OS LAB MANUAL

Experiment 1:Practicing of Basic UNIX Commands

1. pwd (Print Working Directory)

Displays the current directory path.

$ pwd

/home/user

---

2. ls (List Directory Contents)

Lists files and directories in the current directory.

$ ls

file1.txt file2.txt folder1

- **Options**:
  - ls -l: Long format (shows details like permissions, size, and date).
  - ls -a: Includes hidden files.

---

3. cd (Change Directory)

Changes to a different directory.

$ cd folder1

$ pwd

/home/user/folder1

- cd ..: Moves to the parent directory.

---

4. mkdir (Make Directory)

Creates a new directory.

$ mkdir new_folder

$ ls

new_folder

---

5. touch (Create Empty Files)

Creates a new, empty file.

$ touch new_file.txt

$ ls

new_file.txt

---

6. cat (Concatenate and Display Files)

Displays the contents of a file.

$ cat file1.txt

Hello, this is a file.

---

7. cp (Copy Files and Directories)

Copies files or directories.

$ cp file1.txt file2.txt

$ ls

file1.txt file2.txt

---

8. mv (Move or Rename Files)

Moves or renames files.

$ mv file1.txt new_name.txt

$ ls

new_name.txt

---

9. rm (Remove Files or Directories)

Deletes files or directories.

$ rm file2.txt

$ ls

new_name.txt

- **Options**:
    - rm -r folder_name: Removes a directory and its contents.

---

10. man (Manual Pages)

Shows the manual for a command.

$ man ls

---

11. chmod (Change File Permissions)

Changes file permissions.

$ chmod 644 file.txt

- Example permissions:
    - 644: Read and write for owner, read-only for others.

---

12. whoami

Displays the current user.

$ whoami

user

---

13. clear

Clears the terminal screen.

$ clear

---

14. find

Searches for files or directories.

$ find . -name "file1.txt"

./folder1/file1.txt

---

15. grep (Search for Text in Files)

Searches for a specific string in a file.

$ grep "text" file.txt

This is the text you searched for.

---

16. df (Disk Free)

Displays disk space usage.

$ df -h

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       50G   20G   30G  40% /
```

---

17. top

Shows real-time system processes.

$ top

---

18. exit

Logs out of the current shell session.

$ exit

Experiment 2:Write programs using the following UNIX operating system calls fork, exec, getpid, exit, wait, close, stat, opendir and readdir

1. fork and getpid

Creates a child process and displays the process IDs of parent and child.

Program:

```
#include <stdio.h>

#include <unistd.h>
```

```
int main() {
pid_t pid = fork();
if (pid == 0) {
// Child process
printf("Child process ID: %d\n", getpid());
} else if (pid > 0) {
// Parent process
printf("Parent process ID: %d\n", getpid());
} else {
perror("Fork failed");
}
return 0;
}
```

---

2. exec

Executes a new program (ls command) from the current process.

Program:

```
#include <stdio.h>
#include <unistd.h>
int main() {
printf("Executing `ls` command:\n");
execl("/bin/ls", "ls", "-l", NULL);
perror("Exec failed");
return 0;
}
```

---

3. exit and wait

Demonstrates a parent process waiting for the child to exit.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
pid_t pid = fork();
if (pid == 0) {
printf("Child process exiting.\n");
exit(0);
} else if (pid > 0) {
int status;
wait(&status);
printf("Parent process: Child exited with status %d.\n",
WEXITSTATUS(status));
} else {
perror("Fork failed");
}
return 0;
}
```

---

4. close

Opens and closes a file descriptor.

Program:

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

```
int main() {
int fd = open("test.txt", O_CREAT | O_RDWR, 0644);
if (fd < 0) {
perror("Failed to open file");
return 1;
}
printf("File opened with descriptor: %d\n", fd);
close(fd);
printf("File descriptor closed.\n");
return 0;
}
```

---

5. stat

Displays information about a file.

Program:

```
#include <stdio.h>
#include <sys/stat.h>
int main() {
struct stat fileStat;
if (stat("test.txt", &fileStat) < 0) {
perror("Failed to get file stats");
return 1;
}

printf("File size: %ld bytes\n", fileStat.st_size);
printf("Permissions: %o\n", fileStat.st_mode & 0777);
```

```
printf("Last accessed: %ld\n", fileStat.st_atime);
return 0;
}
```

---

6. opendir and readdir

Lists the contents of a directory.

