Bachelor of Computer Applications (BCA)

OPERATING SYSTEMS LAB

(OBCACO306P24)

Self-Learning Material
(SEM III)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

TABLE OF CONTENTS

EXPERT COMMITTEE

Prof. Sunil Gupta
(Computer and Systems Sciences, JNU Jaipur)

Dr. Deepak Shekhawat
(Computer and Systems Sciences, JNU Jaipur)

Dr. Satish Pandey
(Computer and Systems Sciences, JNU Jaipur)

COURSE COORDINATOR

Ms. Rashmi Choudhary
(Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)	Assisting & Proofreading	Unit Editor
Mr. Shish Dubey (Computer and Systems Sciences, JNU Jaipur)	Mr. Ram Lal Yadav (Computer and Systems Sciences, JNU Jaipur)	Dr. Deepak Shekhwat (Computer and Systems Sciences, JNU Jaipur)

Secretarial Assistance

Mr. Mukesh Sharma

COURSE INTRODUCTION

Welcome to the Operating System Lab course! This hands-on lab is designed to give you a deeper understanding of the concepts and principles behind operating systems by allowing you to engage with real-world applications and scenarios. Throughout this course, you'll explore the core functions of operating systems, such as process management, memory management, file systems, and system calls, through practical exercises and experiments.

You'll work with different operating system environments, gaining insights into how various components interact to provide a seamless computing experience. By diving into tasks like process scheduling, inter-process communication, and system performance tuning, you'll not only reinforce theoretical knowledge but also develop valuable skills for troubleshooting and optimizing operating systems.

This lab is structured to enhance your problem-solving abilities and critical thinking through hands-on experience. Expect to work on a series of projects and assignments that will challenge you to apply what you've learned in practical, real-world contexts. Whether you're debugging system issues or designing efficient algorithms for resource management, this lab will equip you with a solid foundation in operating systems and prepare you for future technical challenges.

Course Outcomes:

At the completion of the course, a student will be able to:

1. Remember the functions, structures and history of operating systems.
2. Understand of design issues associated with operating systems
3. Apply concepts including scheduling, synchronization and deadlocks.
4. Distinguish multithreading , Multitasking & Multiprogramming and also able to explain the concept of memory management including virtual Evaluate the requirement for process synchronization and coordination handled by operating system
5. Categorize memory organization and explain the function of each element of a memory hierarchy and analyze its allocation policies.
6. Conceptualize the components involved in designing a contemporary OS.

Acknowledgements:

The content we have utilized is solely educational in nature. The copyright proprietors of the materials reproduced in this book have been tracked down as much as possible. The editors apologize for any violation that may have happened, and they will be happy to rectify any such material in later versions of this book.

Assignment 1: Process Creation and Termination

Program Statement: Write a C program to demonstrate process creation and termination using the **fork()** system call. The program should:

1. Create a child process.
2. Print the process IDs of both the parent and child processes.
3. The child process should print a message and then terminate.
4. The parent process should wait for the child to terminate and then print a termination message.

Solution Description: By copying the caller process, the **fork()** system function creates a new process. The parent and the child processes are the outcomes of this. The child process receives a value of 0 from the **fork()** operation, whereas the parent process receives the child's PID. To ensure synchronized process termination, the parent process uses the **wait()** system function to wait for the child process to end. Understanding process generation, unique process identifiers (PIDs), and synchronization-based inter-process communication are all aided by this assignment.

Assignment 2: Implementing a Simple Shell

Program Statement: Develop a simple shell program in C that can execute basic commands. The shell should:

1. Display a prompt for the user.
2. Read a command from the user input.
3. Execute the command using the **execvp()** system call.
4. Handle commands like **exit** to terminate the shell.

Solution Description: An operating system interface that enables users to communicate with it is called a shell. The program should repeatedly prompt the user for input, parse the command and arguments, and use **fork()** to create a new process to execute the command. The **execvp()** function will replace the child process's memory space with a new program. This assignment covers command parsing, process management, and the use of system calls to execute programs.

