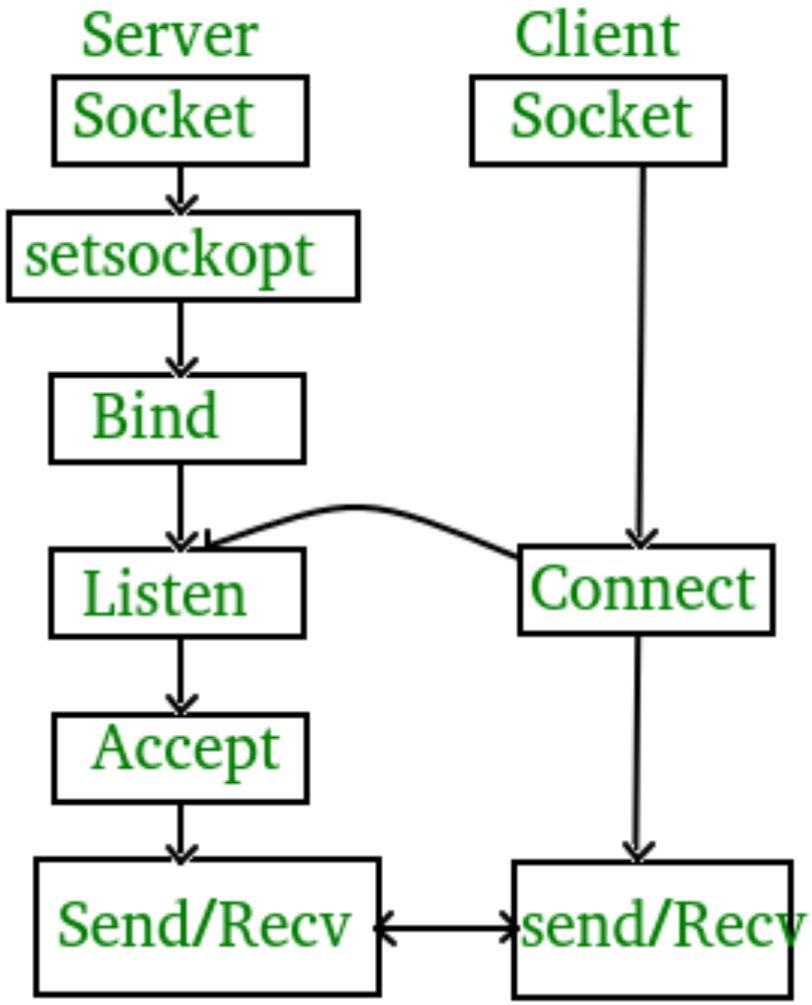
Devices and File Systems

Dubey

Perils of Using Low-level Sockets



- The asymmetric structure is one cause for confusion
- Second, each step shown is a low-level system call that takes in various arguments
  - When used in C/C++ languages, these parameters are pointers to structures that need to be cast from one type to another
  - Other parameters should be provided in such a way that they are all consistent with each other
  - The send-receive loop has to be carefully crafted
  - When used in the context of multi threading, this task becomes even harder
- Third, writing code that is portable across OS is yet another problem area
- All of this leads to very high possibility of committing errors that are hard to debug

See scaffolding code to convince yourself as to how hard it is to write low-level socket code

Emergence of Higher-level Frameworks

- To address these problems, several different middleware frameworks have emerged
- The ACE framework (developed by Prof. Doug Schmidt) is a widely used C++ framework with number of design patterns
- Languages like Java provide portable code
- CORBA like frameworks provide language-agnostic, object-oriented style distributed remote procedure call capabilities along with its own serialization capabilities
- More recently, frameworks like ZeroMQ and gRPC have become more widely used to write networking/distributed systems code using more intuitive and higher-level abstractions
  - These frameworks also support bindings to multiple different programming languages
  - They can leverage different serialization frameworks

A big concept is starting a new process or thread for every new incoming connection.

Devices

Motivation

What good is a computer without any I/O devices?

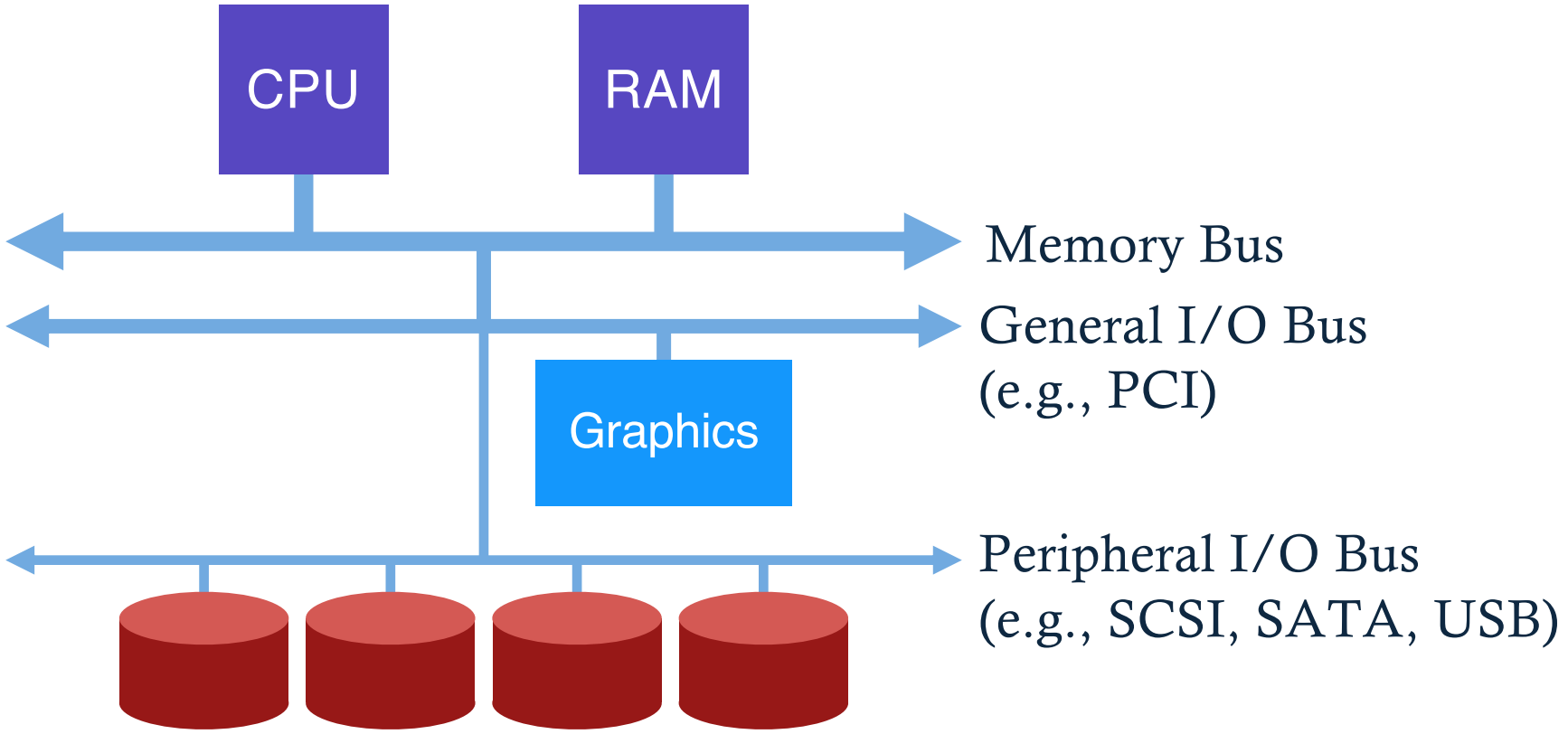
- e.g., keyboard, display, disks

We want:

- **H/W** that will let us plug in different devices
- **OS** that can interact with different combinations

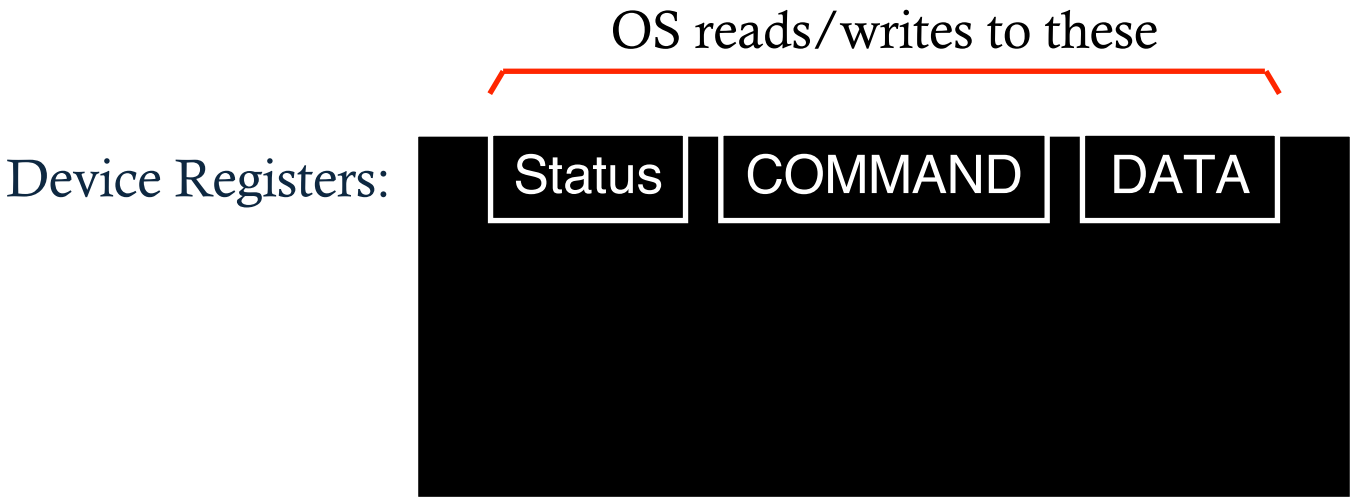
- I/O also allows for *persistence*
  - RAM is *volatile*, i.e., contents are lost when the machine restarts

