFS) CPU scheduling algorithm:

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

FCFS (First-Come, First-Served) is a non-preemptive CPU scheduling algorithm where processes are executed in the order they arrive. The process that arrives first gets executed first, followed by the next process in the queue.

## Code:

#include <stdio.h> struct Process {

int processId; int arrivalTime; int burstTime;

};

void fcfsScheduling(struct Process processes[], int n) { int currentTime = 0;

float totalWaitingTime = 0;

printf("Process\tArrival Time\tBurst Time\tWaiting Time\n");

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

printf("%d\t%d\t\t%d\t\t", processes[i].processId, processes[i].arrivalTime, processes[i].burstTime);

if (currentTime < processes[i].arrivalTime) { currentTime = processes[i].arrivalTime;

}

int waitingTime = currentTime - processes[i].arrivalTime; totalWaitingTime += waitingTime;

printf("%d\n", waitingTime);

currentTime += processes[i].burstTime;

}

printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

struct Process processes[] = {{1, 0, 5}, {2, 1, 3}, {3, 2, 8}, {4, 3, 6}}; int n = sizeof(processes) / sizeof(processes[0]);

fcfsScheduling(processes, n);

return 0;

}

## Output:

The output of the program will display the process details (Process ID, Arrival Time, Burst Time) and the waiting time for each process, followed by the average waiting time.

**Process Arrival Time Burst Time Waiting Time**

|  |  |  |  |
| --- | --- | --- | --- |
| **1** | **0** | **5** | **0** |
| **2** | **1** | **3** | **4** |
| **3** | **2** | **8** | **6** |
| **4** | **3** | **6** | **13** |

**Average Waiting Time: 5.75**

## Result:

* The program simulates the FCFS CPU scheduling algorithm by calculating the waiting time for each process based on its arrival time and burst time.
* It then calculates and displays the average waiting time for all processes in the queue.

# EXPERIMENT-5

**AIM**:Write a program to simulate SJF scheduling algorithm. Here's a C program to simulate the Shortest Job First (SJF) CPU scheduling algorithm:

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

SJF (Shortest Job First) is a CPU scheduling algorithm where the process with the shortest burst time is selected for execution next. This algorithm can be either preemptive or non-preemptive.

## Code:

#include <stdio.h> struct Process {

int processId; int arrivalTime; int burstTime;

};

void sjfScheduling(struct Process processes[], int n) { int currentTime = 0;

float totalWaitingTime = 0; int remainingTime[n];

int completed = 0;

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

remainingTime[i] = processes[i].burstTime;

}

while (completed != n) { int shortestJob = -1;

int shortestTime = 9999;

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

if (processes[i].arrivalTime <= currentTime && remainingTime[i] < shortestTime && remainingTime[i] > 0) {

shortestJob = i;

shortestTime = remainingTime[i];

}

}

if (shortestJob == -1) { currentTime++;

} else {

printf("Process %d started at time %d\n", processes[shortestJob].processId, currentTime);

currentTime += remainingTime[shortestJob]; remainingTime[shortestJob] = 0;

int waitingTime = currentTime - processes[shortestJob].arrivalTime - processes[shortestJob].burstTime;

totalWaitingTime += waitingTime; printf("Process %d completed at time %d\n",

processes[shortestJob].processId, currentTime); completed++;

}

}

printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

struct Process processes[] = {{1, 0, 6}, {2, 1, 8}, {3, 2, 7}, {4, 3, 3}}; int n = sizeof(processes) / sizeof(processes[0]);

sjfScheduling(processes, n);

return 0;

}

## Output:

The output of the program will display the processes' start and completion times, as well as the average waiting time.

**Process 1 started at time 0**

**Process 1 completed at time 6**

**Process 4 started at time 6**

**Process 4 completed at time 9**

**Process 3 started at time 9**

**Process 3 completed at time 16**

**Process 2 started at time 16**

**Process 2 completed at time 24 Average Waiting Time: 6.25**

## Result:

* The program simulates the SJF CPU scheduling algorithm by selecting the process with the shortest remaining burst time for execution.
* It calculates and displays the average waiting time for all processes

in the queue.

# EXPERIMENT-6

**AIM**:Write a program to simulate pre-emptive Priority scheduling algorithm.

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

Preemptive Priority scheduling is a CPU scheduling algorithm where each process is assigned a priority, and the process with the highest priority is selected for execution. In case of a tie, the process with the earliest arrival time is chosen. Preemption can occur if a new process with a higher priority arrives or if a higher priority process becomes available due to I/O completion.

