OS LAB REPORT

Bikal Devkota
076BCT017
DOECE, Pulchowk Campus.
076bct017.bikal@pcampus.edu.np

Ashim Baral
076BCT011
DOECE, Pulchowk Campus.
076bct011.ashim@pcampus.edu.np

Bibek Shrestha
076BCT014
DOECE, Pulchowk Campus.
076bct014.bibek@pcampus.edu.np

Anubhav Khanal
076BCT009
DOECE, Pulchowk Campus.
076bct009.anubhav@pcampus.edu.np

***Abstract*—The OS lab provided practical learning opportunities on various aspects of operating systems such as shell scripting, process management, synchronization, and memory management. Students were introduced to Unix/Linux shells, system calls, process scheduling, synchronization problems, and memory management techniques. The lab helped them develop hands-on skills and a comprehensive understanding of fundamental operating system concepts that are highly beneficial for careers in computer systems or software development.**

I. SHELL PROGRAMMING

Shell programming involves creating and executing scripts in a shell language such as Bash, Zsh or Korn Shell. A shell script is a program that is interpreted by a shell - a command-line interface that allows users to interact with the operating system through commands.

Shell scripts can be used to automate repetitive tasks, perform system administration functions, manipulate data and run other programs. This makes shell programming a widely used skill in Unix and Linux environments, as well as on Windows systems with a Unix-like shell installed.

Knowledge of basic programming concepts and familiarity with the syntax and capabilities of the specific shell language being used are essential for effective shell programming. This skill is particularly valuable for system administrators, developers and other IT professionals who need to automate tasks and manage systems efficiently.

1) CONCATENATION OF TWO STRINGS

```
# To concatenate two strings
echo "Enter the first string:"
read str1
echo "Enter the second string:"
read str2
echo "the concatenated string in
    nepal is "str1$str2
```

This shell script asks the user to input two strings. It then combines them using the syntax “str1str2”. The resulting string is displayed on the terminal using the “echo” command.

2) COMPARISON OF TWO STRINGS

```
# Compare two strings
#!/bin/bash

echo "Enter the first string:"
read str1
echo "Enter the second string:"
read str2
if [ $str1 = $str2 ]
then
    echo "Strings are equal."
else
    echo "Strings are unequal."
fi
```

This shell script asks the user to input two strings. It then combines them using the syntax “str1str2”. The resulting string is displayed on the terminal using the “echo” command.

3) MAXIMUM OF THREE NUMBERS

```
# Greatest of three numbers

echo "Enter A"
read a
echo "Enter B"
read b
echo "Enter C"
read c
if [ $a -gt $b -a $a -gt $c ]
then
    echo "A is greatest"
elif [ $b -gt $a -a $b -gt $c ]
then
    echo "B is greatest"
else
    echo "C is greatest"
fi
```

This shell script prompts the user to enter three numbers and then determines the greatest of the three numbers using if-else statements. The script first reads in three numbers using ”read” command and then compares them in a series of if-else statements to determine the largest number. If the first number entered is the greatest, the script prints ”A is greatest”. If the second number entered is the greatest, the script prints ”B is greatest”. Otherwise, the script prints ”C is greatest”.

4) FIBONACCI SERIES

```
# Generate fibonacci series
#!/bin/bash

i=0
a=0
b=1
while [ $i -lt 10 ]
do
    echo -n "$a "
    c=$((a+b))
    a=$b
    b=$c
    i=$((i+1))
done
```

This shell script generates the first 10 numbers of the Fibonacci sequence using a while loop. The first two numbers in the sequence are initialized to 0 and 1, and each subsequent number is the sum of the two preceding numbers. The numbers are printed to the terminal separated by spaces using the ”echo -n” command. The loop iterates 10 times and then terminates.

5) ARITHMETIC OPERATIONS USING CASE

```
# Arithmetic operation using case
#!/bin/bash

read -p "Enter first number: " a
read -p "Enter second number: " b
read -p "Enter operator (+, -, *, /):
     " op

case $op in
    '+') r=$((a+b)) ;;
    '-') r=$((a-b)) ;;
    '*') r=$((a*b)) ;;
    '/') r=$((a/b)) ;;
    *) echo "Invalid operator" ;;
esac

echo "$a $op $b = $r"
```

This shell script prompts the user to enter two numbers and an arithmetic operator (+, -, *, /) and performs the corresponding arithmetic operation on the two numbers using a case statement. The result of the operation is stored in the variable ”r”. If the operator entered by the user is not one of the valid options, an error message is printed to the terminal.