Hardware support for I/O

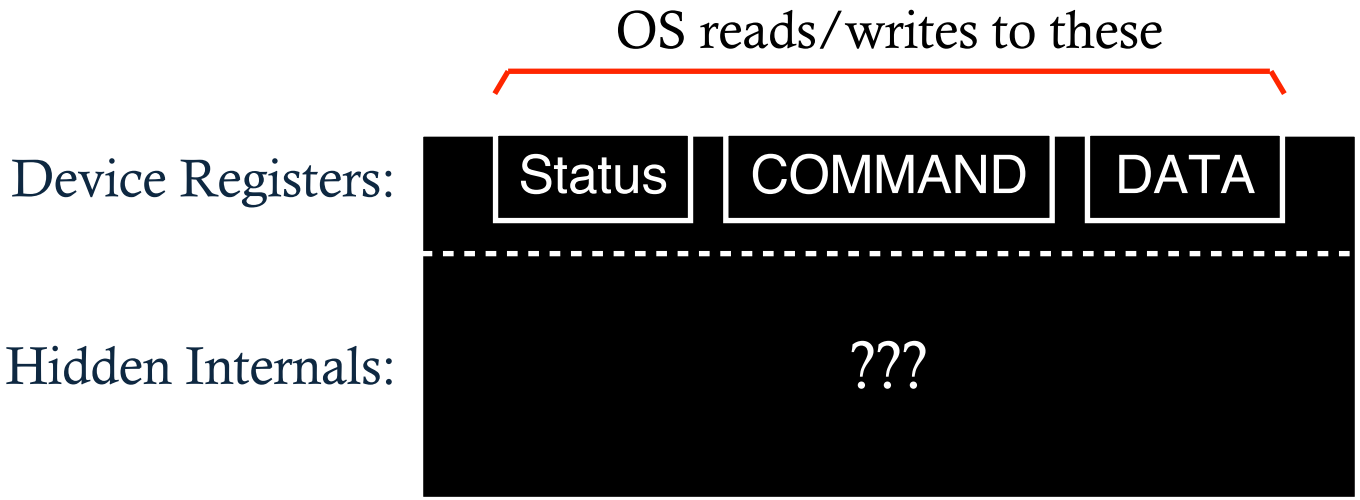


Why use hierarchical buses?

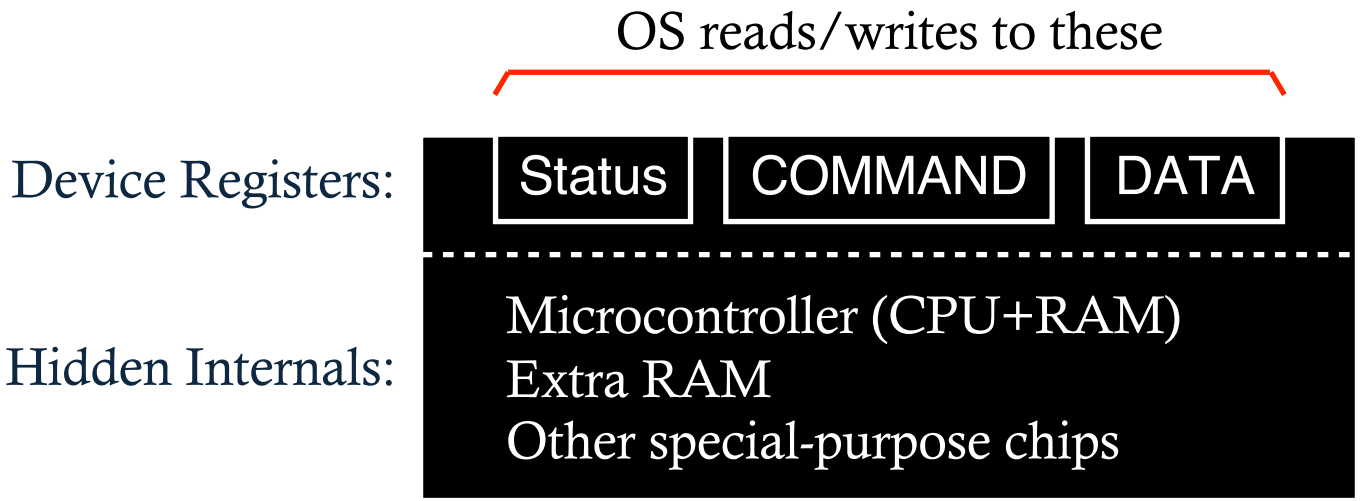
Canonical Device



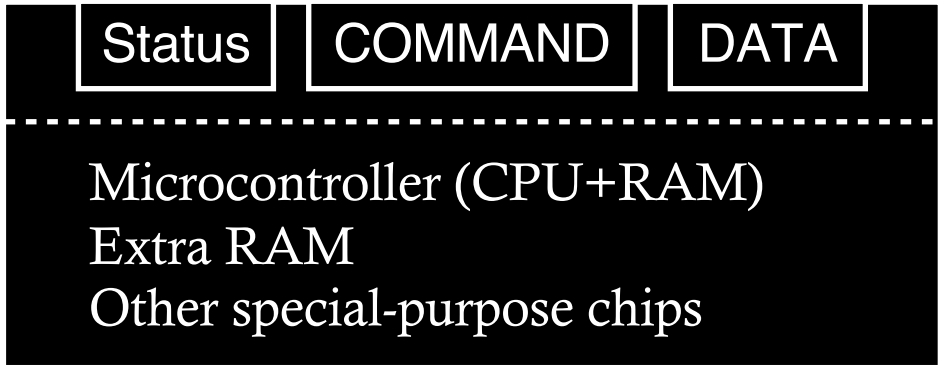
Canonical Device



Canonical Device



Example Write Protocol




```
while (STATUS == BUSY)
  ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
  ; // spin
```

This is called polling when the processor "asks" what the hardware is doing, often continuously

CPU:

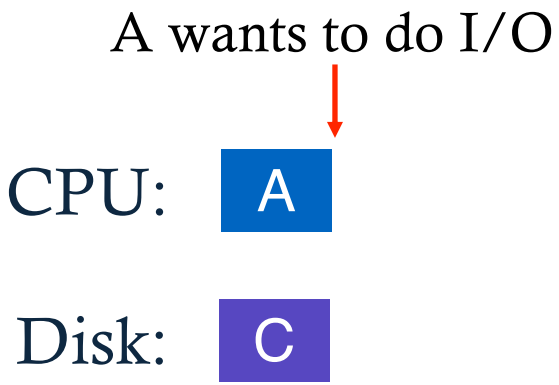
Disk:

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

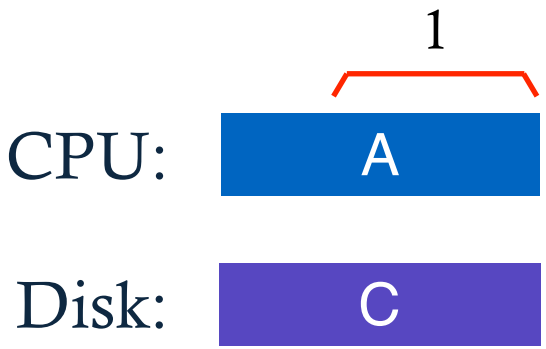
CPU: A

Disk: C

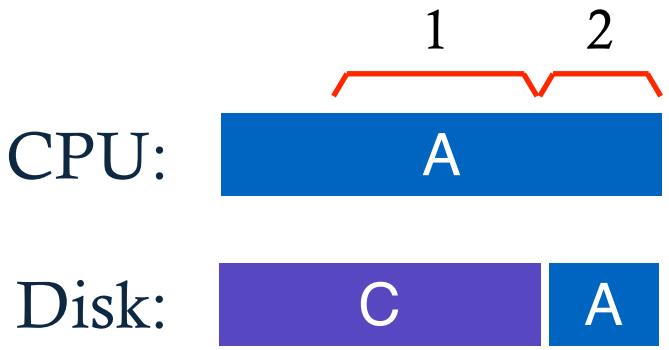
```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



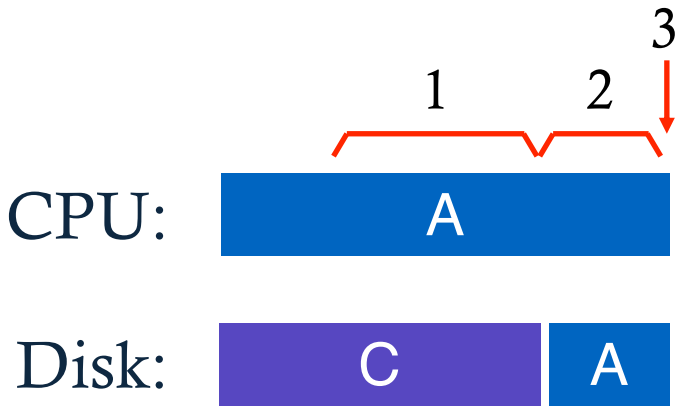

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



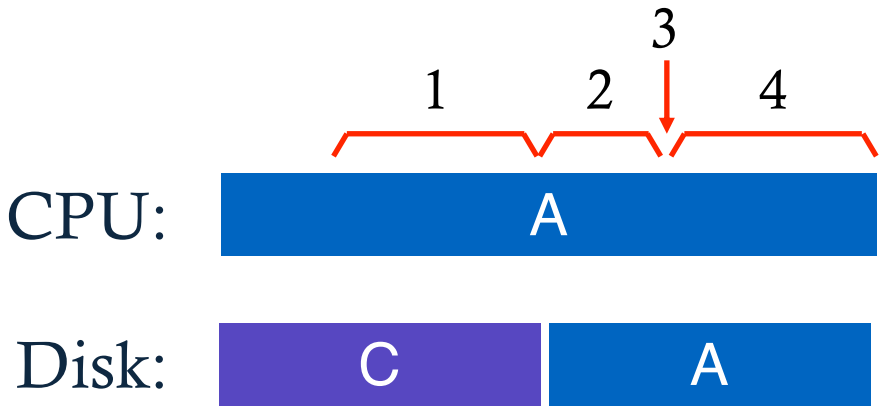

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



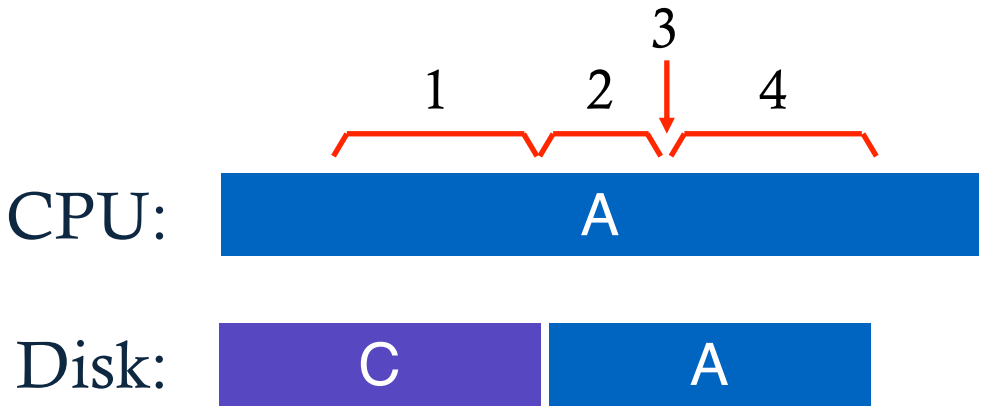

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```




