

**University of Massachusetts Lowell**  
*One University Avenue*  
*Lowell, MA 01854*

College of Arts and Science  
Department of Computer Science

Prof. William F Moloney, Executive Officer  
Tel: (508) 934-3640  
bill@cs.uml.edu

**UNIX Systems Programming**

# UNIX Systems Programming

The topics covered in this course include the following:

- Generic UNIX organization
- System address space, process address space
- General C function calls, kernel access
- C binding format for system calls
- Process attributes and protection mechanisms
- Categories of system services
- I-O services
- File systems and INODES, internal organization
- Device independent I-O system calls
- Advanced I-O and terminal control
- Control operations on INODES
- Line driver discipline
- `ioctl` with terminal drivers
- Processes and process management
- Process states and transitions
- Managing process attributes
- Process scheduling algorithm
- Advanced process control

## UNIX Systems Programming (Cont'd)

- Basic UNIX signal mechanisms
- Global jumps and state saving
- Basic interprocess communications
- Using named pipes (FIFOs)
- System V advanced IPC mechanisms
- Generalized message passing
- Sharing physical memory among processes
- Generalized semaphore facility
- The internetwork and TCP/IP
- Generalized connection mechanism
- Datagrams and unconnected sockets
- System V driver organization
- Block, character and streams overview
- Upper and lower half functionality
- Streams implementation issues

## The UNIX System

The UNIX operating system was created at Bell Laboratories in 1969 by Ken Thompson and Dennis Ritchie. Much of their work reflected Bell's experience with Project Mac from 1965 to 1969. Project Mac was a joint effort by Bell Labs, General Electric and MIT to define a time-sharing operating system. When Bell withdrew from the project in 1969, they were left without a system (they had been using a GE specially modified system) or an operating system. Later that year using a cast-off PDP-7, Thompson and Ritchie wrote UNIX.

UNIX remained as a Bell internal software package through the first 5 years of its life, but a number of significant events occurred during the years from 1970 to 1975. The C language was constructed by Thompson and Ritchie of Bell Laboratories during the early 1970s in an effort to provide a portable language platform for their UNIX operating system. Both UNIX and C were implemented on the early PDP-11 series of systems from DEC, and, in 1975, Bell Laboratories decided to provide the software to non-profit organizations for a small media charge.

## The UNIX System (Cont'd)

When the first DEC Vax machines showed up in 1977 UNIX was quickly ported and became the system of choice within Universities:

- Originally described in 1974 CACM as a general purpose, multi-user, interactive operating system
- The dispatch-able unit is the process, an asynchronous activity
- a compact kernel provides classic multiprocessing support including:
  - dispatching, context switching and state transition
  - interprocess synchronization and communication
  - memory management
  - a device independent i-o system
  - a simple hierarchical stream file system
- A small collection of kernel system calls provide support access directly to the kernel for application programs
- Interactive user interface to the kernel through a type of program called a shell (several exist)

## The UNIX Philosophy

- The basic idea is that that power of a computing system is derived more from the relationship among the programs than the programs themselves
- To extract maximum utility from the UNIX environment you must understand not only how to use the programs provided, but how they fit into the environment
- A UNIX system provides an environment to support the basic principles of software engineering and incremental development by supplying a collection of tailored tools
- The support of team development and resource sharing is also an integral part of the UNIX philosophy

## The UNIX Environment

- UNIX is a classic multiprocessing system
- The existence of multiple processes supports:
  - *Timesharing* , or process multiplexing to share resources
  - *Pipelining* , or coroutine construction to use resources efficiently
  - *Concurrent Processing* , or multiple threads of control for applications which need to synchronize on multiple external events

## The Process Model

Programs execute within a process. A process can best be thought of as a collection of system resources that the operating system views as a unit of management:

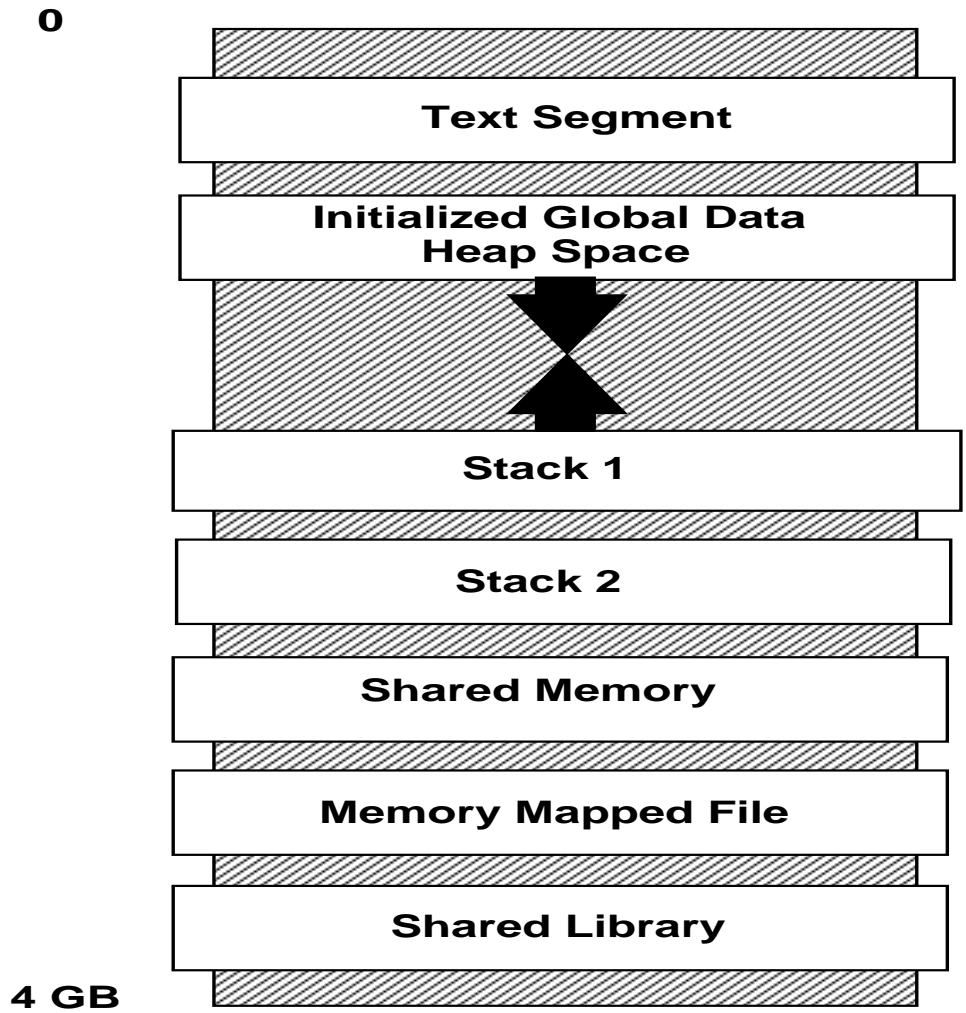
- Computational progress in the system can only be made by a process
- Kernel procedures must be driven by a process
- Such procedures are activated by a process when:
  - the process explicitly requests kernel support by making a system call
  - the process is coerced to execute kernel procedures as a result of an exception of some variety which occurs on the CPU that the process is currently running on

## The Process Model (Cont'd)

Among the resources of a process is an address space and a collection of contiguously addressed objects which occupy some part of this space. The objects typically found on a UNIX system include:

- A Text segment which contains machine instructions
- A Global Data segment which contains static and dynamic global memory allocations (the dynamic allocations are said to belong to the *heap* portion of this segment)
- One or more Stack segments (one per execution path/thread) to manage activation records
- Additional objects which may have been inherited from a parent or mapped dynamically by the process at run time, including:
  - Shared Memory regions
  - Memory Mapped file object regions
  - Shared Library regions
  - Dynamically Mapped Library regions

