**OPERATING SYSTEM LAB FILE**

Submitted by

**Karan Aswal**

R2142220533

**B-2**

B.Tech Computer Science and Engineering

Submitted to

**Alok Jhaldiyal**

Assistant Professor (SS)

Dept. of Systemics

SoCS, UPES

Department of Systemics

School of Computer Science

University of Petroleum and Energy Studies

cid:image003.png@01D81F60.F8BDA810

|  |  |  |
| --- | --- | --- |
| S.no | Experiment | Date |
| 1. | System calls & I/O System calls | 8 sep 2023 |
| 2. | CPU Scheduling | 15 sep 2023 |
| 3. | Inter-process communication | 22 sep 2023 |
| 4. | Semaphore | 29 sep 2023 |
| 5. | Memory management -1 | 6 oct 2023 |
| 6. | Memory management -2 | 13 oct 2023 |
| 7. | File manipulation | 27 oct 2023 |
| 8. | Fork execution | 3 nov 2023 |
| 9. | Deadlock avoidance | 17 nov 2023 |

**Experiment 1**

**Experiment No 1: System calls & I/O System calls**

i) To write programs to perform following operations in UNIX:

a) Process Creation

b) Executing a command

c) Sleep command

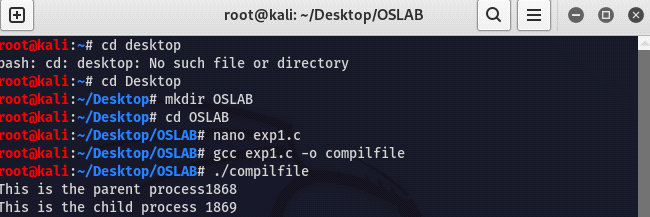
d) Sleep command using get pid

e) Signal handling using kill

f) Wait command

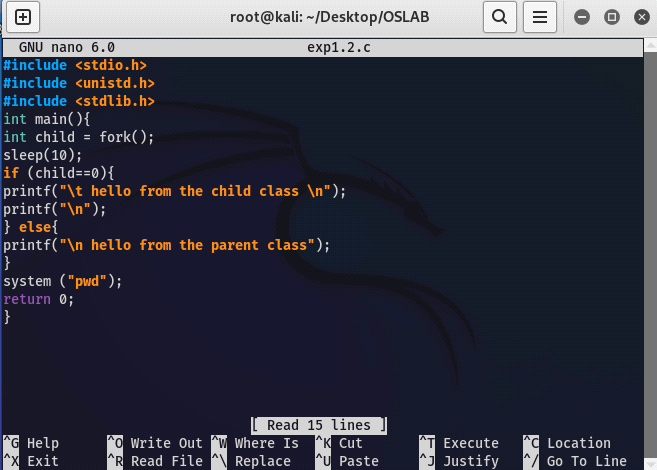
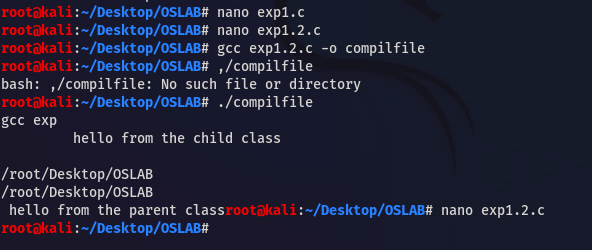
***I) To write programs to perform the following operations in UNIX***

A) Problem - 1.1: Process Creation

**Source Code:** **output -**

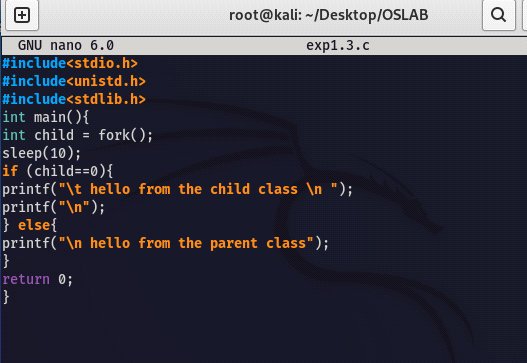
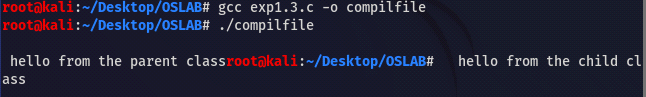
A) Problem - 1.2: Executing a command

