Linux File Systems and Commands

This documents describes our Linux system and the commands to use to retrieve those information. All highlighted text are commands you could give to get the same information

Linux System Information:

The command uname would print the system information

```
cprog> uname -a
```

You can type various options to get these information

uname –r (for release)

uname –s (for kernel release)

uname –p (for processor)

uname –a (Print certain system information. With no OPTION, same as -s)

You can also the same information by running the command

```
cat /proc/version
```

Hardware Information:

There are various commands for this,

lscpu - lscpu gathers CPU architecture information like number of CPUs, threads, cores, sockets, NUMA nodes, information about CPU caches, CPU family, model, byte order and prints it in a human-readable format.

lshw – lshw is a small tool to extract detailed information on the hardware configuration of the machine.

lspci - lspci is a utility for displaying information about PCI buses in the system and devices connected to them.

lsusb - lsusb is a utility for displaying information about USB buses in the system and the devices connected to them.

In the lshw and lscpu output, you will find could of interesting information. Namely, the number of disks, CPU, IDE (# of disks), memory

The actual path of these commands can be determined by typing

which lscpu

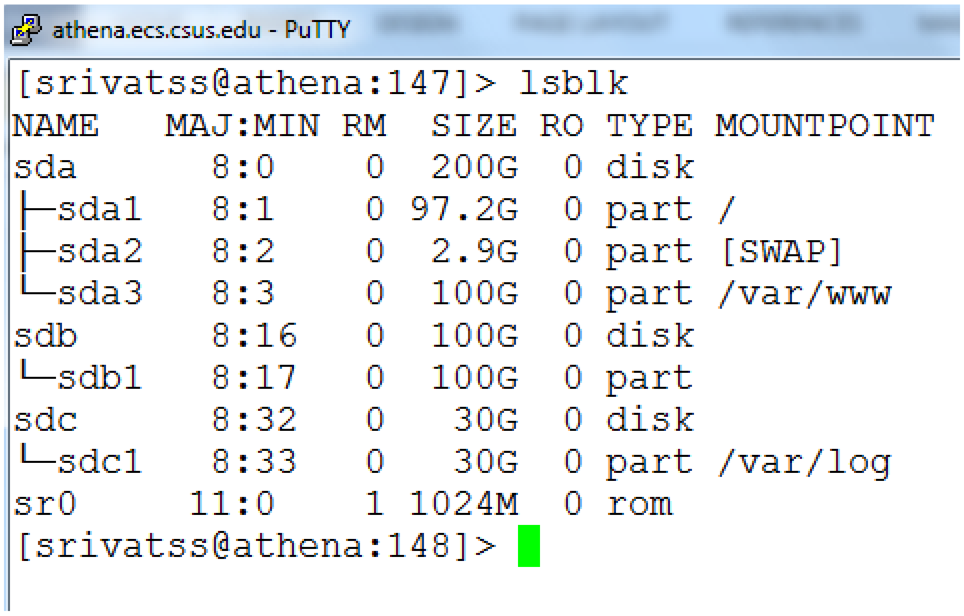
would output /usr/bin/lscpu

In our system, we find there is only one disk (it might change in future).

we can get more information about the disk and partitions and the mounts.

To get the partitions in our system, we type

lsblk



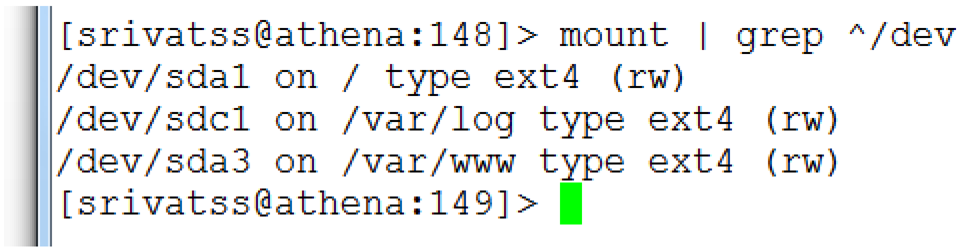
```
[srivatss@athena:147]> lsblk
NAME    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda       8:0    0   200G  0 disk
├─sda1    8:1    0 97.2G  0 part /
├─sda2    8:2    0   2.9G  0 part [SWAP]
└─sda3    8:3    0   100G  0 part /var/www
sdb       8:16   0   100G  0 disk
└─sdb1    8:17   0   100G  0 part
sdc       8:32   0    30G  0 disk
└─sdc1    8:33   0    30G  0 part /var/log
sr0      11:0    1 1024M  0 rom
[srivatss@athena:148]>
```

and you get the above information. The device drivers usr the major and minor numbers uniquely identify the partition, the major and minor numbers each are 16bit numbers. We will ignore this for now. We look at the mounts points which is what we should be looking. These partitions are mapped to a path using which we access the partitions.

Files are stored in a partition, partitions are formatted, and the formatted devices are mounted to a folder name. User have access to mount points.

To see the various filesystems, we have a linux command – mount.   I have piped the output of mount to /dev .

c-prog>mount | grep ^/dev



```
[srivatss@athena:148]> mount | grep ^/dev
/dev/sda1 on / type ext4 (rw)
/dev/sdc1 on /var/log type ext4 (rw)
/dev/sda3 on /var/www type ext4 (rw)
[srivatss@athena:149]>
```

The process of redirecting the output of one command to another command is called piping and is represented as | (vertical bar)

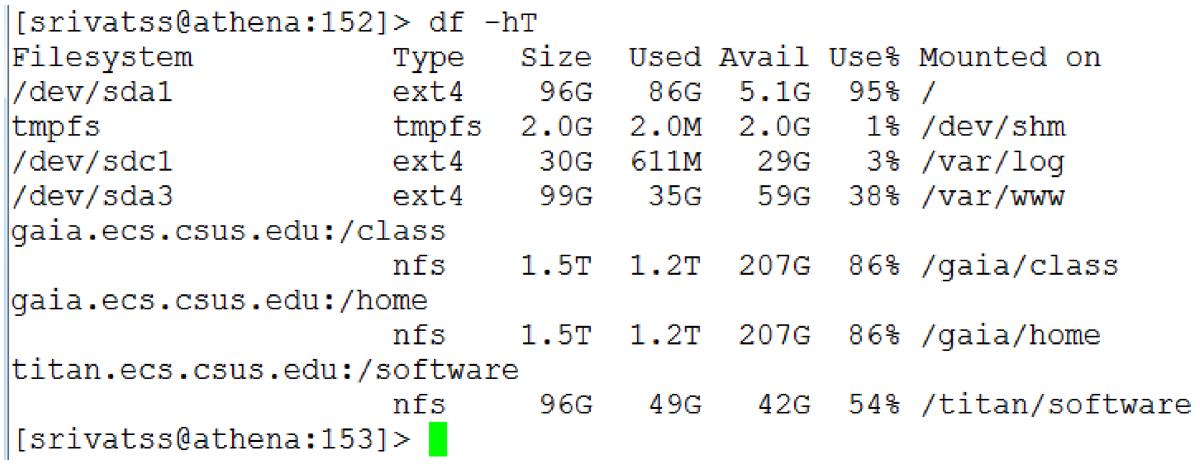
Note: ext2, ext3, ext4 are types of filesystems and the format is very different. Generally the boot is formatted in ext2 format.

You can also open the file /etc/fstab and this files shows all the mount points.

