

Basic Memory Management

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

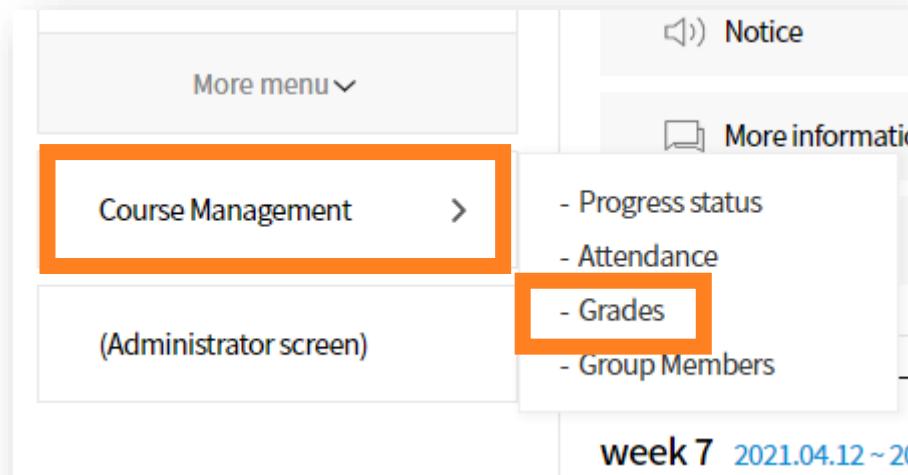
KAIST EE

CAMEL*ab*



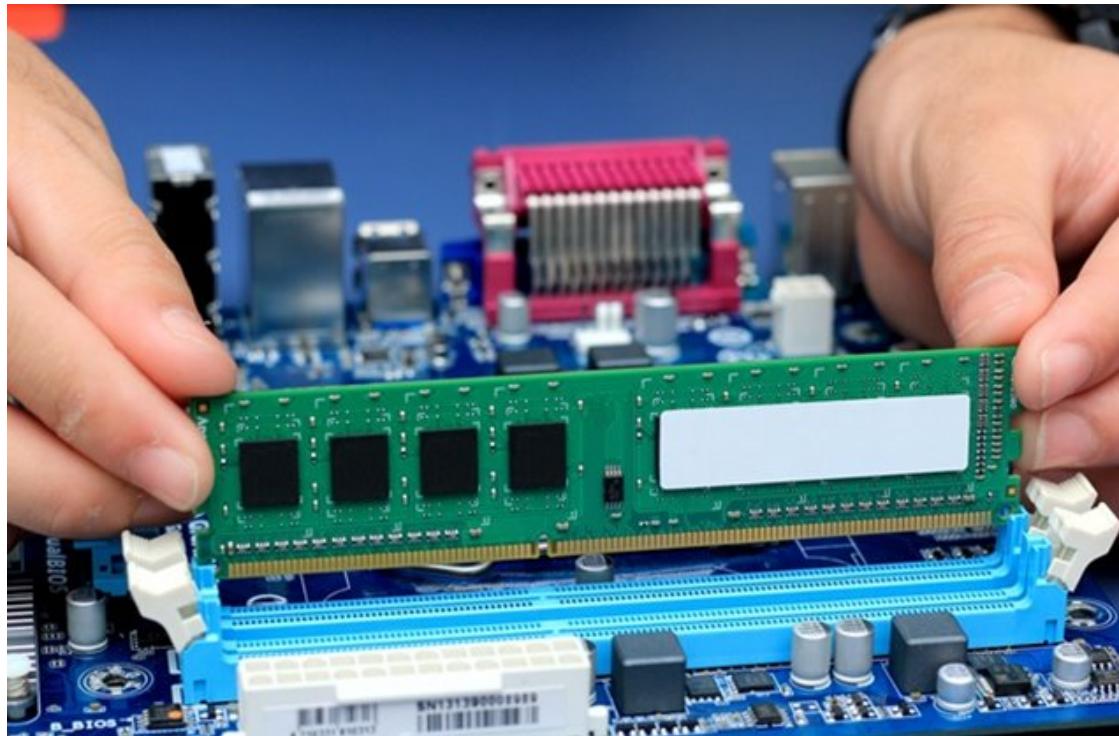
NOTICE: Project 1 Score is Ready

- Available at KLMS > Course Management > Grades
- Contact TA: Hanyeoreum Bae (hyr.bae@camelab.org)
- Claim Deadline: 04/23 (Friday) 23:59:59



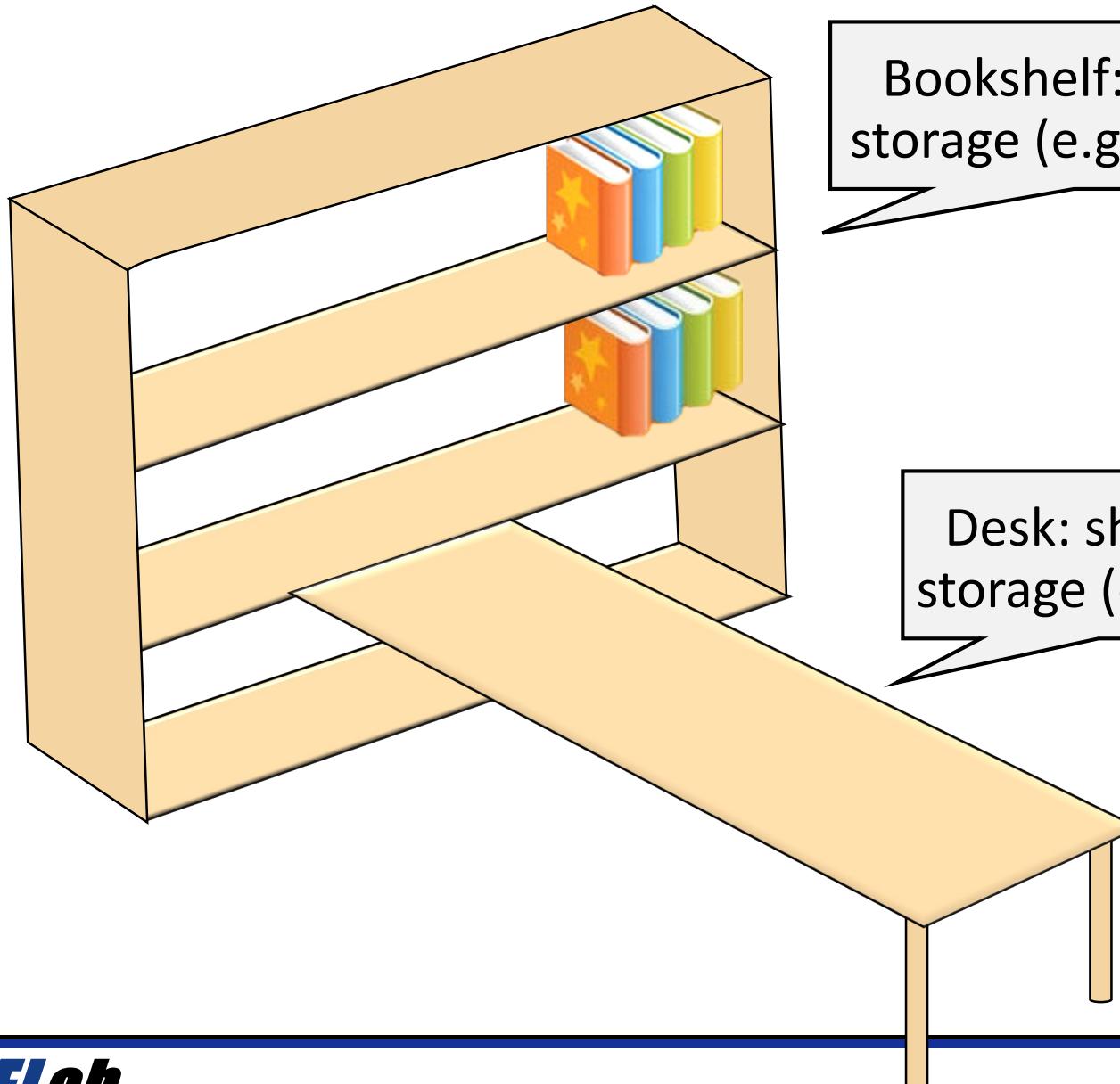
Random Access Memory (RAM)

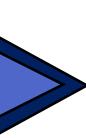
RAM is the working memory that stores instructions and data needed to run all software



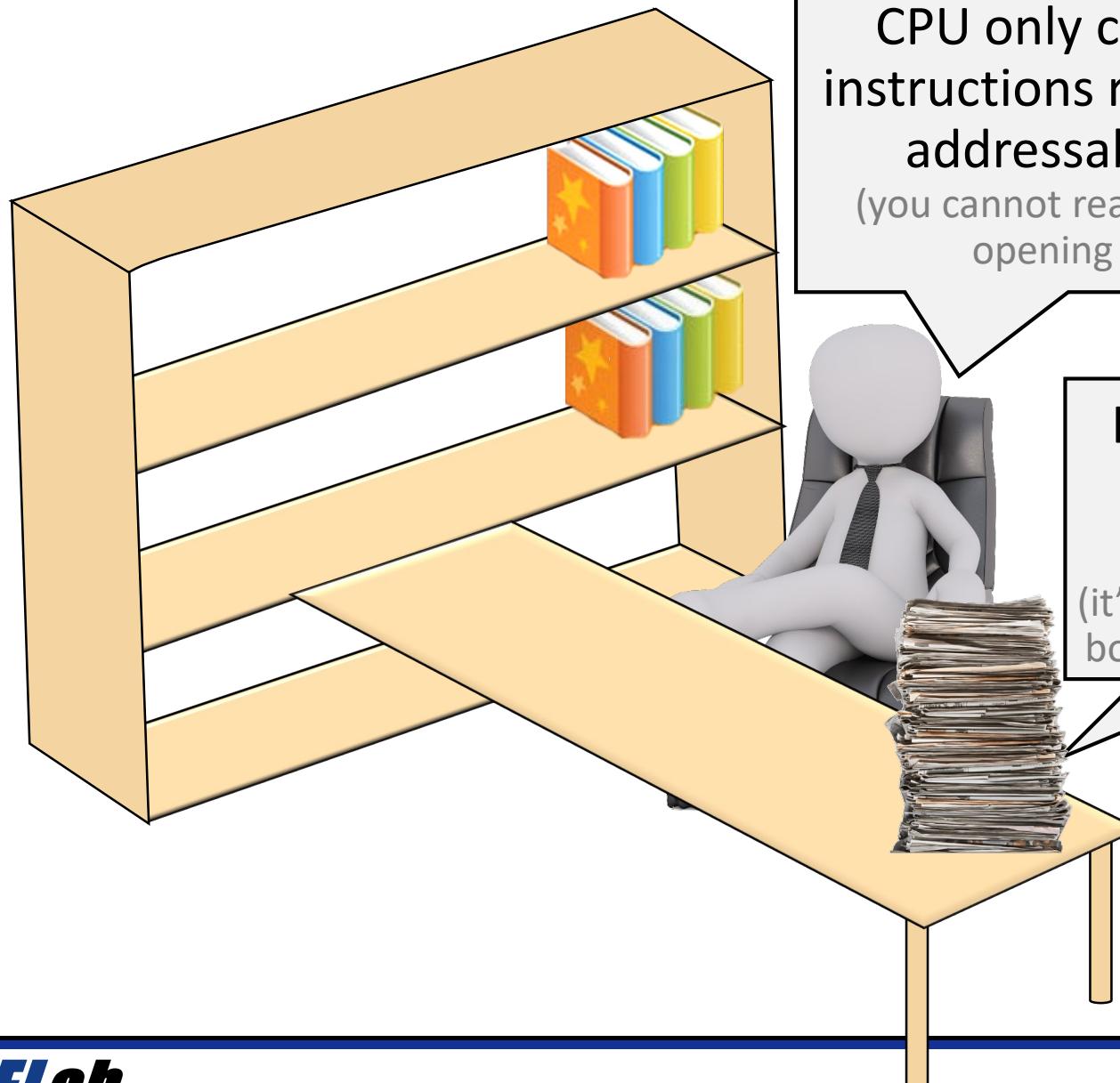


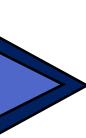
Analogy for RAM: Desk



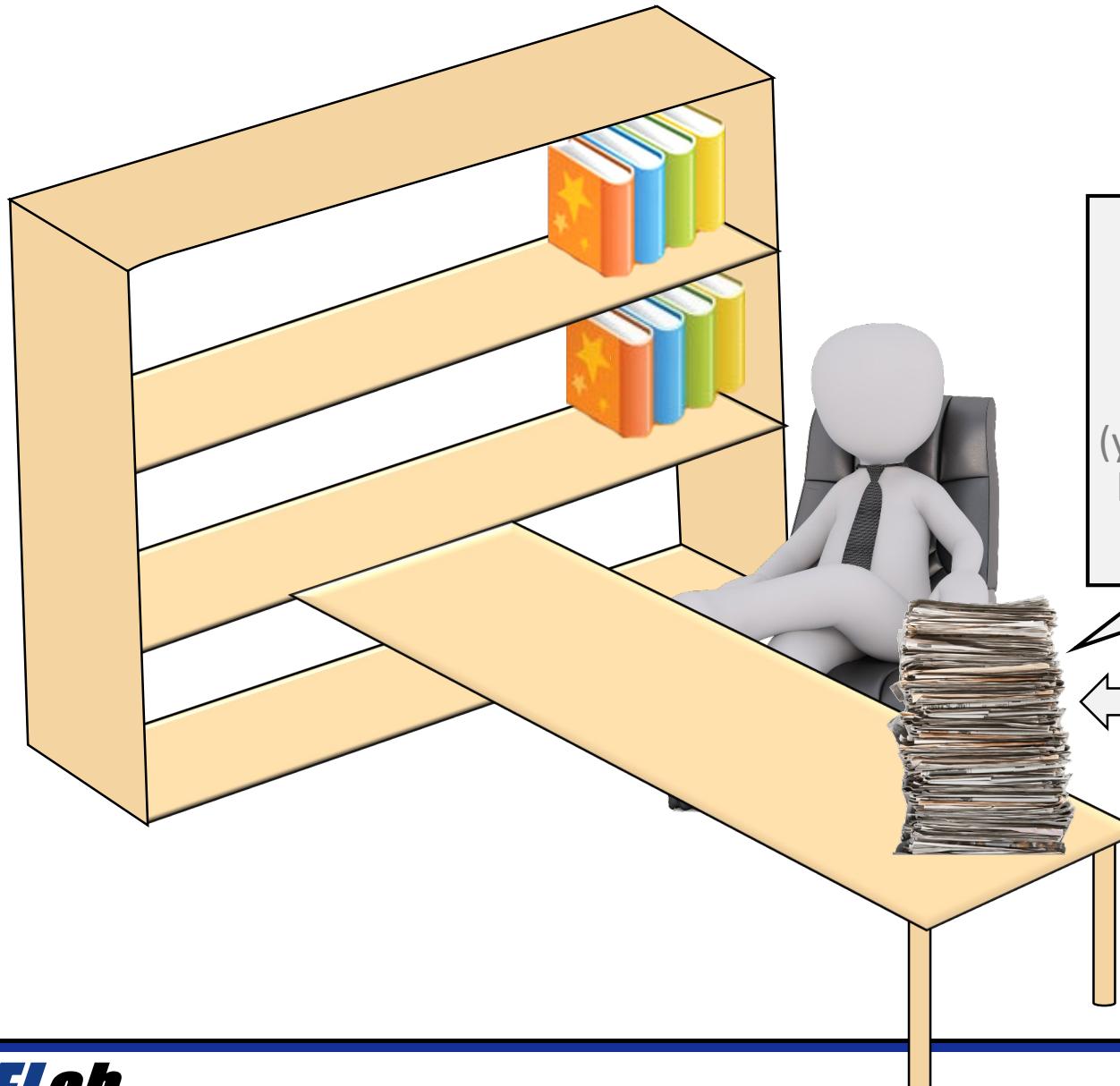


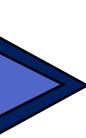
Analogy for RAM: Desk



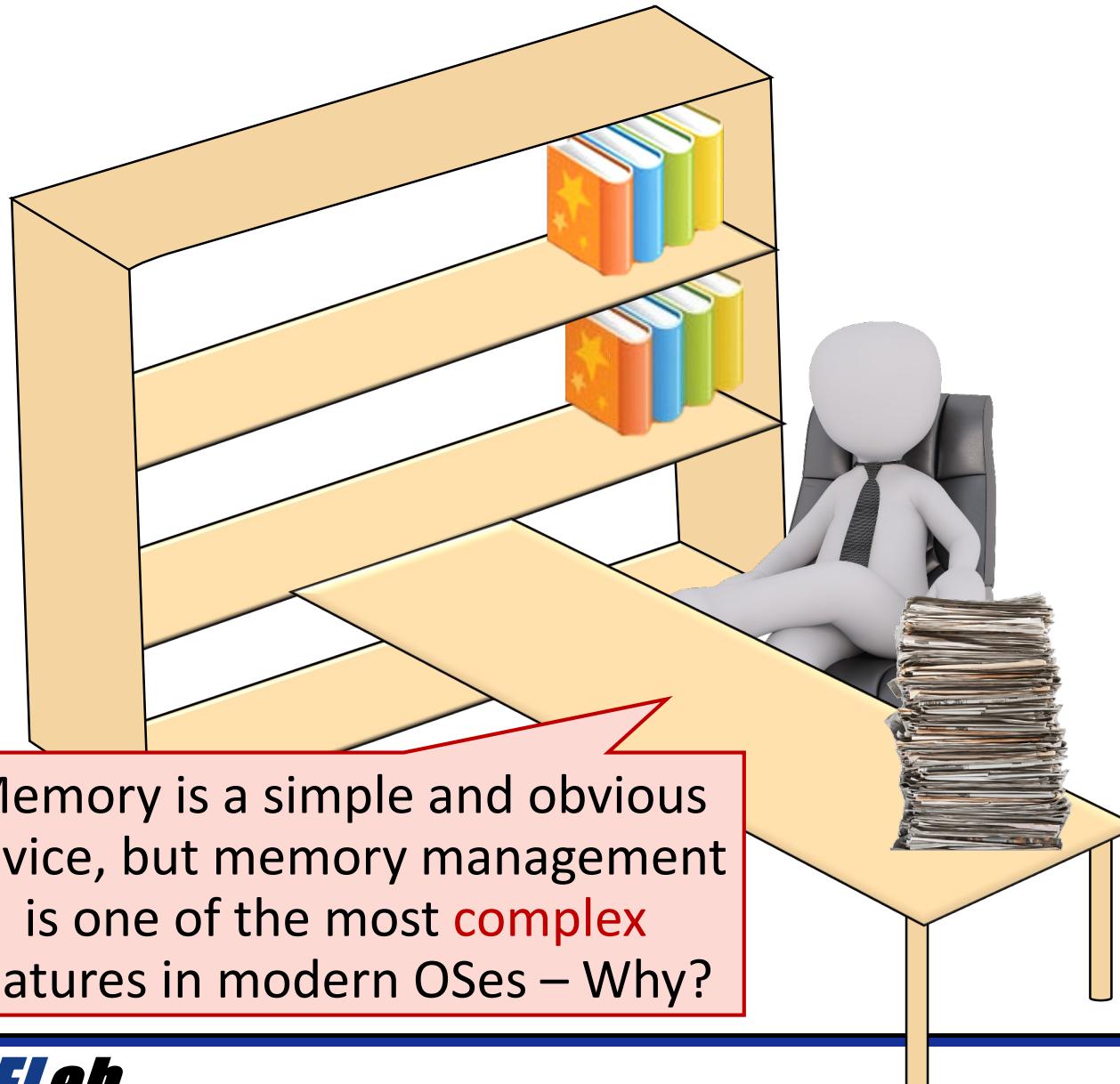


Analogy for RAM: Desk

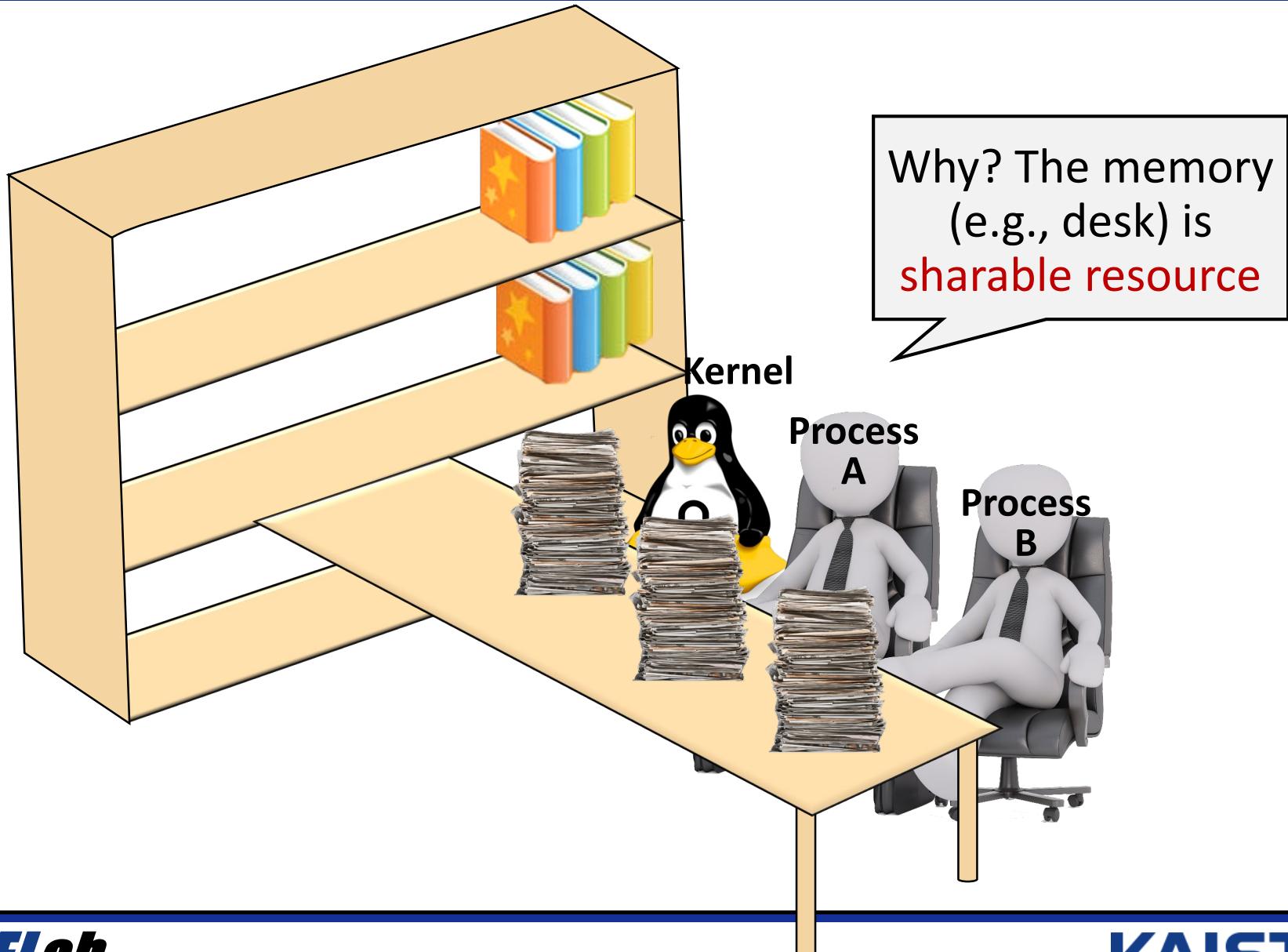


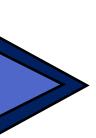


Analogy for RAM: Desk

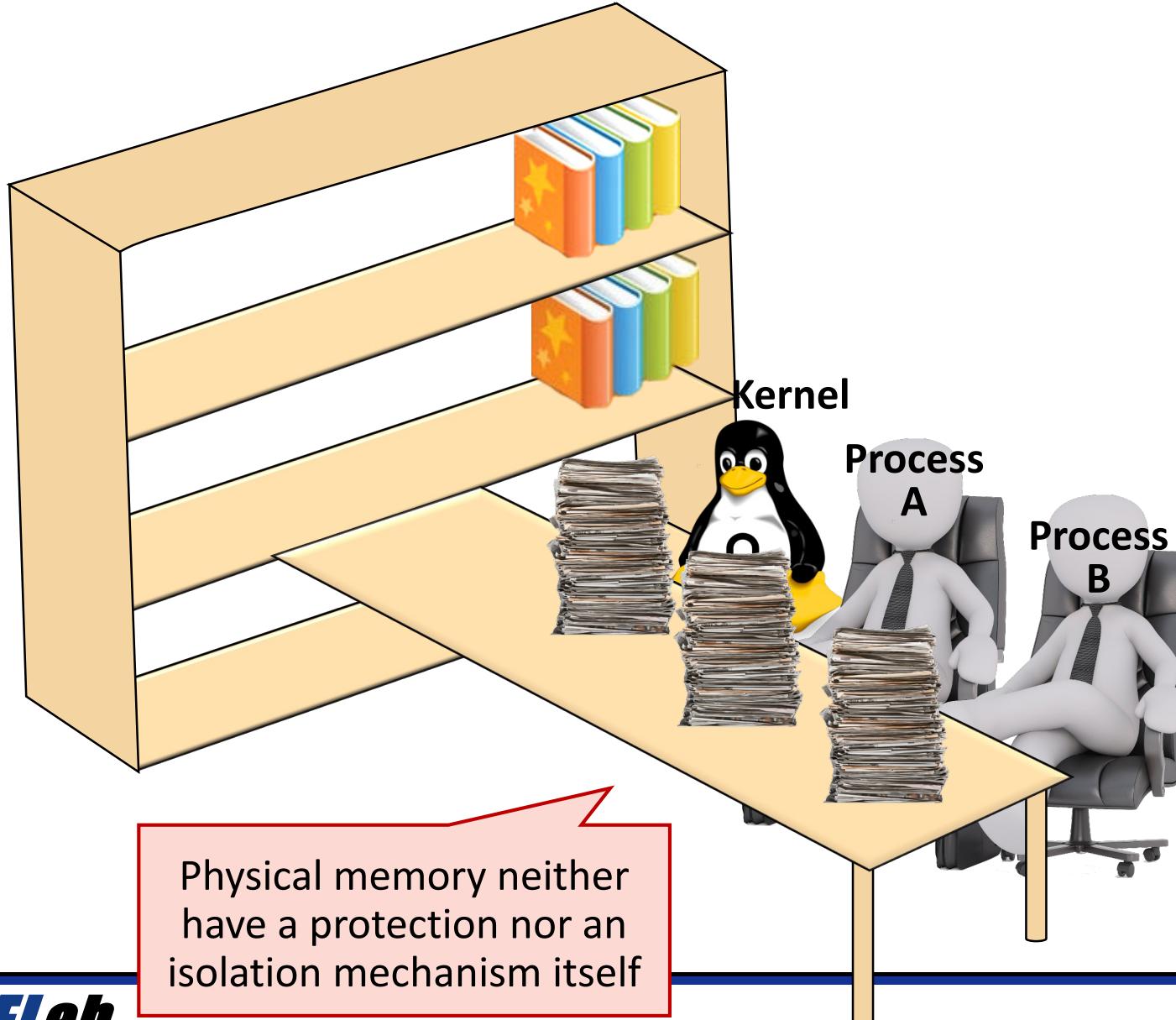


Limitations of Physical Memory



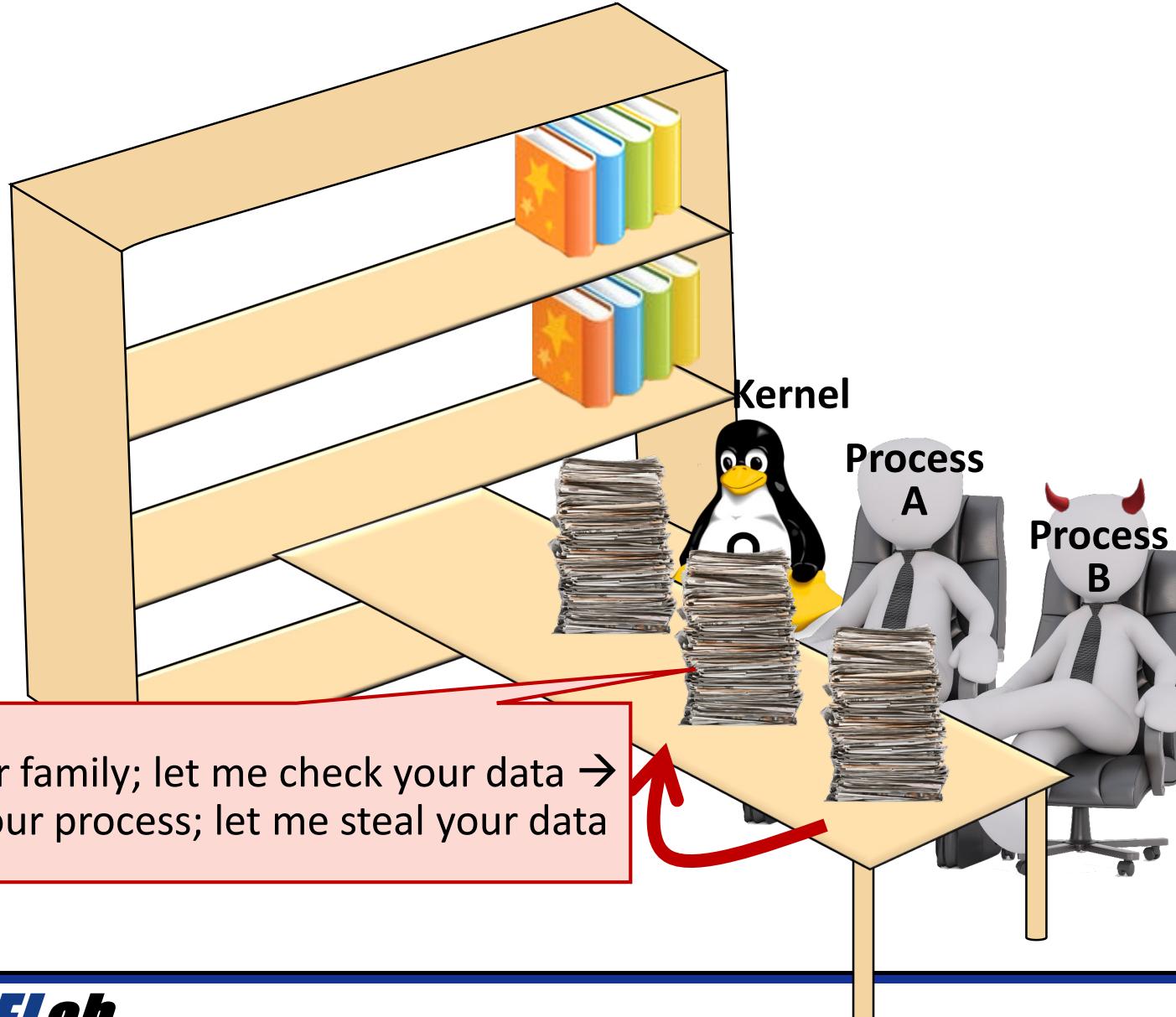


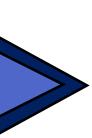
Limitation #1: Protection



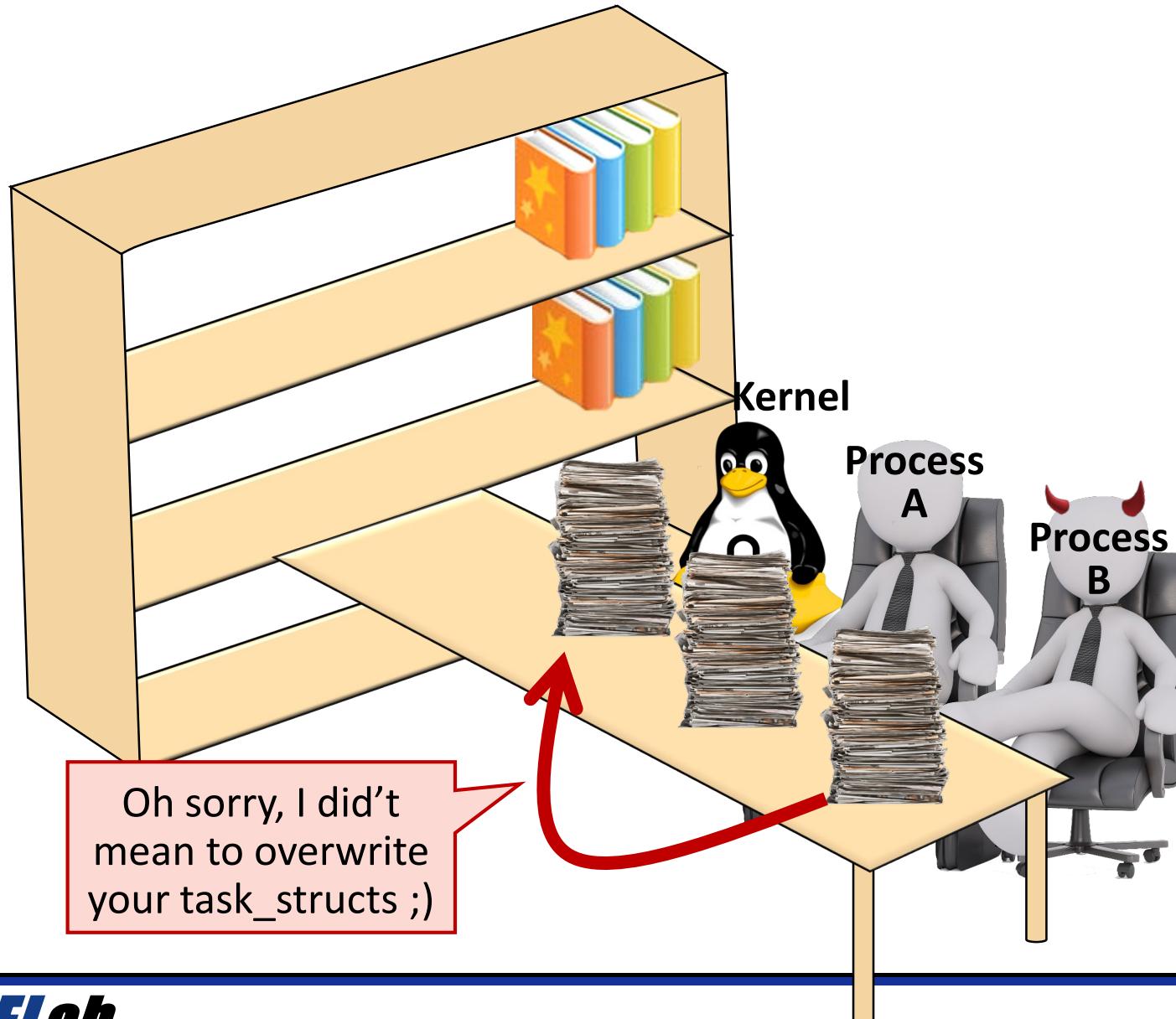


Limitation #1: Protection





Limitation #1: Protection



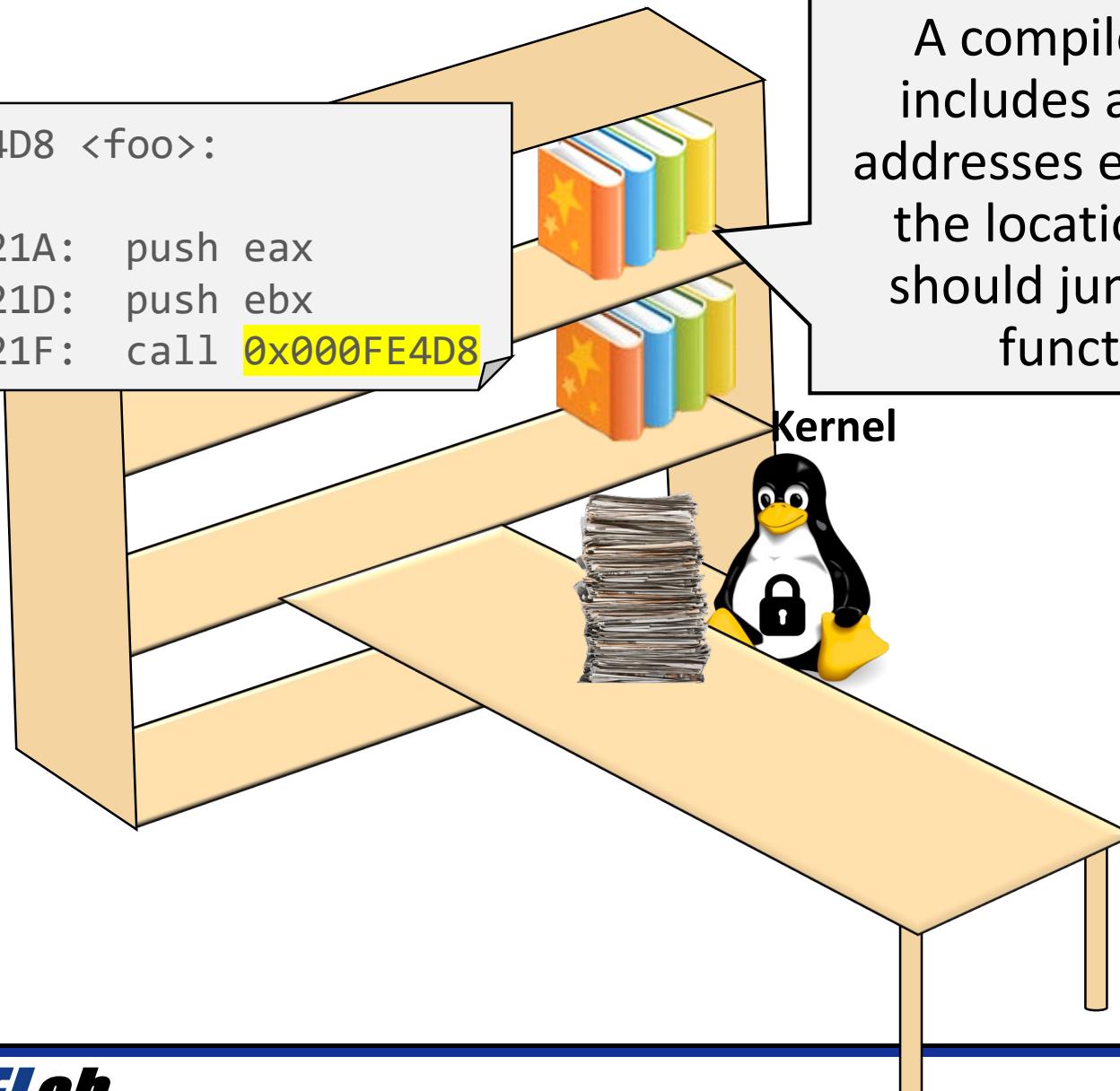
Limitation #2: Addressing

```
000FE4D8 <foo>:
```

```
...
```

```
000FE21A: push eax  
000FE21D: push ebx  
000FE21F: call 0x000FE4D8
```

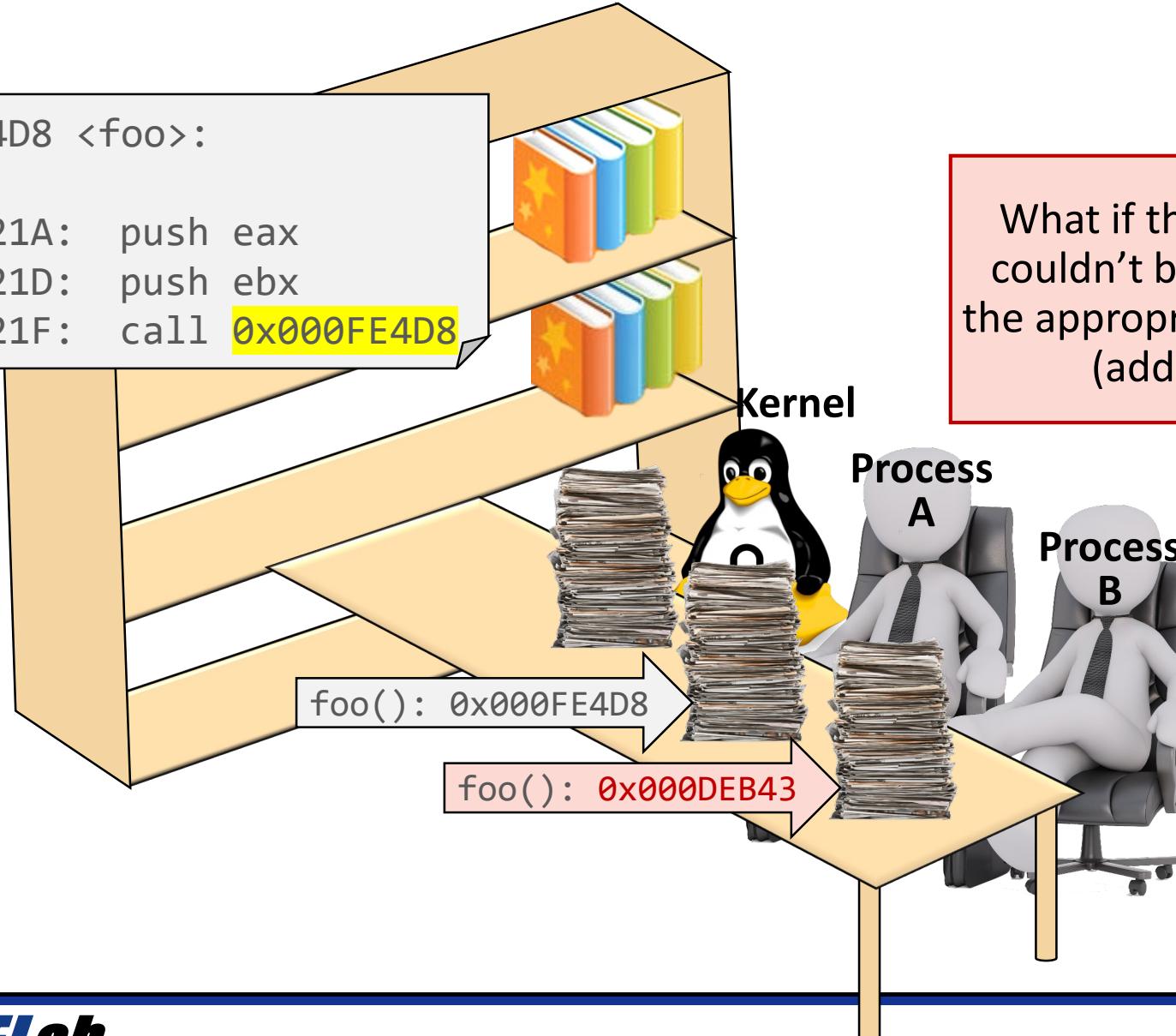
A compiled program includes a set of fixed addresses each indicating the location of what it should jump into (e.g., function call)



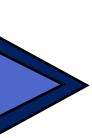
Limitation #2: Addressing

```
000FE4D8 <foo>:
```

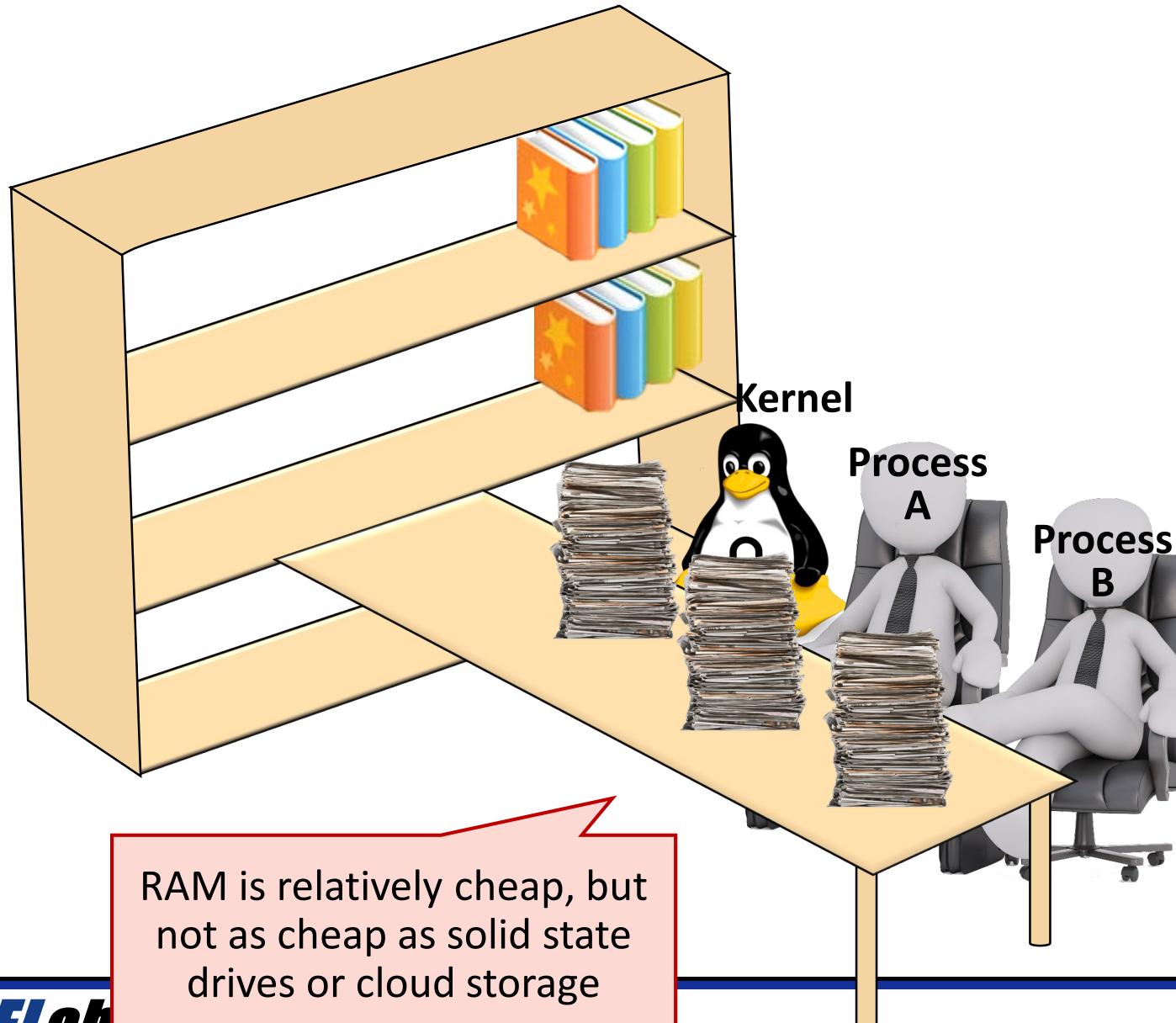
```
...
000FE21A: push eax
000FE21D: push ebx
000FE21F: call 0x000FE4D8
```



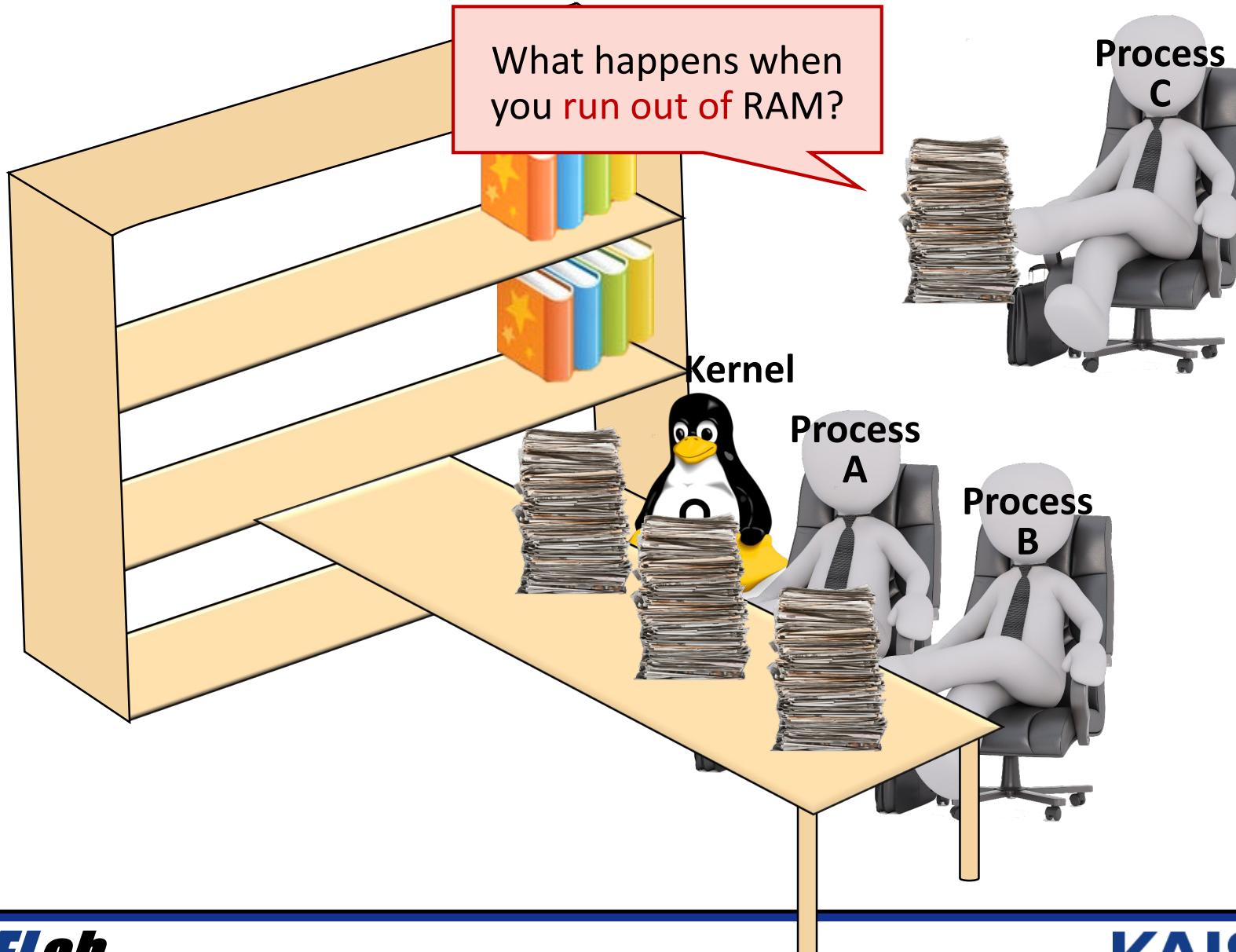
What if the program
couldn't be loaded at
the appropriate location
(address)?



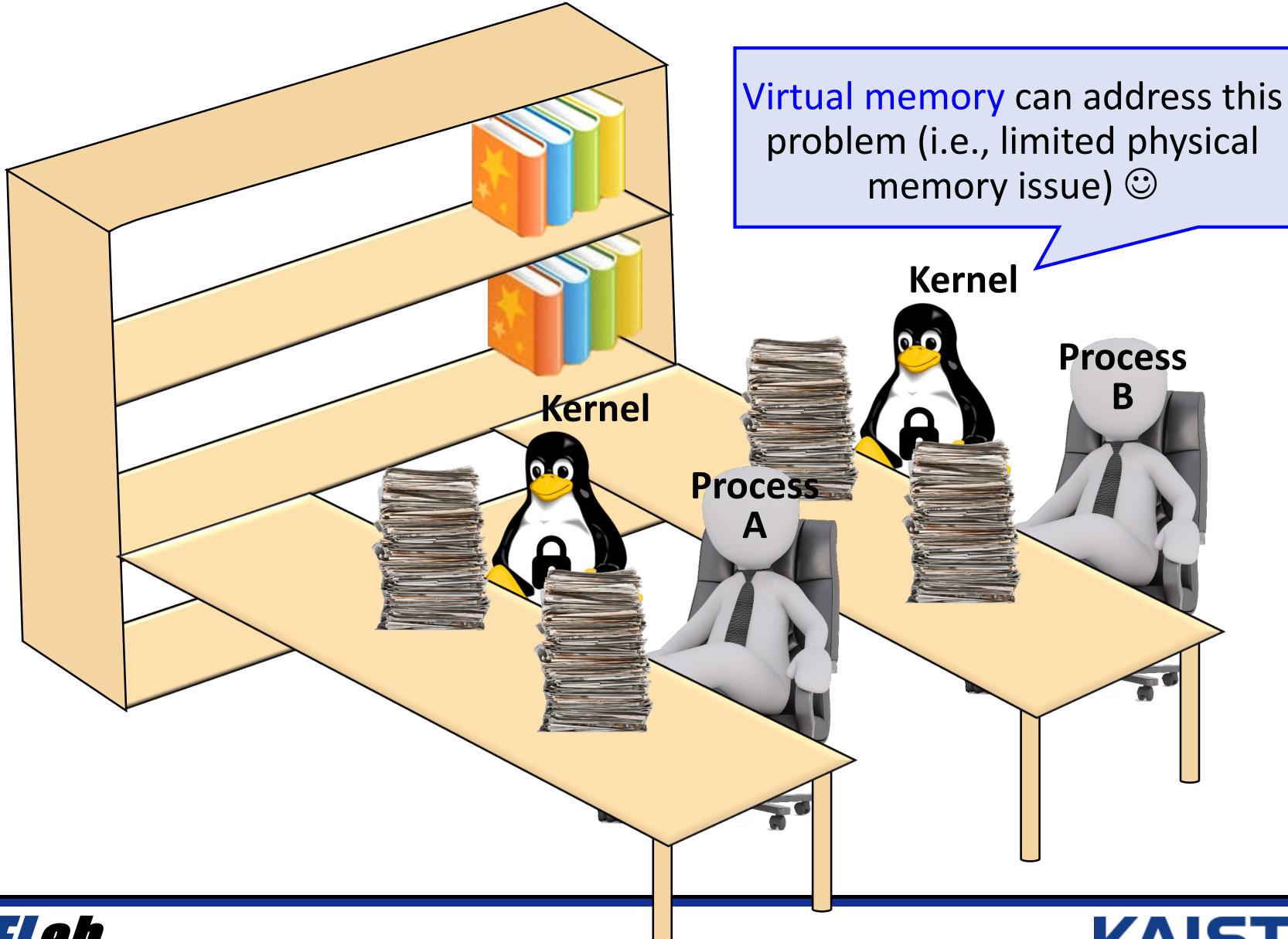
Limitation #3: Limited Size



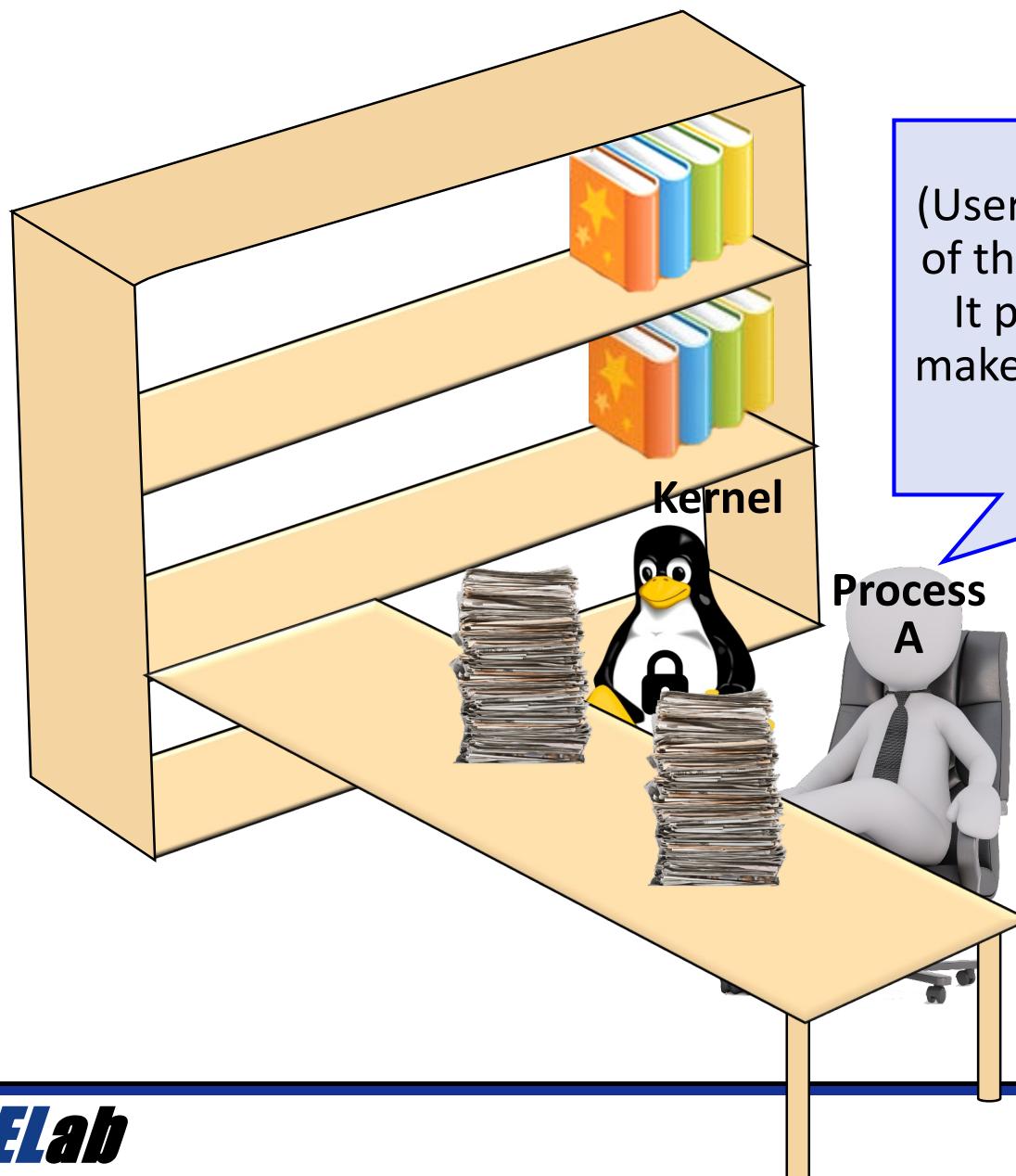
Limitation #3: Limited Size



Towards Virtual Memory

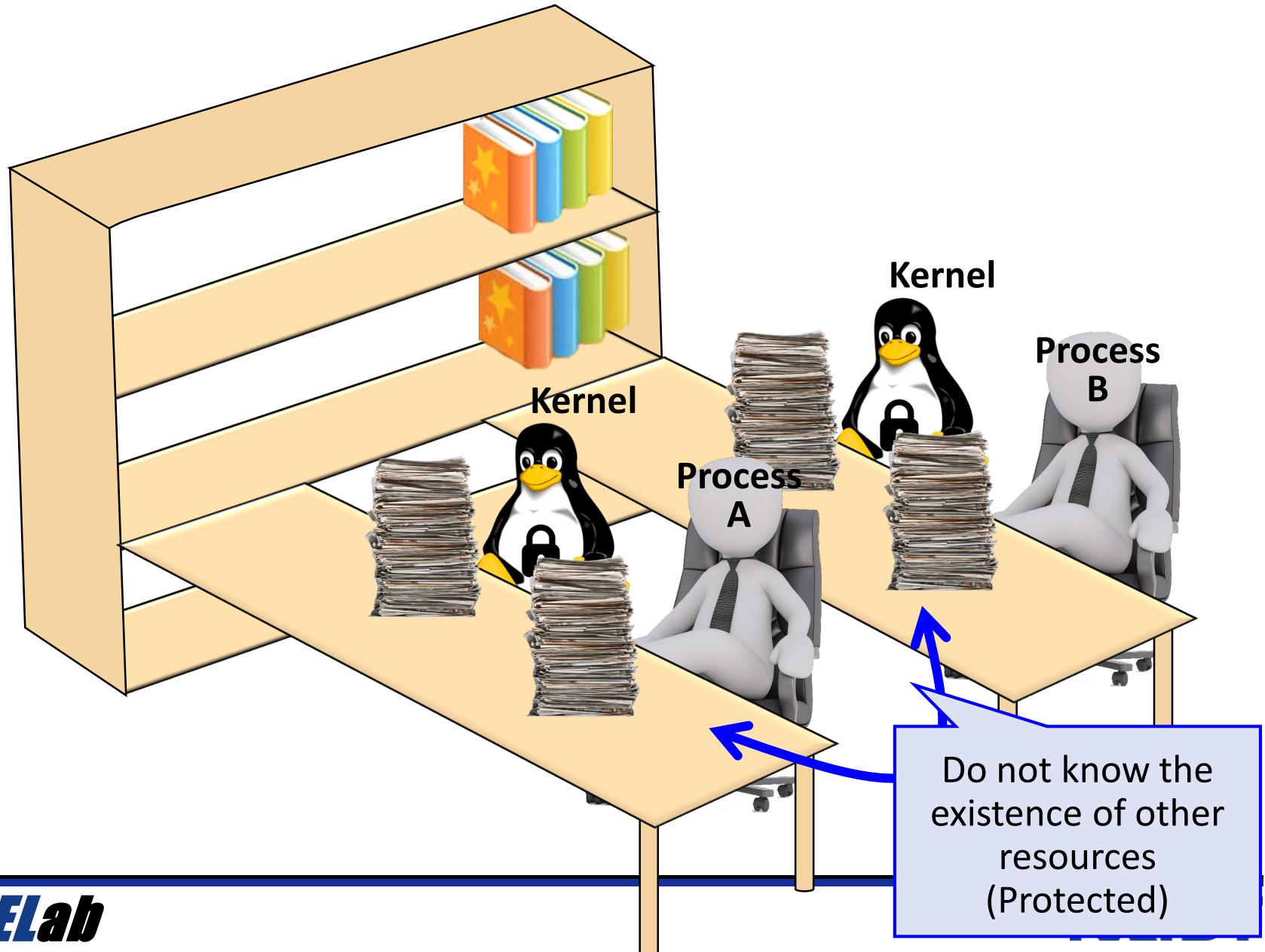


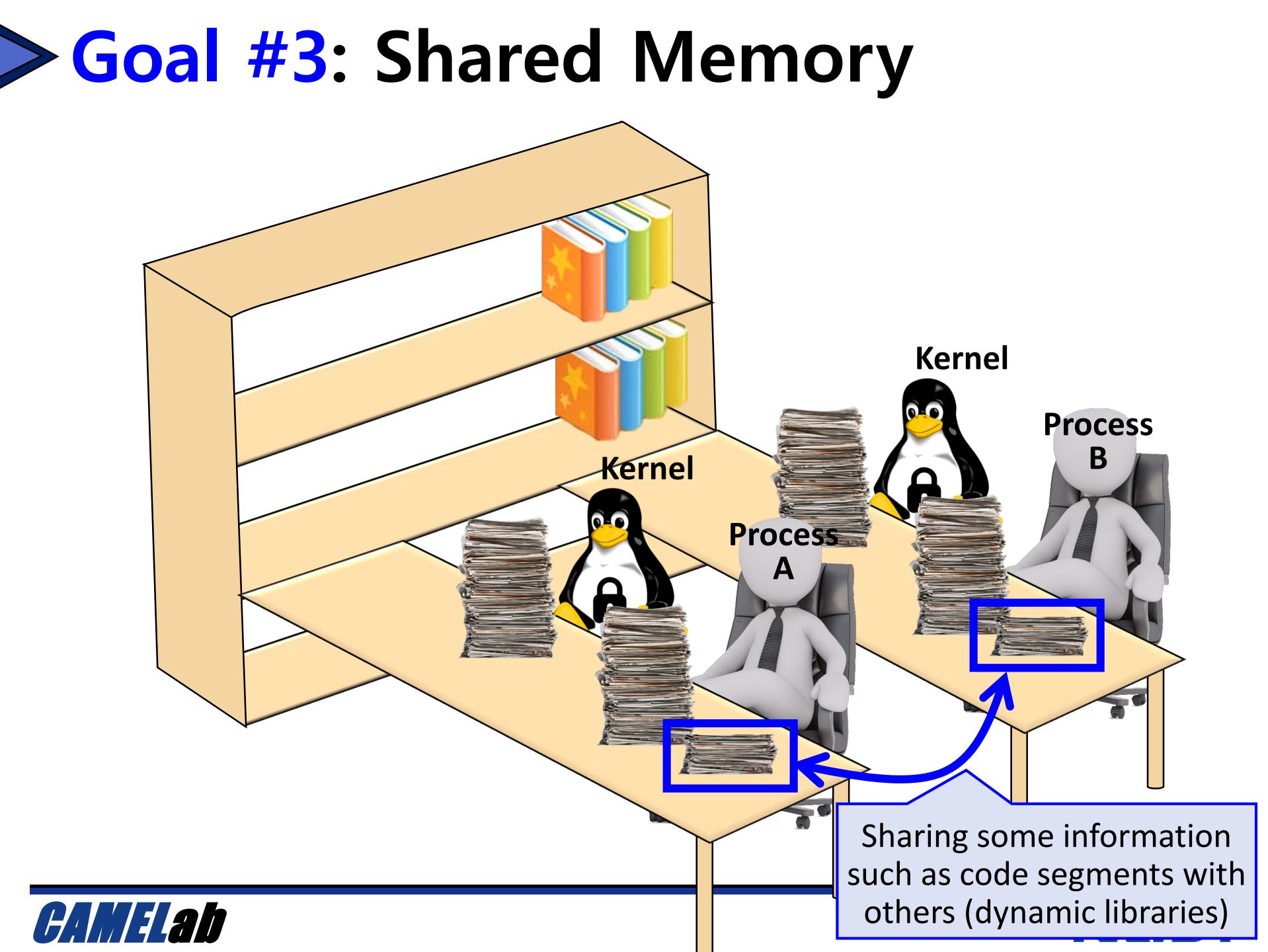
Goal #1: Transparency



(User) processes are unaware of the memory virtualization.
It provides an **illusion** that make a process use the entire memory space

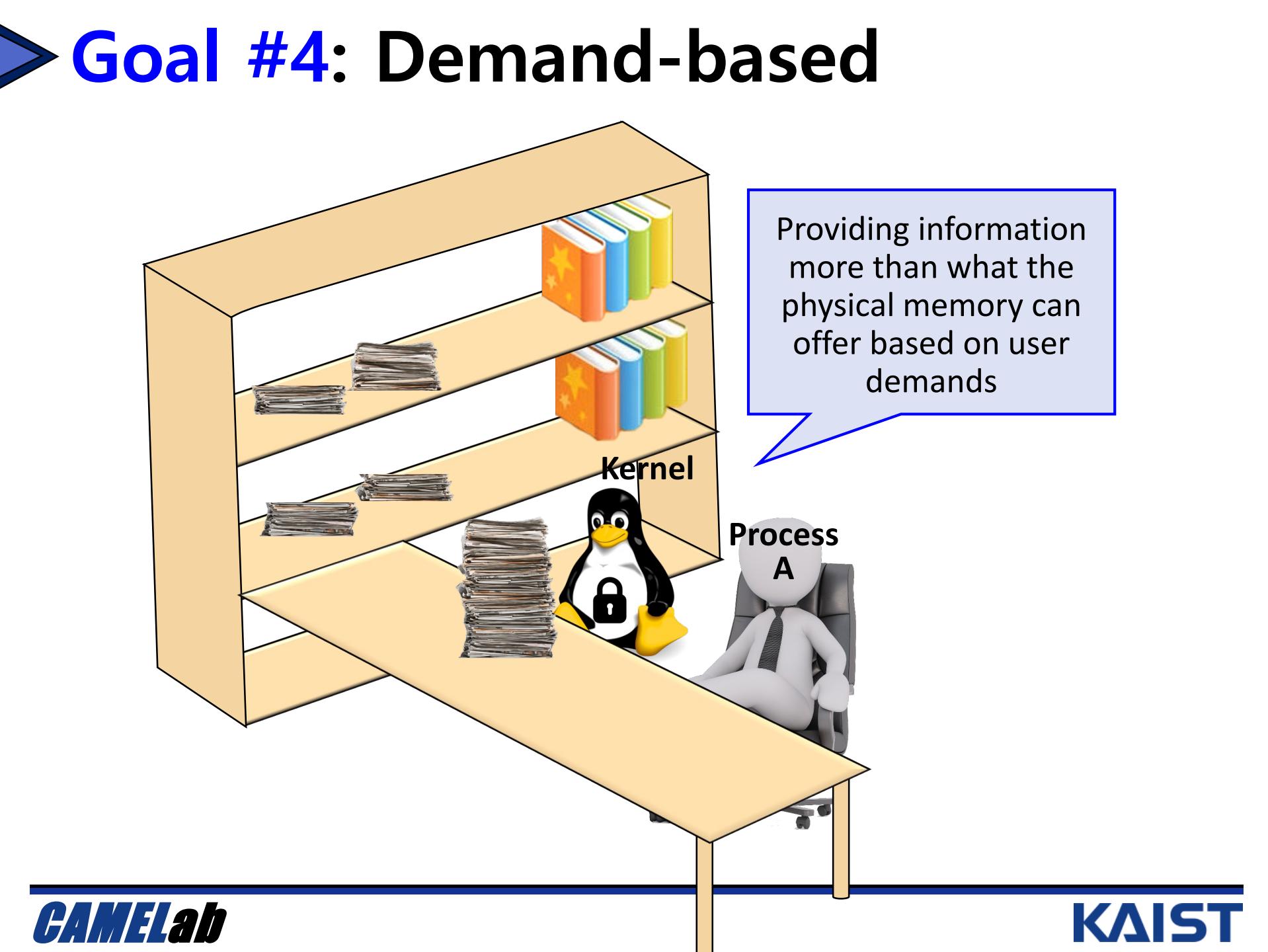
Goal #2: Protection and Isolation

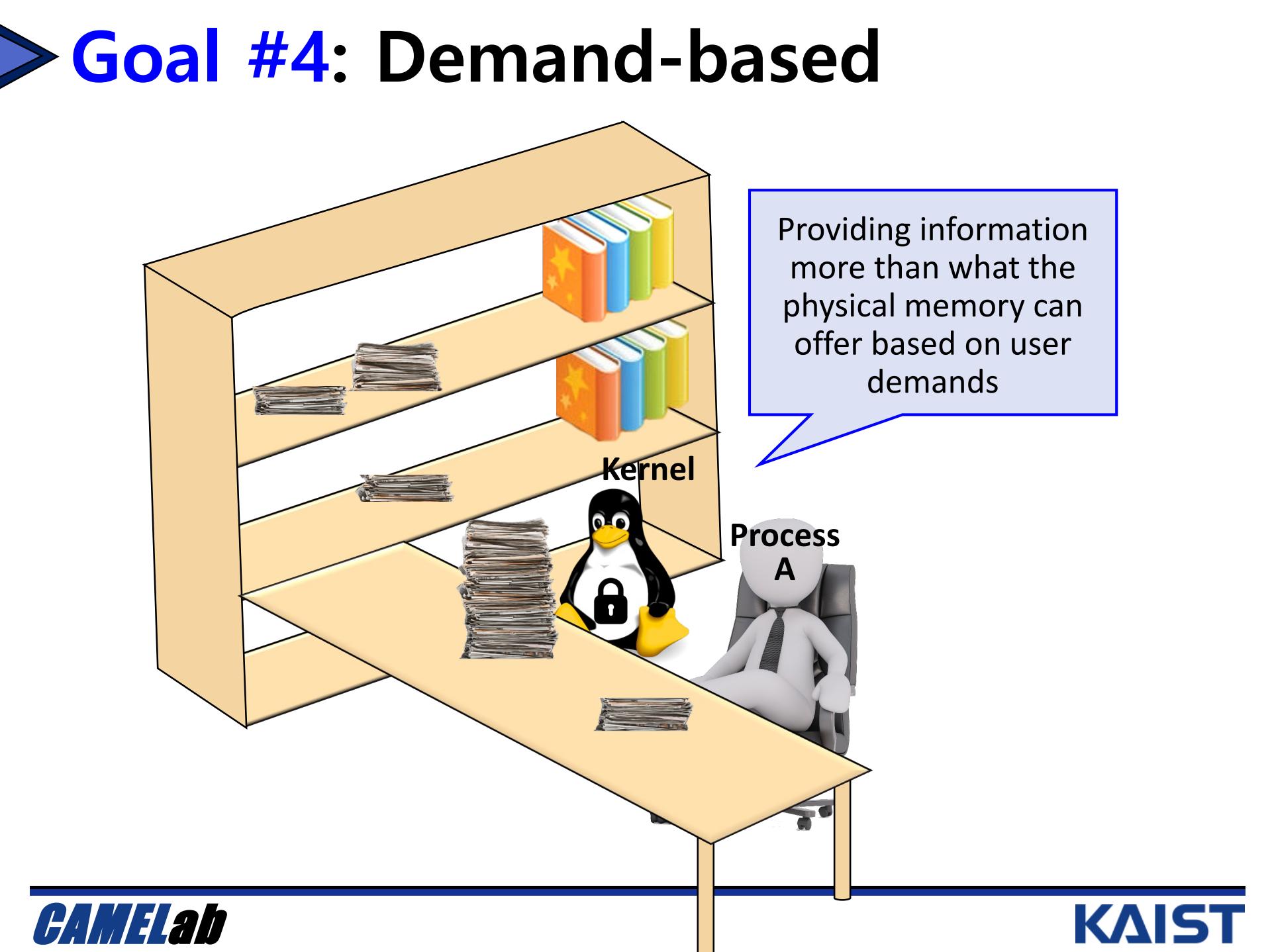


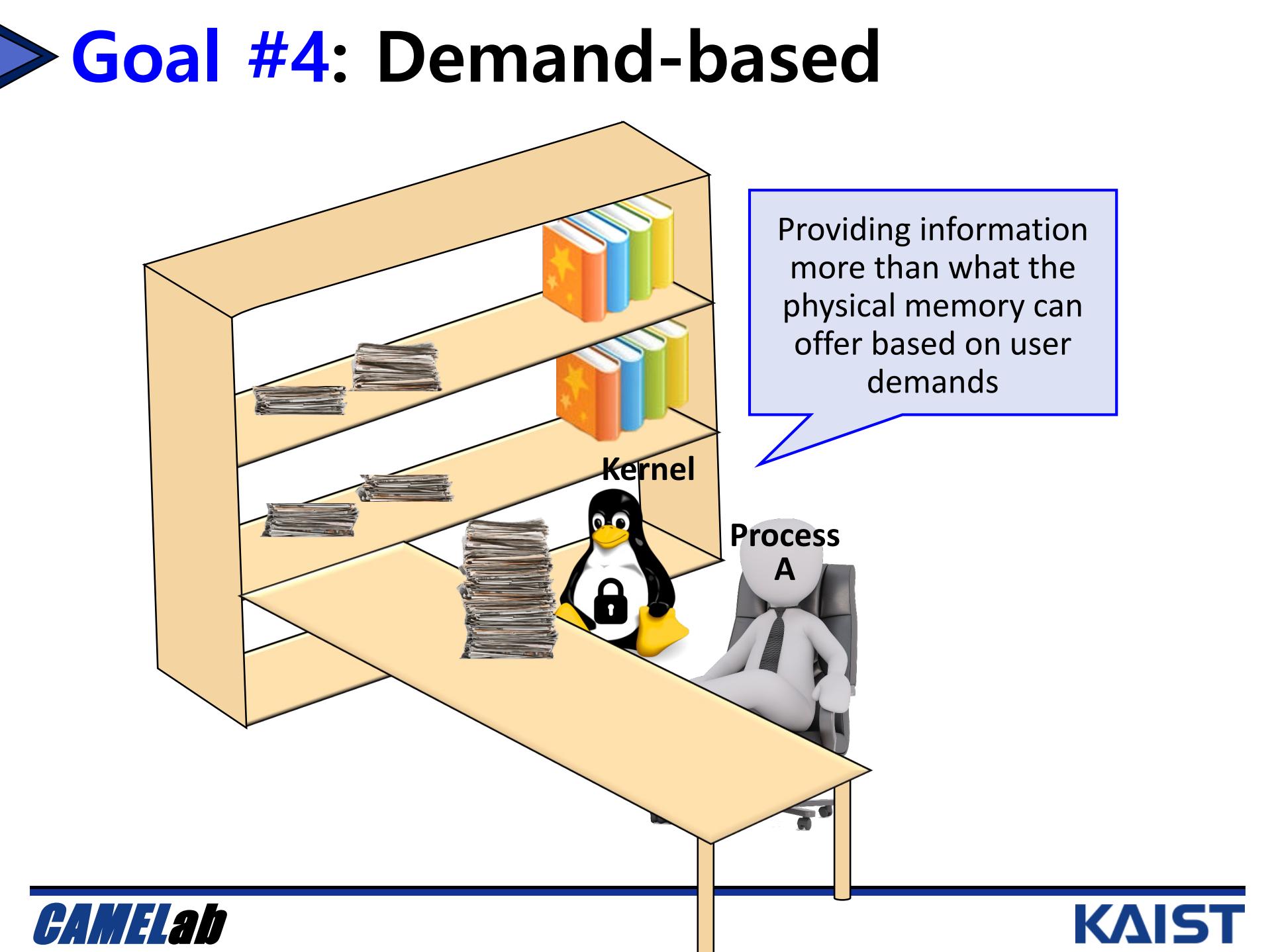


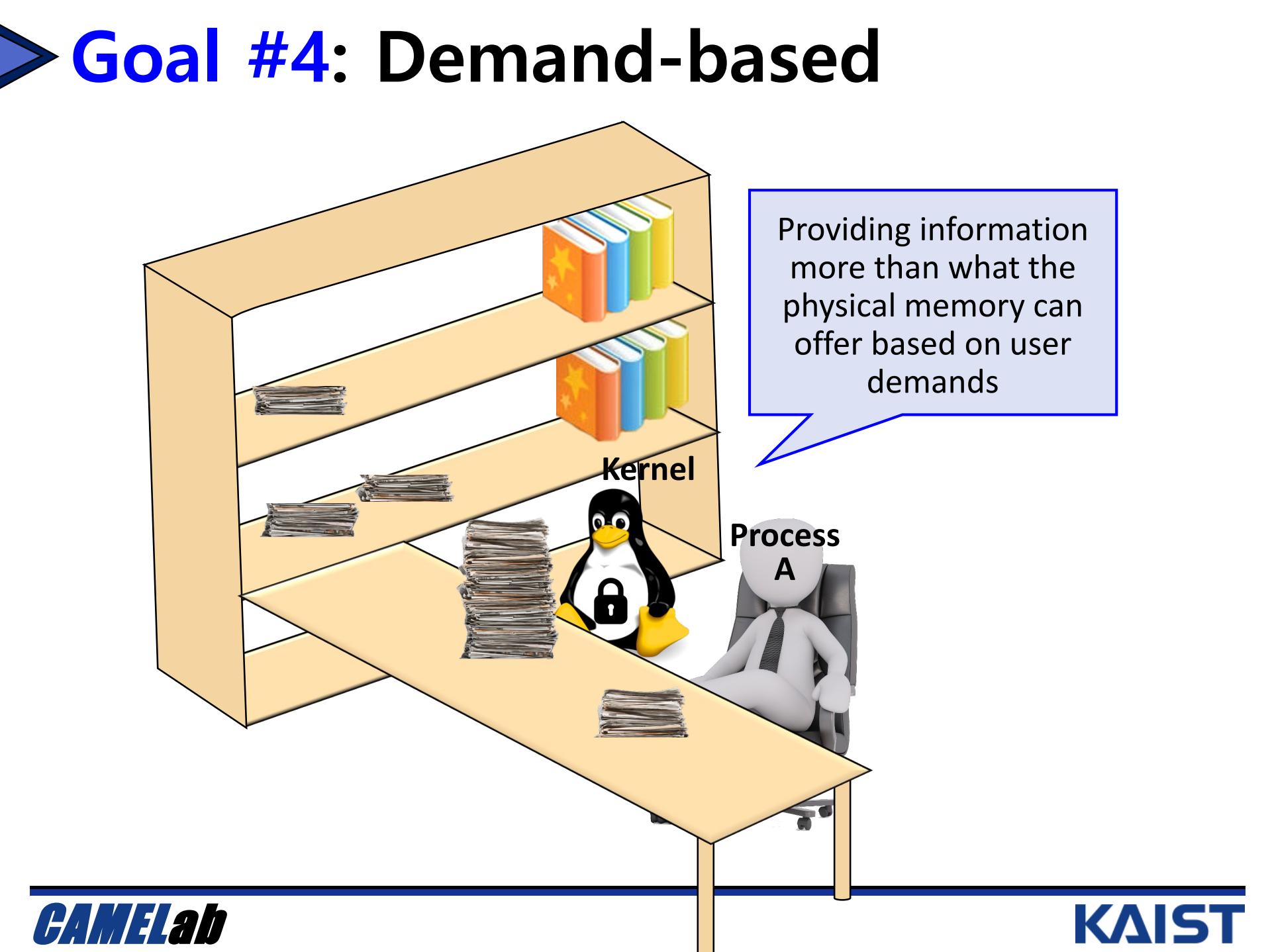
Goal #3: Shared Memory

Sharing some information such as code segments with others (dynamic libraries)



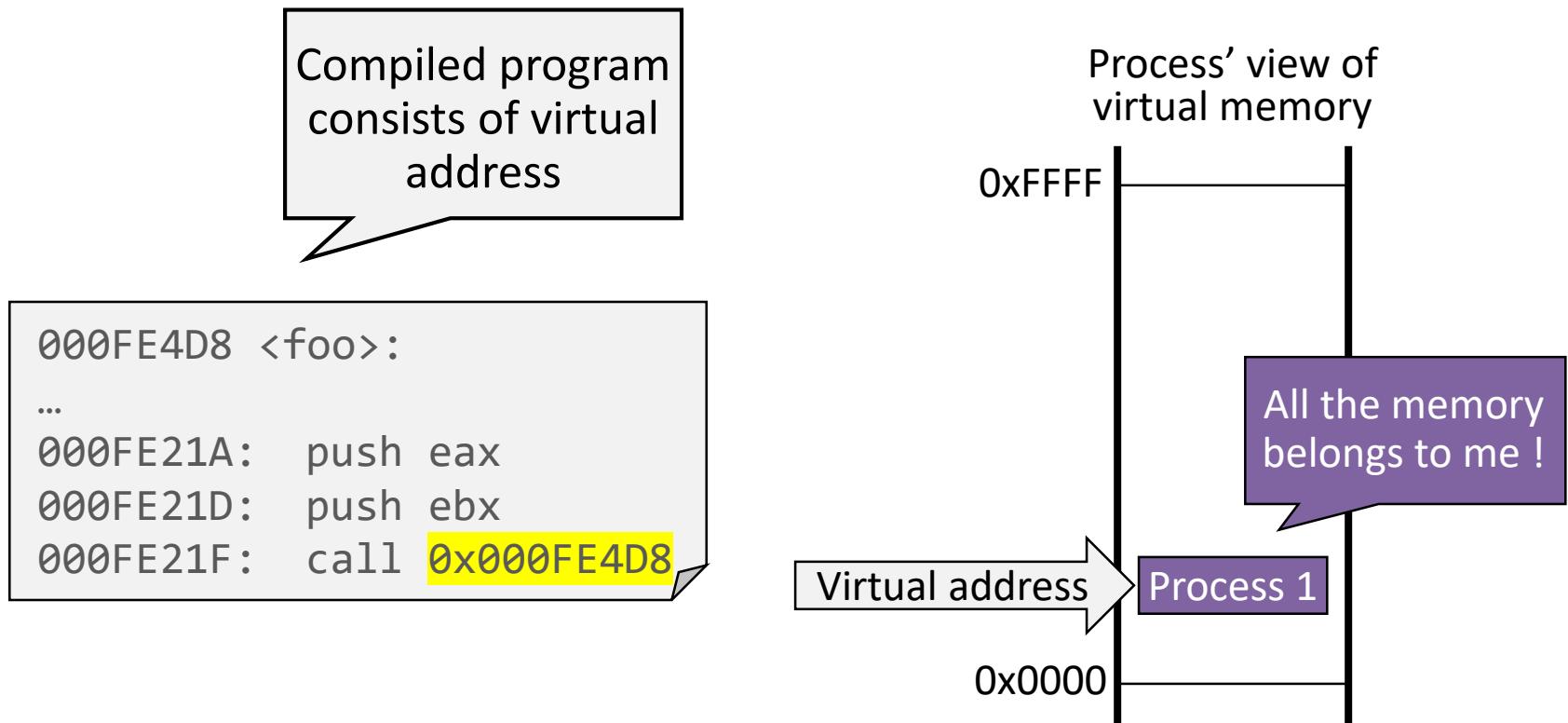






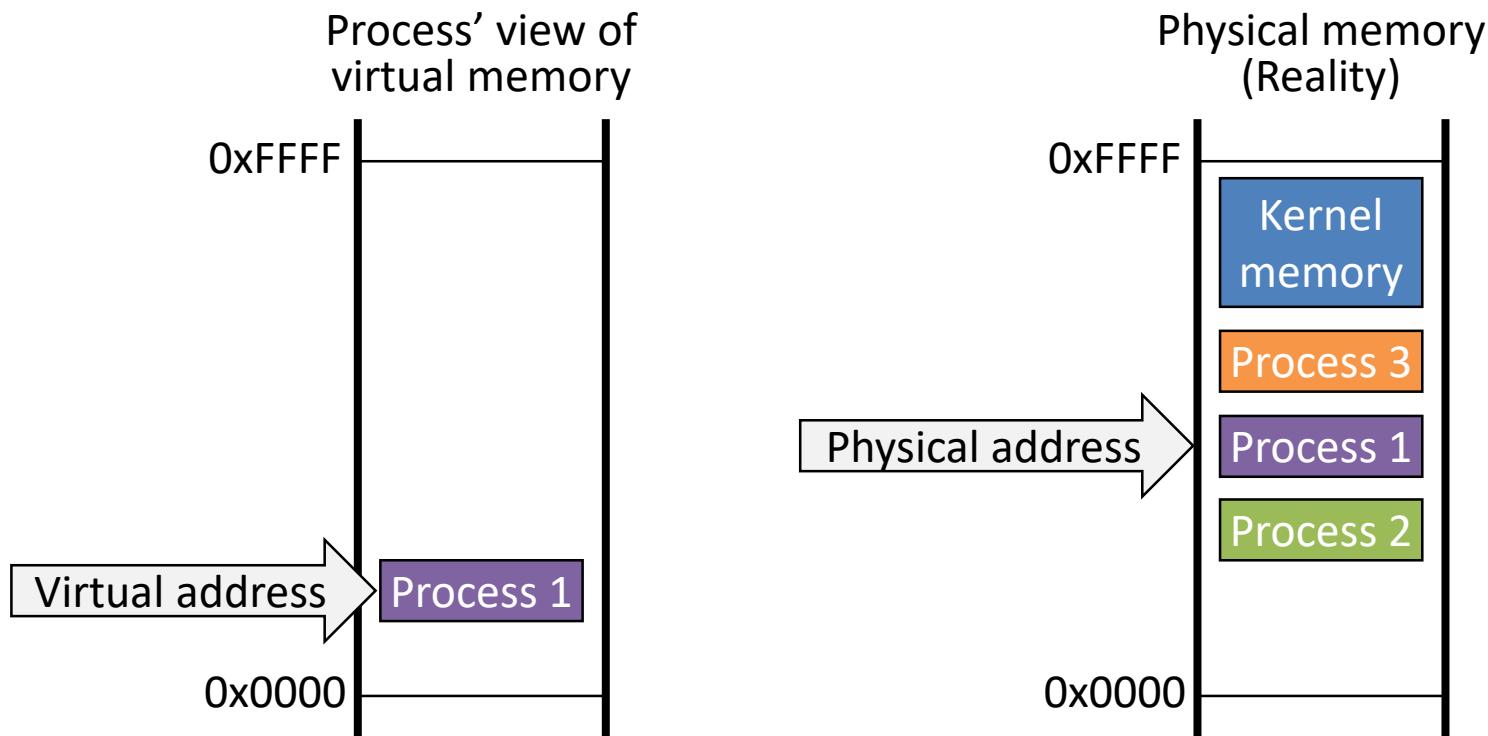
Gap Between Virtual to Physical

- What do you exactly mean by virtual memory?



Gap Between Virtual to Physical

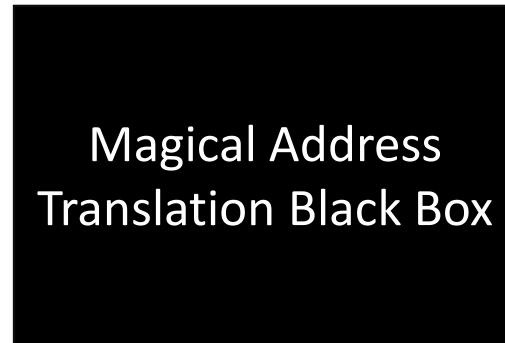
- What do you exactly mean by virtual memory?



Gap Between Virtual to Physical

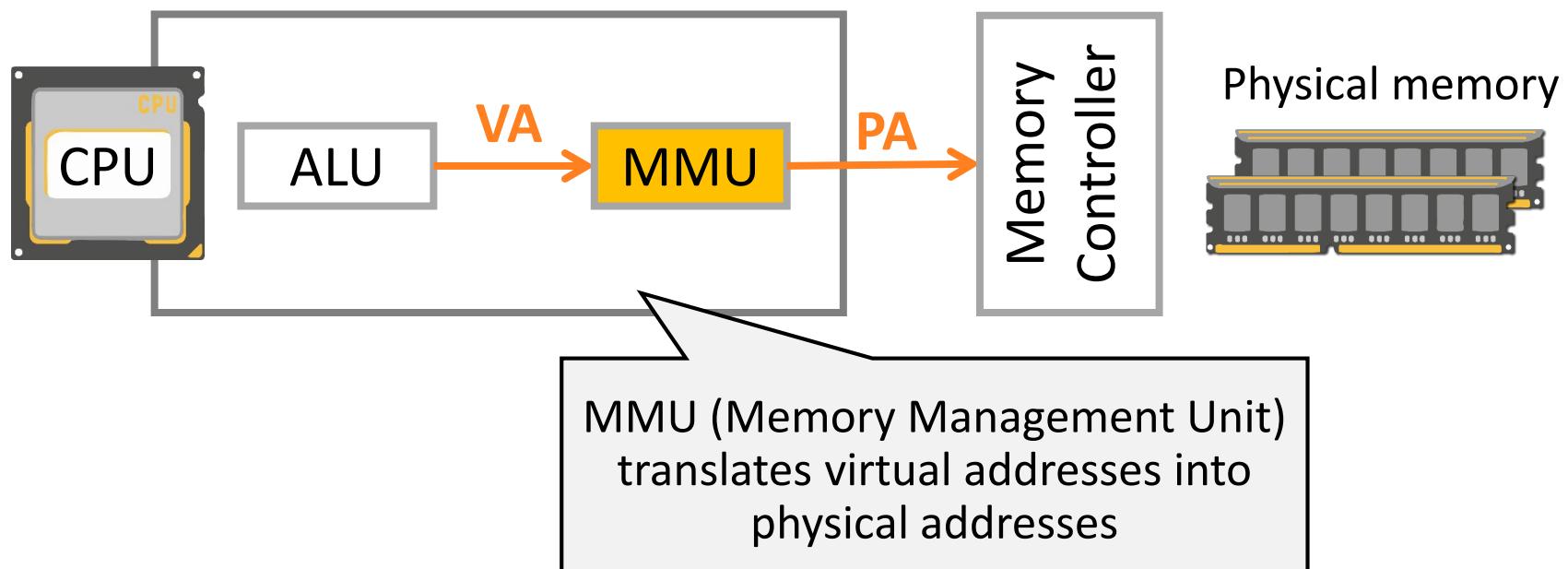
- What do you exactly mean by virtual memory?

In a system with virtual memory, each memory access must be translated



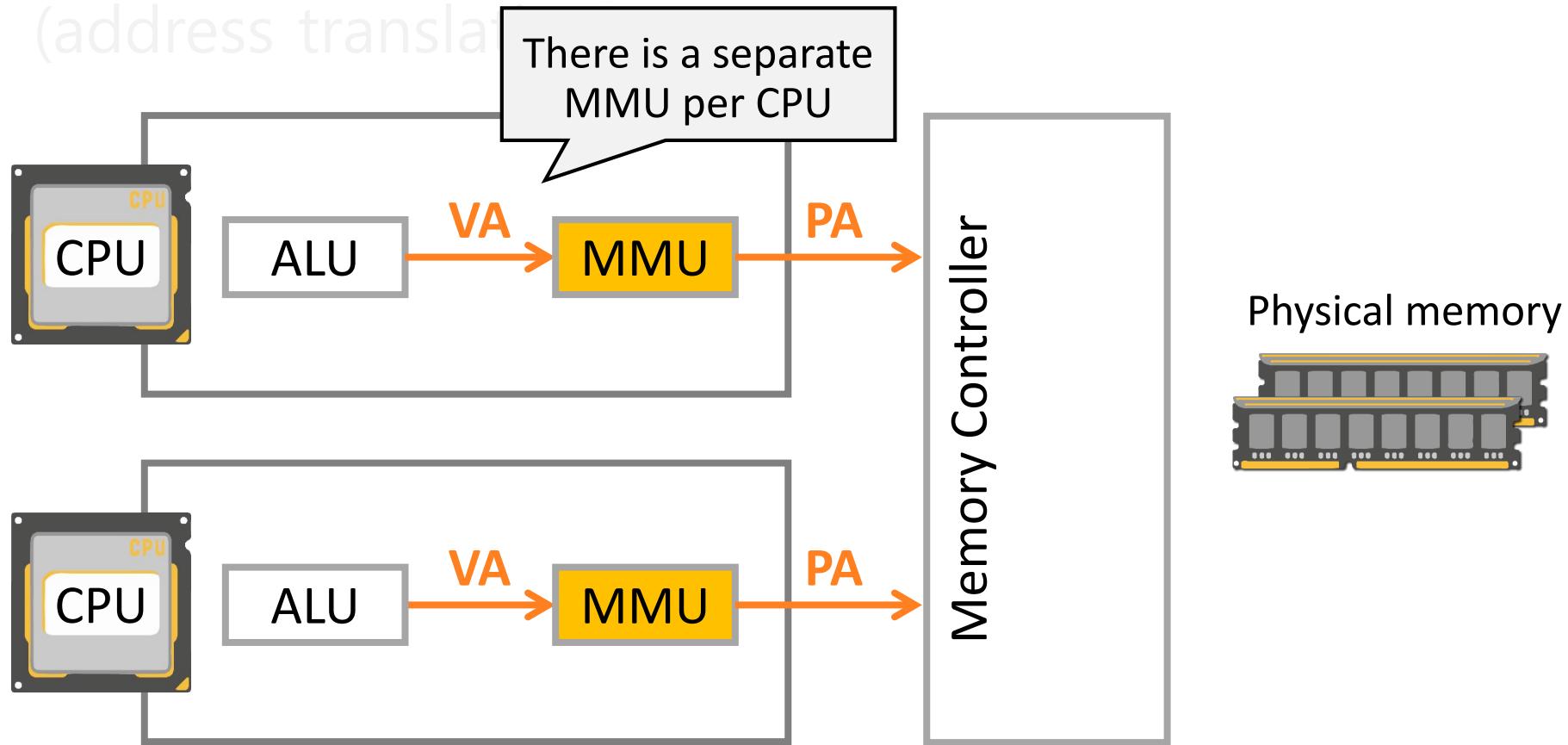
Virtual Memory System (HW)

- It requires hardware support
- CPU vendors facilitates the virtual memory system (address translation)



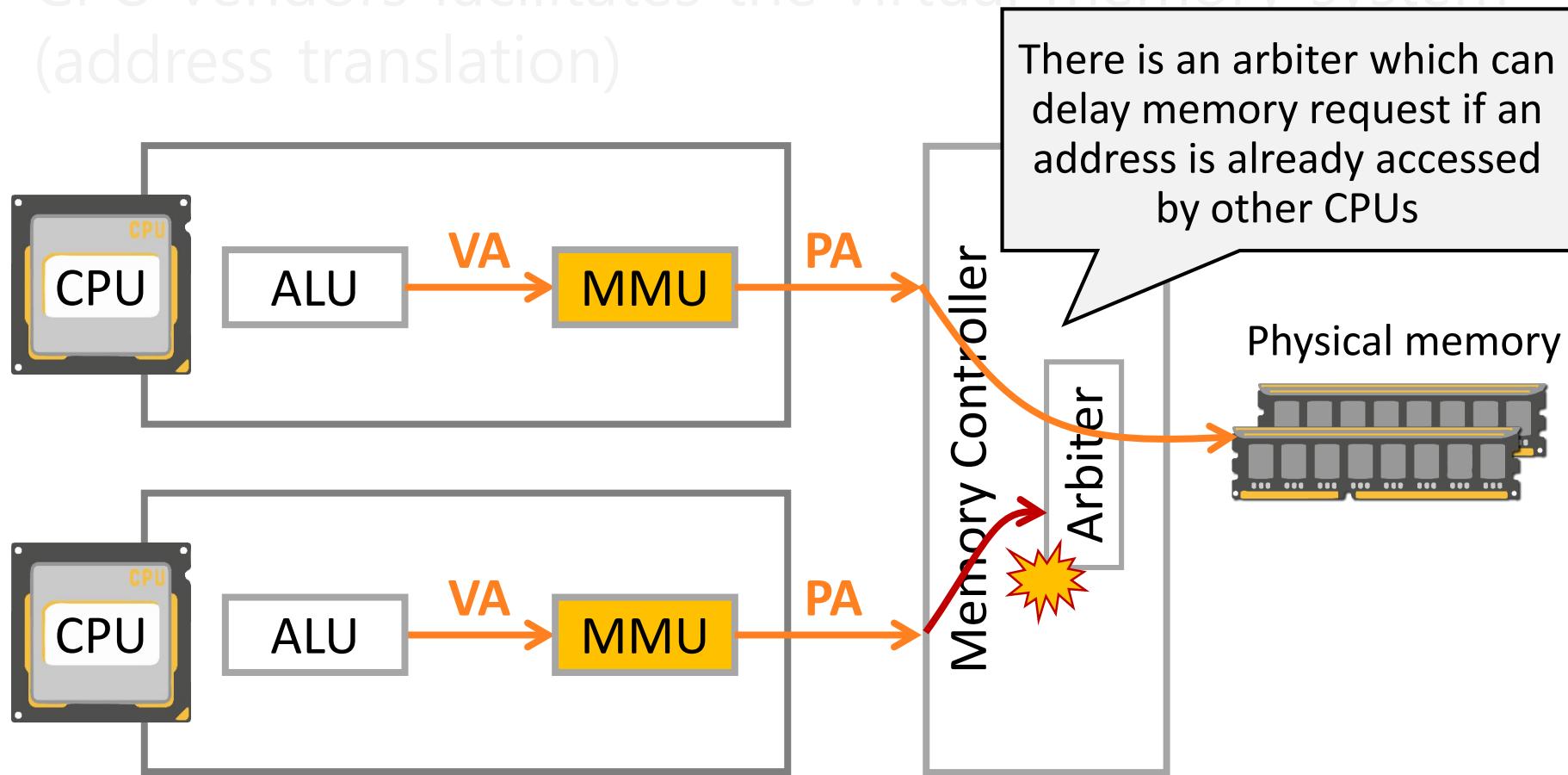
Virtual Memory System (HW)

- It requires hardware support
- CPU vendors facilitates the virtual memory system
(address translation)



Virtual Memory System (HW)

- It requires hardware support
- CPU vendors facilitates the virtual memory system
(address translation)

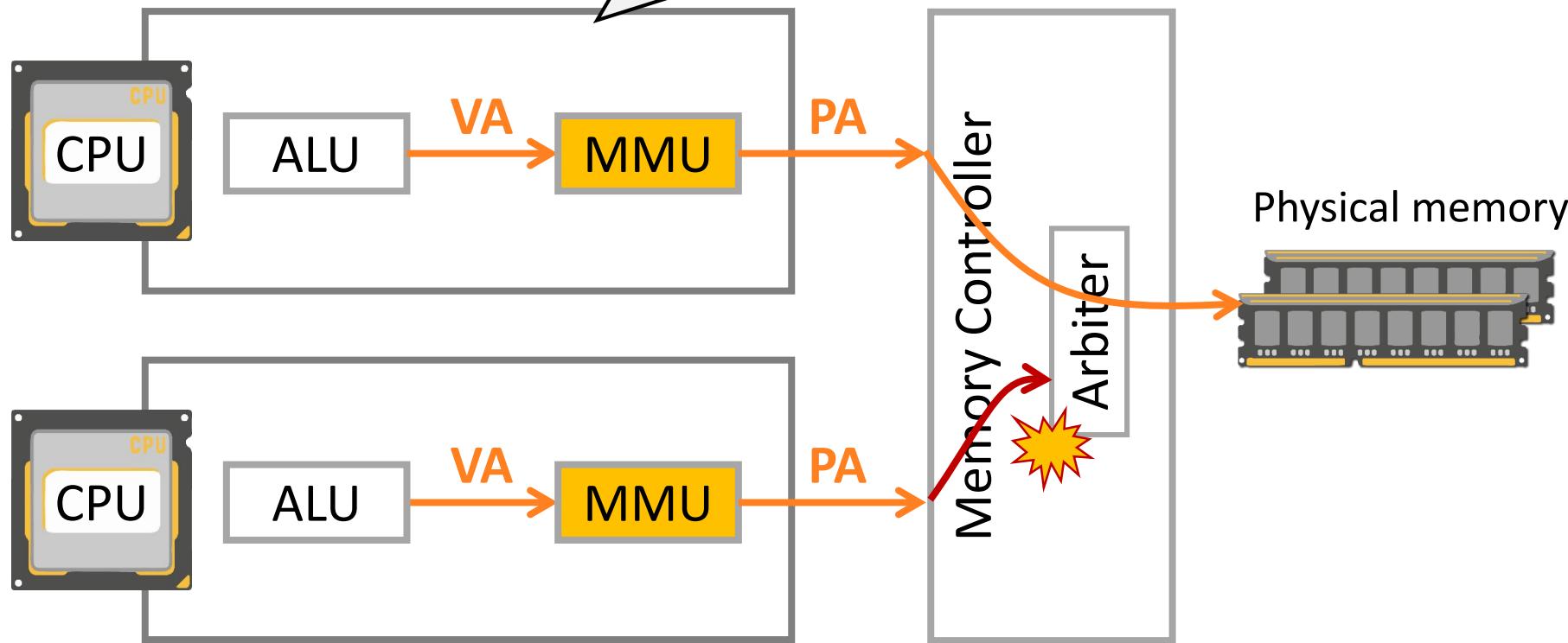


Virtual Memory System (HW)

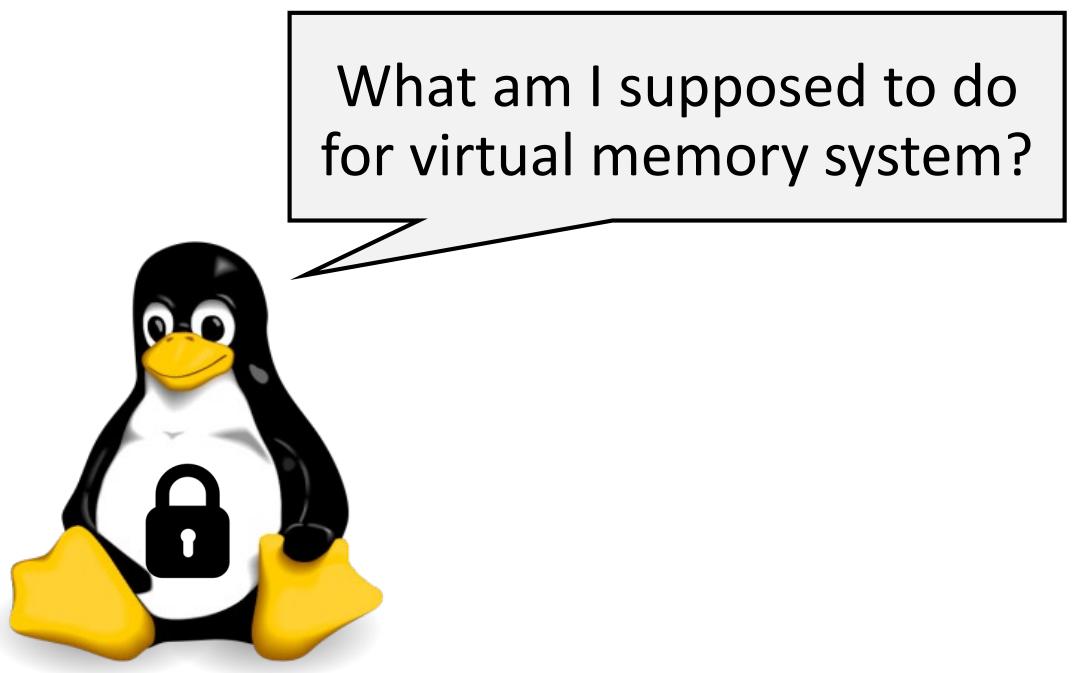
- It requires hardware support

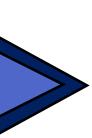
- CPU vendors facilitate
(address translation)

Note that, MMU should cooperate with the OS kernel
for address translation



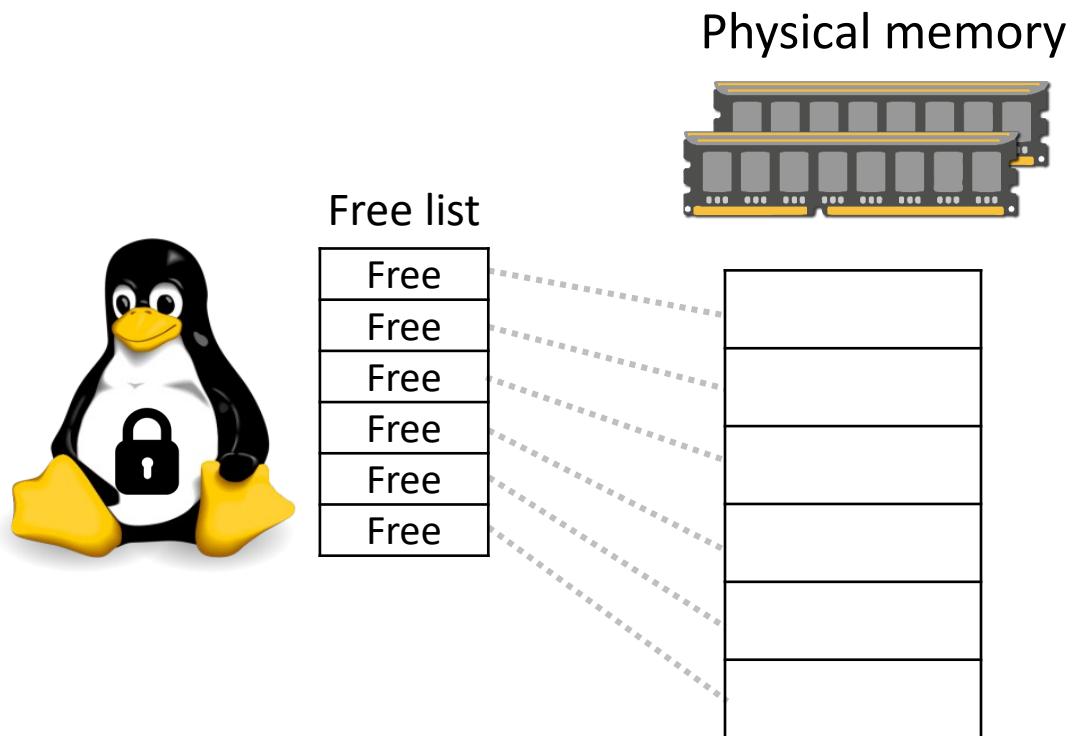
Virtual Memory System (Kernel)

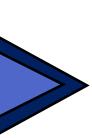




#1: Data Structures for Memory

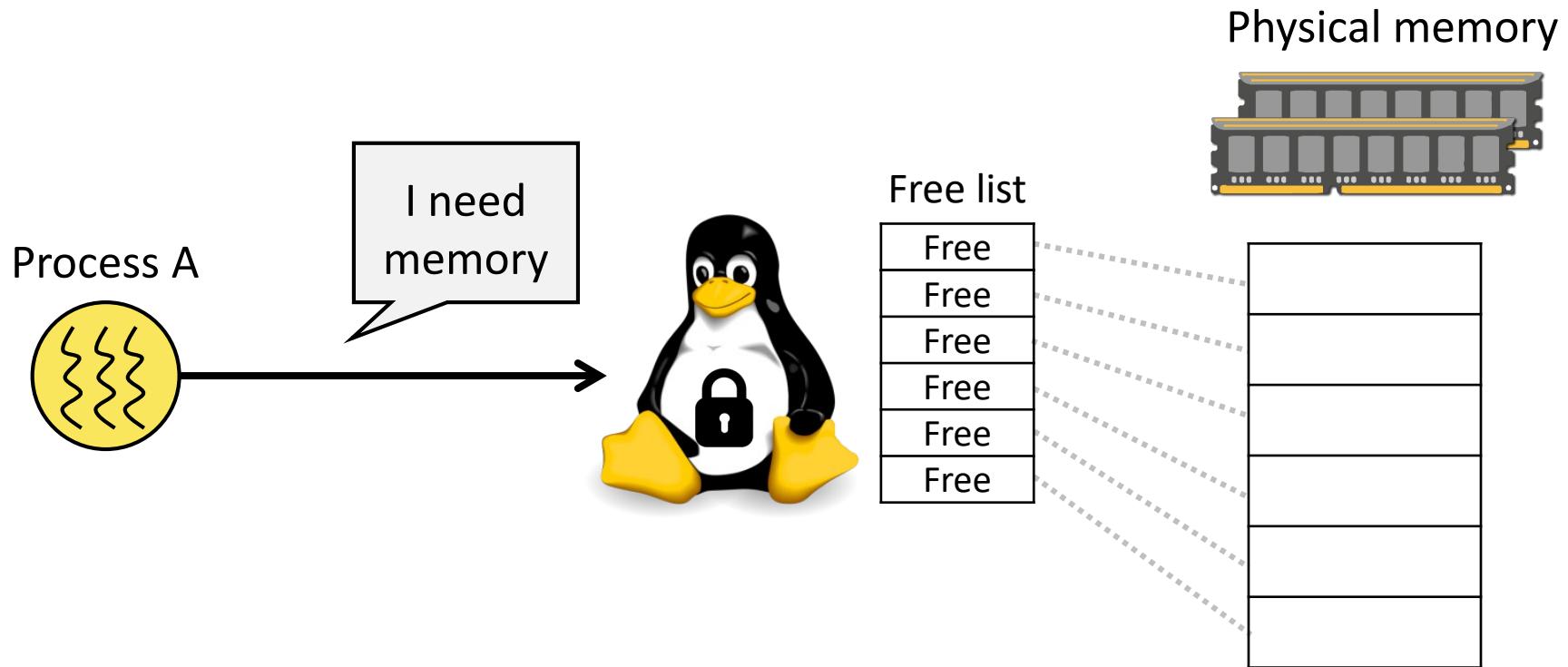
- Linux kernel manages the status of physical memory (occupied/free) and allocates it to the user process based on system demands

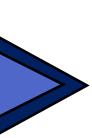




#1: Manage Free List of Memory

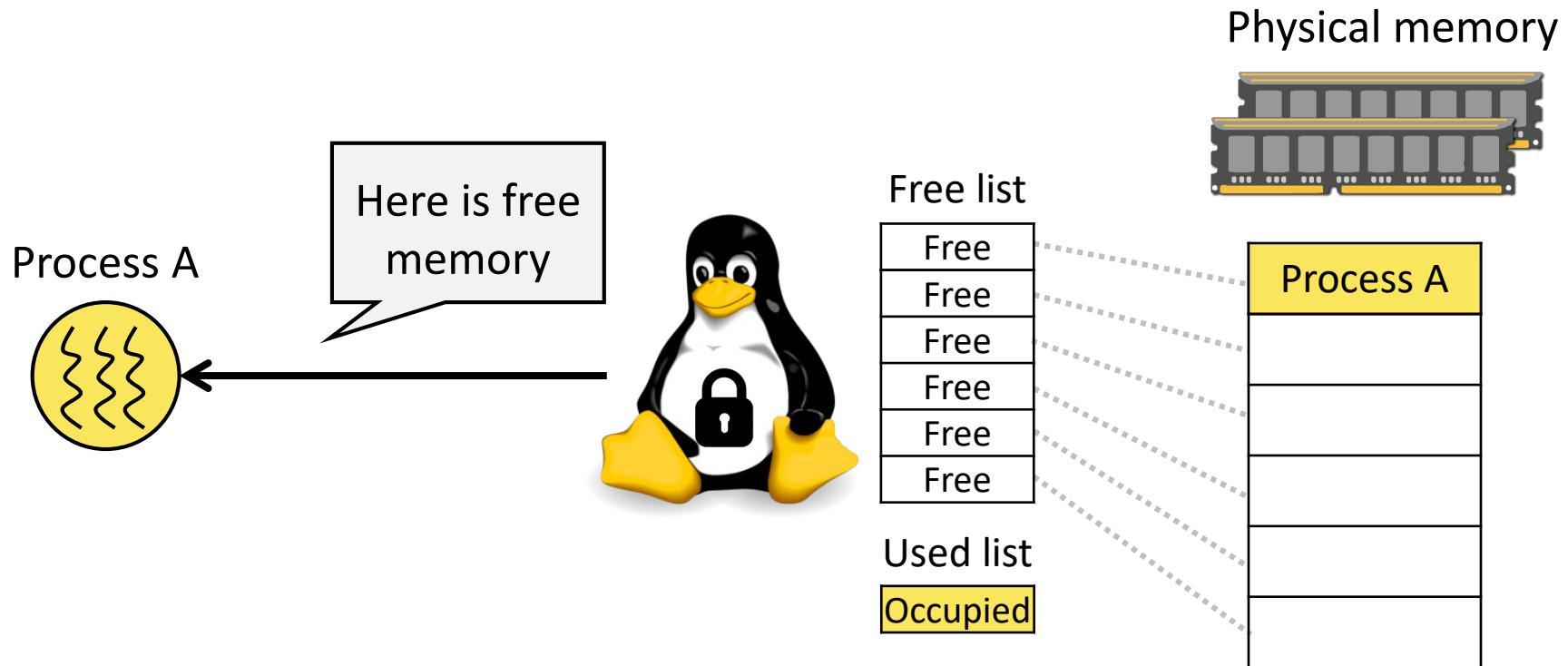
- Linux kernel manages the status of physical memory (occupied/free) and allocates it to the user process based on system demands





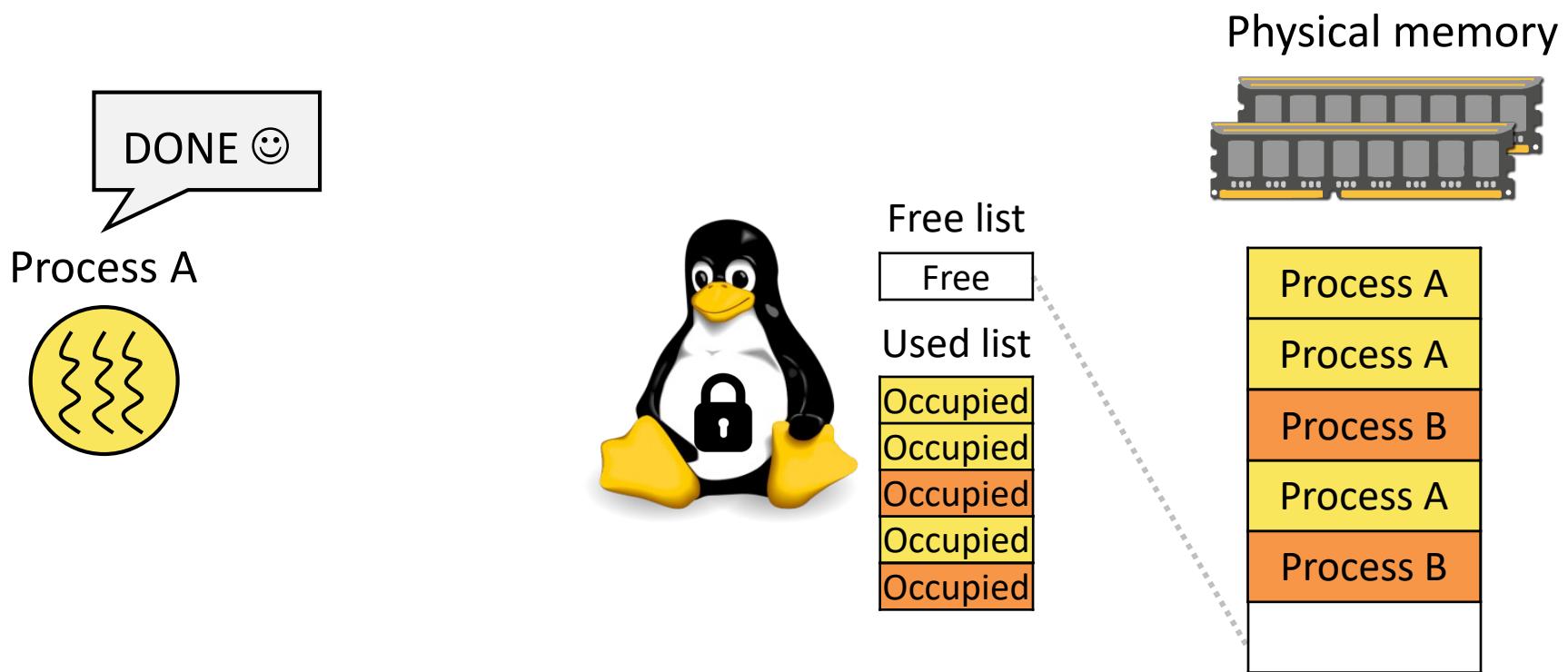
#1: Manage Free List of Memory

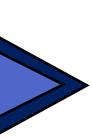
- Linux kernel manages the status of physical memory (occupied/free) and allocates it to the user process based on system demands



#2: Reclaim Memory Space

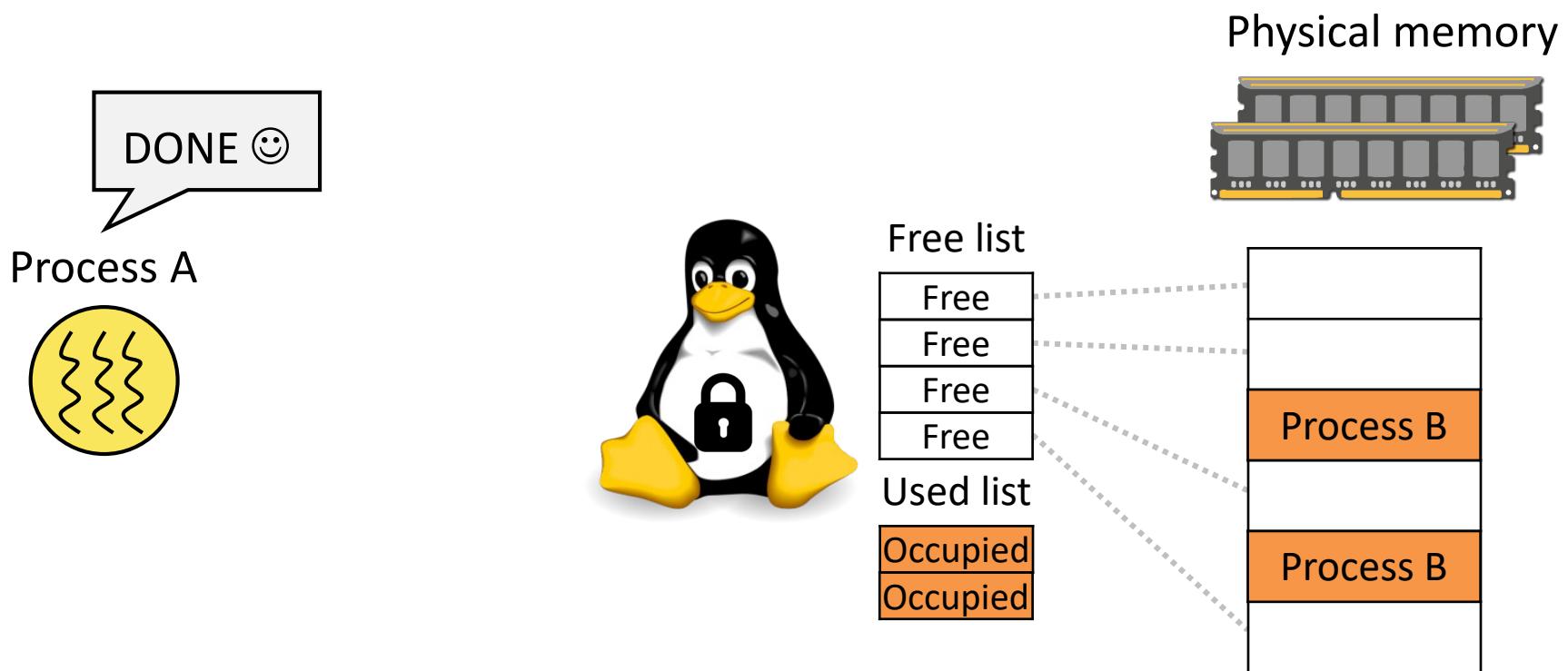
- When a process terminates, OS kernel should reclaim its memory space appropriately

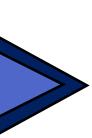




#2: Reclaim Memory Space

- When a process terminates, OS kernel should reclaim its memory space appropriately





#3: Save/Restore Context

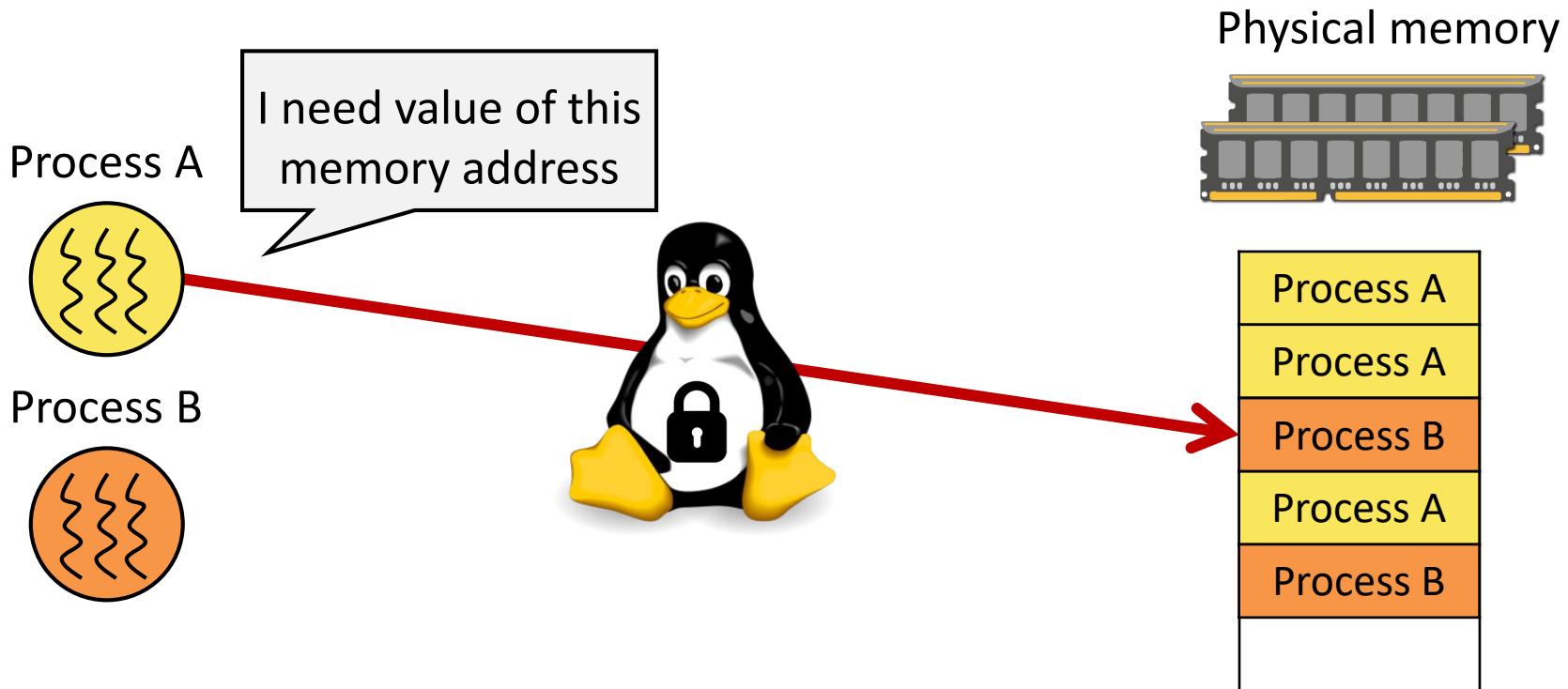
- Virtual memory is given to all processes on the system, but managed by each individually (**per-process** information).
- Thus, OS kernel should save and restore the context related to virtual memory management

Virtual memory context varies based on how MMU to implement it. Let's check the details in later



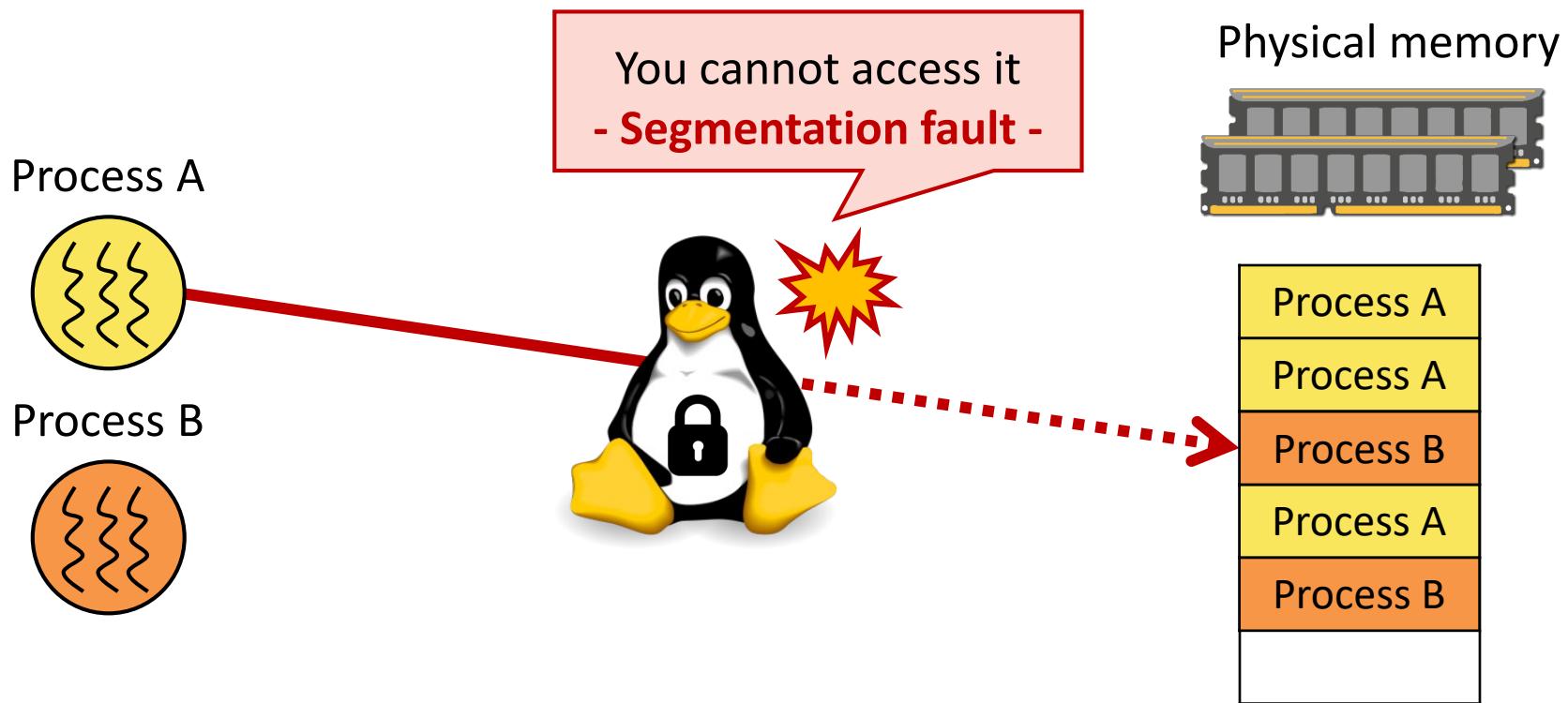
#4: Provide Exception Handler

- OS requires protecting the memory spaces against the accesses who don't have a permission
- If there is, OS should handle the case as an exception appropriately



#4: Provide Exception Handler

- OS requires protecting the memory spaces against the accesses who don't have a permission
- If there is, OS should handle the case as an exception appropriately





#4: Provide Exception Handler

- Segmentation Fault
 - If you try to read/write memory outside memory space assigned to your process

A screenshot of a debugger interface. On the left is a code editor with C++ code:

```
1 #include <QApplication>
2
3 int main(int argc, char *argv[])
4 {
5     QApplication a(argc, argv);
6     return a.exec();
7 }
```

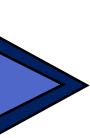
On the right is a "Signal Received" dialog box:

The inferior stopped because it received a signal from the operating system.
Signal name: SIGSEGV
Signal meaning: Segmentation fault

OK

```
gcc release.c -o release && ./release
executing code...
loading wave data
artist : telectart
bpm : 160.00
peak : -0.00 db
size : 29.63 mb
audio data length: 02:56 min
key : f minor -- initialising audio playback...
```

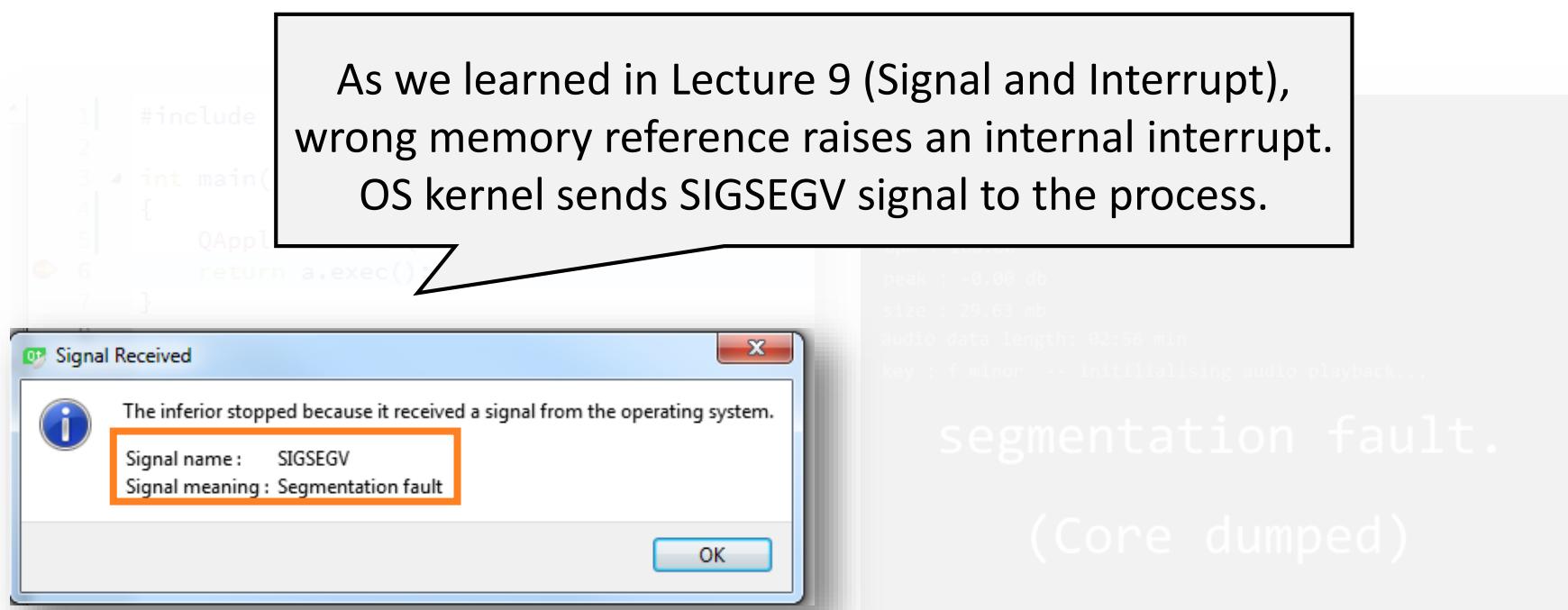
segmentation fault.
(Core dumped)



#4: Provide Exception Handler

- Segmentation Fault
 - If you try to read/write memory outside memory space assigned to your process

As we learned in Lecture 9 (Signal and Interrupt), wrong memory reference raises an internal interrupt. OS kernel sends SIGSEGV signal to the process.



Virtual Memory Implementation

Mapping is everywhere

Roadmap of MMU

We will discuss three representative MMU approaches

- How does it work?
- What features does it offer?
- What would be the limits?



Base and Bound

Segmentation

Paging

Old, simple, limited functionality

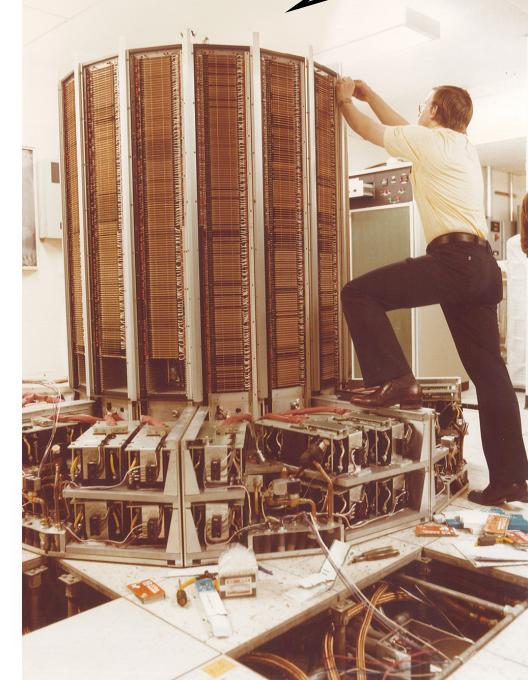
Modern, complex, lots of functionality

Base-and-Bound

Base and
Bound

Segmentation

Old, simple, limited
functionality

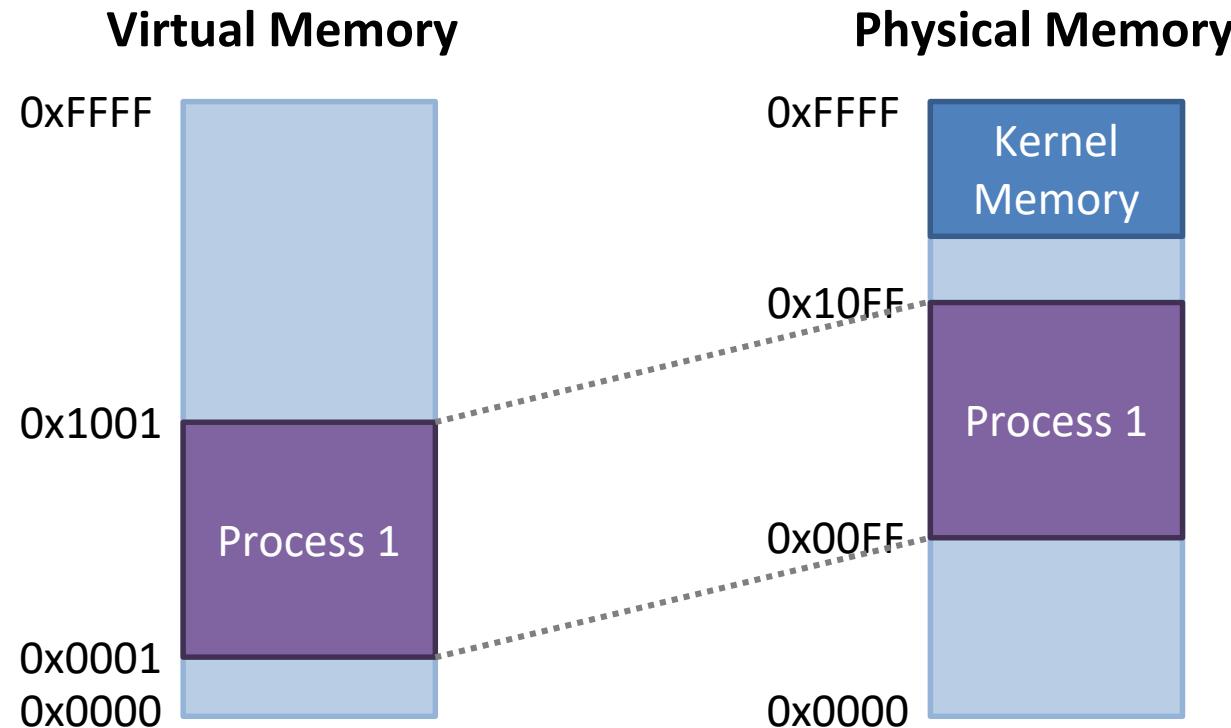


First built in
the Cray1

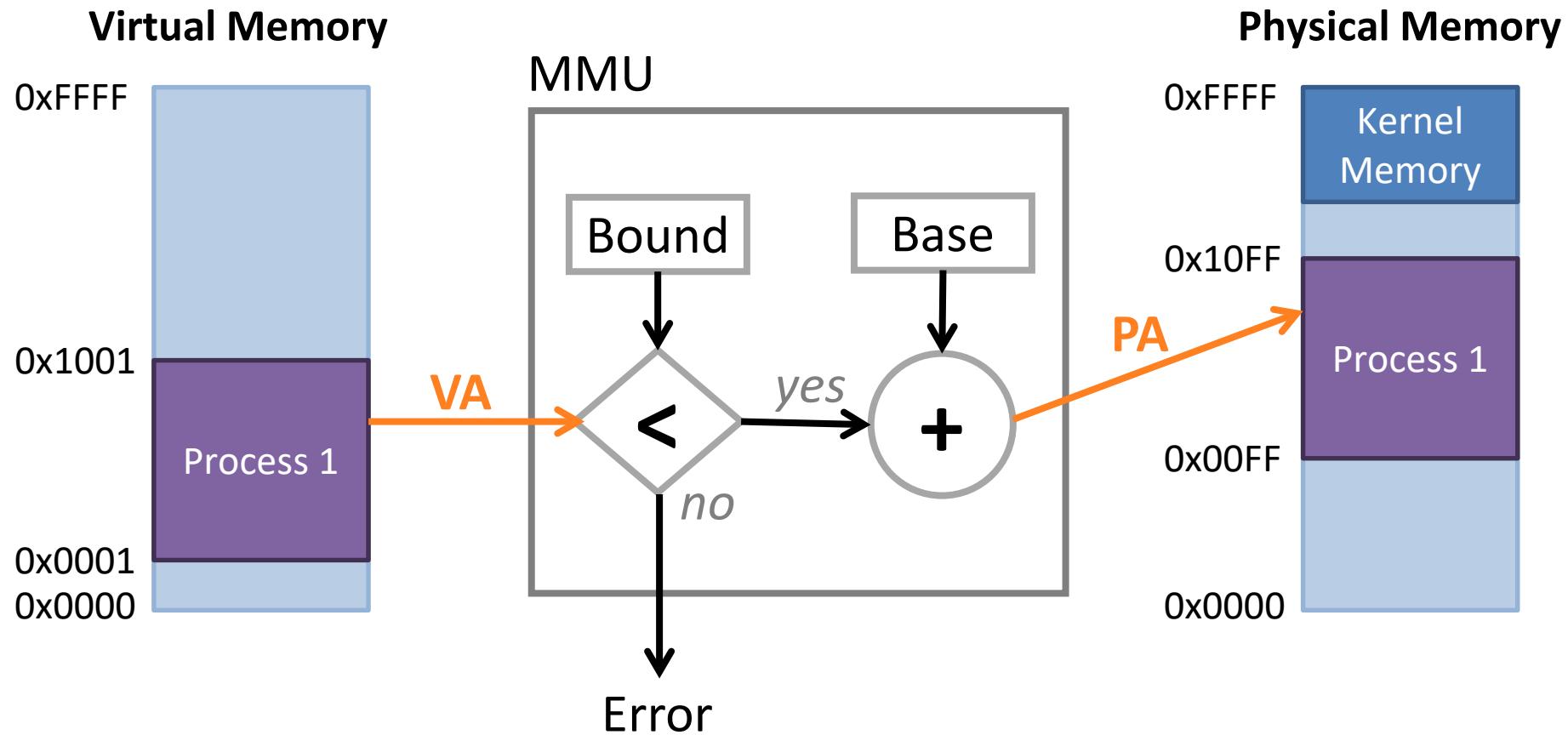
Modern, complex,
lots of functionality

Concept of Base-and-Bound

- A simple mechanism for address translation
- Maps a chunk of physical memory to a chunk of logical memory
(i.e., the check is contiguous memory region)



HW Support



HW Support

CPU facilitates two **protected registers**,
BASE and BOUND in addition to the
general purpose registers (e.g., EAX, etc.)

Instructions with Ring 0 can modify
theses registers for the memory
protection

Bound

Base

CPU will have multiple tasks
(processes/threads), right? How
can you offer the bound and base
for each task (as they need to
have their own memory space)



0x1001
Process 1
VA
0x0001
0x0000

Kernel
Memory
0x10FF
Process 1
0x00FF
0x0000

Error

HW Support

CPU facilitates two **protected registers**,
BASE and BOUND in addition to the
general purpose registers (e.g., EAX, etc.)

