

**School of Computing Science & Engineering (SCOPE)**

**Practical Record File**

**Operating System (CSE3003)**

**Slot: E11+E12+C14**

**Class no: 0629**

**Submitted By: Submitted to:**

**OM RAJ Dr. Gunjan Ansari**

**Associate Professor, SCSE**

**Indicative List of Experiments**

**(All the experiments need to be done in C++/Java Language)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **List of Experiments** | **Date Conducted** | **Remarks** |
| 1 | **Learn and practice basic OS commands.** |  |  |
| 2 | **C program to create a clone of current process** |  |  |
| 3 | **Write a C program to create a new process using fork()** |  |  |
| 4 | **C program that implements a parent-child process relationship and demonstrates process termination using fork() and wait() system calls** |  |  |
| 5 | **Write a C program to simulate first come first serve(fcfs) scheduling** |  |  |
| 6 | Write a C program to implement shortest job first scheduling with or without premption |  |  |
| 7 | **Write a C program to Implement priority scheduling and test with different sets of priorities** |  |  |
| 8 | **Write a C program to simulate round robin scheduling and turn around and waiting tim** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **List of Experiments** | **Date Conducted** | **Remarks** |
| 9 | **C Program to Implement Producer-Consumer Problem** |  |  |
| 10 | **The Dining Philosophers Problem** |  |  |
| 11 | **The Reader-Writer problem** |  |  |
| 12 | **C program to implement Bankers algorithm of deadlock avoidance in operating system** |  |  |

# Experiment - 1

Basic Linux/Unix Commands

## Aim: Learn and practice basic OS commands.

1. File Management Commands and Examples:
   * \*\*`ls`\*\*: Lists files and directories in the current directory.

```bash ls

ls -l ls -a

```

- \*\*`cp`\*\*: Copies files or directories.

```bash touch file1.txt

cp file1.txt file2.txt

```

- \*\*`mv`\*\*: Moves or renames files or directories.

```bash

mv file2.txt file3.txt

```

* \*\*`rm`\*\*: Removes files or directories.

```bash rm file3.txt

```

* \*\*`touch`\*\*: Creates an empty file.

```bash touch file4.txt

1. Process Management Commands and Examples:
   * \*\*`ps`\*\*: Displays information about active processes.

```bash ps

ps aux

```

* + \*\*`kill`\*\*: Terminates a process using its PID (Process ID).

```bash ps aux

kill <PID>

```

* + \*\*`top`\*\*: Displays real-time process monitoring.

```bash top

```

---

1. Directory Navigation Commands and Examples:
   * \*\*`cd`\*\*: Changes the current directory.

```bash

cd /home/user/Documents

```

* + \*\*`pwd`\*\*: Prints the current working directory.

```bash pwd

1. I/O Redirection and Piping Commands and Examples:
   * \*\*`>`\*\*: Redirects standard output to a file (overwrites the file).

```bash

echo "Hello, World!" > output.txt

```

* + \*\*`>>`\*\*: Appends standard output to a file.

```bash

echo "This is appended text." >> output.txt

