**ST. FRANCIS INSTITUE OF TECHNOLOGY**

**MT. POINSUR, BORIVALI (W), MUMBAI**



**LAB MANUAL**

**ON**

**OPERATING SYSTEM LAB**

**SE-CMPN-A & B / IV**

**Academic Year: 2020-2021**

**Prepared By:**

**Ms. Nidhi Gaur**

**Asst. Professor in CMPN Department**

**Experiment No. 1**

**Aim:** Write a program to implement FCFS and SJF CPU Scheduling algorithm

**Theory:** For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

**Algorithm:**

1. Start

2. Read the number of processes to be inserted

3. Read the Burst times and Arrival time of processes

4. Calculate start time and finish time of every process

5. Calculate the waiting time of each process

wt[i]=start[i]-arr[i];

6. Calculate the turnaround time of each process

tat[i]=finish[i]-arr[i];

7. Calculate the average waiting time and average turnaround time.

8. Display the values

9. Stop

**Example:**

Process Burst Time

P1 24

P2 03

P3 03

Suppose that the processes arrive in the order: P1 , P2 , P3

The Gantt Chart for the schedule is:



Waiting time for P1 = 0; P2 = 24; P3 = 27

**Average waiting time: (0 + 24 + 27)/3 = 17**

**Theory:** For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

**Algorithm:**

1. Start

2. Read the number of processes to be inserted

3. Read the Burst times and Arrival time of processes

4. Arrange processes in ascending order w.r.t. their burst time

5. Calculate waiting time and turnaround time of each process

6. Calculate the average waiting time and average turnaround time.

7. Display the values

8. Stop

**SOLVE Example:**

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| *P1* | 7 |
| *P2* | 4 |
| *P3* | 1 |
| *P4* | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| P3 | P2 | P4 | P1 |

0 1 5 9 16

**Average Waiting Time: 3.75**

**CODE:**

**OUTPUT:**

**Viva questions:**

**Q1: What is drawback of FCFS algorithm?**

**Q2: SJF eliminates drawback of FCFS. Is it true or false? Solve one numerical justifying your answer.**

**CONCLUSION:**

**Experiment : 02**

AIM :- To write a program to implement the ROUND ROBIN Scheduling using C

**PROBLEM DESCRIPTION:**

In this algorithm we are assigning some time slice .The process is allocated according to the time slice, if the process service time is less than the time slice then process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than time quantum, the timer will go off and will cause an interrupt to the operating system .A context switch will be executed and the process will be put at the tail of the ready queue.

**ALGORITHM:**

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the burst times of the processes
5. Read the Time Quantum
6. If the burst time of a process is greater than time Quantum then subtract time quantum form the burst time Else Assign the burst time to time quantum.
7. Calculate the average waiting time and turnaround time of the processes.
8. Display the values
9. Stop

**Example Of Round Robin Scheduling**

|  |  |  |
| --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** |
| P0 | 0 | 9 |
| P1 | 1 | 5 |
| P2 | 2 | 3 |
| P4 | 3 | 4 |

Calculate Avg TAT and Avg WT with Time Quantum =2 ms

Code:

Output:

Conclusion

**CONCLUSION: 0-5** Hence, we see that Round Robin Scheduling Algorithm is very fair and no starvation occurs.

**Viva question:**

1. **Solve one numerical using Round Robin**
2. **What is the effect of time slice on context switching?**

**Experiment No. 3**

**Aim**: Implementation of Banker’s Algorithm

**Theory:** The **Banker's algorithm** is a resource allocation and deadlock avoidance algorithm developed by [Edsger Dijkstra](http://en.wikipedia.org/wiki/Edsger_Dijkstra) that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

**Algorithm: Data Structures for the Banker’s Algorithm**

1. Let *n* = number of processes, and *m* = number of resources types.
2. **Available***:* Vector of length *m*.

If available [*j*] = *k*, there are *k* instances of resource type *Rj*available

1. **Max***: n x m* matrix.

If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*

1. **Allocation***: n* x *m* matrix.

If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj*

1. **Need***: n* x *m* matrix.