Instructions with Ring 0 can modify
theses registers for the memory
protection

Bound

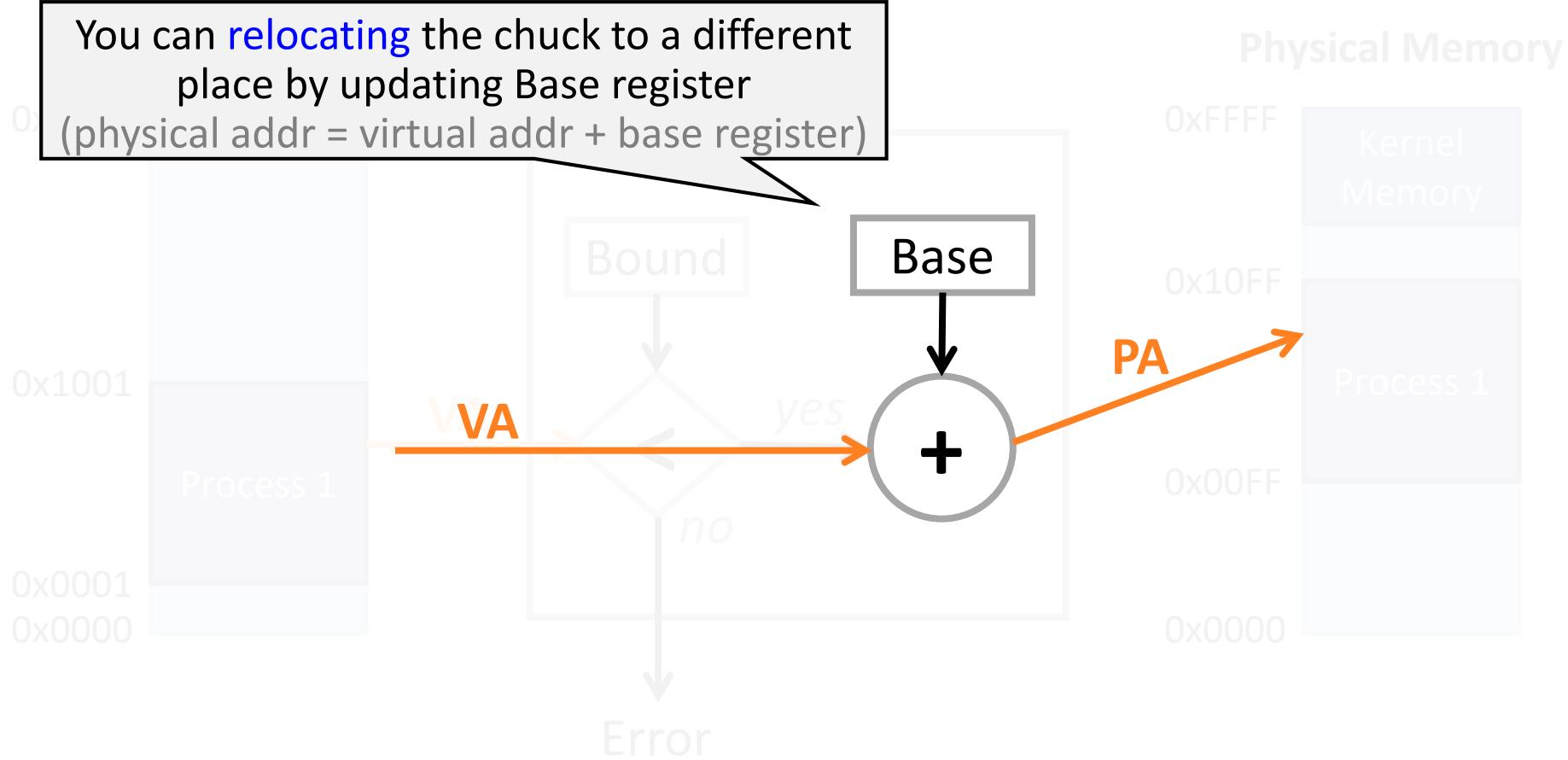
Base

That's not a big deal. We
can consider BASE and
BOUND as per-task
context and deal with
them in **context**
switching



Error

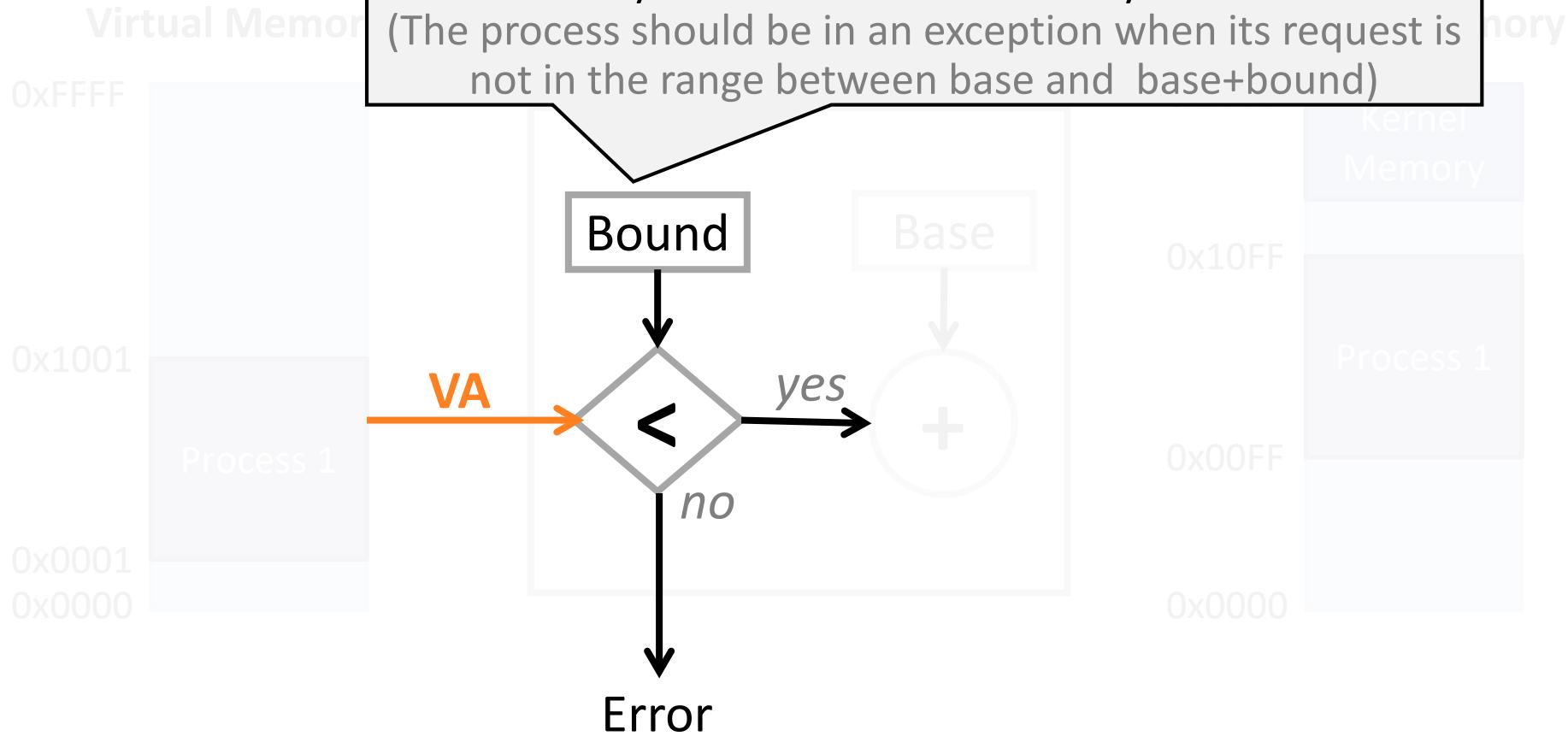
Base Register



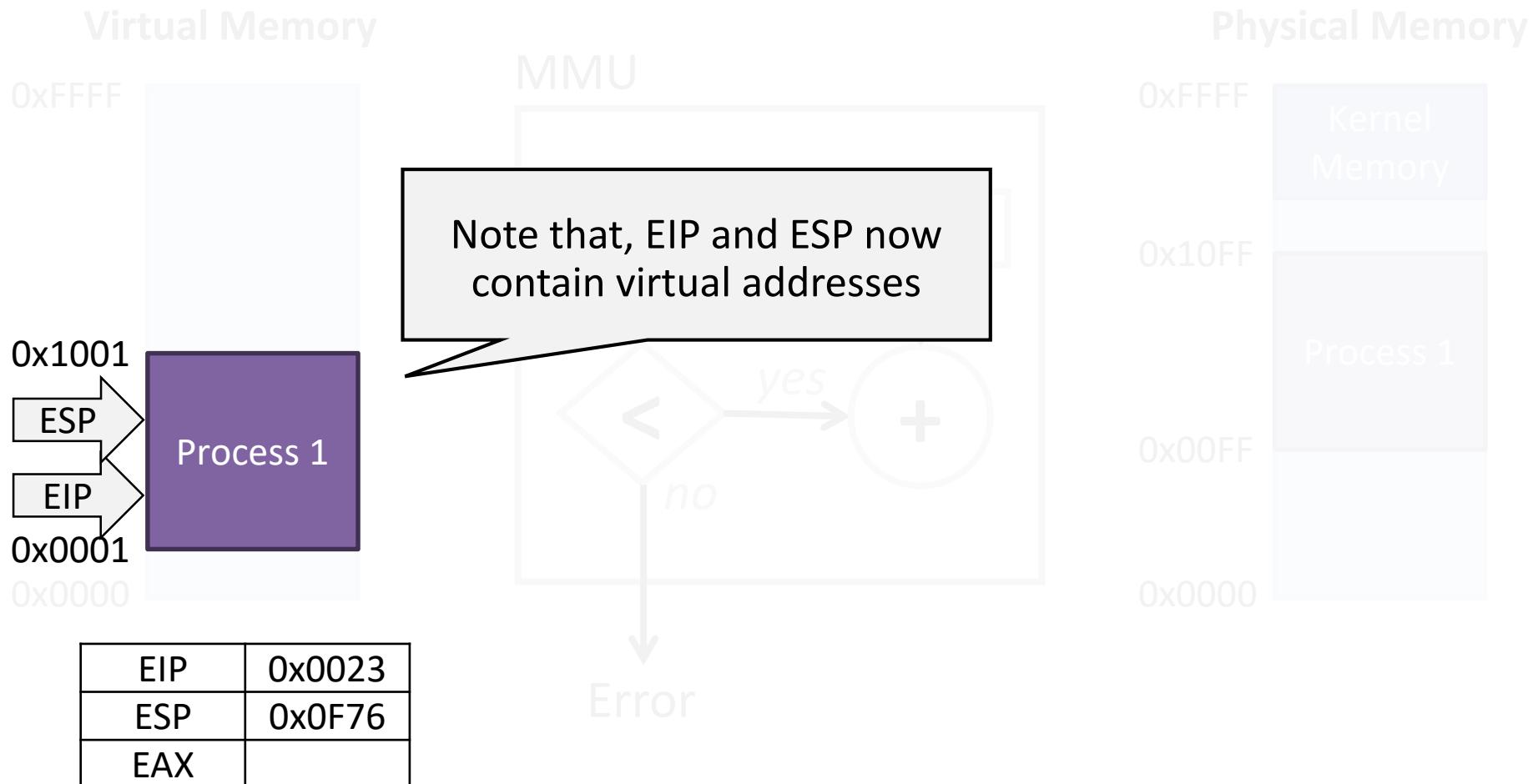
Bound Register

Using Bound, we can **protect** a wrong access against going beyond the allocated memory chunk

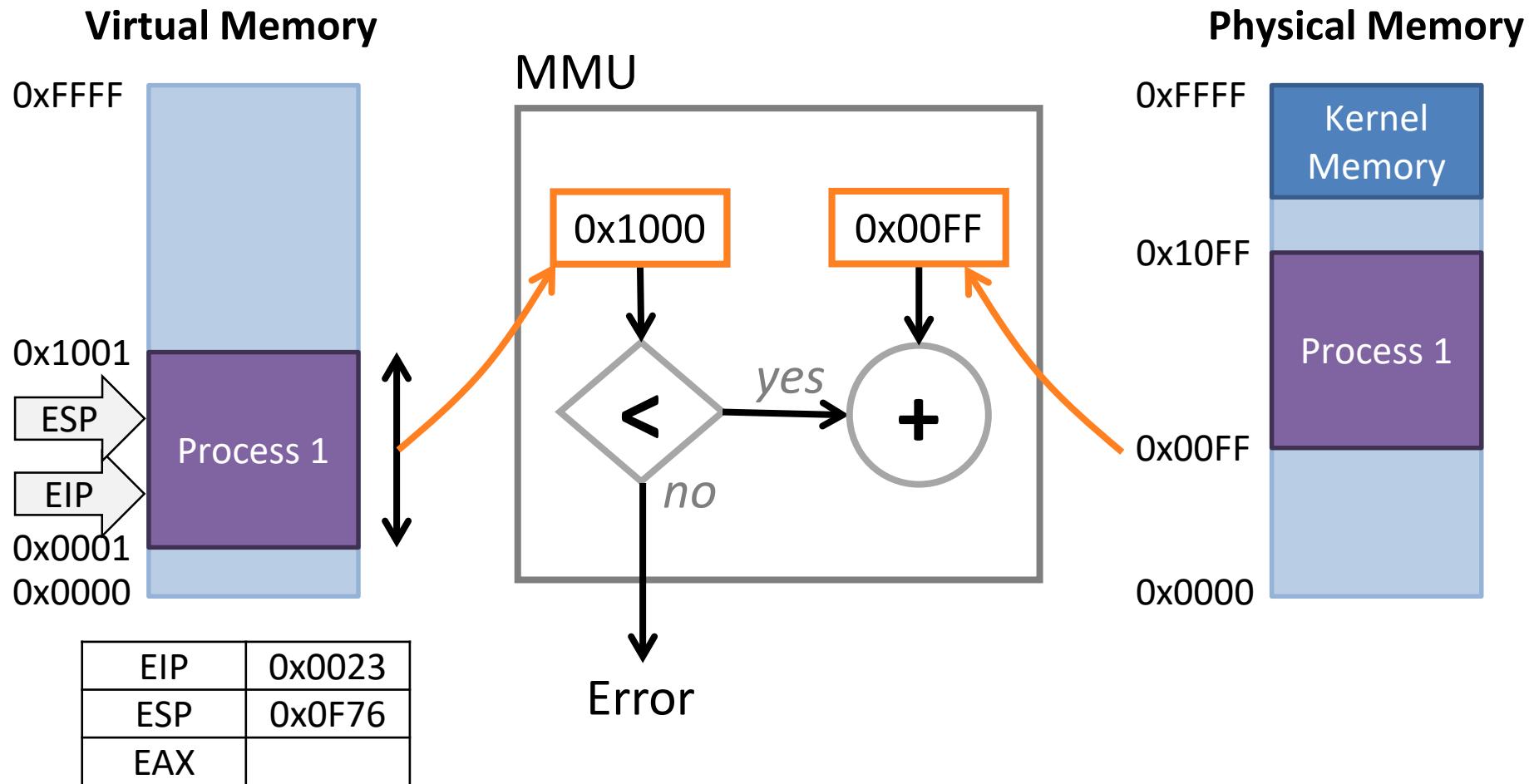
(The process should be in an exception when its request is not in the range between base and base+bound)



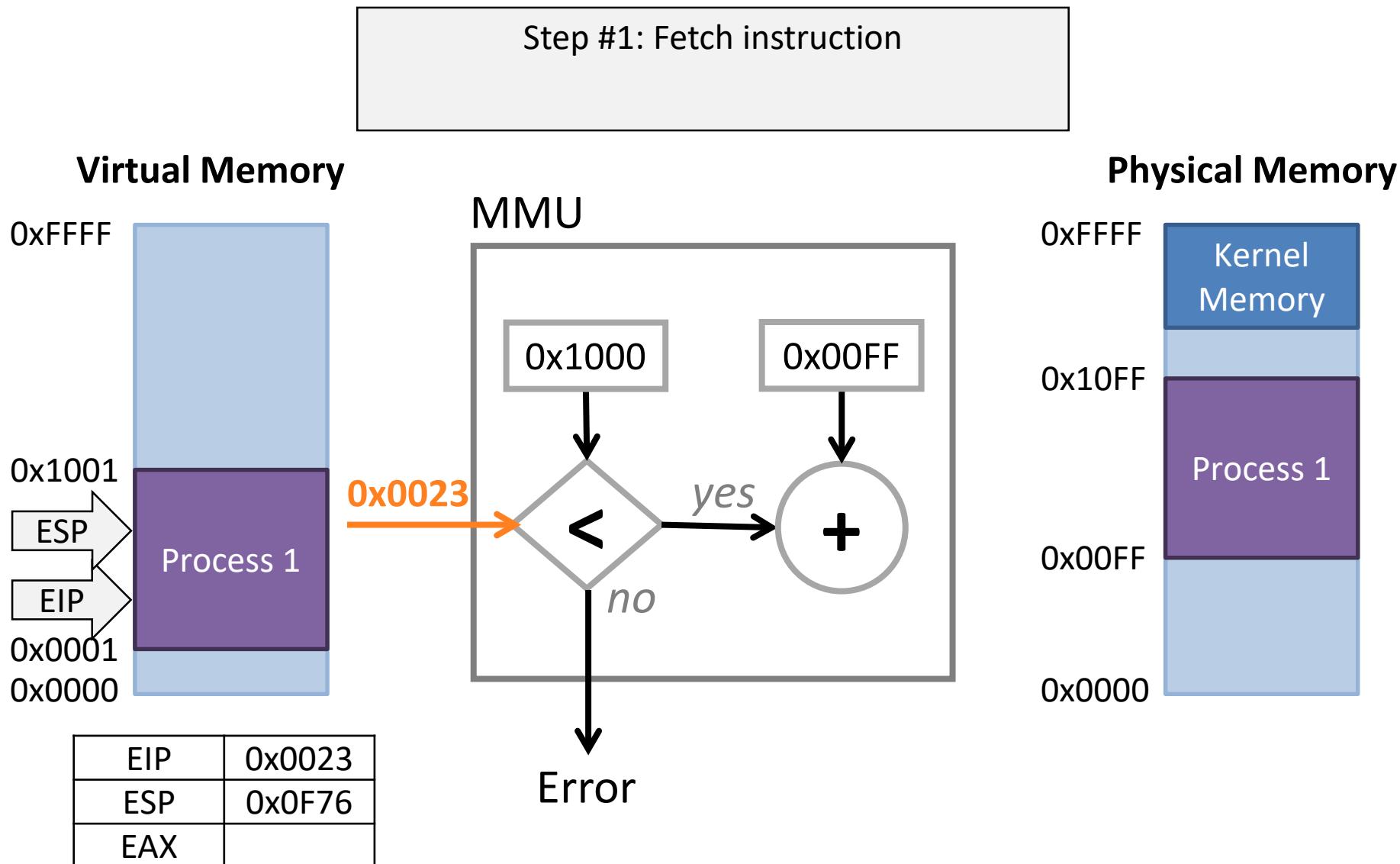
Example



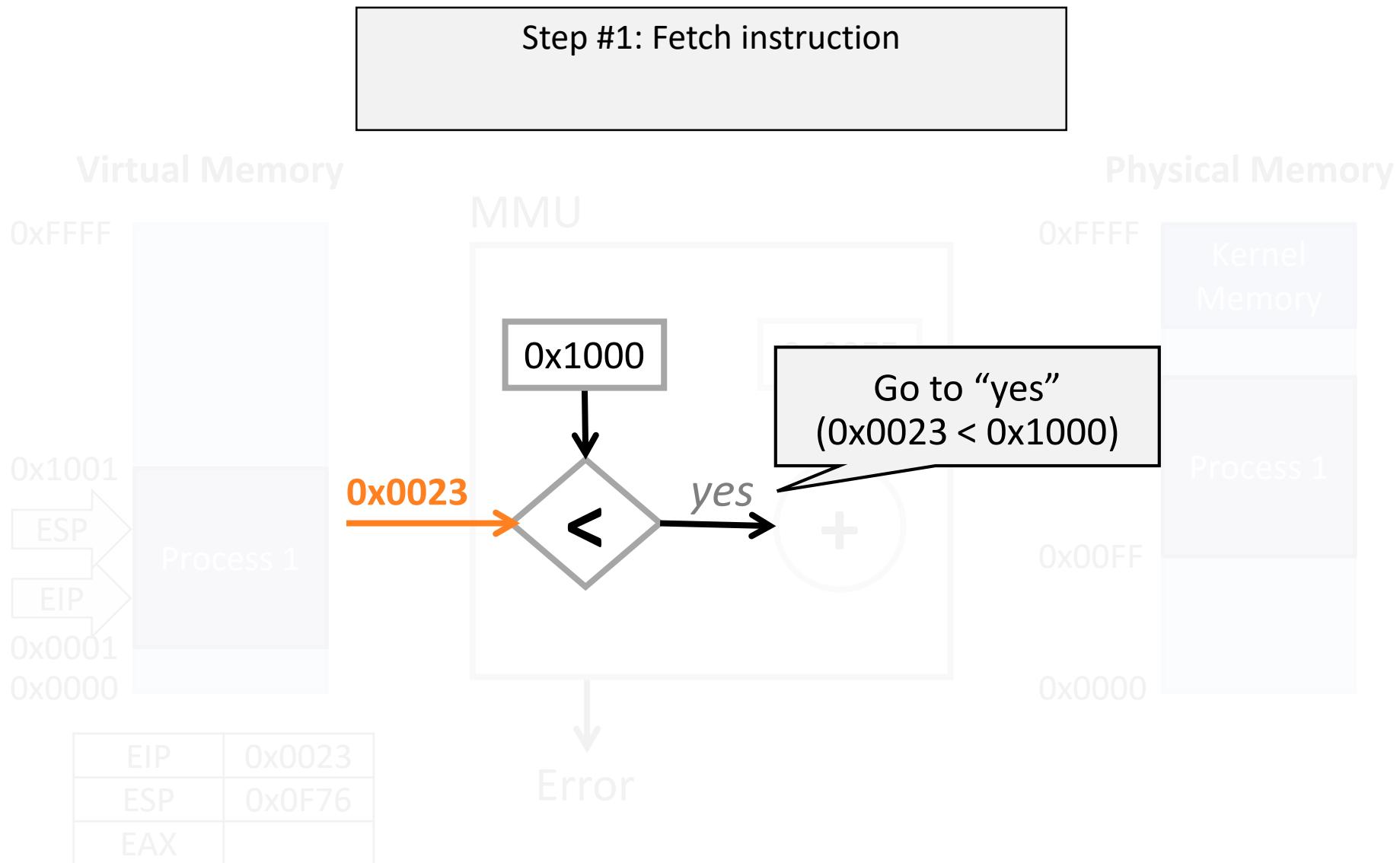
Example



Example

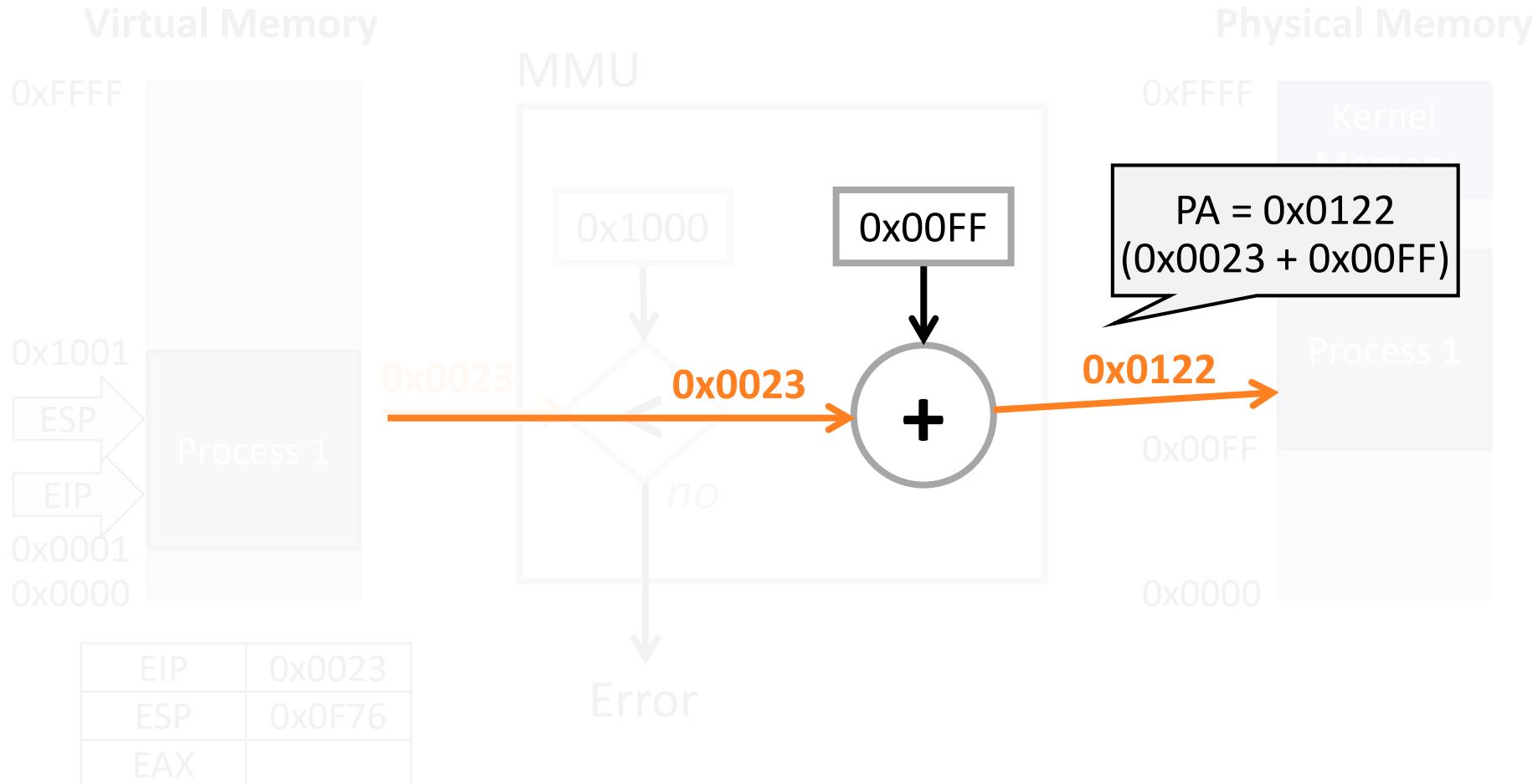


Example



Example

Step #1: Fetch instruction

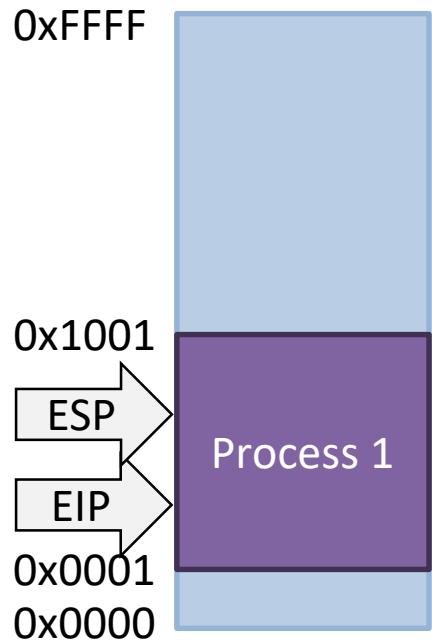


Example

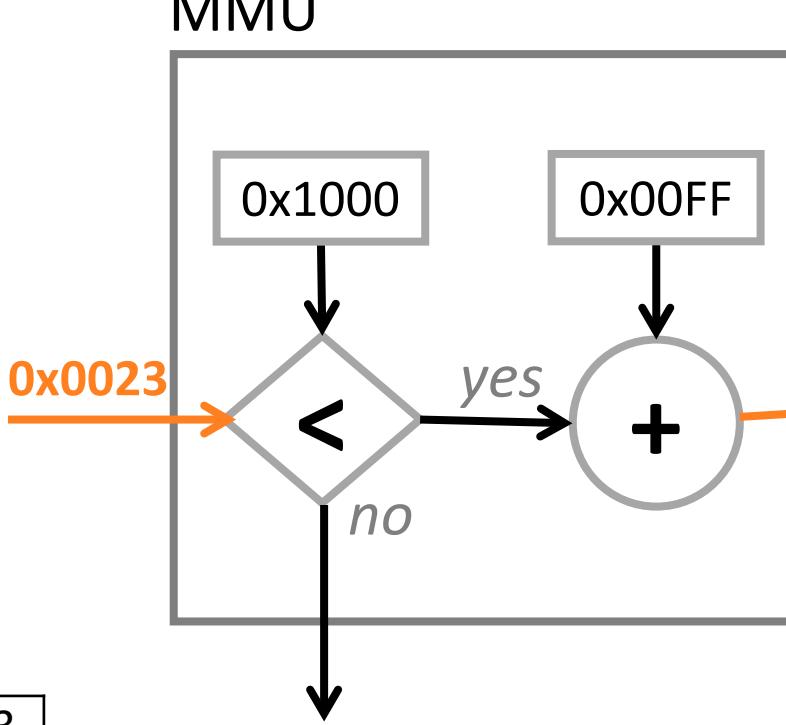
Step #1: Fetch instruction

`mov eax, [esp]`

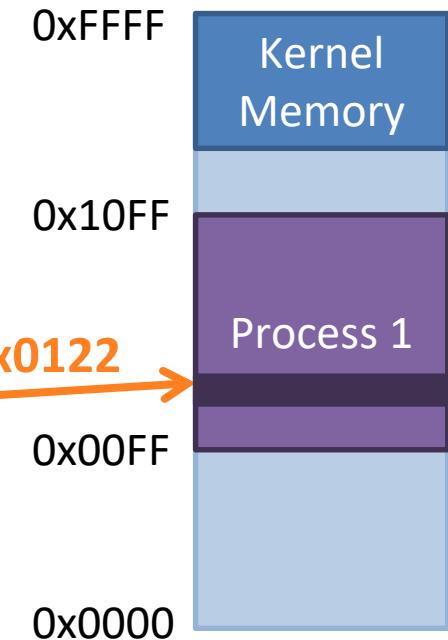
Virtual Memory



MMU



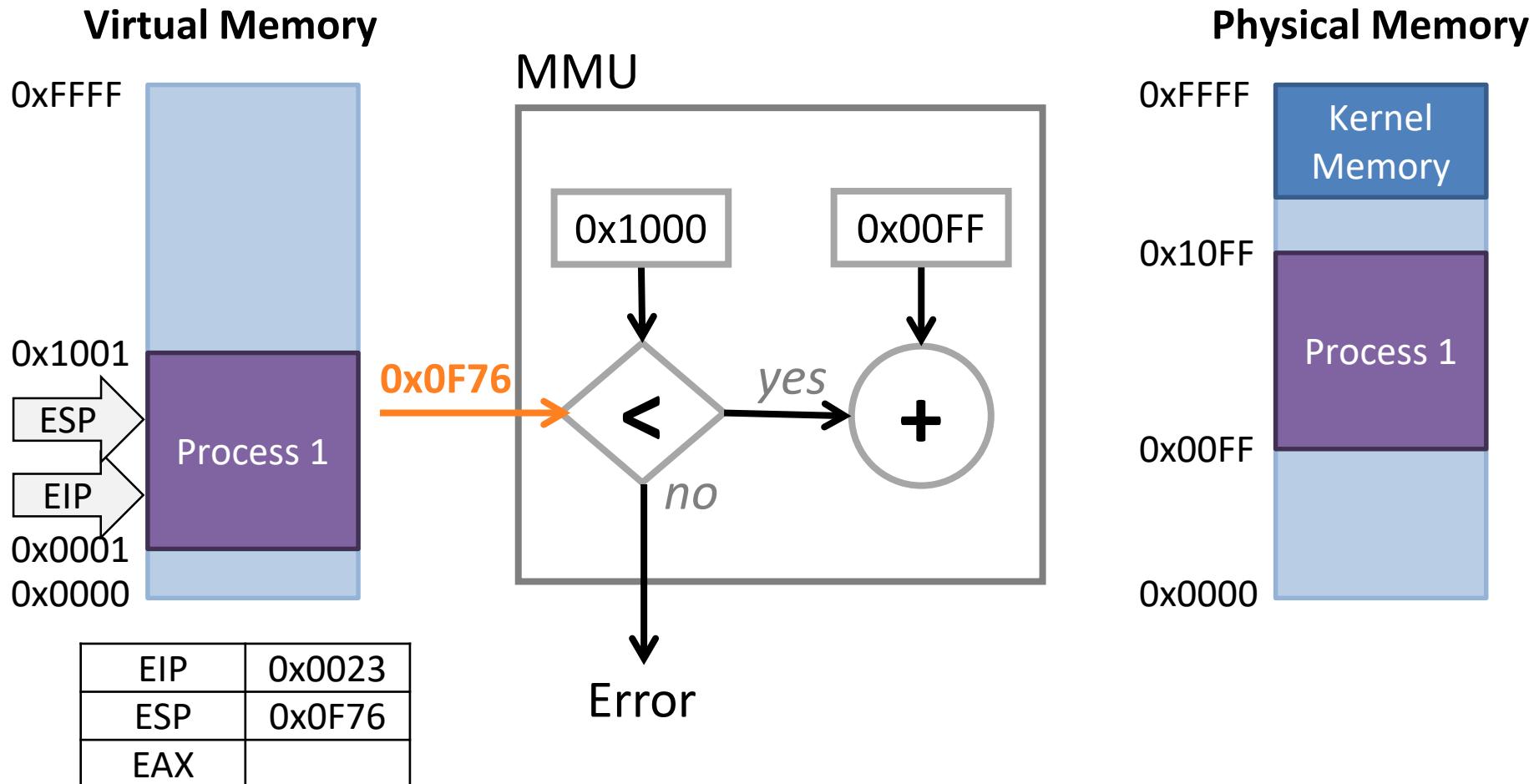
Physical Memory



EIP	0x0023
ESP	0x0F76
EAX	

Example

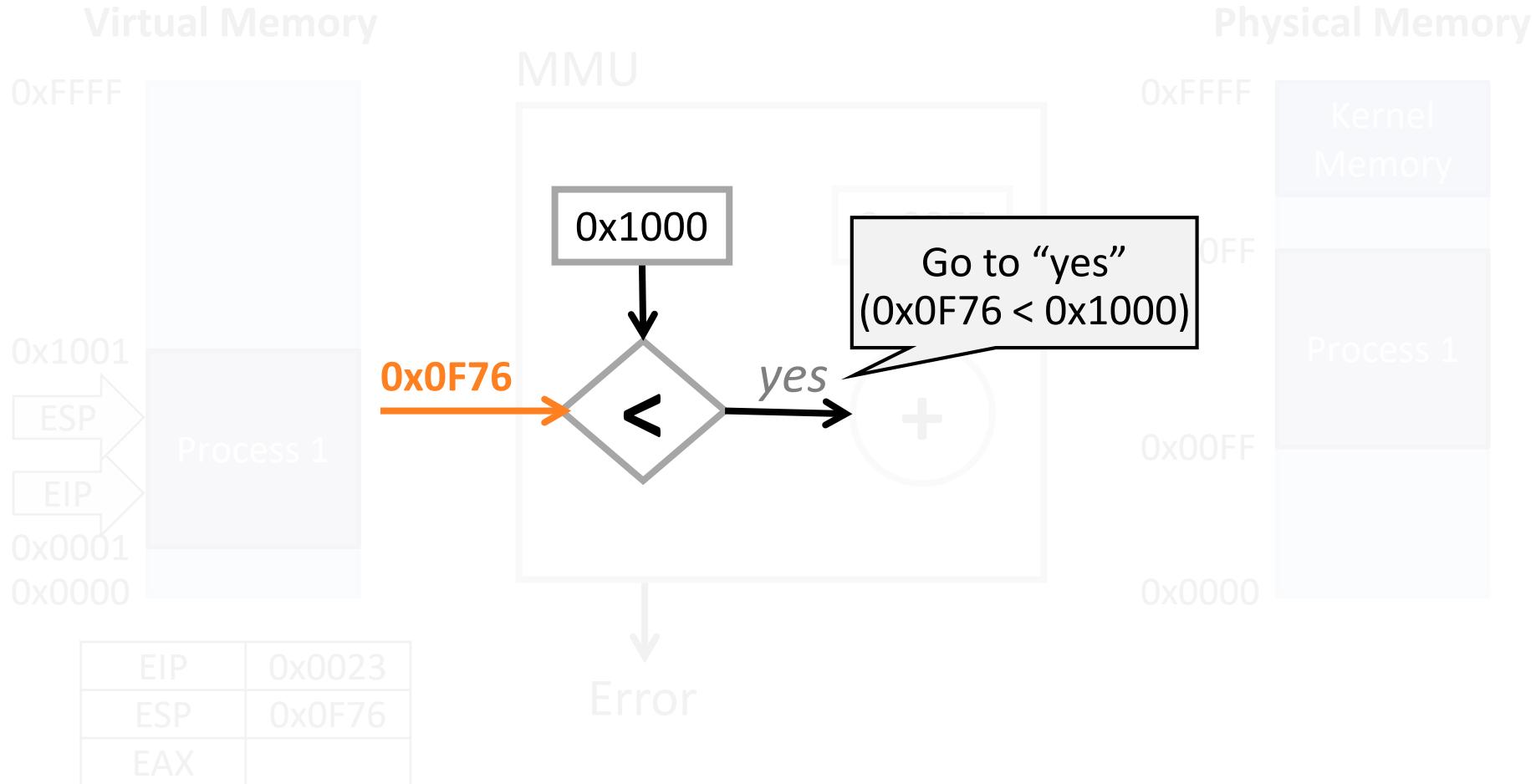
Step #2: Move ESP value to EAX
`mov eax, [esp]`



Example

Step #2: Move ESP value to EAX

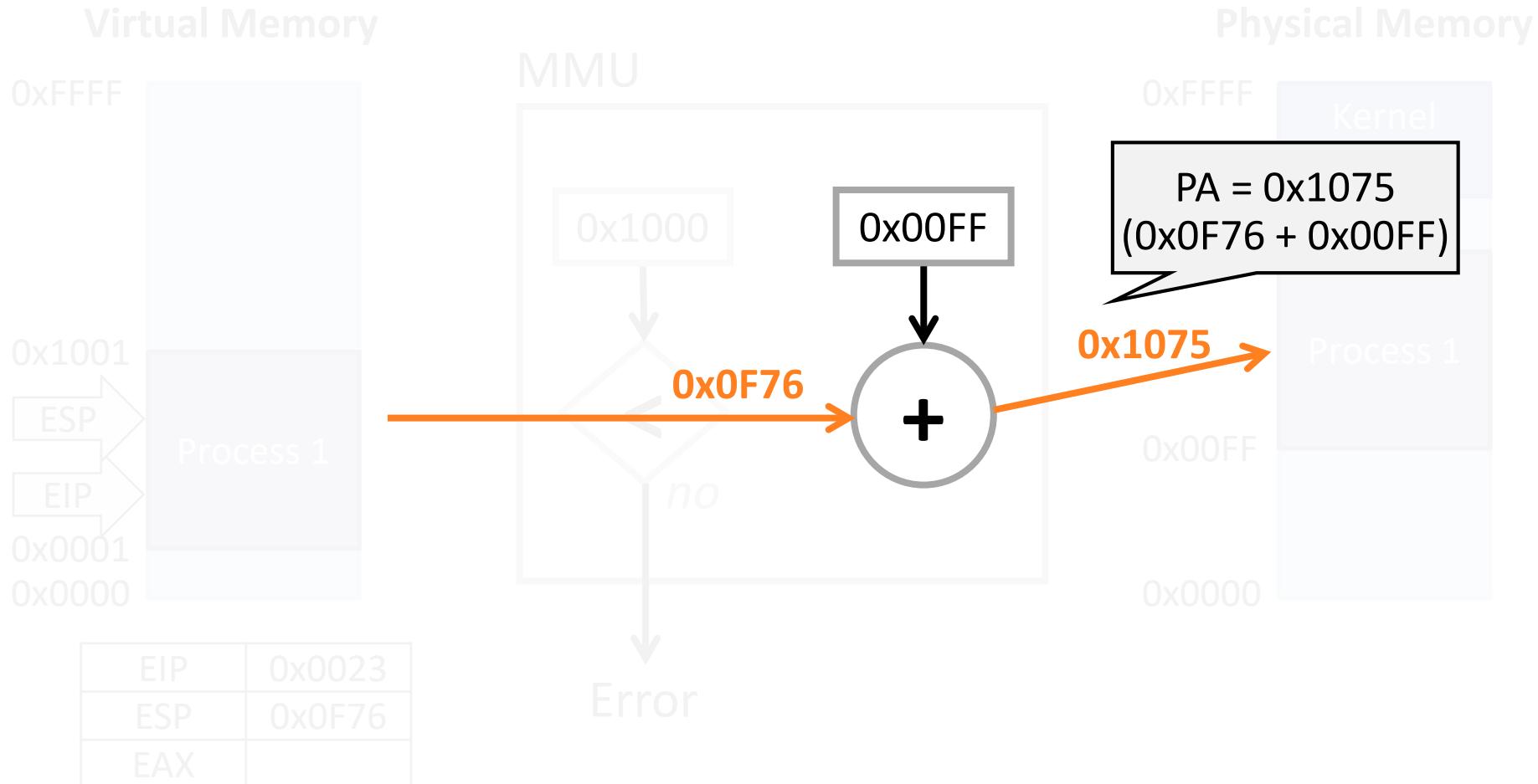
`mov eax, [esp]`



Example

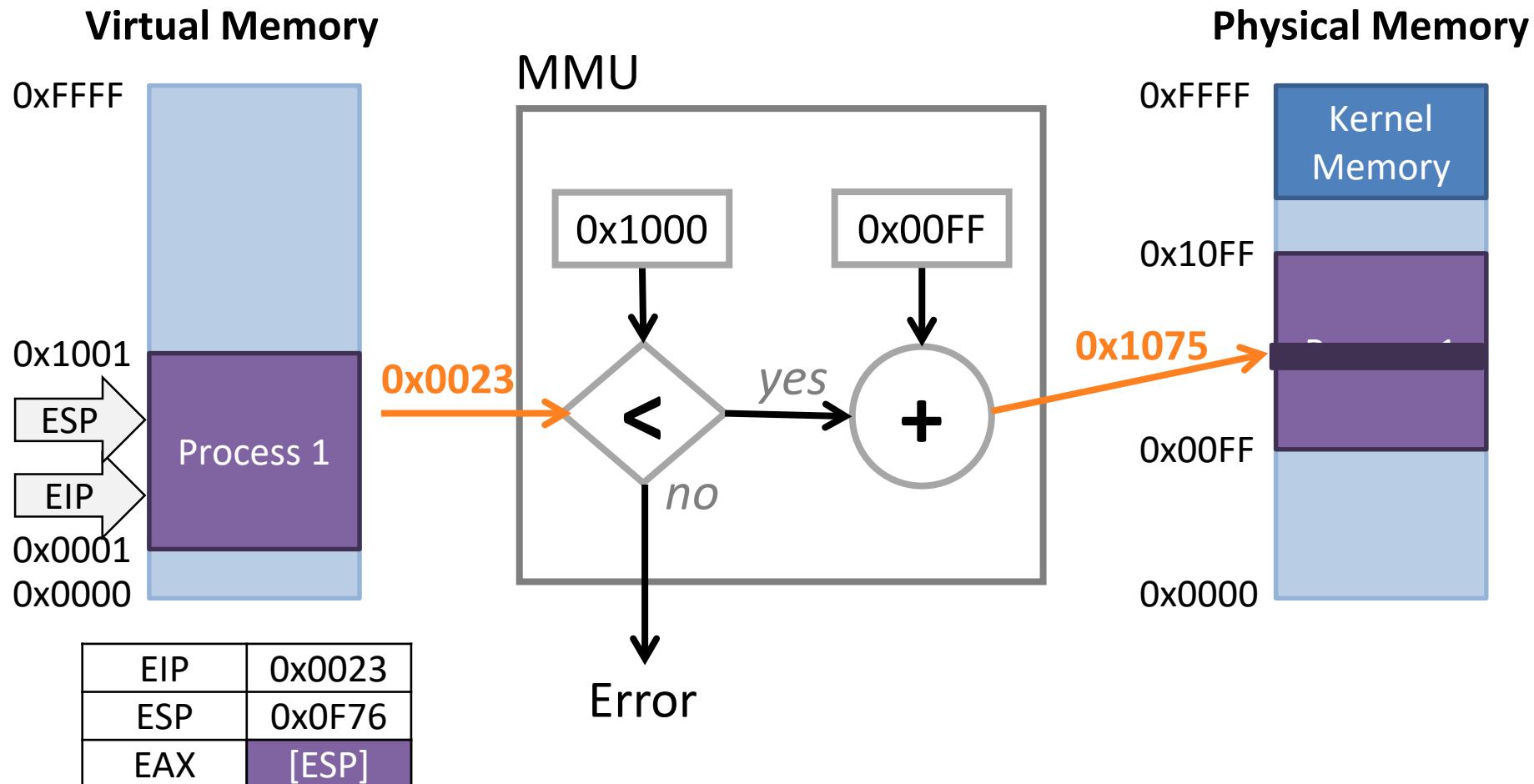
Step #2: Move ESP value to EAX

```
mov eax, [esp]
```



Example

Step #2: Move ESP value to EAX
`mov eax, [esp]`



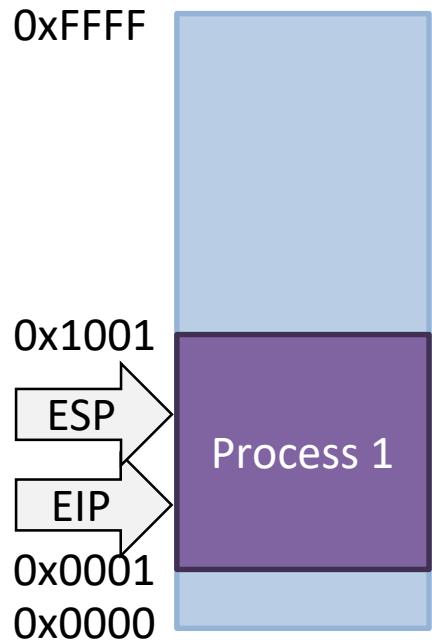
Example

What if the destination address was 0x4234?

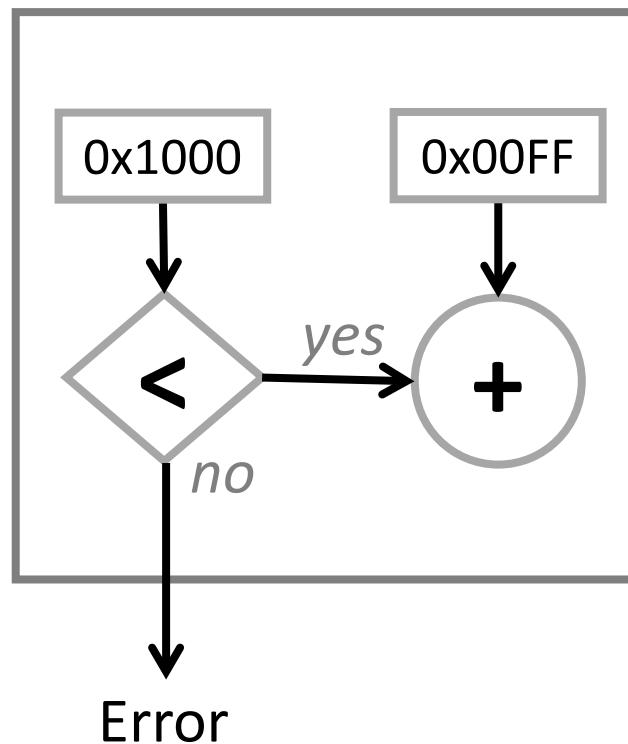
Step #2: Move ESP value to EAX

```
mov eax, [0x4234]
```

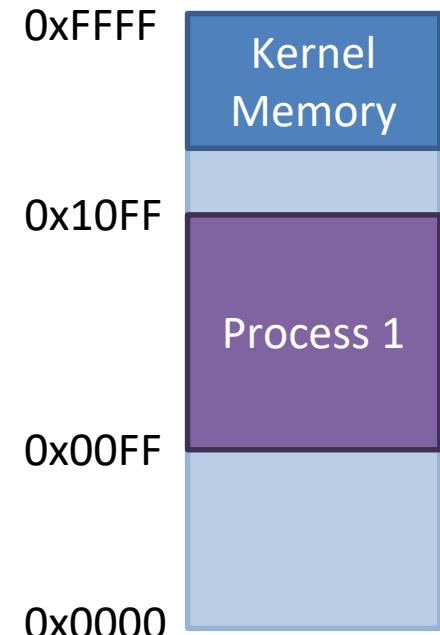
Virtual Memory



MMU



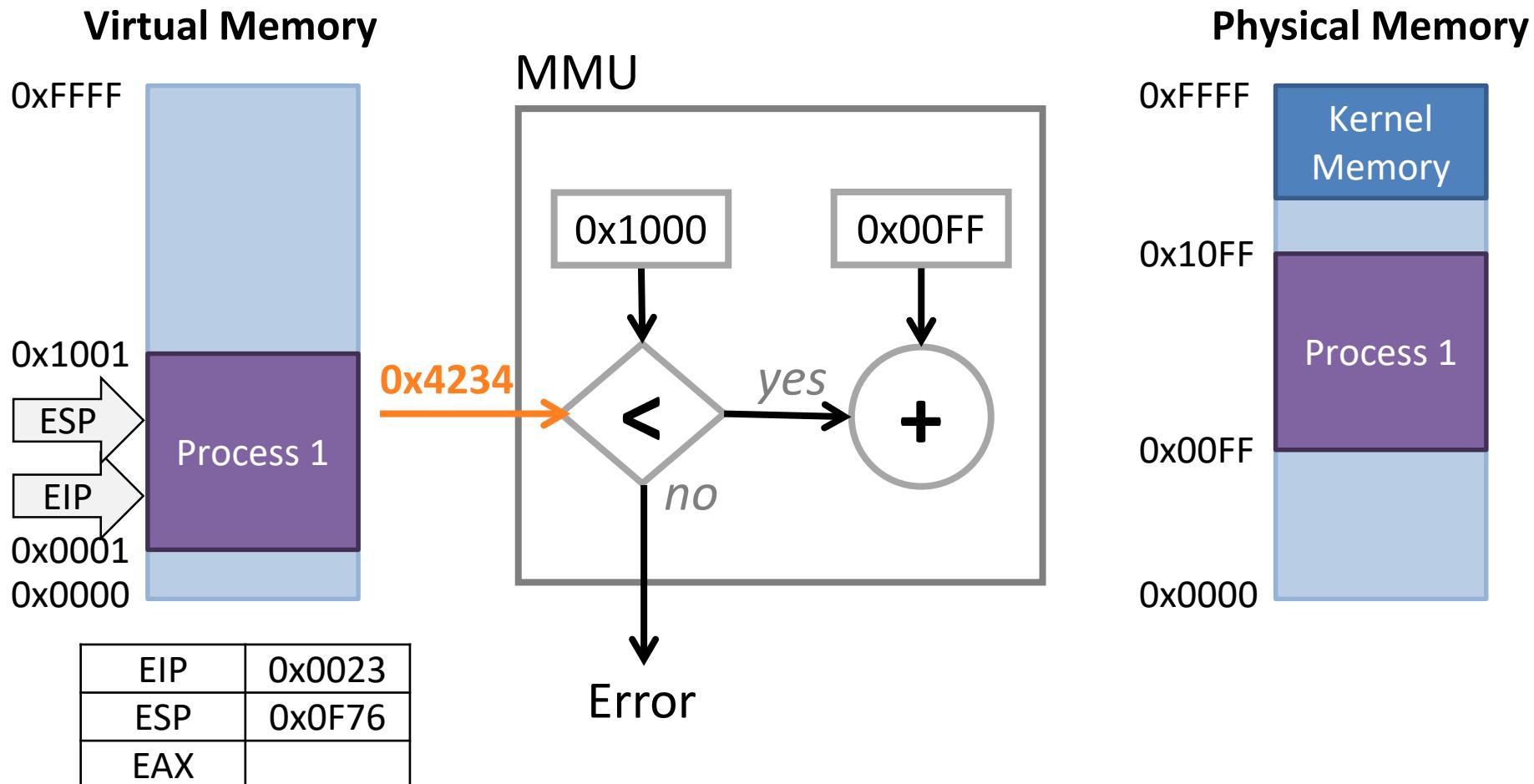
Physical Memory



EIP	0x0023
ESP	0x0F76
EAX	

Example

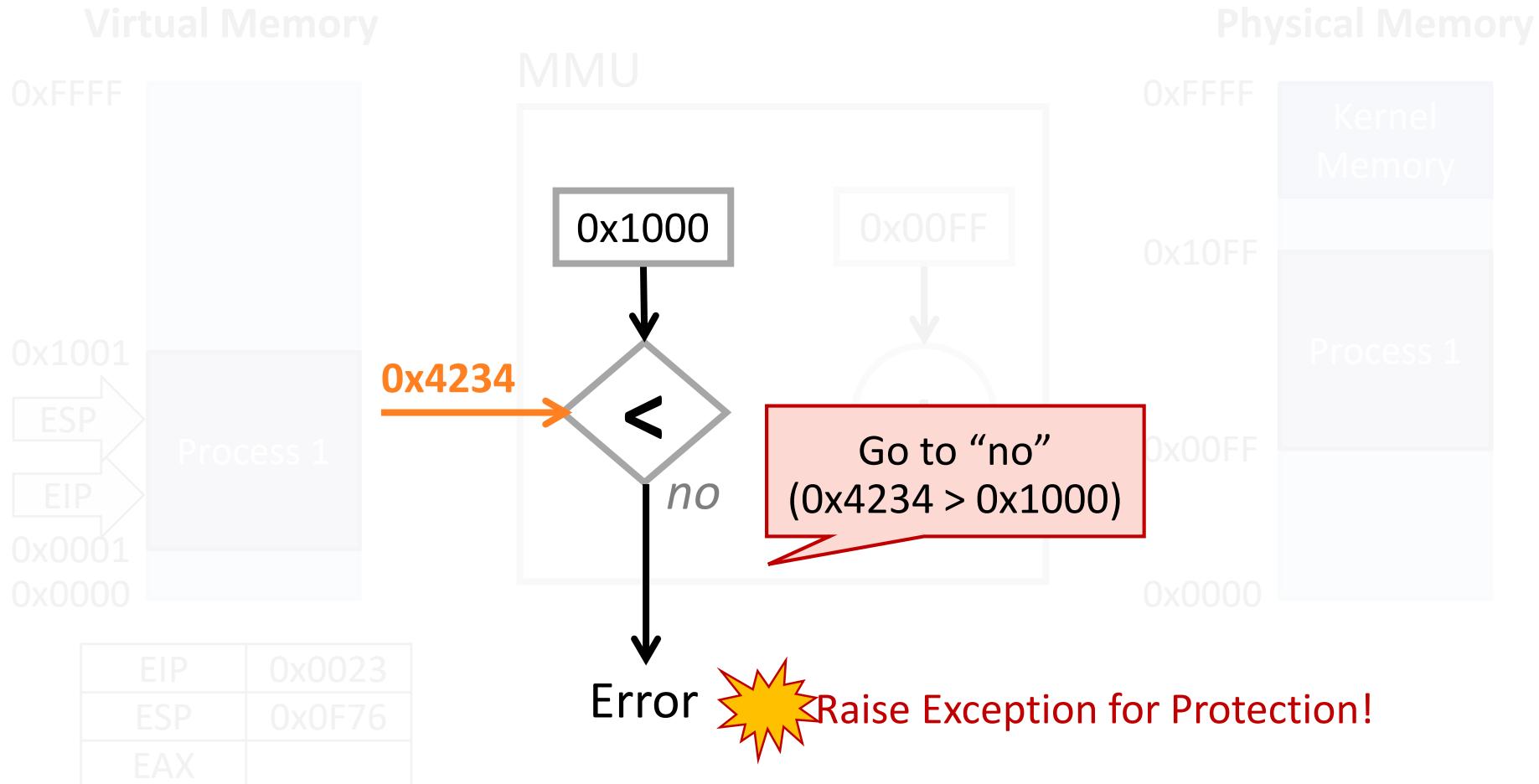
Step #2: Move ESP value to EAX
`mov eax, [0x4234]`

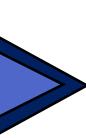


Example

Step #2: Move ESP value to EAX

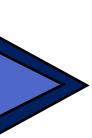
`mov eax, [0x4234]`





Pseudocode of Base and Bound

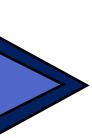
1. PhysAddr = VirtualAddress + BASE
2. if (PhysAddr >= BASE + BOUND)
3. RaiseException(PROTECTION_FAULT)
4. Register = AccessMemory(PhysAddr)



Pseudocode of Base and Bound

OS kernel restores the BASE
and BOUND by referring to
the task_struct

1. PhysAddr = VirtualAddress + BASE
2. if (PhysAddr >= BASE + BOUND)
3. RaiseException(PROTECTION_FAULT)
4. Register = AccessMemory(PhysAddr)



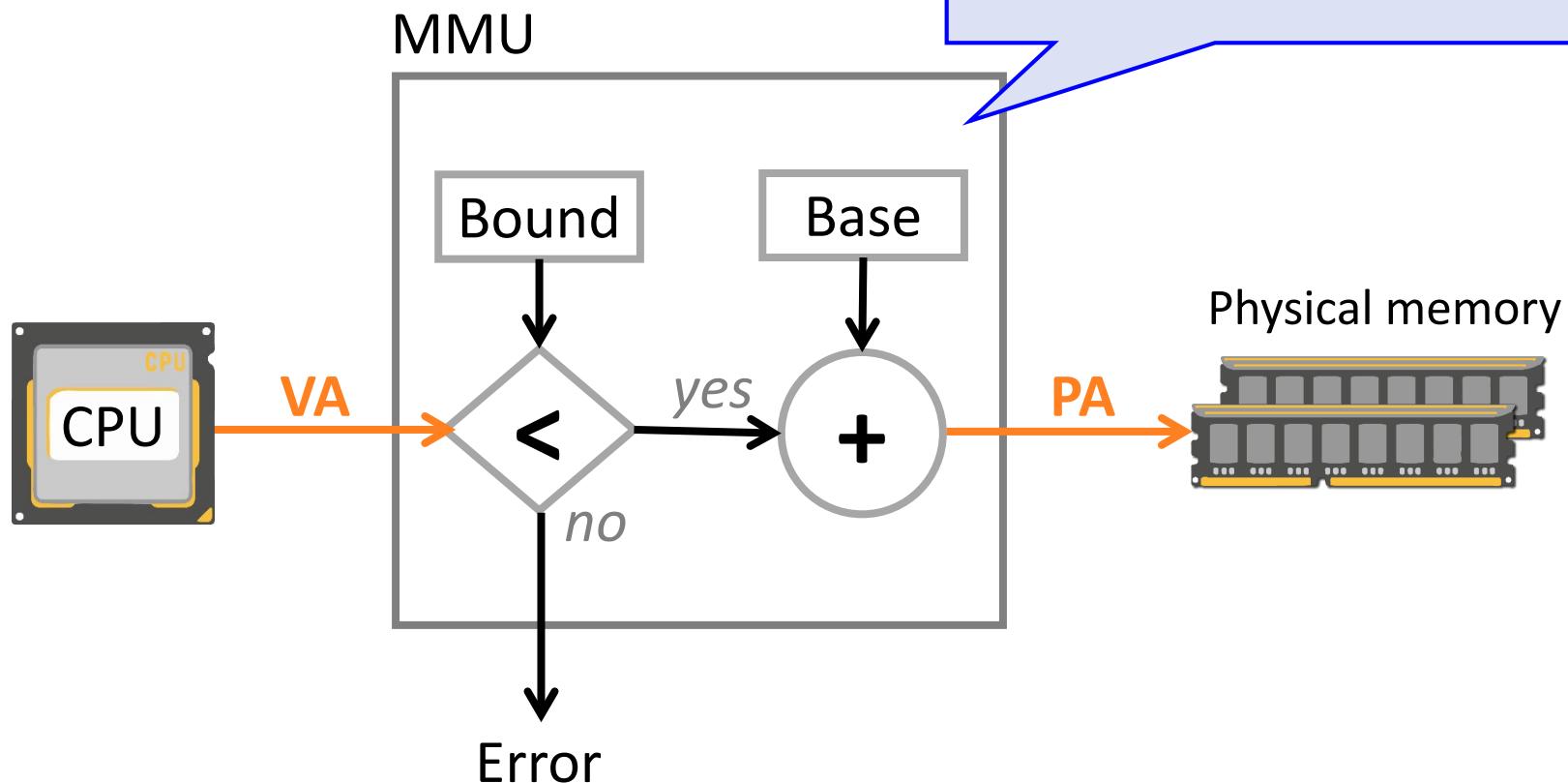
Pseudocode of Base and Bound

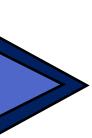
1. PhysAddr = VirtualAddress;
2. if (PhysAddr >= BASE + BOUND)
 RaiseException(PROTECTION_FAULT)
3. Register = AccessMemory(PhysAddr)

If there is a wrong memory reference, CPU raises internal interrupt (exception)

Advantages of Base-and-Bound

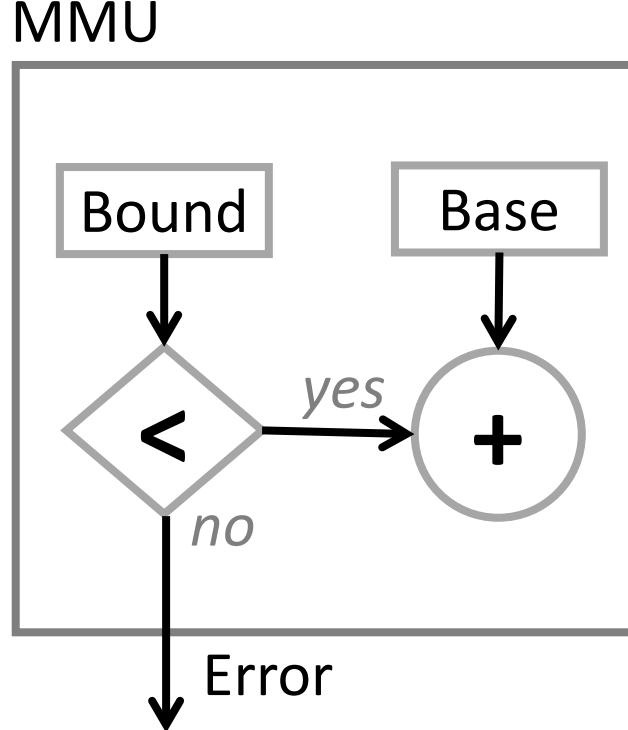
What is the advantage of base-and-bound?



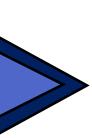


#1: Simple HW Implementation

- With simple HW implementation
 - Can offer protection and isolation
 - Can offer flexible placement of data in memory

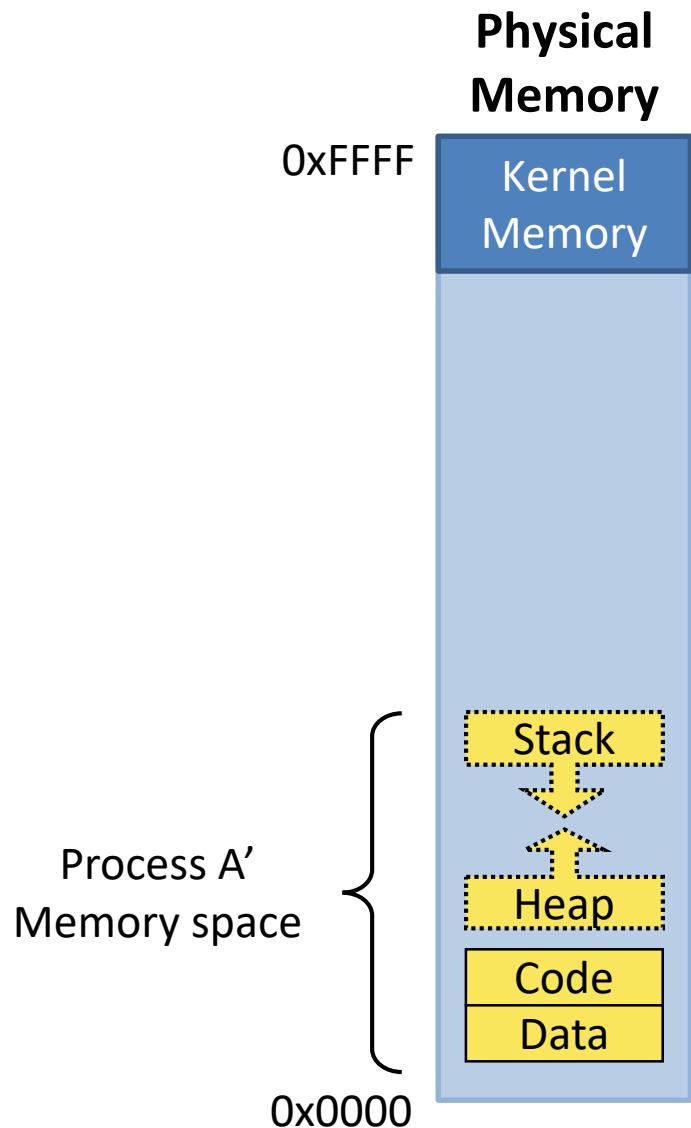


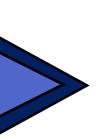
Only two additional registers and simple logic circuits are required



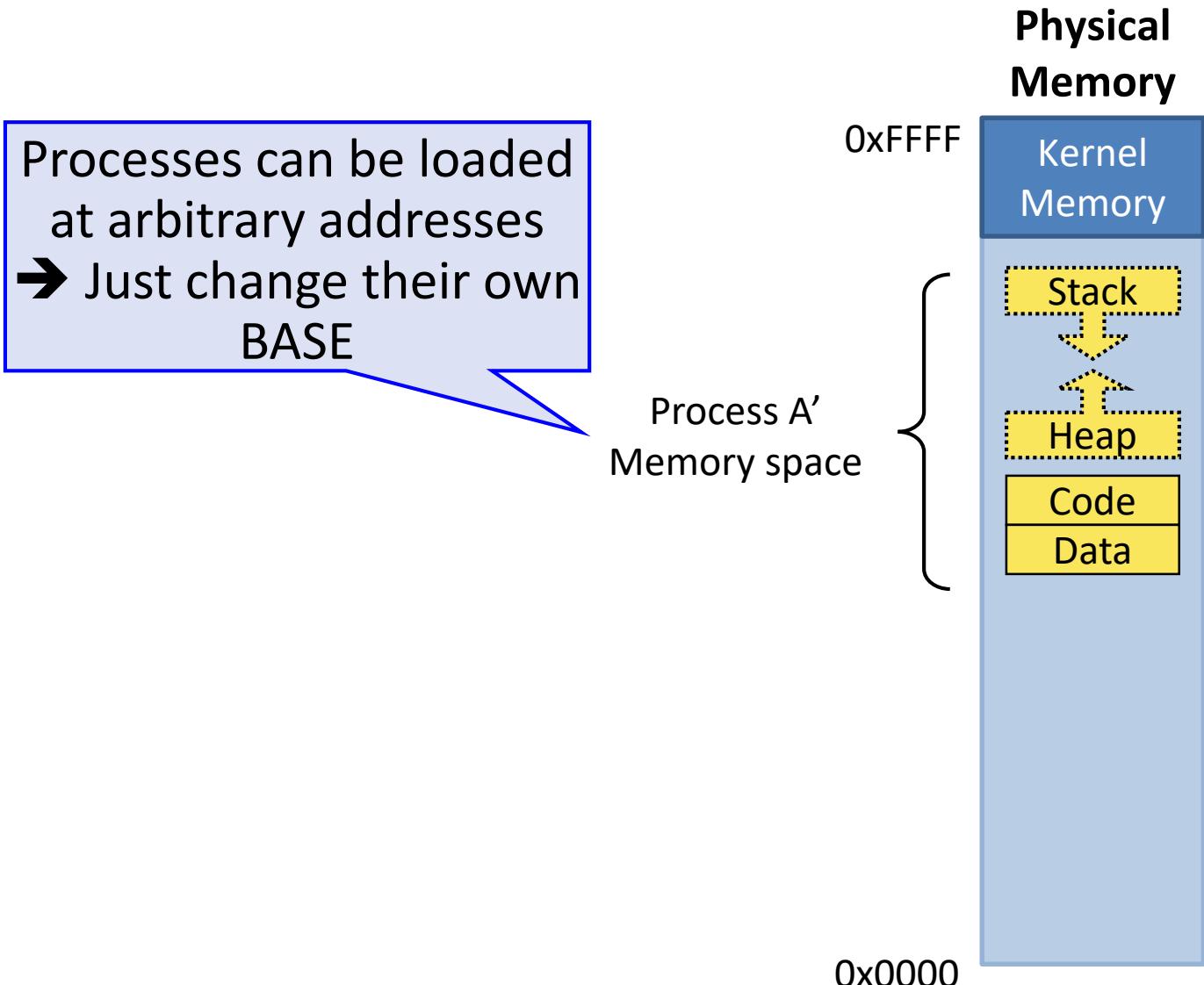
#2: Simple To Manage Virtual Space

Process's memory space is allocated contiguous in a contiguous manner
→ simple to manage

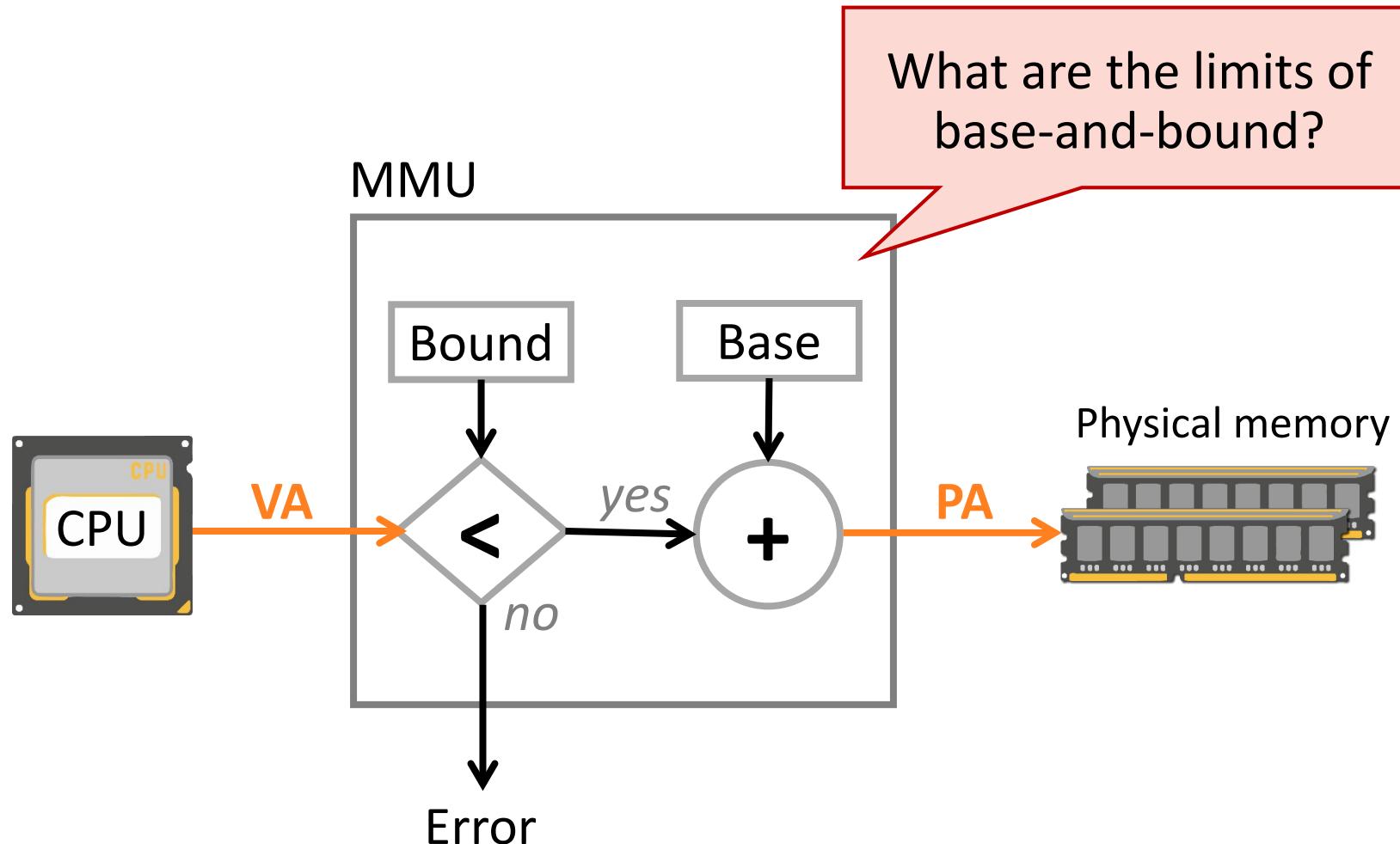




#2: Simple To Manage Virtual Space

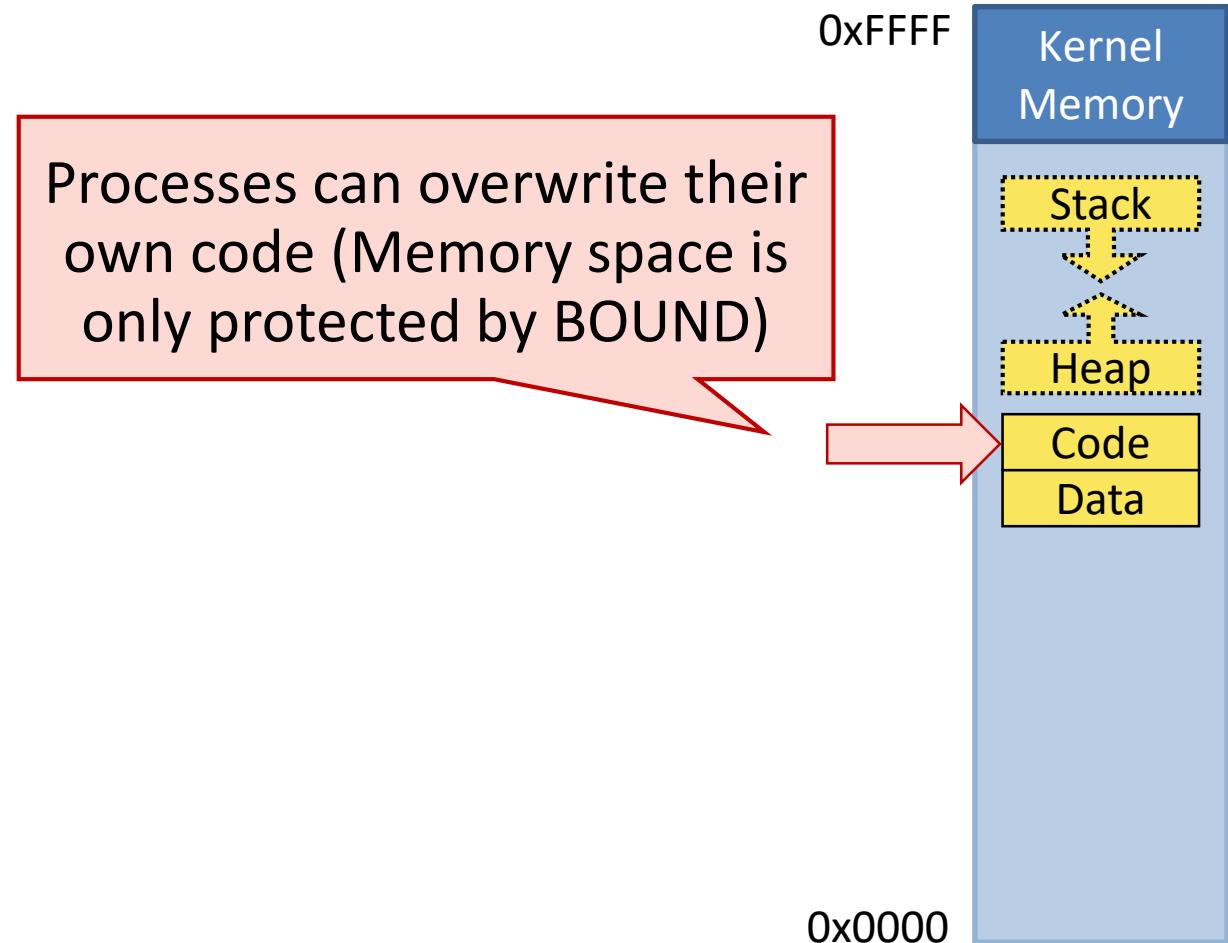


Limitations of Base-and-Bound



#1: No Internal Protection

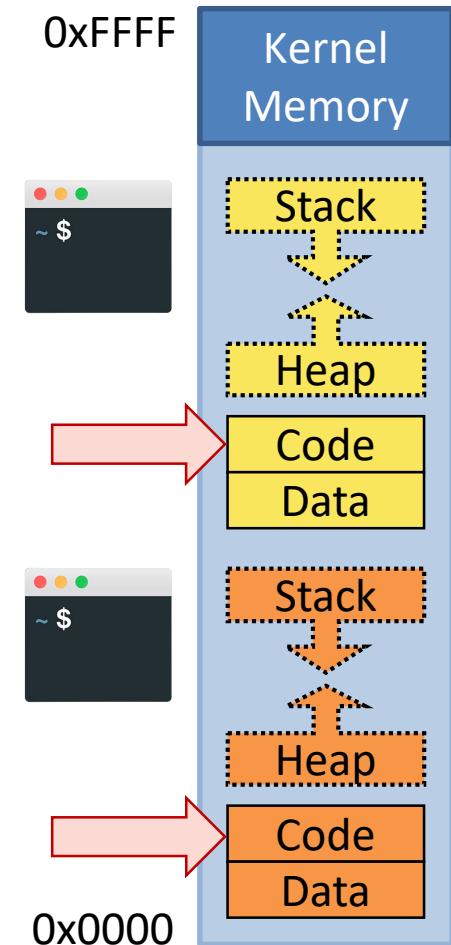
- Processes cannot be protected from themselves



#2: No Sharing of Memory

- Code (read-only) coexist with data (read/write)

Even though processes execute a same program (ex: /bin/bash), their code is duplicated in physical memory 😞

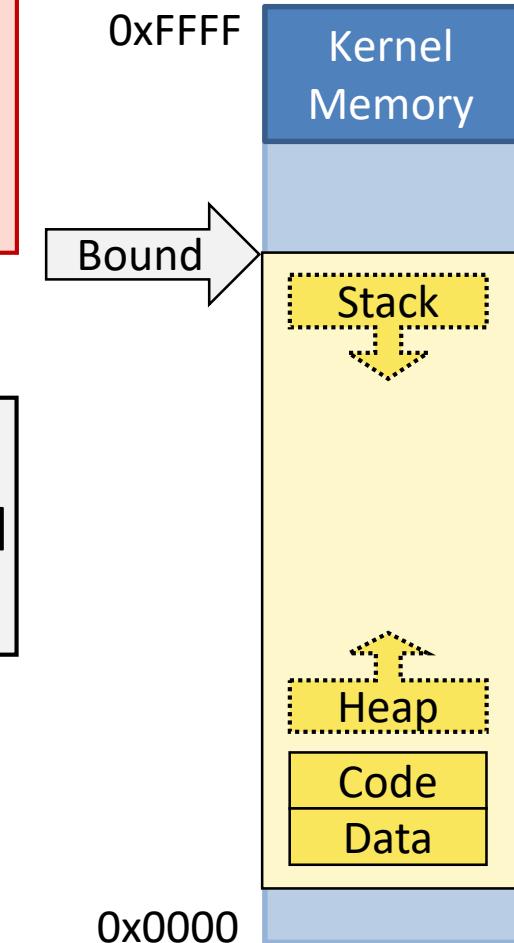


#3: Internal Fragmentation

- Process memory cannot grow dynamically

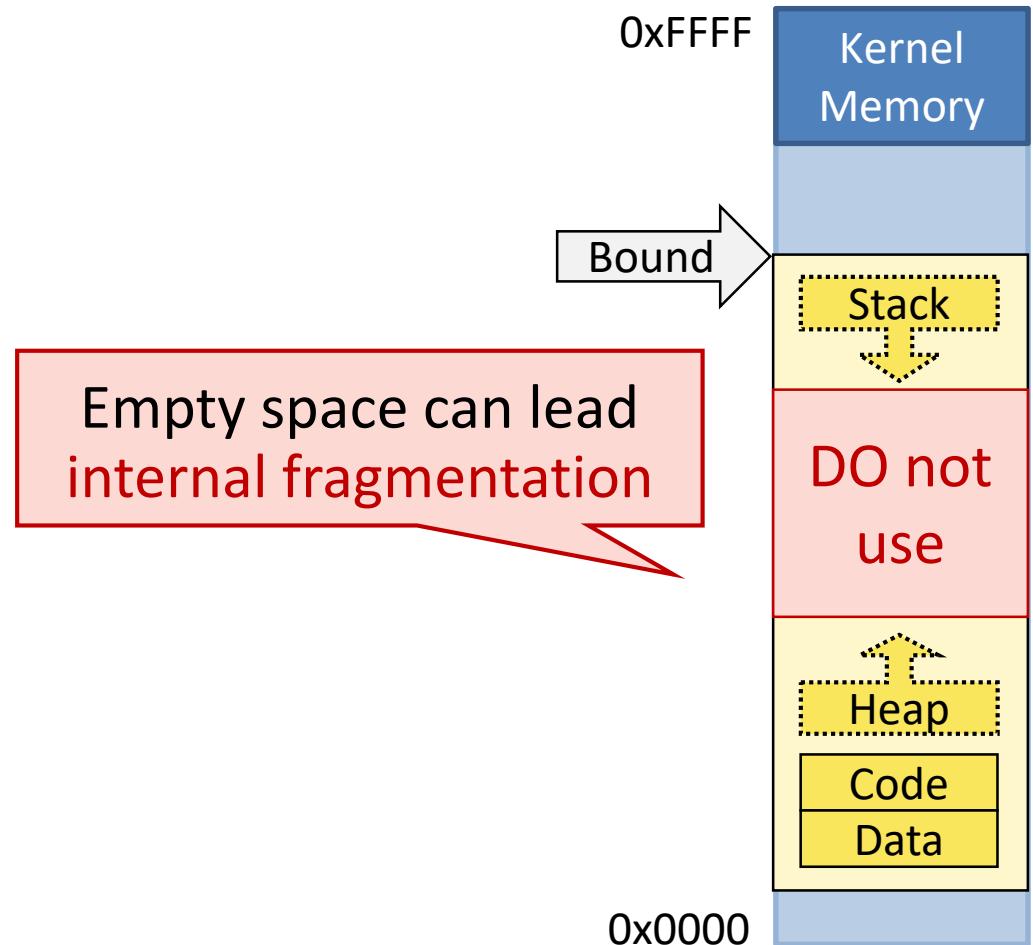
BOUND determines the upper bound of the memory space allocated to its process

Can we speculate how much memory is required for each process?



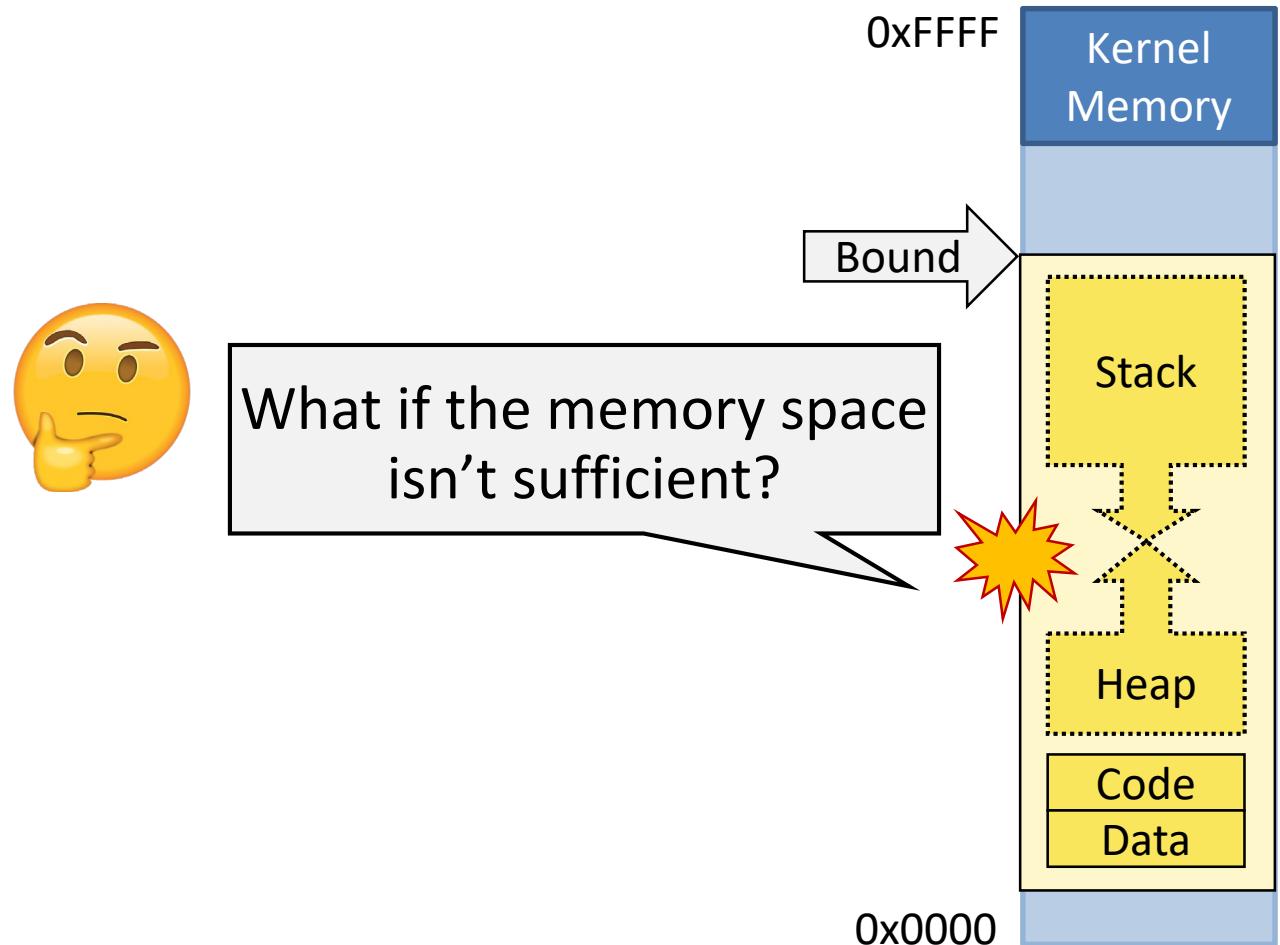
#3: Internal Fragmentation

- Process memory cannot grow dynamically



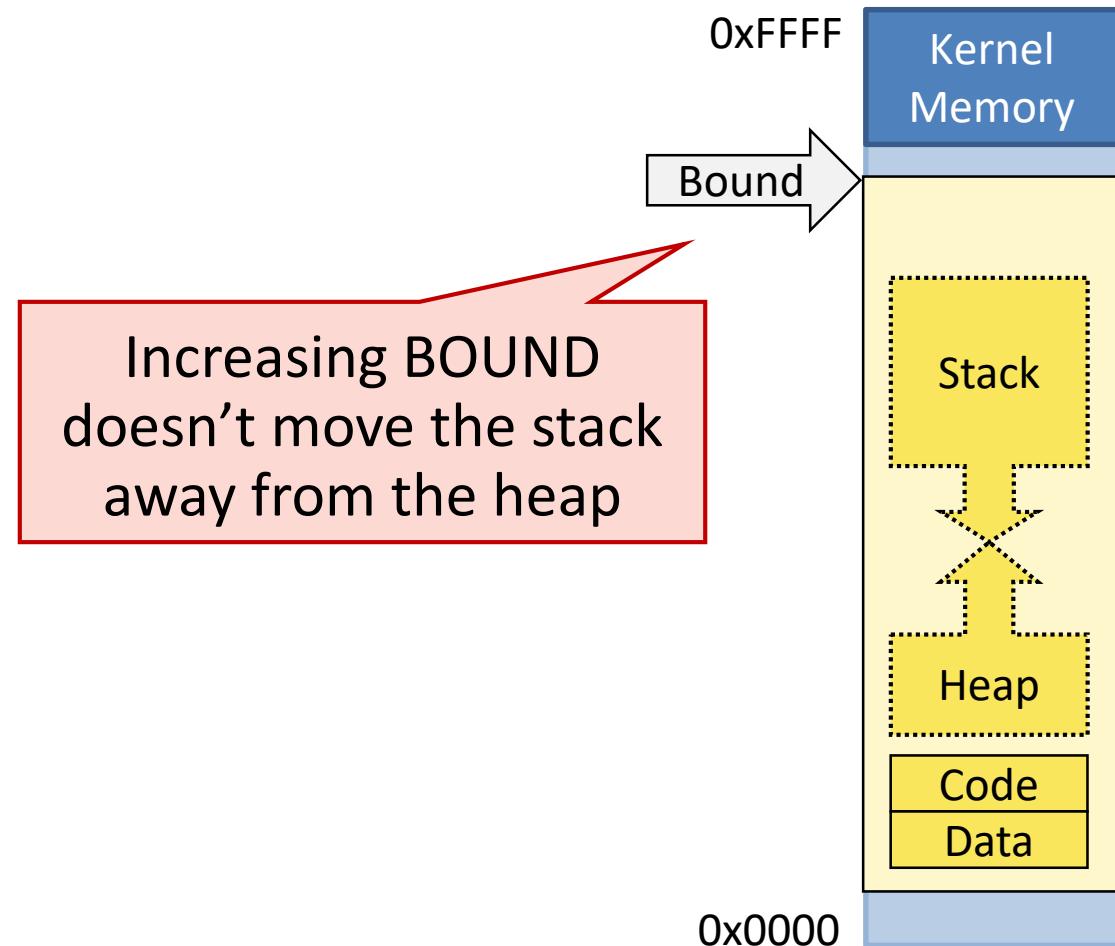
#3: Internal Fragmentation

- Process memory cannot grow dynamically



#3: Internal Fragmentation

- Process memory cannot grow dynamically



What's Next?

- Let's check remained MMU implementation methods, segmentation and paging.

21S EE415

Segmentation and Paging

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



Segmentation and Paging

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

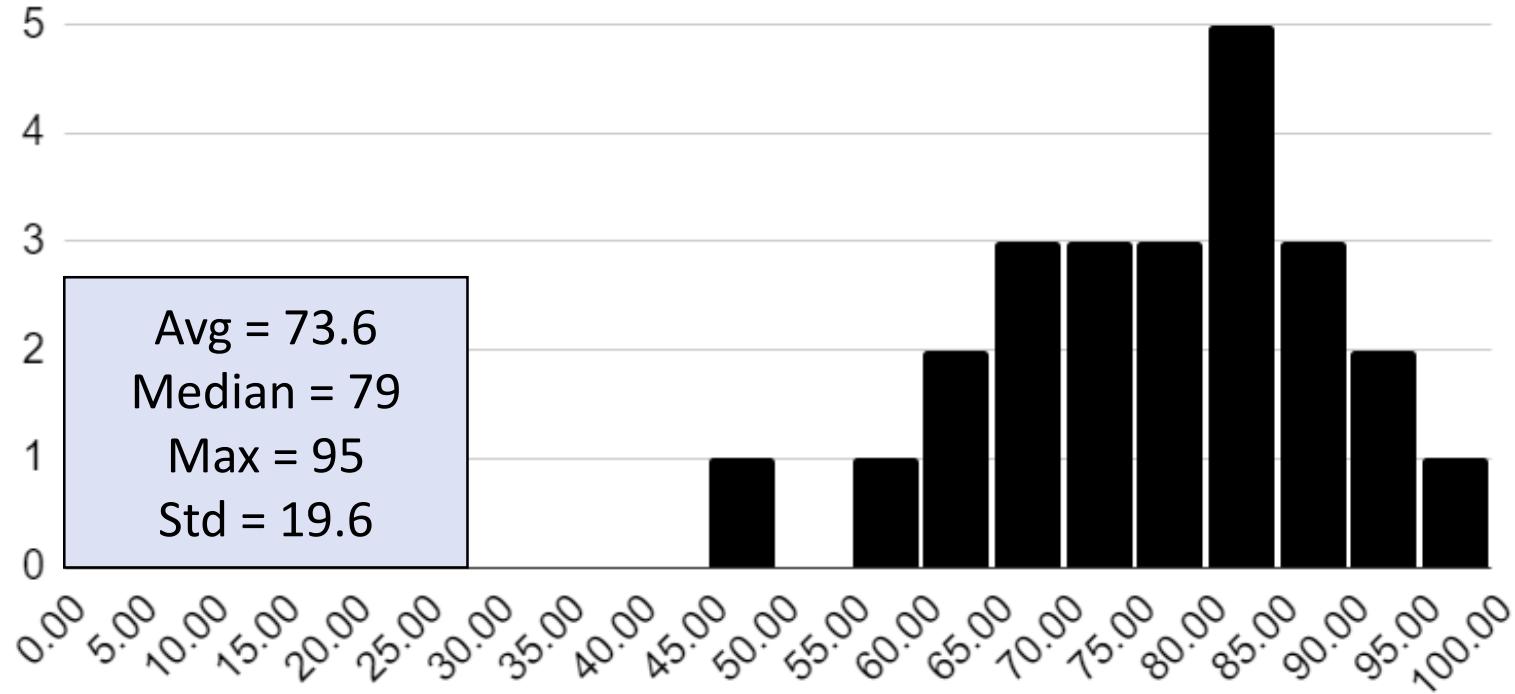
KAIST EE

CAMEL*ab*



NOTICE: Midterm Score Is Ready

- Available at KLMS > Course Management > Grades
- Contact TA: Hanyeoreum Bae (hyr.bae@camelab.org)
- Claim Deadline: 5/2 (Sunday) 23:59:59



Integrated first in the
Burroughs Corporation
B5000 ('61)

Segmentation

Base and
Bound

Segmentation

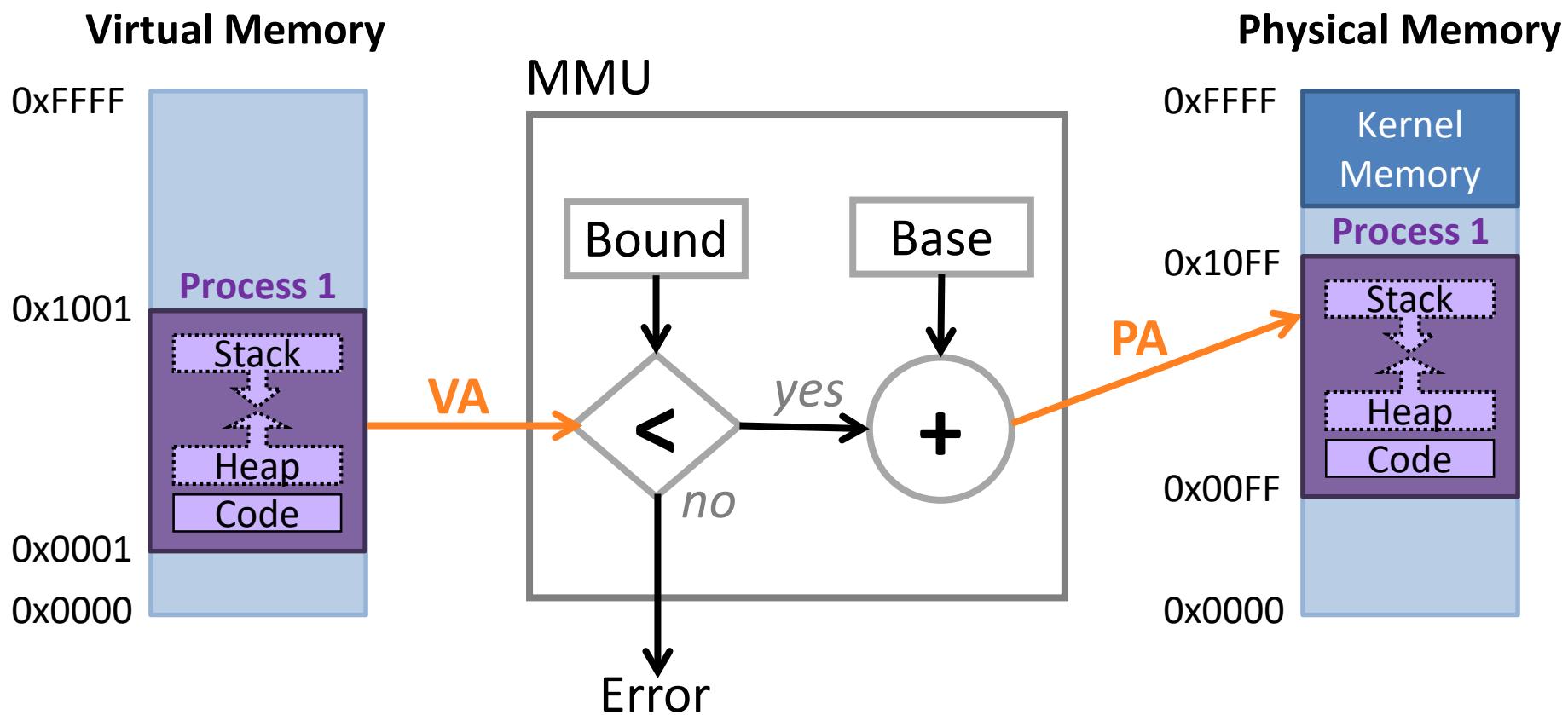


Old, simple, limited
functionality

Modern, complex,
lots of functionality

[Recap] Base and Bound

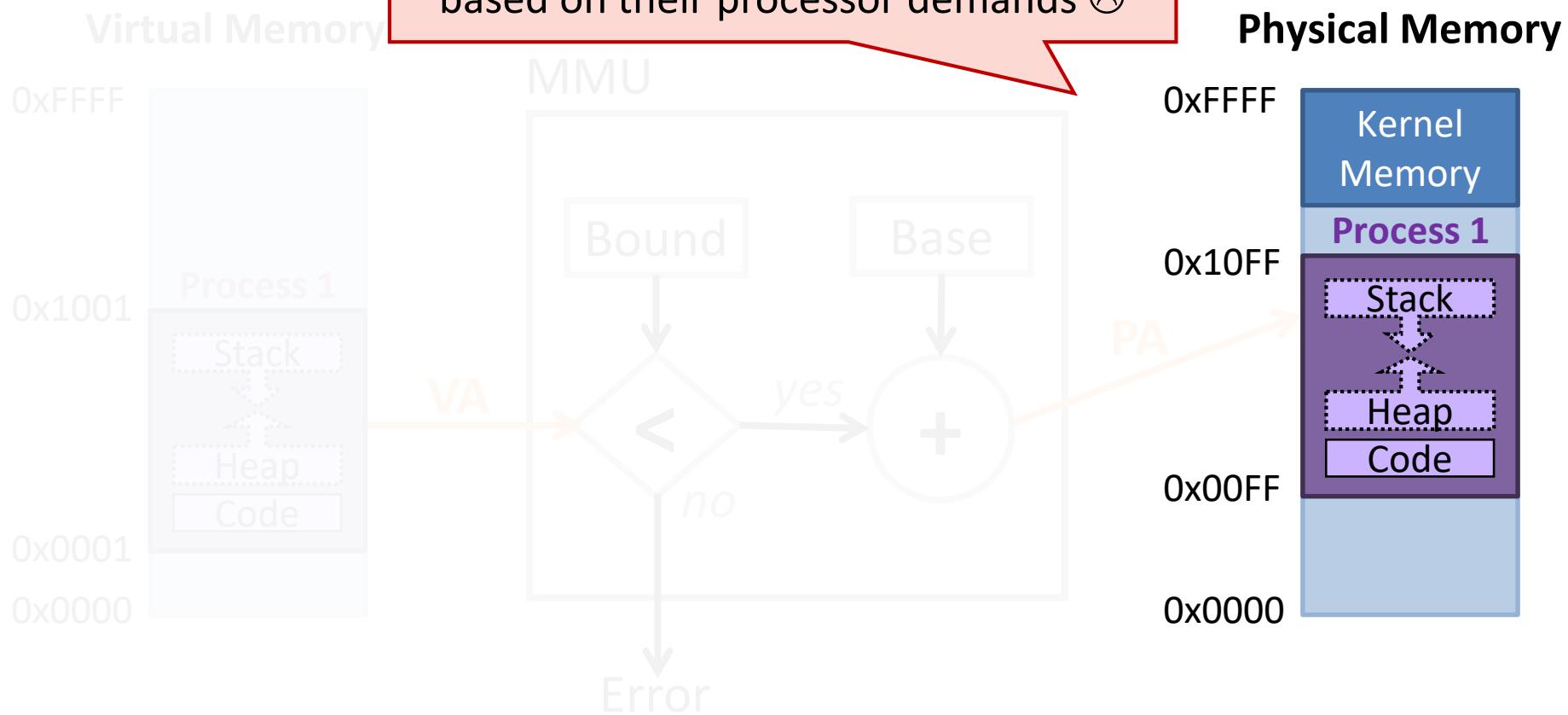
- Representing virtual memory through only two unique registers, BASE and BOUND, means all code, stack, and heap data are all in a single memory region



[Recap] Base and Bound

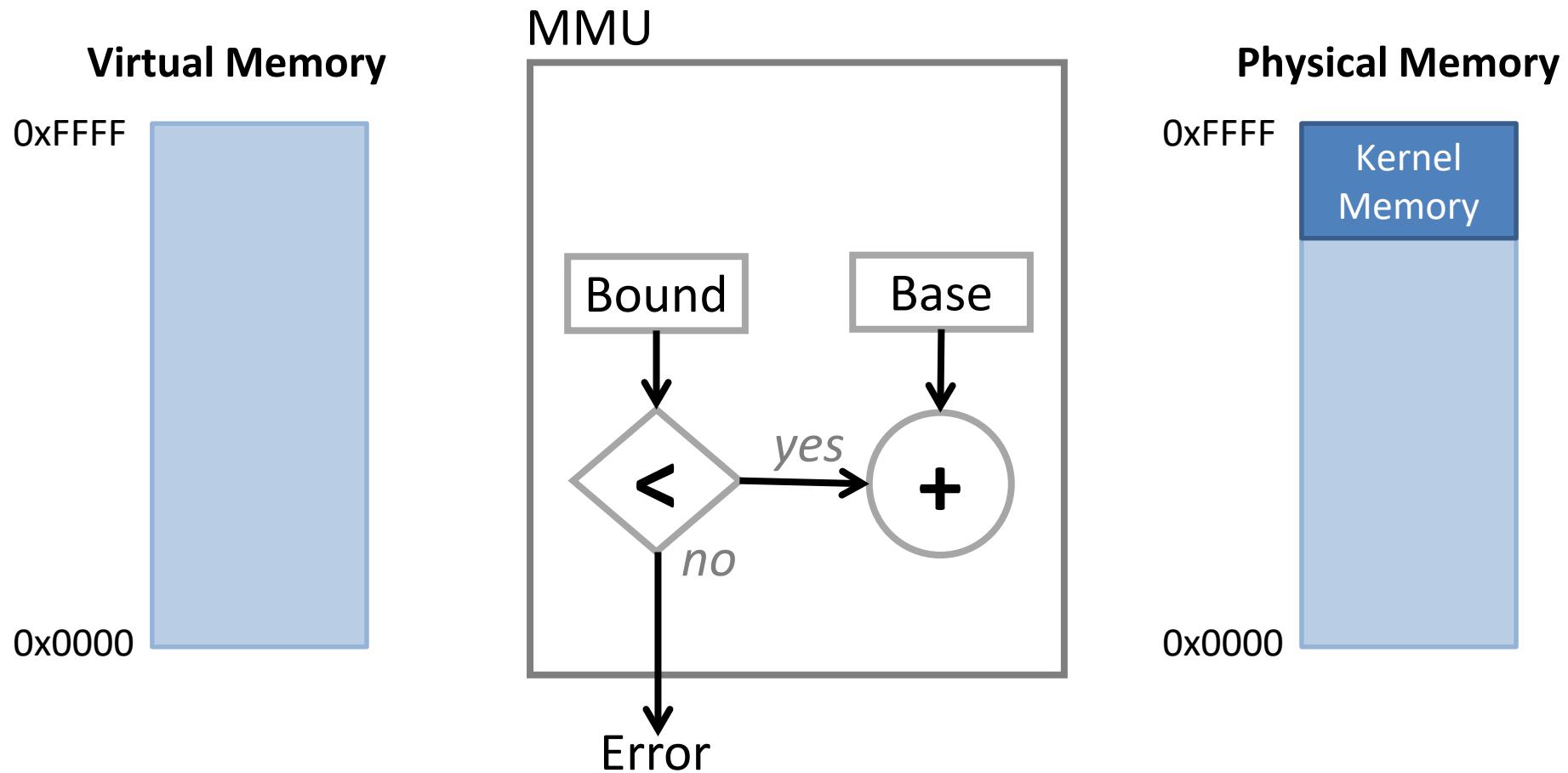
- Representing virtual memory through only two unique registers, BASE and BOUND, means all code, stack, and heap data are

It introduces the internal fragmentation and cannot resize the stack and heap based on their processor demands 😞



Towards Segmented Memory

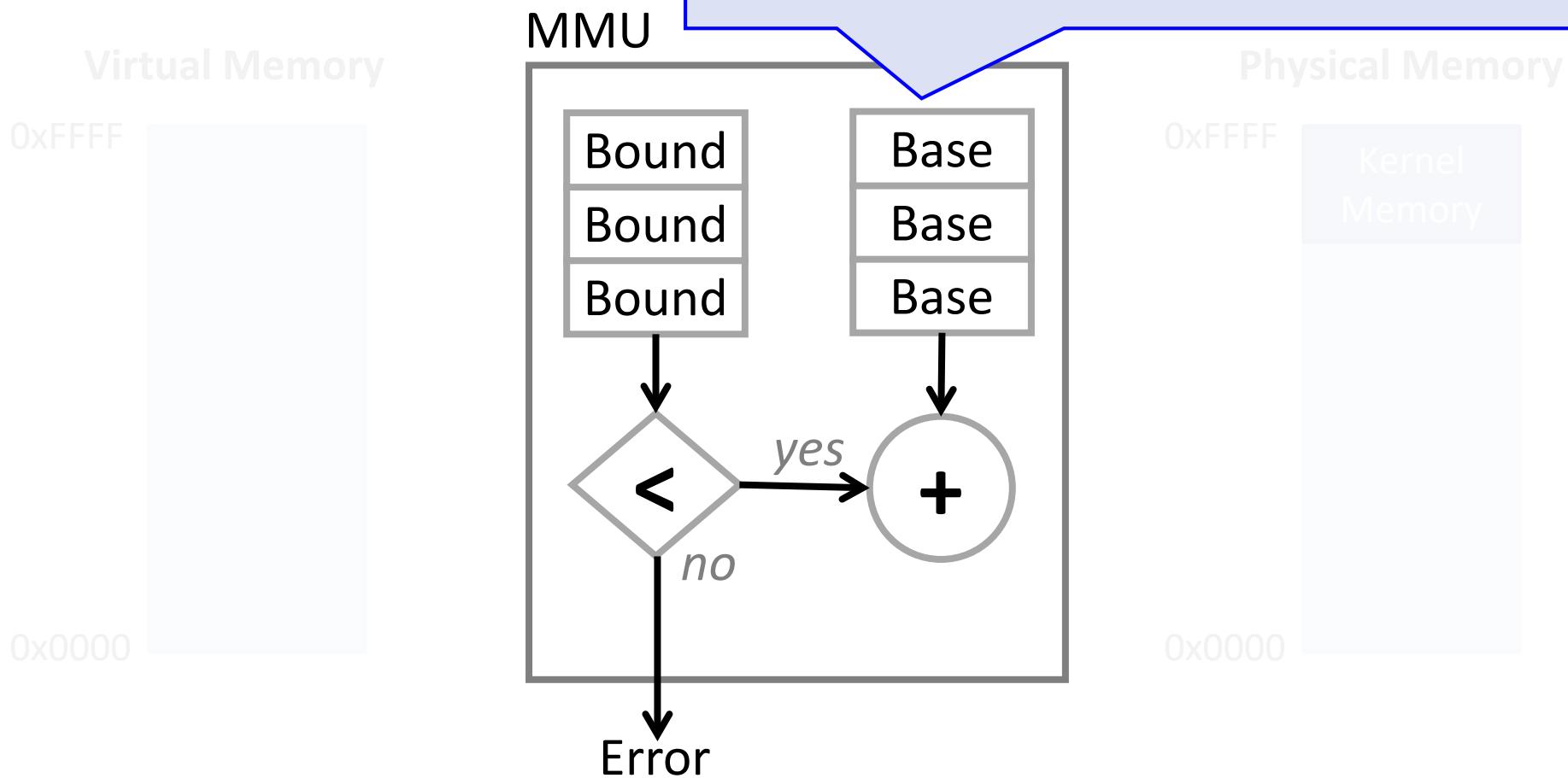
- **Segmentation** is a generalization of the base and bounds approach



Towards Segmented Memory

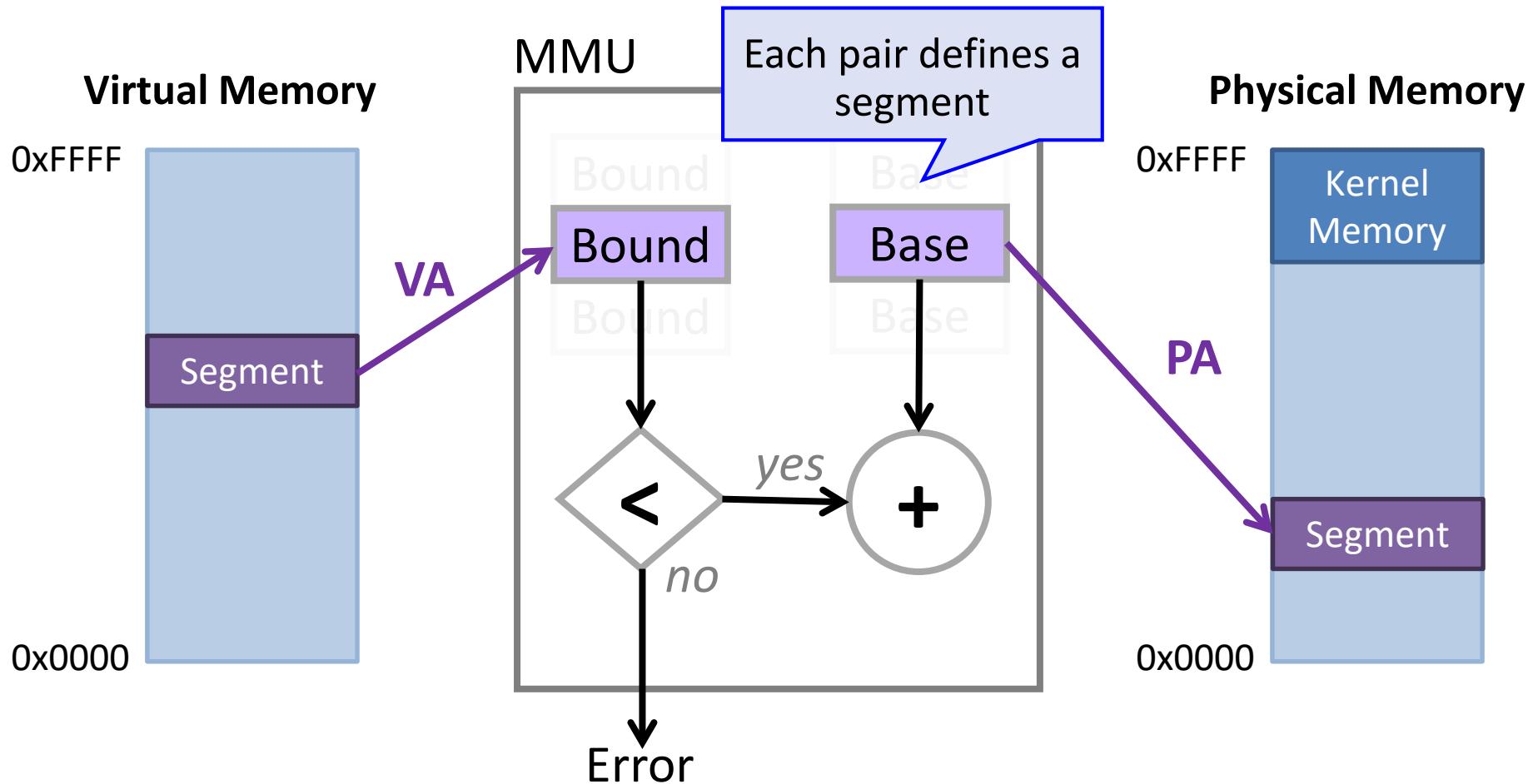
- Segmentation is a general bounds approach

Give each process several **pairs** of base/bounds
(in practice, it can be implemented by using dedicated registers like what we will show or some reserved memory spaces)



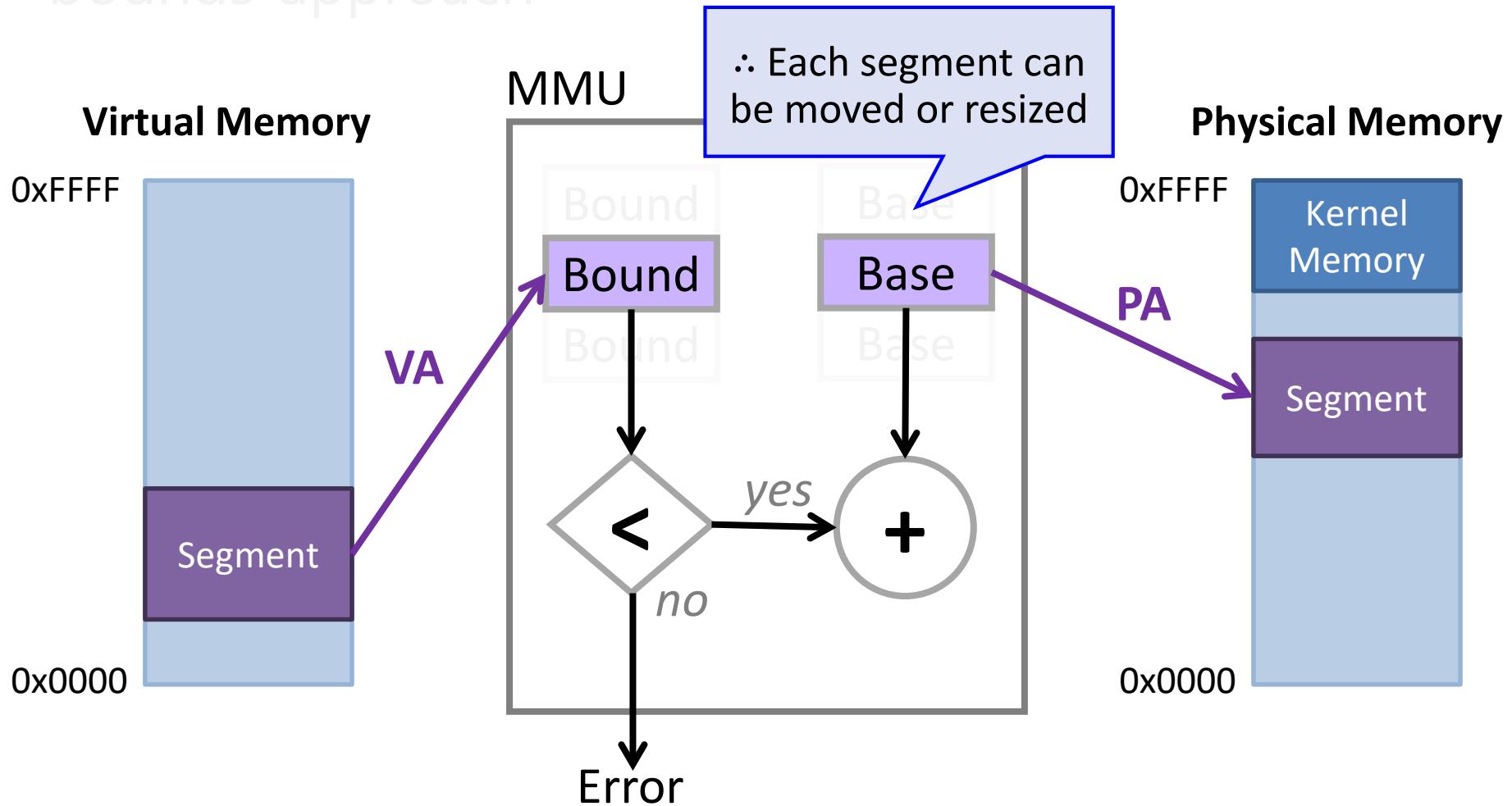
Towards Segmented Memory

- Segmentation is a generalization of the base and bounds approach

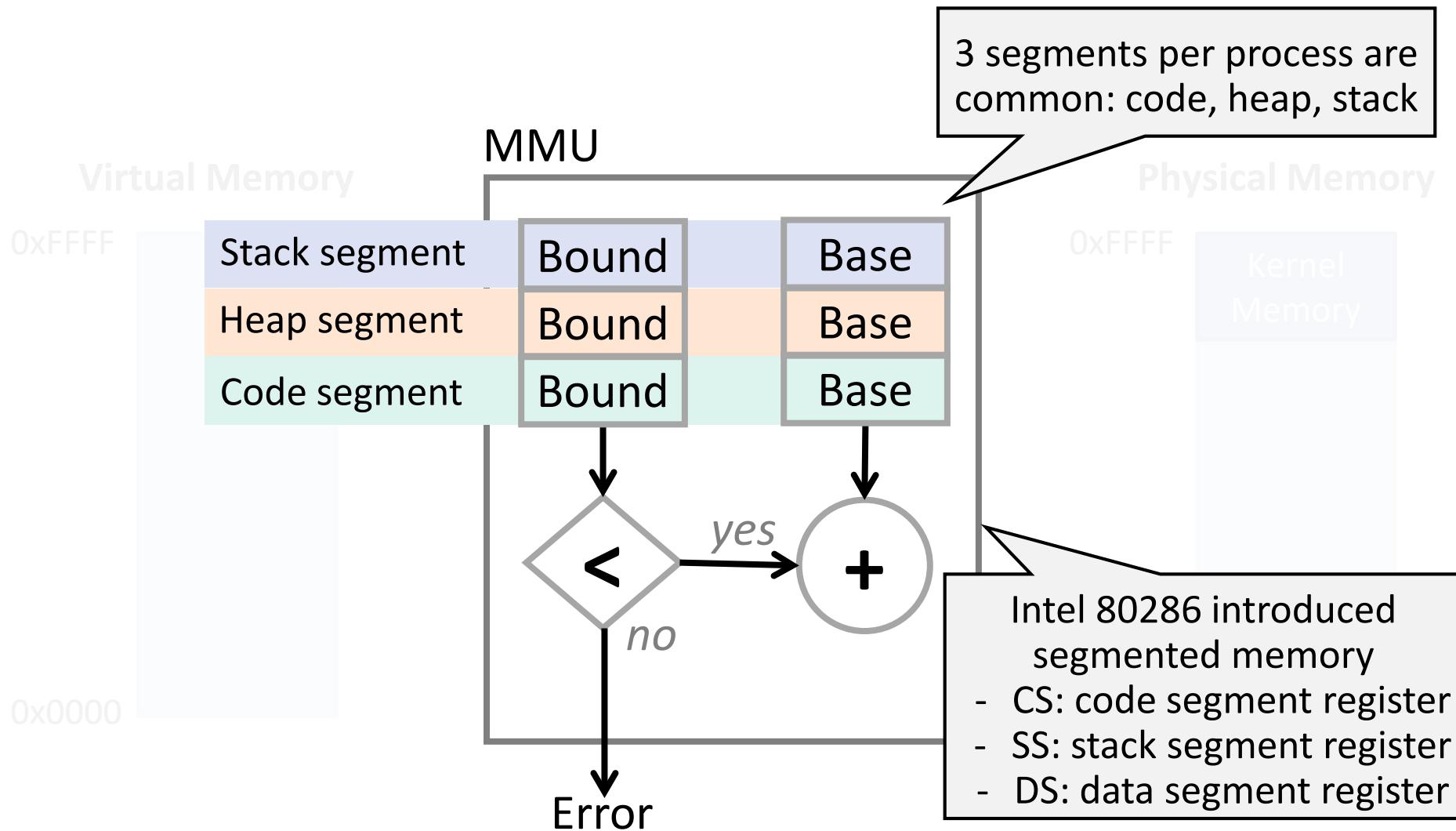


Towards Segmented Memory

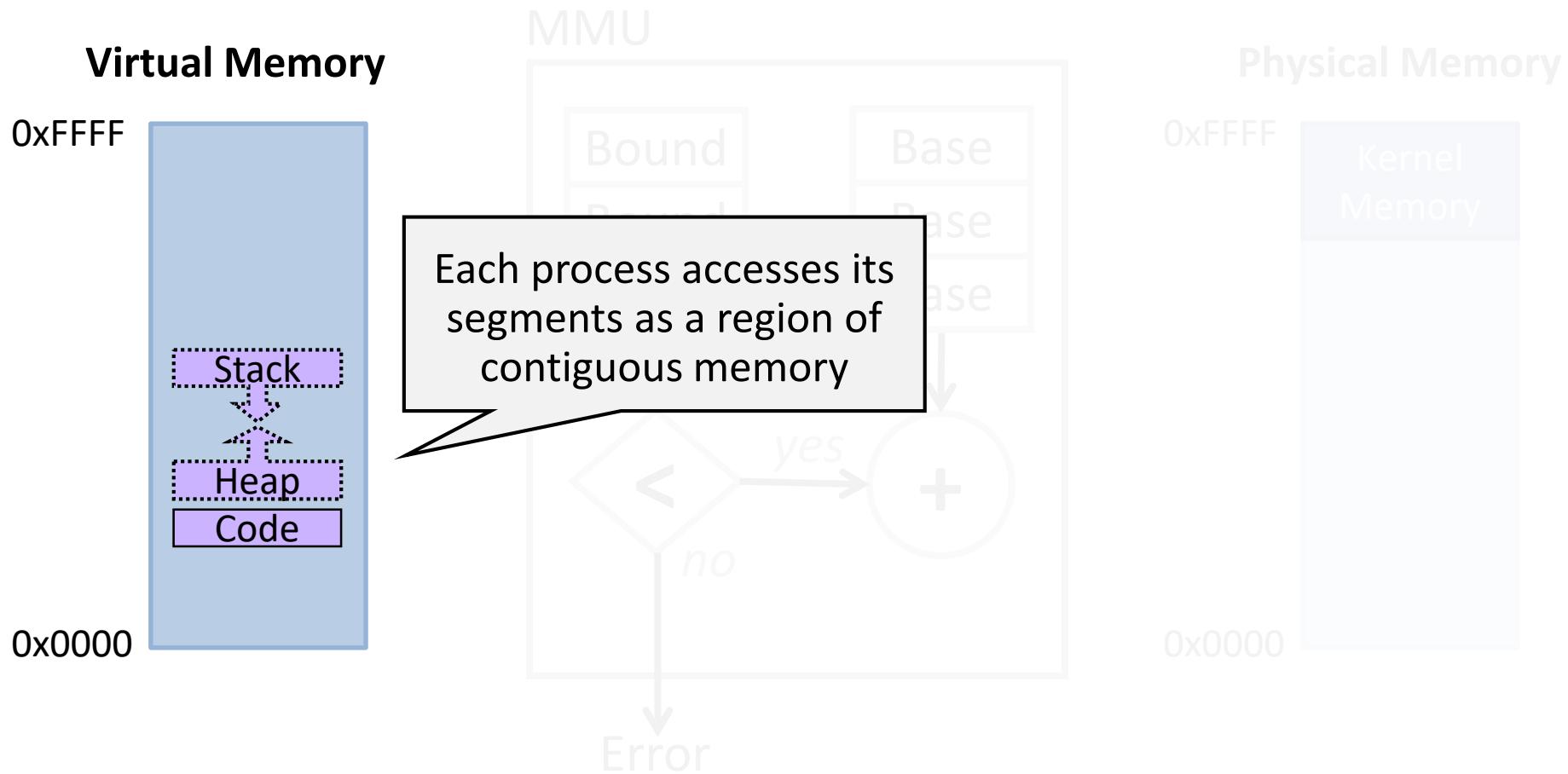
- Segmentation is a generalization of the base and bounds approach



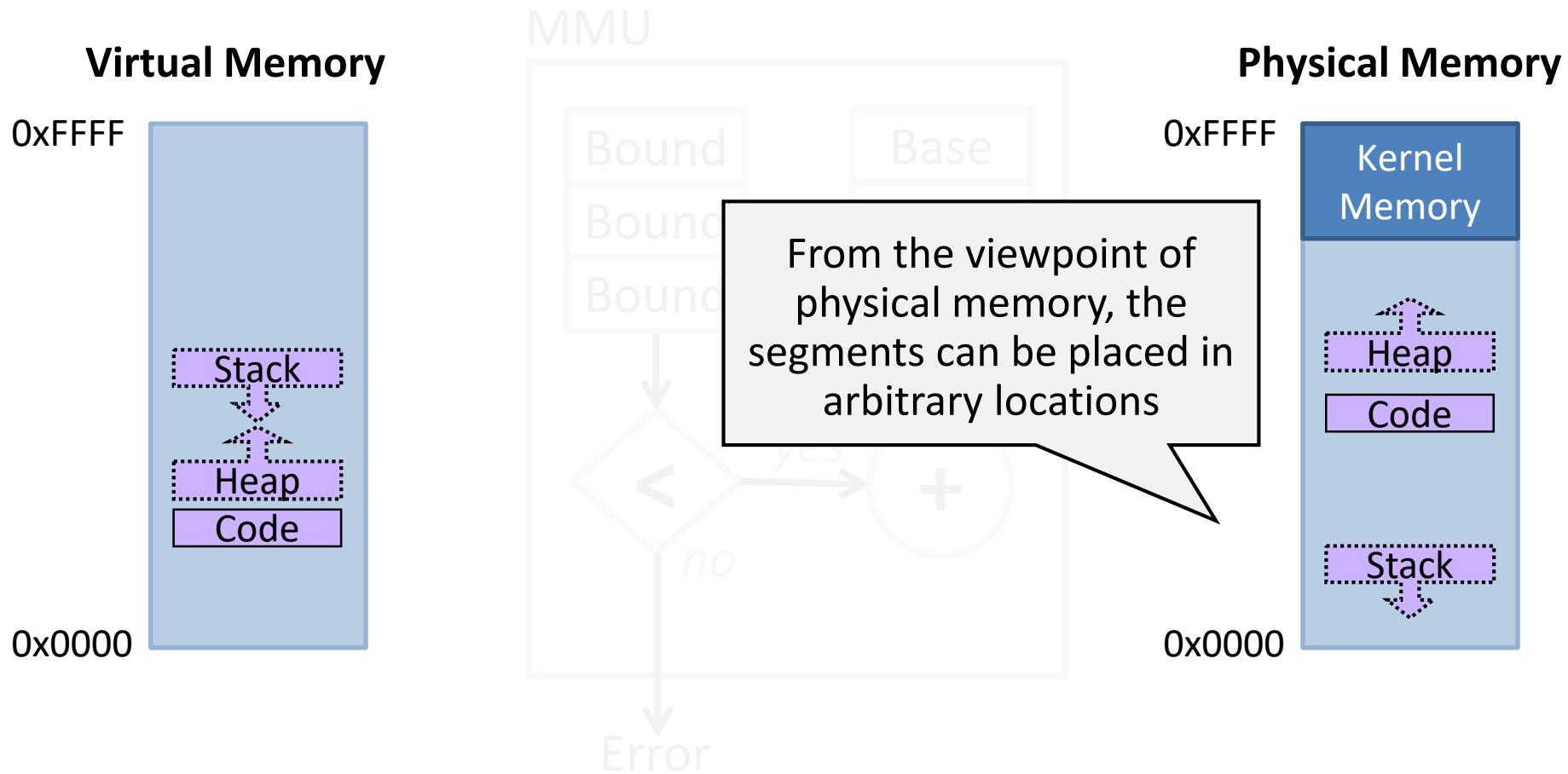
Segmentation Details



Segmentation Details

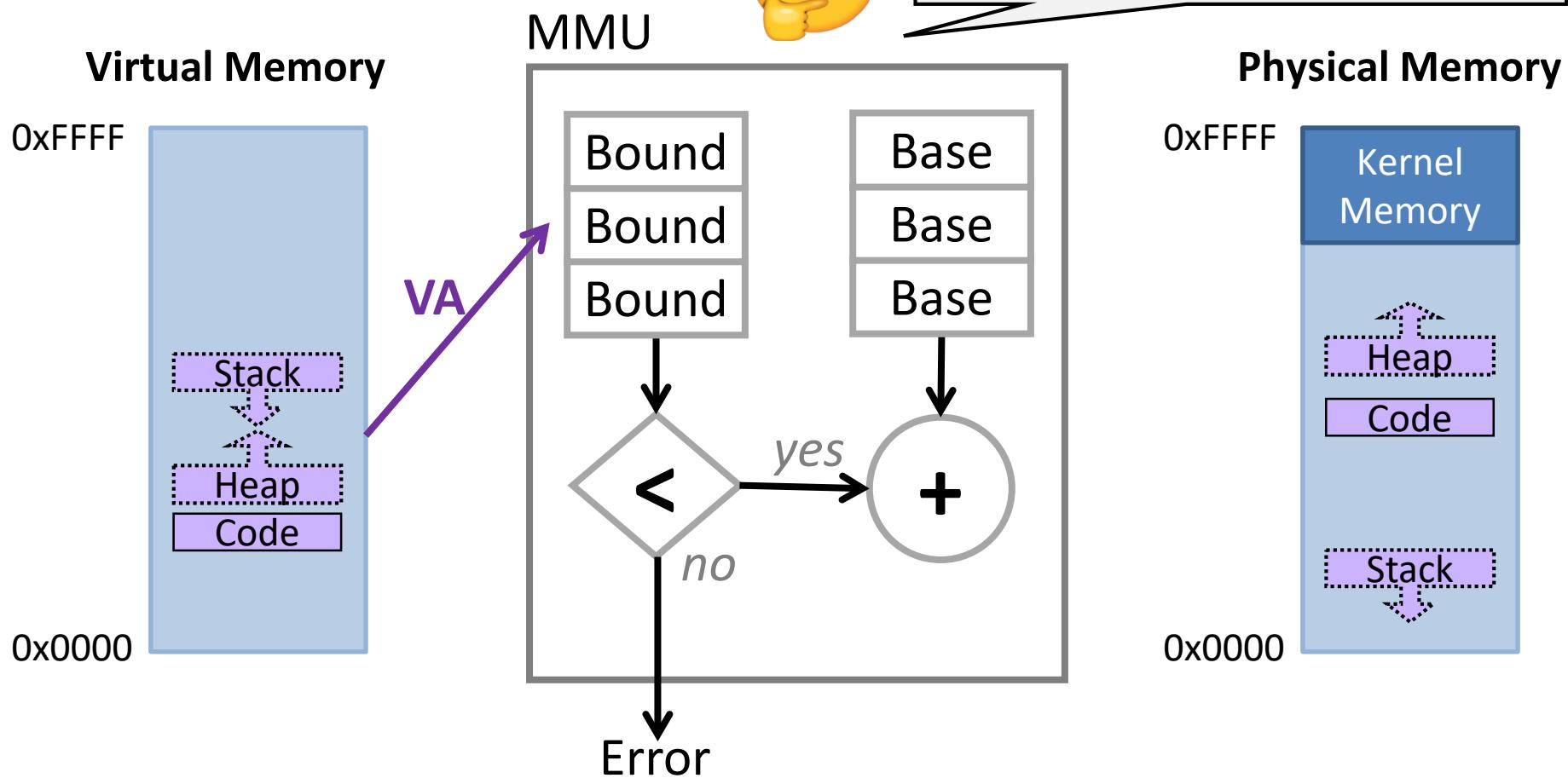


Segmentation Details



Segmentation Details

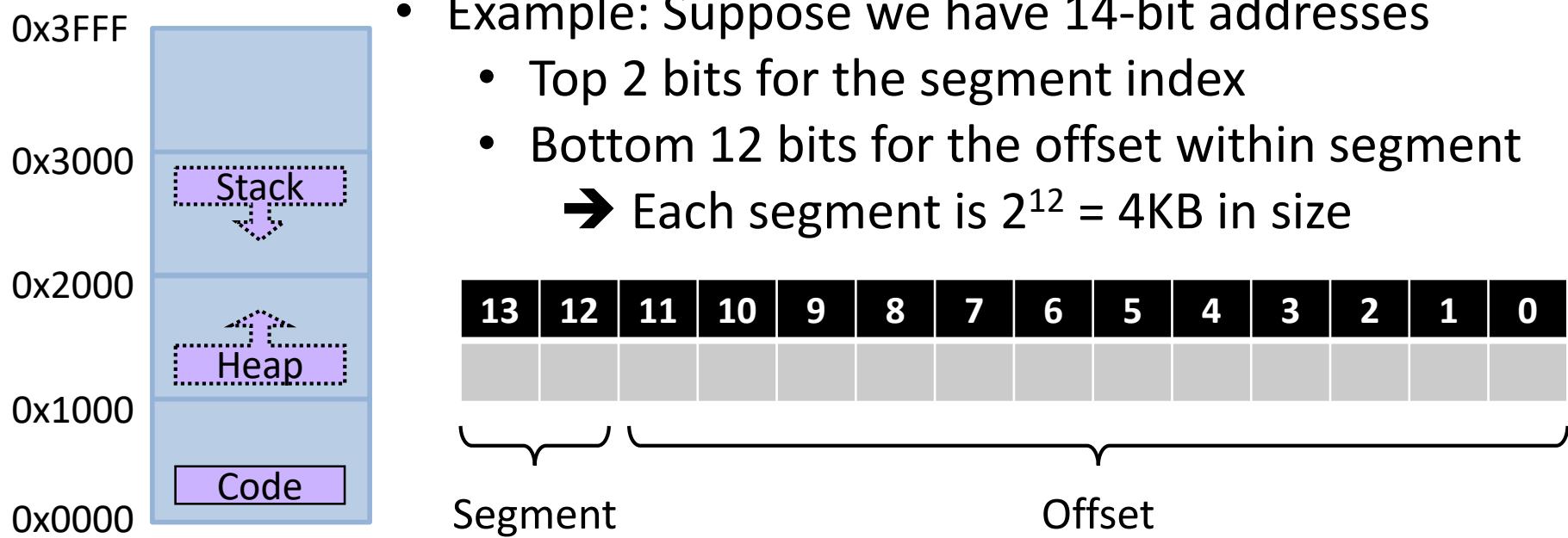
Now, we can have multiple segments per process. How then can CPU know where we have to access among them (based on a given virtual address) ?



