

[537] Flash

Tyler Harter
12/08/14

AFS Review

Whole-File Caching

Whole-file reads: most common workload.

Makes design simple:

- all reads and writes are to local FS
- AFS does most work on open/close

What about partial reads? Streaming I/O?

Cache Consistency

Update visibility:

- client has newer version
- AFS: flush whole file on close

Stale cache:

- server has newer version
- AFS: callbacks

Callbacks

Client 1



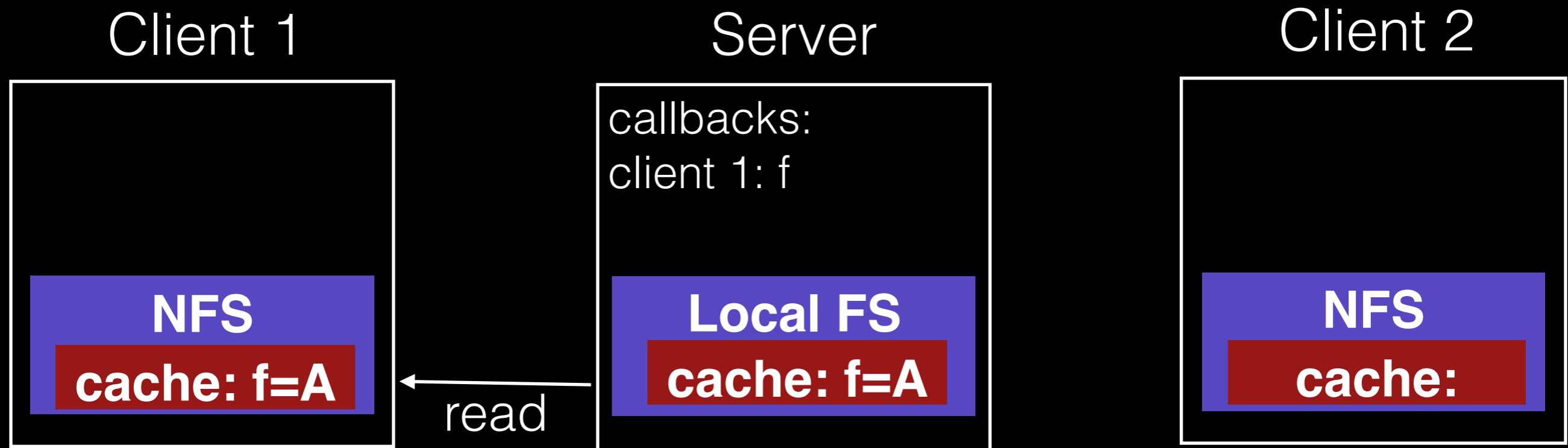
Server



Client 2



Callbacks

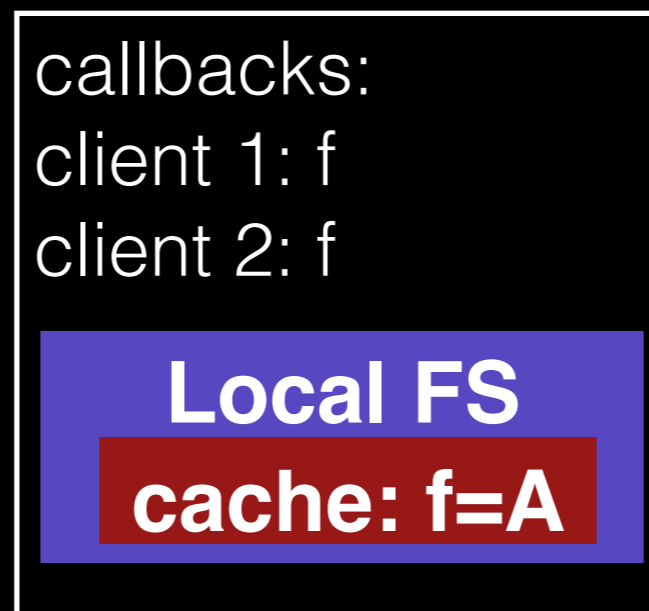


Callbacks

Client 1



Server



Client 2



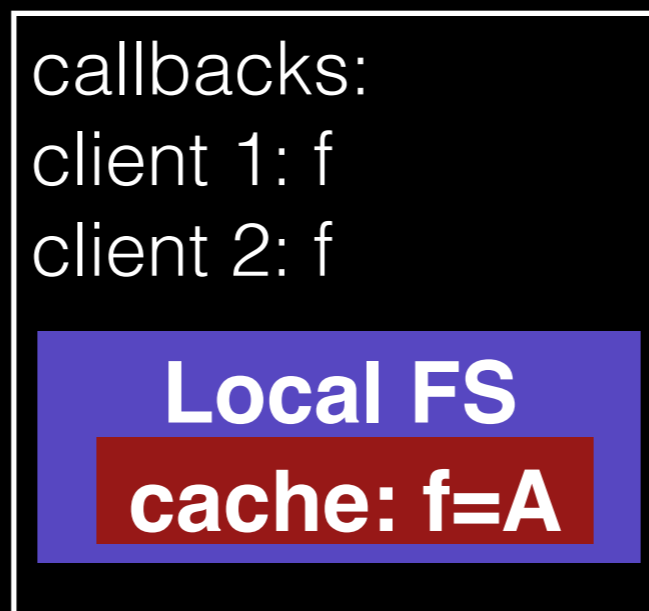
read

Callbacks

Client 1



Server



Client 2

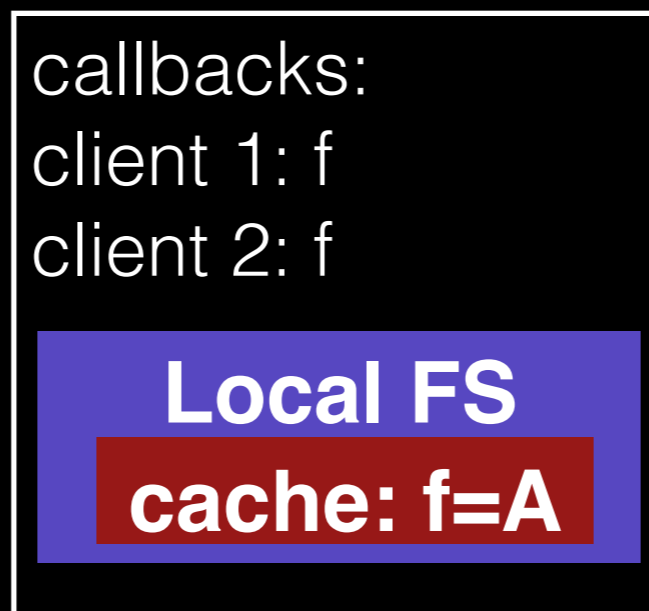


Callbacks

Client 1



Server

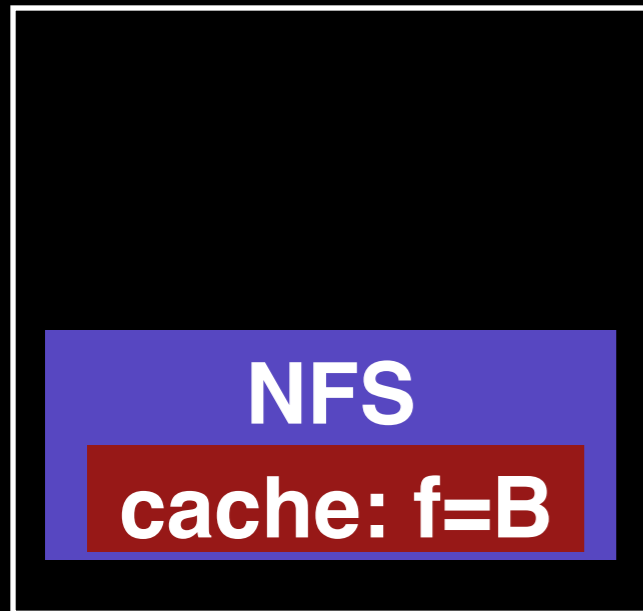


Client 2

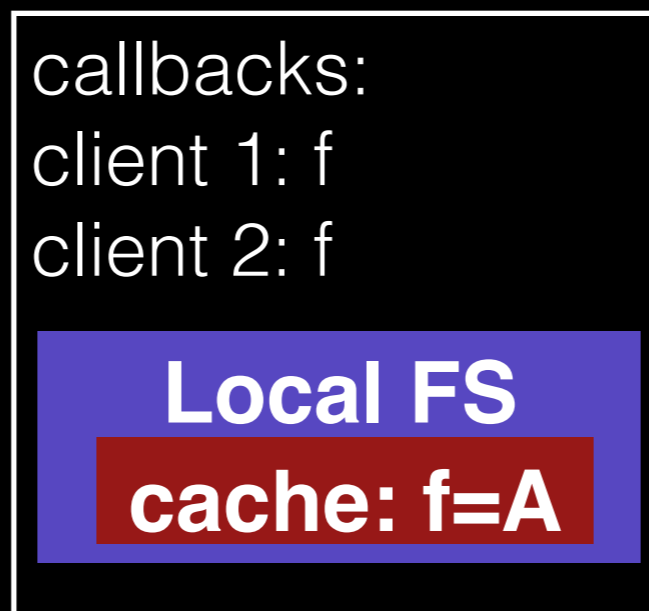


Callbacks

Client 1



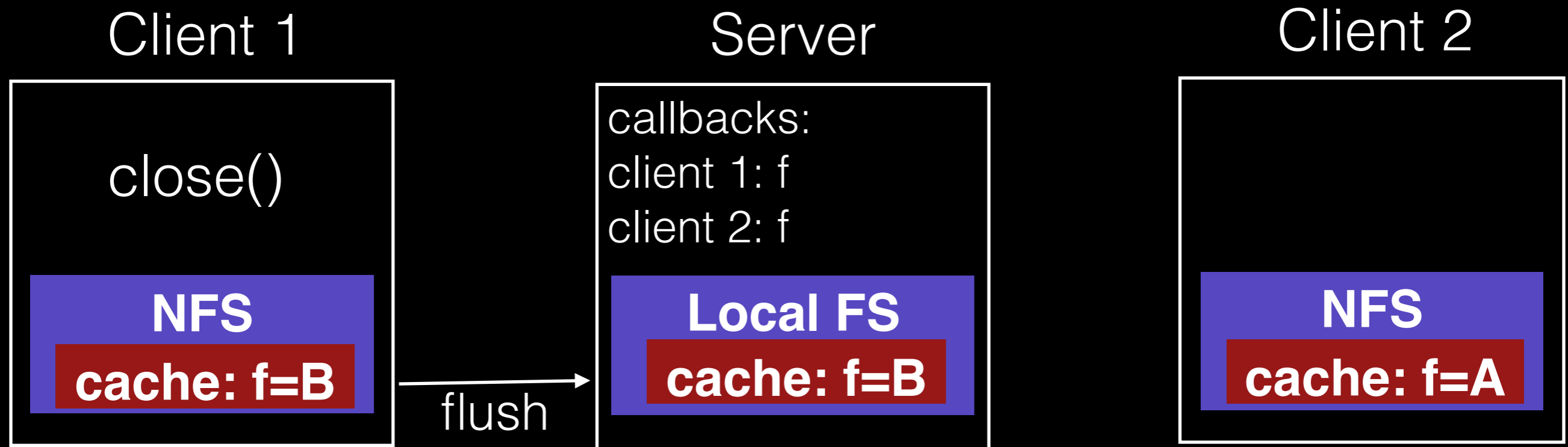
Server



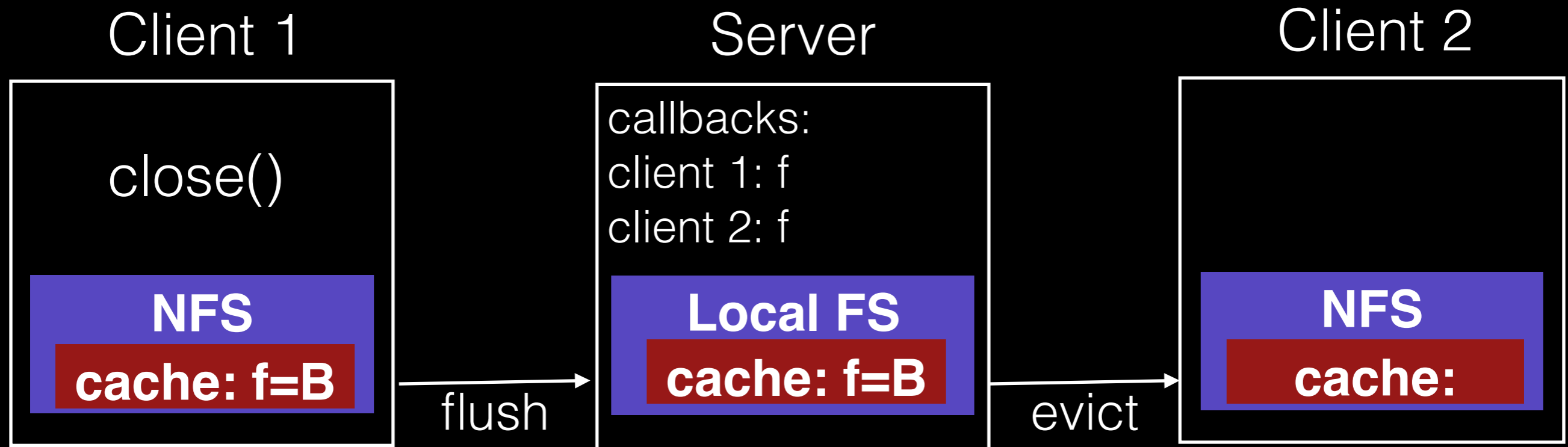
Client 2



Callbacks

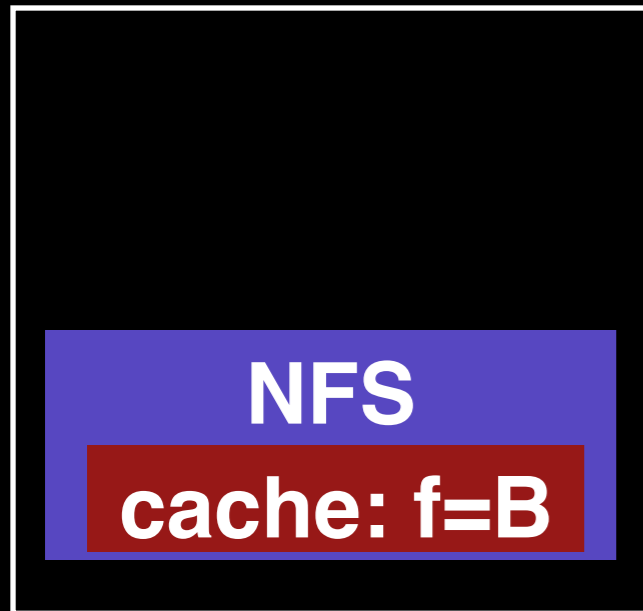


Callbacks

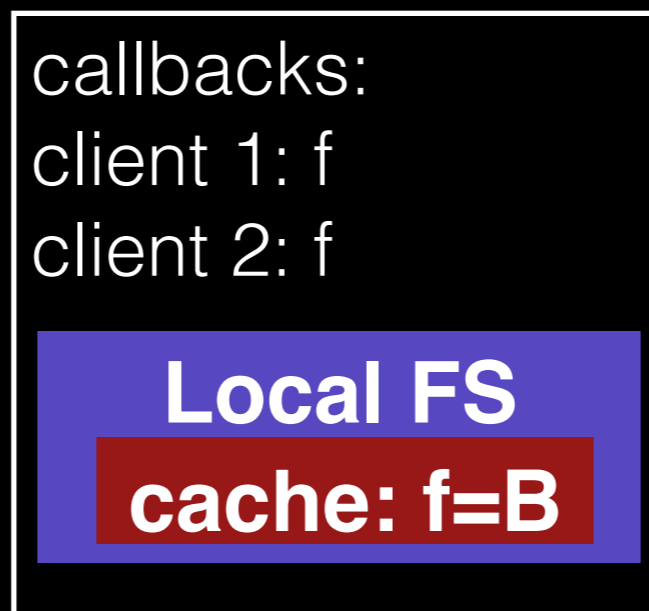


Callbacks

Client 1



Server



Client 2



Volume Migration

Strategy:

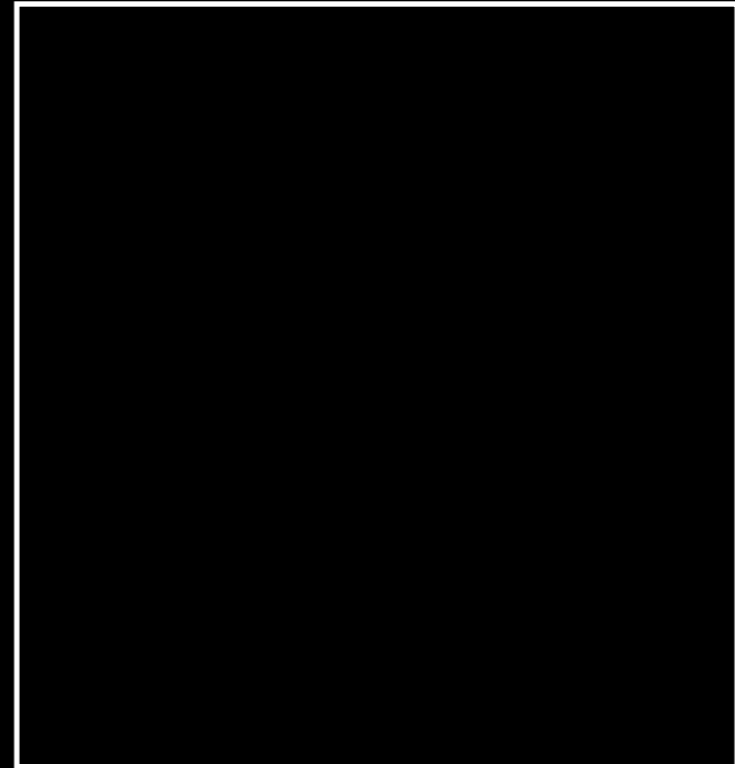
- do most copy with snapshot
- freeze while copying last changes
- leave forwarding address