Disk usage:

df –hT

df displays the amount of disk space available on the file system, h stands for human readable format, T stands for Type. Currently, the output is seen as



```
[srivatss@athena:152]> df -hT
Filesystem            Type   Size  Used Avail Use% Mounted on
/dev/sda1             ext4    96G   86G  5.1G  95% /
tmpfs                 tmpfs  2.0G  2.0M  2.0G   1% /dev/shm
/dev/sdc1             ext4    30G  611M   29G   3% /var/log
/dev/sda3             ext4    99G   35G   59G  38% /var/www
gaia.ecs.csus.edu:/class
                      nfs    1.5T  1.2T  207G  86% /gaia/class
gaia.ecs.csus.edu:/home
                      nfs    1.5T  1.2T  207G  86% /gaia/home
titan.ecs.csus.edu:/software
                      nfs     96G   49G   42G  54% /titan/software
[srivatss@athena:153]>
```

You can see the file system on the left, the size, used and availability and mount point.

What is ROOT: Root is the top most directory under which all other directories reside, note the actual file system they point may be on different file system. You can navigate to the root directory type

```
cd /
```

and navigate to sub-directories as shown in the screenshot.

Various file folders under Linux starting from root.

Command	Comment
cd /	will take you to the root folder. do ls to see all folder. bin is one folder
cd /bin	here all basic shell commands exist. Commands such as ls, mkdir, pwd,
cd /sbin	here all system related commands exist. We will skip here
cd /home	All user accounts are stored. Staff accounts are stored in /home/staff/ and student accounts are stored in /home/student/
cd /lib	all library files are stored here
cd /var	A folder to store temporary files (usually log files which generally removed / purged after sometime)
cd /etc	Here important configurations system files are stored. Files such as user account and password (/etc/passwd) , file partitions (etc/fstab)

Where is User Accounts and Password file:

The user ID, name, password, groupID and the default home directory and shell are stored in a file : /etc/passwd

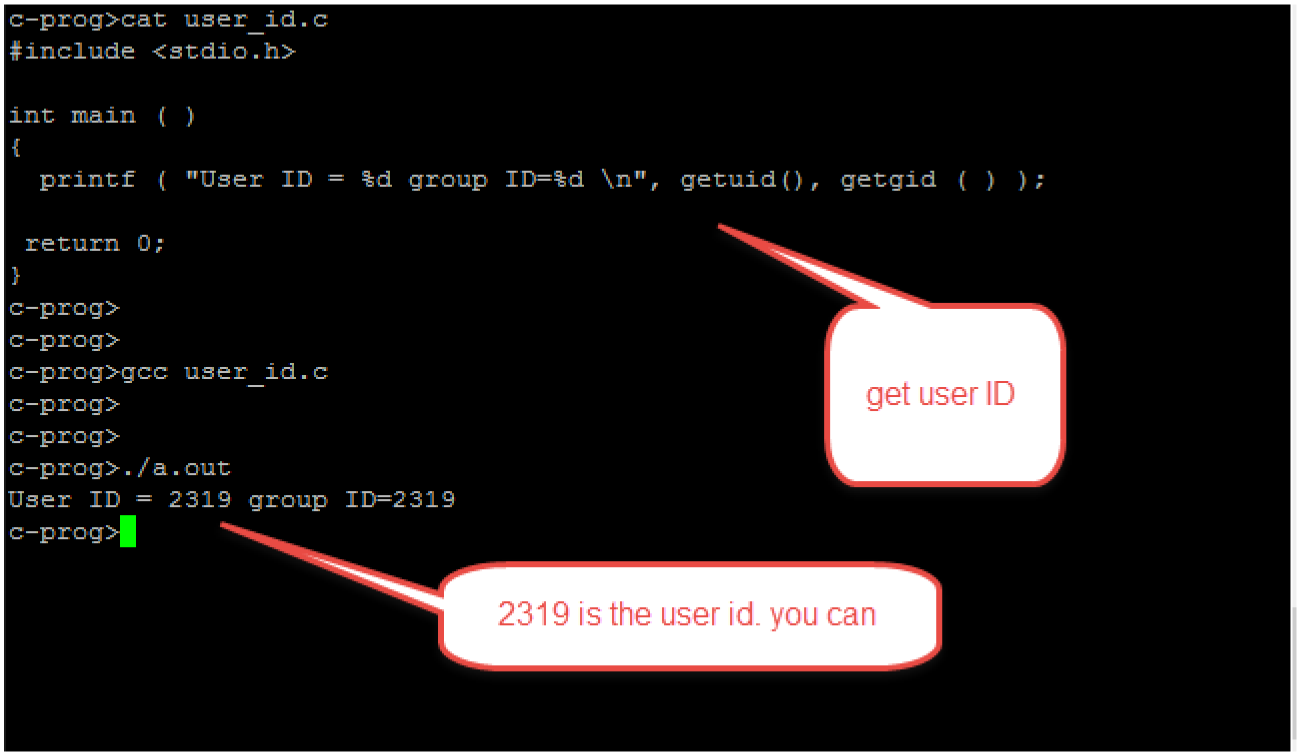
If you open the file, each line looks like this, ssrivatsa:x:2464:2470:Sankar Srivatsa:/home/student/ssrivatsa:/bin/bash

where

ssrivatsa	user loginID
x	indicates presence of a password
2464	user ID
2470	group ID, accounts may belong to a group
full name	seen in the finger command,
/home/student/ssrivatsa	path to the home directory , one can change the default home directory to another directory using usermod command
/bin/bash	default shell program to launch during login

You can get all these information using the command - finger loginid (ex finger ssrivatsa)

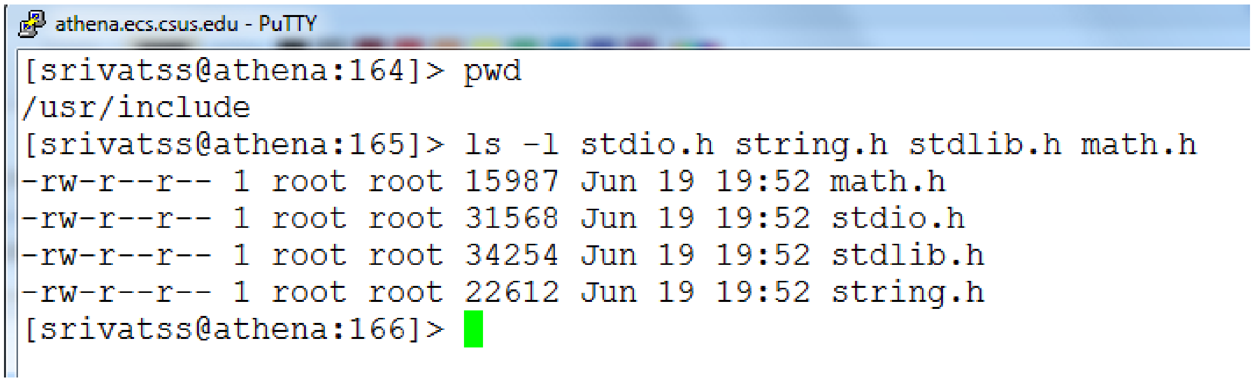
You can get the user id and group id of yours using getuid ( ) and getgid ( ) functions



```
c-prog>cat user_id.c
#include <stdio.h>

int main ( )
{
  printf ( "User ID = %d group ID=%d \n", getuid(), getgid ( ) );

 return 0;
}
c-prog>
c-prog>
c-prog>gcc user_id.c
c-prog>
c-prog>
c-prog>./a.out
User ID = 2319 group ID=2319
c-prog>
```

Where are C library files stored ?