Progarm:

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

int main() {
DIR *dir = opendir(".");
if (dir == NULL) {
perror("Failed to open directory");
return 1;
}

struct dirent *entry;
while ((entry = readdir(dir)) != NULL) {
printf("%s\n", entry->d_name);
}

closedir(dir);
return 0;
}
```

---

1. Save the code to a file, e.g., program.c.

2. Compile using gcc:

gcc program.c -o program

3. Run the program:

./program

<u>Experiment 3:</u>Simulate UNIX commands like cp, Is, grep, etc.,

Simulating cp (Copying Files)

```
#include <stdio.h>
#include <stdlib.h>
// Function to copy the content of the source file to the destination file
void copy_file(char *source, char *destination) {
    FILE *src, *dest;
    char ch;

    // Open the source file in read mode
src = fopen(source, "r");
    if (src == NULL) {
perror("Source file opening failed");
        return;
    }

    // Open the destination file in write mode
dest = fopen(destination, "w");
    if (dest == NULL) {
perror("Destination file opening failed");
fclose(src);
        return;
    }
```

```
    // Copy each character from source file to destination file
    while ((ch = fgetc(src)) != EOF) {
fputc(ch, dest);
    }

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

    // Close both the source and destination files
fclose(src);
fclose(dest);
}

int main() {
    char source[100], destination[100];

    // Prompt user for source and destination file names
printf("Enter source file name: ");
scanf("%s", source);
printf("Enter destination file name: ");
scanf("%s", destination);

    // Call the copy_file function to copy the content
copy_file(source, destination);

    return 0;
}
```

Simulating ls (Listing Files in a Directory)

To simulate the ls command, we can use the opendir and readdir functions from the <dirent.h>

```
#include <stdio.h>

#include <dirent.h>


// Function to list all files in the given directory

void list_files(char *path) {

    struct dirent *entry;

    DIR *dir = opendir(path);


    if (dir == NULL) {

perror("Directory opening failed");

        return;

    }


printf("Files in %s:\n", path);

    while ((entry = readdir(dir)) != NULL) {

printf("%s\n", entry->d_name);

    }


closedir(dir);

}


int main() {

    char path[100];


printf("Enter directory path: ");
```

```
scanf("%s", path); // Added missing semicolon here


list_files(path); // Added missing semicolon here


    return 0;
}
```

Simulating grep (Searching for a Pattern in a File)

This program will search for a specific string (pattern) in a file, similar to how grep works in UNIX.

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


// Function to search for a pattern in a file and print matching lines
void grep_pattern(char *file_name, char *pattern) {
    FILE *file;
    char line[256];
    int line_number = 0;


    file = fopen(file_name, "r");
    if (file == NULL) {
perror("File opening failed");
        return;
    }


    while (fgets(line, sizeof(line), file)) {
line_number++;
        if (strstr(line, pattern)) {
printf("Line %d: %s", line_number, line);
```

```
        }
    }

fclose(file);
}

int main() {
    char file_name[100], pattern[100];

printf("Enter file name: ");
scanf("%s", file_name);
printf("Enter pattern to search: ");
scanf("%s", pattern);

grep_pattern(file_name, pattern);

    return 0; // Properly terminate the main function
}
```

Experiment 4: Simulate the following CPU scheduling algorithms a) FCFS b) SJF c) Priority d) Round Robin

1. First-Come, First-Served (FCFS):

FCFS is the simplest scheduling algorithm where the process that arrives first is executed first.

```
#include <stdio.h>

// Function to implement FCFS scheduling
void FCFS(int n, int burst_time[]) {
    int wait_time = 0, turnaround_time = 0;
```

```
    float avg_wait_time = 0, avg_turnaround_time = 0;

    for (int i = 0; i< n; i++) {
wait_time += turnaround_time;
turnaround_time += burst_time[i];
avg_wait_time += wait_time;
avg_turnaround_time += turnaround_time;
    }

printf("FCFS Scheduling:\n");
printf("Average Waiting Time: %.2f\n", avg_wait_time / n);
printf("Average Turnaround Time: %.2f\n", avg_turnaround_time / n);
}

int main() {
    int n;

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

    int burst_time[n];

printf("Enter the burst times of the processes:\n");
    for (int i = 0; i< n; i++) {
printf("Process %d: ", i + 1);
scanf("%d", &burst_time[i]);
    }
```

```
FCFS(n, burst_time);

    return 0;
}
```

2. Shortest Job First (SJF):

SJF schedules processes with the shortest burst time first.

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

// Function to implement Shortest Job First (SJF) scheduling
void SJF(int n, int burst_time[]) {
    int wait_time = 0, turnaround_time = 0;
    float avg_wait_time = 0, avg_turnaround_time = 0;

    // Sort burst times in ascending order
    for (int i = 0; i< n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (burst_time[j] >burst_time[j + 1]) {
                int temp = burst_time[j];
burst_time[j] = burst_time[j + 1];
burst_time[j + 1] = temp;
            }
        }
    }

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

```
wait_time += turnaround_time;
turnaround_time += burst_time[i];
avg_wait_time += wait_time;
avg_turnaround_time += turnaround_time;
    }

printf("SJF Scheduling:\n");
printf("Average Waiting Time: %.2f\n", avg_wait_time / n);
printf("Average Turnaround Time: %.2f\n", avg_turnaround_time / n);
}

int main() {
    int n;

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

    int burst_time[n];

printf("Enter the burst times of the processes:\n");
    for (int i = 0; i< n; i++) {
printf("Process %d: ", i + 1);
scanf("%d", &burst_time[i]); // Fixed missing semicolon
    }

SJF(n, burst_time);
```

```
    return 0; // Properly terminate the main function
}
```

