

OPERATING SYSTEM

Chapter 8

CASE STUDIES: UNIX

Detailed Study Notes for Engineering Students

Topics Covered

8.1 General Overview of Unix System: System Structure, Operating System Structure

8.2 Introduction to Kernel: Kernel Data Structures, System Administration

8.3 Internal Representation of Files: Inodes, Structure of a Regular File, Super Block

15 MCQs with Answer Key | 10 Broad Questions

Contents

- 8.1 General Overview of Unix System
- 8.2 Introduction to the Kernel
- 8.3 Internal Representation of Files
- 8.4 Chapter Summary and Quick Revision
- 8.5 Multiple Choice Questions (15)
- 8.6 Answer Key
- 8.7 Broad Questions (10)

Poly Notes Hub

8.1 General Overview of Unix System

UNIX is a powerful, multi-user, multitasking, time-sharing operating system. It was developed at **AT&T Bell Laboratories** in 1969 by Ken Thompson and Dennis Ritchie, and rewritten in the **C language** in 1973, which made it one of the first portable operating systems. Two major families grew from it: **System V** (AT&T) and **BSD** (University of California, Berkeley). Modern descendants and look-alikes include Solaris, AIX, HP-UX, macOS and Linux. A case study of UNIX is valuable because its design is clean, small and has influenced almost every later operating system.

8.1.1 Main Features of UNIX

- **Multi-user and multitasking:** many users can work at the same time and each user can run many programs.
- **Portability:** because it is written mainly in C, it can be moved to different hardware with little change.
- **Hierarchical file system:** files are organised as a tree with a single root directory (/).
- **Everything is a file:** ordinary files, directories and devices are all accessed with the same system calls (open, read, write, close).
- **Shell and command language:** a user-replaceable command interpreter that also acts as a programming language (shell scripts).
- **Pipes and filters:** small programs can be combined, for example `ls | wc -l`, so the output of one is the input of the next.
- **Security:** file permissions (read, write, execute for owner, group and others) and user accounts protect resources.
- **Device independence:** a program need not know whether it is reading from a file, a terminal or a device.

8.1.2 System Structure

The UNIX system has a layered structure. The innermost layer is the **hardware**. The **kernel** surrounds it and interacts directly with it, providing common services to programs and hiding hardware details. Above the kernel is the **system-call interface** together with **libraries**, and on top are the **shell, utilities and user application programs**. Programs never touch the hardware directly; they request services from the kernel through system calls.