**Source Code:**

  
  
output -   
  


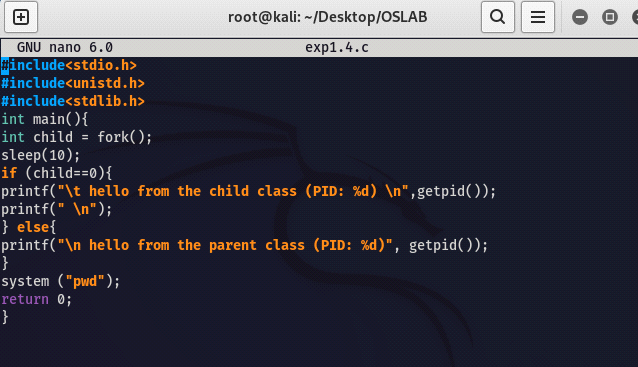
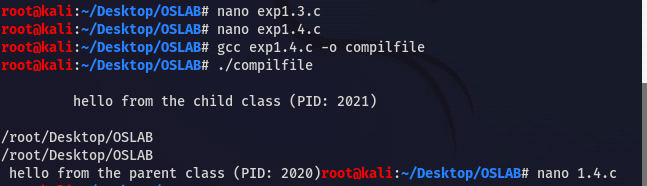
A) Problem - 1.3: Sleep Command

**Source Code:**

  
  
output -   
  


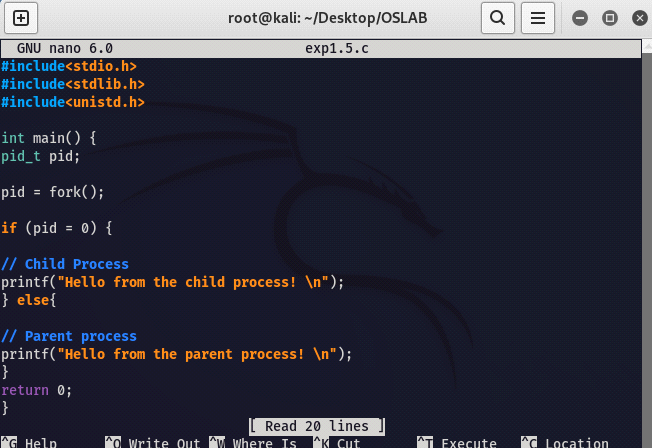
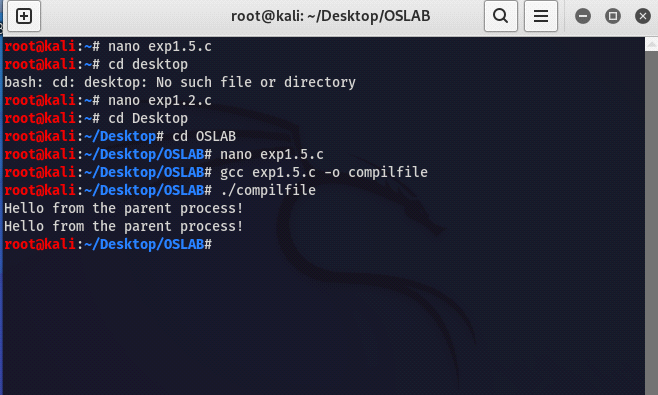
A) Problem - 1.4: Sleep command using get pid

**Source Code:**

  
  
output -   
  


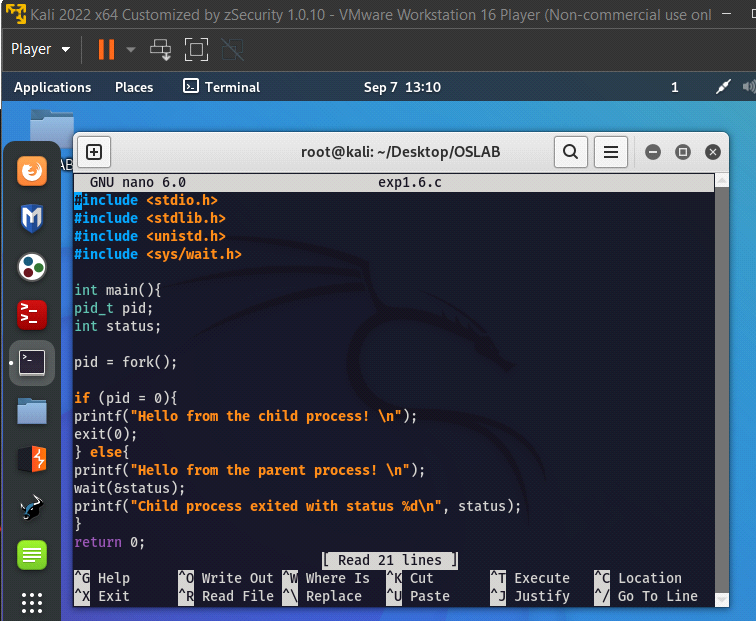
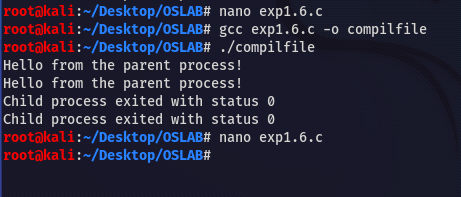
A) Problem - 1.5: Signal Handling Using Kill