3. Priority Scheduling:

Priority scheduling selects the process with the highest priority (lowest numerical value) first.

```
#include <stdio.h>

// Function to implement Priority Scheduling
void Priority(int n, int burst_time[], int priority[]) {
    int wait_time = 0, turnaround_time = 0;
    float avg_wait_time = 0, avg_turnaround_time = 0;
    int temp_burst, temp_priority;

    // Sort processes based on priority
    for (int i = 0; i< n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (priority[j] >priority[j + 1]) {
                // Swap burst times
temp_burst = burst_time[j];
burst_time[j] = burst_time[j + 1];
burst_time[j + 1] = temp_burst;

                // Swap priorities
temp_priority = priority[j];
                priority[j] = priority[j + 1];
priority[j + 1] = temp_priority;
            }
        }
```

```
    }

    // Calculate waiting and turnaround times
    for (int i = 0; i< n; i++) {
wait_time += turnaround_time;
turnaround_time += burst_time[i];
avg_wait_time += wait_time;
avg_turnaround_time += turnaround_time;
    }

printf("Priority Scheduling:\n");
printf("Average Waiting Time: %.2f\n", avg_wait_time / n);
printf("Average Turnaround Time: %.2f\n", avg_turnaround_time / n);
}

int main() {
    int n;

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

    int burst_time[n], priority[n];

printf("Enter the burst times and priorities of the processes:\n");
    for (int i = 0; i< n; i++) {
printf("Process %d:\n", i + 1);
printf("Burst time: ");
```

```
scanf("%d", &burst_time[i]);
printf("Priority: ");
scanf("%d", &priority[i]);
    }

Priority(n, burst_time, priority);

    return 0;
}
```

4. Round Robin (RR):

Round Robin scheduling allocates a fixed time quantum for each process. If a process doesn't complete within its time slice, it's moved to the back of the queue.

```
#include <stdio.h>
void RoundRobin(int n, int burst_time[], int quantum) {
int wait_time = 0, turnaround_time = 0;
int remaining_burst_time[n];
float avg_wait_time = 0, avg_turnaround_time = 0;
// Copy burst time to remaining_burst_time
for (int i = 0; i < n; i++) {
remaining_burst_time[i] = burst_time[i];
}
int time = 0;
while (1) {
int completed = 1;
for (int i = 0; i < n; i++) {
if (remaining_burst_time[i] > 0) {
completed = 0;
```

```
if (remaining_burst_time[i] > quantum) {
time += quantum;
remaining_burst_time[i] -= quantum;
} else {
time += remaining_burst_time[i];
wait_time += time - burst_time[i];
remaining_burst_time[i] = 0;
}
}
}
if (completed) break;
}
for (int i = 0; i < n; i++) {
turnaround_time += burst_time[i];
avg_turnaround_time += turnaround_time;
}
avg_wait_time = (float)wait_time / n;
avg_turnaround_time = (float)avg_turnaround_time / n;
printf("Round Robin Scheduling (Quantum = %d):\n", quantum);
printf("Average Waiting Time: %.2f\n", avg_wait_time);
printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
}
int main() {
int n, quantum;
printf("Enter the number of processes: ");
scanf("%d", &n);
int burst_time[n];
```

```c
printf("Enter the burst times of the processes:\n");
for (int i = 0; i < n; i++) {
printf("Process %d: ", i + 1);
scanf("%d", &burst_time[i]);
}
printf("Enter the time quantum: ");
scanf("%d", &quantum);
RoundRobin(n, burst_time, quantum);
return 0;
}
```

Compile the programs using gcc:

```
gcc FCFS.c -o FCFS
gcc SJF.c -o SJF
gcc Priority.c -o Priority
gcc RR.c -o RR
```

Run the executable:

```
./FCFS
./SJF
./Priority
./RR
```

Experiment 5:Control the number of ports opened by the operating system with a) Semaphore b) Monitors.

1. Using Semaphore to Control the Number of Ports Opened

A **semaphore** is a synchronization primitive that can be used to control access to shared resources. The semaphore works by maintaining a counter that represents the number of available resources. If the counter is positive, processes can acquire a resource; if the counter is zero, processes must wait until the resource becomes available.

Example of Controlling Ports Using Semaphore

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

#define MAX_PORTS 3  // Maximum number of ports that can be opened

sem_tavailable_ports;  // Semaphore to control access to ports

void* open_port(void* id) {
    int thread_id = *(int*)id;
printf("Thread %d: Attempting to open a port...\n", thread_id);

    // Wait for an available port
sem_wait(&available_ports);
printf("Thread %d: Port opened.\n", thread_id);

    // Simulate port usage
sleep(2);

    // Release the port
printf("Thread %d: Closing port.\n", thread_id);
sem_post(&available_ports);

    return NULL;
}
```

```
int main() {
pthread_tthreads[5];  // 5 threads, trying to open ports
    int thread_ids[5];

    // Initialize semaphore with MAX_PORTS available ports
sem_init(&available_ports, 0, MAX_PORTS);

    // Create 5 threads, each trying to open a port
    for (int i = 0; i< 5; i++) {
thread_ids[i] = i + 1;
pthread_create(&threads[i], NULL, open_port, &thread_ids[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i< 5; i++) {
pthread_join(threads[i], NULL);
    }

