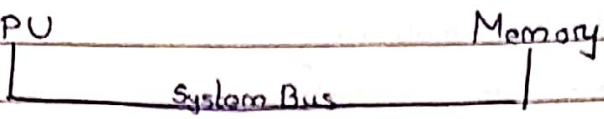
29/3

# I/O SYSTEM




- Network card, Graphics card, keyboard, mouse, disk, etc
  - All are directly connected to motherboard

Types of buses :- Memory bus, PCI, USB, etc

- PCIe is a high speed bus for communication for network card, graphic card

- I/O devices → Block :- Can always access for data. (Can store 'state')
  eg - Disk
  → Stream ('Character devices')
  eg - Mouse, keyboard, network card

→ ~~File System~~

→ Interfacing

- Every device has a 'Status' register
  - Disks store 'busy', 'waiting', etc
  - Network cards can store flags for incoming packet events, etc
- Command Register and Data Register.

(Duh)

Page No.
Date

- OS communicates through device drivers
  - Converts OS code into command and data bits
  - Device driver is kernelspace code
    - Can be present permanently in kernel code, or can be added to kernel code as 'kernel modules' during runtime.

- Reading from registers of I/O devices :-
  - Explicit I/O device
    - Needs separate instructions to be accessed
  - and Memory-mapped I/O device
    Device registers are mapped to RAM addresses. OS can access them the same way it accesses RAM.
    - Easier development ☺
    Certain portion of RAM is reserved for I/O devices

- Process issues write() or read()
  - CPU switches to kernel mode to execute
    - No point in returning back to user code till writing or reading is completed.
  - 'Polling based' design
    - Do not context switch out. Wait till device status is busy
    - Acceptable if device is fast.
    - Cannot poll for devices like network cards (bursty stream of data).
  - 'I

P.T.O.

- 'Interrupt based' design*
  - Device needs to support raising interrupts
  - When interrupt occurs, OS raises a trap, switches to kernel code to handle this interrupt (based on address in IDT
  - For devices that have very high traffic (servers, for example), polling is preferred so as to avoid too many nested interrupts.

- Handling data :-
  - Historically - OS copies read data wherever required after interrupt is raised
    - Wasteful of cycles :(

- DMA - 'Directly Mapped Access' **Direct Memory Access**
  - The device directly writes to a predecided memory location before raising an interrupt. OS can take it from there.
  - Writing -
    OS ~~gives~~ copies data into predecided buffer in memory. The device, through DMA, copies it from there into its own command/data registers.

→ File system
- Provides abstraction so that programmer need not worry about actual blocks of data in disk.

- int fd = open("foo", ...)
  Can create a new file foo, or open an existing file foo
- read() and write() go on sequentially reading/writing through the file.
- Fsync() will write data back to memory

Page No.
Date

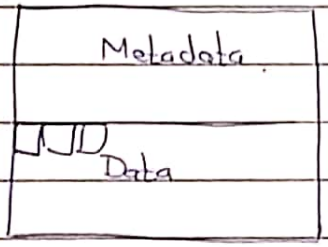
- File descriptor for a file in a program is actually a pointer to file system data
  * FD array also has pointers to info about pipes, sockets, etc.

```
read(fd, [*buffer_name], size)
write(",  "  ")
lseek(fd,                    // Go to particular address so as to be able to read/write there
```

- OS, file systems, do not have to worry about device-specific details. That is done by device drivers

3/4

# I File System




Data has 512 B sized blocks

- For every file in file system, we use a data structure called inode (index node) in metadata
  - Contains pointers to the blocks of data in that file
  - Contains size, last modified, etc., permissions, etc
- Many inodes come together in a data structure called
- inode number is unique for every file in the system.
- Every inode has a limit to how many direct pointers to blocks it can contain.
  - Maximum file size that can have = (512B)(No. of pointers) all direct pointers in one inode

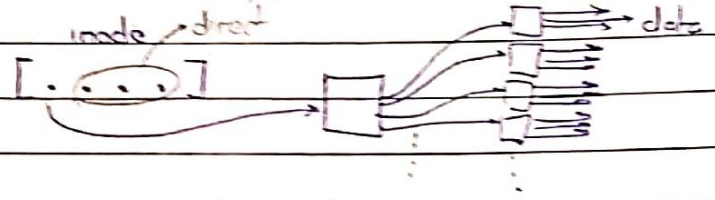
→ File Allocation Table : Alternative to inode-system
- Contains one entry per data block
- Every entry has pointer to next block of the file (NULL if it is last)
- Must remember head pointer (to first block)

| Page No. | | | |
|---|---|---|---|
| Date | | | |

• How inode handles this :- 'Indirect blocks'

- inode contains pointer to a block.
This block contains 128 pointers to other actual data blocks (pointer size ~ 4B)

- Linux also supports double indirect blocks




- If there is an indirect block, other pointers in inode are still direct blocks

→ Directories

- Contains mapping between file name and inode number

- Directory itself is like a file and has an inode, which stores pointers to the data block containing above mapping

eg /foo/bar/ a.txt
b.txt
c.txt

To find inode number of a,
go to inode of 'root' directory
get inode number of 'foo' from there.
& get 'bar'
get 'a.txt'

Page No.
Date

- Directories can also be accessed by create/open/read/close
* inode numbers are globally unique to a file, not just within a directory.