II. SYSTEM CALL

A system call is a way for programs to request services from the kernel of an operating system. It allows user-level programs to perform privileged operations such as accessing hardware devices, managing files and memory. The system call interface provides a standard method for programs to interact with the operating system and enables the operating system to serve multiple programs running on the same computer concurrently. Some examples of system calls are open, close, read, write, fork, exec and exit.

1) PROCESS CREATION

```
// Create a process in UNIX.
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    int id;
    printf("Before fork: Process id:
        %d\n", getpid());

    // Create a new process
    id = fork();
    if (id == 0) {
        printf("Child process, PID: %
            d\n", getpid());
    } else if (id > 0) {
        printf("Parent process, PID:
            %d\n", getpid());
    } else {
        printf("fork() failed.\n");
    }
    return 0;
}
```

This code uses the fork() system call to create a new process in UNIX. It first displays the process ID of the current process using the getpid() function. Then it calls fork() to create a new process.
If fork() returns 0, it indicates that the current process is the child process. The code then displays a message stating that it is the child process and its process ID using getpid().
If fork() returns a value greater than 0, it indicates that the current process is the parent process. The code then displays a message stating that it is the parent process and its process ID using getpid().
If fork() returns a value less than 0, it indicates that the fork failed and an error message is displayed. The code then returns 0 to indicate successful termination of the program.

2) EXECUTING A COMMAND

```
#!/bin/bash
# Program for executing UNIX command
   using shell programming

echo "Program for executing UNIX
   command using shell programming"
echo "Display currently running
   processes using 'ps' command."
ps

echo "List files and directories in a
    directory."
exec ls -a
```

This is a basic Bash script that displays a message on the console and runs the ps command to show the processes currently running on the system. The script's aim is to illustrate how to run UNIX commands using Bash shell programming.

3) SLEEP COMMAND

```
// Create a child with sleep command.
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int id;
    id = fork();

    if (id == 0) {
        sleep(3);
        printf("Child process, PID: %
           d\n", getpid());
    } else if (id > 0) {
        printf("Parent process, PID:
           %d\n", getpid());
        wait(NULL);
    } else {
        printf("fork() failed.\n");
    }
    return 0;
}
```

This code uses the fork() system call to create a child process and then puts it to sleep for 3 seconds using the sleep() function. After 3 seconds have passed, the child process wakes up and prints its process ID (PID) to the console. In the meantime, the parent process prints its own PID to the console and waits for its child process to finish executing before exiting. If for some reason the fork() system call fails, an error message is printed to the console.

4) SLEEP COMMAND USING GETPID

```
// Create a child with sleep command
   using getpid.
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

void display_pid() {
    printf("\n PID: %d\n Its Parent
       process PID:%d\n\n", getpid(),
        getppid());
}

int main() {
    int id;
    id = fork();

    if (id == 0) {
        printf("Child process:");
        display_pid();
        sleep(5);
        printf("Child process after
           sleep=5");
        display_pid();
    } else if (id > 0) {
        printf("Parent process\n\n");
        sleep(10);
        printf("Parent process after
           sleep=10");
        display_pid();
        wait(NULL);
    } else {
        printf("fork() failed.\n");
    }
    return 0;
}
```

This program uses the fork() system call to create a child process. The child process then displays its process ID (PID) and its parent process ID (PPID) using the display$_pid()function.Afterthat,itsleepsfor5secondsbeforedisp$

5) SIGNAL HANDLING

```
#!/bin/bash
# Signal Handling

echo "Program for performing KILL
   operations"
ps

echo "Enter the pid"
read pid

kill -9 $pid
echo "Finished"
```

This shell script allows the user to forcefully terminate a specified process by its process ID (PID). The script first displays a list of all currently running processes using the ps command. Then, it prompts the user to enter the PID of the process they want to terminate. The kill command with the -9 option is used to end the specified process. Once completed, the script outputs a message indicating that the operation has finished

