0.1 UNIX System initialization and Bootstrapping

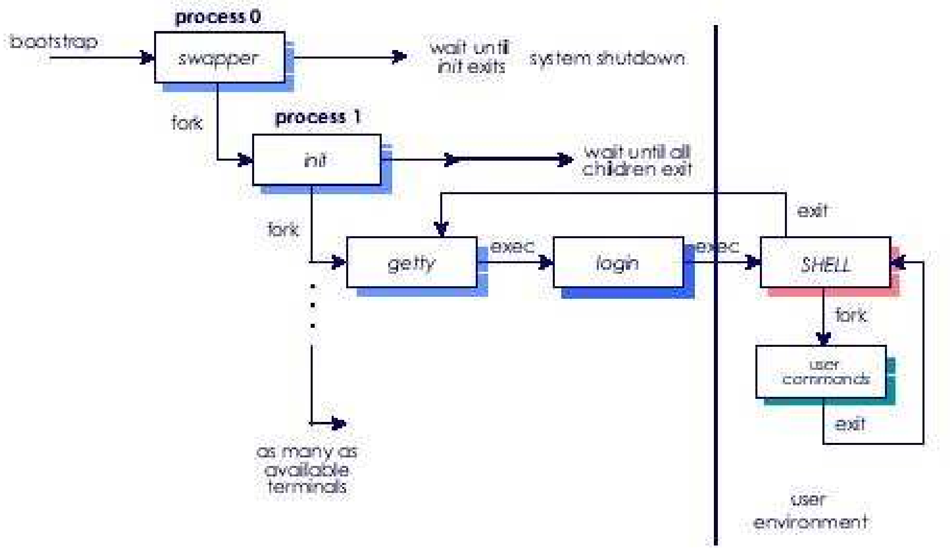
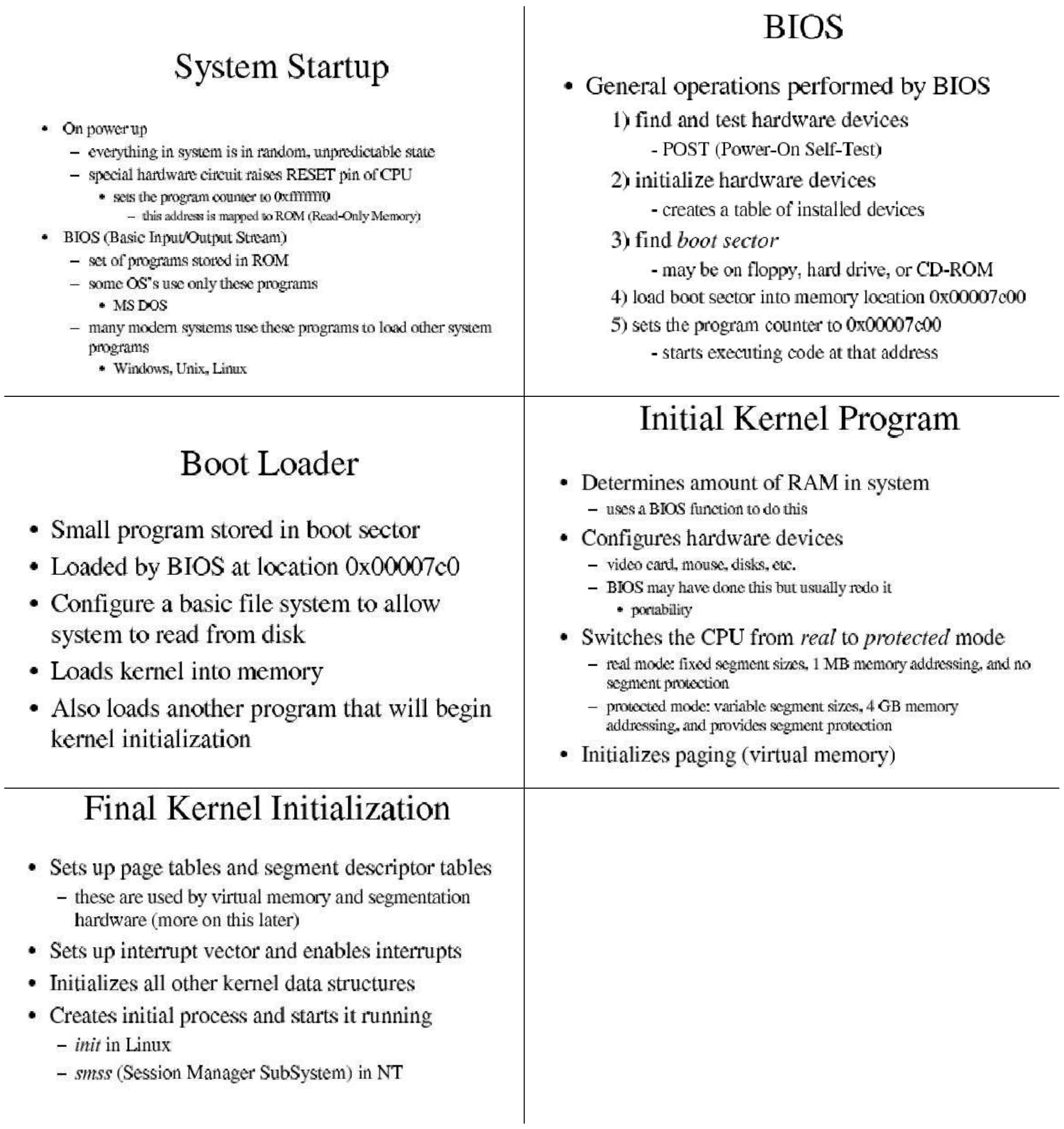


