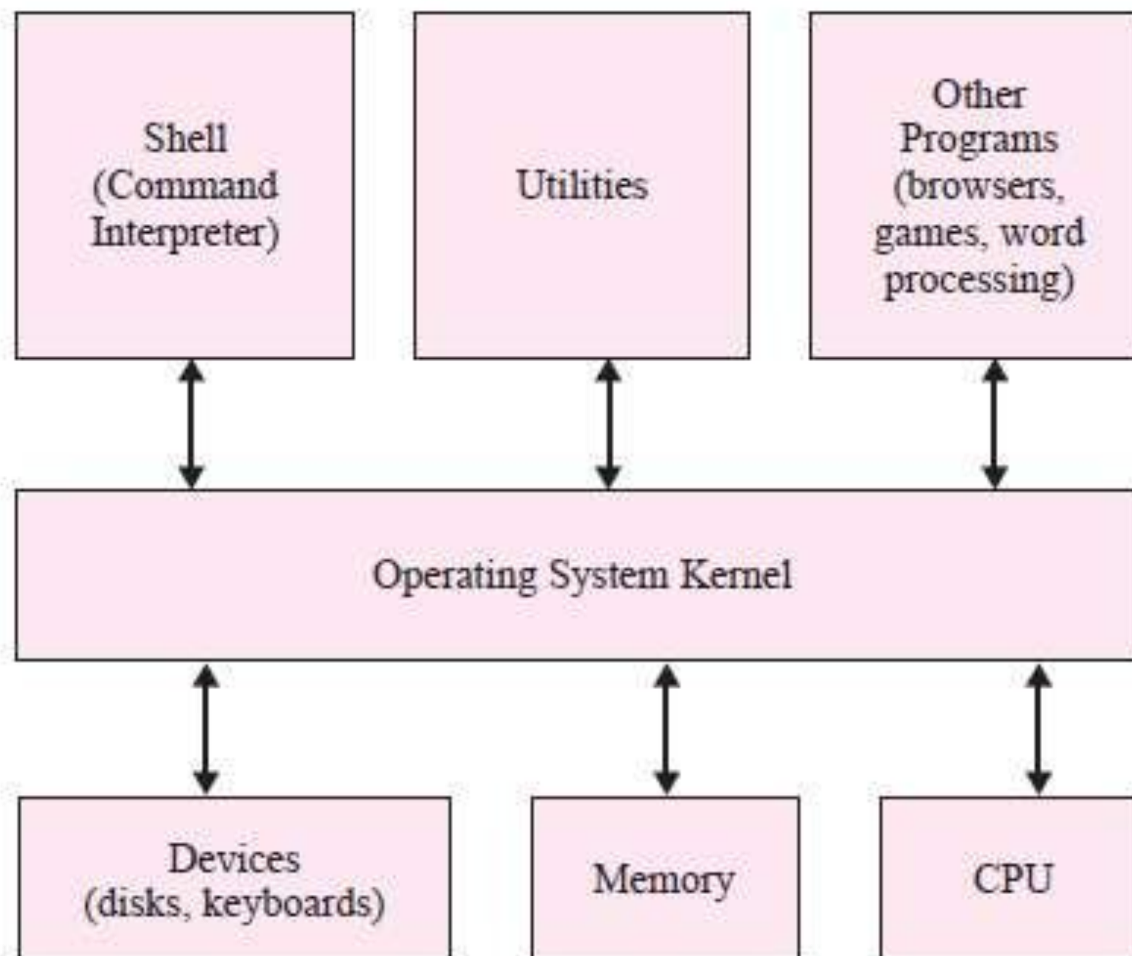
Operating systems

Lecture 2, Michal Vrábel, 15/10/2019

Operating System - definitions

- An OS is a program that **controls the execution of application programs** and acts as an **interface between applications and the computer hardware**.
  - Stalling: Operating systems internals and design principles 5th edition
- An operating system acts as **an intermediary between the user of a computer and the computer hardware**. The purpose of an operating system is to provide an **environment in which a user can execute programs** in a convenient and efficient manner.
  - Silberschatz et al.: Operating System Concepts with Java. 8th ed. 2010
- An operating system is software that **manages the computer hardware**. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system
  - Silberschatz et al.: Operating System Concepts with Java. 8th ed. 2010

FIGURE 1.3
A simplistic view of the OS software in relationship to hardware.



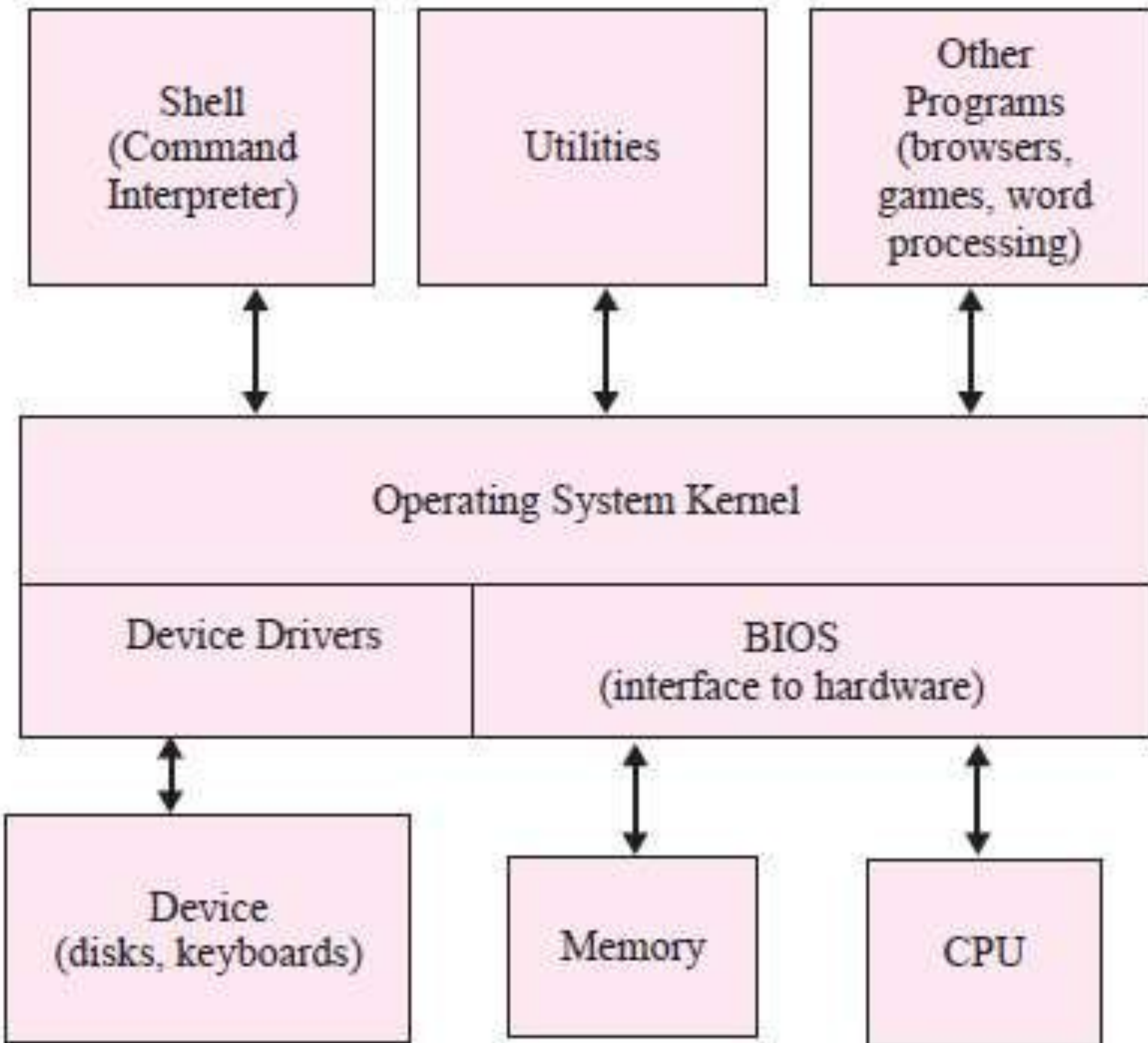
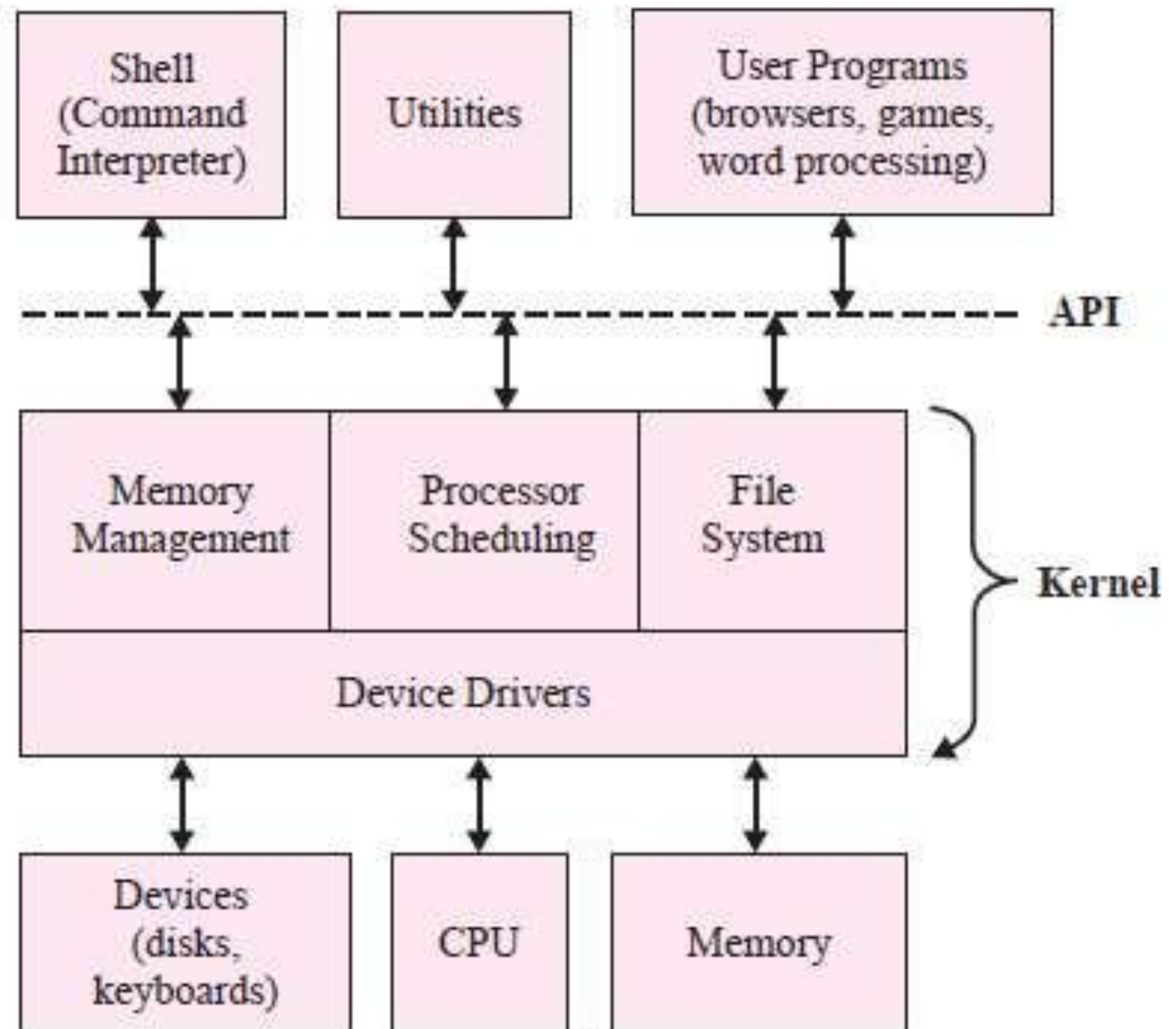