6) WAIT COMMAND

```
// Perform wait command using C
    program.
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int id;
    id = fork();

    if (id == 0) {
        printf("Child process, PID: %
            d\n", getpid());
        return 0;
    } else if (id > 0) {
        int status;
        wait(&status);
        printf("Parent process, PID:
            %d\n", getpid());
        printf(" Its Child process
            PID:%d terminated with
            status %d\n", id, status);
    } else {
        printf("fork() failed.\n");
    }
    return 0;
}
```

This C program shows how to use the wait() command to synchronize a parent process with its child process. The program creates a child process using fork(). The child process prints its PID and exits. The parent process uses the wait() command to wait for the child process to complete. After the child process has finished, the parent prints its own PID, the PID of the child and its exit status. This demonstrates how wait() can be used to make sure that a parent waits for its child to finish before proceeding.

III. FILE HANDLING

File handling is the process of managing files in a computer program. This includes reading from files, writing to files and manipulating them in different ways. Files are often used to store data that needs to be retained between program runs or shared between programs. For instance, a program may read configuration settings or user preferences from a file or write log messages or computation results to a file.

Typically, file handling involves opening a file, performing an operation on it and then closing it. When a file is opened, the program can read from or write to it. When the file is closed, any changes made are saved to disk.

a

1) READING FROM A FILE

```
#include <stdio.h>
int main() {
    char
        str[100];
    FILE *fp;
    fp = fopen("file1.dat", "r");
    while (!feof(fp))
    {
        fscanf(fp, "%s", str);
        printf(" %s ", str);
    }
    fclose(fp);
}
```

This C code prompts the user to enter a string, reads the string using the ”gets” function, and stores it in a character array named ”str”. It then opens a file named ”file1.dat” in write mode using the ”fopen” function, and writes the contents of the ”str” array to the file using the ”fprintf” function.

2) WRITING INTO A FILE

```
#include <stdio.h>
int main() {
    char
        str[100];
    FILE *fp;
    printf("Enter the string");
    gets(str);
fp=fopen("file1.dat","w
+");while(!feof(fp))
{
        fscanf(fp, "%s", str);
}

fprintf(fp,"%s",str);
}
```

This C program reads a string entered by the user using the “gets” function and stores it in a character array called “str”. The program then opens a file named “file1.dat” in write mode with the “fopen” function and writes the contents of the “str” array to the file using the “fprintf” function.

3) FILE CREATION

```
#include <stdio.h>
int main() {
    int id;
    if ((id = creat("z.txt", 0)) ==
        -1) {
```

```
        printf("Cannot create the
            file");
        exit(1);
    }
    else {
        printf("File is created");
        exit(1);
    }
    return 0;
}
```

This C program uses the “creat” function to create a new file called “z.txt” or overwrite an existing file with the same name. The program checks if the “creat” function was successful by assigning its return value to a variable named “id” and comparing it to -1. If the return value is -1, an error message is printed and the program exits with an error code. Otherwise, a success message is printed and the program exits with a success code.

IV. IMPLEMENTATION OF LS COMMAND

```
// C program to simulate the operation of
    'ls' command in UNIX

#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<dirent.h>

int main(int argc, char *argv[]) {
    DIR* dp;
    struct dirent* dirp;
    if (argc<2) {
        printf("You have provided only 1
            argument\n");
        exit(1);
    }
    if ((dp=opendir(argv[1])) == NULL) {
        printf("\nCannot open %s file!\n"
            , argv[1]);
        exit(1);
    }
    while ((dirp=readdir(dp))!=NULL)
        printf("%s\n", dirp->d_name);
    closedir(dp);
}
```

This C program mimics the functionality of the UNIX ‘ls’ command. It accepts a directory path as an input and displays the contents of that directory.
Initially, the program checks if an argument has been provided. If not, it displays an error message and exits. Then it attempts to open the specified directory using the opendir() function. If opendir() is unsuccessful, an error message is displayed and the program exits. If successful, it reads and displays each entry in the directory using a loop with readdir(). Once all entries have been read, the program closes the directory using closedir().

V. PROCESS SCHEDULING

Process scheduling is a technique used by operating systems to allocate resources for the efficient execution of multiple processes. With multiple processes running at the same time, it’s important to allocate CPU time in an optimal way. The scheduling algorithm determines the order of process execution and how long each process runs.