Figure 1: UNIX System initialization

- Once the kernel boots, we have a running Linux system. It isn't very usable, since the kernel doesn't allow direct interactions with "user space".
- So, the system runs one program: **init**. This program is responsible for everything else and is regarded as the father of all processes.
- The kernel then retires to its rightful position as system manager handling "kernel space" (see Fig. 1).
- The process of initializing the computer and loading the operating system is known as *bootstrapping* (see Fig. 2).
- Some portions of the operating system remain in main memory to provide services for critical operations, such as dispatching, interrupt handling, or managing (critical) resources.
- These portions of the OS are collectively called the *kernel*.

Kernel = OS - transient components
remains *comes and goes*



System Startup

- On power up
  - everything in system is in random, unpredictable state
  - special hardware circuit raises RESET pin of CPU
    - sets the program counter to 0xfffffff0
      - this address is mapped to ROM (Read-Only Memory)
- BIOS (Basic Input/Output Stream)
  - set of programs stored in ROM
  - some OS"s use only these programs
    - MS DOS
  - many modern systems use these programs to load other system programs
    - Windows, Unix, Linux

BIOS

- General operations performed by BIOS

  1) find and test hardware devices
     - POST (Power-On Self-Test)
  2) initialize hardware devices
     - creates a table of installed devices
  3) find *boot sector*
     - may be on floppy, hard drive, or CD-ROM
  4) load boot sector into memory location 0x00007c00
  5) sets the program counter to 0x00007c00
     - starts executing code at that address

Boot Loader

- Small program stored in boot sector
- Loaded by BIOS at location 0x00007c0
- Configure a basic file system to allow system to read from disk
- Loads kernel into memory
- Also loads another program that will begin kernel initialization

Initial Kernel Program

- Determines amount of RAM in system
  - uses a BIOS function to do this
- Configures hardware devices
  - video card, mouse, disks, etc.
  - BIOS may have done this but usually redo it
    - portability
- Switches the CPU from *real* to *protected* mode
  - real mode: fixed segment sizes, 1 MB memory addressing, and no segment protection
  - protected mode: variable segment sizes, 4 GB memory addressing, and provides segment protection
- Initializes paging (virtual memory)

Final Kernel Initialization

- Sets up page tables and segment descriptor tables
  - these are used by virtual memory and segmentation hardware (more on this later)
- Sets up interrupt vector and enables interrupts
- Initializes all other kernel data structures
- Creates initial process and starts it running
  - *init* in Linux
  - *smss* (Session Manager SubSystem) in NT

Figure 2: Booting the computer.

