EECS-343 Operating Systems
Lecture 15:
RAID & File Systems

Steve Tarzia

Spring 2019

Northwestern

RAID diagrams by https://en.wikipedia.org/wiki/User:Cburnett

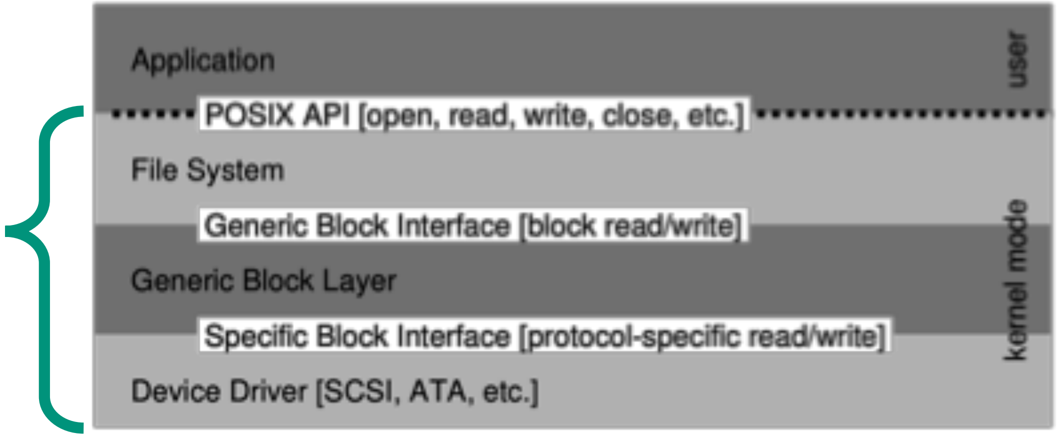
Announcements

- Project 4 is out due two weeks from yesterday.
- Office hours on Monday (Memorial Day) are cancelled.

Last Lecture – I/O and Disks

- OS interacts with devices by reading/writing ***device registers***
  - Each register has an ***I/O port*** address for in/out instructions, or
  - ***memory-mapped I/O*** uses special physical memory addresses (with `mov`)

- Storage is complex, so kernel functionality is divided into at least three layers:



- Random access to a magnetic disk is 1000x slower than sequential
  - Read head must ***seek*** and disk must ***rotate*** to reach a new sector

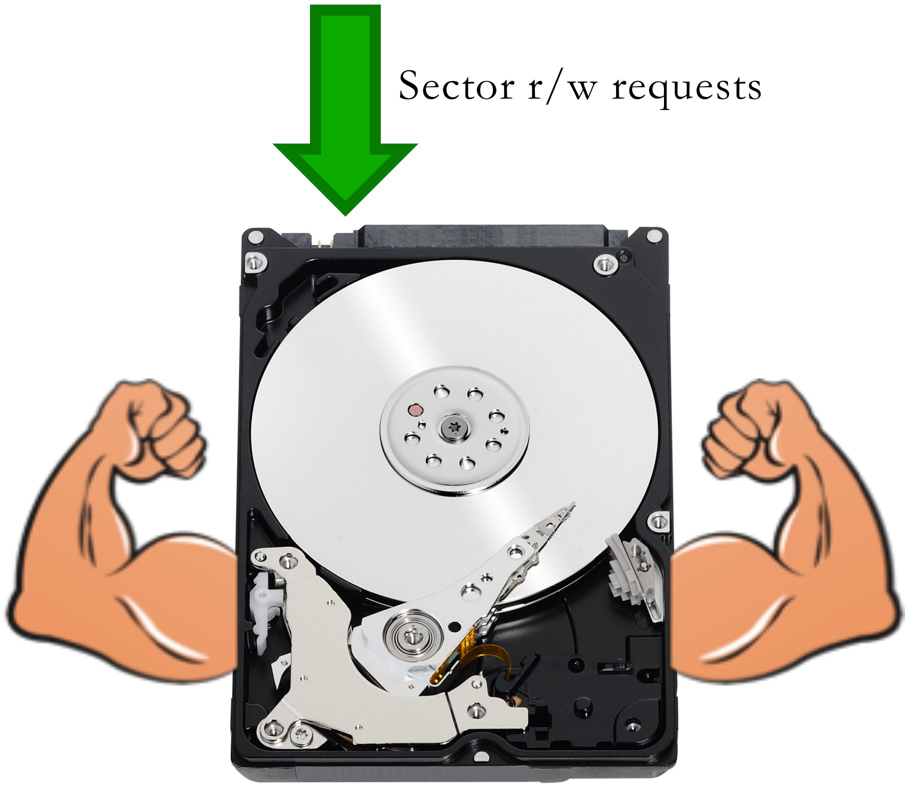
Redundant Array of Independent Disks (RAID)

- Disks have a few shortcomings:
  - ***Limited capacity*** (~8TB)
  - ***Limited throughput*** (~150MB/s)
  - ***Likelihood of failure*** (especially because they are mechanical)
- RAID uses multiple disks to solve these problems
  - Many different variations of RAID, depending on your budget and which of the above three problems are most important.
- Basic ideas are:
  - Increase ***capacity*** by making multiple disks available to store data.
  - Increase ***throughput*** by accessing data in *parallel* on multiple disks.
  - Reduce impact of a disk ***failure*** by storing data redundantly on multiple disks.
- Disk interface is very simple (just an array of sectors), so it's easy to create a **logical/virtual disk** made of sectors from multiple physical disks.

