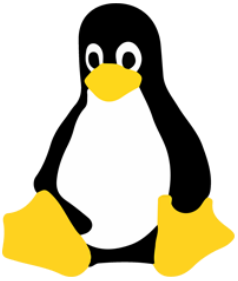
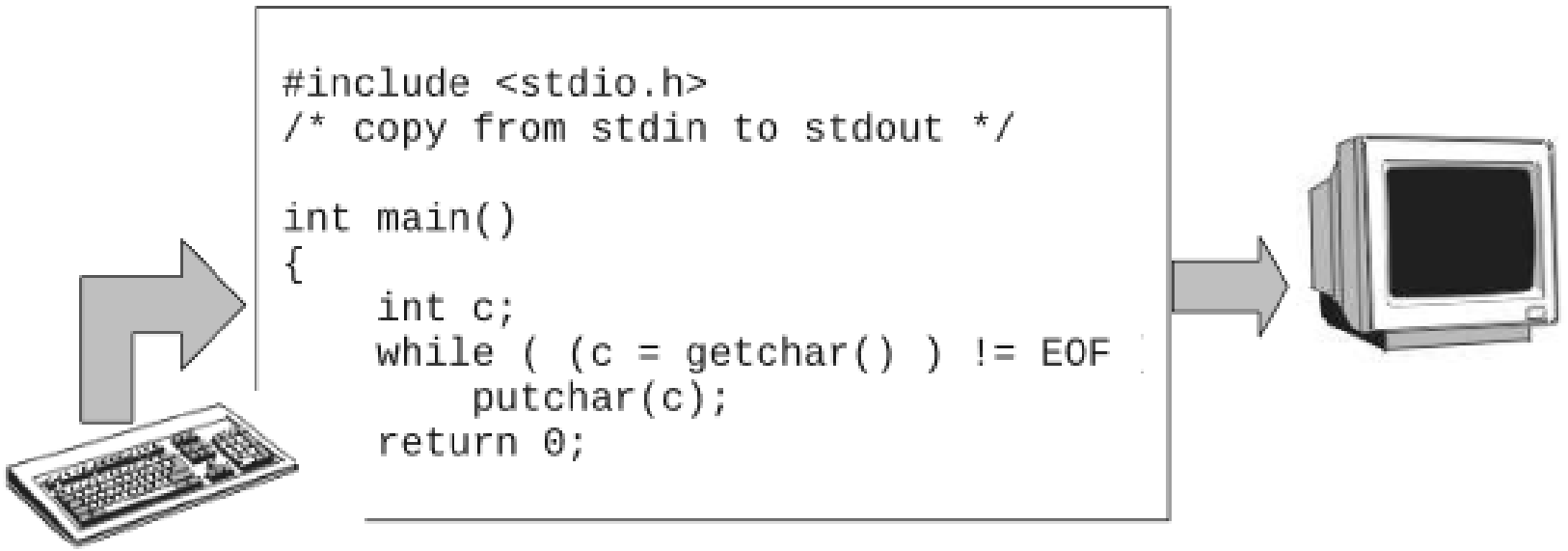