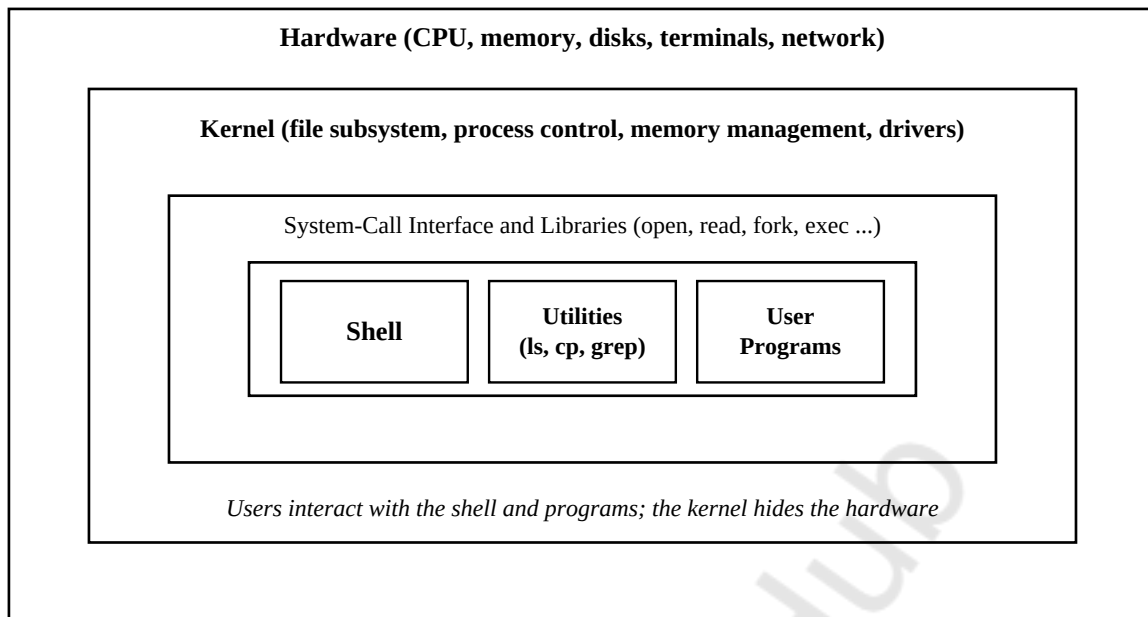


Fig. 8.1: Layered structure of the UNIX system

Layer	Role
Hardware	CPU, main memory, disks, terminals, network interfaces.
Kernel	Memory-resident core of the OS; manages files, processes, memory and devices; runs in privileged (kernel) mode.
System-call interface and libraries	Entry points to the kernel (for example, open, read, fork, exec). Library functions such as those of the C library wrap the system calls.
Shell	Command interpreter that reads user commands and runs programs (sh, csh, ksh, bash).
Utilities and applications	Programs such as ls, cp, grep, vi, compilers, editors and user programs.

8.1.3 Operating System Structure

The UNIX operating system is divided into three levels: the **user level**, the **kernel level** and the **hardware level**. The **system-call interface** and **libraries** represent the border between user programs and the kernel. System calls look like ordinary C functions, and the libraries map these function calls to the primitives needed to enter the operating system. The kernel itself has two major parts, the **file subsystem** and the **process control subsystem**.

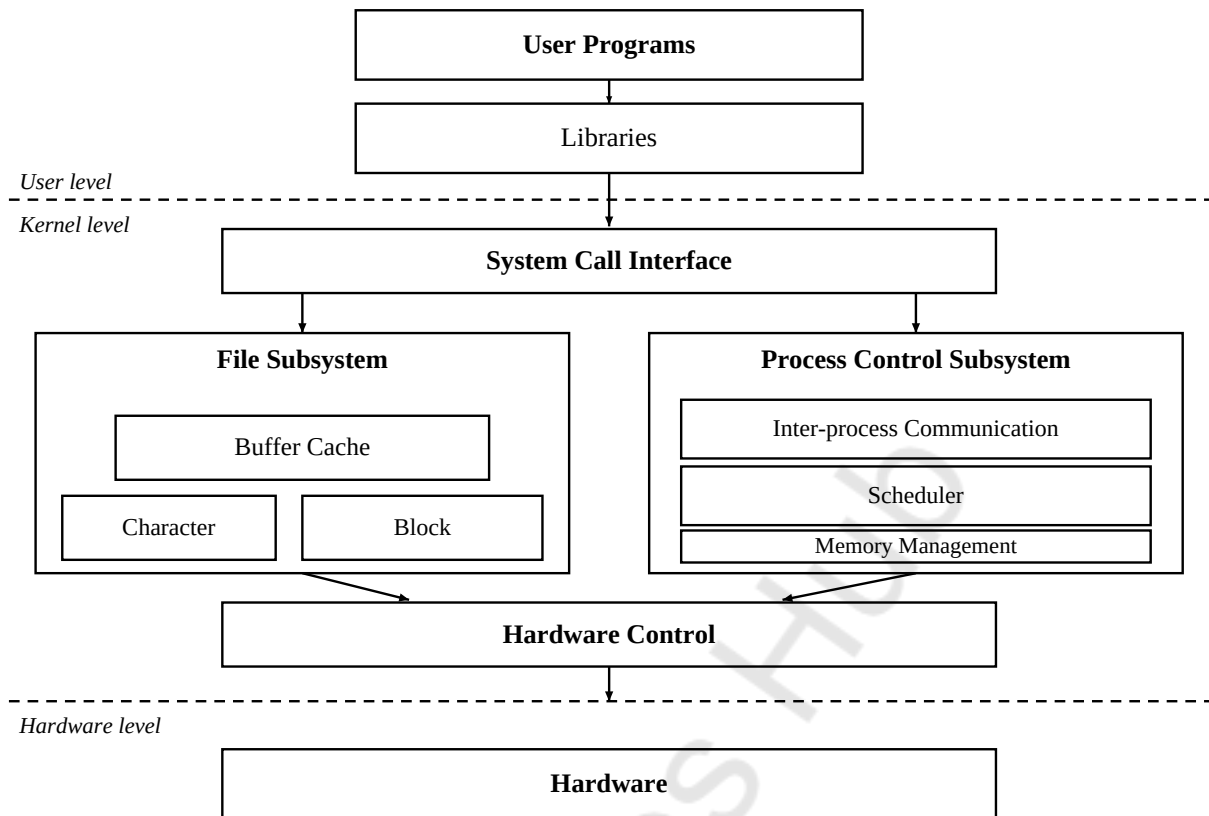


Fig. 8.2: Block diagram of the UNIX system kernel

File Subsystem

The file subsystem manages files, allocates file space, administers free space, controls access to files and retrieves data for users. Processes interact with the file subsystem through specific system calls such as open, close, read, write, stat, chown and chmod. It accesses file data using a **buffer mechanism (buffer cache)** that regulates data flow between the kernel and secondary storage devices. The buffer 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 devices (such as disks) are random-access storage devices; the file system also interacts directly with **raw (character) I/O device drivers** without the intervention of the buffer mechanism. Character devices include all devices that are not block devices, such as terminals.

