**OPERATING SYSTEM LAB FILE**

Submitted by

**Akshat Negi**

**R2142220414**

Batch - 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

**Index**

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Experiment Title** | **Faculty Signature** |
| 1 | **System calls & I/O System calls** |  |
| 2 | **CPU Scheduling** |  |
| 3 | **Inter-process Communication** |  |
| 4 | **Semaphore** |  |
| 5 | **Memory management-1** |  |
| 6 | **Memory management-2** |  |
| 7 | **FILE MANIPULATION** |  |
| 8 | **Fork Execution** |  |
| 9 & 10 | **Deadlock avoidance** |  |

**EXPERIMENT NO-1**

**System calls & I/O System calls**

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

a) Process Creation:

In UNIX, a new process can be created using the fork() system call. This creates a new process that is a copy of the calling process, and both processes continue executing from the point of the fork() call.



b) Executing a command:

To execute a command in UNIX, the system call exec() can be used. This replaces the current process image with a new process image specified by the command.



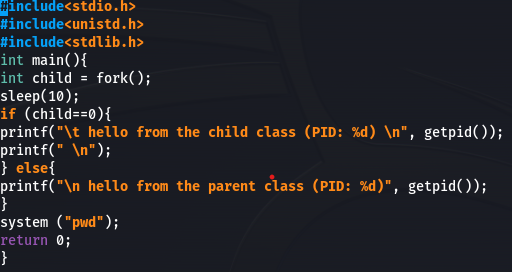
c) Sleep command:

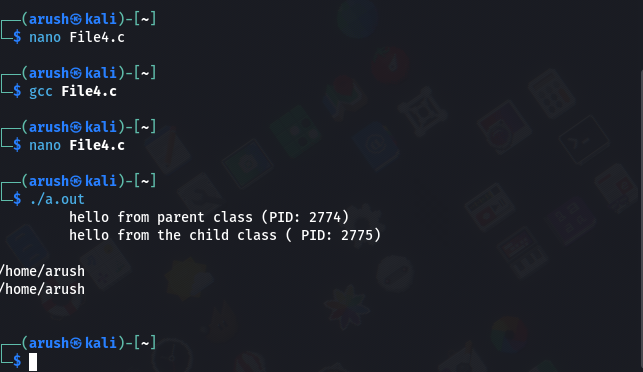
The sleep command in UNIX is used to pause the execution of a script or a command for a specified amount of time. The syntax for the command is "sleep n", where n is the number of seconds to sleep.



d) Sleep command using get pid:

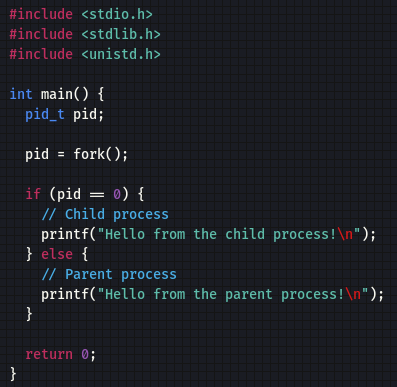
To use the sleep command with the process ID (PID) of a specific process, the command "sleep $(ps -o etime= -p <PID>)" can be used. This retrieves the elapsed time of the specified process and sleeps for that amount of time.





e) Signal handling using kill:

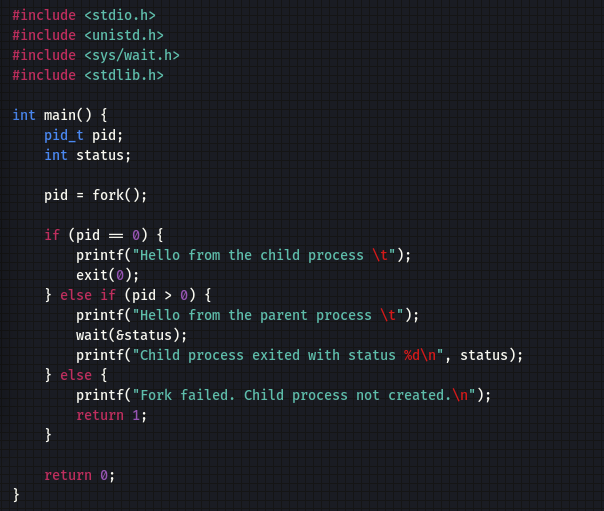
In UNIX, the kill command is used to send a signal to a process. This can be used for signal handling, such as terminating a process or restarting it. The syntax for the command is "kill -<signal> <PID>", where <signal> is the signal to send and <PID> is the process ID of the target process.