```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

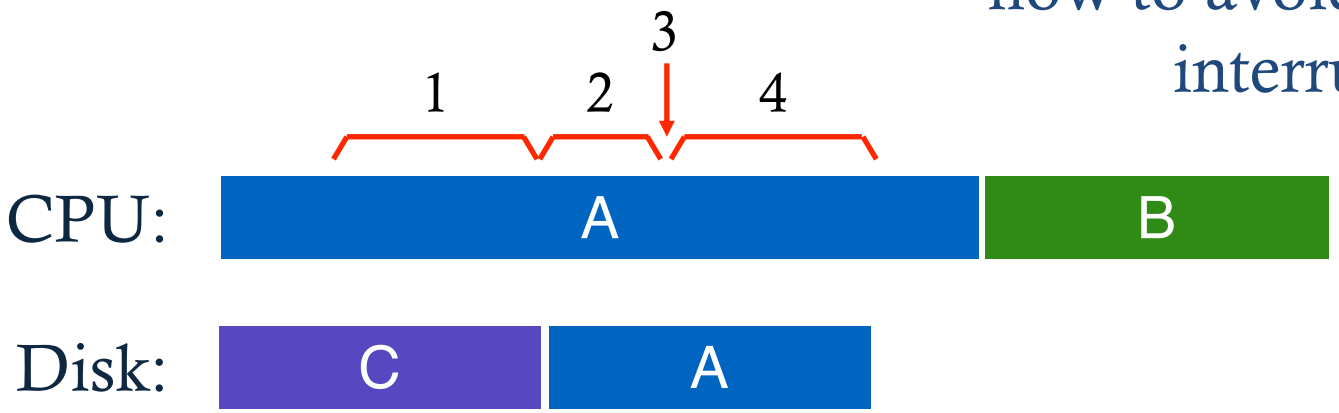



```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



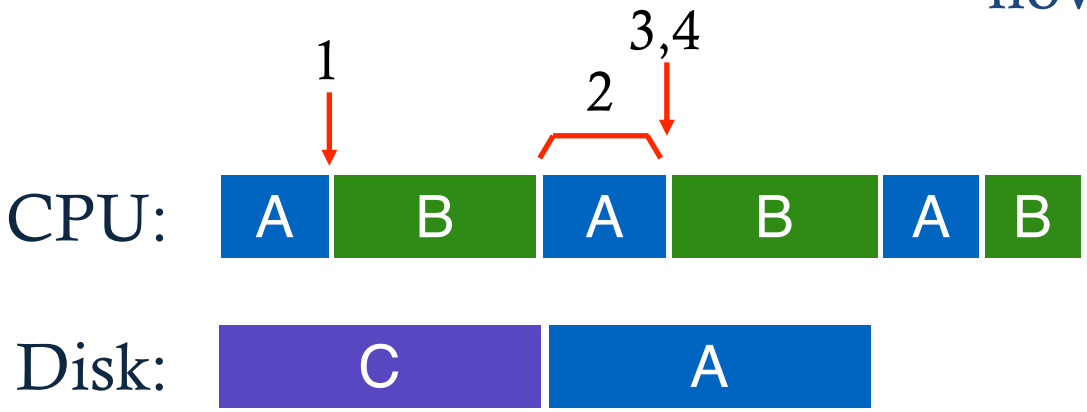

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

how to avoid spinning?
interrupts!




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

how to avoid spinning?
interrupts!




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

Interrupts vs. Polling

Are interrupts ever worse than polling?

Fast device: Better to spin than take interrupt and context-switching overhead

- Device time unknown? Hybrid approach (spin then use interrupts)

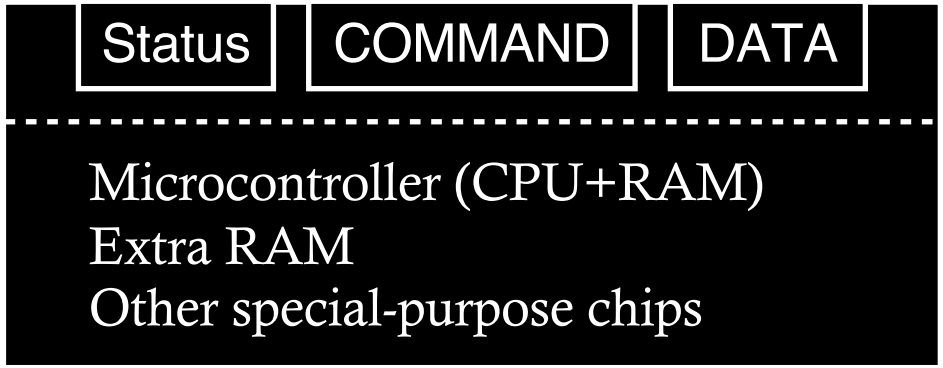
Flood of interrupts arrive

- Can lead to livelock (always handling interrupts)
- Better to ignore interrupts while making some progress handling them

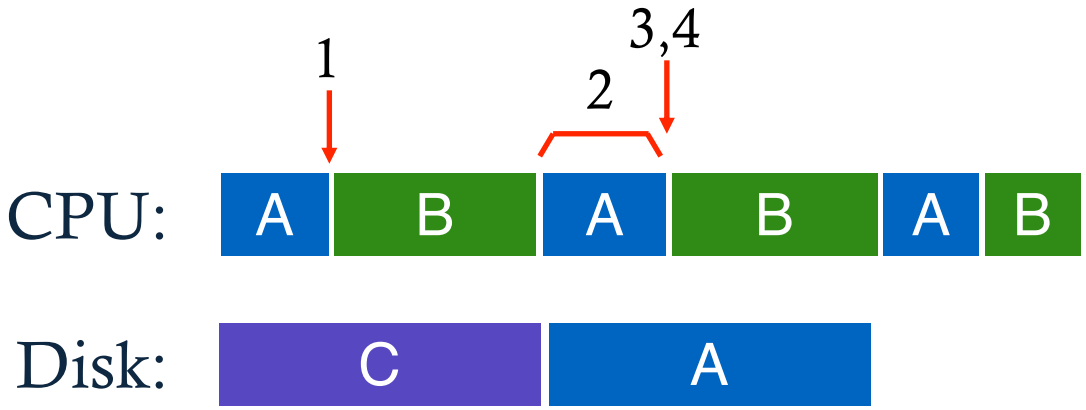
Other improvement

- Interrupt coalescing (hardware batches together several interrupts)

Protocol Variants



- **Status checks**: polling *vs.* interrupts
- **Data**: programmed I/O (PIO) *vs.* direct memory access (DMA)
- **Control**: special instructions *vs.* memory-mapped I/O




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

what else can we optimize? data transfer!

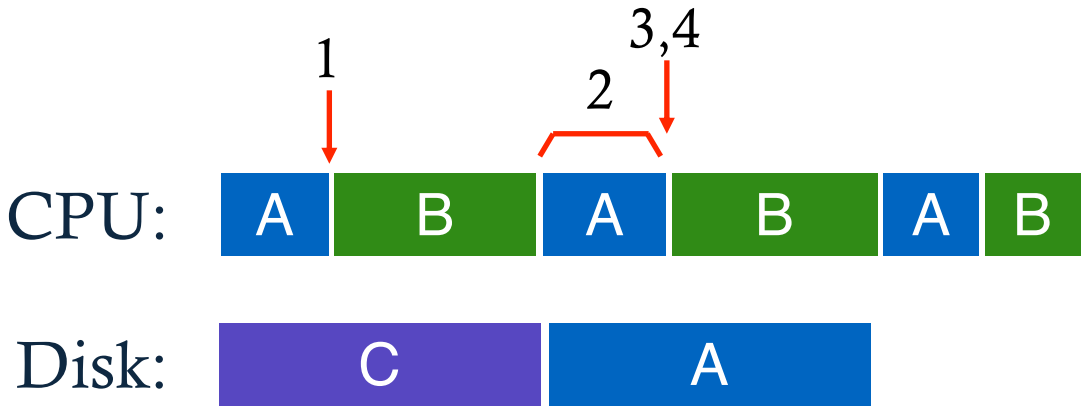
Programmed I/O vs. Direct Memory Access

PIO (Programmed I/O):

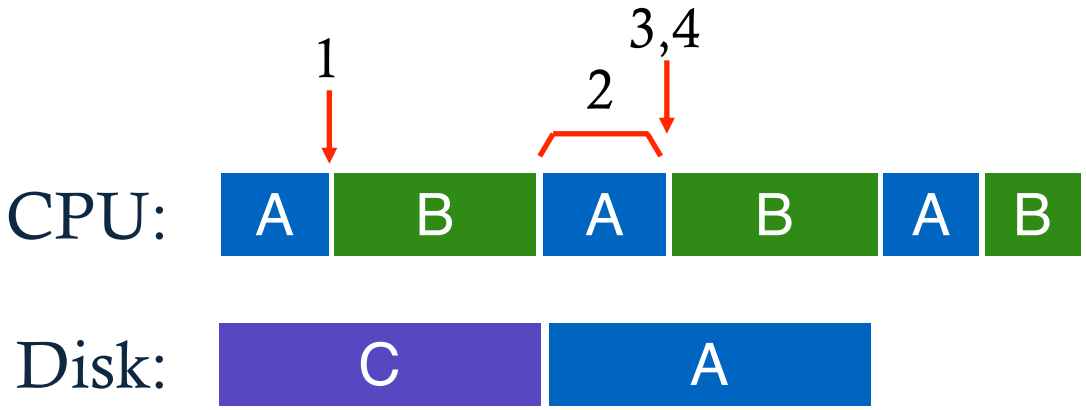
- CPU directly tells device what the data is
- One instruction for each byte/word
- Efficient for a few bytes/words, but *scales terribly*

DMA (Direct Memory Access):

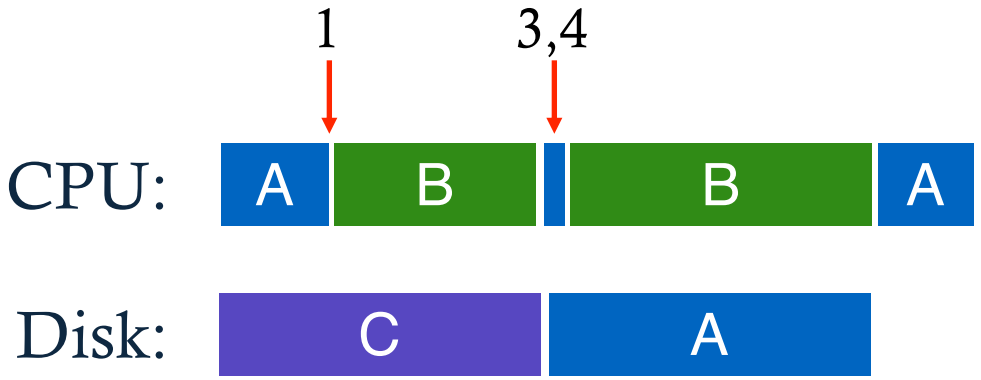
- CPU leaves data in memory
- Device reads/writes data directly from/to memory
- One instruction to send a pointer to the data to send
- Efficient for large data transfers




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

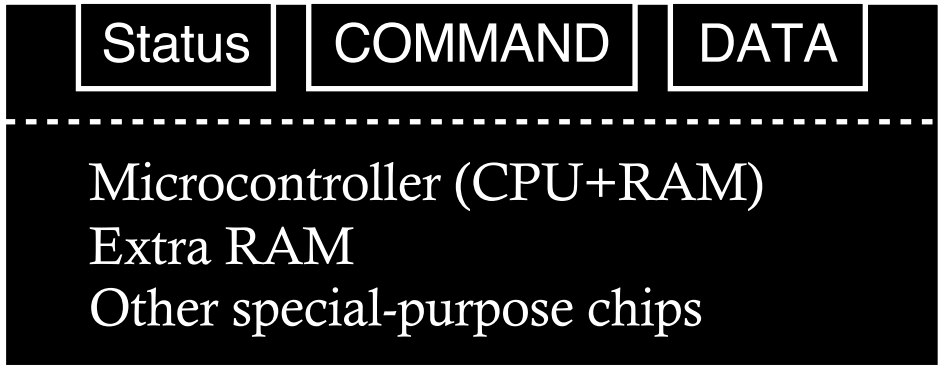