FIGURE 1.5 The PC (small system) model of an OS.

FIGURE 2.4
Layered model of an Operating System.



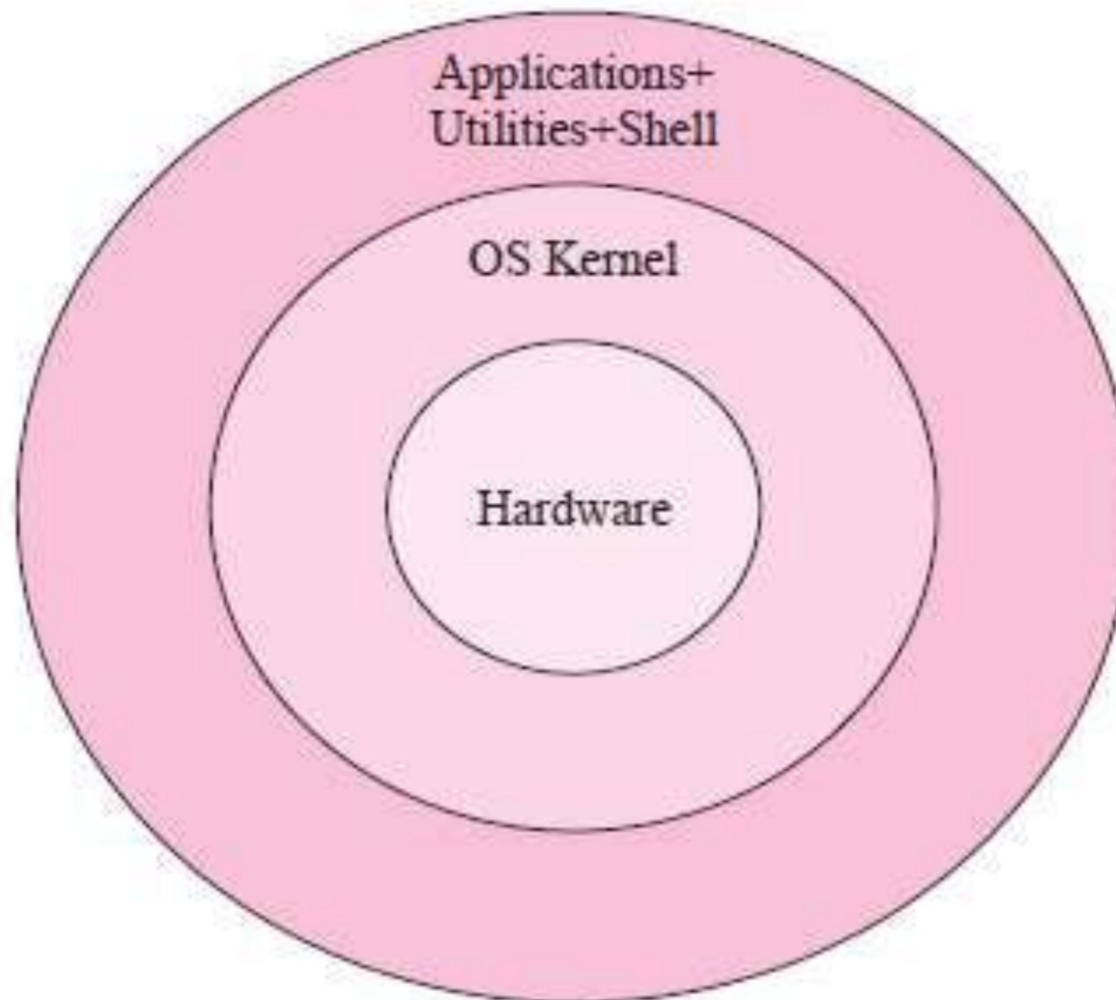


FIGURE 1.4
A layered view of an OS.

Classifications of OS

- Type of processing
  - Batch
  - Interactive
- Number of parallel users
  - Single-user
  - Multi-user
- Response time
  - Guaranteed response time - real-time
  - Not guaranteed
- Use case
  - General / Universal
  - Specialized