    // Destroy the semaphore
sem_destroy(&available_ports);

    return 0;
}
```

2. Using Monitors to Control the Number of Ports Opened

A **monitor** is a higher-level synchronization construct that encapsulates both data and methods for manipulating that data. It ensures that only one thread can execute the critical section of code at a time. In C, we can use mutexes or condition variables (through POSIX threads or pthread library) to implement monitors.

Example of Controlling Ports Using Monitors

Here, we will implement a monitor-like behavior using mutexes and condition variables to simulate controlling the number of ports.

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>  // For sleep function


#define MAX_PORTS 3  // Maximum number of ports that can be opened


pthread_mutex_tmutex;  // Mutex to protect shared resource (ports)

pthread_cond_tcond;    // Condition variable for synchronization

int available_ports = MAX_PORTS;  // Shared resource (number of available
ports)


void* open_port(void* id) {

    int thread_id = *(int*)id;

printf("Thread %d: Attempting to open a port...\n", thread_id);


pthread_mutex_lock(&mutex);  // Enter critical section


    // Wait if no ports are available

    while (available_ports == 0) {

pthread_cond_wait(&cond, &mutex);

    }
```

```
    // Open a port
available_ports--;
printf("Thread %d: Port opened. Remaining ports: %d\n", thread_id,
available_ports);
pthread_mutex_unlock(&mutex);  // Exit critical section

    // Simulate port usage
sleep(2);

pthread_mutex_lock(&mutex);  // Enter critical section to close port
available_ports++;  // Release the port
printf("Thread %d: Closing port. Remaining ports: %d\n", thread_id,
available_ports);

pthread_cond_signal(&cond);  // Signal other threads that a port is available
pthread_mutex_unlock(&mutex);  // Exit critical section

    return NULL;
}

int main() {
pthread_tthreads[5];  // 5 threads, trying to open ports
    int thread_ids[5];

    // Initialize the mutex and condition variable
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond, NULL);
```

```
    // Create 5 threads, each trying to open a port
    for (int i = 0; i< 5; i++) {
thread_ids[i] = i + 1;
pthread_create(&threads[i], NULL, open_port, &thread_ids[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i< 5; i++) {
pthread_join(threads[i], NULL);
    }

    // Destroy the mutex and condition variable
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&cond);

    return 0;
}
```

<u>Experiment 6:</u>Write a program to illustrate concurrent execution of threads using threads library.

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

void* print_message(void* thread_id) {
    long tid = (long) thread_id;
printf("Thread %ld is running\n", tid);
sleep(2);  // Simulate some work by sleeping for 2 seconds
printf("Thread %ld is done\n", tid);
```

```
    return NULL;
}

int main() {
pthread_tthreads[5];  // Array to hold thread identifiers
    int num_threads = 5;

    // Create 5 threads
    for (long i = 0; i<num_threads; i++) {
        int result = pthread_create(&threads[i], NULL, print_message, (void*)i);
        if (result != 0) {
perror("Thread creation failed");
            return -1;
        }
    }

    // Wait for all threads to finish
    for (int i = 0; i<num_threads; i++) {
pthread_join(threads[i], NULL);
    }

printf("All threads have finished execution.\n");
    return 0;
}
```

<u>Experiment 7:</u>Write a program to solve producer-consume problem using Semaphores.

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

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

#define BUFFER_SIZE 5   // Size of the shared buffer
#define PRODUCER_COUNT 3 // Number of producers
#define CONSUMER_COUNT 3 // Number of consumers

int buffer[BUFFER_SIZE];  // Shared buffer
int in = 0, out = 0;      // Indices for producer and consumer
sem_t empty, full, mutex;  // Semaphores

void* producer(void* arg) {
    for (int i = 0; i< 5; i++) {
        int item = rand() % 100; // Produce a random item
sem_wait(&empty);  // Decrement empty (wait for space)
sem_wait(&mutex);  // Enter critical section
        buffer[in] = item; // Add item to buffer
printf("Producer %ld produced item: %d at index %d\n", (long)arg, item, in);
        in = (in + 1) % BUFFER_SIZE; // Move to next position
sem_post(&mutex);  // Exit critical section
sem_post(&full);   // Increment full (notify consumer)
sleep(1);  // Simulate time taken to produce
    }
    return NULL;
}
```

```
void* consumer(void* arg) {
    for (int i = 0; i< 5; i++) {
sem_wait(&full);   // Decrement full (wait for item)
sem_wait(&mutex);  // Enter critical section
        int item = buffer[out]; // Consume an item from the buffer
printf("Consumer %ld consumed item: %d from index %d\n", (long)arg, item,
out);
        out = (out + 1) % BUFFER_SIZE; // Move to next position
sem_post(&mutex);  // Exit critical section
sem_post(&empty);  // Increment empty (notify producer)
sleep(2);  // Simulate time taken to consume
    }
    return NULL;
}

int main() {
pthread_tproducers[PRODUCER_COUNT],
consumers[CONSUMER_COUNT];

    // Initialize semaphores
sem_init(&empty, 0, BUFFER_SIZE);  // Initially, the buffer is empty
sem_init(&full, 0, 0);             // Initially, no items are produced
sem_init(&mutex, 0, 1);            // Mutex for mutual exclusion in buffer

    // Create producer threads
    for (long i = 0; i< PRODUCER_COUNT; i++) {
pthread_create(&producers[i], NULL, producer, (void*)i);
    }
```

```
    // Create consumer threads
    for (long i = 0; i< CONSUMER_COUNT; i++) {
pthread_create(&consumers[i], NULL, consumer, (void*)i);
    }

    // Wait for all producer threads to finish
    for (int i = 0; i< PRODUCER_COUNT; i++) {
pthread_join(producers[i], NULL);
    }

    // Wait for all consumer threads to finish
    for (int i = 0; i< CONSUMER_COUNT; i++) {
pthread_join(consumers[i], NULL);
    }

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

    return 0;
}
```