CS416: Systems Programming

Abdullah Alfarrarjeh

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash

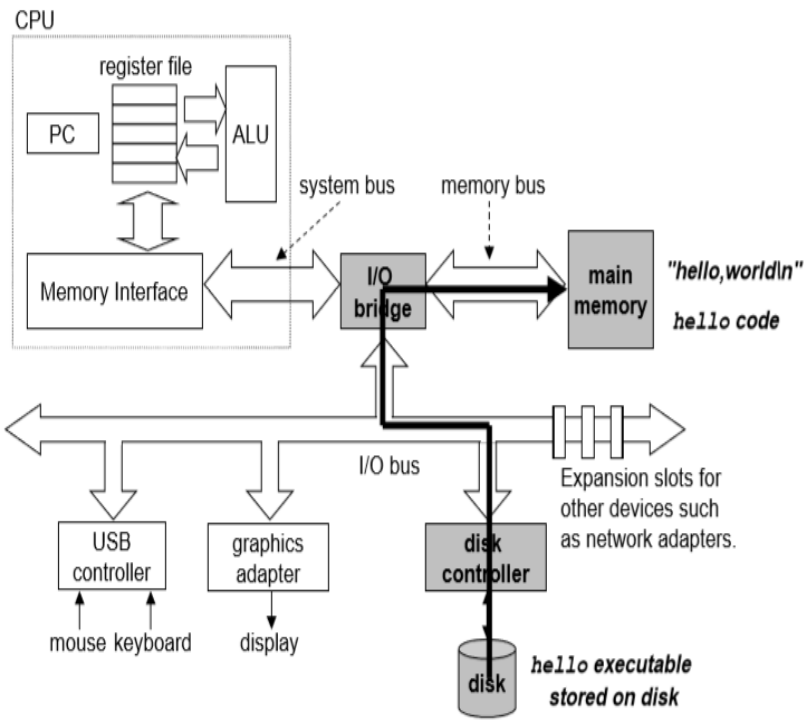
hello.c

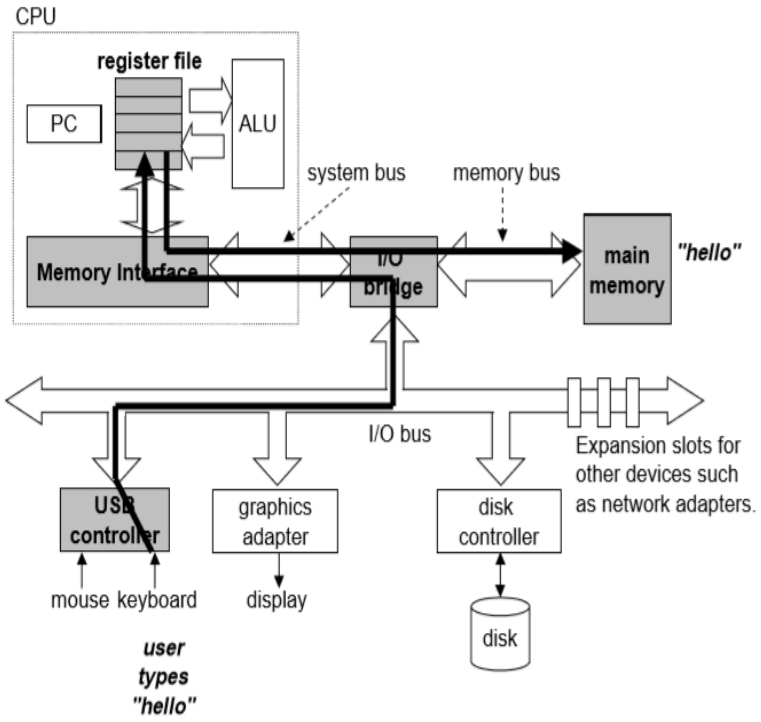


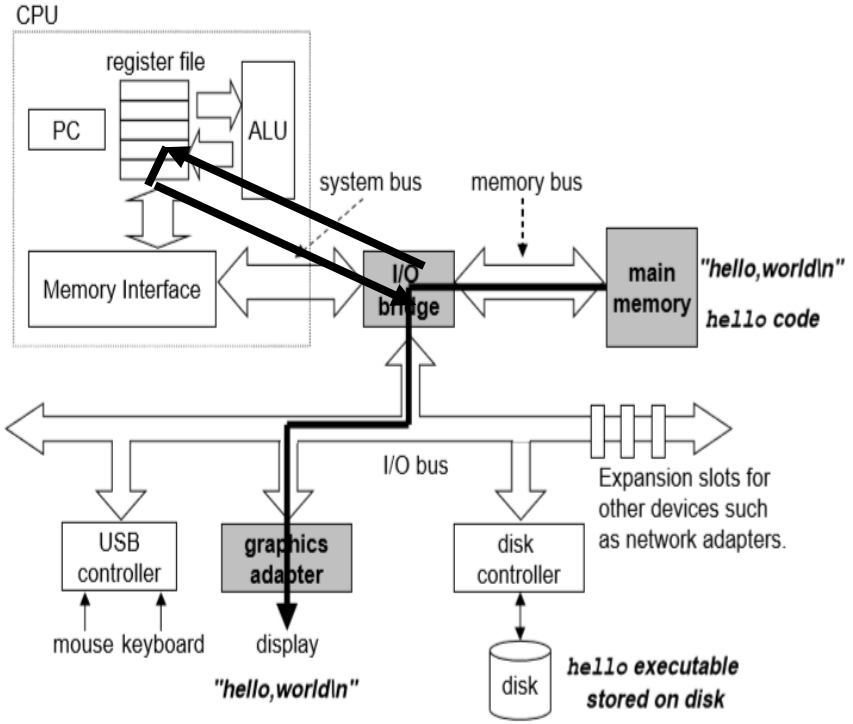
```
#include <stdio.h>
/* copy from stdin to stdout */

int main()
{
    int c;
    while ( (c = getchar() ) != EOF
        putchar(c);
    return 0;
```

