OPERATING SYSTEMS LAB

Code: CS 1313

1. Write programs using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

2. Write programs using the I/O System calls of UNIX operating system (open, read,
write, etc).

3. Write C programs to simulate UNIX commands like ls, grep, etc.

4. Given the list of processes, their CPU burst times and arrival times. Display/print the Gantt chart for FCFS and SJF. For each of the scheduling policies, compute and print the average waiting time and average turnaround time.

5. Given the list of processes, their CPU burst times and arrival times. Display/print the Gantt chart for Priority and Round robin. For each of the scheduling policies, compute and print the average waiting time and average turnaround time.

6. Develop application using Inter-Process Communication (using shared memory, pipes or message queues).

7. Implement the Producer-Consumer problem using semaphores (using UNIX system calls).

8. Implement Memory management schemes like paging and segmentation.

9. Implement Memory management schemes like First fit, Best fit and Worst fit.

10. Implement any file allocation techniques (Contiguous, Linked or Indexed).

<u>**INDEX**</u>

Exp#	**Name**	**Aim of the experiment**
Process System Calls		
1(a)	**fork** system call	To create a new child process using fork system call.
1(b)	**wait** system call	To block a parent process until the child completes using wait system call.
1(c)	**exec** system call	To load an executable program in a child processes exec system call.
1(d)	**stat** system call	To display file status using stat system call.
1(e)	**readdir** system call	To display directory contents using readdir system call

PROCESS SYSTEM CALL

fork()

➢ The fork system call is used to create a new process called child process.
- The return value is 0 for a child process.
- The return value is negative if process creation is unsuccessful.
- For the parent process, return value is positive.

➢ The child process is an exact copy of the parent process.
➢ Both the child and parent continue to execute the instructions following fork call.
➢ The child can start execution before the parent or vice-versa.

getpid() and getppid()

➢ The getpid system call returns process ID of the calling process

- ➢ The getppid system call returns parent process ID of the calling process

wait()

- ➢ The wait system call causes the parent process to be blocked until a child terminates.
- ➢ When a process terminates, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- ➢ Without wait, the parent may finish first leaving a zombie child, to be adopted by init process

execl()

- ➢ The exec family of functions (execl, execv, execle, execve, execlp, execvp) is used by the child process to load a program and execute.
- ➢ execl system call requires path, program name and null pointer

exit()

- ➢ The exit system call is used to terminate a process either normally or abnormally
- ➢ Closes all standard I/O streams.

stat()

- ➢ The stat system call is used to return information about a file as a structure.

opendir(), readdir() and closedir()

- ➢ The opendir system call is used to open a directory
    - It returns a pointer to the first entry
    - It returns NULL on error.
- ➢ The readdir system call is used to read a directory as a dirent structure
    - It returns a pointer pointing to the next entry in directory stream
    - It returns NULL if an error or end-of-file occurs.
- ➢ The closedir system call is used to close the directory stream
- ➢ Write to a directory is done only by the kernel.

Exp#	Name	Aim of the experiment
I/O System Calls		
2(a)	**creat** system call	To create a file and to write contents.
2(b)	**read** system call	To read the given file and to display file contents.
2(c)	**write** system call	To append content to an existing file.

FILE SYSTEM CALL

open()

- ➢ Used to open an existing file for reading/writing or to create a new file.
- ➢ Returns a file descriptor whose value is negative on error.
- ➢ The mandatory flags are O_RDONLY, O_WRONLY and O_RDWR
- ➢ Optional flags include O_APPEND, O_CREAT, O_TRUNC, etc
- ➢ The flags are ORed.
- ➢ The mode specifies permissions for the file.

creat()

- ➢ Used to create a new file and open it for writing.
- ➢ It is replaced with open() with flags O_WRONLY|O_CREAT | O_TRUNC

read()

- ➢ Reads no. of bytes from the file or from the terminal.
- ➢ If read is successful, it returns no. of bytes read.
- ➢ The file offset is incremented by no. of bytes read.
- ➢ If end-of-file is encountered, it returns 0.

write()

- ➢ Writes no. of bytes onto the file.
- ➢ After a successful write, file's offset is incremented by the no. of bytes written.
- ➢ If any error due to insufficient storage space, write fails.

close()

- ➢ Closes an opened file.
- ➢ When process terminates, files associated with the process are automatically closed.