Some of the header files you may use in your programs are stored in /usr/include folder. Navigate to /usr/include



```
[srivatss@athena:164]> pwd
/usr/include
[srivatss@athena:165]> ls -l stdio.h string.h stdlib.h math.h
-rw-r--r-- 1 root root 15987 Jun 19 19:52 math.h
-rw-r--r-- 1 root root 31568 Jun 19 19:52 stdio.h
-rw-r--r-- 1 root root 34254 Jun 19 19:52 stdlib.h
-rw-r--r-- 1 root root 22612 Jun 19 19:52 string.h
[srivatss@athena:166]>
```

Types of Files and Directories:

There are two main types of files : regular files and directories. Technically, directories are stored as files , but we don’t see them as files, rather folders. The various types are

Regular File	The most type of file. There is no distinction between binary or text file. The distinction to be made is left to the application
Directory File	A file that contains the names of other files and pointer to the information on these files. Any process that has read permissions can read the contents of the directory
Block special files	a type of file providing buffered IO to devices such as disk drives
Character file	a type of file providing unbuffered IO.
Socket file	Used for programming network communications (web…)
symbolic files	A type of file that points to another file..
FIFO	used to communicate between processes

Virtual Memory and Physical Memory

The virtual memory is stored as 4K pages , in a linked list data structure - vma_struct.

```
struct vm_area_struct {

       struct mm_struct * vm_mm; /* The address space we belong to. */
       unsigned long vm_start; /* Our start address within vm_mm. */
       unsigned long vm_end; /* The first byte after our end address
within vm_mm. */

      /* linked list of VM areas per task, sorted by address */
     struct vm_area_struct *vm_next;

     pgprot_t vm_page_prot; /* Access permissions of this VMA. */
     unsigned long vm_flags; /* Flags, see mm.h. */


....
}
```

Each page is mapped to a unique base address in the page table. Each process has its own page table which contains the base addresses for each Virtual page. The base address is added to the virtual page to yield the actual physical address. The physical memory is accessed via the data structure mm_struct.

```
struct mm_struct (http://lxr.linux.no/linux+v2.6.28.1/+code=mm_struct) {
    struct vm_area_struct (http://lxr.linux.no/linux+v2.6.28.1/+code=vm_area_struct) * mmap (http://lxr.linux.no/linux+v2.6.28.1/+code=mmap);              /* list of VMAs */
    struct rb_root (http://lxr.linux.no/linux+v2.6.28.1/+code=rb_root) mm_rb (http://lxr.linux.no/linux+v2.6.28.1/+code=mm_rb);
    struct vm_area_struct (http://lxr.linux.no/linux+v2.6.28.1/+code=vm_area_struct) * mmap_cache (http://lxr.linux.no/linux+v2.6.28.1/+code=mmap_cache);      /* last find_vma result */
```

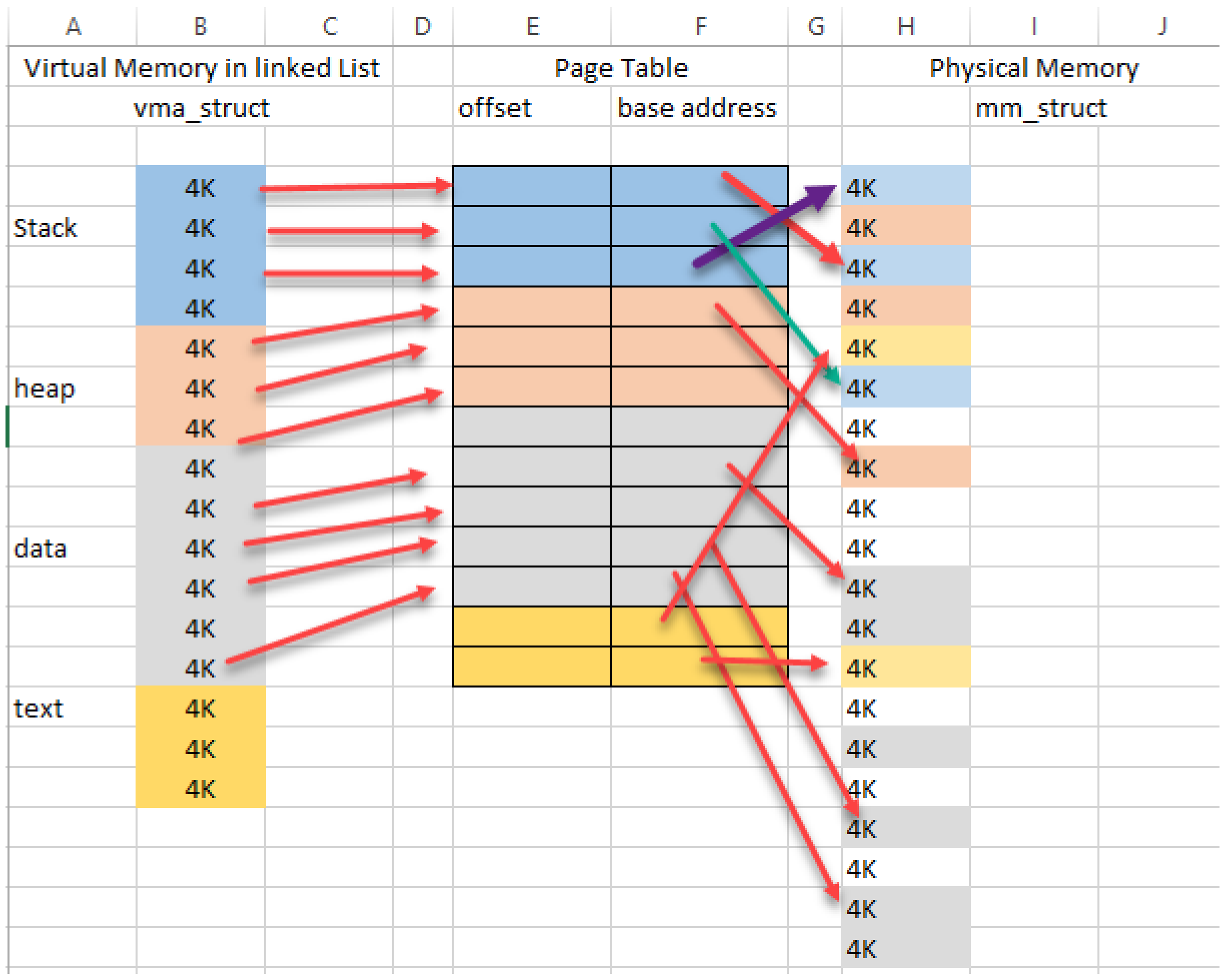
...

} ;

See how the virtual pages are accessed via a pointer mmap

 If for a virtual page is mapped to a base address, but the physical page is not present in RAM, a page fault occurs. The IO manager will have to bring the page from the disk to the physical RAM. But page fault is beyond the scope of this course.

If all RAM is occupied, then page faults occurs a lot losing efficienty and slow performance.



You can view the virtual address in the directory: /proc/process-id/smaps

If your process id is = 14065

then you can do this command to see the virtual pages:

```
cat /proc/14065/smaps
```

What is a.out ?

A executable program such as a.out is a file containing a range of information that describes how to construct a process at run time. This information includes the following:

- Binary format identification : Each program file includes meta information describing the format of the executable file. Most UNIX implementations (including Linux) employ the Executable and Linking Format (ELF). This enables the kernel to interpret information in the file.
- Machine-language instructions: These encode the algorithm of the program.
- Program entry-point address: This identifies the location of the instruction at which execution of the program should commence.
- Data: The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).
- Symbol and relocation tables: These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- Shared-library and dynamic-linking information: The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
- Other information: The program file contains various other information that describes how to construct a process

What is a Process:

A process is an instance of an executing program. Several processes can be running the same program. While running the program, kernel appends additional information required to run the program. Information such as kernel data structures that maintain information about the state of the process. The information recorded in the kernel data structures include various identifier numbers (IDs) associated with the process, virtual memory tables, the table of open file descriptors, information relating to signal delivery and handling (we will talk about Signals soon), process resource usages and limits, the current working directory, and a host of other information. Each running program will have a ID and usually launched by another programs aka parent process. In our system and in all systems, the init process (aka Kernel) has ID = 1 and all processes will be child of the his process.