<u>Experiment 8:</u>Implement the following memory allocation methods for fixed partition a) First fit b) Worst fit c) Best fit

1. First Fit Algorithm

```
#include <stdio.h>
#define PARTITION_COUNT 5  // Number of fixed partitions
```

```
#define PROCESS_COUNT 4    // Number of processes

// Function to perform First Fit allocation
void firstFit(int partitions[], int partitionCount, int processes[], int
processCount) {
    int allocation[processCount];  // Declare allocation array with correct size

    // Initially, no process is allocated
    for (int i = 0; i<processCount; i++) {
        allocation[i] = -1;
    }

    // Try to allocate memory for each process
    for (int i = 0; i<processCount; i++) {
        for (int j = 0; j <partitionCount; j++) {
            if (partitions[j] >= processes[i]) {
                allocation[i] = j;
                partitions[j] -= processes[i];  // Allocate the partition
                break;
            }
        }
    }

    // Output the result
printf("\nFirst Fit Allocation:\n");
    for (int i = 0; i<processCount; i++) {
        if (allocation[i] != -1) {
printf("Process %d allocated to Partition %d\n", i + 1, allocation[i] + 1);
```

```
        } else {
printf("Process %d not allocated\n", i + 1);
        }
    }
}

int main() {
    int partitions[PARTITION_COUNT] = {100, 500, 200, 300, 600}; // Partition sizes
    int processes[PROCESS_COUNT] = {212, 417, 112, 426};        // Process sizes
firstFit(partitions, PARTITION_COUNT, processes, PROCESS_COUNT);
    return 0;
}
```

2. **Worst Fit Algorithm**

```
#include <stdio.h>
#define PARTITION_COUNT 5  // Number of fixed partitions
#define PROCESS_COUNT 4    // Number of processes

// Function to perform Worst Fit allocation
void worstFit(int partitions[], int partitionCount, int processes[], int processCount) {
    int allocation[processCount];  // Allocation array to store partition assignments

    // Initially, no process is allocated
    for (int i = 0; i<processCount; i++) {
        allocation[i] = -1;
    }
```

```
    // Try to allocate memory for each process
    for (int i = 0; i<processCount; i++) {
        int maxIndex = -1;
        int maxSize = -1;

        // Find the partition with the largest size that can accommodate the process
        for (int j = 0; j <partitionCount; j++) {
            if (partitions[j] >= processes[i] && partitions[j] >maxSize) {
maxSize = partitions[j];
maxIndex = j;
            }
        }

        // If a suitable partition is found, allocate it
        if (maxIndex != -1) {
            allocation[i] = maxIndex;
            partitions[maxIndex] -= processes[i];  // Allocate the partition
        }
    }

    // Output the result
printf("\nWorst Fit Allocation:\n");
    for (int i = 0; i<processCount; i++) {
        if (allocation[i] != -1) {
printf("Process %d allocated to Partition %d\n", i + 1, allocation[i] + 1);
        } else {
```

```
printf("Process %d not allocated\n", i + 1);
        }
    }
}

int main() {
    int partitions[PARTITION_COUNT] = {100, 500, 200, 300, 600}; // Partition
sizes
    int processes[PROCESS_COUNT] = {212, 417, 112, 426};        // Process
sizes
worstFit(partitions, PARTITION_COUNT, processes, PROCESS_COUNT);
    return 0;
}
```

3.Best Fit Algorithm

```
#include <stdio.h>
#define PARTITION_COUNT 5  // Number of fixed partitions
#define PROCESS_COUNT 4    // Number of processes
// Function to perform Best Fit allocation
void bestFit(int partitions[], int partitionCount, int processes[], int
processCount) {
int allocation[processCount];
// Initially, no process is allocated
for (int i = 0; i < processCount; i++) {
allocation[i] = -1;
}
// Try to allocate memory for each process
for (int i = 0; i < processCount; i++) {
int minIndex = -1;
```

```
int minSize = 9999999;
// Find the partition with the smallest size that can accommodate the process
for (int j = 0; j < partitionCount; j++) {
if (partitions[j] >= processes[i] && partitions[j] < minSize) {
minSize = partitions[j];
minIndex = j;
}
}
// If a suitable partition is found, allocate it
if (minIndex != -1) {
allocation[i] = minIndex;
partitions[minIndex] -= processes[i];  // Allocate the partition
}
}
// Output the result
printf("\nBest Fit Allocation:\n");
for (int i = 0; i < processCount; i++) {
if (allocation[i] != -1) {
printf("Process %d allocated to Partition %d\n", i + 1, allocation[i] + 1);
} else {
printf("Process %d not allocated\n", i + 1);
}
}
}

int main() {
int partitions[PARTITION_COUNT] = {100, 500, 200, 300, 600}; // Partition
sizes
```

```
int processes[PROCESS_COUNT] = {212, 417, 112, 426};        // Process sizes

bestFit(partitions, PARTITION_COUNT, processes, PROCESS_COUNT);
return 0;
}
```

