LAB 1

CO327 – Operating Systems

NILUCSHAN . S
E/13/239
SEMESTER 6
04/08/2017

Multiprocessing

DEPARTMENT OF COMPUTER ENGINEERING

The simple servers you write in introductory *Socket Programming* exercises only read or write to a single socket at a time. So, how is a web server like *Apache* able to serve thousands of clients simultaneously? To do that we need to be able to perform reads and writes to sockets *concurrently*. There are three concurrency techniques provided by UNIX:

1. Multiprocessing
2. Multithreading
3. Non blocking I/O

This lab introduces the first technique.

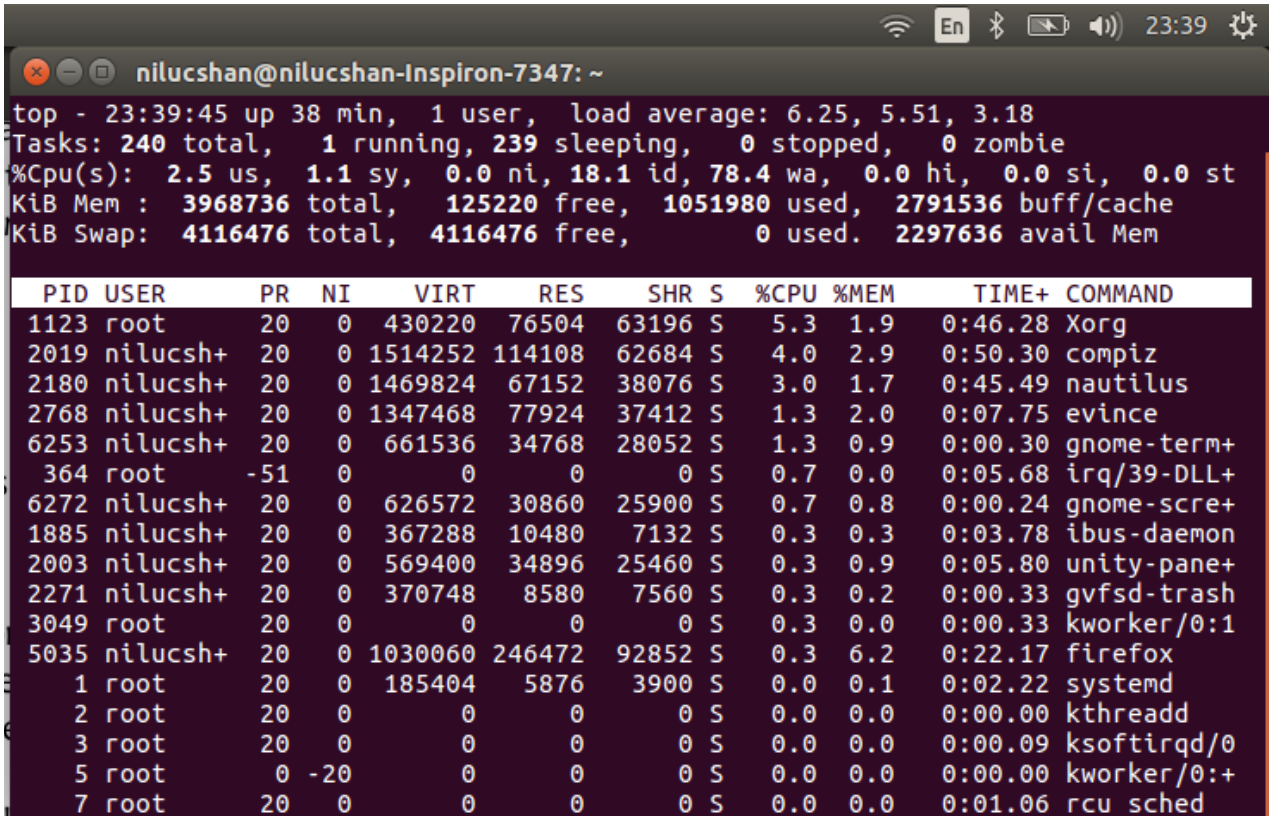
1. Processes

Running a program on UNIX creates a *process*. The kernel creates a *process table* entry to record the resources used by the process, such as memory and open file descriptors. Each process is identified by a unique *process identifier* (PID).

