[537] Final Review

Tyler Harter
12/14/14

Chapters 4+5: Processes

How do we share?

CPU?

Memory?

Disk?

How do we share?

CPU? (a: time sharing)

Memory? (a: space sharing)

Disk? (a: space sharing)

How do we share?

CPU? (a: time sharing) TODAY

Memory? (a: space sharing)

Disk? (a: space sharing)

Goal: processes should NOT even know they are sharing (each process will get its own virtual CPU)

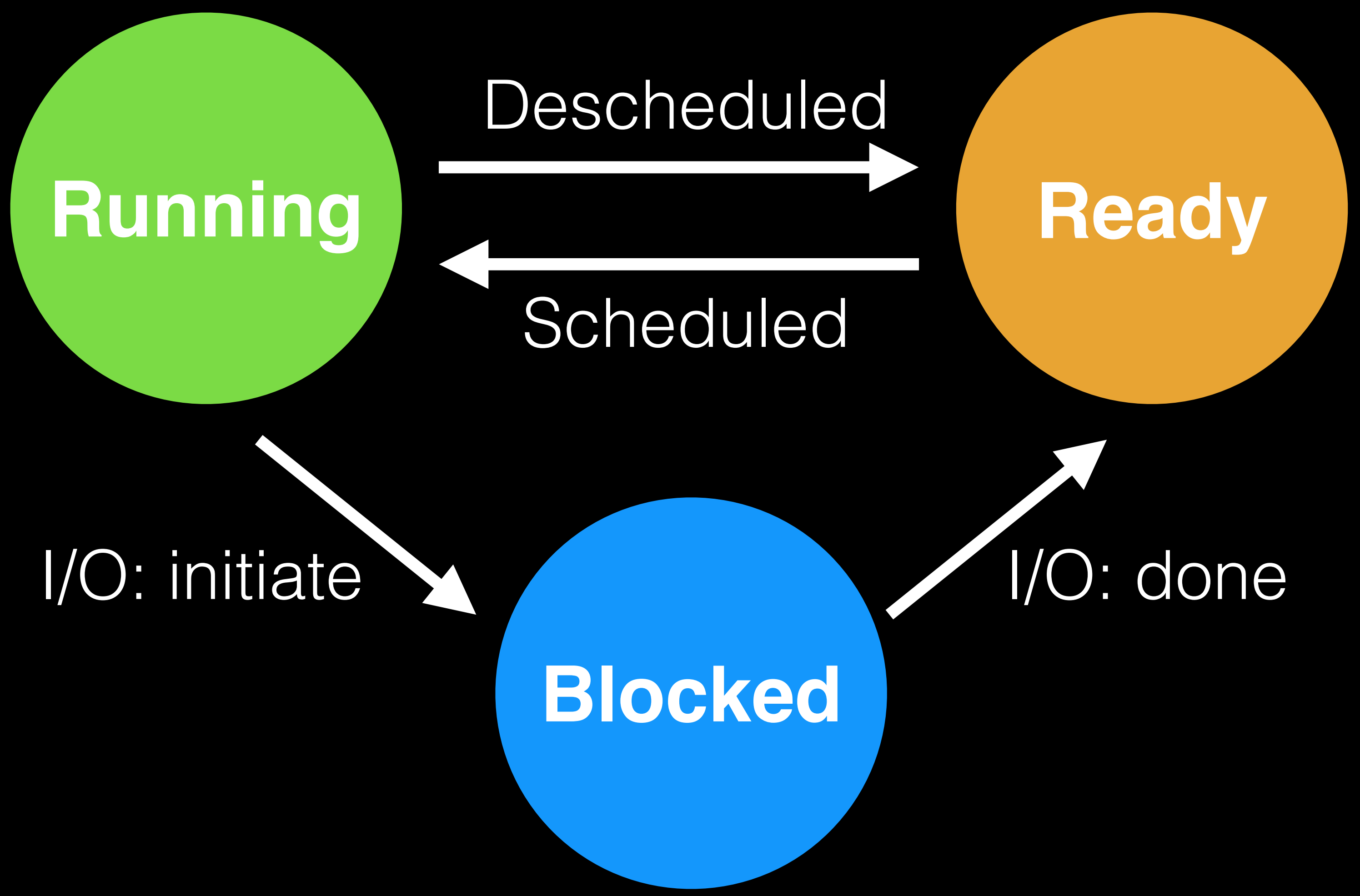
What to Do with Processes That Are Not Running?

A: store context in OS struct

Look in kernel/proc.h

- `context` (CPU registers)
- `ofile` (file descriptors)
- `state` (sleeping, running, etc)

State Transitions



Chapters 6: LDE

CPU Time Sharing

Goal 1: efficiency

OS should have minimal overheard

Goal 2: control

Processes shouldn't do anything bad

OS should decide when processes run

Solution: limited direct execution

What to limit?

General memory access

Disk I/O

Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”

What to limit?

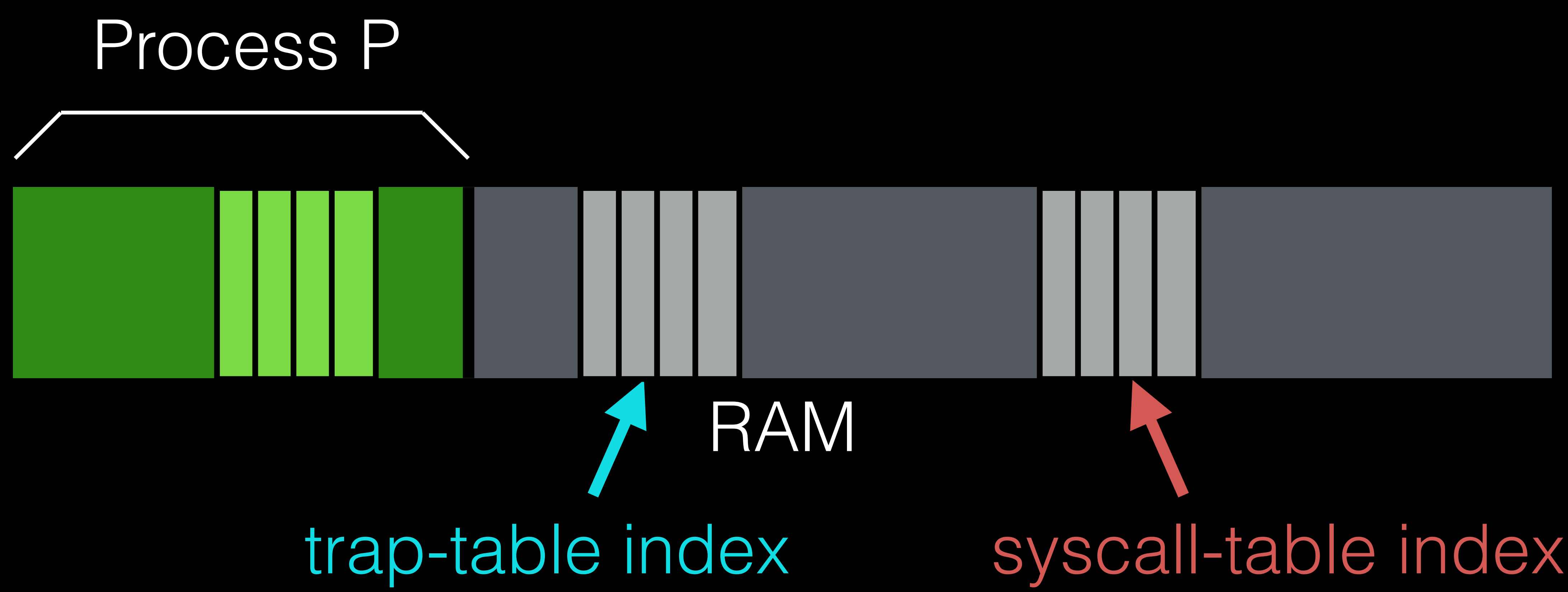
General memory access

Disk I/O

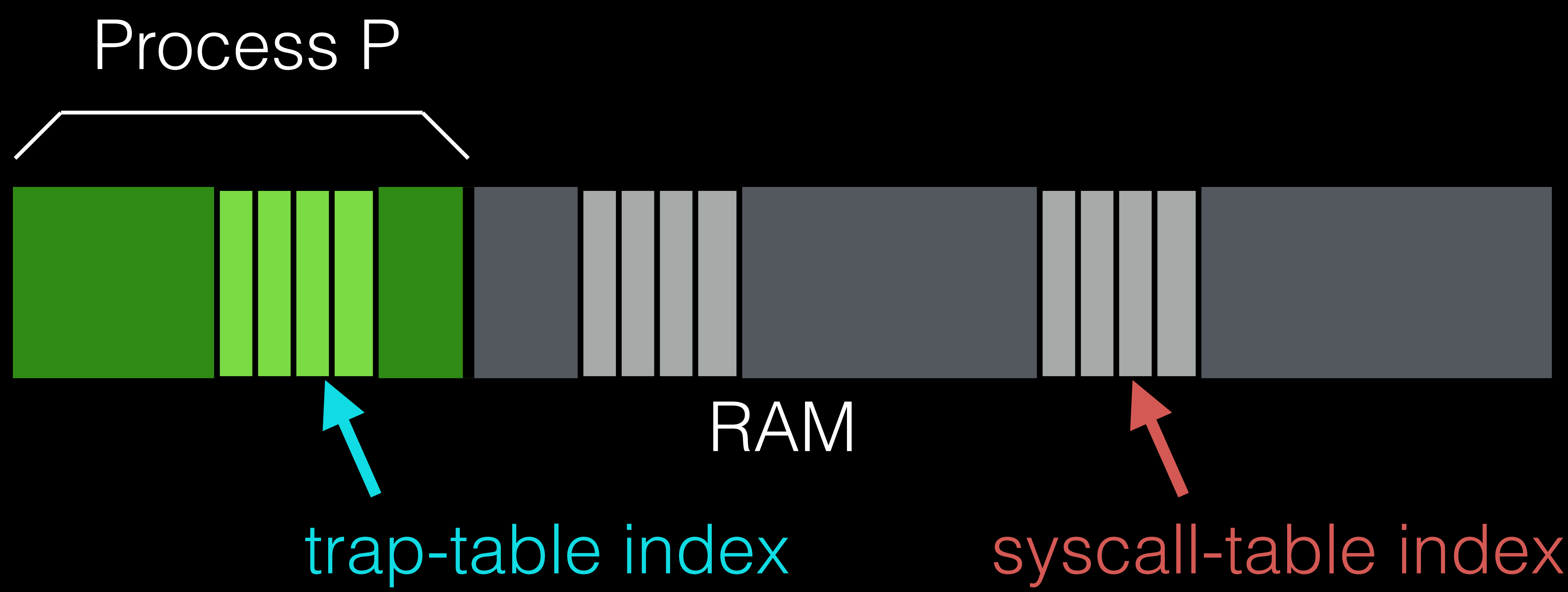
Special x86 instructions like `lidt`

How? Get HW help, put processes in “user mode”

`lidt` example

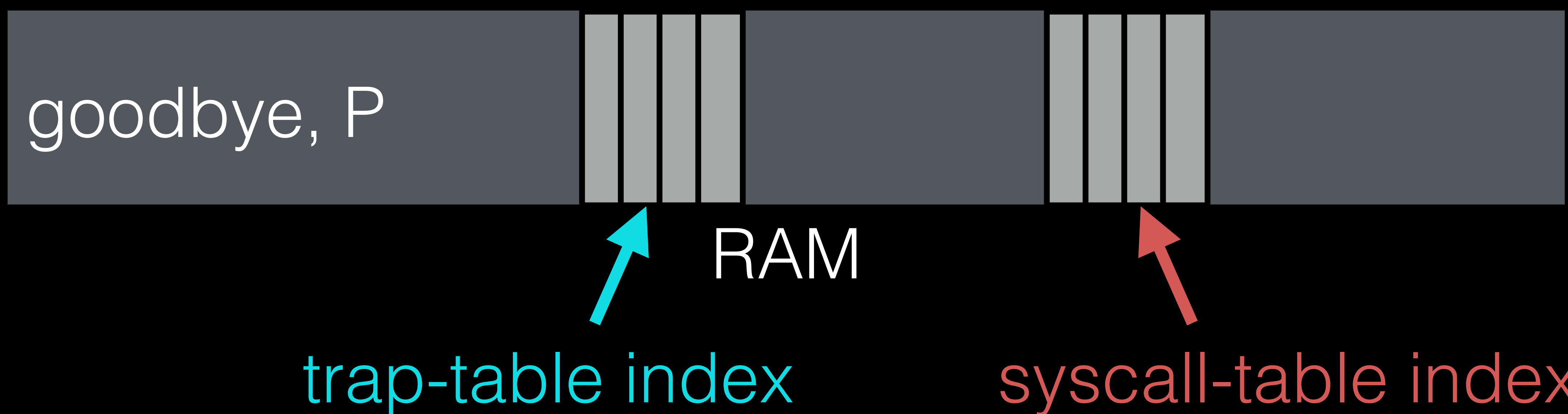


`lidt` example



P tries to call `lidt`!

`lidt` example



CPU warns OS, OS kills P

Context Switch

Problem: when to switch process contexts?

Direct execution => OS can't run while process runs

How can the OS do anything while it's not running?
A: it can't

Solution: **switch on interrupts**. But which interrupt?

Chapters 7: Scheduling

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workloads

Arrival: time at which scheduler is aware of job

Run time: how long does it take if run beginning to end?

Schedulers

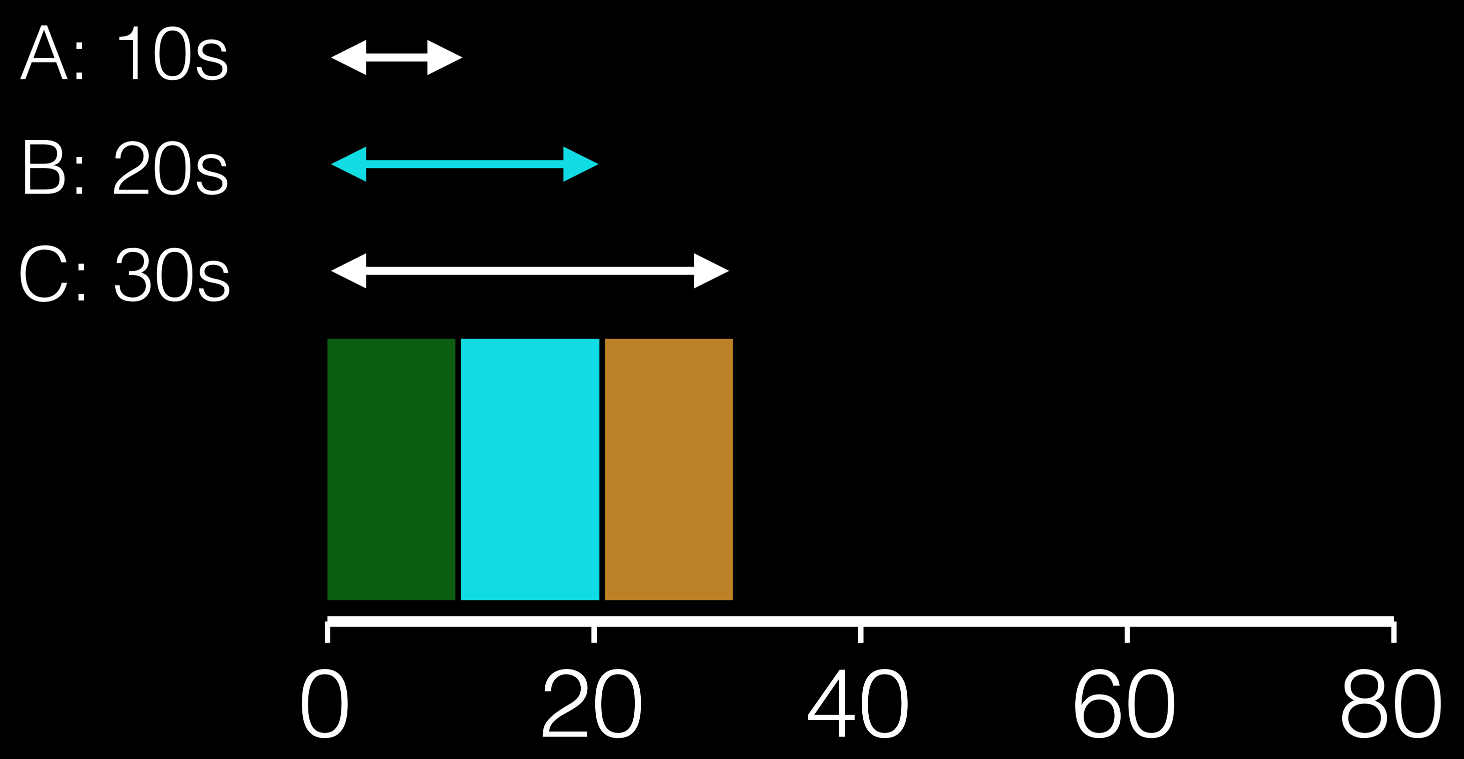
FIFO: first in, first out

SJF: shortest job first (not preemptive)

STCF: shortest time to completion first

RR: round robin

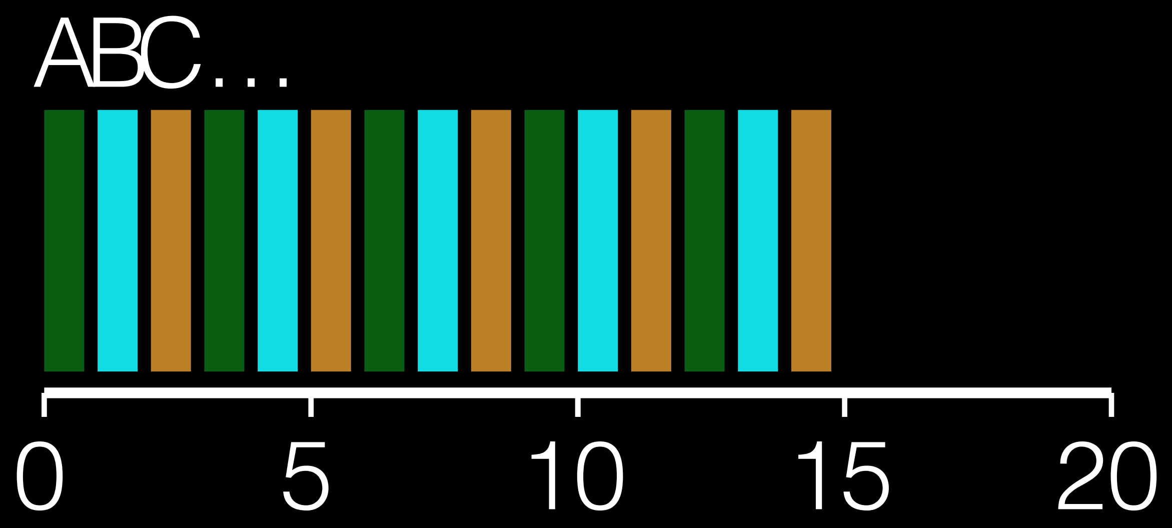
Turnaround Time



What is the average turnaround time? (Q1)

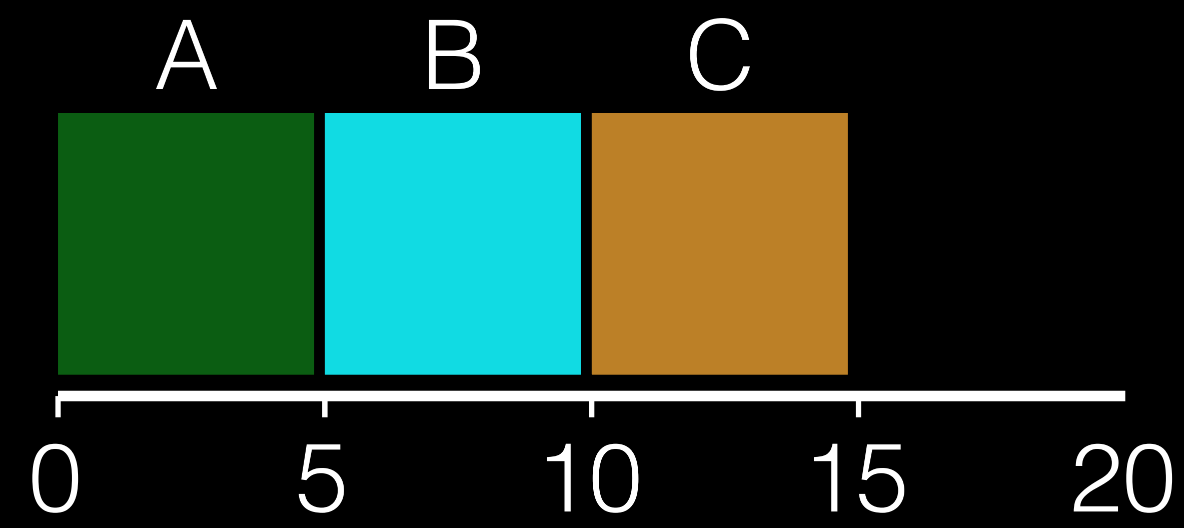
(10 + 20 + 30) / 3 = **20s**

FIFO vs. RR (Q5) — which is each?



Avg Response Time?

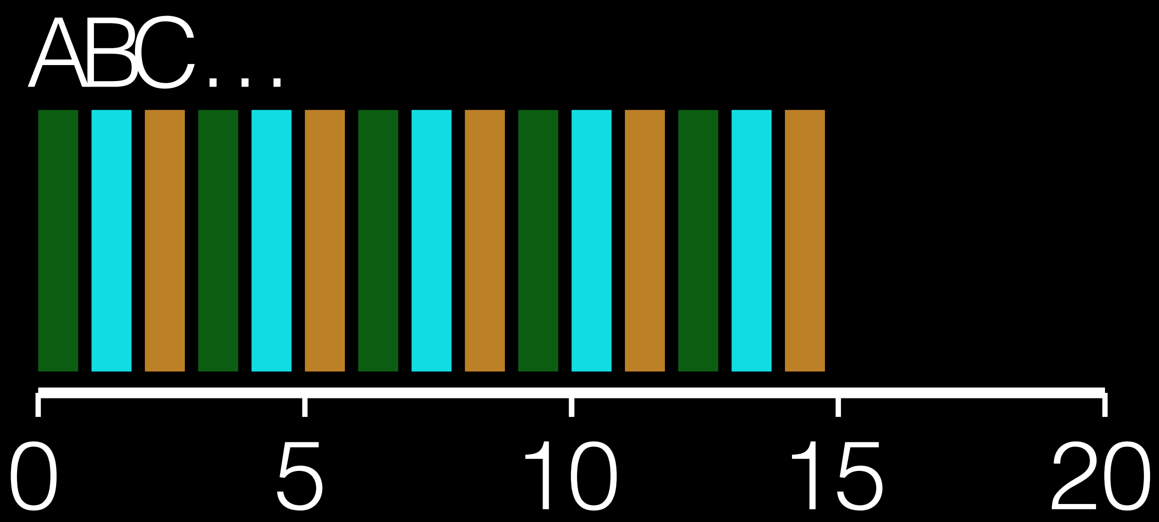
Q5

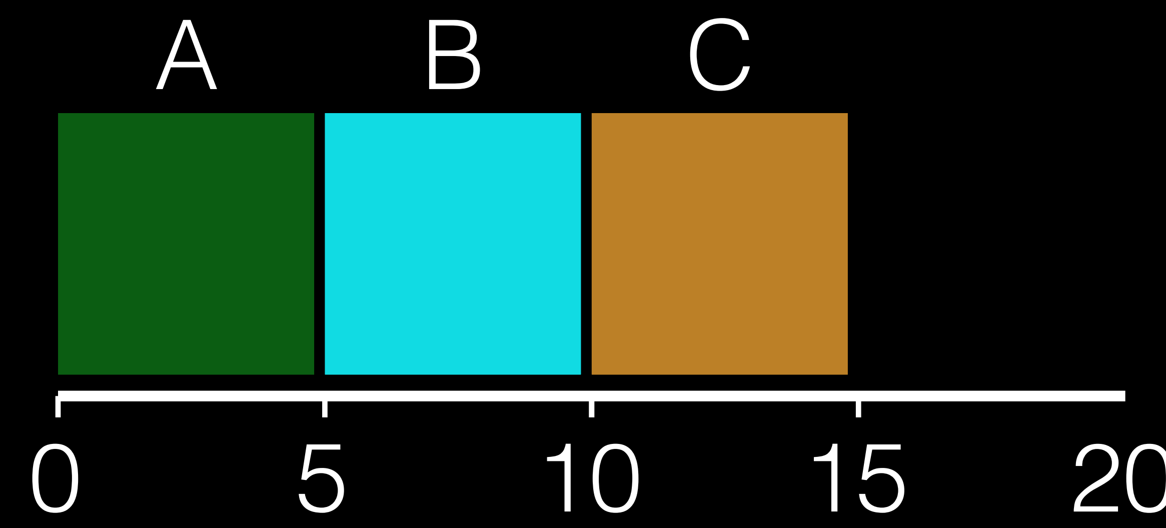


Avg Response Time?

Q5

FIFO vs. RR (Q5) — which is each?





Avg Response Time?
(0+1+2)/3 = **1**

Avg Response Time?
(0+5+10)/3 = **5**

Chapters 16: Segmentation

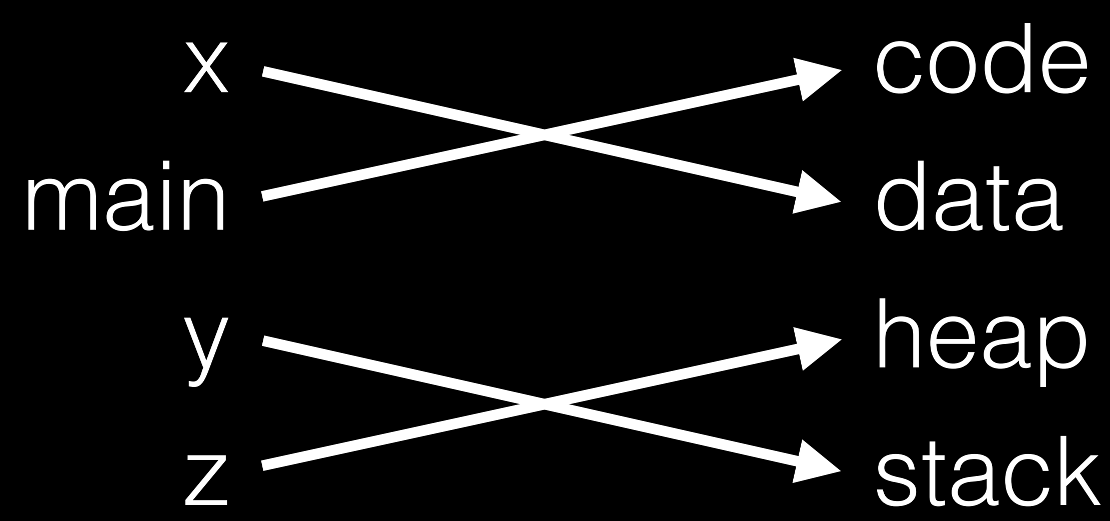
Match that Segment!

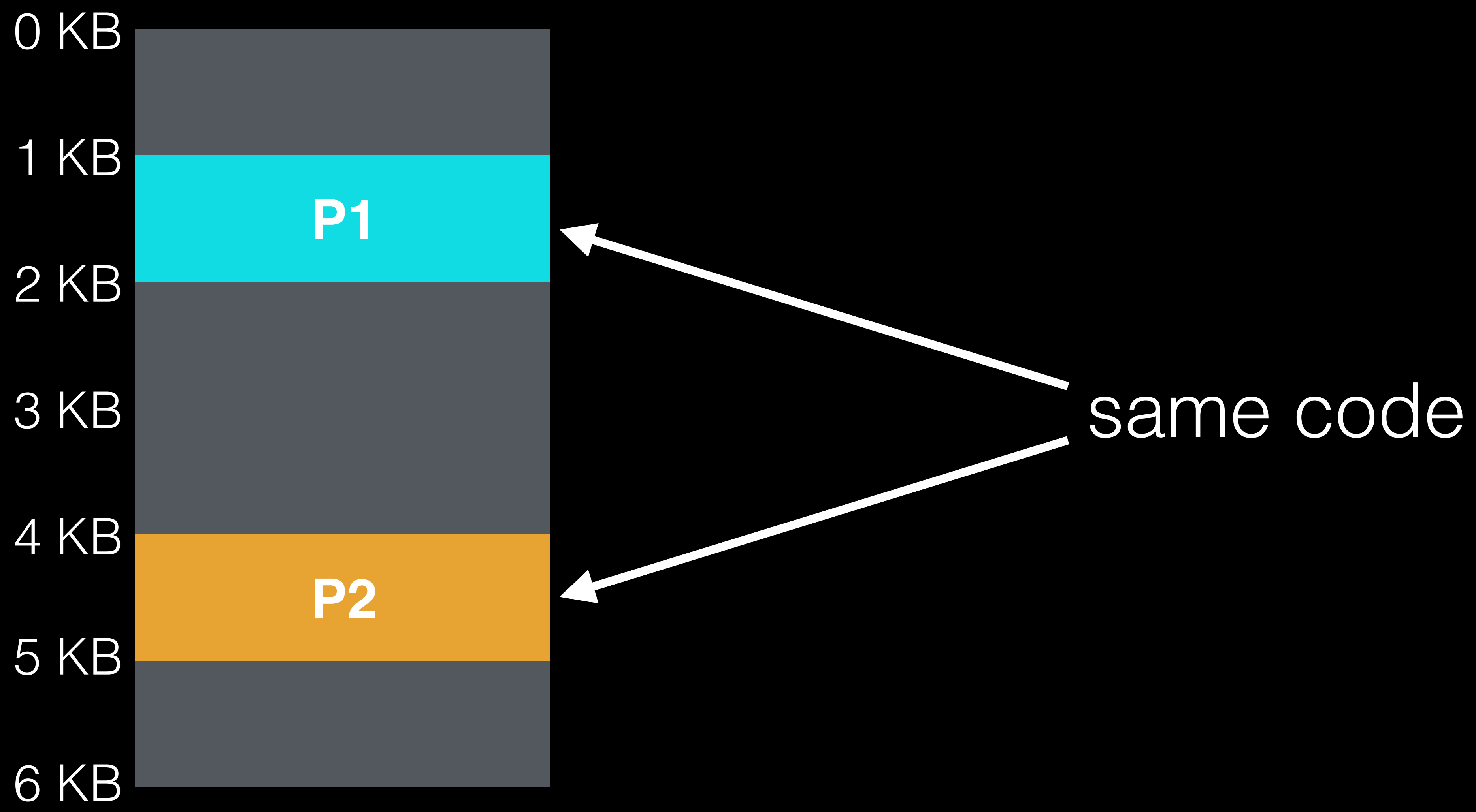
```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof int));
}
```

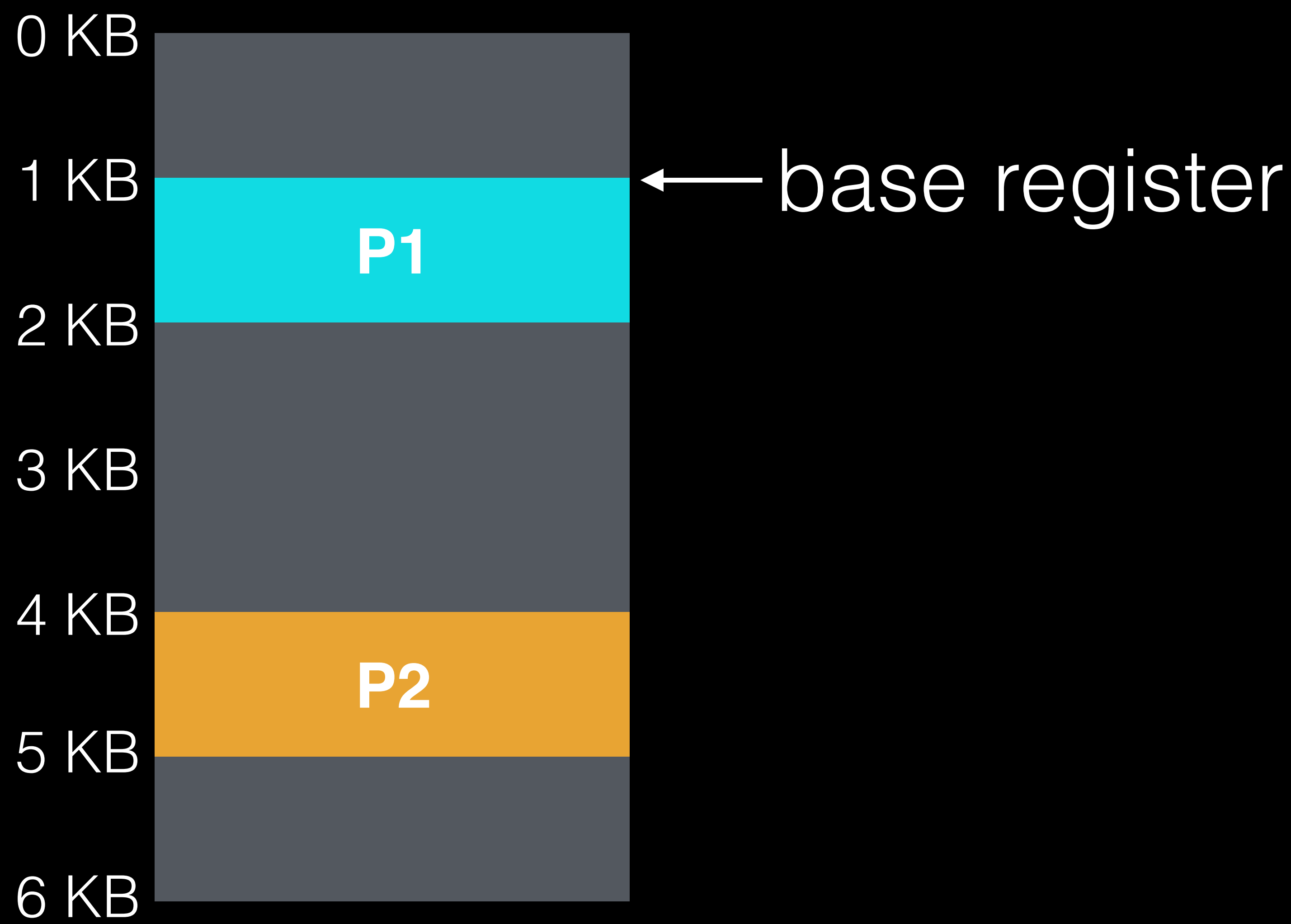
x	code
main	data
y	heap
z	stack

Match that Segment!

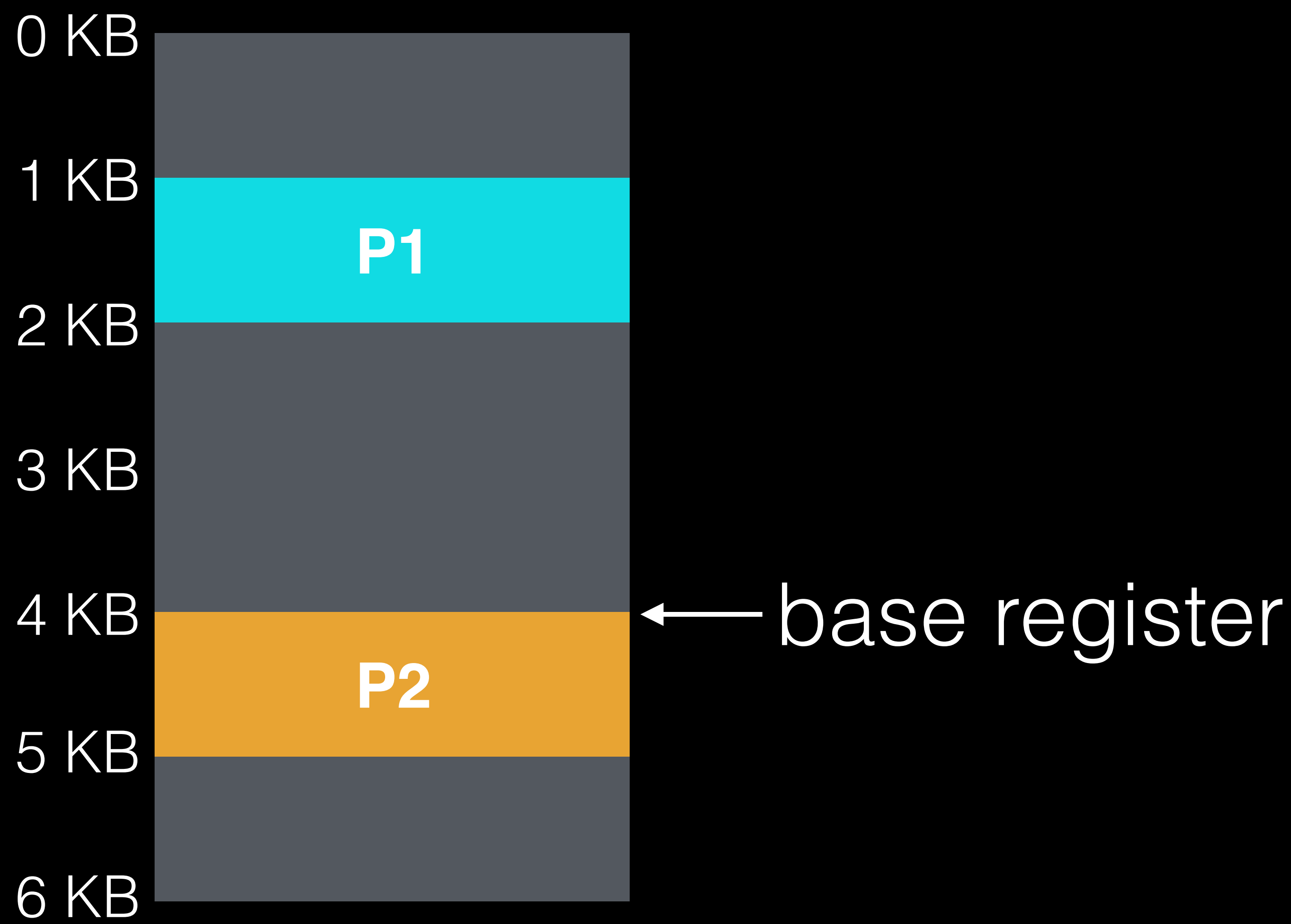
```c
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof int));
}
```



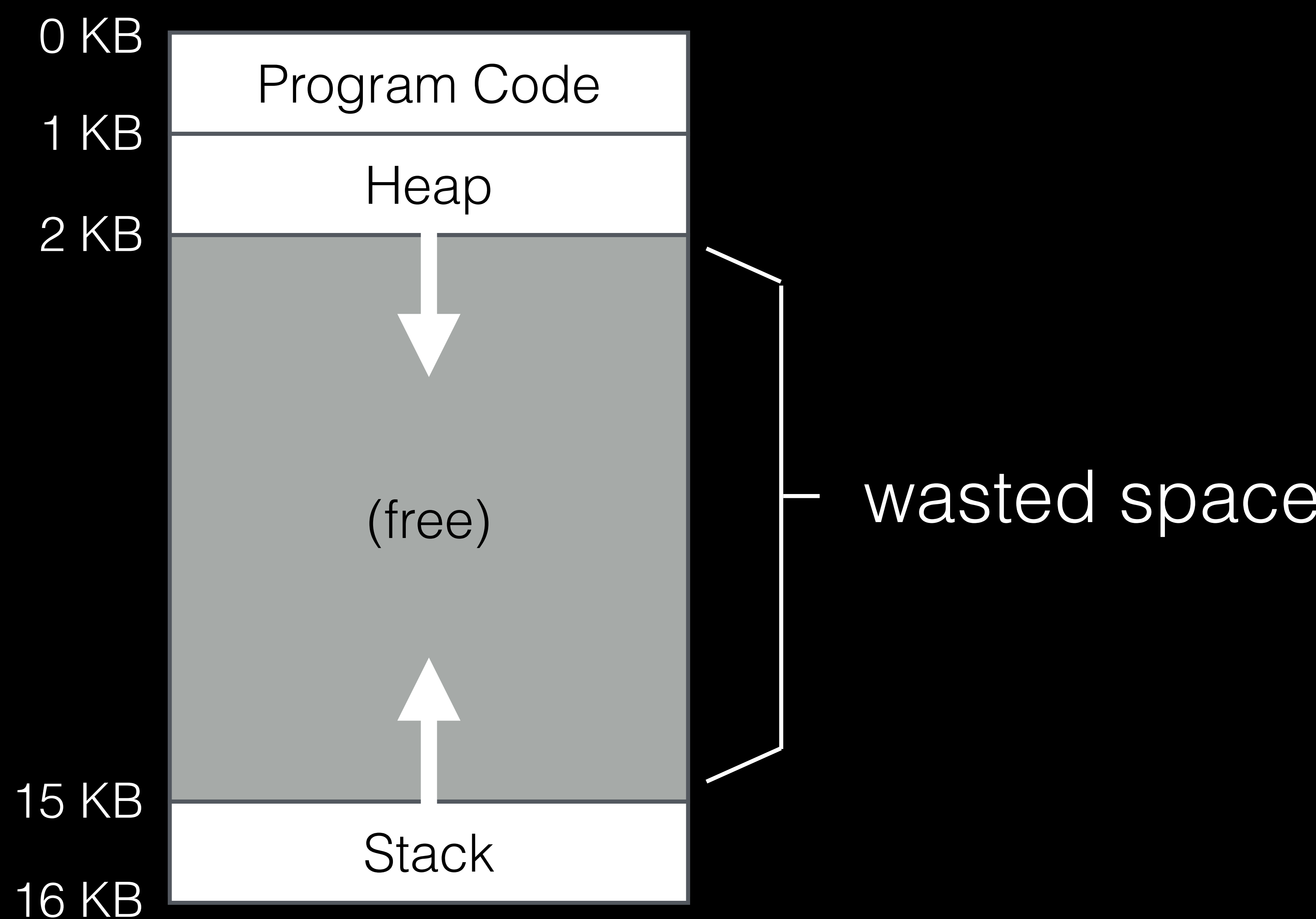

0 KB
1 KB
P1
2 KB
3 KB
4 KB
P2
5 KB
6 KB
same code



P1 is running



P2 is running


0 KB
Program Code
1 KB
Heap
2 KB
(free)
wasted space
15 KB
Stack
16 KB

Multi-segment translation

One (correct) approach:
- break virtual addresses into two parts
- one part indicates segment
- one part indicates offset within segment

Chapters 18: Paging

Paging

Segmentation is too coarse-grained.
Either waste space *OR* memcpy often.