0.2 The Operating System Zoo

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Handheld operating systems
- Embedded operating systems
- Sensor node operating systems
- Real-time operating systems
- Smart card operating systems

0.3 Operating System Concepts

Most operating systems provide certain basic concepts and abstractions such as processes, address spaces, and files that are central to understanding them.
• <u>Processes</u>

- A key concept in all operating systems is the **process**. A process is basically a program in execution.
- Associated with each process is its **address space**, a list of memory locations from 0 to some maximum, which the process can read and write. The address space contains the executable program, the program's data and its stack.
- Also associated with each process is a set of resources, commonly including registers (program counter, stack pointer, ..), a list of open files, outstanding alarms, lists of related processes, and all the other information needed to run the program.
- A process is fundamentally a container that holds all the information needed to run a program.
- In many operating systems, all the information about each process, other than the contents of its own address space, is stored in a table called the **process table**, which is an array (or linked list) of structures, one for each process currently in existence.

- if a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 3.
- Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities. This communication is called **interprocess communication**.

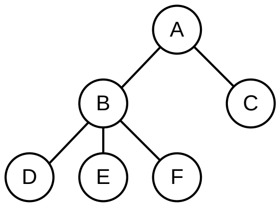


Figure 3: A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

- Address spaces
  - In a very simple operating system, only one program at a time is in the memory. To run second program, the first one has to be removed and the second one placed in memory. Single Tasking System.
    - Simple to implement, Only one process attempting use resources.
    - Few security risks.
    - Poor utilization of the CPU and other resources.
    - i.e., MS-DOS
  - More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with OS), some kind of protection mechanism is needed. Multi Tasking System.
    - Very complex.
    - Serious security issues, how to protect one program from another sharing the same memory.
    - Much higher utilization of system resources.
    - i.e., Unix, Windows NT

- Each process has some set of addresses it can use, typically running from 0 up to some maximum.
- On many computers addresses are 32 or 64 bits, giving an address space of 2^{32} or 2^{64} bytes, respectively.
- *Virtual Memory:* The OS keeps part of the address space in main memory and part on disk and shuttles pieces back and forth betweenthem as needed.

- <u>Files</u>
  - A major function of the OS is to hide the peculiarities of the disks and other I/O devices and present the user/programmer with a nice, clean abstract model of device-independent files.
  - To provide a place to keep files, most OS have the concept of a **directory** as a way of grouping files together (see Fig.).

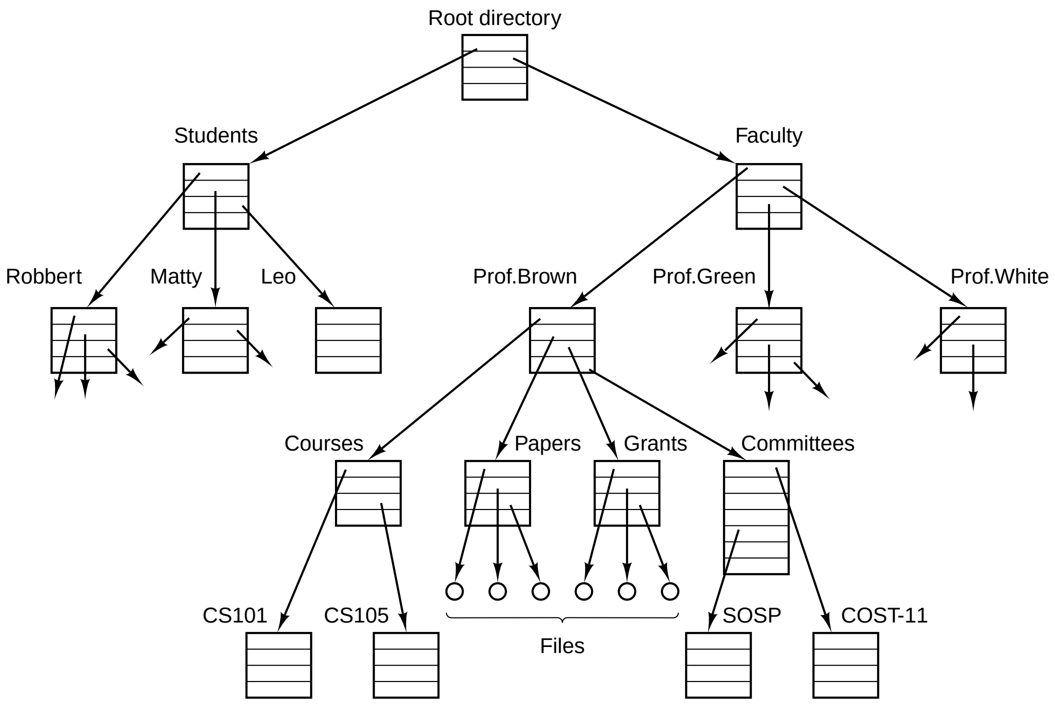


Figure 4: A file system for a university department.