→ Disk contains both data and metadata (both for files and directories)
- To know which blocks of data are free and busy :-
'Bitmap'.
- This bitmap is also stored on disk

Alternate Free list :- Every free block points to next free block.
Superblock points to the first.

→ Superblock
- First block of disk
- Contains info about entire layout of disk
- Which blocks are for bitmap, which are for inode storage, etc.
- Total no. of blocks in memory.

→ Linking of files (via ln command)
- a.txt and b.txt have the exact same contents and must be updated simultaneously.
(~~one~~ they are the same file, but present in different directories)

'Hard Link'

- In the mapping, just use the same inode number in directories of both files.

- Every inode contains a 'link number', ~~or~~ tracking how many places this file is being accessed from.

'Link count' ≡ stored for inode on disk

Page No.
Date

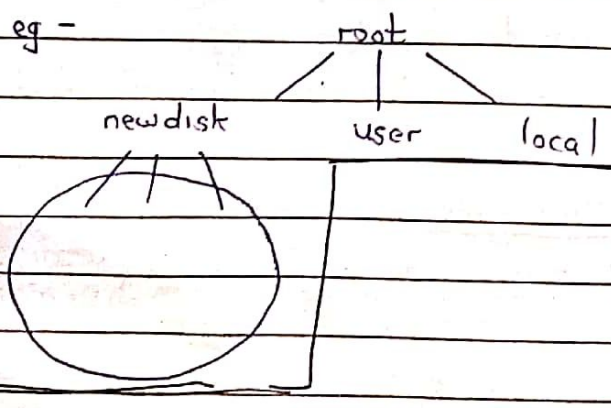
- Deleting one of the linked files does not delete the data. Data is still accessible from other paths.
  'Unlinking'
- OS will delete file if all links are removed.

• Soft Link – (ln -s file1 file2)
Not sharing the same inode number, but sharing the same file name.
eg – When x.txt is soft-linked to a.txt, it will be treated as a file named a.txt in a.txt's directory. It will, like hard link, access inode of original a.txt. Unlike hard link, if a.txt is deleted, x.txt won't work any more and data is actually lost.
- Soft linking does not increment link count.

→ Mounting

Every file system has a 'root' directory. If we want to use another file system (eg – another disk), the root of new file system is listed as a new directory within any directory in original file system.

eg –



Page No.
Date

## → In-memory Data Structures

- Some disk data structures are stored in memory for ease of use.

### ① Open File Table (Global OFT)

- Lists all open files (duh)
- Contains inode info of those open files.
- Also has entries for pipes, sockets.

### ② Per-process File Descriptor Array (Per-process OFT)

- Every element in fd array has a pointer to corresponding open file in OFT
  - Hence, all processes that have same file open will share its inode ~~free~~ data.

→ `fd = open("/foo/bar/a.txt")` - To make inode available in memory.

① Traverse inodes and obtain inode number of a.txt.
- If file does not exist (and if we want to create it), add an inode # entry to mapping of 'bar' directory (Data block for newly created file is allotted only when it is written to)
  - Might need to expand bar inode data blocks.

② Bring info like inode #, file size, etc to OFT.

③ Add pointer to FDA of process, and return the index in FDA as fd.

∴ - Multiple disk ~~calls~~, accesses.

→ Offset :-

- OFT ~~als~~ entry also contains offset for every open file (at what point in the file ~~are~~ we are currently reading from)

- Even if same file is opened by different processes, two separate OFT entries must be made because offsets would be different

* When we want to read a particular, we need to find out the corresponding data block from inode.
Fetch the entire block into RAM in a space called 'Disk buffer Cache'
The required data from this cache is given to user and then 'read' returns
  - The fetched block remains in DBC until it is kicked out

• Varying amount of free space in memory is allocated for use as DBC
• Applications can choose to not use DBC and do direct I/O with disk instead.

→ write (fd, buf)
Read the data block. Make changes. Write back
If modified data does not fit in that data block, ~~and~~ allocate a new data block

Writing at an offset ?

• Writing to a new ~~&~~ file
  - Allocate a data block and update its bitmap
  - Add data block to inode of file.
  - Write data (first back into DBC)

- No need to read before writing to a new data block

Page No.
Date

- If more than one process are writing to the same file, the DBC ensures that this writing is done serially to avoid conflicts.
  - Final block reflects changes made by all

- Types of caches
  1. Write-through — Write back to memory on every modification.
  2. Write-back
     - ∴ Bad if a power failure occurs.