```

* + \*\*`|`\*\*: Pipes the output of one command to another as input.

```bash

ls -l | grep "file"

# Experiment - 2

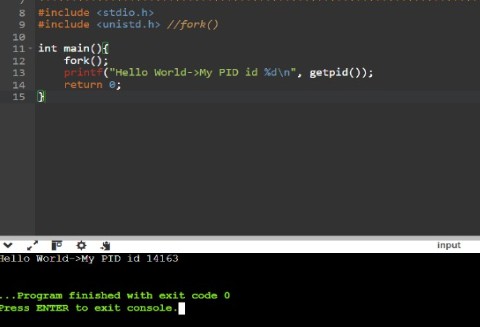
## AIM: C program to create a clone of current process

#include <stdio.h> #include <unistd.h> //fork()

int main(){ fork();

printf("Hello World->My PID id %d\n", getpid()); return 0;

}



# Experiment - 3

## AIM: Write a C program to create a new process using fork()

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

int main(){

// create a new process pid\_t pid = fork();

// Error Handling if(pid<0){

// Fork failed printf("Fork failed \n"); return 1;

}

else if(pid==0){

// child process printf("Child Process\n");

printf("Child Process ID:%d\n",getpid()); printf("Parent process ID:%d\n", getppid());

}

else{

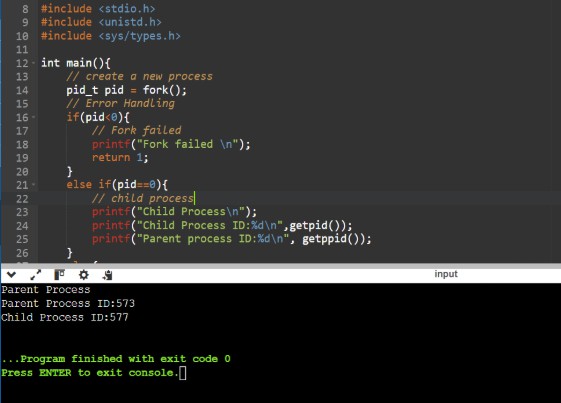
// Parent Process printf("Parent Process\n");

printf("Parent Process ID:%d\n", getpid()); printf("Child Process ID:%d\n", pid);

}

return 0;

}



# Experiment - 4

## AIM: C program that implements a parent-child process relationship and demonstrates process termination using fork() and wait() system calls.

#include <stdlib.h> #include <unistd.h>

#include <sys/types.h> #include <sys/wait.h>

int main(){

pidt pid = fork(); if(pid<0){

// fork failed printf("Fork Failed\n"); return 1;

}

else if(pid==0){

// Child process printf("Child process:\n");

printf("child PID:%d\n", getpid()); printf("Parent PID:%d\n", getppid()); printf("Child Process will terminate now\n"); exit(0);

}

else{

// Parent Process printf("Parent Process:\n")

printf("Parent PID:%d\n", getpid()); printf("Waiting for child process to terminate..\n");

// wait for child process to terminate int status;

wait(&status); if(WIFEXITED(status)){

printf("Child Process terminated withexit status:%d\n", WEXITSTATUS(status));

}

else{

printf("Child process did not terminate successfully\n");

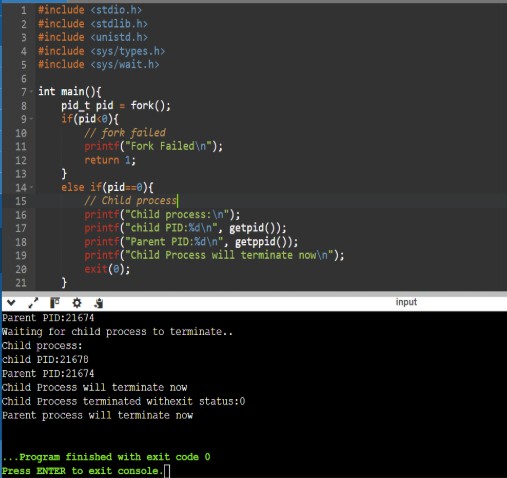
}

printf("Parent process will terminate now\n");

}

return 0;

}



# Experiment - 5

## AIM: Write a C program to simulate first come first serve(fcfs) scheduling

#include <stdio.h> struct process

{

int pid;

int arrival\_time; int burst\_time; int waiting\_time;

int turnaround\_time;

};

int main()

{

int n;

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

struct process processes[n]; for (int i = 0; i < n; i++)

{

printf("Enter details for process %d:\n", i + 1); printf("Arrival Time: ");

scanf("%d", &processes[i].arrival\_time); printf("Burst Time: ");

scanf("%d", &processes[i].burst\_time);

processes[i].pid = i + 1; processes[i].waiting\_time = 0;

processes[i].turnaround\_time = 0;

}

// Sort processes by arrival time in ascending order for (int i = 0; i < n - 1; i++)

{

for (int j = i + 1; j < n; j++)

{

if (processes[i].arrival\_time > processes[j].arrival\_time)

{

struct process temp = processes[i]; processes[i] = processes[j]; processes[j] = temp;

}

}

}

// Calculate waiting time and turnaround time for each process int time = 0;

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

{

if (time < processes[i].arrival\_time)

{

time = processes[i].arrival\_time;

}

processes[i].waiting\_time = time - processes[i].arrival\_time;

time += processes[i].burst\_time;

processes[i].turnaround\_time = time - processes[i].arrival\_time;

}