Running the hello.c program (Hardware side)

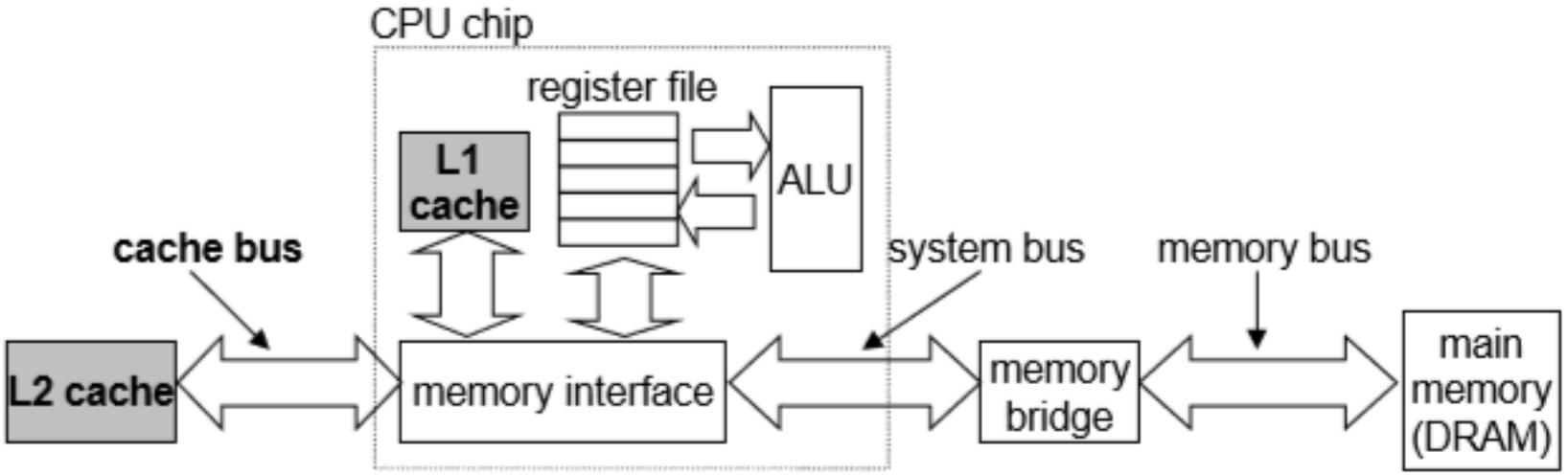






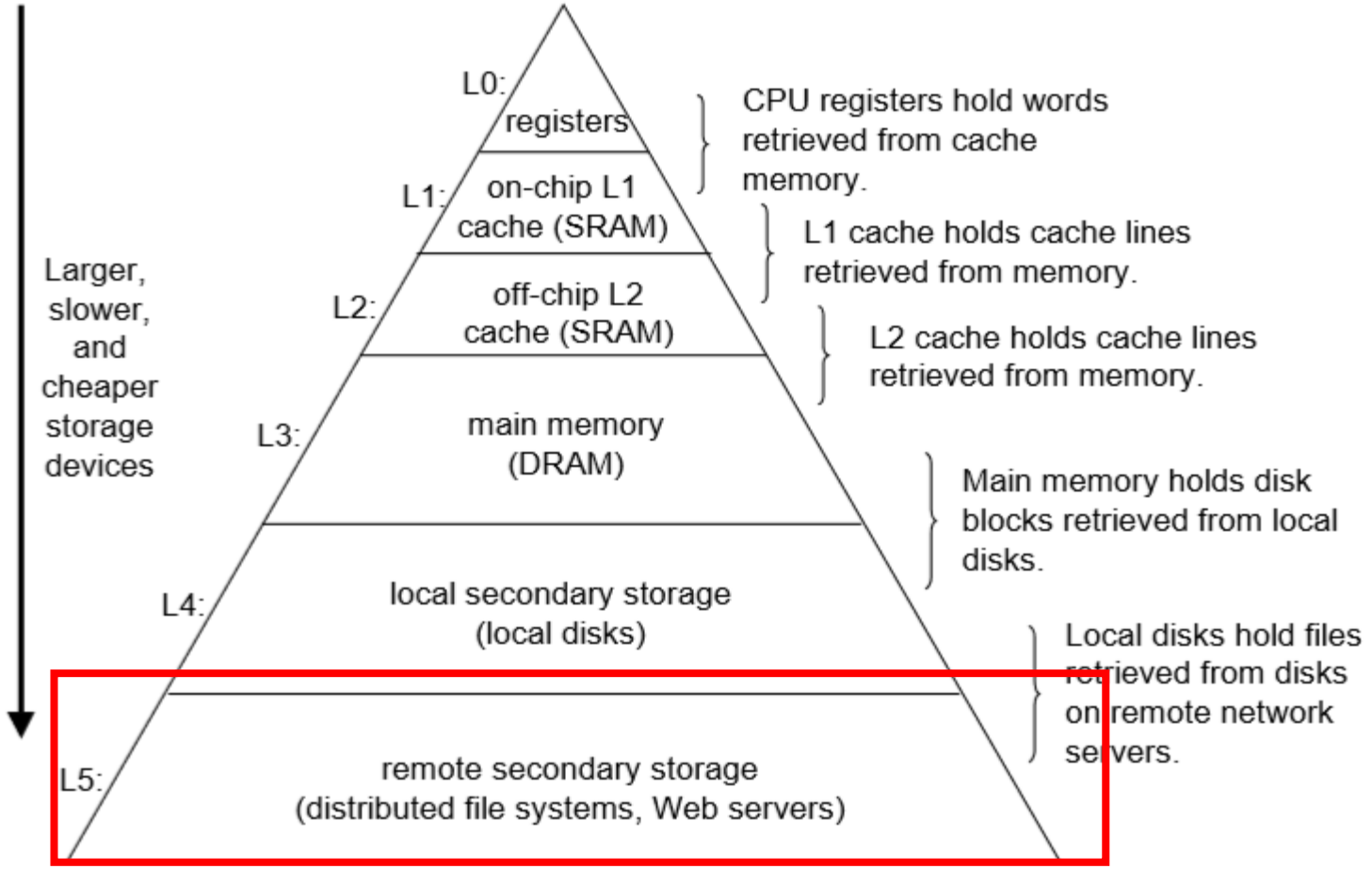
- A system spends a lot time moving information from one place to another. The machine instructions in the hello program are originally stored on disk and copied to main memory then into the processor. Similarly, the data string ”hello,world\n”, originally on disk, is copied to main memory, and then copied from main memory to the display device