## Code:

#include <stdio.h>

struct Process { int processId; int arrivalTime; int burstTime; int priority;

};

void preemptivePriorityScheduling(struct Process processes[], int n) { int currentTime = 0;

int remainingTime[n]; int completed = 0;

int processIndex = -1;

int currentPriority = 9999;

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

remainingTime[i] = processes[i].burstTime;

}

while (completed != n) { processIndex = -1;

currentPriority = 9999;

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

if (processes[i].arrivalTime <= currentTime && remainingTime[i] > 0 && processes[i].priority < currentPriority) {

processIndex = i;

currentPriority = processes[i].priority;

}

}

if (processIndex == -1) { currentTime++;

} else {

printf("Process %d started at time %d\n", processes[processIndex].processId, currentTime);

remainingTime[processIndex]--; currentTime++;

if (remainingTime[processIndex] == 0) { printf("Process %d completed at time %d\n",

processes[processIndex].processId, currentTime); completed++;

}

}

}

}

int main() {

struct Process processes[] = {{1, 0, 6, 2}, {2, 1, 8, 1}, {3, 2, 7, 3}, {4, 3, 3,

4}};

int n = sizeof(processes) / sizeof(processes[0]); preemptivePriorityScheduling(processes, n);

return 0;

}

## Output:

The output of the program will display the start and completion times of processes according to the preemptive priority scheduling algorithm.

**Process 1 started at time 0**

**Process 2 started at time 1**

**Process 2 started at time 2**

**Process 2 started at time 3**

**Process 2 started at time 4**

**Process 2 started at time 5**

**Process 2 started at time 6**

**Process 2 started at time 7**

**Process 2 started at time 8**

**Process 2 completed at time 9**

**Process 1 started at time 9**

**Process 1 started at time 10**

**Process 1 started at time 11**

**Process 1 started at time 12**

**Process 1 started at time 13**

**Process 1 completed at time 14**

**Process 3 started at time 14**

**Process 3 started at time 15**

**Process 3 started at time 16**

**Process 3 started at time 17**

**Process 3 started at time 18**

**Process 3 started at time 19**

**Process 3 started at time 20**

**Process 3 completed at time 21**

**Process 4 started at time 21**

**Process 4 started at time 22**

**Process 4 started at time 23**

**Process 4 completed at time 24**

## Result:

* The program simulates the Preemptive Priority CPU scheduling algorithm by selecting the process with the highest priority and preempting lower priority processes if necessary.
* It displays the execution sequence and completion times of processes.

# EXPERIMENT-7

**AIM**:Write a program to simulate Round Robin scheduling algorithm.

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

Round Robin is a preemptive CPU scheduling algorithm where each process is assigned a fixed time unit (time quantum) for execution. Once a process consumes its time quantum, it is preempted and moved to the back of the ready queue. The next process in the queue is then executed, and this cycle continues until all processes are completed.

## Code:

#include <stdio.h> struct Process {

int processId; int arrivalTime; int burstTime;

int remainingTime;

};

void roundRobinScheduling(struct Process processes[], int n, int timeQuantum) {

int currentTime = 0; int completed = 0;

while (completed != n) { for (int i = 0; i < n; i++) {

if (processes[i].remainingTime > 0) {

if (processes[i].remainingTime <= timeQuantum) { currentTime += processes[i].remainingTime; processes[i].remainingTime = 0;

completed++;

printf("Process %d completed at time %d\n", processes[i].processId, currentTime);

} else {

currentTime += timeQuantum; processes[i].remainingTime -= timeQuantum; printf("Process %d executed for time quantum at time

%d\n", processes[i].processId, currentTime);

}

}

}

}

}

int main() {

struct Process processes[] = {{1, 0, 6, 6}, {2, 1, 8, 8}, {3, 2, 7, 7}, {4, 3,

3, 3}};

int n = sizeof(processes) / sizeof(processes[0]);

int timeQuantum = 2; // Time quantum for Round Robin algorithm roundRobinScheduling(processes, n, timeQuantum);

return 0;

}

## Output:

The output of the program will display the execution sequence and completion times of processes according to the Round Robin scheduling algorithm.

**Process 1 executed for time quantum at time 2 Process 2 executed for time quantum at time 4 Process 3 executed for time quantum at time 6 Process 4 executed for time quantum at time 8 Process 1 executed for time quantum at time 10 Process 2 executed for time quantum at time 12 Process 3 executed for time quantum at time 14 Process 4 completed at time 15**

**Process 1 completed at time 17**

**Process 2 executed for time quantum at time 19 Process 3 executed for time quantum at time 21 Process 2 completed at time 23**

**Process 3 completed at time 24**

## Result:

* The program simulates the Round Robin CPU scheduling algorithm by assigning a fixed time quantum to each process.
* It displays the execution sequence and completion times of processes.