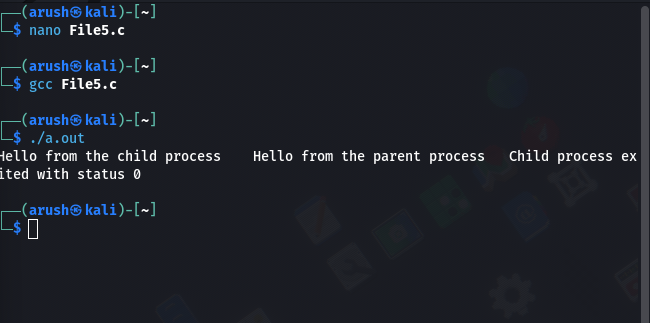




f) Wait command:

The wait command in UNIX is used to wait for the completion of a child process. This is useful when a parent process needs to wait for a child process to finish before continuing execution. The syntax for the command is "wait <PID>", where <PID> is the process ID of the child process to wait for.

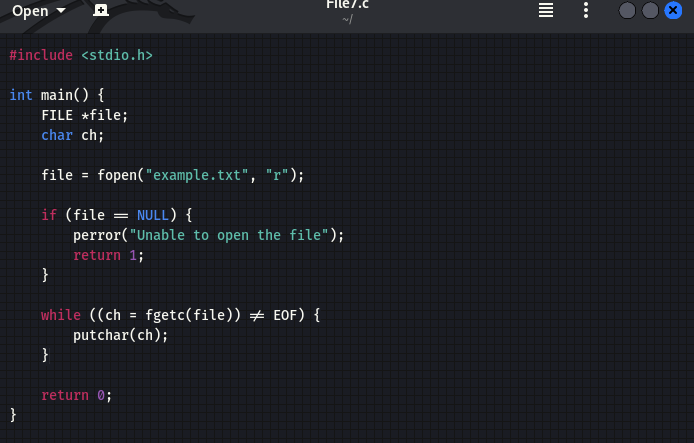


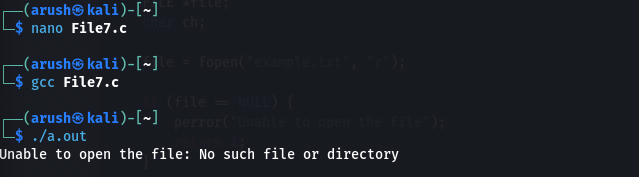


**ii) To write programs to perform following operations in UNIX:**

a) Reading from a file:

In UNIX, the cat command can be used to read the contents of a file. The syntax for the command is "cat <filename>", where <filename> is the name of the file to be read.





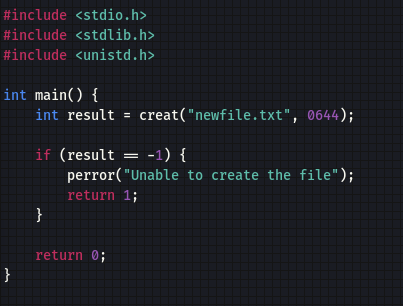
(b) Writing into a file:

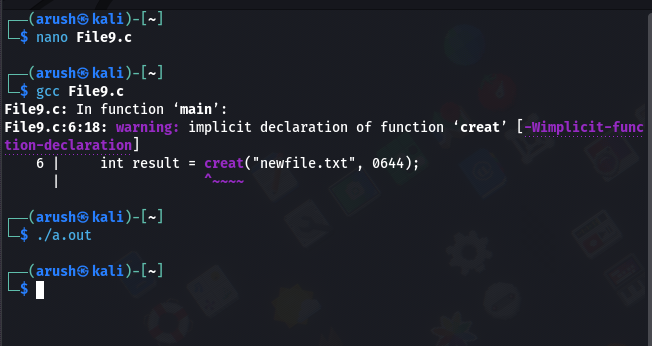
In UNIX, the echo command can be used to write text into a file. The syntax for the command is "echo <text> > <filename>", where <text> is the text to be written and <filename> is the name of the file to write to.