// Calculate average waiting time and turnaround time float avg\_waiting\_time = 0.0, avg\_turnaround\_time = 0.0; for (int i = 0; i < n; i++)

{

avg\_waiting\_time += processes[i].waiting\_time; avg\_turnaround\_time += processes[i].turnaround\_time;

}

avg\_waiting\_time /= n; avg\_turnaround\_time /= n;

// Print the results

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int i = 0; i < n; i++)

{

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time,

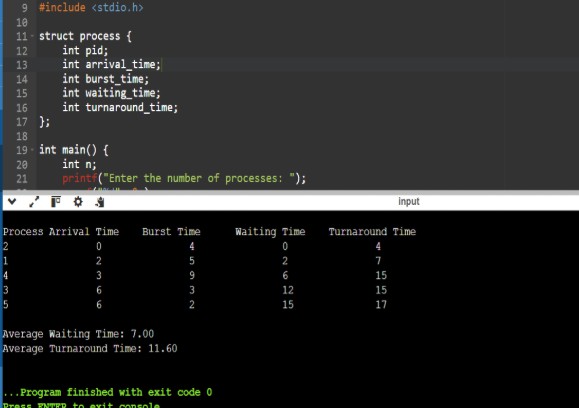
processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time); printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

return 0;

}



# Experiment - 6

## AIM: Write a C program to implement shortest job first scheduling with or without premption

#include <stdio.h>

#define MAX\_PROCESSES 100 struct process {

int pid;

int arrival\_time; int burst\_time;

int remaining\_time; int start\_time;

int waiting\_time;

int turnaround\_time;

};

int main() { int n;

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

if (n > MAX\_PROCESSES) {

printf("Error: Maximum number of processes exceeded.\n"); return 1;

}

struct process processes[MAX\_PROCESSES]; for (int i = 0; i < n; i++) {

printf("Enter details for process %d:\n", i + 1); printf("Arrival Time: ");

scanf("%d", &processes[i].arrival\_time); printf("Burst Time: ");

scanf("%d", &processes[i].burst\_time);

processes[i].pid = i + 1;

processes[i].remaining\_time = processes[i].burst\_time; processes[i].start\_time = 0;

processes[i].waiting\_time = 0;

processes[i].turnaround\_time = 0;

}

// Sort processes by arrival time in ascending order for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

if (processes[i].arrival\_time > processes[j].arrival\_time) { struct process temp = processes[i];

processes[i] = processes[j]; processes[j] = temp;

}

}

}

// Choose preemptive or non-preemptive SJF int preemptive;

printf("Enter 1 for preemptive SJF, 0 for non-preemptive SJF: "); scanf("%d", &preemptive);

int time = 0;

int completed = 0;

while (completed < n) { int shortest\_index = -1;

int shortest\_remaining = 10000; // A large initial value

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

if (processes[i].arrival\_time <= time && processes[i].remaining\_time > 0) { if (processes[i].remaining\_time < shortest\_remaining) {

shortest\_index = i;

shortest\_remaining = processes[i].remaining\_time;

}

}

}

if (shortest\_index != -1) {

if (processes[shortest\_index].start\_time == 0) { processes[shortest\_index].start\_time = time;

}

if (preemptive) { processes[shortest\_index].remaining\_time--; time++;

if (processes[shortest\_index].remaining\_time == 0) { processes[shortest\_index].turnaround\_time = time -

processes[shortest\_index].arrival\_time;

processes[shortest\_index].waiting\_time = processes[shortest\_index].turnaround\_time - processes[shortest\_index].burst\_time;

completed++;

}

} else {

time += processes[shortest\_index].remaining\_time; processes[shortest\_index].remaining\_time = 0; processes[shortest\_index].turnaround\_time = time -

processes[shortest\_index].arrival\_time;

processes[shortest\_index].waiting\_time = processes[shortest\_index].turnaround\_time

- processes[shortest\_index].burst\_time; completed++;

}

} else {

time++;

}

}

// Calculate average waiting time and turnaround time float avg\_waiting\_time = 0.0, avg\_turnaround\_time = 0.0; for (int i = 0; i < n; i++) {

