GENERAL OVERVIEW OF THE SYSTEM:

The UNIX system has become quite popular since its inception in 1969, running on machines of varying processing power from microprocessors to mainframes and providing a common execution environment across them. The system is divided into two parts. The first part consists of programs and services that have made the UNIX system environment so popular; it is the part readily apparent to users, including such programs as the shell, mail, text processing packages, and source code control systems. The second part consists of the operating system that supports these programs and services.

HISTORY

In 1965, Bell Telephone Laboratories joined an effort with the General Electric Company and Project MAC of the Massachusetts Institute of Technology to develop a new operating system called Multics. The goals of the Multics system were to provide simultaneous computer access to a large community of users, to supply ample computation power and data storage, and to allow users to share their data easily, if desired.

Many people who later took part in the early development of the UNIX system participated in the Multics work at Bell Laboratories. Although a primitive version of the Multics system was running on a GE 645 computer by 1969, it did not provide the general service computing for which it was intended, nor was it clear when its development goals would be met.

JKDirectory!

Consequently, Bell Laboratories ended its participation in the project. With the end of their work on the Multics project, members of the Computing Science Research Center at Bell Laboratories were left without a "convenient interactive computing service". In an attempt to improve their programming environment, Ken Thompson, Dennis Ritchie, and others sketched a paper design of a file system that later evolved into an early version of the UNIX file system.

Thompson wrote programs that simulated the behavior of the proposed file system and of programs in a demand-paging environment, and he even encoded a simple kernel for the GE 645 computer. At the same time, he wrote a game program, "Space Travel," in Fortran for a GECOS system (the Honeywell 635), but the program was unsatisfactory because it was difficult to control the "space ship" and the program was expensive to run. Thompson later found a little-used PDP-7 computer that provided good graphic display and cheap executing power.

Programming "Space Travel" for the PDP-7 enabled Thompson to learn about the machine, but its environment for program development required cross-assembly of the program on the GECOS machine and carrying paper tape for input to the PDP-7.

To create a better development environment, Thompson and Ritchie implemented their system design on the PDP-7, including an early version of the UNIX file system, the process subsystem, and a small set of utility programs.

Eventually, the new system no longer needed the GECOS system as a development environment but could support itself. **The new system was given the name UNIX** a pun on the name Multics coined by another member of the Computing Science Research Center, Brian Kernighan.

Several reasons have been suggested for the popularity and success of the UNIX system:

- ✓ The system is written in a high-level language, making it easy to read, understand, change, and move to other machines.
- ✓ It has a simple user interface that has the power to provide the services that users want.
- ✓ It provides primitives that permit complex programs to be built from simpler programs.
- ✓ It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- ✓ It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- ✓ It provides a simple, consistent interface to peripheral devices.
- ✓ It is a multi-user, multiprocess system; each user can execute several processes simultaneously.
- ✓ It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations.

JKDirectory!

Although the operating system and many of the command programs are written in C, UNIX systems support other languages, including FORTRAN, Basic, Pascal, Ada, COBOL, Lisp, and Prolog. The UNIX system can support any language that has a compiler or interpreter and a system interface that maps user requests for operating system services to the standard set of requests used on UNIX systems.

SYSTEM STRUCTURE:

Figure 1.1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services. The operating system interacts directly with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies.

Viewing the system as a set of layers, the operating system is commonly called the system kernel, or just the kernel, emphasizing its isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware if the programs do not make assumptions about the underlying hardware.