  - The process and file hierarchies both are organized as trees, but the similarity stops there;

  - Process hierarchies are not very deep,
  - Process hierarchies are typically short-lived,
  - Ownership and protection also differs for processes and files.

- Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**.
  - Absolute path
  - Relative path

- Before a file can be read or written, it must be opened, at which time the permissions are checked. Ä°f the access is permitted, the system returns a small integer called a **file descriptor** to use a subsequent operations.

- *Mounting:* Consider the situation of Fig. 5a. Before the mount call (system call), the **root file system**, on the hard disk, and a second file system, on a CD-ROM, are separate and unrelated.

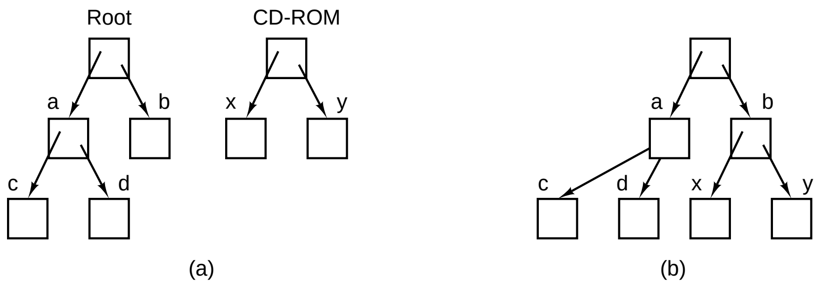


Figure 5: (a) Before mounting, the files on the CD-ROM are not accessible. (b) After mounting, they are part of the file hierarchy.

- Another important concept in UNIX is the **special file**. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files.
  - **Block special files** are used to model devices that consist of a collection of randomly addressable blocks, such as disks.

- **Character special files** are used to model printers, modems, and other devices that accept or output character stream.
- */dev* is the directory. */dev/lp* might be the printer (once called the line printer).

- Input/Output
  - Many kinds of input and output devices exist, including keyboards, monitors, printers, and so on. It is up to the OS to manage these devices.
  - Consequently, every OS has an I/O subsystem for managing its I/O devices.
    - Some of the I/O software is device independent, that is, applies to many or all I/O devices equally well.
    - Other parts of it, such as device drivers, are specific to particular I/O devices.
- Protection
  - It is up to the OS to manage the system security so that files are only accessible to authorized users.
  - Files in UNIX are protected by assigning each one a 9-bit binary protection code.
    - Three bit fields, one for owner, one for other members of the owner's group, and one for everone else.
    - Each field has a bit for **read** access, a bit for **write** access, and a bit for **execute** access.
    - $(d)rwxr-x--x$
  - In addition to file protection, there are many other security issues.
    - OS must protect itself from users; reserved memory only accessible by OS. The OS is responsible for allocating access to memory space and CPU time and peripherals etc., and it will control dedicated hardware facilities:
      * The memory controller, control to detect and prevent unauthorized access.
      * A timer will also be under operating system control to manage CPU time allocation to programs competing for resources.

- OS may protect users from another user. A fundamental requirement of multiple users of a shared computer system is that they do not interfere with each other. This gives rise to the need for separation of the programs in terms of their resource use and access:
    - If one program attempts to access main memory allocated to some other program, the access should be denied and an exception raised.
    - If one program attempts to gain a larger proportion of the shared CPU time, this should be prevented.

- One approach to implementing resource allocation is to have at least two modes of CPU operation. Modes of operation:
  - Supervisor (protected, kernel) mode: *all* (basic and privileged) instructions available.
    - All hardware and memory available. Mode the OS runs in. Never let the user run in supervisory mode.
  - User mode: a *subset* of instructions.
    - Limited set of hardware and memory available. Mode all user programs run in.
    - *I/O protection*, all I/O operations are privileged; so user programs can only access I/O by sending a request to the (controlling) operating system.
    - *Memory protection*, base/limit registers (in early systems), memory management unit, (MMU, in modern systems); so user programs can only access the memory that the operating system has allocated.
    - *CPU control*, timer (alarm clock), context switch; so user programs can only read the time of day, and can only have as much CPU time as the operating system allocates.