# EXPERIMENT-8

**AIM**:Write a program to simulate Multilevel Feedback Queue scheduling algorithm.

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

Multilevel Feedback Queue is a CPU scheduling algorithm that uses multiple queues with different priority levels. Processes are initially placed in the highest priority queue. If a process uses up its time quantum without completing, it is moved to a lower priority queue. This process continues until the process completes or reaches the lowest priority queue.

## Code:

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

struct Process { int processId; int arrivalTime; int burstTime;

int remainingTime; int priority;

};

void mlfqScheduling(struct Process processes[], int n, int quantum[], int numberOfQueues) {

int currentTime = 0; bool completed[n];

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

processes[i].remainingTime = processes[i].burstTime; completed[i] = false;

}

while (1) {

bool allCompleted = true;

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

if (processes[i].remainingTime > 0) { allCompleted = false;

if (processes[i].remainingTime > quantum[processes[i].priority]) {

currentTime += quantum[processes[i].priority]; processes[i].remainingTime -=

quantum[processes[i].priority];

printf("Process %d executed for time quantum at time

%d\n", processes[i].processId, currentTime);

} else {

currentTime += processes[i].remainingTime; processes[i].remainingTime = 0; completed[i] = true;

printf("Process %d completed at time %d\n", processes[i].processId, currentTime);

}

if (currentTime >= processes[i].arrivalTime + quantum[processes[i].priority]) {

processes[i].priority = (processes[i].priority == numberOfQueues - 1) ? processes[i].priority : processes[i].priority + 1;

}

}

}

if (allCompleted) { break;

}

}

}

int main() {

struct Process processes[] = {{1, 0, 6, 6, 0}, {2, 1, 8, 8, 0}, {3, 2, 7, 7, 0},

{4, 3, 3, 3, 0}};

int n = sizeof(processes) / sizeof(processes[0]);

int quantum[] = {2, 4, 6}; // Quantum for each queue in MLFQ int numberOfQueues = sizeof(quantum) / sizeof(quantum[0]);

mlfqScheduling(processes, n, quantum, numberOfQueues);

return 0;

}

## Output:

The output of the program will display the execution sequence and completion times of processes according to the Multilevel Feedback Queue scheduling algorithm.

**Process 1 executed for time quantum at time 2 Process 2 executed for time quantum at time 4 Process 3 executed for time quantum at time 6 Process 4 executed for time quantum at time 8 Process 1 completed at time 12**

**Process 2 executed for time quantum at time 16 Process 3 executed for time quantum at time 20 Process 4 completed at time 21**

**Process 2 completed at time 23**

**Process 3 completed at time 24**

## Result:

* The program simulates the Multilevel Feedback Queue CPU scheduling algorithm by using multiple queues with different priorities.
* It displays the execution sequence and completion times of processes.

# EXPERIMENT-9

**AIM:**Write a program to simulate deadlock avoidance.

Deadlock avoidance is a technique used to prevent deadlock in a system by ensuring that the system never enters an unsafe state. One of the popular algorithms for deadlock avoidance is the Banker's algorithm. Below is a C program to simulate deadlock avoidance using the Banker's algorithm:

## Software Required:

* Linux operating system
* C compiler (e.g., GCC)

## Theory:

The Banker's algorithm is a deadlock avoidance algorithm used in resource allocation systems. It works by simulating the allocation of resources to processes and checking if the system enters a safe state. If a request from a process can be granted without leading to an unsafe state (i.e., avoiding deadlock), the request is granted; otherwise, the process is made to wait.

## Code:

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

int main() {

int processes, resources;

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

printf("Enter the number of resources: ");

scanf("%d", &resources);

int allocation[processes][resources]; int max[processes][resources];

int available[resources]; bool finish[processes]; int work[resources];

int safeSequence[processes]; int count = 0;

printf("Enter the allocation matrix:\n"); for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) { scanf("%d", &allocation[i][j]);

}

finish[i] = false;

}

printf("Enter the maximum matrix:\n"); for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) { scanf("%d", &max[i][j]);

}

}

printf("Enter the available resources:\n"); for (int i = 0; i < resources; i++) {

scanf("%d", &available[i]);

}

for (int i = 0; i < resources; i++) { work[i] = available[i];

}

while (count < processes) { bool found = false;

for (int i = 0; i < processes; i++) { if (!finish[i]) {

bool canExecute = true;

for (int j = 0; j < resources; j++) {

if (max[i][j] - allocation[i][j] > work[j]) { canExecute = false;

break;

}

}

if (canExecute) {

for (int j = 0; j < resources; j++) { work[j] += allocation[i][j];

}

finish[i] = true; safeSequence[count++] = i; found = true;