System services of an Operating system

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

today →

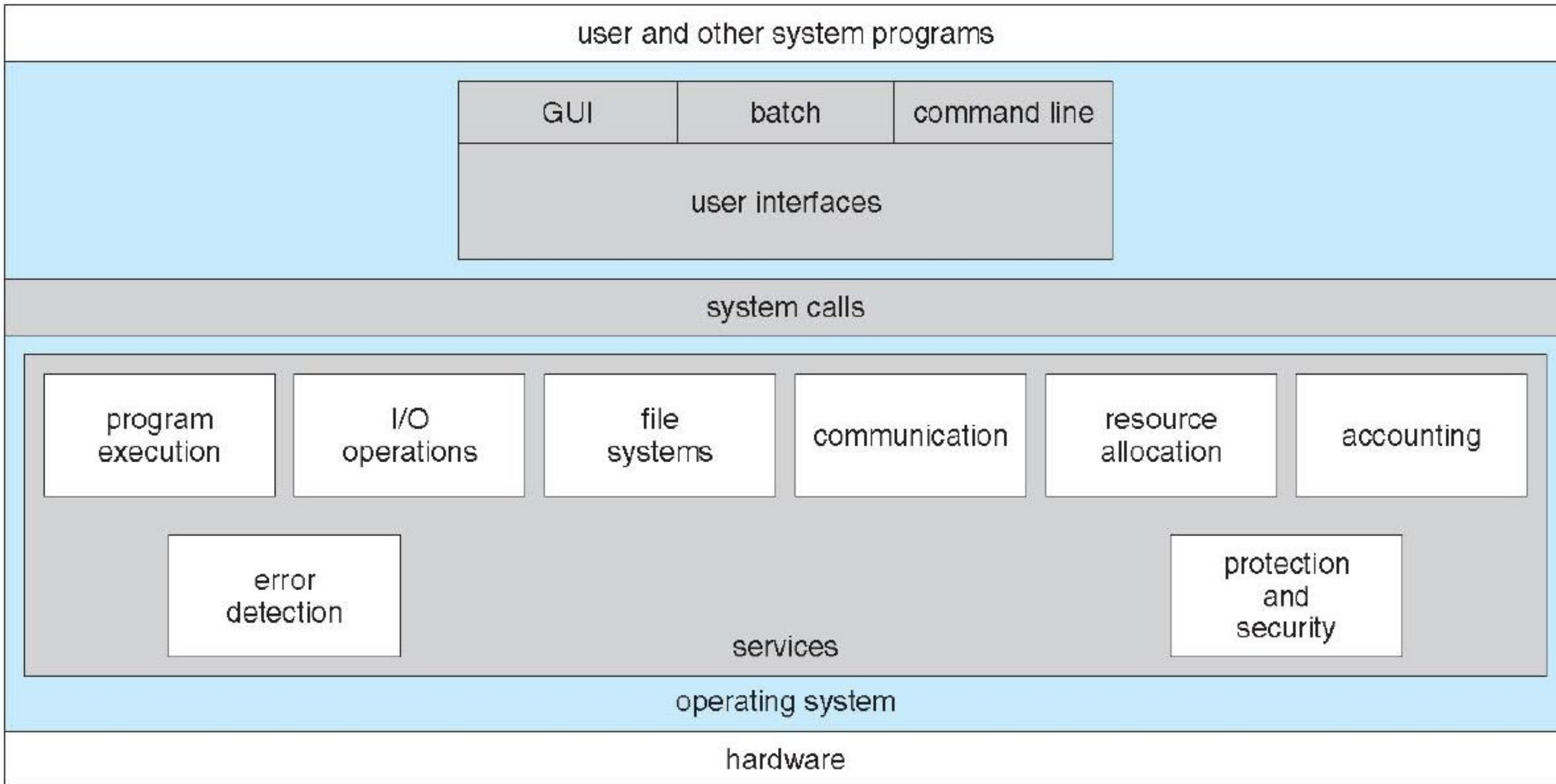
System services of an Operating system

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

System services of an Operating system

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

System services of an Operating system



System Call Implementation

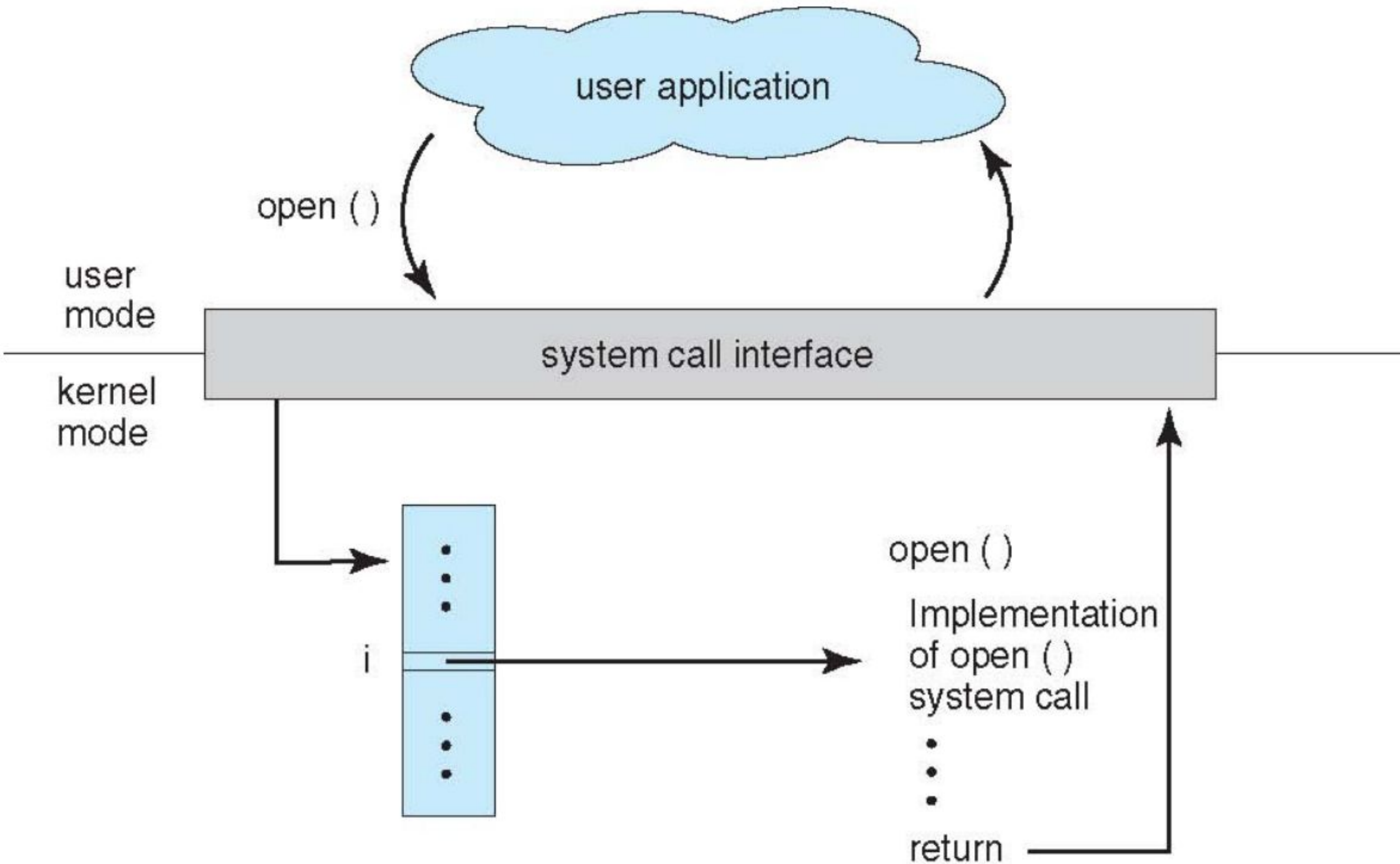
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

  (Note that the system-call names used throughout this text are generic)

System call implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Types of System Calls

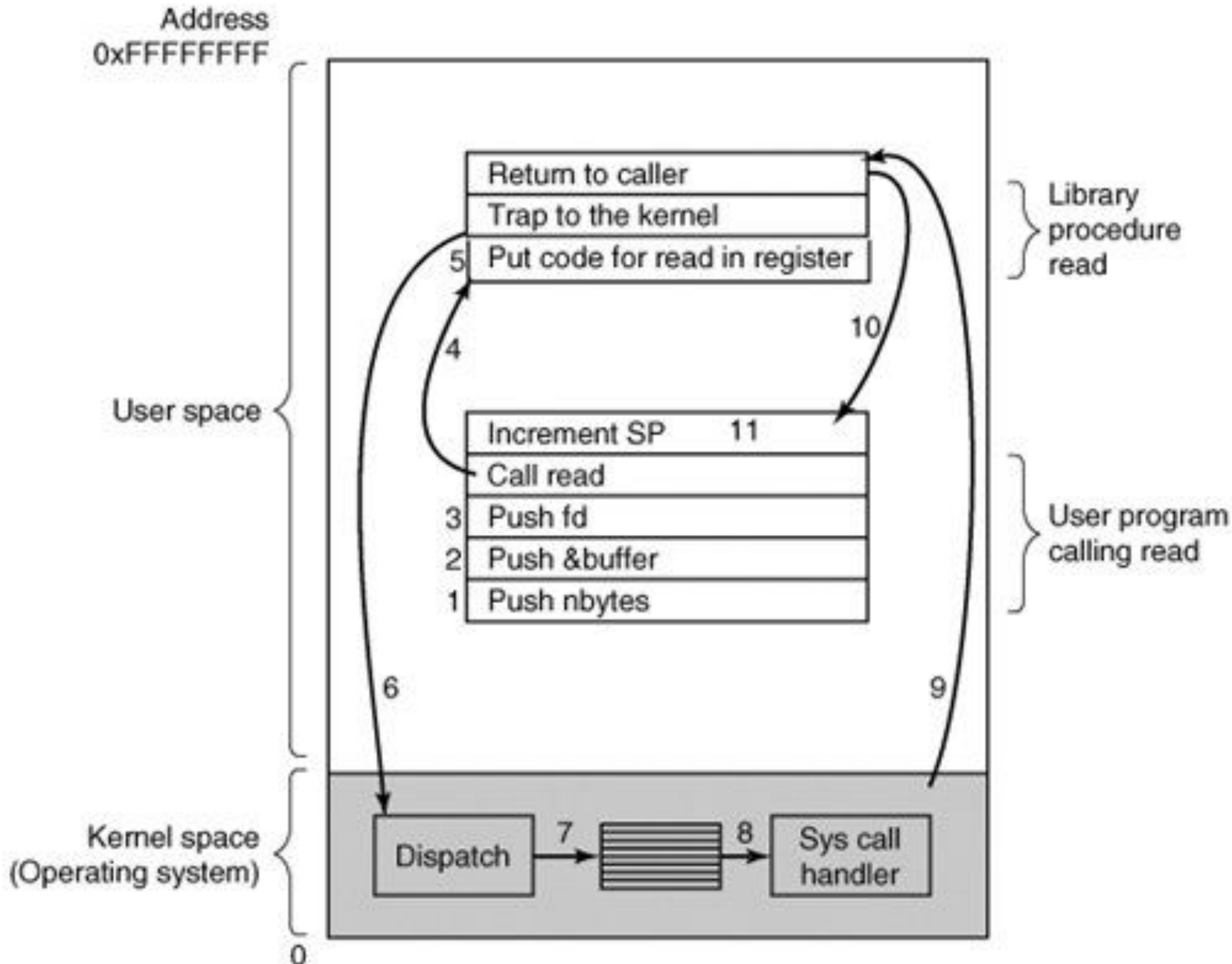
- **Process control**
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach device
- **Information maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **Communications**
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Communication between an application and OS

- The 11 steps in making the system call `read(fd, buffer, nbytes)`



Operating Systems Design and Implementation, Third Edition By Andrew S. Tanenbaum

Communication between an application and OS

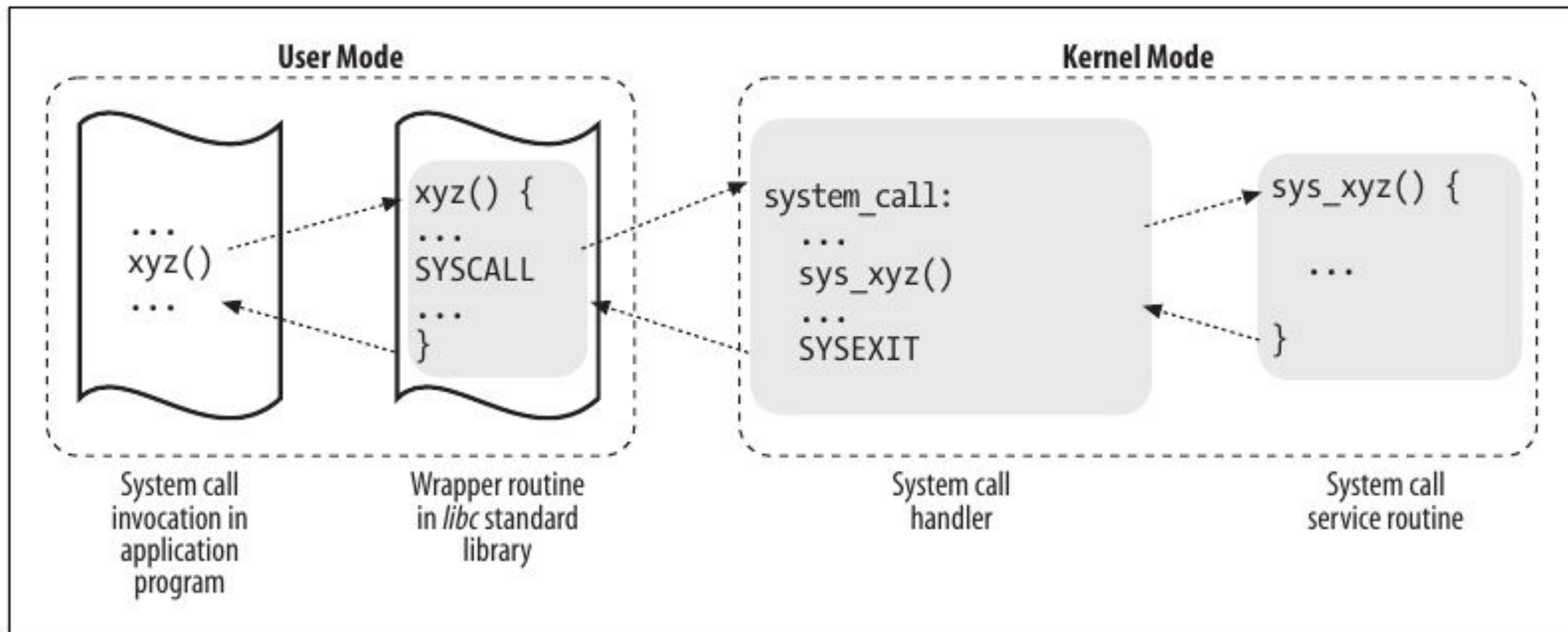


Figure 10-1. Invoking a system call

The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

1) Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).
2) Handles the system call by invoking a corresponding C function called the system call service routine.
3) Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

Understanding the Linux Kernel, Third Edition 3rd Edition by Daniel P. Bovet, Page 400

Issuing a system call

- Native applications* can invoke a system call in two different ways:
  - By executing the int $0x80 assembly language instruction; in older versions of the Linux kernel, this was the only way to switch from User Mode to Kernel Mode.
  - By executing the sysenter assembly language instruction, introduced in the Intel Pentium II microprocessors; this instruction is now supported by the Linux 2.6 kernel.