- <u>The shell</u> (see Fig. 1)
  - UNIX command interpreter, called the **shell**. Although it is not part of the OS, it makes heavy use of many operating system features and serves as a good example of hot the system calls can be used.
  - It is also the primary interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface.

- When any user logs in, a shell is started up. The shell has the terminal as standart input and standard output. It starts out by typing the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command.
- When a command is typed, the shell **fork**s off a new process. This child process must execute the user command.
- A highly simplified shell illustrating the use of fork, waitpid, and execve is shown in Fig. 6.

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
     type_prompt( );                            /* display prompt on the screen */
     read_command(command, parameters);         /* read input from terminal */

     if (fork( ) != 0) {                        /* fork off child process */
         /* Parent code. */
         waitpid(−1, &status, 0);               /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);        /* execute command */
     }
}
```

Figure 6: A stripped-down shell.

0.4 System Calls

- The *system calls* available in the abstraction interface vary from OS to OS (although the underlying concepts tend to be similar).
- Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading a data from a file, it has to execute a trap instruction to tarnsfer control to the OS.
- OS carries out the system call and returns control to the instruction following the system call.
- $count = read(fd, buffer, nbytes)$; read - system call.

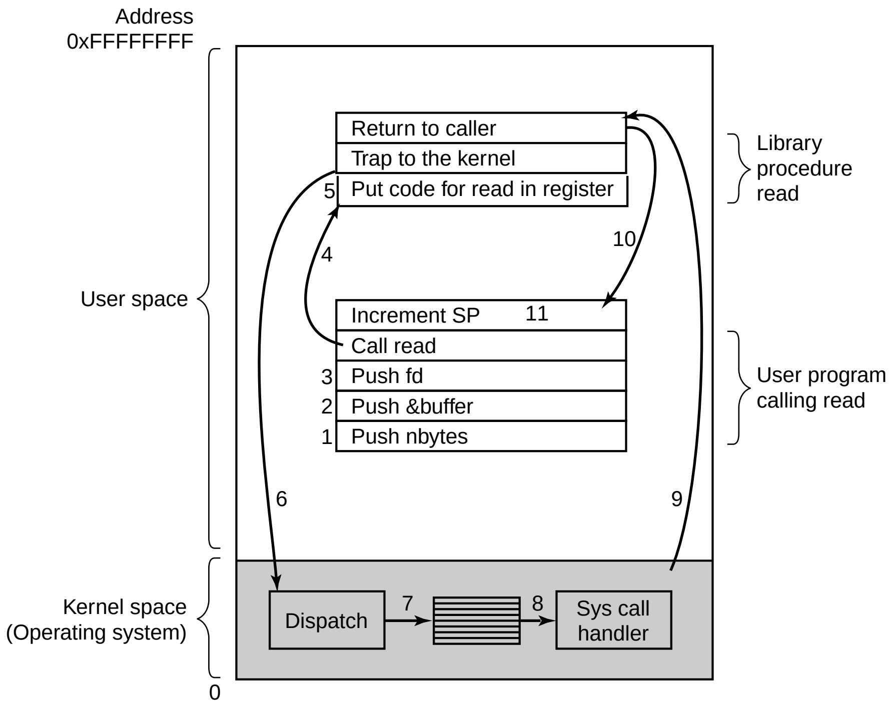


Figure 7: The 11 steps in making the system call - read(fd, buffer, nbytes).

- Step 4: Actual call to the library procedure.
- Step 5: This procedure puts the system call number in a place where the OS expects it, such as a register.
- Step 6: Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel.The 11 steps in making the system call read(fd, buffer, nbytes).
- Step 7: The kernel code that starts following the TRAP examines the system call number and then dispatches to the correct system call handler.
- Step 8: The system call handler runs.
- Step 9: Once the system call handler has completed its work, control may be returned to the user-space library procedure.