# Typical Process Address Space

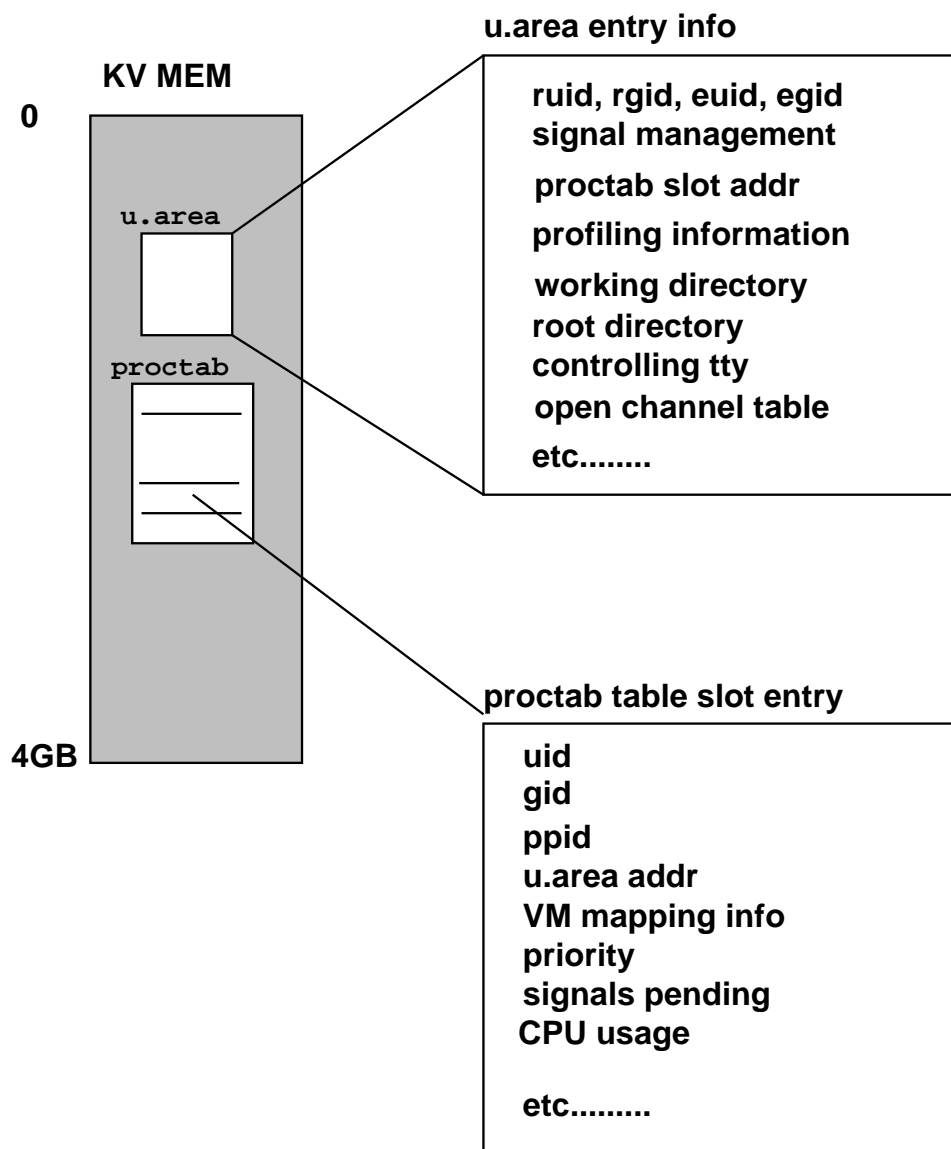


## The Process Model (Cont'd)

A process also contains a *system segment* , but this segment lives in the kernel's address space. It is composed of two components known as the *u.area* and the *proctab* entry. While the process has direct address manipulation privileges on the text, code, etc. segments, the system segment is not directly part of the process space, and only part of it is accessible to the process, and then only by using system calls:

- The process accessible part of the system segment is called the *u.area*
- Among other pieces of process status which are kept here, the process I-O channel table resides in the *u.area*
- All I-O system calls require a channel number as a vector into this table to read or write (or otherwise access) an external file object
- When a process is associated with a control terminal (as a login shell process is) channel 0 (slot 0 in the table) is assigned as standard input and connected to the terminal keyboard while channel 1 is assigned as standard output and is assigned to the display