<u>Experiment 9:</u>Simulate the following page replacement algorithms a) FIFO b) LRU c) LFU

1. **FIFO (First-In-First-Out)**

```
#include <stdio.h>
#include <stdlib.h>
#define FRAME_COUNT 4 // Number of frames in memory

// Function to simulate FIFO page replacement
void fifo(int pages[], int pageCount) {
    int frames[FRAME_COUNT];
    int pageFaults = 0;
    int i, j;
    int isPresent;

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

printf("FIFO Page Replacement:\n");

    for (i = 0; i<pageCount; i++) {
isPresent = 0;
```

```
        // Check if the page is already in memory
        for (j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == pages[i]) {
isPresent = 1;
                break;
            }
        }

        // Page fault if the page is not in memory
        if (!isPresent) {
frames[pageFaults % FRAME_COUNT] = pages[i]; // Replace the oldest
page
pageFaults++;
        }

        // Print current memory state
printf("After accessing page %d: ", pages[i]);
        for (j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == -1) {
printf(" _ ");
            } else {
printf(" %d ", frames[j]);
            }
        }
printf("\n");
    }
```

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

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 3};
    int pageCount = sizeof(pages) / sizeof(pages[0]);

fifo(pages, pageCount);
    return 0;
}
```

2. LRU (Least Recently Used)

```
#include <stdio.h>
#include <stdlib.h>
#define FRAME_COUNT 4 // Number of frames in memory

// Function to simulate LFU page replacement
void lfu(int pages[], int pageCount) {
    int frames[FRAME_COUNT];
    int frequency[FRAME_COUNT] = {0}; // Frequency of page access
    int pageFaults = 0;
    int i, j;
    int isPresent, lfuIndex, minFrequency;

    // Initialize frames to -1 (empty)
    for (i = 0; i< FRAME_COUNT; i++) {
        frames[i] = -1;
    }
```

```
printf("LFU Page Replacement:\n");

    for (i = 0; i<pageCount; i++) {
isPresent = 0;

        // Check if the page is already in memory
        for (j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == pages[i]) {
isPresent = 1;
                frequency[j]++; // Increment frequency for this page
                break;
            }
        }

        // Page fault if the page is not in memory
        if (!isPresent) {
            // Find the least frequently used page
lfuIndex = 0;
minFrequency = frequency[0];
            for (j = 1; j < FRAME_COUNT; j++) {
                if (frequency[j] <minFrequency) {
minFrequency = frequency[j];
lfuIndex = j;
                }
            }
            frames[lfuIndex] = pages[i];
```

```
            frequency[lfuIndex] = 1; // Set frequency to 1 for the new page
pageFaults++;
        }

        // Print current memory state
printf("After accessing page %d: ", pages[i]);
        for (j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == -1) {
printf(" _ ");
            } else {
printf(" %d ", frames[j]);
            }
        }
printf("\n");
    }

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

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 3};
    int pageCount = sizeof(pages) / sizeof(pages[0]);  // Missing semicolon
here

lfu(pages, pageCount);
    return 0;
}
```

3. LFU (Least Frequently Used)

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

#define FRAME_COUNT 4 // Number of frames in memory

// Function to simulate LFU page replacement
void lfu(int pages[], int pageCount) {
    int frames[FRAME_COUNT];
    int frequency[FRAME_COUNT] = {0}; // Frequency of page access
    int pageFaults = 0;
    int i, j;
    int isPresent, lfuIndex, minFrequency; // Added semicolon here

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

printf("LFU Page Replacement:\n");

    for (i = 0; i<pageCount; i++) {
isPresent = 0;

        // Check if the page is already in memory
        for (j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == pages[i]) {
isPresent = 1;
```

```
                frequency[j]++; // Increment frequency for this page
                break;
            }
        }

        // Page fault if the page is not in memory
        if (!isPresent) {
            // Find the least frequently used page
lfuIndex = 0;
minFrequency = frequency[0];
            for (j = 1; j < FRAME_COUNT; j++) {
                if (frequency[j] <minFrequency) {
minFrequency = frequency[j];
lfuIndex = j;
                }
            }
            frames[lfuIndex] = pages[i];
            frequency[lfuIndex] = 1; // Set frequency to 1 for the new page
pageFaults++;
        }

        // Print current memory state
printf("After accessing page %d: ", pages[i]);
        for (j = 0; j < FRAME_COUNT; j++) {
            if (frames[j] == -1) {
printf(" _ ");
            } else {
```

```
printf(" %d ", frames[j]);
            }
        }
printf("\n");
    }

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

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 3};
    int pageCount = sizeof(pages) / sizeof(pages[0]);
lfu(pages, pageCount);
    return 0;
}
```

Experiment 10:Simulate Paging Technique of memory management.