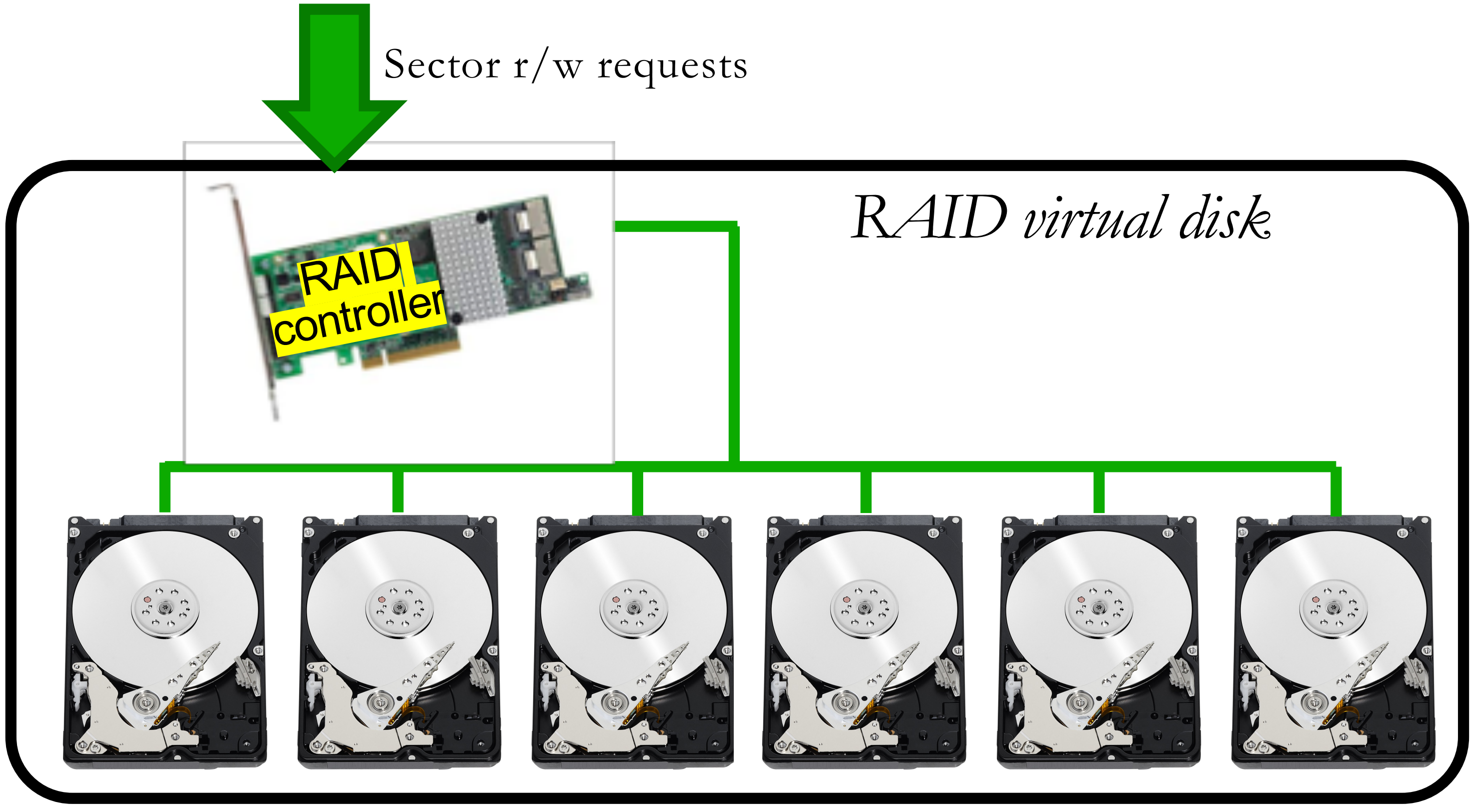
Basic idea of RAID

- Combine many disks to create one *superior* virtual disk.
- The RAID array provides the same interface as a single disk.

OS thinks it's dealing with this:



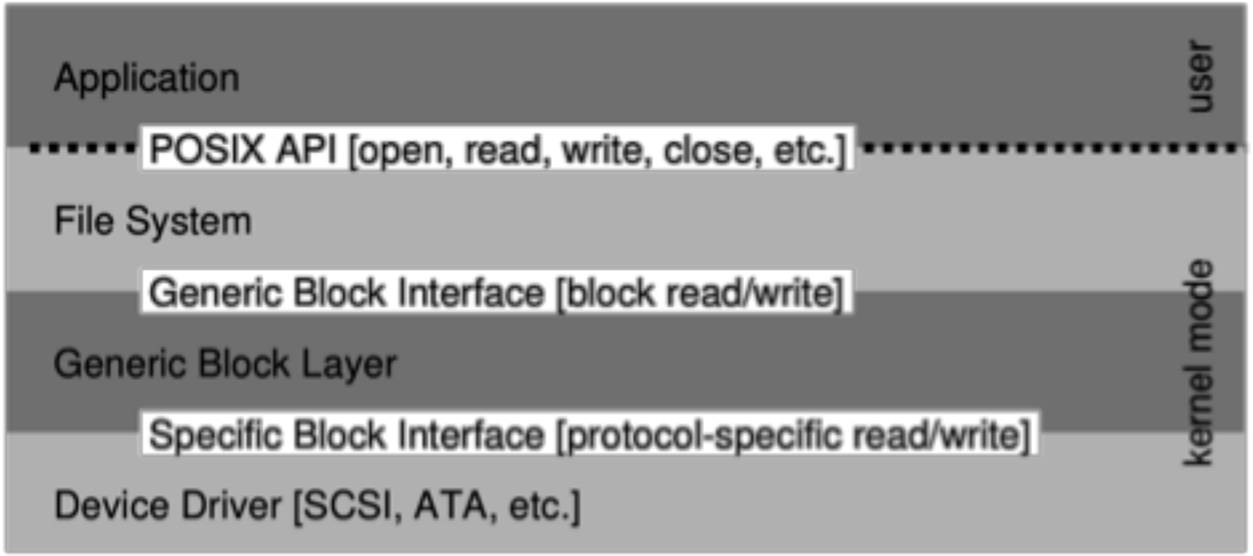
But it's just an illusion. The reality is:



How does RAID fit into the OS?

- ***Software RAID*** means that the OS is responsible for assembling multiple disks into a RAID.
  - Implements a generic block device.

SW and HW RAID work on different layers.



- ***Hardware RAID*** requires a specialized controller card that coordinates the multiple disks and presents the OS with the *illusion* of a single disk. (The previous slide showed hardware RAID.)
  - OS just needs a driver for the RAID controller, like any other disk controller.

RAID levels

- RAID 0 – ***Striping***:
  distribute data across 2 disks for twice the peak throughput
- RAID 1 – ***Mirroring***: copy data onto 2 disks to tolerate failure of one.
- RAID 4/5 – ***Parity:***

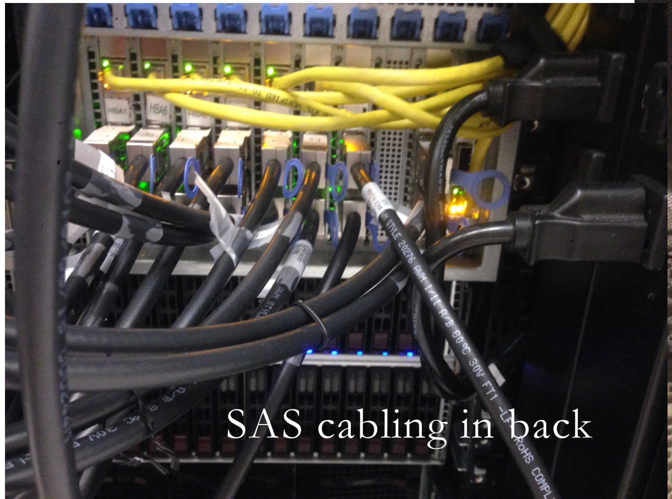
  start with N-1 striped disks, add disk N with parity information to tolerate failure of any of the disks.

  - Typically involves 3-7 disks.
  - Can include 2 parity disks for double-fault tolerance (called RAID 6)