# The Process System Segment

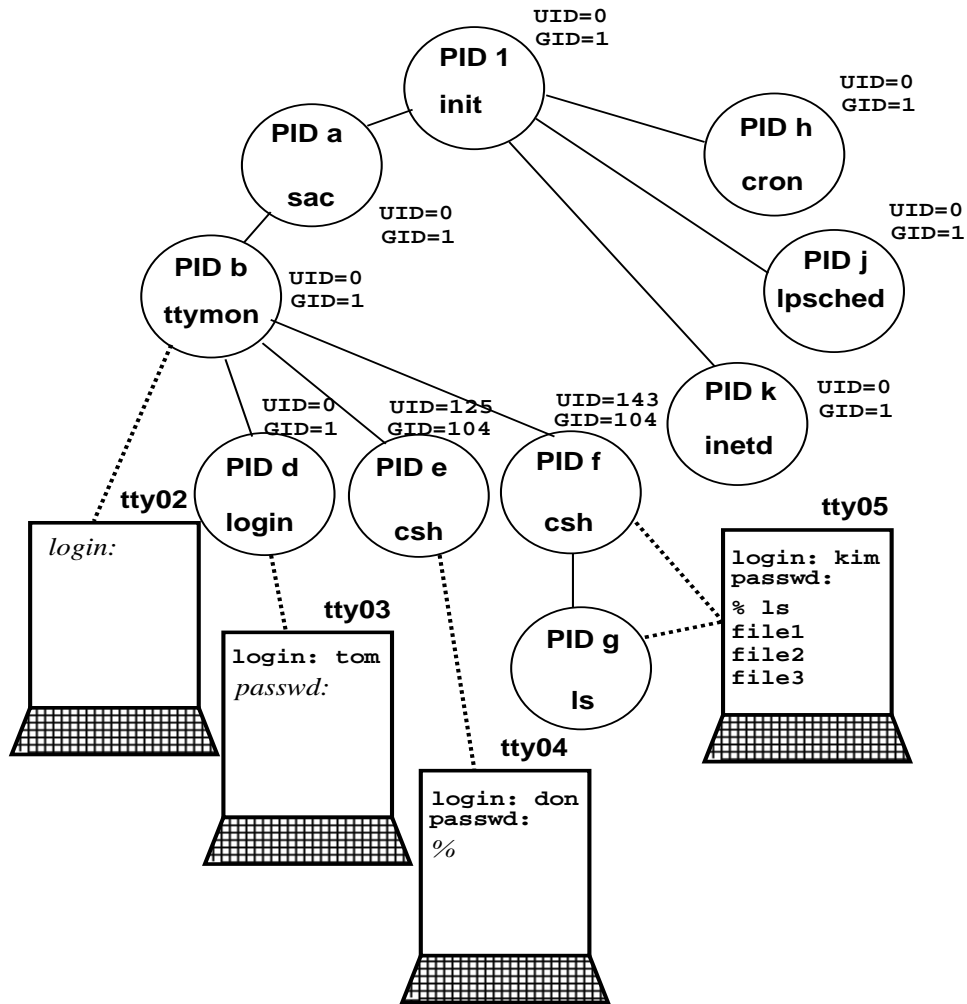


## The Process Model (Cont'd)

At system boot time, the UNIX kernel may create one or more kernel processes (threads) to support the environment. Such processes are generally of little direct interest to a user of the system, but one of them, the so-called *init* process, will serve as the ancestral forefather for all user processes which can ever exist in the system.

The process which runs the *init* program is allocated the special process ID (*PID*) of 1. This *PID* is treated in a unique way in all UNIX systems, and upon exit, will bring the system down. In USL System Five (SV) environments, the *init* process comes to life and looks for a file called **/etc/inittab** to determine how to set up the system for the boot level indicated by the system administrator upon start-up. The *inittab* file generally directs the *init* process to create children to support various system needs such as user access (i.e. login), printer scheduling, network monitoring, etc.

# The Process Hierarchy



```

root:-pw-:0:1: super user :/usr/bin/sh
tom:-pw-:117:104: Tom Mix x234 :/usr/home/tom:/usr/bin/csh
don:-pw-:125:104: Don Ho x567 :/usr/home/don:/usr/bin/csh
kim:-pw-:143:104: Kim Novak x890 :/usr/home/kim:/usr/bin/csh
  
```

## Objects and Protection

Processes are the only active objects in the UNIX world. They may attempt to access other processes or one of the passive objects which include file objects, message queues, shared memory segments and semaphores:

- Processes
- File Objects
  - ordinary
  - directory
  - character device special
  - block device special
  - named pipes (FIFOs)
  - symbolic link
  - UNIX domain socket
- Message queues
- Shared memory segments
- Semaphores