Assignment 3: Implementing a Multithreaded Program

Program Statement: Write a C program to create multiple threads using the **pthread** library. The program should:

1. Create at least three threads.
2. Each thread should print a unique message.

3. The main thread should wait for all threads to complete before exiting.

Solution Description: Threads are a way to achieve concurrency in a program. To start new threads, use the **pthread_create()** function; to wait for the threads to finish, use the **pthread_join()** method.Each thread will execute a separate function that prints a unique message. This assignment helps understand thread creation, execution, and synchronization in a multithreaded environment.

Assignment 4: Implementing Producer-Consumer Problem Using Semaphores

Program Statement: Implement the producer-consumer problem using semaphores in C. The program should:

1. Use two semaphores to synchronize access to a shared buffer.
2. The producer thread should add items to the buffer.
3. The consumer thread should remove items from the buffer.
4. Ensure mutual exclusion and avoid race conditions.

Solution Description:A typical synchronization challenge, the producer-consumer dilemma involves two different process types sharing a shared buffer: producers and consumers. Semaphores are used to manage access to the buffer, ensuring that only one process can modify the buffer at a time. The **sem_wait()** and **sem_post()** functions are used to decrement and increment the semaphore values, respectively. This assignment covers synchronization mechanisms and the prevention of race conditions in concurrent programming.

Assignment 5: Simulating a CPU Scheduling Algorithm

Program Statement: Write a C program to simulate the Round Robin CPU scheduling algorithm. The program should:

1. Accept the number of processes and their burst times.
2. Use a time quantum to allocate CPU time to each process in a round-robin fashion.
3. Display the order of execution and the completion time for each process.

Solution Description: Round Robin is a preemptive scheduling technique in which a predetermined time slot is allotted to each task in a cyclical sequence. The program will maintain a queue of processes, allocate the CPU to each process for the duration of the time quantum, and then move to the next process. If a process is not finished, it is re-added to the end of the queue. This assignment helps understand pre-emptive scheduling, time slicing, and process management.

Assignment 6: Simulating Virtual Memory with Paging

Program Statement: Create a C program to simulate virtual memory management using paging. The program should:

1. Accept a sequence of memory access requests.
2. Use a fixed-size page table to map virtual addresses to physical addresses.
3. Implement page replacement using the Least Recently Used (LRU) algorithm.
4. Display the page table and the number of page faults.

Solution Description:A memory management technique called paging makes it unnecessary to allocate physical memory in a contiguous manner. The program will simulate a page table that maps virtual addresses to physical frames. The LRU algorithm will be used to replace the least recently used page when a page fault occurs. This assignment covers virtual memory concepts, page tables, and page replacement algorithms.

Assignment 7: Simulating File System Operations

Program Statement: Write a C program to simulate basic file system operations. The program should:

1. Create a file.
2. Write data to the file.
3. Read data from the file.
4. Delete the file.

Solution Description: File system operations are crucial for managing data on a storage device. The program will use system calls such as **open()**, **write()**, **read()**, and **unlink()** to perform file operations. The **open()** system call creates a file, **write()** and **read()** handle data input/output, and **unlink()** deletes the file. This assignment helps in understanding file I/O operations and system call usage in Unix-like operating systems.

Assignment 8: Implementing a Memory Allocator

Program Statement: Develop a simple memory allocator in C using the **malloc()** and **free()** functions. The allocator should:

1. Allocate memory blocks of a specified size.
2. Maintain a free list of available memory blocks.
3. Reuse freed memory blocks for new allocations.

Solution Description: Memory allocation involves managing the allocation and deallocation of memory blocks. The program will use **malloc()** to allocate memory and maintain a free list to keep track of available blocks. The **free()** function will add freed blocks back to the free

list. This assignment covers dynamic memory management and the implementation of a custom memory allocator.

Assignment 9: Simulating Disk Scheduling Algorithms

Program Statement: Write a C program to simulate disk scheduling algorithms. The program should:

1. Implement FCFS (First-Come, First-Served) and SSTF (Shortest Seek Time First) algorithms.
2. Accept a sequence of disk access requests.
3. Calculate the total seek time for each algorithm.
4. Display the order of request servicing and the total seek time.