Copy

Machine 1



Machine 2

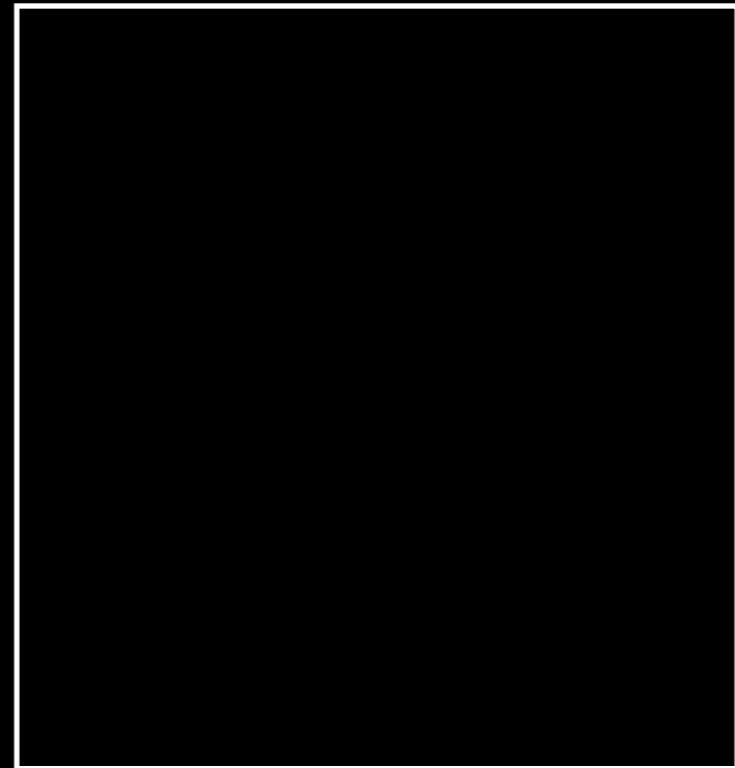


Copy

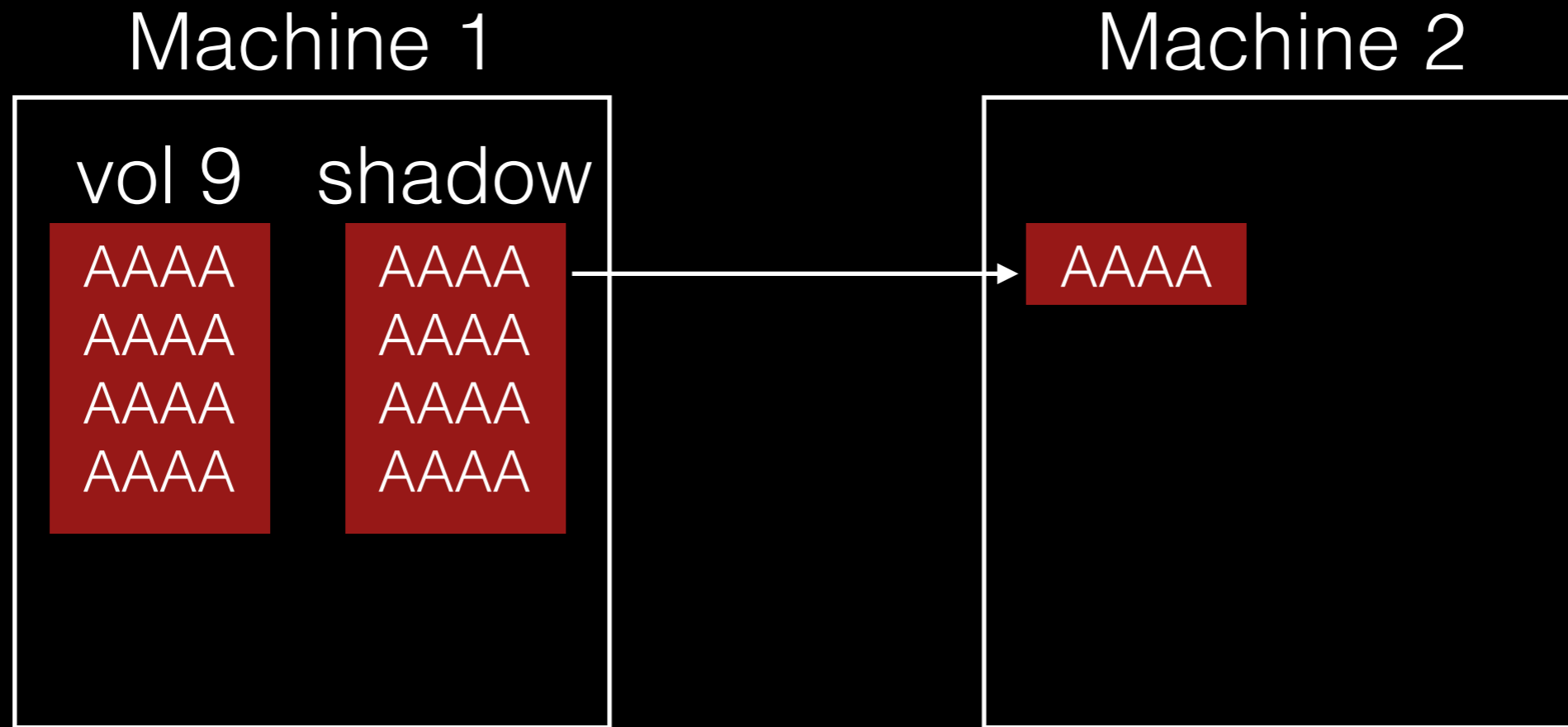
Machine 1



Machine 2



Copy



Copy

Machine 1

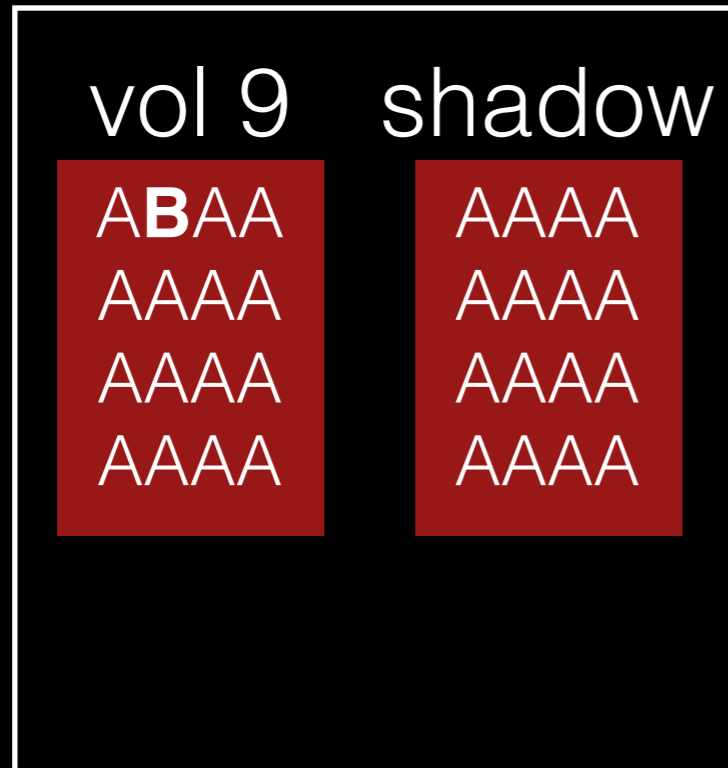


Machine 2



Copy

Machine 1



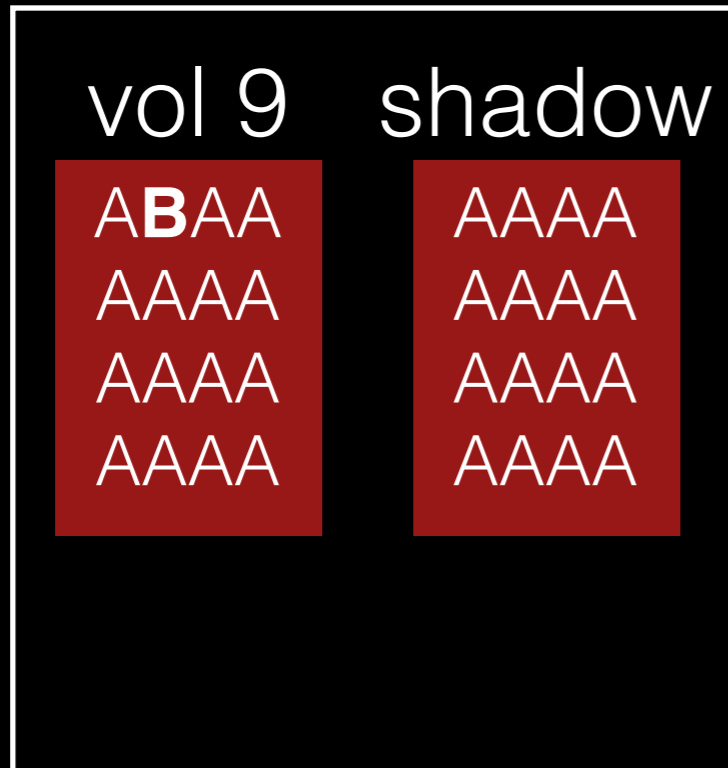
write →

Machine 2

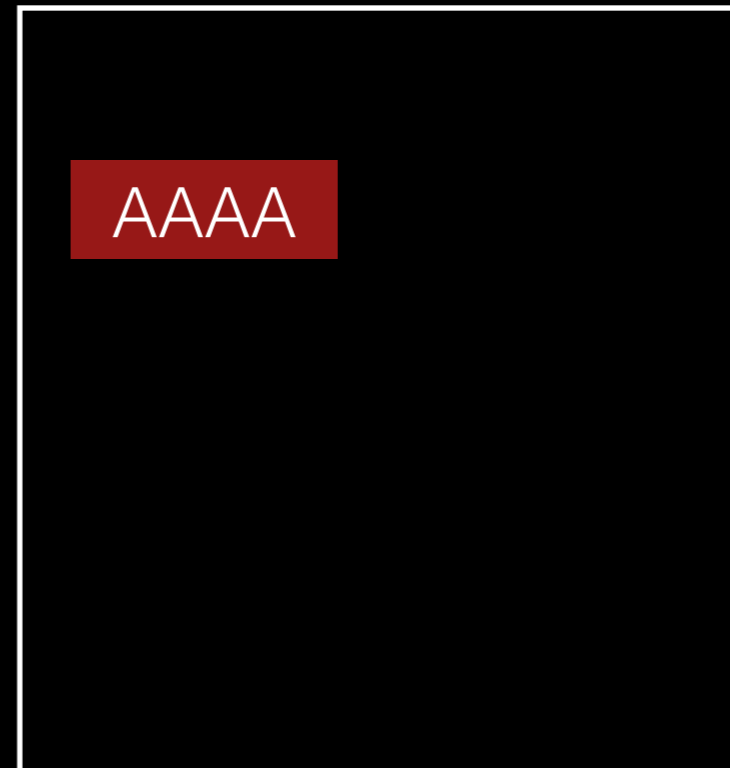


Copy

Machine 1

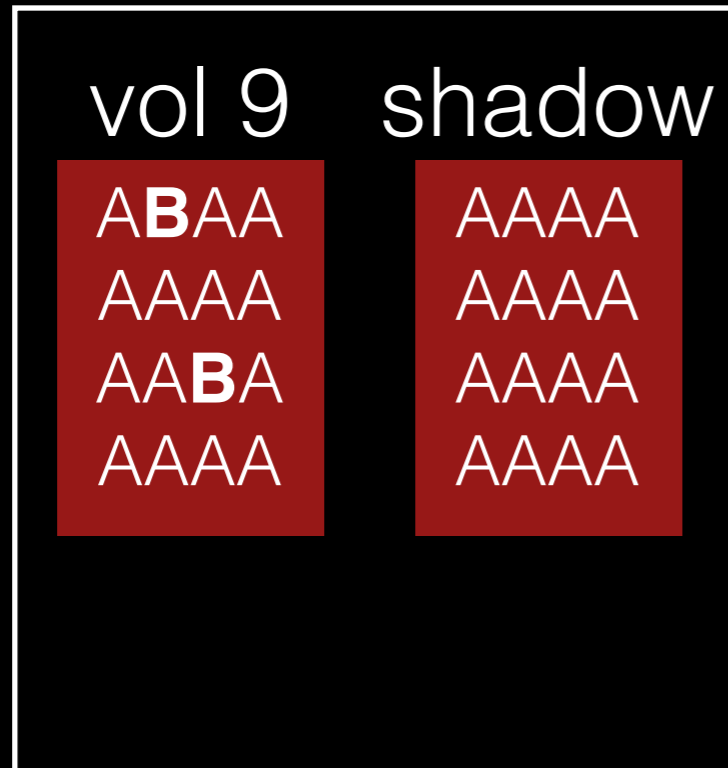


Machine 2



Copy

Machine 1

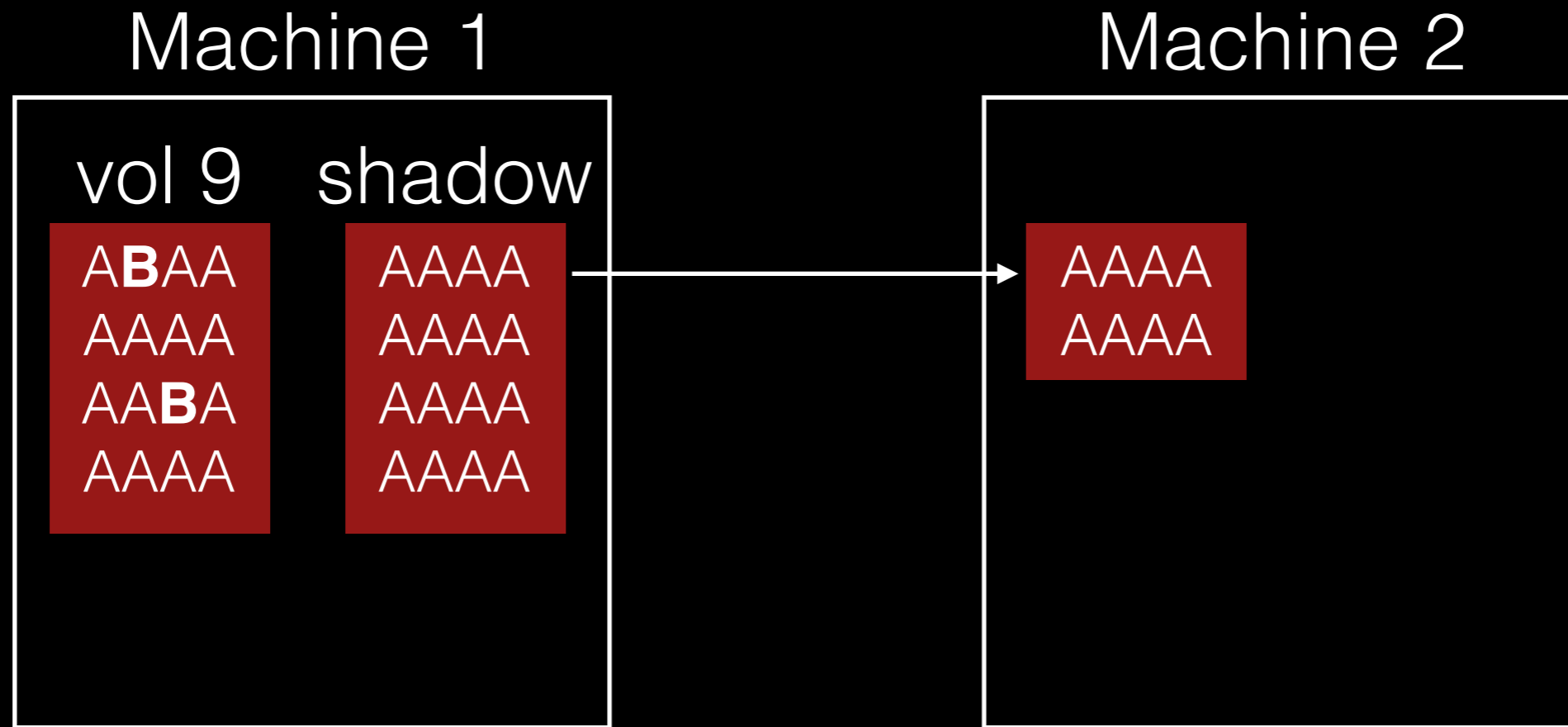


write →

Machine 2



Copy

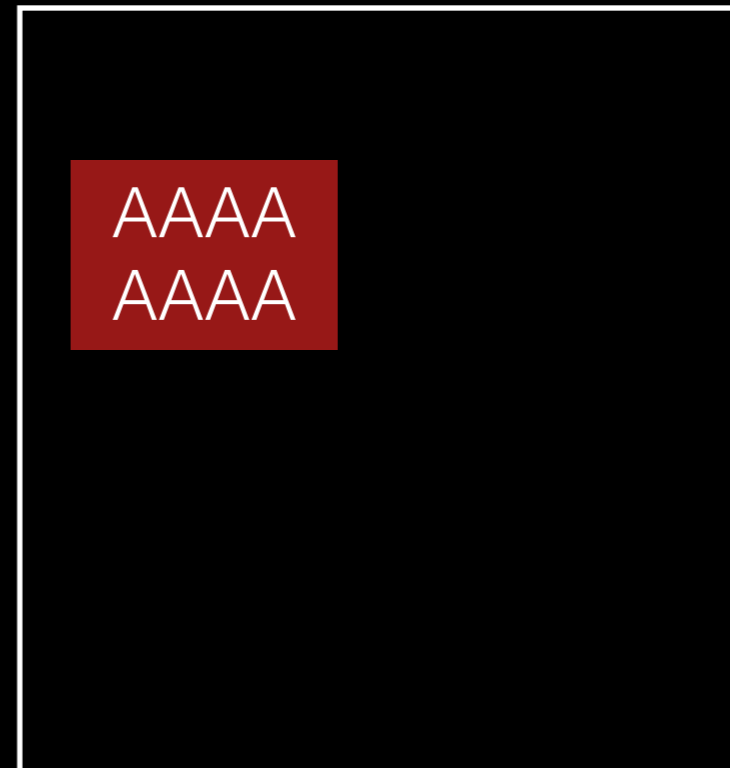


Copy

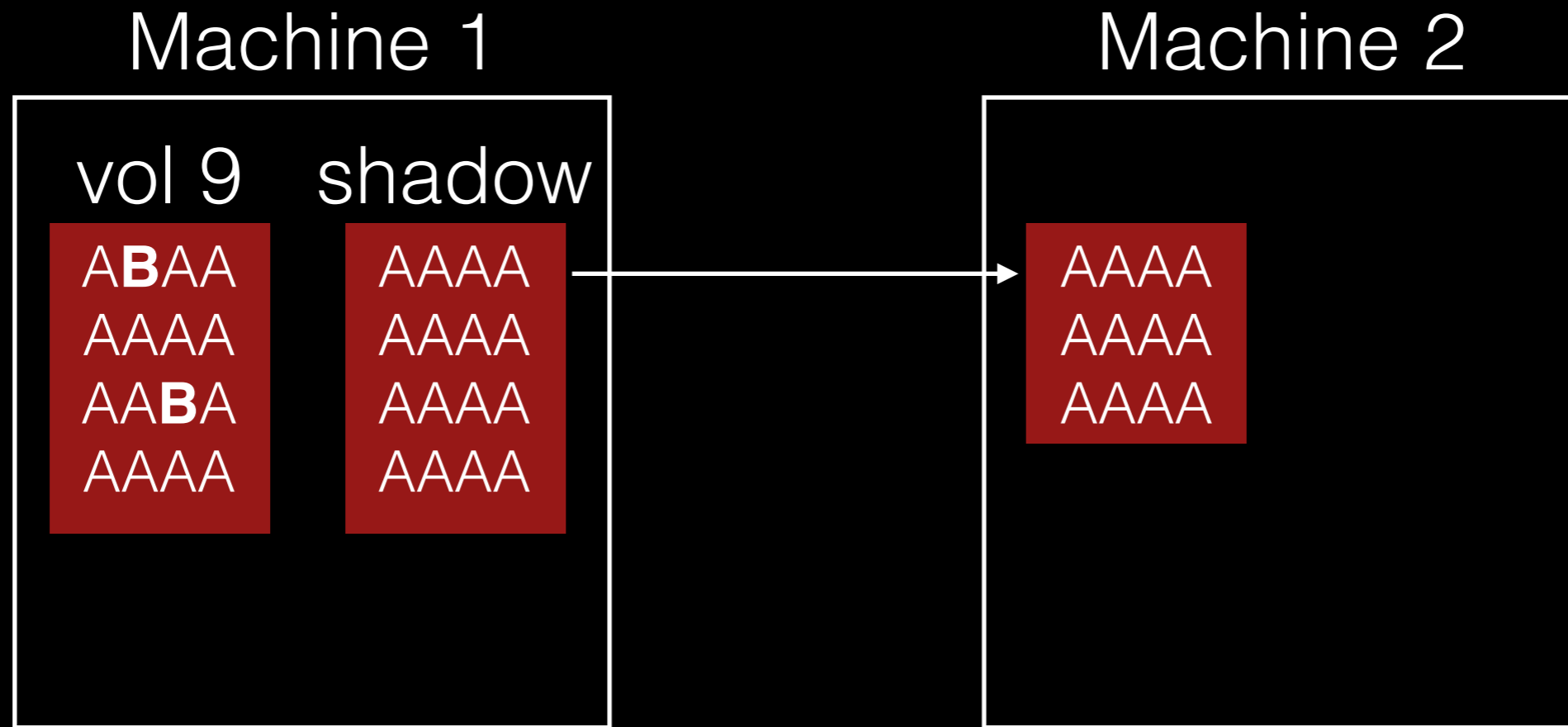
Machine 1



Machine 2

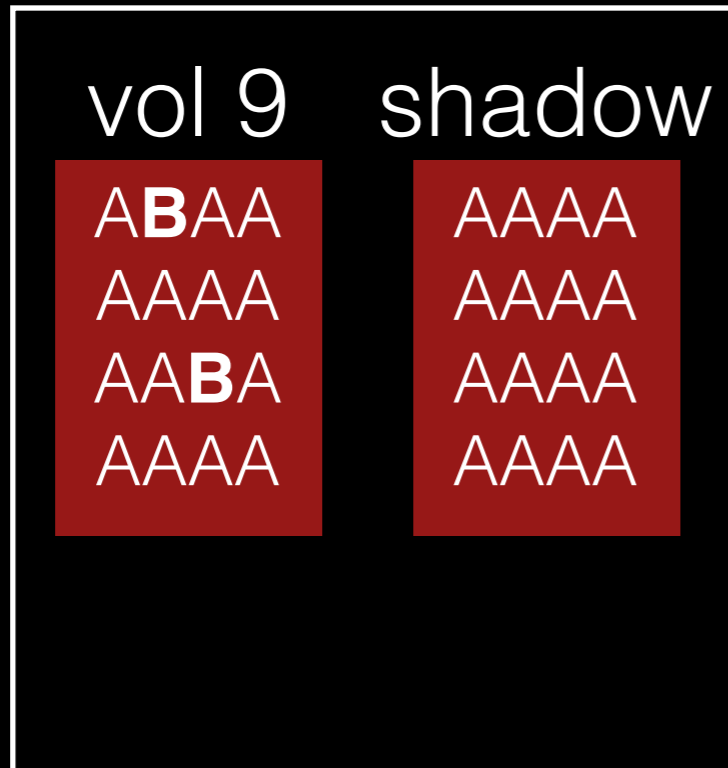


Copy

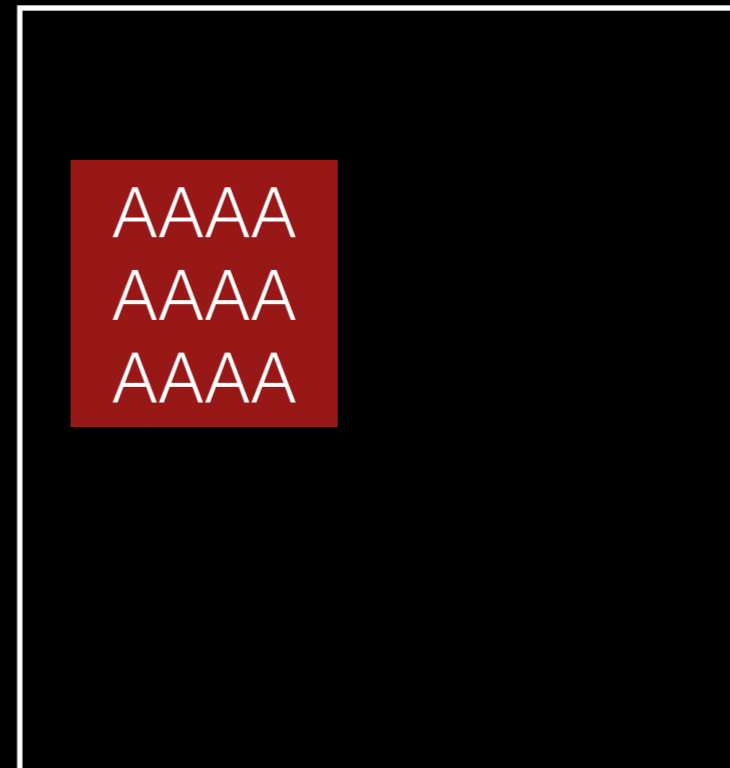


Copy

Machine 1



Machine 2



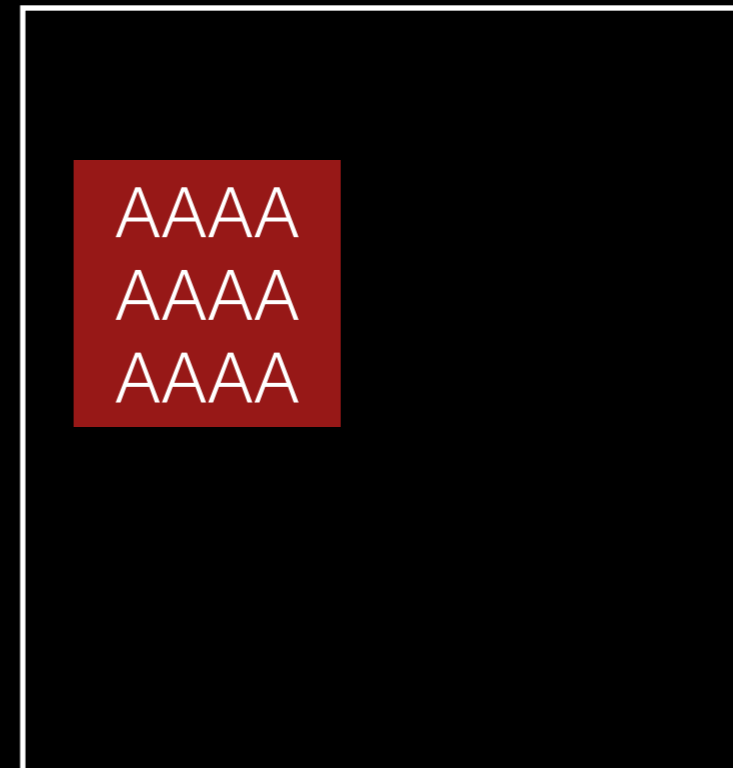
Copy

Machine 1



write →

Machine 2

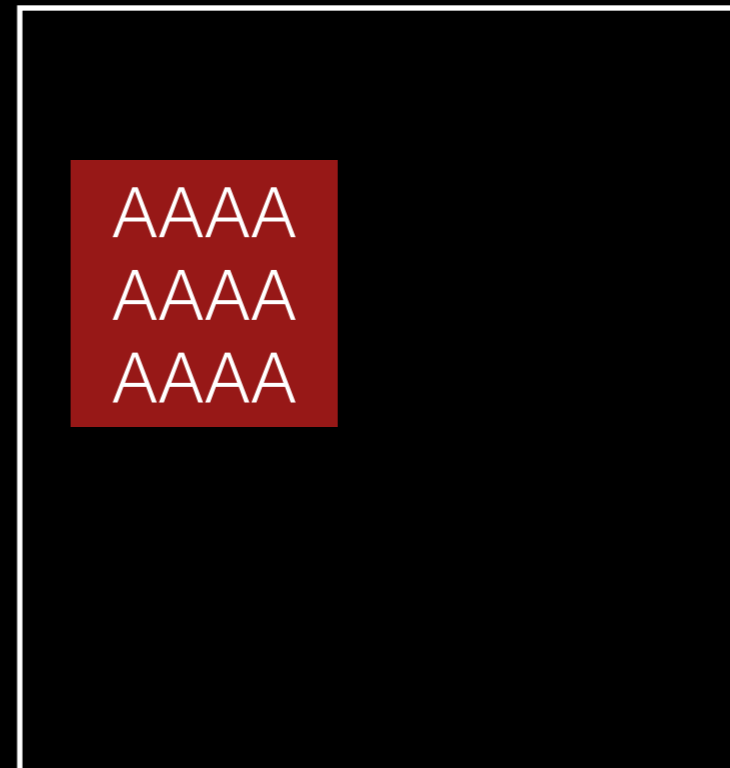


Copy

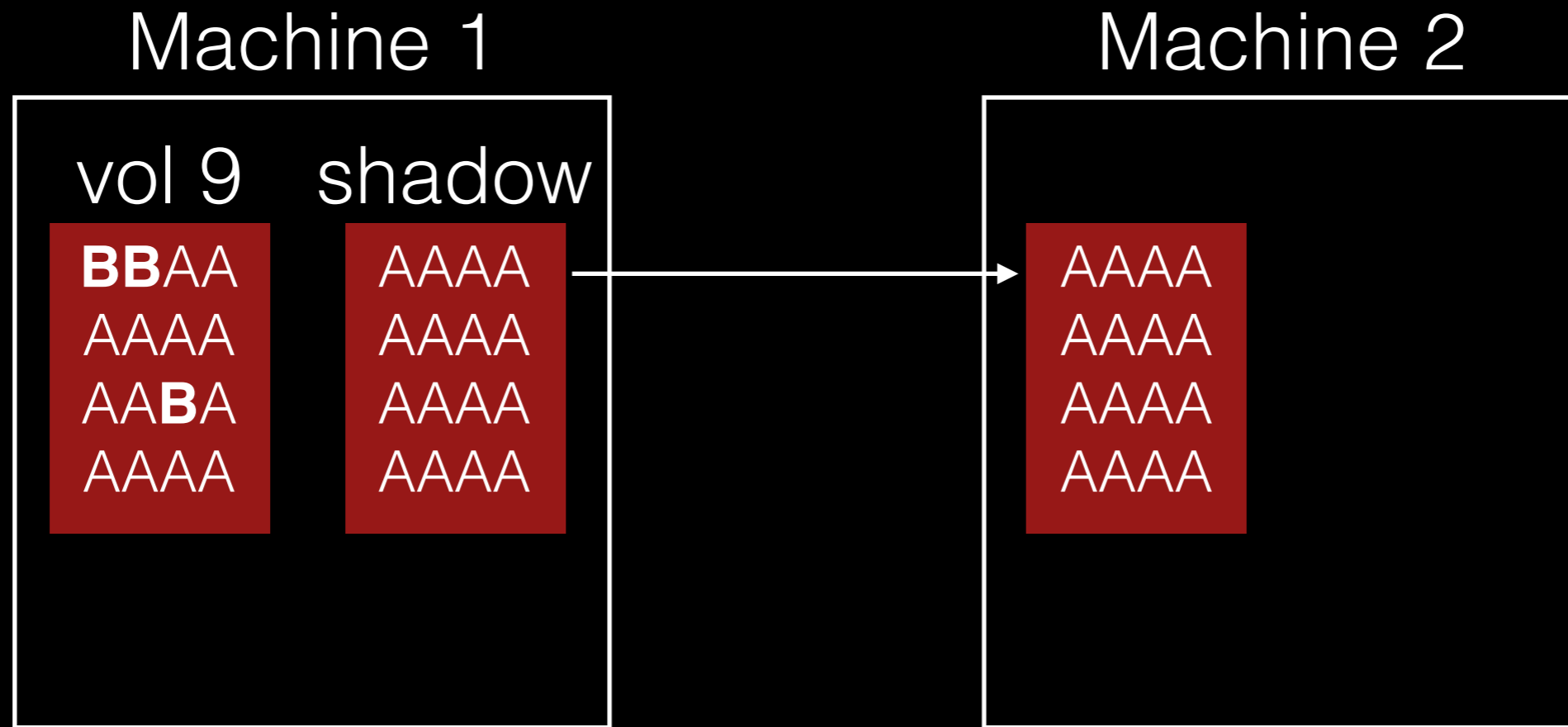
Machine 1



Machine 2



Copy

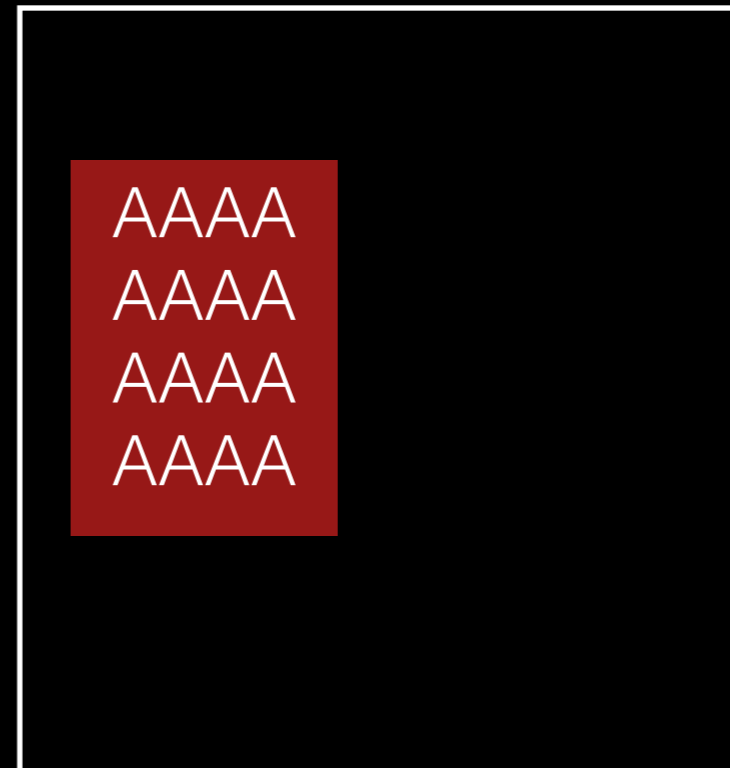


Copy

Machine 1



Machine 2

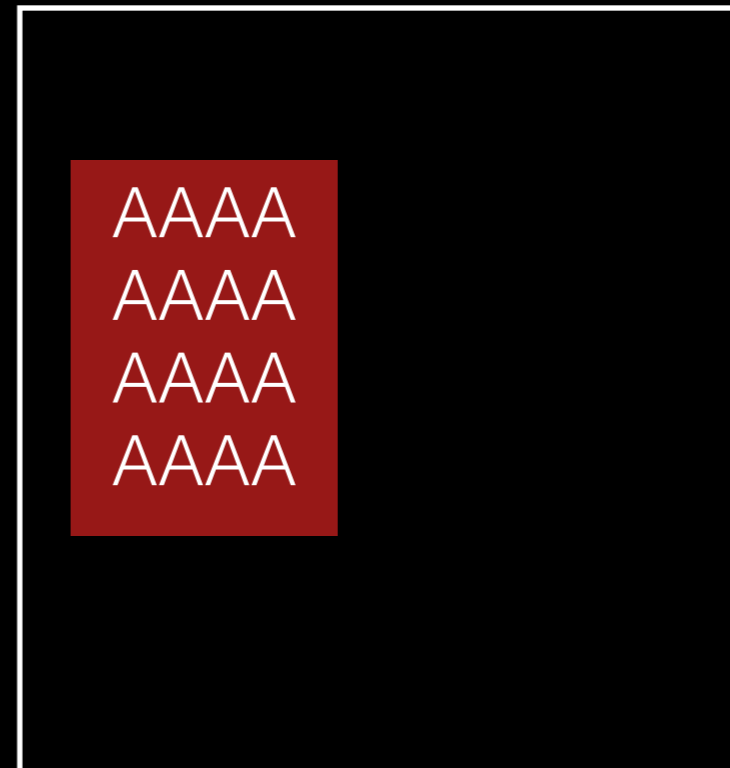


Copy

Machine 1



Machine 2

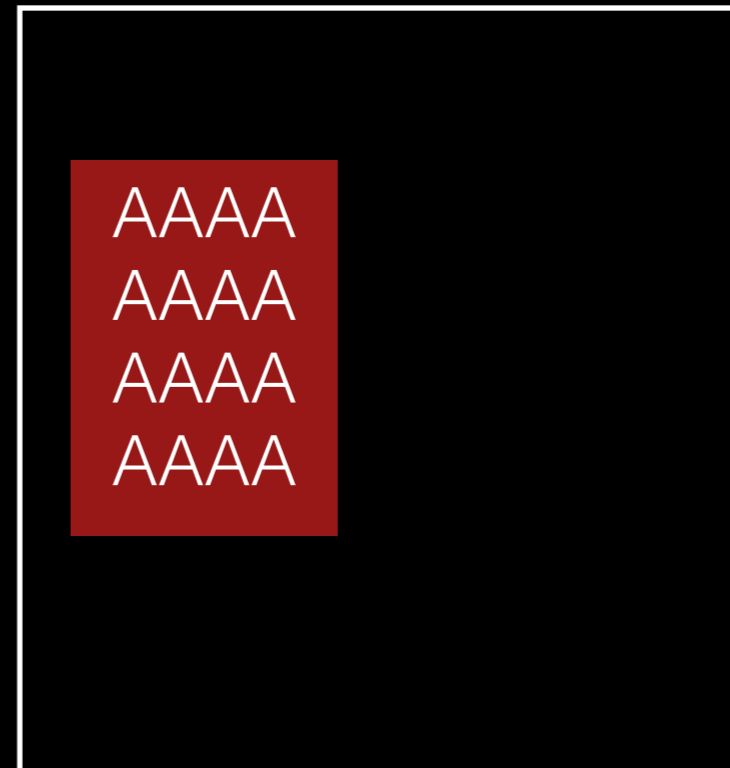


Copy

Machine 1



Machine 2



Copy

Machine 1

vol 9

BBAA
AAAA
AABA
AAAA

(freeze)

write →
(blocked)

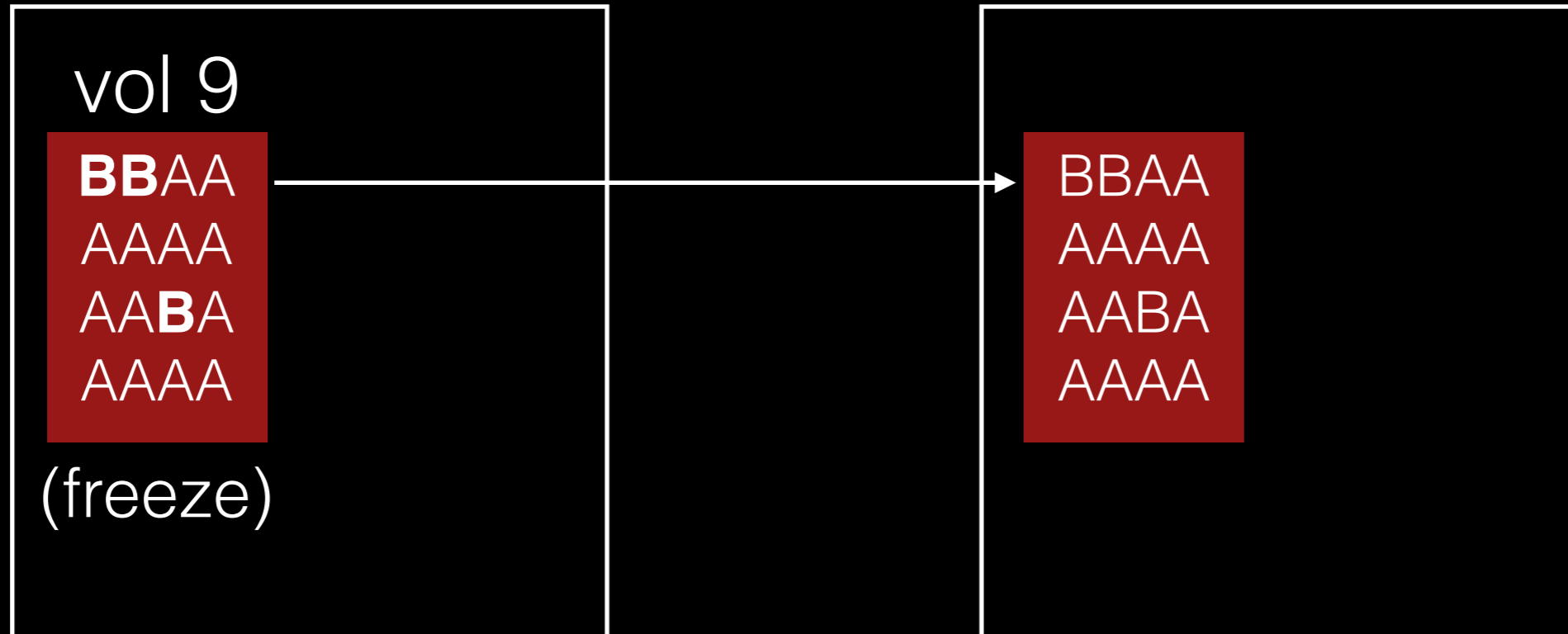
Machine 2

AAAA
AAAA
AAAA
AAAA

Copy

Machine 1

Machine 2



Copy

Machine 1

vol 9

BBAA
AAAA
A**ABA**
AAAA

(freeze)

write →
(blocked)

Machine 2

BBAA
AAAA
AABA
AAAA

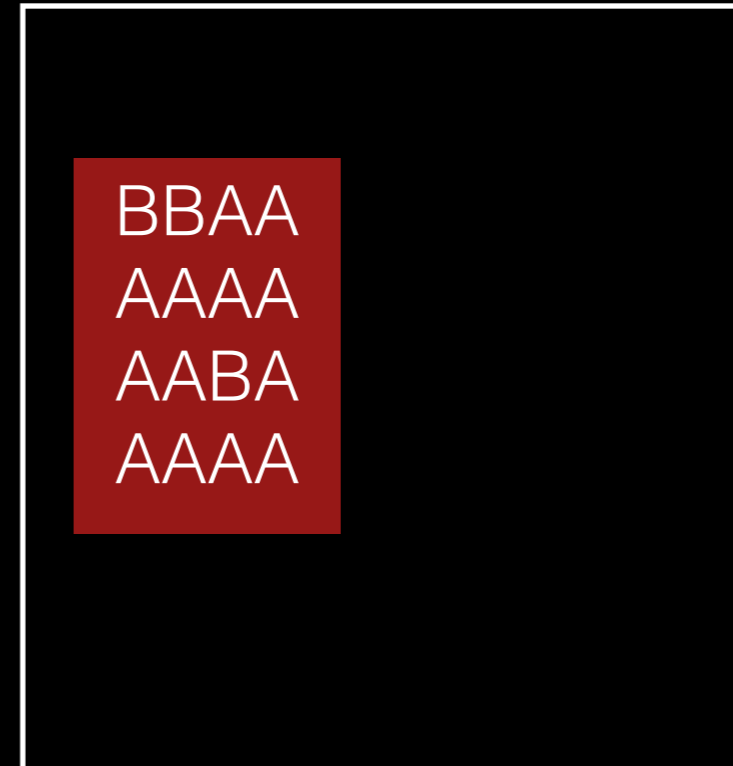
Copy

Machine 1



write →
(blocked)