Process Control Subsystem

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

- **Memory management module:** controls the allocation of memory. If the system does not have enough physical memory at any time, the kernel moves processes between main memory and secondary memory (swapping or paging) so that all processes get a fair chance to execute.
- **Scheduler:** allocates the CPU to processes, which run until they voluntarily relinquish the CPU while waiting for a resource or until the kernel pre-empted them when their time slice expires.
- **Interprocess communication (IPC):** allows processes to exchange data and synchronise, through pipes, signals, messages, shared memory and semaphores.

- **Hardware control:** responsible for handling interrupts and for communication with the machine; devices such as disks or terminals may interrupt the CPU while a process is executing, and the kernel resumes the interrupted process after servicing.

User Mode and Kernel Mode

Basis	User mode	Kernel mode
Executes	User programs and library code	Kernel code (system calls, interrupt handlers)
Privilege	Restricted; privileged instructions not allowed	Full; can execute all instructions and access all memory and devices
Entry	Normal execution of a process	On a system call, interrupt or exception
Memory access	Only its own address space	Its own and the kernel address space

Categories of System Calls

Category	Examples
File management	open, close, read, write, lseek, creat, link, unlink, stat, chmod, chown
Process control	fork, exec, exit, wait, kill, getpid, nice
Interprocess communication	pipe, signal, msgget, shmget, semget
Device and file system	mount, umount, ioctl, mknod
Information and time	time, alarm, getuid, sleep

8.1.4 Standard File System Layout

Directory	Contents
/	Root of the file system tree
/bin	Essential user commands (binaries)
/etc	System configuration files (passwd, group, inittab)
/dev	Special files that represent devices
/usr	User programs, libraries and documentation
/home	Home directories of users
/tmp	Temporary files
/var	Variable data such as logs and spool files
/mnt	Mount point for other file systems

8.2 Introduction to the Kernel

The **kernel** is the core of the UNIX operating system. It is a program that is loaded into memory when the system is booted and stays there until shutdown. It provides all the essential services (process management, memory management, file management and device I/O) and enforces protection. The kernel is not a separate process; it executes on behalf of user processes when they make a system call and also when a hardware interrupt arrives. Its code is kept in the file /unix (or /vmunix, /kernel) on disk.

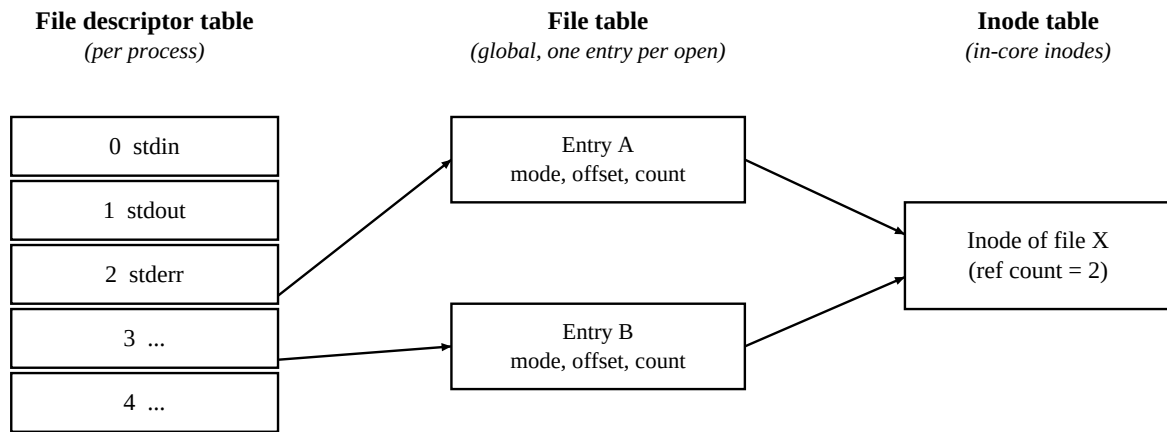
Services Provided by the Kernel

- Controlling the execution of processes: creation, termination, suspension and inter-process communication.
- Scheduling processes fairly for execution on the CPU.
- Allocating main memory for an executing process, including swapping and paging.
- Allocating secondary memory for efficient storage and retrieval of user data (the file system).
- Allowing processes controlled access to peripheral devices such as terminals, disk drives and network devices.