Solution: Segments and Offsets

- Key idea: split virtual addresses into a segment index and an offset

Virtual Memory



Solution: Segments and Offsets

In 16-bit (real mode) x86 assembly, segment:offset notation is common

mov [ds:eax], 42

mov [esp], 23

→ Each segment is $2^{12} = 4\text{KB}$ in size





Solution: Segments and Offsets

- Key idea: split virtual addresses into a segment

Move '42' to the memory location designated by the EAX w/ the current VA's data segment



Example: Segment

- Top 2 bits
- Bottom 12 bits

In 16-bit (real mode) x86 assembly, segment:offset notation is common

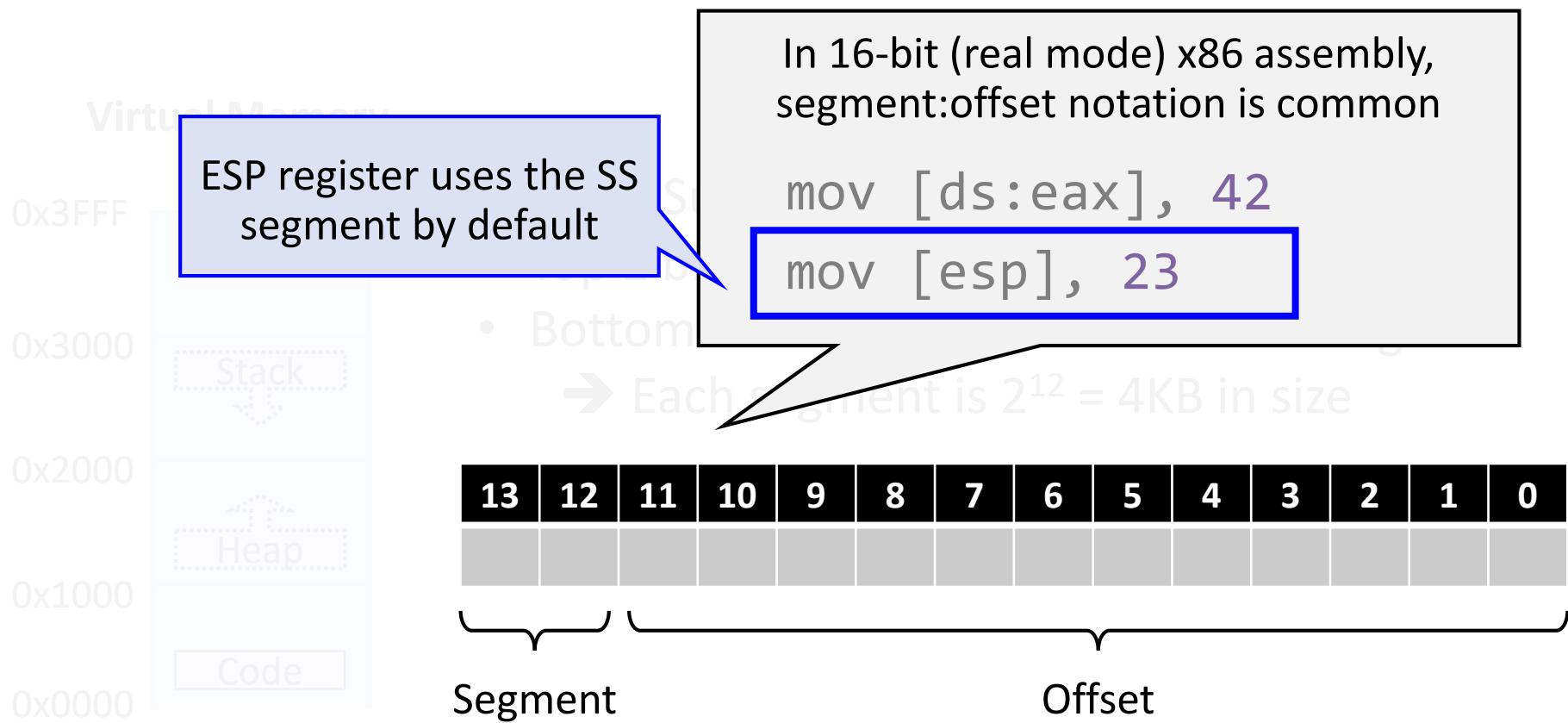
```
mov [ds:eax], 42
```

```
mov [esp], 23
```

→ Each segment is $2^{12} = 4\text{KB}$ in size

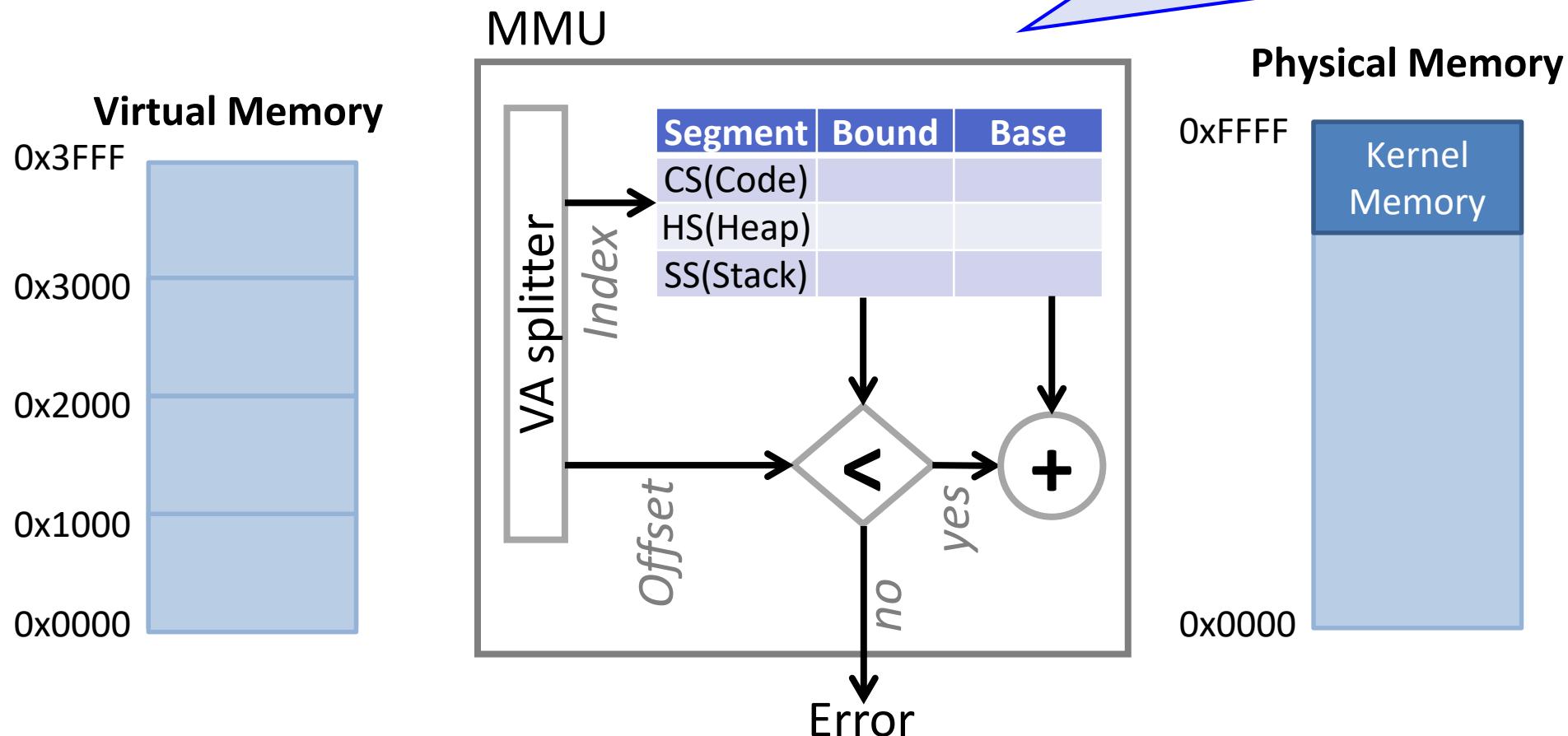


Solution: Segments and Offsets



Separation of Responsibility

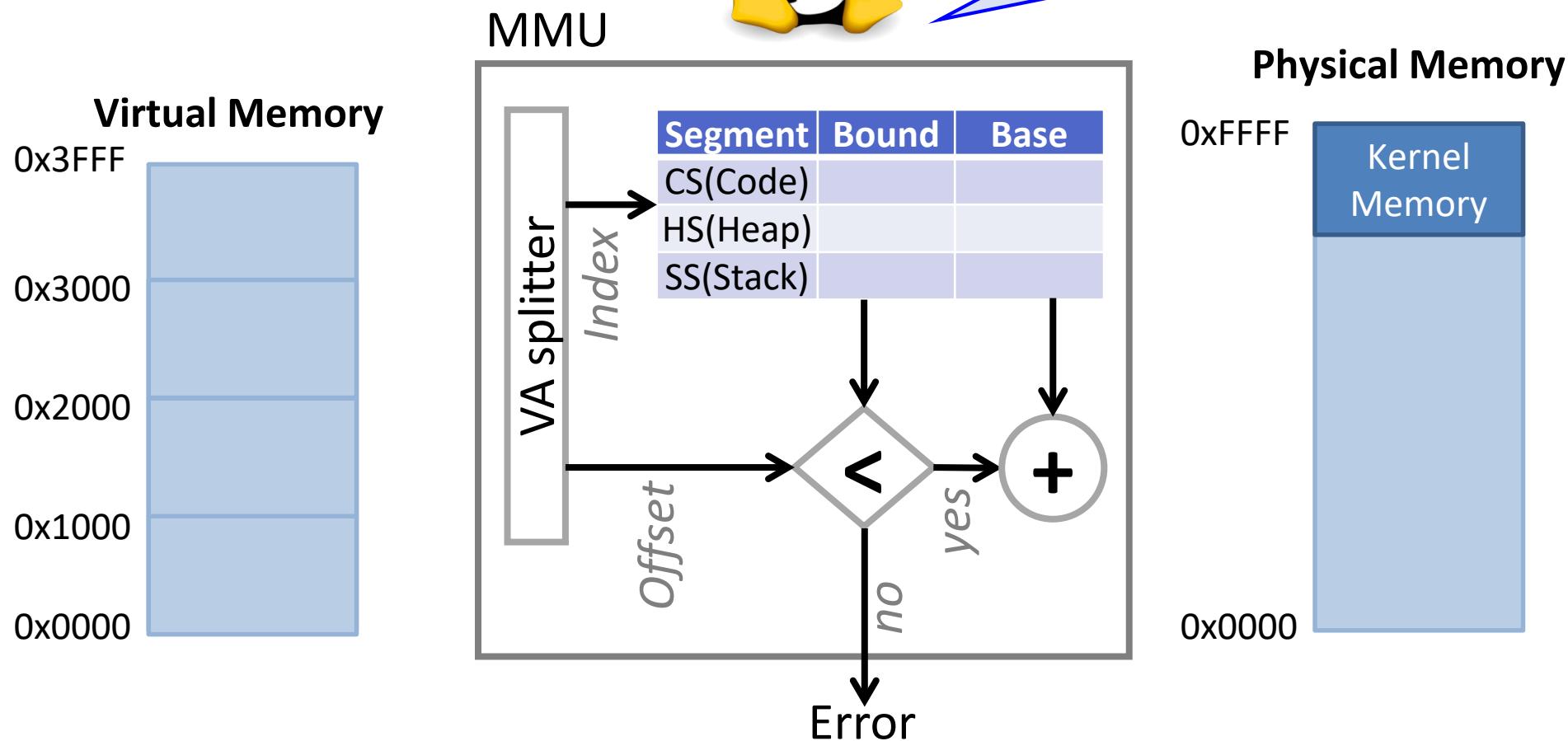
HW Support: CPU translates virtual addresses to physical addresses on demand using segment registers; their values are configured by OS though



Separation of Responsibility



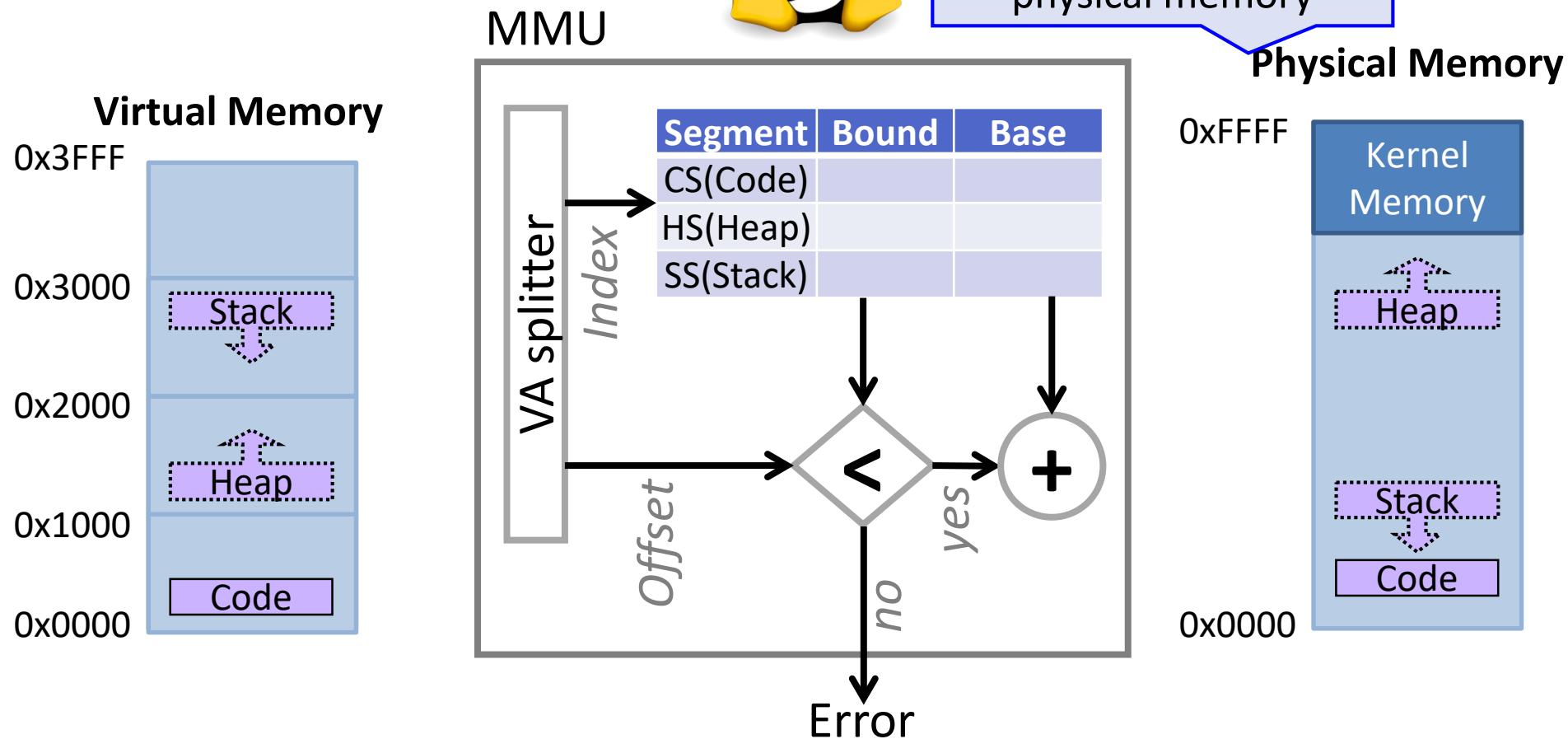
SW Support: The OS manages segments and their indexes



Separation of Responsibility



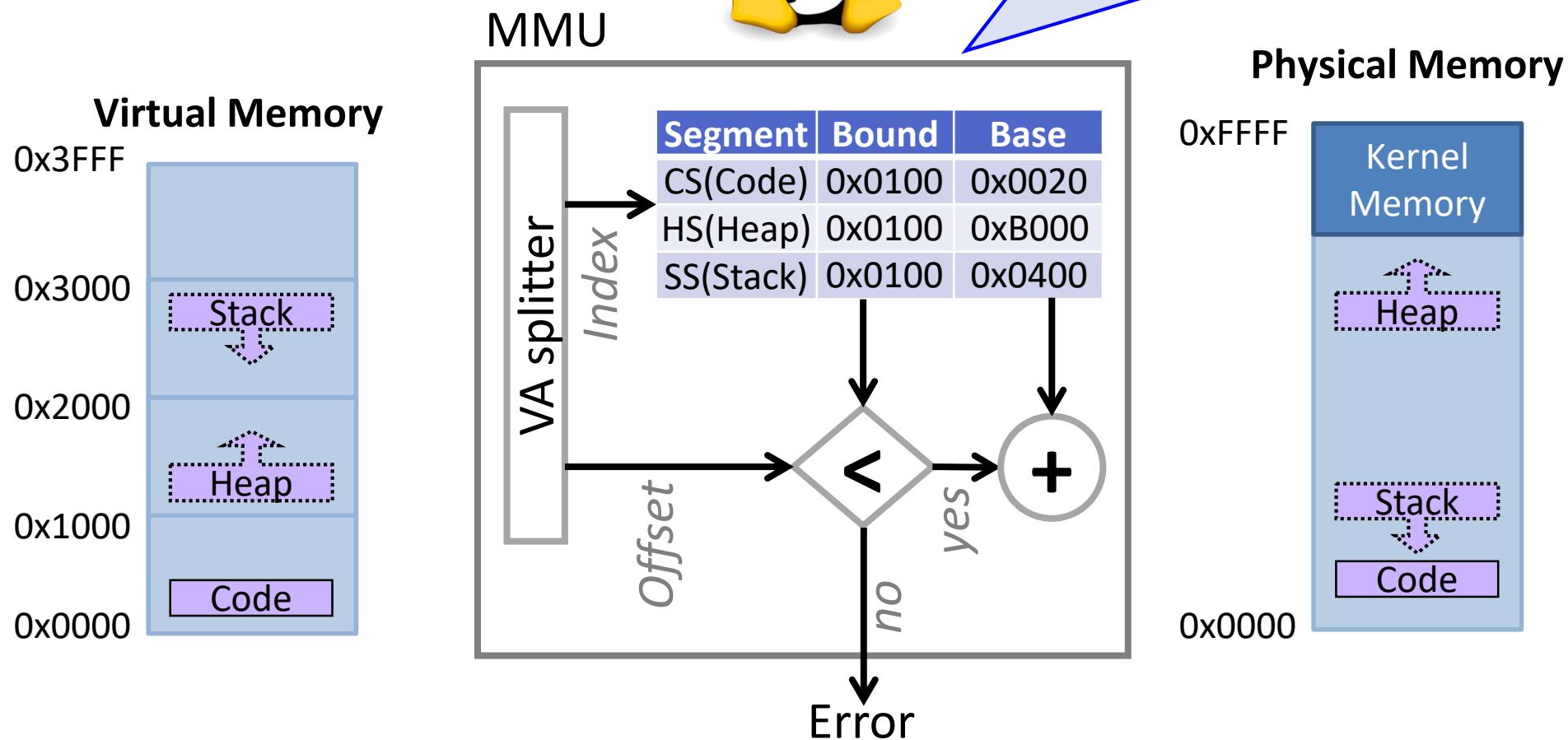
- ① Create segments for a new process in available spaces of the physical memory



Separation of Responsibility



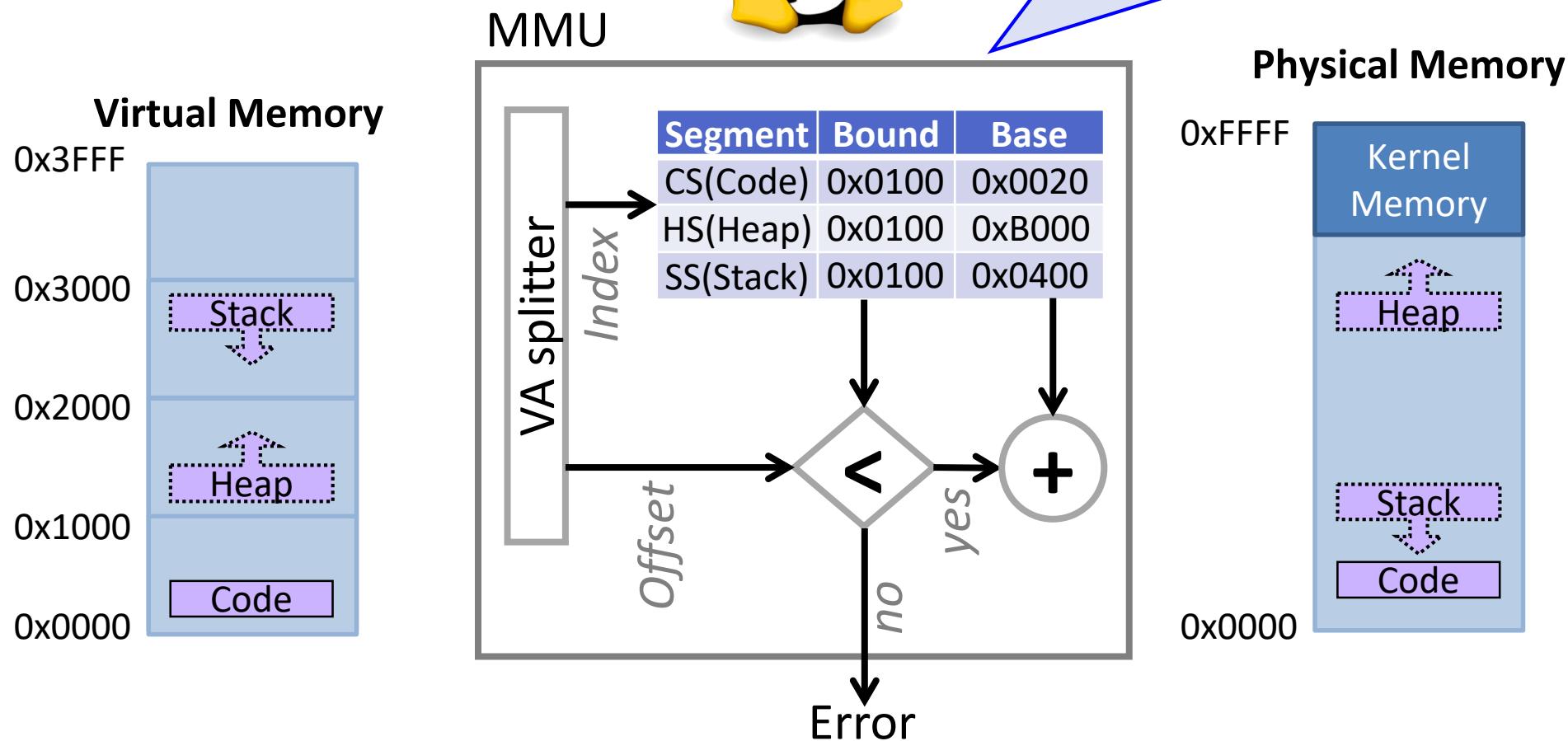
② Configure registers to let each segments to know its base address and bound appropriately



Separation of Responsibility



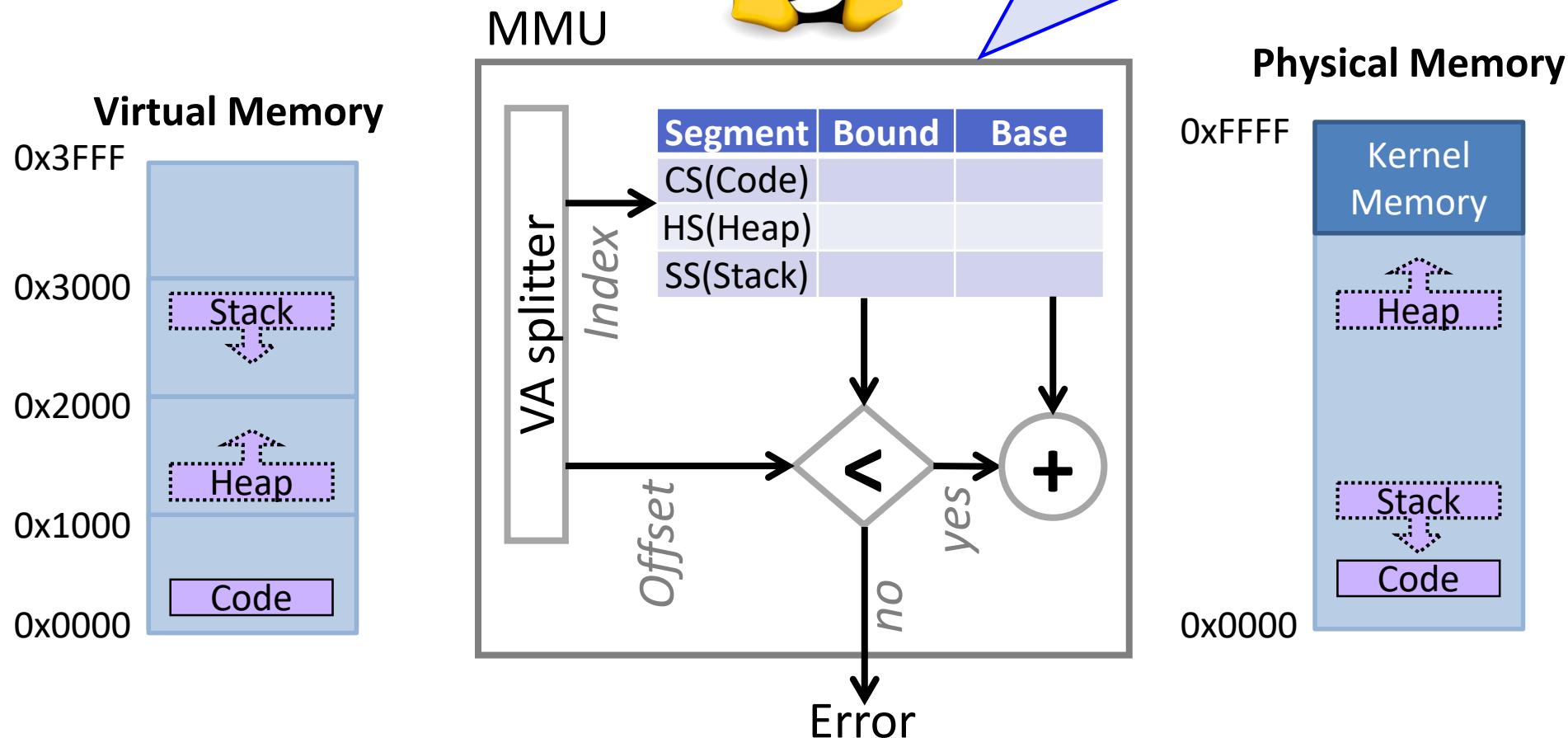
③ Swaps out the segment registers to the underlying memory during a context switch



Separation of Responsibility

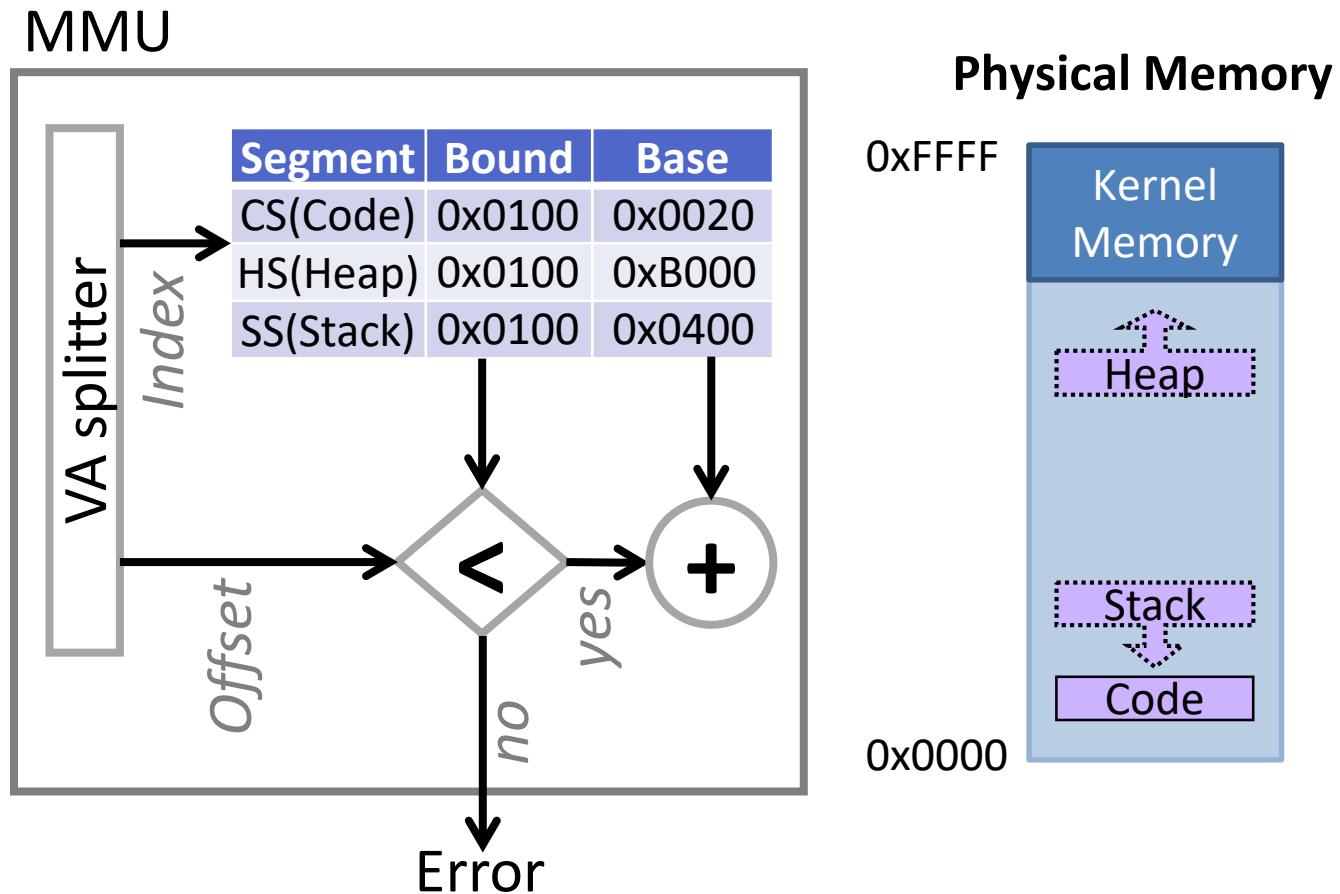
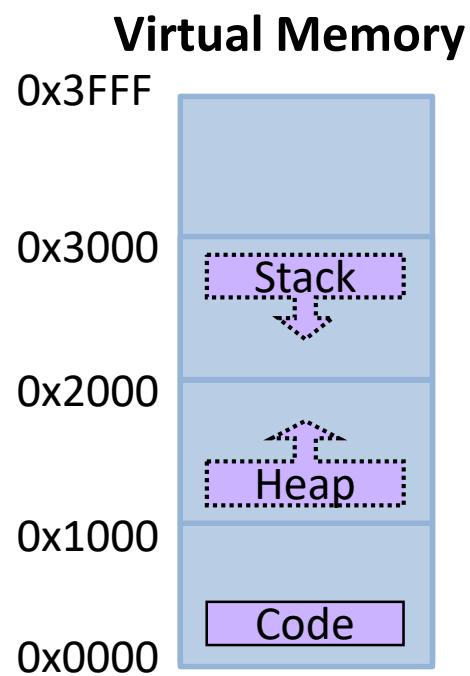


- 4 When process terminates, frees all the associated segments from the physical memory

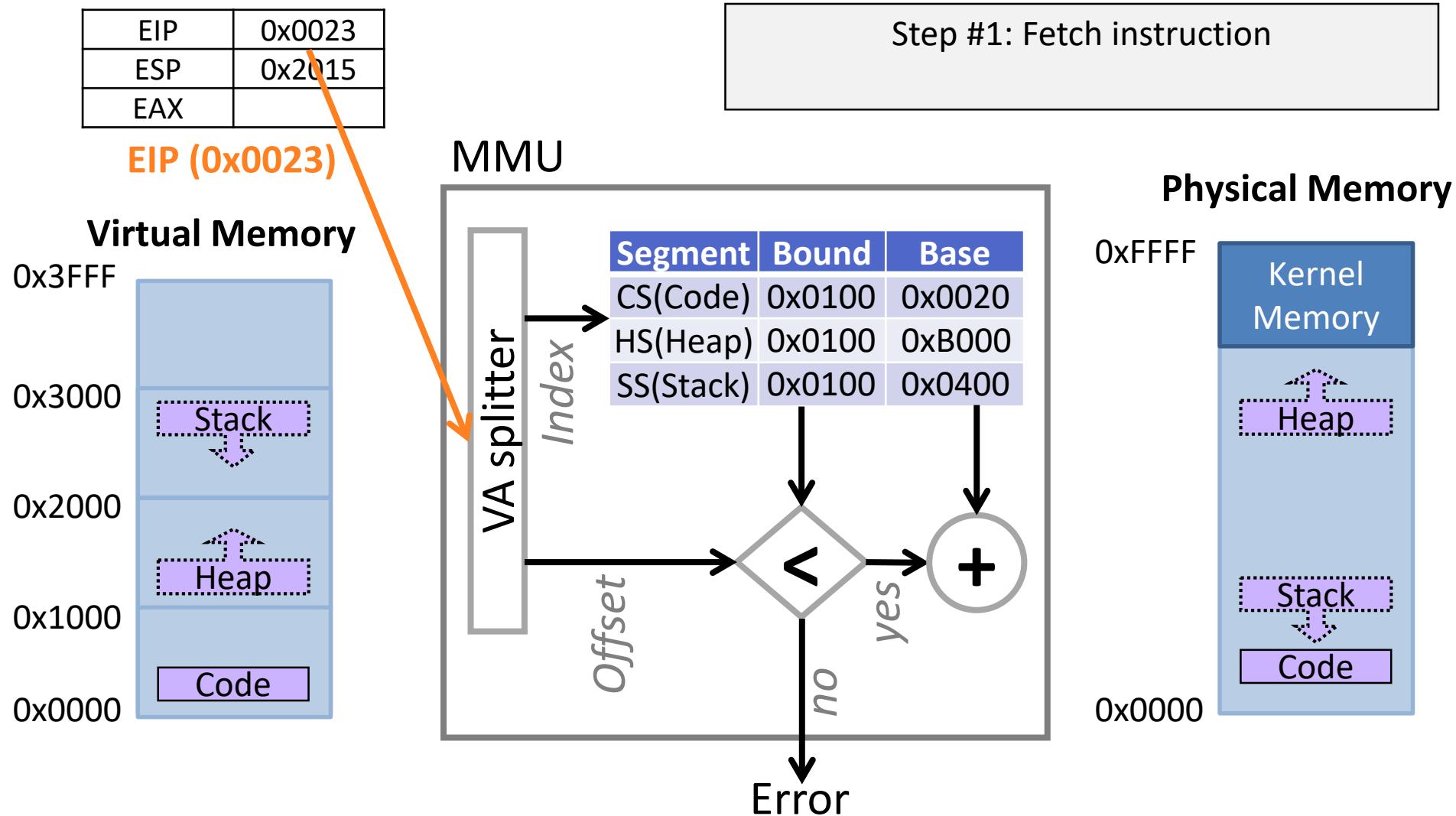


Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	



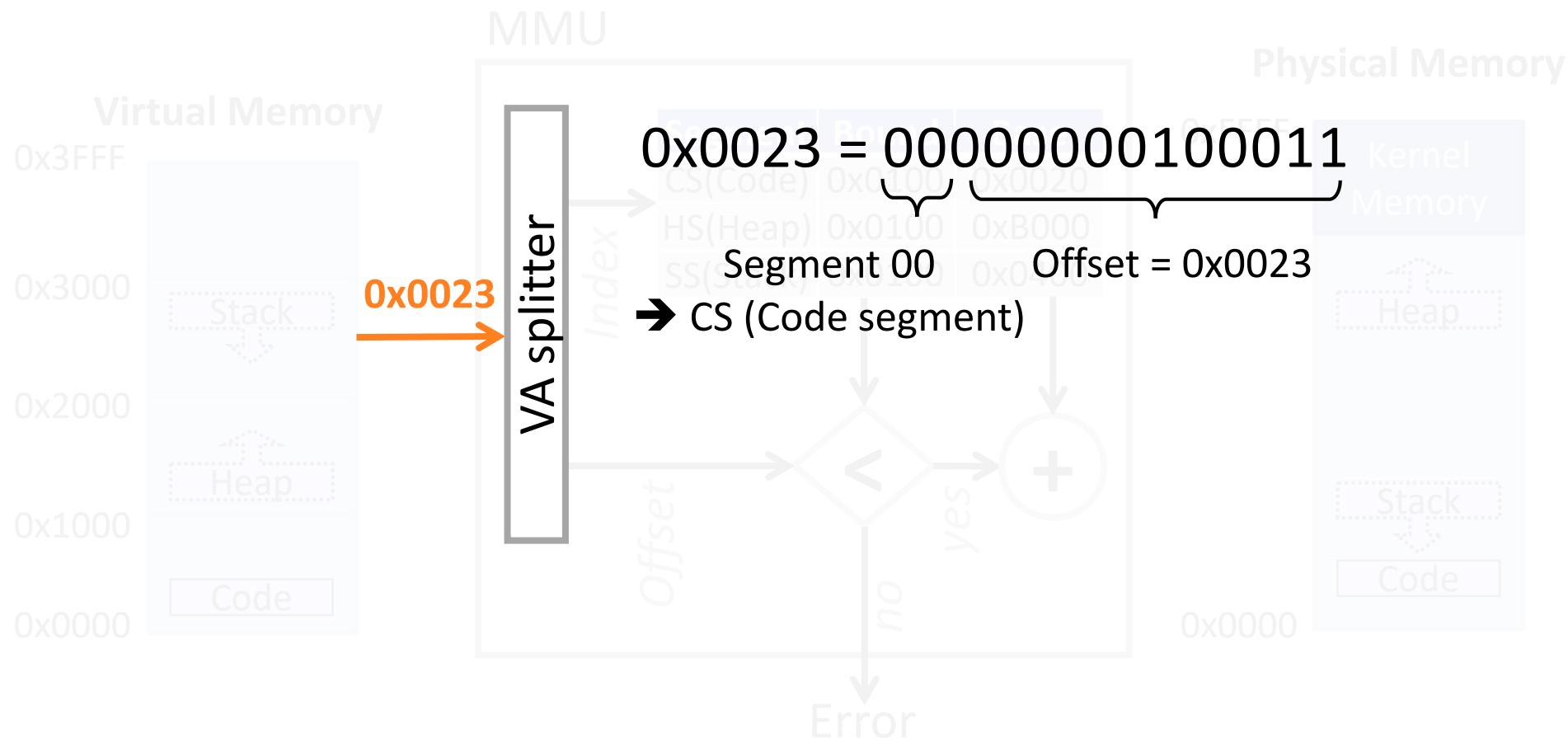
Segmentation Example



Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	

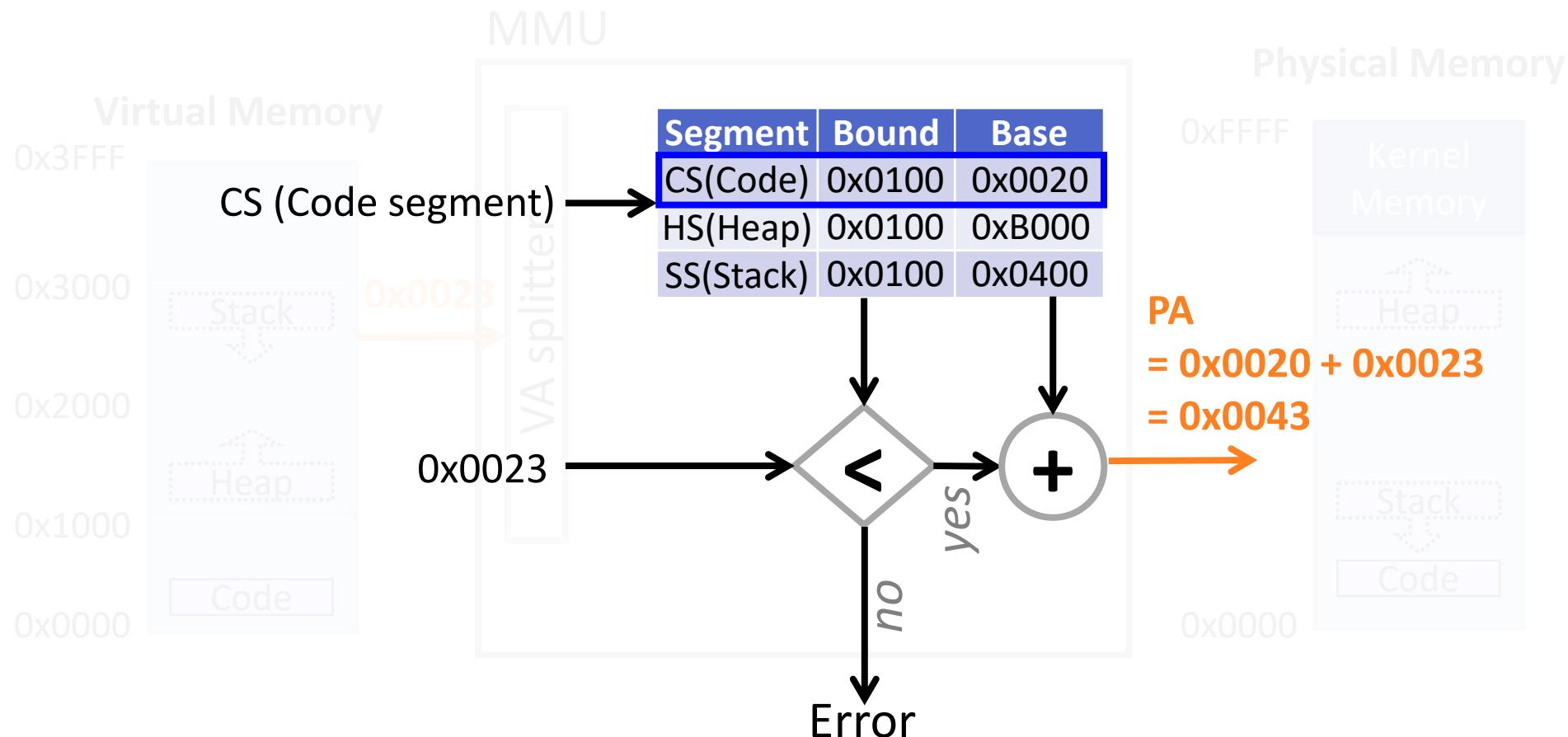
Step #1: Fetch instruction



Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	

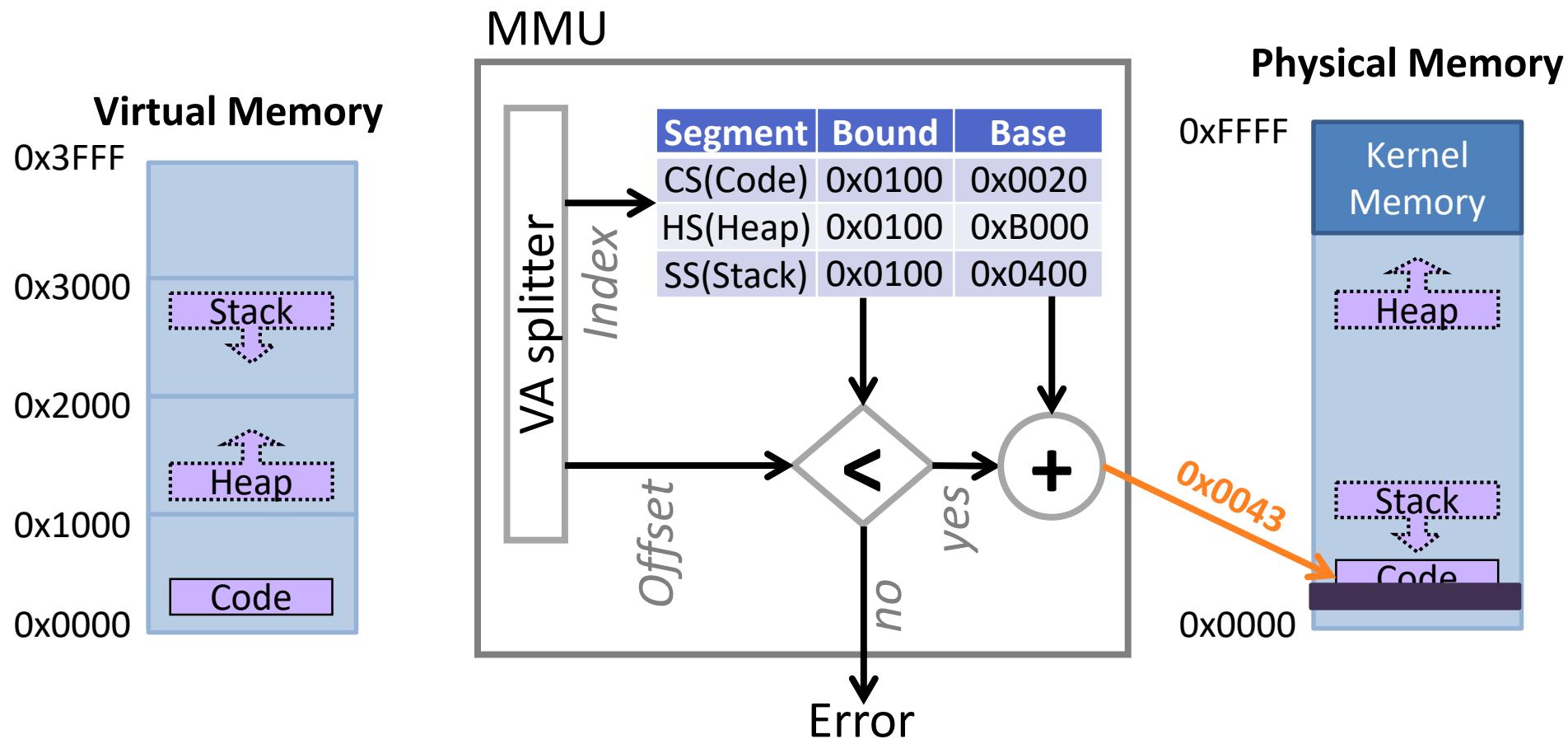
Step #1: Fetch instruction



Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	

Step #1: Fetch instruction
`mov eax, [esp]`

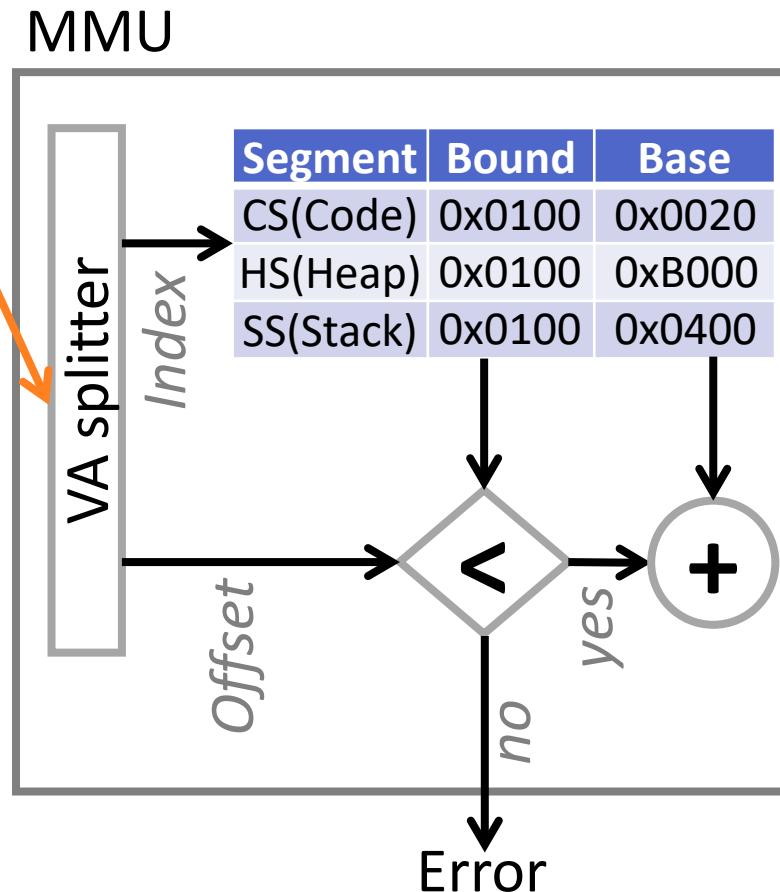
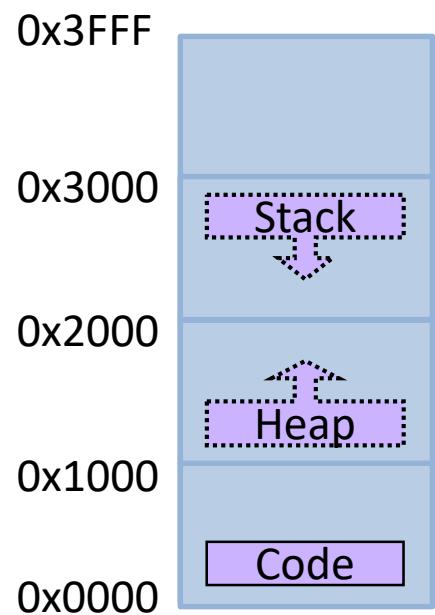


Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	

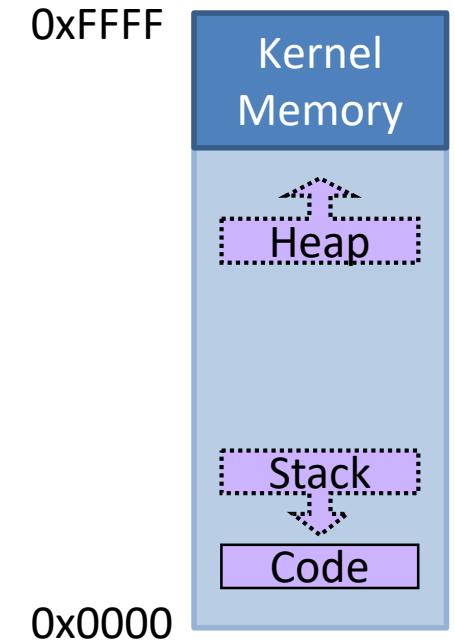
ESP (0x2015)

Virtual Memory



Step #2: Move ESP value to EAX
mov eax, [esp]

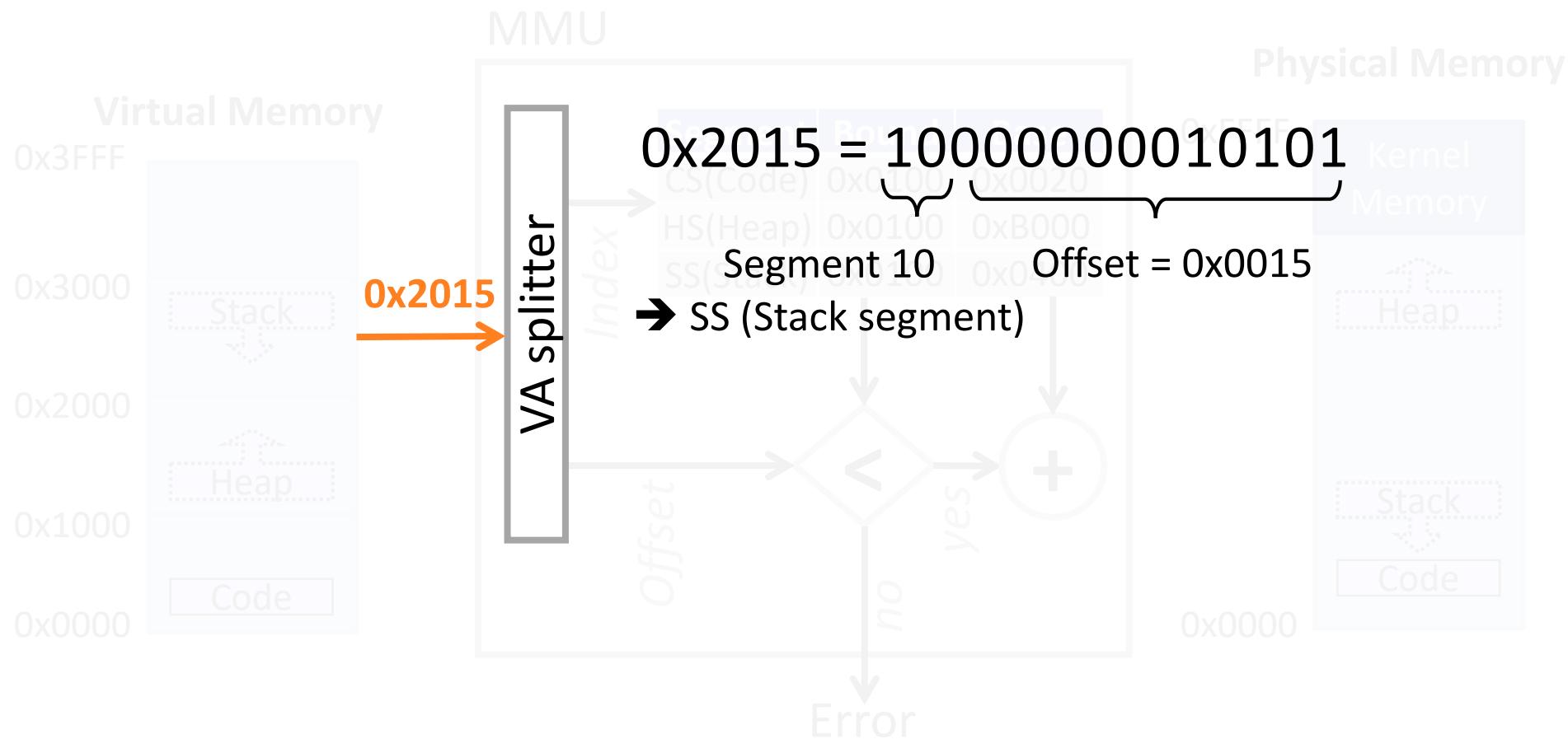
Physical Memory



Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	

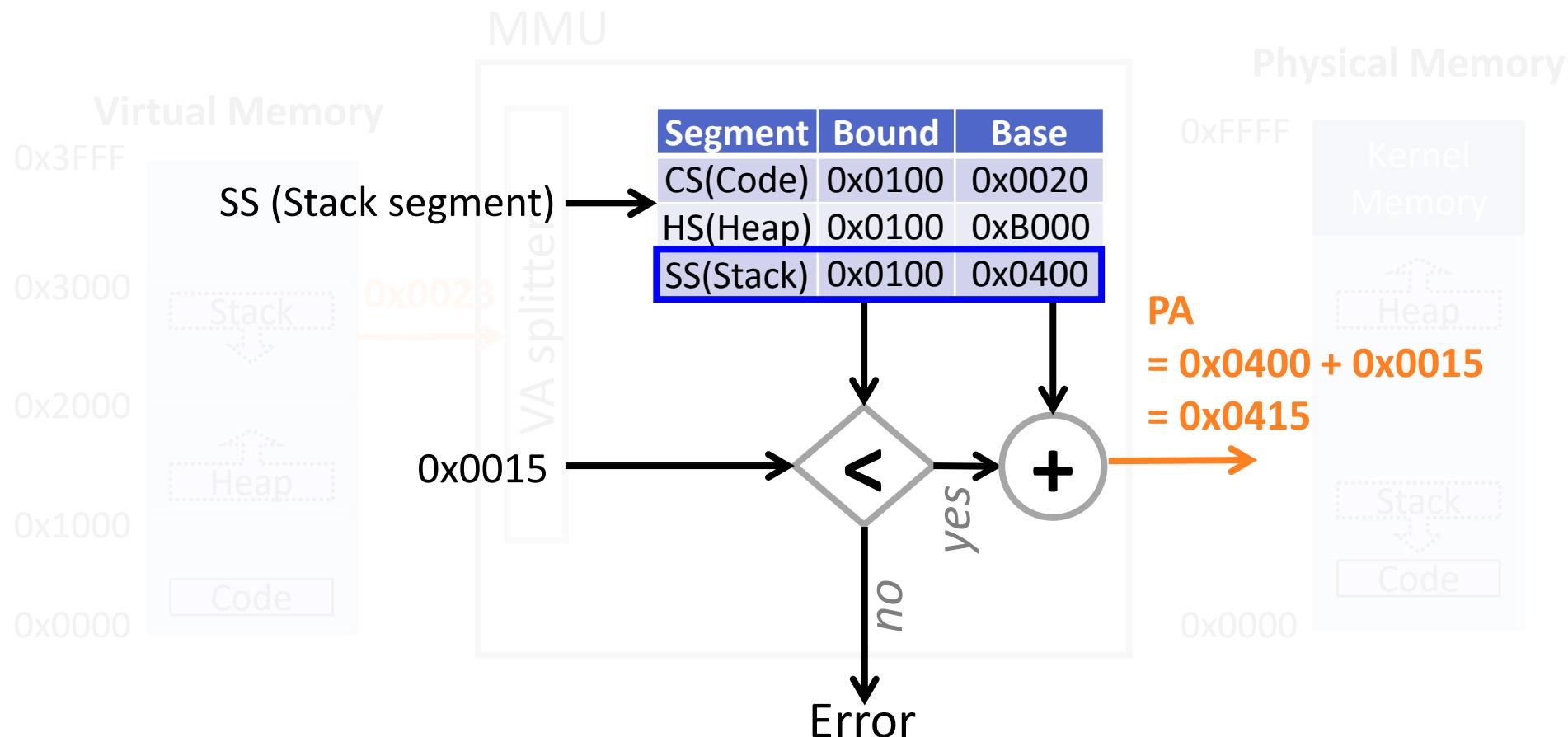
Step #2: Move ESP value to EAX
`mov eax, [esp]`



Segmentation Example

EIP	0x0023
ESP	0x2015
EAX	

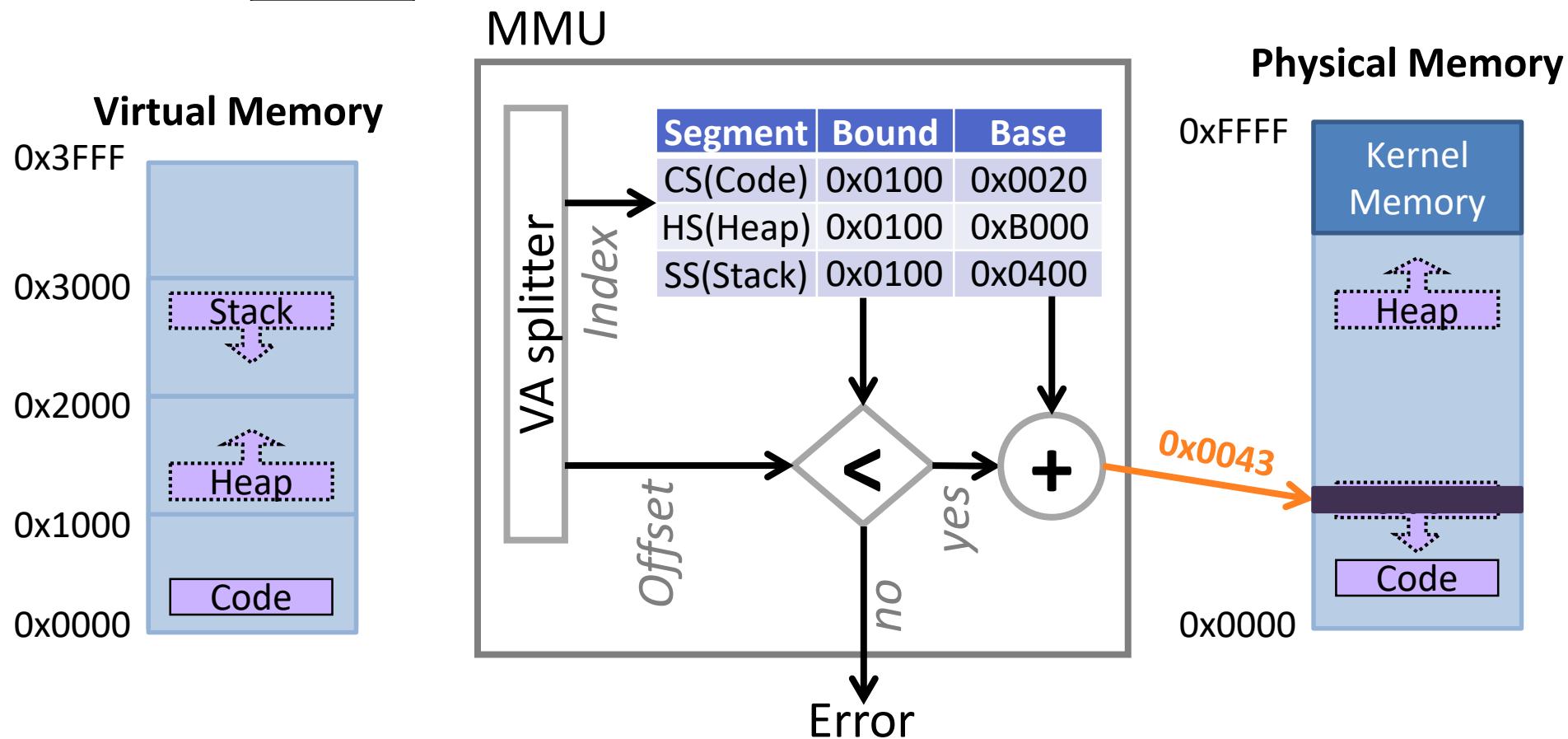
Step #2: Move ESP value to EAX
`mov eax, [esp]`



Segmentation Example

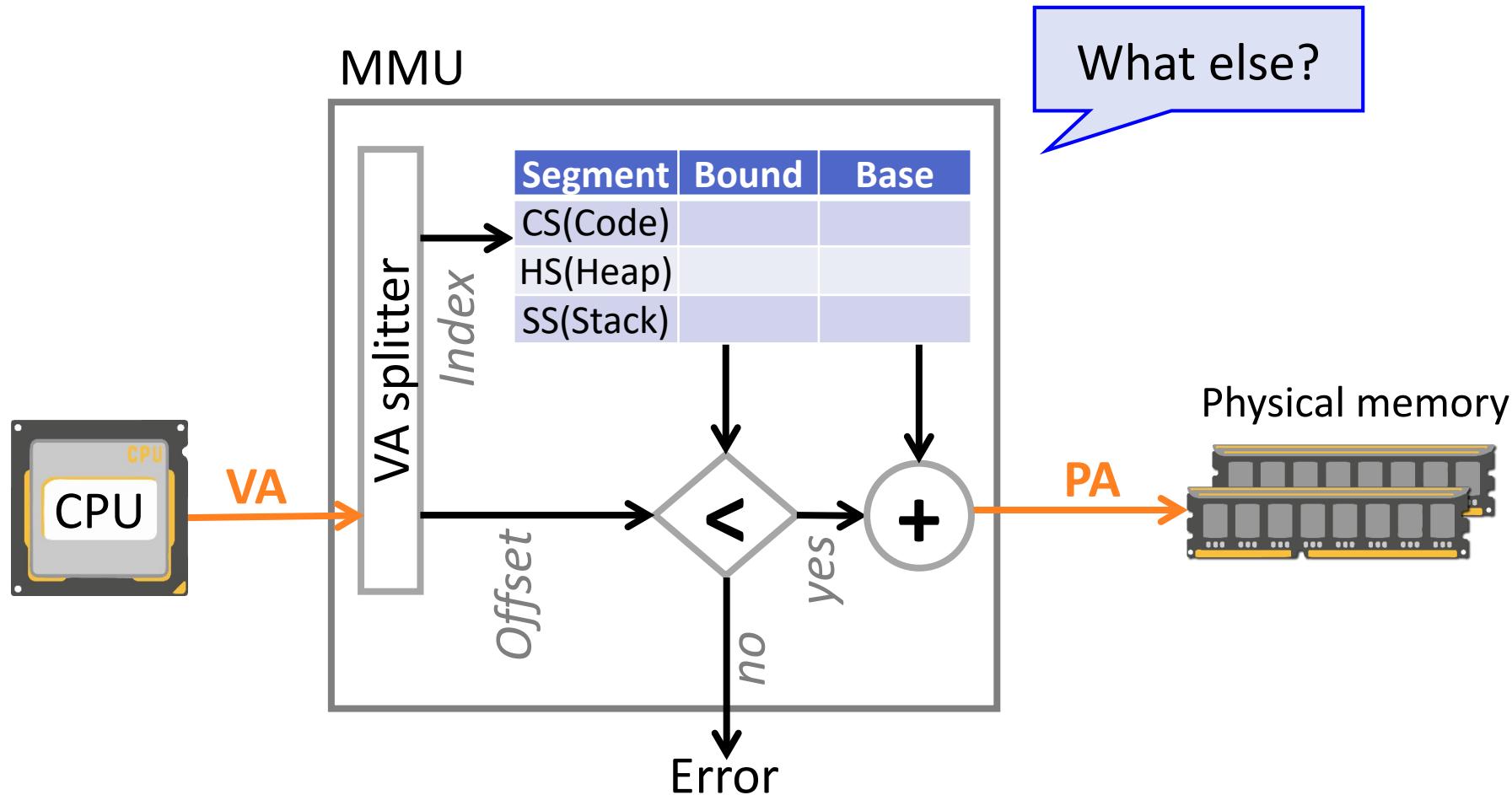
EIP	0x0023
ESP	0x2015
EAX	[ESP]

Step #2: Move ESP value to EAX
`mov eax, [esp]`



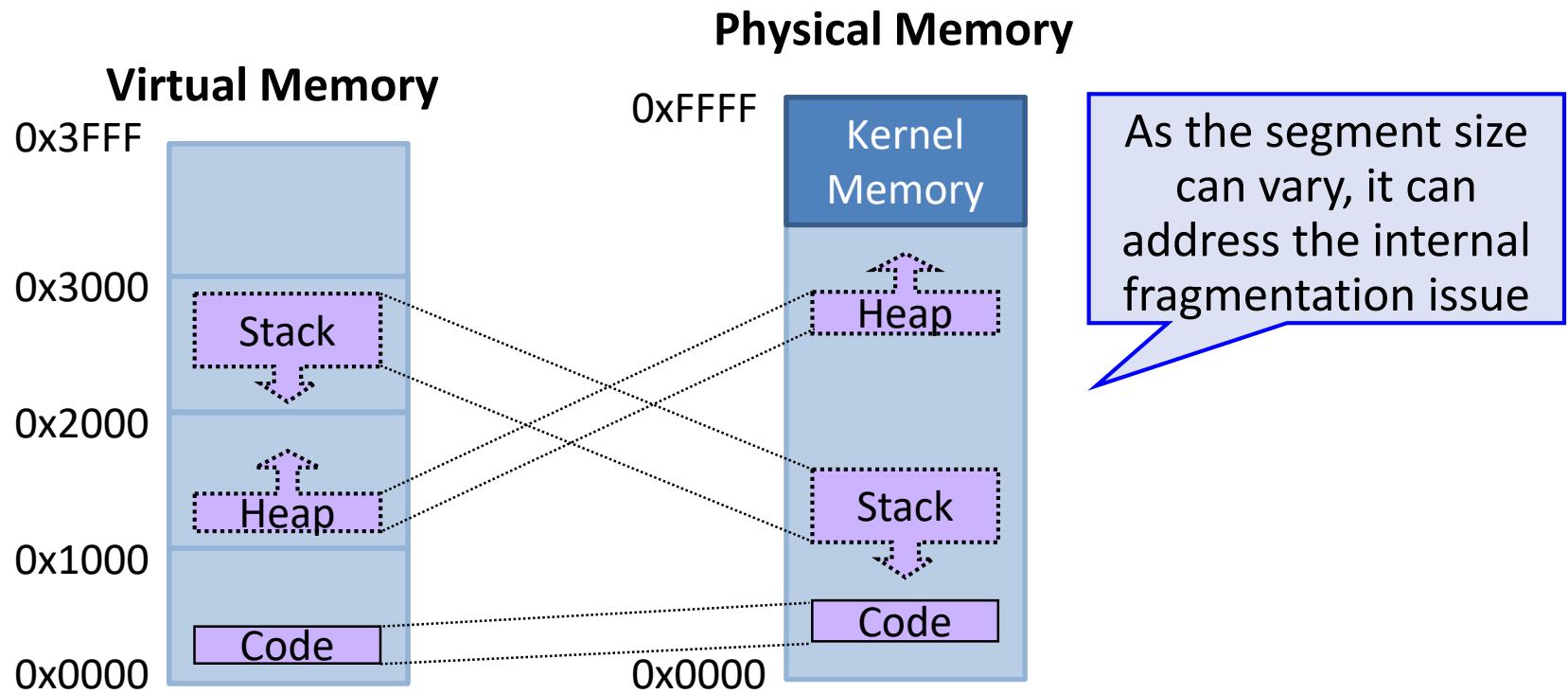
Advantages of Segmentation

- Segmentation can take the advantages in all aspect of base and bound



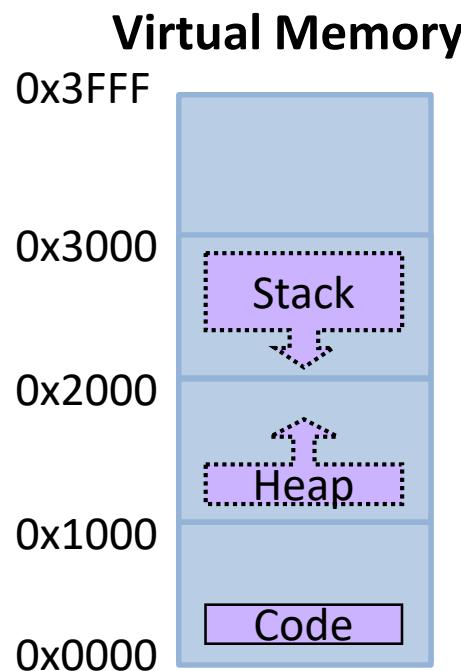
#1: Better For Sparse Address Spaces

- Code, heap, and stack are individually managed by a “separate” memory space



#2: Per Segment Permissions

- Many CPUs (including x86) support permissions on segments (R: read, W: write, X: executable)

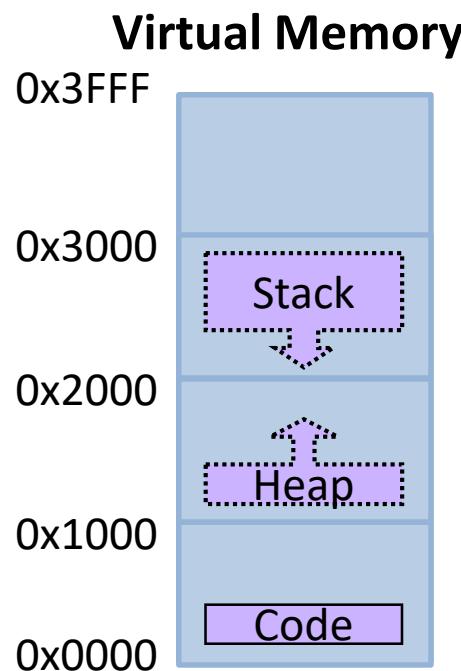


What if a process tries to write(W) something on the code segment?

Index	Segment	Bound	Base	Permissions
00	CS(Code)	0x0100	0x0020	RX
01	HS(Heap)	0x0100	0xB000	RW
10	SS(Stack)	0x0100	0x0400	RW

#2: Per Segment Permissions

- Many CPUs (including x86) support permission control along with segments (R: read, W: write, X: executable)



Same with the bound exception,
a malicious/wrong access triggers
the corresponding
segmentation's fault exception

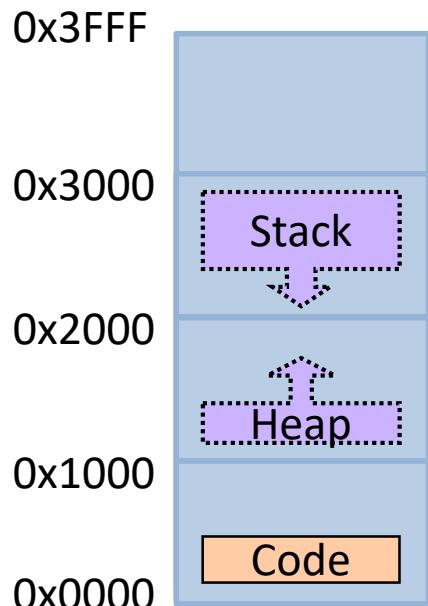
```
gcc release.c -o release && ./release
executing code...
loading wave data
artist : telestart
bpm : 160.00
peak : -0.00 db
size : 29.63 mb
audio data length: 02:56 min
key : f minor -- initialising audio playback...
```

segmentation fault.
(Core dumped)

#3: Supports Shared Memory

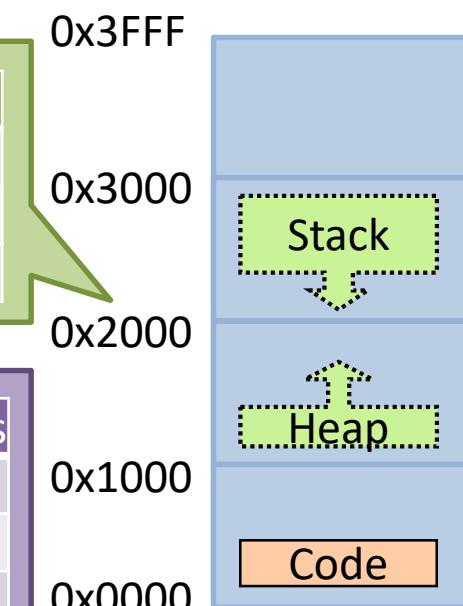
- By using the information of segments, multiple processes can easily share their memory

Process 1's View of Virtual Memory



Segment	Bound	Base	Permissions
CS(Code)	0x0100	0x0020	RX
HS(Heap)	0x0100	0xC000	RW
SS(Stack)	0x0100	0x0600	RW

Process 2's View of Virtual Memory

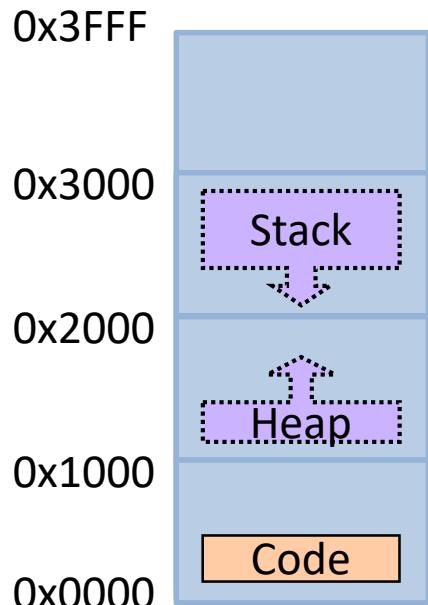


Segment	Bound	Base	Permissions
CS(Code)	0x0100	0x0020	RX
HS(Heap)	0x0100	0xB000	RW
SS(Stack)	0x0100	0x0400	RW

#3: Supports Shared Memory

- By using the information of segments, multiple processes can easily share their memory

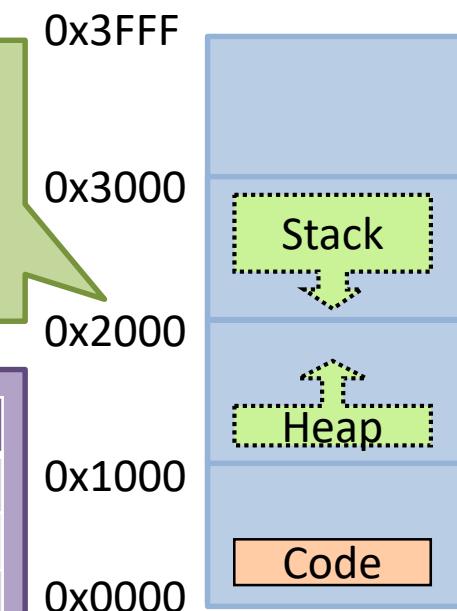
Process 1's View of Virtual Memory



Same physical segment for Code			
Segment	Bound	Base	Permissions
CS(Code)	0x0100	0x0020	RX
HS(Heap)	0x0100	0xC000	RW
SS(Stack)	0x0100	0x0600	RW

Same physical segment for Code			
Segment	Bound	Base	Permissions
CS(Code)	0x0100	0x0020	RX
HS(Heap)	0x0100	0xB000	RW
SS(Stack)	0x0100	0x0400	RW

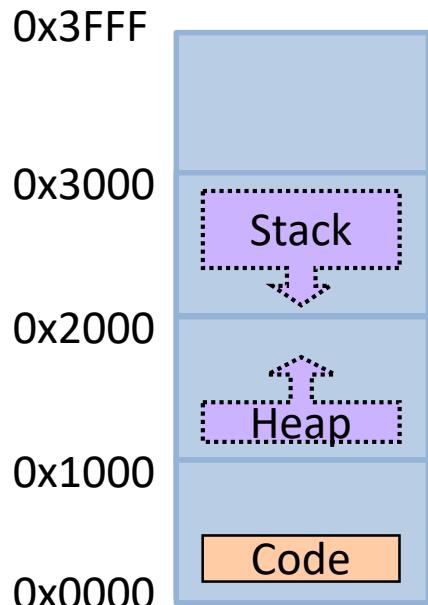
Process 2's View of Virtual Memory



#3: Supports Shared Memory

- By using the information of segments, multiple processes can easily share their memory

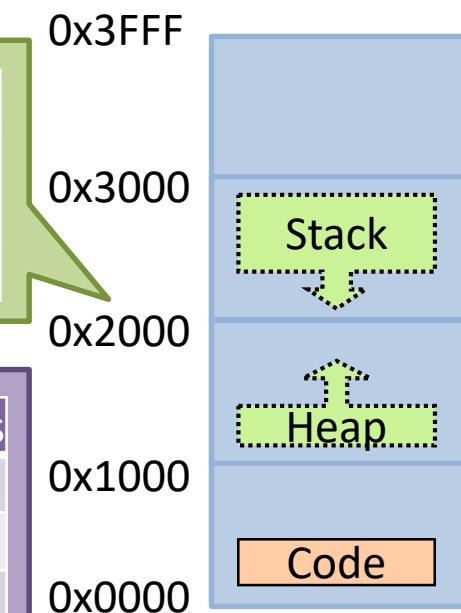
Process 1's View of Virtual Memory



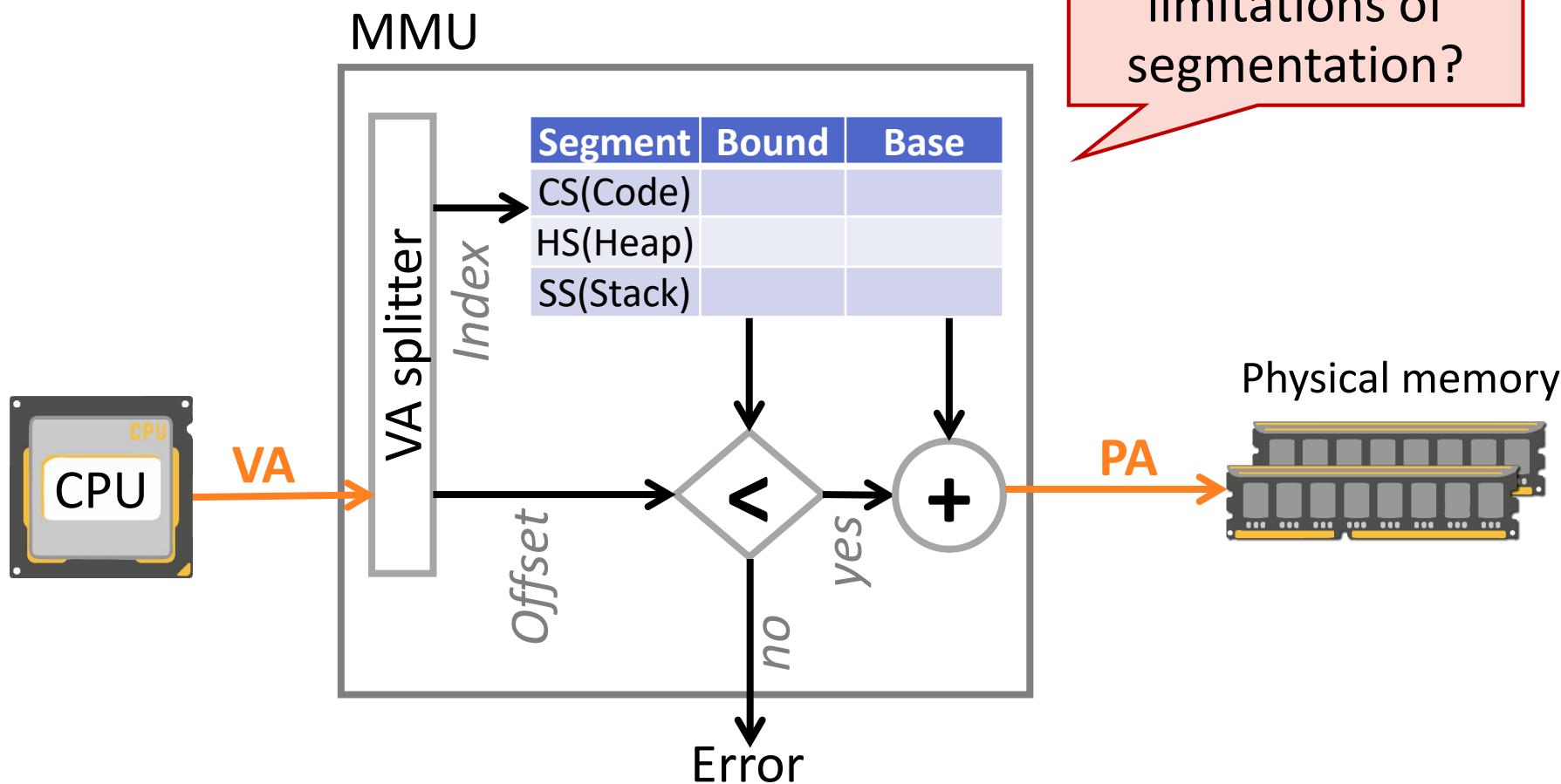
Different physical segment for Stack and Heap (which has app data)

Segment	Bound	Base	Permissions
CS(Code)	0x0100	0x0020	RX
HS(Heap)	0x0100	0xC000	RW
SS(Stack)	0x0100	0x0600	RW

Process 2's View of Virtual Memory



Limitations of Segmentation



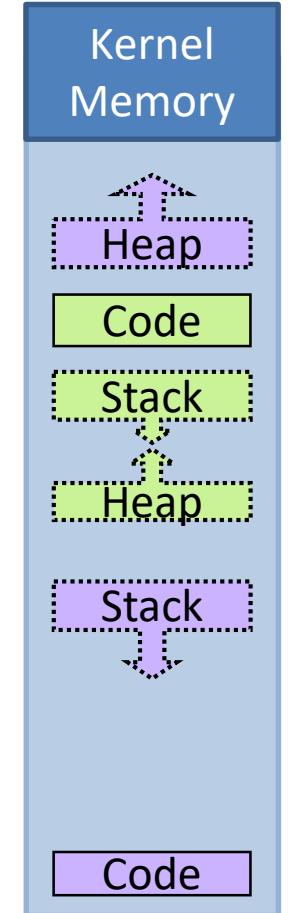


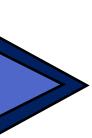
Problem: External Fragmentation

- Memory gets broken into random size, non-contiguous pieces

Physical Memory

Variable size segments can lead to external fragmentation



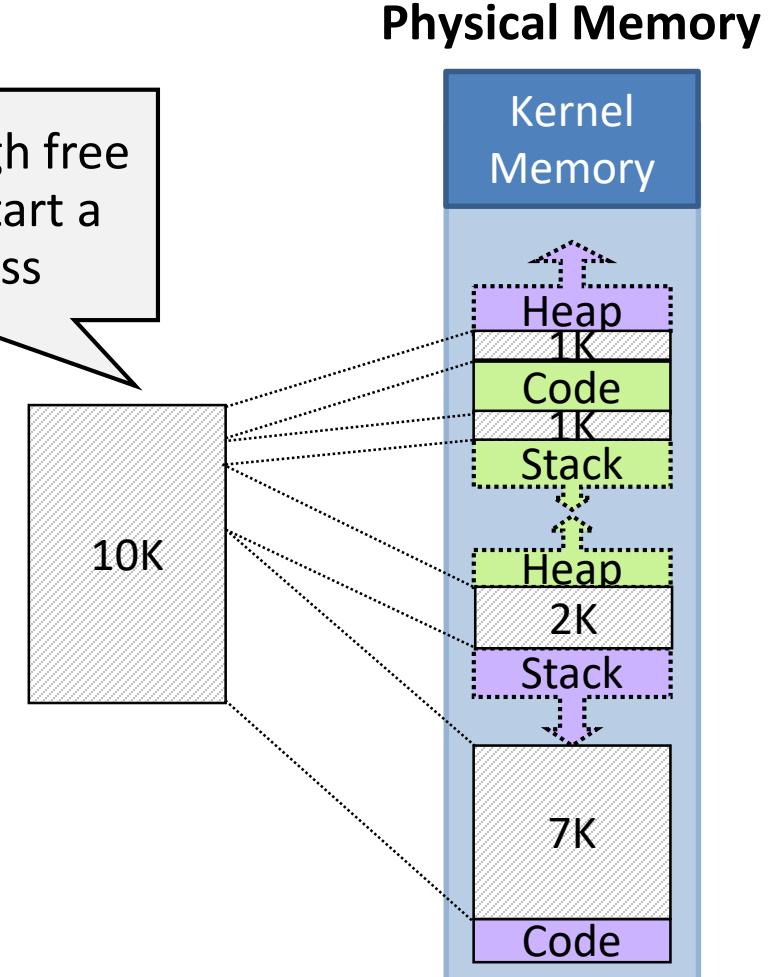
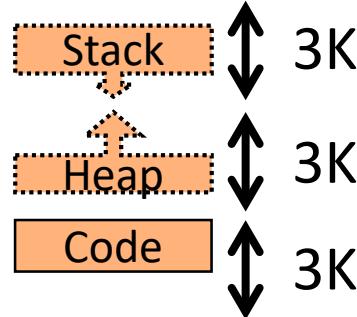


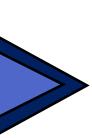
Problem: External Fragmentation

- Memory gets broken into random size, non-contiguous pieces

Example scenario:
New process requires
totally 9K memory
spaces for segments

There is enough free
memory to start a
new process

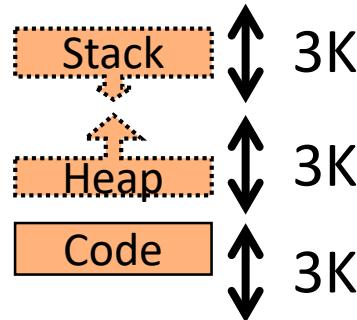




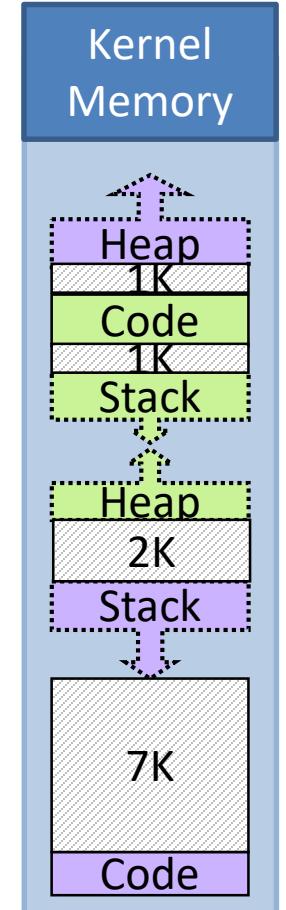
Problem: External Fragmentation

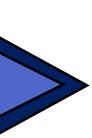
- Memory gets broken into random size, non-contiguous pieces

Let's place segments on the physical memory



Physical Memory

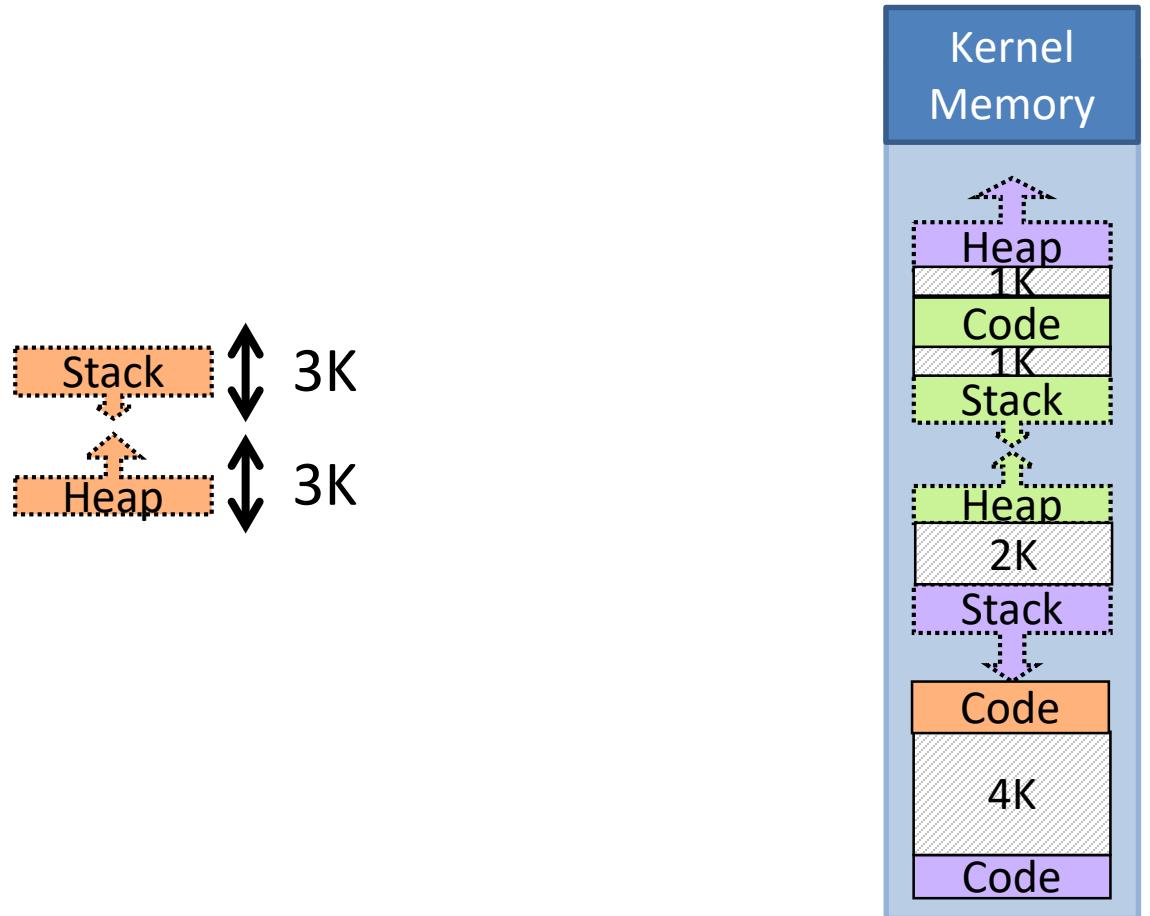


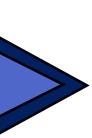


Problem: External Fragmentation

- Memory gets broken into random size, non-contiguous pieces

Physical Memory

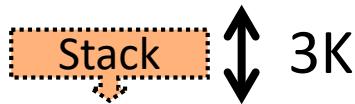




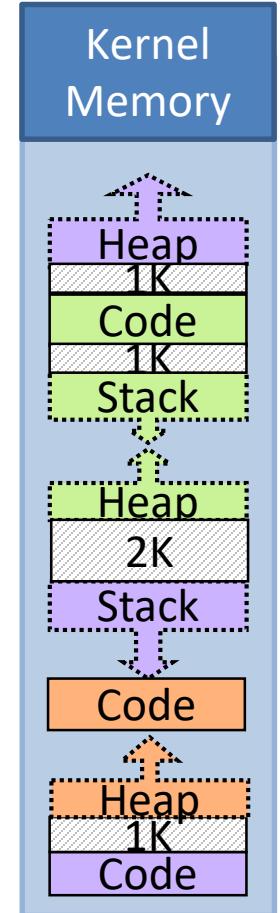
Problem: External Fragmentation

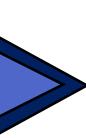
- Memory gets broken into random size, non-contiguous pieces

Physical Memory



Oops... There is 4K free space,
but we cannot place 3K stack
segment on the memory
→ Memory is fragmented ☹



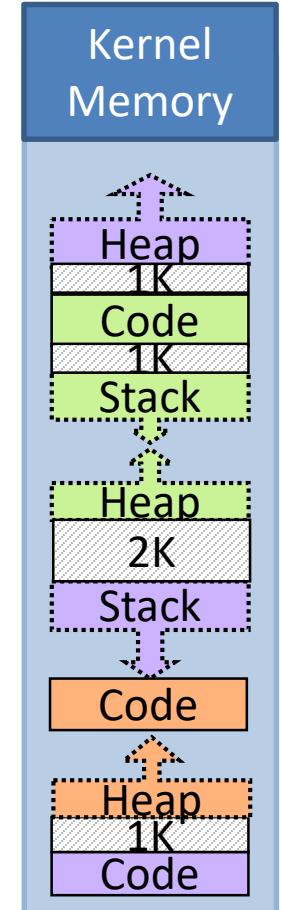


Solution: Compaction

- Compaction can fix the problem
 - But it is extremely expensive (memcpy)



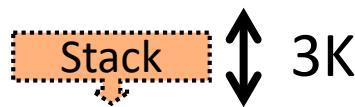
Physical Memory



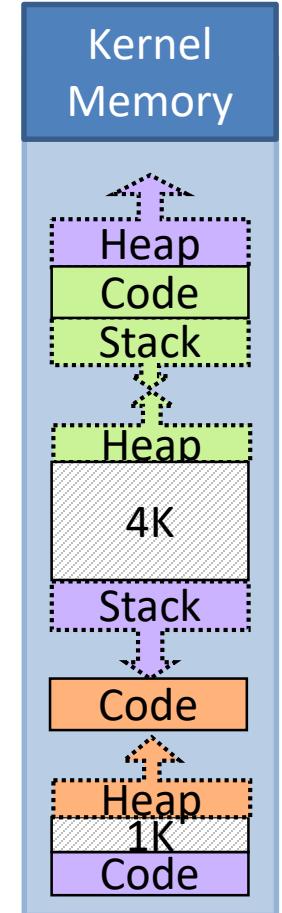


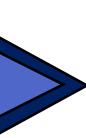
Solution: Compaction

- Compaction can fix the problem
 - But it is extremely expensive (memcpy)



Physical Memory

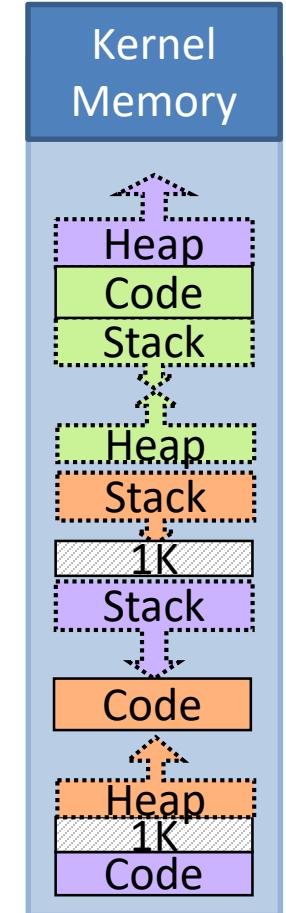




Solution: Compaction

- Compaction can fix the problem
 - But it is extremely expensive (memcpy)

Physical Memory



Segments improve on base and bound, but they still aren't enough
(i.e., external fragmentation)

Paging

Applied first in the
Atlas Computer ('62)

Base and
Bound



Paging

Old, simple, limited
functionality

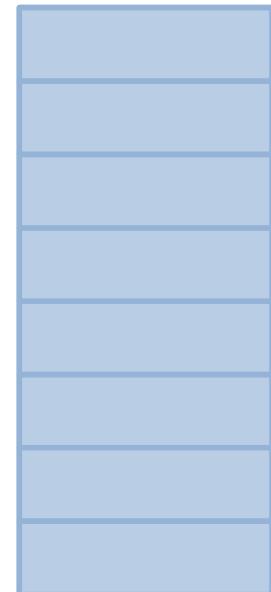
Modern, complex,
lots of functionality

Towards Paged Memory

- The **paged memory model** is a generalization of the segmented memory model

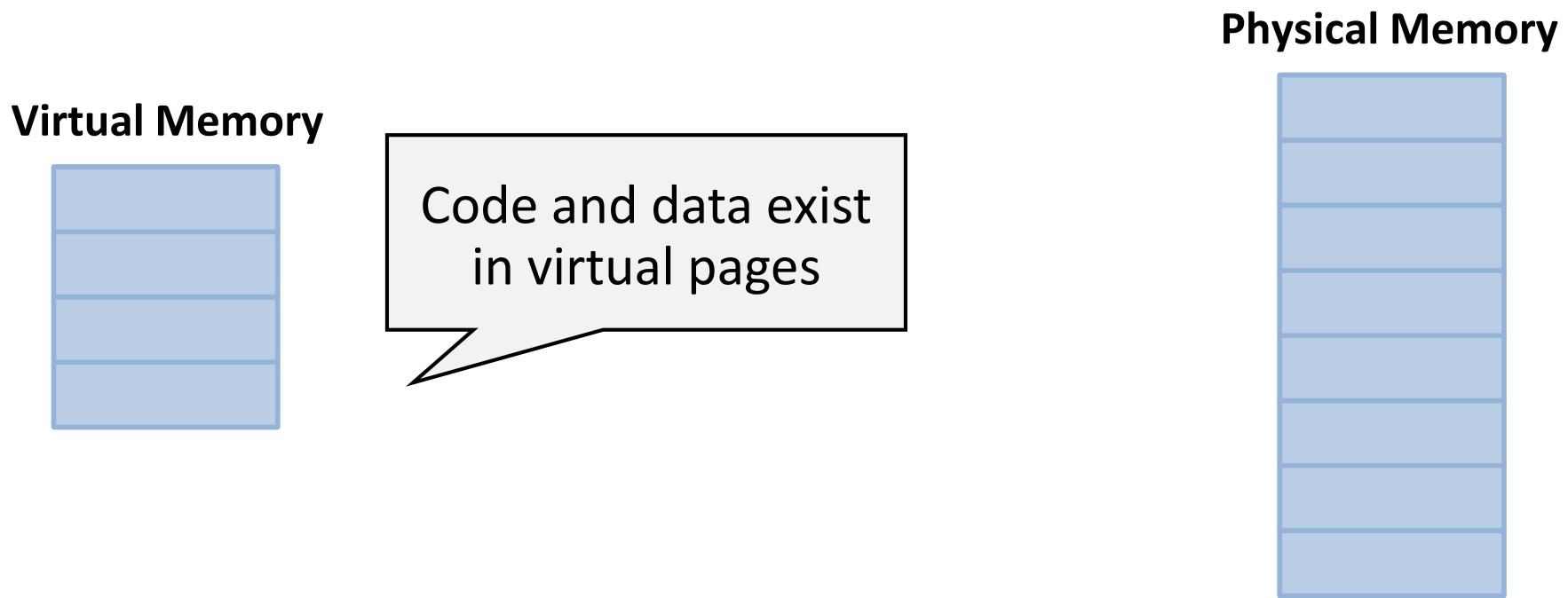
Physical Memory

Physical memory is divided up into same size of physical pages (a.k.a. frames)



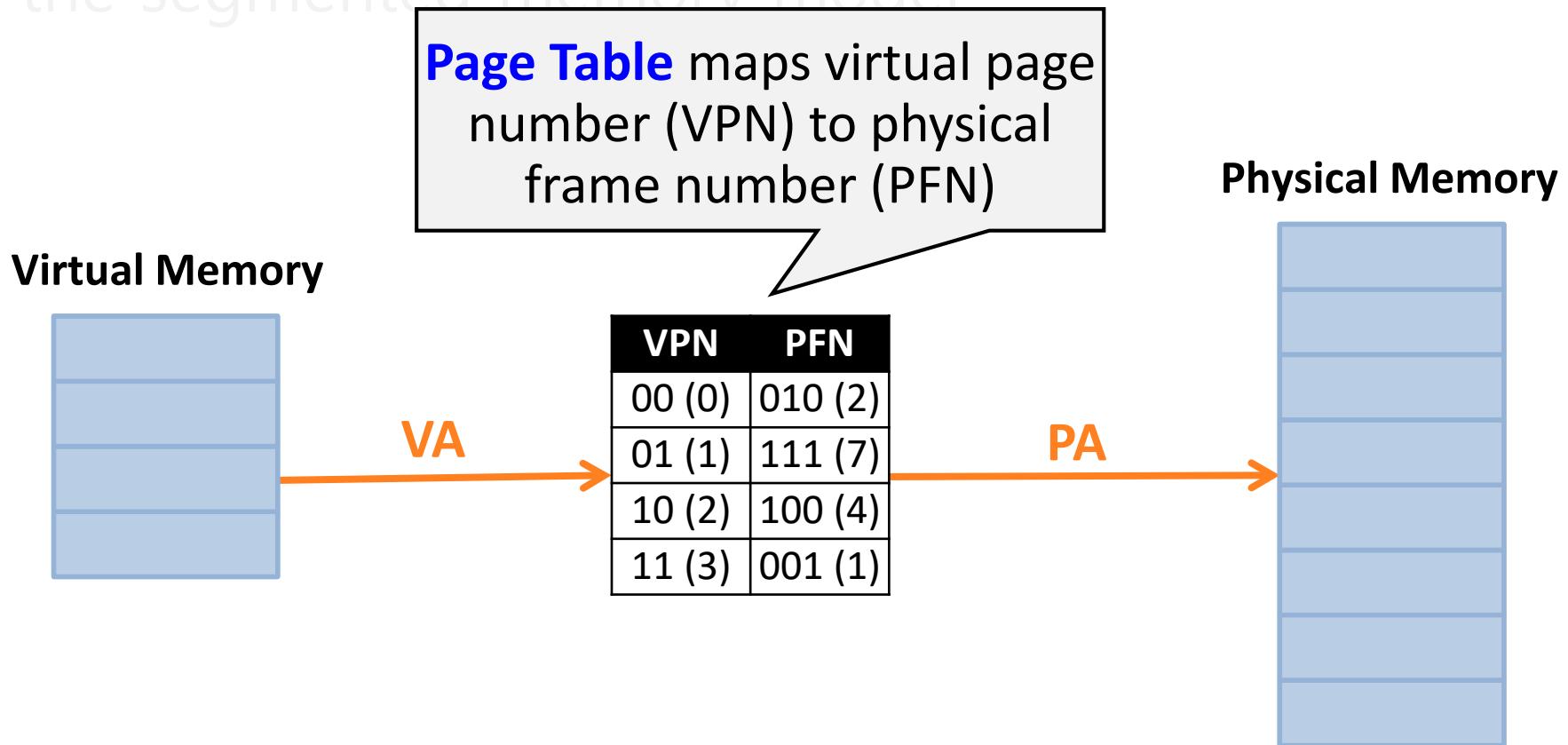
Towards Paged Memory

- The **paged memory model** is a generalization of the segmented memory model



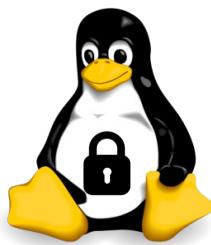
Towards Paged Memory

- The **paged memory model** is a generalization of the segmented memory model

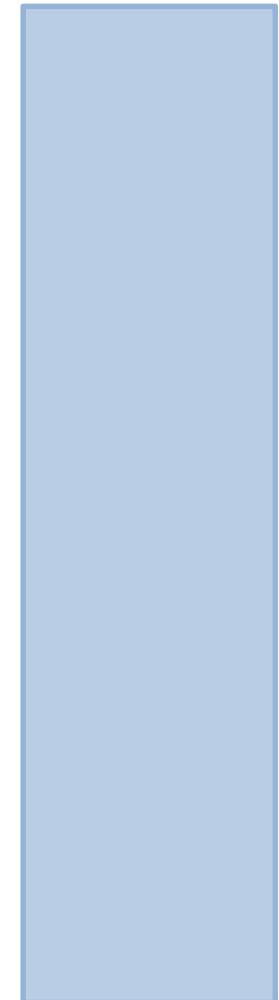


Page Table Implementation

OS creates the page table for each process



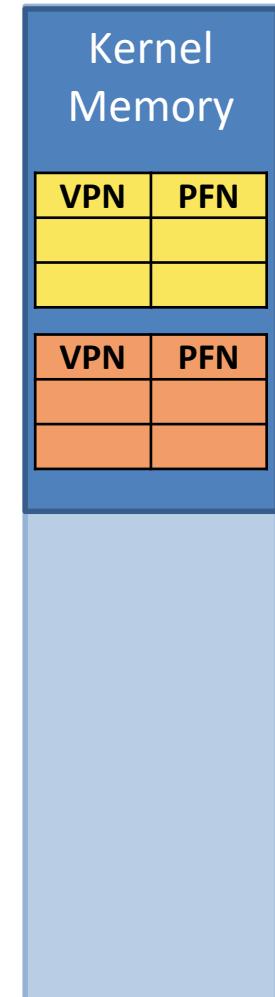
Physical Memory



Page Table Implementation



Physical Memory

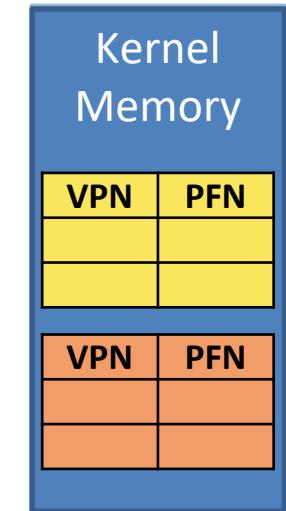


- ① Page tables are typically stored in kernel memory

Page Table Implementation

Physical Memory

Note that, for the sake of brevity, we show the page in this simple way

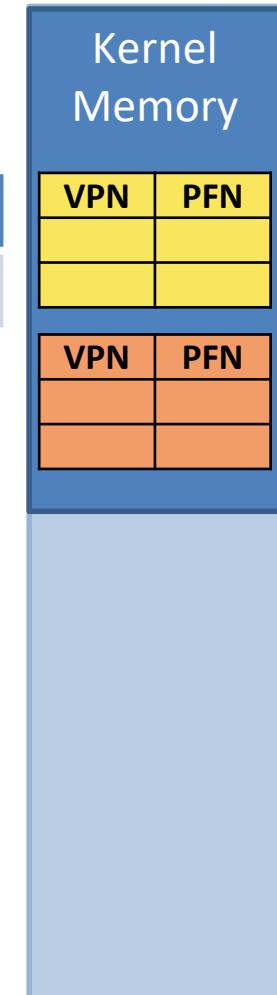


Page Table Implementation

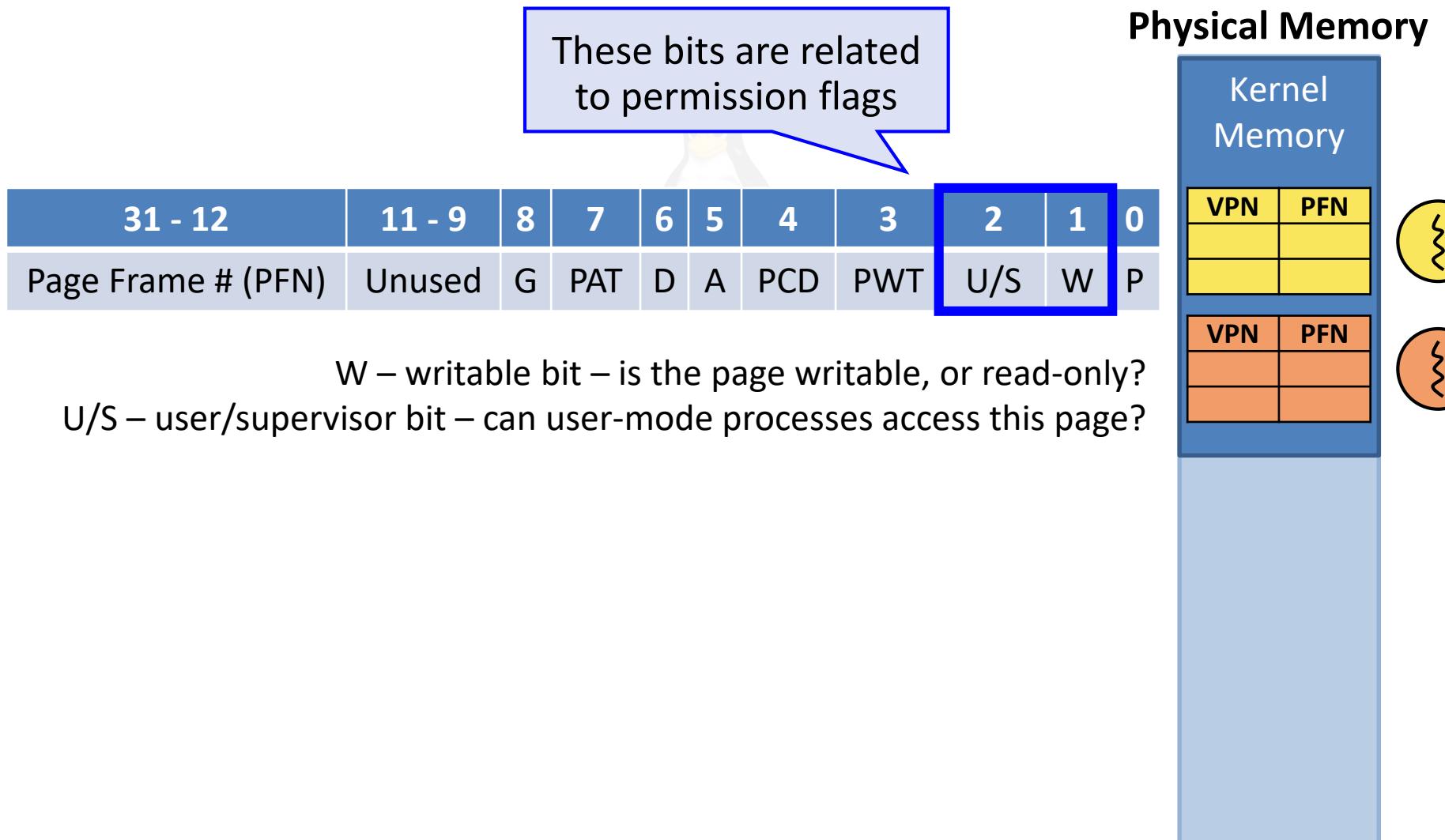
However, page table entries (PTE) practically include more information than just mapping – 4 bytes on x86

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame # (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

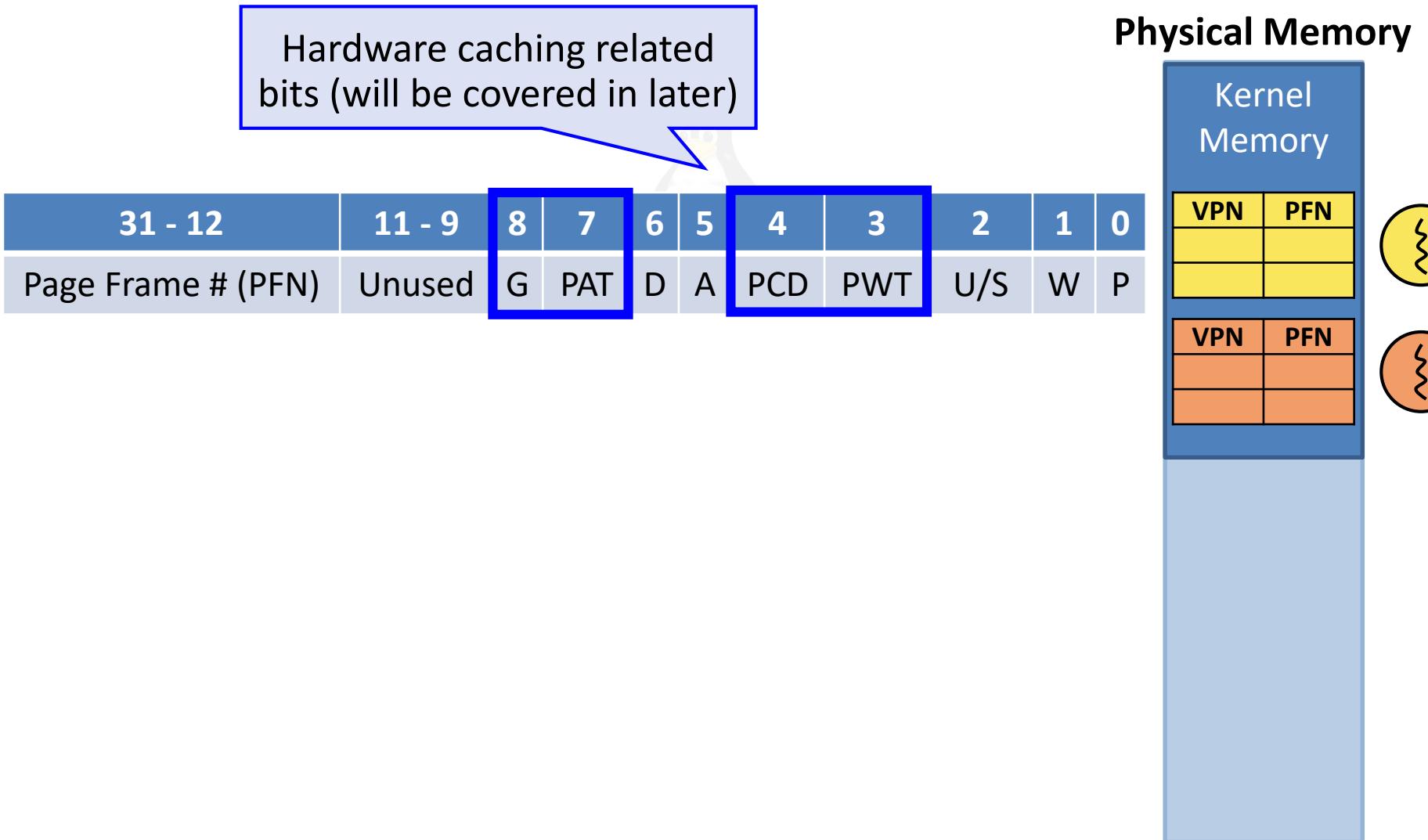
Physical Memory



Page Table Implementation



Page Table Implementation

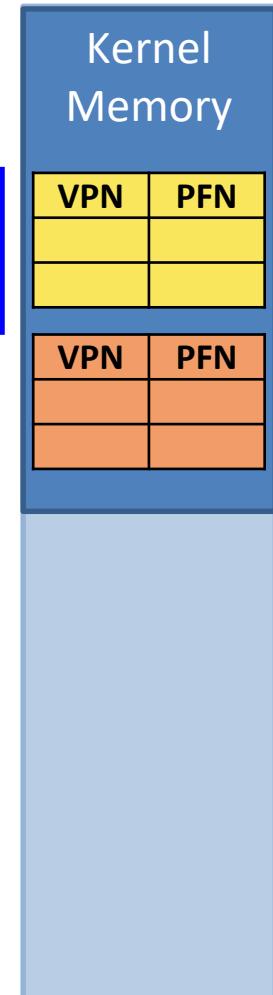


Page Table Implementation

Bits related to swapping
(also will be covered in later)

31 - 12	11 - 9	8	7	6 5	4	3	2	1	0
Page Frame # (PFN)	Unused	G	PAT	D A	PCD	PWT	U/S	W	P

Physical Memory



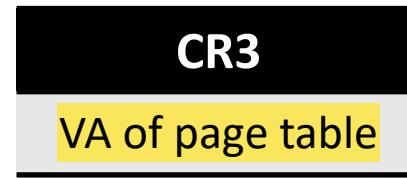
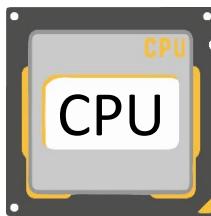
P – present bit – is this page in physical memory?

A – accessed bit – has this page been read recently?

D – dirty bit – has this page been written recently?

Page Table Implementation

- ② OS stores a pointer to the page table in a special register in the CPU (**CR3** register in x86)

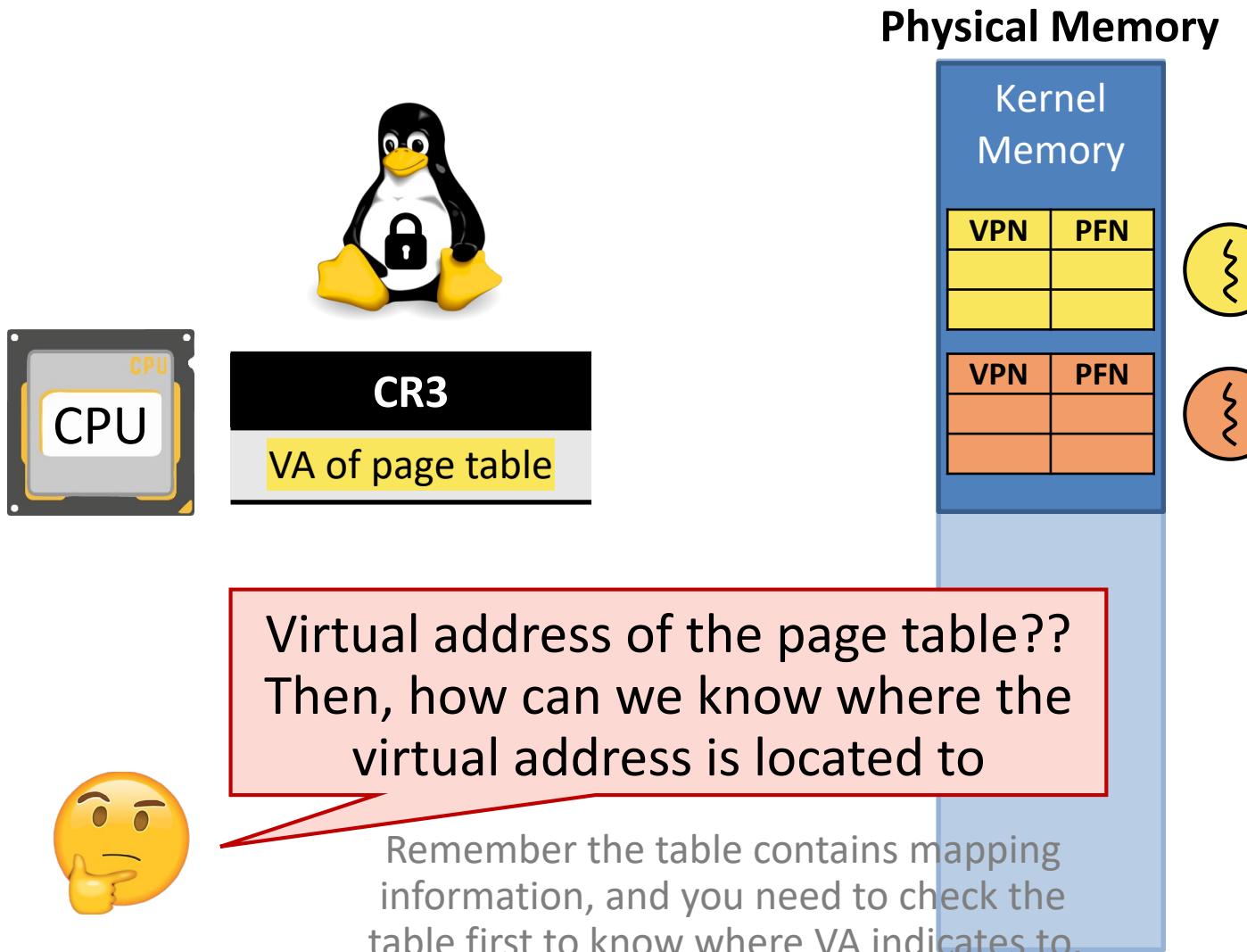


Physical Memory

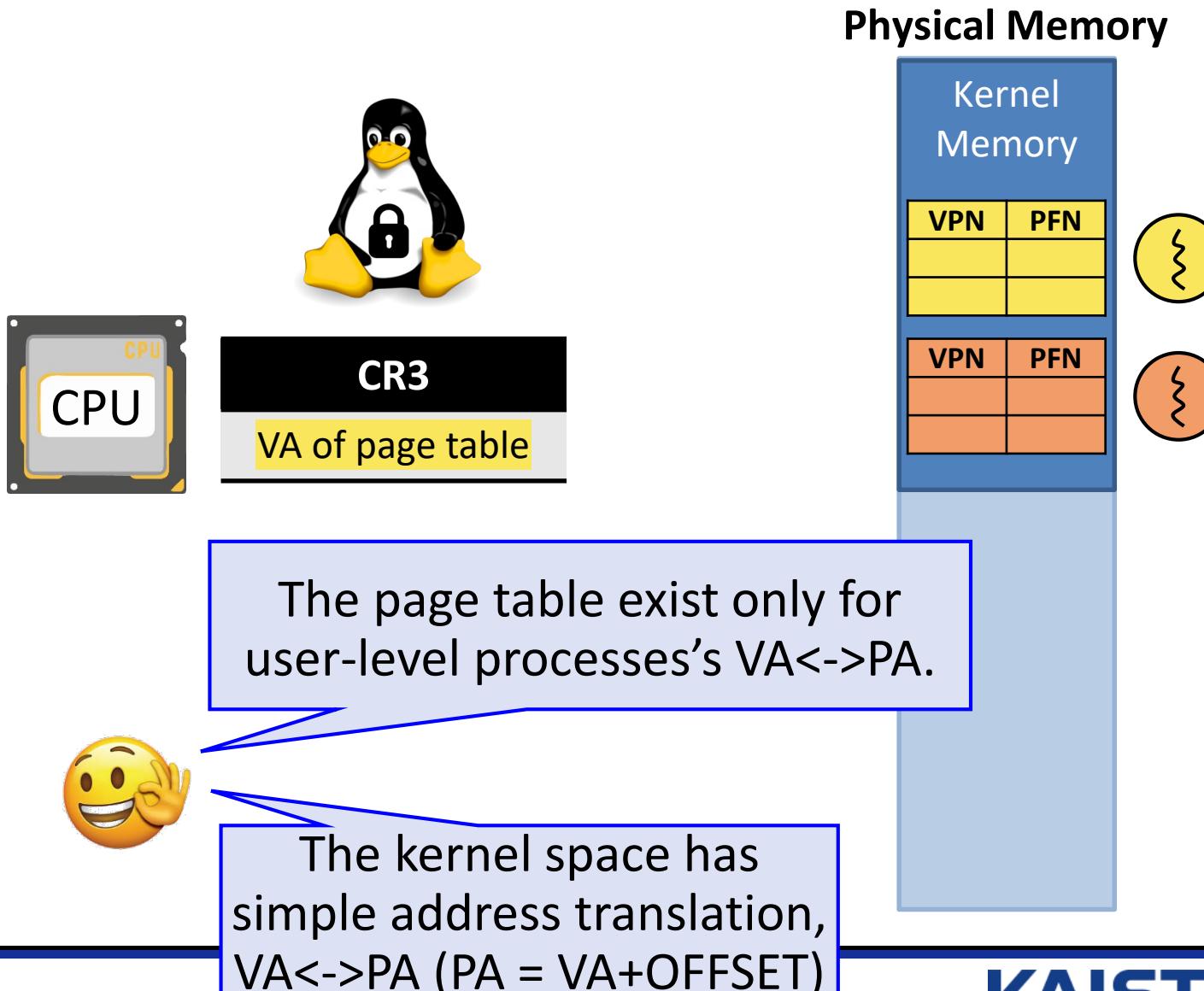
Kernel Memory	
VPN	PFN
VPN	PFN



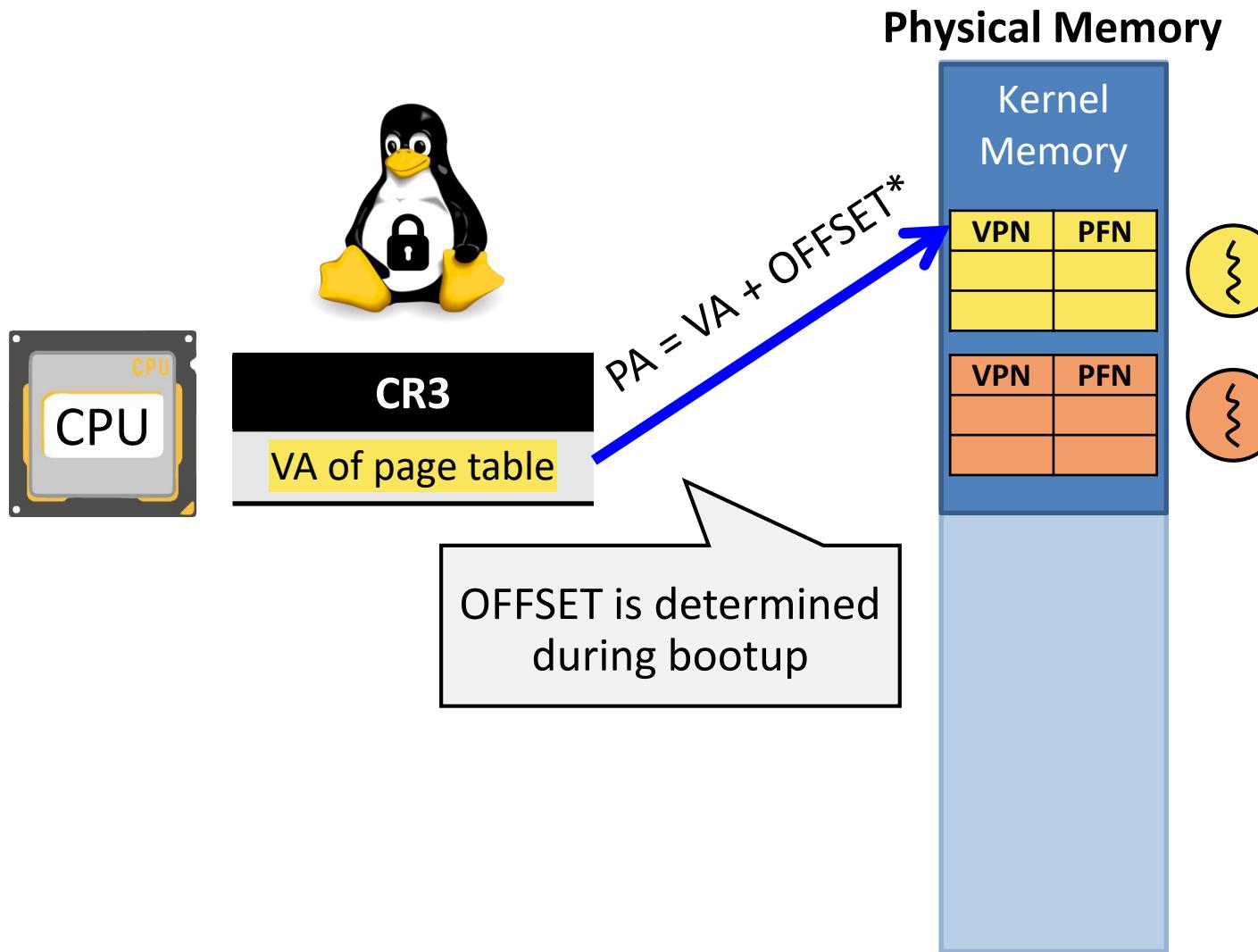
Page Table Implementation



Page Table Implementation

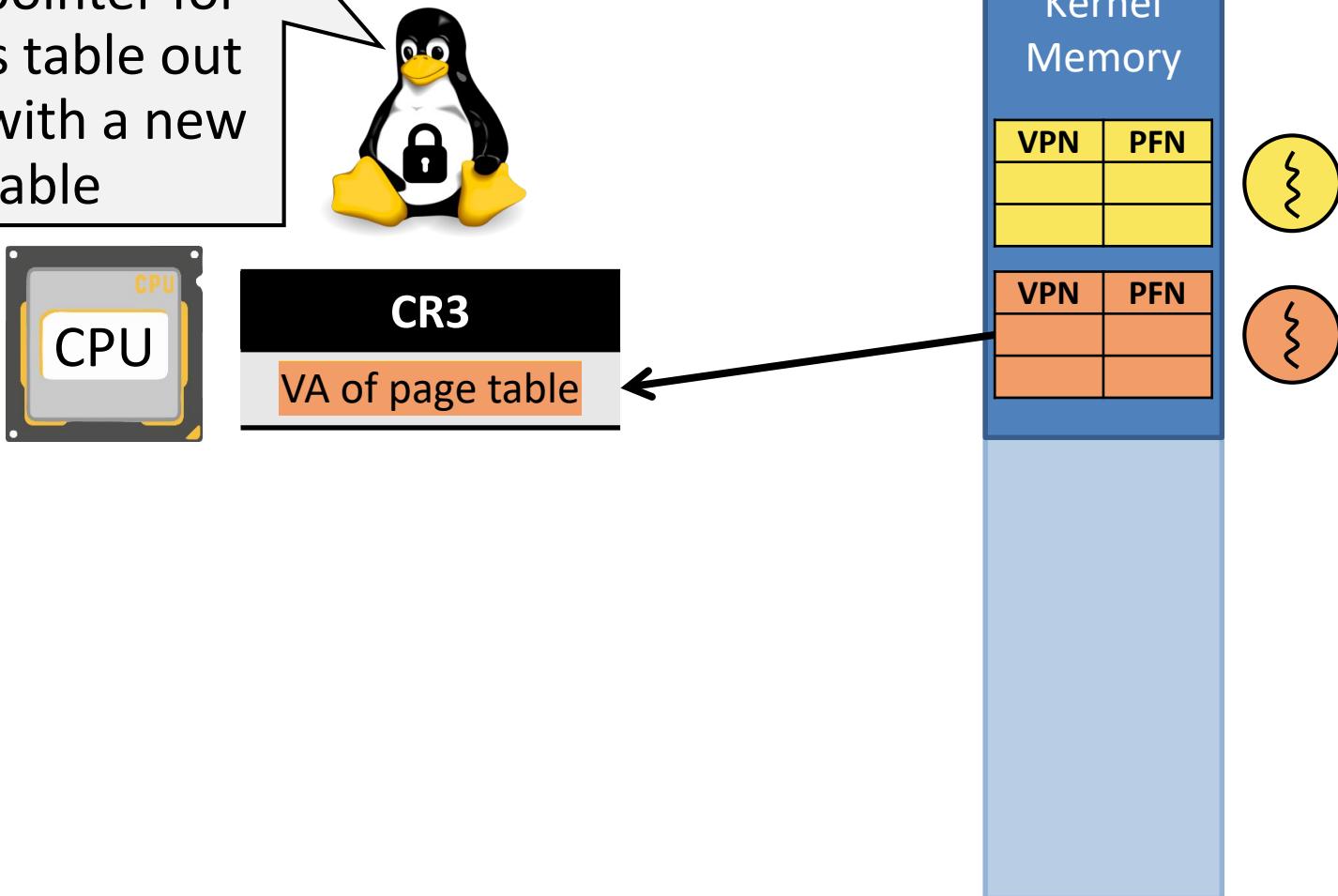


Page Table Implementation

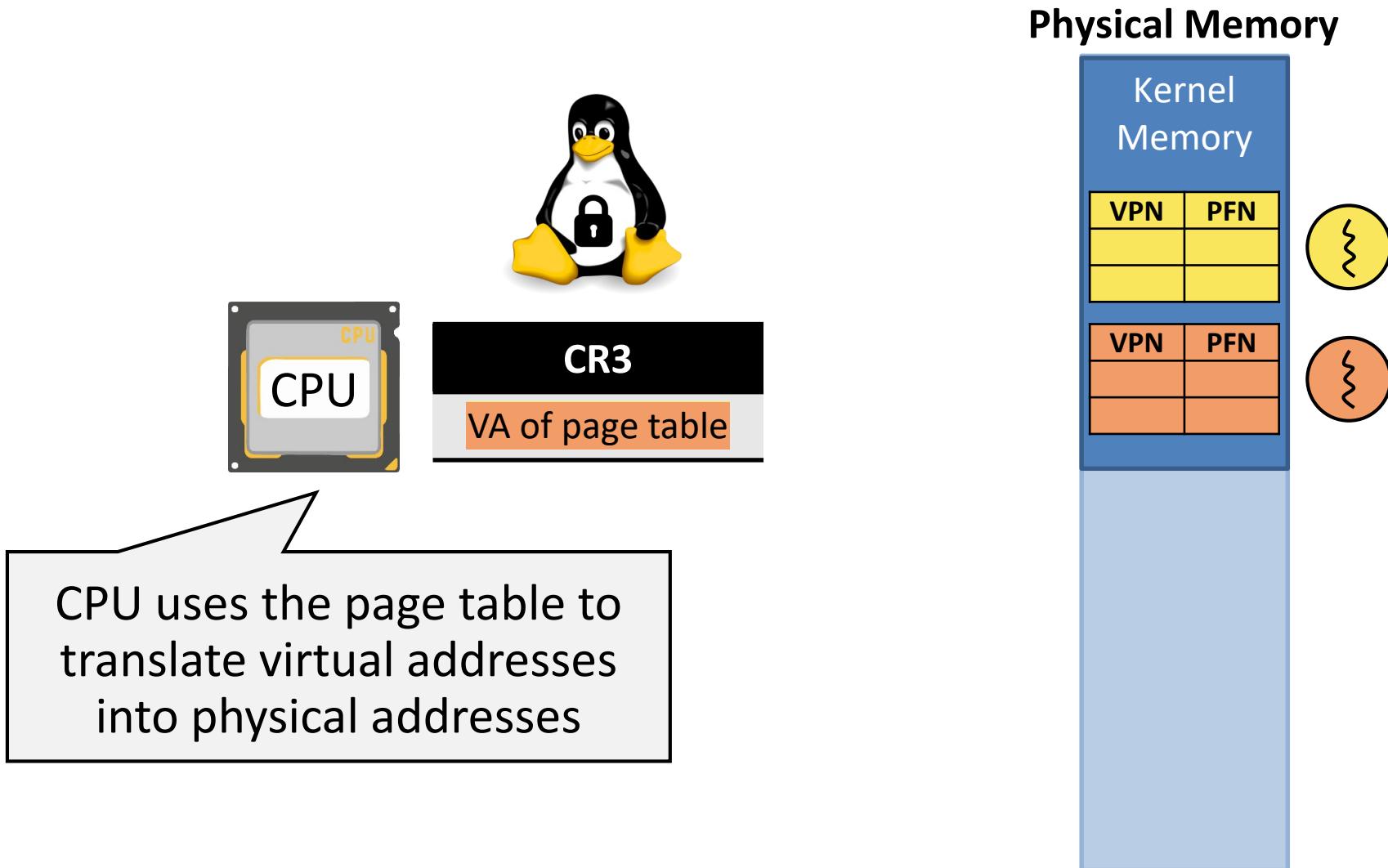


Page Table Implementation

- ③ On a context switch, the OS swaps the pointer for the old process table out and replace it with a new process table

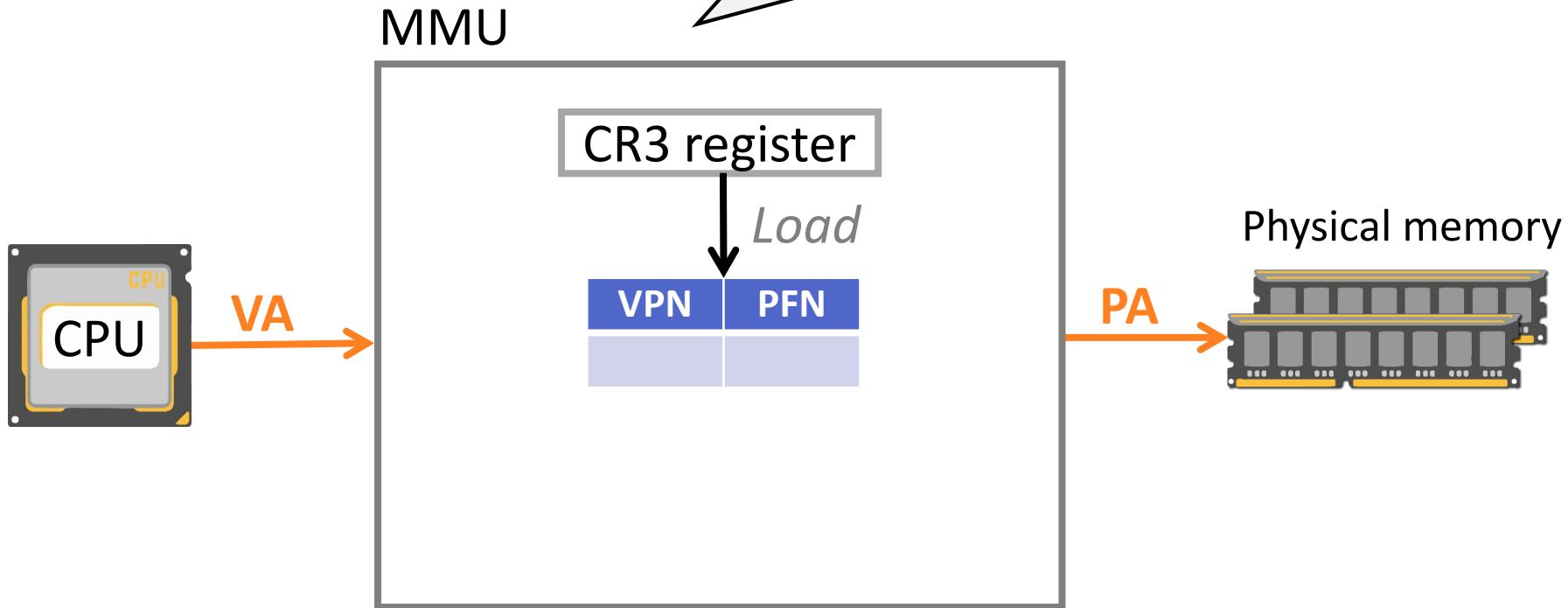


Page Table Implementation

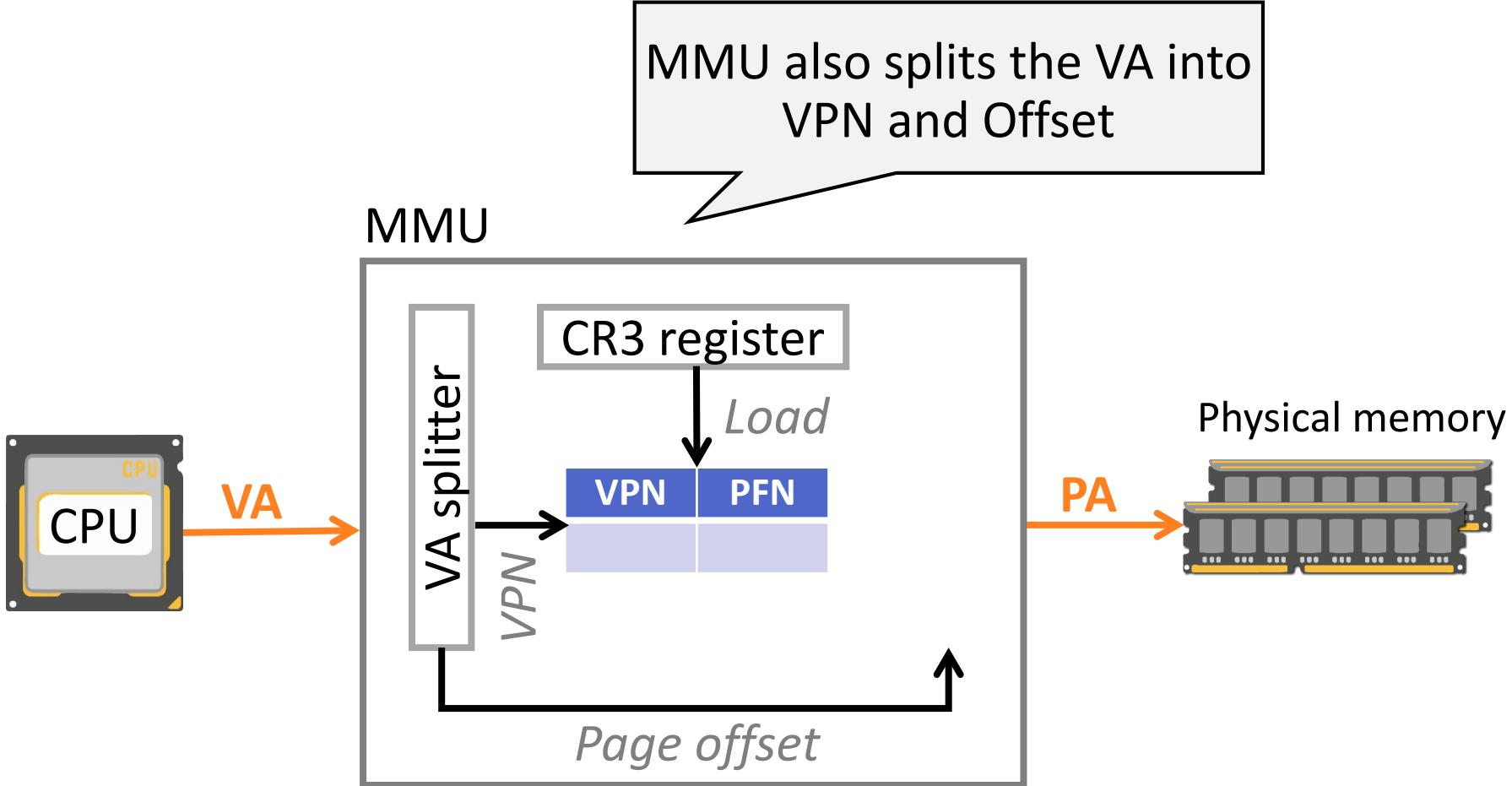


MMU Implementation

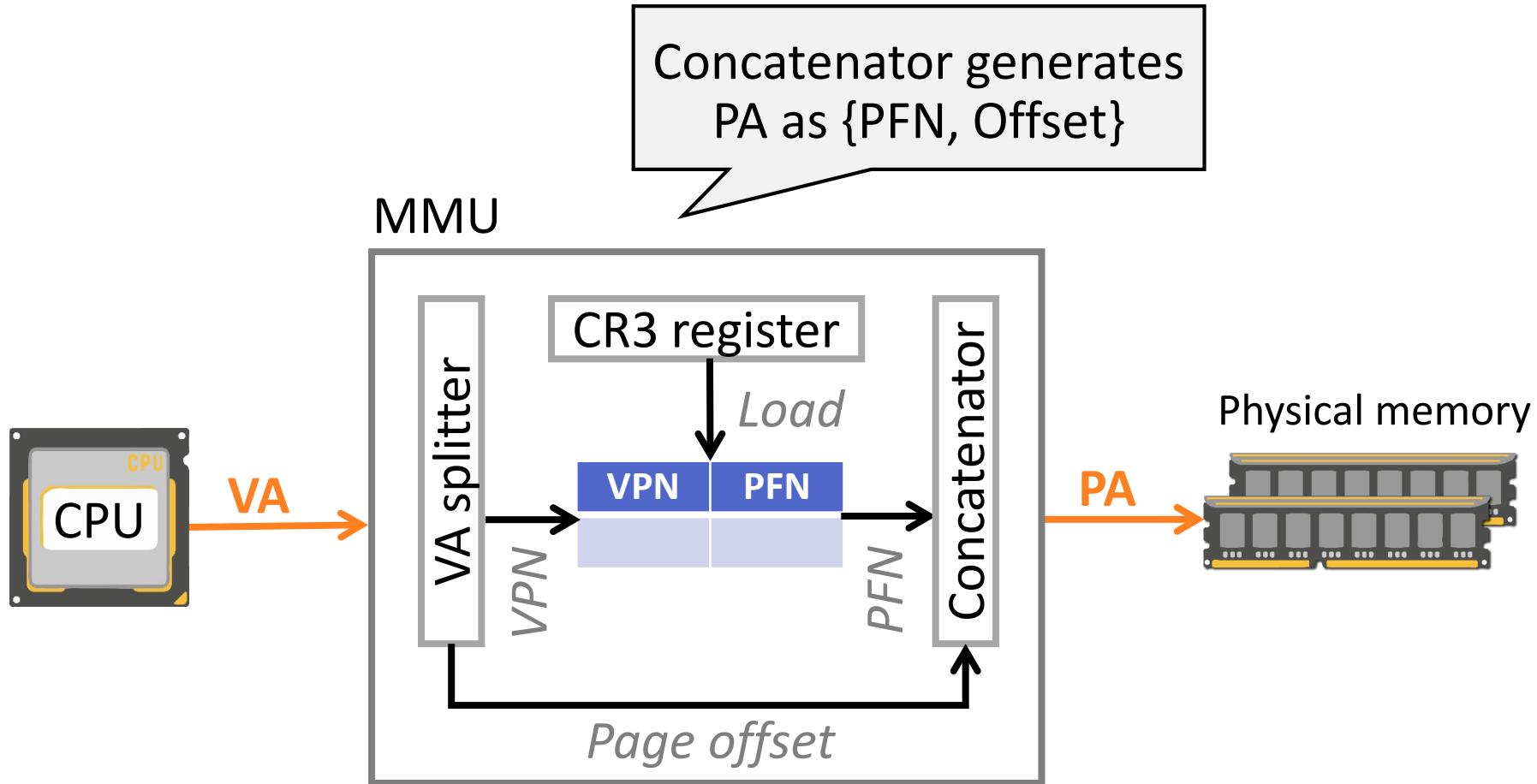
MMU refers CR3 register
to know the VPN <-> PFN
mapping information

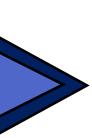


MMU Implementation



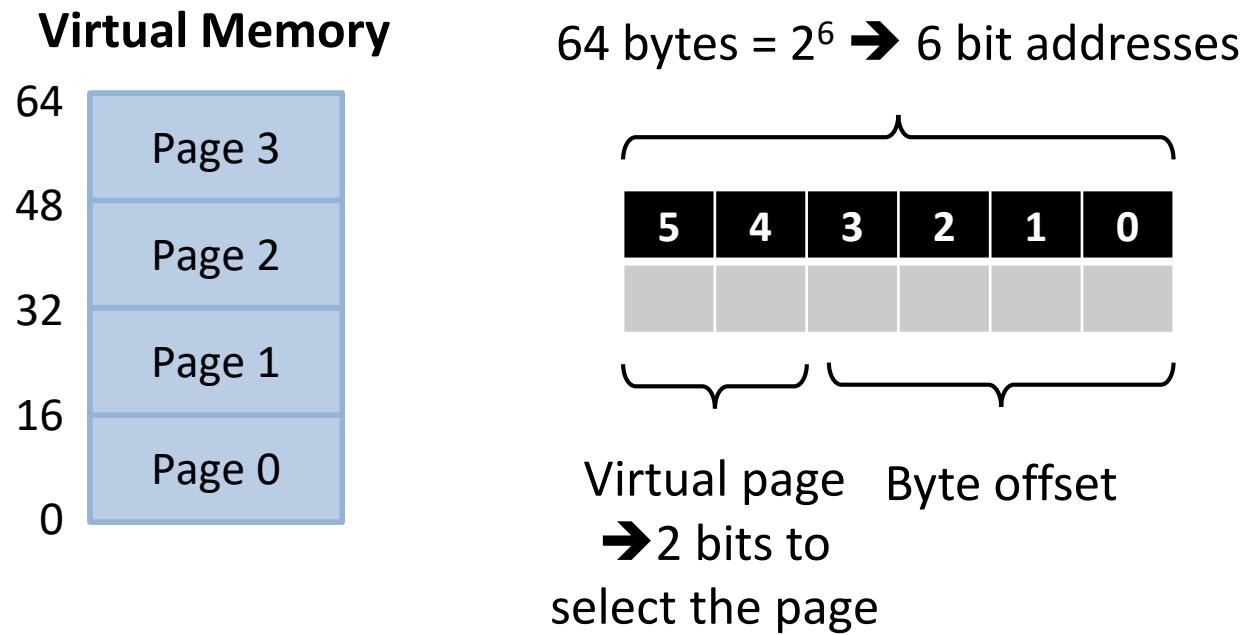
MMU Implementation

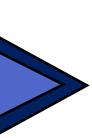




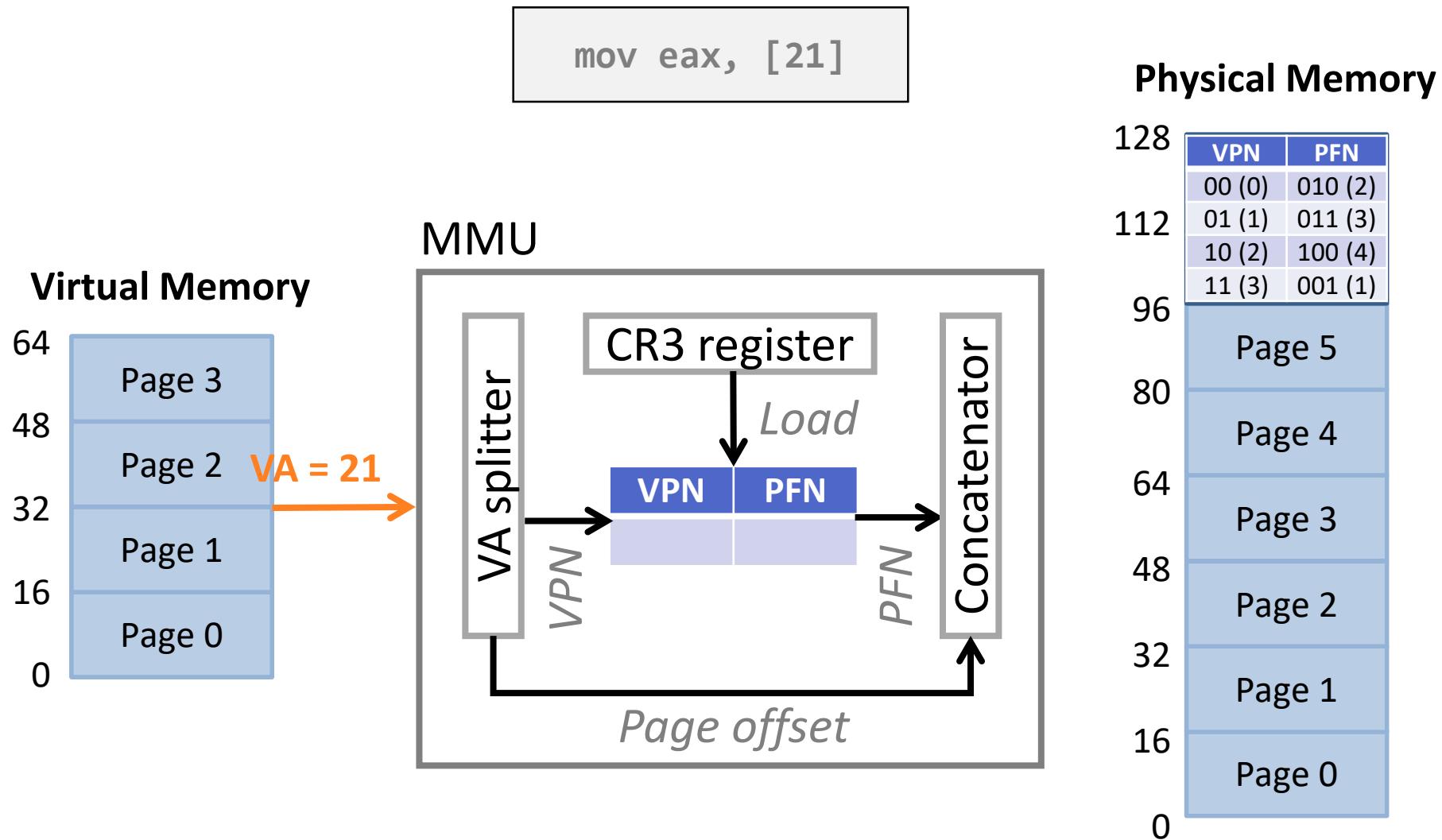
Toy Example

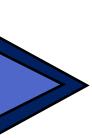
- Suppose we have a 64-byte virtual address space
 - Let's specify 16 bytes per page
- For the sake of brevity, this example assume that the physical memory address space is greater than that of logical memory.



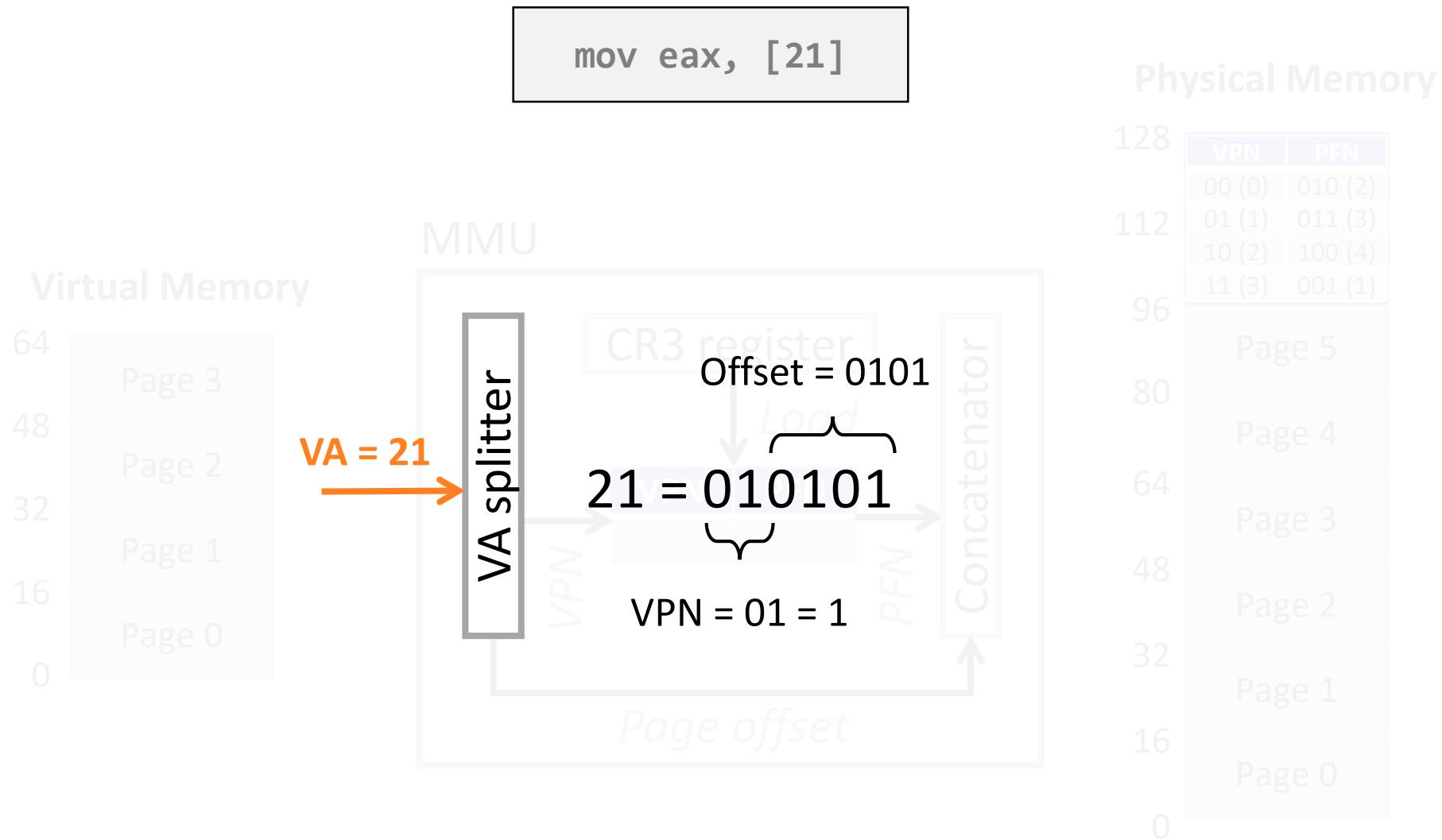


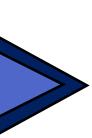
Toy Example



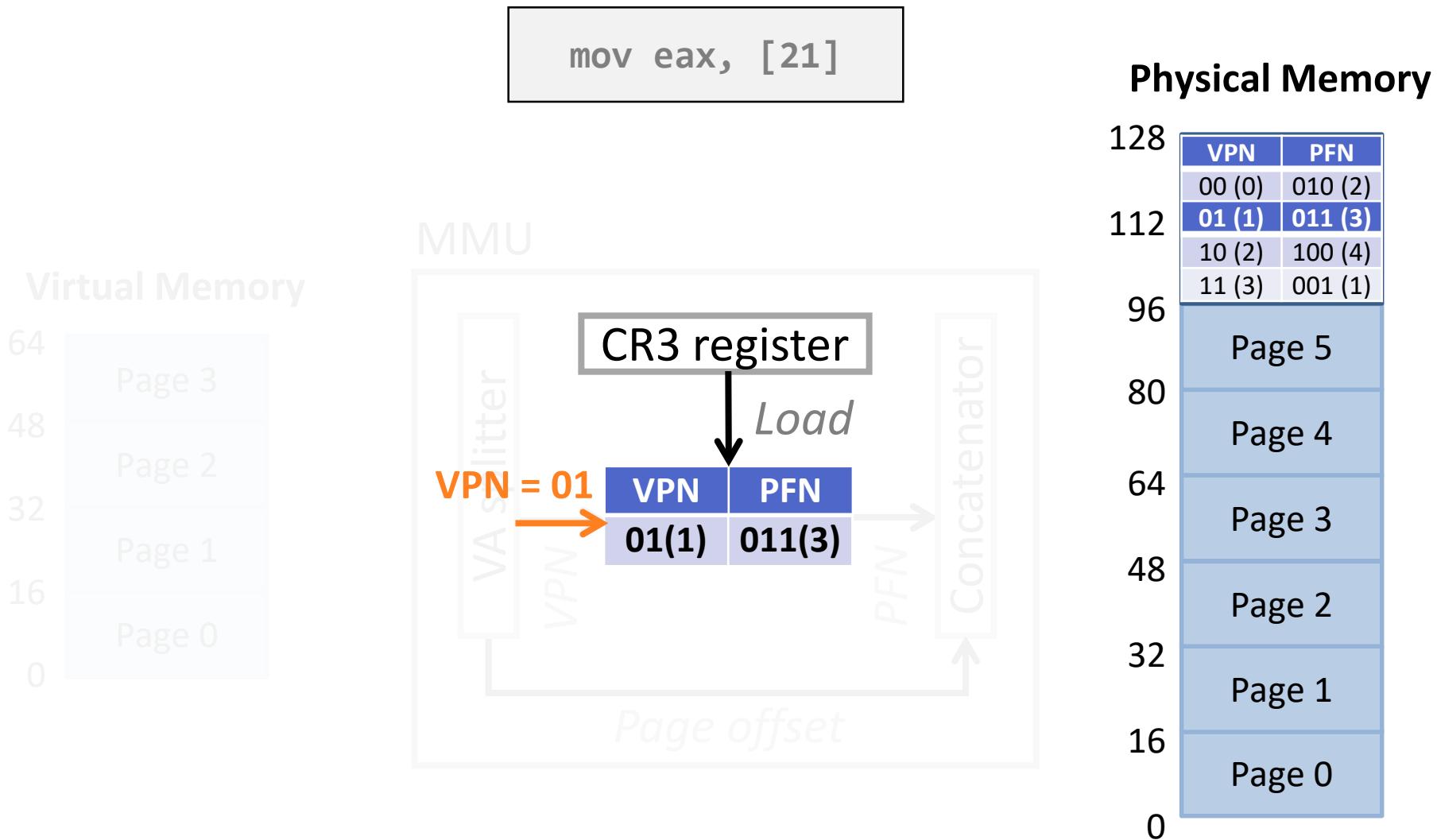


Toy Example



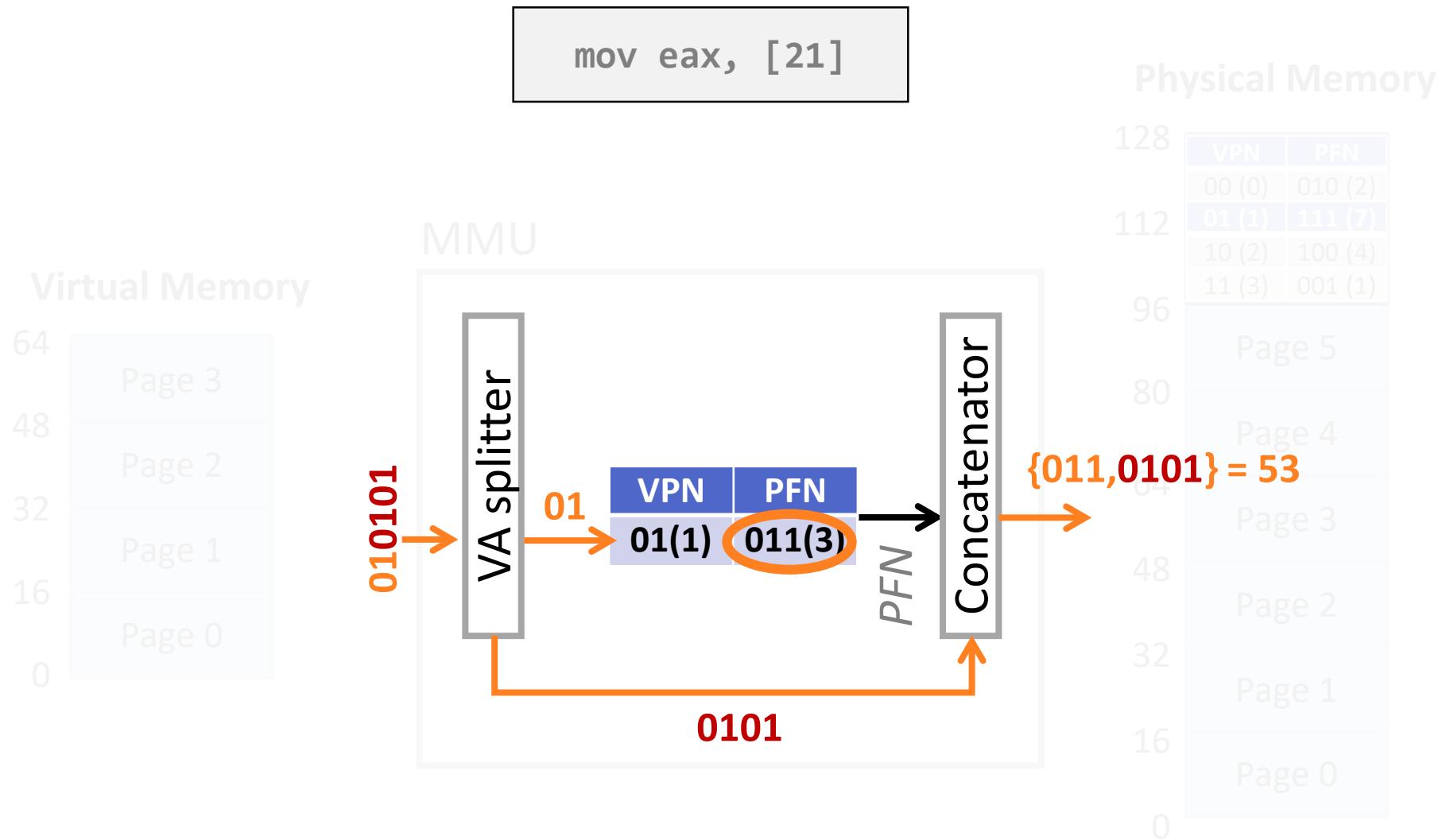


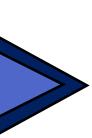
Toy Example



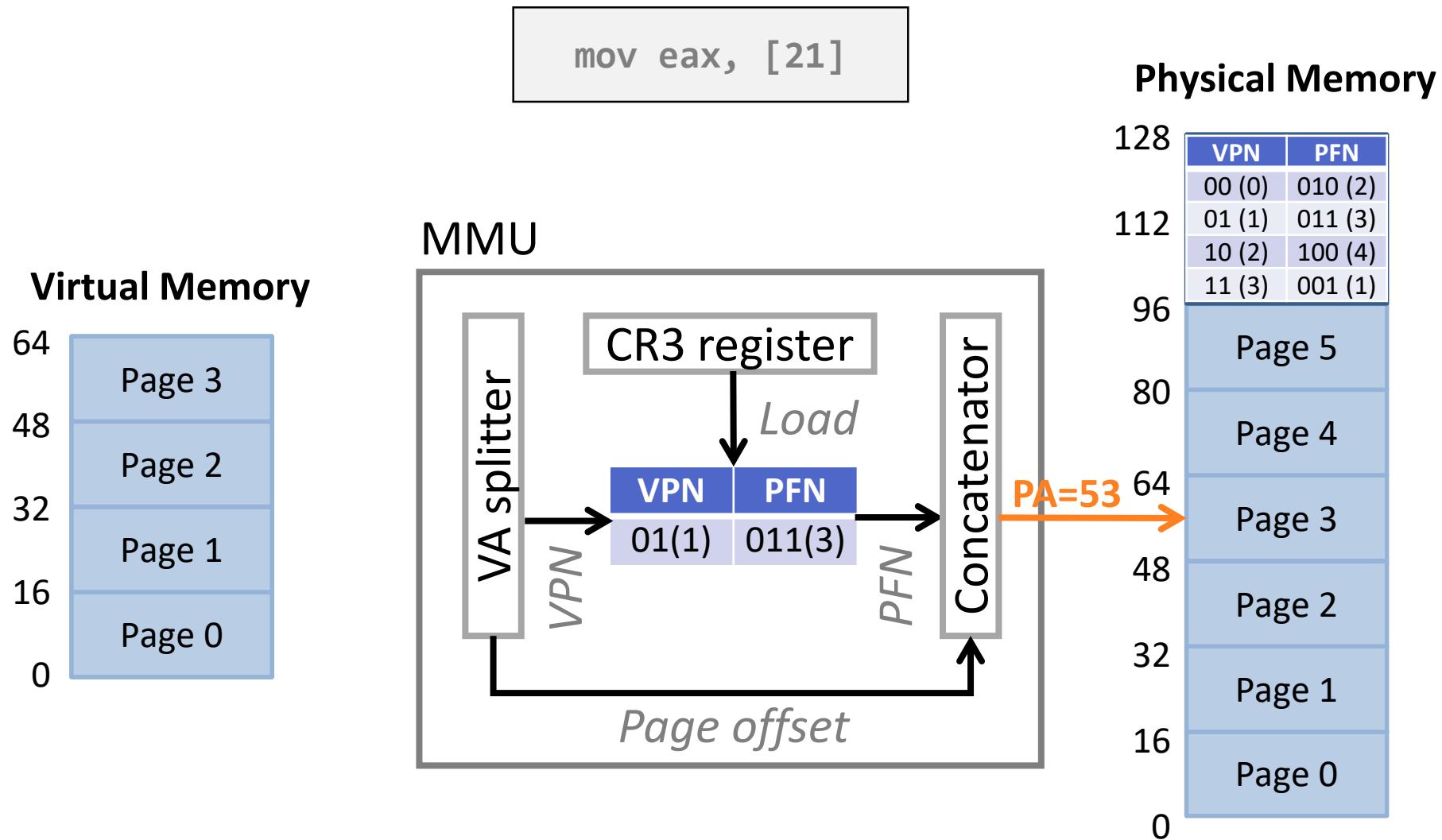


Toy Example



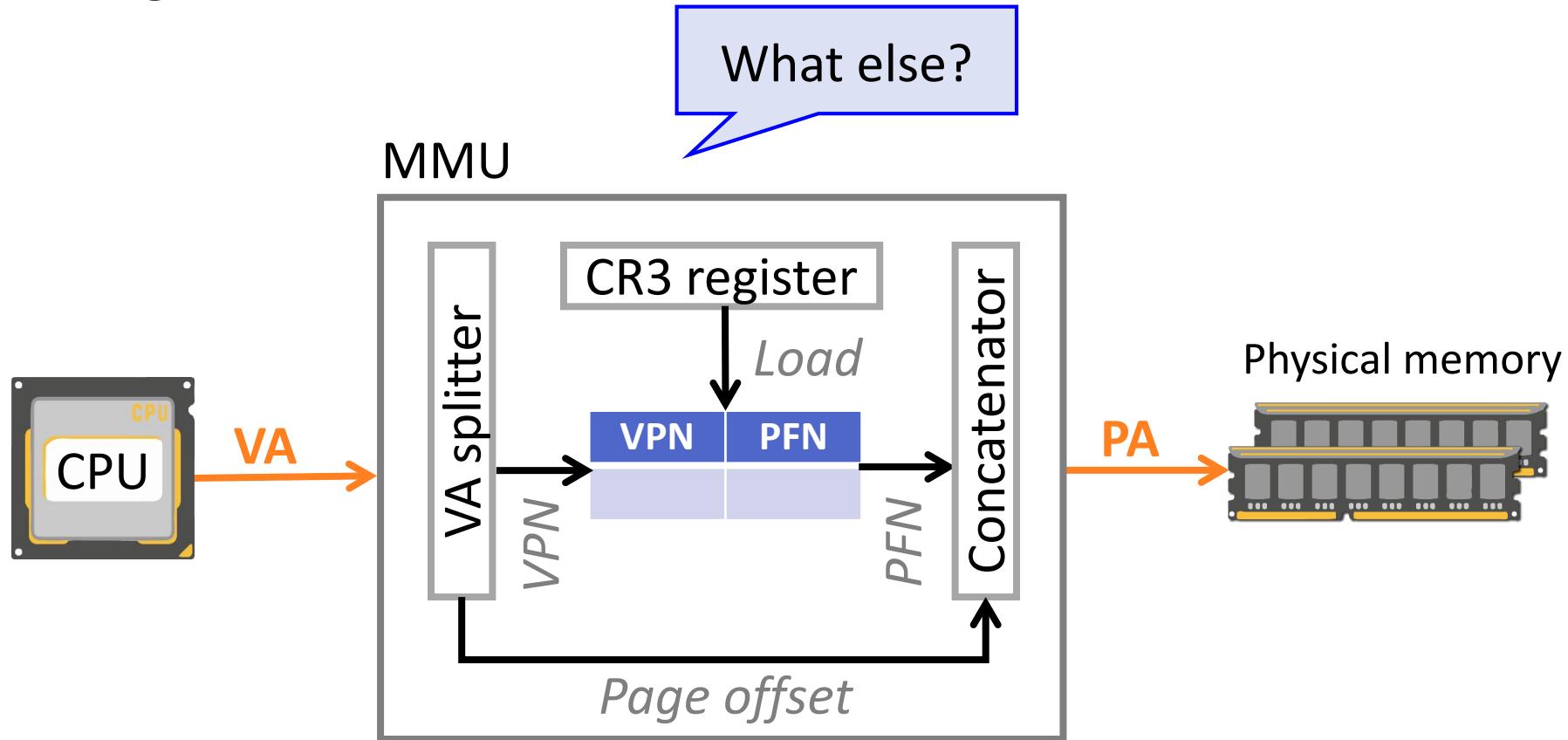


Toy Example



Advantages of Paging

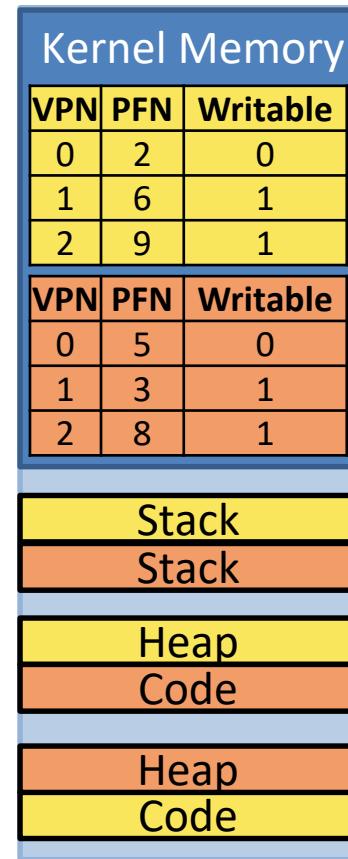
- Paging enjoys all the advantages that segmentation brings



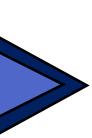
#1: Better for Sparse Address Spaces

- Allocating pages to each process in a fine-grained manner

Physical Memory

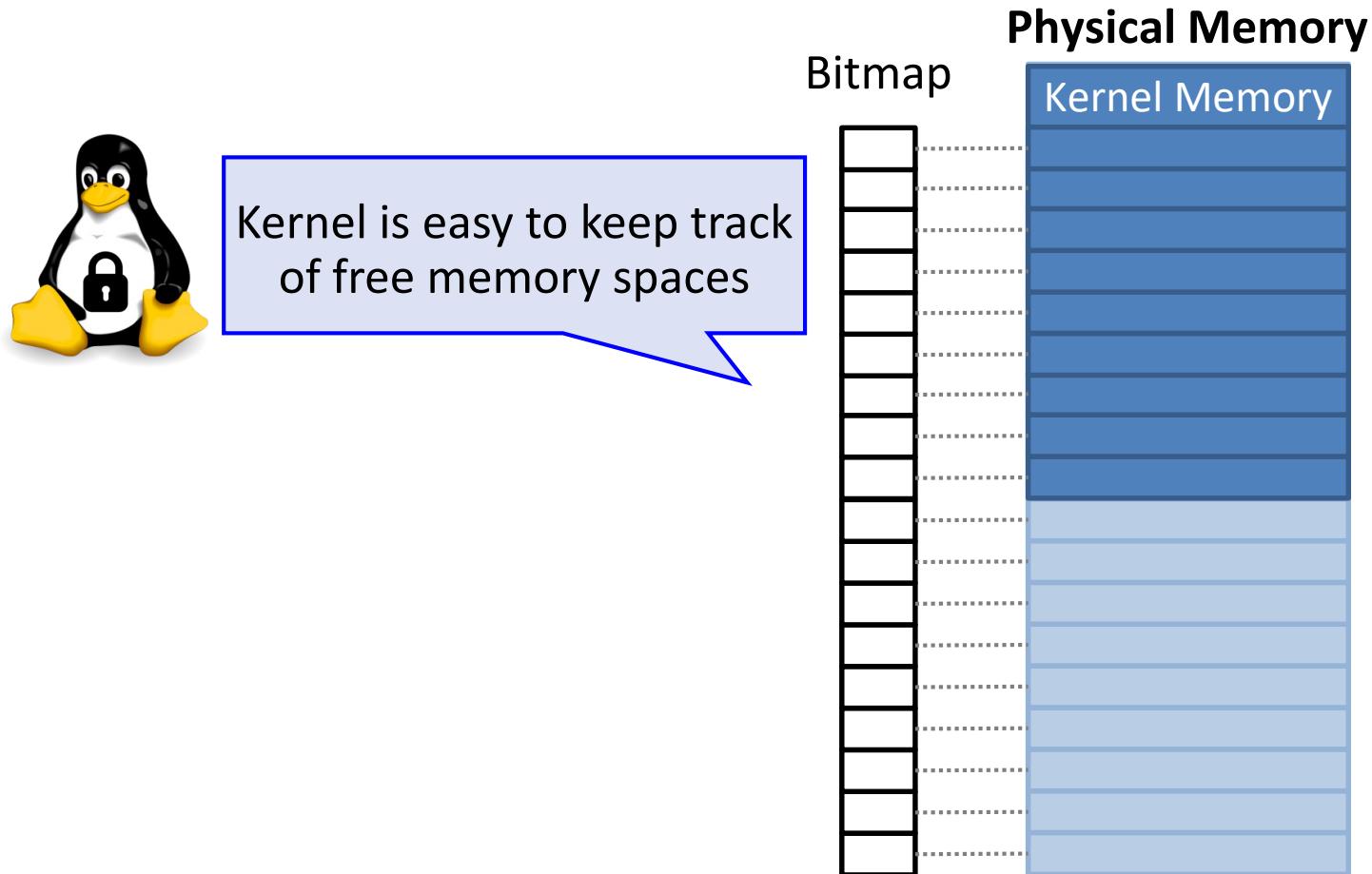


The size of page is relatively small (4KB), paging can address the external fragmentation issue

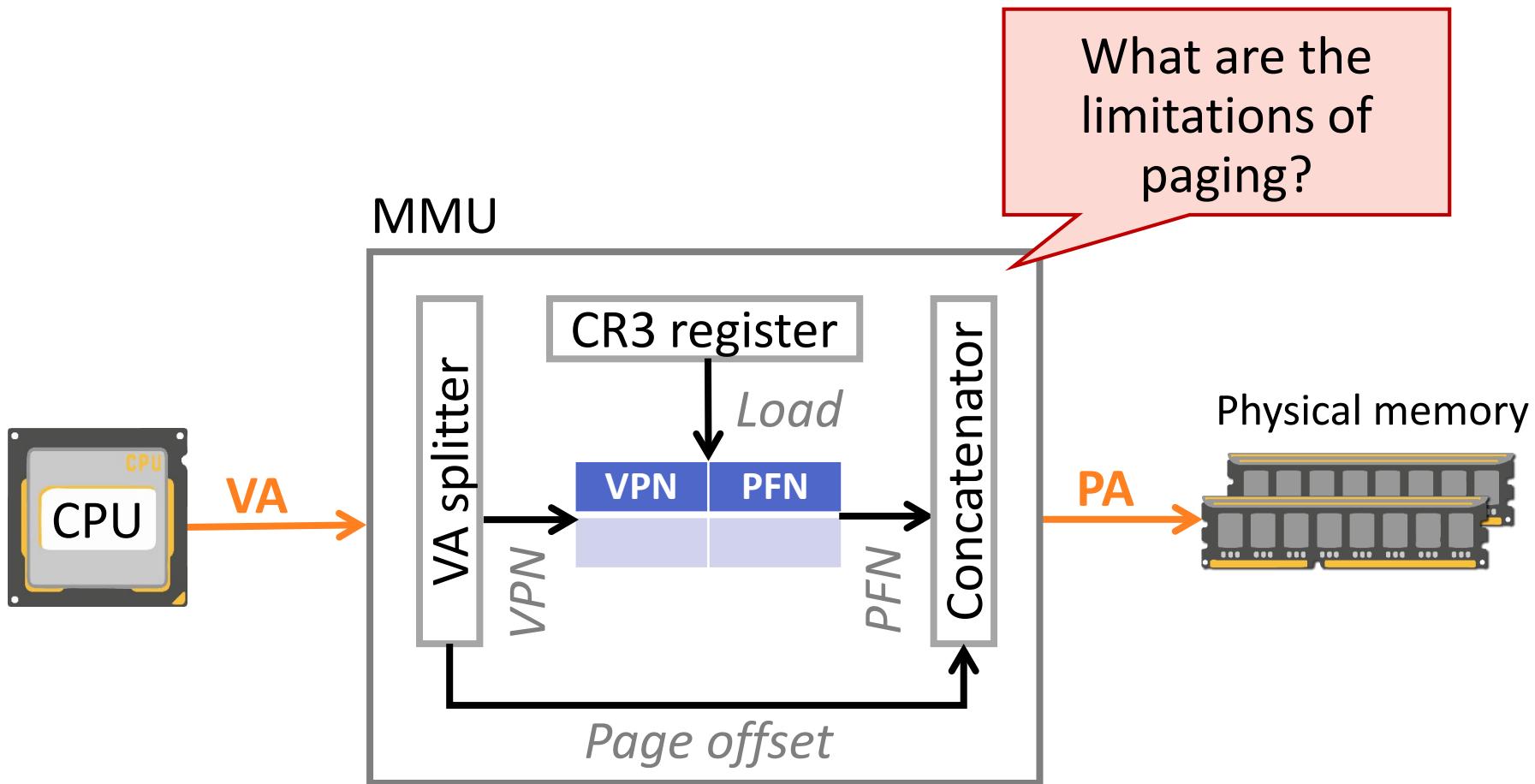


#2: Easy to Manage Physical Memory

- All pages are managed by the same size



Limitations of Paging



#1: Page Tables Are Huge

- Most modern operating systems support a very large logical address space (2^{32} or 2^{64})

- Example: 32-bit system

- Page size = 4KB (2^{12})
- Each page table entry size = 4B

The many of entries
can be
empty/invalid!