(c) File Creation:

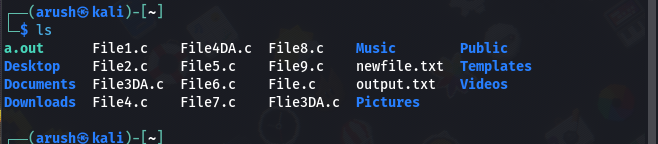
In UNIX, the touch command can be used to create a new file. The syntax for the command is "touch <filename>", where <filename> is the name of the file to be created.





(d) Implementation of ls command:

The ls command in UNIX is used to list the files and directories in a directory. The syntax for the command is "ls <directory>", where <directory> is the name of the directory to list. The command can be further customized with options such as -l to display the files in a long format and -a to display hidden files.

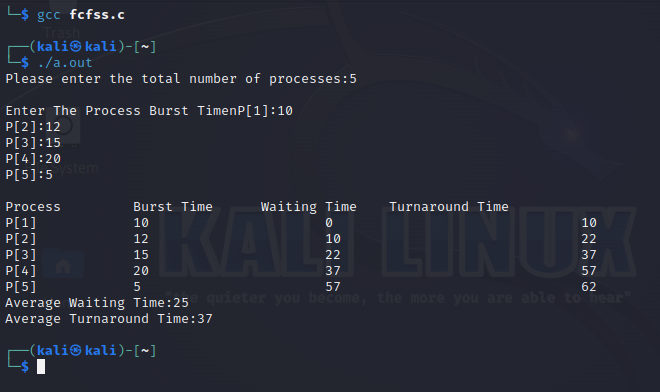


**EXPERIMENT NO-2**

**CPU SCHEDULING**

**1) First come First serve**

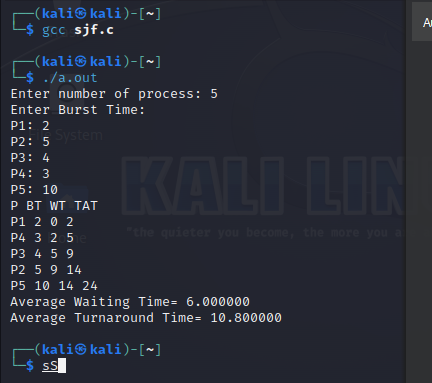
#include<stdio.h>  
int main()  
{  
 int n,bt[30],wait\_t[30],turn\_ar\_t[30],av\_wt\_t=0,avturn\_ar\_t=0,i,j;  
 printf("Please enter the total number of processes:");   
 scanf("%d",&n);  
 printf("\nEnter The Process Burst Timen");  
 for(i=0;i<n;i++) // burst time for every process will be taken as input  
 {  
 printf("P[%d]:",i+1);  
 scanf("%d",&bt[i]);  
 }  
 wait\_t[0]=0;   
 for(i=1;i<n;i++)  
 {  
 wait\_t[i]=0;  
 for(j=0;j<i;j++)  
 wait\_t[i]+=bt[j];  
 }  
 printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");  
 for(i=0;i<n;i++)  
 {  
 turn\_ar\_t[i]=bt[i]+wait\_t[i];  
 av\_wt\_t+=wait\_t[i];  
 avturn\_ar\_t+=turn\_ar\_t[i];  
 printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,bt[i],wait\_t[i],turn\_ar\_t[i]);  
 }  
 av\_wt\_t/=i;  
 avturn\_ar\_t/=i; // average calculation is done here  
 printf("\nAverage Waiting Time:%d",av\_wt\_t);  
 printf("\nAverage Turnaround Time:%d",avturn\_ar\_t);  
 return 0;  
}