**Source Code:**

  
  
Output -   
  


A) Problem - 1.6: Wait Command

**Source Code:**

  
  
Output -   
  


Experiment 2:

CPU Scheduling

**Aim -**

1. To write a C program to implement the CPU scheduling algorithm for FIRST COME FIRST SERVE.
2. To write a C program to implement the CPU scheduling algorithm for Shortest Job First
3. To write a C program to implement the CPU scheduling algorithm for Round Robin
4. To write a C program to implement the CPU scheduling algorithm for Priority Scheduling.

**1.To write a C program to implement the CPU scheduling algorithm for FIRST COME FIRST SERVE.**

**Theory - First-Come-First-Serve (FCFS) is one of the simplest CPU scheduling algorithms. In FCFS, processes are executed in the order they arrive in the ready queue. The process that arrives first is scheduled first, and the process that arrives later is scheduled later. FCFS is non-preemptive, meaning once a process starts executing, it continues until it finishes or enters a waiting state.**

**Code-**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrivalTime; // Arrival Time

int burstTime; // Burst Time

int waitingTime; // Waiting Time

int turnaroundTime; // Turnaround Time

};

void calculateTimes(struct Process processes[], int n)

processes[0].waitingTime = 0;

processes[0].turnaroundTime = processes[0].burstTime;

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

processes[i].waitingTime = processes[i - 1].turnaroundTime; processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

}

}

int main() {

int n;

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

scanf("%d", &n)

struct Process processes[n];

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

processes[i].id = i + 1;

printf("Enter arrival time for Process %d: ", i + 1);

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

printf("Enter burst time for Process %d: ", i + 1);

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

}

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].arrivalTime > processes[j + 1].arrivalTime) {

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

calculateTimes(processes, n);

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime, processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);

}

return 0;

}

**2.** **To write a C program to implement the CPU scheduling algorithm for Shortest Job First.**

**Theory-** **Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time from the ready queue for execution. The idea behind SJF scheduling is to minimize the average waiting time of processes. It can be implemented in two ways:**

**Preemptive SJF: In this version, if a new process with a shorter burst time arrives and is ready to run, it preempts the currently running process and starts executing.**

**Non-Preemptive SJF: In this version, the currently running process will continue to execute until it completes, even if a process with a shorter burst time arrives in the meantime. The new process will be scheduled once the CPU becomes idle.**

**Code-**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrivalTime; // Arrival Time

int burstTime; // Burst Time

int waitingTime; // Waiting Time

int turnaroundTime; // Turnaround Time

};

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

int remainingTime[n];

int complete = 0, currentTime = 0, shortest = 0, minBurst = INT\_MAX;

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

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

}

while (complete < n) {

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

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

minBurst = remainingTime[i];

shortest = i;

}

}

remainingTime[shortest] = 0;

processes[shortest].waitingTime = currentTime - processes[shortest].arrivalTime;

processes[shortest].turnaroundTime = processes[shortest].waitingTime + processes[shortest].burstTime;

currentTime += processes[shortest].burstTime;

minBurst = INT\_MAX;

complete++;

}

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

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

processes[i].id = i + 1;

printf("Enter arrival time for Process %d: ", i + 1);

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

printf("Enter burst time for Process %d: ", i + 1);

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

}

calculateTimes(processes, n);

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime, processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);

}

return 0;

}

**3.** **To write a C program to implement the CPU scheduling algorithm for Round Robin.**

Theory- Round Robin (RR) is a CPU scheduling algorithm that assigns a fixed time quantum to each process in a queue. When a process's time quantum expires, it's moved to the back of the queue, and the next process in the queue is allowed to execute. This continues until all processes have completed execution. Round Robin is a preemptive scheduling algorithm that ensures fairness by giving each process an equal share of CPU time. It's often used in time-sharing systems.

