## Error Handling

- I/O devices and physical media aren't perfect
- Device drivers have to cope with error conditions
- Recovery can be complex and device-dependent
- Often, errors are detected by adding check bytes or ECC bytes to physical blocks

1 / 37

## Example: Tape Errors

- Suppose a tape drive indicated a bad read
- First, retry up to 10 times
- Backspace the tape a few inches and "shoe shine" it — move it back and forth so the bad spot is scraped against the edge of the read head
- The idea is to dislodge any pieces of dirt

2 / 37

## Disk I/O Errors

- If a disk write fails, retry a few times
- If the write still fails, mark the block as "bad" — how this is done is device-dependent — and allocate a spare block
- Somehow, indicate that this remapping has taken place

3 / 37

## Testing for Errors

- How do you test your error recovery code?
- Where do you get media that are just bad enough?
- Remember that failure modes can be arbitarily weird
- Loose cables can generate very strange signals

4 / 37

## From the NetBSD Printer Driver

1. You should be able to write to and read back the same value to the data port. Do an alternating zeros, alternating ones, walking zero, and walking one test to check for stuck bits.
2. You should be able to write to and read back the same value to the control port lower 5 bits, the upper 3 bits are reserved per the IBM PC technical reference manauls and different boards do different things with them. Do an alternating zeros, alternating ones, walking zero, and walking one test to check for stuck bits.
3. ...

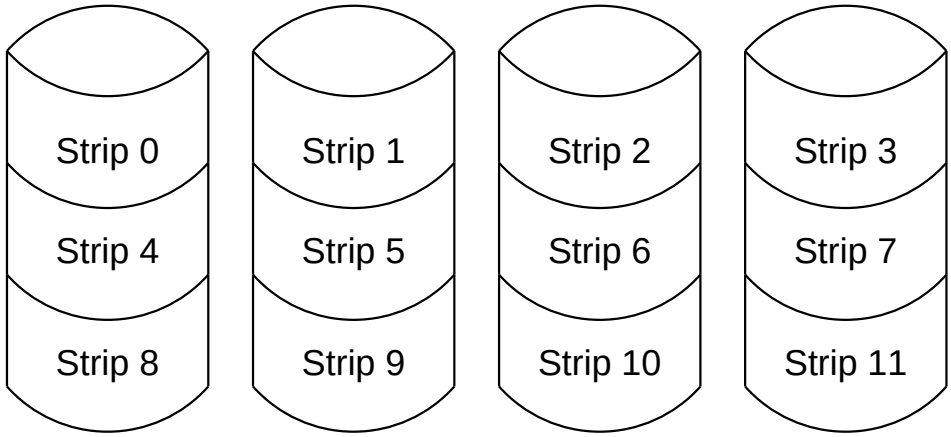
5 / 37

## RAID

- One way of coping with errors: redundancy
- Example: RAID (*Redundant Array of Inexpensive (or Independent) Disks*)
- Many configurations
- Some use extra disks for error recovery, some use them for better performance, some do both
- Can be implemented in controller or OS
- In OS, implemented as pseudo-device driver layerd on top of real devices

6 / 37

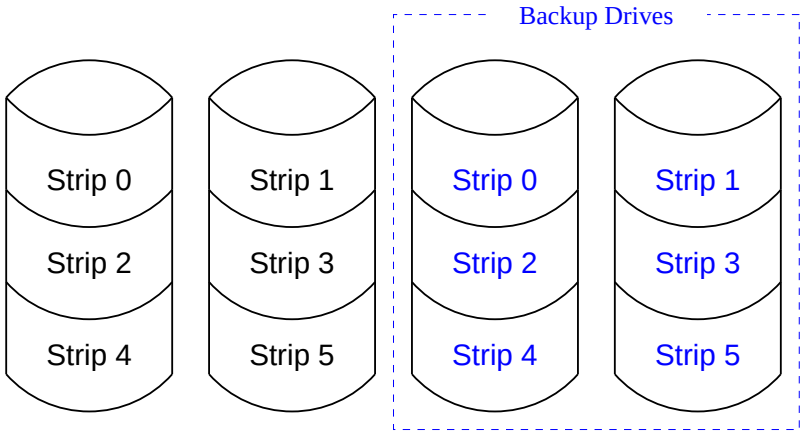
## RAID 0




- Put *strips* of sectors on successive disks — called *striping*
- Improves throughput — can be doing I/O from multiple drives simultaneously to satisfy a single (large) request
- Parallelism improves with more drives
- But — hurts reliability