A Database Server @ NU

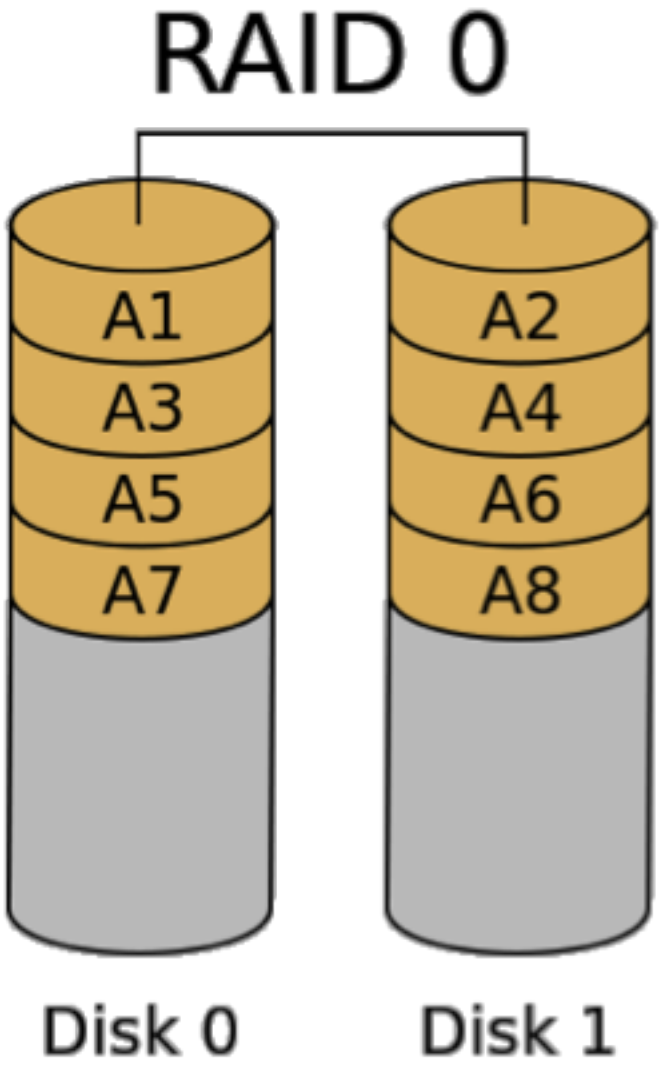
- 264 fast (10k RPM) magnetic disks (for production)
- 56 slow (7200 RPM) magnetic disks (for backup)
- ~150 TB storage capacity
- Comprised of 6 physical chassis (boxes) in one big cabinet, about the size of a coat closet.

Front view



SAS cabling in back

RAID 0 – Striping *(for throughput and capacity)*



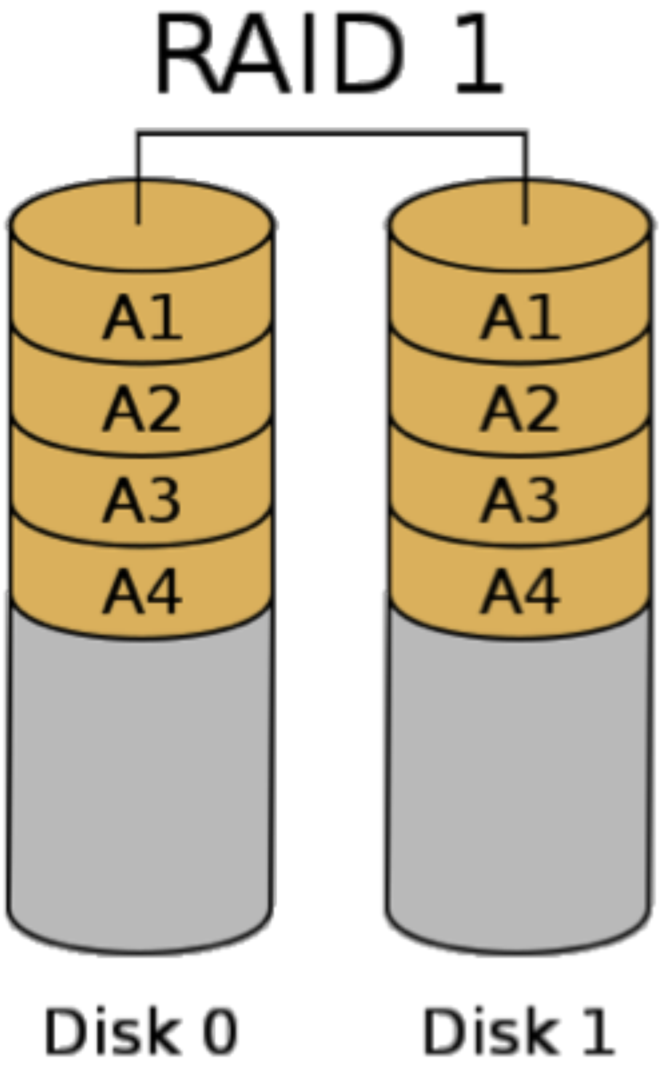
- Divide the logical disk into chunks (A1, A2, A3 …) ~128 kB
- Distribute the chunks regularly over two or more (*N*) physical disks.
- (+) Throughput for both random and sequential access scales with N.

$$T_{RAID0} = N * T_{disk}$$

- (+) Cost per byte is identical
- (–) But mean time to failure is worse because failure of a single disk is catastrophic:

$$MTTF_{RAID0} = MTTF_{disk}/N$$

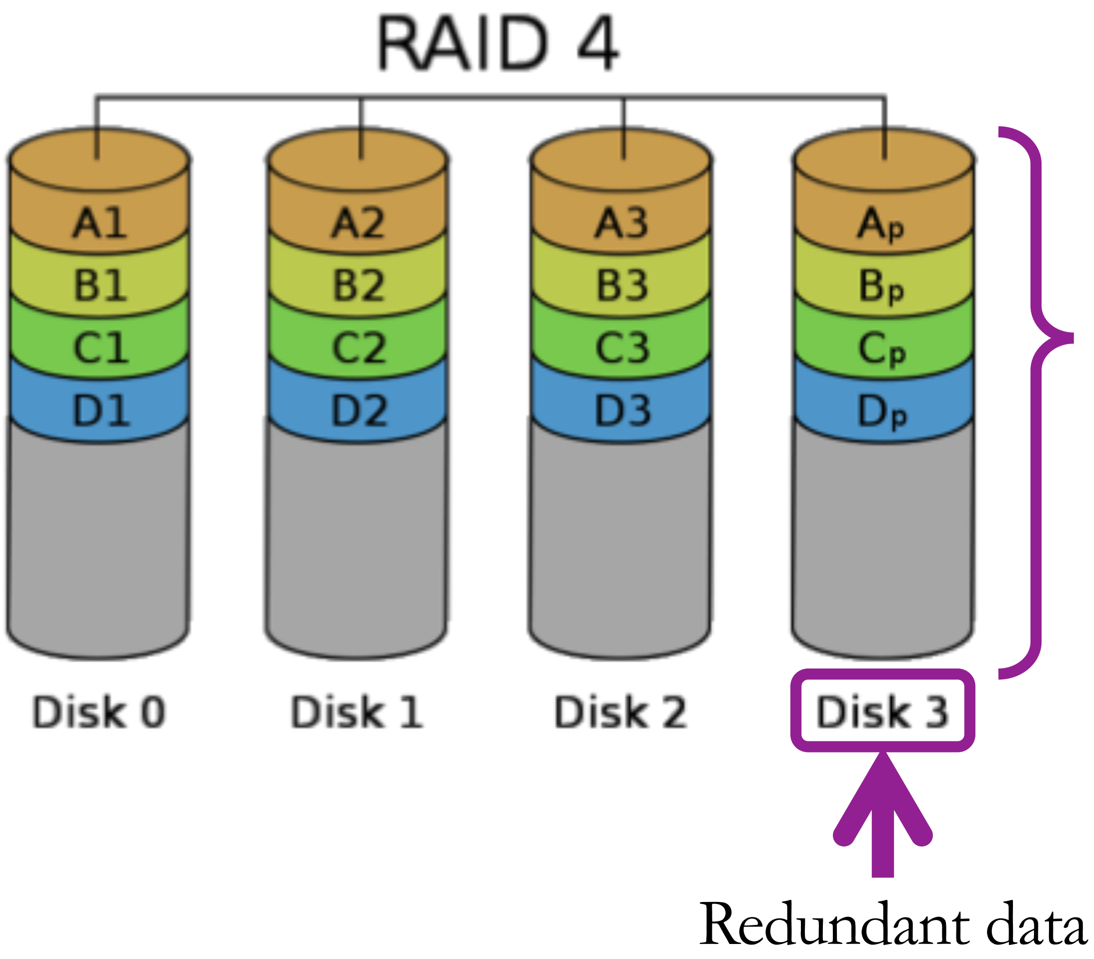
RAID 1 – Mirroring *(for fault tolerance)*