Understanding the Linux Kernel, Third Edition 3rd Edition by Daniel P. Bovet, Page 400

Issuing a system call

Issuing a System Call via the int $0x80 Instruction

The "traditional" way to invoke a system call makes use of the `int` assembly language instruction, which was discussed in the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4.

The vector 128—in hexadecimal, `0x80`—is associated with the kernel entry point. The `trap_init()` function, invoked during kernel initialization, sets up the Interrupt Descriptor Table entry corresponding to vector 128 as follows:

```
set_system_gate(0x80, &system_call);
```

The call loads the following values into the gate descriptor fields (see the section "Interrupt, Trap, and System Gates" in Chapter 4):

Segment Selector
: The `__KERNEL_CS` Segment Selector of the kernel code segment.

Offset
: The pointer to the `system_call()` system call handler.

Type
: Set to 15. Indicates that the exception is a Trap and that the corresponding handler does not disable maskable interrupts.

DPL (Descriptor Privilege Level)
: Set to 3. This allows processes in User Mode to invoke the exception handler (see the section "Hardware Handling of Interrupts and Exceptions" in Chapter 4).

Therefore, when a User Mode process issues an `int $0x80` instruction, the CPU switches into Kernel Mode and starts executing instructions from the `system_call` address.

Understanding the Linux Kernel, Third Edition 3rd Edition by Daniel P. Bovet, Page 402

Implementations of VFS System Calls

1) shell command that copies the MS-DOS file /floppy/TEST to the Ext2 file /tmp/test.
2) Cp executes

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    len = read(inf, buf, 4096);
    write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

Actually, the code of the real cp program is more complicated, because it must also
check for possible error codes returned by each system call.

Understanding the Linux Kernel, Third Edition 3rd Edition by Daniel P. Bovet, Page 505

Implementations of VFS System Calls

3) The open( ) system call is serviced by the sys_open( ) function,

sys_open( ) function. It performs the following steps:

Understanding the Linux Kernel, Third Edition 3rd Edition by Daniel P. Bovet, Page 507

1. Invokes getname( ) to read the file pathname from the process address space.
2. Invokes get_unused_fd( ) to find an empty slot in current->files->fd. The corresponding index (the new file descriptor) is stored in the fd local variable.
3. Invokes the filp_open() function, passing as parameters the pathname, the access mode flags, and the permission bit mask. This function, in turn, executes the following steps:
   a. Copies the access mode flags into namei_flags, but encodes the access mode flags O_RDONLY, O_WRONLY, and O_RDWR with a special format: the bit at index 0 (lowest-order) of namei_flags is set only if the file access requires read privileges; similarly, the bit at index 1 is set only if the file access requires write privileges. Notice that it is not possible to specify in the open() system call that a file access does not require either read or write privileges; this makes sense, however, in a pathname lookup operation involving symbolic links.
   b. Invokes open_namei( ), passing to it the pathname, the modified access mode flags, and the address of a local nameidata data structure. The function performs the lookup operation in the following manner:
      - If O_CREAT is not set in the access mode flags, starts the lookup operation with the LOOKUP_PARENT flag not set and the LOOKUP_OPEN flag set. Moreover, the LOOKUP_FOLLOW flag is set only if O_NOFOLLOW is cleared, while the LOOKUP_DIRECTORY flag is set only if the O_DIRECTORY flag is set.
      - If O_CREAT is set in the access mode flags, starts the lookup operation with the LOOKUP_PARENT, LOOKUP_OPEN, and LOOKUP_CREATE flags set. Once the path_lookup() function successfully returns, checks whether the requested file already exists. If not, allocates a new disk inode by invoking the create method of the parent inode.

      The open_namei() function also executes several security checks on the file located by the lookup operation. For instance, the function checks whether the inode associated with the dentry object found really exists, whether it is a regular file, and whether the current process is allowed to access it according to the access mode flags. Also, if the file is opened for writing, the function checks that the file is not locked by other processes.
   c. Invokes the dentry_open() function, passing to it the addresses of the dentry object and the mounted filesystem object located by the lookup operation, and the access mode flags. In turn, this function:
      1. Allocates a new file object.
      2. Initializes the f_flags and f_mode fields of the file object according to the access mode flags passed to the open() system call.
      3. Initializes the f_dentry and f_vfsmnt fields of the file object according to the addresses of the dentry object and the mounted filesystem object passed as parameters.
      4. Sets the f_op field to the contents of the i_fop field of the corresponding inode object. This sets up all the methods for future file operations.
      5. Inserts the file object into the list of opened files pointed to by the s_files field of the filesystem's superblock.
      6. If the open method of the file operations is defined, the function invokes it.
      7. Invokes file_ra_state_init() to initialize the read-ahead data structures (see Chapter 16).
      8. If the O_DIRECT flag is set, it checks whether direct I/O operations can be performed on the file (see Chapter 16).
      9. Returns the address of the file object.
   d. Returns the address of the file object.
4. Sets current->files->fd[fd] to the address of the file object returned by dentry_open().
5. Returns fd.

- Syscall table:
  - https://elixir.bootlin.com/linux/v2.6.39/source/arch/x86/kernel/syscall_table_32.S
  - https://elixir.bootlin.com/linux/v4.15/source/include/linux/syscalls.h#L553
- Handler?
  - https://elixir.bootlin.com/linux/v2.6.39/source/fs/open.c#L1050
  - https://elixir.bootlin.com/linux/v4.15/source/fs/open.c#L1117

```
00000000 <__libc_open>:
   0:   83 3d 00 00 00 00 00    cmpl   $0x0,0x0
   7:   75 21                   jne    2a <__libc_open+0x2a>
   9:   53                      push   %ebx
   a:   8b 54 24 10             mov    0x10(%esp),%edx
   e:   8b 4c 24 0c             mov    0xc(%esp),%ecx
  12:   8b 5c 24 08             mov    0x8(%esp),%ebx
  16:   b8 05 00 00 00          mov    $0x5,%eax
  1b:   cd 80                   int    $0x80
  1d:   5b                      pop    %ebx
  1e:   3d 01 f0 ff ff          cmp    $0xfffff001,%eax
  23:   0f 83 fc ff ff ff       jae    25 <__libc_open+0x25>
  29:   c3                      ret
  2a:   e8 fc ff ff ff          call   2b <__libc_open+0x2b>
  2f:   50                      push   %eax
  30:   53                      push   %ebx
  31:   8b 54 24 14             mov    0x14(%esp),%edx
  35:   8b 4c 24 10             mov    0x10(%esp),%ecx
  39:   8b 5c 24 0c             mov    0xc(%esp),%ebx
  3d:   b8 05 00 00 00          mov    $0x5,%eax
  42:   cd 80                   int    $0x80
  44:   5b                      pop    %ebx
  45:   87 04 24                xchg   %eax,(%esp)
  48:   e8 fc ff ff ff          call   49 <__libc_open+0x49>
  4d:   58                      pop    %eax
  4e:   3d 01 f0 ff ff          cmp    $0xfffff001,%eax
  53:   0f 83 fc ff ff ff       jae    55 <__libc_open+0x55>
  59:   c3                      ret
```

File management

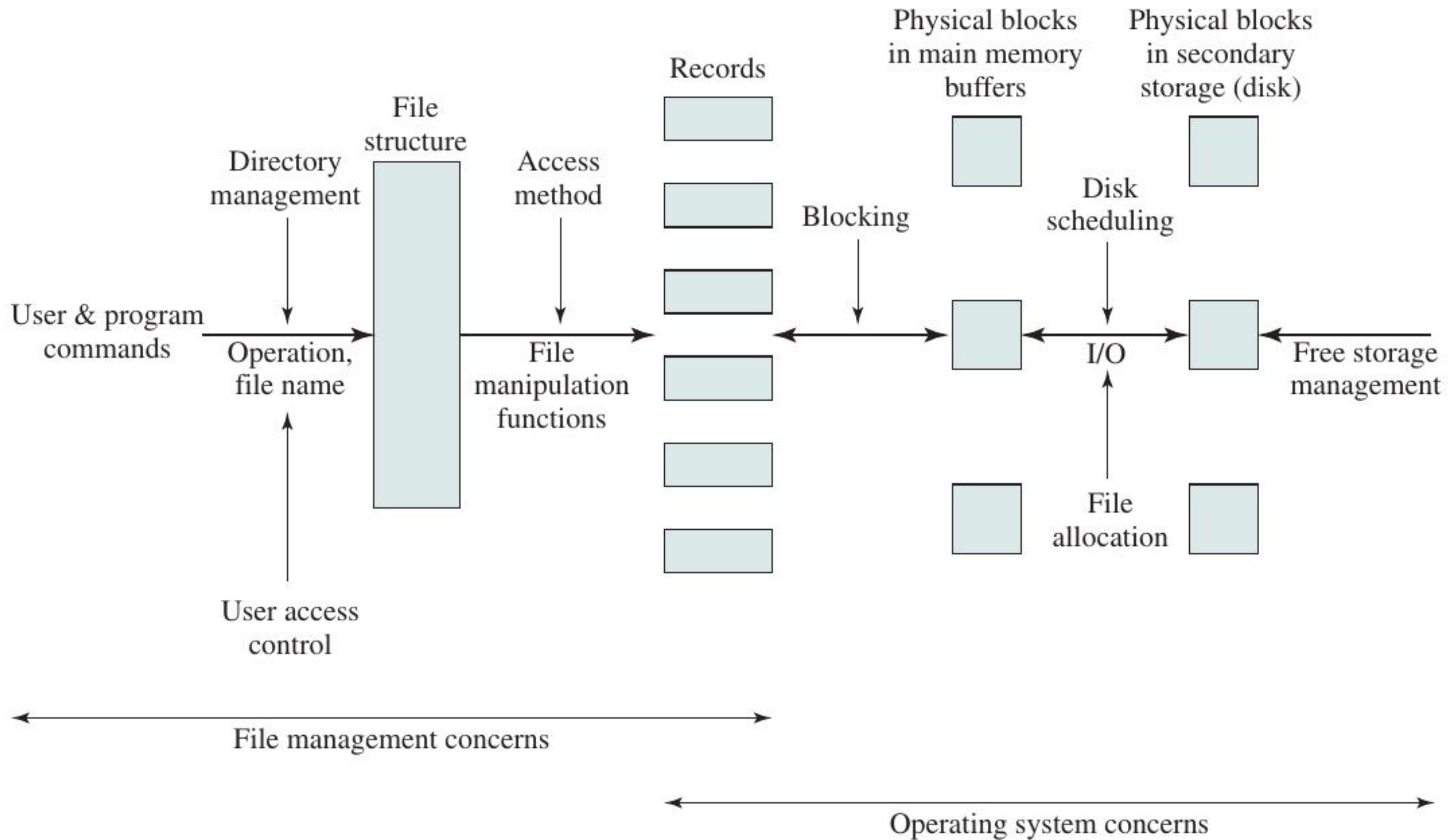


Figure 12.2 **Elements of File Management**

Secondary storage management

- On secondary storage, a file consists of a collection of blocks.
- Several issues are involved in **file allocation**:

1) When a new file is created, is the **maximum space required for the file allocated at once**? -
PREALLOCATION VERSUS DYNAMIC ALLOCATION

2) Space is allocated to a file as one or more contiguous units, which we shall refer to as **portions**. That is, a portion is a **contiguous set of allocated blocks**. The size of a portion can range from a single block to the entire file. What size of portion should be used for file allocation? -
PORTION SIZE

3) What sort of data structure or table is used to **keep track of the portions assigned to a file**? An example of such a structure is a **file allocation table** (FAT), found on DOS and some other systems. -
FILE ALLOCATION METHODS

Secondary storage management - preallocation

- Preallocation policy requires that the **maximum size of a file be declared at the time of the file creation request**.
- **In a number of cases**, such as program compilations, the production of summary data files, or the transfer of a file from another system over a communications network, this value **can be reliably estimated**.
- However, **for many applications, it is difficult** if not impossible to estimate reliably the maximum potential size of the file. In those cases, users and application programmers would tend to overestimate.
- There are advantages to the use of **dynamic allocation**, which allocates space to a file in portions as needed

Secondary storage management - portion size

- Trade-off between efficiency from the point of view of a **single file versus overall system efficiency**
- Two major alternatives:
  - **Variable, large contiguous portions**:
    - This will provide better performance. The variable size avoids waste, and the file allocation tables are small.
    - However, space is hard to reuse.
  - **Blocks**
    - Small fixed portions provide greater flexibility.
    - They may require large tables or complex structures for their allocation.
    - Contiguity has been abandoned as a primary goal; blocks are allocated as needed.

- With variable-size portions, we need to be concerned with the fragmentation of free space.
- **First fit:** Choose the first unused contiguous group of blocks of sufficient size from a free block list.
- **Best fit:** Choose the smallest unused group that is of sufficient size.
- **Nearest fit:** Choose the unused group of sufficient size that is closest to the previous allocation for the file to increase locality.

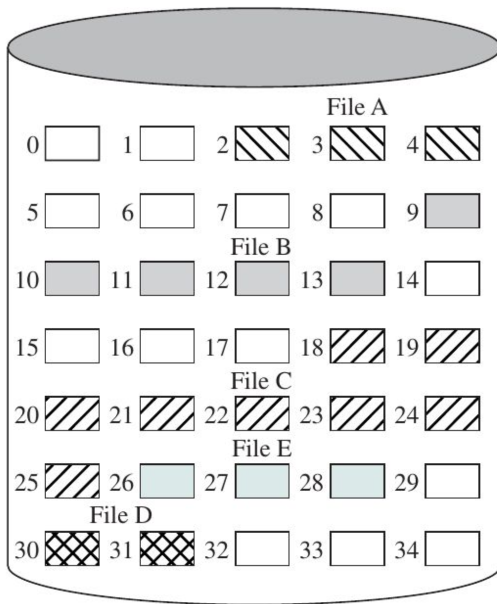
Secondary storage management - File allocation methods

- Three methods are in common use: contiguous, chained, and indexed
- **contiguous allocation**
  - a single contiguous set of blocks is allocated to a file at the time of file creation
  - it will be necessary to perform a compaction algorithm
  - External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length
- **chained allocation**
  - block contains a pointer to the next block in the chain
  - no external fragmentation

Table 12.3 File Allocation Methods

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or Variable Size Portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion Size	Large	Small	Small	Medium
Allocation Frequency	Once	Low to high	High	Low
Time to Allocate	Medium	Long	Short	Medium
File Allocation Table Size	One entry	One entry	Large	Medium

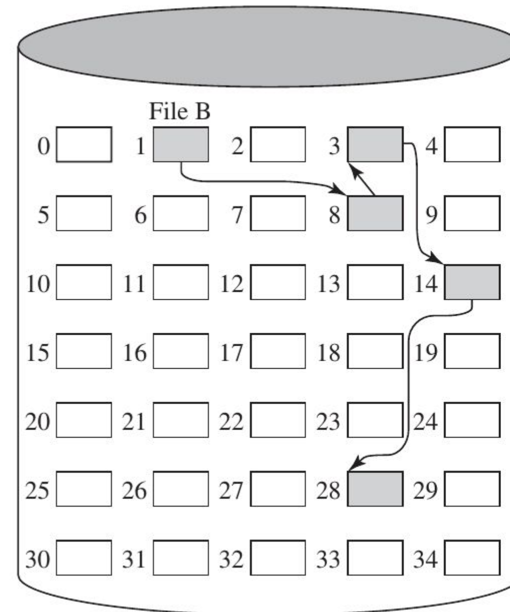
Secondary storage management - File allocation methods (contiguous, chained)



File allocation table

File name	Start block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.9 **Contiguous File Allocation**

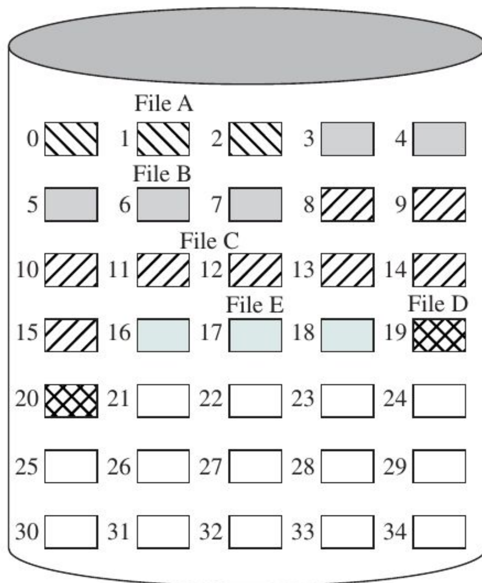


File allocation table

File name	Start block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Figure 12.11 **Chained Allocation**

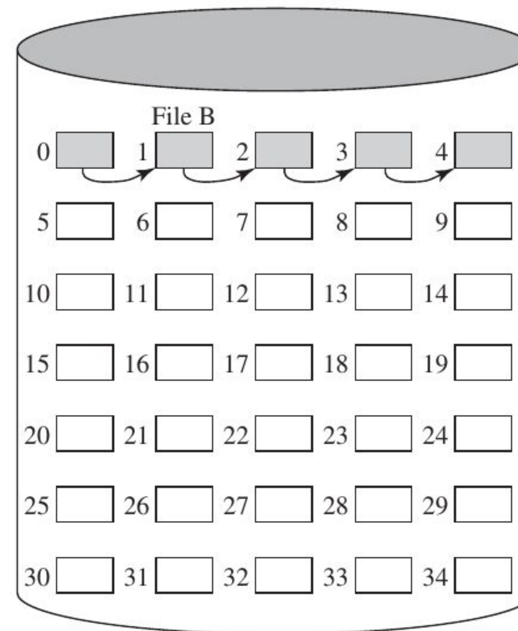
Secondary storage management - File allocation methods (contiguous, chained)



File allocation table

File name	Start block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.10 **Contiguous File Allocation (After Compaction)**



File allocation table

File name	Start block	Length
•••	•••	•••
File B	0	5
•••	•••	•••

Figure 12.12 **Chained Allocation (After Consolidation)**

Secondary storage management - File allocation methods (indexed)

- Indexed allocation addresses many of the problems of contiguous and chained allocation.
- In this case, the file **allocation table contains a separate one-level index for each file**;
- the index has one entry for each portion allocated to the file.
- Typically, **the file indexes are not physically stored as part of the file allocation table**.
- Rather, the file **index for a file is kept in a separate block**, and the entry for the file in the file allocation table points to that block.
- File consolidation may be done from time to time

Secondary storage management - File allocation methods (indexed)

- Allocation may be on the basis of either fixed-size blocks (Figure 12.13) or variable-size portions (Figure 12.14).