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

#define FRAME_COUNT 4   // Number of frames in physical memory
#define PAGE_COUNT 8    // Number of pages in the process
#define PAGE_SIZE 4     // Size of each page (number of addresses per page)

// Page Table: Maps page numbers to frame numbers
int pageTable[PAGE_COUNT] = {-1};  // Initialize all page entries as -1 (not allocated)
```

```
// Function to simulate page allocation and memory mapping
void pagingSimulation(int processPages[], int pageCount) {
    int physicalMemory[FRAME_COUNT][PAGE_SIZE] = {0};  // Physical
memory divided into frames
    int pageFaults = 0;
    int i, pageIndex, frameIndex;

printf("Paging Simulation:\n");

    for (i = 0; i<pageCount; i++) {
pageIndex = processPages[i];

        // Check if the page is already mapped to a frame (no page fault)
        if (pageTable[pageIndex] != -1) {
frameIndex = pageTable[pageIndex];
printf("Page %d is already in frame %d\n", pageIndex, frameIndex);
        } else {
            // Page fault, find an empty frame
            int foundEmptyFrame = 0;
            for (frameIndex = 0; frameIndex< FRAME_COUNT; frameIndex++)
{
                if (physicalMemory[frameIndex][0] == 0) {  // Frame is empty
                    // Allocate the page to the frame
pageTable[pageIndex] = frameIndex;
physicalMemory[frameIndex][0] = pageIndex;
foundEmptyFrame = 1;
pageFaults++;
```

```
printf("Page %d caused a page fault. Allocating to frame %d\n", pageIndex,
frameIndex);
                    break;
                }
            }

            if (!foundEmptyFrame) {
                // If no empty frame, replace the first frame (simple replacement
policy)
frameIndex = 0;
pageTable[pageIndex] = frameIndex;
physicalMemory[frameIndex][0] = pageIndex;
printf("Page %d caused a page fault. Replacing frame %d\n", pageIndex,
frameIndex);
pageFaults++;
            }
        }

        // Display the page table and physical memory status
printf("Current Page Table:\n");
        for (int j = 0; j < PAGE_COUNT; j++) {
printf("Page %d -> Frame %d\n", j, pageTable[j]);
        }
printf("Current Physical Memory:\n");
        for (int j = 0; j < FRAME_COUNT; j++) {
            if (physicalMemory[j][0] != 0) {
printf("Frame %d: Page %d\n", j, physicalMemory[j][0]);
            } else {
```

```
printf("Frame %d: Empty\n", j);
            }
        }

printf("\n");
    }

printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    // Simulate a process with page references (logical addresses)
    int processPages[] = {2, 4, 1, 2, 5, 6, 3, 2, 4, 1};  // Logical page accesses
    int pageCount = sizeof(processPages) / sizeof(processPages[0]);  // Added
semicolon here

pagingSimulation(processPages, pageCount);  // Added semicolon here

    return 0;
}
```

Experiment 11:Implement Bankers Algoriocat Dest Lock avoidance and prevention

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

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

```
// Function to check if a state is safe using Banker's Algorithm
bool isSafeState(int processes[], int avail[], int max[][R], int allot[][R], int
need[][R]) {
    int work[R], finish[P];
    int safeSeq[P];

    // Initialize work[] and finish[] arrays
    for (int i = 0; i< R; i++) {
        work[i] = avail[i];
    }
    for (int i = 0; i< P; i++) {
        finish[i] = 0;
    }

    int count = 0; // Number of processes that can finish
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                bool canFinish = true;
                // Check if the process can be finished (all needed resources are
available)
                for (int j = 0; j < R; j++) {
                    if (need[p][j] > work[j]) {
canFinish = false;
                        break;
                    }
                }
```

```
}

                if (canFinish) {
                    // Add allocated resources to work[] and mark the process as
finished
                    for (int j = 0; j < R; j++) {
                        work[j] += allot[p][j];
                    }
                    finish[p] = 1; // Mark process p as finished
safeSeq[count++] = p;
                    found = true;
                }
            }
        }
        if (!found) {
            return false; // No process can finish, system is in unsafe state
        }
    }

    // Print the safe sequence
printf("Safe Sequence: ");
    for (int i = 0; i< P; i++) {
printf("P%d ", safeSeq[i]);
    }
printf("\n");

    return true; // System is in a safe state
}

// Function to request resources using Banker's Algorithm
```

```
bool requestResources(int p, int request[], int processes[], int avail[], int
max[][R], int allot[][R], int need[][R]) {
    // Check if the request is less than or equal to the needed resources
    for (int i = 0; i< R; i++) {
        if (request[i] > need[p][i]) {
printf("Error: Process has exceeded its maximum claim\n");
            return false;
        }
    }
    // Check if the request is less than or equal to the available resources
    for (int i = 0; i< R; i++) {
        if (request[i] > avail[i]) {
printf("Error: Resources are not available\n");
            return false;
        }
    }
    // Pretend to allocate resources and check if the system is still in a safe
state
    for (int i = 0; i< R; i++) {
        avail[i] -= request[i];
        allot[p][i] += request[i];
        need[p][i] -= request[i];
    }