Q) What is the size of page table?

A) # of entries in page table = $2^{32}/2^{12} = 2^{20}$
 $\therefore 2^{20} \times 2^2 = 4\text{MB}$

Q) How about the 64-bit system?

A) # of entries in page table = $2^{64}/2^{12} = 2^{52}$
 $\therefore 2^{52} \times 2^2 = 16\text{PB}$

#2: Page Tables Are Slow

```
0x1024 mov [edi + eax * 4], 0x0
0x1028 inc eax
0x102C cmp eax, 0x03E8
0x1030 jne 0x1024
```

Q) How many memory accesses occur during each iteration of the loop?

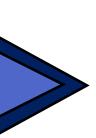
A) - 4 instructions are read from memory

#2: Page Tables Are Slow

```
0x1024 mov [edi + eax * 4], 0x0
0x1028 inc eax
0x102C cmp eax, 0x03E8
0x1030 jne 0x1024
```

Q) How many memory accesses occur during each iteration of the loop?

- A) - 4 instructions are read from memory
- $[edi + eax * 4]$ writes to one location in memory



#2: Page Tables Are Slow

```
0x1024 mov [edi + eax * 4], 0x0  
0x1028 inc eax  
0x102C cmp eax, 0x03E8  
0x1030 jne 0x1024
```

Q) How many memory accesses occur during each iteration of the loop?

- A)
- 4 instructions are read from memory
 - $[edi + eax * 4]$ writes to one location in memory
 - 5 page table lookups

Page tables themselves are in memory and each memory access must be translated

Advanced Functionality

- #1: Shared memory
- #2: Copy-on-Write
- #3: Mapped files



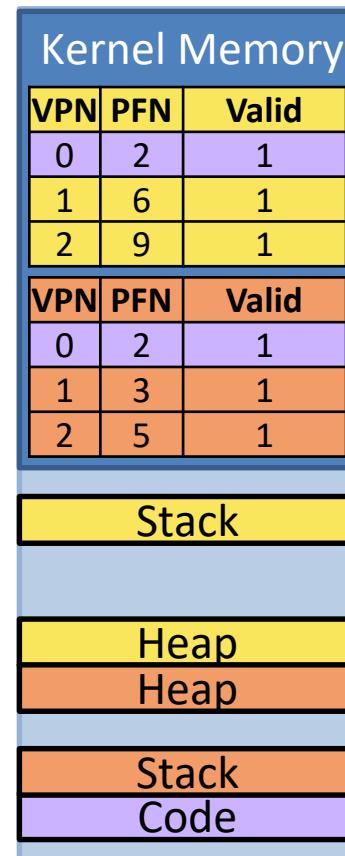
Now we're going to look at some advanced functionalities that the OS can provide applications using paging virtual memory **tricks**

Functionality #1: Shared Memory

Sharing

- We can use shared memory to allow processes to share data using direct memory references

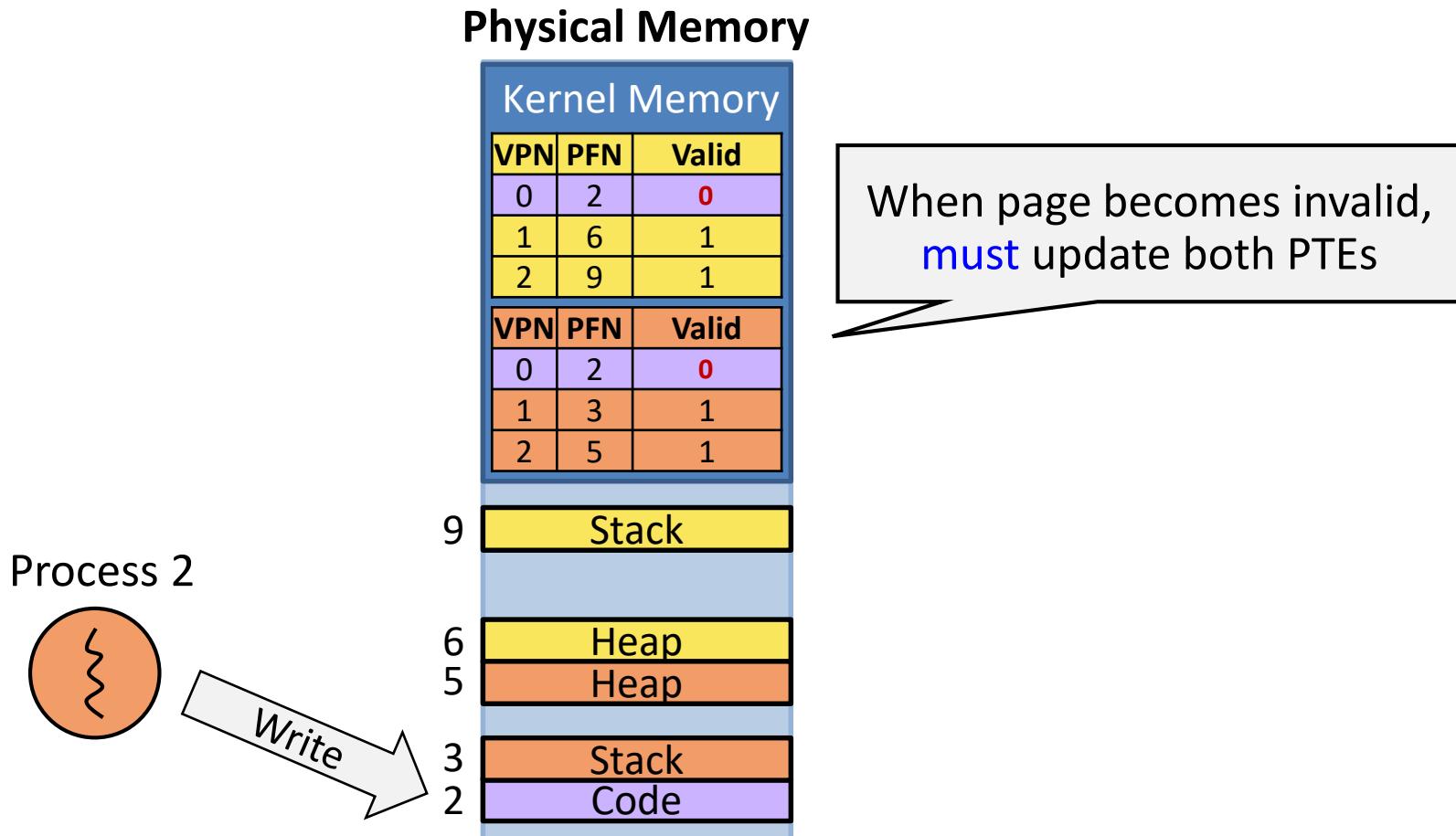
Physical Memory



In the paging system,
processes can share memory
by setting the same physical
frame

Sharing

- We can use shared memory to allow processes to share data using direct memory references

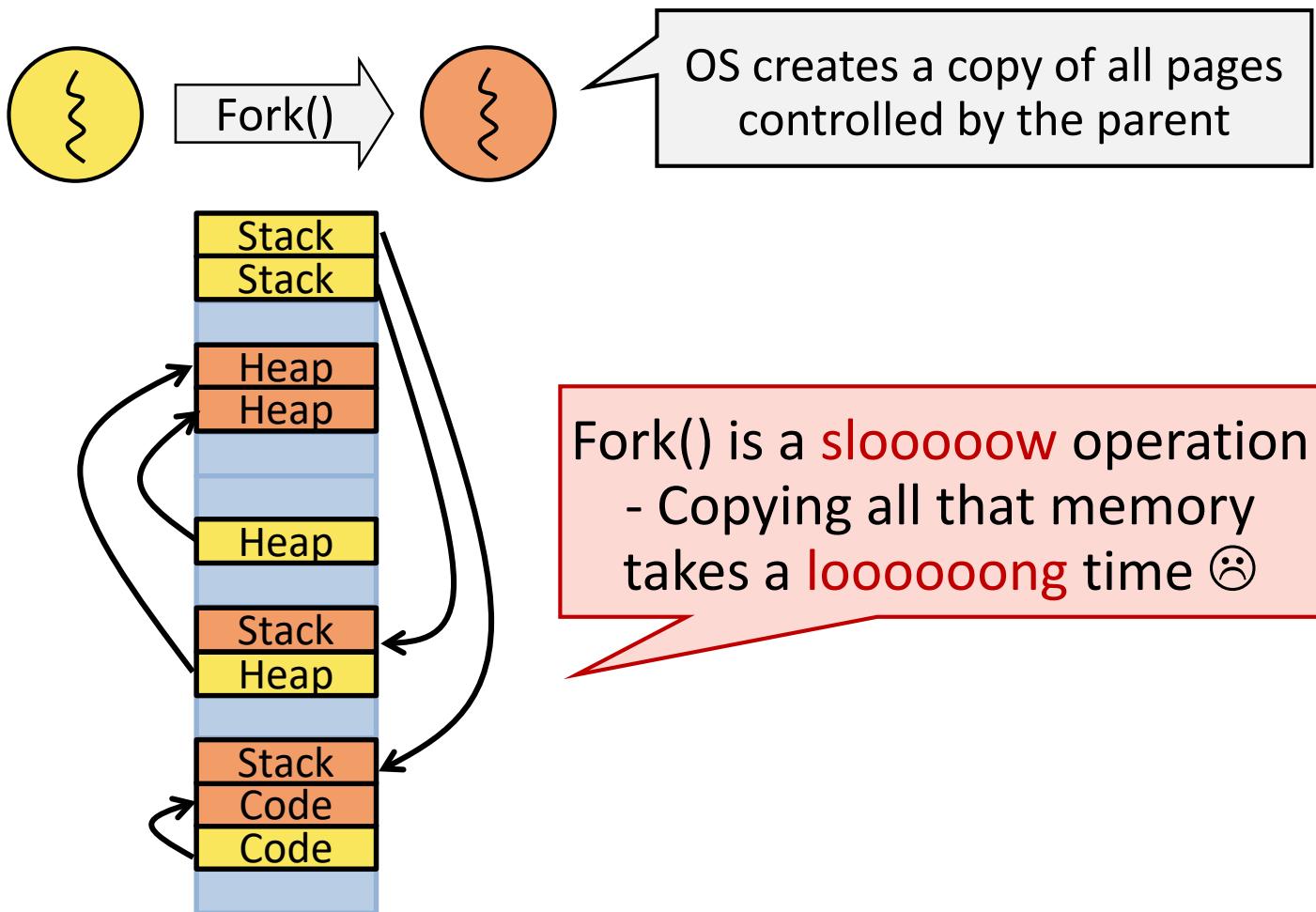


Functionality #2:

Copy-on-Write

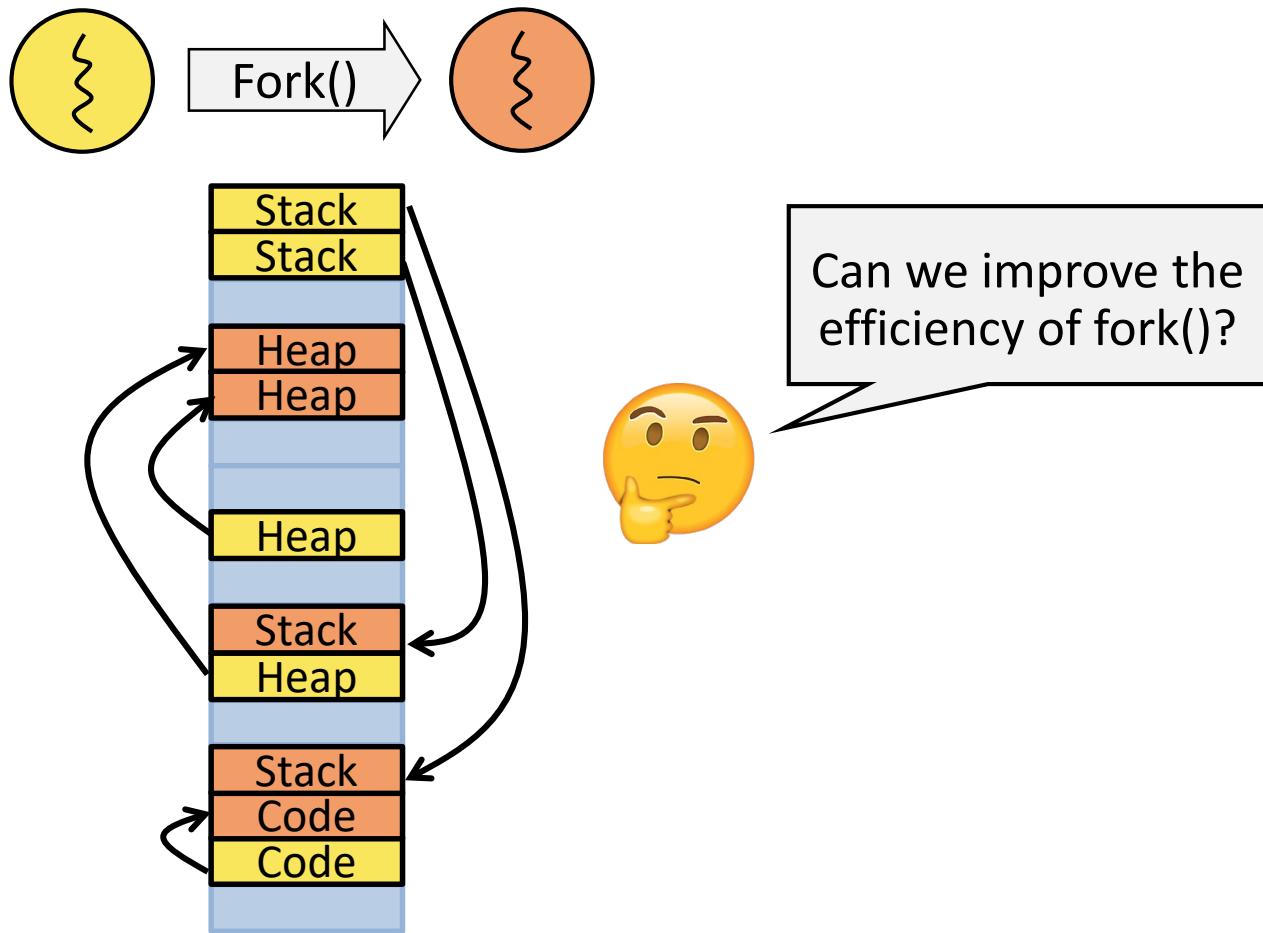
[Recall] fork()

- Recall how fork() is implemented



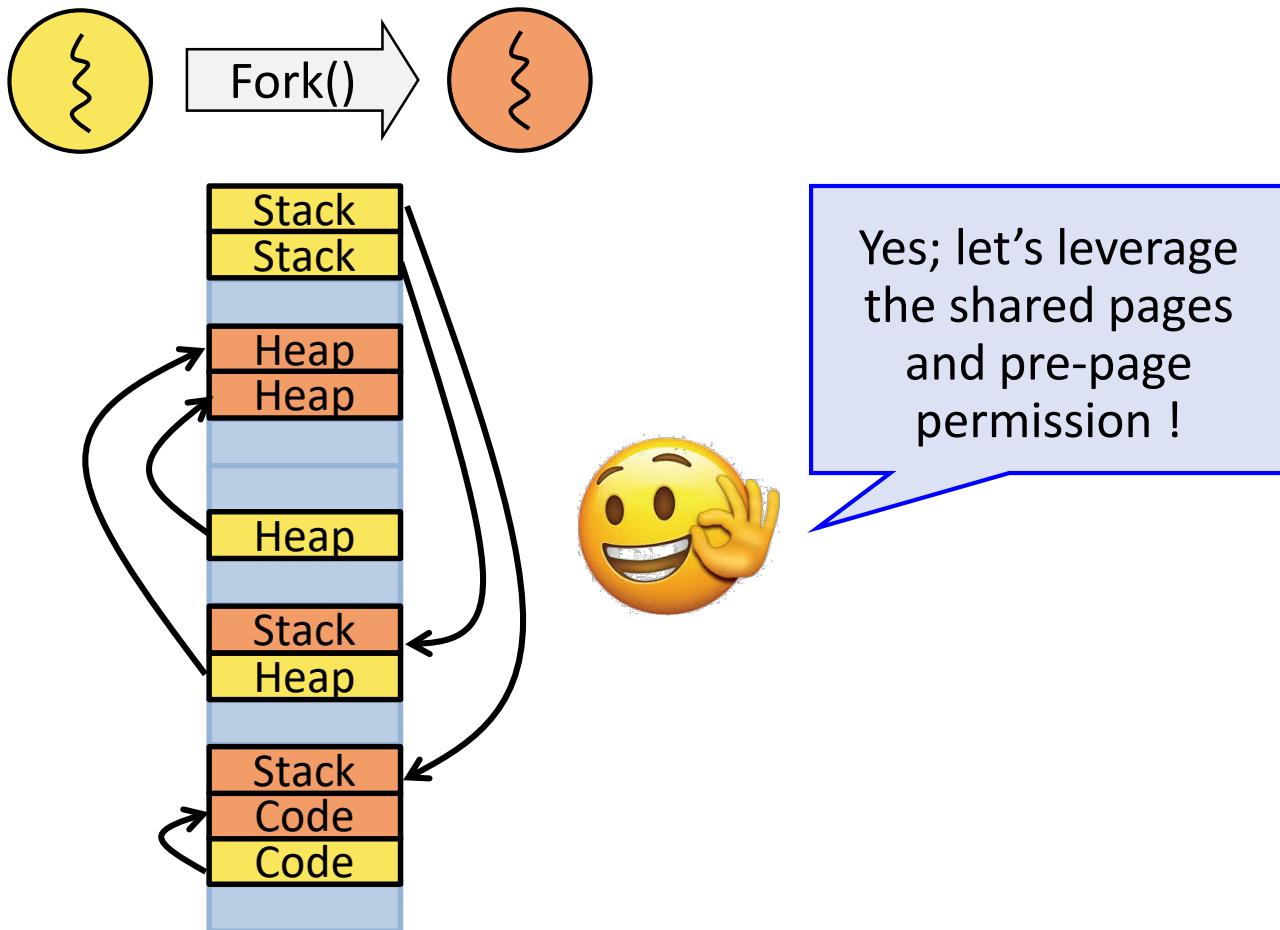
[Recall] fork()

- Recall how fork() is implemented



[Recall] fork()

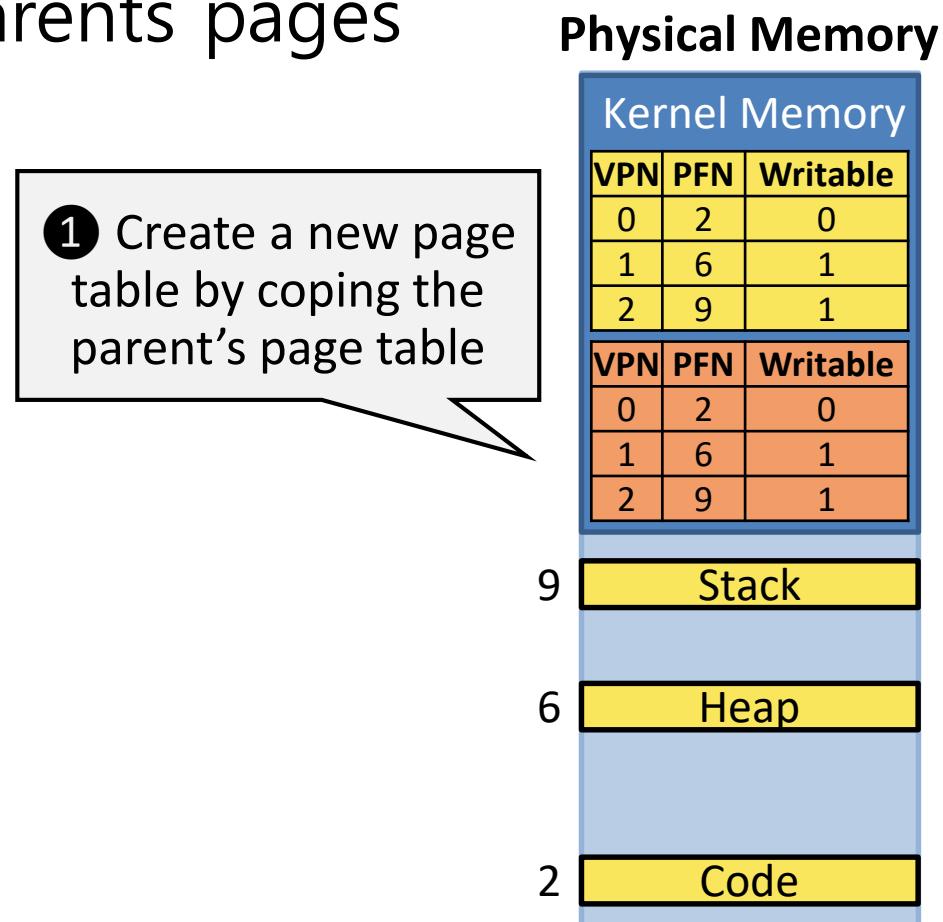
- Recall how fork() is implemented





Solution: Copy-on-Write

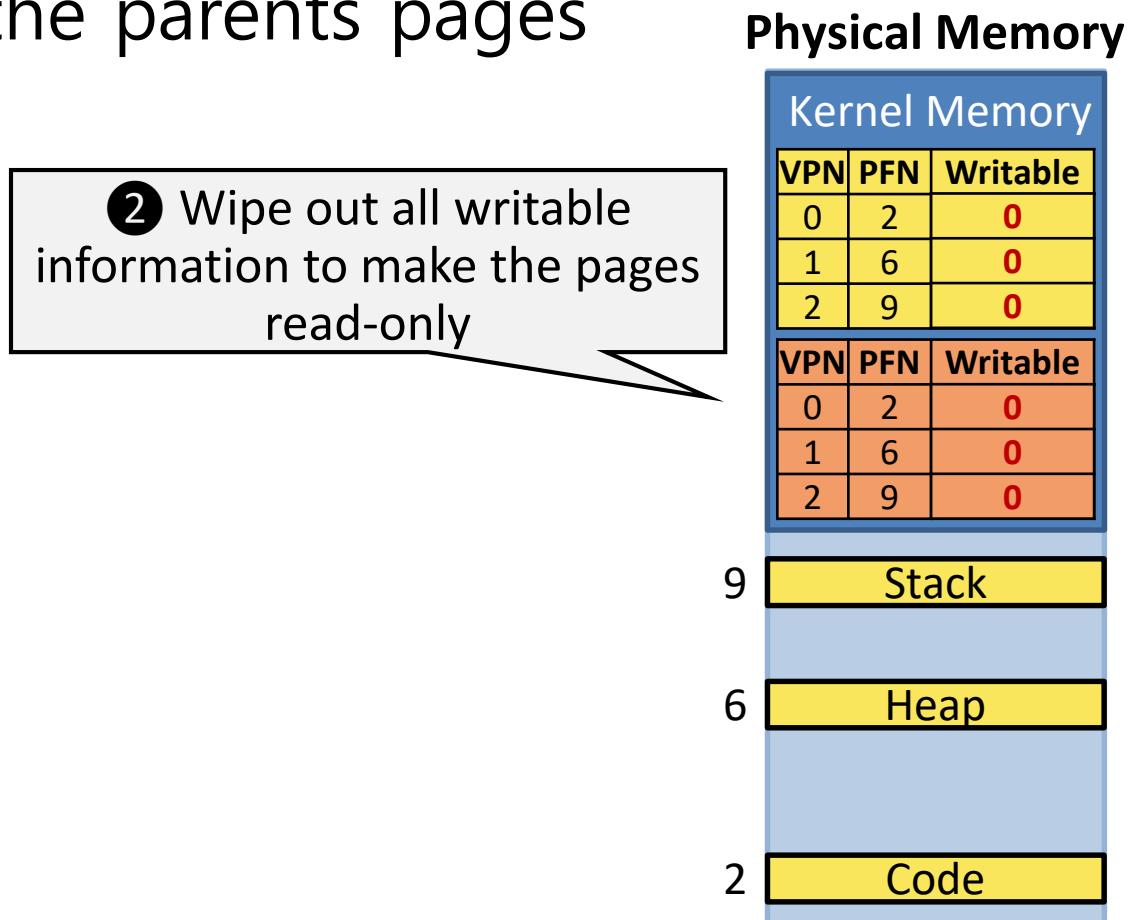
- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages





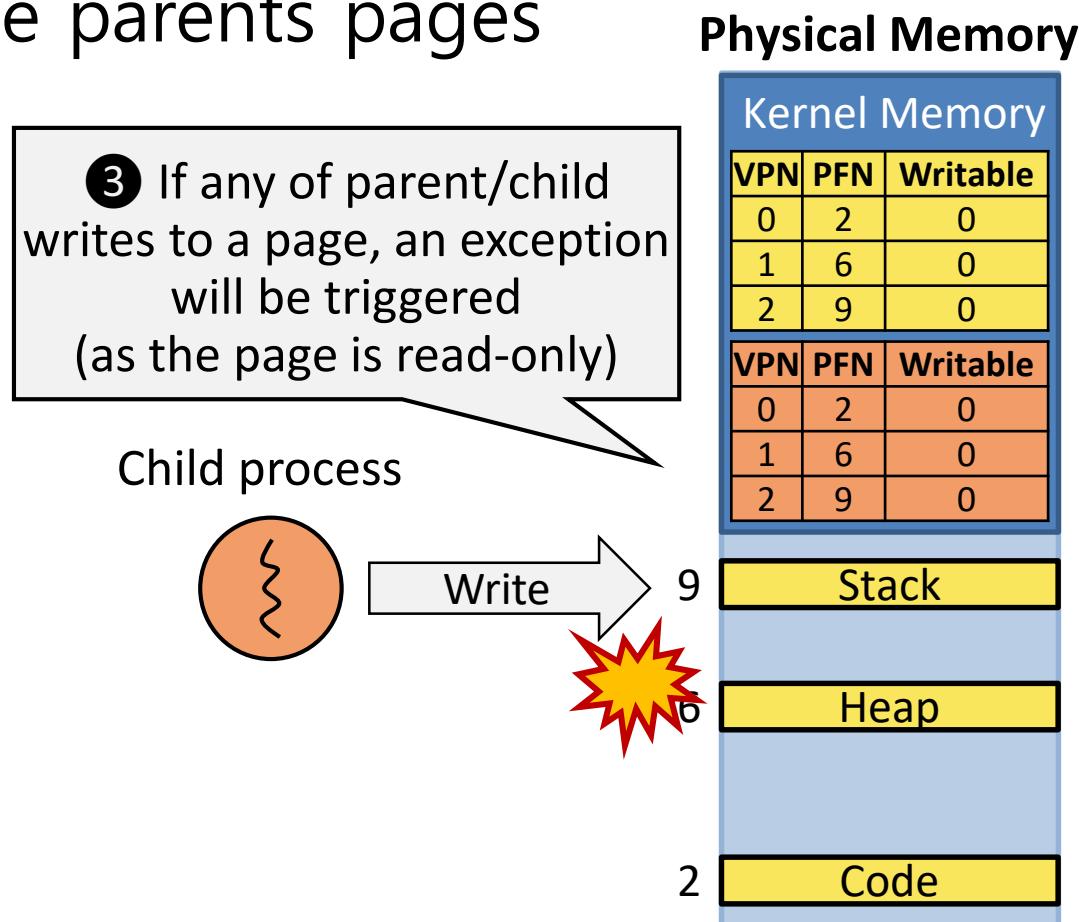
Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages



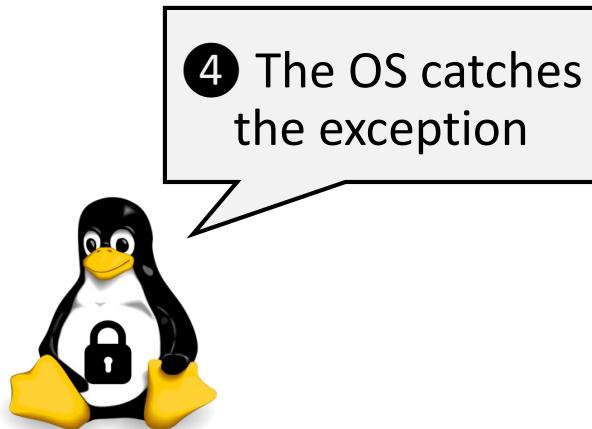
Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages



Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages



Physical Memory

Kernel Memory		
VPN	PFN	Writable
0	2	0
1	6	0
2	9	0

VPN	PFN	Writable
0	2	0
1	6	0
2	9	0

The diagram illustrates the layout of physical memory. At the top is the Kernel Memory, consisting of two identical tables for pages 0, 1, and 2. Below the kernel is the User Memory, divided into four sections: Stack (top), Heap, and Code (bottom). The Stack section is at address 9, the Heap at 6, and the Code at 2.

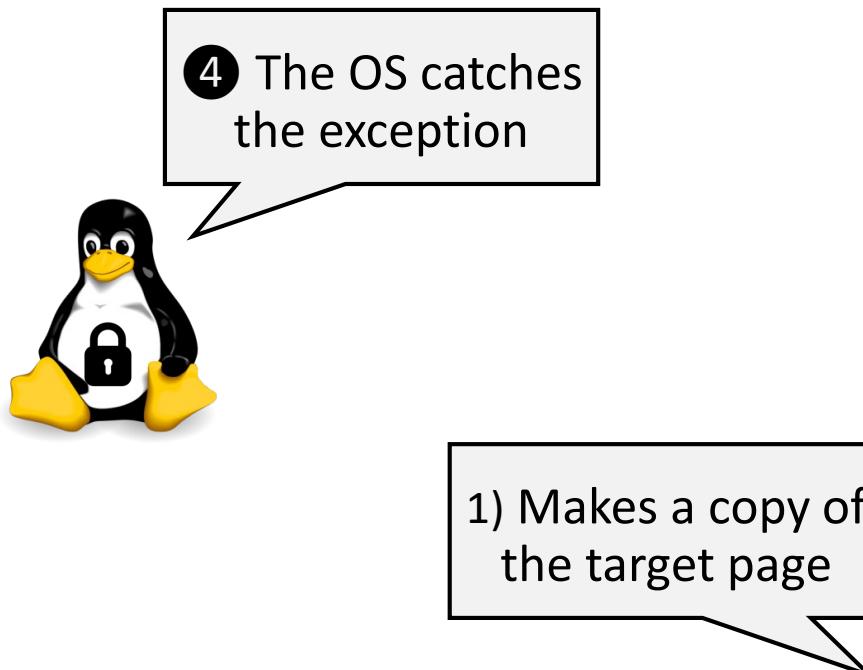
Stack

Heap

Code

Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages



Physical Memory

Kernel Memory		
VPN	PFN	Writable
0	2	0
1	6	0
2	9	0

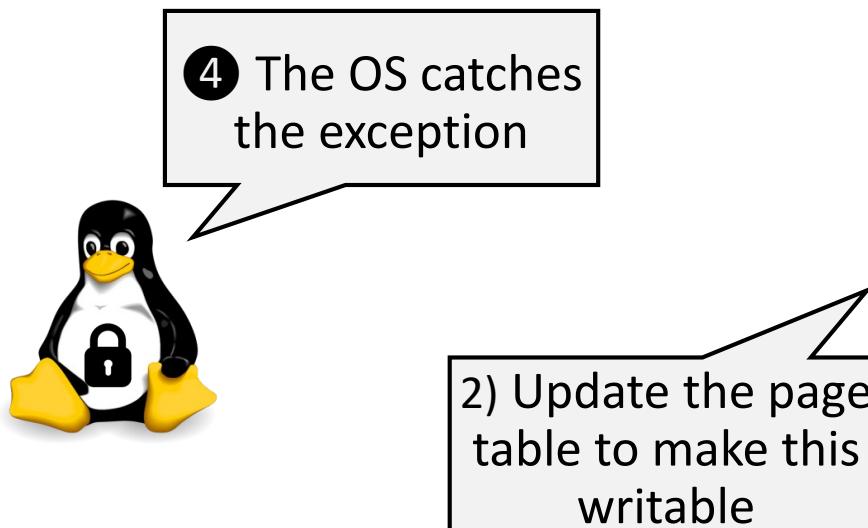
VPN	PFN	Writable
0	2	0
1	6	0
2	9	0

The physical memory diagram shows pages 2, 6, and 9. Page 2 is orange (Stack), page 6 is light blue (Heap), and page 9 is yellow (Stack). A curved arrow points from the bottom stack page in the kernel memory table to the bottom stack page in physical memory.

Page	Type
9	Stack
6	Heap
3	Stack
2	Code

Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages



Physical Memory

Kernel Memory		
VPN	PFN	Writable
0	2	0
1	6	0
2	9	1

VPN	PFN	Writable
0	2	0
1	6	0
2	3	1

9 Stack

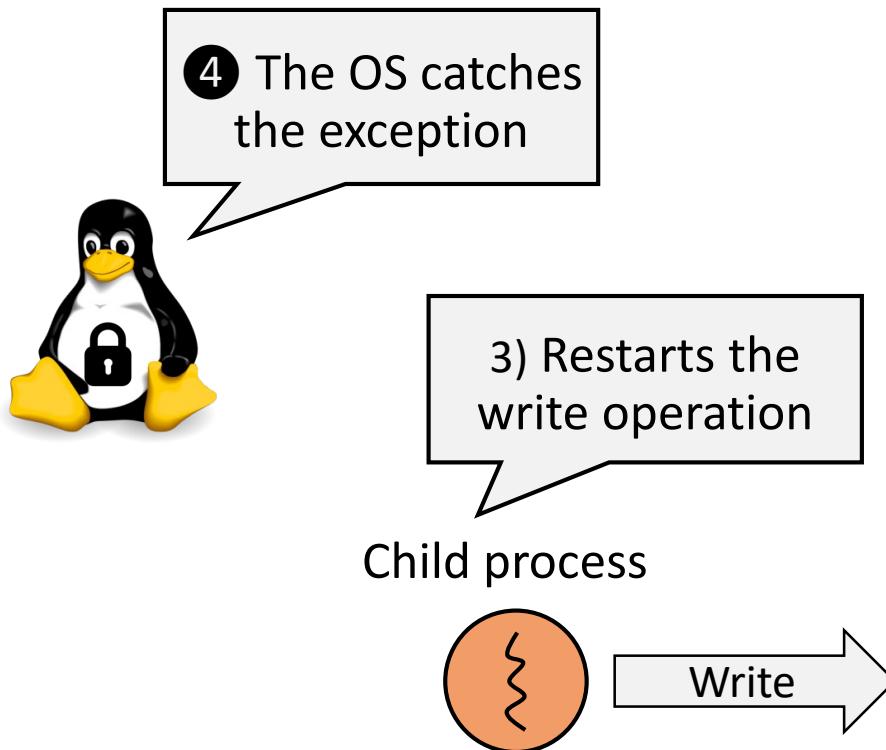
6 Heap

3 Stack

2 Code

Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages



Physical Memory

Kernel Memory		
VPN	PFN	Writable
0	2	0
1	6	0
2	9	1

VPN	PFN	Writable
0	2	0
1	6	0
2	3	1

9	Stack	
6	Heap	
3	Stack	
2	Code	



Solution: Copy-on-Write

- Key idea: rather than copying all of the parents pages, create a new page table for the child, which maps to all of the parents pages

All unmodified data are shared without a copy 😊
- This modifies the pages only if there are requests (on-demand)

Physical Memory

Kernel Memory		
VPN	PFN	Writable
0	2	0
1	6	0
2	9	1

VPN	PFN	Writable
0	2	0
1	6	0
2	3	1

Physical Memory

Kernel Memory

Physical Memory

9 Stack

6 Heap

3 Stack

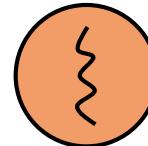
2 Code



Zero-on-Reference

What if the child process touches the heap?

Child process



Write

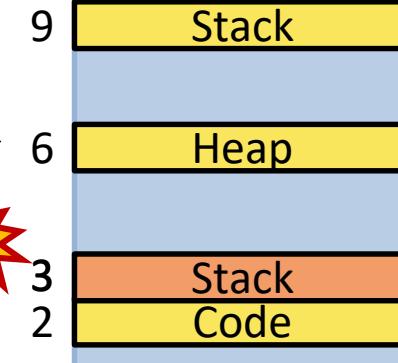


Physical Memory

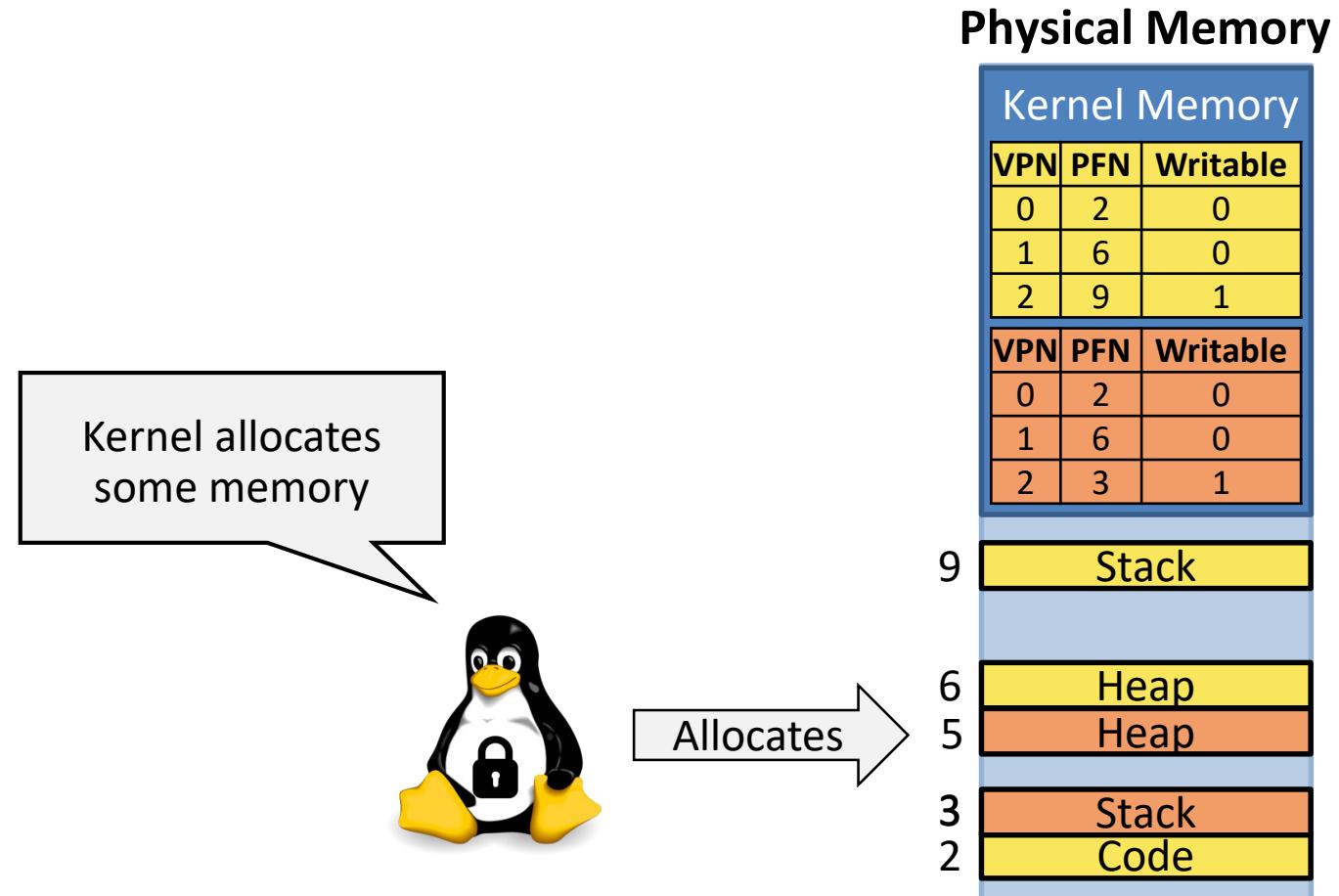
Kernel Memory

VPN	PFN	Writable
0	2	0
1	6	0
2	9	1

VPN	PFN	Writable
0	2	0
1	6	0
2	3	1



Zero-on-Reference



Zero-on-Reference

Zeros the memory as well
→ To avoid accidentally leaking information!



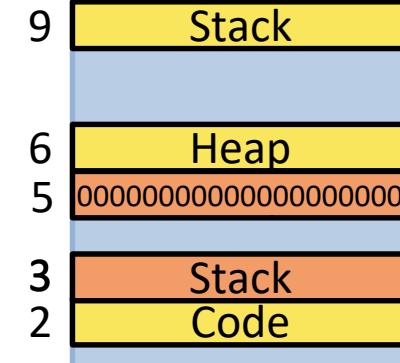
Allocates

Physical Memory

Kernel Memory

VPN	PFN	Writable
0	2	0
1	6	0
2	9	1

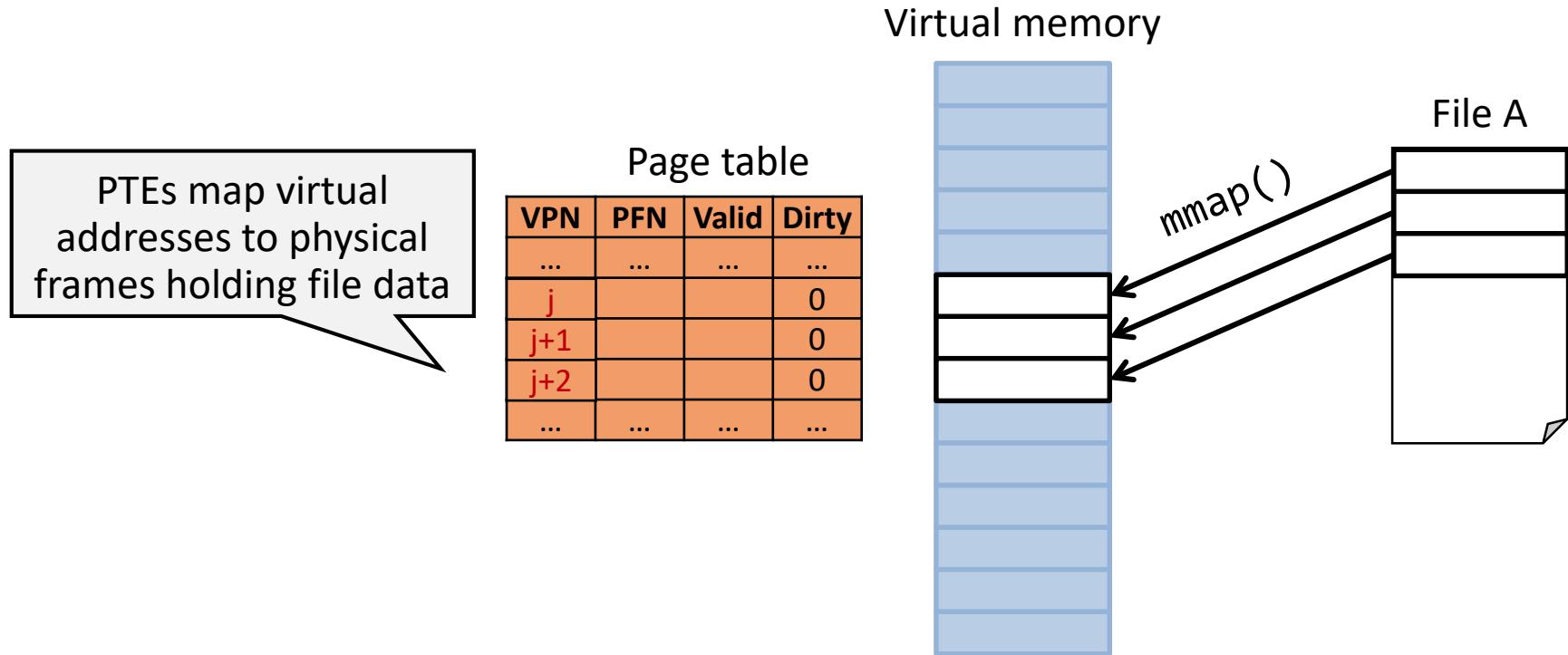
VPN	PFN	Writable
0	2	0
1	6	0
2	3	1



Functionality #3: Mapped Files

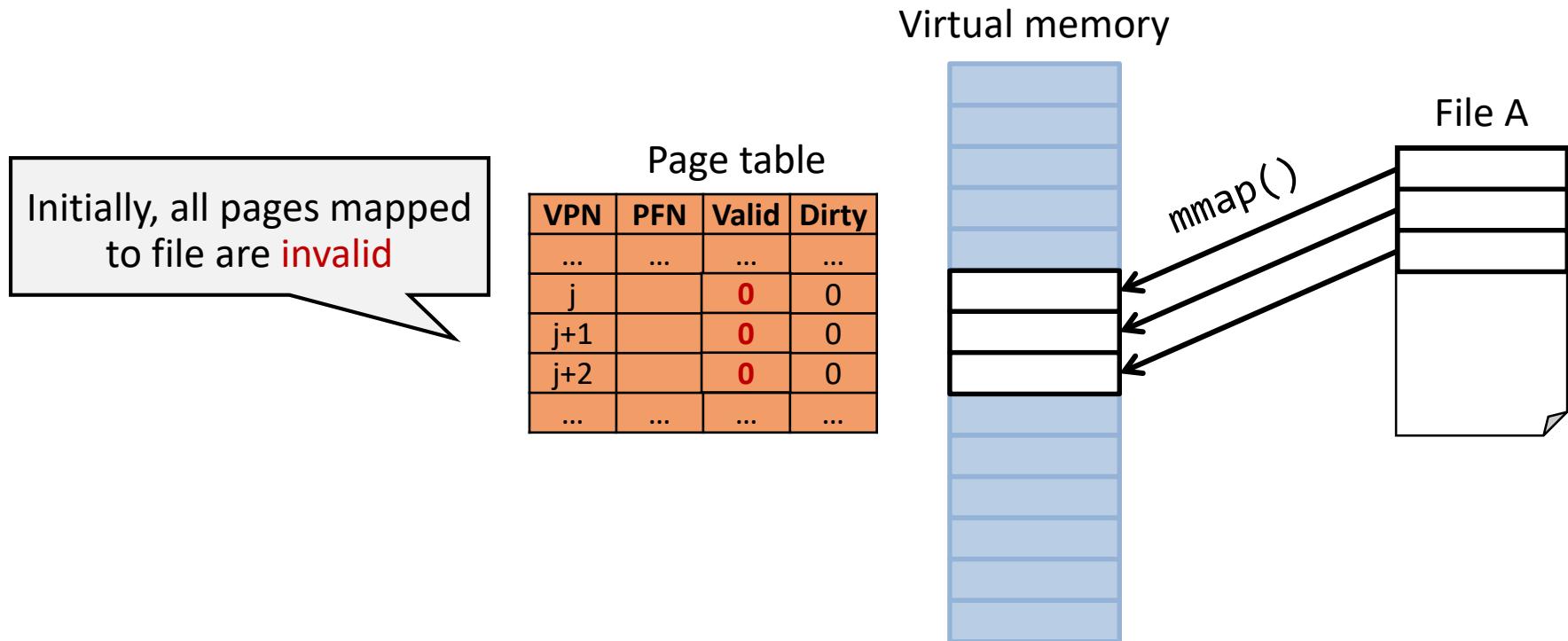
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores (Instead of "open, read into buffer, operate on buffer, ...")



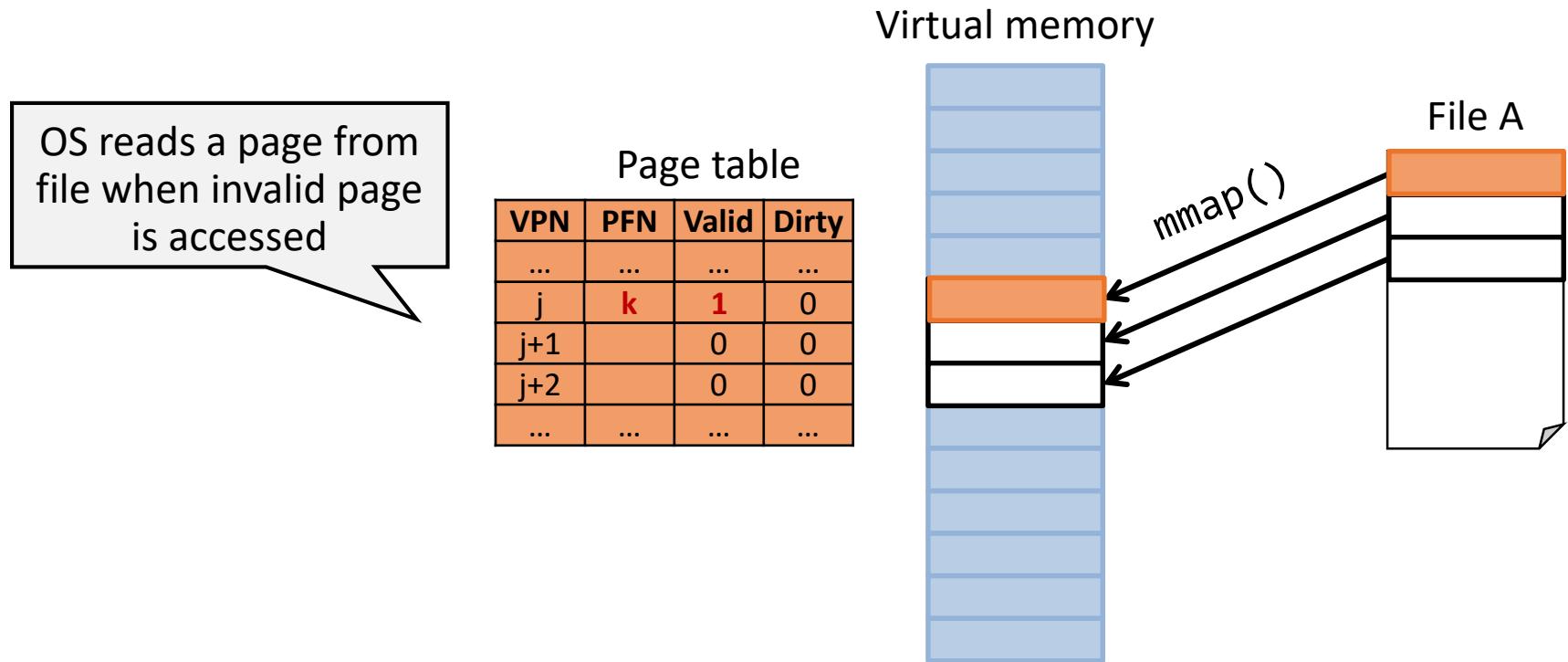
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores (Instead of "open, read into buffer, operate on buffer, ...")



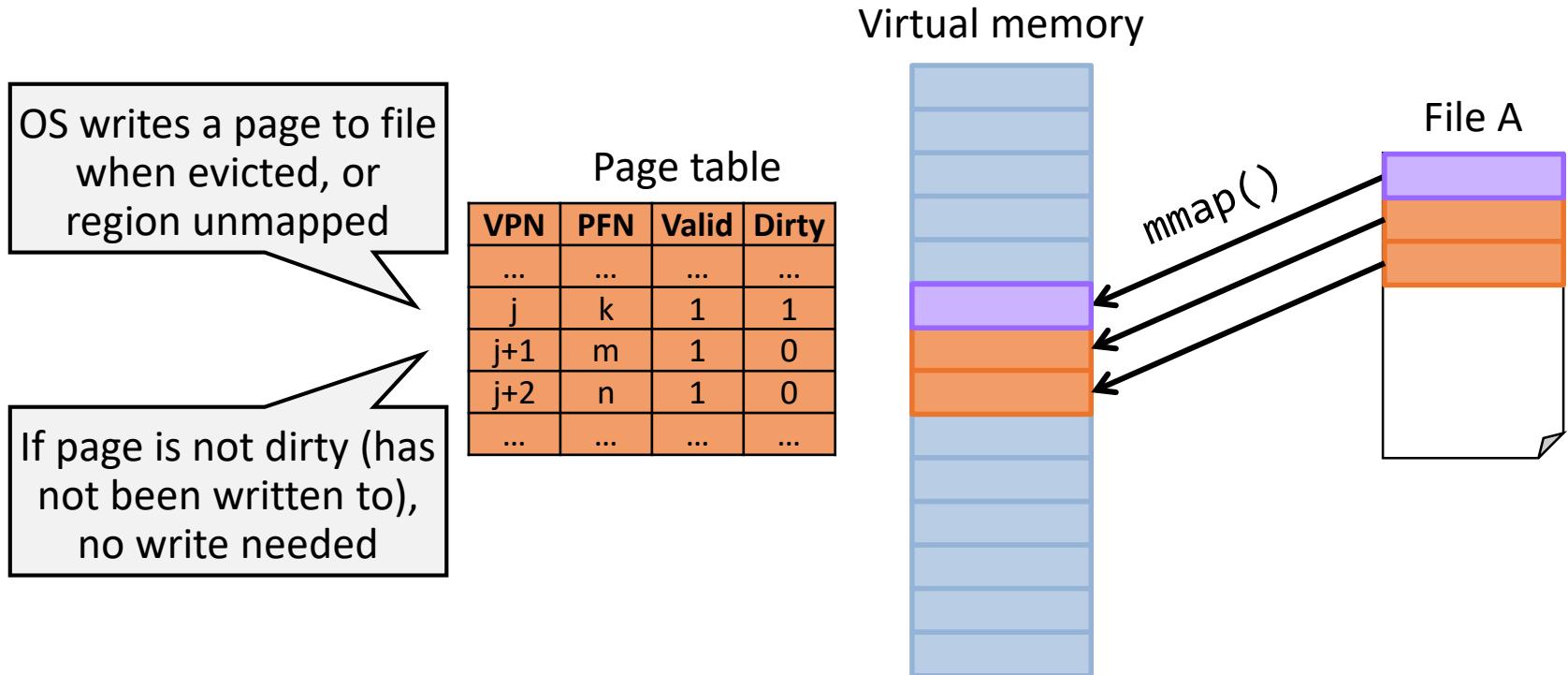
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores (Instead of "open, read into buffer, operate on buffer, ...")



Mapped Files

- Mapped files enable processes to do file I/O using loads and stores (Instead of "open, read into buffer, operate on buffer, ...")



What's Next?

- Let's check how the paging system can be optimized ;)

21S EE415

Advanced Paging

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



Advanced Paging

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

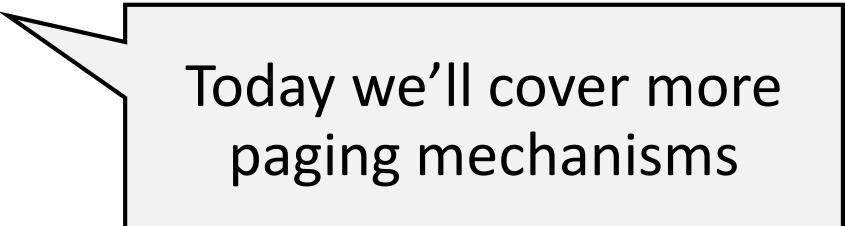
CAMEL*ab*



Optimizations

#1: Managing page tables (space-efficient)

#2: Efficient translations (time-efficient)

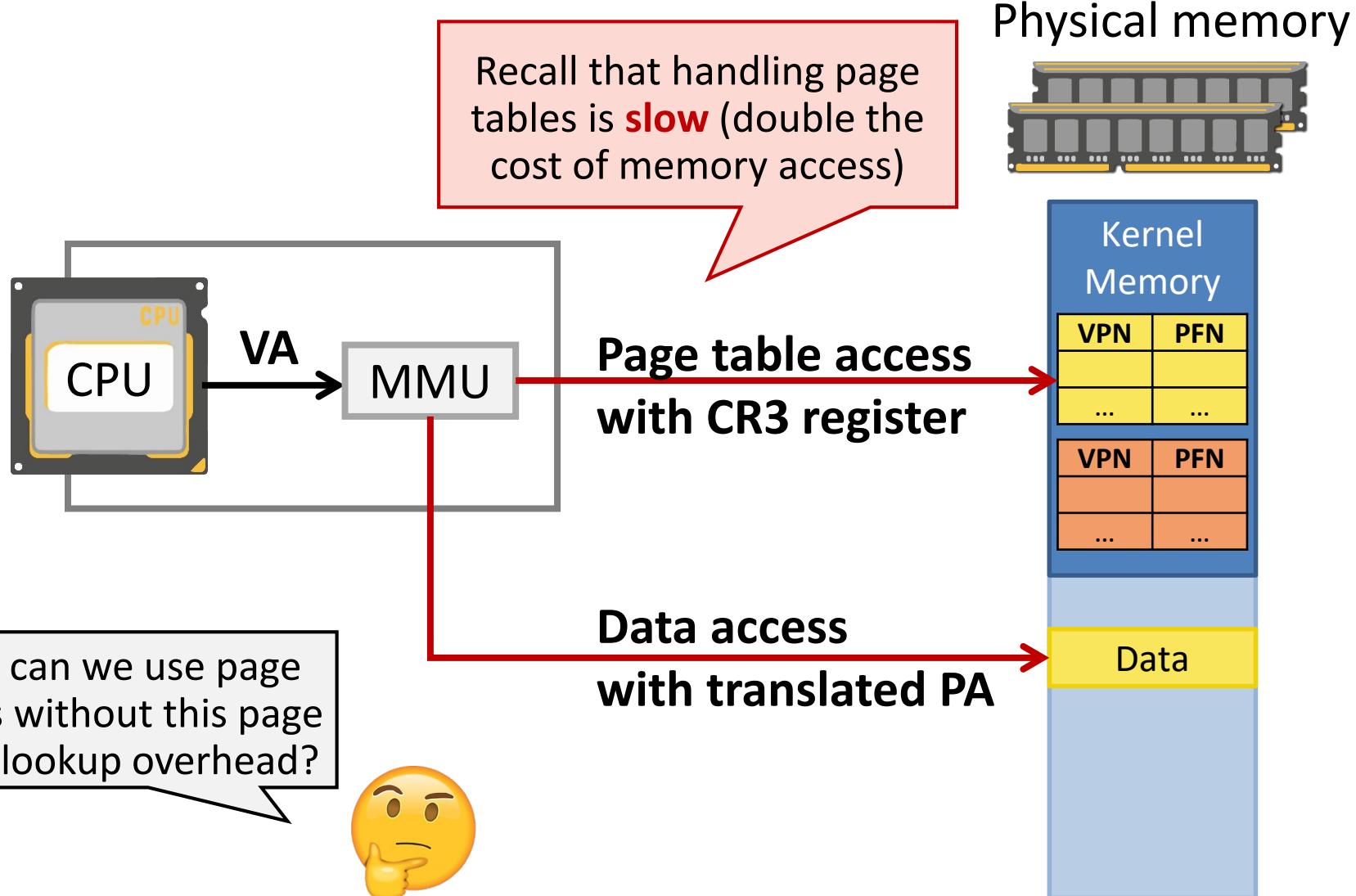


Today we'll cover more
paging mechanisms

Optimization #1: Efficient translations

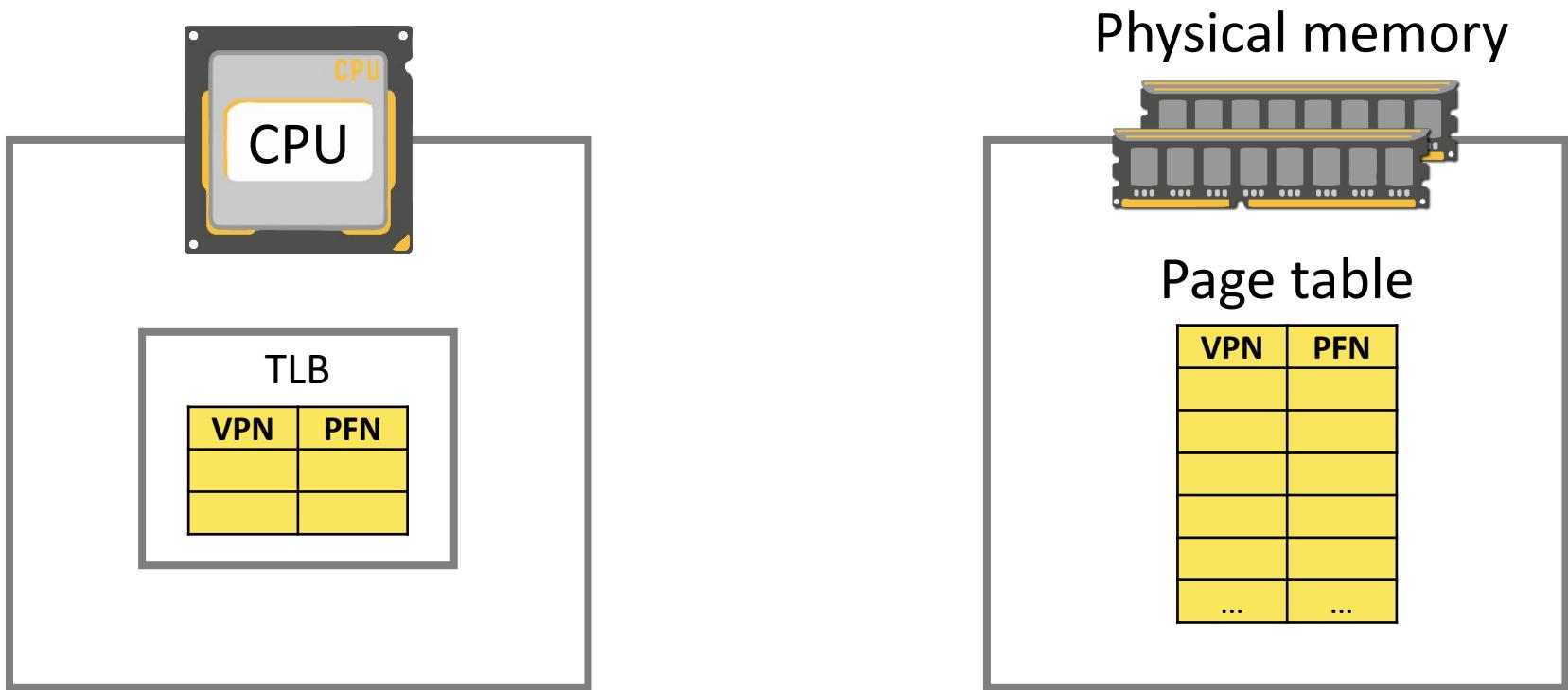
- Time optimization

Problem: Page Table Speed



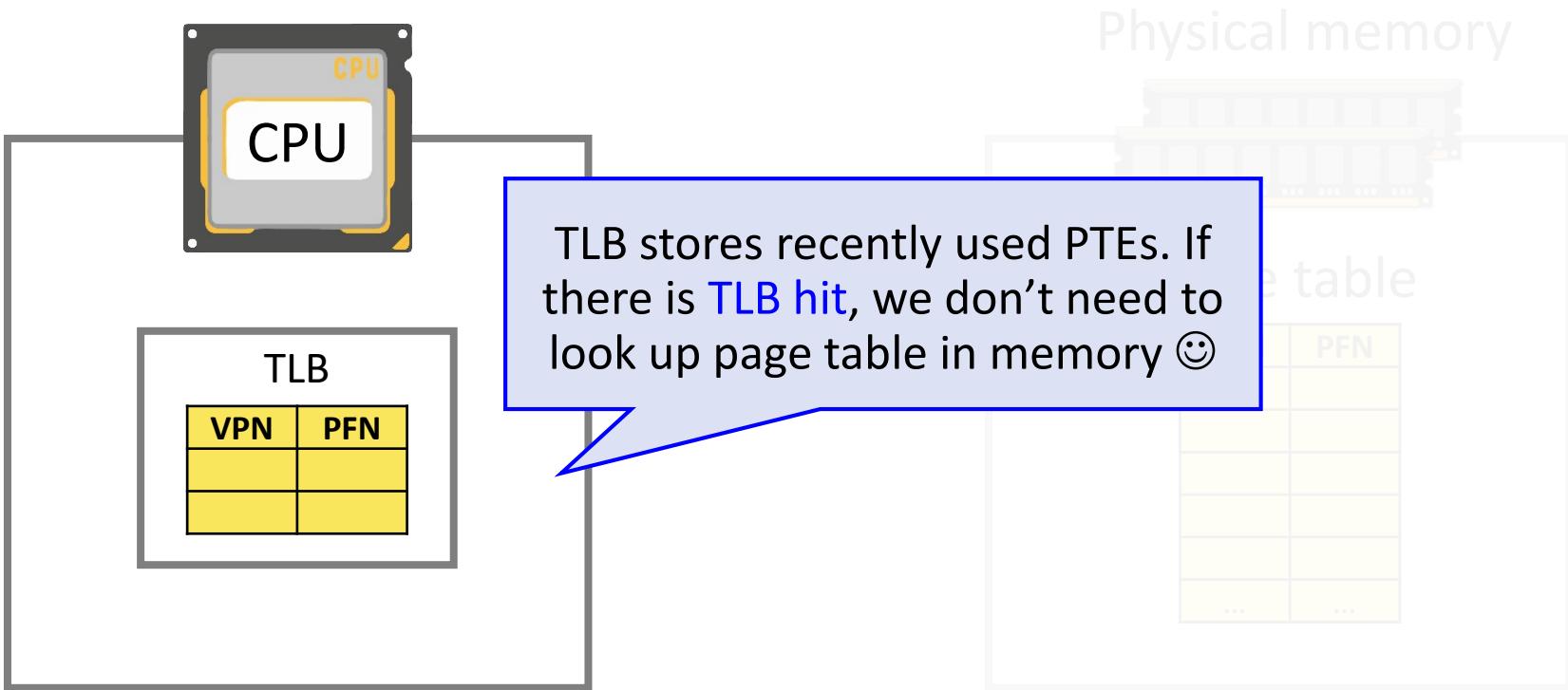
Solution: Translation Lookaside Buffer (TLB)

- Key idea: Caching page table entries directly in the CPU's MMU (i.e., address translation cache)



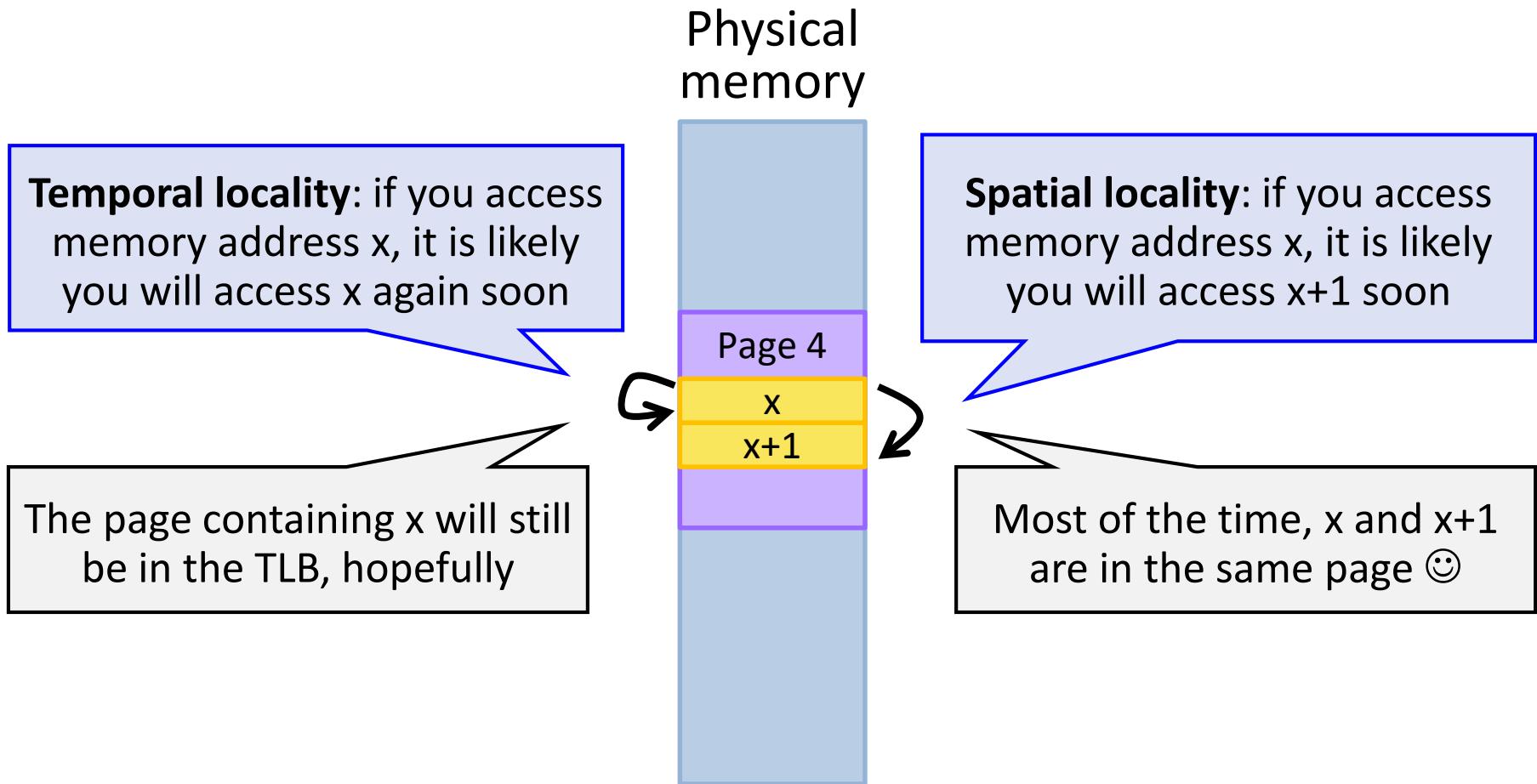
Solution: Translation Lookaside Buffer (TLB)

- Key idea: Caching page table entries directly in the CPU's MMU (i.e., address translation cache)



Insight: Locality

- Like any cache, TLB is effective because of locality



Toy Example

- Reading an array

Process 1's view
of virtual memory



[Assumption]
We have a 10KB array
of integers

[0]	[1]	[2]	[3]	[4]	..	[2558]	[2559]
-----	-----	-----	-----	-----	----	--------	--------

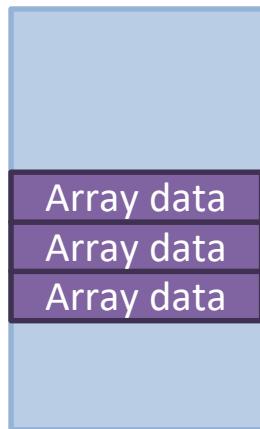


$10\text{KB} / 4\text{B} = 2560$ integers in the array

Toy Example

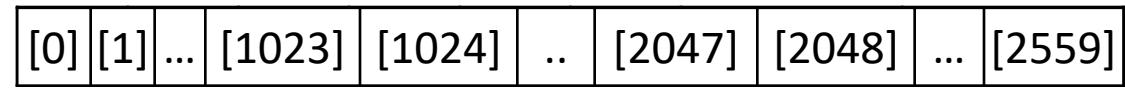
- Reading an array

Process 1's view
of virtual memory



3 pages are required
to store the array
(4KB pages)

Page j+2
Page j+1
Page j

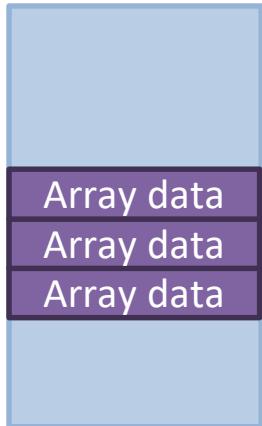


$$10\text{KB} / 4\text{B} = 2560 \text{ integers in the array}$$

Toy Example (no TLB)

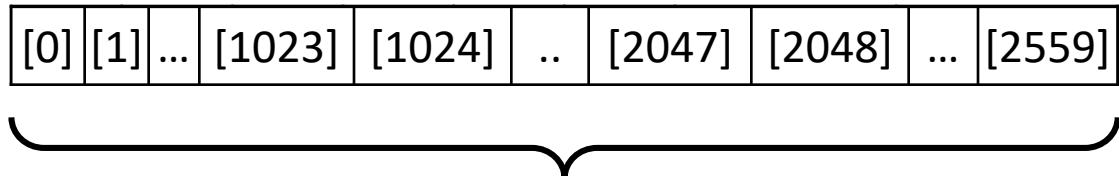
- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j

With no TLB, how many
memory accesses are required
to read the whole array?

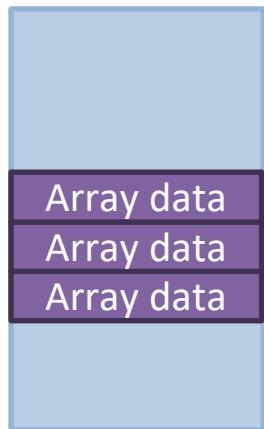


$$10\text{KB} / 4\text{B} = 2560 \text{ integers in the array}$$

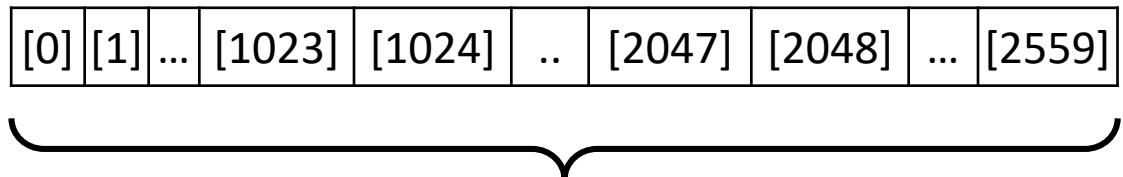
Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

Every time where you access an entry
of the array, it induces a page table
lookup and a memory read

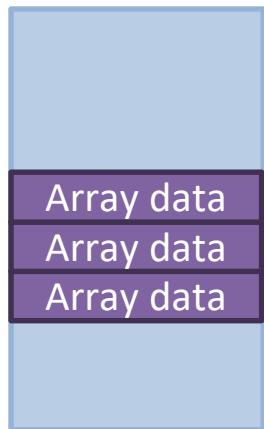
Memory accesses

Page table lookups	Data read

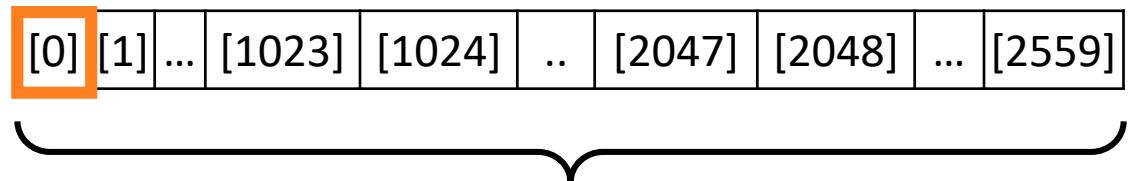
Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

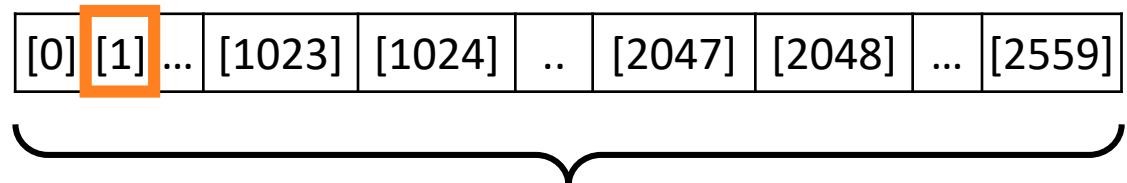
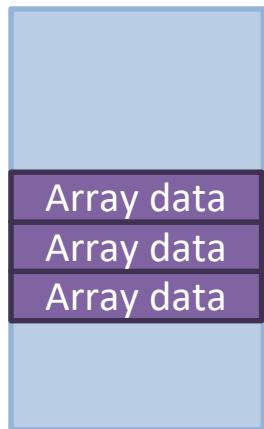
Memory accesses

Page table lookups	Data read
1	1

Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



$10\text{KB} / 4\text{B} = 2560$ integers in the array

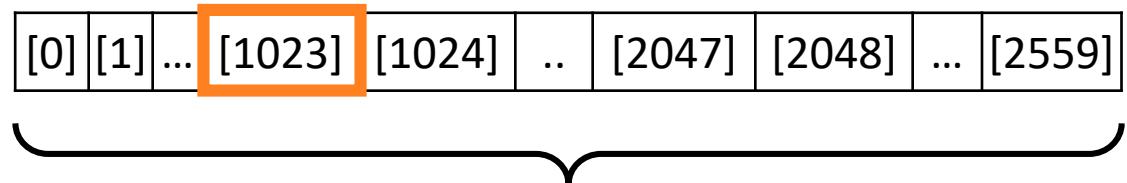
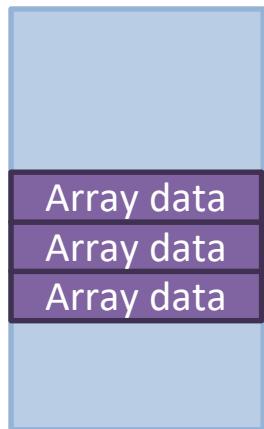
Memory accesses

Page table lookups	Data read
2	2

Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



$10\text{KB} / 4\text{B} = 2560$ integers in the array

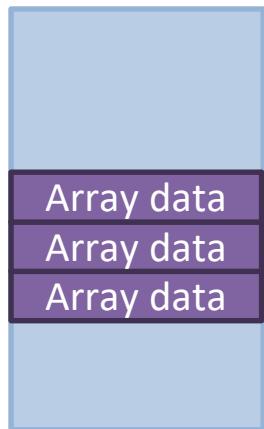
Memory accesses

Page table lookups	Data read
1024	1024

Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

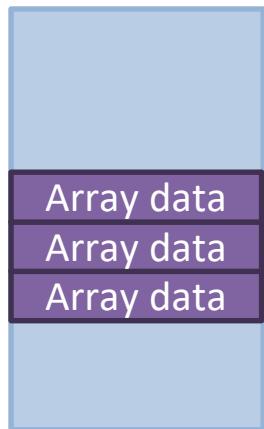
Memory accesses

Page table lookups	Data read
1025	1025

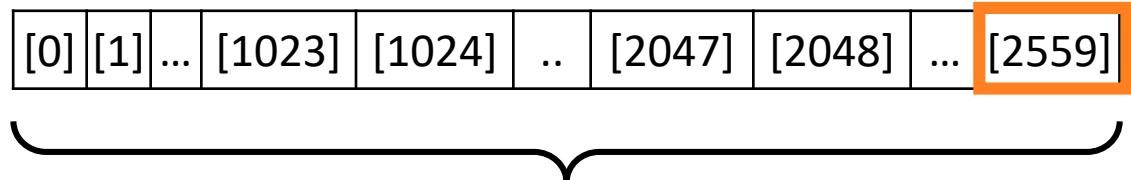
Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

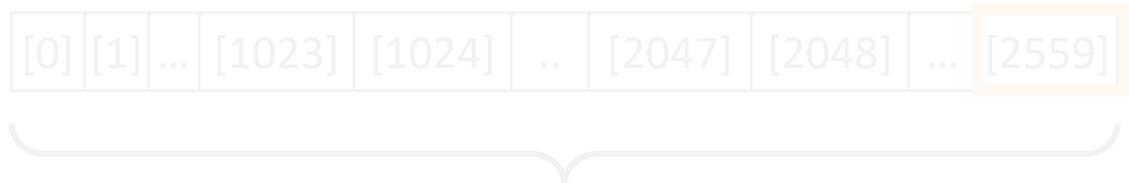
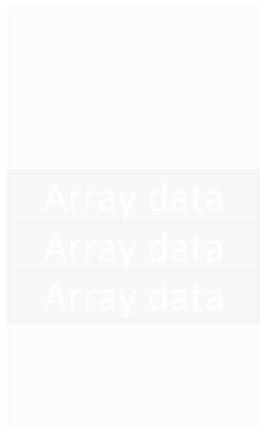
Memory accesses

Page table lookups	Data read
2560	2560

Toy Example (no TLB)

- Reading an array

Process 1's view
of virtual memory



$10\text{KB} / 4\text{B} = 2560$ integers in the array

For data, there are **2560 * 2** reads in total. There are more; remember, instructions to handle the operand also access memory.

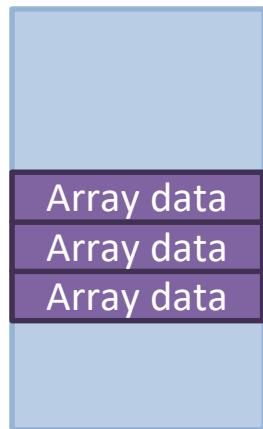
Memory accesses

Page table lookups	Data read
2560	2560

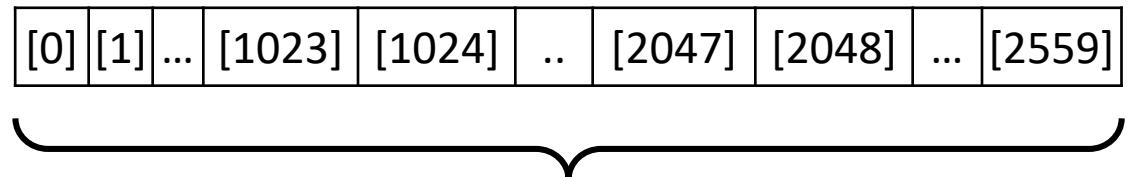
Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

VPN	PFN

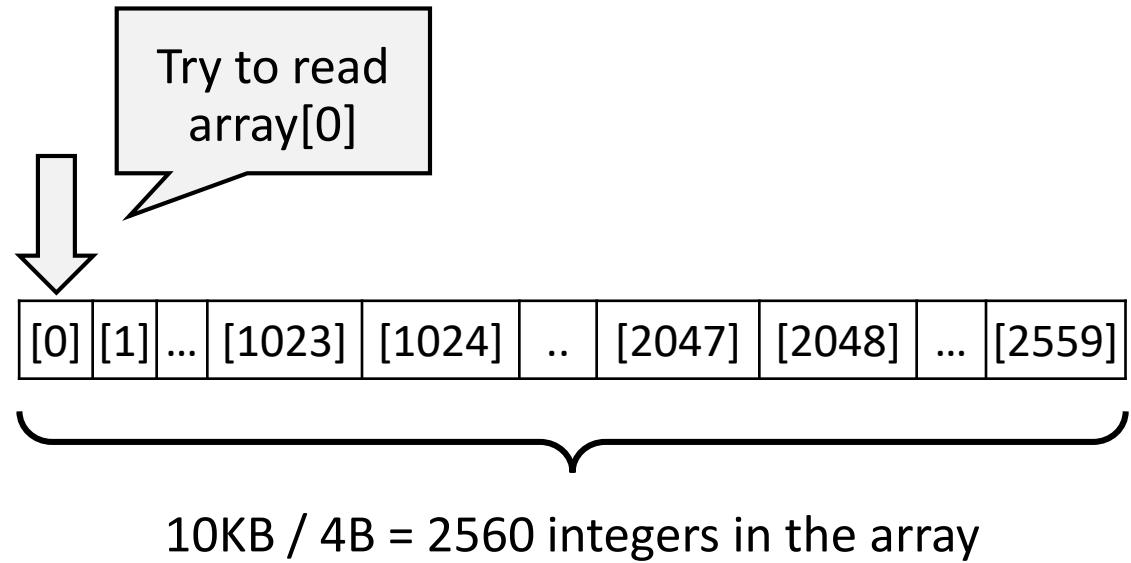
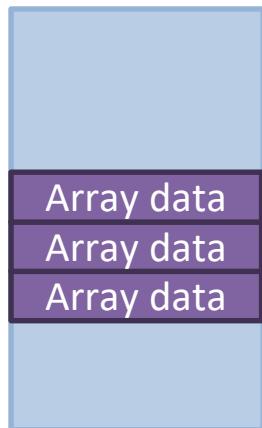
[Assumption]
The TLB starts off
cold (i.e., empty)

# Memory accesses	
Page table lookups	Data read

Toy Example (with TLB)

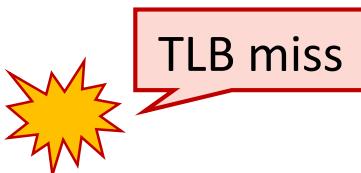
- Reading an array

Process 1's view
of virtual memory



TLB

VPN	PFN



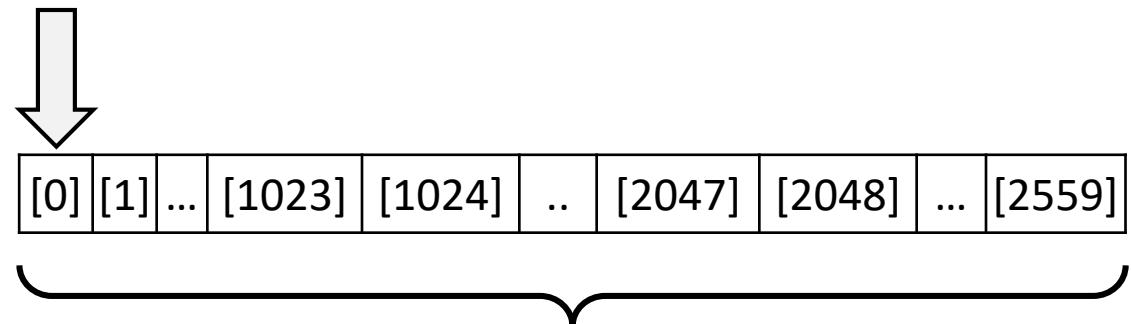
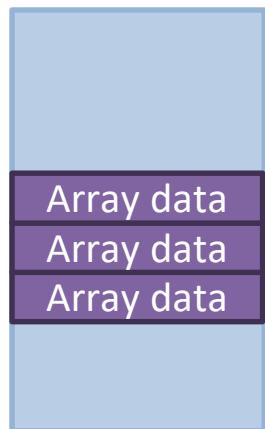
Memory accesses

Page table lookups	Data read

Toy Example (with TLB)

- Reading an array

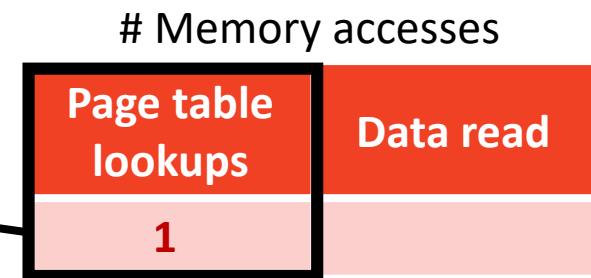
Process 1's view
of virtual memory



$10\text{KB} / 4\text{B} = 2560$ integers in the array

TLB

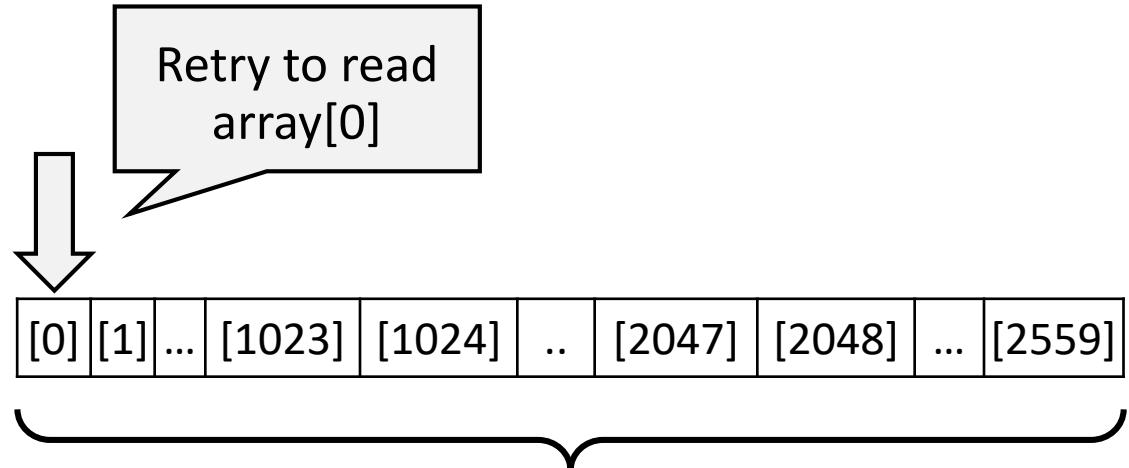
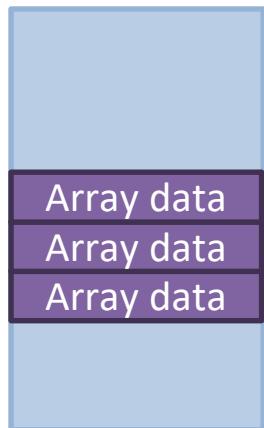
VPN	PFN
j	a



Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



TLB

VPN	PFN
j	a

TLB hit



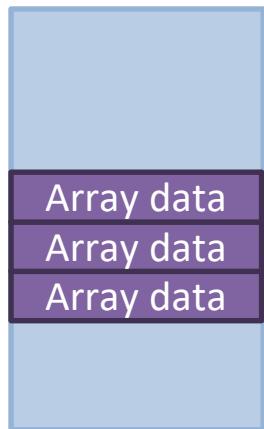
Memory accesses

Page table lookups	Data read
1	1

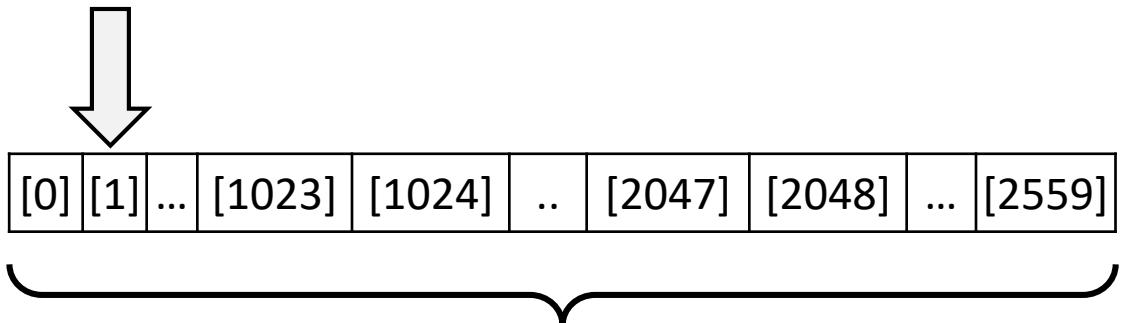
Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

TLB

VPN	PFN
j	a

TLB hit



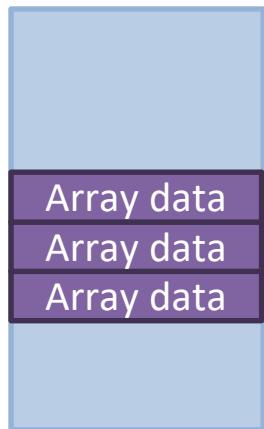
Memory accesses

Page table lookups	Data read
1	2

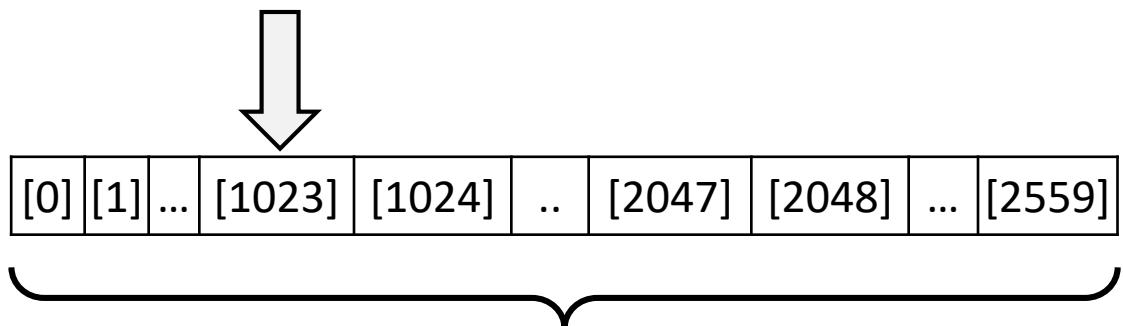
Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

TLB

VPN	PFN
j	a

TLB hit



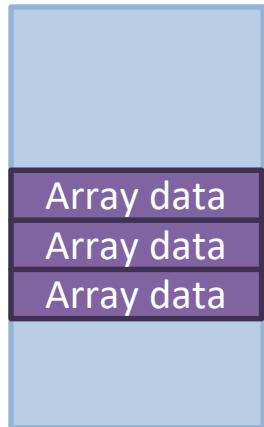
Memory accesses

Page table lookups	Data read
1	1024

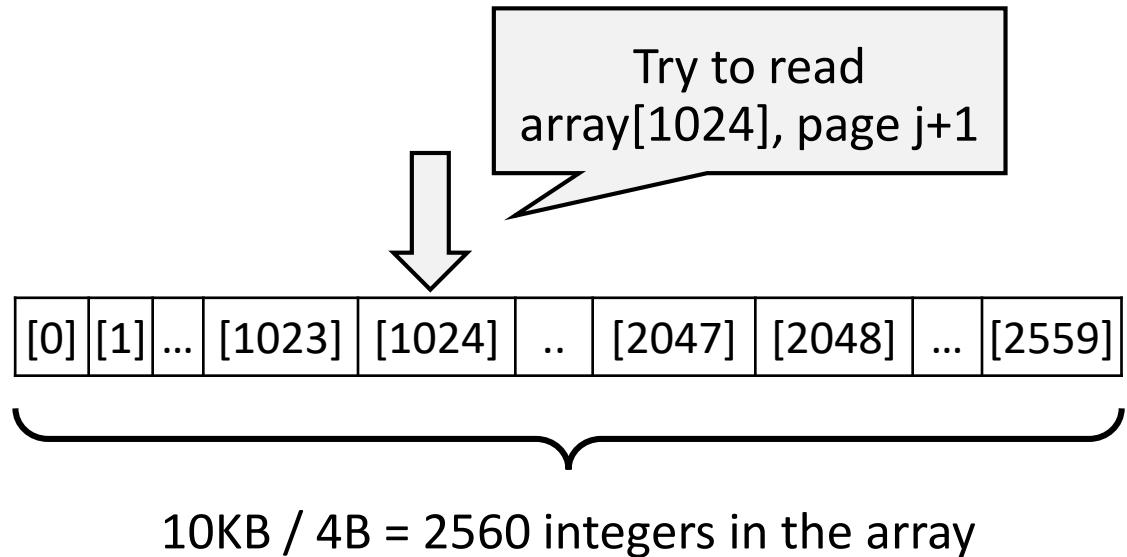
Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory

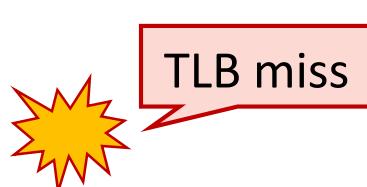


Page j+2
Page j+1
Page j



TLB

VPN	PFN
j	a
j+1	b



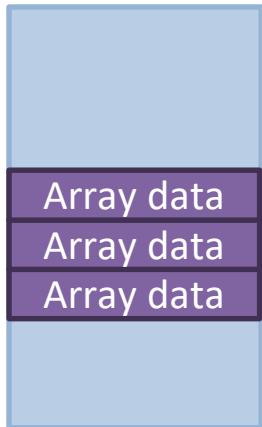
Memory accesses

Page table lookups	Data read
2	1025

Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j

Try to read
array[2048], page j+2



$10\text{KB} / 4\text{B} = 2560$ integers in the array

TLB

VPN	PFN
j	a
j+1	b
j+2	c

TLB miss

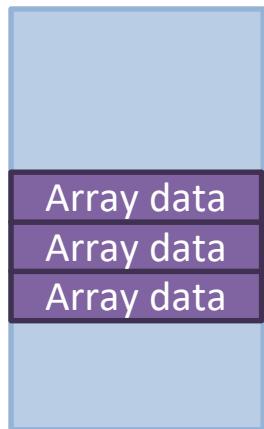
Memory accesses

Page table lookups	Data read
3	2049

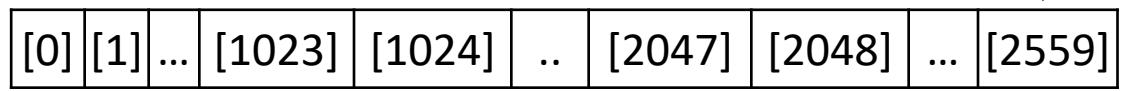
Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



Try to read
array[2559], page j+2



10KB / 4B = 2560 integers in the array

TLB

VPN	PFN
j	a
j+1	b
j+2	c



TLB hit

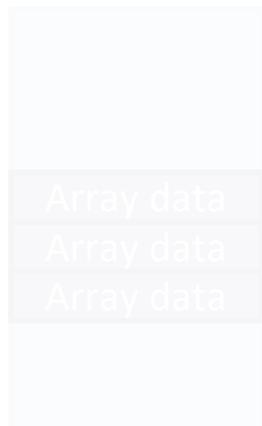
Memory accesses

Page table lookups	3
Data read	2560

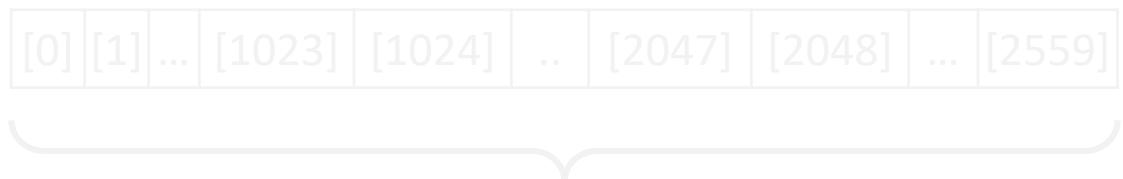
Toy Example (with TLB)

- Reading an array

Process 1's view
of virtual memory



Page j+2
Page j+1
Page j



$10\text{KB} / 4\text{B} = 2560$ integers in the array

TLB

VPN	PFN
j	a
j+1	b
j+2	c

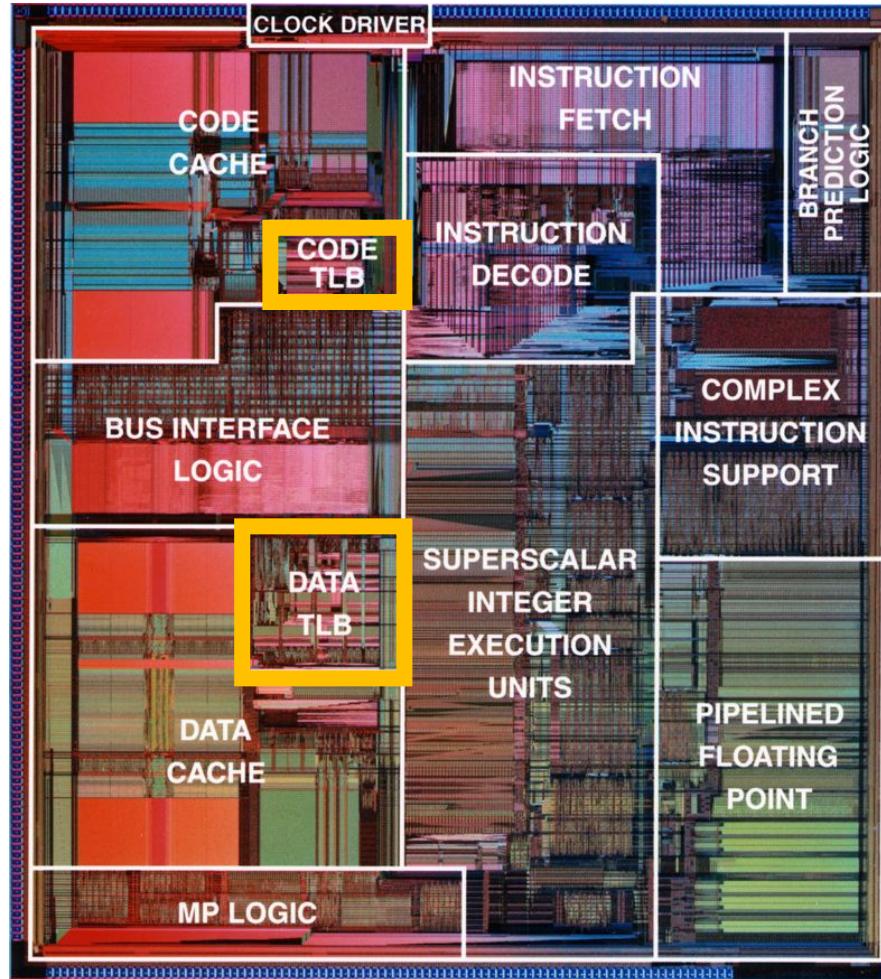
There are 2560 + 3 reads for data
in total → TLB hit rate: 96%

Memory accesses

Page table lookups	Data read
3	2560

Implementation

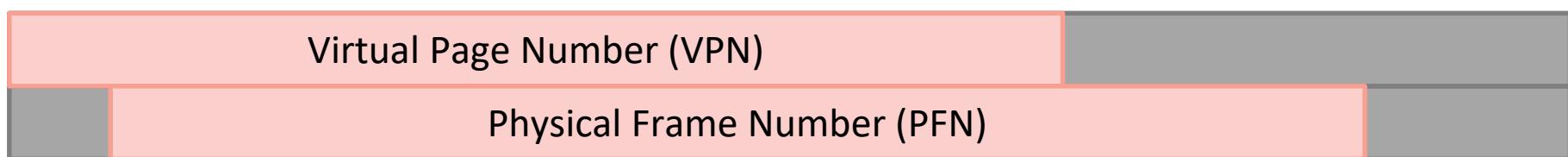
Let's check how the TLB can be implemented



Intel Pentium Processor (1993)

#1: TLB Entry

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



- VPN & PFN – virtual and physical pages

We should be careful to manage TLB by considering caches. TLB entries may not be valid after a context switch

#1: TLB Entry

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Virtual Page Number (VPN)

Physical Frame Number (PFN)

- VPN & PFN – virtual and physical pages

Process 1's Page Table

VPN	PFN
i	d
j	b
k	a

TLB

VPN	PFN
i	d
j	b
k	a

TLB caches the part of the process 1's page table

#1: TLB Entry

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Virtual Page Number (VPN)

Physical Frame Number (PFN)

- VPN & PFN – virtual and physical pages

Process 2's PA shouldn't be
the same with Process'1 A
– PFN in TLB is stale when
there is a context switch ☹

i	d
j	b
k	a

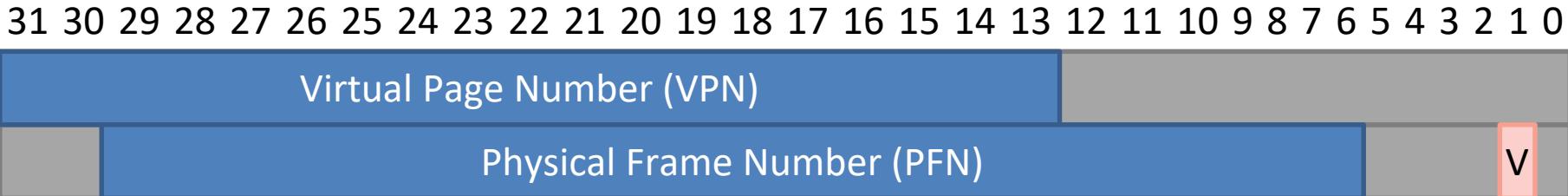
TLB

VPN	PFN
i	d
j	b
k	a

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

#1: TLB Entry



- V – valid bit – is this entry in the TLB valid?

Possible solution #1: clear the TLB for each context switch (mark all entries as invalid)

i	d
j	b
k	a

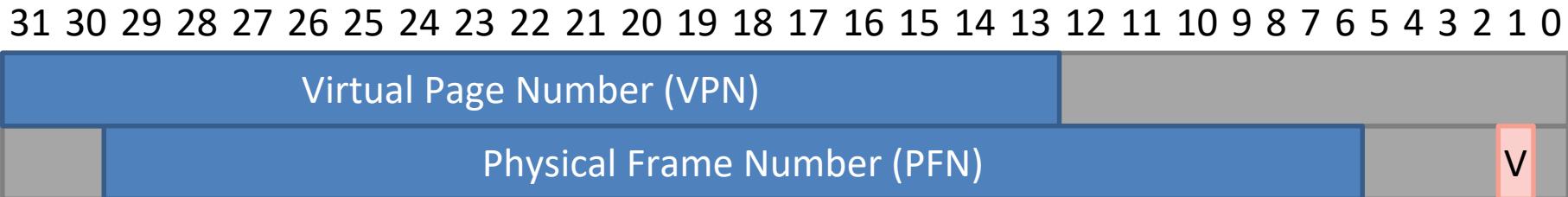
TLB

VPN	PFN

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

#1: TLB Entry



- V – valid bit – is this entry in the TLB valid?

It works, but very inefficient. Clearing TLB makes all processes to start with cold cache ☹

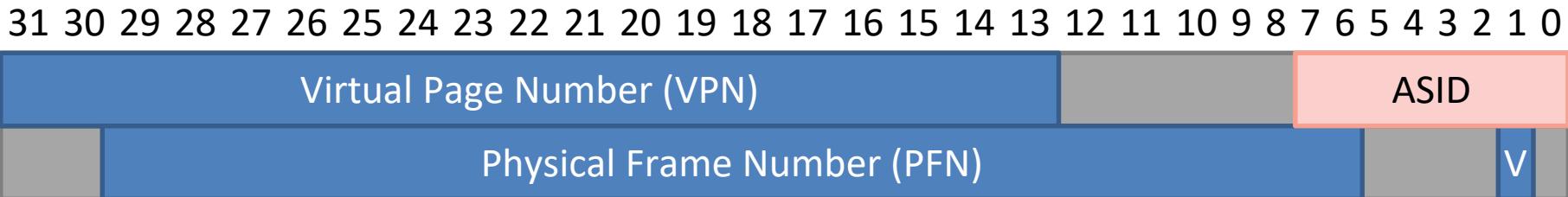
j	b
k	a

TLB	VPN	PFN

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

#1: TLB Entry



- ASID – address space ID (= process ID in the kernel)

Possible solution #2: Associate an ASID with each process → CPU can compare the ASID of the active (currently working) process to that stored in each TLB entry

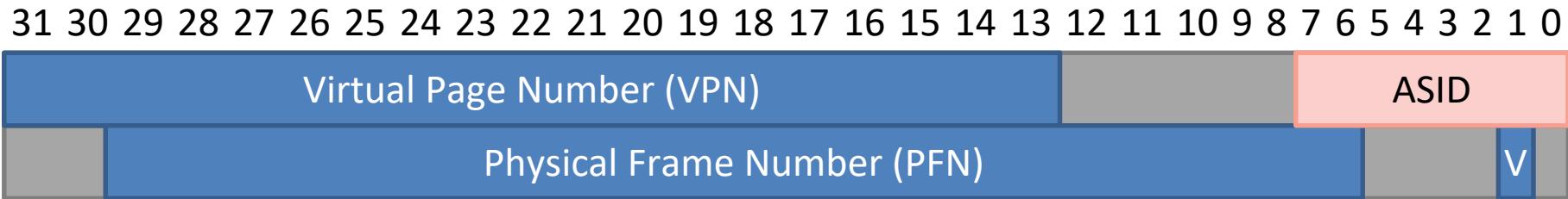
TLB

VPN	PFN	ASID
i	d	1
j	b	2
k	a	2

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

#1: TLB Entry



- ASID – address space ID (= process ID in the kernel)

If ASID doesn't match with active process ones, then clear that TLB entry (making it invalid)

TLB

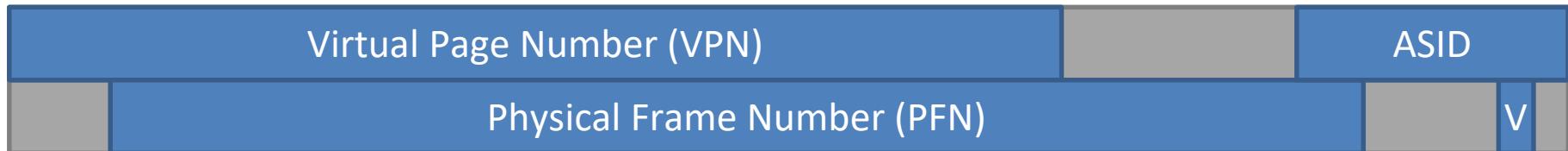
VPN	PFN	ASID
j	b	2
k	a	2

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

#1: TLB Entry

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



What if a page is shared by one or more processes?



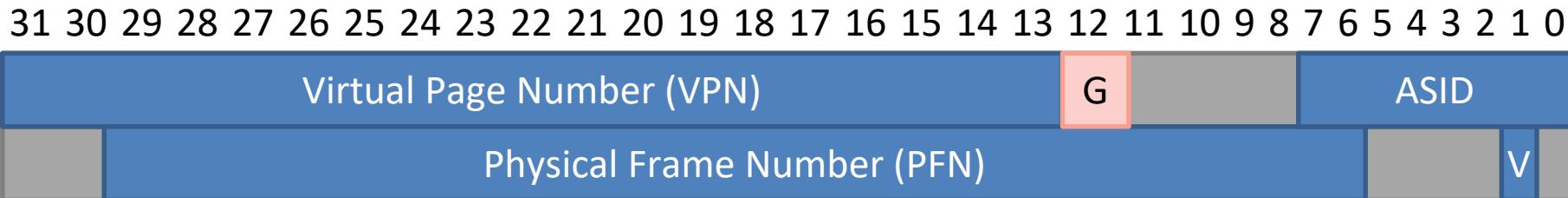
TLB

VPN	PFN	ASID
i	d	1
j	b	2
k	a	2

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

#1: TLB Entry



- G – is this page global (i.e., accessible by all processes)?

If the G flag is set, then
ASID is ignored ☺

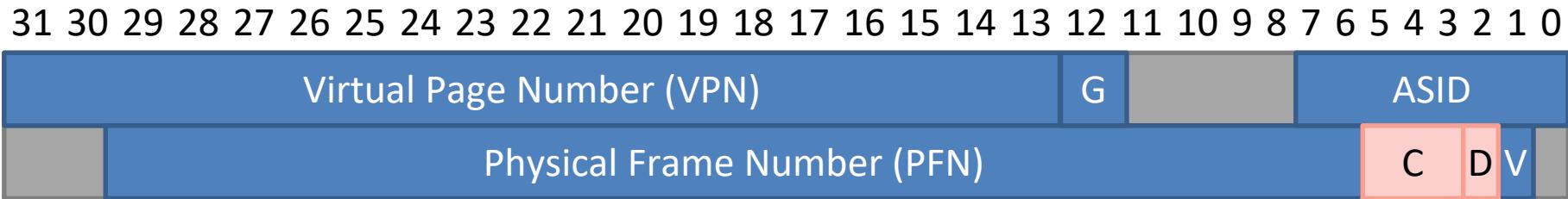
TLB

VPN	PFN	ASID
i	d	1
j	b	2
k	a	2

Process 2 Page Table

VPN	PFN
i	d
j	u
k	s

#1: TLB Entry

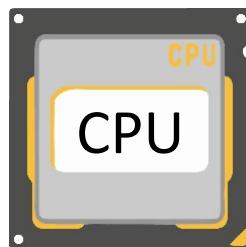


- C – cache coherency bits – for multi-core systems
- D – dirty bit – has this page been written recently?

TLB is also a kind of cache. Thus, it should synchronize the data of other cores. If there is an update of my side, I have to let other know about this new data.



#2: TLB Management



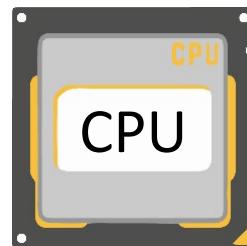
HW MMU

By far, our discussion has focused on
hardware managed TLBs (e.g., x86)
→ CPU manages all TLB entries





#2: TLB Management



HW MMU



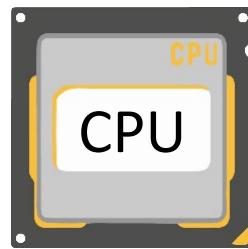
SW MMU

However, software managed
TLBs are also possible
(e.g., MIPS and SPARC)





#2: TLB Management



HW MMU

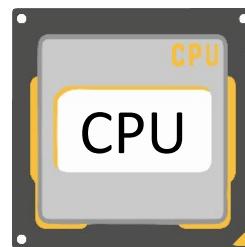


SW MMU

Note that, TLB management instructions
are working w/ **privilege mode**
→ Only the kernel can modify the TLB
(user program cannot)



#2: TLB Management



HW MMU



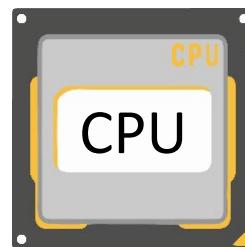
SW MMU

Advantages	<p>Less work for kernel developers, CPU does a lot of work for you</p>	<p>No predefined data structure for the page table</p>
------------	--	--

Easier to program

Greater flexibility

#2: TLB Management



HW MMU



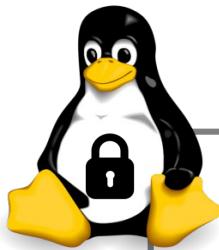
SW MMU

Advantages	<p>Less work for kernel developers, CPU does a lot of work for you</p>	<p>No predefined data structure for the page table</p>
Disadvantages	<p>Kernel data structure should be aligned with what HW specification intends to</p>	<p>More work for kernel developers</p>

e.g.,) Developers need to control all extra things on
address translation such as exception handling,
address splitting, sanity checks, etc.



#2: TLB Management

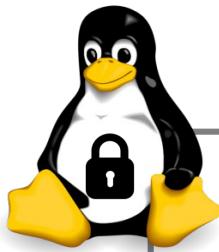


SW MMU

Kernel parses VPN

1. $\text{VPN} = (\text{VirtualAddress} \ \& \ \text{VPN_MASK}) \gg \text{SHIFT}$
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.
- 11.

#2: TLB Management

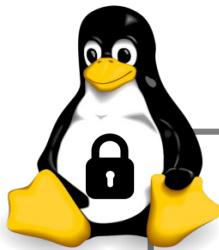


SW MMU

Kernel looks up TLB and
returns success or not

1. $\text{VPN} = (\text{VirtualAddress} \& \text{VPN_MASK}) \gg \text{SHIFT}$
2. $(\text{Success}, \text{TlbEntry}) = \text{TLB_Lookup}(\text{VPN})$
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.
- 11.

#2: TLB Management

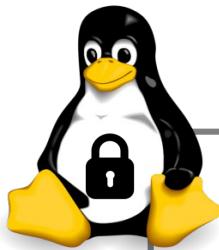


SW MMU

1. `VPN = (VirtualAddress &`
2. `(Success, TlbEntry) = TLB_Lookup(VPN)`
3. `if (Success == True) // TLB Hit`
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
10. `else // TLB Miss`
- 11.

If success → TLB hit
If fail → TLB miss

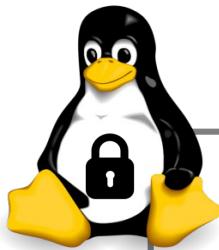
#2: TLB Management



SW MMU

1. $\text{VPN} = (\text{VirtualAddress} \ \& \ \text{VPN_MASK}) \gg \text{SHIFT}$
2. $(\text{Success}, \ \text{TlbEntry}) = \text{TLB_Lookup}(\text{VPN})$
3. if ($\text{Success} == \text{True}$) // TLB Hit
- 4.
5. On TLB miss, the OS must handle the exception
6. 1) Locate the correct page table entry in memory
7. 2) Insert the PTE into the TLB (evict if necessary)
8. 3) Tell the CPU to retry the previous instruction
- 9.
10. else // TLB Miss
11. RaiseException(TLB_MISS)

#2: TLB Management

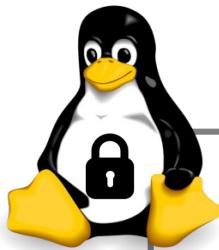


SW MMU

1. VPN = (VirtualAddress)
2. (Success, TlbEntry) = TLB_Lookup(VPN)
3. if (Success == True) // TLB Hit
4. if (CanAccess(TlbEntry.ProtectBits) == True)
- 5.
- 6.
- 7.
8. else
- 9.
10. else // TLB Miss
11. RaiseException(TLB_MISS)

If TLB hits, check the protect bits
whether this process can access
the PFN or not

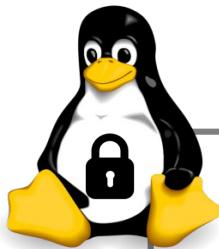
#2: TLB Management



SW MMU

1. `VPN = (VirtualAddress & VPN_MASK) >> SHIFT`
2. `(Success, TlbEntry) = TLB_Lookup(VPN)`
3. `if (Success == True) // TLB Hit`
4. `if (CanAccess(TlbEntry.ProtectBits) == True)`
5. `...`
6. `If the PFN is prohibited, raise the PROTECTION_FAULT exception`
7. `...`
8. `else`
9. `RaiseException(PROTECTION_FAULT)`
10. `else // TLB Miss`
11. `RaiseException(TLB_MISS)`

#2: TLB Management



SW MMU

```
1. VPN = (VirtualAddress  
2. (Success, TlbEntry) =  
3. if (Success == True) // TLB hit  
4.   if (CanAccess(TlbEntry.ProtectBits) == True)  
5.     Offset = VirtualAddress & OFFSET_MASK  
6.     PhysAddr = (TlbEntry.PFN << SHIFT) | Offset  
7.     Register = AccessMemory(PhysAddr)  
8.   else  
9.     RaiseException(PROTECTION_FAULT)  
10. else // TLB Miss  
11. RaiseException(TLB_MISS)
```

If the PFN is allowed to access,
kernel calculates the PA

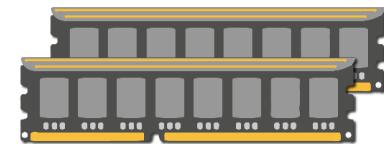
Optimization #2:

Managing page tables

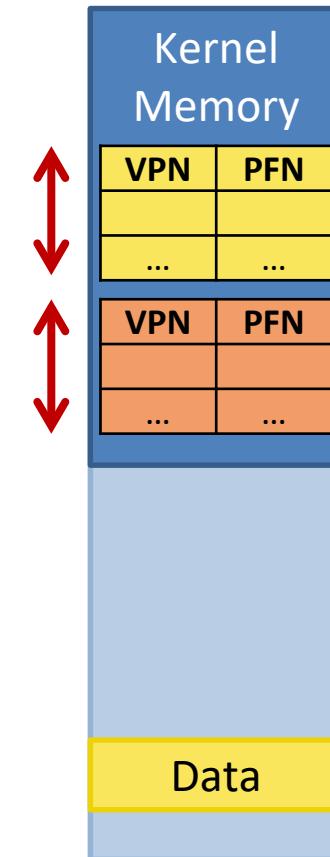
- Space optimization

Problem: Page Table Size

Physical memory



Recall that page tables are **large and sparse** (i.e., 4MB page table per process in 32-bit Linux system)

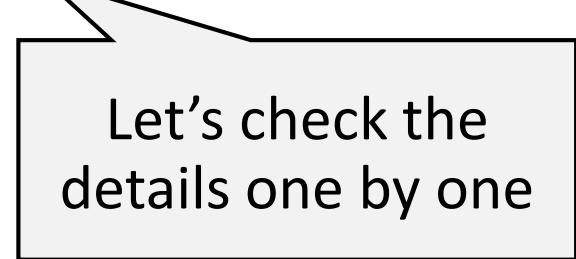


How can we reduce the size of the page tables?



Many Possible Solutions

- Solution #1: Huge pages
- Solution #2: Alternate data structures
- Solution #3: Inverted page table
- Solution #4: Multi-level page tables



Let's check the details one by one



Solution #1: Huge Pages

- Key idea: Choose larger page size (e.g., 8KB or even 4MB) to reduce the # of page table entries
- Example: 32-bit system, 4B of PTE size

4KB pages

- $2^{32}/2^{12} = 1,048,576$ pages per process
- $1,048,576 * 4B = 4MB$ page tables

4MB pages

- $2^{32}/2^{22} = 1,024$ pages per process
- $1024 * 4B = 4KB$ page tables



Solution #1: Huge Pages

- Key idea: Choose larger page size (even 4MB) to reduce the # of pages
- Example: 32-bit system, 4B of memory

Wait a second...
Could we use this solution
in a real scenario?



4KB pages

- $2^{32}/2^{12} = 1,048,576$ pages per process
- $1,048,576 * 4B = 4MB$ page tables

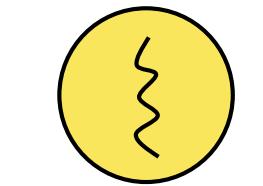
4MB pages

- $2^{32}/2^{22} = 1,024$ pages per process
- $1024 * 4B = 4KB$ page tables

Drawback of Huge Page

- Again, there is **internal fragmentation** issue, like segmentation-based virtual memory

Process A



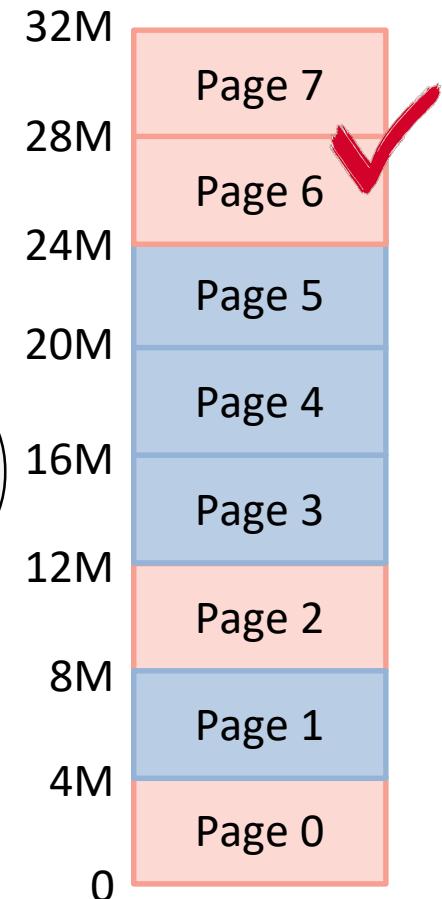
Give me memory



Here is 4MB page

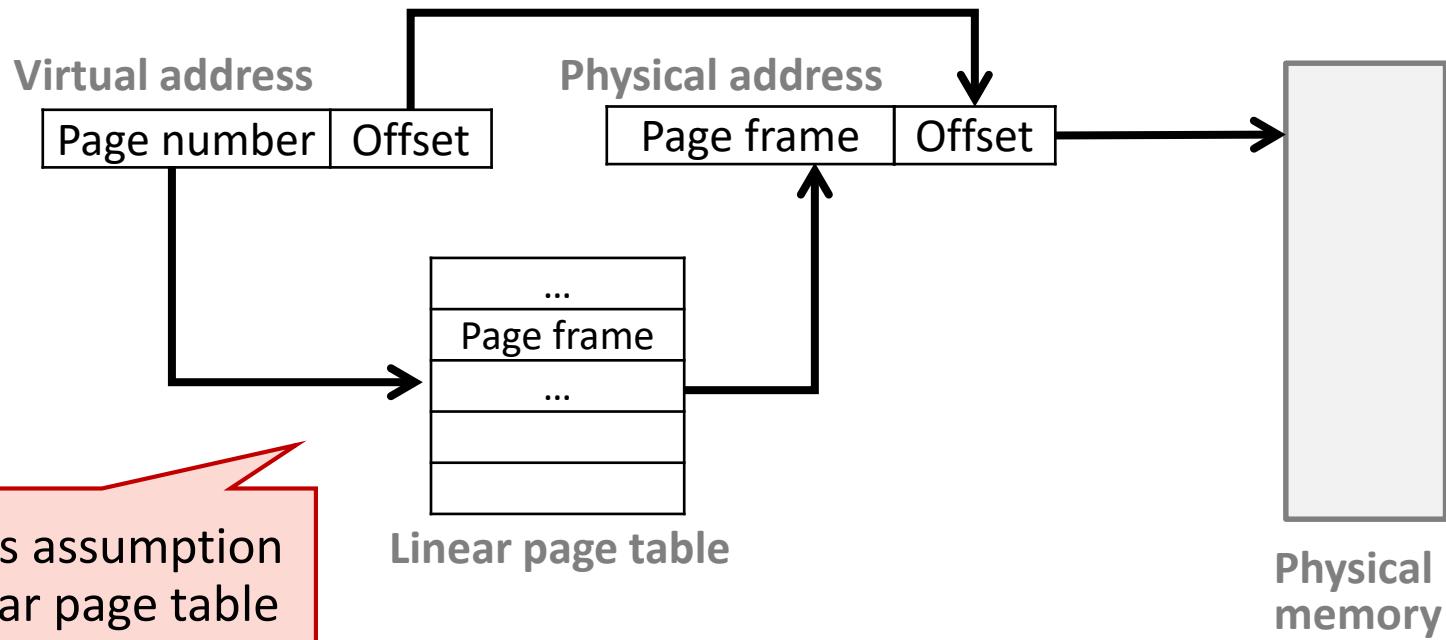
Page 6
(4MB)

How many programs actually need
4MB of code, 4MB of stack, and
4MB of heap data? – It's too **big** 😞



Solution #2: Alternate Data Structures

- Previous assumption: Thus far, we've assumed linear page tables (i.e., an array of page table entries).





Solution #2: Alternate Data Structures

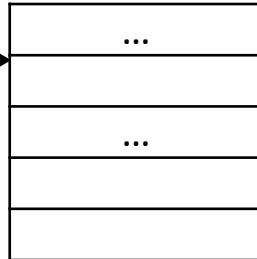
- **Key idea:** Let's switch to an space efficient data structure (e.g., hash table, rbtree)

Virtual address

Page number

Offset

Hash
function

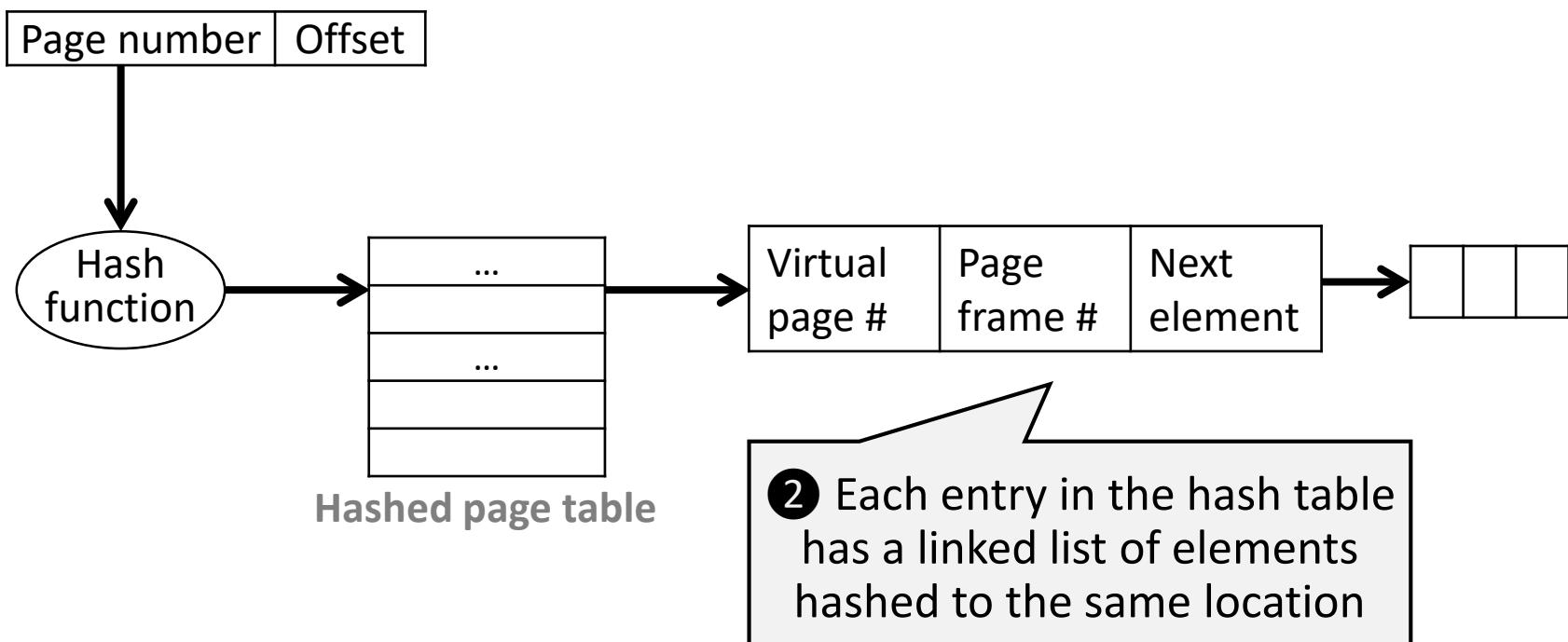


① Virtual page
number is hashed
into the hash table

Solution #2: Alternate Data Structures

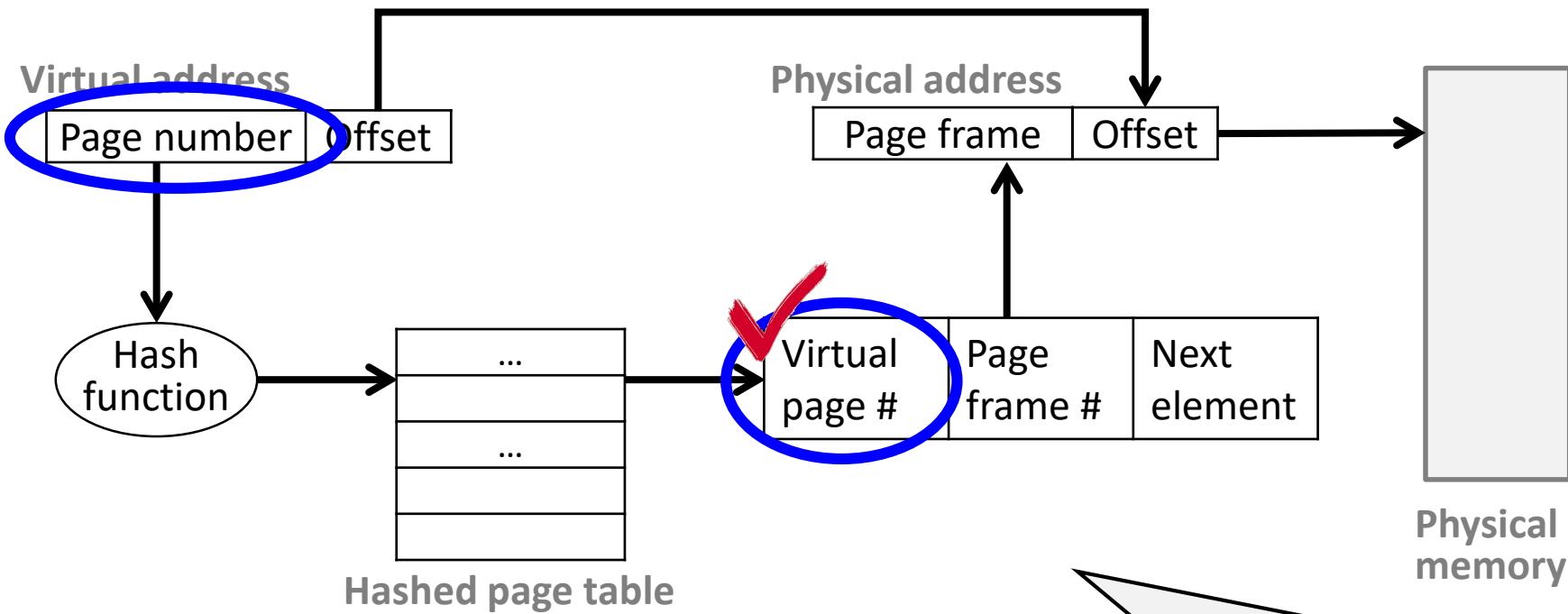
- **Key idea:** Let's switch to an space efficient data structure (e.g., hash table, rbtree)

Virtual address



Solution #2: Alternate Data Structures

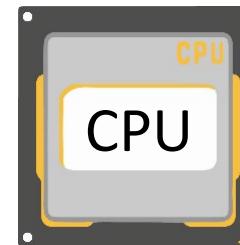
- Key idea: Let's switch to an space efficient data structure (e.g., hash table, rbtree)



③ The virtual page # is compared with field 1 in the linked list. If it matches, the corresponding page frame (field 2) is used to form the desired physical address

Limit of Switching Data Structures

- Switching data structures is not always feasible.
- Let's recall the TLB implementation methods



If the TLB is software managed (kernel), then we can easily switch the page table structure

If the TLB is hardware managed (CPU), then the OS must use the page table format specified by the CPU



Solution #3: Inverted Page Tables

- Previous assumption: our current discussion focuses on tables that map virtual pages to physical pages

Traditional page tables

Standard page tables:
one per process

VPN	PFN	
i	VPN	PFN
j	i	VPN
k	j	i
	k	j
	k	s

Solution #3: Inverted Page Tables

- Key idea: What if we flip the table: map physical pages to virtual pages?

Since there is only one physical memory, we only need one inverted page table 😊

Traditional page tables

VPN	PFN		
i	VPN		PFN
j	i	VPN	PFN
k	j	i	r
	k	j	u
	k		s

Inverted Table

PFN	VPN
i	d
j	b
k	a



Drawback of Inverted Page Table

- Lookups are more expensive
- Further, same virtual addresses can be associated with different processes. How can we distinguish them?