}

}

}

if (!found) {

printf("System is in unsafe state, deadlock avoided!\n"); return 0;

}

}

printf("System is in safe state, safe sequence: "); for (int i = 0; i < processes; i++) {

printf("%d ", safeSequence[i]);

}

printf("\n"); return 0;

}

## INPUT:

**Enter the number of processes: 5 Enter the number of resources: 3**

**Enter the allocation matrix: 1 1 2**

**1 3 3**

**1 2 1**

**1 4 2**

**2 3 2**

**Enter the maximum matrix: 3 3 3**

**1 6 6**

**2 4 2**

**3 7 4**

**4 3 3**

**Enter the available resources: 1 2 2**

## Output:

* If the system is in a safe state, the program will display the safe sequence.
* If the system is in an unsafe state, the program will display a message indicating that deadlock was avoided.

**System is in safe state, safe sequence: 2 3 4 0 1**

## Result:

* The program simulates deadlock avoidance using the Banker's algorithm by checking if the system can enter a safe state before granting resource requests.
* It demonstrates how deadlock can be avoided by ensuring that resource allocations do not lead to an unsafe state.

# EXPERIMENT-10

**AIM**:Write a program to simulate deadlock detection.

## SOFTWARE REQUIRED-

Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

Deadlock detection is a technique used to identify whether a system is in a deadlock state, where processes are unable to proceed due to circular waiting for resources. The resource allocation graph method is one approach to detect deadlock. In this method, the system's resource allocation state is represented as a directed graph, where processes are nodes, and edges represent resource allocation and resource requests.

## CODE:

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

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int processes, resources;

int allocation[MAX\_PROCESSES][MAX\_RESOURCES]; int request[MAX\_PROCESSES][MAX\_RESOURCES];

int available[MAX\_RESOURCES]; bool finish[MAX\_PROCESSES]; bool marked[MAX\_PROCESSES];

bool isSafeState(int work[], bool finish[]) { int tempWork[resources];

for (int i = 0; i < resources; i++) { tempWork[i] = work[i];

}

for (int i = 0; i < processes; i++) { if (!finish[i]) {

bool canExecute = true;

for (int j = 0; j < resources; j++) {

if (request[i][j] > tempWork[j]) { canExecute = false;

break;

}

}

if (canExecute) {

finish[i] = true;

for (int j = 0; j < resources; j++) { tempWork[j] += allocation[i][j];

}

i = -1; // Start from the beginning to check again

}

}

}

for (int i = 0; i < processes; i++) { if (!finish[i]) {

return false; // Unsafe state

}

}

return true; // Safe state

}

bool checkDeadlock() {

int work[resources];

for (int i = 0; i < resources; i++) { work[i] = available[i];

}

for (int i = 0; i < processes; i++) { marked[i] = false;

}

for (int i = 0; i < processes; i++) { if (!marked[i]) {

marked[i] = true;

for (int j = 0; j < resources; j++) { work[j] -= request[i][j];

}

if (!isSafeState(work, finish)) { return true; // Deadlock detected

}

marked[i] = false;

for (int j = 0; j < resources; j++) { work[j] += request[i][j];

}

}

}

return false; // No deadlock

}

int main() {

// Input allocation matrix, request matrix, and available resources here

// Check for deadlock if (checkDeadlock()) {

printf("Deadlock detected!\n");

} else {

printf("No deadlock detected.\n");

}

return 0;