**Code-** **#include <stdio.h>**

**#include <stdbool.h>**

**struct Process {**

**int id; // Process ID**

**int burstTime; // Burst Time**

**int remainingTime; // Remaining Time**

**int arrivalTime; // Arrival Time**

**int waitingTime; // Waiting Time**

**int turnaroundTime; // Turnaround Time**

**};**

**void roundRobin(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) {**

**int executeTime = (processes[i].remainingTime < timeQuantum) ? processes[i].remainingTime : timeQuantum;**

**processes[i].remainingTime -= executeTime;**

**currentTime += executeTime;**

**if (processes[i].remainingTime == 0) {**

**completed++;**

**processes[i].turnaroundTime = currentTime - processes[i].arrivalTime;**

**processes[i].waitingTime = processes[i].turnaroundTime - processes[i].burstTime;**

**}**

**}**

**}**

**}**

**}**

**int main() {**

**int n, timeQuantum;**

**printf("Enter the number of processes: ");**

**scanf("%d", &n);**

**printf("Enter the time quantum: ");**

**scanf("%d", &timeQuantum);**

**struct Process processes[n];**

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

**processes[i].id = i + 1;**

**printf("Enter arrival time for Process %d: ", i + 1);**

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

**printf("Enter burst time for Process %d: ", i + 1);**

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

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

**}**

**roundRobin(processes, n, timeQuantum);**

**printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");**

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

**printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime, processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);**

**}**

**return 0;**

**}**

**4.** **To write a C program to implement the CPU scheduling algorithm for Priority Scheduling.**

**Theory-**

Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority, and the CPU scheduler selects the process with the highest priority for execution. Processes with higher priority values are executed before those with lower priority values. Priority scheduling can be both preemptive and non-preemptive. In preemptive priority scheduling, a running process can be interrupted if a higher-priority process becomes available. In non-preemptive priority scheduling, the currently running process continues until it completes.

**Code-**

**#include <stdio.h>**

**struct Process {**

**int id; // Process ID**

**int priority; // Priority (lower value indicates higher priority)**

**int burstTime; // Burst Time**

**int waitingTime; // Waiting Time**

**int turnaroundTime; // Turnaround Time**

**};**

**void priorityScheduling(struct Process processes[], int n) {**

**int currentTime = 0;**

**for (int i = 0; i < n - 1; i++) {**

**for (int j = 0; j < n - i - 1; j++) {**

**if (processes[j].priority > processes[j + 1].priority) {**

**struct Process temp = processes[j];**

**processes[j] = processes[j + 1];**

**processes[j + 1] = temp;**

**}**

**}**

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

**processes[i].waitingTime = currentTime;**

**processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;**

**currentTime += processes[i].burstTime;**

**}**

**}**

**int main() {**

**int n;**

**printf("Enter the number of processes: ");**

**scanf("%d", &n);**

**struct Process processes[n];**

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

**processes[i].id = i + 1;**

**printf("Enter priority for Process %d: ", i + 1);**

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

**printf("Enter burst time for Process %d: ", i + 1);**

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

**}**

**priorityScheduling(processes, n);**

**printf("\nProcess\tPriority\tBurst Time\tWaiting Time\tTurnaround Time\n");**

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

**printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].priority, processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);**

**}**

**return 0;**

**}**

EXPERIMENT 3:

INTER PROCESS COMMUNICATION

Aim:- The aim of this program is to demonstrate interprocess

communication (IPC) using pipes in a C program. Specifically, it creates

a parent process and a child process. The parent process writes a

message to a pipe, and the child process reads the message from the pipe

and prints it to the screen.

Theory:- This program showcases interprocess communication

(IPC) using pipes. It creates a parent and child process, with the parent

writing a message to the pipe, and the child reading and displaying the

message. Error handling ensures proper execution.

Pipe:- A unidirectional communication channel that enables data

transmission between parent and child processes in a multi-process

environment.

Child Process:- A process created by forking an existing process,

typically to perform a specific task or function within a larger program.

Parent Process:- The original process that creates and manages child

processes, often coordinating their execution and communication.