**OUTPUT**

**2 ) Shortest Job First (SJF)**

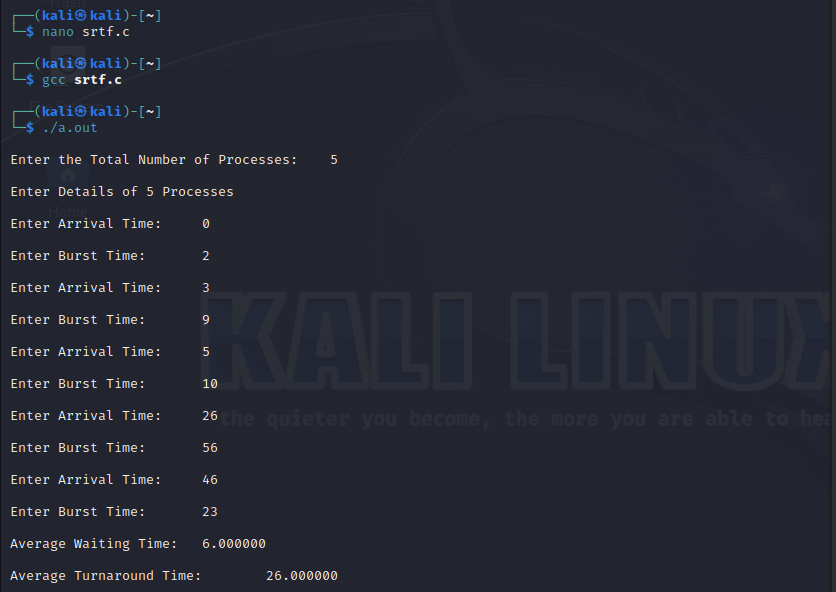
#include <stdio.h>  
int main()  
{  
int A[100][4];  
int i, j, n, total = 0, index, temp;  
float avg\_wt, avg\_tat;  
printf("Enter number of process: ");  
scanf("%d", &n);  
printf("Enter Burst Time:\n");  
for (i = 0; i < n; i++) {  
printf("P%d: ", i + 1);  
scanf("%d", &A[i][1]);  
A[i][0] = i + 1;  
}  
for (i = 0; i < n; i++) {  
index = i;  
for (j = i + 1; j < n; j++)  
if (A[j][1] < A[index][1])  
index = j;  
temp = A[i][1];  
A[i][1] = A[index][1];  
A[index][1] = temp;  
temp = A[i][0];  
A[i][0] = A[index][0];  
A[index][0] = temp;  
}  
A[0][2] = 0;  
for (i = 1; i < n; i++) {  
A[i][2] = 0;  
for (j = 0; j < i; j++)  
A[i][2] += A[j][1];  
total += A[i][2];  
}  
avg\_wt = (float)total / n;  
total = 0;  
printf("P BT WT TAT\n");  
for (i = 0; i < n; i++) {  
A[i][3] = A[i][1] + A[i][2];  
total += A[i][3];  
printf("P%d %d %d %d\n", A[i][0],  
A[i][1], A[i][2], A[i][3]);  
}  
avg\_tat = (float)total / n;  
printf("Average Waiting Time= %f", avg\_wt);  
printf("\nAverage Turnaround Time= %f", avg\_tat);  
}

**OUTPUT**



**3) Shortest Remaining Time First(SRTF):**

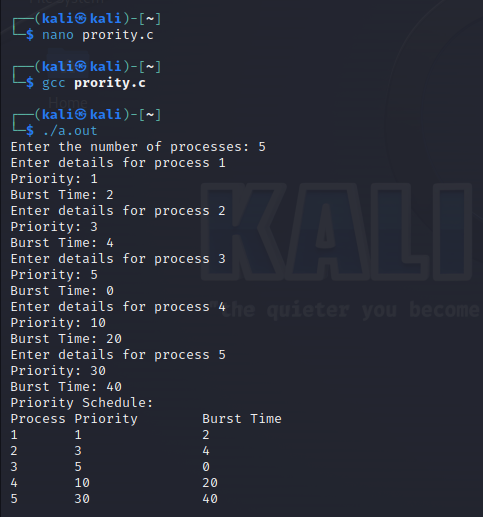
#include <stdio.h>  
int main()   
{  
int arrival\_time[10], burst\_time[10], temp[10];  
int i, smallest, count = 0, time, limit;  
double wait\_time = 0, turnaround\_time = 0, end;  
float average\_waiting\_time, average\_turnaround\_time;  
printf("\nEnter the Total Number of Processes:\t");  
scanf("%d", &limit);   
printf("\nEnter Details of %d Processes\n", limit);  
for(i = 0; i < limit; i++)  
{  
printf("\nEnter Arrival Time:\t");  
scanf("%d", &arrival\_time[i]);  
printf("\nEnter Burst Time:\t");  
scanf("%d", &burst\_time[i]);   
temp[i] = burst\_time[i];  
}  
burst\_time[9] = 9999;   
for(time = 0; count != limit; time++)  
{  
smallest = 9;  
for(i = 0; i < limit; i++)  
{  
if(arrival\_time[i] <= time && burst\_time[i] < burst\_time[smallest] && burst\_time[i] >   
0)  
{  
smallest = i;  
}  
}  
burst\_time[smallest]--;  
if(burst\_time[smallest] == 0)  
{  
count++;  
end = time + 1;  
wait\_time = wait\_time + end - arrival\_time[smallest] - temp[smallest];  
turnaround\_time = turnaround\_time + end - arrival\_time[smallest];  
}  
}  
average\_waiting\_time = wait\_time / limit;   
average\_turnaround\_time = turnaround\_time / limit;  
printf("\nAverage Waiting Time:\t%lf\n", average\_waiting\_time);  
printf("\nAverage Turnaround Time:\t%lf\n", average\_turnaround\_time);  
return 0;  
}