If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj*to complete its task  
 *Need* [*i,j]* = *Max*[*i,j*] – *Allocation* [*i,j*]

**Safety algorithm**

Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

1. *Work* = *Available*

*Finish* [*i*] = *false* for *i* = 0, 1, …, *n-* 1

2. Find an *i* such that both:

(a) *Finish* [*i*] = *false*

(b) *Needi* ? *Work*

If no such *i* exists, go to step 4

*3. Work* = *Work* + *Allocationi*  
 *Finish*[*i*] = *true*  
 go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Code:

Output:

Conclusion

Viva questions:

Solve one numerical of banker’s algorithm

**Experiment No. 4**

**Aim**: Implementation of Memory allocation algorithm (Best Fit, First Fit, Worst Fit)

**Theory:** One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

**Algorithm:**

**First Fit**

1. Input memory blocks with size and processes with size.

2- Initialize all memory blocks as free.

3- Start by picking each process and check if it can be assigned to current block.

4- If size-of-process <= size-of-block if yes then assign and check for next process.

5- If not then keep checking the further blocks.

**Best Fit**

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

3- Start by picking each process and find the minimum block size that can be assigned to

current process

i.e., find min(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current],

if found then assign it to the current process.

4- If not then leave that process and keep checkingthe further processes.

**Worst Fit**

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

3- Start by picking each process and find the lsrgest block size that can be assigned to

current process

i.e., find max(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current],

if found then assign it to the current process.

4- If not then leave that process and keep checking the further processes.

Code:

Output:

Conclusion

**Viva questions:**

**Solve one numerical related to memory allocation techniques using diagrams.**

**Experiment No.5**

**Aim**: Implementation of Page Replacement Algorithm( FIFO, OPT and LRU)

**Theory:** Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme.

A **FIFO replacement algorithm** associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.

b) Simultaneously maintain the pages in the queue to perform FIFO.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Remove the first page from the queue as it was the first to be entered in the memory

b) Replace the first page in the queue with the current page in the string.

c) Store current page in the queue.

d) Increment page faults.

2. Return page faults.

In **Optimal Page replacement** algorithm, pages are replaced which are not used for the longest duration of time in the future.

1. Start the program

2. Read the number of frames

3. Read the number of pages

4. Read the page numbers

5. Initialize the values in frames to -1

6. Allocate the pages in to frames by selecting the page that will not be used for the longest period of time.

7. Display the number of page faults.

8. Stop the program

**LRU replacement** associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.

b) Simultaneously maintain the recent occurred index of each page in a map called indexes.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Find the page in the set that was least recently used. We find it using index array.

We basically need to replace the page with minimum index.

b) Replace the found page with current page.

c) Increment page faults.

d) Update index of current page.

2. Return page faults.

Code:

Output:

Conclusion

**Viva questions:**

**Solve one numerical related to paging techniques.**

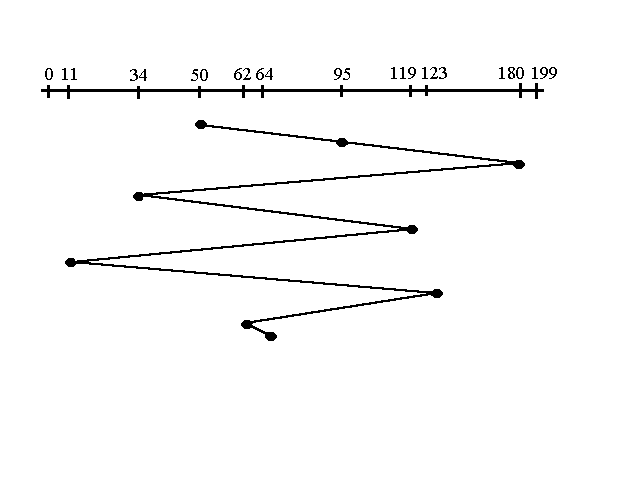
**Experiment No. 6**

**Aim:** To implement I/O scheduling algorithms (FCFS and SCAN).

**Theory:** In operating systems, seek time is very important. Since all device requests are linked in queues, the seek time is increased causing the system to slow down. Disk Scheduling Algorithms are used to reduce the total seek time of any request. There are following disk scheduling algorithms:  
First Come-First Serve (FCFS)  
Shortest Seek Time First (SSTF)  
Elevator (SCAN)   
Circular SCAN (C-SCAN)  
LOOK  
C-LOOK