Process ID and parent process ID

Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process.

#include <unistd.h>

pid_t getpid(void);	Returns: process ID of calling process
pid_t getppid(void);	Returns: parent process ID of calling process

uid_t getuid(void);	Returns: real user ID of calling process
uid_t geteuid(void);	Returns: effective user ID of calling process
gid_t getgid(void);	Returns: real group ID of calling process
gid_t getegid(void);	Returns: effective group ID of calling process

With the exception of a few system processes such as init (whose process ID is always 1), process ID will very different next time you run the program. The Linux kernel limits process IDs to being less than or equal to 32,767. When a new process is created, it is assigned the next sequentially available process ID. Each time the limit of 32,767 is reached, the kernel resets its process ID counter so that process IDs are assigned starting from low integer values.

Unix Commands

Processes

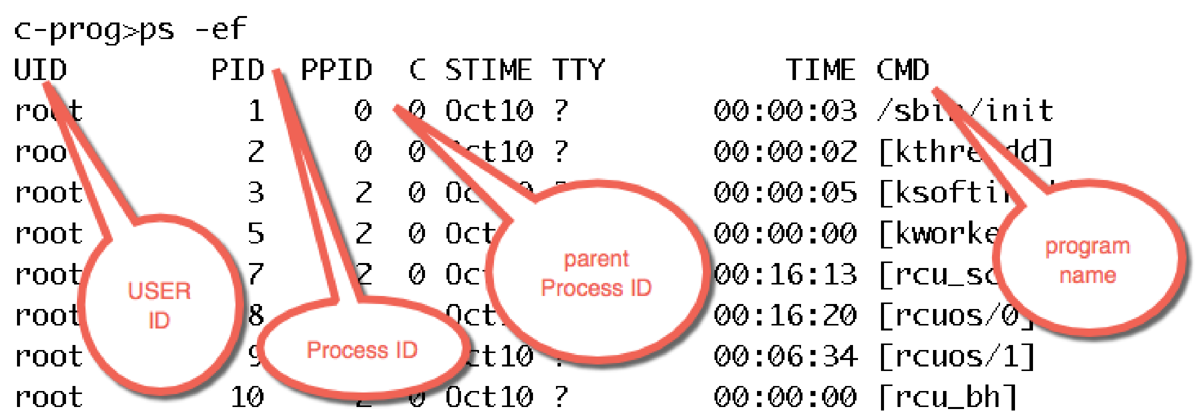
A program is a set of instructions in passive state stored in a file. A process executes this program in active state and is executed by a processor. Every process in Linux gets a process ID.

The process ID of a process can be printed using the system function getpid ( ) and the parent process ID as getppid ( ) . You can also print the process ID with the command

ps -ef

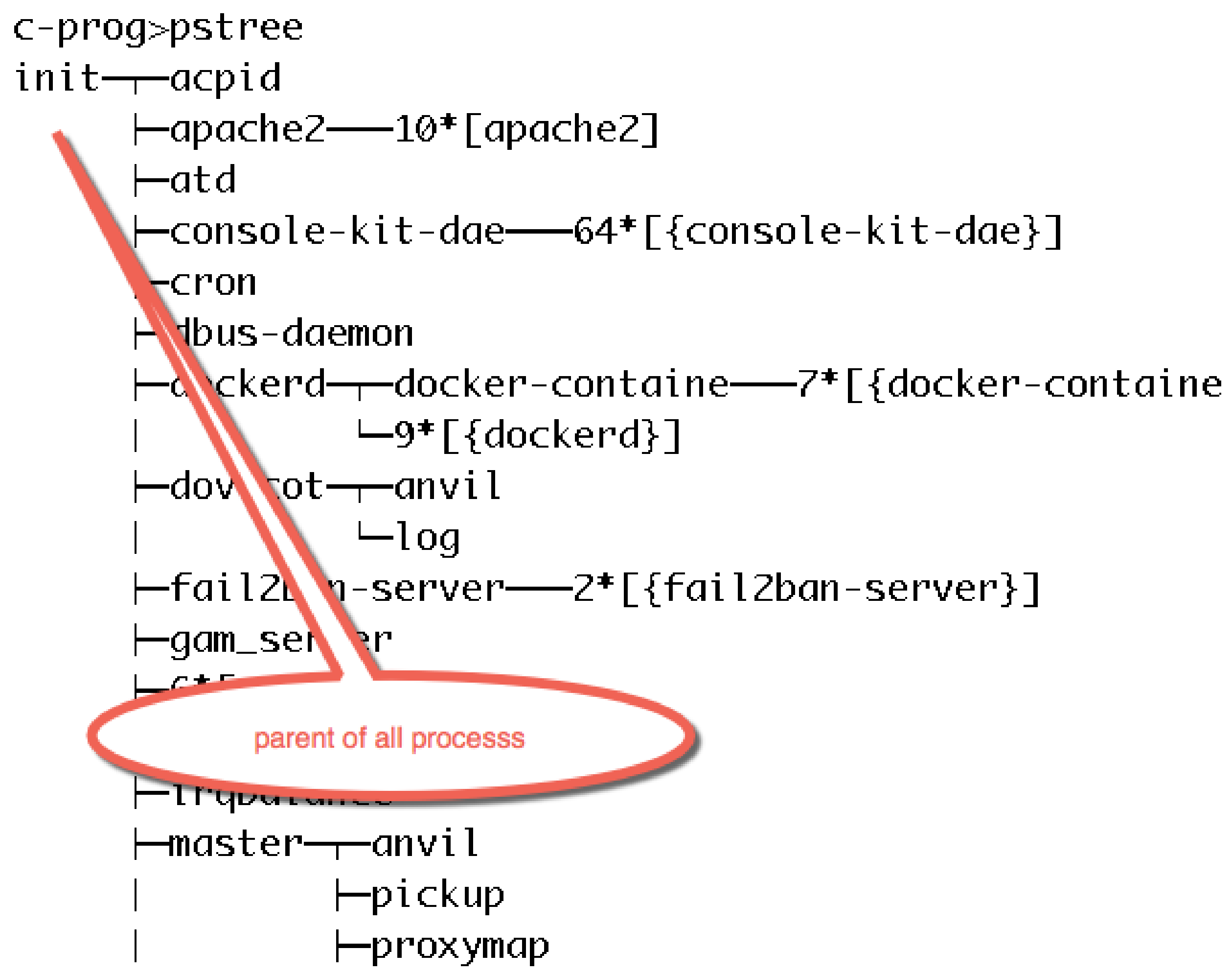
Every process is a child of another process , except the init process.

see the screen shot



```
c-prog>ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
ro t         1     0    Oct10 ?        00:00:03 /sbi /init
roo          2     0  0   t10 ?        00:00:02 [kthre dd]
root         3     2  0 Oc             00:00:05 [ksofti
root         5     2  0 Oct            00:00:00 [kworke
root         7     2  0 Oc             00:16:13 [rcu_sc
root         8            Oct          00:16:20 [rcuos/0
root         9              t10        00:06:34 [rcuos/1]
root        10          0 Oct10 ?      00:00:00 [rcu_bh]
```

You can view the processes in a tree structure using the command pstree command.