- Duplicate each chunk on each of N physical disks.
- (+) It is impossible to lose data unless all disks fail simultaneously.
  - Ie., failure window is reduced to the time it takes to replace a broken disk.
- (–) Throughput is not improved
- (–) Cost per byte is greater

$$\$_{RAID1} = N * \$_{disk}$$

RAID 4 – Parity *(for fault tolerance, capacity & throughput)*



- Distribute the chunks across the first (N-1) disks.
- On the Nth disk, store a corresponding ***parity*** chunk.
  - Parity chunk is redundant data about a set of chunks (a ***stripe***)
- Can tolerate loss of any one disk

Parity bit is added to allow filling-in a missing bit

- ***Even parity*** – add a 0 or 1 such that the total number of 1's is even.
- eg., given [0, 0, 1, 0, 1, 1] parity bit would be 1. The sequence has three ones, so we add a one to yield an even number of ones (4):

```
[0, 0, 1, 0, 1, 1, 1]
```

original data *parity bit*

- If a bit is lost, the parity bit allows us to infer the lost bit's value:

```
[?, 0, 1, 0, 1, 1, 1]
```

  - The missing first bit must have been 0 because we already have an even number of ones in the remaining positions.

Parity chunk

Disk 0	Disk 1	Disk 2	Disk 3 (parity)
0001 0010 1100 1100	0000 1111 0000 1111	1101 1111 0011 0001	1100 0010 1111 0010
1111 1111 1111 1111	0001 0001 0001 0001	1101 1001 0110 0110	0011 0111 1000 1000
0000 0000 0000 0000	1101 1011 0011 0011	1111 0011 0011 1000	0010 1000 0000 1011

Useful storage capacity

Redundancy overhead

- Parity is computed bit-wise across corresponding chunks.
- Chunks are ~128 kB
- Writing a small file will involve one disk *plus the parity disk*.
  - (parity disk can become a bottleneck)
- Writing a large file will involve all the disks.

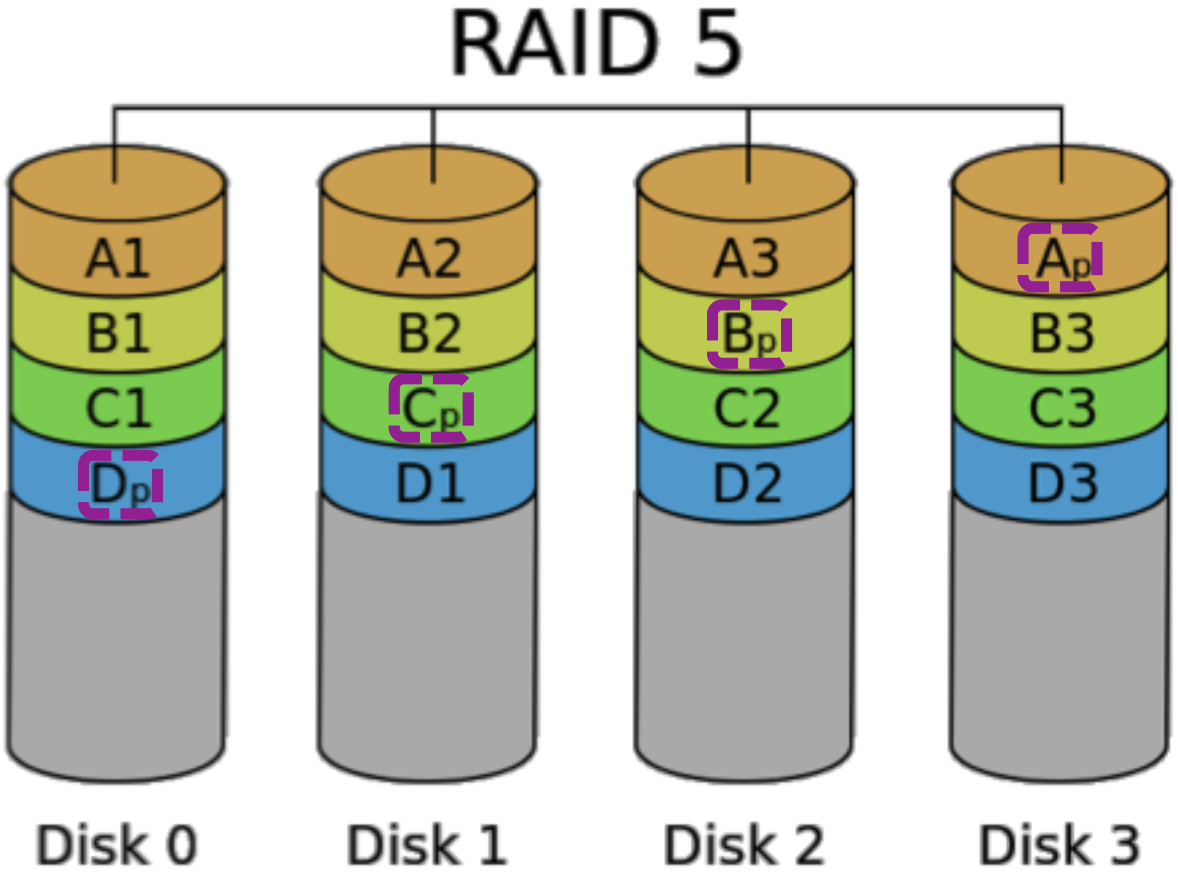
Rebuilding an array after failure

Disk 0	Disk 1	Disk 2	Disk 3 (parity)
0001 0010 1100 1100		1101 1111 0011 0001	1100 0010 1111 0010
1111 1111 1111 1111		1101 1001 0110 0110	0011 0111 1000 1000
0000 0000 0000 0000		1111 0011 0011 1000	0010 1000 0000 1011

Disk failed!

- If a disk fails, then we remove it and replace it with a working disk.
- Then scan through the entire array to compute and write missing data.
  - This is called “rebuilding” the array
  - We cannot tolerate another disk failure until rebuild completes.
  - Reads/write can continue while array is rebuilding!