Exp#	Name	Aim of the experiment
Command Simulation		
3(a)	**ls** command	To simulate ls command using UNIX system calls.
3(b)	**grep** command	To simulate grep command using UNIX system call.
3(c)	**cp** command	To simulate cp command using UNIX system call.
3(d)	**rm** command	To simulate rm command using UNIX system call.

COMMAND SIMULATION

- ➢ Using UNIX system calls, most commands can be emulated in a similar manner.
- ➢ Simulating a command and all of its options is an exhaustive exercise.
- ➢ Command simulation harnesses one's programming skills.
- ➢ Command simulation helps in the development of standard routines to be customized to the application needs.
- ➢ Generally file I/O commands are simulated.

Exp#	Name	Aim of the experiment
Process Scheduling		
4(a)	**FCFS** scheduling	To schedule snapshot of processes queued according to FCFS (First Come First Serve) scheduling.
4(b)	**SJF** scheduling	To schedule snapshot of processes queued according to SJF (Shortest Job First) scheduling.
4(c)	**Priority** scheduling	To schedule snapshot of processes queued according to Priority scheduling.
4(d)	**Round Robin** scheduling	To schedule snapshot of processes queued according to Round robin scheduling.

Process Scheduling

- CPU scheduling is used in multiprogrammed operating systems.
- By switching the CPU among processes, efficiency of the system can be improved.
- Some scheduling algorithms are FCFS, SJF, Priority, Round-Robin, etc.
- Gantt chart provides a way of visualizing CPU scheduling and enables to understand better.

First Come First Serve (FCFS)

- Process that comes first is processed first
- FCFS scheduling is non-preemptive
- Not efficient as it results in long average waiting time.
- Can result in starvation, if processes at the beginning of the queue have long bursts.

Shortest Job First (SJF)

- Process that requires smallest burst time is processed first.
- SJF can be preemptive or non–preemptive
- When two processes require the same amount of CPU utilization, FCFS is used to break the tie.
- Generally efficient as it results in minimal average waiting time.

- ➢ Can result in starvation, since long critical processes may not be processed.

Priority

- ➢ Process that has higher priority is processed first.
- ➢ Priority can be preemptive or non–preemptive
- ➢ When two processes have the same priority, FCFS is used to break the tie.
- ➢ Can result in starvation, since low priority processes may not be processed.

Round Robin

- ➢ All processes are processed one by one as they have arrived, but in rounds.
- ➢ Each process cannot take more than the time slice per round.
- ➢ Round robin is a fair preemptive scheduling algorithm.
- ➢ A process that is yet to complete in a round is preempted after the time slice and put at the end of the queue.
- ➢ When a process is completely processed, it is removed from the queue.

Exp#	Name	Aim of the experiment
Inter-process Communication		
5(a)	Fibonacci & Prime number	To generate 25 Fibonacci numbers and determine prime amongst them using pipe.
5(b)	**who|wc -l**	To determine the number of users logged in using pipe.
5(c)	Chat Messaging	To exchange messages between server and client using message queue.
5(d)	Shared memory	To demonstrate communication between processes using shared memory.
5(e)	Producer-Consumer problem	To synchronize producer and consumer processes using semaphore.

Inter-process Communication

➢ Inter-Process communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange data.

➢ IPC in linux can be implemented using pipe, shared memory, message queue, semaphore, signal or sockets.

Pipe

➢ Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process.
➢ A pipe is created using the system call pipe that returns a pair of file descriptors.
➢ The descriptor pfd[0] is used for reading and pfd[1] is used for writing.
➢ Can be used only between parent and child processes.