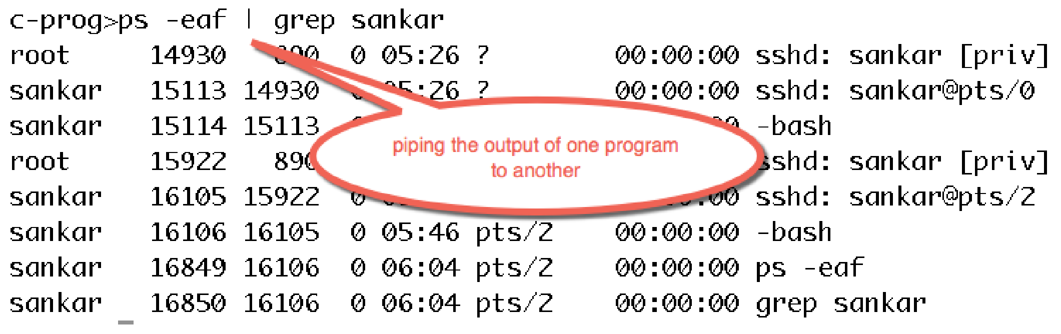


```
c-prog>pstree
init─┬─acpid
     ├─apache2───10*[apache2]
     ├─atd
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─dbus-daemon
     ├─dockerd─┬─docker-containe───7*[{docker-containe
     │         └─9*[{dockerd}]
     ├─dovecot─┬─anvil
     │         └─log
     ├─fail2ban-server───2*[{fail2ban-server}]
     ├─gam_server
     ├─6*[[illegible]
     [illegible]
     ├─irqbalance
     ├─master─┬─anvil
     │        ├─pickup
     │        ├─proxymap
```

You can view the processes created by you, by typing

```
c-prog>ps -u <USERID>
  PID TTY          TIME CMD
15113 ?        00:00:00 sshd
15114 pts/0    00:00:00 bash
16105 ?        00:00:00 sshd
16106 pts/2    00:00:00 bash
16913 pts/2    00:00:00 ps
```

or you could view in a long format using the pipe command to ps -eaf



```
c-prog>ps -eaf | grep sankar
root      14930   890  0 05:26 ?        00:00:00 sshd: sankar [priv]
sankar    15113 14930  0 05:26 ?        00:00:00 sshd: sankar@pts/0
sankar    15114 15113                        0 -bash
root      15922   89                          shd: sankar [priv]
sankar    16105 15922  0                   :00 sshd: sankar@pts/2
sankar    16106 16105  0 05:46 pts/2    00:00:00 -bash
sankar    16849 16106  0 06:04 pts/2    00:00:00 ps -eaf
sankar    16850 16106  0 06:04 pts/2    00:00:00 grep sankar
```

When a process is executed by the CPU, the process accesses all registers and RAM. Because there are several programs share CPU time, Operating System uses a scheduling algorithm (a popular being Round robin) which gives equal time slice to all processes and processes may have different priority to run. Some processes may have higher priority than others. When a process is given the CPU time, it might finish execution during the time slice it is allocated or it may be swapped out with another process that is next in the pipeline. This is called preempted.

Try this command

top

The program that is running is flagged as R and the programs in waiting/sleeping flagged as S. You quit the top program by typing q .

Memory Layout of a C Program

A program may contain global variables (initialized and uninitialized) and functions. When you launch your program, it is loaded into memory . While running, our program can be swapped/switched with another, and we don't want to lose the status of our program. To store the status of our program, our program is compartmentalized into various sections. We can get the size of each of these sections using the command

size a.out <enter>

Let us examine the sections.   The various sections are given below

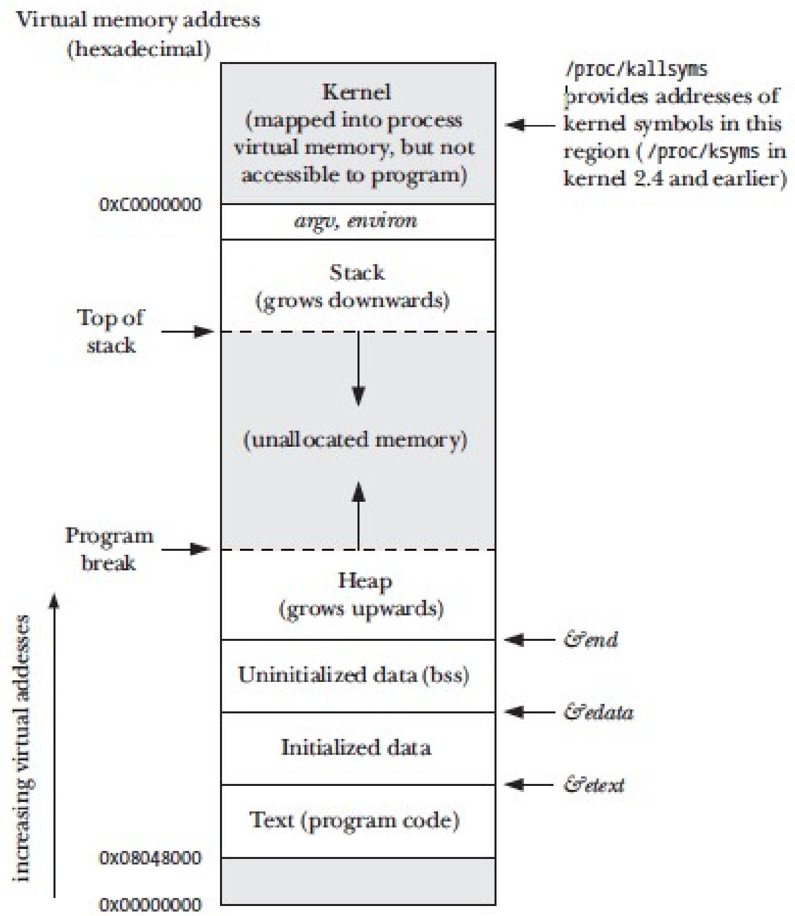

Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
increasing virtual addesses
0x08048000
0x00000000

Figure 6-1: Typical memory layout of a process on Linux/x86-32

Text Area: The text segment contains the machine-language instructions of the program run by the process. The text segment is made read-only so that a process doesn't accidentally modify its own instructions via a bad pointer value. Since many processes may be running the same program, the text segment is made sharable so that a single copy of the program code can be mapped into the virtual address space of all of the processes.

Stack Area: This section is meant for functions. When your program calls functions, the instructions, local variables, return address and other information is loaded in this area.

Heap Area: If your programs allocates dynamic memory, this memory is allocated in the heap area. As your program allocates more memory, all this is allocated in the heap area. Managing this area is very complicated.

Stack vs Heap: Management of this stack area is little easier than Heap area because sometime your program will deallocate the dynamic memory in heap causing gaps resulting in memory gaps. Sometimes, your program may not deallocate the dynamic memory resulting in memory leaks. The stack area grows linearly from top to bottom and management of this memory is little easier.

BSS Area:

This uninitialized data segment contains global and static variables that are not explicitly initialized. Before starting the program, the system initializes all memory in this segment to 0. For historical reasons, this is often called the bss segment, a name derived from an old assembler mnemonic for "block started by symbol." The main reason for placing global and static variables that are initialized into a separate segment from those that are uninitialized is that, when a program is stored on disk, it is not necessary to allocate space for the uninitialized data. Instead, the executable merely needs to record the location and size required for the uninitialized data segment, and this space is allocated by the program loader at run time

Data section : This initialized data segment contains global and static variables that are explicitly initialized. The values of these variables are read from the executable file when the program is loaded into memory.