Machine 2



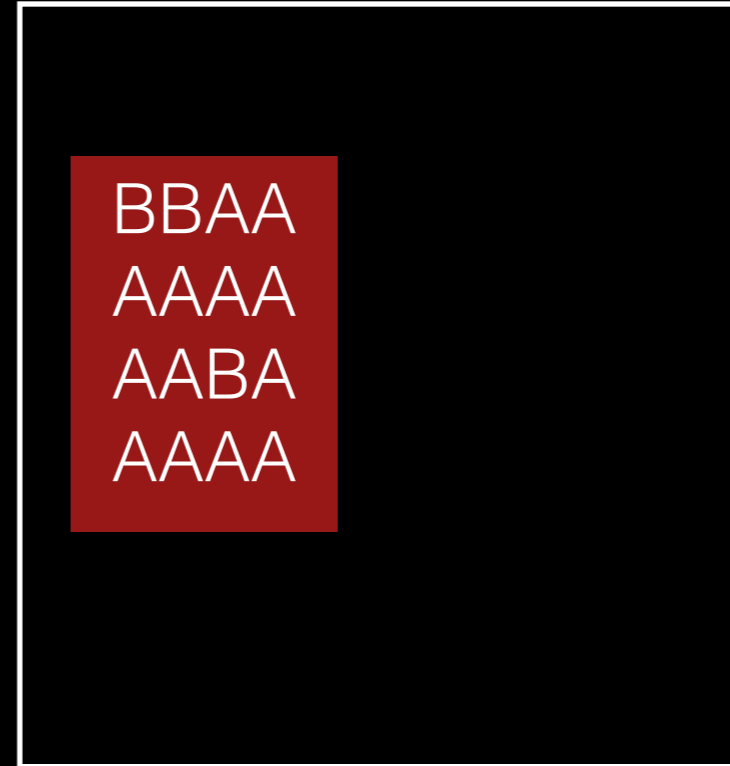
Copy

Machine 1

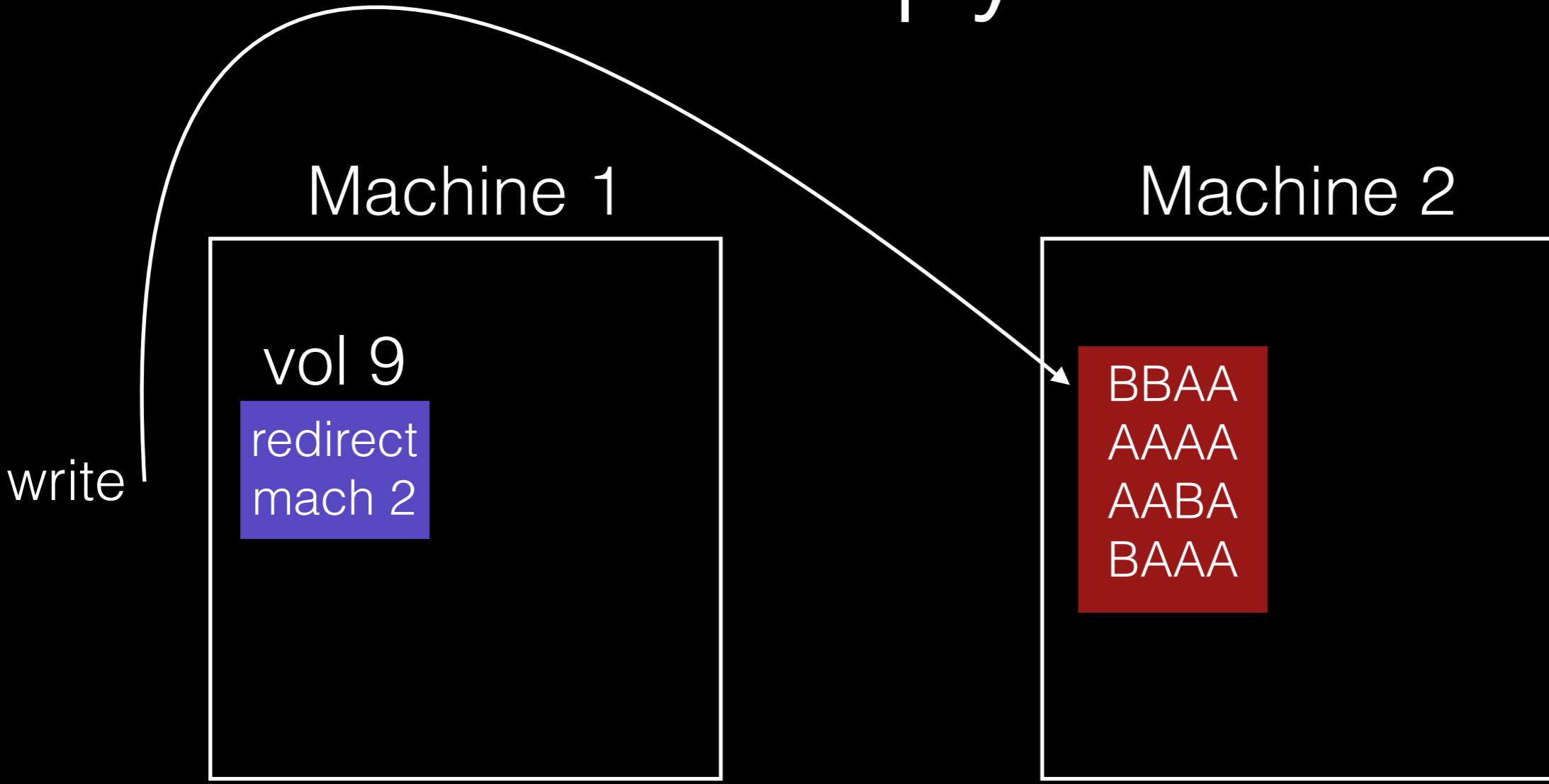


write →

Machine 2



Copy



Copy

Machine 1

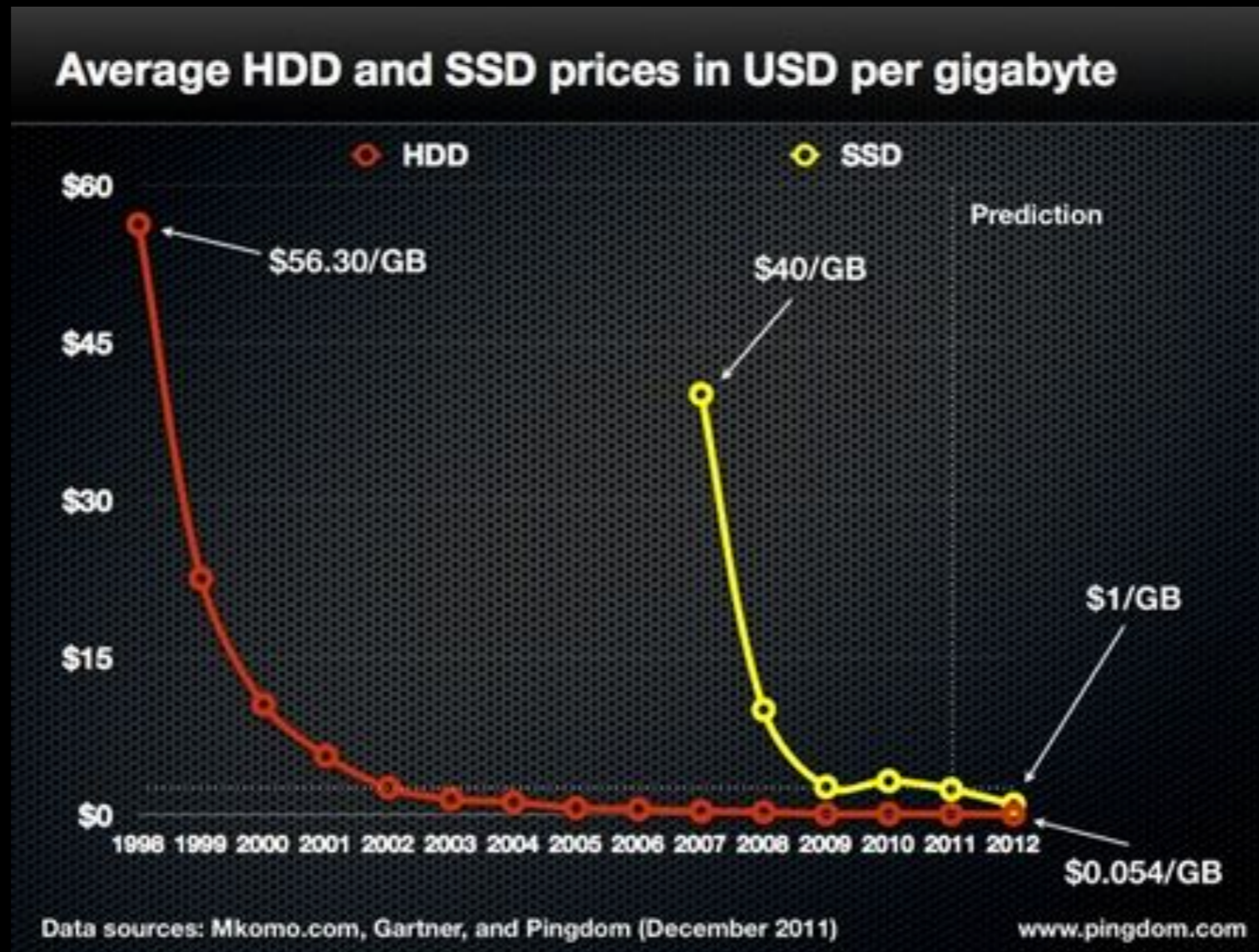
vol 9
redirect
mach 2

Machine 2

BBAA
AAAA
AABA
BAAA

Flash

Cost: HDD vs. SSD



Source: <http://www.tomshardware.com/news/ssd-hdd-solid-state-drive-hard-disk-drive-prices,14336.html>

Disk Overview

I/O requires: seek, rotate, transfer

Inherently:

- not parallel (only one head)
- slow (mechanical)
- poor random I/O (locality around disk head)

Random requests take 10ms+

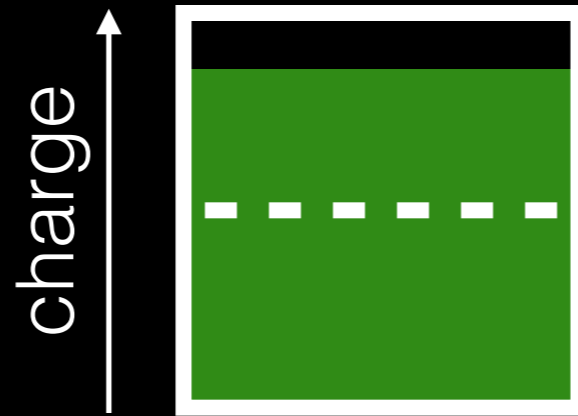
Flash

Hold charge in cells. No moving parts!

Inherently parallel.

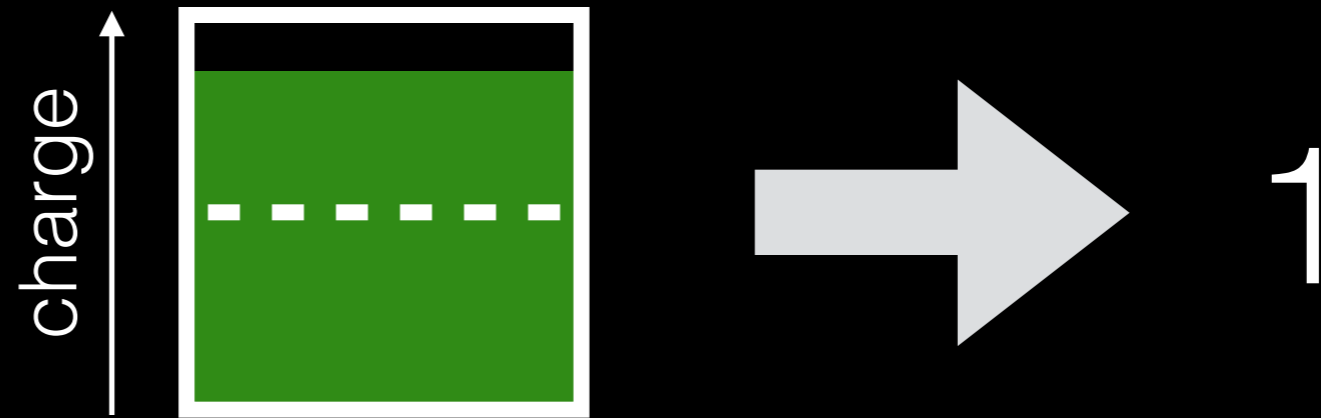
No seeks!

SLC: Single-Level Cell



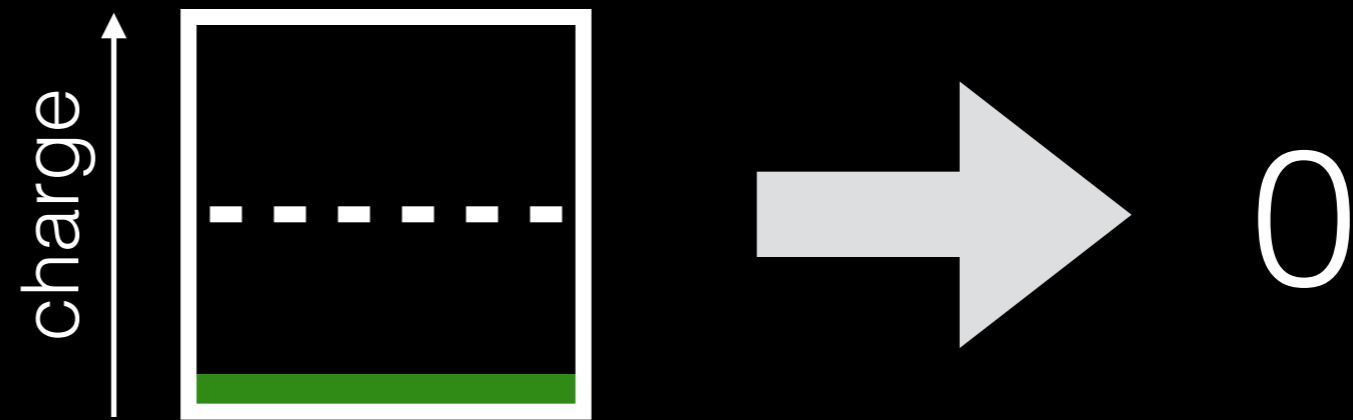
NAND Cell

SLC: Single-Level Cell



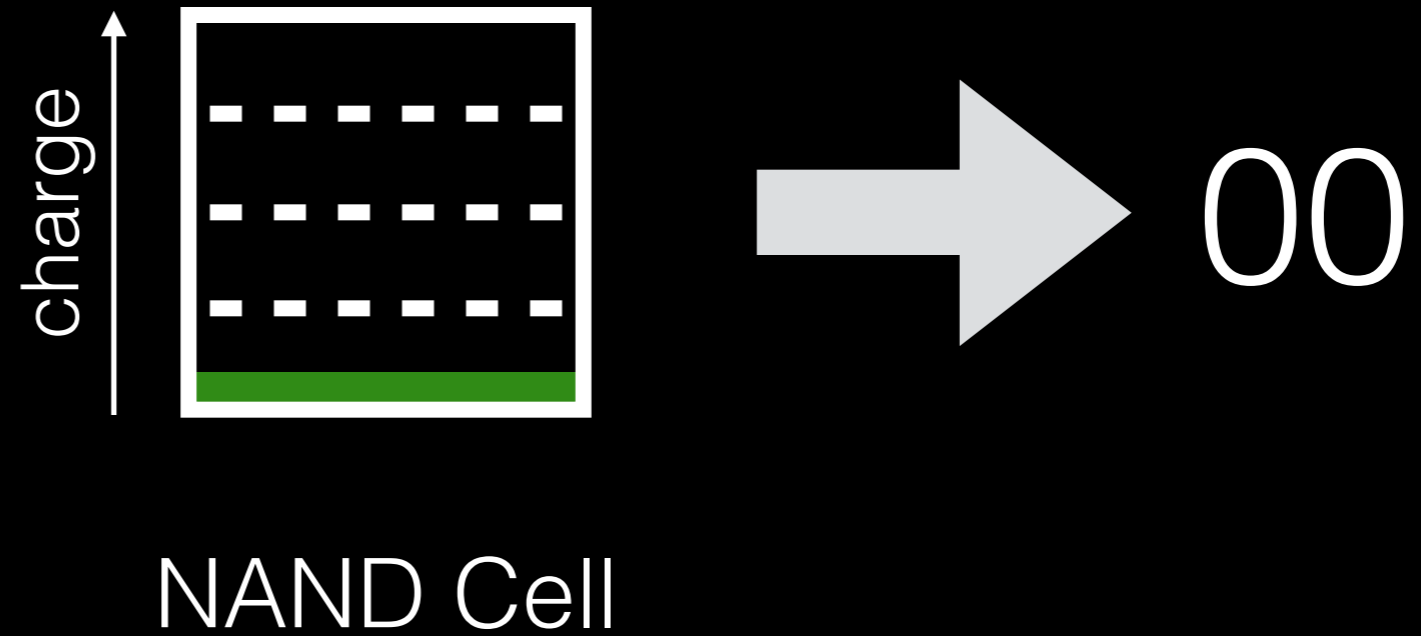
NAND Cell

SLC: Single-Level Cell

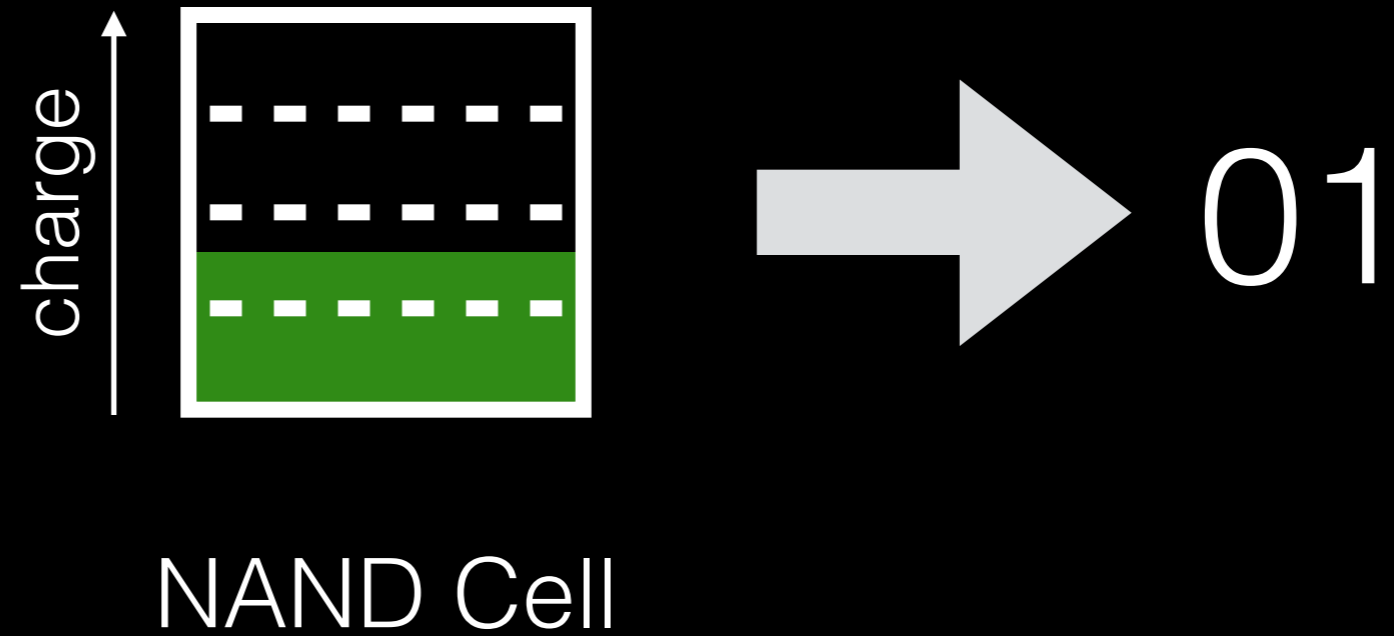


NAND Cell

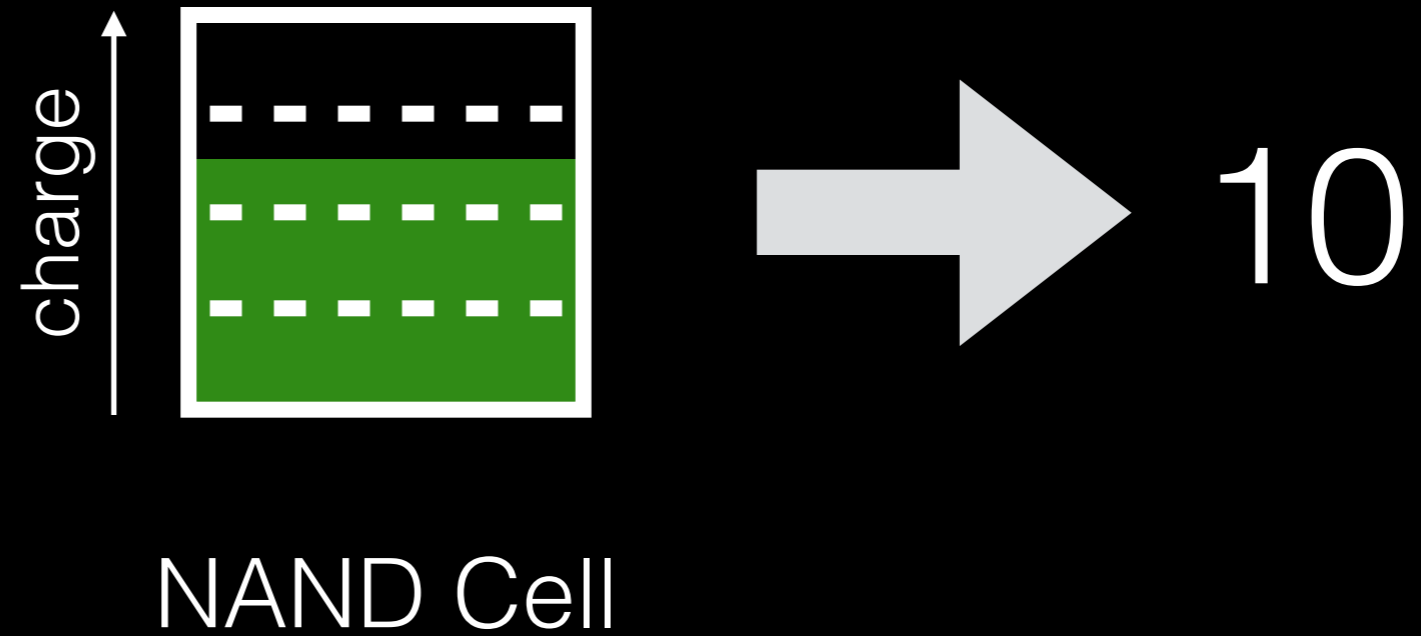
MLC: Multi-Level Cell



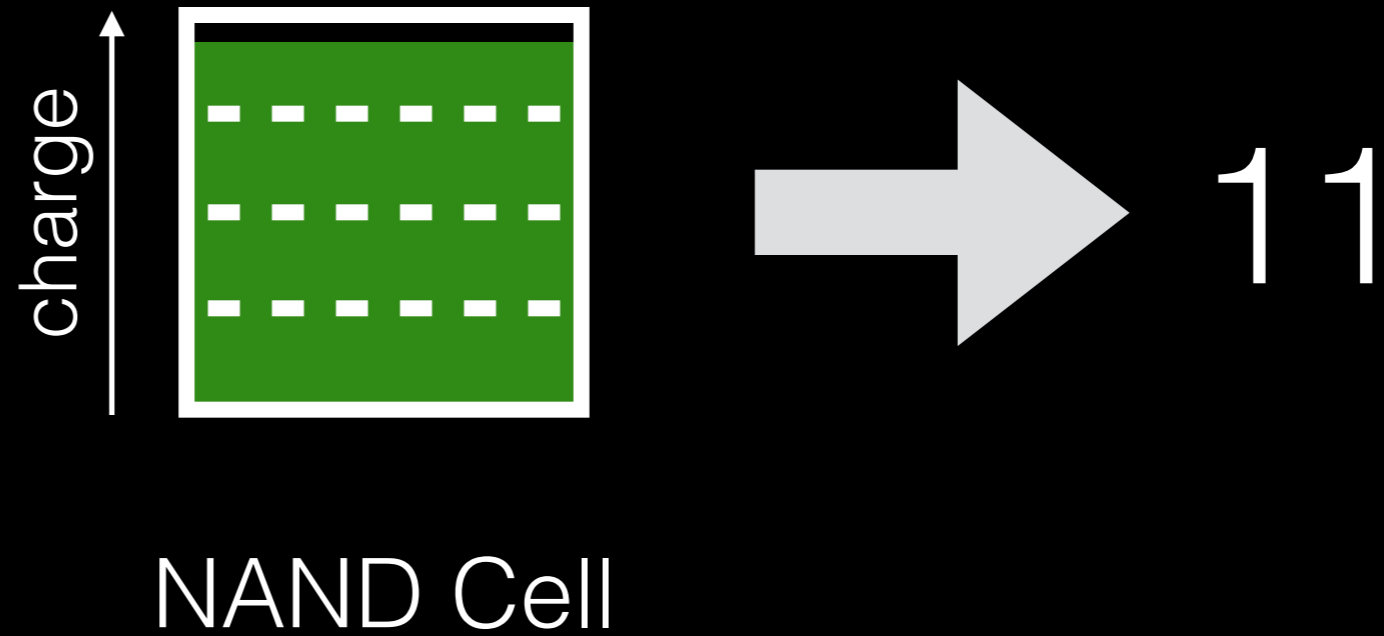
MLC: Multi-Level Cell



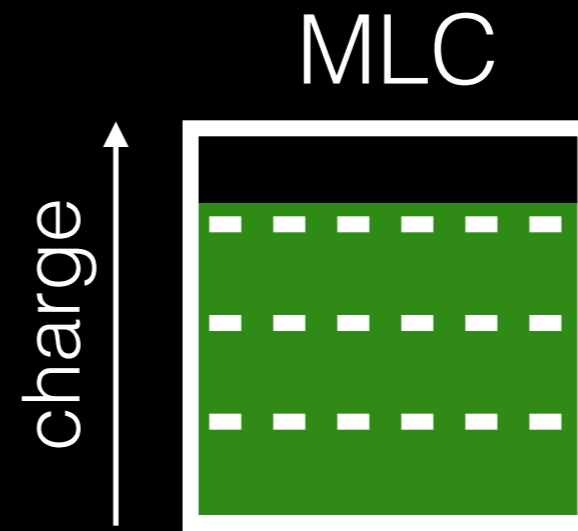
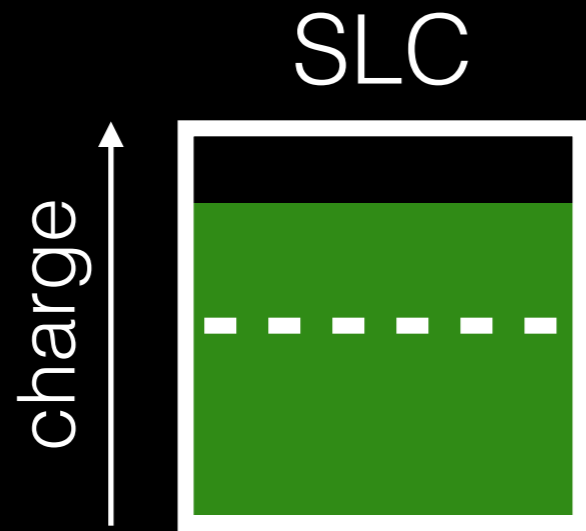
MLC: Multi-Level Cell



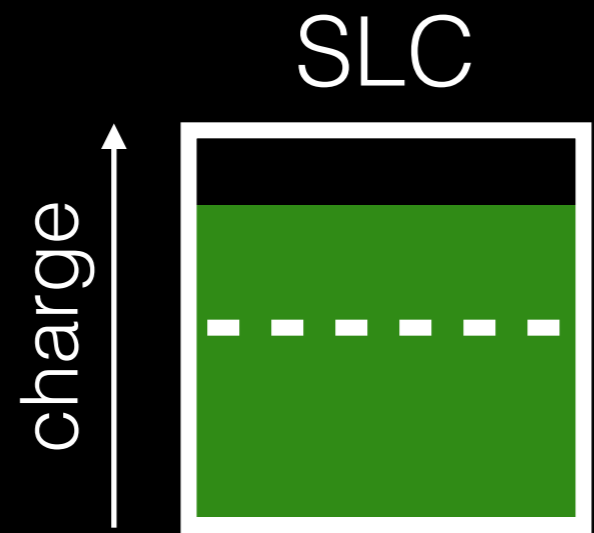
MLC: Multi-Level Cell



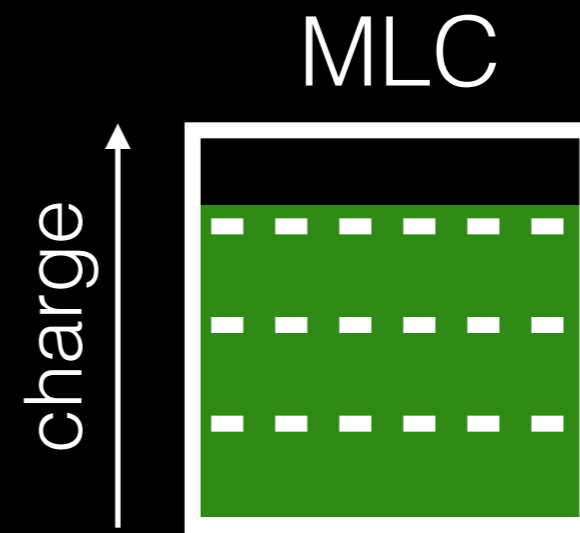
Single- vs. Multi-Level Cell



Single- vs. Multi-Level Cell



expensive
robust



cheap
sensitive

Wearout

Problem: flash cells wear out after being overwritten too many times.