**OUTPUT**

**4) Priority Scheduling:**

#include <stdio.h>  
#define MAX\_PROCESSES 10  
struct Process {  
 int id;  
 int priority;  
 int burst\_time;  
};  
void priorityScheduling(struct Process processes[], int n) {  
 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;  
}}}}  
void displaySchedule(struct Process processes[], int n) {  
 printf("Process\tPriority\tBurst Time\n");  
 for (int i = 0; i < n; i++) {  
 printf("%d\t%d\t\t%d\n", processes[i].id, processes[i].priority,   
processes[i].burst\_time);  
 }  
}  
int main() {  
 int n;  
 printf("Enter the number of processes: ");  
 scanf("%d", &n);  
 struct Process processes[MAX\_PROCESSES];  
 for (int i = 0; i < n; i++) {  
 printf("Enter details for process %d\n", i + 1);  
 processes[i].id = i + 1;  
 printf("Priority: ");  
 scanf("%d", &processes[i].priority);  
 printf("Burst Time: ");  
 scanf("%d", &processes[i].burst\_time);  
 }  
 priorityScheduling(processes, n);  
 printf("Priority Schedule:\n");  
 displaySchedule(processes, n);  
 return 0;  
}

**OUTPUT**



**EXPERIMENT NO-3**

**Inter-process Communication**

**Q - Write a program that creates a child process. Parent process writes data to pipe and child process reads the data from pipe and prints it on the screen.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/wait.h>

int main() {

int pipe\_fd[2];

pid\_t pid;

if (pipe(pipe\_fd) == -1) {

perror("Pipe creation failed");

exit(EXIT\_FAILURE);

}

pid = fork();

if (pid == -1) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

close(pipe\_fd[1]);

char buffer[1024];

int bytes\_read;

bytes\_read = read(pipe\_fd[0], buffer, sizeof(buffer));

if (bytes\_read == -1) {

perror("Read from pipe failed");

exit(EXIT\_FAILURE);

}

printf("Child Process: Received Data from Parent: %s\n", buffer);

close(pipe\_fd[0]);

exit(EXIT\_SUCCESS);

} else {

close(pipe\_fd[0]);

char data[] = "Hello, child process!";

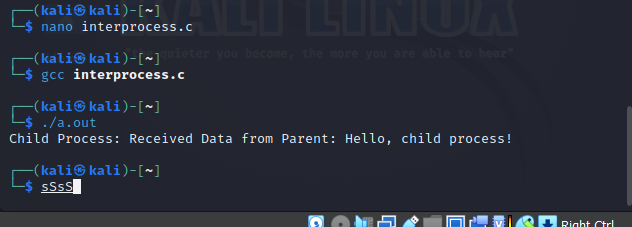
if (write(pipe\_fd[1], data, strlen(data) + 1) == -1) {

perror("Write to pipe failed");

exit(EXIT\_FAILURE);

}   
 close(pipe\_fd[1]);  
 wait(NULL);  
 exit(EXIT\_SUCCESS);  
 }  
}

**OUTPUT**



**EXPERIMENT NO-4**

**SEMAPHORES**

**(i) Write a program that demonstrates how two processes can share a variable using semaphore**

#include<pthread.h>

#include<stdio.h>

#include<semaphore.h>

#include<unistd.h>

void \*fun1();

void \*fun2();

int shared=1; //shared variable

sem\_t s; //semaphore variable

int main()

