

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.

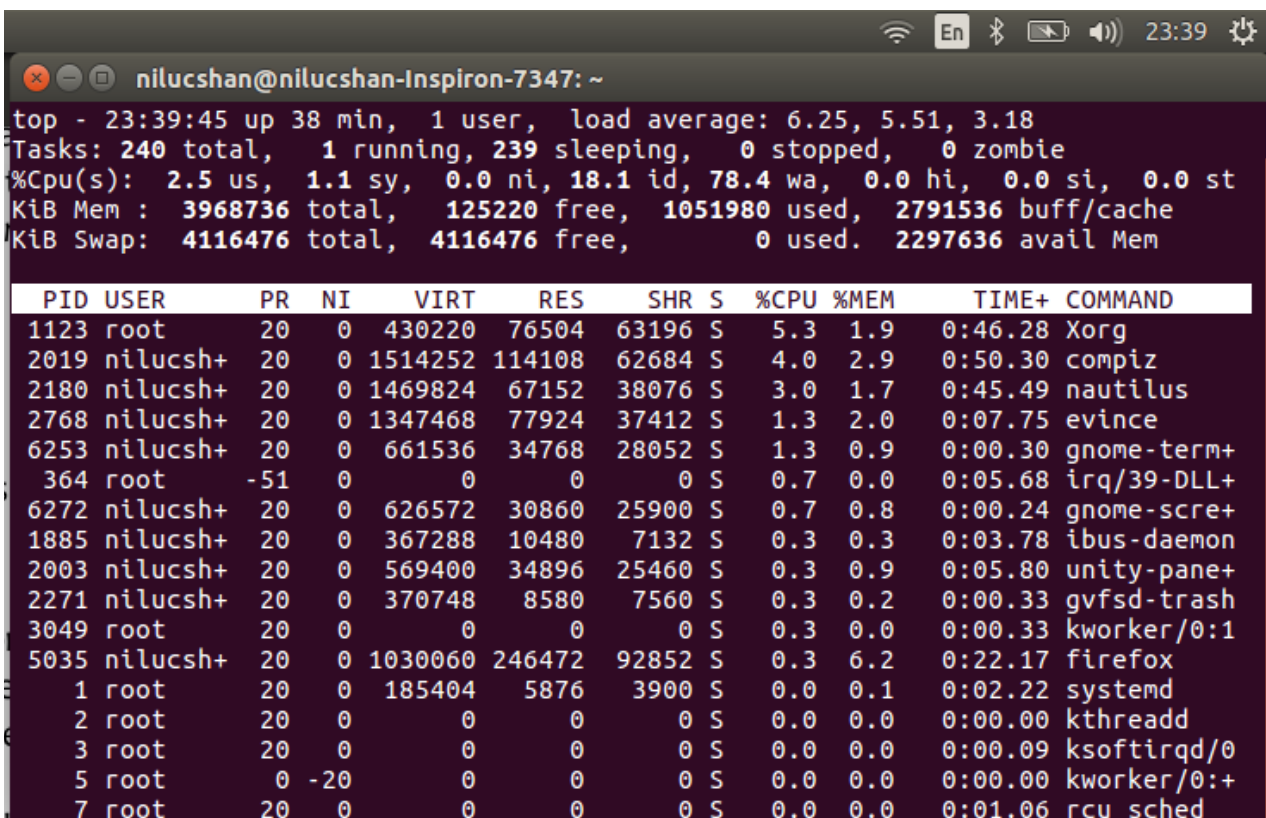
1. Processes

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 - 23:39:45 up 38 min, 1 user, load average: 6.25, 5.51, 3.18
Tasks: 240 total, 1 running, 239 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.5 us, 1.1 sy, 0.0 ni, 18.1 id, 78.4 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3968736 total, 125220 free, 1051980 used, 2791536 buff/cache
KiB Swap: 4116476 total, 4116476 free, 0 used. 2297636 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 1123 root        20   0  430220  76504  63196 S   5.3   1.9   0:46.28 Xorg
 2019 nilucsh+  20   0 1514252 114108  62684 S   4.0   2.9   0:50.30 compiz
 2180 nilucsh+  20   0 1469824  67152  38076 S   3.0   1.7   0:45.49 nautilus
 2768 nilucsh+  20   0 1347468  77924  37412 S   1.3   2.0   0:07.75 evince
 6253 nilucsh+  20   0  661536  34768  28052 S   1.3   0.9   0:00.30 gnome-term+
   364 root       -51   0     0     0     0 S   0.7   0.0   0:05.68 irq/39-DLL+
 6272 nilucsh+  20   0  626572  30860  25900 S   0.7   0.8   0:00.24 gnome-scre+
 1885 nilucsh+  20   0  367288  10480   7132 S   0.3   0.3   0:03.78 ibus-daemon
 2003 nilucsh+  20   0  569400  34896  25460 S   0.3   0.9   0:05.80 unity-pane+
 2271 nilucsh+  20   0  370748   8580   7560 S   0.3   0.2   0:00.33 gvfsd-trash
 3049 root        20   0     0     0     0 S   0.3   0.0   0:00.33 kworker/0:1
 5035 nilucsh+  20   0 1030060 246472  92852 S   0.3   6.2   0:22.17 firefox
     1 root        20   0  185404   5876   3900 S   0.0   0.1   0:02.22 systemd
     2 root        20   0     0     0     0 S   0.0   0.0   0:00.00 kthreadd
     3 root        20   0     0     0     0 S   0.0   0.0   0:00.09 ksoftirqd/0
     5 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 kworker/0:++
     7 root        20   0     0     0     0 S   0.0   0.0   0:01.06 rcu sched