We need a fine-grained alternative!

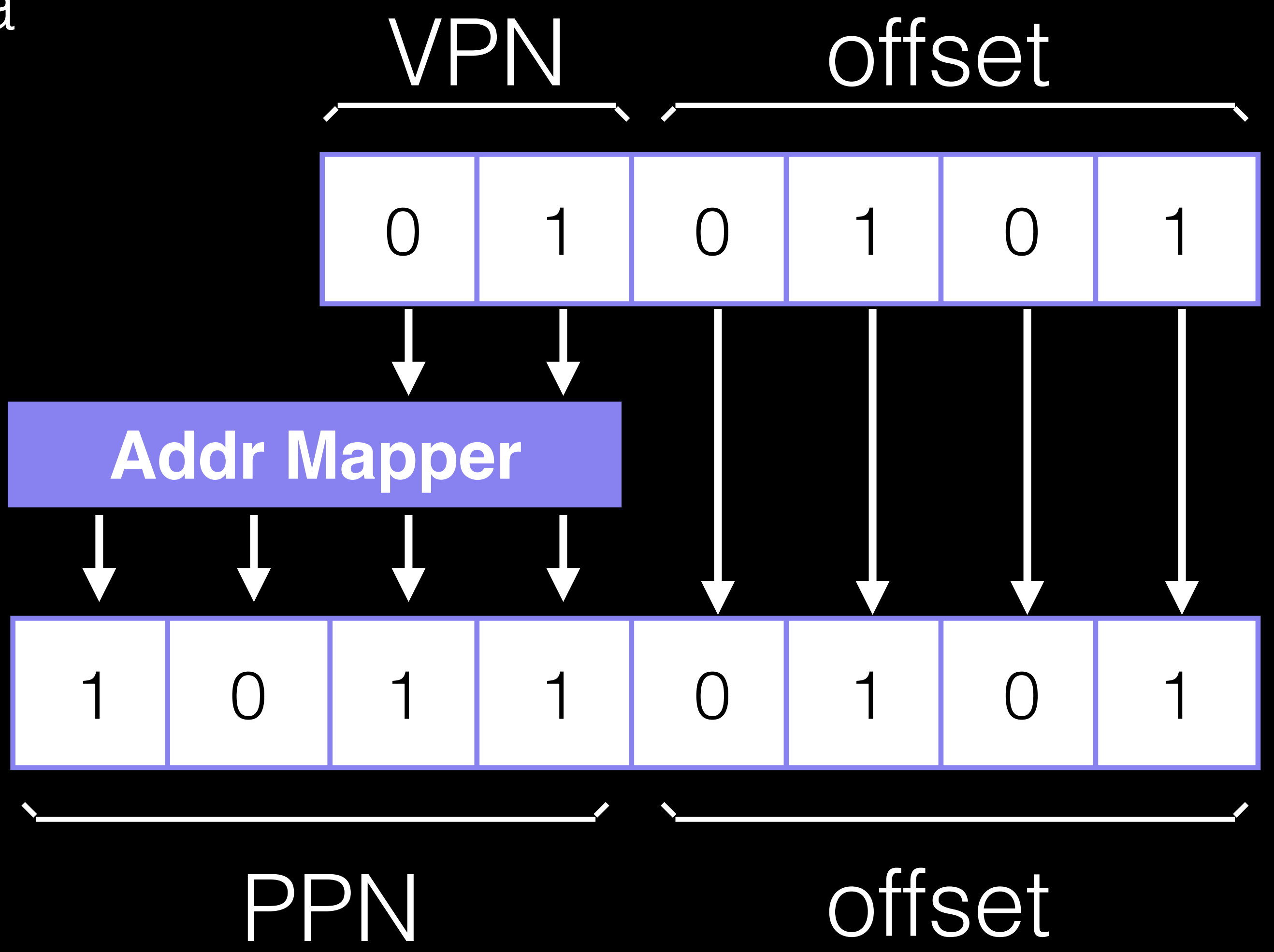
Paging idea:
- break mem into small, fix-sized chunks (aka pages)
- each virt page is independently mapped to a phys page
- grow memory segments however we please!

Virt => Phys Mapping

For segmentation, we used a formula (e.g., phys = virt_offset + base_reg)

Now, we need a more general mapping mechanism.

What data structure is good?
Big array, called a pagetable



Where Are Pagetable's Stored?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries (or this could be less)
- 2 ^ (32 - log(4KB)) * 4 = **4 MB**

Store in memory.
CPU finds it via register (e.g., CR3 on x86)

Other PT info

What other data should go in pagetable entries besides translation?

- valid bit
- protection bits
- present bit
- reference bit
- dirty bit

Chapters 19: TLBs

Translation Steps

H/W: for each mem reference:

(cheap) 1. extract **VPN** (virt page num) from **VA** (virt addr)
(cheap) 2. calculate addr of **PTE** (page table entry)
(expensive) 3. fetch **PTE**
(cheap) 4. extract **PFN** (page frame num)
(cheap) 5. build **PA** (phys addr)
(expensive) 6. fetch **PA** to register

Which expensive step can we avoid?

Array Iterator

```
int sum = 0;
for (i=0; i<N; i++) {
   sum += a[i];
}
```

Array Iterator

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

Array Iterator

Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Array Iterator

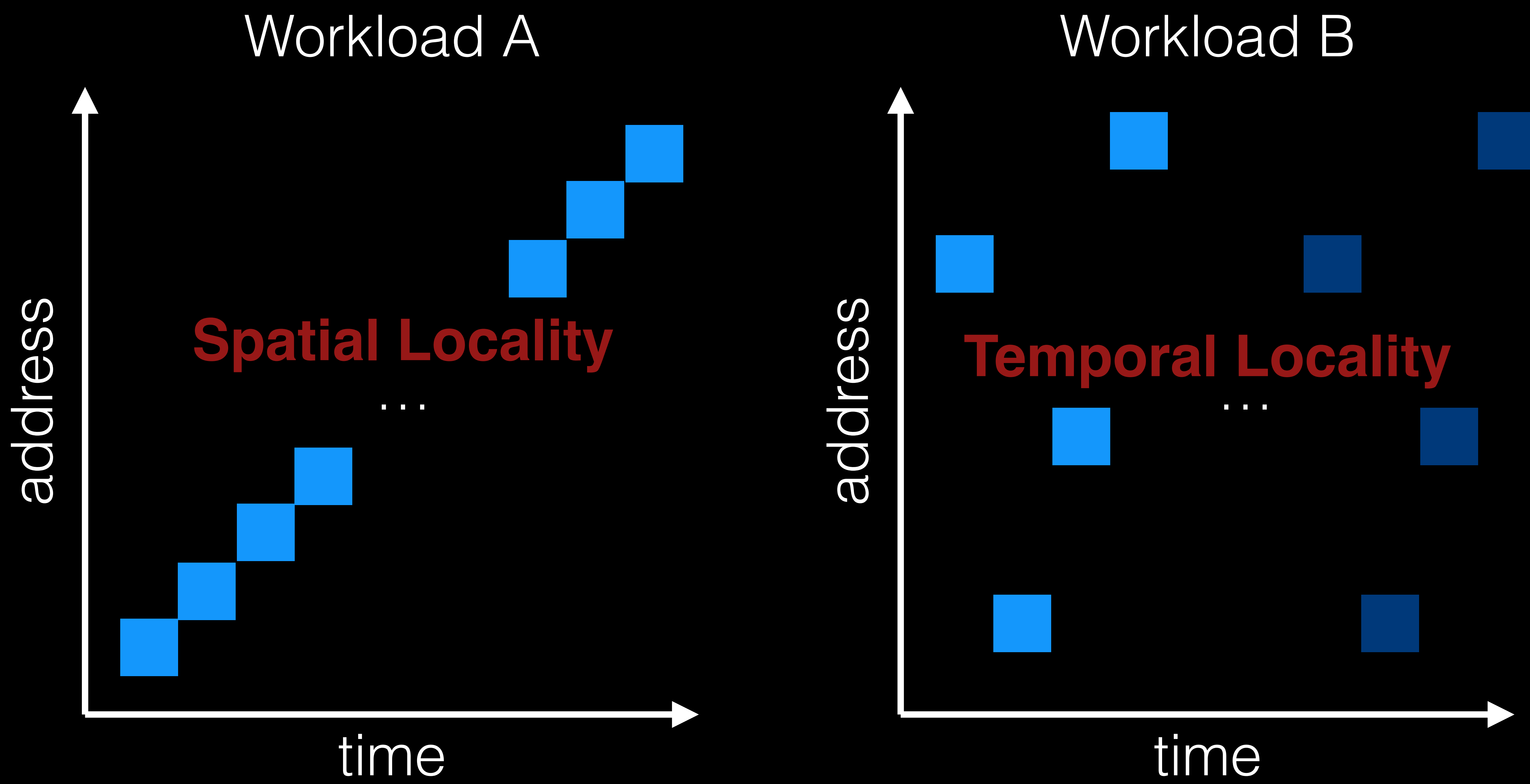
Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Array Iterator

Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Array Iterator

Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C


Workload A
Workload B
address
Spatial Locality
...
time
address
Temporal Locality
...
time

Address **S**pace **Id**entifier

Tag each TLB entry with an 8-bit ASID
- how many ASIDs to we get?
- why not use PIDs?
- what if there are more PIDs than ASIDs?

Security

Modifying TLB entries is privileged
 - otherwise what could you do?

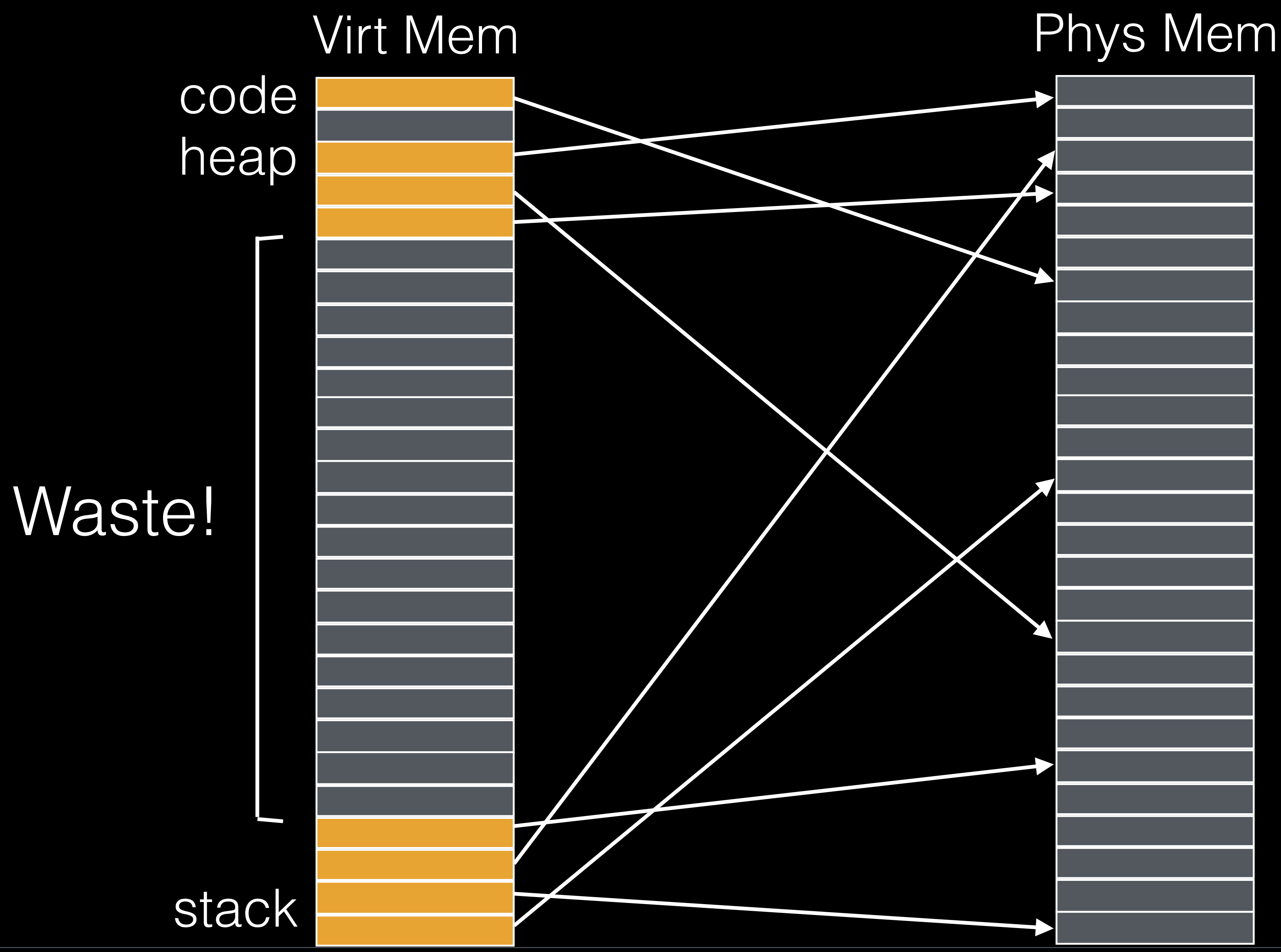
Need same protection bits in TLB as pagetable
 - rwx

Chapters 20: multi-level PTs

Motivation

Why do we want big virtual address spaces?
- programming is easier
- applications need not worry (as much) about fragmentation

Paging goals:
- space efficiency (don't waste on invalid data)
- simplicity (no bookkeeping should require contiguous pages)


Virt Mem
Phys Mem
code
heap
Waste!
stack

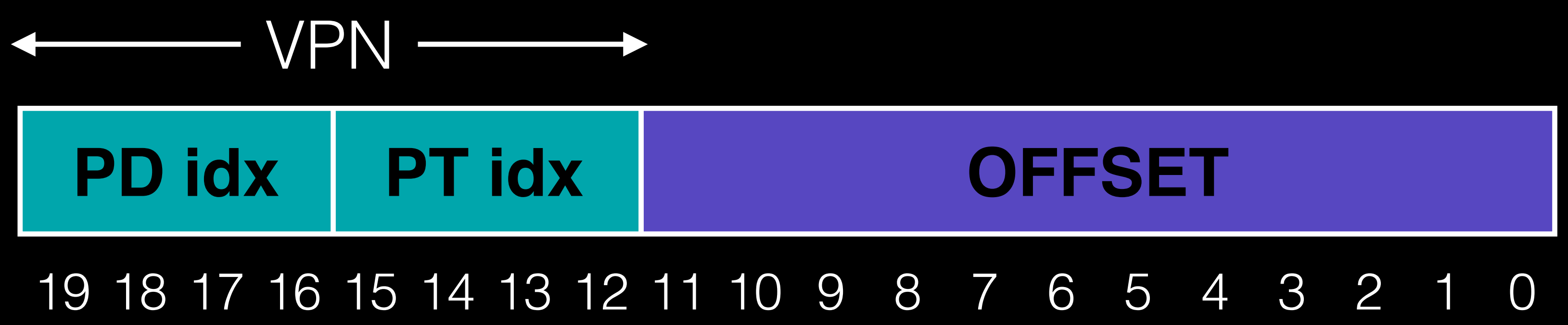
Many invalid PT entries

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

Multi-Level Page Tables

Idea: break PT itself into pages
- a page directory refers to pieces
- only have pieces with >0 valid entries

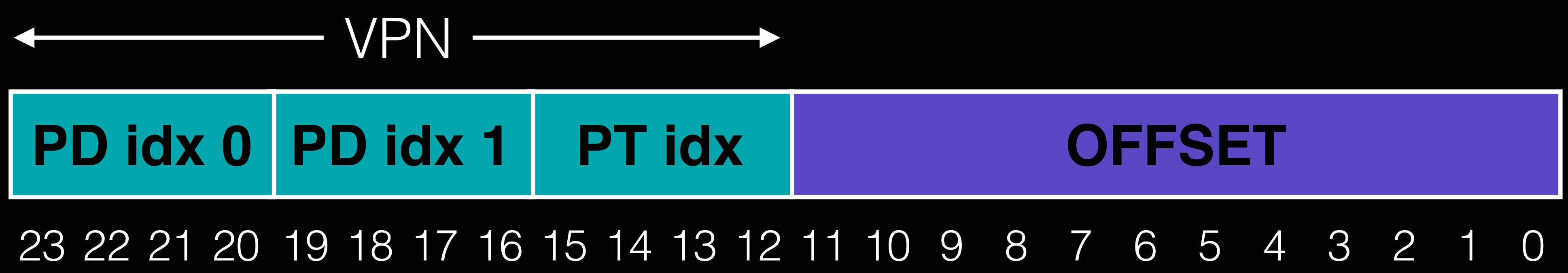
Used by x86.



>2 Levels

Problem: page directories may not fit in a page

Solution: split page directories into pieces.
Use another page dir to refer to the page dir pieces.



Chapters 22: cache policy

Cache

Upon access, we must load the desired page.

Do we **prefetch** other adjacent pages?
(remember disks have high fixed costs)

Prefetching more means we will have to **evict** more.

What to **evict**?

FIFO

Items are evicted in the order they are inserted

Belady's Anomaly

(a) size 3

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1	no	3,4,1
2	no	4,1,2
5	no	1,2,5
1	yes	1,2,5
2	yes	1,2,5
3	no	2,5,3
4	no	5,3,4
5	yes	5,3,4

(b) size 4

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,3,4
1	yes	1,2,3,4
2	yes	1,2,3,4
5	no	2,3,4,5
1	no	3,4,5,1
2	no	4,5,1,2
3	no	5,1,2,3
4	no	1,2,3,4
5	no	2,3,4,5

LRU, MRU

LRU: evict least-recently used
 - consider history

MRU: evict most-recently used

Discuss

Can Belady's anomaly happen with LRU?

Stack property: smaller cache always subset of bigger

LRU Hardware Support

What is needed?

Timestamps. Why can't OS alone track this?

LRU Hardware Support

What is needed?

Timestamps. Why can't OS alone track this?

Cheap approximation: reference (or use) bits.
- set upon access, cleared by OS
- useful for clock algorithm

Thrashing

A machine is **thrashing** when there is not enough RAM, and we constantly swap in/out pages

Solutions?

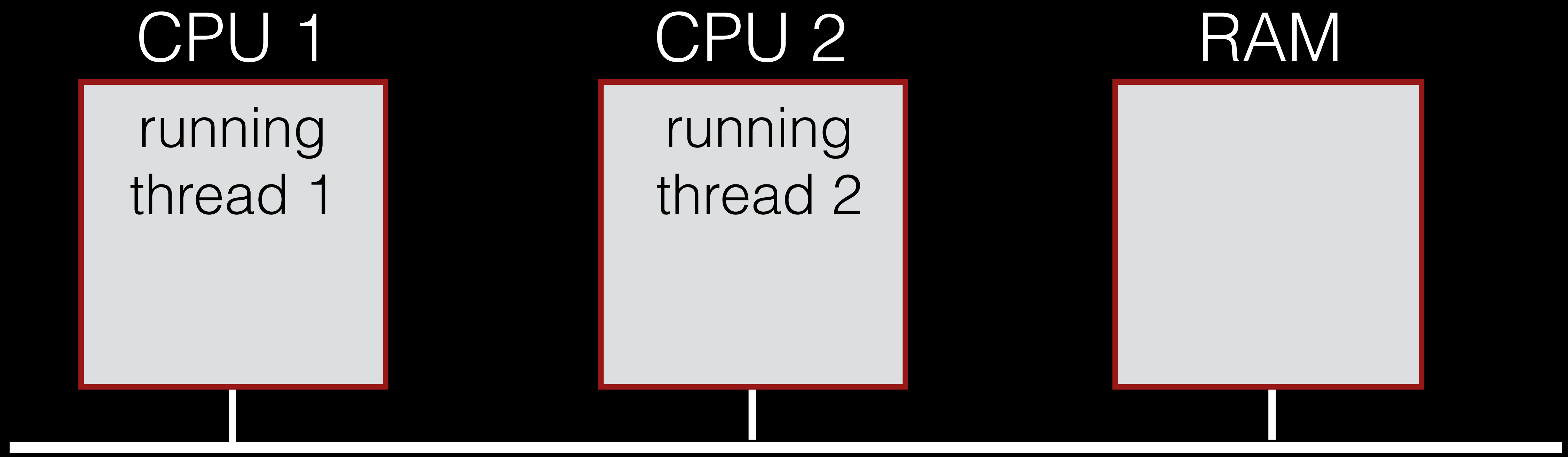
- admission control (like scheduler project)
- buy more memory
- Linux out-of-memory killer!

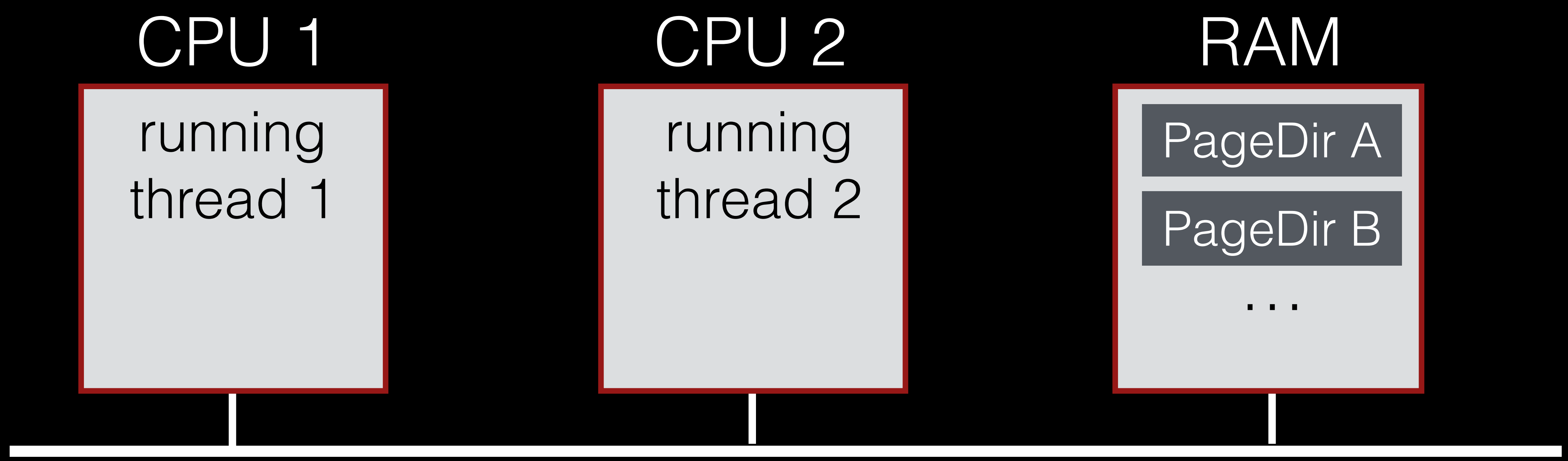
Chapters 26+27: threads

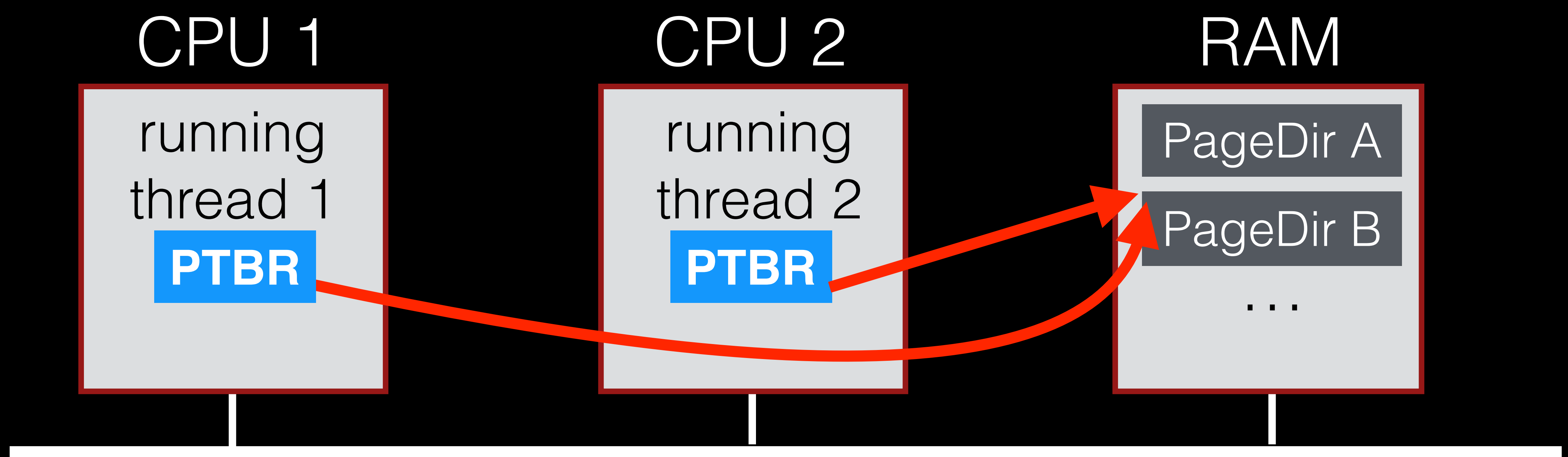
Strategy 2

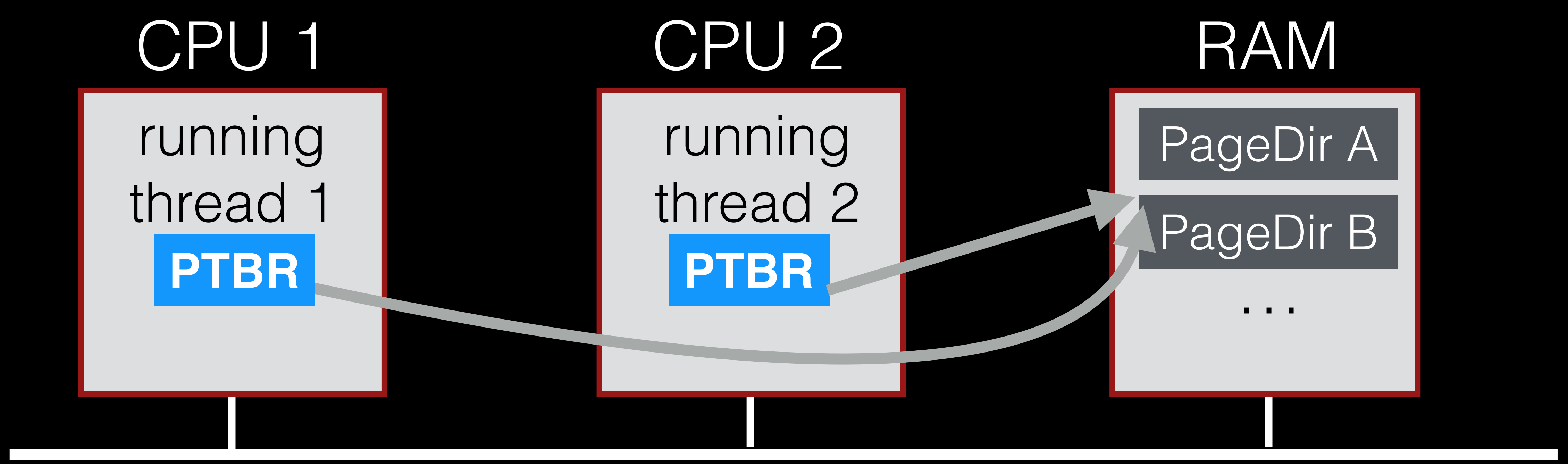
New abstraction: the **thread**.

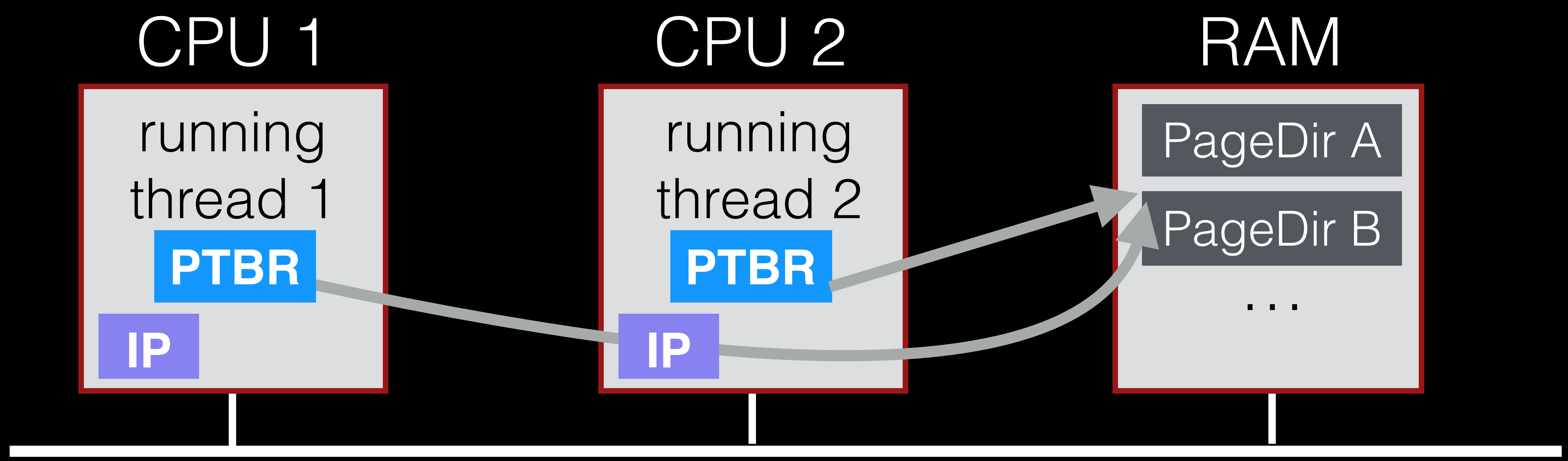
Threads are just like processes, but they share the address space (e.g., using same PT).

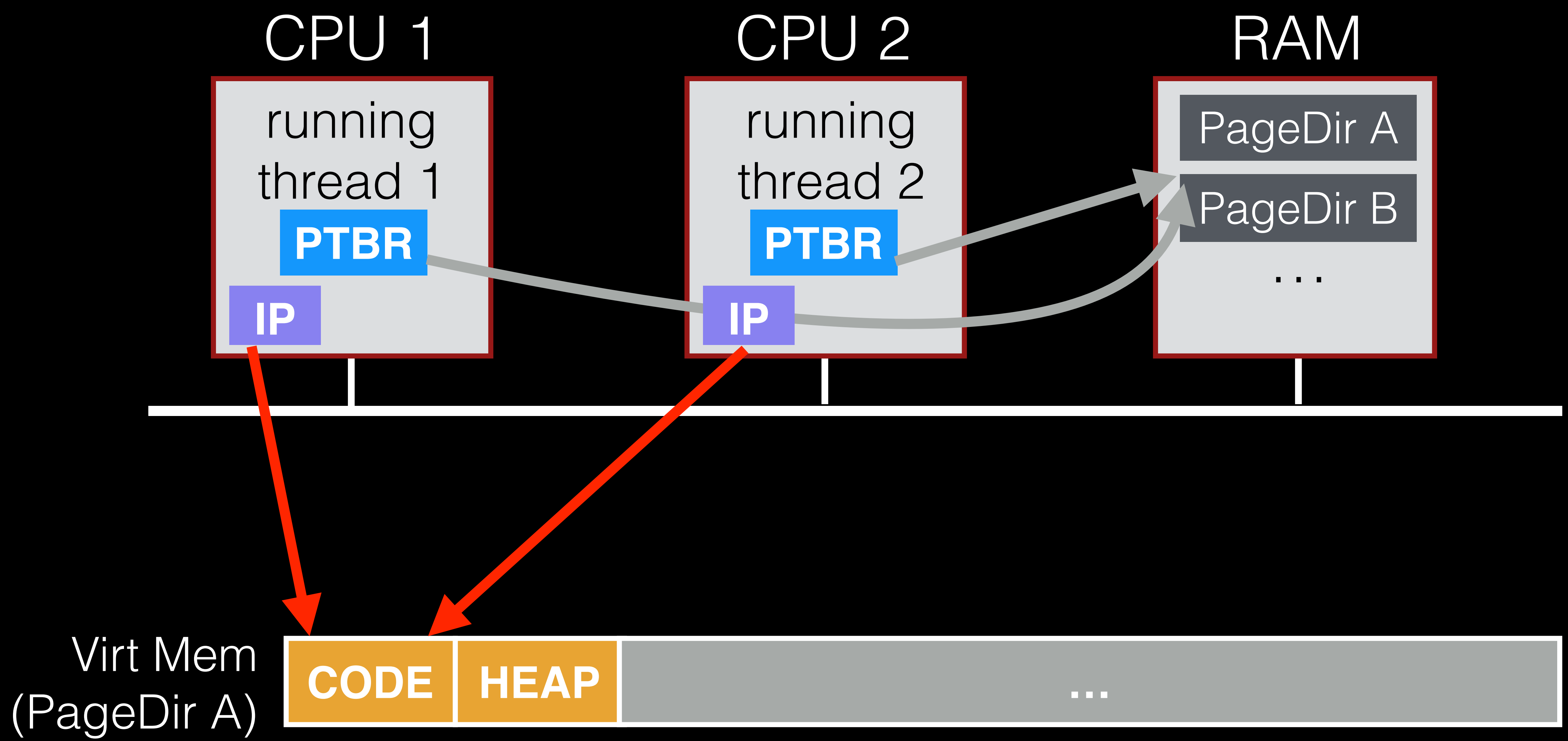

CPU 1
running
thread 1
CPU 2
running
thread 2
RAM

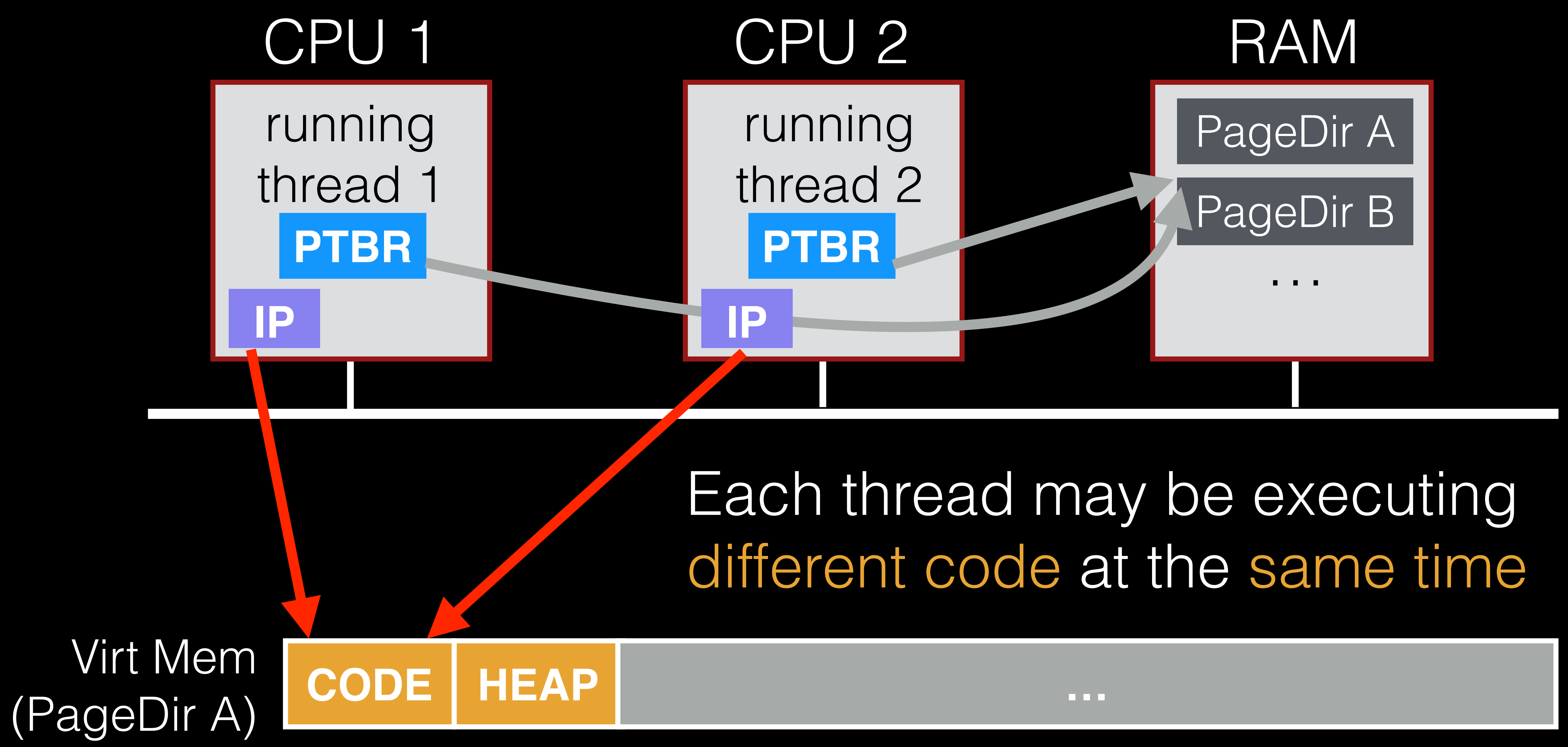

CPU 1
running thread 1
CPU 2
running thread 2
RAM
PageDir A
PageDir B
...


CPU 1
running
thread 1
PTBR
CPU 2
running
thread 2
PTBR
RAM
PageDir A
PageDir B
...

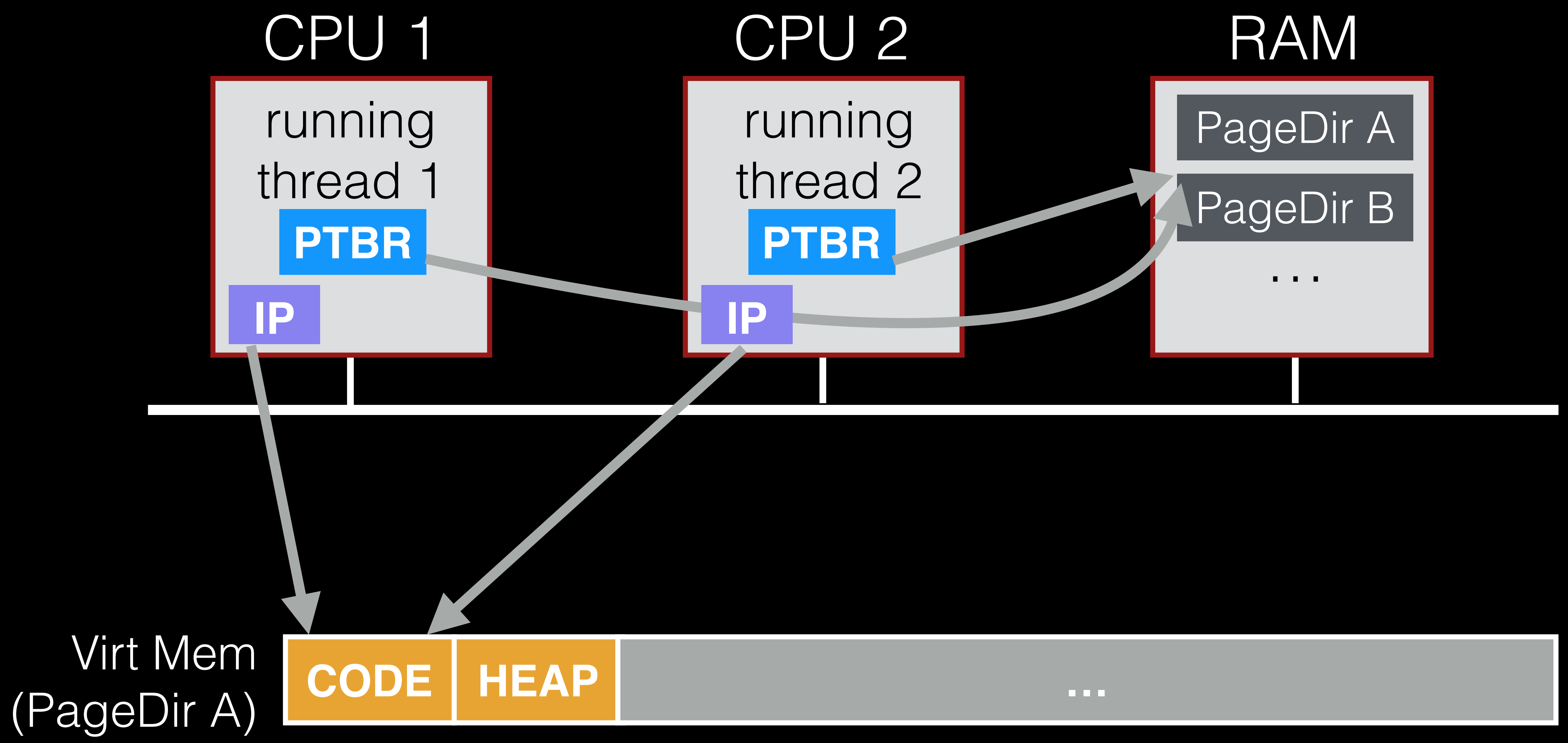

CPU 1
running thread 1
PTBR
CPU 2
running thread 2
PTBR
RAM
PageDir A
PageDir B
...