{

sem\_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is number of processes sharing semaphore, 3rd argument is the initial value of semaphore variable

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); //prints the last updated value of shared variable

}

void \*fun1()

{

int x;

sem\_wait(&s); //executes wait operation on s

x=shared;//thread1 reads value of shared variable

printf("Thread1 reads the value as %d\n",x);

x++; //thread1 increments its value

printf("Local updation by Thread1: %d\n",x);

sleep(1); //thread1 is preempted by thread 2

shared=x; //thread one updates the value of shared variable

printf("Value of shared variable updated by Thread1 is: %d\n",shared);

sem\_post(&s);

}

void \*fun2()

{

int y;

sem\_wait(&s);

y=shared;//thread2 reads value of shared variable

printf("Thread2 reads the value as %d\n",y);

y--; //thread2 increments its value

printf("Local updation by Thread2: %d\n",y);

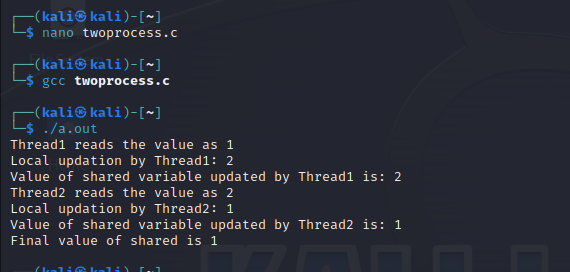
sleep(1); //thread2 is preempted by thread 1

shared=y; //thread2 updates the value of shared variable

printf("Value of shared variable updated by Thread2 is: %d\n",shared);

sem\_post(&s);

}

**OUTPUT**

**(ii)To write a C program to implement the Producer & consumer Problem (Semaphore)**

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size

// of buffer

int empty = 10, x = 0;

// Function to produce an item and

// add it to the buffer

void producer()

{

// Decrease mutex value by 1

--mutex;

// Increase the number of full

// slots by 1

++full;

// Decrease the number of empty

// slots by 1

--empty;

// Item produced

x++;

printf("\nProducer produces"

"item %d",

x);

// Increase mutex value by 1

++mutex;

}

// Function to consume an item and

// remove it from buffer

void consumer()

{

// Decrease mutex value by 1

--mutex;

// Decrease the number of full

// slots by 1

--full;

// Increase the number of empty

// slots by 1

++empty;

printf("\nConsumer consumes "

"item %d",

x);

x--;

// Increase mutex value by 1

++mutex;

}

// Driver Code

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'

// can give wrong value due to

// synchronization issues.

// 'critical' specifies that code is

// executed by only one thread at a

// time i.e., only one thread enters

// the critical section at a given time

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

// Switch Cases

switch (n) {

case 1:

// If mutex is 1 and empty

// is non-zero, then it is

// possible to produce

if ((mutex == 1)

&& (empty != 0)) {

producer();

}

// Otherwise, print buffer

// is full

else {

printf("Buffer is full!");

}

break;

case 2:

// If mutex is 1 and full

// is non-zero, then it is

// possible to consume

if ((mutex == 1)

&& (full != 0)) {

consumer();

}

// Otherwise, print Buffer

// is empty

else {

printf("Buffer is empty!");

}

break;

// Exit Condition

case 3:

exit(0);

break;

}

}

}

**OUTPUT**



**EXPERIMENT NO-5**

**Memory management-1**

#include <stdio.h>

#include <stdlib.h>

#define PAGE\_SIZE 4096

#define NUM\_PAGES 256

#define MEMORY\_SIZE (PAGE\_SIZE \* NUM\_PAGES)

// Page table entry structure

typedef struct {

int valid; // Flag indicating if the page is in memory (1) or not (0)

int frame\_number; // Frame number where the page is stored

} PageTableEntry;

// Physical memory (frames)

char physical\_memory[MEMORY\_SIZE];

// Page table

PageTableEntry page\_table[NUM\_PAGES];

// Function to allocate a page

int allocate\_page() {

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

if (page\_table[i].valid == 0) {

page\_table[i].valid = 1;

return i;