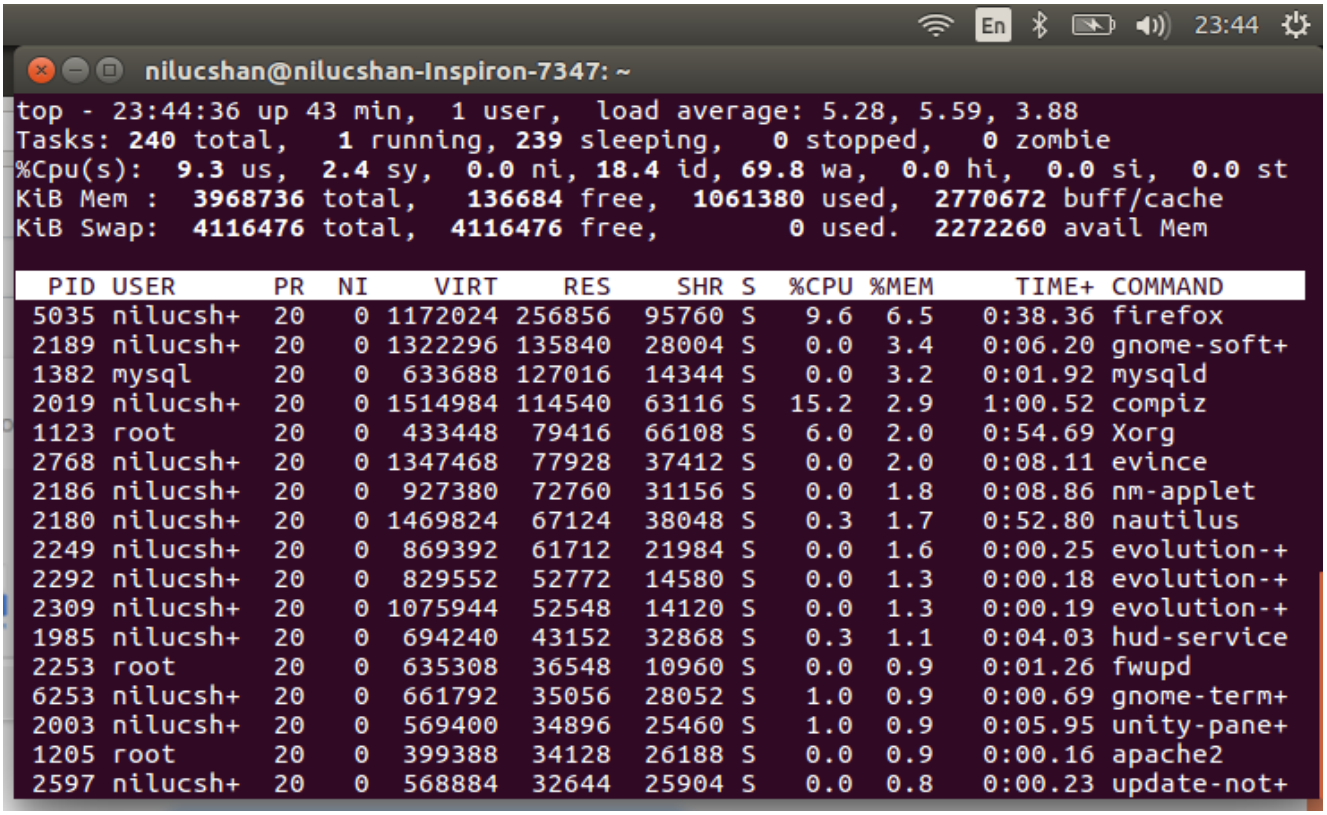
Exercise 1: Use the following commands to view details of the processing running on your system. Note the PIDs.

i. *top* shows you details of active processes. The processes are sorted by CPU usage by default. Sort them by memory usage.