We can more information about these segments using command

```
objdump —-f -h a.out
```

We will write several versions of a simple program adding variables in each version and check how the sections of the code varies in size.   Here is the table , the left column is our program, the center column summarizes the changes we made, the right column shows the size of Text, Data and BSS sections. The command to get the size of these sections is size a.out

Summary of our work		
Program : mem.c	Type of variables in the program	size a.out<enter>

#include <stdio.h>
int main ( )
{
}
	no variables.	text data bss
		1076 496 16

#include <stdio.h>
int age = 20;
int main ( )
{
}
	one global variable initialized.	text data bss
		1076 500 16

#include <stdio.h>
int age = 20;
int myAge ;
main ( )
{
}
	one global variable initialize
	one global variable uninitialized
		text data bss
		1076 500 24

#include <stdio.h>
int age = 20;
int myAge;
main ( )
{
int mainAge = 30;
}
	one global variable initialize
	one global variable uninitialized
	one local variable
		text data bss
		1092 500 24

Global Variables : etext, edata and end

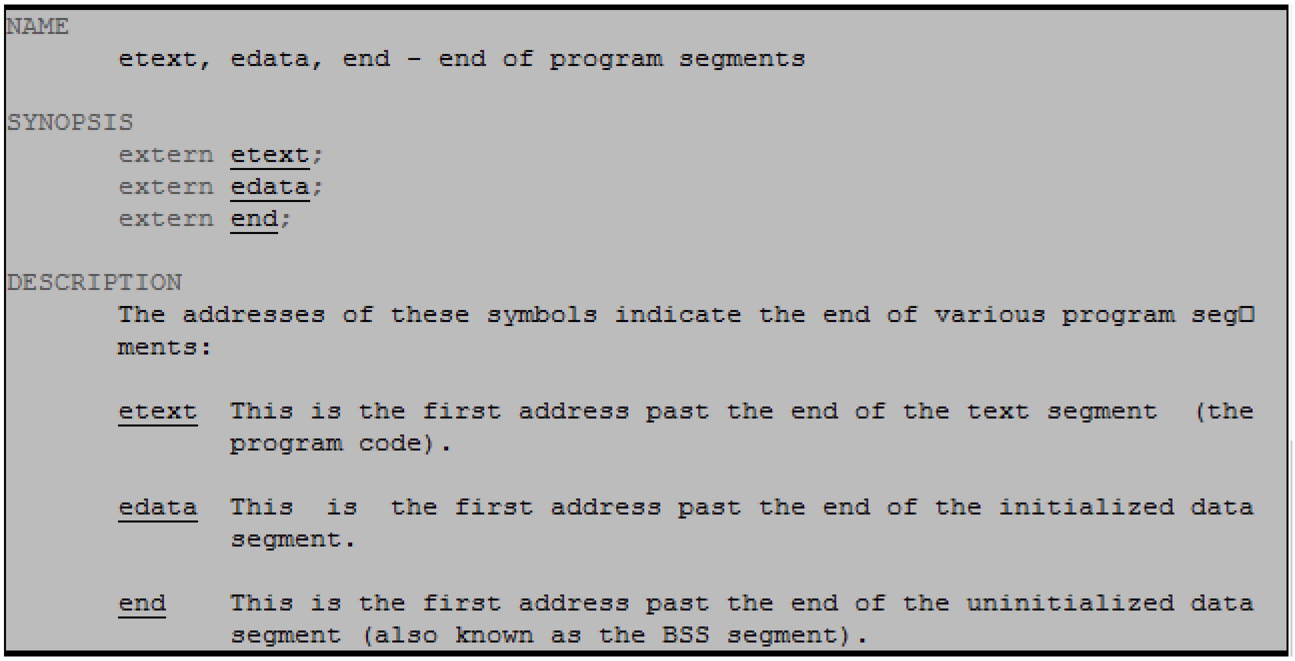
Most UNIX implementations (including Linux) provides three global symbols: etext , edata , and end . These symbols can be used from within a program to obtain the addresses of the next byte past, respectively, the end of the program text, the end of the initialized data segment, and the end of the uninitialized data segment. To make use of these symbols, we must explicitly declare them, as follows:

.

#include <stdio.h>
extern etext, edata, end ;
main ( )
{
printf ("End of Text segment %10p \n", &etext);
printf ("End of Data segment or start of BSS %10p \n", &edata);
printf ("End of BSS data %10p \n", &end);
}
	/* &etext gives the address of the end of the program text / start of initialized data */

You can get information about these variables using man command

```
man end
```



```
NAME
       etext, edata, end - end of program segments

SYNOPSIS
       extern etext;
       extern edata;
       extern end;

DESCRIPTION
       The addresses of these symbols indicate the end of various program seg□
       ments:

       etext  This is the first address past the end of the text segment  (the
              program code).

       edata  This  is  the first address past the end of the initialized data
              segment.

       end    This is the first address past the end of the uninitialized data
              segment (also known as the BSS segment).
```

when you run this program

a.out<enter>

the output is

```
End of Text segment   0x400626

End of Data Segment or start of BSS    0x601020

End of BSS data   0x601030
```

Virtual Memory Management:

All the addresses we print in our programs are virtual address, ie they are not real physical addresses. Linux employs a technique known as virtual memory management. The aim of this technique is to make efficient use of both the CPU and RAM (physical memory). A virtual memory scheme splits the memory used by each program into small, fixed-size units called pages . Correspondingly, RAM is divided into a series of page frames of the same size. At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages form the so-called resident set . Copies of the unused pages of a program are maintained in the swap area —a reserved area of disk space used to supplement the computer's RAM—and loaded into physical memory only as required. When a process references a page that is not currently resident in physical memory, a page fault occurs, at which point the kernel suspends execution of the process while the page is loaded from disk into memory. On x86-32, pages are 4096 bytes in size. We can get the page size in our system using the program here

```
#include <stdio.h>
#include <unistd.h>
int main ( )
{
long sz = sysconf(_SC_PAGESIZE);
 printf ( "page size = %ld \n", sz );

}
c-prog>gcc pagesize.c
c-prog>./a.out
page size = 4096
```

In our system, the page size is 4K.

How are virtual addresses mapped to physical addresses then ?
The kernel maintains a page table for each process (see the Figure below). The page table describes the location of each page in the process's virtual address space (the set of all virtual memory pages available to the process). Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.

Virtual memory management separates the virtual address space of a process from the physical address space of RAM. This provides many advantages :

- Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel. This is accomplished by having the page-table entries for each process point to distinct sets of physical pages in RAM (or in the swap area).
- Where appropriate, two or more processes can share memory. The kernel makes this possible by having page-table entries in different processes refer to the same pages of RAM. This could happen when we fork a process, the child and parent may share the same pages until one of them updates the memory, then the copy of the memory is created , this is copy-on-write concept.
- The implementation of memory protection schemes is facilitated; that is, pagetable entries can be marked to indicate that the contents of the corresponding page are readable, writable, executable, or some combination of these protections.
- Where multiple processes share pages of RAM, it is possible to specify that each process has different protections on the memory; for example, one process might have read-only access to a page, while another has read-write access.
- Programmers, and tools such as the compiler and linker, don't need to be concerned with the physical layout of the program in RAM.
- Because only a part of a program needs to reside in memory, the program loads and runs faster.
- One final advantage of virtual memory management is that since each process uses less RAM, more processes can simultaneously be held in RAM. This typically leads to better CPU utilization, since it increases the likelihood that, at any moment intime, there is at least one process that the CPU can execute.