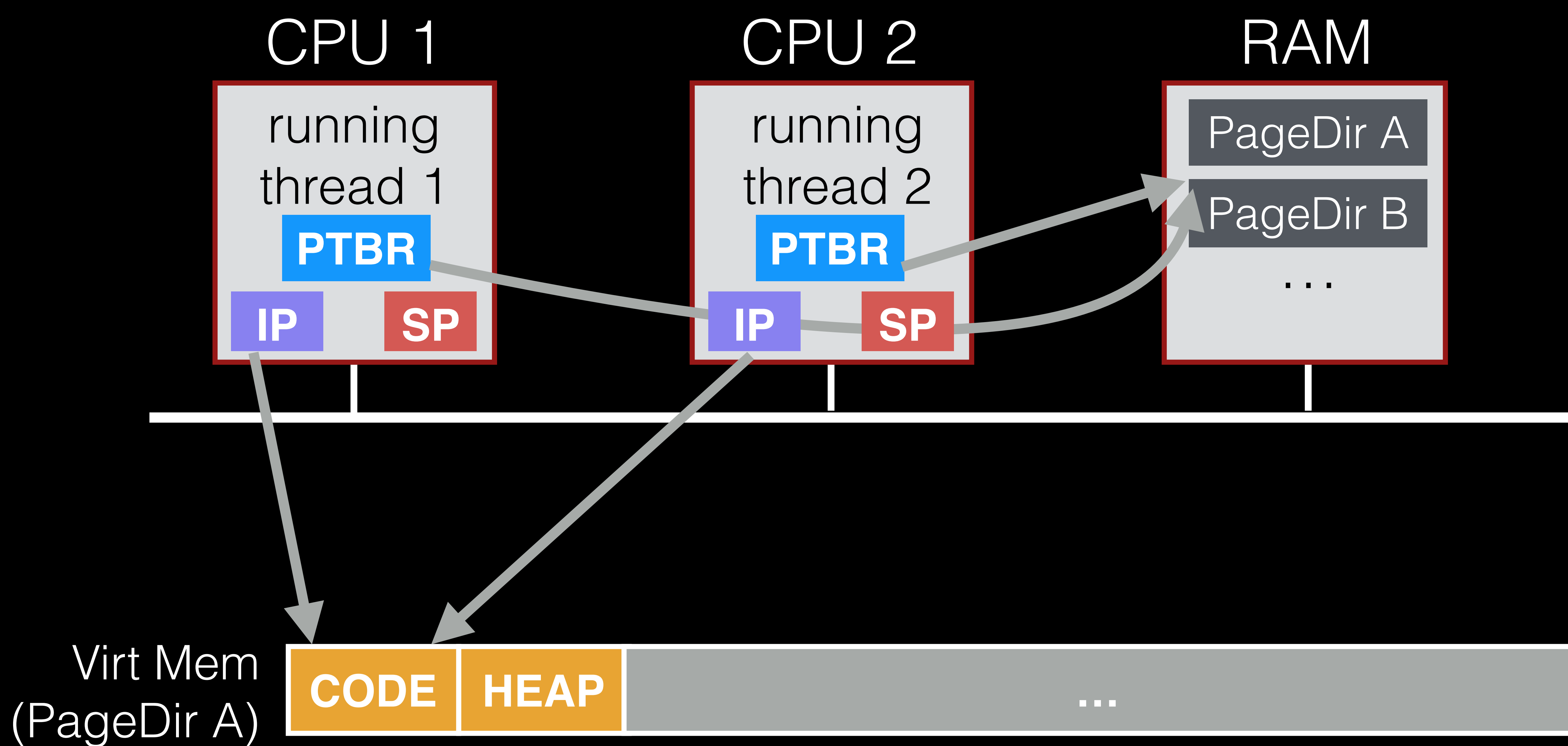

CPU 1
running thread 1
PTBR
IP
CPU 2
running thread 2
PTBR
IP
RAM
PageDir A
PageDir B
...


CPU 1
running thread 1
PTBR
IP
CPU 2
running thread 2
PTBR
IP
RAM
PageDir A
PageDir B
...
Virt Mem
(PageDir A)
CODE
HEAP
...

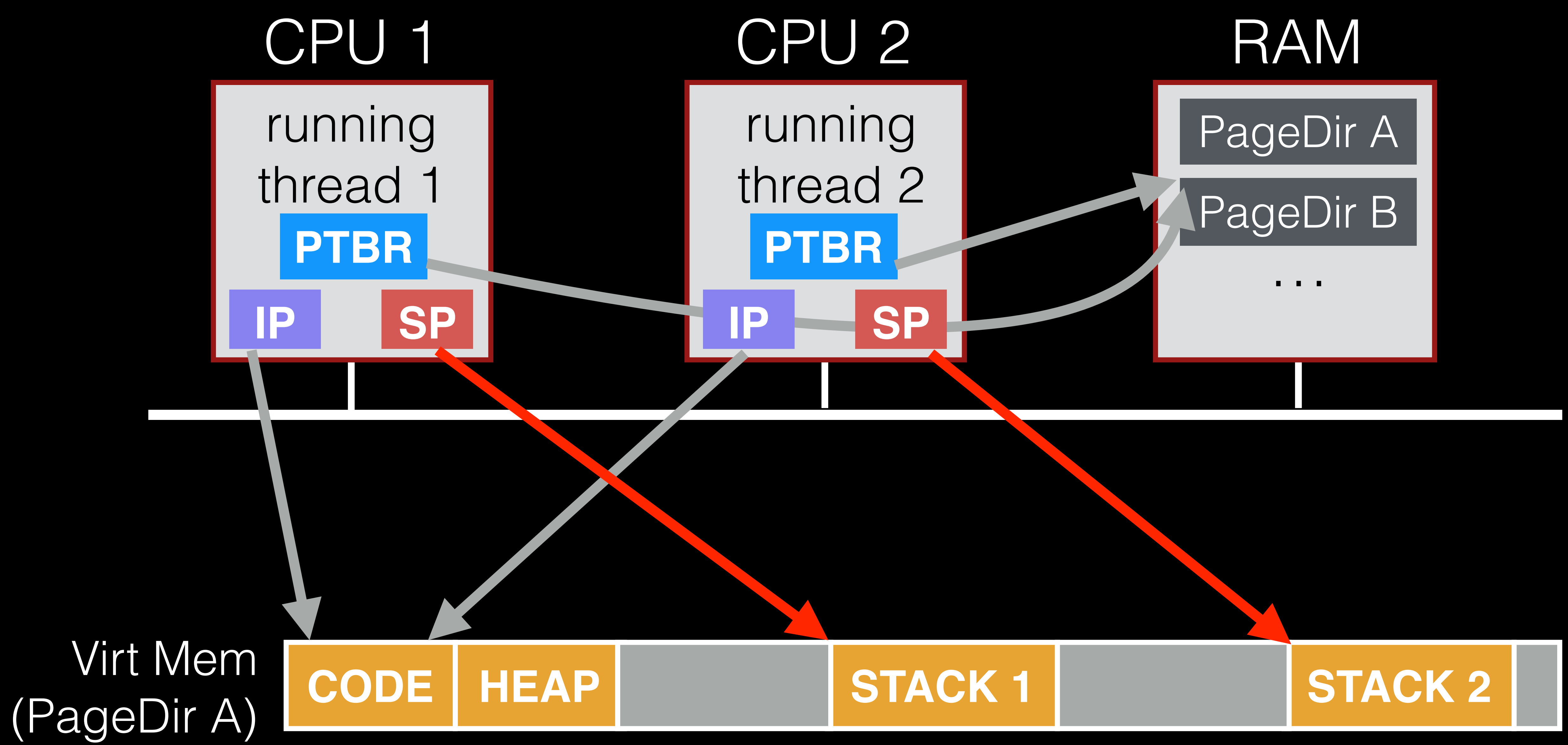


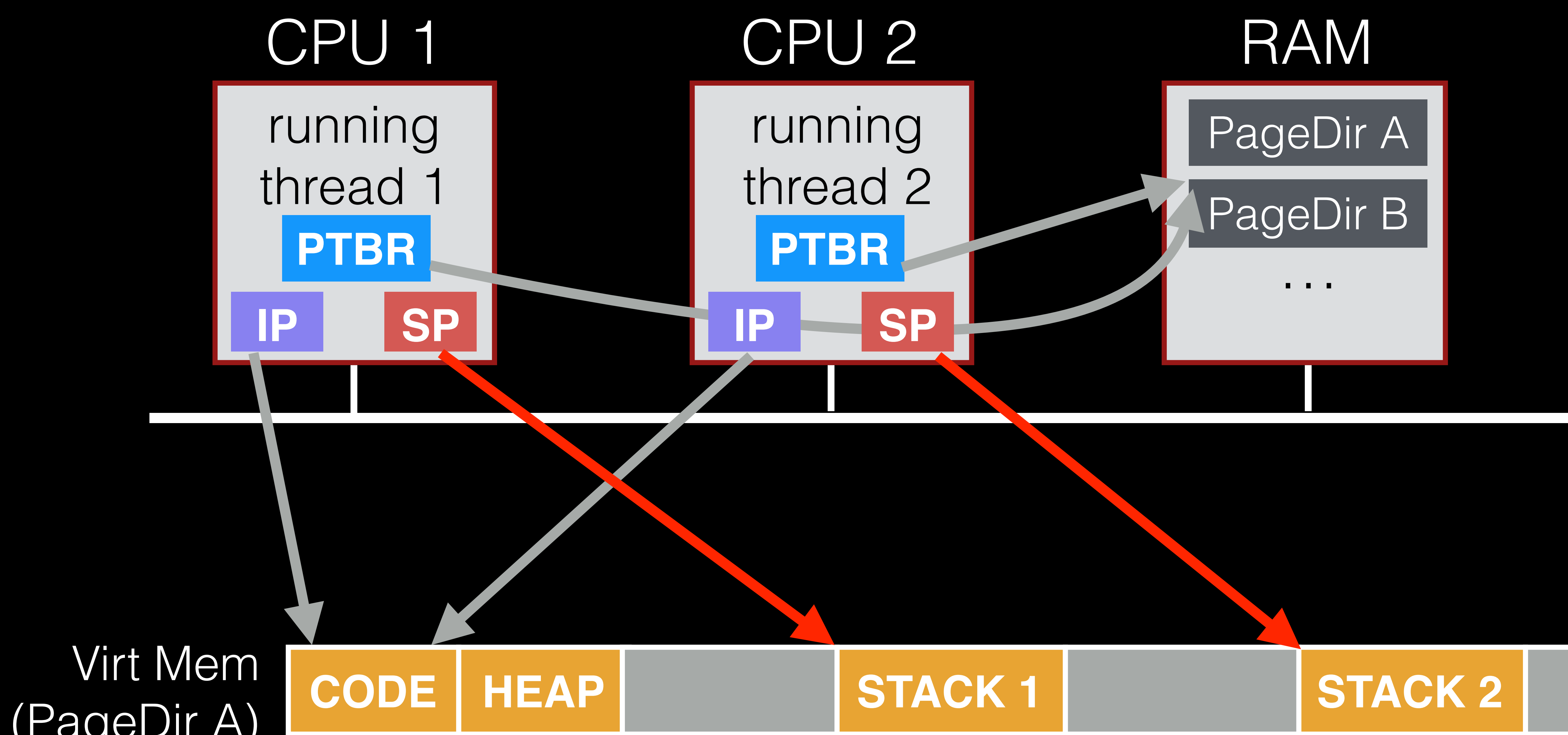
Each thread may be executing different code at the same time


CPU 1
running thread 1
PTBR
IP
CPU 2
running thread 2
PTBR
IP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir A)
CODE
HEAP
...



CPU 1
running thread 1
PTBR
IP
SP
CPU 2
running thread 2
PTBR
IP
SP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir A)
CODE
HEAP
...


CPU 1
running thread 1
PTBR
IP
SP
CPU 2
running thread 2
PTBR
IP
SP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir A)
CODE
HEAP
STACK 1
STACK 2



threads executing different functions need different stacks

Chapters 28: spinlocks

Lock Goals

Correctness

Fairness

Performance

Test-and-set Spinlock

```
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin


void SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```

Test-and-set Spinlock (optimized)

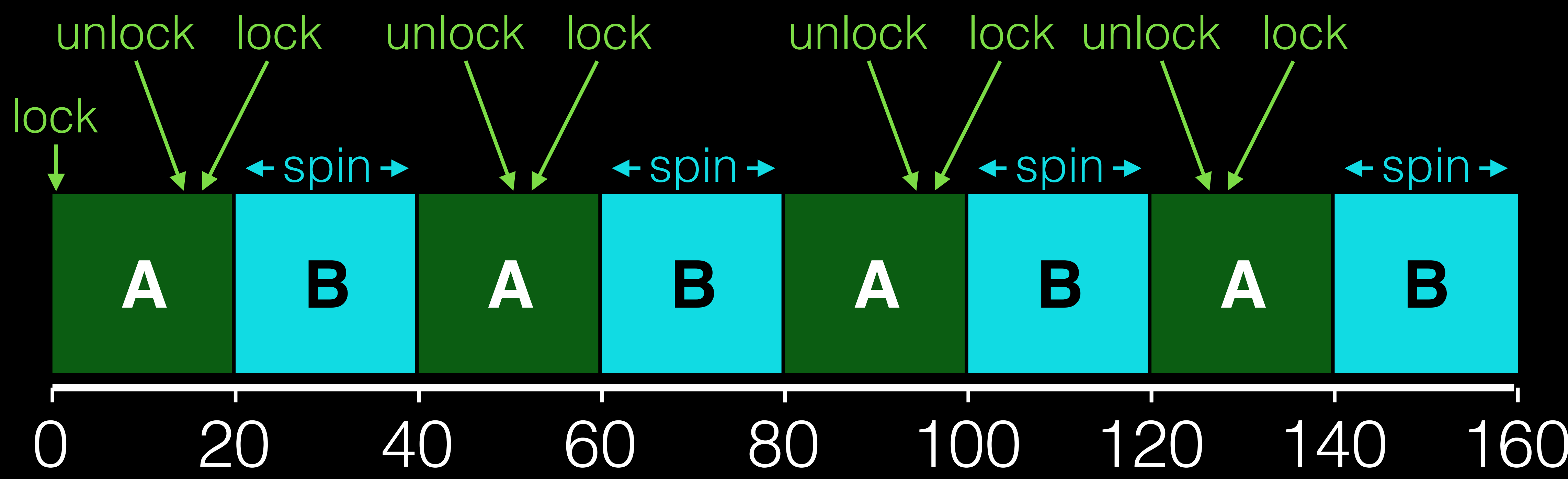
```
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin


void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Test-and-set Spinlock (optimized)