8.2.1 Kernel Data Structures

The kernel keeps most of its information in **tables** and linked structures stored in kernel memory. Most of these are fixed-size arrays or lists that are allocated when the kernel starts. The important data structures are described below.

Data structure	Description
Process table	One entry (proc structure) per process, always resident in memory. It holds process ID, state, priority, signals, pointers to the memory regions and to the u area, and other information needed even when the process is not running.
u area (user area)	Per-process information needed only while the process is running: user file-descriptor table, current directory, user and group IDs, signal handlers, and limits. It is swapped out with the process.
Region table	Describes regions of memory (text, data, stack): where each is located, its size, and a reference count for sharing.
Per-process region table	Maps virtual addresses of a process to its regions.
User file-descriptor table	In the u area; indexed by file descriptor (0, 1, 2, ...), each entry points to an entry in the file table.
File table	Global table with one entry for each open instance of a file. It stores the mode (read or write), the current byte offset and a pointer to the inode.
Inode table (in-core inodes)	Memory copy of the disk inodes of files that are currently in use.
Mount table	Information about each mounted file system: device, root inode and super block.
Buffer cache	Pool of buffers (with buffer headers, hash queues and a free list) that hold recently used disk blocks.
Device switch tables	bdevsw (block) and cdevsw (character) tables that map a device's major number to the driver routines.



Two opens of the same file X: separate offsets, one shared inode

Fig. 8.3: Relationship between the file-descriptor table, file table and inode table

8.2.2 Process Structure and Context

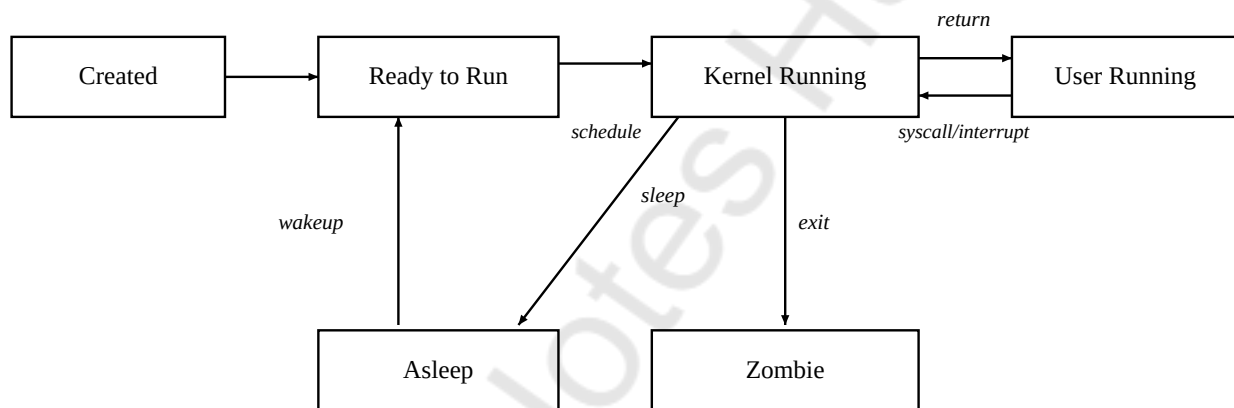
A **process** is the execution of a program. Its image in memory consists of a **text** region (program code, shareable), a **data** region (initialised and uninitialised data) and a **stack** region. The **context** of a process is its state, defined by its text, values of global user variables and data structures, values of machine registers, values stored in its process table slot and u area, and the contents of its user and kernel stacks. It has three parts:

- **User-level context:** text, data, user stack and shared memory regions.
- **Register context:** program counter, processor status register, stack pointer and general registers.
- **System-level context:** process table entry, u area, region tables and the kernel stack (static part and dynamic part).

When the kernel decides to run another process, it performs a **context switch**: it saves the context of the old process and loads the context of the new one. A system call (trap) or an interrupt causes a **mode switch** from user mode to kernel mode, which is not the same as a context switch because the same process continues.

8.2.3 Process States

State	Meaning
Created	Process is newly created (by fork) and is not yet ready or asleep.
Ready to run (in memory)	Process is ready and will run as soon as the scheduler picks it.
User running	Process is executing in user mode.
Kernel running	Process is executing in kernel mode (a system call or interrupt).
Asleep (in memory)	Process cannot continue until an event occurs, such as the completion of I/O.
Ready and swapped	Ready but the process is in secondary memory; the swapper must bring it in.
Sleeping and swapped	Process is both asleep and swapped out.
Pre-empted	Process is returning from kernel to user mode but the kernel pre-empts it to run another process.
Zombie	Process has called exit and no longer exists, but its record (exit status) remains for the parent.