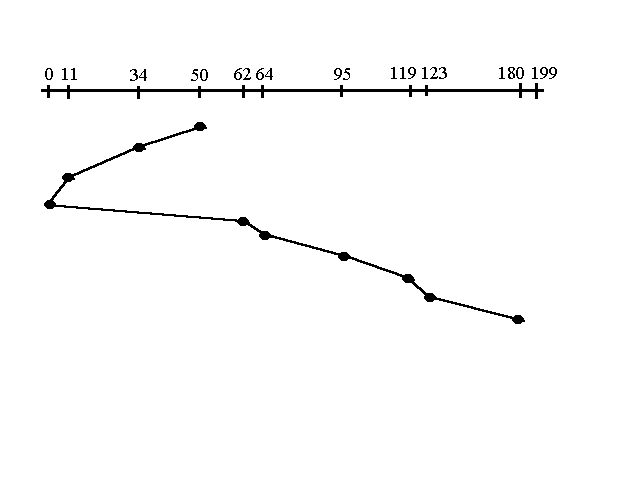
Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.

**1. First Come -First Serve (FCFS):** All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served.



Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.

**2. Elevator (SCAN):** This approach works like an elevator does. It scans down towards the nearest end and then when it hits the bottom it scans up servicing the requests that it didn't get going down. If a request comes in after it has been scanned it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks. Once again this is more optimal than the previous algorithm, but it is not the best.



**Algorithm**

**1. FCFS**

Pseudo code for disk head movement: Move the head to the next requested cylinder in the queue.

No queue sorting is necessary.

**2. SCAN:** In simple scan scheduling, the disk head is only ever moving outwards.

1. Move the disk head outwards to the next request above it and service it

If the disk head is at the edge of the disk Reset the disk head to the track zero

When sorting the queue for the scan algorithm, it must be ordered so that the request in front of the head is first in the list, and the request that is closest behind the head position is the last. Pseudo code for queue sorting:

1. Split the access queue into two parts, one above the disk head, one below

2. Sort the lists according to distance from the disk head

In descending order above the disk head

In ascending order below it

3. Place the list from above the disk head back into the request queue

4. Place the list from below the disk head back into the request queue

Code:

Output:

Conclusion

**Viva questions:**

**Solve one numerical of FCFS and SCAN disk scheduling techniques.**

**Experiment No. 7**

**Exp No.1.a**: Explore the internal commands of Linux like ls, chdir, mkdir, chown, chmod, chgrp, ps

**Theory**: Linux is an Operating System’s Kernel. It was created by Linus Torvalds from Scratch. Linux is free and open-source, that means that you can simply change anything in Linux and redistribute it in your own name! There are several Linux Distributions, commonly called “distros”. A few of them are:

|  |  |
| --- | --- |
| Ubuntu Linux | Red Hat Enterprise Linux |
| Linux Mint | Debian |
| Fedora |  |

**Linux Shell or “Terminal”**

Shell is a program that receives commands from the user and gives it to the OS to process and it shows the output. Linux’s shell is its main part. Its distros come in GUI (Graphical User Interface) but basically, Linux has a CLI (Command Line Interface)

**Linux Commands**

**Basic Commands**

**ls -** The most basic feature of the shell is the ability to see what files are available on the system. The list command (ls) is the tool that helps do that. It displays the files and directories located in your current directory. It produces listing in alphabetical order.

|  |  |
| --- | --- |
| **–l** | It produces long listing of files and folders |
| **–n** | Its output is same as “-l option” except username is replaced by UID and group name is replaced by GID. |
| **–R** | It will list sub-directory components recursively. |
| **–i** | It displays inode number of each file and folder |
| **–r** | It displays the output in reverse order |
| **–F** | This option is used to easily distinguish files from directories. It flags the directories with a forward slash, to help identify them in the listing. Similarly, it flags executable files (like the myprog file below) with an asterisk, to help you find the files that can be run on the system easier. |

**OPTIONS of *‘ls’* command**

1. **cd -** This command is used to change directory. It allows us to move from one directory to another**.**

**Syntax: cd destination\_folder\_name**

1. **mkdir -** This command is used to create a directory. It is followed by the name of the directory to be created.