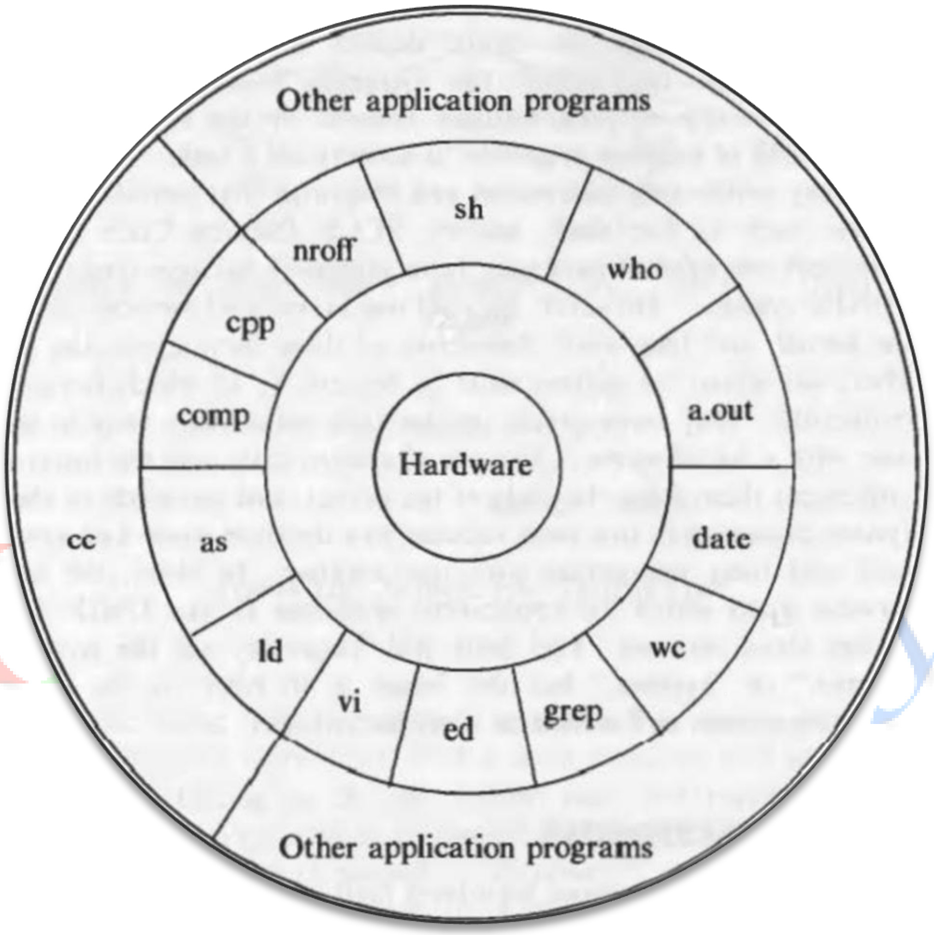


Figure 1.1 Architecture of UNIX System

For instance, programs that assume the size of a machine word are more difficult to move to other machines than programs that do not make this assumption. Programs such as the shell and editors (ed and vi) shown in the outer layers interact with the kernel by invoking a well defined set of system calls.

The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in the figure are in standard system configurations and are known as commands, but private user programs may also exist in this layer as indicated by the program whose name is a.out, the standard name for executable files produced by the C compiler.

Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the standard C compiler, cc, is in the outermost layer of the figure: it invokes a C preprocessor, two-pass compiler, assembler, and loader Oink-editor), all separate lower-level programs.

USER PERSPECTIVE:

This will briefly review high-level features of the UNIX system such as the file system, the processing environment, and building block primitives (for example, pipes).

The File System:

The UNIX file system is characterized by

- A hierarchical structure,
- Consistent treatment of file data,
- The ability to create and delete files,
- Dynamic growth of files,
- The protection of file data,
- The treatment of peripheral devices (such as terminals and tape units) as files.

JKDirectory!

The file system is organized as a tree with a single root node called root (written as “/"); every non-leaf node of the file system structure is a directory of files, and files at the leaf nodes of the tree are either directories, regular files, or special device files.

The name of a file is given by a path name that describes how to locate the file in the file system hierarchy.

A ***path name*** is a sequence of component names separated by slash characters; a component is a sequence of characters that designates a file name that is uniquely contained in the previous (directory) component.

A *full path name* starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name. Thus, the path names “/etc/passwd”, “/bin/who”, and “/usr/src/cmd/who.c” designate files in the tree shown in Figure 1.2, but “/bin/passwd” and “/usr/src/date.c” do not.

A path name does not have to :start from root but can be designated relative to the current directory of an executing process, by omitting the initial slash in the path name.

Thus, starting from directory "/dev", the path name "tty01" designates the file whose full path name is "/dev/ttyo1".

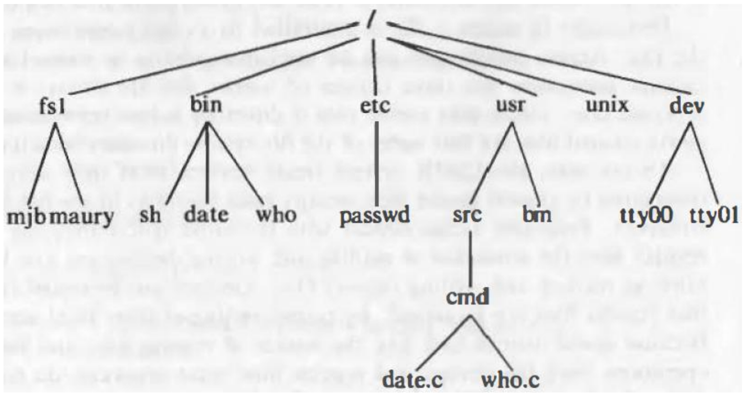


Figure 1.2 Sample File System Tree

Programs in the UNIX system have no knowledge of the internal format in which the kernel stores file data, treating the data as an unformatted stream of bytes. Programs may interpret the byte stream as they wish, but the interpretation bas no bearing on how the operating system stores the data. Thus, the syntax of accessing the data in a file is defined by the system and is identical for all programs, but the semantics of the data are imposed by the program.