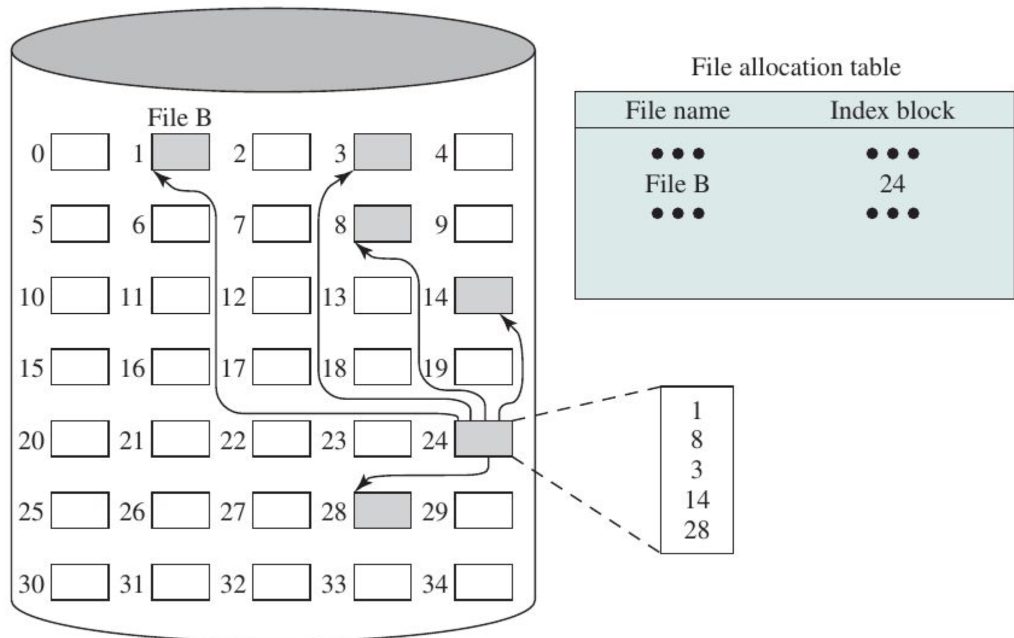


Figure 12.13 Indexed Allocation with Block Portions

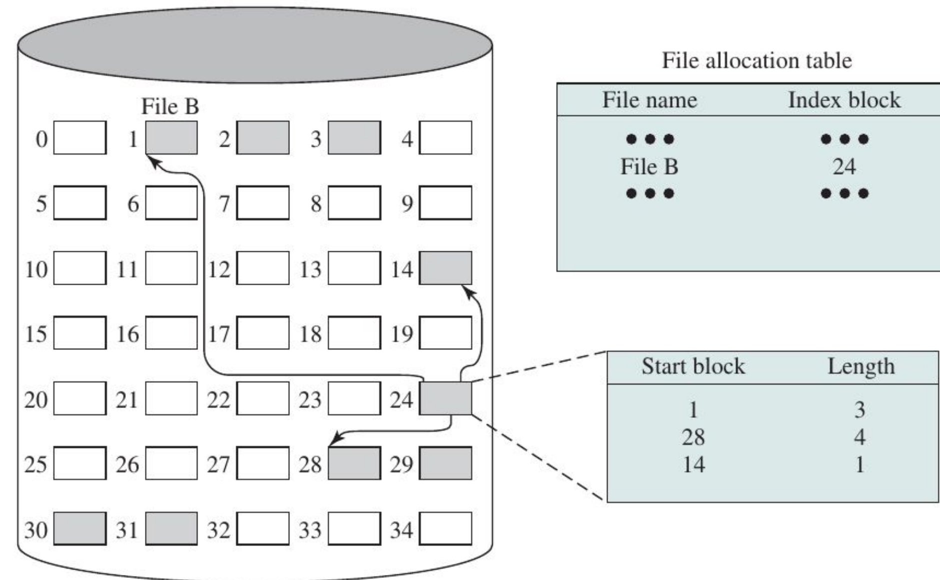


Figure 12.14 Indexed Allocation with Variable-Length Portions

Secondary storage management - Free Space Management

- To perform any of the file allocation techniques described previously, **it is necessary to know what blocks on the disk are available**.
- Thus we need a **disk allocation table in addition to a file allocation table**. We discuss here a number of techniques that have been implemented.
- **BIT TABLES**
  - This method uses a vector containing one bit for each block on the disk.
  - Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use.
- **CHAINED FREE PORTIONS**
  - The free portions may be chained together by using a pointer and length value in each free portion.
- **INDEXING**
  - The indexing approach treats free space as a file and uses an index table as described under file allocation. For efficiency, the index should be on the basis of variable-size portions rather than blocks. Thus, there is one entry in the table for every free portion on the disk.
- **FREE BLOCK LIST**
  - In this method, each block is assigned a number sequentially and the list of the numbers of all free blocks is maintained in a reserved portion of the disk

Volumes

- In essence a volume is a **logical disk**.
- A collection of **addressable sectors in secondary memory** that an OS or application can use for data storage.
- The sectors in a volume **need not be consecutive on a physical storage device**; instead, they need only appear that way to the OS or application.
- A volume may be the result of assembling and **merging smaller volumes**.
- In the simplest case, a single disk equals one volume. Frequently, a disk is divided into partitions, with each partition functioning as a separate volume.
- It is **also common to treat multiple disks as a single volume** or partitions on multiple disks as a single volume.

Reliability

- The following steps could be performed when a file allocation is requested:
  - **Lock the disk allocation table on disk.** This prevents another user from causing alterations to the table until this allocation is completed.
  - **Search the disk allocation table for available space.** This assumes that a copy of the disk allocation table is always kept in main memory. If not, it must first be read in.
  - **Allocate space, update the disk allocation table, and update the disk.** Updating the disk involves writing the disk allocation table back onto disk. For chained disk allocation, it also involves updating some pointers on disk.
  - **Update the file allocation table and update the disk.**
  - **Unlock the disk allocation table.**

File System Security - Access Rights

- Access rights that can be assigned to a particular user for a particular file.
- None: The user may not even learn of the existence of the file, much less access it. To enforce this restriction, the user would not be allowed to read the user directory that includes this file.
- **Knowledge:** The user can determine that the file exists and who its owner is.
  - The user is then able to petition the owner for additional access rights.
- **Execution:** The user can load and execute a program but cannot copy it.
  - Proprietary programs are often made accessible with this restriction.
- **Reading:** The user can read the file for any purpose, including copying and execution.
  - Some systems are able to enforce a distinction between viewing and copying. In the former case, the contents of the file can be displayed to the user, but the user has no means for making a copy.
- **Appending:** The user can add data to the file, often only at the end, but cannot modify or delete any of the file's contents.
  - This right is useful in collecting data from a number of sources.
- **Updating:** The user can modify, delete, and add to the file's data. This normally includes writing the file initially, rewriting it completely or in part, and removing all or a portion of the data. Some systems distinguish among different degrees of updating.
- **Changing protection**: The user can change the access rights granted to other users. Typically, this right is held only by the owner of the file. In some systems, the owner can extend this right to others.
- **Deletion:** The user can delete the file from the file system.

File System Security - Access Rights

- Access can be provided to different classes of users:
- **Specific user**: Individual users who are designated by user ID
- **User groups**: A set of users who are not individually defined. The system must have some way of keeping track of the membership of user groups.
- **All:** All users who have access to this system. These are public files.

Directory

- To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory:
- **Search:** When a user or application references a file, the directory must be searched to find the entry corresponding to that file.
- **Create file:** When a new file is created, an entry must be added to the directory.
- **Delete file:** When a file is deleted, an entry must be removed from the directory.
- **List directory:** All or a portion of the directory may be requested. Generally,this request is made by a user and results in a listing of all files owned by that user, plus some of the attributes of each file (e.g., type, access control information, usage information).
- **Update directory:** Because some file attributes are stored in the directory, a change in one of these attributes requires a change in the corresponding directory entry.

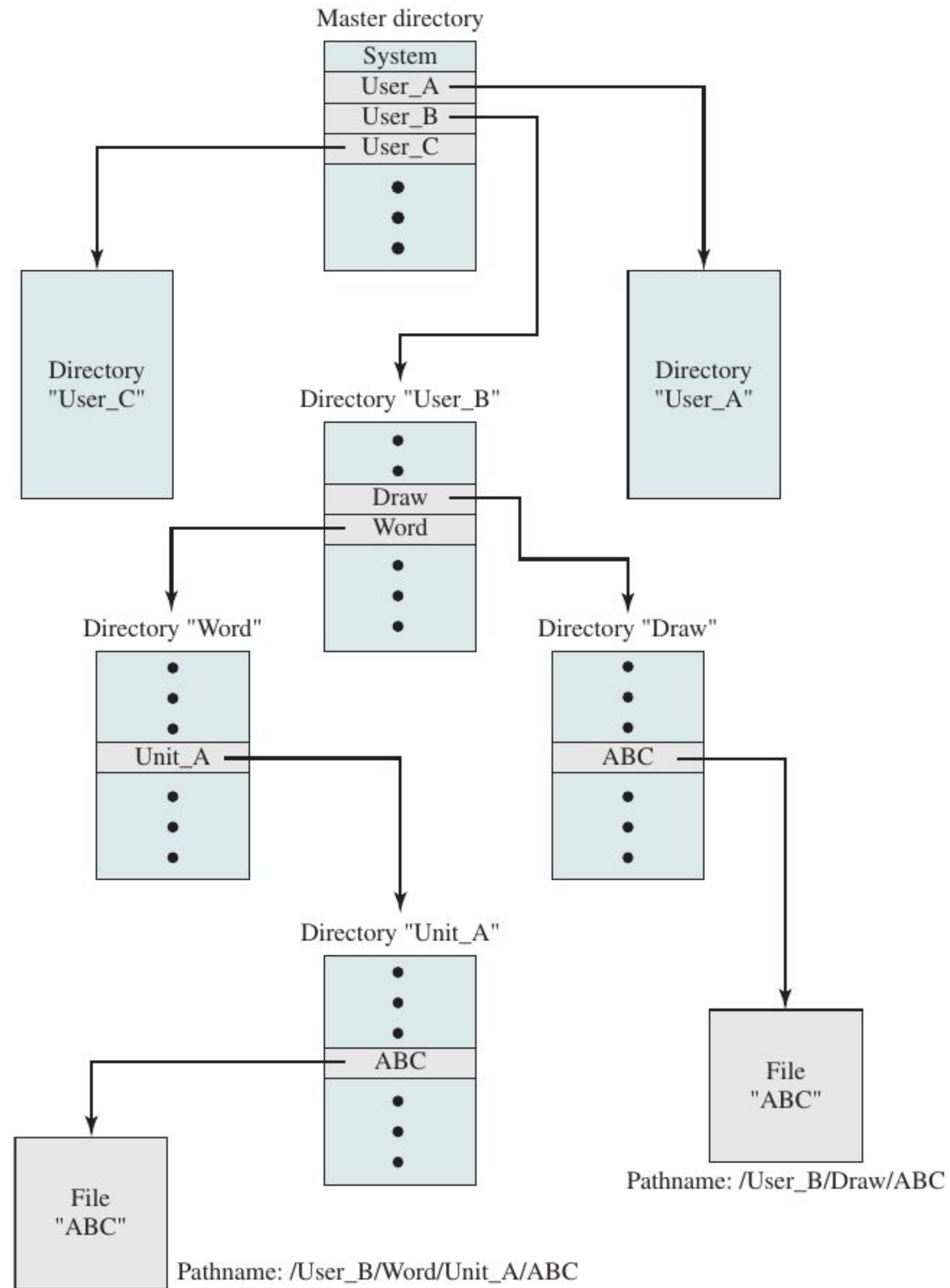


Figure 12.7 **Example of Tree-Structured Directory**

Table 12.2 Information Elements of a File Directory

	Basic Information
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
	Address Information
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
	Access Control Information
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, and transmitting over a network
	Usage Information
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

Unix File Management

- Six types of files are distinguished:
  - **Regular, or ordinary**: Contains arbitrary data in zero or more data blocks.
    - Regular files contain information entered in them by a user, an application program, or a system utility program. The file system does not impose any internal structure to a regular file but treats it as a stream of bytes.
  - **Directory:** Contains a list of file names plus pointers to associated inodes (index nodes), described later. Directories are hierarchically organized.
  - **Special:** Contains no data but provides a mechanism to map physical devices to file names.
    - The file names are used to access peripheral devices, such as terminals and printers. Each I/O device is associated with a special file
  - **Named pipes:** As discussed in Section 6.7, a pipe is an interprocess communications facility.
    - A pipe file buffers data received in its input so that a process that reads from the pipe's output receives the data on a first-in-first-out basis.
  - **Links:** In essence, a link is an alternative file name for an existing file.
  - **Symbolic links:** This is a data file that contains the name of the file it is linked to.