MLC: ~10K times

SLC: ~100K times

Usage strategy:

Wearout

Problem: flash cells wear out after being overwritten too many times.

MLC: ~10K times

SLC: ~100K times

Usage strategy: **wear leveling**.

- prevents some cells from wearing out while others still fresh.

Banks

Flash devices are divided into banks (aka, planes).

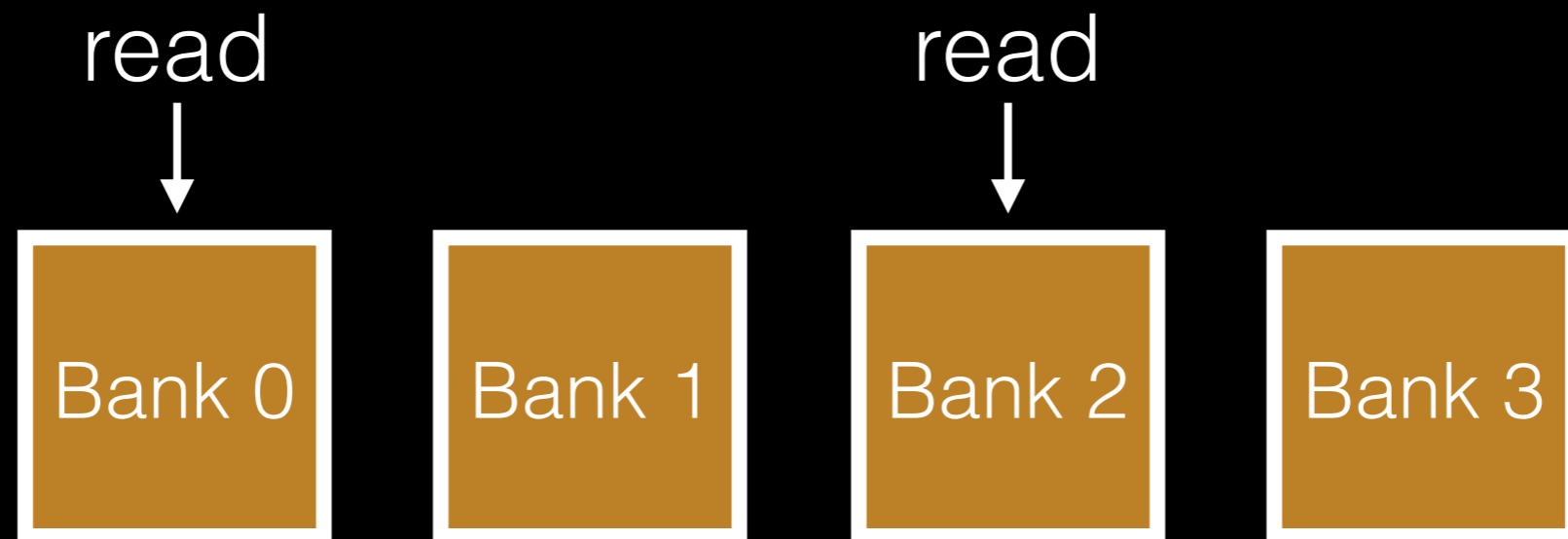
Banks can be accessed in parallel.



Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.



Banks

Flash devices are divided into banks (aka, planes).

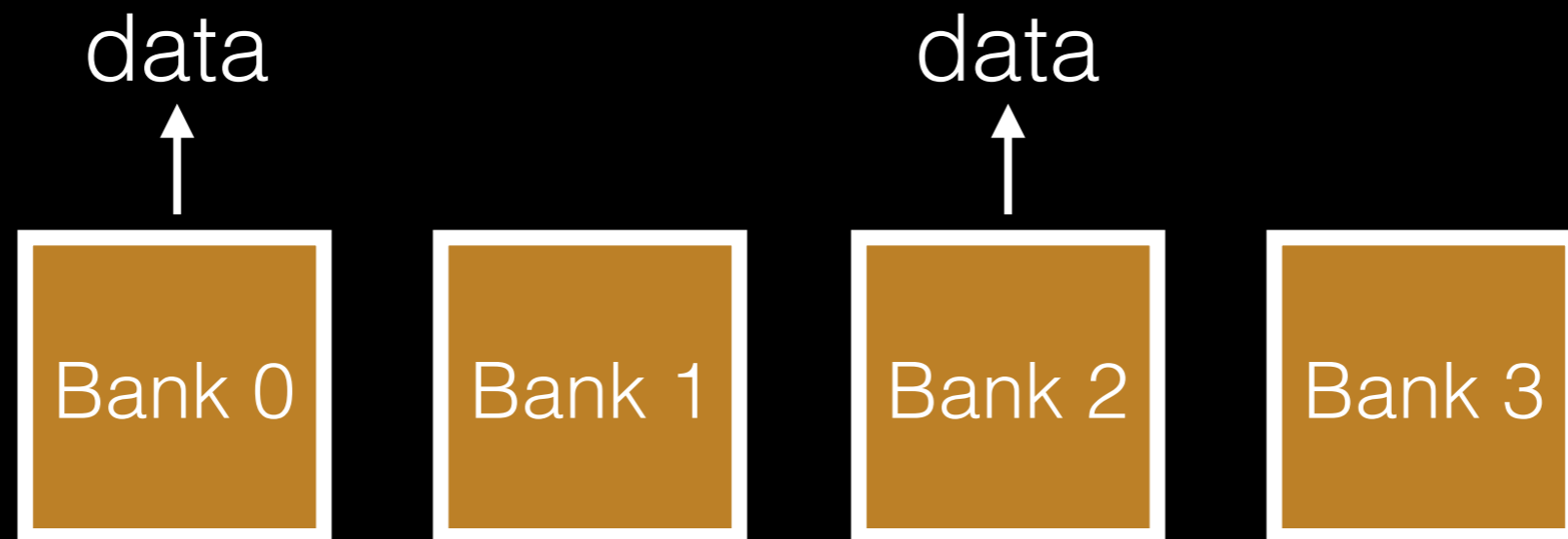
Banks can be accessed in parallel.



Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.



Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.



Flash Writes

Writing 0's:

- fast, fine-grained

Writing 1's:

- slow, course-grained

Flash Writes

Writing 0's:

- fast, fine-grained
- called "program"

Writing 1's:

- slow, course-grained
- called "erase"

Flash Writes

Writing 0's:

- fast, fine-grained [pages]
- called "program"

Writing 1's:

- slow, coarse-grained [blocks]
- called "erase"

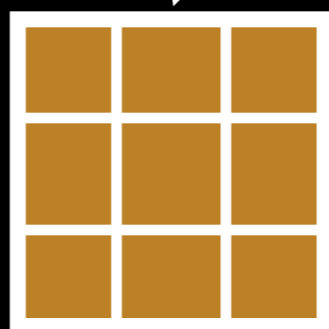
Bank 0

Bank 1

Bank 2

Bank 3

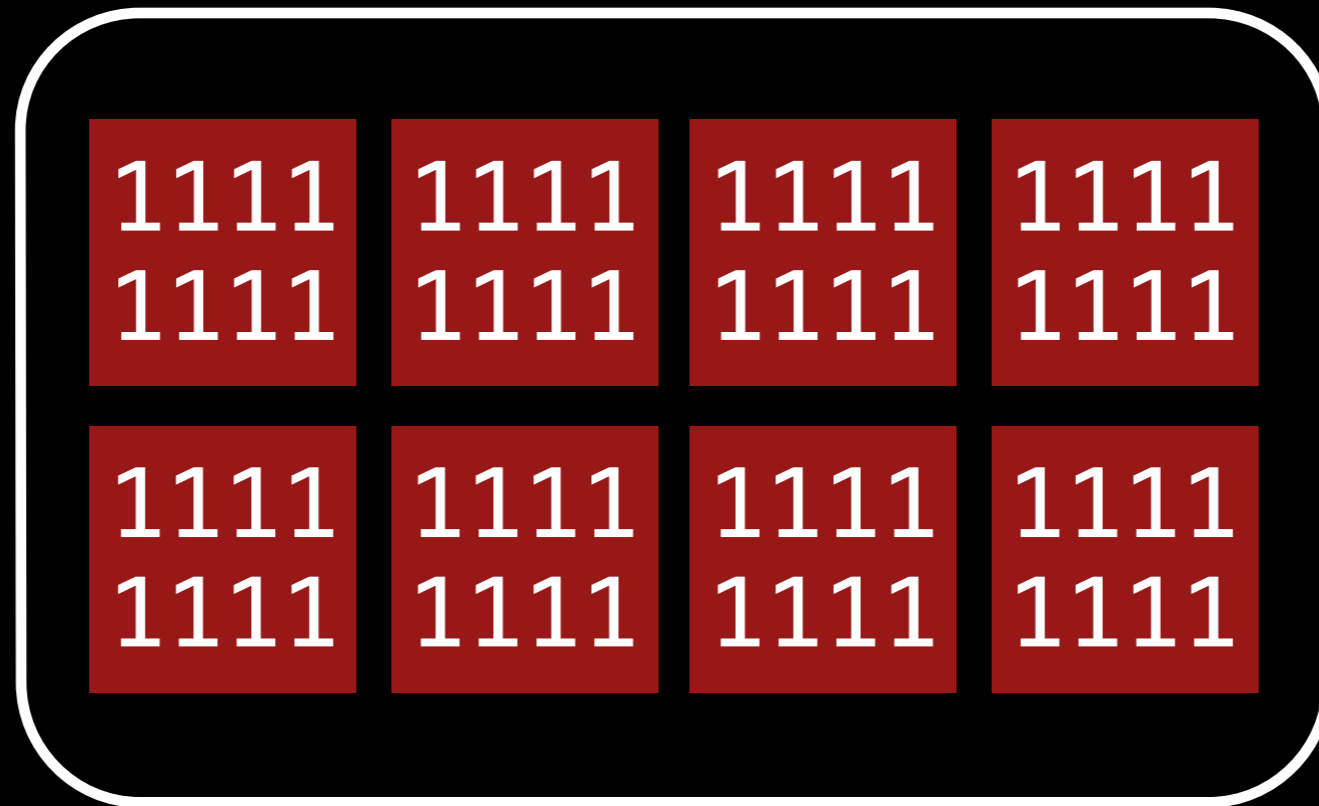
each bank contains
many "blocks"



Block

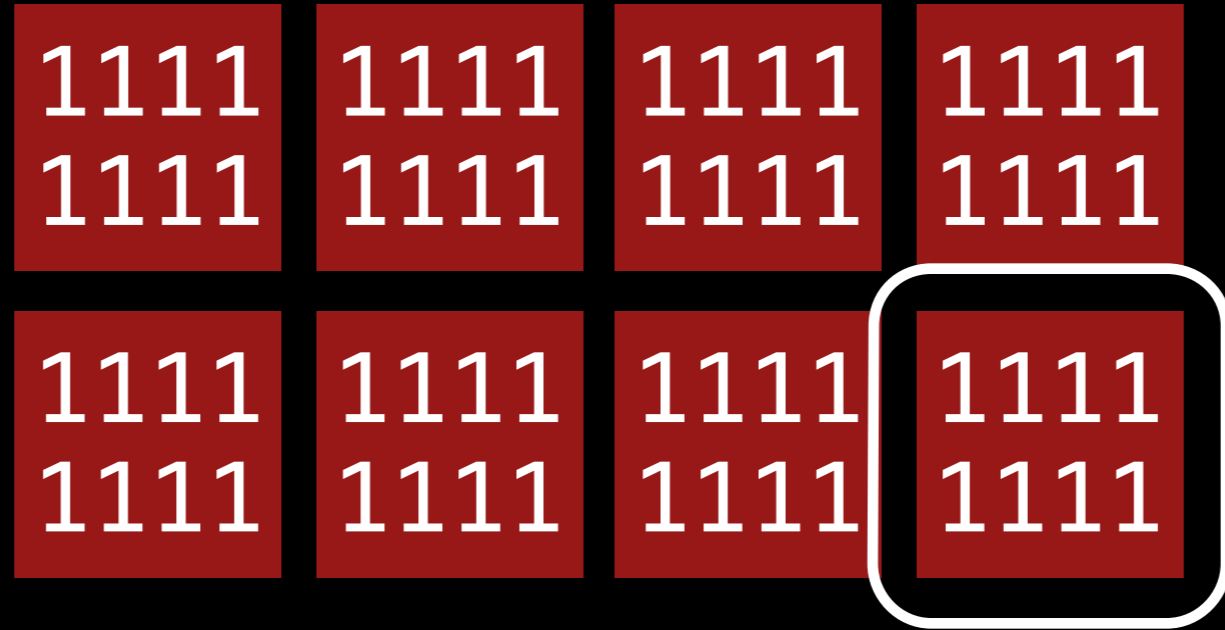


Block



one block

Block



one page

Block



Block

program



Block

1111 1111	1111 1111	1111 1111	1001 1111
1111 1111	1111 1111	1111 1111	1111 1111

Block

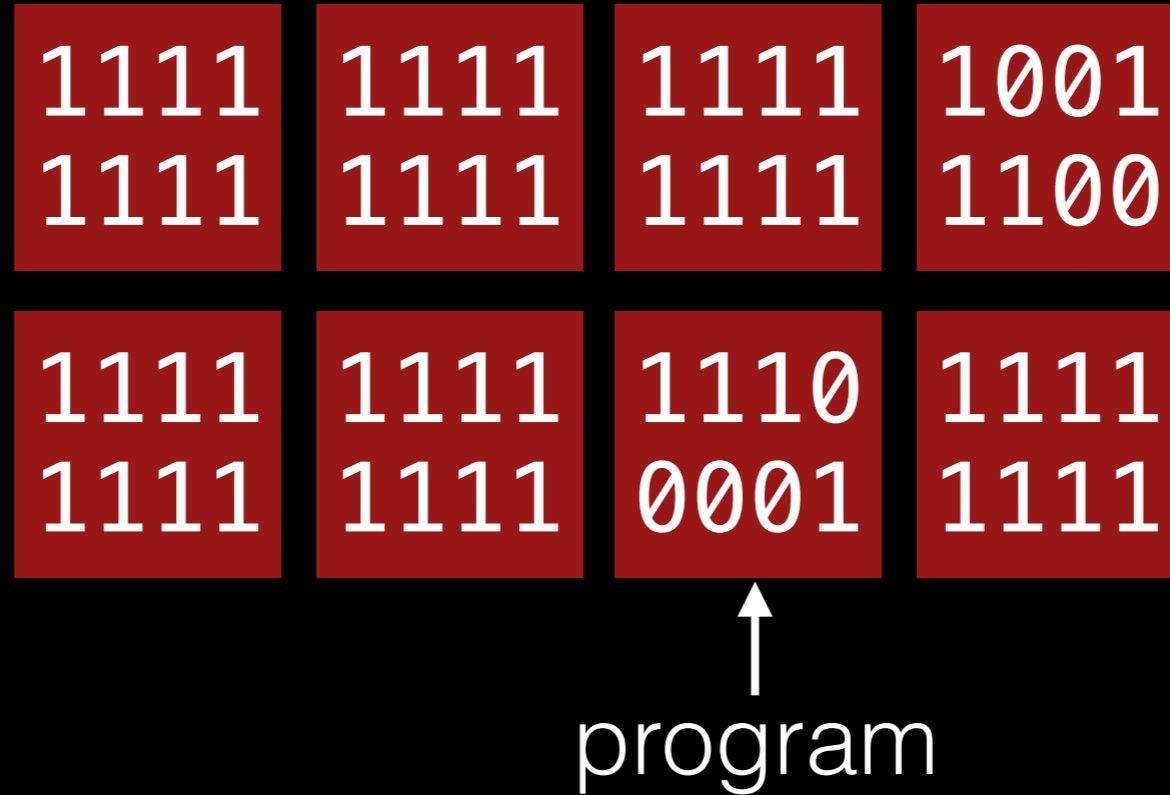
program



Block

1111 1111	1111 1111	1111 1111	1001 1100
1111 1111	1111 1111	1111 1111	1111 1111

Block



Block

1111 1111	1111 1111	1111 1111	1001 1100
1111 1111	1111 1111	1110 0001	1111 1111

Block

1111 1111	1111 1111	1111 1111	1001 1100
1111 1111	1111 1111	1110 0001	1111 1111

erase

Block



erase

Block



APIs

disk

flash

read

write

APIs

disk

flash

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

Flash Hierarchy

Plane: 1024 to 4096 blocks

- planes accessed in parallel

Block: 64 to 256 pages

- unit of **erase**

Page: 2 to 8 KB

- unit of **read** and **program**

Disk vs. Flash Performance

Throughput:

- disk: ~130 MB/s (sequential)
- flash: ~200 MB/s

Disk vs. Flash Performance

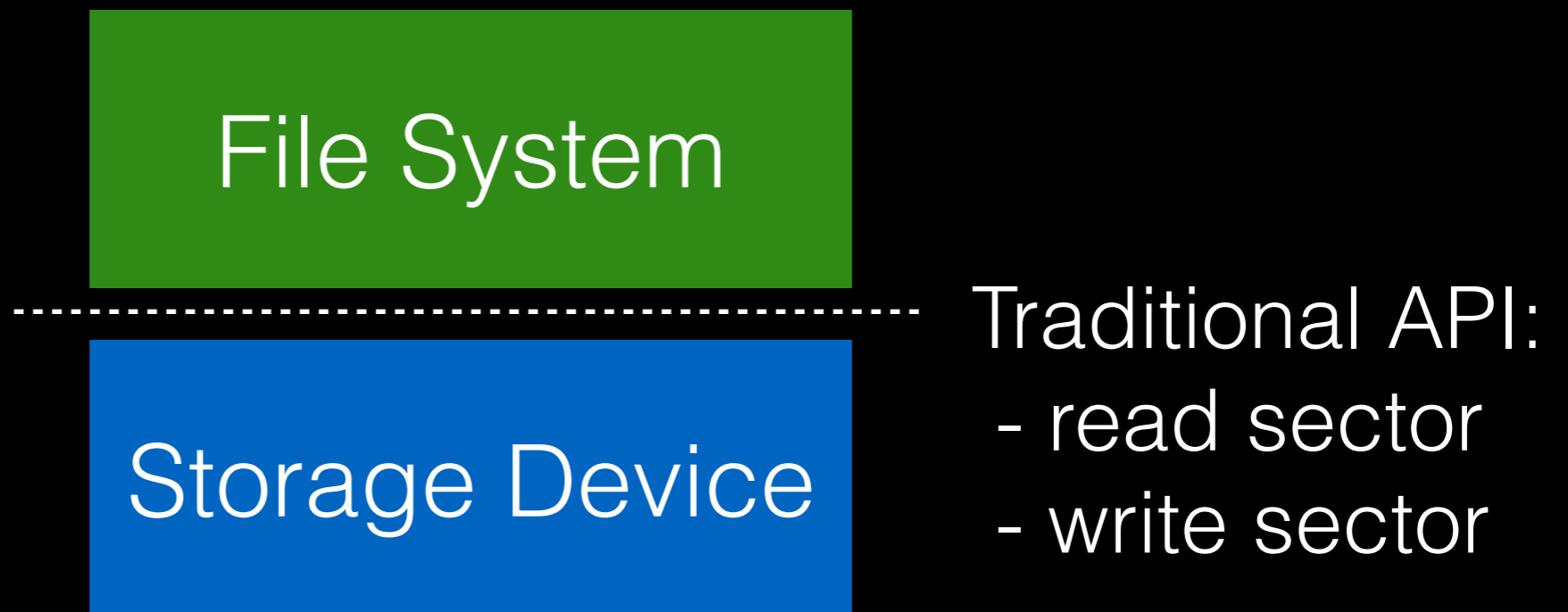
Throughput:

- disk: ~130 MB/s (sequential)
- flash: ~200 MB/s

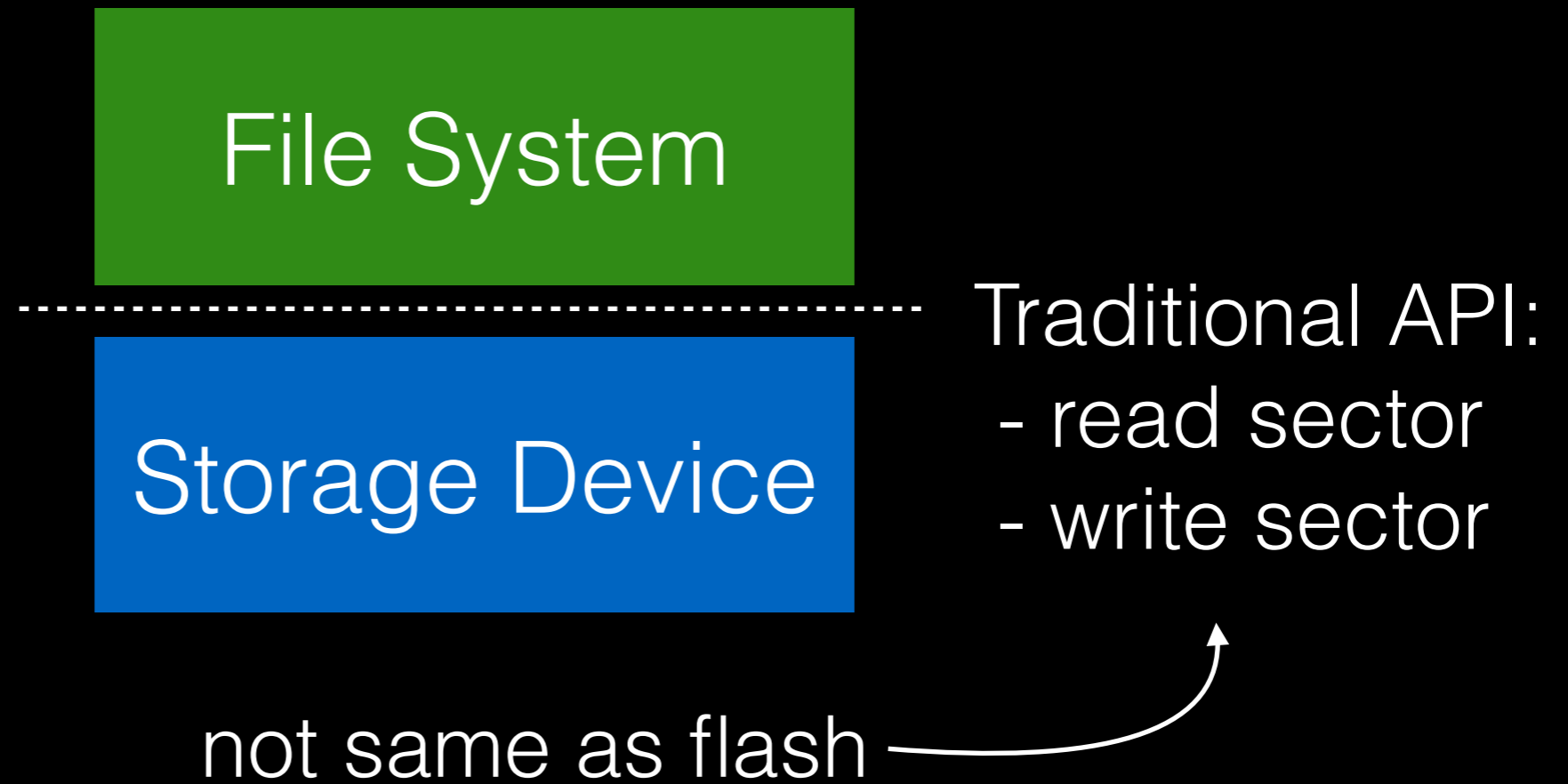
Latency

- disk: ~10 ms (one op)
- flash
 - read: 10-50 us
 - program: 200-500 us
 - erase: 2 ms

Traditional File Systems



Traditional File Systems



Options

1. Build/use new file systems for flash
 - JFFS, YAFFS
 - lot of work!
2. Build traditional API over flash API.
 - use FFS, LFS, whatever we want

Traditional API with Flash

```
read(addr):
```

```
    return flash_read(addr)
```

```
write(addr, data):
```

```
    block_copy = flash_read(block of addr)
```

```
    modify block_copy with data
```

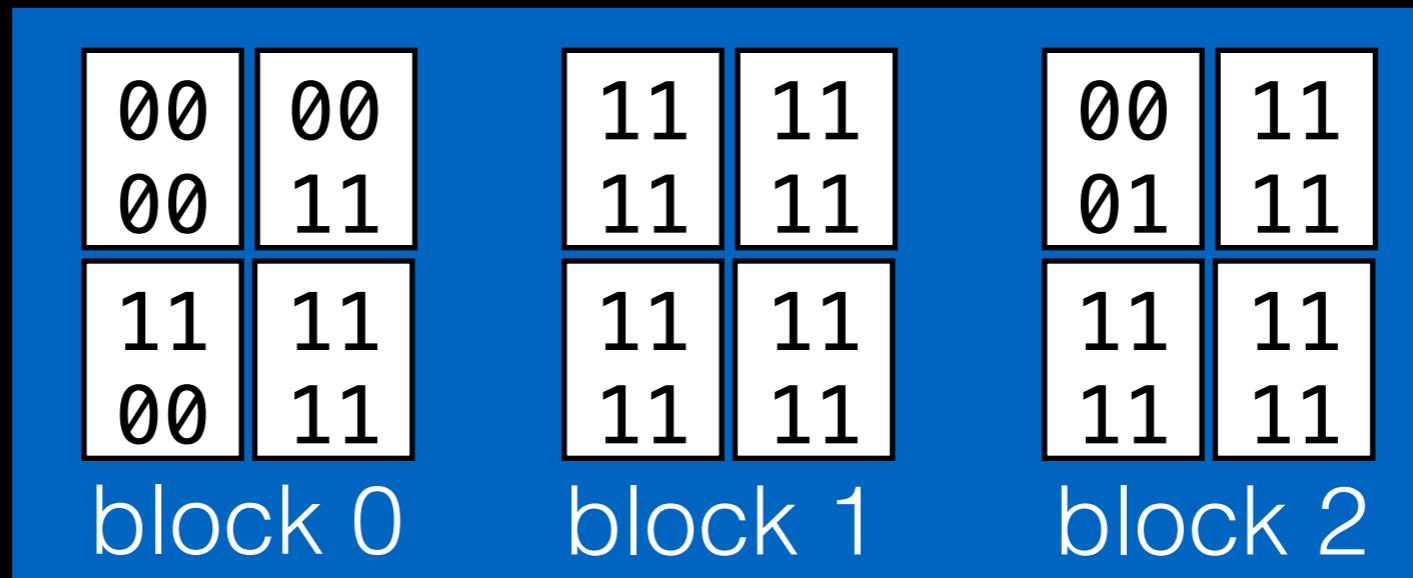
```
    flash_erase(block of addr)
```

```
    flash_program(block of addr, block_copy)
```

Memory:



Flash:



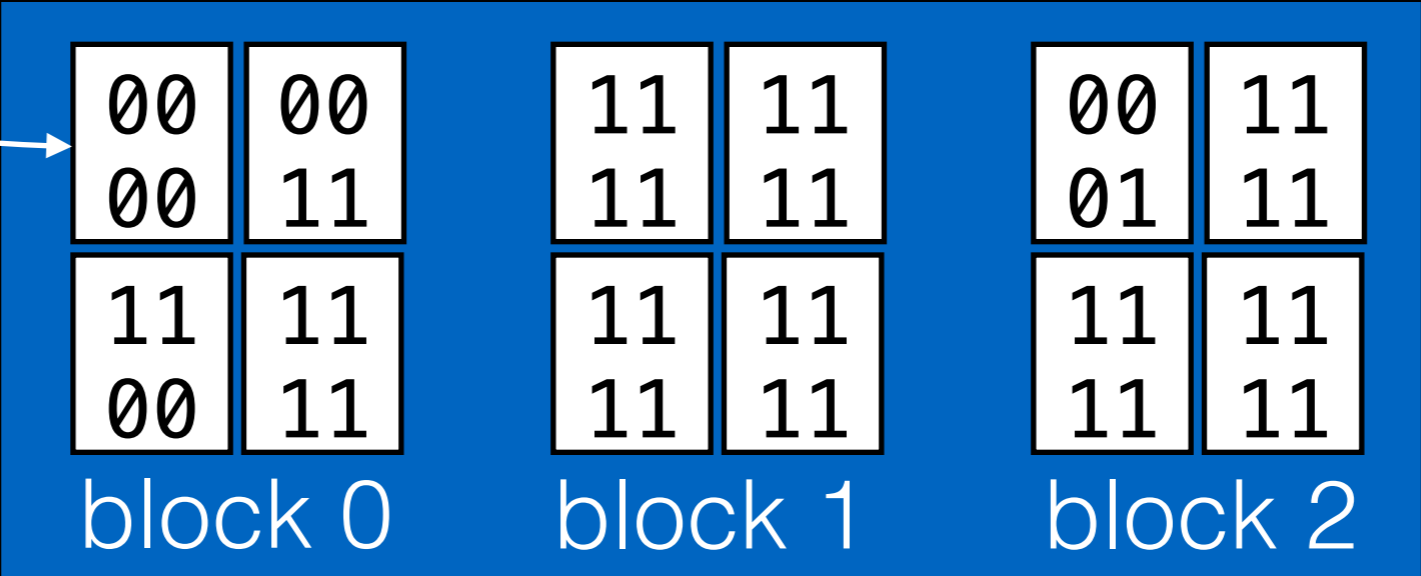
Memory:



FS wants to write 0001



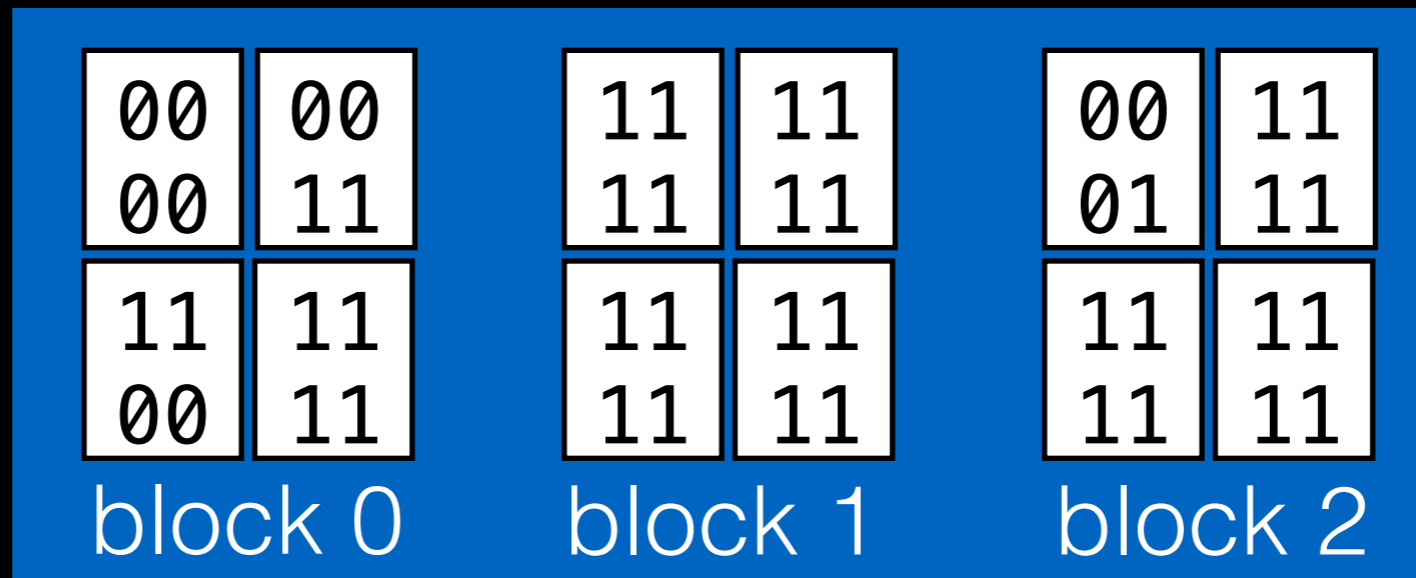
Flash:



Memory:



Flash:



Memory:

00	00
00	11
11	11
00	11

read all other
pages in block

Flash:

00	00
00	11
11	11
00	11

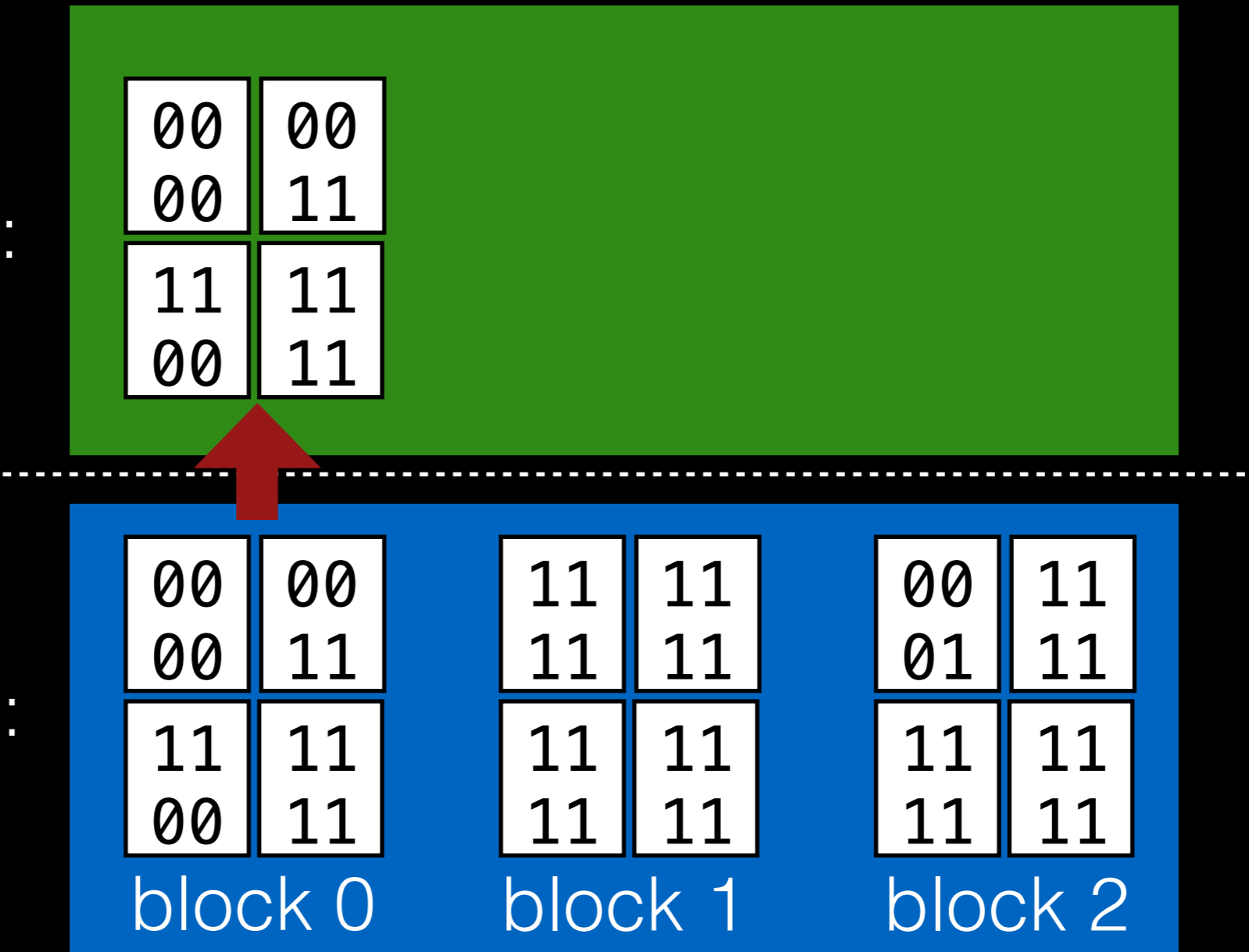
block 0

11	11
11	11
11	11
11	11

block 1

00	11
01	11
11	11
11	11

block 2



Memory:

00	00
00	11
11	11
00	11

Flash:

00	00
00	11
11	11
00	11

block 0

11	11
11	11
11	11
11	11

block 1

00	11
01	11
11	11
11	11

block 2

Memory:

00	00
01	11
11	11
00	11

modify target
page in memory

Flash:

00	00
00	11

11	11
00	11

block 0

11	11
11	11

11	11
11	11

block 1

00	11
01	11

11	11
11	11

block 2

Memory:

00	00
01	11
11	11
00	11

Flash:

00	00
00	11
11	11
00	11

block 0

11	11
11	11
11	11
11	11

block 1

00	11
01	11
11	11
11	11

block 2

Memory:

00	00
01	11
11	11
00	11

erase block

Flash:

11	11
11	11
11	11
11	11

block 0

11	11
11	11
11	11
11	11

block 1

00	11
01	11
11	11
11	11

block 2

Memory:

00	00
01	11
11	11
00	11

Flash:

11	11
11	11

11	11
11	11

block 0

11	11
11	11

11	11
11	11

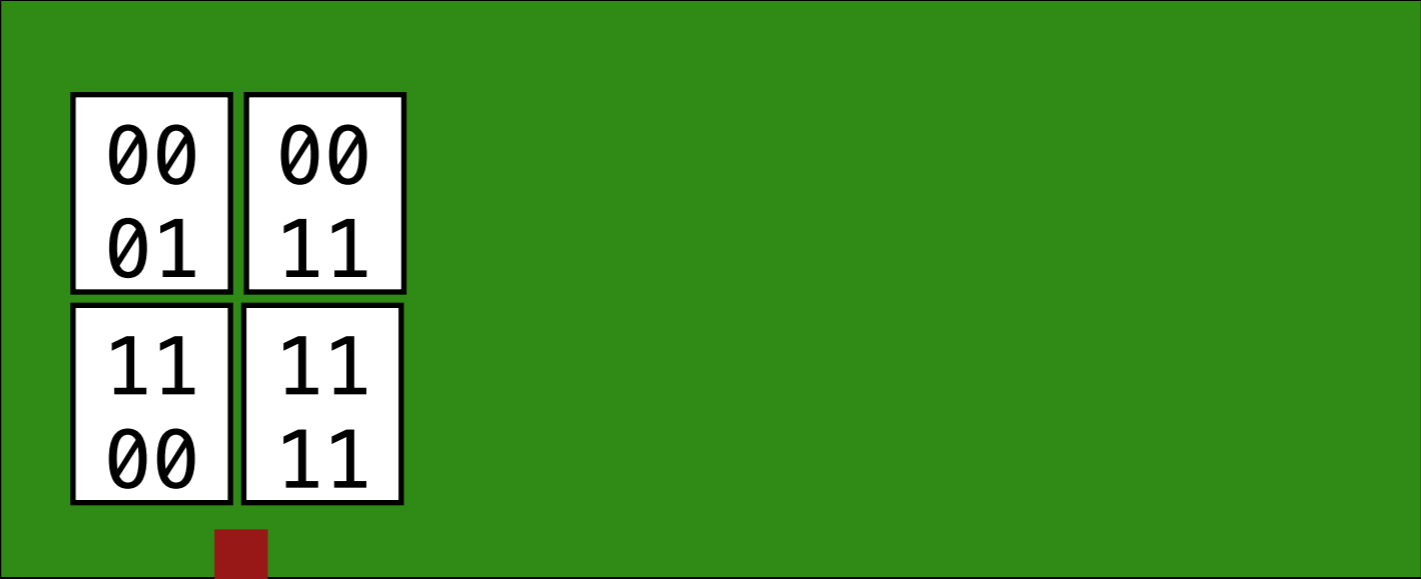
block 1

00	11
01	11

11	11
11	11

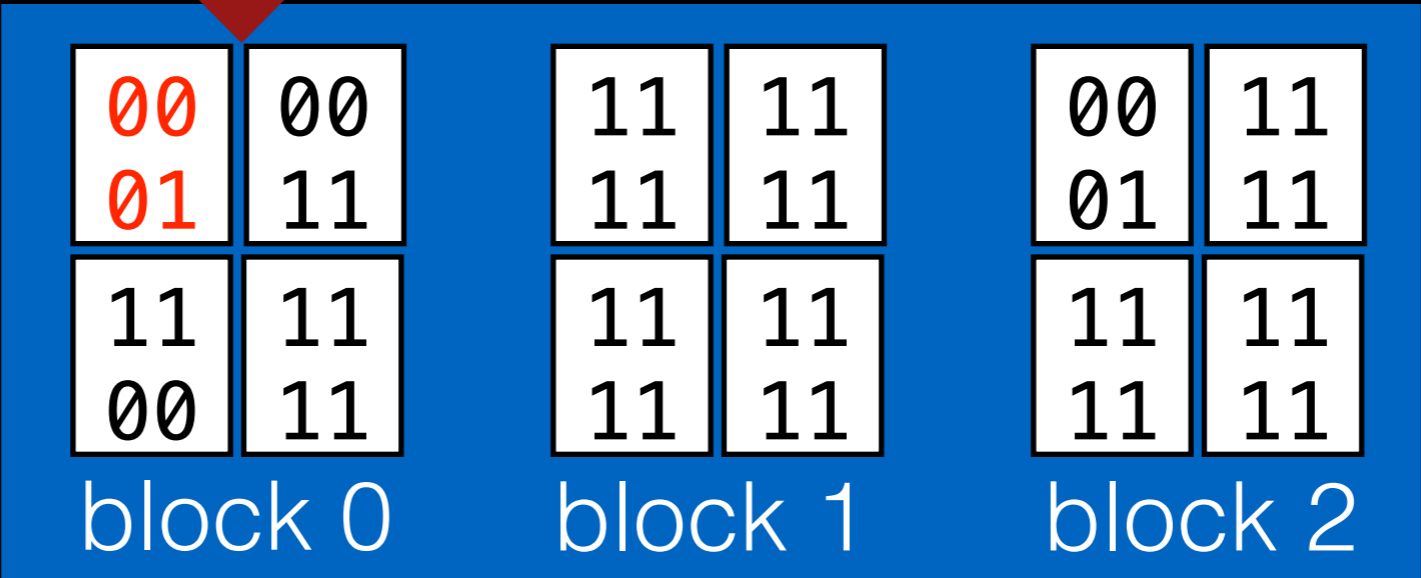
block 2

Memory:



program all
pages in block

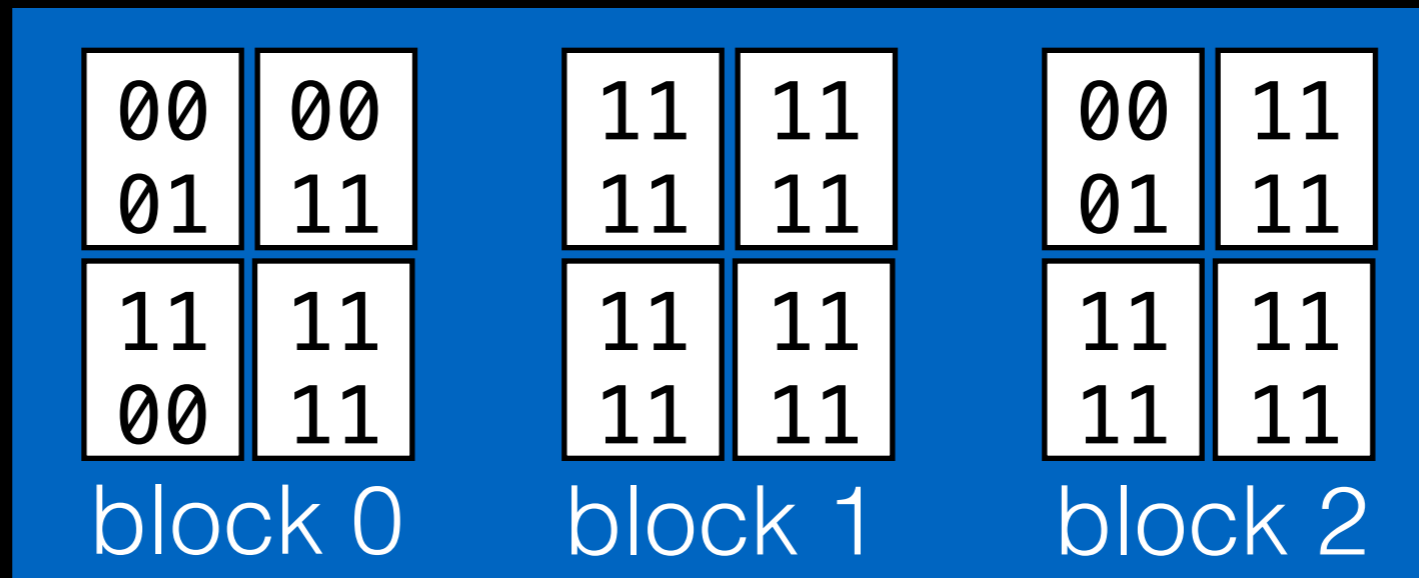
Flash:



Memory:



Flash:



Write Amplification

Random writes are extremely expensive!

Writing one **2KB** page may cause:

- read, erase, and program of **256KB** block.

Write Amplification

Random writes are extremely expensive!

Writing one **2KB** page may cause:

- read, erase, and program of **256KB** block.

Would FFS or LFS be better with flash?

File Systems over Flash

Copy-On-Write FS may prevent some expensive random writes.

What about **wear leveling**? LFS won't do this.

What if we want to use some other FS?

Better Solution

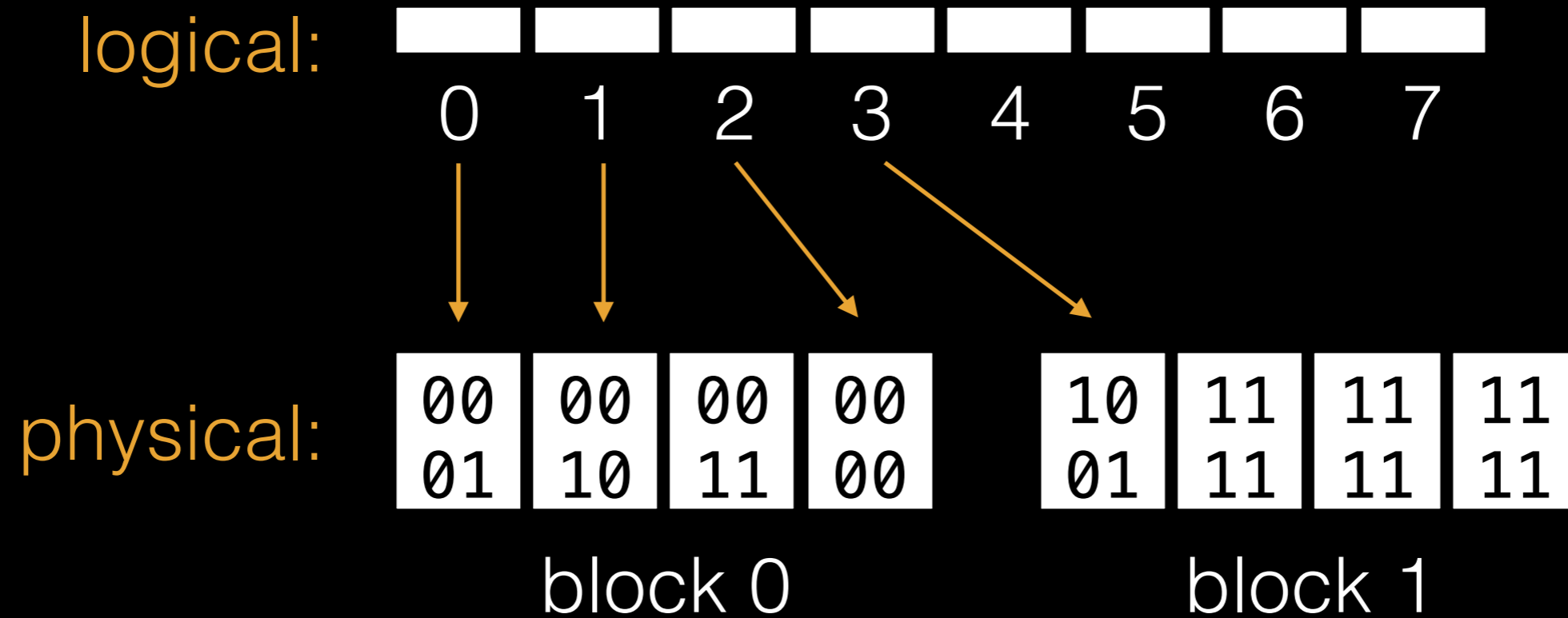
Add **copy-on-write** layer between FS and flash.
Avoids RMW (read-modify-write) cycle.

Translate logical device addrs to physical addrs.

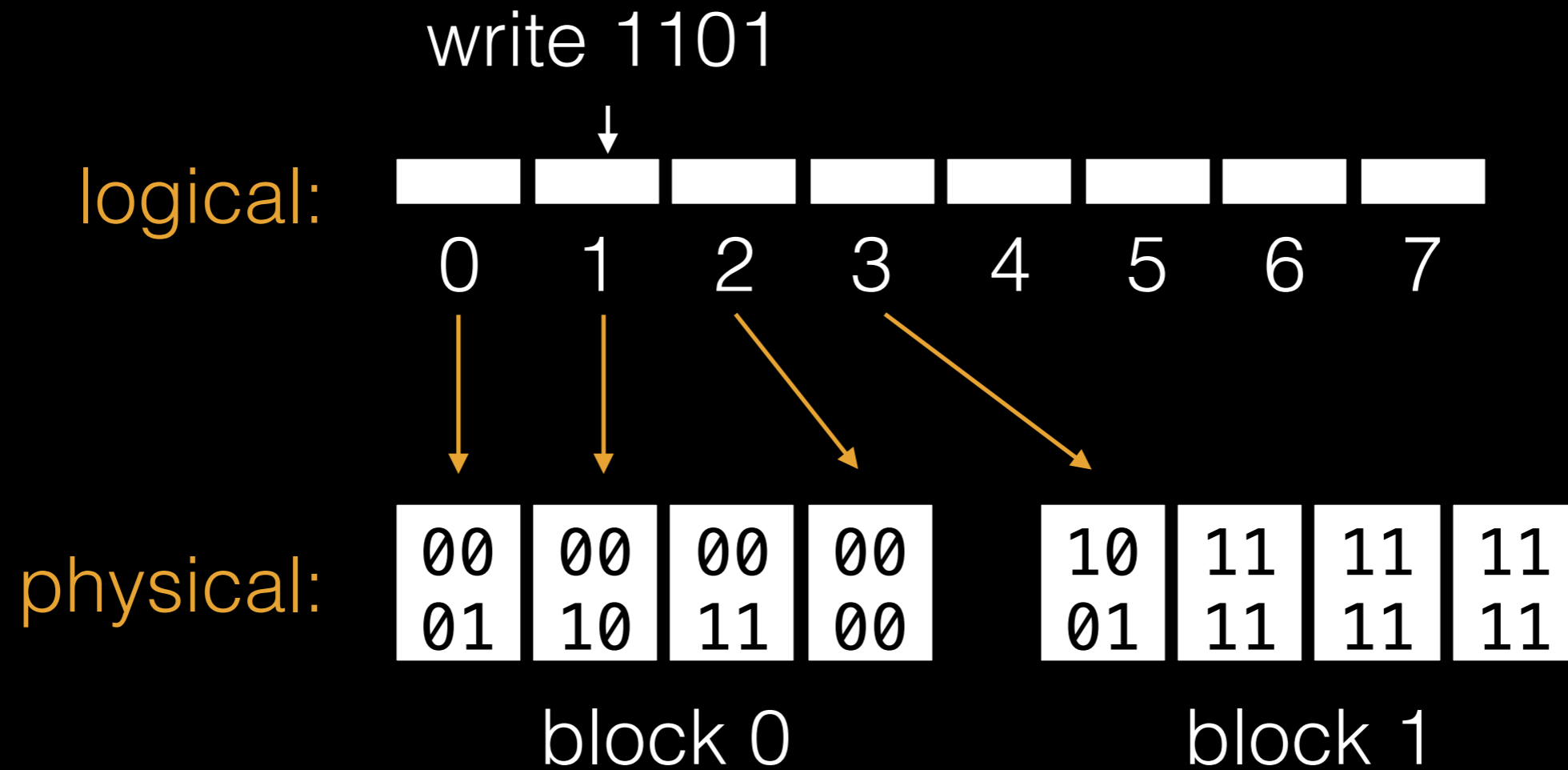
FTL: Flash Translation Layer.

Should translation use math or data structure?

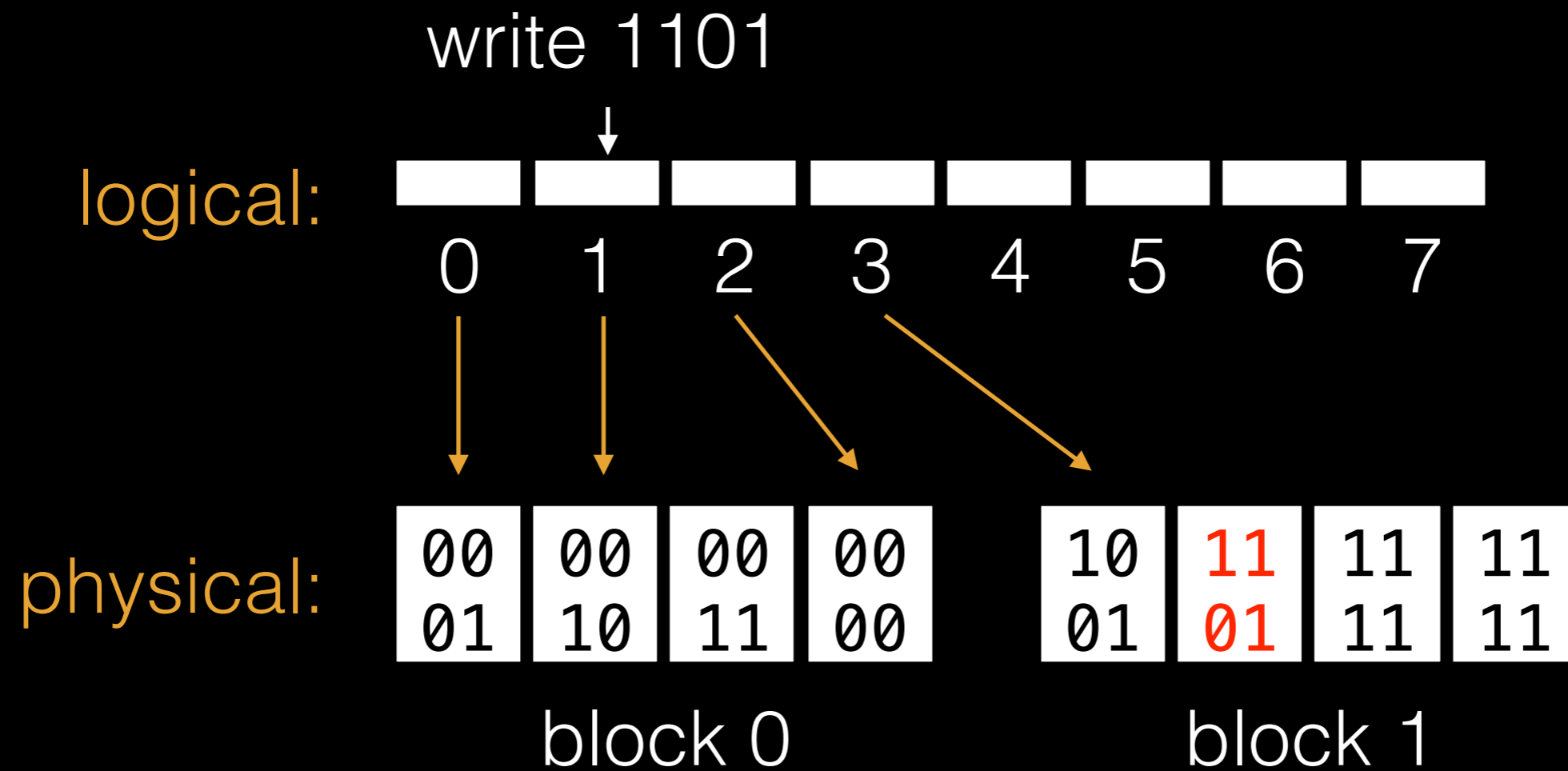
Flash Translation Layer



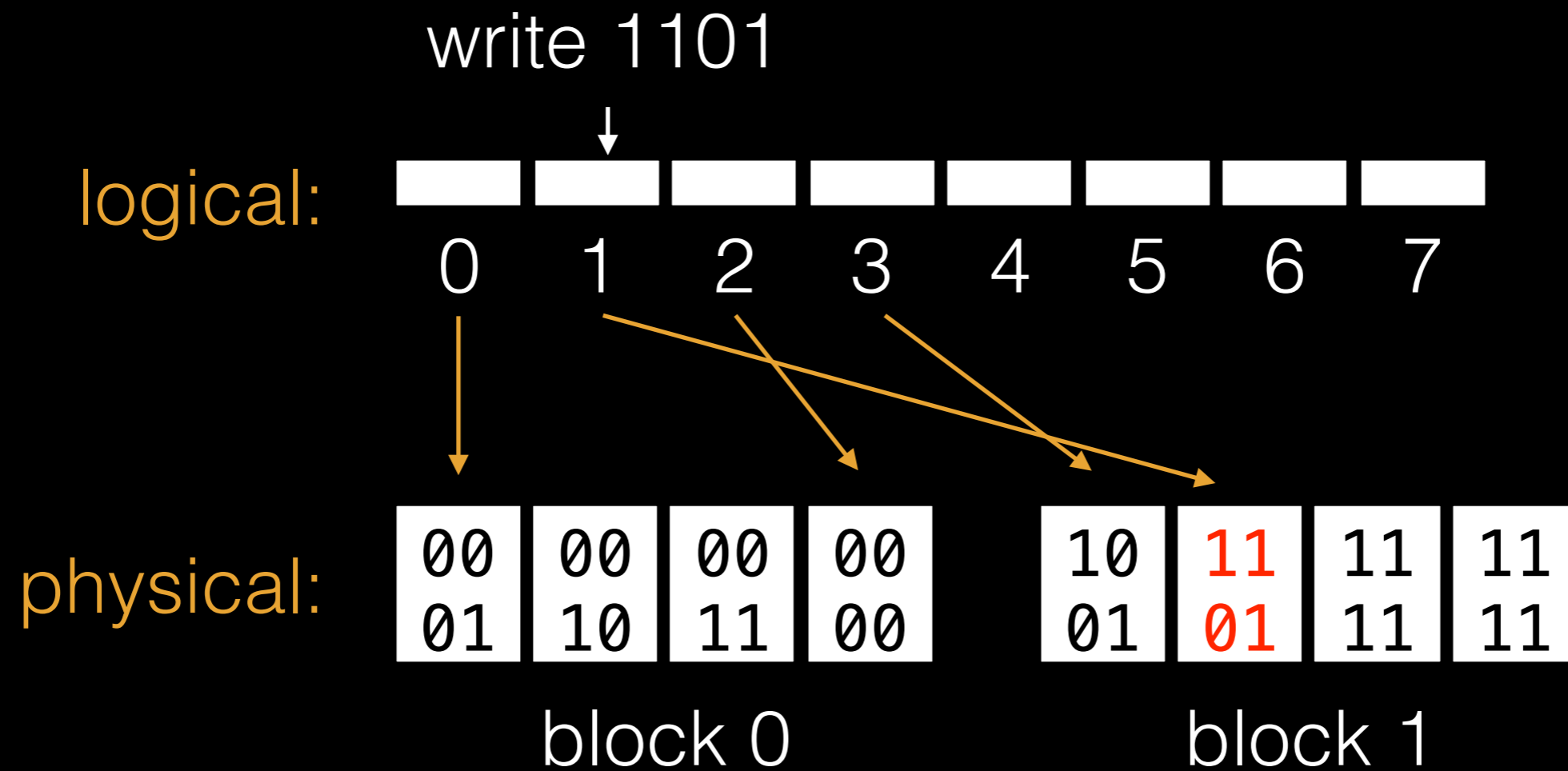
Flash Translation Layer



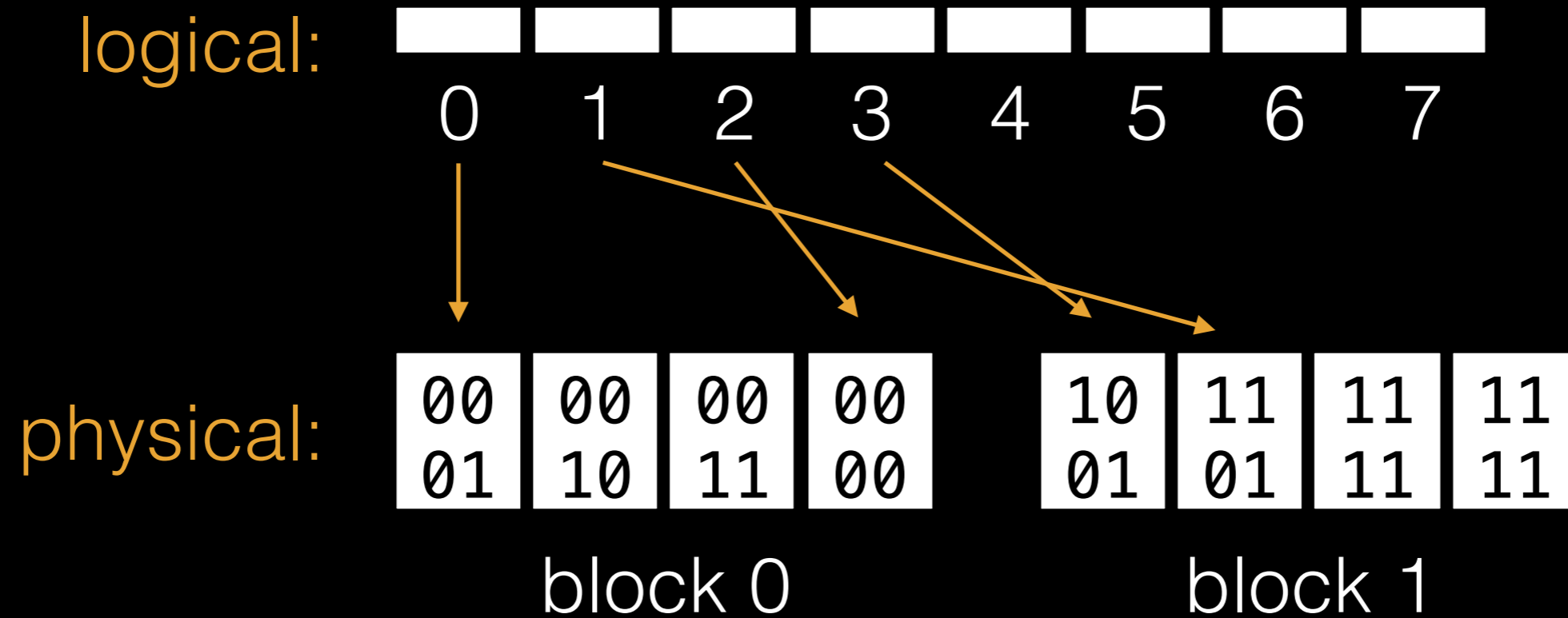
Flash Translation Layer



Flash Translation Layer



Flash Translation Layer



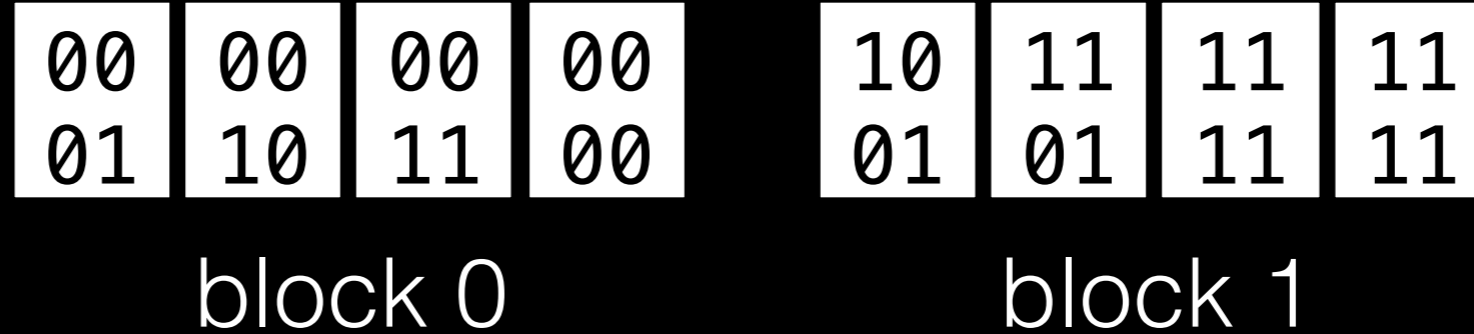
Flash Translation Layer

logical:



must eventually
be garbage collected

physical:



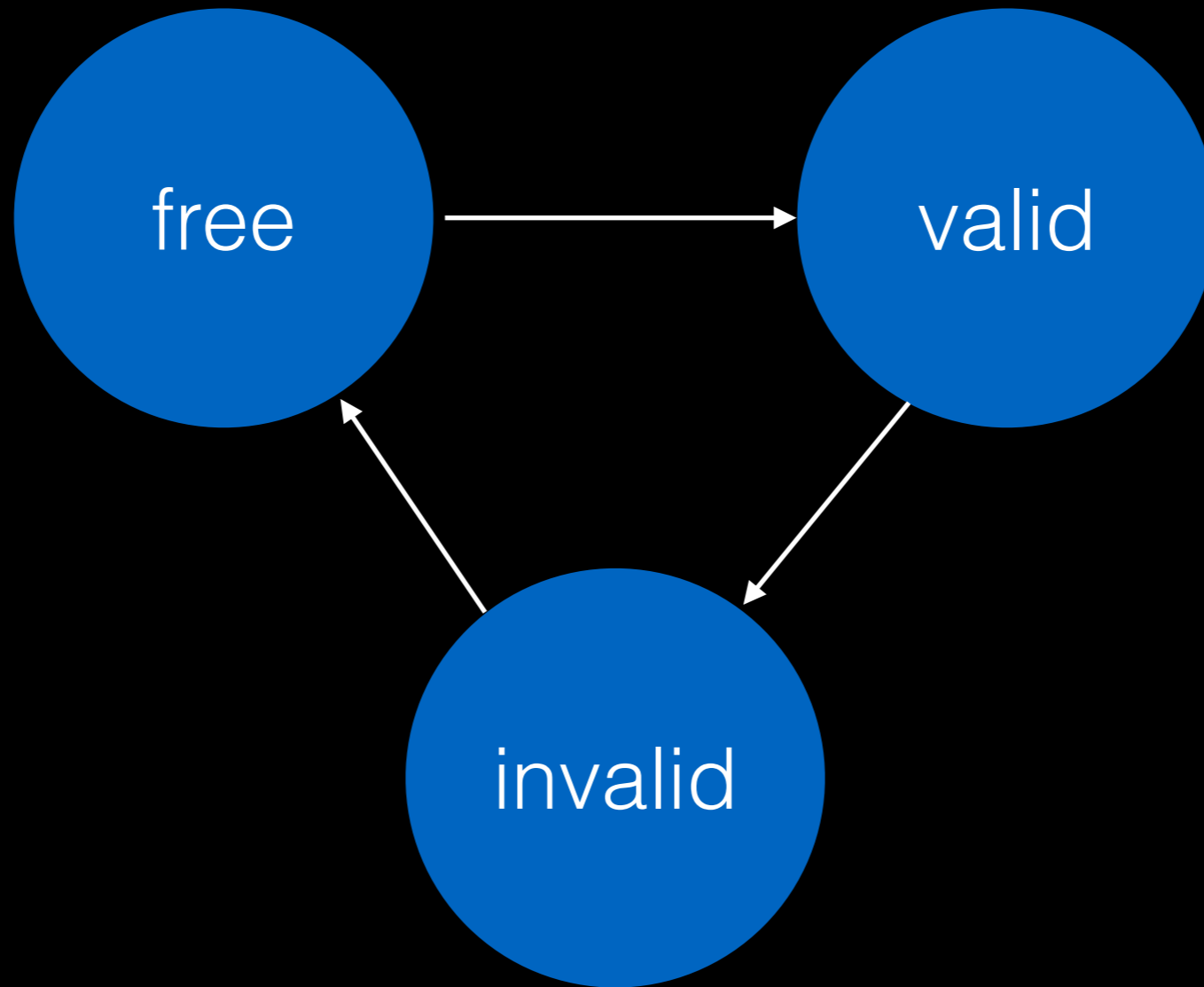
FTL

Could be implemented as device driver or in firmware (usually the latter).

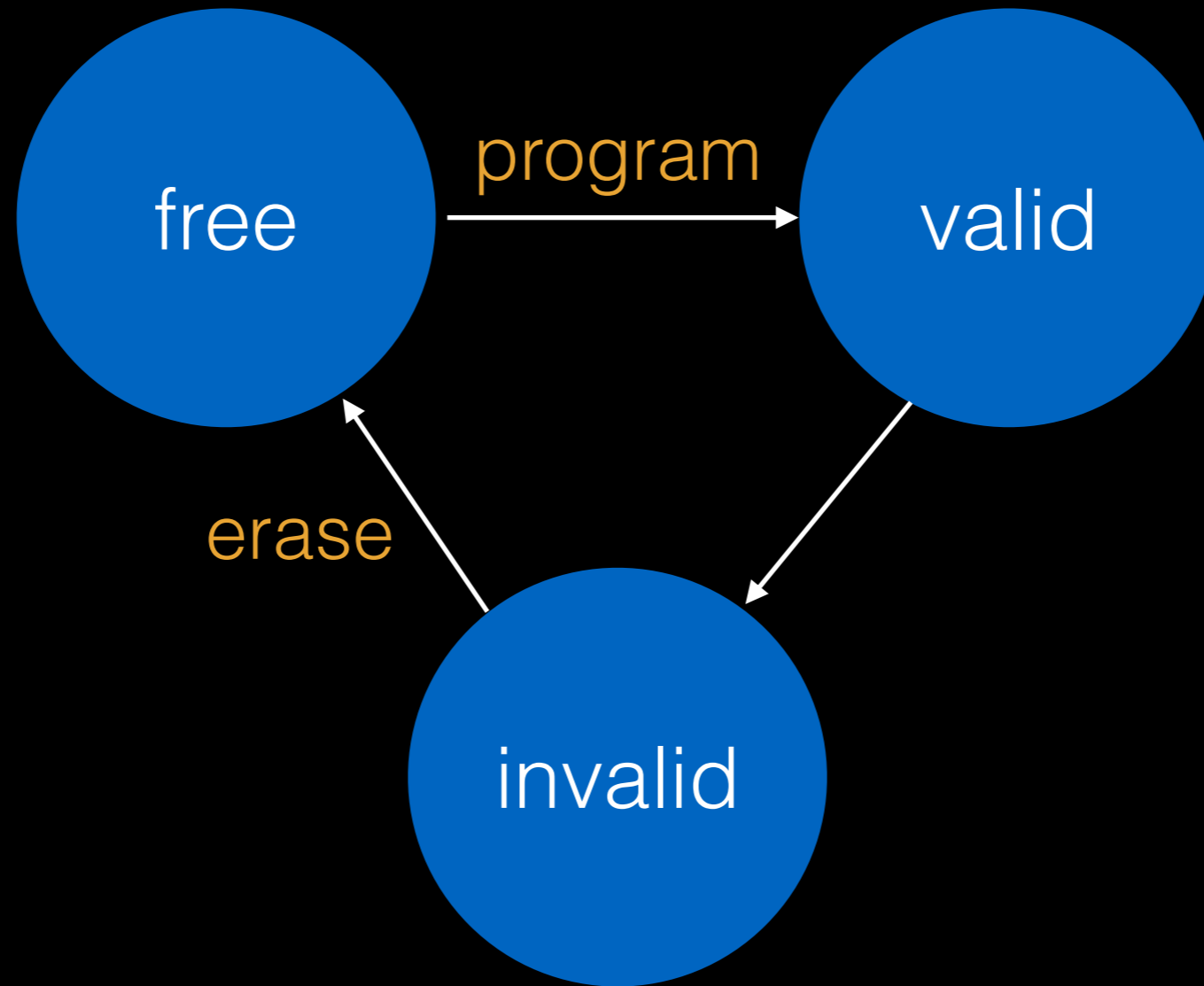
Where to store mapping? SRAM.

Physical pages can be in three states:
- valid, invalid, free

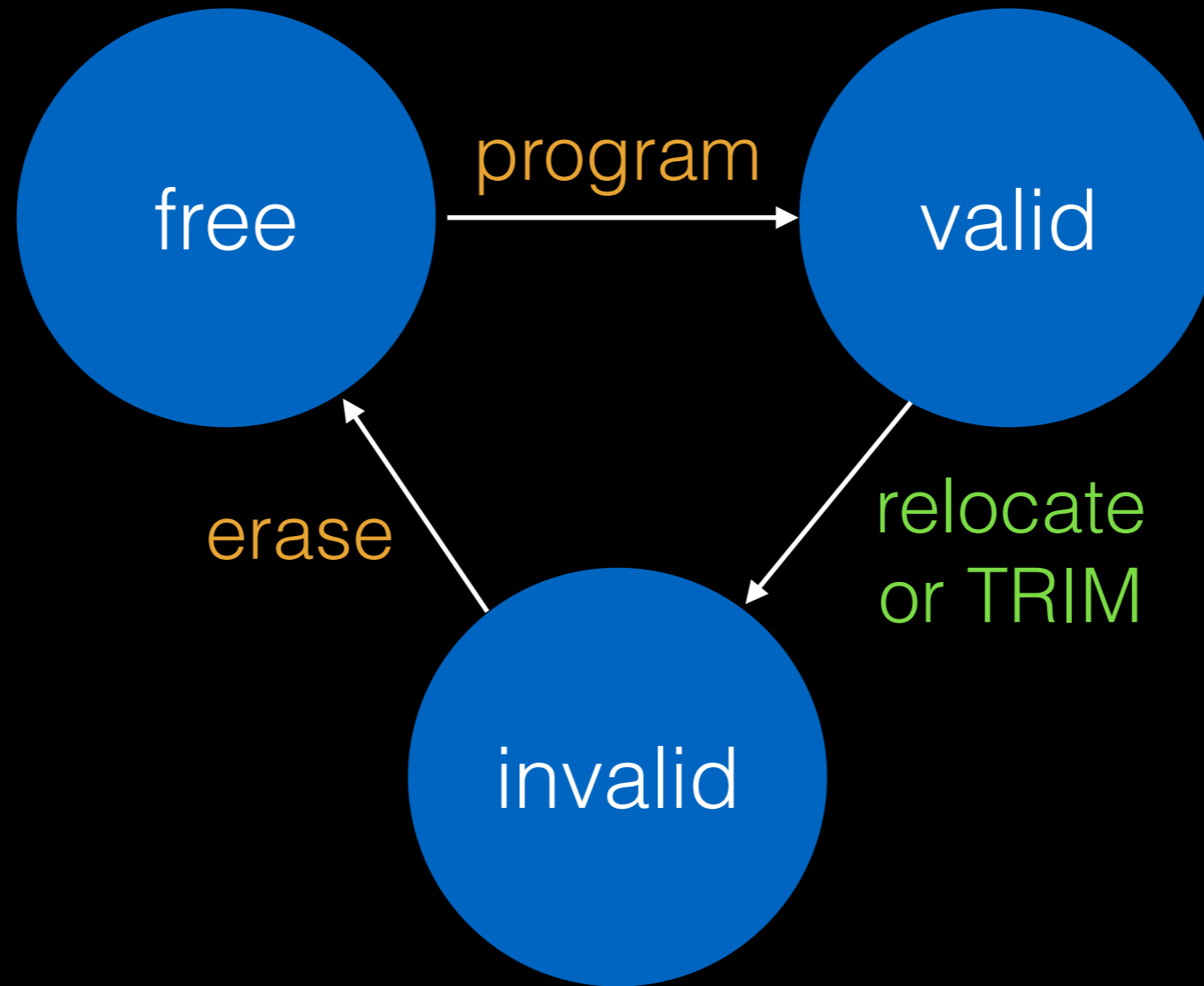
States



States

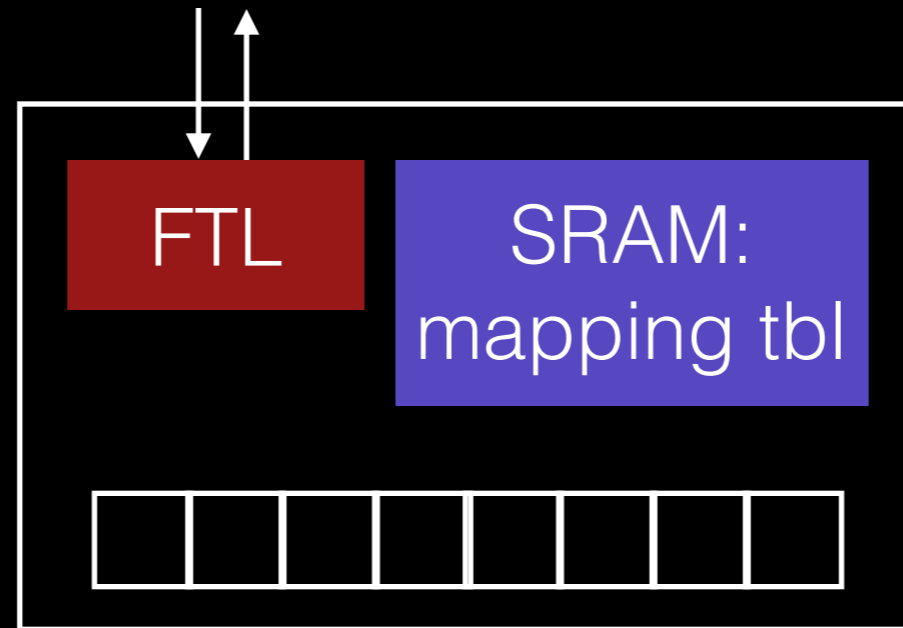


States



SSD Architecture

SSD: looks like disk



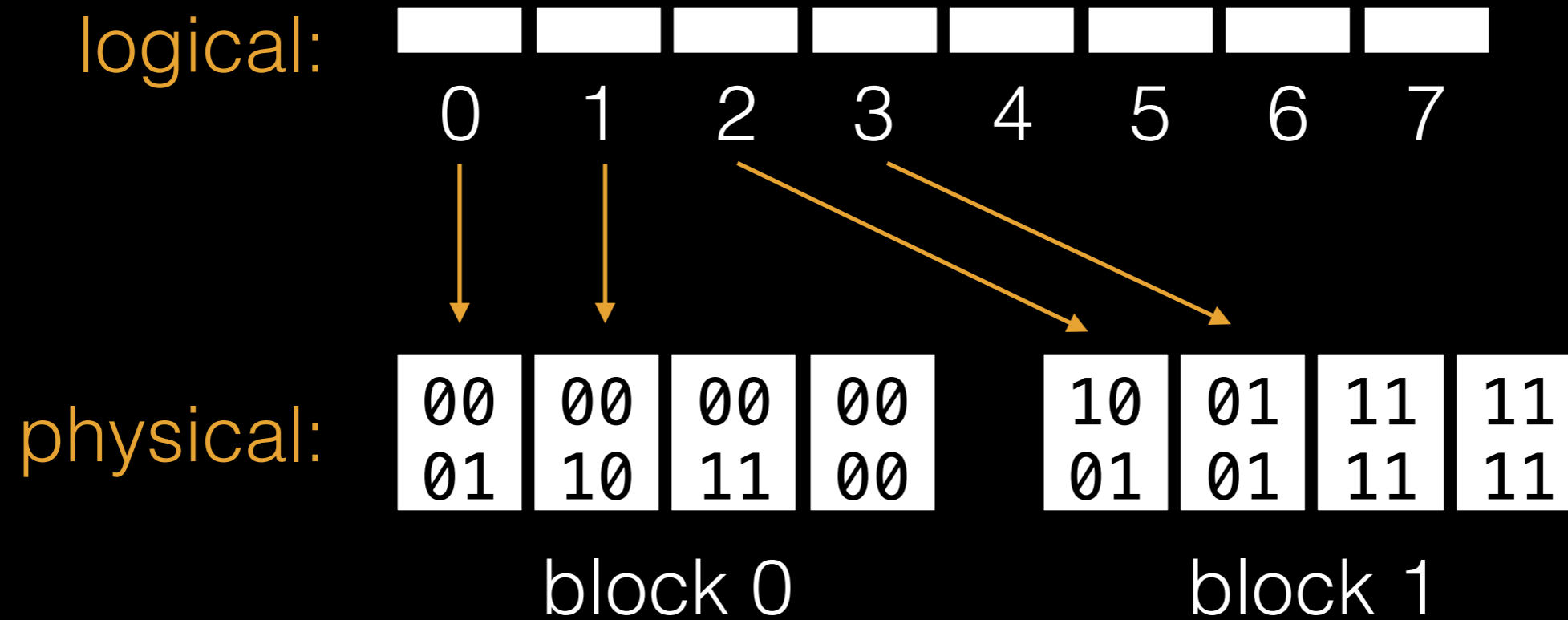
Problem: Big Mapping Table

Assume 200GB device, 2KB pages, 4-byte entries.

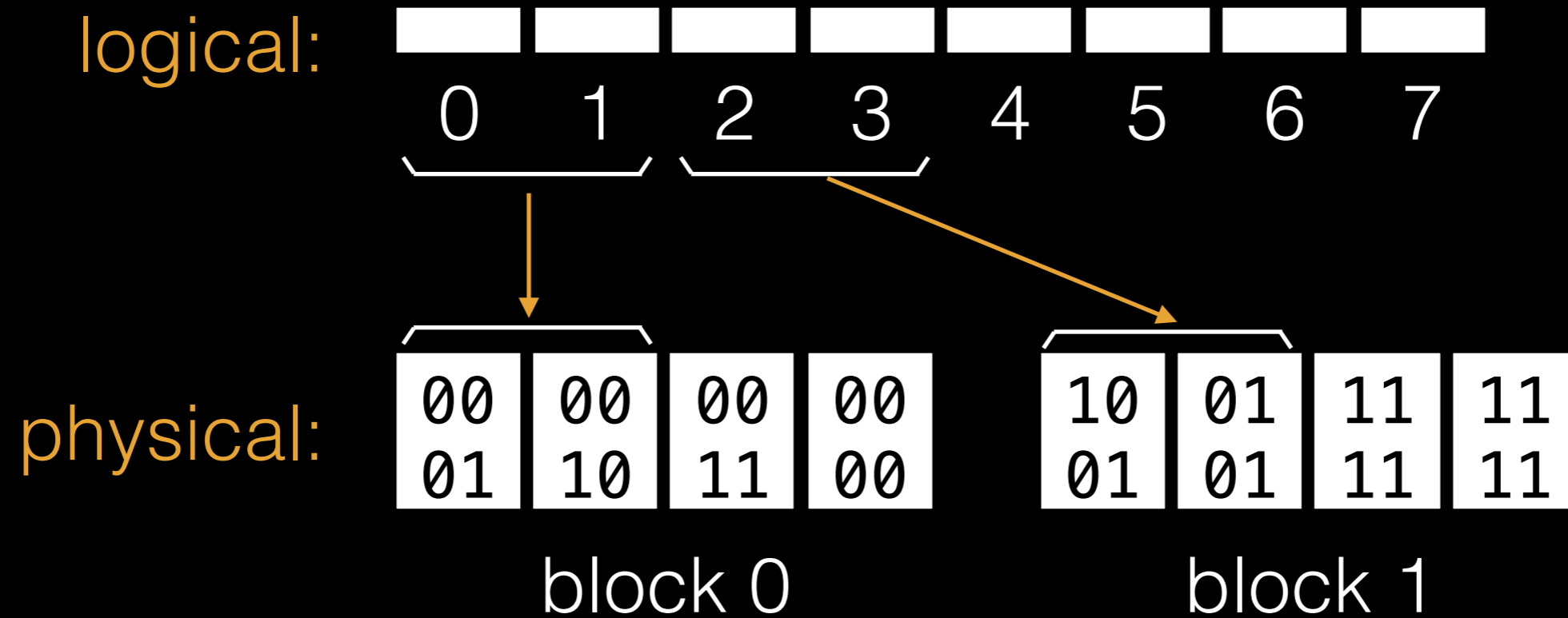
SRAM needed: $(200\text{GB} / 2\text{KB}) * 4 \text{ bytes} = 400 \text{ MB}$.

Too big, SRAM is expensive!

Page Translations



2-Page Translations

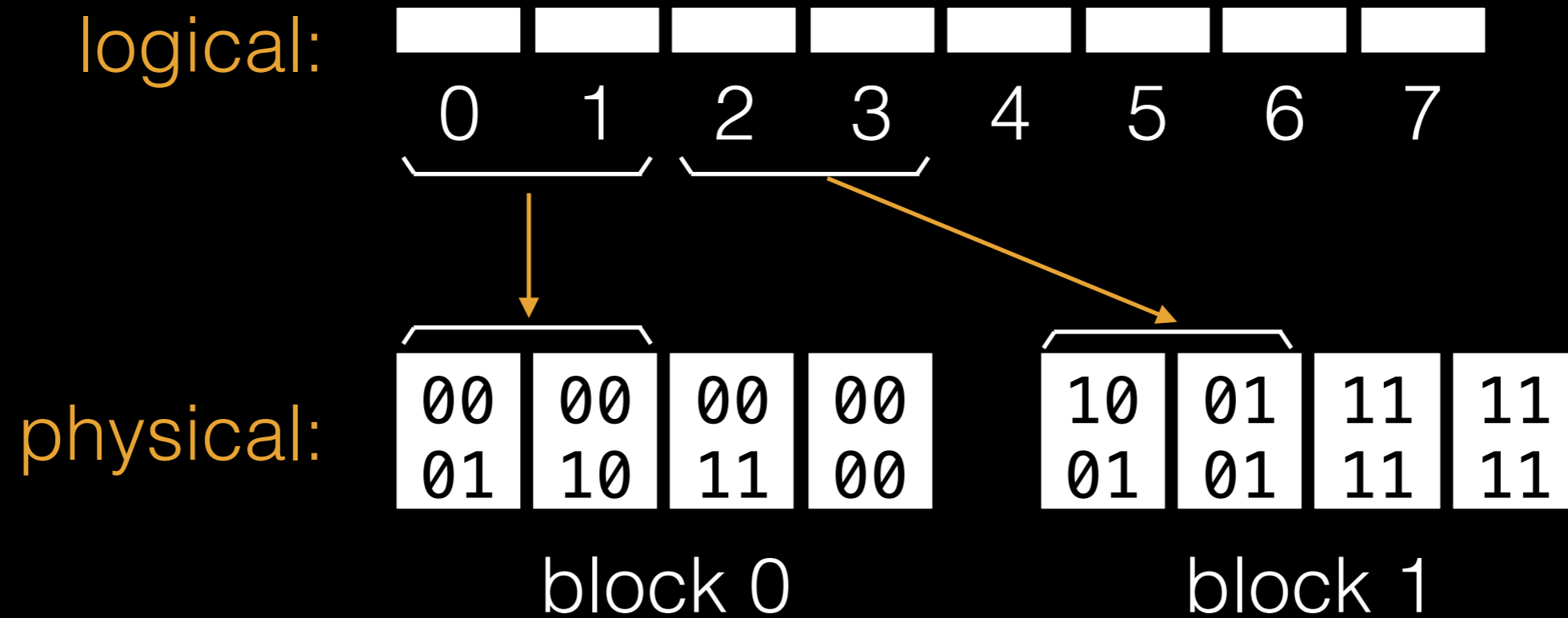


Larger Mappings

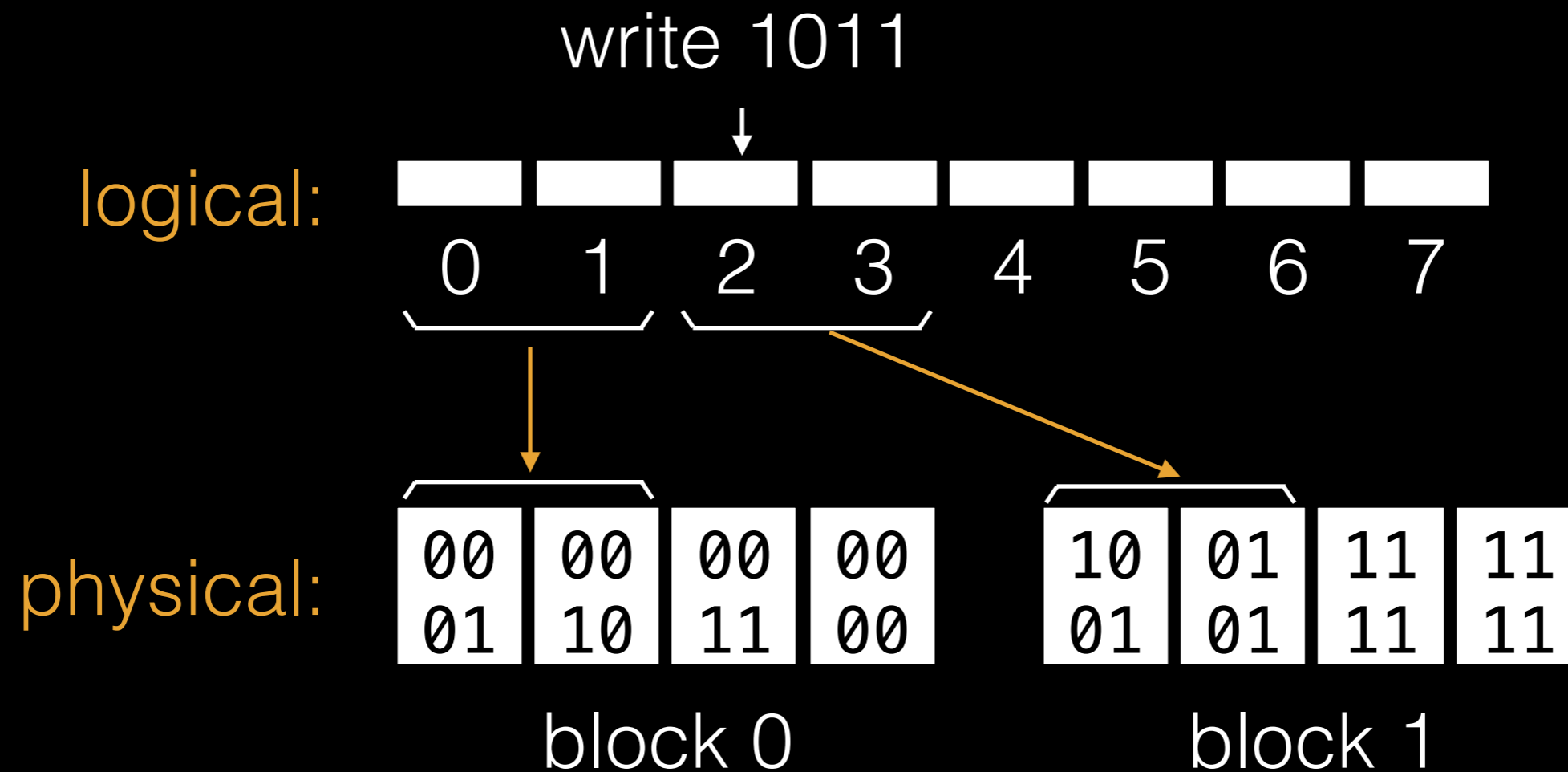
Advantage: larger mappings decrease table size.

Disadvantage?