Simplified process state diagram of UNIX

Fig. 8.4: Simplified process state transitions

8.2.4 System Administration

System administration means the tasks required to keep a UNIX system running properly, securely and efficiently. It is performed by the **system administrator**, who works as the **superuser (root)**. The root user has user ID 0 and is not subject to the usual permission checks, so it must be used with care. Administrative processes are the **init** process (process ID 1, the ancestor of all other processes), and the **getty** and **login** processes that start a user session.

Typical Administrative Tasks

Task	Description	Commands / files
User and group management	Create, modify and delete accounts; set passwords, home directory and shell	useradd, usermod, userdel, passwd; /etc/passwd, /etc/group, /etc/shadow
Starting and stopping the system	Boot the kernel, start init and services, shut down safely	init, shutdown, reboot; /etc/inittab, run levels
File-system management	Create file systems, mount and unmount them, check consistency	mkfs, mount, umount, fsck, df, du
Backup and restore	Take periodic copies to protect against loss	tar, cpio, dump, restore
Process management	Monitor the system and kill runaway processes	ps, top, kill, nice, cron, at
Security	Set file permissions, monitor logins, apply updates	chmod, chown, last, who
Software and device installation	Add programs, drivers and peripherals	package tools, mknod
Logging and monitoring	Study system logs and performance	/var/log, syslog, vmstat, iostat

Boot and Login Sequence

1. The bootstrap program loads the kernel from disk into memory and starts it.
2. The kernel initialises its data structures, mounts the root file system and creates process 0 (the swapper).
3. Process 0 forks process 1, **init**, which reads /etc/inittab and starts the system services.
4. init spawns a **getty** process for each terminal line, which displays the login prompt.
5. After a user types the name, getty runs **login**, which checks /etc/passwd; on success it starts the user's **shell**.

8.3 Internal Representation of Files

Every file in UNIX has a unique **inode**. The inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size and the location of the file's data on disk. Processes access files by a well-defined set of system calls and specify a file by a character-string path name; each path name component is converted to an inode, and the kernel uses the inode to find the file's data. Names of files are kept in directories, not in the inode.

8.3.1 Layout of a File System on Disk

A UNIX file system is a sequence of logical blocks (commonly 512 bytes, 1 KB or more per block). Each file system occupies a disk partition or logical volume and has the following parts:

Boot block (block 0)	Super block (block 1)	Inode list (inode 1, 2, 3 ...)	Data blocks (files, directories, indirect blocks)
-------------------------	--------------------------	-----------------------------------	--

Layout of a UNIX file system on disk

Fig. 8.5: Layout of a UNIX file system

Part	Contents
Boot block	Occupies the beginning of the file system (block 0). It contains the bootstrap code that loads the operating system; only one boot block is used on the booting file system but each has one.
Super block	Describes the state of the file system: its size, how many files it can store, where free space is, and more.
Inode list	A list of inodes that follows the super block. Administrators specify its size when configuring the file system. The kernel refers to inodes by an index into this list (inode number).
Data blocks	Start after the inode list; contain file data, directory data and indirect blocks. An allocated data block belongs to exactly one file.

8.3.2 Inodes

Inodes exist in a static form on disk and the kernel reads them into an in-core (memory) inode to manipulate them. The **disk inode** contains the following fields:

Field	Description
File owner identifier	User ID (UID) and group ID (GID) of the owner.
File type	Regular file, directory, character special, block special, or FIFO (pipe). An inode with type zero is free.
File access permissions	Nine permission bits (read, write, execute for owner, group and others) plus the set-user-ID, set-group-ID and sticky bits.
File access times	Time the file was last modified, time it was last accessed, and time the inode was last changed.
Number of links	Number of names (directory entries) that refer to the file.
Table of contents (disk addresses)	Array of 13 entries pointing to the data blocks of the file: 10 direct, 1 single indirect, 1 double indirect and 1 triple indirect.
File size	Number of bytes in the file.

Important: The inode does not contain the file name. The name is stored in a directory entry, which pairs a name with an inode number. This is why one file can have several names (hard links).