The processor memory gab



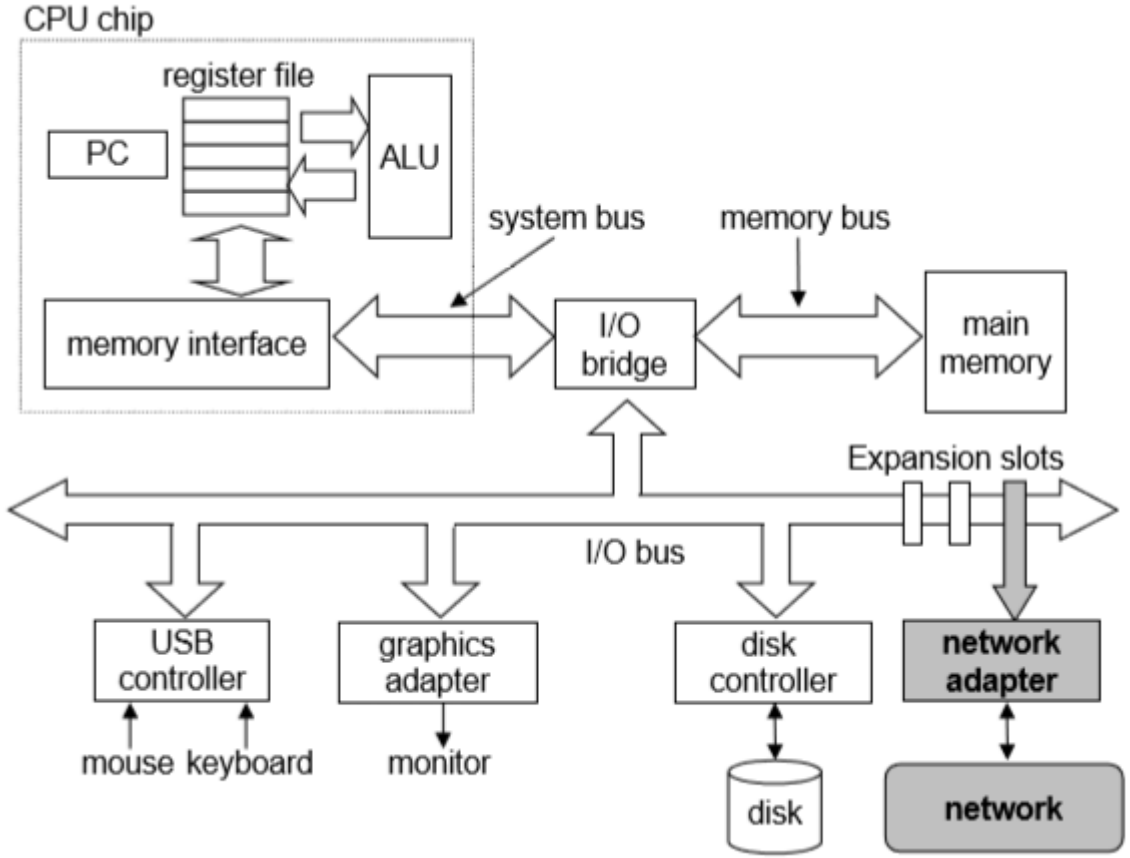
- It is easier and cheaper to make processors run faster than it is to make main memory run faster.
- To handle the processor memory gab problem, the cash memory was introduced.

larger storage devices are slower than smaller storage devices



- The disk drive on a typical system might be 100 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory

Network is just like another I/O device



- When the system copies a sequence of bytes from main memory to the network adapter, the data flows across the network to another machine, instead of say, to a local disk drive or to the main memory