## Objects and Protection (Cont'd)

All UNIX objects have an identity which consists of the user-ID (*UID*) and group-id (*GID*) of the process (if the object is a process) or the UID and GID of the creating process if the object is one of the passive types. The passive objects can only exist if a process creates them, and at creation time their controlling structures are stamped with the creator's UID, GID pair. Access to a process requires that the initiator have an *effective* or *real* UID of the target process (or is running with  $\text{EUID} = 0$ ). Sending a *signal* is an example of this type of access. Access to all other objects proceeds according to the protection domain which an accessing process occupies with respect to the target:

- Protection domains:
  - **owner** initiator UID matches target UID (no further checking done)
  - **group** initiator GID matches target GID iff UIDs don't match
  - **other** neither a UID nor a GID match occurs

## Objects and Protection (Cont'd)

General read, write and execute access is maintained for objects with the obvious interpretation. For directories however, the execute privilege indicates whether a user can use the name in a pathname, and without this privilege, read and write are irrelevant. Read and execute on a directory allow the *ls* command, and write and execute allow the user to create or delete to/from the directory. The set UID and set GID pertain to ordinary binary or script executables and allow the executing process to change its *effective* identity as discussed:

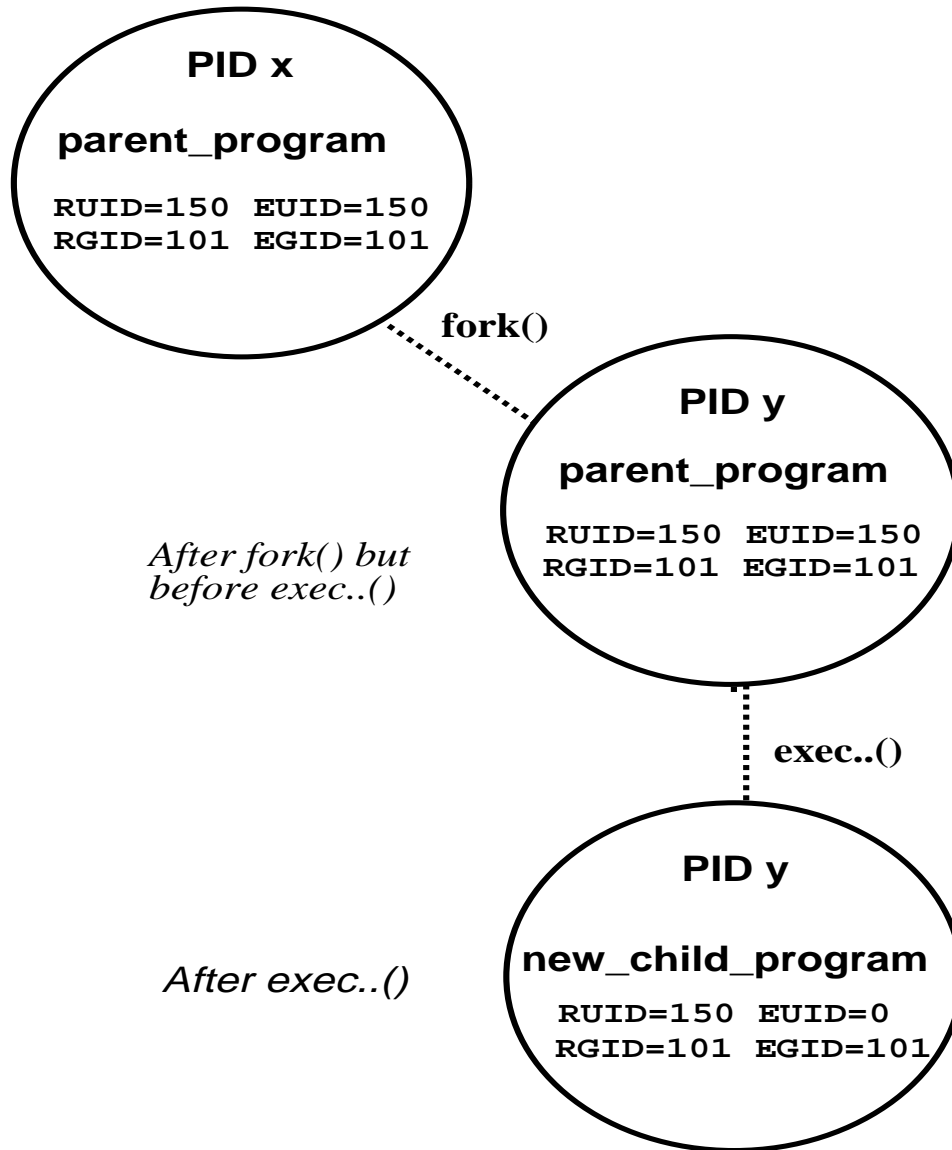
- Access privileges:
  - read (r)
  - write (w)
  - execute (x)
  - set UID (s)
  - set GID (s)