}

```

## OUTPUT:

* If a deadlock is detected, the program will output "Deadlock detected!".
* If no deadlock is detected, the program will output "No deadlock detected.”

**No deadlock detected.**

## RESULT:

* Running the program with appropriate input data will determine whether the system is in a deadlock state or not.

# EXPERIMENT-11

**AIM-**Write a program to simulate best-fit contiguous memory allocation.

## SOFTWARE REQUIRED-

Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

Best-fit contiguous memory allocation is a memory management technique where the operating system allocates the smallest available block of memory that is sufficient to satisfy a process's memory request. The goal is to minimize memory wastage and fragmentation.

## CODE:

#include <iostream> #include <vector>

using namespace std;

struct MemoryBlock { int id;

int size;

bool allocated;

};

void bestFit(vector<MemoryBlock>& memory, int processId, int processSize) {

int bestFitIndex = -1;

int minFragmentation = INT\_MAX;

for (int i = 0; i < memory.size(); i++) {

if (!memory[i].allocated && memory[i].size >= processSize) { int fragmentation = memory[i].size - processSize;

if (fragmentation < minFragmentation) { minFragmentation = fragmentation; bestFitIndex = i;

}

}

}

if (bestFitIndex != -1) { memory[bestFitIndex].allocated = true; memory[bestFitIndex].id = processId;

cout << "Process " << processId << " allocated at memory block " << bestFitIndex << endl;

} else {

cout << "Process " << processId << " cannot be allocated due to insufficient memory" << endl;

}

}

int main() {

vector<MemoryBlock> memory = {{1, 100, false}, {2, 50, false}, {3,

200, false}, {4, 70, false}};

int processId, processSize;

cout << "Enter process ID and size: "; cin >> processId >> processSize;

bestFit(memory, processId, processSize);

return 0;

}

## OUTPUT:

The program will output whether the process can be allocated and if so, at which memory block.

**Enter process ID and size: 1 2**

**Process 1 allocated at memory block 1**

## RESULT:

Execute the program with different process sizes to observe how best- fit contiguous memory allocation minimizes fragmentation and optimally utilizes available memory blocks.

**EXPERIMENT-12**

**AIM-**Write a program to simulate FIFO page replacement algorithm.

**SOFTWARE REQUIRED:**Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

FIFO page replacement is a simple page replacement algorithm where the oldest page in memory (i.e., the page that was brought into memory first) is replaced when a page fault occurs.

This algorithm works on the principle of First-In-First-Out, similar to a queue data structure.

## CODE:

#include <iostream> #include <queue> #include <unordered\_set>

using namespace std;

int fifoPageReplacement(int pages[], int n, int capacity) { queue<int> fifoQueue;

unordered\_set<int> pageSet; int pageFaults = 0;

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

if (pageSet.find(pages[i]) == pageSet.end()) { pageFaults++;

if (fifoQueue.size() == capacity) { int frontPage = fifoQueue.front(); fifoQueue.pop(); pageSet.erase(frontPage);

}

fifoQueue.push(pages[i]); pageSet.insert(pages[i]);

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages) / sizeof(pages[0]); int capacity = 3;

int pageFaults = fifoPageReplacement(pages, n, capacity); cout << "Page Faults using FIFO: " << pageFaults << endl;

return 0;

}

## OUTPUT:

The program will output the total number of page faults encountered during the simulation using the FIFO page replacement algorithm.

**Page Faults using FIFO: 10**

## RESULT:

Execute the program with different page sequences and capacities to observe how the FIFO page replacement algorithm manages page faults by replacing the oldest page in memory.

# EXPERIMENT-13

**AIM:**Write a program to simulate LRU page replacement algorithm.

**SOFTWARE REQUIRED**:Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

LRU page replacement is a memory management algorithm that replaces the least recently used page in memory when a new page needs to be loaded and there is no free space.

LRU works on the principle of locality, assuming that pages that have been accessed recently are more likely to be accessed again in the near future.

## CODE:

#include <iostream> #include <unordered\_map> #include <list>

using namespace std;

int lruPageReplacement(int pages[], int n, int capacity) { unordered\_map<int, list<int>::iterator> pageMap; list<int> lruList;

int pageFaults = 0;

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

if (pageMap.find(pages[i]) == pageMap.end()) { pageFaults++;

if (lruList.size() == capacity) {

int lruPage = lruList.back(); lruList.pop\_back(); pageMap.erase(lruPage);

}

lruList.push\_front(pages[i]); pageMap[pages[i]] = lruList.begin();

} else {

lruList.erase(pageMap[pages[i]]); lruList.push\_front(pages[i]); pageMap[pages[i]] = lruList.begin();

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages) / sizeof(pages[0]); int capacity = 3;

int pageFaults = lruPageReplacement(pages, n, capacity); cout << "Page Faults using LRU: " << pageFaults << endl;

return 0;

}

## OUTPUT:

The program will output the total number of page faults encountered during the simulation using the LRU page replacement algorithm.