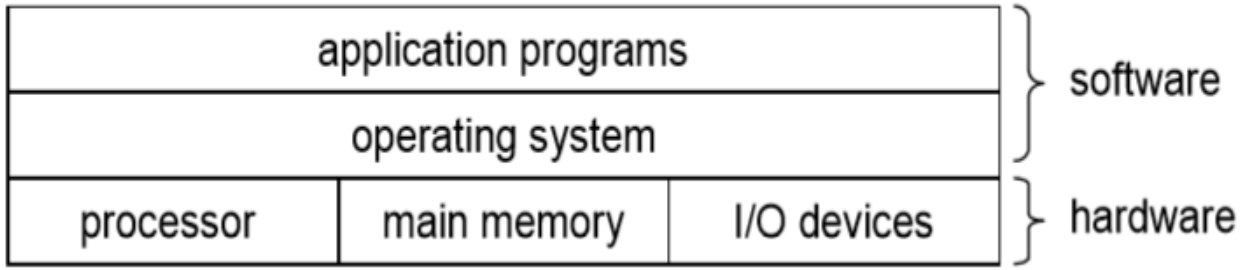
How the OS looks at applications

OS

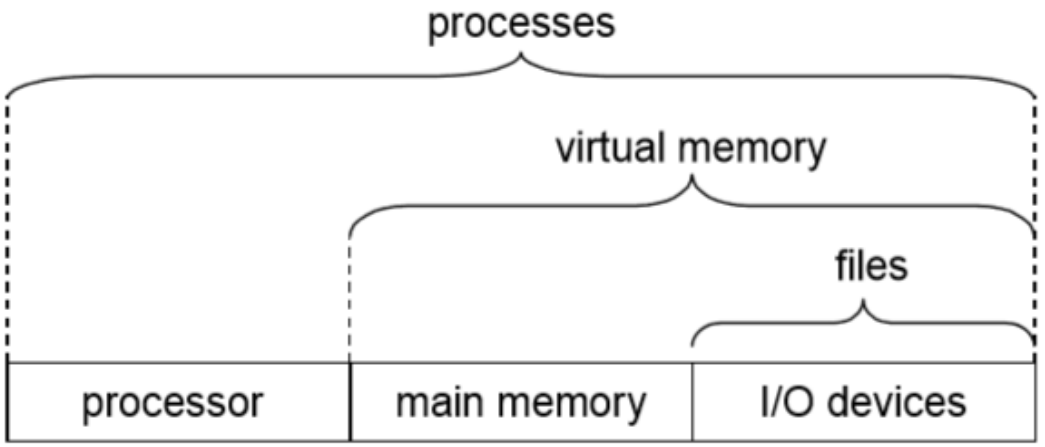
Must protect the hardware from this reckless driver

The Operating System Manages the Hardware

- Layered view of a computer system

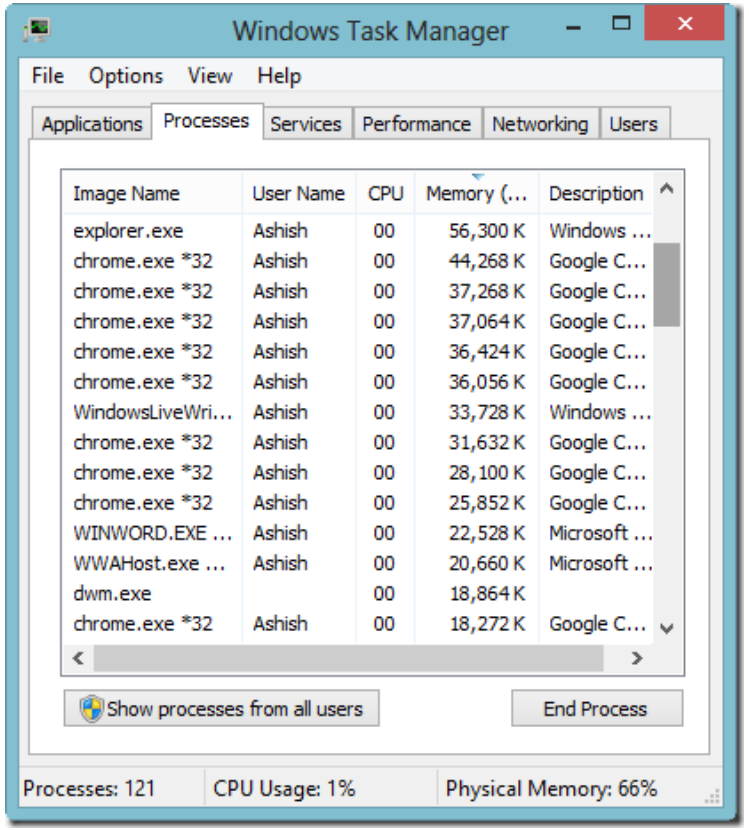


- OS abstractions



- The OS has <u>two</u> primary purposes:
  1. Protect the hardware from misuse by runaway applications
  2. Provide applications with uniform mechanisms for manipulating low-level hardware devices

