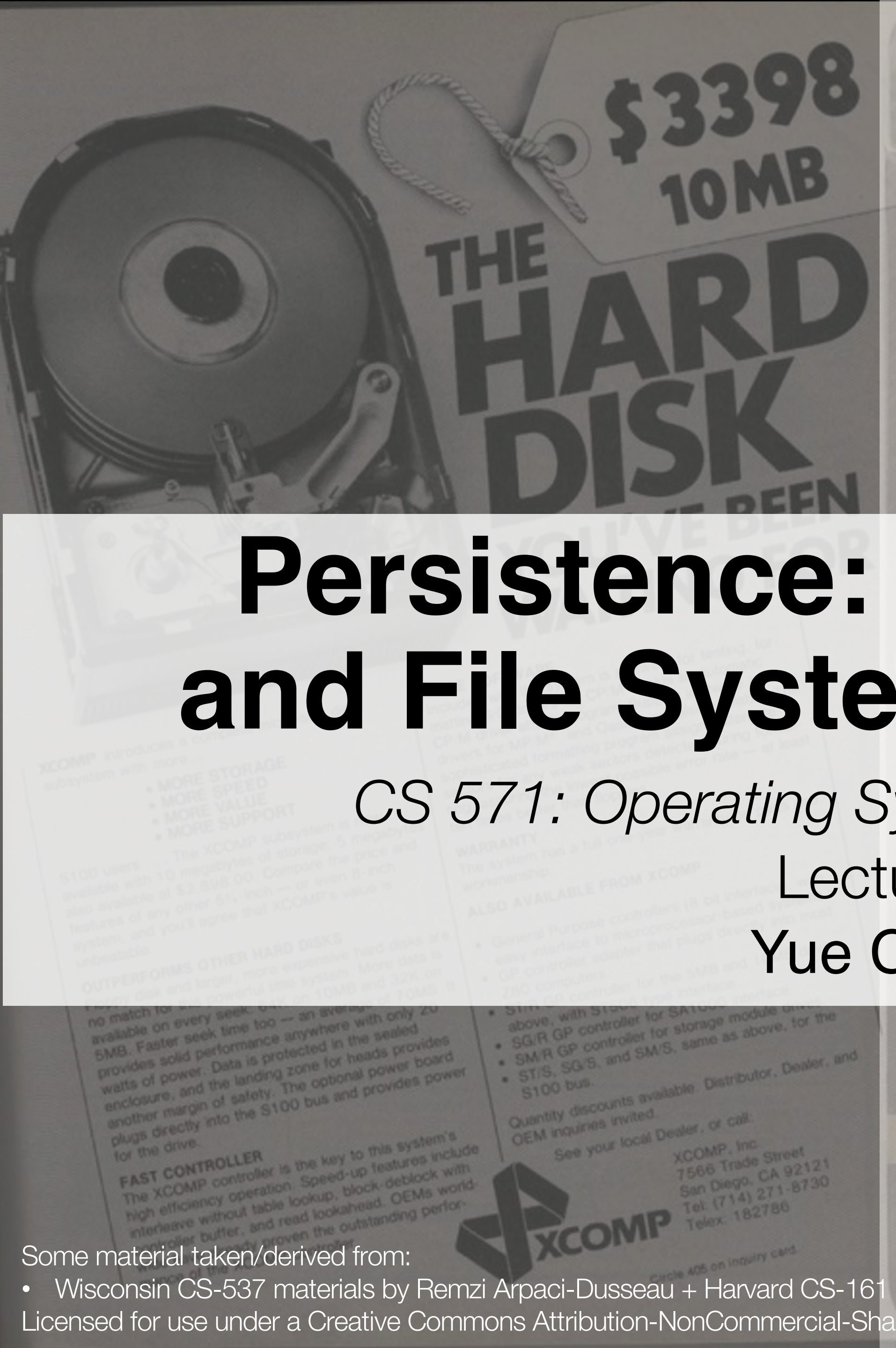


Persistence: HDDs, SSDs, and File System Abstraction

CS 571: Operating Systems (Spring 2022)

Lecture 8

Yue Cheng

Hard Disk Drives (HDDs)

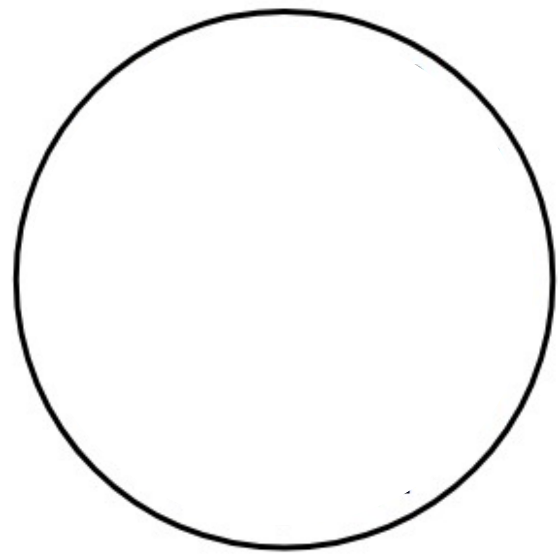
Basic Interface

- A magnetic disk has a sector-addressable address space
  - You can think of a disk as an array of sectors
  - Each sector (logical block) is the smallest unit of transfer
- Sectors are typically 512 or 4096 bytes
- Main operations
  - Read from sectors (blocks)
  - Write to sectors (blocks)

Disk Structure

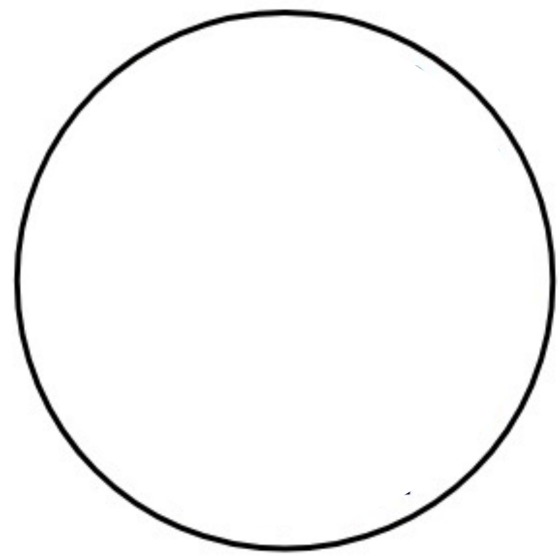
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address should be easy
    - Except for bad sectors

Internals of Hard Disk Drive (HDD)



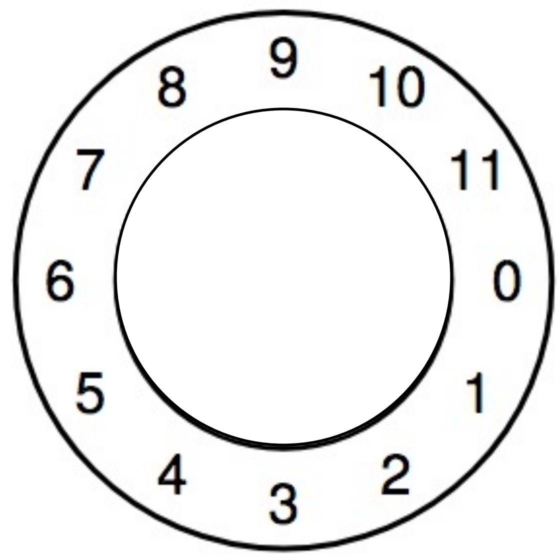
Internals of Hard Disk Drive (HDD)

Platter
Covered with a magnetic film



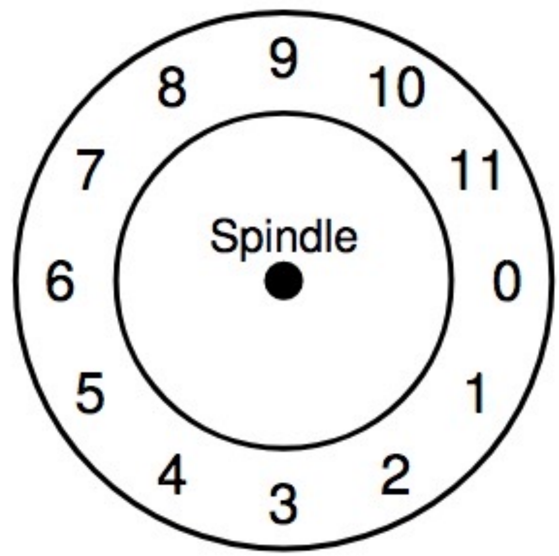
Internals of Hard Disk Drive (HDD)

A single track example



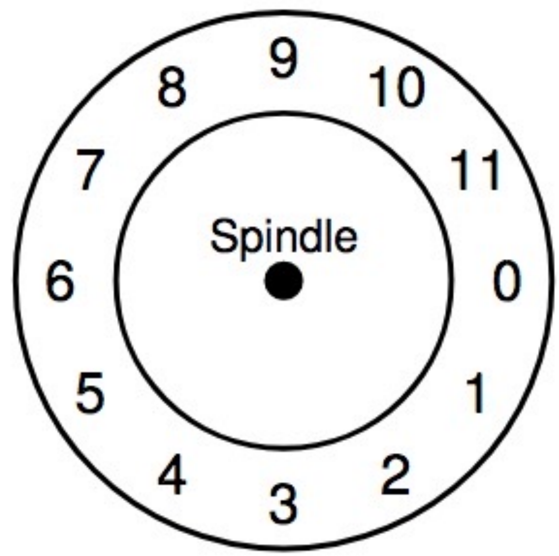
Internals of Hard Disk Drive (HDD)

Spindle in the center of the surface



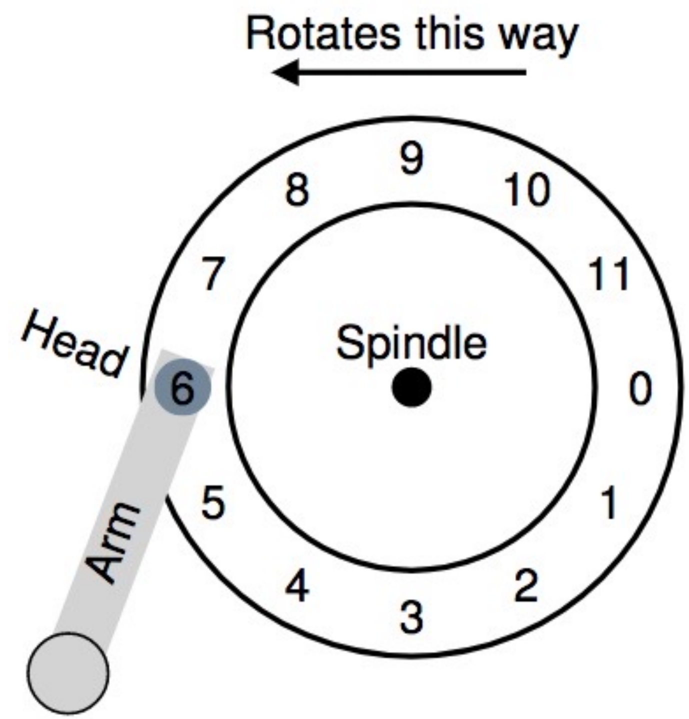
Internals of Hard Disk Drive (HDD)

The track is divided into numbered sectors

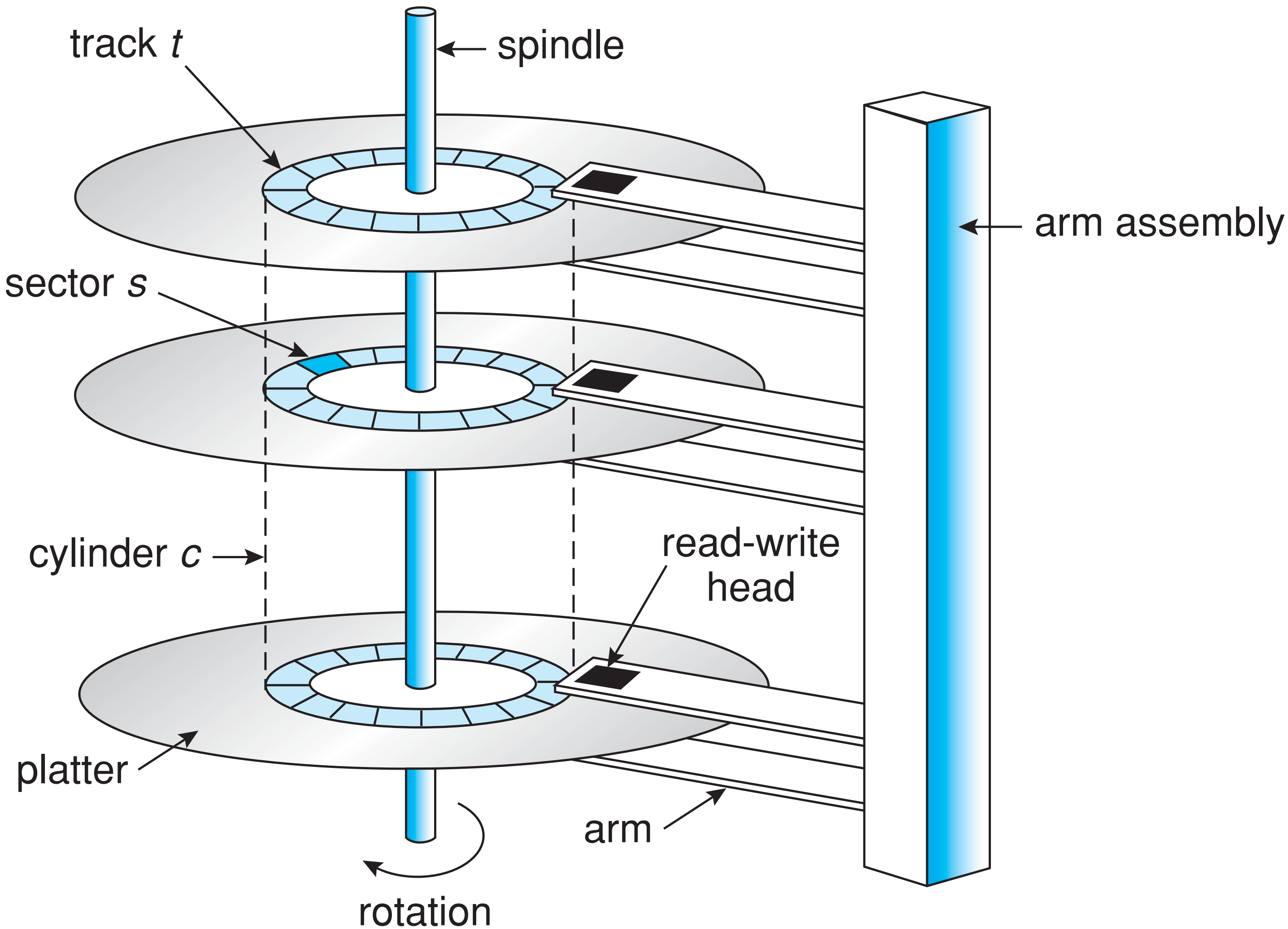


Internals of Hard Disk Drive (HDD)

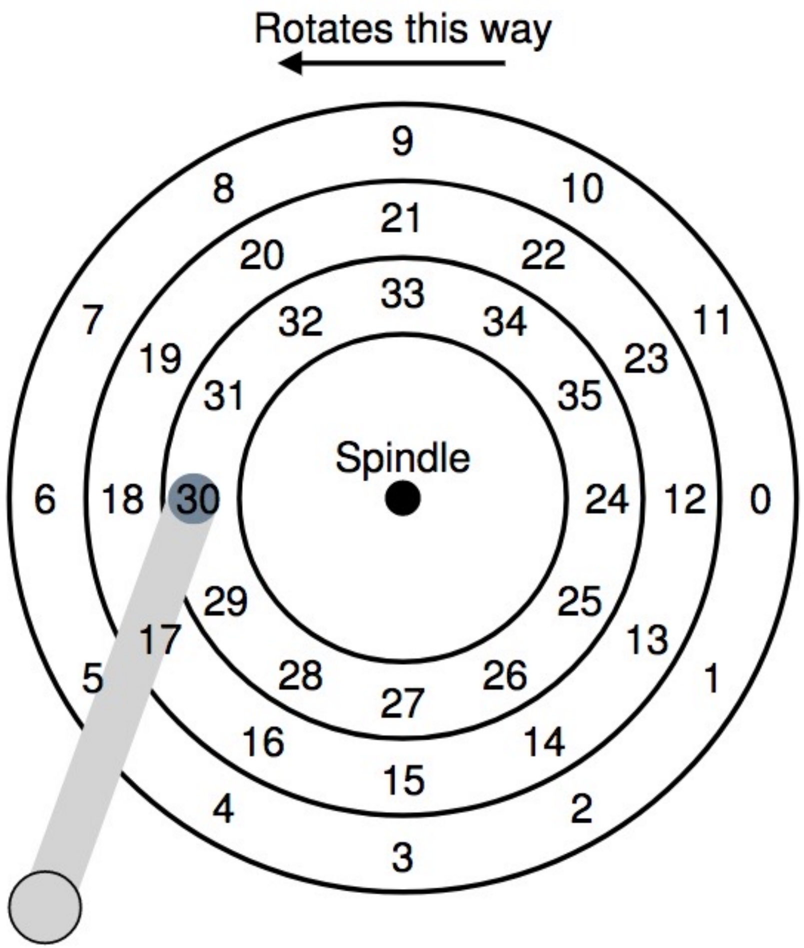
A single track + an arm + a head



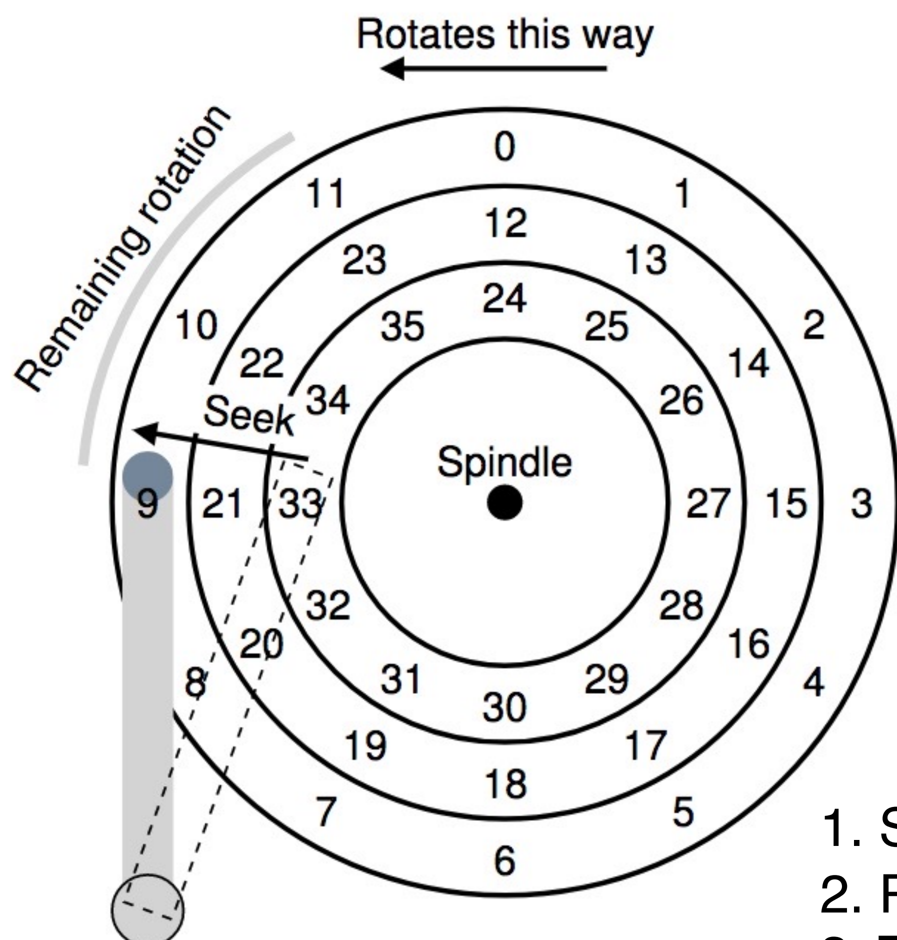
HDD Mechanism (3D view)



Let's Read Sector 0



Let's Read Sector 0



1. Seek for right track
2. Rotate (sector 9 → 0)
3. Transfer data (sector 0)

Don’t Try This at Home!

https://www.youtube.com/watch?v=9eMWG3fwiEU&feature=youtu.be&t=30s

Disk Performance

- I/O latency of disks

$$L_{I/O} = L_{seek} + L_{rotate} + L_{transfer}$$

- Disk access latency at millisecond level

Seek, Rotate, Transfer

- Seek may take several milliseconds (ms)
- Settling along can take 0.5 - 2ms
- Entire seek often takes 4 - 10ms

Seek, Rotate, Transfer

- Rotation per minute (RPM)
  - 7200 RPM is common nowadays
  - 15000 RPM is high end
  - Old computers may have 5400 RPM disks

- `1 / 7200 RPM = 1 minute / 7200 rotations =`

  `1 second / 120 rotations =` **`8.3 ms`** `/ rotation`

Seek, Rotate, Transfer

- Rotation per minute (RPM)
  - 7200 RPM is common nowadays
  - 15000 RPM is high end
  - Old computers may have 5400 RPM disks

- `1 / 7200 RPM = 1 minute / 7200 rotations =`

  `1 second / 120 rotations =` **8.3 ms** `/ rotation`

- So it may take 4.2 ms **on average** to rotate to target (0.5 * 8.3 ms)

Seek, Rotate, Transfer

- Relatively fast
  - Depends on RPM and sector density
- 100+ MB/s is typical for SATA I (1.5Gb/s max)
  - Up to 600MB/s for SATA III (6.0Gb/s)
- `1s / 100MB = 10ms / MB = 4.9us/sector`
  - Assuming 512-byte sector

Workloads

- Seeks and rotations are slow while transfer is relatively fast

- What kind of workload is best suited for disks?