}

}

return -1; // No available pages

}

// Function to load data into a page

void load\_page(int page\_number, char\* data) {

if (page\_number >= 0 && page\_number < NUM\_PAGES) {

int frame\_start = page\_number \* PAGE\_SIZE;

for (int i = 0; i < PAGE\_SIZE; i++) {

physical\_memory[frame\_start + i] = data[i];

}

page\_table[page\_number].frame\_number = frame\_start;

}

}

// Function to access memory

char access\_memory(int logical\_address) {

int page\_number = logical\_address / PAGE\_SIZE;

int offset = logical\_address % PAGE\_SIZE;

if (page\_table[page\_number].valid == 1) {

int frame\_start = page\_table[page\_number].frame\_number;

return physical\_memory[frame\_start + offset];

} else {

printf("Page %d is not in memory.\n", page\_number);

return '\0';

}

}

int main() {

// Initialize page table entries

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

page\_table[i].valid = 0;

}

// Allocate and load pages

int page1 = allocate\_page();

int page2 = allocate\_page();

char data1[PAGE\_SIZE] = "This is page 1.";

char data2[PAGE\_SIZE] = "This is page 2.";

load\_page(page1, data1);

load\_page(page2, data2);

// Access memory

printf("Data at logical address 0: %c\n", access\_memory(0));

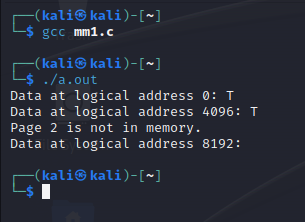
printf("Data at logical address 4096: %c\n", access\_memory(4096));

printf("Data at logical address 8192: %c\n", access\_memory(8192));

return 0;

}

**OUTPUT**

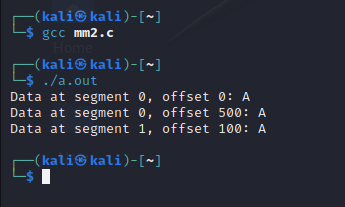


**EXPERIMENT NO-6**

**Memory management-2**

#include <stdio.h>  
#include <stdlib.h>  
  
#define MAX\_SEGMENTS 10  
  
// Segment table entry structure  
typedef struct {  
 int base\_address; // Base address of the segment  
 int size; // Size of the segment  
} SegmentTableEntry;  
  
// Segment table  
SegmentTableEntry segment\_table[MAX\_SEGMENTS];  
  
// Function to allocate a segment  
int allocate\_segment(int size) {  
 for (int i = 0; i < MAX\_SEGMENTS; i++) {  
 if (segment\_table[i].size == 0) {  
 segment\_table[i].base\_address = rand() % 10000; // Assign a random base address (for simplicity)  
 segment\_table[i].size = size;  
 return i;  
 }  
 }  
 return -1; // No available segments  
}  
  
// Function to access memory  
char access\_memory(int segment\_number, int offset) {  
 if (segment\_number >= 0 && segment\_number < MAX\_SEGMENTS) {  
 int base\_address = segment\_table[segment\_number].base\_address;  
 int size = segment\_table[segment\_number].size;  
  
 if (offset >= 0 && offset < size) {  
 // You can access memory within the segment's size here  
 // For simplicity, we're returning a character 'A' as data  
 return 'A';  
 }  
 }  
 printf("Segment %d is invalid or offset is out of bounds.\n", segment\_number);  
 return '\0';  
}  
  
int main() {  
 // Initialize segment table entries  
 for (int i = 0; i < MAX\_SEGMENTS; i++) {  
 segment\_table[i].base\_address = 0;  
 segment\_table[i].size = 0;  
 }  
 // Allocate segments  
 int segment1 = allocate\_segment(1000);  
 int segment2 = allocate\_segment(500);  
 // Access memory within the allocated segments  
 printf("Data at segment %d, offset 0: %c\n", segment1, access\_memory(segment1, 0));  
 printf("Data at segment %d, offset 500: %c\n", segment1, access\_memory(segment1, 500));  
 printf("Data at segment %d, offset 100: %c\n", segment2, access\_memory(segment2, 100));  
  
 return 0;  
}

**OUTPUT**



**EXPERIMENT NO-7**

**FILE MANIPULATION**

1. **Write a program that displays all the files and directories.**

#include <stdio.h>

#include <dirent.h>

int main(void)

{

struct dirent \*de; // Pointer for directory entry

// opendir() returns a pointer of DIR type.