Solution Description: Disk scheduling algorithms manage the order in which disk access requests are serviced to minimize seek time. While SSTF chooses the request with the quickest seek time from the current head position, FCFS responds to requests in the order that they come in.The program will simulate both algorithms, process the sequence of requests, and calculate the total seek time for comparison. This assignment covers disk scheduling techniques and performance evaluation.

Assignment 10: Implementing a Simple Synchronization Mechanism Using Mutex

Program Statement: Create a C program to implement a simple synchronization mechanism using mutex locks. The program should:

1. Create multiple threads that access a shared resource.
2. Use a mutex to ensure mutual exclusion.
3. Each thread should increment a shared counter.
4. Display the final value of the counter.

Solution Description: Mutex locks provide a mechanism for ensuring that only one thread accesses a shared resource at a time. The **pthread_mutex_lock()** and **pthread_mutex_unlock()** procedures will be used by the application to lock and unlock the mutex, which will be created using the **pthread_mutex_t** type.Each thread will increment a shared counter, ensuring mutual exclusion to prevent race conditions. This assignment helps understand synchronization and mutual exclusion in multithreaded programs.

Assignment 11: Implementing a Banker's Algorithm for Prevention of Deadlock

Program Statement: Write a C program to implement the Banker's algorithm for prevention of deadlock. The program should:

1. Accept the number of processes and resource types.
2. Take the allocation and maximum demand matrices as input.
3. Determine if the system is in a safe state.
4. Display the safe sequence if it exists.

Solution Description: The Banker's algorithm is used to prevent deadlocks by ensuring that resource allocation does not lead to an unsafe state. The program will accept matrices representing current allocations and maximum demands, calculate the need matrix, and check for a safe sequence of process execution. If a safe sequence exists, the program will display it; otherwise, it will indicate that the system is in an unsafe state. This assignment covers deadlock avoidance techniques and resource allocation.

Assignment 12: Implementing a Page Replacement Algorithm

Program Statement: Develop a C program to implement the FIFO (First-In, First-Out) page replacement algorithm. The program should:

1. Accept a sequence of page requests.
2. Use a fixed-size frame buffer.
3. Replace pages using the FIFO algorithm.
4. Display the contents of the frame buffer after each request and the number of page faults.

Solution Description: The FIFO page replacement algorithm replaces the oldest page in the buffer when a page fault occurs. The program will maintain a queue to keep track of the order of pages in the frame buffer. Each new page request will be processed, and ifHere are more detailed lab assignments for an Operating Systems course, each with a program statement and solution description, focusing on common operating system concepts.

Assignment 13: Process Creation and Termination

Program Statement: Write a C program to demonstrate process creation and termination using the **fork()** system call. The program should:

1. Create a child process.
2. Print the process IDs of both the parent and child processes.
3. The child process should print a message and then terminate.
4. The parent process should wait for the child to terminate and then print a termination message.

Solution Description: By copying the caller process, the **fork()** system function creates a new process. The parent and the child processes are the outcomes of this. The **fork()** method

returns a value of 0 to the child process and returns the child's PID to the parent process. To ensure synchronized process termination, the parent process uses the **wait()** system function to wait for the child process to end.

This assignment helps in understanding process creation, unique process identifiers (PIDs), and inter-process communication through synchronization.

```
“#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
pid_tpid = fork();

   if (pid == 0) {
      // Child process
printf("Child process: PID = %d\n", getpid());
printf("Child process is terminating.\n");
   } else if (pid> 0) {
      // Parent process
printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
wait(NULL); // Wait for child process to terminate
printf("Parent process: Child has terminated.\n");
   } else {
      // Fork failed
printf("Fork failed!\n");
   }

return 0;
}
```

Assignment 14: Implementing a Simple Shell

Program Statement: Develop a simple shell program in C that can execute basic commands. The shell should:

1. Display a prompt for the user.
2. Read a command from the user input.
3. Execute the command using the **execvp()** system call.
4. Handle commands like **exit** to terminate the shell.

Solution Description: An operating system interface that enables users to communicate with it is called a shell.The program should repeatedly prompt the user for input, parse the command and arguments, and use **fork()** to create a new process to execute the command. The **execvp()** function will replace the child process's memory space with a new program. This assignment covers command parsing, process management, and the use of system calls to execute programs.