Workloads

- Seeks and rotations are slow while transfer is relatively fast

- What kind of workload is best suited for disks?
  - Sequential I/O: access sectors in order (transfer dominated)

- Random workloads access sectors in a random order (seek+rotation dominated)
  - Typically slow on disks
  - Never do random I/O unless you must! E.g., Quicksort is a terrible algorithm for disk!

Disk Performance Calculation

- Seagate Enterprise SATA III HDD

Metric	Perf
RPM	7200
Avg seek	4.16ms
Max transfer	500MB/s

- How long does an average 4KB read take?

Disk Performance Calculation

- Seagate Enterprise SATA III HDD

Metric	Perf
RPM	7200
Avg seek	4.16ms
Max transfer	500MB/s

- How long does an average 4KB read take?

$$transfer = \frac{1\ sec}{500\ MB} \times 4\ KB \times \frac{1{,}000{,}000\ us}{1\ sec} = 8\ us$$

Disk Performance Calculation

- Seagate Enterprise SATA III HDD

Metric	Perf
RPM	7200
Avg seek	4.16ms
Max transfer	500MB/s

- How long does an average 4KB read take?

$$transfer = \frac{1\,sec}{500\,MB} \times 4\,KB \times \frac{1{,}000{,}000\,us}{1\,sec} = 8\,us$$

```
Latency = 4.16 ms + 4.2 ms + 8 us =
8.368 ms
```

Avg Seek (4.16 ms) — Avg Rotate (4.2 ms)

Solid State Drives (SSDs)

Disk Recap

- I/O requires: seek, rotate, transfer

- Inherently:
  - Not parallel (only one head)
  - Slow (mechanical)
  - Poor random I/O (locality around disk head)

- Random requests each taking ~10+ ms

SSD Overview

- Hold charge in cells. No moving (mechanical) parts (no seeks)!
  - SSDs use transistors (just like DRAM), but SSD data persists when the power goes out
  - NAND-based flash is the most popular technology, so we'll focus on it
- SSD is Inherently parallel!

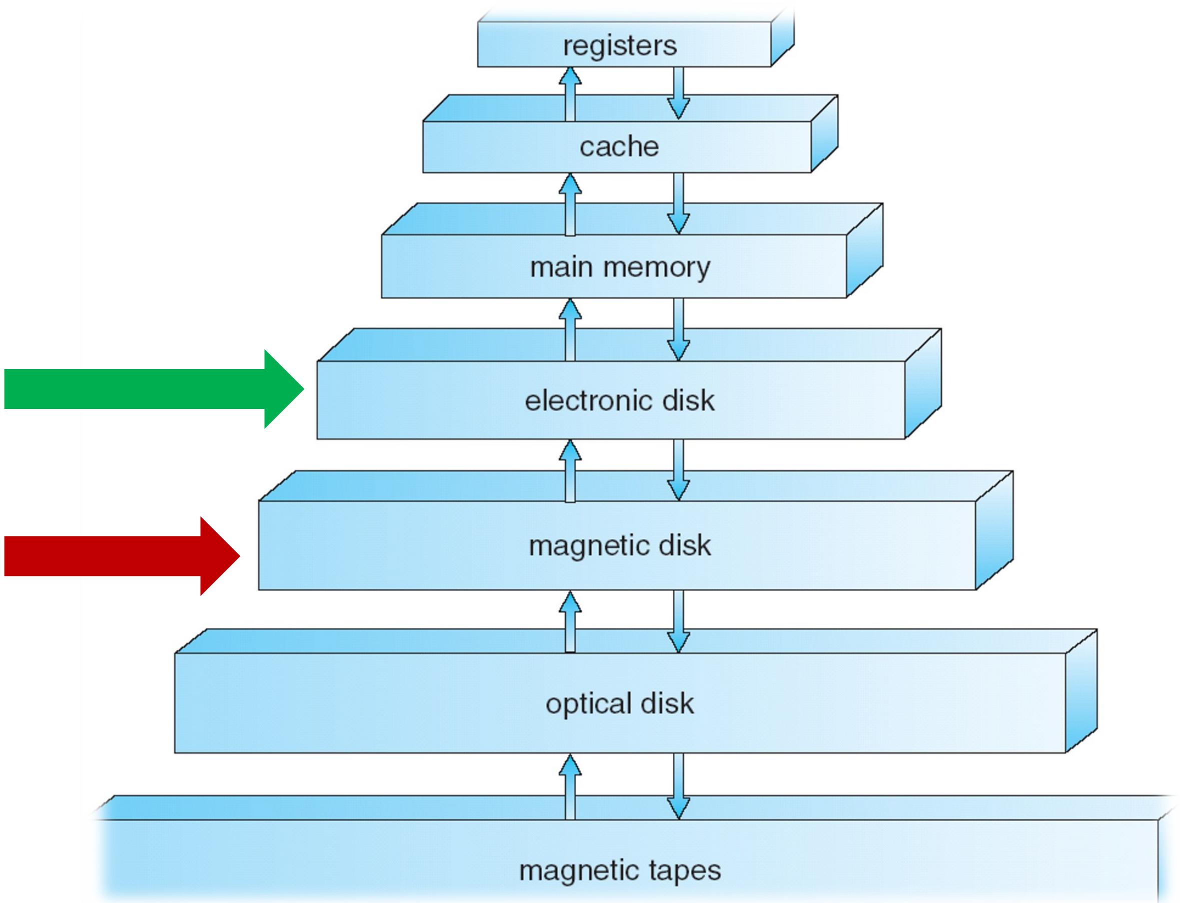
SSD Overview

- Hold charge in cells. No moving (mechanical) parts (no seeks)!
  - SSDs use transistors (just like DRAM), but SSD data persists when the power goes out
  - NAND-based flash is the most popular technology, so we'll focus on it
- SSD is Inherently parallel!
- **High-level takeaways**
  1. SSDs have a higher $/bit than HDDs, but better performance
  2. SSDs handle writes in a strange way; this has implications for file system design

Storage Hierarchy Overview

SSD:
Smaller capacity
Higher $/bit
Faster accesses

HDD:
Larger capacity
Lower $/bit
Way slower accesses



Disk vs. SSD: Performance

- Throughput
  - Disk: ~130MB/s (sequential)
  - SSD: ~400MB/s

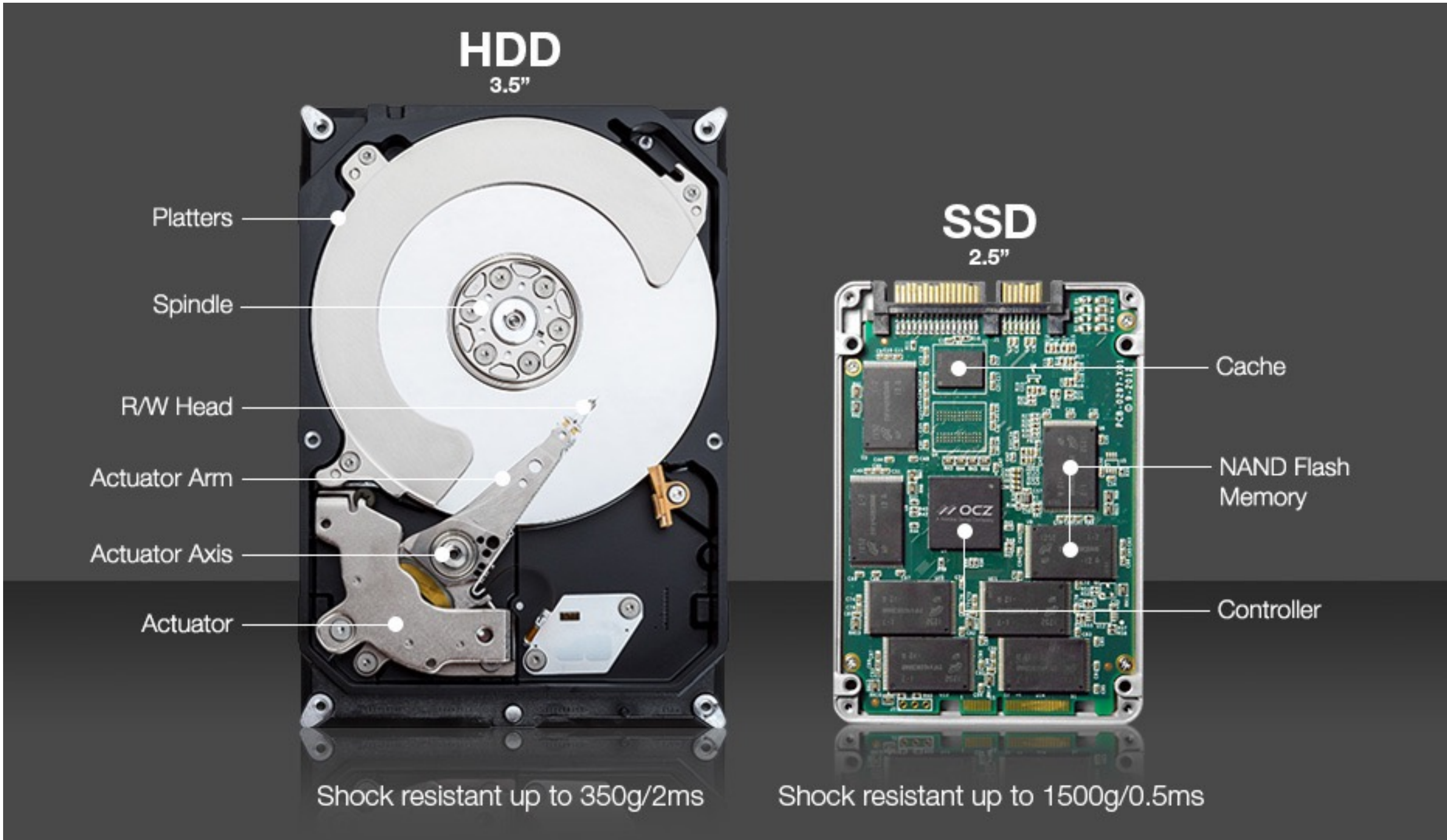
- Latency
  - Disk: ~10ms (one op)
  - SSD:
    - Read: 10-50us
    - Program: 200-500us
    - Erase: 2ms

Disk vs. SSD: Performance

- Throughput
  - Disk: ~130MB/s (sequential)
  - SSD: ~400MB/s
- Latency
  - Disk: ~10ms (one op)
  - SSD:
    - Read: 10-50us
    - Program: 200-500us
    - Erase: 2ms

Types of write, more later…

Disk vs. SSD: Internal



* https://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd.html

Disk vs. SSD: Capacity & Price

"An obvious question is why are we talking about spinning disks at all, rather than SSDs, which have higher IOPS and are the "future" of storage. The root reason is that the cost per GB remains too high, and more importantly that the growth rates in capacity/$ between disks and SSDs are relatively close (at least for SSDs that have sufficient numbers of program-erase cycles to use in data centers), so that cost will not change enough in the coming decade. We do make extensive use of SSDs, but primarily for high performance workloads and caching, and this helps disks by shifting seeks to SSDs.

~ Eric Brewer et al. Google

Source: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44830.pdf

	SSD	HDD
Price	$0.25-$0.27 per GB average	$0.2-$0.03 per GB average

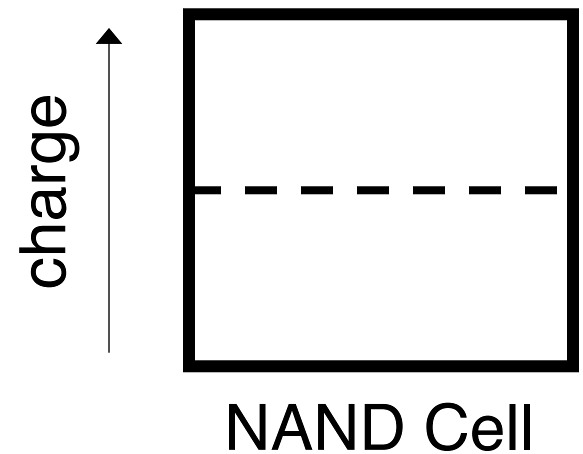
Disk vs. SSD: Summary

	SSD	HDD
Price	$0.25-$0.27 per GB average	$0.2-$0.03 per GB average
Lifespan	30-80% test developed bad block in their lifetime	3.5% developed bad sectors comparatively
Ideal for	High performance processing Residing in APA or Tier 0/1 media in hybrid arrays	High capacity nearline tiers Long-term retained data
Read/write speeds	200 MB/s to 2500 MB/s	up to 200 MB/s
Benefits	Higher performance for faster read/write operations and fast load times	Less expensive Mature technology and massive installed user base
Drawbacks	May not be as durable/reliable as HDDs Not good for long-term archival data	Mechanical components take longer to read-write than SSDs

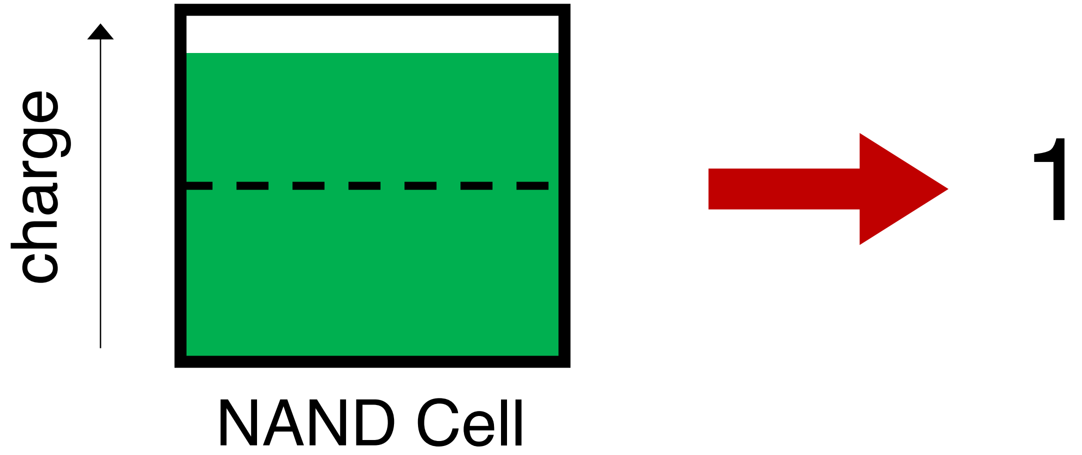
* https://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd.html

