Operating Systems

Unit 4

* File-System Interface

* File → a logical storage unit

(i) text file : a sequence of characters organized into lines and pages

(ii) source file : a sequence of functions, organized as declarations and executable statements

(iii) executable file:- a series of code sections that the loader can bring into the memory and execute.

* File Attributes

→ name
→ identifier
→ type
→ location
→ size
→ protection
→ time, data & user identification

* File Operations

(i) creation
(ii) write file
(iii) read file
(iv) repositioning aka seek
(v) deleting a file
(vi) truncating a file

* Opening Files

open file table : has info about the files that are open

no need to search

file pointer points to last read / location

File open count : when files are closed, decrement counter

allows removal data from the open-file table

when the last process closes it

* Open File locking

→ to prevent other processes from gaining access to it

Shared lock - reader lock - several processes can acquire the lock concurrently

Exclusive lock - only one process can acquire lock at a time,

writer's lock

Mandatory lock - once an exclusive lock is acquired, OS prevents any other process from accessing it

Advisory locks - upto developers to find lock status & decide what to do

File Access Methods

(i) Sequential access : info is processed in order, one record after the other

read_next()
write next()

(ii) Direct access

→ file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order

→ based on disk model, since disks allow random access to any file block

(iii) With an index

→ maintain index for each file (like a book)

→ contains pointers to different blocks

* Directory and Disk Structure

→ disk divided into partitions

→ partitions can be raw - w/o a file system or formatted - w/ a file system

→ an entity w/ a file system called a volume

→ each volume has the device directory and table of contents

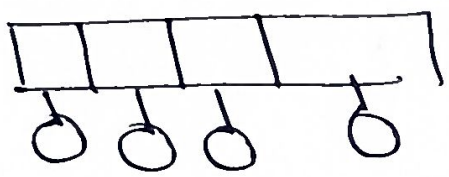
→

* Directory operations

(i) search
(ii) create
(iii) read
(iv) delete
(v) list
(vi) rename
(vii) traverse

* Logical Structures of Directories

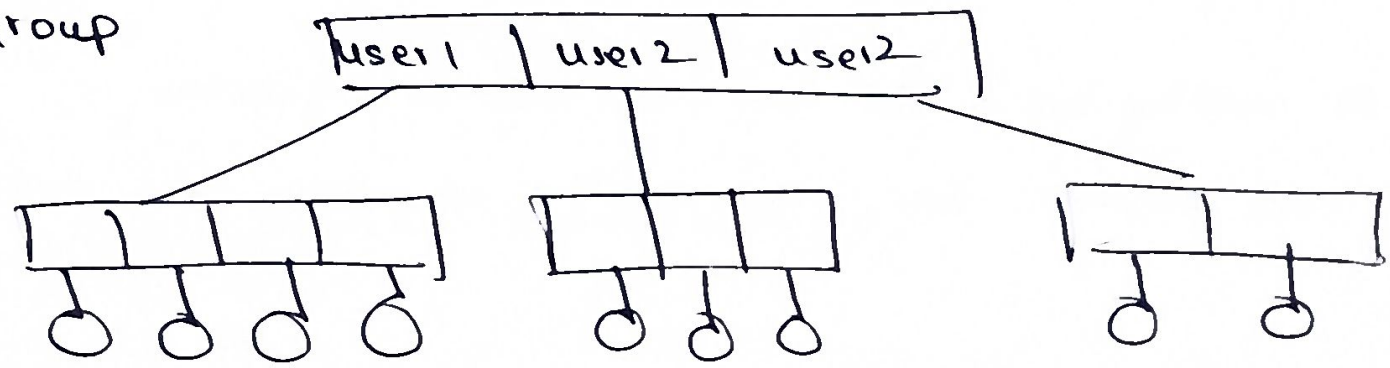
(1) Single - Level



→ single directory for all users
→ naming problem
→ grouping problem
→ difficult to search

(2) Two-level Directory

→ separate directory for each user.
→ each has their own UFD- User File Directory
→ search for UFD in Master File Directory - MFD
→ accessed w/ path name
→ can have same file name for diff. users
→ easy to search
→ cannot group



(3) Tree - Structured Directory

→ users can create their own subdirectories & organize their files accordingly.

→ one bit defines the entry:

file = 0

subdirectory = 1

→ use file path name

absolute - from root

relative - from cwd

→ efficient searching

→ grouping capacity

(4) Acyclic Graph Directories

→ allows for the sharing of subdirectories

→ changes made by 1 person visible to other immediately

→ implemented using links

↳ a pointer to another file/subdirectory

resolve the link → use path name to locate real file

Hard links vs. Soft links

↓ ↓

direct pointer to original inode

a file that has the information to point to another file/inode.

(5) General Graph Directory

→ may have cycles

→ would lead to infinite loops

How to guarantee no cycles?