The **in-core inode** contains all the fields of the disk inode plus the following:

Extra field in in-core inode	Meaning
Status	Flags: inode is locked, a process is waiting for it to be unlocked, the inode data differs from the disk copy, the file data differs from the disk copy, or the file is a mount point.
Logical device number	Device of the file system that contains the file.
Inode number	Position of the inode in the inode list on disk (disk inodes do not store this because the position gives it).
Pointers to other in-core inodes	Hash queue and free-list pointers used to find an inode quickly.
Reference count	Number of instances of the file that are active (for example, opened).

Directories

A **directory** is a file whose data is a sequence of entries. In classic System V each entry has 16 bytes: a 2-byte **inode number** and a 14-byte **file name**. The first two entries are always "." (the directory itself) and ".." (its parent). An entry with inode number 0 is empty. The kernel function **namei** converts a path name into an inode by parsing the name one component at a time, starting at the root inode (for a path starting with /) or the current directory inode, and searching each directory for the next component.

Accessing Inodes: *iget* and *iput*

- **iget** returns a locked in-core inode for a given inode number; it checks the inode hash queue first, and otherwise takes an inode from the free list and reads the disk inode into it.
- **iput** releases the in-core inode; it decrements the reference count, and when the count becomes zero and the link count is zero it frees the file's data blocks and the inode (and puts the inode on the free list).

8.3.3 Structure of a Regular File

A file's data is stored in disk blocks that need not be contiguous. To let the data of a large file be found quickly while keeping the inode small, the table of contents in the inode holds **13 entries**: the first **10 are direct** block addresses, the 11th is a **single-indirect** block, the 12th is a **double-indirect** block and the 13th is a **triple-indirect** block. This scheme also lets the kernel treat a file as a sequence of bytes, so a byte offset must be converted to a logical block number and then to a physical block.

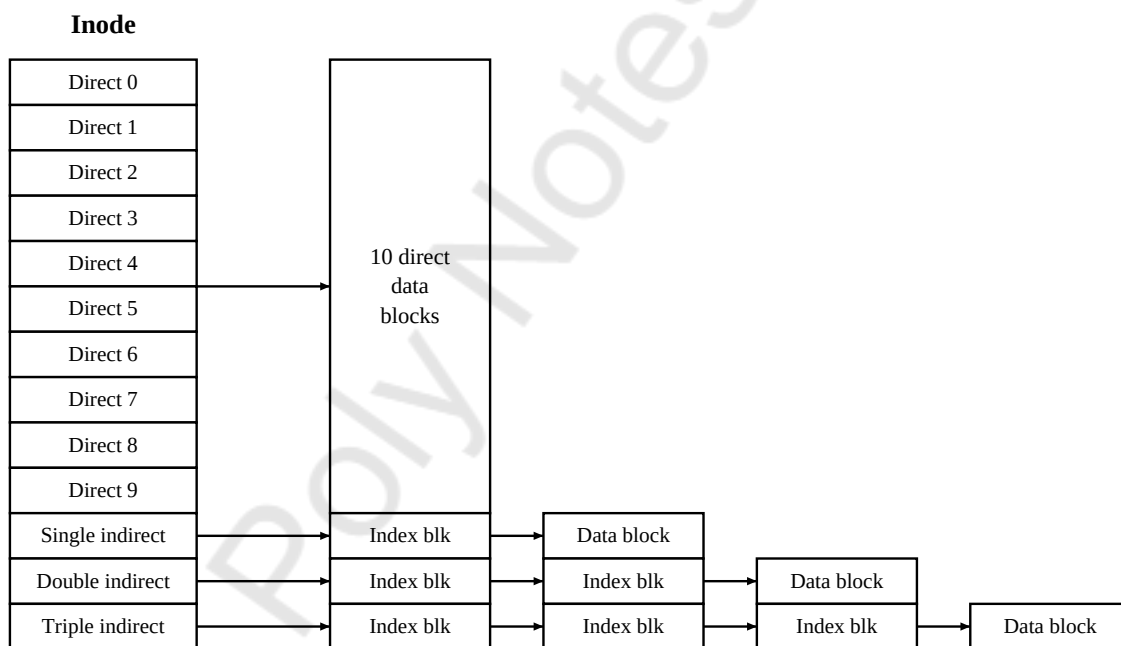


Fig. 8.6: Block addressing in an inode (direct and indirect blocks)

Maximum File Size

Assume a block size of 1 KB (1024 bytes) and a disk block number that needs 4 bytes. Then one indirect block can hold $1024 / 4 = 256$ block numbers.