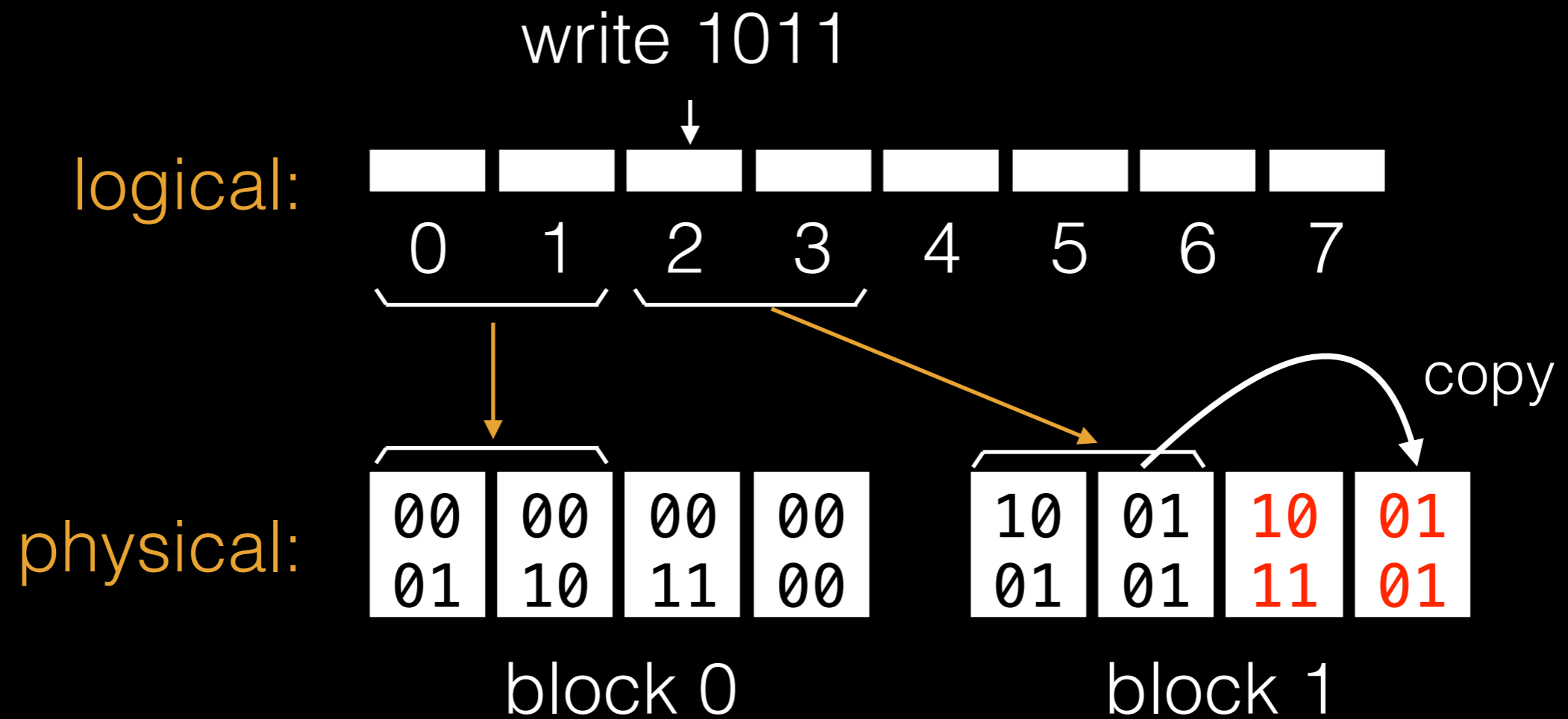
2-Page Translations



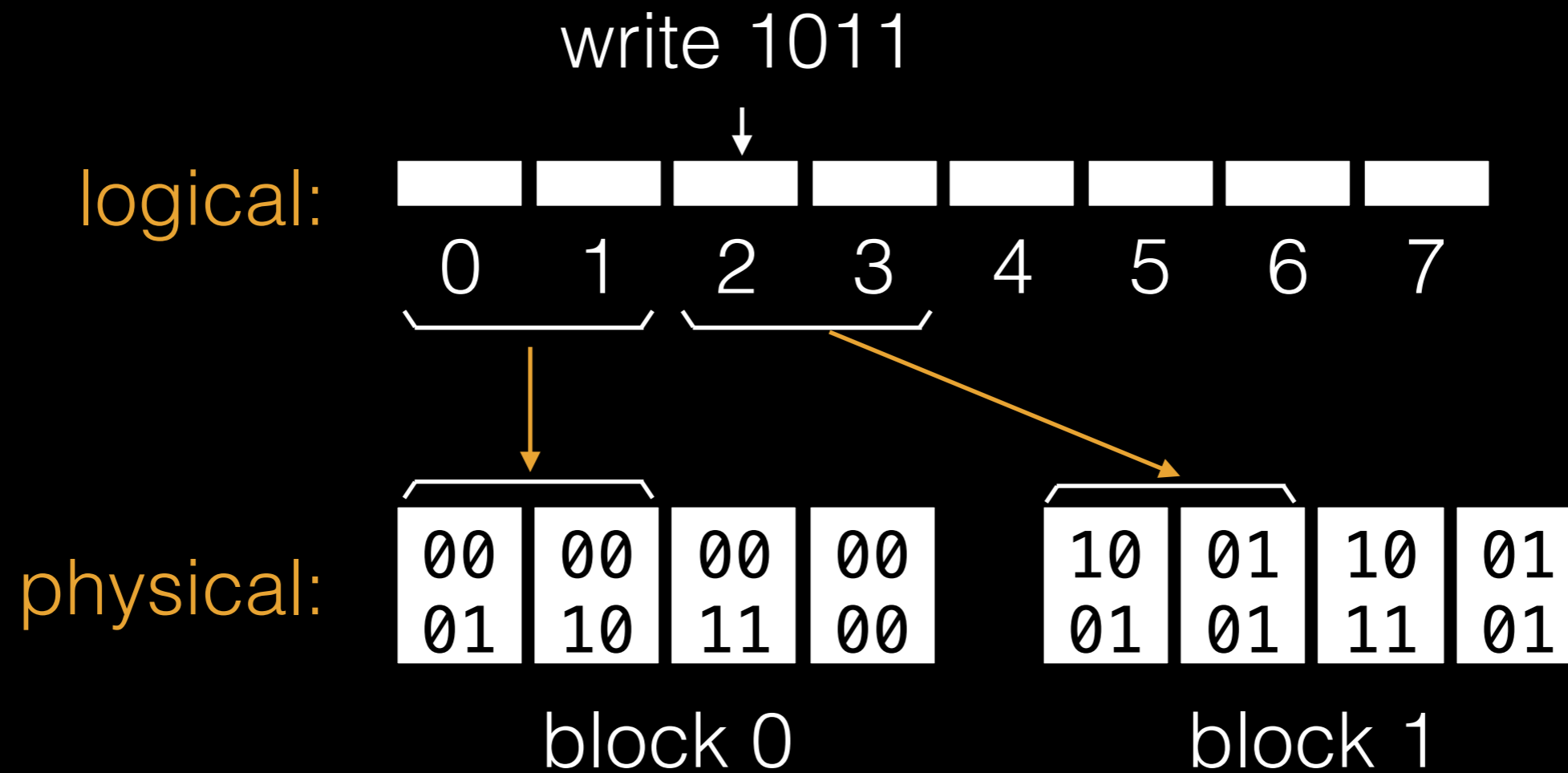
2-Page Translations



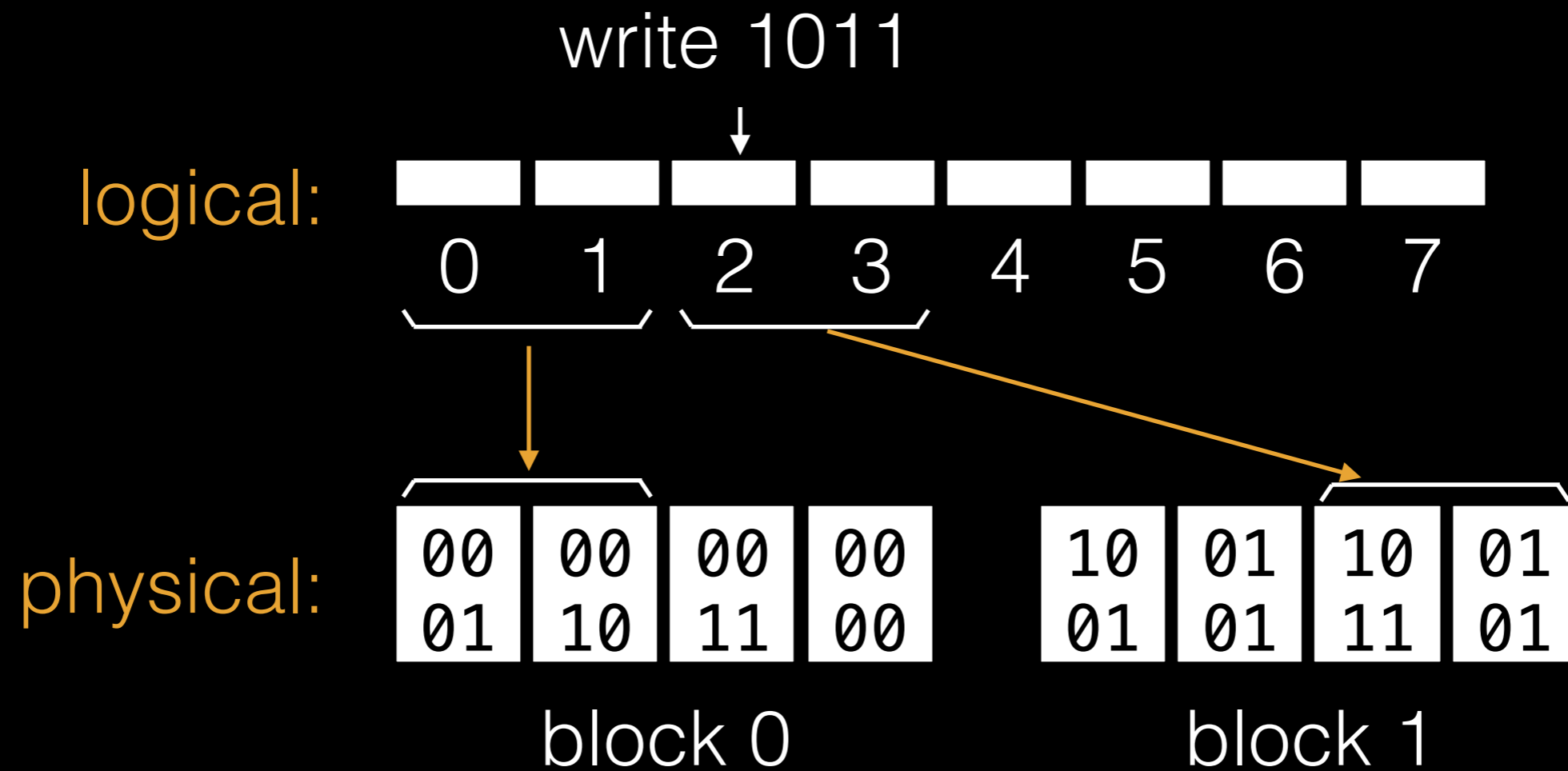
2-Page Translations



2-Page Translations



2-Page Translations



Larger Mappings

Advantage: larger mappings decrease table size.

Disadvantages?

- more read-modify-write updates
- more garbage
- less flexibility for placement

Hybrid FTL

Use course-grained mapping for most (e.g., 95%) of data. Map at **block level**.

Use fine-grained mapping for recent data.
Map at **page level**.

Log Blocks

Write changed pages to designated **log blocks**.

After blocks become full, **merge** changes with old data.

Eventually garbage collect old pages.

Merging

Merging technique depends on I/O pattern.

Three merge types:

- full merge
- partial merge
- switch merge

Merging

Merging technique depends on I/O pattern.

Three merge types:

- full merge
- partial merge
- switch merge

logical:

--	--	--	--

 ...

0 1 2 3



physical:

A	B	C	D
---	---	---	---

block 0

11	11	11	11
11	11	11	11

block 1 (log)

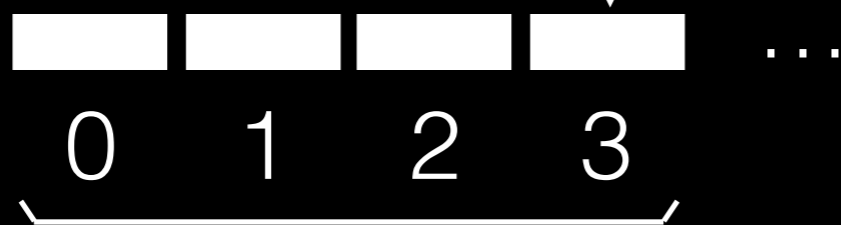
11	11	11	11
11	11	11	11

block 2

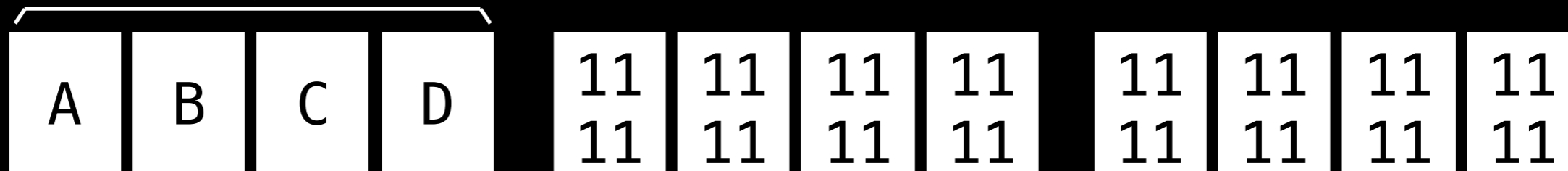
write D2



logical:



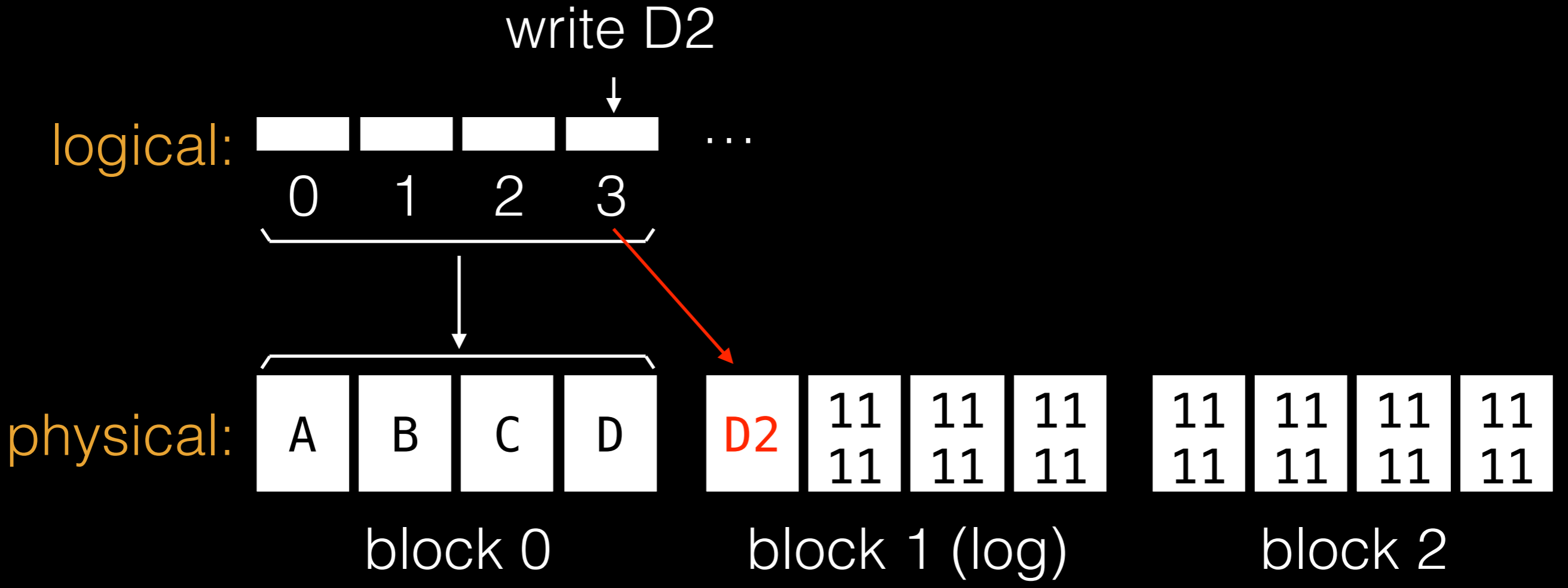
physical:



block 0

block 1 (log)

block 2

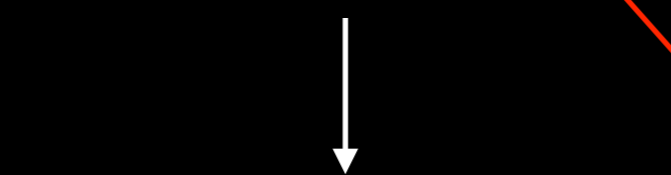


logical:

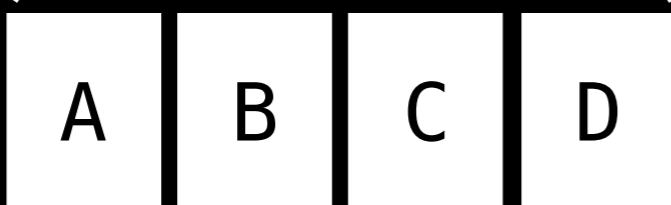
--	--	--	--

 ...

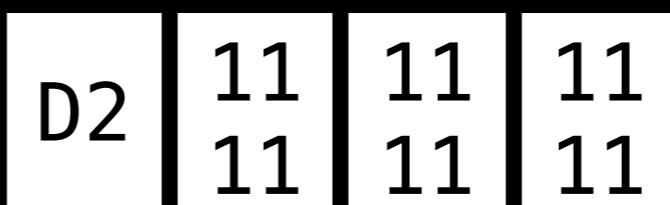
0 1 2 3



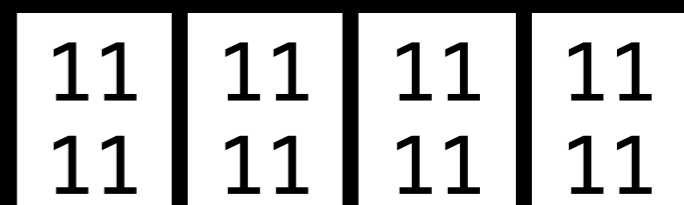
physical:



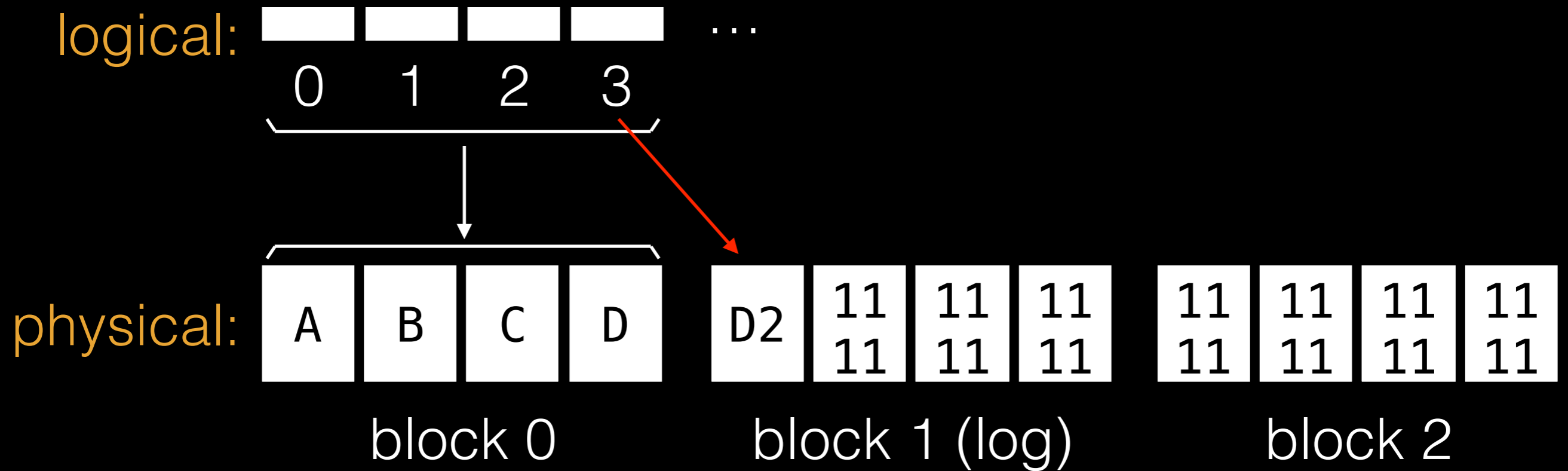
block 0



block 1 (log)



block 2



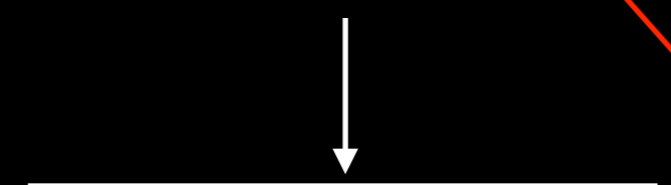
eventually, we need to get rid of red arrows,
as these represent expensive mappings

logical:

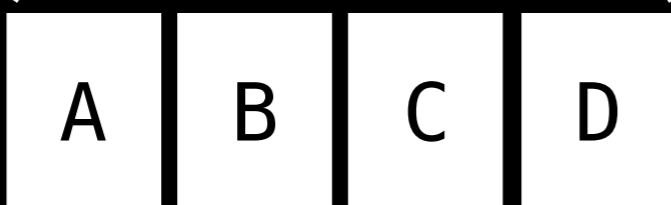
--	--	--	--

 ...

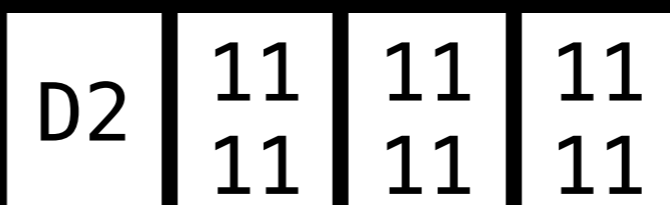
0 1 2 3



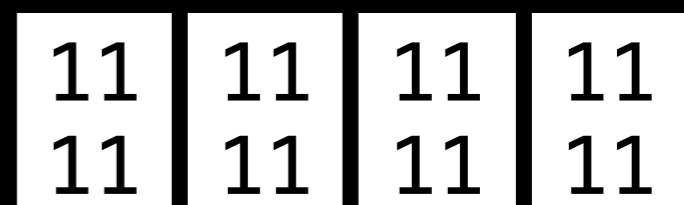
physical:



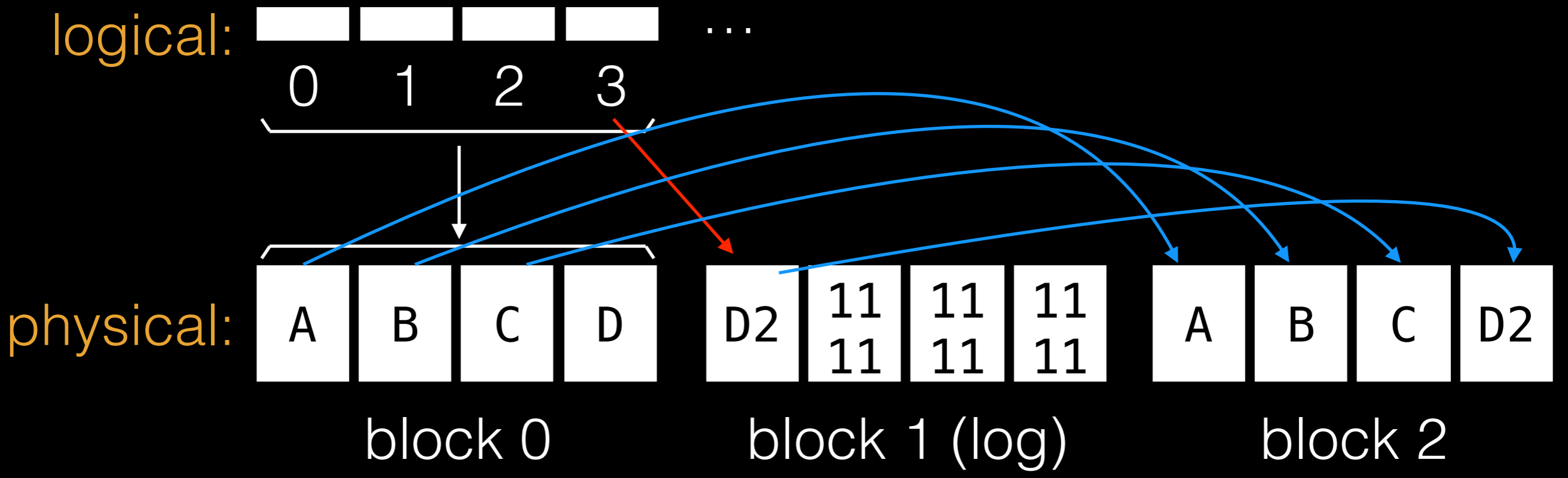
block 0



block 1 (log)



block 2

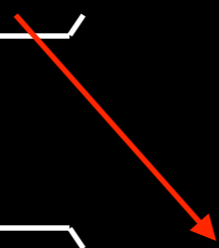
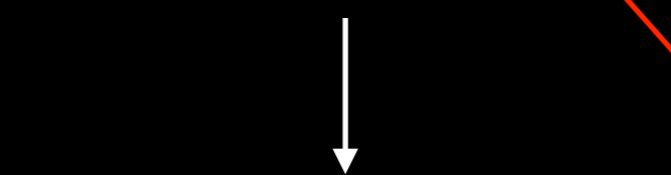


logical:

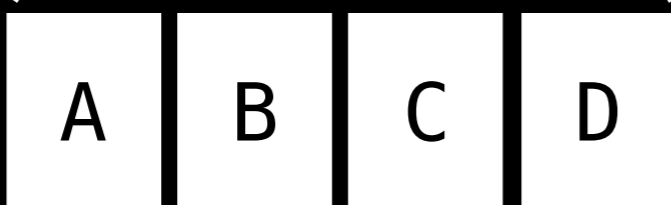
--	--	--	--

 ...

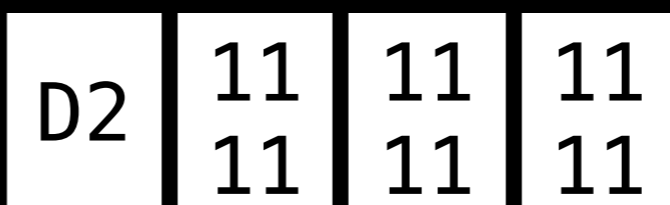
0 1 2 3



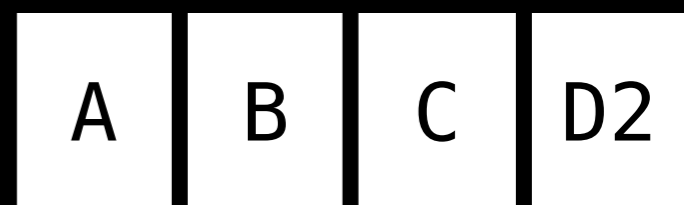
physical:



block 0



block 1 (log)



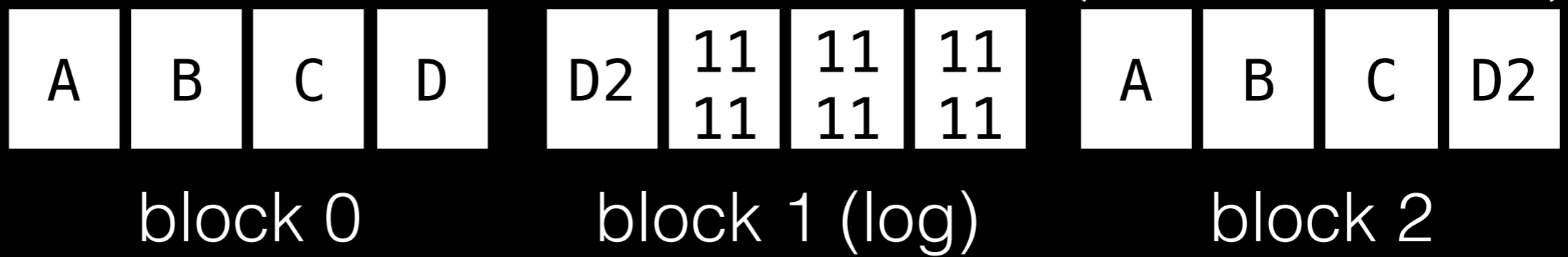
block 2

logical:

0	1	2	3

 ...

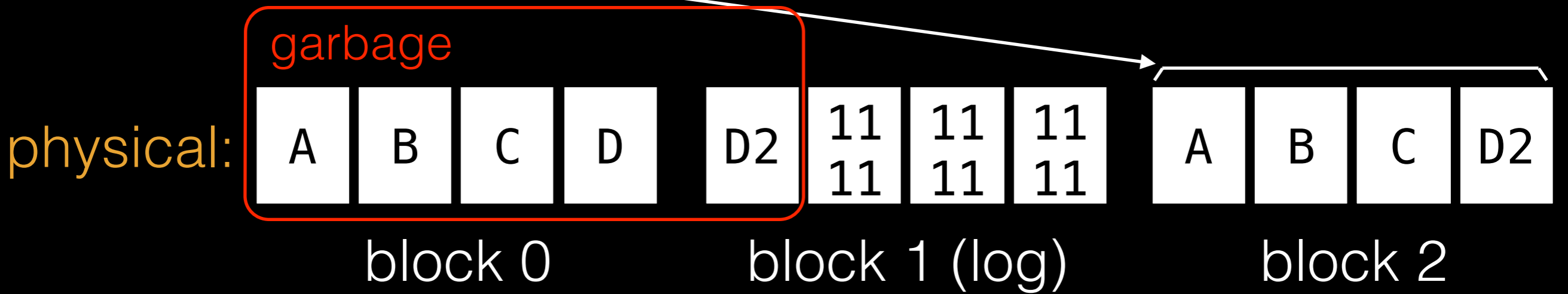
physical:



logical:

0	1	2	3

 ...



Merging

Merging technique depends on I/O pattern.