Program:-

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

int pipe\_fd[2];

pid\_t pid;

char message[] = "Hello, child process!\n";

char read\_buffer[100];

pipe(pipe\_fd);

pid = fork();

if (pid == 0) {

close(pipe\_fd[1]);

read(pipe\_fd[0], read\_buffer, sizeof(read\_buffer));

printf("Child Process: Received data from parent: %s",

read\_buffer);

close(pipe\_fd[0]);

} else {

close(pipe\_fd[0]);

write(pipe\_fd[1], message, sizeof(message));

close(pipe\_fd[1]);

}

return 0;

}

Input/Output:



Experiment 4:

**Experiment 4: Semaphore**

Aim:-

1.) Write a program that demonstrates how two processes can share a variable using semaphore

2.) To write a C program to implement the Producer & consumer Problem (Semaphore)

1.) **Program** :-

#include<pthread.h>

#include<stdio.h>

#include<semaphore.h>

#include<unistd.h>

void \*fun1();

void \*fun2();

int shared=1;

sem\_t s;

int main()

{

sem\_init(&s,0,1);

pthread\_t thread1,thread2;

pthread\_create(&thread1,NULL,fun1,NULL);

pthread\_create(&thread2,NULL,fun2,NULL);

pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

printf("Final value of shared is %d\n",shared);

}

void \*fun1()

{

int x;

sem\_wait(&s);

x=shared;

printf("thread1 reads the value as %d\n",x);

x++;

printf("local updation by thread1: %d\n",x);

sleep(1);

shared=x;

printf("value fo shared variable updated by thread1 is: %d\n",shared);

sem\_post(&s);

}

void \*fun2()

{

int y;

sem\_wait(&s);

y=shared;

printf("thread2 reads value as %d\n",y);

y--;

printf("local updation by thread2: %d \n",y);

sleep(1);

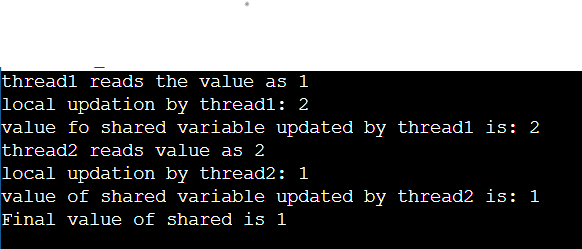
shared=y;

printf("value of shared variable updated by thread2 is: %d\n",shared);

sem\_post(&s);

}

**Result :**



2.)

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <unistd.h>

#include <time.h>

#include <semaphore.h>

#define THREAD\_NUM 8

sem\_t semEmpty;

sem\_t semFull;

pthread\_mutex\_t mutexBuffer;

int buffer[10];

int count = 0;

void\* producer(void\* args) {

while (1) {

int x = rand() % 100;

sleep(1);

sem\_wait(&semEmpty);

pthread\_mutex\_lock(&mutexBuffer);

buffer[count] = x;

count++;

pthread\_mutex\_unlock(&mutexBuffer);

sem\_post(&semFull);

}

}

void\* consumer(void\* args) {

while (1) {

int y;

sem\_wait(&semFull);

pthread\_mutex\_lock(&mutexBuffer);

y = buffer[count - 1];

count--;

pthread\_mutex\_unlock(&mutexBuffer);

sem\_post(&semEmpty);

printf("Got %d\n", y);

sleep(1);

}

}

int main(int argc, char\* argv[]) {

srand(time(NULL));

pthread\_t th[THREAD\_NUM];

pthread\_mutex\_init(&mutexBuffer, NULL);

sem\_init(&semEmpty, 0, 10);

sem\_init(&semFull, 0, 0);

int i;

for (i = 0; i < THREAD\_NUM; i++) {

if (i > 0) {

if (pthread\_create(&th[i], NULL, &producer, NULL) != 0) {

perror("Failed to create thread");

}

} else {

if (pthread\_create(&th[i], NULL, &consumer, NULL) != 0) {

perror("Failed to create thread");

}

}

}

for (i = 0; i < THREAD\_NUM; i++) {

if (pthread\_join(th[i], NULL) != 0) {

perror("Failed to join thread");

}

}

sem\_destroy(&semEmpty);

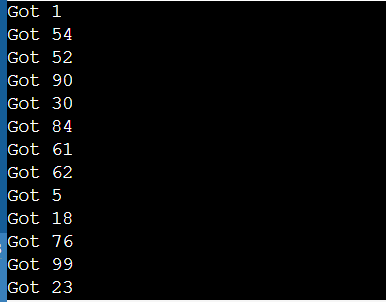
sem\_destroy(&semFull);

pthread\_mutex\_destroy(&mutexBuffer);

return 0;

}

Result :



Experiment no 9 & 10: Deadlock avoidance

1.To implement Banker's algorithm for a multiple resources.