The goal of process scheduling is to achieve good system performance through fast response times, high throughput and efficient resource utilization. Scheduling algorithms can be either preemptive or non-preemptive, depending on whether a running process can be interrupted.

There are several types of scheduling algorithms including First-Come-First-Served (FCFS), Shortest-Job-First (SJF), Round Robin (RR), Priority Scheduling and Multilevel Feedback Queue (MLFQ). The choice of algorithm depends on system requirements and desired performance.

a

1) FIRST COME FIRST SERVE

```
// First Come First Serve (FCFS) CPU
    Scheduling Alogrithm

#include <stdio.h>

struct Process {
    int pid, bt, at, wt, tat;
};

int main() {
    int n, i, j;
    printf("Enter the number of
        processes: ");
    scanf("%d", &n);

    struct Process p[n], temp;
    printf("Enter burst time and
        arrival time:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ",i+1);
        scanf("%d %d", &p[i].bt, &p[i
            ].at);
        p[i].pid = i+1;
    }


    for (i = 0; i < n; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (p[j].at > p[j+1].at)
                {
```

```
                temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
        }
    }


    int current_time = 0;
    float sum_wt=0, avg_wt;
    float sum_tat=0, avg_tat ;
    printf("Process\t\tBurst Time\
       tArrival Time\tWaiting Time\
       tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        p[i].wt = current_time;
        current_time += p[i].bt;
        p[i].tat = current_time-p[i].
            at;
        sum_wt += p[i].wt;
        sum_tat += p[i].tat;
        printf("P%d\t\t%d\t\t%d\t\t%d
            \t\t%d\n", p[i].pid, p[i].
            bt, p[i].at, p[i].wt, p[i
            ].tat);
    }

    avg_wt = sum_wt/n;
    avg_tat = sum_tat/n;
    printf("\nAverage Waiting time (
       AWT) = %.2f", avg_wt);
    printf("\nAverage Turnaround time
         (ATAT) = %.2f\n", avg_tat);

    return 0;
}
```

This C program simulates the First Come First Serve (FCFS) CPU Scheduling algorithm. It prompts the user to enter the number of processes and their burst time and arrival time. These details are stored in an array of Process structures.
The program then sorts the processes based on their arrival time using bubble sort. After sorting, it calculates the waiting time and turnaround time for each process according to the FCFS algorithm.
The program outputs the details of each process including Process ID, Burst Time, Arrival Time, Waiting Time and Turnaround Time. It also computes and displays the average waiting time and average turnaround time for all processes.

2) SHORTEST JOB FIRST

```
// Shortest Job First (SJF) CPU
   Scheduling Alogrithm

#include <stdio.h>

struct Process {
    int pid, bt, wt, tat;
};

int main() {
    int n, i, j;
    printf("Enter the number of
       processes: ");
    scanf("%d", &n);

    struct Process p[n], temp;
    printf("Enter burst time:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ",i+1);
        scanf("%d", &p[i].bt);
        p[i].pid = i+1;
    }


    for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++) {
            if (p[i].bt > p[j].bt) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }


    int current_time = 0;
    float sum_wt=0, avg_wt;
    float sum_tat=0, avg_tat ;
    printf("Process\t\tBurst Time\
       tWaiting Time\tTurnaround Time
       \n");
    for (i = 0; i < n; i++) {
        p[i].wt = current_time;
        current_time += p[i].bt;
        p[i].tat = current_time;
        sum_wt += p[i].wt;
        sum_tat += p[i].tat;
        printf("P%d\t\t%d\t\t%d\t\t%d
            \n", p[i].pid, p[i].bt, p[
            i].wt, p[i].tat);
    }

    avg_wt = sum_wt/n;
    avg_tat = sum_tat/n;
    printf("\nAverage Waiting time (
       AWT) = %.2f", avg_wt);
    printf("\nAverage Turnaround time
         (ATAT) = %.2f\n", avg_tat);

    return 0;
}
```

This C program implements the Shortest Job First (SJF) CPU Scheduling algorithm. It prompts the user to enter the number of processes and their burst time. The processes are then sorted in ascending order based on their burst time.
The program calculates the waiting time and turnaround time for each process and displays the results in a table. It also computes and displays the average waiting time and average turnaround time for all processes.
The SJF algorithm selects the process with the shortest burst time to be executed next. This results in a lower average waiting time and turnaround time compared to other scheduling algorithms. However, this algorithm may not always be optimal as it can result in starvation of longer processes that arrive first.