top command sorted by CPU usage – Command **top**

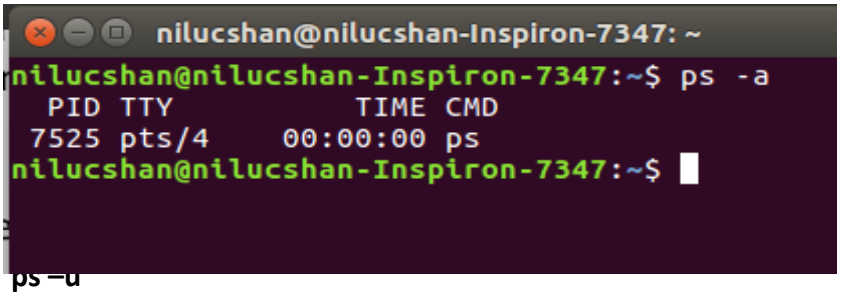


top command sorted by memory usage – Command **top -> press M or Shift + f**



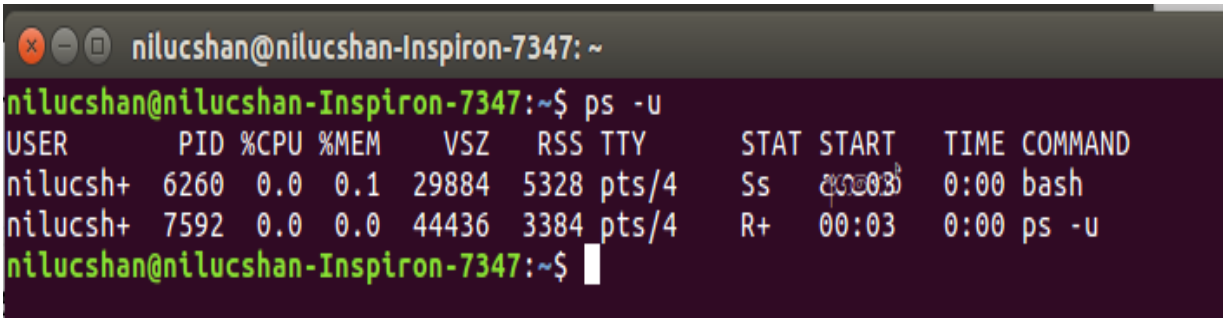
ii. Run *ps* with the following options: *-a, -x, -u, -w*. What is the name of the process with PID 1?