JKDirectory!

Directories are like regular files in this respect; the system treats the data in a directory as a byte stream, but the data contains the names of the files in the directory in a predictable format so that the operating system and programs such as Is (list the names and attributes of files) can discover the files in a directory.

Permission to access a file is controlled by access permissions associated with the file. Access permissions can be set independently to control read, write, and execute permission for three classes of users: the file owner, a file group, and everyone else. Users may create files if directory access permissions allow it. The newly created files are leaf nodes of the file system directory structure.

To the user, the UNIX system treats devices as if they were files. Devices, designated by special device files, occupy node positions in the file system directory structure.

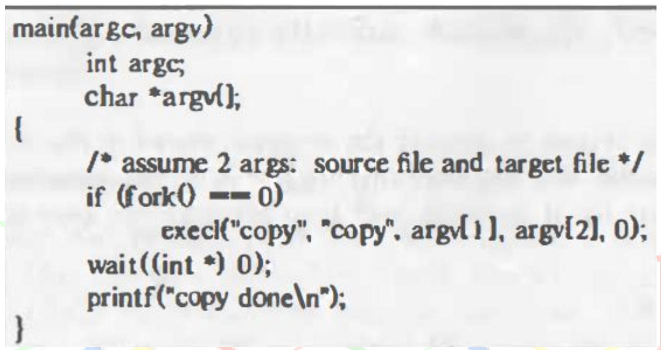
Programs access devices with the same syntax they use when accessing regular files; the semantics of reading and writing devices are to a large degree the same as reading and writing regular files.

Devices are protected in the same way that regular files are protected: by proper setting of their (file) access permissions.

Processing Environment:

A program is an executable file, and a process is an instance of the program in execution. Many processes can execute simultaneously on UNIX systems (this feature is sometimes called multiprogramming or multitasking) with no logical limit to their number, and many instances of a program (such as copy) can exist simultaneously in the system. Various system calls allow processes to create new processes, terminate processes, synchronize stages of process execution, and control reaction to various events. Subject to their use of system calls, processes execute independently of each other.

For example, a process executing the program in Figure 1.3 executes the *fork* system call to create a new process. The new process, called the child process, gets a 0 return value from *fork* and invokes *execl* to execute the program copy.



```
main(argc, argv)
        int argc;
        char *argv[];
{
        /* assume 2 args: source file and target file */
        if (fork() == 0)
                execl("copy", "copy", argv[1], argv[2], 0);
        wait((int *) 0);
        printf("copy done\n");
}
```

JKDirectory!

Figure 1.3 Program that Creates a New Process to Copy Files

Building Block Primitives:

The philosophy of the UNIX system is to provide operating system primitives that enable users to write small, modular programs that can be used as building blocks to build more complex programs.

One such primitive visible to shell users is the capability to redirect I/O. Processes conventionally have access to three files: they read from their standard input file, write to their standard output file, and write error messages to their standard error file.

Processes executing at a terminal typically use the terminal for these three files, but each may be "redirected" independently.

For instance, the command line ***ls*** lists all files in the current directory on the standard output. But the command line **ls > output** redirects the standard output to the file called "output" in the current directory.

The second building block primitive is the pipe, a mechanism that allows a stream of data to be passed between reader and writer processes.

Processes can redirect their standard output to a pipe to be read by other processes that have redirected their standard input to come from the pipe. The data that the first processes write into the pipe is the input for the second processes.

The second processes could also redirect their output, and so on, depending on programming need. Again, the processes need not know what type of file their standard output is; they work regardless of whether their standard output is a regular file, a pipe, or a device.

When using the smaller programs as building blocks for a larger, more complex program, the programmer uses the pipe primitive and redirection of I/O to integrate the piece parts.

Indeed, the system tacitly encourages such programming style so that new programs can work with existing programs. For example, the program grep searches a set of files (parameters to grep) for a given pattern:

grep main a.c b.c c.c

searches the three files a.c, b.c, and c.c for lines containing the string "main" and prints the lines that it finds onto standard output. Sample output may be:

JKDirectory!

```
a.c: main(argc, argv)
c.c: /* here is the main loop in the program */
c.c: main()
```

The program we with the option -1 counts the number of lines in the standard input file. The command line ***grep main a.c b.c c.c | wc -l*** counts the number of lines in the files that contain the string "main"; the output from grep is "piped" directly into the wc command. For the previous sample output from grep, the output from the piped command is ***3.*** The use of pipes frequently makes it unnecessary to create temporary files.

OPERATING SYSTEM SERVICES:

Figure 1.1 depicts the kernel layer immediately below the layer of user application programs. The kernel performs various primitive operations on behalf of user processes to support the user interface.

Among the services provided by the kernel are:

- ✓ Controlling the execution of processes by allowing their creation, termination or suspension, and communication.