```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```




```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```

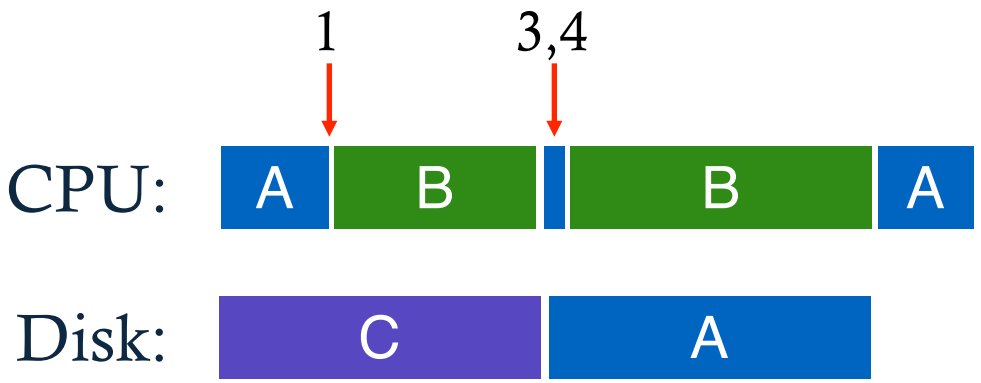
Protocol Variants



Status checks: polling *vs*. interrupts

Data: PIO *vs*. DMA

Control: special instructions *vs*. memory-mapped I/O