Traditional Table

VPN	PFN
i	d
j	b
k	a

index = k

VPN serves as an index into the array, thus O(1) lookup time

Inverted Table

PFN	VPN
d	i
b	j
a	k

index = k

Table must be scanned to locate a given VPN, thus O(n) lookup time



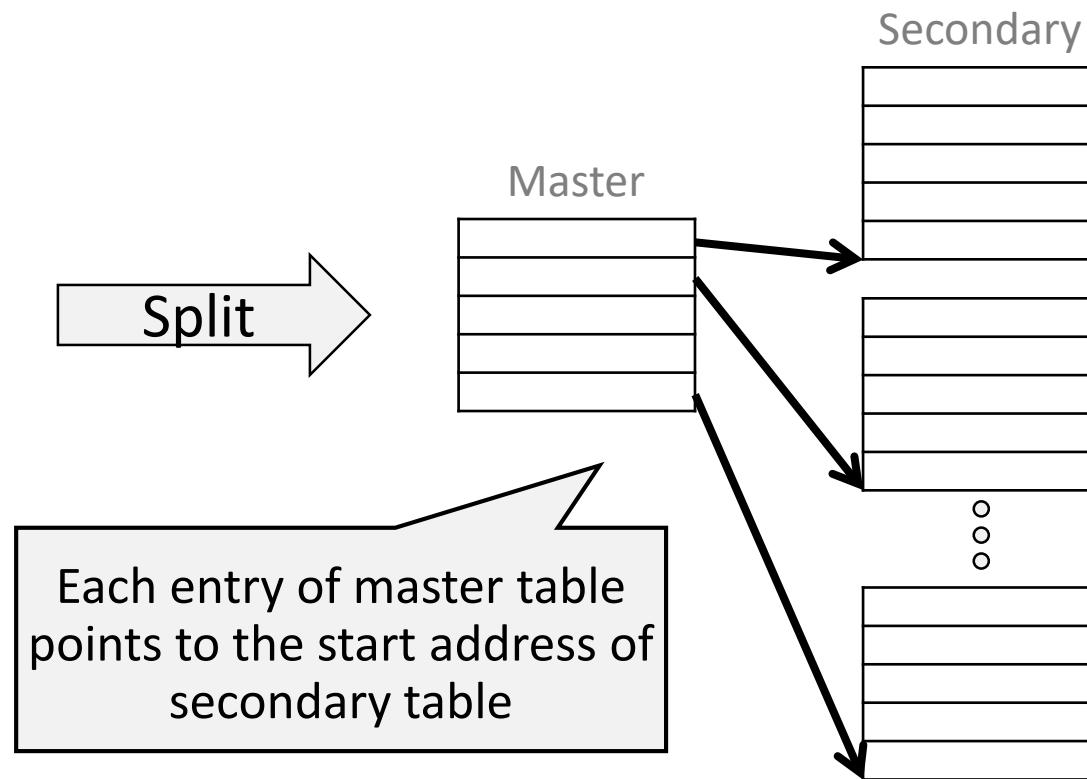
Solution #4: Multi-level Page Tables

- Key idea: split the linear page table into a tree of sub-tables

Linear (Flat) Page Table



Hierarchical (Multi-level) Page Table





Solution #4: Multi-level Page Tables

- Key idea: split the linear page table into a tree of sub-tables

Lin

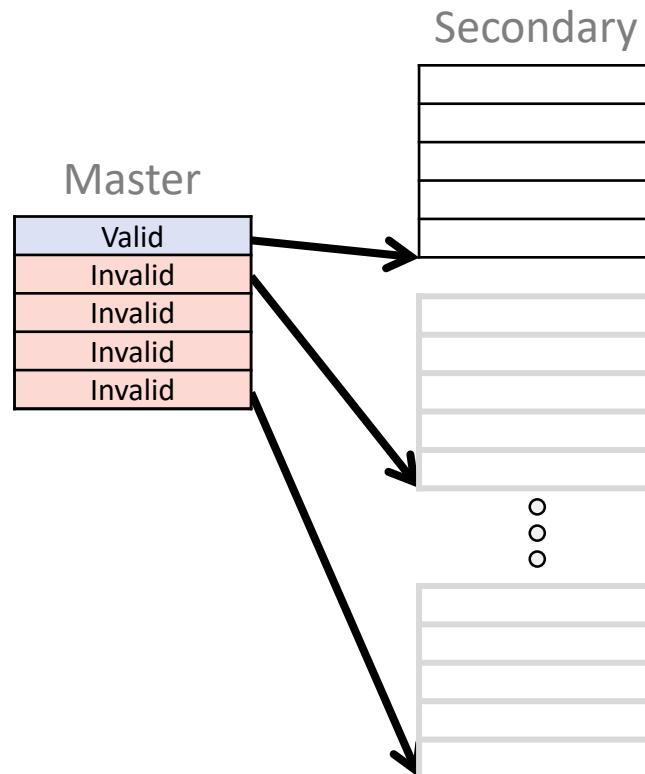
Benefit:

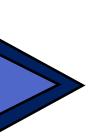
If there is no access, we don't need to allocate mapping information thereby saving the space! 😊



Split

Hierarchical (Multi-level) Page Table





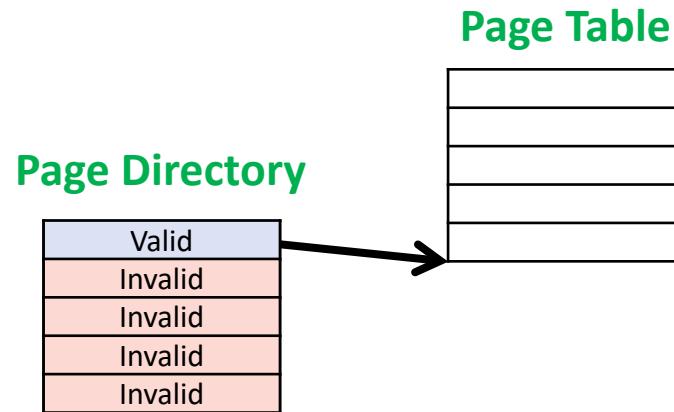
Solution #4: Multi-level Page Tables

- Key idea: split the linear page table into a tree of sub-tables

The master table and secondary table are as generally known as “Page Directory”, and “Page Table”, respectively



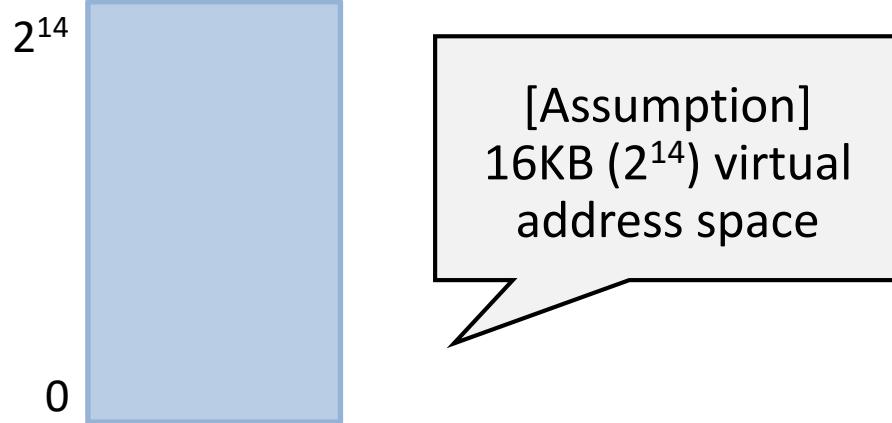
Hierarchical (Multi-level) Page Table





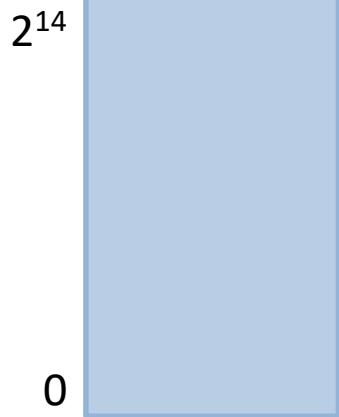
Toy Example

Process 1's view
of virtual memory



Toy Example

Process 1's view
of virtual memory



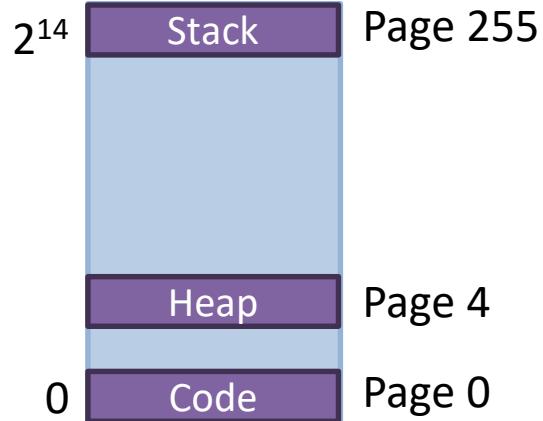
Page 255

Page 0

[Assumption]
64B (2^6) page size
→ 256 (2^8) pages

Toy Example

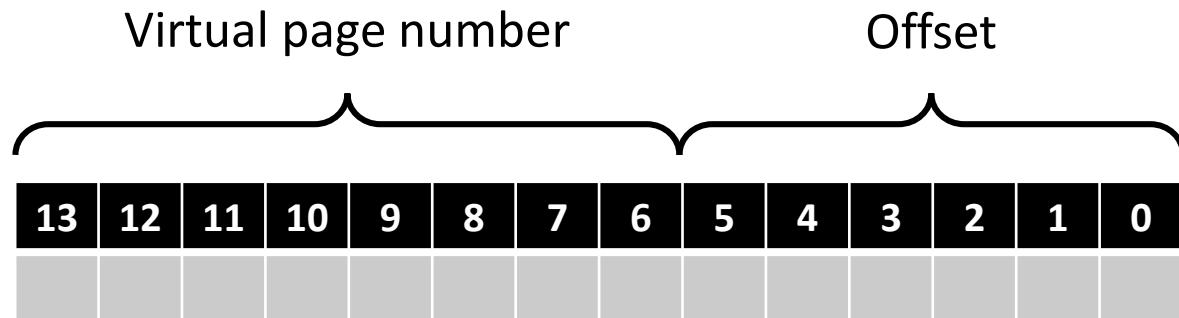
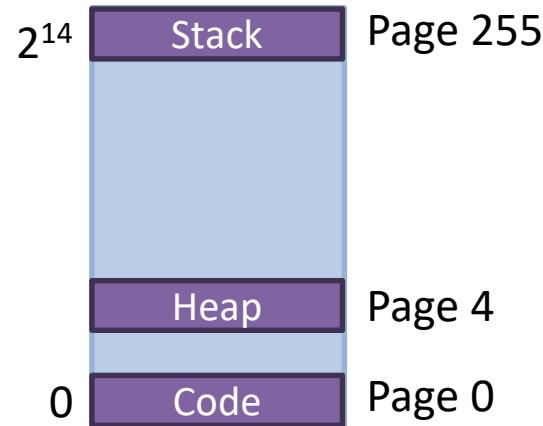
Process 1's view
of virtual memory



[Assumption]
3 pages out of 256
total pages are in use

Toy Example

Process 1's view
of virtual memory

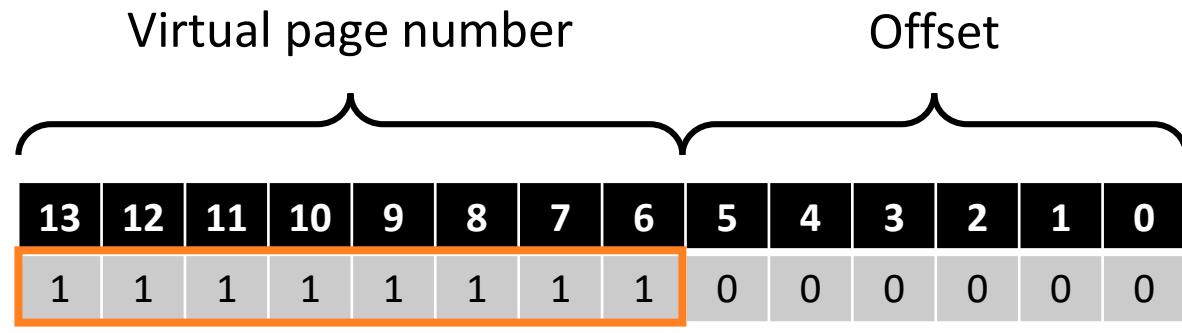
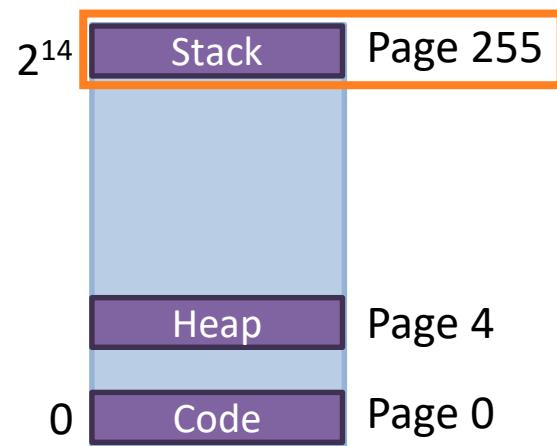


VPN	PFN	Valid?
00000000	a	1
...		0
00000100	b	1
...		0
11111111	c	1

Linear Page Table

Toy Example

Process 1's view
of virtual memory



Linear Page Table (LPT) showing the mapping between Virtual Page Number (VPN) and Physical Page Number (PFN).

VPN	PFN	Valid?
00000000	a	1
...		0
00000100	b	1
...		0
11111111	c	1

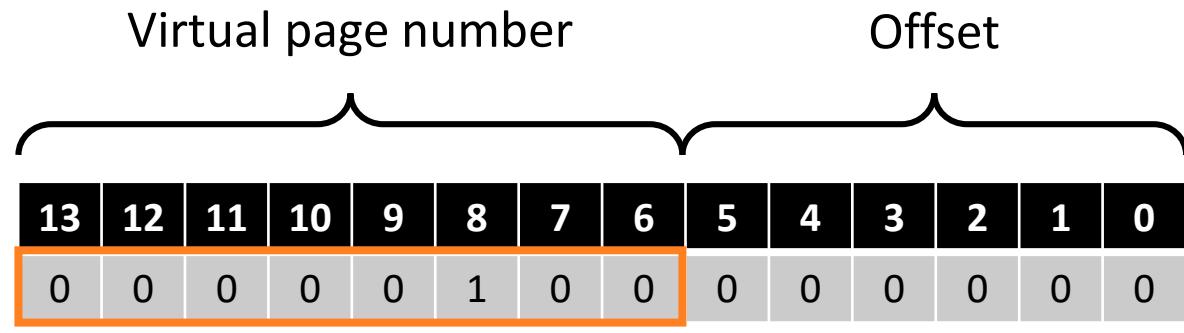
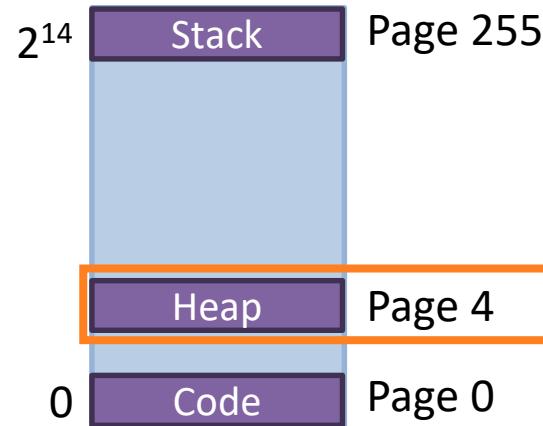
A callout box points from the text "PFN of Page 255 = c" to the row where VPN is 11111111 and PFN is c.

Linear Page Table



Toy Example

Process 1's view
of virtual memory



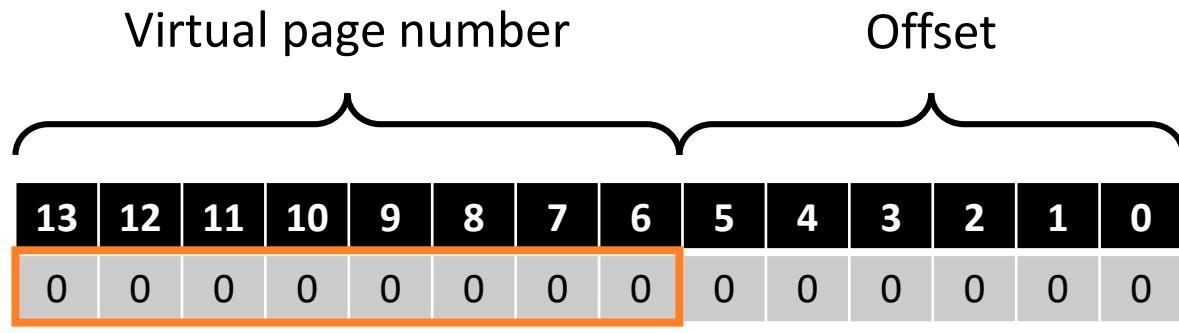
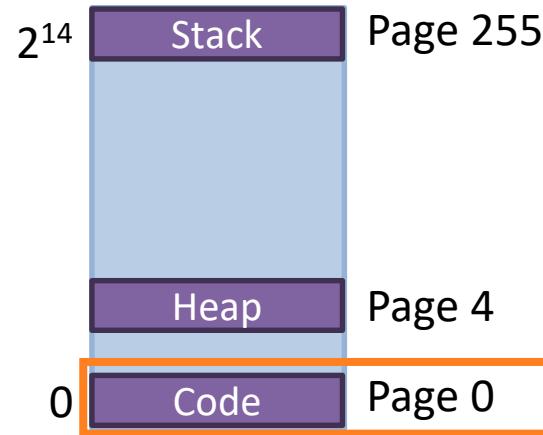
VPN
PFN of
Page 4 = b

VPN	PFN	Valid?
00000000	a	1
...		0
00000100	b	1
...		0
11111111	c	1

Linear Page Table

Toy Example

Process 1's view
of virtual memory



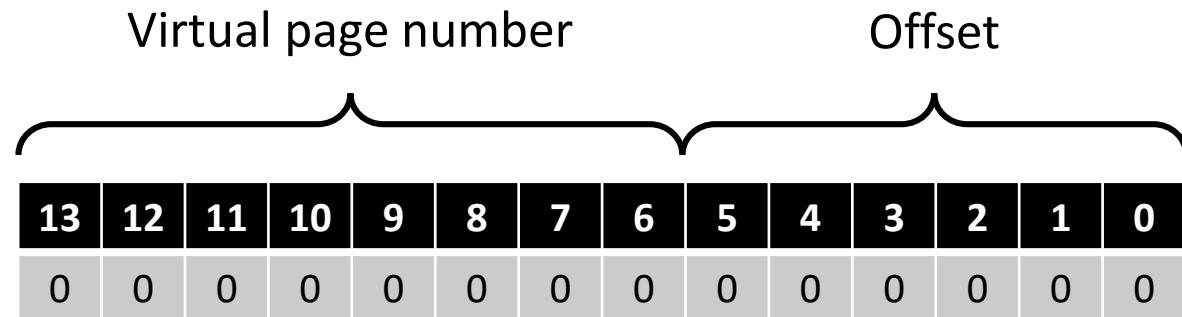
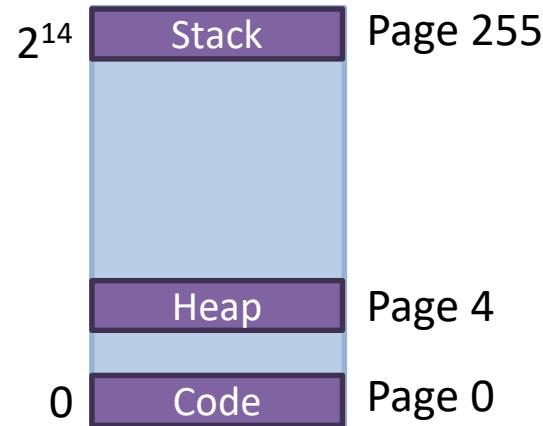
VPN of
Page 4 = a

VPN	PFN	Valid?
00000000	a	1
...		0
00000100	b	1
...		0
11111111	c	1

Linear Page Table

Toy Example

Process 1's view
of virtual memory



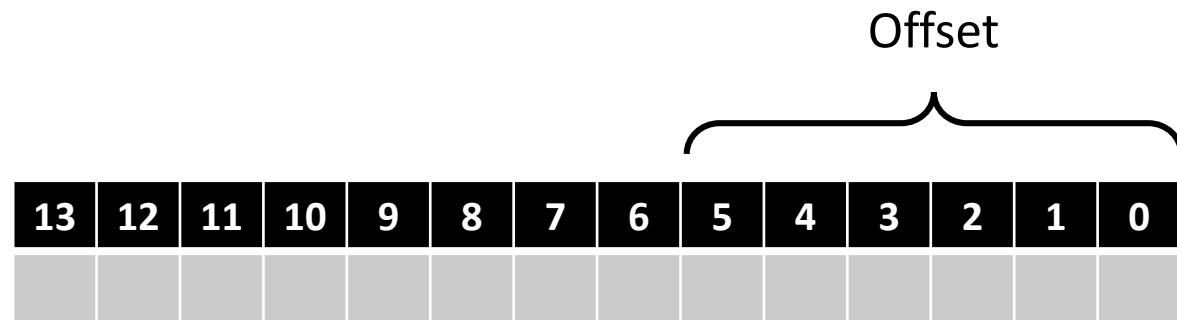
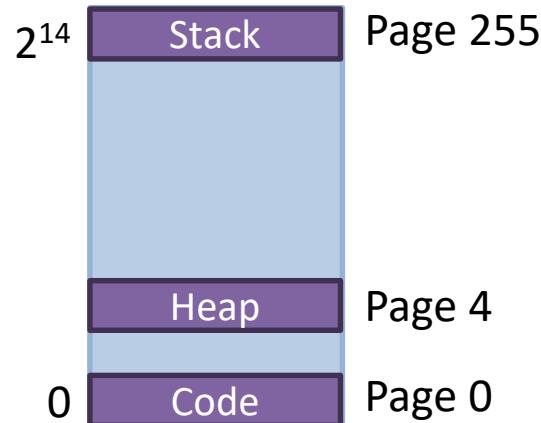
253 table entries stand with nothing, which wasted your memory a lot 😞

VPN	PFN	Valid?
00000000	a	1
...		0
00000100	b	1
...		0
11111111	c	1

Linear Page Table

Toy Example

Process 1's view
of virtual memory

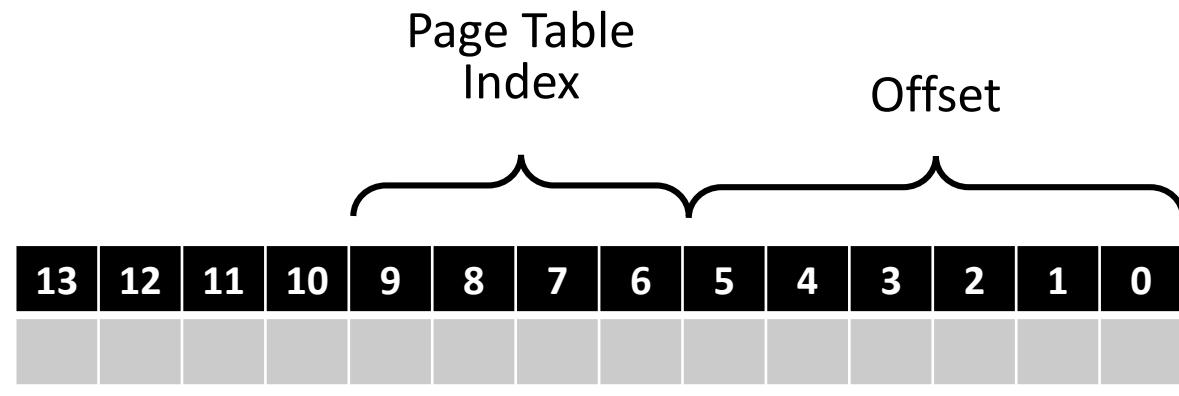
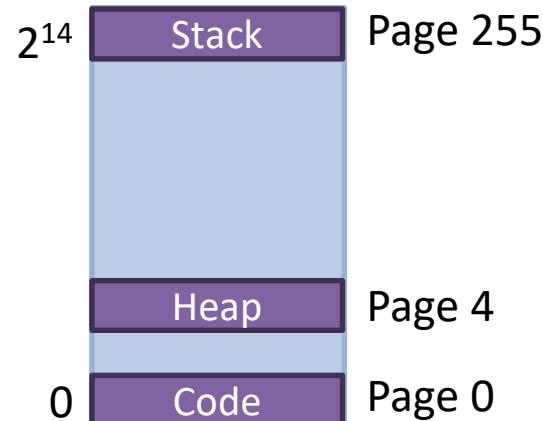


How can we change linear table walking logic to an address translator using a multi-level table?



Toy Example

Process 1's view
of virtual memory



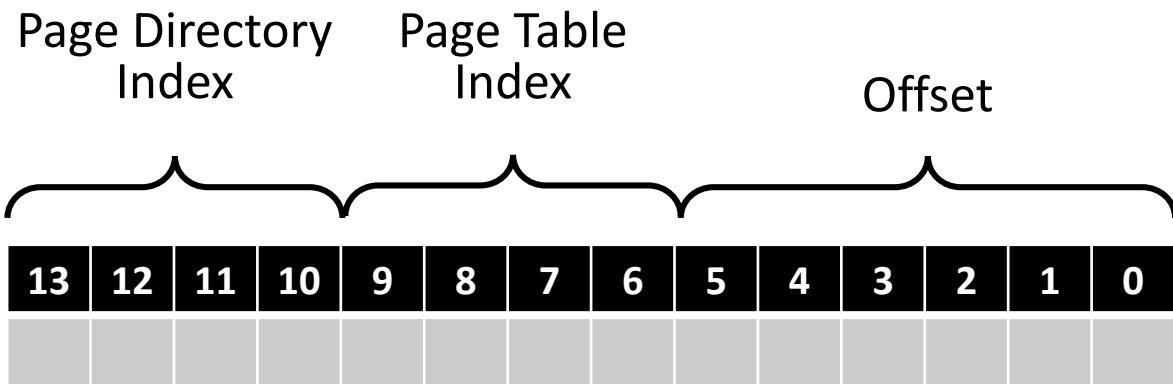
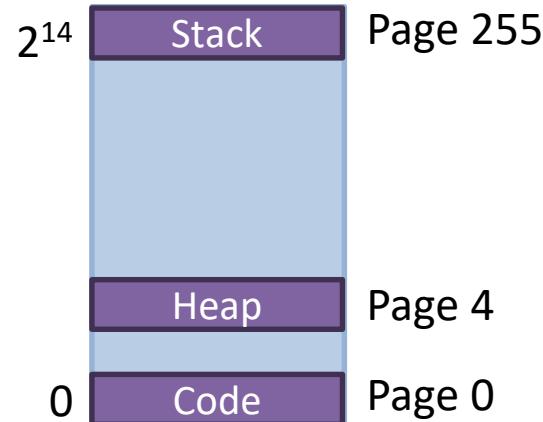
PTEs
= [page_size /entry_size](#) (e.g.,)
= 64 bytes (2^6) / 4 bytes (2^2)
= 16 entries (2^4)

Separate the “secondary table”
that sits on a linear memory space
into multiple page-sized mapping
chunks.



Toy Example

Process 1's view
of virtual memory



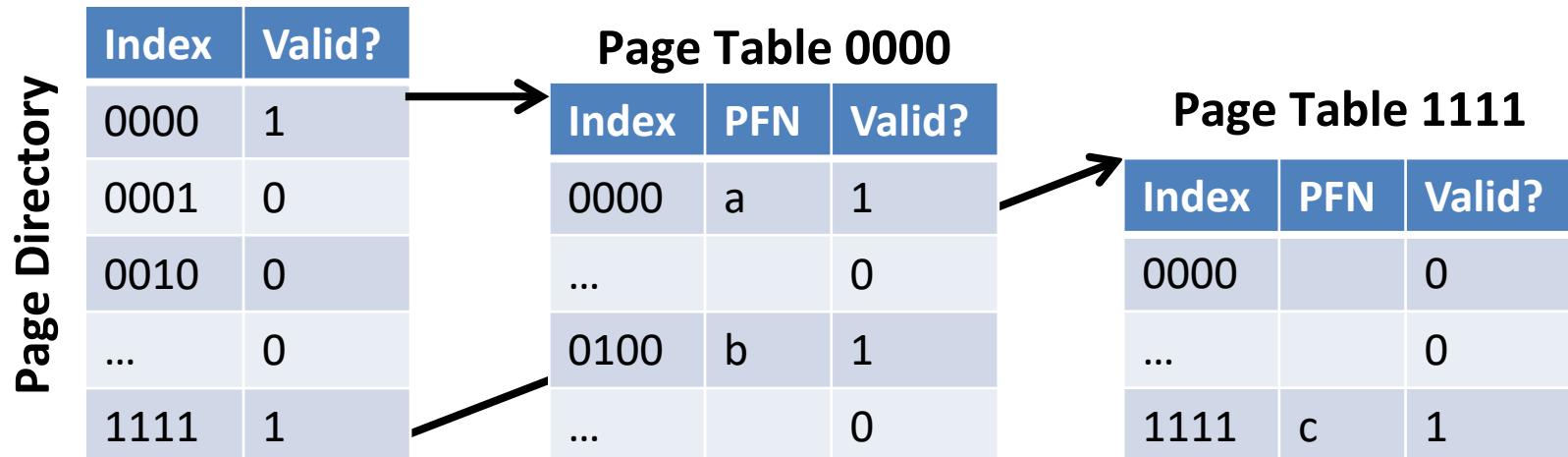
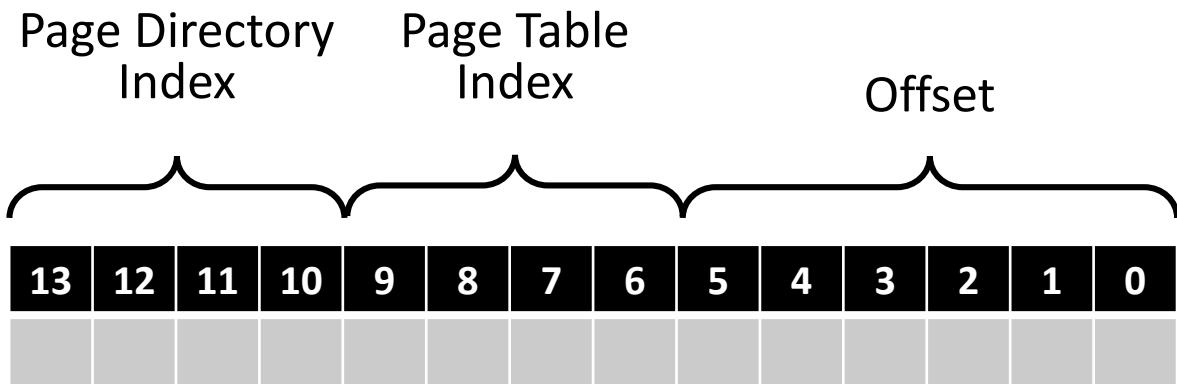
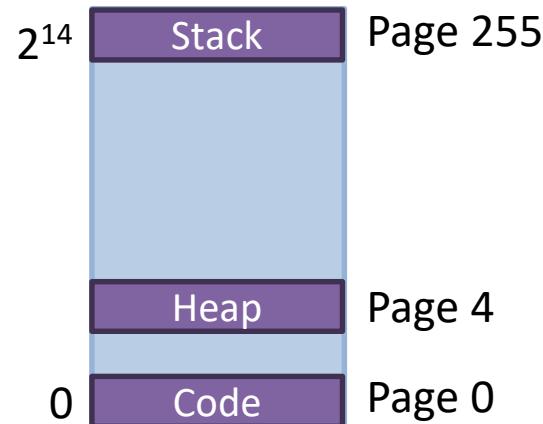
page tables
= `page_table_size / page_size` (e.g.,)
= 1KB (2^{10}) / 64 bytes (2^6)
= 16 page tables (2^4)



As the mapping information has been split, PD consolidates the locations of each secondary table's entry

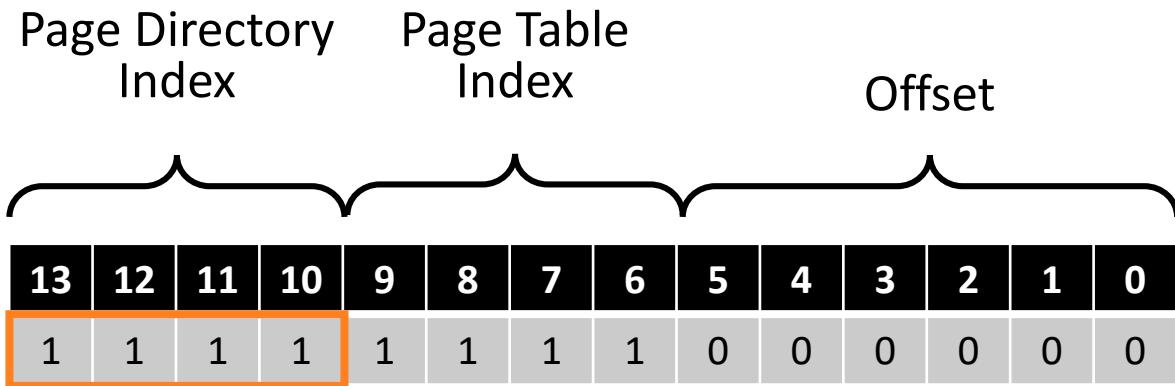
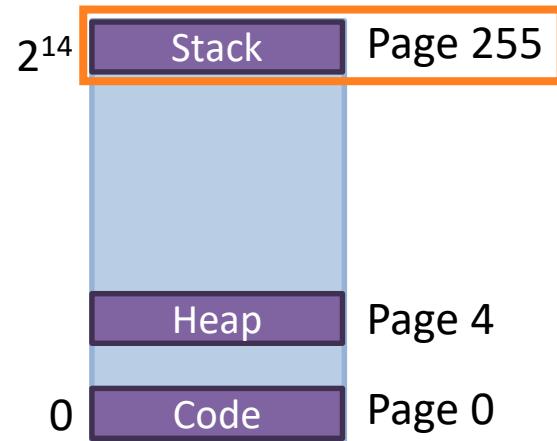
Toy Example

Process 1's view
of virtual memory



Toy Example

Process 1's view
of virtual memory



Index	Valid?
0000	1
0001	0
0010	0
...	0
1111	1

Page Table 0000

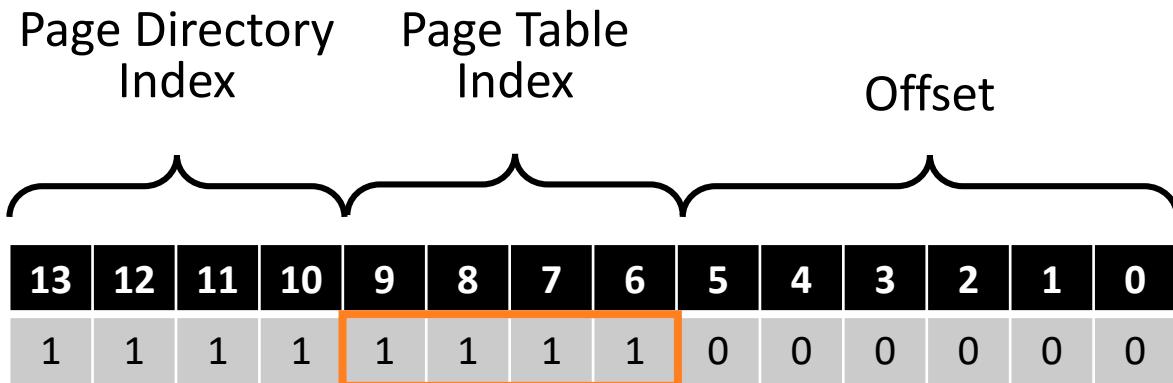
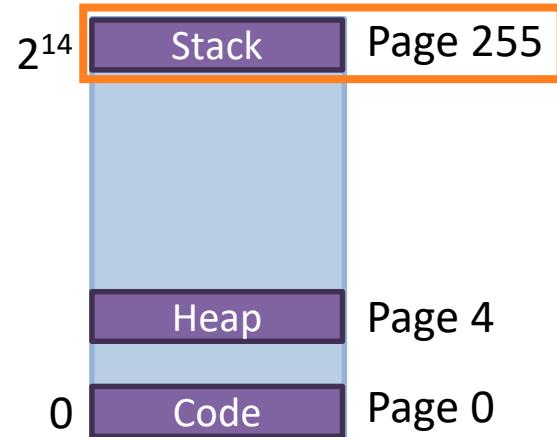
Index	PFN	Valid?
0000	a	1
...		0
0100	b	1
...		0

Page Table 1111

Index	PFN	Valid?
0000		0
...		0
1111	c	1

Toy Example

Process 1's view
of virtual memory



Index	Valid?
0000	1
0001	0
0010	0
...	0
1111	1

Page Table 0000

Index	PFN	Valid?
0000	a	1
...		0
0100	b	1
...		0

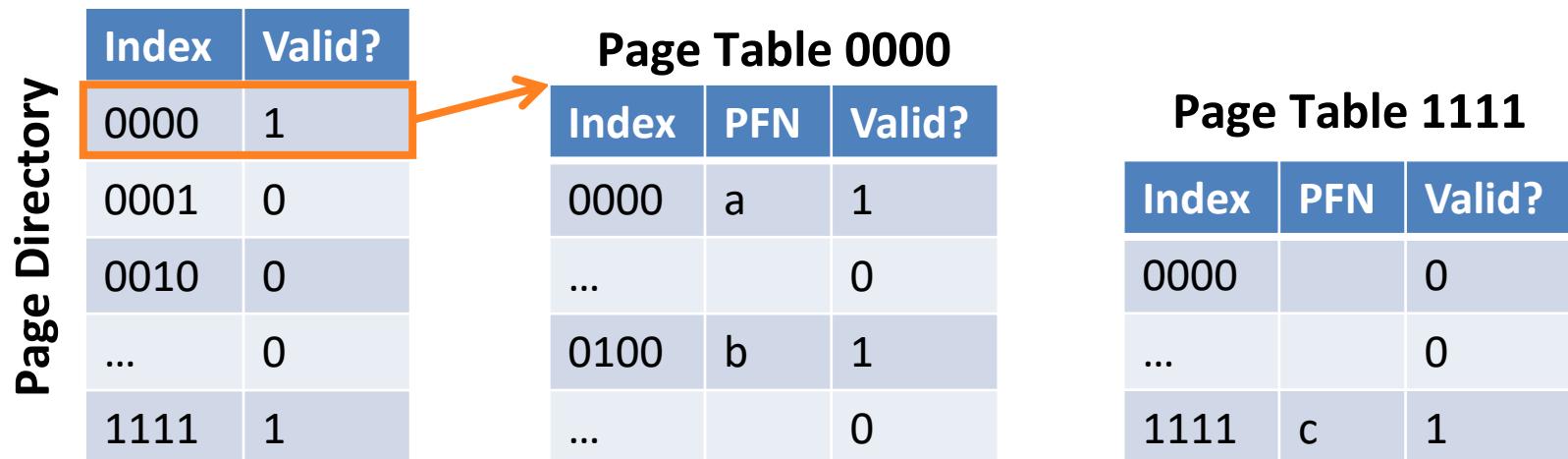
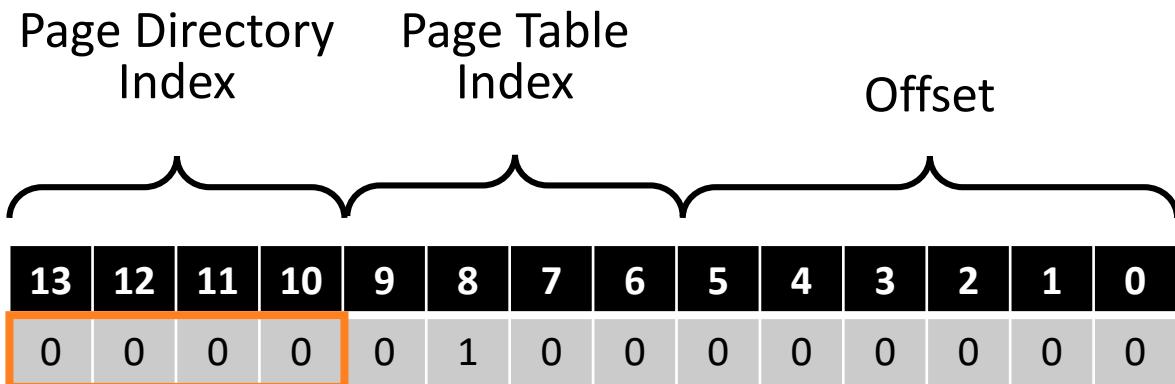
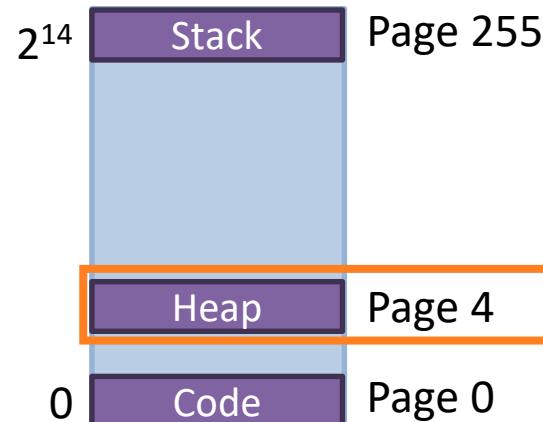
PFN of
Page 255 = c

Page Table 1111

Index	PFN	Valid?
0000		0
...		0
1111	c	1

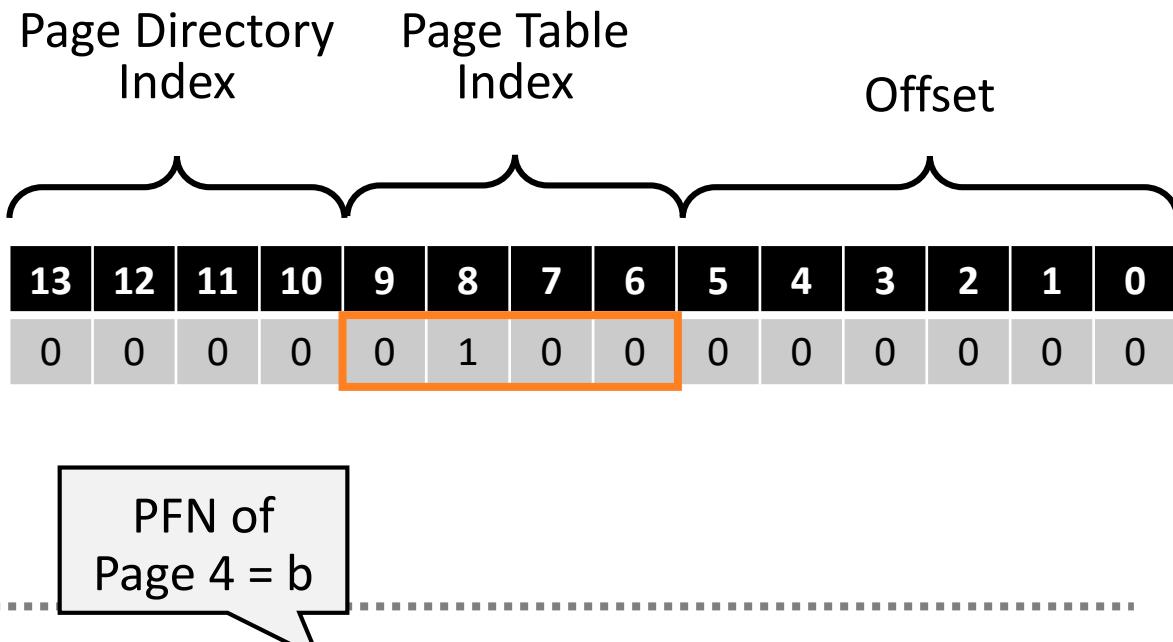
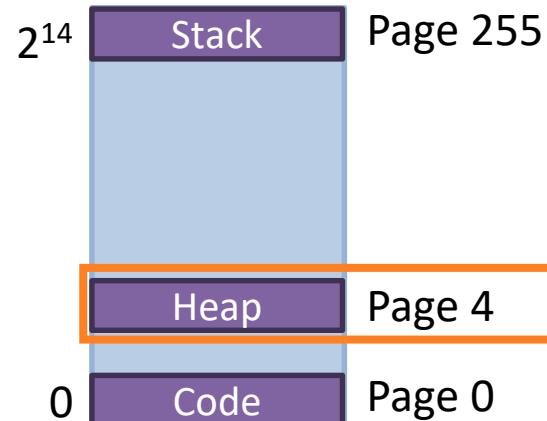
Toy Example

Process 1's view
of virtual memory



Toy Example

Process 1's view
of virtual memory



Index	Valid?
0000	1
0001	0
0010	0
...	0
1111	1

Page Table 0000

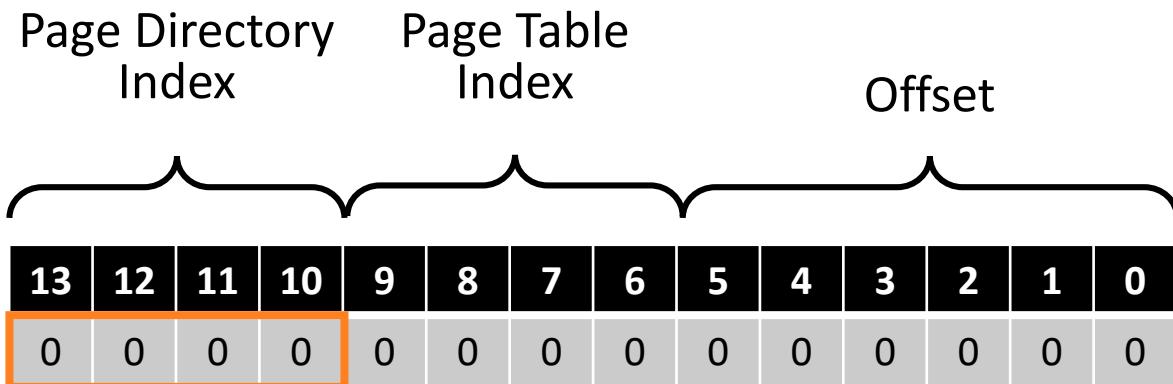
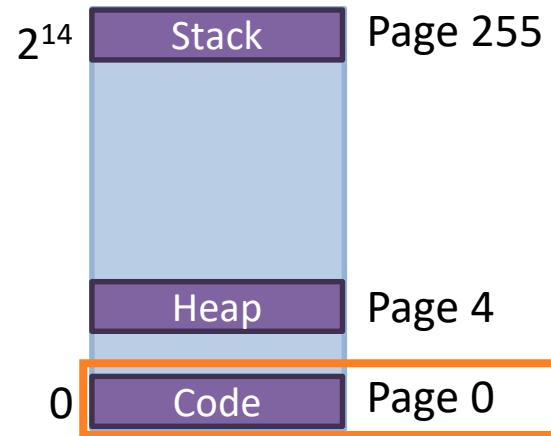
Index	PFN	Valid?
0000	a	1
...		0
0100	b	1
...		0

Page Table 1111

Index	PFN	Valid?
0000		0
...		0
1111	c	1

Toy Example

Process 1's view
of virtual memory



Index	Valid?
0000	1
0001	0
0010	0
...	0
1111	1

Page Table 0000

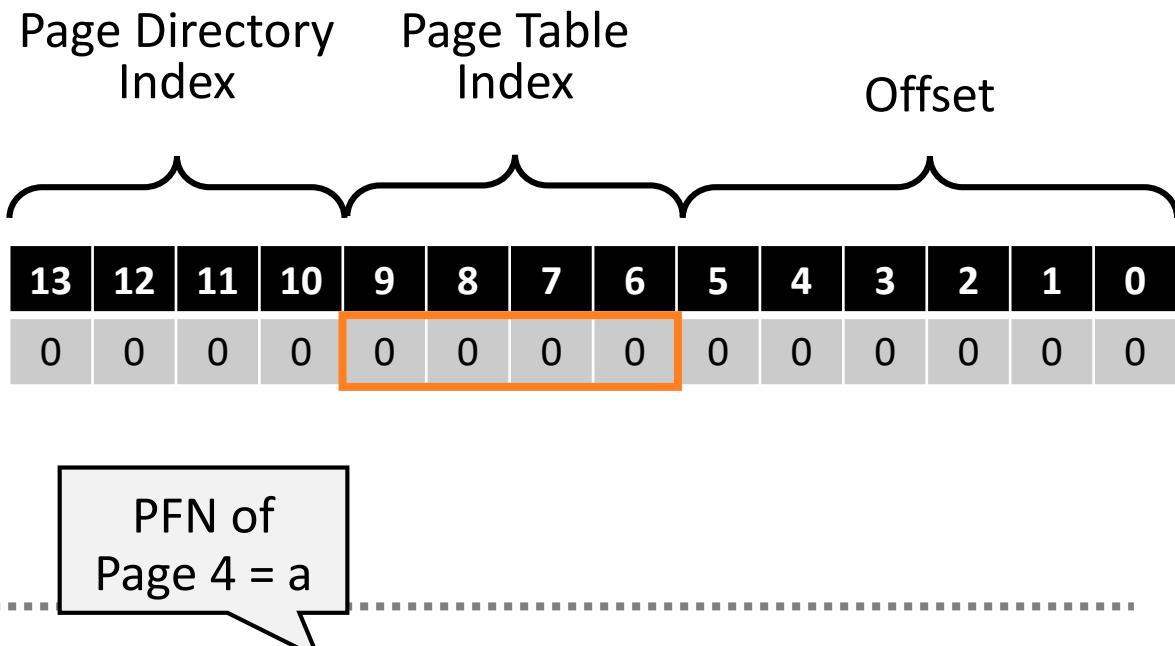
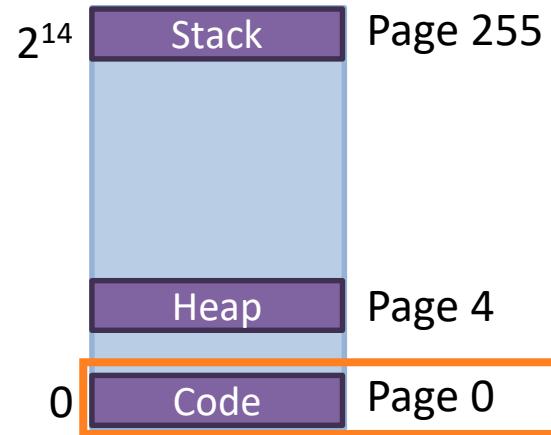
Index	PFN	Valid?
0000	a	1
...		0
0100	b	1
...		0

Page Table 1111

Index	PFN	Valid?
0000		0
...		0
1111	c	1

Toy Example

Process 1's view
of virtual memory



Page Directory

Index	Valid?
0000	1
0001	0
0010	0
...	0
1111	1

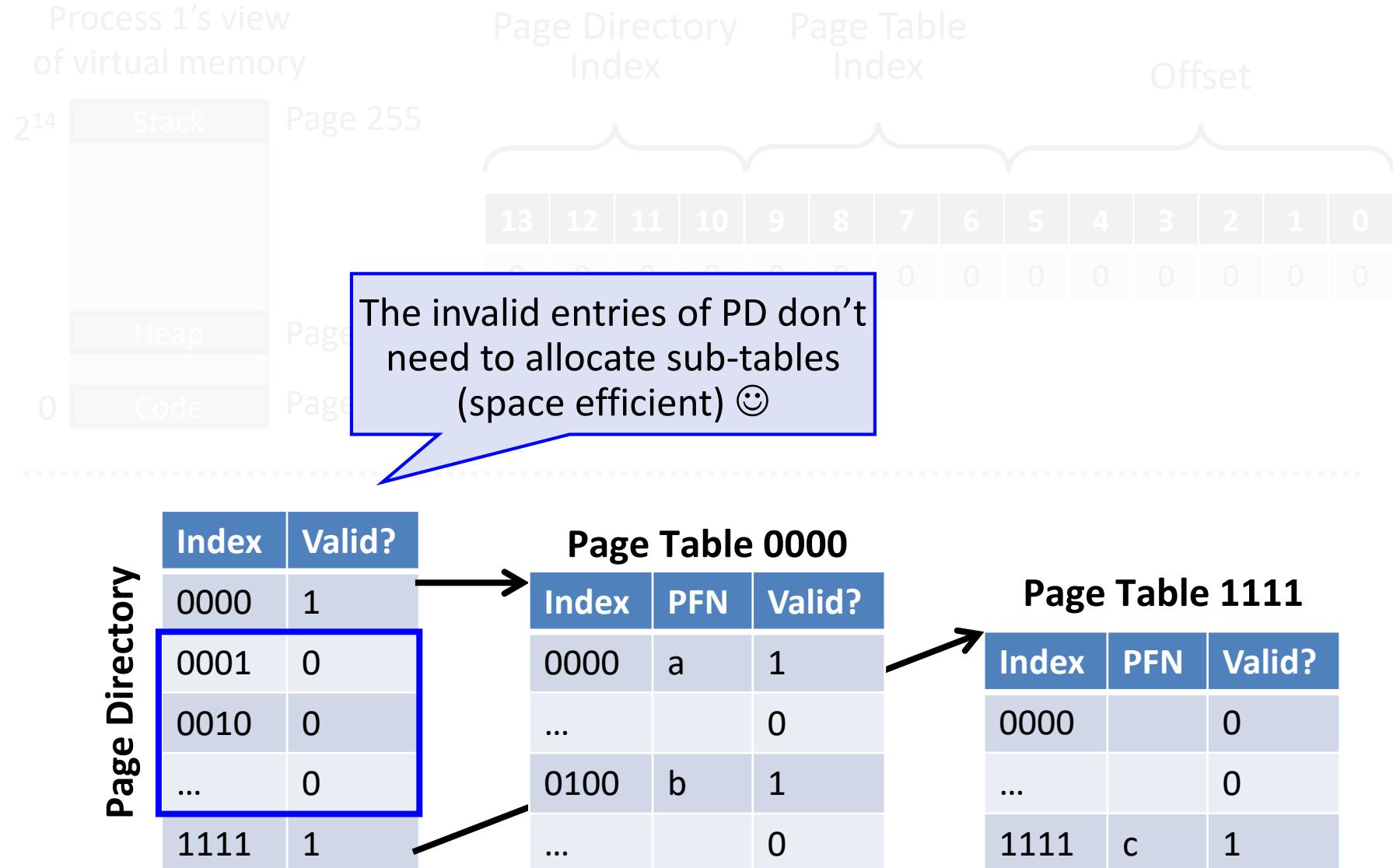
Page Table 0000

Index	PFN	Valid?
0000	a	1
...		0
0100	b	1
...		0

Page Table 1111

Index	PFN	Valid?
0000		0
...		0
1111	c	1

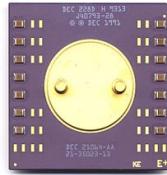
Toy Example





Most Common Solution

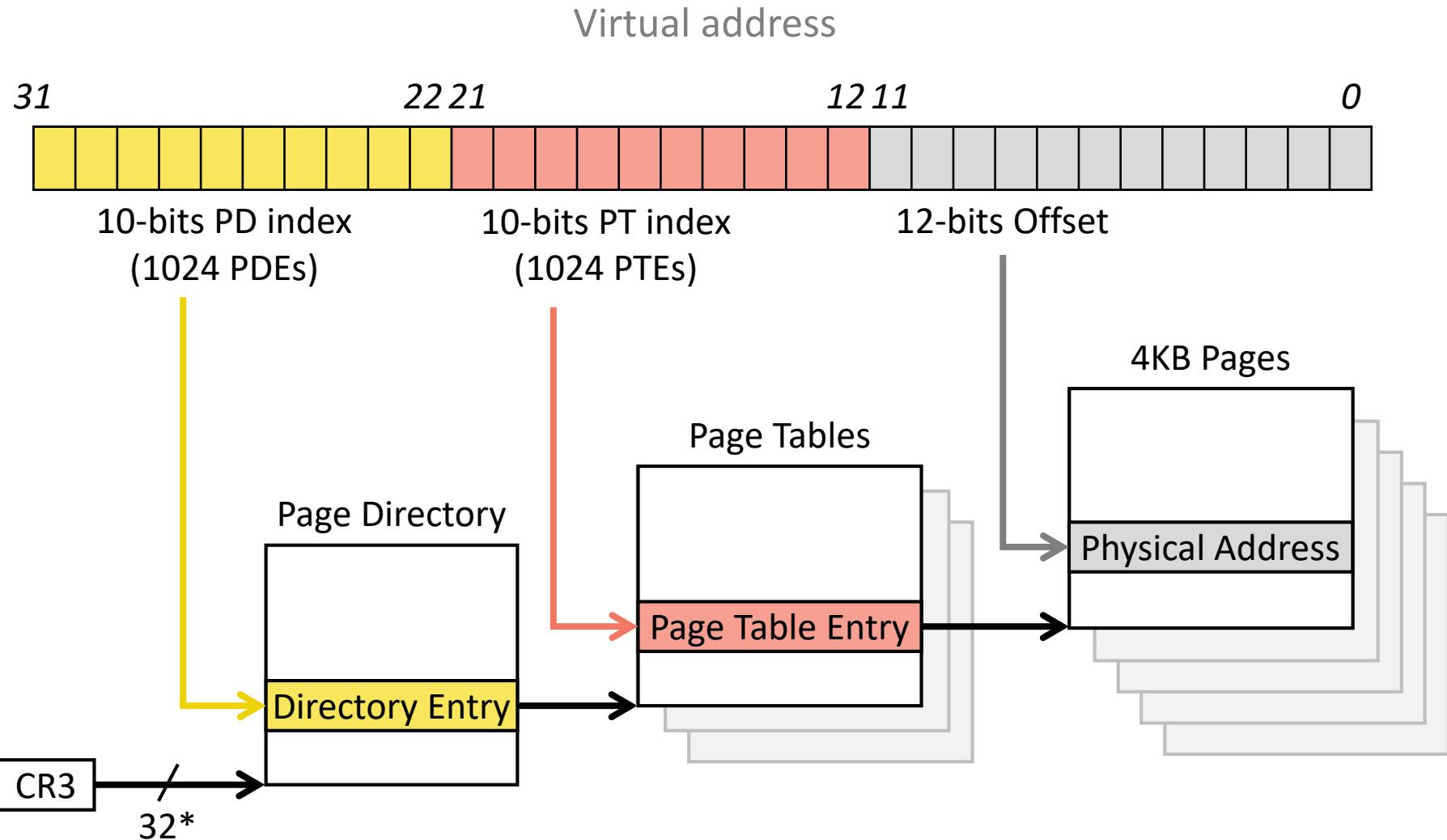
- Multi-level page table is commonly used in many systems



Platform	Page size	# of paging levels
alpha	8KB	3
ia64	4KB	3
ppc64	4KB	3
x86_32	4KB	2
x86_64	4KB	4

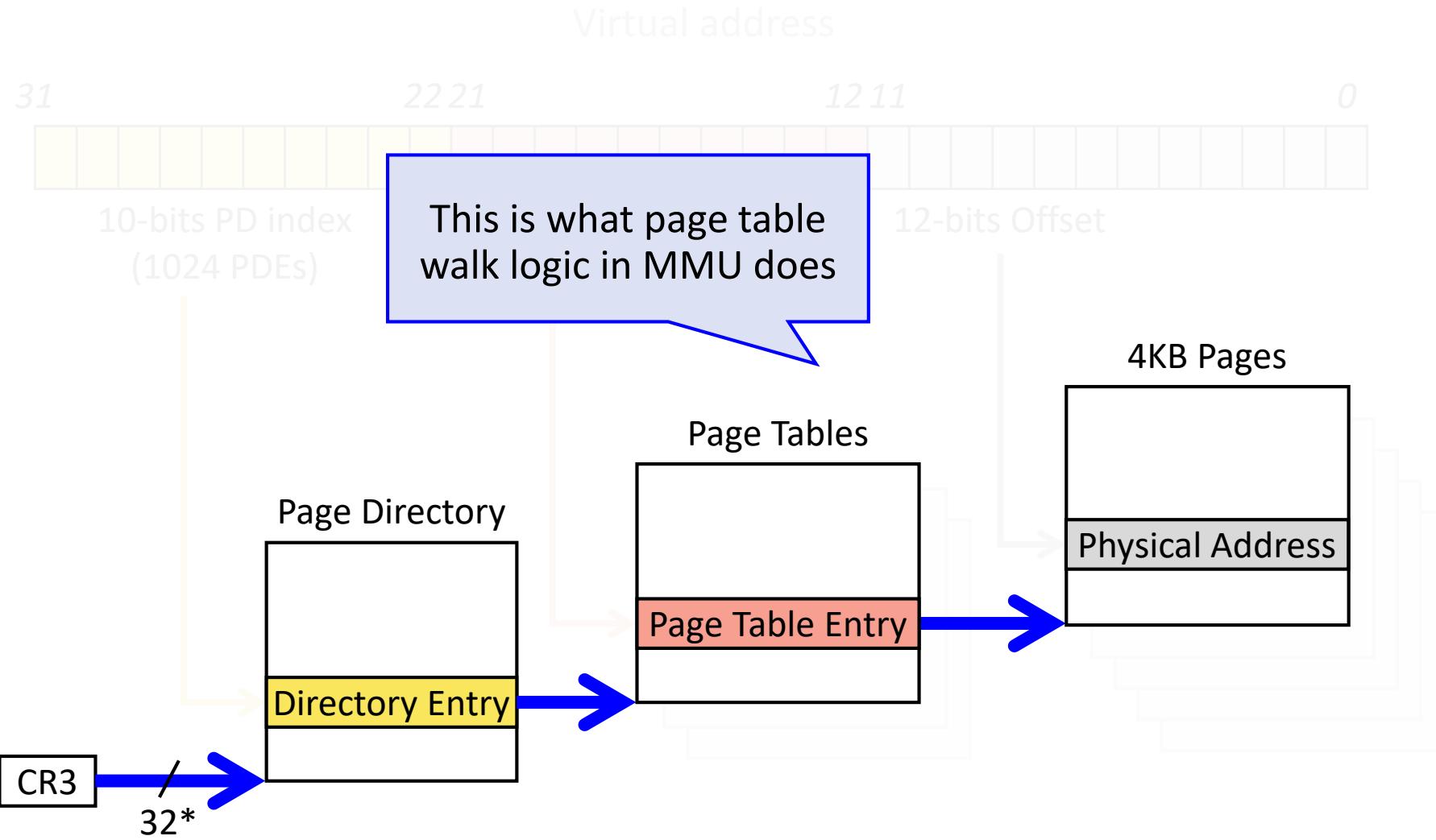


Example: 32-bit x86 (Two-level)



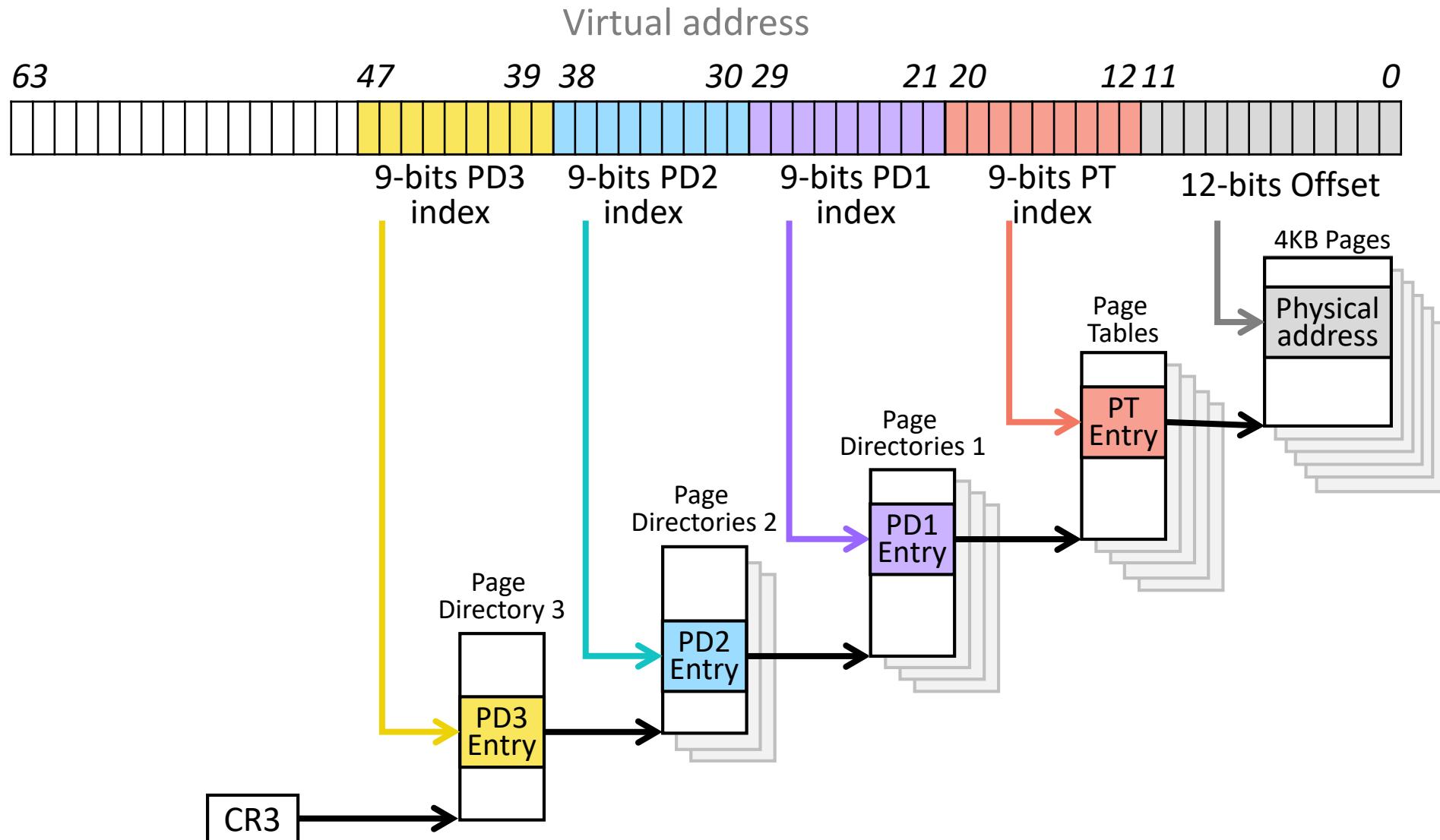


Example: 32-bit x86 (Two-level)

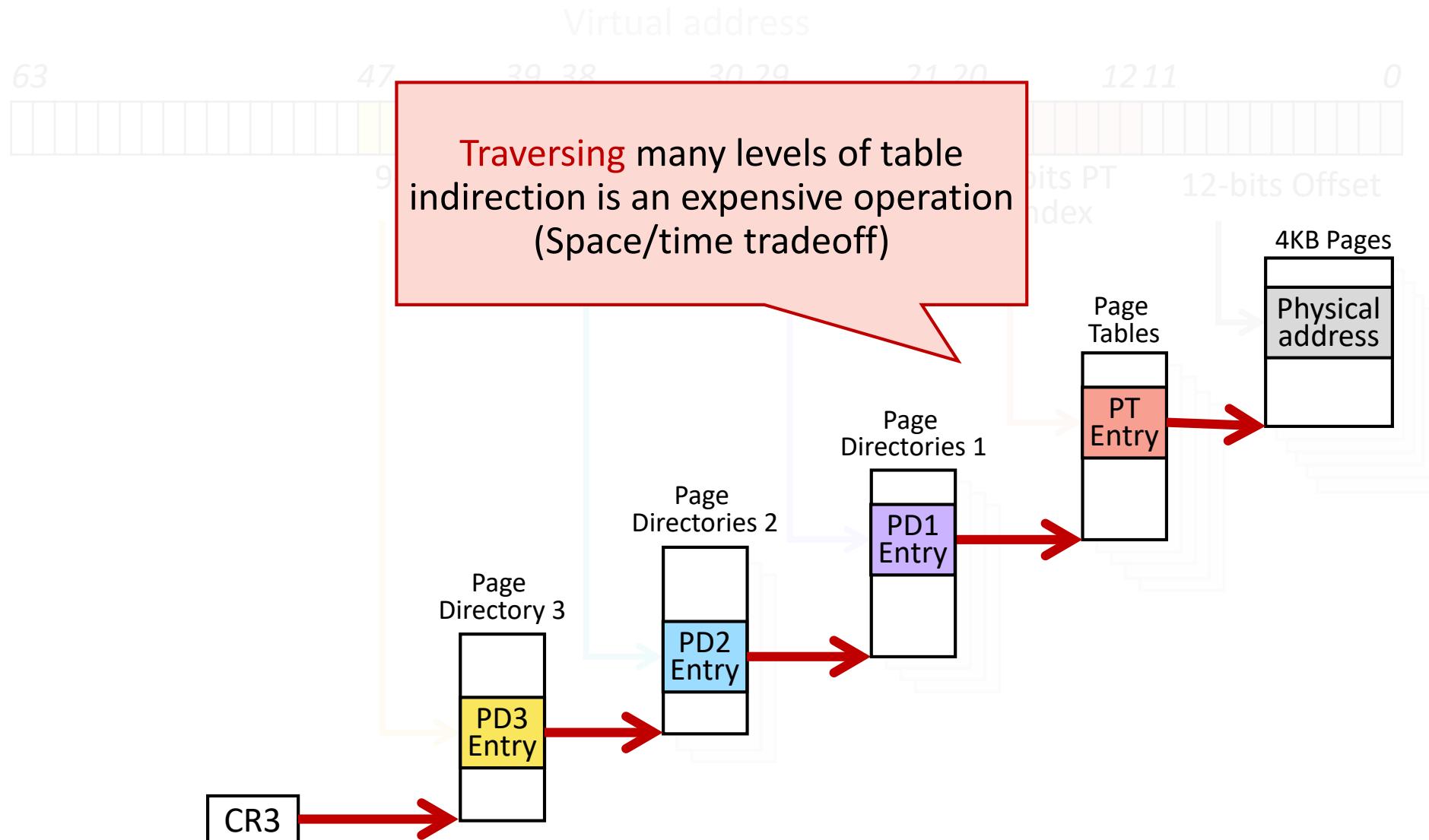




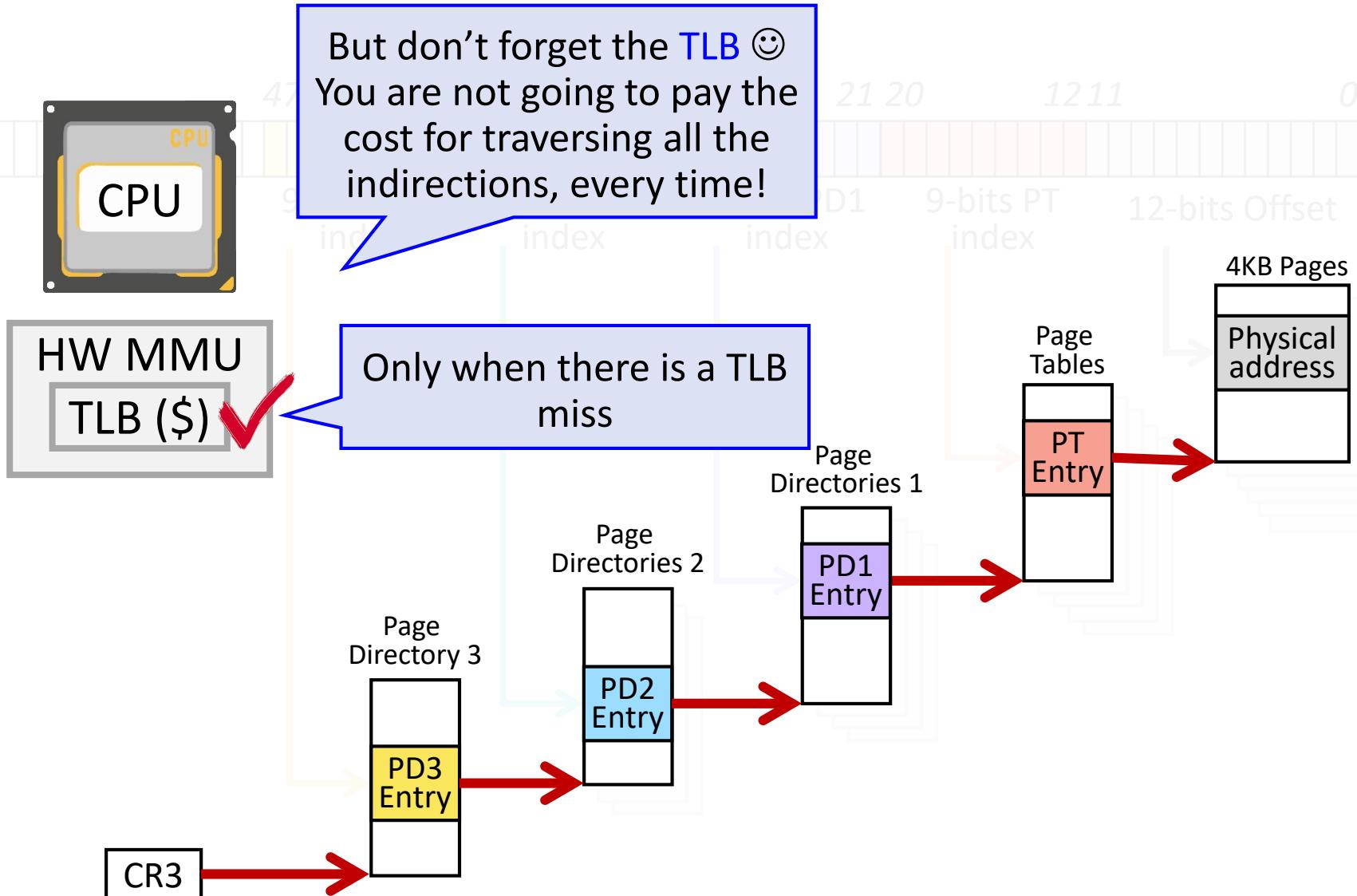
Example: 64-bit x86 (Four-level)



Drawback of Multi-level Page Table



Drawback of Multi-level Page Table



What's Next?

- Let's check the paging policies (page replacement, page allocation)

21S EE415

Page Replacement

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



Page Replacement

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab*

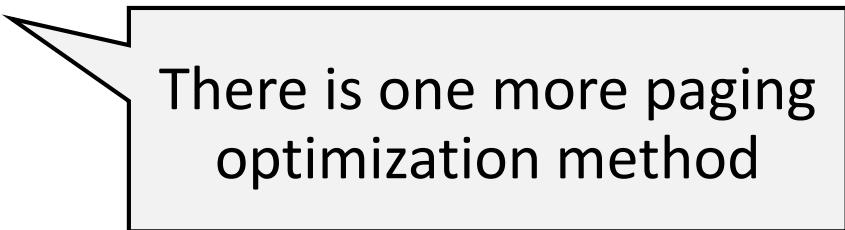


Optimizations

#1: Managing page tables (space-efficient)

#2: Efficient translations (time-efficient)

#3: Demand paging (space-efficient)



There is one more paging optimization method

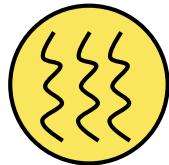
Optimization #3: Demand Paging

- Space optimization

Problem: Run Out Of Memory

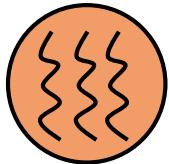
- What if we completely run out of physical memory?

Process A



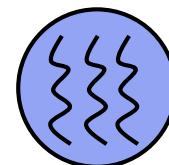
More memory

Process B

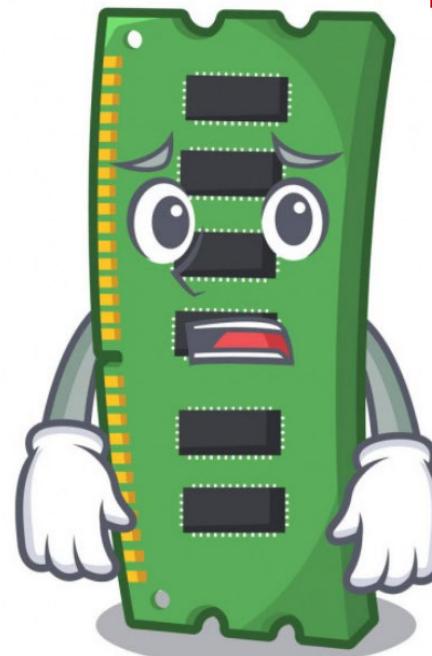


More memory

Process C



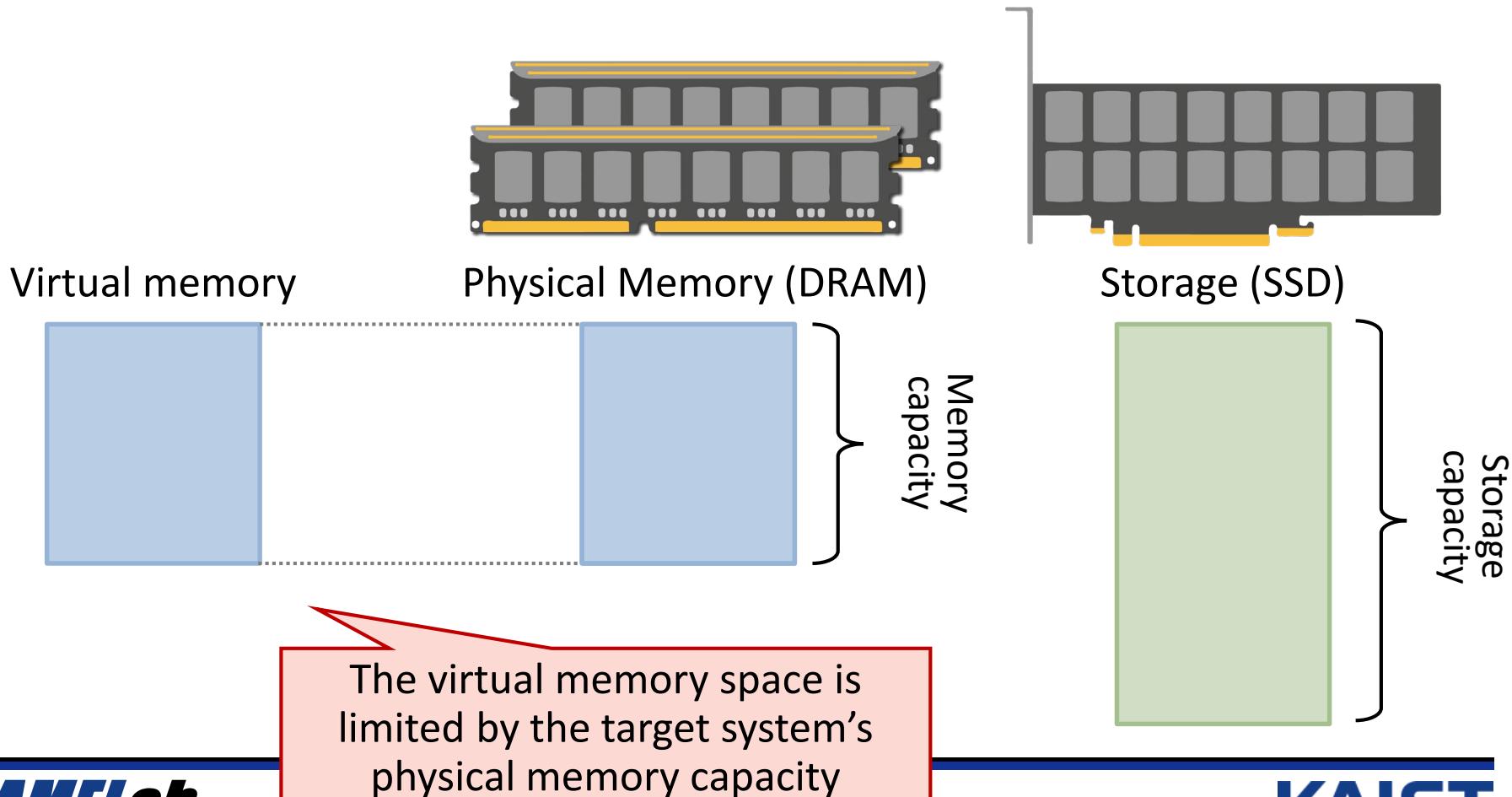
More memory



Oops... I do not have
extra free page frames

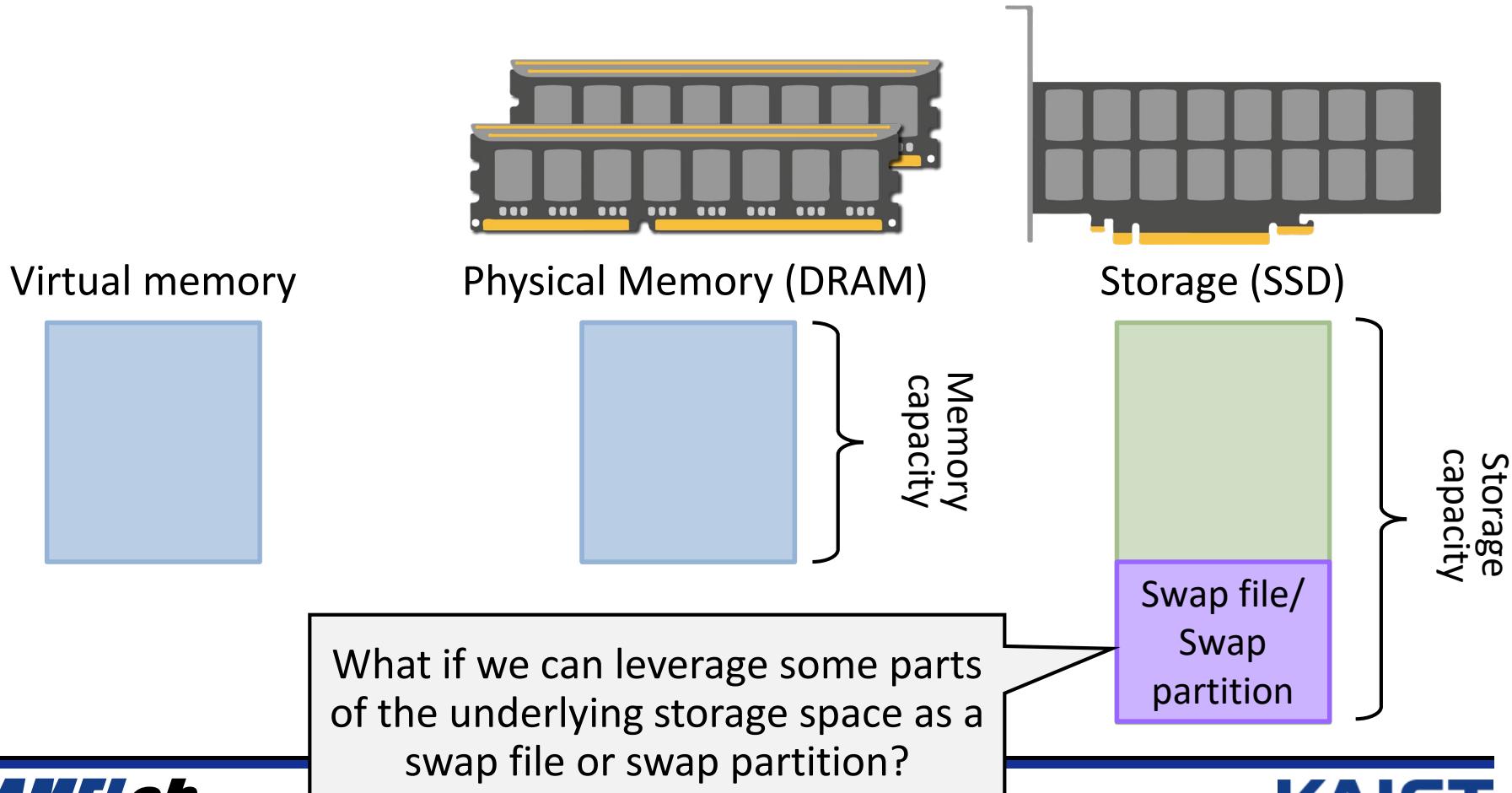
Solution: Swap Space

- **Key idea:** Use storage (HDD/SSD) to simulate a larger virtual memory than a physical memory



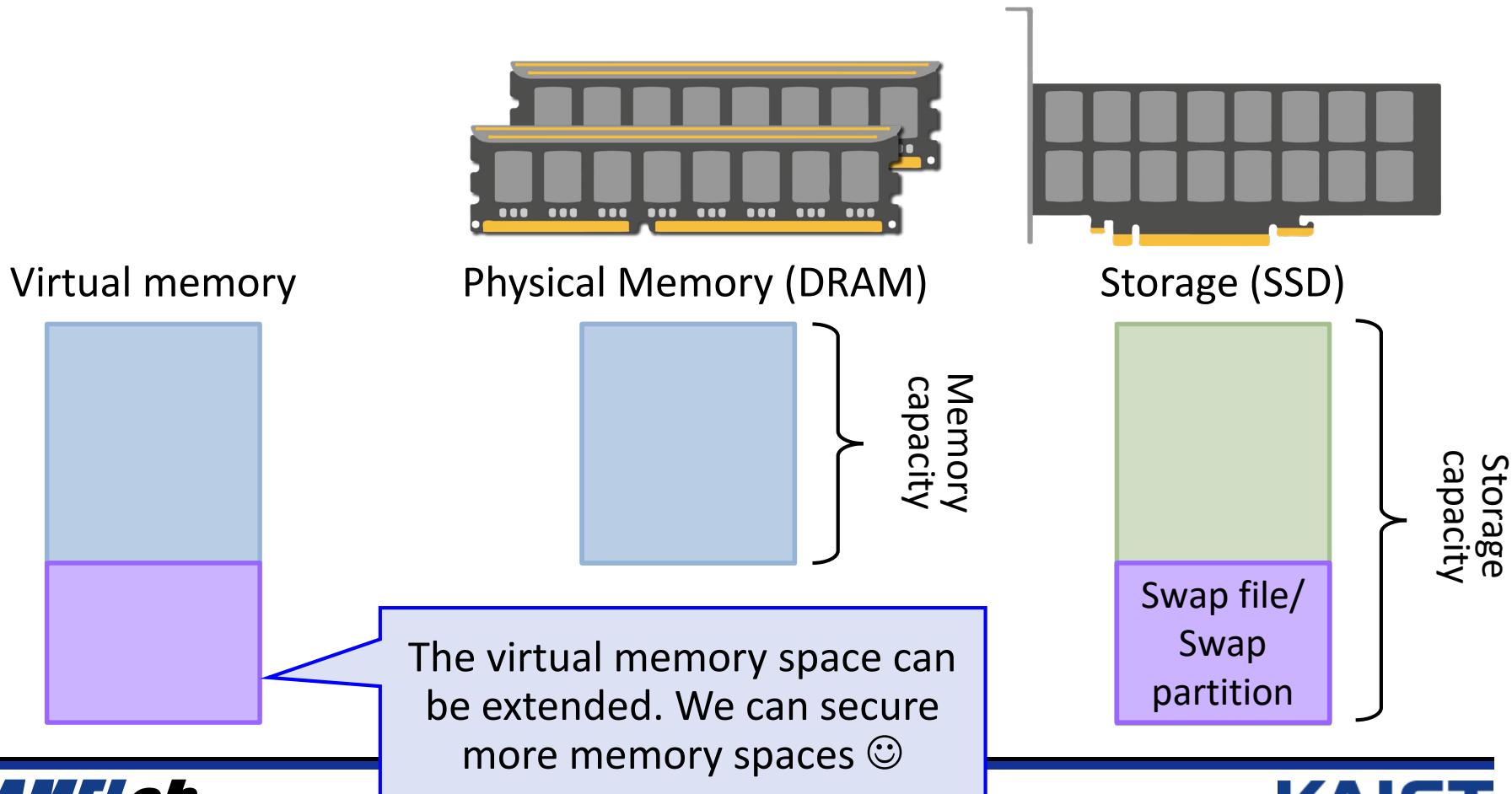
Solution: Swap Space

- Key idea: Use storage (HDD/SSD) to simulate larger virtual memory than physical memory



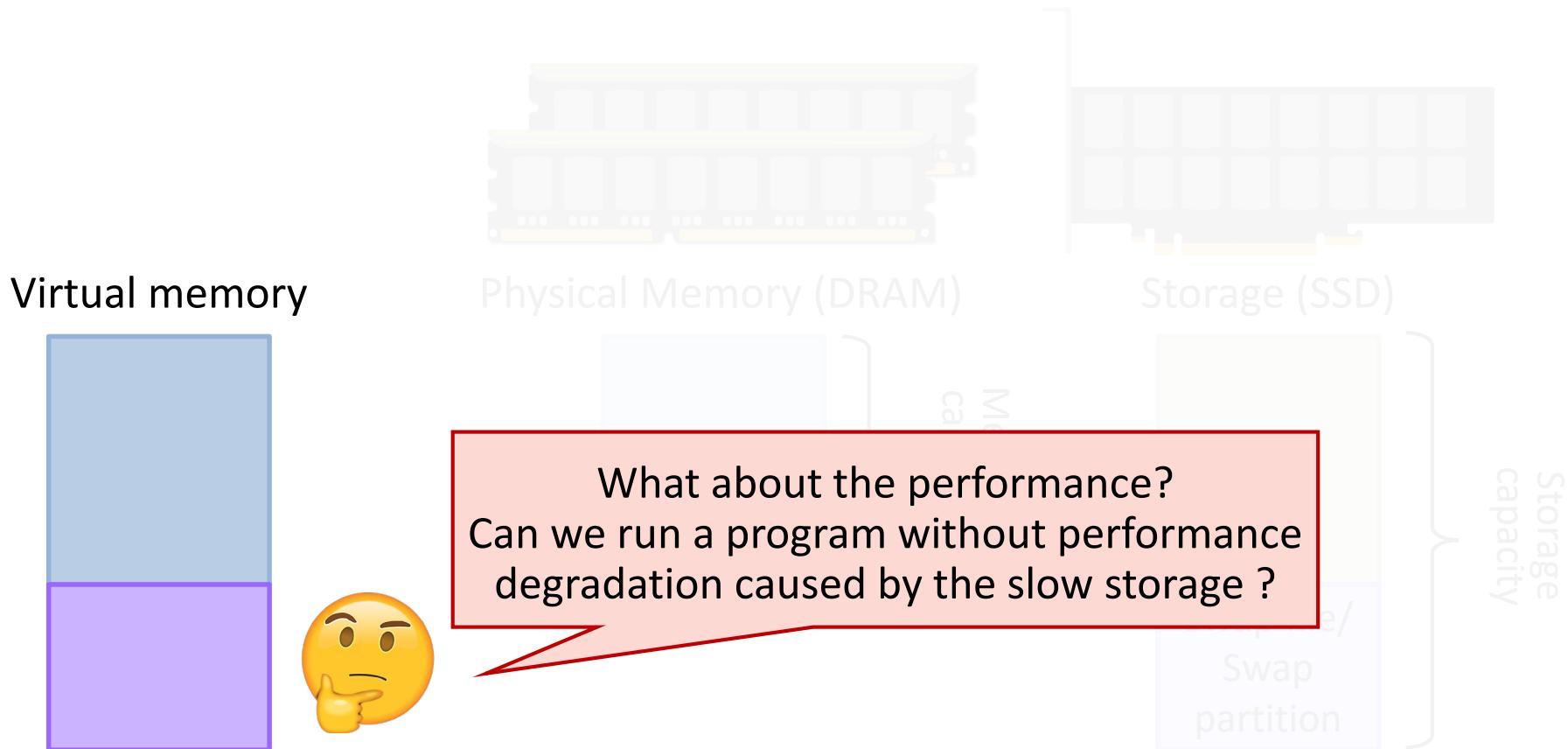
Solution: Swap Space

- Key idea: Use storage (HDD/SSD) to simulate larger virtual memory than physical memory



Solution: Swap Space

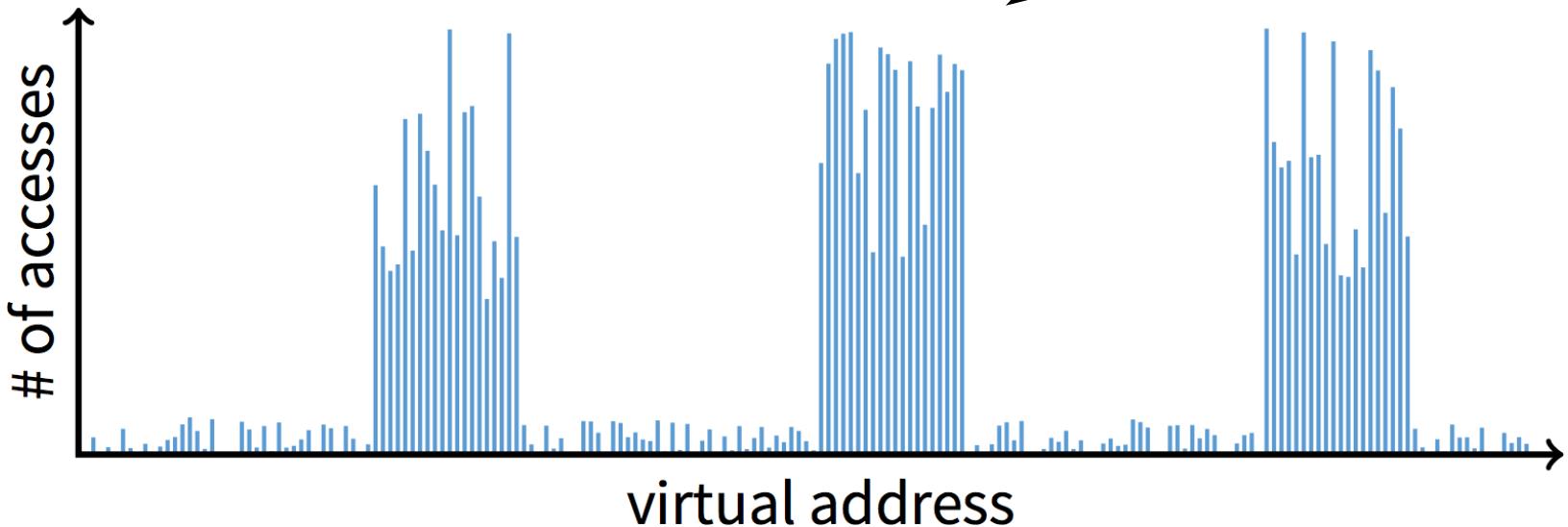
- Key idea: Use storage (HDD/SSD) to simulate larger virtual memory than physical memory



Insight: Locality

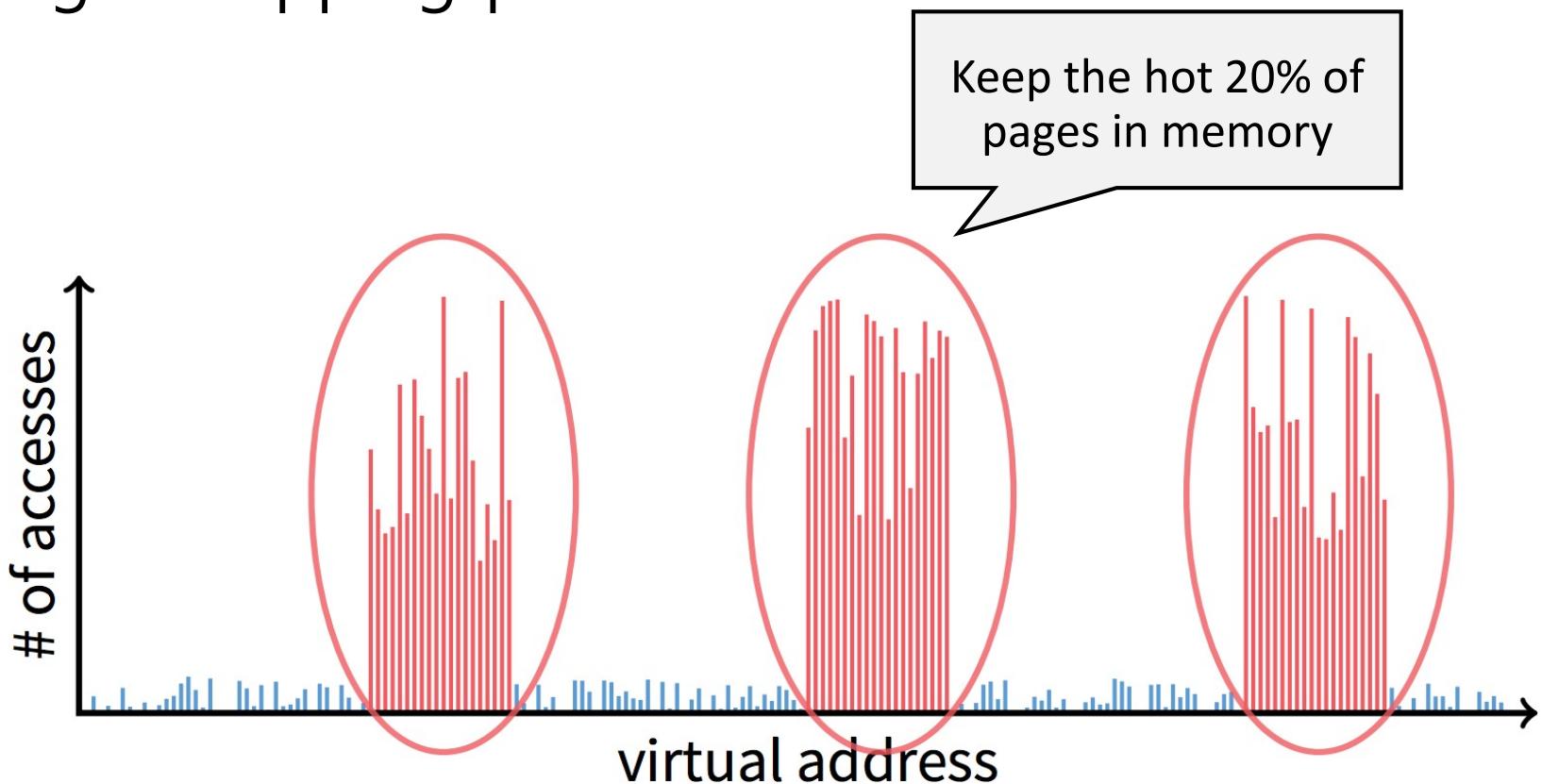
- Processes usually exhibit both temporal locality and spatial locality during their execution, which can in turn make page swapping practical

[Working Set Model]
80/20 rule: 20% of memory gets 80% of memory accesses



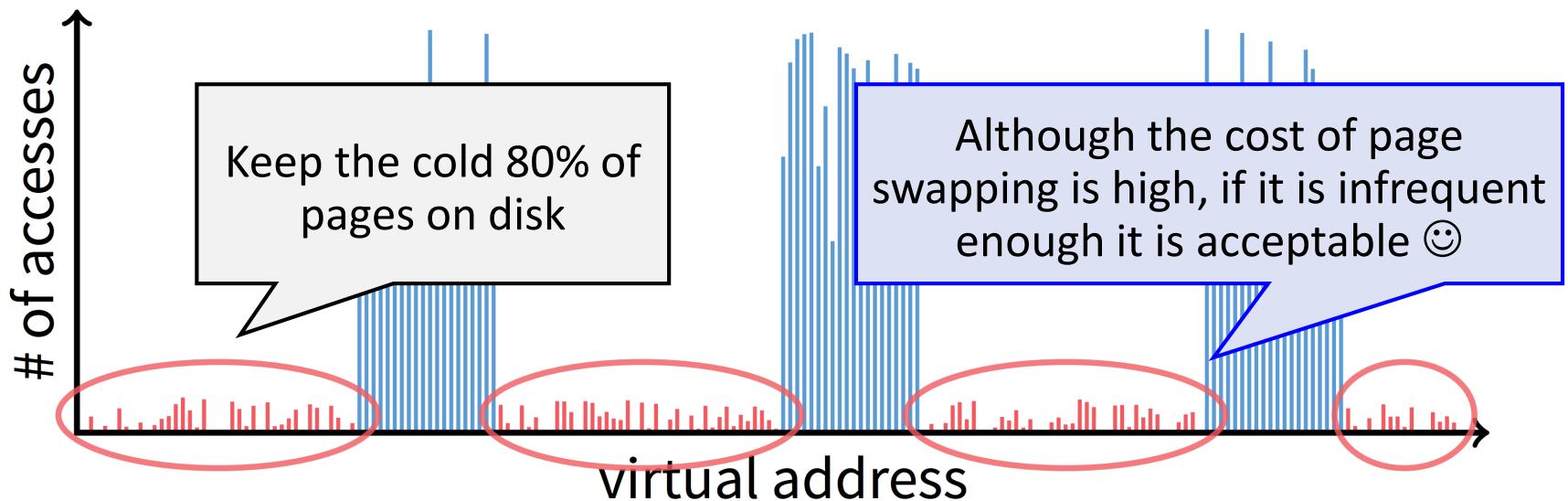
Insight: Locality

- Processes usually exhibit both temporal locality and spatial locality during their execution, making page swapping practical



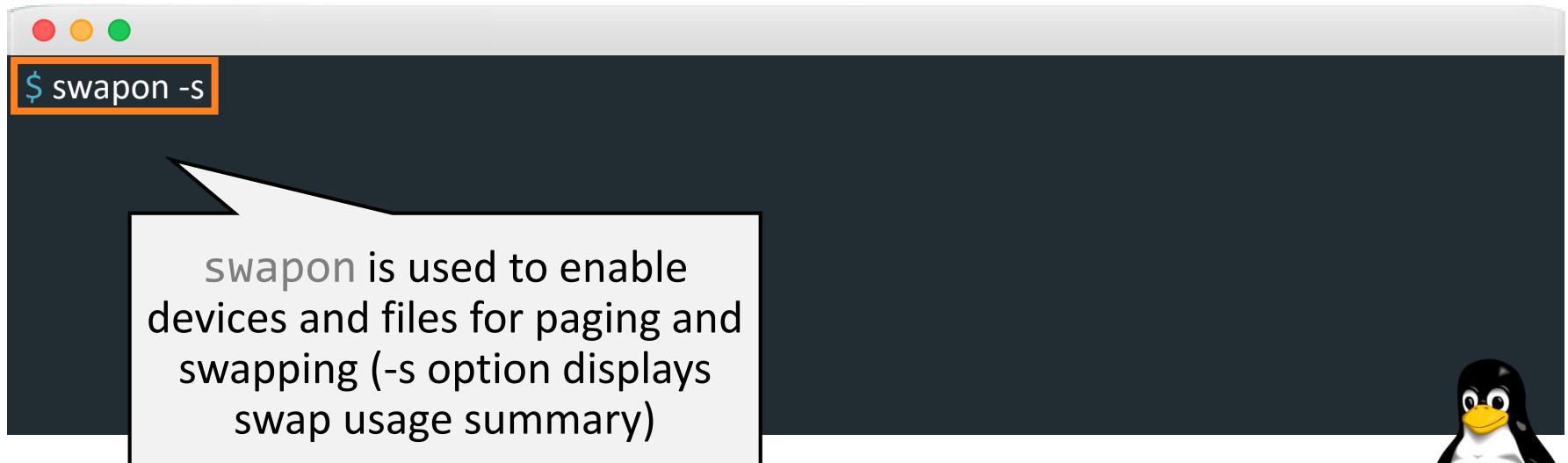
Insight: Locality

- Processes usually exhibit both temporal locality and spatial locality during their execution, which can in turn make page swapping practical



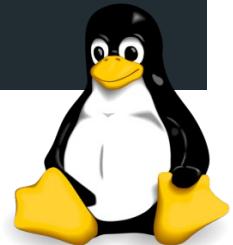
All Modern OSes Support Swapping

- On Linux, you create a *swap file/partition* along with your normal ext3/4 filesystem
 - Swapped pages are stored in this separate file/partition



```
$ swapon -s
```

swapon is used to enable devices and files for paging and swapping (-s option displays swap usage summary)

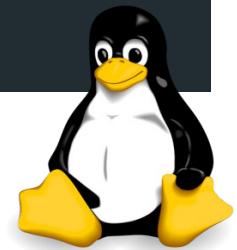


All Modern OSes Support Swapping

- On Linux, you create a *swap file/partition* along with your normal ext3/4 filesystem
 - Swapped pages are stored in this separate file/partition

```
$ swapon -s
Filename          Type  Size   Used  Priority
/dev/dm-1        partition 3906556 998036  -1
/mnt/swapfile    file    3906556      0     -2
```

This system employs the partition type as well as file type of swap area (also both are 3.7 GB)

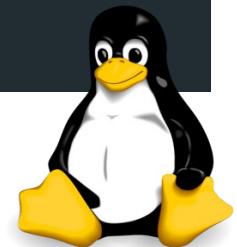


All Modern OSes Support Swapping

- On Linux, you create a *swap file/partition* along with your normal ext3/4 filesystem
 - Swapped pages are stored in this separate file/partition

```
$ swapon -s
Filename          Type      Size    Used   Priority
/dev/dm-1        partition 3906556 998036 -1
/mnt/swapfile    file      3906556      0     -2
$ free -h
```

free is used for monitoring the amount of free and used memories in the system

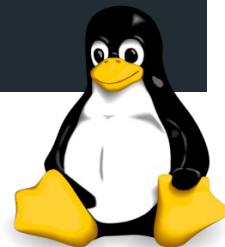


All Modern OSes Support Swapping

- On Linux, you create a *swap file/partition* along with your normal ext3/4 filesystem
 - Swapped pages are stored in this separate file/partition

```
$ swapon -s
Filename          Type      Size     Used     Priority
/dev/dm-1        partition 3906556  998036    -1
/mnt/swapfile    file      3906556      0       -2
$ free -h
               total   used   free
Mem:           64G    6.2G  55.8G
Swap:          7.4G   974M  7.3G
```

This shows the physical memory and swap memory states in the system

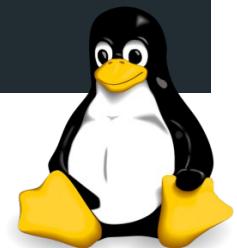


All Modern OSes Support Swapping

- On Linux, you create a *swap file/partition* along with your normal ext3/4 filesystem
 - Swapped pages are stored in this separate file/partition

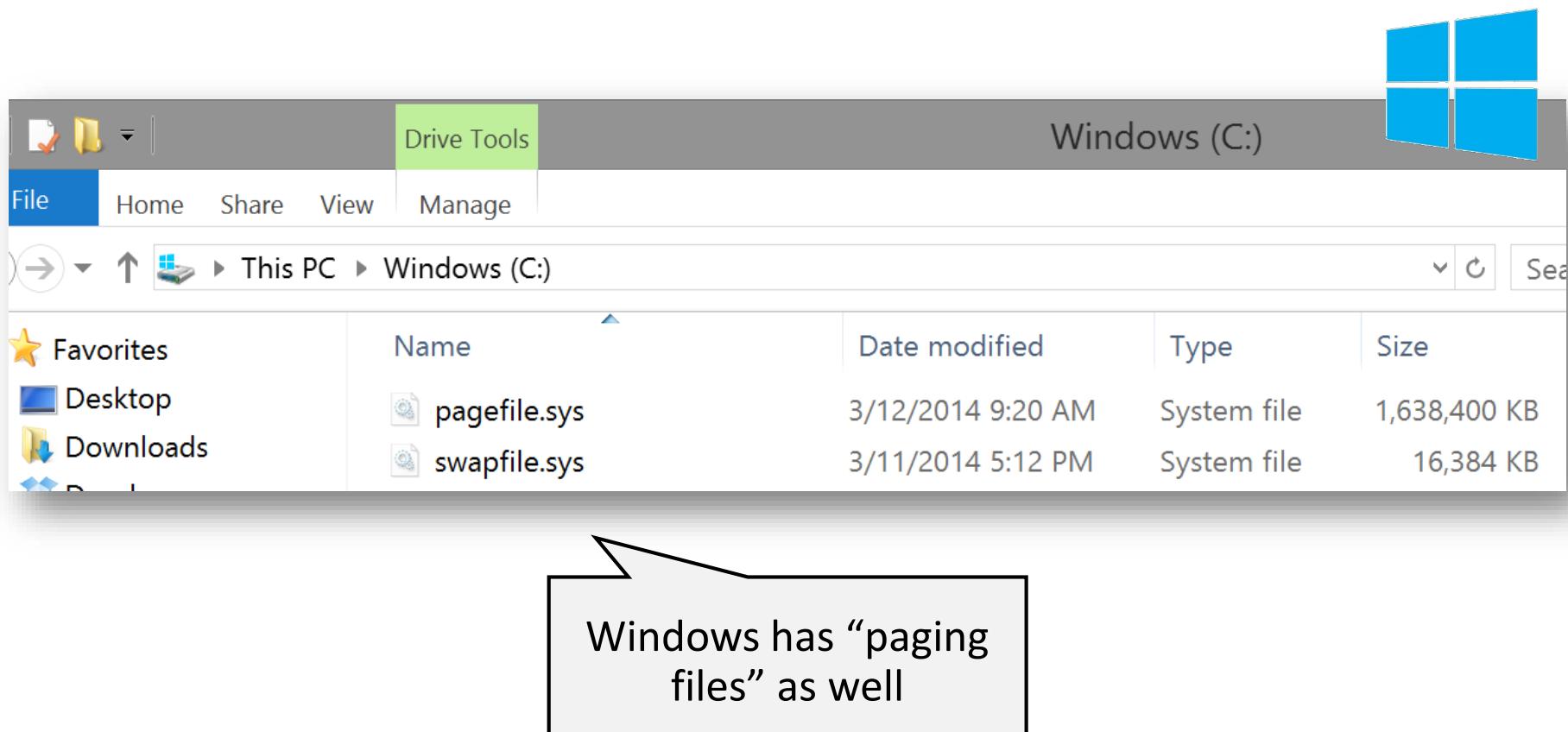
```
$ swapon -s
Filename          Type      Size        Used       Priority
/dev/dm-1         partition 3906556    998036      -1
/mnt/swapfile     file      3906556     0          -2
$ free -h
              total        used        free        ...
Mem:           64G        6.2G       55.8G
Swap:          7.4G       974M       7.3G
```

We only have 64GB of memory, but the virtual memory system thinks there are 71.4GB of memory in total.

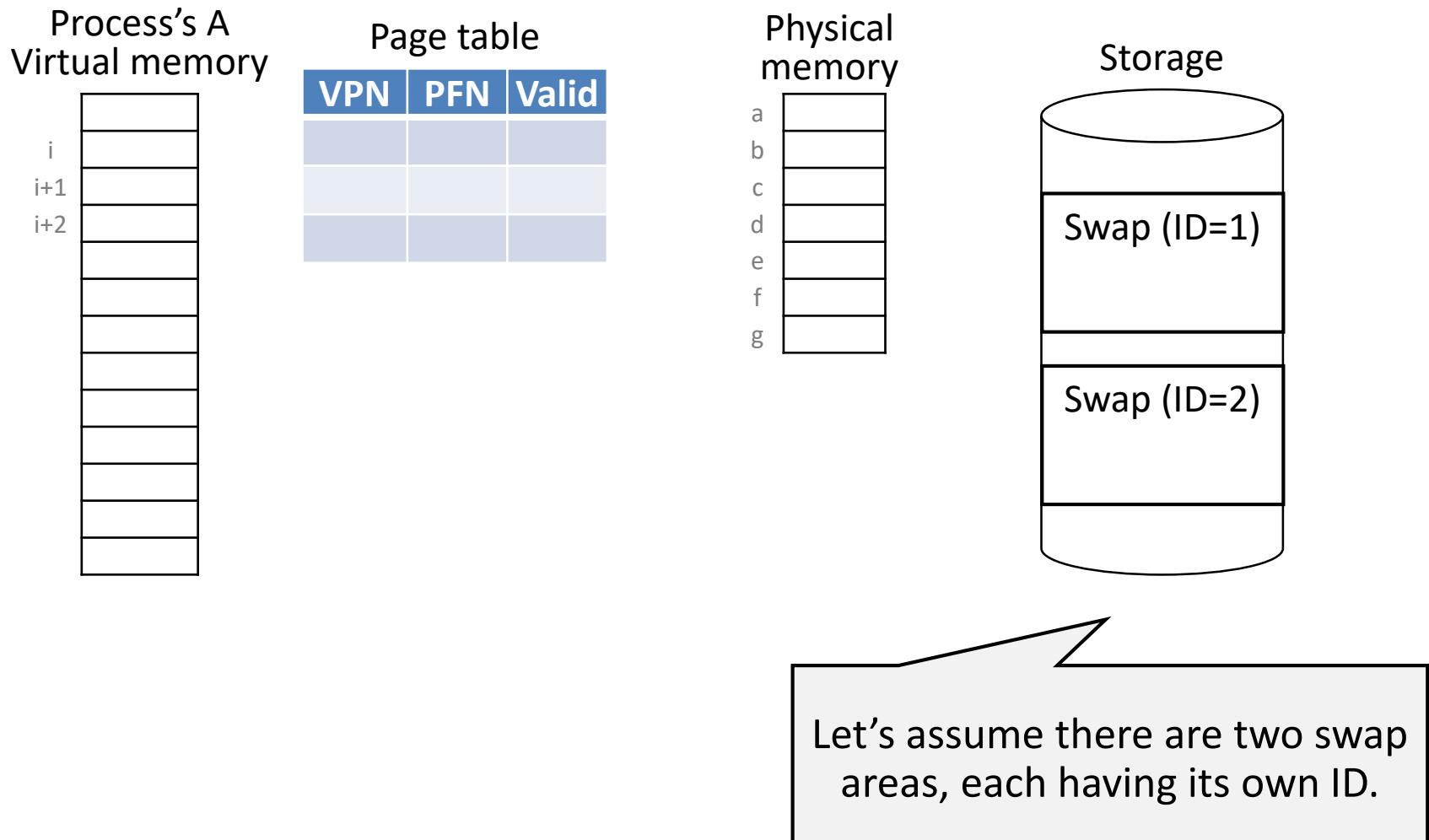


All Modern OSes Support Swapping

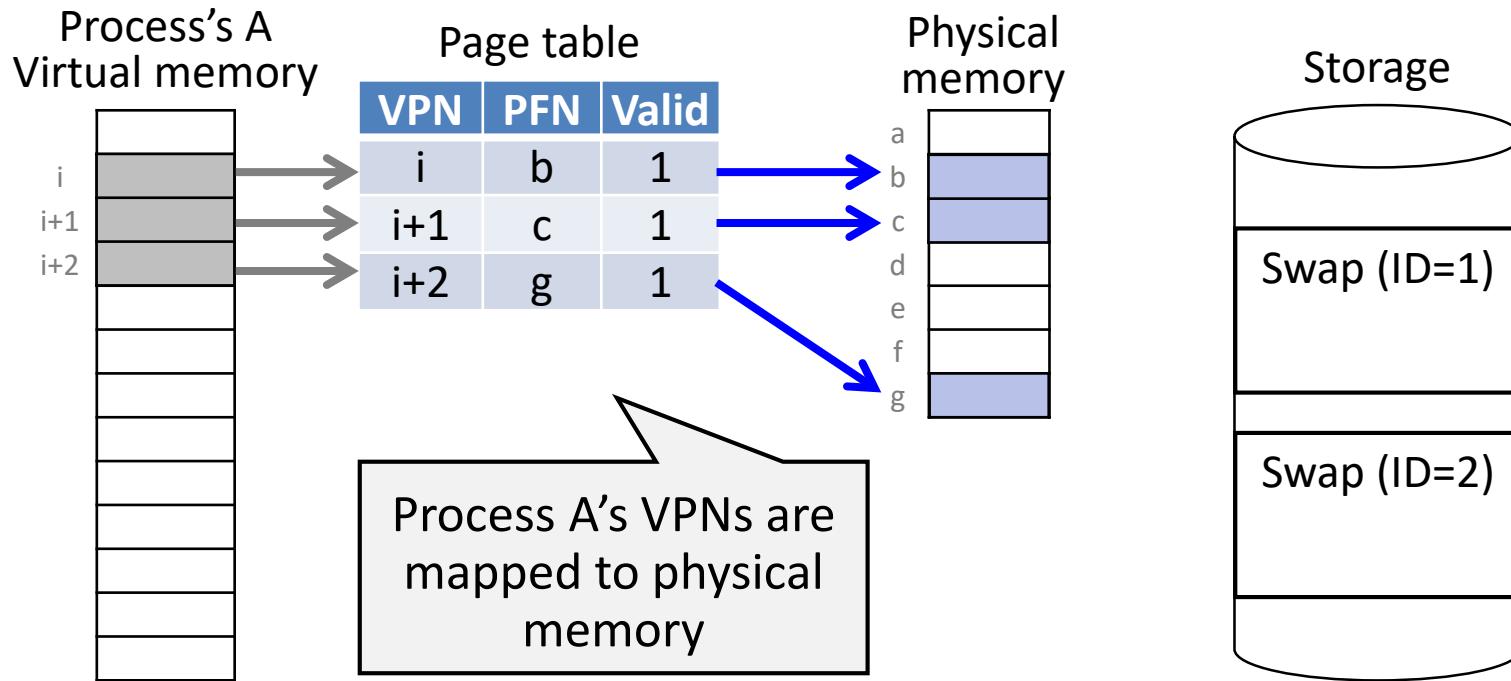
- Windows



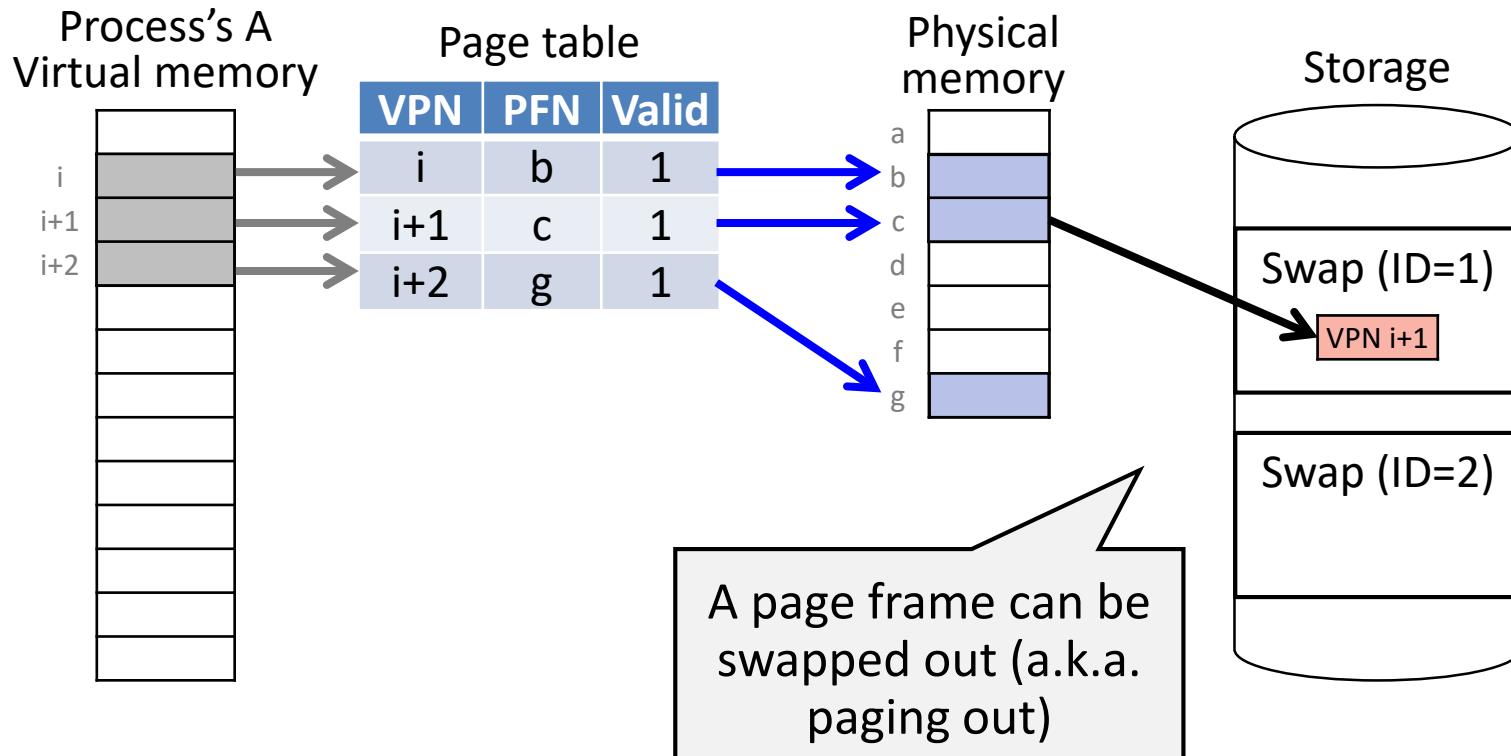
Implementing Swap



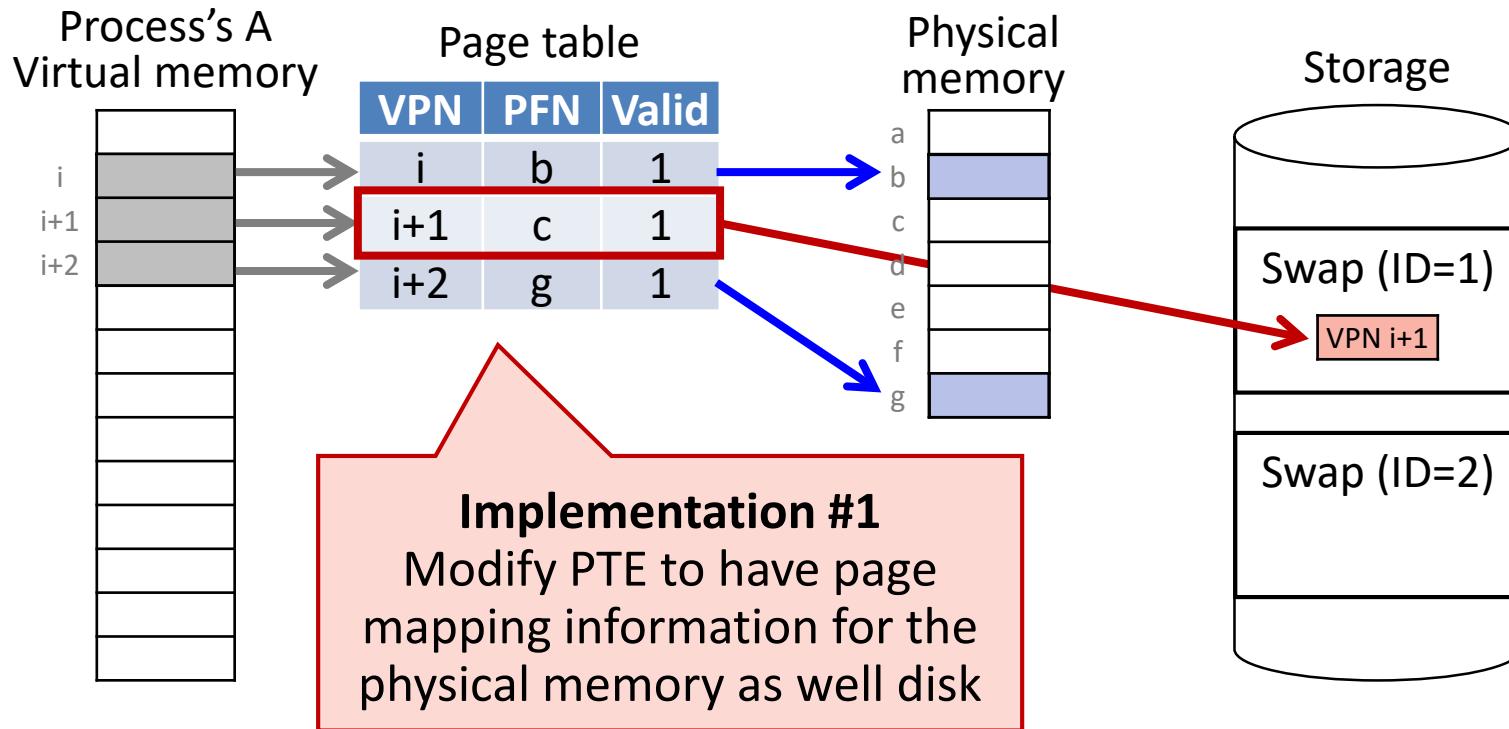
Implementing Swap



Implementing Swap

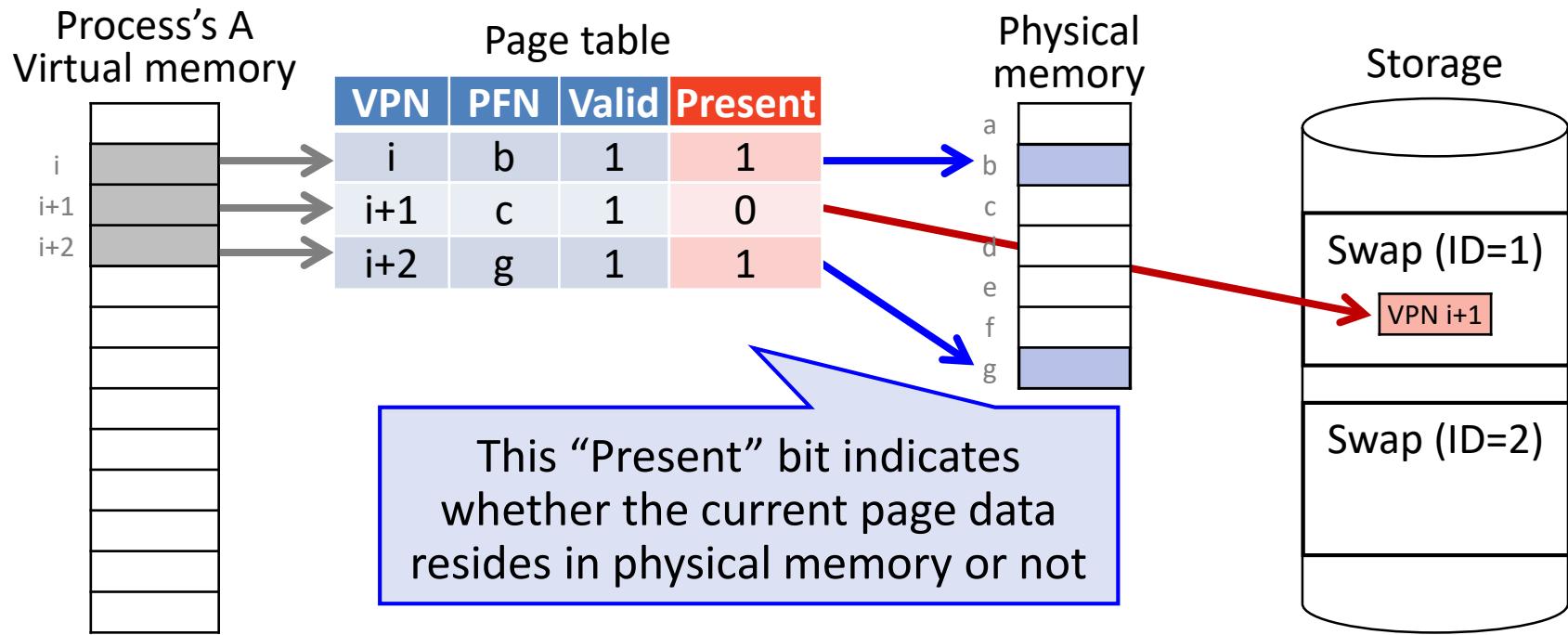


Implementing Swap



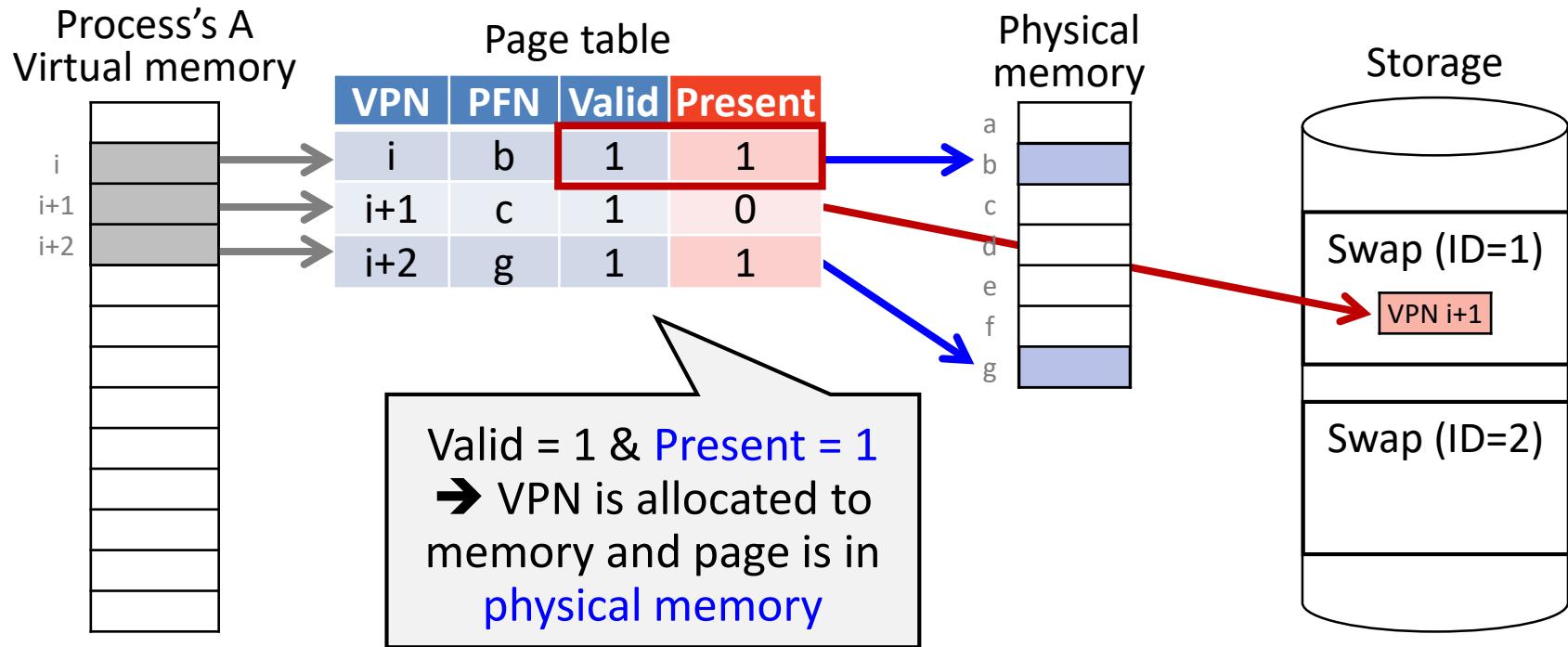


Modified PTE: Add Present Bit



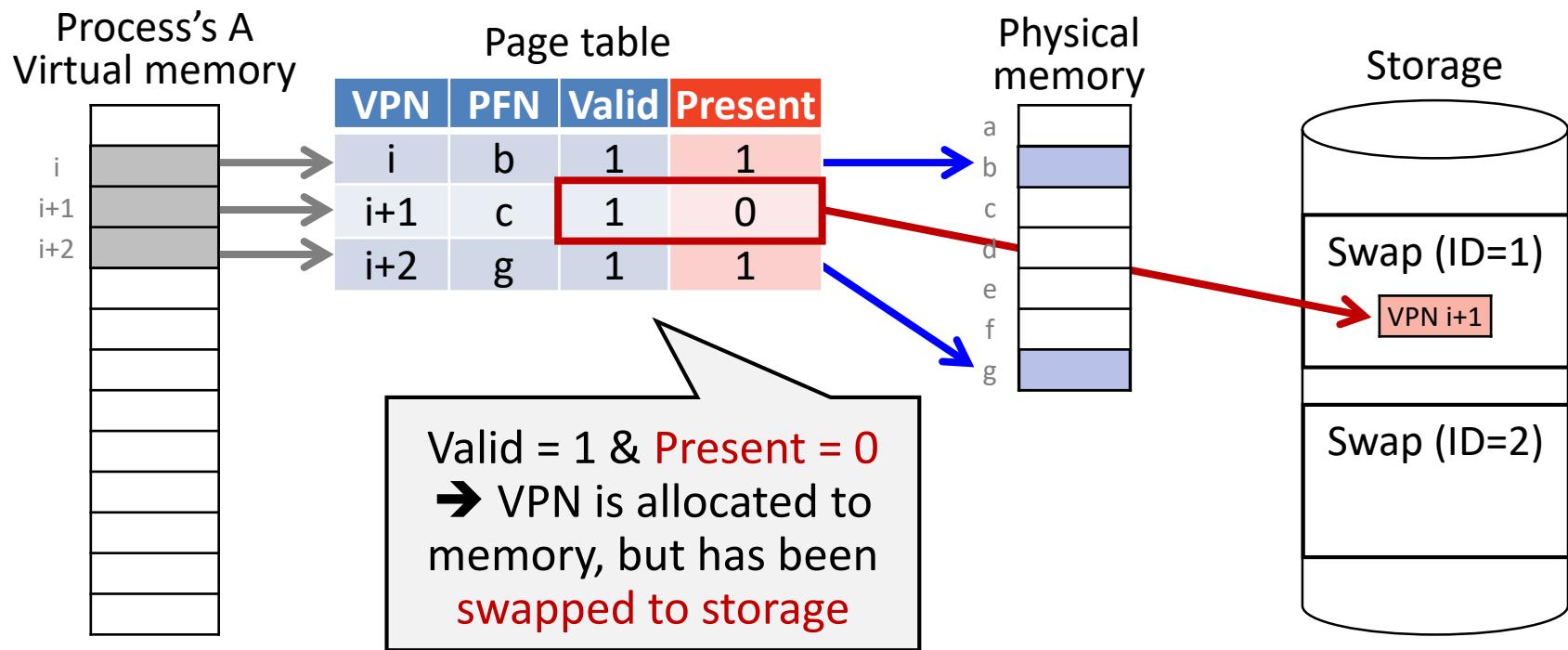


Modified PTE: Add Present Bit



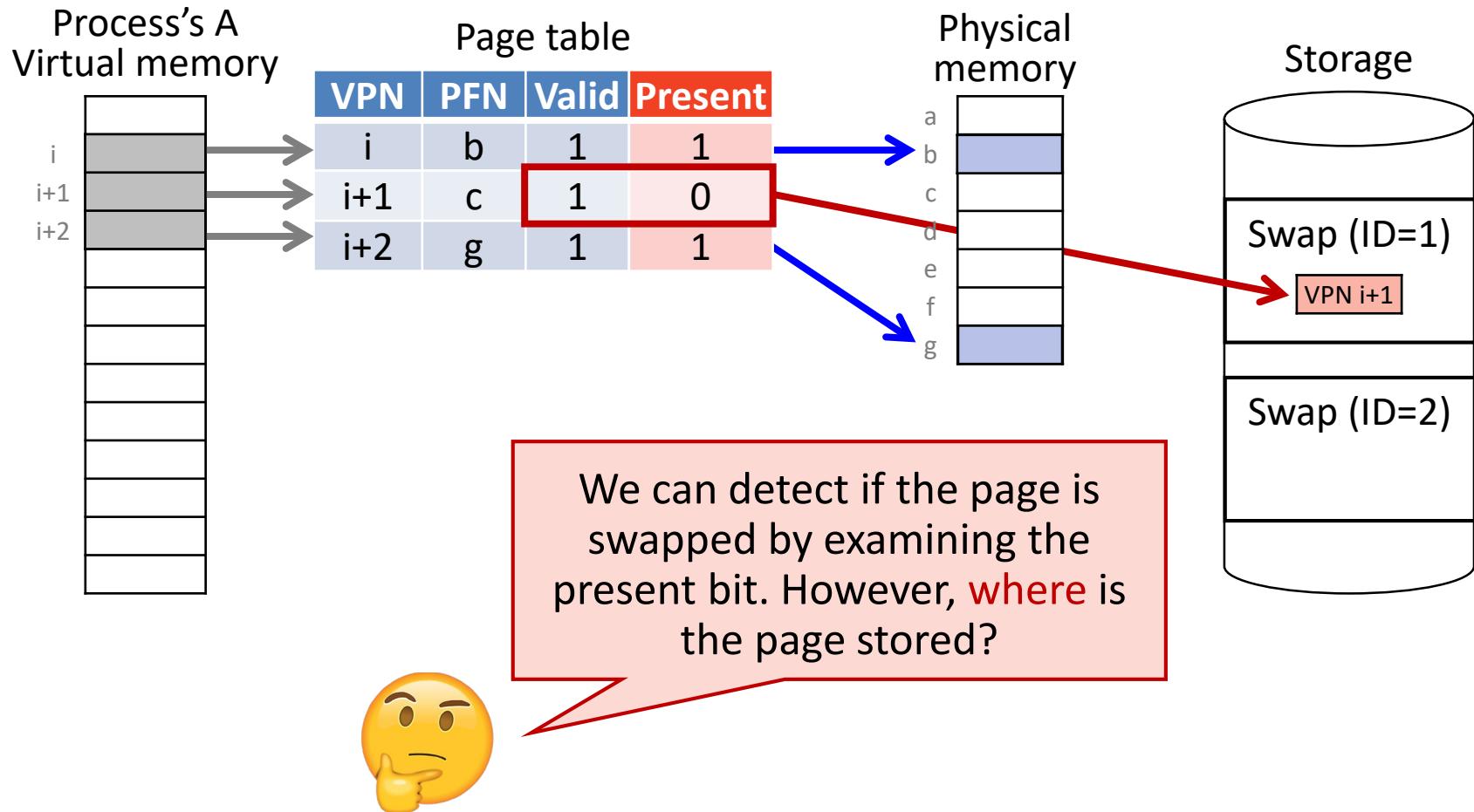


Modified PTE: Add Present Bit

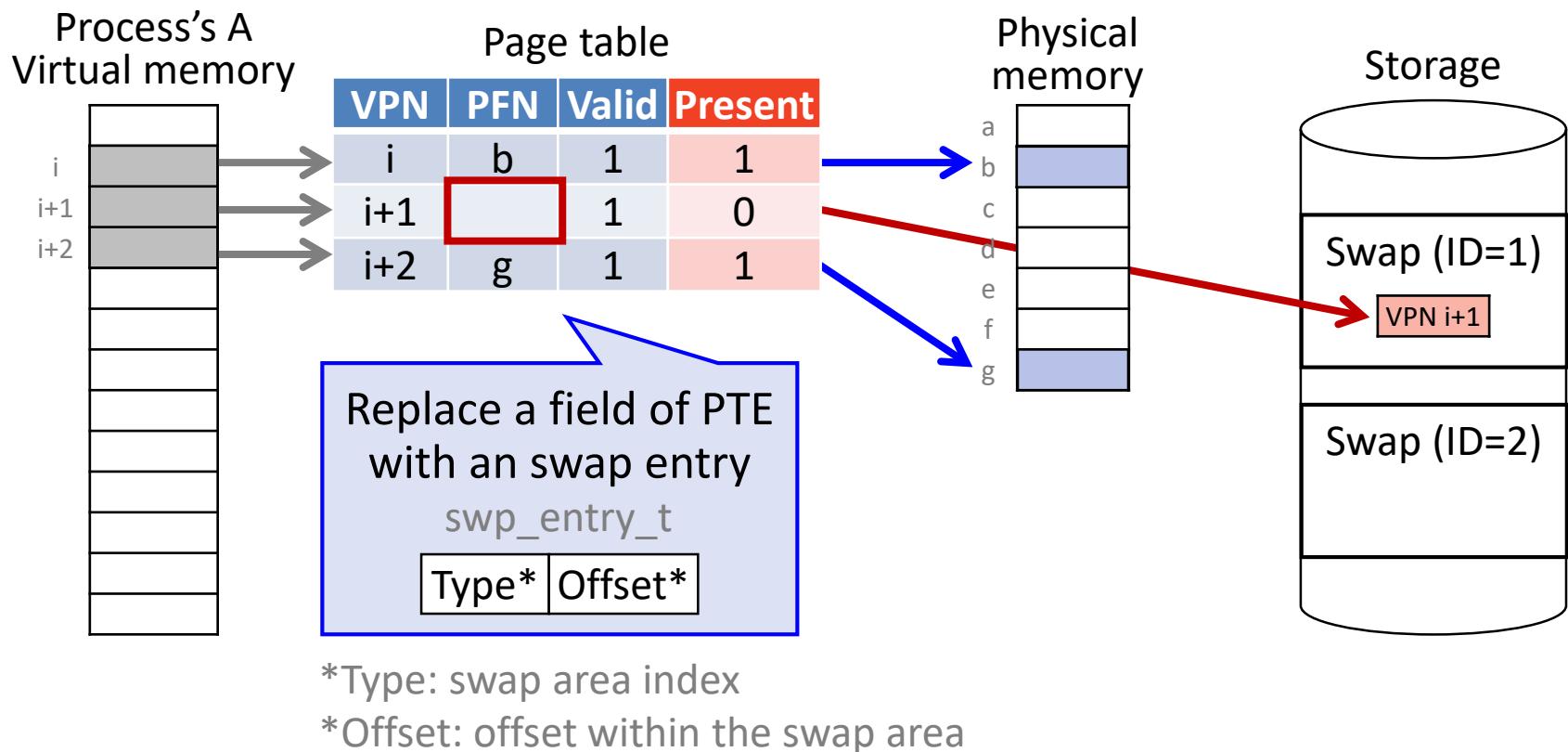




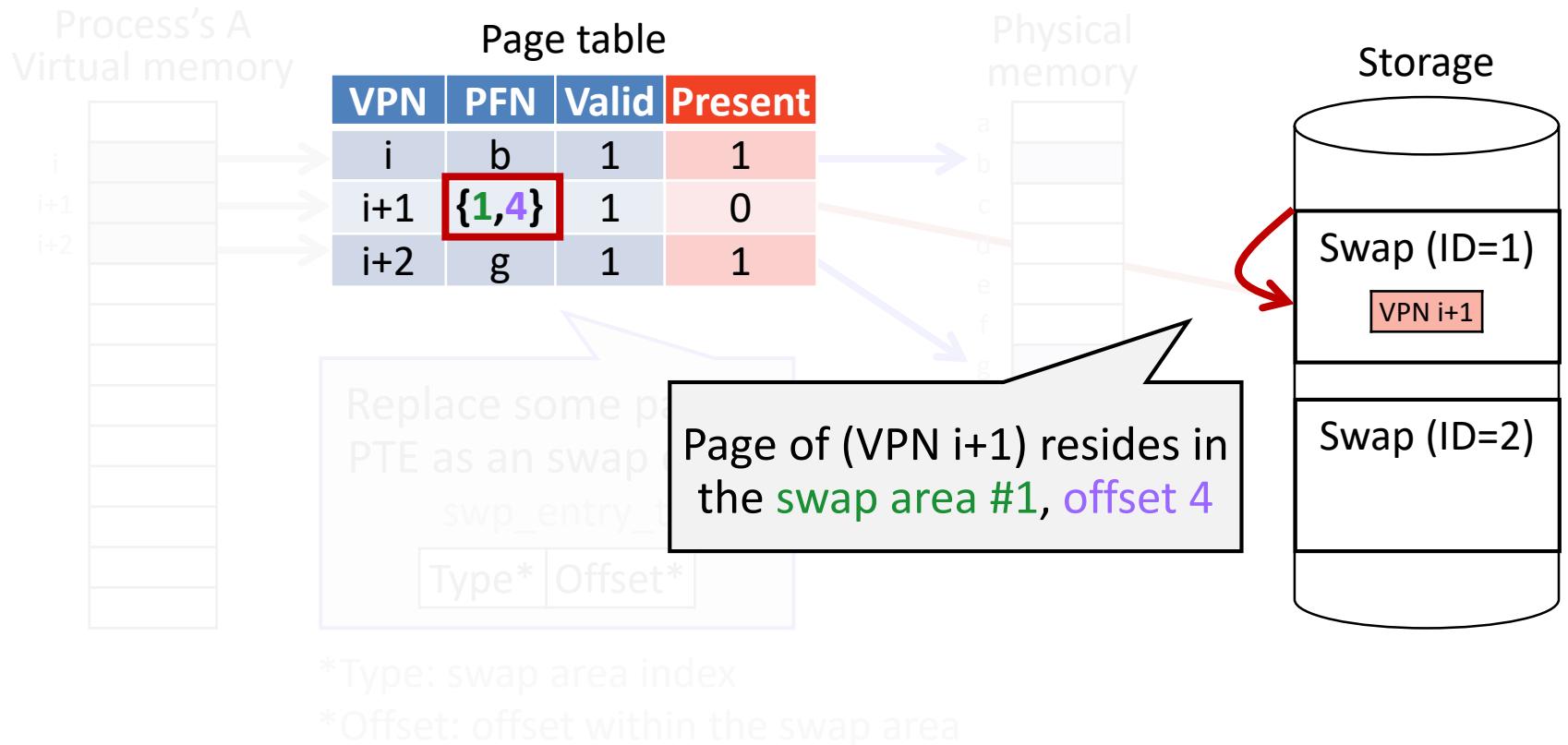
Modified PTE: Add Present Bit



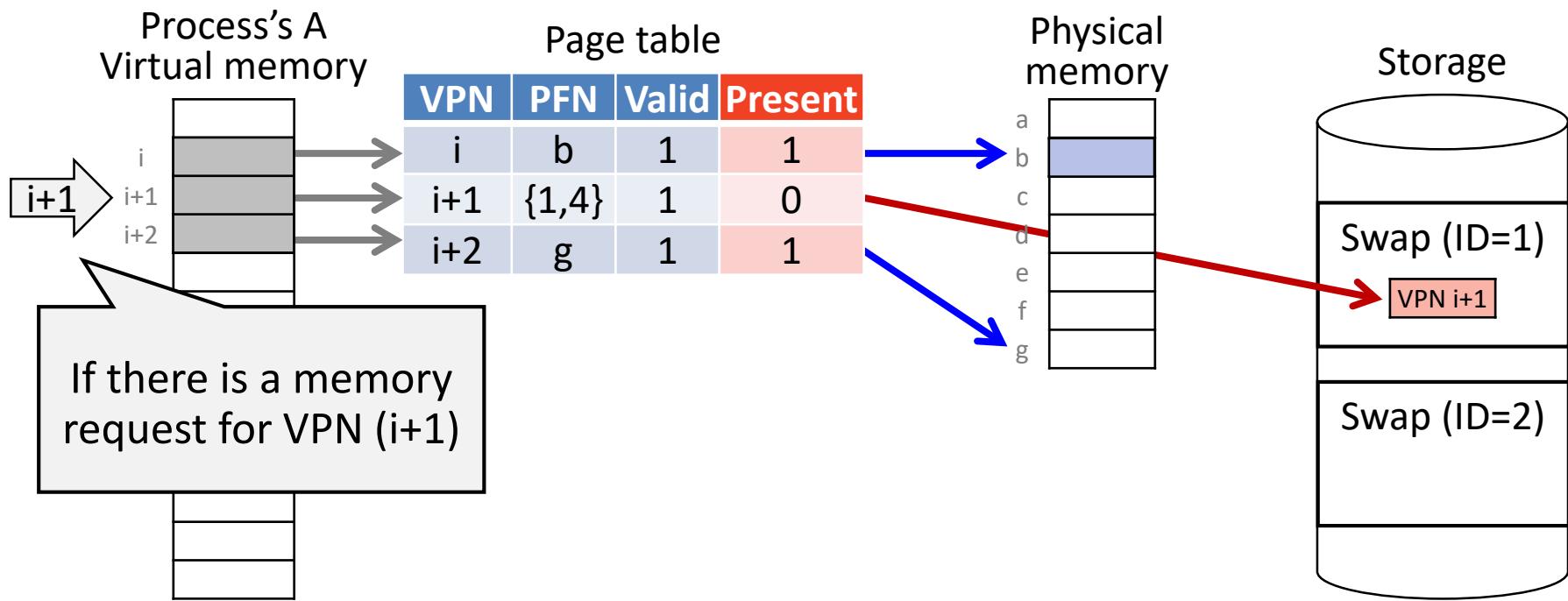
Modified PTE: PTE as Swap Entry



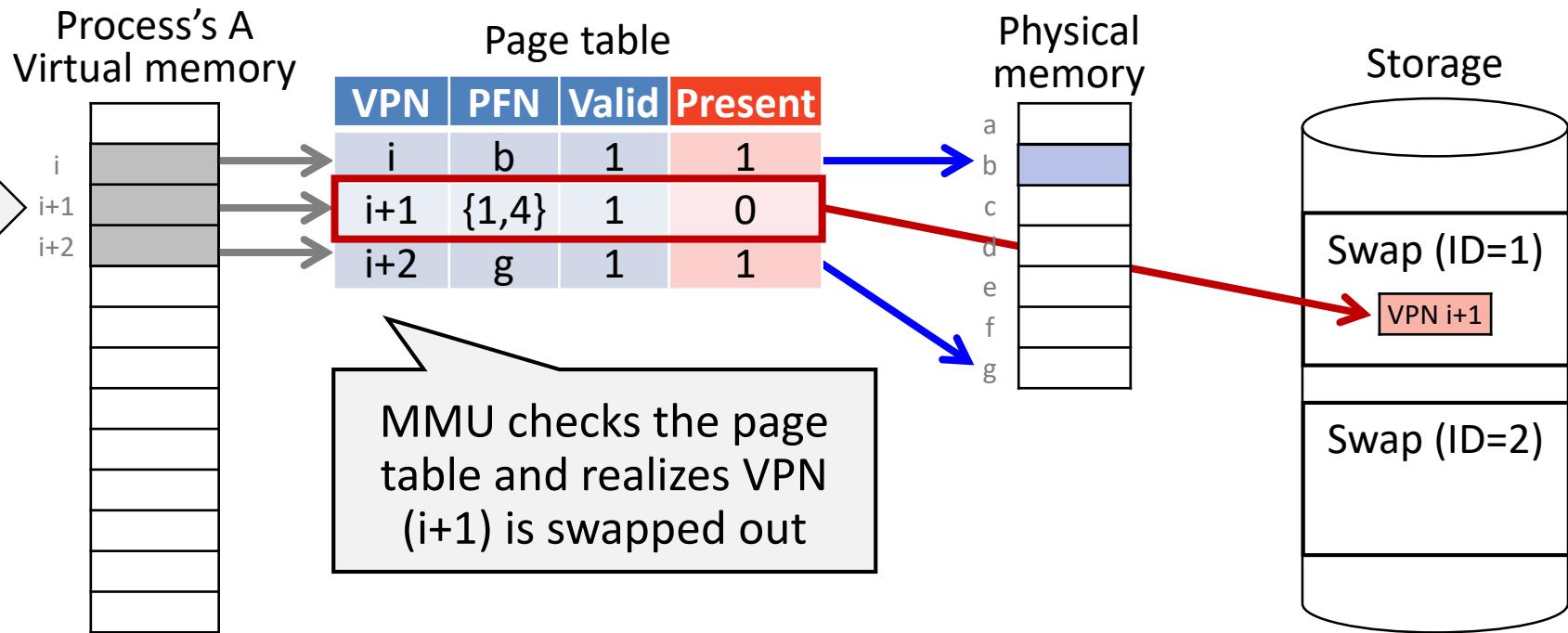
Modified PTE: PTE as Swap Entry



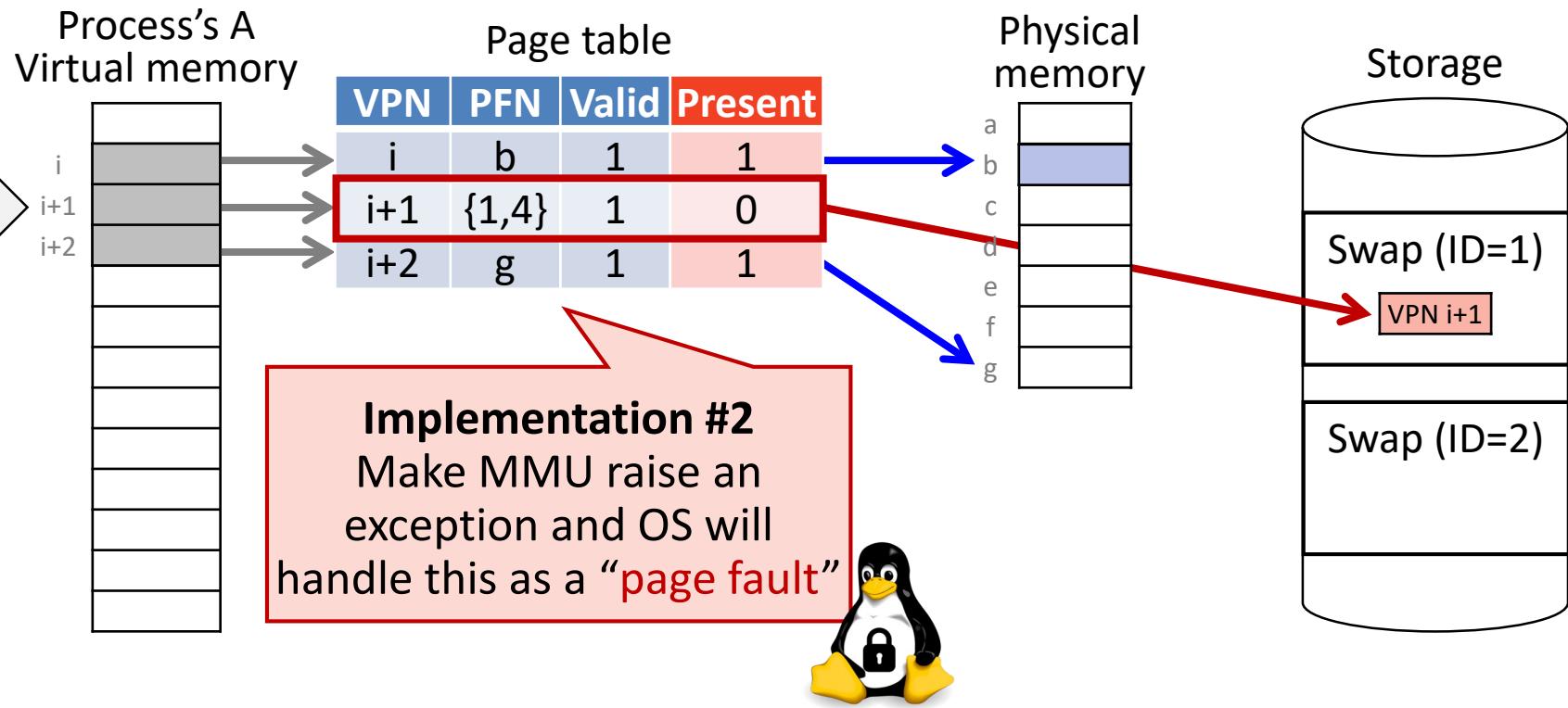
Implementing Swap



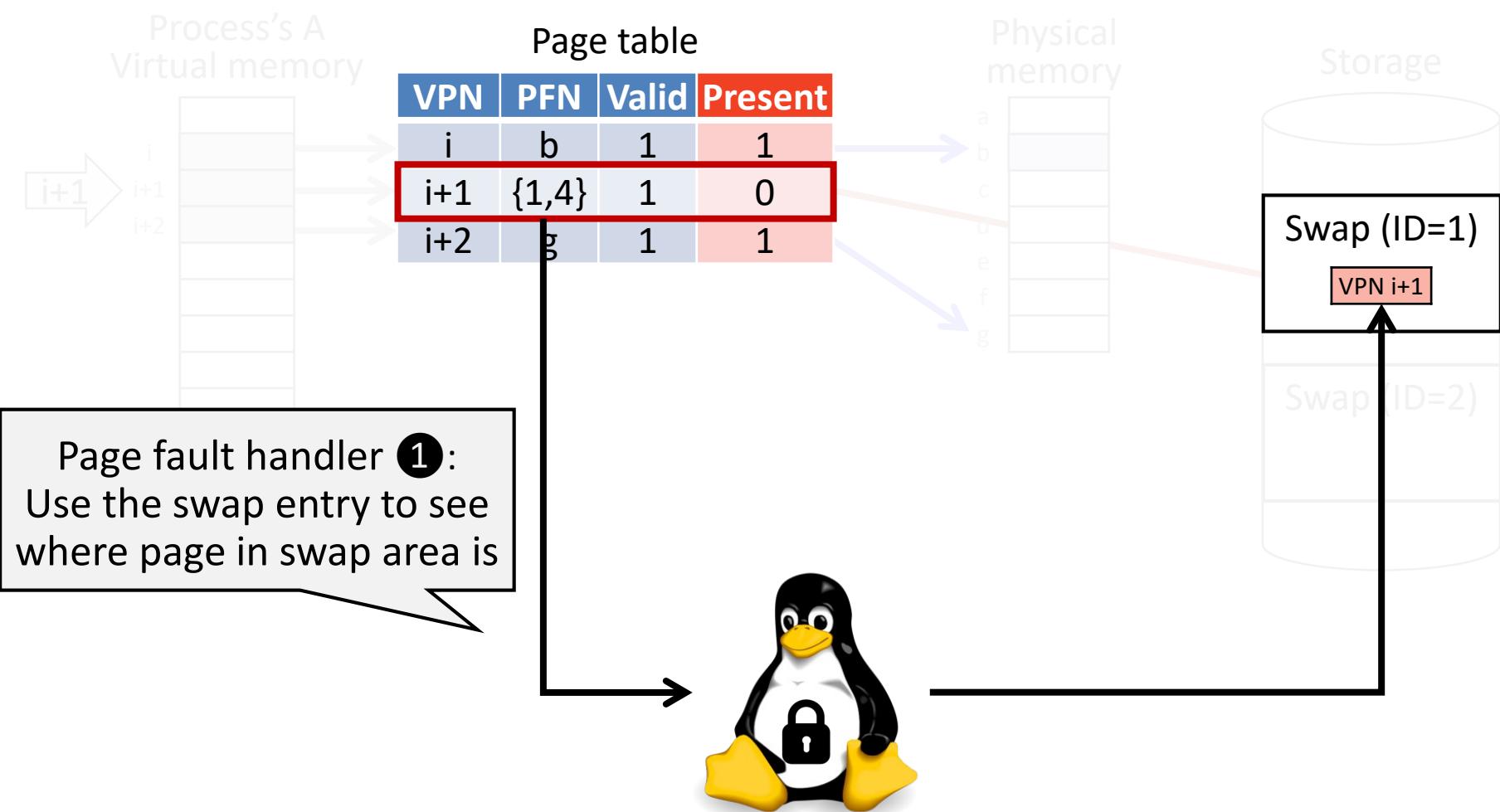
Implementing Swap



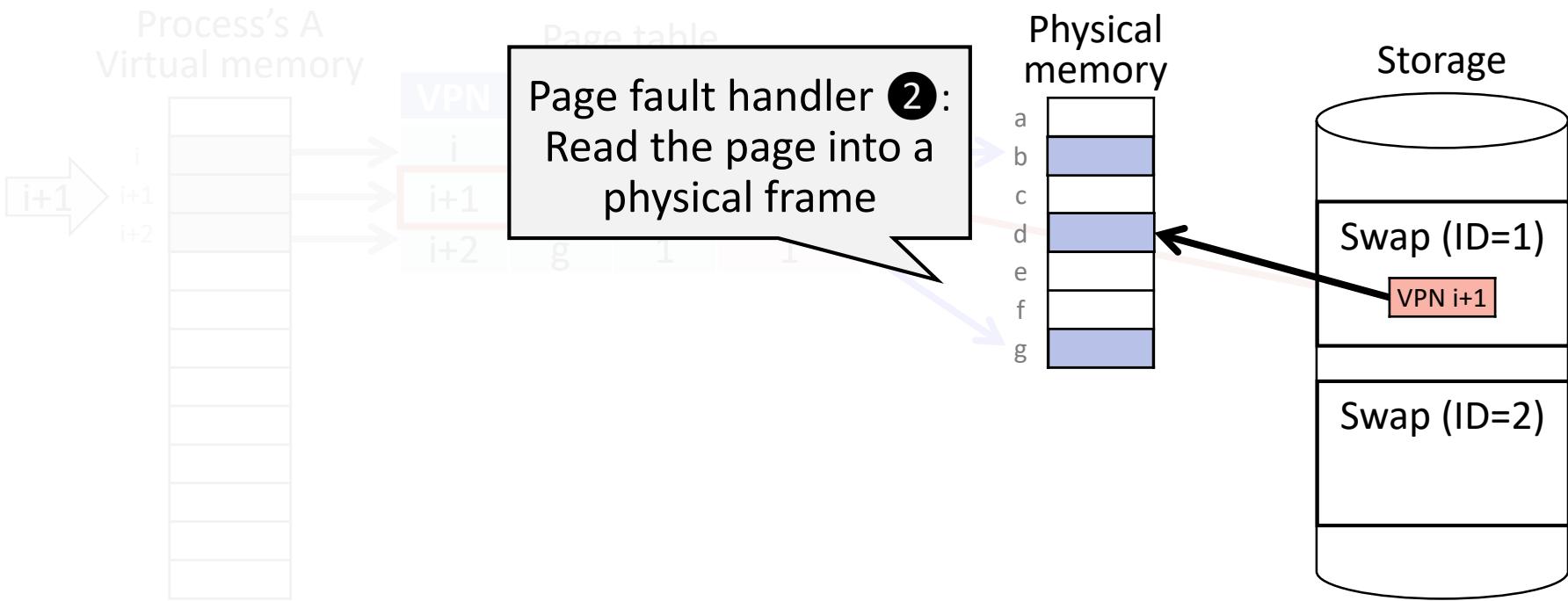
Implementing Swap



Page Fault: Loads the Faulted Page



Page Fault: Loads the Faulted Page





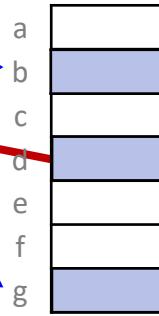
Page Fault: Loads the Faulted Page

Process's A
Virtual memory

Page table

VPN	PFN	Valid	Present
i	b	1	1
i+1	d	1	1
i+2	g	1	1

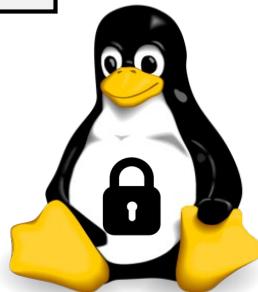
Physical
memory



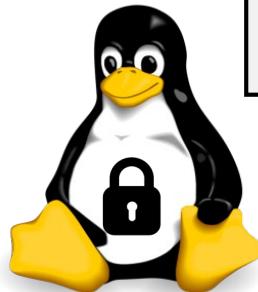
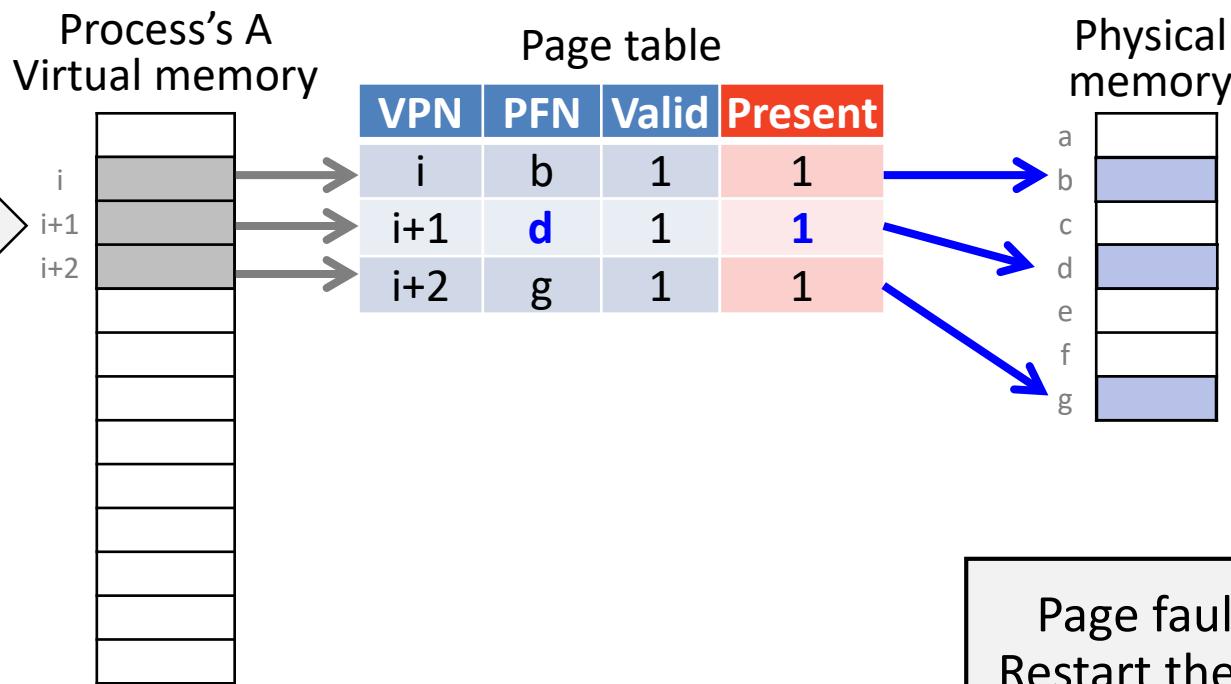
Storage