SSD Architecture

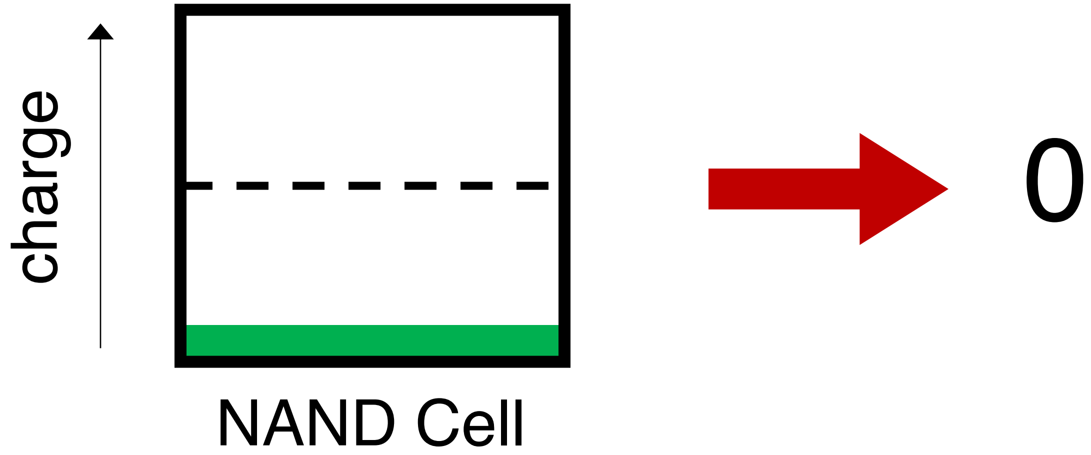
SLC: Single-Level Cell



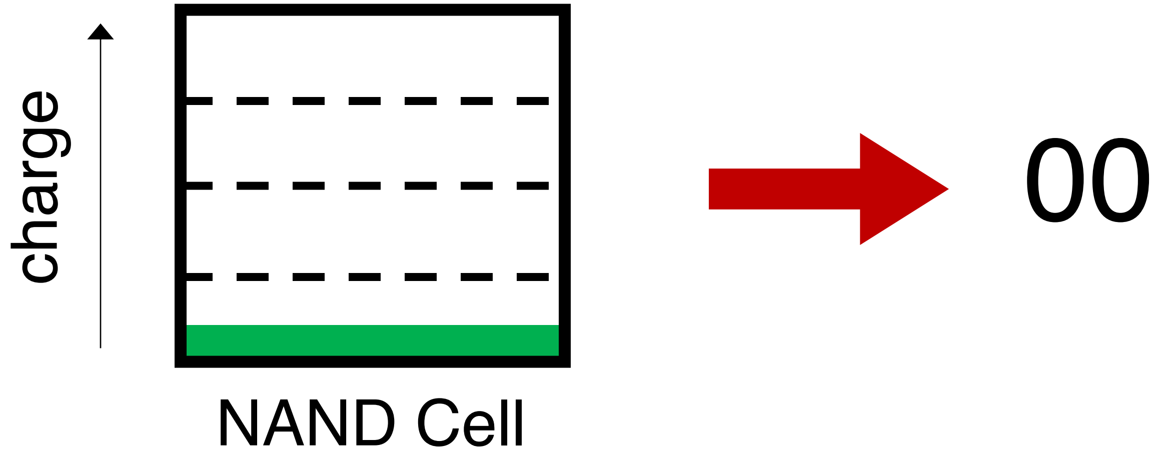
SLC: Single-Level Cell



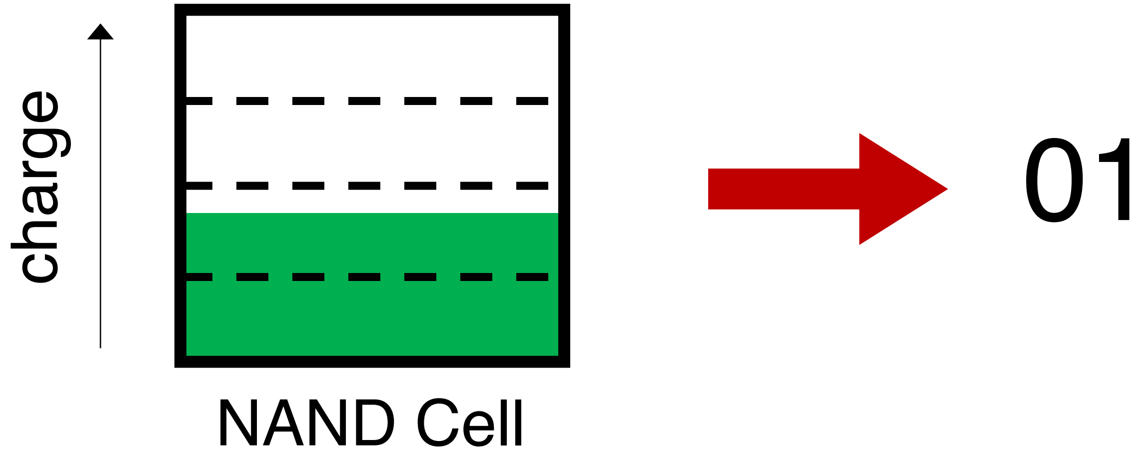
SLC: Single-Level Cell



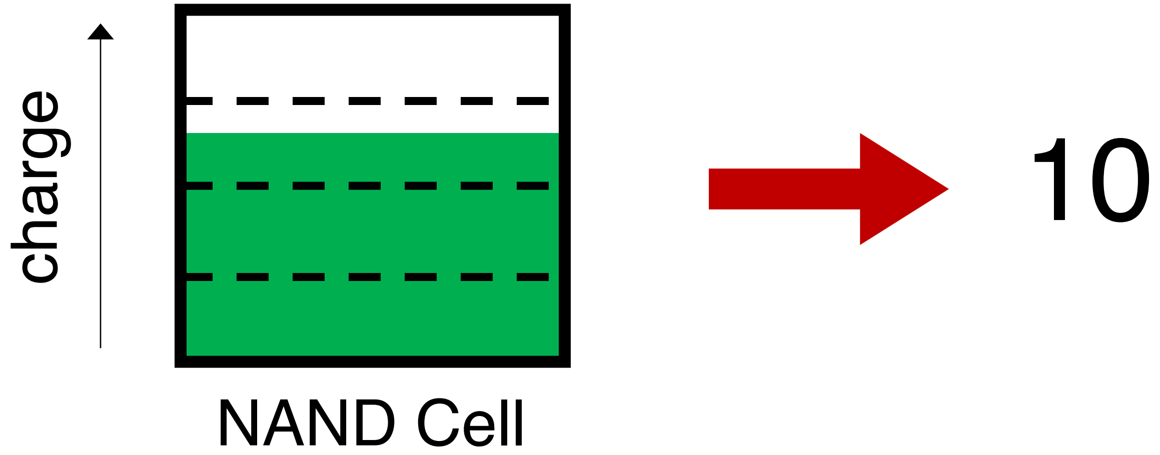
MLC: Multi-Level Cell



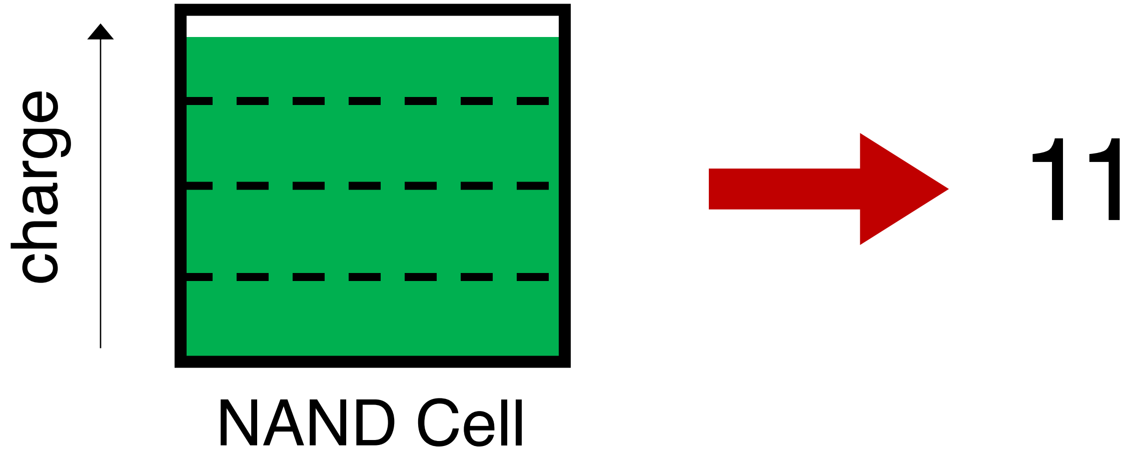
MLC: Multi-Level Cell



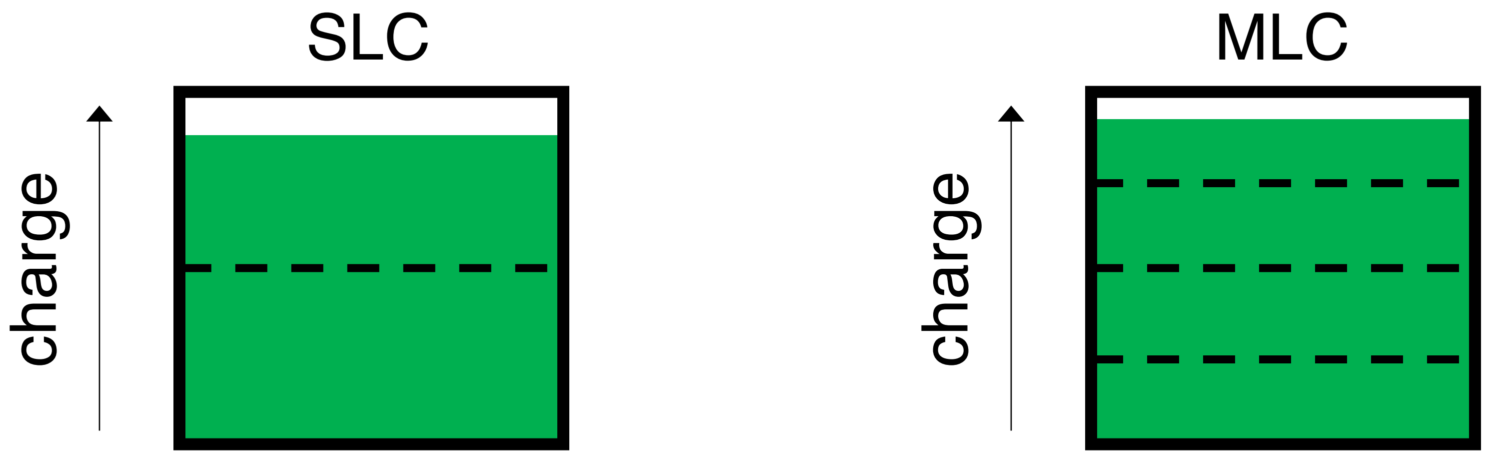
MLC: Multi-Level Cell



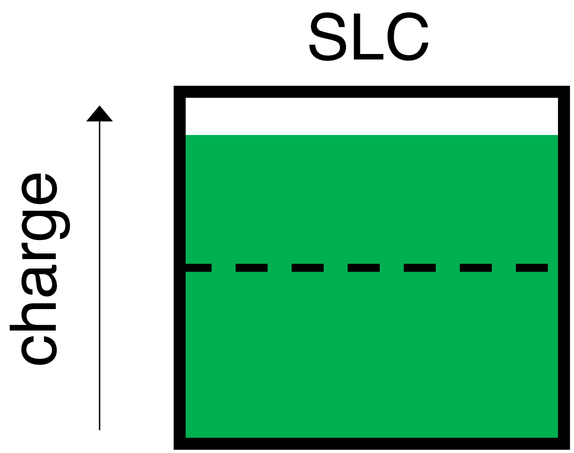
MLC: Multi-Level Cell



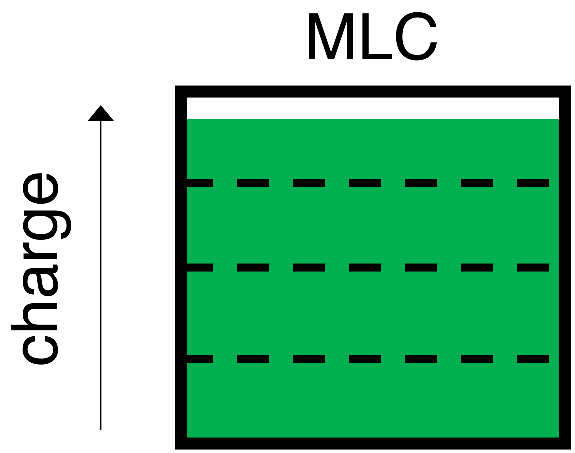
Single- vs. Multi-Level Cell



Single- vs. Multi-Level Cell



expensive
robust
1 cell: 1 bit



cheap
sensitive
1 cell: multi-bit

Wearout

- Problem: flash cells wear out after being erased too many times

- MLC: ~10K times
- SLC: ~100K times

- Usage strategy: ???

Wearout

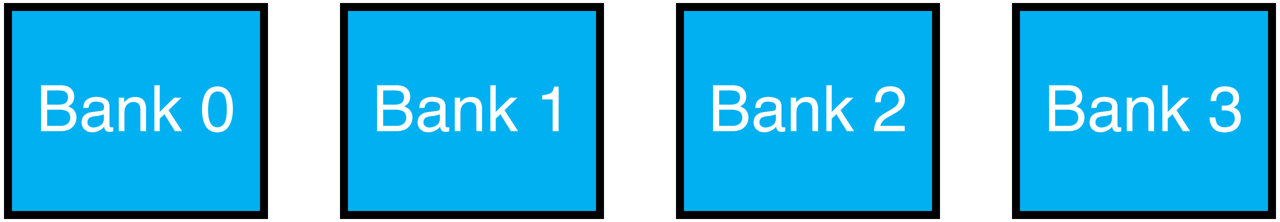
- Problem: flash cells wear out after being erased too many times

- MLC: ~10K times
- SLC: ~100K times

- Usage strategy: wear leveling
  - Prevents some cells from being wornout while others still fresh

Banks

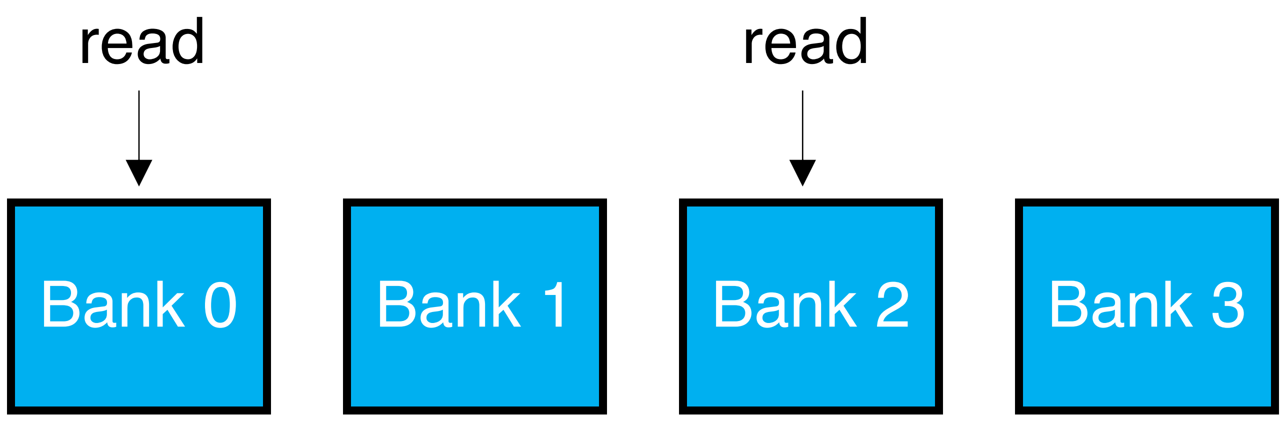
- SSD devices are divided into banks (aka. planes)
- Banks can be accessed in parallel



Banks

- SSD devices are divided into banks (aka. planes)

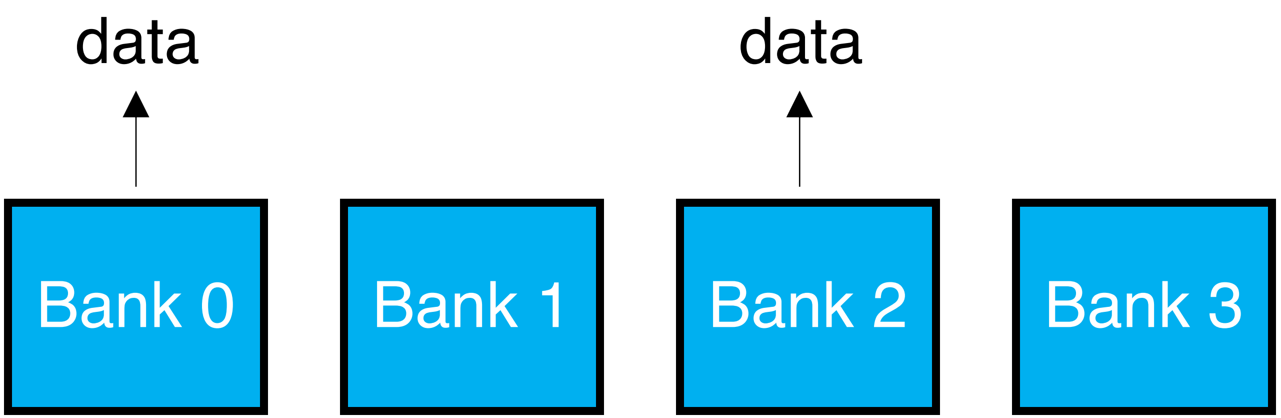
- Banks can be accessed in parallel



Banks

- SSD devices are divided into banks (aka. planes)

- Banks can be accessed in parallel



SSD Writes

- Writing 0's
  - Fast, fine-grained

- Writing 1's
  - Slow, coarse-grained

SSD Writes

- Writing 0's
  - Fast, fine-grained
  - called "program"
- Writing 1's
  - Slow, coarse-grained
  - called "erase"

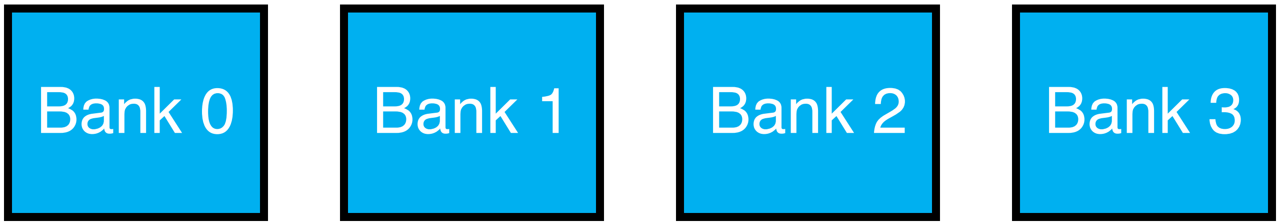
SSD Writes

- Writing 0’s
  - Fast, fine-grained [page-level]
  - called “program”
- Writing 1’s
  - Slow, coarse-grained [block-level]
  - called “erase”

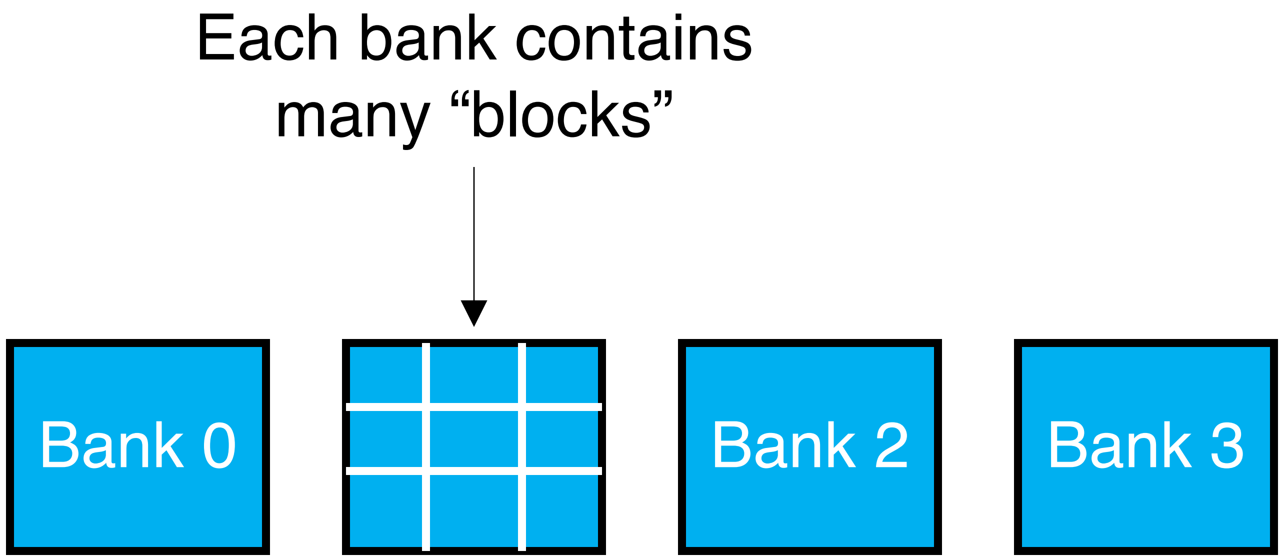
SSD Writes

- Writing 0's
  - Fast, fine-grained [page-level]
  - called "program"
- Writing 1's
  - Slow, coarse-grained [block-level]
  - called "erase"
- SSD can only "write" (program) into clean pages
  - "clean": pages containing all 1's (pages that have been erased)
  - SSD does not support in-place overwrite!