```
while (STATUS == BUSY)                    // 1
      wait for interrupt;
Write data to DATA register               // 2
Write command to COMMAND register         // 3
while (STATUS == BUSY)                    // 4
      wait for interrupt;
```

how does OS read and write registers?

Special Instructions vs. Mem-Mapped I/O

Special instructions

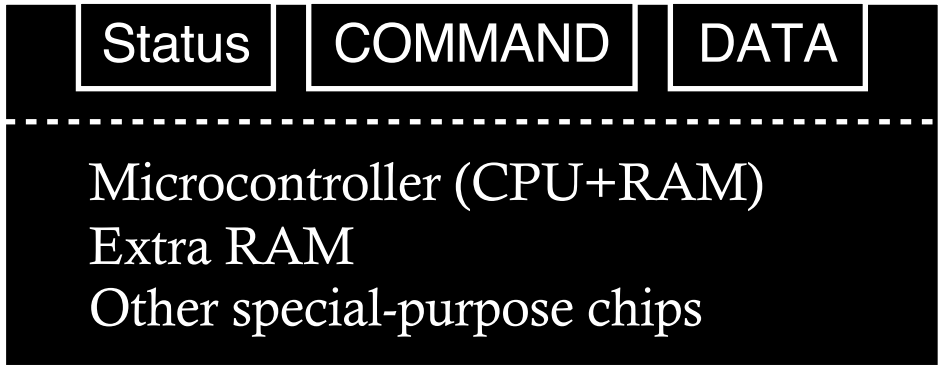
- each device has a port
- in/out instructions (x86) communicate with device

Memory-Mapped I/O

- H/W maps registers into address space
- loads/stores sent to device

Doesn’t matter much (both are used)

Protocol Variants



Status checks: polling *vs.* interrupts

Data: PIO *vs.* DMA

Control: special instructions *vs.* memory-mapped I/O

Variety is a Challenge

Problem:

- many, many devices
- each has its own protocol

How can we avoid writing a slightly different OS for each H/W combination?

Write device driver for each device

Drivers are **70%** of Linux source code

File systems

Overview

- A filesystem is an organized collection of files and directories
- The Linux kernel maintains a single hierarchical directory structure to organize all files in the system
  - Not like Windows where each drive (C, D, E, etc) has its own hierarchy
- Root directory is named /
  - Pronounced “slash”

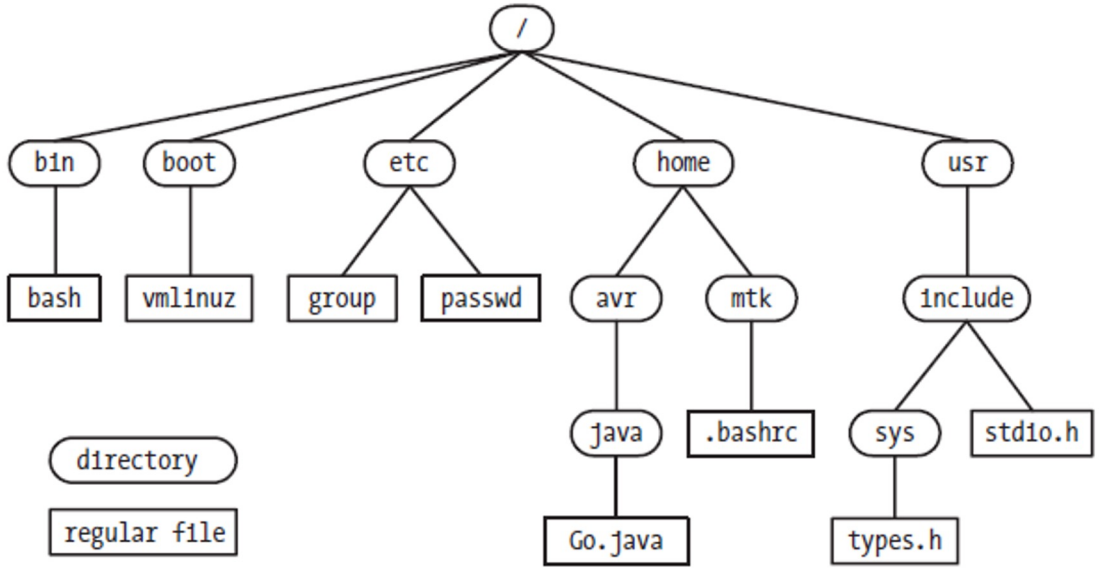


Figure 2-1: Subset of the Linux single directory hierarchy

*Figure from *The Linux Programming Interface* by Michael Kerrisk

Filenames and Pathnames

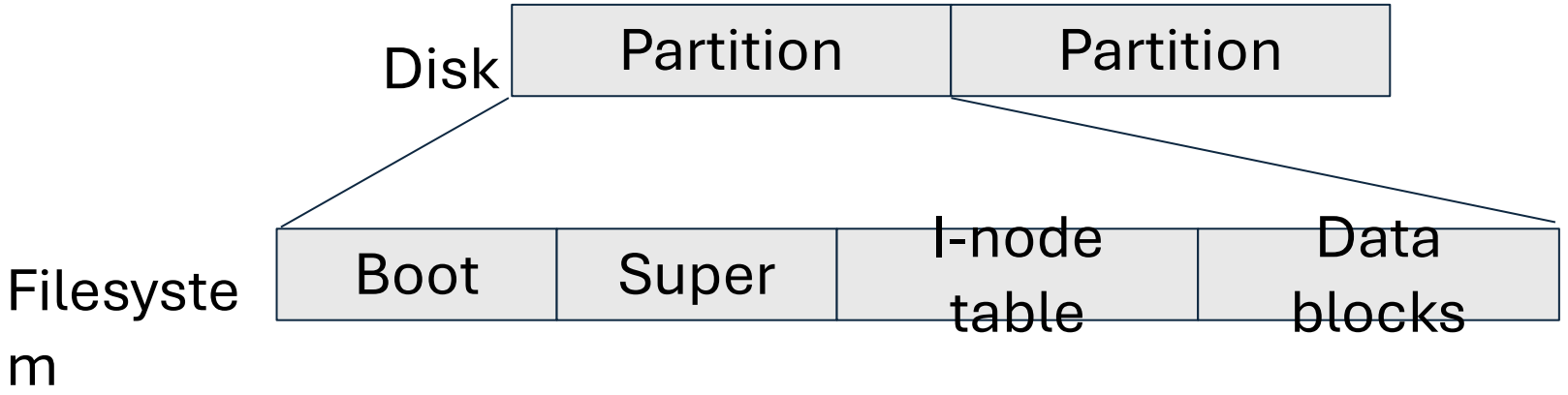
- Linux filesystems allow filenames:
  - Up to 255 characters
  - To contain any characters except slashes (/) and null characters (\0)
  - Recommended to only use letters, digits, . (period), _ (underscore), and - (hyphen)
- A pathname is a string with an optional / followed by a series of filenames separated by slashes
  - Absolute pathname: starts with a /
    - Examples: /home/student/ (student's home directory), /home/student/hw.txt (student's homework file)
  - Relative pathname: relative to the current directory
    - Examples: assignment-8/build (assignment-8 is a subdirectory of my current directory)

File Permissions

- Each file has a user ID and group ID that defines the owner and group
  - Ownership determines file access rights to users of the file
- Three categories of users:
  - Owner
  - Group: users who are members of the group
  - Other: everyone else
- There are three permission bits for each category: read, write, execute
  - Examples:
    - 111 110 000 (760): owner can read, write, and execute; group can read and write; everyone else cannot access the file
    - 110 000 000 (600): owner can read and write; everyone else cannot access
    - Look at the permission bits on ~/.ssh/id_rsa -- what is the reason for this?

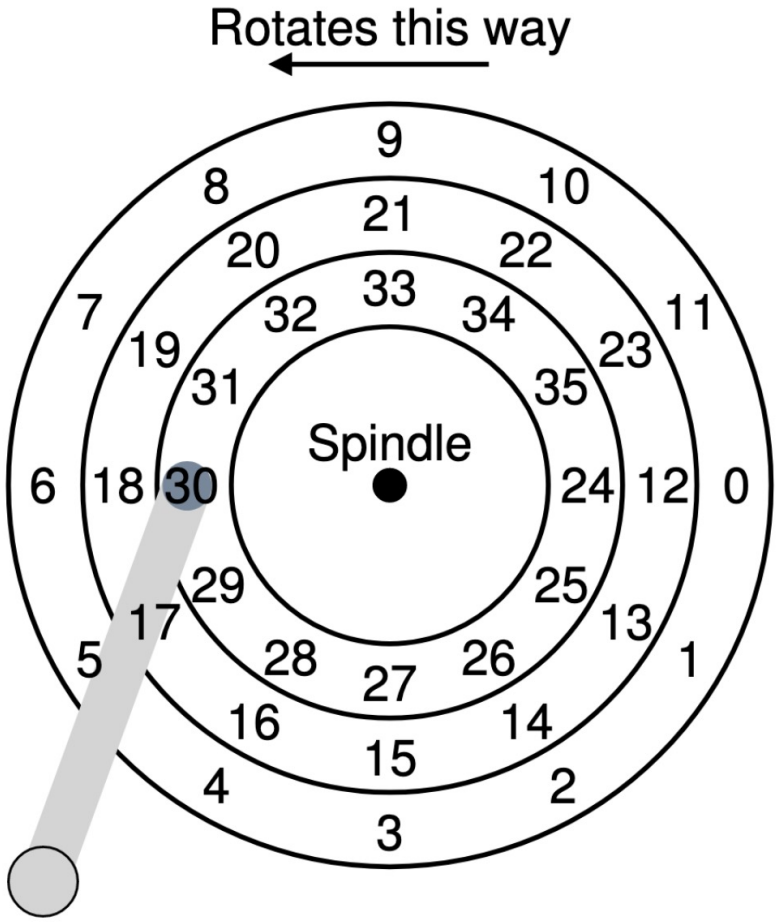
Back to Filesystems