- ✓ Scheduling processes fairly for execution on the CPU. Processes share the CPU in a time-shared manner: the CPU executes a process, the kernel suspends it when its time quantum elapses, and the kernel schedules another process to execute. The kernel later reschedules the suspended process.

- ✓ Allocating main memory for an executing process. The kernel allows processes to share portions of their address space under certain conditions, but protects the private address space of a process from outside tampering.

  - If the system runs low on free memory, the kernel frees memory by writing a process temporarily to secondary memory, called a swap device.

  - If the kernel writes entire processes to a swap device, the implementation of the UNIX system is called a swapping system; if it writes pages of memory to a swap device, it is called a paging system.

- ✓ Allocating secondary memory for efficient storage and retrieval of user data. This service constitutes the file system. The kernel allocates secondary storage for user files, reclaims unused storage, structures the file system in a well understood manner, and protects user files from illegal access.

JKDirectory!

- ✓ Allowing processes controlled access to peripheral devices such as terminals, tape drives, disk drives. And network devices. The kernel provides its services transparently. For example, it recognizes that a given file is a regular file or a device, but hides the distinction from user processes.

Similarly, it formats data in a file for internal storage, but hides the internal format from user processes, returning an unformatted byte stream.

ASSUMPTIONS ABOUT HARDWARE:

The execution of user processes on UNIX systems is divided into two levels: ***user and kernel.***

When a process executes a system call, the execution mode of the process changes from user mode to kernel mode: the operating system executes and attempts to service the user request, returning an error code if it fails.

Even if the user makes no explicit requests for operating system services, the operating system still does bookkeeping operations that relate to the user process, handling interrupts, scheduling processes, managing memory, and so on.

Many machine architectures (and their operating systems) support more levels than the two outlined here, but the two modes, user and kernel, are sufficient for UNIX systems.

The differences between the two modes are

Processes in user mode can access their own instructions and data but not kernel instructions and data.

Processes in kernel mode, however, can access kernel and user addresses. For example, the virtual address space of a process may be divided between addresses that are accessible only in kernel mode and addresses that are accessible in either mode.

Some machine instructions are privileged and result in an error when executed in user mode. For example, a machine may contain an instruction that manipulates the processor status register; processes executing in user mode should not have this capability.

Put simply, the hardware views the world in terms of kernel mode and user mode and does not distinguish among the many users executing programs in those modes. The operating system keeps internal records to distinguish the many processes executing on the system.

JKDirectory!

Figure 1.4 shows the distinction: the kernel distinguishes between processes A, B, C, and D on the horizontal axis and the hardware distinguishes the mode of execution on the vertical axis.

	A	B	C	D
KERNEL MODE	K			K
USER MODE		U	U	

Figure 1.4 Multiple Processes and Modes of Execution

Although the system executes in one of two modes, the kernel runs on behalf of a user process. The kernel is not a separate set of processes that run in parallel to user processes, but it is part of each user process.

Interrupts and Exceptions:

The UNIX system allows devices such as I/O peripherals or the system clock to interrupt the CPU asynchronously. On receipt of the interrupt, the kernel saves its current context, determines the cause of the interrupt, and services the interrupt. After the kernel services the interrupt, it restores its interrupted context and proceeds as if nothing had happened.

Tile hardware usually prioritizes devices according to the order that interrupts should be handled: When the kernel services an interrupt, it blocks out lower priority interrupts but services higher priority interrupts.

An exception condition refers to unexpected events caused by a process, such as addressing illegal memory, executing privileged instructions, dividing by zero, and so on. They are distinct from interrupts, which are caused by events that are external to a process.

Exceptions happen "in the middle" of the execution of an instruction, and the system attempts to restart the instruction after handling the exception; interrupts are considered to happen between the execution of two instructions, and the system continues with the next instruction after servicing the interrupt. The UNIX system uses one mechanism to handle interrupts and exception conditions.

Processor Execution Levels:

The kernel must sometimes prevent the occurrence of interrupts during critical activity, which could result in corrupt data if interrupts were allowed. For instance, the kernel may not want to receive a disk interrupt while manipulating linked lists, because handling the interrupt could corrupt the pointers.

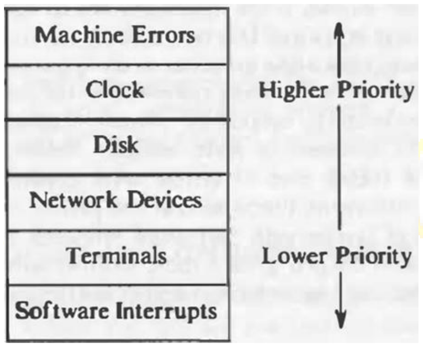


Figure 1.5 Typical Interrupt levels

Figure 1.5 shows a sample set of execution levels. If the kernel masks out disk interrupt, all interrupts except for clock interrupts and machine error interrupts are prevented. If it masks out software interrupts, all other interrupts may occur.