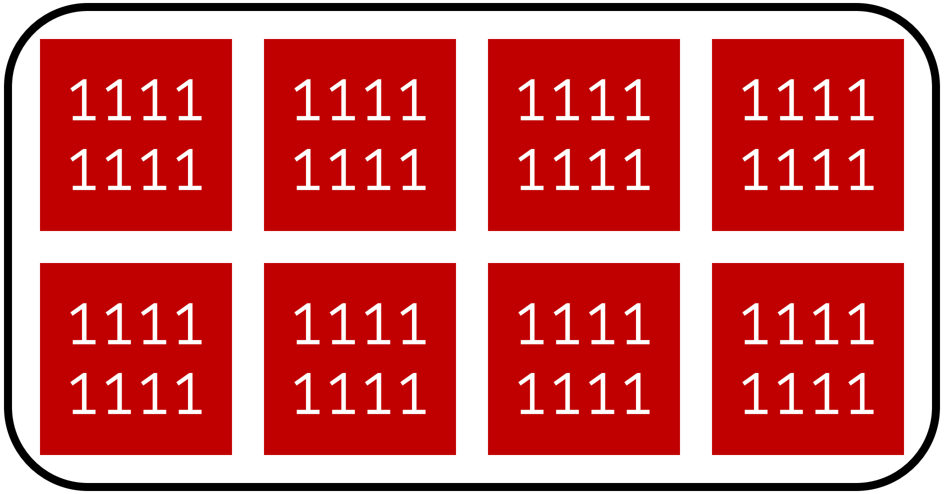
Banks and Blocks



Banks and Blocks

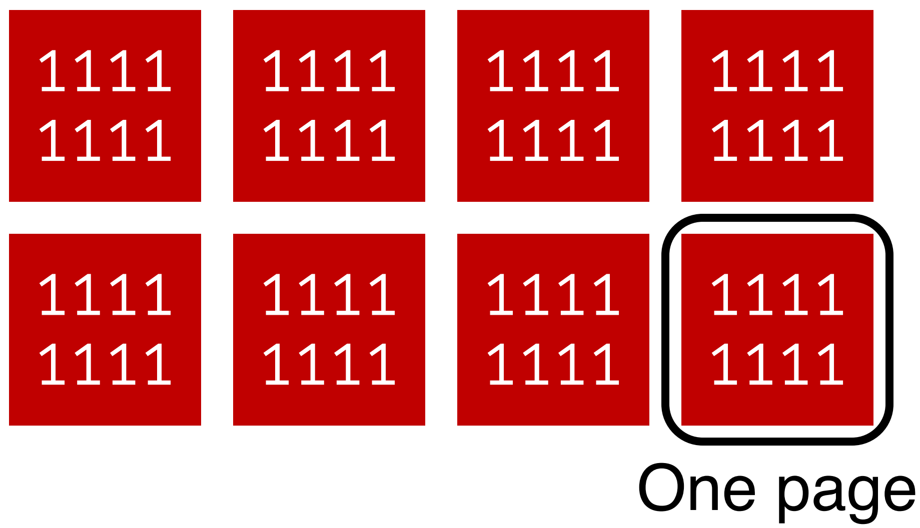


Block and Pages

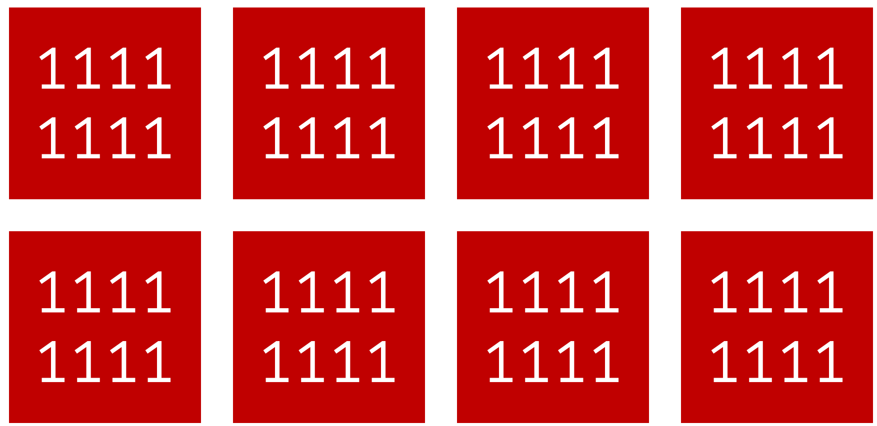


One block

Block and Pages

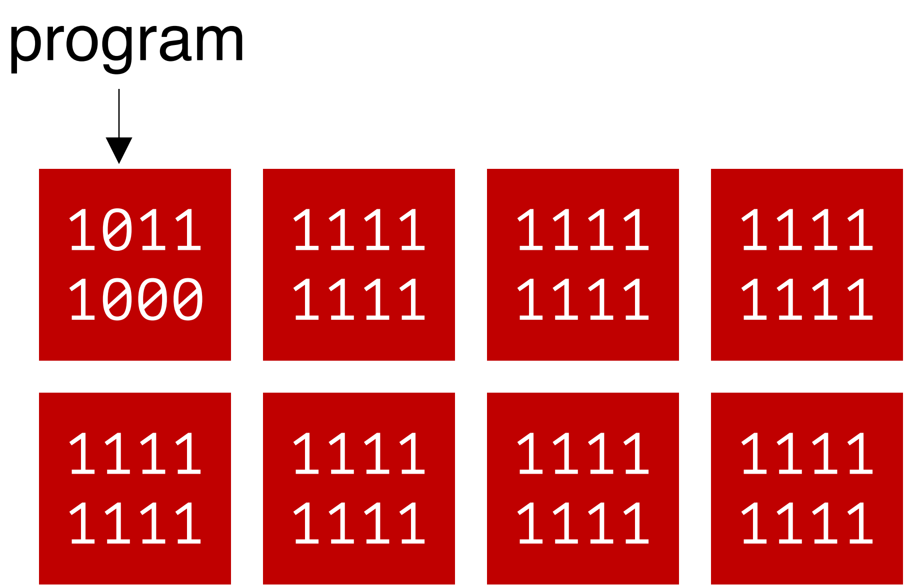


Block and Pages

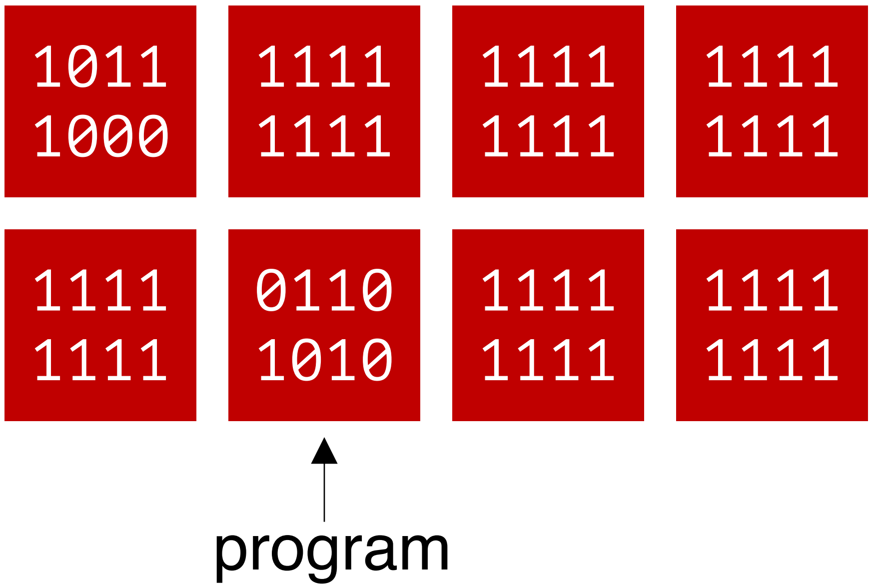


All pages are clean
(“programmable”)

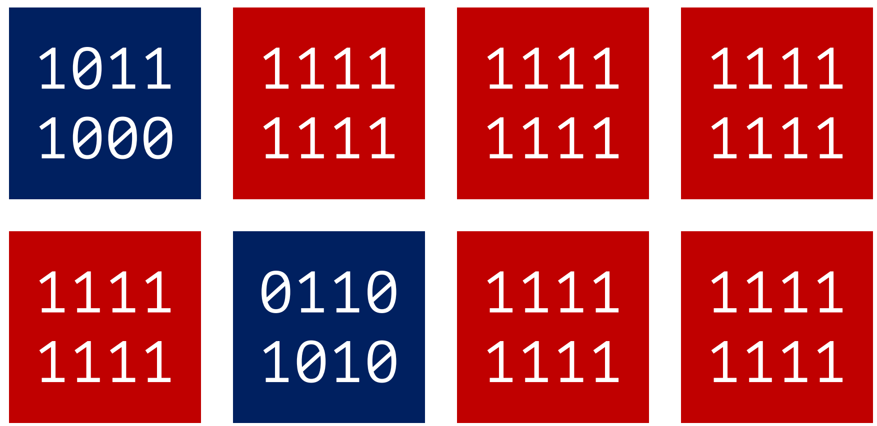
Block


program
1011
1000
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111

Block


1011
1000
1111
1111
1111
1111
1111
1111
1111
1111
0110
1010
1111
1111
1111
1111
program

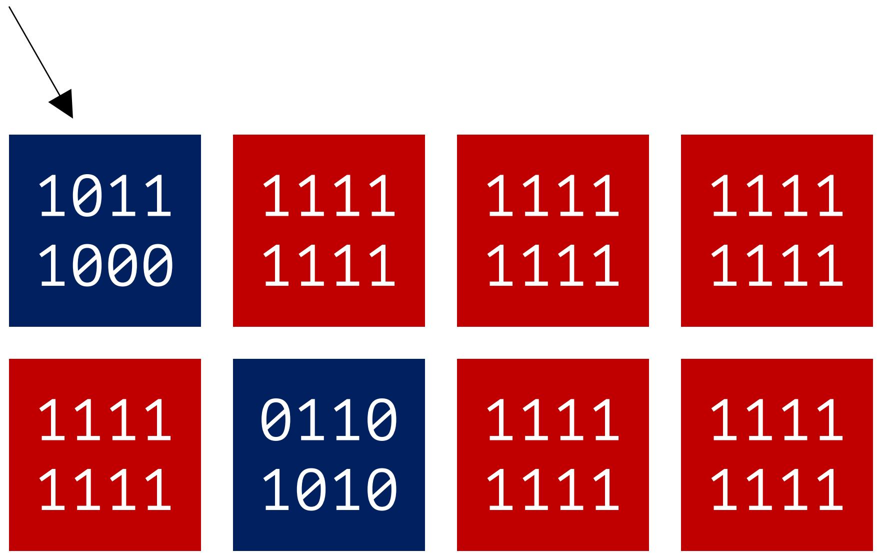
Block



Two pages hold data
(**cannot be overwritten**)

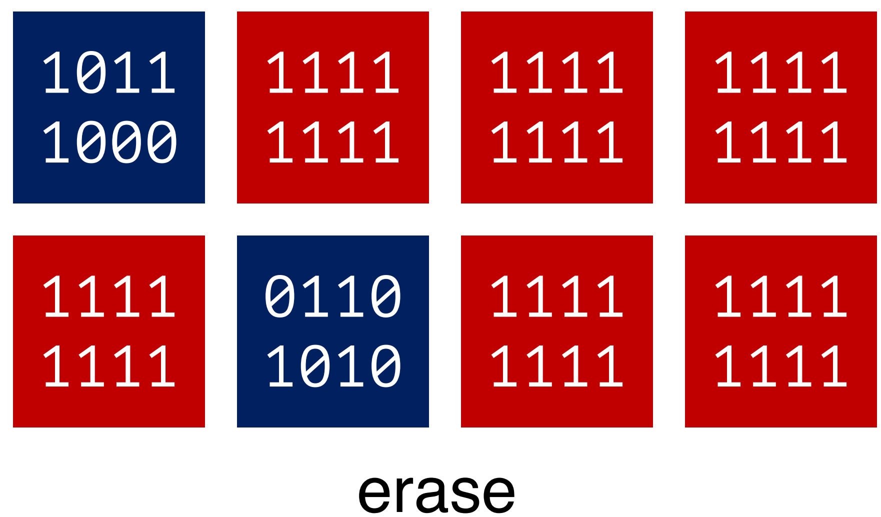
Block

still want to write data into this page???

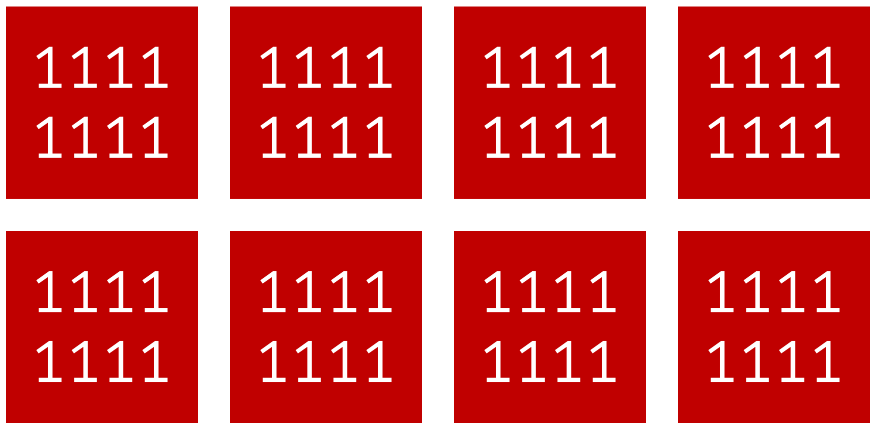


Two pages hold data
(**cannot be overwritten**)

Block

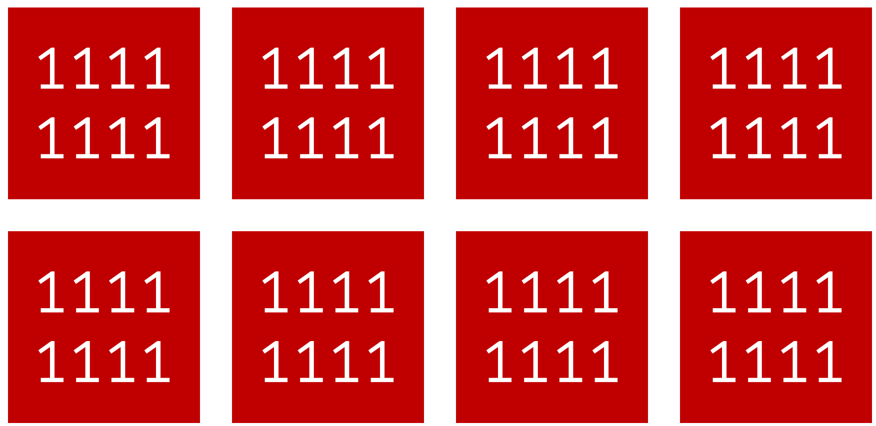

1011
1000
1111
1111
1111
1111
1111
1111
1111
1111
0110
1010
1111
1111
1111
1111
erase

Block



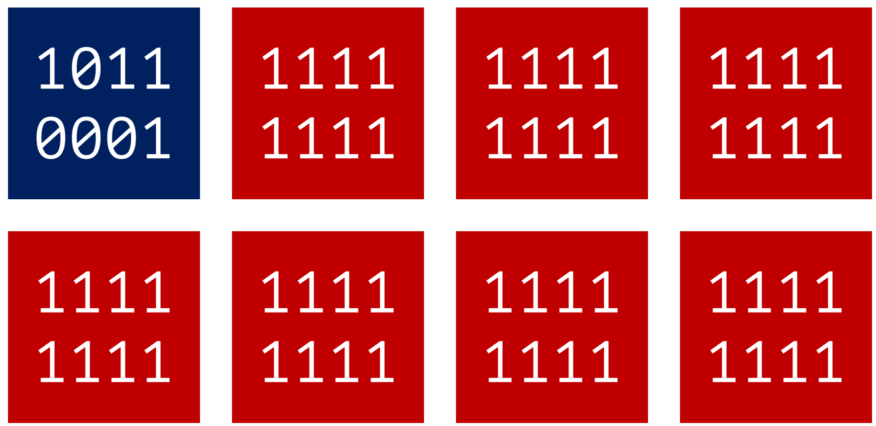
erase
(the whole block)

Block



After erase, again, **free state**
(can write new data in any page)

Block



This dark blue page holds data

SSD vs. Disk: APIs

	disk	flash
read		
write		

SSD vs. Disk: APIs

	disk	flash
read	read sector	read page
write		

SSD vs. Disk: APIs

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

SSD Architecture

- Bank/plane: 1024 to 4096 blocks
  - Banks accessed in parallel
- Block: 64 to 256 pages
  - Unit of erase
- Page: 2 to 8 KB
  - Unit of read and program

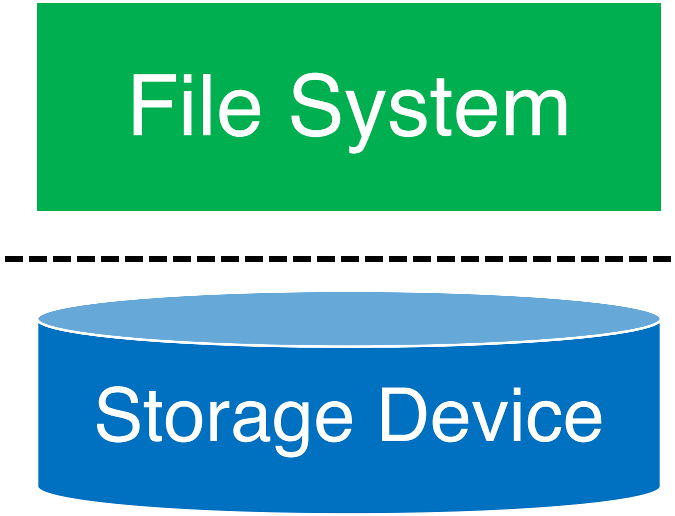
Disk vs. SSD: Performance

- Throughput
  - Disk: ~130MB/s (sequential)
  - Flash: ~400MB/s

- Latency
  - Disk: ~10ms (one op)
  - Flash:
    - Read: 10-50us
    - Program: 200-500us
    - Erase: 2ms

Working with File System

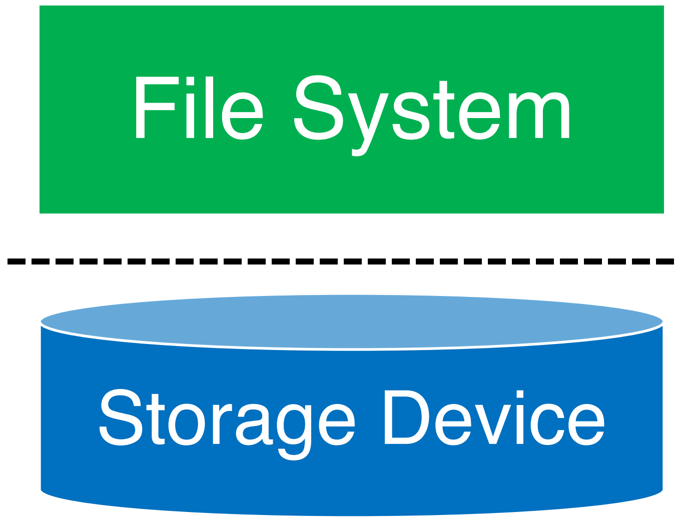
Traditional File Systems



Traditional API:

- read sector
- write sector

Traditional File Systems



Traditional API:

- read sector
- write sector

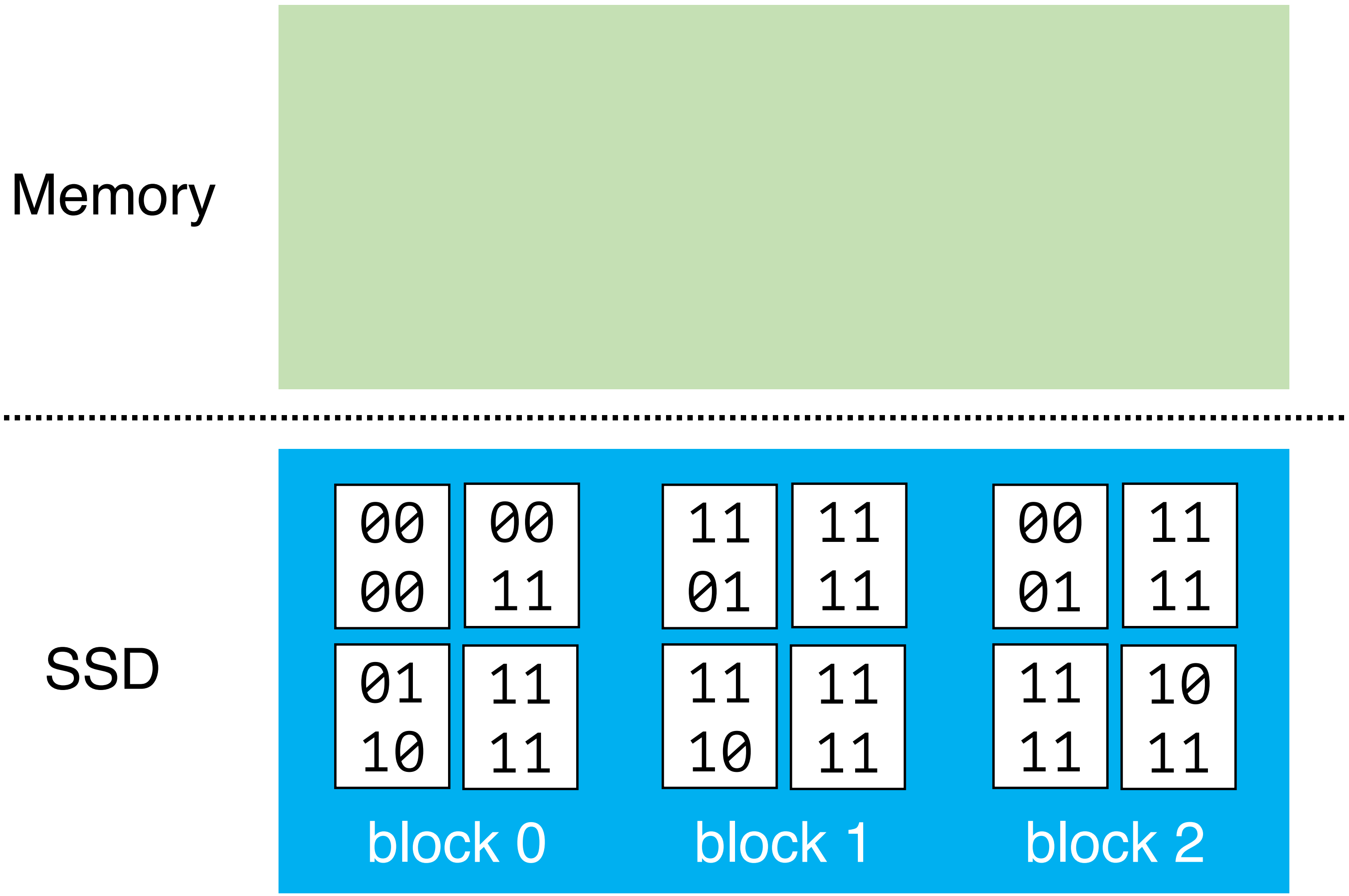
Mismatch with SSD!