**Page Faults using LRU: 9**

## RESULT:

Execute the program with different page sequences and capacities to observe how the LRU page replacement algorithm manages page faults by replacing the least recently used pages in memory.

# EXPERIMENT-14

**AIM**:Write a program to simulate Second Chance page replacement algorithm.

## SOFTWARE REQUIRED:

Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

The Second Chance page replacement algorithm is an enhancement of the FIFO algorithm. It works by keeping track of the "second chance" given to pages that are candidates for replacement.

Pages are maintained in a circular queue and are given a chance to stay in memory (second chance) before being considered for replacement again.

## CODE:

#include <iostream> #include <unordered\_map> #include <list>

using namespace std;

int secondChancePageReplacement(int pages[], int n, int capacity) { unordered\_map<int, bool> secondChanceMap;

list<int> secondChanceList; int pageFaults = 0;

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

if (secondChanceMap.find(pages[i]) == secondChanceMap.end()) { pageFaults++;

if (secondChanceList.size() == capacity) { while (true) {

int page = secondChanceList.front(); secondChanceList.pop\_front();

if (secondChanceMap[page]) { secondChanceMap[page] = false; secondChanceList.push\_back(page);

} else {

secondChanceMap.erase(page); break;

}

}

}

secondChanceList.push\_back(pages[i]); secondChanceMap[pages[i]] = true;

} else {

secondChanceMap[pages[i]] = true;

}

}

return pageFaults;

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages) / sizeof(pages[0]); int capacity = 3;

int pageFaults = secondChancePageReplacement(pages, n, capacity);

cout << "Page Faults using Second Chance: " << pageFaults << endl; return 0;

}

## OUTPUT:

The program will output the total number of page faults encountered during the simulation using the Second Chance page replacement algorithm.

**Page Faults using Second Chance: 9**

## RESULT:

Execute the program with different page sequences and capacities to observe how the Second Chance page replacement algorithm manages page faults by giving a "second chance" to pages before replacement.

# EXPERIMENT-15

**AIM**:Write a program to simulate Enhanced Second Chance page replacement algorithm.

## SOFTWARE REQUIRED:

Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

The Enhanced Second Chance page replacement algorithm is an enhancement of the Second Chance algorithm. It adds the concept of page modification (dirty bit) and aging to approximate the Least Recently Used (LRU) algorithm better. Pages are given a "second chance" to stay in memory, taking into account their modification status and age.

## CODE:

#include <iostream> #include <unordered\_map> #include <list>

using namespace std;

class EnhancedSecondChance { private:

struct Page {

bool referenced; bool modified; int age;

int frameNumber;

};

int capacity; list<Page> pageList;

unordered\_map<int, list<Page>::iterator> pageTable; int ageCounter;

int pageFaults;

public:

EnhancedSecondChance(int cap) : capacity(cap), ageCounter(0), pageFaults(0) {}

void referencePage(int pageNum, bool isModified) { if (pageTable.find(pageNum) == pageTable.end()) {

if (pageList.size() == capacity) { while (true) {

Page frontPage = pageList.front(); pageList.pop\_front();

if (frontPage.referenced) { frontPage.referenced = false; pageList.push\_back(frontPage);

} else {

pageTable.erase(frontPage.frameNumber); break;

}

}

}

Page newPage = {true, isModified, ageCounter++, pageNum}; pageList.push\_back(newPage);

pageTable[pageNum] = prev(pageList.end()); pageFaults++;

} else {

auto it = pageTable[pageNum]; it->referenced = true;

it->modified |= isModified;

}

}

int getPageFaults() const { return pageFaults;

}

};

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages) / sizeof(pages[0]); int capacity = 3;

EnhancedSecondChance esc(capacity);

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

esc.referencePage(pages[i], false); // Assume pages are not modified for simplicity

}

int pageFaults = esc.getPageFaults();

cout << "Page Faults using Enhanced Second Chance: " << pageFaults << endl;

return 0;

}

## OUTPUT:

The program will output the total number of page faults encountered during the simulation using the Enhanced Second Chance page replacement algorithm.

**Page Faults using Enhanced Second Chance: 9**

## RESULT:

Execute the program with different page sequences and memory capacities to observe how the Enhanced Second Chance algorithm manages page faults based on page reference, modification status, and age. Analyze the output to understand the effectiveness of this algorithm compared to other page replacement algorithms.

# EXPERIMENT:16

**AIM**:Write a program to simulate LFU page replacement algorithm.

**SOFTWARE REQUIRED**:Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

The Least Frequently Used (LFU) page replacement algorithm replaces the page that has been accessed the least frequently. It maintains a count of how often each page is accessed. When a page needs to be replaced, the algorithm selects the page with the lowest access frequency count.

## Code:

#include <iostream> #include <unordered\_map> #include <list>