avg\_waiting\_time += processes[i].waiting\_time; avg\_turnaround\_time += processes[i].turnaround\_time;

}

avg\_waiting\_time /= n; avg\_turnaround\_time /= n;

// Print the results

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int i = 0; i < n; i++) {

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time); printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

return 0;

}

# Experiment - 7

## AIM: Write a C program to Implement priority scheduling and test with different sets of priorities.

#include <stdio.h>

#define MAX\_PROCESSES 100 struct process {

int pid;

int arrival\_time; int burst\_time; int priority;

int waiting\_time;

int turnaround\_time;

};

int main() { int n;

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

if (n > MAX\_PROCESSES) {

printf("Error: Maximum number of processes exceeded.\n"); return 1;

}

struct process processes[MAX\_PROCESSES]; for (int i = 0; i < n; i++) {

printf("Enter details for process %d:\n", i + 1); printf("Arrival Time: ");

scanf("%d", &processes[i].arrival\_time); printf("Burst Time: ");

scanf("%d", &processes[i].burst\_time); printf("Priority: ");

scanf("%d", &processes[i].priority);

processes[i].pid = i + 1; processes[i].waiting\_time = 0;

processes[i].turnaround\_time = 0;

}

// Sort processes by arrival time in ascending order for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

if (processes[i].arrival\_time > processes[j].arrival\_time) { struct process temp = processes[i];

processes[i] = processes[j]; processes[j] = temp;

}

}

}

// Calculate waiting time and turnaround time for each process int time = 0;

int completed = 0;

while (completed < n) {

int highest\_priority\_index = -1;

int highest\_priority = -1;

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

if (processes[i].arrival\_time <= time && processes[i].priority > highest\_priority) { highest\_priority\_index = i;

highest\_priority = processes[i].priority;

}

}

if (highest\_priority\_index != -1) { processes[highest\_priority\_index].waiting\_time = time -

processes[highest\_priority\_index].arrival\_time;

time += processes[highest\_priority\_index].burst\_time; processes[highest\_priority\_index].turnaround\_time = time -

processes[highest\_priority\_index].arrival\_time; completed++;

} else {

time++;

}

}

// Calculate average waiting time and turnaround time float avg\_waiting\_time = 0.0, avg\_turnaround\_time = 0.0; for (int i = 0; i < n; i++) {

avg\_waiting\_time += processes[i].waiting\_time; avg\_turnaround\_time += processes[i].turnaround\_time;

}

avg\_waiting\_time /= n; avg\_turnaround\_time /= n;

// Print the results

printf("\nProcess\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n"); for (int i = 0; i < n; i++) {

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time, processes[i].priority,

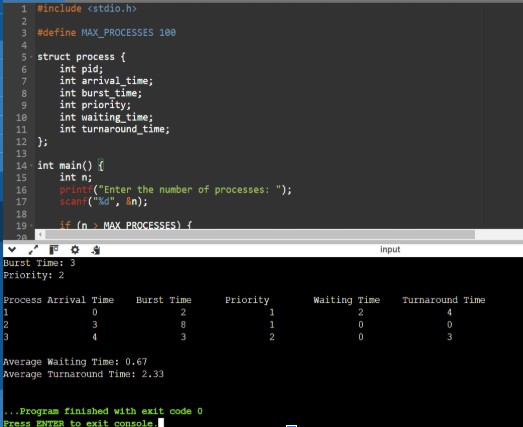
processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time); printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

return 0;

}



# Experiment - 8

## AIM: Write a C program to simulate round robin scheduling and turn around and waiting time.

#include <stdio.h>

#define MAX\_PROCESSES 100 struct process {

int pid;

int arrival\_time; int burst\_time;

int remaining\_time; int waiting\_time;

int turnaround\_time;

};

int main() {

int n, time\_quantum;

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

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