Traditional APIs wrapping around SSD APIs

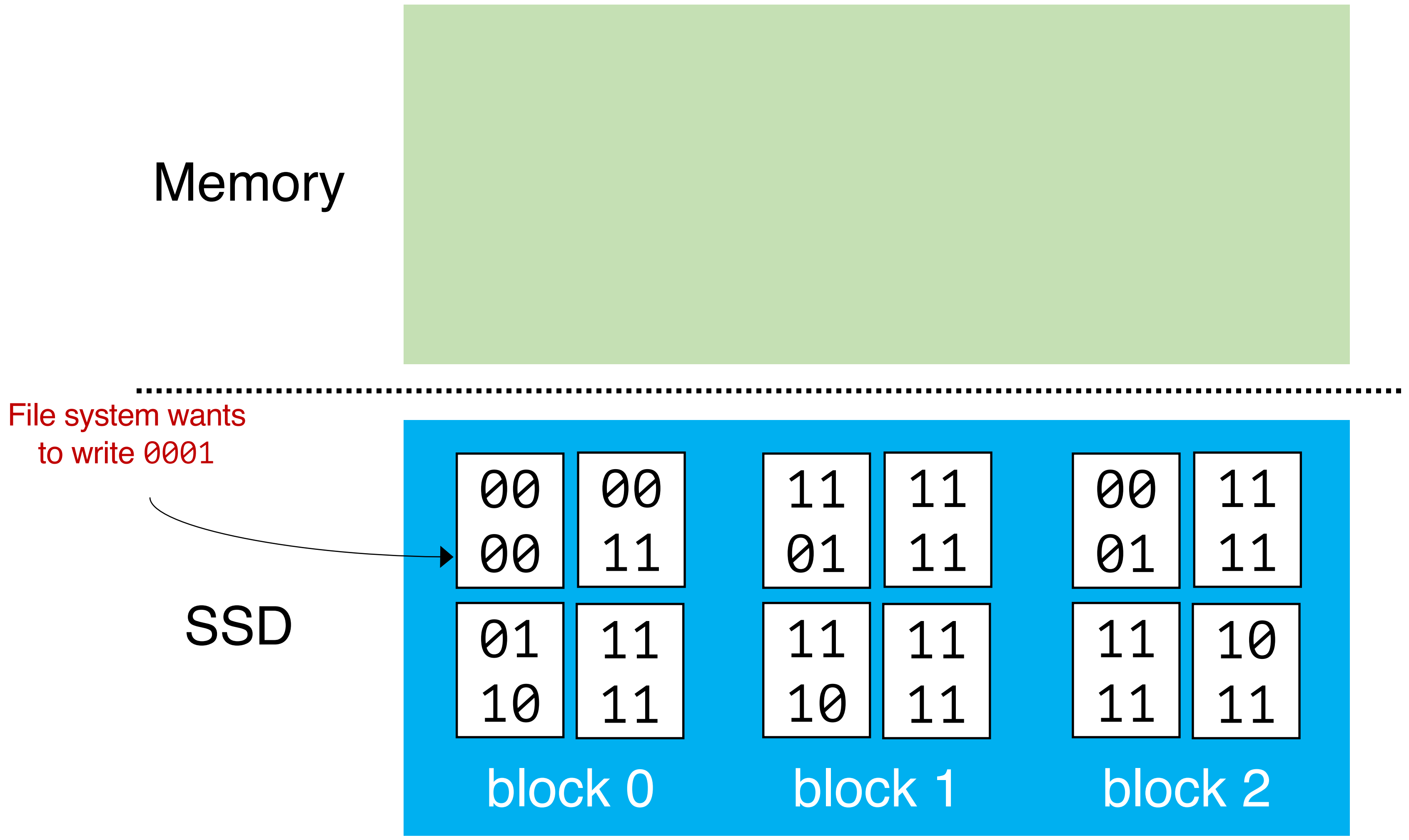
```
read(addr):
    return ssd_read(addr)

write(addr, data):
    block_copy = ssd_read(all pages of block)
    modify block_copy with data
    ssd_erase(block of addr)
    ssd_program(all pages of block_copy)
```