Code:

#include <stdio.h>

int main()

{

int n, m, i, j, k;

n = 5;

m = 3; int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;

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

f[k] = 0;

}

int need[n][m];

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

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

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

}

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

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

if (need[i][j] > avail[j]){

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

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

avail[y] += alloc[i][y];

f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if(f[i]==0)

{

flag=0;

printf("The following system is not safe");

break;

}

}

if(flag==1)

{

printf("Following is the SAFE Sequence\n");

for (i = 0; i < n - 1; i++)

printf(" P%d ->", ans[i]);

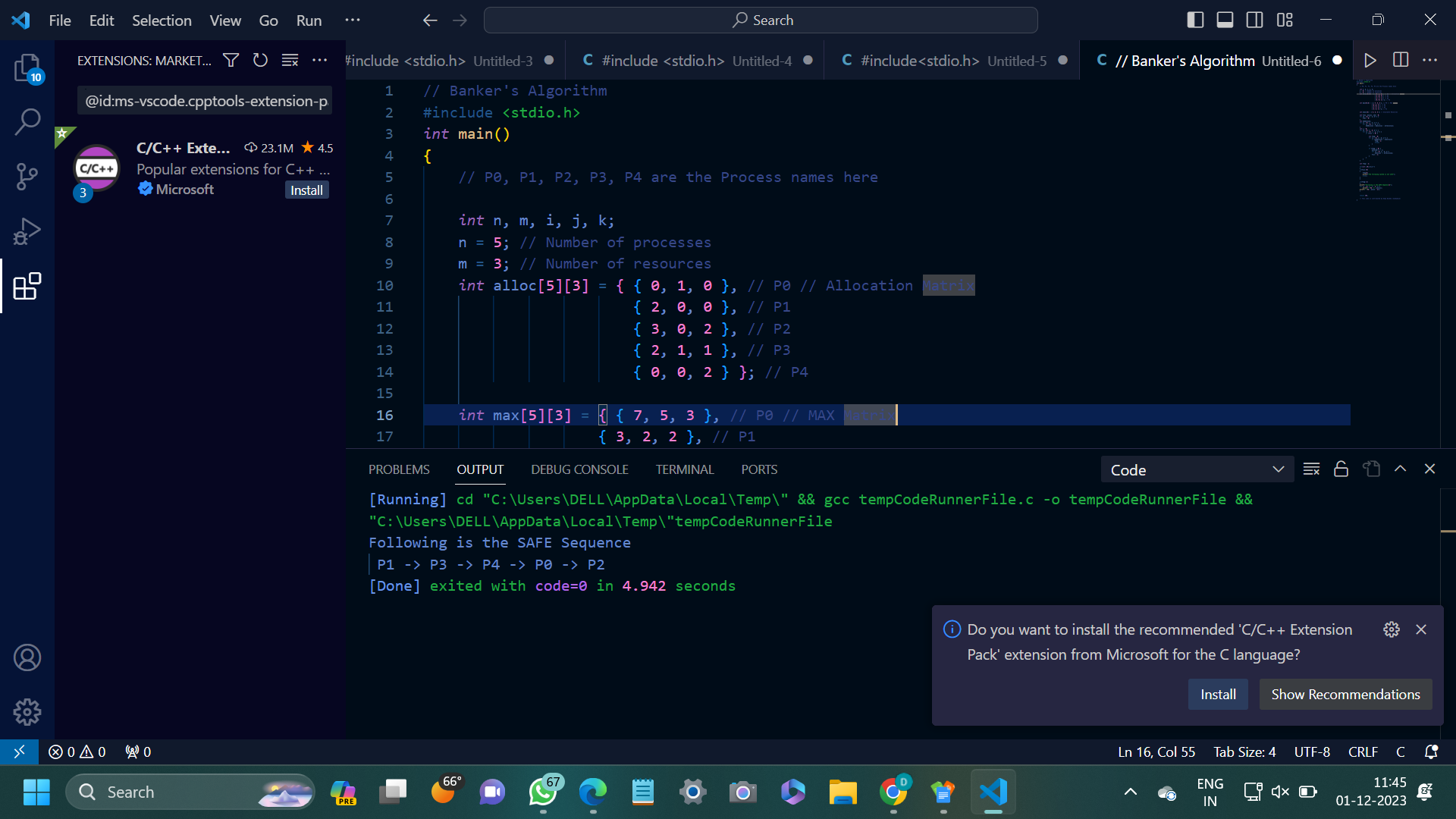
printf(" P%d", ans[n - 1]);