RAID 5 – Distributed Parity *(the winner in practice)*

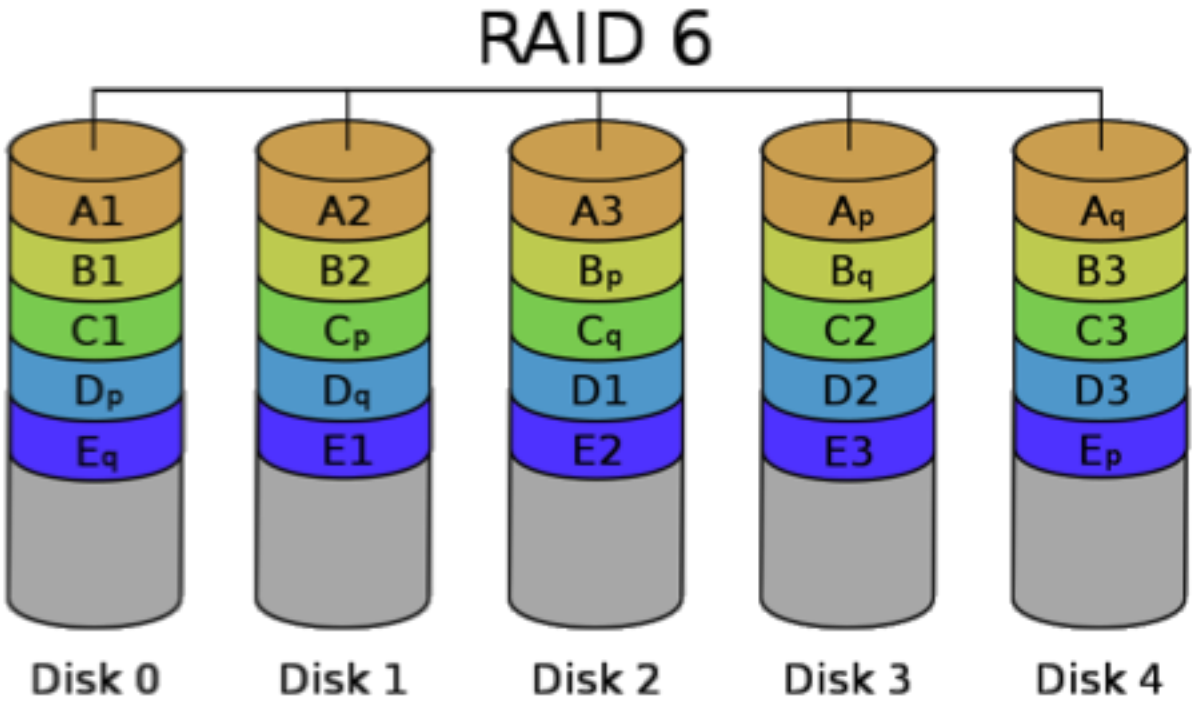


- Distribute parity chunks across the disks, to avoid a small-write bottleneck
- (+) Failure of one disk is OK
- (+) Throughput is good

  $T_{RAID5} = (N-1) * T_{disk}$
- (+) Cost per byte is good

  $\$_{RAID1} = N/(N-1) * \$_{disk}$
- (–) High overhead for small N
- (–) Failure risk is high for large N
- N is typically 3 to 8

RAID 6 – Double Parity *(for large arrays)*



- Add another disk and keep two parity chunks per stripe
  - 2nd parity is computed differently
- (+) Failure of ***two*** disks is OK
- (~) Throughput is less:

$$T_{RAID5} = (N-2) * T_{disk}$$

- (~) Cost per byte is higher:

$$\$_{RAID1} = N/(N-2) * \$_{disk}$$

- Makes sense for larger N (>8)

Intermission

"It was much nicer before people started storing all their personal information in the cloud."

Filesystem basics

- A ***filesystem*** is an abstraction & interface for persistent storage.
  - Storage devices (eg., disks) are just big arrays of bytes.
  - The filesystem organizes the storage space for ease-of-use and sharing among many processes/users on a system.
  - Unlike memory, it's often directly accessible to the computer user.
- Usually structured as:
  - A tree of ***directories/folders***
  - ***Files*** to store an array of bytes, each located within a directory
  - With unique ***names*** for each file or directory with a directory, and
  - ***Metadata*** for each file/directory (permissions, owner, modified time, etc.)
- Many different filesystems have been developed over the years:
  - FAT32, NTFS (Windows), ext4 (Linux), HFS, APFS (Mac), ZFS, etc.
  - Some include extra features like encryption, compression, backups.

In other words…

- A **filesystem** is a ***data structure*** for storing files on a disk.

Key Challenges:

- Files are added and deleted over time. Free space must be managed.
- Files can grow and shrink.
- Should tolerate sudden electrical power loss without corruption.
- Performance should be optimized for magnetic disks:
  - Random access is slow, but sequential access is fast.

Application-level interface (syscalls)

- `open` (or create) a file with a given *path* (directories & name) and set the file pointer to the beginning of the file
- `read` up to a certain number of bytes from an open file, and move the file pointer for the next read.
- `write` an array of bytes to an open file (and move the pointer)
- `close` an open file
- `lseek` to move the file pointer to a certain index in the file
- `fsync` to push changes to disk immediately (flush dirty data)

Syscall trace

- `strace` command (on Linux) shows syscalls used by a process
- Here, open() return *file descriptor* number 3.
- Unix standard file descriptors are:
  - 0: stdin
  - 1: stdout
  - 2: stderr
  - These are always open and available in a Unix process.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)        = 3
read(3, "hello\n", 4096)                 = 6
write(1, "hello\n", 6)                   = 6
hello
read(3, "", 4096)                        = 0
close(3)                                 = 0
...
prompt>
```

- Unix also uses file interface for many "non-file" things that can be read/written to, like stdout/stdin to/from the terminal.

More file-related syscalls

- `stat/fstat` gets file metadata (data about the data)
- `rename` to move a file
- `unlink` to remove a file
- `mkdir` to make a directory
- Linux:
  - `getdents` to list the contents of a directory
- xv6:
  - `open` and `read` a directory to get a raw directory listing

File/directory metadata (Linux)

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number (low-level name) */
    mode_t    st_mode;        /* File type and mode (permissions) */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
};
```

Filesystem Links

- **`ln`** unix command creates a link to a file – like a pointer.
  - Allows a file to exist in multiple paths without wasting space
- ***Hard link*** creates another entry in a directory referring to the same inode number (disk address).
- ***Symbolic/Soft link*** is a special file whose contents is just the string path of another file.
  - Symlinks are much more common in modern practice (`ln -s`)
  - Allow referring to file in other filesystems
  - But may lead to a ***dangling reference*** – the referred-to file may be deleted

Making and mounting a filesystem

- On Linux, **`mkfs`** command creates a new filesystem on a block device.
- xv6 Makefile creates a *filesystem in a file!* – `fs.img`
- The **`mount`** command tells the OS to add a filesystem under a directory in the virtual file system. Without any parameters is describes current mount points:

```
[steve@vortex ~]$ mount
/dev/md3 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw,rootcontext="system_u:object_r:tmpfs_t:s0")
/dev/md1 on /boot type ext4 (rw)
/dev/md4 on /home type ext4 (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
192.168.0.5:/pool2 on /mnt/pool2 type nfs
(rw,rsize=8192,wsize=8192,vers=4,addr=192.168.0.5,clientaddr=192.168.0.6)
```