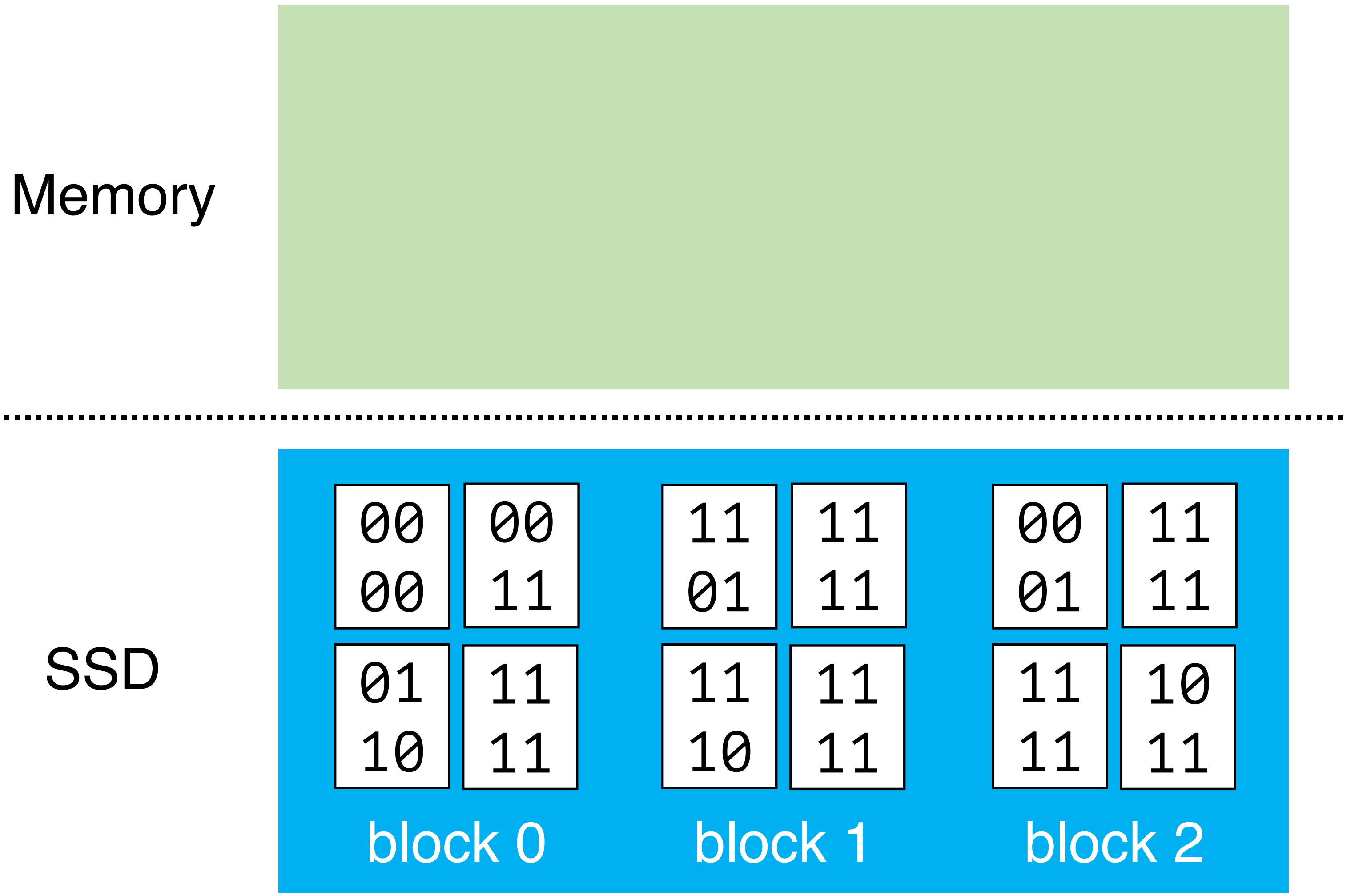
Awkward SSD Write



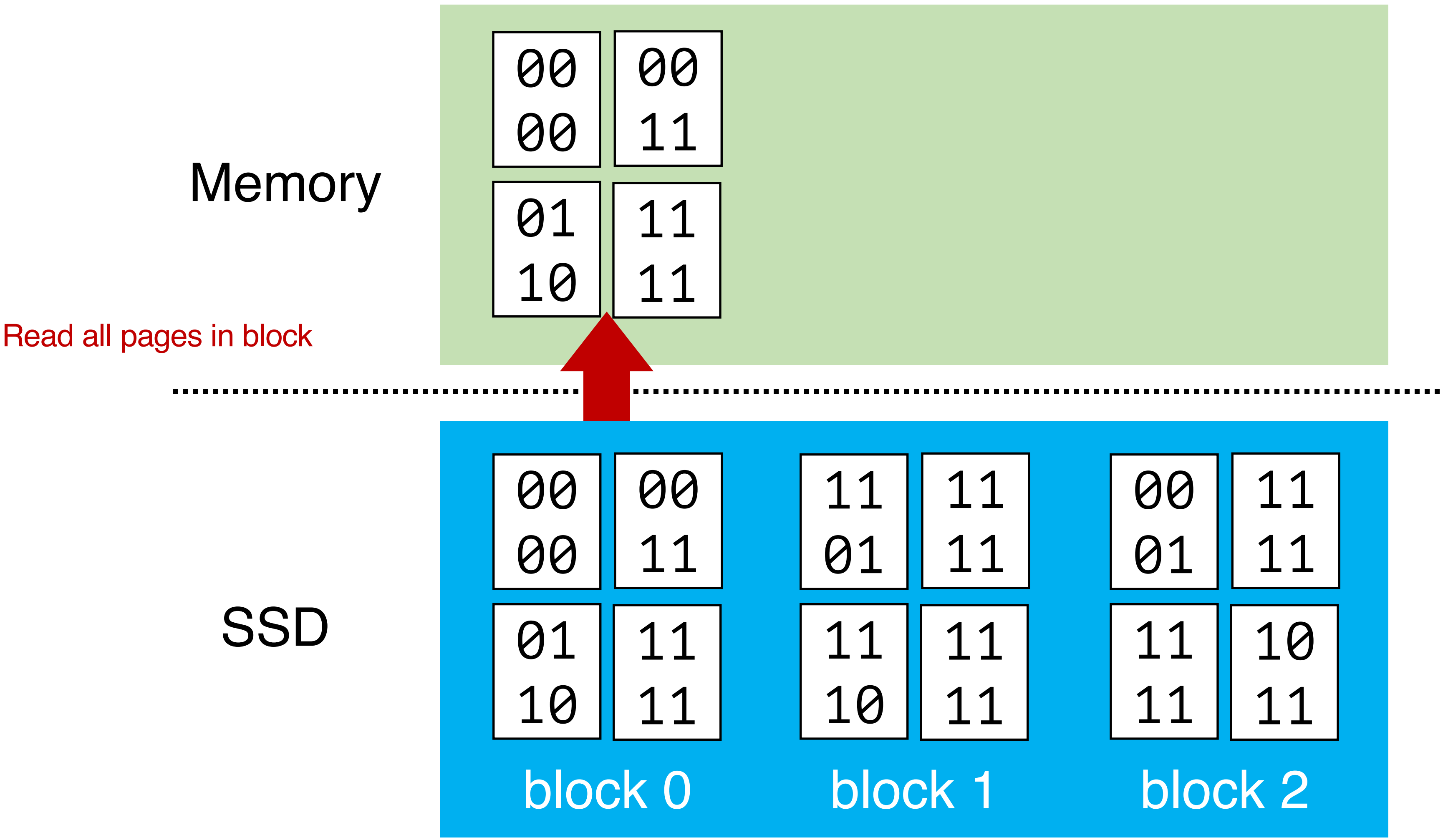
Awkward SSD Write



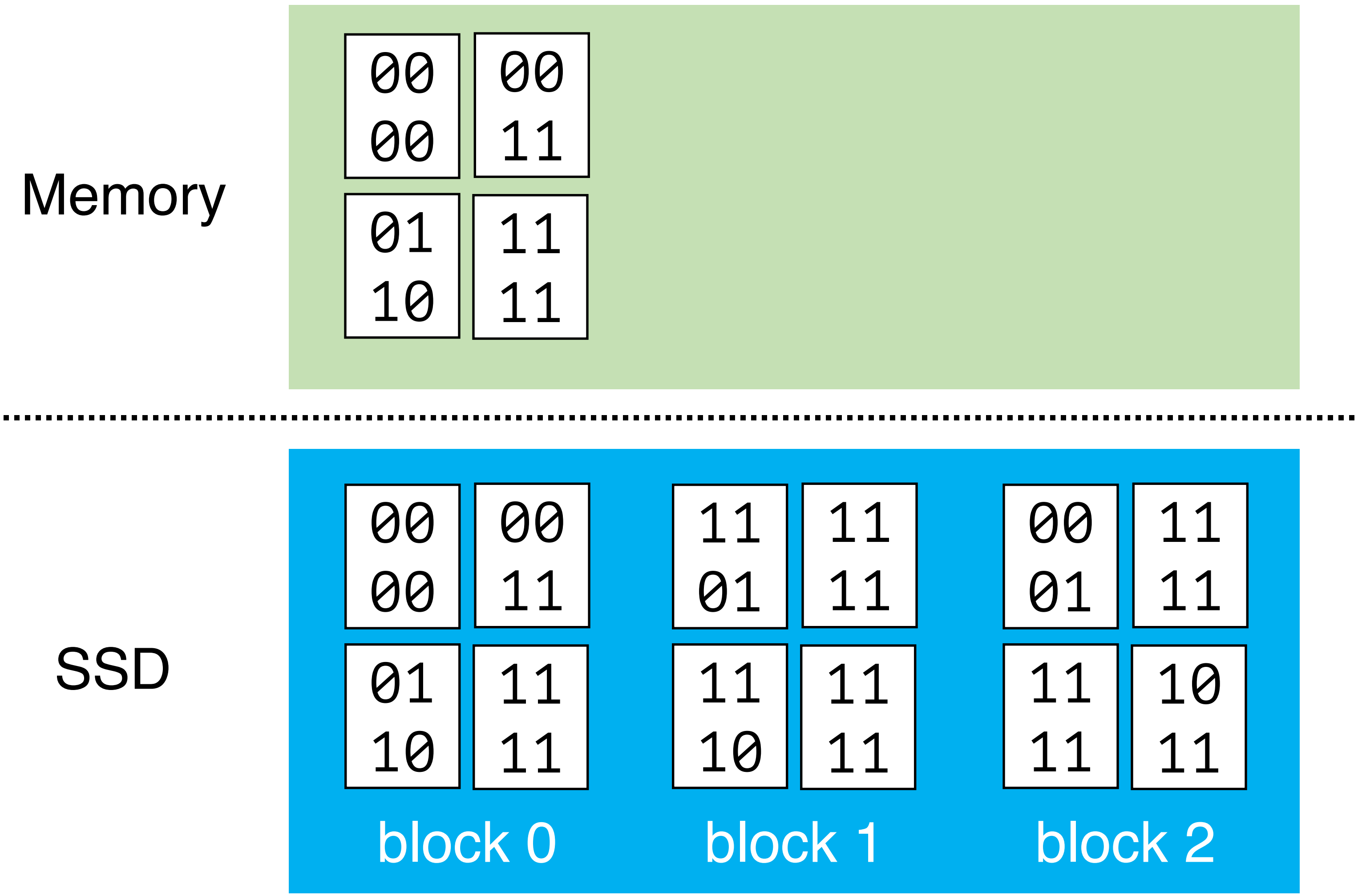
Awkward SSD Write



Awkward SSD Write



Awkward SSD Write

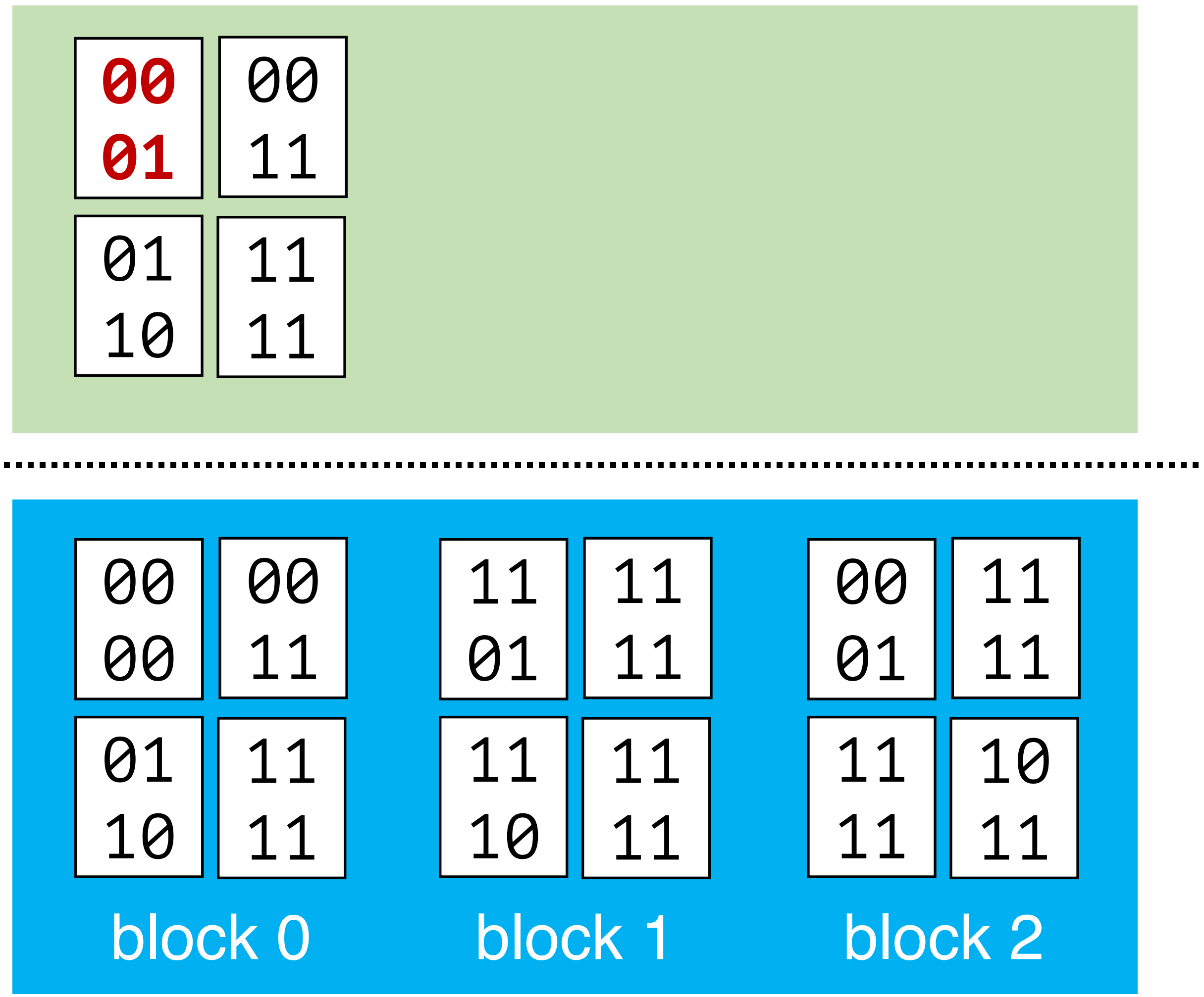


Awkward SSD Write

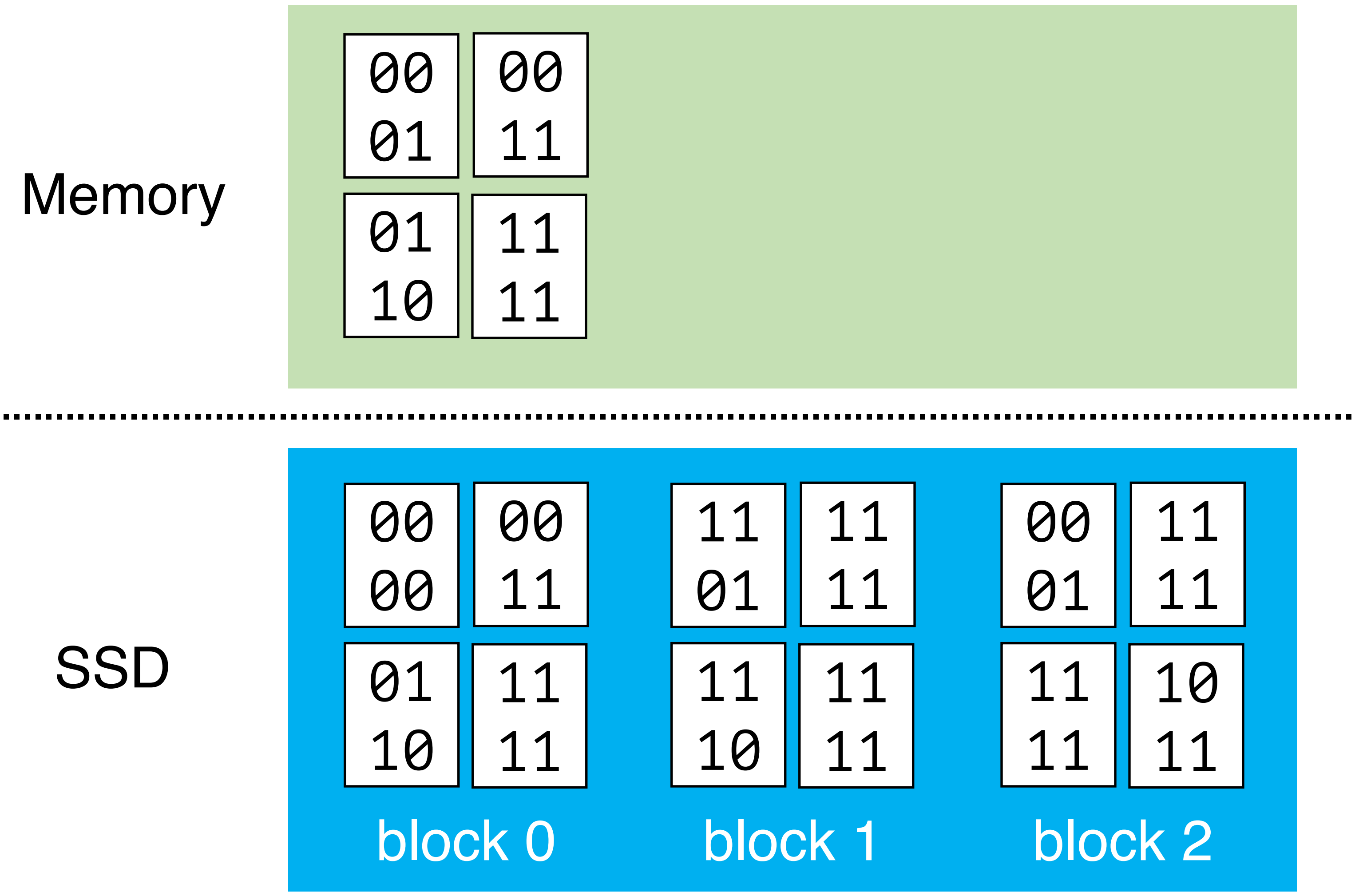
Modify target page in memory

Memory

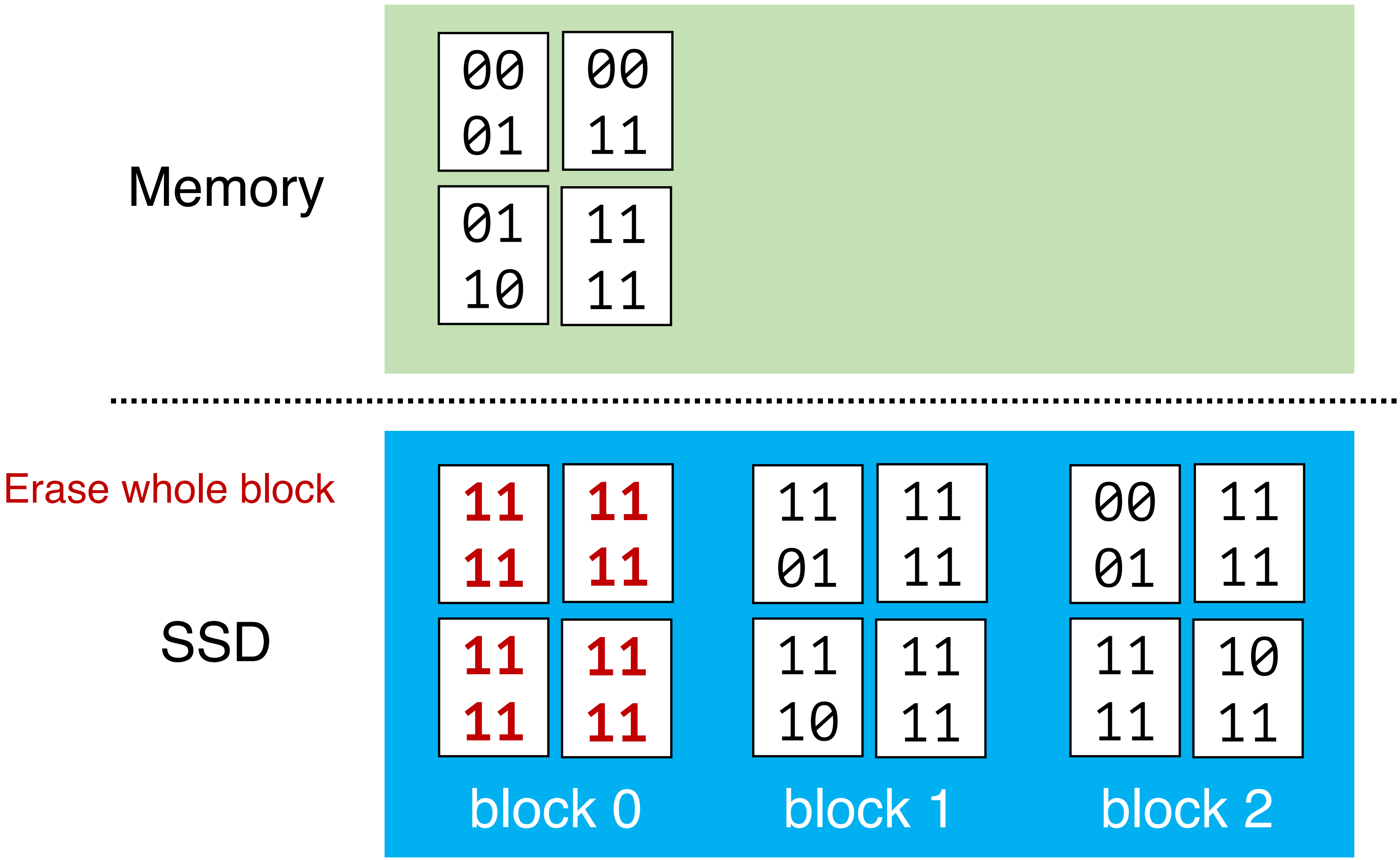
SSD



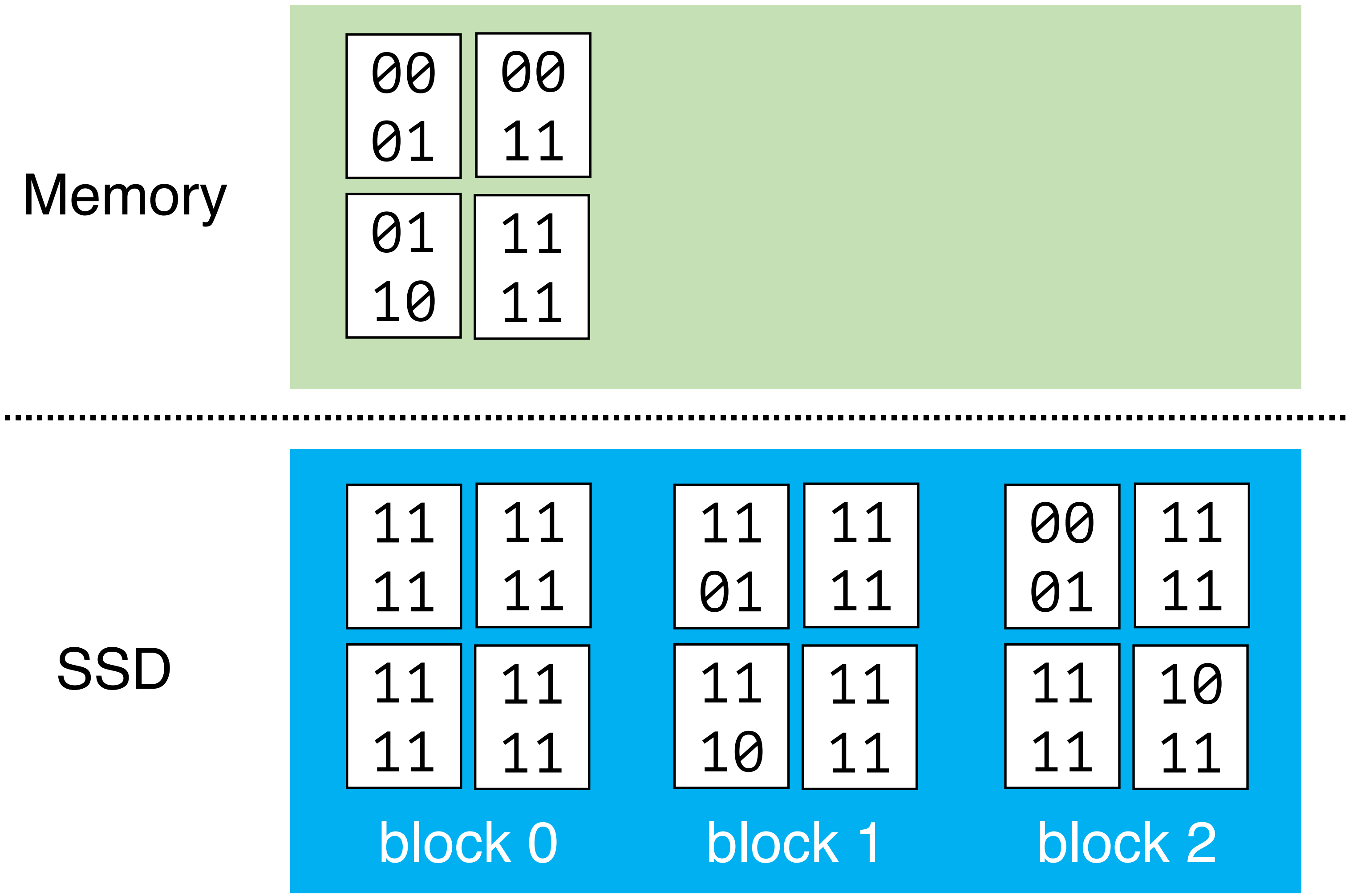
Awkward SSD Write



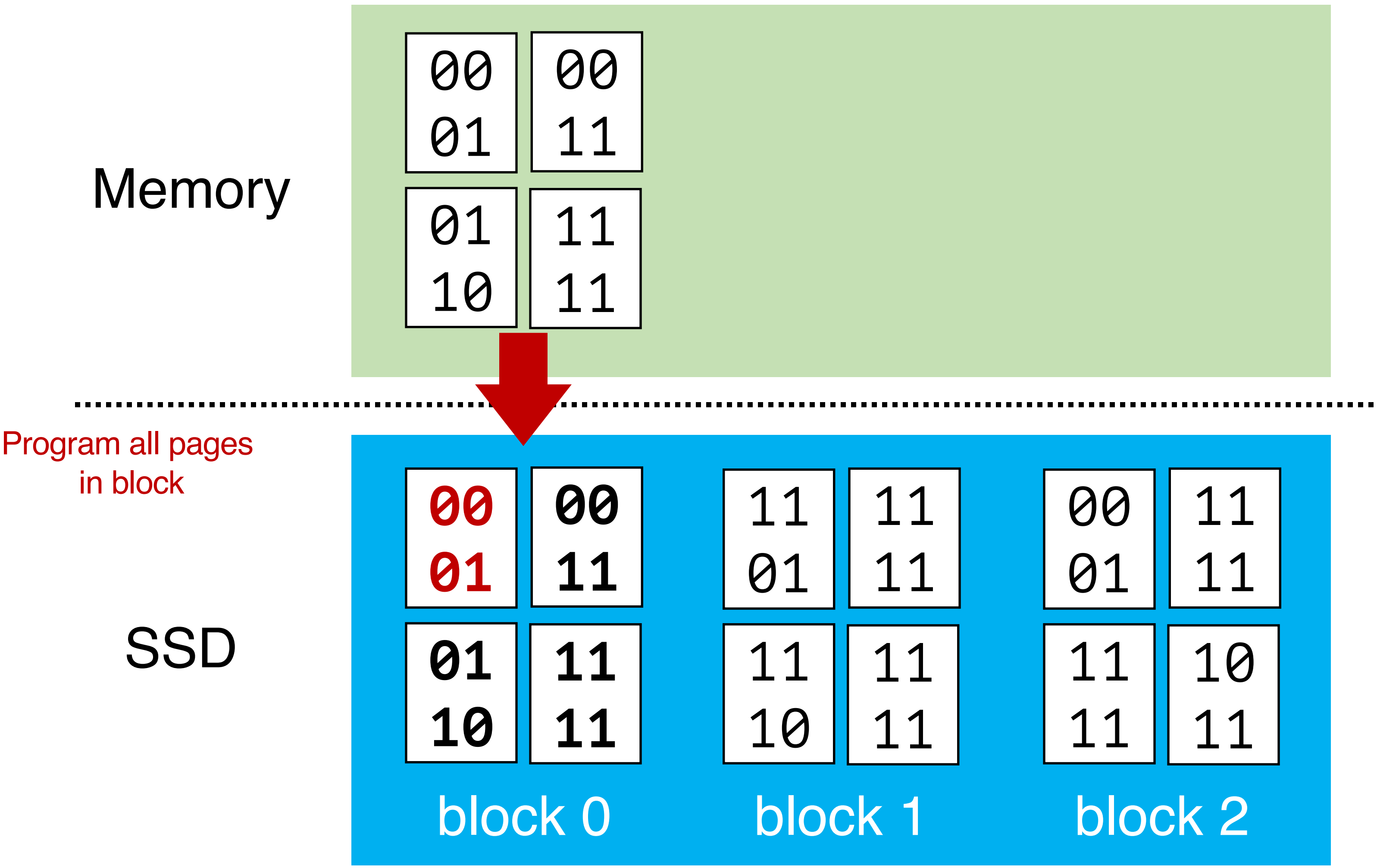
Awkward SSD Write



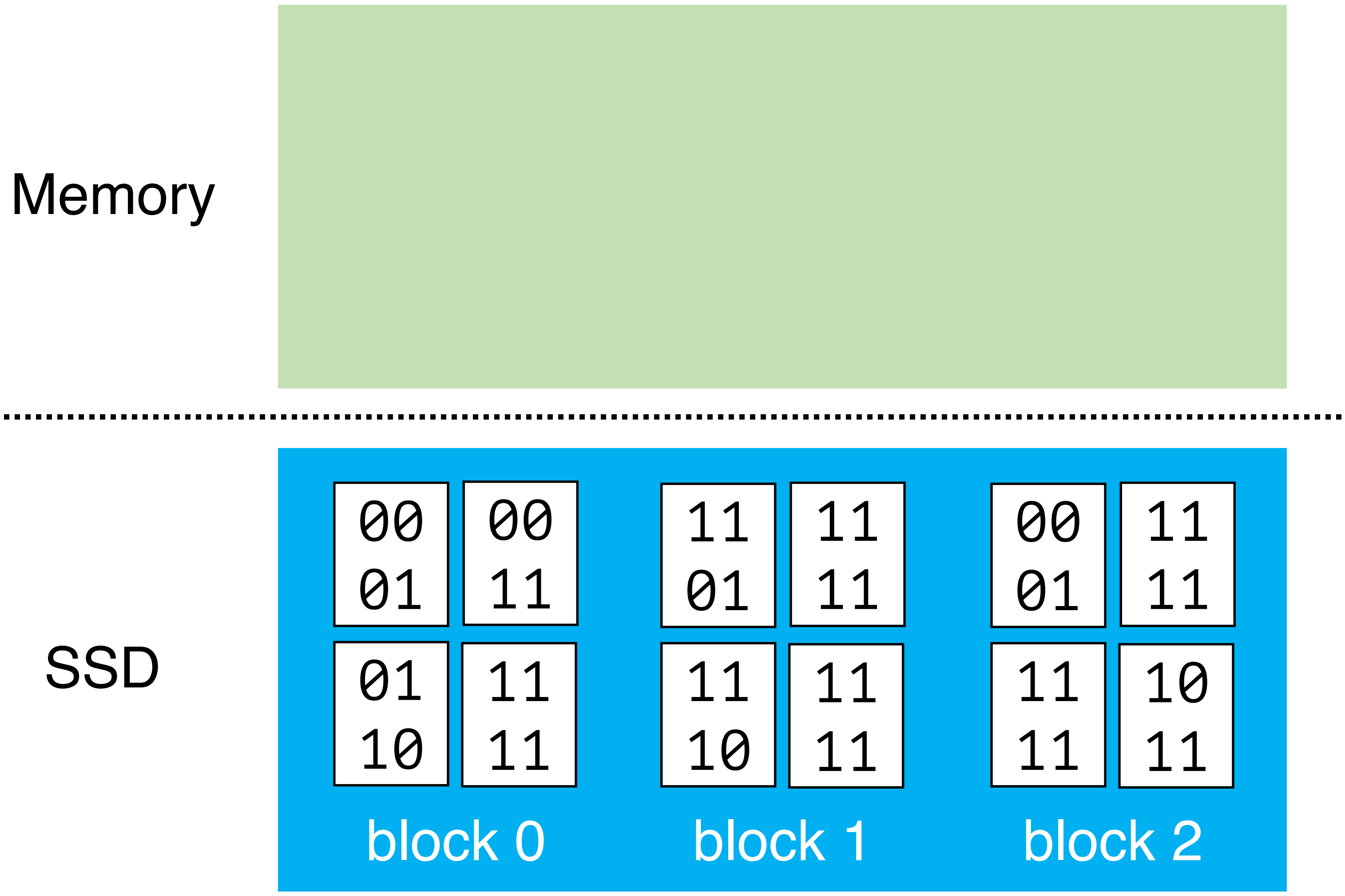
Awkward SSD Write



Awkward SSD Write



Awkward SSD Write



Issue: Write Amplification

- Random writes are expensive for flash!

- Writing one 4KB page may cause:
  - `read`, `erase`, and `program` of the whole 256KB block

Flash Translation Layer (FTL)

Flash Translation Layer (FTL)

- Add an address translation layer between upper-level file system and lower-level flash
  - Translate logical device addresses to physical addresses
  - Convert in-place write into append-write (log-structured)
  - Essentially, a virtualization & optimization layer

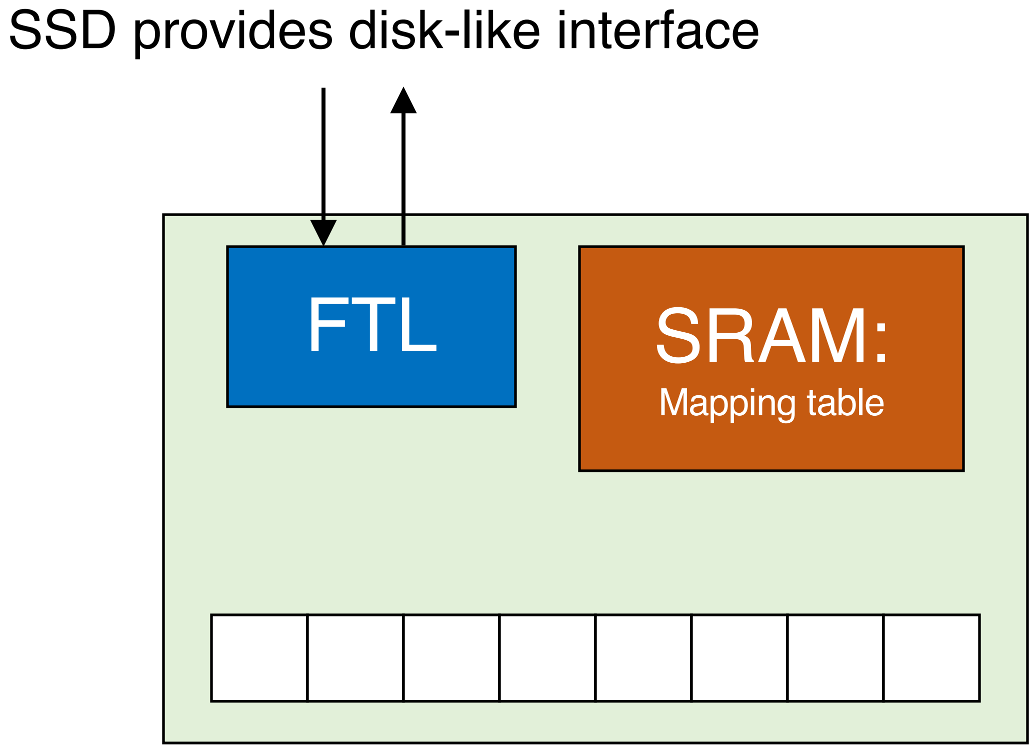
Flash Translation Layer (FTL)

- Usually implemented in SSD device's firmware (hardware)
  - But is also implemented in software for some SSDs