Page fault handler ③:
Update PTE to point
the actual location of
physical memory

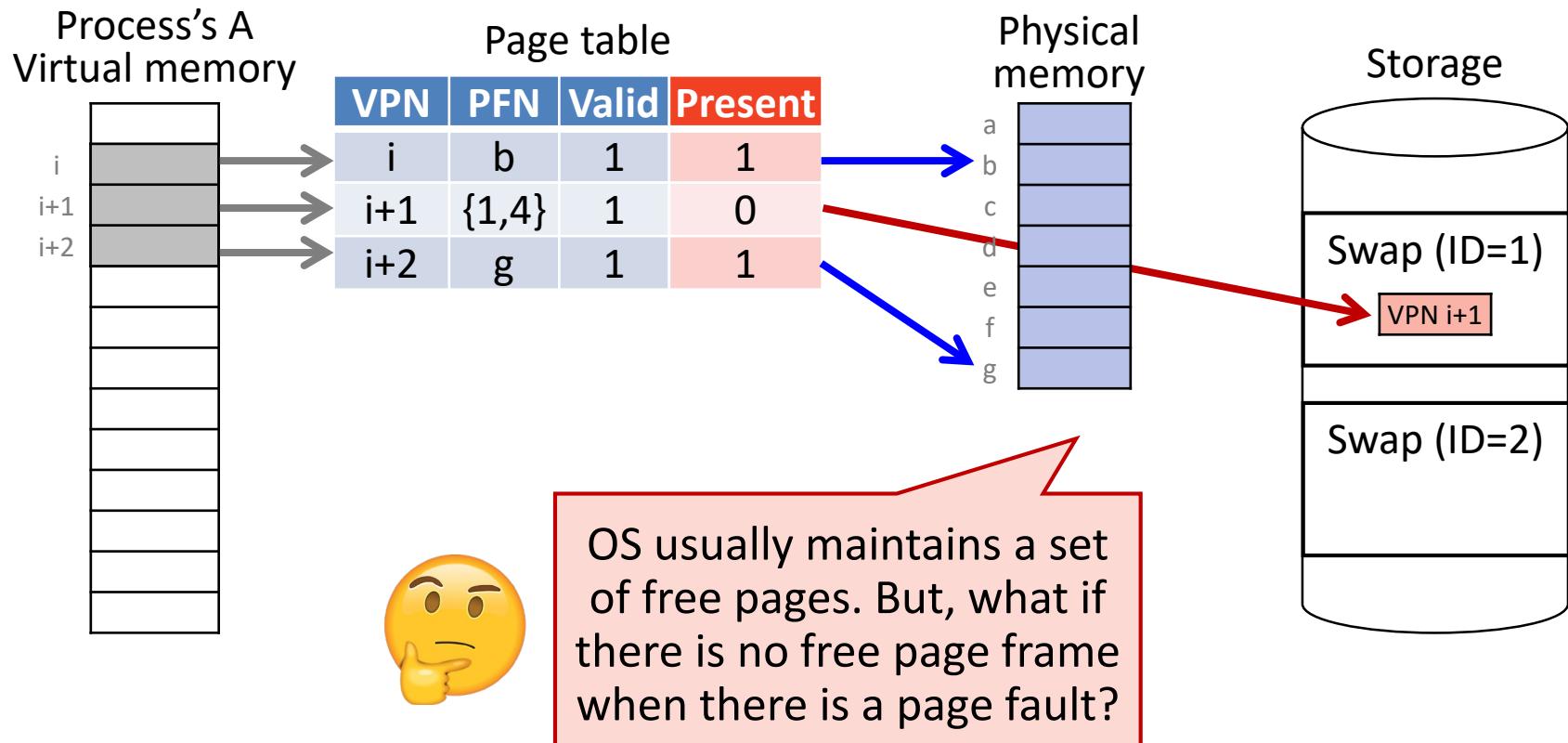


Page Fault: Loads the Faulted Page

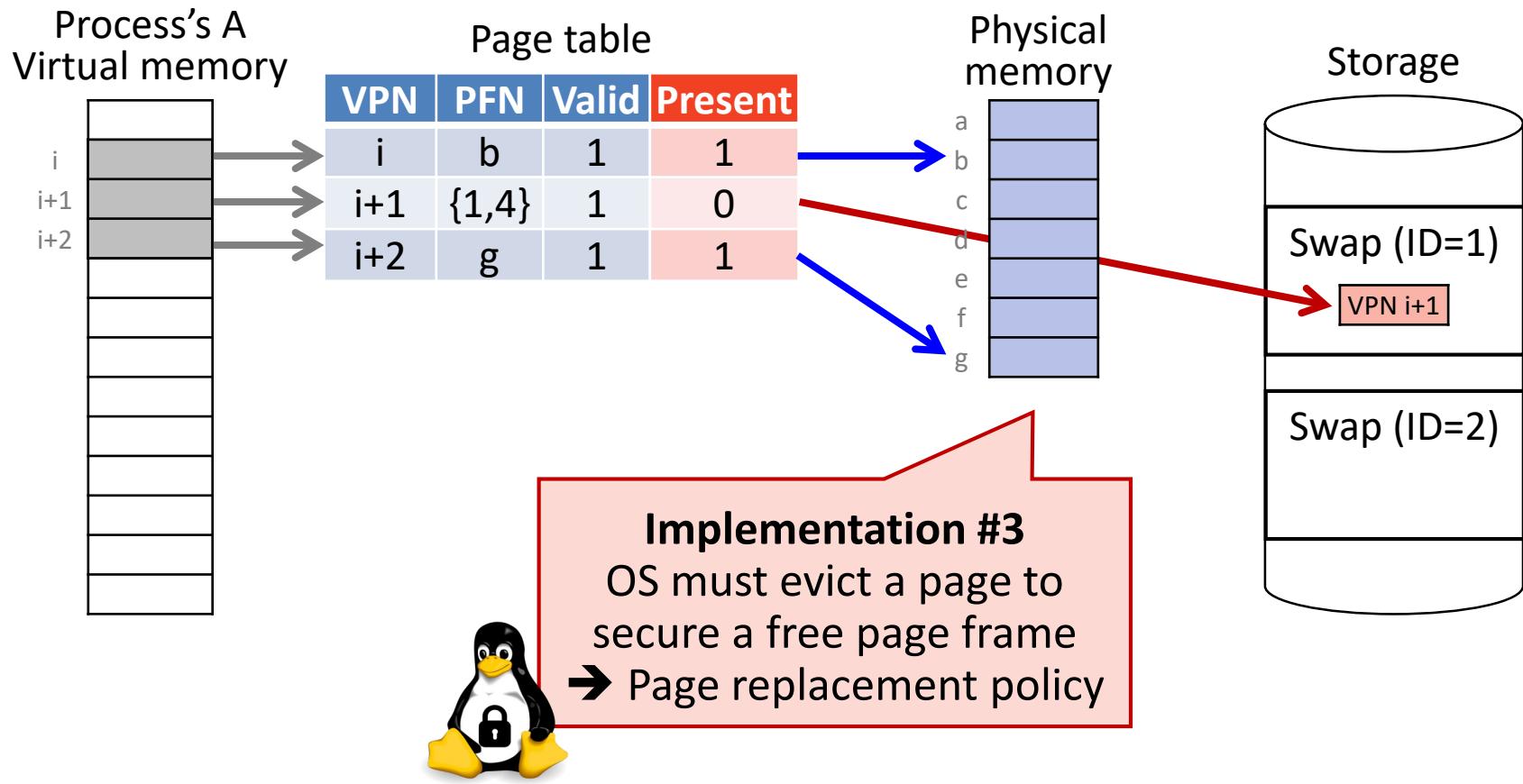


Page fault handler 4 :
Restart the process (Retry
the memory access)

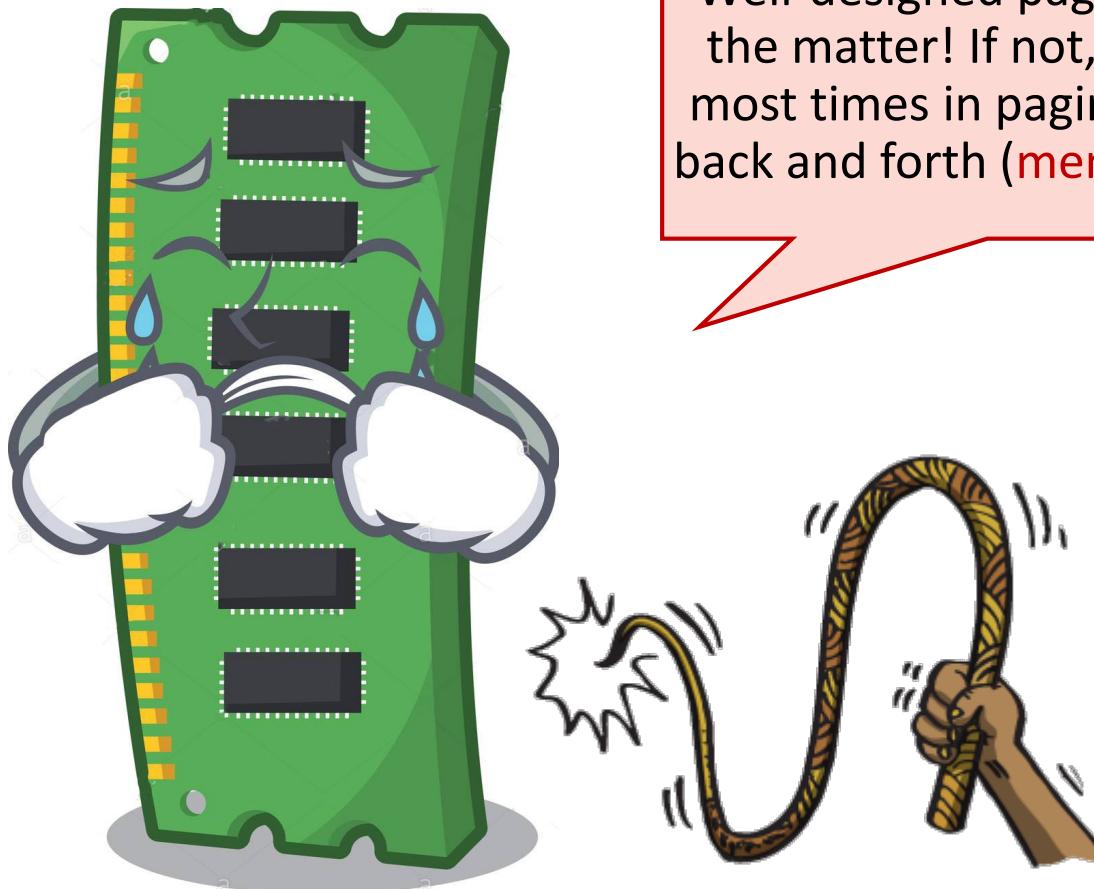
Implementing Swap



Implementing Swap



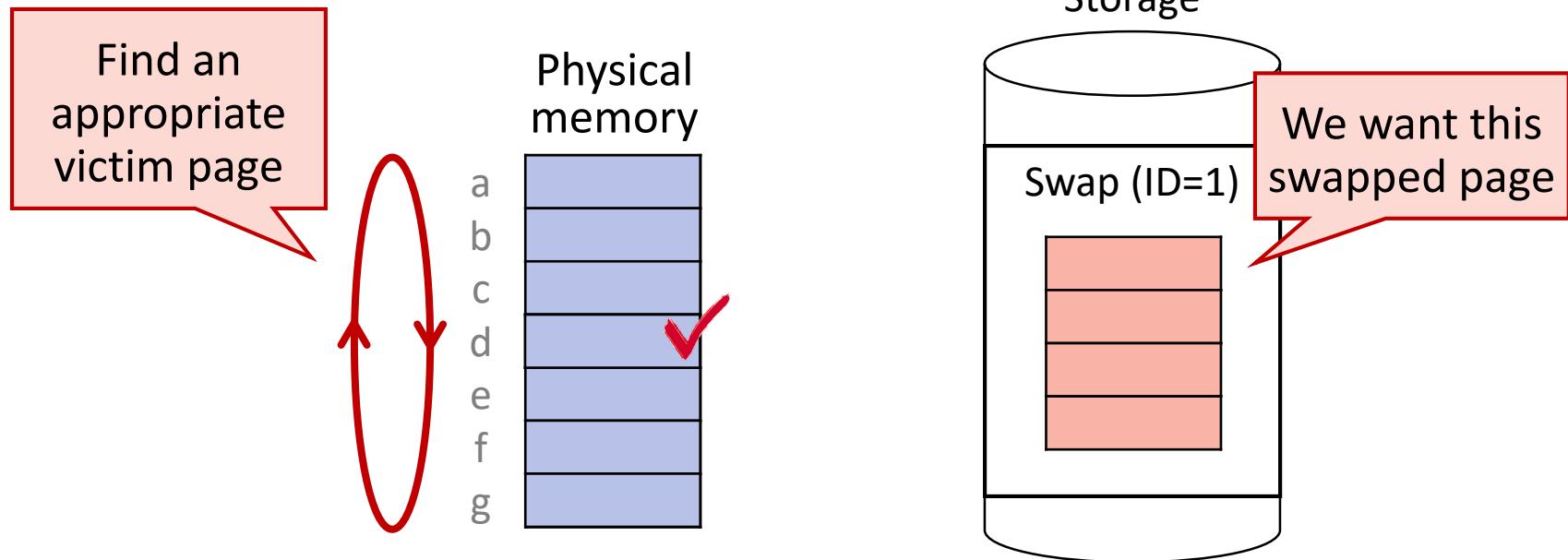
Possible Issue: Memory Thrashing



Well-designed page placement is the matter! If not, the OS spend most times in paging data w/ disk back and forth (**memory thrashing**)

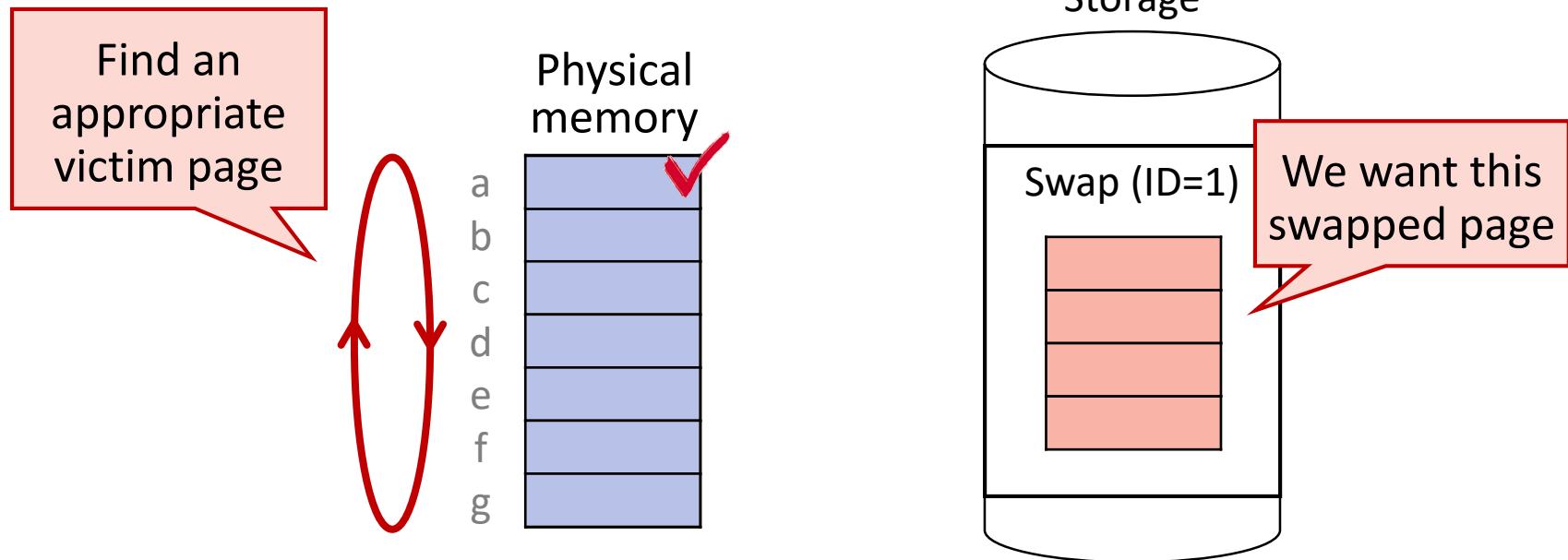


Possible Issue: Memory Thrashing



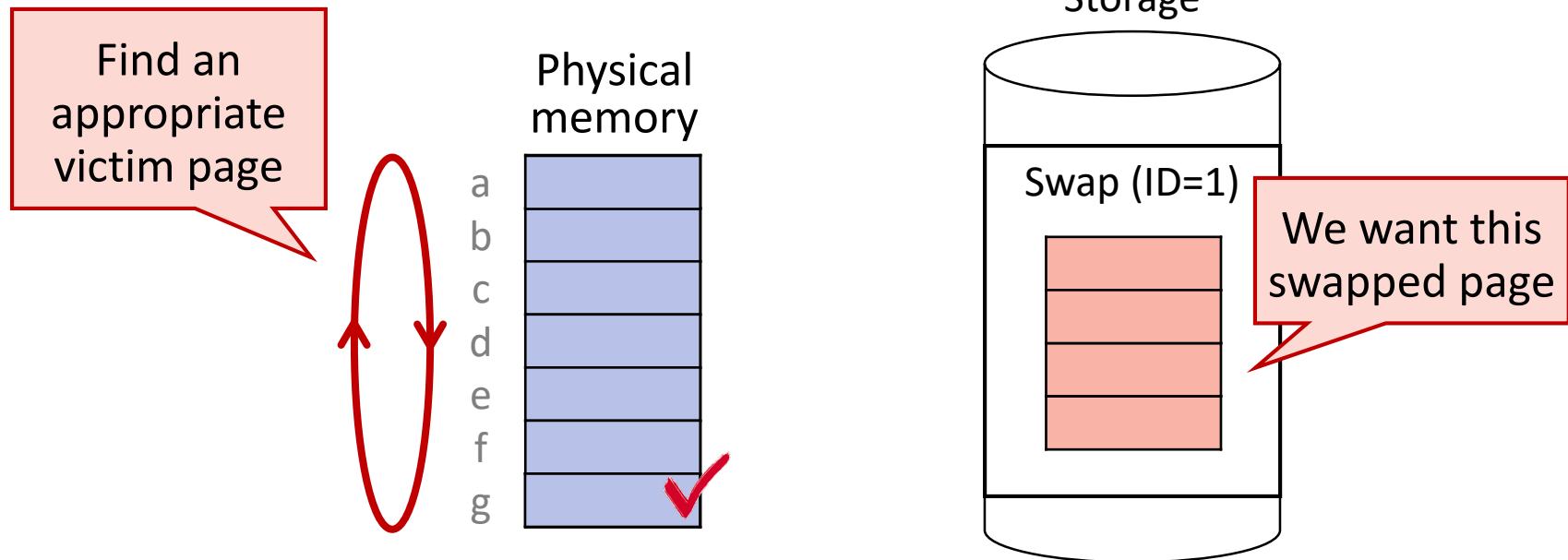


Possible Issue: Memory Thrashing



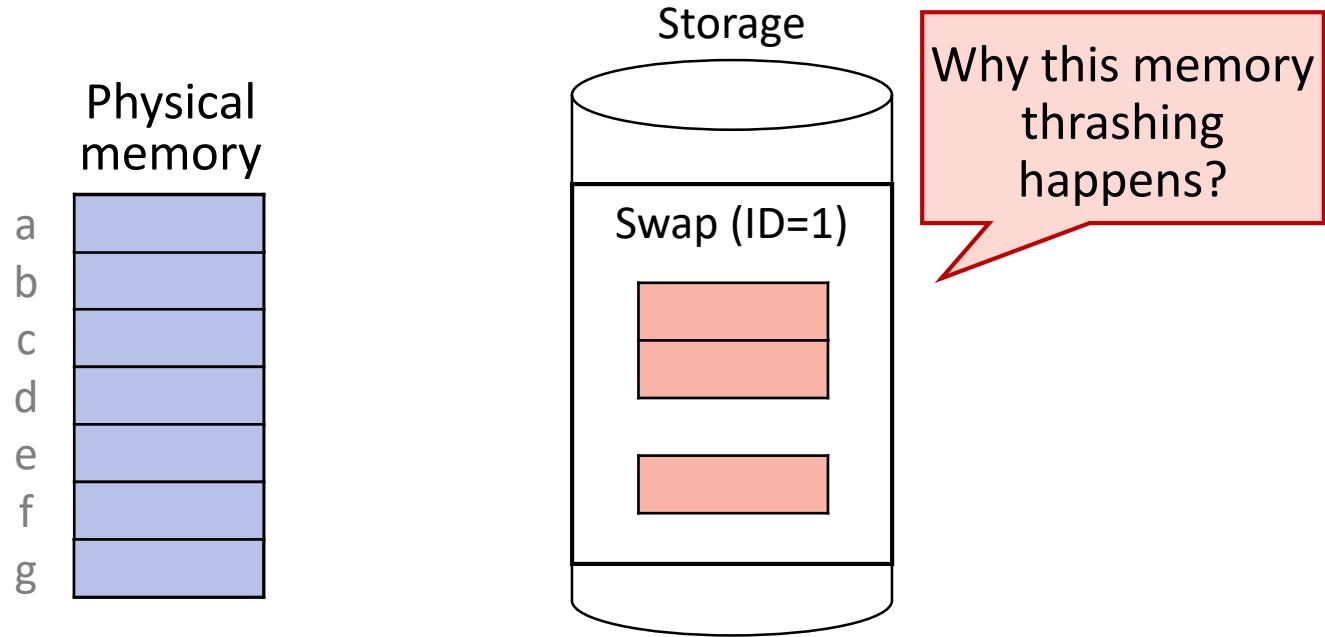


Possible Issue: Memory Thrashing





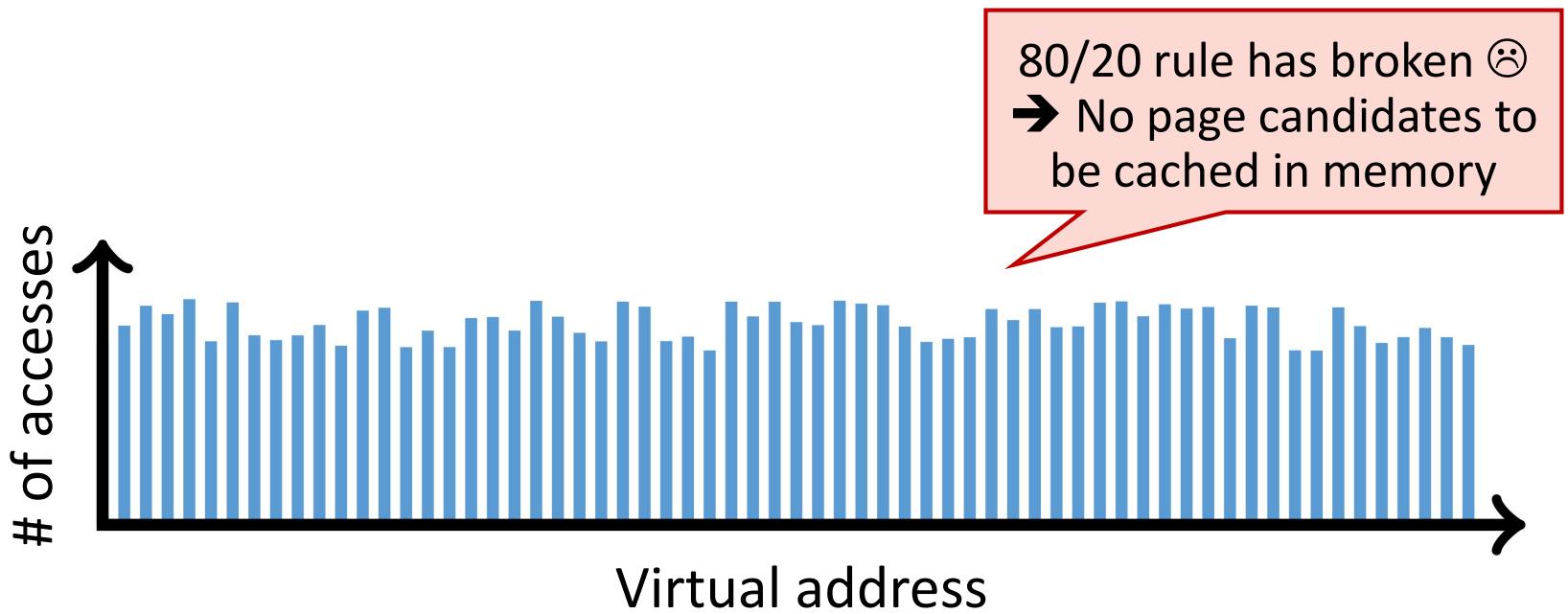
Possible Issue: Memory Thrashing





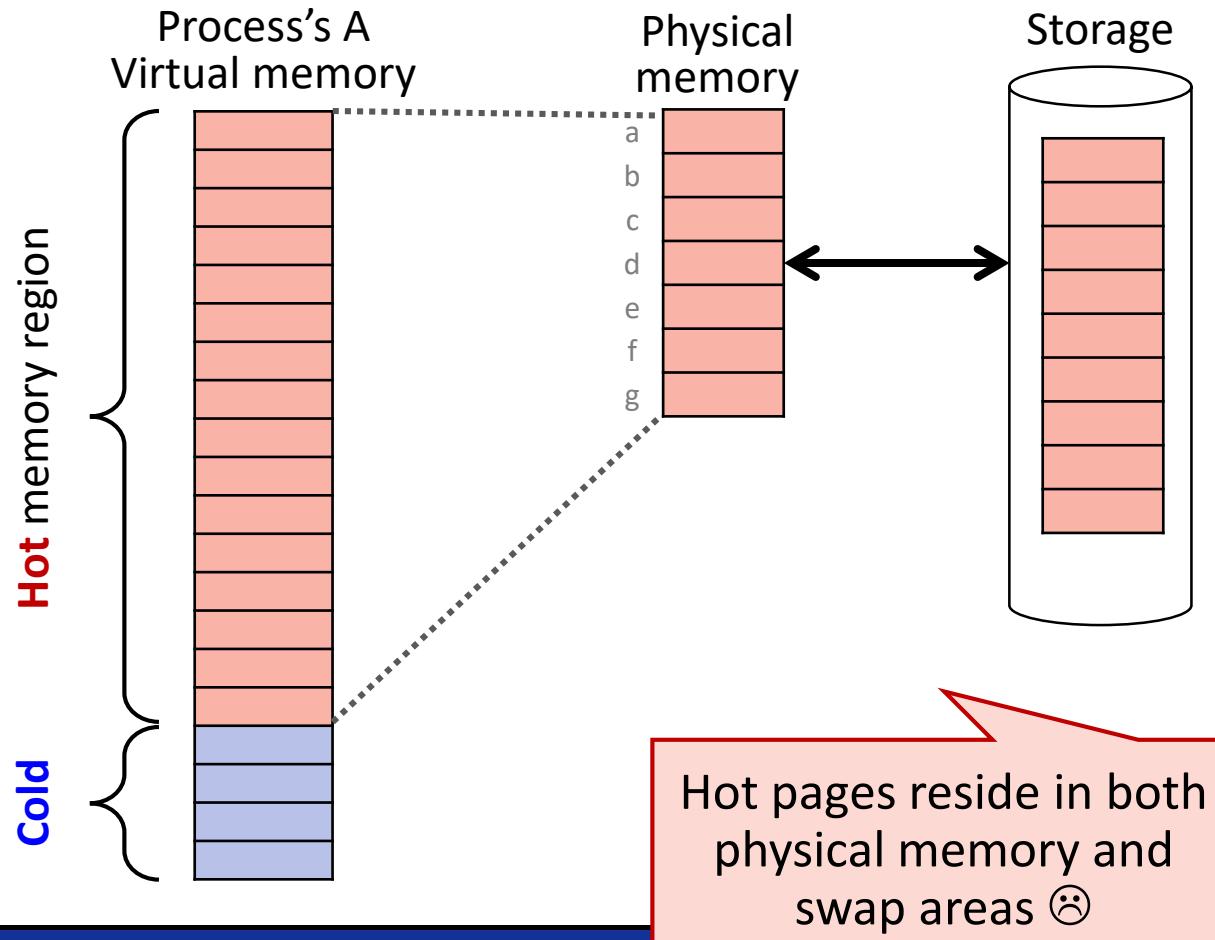
Reasons for Thrashing

- Reason #1: Memory access pattern has no temporal locality (past $\not\approx$ future)



Reasons for Thrashing

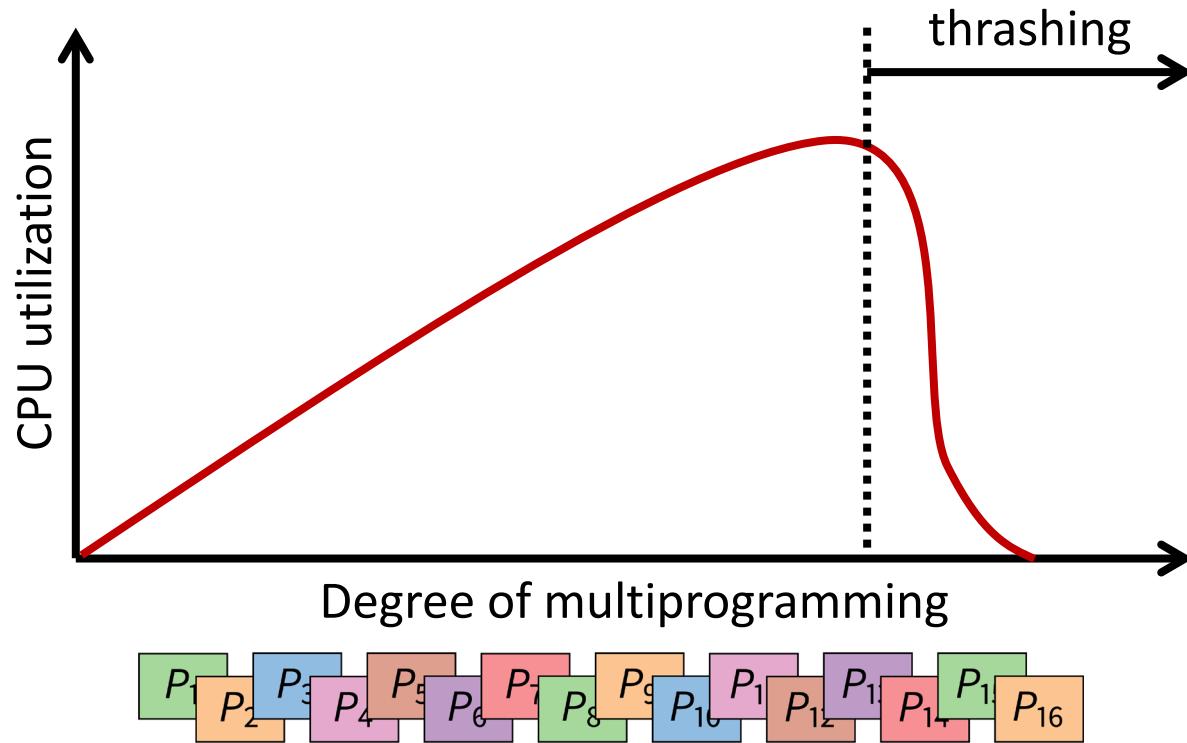
- Reason #2: 80/20 rule works, but hot data do not fit into the physical memory





Reasons for Thrashing

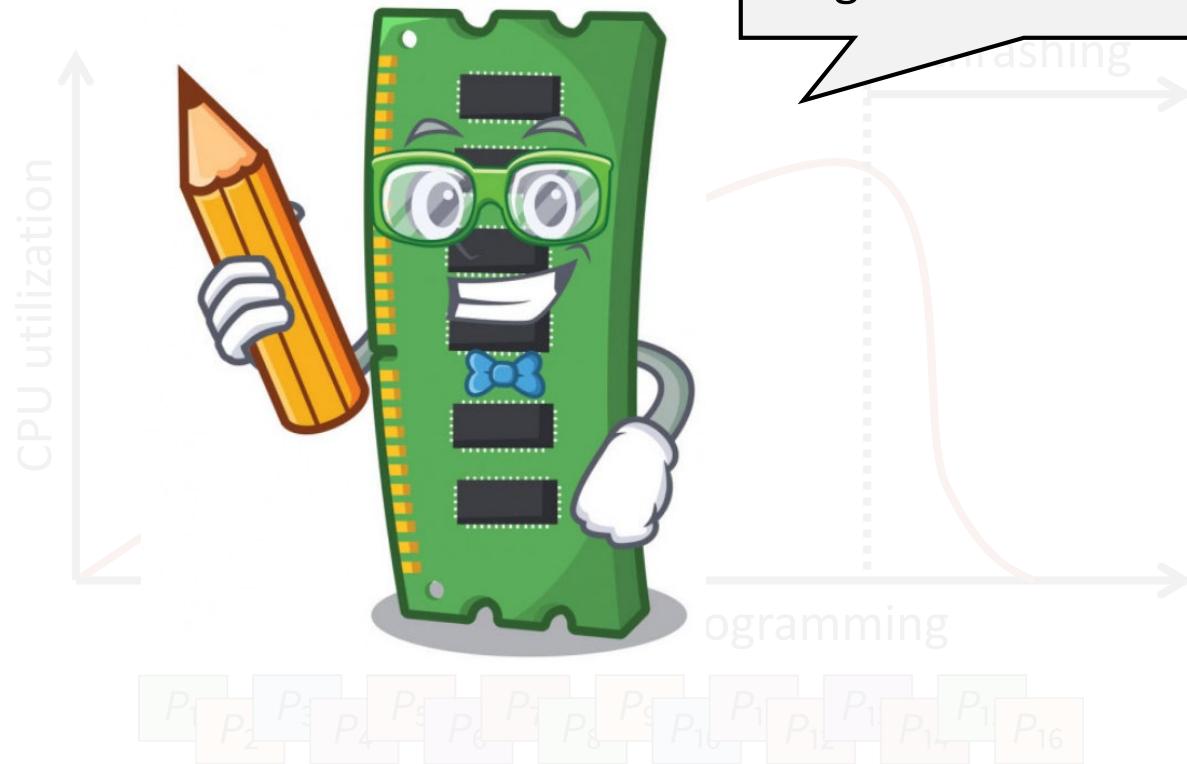
- Reason #3: Each process fits individually, but too many for system



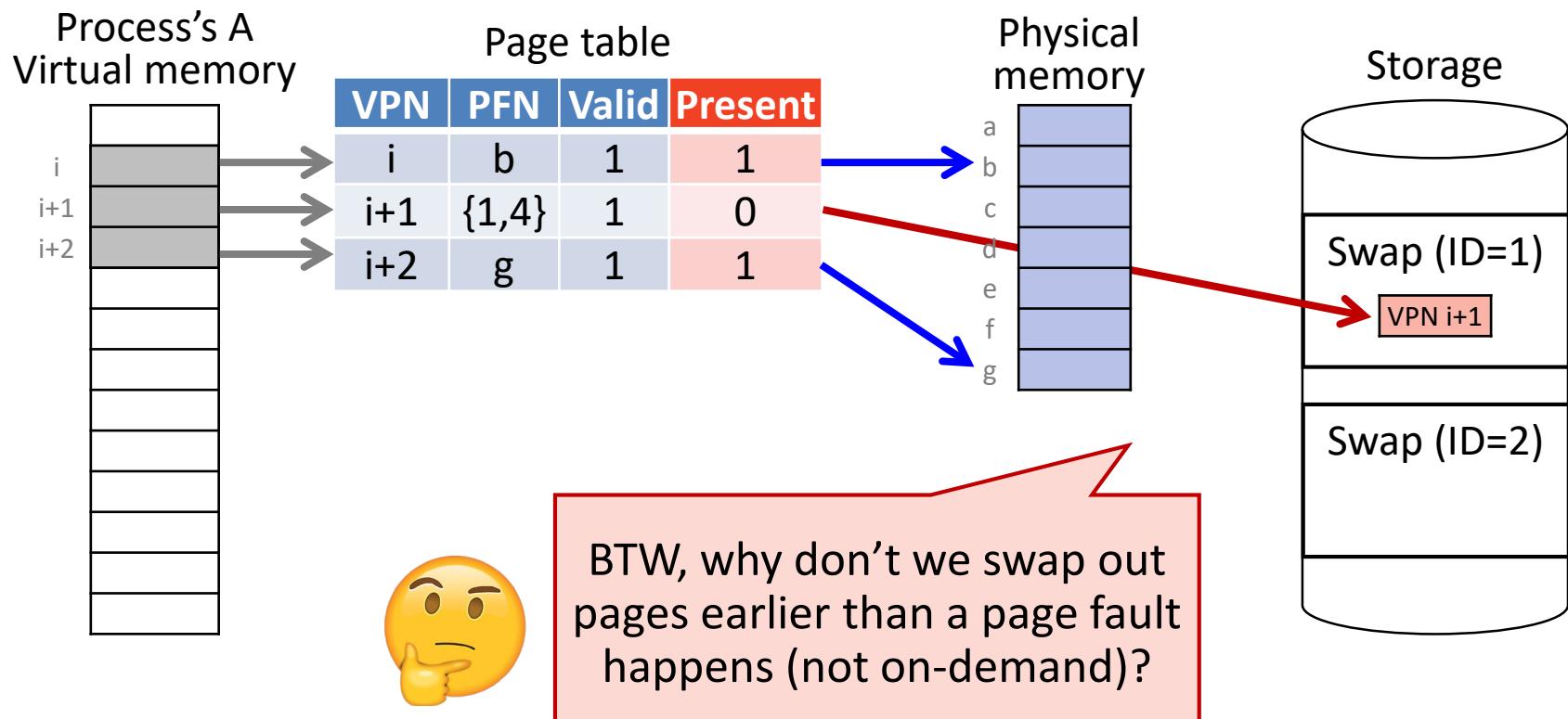
Reasons for Thrashing

- Reason #3: Each process fits individually, but too many for system

Let's survey the various types
of page replacement
algorithms few slides later

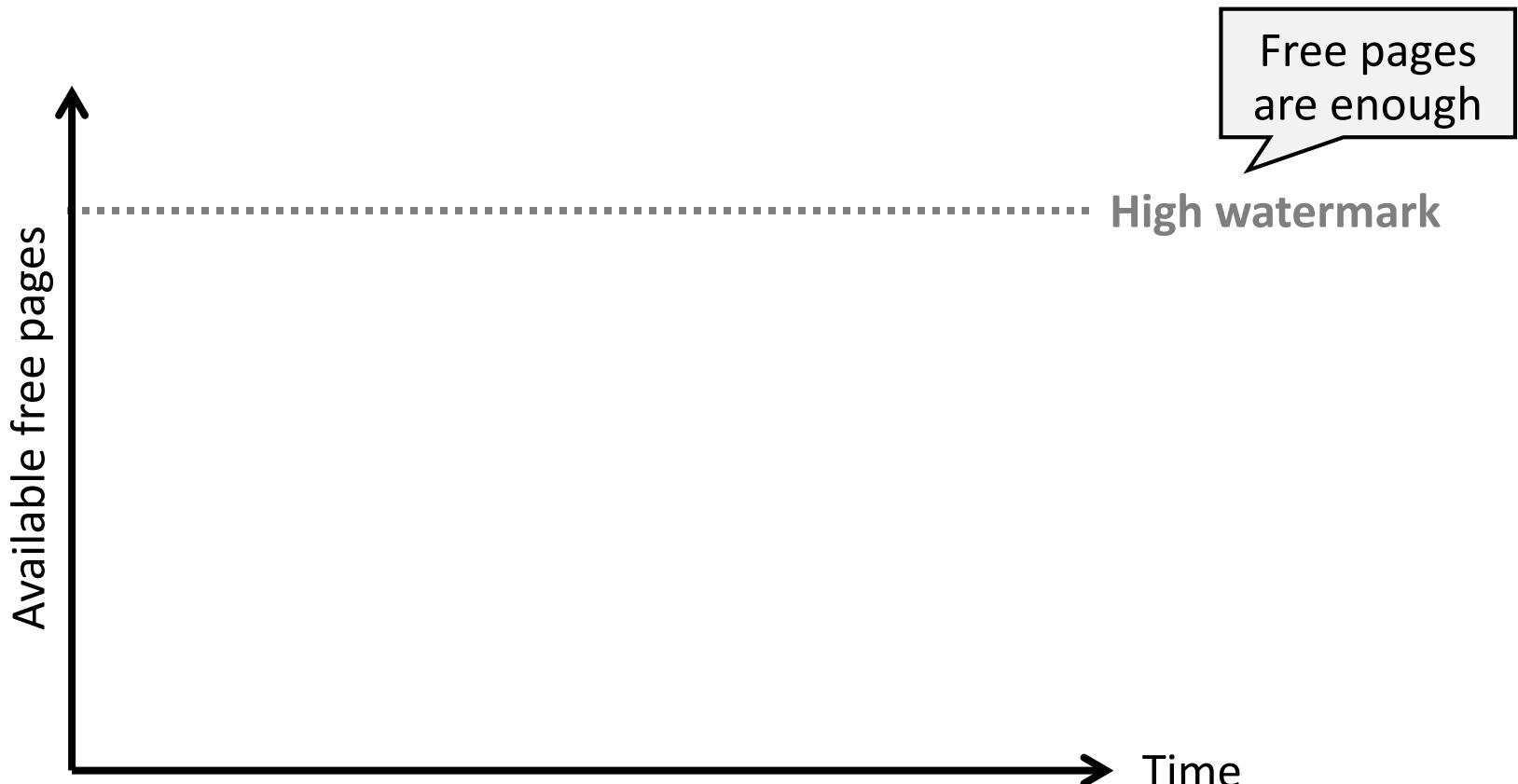


Implementing Swap



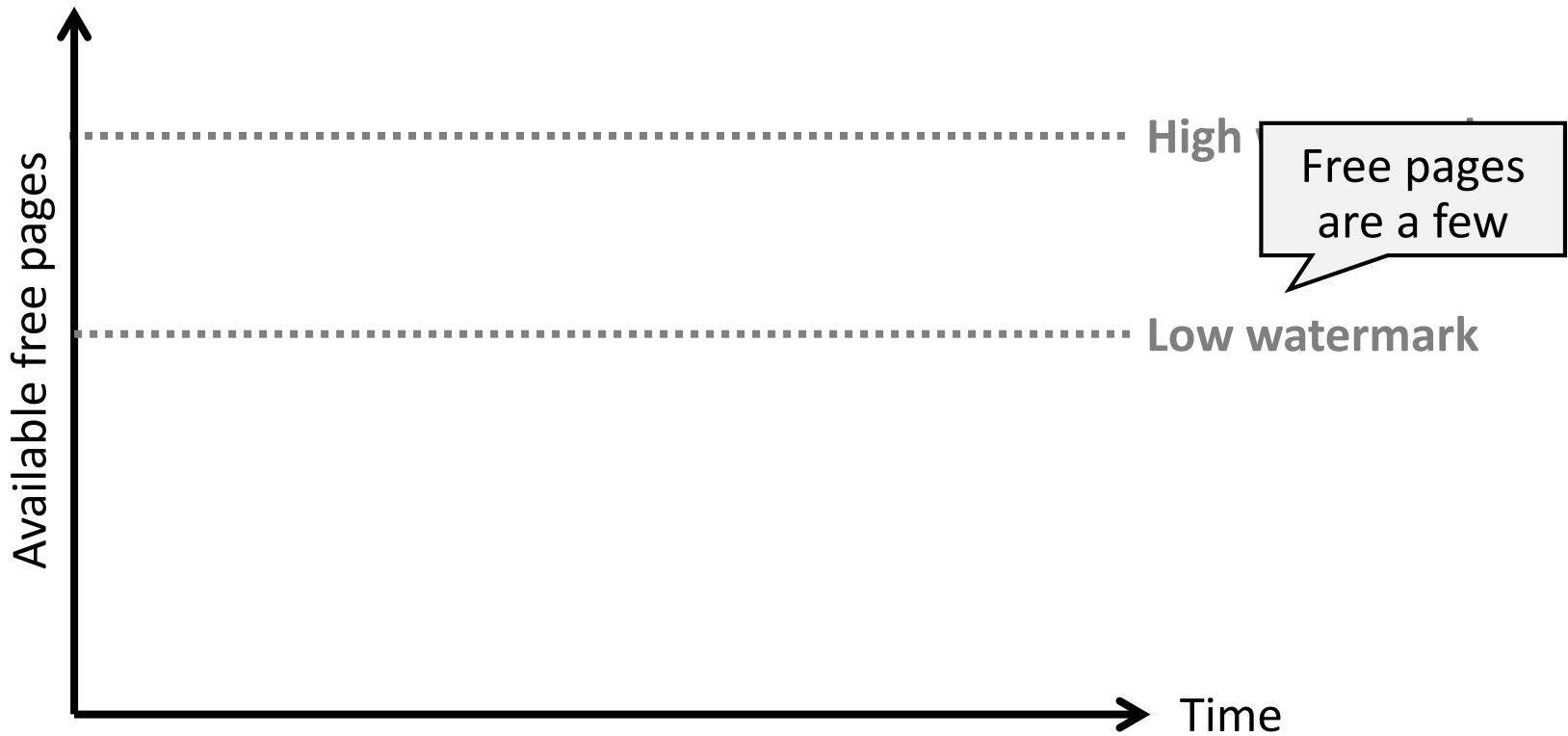
Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exists



Proactive Page Eviction

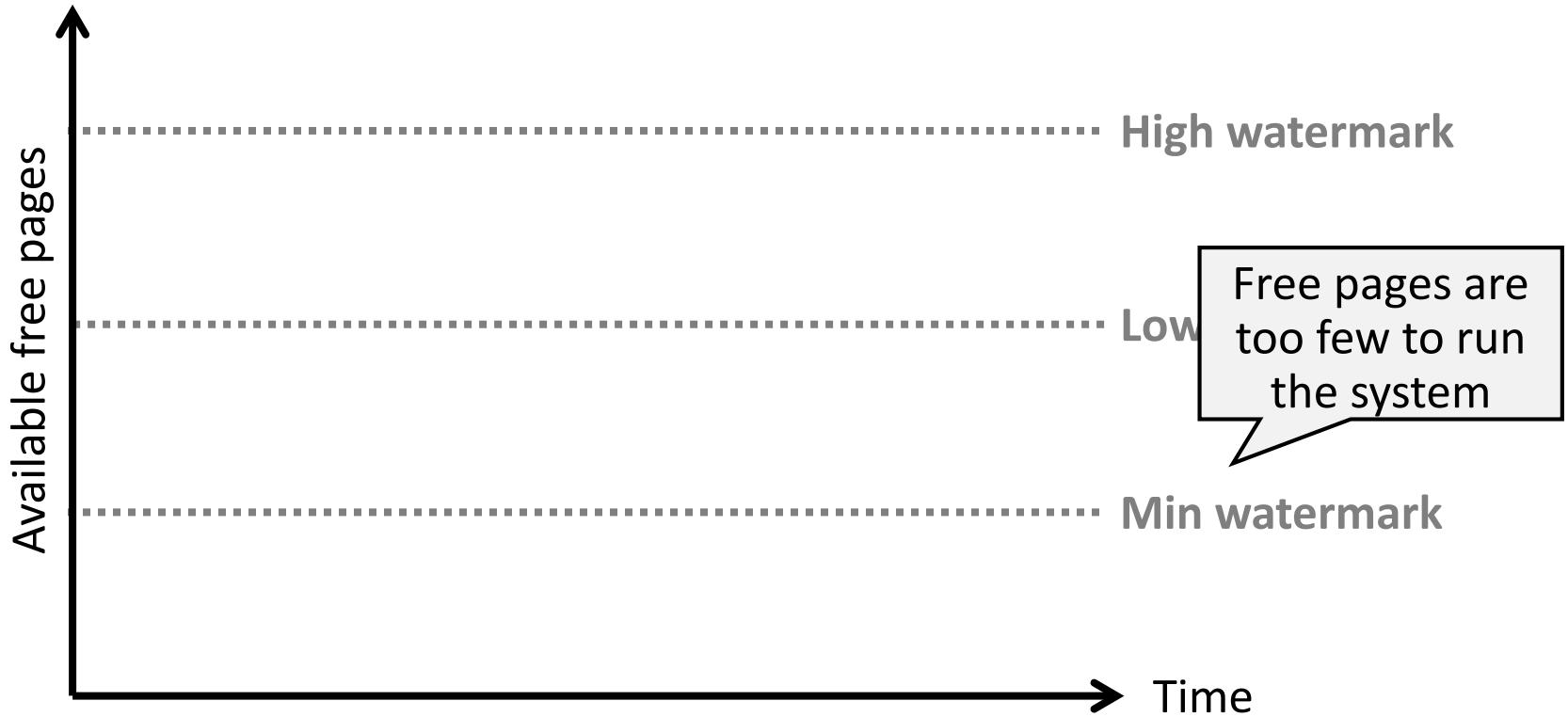
- Most OSes implements “watermark(s)” that indicates how much free page frames exists





Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exists

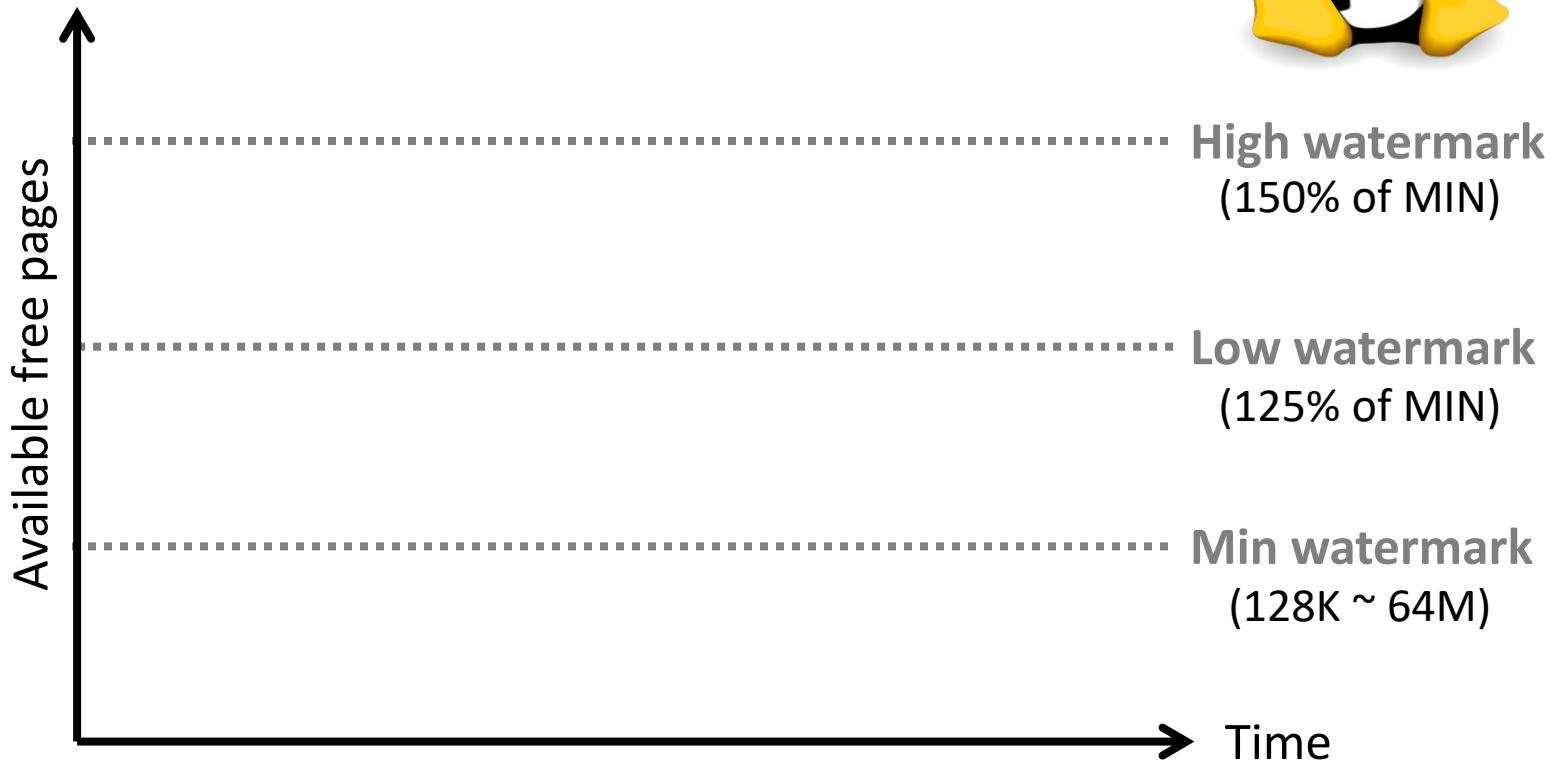
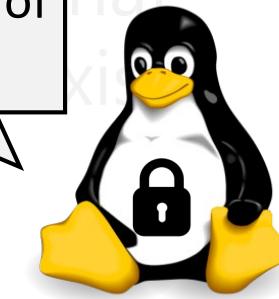




Proactive Page Eviction

- Most OSes implements "watermarks" that indicates how much free pages

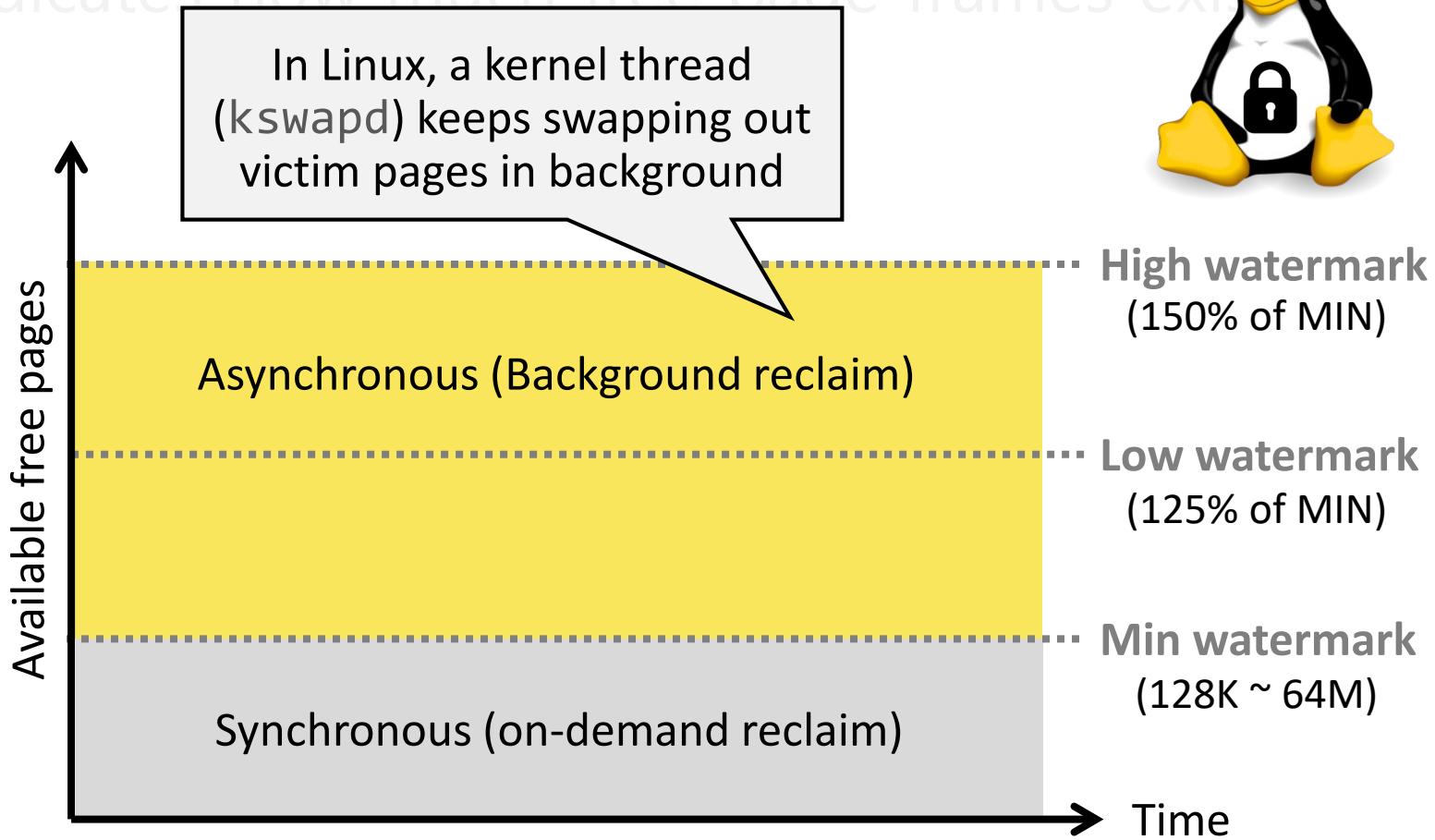
General setting of Linux system





Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

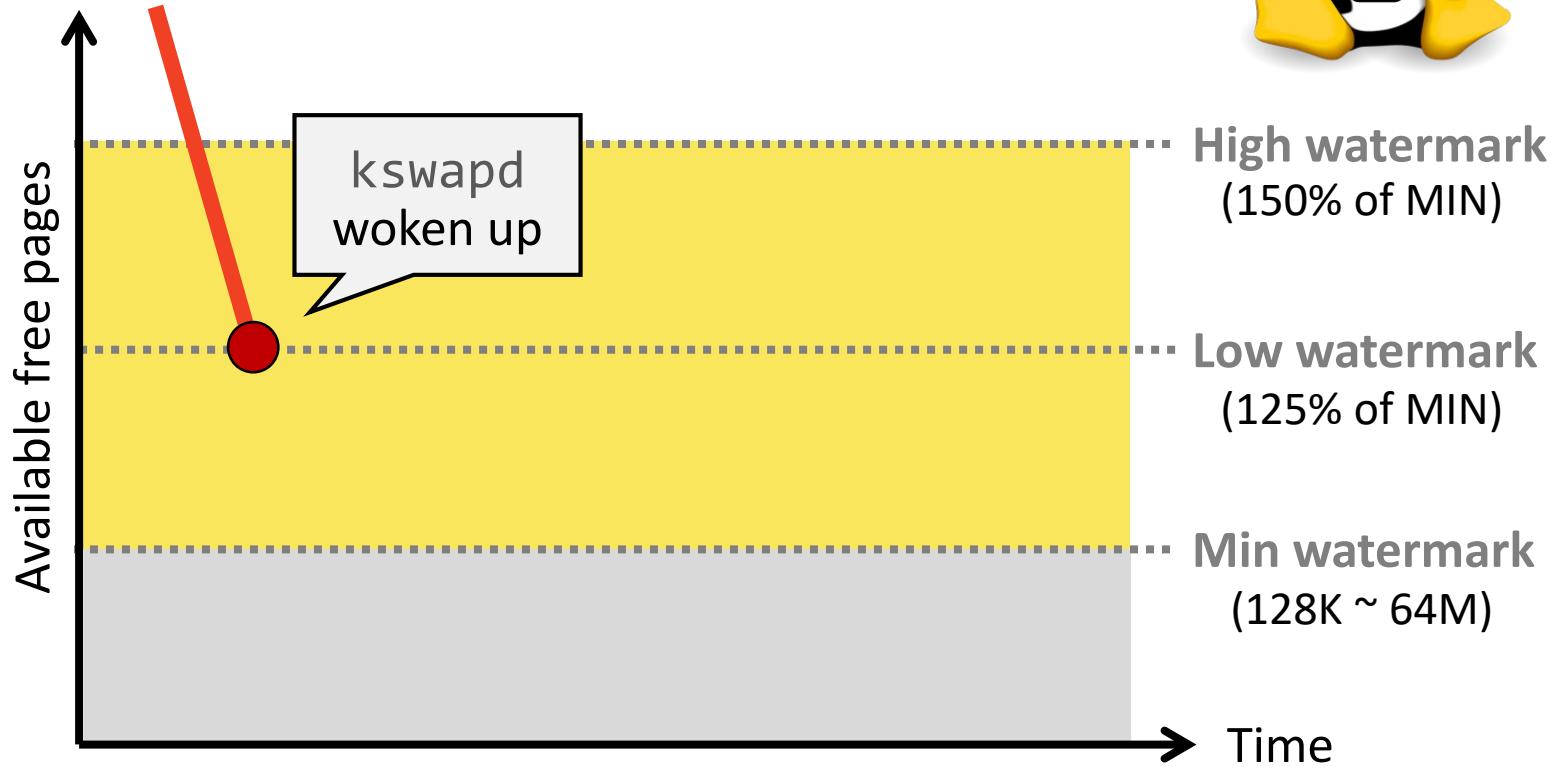




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

Free page change

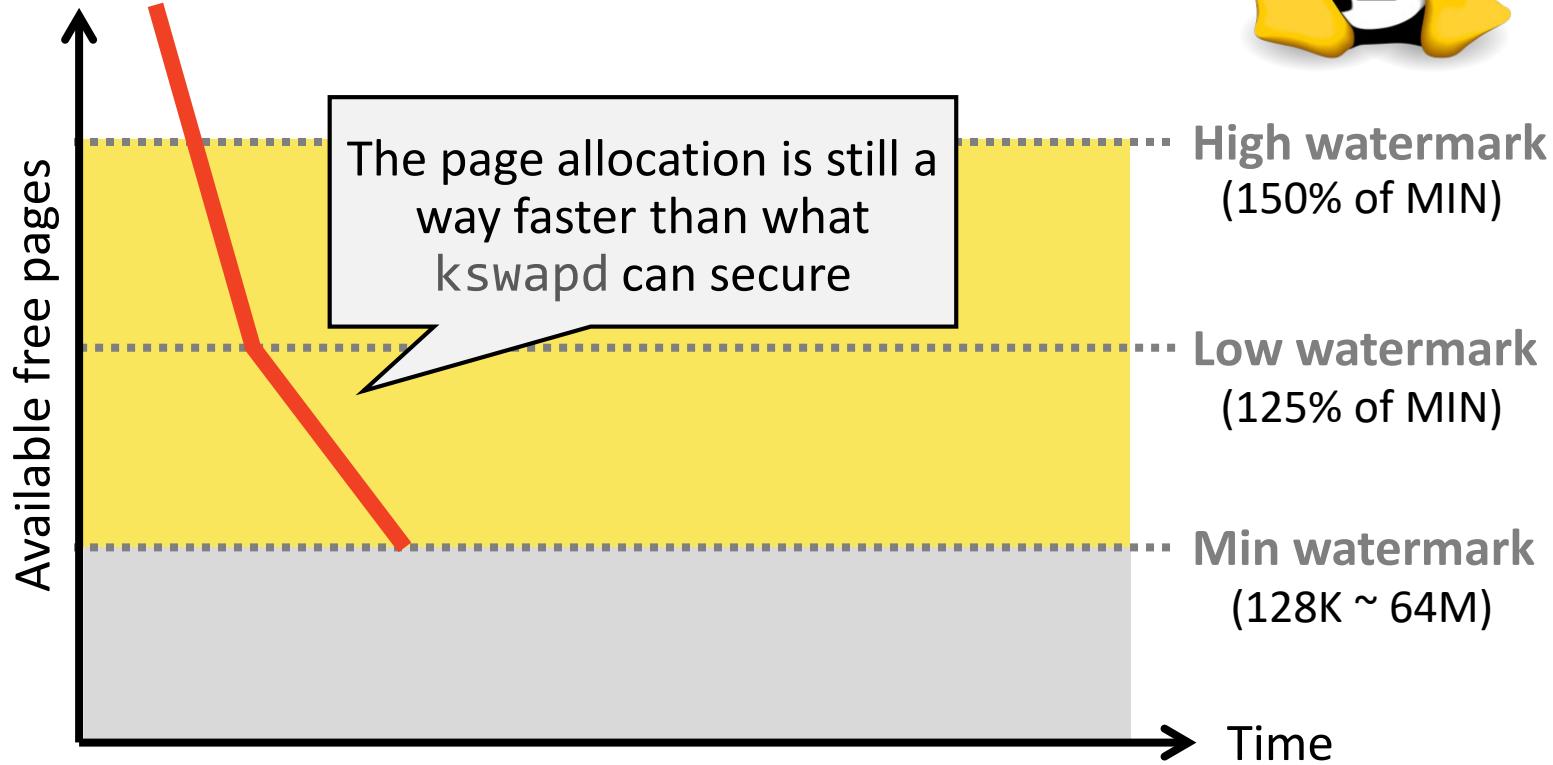




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

Free page change

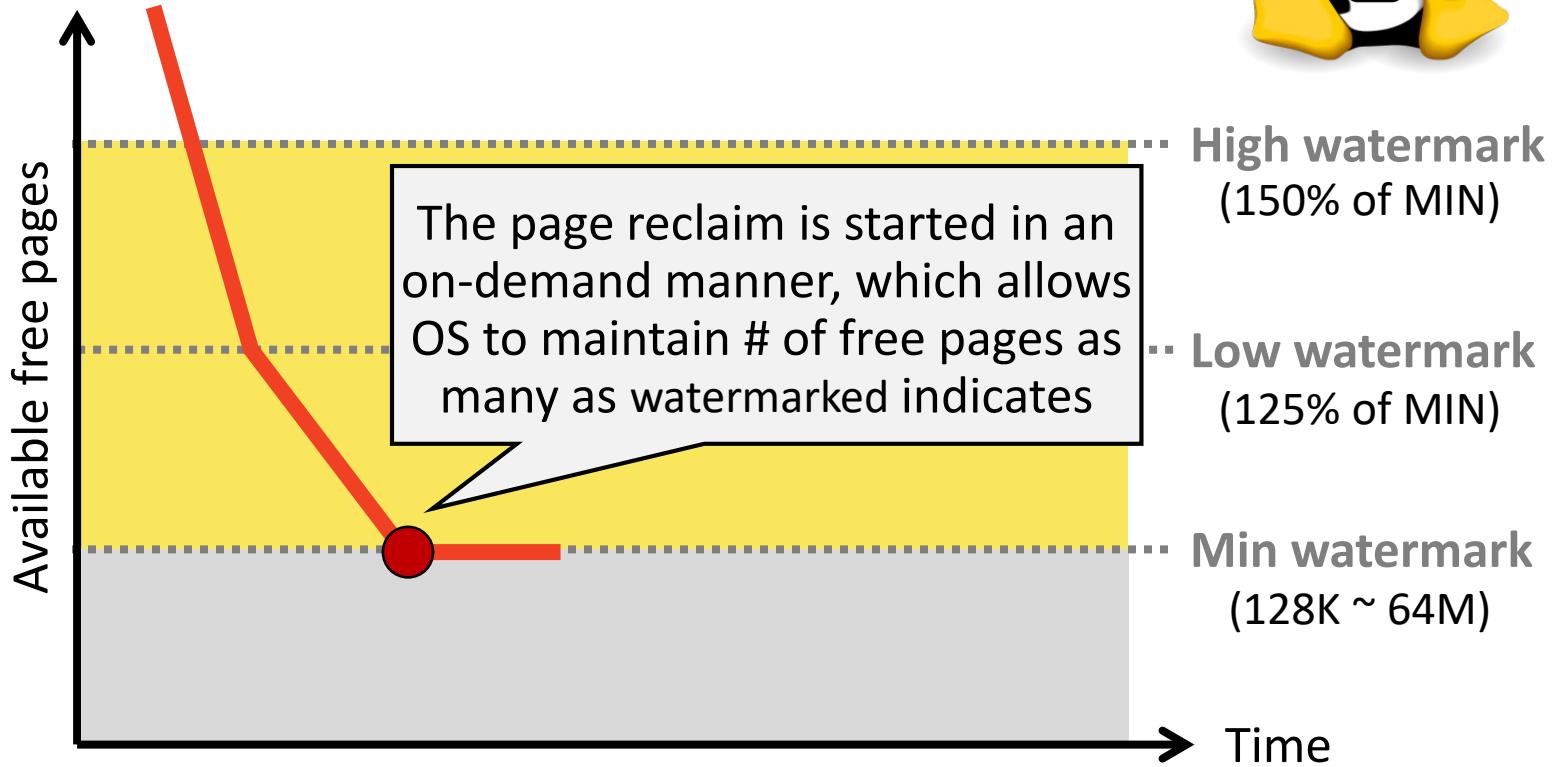




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

Free page change

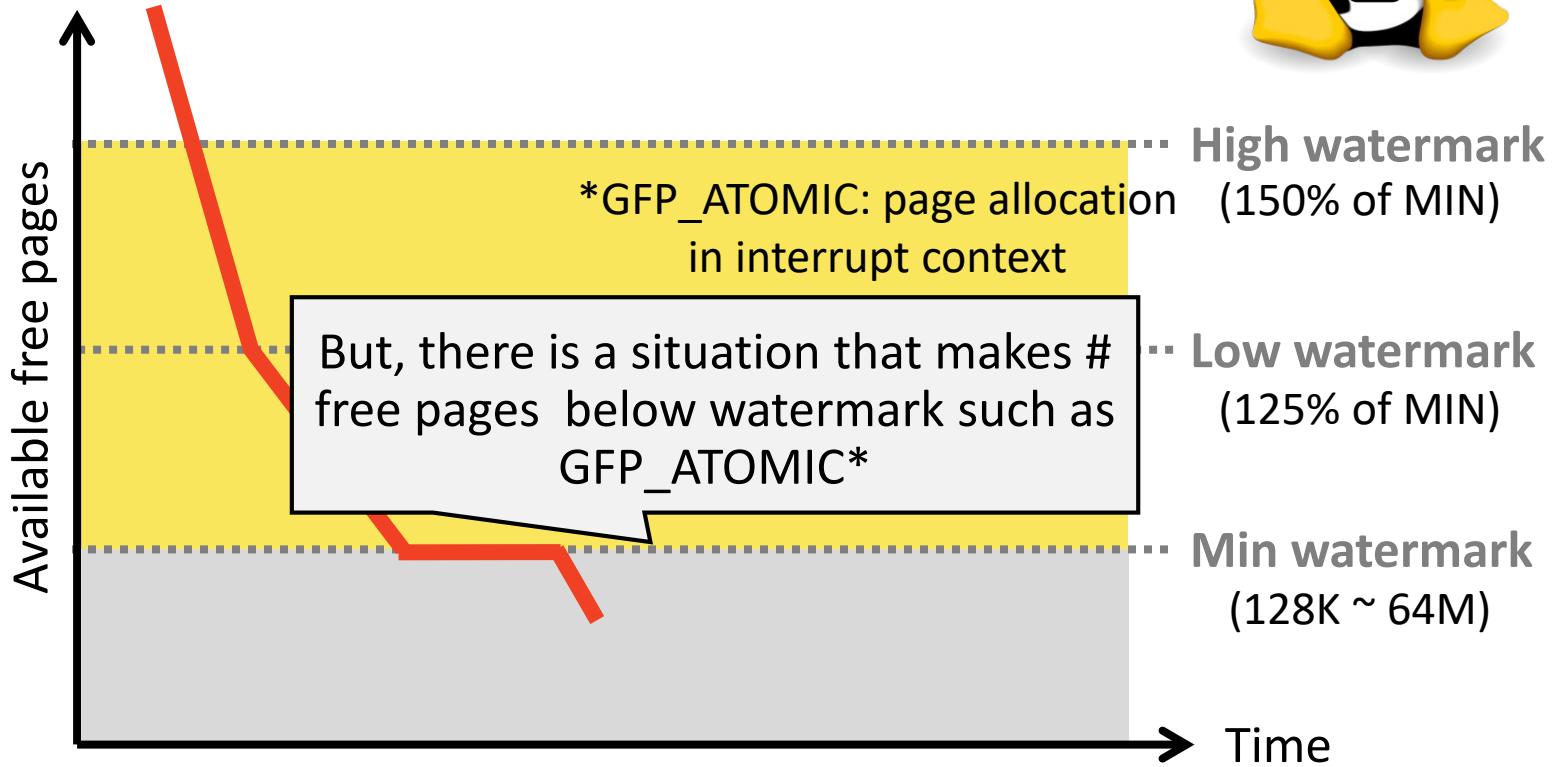




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

Free page change

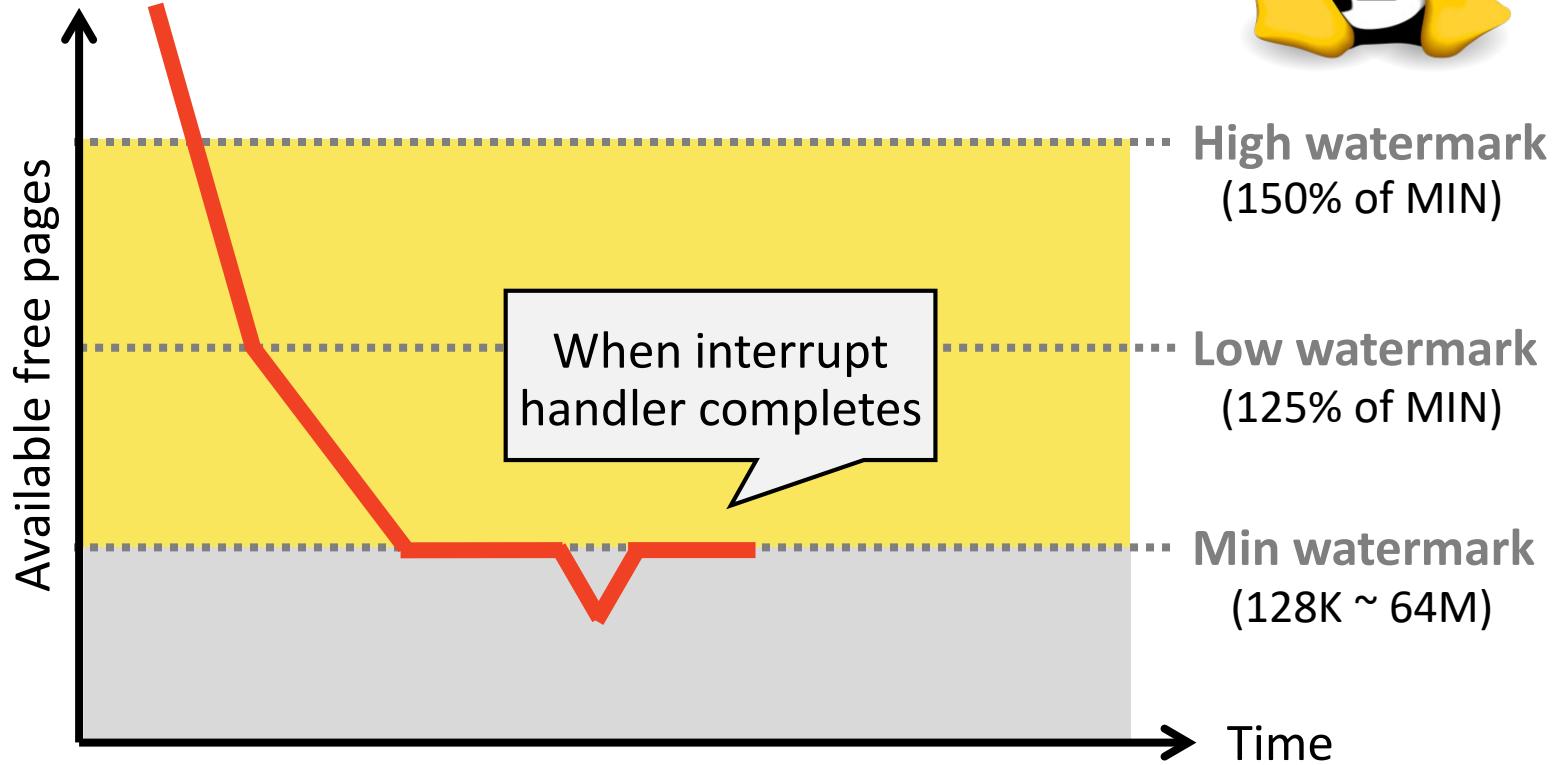




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

Free page change

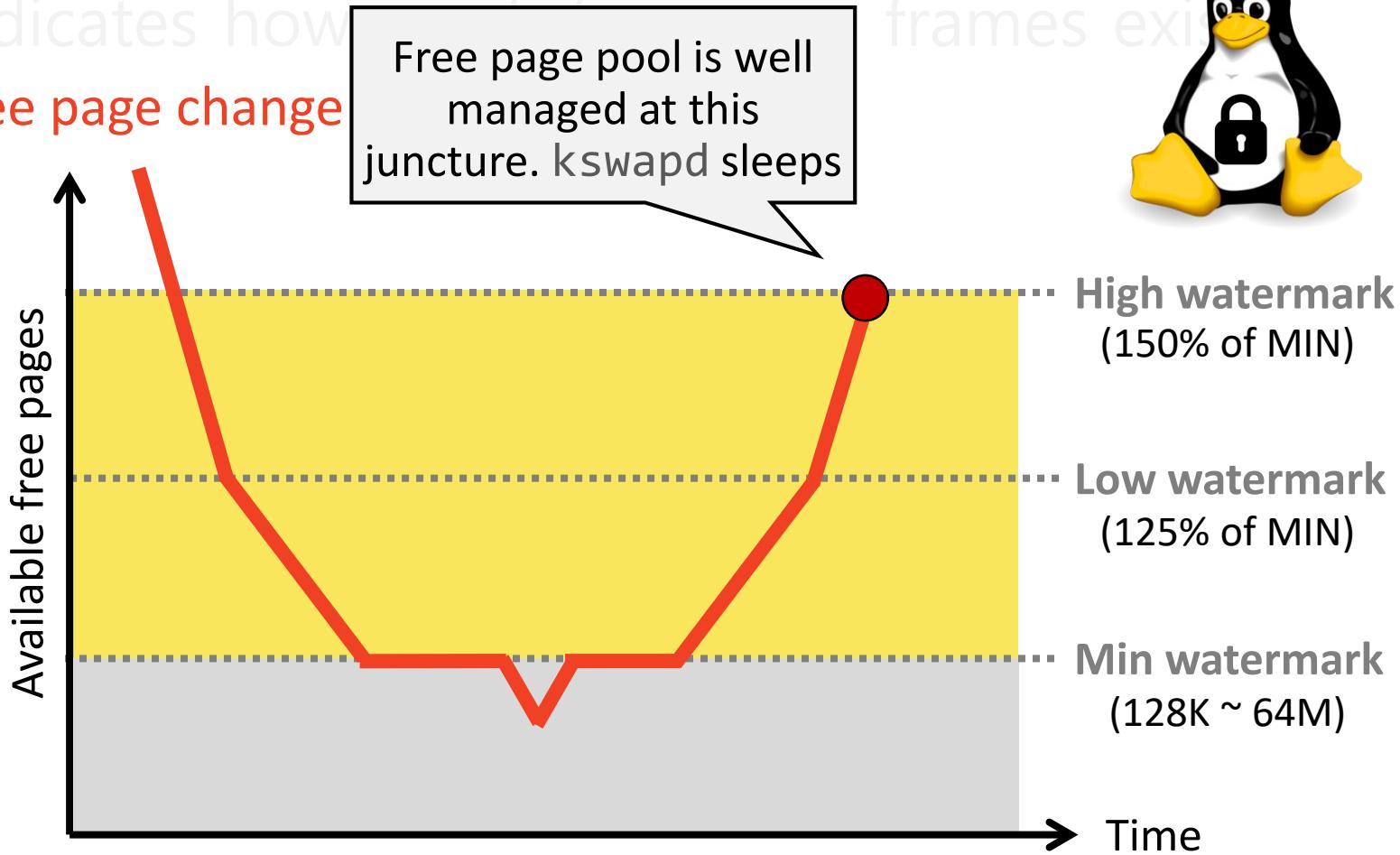




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how many free frames exist.

Free page change

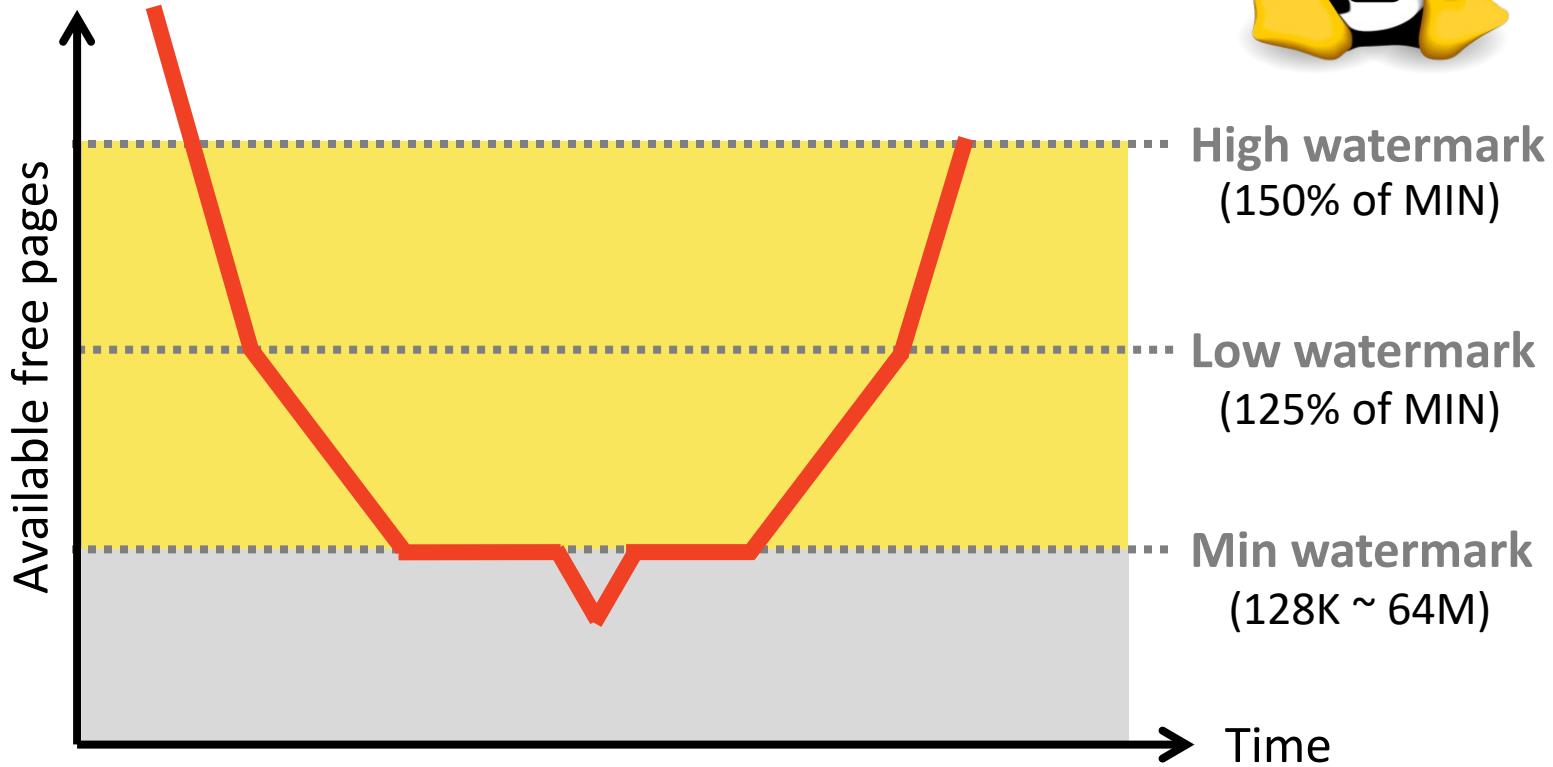




Proactive Page Eviction

- Most OSes implements “watermark(s)” that indicates how much free page frames exist

Free page change



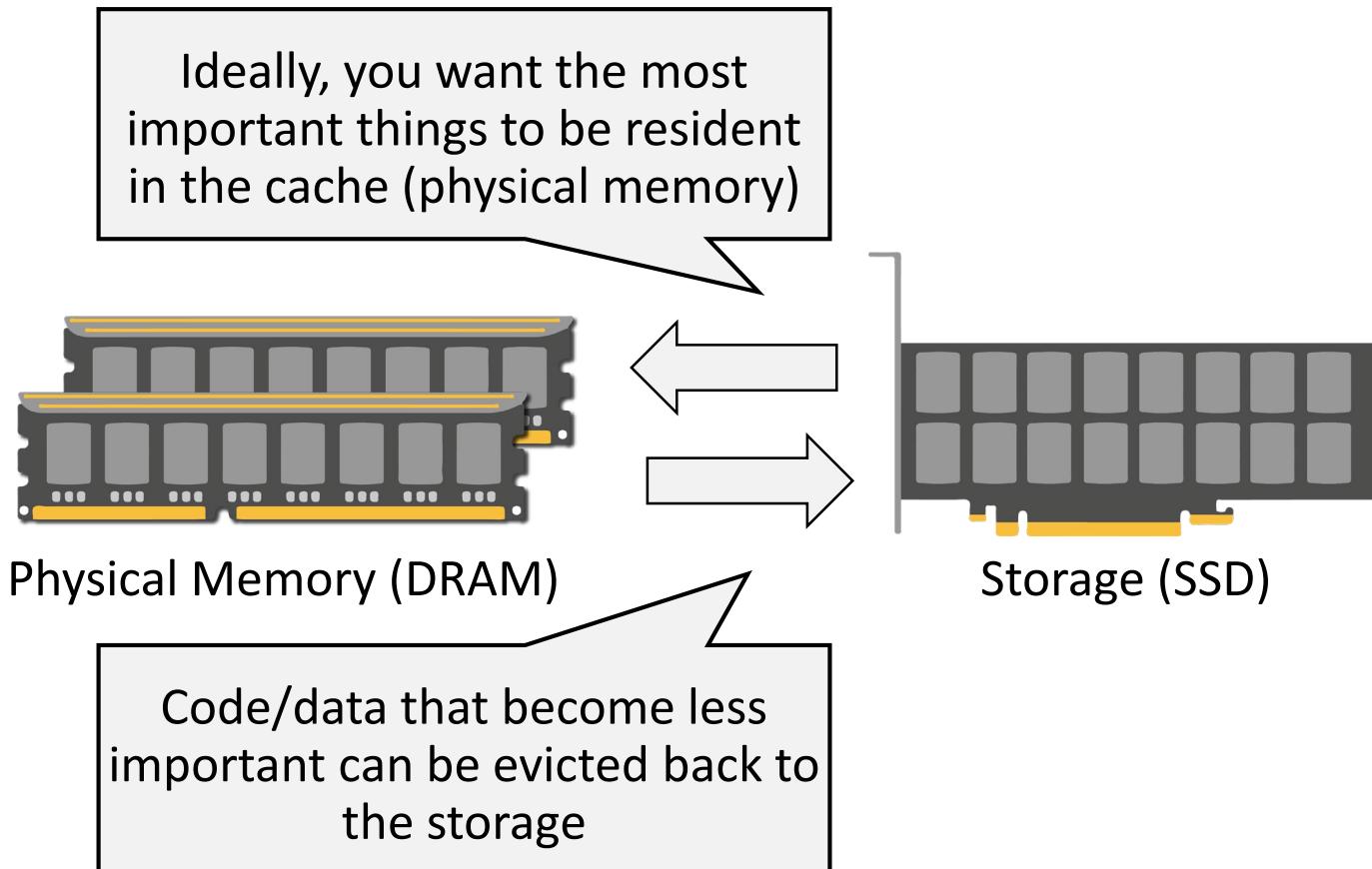
Page Replacement Algorithms



We've covered all of the pieces of paging. Now let's survey the page replacement algorithms

Insight: Memory as a Cache

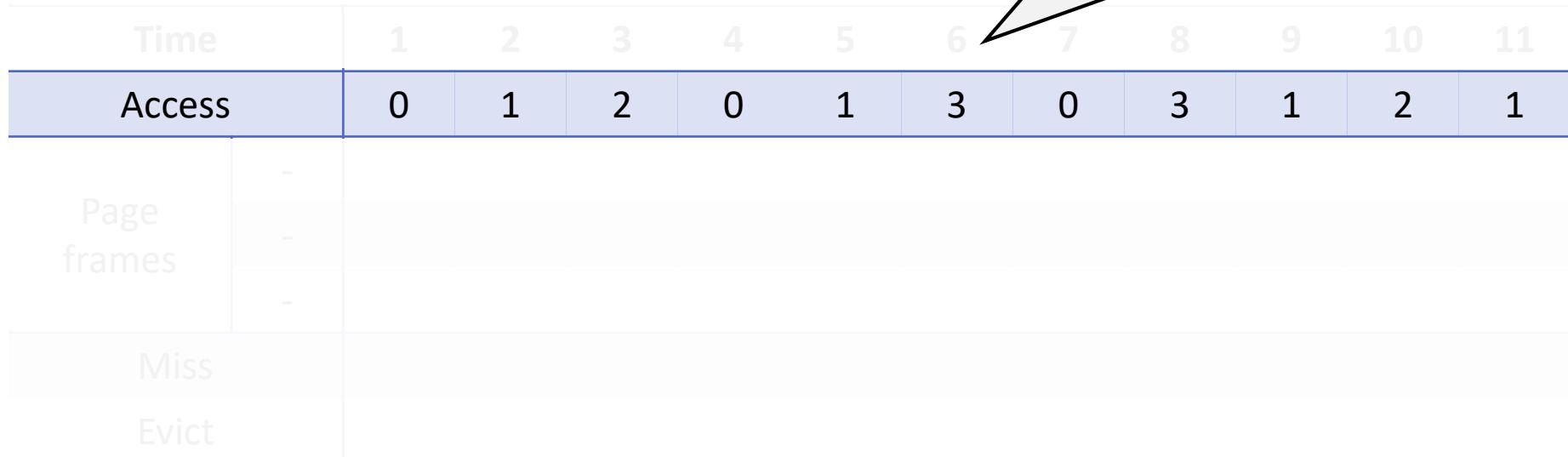
- Physical memory can be viewed as a high-speed cache for your large-but-slow storage



Cache Management

- **Goal:** Picking a replacement policy for the cache
 - To minimize the number of cache misses

Example workload:
number means
accessed VPN



Cache Management

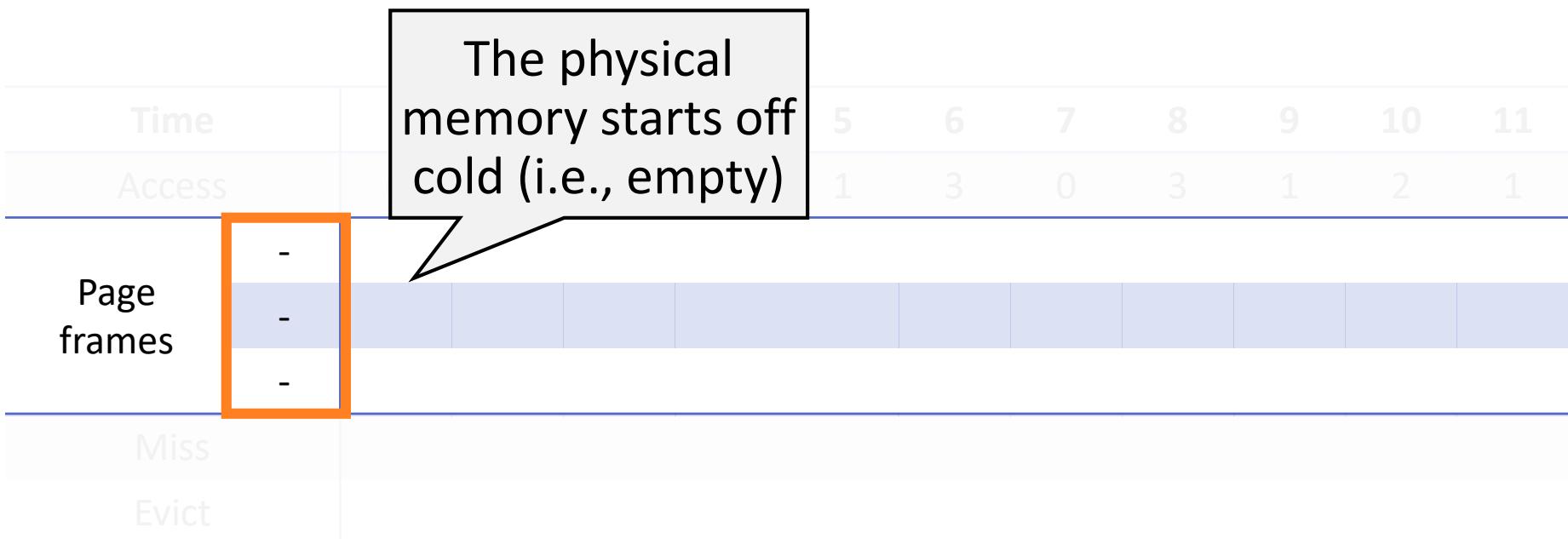
- **Goal:** Picking a replacement policy for the cache
 - To minimize the number of cache misses

Let's assume there are only 3 page frames. And this table shows the mapped VPN

Time	1	2	3	4	5	6	7	8	9	10	11
Access	0	1	2	0	1	3	0	3	1	2	1
Page frames	-	-	-	-	-	-	-	-	-	-	-
Miss	-	-	-	-	-	-	-	-	-	-	-
Evict	-	-	-	-	-	-	-	-	-	-	-

Cache Management

- **Goal:** Picking a replacement policy for the cache
 - To minimize the number of cache misses



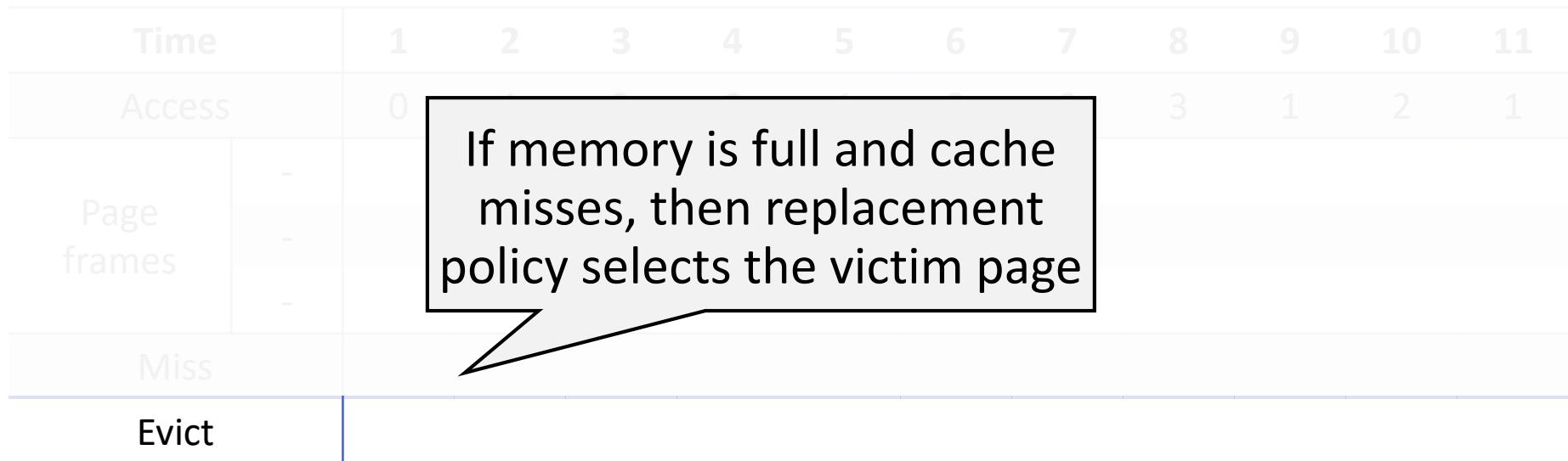
Cache Management

- Goal: Picking a replacement policy for the cache
 - To minimize the number of cache misses

Time	1	0	7	8	9	10	11
Access	0	0	0	3	1	2	1
Page frames	-	-	-	-	-	-	-
Miss	If page is swapped out (valid bit = 0 & present bit = 0), it is cache miss						
Evict							

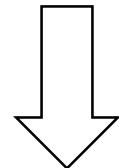
Cache Management

- **Goal:** Picking a replacement policy for the cache
 - To minimize the number of cache misses



► #1: The Optimal Replacement Policy

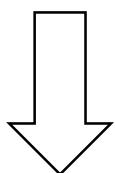
- Idea: replaces the page that will be accessed furthest in the future



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-											
	-											
	-											
Miss												
Evict												

► #1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0								
	-	-	1	1								
	-	-	-	2								
Miss		●	●	●								
Evict												

A red box highlights the entry at index 3 of the page frames row, containing the text "Cold-start cache miss". A red arrow points from this box to the third "●" in the Miss row.

#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future

Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0						
	-	-	1	1	1							
	-	-	-	2	2	2						
Miss		•	•	•								
Evict												

#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future

The diagram illustrates the optimal replacement policy across 11 time steps. A large downward arrow points from the top to the time step 6. The table below tracks page frame usage and identifies a capacity miss at time step 6.

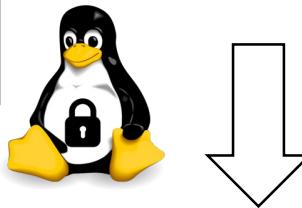
Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0					
	-	-	1	1	1	1	1					
	-	-	-	2	2	2	2					
Miss		●	●	●			●					
Evict												

A red box highlights the entry at time step 6 in the third row of the page frames column, labeled "Capacity miss". A red arrow points from this box to the red dot in the "Miss" row at time step 6.

#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future

Finding victim page with optimal replacement policy

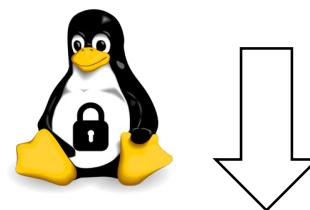


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0	-	-	-	-	-
	-	-	1	1	1	1	1	-	-	-	-	-
	-	-	-	2	2	2	2	-	-	-	-	-
Miss		●	●	●				●				
Evict												

Page 2 will be accessed furthest in the future

#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0					
	-	-	1	1	1	1	1					
	-	-	-	2	2	2	3					
Miss		•	•	•			•					
Evict							2					

Evict Page 2 and reads Page 3 from storage

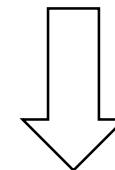
#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future

Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0	0	0	0	0	0
	-	-	1	1	1	1	1	1	1	1	1	1
	-	-	-	2	2	2	3	3	3	3	3	3
Miss		●	●	●			●					
Evict									2			

#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future

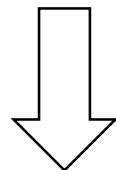


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0	0	0	0	0	
	-	-	1	1	1	1	1	1	1		1	
	-	-	-	2	2	2	3	3		2		
Miss		●	●	●			●			●		
Evict							2			3		

Capacity miss

#1: The Optimal Replacement Policy

- Idea: replaces the page that will be accessed furthest in the future



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0	0	0	0	0	0
	-	-	1	1	1	1	1	1	1	1	1	1
	-	-	-	2	2	2	3	3	3	3	2	2
Miss		●	●	●			●					
Evict									2			
											Cache hit	



#1: The Optimal Replacement Policy

- Idea: replaces the page that is furthest in the future



This policy is optimal (6 hits). But we have to know the future ☹ -- Is there any **practical** policies?

Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0	0	0	0	0	0	0	0	0	0
	-	-	1	1	1	1	1	1	1	1	1	1
	-	-	-	2	2	2	3	3	3	3	2	2
Miss		•	•	•				•			•	
Evict							2			3		



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



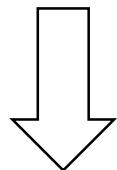
Time	1	2	3	4	5	6	7	8	9	10	11
Access	0	1	2	0	1	3	0	3	1	2	1
Page frames	-	-	-	-	-	-	-	-	-	-	-
Miss											
Evict											

Think about this as a queue



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

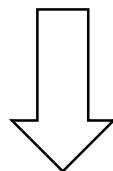


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0										
	-	-										
	-	-										
Miss		●										
Evict												



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

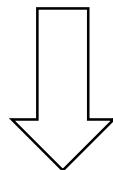


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0									
	-	-	-									
	-	-										
Miss		●	●									
Evict												



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

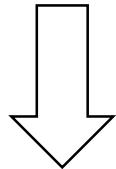


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1									
	-	-	0									
	-	-	-									
Miss		●	●									
Evict												



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

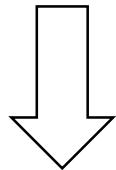


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	1								
	-	-	0	0								
	-	-	-	-	-							
Miss		●	●	●								
Evict												



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2								
	-	-	0	1								
	-	-	-	0								
Miss		●	●	●								
Evict												



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

The diagram illustrates the FIFO (First-In First-Out) cache replacement policy using a timeline table. A large downward arrow is positioned above the table, indicating the progression of time from left to right.

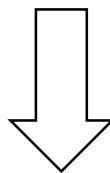
Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2						
	-	-	0	1	1	1						
	-	-	-	0	0	0						
Miss		•	•	•								
Evict												

A blue box labeled "Cache hit" is drawn around the third column of the "Access" row, indicating a hit in the cache. Blue circles highlight the values in the "Page frames" row at time steps 5 and 4, which are both circled in blue, indicating they are being considered for eviction. A blue arrow points from the "Cache hit" box to the value "0" at time step 4 in the "Page frames" row.



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	2					
	-	-	0	1	1	1	1					
	-	-	-	0	0	0	0					
Miss		●	●	●			●					
Evict												

A red box highlights the entry at time 6 in the "Page frames" row, labeled "Capacity miss". A red arrow points from this box to the red dot in the "Miss" row at the same time step.



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

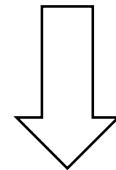


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	2					
	-	-	0	1	1	1	1					
	-	-	-	0	0	0	0					
Miss		•	•	•			•					
Evict												



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

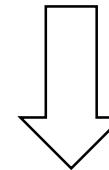


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3					
	-	-	0	1	1	1	2					
	-	-	-	0	0	0	1					
Miss		●	●	●			●					
Evict								0				



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

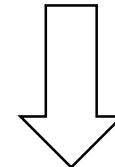


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	3				
	-	-	0	1	1	1	2	2				
	-	-	-	0	0	0	1	1				
Miss		●	●	●			●	●				
Evict									0			



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

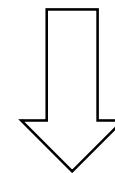


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0				
	-	-	0	1	1	1	2	3				
	-	-	-	0	0	0	1	2				
Miss		●	●	●			●	●				
Evict							0	1				



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



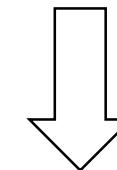
Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0			
	-	-	0	1	1	1	2	3	3			
	-	-	-	0	0	0	1	2	2			
Miss		●	●	●			●	●				
Evict							0	1				

A callout box labeled "Cache hit" points to the value "3" in the 8th column of the "Page frames" row, which is circled in blue.



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



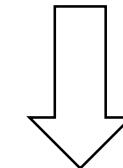
Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0	0		
	-	-	0	1	1	1	2	3	3	3		
	-	-	-	0	0	0	1	2	2	2		
Miss		●	●	●			●	●		●		
Evict							0	1				

Capacity miss

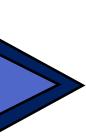


#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

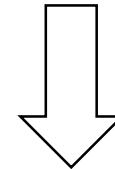


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0	1		
	-	-	0	1	1	1	2	3	3	0		
	-	-	-	0	0	0	1	2	2	3		
Miss		●	●	●			●	●		●		
Evict							0	1		2		



#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted

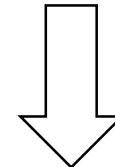


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0	1	1	
	-	-	0	1	1	1	2	3		0		
	-	-	-	0	0	0	1	2			3	
Miss		●	●	●			●	●		●		●
Evict							0	1		2		



#2: FIFO (First-In First-Out) Policy

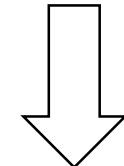
- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0	1	2	
	-	-	0	1	1	1	2	3	3	0	1	
	-	-	-	0	0	0	1	2	2	3	0	
Miss		•	•	•			•	•		•	•	
Evict							0	1	2	3		

#2: FIFO (First-In First-Out) Policy

- Idea: pages are simply placed in a queue when they enter the system and the “first-in” page is evicted



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0	1	2	2
	-	-	0	1	1	1	2	3	3	0	1	1
	-	-	-	0	0	0	1	2	2	3	0	0
Miss		●	●	●			●	●				
Evict							0	1				

Cache hit



#2: FIFO (First-In First-Out) Policy

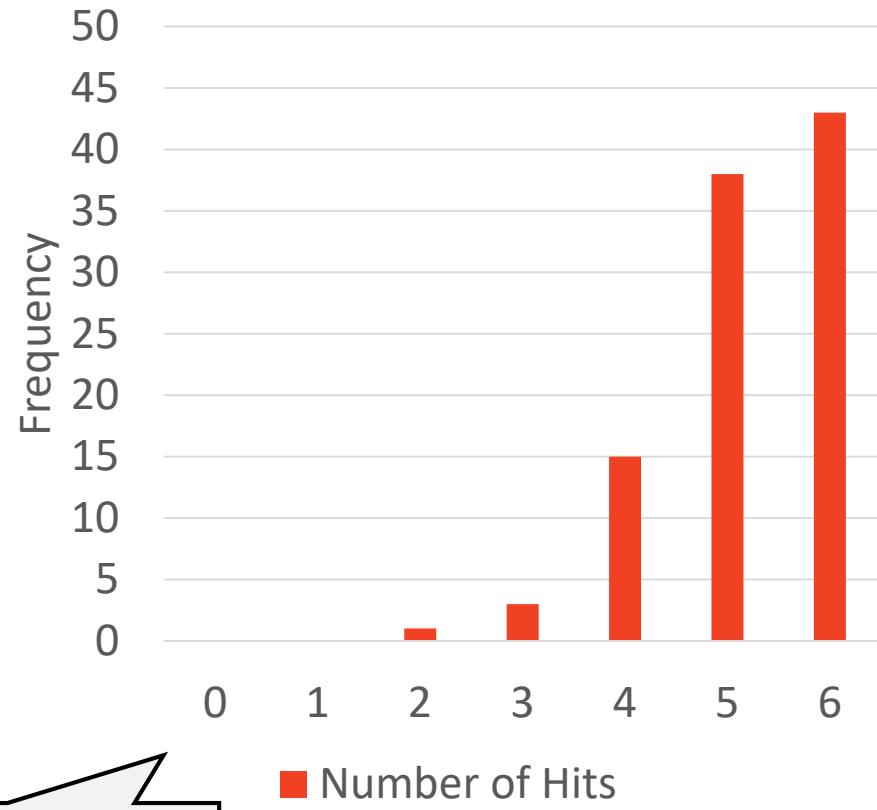
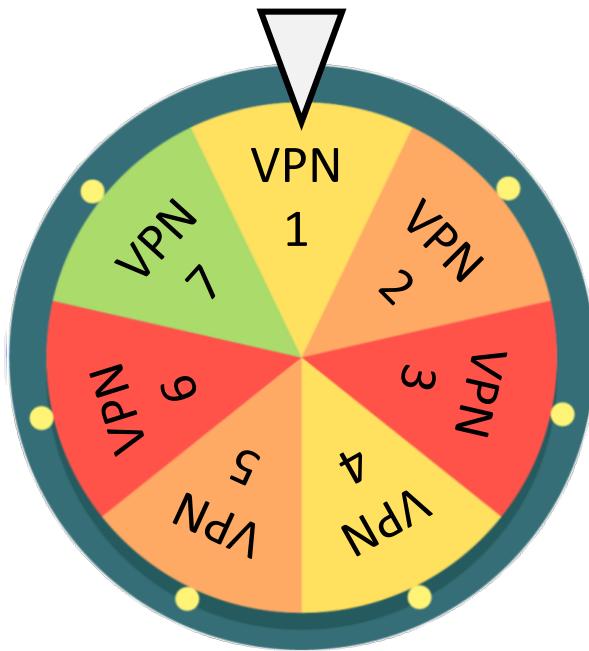
- Idea: pages are simply placed in a queue when then enter the system and the “first-in” page is evicted

FIFO does
notably worse ☹
(Only 4 hits)

Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2	2	3	0	0	1	2	2
	-	-	0	1	1	1	2	3	3	0	1	1
	-	-	-	0	0	0	1	2	2	3	0	0
Miss		•	•	•				•	•	•	•	•
Evict								0	1	2	3	

#3: Random Policy

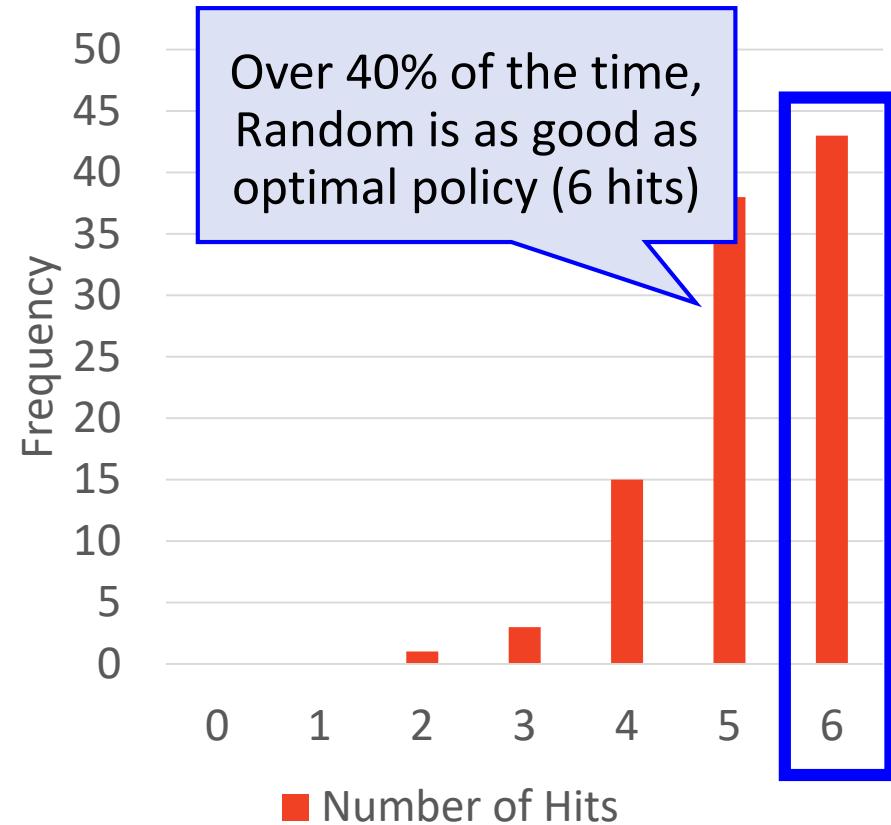
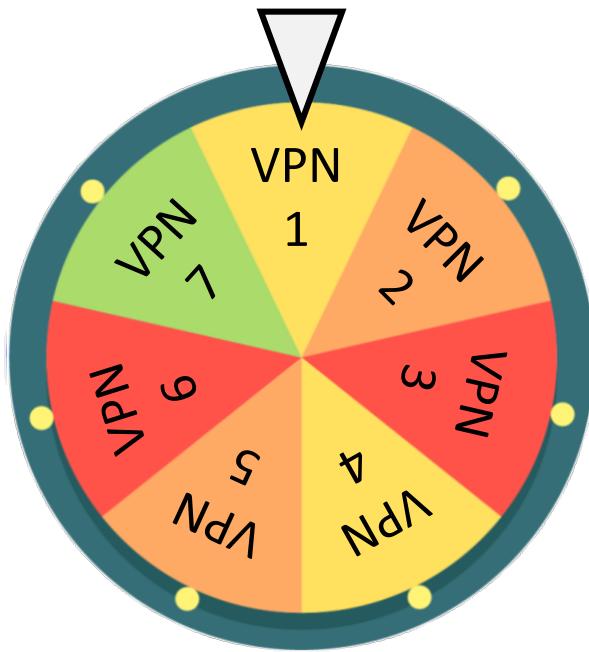
- Random policy depends on the luck of the draw



Random performance
over 10,000 trials

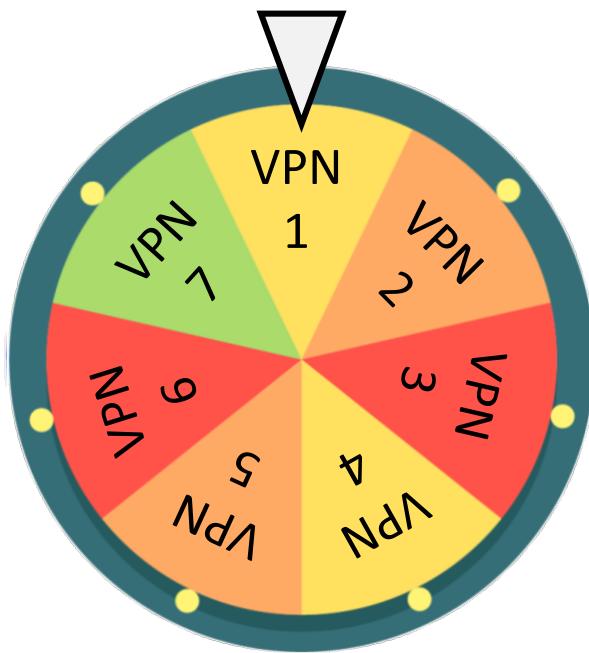
#3: Random Policy

- Random policy depends on the luck of the draw



#3: Random Policy

- Random policy depends on the luck of the draw



#4: LRU (Least Recently Used) Policy

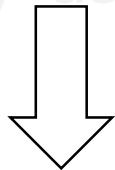
- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using **history** information

Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-											
	-											
	-											
Miss												
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information



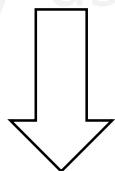
Time	1	2	3	4	5	6	7	8	9	10	11
Access	0	1	2	0	1	3	0	3	1	2	1
Page frames	-	-	-								
Miss											
Evict											

Think about this as a queue, but if the VPN is accessed then reinsert it at the end of queue



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

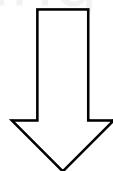


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0										
	-	-										
	-	-										
Miss		●										
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

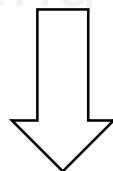


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	0									
	-	-	-									
	-	-										
Miss		●	●									
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

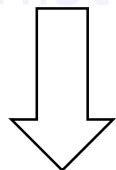


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1									
	-	-	0									
	-	-	-									
Miss		●	●									
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

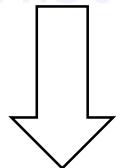


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	1								
	-	-	0	0								
	-	-	-	-	-							
Miss		●	●	●								
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

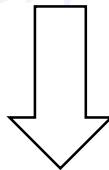


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2								
	-	-	0	1								
	-	-	-	0								
Miss		●	●	●								
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

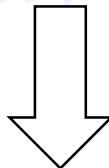


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	2							
	-	-	0	1	1							
	-	-	-	0	0							
Miss		●	●	●	Cache hit - reinserts it at the end of queue							
Evict												



#4: LRU (Least Recently Used) Policy

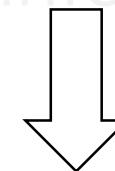
- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0							
	-	-	0	1	2							
	-	-	-	0	1							
Miss		●	●	●								
Evict												

#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information



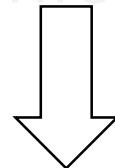
Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	0						
	-	-	0	1	2	2						
	-	-	-	0	1	1						
Miss		●	●	●								
Evict												

Cache hit -
reinserts it at the
end of queue



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

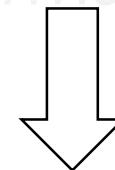


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1						
	-	-	0	1	2	0						
	-	-	-	0	1	2						
Miss		●	●	●								
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	1					
	-	-	0	1	2	0	0					
	-	-	-	0	1	2	2					
Miss		●	●	●			●					
Evict												

Capacity miss



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality

First of queue means least recently used

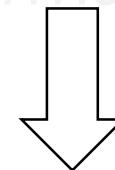


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	1					
	-	-	0	1	2	0	0					
	-	-	-	0	1	2	2					
Miss		•	•	•			•					
Evict												



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

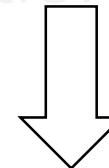


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3					
	-	-	0	1	2	0	1					
	-	-	-	0	1	2	0					
Miss		●	●	●			●					
Evict								2				



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

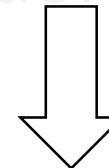


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	3				
	-	-	0	1	2	0	1	1				
	-	-	-	0	1	2	0	0				
Miss		●	●	●			●					
Evict								2				



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

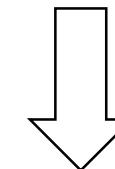


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0				
	-	-	0	1	2	0	1	3				
	-	-	-	0	1	2	0	1				
Miss		●	●	●			●					
Evict								2				



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

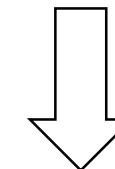


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	0			
	-	-	0	1	2	0	1	3	3			
	-	-	-	0	1	2	0	1	1			
Miss		●	●	●			●					
Evict									2			



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

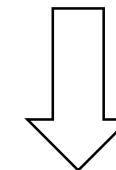


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3			
	-	-	0	1	2	0	1	3	0			
	-	-	-	0	1	2	0	1	1			
Miss		●	●	●			●					
Evict									2			



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

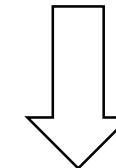


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3	3		
	-	-	0	1	2	0	1	3	0	0		
	-	-	-	0	1	2	0	1	1	1		1
Miss		●	●	●			●					
Evict									2			



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

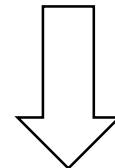


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3	1		
	-	-	0	1	2	0	1	3	0	3		
	-	-	-	0	1	2	0	1	1	0		
Miss		●	●	●			●					
Evict									2			



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

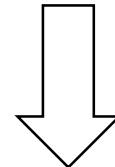


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3	1	1	
	-	-	0	1	2	0	1	3			3	
	-	-	-	0	1	2	0	1			0	
Miss		●	●	●			●			●		
Evict									2			



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information

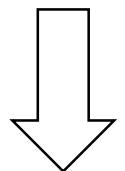


Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3	1	2	
	-	-	0	1	2	0	1	3	0	3	1	
	-	-	-	0	1	2	0	1	1	0	3	
Miss		●	●	●			●				●	
Evict							2				0	



#4: LRU (Least Recently Used) Policy

- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principles of locality by using history information



Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3	1	2	2
	-	-	0	1	2	0	1	3	0	3	1	1
	-	-	-	0	1	2	0	1	1	0	3	3
Miss		●	●	●			●			●		
Evict							2				0	



#4: LRU (Least Recently Used) Policy

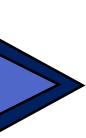
- FIFO and Random might kick out an important page, one that is about to be referenced again
- Idea: LRU policy is based on the principle of locality by using history information

LRU achieves 6 hits ! It matches optimal in its performance 😊

Time		1	2	3	4	5	6	7	8	9	10	11
Access		0	1	2	0	1	3	0	3	1	2	1
Page frames	-	0	1	2	0	1	3	0	3	1	2	1
	-	-	0	1	2	0	1	3	0	3	1	2
	-	-	-	0	1✓	2✓	0	1✓	1✓	0	3	3✓
Miss		•	•	•			•			•	•	•
Evict							2				0	

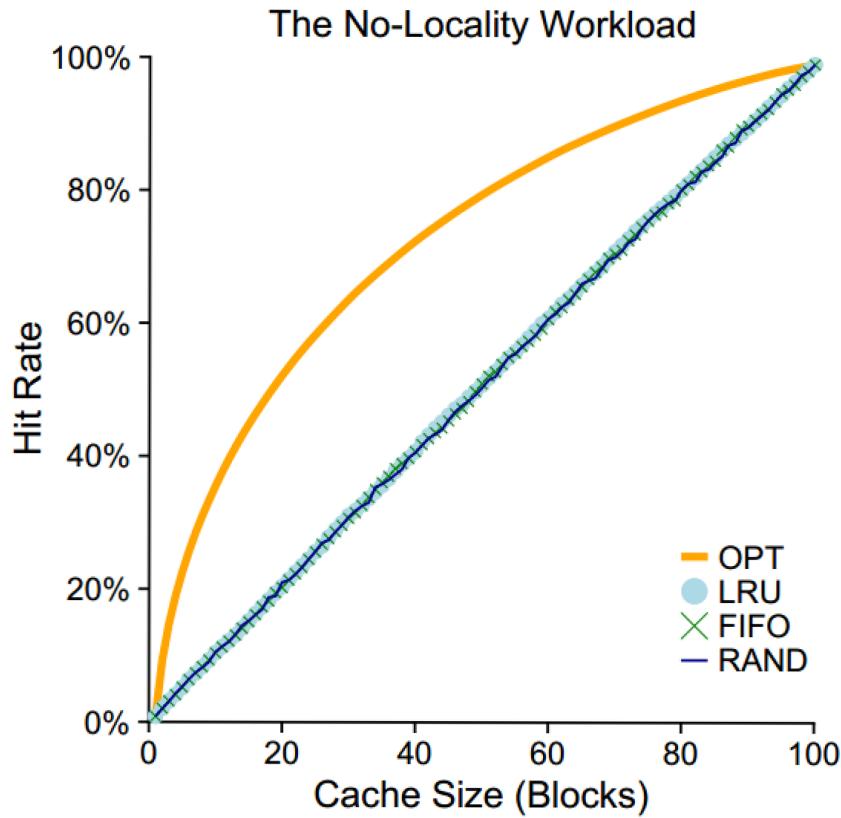
Optimal vs. Practical Algorithms

- See how each policy (Opt, FIFO, Random, and LRU) behaves over the range of cache sizes
 - Varies the cache size from very small (1 page) to enough to hold all the unique pages (100 pages)



#1: 100% Random Workload

- Accesses 100 unique pages over time, choosing the next page to refer to at random

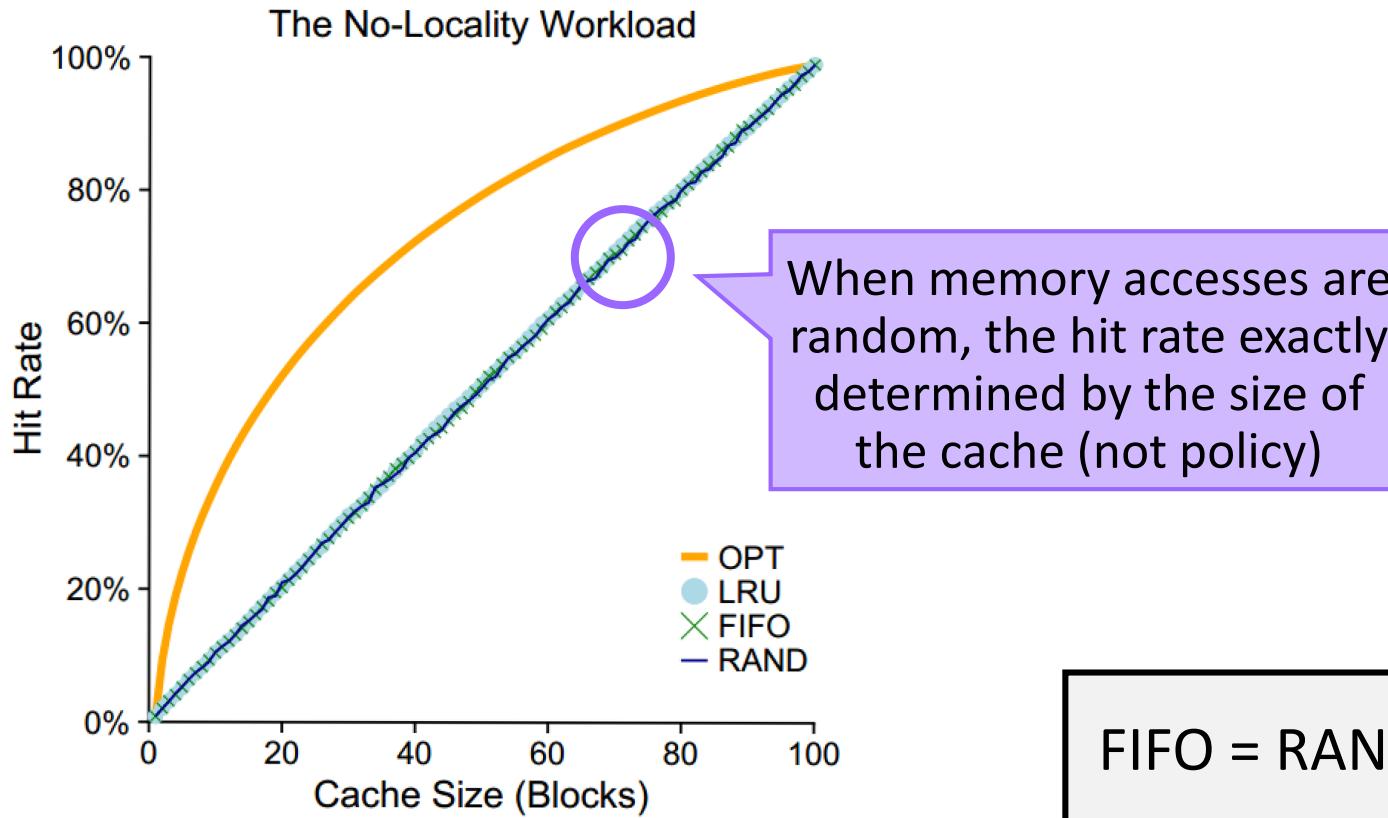


FIFO = RAND = LRU < OPT



#1: 100% Random Workload

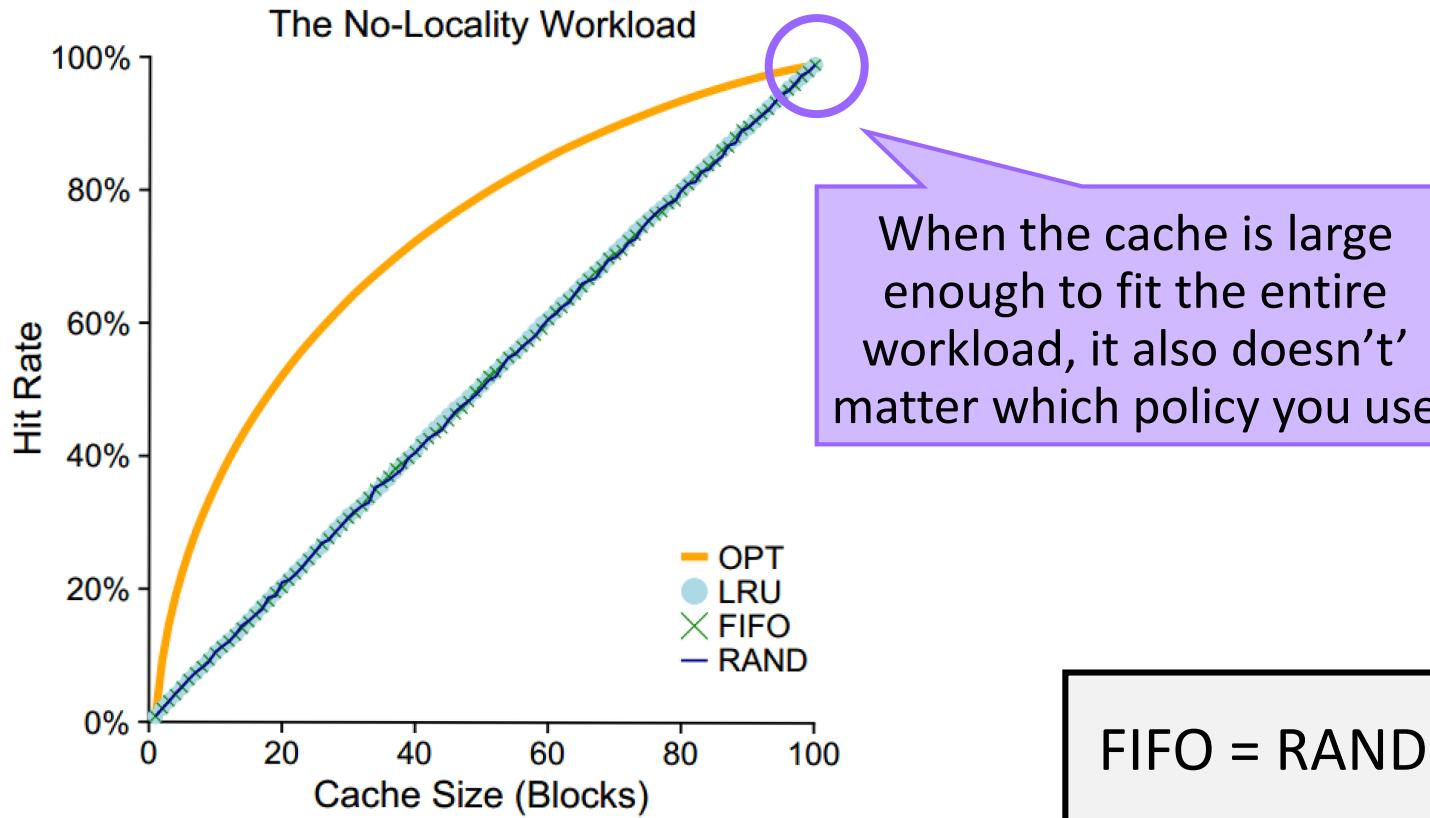
- Accesses 100 unique pages over time, choosing the next page to refer to at random





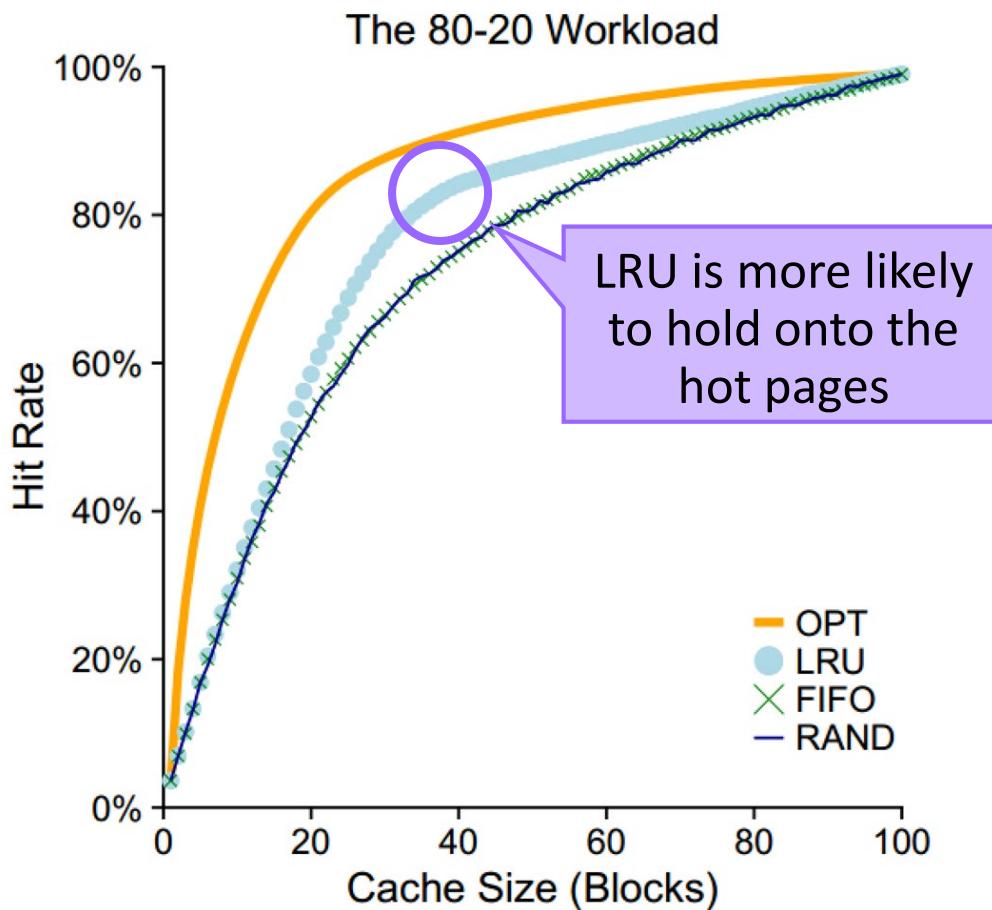
#1: 100% Random Workload

- Accesses 100 unique pages over time, choosing the next page to refer to at random



#2: 80/20 Workload

- 80% of memory accesses are for 20% of pages

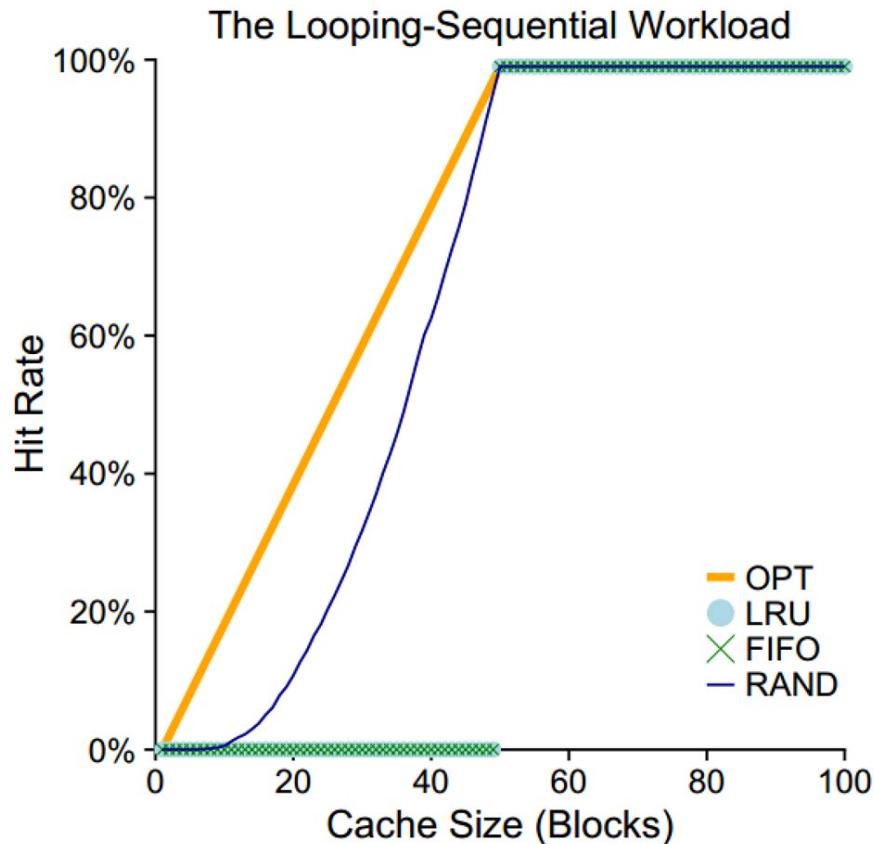


FIFO = RAND < LRU < OPT



#3: Looping-Sequential Workload

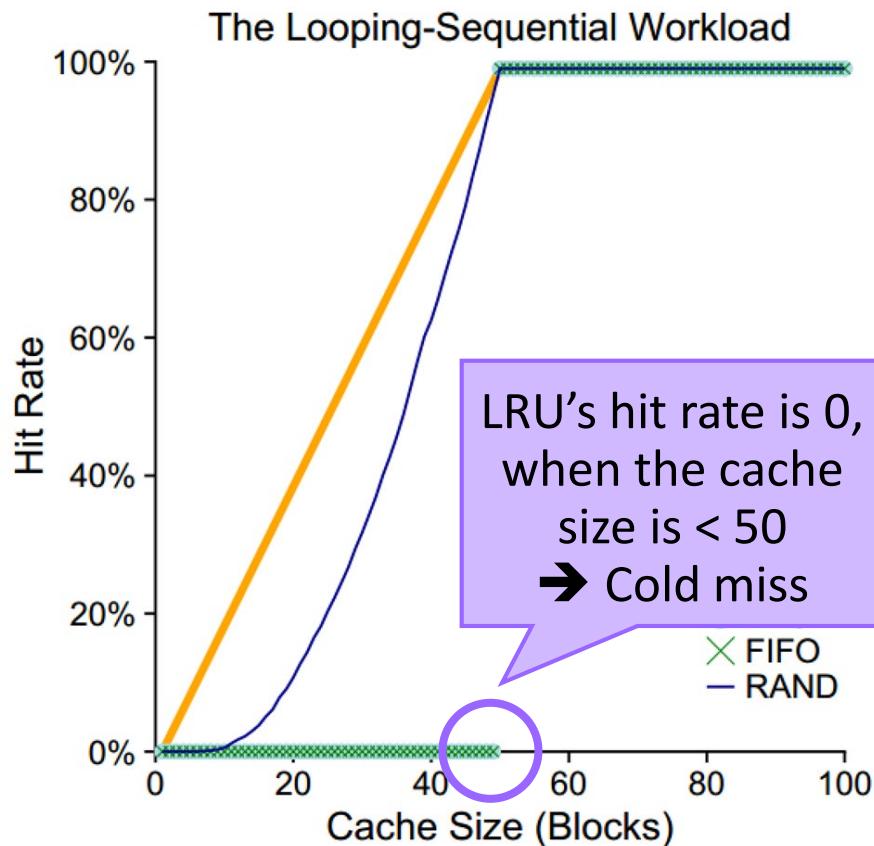
- Refers to 50 pages in sequence 0,1,...,49 and repeats this loop, for total 10,000 accesses



FIFO = LRU < **RAND** < OPT

#3: Looping-Sequential Workload

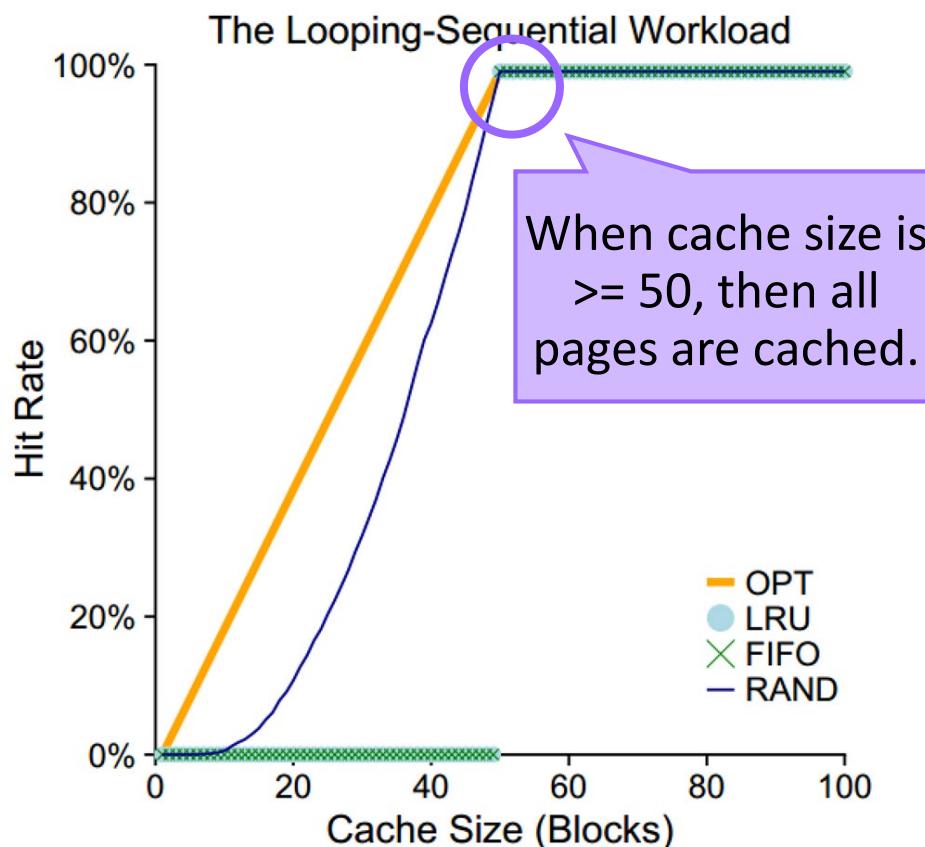
- Refers to 50 pages in sequence 0,1,...,49 and repeats this loop, for total 10,000 accesses



FIFO = LRU < **RAND** < OPT

#3: Looping-Sequential Workload

- Refers to 50 pages in sequence 0,1,...,49 and repeats this loop, for total 10,000 accesses

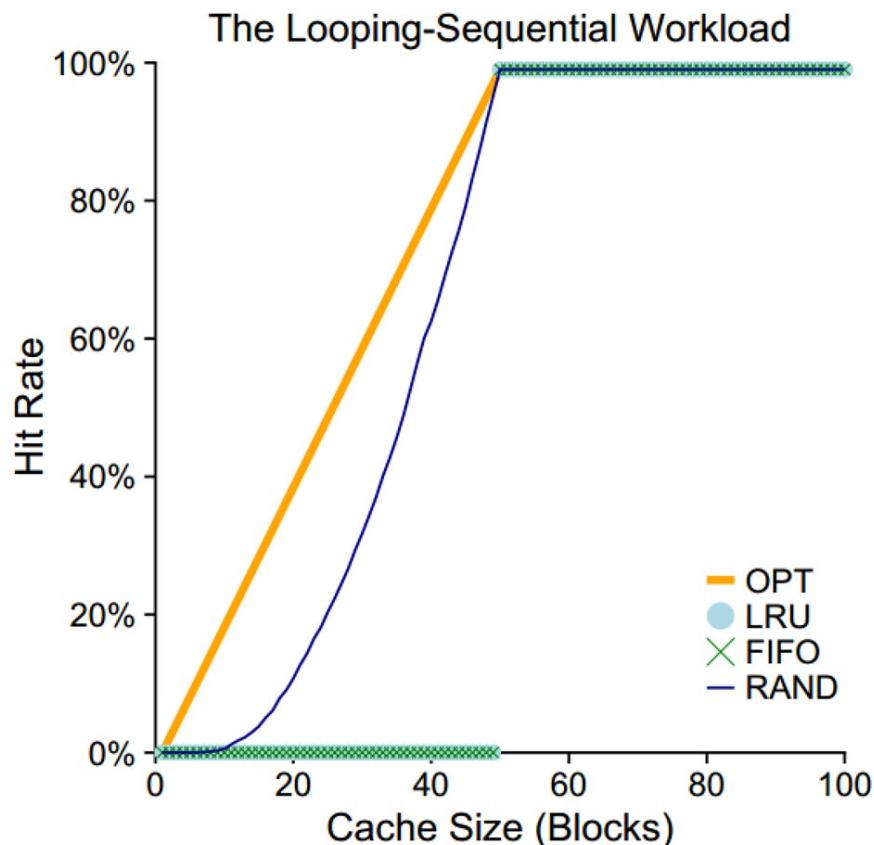


When cache size is
 ≥ 50 , then all
pages are cached.