Shared memory

➢ Two or more processes share a single chunk of memory to communicate randomly.
➢ Semaphores are generally used to avoid race condition amongst processes.
➢ Fastest amongst all IPCs as it does not require any system call.
➢ It avoids copying data unnecessarily.

Message Queue

➢ A message queue is a linked list of messages stored within the kernel
➢ A message queue is identified by a unique identifier
➢ Every message has a positive long integer type field, a non-negative length, and the actual data bytes.
➢ The messages need not be fetched on FCFS basis. It could be based on type field.

Semaphores

➢ A semaphore is a counter used to synchronize access to a shared data amongst multiple processes.
➢ To obtain a shared resource, the process should:

- Test the semaphore that controls the resource.
- If value is positive, it gains access and decrements value of semaphore.
- If value is zero, the process goes to sleep and awakes when value is > 0.

When a process relinquishes resources, it increments the value of the semaphore by 1.

Producer-Consumer problem

➢ A producer process produces information to be consumed by a consumer process
➢ A producer can produce one item while the consumer is consuming another one.
➢ With bounded-buffer size, consumer must wait if buffer is empty, whereas producer must wait if the buffer is full.

➢ The buffer can be implemented using any IPC facility.

Exp#	Name	Aim of the experiment
Memory Management		
6(a)	First Fit	To allocate memory requirements for processes using first fit allocation.
6(b)	Best Fit	To allocate memory requirements for processes using best fit allocation.
6(c)	**FIFO** Page Replacement	To implement demand paging for a reference string using FIFO method.
6(d)	**LRU** Page Replacement	To implement demand paging for a reference string using LRU method.

Memory Management

➢ The first-fit, best-fit, or worst-fit strategy is used to select a free hole from the set
of available holes.

First fit

➢ Allocate the first hole that is big enough.

➢ Searching starts from the beginning of set of holes.

Best fit

➢ Allocate the smallest hole that is big enough.

➢ The list of free holes is kept sorted according to size in ascending order.

➢ This strategy produces smallest leftover holes

Worst fit

➢ Allocate the largest hole.

➢ The list of free holes is kept sorted according to size in descending order.

➢ This strategy produces the largest leftover hole.

The widely used page replacement algorithms are FIFO and LRU.

FIFO

- ➢ Page replacement is based on when the page was brought into memory.

When a page should be replaced, the oldest one is chosen.

- ➢ Generally, implemented using a FIFO queue.
- ➢ Simple to implement, but not efficient.
- ➢ Results in more page faults.
- ➢ The page-fault may increase, even if the frame size is increased (Belady's anomaly)

LRU

- ➢ Pages used in the recent past are used as an approximation of future usage.
- ➢ The page that has not been used for a longer period of time is replaced.
- ➢ LRU is efficient but not optimal.
- ➢ Implementation of LRU requires hardware support, such as counters/stack.

Exp#	**Name**	**Aim of the experiment**
File Allocation		
7	Contiguous	To implement file allocation on free disk space in a contiguous manner.

File Allocation

The three methods of allocating disk space are:

1. Contiguous allocation
2. Linked allocation
3. Indexed allocation

Contiguous

- ➢ Each file occupies a set of contiguous block on the disk.
- ➢ The number of disk seeks required is minimal.
- ➢ The directory contains address of starting block and number of contiguous

block (length) occupied.

- ➢ Supports both sequential and direct access.
- ➢ First / best fit is commonly used for selecting a hole.

Linked

- ➢ Each file is a linked list of disk blocks.
- ➢ The directory contains a pointer to first and last blocks of the file.
- ➢ The first block contains a pointer to the second one, second to third and so on.
- ➢ File size need not be known in advance, as in contiguous allocation.
- ➢ No external fragmentation.
- ➢ Supports sequential access only.

Indexed

➢ In indexed allocation, all pointers are put in a single block known as index Block.

- ➢ The directory contains address of the index block.
- ➢ The i'th entry in the index block points to i'th block of the file.
- ➢ Indexed allocation supports direct access.
- ➢ It suffers from pointer overhead, i.e wastage of space in storing pointers.