- Step 10: This procedure then returns to the user program in the usual way procedure call returns.

- Step 11: To finish the job, the user program has to clean up the stack, as it does after any procedure call. Assuming the stack grows downward, the compiled code increments the stack pointer exactly enough to remove the parameters pushed before the call to *read*.

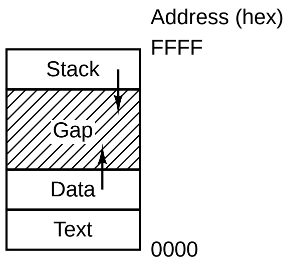


Figure 8: Processes have three segments: text, data, and stack.

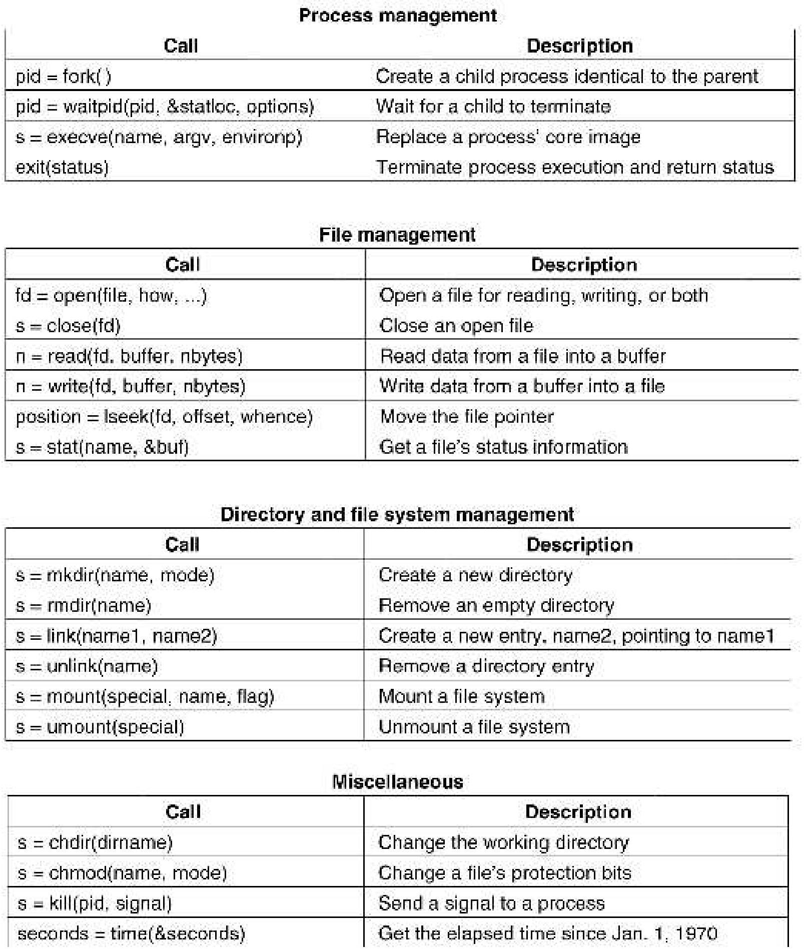
- Some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls are given in Fig. 9.
- These system calls can be roughly grouped as the following:
  - Process control; fork, exec, wait, abort.
  - File manipulation; chmod, link, stat, creat.
  - Device manipulation; open, close, ioctl, select.
  - Information maintenance; time, acct, gettimeofday.
  - Communications; socket, accept, send, recv.

0.5 Operating Systems Structure

An operating system generally consists of the following components (see Fig. 10):
• Monolithic Systems, Layered Systems, Microkernels, Client-server Systems, Virtual Machines, Exokernels.

0.5.1 Monolithic Systems

- The most common organization. The structure is that there is no structure.



Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 9: Some of the major POSIX system calls.

- The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to.
- To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file using the system linker.
- In terms of information hiding, there is essentially none – every procedure is visible to every other procedure.
- Even in monolithic systems, however, it is possible to have at least a

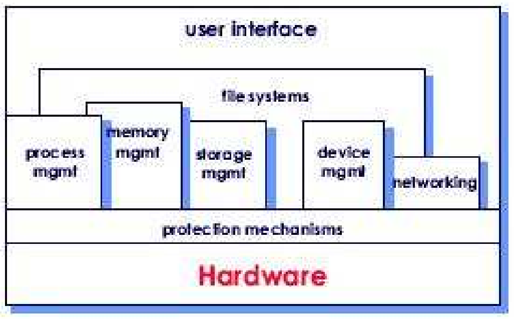


Figure 10: OS Architecture

little structure, remember the system calls at Fig. 7).

- A main program that invokes the requested service procedure.
- A set of service procedures that carry out the system calls.
- A set of utility procedures (such as fetching data from user programs) that help the service procedures.

- This division of the procedures into three layers is shown in Fig. 11.

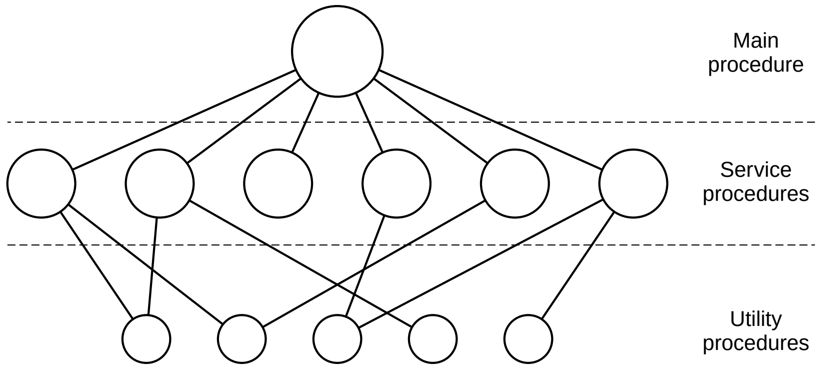


Figure 11: A simple structuring model for a monolithic system.

0.5.2 Layered Systems

- A generalization of this division of the procedures approach is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it.

- The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students.