- A disk drive is divided into circles called tracks
  - Tracks are divided into sectors
    - Sectors are a series of physical blocks
      - Physical block: the smallest unit a disk can read or write
        - Usually 512 bytes (older disks) or 4096 bytes (newer disks)
- Each disk is divided into partitions
  - Each is a separate device under /dev
  - A partition holds either
    - *Filesystem*
    - Data area (raw-mode device)
    - Swap (for virtual memory)



Spinning Disks

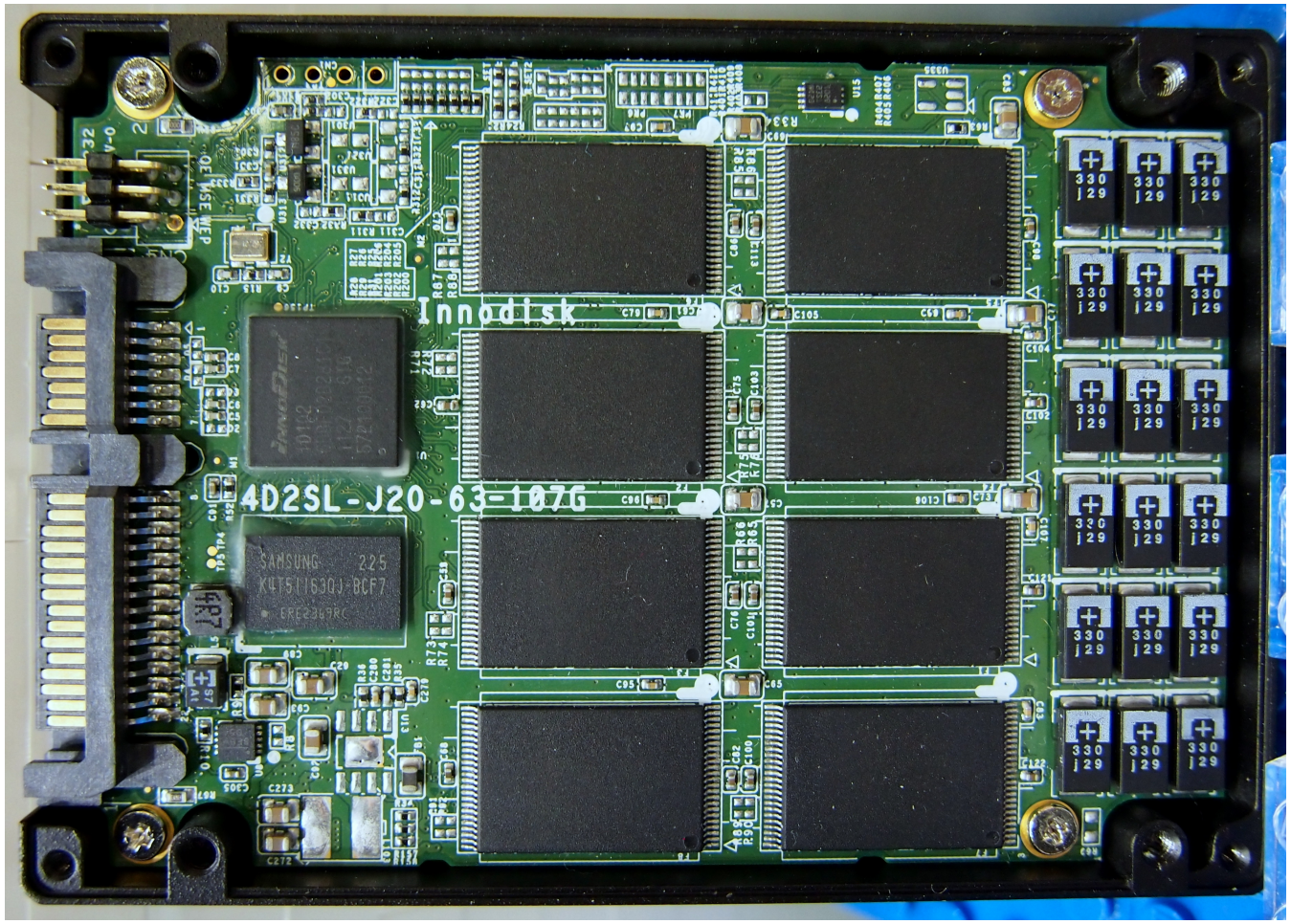
- Disks are composed of platters that store data
- Platters spin around a spindle
  - 7,200 RPM for consumer drives
  - 15,000 RPM for commercial drives
- Reading and writing is done by a disk head
- Head controlled by disk arm
- Disk geometry and access patterns affect performance



OSTEP Fig. 37.3

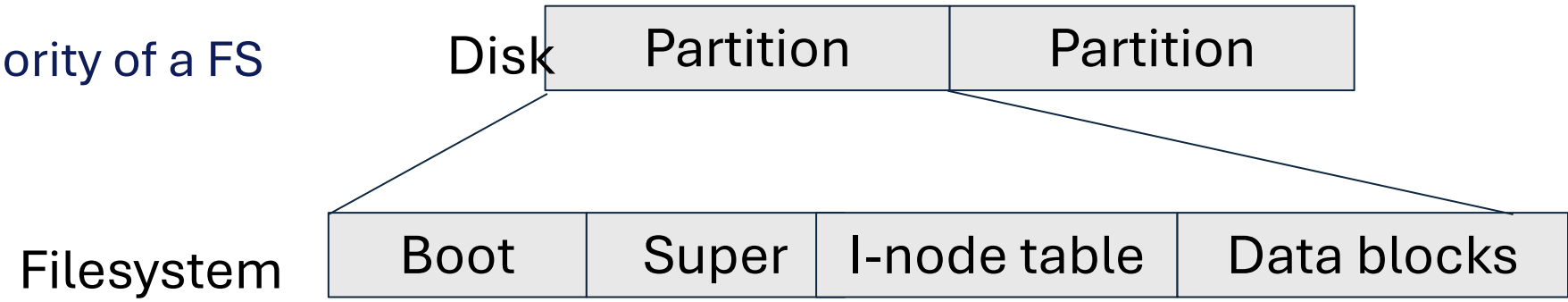
Solid-State Drives

- Many devices today do not have spinning disks and instead have solid-state drives (SSDs)
- No arm to seek or platter to spin
- Other filesystem considerations for SSDs
  - Reduce writes to preserve device lifespan



Filesystem Structure

- Many types of filesystems -- too many to list!
  - https://en.wikipedia.org/wiki/List_of_file_systems
- Boot block: always the first block in a FS
  - Not used by FS; contains info to boot the OS
  - Only one needed by OS
- Super block: contains parameter info about the filesystem
  - Size of the i-node table, size of logical blocks, size of the filesystem (in logical blocks)
- I-node table: contains one (unique) entry for every file in file system
  - Contains most of the “metadata” about individual file
- Data blocks: the (logical) blocks that contain the data for files and directories
  - This is the vast majority of a FS



I-Nodes

- Index nodes (i-nodes) contains the following metadata about a file
  - File type (for example, regular, char device, block device, directory, symbolic link)
  - Owner of the file
  - Group of the file
  - File access permissions (user, group, other)
  - Three timestamps:
    - Time of last access (ls -lu)
    - Time of last modification (default timestamp in ls -l)
    - Time of last status change (change to i-node info) (ls -lc)
  - Number of hard links to file
  - Size of the file (in bytes)
- Number of blocks allocated to file
  - Pointers to the data blocks

Directories

- A directory is stored in a filesystem in a similar way as a regular file, but
  - It is marked as a directory in its i-node
  - It's a file with a special organization: it's a table consisting of filenames and i-node numbers
- Example is on the right
- Note: the i-node doesn't have a filename!
  - Implication: you can have multiple links to the same file!

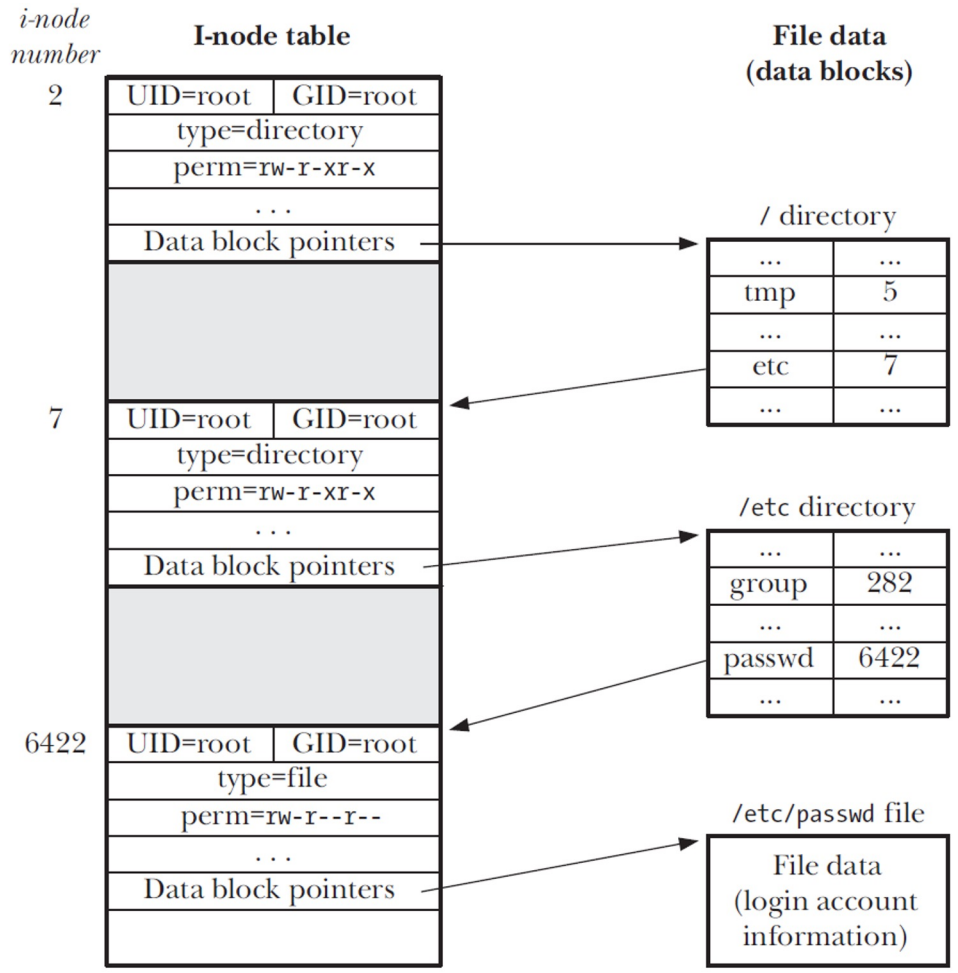
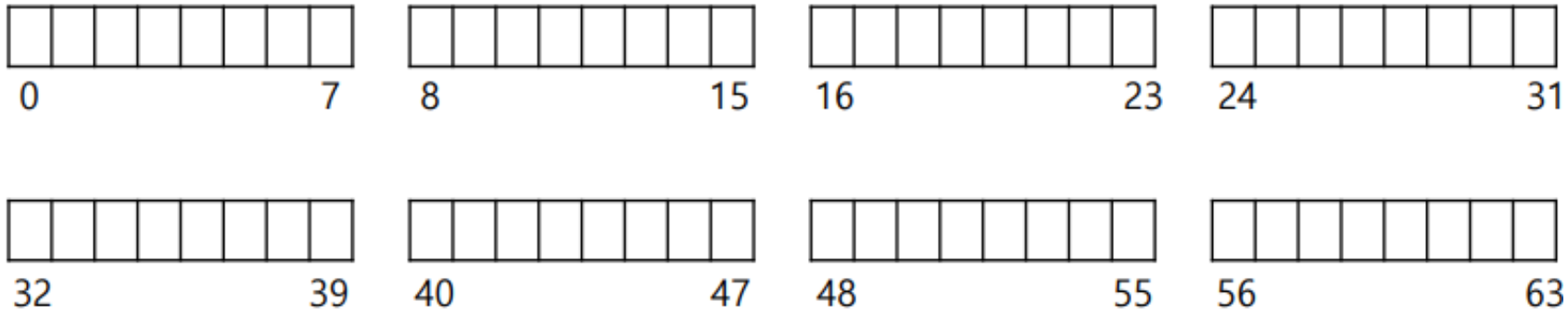


Figure 18-1: Relationship between i-node and directory structures for the file `/etc/passwd`