Memory Management:

The kernel permanently resides in main memory as does the currently executing process. When compiling a program, the compiler generates a set of addresses in the program that represent addresses of variables and data structures or the addresses of instructions such as functions.

The compiler generates the addresses for a virtual machine as if no other program will execute simultaneously on the physical machine. When the program is to run on the machine, the kernel allocates space in main memory for it, but the virtual addresses generated by the compiler need not be identical to the physical addresses that they occupy in the machine.

The kernel coordinates with the machine hardware to set up a virtual to physical address translation that maps the compiler-generated addresses to the physical machine addresses. The mapping depends on the capabilities of the machine hardware, and the parts of UN IX systems that deal with them are therefore machine dependent.

INTRODUCTION TO THE KERNEL:

ARCHITECTURE OF THE UNIX OPERATING SYSTEM:

Figure 1.6 gives a block diagram of the kernel, showing various modules and their relationships to each other. In particular, it shows the file subsystem on the left and the process control subsystem on the right, the two major components of the kernel. The diagram serves as a useful logical view of the kernel, although in practice the kernel deviates from the model because some modules interact with the internal operations of others.

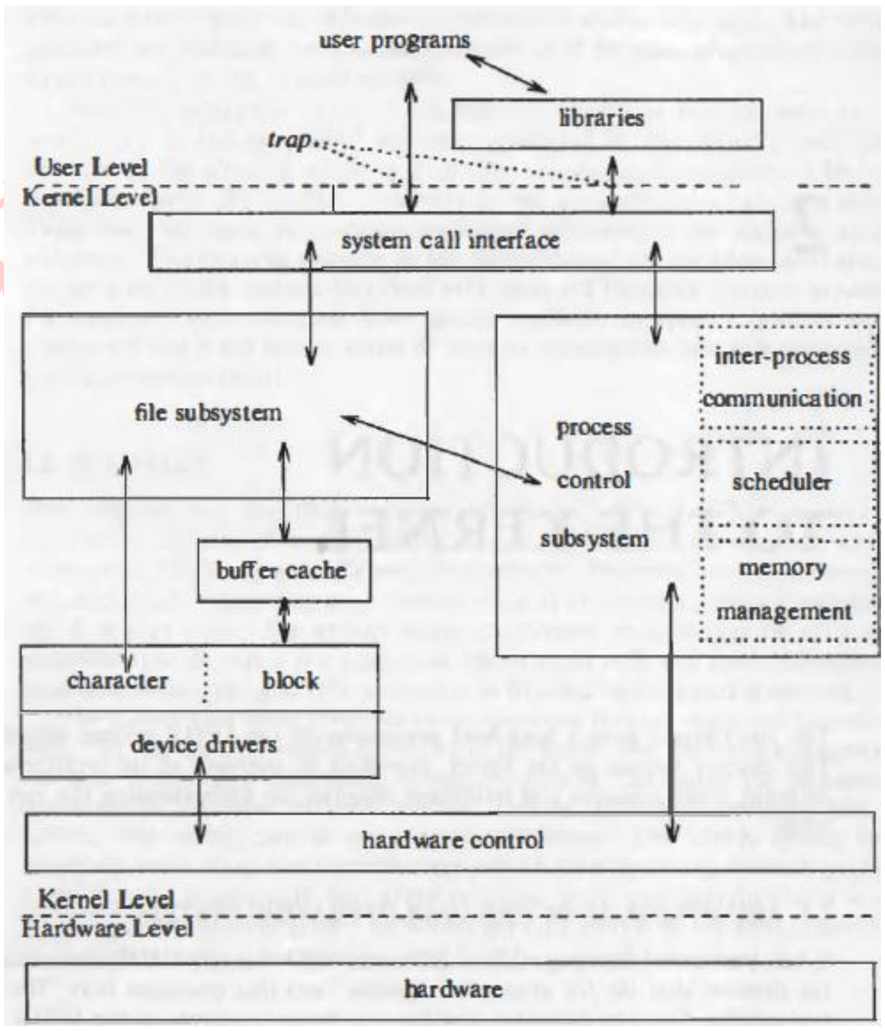


Figure 1.6 Block Diagram of the System Kernel

Figure 1.6 shows three levels: user, kernel, and hardware. The system call and library interface represent the border between user programs and the kernel depicted in Figure 1.1. System calls look like ordinary function calls in C programs, and libraries map these function calls to the primitives needed to enter the operating system.

Assembly language programs may invoke system calls directly without a system call library, however. Programs frequently use other libraries such as the standard 1/0 library to provide a more sophisticated use of the system calls. The libraries are linked with the programs at compile time.