## Objects and Protection (Cont'd)

Output from an `ls -gl` Command

```
- rw- r-- r-- 2 bill staff 6326 Sep 26 09:56 fall94.sched
- rw- --- --- 1 bill staff 18269 Oct 19 19:25 fall94.rooms
- rw- --- --- 2 bill staff 1200 Feb 27 1994 gradeprint
- rws r-x --x 1 bill staff 65541 Aug 19 19:30 rsched
d rwx r-x --x 2 bill staff 24 Jul 19 19:27 subdir
d rwx r-x --x 4 bill staff 24 Jul 19 19:27 subdir1
l rwx --x --x 1 bill staff 9 Oct 19 19:33 tape1 -> /dev/rmt8
| | | | | | | | | | |
| | | | | | | | | | +- FILE NAME
| | | | | | | | | +- LAST MODIFY DATE
| | | | | | | | +- FILE SIZE IN BYTES
| | | | | | +- GROUP AFFILIATION
| | | | +- OWNER NAME
| | | +- NUMBER OF LINKS (other names to this object)
| | +- ACCESS ALLOWED TO NON OWNER NON GROUP PROCESSES
| +- ACCESS ALLOWED TO NON OWNER BUT GROUP MEMBER PROCESSES
| +- ACCESS ALLOWED TO OWNER
+- FILE TYPE AS SHOWN: - ordinary
d directory
l symbolic link
p pipe
s socket (UNIX domain)
c character special (device)
b block special (file system device)
```

# Set UID and Set GID Execution



## The System Call Interface

Most UNIX system calls return an integer, however there are some which return some other type (i.e. pointer type). When a program makes a system call it should check the return for an error. Regardless of the return type, when a UNIX system call errors it returns a value of -1 and sets the global variable **errno** to the error code for this error.

- UNIX system calls have the following format:
  - return = **system\_call** (arg1, ..., argn )
- The successful return type and value are specific to each call
- When a system call fails it will return a value of -1 and will set an integer error code into the linker supplied global variable named **errno**
- Three linker supplied global variables are important to UNIX C programmers, they are:
  - extern int errno, sys\_nerr;
  - extern char \*sys\_errlist[];

## A Useful Function For Error Returns

```
extern int errno, sys_nerr;
extern char *sys_errlist[];

void syserr(char *some_string)
{
    printf("error number %d in %s is: ",
           errno, some_syscall);
    if(errno < sys_nerr){
        /* THEN ERROR NUMBER HAS A CORRESPONDING TEXT */
        printf("%s \n", sys_errlist[errno]);
    }else{
        /* ELSE NO ERROR TEXT AVAILABLE FOR THIS ERROR */
        printf("not documented \n");
    }
}
```

Example Calling Sequence:

```
int sys_call_ret, chan, numbytes;
char buf[];

if( (sys_call_ret = read(chan, buf, numbytes)) == -1 ){
    /* THEN AN ERROR HAS OCCURRED IN READ */
    syserr("read myfile");
    exit(n);
}
```

