Operating Systems | wc-Systems-2017

Handout 1 | Process control and Memory Usage

Process control – System Calls:

The demonstration of fork, and wait system calls along with zombie and orphan states.

1. Use *fork()* system call to create the process. Write simple program which uses this *fork()*.
   Example: Create a child process and assign task of computing the Fibonacci series.
   Solution: fork_fibonacci.c

2. Implement **Orphan** and **Zombie** processes using *fork()*, *sleep()*, *wait()* and *waitpid()* system calls (Hint: Use **$ ps aux | grep Z** command to check for Zombie processes).
   Solution: fork_orphan.c, fork_zombie.c

3. Use *fork()* system call for **3** times to check the number of child processes being created: __________ .
   Solution: fork.c

4. Implement a **C program** in which main program accepts the integers to be sorted. Main program uses the *fork()* system call to create a new process called a child process. **Parent** process sorts the integers using **merge sort** and waits for **child** process using *wait()* system call to sort the integers using **quick sort**.
   Solution: fork_fibonacci.c

Memory Usage:

Execute the following commands on **Linux** based machine:

- **$** free **#** memory management
- **$** free -m **# m**– mega bytes
- **$** cat /proc/meminfo **# /proc** doesnot contain real files
- **$** vmstat **# vmstat**- virtual memory statistics
- **$** vmstat -s **# s**– statistics
- **$** cat /proc/cpuinfo **#** CPU related information
- **$** top **#** memory management
- **$** htop **#** diffences between **VIRT**, **RES**, **SHR**
- **$** sudo dmidecode -t 17 **# DMI Decode**– Desktop Management Interface Decode

Hints:

- Use **$ kill -9 PID** to kill a running process with the process ID = **PID**.
- Use **waitpid(PID, &status, options)** to use `waitpid` and **wait(&status)** to use `wait`.
- Use **qsort(arr, arr_size, sizeof(int), compare)** in **C** to implement Quick Sort using:
  int compare **(const void** * a, **const void** * b) { **return** (*(**int***)a – *(**int***)b) ; }.

Operating Systems | wc-Systems-2017

Handout 2 | IPC – Pipes

1. To generate 25 fibonacci numbers and determine prime amongst them using pipe.
Solution: pipe.cpp

Algorithm

1. Declare a array to store fibonacci numbers
2. Decalre a array **pfd** with two elements for pipe descriptors.
3. Create pipe on pfd using pipe function call.
    - *If return value is -1 then stop*.
4. Using fork system call, create a child process.
5. Let the child process generate **25** fibonacci numbers and store them in a array.
6. Write the array onto pipe using **write** system call.
7. Block the parent till child completes using **wait** system call.
8. Store fibonacci nos. written by child from the pipe in an array using **read** system call
9. Inspect each element of the fibonacci array and check whether they are prime
    - *If prime then print the fibonacci term.*
10. Stop.

Handout 3 | IPC – Two Pipes and FIFOs

Implement Two - Pipes:

1. Write a program that creates **two pipes** and uses them for two way communication between client and server.
Solution: two_pipes.cpp

Algorithm

1. Create two pipes, pipe1 (*pipe1fd[2]*) and pipe2 (*pipe2fd[2]*).
2. Call *fork()* to create a child process and let child process run client code.
3. In child process close **write** end of pipe1 (*pipe1fd[1]*) and **read** end of pipe2 (*pipe2fd[0]*).
4. Parent process close **read** end of pipe1 (*pipe1fd[0]*) and **write** end of pipe2 (*pipe2fd[1]*).

Implement FIFOs:

1. Write a **client – sever** program to make the client send the file name and to make the server send back the contents of the requested file implementing it by FIFO.
Solution: fifo_client.cpp, fifo_server.cpp

Handout 4 | Pipes Application

Experiment: who | wc -l
Aim: To determine number of users logged in using pipe.
Solution: pipe_who.cpp

Algorithm

1. Declare an array *pfd* with two elements for pipe descriptors.
2. Create pipe on pfd using pipe function call.
    - *If return value is -1 then stop.*
3. Using *fork()* system call, create a child process.
4. Free the standard output (1) using close system call to redirect the output to pipe.
5. Make a copy of write end of the pipe using **dup** system call.
6. Execute *who* command using **execlp** system call.
7. Free the standard input (0) using close system call in the other process.
8. Make a close of read end of the pipe using **dup** system call.
9. Execute *wc –l* command using execlp system call.
10. Stop. Thus standard output of **who** is connected to standard input of **wc** using pipe to compute number of users logged in.

Handout 5 | IPC – Message Queues

Implementation of Message Queues:

1. Implement a Simple message queue to send and receive **one** message between the sender and receiver.
   Solution: msgq_sender.cpp, msgq_receiver.cpp
2. Write **2 programs** that will both send and messages and construct the following dialog between them:
   - (Process 1) Sends the message: "**Excited about Webclub mentorship?**"
   - (Process 2) Receives the message and replies: "**Yuhuh! Totally!**"
   - (Process 1) Receives the reply and then says: "**Good to know!**"

   Solution: msgq_chat_sender.cpp, msgq_chat_receiver.cpp

Hint:

- Use **$ ipcs -l** and **$ ipcs -q** to check for limits and messages on the queue respectively.

Handout 7 | IPC – Shared Memory

Implementation of Shared Memory:

1. Implement the basic concept of shared memory using **child** and **parent** processes.
   Solution: shm_fibonacci.cpp

2. Implement the basic concept of shared memory using **server** and **client**.
   Solution: shm_sender.cpp, shm_receiver.cpp