Level	Blocks addressed	Size (1 KB blocks)
10 direct blocks	10	10 KB
Single indirect	256	256 KB
Double indirect	$256 \times 256 = 65,536$	64 MB
Triple indirect	$256 \times 256 \times 256 = 16,777,216$	16 GB
Total (approx.)	$10 + 256 + 65,536 + 16,777,216$	about 16 GB

In the original system the file-size field is a 32-bit number, so the real maximum file size is limited to 2^{32} bytes (4 GB), even though the triple-indirect block could address more.

Example: Finding the Block for a Byte Offset

Byte offset	Logical block (offset / 1024)	How the block is found
9,000	8	Within the first 10, so it is a direct block (entry 8).
350,000	341	Blocks 0 to 9 are direct; 10 to 265 are single-indirect; 266 to 65,801 are double-indirect . Block 341 is in the double-indirect range: $341 - 266 = 75$, so use entry 0 of the first-level index and entry 75 of the second-level index.

A process reads a byte offset by calling the kernel algorithm **bmap**, which converts the logical byte offset in the file to a physical disk block number, using the inode table of contents and, if needed, reading indirect blocks from disk (these reads may come from the buffer cache).

Advantages and Notes

- Small files (the majority) are reached with direct blocks and need no extra disk reads.
- Large files need up to three extra block reads, but indirect blocks are cached.
- Files can have **holes**: a block address of zero in the inode means that part of the file has never been written and reads as zeros; no disk space is used for it.
- Disk blocks need not be contiguous, so external fragmentation of the file system is avoided.

8.3.4 Super Block

The **super block** describes the state of a file system. It contains the following fields:

Field	Description
Size of the file system	Total number of blocks.
Number of free blocks	Free blocks available in the file system.
List of free blocks	Array holding the numbers of free blocks (a partial list).
Index of next free block	Pointer to the next entry in the free-block list.
Size of the inode list	Number of inodes in the file system.
Number of free inodes	Count of unused inodes.
List of free inodes	Array holding numbers of free inodes (a partial list).
Index of next free inode	Pointer to the next entry in the free-inode list.
Lock fields	Locks for the free-block list and the free-inode list.
Modified flag	Shows that the super block has been changed and must be written back to disk.

The super block is read into memory when the file system is **mounted** and written back periodically (and at unmount), so the in-memory copy may be newer than the disk copy; this is why a system must be shut down correctly. If the system crashes, **fsck** is used to check and repair the file system.

Allocation of Inodes

When a process needs a new inode (for example, to create a file), the kernel runs the algorithm **ialloc**. The super block holds only a partial list of free inode numbers. If the list is not empty, the kernel takes the next number, gets the inode with **iget**, initialises it and decrements the free-inode count. If the list is empty, the kernel searches the disk inode list for free inodes (type zero), refills the super block list, and remembers the highest inode number found so that the next search starts from there. When an inode is freed (**ifree**), its number is placed in the super block list if there is room.

Allocation of Disk Blocks

Free disk blocks are kept as a **linked list of index blocks**. The super block holds a partial list of free block numbers; the first block number in the list is a link to a disk block which holds more free block numbers (up to 100 in the classic design). To allocate a block (**alloc**), the kernel takes the next free block number from the super block list; when only one entry remains, that entry points to a block of further free-block numbers, which is read into the super block list. To free a block (**free**), the number is added to the super block list; if the list is full, the old list is written into the freed block, and the freed block becomes the new head of the chain.

Basis	Inode allocation	Disk-block allocation
List in super block	Partial list of free inode numbers	Partial list of free block numbers
When list is empty	Search the inode list on disk for free inodes	Read the next block of the free-block chain into the list
Free operation	Number added to the list if room (no linking)	Number added; full list is written into the freed block
Special aid	Remembered inode number to speed up the next search	Linked chain of index blocks

8.4 Chapter Summary and Quick Revision

- UNIX is a multi-user, multitasking, portable OS written in C; developed at Bell Labs; main families System V and BSD.
- Structure: hardware, kernel, system-call interface and libraries, then shell and user programs; three levels: user, kernel and hardware.
- The kernel has two subsystems: the **file subsystem** (buffer cache, character and block drivers) and the **process control subsystem** (IPC, scheduler, memory management).
- Key kernel data structures: process table, u area, region table, file-descriptor table, file table, inode table, mount table and buffer cache.
- Process states include created, ready, user running, kernel running, asleep, swapped states, pre-empted and zombie.
- System administration is done by the superuser (root): users, mounting, backups, boot (init), security and monitoring.
- Inode = file metadata (owner, type, permissions, times, link count, 13 disk addresses, size); the file name is not in the inode.
- 13 disk addresses = 10 direct + single + double + triple indirect; with 1 KB blocks and 256 pointers per indirect block the maximum size is about 16 GB.
- File system layout: boot block, super block, inode list, data blocks.
- Super block holds file-system size, free blocks and inodes (partial lists) and indexes to the next free ones.