if (n > MAX\_PROCESSES) {

printf("Error: Maximum number of processes exceeded.\n"); return 1;

}

struct process processes[MAX\_PROCESSES]; for (int i = 0; i < n; i++) {

printf("Enter details for process %d:\n", i + 1); printf("Arrival Time: ");

scanf("%d", &processes[i].arrival\_time); printf("Burst Time: ");

scanf("%d", &processes[i].burst\_time);

processes[i].pid = i + 1;

processes[i].remaining\_time = processes[i].burst\_time; processes[i].waiting\_time = 0;

processes[i].turnaround\_time = 0;

}

// Sort processes by arrival time in ascending order for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

if (processes[i].arrival\_time > processes[j].arrival\_time) { struct process temp = processes[i];

processes[i] = processes[j]; processes[j] = temp;

}

}

}

int time = 0;

int completed = 0;

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

while (i < n) {

if (processes[i].arrival\_time <= time && processes[i].remaining\_time > 0) { if (processes[i].remaining\_time <= time\_quantum) {

time += processes[i].remaining\_time; processes[i].remaining\_time = 0;

processes[i].turnaround\_time = time - processes[i].arrival\_time; processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time; completed++;

} else {

time += time\_quantum; processes[i].remaining\_time -= time\_quantum;

}

}

i++;

}

}

// Calculate average waiting time and turnaround time float avg\_waiting\_time = 0.0, avg\_turnaround\_time = 0.0; for (int i = 0; i < n; i++) {

avg\_waiting\_time += processes[i].waiting\_time; avg\_turnaround\_time += processes[i].turnaround\_time;

}

avg\_waiting\_time /= n; avg\_turnaround\_time /= n;

// Print the results

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n"); for (int i = 0; i < n; i++) {

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time,

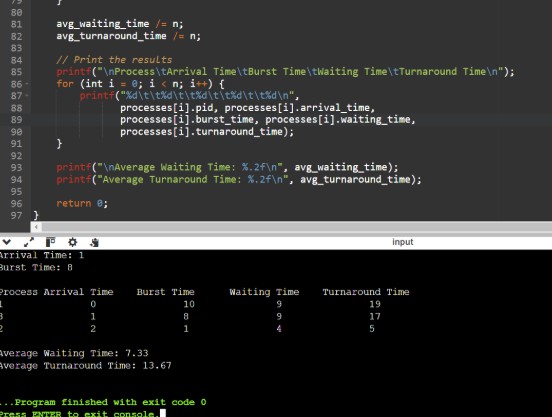
processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time); printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

return 0;

}



# Experiment - 9

## Aim: C Program to Implement Producer-Consumer Problem

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

#define BUFFER\_SIZE 5 // Size of the buffer int buffer[BUFFER\_SIZE]; // Shared buffer

int in = 0; // Index to place the next produced item

int out = 0; // Index to remove the next consumed item

// Semaphores

sem\_t empty; // Counts the empty buffer slots sem\_t full; // Counts the filled buffer slots pthread\_mutex\_t mutex; // Mutex lock for buffer

// Producer function

void\* producer(void\* arg) { int item;

while (1) {

item = rand() % 100; // Produce a random item sem\_wait(&empty); // Wait if buffer is full pthread\_mutex\_lock(&mutex); // Lock the buffer

// Add item to the buffer buffer[in] = item;

printf("Producer produced: %d\n", item); in = (in + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex); // Unlock the buffer sem\_post(&full); // Signal that the buffer has one more full slot

sleep(1); // Simulate production time

}

}

// Consumer function

void\* consumer(void\* arg) { int item;

while (1) {

sem\_wait(&full); // Wait if buffer is empty pthread\_mutex\_lock(&mutex); // Lock the buffer

// Remove item from the buffer item = buffer[out];

printf("Consumer consumed: %d\n", item); out = (out + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex); // Unlock the buffer sem\_post(&empty); // Signal that the buffer has one more empty slot