FIFO = LRU < **RAND** < OPT

#3: Looping-Sequential Workload

- Refers to 50 pages in sequence 0,1,...,49 and repeats this loop, for total 10,000 accesses

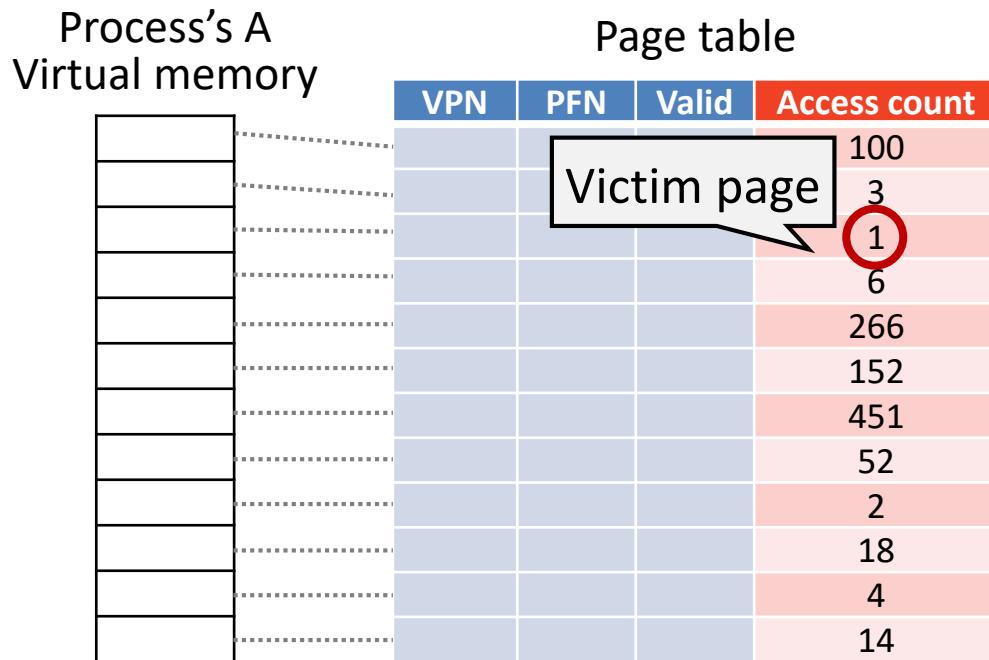


RAND has a nice property than LRU (not having weird corner-case behaviors)

FIFO = LRU < RAND < OPT

Implementing Historical Algorithms

- LRU policy is history-based
- Implementation #1: Perfect LRU



To keep track of which pages have been least- and most-recently used, the system has to do some accounting work on every memory reference

Implementing Historical Algorithms

- LRU policy is history-based
- Implementation #1: Perfect LRU

Process's A
Virtual memory

Page table

VPN	PFN	Valid	Access count
			100
			3
			1
			6
			266
			152
			451
			52
			2
			18
			4
			14

Such accounting could greatly reduce performance

- Compare all access counts
- also require memory space

Approximating LRU

- Idea: requires hardware support in the form of a use bit (reference bit)

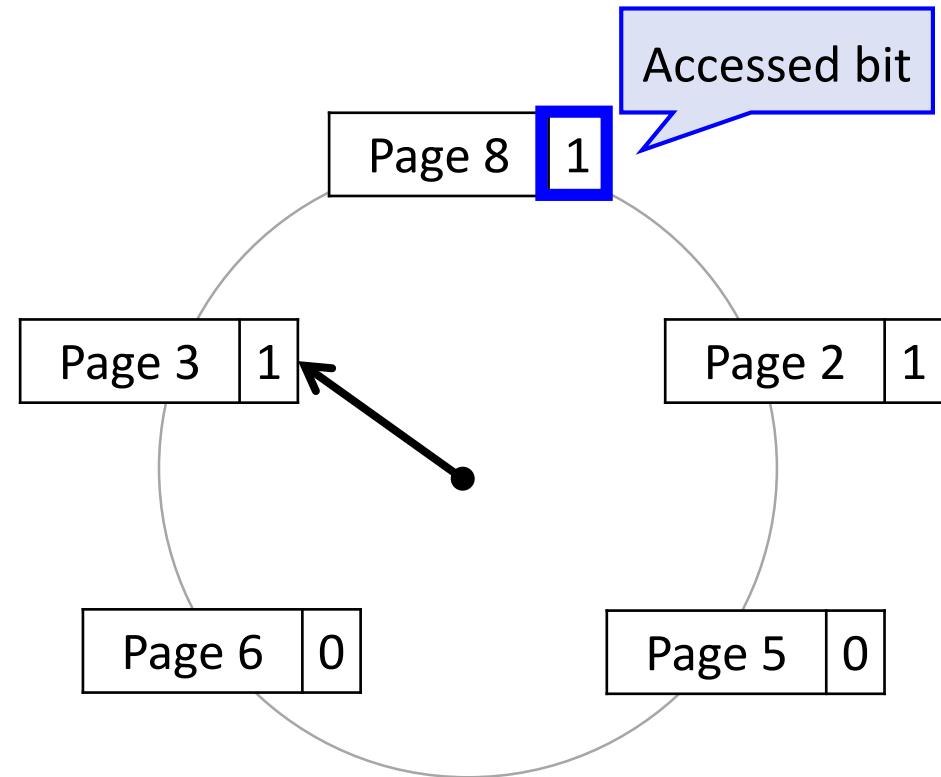
[Recall] Page Table Entry of x86

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

X86 adds accessed bit in the page table entry - whenever a page is referenced, the accessed bit is set by hardware to 1

Clock Algorithm

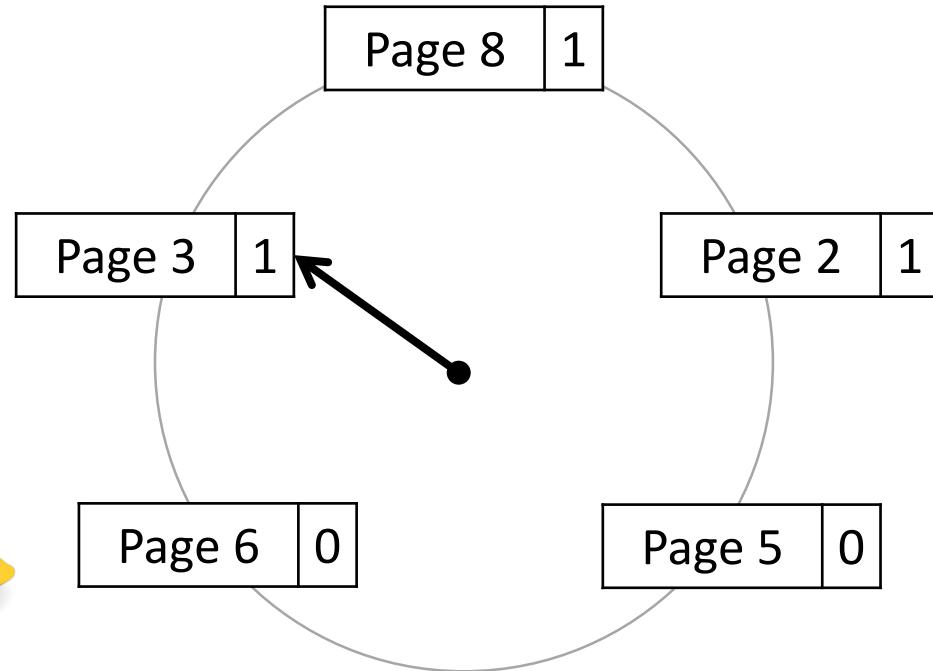
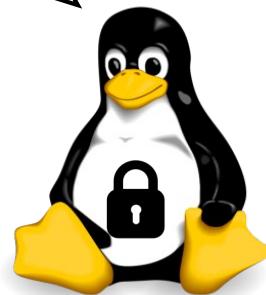
- Keep pages in circular FIFO list



Clock Algorithm

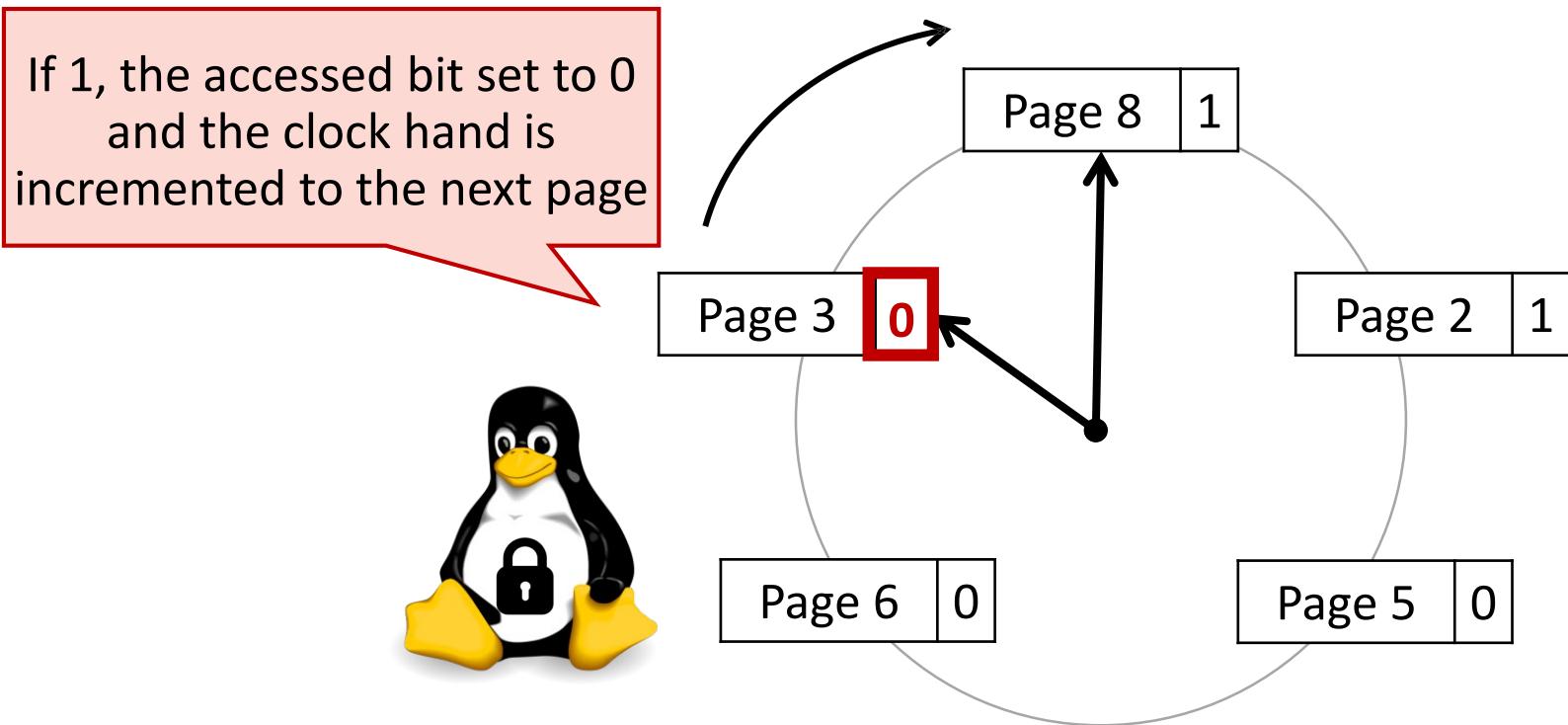
- Keep pages in circular FIFO list

When a replacement must occur, the OS checks if the currently-pointed to page 3 has a reference bit of 1 or 0



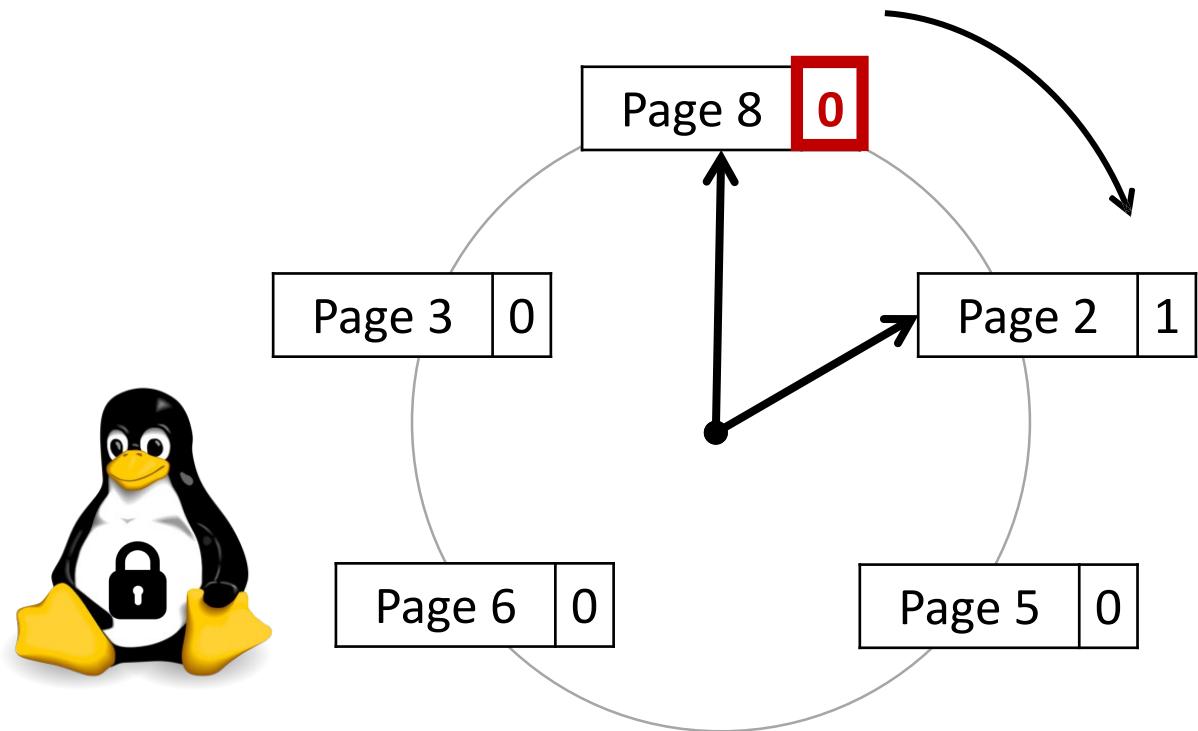
Clock Algorithm

- Keep pages in circular FIFO list



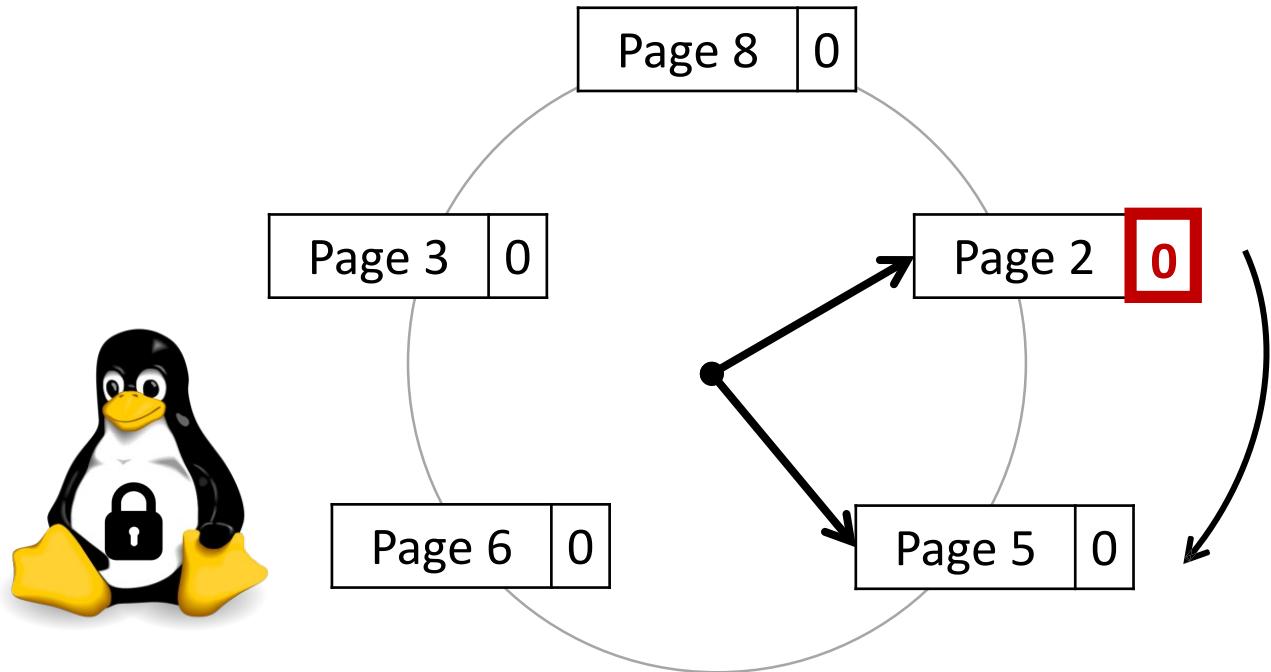
Clock Algorithm

- Keep pages in circular FIFO list



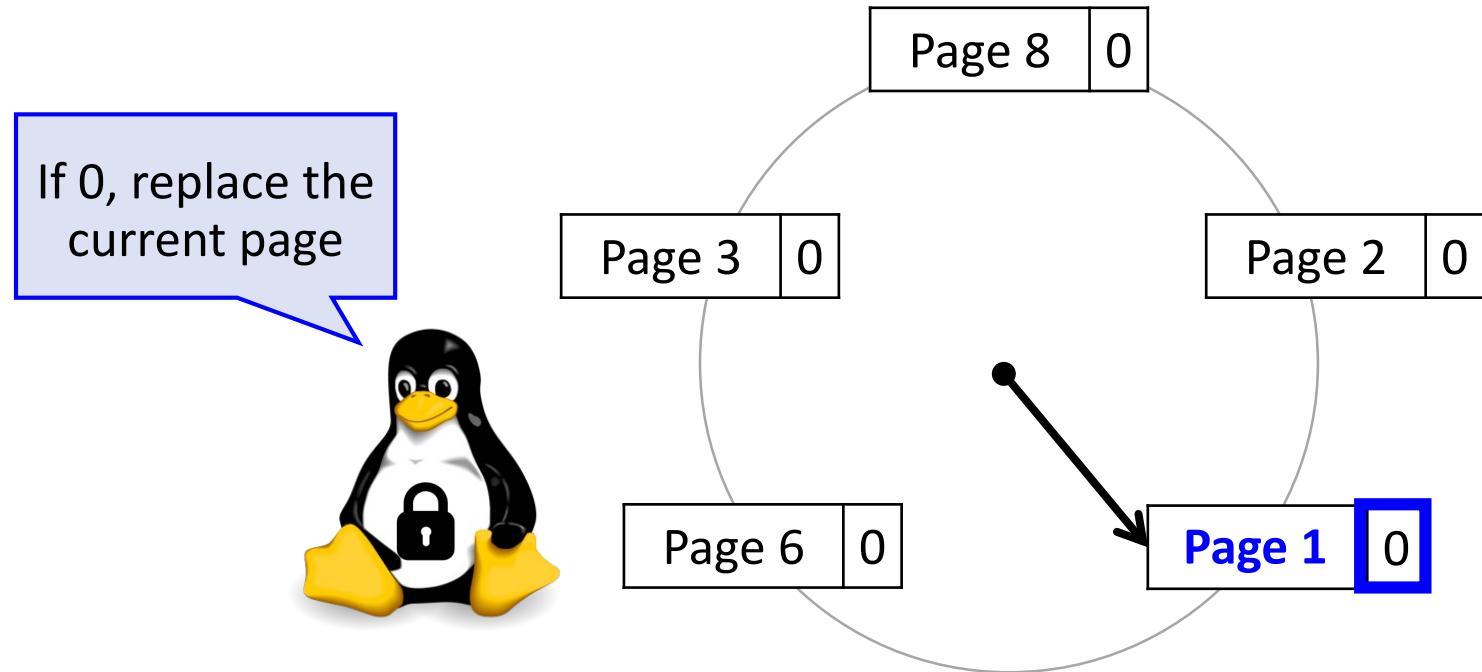
Clock Algorithm

- Keep pages in circular FIFO list



Clock Algorithm

- Keep pages in circular FIFO list





Revised Clock Algorithm

- Considering dirty pages

[Recall] Page Table Entry of x86

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

X86 adds dirty bit in the page table entry - whenever a page is updated, the dirty bit is set by hardware to 1



Revised Clock Algorithm

- Considering dirty pages

[Recall] Page Table Entry of x86

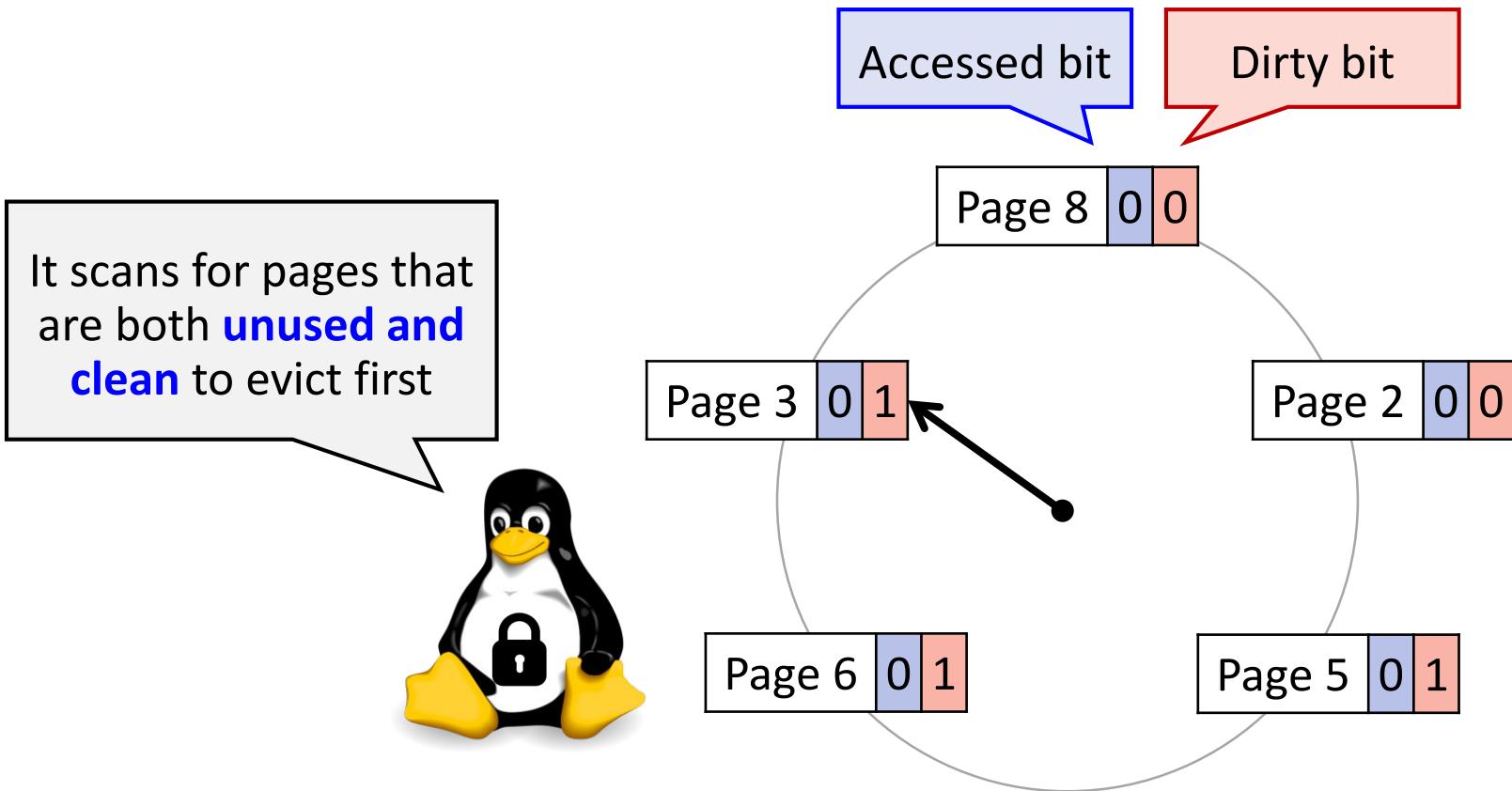
31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

If a page has been modified (**dirty**), it must be written back to storage to evict it → Expensive ! Let's skip this page



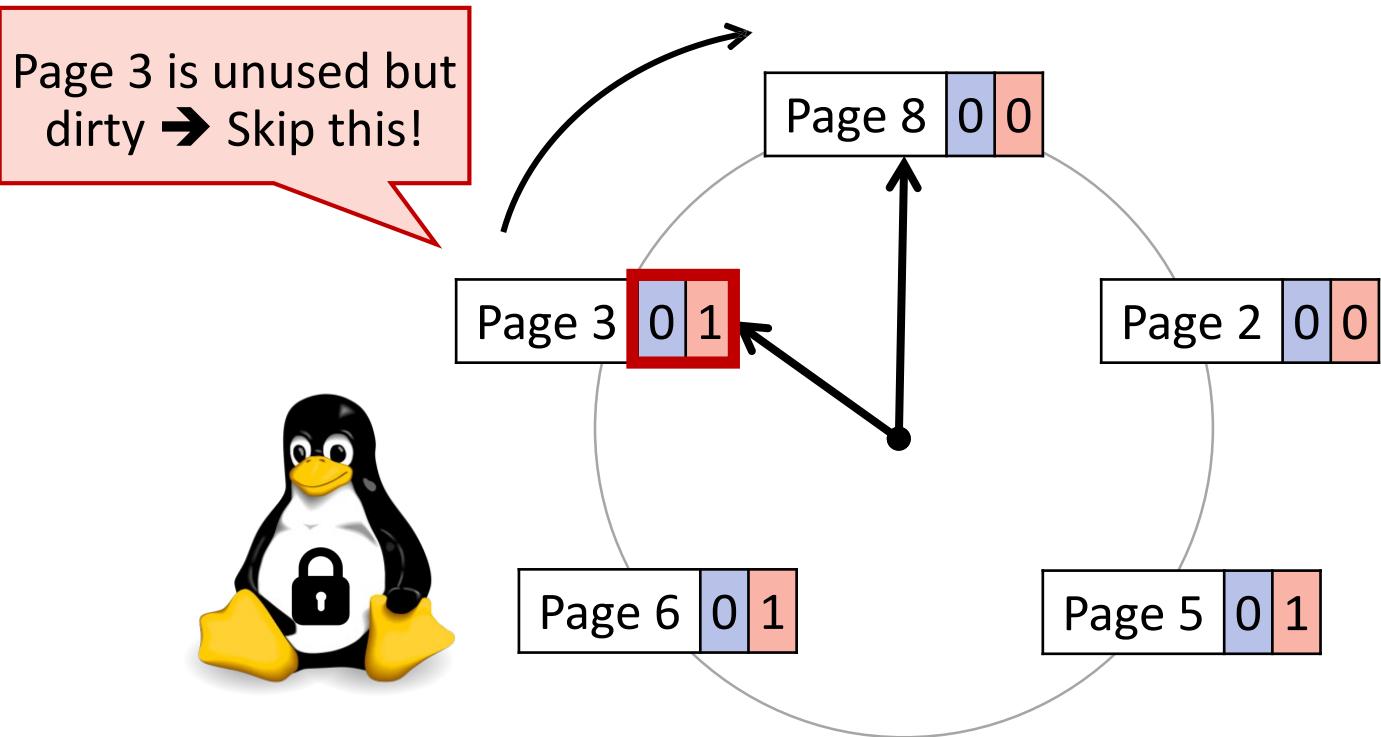


Revised Clock Algorithm



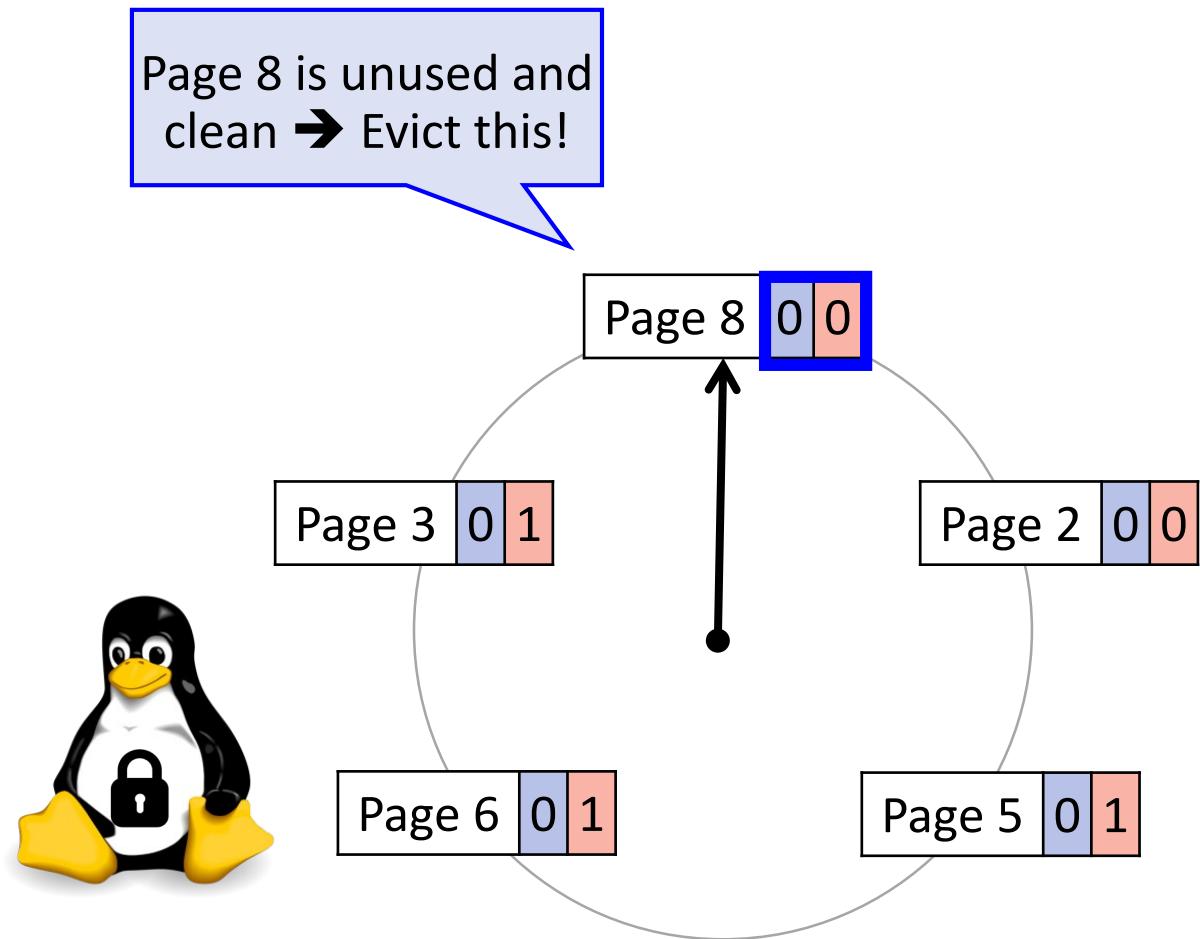


Revised Clock Algorithm





Revised Clock Algorithm



What's Next

- Let's check how the OS allocates memory (page frame) to processes

21S EE415

Page Allocation

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



Page Allocation

Myoungsoo Jung
Computer Division

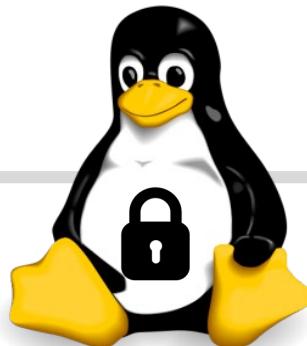
Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab* 

Linux Memory Management

Linux memory is managed by two distinct components: i) virtual memory management and ii) physical memory management



Virtual memory
management
(Process view point)

Physical memory
management
(CPU view point)

Virtual Memory Management

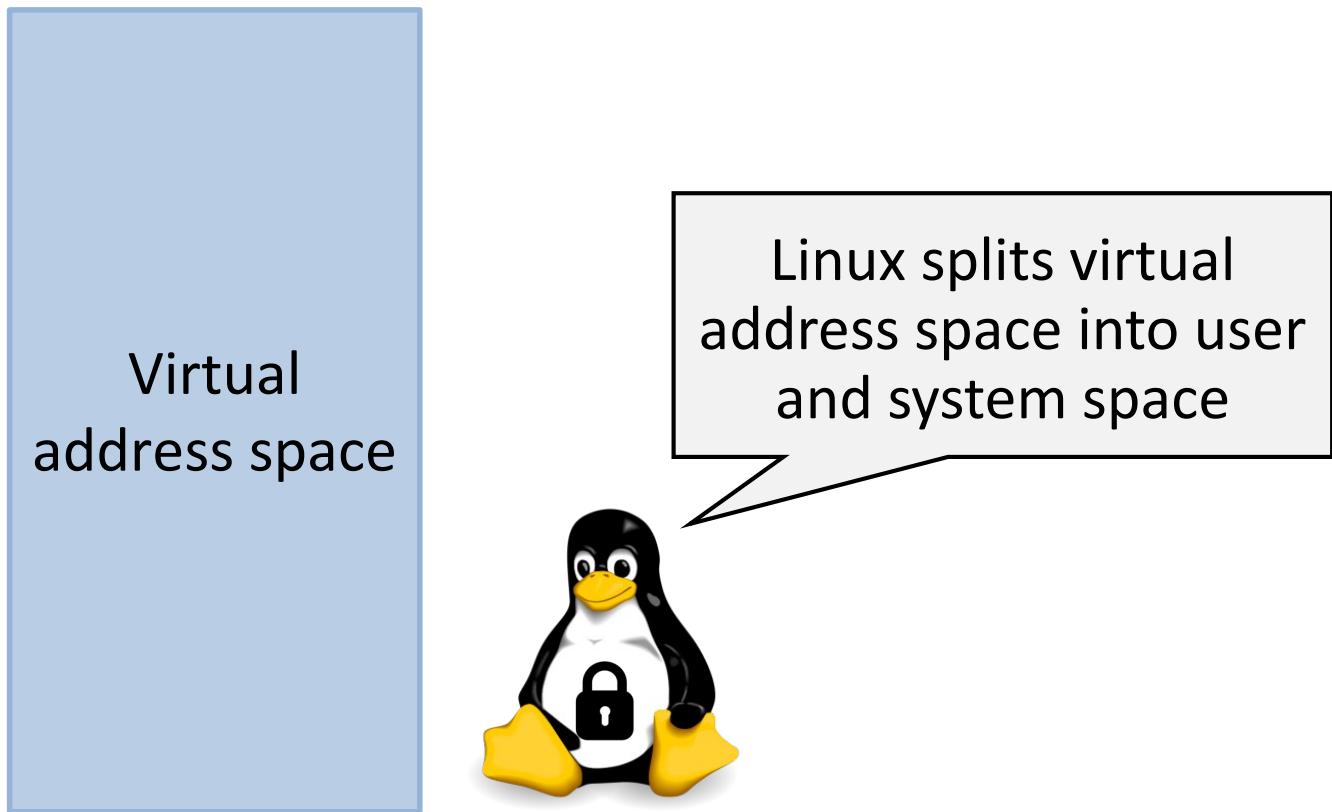
Virtual memory
management
(Process view point)

Physical memory
management
(CPU view point)



Virtual Address Spaces

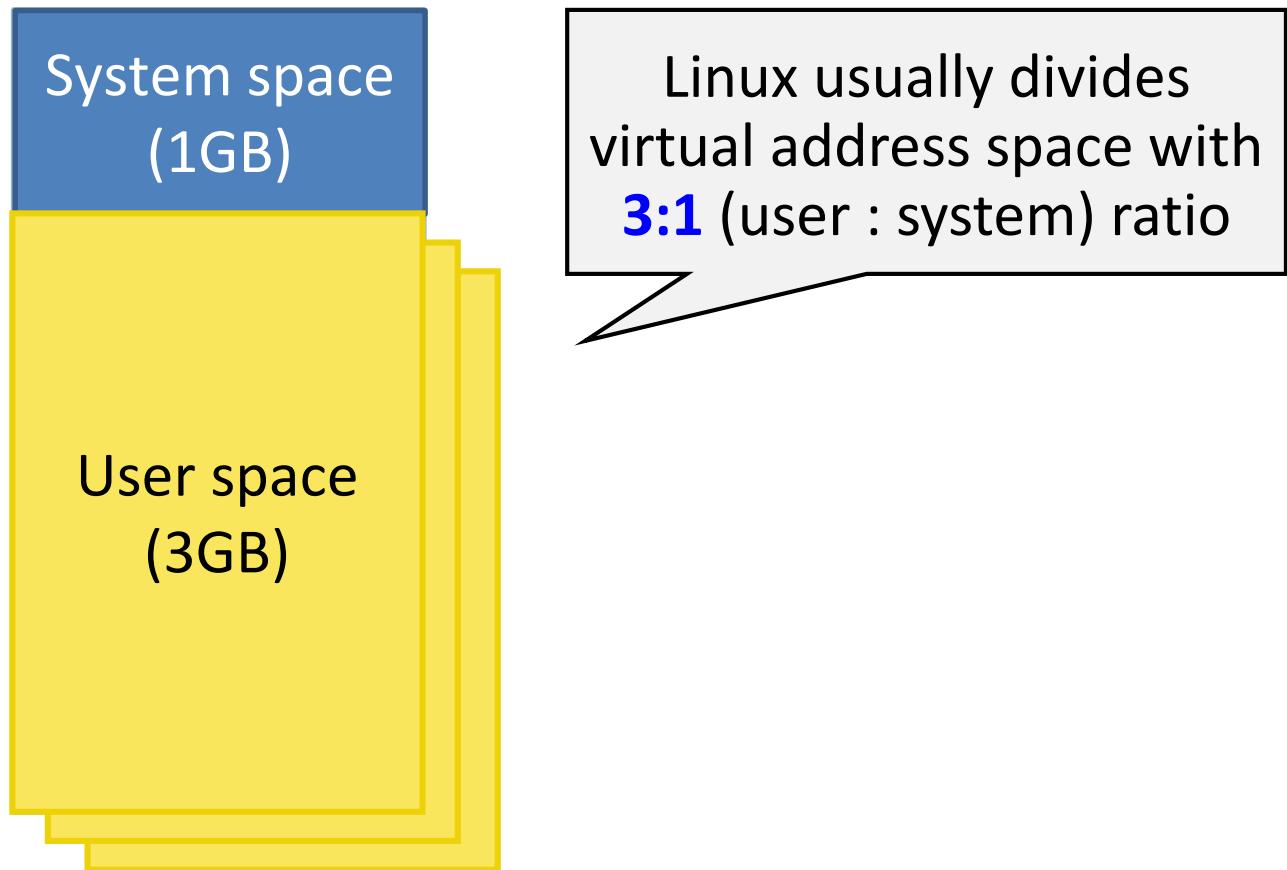
- Each process in a multi-tasking OS runs in its virtual address space





User and System Space (32bit)

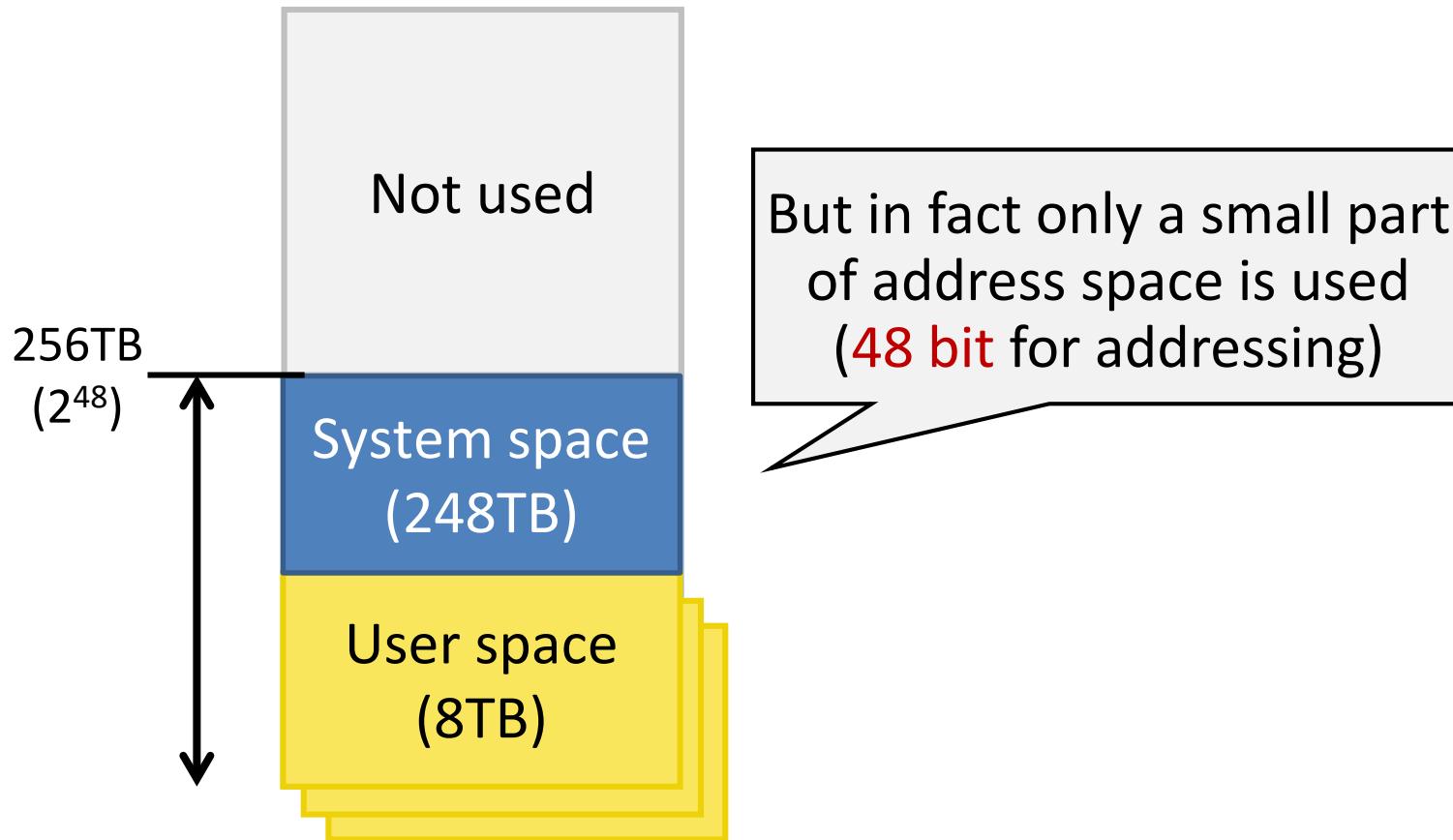
- In 32bit system, a virtual address space available for a total of 2^{32} bytes (4GB)



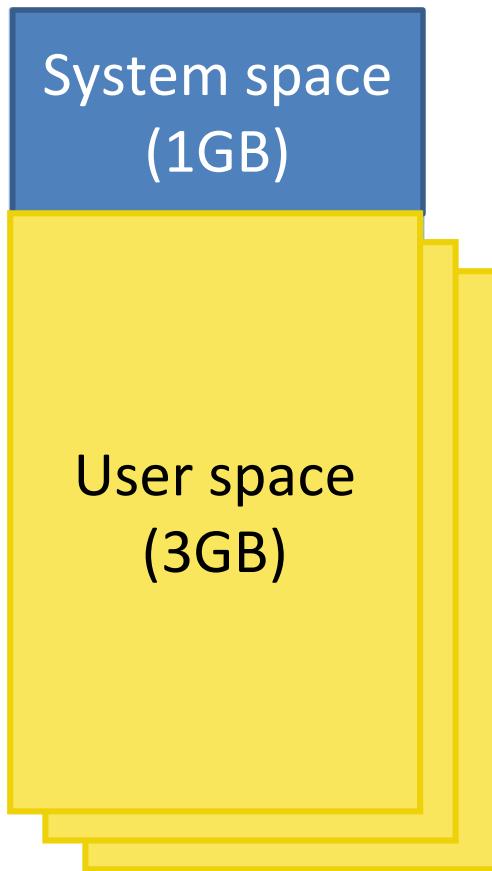


User and System Space (64bit)

- In 64bit system, the **theoretical** size of the virtual address space is 2^{64} bytes (16EB)

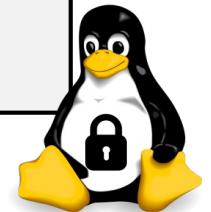


Process Address Space



In this lecture, we will assume 32bit system

How does the kernel manage process memory?



Example of Address Mapping

```
C myCode.c      x
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *value;

    value = (int*) malloc(sizeof(int));
    *value = 1;
    printf("%d\n", *value);
    free(value);

    return 0;
}
```

Virtual memory



When user executes
a program, OS loads
some part of ELF
(a.out) to memory

Example of Address Mapping

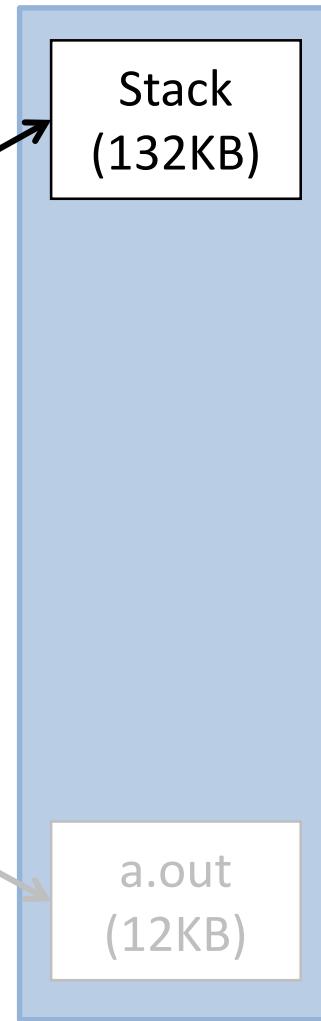
```
C myCode.c      x
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *value;

    value = (int*) malloc(sizeof(int));
    *value = 1;
    printf("%d\n", *value);
    free(value);

    return 0;
}
```

Virtual memory



main() requires
memory space to
execute the code
(e.g., stack frame)

Example of Address Mapping

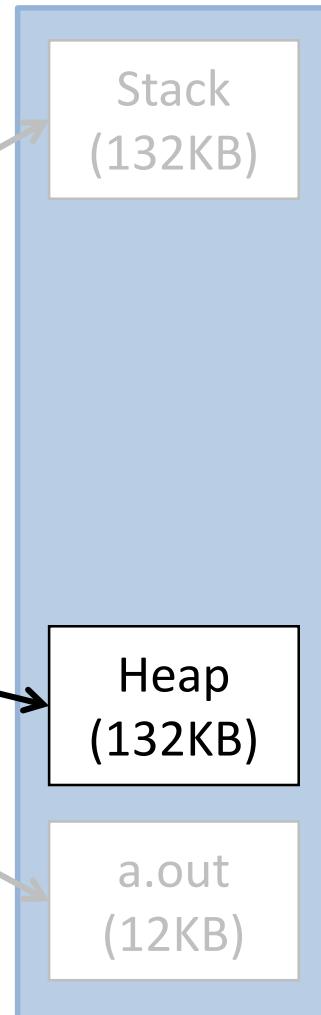
```
C myCode.c      x
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *value;

    value = (int*) malloc(sizeof(int));
    *value = 1;
    printf("%d\n", *value);
    free(value);

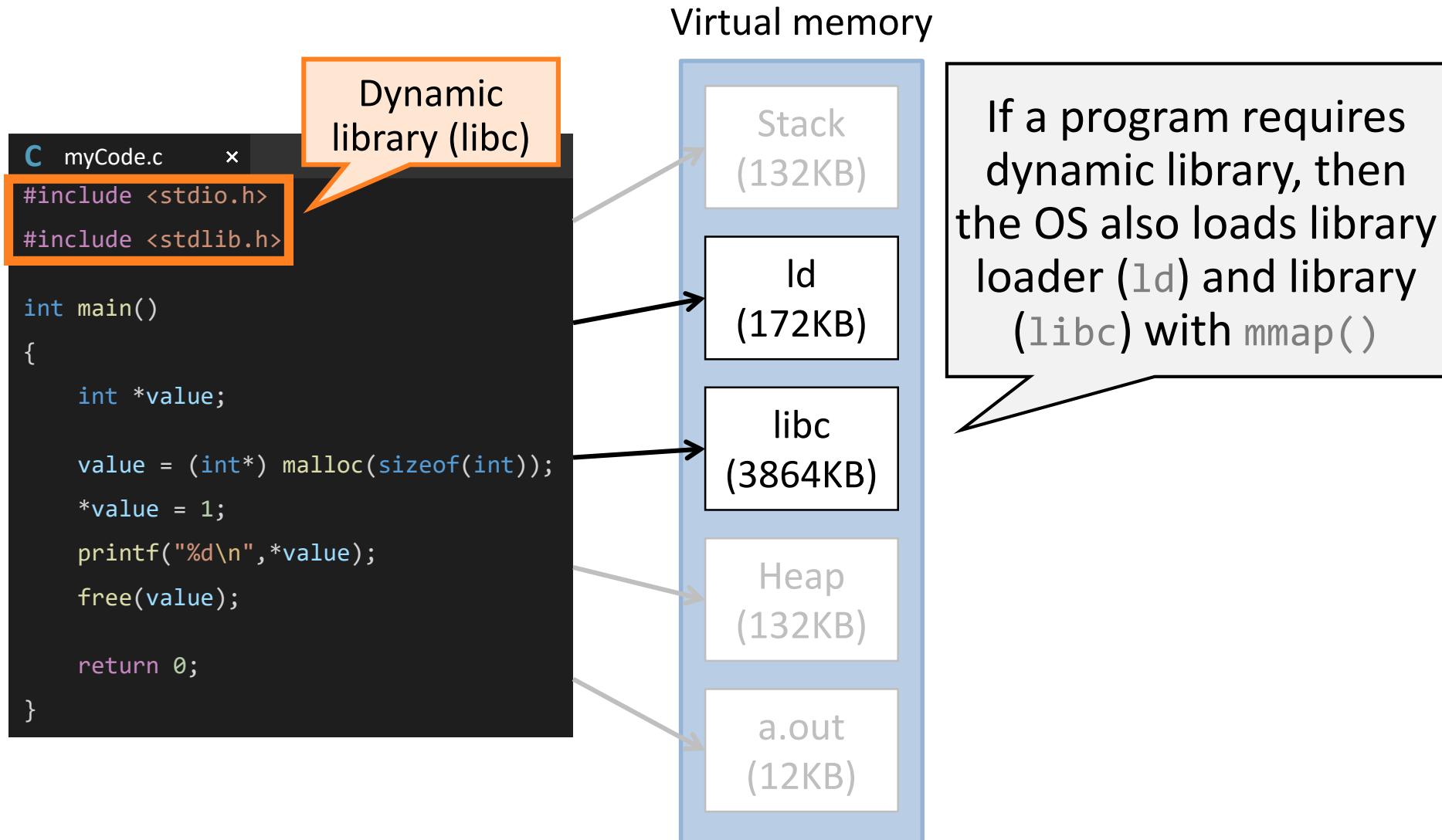
    return 0;
}
```

Virtual memory



Dynamic memory allocation (`malloc`) requires “Heap” space

Example of Address Mapping



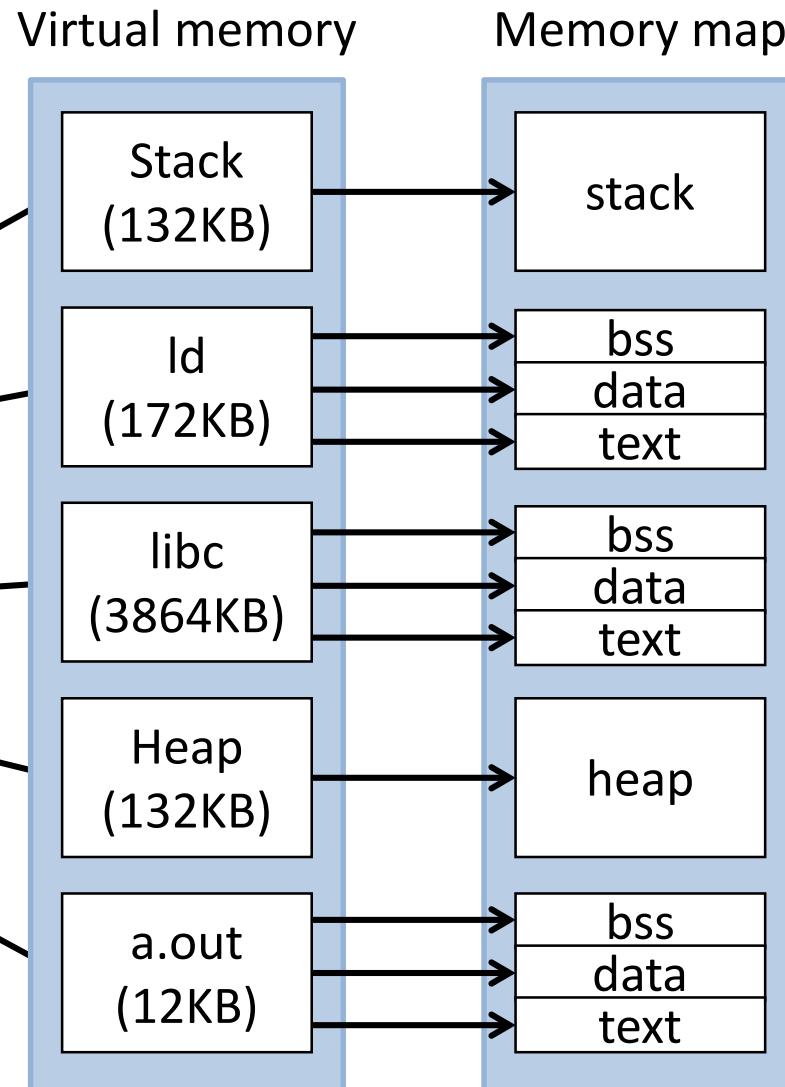
Example of Address Mapping

```
C myCode.c      x
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *value;

    value = (int*) malloc(sizeof(int));
    *value = 1;
    printf("%d\n", *value);
    free(value);

    return 0;
}
```



Example of Address Mapping

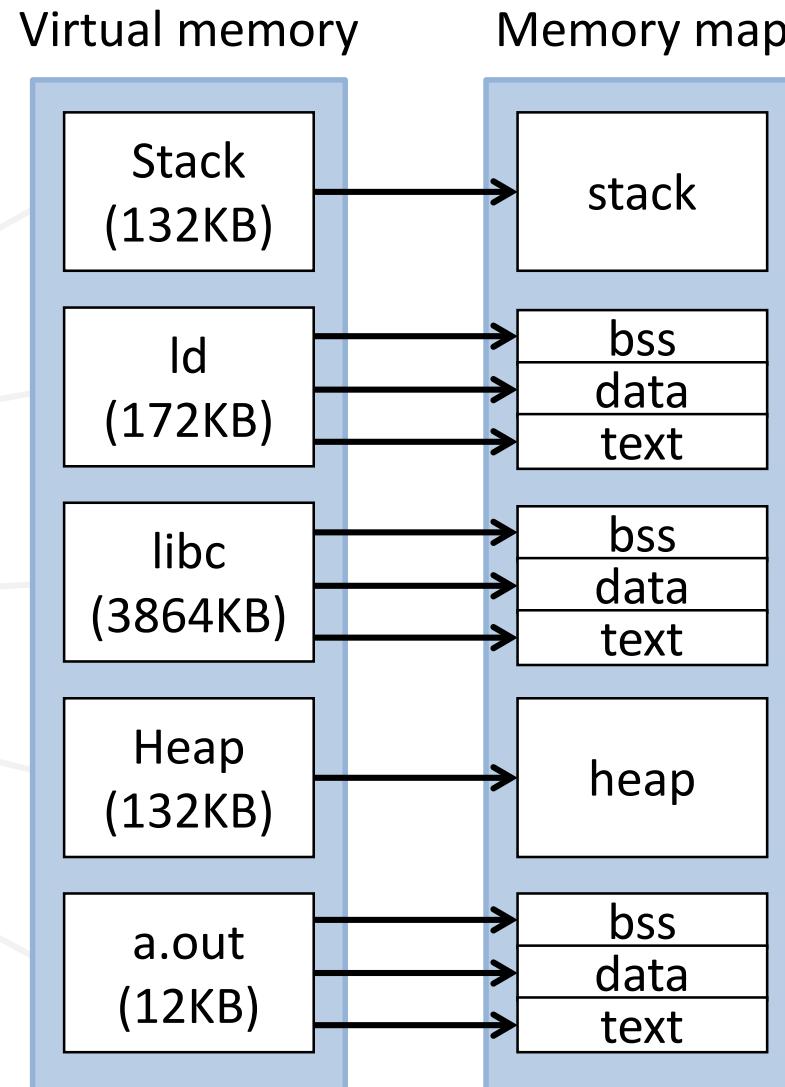
```
C myCode.c
#include <cs50.h>
#include <stdio.h>

int main()
{
    int *value;

    value = (int*) malloc(sizeof(int));
    *value = 1;
    printf("%d\n", *value);
    free(value);

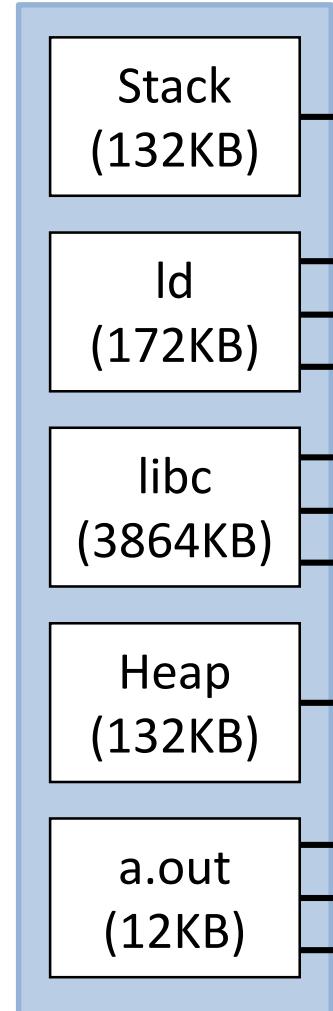
    return 0;
}
```

Linux organizes virtual memory as a collection of “areas”



Example of Address Mapping

Virtual memory

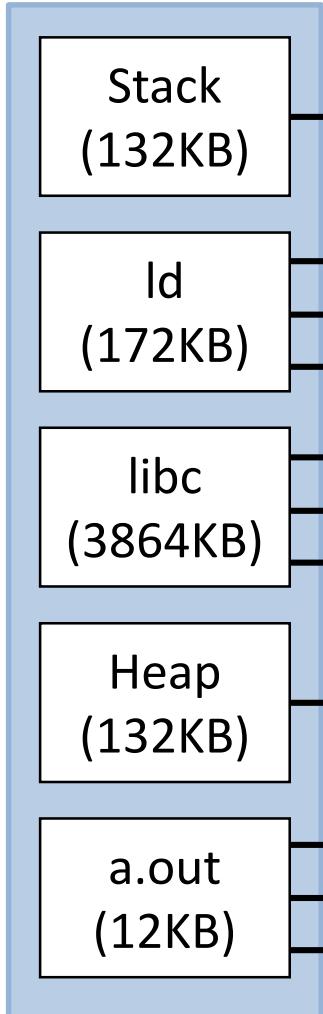


Memory map

“Virtual Memory Areas (VMA)”	
Permissions/addresses	
rw-p	bffd000-c0000000
rw-p	401f3000-401f4000
r--p	401f2000-401f3000
r-xp	401cc000-401f2000
rw-p	401c4000-401c6000
r--p	401c0000-401c4000
r-xp	40000000-401c0000
rw-p	012d7000-012f8000
rw-p	00601000-00602000
r--p	00600000-00601000
r-xp	00400000-00401000

Example of Address Mapping

Virtual memory



Memory map

rw-p	text: r-xp
rw-p	data: rw-p
r--p	bss: rw-p
r-xp	heap: rw-p
	stack: rw-p
rw-p	401c4000-401c6000
r--p	401c4000-401c6000
r-xp	40000000-40000000
	401c6000-401c8000
rw-p	012c0000-012c2000
	012c2000-012c4000
rw-p	00601000-00602000
r--p	00600000-00601000
r-xp	00400000-00401000

Legend:

- r: readable
- w: writable
- x: executable
- p: private (copy on write)

Example of Address Mapping

```
$ ./a.out >null &  
[1] 673
```

Let's suppose the PID of
a.out instance as 673

Example of Address Mapping

```
$ ./a.out >null &  
[1] 673  
  
$ cat /proc/673/maps
```

00400000-00301000	r-xp	00000000 fc:00 85362479	/home/ee415/a.out
00600000-00601000	r--p	00000000 fc:00 85362479	/home/ee415/a.out
00601000-00602000	rw-p	00001000 fc:00 85362479	/home/ee415/a.out
012d7000-012f8000	rw-p	00000000 00:00 0	[heap]
40000000-401c0000	r-xp	00000000 fc:00 29622698	/lib/ld-2.1.3.so
401c0000-401c4000	r--p	001c0000 fc:00 29622698	/lib/ld-2.1.3.so
401c4000-401c6000	rw-p	001c0000 fc:00 29622698	/lib/ld-2.1.3.so
401cc000-401f2000	r-xp	00000000 fc:00 29622674	/lib/libc-2.1.3.so
401f2000-401f3000	r--p	00025000 fc:00 29622674	/lib/libc-2.1.3.so
401f3000-401f4000	r--p	00026000 fc:00 29622674	/lib/libc-2.1.3.so
bffdf000-c0000000	rw-p	00000000 00:00 0	[stack]

The address map of the running process is /proc/<pid>/maps

Example of Address Mapping

```
$ ./a.out >null &
[1] 673
$ cat /proc/673/maps
00400000-00301000
00600000-00601000
00601000-00602000
012d7000-012f8000
40000000-401c0000
401c0000-401c4000
401c4000-401c6000
401cc000-401f2000
401f2000-401f3000
401f3000-401f4000
bffdf000-c0000000
```

Address: starting and ending address of the VMA region

rwx	00001000 fc:00 85362479	/home/ee415/a.out
rwx	00000000 00:00 0	[heap]
r-xp	00000000 fc:00 29622698	/lib/ld-2.1.3.so
r--p	001c0000 fc:00 29622698	/lib/ld-2.1.3.so
rw-p	001c0000 fc:00 29622698	/lib/ld-2.1.3.so
r-xp	00000000 fc:00 29622674	/lib/libc-2.1.3.so
r--p	00025000 fc:00 29622674	/lib/libc-2.1.3.so
r--p	00026000 fc:00 29622674	/lib/libc-2.1.3.so
rw-p	00000000 00:00 0	[stack]

Example of Address Mapping

\$./a.out >null &		
[1] 673		
\$ cat /proc/673/maps		
00400000-00301000	r-xp	/home/ee415/a.out
00600000-00601000	r--p	/home/ee415/a.out
00601000-00602000	rw-p	/home/ee415/a.out
012d7000-012f8000	rw-p	00000000 00:00 0 [heap]
40000000-401c0000	r-xp	00000000 fc:00 29622698 /lib/ld-2.1.3.so
401c0000-401c4000	r--p	001c0000 fc:00 29622698 /lib/ld-2.1.3.so
401c4000-401c6000	rw-p	001c0000 fc:00 29622698 /lib/ld-2.1.3.so
401cc000-401f2000	r-xp	00000000 fc:00 29622674 /lib/libc-2.1.3.so
401f2000-401f3000	r--p	00025000 fc:00 29622674 /lib/libc-2.1.3.so
401f3000-401f4000	r--p	00026000 fc:00 29622674 /lib/libc-2.1.3.so
bffdf000-c0000000	rw-p	00000000 00:00 0 [stack]

Permissions: how pages in the region can be accessed

Example of Address Mapping

```
$ ./a.out >null &
[1] 673
$ cat /proc/673/maps
00400000-00301000 r-xp 00000000 fc:00 29622698 /lib/ld-2.1.3.so
00600000-00601000 r--p 00000000 fc:00 29622698 /lib/ld-2.1.3.so
00601000-00602000 rw-p 00001000 fc:00 29622698 /home/ee415/a.out
012d7000-012f8000 rw-p 00000000 00:00 0 [heap]
40000000-401c0000 r-xp 00000000 fc:00 29622698 /lib/libc-2.1.3.so
401c0000-401c4000 r--p 001c0000 fc:00 29622698 /lib/libc-2.1.3.so
401c4000-401c6000 rw-p 001c0000 fc:00 29622698 /lib/libc-2.1.3.so
401cc000-401f2000 r-xp 00000000 fc:00 29622674 /lib/libc-2.1.3.so
401f2000-401f3000 r--p 00025000 fc:00 29622674 /lib/libc-2.1.3.so
401f3000-401f4000 r--p 00026000 fc:00 29622674 /lib/libc-2.1.3.so
bffdf000-c0000000 rw-p 00000000 00:00 0 [stack]
```

Offset: if the region was mapped from a file (using mmap), this is the offset in the file

Example of Address Mapping

```
$ ./a.out >null &
[1] 673
$ cat /proc/673/maps
00400000-00301000 r-xp 00000000 fc:00 353
00600000-00601000 r--p 00000000 fc:00 353
00601000-00602000 rw-p 00001000 fc:00 353624 /home/ee415/a.out

012d7000-012f8000 rw-p 00000000 00:00 0 [heap]

40000000-401c0000 r-xp 00000000 fc:00 29622698 /lib/ld-2.1.3.so
401c0000-401c4000 r--p 001c0000 fc:00 29622698 /lib/ld-2.1.3.so
401c4000-401c6000 rw-p 001c0000 fc:00 29622698 /lib/ld-2.1.3.so

401cc000-401f2000 r-xp 00000000 fc:00 29622674 /lib/libc-2.1.3.so
401f2000-401f3000 r--p 00025000 fc:00 29622674 /lib/libc-2.1.3.so
401f3000-401f4000 r--p 00026000 fc:00 29622674 /lib/libc-2.1.3.so

bffdf000-c0000000 rw-p 00000000 00:00 0 [stack]
```

Device: if the region was mapped from a file, this is the device number where the file lives (fc:00 → SSD in this system)

Example of Address Mapping

```
$ ./a.out >null &
```

```
[1] 673
```

```
$ cat /proc/673/m
```

```
00400000-00301000
```

```
00600000-00601000
```

```
00601000-00602000
```

Inode: if the region was mapped from a file, this is the file number

```
012d7000-012f8000
```

```
rw-p
```

```
00000000 00:00 0
```

```
/home/ee415/a.out
```

```
/home/ee415/a.out
```

```
/home/ee415/a.out
```

[heap]

```
40000000-401c0000
```

```
r-xp
```

```
00000000 fc:00 29622698
```

```
/lib/ld-2.1.3.so
```

```
401c0000-401c4000
```

```
r--p
```

```
001c0000 fc:00 29622698
```

```
/lib/ld-2.1.3.so
```

```
401c4000-401c6000
```

```
rw-p
```

```
001c0000 fc:00 29622698
```

```
/lib/ld-2.1.3.so
```

```
401cc000-401f2000
```

```
r-xp
```

```
00000000 fc:00 29622674
```

```
/lib/libc-2.1.3.so
```

```
401f2000-401f3000
```

```
r--p
```

```
00025000 fc:00 29622674
```

```
/lib/libc-2.1.3.so
```

```
401f3000-401f4000
```

```
r--p
```

```
00026000 fc:00 29622674
```

```
/lib/libc-2.1.3.so
```

```
bffdf000-c0000000
```

```
rw-p
```

```
00000000 00:00 0
```

[stack]

Example of Address Mapping

```
$ ./a.out >null &  
[1] 673  
$ cat /proc/673/maps  
00400000-00301000  
00600000-00601000  
00601000-00602000  
  
012d7000-012f8000    rw-p    00000000 00:00  
40000000-401c0000    r-xp   00000000 fc:00 29622698  
401c0000-401c4000    r--p   001c0000 fc:00 29622698  
401c4000-401c6000    rw-p   001c0000 fc:00 29622698  
  
401cc000-401f2000    r-xp   00000000 fc:00 29622674  
401f2000-401f3000    r--p   00025000 fc:00 29622674  
401f3000-401f4000    r--p   00026000 fc:00 29622674  
  
bffdf000-c0000000    rw-p    00000000 00:00 0
```

Pathname: if the region was mapped from a file, this is the name of the file. There are also special regions with names like [heap], [stack]

/home/ee415/a.out
/home/ee415/a.out
/home/ee415/a.out

[heap]

/lib/ld-2.1.3.so
/lib/ld-2.1.3.so
/lib/ld-2.1.3.so

/lib/libc-2.1.3.so
/lib/libc-2.1.3.so
/lib/libc-2.1.3.so

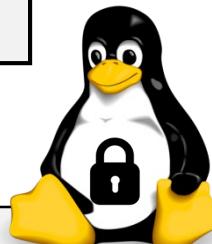
[stack]

Example of Address Mapping

\$./a.out >null &			
[1] 672			
File-related information will be covered in the future lecture (e.g., File System) ☺			
00000000-00001000	... p	00000000 fc:00 85362479	/home/ee415/a.out
012d7000-012f8000	rw-p	00000000 fc:00 85362479	/home/ee415/a.out
40000000-401c0000	r-xp	00000000 fc:00 85362479	/home/ee415/a.out
401c0000-401c4000	r--p	00000000 00:00 0	[heap]
401c4000-401c6000	rw-p	00000000 fc:00 29622698	/lib/ld-2.1.3.so
401cc000-401f2000	r-xp	00000000 fc:00 29622698	/lib/ld-2.1.3.so
401f2000-401f3000	r--p	00000000 fc:00 29622698	/lib/ld-2.1.3.so
401f3000-401f4000	r--p	00000000 fc:00 29622674	/lib/libc-2.1.3.so
bffdf000-c0000000	rw-p	00000000 fc:00 29622674	/lib/libc-2.1.3.so
		00000000 fc:00 29622674	/lib/libc-2.1.3.so
		00000000 fc:00 29622674	/lib/libc-2.1.3.so
		00000000 00:00 0	[stack]

Virtual Memory Area (VMA)

VMA is represented by
an object of type struct
`vm_area_struct`



`vm_area_struct`

```
unsigned long          vm_start;
unsigned long          vm_end;
pgprot_t               vm_page_prot;

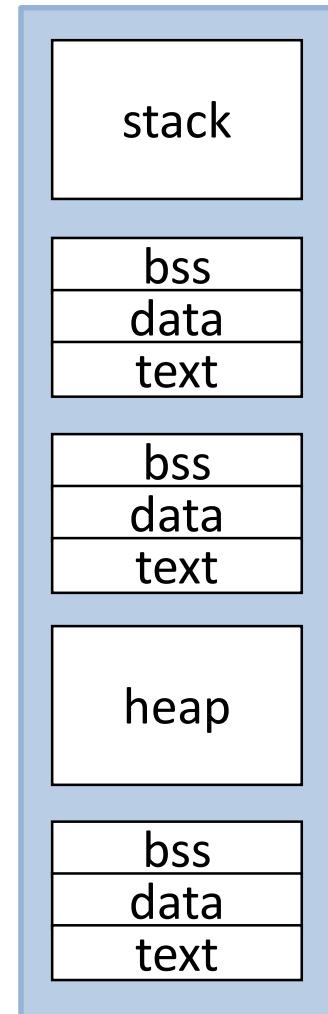
struct file            *vm_file;

struct vm_operations_struct *vm_ops;

struct mm_struct        *vm_mm;
struct vm_area_struct   *vm_next;
struct vm_area_struct   *vm_prev;
```

TRY ME! search `vm_area_struct` in "include/linux/mm_types.h"

Memory map



Example: VMA of a.out

Memory map

```
vm_area_struct  
  
unsigned long          vm_start;  
unsigned long          vm_end;  
pgprot_t               vm_page_prot;  
  
struct file            *vm_file;  
  
struct vm_operations_struct *vm_ops;  
  
struct mm_struct        *vm_mm;  
struct vm_area_struct   *vm_next;  
struct vm_area_struct   *vm_prev;
```

vm_page_prot
(permission)

→ r-xp

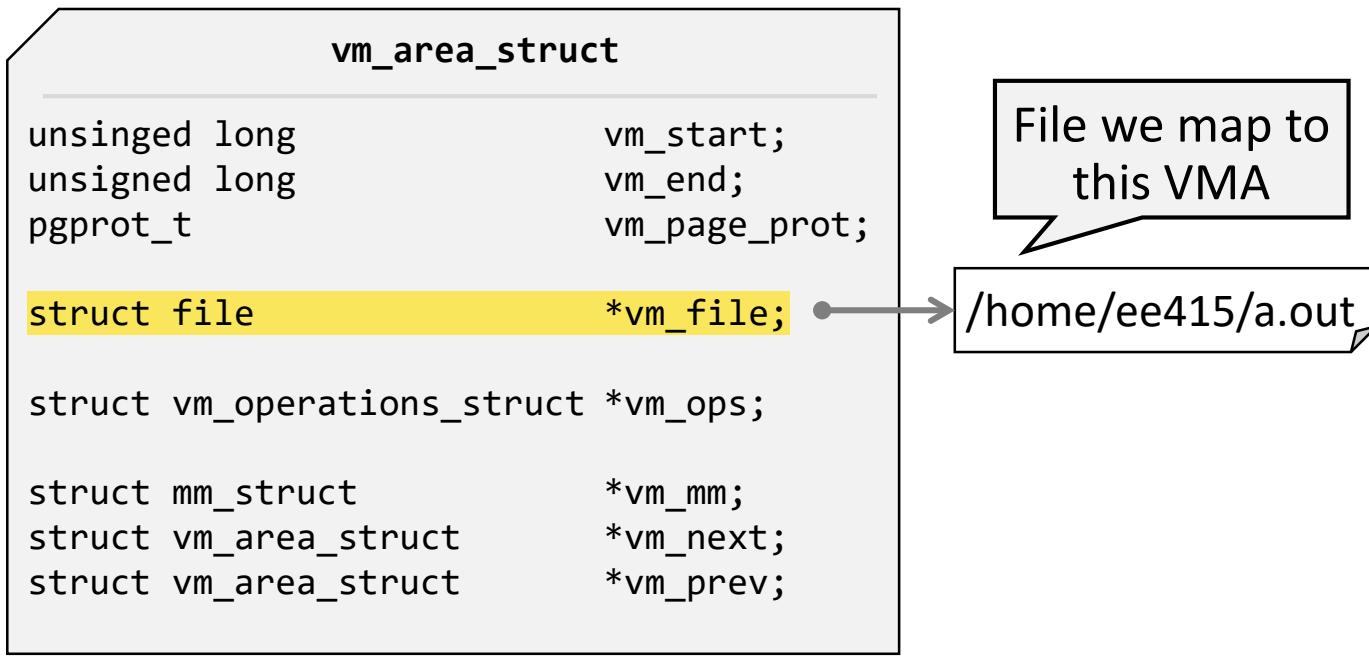
vm_end

vm_start



Example: VMA of a.out

Memory map



Example: VMA of a.out

Memory map

```
vm_area_struct  
  
unsigned long          vm_start;  
unsigned long          vm_end;  
pgprot_t               vm_page_prot;  
struct file            *vm_file;  
  
struct vm_operations_struct *vm_ops;  
  
struct mm_struct        *vm_mm;  
struct vm_area_struct   *vm_next;  
struct vm_area_struct   *vm_prev;
```

Dispatching vector of
functions/methods
on this object



Example: VMA of a.out

Memory map

stack

bss

data

text

bss

data

text

heap

bss

data

text

vm_operations_struct

```
void (*open)(struct vm_area_struct * area);  
void (*close)(struct vm_area_struct * area);  
vm_fault_t (*fault)(struct vm_fault * vmf);  
...
```

TRY ME! search `vm_operations_struct` in "include/linux/mm.h"

functions/methods
on this object

unsigned long
unsigned long
pgprot_t

struct file

struct vm_operations_struct *vm_ops;

Struct mm_struct
struct vm_area_struct
struct vm_area_struct
 *vm_mm;
 *vm_next;
 *vm_prev;

Example: VMA of a.out

Memory map

When the VMA is added/removed to an address space, OS executes this function

vm_operations_struct

```
void (*open)(struct vm_area_struct * area);
void (*close)(struct vm_area_struct * area);
vm_fault_t (*fault)(struct vm_fault * vmf);
...
```

TRY ME! search `vm_operations_struct` in "include/linux/mm.h"

functions/methods
on this object

unsigned long
unsigned long
pgprot_t

struct file

struct vm_operations_struct *vm_ops;

Struct mm_struct
struct vm_area_struct
struct vm_area_struct

*vm_mm;
*vm_next;
*vm_prev;

Example: VMA of a.out

unsigned long
unsigned long
pgprot_t

struct file

struct vm_operations_struct *vm_ops;

Struct mm_struct

struct vm_area_struct
struct vm_area_struct

TRY ME! search vm_operations_struct in "include/linux/mm.h"

*vm_file;

on this object

vm_operations_struct

```
void (*open)(struct vm_area_struct * area);
void (*close)(struct vm_area_struct * area);
vm_fault_t (*fault)(struct vm_fault * vmf);
...
...
```

This is **page fault handler**, what we learned 😊

Memory map

stack

bss

data

text

heap

bss

data

text



Example: VMA of a.out

Memory map

```
vm_area_struct

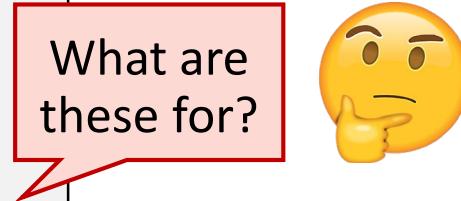
unsigned long          vm_start;
unsigned long          vm_end;
pgprot_t               vm_page_prot;

struct file            *vm_file;

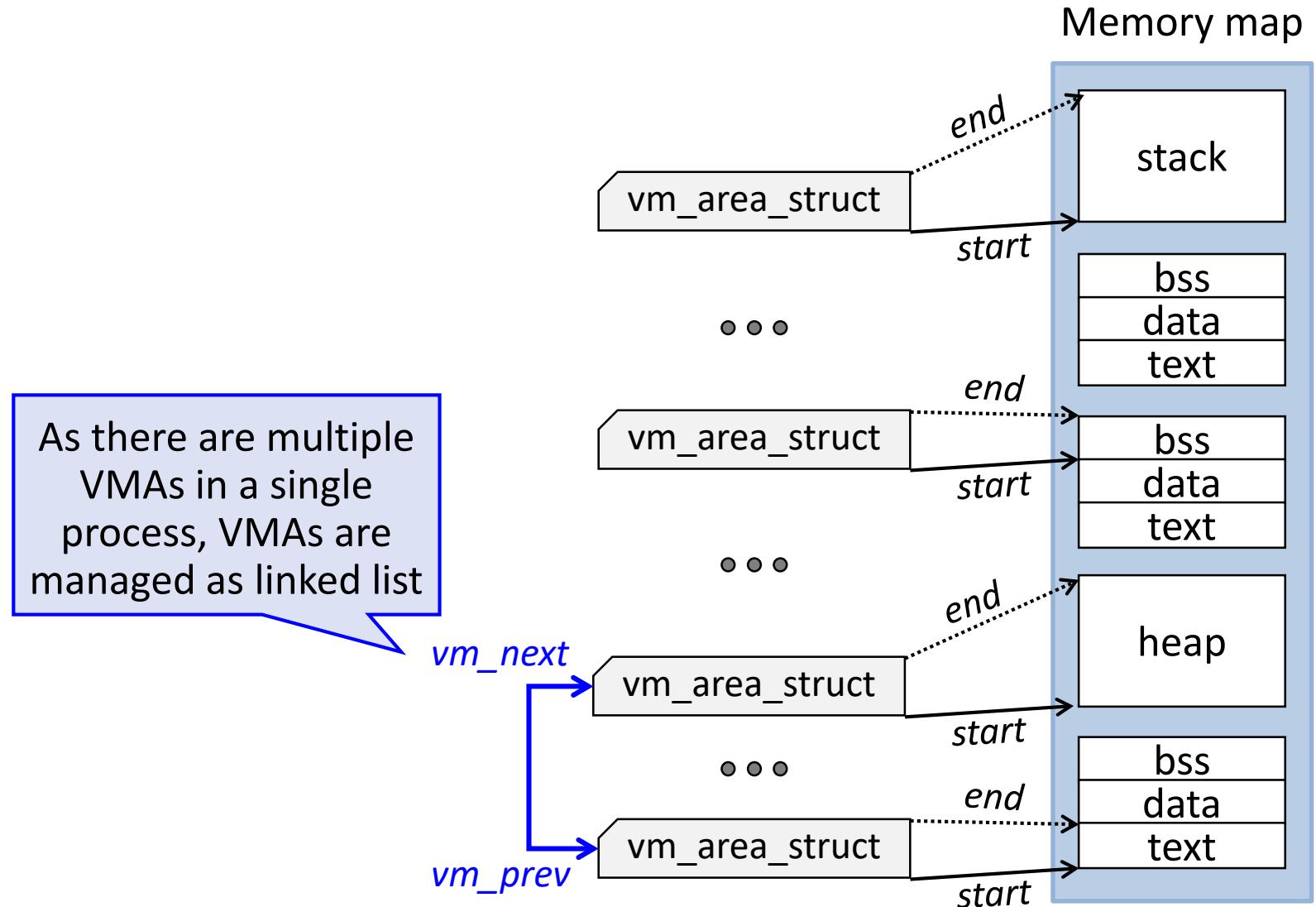
struct vm_operations_struct *vm_ops;

struct mm_struct        *vm_mm;
struct vm_area_struct   *vm_next;
struct vm_area_struct   *vm_prev;
```

What are
these for?

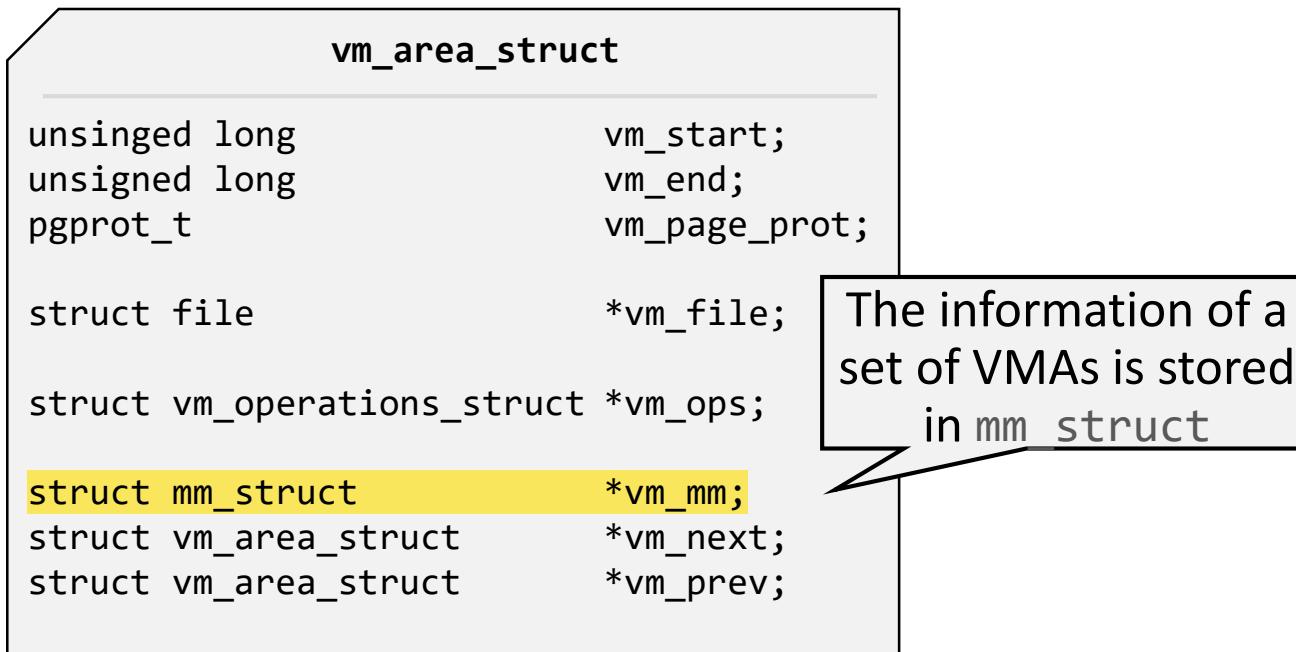


Example: VMA of a.out



Example: VMA of a.out

Memory map





Memory Management: mm_struct

task_struct

```
pid_t           pid;
long            state;
struct thread_struct thread;

pid_t           tgid;
struct list_head tasks;

struct task_struct *parent;
struct list_head children;
struct list_head sibling;

struct mm_struct *mm;
struct files_struct *files;
struct sighand_struct *sighand;
struct signal_struct *signal;
void            *stack
```

In other words, the kernel represents a process's address space with a data structure, called the memory descriptor (mm_struct)

TRY ME! search task_struct in "include/linux/sched.h"

Memory Management: mm_struct

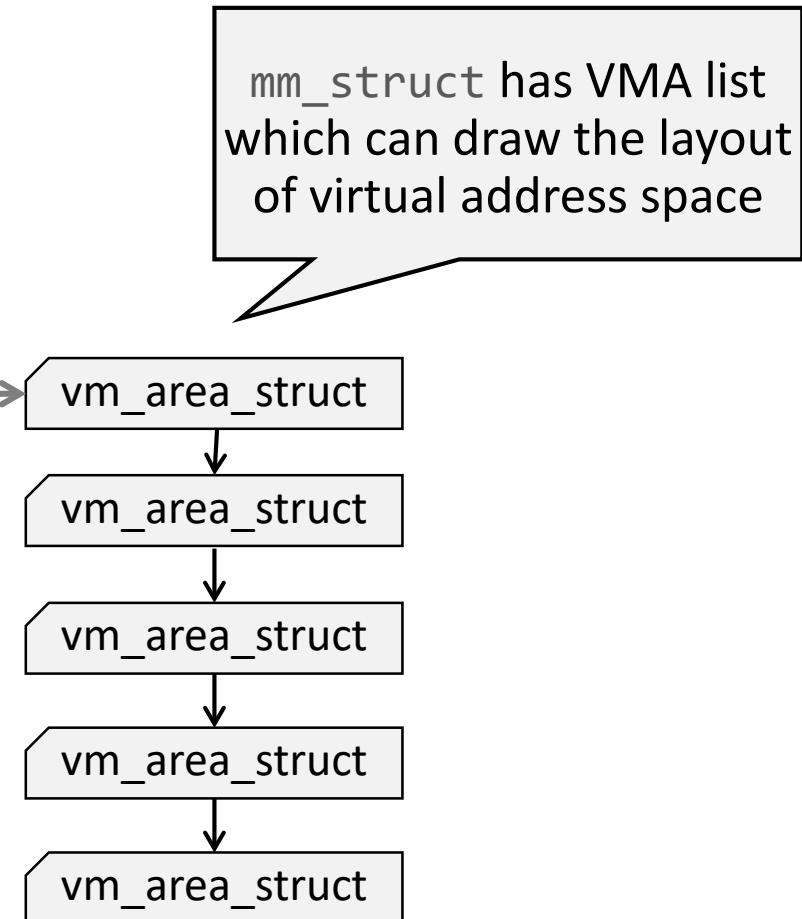
```
task_struct  
-----  
pid_t           pid;  
long            state;  
struct thread_struct    thread;  
  
pid_t           tgid;  
struct list_head tasks;  
  
struct task_struct *parent;  
struct list_head children;  
struct list_head sibling;  
  
struct mm_struct *mm;  ●  
struct files_struct *files;  
struct sighand_struct *sighand;  
struct signal_struct *signal;  
void             *stack
```

```
mm_struct  
-----  
/* Page table address */  
pgd_t           *pgd;  
/* List of VMAs */  
struct vm_area_struct *mmap;  
struct rb_root   mm_rb;  
/* Pointer of VMAs */  
unsigned long    task_size;  
unsigned long    start_code;  
unsigned long    end_code;  
unsigned long    start_data;  
unsigned long    end_data;  
unsigned long    start_brk;  
unsigned long    brk;  
unsigned long    start_stack;  
unsigned long    mmap_base;
```

TRY ME! search mm_struct in "include/linux/mm_types.h"

Memory Management: mm_struct

```
mm_struct
/* Page table address */
pgd_t *pgd;
/* List of VMAs */
struct vm_area_struct *mmap;
struct rb_root mm_rb;
/* Pointer of VMAs */
unsigned long task_size;
unsigned long start_code;
unsigned long end_code;
unsigned long start_data;
unsigned long end_data;
unsigned long start_brk;
unsigned long brk;
unsigned long start_stack;
unsigned long mmap_base;
```



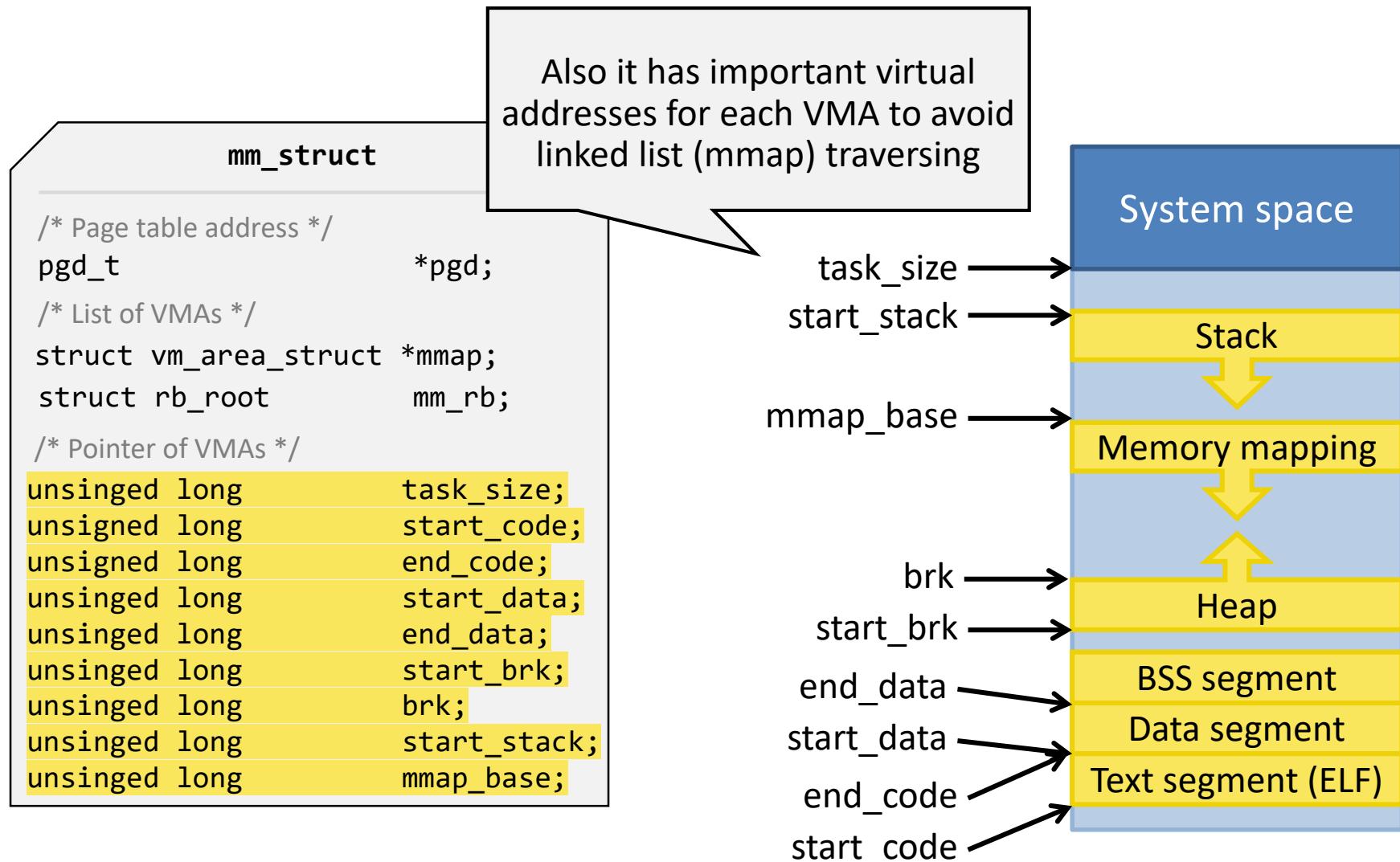
Memory Management: mm_struct

mm_struct

```
/* Page table address */  
pgd_t *pgd;  
  
/* List of VMAs */  
struct vm_area_struct *mmap;  
struct rb_root mm_rb;  
  
/* Pointer of VMAs */  
unsigned long task_size;  
unsigned long start_code;  
unsigned long end_code;  
unsigned long start_data;  
unsigned long end_data;  
unsigned long start_brk;  
unsigned long brk;  
unsigned long start_stack;  
unsigned long mmap_base;
```

mm_struct has the address of page directory (page table) → which will be filling in CR3 register during context switching

Memory Management: mm_struct



Physical Memory Management

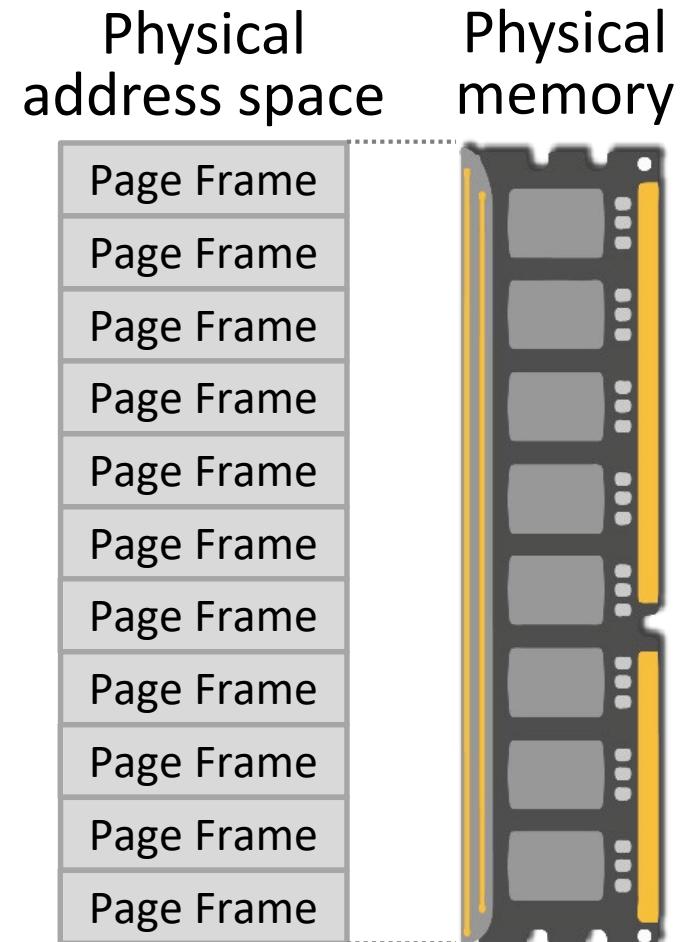
Virtual memory management
(Process view point)



Physical memory management
(CPU view point)

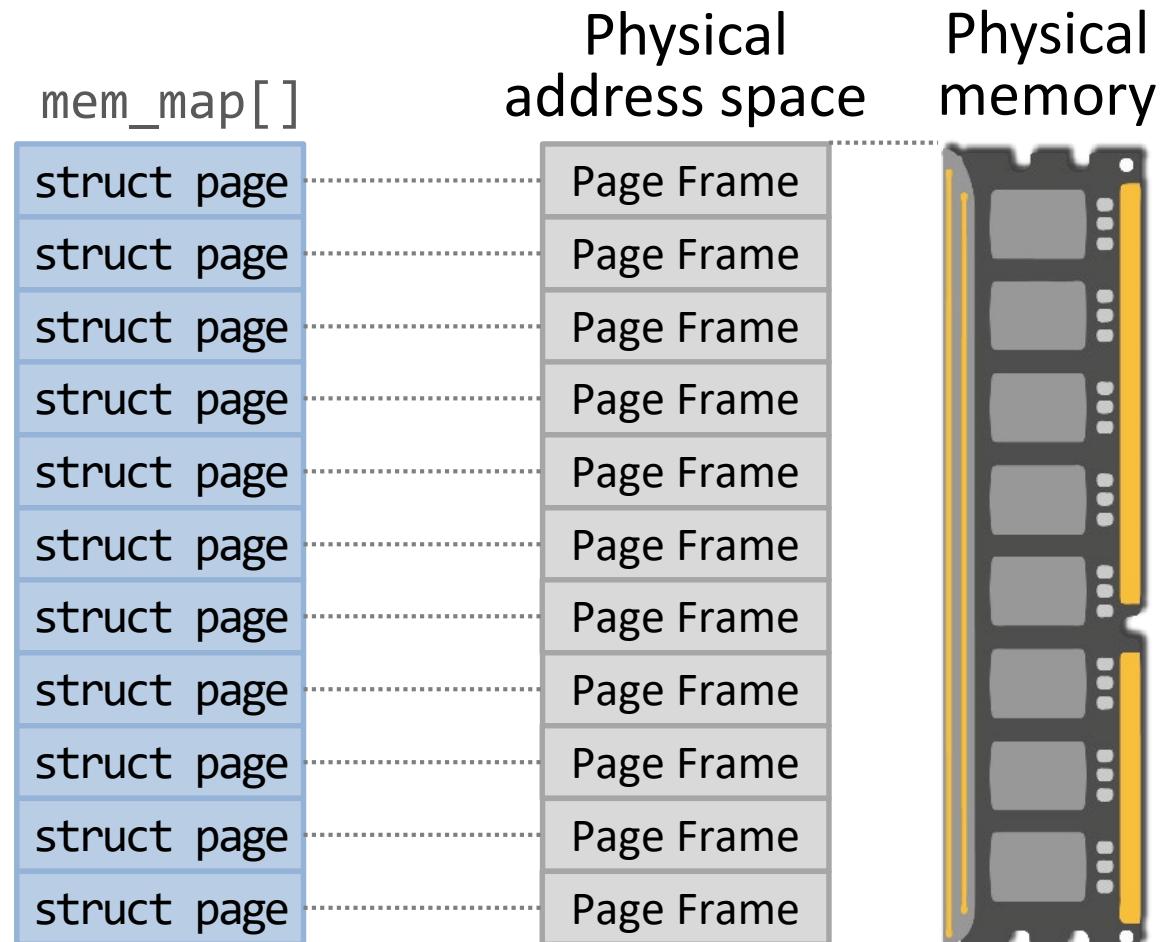
Physical Memory Management

Memory is divided by
4KB page frames



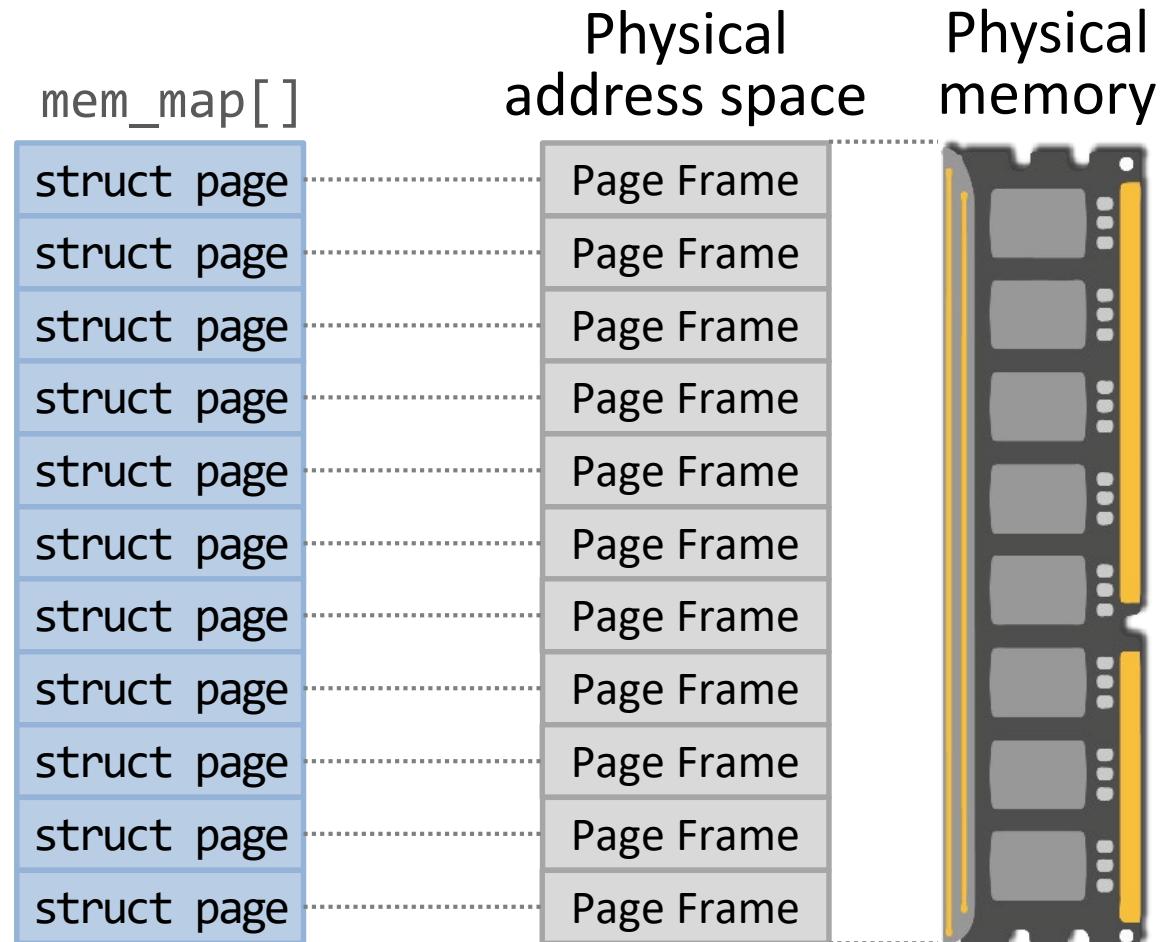
Physical Memory Management

Each physical page frame in the system has a `struct page` associated with it



Physical Memory Management

All `struct pages` in the system are managed by global variable `mem_map[]` and can be easily accessed by referring PFN index (`mem_map[PFN]`)





Memory Management: struct page

struct page

```
unsigned long           flags;  
atomic_t                _refcount;  
void                     *virtual;  
...
```

With this struct page, OS can figure out the purpose of physical page frame and whether the page frame is allocated or not



TRY ME! search struct page in "include/linux/mm_types.h"



Memory Management: struct page

```
struct page  
{  
    unsinged long      flags;  
    atomic_t           _refcount;  
    void              *virtual;  
    ...  
}
```

The flags field stores the status of the page

TRY ME! search struct page in "include/linux/mm_types.h"

Page flags	Explanation
PG_dirty	This page is modified
PG_reserved	This page is special in some way (e.g., kernel text) → Cannot be swapped out
PG_reclaim	This page is to be reclaimed ASAP → Selected by page replacement policy
...	

TRY ME! search page_flags in "include/linux/page-flags.h"

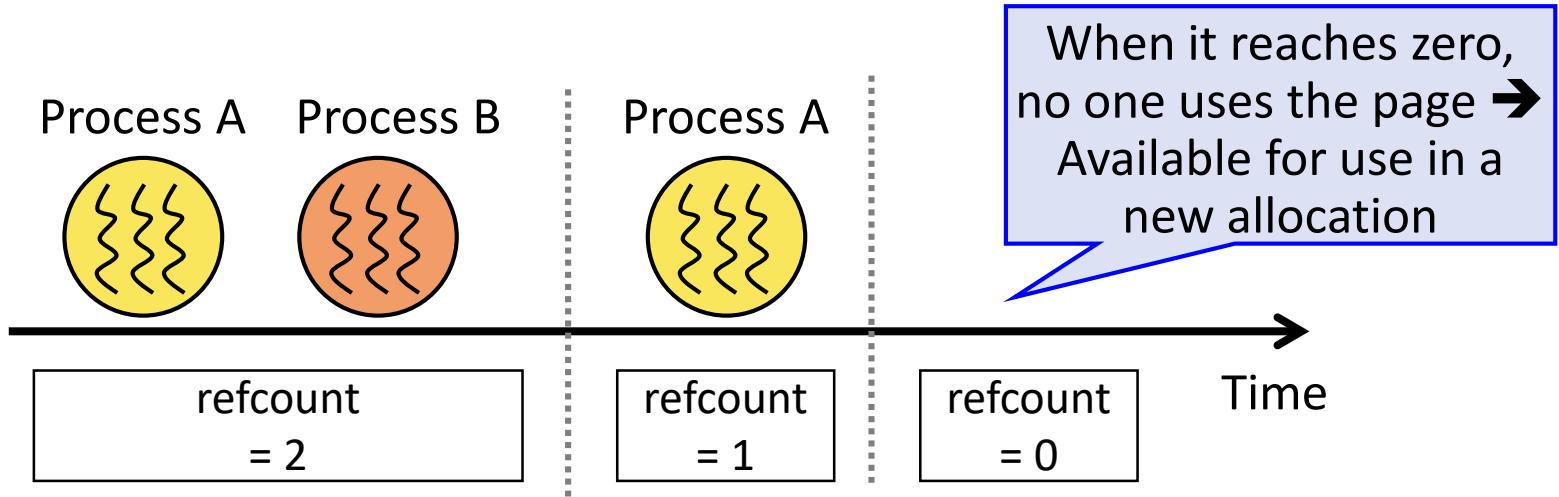


Memory Management: struct page

```
struct page  
{  
    unsigned long      flags;  
    atomic_t           _refcount;  
    void              *virtual;  
    ...  
}
```

The `_refcount` field stores the usage count of the page

TRY ME! search `struct page` in "include/linux/mm_types.h"



Memory Management: struct page

```
struct page
```

```
unsigned long           flags;  
atomic_t                _refcount;  
void                   *virtual;  
...
```

The virtual field is the page's virtual address

TRY ME! search struct page in "include/linux/mm_types.h"

In default, struct page does not include the **owner information** to reduce the memory requirement of struct page. (It can be enabled with “page_owner = on” booting parameter); it’s often used for debugging



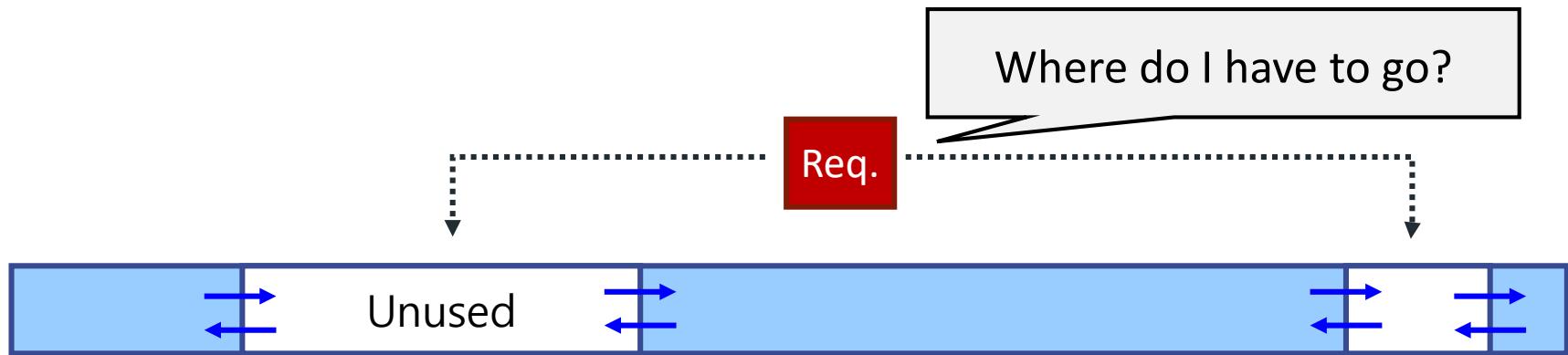
Design Consideration (Page Alloc)

- We require reducing
 - internal fragmentation
 - external fragmentation
 - the overhead imposed by tracking memory
- [REMIND] internal fragmentation
 - Memory is allocated but unused
 - Fixed allocation size
- [REMIND] external fragmentation
 - Memory remains unallocated
 - Variable allocation sizes



Basic Allocation Algorithms

- Best Fit
 - Pick the “smallest” area/hole that will satisfy the request
- Consideration
 - Should search entire list every time
 - Tend to leave lots of small memory area (external fragmentation)





Basic Allocation Algorithms

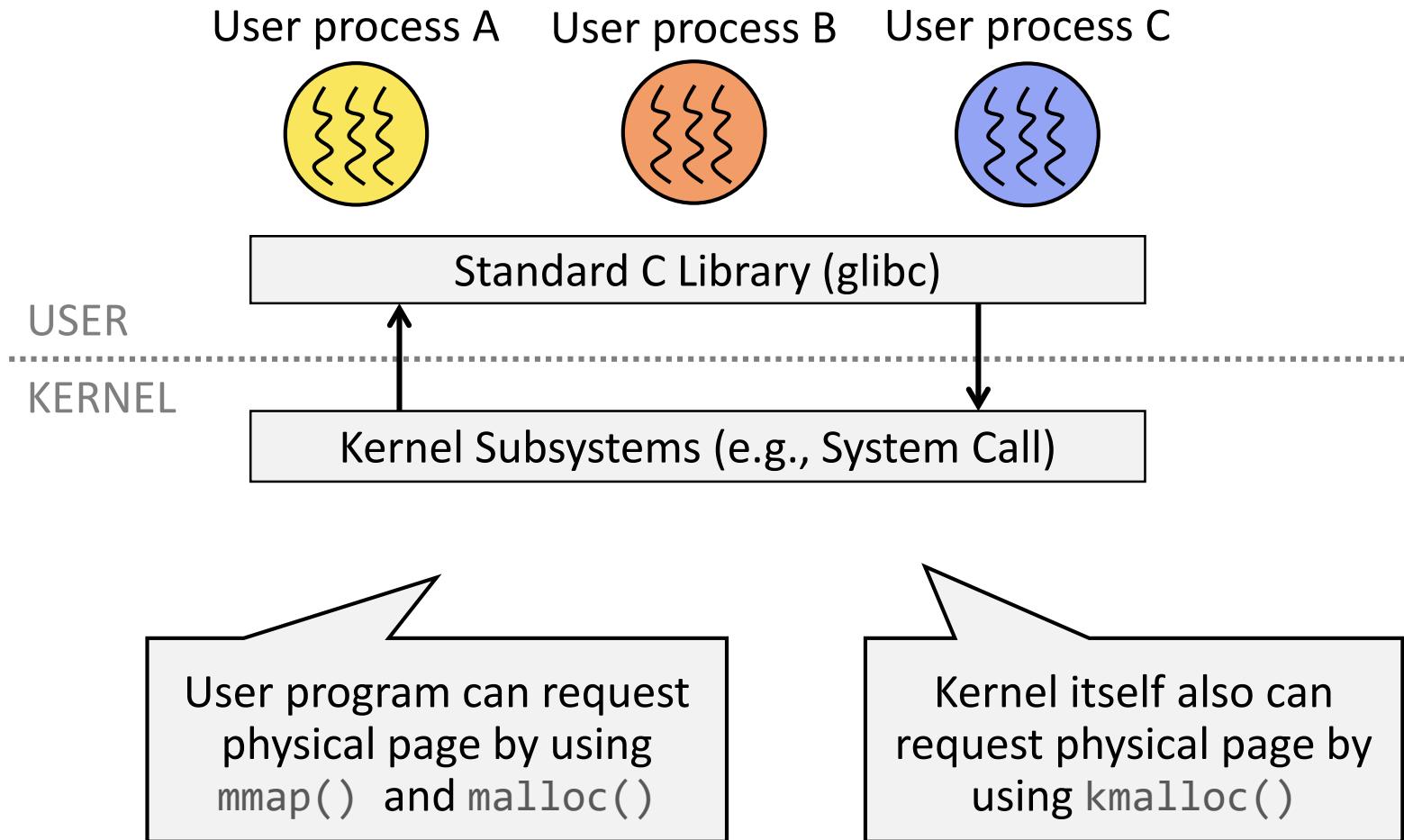
- Worst Fit
 - Pick the largest hole to satisfy the request
 - Consideration
 - Yet, it should search entire list
 - It still lead fragmentation issues
 - Practically, it's worse than the best fit
- First Fit
 - Pick the first hole large enough to satisfy the request
 - Consideration
 - Much faster than the best and worst fit
 - Yet, it also has fragmentations just like the best fit
- Next Fit
 - Exactly same with the first fit except start search from where the last search left off



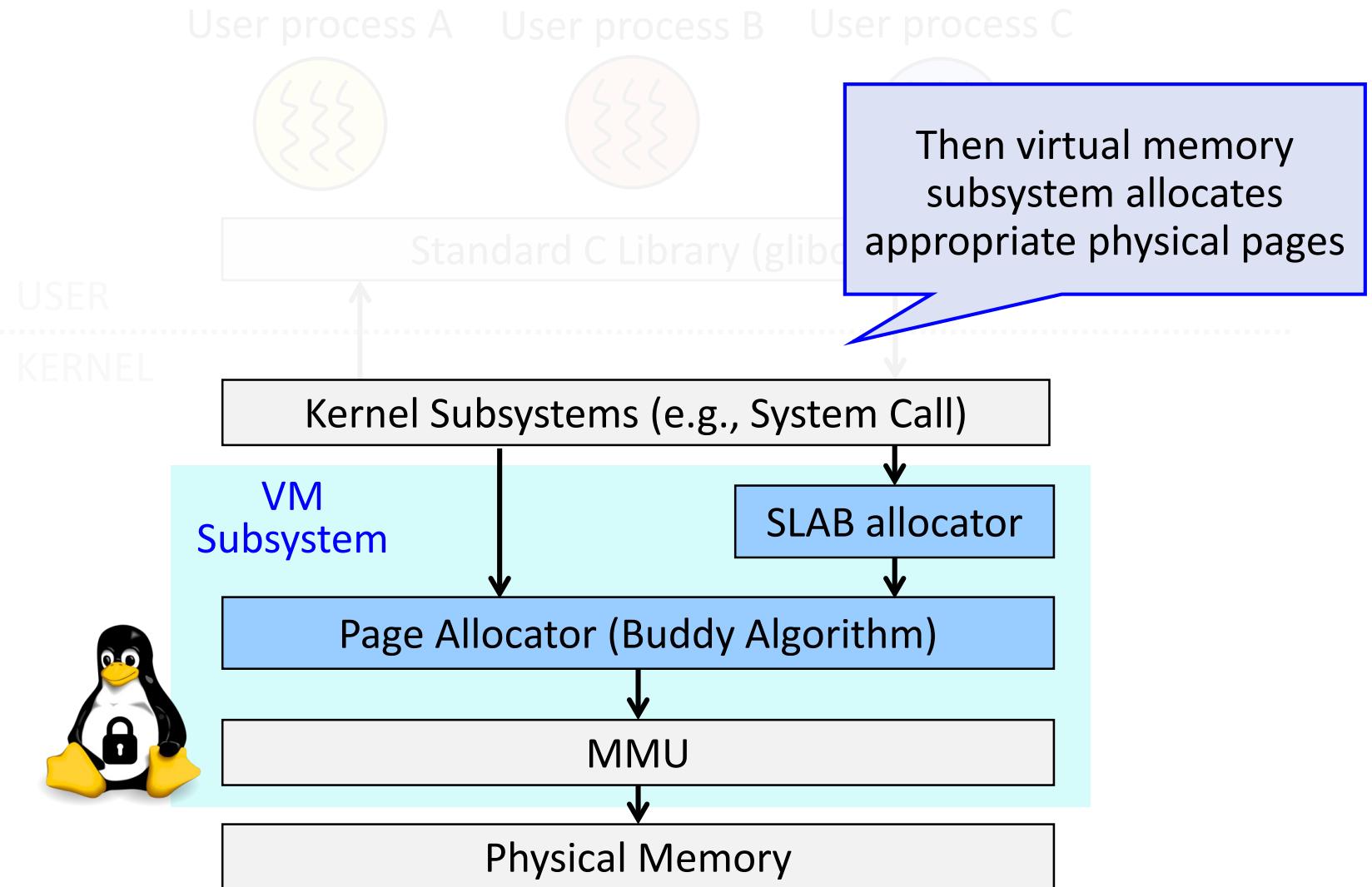
Solution?

- All the previous algorithms introduce external fragmentation
- Why don't we allocate "multiple" fixed size of hole
 - Keep multiple list of different size holes for different size requests
 - Take hole from a list that most closely matches size of request
 - Yet, it leads to internal fragmentation, but not too much
 - this type of fragmentation can be addressed by compaction

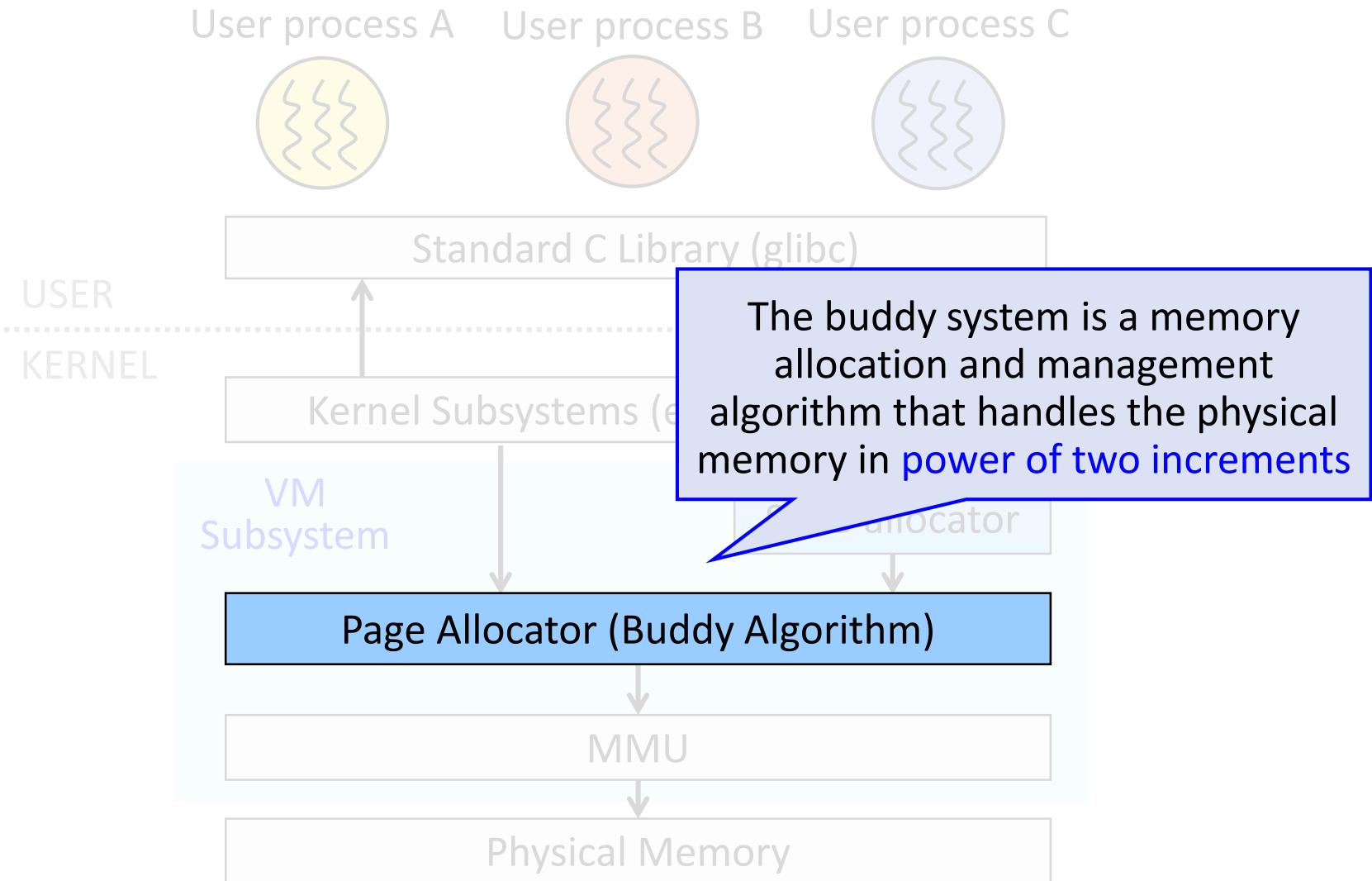
Physical Page Allocations



Physical Page Allocations

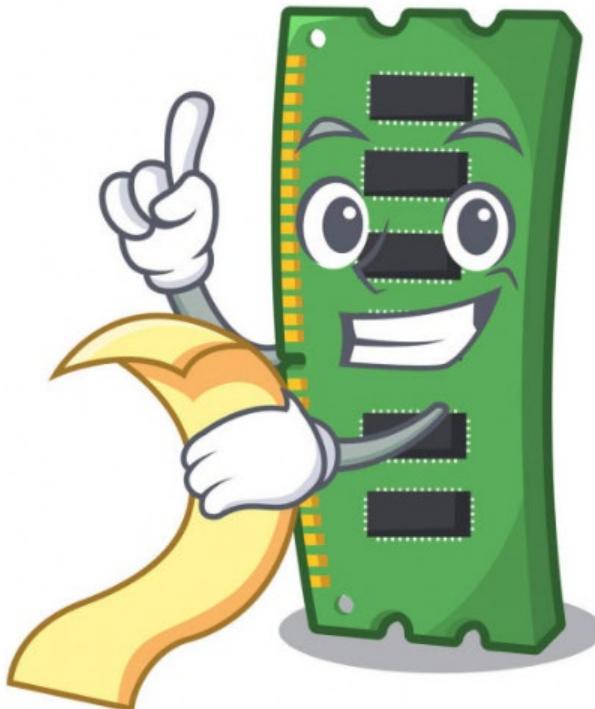


Buddy Allocator



Goal of Buddy Allocator

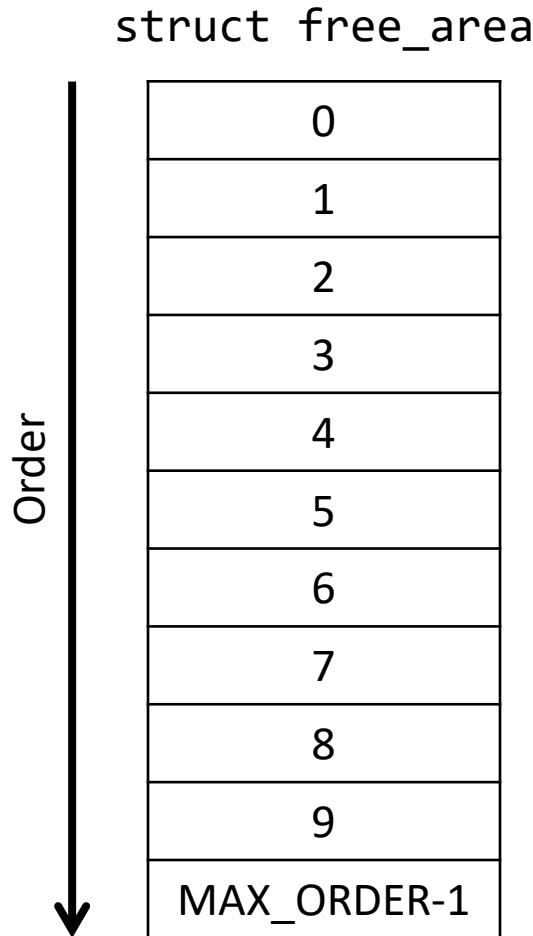
- With page-based virtual memory, processes can freely get available discontinuous pages



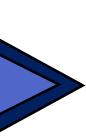
But, allocates continuous pages as much as possible, if there are available chunk of pages (**buddies**)



Buddy Allocator Implementation

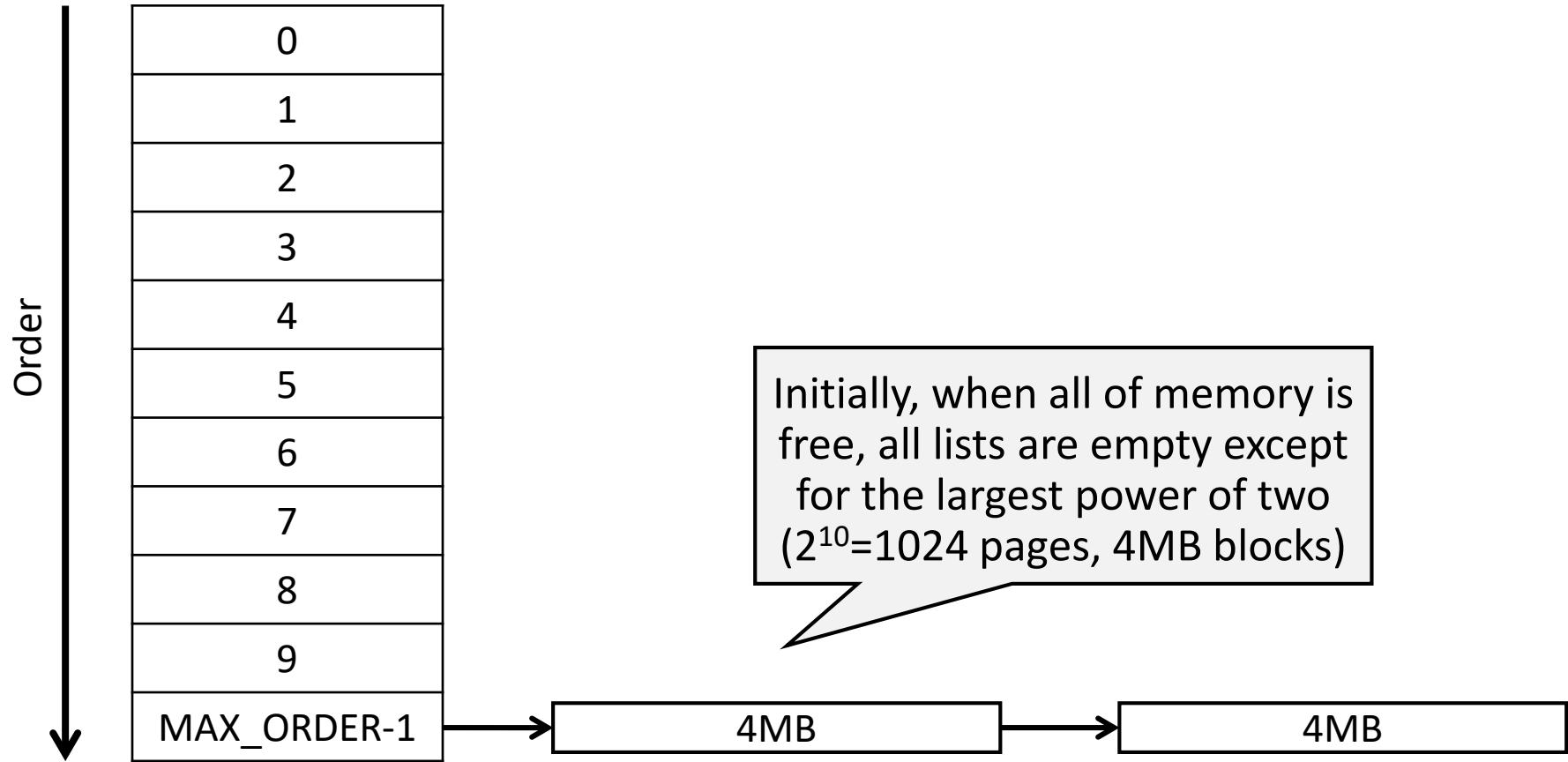


A memory manager (e.g., the Linux page allocator) keeps lists of free blocks that are sizes of powers of two ($2^0, 2^1, 2^2, \dots$)



