9/2/24

COMPREHENSIVE STUDY ON BASIC LINUX COMMANDS AND SYSTEM COMPONENTS.

AIM:

The aim of this study is to delve into the fundamentals of Linux, focusing on basic commands, the proc file system, disk I/O operations, buffer caches, and disk monitoring tools, to equip participants with essential knowledge for system administration and troubleshooting.

THEORY:-

1. Basic Linux Commands:

- Navigation commands (cd, ls, pwd) for directory traversal and orientation.
- File manipulation commands (cp, mv, rm) for managing files and directories.
- Permissions commands (chmod) to regulate access rights to files and directories.
- Text processing commands (cat, grep) for efficient data extraction and manipulation.
- Package management commands (apt, pacman) for software installation and updates.

5. Disk Monitoring Tools:

- Overview of disk monitoring tools such as vmstat, htop.
- Hands on demonstration of using disk monitoring tools for real-time performance analysis.
- Interpretation of tools outputs to identify disk-related issues and optimize system resources.

2. Proc file system:

- Exploration of the /proc directory structure and its role in providing system information.
- Examination of various files within /proc for insights into system processes, hardware, and kernel parameters.
- Practical examples illustrating how to interpret /proc files for system monitoring and troubleshooting.

3. Disk I/O operations:

- Definition and significance of disk I/O in system performance.
- Introduction to commands such as iostat for monitoring disk I/O activities.
- Analysis of disk I/O patterns, throughput and latency to identify performance bottlenecks.

4. Buffer Caches:

- Overview of buffer caches and their role in optimizing disk I/O operations.

12
10/24

IMPLEMENTATION OF SHELL PROGRAMING.

AIM:

To write simple shell programs by using conditional, branching and looping statements.

THEORY:

Shell programming, often referred to as shell scripting, involves writing scripts or programs that are interpreted by a shell which is a command-line interpreter for operating systems. In unix-like operating systems (such as Linux, macOS), the shell is the primary interface between the user and the operating system kernel. The shell interprets commands entered by the user or scripts written by the user and executes them. It uses the extension ".sh".

PROCEDURE:

Step 1: Open LINUX terminal.

Step 2: Create a bash program using "NANO" text editor in the terminal.

Step 3: Write the bash script and save it with the extension ".sh".

Step 4: Upscale the permission using $ chmod +x (filename).sh

Step 5 : Run the file $ sh (filename).sh

ALTERNATE PROCEDURE :

Step 1 : Write the code in an online bash compiler.

Step 2 : Execute it!

program for system calls of UNIX operating system (fork, getpid, exit):-

PROCEDURE:

Step 1: Start the program

Step 2: Declare the variables pid, pid1, pid2.

Step 3: Call fork() system call to create process.

Step 4: If pid == -1, exit.

Step 5: If pid != -1, get the process id using getpid().

Step 6: Print the process id.

Step 7: Stop the program.

Ex. No: 3 DATE: 9/3/24	IMPLEMENTATION OF UNIX SYSTEM CALLS.

AIM:

To write c programs using the following system calls of UNIX operating system fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

Program for system calls of UNIX operating systems (OPENDIR, READDIR, CLOSEDIR):-

PROCEDURE:

Step 1: Start the program.

Step 2: Create struct dirent.

Step 3: declare the variable buff and pointer dptr.

Step 4: Get the directory name.

Step 5: open the directory

Step 6: Read the contents in directory and print it.

Step 7: Close the directory.

EX.NO: 4

DATE: 13/3/24

CPU SCHEDULING ALGORITHMS

FIRST COME FIRST SERVE (FCFS).

AIM:

To write a C program for implementation of FCFS and SJF scheduling algorithms.

PROCEDURE:

Step 1: Inside the structure declare the variables.

Step 2: Declare the variable i,j as integer, tototime and tottime is equal to zero.

Step 3: Get the value of 'n' assign pid as I and get the value of p[i].btime.

Step 4: Assign p[0] wtime as zero and tottime as btime and inside the loop calculate wait time and turnaround time.

Step 5: Calculate total wait time and total turnaround time by dividing by total number of process.

Step 6: Print total wait time and total turnaround time.

Step 7: Stop the program.

Ex.No: 4 DATE: 13/3/24	IMPLEMENTATION OF NON-PREEMPTIVE AND PREEMPTIVE CPU SCHEDULING ALGORITHMS.

PRIORITY.

AIM:

To write a c program for implementation of priority scheduling algorithms.

PROCEDURE:

Step 1: Inside the structure declare the variables.

Step 2: Declare the variable i, j as integer, totwtime and tottime is equal to zero.

Step 3: Get the value of 'n' assign p and allocate the memory.

Step 4: Inside the for loop get the value of burst time and priority.

Step 5: Assign witime as zero.

Step 6: Check p[i].pri is greater than p[j].pri.

Step 7: Calculate the total of burst time and waiting time and assign as turnaround time.

Step 8: Stop the program.

EX.NO: 4
DATE: 13/3/24

CPU SCHEDULING ALGORITHMS
ROUND ROBIN SCHEDULING.