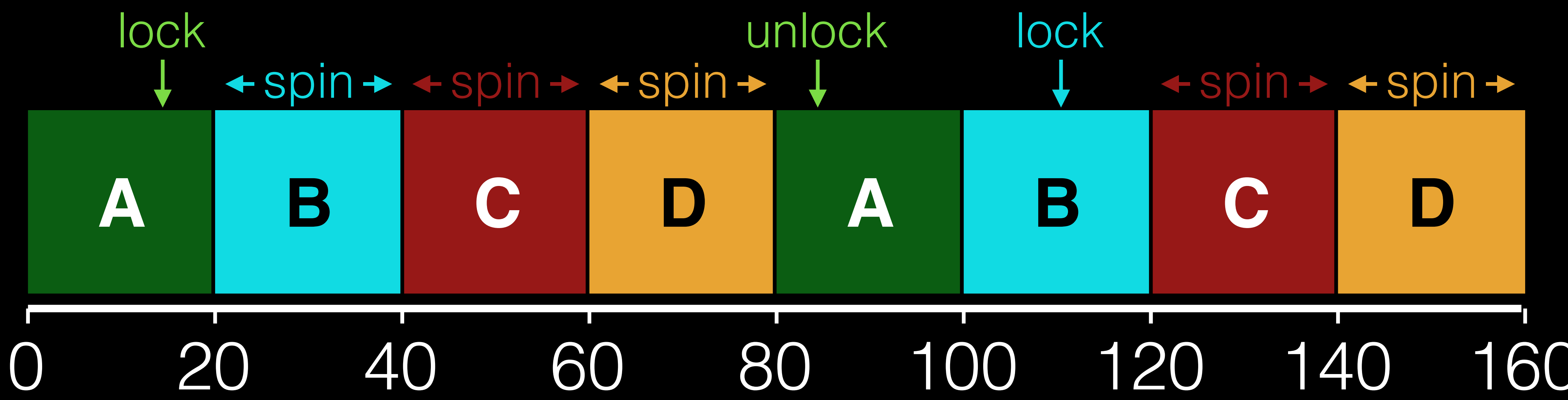
```
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin

void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Works on newer x86 processors.
Not on all CPUs (sometimes due to CPU bugs!)

Basic Spinlocks are Unfair



CPU Scheduler is Ignorant



CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

Chapters 30: condition variables

(and sleeping locks)

Queue Lock

RUNNABLE: A, B, C, D

RUNNING: <empty>

WAITING: <empty>

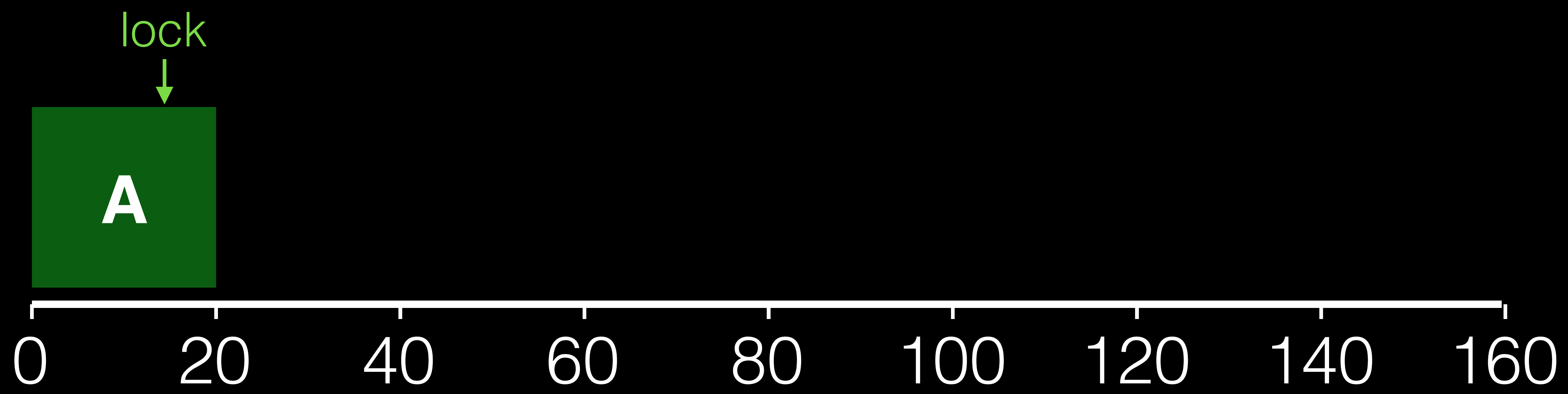
0 20 40 60 80 100 120 140 160

Queue Lock

RUNNABLE: B, C, D

RUNNING: A

WAITING: <empty>

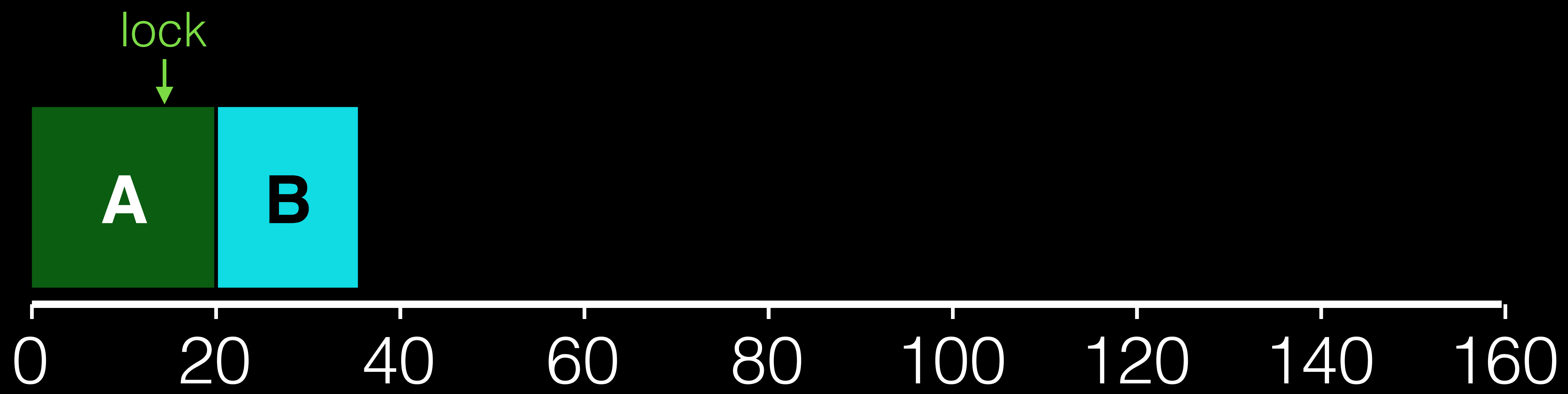


Queue Lock

RUNNABLE: C, D, A

RUNNING: B

WAITING: <empty>

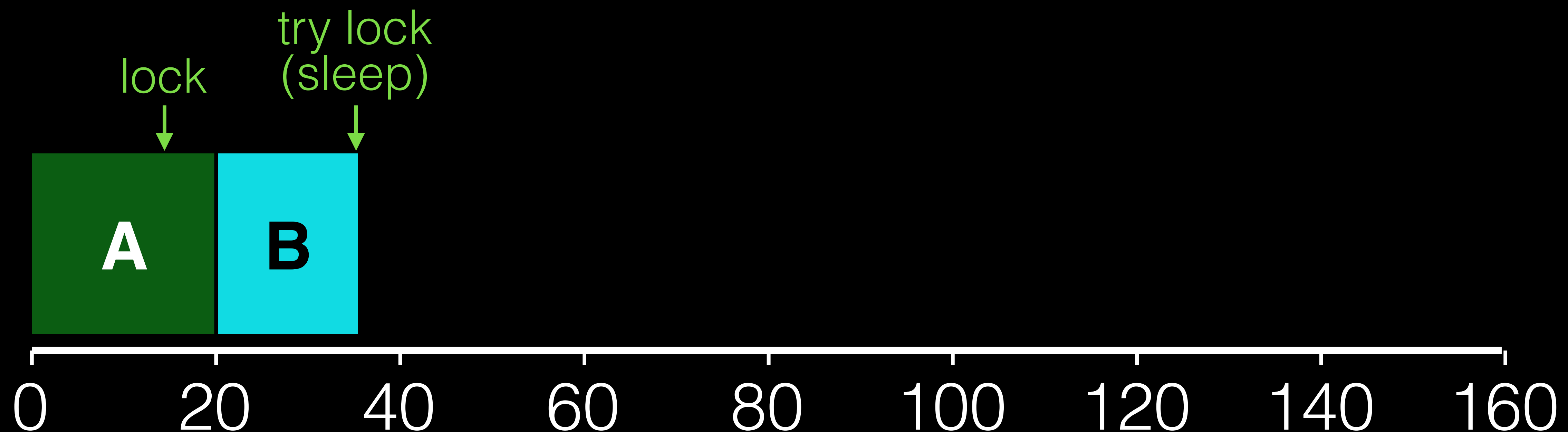


Queue Lock

RUNNABLE: C, D, A

RUNNING:

WAITING: B

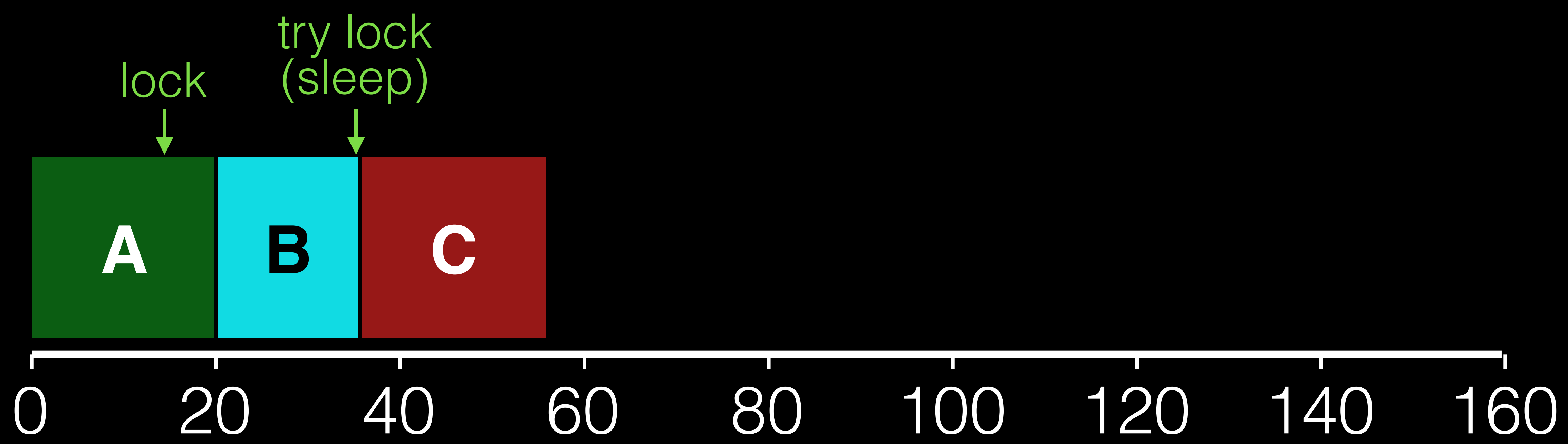


Queue Lock

RUNNABLE: D, A

RUNNING: C

WAITING: B

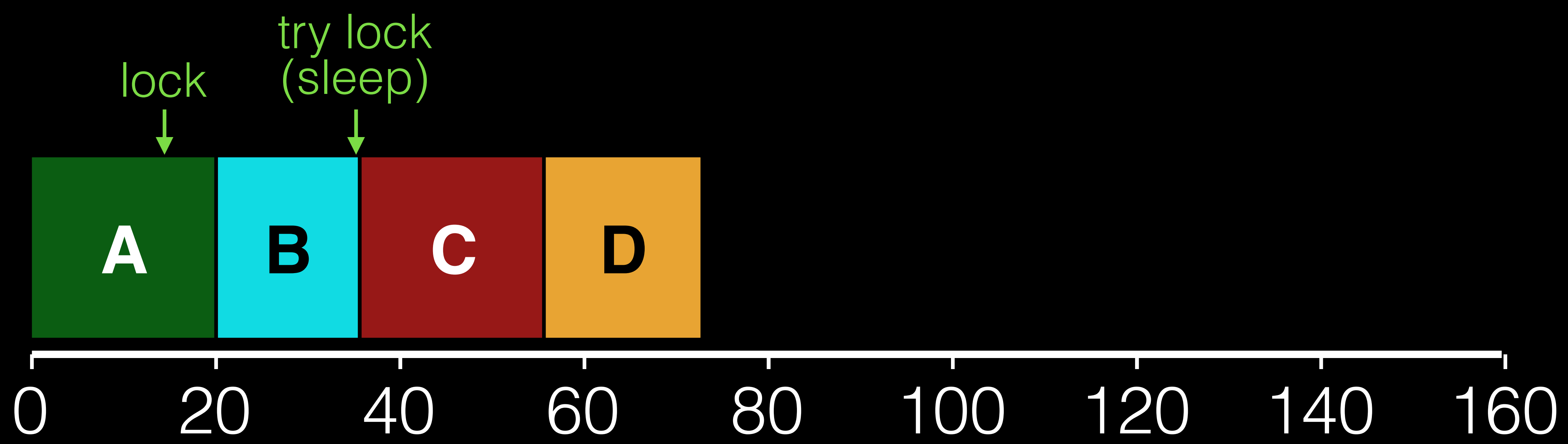


Queue Lock

RUNNABLE: A, C

RUNNING: D

WAITING: B

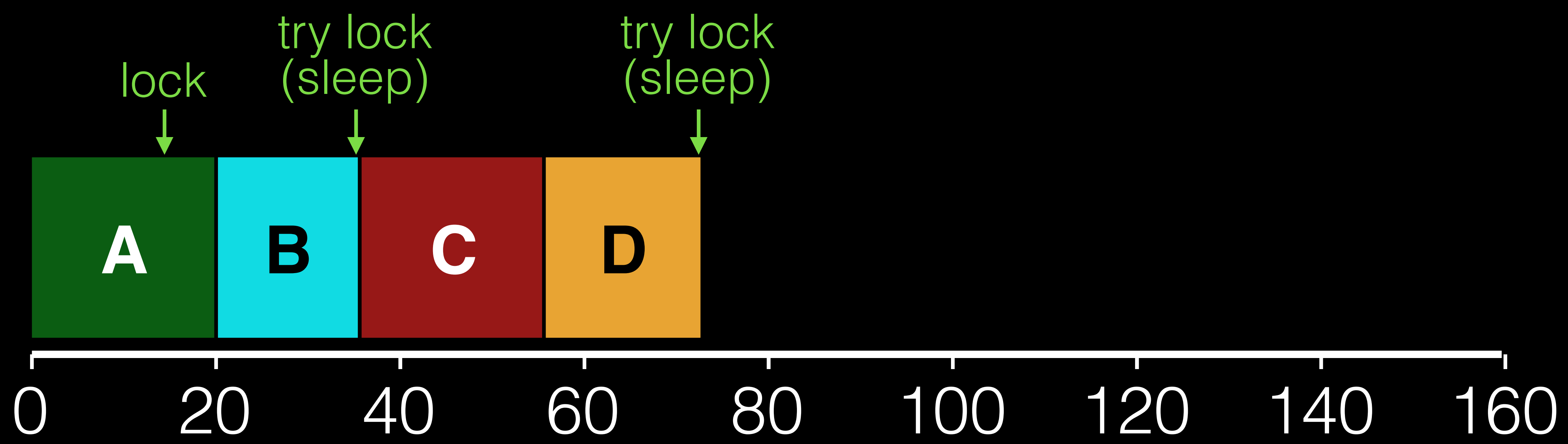


Queue Lock

RUNNABLE: A, C

RUNNING:

WAITING: B, D

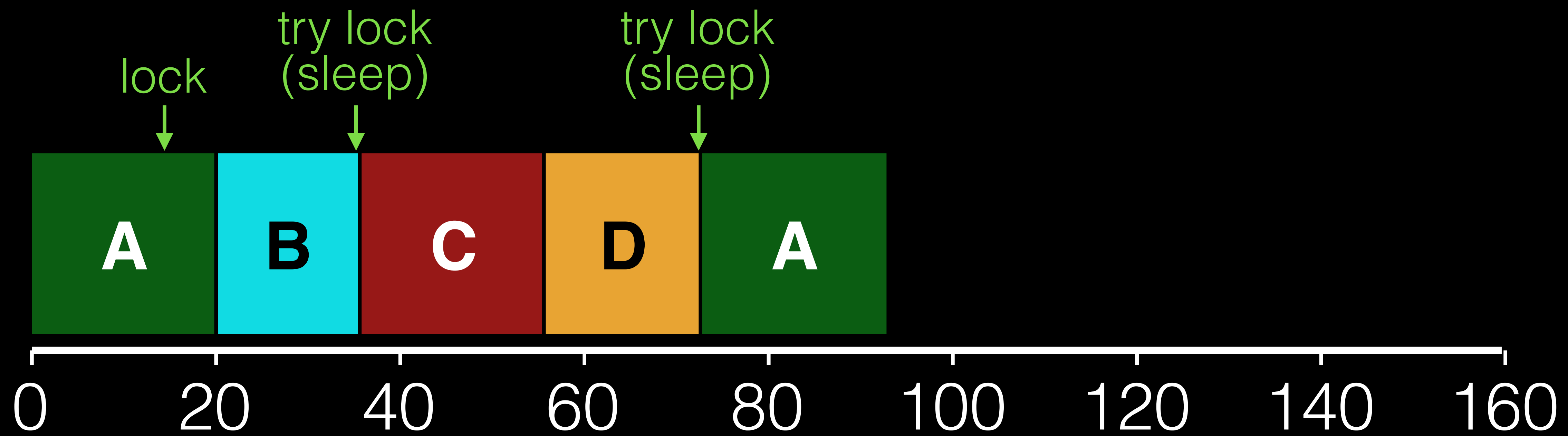


Queue Lock

RUNNABLE: C

RUNNING: A

WAITING: B, D

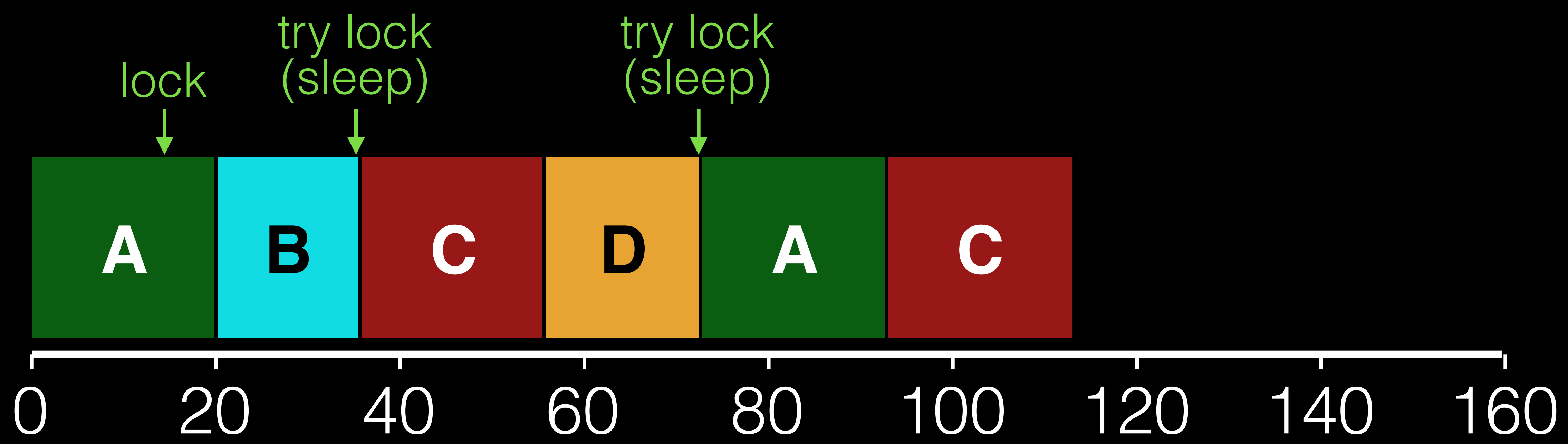


Queue Lock

RUNNABLE: A

RUNNING: C

WAITING: B, D

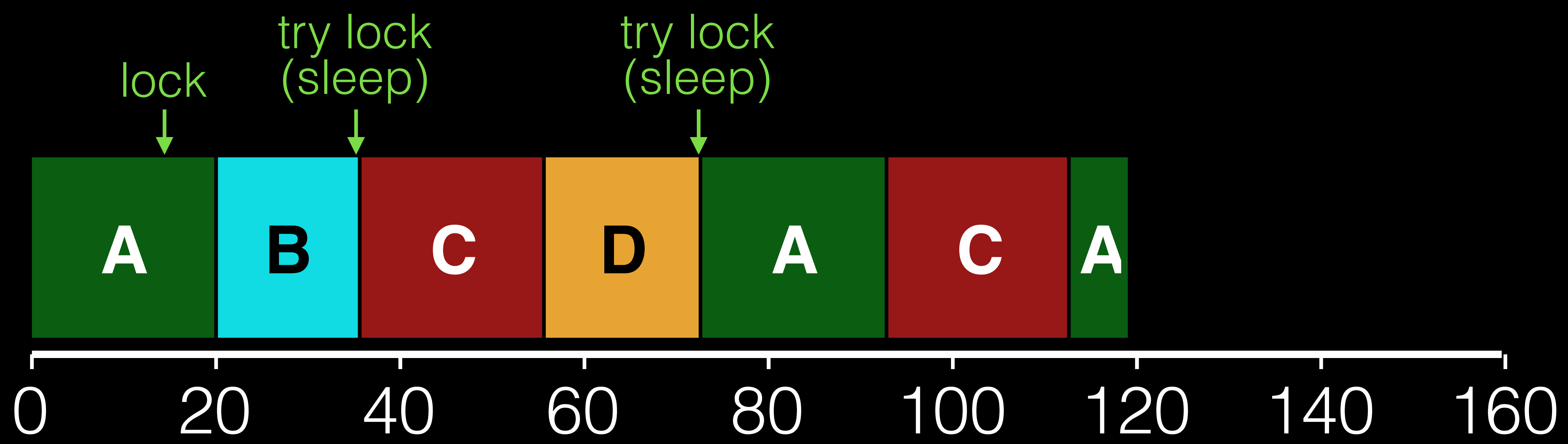


Queue Lock

RUNNABLE: C

RUNNING: A

WAITING: B, D

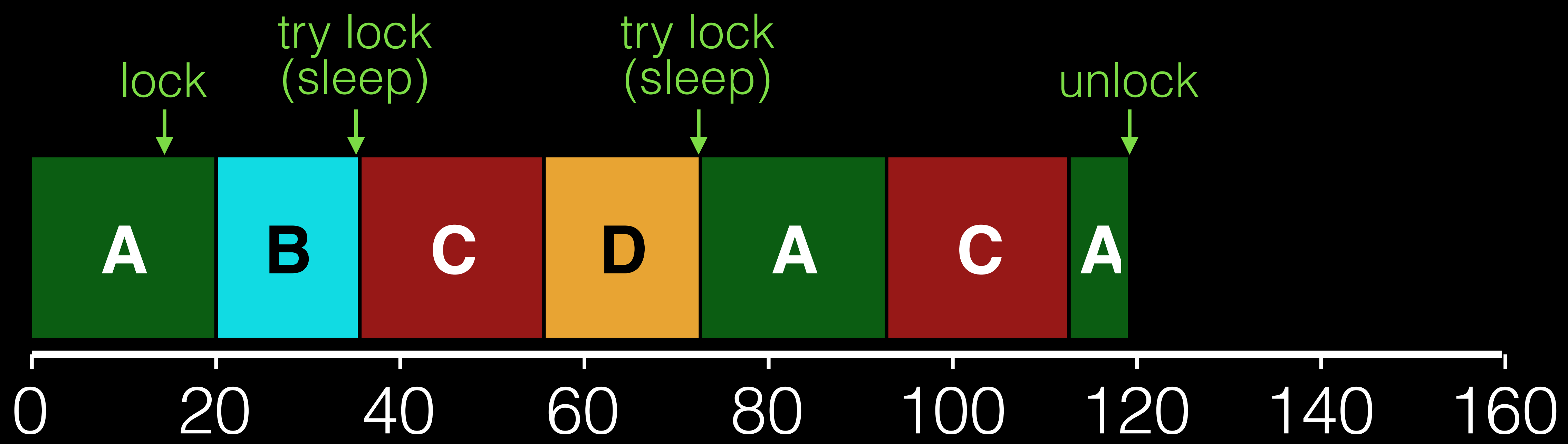


Queue Lock

RUNNABLE: B, C

RUNNING: A

WAITING: D

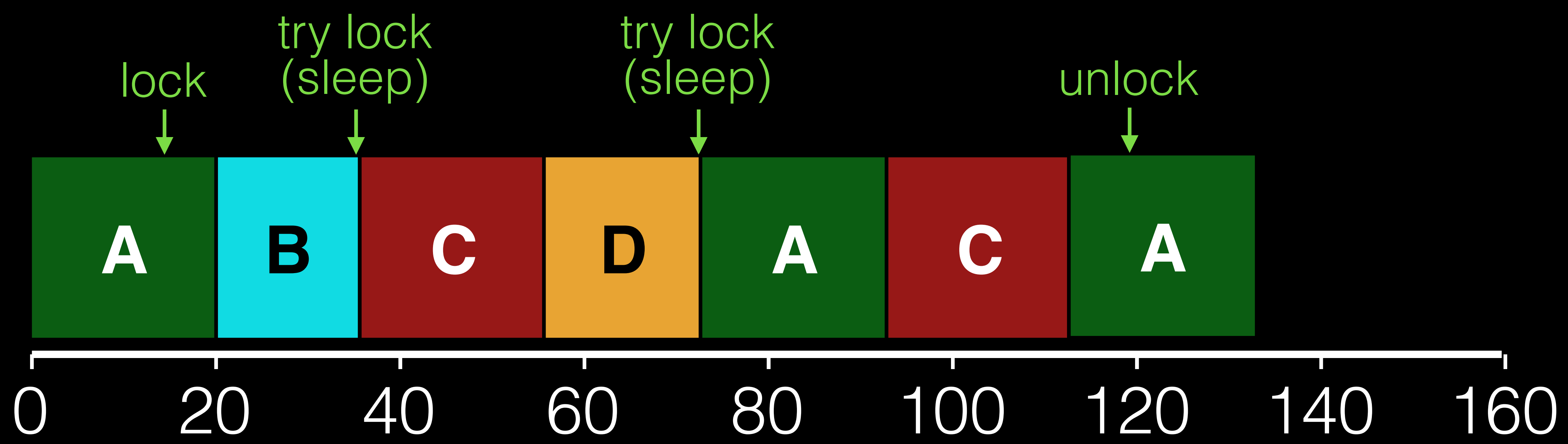


Queue Lock

RUNNABLE: B, C

RUNNING: A

WAITING: D

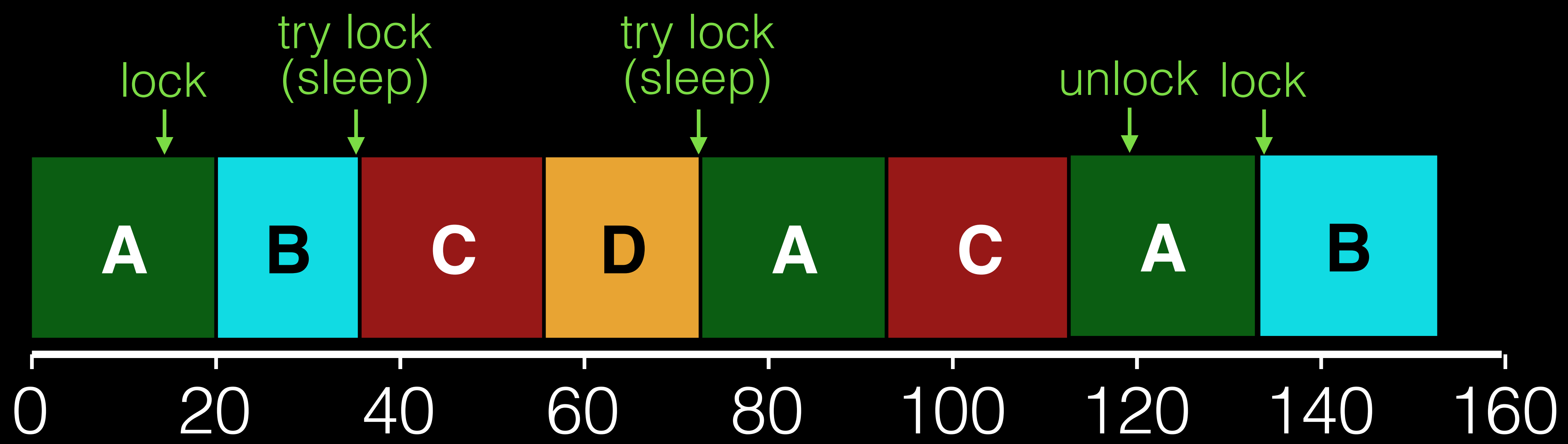


Queue Lock

RUNNABLE: C, A

RUNNING: B

WAITING: D



Concurrency Objectives

Mutual exclusion (e.g., A and B don’t run at same time)
- solved with *locks*

Ordering (e.g., B runs after A)
- solved with *condition variables*

Correct CV's

requires kernel support!

wait(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

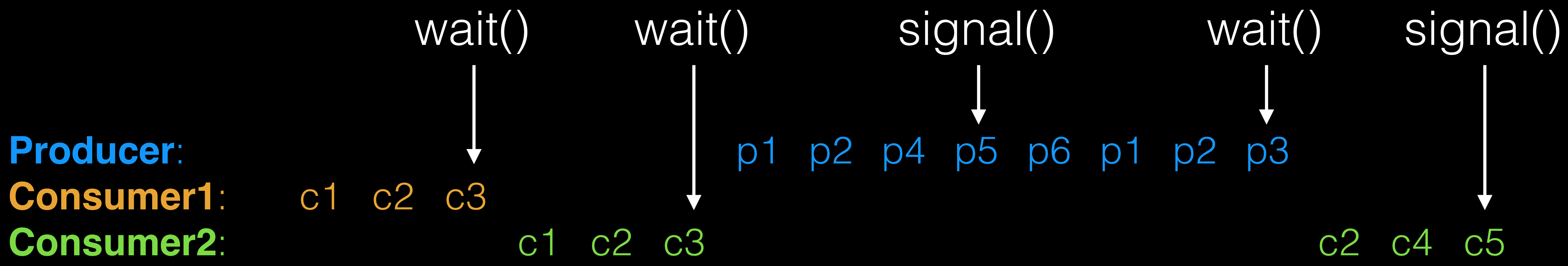
Produce/Consumer

Pipes
Web servers
Memory allocators
Device I/O
…

General strategy: use condition variables to make consumers wait when there is nothing to consume, and make producers wait when buffers are full.

What about 2 consumers (v1)?

Can you find a problematic timeline?



What about 2 consumers (v1)?

Can you find a problematic timeline?

wait() wait() signal() wait() signal()

Producer: p1 p2 p4 p5 p6 p1 p2 p3

Consumer1: c1 c2 c3

Consumer2: c1 c2 c3 c2 c4 c5

does this wake producer or consumer2?

How to wake the right thread?

One solution:

Better solution (usually): use two condition variables.

Chapters 31: semaphores

CV’s vs. Semaphores

CV rules of thumb:

- Keep state in addition to CV’s
- Always do wait/signal with lock held
- Whenever you acquire a lock, recheck state

How do semaphores eliminate these needs?

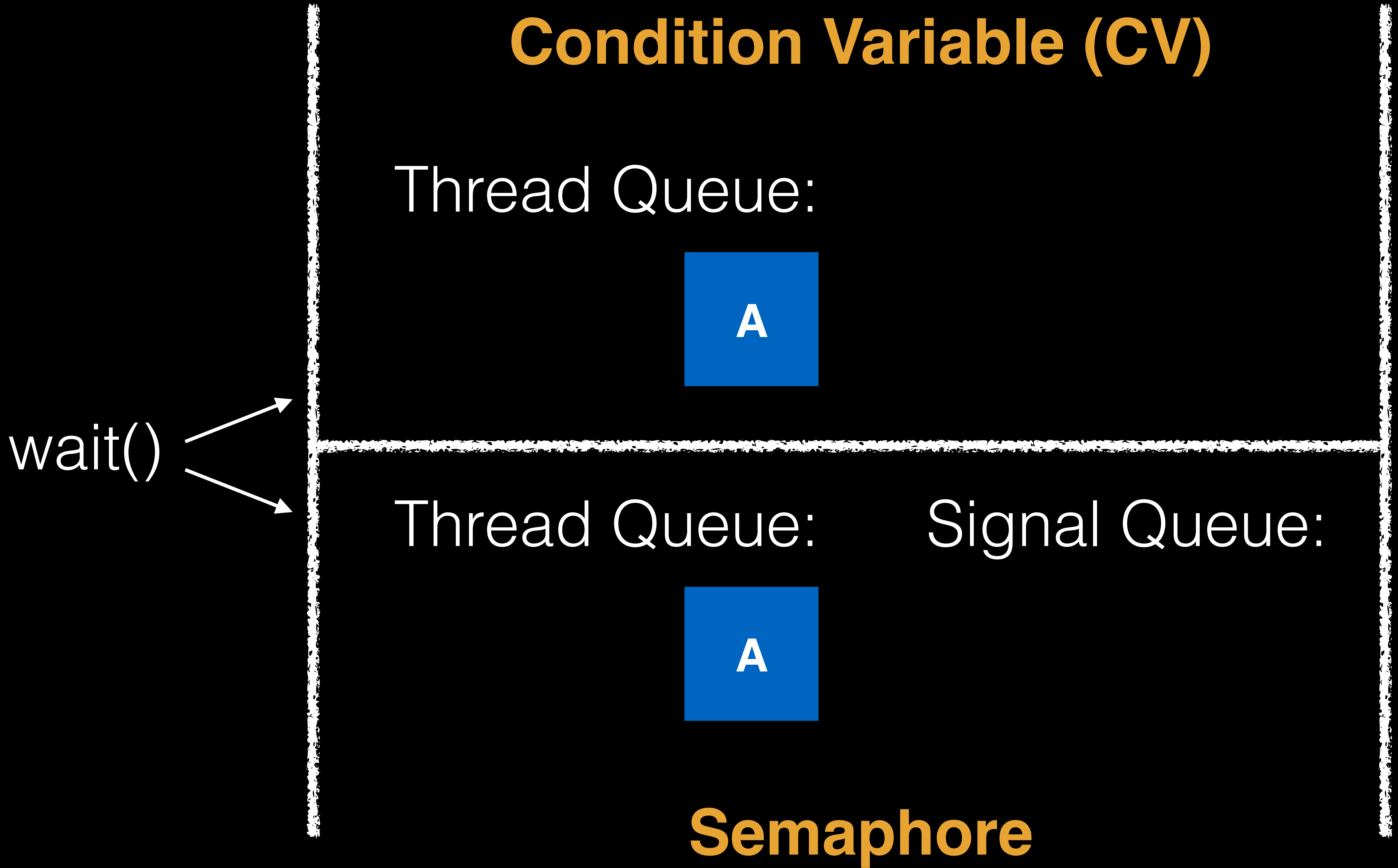
Condition Variable (CV)

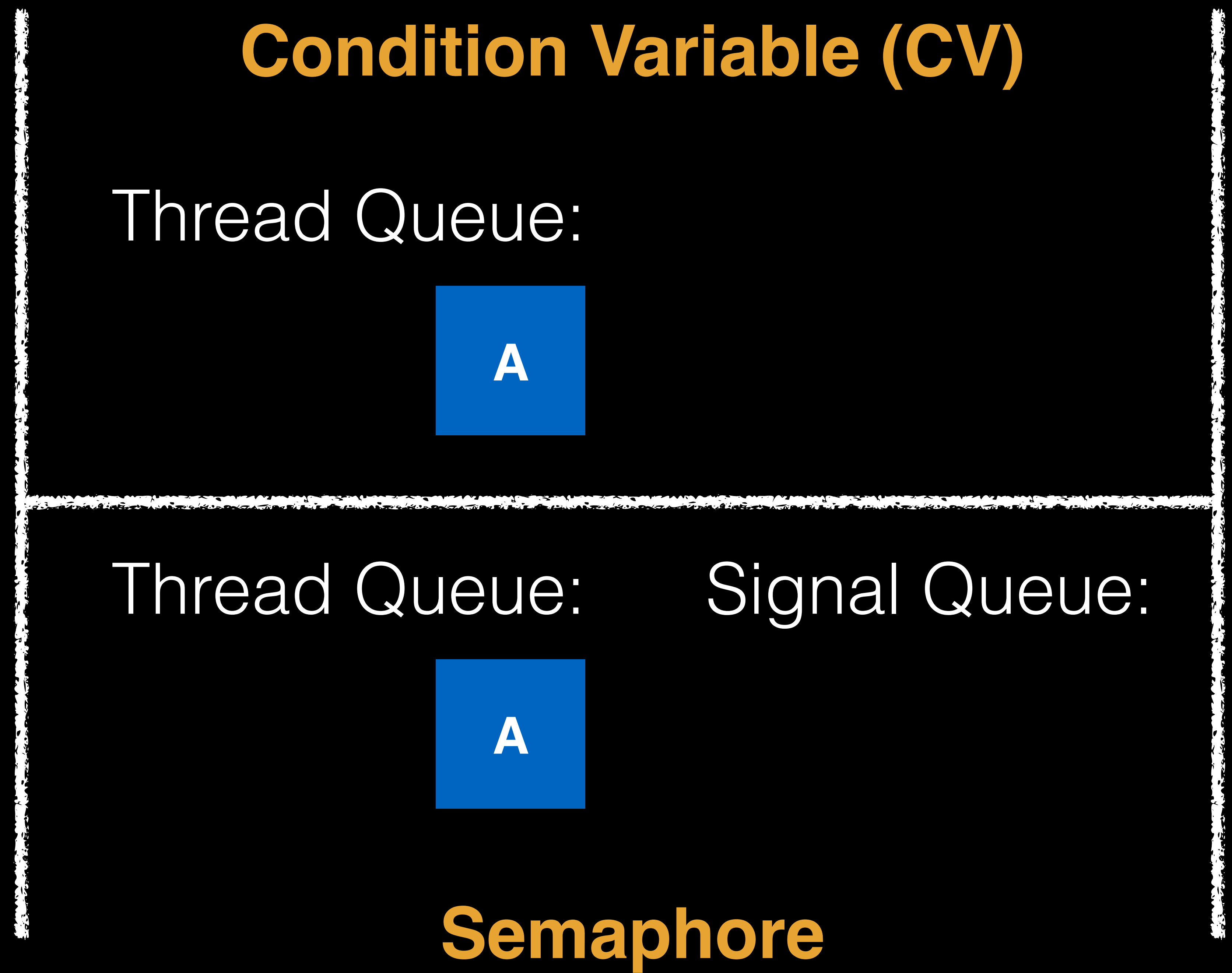
Thread Queue:

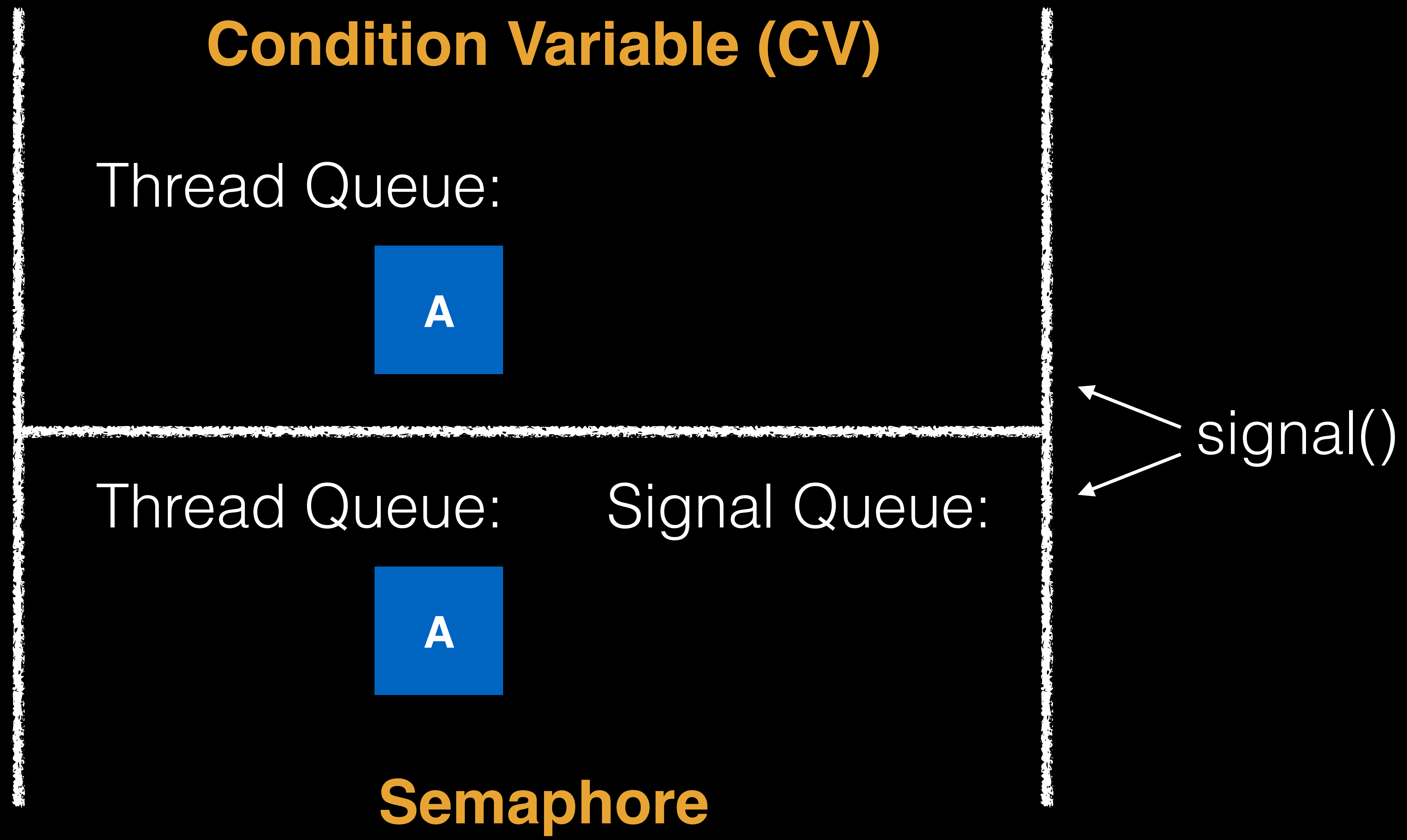
Thread Queue:

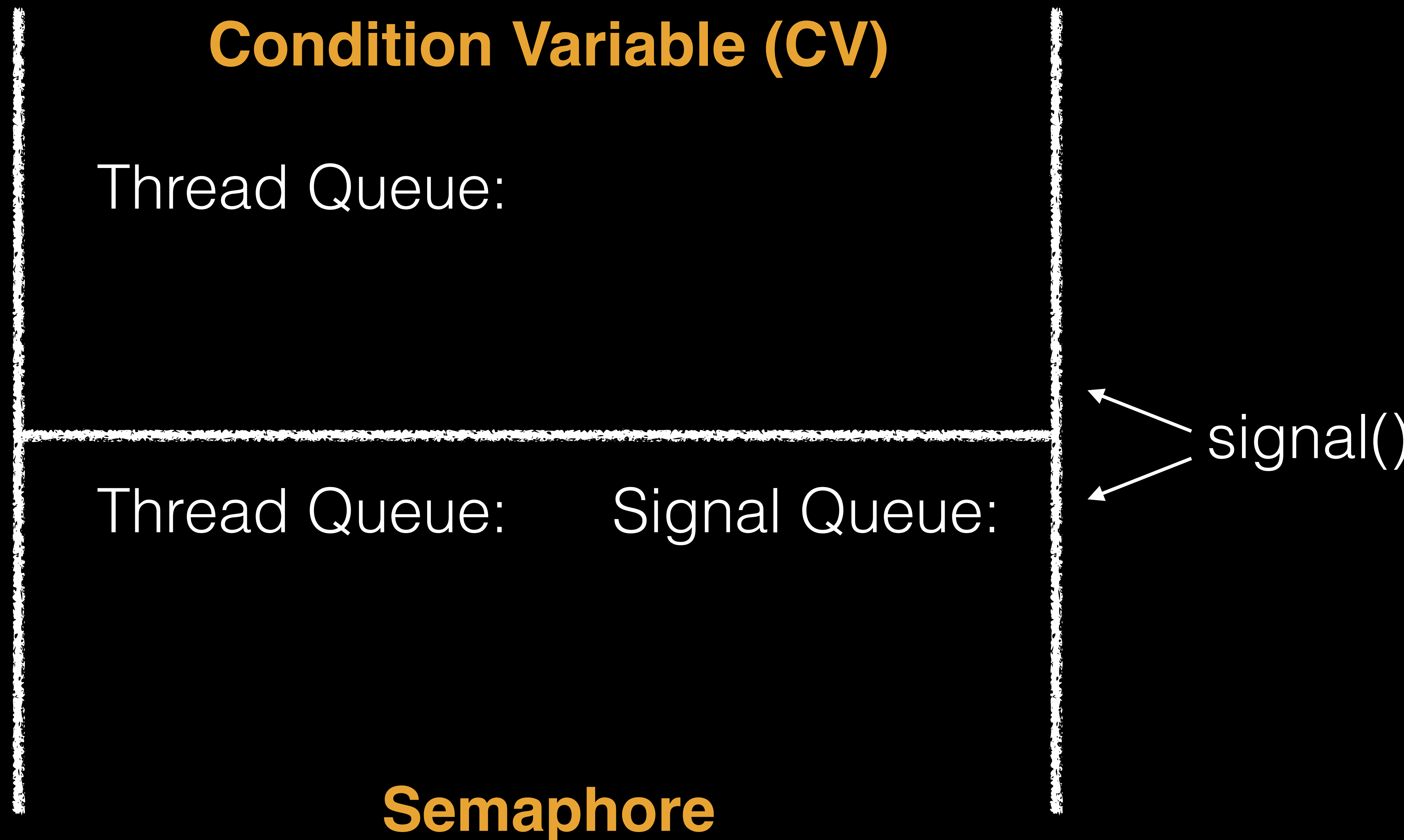
Signal Queue:

Semaphore


Condition Variable (CV)
Thread Queue:
A
wait()
Thread Queue:
Signal Queue:
A
Semaphore


Condition Variable (CV)
Thread Queue:
A
Thread Queue:
Signal Queue:
A
Semaphore


Condition Variable (CV)
Thread Queue:
A
signal()
Thread Queue:
Signal Queue:
A
Semaphore


Condition Variable (CV)
Thread Queue:
signal()
Thread Queue:
Signal Queue:
Semaphore

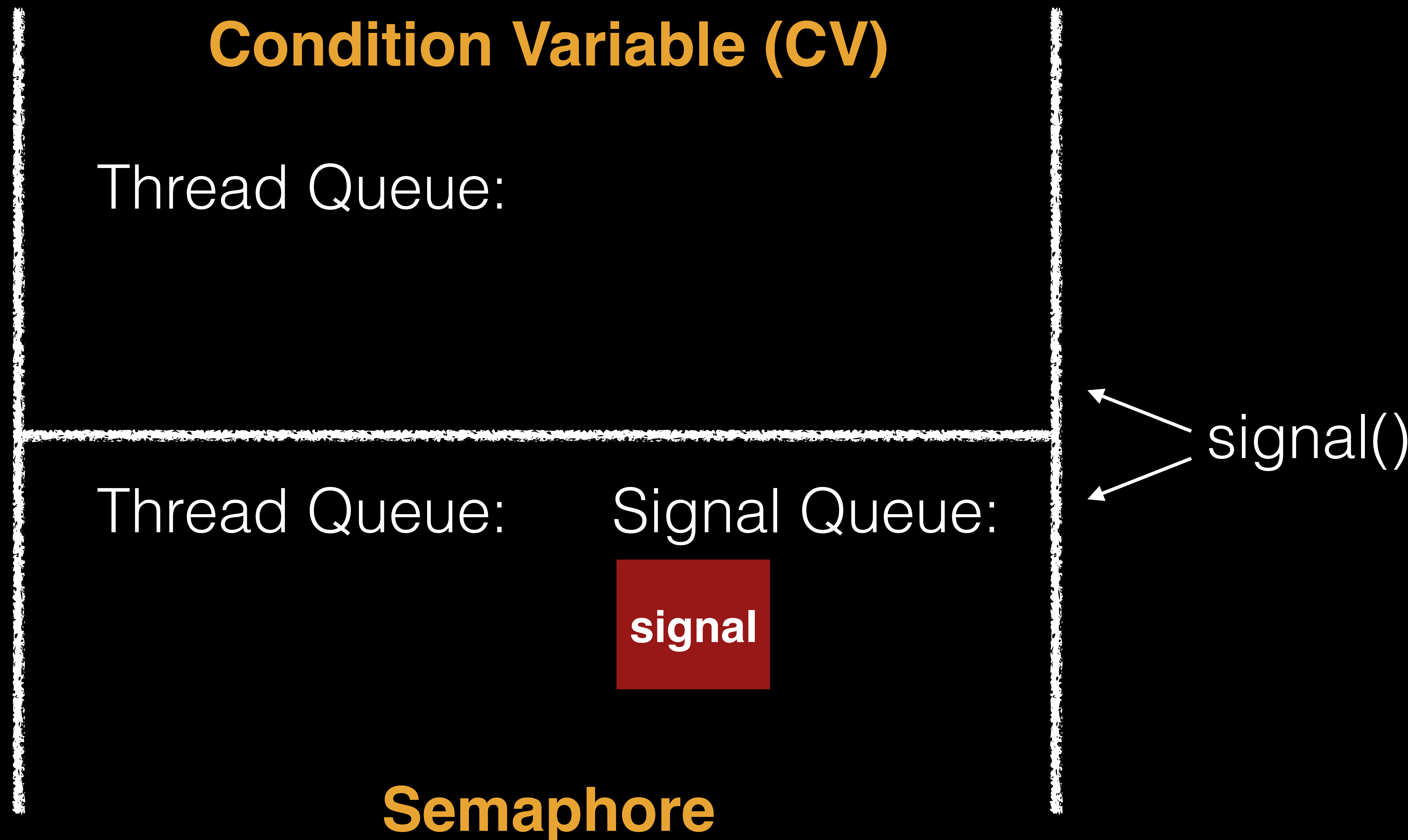
Condition Variable (CV)

Thread Queue:

Thread Queue:

Signal Queue:

Semaphore


Condition Variable (CV)
Thread Queue:
signal()
Thread Queue:
Signal Queue:
signal
Semaphore

Condition Variable (CV)

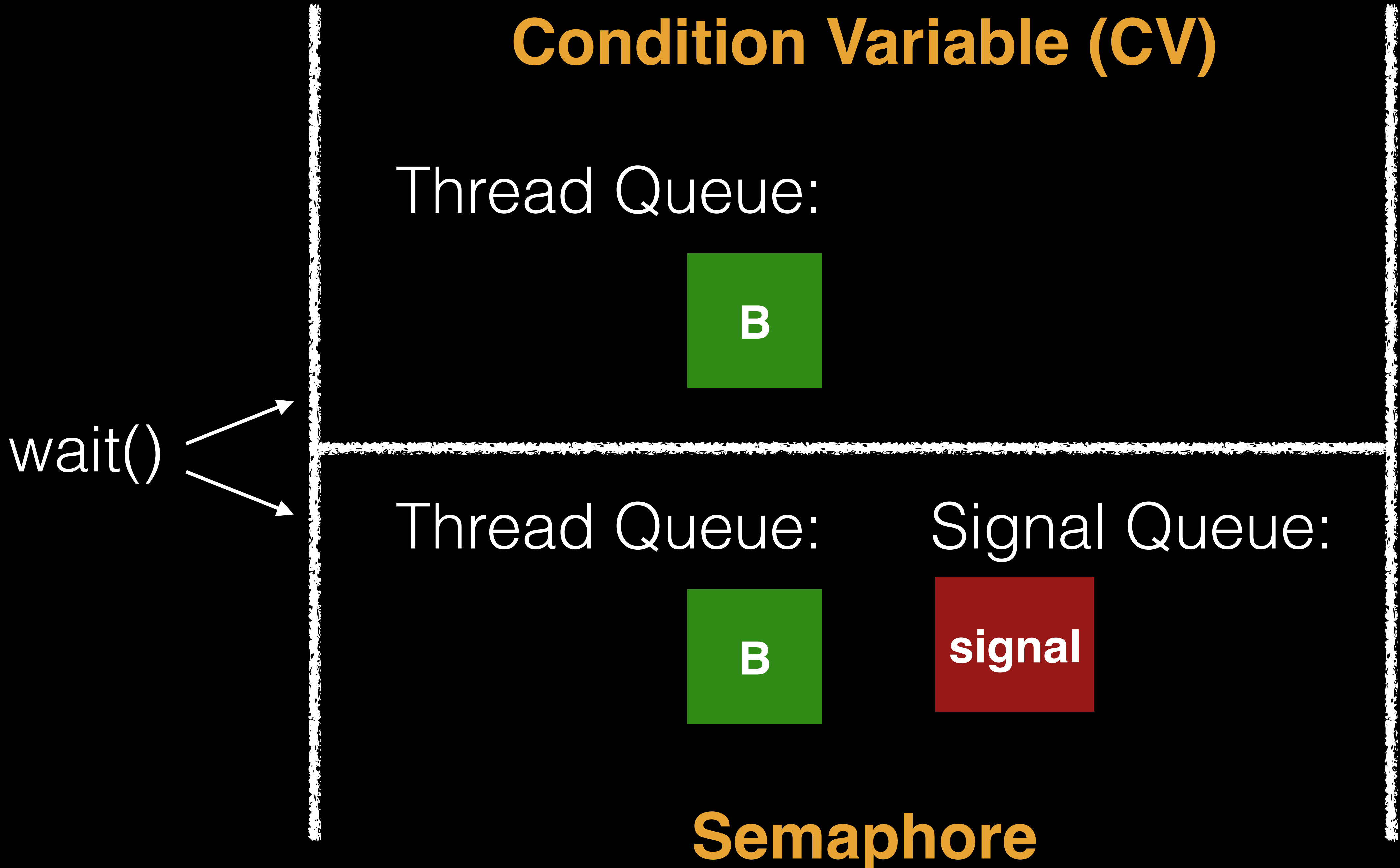
Thread Queue:

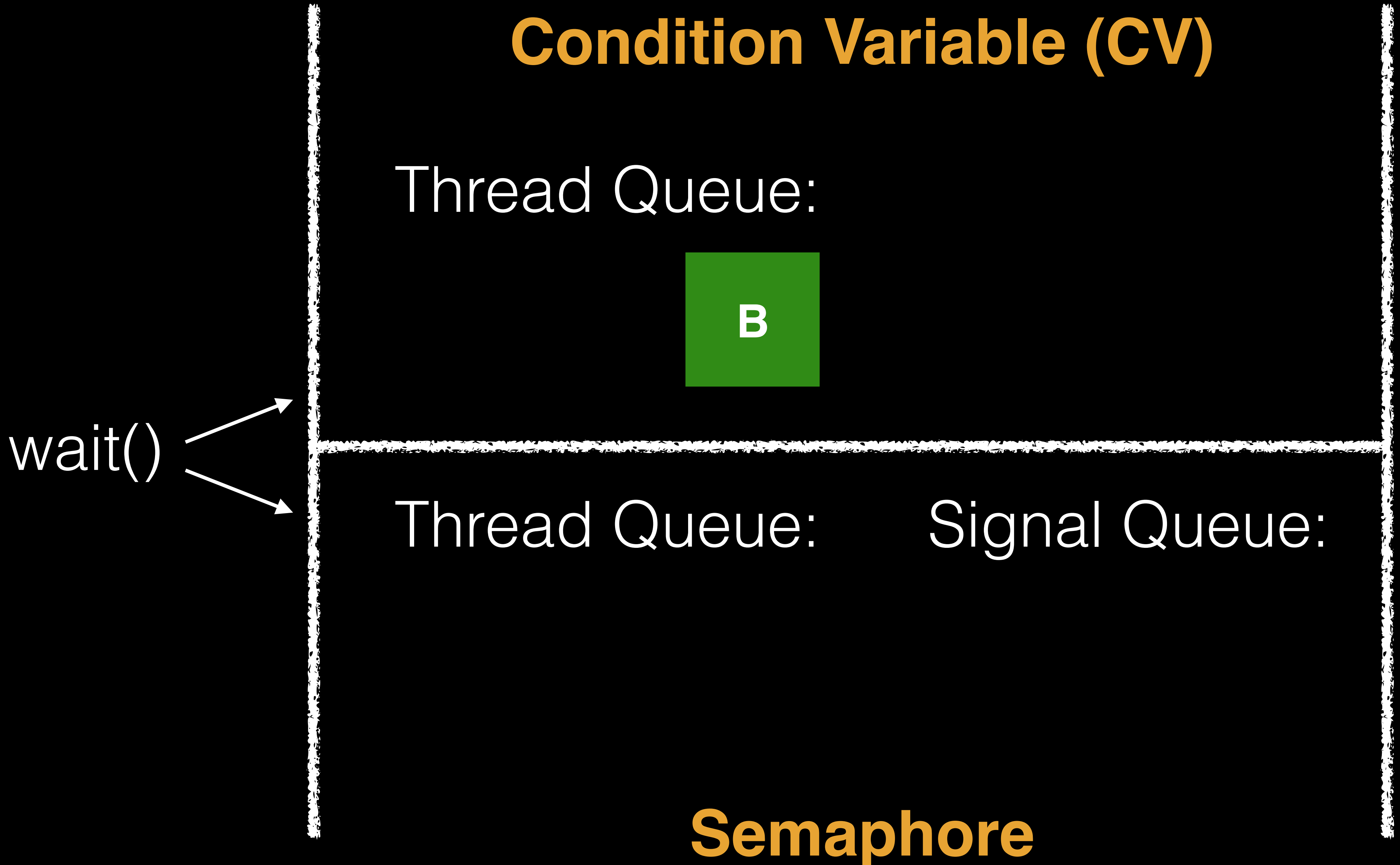
Thread Queue:

Signal Queue:

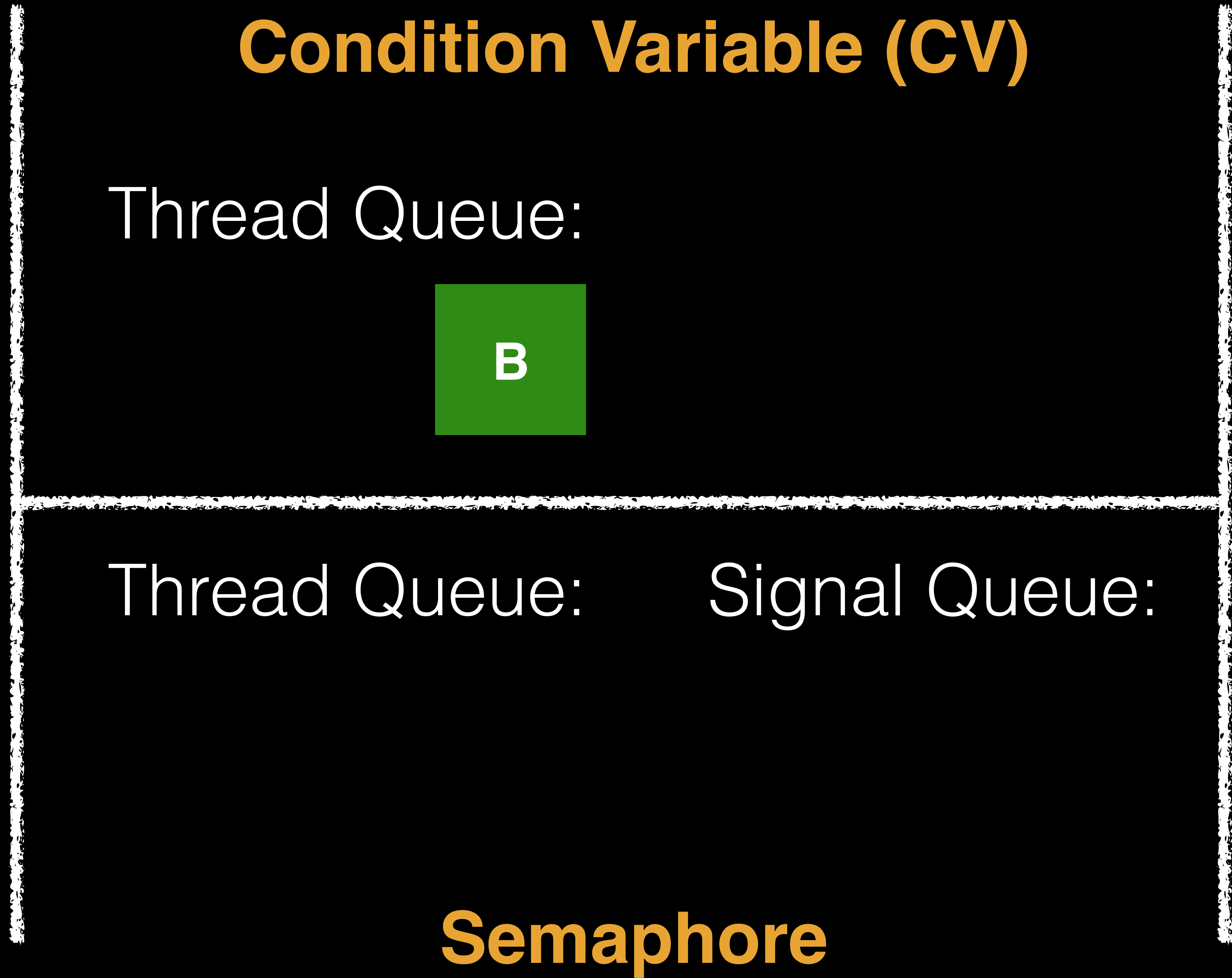
signal

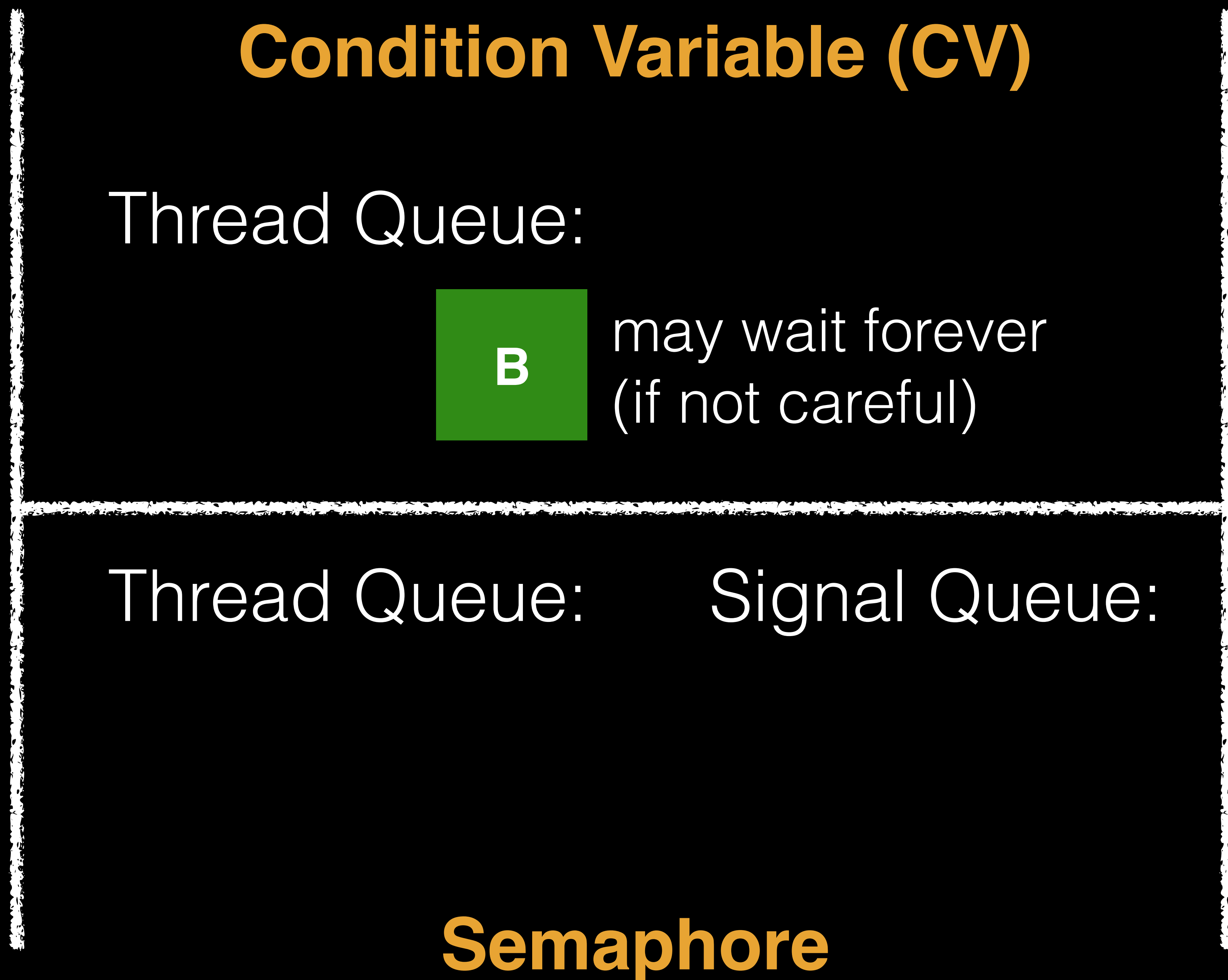
Semaphore


Condition Variable (CV)
Thread Queue:
B
wait()
Thread Queue:
Signal Queue:
B
signal
Semaphore



Condition Variable (CV)
Thread Queue:
B
wait()
Thread Queue:
Signal Queue:
Semaphore


Condition Variable (CV)
Thread Queue:
B
Thread Queue:
Signal Queue:
Semaphore



Condition Variable (CV)
Thread Queue:
B
may wait forever
(if not careful)
Thread Queue:
Signal Queue:
Semaphore

Condition Variable (CV)

Thread Queue:

B

may wait forever
(if not careful)

Thread Queue:

~~Signal Queue:~~

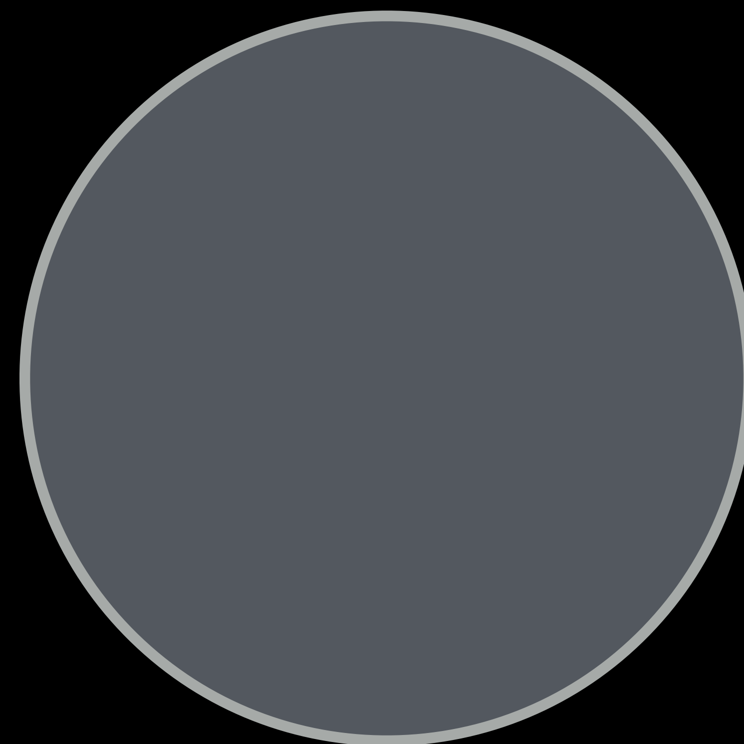
just use counter

Semaphore

Chapters 37: disks

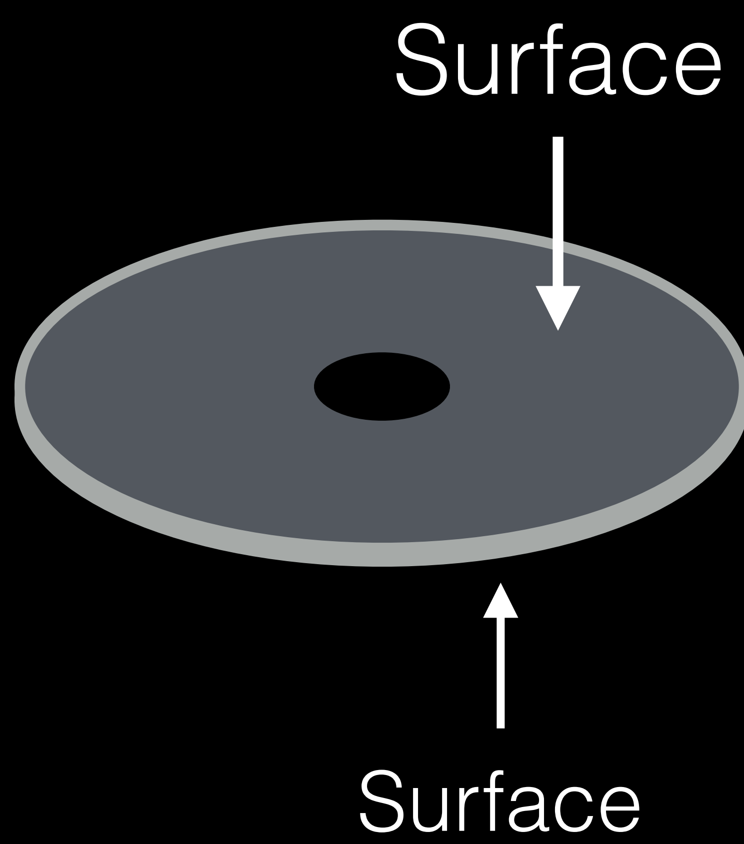
Disk Internals

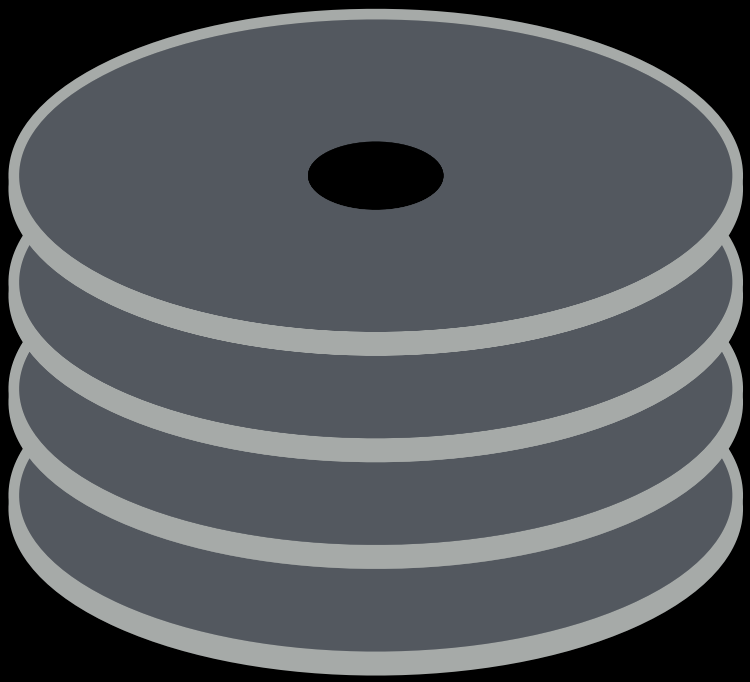
Platter



Platter is covered with a magnetic film.

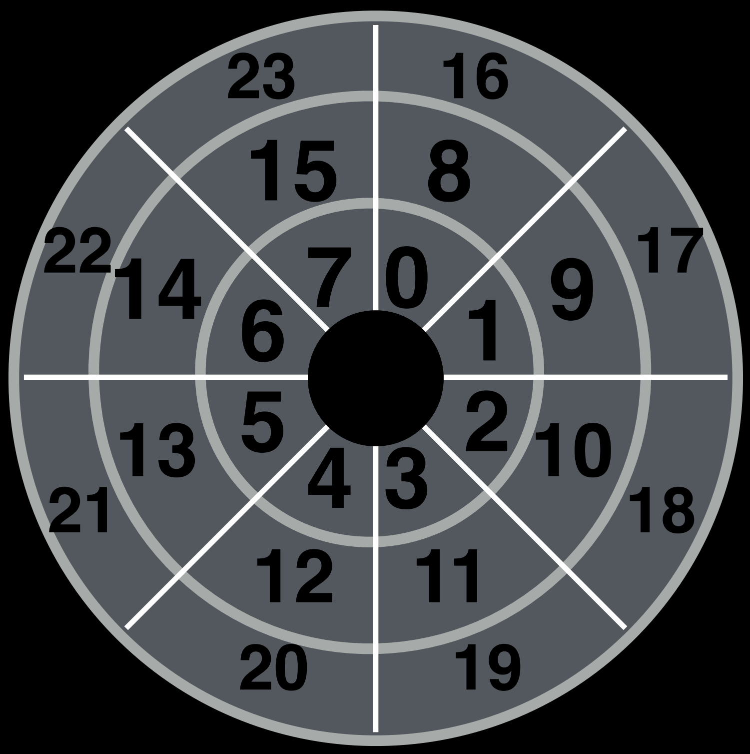
Spindle


Surface
Surface



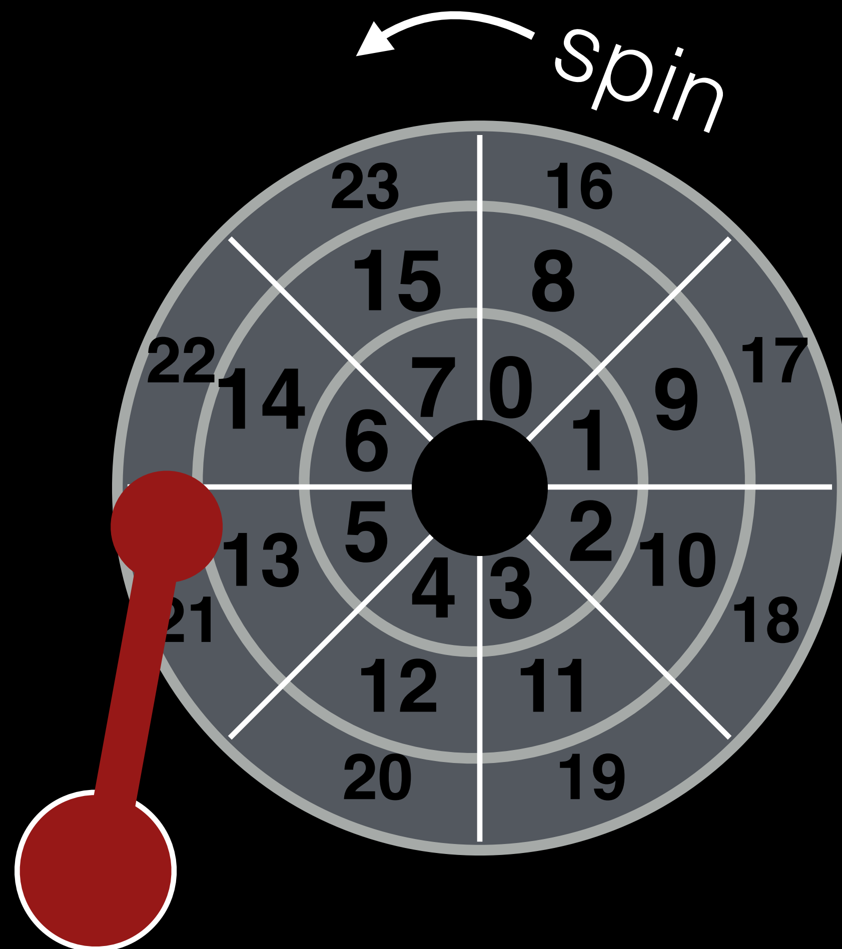
Many platters may be bound to the spindle.

Each surface is divided into rings called <u>tracks</u>.
A stack of tracks (across platters) is called a <u>cylinder</u>.



The tracks are divided into numbered sectors.

Heads on a moving arm can read from each surface.



Spindle/platters rapidly spin.

Workload

So…
- seeks are slow
- rotations are slow
- transfers are fast

What kind of workload is fastest for disks?

Sequential: access sectors in order (transfer dominated)

Random: access sectors arbitrarily (seek+rotation dominated)

Other Improvements

Track Skew

Zones

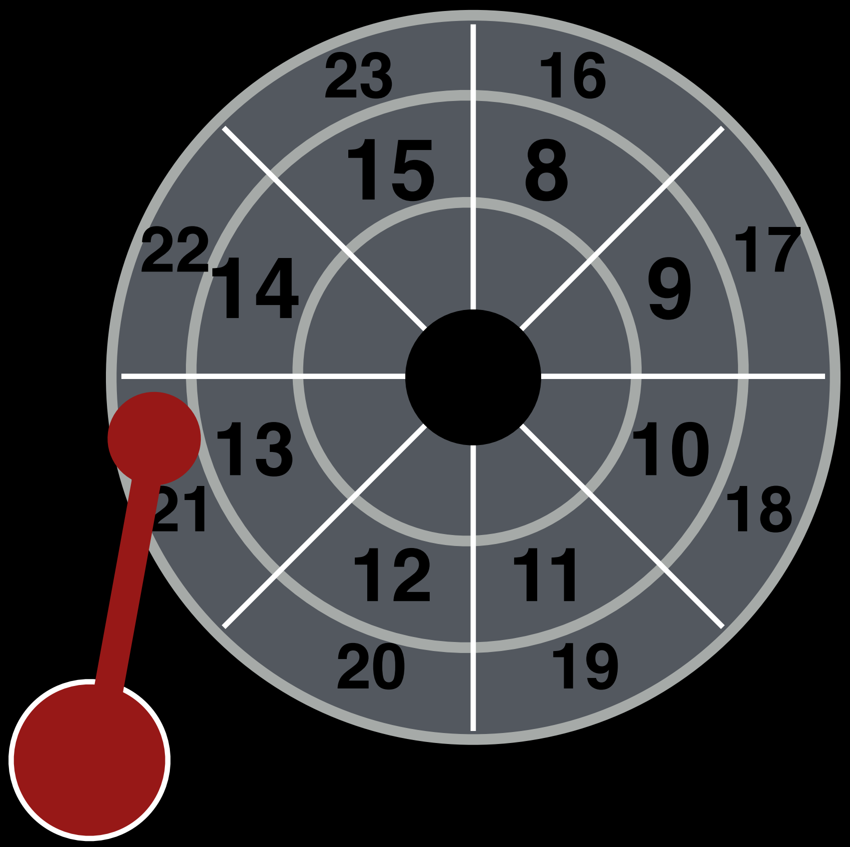
Cache

Other Improvements

Track Skew

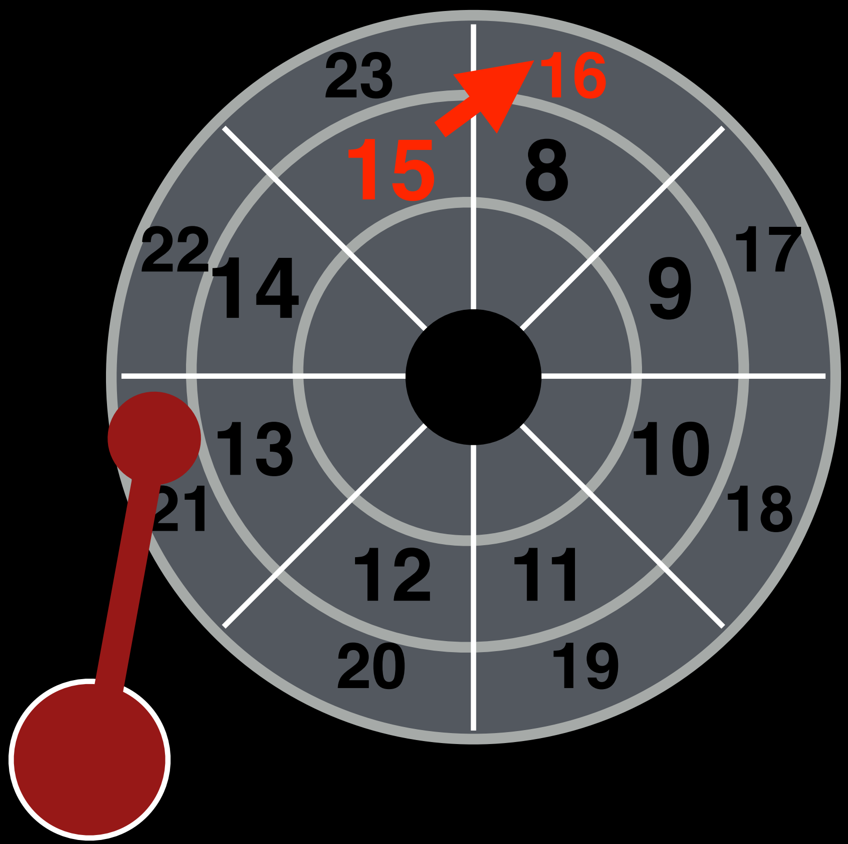
Zones

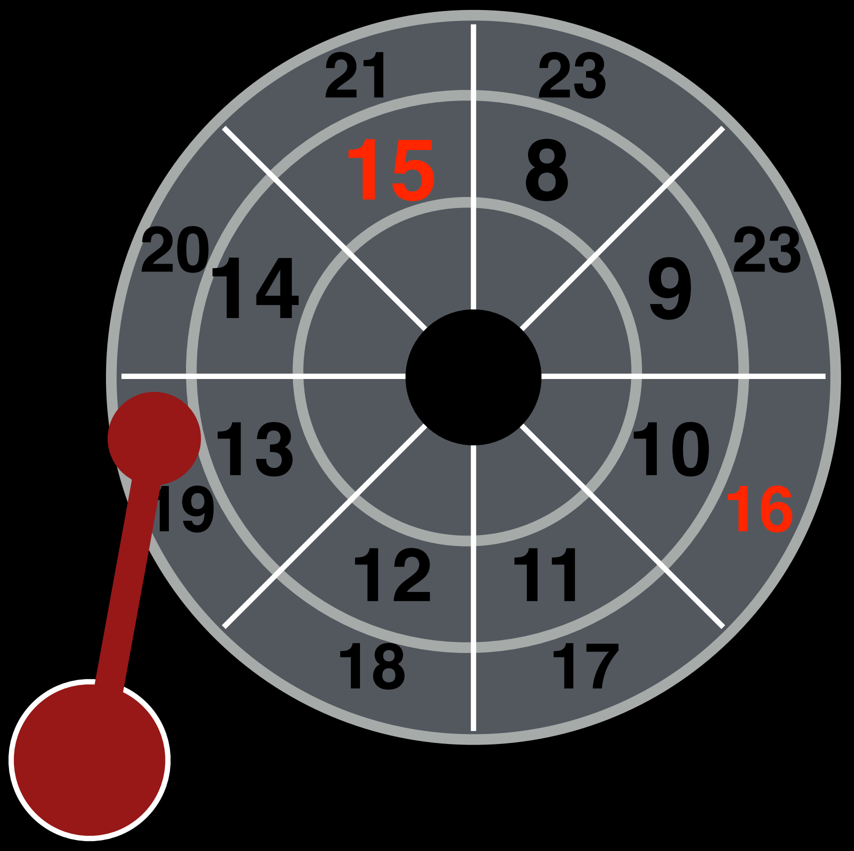
Cache



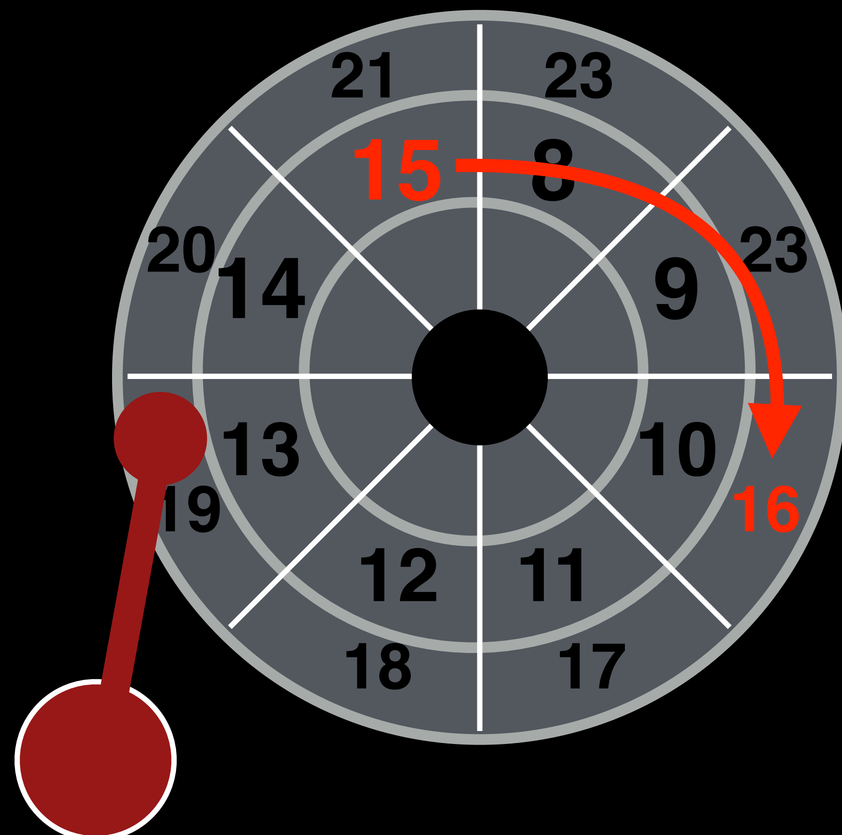
23
16
15
8
22
14
17
9
13
10
21
18
12
11
20
19

When reading 16 after 15, the head won't settle quick enough, so we need to do a rotation.




21
23
15
8
20
14
9
23
13
10
19
16
12
11
18
17

enough time to settle now

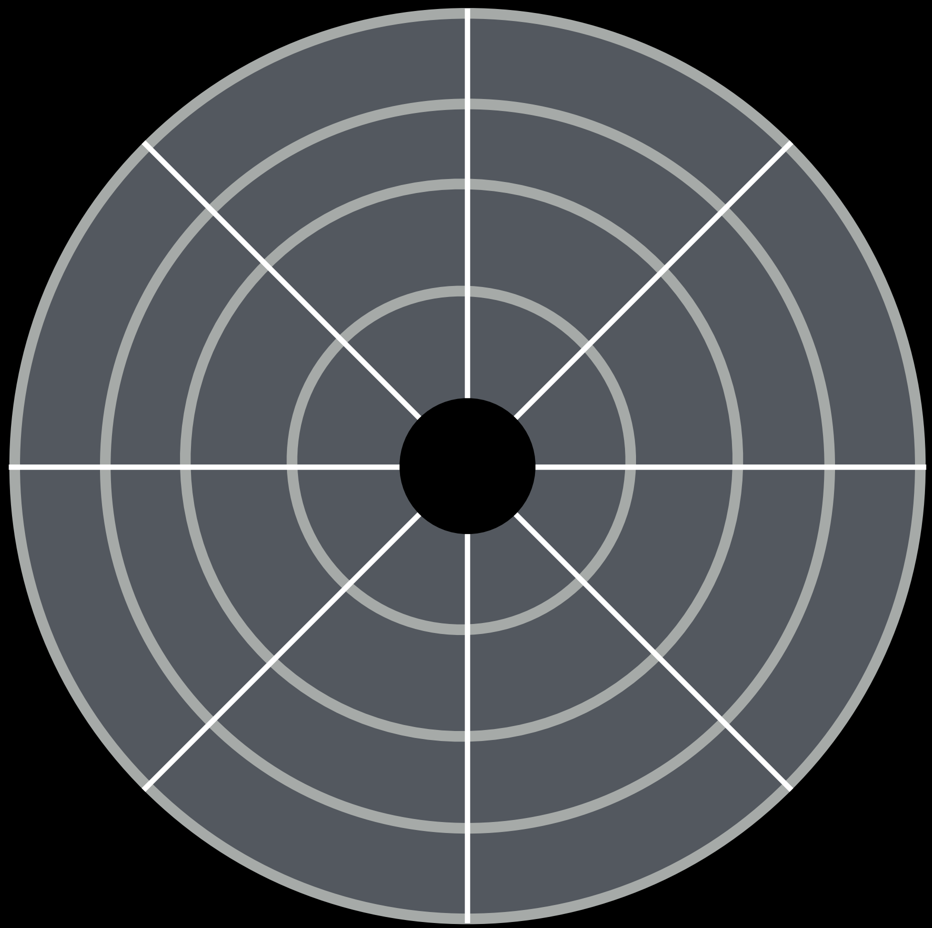


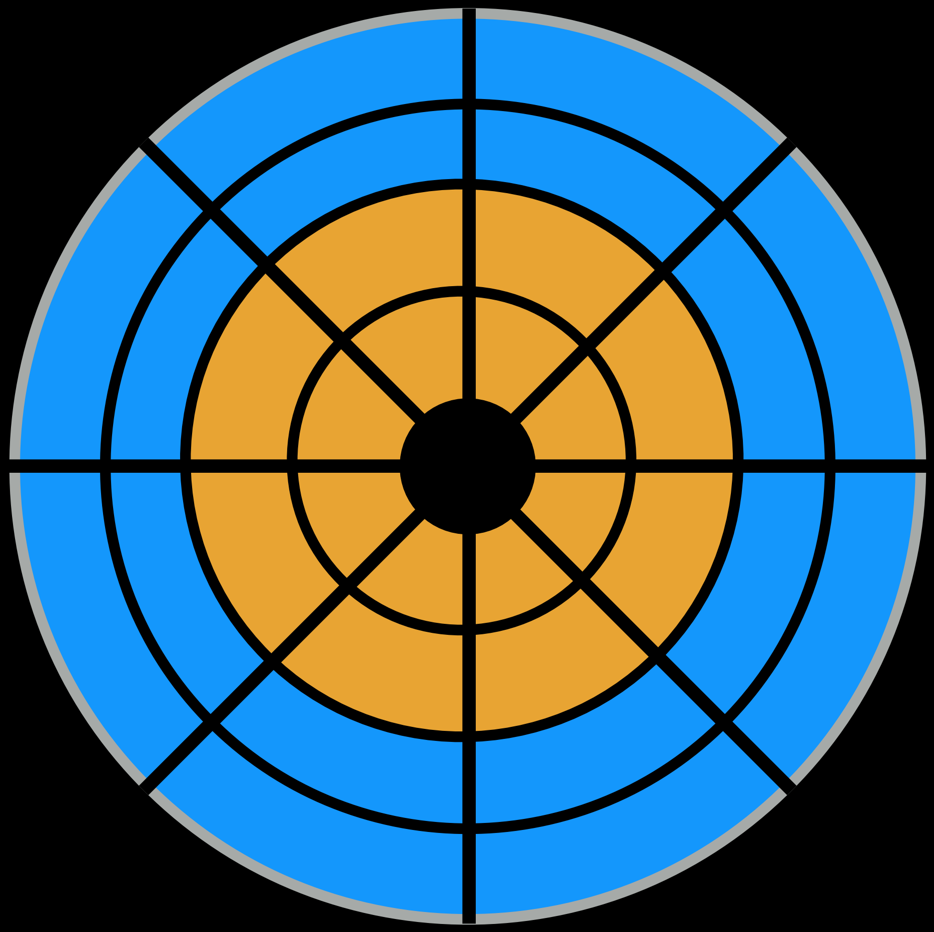
Other Improvements

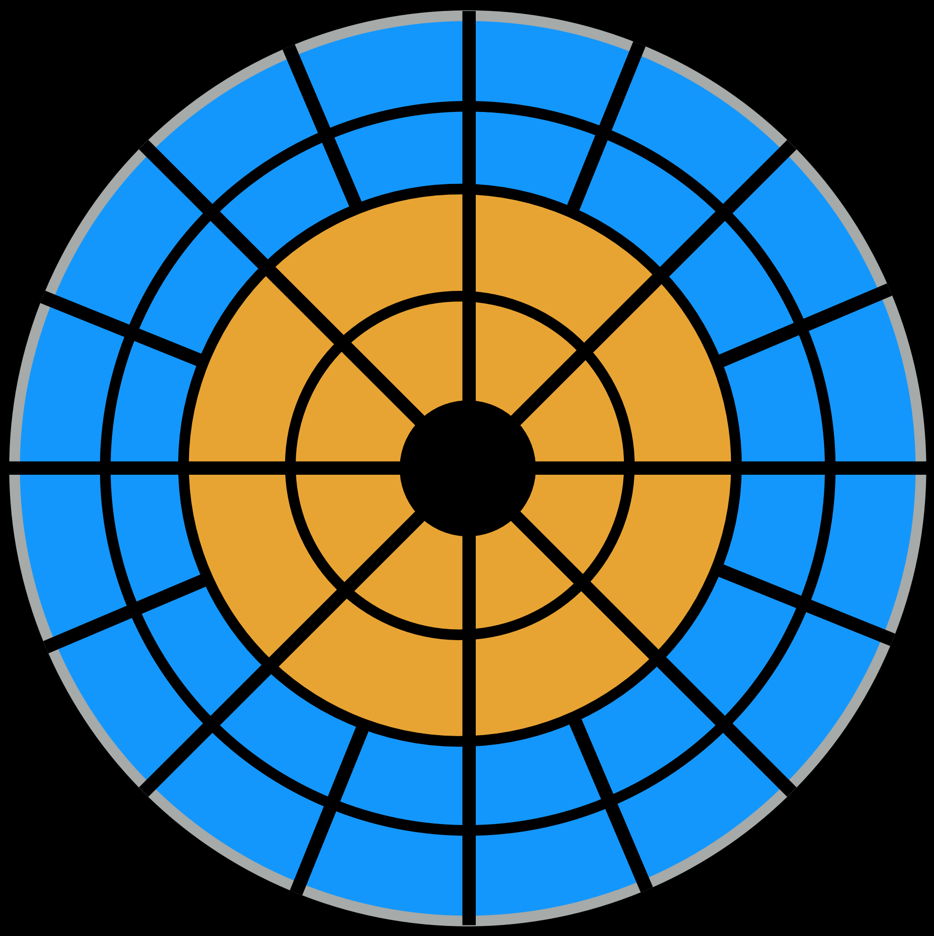
Track Skew

Zones

Cache







Other Improvements

Track Skew

Zones

Cache

Drive Cache

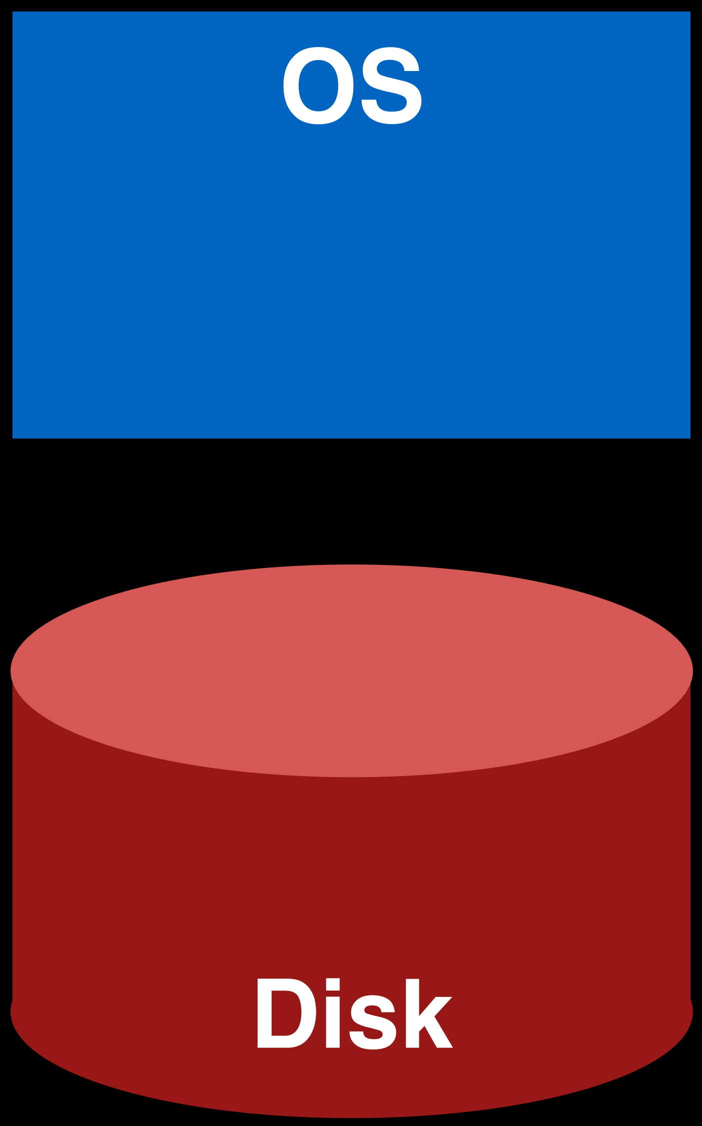
Drives may cache both reads and writes.

OS does this to.

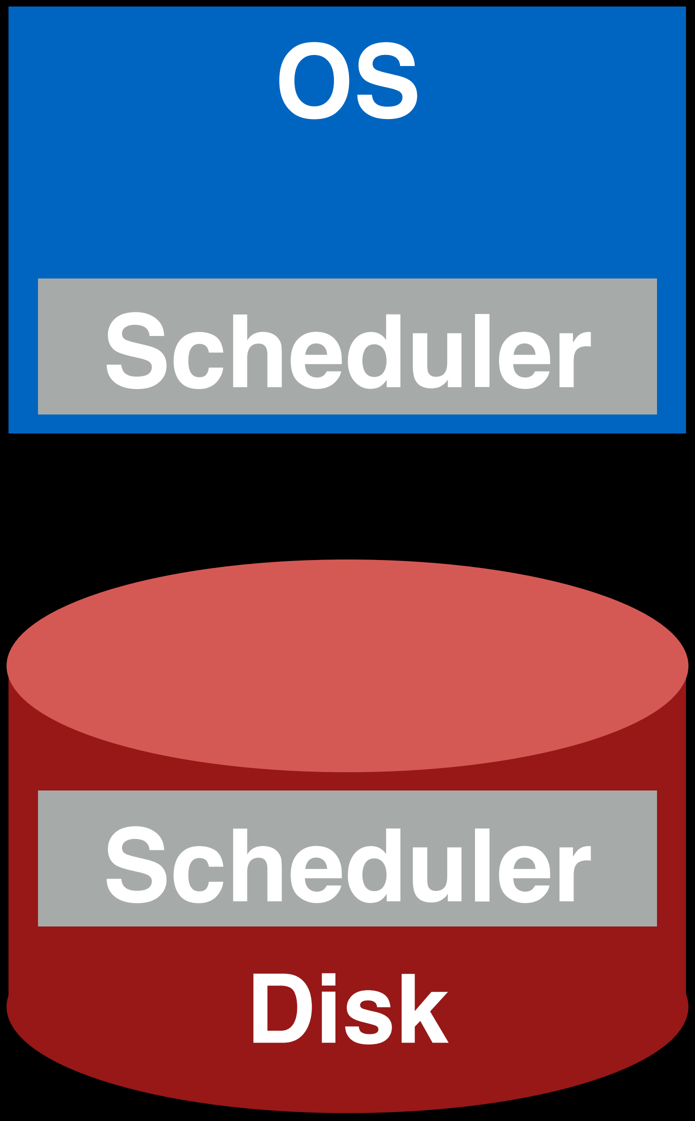
What advantage does drive have for reads?

What advantage does drive have for writes?

Schedulers



Schedulers



Where should the scheduler go?

SPTF (Shortest Positioning Time First)

Strategy: always choose the request that will take the least time for seeking and rotating.

How to implement in disk?
How to implement in OS?

SPTF (Shortest Positioning Time First)

Strategy: always choose the request that will take the least time for seeking and rotating.

How to implement in disk?
How to implement in OS?

Disadvantages?

SCAN

Sweep back and forth, from one end of disk to the other, serving requests as you go.

Pros/Cons?

SCAN

Sweep back and forth, from one end of disk to the other, serving requests as you go.

Pros/Cons?

Better: C-SCAN (circular scan)
 - only sweep in one direction

Chapters 38: RAID

All RAID

	Reliability	Capacity
RAID-0	0	C*N
RAID-1	1	C*N/2
RAID-4	1	N-1
RAID-5	1	N-1

All RAID

	Read Latency	Write Latency
RAID-0	D	D
RAID-1	D	D
RAID-4	D	2D
RAID-5	D	2D

All RAID

	Read Latency	Write Latency
RAID-0	D	D
RAID-1	D	D
RAID-4	D	2D
RAID-5	D	2D

but RAID-5 can
do more in parallel

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-4	(N-1)*S	(N-1)*S	(N-1)*R	R/2
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-4	(N-1)*S	(N-1)*S	(N-1)*R	R/2
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

RAID-5 is strictly better than RAID-4

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

RAID-0 is always fastest and has best capacity.
(but at cost of reliability)

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

RAID-5 better than RAID-1 for sequential.

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

All RAID

	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	N * S	N * S	N * R	N * R
RAID-1	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-5	(N-1)*S	(N-1)*S	N * R	N/4 * R

RAID-1 better than RAID-4 for random write.

Chapters 39: File-System API

File Names

Three types of names:

- inode
- path
- file descriptor

Atomic File Update