→ allow links to only files not subdirectories

→ garbage collection - determine when the last reference has been deleted and the disk space can be reallocated

involves traversing entire file system, marking everything that can be accessed

→ use a cycle detection algorithm

*File System Mounting

→ a FS must be mounted before it is accessible to the processes on the system

→ need name of device & mount point

* Protection

→ access control lists

→ specifies user names & types of access allowed

→ when a user requests access, OS checks

3 classifications : (i) owner - who made file
(ii) group - users sharing file
(iii) universe - all other users

* File Sharing

(i) Multiple Users

implement sharing w/ classification, owner, user & group

user ids
group ids

(ii) Remote File Systems

→ use ftp (for anonymous & auth access)
→ use a distributed file system
→ via the WW

client-server model

client seeks files from server
NFS, CFS

failure modes

→ corruption of dir. structures - metadata
→ RFS have even more failure modes - network failure
- server failure
→ maintain state information
→ stateless DFS - assumes that a client request wouldn't have happened unless the file was already open.
→ easy but insecure.

* File-System Structure

→ File systems stored on disks because:

(i) disks can be rewritten in place

(ii) disk can directly access any block of info it contains - can access file either sequentially or randomly

→ I/O transfers between memory & disk happen in blocks

→ A basic file system issues generic read & write cmds

→ FCB - contains information about the file, including ownership, permissions & location of file contents

→ use a layered structure to avoid duplicacy

Layers (i) basic file system - read, write operations

(ii) device driver - manage I/O devices

(iii) file organization module - understand files, logical addres & physical blocks

* File-System Implementation

(i) boot control block - has info needed to boot an OS

(ii) volume control block - has info about the no. of blocks in the partition, size of blocks

(iii) directory structure

(iv) FCB per file - file details

* Partitions and Mounting

→ a disk can be sliced into multiple partitions

→ raw / cooked partitions

↓

has no filesystem

→ root partition - contains the OS, mounted at boot time.

→ boot loader - loads kernel & starts executing it

* Virtual File System

→ provides an object oriented way of implementing file systems

→ separates file-system generic operations from their implementation

→ provides a mechanism to uniquely represent a file throughout a network - Vnode

→ In Linux, the 4 main object types for VFS are:

(i) inode

(ii) file object

(iii) superblock object

(iv) dentry object

* Directory Implementation

(i) Linear list - time consuming, simple

(ii) Hash Table - linear list w/ hash data structure
- decreases search time, collisions

* Allocation methods

(i) Contiguous Allocation - each file occupies contiguous blocks

- simple
- need to find space
- need to know filesize
- external fragmentation
- compaction

(ii) Extent Based Systems

- modified contiguous allocation scheme
- allocates in contiguous blocks of disks
- a file has 1 or more extents

(iii) Linked Allocation - each file a linked list of blocks

- file ends at NULL pointer
- no compaction, external frag.
- not reliable
- takes time to locate block