The figure partitions the set of system calls into those that interact with the file subsystem and those that interact with the process control subsystem.

The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users.

Processes interact with the file subsystem via a specific set of system calls, such as open (to open a file for reading or writing), close, read, write, stat (query the attributes of a file), chown (change the record of who owns the file), and chmod (change the access permissions of a file).

JKDirectory!

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. Device drivers are the kernel modules that control the operation of peripheral devices. Block I/O devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system.

For example, a tape driver may allow the kernel to treat a tape unit as a random access storage device. The file subsystem also interacts directly with "raw" I/O device drivers without the intervention of a buffering mechanism. Raw devices, sometimes called character devices, include all devices that are not block devices.

The process control subsystem is responsible for process synchronization, inter process communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact when loading a file into memory for execution where the process subsystem reads executable files into memory before executing them.

Some of the system calls for controlling processes are: fork (create a new process), exec (overlay the image of a program onto the running process), exit (finish executing a process), wait (synchronize process execution with the exit of a previously forked process), brk (control the size of memory allocated to a process), and signal (control process response to extraordinary events).

The memory management module controls the allocation of memory. If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute.

The scheduler module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it is the highest priority eligible process available.

INTRODUCTION TO SYSTEM CONCEPTS

AN OVERVIEW OF THE FILE SUBSYSTEM:

The internal representation of a file is given by an inode, which contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times. The term inode is a contraction of the term index node and is commonly used in literature on the UNIX system. Every file has one inode, but it may have several names, all of which map into the inode.

JKDirectory!

Each name is called a link. When a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file.

For example, if a process calls ***open ("/fs2/mjb/rje/sourcefile", l);*** the kernel retrieves the inode for *"/fs2/mjb/rje/sourcefile"*. When a process creates a new file, the kernel assigns it an unused inode.

The kernel contains two other data structures, the ***file table*** and the ***user file descriptor*** table. ***The file table is a global kernel structure, but the user file descriptor table is allocated per process.***

When a process opens or creates a file, the kernel allocates an entry from each table, corresponding to the file's inode. Entries in the three structures - user file descriptor table, file table, and inode table - maintain the state of the file and the user's access to it.

The file table keeps track of the byte offset in the file where the user's next read or write will start, and the access rights allowed to the opening process. The user file descriptor table identifies all open files for a process.

Figure 1.7 shows the tables and their relationship to each other.

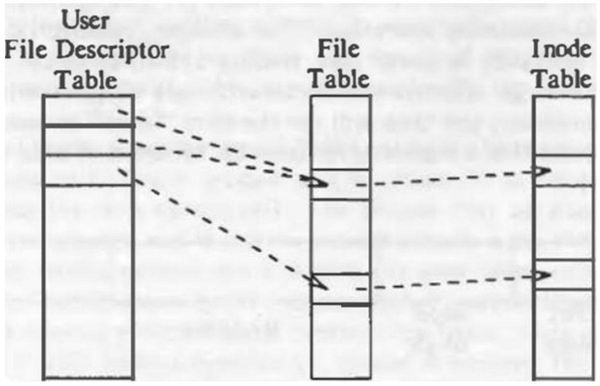


Figure 1.7 File Descriptors, File Table, and Inode Table

The kernel returns a file descriptor for the *open* and *creat* system calls, which is an index into the user file descriptor table. When executing *read* and *write* system calls, the kernel uses the file descriptor to access the user file descriptor table, follows pointers to the file table and inode table entries and, from the inode, finds the data in the file.

A file system consists of a sequence of logical blocks, each containing 512, 1024, 2048, or any convenient multiple of 512 bytes, depending on the system implementation. The size of a logical block is homogeneous within a file system but may vary between different file systems in a system configuration.

JKDirectory!

Using large logical blocks increases the effective data transfer rate between disk and memory, because the kernel can transfer more data per disk operation and therefore make fewer time-consuming operations.

A file system has the following structure (Figure 1.8):

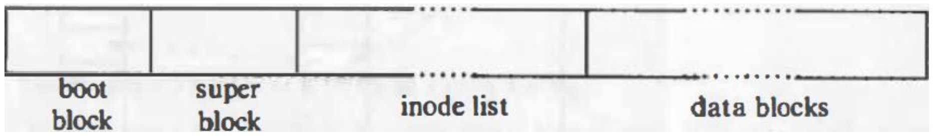


Figure 1.8 File System Layouts

The boot block occupies the beginning of a file system, typically the first sector, and may contain the bootstrap code that is read into the machine to boot, or initialize, the operating system. Although only one boot block is needed to boot the system, every file system has a (possibly empty) boot block.

The super block describes the state of a file system - how large it is, how many files it can store, where to find free space on the file system, and other information.