ps -a

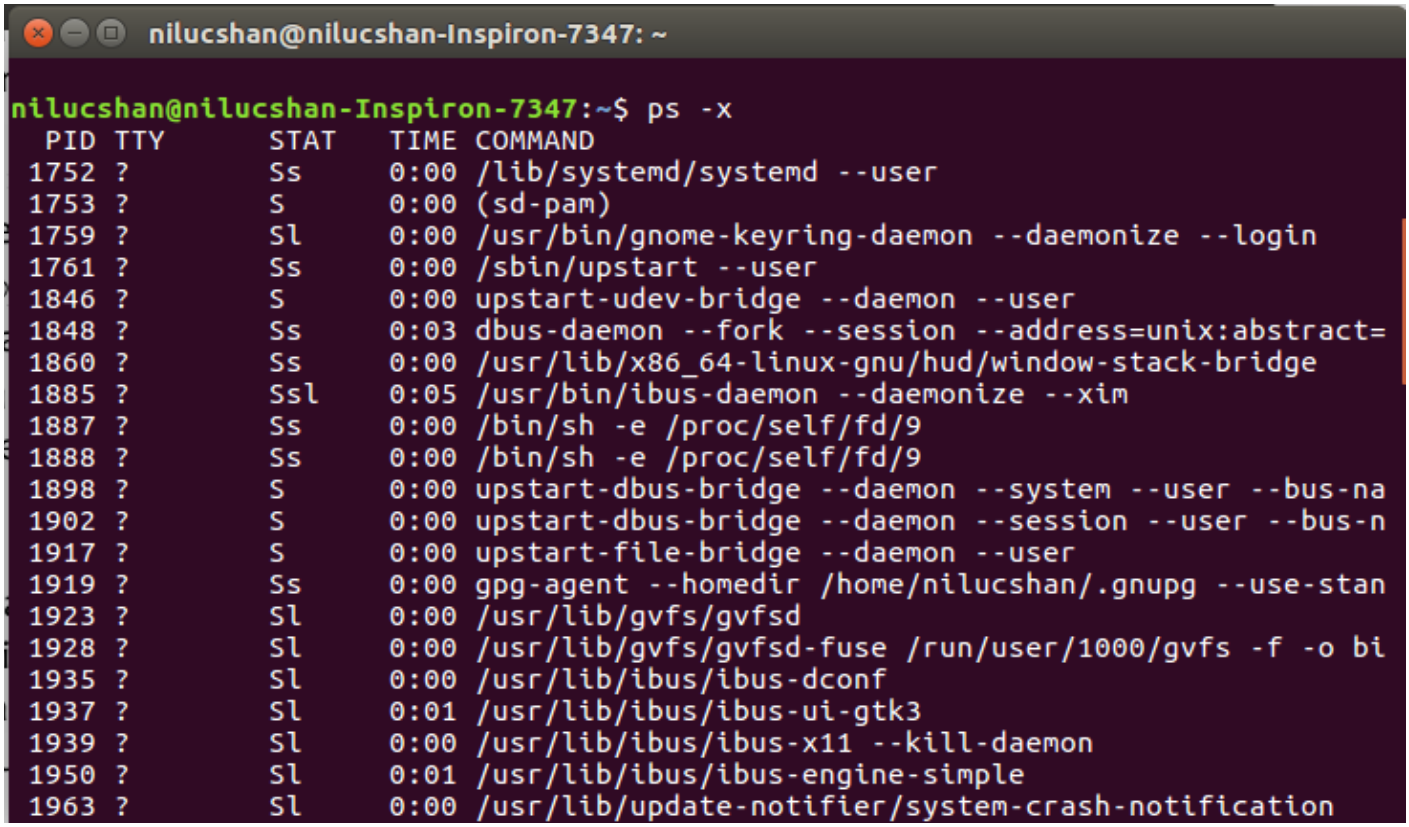


ps –u

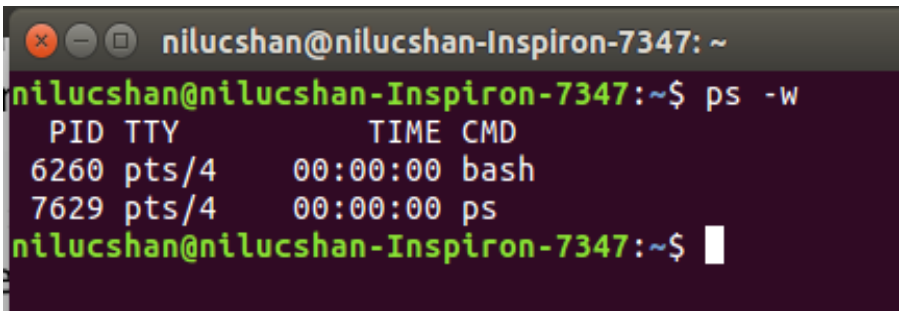
ps -u



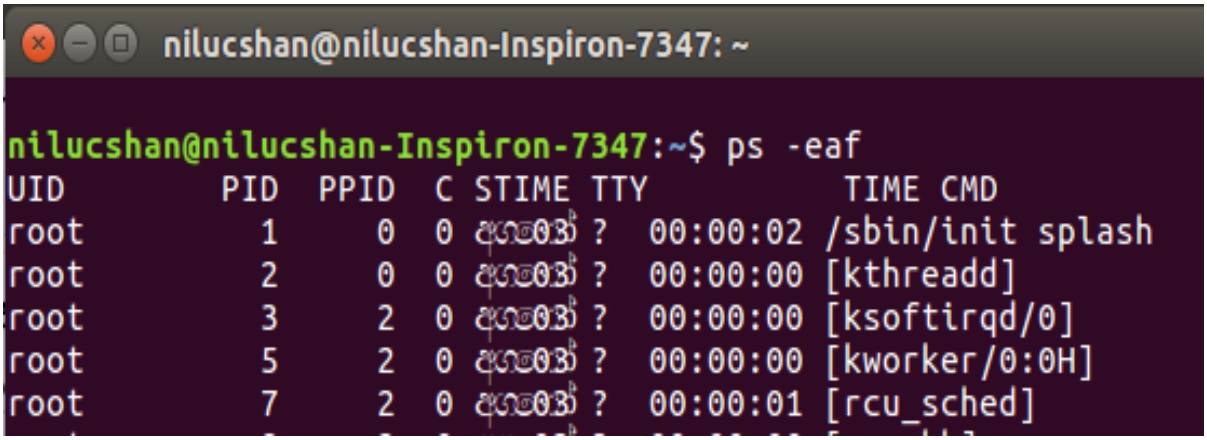
ps –x



ps –w



The process with the PID=1 will be always the **init** process. The init is the process which start and shuts down the system. So obviously it is the first process which should take place in a system. It also can be checked as follows by using the command **ps –eaf**



In the above figure it can be seen that the process with PID=1 is /sbin/init which also has the parent process ID (PPID) as 0.

1.1. Creating a new process

To programmatically create a process UNIX provides the *fork()* system call. When *fork()*is called it creates a new process and *returns twice*, once in the *parent process* and once in the new *child process*. To check whether you are in the parent or child, you check the return value of *fork()*.

```
int     main(void)
{
    int     pid;
    pid     =   fork();
    if (pid         <   0)
    {
        perror("fork"); exit(1);
    }
    if (pid         ==   0)
        puts("This          is the      child       process");
    else
        puts("This          is the      parent      process");


    return      0;
}
```

This is the standard pattern that is used when calling *fork*.