8.5 Multiple Choice Questions (15)

Choose the correct option for each question. The answer key is given in Section 8.6.

Q1. UNIX was originally developed at:

- (a) IBM
- (b) AT&T Bell Laboratories
- (c) Microsoft
- (d) Sun Microsystems

Q2. UNIX was rewritten in which language to make it portable?

- (a) Assembly
- (b) Pascal
- (c) C
- (d) FORTRAN

Q3. The two major subsystems of the UNIX kernel are:

- (a) Shell and utilities
- (b) File subsystem and process control subsystem
- (c) Compiler and linker
- (d) Editor and loader

Q4. Which of the following is part of the process control subsystem?

- (a) Buffer cache
- (b) Block device driver
- (c) Scheduler
- (d) Super block

Q5. The kernel mechanism that regulates data flow between the kernel and disks is the:

- (a) Region table
- (b) Buffer cache
- (c) Process table
- (d) Mount table

Q6. Per-process information needed only while the process is running is kept in the:

- (a) Process table
- (b) u area
- (c) Mount table
- (d) Super block

Q7. The superuser (root) has the user ID:

- (a) 0
- (b) 1
- (c) 10
- (d) 100

Q8. The process with ID 1 in UNIX is:

- (a) swapper
- (b) init
- (c) getty
- (d) login

Q9. Which process state means the process has terminated but its exit status is still stored?

- (a) Asleep
- (b) Pre-empted
- (c) Zombie
- (d) Created

Q10. Which of the following is NOT stored in an inode?

- (a) File size
- (b) Owner ID
- (c) File name
- (d) Number of links

Q11. The table of contents in a classic UNIX inode has how many disk address entries?

- (a) 10
- (b) 12
- (c) 13
- (d) 15

Q12. With 1 KB blocks and 4-byte block numbers, one indirect block holds how many block numbers?

- (a) 128
- (b) 256
- (c) 512
- (d) 1024

Q13. The order of parts in a UNIX file system on disk is:

- (a) Super block, boot block, inode list, data blocks
- (b) Boot block, super block, inode list, data blocks
- (c) Boot block, inode list, super block, data blocks
- (d) Inode list, boot block, super block, data blocks

Q14. Which structure contains the list of free blocks and free inodes of a file system?

- (a) Boot block
- (b) Super block
- (c) Directory
- (d) u area

Q15. A directory entry in classic System V UNIX consists of:

- (a) Inode number and file name
- (b) File size and file name
- (c) Owner and permissions
- (d) Block number and size

8.6 Answer Key

Q. No.	Answer	Short Reason
1	(b)	Developed by Thompson and Ritchie at Bell Labs.
2	(c)	The C rewrite in 1973 made it portable.
3	(b)	File subsystem and process control subsystem.
4	(c)	The scheduler, IPC and memory management belong to process control.
5	(b)	The buffer cache sits between the kernel and block devices.
6	(b)	The u area is the user area of the process.
7	(a)	Root has UID 0.
8	(b)	init is the first user process and ancestor of all others.
9	(c)	A zombie has exited but not been waited for.
10	(c)	File names are stored in directories.
11	(c)	10 direct + 3 indirect = 13.
12	(b)	$1024 / 4 = 256$.
13	(b)	Boot block, super block, inode list, data blocks.
14	(b)	The super block has the partial lists.
15	(a)	A 2-byte inode number and a 14-byte name.

8.7 Broad Questions (10)

1. Describe the history and main features of the UNIX operating system.
2. Explain the system structure of UNIX with a neat layered diagram. What is the role of the shell and the system-call interface?
3. Explain the architecture (block diagram) of the UNIX kernel. Describe the file subsystem and the process control subsystem.
4. Distinguish between user mode and kernel mode. Explain how a system call works and list the categories of system calls.
5. What are the major kernel data structures? Explain the process table, u area, file table and inode table and how they are related.
6. Explain the process states in UNIX with a state transition diagram. Differentiate between a mode switch and a context switch.
7. What is system administration? Explain the role of the superuser and the main tasks of a system administrator. Describe the boot and login sequence.
8. What is an inode? Describe the fields of a disk inode and the extra fields of an in-core inode.
9. Explain the structure of a regular file in UNIX. How do direct, single, double and triple indirect blocks work? Calculate the maximum file size for 1 KB blocks and 4-byte block numbers.
10. Explain the layout of a UNIX file system. Describe the super block and how free inodes and free disk blocks are allocated.