Say we want to update file.txt.

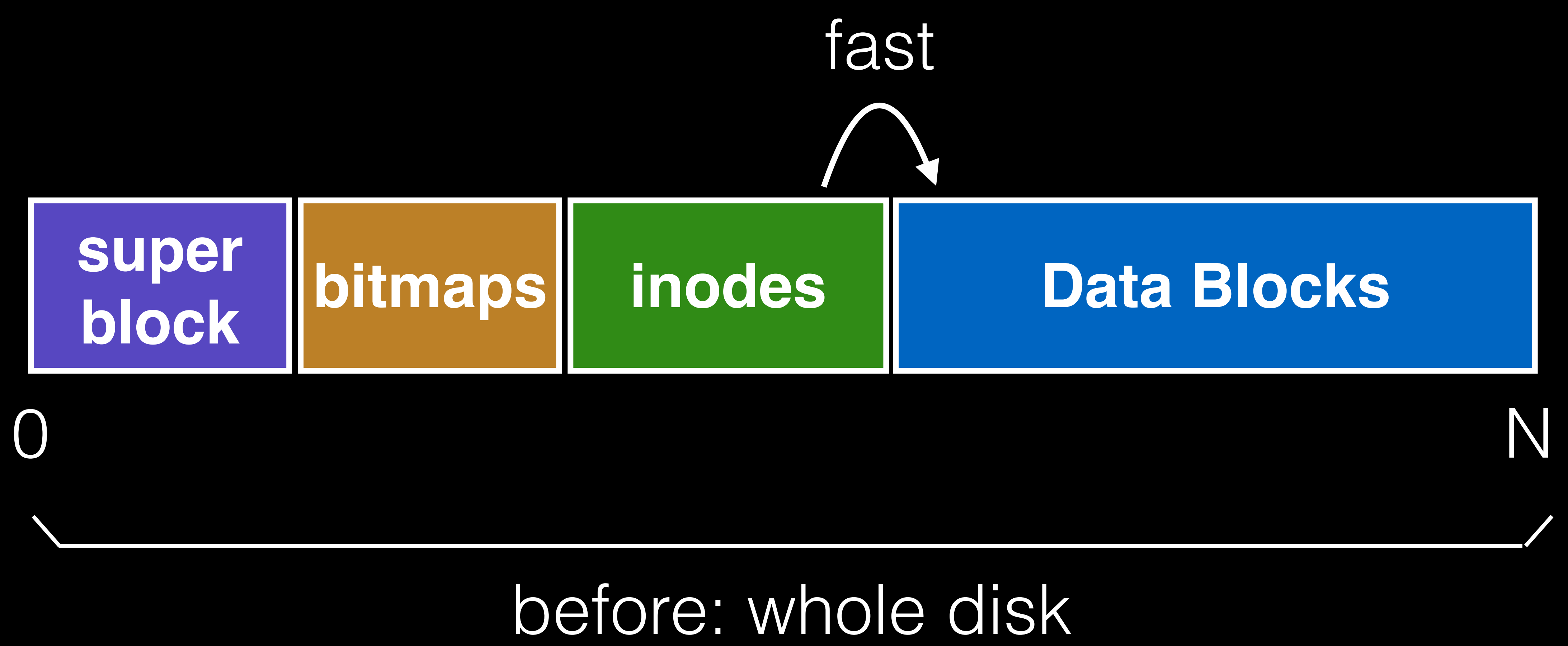
1. write new data to new file.txt.tmp file
2. fsync file.txt.tmp
3. rename file.txt.tmp over file.txt, replacing it

Chapters 41: FFS

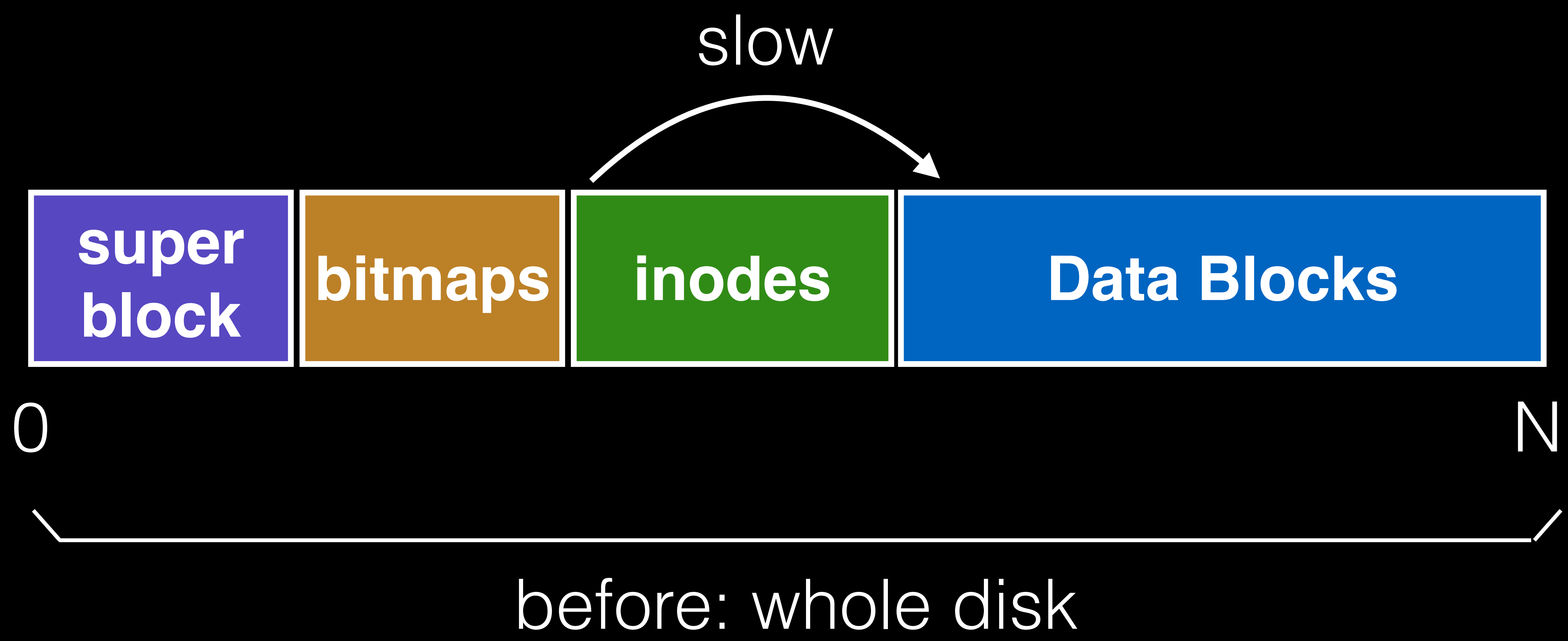
Treat a disk like a disk!

Place related data together: hopefully makes future reads faster.

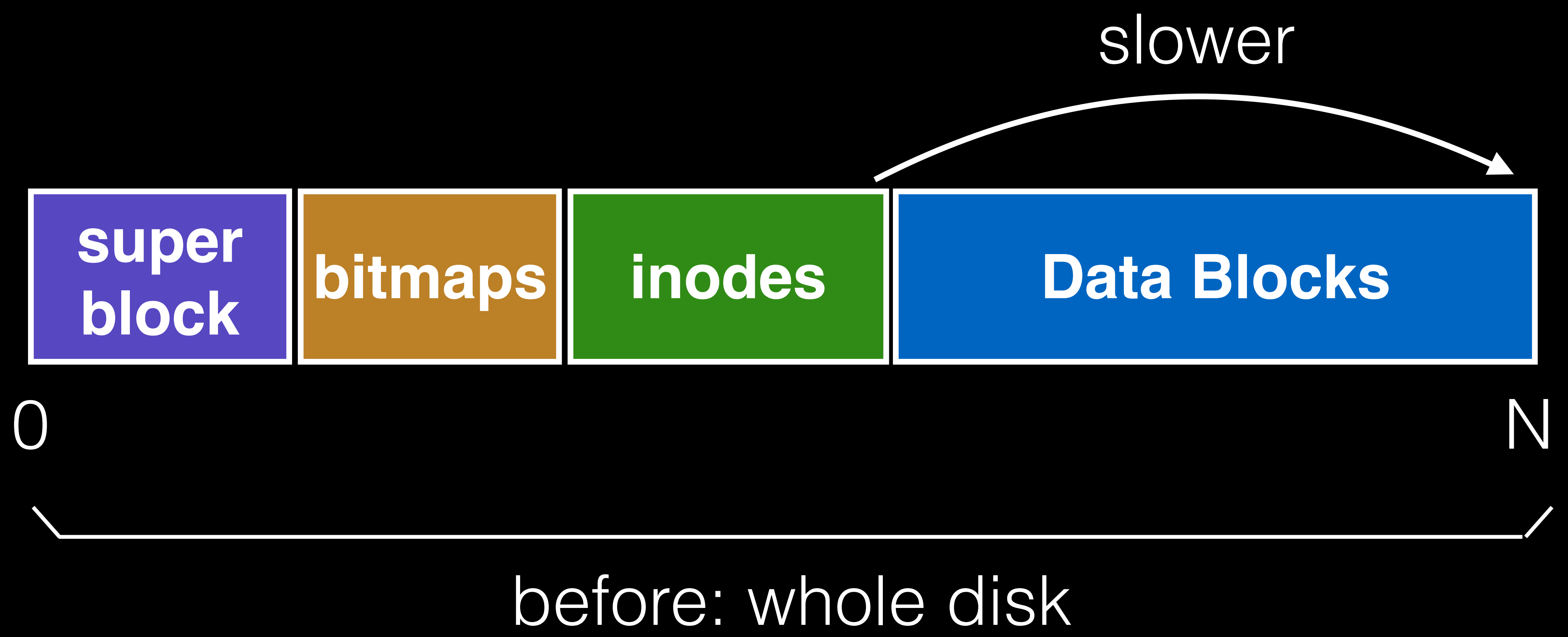
Technique 2: Groups



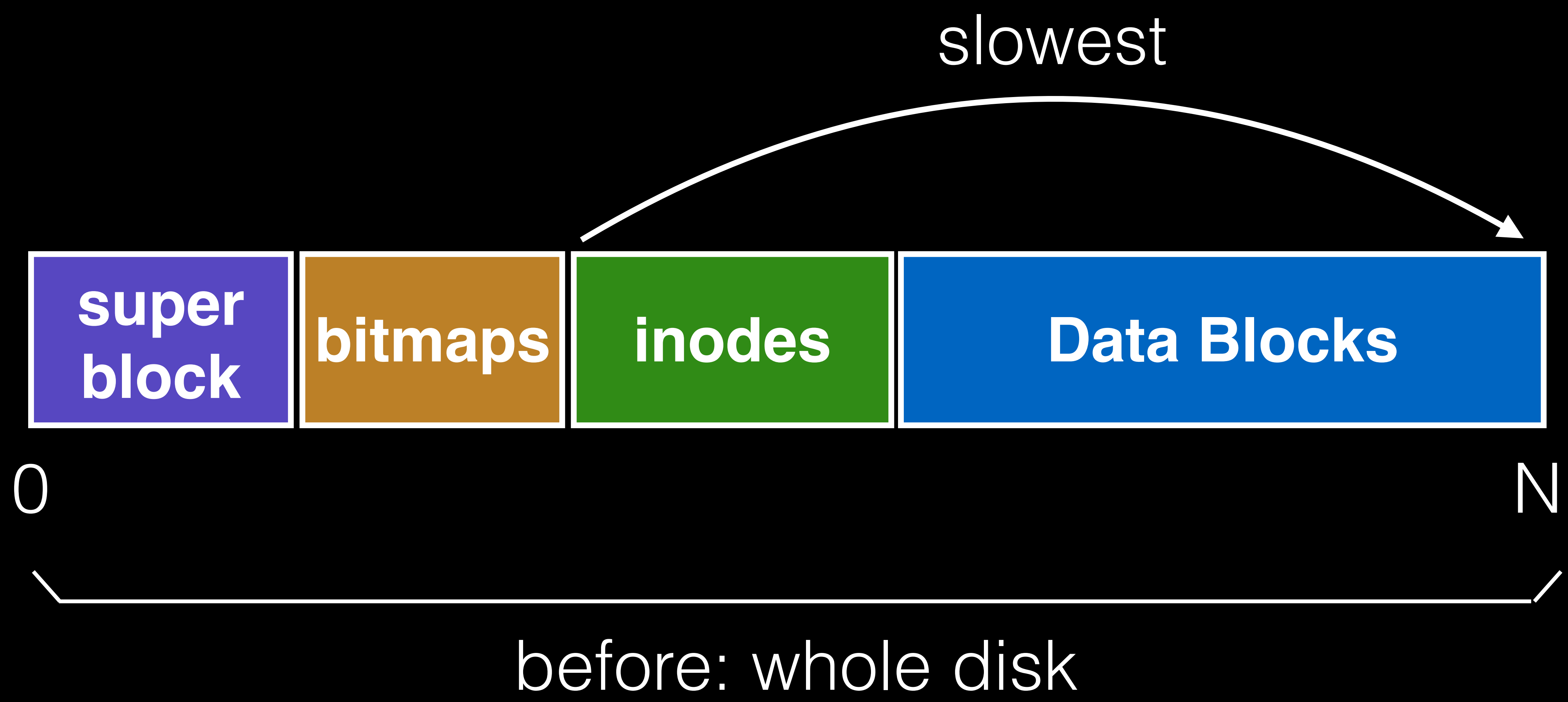
Technique 2: Groups



Technique 2: Groups

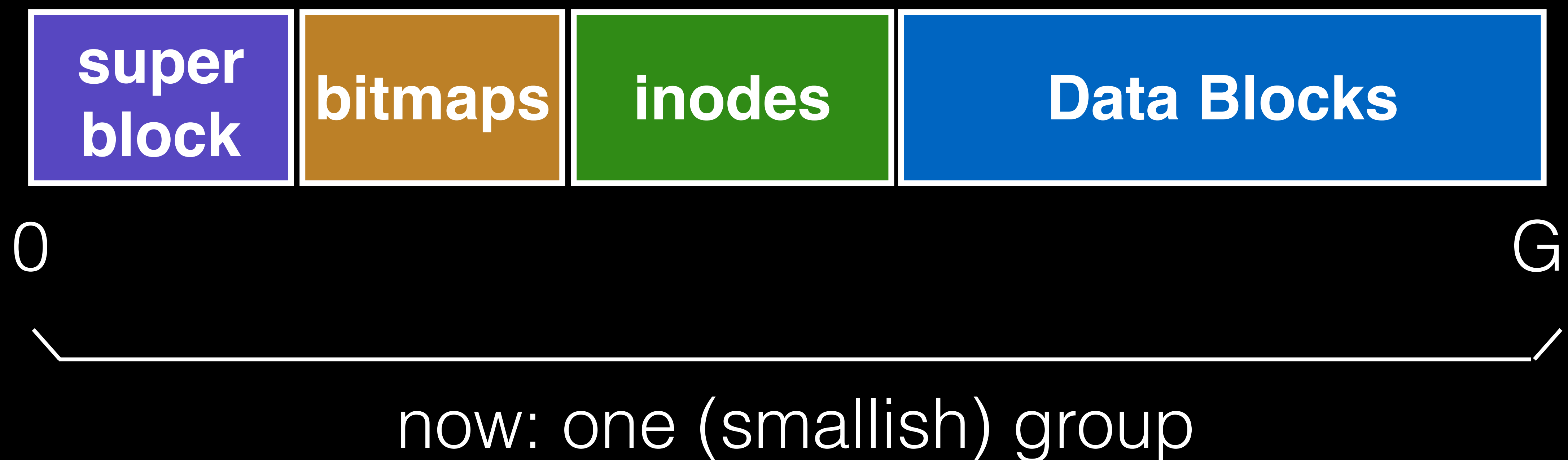


Technique 2: Groups



Technique 2: Groups

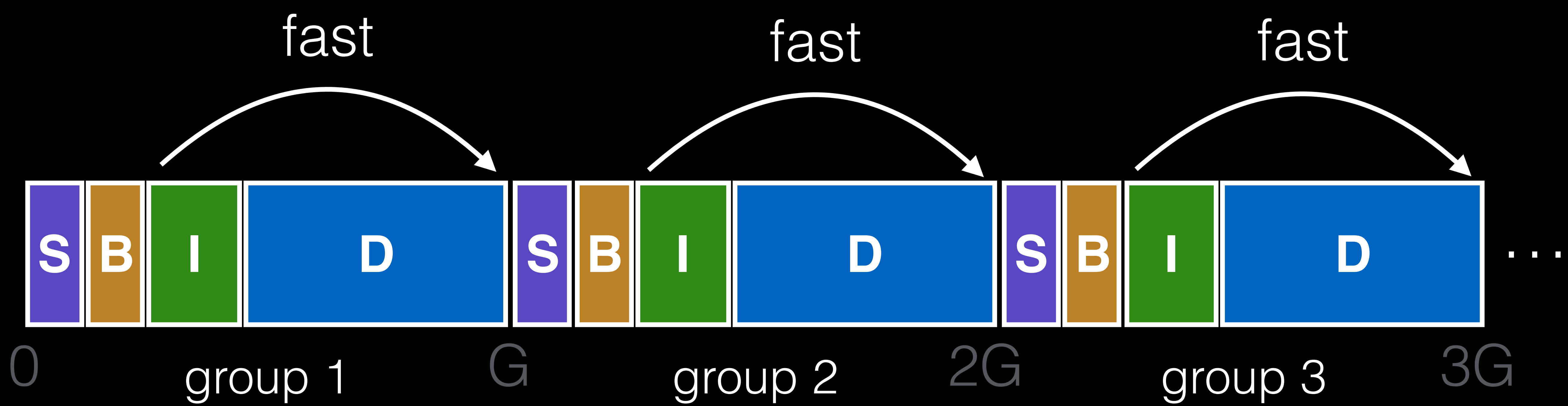
Technique 2: Groups



Technique 2: Groups

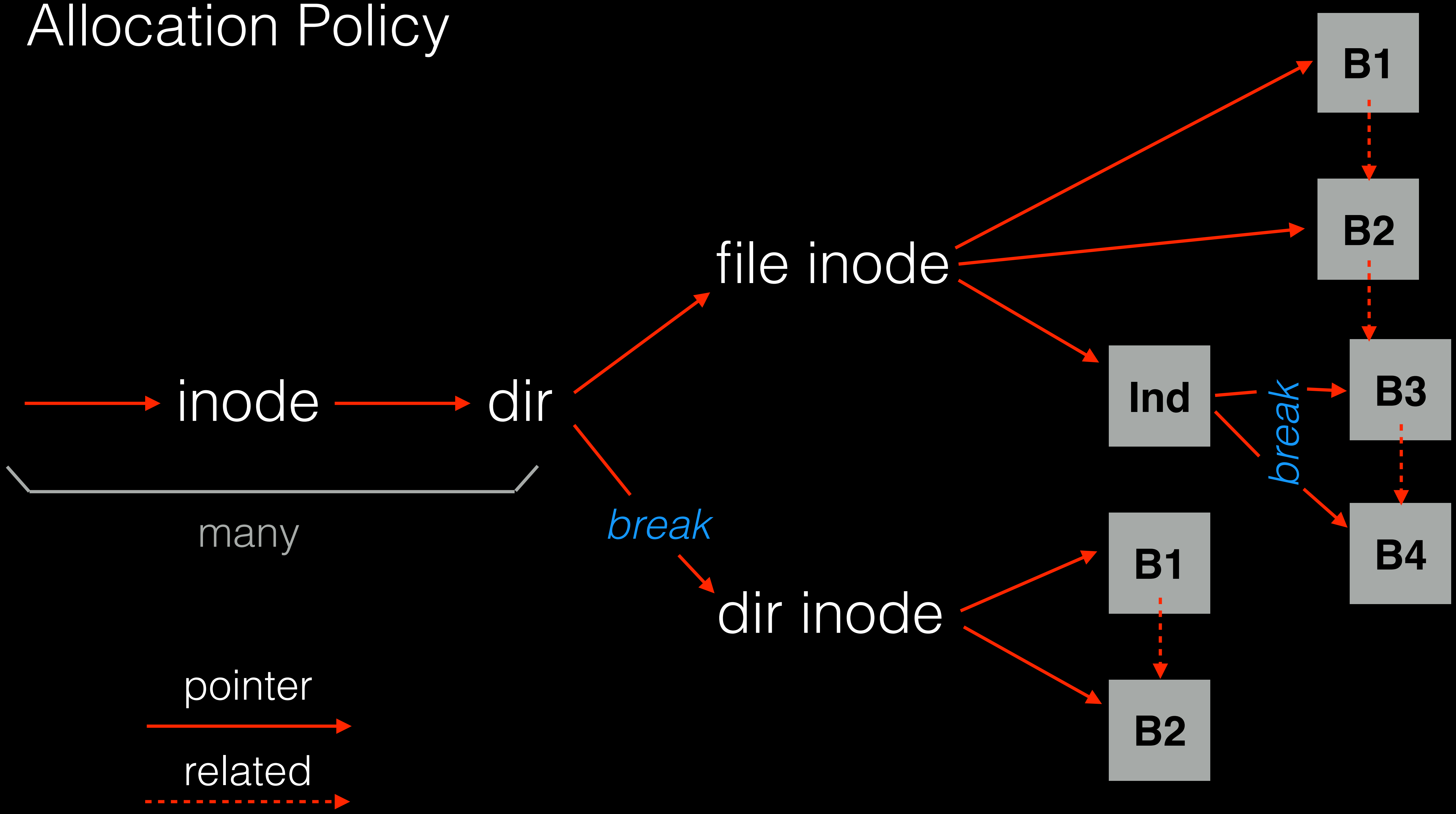
zoom out

Technique 2: Groups



strategy: allocate inodes and data blocks in same group.

Allocation Policy



Chapters 42: Journaling

Redundancy

Definition: if *A* and *B* are two pieces of data, and knowing *A* eliminates some or all the values *B* could *B*, there is <u>redundancy</u> between *A* and *B*.

RAID examples:
- mirrored disk (complete redundancy)
- parity blocks (partial redundancy)

Problem 3

Give 5 examples of redundancy in FFS (or files systems in general).

Problem 3

Give 5 examples of redundancy in FFS
(or files systems in general).

Dir entries AND inode table.
Dir entries AND inode link count.
Data bitmap AND inode pointers.
Data bitmap AND group descriptor.
Inode file size AND inode/indirect pointers.
…

fsck

FSCK = file system checker.

Strategy: after a crash, scan whole disk for contradictions.

For example, is a bitmap block correct?

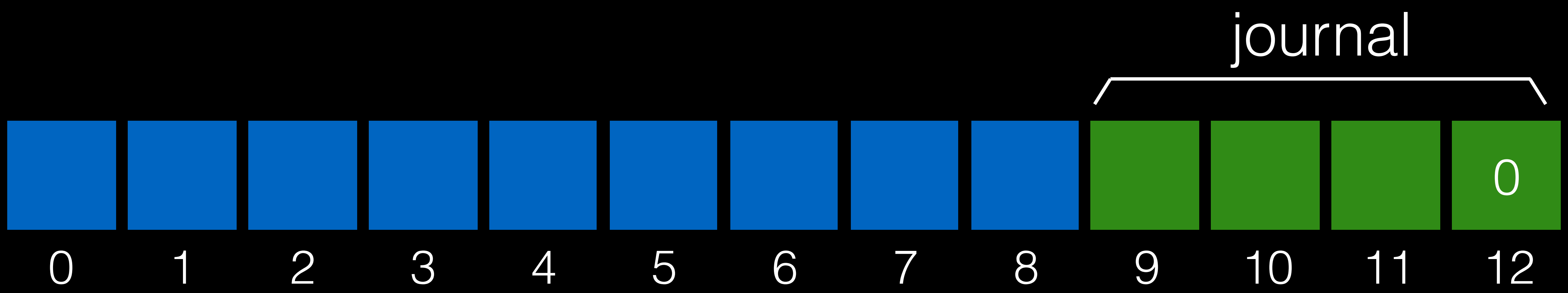
Read every valid inode+indirect. If an inode points to a block, the corresponding bit should be 1

Journal: General Strategy

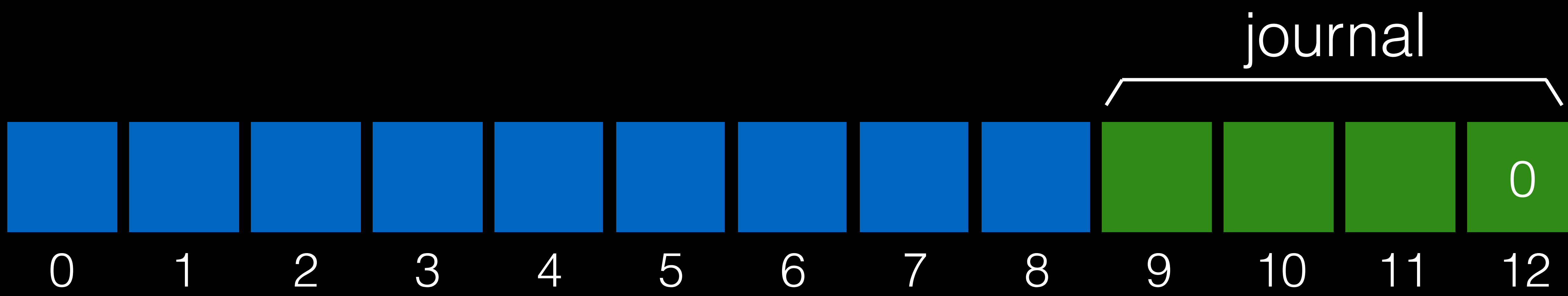
Never delete ANY old data, until,
ALL new data is safely on disk.

Ironically, this means we're adding redundancy to fix the problem caused by redundancy.

New Layout



New Layout

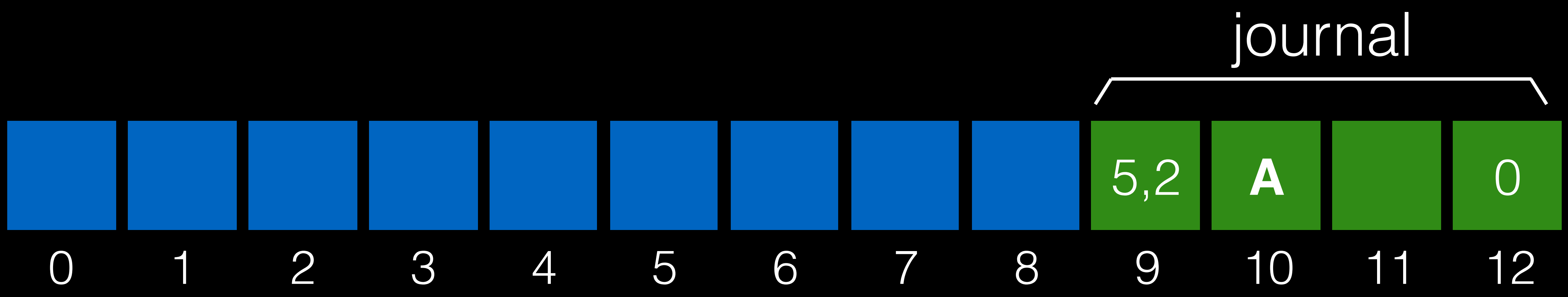


transaction: write A to block 5; write B to block 2

New Layout

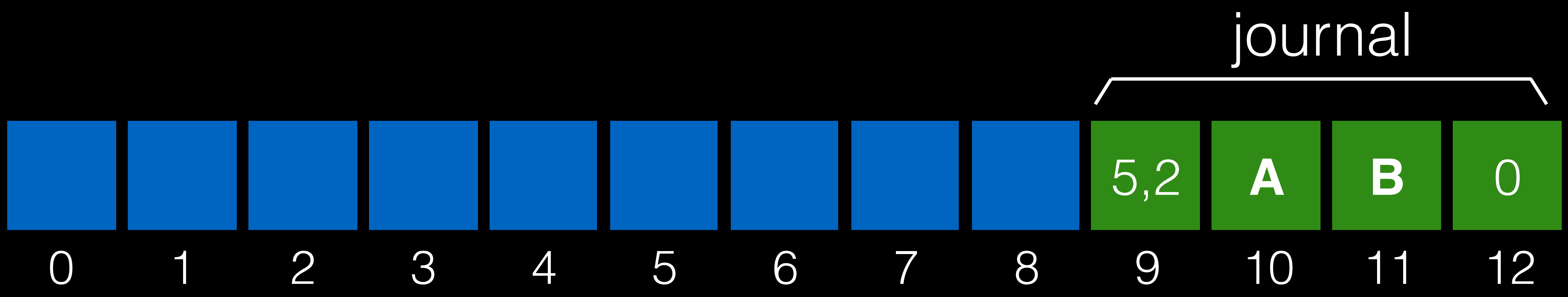
transaction: write A to block 5; write B to block 2

New Layout



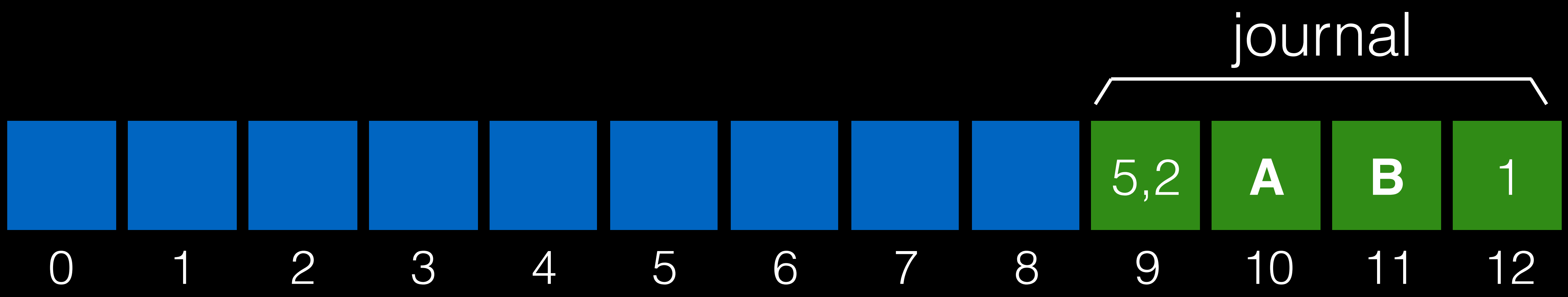
transaction: write A to block 5; write B to block 2

New Layout



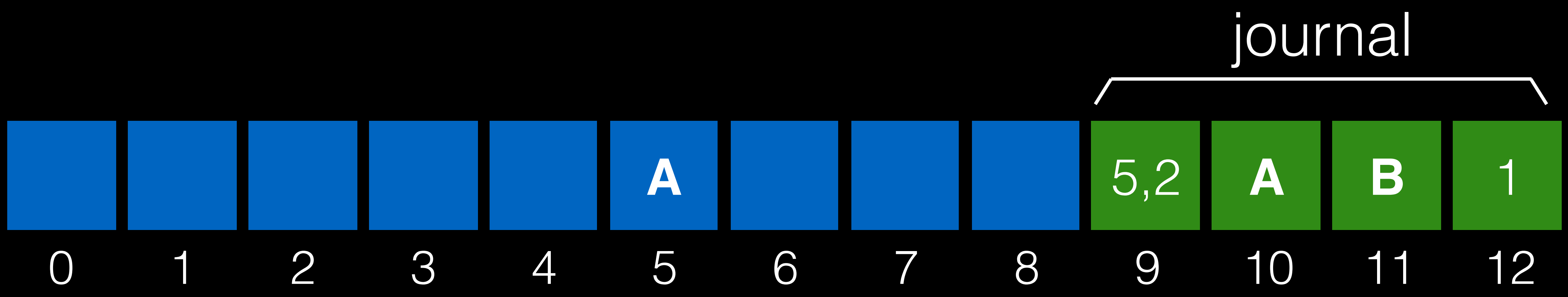
transaction: write A to block 5; write B to block 2

New Layout



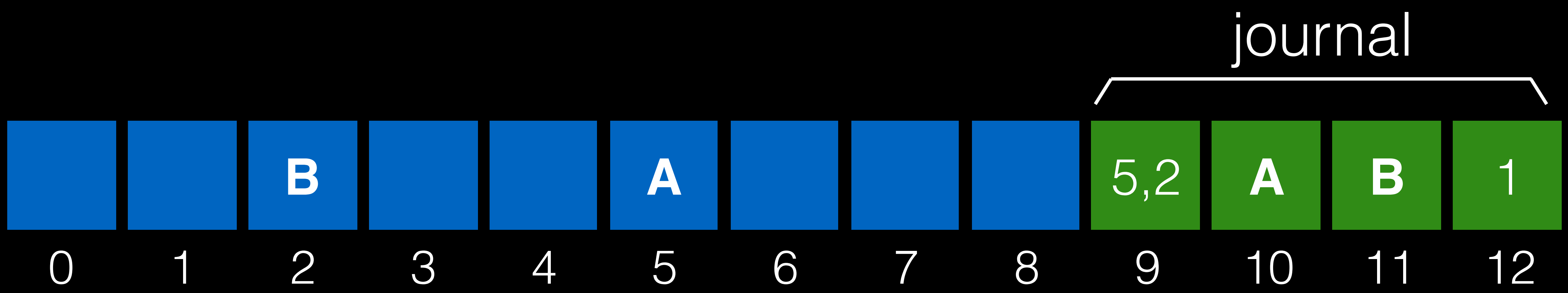
transaction: write A to block 5; write B to block 2

New Layout



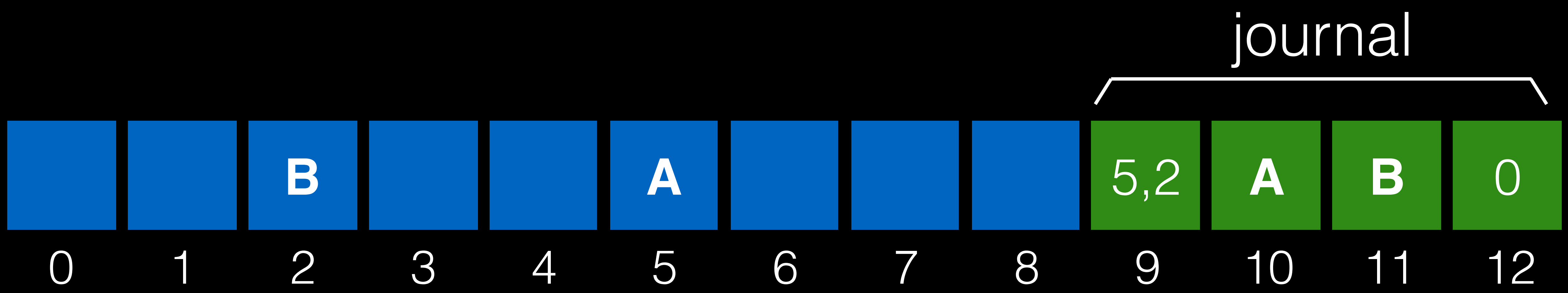
transaction: write A to block 5; write B to block 2

New Layout



transaction: write A to block 5; write B to block 2

New Layout



Optimizations

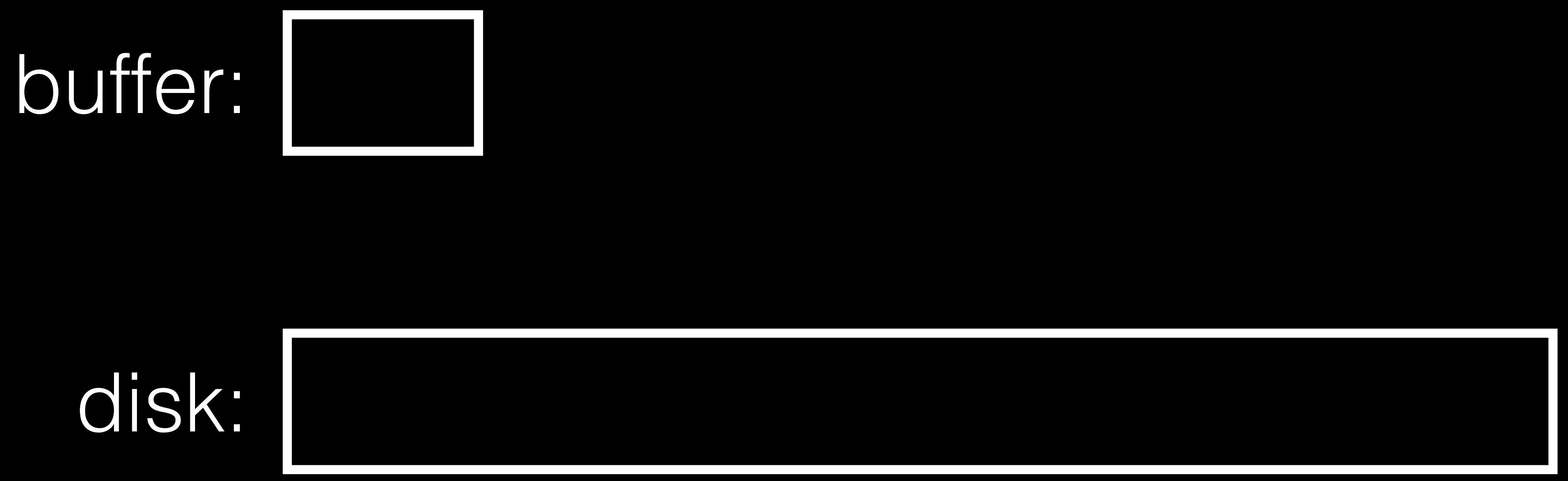
1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Logical journal

Chapters 43: LFS

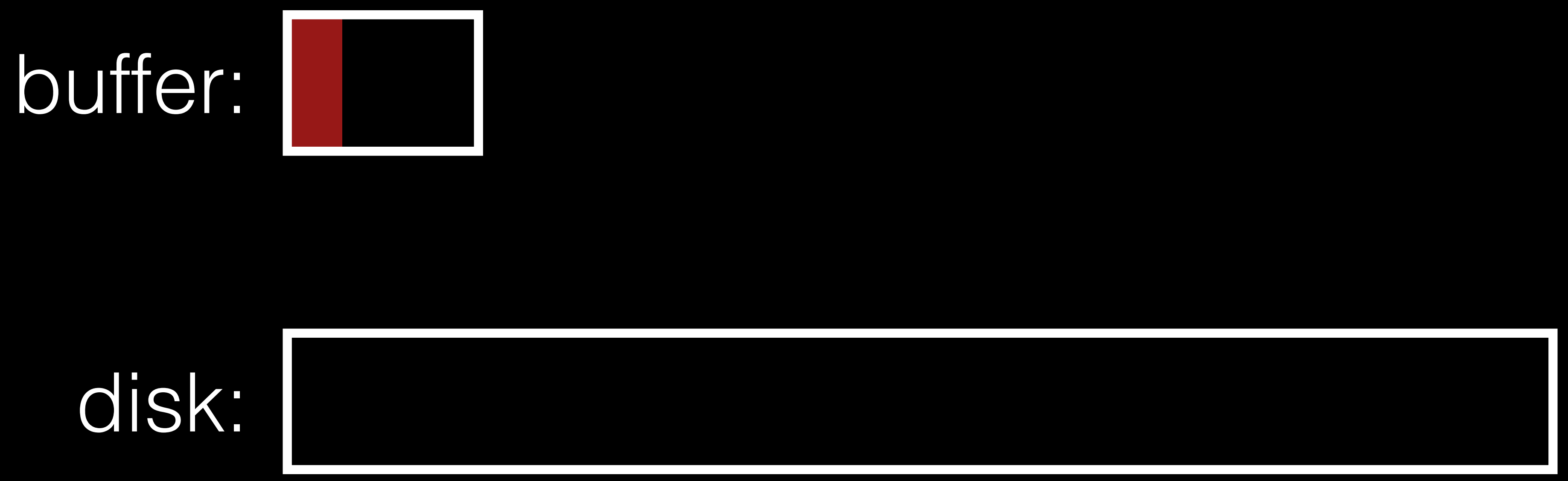
Write data fastest way possible… Sequentially!

Reads may be slower later (scattered).

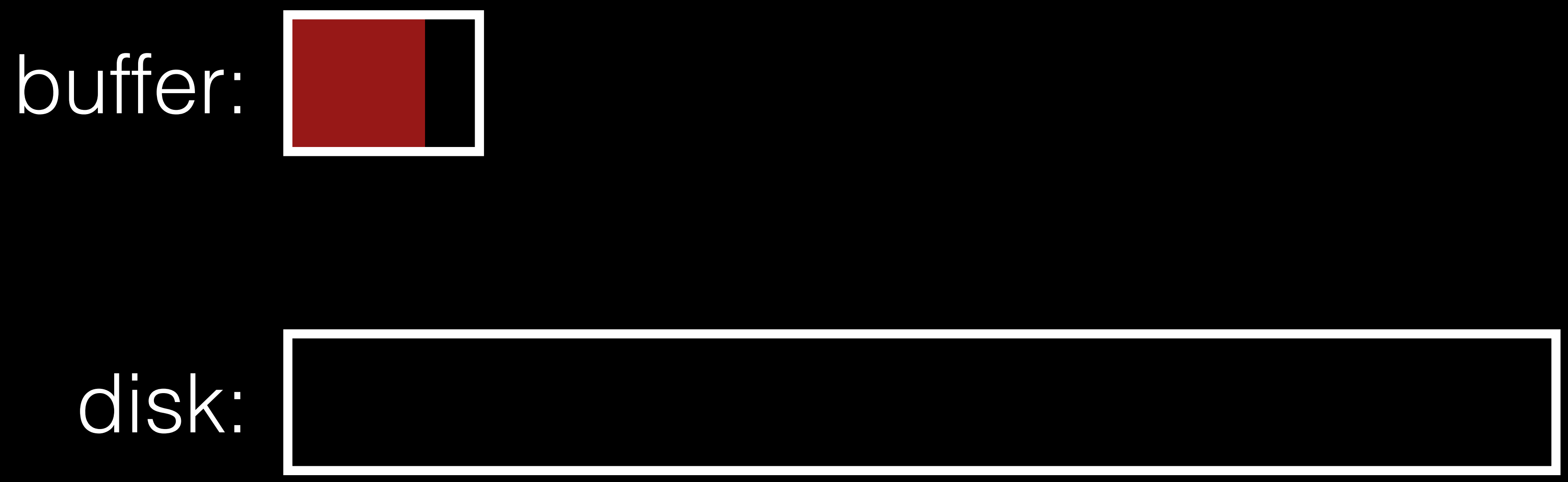
Big Picture



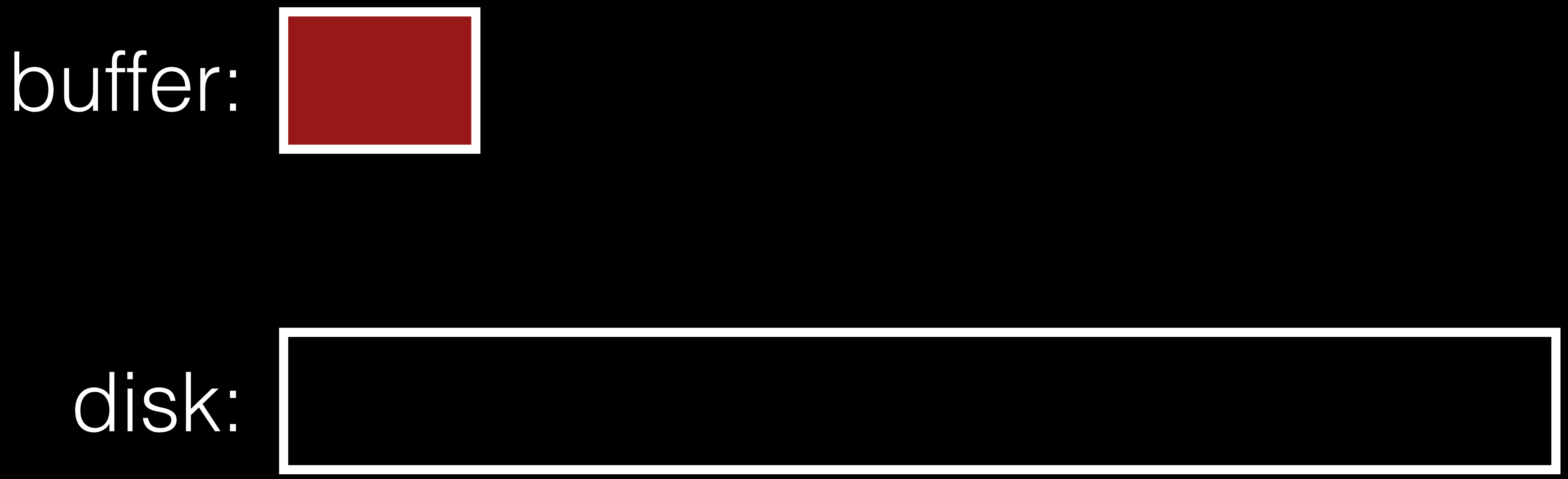
Big Picture



Big Picture

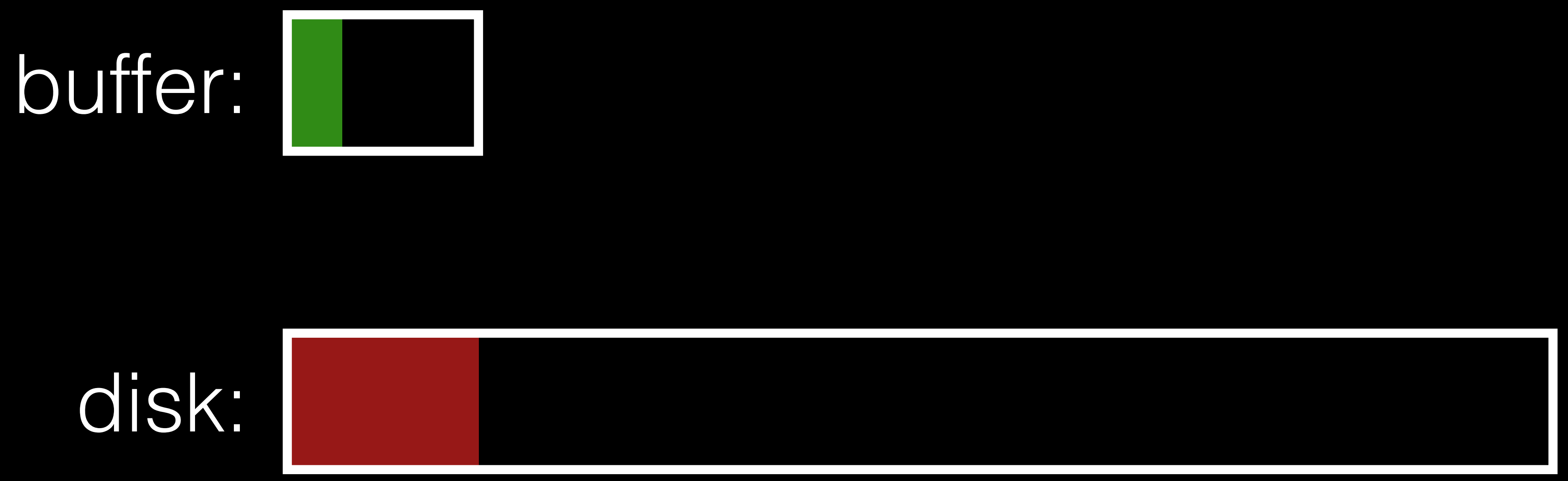


Big Picture

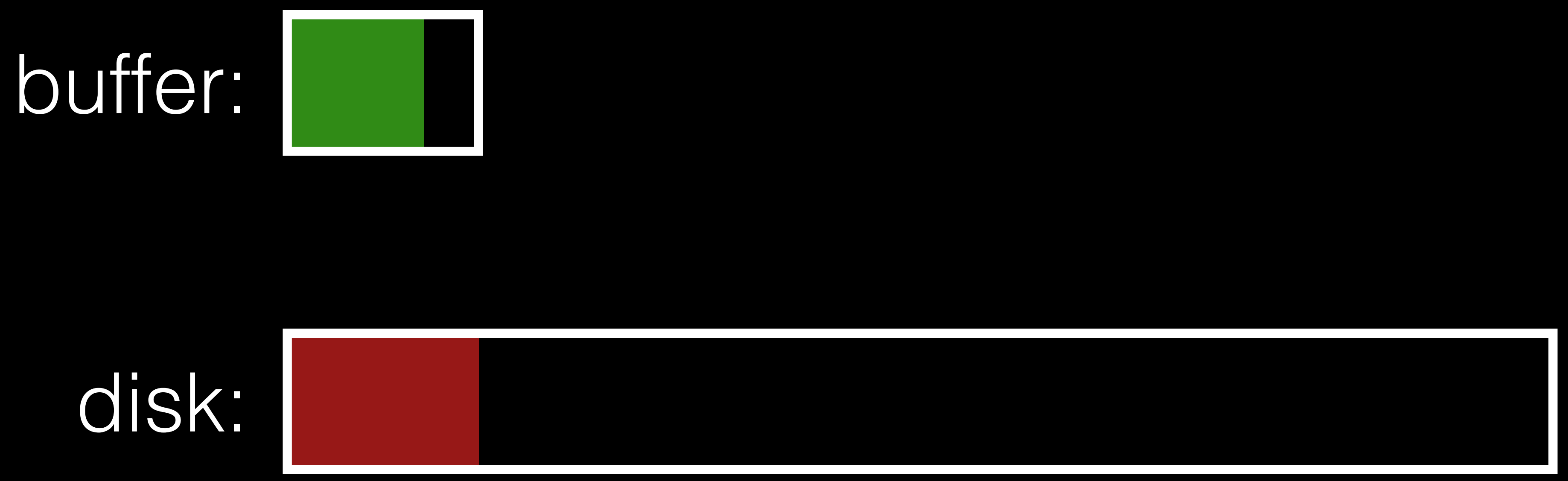


Big Picture

Big Picture



Big Picture



Big Picture

Big Picture

Big Picture

Big Picture

Inode Numbers

Problem: for every data update, we need to do updates all the way up the tree.

Why? We change inode number when we copy it.

Solution: keep inode numbers constant. Don't base on offset.

Before we found inodes with math. How now?

Data Structures (attempt 2)

What can we get rid of from FFS?
- allocation structs: data + inode bitmaps

Inodes are no longer at fixed offset.
- use imap struct to map number => inode.

Garbage Collection

Is data alive? Use segment summary.

How to clean? Copy clean data out of M segments into N new segments (N < M).

Which segments to clean? Cold, invalid, etc.

Chapters 44: Integrity

Checksums…

Chapters 47: Distributed Systems

Channels

UDP: unreliable

TCP: reliable
 - seq numbers, buffering, retry

RPC

Machine A