The inode list is a list of inode's that follows the super block in the file system. Administrators specify the size of the inode list when configuring a file system. The kernel references inode's by index into the inode list.

The data blocks start at the end of the inode list and contain file data and administrative data. An allocated data block can belong to one and only one file in the file system.

Processes:

A process is the execution of a program and consists of a pattern of bytes that the CPU interprets as machine instructions which is called as "text", stack and data.

Many processes appear to execute simultaneously as the kernel schedules them for execution, and several processes may be instances of one program.

A process executes by following a strict sequence of instructions that is self-contained and does not jump to that of another process; it reads and writes its data and stack sections, but it cannot read or write the data and stack of other processes. Processes communicate with other processes and with the rest of the world via system calls.

JKDirectory!

A process on a UNIX system is the entity that is created by the fork system call. Every process except process 0 is created when another process executes the fork system call. The process that invoked the fork system call is the parent process, and the newly created process is the child process.

Every process has one parent process, but a process can have many child processes. The kernel identifies each process by its process number, called the process ID (PID).

Process 0 is a special process that is created "by hand" when the system boots; after forking a child process (process 1), process 0 becomes the swapper process. Process 1, known as init, is the ancestor of every other process in the system and enjoys a special relationship with them.

The kernel loads an executable file into memory during an exec system call, and the loaded process consists of at least three parts, called regions: text, data, and the stack.

The text and data regions correspond to the text and data sections of the executable file, but the stack region is automatically created and its size is dynamically adjusted by the kernel at run time.

The stack consists of logical stack frames that are pushed when calling a function and popped when returning; a special register called the stack pointer indicates the current stack depth.

A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the program counter and stack pointer at the time of the function call.

The program code contains instruction sequences that manage stack growth, and the kernel allocates space for the stack, as needed.

The kernel stack contains the stack frames for functions executing in kernel mode. The function and data entries on the kernel stack refer to functions and data in the kernel, not the user program, but its construction is the same as that of the user stack.

The kernel stack of a process is null when the process executes in user mode. Every process has an entry in the kernel process table, and each process is allocated a u area that contains private data manipulated only by the kernel.

The process table contains (or points to) a per process region table, whose entries point to entries in a region table. A region is a contiguous area of a process's address space, such as text, data, and stack.

Region table entries describe the attributes of the region, such as whether it contains text or data, whether it is shared or private, and where the "data" of the region is located in memory.

JKDirectory!

When a process invokes the exec system call, the kernel allocates regions for its text, data, and stack after freeing the old regions the process had been using.

When a process invokes fork, the kernel duplicates the address space of the old process, allowing processes to share regions when possible and making a physical copy otherwise. When a process invokes exit, the kernel frees the regions the process had used.

Figure 1.9 shows the relevant data structures of a running process: The process table points to a per process region table with pointers to the region table entries for the text, data, and stack regions of the process.

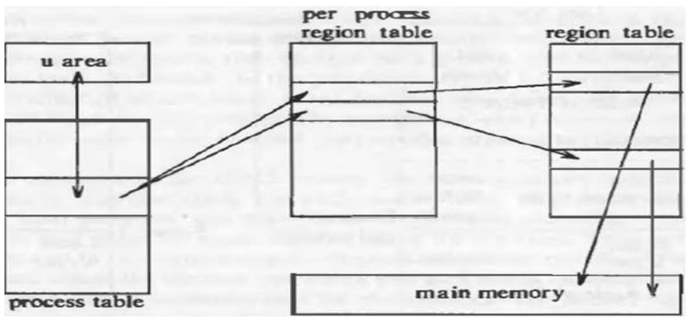


Figure 1.9 Data structures for processes

The process table entry and the u area contain control and status information about the process. The u area is an extension of the process table entry. The u area contains information describing the process that needs to be accessible only when the process is executing. The important fields are:

- A pointer to the process table slot of the currently executing process.
- Parameters of the current system call, return values and error codes.
- File descriptors for all open files.
- Internal I/O parameters.
- Current directory and current root.
- Process and file size limits.

The kernel can directly access fields of the u area of the executing process but not of the u area of other processes.

Context of a process:

JKDirectory!

The context of a process is its state, as defined by its text, the values of its global user variables and data structures, the values of machine registers it uses, the values stored in its process table slot and u area, and the contents of its user and kernel stacks.

The text of the operating system and its global data structures are shared by aU processes but do not constitute part of the context of a process.

When executing a process, the system is said to be executing in the context of the process. When the kernel decides that it should execute another process, it does a context switch, so that the system executes in the context of the other process. The kernel allows a context switch only under specific conditions.

When doing a context switch, the kernel saves enough information so that it can later switch back to the first process and resume its execution. Similarly, when moving from user to kernel mode, the kernel saves enough information so that it can later return to user mode and continue execution from where it left off.

Note: Moving between user and kernel mode is a change in mode, not a context switch.