(iv) FAT variation - beginning of volume has table indexed by block no

(v) Indexed Allocation - each file has its own index blocks

- random access
- need index table
- overhead of index block

(vi) Combined Scheme

→ say there are 15 ptrs

→ first 12 - direct blocks

→ next 3 = indirect blocks

(i) single indirect has addresses of blocks w/ data

(ii) double indirect - addresses of blocks that contain addresses of blocks

(iii) triple indirect block

* Free Space Management

→ keep track of free disk space

→ maintain free-space list

(1) Bit Vector

→ each block is 0 or 1

1 ⇒ free

0 = allocated

→ simple to implement

→ easy to find first n consecutive free blocks

(2) Linked List

→ maintain a linked list of free blocks

→ cannot get continuous space easily

→ no waste of space

→ no need to traverse entire list

③ Grouping

→ modify linked list to store address of the next n-1 free blocks

④ Counting

→ keep address of first free block & count of following free blocks

→ free space list has entries containing addresses and counts

⑤ Space Maps

→ used in Oracle's ZFS filesystem

→ create metaslabs to divide space on disk into manageable size

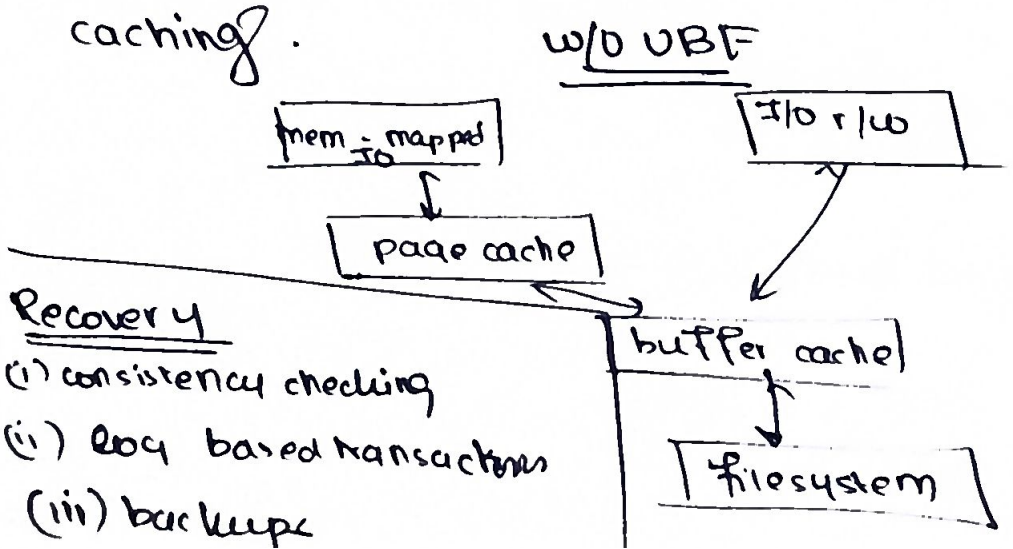
→ each metaslab has an associated space map.

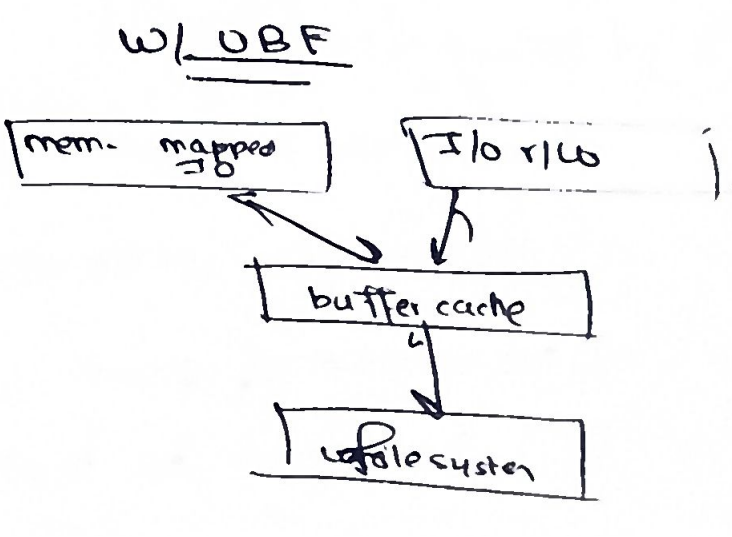
→ use the counting algo to store info about free blocks

→ use a log

* I/O w/ a Unified Buffer Cache

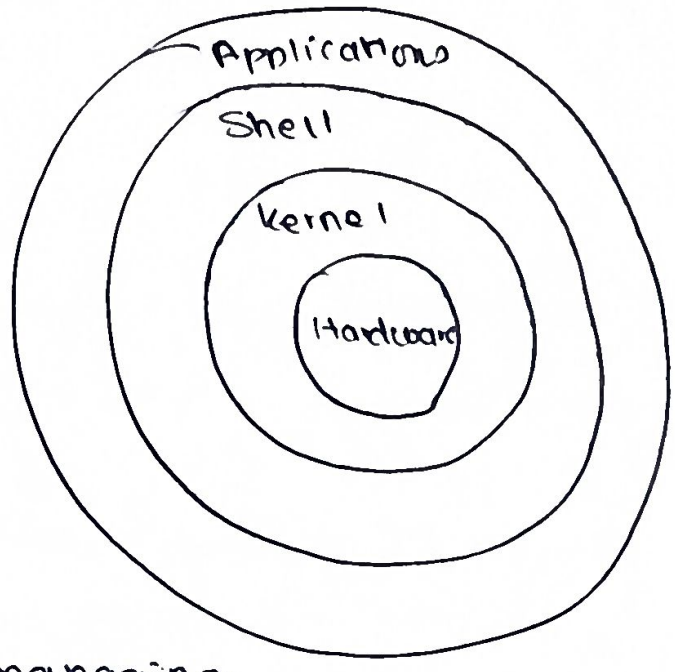
→ a UBF uses the same page cache to cache both memory mapped pages and ordinary file system I/O to avoid double caching.





Unit-5

* Linux Architecture



(i) multiuser

(ii) preemptively multitasking

(iii) adheres to traditional UNIX semantics

(1)

Kernel → responsible for maintaining / managing system resources & providing services to applications

→ monolithic kernel - all core os functionality is contained in a single executable file

→ no unnecessary context switches

→ highly modular and configurable

→ all kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer. (kernel mode)

→ user mode - has access to only a controlled subset of system's resources

Kernel Modules

module mgmt - load modules into memory comm. w/ rest of kernel

module loader & unloader - user mode utilities

driver - reg system - allows modules to tell kernel that a new driver is available