```

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

```
nilucshan@nilucshan-Inspiron-7347: ~
top - 23:44:36 up 43 min,  1 user,  load average: 5.28, 5.59, 3.88
Tasks: 240 total,  1 running, 239 sleeping,  0 stopped,  0 zombie
%Cpu(s):  9.3 us,  2.4 sy,  0.0 ni, 18.4 id, 69.8 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 3968736 total, 136684 free, 1061380 used, 2770672 buff/cache
KiB Swap: 4116476 total, 4116476 free,  0 used. 2272260 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 5035 nilucsh+  20   0 1172024 256856 95760 S   9.6   6.5   0:38.36 firefox
 2189 nilucsh+  20   0 1322296 135840 28004 S   0.0   3.4   0:06.20 gnome-soft+
 1382 mysql    20   0  633688 127016 14344 S   0.0   3.2   0:01.92 mysqld
 2019 nilucsh+  20   0 1514984 114540 63116 S  15.2   2.9   1:00.52 compiz
 1123 root     20   0  433448  79416 66108 S   6.0   2.0   0:54.69 Xorg
 2768 nilucsh+  20   0 1347468  77928 37412 S   0.0   2.0   0:08.11 evince
 2186 nilucsh+  20   0  927380  72760 31156 S   0.0   1.8   0:08.86 nm-applet
 2180 nilucsh+  20   0 1469824  67124 38048 S   0.3   1.7   0:52.80 nautilus
 2249 nilucsh+  20   0  869392  61712 21984 S   0.0   1.6   0:00.25 evolution-+
 2292 nilucsh+  20   0  829552  52772 14580 S   0.0   1.3   0:00.18 evolution-+
 2309 nilucsh+  20   0 1075944  52548 14120 S   0.0   1.3   0:00.19 evolution-+
 1985 nilucsh+  20   0  694240  43152 32868 S   0.3   1.1   0:04.03 hud-service
 2253 root     20   0  635308  36548 10960 S   0.0   0.9   0:01.26 fwupd
 6253 nilucsh+  20   0  661792  35056 28052 S   1.0   0.9   0:00.69 gnome-term+
 2003 nilucsh+  20   0  569400  34896 25460 S   1.0   0.9   0:05.95 unity-pane+
 1205 root     20   0  399388  34128 26188 S   0.0   0.9   0:00.16 apache2
 2597 nilucsh+  20   0  568884  32644 25904 S   0.0   0.8   0:00.23 update-not+
```

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

ps -a