```
int main(…) {

}
```

Machine B

```
int foo(char *msg) {
    …
}
```

RPC

Machine A

```
int main(…) {
    int x = foo();
}
```

Machine B

```
int foo(char *msg) {
    …
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}
```

Machine B

```
int foo(char *msg) {
    …
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

Actual calls.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

What it feels like for programmer.

RPC

Machine A

```
int main(...) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

client wrapper

Machine B

```
int foo(char *msg) {
    ...
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

server wrapper

Wrappers.

RPC Tools

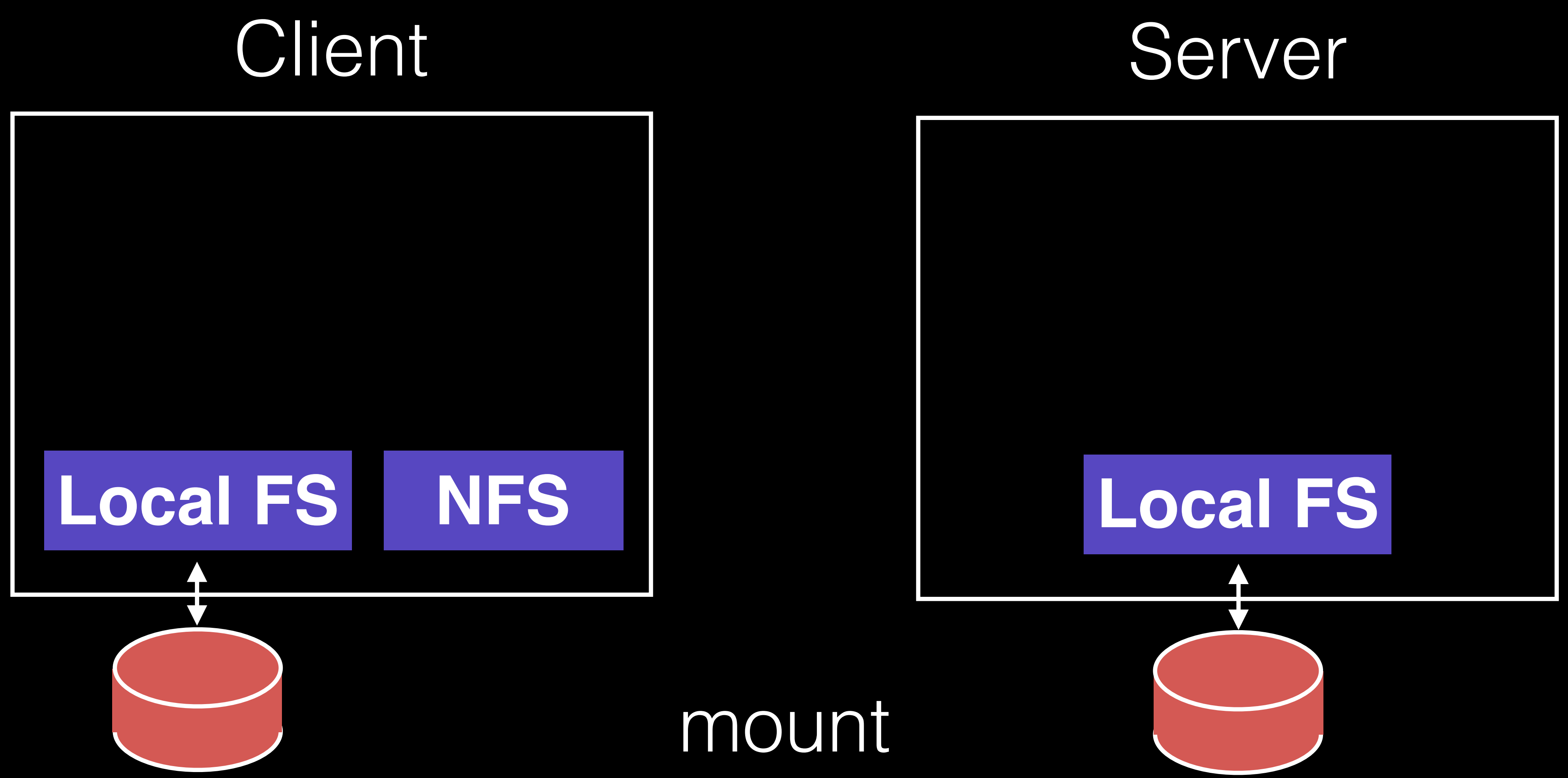
RPC packages help with this with two components.

(1) Stub generation
- create wrappers automatically

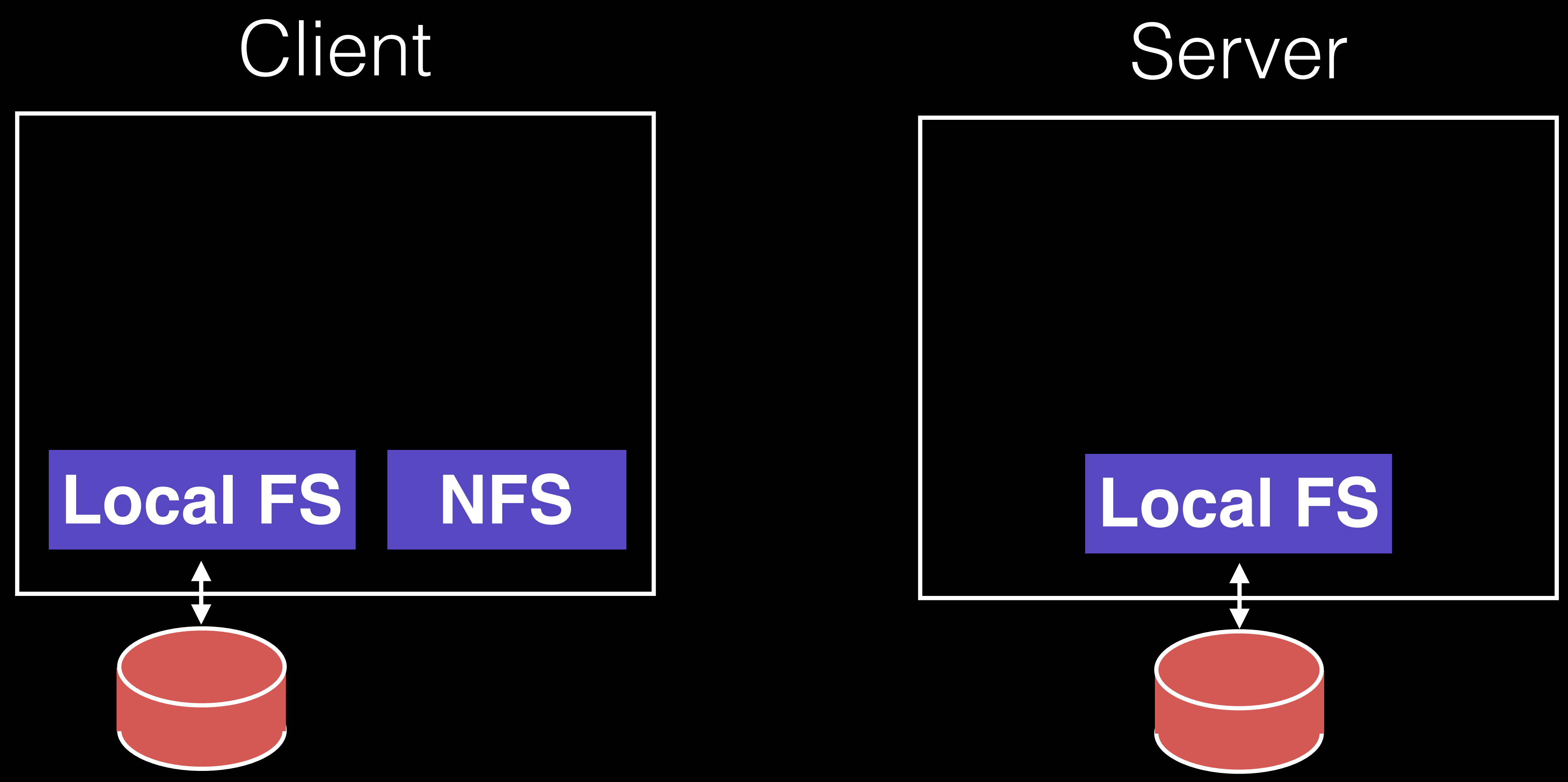
(2) Runtime library
- thread pool
- socket listeners call functions on server

Chapters 48: NFS

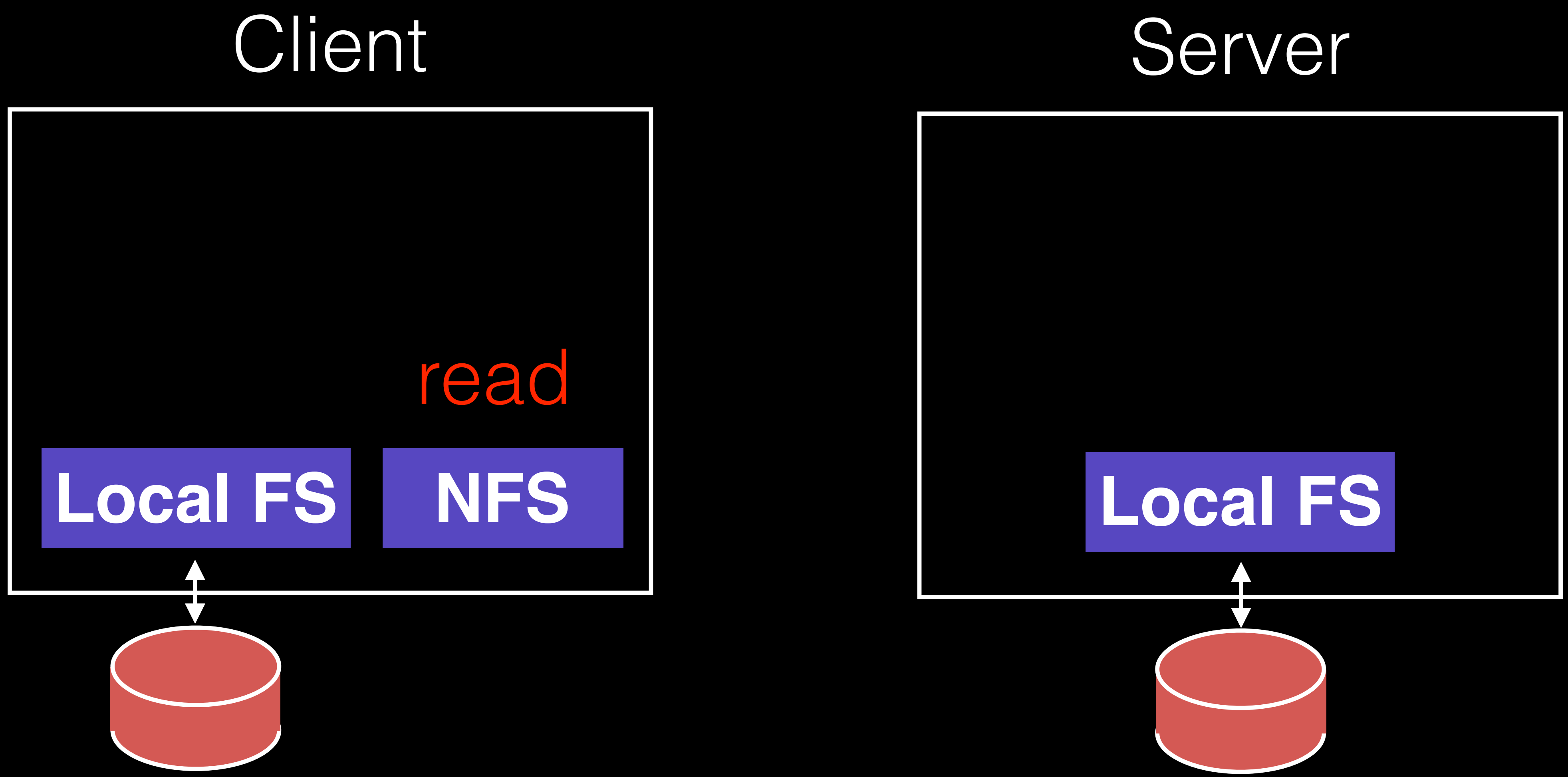
General Strategy: Export FS



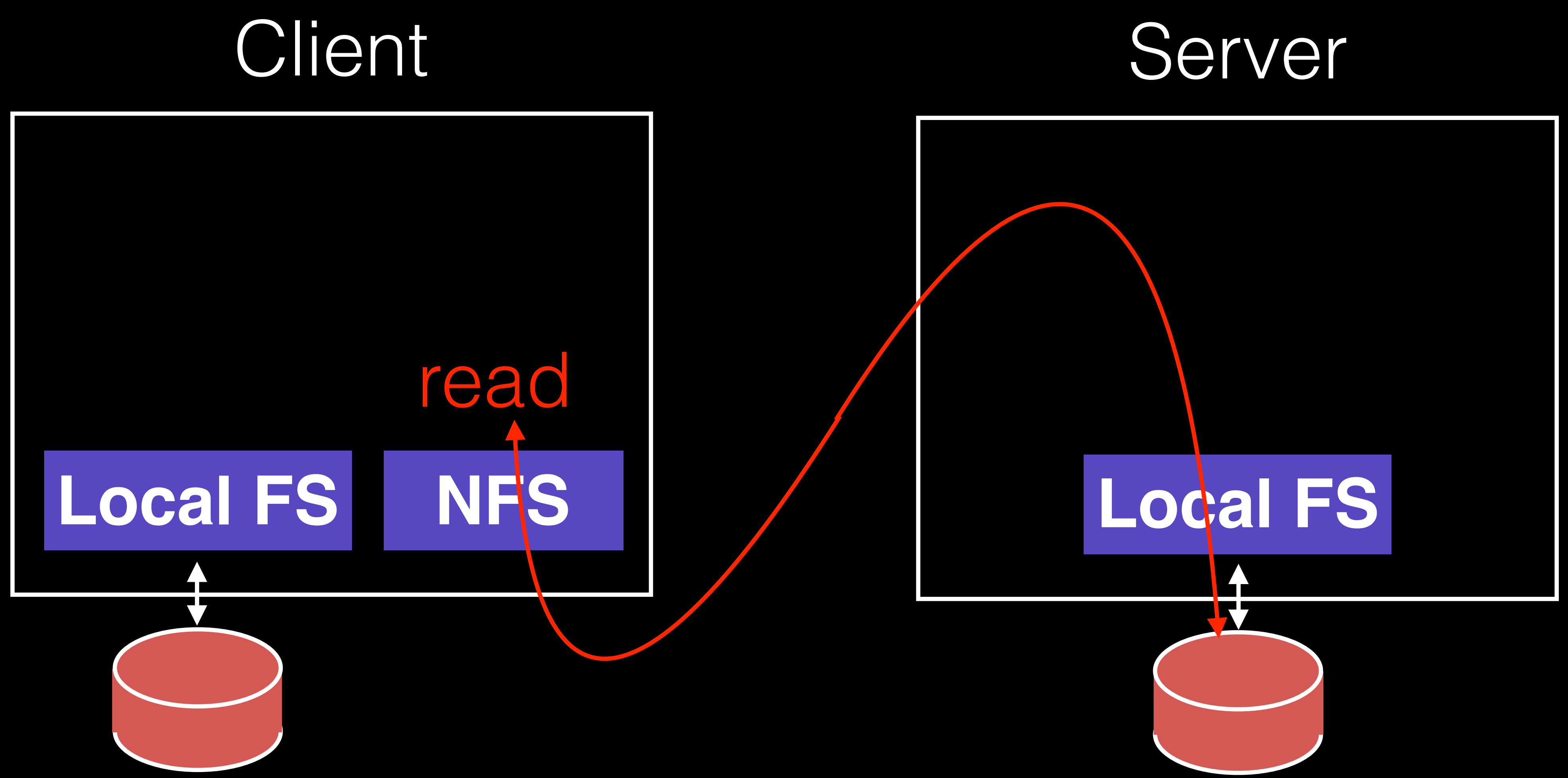
General Strategy: Export FS



General Strategy: Export FS



General Strategy: Export FS



Stateless

Requests understandable without any context about clients.

No fds!

Idempotent

Design API so that there is no harm is executing a call more than once.

An API call that has this is “idempotent”. If f() is idempotent, then:
f() has the same effect as f(); f(); … f(); f()

Cache Consistency

Know update visibility, stale cache.

Chapters 49: AFS

AFS Goals

Primary goal: scalability! (many clients per server)

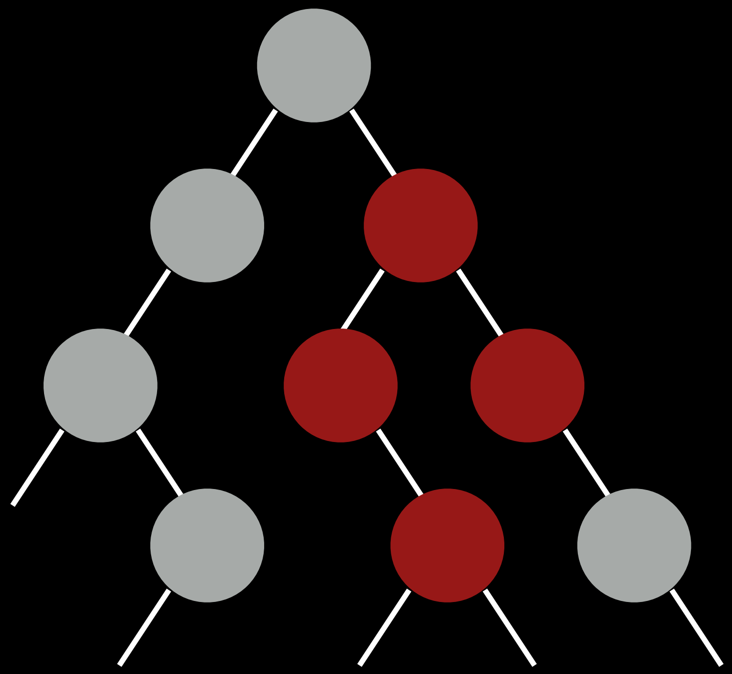
More reasonable semantics for concurrent file access.

Not good about handling some failure scenarios.

AFS Design

NFS: export local FS

AFS: present big file tree, store across many machines.



Break tree into "volumes."
I.e., partial sub trees.

Update Visibility

Clients updates not seen on servers yet.

AFS solution:
- flush on close
- buffer whole files on local disk

Concurrent writes? Last writer (i.e., closer) wins.

Never get mixed data.

Stale Cache

AFS solution: tell clients when data is overwritten.

When clients cache data, ask for “callback” from server.

No longer stateless!

Callbacks

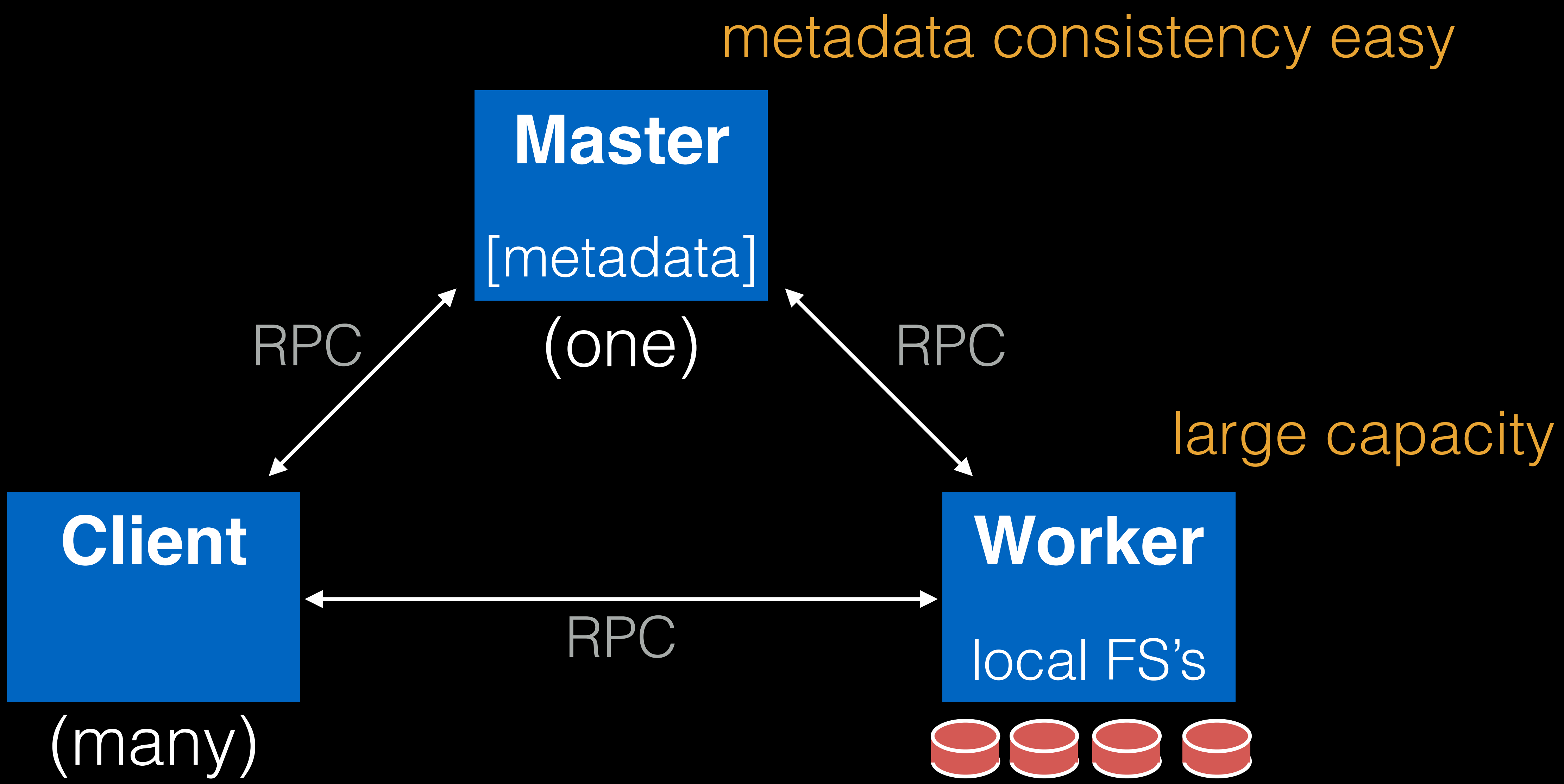
What if client crashes?

What if server runs out of memory?

What if server crashes?

GFS

Architecture



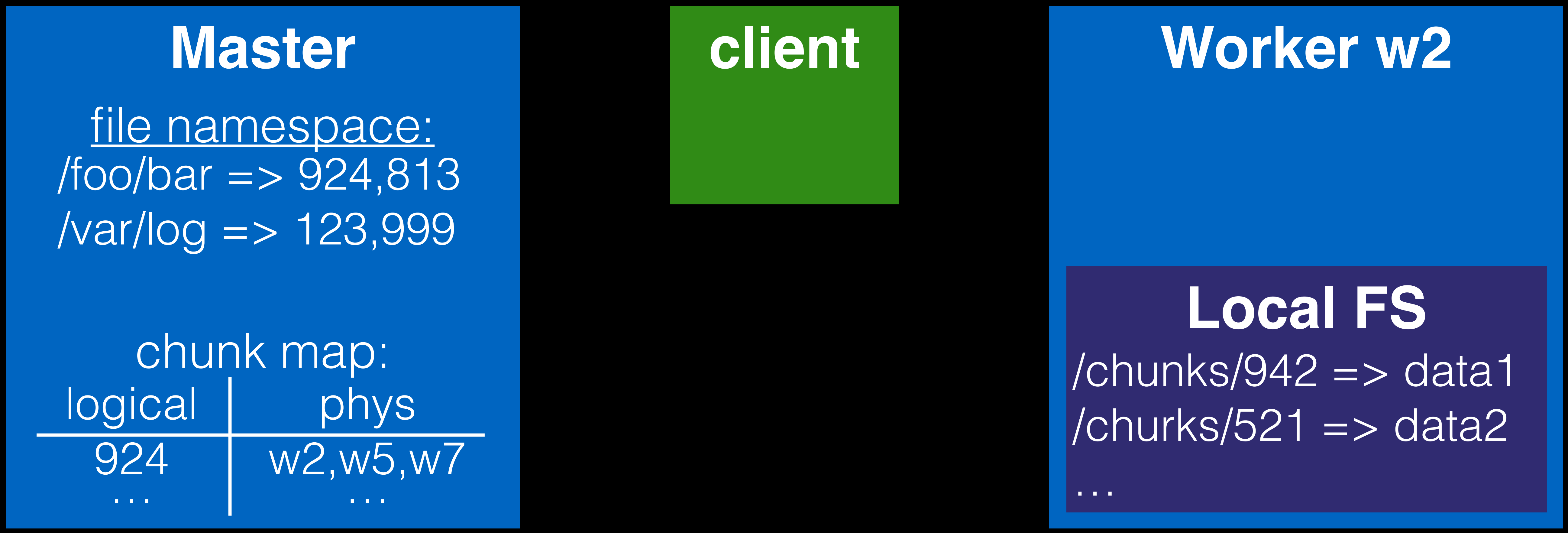
Chunk Layer

Break GFS files into large chunks (e.g., 64MB).

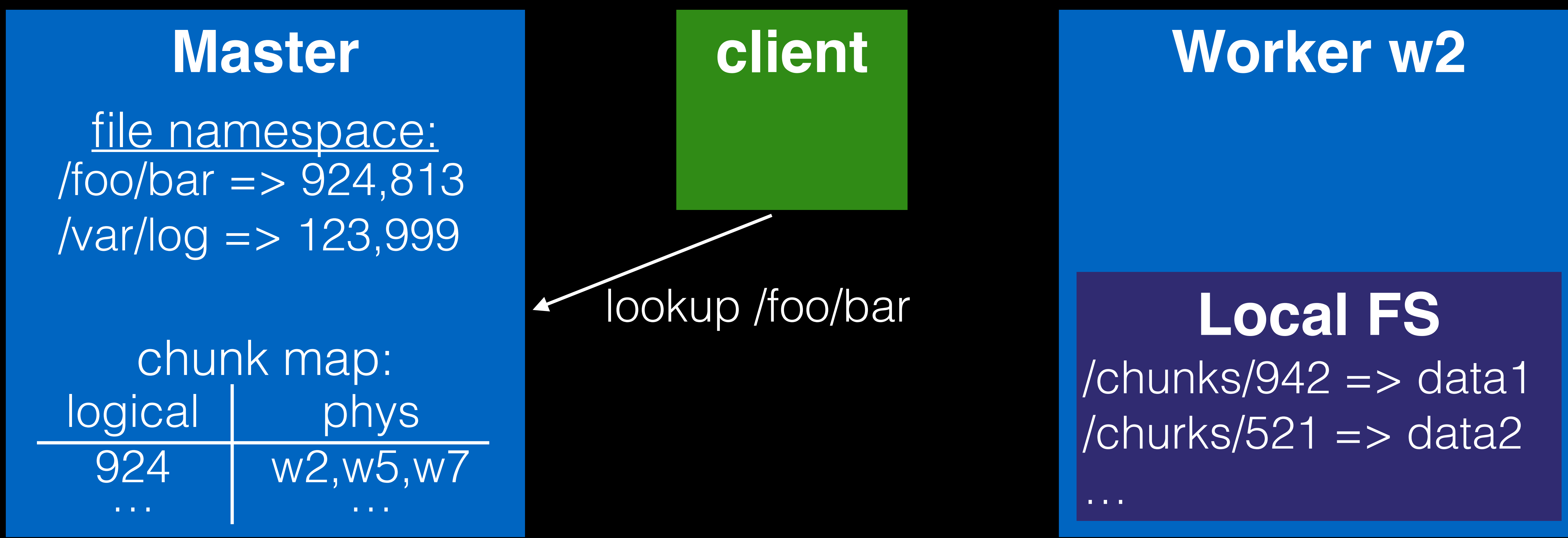
Workers store physical chunks in Linux files.

Master maps logical chunk to physical chunk locations.

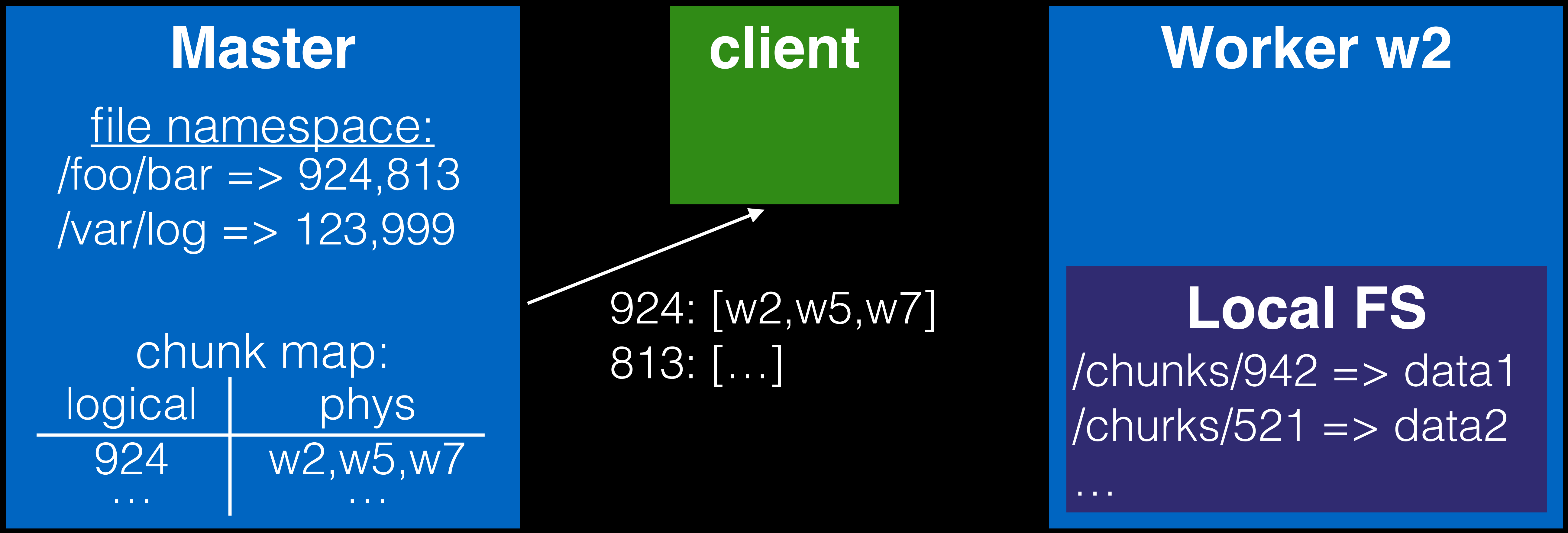
File Namespace



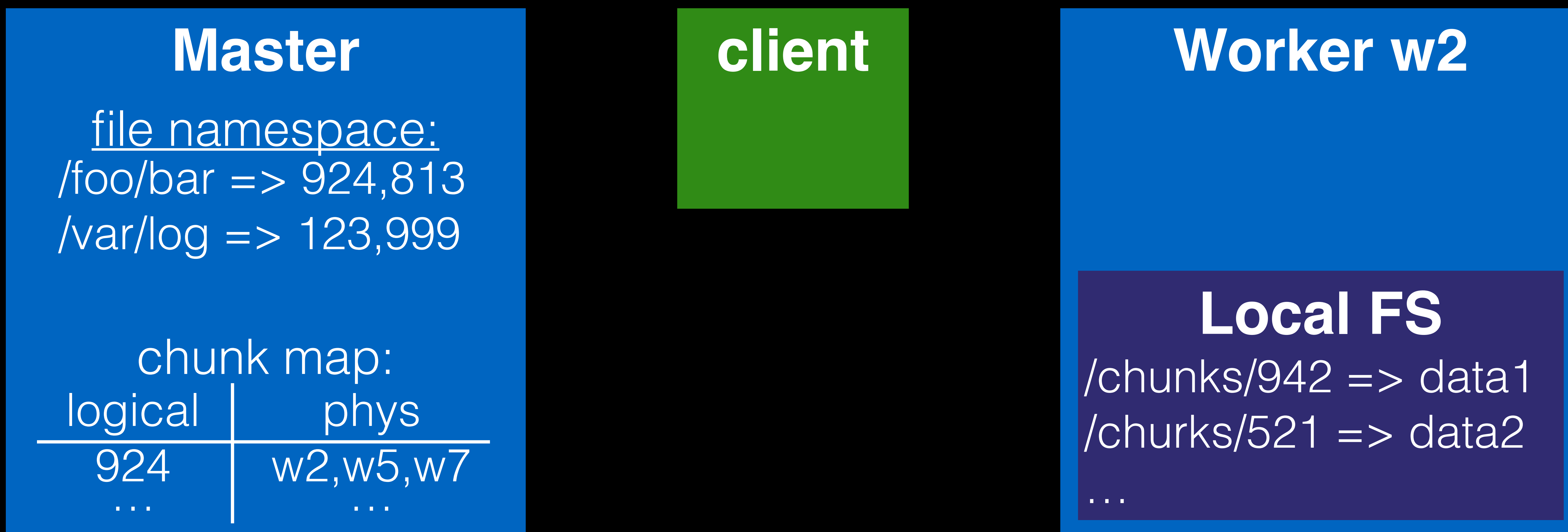
File Namespace



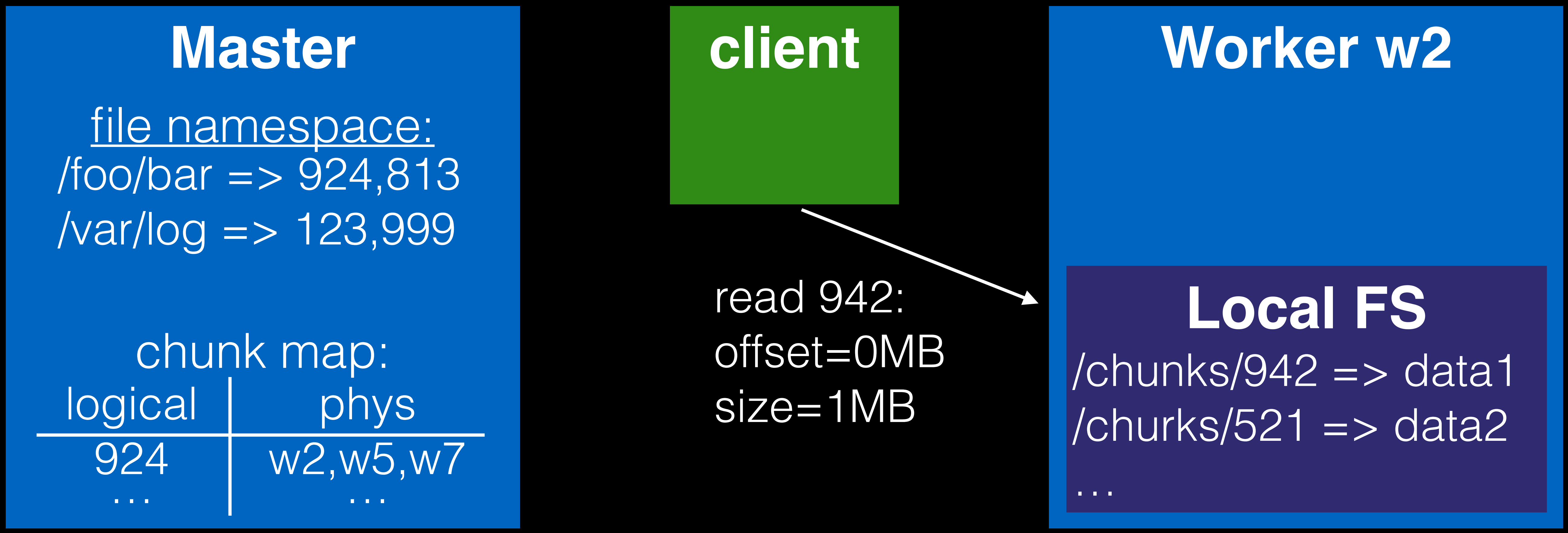
File Namespace



File Namespace



File Namespace



Master: Crashes + Consistency

File namespace and chunk map are 100% in RAM.
- allows master to work with 1000's of workers
- what master crashes?

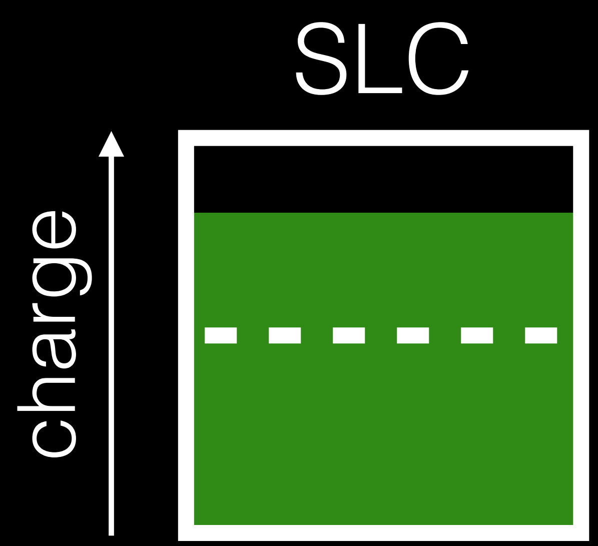
MapReduce

```
public void map(LongWritable key, Text value) {
   String line = value.toString();
   StringToke st = new StringToke(line);
   while (st.hasMoreTokens())
      output.collect(st.nextToken(), 1);
}

public void reduce(Text key,
                   Iterator<IntWritable> values) {
  int sum = 0;
  while (values.hasNext())
    sum += values.next().get();
  output.collect(key, sum);
}
```

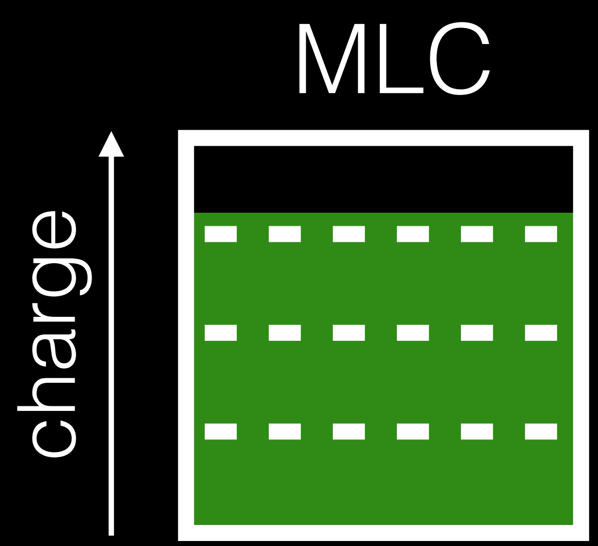
what does this do?