#include <algorithm> using namespace std;

class LFUPageReplacement { private:

struct Page {

int pageNumber; int frequency; int age;

};

int capacity; list<Page> pageList;

unordered\_map<int, list<Page>::iterator> pageTable;

public:

LFUPageReplacement(int cap) : capacity(cap) {}

void referencePage(int pageNum) {

if (pageTable.find(pageNum) == pageTable.end()) { if (pageList.size() == capacity) {

auto leastFrequentPage = min\_element(pageList.begin(), pageList.end(),

[](const Page& a, const Page& b) { return a.frequency < b.frequency; });

pageTable.erase(leastFrequentPage->pageNumber); pageList.erase(leastFrequentPage);

}

pageList.push\_back({pageNum, 1, 0}); pageTable[pageNum] = prev(pageList.end());

} else {

auto it = pageTable[pageNum]; it->frequency++;

}

}

int getPageFaults() const { return pageTable.size();

}

};

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages) / sizeof(pages[0]); int capacity = 3;

LFUPageReplacement lfu(capacity);

for (int i = 0; i < n; i++) { lfu.referencePage(pages[i]);

}

int pageFaults = lfu.getPageFaults();

cout << "Page Faults using LFU: " << pageFaults << endl;

return 0;

}

## Output:

The program will output the total number of page faults encountered during the simulation using the LFU page replacement algorithm.

**Page Faults using LFU: 3**

## Result:

Execute the program with different page sequences and memory capacities to observe how the LFU algorithm manages page faults based on page access frequency. Analyze the output to understand the effectiveness of this algorithm compared to other page replacement algorithms in minimizing page faults.

# Experiment:17

**AIM:** Write a program to simulate FCFS disk scheduling algorithm.

## SOFTWARE REQUIRED:

Any C or C++ compiler (e.g., GCC for C, g++ for C++)

## THEORY:

FCFS (First-Come-First-Served) is a non-preemptive disk scheduling algorithm where disk I/O requests are serviced in the order they arrive. It serves the requests in the sequence they are received without considering their priority or waiting time. This algorithm is straightforward but can lead to high average seek time and inefficient disk utilization, especially in scenarios with mixed I/O requests.

## Code:

#include <iostream> #include <vector> #include <cmath>

using namespace std;

class FCFS { private:

vector<int> requests; int headPosition;

int totalMovement;

public:

FCFS(const vector<int>& req, int headPos) : requests(req), headPosition(headPos), totalMovement(0) {}

void simulateFCFS() {

int currentTrack = headPosition;

for (int i = 0; i < requests.size(); i++) {

totalMovement += abs(requests[i] - currentTrack); currentTrack = requests[i];

}

}

int getTotalMovement() const { return totalMovement;

}

};

int main() {

vector<int> requests = {98, 183, 37, 122, 14, 124, 65, 67}; int headPosition = 53;

FCFS fcfs(requests, headPosition); fcfs.simulateFCFS();

int totalMovement = fcfs.getTotalMovement();

cout << "Total head movement using FCFS: " << totalMovement << endl;

return 0;

}

## Output:

The program will output the total head movement required to service the disk requests using the FCFS disk scheduling algorithm.

**Total head movement using FCFS: 640**

## Result:

Execute the program with different disk request sequences and initial head positions to observe how the FCFS algorithm handles disk I/O requests and computes the total head movement. Analyze the output to understand the behavior of FCFS in disk scheduling.

# Experiment-18

**Aim:**Write a program to simulate SSTF disk scheduling algorithm.

**Software Required:** C++ compiler (e.g., GCC, g++), Text editor or integrated development environment (IDE)

## Theory:

SSTF (Shortest Seek Time First) is a disk scheduling algorithm that selects the request with the shortest seek time from the current head position to the next requested track. This algorithm aims to minimize the seek time and improve disk I/O performance by prioritizing closer tracks over farther ones. SSTF is a type of non-preemptive algorithm that considers only the current pending requests for scheduling.

## Code:

#include <iostream> #include <vector> #include <cmath> #include <algorithm>

using namespace std; class SSTF {

private:

vector<int> requests; int headPosition;

int totalMovement; public:

SSTF(const vector<int>& req, int headPos) : requests(req), headPosition(headPos), totalMovement(0) {}

void simulateSSTF() {

int currentTrack = headPosition;

while (!requests.empty()) {

auto minSeek = min\_element(requests.begin(), requests.end(), [currentTrack](int a, int b) { return abs(a - currentTrack) <

abs(b - currentTrack); });

totalMovement += abs(\*minSeek - currentTrack); currentTrack = \*minSeek; requests.erase(minSeek);

}

}

int getTotalMovement() const { return totalMovement;

}

};

int main() {

vector<int> requests = {98, 183, 37, 122, 14, 124, 65, 67}; int headPosition = 53;

SSTF sstf(requests, headPosition); sstf.simulateSSTF();

int totalMovement = sstf.getTotalMovement();

cout << "Total head movement using SSTF: " << totalMovement << endl;

return 0;

}

## Output:

The program will output the total head movement required to service the disk requests using the SSTF disk scheduling algorithm.