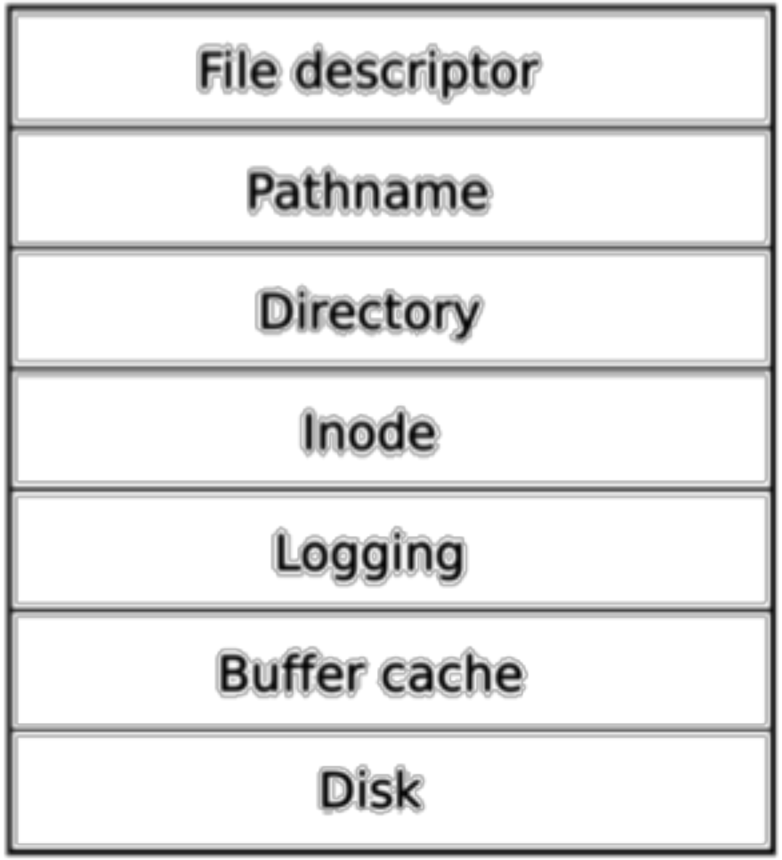
xv6 file system goals

- Create on-disk data structures to:
  - Represent a tree of directories and files
  - Record which disk blocks store each file's data
  - Track free blocks on the disk
- Support ***crash recovery***
  - Behave reasonably if the machine if powered off at any time
- Support concurrent access by many processes.
- Operate quickly by using an in-memory cache.

Note: in xv6, disk ***block*** means one disk sector – 512 byte unit of disk storage

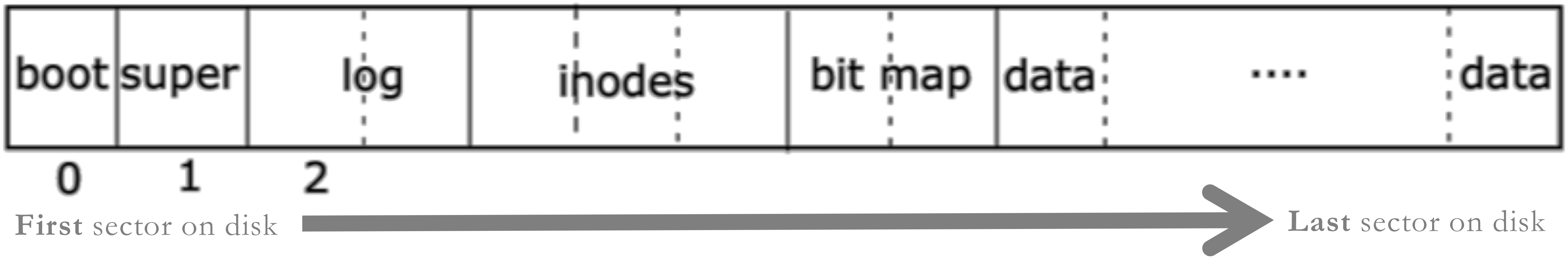
Layered storage system design (in xv6)

- Layered design makes code easier to understand & write.
- Lower-level details are hidden at each layer



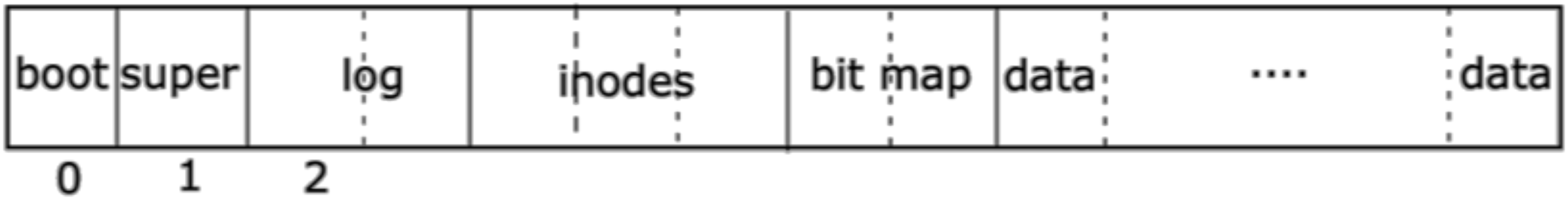
- ➢ Apps use these to open, close, read, write, etc.
- ➢ Absolute location, eg. "/home/steve/hello.txt"
- ➢ Containers associating names with inodes
- ➢ Organizes used blocks (& mutex, caching)
- ➢ Makes block writes appear as atomic transactions
- ➢ For performance & mutual exclusion
- ➢ Low-level driver handles device registers

Layout of xv6 filesystem on disk



- ***Boot block*** contains OS initialization code
- ***Superblock*** has some high-level info about the filesystem structure
- ***Log*** stores block writes which are not completed yet
- ***Inodes*** (each is 64 bytes) store metadata for one file or directory.
- ***Bitmap*** indicates which disk blocks are free (data blocks in particular)
- ***Data blocks*** store file data (inodes refer to data blocks)