sleep(1); // Simulate consumption time

}

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// Initialize semaphores and mutex

sem\_init(&empty, 0, BUFFER\_SIZE); // Initially, buffer is empty sem\_init(&full, 0, 0); // Initially, no items in the buffer pthread\_mutex\_init(&mutex, NULL);

// Create producer and consumer threads pthread\_create(&prod\_thread, NULL, producer, NULL); pthread\_create(&cons\_thread, NULL, consumer, NULL);

// Join the threads (they will run indefinitely in this example) pthread\_join(prod\_thread, NULL); pthread\_join(cons\_thread, NULL);

// Destroy semaphores and mutex sem\_destroy(&empty); sem\_destroy(&full); pthread\_mutex\_destroy(&mutex);

return 0;

}

# Experiment - 10

## Aim: The Dining Philosophers Problem

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <semaphore.h> #include <unistd.h>

#define NUM\_PHILOSOPHERS 5

sem\_t chopsticks[NUM\_PHILOSOPHERS]; // Semaphores for chopsticks pthread\_t philosophers[NUM\_PHILOSOPHERS]; // Philosopher threads int philosopher\_ids[NUM\_PHILOSOPHERS]; // IDs for philosophers

// Function for each philosopher's behavior void\* philosopher(void\* num) {

int id = \*(int\*)num;

while (1) {

printf("Philosopher %d is thinking.\n", id); sleep(rand() % 3); // Simulate thinking

// Pick up left and right chopsticks

printf("Philosopher %d is hungry and trying to pick up chopsticks.\n", id); sem\_wait(&chopsticks[id]); // Pick up left chopstick

sem\_wait(&chopsticks[(id + 1) % NUM\_PHILOSOPHERS]); // Pick up right chopstick

// Eating

printf("Philosopher %d is eating.\n", id); sleep(rand() % 3); // Simulate eating

// Put down chopsticks

sem\_post(&chopsticks[id]); // Put down left chopstick

sem\_post(&chopsticks[(id + 1) % NUM\_PHILOSOPHERS]); // Put down right chopstick

printf("Philosopher %d has finished eating and is thinking again.\n", id); sleep(rand() % 3); // Simulate thinking before the next round

}

}

int main() { int i;

// Initialize semaphores for chopsticks (initial value 1, meaning available) for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_init(&chopsticks[i], 0, 1);

}

// Create philosopher threads

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

philosopher\_ids[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, &philosopher\_ids[i]);

}

// Join philosopher threads (this will run indefinitely) for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

}

// Destroy semaphores (not reached in this example) for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_destroy(&chopsticks[i]);

}

return 0;

}

# Experiment - 11

## Aim: The Reader-Writer problem

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <semaphore.h> #include <unistd.h>

sem\_t resource; // Semaphore to ensure mutual exclusion on the resource (file or database)

sem\_t rmutex; // Semaphore to ensure mutual exclusion on reader count int reader\_count = 0; // Number of active readers

// Reader function

void\* reader(void\* arg) { int id = \*(int\*)arg;

while (1) {

// Entry section for readers

sem\_wait(&rmutex); // Lock the reader count semaphore reader\_count++;

if (reader\_count == 1) {

sem\_wait(&resource); // First reader locks the resource (no writers allowed)

}

sem\_post(&rmutex); // Unlock the reader count semaphore

// Reading the resource

printf("Reader %d is reading the resource.\n", id); sleep(rand() % 3); // Simulate reading time