Possible Output:

error number 9 in read myfile is: Bad file number

# System Services Categories

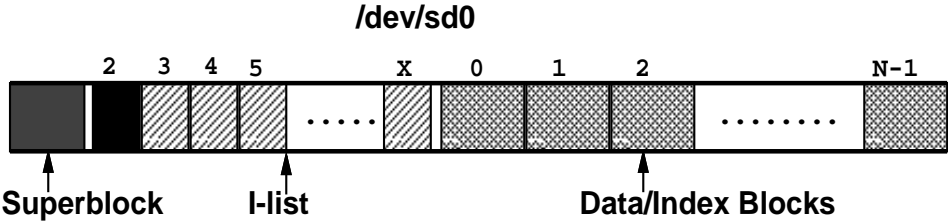
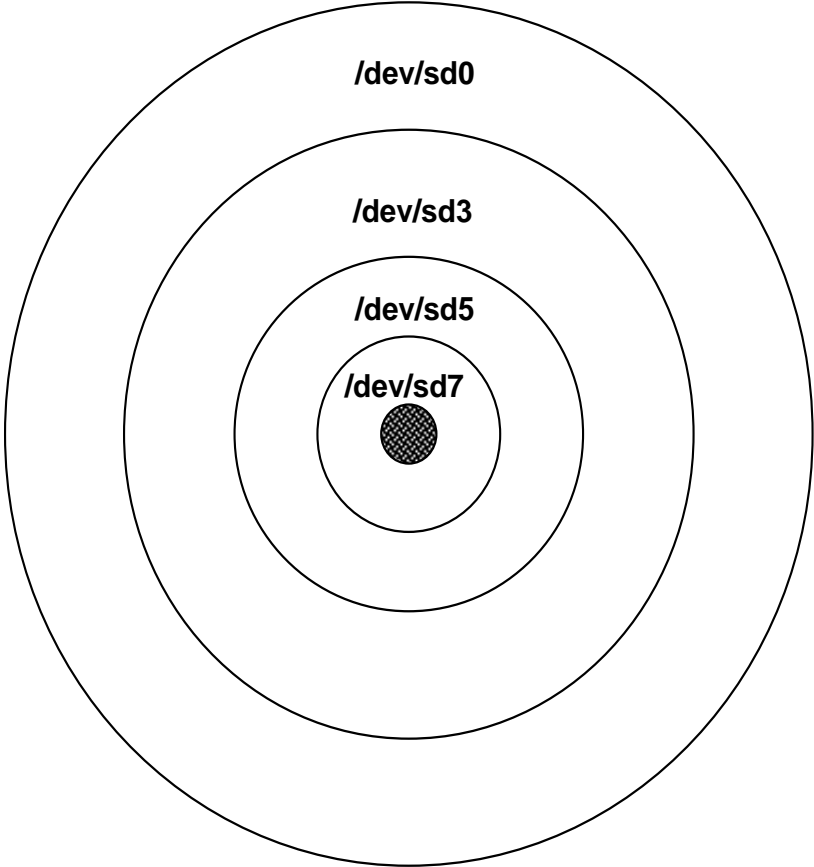
- Basic I-O:
  - creat, unlink, open, close, read, write, lseek
- Advanced and Terminal I-O:
  - mknod, link, symlink, chmod, chdir, chown, fcntl, access, stat, fstat, select, mmap, ioctl
- Processes and Process Management:
  - exec family, fork, exit, getuid, getgid, geteuid, getegid, getpid, getpgrp, nice, setuid, setgid, setpgrp, getpriority, setpriority, signal, sigvec, sigaction, pause, sigprocmask, sigsuspend, sigsend, kill, wait, setjmp, longjmp
- Basic Pipe and Advanced IPC Mechanisms:
  - pipe, dup, dup2, FIFOs with mknod, msgOPS, shmOPS, semOPS
- Interhost IPC with Sockets:
  - socket, socketpair, bind, connect, listen, accept, gethostbyname, getsockname, recv, send
- Miscellaneous System Services:
  - umask, brk, sbrk, ptrace, profile, mount, umount

## The UNIX Filesystem Abstraction

While the UNIX filesystem is subject to implementation differences among various vendor platforms, the abstract filesystem is essentially the same for all platforms. The abstract filesystem consists of a region of storage which supports block data transfer (e.g. a disk) and has a type **b** driver interface. A filesystem is formatted onto a collection of logically contiguous blocks using the **mkfs** maintenance command. The resulting format generally organizes the blocks into three regions:

- A *superblock* to maintain statistics for and pointers to the other two regions
- An *I-list* which contains a fixed number of **inodes** used as file control blocks (FCBs), and allocated at the rate of exactly one per file object (setting a limit on the number of file objects in a filesystem)
- A collection of *data/index* blocks, used to carry the data and possible indirect index pointers of a file