**Syntax:** $ mkdir dir\_name

**Example:** $ mkdir OS

The above command creates a directory named “OS” under your current working directory.

1. **chmod -** stands for "change mode", and it is used to define the way a file can be accessed.

**Syntax:** chmod permissions filename

Permissions defines the permissions for the owner of the file (the "**user**"), members of the group who owns the file (the "**group**"), and anyone else ("**others**"). There are two ways to represent these permissions: with symbols (alphanumeric characters), or with octal numbers (the digits 0 through 7).

1. **chown –** The chown command **changes ownership** of files and directories in a **Linux** [**file system**](https://www.computerhope.com/jargon/f/filesyst.htm)**.** You can specify either the login name or the numeric UID for the new owner of the file:

# chown dan newfile

# ls -l newfile

-rw-rw-r-- 1 dan rich 0 Sep 20 19:16 newfile\*

#

The chown command also allows you to change both the user and group of a file:

# chown dan.dan newfile

# ls -l newfile

-rw-rw-r-- 1 dan dan 0 Sep 20 19:16 newfile\*

#

1. **chgrp –** The chgrp command provides an easy way to change just the default group for a file or directory:

$ chgrp shared newfile

$ ls -l newfile

-rw-rw-r-- 1 rich shared 0 Sep 20 19:16 newfile\*

1. **ps -** The ps command produces a list of the currently running processes on your computer.

To invoke ps simply type the following: ps

The output will show rows of data containing the following information:

**PID** : The PID is the process ID which identifies the running process.

**TTY** : The TTY is the terminal type.

**Time and Command**

To view all the running processes use either of the following commands: ps -A , ps -e

**Exp No.1.b.** Explore the internal commands of Linux like cat, touch, ln, who, who am I, cal, date, cut, cp, mv, rm.

**Theory:**

1. **cat command:** This command is useful for creating, displaying, appending information and concatenating two or more files.

**Syntax:- $ cat > filename --- Creating file**

**$ cat filename --- Displaying the contents of file**

**$ cat >> filename -- Appending information to file**

**$ cat file1 file2 > Concat\_file ----- Concatenation**

1. **touch command:** The touch command is used to create an empty file.

Syntax:- touch filename

Example:- touch Sunday

The above command creates the new file “Sunday” and assigns your username as file owner.

The touch command can also be used to change the access and modification times on an existing file without changing the file contents. If you want to change only the access time, use the -a parameter. To change only the modification time, use the –mparameter.By default touch uses the current time. You can specify the time by using the –t parameter with a specific timestamp:

Syntax:- $ touch -t 200812251200 test1

Output:-

$ ls -l test1

-rw-r--r-- 1 rich rich 0 Dec 25 2008 test1

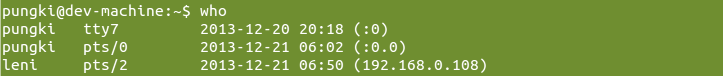
1. **ln command:** ln creates a link to file TARGET with the name LINKNAME. If LINKNAME is omitted, a link to TARGET is created in the current directory, using the name of TARGET as the LINKNAME.

ln creates hard links by default, or symbolic links if the -s (--symbolic) option is specified. When creating hard links, each TARGET must exist.

1. **who command:** Linux maintains an account of all users who are currently logged in to the system.

**Syntax:- $** who

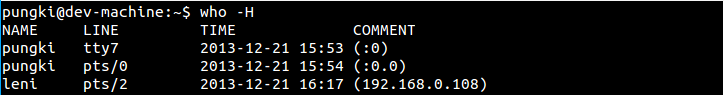
**Output:**



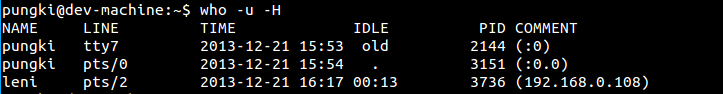
* 1st column show the user name
* 2nd column show how the user connected. Tty means the user is connected directly to the computer, while pts means the user is connected from remote
* 3rd and 4th columns show the date and time
* 5th column show the IP Address where the users are connected

**OPTIONS**

**-H:-** Prints the column headings



**-u:-** Gives the detailed output



The idle time contains the number of hours and minutes since last activity occurred. So 00:13 means that user leni has been idle for 13 minutes. The dot (.) sign tell us that the terminal has seen activity on the last minute. During that time, we can call it “current”. The PID is a process ID of the user’s shell.