```
nilucshan@nilucshan-Inspiron-7347: ~
nilucshan@nilucshan-Inspiron-7347:~$ ps -a
  PID TTY          TIME CMD
 7525 pts/4        00:00:00 ps
nilucshan@nilucshan-Inspiron-7347:~$
```

ps -u

ps -u

```
nilucshan@nilucshan-Inspiron-7347: ~
nilucshan@nilucshan-Inspiron-7347:~$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
nilucsh+  6260  0.0  0.1  29884  5328 pts/4    Ss   00:03   0:00 bash
nilucsh+  7592  0.0  0.0  44436  3384 pts/4    R+   00:03   0:00 ps -u
nilucshan@nilucshan-Inspiron-7347:~$
```

ps -x

```
nilucshan@nilucshan-Inspiron-7347: ~  
nilucshan@nilucshan-Inspiron-7347:~$ ps -x  
  PID TTY          STAT       TIME COMMAND  
 1752 ?        Ss          0:00    /lib/systemd/systemd --user  
 1753 ?        S            0:00    (sd-pam)  
 1759 ?        Sl          0:00    /usr/bin/gnome-keyring-daemon --daemonize --login  
 1761 ?        Ss          0:00    /sbin/upstart --user  
 1846 ?        S            0:00    upstart-udev-bridge --daemon --user  
 1848 ?        Ss          0:03    dbus-daemon --fork --session --address=unix:abstract=  
 1860 ?        Ss          0:00    /usr/lib/x86_64-linux-gnu/hud/window-stack-bridge  
 1885 ?        Ssl         0:05    /usr/bin/ibus-daemon --daemonize --xim  
 1887 ?        Ss          0:00    /bin/sh -e /proc/self/fd/9  
 1888 ?        Ss          0:00    /bin/sh -e /proc/self/fd/9  
 1898 ?        S            0:00    upstart-dbus-bridge --daemon --system --user --bus-na  
 1902 ?        S            0:00    upstart-dbus-bridge --daemon --session --user --bus-n  
 1917 ?        S            0:00    upstart-file-bridge --daemon --user  
 1919 ?        Ss          0:00    gpg-agent --homedir /home/nilucshan/.gnupg --use-stan  
 1923 ?        Sl          0:00    /usr/lib/gvfs/gvfsd  
 1928 ?        Sl          0:00    /usr/lib/gvfs/gvfsd-fuse /run/user/1000/gvfs -f -o bi  
 1935 ?        Sl          0:00    /usr/lib/ibus/ibus-dconf  
 1937 ?        Sl          0:01    /usr/lib/ibus/ibus-ui-gtk3  
 1939 ?        Sl          0:00    /usr/lib/ibus/ibus-x11 --kill-daemon  
 1950 ?        Sl          0:01    /usr/lib/ibus/ibus-engine-simple  
 1963 ?        Sl          0:00    /usr/lib/update-notifier/system-crash-notification
```

ps -w

```
nilucshan@nilucshan-Inspiron-7347: ~  
nilucshan@nilucshan-Inspiron-7347:~$ ps -w  
  PID TTY          TIME CMD  
 6260 pts/4      00:00:00 bash  
 7629 pts/4      00:00:00 ps  
nilucshan@nilucshan-Inspiron-7347:~$
```

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**

```
nilucshan@nilucshan-Inspiron-7347: ~  
nilucshan@nilucshan-Inspiron-7347:~$ ps -eaf  
UID          PID     PPID  C STIME TTY          TIME CMD  
root           1         0  0  0:00 ?        00:00:02 /sbin/init splash  
root           2         0  0  0:00 ?        00:00:00 [kthreadd]  
root           3         2  0  0:00 ?        00:00:00 [ksoftirqd/0]  
root           5         2  0  0:00 ?        00:00:00 [kworker/0:0H]  
root           7         2  0  0:00 ?        00:00:01 [rcu_sched]
```

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

The changed program 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 waitfor* 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 process which the handles client connection */
        close(sockfd);
        handle_client(newsockfd); exit(0);
    }
    else
    /* In parent process which continues to listen for new clients */
}
```

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 (highlighted) 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?