Unix File Management: Inodes

- All types of UNIX files are administered by the OS by means of **inodes**.
- An inode (index node) is a **control structure** that contains the key information needed by the operating system for a particular file.
- **Several file names may be associated with a single inode**, but an active inode is associated with exactly one file, and each file is controlled by exactly one inode.

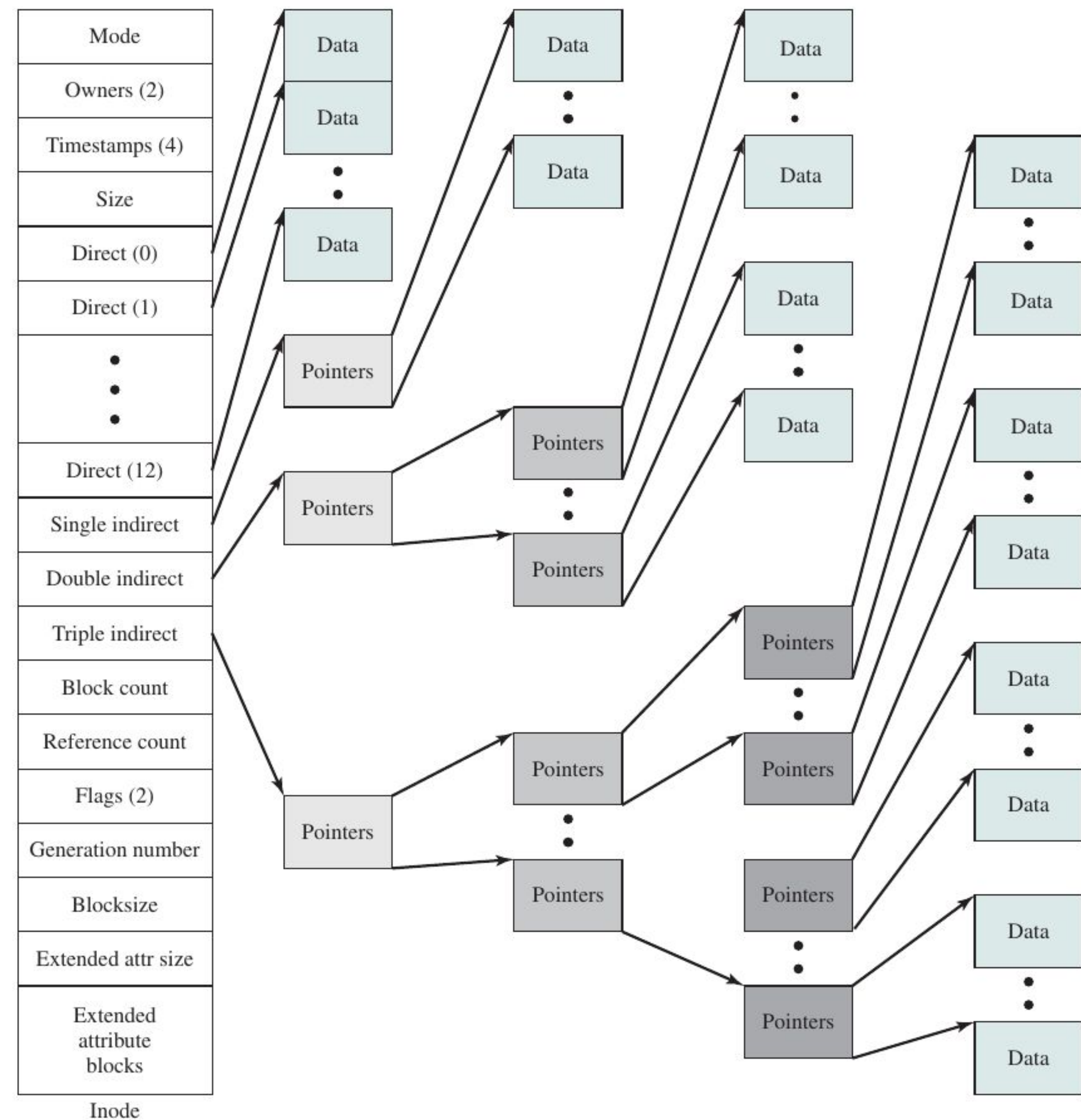


Figure 12.16 Structure of FreeBSD Inode and File

Example: Inode and dentry management in GlusterFS

- https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Developer-guide/datastructure-inode/
- Inode table in glusterfs mainly contains a hash table for maintaining inodes.
- In general a file/directory is considered to be existing if there is a corresponding inode present in the inode table.
- If some fop is happening on the inode, then the inode will be present in the **active inodes list** maintained by the inode table. unlink/rmdir/forget).

- Apart from the hash table, inode table also maintains 3 important list of inodes

1) **Active list:** Active list contains all the active inodes (i.e inodes which are currently part of some fop). Active inodes are those inodes whose refcount is greater than zero.
2) **Lru list:** Least recently used inodes list. A limit can be set for the size of the lru list. For bricks it is 16384 and for clients it is infinity.
3) **Purge list:** List of all the inodes which have to be purged (i.e inodes which have to be deleted from the inode table due to unlink/rmdir/forget)

Unix File Management: File Allocation

- File allocation is done on a block basis. Allocation is dynamic, as needed, rather than using preallocation.
- Hence, the blocks of a file on disk are not necessarily contiguous.
- An indexed method is used to keep track of each file, with part of the index stored in the inode for the file.
- In all UNIX implementations, the inode includes a number of direct pointers and three indirect pointers (single, double, triple).

Unix File Management: Directories

- Directories are structured in a hierarchical tree.
- Directory is simply a file that contains a list of file names plus pointers to associated inodes.
- Each directory entry (dentry) contains a name for the associated file or subdirectory plus an integer called the i-number (index number).
- When the file or directory is accessed, its i-number is used as an index into the inode table.

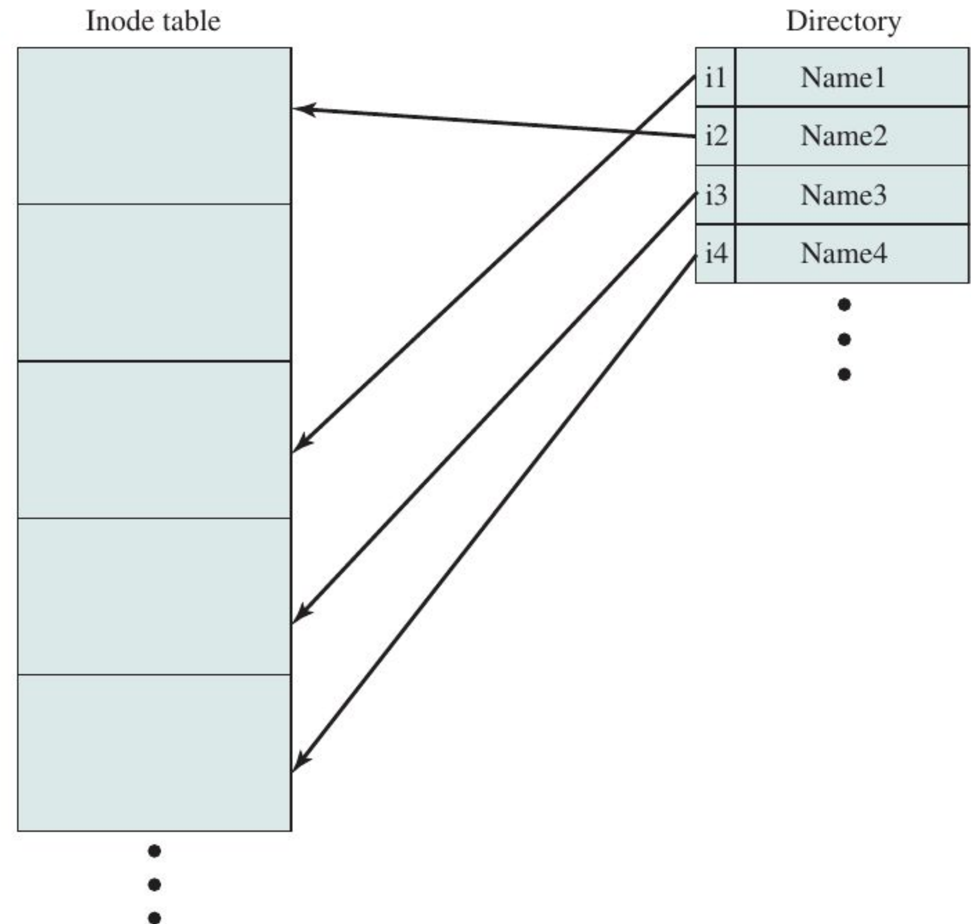


Figure 12.17 UNIX Directories and Inodes

Unix File Management: Volume Structure

- Boot block: Contains code required to boot the operating system
- Superblock: Contains attributes and information about the file system, such as partition size, and inode table size
- Inode table: The collection of inodes for each file
- Data blocks: Storage space available for data files and subdirectories