    // Now check if the system is still in a safe state after allocation
    if (isSafeState(processes, avail, max, allot, need)) {
printf("Resources allocated successfully\n");
        return true;
```

```
    } else {
        // Rollback the allocation if not in a safe state
        for (int i = 0; i< R; i++) {
            avail[i] += request[i];
            allot[p][i] -= request[i];
            need[p][i] += request[i];
        }
printf("Resources could not be allocated due to unsafe state\n");
        return false;
    }
}

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

    // Maximum resources needed by each process
    int max[P][R] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    // Resources allocated to each process
    int allot[P][R] = {
```

```
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    // Calculate the need matrix (max - allot)
    int need[P][R];
    for (int i = 0; i< P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }

    // Check if the system is in a safe state initially
    if (isSafeState(processes, avail, max, allot, need)) {
printf("System is in a safe state\n");
    } else {
printf("System is in an unsafe state\n");
    }

    // Request resources for process 1 (e.g., request 1 unit of resource 0, 0 of resource 1, 2 of resource 2)
    int request[] = {1, 0, 2};
requestResources(1, request, processes, avail, max, allot, need);

    return 0;
```

}

Experiment 12:Simulate the following file allocation strategies a) Sequential b) Indexed e) Linked explain all with code

a) Sequential Allocation:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_BLOCKS 10  // Maximum number of blocks in the file
system


typedef struct {

   int block[MAX_BLOCKS];  // Represents the storage blocks

   int allocated[MAX_BLOCKS];  // Allocation status (1 for allocated, 0 for
free)

} Disk;


void sequentialAllocation(Disk *disk, int fileSize) {

   int blocksNeeded = (fileSize + MAX_BLOCKS - 1) / MAX_BLOCKS;

   int i;

   for (i = 0; i<blocksNeeded; i++) {

      if (disk->allocated[i] == 1) {

printf("Block %d already allocated!\n", i);

      } else {

         disk->allocated[i] = 1;

printf("Allocating block %d to file\n", i);

      }

   }

}  // <-- Closing brace for sequentialAllocation function
```

```
int main() {
    Disk disk = {0};
    int fileSize = 15;  // Size of the file to be allocated in blocks
sequentialAllocation(&disk, fileSize);
    return 0;
}
```

b) Indexed Allocation:

```
#include <stdio.h>
#include <stdlib.h>  // Closing bracket added here.

#define MAX_BLOCKS 10  // Maximum number of blocks in the file
system
#define INDEX_SIZE 3   // Size of the index block

typedef struct {
    int block[MAX_BLOCKS];  // Represents the storage blocks
    int allocated[MAX_BLOCKS];  // Allocation status (1 for allocated, 0 for
free)
    int index[INDEX_SIZE];  // Index block with pointers to data blocks
} Disk;

void indexedAllocation(Disk *disk, int fileSize) {
    int blocksNeeded = (fileSize + MAX_BLOCKS - 1) / MAX_BLOCKS;
    int i;
    for (i = 0; i<blocksNeeded; i++) {
        if (disk->allocated[i] == 1) {
printf("Block %d already allocated!\n", i);
```

```
        } else {
            disk->allocated[i] = 1;
            disk->index[i % INDEX_SIZE] = i;  // Store block index in the index
block
printf("Allocating block %d to file (Index: %d)\n", i, i % INDEX_SIZE);
        }
    }
}

int main() {
    Disk disk = {0};
    int fileSize = 12;  // Size of the file to be allocated in blocks
indexedAllocation(&disk, fileSize);
    return 0;
}
```

c) Linked Allocation:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 10  // Maximum number of blocks in the file
system
typedef struct {
int block[MAX_BLOCKS];  // Represents the storage blocks
int allocated[MAX_BLOCKS];  // Allocation status (1 for allocated, 0 for
free)
int next[MAX_BLOCKS];  // Pointers to the next block in the chain
} Disk;
void linkedAllocation(Disk *disk, int fileSize) {
int blocksNeeded = (fileSize + MAX_BLOCKS - 1) / MAX_BLOCKS;
```

```
int i, prevBlock = -1;
for (i = 0; i < blocksNeeded; i++) {
int blockIndex = -1;
for (int j = 0; j < MAX_BLOCKS; j++) {
if (disk->allocated[j] == 0) {
blockIndex = j;
break;
}
}
if (blockIndex == -1) {
printf("No available blocks for allocation\n");
return;
} else {
disk->allocated[blockIndex] = 1;
if (prevBlock != -1) {
disk->next[prevBlock] = blockIndex;  // Set pointer from previous block
}
disk->next[blockIndex] = -1;  // Last block points to -1 (end of chain)

if (prevBlock == -1) {
printf("Allocating block %d to file (Start of chain)\n", blockIndex);
} else {
printf("Allocating block %d to file (Linked to block %d)\n", blockIndex,
prevBlock);
}
prevBlock = blockIndex;
}
}
```

```
}
int main() {
Disk disk = {0};
int fileSize = 6;  // Size of the file to be allocated in blocks
linkedAllocation(&disk, fileSize);
return 0;
}
```