- The system had 6 layers, as shown in Fig. 12.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 12: Structure of the THE operating system.

  - Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Layer 0 provided the basic multiprogramming of the CPU.
  - Layer 1 did the memory management. It allocated space for processes in main memory. Layer 1 software took care of making sure pages were brought into memory whenever they were needed.
  - Layer 2 handled communication between each process and the operator console.
  - Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities.
  - Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management.
  - The system operator process was located in layer 5.
  - A further generalization of the layering concept was present in the MULTICS (Multiplexed Information and Computing Service, an extremely influential early time-sharing operating system, 1964) system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones.

0.5.3 Microkernels

- Bug density depends on module size, module age, and more, but a ballpark figure for serious industrial systems is ten bugs per thousand lines of code. This means that a monolithic OS of five million lines of code is likely to contain something like 50000 kernel bugs.
- The basic idea behind the microkernel design is to achieve high reliability by splitting the OS up into small, well-defined modules.
- If the hardware provides multiple privilege levels, then the microkernel is the only software executing at the most privileged level (generally referred to as supervisor or kernel mode).
- Actual operating system services, such as device drivers, protocol stacks, file systems and user interface code are contained in user space.
- The MINIX 3 microkernel is only about 3200 lines of C and 800 lines of assembler for very low-level functions such as catching interrupts and switching processes.
- The C code manages and schedules processes, handles interprocess communication (by passing message between processes), and offers a set of about 35 kernel calls. The process structure of MINIX 3 is shown in Fig. 13

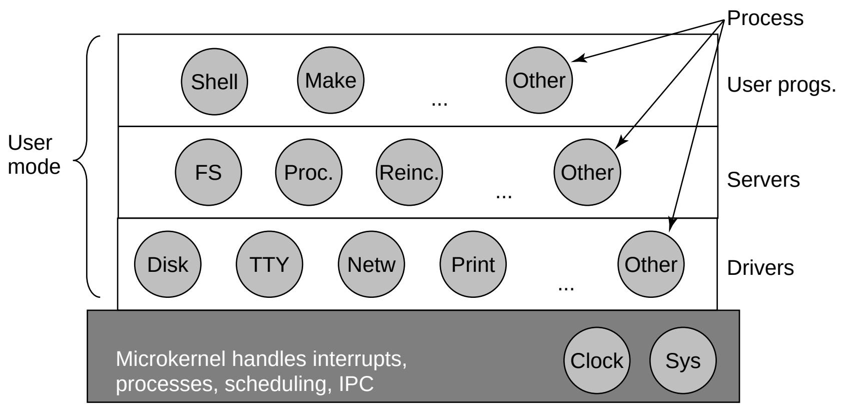


Figure 13: Structure of the MINIX 3 system.

0.5.4 Client-Server Model

- A slight variation of the microkernel idea is to distinguish two classes of processes,

    1. the **servers**, each of which provides some service
    2. the **clients**, which use these services.