**Total head movement using SSTF: 236**

## Result:

Execute the program with different disk request sequences and initial head positions to observe how the SSTF algorithm minimizes seek time by selecting the closest track for servicing. Analyze the output to understand the effectiveness of SSTF in optimizing disk access.

# EXPERIMENT-19

**Aim:** Simulate the C-SCAN disk scheduling algorithm.

## Software Required:

C++ compiler (e.g., GCC, g++), Text editor or integrated development environment (IDE)

## Theory:

C-SCAN (Circular SCAN) is a disk scheduling algorithm that services disk I/O requests in a circular manner. The disk arm moves in one direction (e.g., from innermost to outermost tracks), serving all requests in that direction, and then "wraps around" to the beginning of the disk to continue servicing requests in the same direction. This algorithm prevents starvation of requests near the end of the disk by ensuring that the arm returns to the starting point after completing one scan.

## Code:

#include <iostream> #include <vector> #include <algorithm>

using namespace std;

class CSCAN { private:

vector<int> requests; int headPosition;

int totalMovement; public:

CSCAN(const vector<int>& req, int headPos) : requests(req), headPosition(headPos), totalMovement(0) {}

void simulateCSCAN() { sort(requests.begin(), requests.end());

int currentTrack = headPosition;

int diskSize = 200; // Assuming disk size of 200 tracks

// Forward pass

for (int i = 0; i < requests.size(); i++) { if (requests[i] >= currentTrack) {

totalMovement += abs(requests[i] - currentTrack); currentTrack = requests[i];

}

}

// Move to the end of the disk totalMovement += (diskSize - currentTrack);

// Wrap around to the beginning of the disk currentTrack = 0;

totalMovement += currentTrack;

// Backward pass

for (int i = 0; i < requests.size(); i++) { if (requests[i] < currentTrack) {

totalMovement += abs(requests[i] - currentTrack); currentTrack = requests[i];

}

}

}

int getTotalMovement() const

return totalMovement;

}

};

int main() {

vector<int> requests = {98, 183, 37, 122, 14, 124, 65, 67}; int headPosition = 53;

CSCAN cscan(requests, headPosition); cscan.simulateCSCAN();

int totalMovement = cscan.getTotalMovement();

cout << "Total head movement using C-SCAN: " << totalMovement << endl;

return 0;

}

## Output:

The program will output the total head movement required to service the disk requests using the C-SCAN disk scheduling algorithm.

**Total head movement using C-SCAN: 147**

## Result:

Execute the program with different disk request sequences and initial head positions to observe how the C-SCAN algorithm performs circular scanning and minimizes head movements. Analyze the output to understand the effectiveness of C-SCAN in disk scheduling.

# EXPERIMENT-20

**Aim:**Write a program to simulate LOOK disk scheduling algorithm.

## Software Required:

C++ compiler (e.g., GCC, g++), Text editor or integrated development environment (IDE)

## Theory:

LOOK is a disk scheduling algorithm that services disk I/O requests by scanning the disk arm in a specific direction until there are no more pending requests in that direction, and then it reverses direction and scans back. This algorithm prevents the disk arm from unnecessarily traversing empty areas of the disk, improving efficiency compared to the SCAN algorithm.

## Code:

#include <iostream> #include <vector> #include <algorithm>

using namespace std;

class LOOK { private:

vector<int> requests; int headPosition;

int totalMovement;

public:

LOOK(const vector<int>& req, int headPos) : requests(req), headPosition(headPos), totalMovement(0) {}

void simulateLOOK() { sort(requests.begin(), requests.end());

int currentTrack = headPosition;

int direction = 1; // 1 for moving up, -1 for moving down

while (!requests.empty()) { if (direction == 1) {

auto nextRequest = upper\_bound(requests.begin(), requests.end(), currentTrack);

if (nextRequest == requests.end()) { direction = -1; // Change direction

} else {

totalMovement += abs(\*nextRequest - currentTrack); currentTrack = \*nextRequest; requests.erase(nextRequest);

}

} else { // direction == -1

auto prevRequest = lower\_bound(requests.begin(), requests.end(), currentTrack);

if (prevRequest == requests.begin()) { direction = 1; // Change direction

} else {

prevRequest--; // Move to the previous request totalMovement += abs(\*prevRequest - currentTrack); currentTrack = \*prevRequest; requests.erase(prevRequest);

}

}

}

}

int getTotalMovement() const { return totalMovement;

}

};

int main() {

vector<int> requests = {98, 183, 37, 122, 14, 124, 65, 67}; int headPosition = 53;

LOOK look(requests, headPosition); look.simulateLOOK();

int totalMovement = look.getTotalMovement();

cout << "Total head movement using LOOK: " << totalMovement << endl;

return 0;

}

## Output:

The program will output the total head movement required to service the disk requests using the LOOK disk scheduling algorithm.

**Total head movement using LOOK: 299**

## Result:

Execute the program with different disk request sequences and initial head positions to observe how the LOOK algorithm optimizes head movements by scanning only the relevant tracks. Analyze the output to understand the effectiveness of LOOK in disk scheduling.