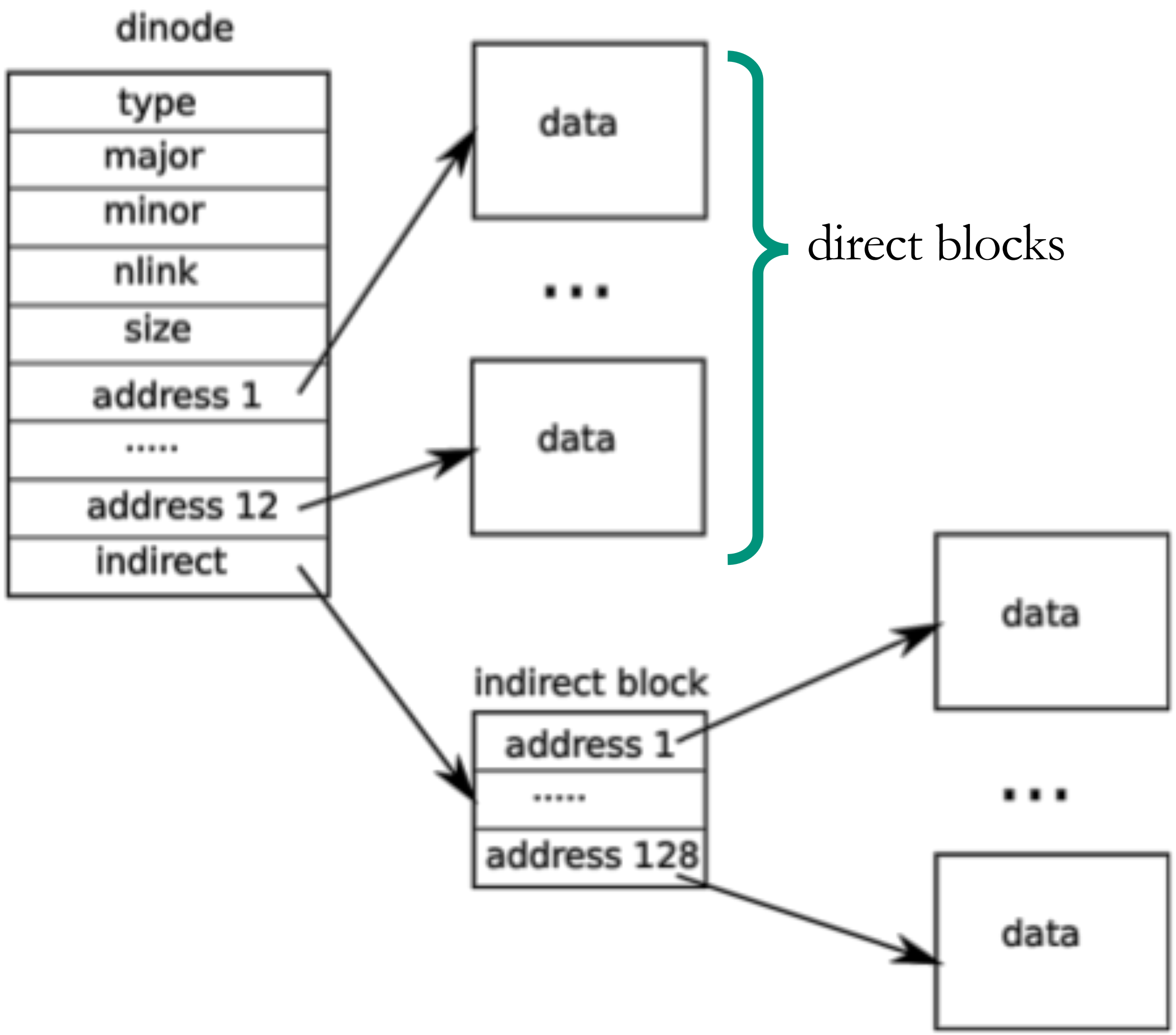
An inflexible design



- The size of each of these regions is hard-coded in the superblock
- Number of inodes is determined when the filesystem is created (mkfs)
  - Inode count is the maximum number of files/directories allowed
    - (by default, mkfs creates 200 inodes, consuming 25 blocks)
  - Tradeoff between inode region size and data region size
  - Allocate fewer inodes if you plan to store just a few big files
  - But, in xv6, you cannot change the number of inodes after ***formatting***
- But, the presence of a transaction log is a nice, modern feature.

Inodes (xv6)

- Each file/directory is represented by an inode (`struct dinode`)
- A file inode stores:
  - Reference count (# of hard links)
  - Total file size
  - Array of data blocks storing the file's data (***direct blocks***)
  - Optional ***indirect block*** address, for files larger than 6kB.
- xv6 files can be 70kB at most!
- Inodes are 64 bytes each



Directory inodes (xv6)

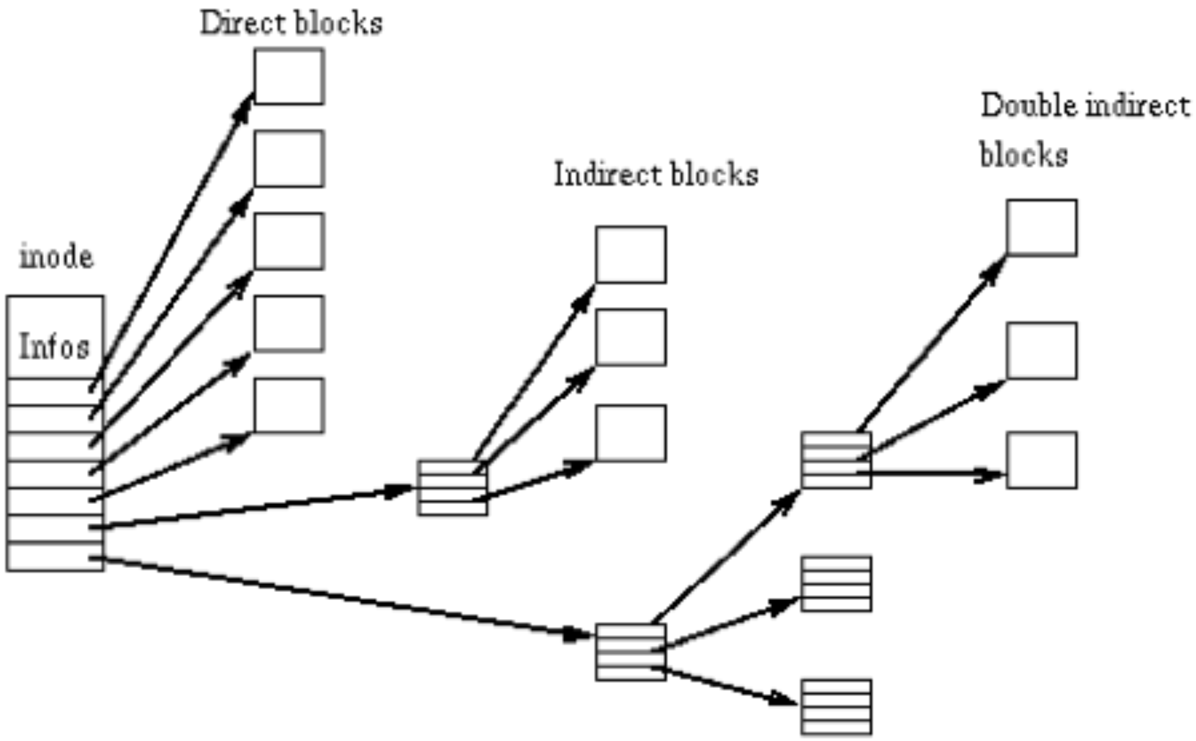
- A Directory is like a file containing an array of <name, inode> pairs:

```
struct dirent {
  ushort inum;
  char name[DIRSIZ]; // 14
};
```

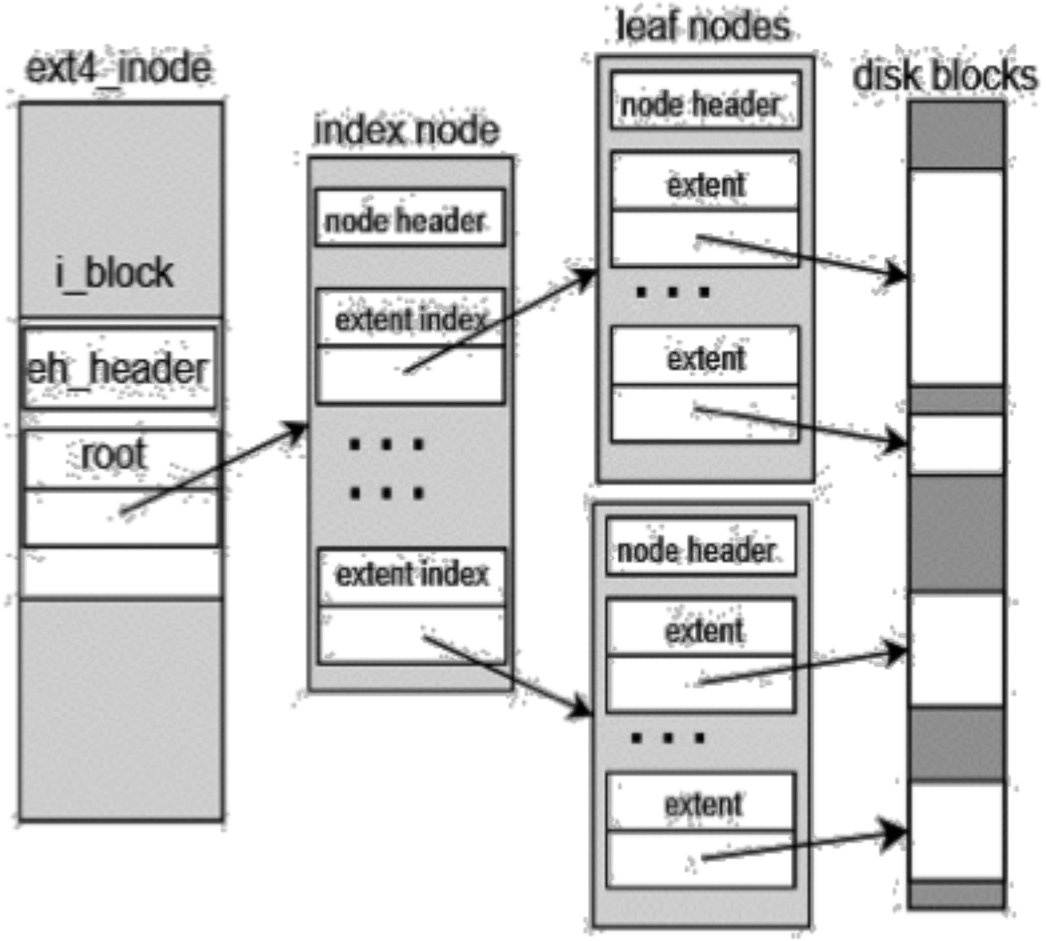
- Inode type is set to T_DIR instead of T_FILE
- Every directory contains two special entries:
  - "`..`" pointing to parent directory
  - "`.`" pointing to self