  * fsync() or fflush() system calls can be used to write dirty blocks back to disk

→ Forking

- Child inherits FDA
  (Child's FDA is exact copy of parent's FDA)
  ∴ ~~Every~~ Child's and parent's fd point to same OFT entry (same offset)
  * Pipes between two processes also work by sharing same OFT entry. (one entry for read end, one for write end)
    pipe(fd[2])
    We have two R and W ends in both processes.
    One process closes read end, other closes write end. so that there is no discrepancy caused by sharing offsets.
    (There is no real file, it's just a shared memory in memory image)

Page No.
Date

- The OFT entry also tracks 'count' of no. of FDA elements pointing to that entry.
  OFT entry can be deleted only after count drops to zero.

- In most cases, the parent or the child will close their fd to prevent same offset discrepancy.

- dup() :- For sharing same OFT entry.
  fd1 = dup(fd)

eg Initially fda[0] → STD_OUTPUT
fda[3] → a.txt

close(0)
dup(3)
// Now fda[3] is the new STD_OUTPUT

- init process creates STD_OUTPUT, STD_INPUT, STD_ERROR fd's.
  Every descendent inherits all three fd's, all three of which point to same OFT entry.

- Multiple OFT entries for the same file created by different processes will point to same inode
  - ∴ Changes in file made by one process will be reflected for the other process
  - The different processes cannot see each other's offsets.

Page No.
Date

- Every inode also maintains 'reference count' which tracks how many OFT entries point to that inode. The in-memory inode can be deleted only when reference count is zero
(disk inode, of course, cannot be deleted)

10/4 - Reference count → 0 : Delete the inode from memory
Links : The file has been deleted from disk itself. Delete disk inode too.

## VIRTUAL FILE SYSTEM (Linux)

* The file system is implemented using objects (structures) for inodes, directories, etc. and their corresponding functions.

- VFS on Linux supports these objects and functions
  - Any new file system must implement all of the above objects and their functions and hand over their pointers to VFS so that the file system can be used

- VFS provides abstraction to system calls. ☺
  - System calls take object pointer as arguments
    directory.find("foo.txt")
    inode.getblock(offset)

as opposed to having to traverse through and work on actual data structures of file systems.

Page No.
Date

Hierarchy

System Calls
↓
VFS Code
↓
File system implementation
↓ ↑
Disk Buffer Cache
↓
Device Driver
↓
Disk

## MEMORY MAPPING

mmap ( ____ , size .... )

anonymous
Gives general purpose memory to store programs data

file name 'File-backed page'
- Fetch the file from disk and put its data blocks into a new page in memory image
- If size is 4 kB, only first 4kB of file will be brought to RAM.
- Now, there is no need to use fd to edit a file. Can edit directly in RAM
- Write-back or write-through (via 'munmap()') are available.

- Read/write performance could be better for memory mapping or using file descriptors
- Memory mapping brings all data to memory in one shot.

Page No.
Date

# CONSISTENCY

- When user calls write(), and a new data block needs to be allotted
  1. Updating data bitmap
  2. Write the actual data block
  3. Update inode to include pointer to new block

What if only a subset of these 3 tasks has bee completed when power failure occurs?

- Perform the above operation in the order ~~1-2-3~~ 2-1-3
  - Write data first, then write metadata
    - The problem of showing garbage data to user is worse than losing the data and metadata

- If computer was not shut down properly, Linux performs fscheck to perform consistency checks and *may* be able to recover from inconsistency

→ Journalling
- Implemented by some file systems to ~~auto~~ recover from inconsistencies.
- Not implemented in Linux by default.

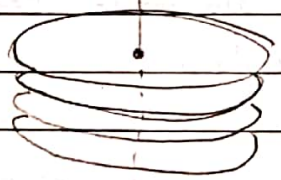
- Maintains a log of all changes to be made. Log is written to before making actual changes to disk.

- The entire set of changes is first logged into journal completely. Actual changes are made after logging. After entire set of changes is made, journal is erased.

- If during bootup, log is found to not be empty, the entire set of changes is made again to recover.

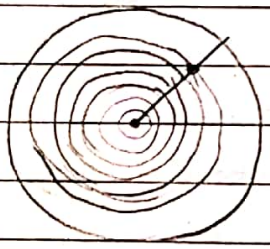
- Less time-efficient method (duh)

HARD DISK



Pile of spinning magnetic disks.

The arm of the disk can write/read from any given 'sector' of circle.

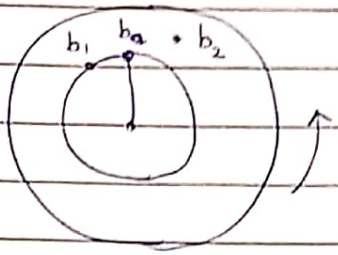


Random reads are much slower than reading from contiguous data blocks.

- The disk will accumulate all pending disk accesses and optimize rotation of disk, using 'scheduling algorithm's

① Shortest Seek position.
Handle requests at same radius first, then change arm position.

② Shortest positional latency




- Will have to wait almost a complete rotation to reach $b_1$
- Just change arm position and go to $b_2$ first.

∴ Shortest algorithms ① and ② will starve blocks that are far away on disk

X Fairness

③ Elevator Algorithm

Move arm in increasing order of radius.

Address all requests at that radius before seeking next radius.