Process states:

The lifetime of a process can be divided into a set of states as described below, each with certain characteristics that describe the process.

1. The process is currently executing in user mode.
2. The process is currently executing in kernel mode.
3. The process is not executing, but it is ready to run as soon as the scheduler chooses it. Many processes may be in this state, and the scheduling algorithm determines which one will execute next.
4. The process is sleeping. A process puts itself to sleep when it can no longer continue executing, such as when it is waiting for I/O to complete.

Because a processor can execute only one process at a time, at most one process may be in states 1 and 2. The two states correspond to the two modes of execution, user and kernel.

State transition Diagram:

The process states described above give a static view of a process, but processes move continuously between the states according to well-defined rules.

A state transition diagram is a directed graph whose nodes represent the states a process can enter and whose edges represent the events that cause a process to move from one state to another.

JKDirectory!

State transitions are legal between two states if there exists an edge from the first state to the second. Several transitions may emanate from a state, but a process will follow one and only one transition depending on the system event that occurs. Figure 2.0 shows the state transition diagram for the process states defined above.

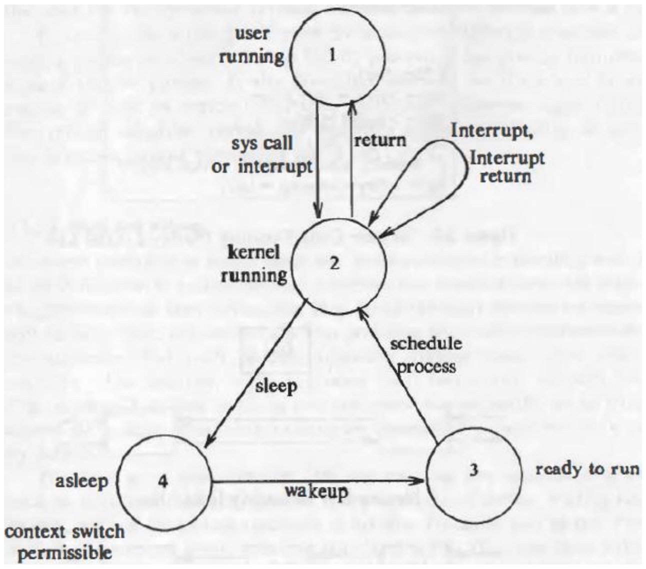


Figure 2.0: Processes States and Transitions

Several processes can execute simultaneously in a time-shared manner, and they may all run simultaneously in kernel mode.

If they were allowed to run in kernel mode without constraint, they could corrupt global kernel data structures. By prohibiting arbitrary context switches and controlling the occurrence of interrupts, the kernel protects its consistency.

The kernel allows a context switch only when a process moves from the state "kernel running" to the state "asleep in memory."

Processes running in kernel mode cannot be preempted by other processes; therefore the kernel is sometimes said to be non-preemptive, although the system does preempt processes that are in user mode.

The kernel maintains consistency of its data structures because it is non-preemptive, thereby solving the mutual exclusion problem - making sure that critical sections of code are executed by at most one process at a time.

Sleep and wakeup:

JKDirectory!

A process executing in kernel mode has great autonomy in deciding what it is going to do in reaction to system events. Processes can communicate with each other and "suggest" various alternatives, but they make the final decision by themselves.

Processes go to sleep because they are awaiting the occurrence of some event, such as waiting for I/O completion from a peripheral device, waiting for a process to exit, waiting for system resources to become available, and so on.

Processes are said to sleep on an event, meaning that they are in the sleep state until the event occurs, at which time they wake up and enter the state "ready to run."

When a process wakes up, it follows the state transition from the "sleep" state to the "ready-to-run" state, where it is eligible for later scheduling; it does not execute immediately.

Sleeping processes do not consume CPU resources: The kernel does not constantly check to see that a process is still sleeping but waits for the event to occur and awakens the process then.

KERNEL DATA STRUCTURES:

Most kernel data structures occupy fixed-size tables rather than dynamically allocated space. The advantage of this approach is that the kernel code is simple, but it limits the number of entries for a data structure to the number that was originally configured when generating the system: If, during operation of the system, the kernel should run out of entries for a data structure, it cannot allocate space for new entries dynamically but must report an error to the requesting user.

If, on the other hand, the kernel is configured so that it is unlikely to run out of table space, the extra table space may be wasted because it cannot be used for other purposes.

SYSTEM ADMINISTRATION:

Administrative processes are loosely classified as those processes that do various functions for the general welfare of the user community.

Such functions include disk formatting, creation of new file systems, repair of damaged file systems, kernel debugging, and others. Conceptually, there is no difference between administrative processes and user processes: They use the same set of system calls available to the general community. They are distinguished from general user processes only in the rights and privileges they are allowed.

JKDirectory!