- An obvious generalization of the client-server model is its adaptability to use in distributed systems (see Fig. 14).

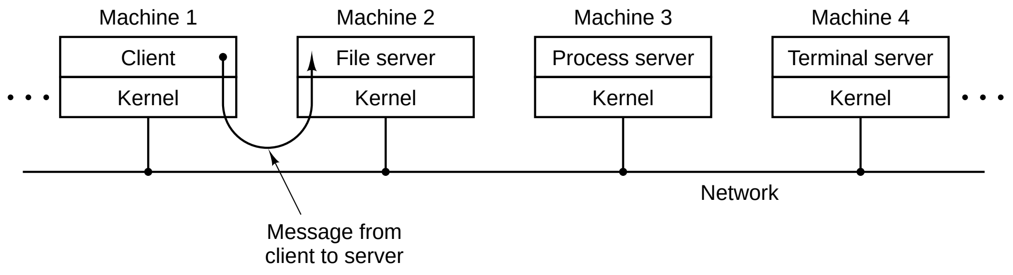


Figure 14: The client-server model over a network.

- If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

0.5.5 Virtual Machines

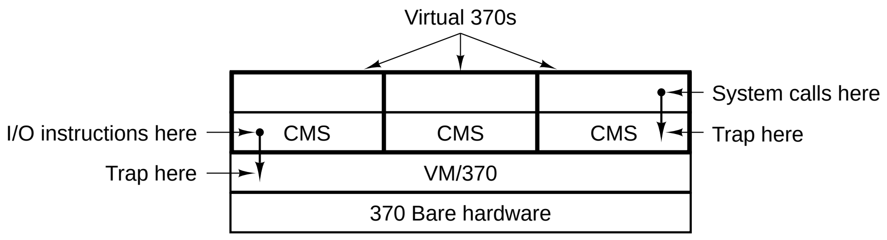


Figure 15: The structure of VM/370 with CMS (Conversational Monitor System).

- The heart of the system, known as the virtual machine monitor, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 151-26.
- However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are exact copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.
- Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware.
- Virtual Machines Rediscovered. Another use of virtualization is for end users who want to be able to run two or more OS at the same time, say Windows and Linux. This situation is illustrated in Fig. 16a, where the term "virtual machine monitor" has been renamed type 1 **hypervisor** in recent years.

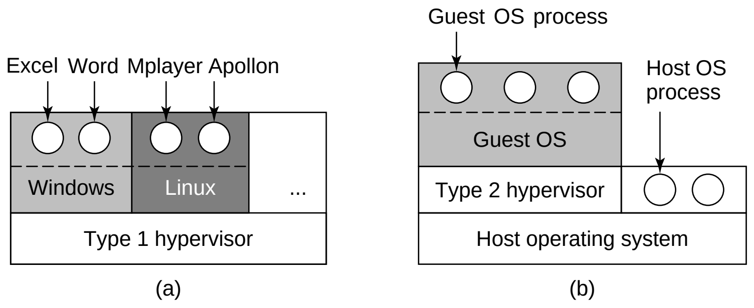


Figure 16: (a) A type 1 hypervisor. (b) A type 2 hypervisor.

- Another area where virtual machines are used, but in a somewhat different way, is for running Java programs.
- When Sun Microsystems invented the Java programming language, it also invented a virtual machine (i.e., a computer architecture) called the JVM (Java Virtual Machine).
- The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there.

0.5.6 Exokernels

- Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

- At the bottom layer, running in kernel mode, is a program called the exokernel. Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources.

- Each user-level virtual machine can run its own operating system, as on VM/370 and the Pentium virtual 8086s, except that each one is restricted to using only the resources it has asked for and been allocated.

- The advantage of the exokernel scheme is that it saves a layer of mapping.
  - In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources).
  - With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource.

0.6 Problems in building OS

- *Large Systems:* 100k's to millions of lines of code involving 100 to 1000 man-years of work

- *Complex:* Performance is important while there is conflicting needs of different users, Cannot remove all bugs from such complex and large software

- Behavior is hard to predict; tuning is done by guessing