Three merge types:

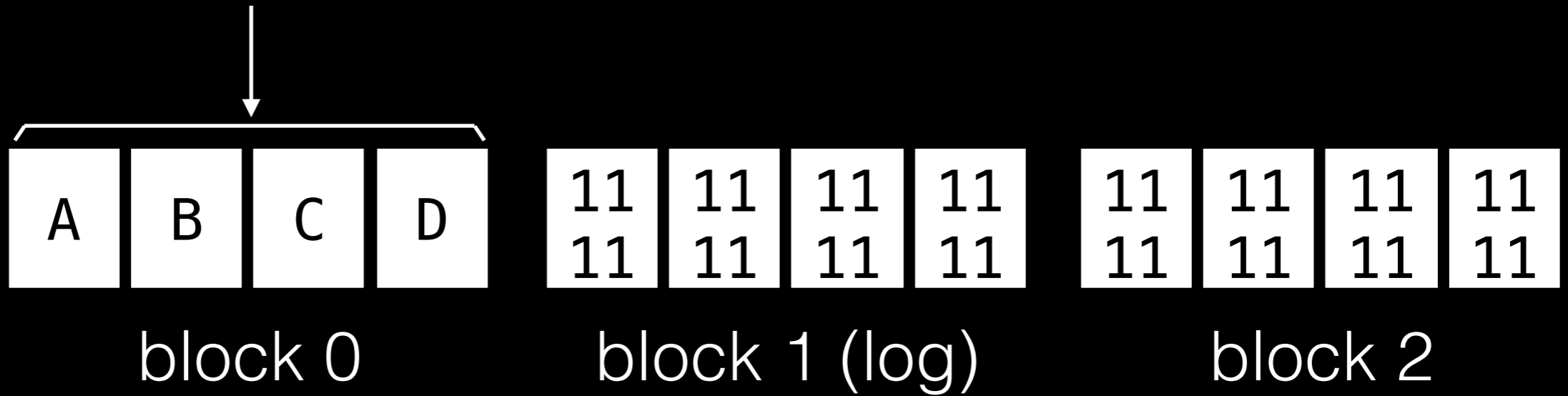
- full merge
- **partial merge**
- switch merge

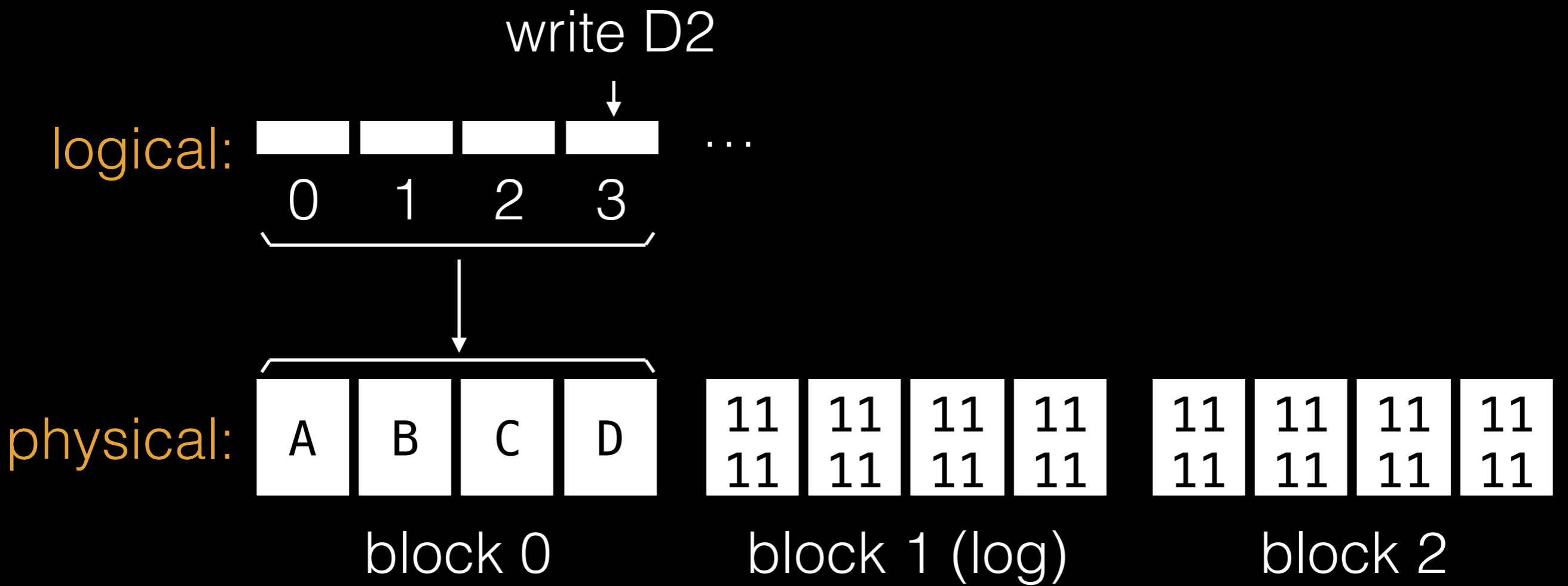
logical:

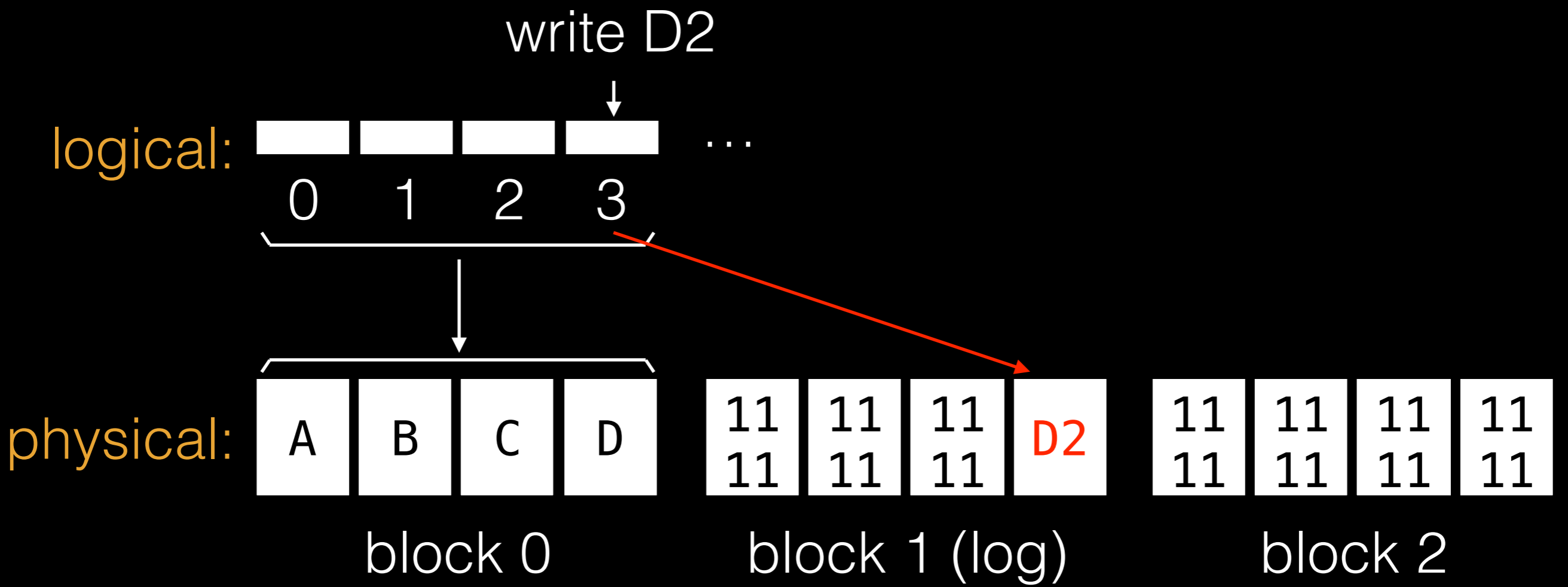
0	1	2	3


 ...

physical:



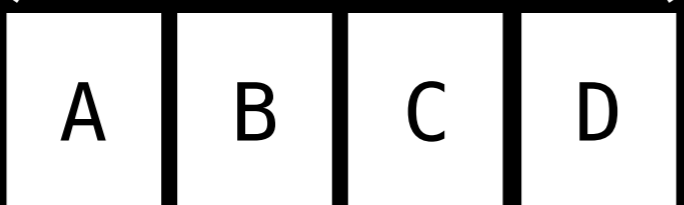




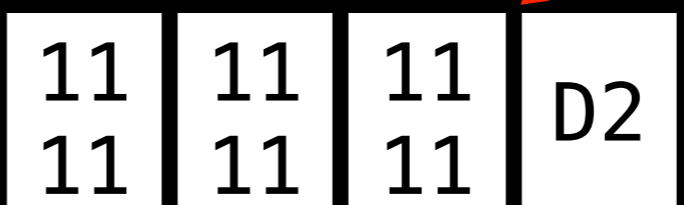
logical:  ...

0 1 2 3

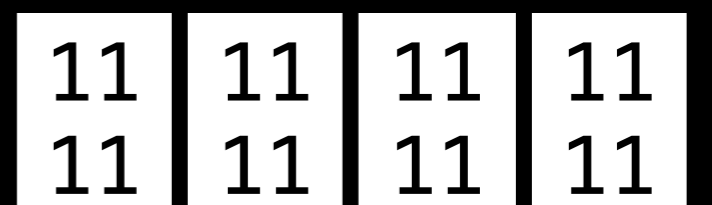
physical:



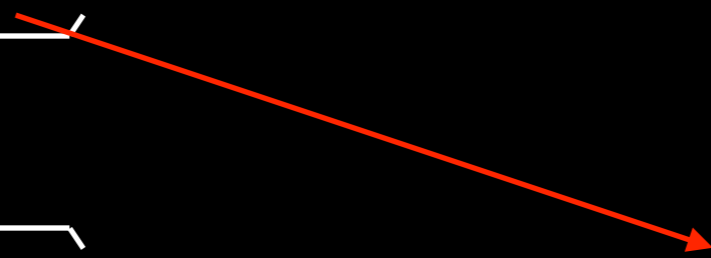
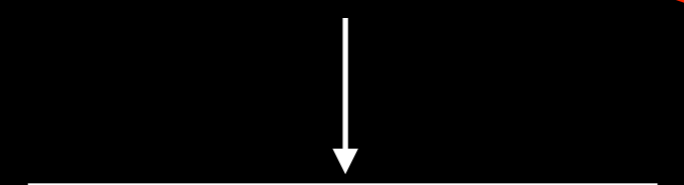
block 0



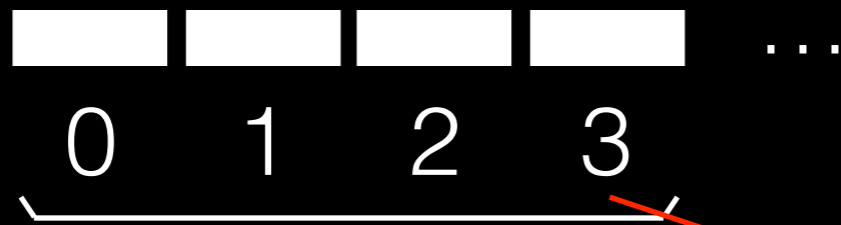
block 1 (log)



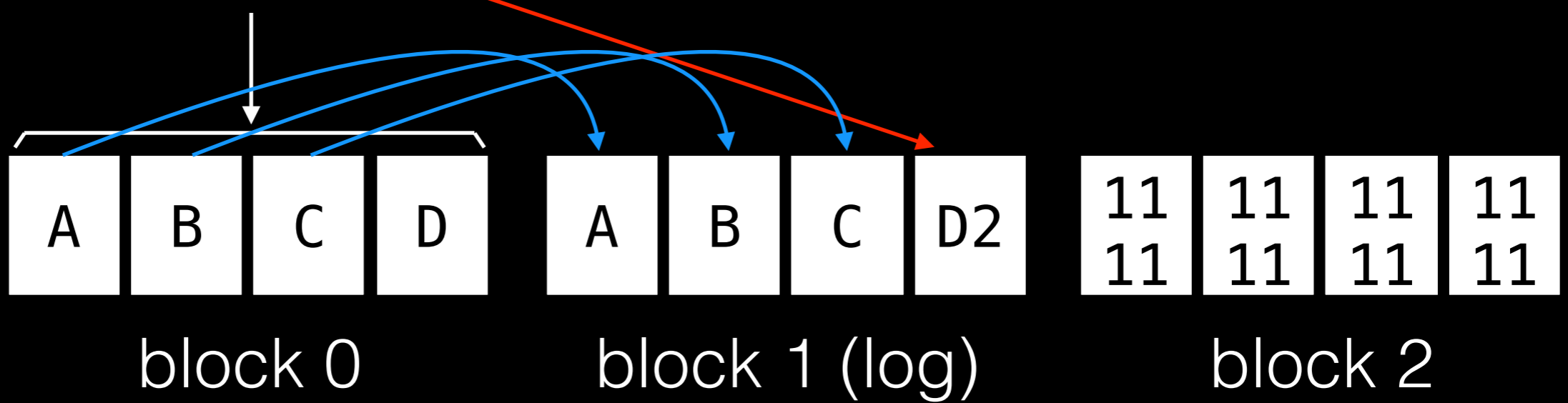
block 2



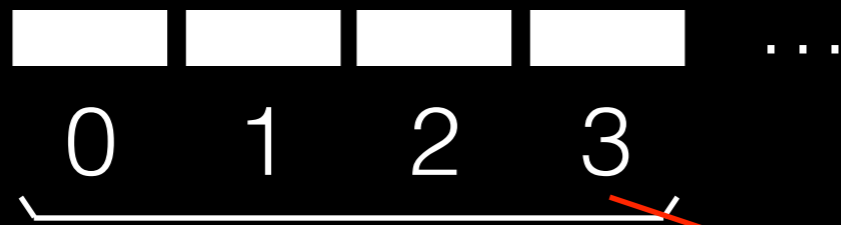
logical:



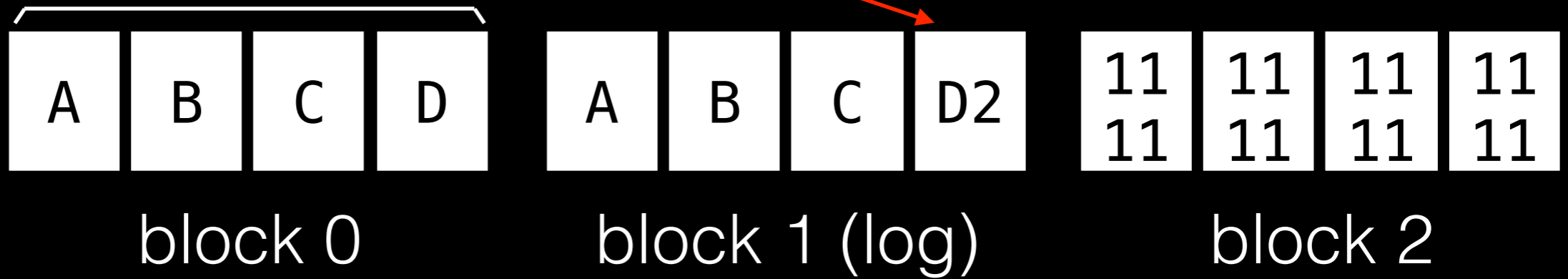
physical:



logical:



physical:

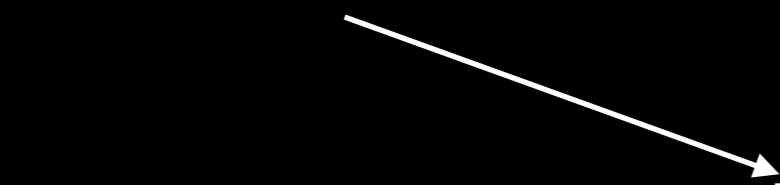


logical:

--	--	--	--

 ...

0 1 2 3



physical:

A	B	C	D
---	---	---	---

block 0

A	B	C	D2
---	---	---	----

block 1

11	11	11	11
11	11	11	11

block 2

Merging

Merging technique depends on I/O pattern.

Three merge types:

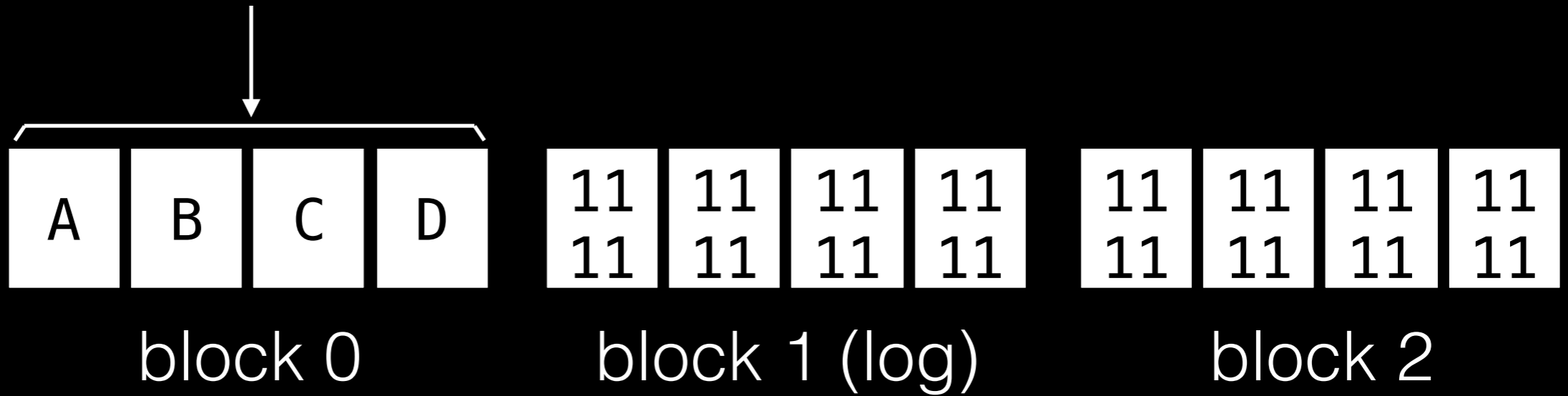
- full merge
- partial merge
- switch merge

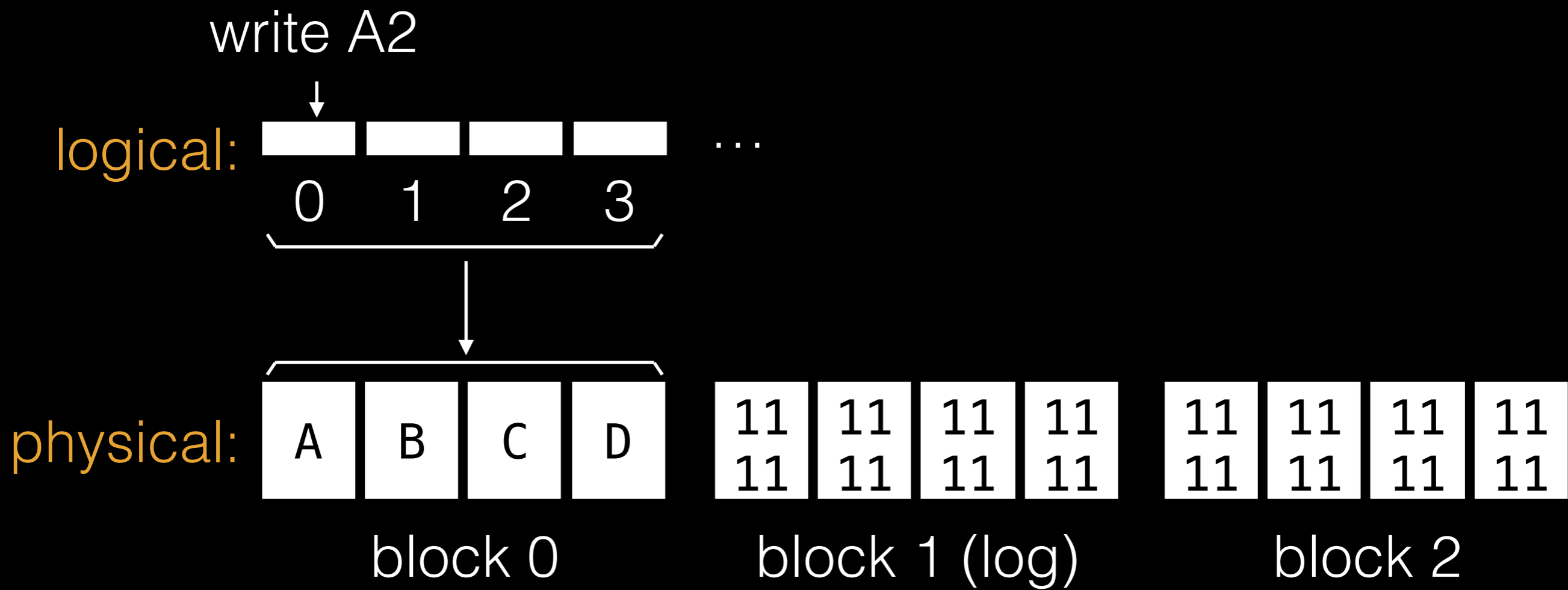
logical:

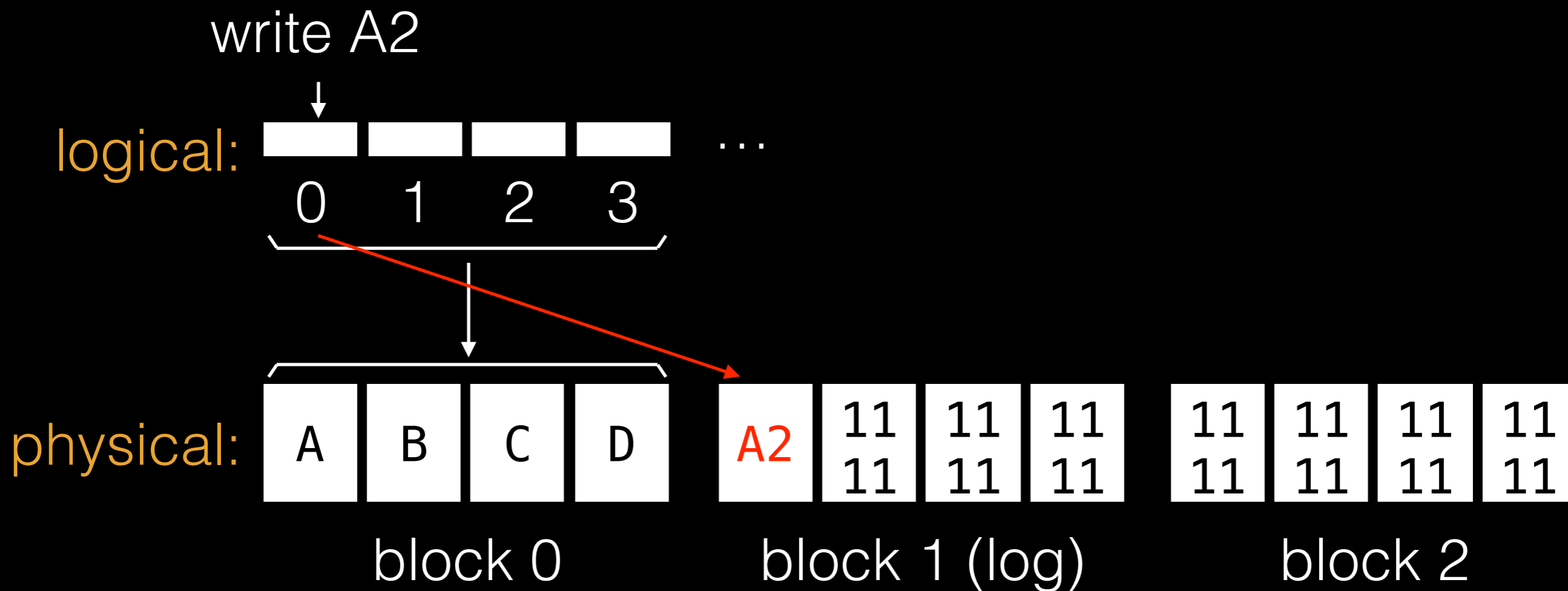
0	1	2	3

 ...

physical:



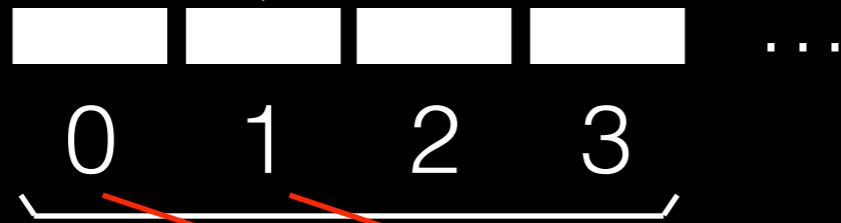




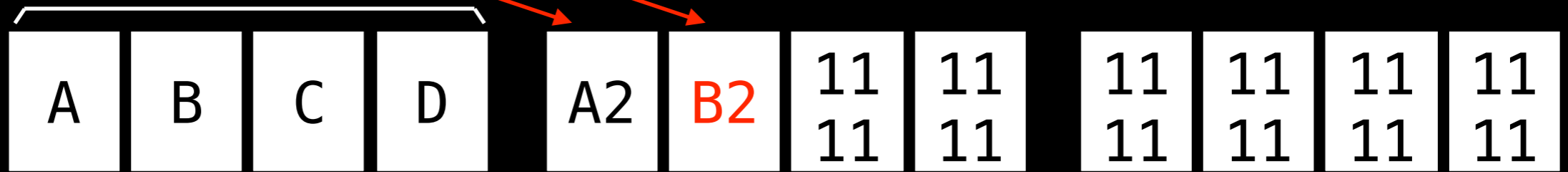
write B2



logical:



physical:



block 0

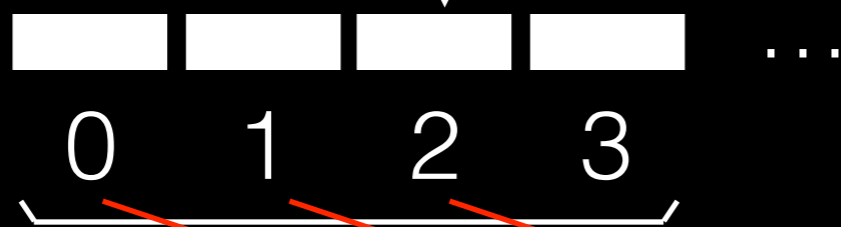
block 1 (log)

block 2

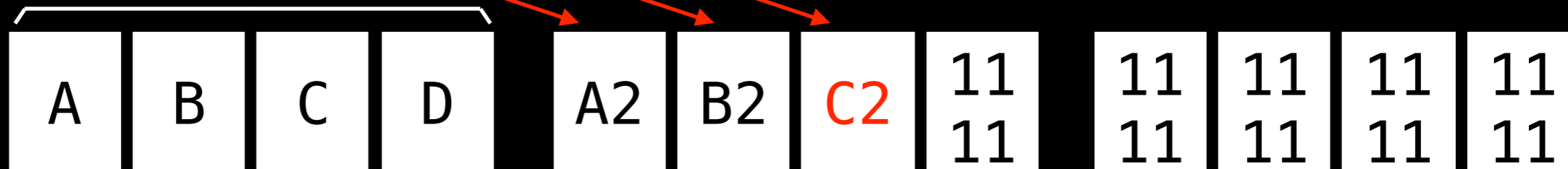
write C2



logical:



physical:



block 0

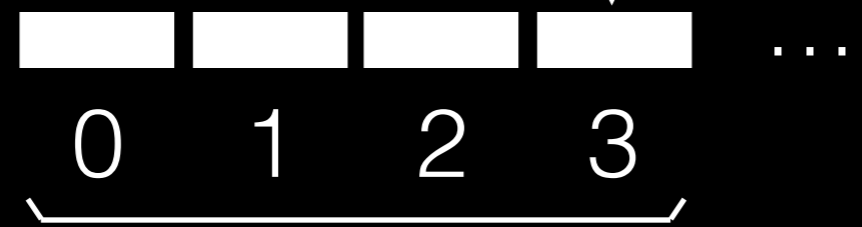
block 1 (log)

block 2

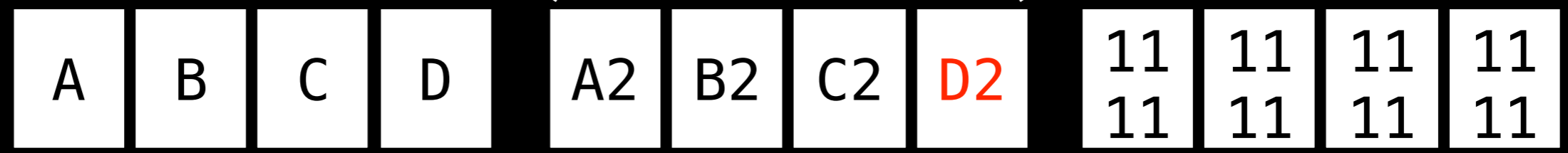
write D2



logical:



physical:



block 0

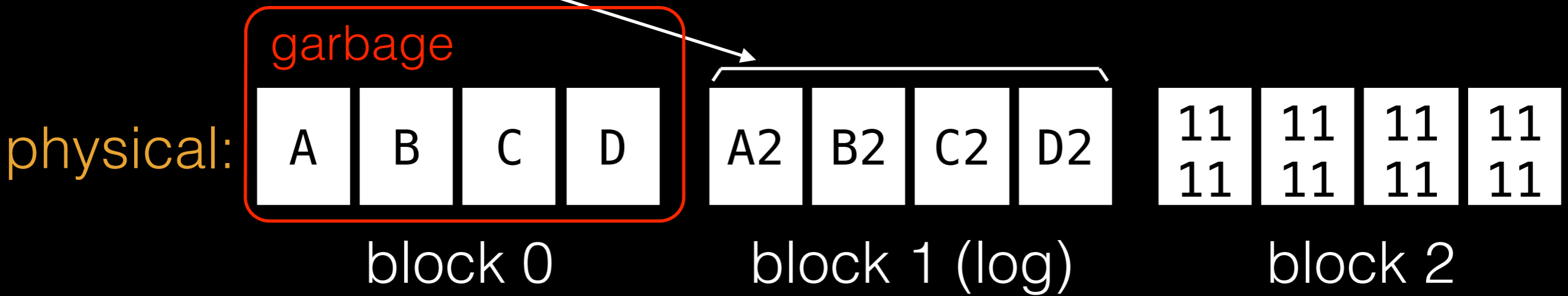
block 1 (log)

block 2

logical:

0	1	2	3

 ...



Merging

Merging technique depends on I/O pattern.

Three merge types:

- full merge
- partial merge
- switch merge

Summary

Flash is much faster than disk, but...

It is more expensive.

It's not a drop-in replacement beneath an FS without a complex layer for emulating hard disk API.

Announcements

Office hours after class in lab.

p5a and p5b due Fri.