3) ROUND ROBIN

```
// Round Robin Scheduling Algorithm

#include <stdio.h>

struct Process {
    int pid, bt, at, wt, rt, tat;
};

int main() {
    int n, i, j, quantum;
    printf("Enter the number of
       processes: ");
    scanf("%d", &n);
    printf("Enter the quantum time: "
       );
    scanf("%d", &quantum);

    struct Process p[n];
    printf("Enter burst time and
       arrival time:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ",i+1);
        scanf("%d %d", &p[i].bt, &p[i
           ].at);
        p[i].pid = i+1;
        p[i].rt = p[i].bt;
    }

    int current_time = 0;
    int completed_processes = 0;
    while (completed_processes < n) {
        for (i = 0; i < n; i++) {
            if (current_time >= p[i].
               at && p[i].rt > 0) {
                if (p[i].rt > quantum
                   ) {
                    current_time +=
                       quantum;
                    p[i].rt -=
                       quantum;
                } else {
                    current_time += p
                       [i].rt;
                    p[i].wt =
                       current_time -
                        p[i].at - p[i
                       ].bt;
                    p[i].tat =
                       current_time -
                        p[i].at;
                    p[i].rt = 0;
                    completed_processes
                       ++;
                }
            }
        }
    }


    float sum_waiting = 0,
       sum_turnaround = 0,
       avg_waiting = 0,
       avg_turnaround = 0;
    printf("Process ID\tBurst Time\
       tArrival Time\tWaiting Time\
       tTurnaround Time\n");
    for (i = 0; i < n; i++) {
         printf("%d\t\t%d\t\t%d\t\t%d\
            t\t%d\n", p[i].pid, p[i].
            bt, p[i].at, p[i].wt, p[i
            ].tat);
         sum_waiting += p[i].wt;
         sum_turnaround += p[i].tat;
    }
    avg_waiting = sum_waiting/n;
    avg_turnaround = sum_turnaround/n
       ;
    printf("\nAverage Waiting time (
       AWT) = %.2f", avg_waiting);
    printf("\nAverage Turnaround time
        (ATAT) = %.2f\n",
       avg_turnaround);
    return 0;
}
```

This code implements the Round Robin CPU Scheduling algorithm. The program takes input for the number of processes and quantum time from the user. It then prompts the user to enter burst time and arrival time for each process. A structure Process is defined to store process information such as process ID, burst time, arrival time, waiting time, response time and turnaround time.
The Round Robin algorithm is implemented using a

while loop that runs until all processes are completed. Inside this loop is a for loop that iterates over each process. If the current time is greater than or equal to the arrival time of a process and its remaining time is greater than 0, then the process is executed for quantum time. If its remaining time is less than or equal to quantum time, then it's completed and its waiting time, turnaround time and response times are calculated. Otherwise, its remaining time is updated. After all processes are completed, the program prints each process's information such as ID, burst time, arrival time, waiting and turnaround times. It also calculates and displays average waiting and turnaround times.

4) PRIORITY SCHEDULING

```
// Priority Scheduling Alogrithm

#include <stdio.h>

struct Process {
    int pid, bt, wt, tat, priority;
};

int main() {
    int n, i, j;
    printf("Enter the number of
       processes: ");
    scanf("%d", &n);

    struct Process p[n], temp;
    printf("Enter burst time and
       priority no:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ",i+1);
        scanf("%d %d", &p[i].bt, &p[i
           ].priority);
        p[i].pid = i+1;
    }


    for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++) {
            if (p[i].priority > p[j].
               priority) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }


    int current_time = 0;
    float sum_wt=0, avg_wt;
    float sum_tat=0, avg_tat ;
    printf("Process\t\tPriority\t\
       tBurst Time\tWaiting Time\
       tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        p[i].wt = current_time;
        current_time += p[i].bt;
        p[i].tat = current_time;
        sum_wt += p[i].wt;
        sum_tat += p[i].tat;
        printf("P%d\t\t%d\t\t%d\t\t%d
           \t\t%d\n", p[i].pid, p[i].
           priority, p[i].bt, p[i].wt
           , p[i].tat);
    }

    avg_wt = sum_wt/n;
    avg_tat = sum_tat/n;
    printf("\nAverage Waiting time (
       AWT) = %.2f", avg_wt);
    printf("\nAverage Turnaround time
        (ATAT) = %.2f\n", avg_tat);

    return 0;
}
```