**-b:-** Indicate the time and date of the last reboot

Who last reboot

1. **who am i:** Displays the details of the user who invoked the command.
2. **cal command:** The cal command is used to see the calendar of any specific month or a complete year (from 1 – 9999). This facility is total accurate and takes into account the leap year adjustments that took place in the year 1752.

**Syntax: $ cal**

**OPTIONS**

|  |  |
| --- | --- |
| **-1** | Display a single month. This is the default. |
| **-3** | Display three months: last month, this month, and next month. |
| **-s** | Display the calendar using Sunday as the first day of the week. |
| **-m** | Display Monday as the first day of the week. |
| **-j** | Display dates of the Julian calendar. |
| **-y** | Display a calendar for the entire current year. |

**$ cal yyyy =** Displays calendar for the specified year.

1. **date command:** The Linux system maintains an internal clock meant to run perpetually. When the system is shut down a battery backup keeps the clock ticking.

**Syntax**: $ date

**Output**: Sat Nov 7 22:44:59 IST 2009

1. **cut command:** This command is used for text processing. It is used to extract portion of text from a file by selecting columns.

**Syntax: $ cut options filename**

**Options:-**

**-d:-** It is to specify a delimiter

**-f:-** It is used to extract fields from a specific file

**Example**

**1. Display a specific field from a file**

$ cut -d ',' -f 2 employees.txt

Employee Name

John Davies

Mary Fernandes

Jacob Williams

**2. Displaying Multiple Fields from a File**

$ cut -d ',' -f 2-4 employees.txt

Employee Name, Age, Gender

John Davies, 35, M

Mary Fernandes, 29, F

Jacob Williams, 40, M

**3. Change the Delimiter in the Output**

$ cut -d ',' -f 2-4 --output-delimiter='|' employees.txt

Employee Name| Age| Gender

John Davies| 35| M

Mary Fernandes| 29| F

Jacob Williams| 40| M

1. **mv command:** The mv command renames/moves files. It has two distinct functions; (i) It renames a file or directory or (ii) It moves a group of files to different directory.

**Example:-** $ mv test1 test5

The above command renames file from test1 to test5

**Example:-** $ mv linux Ty\_linux

The above command renames a directory from linux to Ty\_linux

**Example:-** $ mv file? Ty\_linux

The above command moves all files starting from word “file” from current directory to Ty\_linux directory.

Note:- ‘?’ and ‘\*’ are considered as wild card characters. ‘?’ matches a single character whereas “\*” matches zero or more characters

1. **rm command:** The rm command deletes one or more files. It normally operates silently and hence should be used with caution

Syntax:- $ rm filename

Example:- $ rm file1

The above command removes/deletes the file from the system without any confirmation.

For removing files that are placed in another directory can be done without using “cd command”.

1. **cp command:** The destination parameter for cp command can be directory also. If the destination is a directory then cp command will copy the file from the existing folder to the destination folder (i.e. Copy and paste operation)

This command allows copying of files and directories from one location to another.

**Syntax:-** $ cp source\_file destintion\_file

When both the source and destination parameters are filenames, the cp command copies the source file to a new file with the filename specified as the destination.

**Example: $** cp file1 dest\_file

In the above example, the destination file “dest\_file” does not exist hence it will be first created then copying will be done. This new file created has a different inode number indicating that it’s a completely new file.

If the destination file already exists, then cp command overwrites the destination file without giving any warning.

**Example: $** cp file1 file2

In the above example, the destination file “file2” already exists hence; it will be overwritten without any warning from the system.

**Screenshot of commands execution:**

**Conclusion**

**Experiment No. 8**

**Aim**: Write shell scripts to do the following:

1. **Display top 10 processes in descending order**

**ps -eo pid,ppid,cmd,%mem,%cpu | head;;**

1. **Display processes with highest memory usage**

The following command will show the list of top processes w.r.t memory usage

**# ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head**

The -o option of ps allows you to specify the output format. --sort is used to sort processes either by %mem or %cpu. By default, the output will be sorted in ascending order, but to reverse that order add a minus sign in front of the sort criteria.