DIR \*dr = opendir(".");

if (dr == NULL) // opendir returns NULL if couldn't open directory

{

printf("Could not open current directory" );

return 0;

}

// Refer http://pubs.opengroup.org/onlinepubs/7990989775/xsh/readdir.html

// for readdir()

while ((de = readdir(dr)) != NULL)

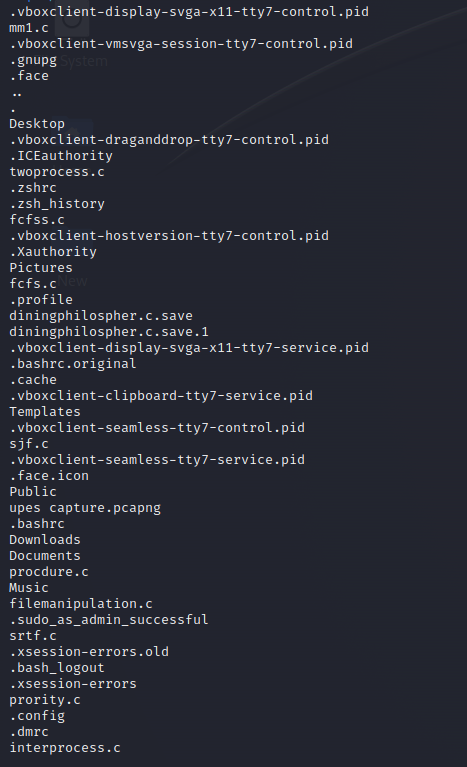
printf("%s\n", de->d\_name);

closedir(dr);

return 0;

}

**OUTPUT**



1. **Write a program to create new directory.**

// C program to create a directory

// using mkdir() function

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int main()

{

char dirName[16];

int ret = 0;

printf("Enter directory name: ");

scanf("%s", dirName);

ret = mkdir(dirName, 0755);

if (ret == 0)

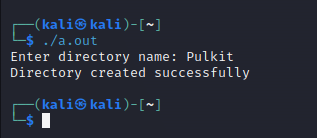
printf("Directory created successfully\n");

else

printf("Unable to create directory %s\n", dirName);

return 0;

}

**OUTPUT**

**EXPERIMENT NO-8**

**Fork Execution**

**1) Simple fork execution**

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t child\_pid;

child\_pid = fork();

if (child\_pid == -1) {

perror("Fork failed");

return 1;

}

if (child\_pid == 0) {

printf("Child process is running. PID: %d\n", getpid());

} else {

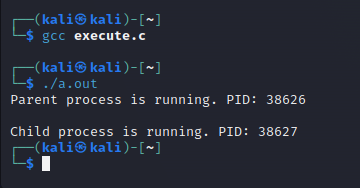
printf("Parent process is running. PID: %d\n", getpid());

}

return 0;

}

**OUTPUT**



**2) fork system call**

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t child\_pid;

child\_pid = fork();

if (child\_pid == -1) {

perror("Fork failed");

return 1;

}

if (child\_pid == 0) {

printf("Child process is running. PID: %d\n", getpid());

} else {

printf("Parent process is running. PID: %d\n", getpid());

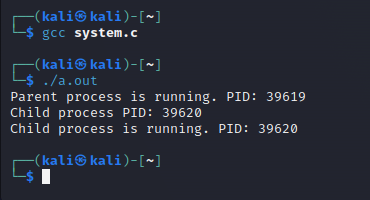
printf("Child process PID: %d\n", child\_pid);

}

return 0;

}

**OUTPUT**



**EXPERIMENT NO-9 & 10**

**IMPLEMENTATION OF BANKER’S AND DINING PHILOSPHER’S ALGORITHM**

**1) Write a program to implement banker’s algorithm.**

// 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]);

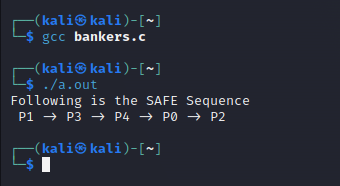
printf(" P%d", ans[n - 1]);

}

return (0);

}

**OUTPUT**



**2) Write a program to implement Dining Philospher’s algorithm.**

#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**