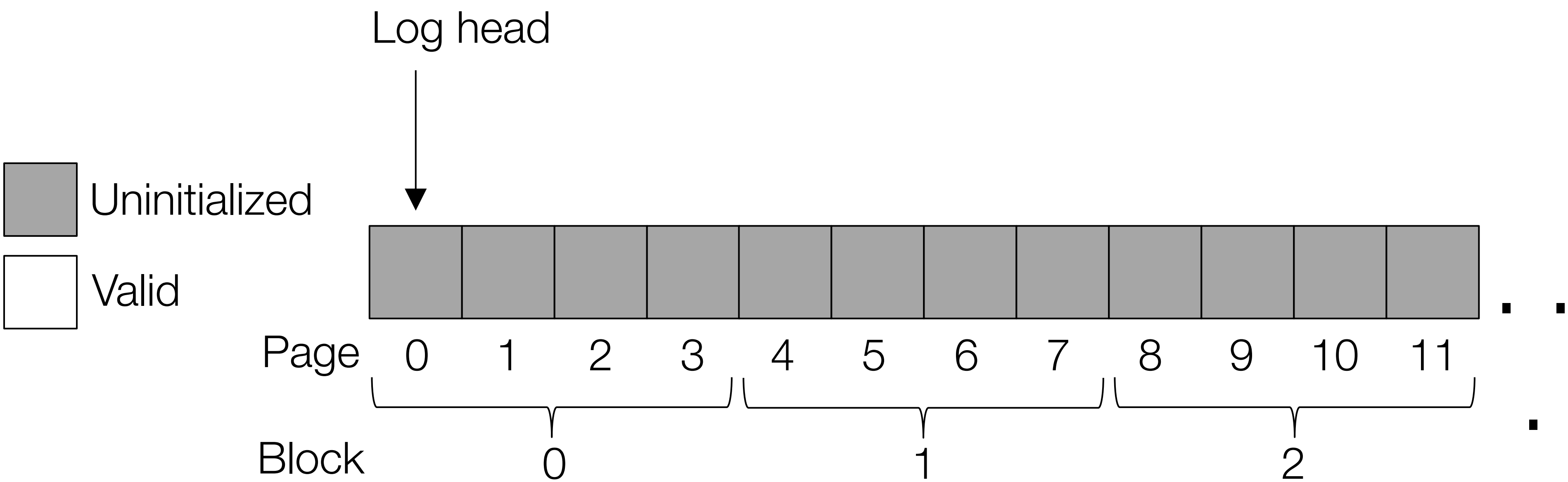
- Where to store mapping?
  - SRAM

- Physical pages can be in three states
  - uninitialized, valid, invalid

SSD Architecture with FTL



Logical-to-physical map

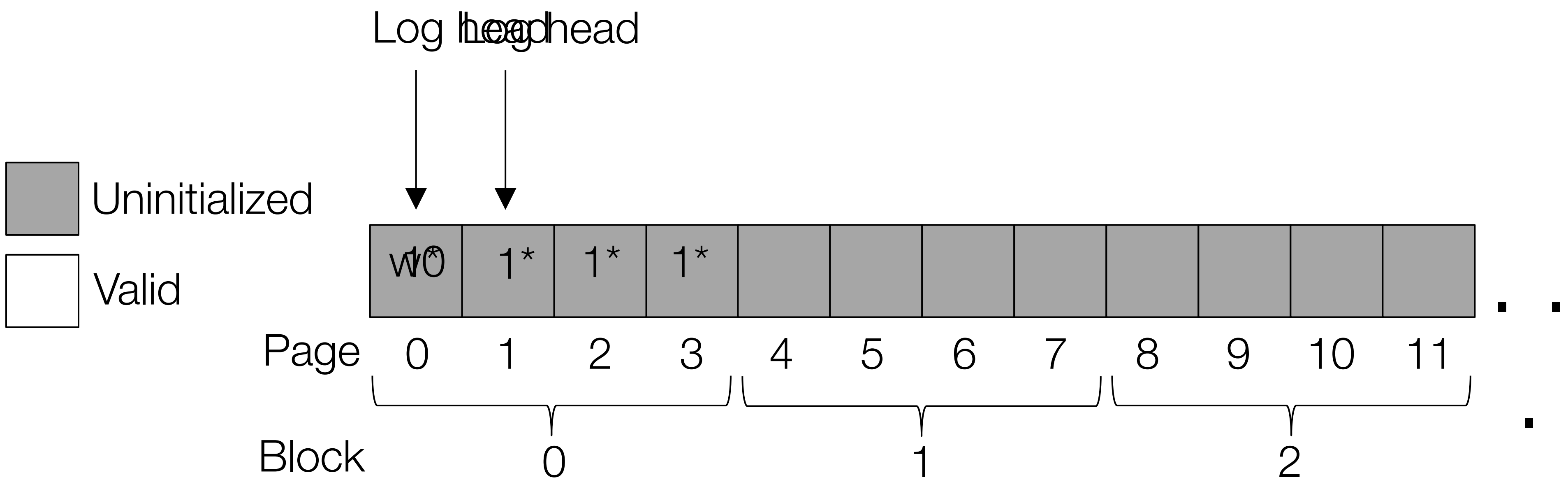


write(page=92, data=w0)

- erase(block0)
- program(page0, w0)
- logHead++

Logical-to-physical map

92 --> 0



write(page=92, data=w0)

- erase(block0)
- program(page0, w0)
- logHead++

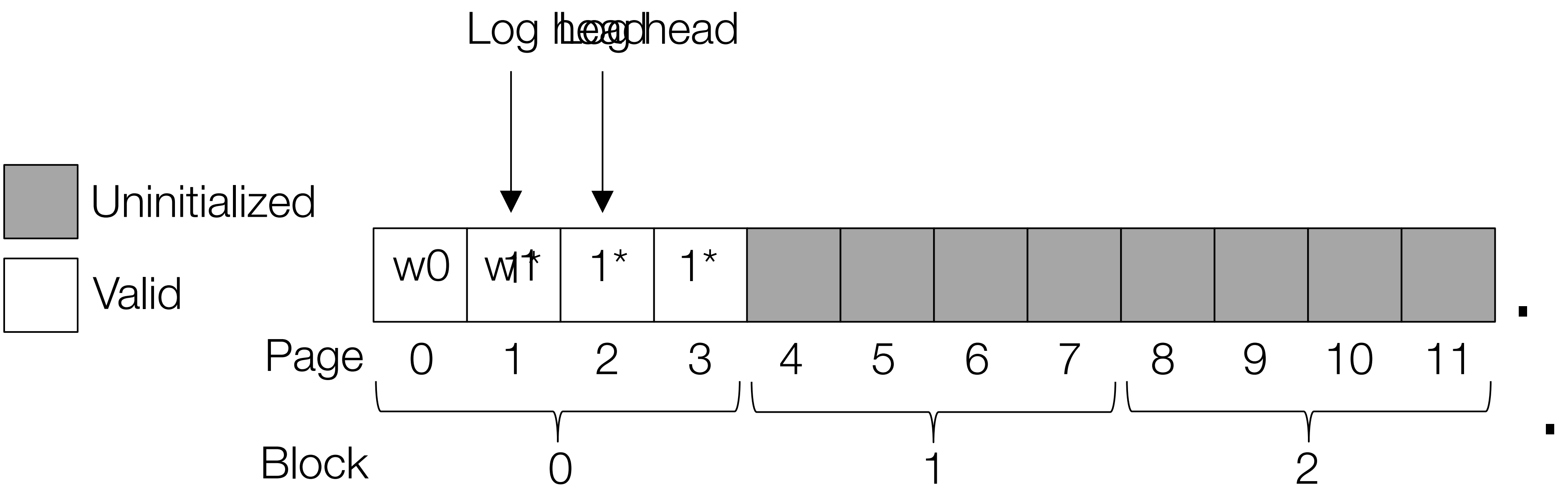
write(page=17, data=w1)

- program(page1, w1)
- logHead++

Logical-to-physical map

92 --> 0

17 --> 1



write(page=92, data=w0)
- erase(block0)
- program(page0, w0)
- logHead++

write(page=17, data=w1)
- program(page1, w1)
- logHead++

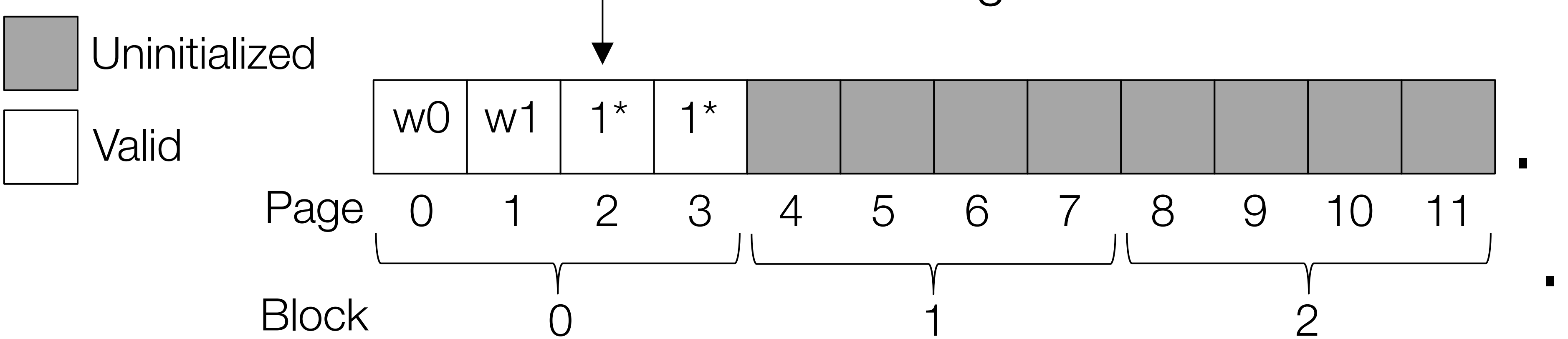
Logical-to-physical map

92 --> 0

17 --> 1

Advantages w.r.t. direct mapping

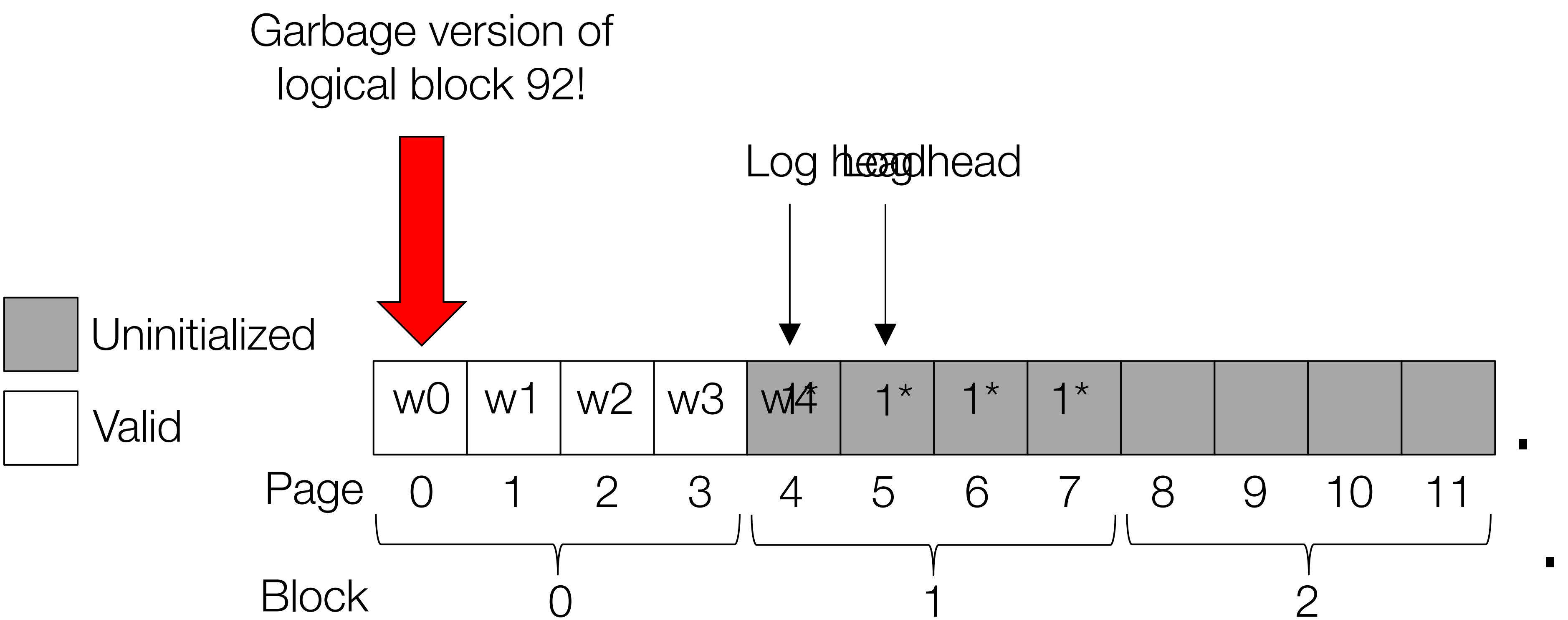
- Avoids expensive read-modify-write behavior
- Better wear levelling: writes get spread across pages, even if there is spatial locality in writes at logical level



write(page=92, data=w4)
- erase(block1)
- program(page4, w4)
- logHead++

Logical-to-physical map

~~92 --> 0~~ 92 --> 4
17 --> 1
33 --> 2
68 --> 3

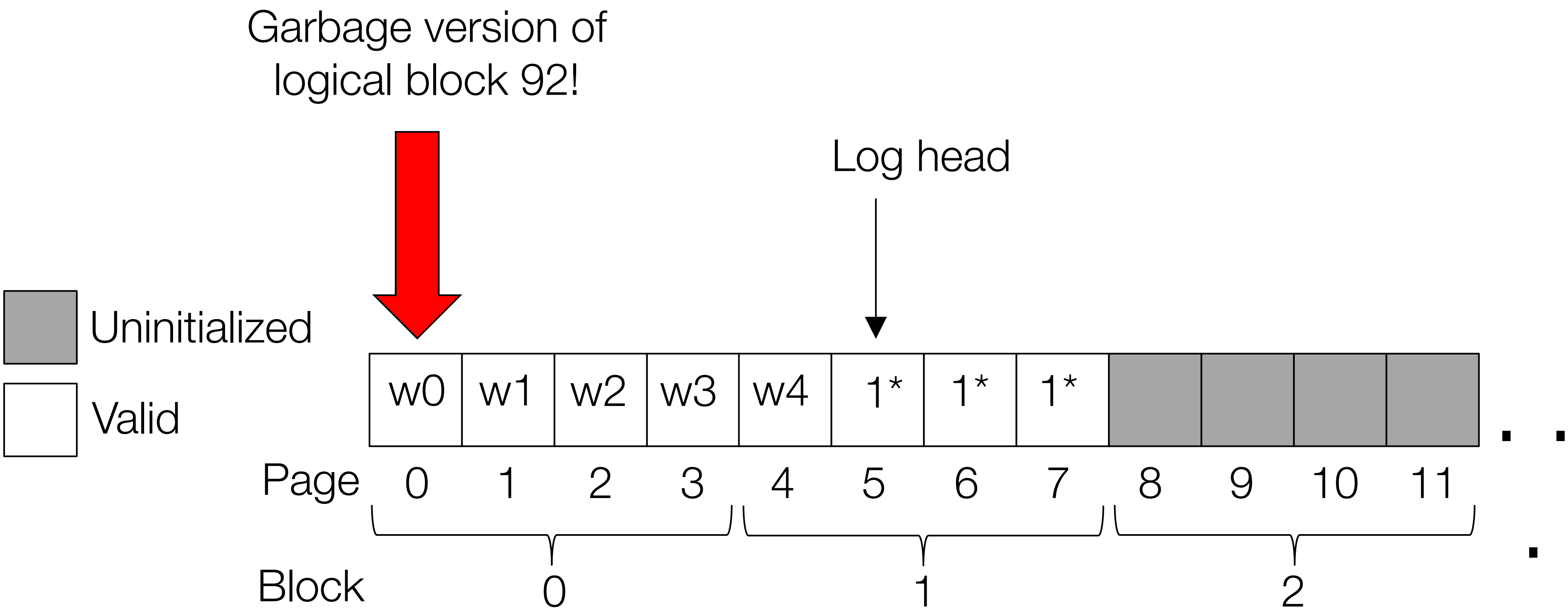


At some point, FTL must:

- Read all pages in physical block 0
- Write out the second, third, and fourth pages to the end of the log
- Update logical-to-physical map

<u>Logical-to-physical map</u>

~~92 --> 0~~ 92 --> 4
17 --> 1
33 --> 2
68 --> 3



Trash Day is the Worst Day

- Garbage collection requires extra read+write traffic
- Overprovisioning makes GC less painful
  - SSD exposes a logical page space that is smaller than the physical page space
  - By keeping extra, “hidden” pages around, the SSD tries to defer GC to a background task (thus removing GC from critical path of a write)
- SSD will occasionally shuffle live (i.e., non-garbage) blocks that never get overwritten
  - Enforces wear leveling

File System Abstraction

What is a File?

- File: Array of bytes
  - Ranges of bytes can be read/written

- File system (FS) consists of many files

- Files need names so programs can choose the right one

File Names

- Three types of names (abstractions)
  - inode (low-level names)
  - path (human readable)
  - file descriptor (runtime state)

Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes

Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes
- Show inodes via `stat`
  - `$ stat <file or dir>`

'stat' Example

```
PROMPT>: stat test.dat
File: 'test.dat'  Size: 5     Blocks: 8     IO Block: 4096   regular
file
Device: 803h/2051d        Inode: 119341128   Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1001/     yue)   Gid: ( 1001/     yue)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2015-12-17 04:12:47.935716294 -0500
Modify: 2014-12-12 19:25:32.669625220 -0500
Change: 2014-12-12 19:25:32.669625220 -0500
Birth: -
```

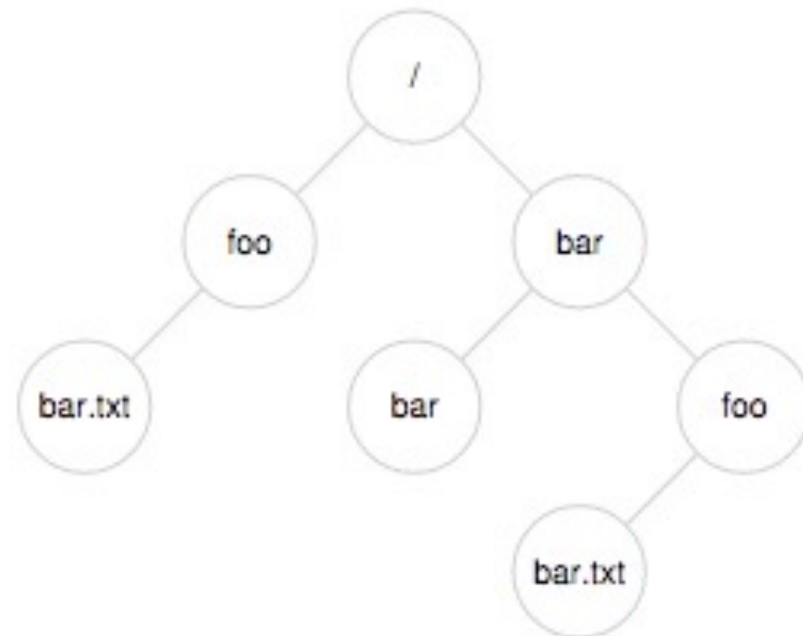
Path (multiple directories)

- A directory is a file
  - Associated with an inode

- Contains a list of <user-readable name, low-level name> pairs

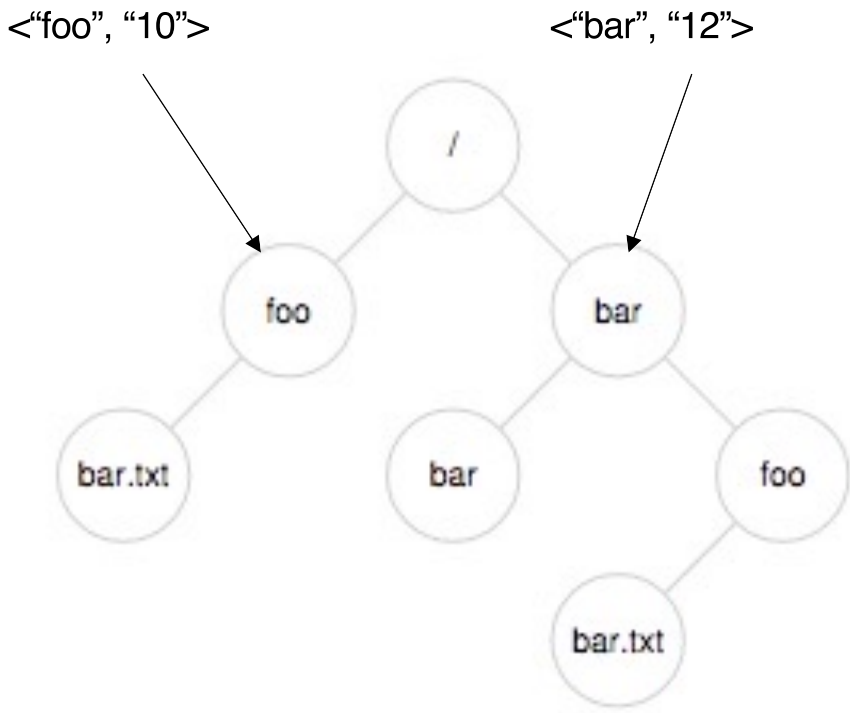
Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs



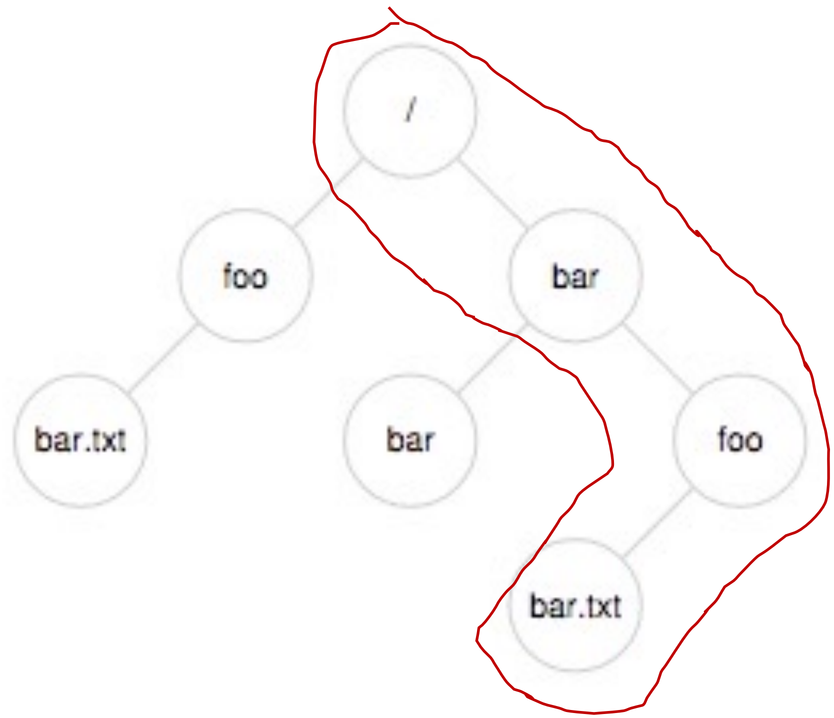
Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs



Path (multiple directories)

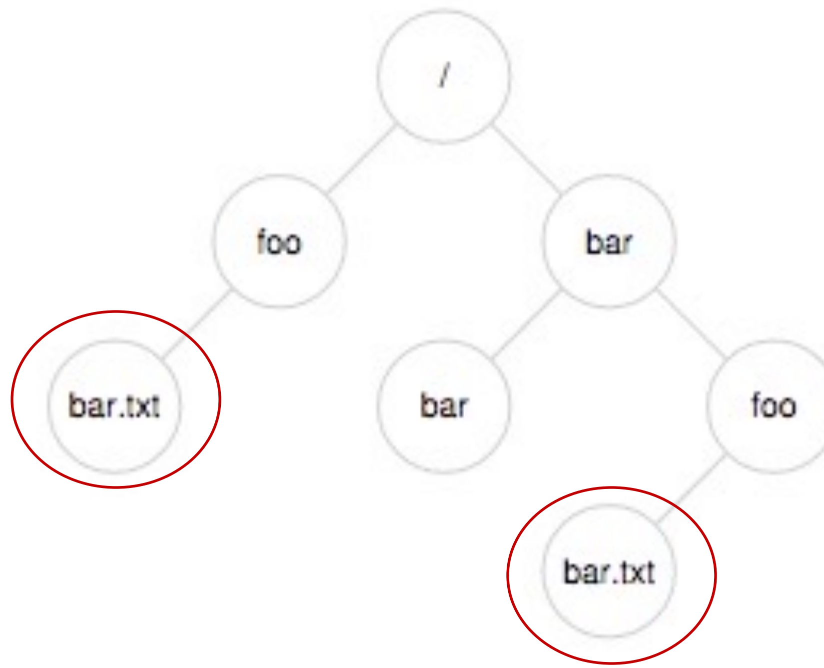
- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs
- Directory tree: reads for getting final inode called traversal



[traverse /bar/foo/bar.txt]

File Naming

- Directories and files can have the same name as long as they are in different locations of the file-system tree
- .txt, .c, etc.
  - Naming convention
  - In UNIX-like OS, no enforcement for extension name



Special Directory Entries

```
prompt> ls -al
total 216
drwxr-xr-x   19 yue   staff   646 Nov 23 16:28 .
drwxr-xr-x+  40 yue   staff  1360 Nov 15 01:41 ..
-rw-r--r--@   1 yue   staff  1064 Aug 29 21:48 common.h
-rwxr-xr-x    1 yue   staff  9356 Aug 30 14:03 cpu
-rw-r--r--@   1 yue   staff   258 Aug 29 21:48 cpu.c
-rwxr-xr-x    1 yue   staff  9348 Sep  6 12:12 cpu_bound
-rw-r--r--    1 yue   staff   245 Sep  5 13:10 cpu_bound.c
...
```

File System Interfaces

Creating Files

- UNIX system call: open( )

```
int fd = open(char *path, int flag, mode_t mode);

-OR-

int fd = open(char *path, int flag);
```

File Descriptor (`fd`)

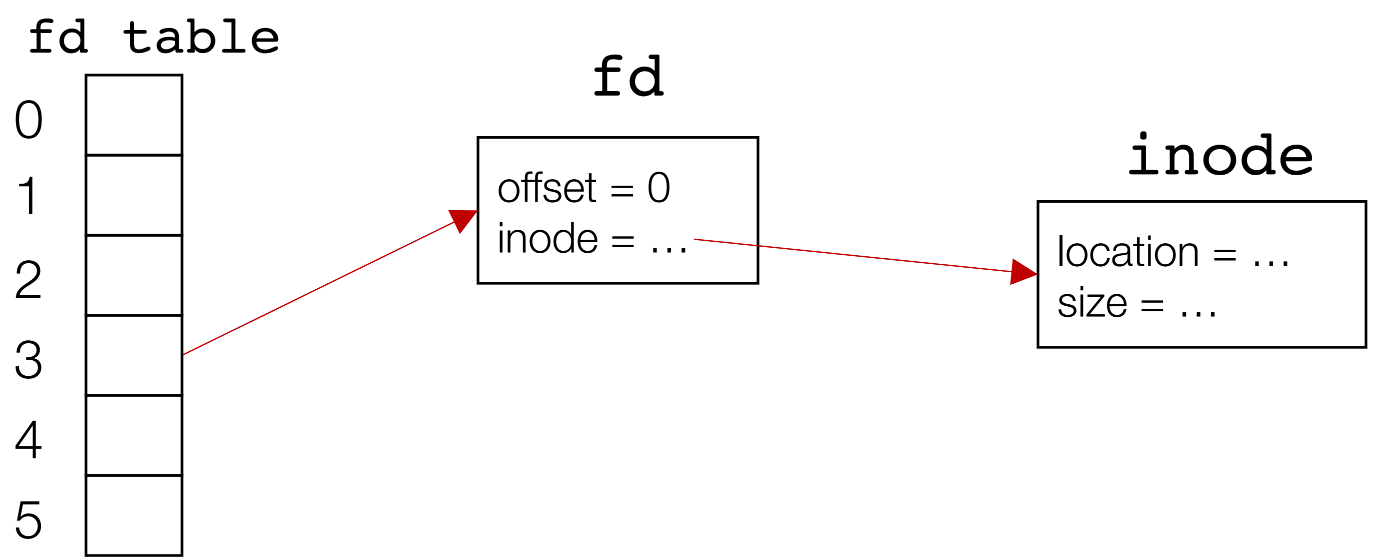
- `open()` returns a file descriptor (`fd`)
  - A `fd` is an integer
  - Private per process
- An opaque handle that gives caller the power to perform certain operations
- Think of a `fd` as a pointer to an object of the file
  - By owning such an object, you can call other "methods" to access the file

open() Example

```
int fd1 = open(“file.txt”, O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open(“file.txt”, O_WRONLY); // return 4
int fd3 = dup(fd2);                   // return 5
```

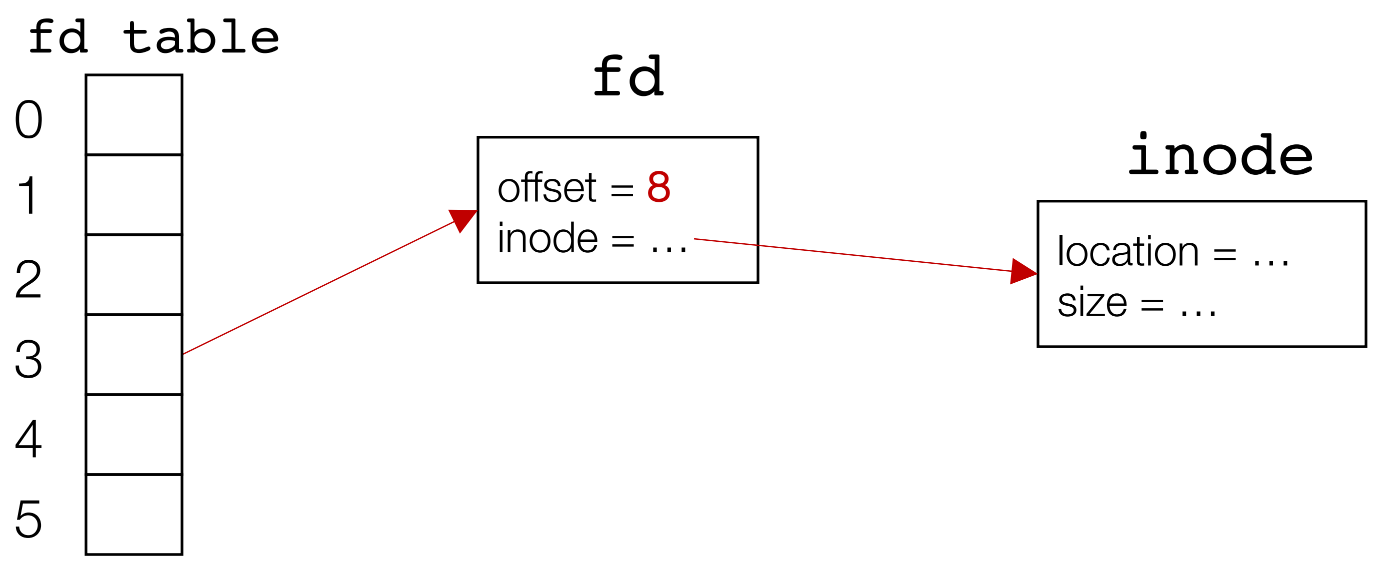
open( ) Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
```



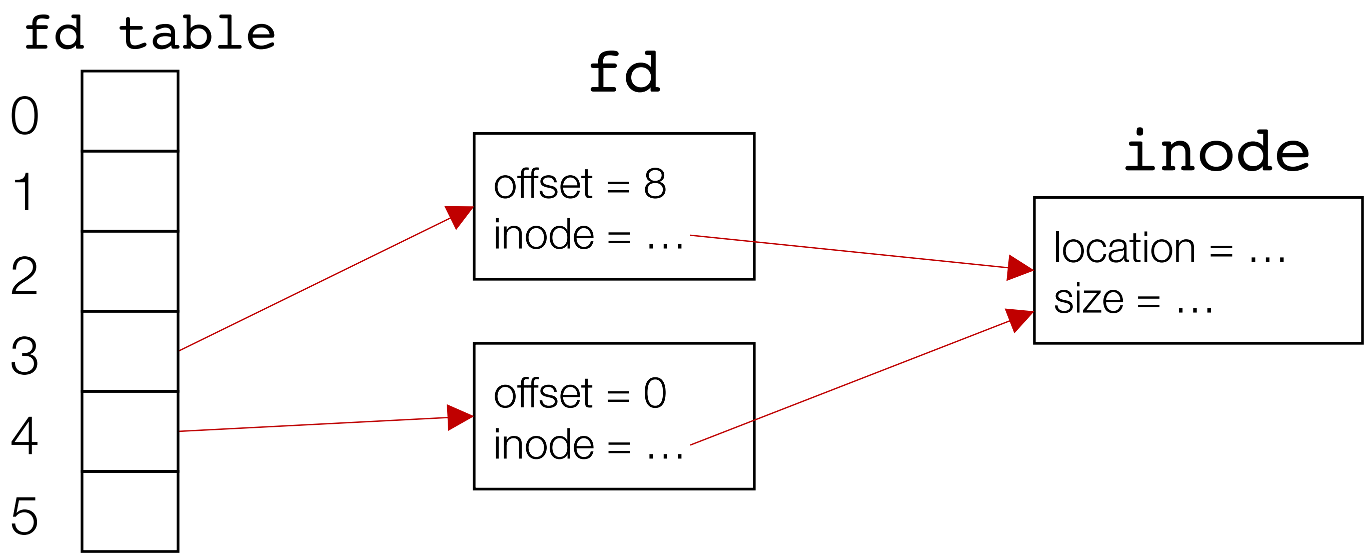
open( ) Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
```



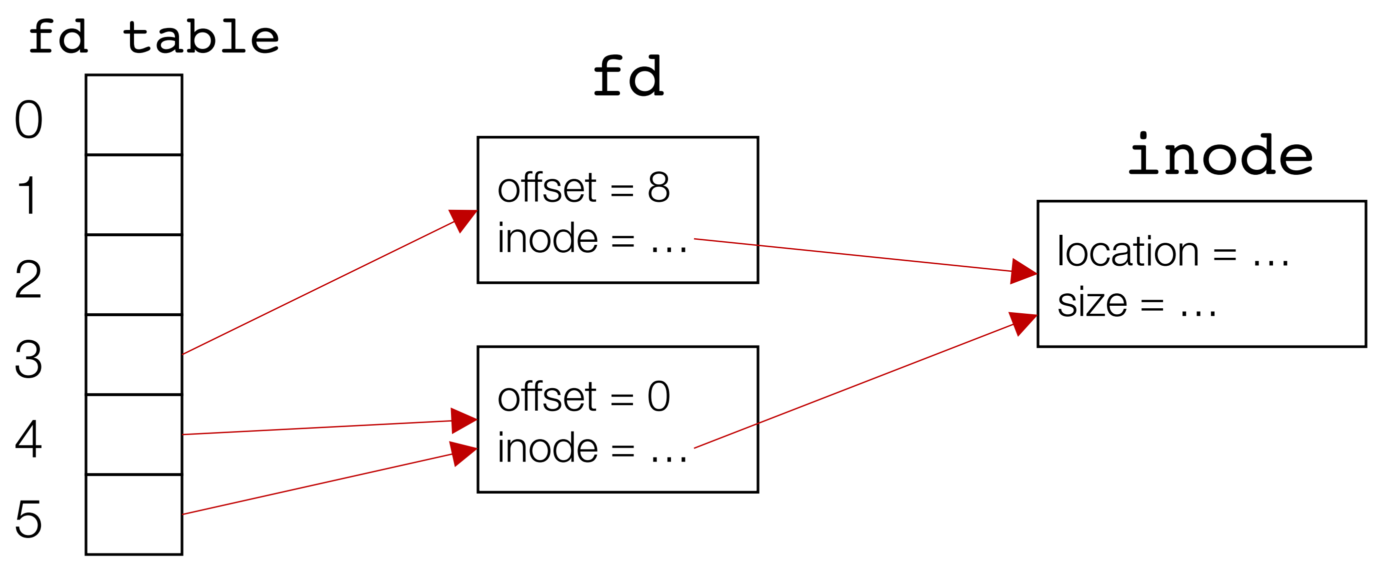
open( ) Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open("file.txt", O_WRONLY); // return 4
```



open() Example

```
int fd1 = open("file.txt", O_CREAT);  // return 3
read(fd1, buf, 8);
int fd2 = open("file.txt", O_WRONLY); // return 4
int fd3 = dup(fd2);                   // return 5
```



UNIX File Read and Write APIs

```
int fd = open(char *path, int flag, mode_t mode);
-OR-
int fd = open(char *path, int flag);

ssize_t sz = read(int fd, void *buf, size_t count);

ssize_t sz = write(int fd, void *buf, size_t count);

int ret = close(int fd);
```

Reading and Writing Files

```
prompt> echo hello > file.txt
prompt> cat file.txt
hello
prompt>
```

Reading and Writing Files

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)              = 3
read(3, "hello\n", 65536)               = 6
write(1, "hello\n", 6)                  = 6
read(3, "", 65536)                      = 0
close(3)                                = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)            = 3
read(3, "hello\n", 65536)             = 6
write(1, "hello\n", 6)                = 6
read(3, "", 65536)                    = 0
close(3)                              = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)           = 3
read(3, "hello\n", 65536)            = 6
write(1, "hello\n", 6)               = 6
read(3, "", 65536)                   = 0
close(3)                             = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)             = 3
read(3, "hello\n", 65536)              = 6
write(1, "hello\n", 6)                 = 6
read(3, "", 65536)                     = 0
close(3)                               = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

`cat` tries to read more but reaches EOF

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)          = 3
read(3, "hello\n", 65536)           = 6
write(1, "hello\n", 6)              = 6
read(3, "", 65536)                  = 0
close(3)                            = 0
...
prompt>
```

Reading and Writing Files

Open the file with read only mode

Read content from file

Write string to std output `fd` 1

`cat` tries to read more but reaches EOF

`cat` done with file ops and closes the file

```
prompt> strace cat file.txt
...
open("file.txt", O_RDONLY)             = 3
read(3, "hello\n", 65536)              = 6
write(1, "hello\n", 6)                 = 6
read(3, "", 65536)                     = 0
close(3)                               = 0
...
prompt>
```

Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

whence:

- If `whence` is `SEEK_SET`, the offset is set to `offset` bytes
- If `whence` is `SEEK_CUR`, the offset is set to its current location plus `offset` bytes
- If `whence` is `SEEK_END`, the offset is set to the size of the file plus `offset` bytes

Non-Sequential File Operations

```
off_t offset = lseek(int fd, off_t offset, int whence);
```

whence:

- If `whence` is `SEEK_SET`, the offset is set to `offset` bytes
- If `whence` is `SEEK_CUR`, the offset is set to its current location plus `offset` bytes
- If `whence` is `SEEK_END`, the offset is set to the size of the file plus `offset` bytes

Note: Calling `lseek()` does not perform a disk seek!

Writing Immediately with `fsync()`

```
int fd = fsync(int fd);
```

- `fsync(fd)` forces buffers to flush to disk, and (usually) tells the disk to flush its write cache too
  - To make the data durable and persistent
- Write buffering improves performance

Renaming Files

```
prompt> mv file.txt new_name.txt
```

Renaming Files

```
prompt> strace mv file.txt new_name.txt
...
rename("file.txt", "new_name.txt")   = 0
...
prompt>
```

Renaming Files

System call `rename()` atomically renames a file

```
prompt> strace mv file.txt new_name.txt
...
rename("file.txt", "new_name.txt")   = 0
...
prompt>
```

File Renaming Example

```
prompt> vim file.txt
```

```
int fd = open(".file.txt.swp",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
```

Using `vim` to edit a file and then save it

File Renaming Example

```
prompt> vim file.txt
… vim editing session …
```

```
int fd = open(".file.txt.swp",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file (editing…)
```

Using `vim` to edit a file and then save it

File Renaming Example

```
prompt> vim file.txt
… vim editing session …
prompt>
```

:wq

```
int fd = open(".file.txt.swp",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);               // make data durable
close(fd);               // close tmp file
rename(".file.txt.swp", "file.txt");// change name and replacing old file
```

Using `vim` to edit a file and then save it

Deleting Files

```
prompt> rm file.txt
```

Deleting Files

```
prompt> strace rm file.txt
...
unlink("file.txt")              = 0
...
prompt>
```

Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt")                = 0
...
prompt>
```

Deleting Files

System call `unlink()` is called to delete a file

```
prompt> strace rm file.txt
...
unlink("file.txt")                = 0
...
prompt>
```

Directories are deleted when `unlink()` is called

Q: File descriptors are deleted when ???