}

return (0);

// This code is contributed by Deep Baldha (CandyZack)

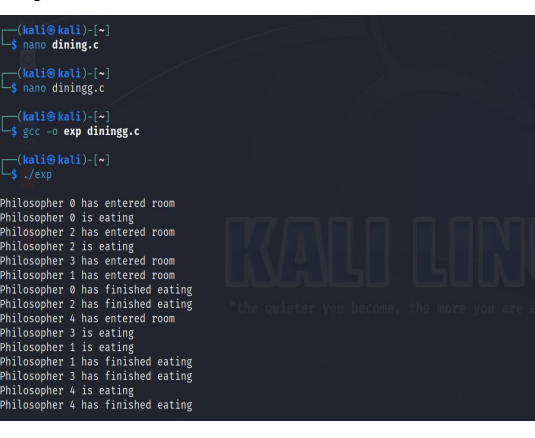
}



2.To implement dinning philosopher’s problem.

Code:

#include #include #include #include #include sem\_t room; sem\_t chopstick[5]; void \* philosopher(void \*); void eat(int); int main() { int i,a[5]; pthread\_t tid[5]; sem\_init(&room,0,4); for(i=0;i<5;i++) sem\_init(&chopstick[i],0,1); for(i=0;i<5;i++){ a[i]=i; pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]); } for(i=0;i<5;i++) pthread\_join(tid[i],NULL); } void \* philosopher(void \* num) { int phil=\*(int \*)num; sem\_wait(&room); printf("\nPhilosopher %d has entered room",phil); sem\_wait(&chopstick[phil]); sem\_wait(&chopstick[(phil+1)%5]); eat(phil); sleep(2); printf("\nPhilosopher %d has finished eating",phil); sem\_post(&chopstick[(phil+1)%5]); sem\_post(&chopstick[phil]); sem\_post(&room); } void eat(int phil) { printf("\nPhilosopher %d is eating",phil); }



EXPERIMENT:7 FILE MANIPULATION

Aim:

i. Displays the file and Directory

ii. Creating new Directory

(i) Theory:

mkdir command

The mkdir command creates directories. This command can create multiple directories at

once as well as set the permissions for the directories. It is important to note that the user

executing this command must have enough permissions to create a directory in the parent

directory.

Program:

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

int main()

{

char dir[15];

int c=0;

printf("Enter the directory name: ");

scanf("%s",dir);

c = mkdir(dir,777);

if(c==0)

printf("Directory created successfully");

else

printf("Unable to create directory");

return 0;

}

Output:



(ii) Theory

ls command

ls is a Linux shell command that lists directory contents of files and directories. It provides

valuable information about files, directories, and their attributes.

Program

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main()

{

char command[50];

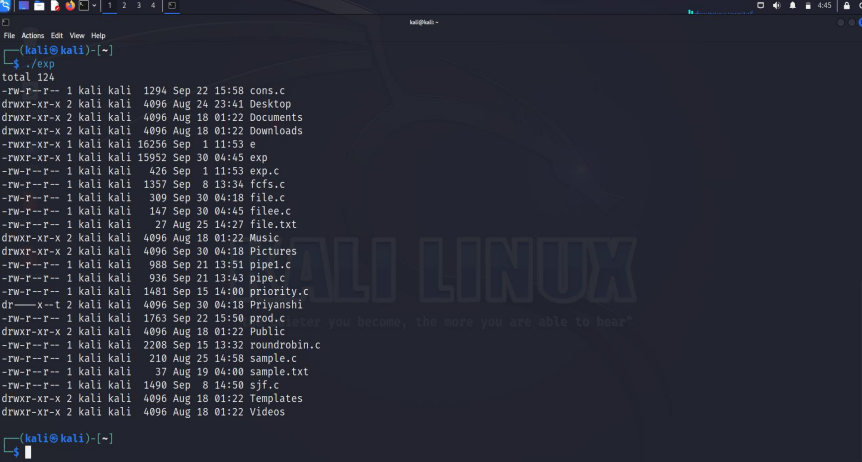
strcpy(command, "ls -l");

system(command);

return 0;

}

Output:



EXPERIMENT: 8 FORK EXECUTION

Aim:

i. Simple fork execution

ii. Execute the fork command two times

iii. Execute the fork command three times

Theory:

fork command()

The Fork system call, i.e., fork() is used for creating a new process in Linux, and Unix

systems, which is called the child process, which runs concurrently with the process that

makes the fork() call (parent process). After a new child process is created, both processes

will execute the next instruction following the fork() system call.