This program implements the Priority Scheduling algorithm for process scheduling. It takes the number of processes and their burst time and priority as input from the user. Then it sorts the processes in ascending order of their priority.
The program simulates scheduling by assigning waiting and turnaround times to each process. The waiting time of a process is the sum of burst times of all previous processes while its turnaround time is the sum of burst times of all previous processes and its own burst time. Finally, it calculates average waiting and turnaround times for all processes and prints them.

VI. PRODUCER CONSUMER PROBLEM USING SEMAPHORE

A semaphore is a synchronization object used to control access to shared resources in concurrent systems. It's typically a variable that can be accessed by multiple processes or threads and used to signal between them.

The producer-consumer problem is a classic synchronization problem in computer science involving two processes sharing a common buffer or queue. The producer generates data and adds it to the buffer while the consumer reads data from the buffer and processes it. The challenge is ensuring that the producer and consumer don't interfere with each other's operations and that the buffer is used efficiently. This problem can be solved using various synchronization techniques such as semaphores, mutexes, and monitors among others.

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

```c
#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t mutex, empty, full;

void* producer(void* arg) {
    int item;
    while(1) {
        // Produce an item
        item = rand() % 100;

        // Wait for an empty slot
        sem_wait(&empty);
        sem_wait(&mutex);

        // Put the item in the buffer
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;

        printf("Produced item %d\n", item
           );

        // Signal that the buffer is no
           longer empty
        sem_post(&mutex);
        sem_post(&full);
    }
}

void* consumer(void* arg) {
    int item;
    while(1) {
        // Wait for a full slot
        sem_wait(&full);
        sem_wait(&mutex);

        // Take an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        printf("Consumed item %d\n", item
           );

        // Signal that the buffer is no
           longer full
        sem_post(&mutex);
        sem_post(&empty);

        // Consume the item
        // ...
    }
}

int main() {
    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Create threads
    pthread_t producer_thread,
       consumer_thread;
    pthread_create(&producer_thread, NULL
       , producer, NULL);
    pthread_create(&consumer_thread, NULL
       , consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Clean up semaphores
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

This is a multi-threaded program that demonstrates the producer-consumer problem using semaphores in C. The program has two threads: a producer thread and a consumer thread. The producer thread generates random integer items and puts them in a shared buffer while the consumer thread takes items from the buffer and consumes them (in this case, just prints them to the console). The program uses three semaphores (mutex, empty, and full) to synchronize access to the buffer between the two threads. Mutex is a binary semaphore that ensures only one thread at a time can access the buffer while empty and full are counting semaphores that keep track of the number of empty and full slots in the buffer respectively.

VII. MEMORY MANAGEMENT

Memory management in operating systems is the process of managing computer memory resources efficiently. This includes tracking how memory is used, allocating memory to processes that need it and freeing up memory that is no longer needed. The main goals of memory management are to ensure that every process has enough memory to run, that memory is allocated efficiently and that no process can interfere with another process's memory. This involves keeping track of which memory is available and which is in use and using techniques such as virtual memory, paging and segmentation to make the most efficient use of available memory.

a

1) MEMORY MANAGEMENT SCHEME- PAGING

```c
// Memory Management Scheme - Paging

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

int main() {
    int base_address, page_size,
       num_pages, mem_unit,
       memory_limit;
    int page_number, displacement,
       physical_address;
    int *page_table;

    printf("Program for Paging");
    printf("Enter:\n");

    printf("base address: ");
    scanf("%d", &base_address);

    printf("page size: ");
    scanf("%d", &page_size);

    printf("number of pages: ");
    scanf("%d", &num_pages);

    printf("memory unit: ");
    scanf("%d", &mem_unit);

    memory_limit = mem_unit *
       num_pages;
    if (memory_limit < base_address)
       {
        printf("Memory limit is less
           than base address.\n");
        return 0;
    }