Processes




```
top - 17:07:00 up  2:38,  1 user,  load average: 0.45, 0.70, 0.62
Tasks: 104 total,   1 running, 102 sleeping,   0 stopped,   1 zombie
Cpu(s): 23.8% us,  3.3% sy,  0.0% ni, 72.8% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    768600k total,   691768k used,    76832k free,    37960k buffers
Swap:   979956k total,        0k used,   979956k free,   368184k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4421 tv        15   0  148m  81m  18m S  9.9 10.8  13:56.36 firefox-bin
 4090 root       6 -10  173m  41m 3036 S  6.9  5.5   7:32.76 XFree86
 6109 tv        15   0 29604  14m  12m S  4.3  2.0   0:01.32 ksnapshot
 4339 tv        15   0 31440  14m  11m S  2.3  1.9   0:20.42 kicker
 4335 tv        15   0 27736  12m 9.8m S  2.0  1.6   0:13.47 kwin
 4424 tv        15   0 31380  14m  11m S  0.7  2.0   0:17.15 konsole
 6090 root      16   0  2272 1152  872 R  0.3  0.1   0:00.25 top
    1 root      16   0  1940  664  568 S  0.0  0.1   0:00.25 init
    2 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    3 root      10  -5     0    0    0 S  0.0  0.0   0:00.35 events/0
    4 root      11  -5     0    0    0 S  0.0  0.0   0:00.01 khelper
    5 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 kthread
    7 root      10  -5     0    0    0 S  0.0  0.0   0:00.08 kblockd/0
    8 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 kacpid
  117 root      20   0     0    0    0 S  0.0  0.0   0:00.00 pdflush
  118 root      15   0     0    0    0 S  0.0  0.0   0:00.03 pdflush
  120 root      11  -5     0    0    0 S  0.0  0.0   0:00.00 aio/0
  119 root      25   0     0    0    0 S  0.0  0.0   0:00.00 kswapd0
  709 root      10  -5     0    0    0 S  0.0  0.0   0:00.01 kseriod
  811 root      15   0     0    0    0 S  0.0  0.0   0:00.39 kjournald
 1404 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 khubd
 1906 root      11  -5     0    0    0 S  0.0  0.0   0:00.00 scsi_eh_1
 1909 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 usb-storage
 3102 daemon    16   0  1712  376  280 S  0.0  0.0   0:00.00 portmap
 3353 root      16   0  1660  624  512 S  0.0  0.1   0:00.26 syslogd
 3359 root      16   0  2428 1304  380 S  0.0  0.2   0:00.08 klogd
 3367 dnsmasq   16   0  1888  732  608 S  0.0  0.1   0:00.08 dnsmasq
 3516 root      16   0  1764  696  572 S  0.0  0.1   0:00.02 automount
 3523 root      16   0  1756  680  564 S  0.0  0.1   0:00.00 automount
 3586 root      16   0  1764  696  572 S  0.0  0.1   0:00.00 automount
```

what do you know about process?

>> ps –aux | less

Process

- **A running instance of a program** is called a **process**
- If you run the hello.c program in the terminal, how many related processes will you see??

  at least ……2……

- Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created

Process ID

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

int main ()
{
        printf (“The process ID is %d\n”, (int) getpid ());
        printf (“The parent process ID is %d\n”, (int) getppid ());
        return 0;
}
```

Creating processes

- **Method (1) using the "*system*" function**
  - The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell

```
#include <stdlib.h>
int main ()
{
        int return_value;
        return_value = system ("ls -l /");
        return return_value;
}
```

Creating processes

- **Method (2) using *fork* and *exec***
  - Linux provides one function, **fork**, that makes a child process that is an exact copy of its parent process
  - Linux provides another set of functions, the **exec** family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program
  - To **spawn** a new process, you first use **fork** to make a copy of the current process. Then you use **exec** to transform one of these processes into an instance of the program you want to spawn

Using fork to duplicate a program's process

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
        pid_t child_pid;
        printf ("the main program process ID is %d\n", (int) getpid ());
        child_pid = fork ();
        if (child_pid != 0)
                {
                        printf ("this is the parent process, with id %d\n", (int) getpid ());
                        printf ("the child's process ID is %d\n", (int) child_pid);
                }
        else
                printf ("this is the child process, with id %d\n", (int) getpid ());
        return 0;
}
```