Flash

Single- vs. Multi- Level Cell



expensive
robust



cheap
sensitive

Wearout

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

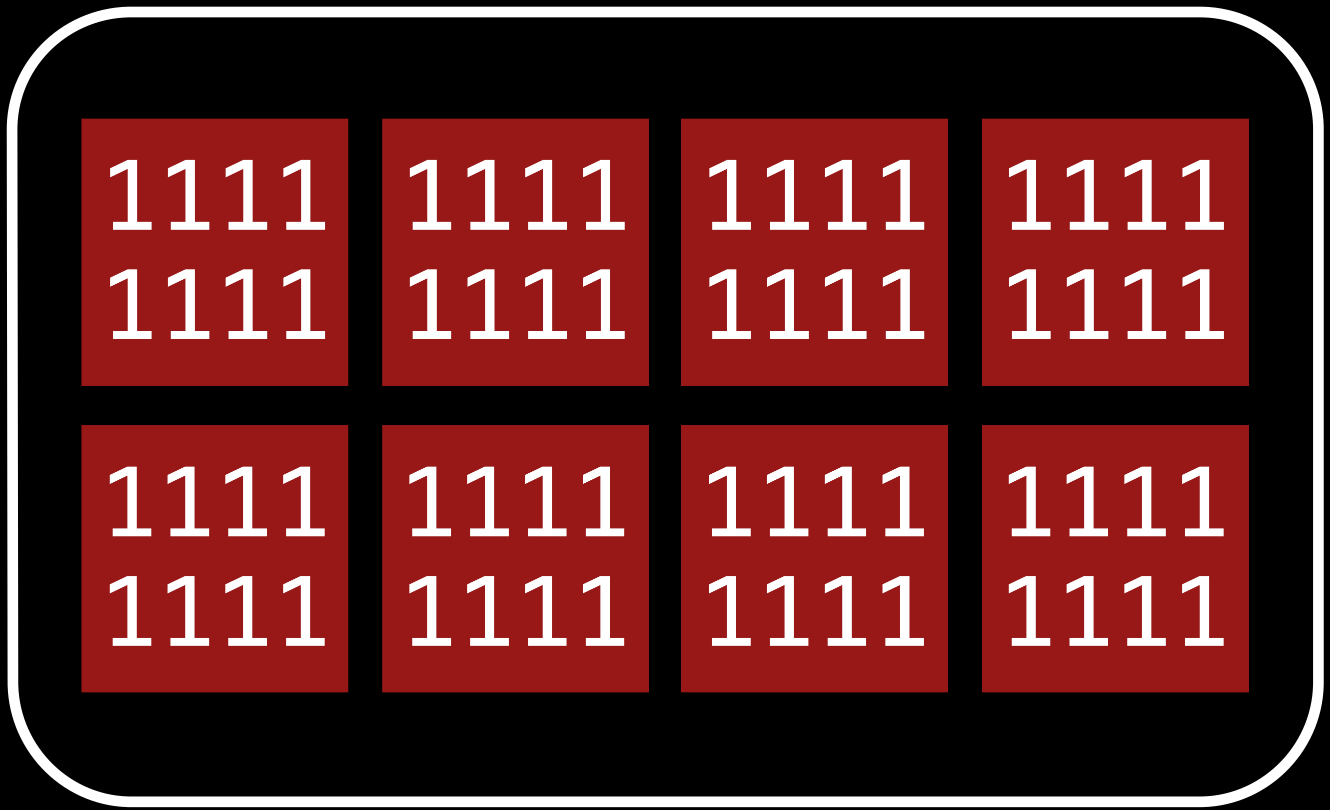
MLC: ~10K times
SLC: ~100K times

Usage strategy: wear leveling.
- prevents some cells from wearing out while others still fresh.

Block


1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111

Block



one block

Block


1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
1111
one page

Flash Hierarchy

Plane: 1024 to 4096 blocks
- planes accessed in parallel

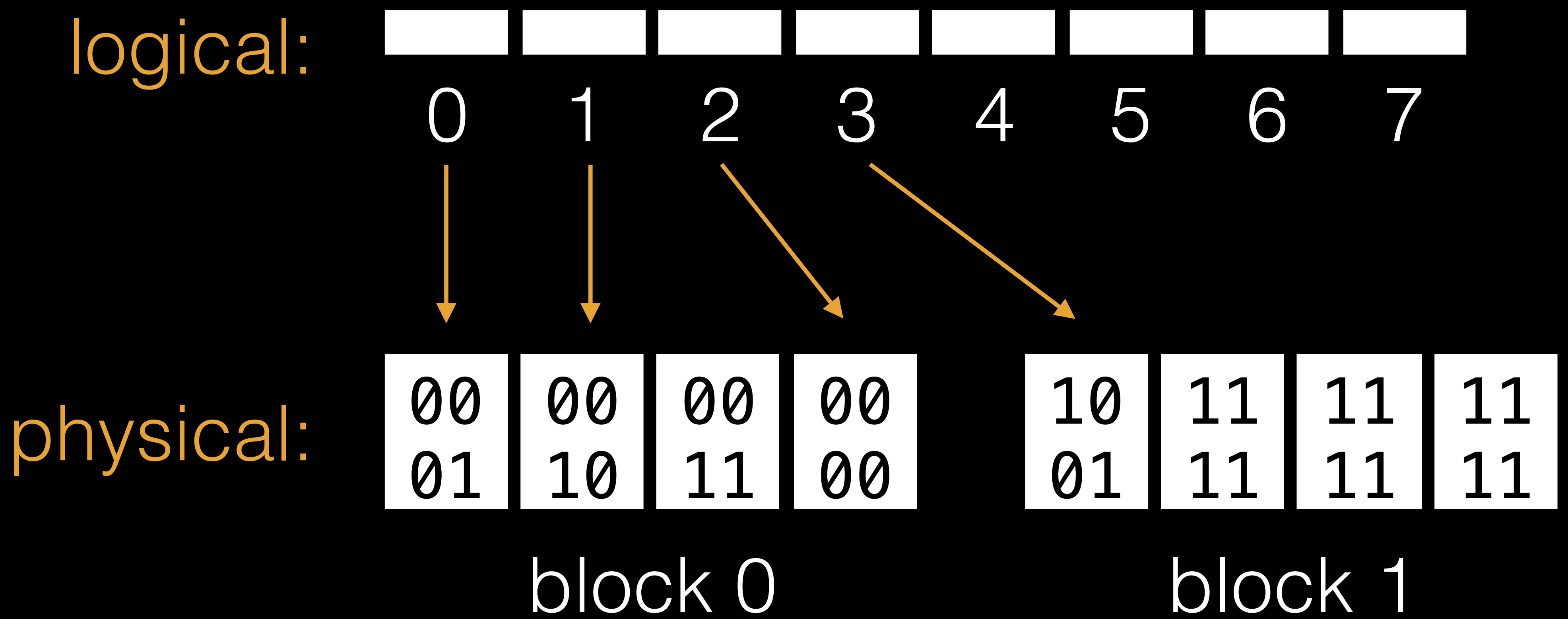
Block: 64 to 256 pages
- unit of erase

Page: 2 to 8 KB
- unit of read and program

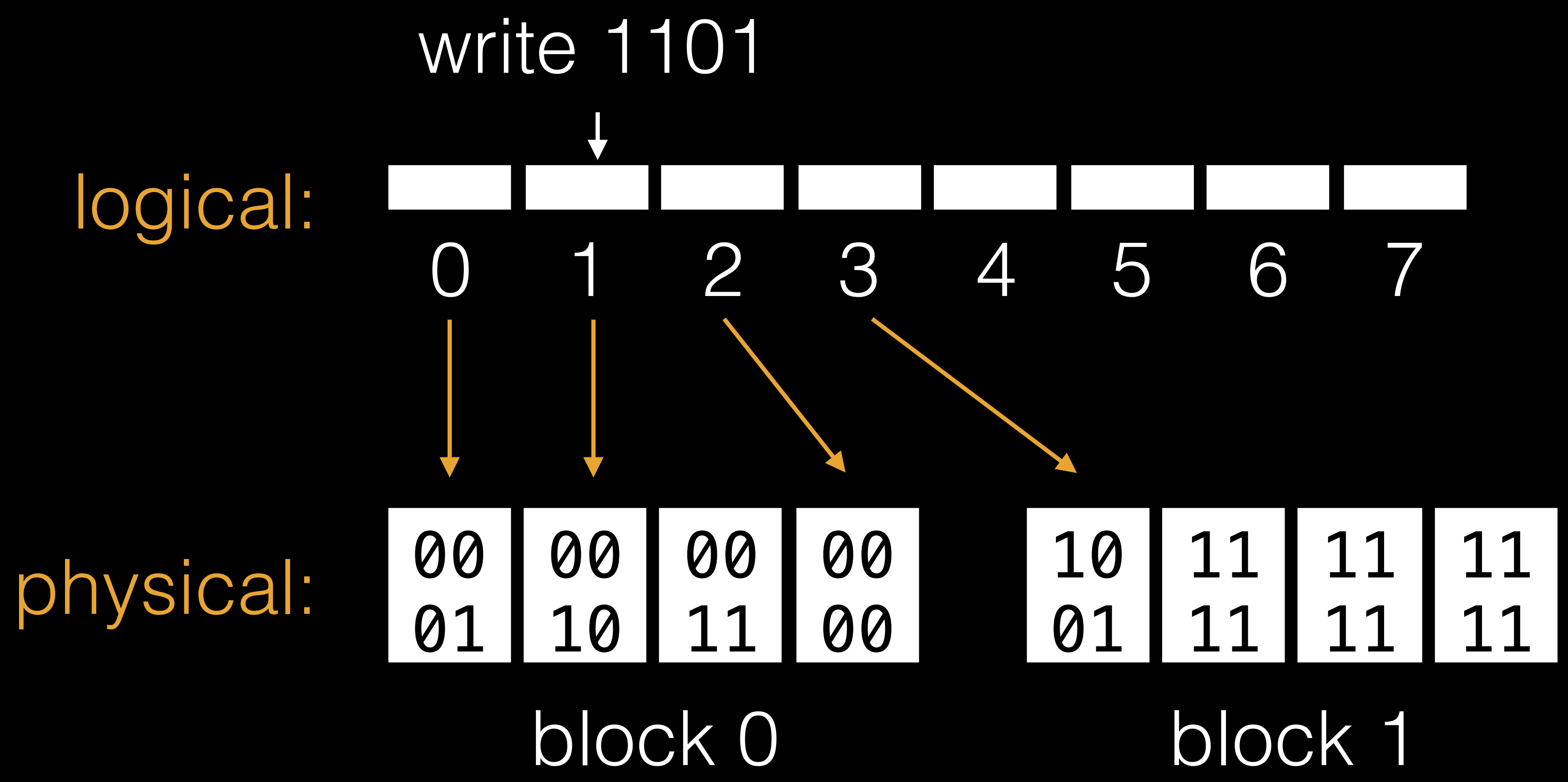
APIs

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

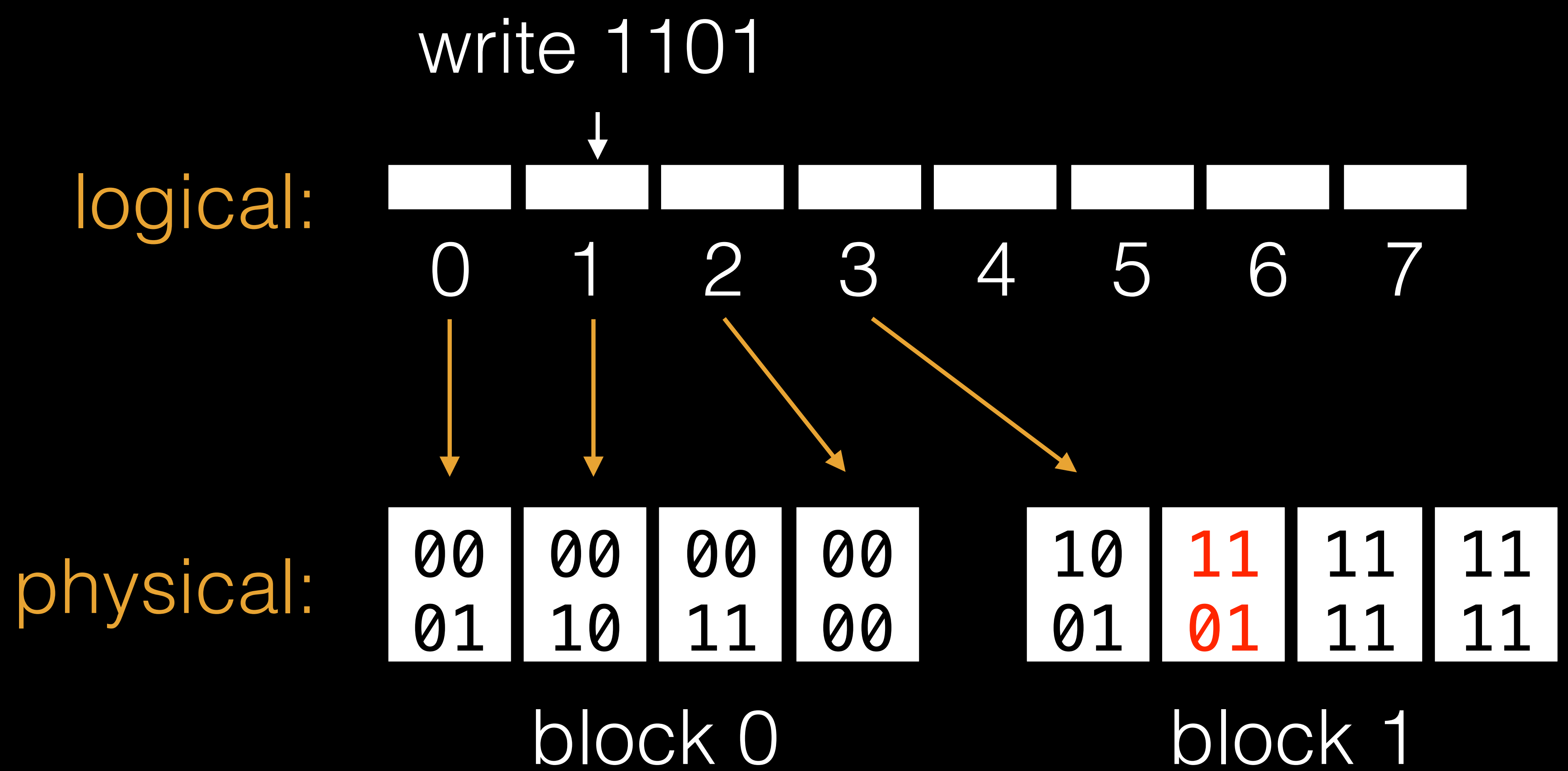
Flash Translation Layer



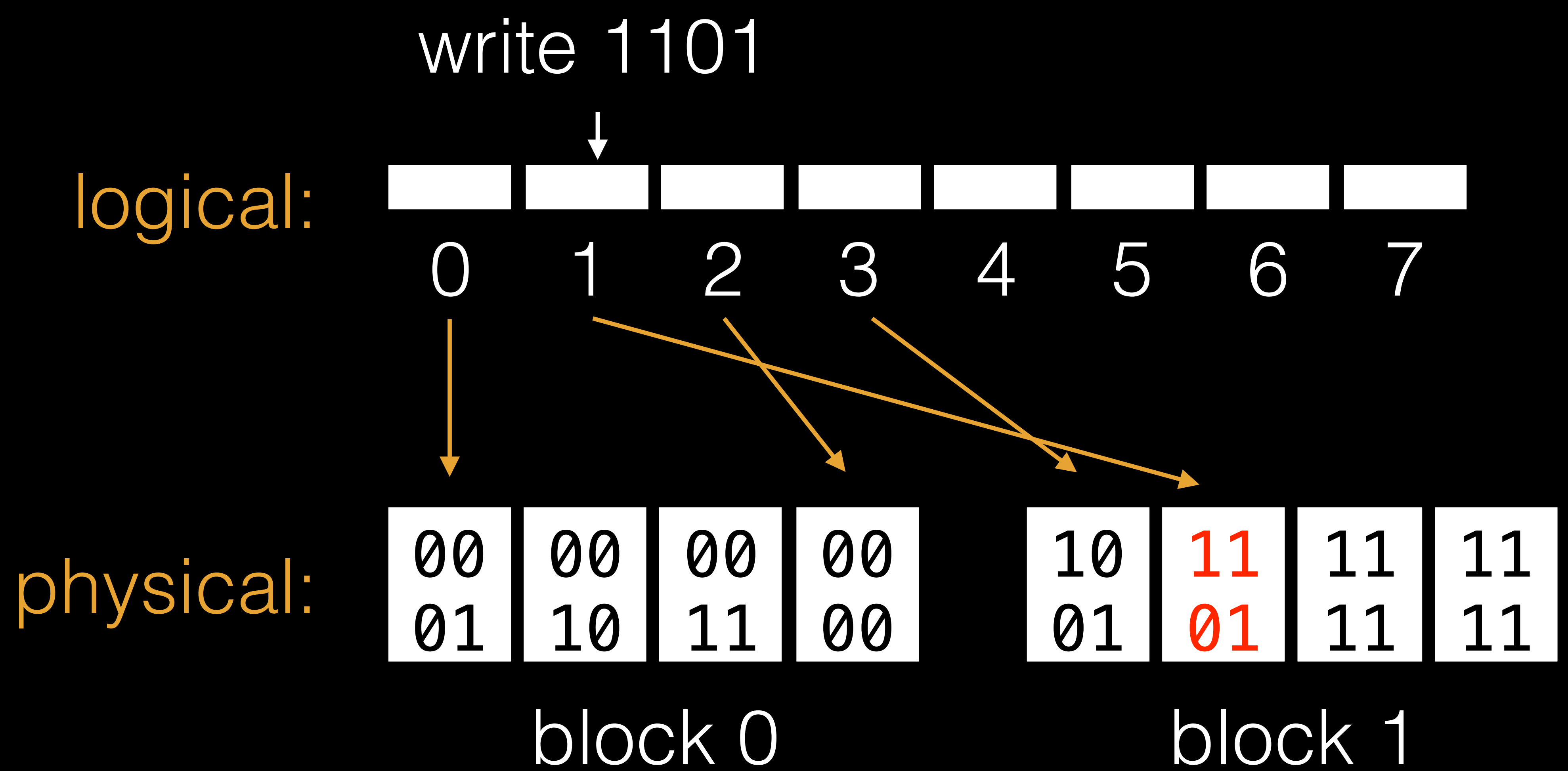
Flash Translation Layer



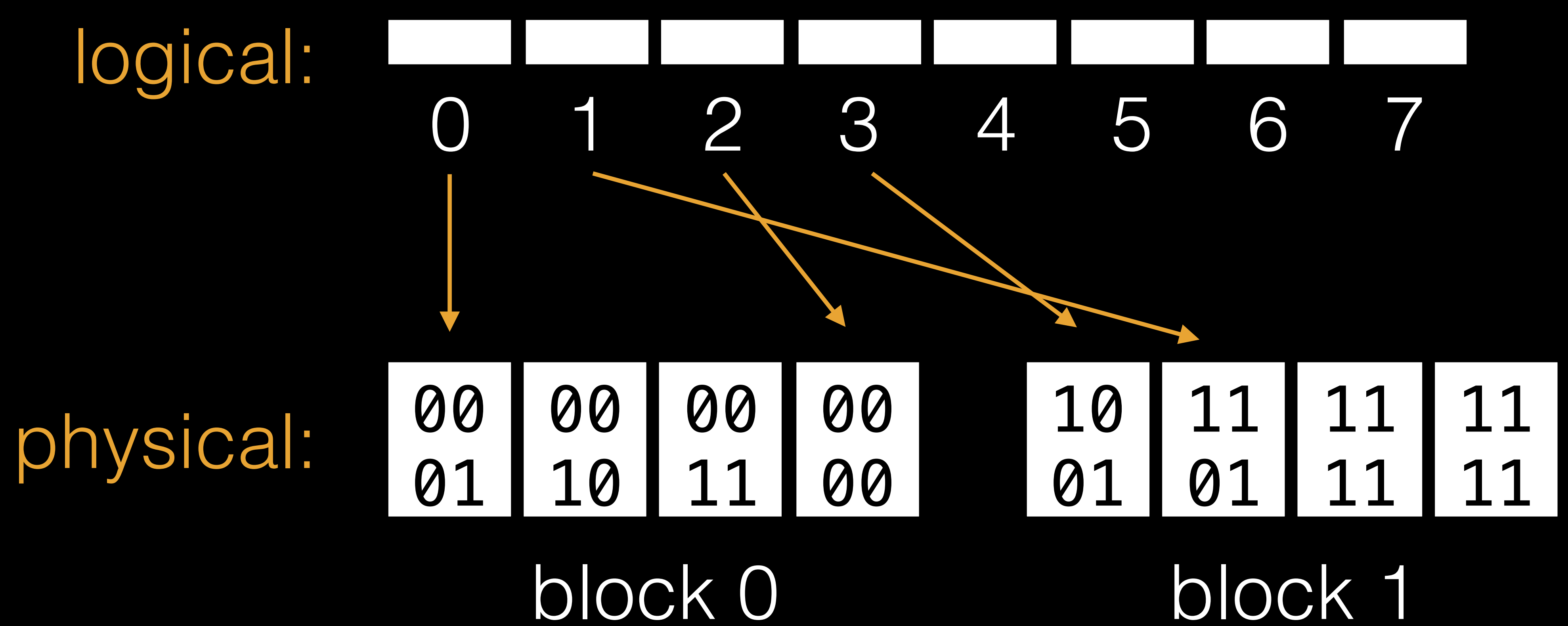
Flash Translation Layer



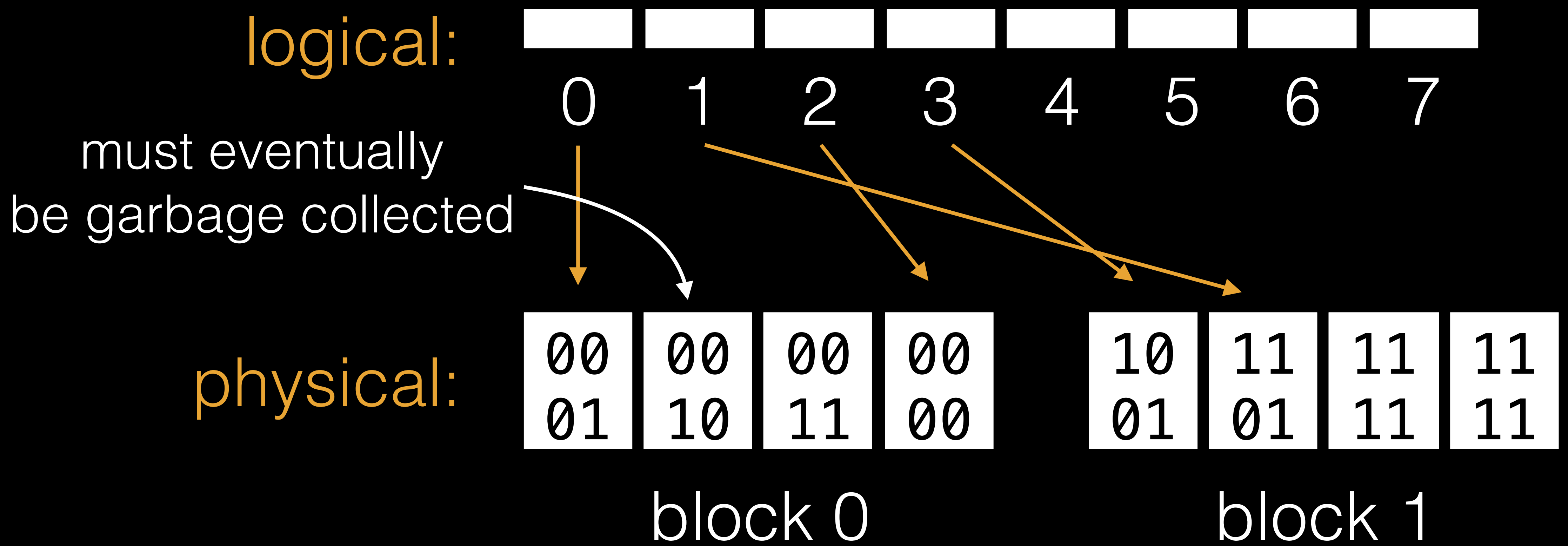
Flash Translation Layer



Flash Translation Layer



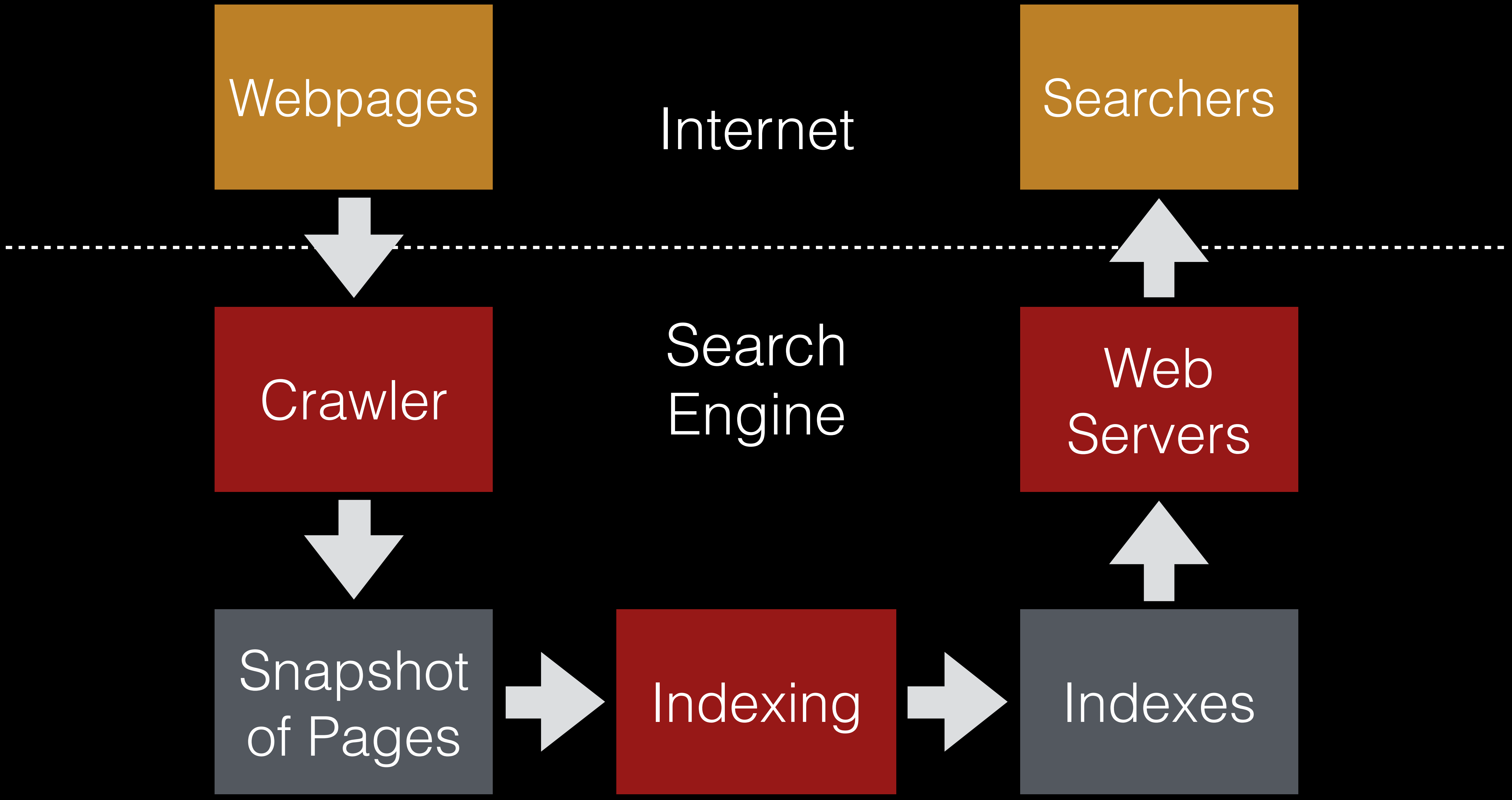
Flash Translation Layer



Search Engines

PageRank: important?

Inverted index: relevant?


Webpages
Internet
Searchers
Crawler
Search Engine
Web Servers
Snapshot of Pages
Indexing
Indexes

Strategy: Count Backlinks

Importance:

A = 1

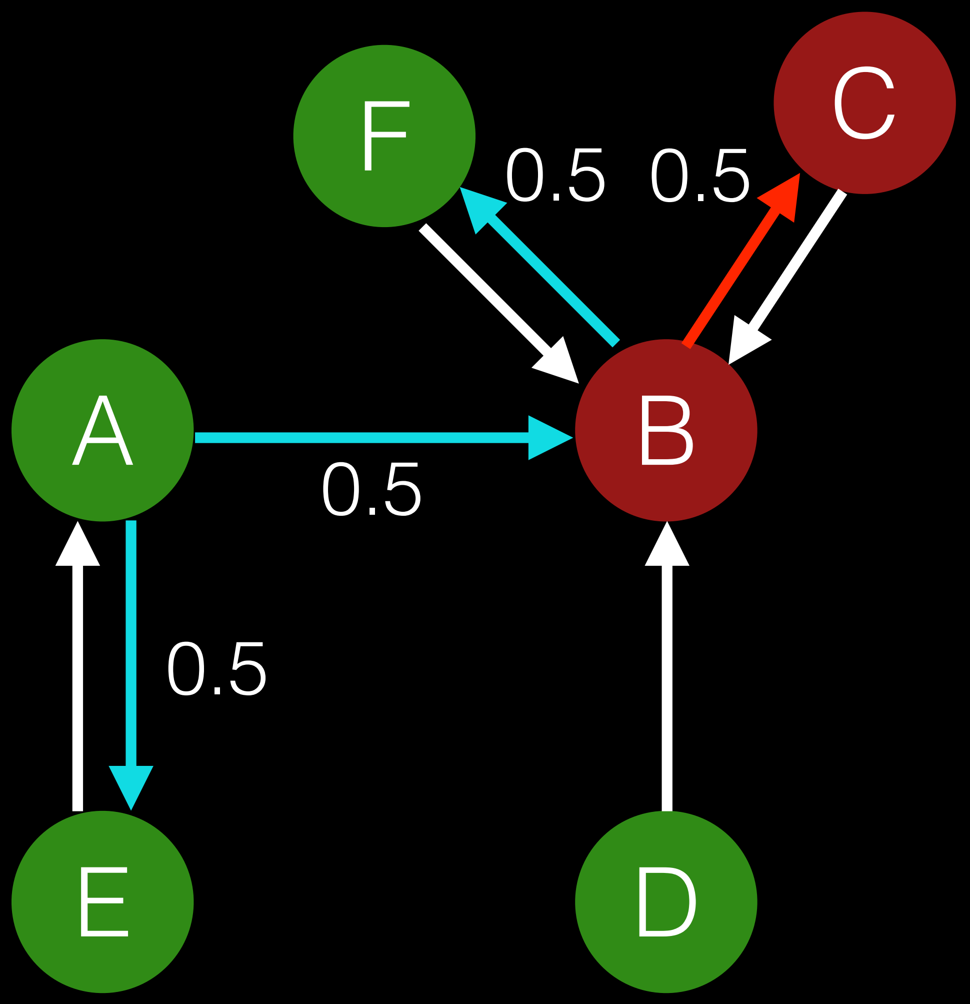
B = 3.5

C = 0.5 (from B's vote)

D = 0

E = 0.5 (from A's vote)

F = 0.5



Why do A and B get same votes? B is more important.

Circular Votes

Want: number of votes you get determines number of votes you give.

Problem: changing A's votes changes B's votes changes A's votes…

Fortunately, if you just keep updating every PageRank, it eventually converges.

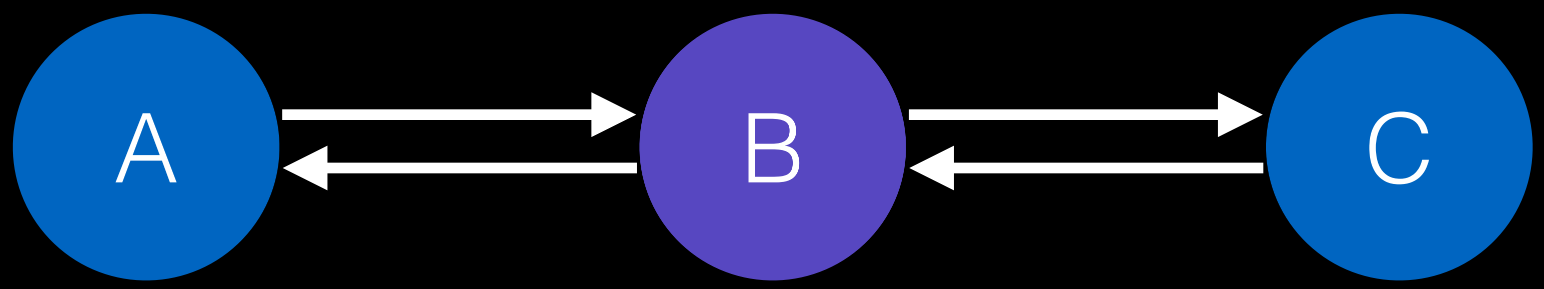
Intuition: Random Surfer

Imagine!

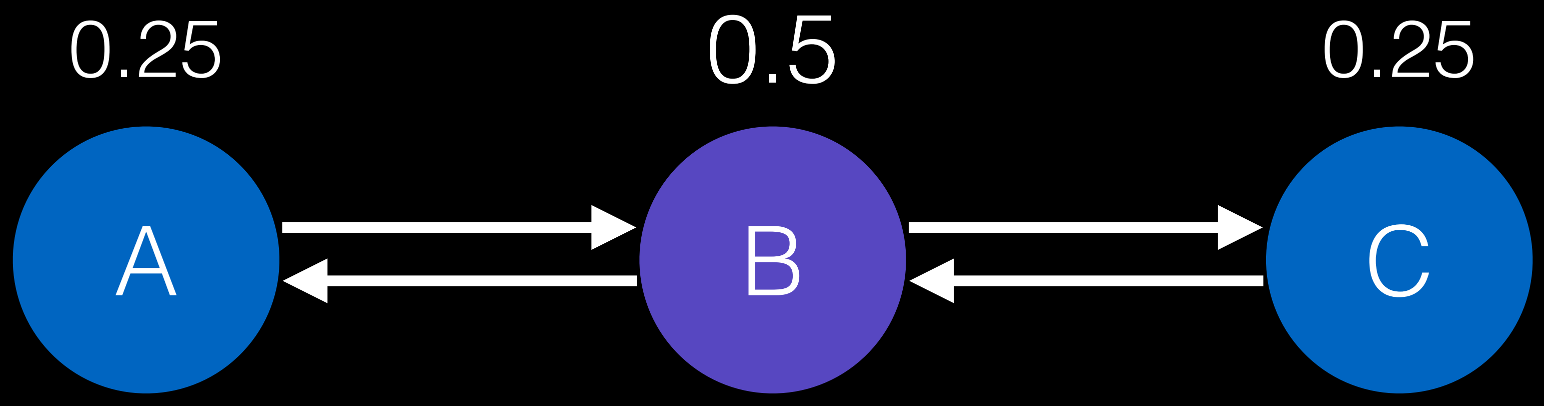
1. a bunch of web surfers start on various pages
2. they randomly click links, forever
3. you measure webpage visit frequency

Visit frequency will be proportional to PageRank.

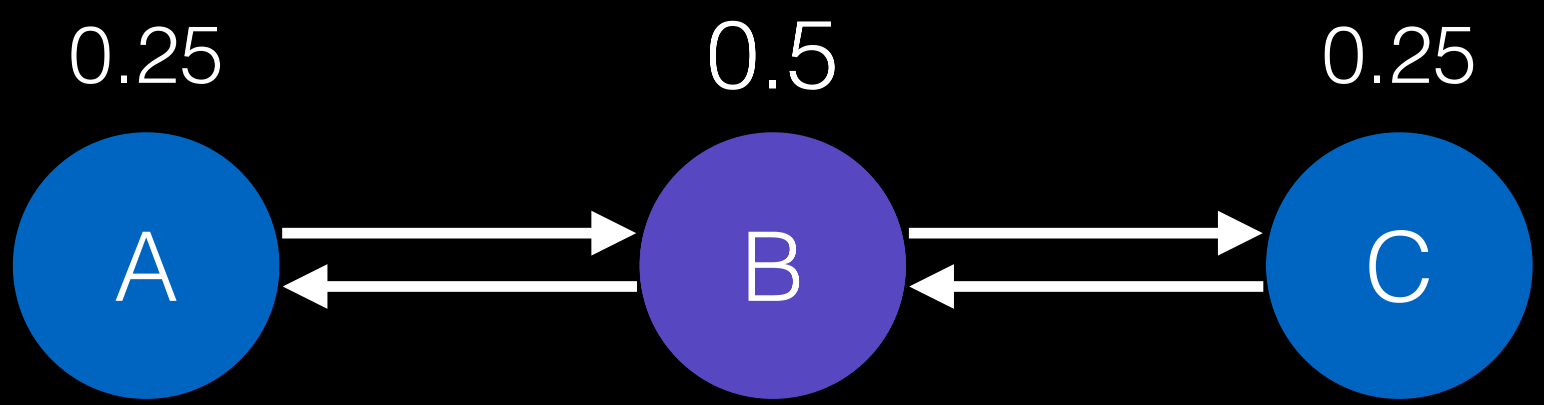
Graph 1



Graph 1



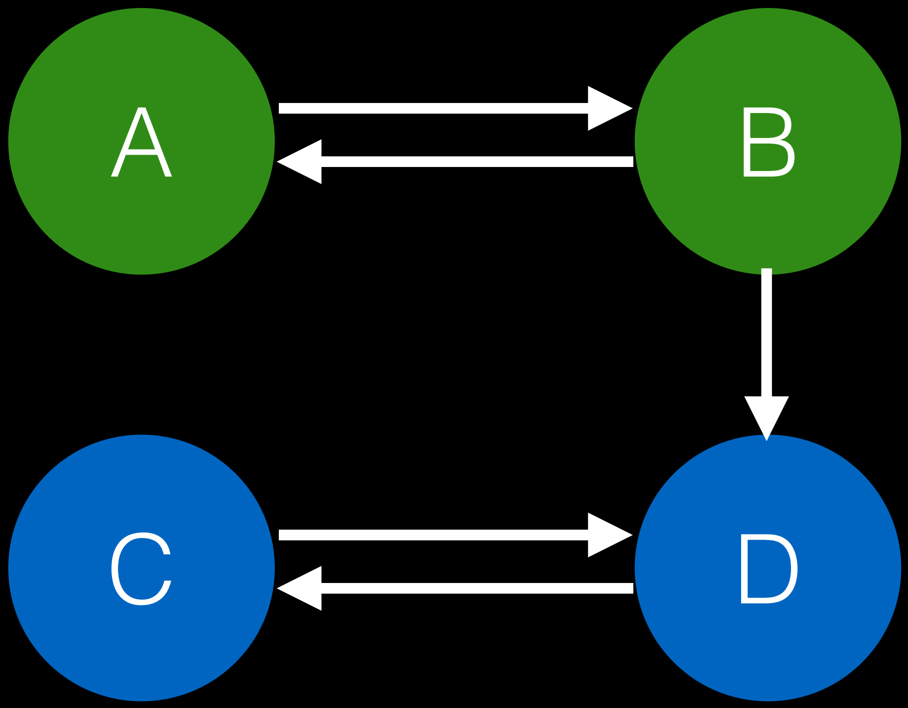
Graph 1



Rank(B) = (0.25 / 1) + (0.25 / 1) = 0.5
Rank(A) = (0.5 / 2) = 0.25
Rank(C) = (0.5 / 2) = 0.25

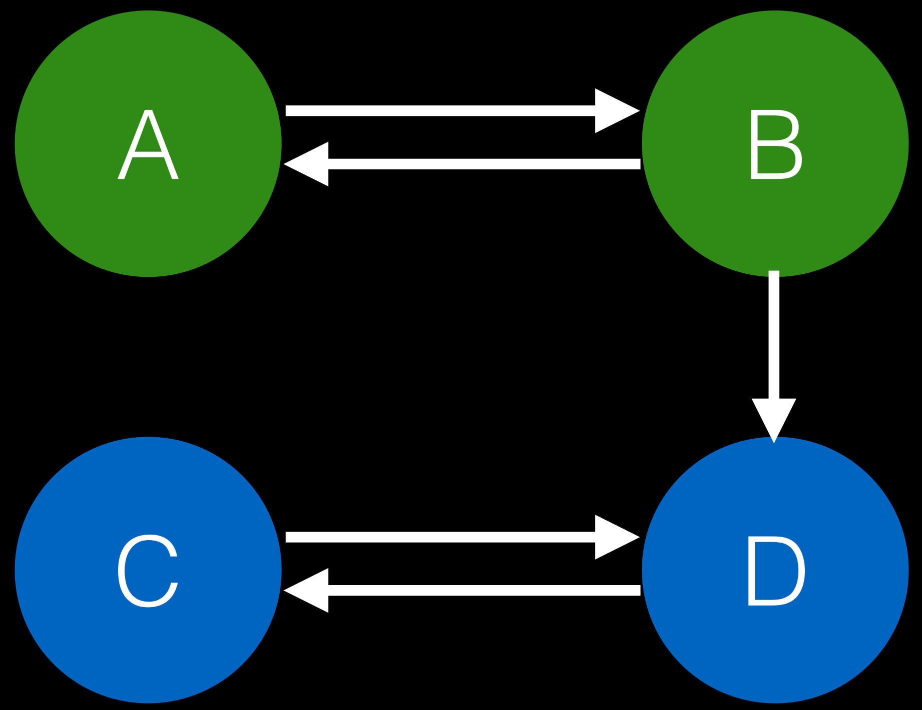
$$\text{Rank}(x) = c \sum_{y \in \text{LinksTo}(x)} \frac{\text{Rank}(y)}{N_y}$$

Graph 3



Problem: ???

Graph 3



Problem: Surfers get stuck in C and D.
C+D called a rank "sink". A and B get 0 rank.

Inverted Index

forward index

docID	wordID
1442	5
1442	922
1442	2
1442	66
1442	42
1442	5
...	...

Inverted Index

forward index

docID	wordID
1442	5
1442	922
1442	2
1442	66
1442	42
1442	5
…	…

docID	wordID
1442	5
1442	922
1442	2
1442	66
1442	42
1442	5
…	…

Inverted Index

forward index

docID	wordID
1442	5
1442	922
1442	2
1442	66
1442	42
1442	5
...	...

swap columns

wordID	docID
5	1442
922	1442
2	1442
66	1442
42	1442
5	1442
...	...

Inverted Index

forward index

docID	wordID
1442	5
1442	922
1442	2
1442	66
1442	42
1442	5
…	…

sort by wordID

wordID	docID
1	244
2	1442
5	1442
5	1442
5	999
6	133
…	…

Inverted Index

forward index

docID	wordID
1442	5
1442	922
1442	2
1442	66
1442	42
1442	5
…	…

inverted index

wordID	docID
1	244
2	1442
5	1442,1442,999
6	133,411
7	1442,133,999
9	411,875
…	…