1. **Display current logged in user and logname**

Environment variable $USERNAME and $LOGNAME displays the details of current user

1. **Display current shell, home directory, operating system type, current path, setting path, current working directory**

Following commands are used to display;

1. Current Shell - $SHELL
2. Home Directory - $HOME
3. Operating system type – uname -o
4. Current path - pwd
5. Current Working Directory – pwd
6. **Display OS version, release number, kernel version**

Following commands are used to display;

1. OS version - $ cat /proc/version
2. OS release number - uname -r
3. Kernel version - uname –v

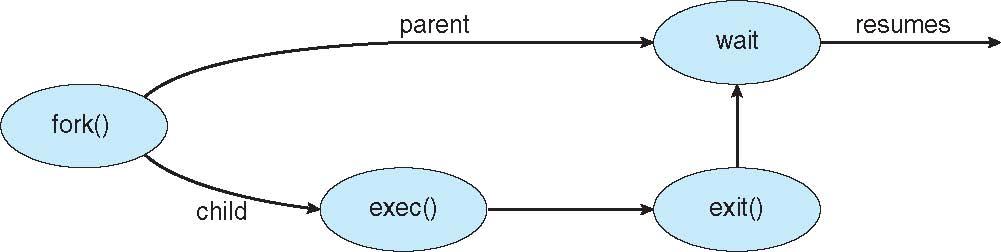
**Screenshots of command execution:**

**Conclusion:**

**Experiment No. 9**

**Aim:** Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call. Explore wait and waitpid before termination of process.

**Theory:** Parent process create children processes, which, in turn create other processes, forming a tree of processes. Generally, process identified and managed via a process identifier (pid). In resource sharing Parent and children share all resources or Children share subset of parent’s resources or Parent and child share no resources. Parent and children execute concurrently or parent waits until children terminate. System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork(). If fork() returns a negative value, the creation of a child process was unsuccessful. fork() returns a zero to the newly created child process. fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid\_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

****

Process executes last statement and then asks the operating system to delete it using the exit() system call. It returns status data from child to parent (via wait()). Process’ resources are deallocated by operating system. Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:

* + Child has exceeded allocated resources
  + Task assigned to child is no longer required
  + The parent is exiting and the operating systems does not allow a child to continue if its parent terminates.

**Algorithm:**

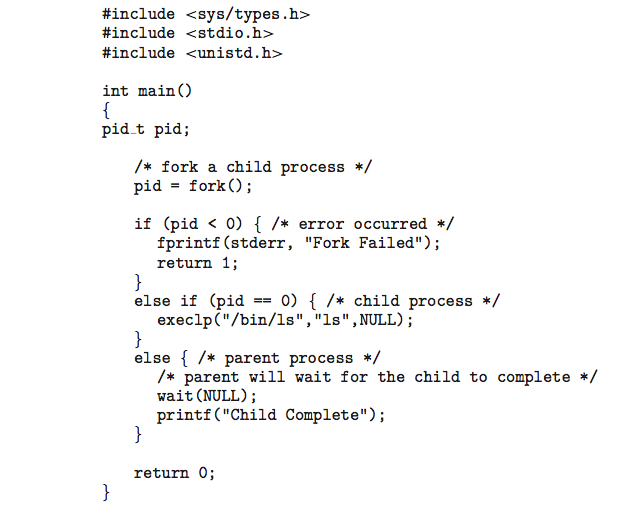
STEP 1: Start the program.

STEP 2: Declare pid as integer.

STEP 3: Create the process using Fork command.

STEP 4: Check pid is less than 0 then print error else if pid is equal to 0 then execute command else parent process wait for child process.

STEP 5: Stop the program.

**C Program Forking Separate Process**

Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated. The termination is initiated by the operating system. The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process

pid = wait(&status);

If no parent waiting (did not invoke wait()) process is a zombie(process) that has completed execution but still has an entry in the process table). If parent terminated without invoking wait , process is an orphan(Parent process no more waiting)

Code:

Output:

Conclusion