// Exit section for readers

sem\_wait(&rmutex); // Lock the reader count semaphore reader\_count--;

if (reader\_count == 0) {

sem\_post(&resource); // Last reader unlocks the resource (writers can now write)

}

sem\_post(&rmutex); // Unlock the reader count semaphore

printf("Reader %d has finished reading.\n", id);

sleep(rand() % 3); // Simulate the time between reading sessions

}

}

// Writer function

void\* writer(void\* arg) { int id = \*(int\*)arg;

while (1) {

// Writer tries to access the resource

sem\_wait(&resource); // Lock the resource (no readers or writers allowed)

// Writing to the resource

printf("Writer %d is writing to the resource.\n", id); sleep(rand() % 3); // Simulate writing time

sem\_post(&resource); // Unlock the resource (allow other writers/readers) printf("Writer %d has finished writing.\n", id);

sleep(rand() % 3); // Simulate the time between writing sessions

}

}

int main() {

pthread\_t r\_threads[5], w\_threads[5]; int ids[5];

// Initialize semaphores

sem\_init(&rmutex, 0, 1); // Reader count semaphore (mutex) sem\_init(&resource, 0, 1); // Resource semaphore (mutex)

// Create reader and writer threads for (int i = 0; i < 5; i++) {

ids[i] = i + 1;

pthread\_create(&r\_threads[i], NULL, reader, &ids[i]); pthread\_create(&w\_threads[i], NULL, writer, &ids[i]);

}

// Join reader and writer threads (this will run indefinitely) for (int i = 0; i < 5; i++) {

pthread\_join(r\_threads[i], NULL); pthread\_join(w\_threads[i], NULL);

}

// Destroy semaphores (not reached in this example) sem\_destroy(&rmutex);

sem\_destroy(&resource);

return 0;

}

# Experiment - 12

C program to implement Bankers algorithm of deadlock avoidance in operating system.

#include <stdio.h> int main() {

int n, m, i, j, k;

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

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

int allocation[n][m], max[n][m], need[n][m], available[m], work[m], finish[n]; int safeSequence[n], count = 0;

// Input Allocation Matrix printf("Enter Allocation Matrix:\n"); for (i = 0; i < n; i++)

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

// Input Max Matrix printf("Enter Max Matrix:\n"); for (i = 0; i < n; i++)

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

// Input Available Resources printf("Enter Available Resources:\n"); for (j = 0; j < m; j++)

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

// Calculate Need Matrix for (i = 0; i < n; i++)

for (j = 0; j < m; j++)

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

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

finish[i] = 0; // Initially all processes are unfinished

// Work = Available for (j = 0; j < m; j++)

work[j] = available[j]; printf("\nSafe Sequence is: ");

while (count < n) { int found = 0;

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

if (!finish[i]) { // Process not finished int canAllocate = 1;

for (j = 0; j < m; j++) {

if (need[i][j] > work[j]) { canAllocate = 0; break;

}

}

if (canAllocate) {

// Allocate resources for (k = 0; k < m; k++)

work[k] += allocation[i][k]; safeSequence[count++] = i; finish[i] = 1;

found = 1;

}

}

}

if (!found) {

printf("System is not in a safe state.\n"); return 0;

}

}

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

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

printf("\nSystem is in a safe state.\n"); return 0;

}