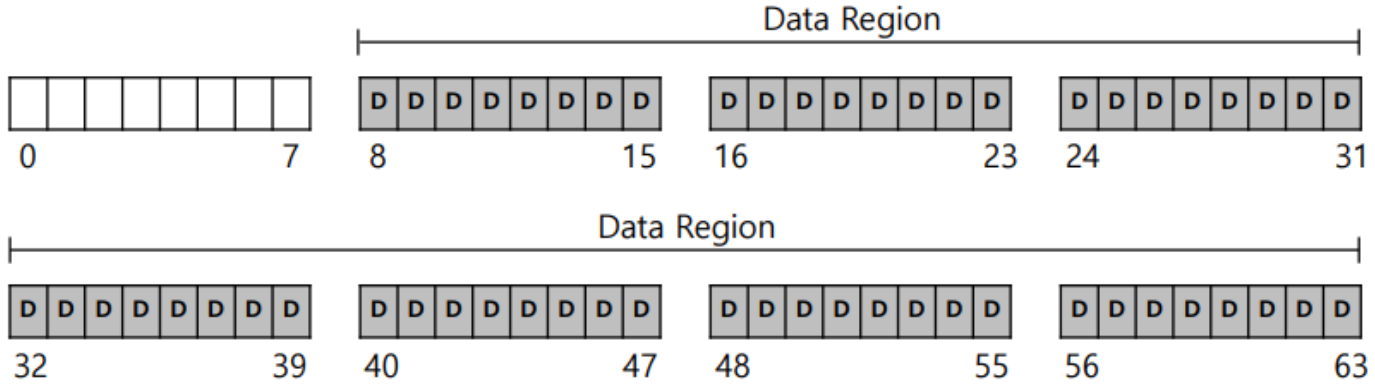
*Figure from *The Linux Programming Interface* by Michael Kerrisk

Very Simple File System (VSFS) Data Structures

- Divide disk into blocks
- Use one block size (4KB)
- Blocks are addressed from 0 to *N*-1 (N is the number of blocks)

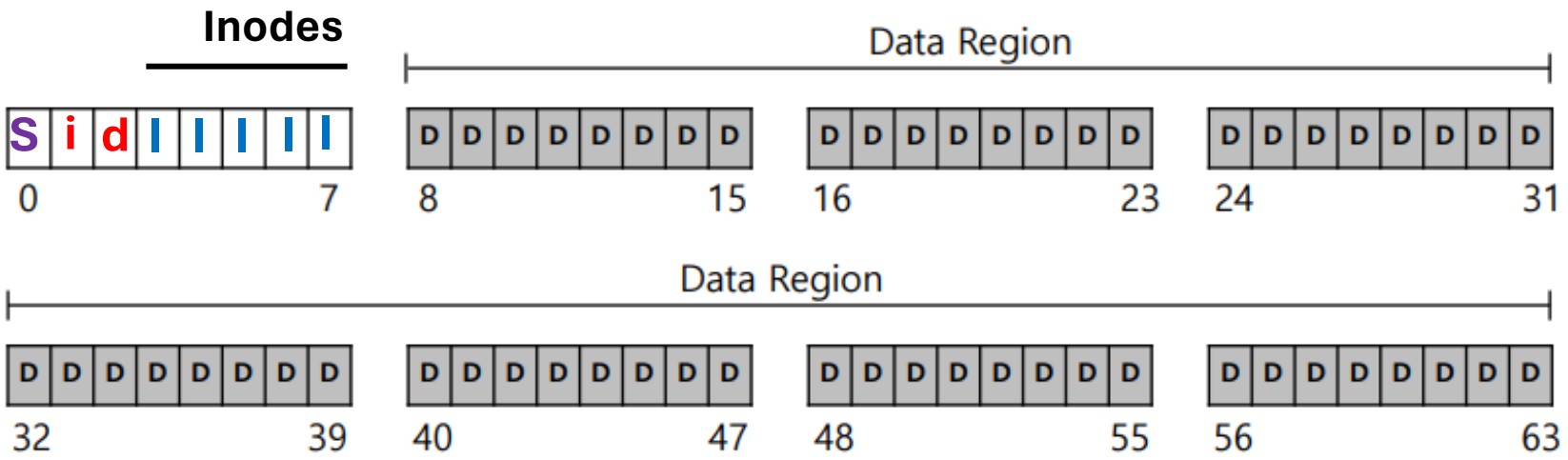


- Store user data in *data region* (e.g., files and directories)



VSFS Data Structures (cont.)

- File system tracks information (metadata) about files.
  - E.g., a subset of data blocks that form a file, file size, access rights
- Metadata is stored in a structure called *inode*
  - inode table is an array of inodes
  - 256-byte inode
  - 16 inodes per block; 80 inodes in 5 blocks
- Use bitmap for tracking free inodes and data blocks
  - A bit indicates whether an inode or a block is free.
- Superblock stores information about a certain file system.
  - E.g., number of inodes and data blocks, the start of inode table



Data Blocks

- How can files of very different sizes be supported?
  - One method: store pointers to the data blocks!
- Figure on the right shows how ext2 does this
  - Small files might fit entirely in direct pointers
- Bigger files use:
  - Indirect pointers
  - Double-indirect pointers
  - Triple-indirect pointers
- Advantages of pointers
  - Fixed-size i-node
    - But arbitrary size files
  - Store blocks non-contiguously

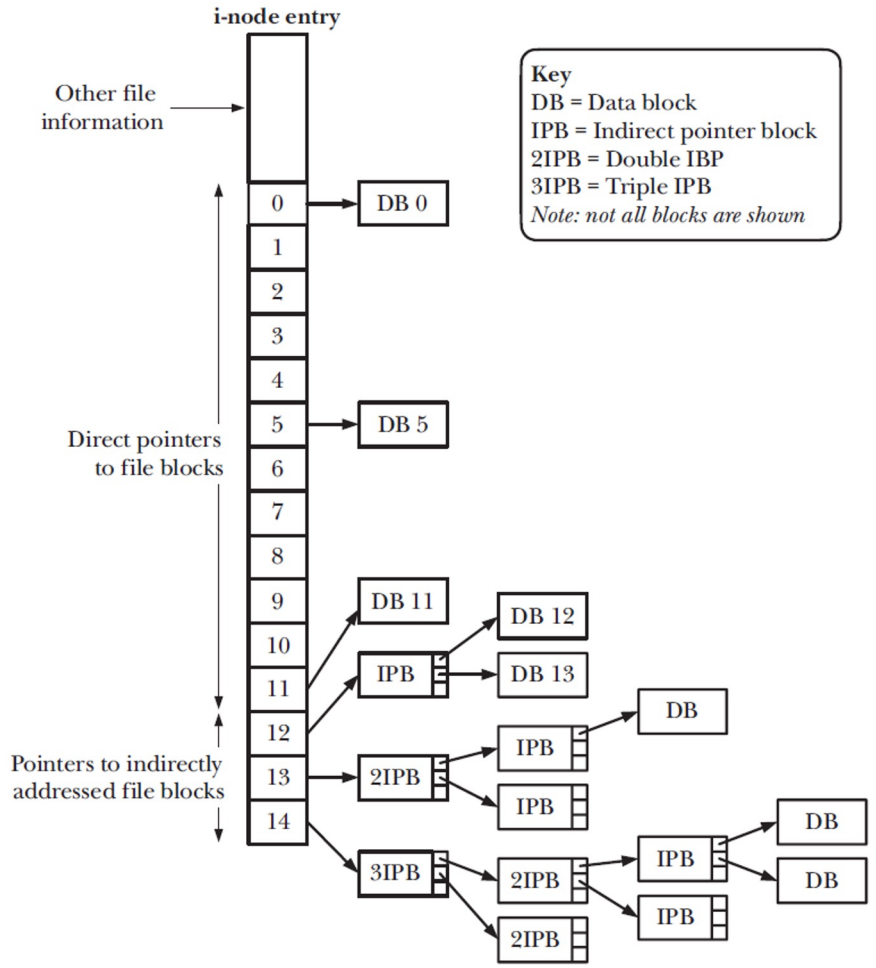
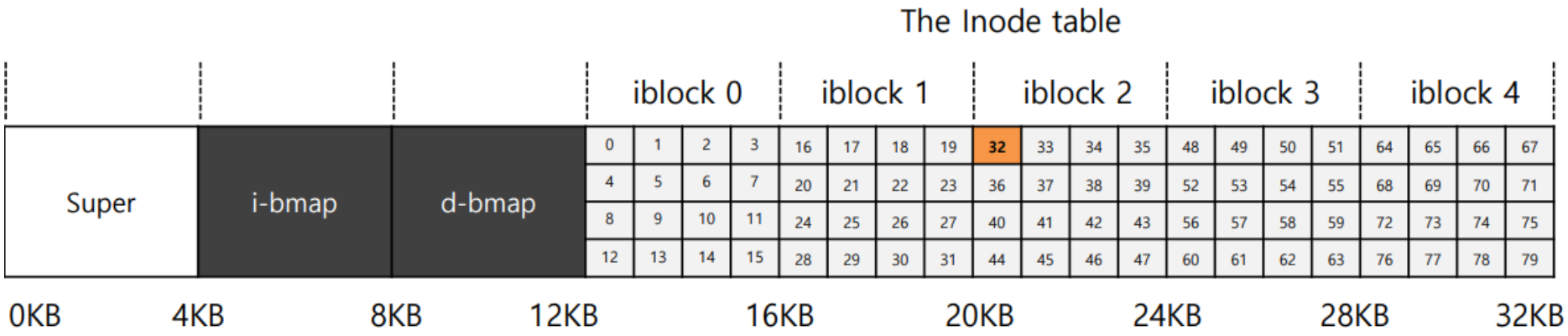


Figure 14-2: Structure of file blocks for a file in an *ext2* file system

*Figure from *The Linux Programming Interface* by Michael Kerrisk

The I-Node

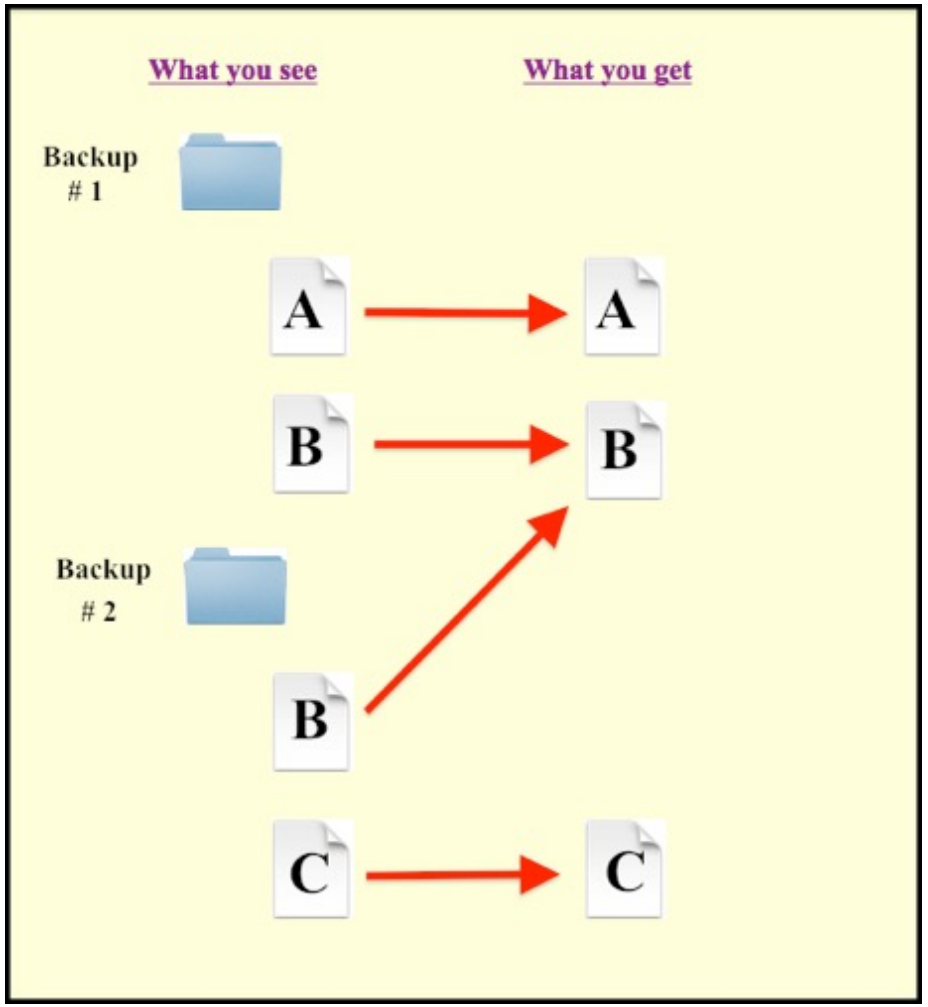
- Index node (inode): array of nodes is indexed
- Each inode is identified by an i-number
  - Used for indexing an array of inodes
- Find the byte address for the inode with i-number 32
  - Compute the offset into the inode table: 32 * sizeof(inode) = 8192 (8KB)
  - Add the offset to start address of inode table: 12KB + 8KB = 20KB
- inodes are fetched using *sectors* (a block consists of sectors)
  - 512-byte sectors
  - Sector number: (20 * 1024)/512 = 40



Hard Links

- Multiple filesystem files can be represented by a single file on disk
  - Hence inodes do not store the filename
- Inode stores the number of links
  - When links = 0, file can be deleted
- Hard links are used in Mac Time Machine
  - Create illusion of snapshots or copies of files
  - Only create new files on disk for changed files
  - Delete backup simply decrements link count

Mac Time Machine uses hard links

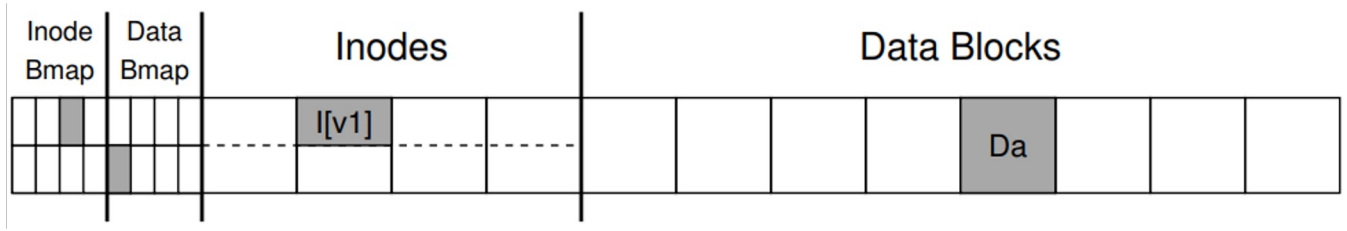


Source: http://oldtoad.net/pondini.org/TM/Works.html

Journaling

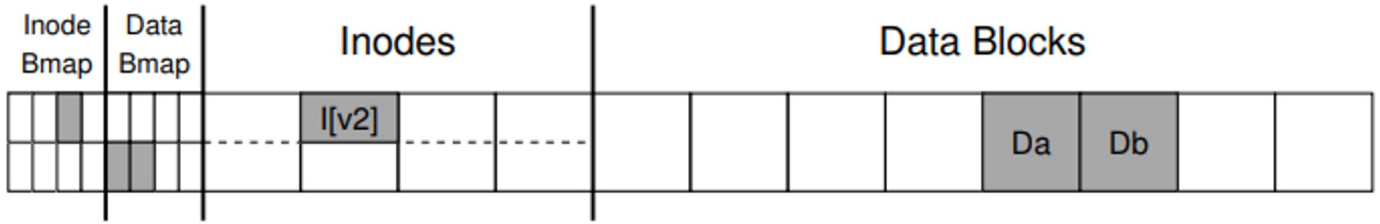
Crash Scenario

- Consider the following simple filesystem scenario from the textbook
  - One file with one allocated data block




```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

- Suppose we want to append to this file; we have to
  - (1) write the data to a new data block, (2) update the i-node, (3) update the data bitmap
    - This requires three (separate) writes to disk
  - For example, we want the updated filesystem to look like this:



Crash Scenario (cont.)

- But what if the system crashes after only 1 of the writes is performed?
  - **Just the data block is written**: the data is there, but the i-node doesn't know about it
  - **Just the i-node is updated**: the data isn't there, so we'll read garbage from disk
    - And the filesystem is inconsistent: the i-node points to a data block, but the data block bitmap thinks it hasn't been allocated.
  - **Just the data block bitmap is updated**: no i-node points to the data block, so there's a data leak

- What if the system crashes after 2 writes (out of 3) are performed?
  - **I-node and bitmap updated, data block not updated**: file contains garbage
  - **I-node and data block are written**: the bitmap thinks the block hasn't been allocated
  - **Bitmap and data block are written**: no i-node points to the data (so the data is effectively lost!)

A Solution: Filesystem Check

- A filesystem check (fsck) scans the filesystem for inconsistencies and repairs them
  - Runs before filesystem is made available to users
    - fsck:
      - Check the superblock; if it looks corrupted, you can try to use an alternate copy
      - Scan the i-nodes to see which blocks are in use and resolve inconsistencies in the allocation bitmaps
      - Check that each i-node is ok
      - Scan the entire directory tree (starting at root dir) and verify link counts
      - Check for bad blocks (e.g., data pointer points to an invalid address); just clear the bad pointers
      - Check that directories are consistent (e.g., contain “.” and “..”, there are i-nodes for each file entry)

Journaling

- A filesystem check can be very slow, especially for large filesystems
- An alternative is journaling, also called write-ahead logging
- Simple idea: before updating the on-disk structures, write information about the update to a *journal* (also called a log)
  - If there's a crash before the disk can be updated, the journal contains enough information to recover
    - You don't have to scan the entire disk!

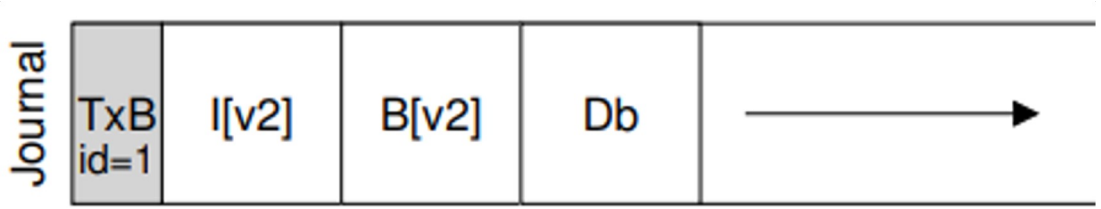
Example: ext2 vs. ext3

- The basic layout of ext2 (no journaling):

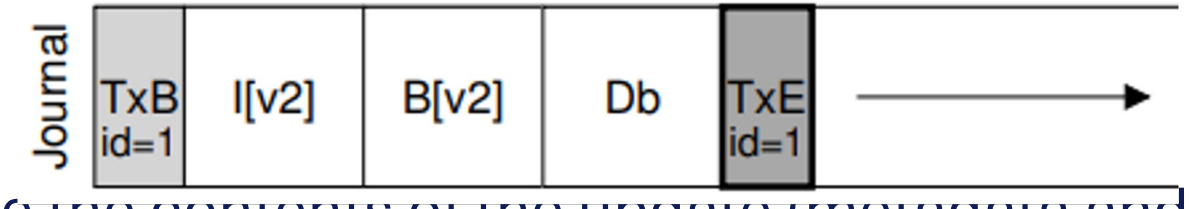
- The basic layout of ext3 (with journaling):

Journaling Phases

- Four basic phases:
  - Journal write: write contents of transaction to journal; wait for these to complete



  - Journal commit: write the transaction commit block; wait for write to complete; transaction is committed



  - Checkpoint: write the contents of the update (metadata and data) to disk
  - Free: mark the transaction as “free” in the journal by updating the journal superblock
    - Done at “some point” in the future

Recovery

- When a crash happens before *journal commit* completes, pending updates in a transaction are skipped.

- When a crash happens after *journal commit* completes and before *checkpoint* completes, transactions are replayed (redo logging)
  - File system finds out transactions that were committed successfully
  - The blocks of those specific transactions are written to their final disk locations again