    // Create the page table
    printf("\nPage Table:\nPageNo.\
       tBase Address");
    page_table = (int *)malloc(
       num_pages * sizeof(int));
    for (int i = 0; i < num_pages; i
       ++) {
        page_table[i] = base_address
           + i * page_size;
        printf("\n%d\t%d", i,
           page_table[i]);
    }

    printf("\n\nEnter the page number
        and displacement value: ");
    scanf("%d %d", &page_number, &
       displacement);

    if (page_number >= num_pages ||
       displacement >= page_size) {
        printf("Page not found or
           displacement should be
           less than page size.\n");
        return 0;
    }

    // Calculate the physical address
    physical_address = page_table[
       page_number] + displacement;
    printf("\nPage No.\tBase Address\
       tPhysical Address\n");
    printf("%d\t\t%d\t\t%d\n",
       page_number, page_table[
       page_number], physical_address
       );
    return 0;
}
```

This program is a basic implementation of a paging memory management system. It prompts the user to input values for base address, page size, number of pages and memory unit size. Using these inputs, it calculates the memory limit and creates a page table using the malloc() function. The page table is then displayed on the console.
The user is then asked to enter a page number and displacement value. The program verifies that the input is valid and calculates the physical address by adding the base address of the specified page number to the displacement value.

2) MEMORY MANAGEMENT SCHEME-SEGMENTATION

```
// Memory Management Scheme -
   Segmentation

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

void main() {
    int base_address[20], limit[20],
       num_segments, memory_limit;
    int segment_number, displacement,
        physical_address;

    printf("Enter number of segments:
        ");
    scanf("%d", &num_segments);
    printf("Enter memory limit: ");
    scanf("%d", &memory_limit);

    printf("\nEnter base address and
       limit of each segment:\n");
    for (int i = 0; i < num_segments;
        i++) {
        printf("Segment %d: ", i);
        scanf("%d %d", &base_address[
           i], &limit[i]);
        if (base_address[i] + limit[i
           ] > memory_limit) {
            printf("Invalid memory
               limit \n");
            exit(0);
        }
    }

    printf("\nEnter the segment
       number and displacement value:
        ");
    scanf("%d %d", &segment_number, &
       displacement);

    if (segment_number >=
       num_segments || displacement
       >= limit[segment_number]) {
        printf("Invalid segment
           number or displacement.\n"
           );
        exit(0);
    }

    // Calculate the physical address
    physical_address = base_address[
       segment_number] + displacement
       ;
    printf("\nSegment No.\tBase
       Address\tPhysical Address\n");
    printf("%d\t\t%d\t\t%d\n",
       segment_number, base_address[
       segment_number],
       physical_address);
}
```

This code is an implementation of the Segmentation memory management technique. It starts by asking the user to enter the number of segments and memory limit. Then it requests the base address and limit for each segment. If any segment's base address plus its limit exceeds the memory limit, the program terminates. Next, it prompts for a segment number and displacement value to compute the physical address. If either of these values is invalid, the program terminates. Finally, it calculates and displays the physical address along with its corresponding segment number and base address.

VIII. CONCLUSION

In the OS lab, we studied several significant topics related to operating systems. The first lab introduced us to shell programming and basic shell commands and vi - essential tools for Linux users. The second lab focused on process creation and execution and taught us about sleep, wake, and PID - crucial concepts for understanding how processes function in an operating system.

In the third lab, we implemented the ls command in C, deepening our understanding of file management in operating systems. The fourth lab covered process scheduling algorithms used to manage resources and ensure efficient process execution. We also learned about synchronization and semaphore through the producer-consumer problem - an important concept for preventing race conditions and ensuring synchronized process execution.

Finally, we studied memory management in operating systems - a critical concept for understanding resource usage by processes. We learned about paging and segmentation - two techniques commonly used by operating systems to manage memory allocation.

Overall, this course gave us a thorough understanding of how operating systems function and the key concepts and tools used to manage system resources. By implementing various commands and algorithms, we gained practical experience and enhanced our knowledge of operating systems.