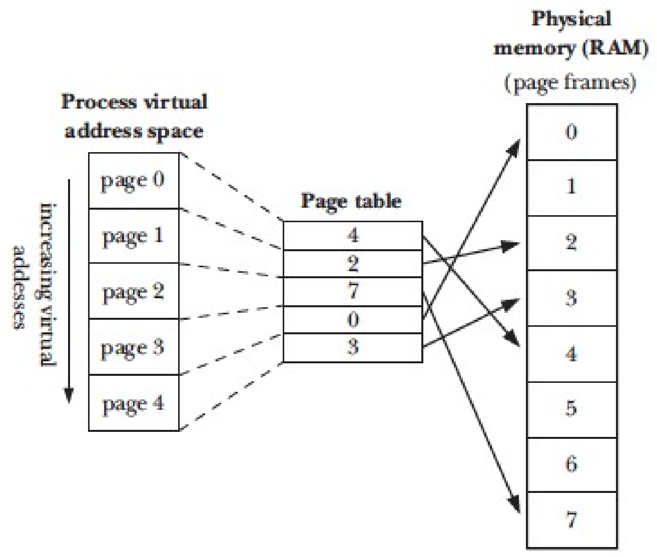


Figure 6-2: Overview of virtual memory

SYSTEM CALL

User vs Kernel Space :

Before we talk about system call operations, we need to understand that most CPUs typically operate in at least two different modes: user mode and kernel mode (sometimes also referred to as supervisor mode). When running in user mode, the CPU can access only memory that is marked as being in user space; attempts to access memory in kernel space result in an exception. For example, a code

```
int x= 0;

scanf ( "%d", x) ;
```

will result in exception because the user program is trying to assign a value to address 0 which may be in kernel space. The actual code should be of course

```
scanf ( "%d", &x) ;
```

This ensures that user processes are not able to access the instructions and data structures of the kernel, or to perform operations that would adversely affect the operation of the system. When running in kernel mode, the CPU can access both user and kernel memory space.

System Calls A system call is a controlled entry point into the kernel, allowing a process to request the kernel perform some action on the process's behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication. (The syscalls(2) manual page lists the Linux system calls and in our linux system the /usr/include/asm/unistd_32.h) Before going into the details of how a system call works, some notable points are :

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each system call is identified by a unique number. (This numbering scheme is not normally visible to programs, which identify system calls by name.) Each number is nothing but an array index to a data structure called System Call Table. Luckily, we don't have to worry about the numbers, we only have to know the system call name and parameters.
- Before kernel is starting to execute the system call, the user program may have a set of arguments that specify information to be transferred from user space to kernel space and vice versa.

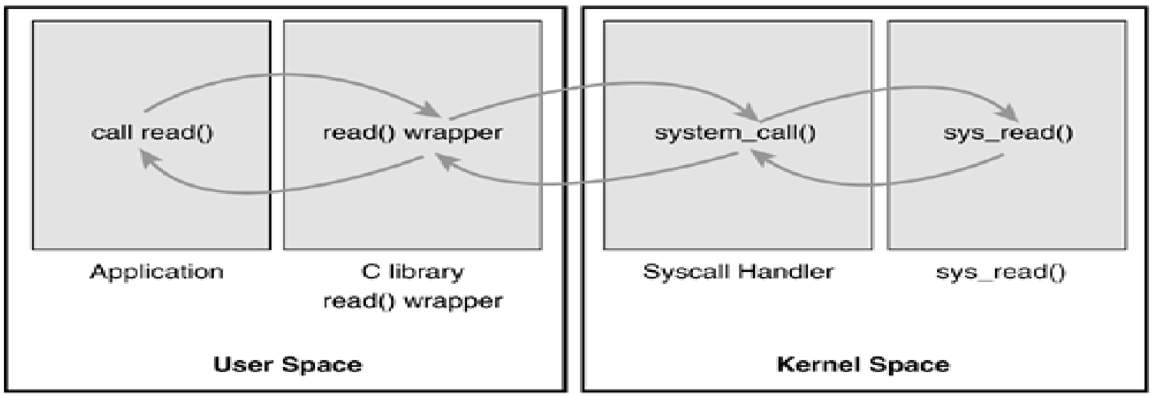
From a programming point of view, invoking a system call looks much like calling a C function. However, behind the scenes, many steps occur during the execution of a system call.

syscall ( ) function - How does the system execute library calls in kernel space ?

From a programming point of view, invoking a system call looks much like calling a C function. We know a function has a name and may have a bunch of input parameters. In the system, each system function name is mapped to a unique number. We will see how this works :

IN USER SPACE:

1. Because we cannot call the system function directly, a wrapper function is given to us. The application program makes a system call by invoking this wrapper function indirectly in the C library.



2. Because Kernel expects all arguments of the function in specific registers in the CPU, the wrapper function copies the arguments to these registers. These registers are mentioned below

Architecture	arg1	arg2	arg3	arg4	arg5	arg6
i386	ebx	ecx	edx	esi	edi	ebp
x86_64	rdi	rsi	rdx	r10	r8	r9
x32	rdi	rsi	rdx	r10	r8	r9

3. The wrapper function then copies the unique number onto a specific CPU register eax

4. Lastly, the wrapper function executes a trap machine instruction 0x80 which causes the processor to switch from user mode to kernel mode. New architectures use syscall. The program now in kernel space.

IN KERNEL SPACE :

5. Now in kernel space, in response to the trap to location 0x80, the kernel invokes its system_call() routine to execute the function on behalf of the user program.

This routine :

a) Saves register values onto the kernel stack. That moves data to its internal memory.

b) VALIDATES THE FUNCTION CALL: Checks the validity of the system call number.

c) VALIDATES THE ARGUMENTS: If the system call service routine has any arguments, it first checks their validity.

d) Invokes the system call service routine mapped to the unique number.

e) Once the routine finishes, the service routine returns

d) Restores register values from the kernel stack and places the system call return value on the stack.

e) Returns to the wrapper function, simultaneously returning the processor to user mode.

6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable errno using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Here is the simple C program :

```
int main ( )

{

 printf ( " Hello World \n"  ) ;  gets converted into Syscall by the compiler

 return 0 ;

}
```

The assembly code for the above C code is something like this:

```
.global _start
   .text
_start:
   # write(1, message, 13)
        mov     $4, %eax                # system call 4 is write
        mov     $1, %ebx                # file handle 1 is stdout
        mov     $message, %ecx          # address of string to output
        mov     $13, %edx               # number of bytes to write
        int     $0x80                   # invoke operating system code

        # exit(0)
        mov     $1, %eax                # system call 1 is exit
        xor     %ebx, %ebx              # we want return code 0
        int     $0x80                   # invoke operating system code
message:
        .ascii  "Hello, World\n"
```

ERROR HANDLING

When a system function encounters an error, it notifies the caller about the error using a negative value -1. There is no additional information about the error. But additional information can be viewed through the global integer variable errno, and is usually set by the system call to a constant value that gives additional information. Note: We don't set this value, it is set by the system.

On Linux, the error constants are listed in the errno manual page. You can also view the values in the header file <errno.h>. Valid error numbers are all nonzero; errno is never set to zero by any system call or library function.

errno is defined by the ISO C standard to be a modifiable lvalue of type int, and must not be explicitly declared like this;

```
extern int errno;
```

The value returned is valid only immediately after an errno-setting function indicates an error (usually by returning -1). Developers should quickly process this error and make suitable action. Because the variable can be modified during the successful execution of another or this function.

The errno variable may be read or written directly; it is a modifiable lvalue. The value of errno maps to the textual description of a specific error. A preprocessor #define also maps to the numeric errno value. For example, the preprocessor define EACCESS equals 1

Some examples of the description mapped to errno are

EACCES	Permission denied (POSIX.1)
EISDIR	Is a directory (POSIX.1)
ENAMETOOLONG	Filename too long (POSIX.1)

The C library provides a handful of functions for translating an errno value to the corresponding textual representation. This is needed only for error reporting, and the like; checking and handling errors can be done using the preprocessor defines and errno directly.

The two functions we need to be aware are

```
        #include <string.h>
```

1. char *strerror(int errnum);

Returns: pointer to message string

This function takes the errnum argument, which is typically the errno value and returns a pointer to the string.