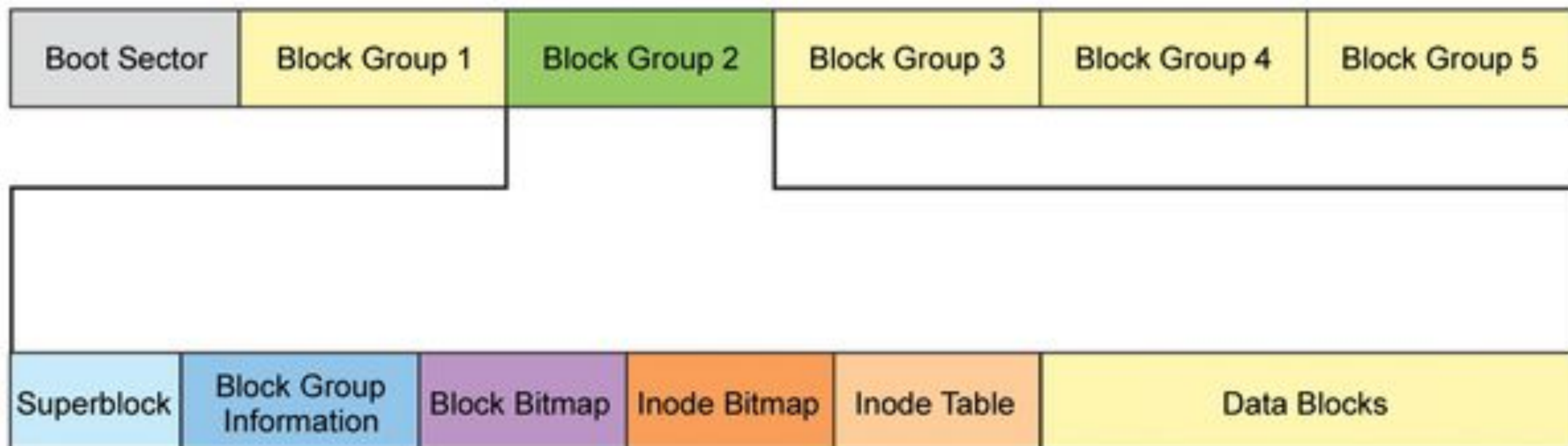
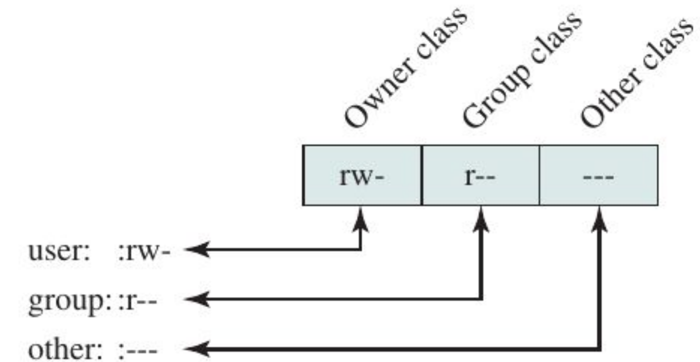


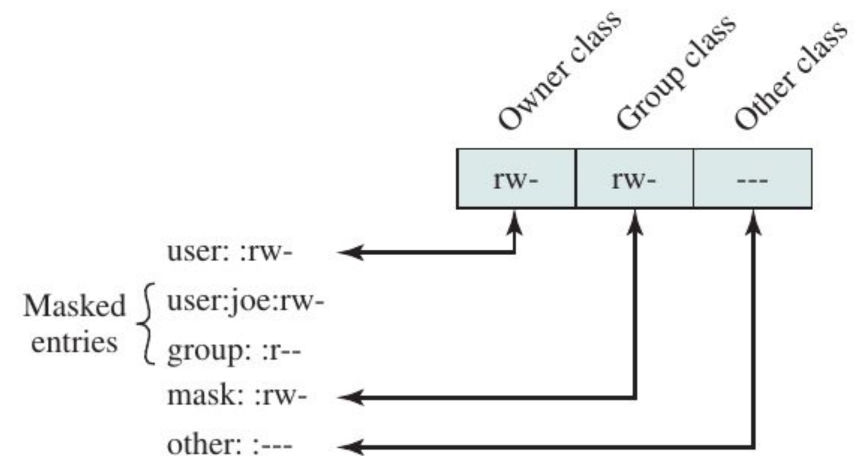
Image: http://www.linux-magazine.com/Issues/2013/156/The-ext-Filesystem

Unix File Management: File Access Control

- Associated with each file is a set of 12 protection bits.
  - 9: rwxrwxrwx
  - 3: SetUID, SetGID, Sticky bit
- Access Control Lists in UNIX
  - Many modern UNIX and UNIX-based operating systems support access control lists, including FreeBSD, OpenBSD, Linux, and Solaris.
  - setfacl command



(a) Traditional UNIX approach (minimal access control list)



(b) Extended access control list

Figure 12.18 UNIX File Access Control

Unix File Management: File Access Control

- **SetUID, SetGID:**
  - When a user (with execute privileges for this file) executes the file, **the system temporarily allocates the rights of the user's ID of the file creator, or the file's group,** respectively
- **"Sticky" bit**
  - When set on a file, this originally indicated that the system should retain the file contents in memory following execution. This is no longer used.
  - When applied to a directory, though, it specifies that only the owner of any file in the directory can rename, move, or delete that file.
    - Useful for managing files in shared temporary directories

Linux virtual file system

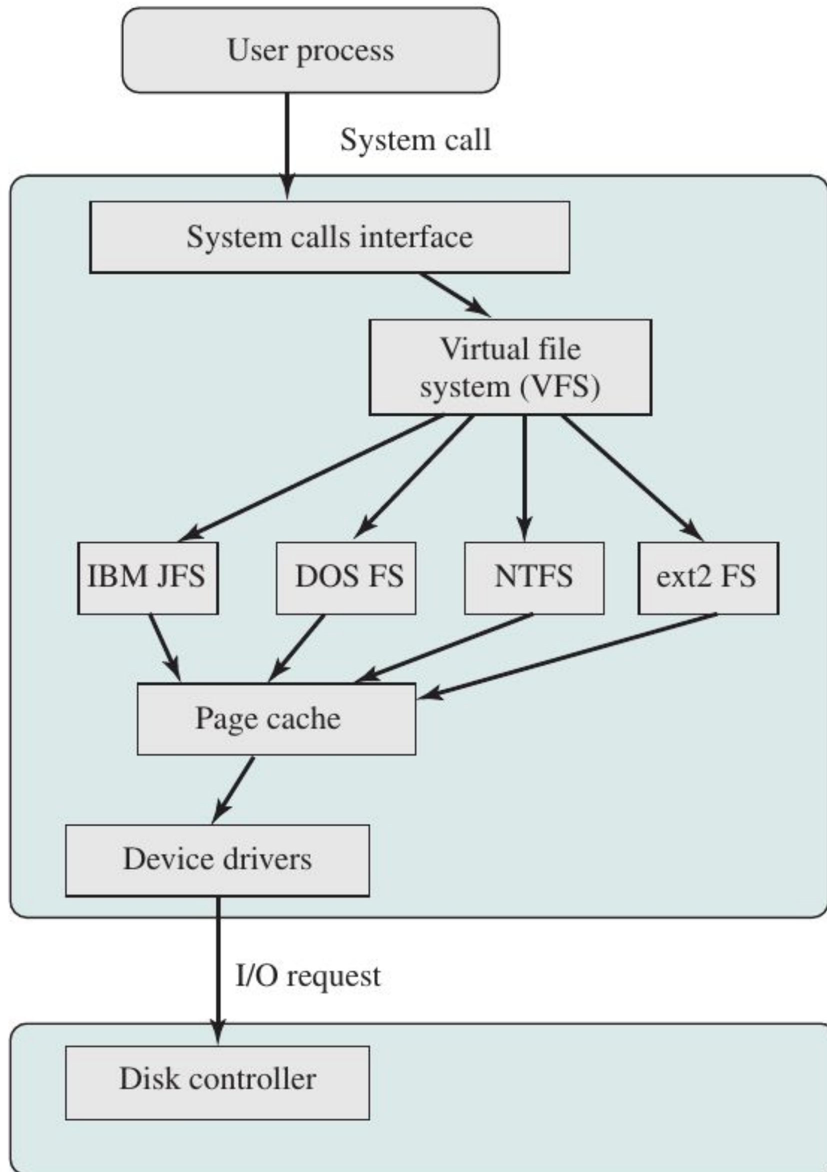


Figure 12.19 Linux Virtual File System Context

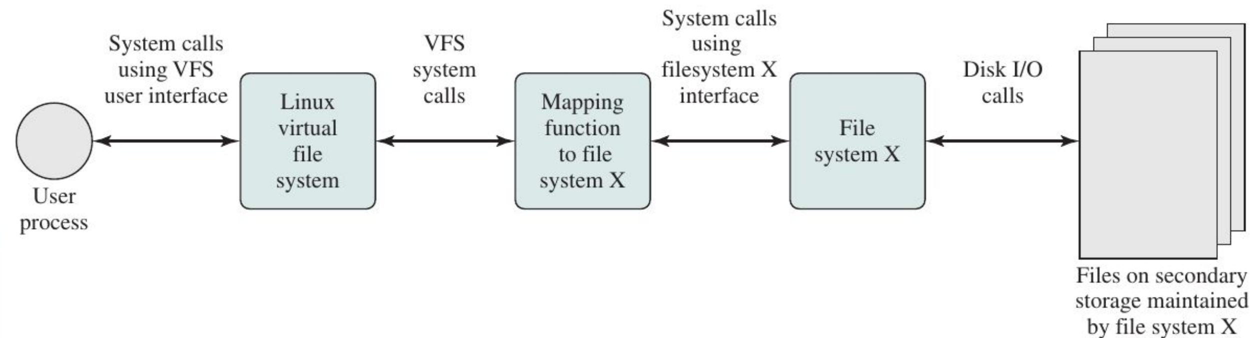


Figure 12.20 Linux Virtual File System Concept

- **Superblock object:** Represents a specific mounted file system
- **Inode object:** Represents a specific file
- **Dentry object:** Represents a specific directory entry
- **File object:** Represents an open file associated with a process