AIM:

To write a C program for implementation of Round Robin scheduling algorithms.

PROCEDURE:

Step 1: Inside the structure declare the variables.

Step 2: Declare the variable i, j as integer, totwtime and tottime is equal to zero.

Step 3: Get the value of 'n' assign p and allocate the memory.

Step 4: Inside the for loop get the value of burst time and priority and read the time quantum.

Step 5: Assign wtime as zero.

Step 6: Check P[i].pri is greater than P[j].pri.

Step 7: Calculate the total of burst time and waiting time and assign as turnaround time.

Step 8: Stop the program.

EX.NO: 4 DATE: 13/3/24	CPU SCHEDULING ALGORITHMS SHORTEST JOB FIRST (SJF) SCHEDULING.

AIM:

To write a c program for implementation of SJF scheduling algorithms.

PROCEDURE:

step 1: Inside the structure declare the variables.

step 2: Declare the variable i, j as integer, totwtime and tottime is equal to zero.

step 3: Get the value of 'n' assign pid as I and get the value of p[i].btime.

step 4: Assign p[0] wtime as zero and tot time as btime and inside the loop calculate wait time and turnaround time.

step 5: Calculate total wait time and total turnaround time by dividing by total number of process.

step 6: Print total wait time and total turnaround time.

step 7: Stop the program.

EX.NO: 5
DATE: 15/3/24

IMPLEMENTATION OF DINING PHILOSOPHER'S PROBLEM TO DEMONSTRATE PROCESS SYNCHRONISATION

AIM:

To write a c program to implement Dining philosopher's problem to demonstrate process synchronisation.

PROCEDURE:

Step 1: Start the program.

Step 2: Initialize the semaphores for each fork to 1

Step 3: Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.

Step 4: Create separate threads to attempt to acquire the semaphore for fork to the left and right.

Step 5: Stop the program.

NO: 6

E: 19/8/24

BANKERS ALGORITHM FOR DEADLOCK AVOIDANCE.

M:

To write a C program to implement banker's gorithm for deadlock avoidance.

OCEDURE:

ep 1: Start the program.

ep 2: Declare the memory for the process.

ep 3: Read the number of process, resources, location matrix and available matrix.

ep 4: Compare each and every process using the nkers algorithm.

ep 5: If the process is in safe state then it is t a deadlock process otherwise it is a adlock process.

ep 6: produce the result of state of process.

ep 7: Stop the program.

Ex. No: 7 MEMORY ALLOCATION METHODS FOR FIXED PARITION - FIRST FIT

DATE:

AIM:

To write a C++ program for implementation memory allocation methods for fixed partition using first fit.

PROCEDURE:

Step 1: Define the max as 25.

Step 2: Declare the variable frag[max], b[max], f[max], i, j, nb, nf, temp, highest = 0, bf[max], ff[max].

Step 3: Get the number of blocks, files, size of the blocks using for loop.

Step 4: In for loop check bf[j]! = 1, if so temp = b[j] - f[i]

Step 5: Check highest < temp, if so assign ff[i] = j, highest = temp.

Step 6: Assign frag[i] = highest, bf[ff[i]] = 1, highest = 0.

Step 7: Repeat step 4 to step 6.

Step 8: Print file no, size, block no, size and fragment.

Step 9: Stop the program.

EX. NO: 7	MEMORY ALLOCATION METHODS FOR FIXED
DATE :	PARTITION - WORST FIT.

AIM:

To write a C++ program for implementation of memory allocation methods for fixed partition using worst-fit.

PROCEDURE:

Step 1: Define the max as 25.

Step 2: Declare the variable frag[max], f[max], i, j, nb, nf, temp, highest = 0, bf [max], ff[max].

Step 3: Get the number of blocks, files, size of the blocks using for loop.

Step 4: In for loop check bf [j]!=1, if so temp=b[j]-f[i].

Step 5: check temp>=0, if so assign ff[i]=j break the for loop.

Step 6: Assign frag [i]=temp, bf[ff[i]]=1;

Step 7: Repeat step 4 to step6.

Step 8: Print file no, size, block no, size and fragment.

Step 9: Stop the program.

EX. NO: 7	MEMORY ALLOCATION METHODS FOR FIXED
DATE :	PARTITION - BEST FIT.

AIM:

To write a C++ program for implementation of memory allocation methods for fixed partition using best fit.

PROCEDURE:

step 1: Define the max as 25.

step 2: Declare the variable frag [max], b [max], f[max], i, j, nb, nf, temp, highest=0, bf[max], ff [max].

Step 3: Get the number of blocks, files, size of the blocks using for loop.

Step 4: In for loop check bf [j] != 1, if so temp=b[j]-f[i].

step 5: Check lowest > temp, if so assign ff[i] = j, highest = temp.

step 6: Assign frag[i] = lowest, bf [ff[i]] = 1, lowest = 10000

Step 7: Repeat step 4 to step 6.

step 8: Print file no, size, block no, size, and fragment.

step 9: Stop the program.