“#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#define MAX_LINE 80 // Maximum length of a command

int main() {

char *args[MAX_LINE / 2 + 1]; // Command line arguments

char input[MAX_LINE]; // User input

while (1) {

printf("osh> ");

fflush(stdout);

if (!fgets(input, MAX_LINE, stdin)) break;

input[strcspn(input, "\n")] = 0; // Remove newline character

```
if (strcmp(input, "exit") == 0) break;

pid_tpid = fork();
        if (pid == 0) {
            // Child process
char *token = strtok(input, " ");
int i = 0;
while (token != NULL) {
args[i++] = token;
token = strtok(NULL, " ");
            }
args[i] = NULL;

execvp(args[0], args);
perror("execvp failed");
exit(1);
        } else if (pid> 0) {
            // Parent process
wait(NULL);
        } else {
            // Fork failed
perror("fork failed");
        }
    }

return 0;
}
```

Assignment 15: Implementing a Multithreaded Program

Program Statement: Write a C program to create multiple threads using the **pthread** library. The program should:

1. Create at least three threads.
2. Each thread should print a unique message.
3. The main thread should wait for all threads to complete before exiting.

Solution Description: Software can accomplish concurrency through the use of threads. To start new threads, use the **pthread_create()** function; to wait for the threads to finish, use the **pthread_join()** method. Each thread will execute a separate function that prints a unique message. This assignment helps understand thread creation, execution, and synchronization in a multithreaded environment.

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

“

void *printMessage(void *threadid) {

longtid = (long)threadid;

printf("Thread %ld: Hello, World!\n", tid);

pthread_exit(NULL);

}


int main() {

pthread_t threads[3];

intrc;

long t;

   for (t = 0; t < 3; t++) {

printf("Creating thread %ld\n", t);

rc = pthread_create(&threads[t], NULL, printMessage, (void *)t);

      if (rc) {

printf("ERROR; return code from pthread_create() is %d\n", rc);

exit(-1);

      }
```

```
    }
    for (t = 0; t < 3; t++) {
pthread_join(threads[t], NULL);
    }
return 0;
}
```

Assignment 16: Implementing Producer-Consumer Problem Using Semaphores

Program Statement: Implement the producer-consumer problem using semaphores in C. The program should:

1. Use two semaphores to synchronize access to a shared buffer.
2. The producer thread should add items to the buffer.
3. The consumer thread should remove items from the buffer.
4. Ensure mutual exclusion and avoid race conditions.

Solution Description:A typical synchronization challenge, the producer-consumer dilemma involves two different process types sharing a shared buffer: producers and consumers. Semaphores are used to manage access to the buffer, ensuring that only one process can modify the buffer at a time. The **sem_wait()** and **sem_post()** functions are used to decrement and increment the semaphore values, respectively. This assignment covers synchronization mechanisms and the prevention of race conditions in concurrent programming.

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

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int count = 0;
sem_t empty, full, mutex;

void *producer(void *param) {
int item;
```

```
    for (int i = 0; i < 20; i++) {
item = rand() % 100; // Produce an item
sem_wait(&empty);
sem_wait(&mutex);

        buffer[count++] = item; // Add item to the buffer
printf("Producer produced %d\n", item);

sem_post(&mutex);
sem_post(&full);
    }
pthread_exit(NULL);
}

void *consumer(void *param) {
int item;
    for (int i = 0; i < 20; i++) {
sem_wait(&full);
sem_wait(&mutex);

item = buffer[--count]; // Remove item from the buffer
printf("Consumer consumed %d\n", item);

sem_post(&mutex);
sem_post(&empty);
    }
pthread_exit(NULL);
}

int main() {
```

```
pthread_t prod, cons;

sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);
sem_init(&mutex, 0, 1);

pthread_create(&prod, NULL, producer, NULL);
pthread_create(&cons, NULL, consumer, NULL);

pthread_join(prod, NULL);
pthread_join(cons, NULL);

sem_destroy(&empty);
sem_destroy(&full);
sem_destroy(&mutex);

return 0;}
```