7 / 37

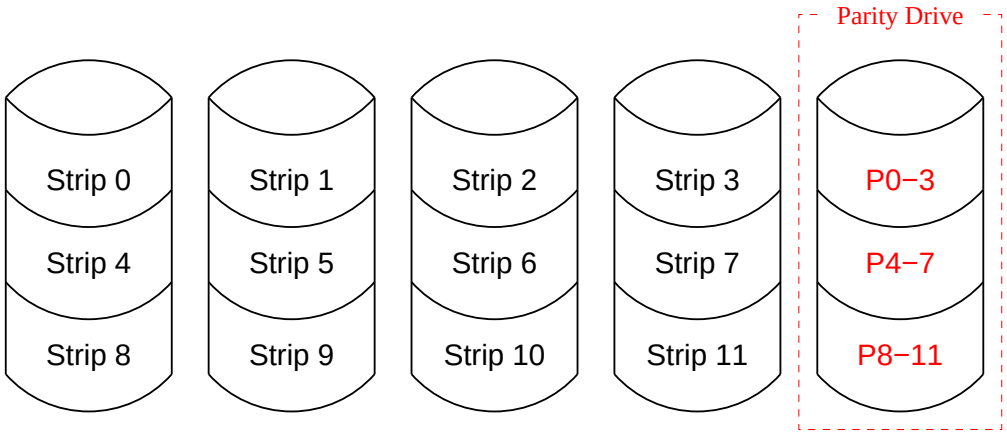
## RAID 1




- Replicate each sector; write both copies
- Improves read throughput; hurts write throughput
- If a drive fails, ignore it and use the other one
- Rebuild the failed drive in the background

8 / 37

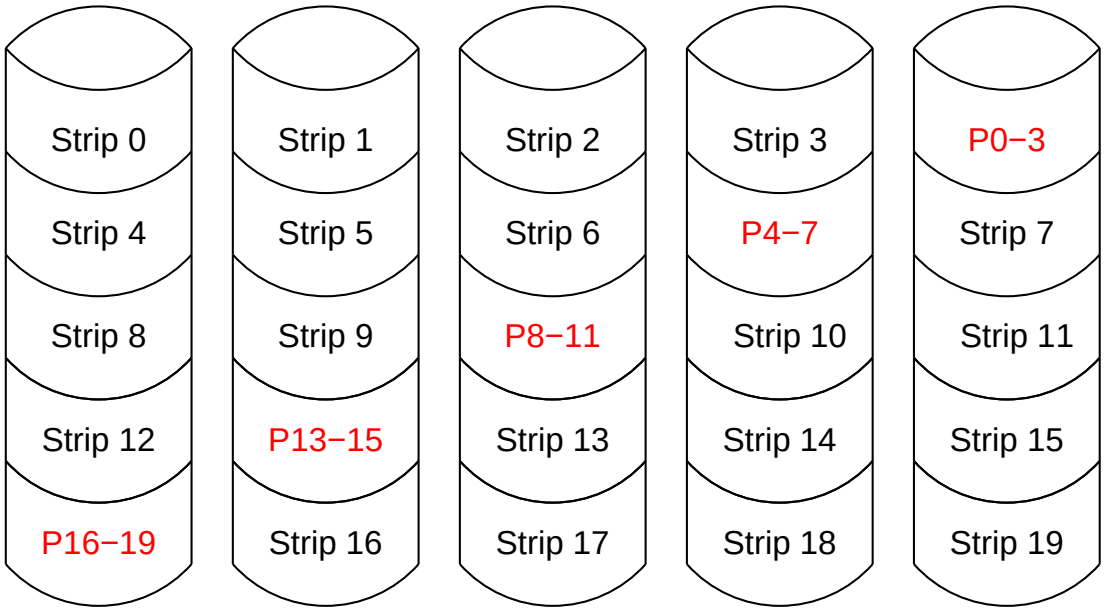
## RAID 4




- The last drive is the *parity* drive — it has the XOR of all of the other drives
- If a drive crashes, reconstruct its contents from all of the other drives, including the parity drive
- Poor performance — must read old data to recompute parity if a block changes
- Heavy load on parity drive

9 / 37

## RAID 5




- Similar to RAID 4, but the parity strips are spread out
- Avoids bottleneck
- But — rebuilding after a drive failure is complex

10 / 37

## User-Level Daemons

- How is a replacement RAID drive reconstructed?
- Not everything is best done in the kernel
- It could be done in the kernel — but should it be?

11 / 37

## What's in the Kernel?

- What are the criteria for putting something in the kernel
- On the other hand — why should we avoid putting things in the kernel?
- Different operating systems make different choices

12 / 37

## In the Kernel

- Something that's in the kernel can't be changed by a user or a site
- Kernel bugs frequently crash the system
- There's no intra-kernel memory protection
- But — kernel-to-kernel operations are faster than kernel-to-user; there's no context switch involved
- For a user program to talk to a (privileged) user level program involves copying the data to and from the kernel; this can be expensive

13 / 37

## Criteria

- Put something in the kernel if there *should* be only one way of doing it (i.e., loading the virtual memory registers)
- Put something in the kernel if it's needed before the system is really running (i.e., the disk I/O driver)
- Put something in the kernel if it needs frequent access to kernel-only resources (i.e., the page frame reclamation algorithm)
- Basically, put something in the kernel if and only if there's no choice

14 / 37

## Avoiding the Kernel

- Put it in user space if possible
- If there's no significant performance issue, put it in userspace
- If there is a performance hit but it's rarely done, put it in user space
- If you may want to change its behavior, put it in userspace
- If you can stop and restart it, put it in userspace
- If it uses files for configuration, put it in userspace

15 / 37

## The Bottom Line

- If flexibility is more important, put it in userspace
- If performance is more important, put it in the kernel
- Put policy in userspace; put mechanism in the kernel

16 / 37

### Kernel Interfaces

- If we're going to put things in userland, we need proper interfaces
- Linux relies on pseudo-file systems — `/proc`, `/sys`, and more — for many of its interfaces
- Not a bad idea — everything appears to be a file, and file permissions can be used to restrict access

17 / 37

## Daemons

18 / 37

### Daemons

- *Daemon*: "A program that runs unattended to perform continuous or periodic systemwide functions"
- Name originally used in Multics; some say it came from Maxwell's Demon, but `multicians.org` says it's from the Greek word $\delta\alpha\iota\mu\omega\nu$: "A supernatural being of a nature intermediate between that of gods and men"
- Either way — daemons perform system functions, but are not part of the operating system

18 / 37

## Example: The Line Printer Daemon

- We can't let user programs talk directly to the printer
- Instead, we let them talk to the printer daemon; it talks to the printer

19 / 37

## Wny Can't User Programs Talk to the Printer?

- Speed — the printer is slower than programs
- Conflict — more than one program may need to print simultaneously
- Accounting — many sites charge users for pages printed
- Security — prevent interference with another job
- Labeling — make sure output is properly labeled
- Indirection — multiple printers, different output formats, etc

20 / 37

## The Line Printer Daemon

- Several pieces — user-facing, printer-facing, administrative
- User-facing: invoked by user with files or output to print
- Printer-facing: picks up requests submitted by users; copies them to printer
- Administrative: group printers, set permissions, set up or down, describe output format, etc.

21 / 37

## Communications

- How do the different pieces talk?
- Pipes? Files? Signals? Network connections? Something else?
- Any and all of the above!
- But — must be careful about security?

22 / 37

## Security and the Printer Daemon

- Users must not be able to overwrite printer daemon files
- Easy — use ordinary Unix file permissions
- Make user-facing program setuid
- Files to be printed must be readable by the user (for security) and by the printer daemon (for efficiency)
- If not readable by the daemon, copy the file; if not readable by the user, scream



## Linking versus Copying

- It's tempting to create a link to the file the user wants printed
- Very difficult — does the user have read permission on it?
- Watch out for race conditions

## How to Link

- With user's permissions, determine the current directory and rewrite relative paths as absolute paths
- With daemon's permissions, change to a world-writable directory underneath a protected directory
- With user's permissions, create a hard link to the file in this world-writable directory
- With user's permissions, verify that it can be opened
- Don't record any decisions on size, formatting, etc., now, because the user still has a link and ownership, and can change that stuff

25 / 37

## Special Code and Pseudo-Devices

- On Unix, the printer daemon is easy because the printer, too, is just a file.
- Programs that expect to talk to the printer directly can generally talk to the daemon via a pipe, or write to a file, with no changes
- On other operating systems, applications need special printer interface code
- This can be good, because it lets the programs know what resolution output to generate
- On some systems, *pseudo-devices* — software that looks like a hardware printer — must be used

26 / 37

## Daemons and Permissions

- What permissions do daemons run with?
- Some run as root, especially those that need to permit logins (i.e., some network daemons)
- Better to use dedicated login: `lpd`, `games`, `named`, etc.
- On Multics, most daemons ran in group `sysdaemon` or `netdaemon`
- Principle of Least Privilege

27 / 37

## NFS

- NFS (Network File System) uses user- and kernel-mode pieces
- The actual disk I/O is in the kernel, for performance reasons
- But file locking is done in a user-level daemon. Why?

28 / 37

## NFS Lock Daemon

- NFS, by design, is stateless; clients and servers can be rebooted freely
- Also helps guard against network outages
- But locking is inherently stateful
- Lock daemon keeps track of file locks; stores them on lock disk, for crash recovery

29 / 37

## Other Vital Commands

- Login and X
- Mount — make file systems available
- Fsck — repair file systems after a crash
- Halt and reboot
- Shells
- Init

30 / 37

## Login

- Runs as root, but is not setuid; always invoked as root
- Accepts login name and password
- Sets up user environment

31 / 37

## The X Window System

- On Unix, X Windows is not a part of the kernel
- The *only* thing special about X is that it has permission to open the graphics card
- Admittedly, that often requires root privileges
- Frequently has a login component: xdm, gdm, etc.

32 / 37

## Shells

- No required user shell
- Example: CS dept machines have at least seven different real shells in use
- Can have special-purpose shells for restricted logins
- But — system administration and startup require `/bin/sh`
- For (complex) security reasons, sometimes need a list of "official" general-purpose shells

33 / 37

# /sbin/init

34 / 37

## /sbin/init

- Process 1 — the beginning and end of all processes
- Started by the kernel at boot time — a hand-crafted process table entry
- Initiates system setup

34 / 37

## System Setup

- `init` reads a configuration file describing *run levels* and what to do when entering them
- One pseudo-run level is `sysinit`: system initialization
- Sysinit: sets host names, mounts /proc and /sys, initializes certain hardware, runs fsck

35 / 37

## Entering Run Levels

- Different run-levels select different system services
- "1" is single-user, "2" is multi-user without NFS, "3" is everything, "5" is X11, "6" is reboot
- `init` invokes `/etc/rc` to change run levels; it invokes all files in `/etc/rc`*N*`.d` to enter run level *N*

36 / 37

## /etc/rc*N*.d

- Contains `K` and `S` scripts
- `S` starts a service; `K` kills one
- Executed in numerical order
- Examples: K50nfs, S80sendmail