# Filesystem Layout



## UNIX inode Organization

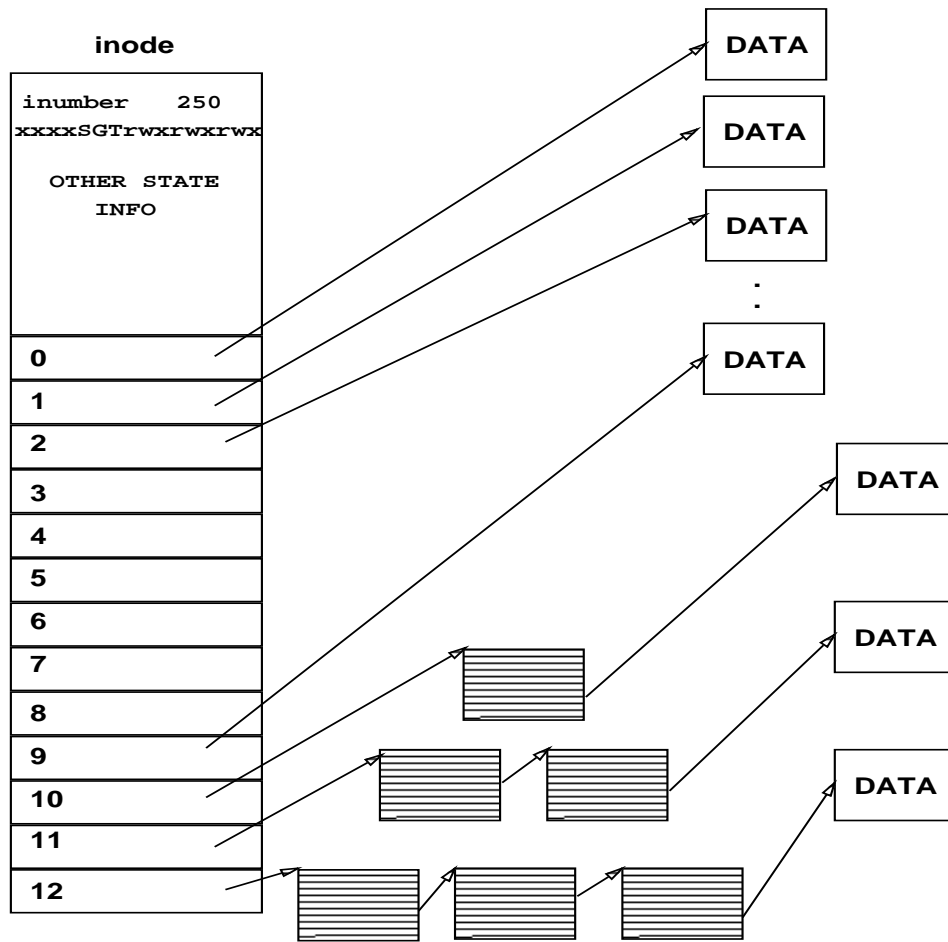
An inode is used as a file control block and each object in the file system requires one inode. An inode is logically divided into two parts, a collection of file state information and a set of 13 pointers used to point to the file's data/index blocks, it contains:

- The I-number of this node
- The type and mode of the file, where mode is the set of read-write-execute privileges and types are as follows:
  - ordinary (-)
  - directory (d)
  - character (c)
  - block (b)
  - pipe or FIFO (p)
  - symbolic link (l)
  - socket (s)

## **UNIX inode Organization (Cont'd)**

- The number of links to the file
- The owner's user-id number (UID)
- The group-id (GID) of the file
- The number of bytes in the file
- The device codes of the inode's device
- The device codes of device inode points to
- The date and time last accessed
- The date and time last modified
- The date and time i-node last modified
- An array of 13 disk block addresses

# I-node Layout



Assume data block is 1BK ( $2^{10}$ ) and pointers are 4 bytes

An INDEX block could then hold 256 pointers (1K/4)

Direct	-	10 ptrs * 1K	=	10 KB
1st Level Indirect	-	256 ptrs * 1K +	=	266 KB
2nd Level Indirect	-	256 ** 2 ptrs * 1K +	=	64+ MB
3rd Level Indirect	-	256 ** 3 ptrs * 1K +	=	16+ GB