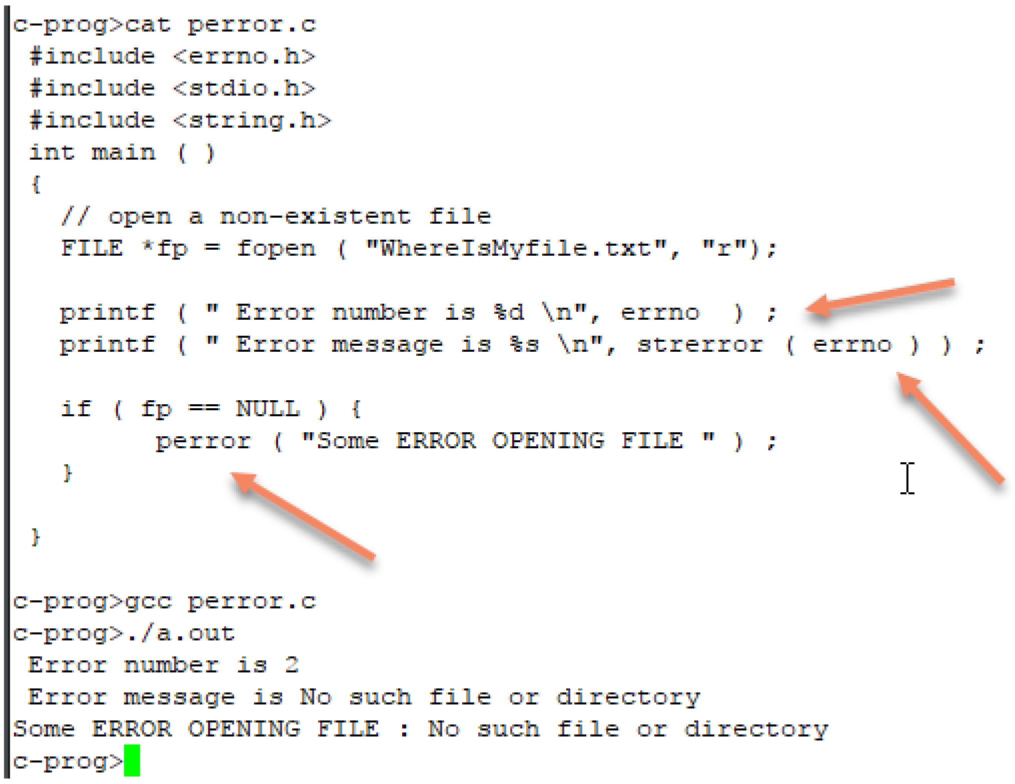
The other function , perror function produces an error message on the standard error, based on the current value of errno, and returns.

#include <stdio.h>

2. void perror(const char *msg) ;

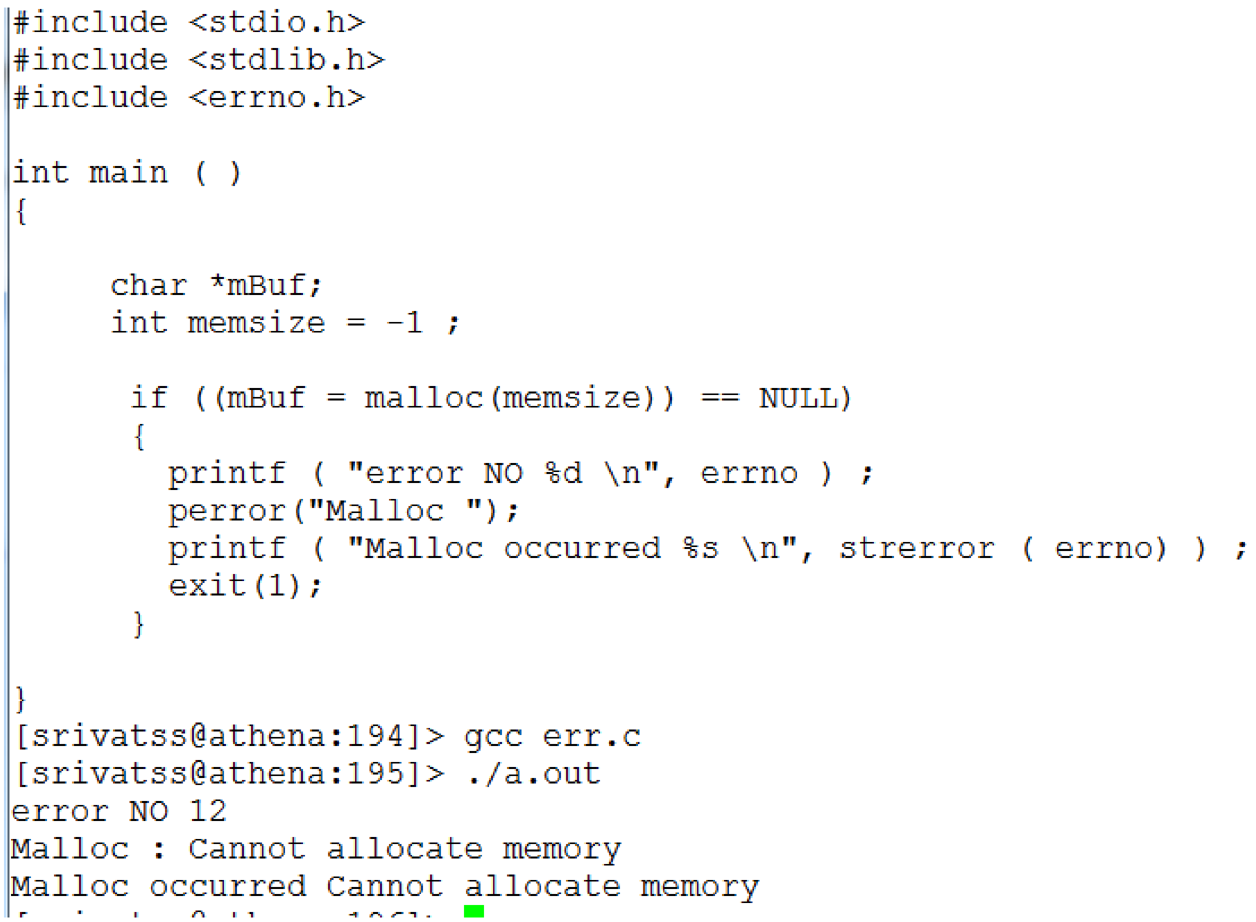
This outputs the string (usually passed by the application) pointed to by msg, followed by a colon and a space, followed by the error message corresponding to the value of errno, followed by a newline.

Here is the sample output



```
c-prog>cat perror.c
 #include <errno.h>
 #include <stdio.h>
 #include <string.h>
 int main ( )
 {
   // open a non-existent file
   FILE *fp = fopen ( "WhereIsMyfile.txt", "r");

   printf ( " Error number is %d \n", errno  ) ;
   printf ( " Error message is %s \n", strerror ( errno ) ) ;

   if ( fp == NULL ) {
         perror ( "Some ERROR OPENING FILE " ) ;
   }

 }

c-prog>gcc perror.c
c-prog>./a.out
 Error number is 2
 Error message is No such file or directory
Some ERROR OPENING FILE : No such file or directory
c-prog>
```

Another example is :



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

int main ( )
{

     char *mBuf;
     int memsize = -1 ;

      if ((mBuf = malloc(memsize)) == NULL)
      {
        printf ( "error NO %d \n", errno ) ;
        perror("Malloc ");
        printf ( "Malloc occurred %s \n", strerror ( errno) ) ;
        exit(1);
      }

}
[srivatss@athena:194]> gcc err.c
[srivatss@athena:195]> ./a.out
error NO 12
Malloc : Cannot allocate memory
Malloc occurred Cannot allocate memory
```