(i) Simple Fork Execution

Program:

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// make two process which run same

// program after this instruction

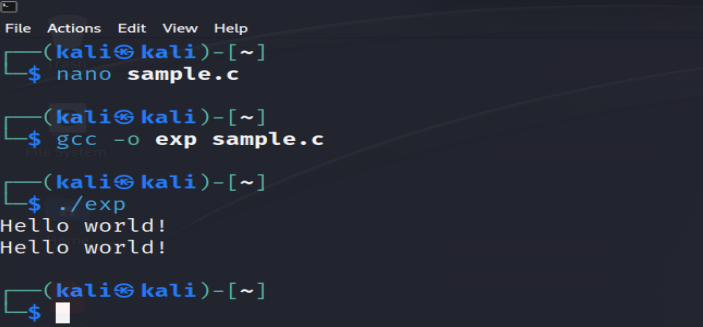
fork();

printf("Hello world!\n");

return 0;

}

Output:



(ii) Execute the fork command two times

Program:

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// make two process which run same

// program after this instruction

fork();

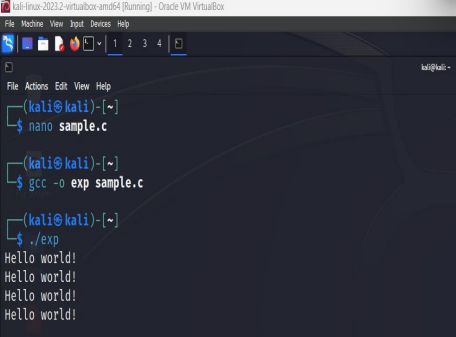
fork();

printf("Hello world!\n");

return 0;

}

Output:



(iii) Execute the fork command three times

Program:

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// make two process which run same

// program after this instruction

fork();

fork();

fork();

printf("Hello world!\n");

return 0;

}

Output:



EXPERIMENT: 9 & 10 DEADLOCK

AVOIDANCE

Aim:

i. To implement Banker's algorithm for a multiple resources.

ii. To implement dinning philosopher’s problem.

(i) Theory:

Banker’s algorithm

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests

for safety by simulating the allocation for the predetermined maximum possible amounts of

all resources, then makes an “s-state” check to test for possible activities, before deciding

whether allocation should be allowed to continue.

Program:

// Banker's Algorithm

#include <stdio.h>

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;

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

f[k] = 0;

}

int need[n][m];

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

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

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

}

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

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

if (need[i][j] > avail[j]){

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

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

avail[y] += alloc[i][y];

f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if(f[i]==0)

{

flag=0;

printf("The following system is not safe");

break;

}

}

if(flag==1)

{

printf("Following is the SAFE Sequence\n");

for (i = 0; i < n - 1; i++)

printf(" P%d ->", ans[i]);

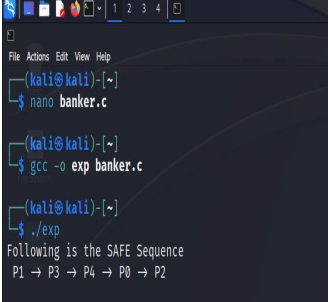
printf(" P%d", ans[n - 1]);

}

return (0);

}

Output:



(ii) Dining Philosopher Problem

Theory:

The Dining Philosopher Problem states that K philosophers are seated around a circular table

with one chopstick between each pair of philosophers. There is one chopstick between each

philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One

chopstick may be picked up by any one of its adjacent followers but not both.

Program:

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

sem\_t room;

sem\_t chopstick[5];

void \* philosopher(void \*);

void eat(int);

int main()

{

int i,a[5];

pthread\_t tid[5];

sem\_init(&room,0,4);

for(i=0;i<5;i++)

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

for(i=0;i<5;i++){

a[i]=i;

pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);

}

for(i=0;i<5;i++)

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

}

void \* philosopher(void \* num)

{

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

sem\_wait(&room);

printf("\nPhilosopher %d has entered room",phil);

sem\_wait(&chopstick[phil]);

sem\_wait(&chopstick[(phil+1)%5]);

eat(phil);

sleep(2);

printf("\nPhilosopher %d has finished eating",phil);

sem\_post(&chopstick[(phil+1)%5]);

sem\_post(&chopstick[phil]);

sem\_post(&room);

}

void eat(int phil)

{

printf("\nPhilosopher %d is eating",phil);

}

Output:

