6.5 Flash Memory

- Flash Technology
- Flash-Aware File Systems
- Augmenting Disk Storage

Beyond Disks: Flash

- Pro
  - Flash block ≈ file-system block
  - Random access
    - no seek, no rotational latency
  - Low power
  - Vibration-resistant
- Con
  - Limited lifetime (compared to disks)
  - Write is expensive
  - Cost more than disks
    - 128GB SSD: ~$300
    - 1TB disk: ~$60

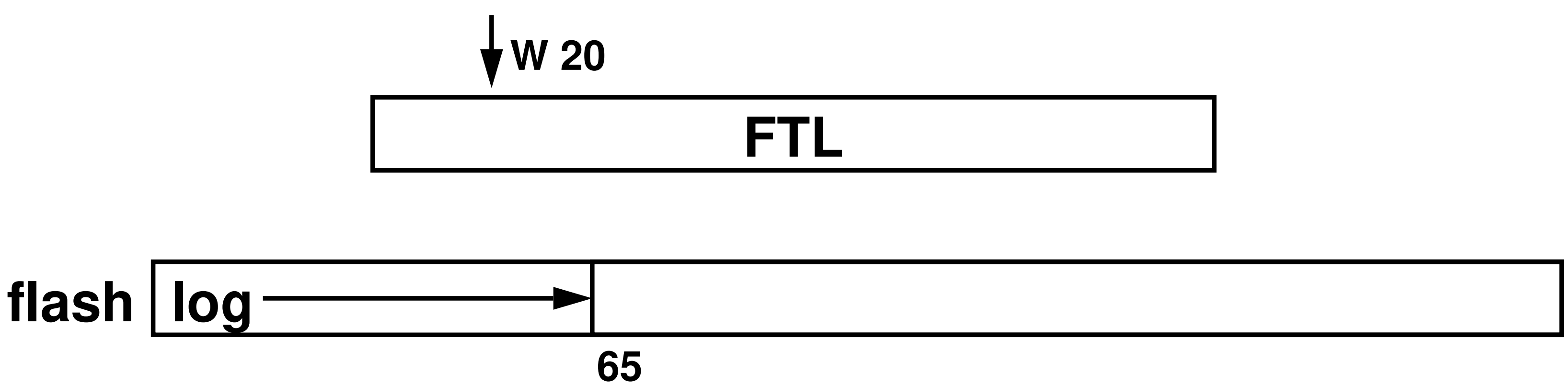
Flash Memory

- Two technologies
  - NOR
    - byte addressable
  - NAND
    - page addressable (about 1-4KB per page and 512KB per block)
    - cheaper
      - suitable for file systems use
    - limit on P/E (program/erase) cycle, about 10,000

- Writing
  - newly "erased" block contains all 1’s
  - "programming" changes some 1’s to 0’s
    - per byte in NOR; per page in NAND (multiple pages/block)
    - to change zeroes to ones, must erase entire block
    - can erase no more than ~100k times/block

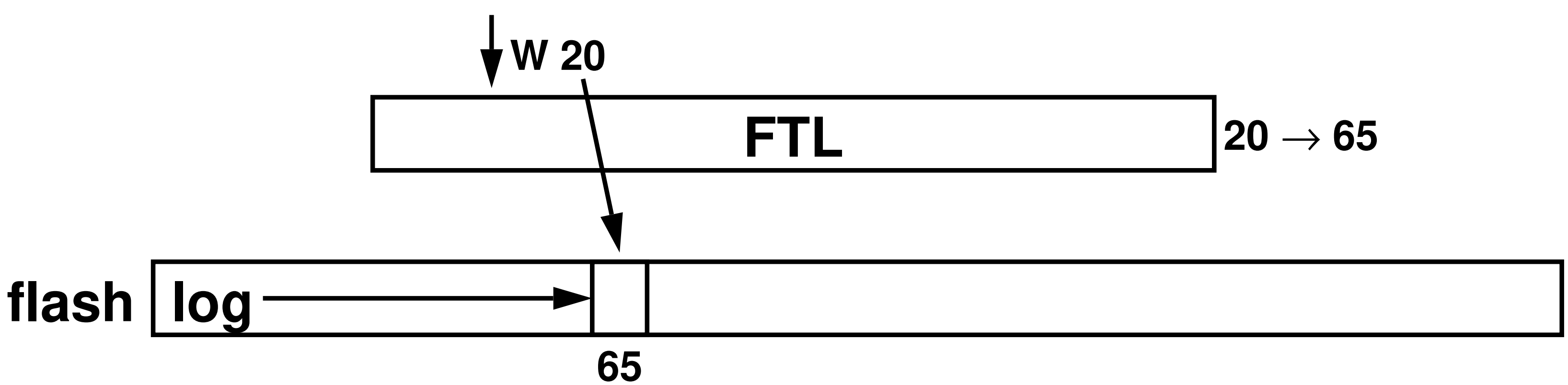
Coping

- *Wear leveling*
  - spread writes (erasures) across entire drive
  - approaches:
    - *flash translation layer (FTL)*
    - *log-structured file system*
      - blocks on the flash drive are used sequentially
- FTL: Flash translation layer (on a separate device, in memory, etc.)
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



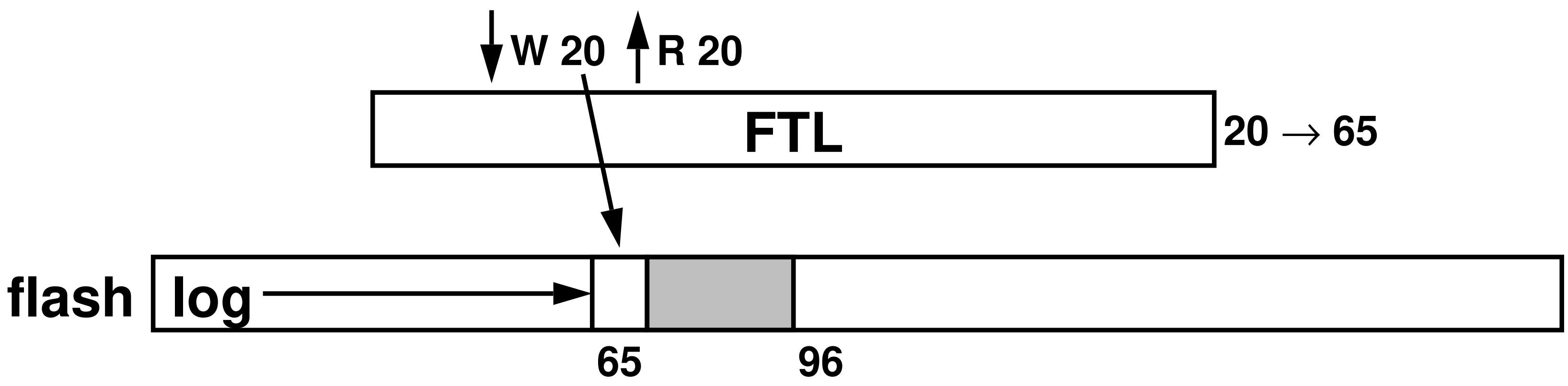
Coping

- *Wear leveling*
  - spread writes (erasures) across entire drive
  - approaches:
    - *flash translation layer (FTL)*
    - *log-structured file system*
      - blocks on the flash drive are used sequentially
- FTL: Flash translation layer (on a separate device, in memory, etc.)
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



Coping

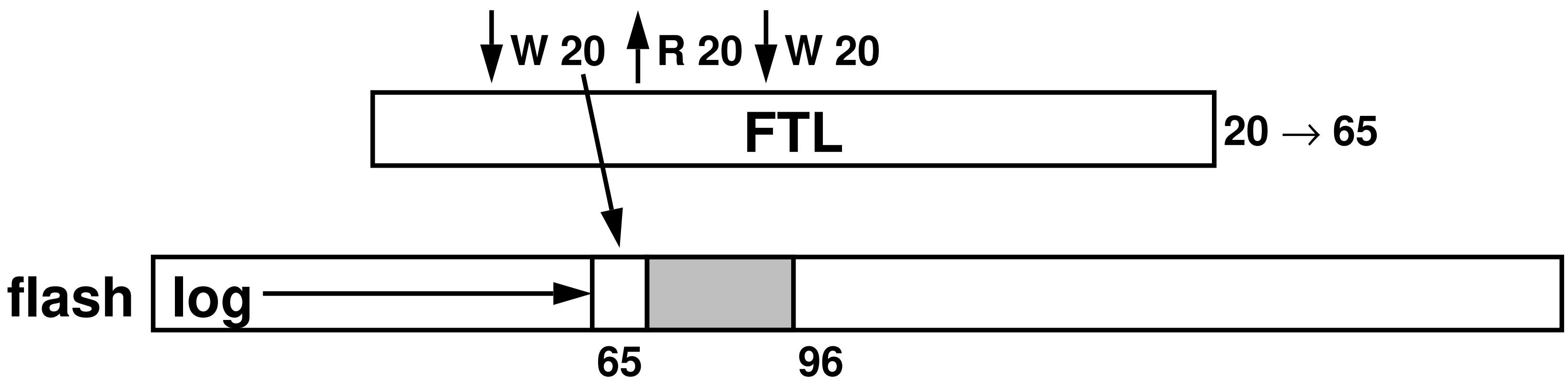
- *Wear leveling*
  - spread writes (erasures) across entire drive
  - approaches:
    - *flash translation layer (FTL)*
    - *log-structured file system*
      - blocks on the flash drive are used sequentially
- FTL: Flash translation layer (on a separate device, in memory, etc.)
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



Coping

- *Wear leveling*
  - spread writes (erasures) across entire drive
  - approaches:
    - *flash translation layer (FTL)*
    - *log-structured file system*
      - blocks on the flash drive are used sequentially

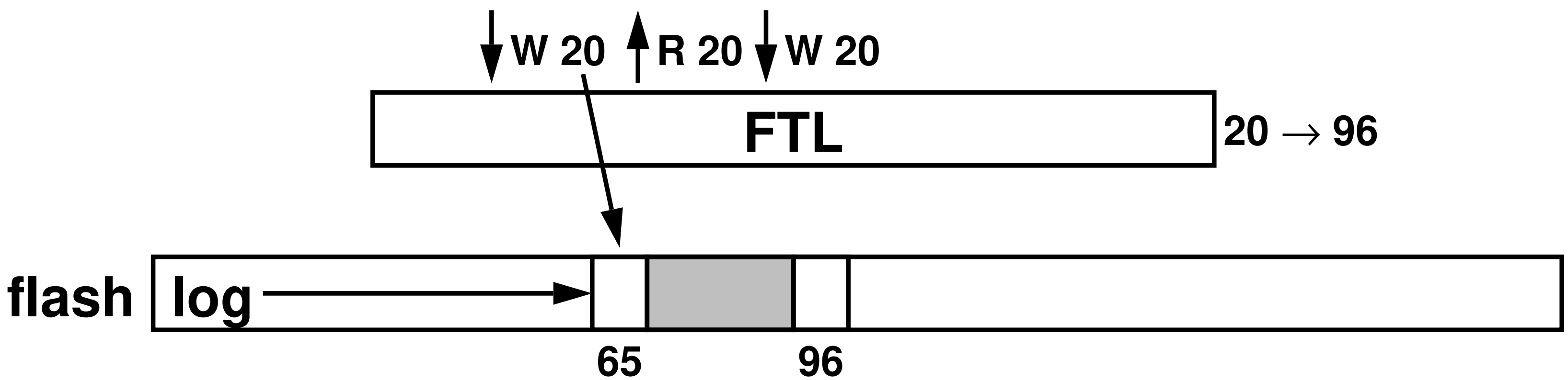
- FTL: Flash translation layer (on a separate device, in memory, etc.)
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



Coping

- *Wear leveling*
  - spread writes (erasures) across entire drive
  - approaches:
    - *flash translation layer (FTL)*
    - *log-structured file system*
      - blocks on the flash drive are used sequentially

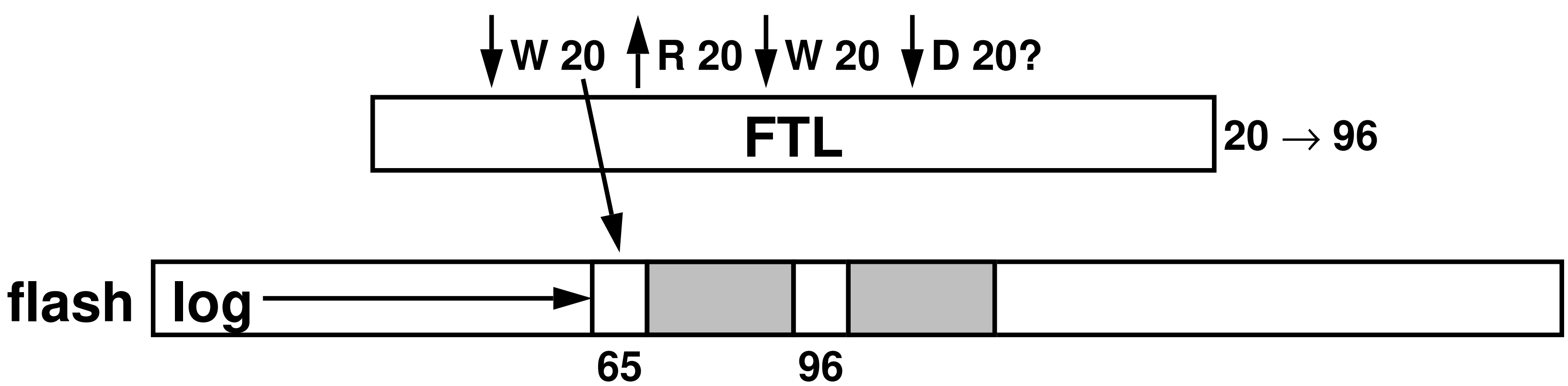
- FTL: Flash translation layer (on a separate device, in memory, etc.)
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



Coping

- ***Wear leveling***
  - spread writes (erasures) across entire drive
  - approaches:
    - ***flash translation layer (FTL)***
    - ***log-structured file system***
      - blocks on the flash drive are used sequentially

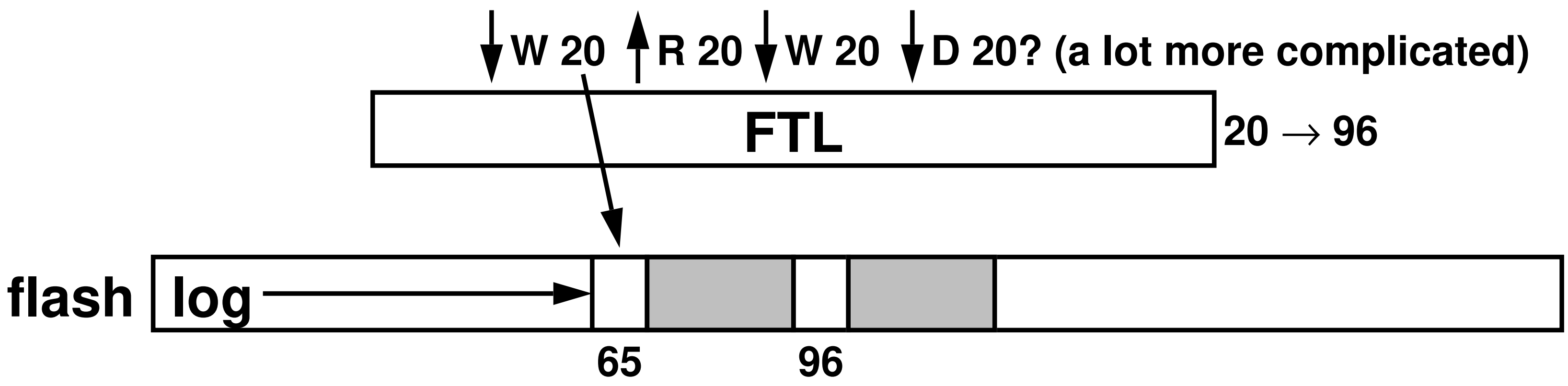
- **FTL: Flash translation layer (on a separate device, in memory, etc.)**
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



Coping

- *Wear leveling*
  - spread writes (erasures) across entire drive
  - approaches:
    - *flash translation layer (FTL)*
    - *log-structured file system*
      - blocks on the flash drive are used sequentially

- FTL: Flash translation layer (on a separate device, in memory, etc.)
  - specification from 1994
  - provides disk-like block interface (firmware on device controller)
  - maps disk blocks to flash blocks
    - mapping changed dynamically to effect wear-leveling



Flash with FTL

- Which file system?
  - FAT32 (sort of like S5FS, but from Microsoft)
  - NTFS
  - FFS
  - Ext3
- All were designed to exploit disks
  - much of what they do are irrelevant for flash

Flash without FTL

- Known as memory technology device (MTD)
  - software wear-leveling
  - perhaps other tricks

JFFS and JFFS2

- Journaling flash file system
  - log-based: no journal!
    - each log entry contains inode info and some data
    - garbage collection copies info out of partially obsoleted blocks, allowing block to be erased
    - complete index of inodes (i.e., meta-data) kept in RAM
      - entire file system must be read when mounted

UBI / UBIFS

- UBI (unsorted block images)
  - supports multiple logical volumes on one flash device
  - performs wear-leveling across entire device
  - handles bad blocks

- UBIFS
  - file system layered on UBI
  - it really has a journal (originally called JFFS3)
  - file map kept in flash as B+ tree
  - no need to scan entire file system when mounted

Flash as Part of the Hierarchy

- Flash as log device
  - aggregate write throughput sufficient, but latency is bad
  - augment with DRAM and a "super-capacitor"
- Flash as cache
  - large level-2 cache
    - integrated into ZFS
    - can use cheaper (slower) disks with no loss of performance
      - reduced power consumption

6.6 Case Studies

- FFS
- Ext3
- Reiser FS
- NTFS
- WAFL
- ZFS

Linux

- Linux had 57 file systems built for it to date!
- FFS
  - ext2
  - ext3 (journaling - crash resiliency)
    - ReiserFS (B-tree everywhere)
  - ext4
    - extents (optimize read/write)
    - LVM
    - hash trees for directories
  - BtrFS (Oracle)

Windows NT

- **NTFS**
  - extents (optimize read/write)
  - B-trees (optimize directory lookup)
  - journaling (crash resiliency)

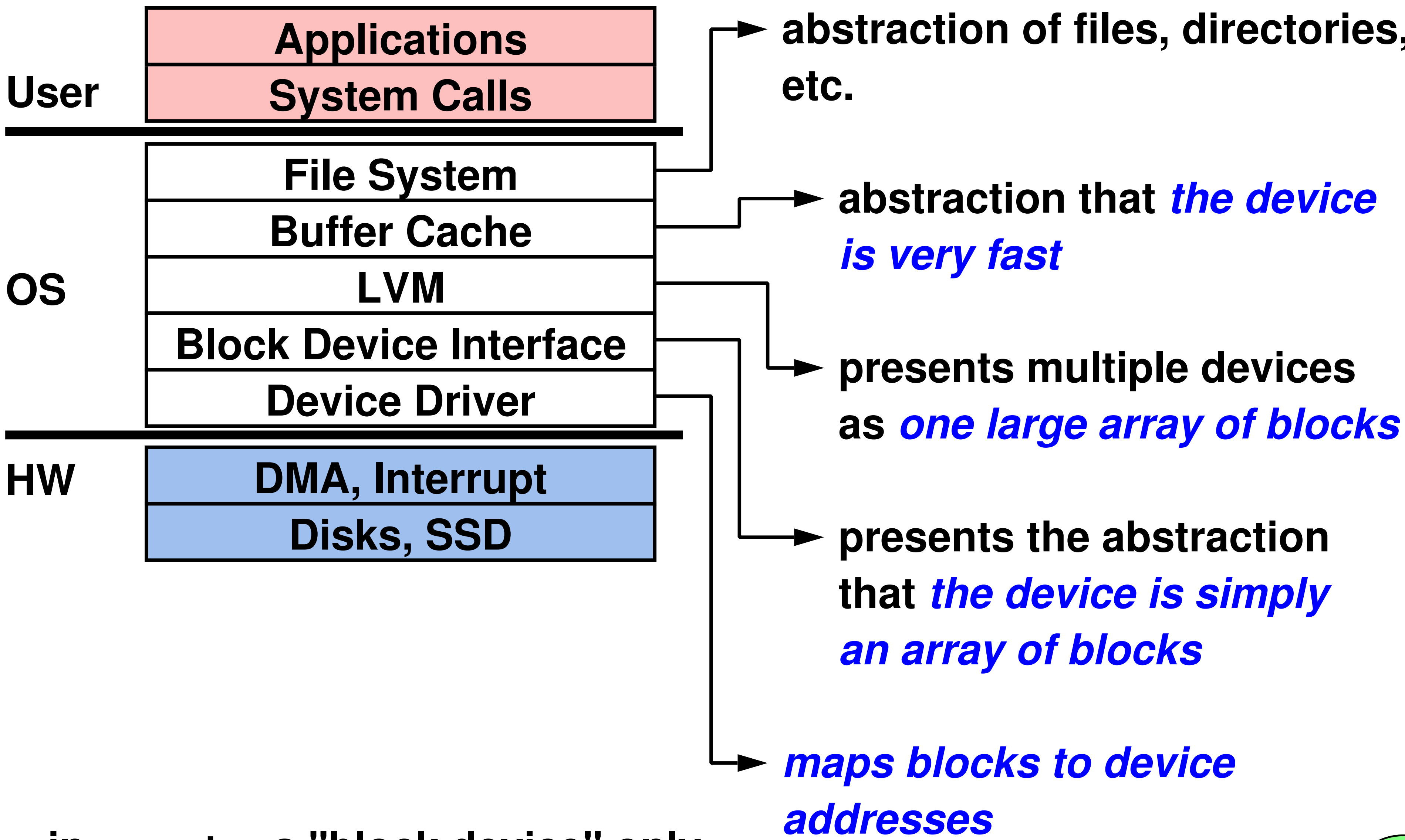
Mac OS X

- **Mac OS X**
  - HFS+ (planned to use ZFS but dropped the idea)
    - extents (optimize read/write)
    - B*-trees (optimize directory lookup)
    - journaling (crash resiliency)

Journaling

- Why did everyone choose journaling and not shadow pages?
  - journaling can be added to any existing file system

File System Summary



- in **weenix**, a "block device" only supports `read_block()` and `write_block()`

4.1 A Simple System (Monolithic Kernel)

- A Framework for Devices
- *Low-level Kernel*
- Processes & Threads
- Storage Management

Low-Level Kernel

- Let's talk about how devices are handled, starting at the lowest levels of the kernel
  - (although *bottom-up* is not a good way to *design* an OS)
    - but it may be a reasonable way to implement OS components
- We will start by looking at two such devices
  - terminals
  - network communication

Terminals

- VT100
  - type "`echo $TERM`" on Unix/Linux

Terminals

- Long obsolete, but still relevant
  - on Linux, you would probably use a "terminal" program (such as `xterm` or `gnome-terminal`) to interact with the system
  - on Windows, `putty` or `xwin-32` brings up a "terminal" for you to interact with a remote system
    - `ssh` client program interact with `sshd` on a server
      - once authenticated, `sshd` forks to exec `tcsh`/`bash`
    - you login session is *on the target machine*
      - i.e., if you login as root and type "`halt`", you would halt the machine!

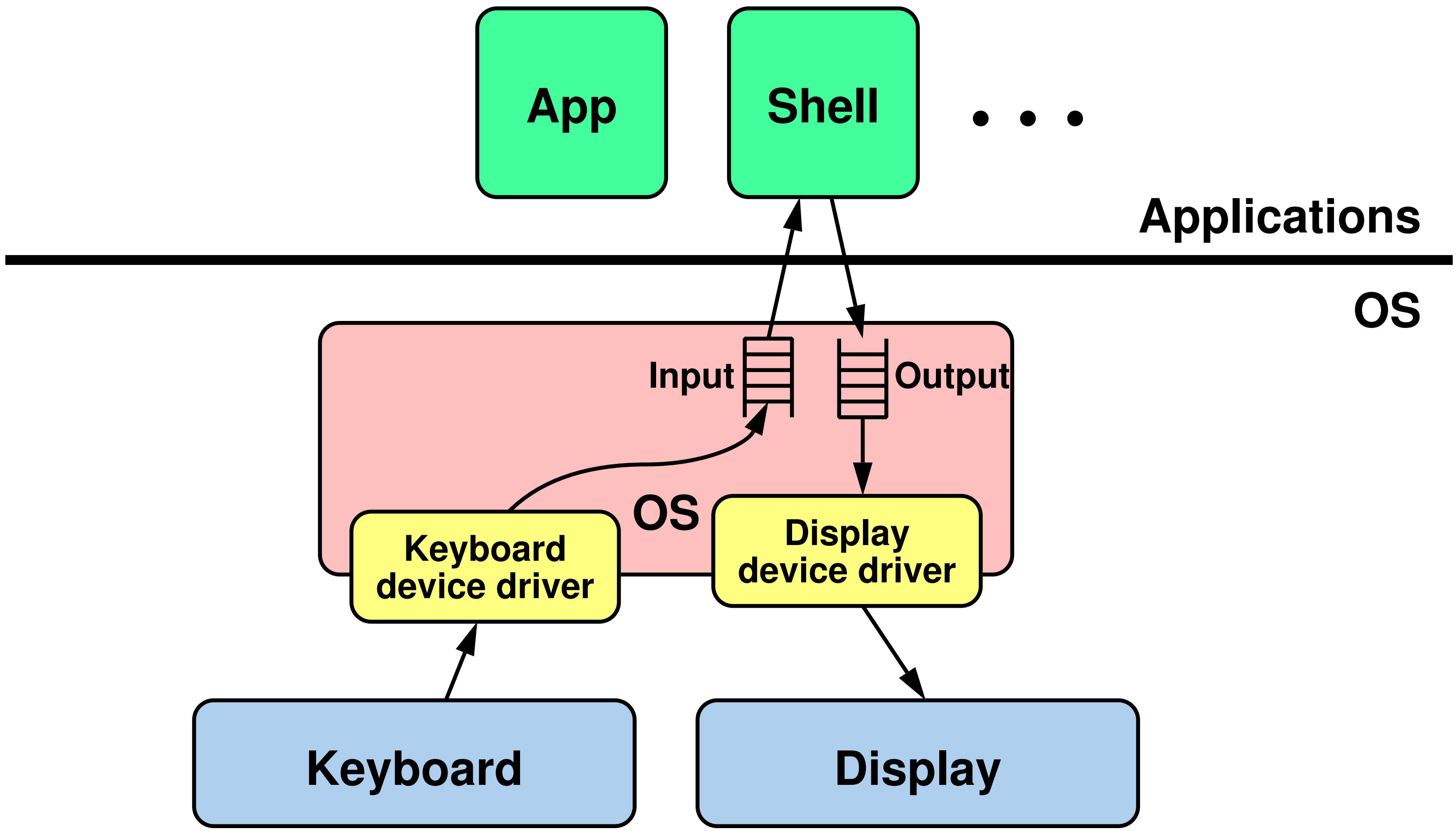
- How to interact with a *terminal device?*
  - what is the right amount of *device independence* for your kernel?
  - characters to be displayed are simply sent to the output routine of the *serial-line driver*
    - to fetch characters that have been typed at the keyboard, a call can be made to its input routine
  - as it turns out, not so simple

Terminals

- In implementing a device driver, need to take *device-specific characteristics* into account
  - but how *device-specific* does it have to be?

- Issues for terminals
  1) terminals are slow and characters generation are too fast
     - need to tell the application to slow down and wait for the terminal to catch up
     - so, we need an *output buffer* to buffer the output and send characters to the terminal from the buffer
     - we have an instance of the *producer-consumer* problem!
  2) characters arrive from the keyboard even though there isn't a waiting read request from an application
     - so, we need an *input buffer* to buffer incoming characters and wait for an application to issue a read request
     - we have another instance of the *producer-consumer* problem!

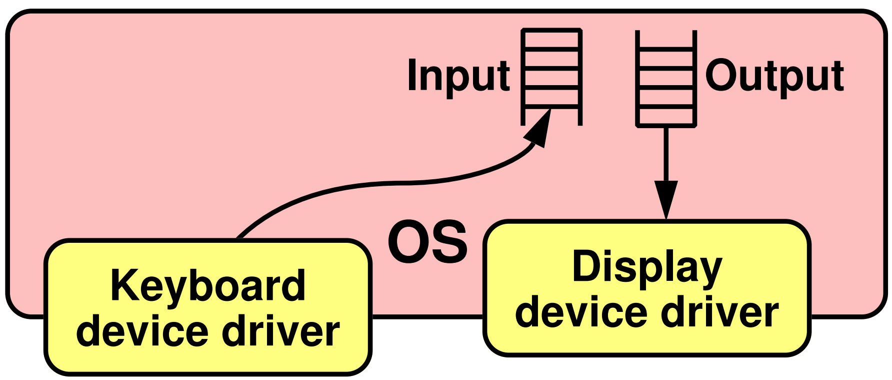
Terminals



- in the old days, only one "terminal driver"

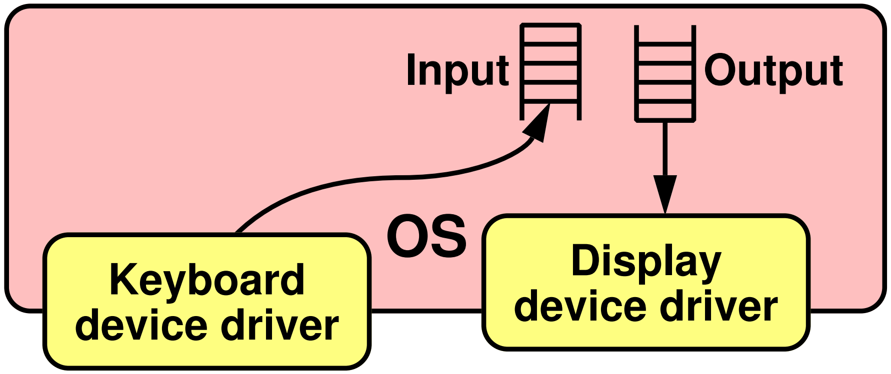
Terminals

- To deal with concerns (1) and (2)
  - use two queues, one for input and one for output
  - characters are placed on the output queue and taken from the input queue in the context of the *application thread*
    - i.e., application write to the output queue and read from the input queue
    - a thread producing output would block if output queue is full
    - a thread consuming input would block if input queue is empty
  - what about the other ends of these queues? who are handling them?



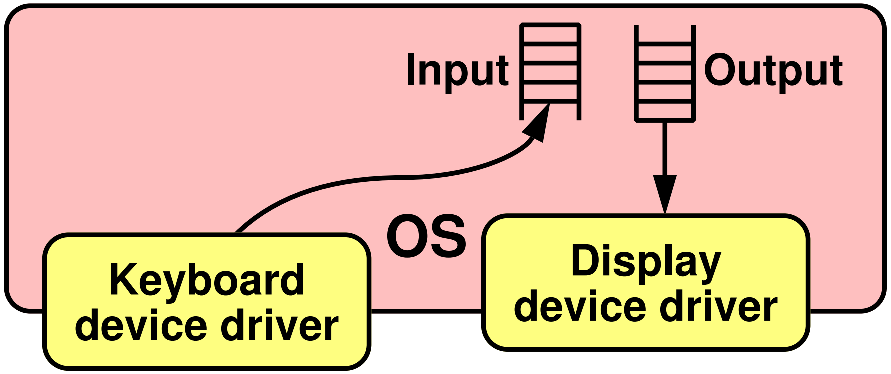
Terminals

- To deal with concerns (1) and (2)
  - how about using a keyboard reading thread (that would do the following)?
    1) issue a read to the device
    2) block itself and wait for interrupt from the device
    3) when interrupt occurs, the thread is woken up
    4) the thread reads from the device and move one character from the device to the input queue
    5) goto step 1
  - this approach of using *thread context* seems to be an overkill and may be inefficient



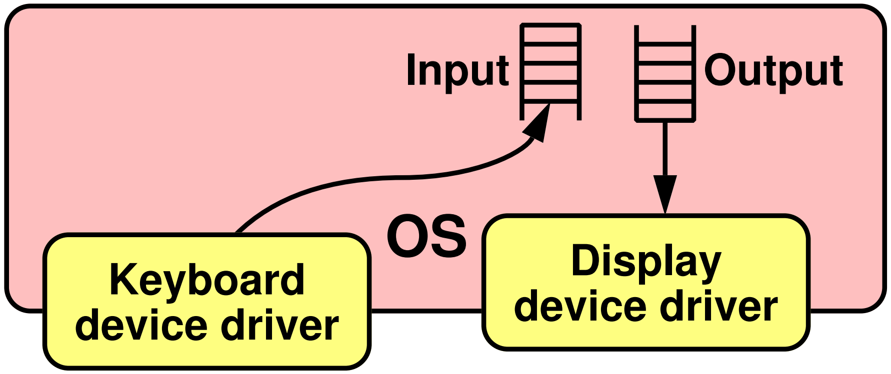
Terminals

- To deal with concerns (1) and (2)
  - how about just using an *interrupt handler*?
    - in the *read-completion interrupt*, the handler moves one character from the device to the input queue and issue another read request to the device and blocks
      - if the queue is *full*, the character is *thrown away* (is this okay?!)
      - the application thread must *mask interrupts* when it's taking a character from the queue
  - can do the same for the output queue ...



Terminals

- To deal with concerns (1) and (2)
  - how about just using an *interrupt handler*?
  - can do the same for the output queue
    - in the *write-completion interrupt*, the handler moves one character from the output queue and issue a write request to the device
      - if the application writes to an *empty queue*, it would *setup the write-completion interrupt handler* and issue a write request to the device
      - for output, losing characters is *not permitted*

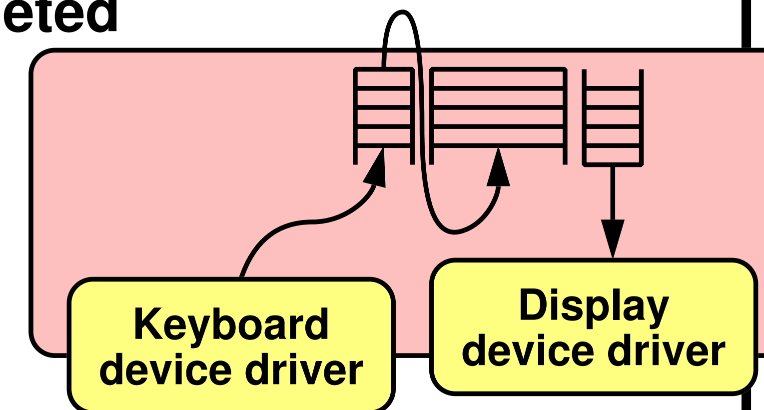


Terminal

- Additional issue for a terminal driver
  3) input characters may need to be processed in some way before they reach the application
     a) characters may be grouped into lines of text and subject to simple *editing* (such as backspace)
     b) some applications prefer to process all characters themselves, including their editing
     c) e.g., characters typed at the keyboard are *echoed* back to the display

Terminals

- To deal with concern (3a)
  - remember, once you allow an application to take a character, you cannot ask for it back
    - can only give it to the application when there is no chance that you will want it back
      - this happens when a line is completed
  - therefore, we need *two input queues*
    - one for the *partial-line*
      - subject to editing
    - the other contain characters from *completed lines*
    - in the read-completion interrupt, the handler moves one character from the device to the *partial-line queue*
      - if the input character is a carriage-return, the entire content of the partial-line queue is moved to the completed-line queue



Terminals

- To deal with concern (3b)
  - use a system call to select single vs. multiple input queues
- To deal with concern (3c)
  - when a character is typed, it should go to the display immediately (due to the *echoing* requirement)
    - it may be competing with the output thread, but it's okay (and that's how it's done in Unix)
    - Windows handle this differently
      - typed characters are only echoed when an application consumes them
      - therefore, echoing is not done in the interrupt context
      - echoing is done in the context of the thread consuming the characters (i.e., the one that calls `read()`)