*fork()*is easy, since it can only return three things:

0 If it returns 0, you are the child process. You can get the parent's PID by calling *getppid()*. Of course, you can get your own PID by calling *getpid()*.

-1: If it returns -1, something went wrong, and no child was created. Use *perror()* to see what happened.

else: Any other value returned by *fork()* means that you're the parent and the value returned is the PID of your child. This is the only way to get the PID of your child, since there is no *getcpid()*call. Why?

Since there is no use in invoking getcpid() in a child process.

Exercise 2:

1. In what order are the messages from parent and child printed? Is the order *always* the same?
   Always the parent is printed first and then the child process is printed.

2. How many children will the following program spawn? Draw a diagram illustrating the parent-child relationships between processes.

```
int     main(void)
{
    for (int i=0;     i<3;     i++) fork();
}
```

The program was changes in a way such that we can identify how many child and parents are created when above program is run.

The changed programs is given below. ***(Included as exercise2_2.c in the submission)***

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
    int pid;
    for (int i=0;i<3;i++){
        pid=fork();
        if(pid<0){
            perror("fork");
            exit(1);
        }
        if(pid==0){
            puts("A Child is created");
            printf("My pid(child) is %d\n",getpid());
            printf("My parent pid(child) is %d\n",getppid());
            wait(2);
        }
        else{
            puts("A Parent is created");
            printf("My pid(parent) is  %d\n",getpid());
            printf("My parent(parent) id is  %d\n",getppid());
            wait(2);
        }
    }
return 0;
}
```

Above code was run and the output was stored in a text file. ***(Included as Exercise2_2.txt)***

1.2. Waiting for children

The system call *wait()*lets the parent process wait until the child process has exited. For example, a shell must wait until a command a user has run completes before prompting the user for the next command.

Exercise 3: Modify the program in section 1.1 so that the parent always prints its message *after the* child. Refer to *man 2 wait*for details.

Modified program is included as **exercise3.c**

wait(NULL) is included in the if condition for parent.

1.3. Replacing the process image

In certain cases we would like to execute another program within a process. For example, a shell must create a new process and then run an external program within that process. This is made possible by the *exec()* system call. Doing an exec replaces the current process image in memory with a new program. Therefore a call to exec does not return.

```
int     main(char      argc,      char **argv)
{
    execl("/bin/ls",        "-l", argv[1],    NULL); puts("Program
                            ls has terminated");
}
```

This example is using the *execl()*variation provided by the standard library. See *man 3 exec* for details.

Exercise 4:

1. Compile and run the above code giving it a path as an argument. How many times is the message "*Program ls has terminated*" printed?

   Message was not printed even once.

2. Write a very simple shell that repeatedly prompts the user for a command and runs it with any arguments given. Make sure your shell waits until the command has completed before prompting the user for the next command.

2. Multiprocess servers

We can now apply these techniques to build servers that concurrently handle multiple client requests using multiple server processes. A socket is set up to *listen()*for client connection requests in the same way as iterative servers. When a new client request arrives *accept()*returns a new socket connected to the client.

The main loop of a multiprocess server is where the difference lies. Instead of handling the request itself, the server spawns a child process to handle the client while the parent process continues to *listen()* for new connections. In this way the server is able to handle multiple clients concurrently.

```
listen(sockfd,5);
clilen = sizeof(cli_addr); while
        (1)
{
    /* New socket descriptor is returned each time a client connects*/ newsockfd = accept(sockfd,
    (struct sockaddr *) &cli_addr,&clilen); if (newsockfd < 0)
    {
        perror("ERROR on accept");
        exit(1);
    }
    pid = fork();
    if(pid < 0)
    {
        perror("ERROR on fork");
        exit(1);
    }
    if(pid == 0)
    {
    /* In child processwhich the handles client connection */
        close(sockfd);
        handle_client(newsockfd); exit(0);
    }
    else
    /* In parent processwhich continues to listen for new clients */
        close(newsockfd);
}
```

Exercise 5:

1. Open three terminals and run the server in one. Use *nc()* to connect as two clients concurrently on port 12345. Type some text in both clients and examine the client and server outputs.

   Modified code is included as **exercise5.c**

2. Suppose we modify the server parent process to call *wait()* on the last line above (highlited) to wait until the child serving a client terminates. What would happen?
3. What happens if you terminate the the server while a client is connected, and then try to restart it? (Resolving this issue requires a *signal handler*.)
4. Modify this server to do the following: The client sends the path to a file whose contents the server will send back to the client (if the file exists.) Verify that your new server can handle multiple concurrent connections by using *nc()*. Can two concurrent clients request the same file?