Storing larger files

ext3: double & *triple* indirect blocks

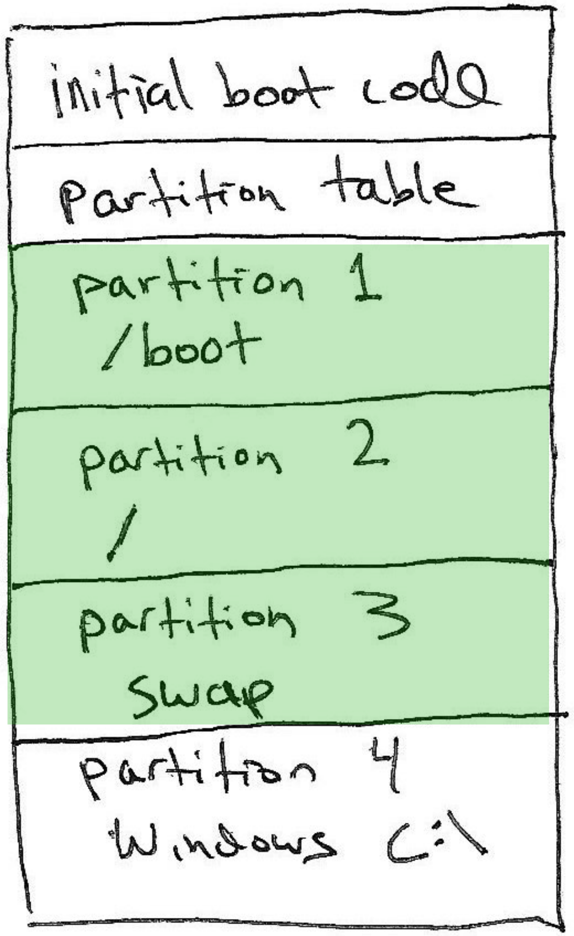


ext4: extents



Disk partitions

Disk A



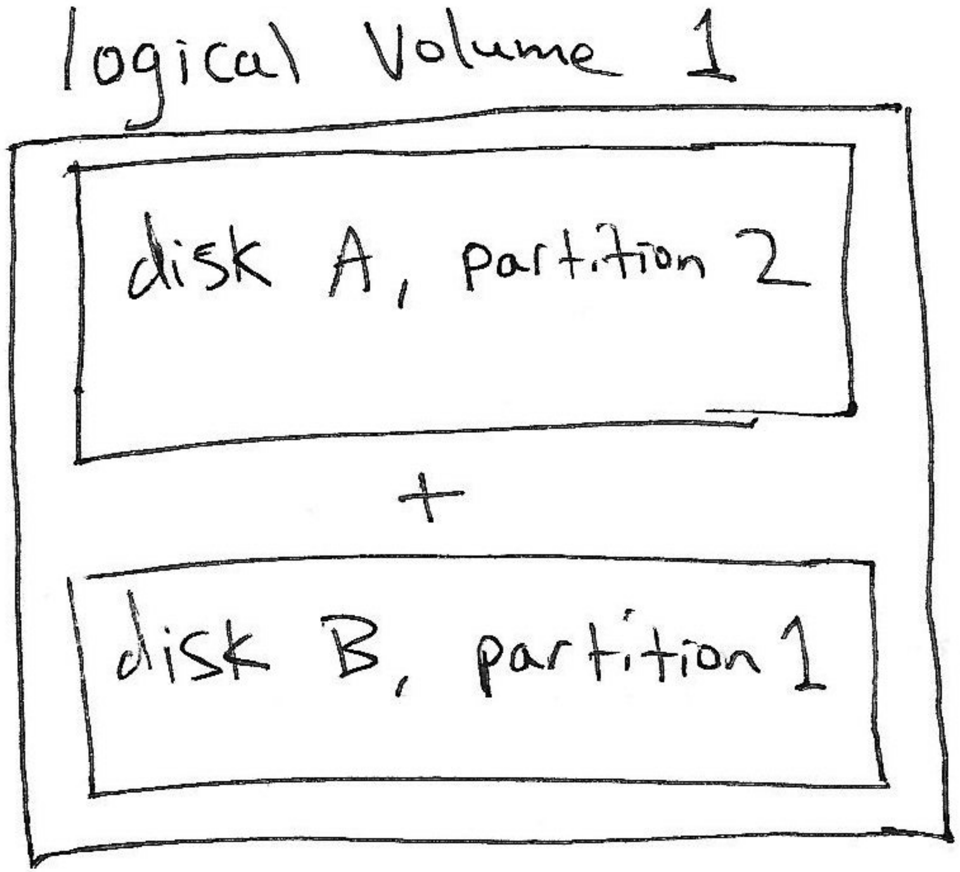
(not drawn to scale)

- Most computers have one physical disk,
- But they may require multiple filesystems.
- A disk partition is a contiguous chunk of the disk that can be formatted to store a filesystem.

At left, we have:

- Three different Linux partitions: /boot, swap, /
- A Windows partition.
  - Each of the partitions may be formatted differently.
- At bootup, initial boot code will present user with a menu to choose Windows or Linux boot.

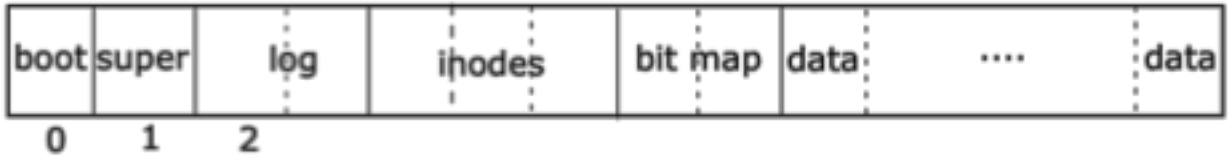
Logical Volume Management (LVM)



- It's sometimes convenient to combine multiple disk partitions into a bigger **logical volume**.
- The concept is similar to software RAID, but it does not provide performance or redundancy benefits.
- Allows user to increase the size of a filesystem by later adding another disk.
- I think this feature is overused as a default setting on modern Linux distributions.

Recap – RAID & File Systems

- RAID allows multiple disks to act together for better throughput, capacity, and/or fault tolerance.
  - ***Parity*** is used in ***RAID5*** to achieve all of the above.
- OSes have a application-level API (syscalls) for file I/O:
  - open, read, write, seek, stat, fsync, rename, unlink, mkdir
- ***Filesystem*** is a data structure the OS uses to organize disk space.



- Each file/directory has an ***inode*** storing metadata & pointers to data blocks.