Buddy Allocator Implementation

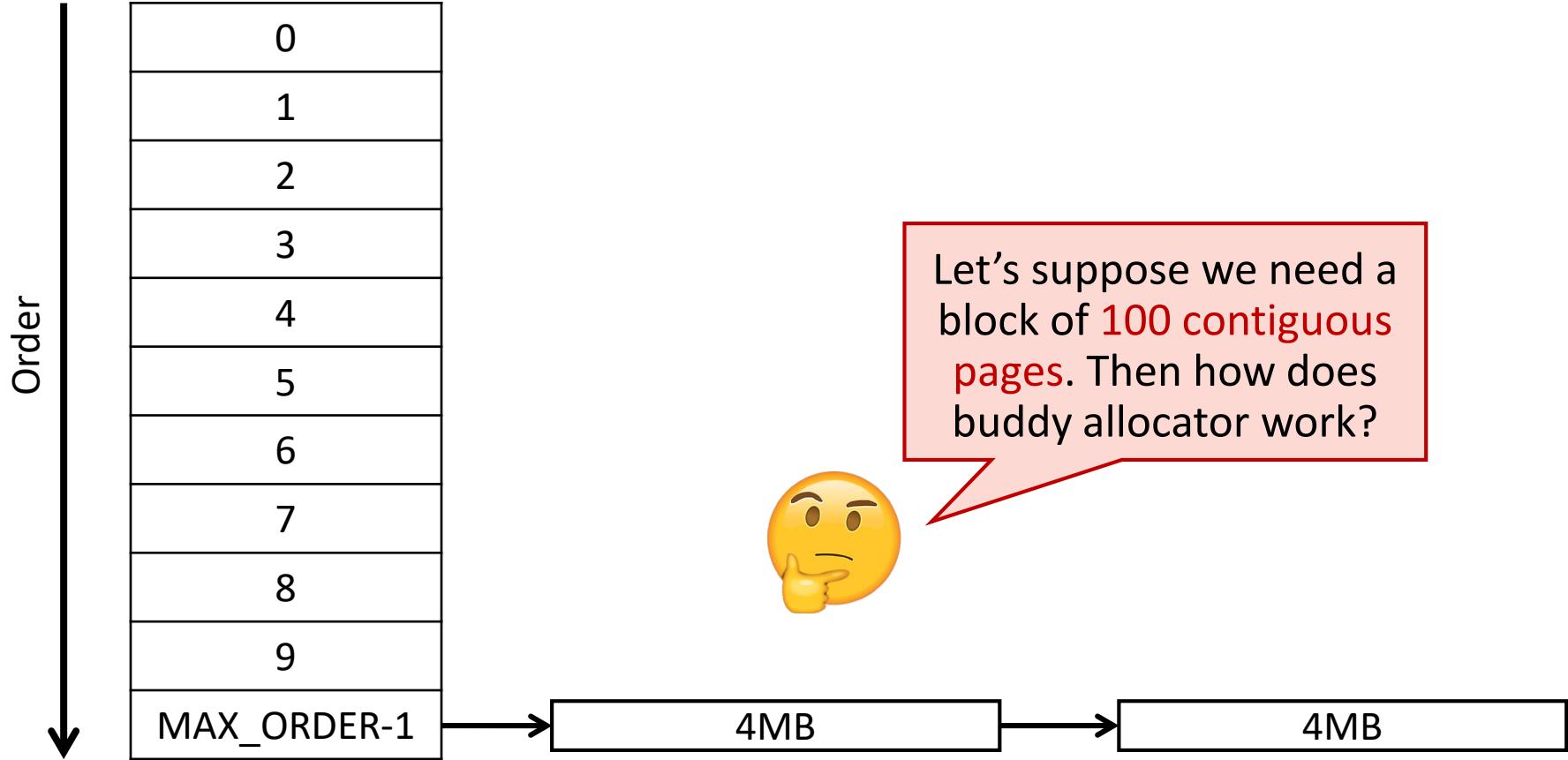
```
struct free_area
```





Buddy Allocator Implementation

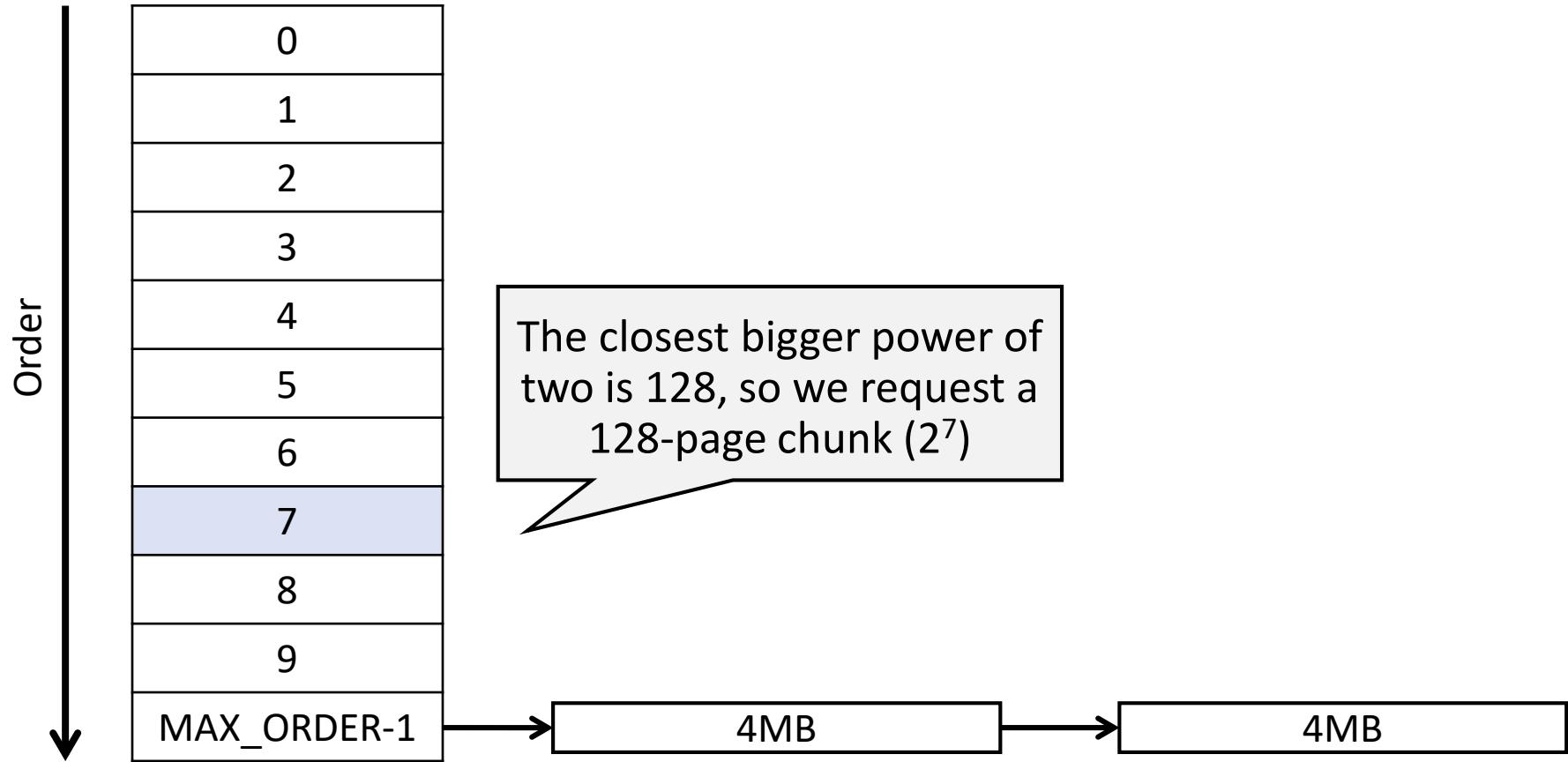
```
struct free_area
```





Buddy Allocator Implementation

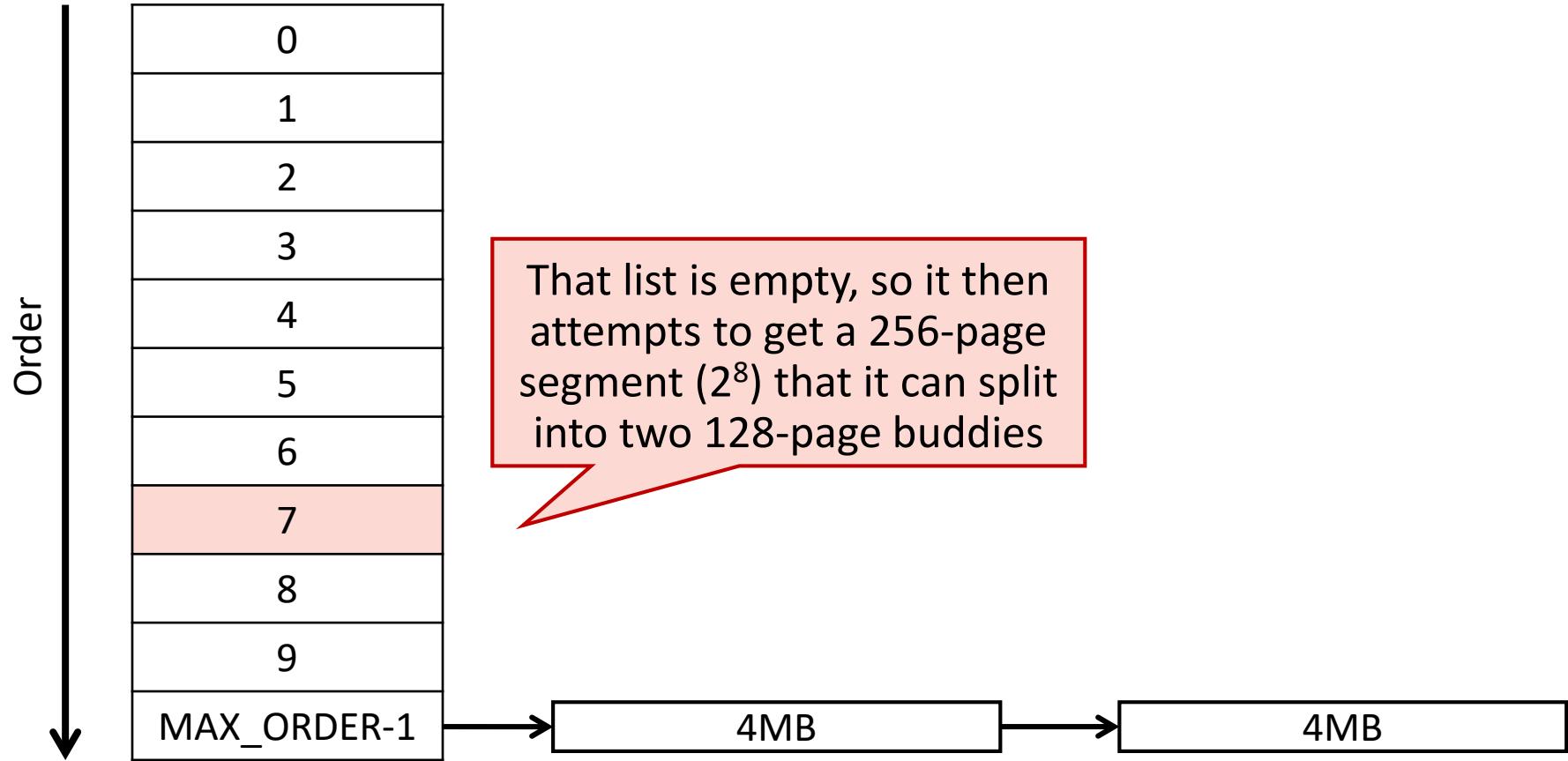
```
struct free_area
```





Buddy Allocator Implementation

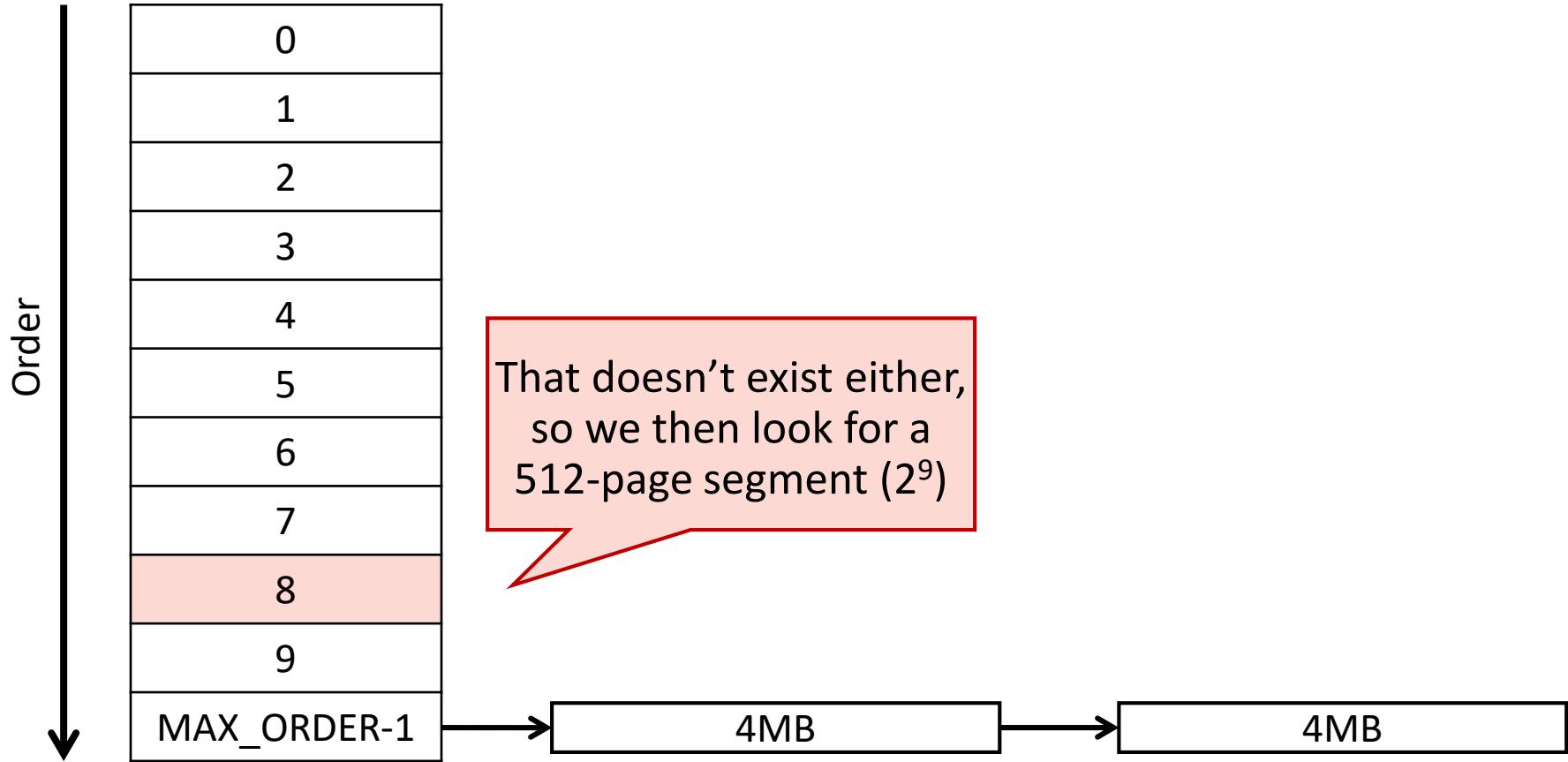
```
struct free_area
```





Buddy Allocator Implementation

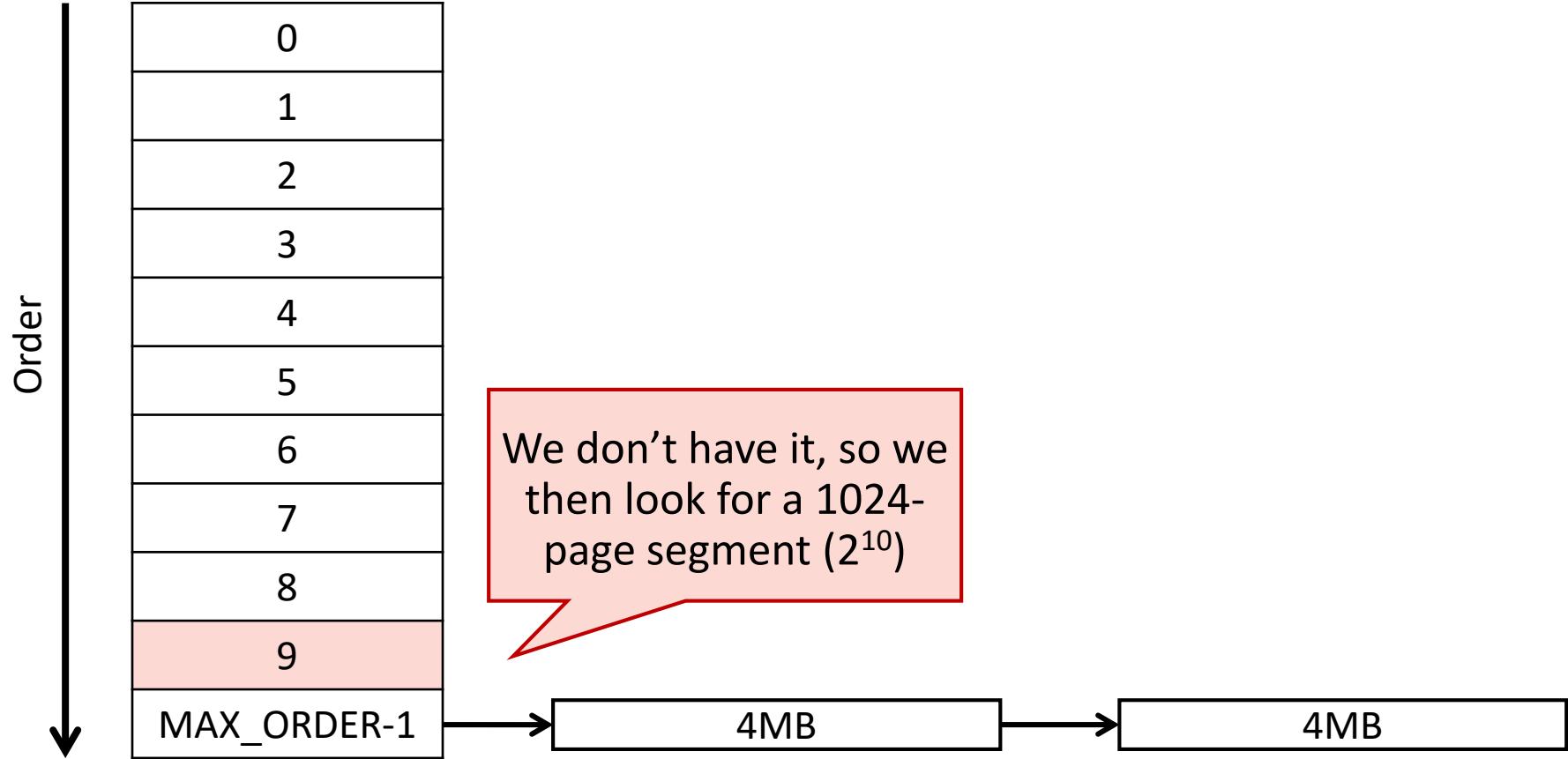
```
struct free_area
```





Buddy Allocator Implementation

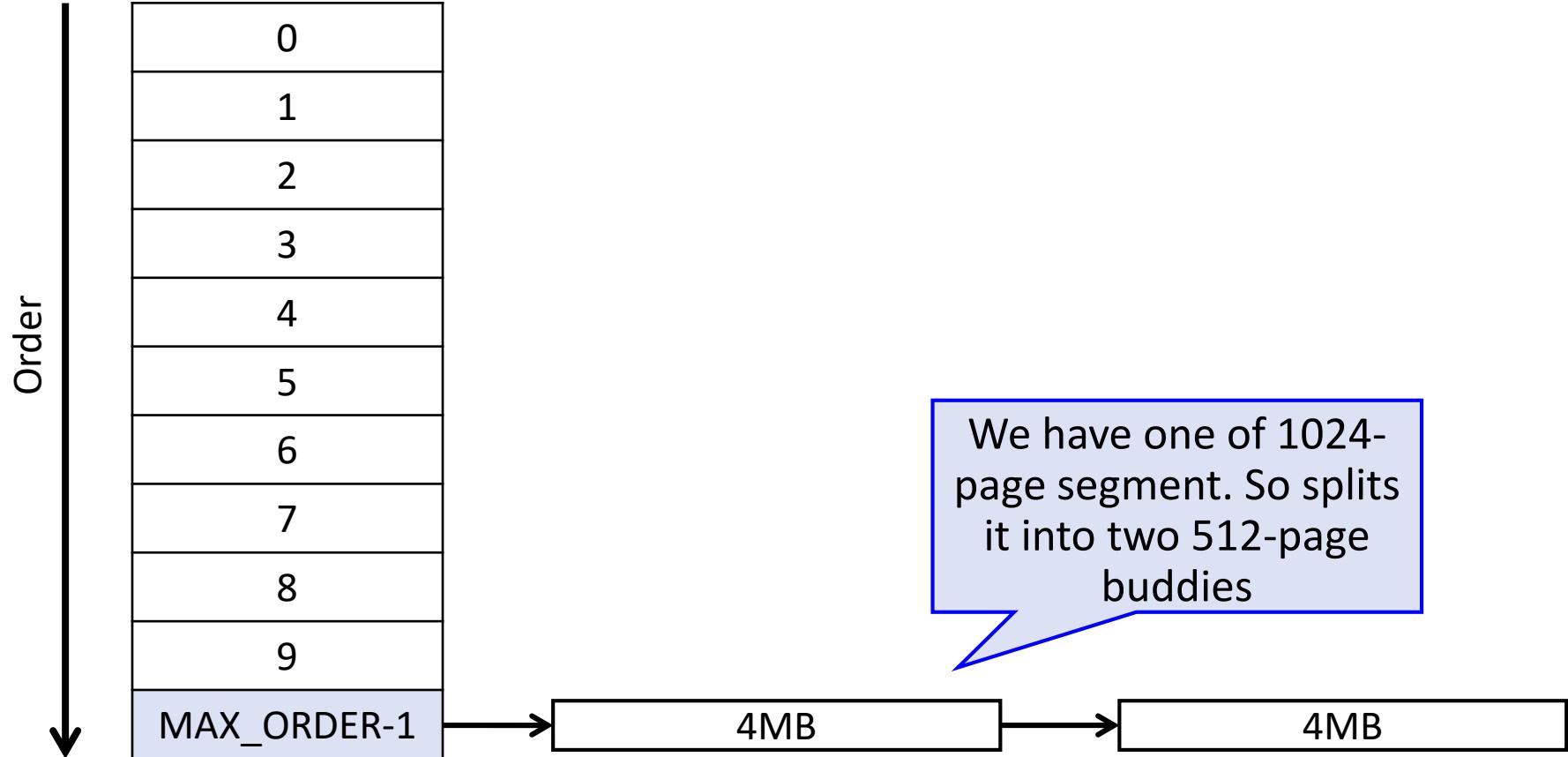
```
struct free_area
```





Buddy Allocator Implementation

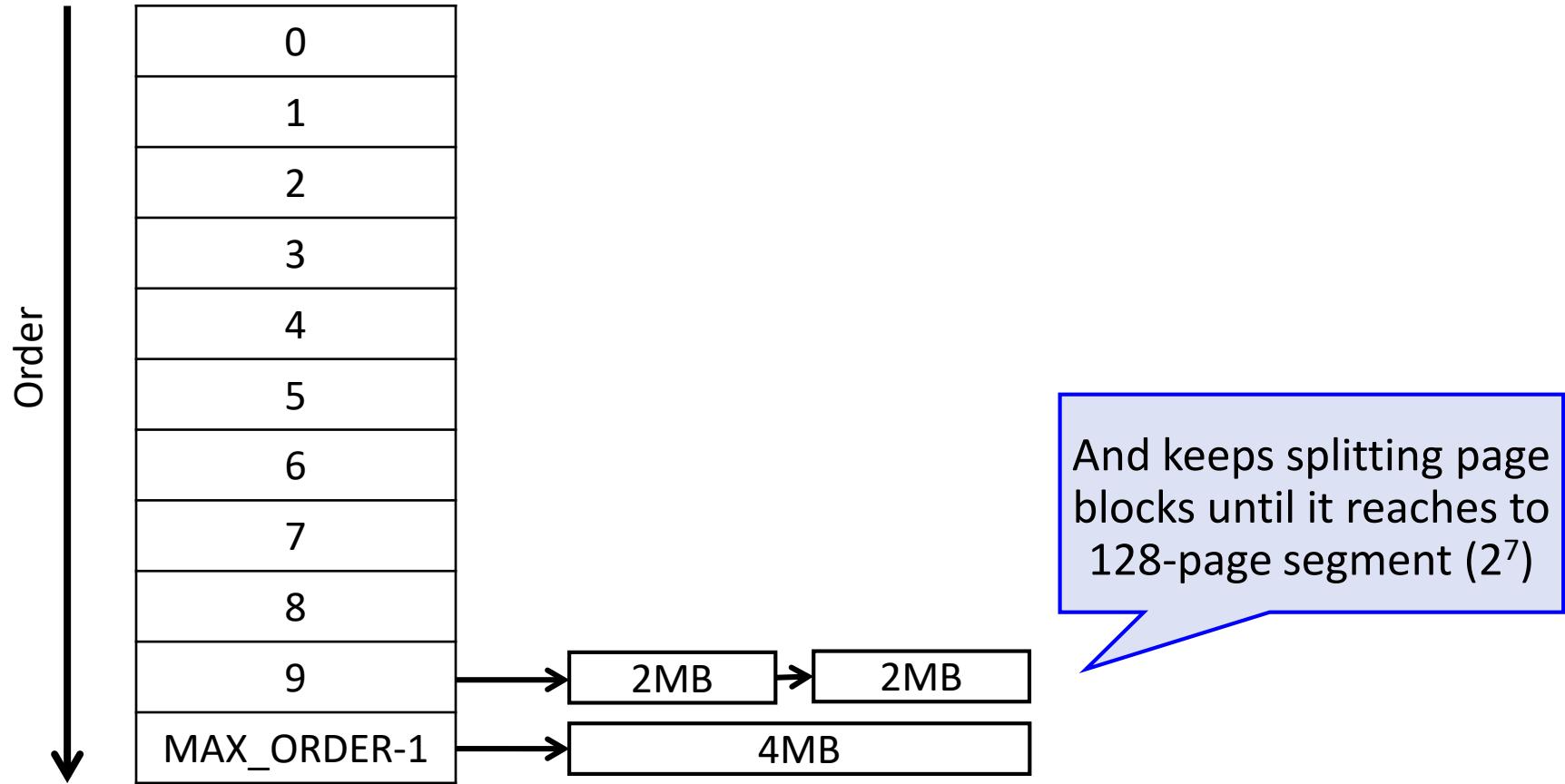
```
struct free_area
```





Buddy Allocator Implementation

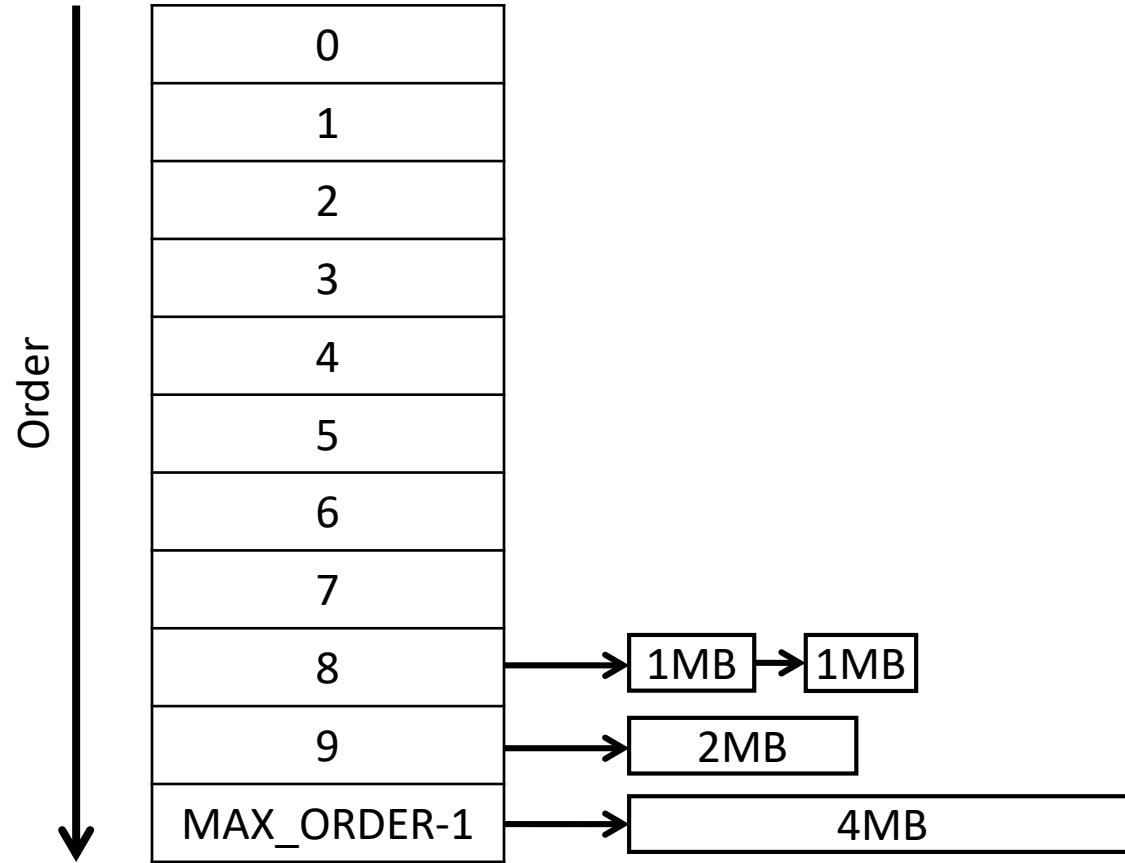
struct free_area





Buddy Allocator Implementation

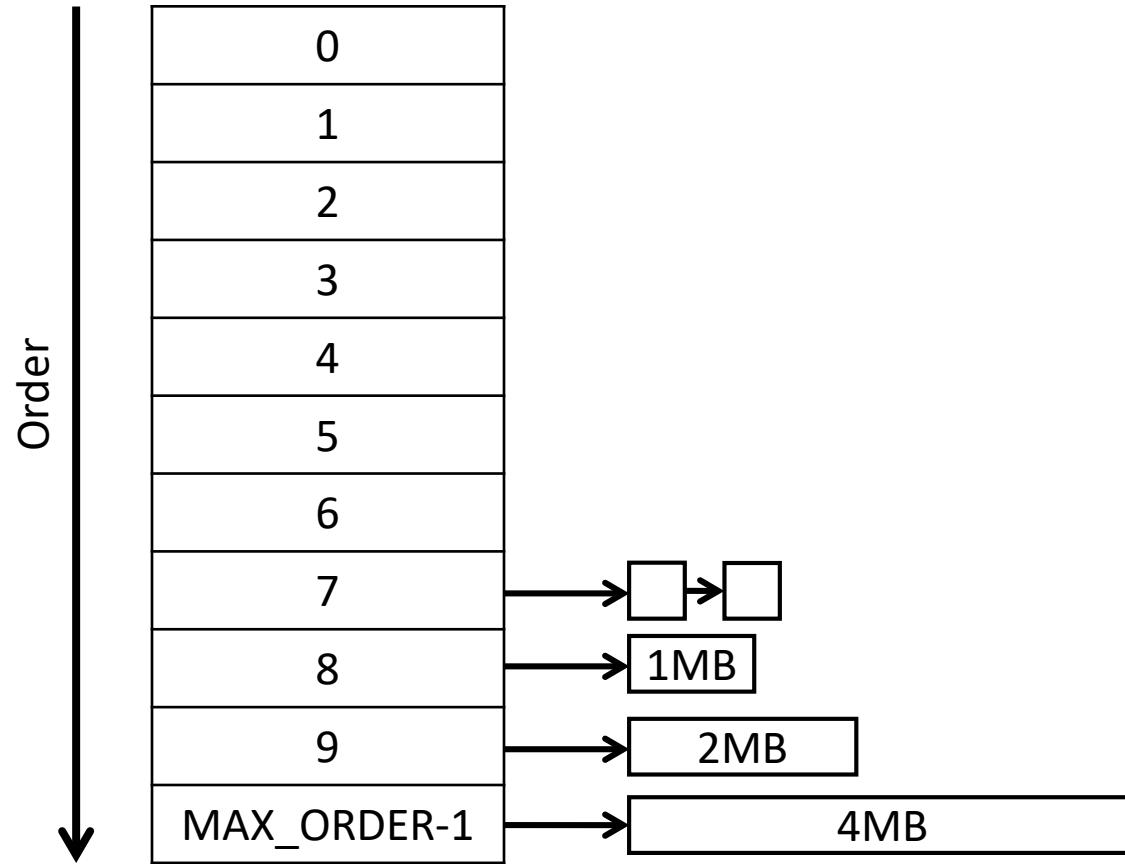
struct free_area





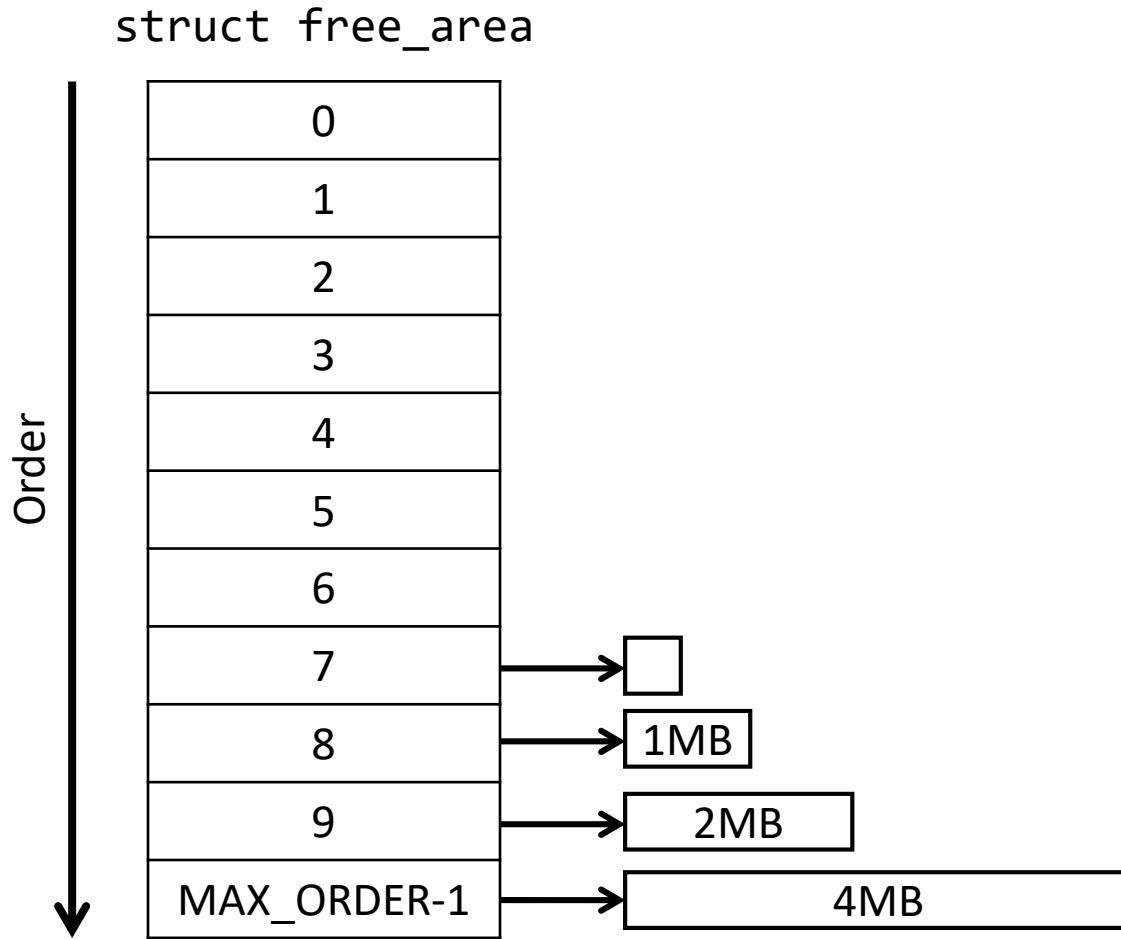
Buddy Allocator Implementation

struct free_area





Buddy Allocator Implementation

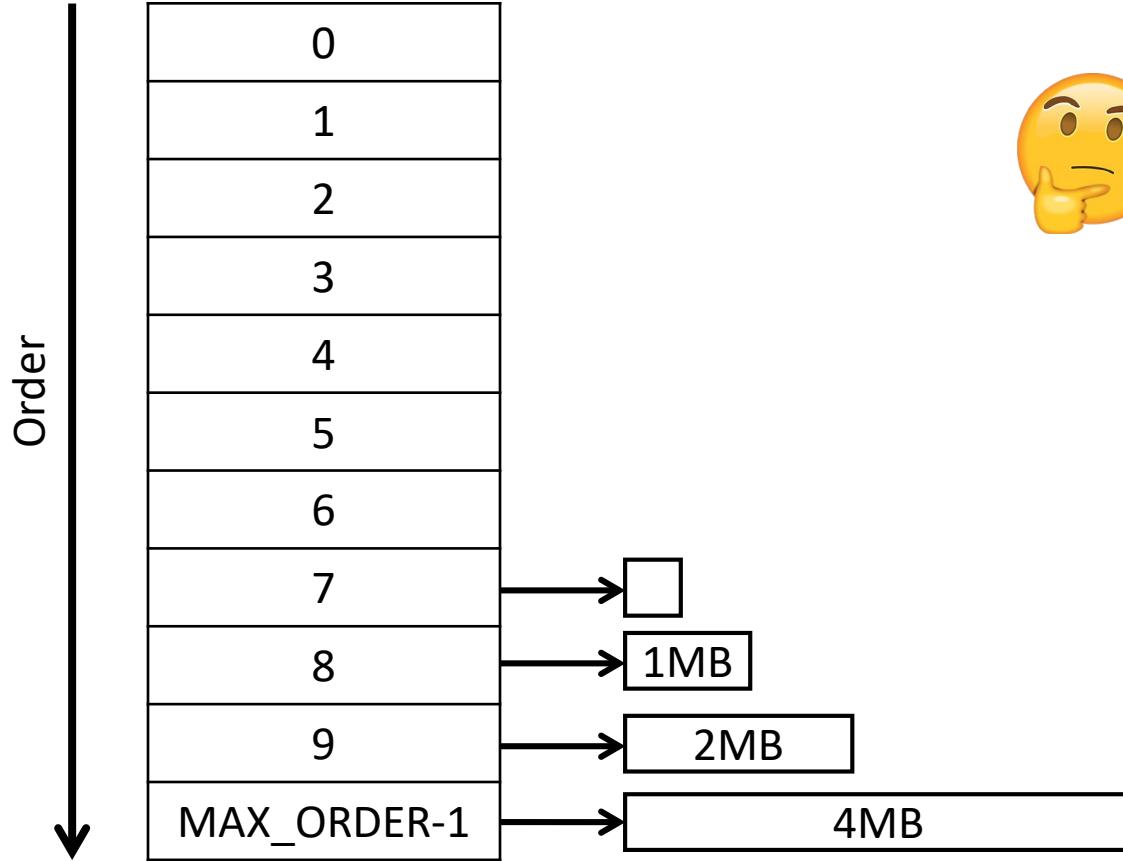


Allocate one of 128-page segment (2^7) for 100 contiguous page



Buddy Allocator Implementation

struct free_area

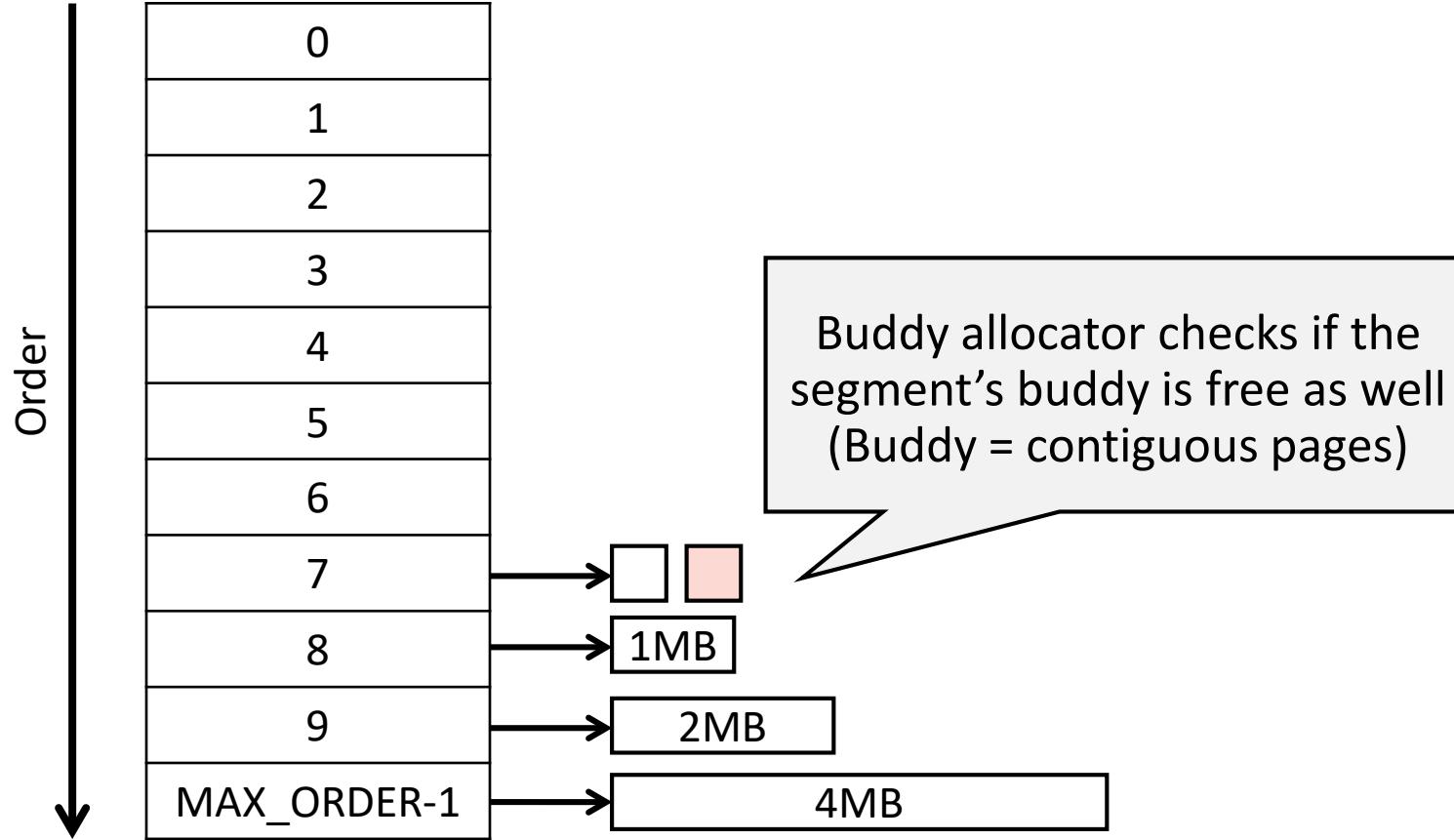


What if a 128-page segment is freed?



Buddy Allocator Implementation

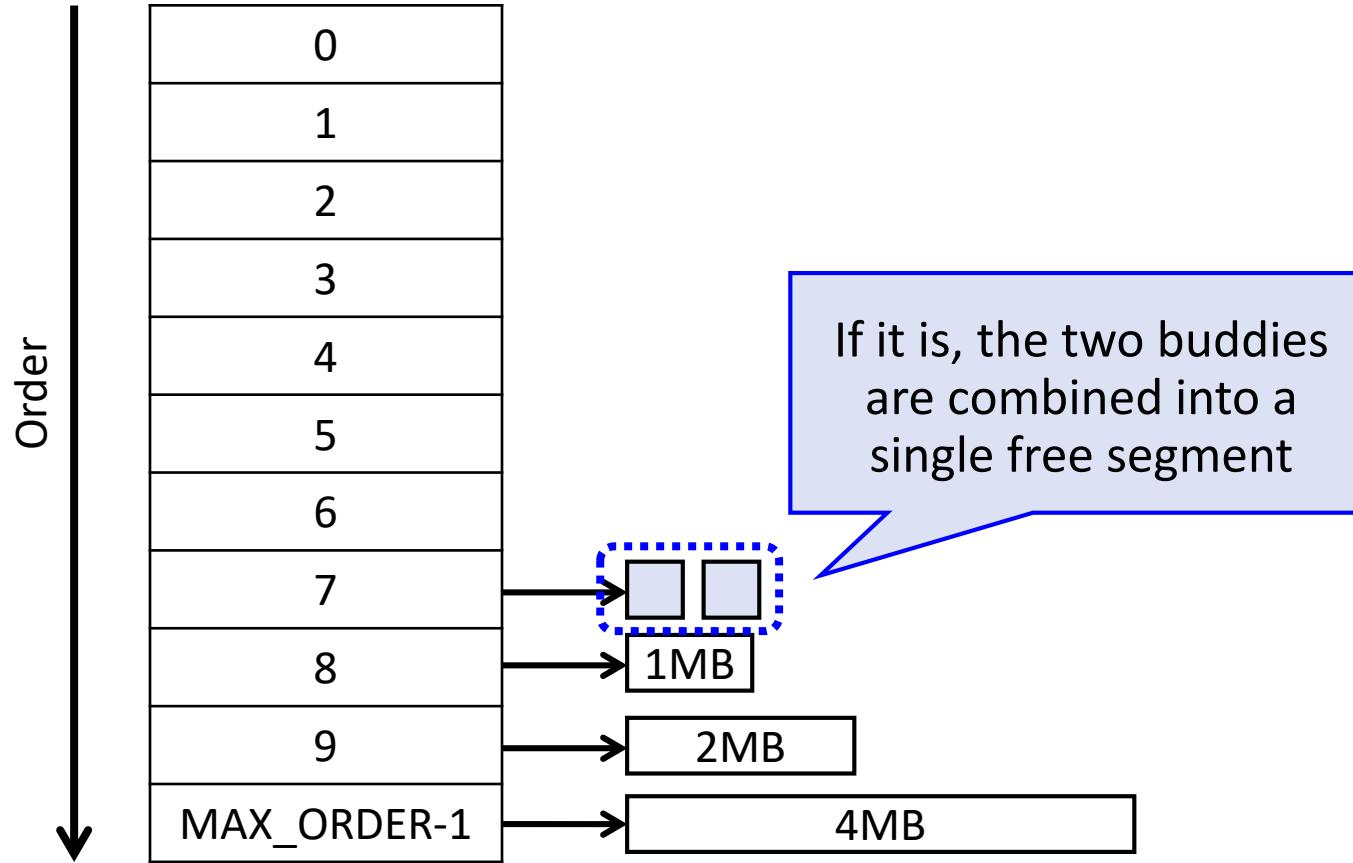
```
struct free_area
```





Buddy Allocator Implementation

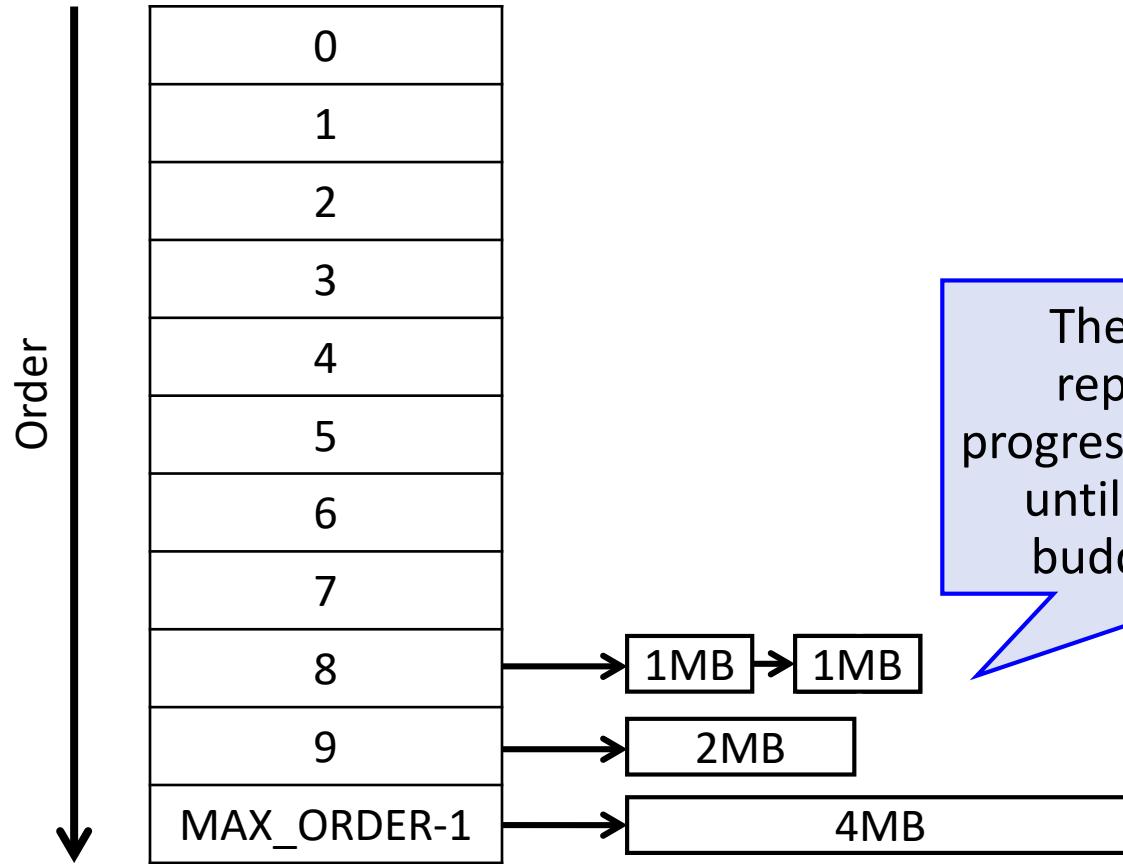
```
struct free_area
```





Buddy Allocator Implementation

```
struct free_area
```

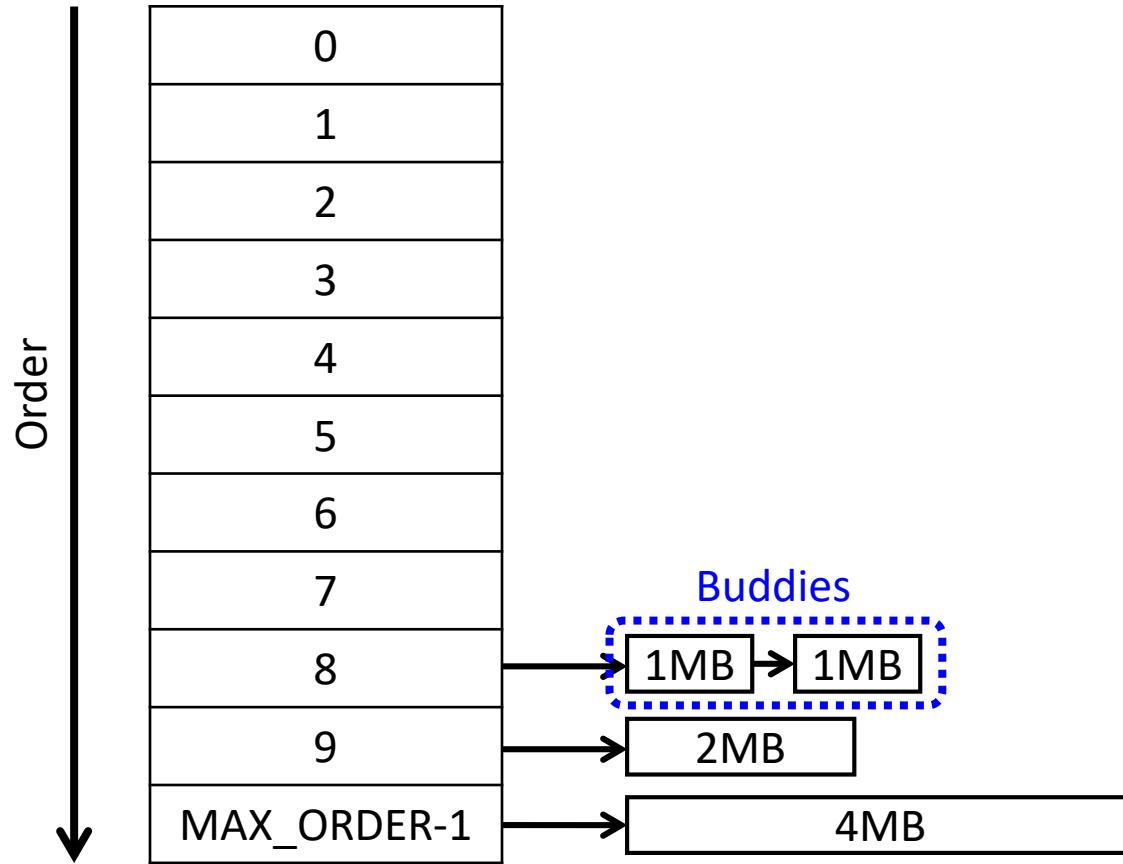


The merge process is repeated to create a progressively bigger segment until there are no more buddies to consolidate



Buddy Allocator Implementation

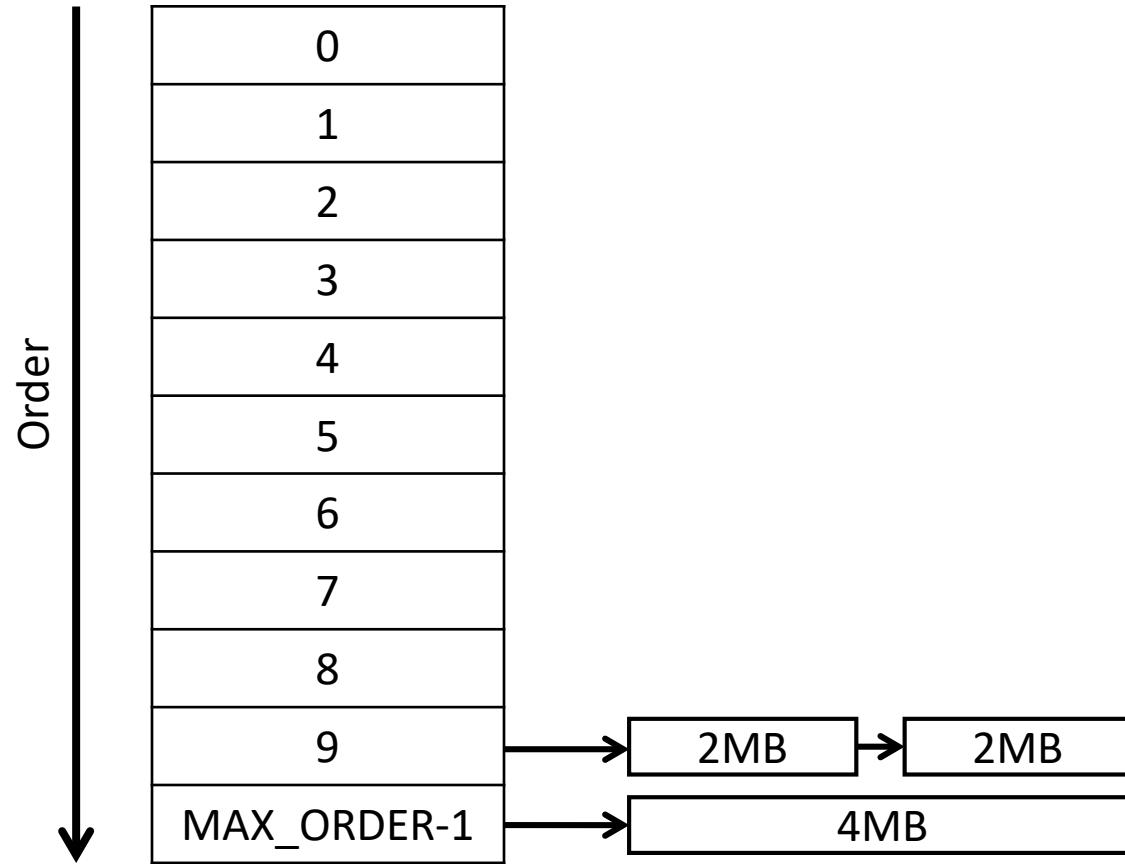
struct free_area





Buddy Allocator Implementation

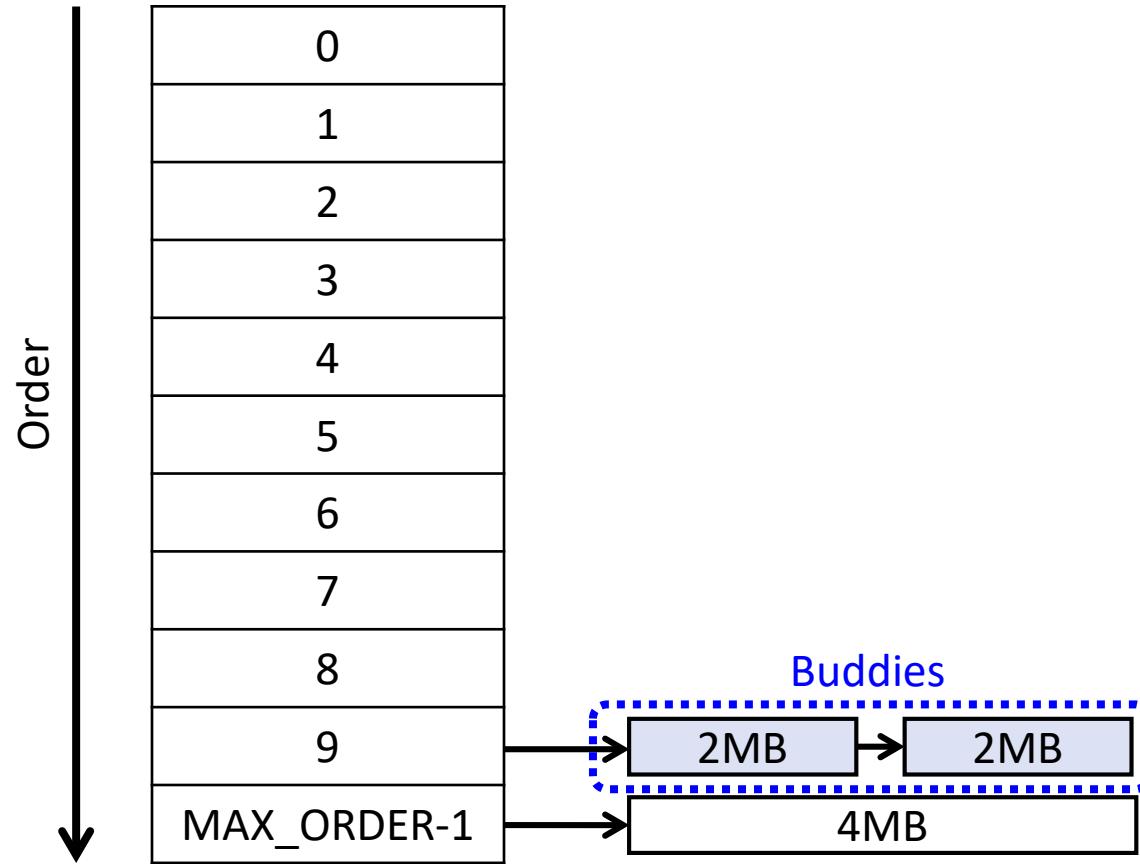
struct free_area





Buddy Allocator Implementation

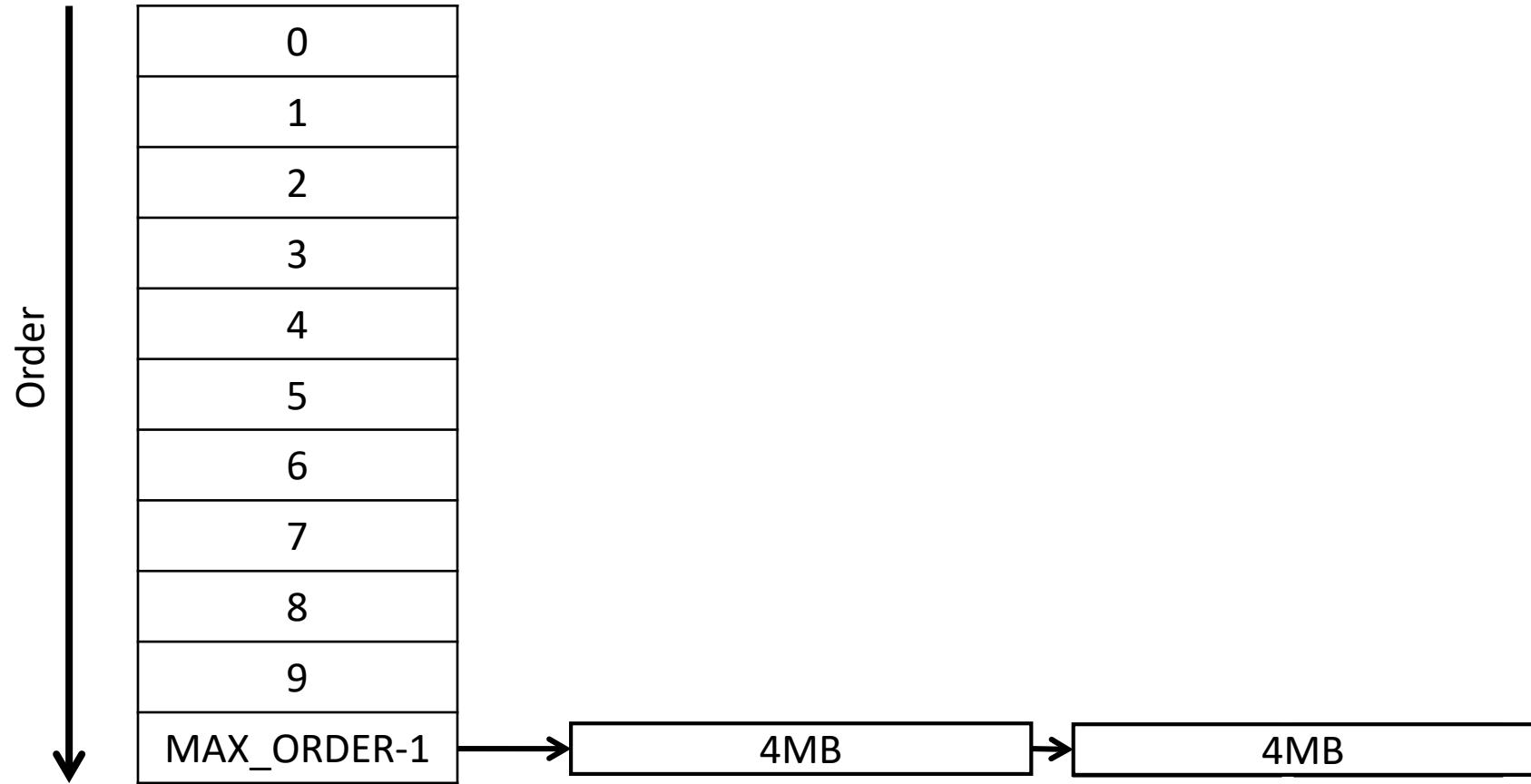
struct free_area





Buddy Allocator Implementation

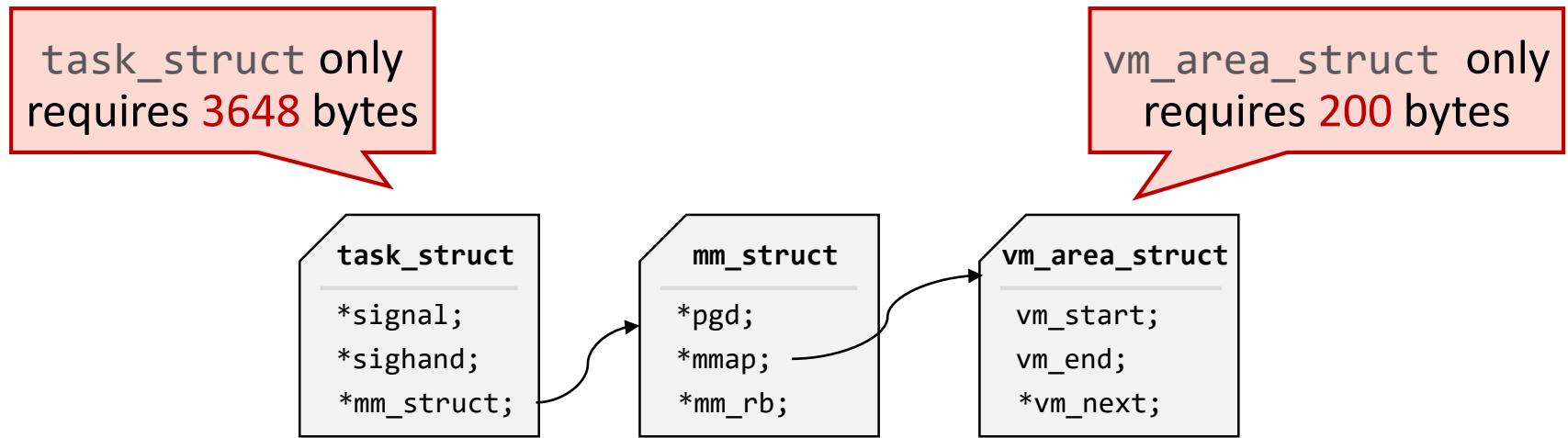
struct free_area





Pros and Cons of Buddy Allocator

- Buddy algorithm is **fast** and makes merging (of contiguous pages) very easy.
- Many memory allocations are practically small, **not entire pages** or groups of pages





Pros and Cons of Buddy Allocator

- Buddy algorithm is fast (allocating contiguous pages) very efficiently
- But, a lot of memory allocated as entire pages or groups of pages

Okay, we use page-based virtual memory system. But, do we really need to allocate 4KB-sized page for each `task_struct` and `vm_area_struct`?



`task_struct` only requires **3648** bytes

`task_struct`

```
*signal;  
*sighand;  
*mm_struct;
```

`mm_struct`

```
*pgd;  
*mmap;  
*mm_rb;
```

`vm_area_struct`

```
vm_start;  
vm_end;  
*vm_next;
```

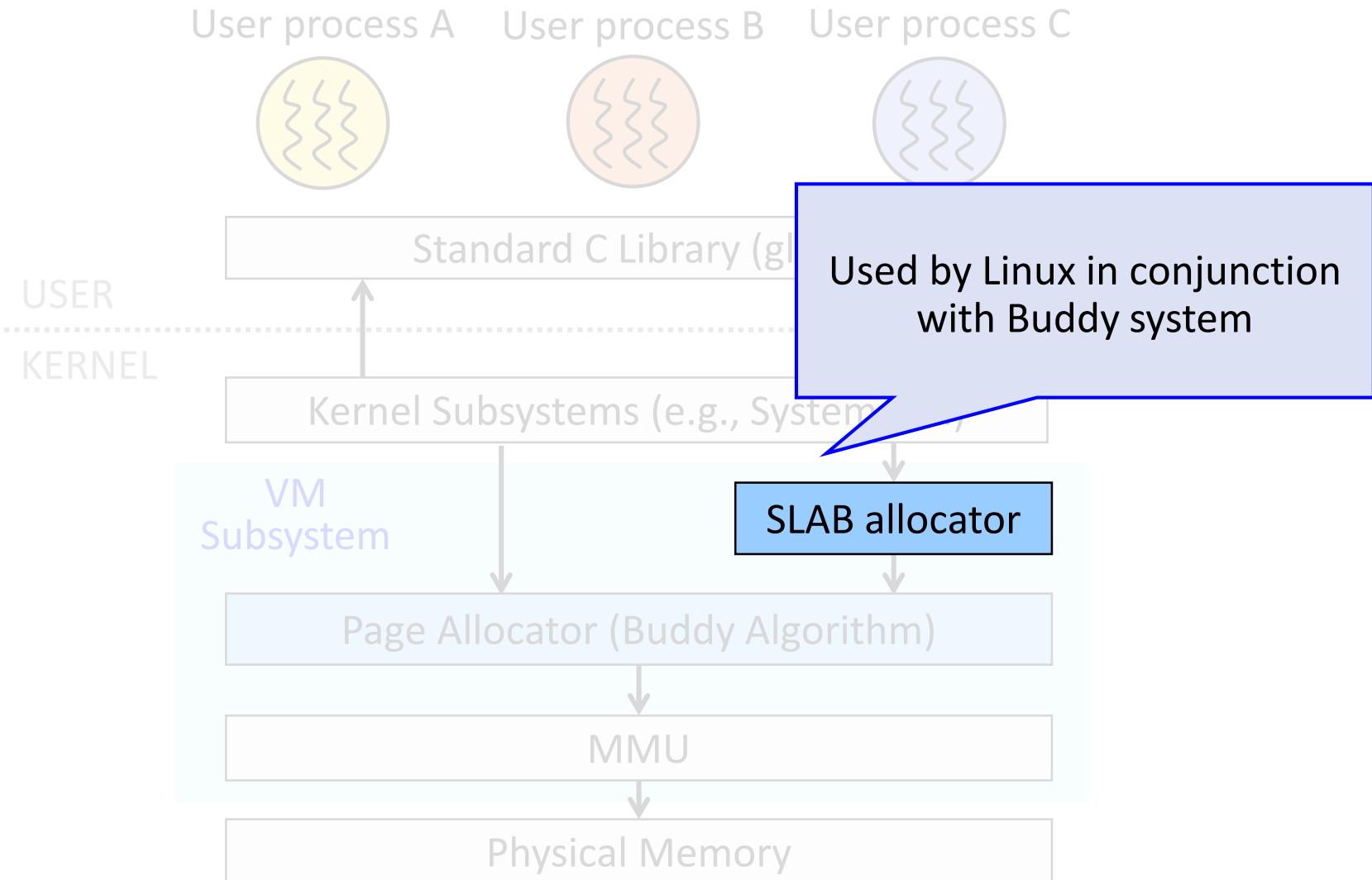
`vm_area_struct` only requires **200** bytes



Need to Accelerate

- Many allocations occur repeatedly
 - User program request a new node in linked list
 - OS requests a new process descriptor
- The size of memory allocations practically much smaller than a page frame
- Instead of search free list for new memory, keep a cache of recently deallocated memory
 - Call these allocation, deallocation units, objects
- On a new request, see if there is memory in the cache to allocate

Slab Allocator



Goal of Slab Allocator

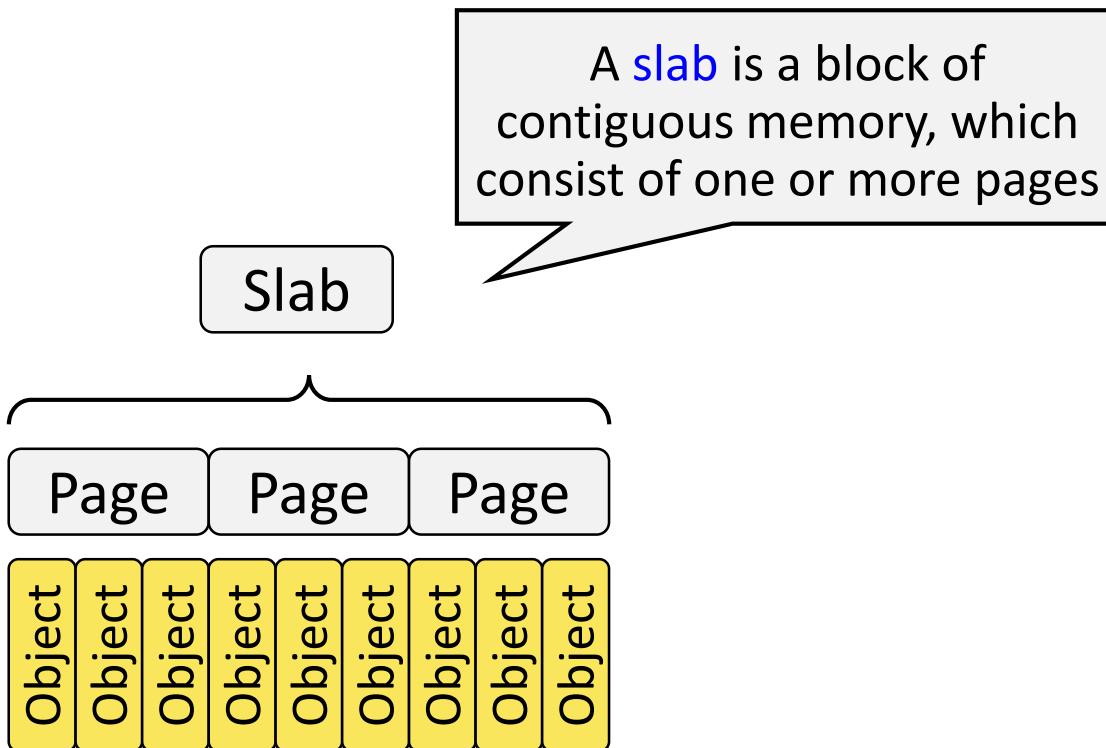
- With memory paging, the granularity is page (4KB)



But, allocates only part of page for
small sized kernel objects which are
frequently created/destroyed.

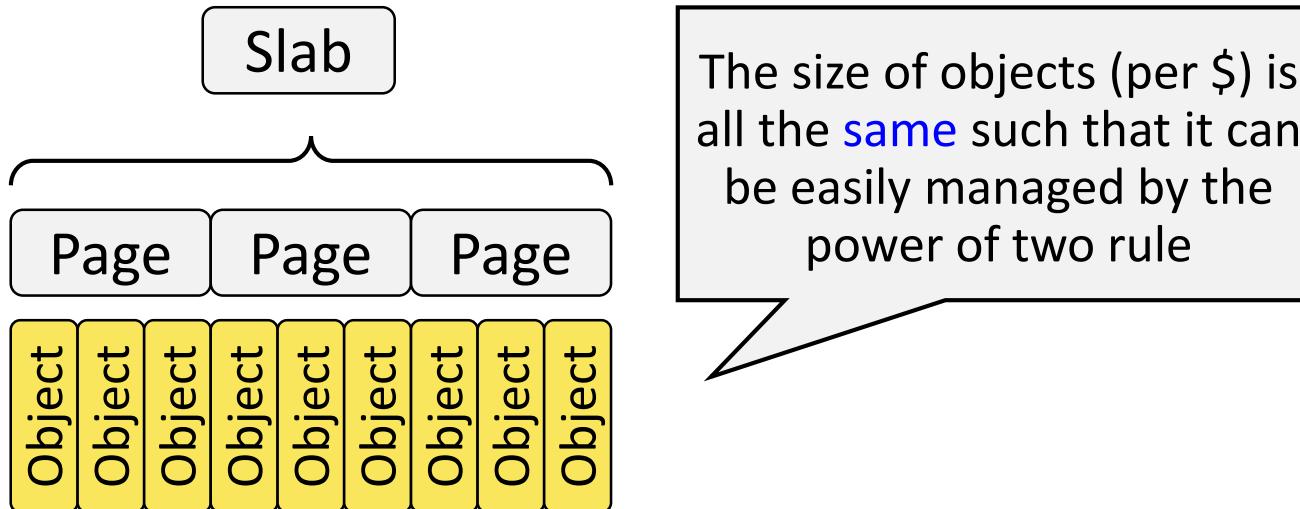


Slab Allocator Implementation



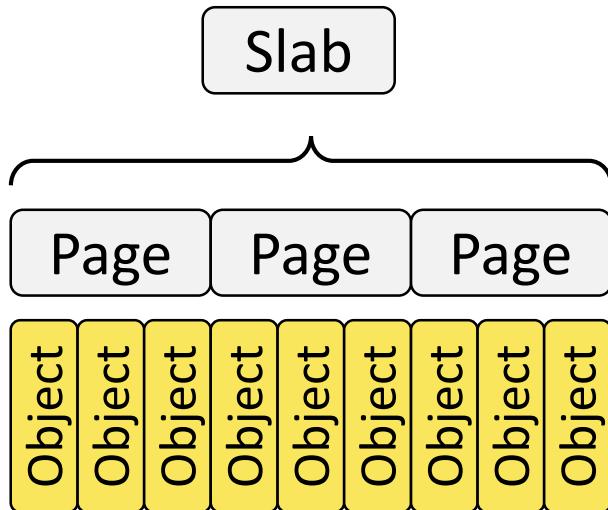


Slab Allocator Implementation



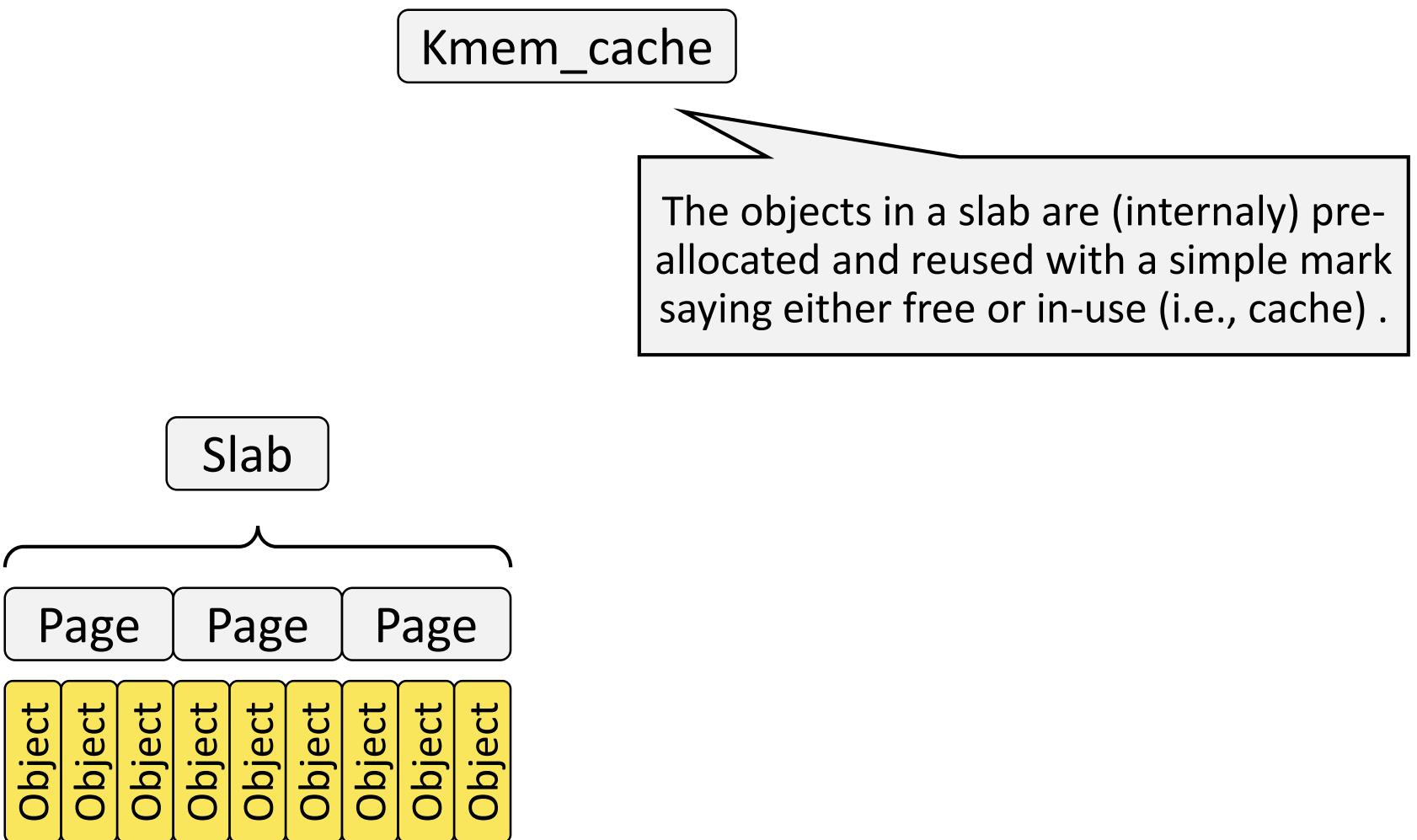


Slab Allocator Implementation

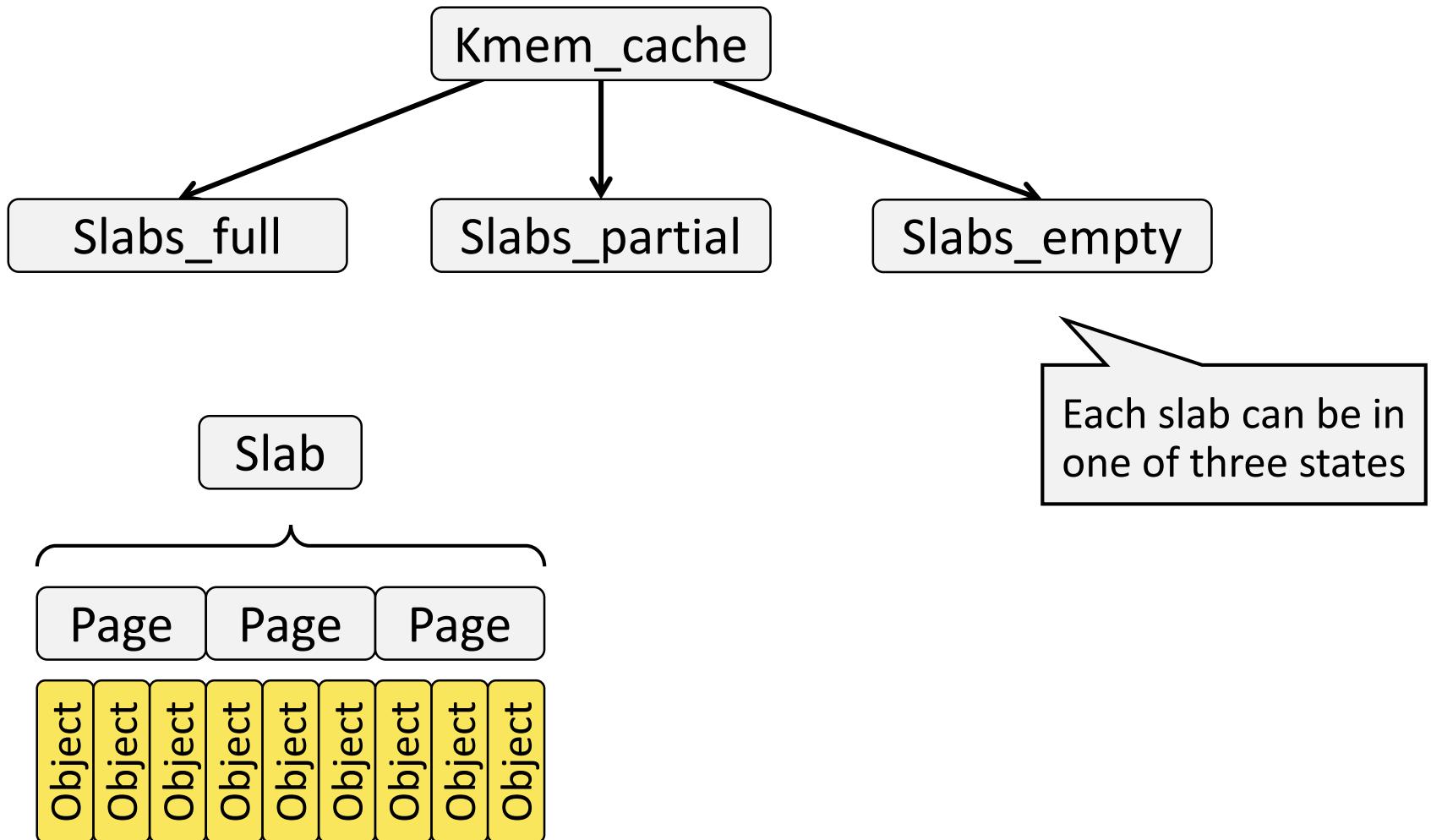


For example, there are **20** `vm_area_struct` (200 bytes) within a single page

Slab Allocator Implementation

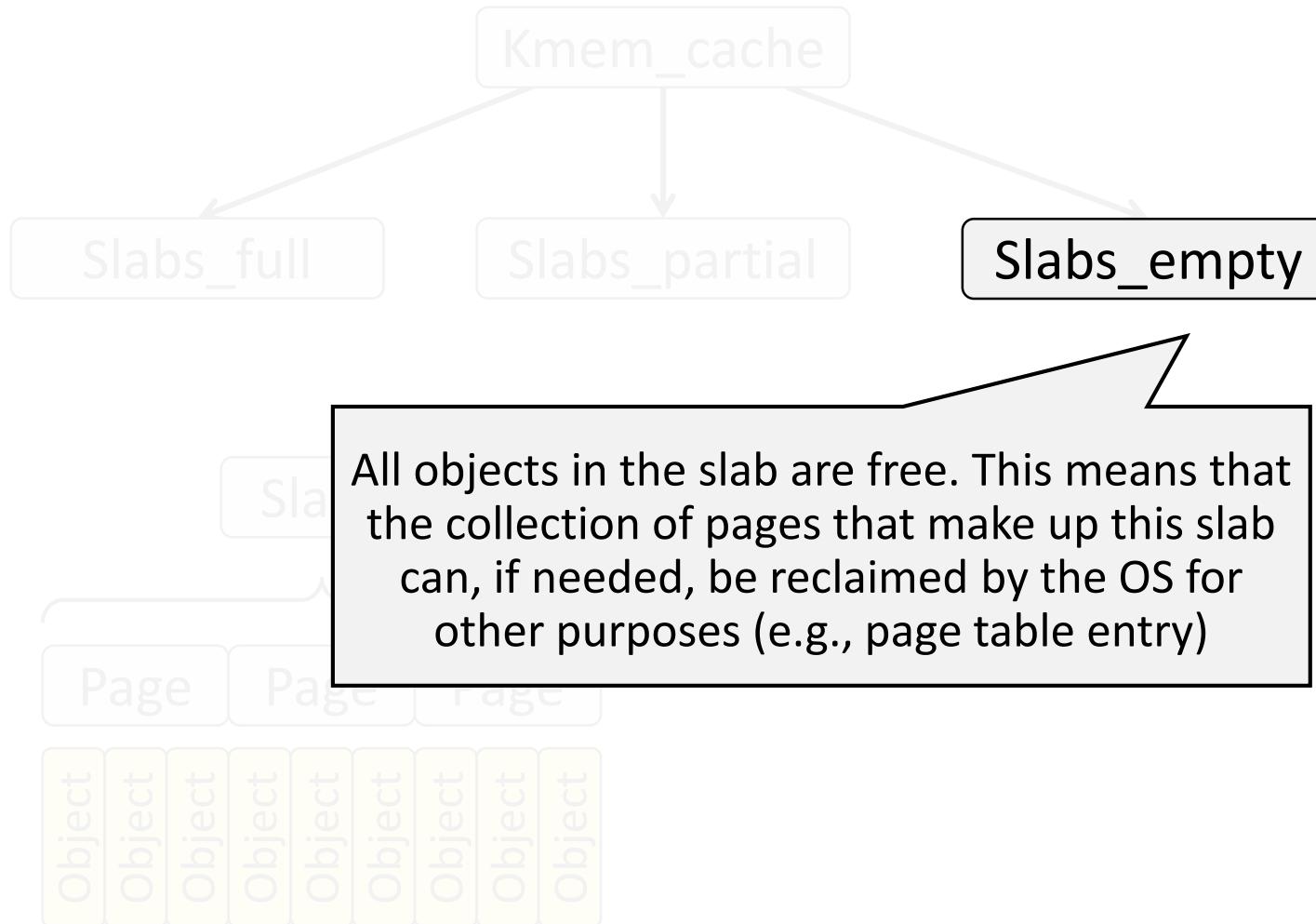


Slab Allocator Implementation



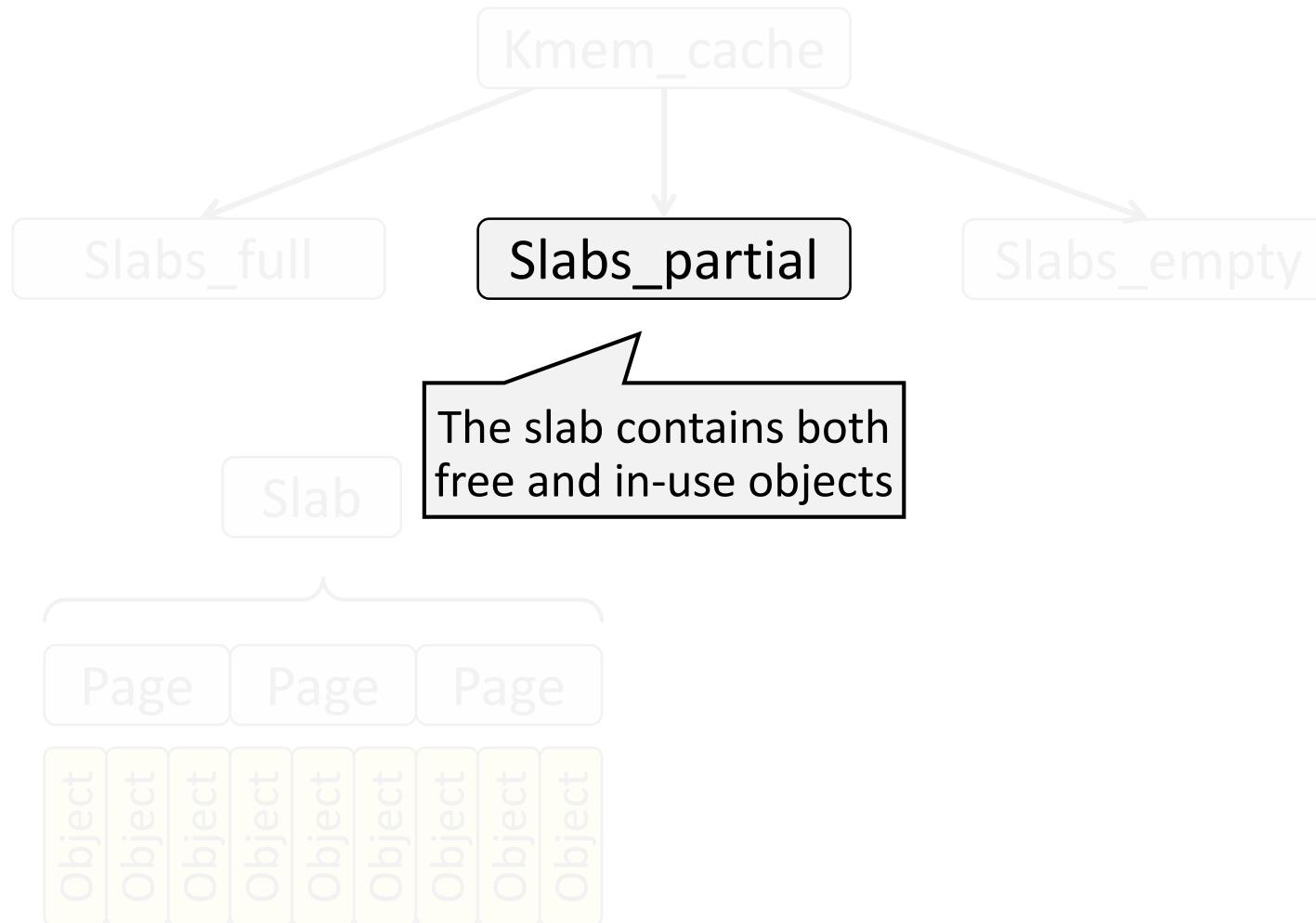


Slab Allocator Implementation



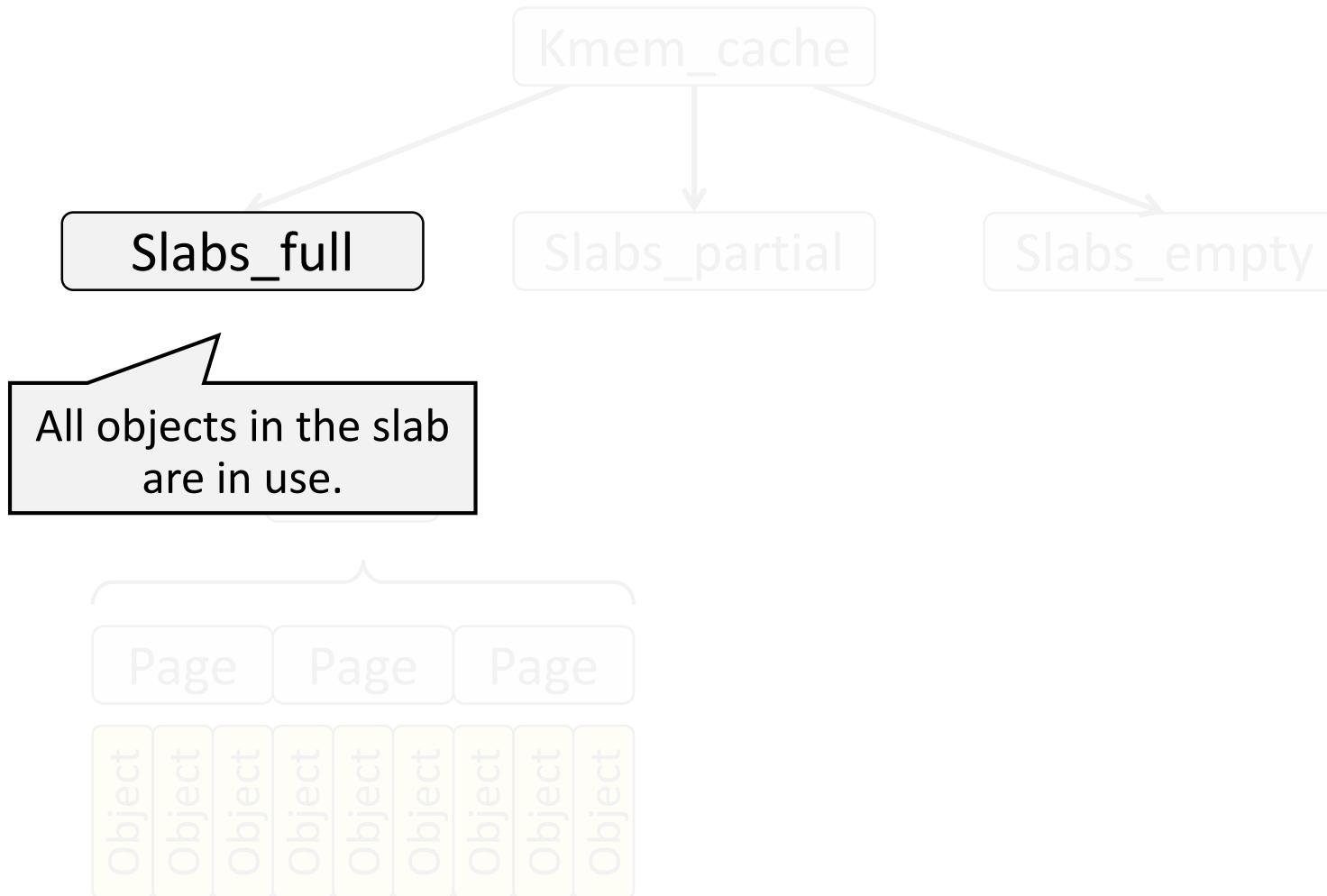


Slab Allocator Implementation

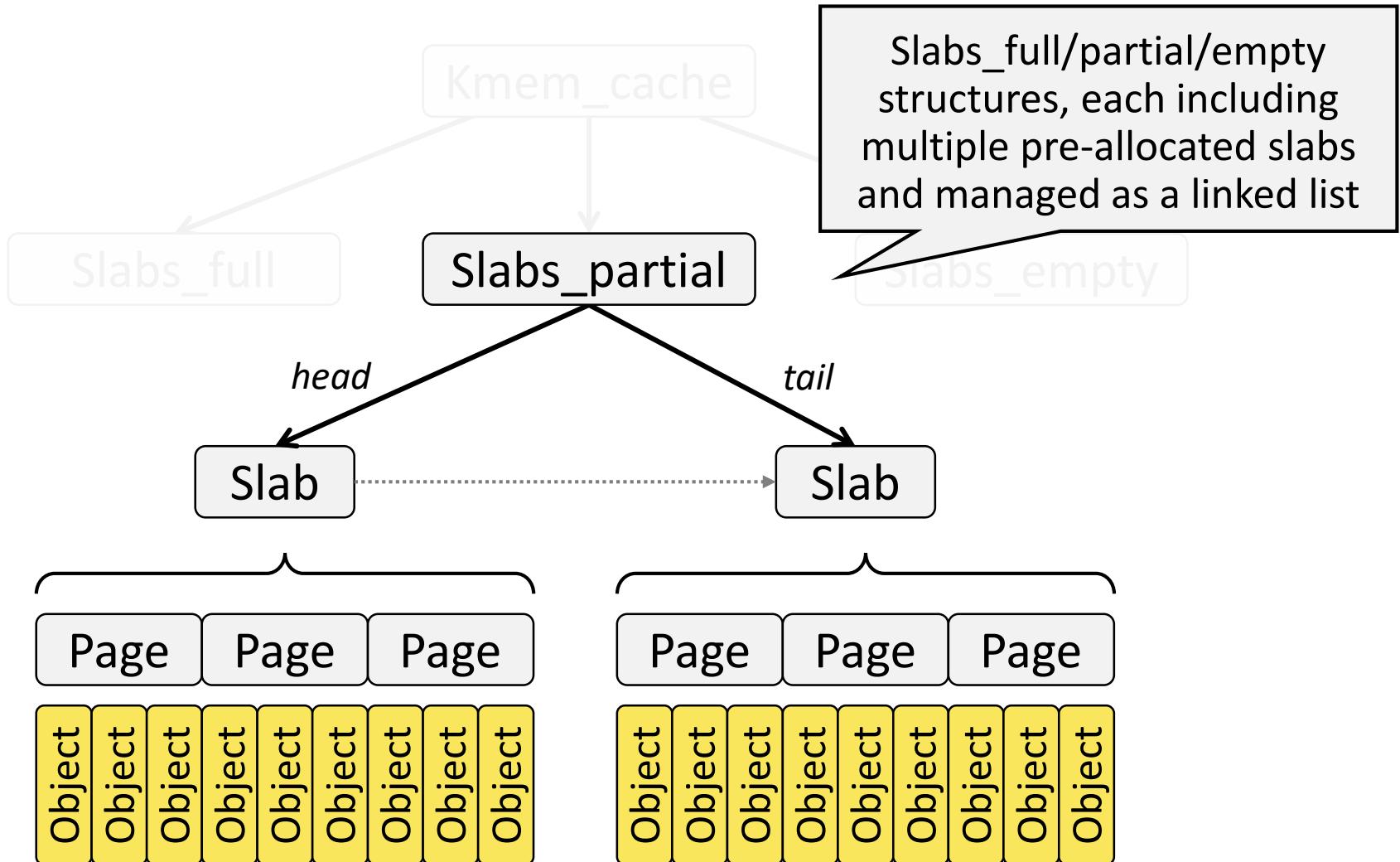




Slab Allocator Implementation

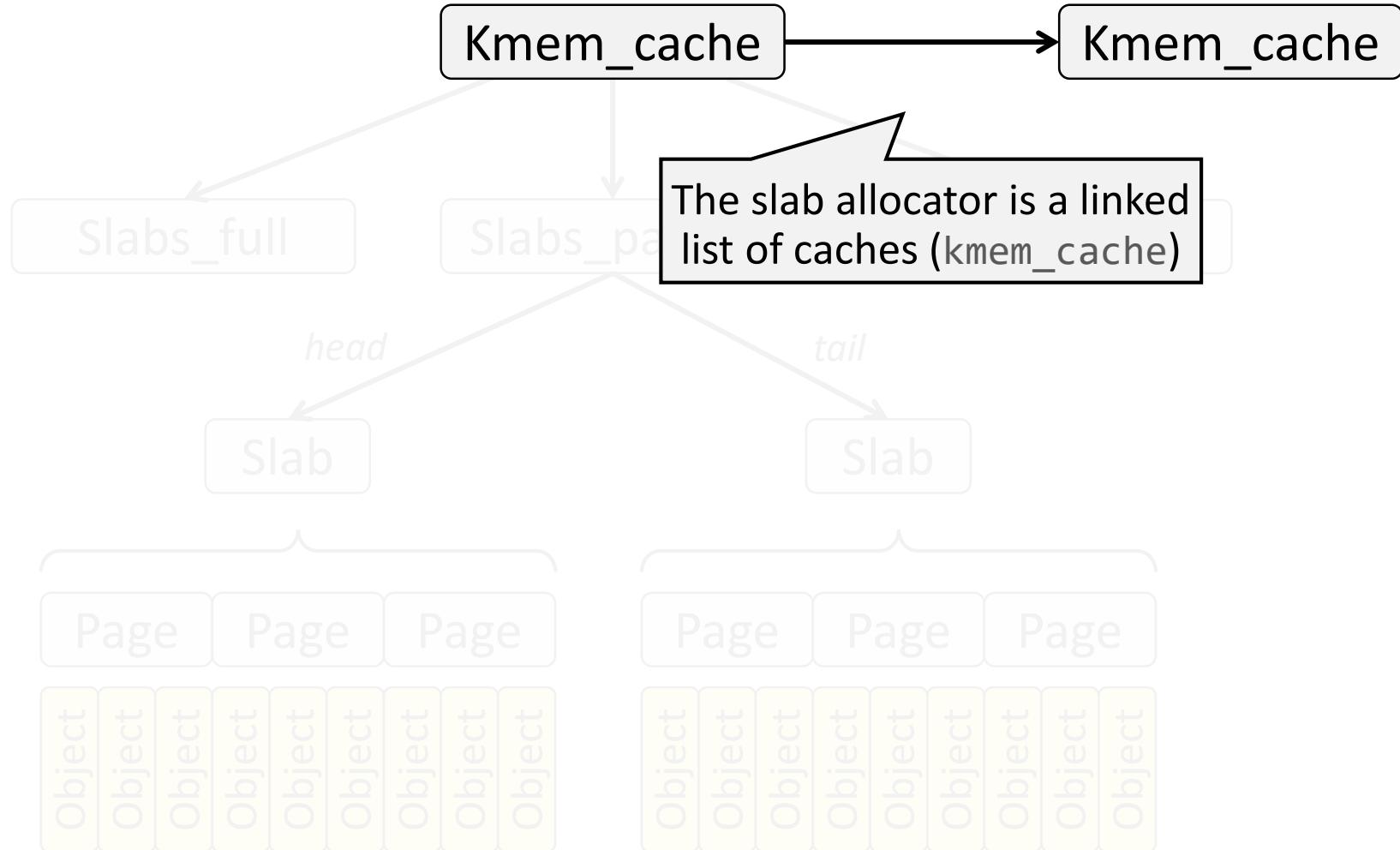


Slab Allocator Implementation

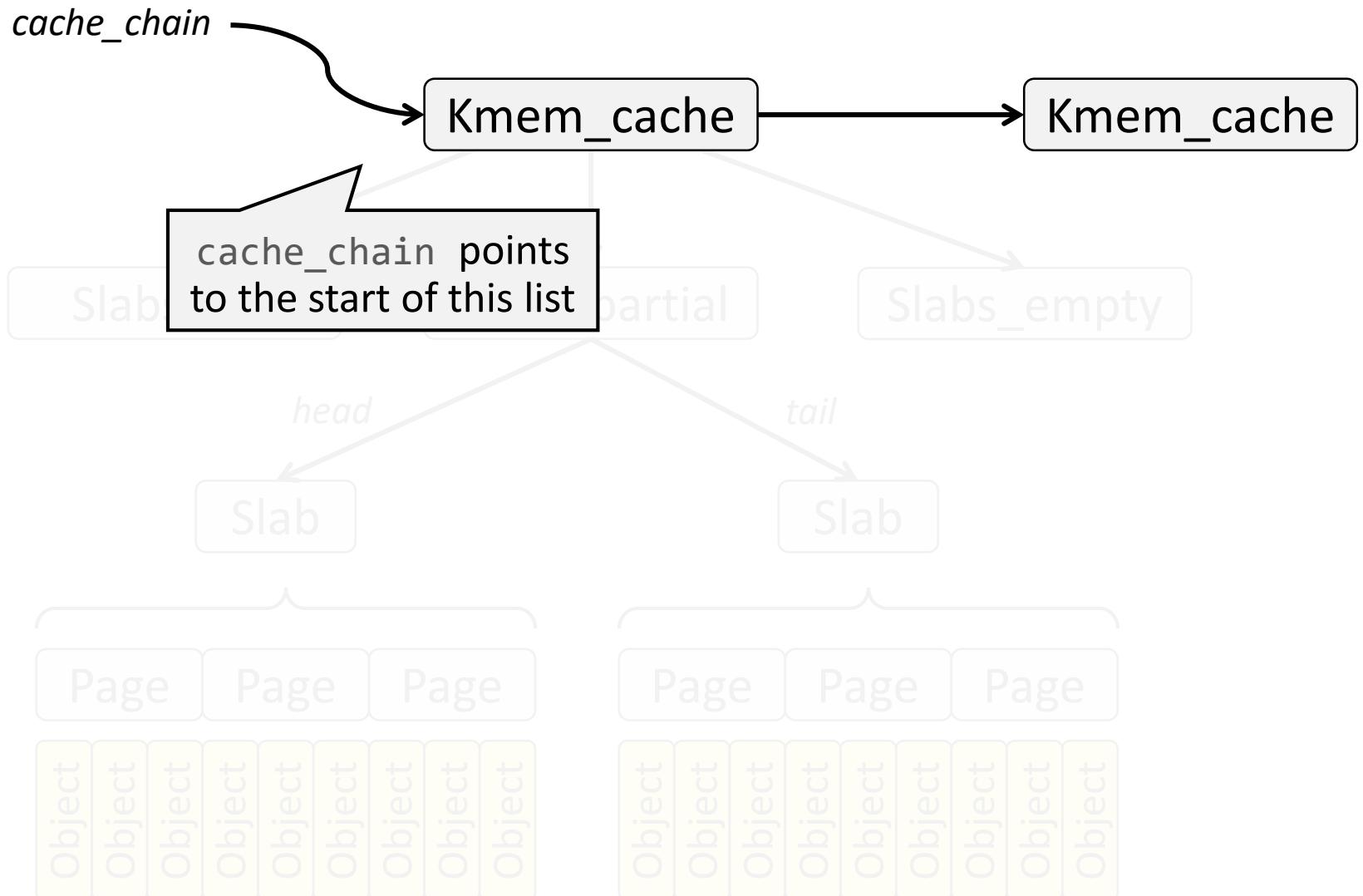




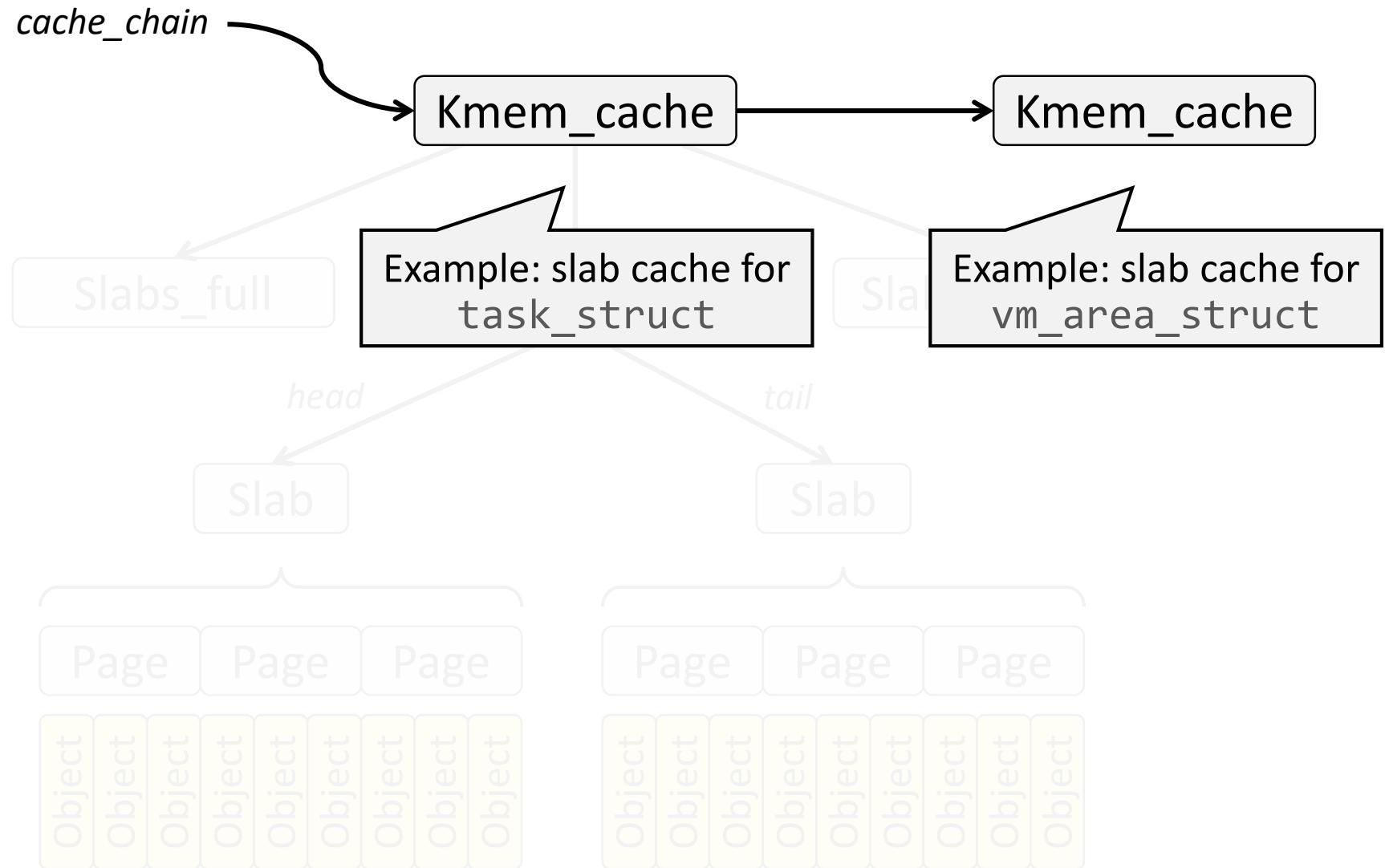
Slab Allocator Implementation



Slab Allocator Implementation



Slab Allocator Implementation





Example of Slab Cache

```
$ cat /proc/slabinfo
```

slabinfo prints the statistics
of kernel slab allocator

Example of Slab Cache

Fields mean
<active objs> <num objs> <objsize> ...

```
$ cat /proc/slabinfo
```

sigqueue	1224	1224	160	51	2 : tunables	0	0	0 : slabdata	24	24	0
vm_area_struct	51207	62960	200	40	2 : tunables	0	0	0 : slabdata	1574	1574	0
task_struct	6193	6624	3648	8	8 : tunables	0	0	0 : slabdata	828	828	0
ip6-frgs	34	34	240	34	2 : tunables	0	0	0 : slabdata	1	1	0
UDPLITEv6	0	0	1088	30	8 : tunables	0	0	0 : slabdata	0	0	0
UDPV6	720	720	1088	30	8 : tunables	0	0	0 : slabdata	24	24	0
tw_sock_TCPv6	551	551	280	29	2 : tunables	0	0	0 : slabdata	19	19	0
request_sock_TCPv6	833	833	328	49	4 : tunables	0	0	0 : slabdata	17	17	0
TCPv6	525	525	2112	15	8 : tunables	0	0	0 : slabdata	35	35	0
kcopyd_job	0	0	3312	9	8 : tunables	0	0	0 : slabdata	0	0	0
dm_uevent	0	0	2632	12	8 : tunables	0	0	0 : slabdata	0	0	0
cfq_queue	0	0	232	35	2 : tunables	0	0	0 : slabdata	0	0	0
bsg_cmd	0	0	312	52	4 : tunables	0	0	0 : slabdata	0	0	0
mqueue_inode_cache	756	756	896	36	8 : tunables	0	0	0 : slabdata	21	21	0
fuse_request	960	960	400	40	4 : tunables	0	0	0 : slabdata	24	24	0
fuse_inode	25805	25805	768	42	8 : tunables	0	0	0 : slabdata	647	647	0

...

Example of Slab Cache

```
$ cat /proc/slabinfo
sigqueue          1224 1224 160 51 2 : tunables 0 0 0 : slabdata 24 24 0
vm_area_struct   51207 62960 200 40 2 : tunables 0 0 0 : slabdata 1574 1574 0
task_struct      6193 6624 3648 8 8 : tunables 0 0 0 : slabdata 828 828 0
ip6-frags        34 34 240 34 2 : tunables 0 0 0 : slabdata 1 1 0
UDPLITEv6        0 0 1088 0 0 0 : tunables 0 0 0 : slabdata 0 0 0
UDPV6            720 720 1088 0 0 0 : tunables 0 0 0 : slabdata 24 24 0
tw_sock_TCPv6    551 551 551 0 0 0 : tunables 0 0 0 : slabdata 19 19 0
request_sock_TCPv6 833 833 833 0 0 0 : tunables 0 0 0 : slabdata 17 17 0
TCPv6            525 525 2112 0 0 0 : tunables 0 0 0 : slabdata 35 35 0
kcopyd_job       0 0 3312 0 0 0 : tunables 0 0 0 : slabdata 0 0 0
dm_uevent        0 0 2632 12 8 : tunables 0 0 0 : slabdata 0 0 0
cfq_queue        0 0 232 35 2 : tunables 0 0 0 : slabdata 0 0 0
bsg_cmd           0 0 312 52 4 : tunables 0 0 0 : slabdata 0 0 0
mqueue_inode_cache 756 756 896 36 8 : tunables 0 0 0 : slabdata 21 21 0
fuse_request     960 960 400 40 4 : tunables 0 0 0 : slabdata 24 24 0
fuse_inode       25805 25805 768 42 8 : tunables 0 0 0 : slabdata 647 647 0
...

```

<vm_area_struct>
Object size is only 200 bytes,
and there are 62960 number
of pre-allocated objects and
51207 are in use.

Example of Slab Cache

```
$ cat /proc/slabinfo
sigqueue          1224 1224 160 51 2 : tunables 0 0 0 : slabdata 24 24 0
vm area struct   51207 62960 200 40 2 : tunables 0 0 0 : slabdata 1574 1574 0
task_struct      6193 6624 3648 8 8 : tunables 0 0 0 : slabdata 828 828 0
ip6-frags        34 34 240 34 2 : tunables 0 0 0 : slabdata 1 1 0
UDPLITEv6        0 0 1088 30 2 : tunables 0 0 0 : slabdata 0 0 0
UDPV6            720 720 1088 24 24 0
tw_sock_TCPv6    551 551           data 19 19 0
request_sock_TCPv6 833 833         abdata 17 17 0
TCPv6            525 525 2112 35 35 0
kcopyd_job       0 0 3312 0 0 0
dm_uevent        0 0 2632 0 0 0
cfq_queue        0 0 232 35 2 : tunables 0 0 0 : slabdata 0 0 0
bsg_cmd           0 0 312 52 4 : tunables 0 0 0 : slabdata 0 0 0
mqueue_inode_cache 756 756 896 36 8 : tunables 0 0 0 : slabdata 21 21 0
fuse_request     960 960 400 40 4 : tunables 0 0 0 : slabdata 24 24 0
fuse_inode       25805 25805 768 42 8 : tunables 0 0 0 : slabdata 647 647 0
...

```

<task_struct>
Object size is only 3648 bytes,
and there are 6624 number
of pre-allocated objects and
6193 are in use.

Example of Slab Cache

```
$ cat /proc/slabinfo
```

sigqueue	1224	1224	160	51	2 : tunables	0	0	0 : slabdata	24	24	0
vm_area_struct	51207	62960	200	40	2 : tunables	0	0	0 : slabdata	1574	1574	0
task_struct	6193	6624	3648	8	8 : tunables	0	0	0 : slabdata	828	828	0
ip6-frags	34	34	240	34	2 : tunables	0	0	0 : slabdata	1	1	0
UDPLITEv6	0	0						slabdata	0	0	0
UDPv6	720	720						: slabdata	24	24	0
tw_sock_TCPv6	551							0 : slabdata	19	19	0
request_sock_TCPv6	83							0 : slabdata	17	17	0
TCPv6	525	525						slabdata	35	35	0
kcopyd_job	0	0	331		3 : tunables	0	0	0 : slabdata	0	0	0
dm_uevent	0	0	2632	12	8 : tunables	0	0	0 : slabdata	0	0	0
cfq_queue	0	0	232	35	2 : tunables	0	0	0 : slabdata	0	0	0
bsg_cmd	0	0	312	52	4 : tunables	0	0	0 : slabdata	0	0	0
mqueue_inode_cache	756	756	896	36	8 : tunables	0	0	0 : slabdata	21	21	0
fuse_request	960	960	400	40	4 : tunables	0	0	0 : slabdata	24	24	0
fuse_inode	25805	25805	768	42	8 : tunables	0	0	0 : slabdata	647	647	0
...											

Not only for task_struct related object, there are many slab caches for network and I/O

What's Next?

- We've covered OS abstractions for CPU and memory so far.
- Let's check the I/O management which is another major component of OS.

21S EE415

IO & Disks

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



IO & Disks I

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab*



Input/Output (I/O) Devices



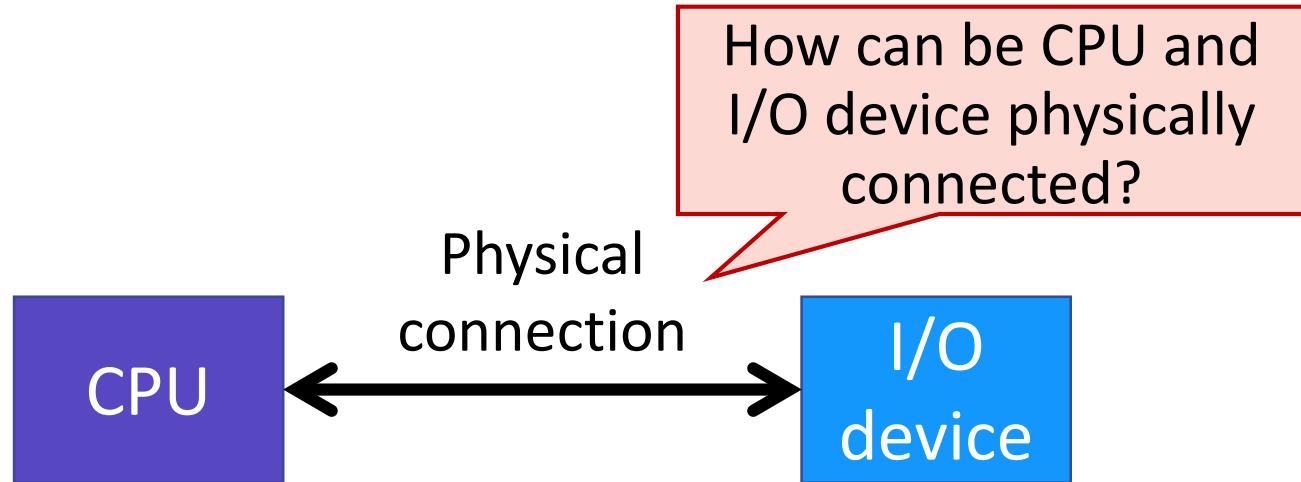
There are
many, many
I/O devices



...

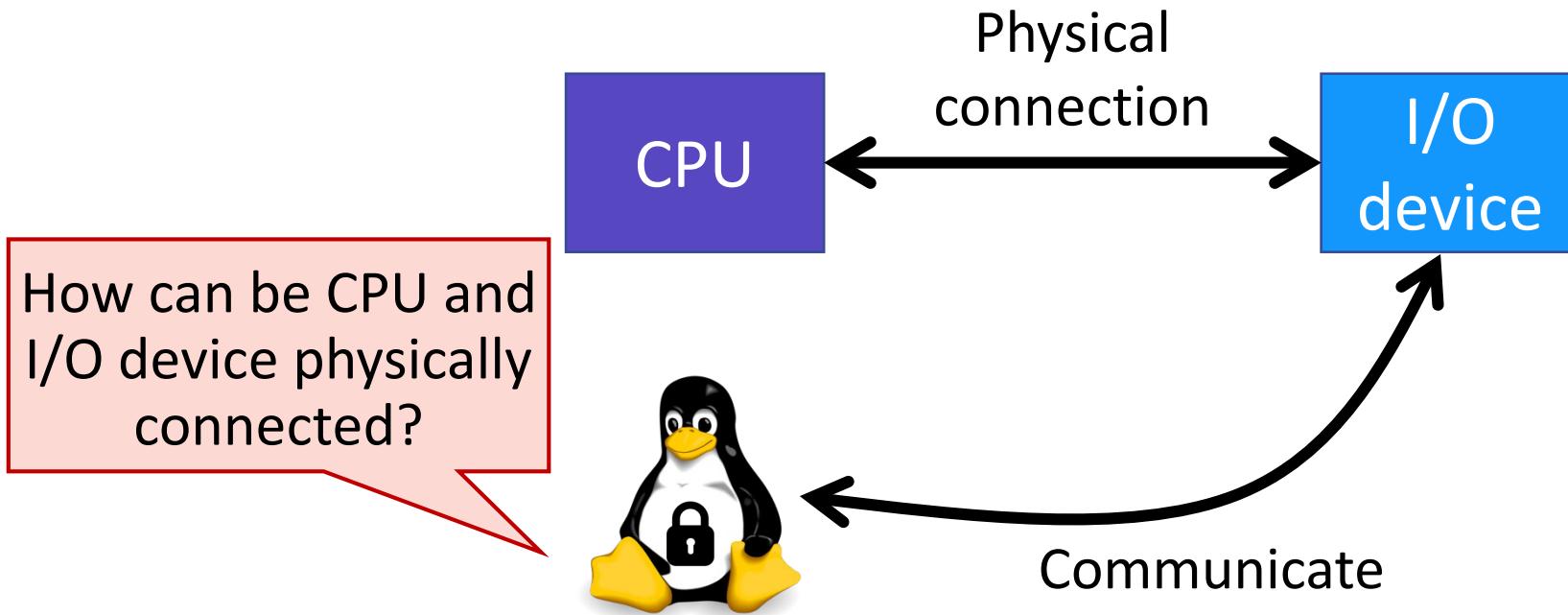
Today's Roadmap

- Today we will mainly answer two questions



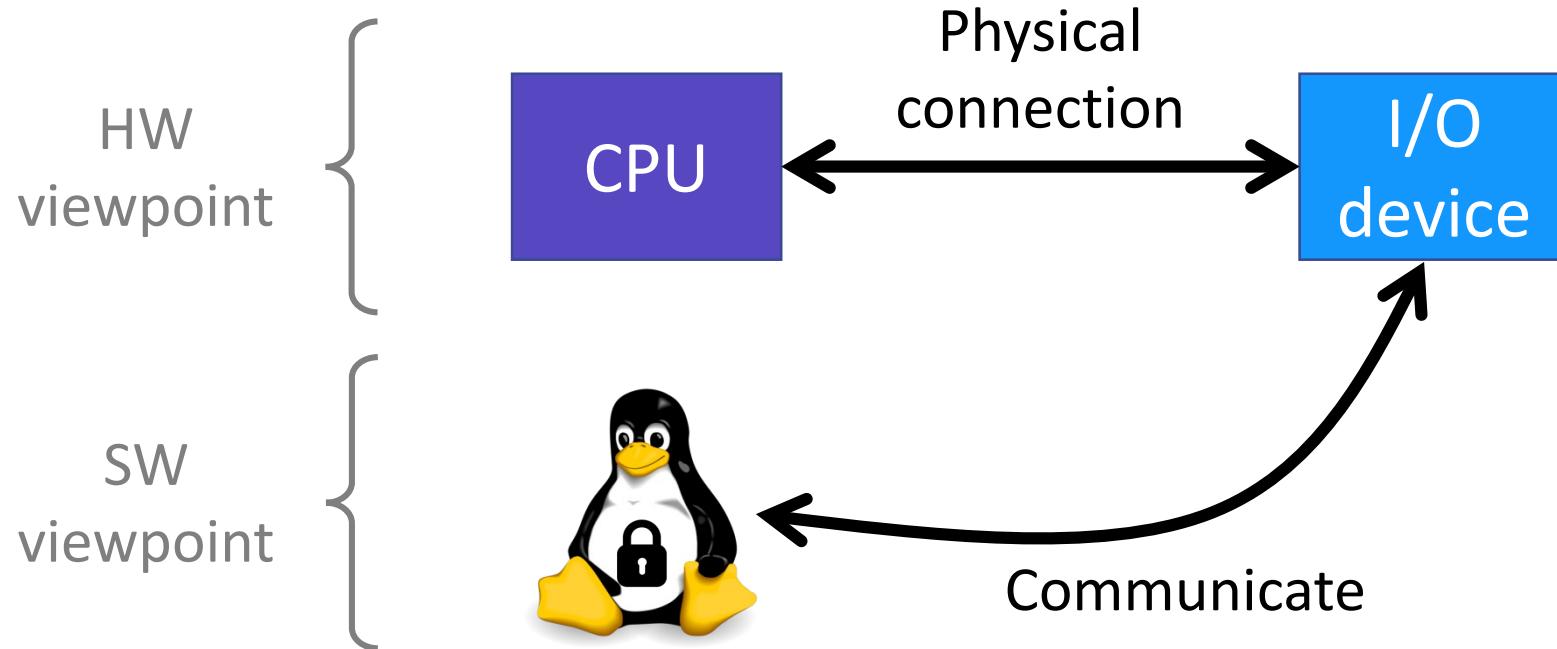
Today's Roadmap

- Today we will mainly answer two questions

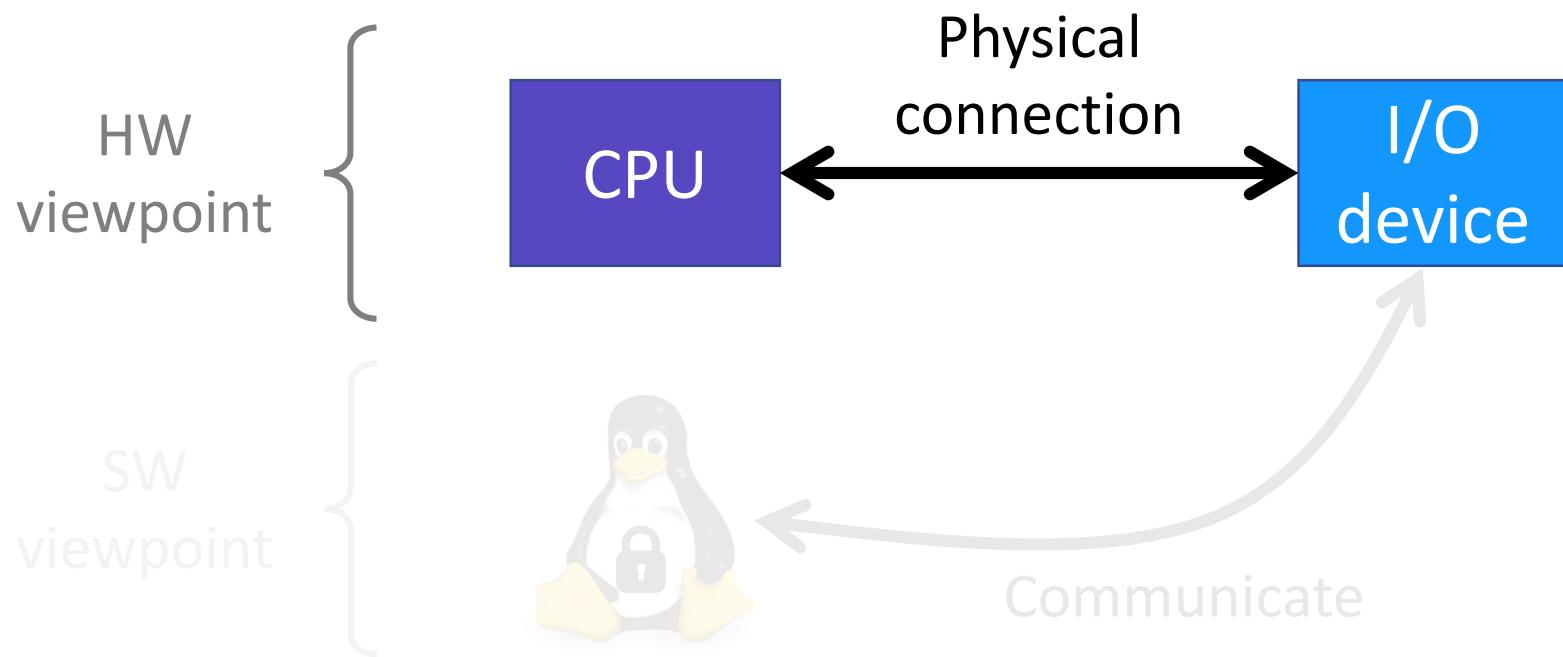


Today's Roadmap

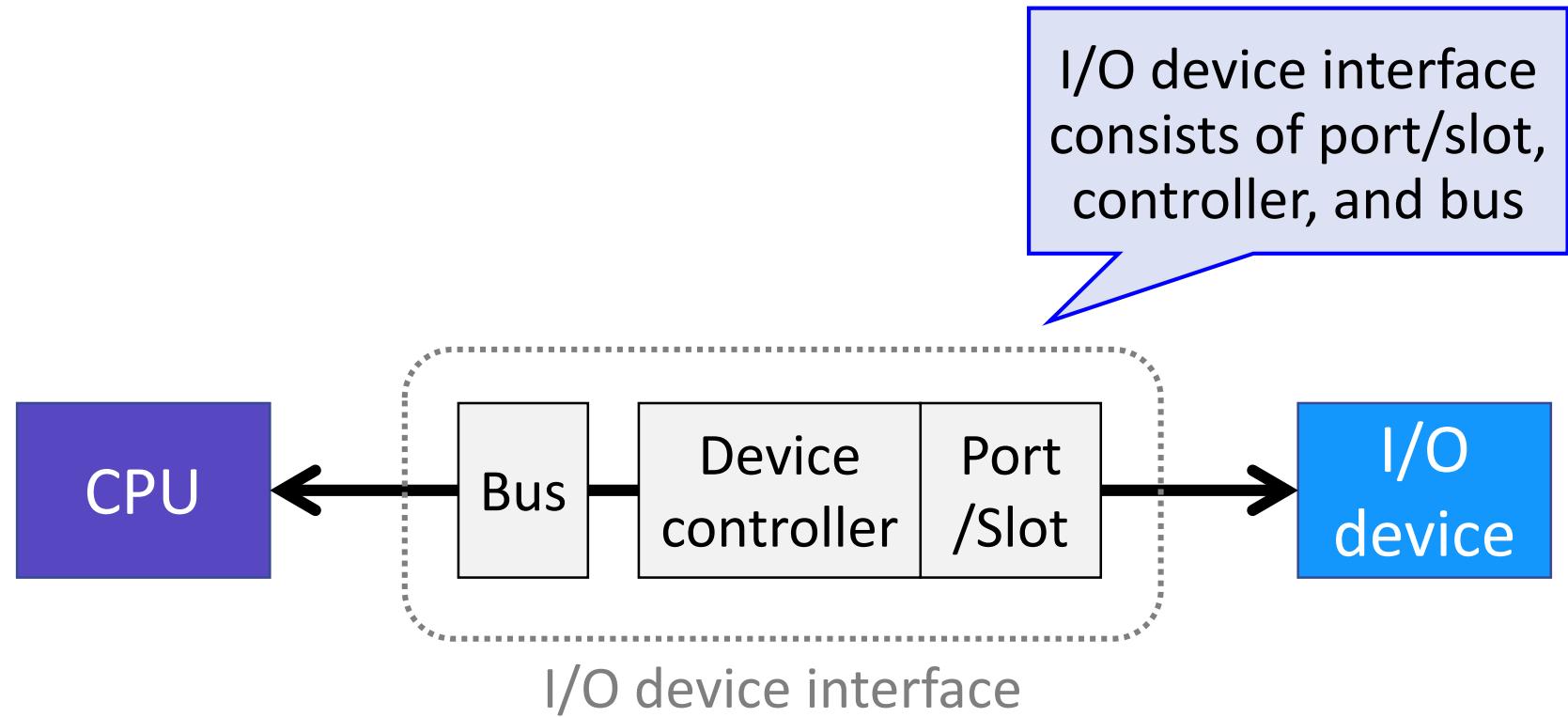
- Today we will mainly answer two questions



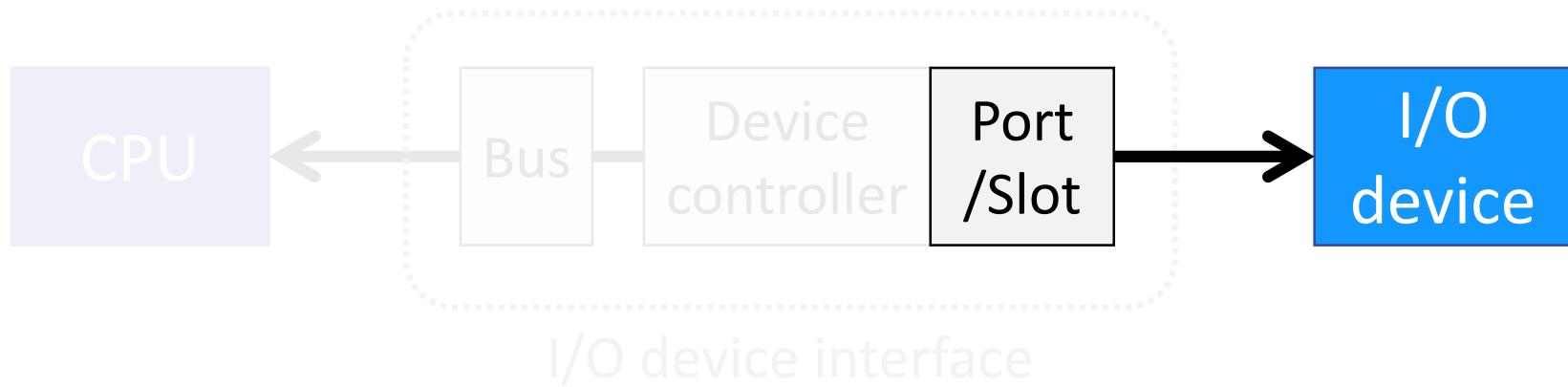
I/O Device Interface



I/O Device Interface



Interface #1: Port/Slot





Input/Output Ports

- Ports are physical connection points that allow the hardware of a computer to be expanded (socket type)

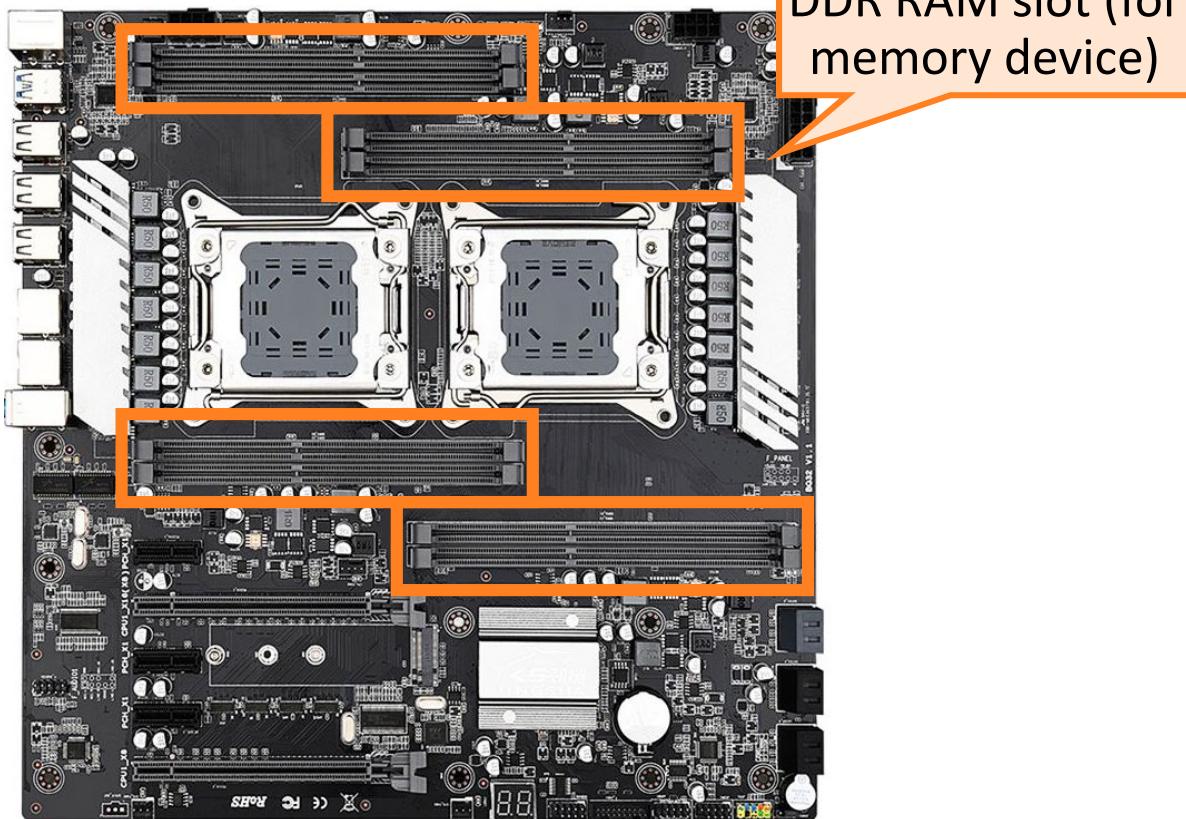


► Connecting Peripherals



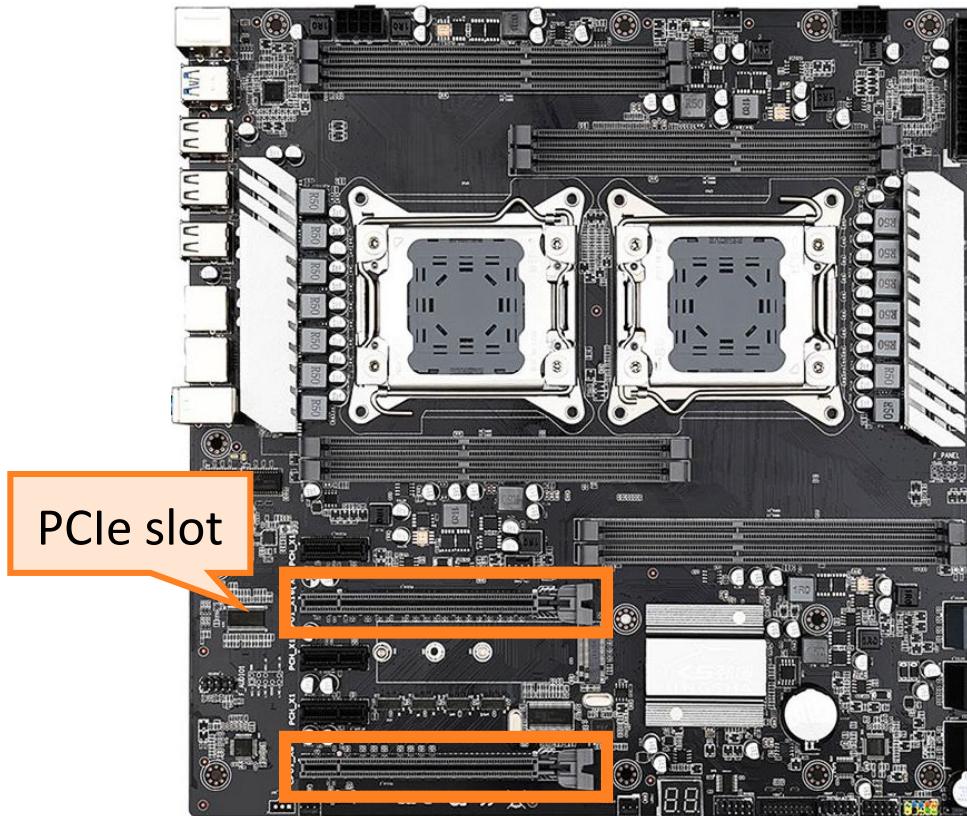
Expansion Slots

- Slots are physical connection points that allow the hardware of a computer to be expanded (a dock for a printed circuit board)



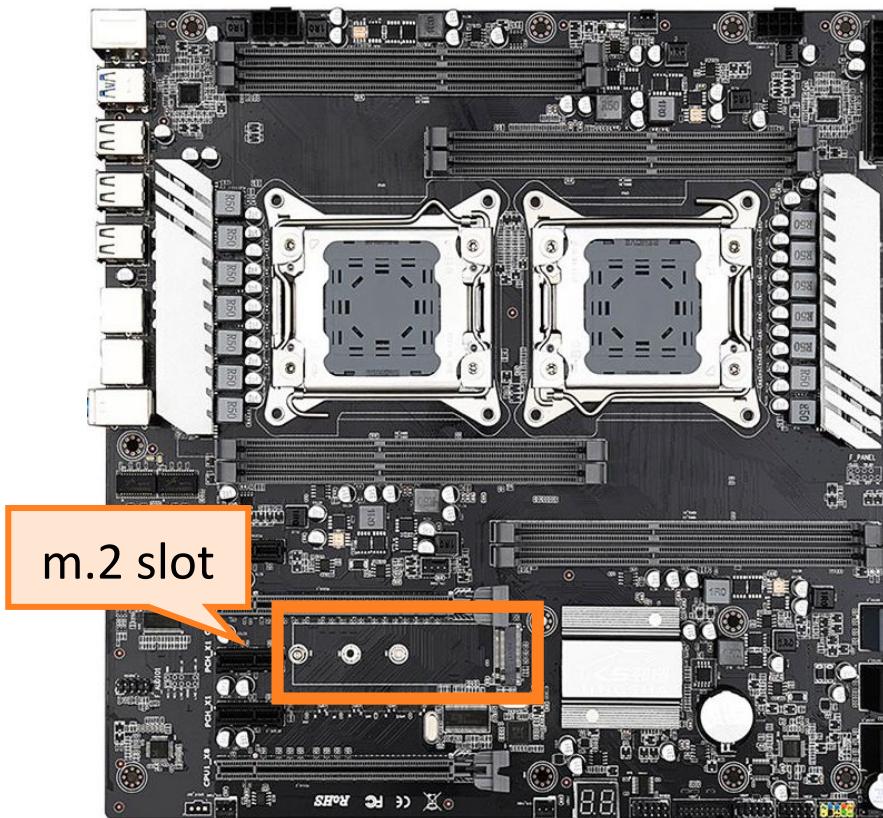
Expansion Slots

- Slots are physical connection points that allow the hardware of a computer to be expanded (a dock for a printed circuit board)

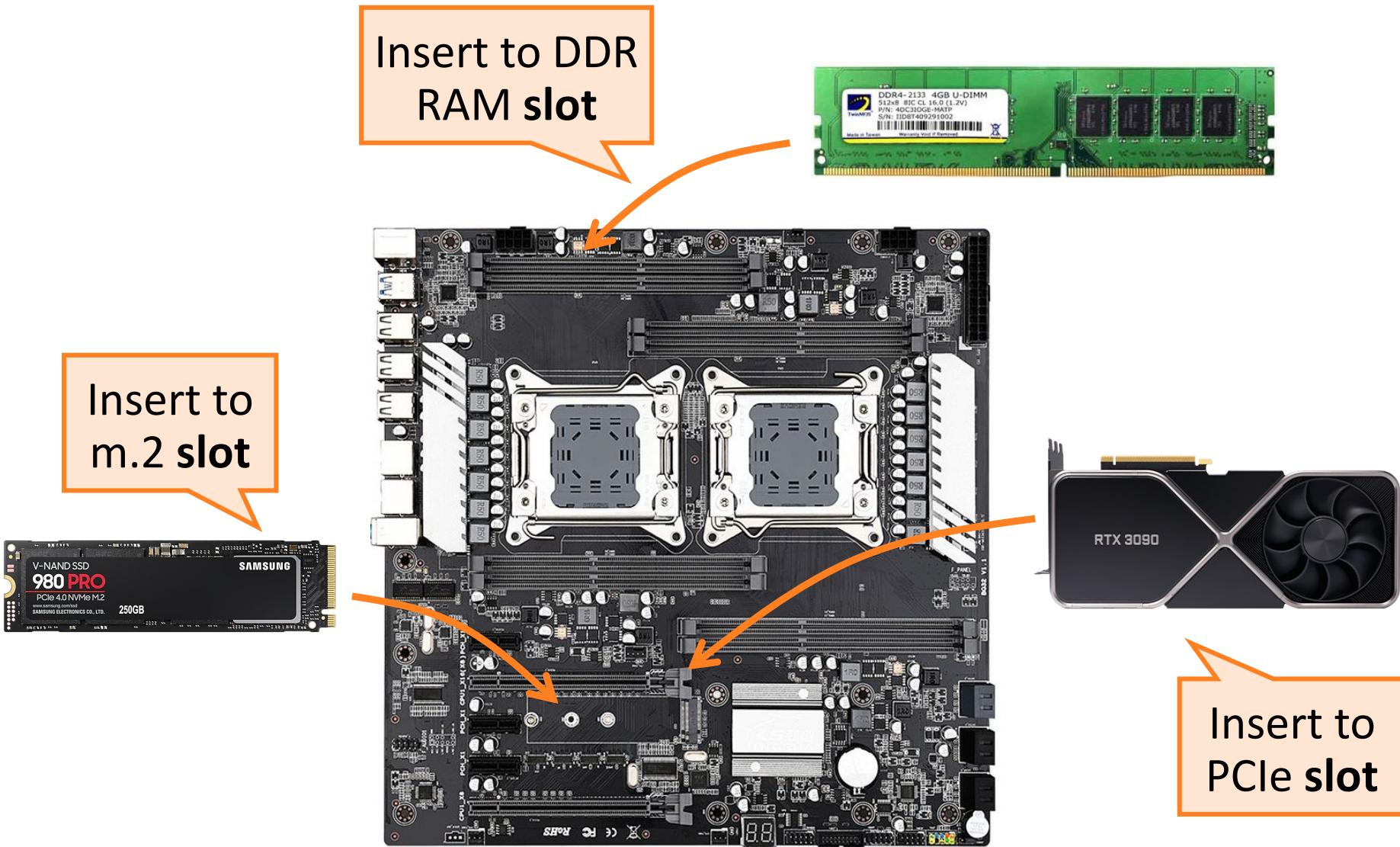


Expansion Slots

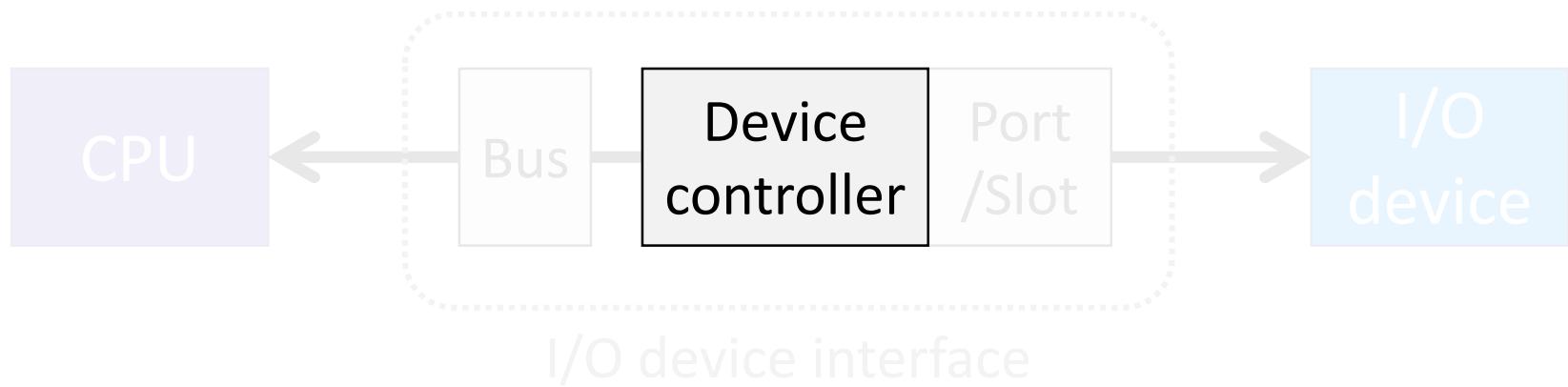
- Slots are physical connection points that allow the hardware of a computer to be expanded (a dock for a printed circuit board)



Connecting Peripherals

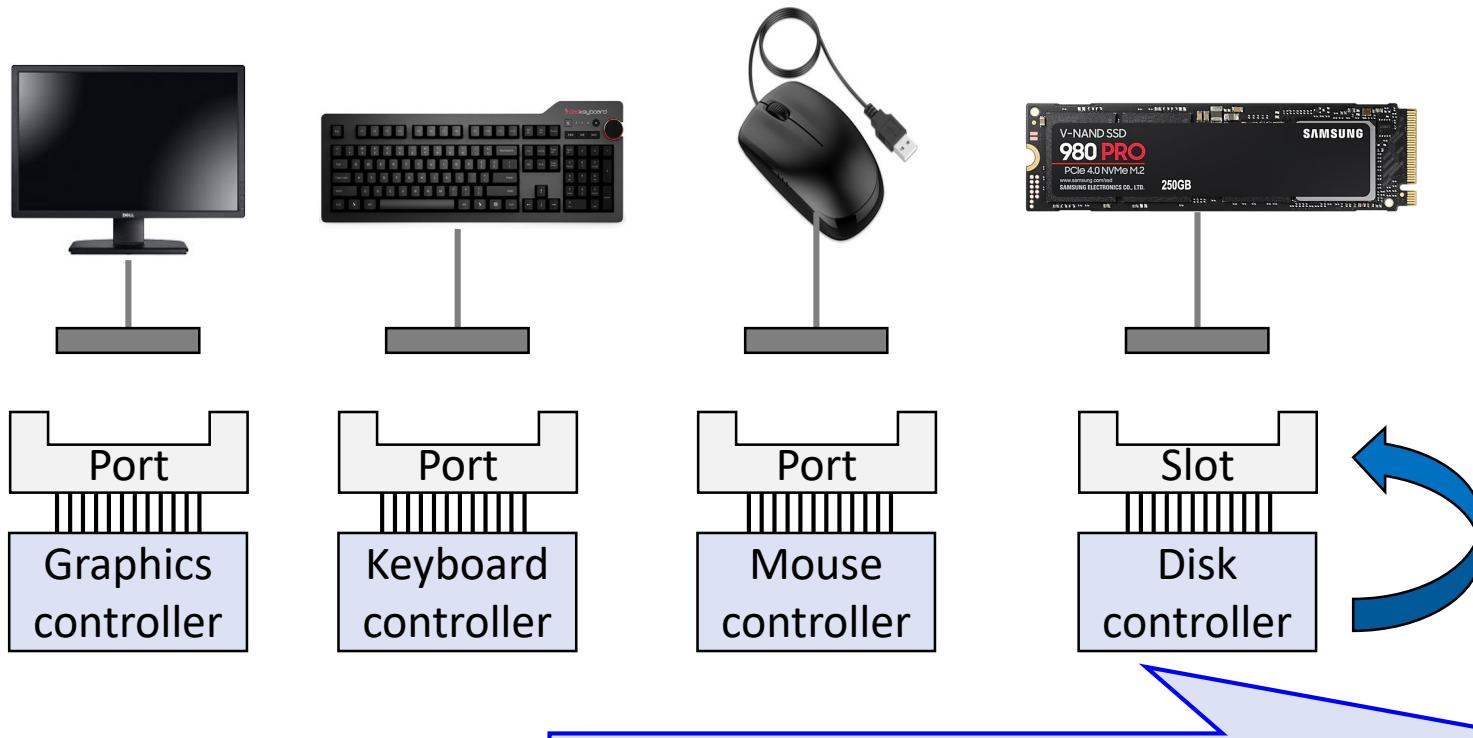


Interface #2: Device Controller



Controller (a.k.a host adapter)

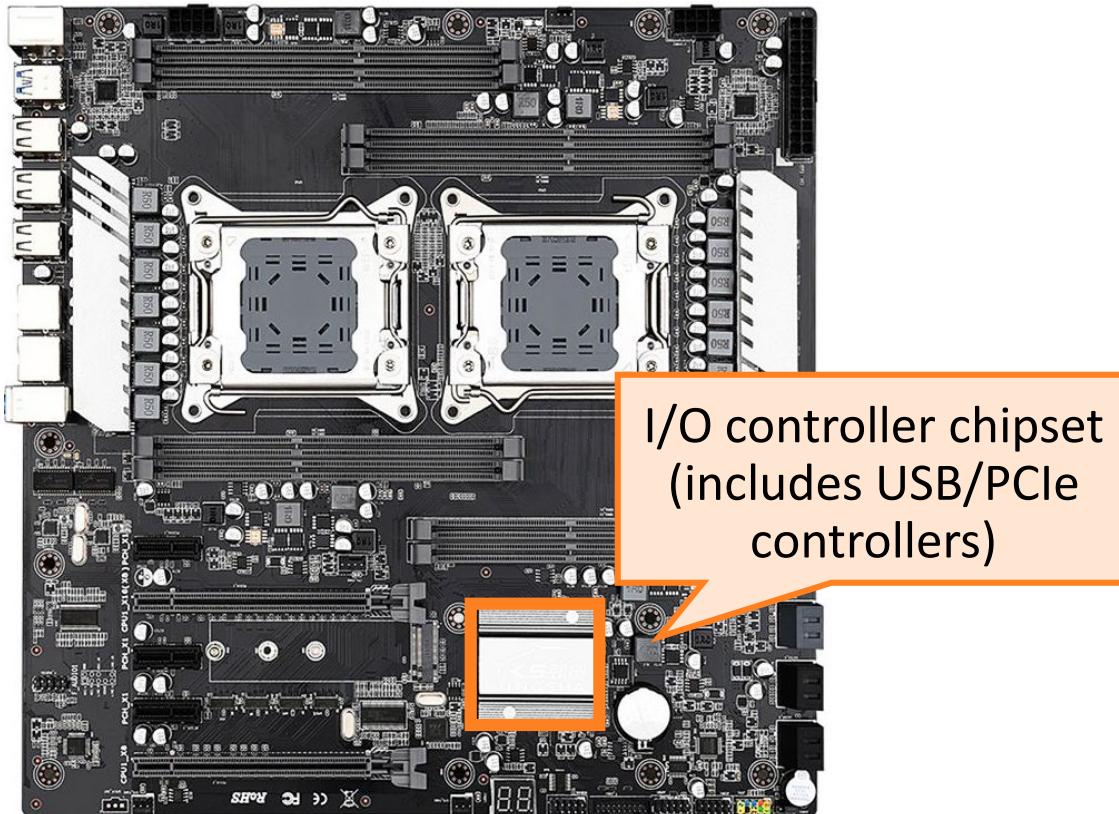
- Electronics that operate port/slot





#1: Integrated Controller

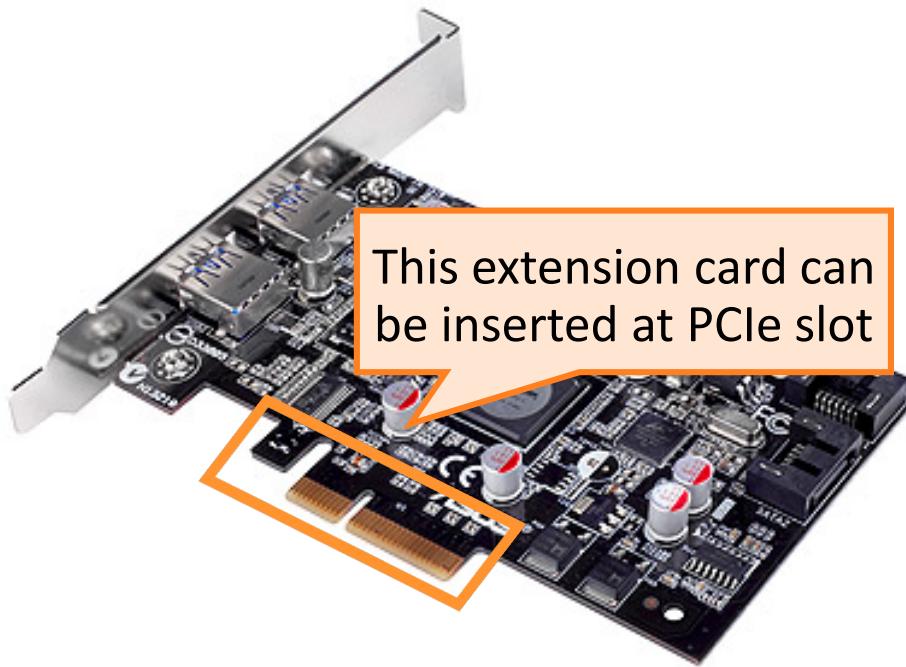
- Sometimes I/O controller is integrated into the mainboard





#2: Separated Circuit Board

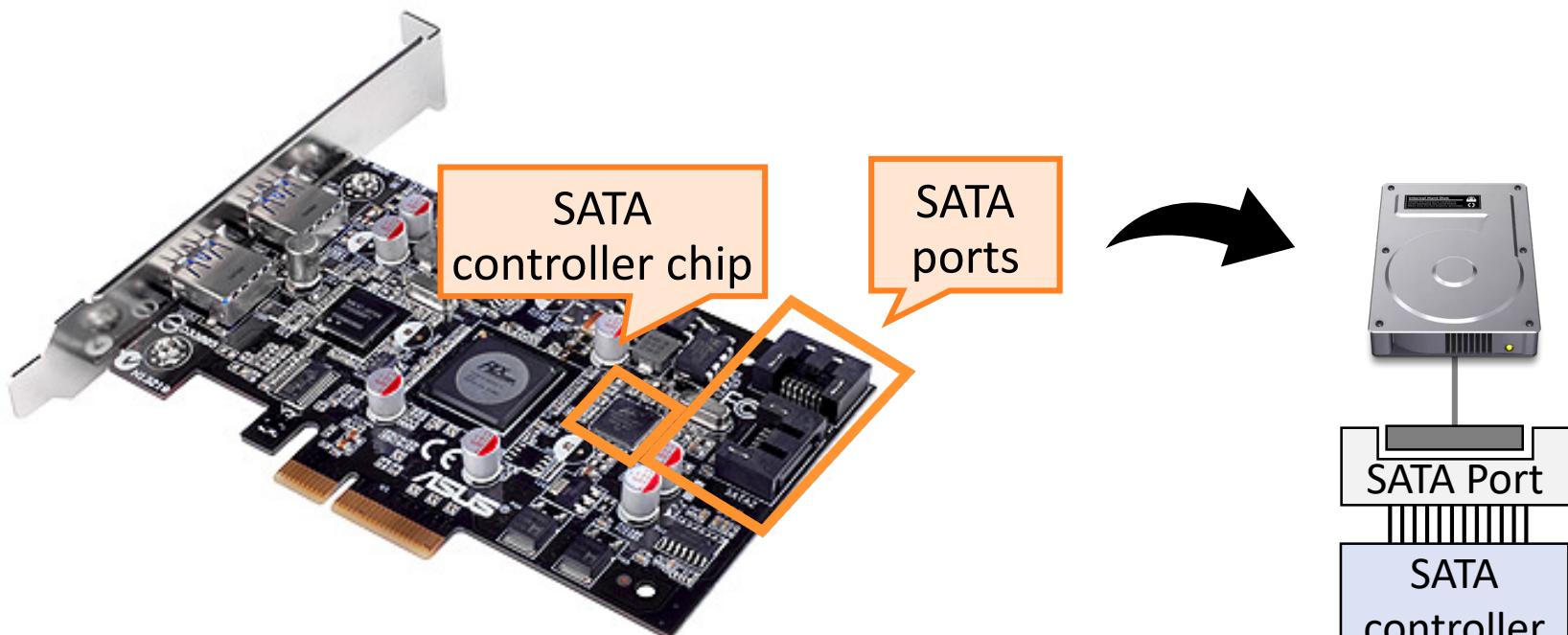
- Sometimes I/O controller is separated as a extension card



ASUS U3S6 2xUSB 3.0 / 2x SATA

#2: Separated Circuit Board

- Sometimes I/O controller is separated as a extension card

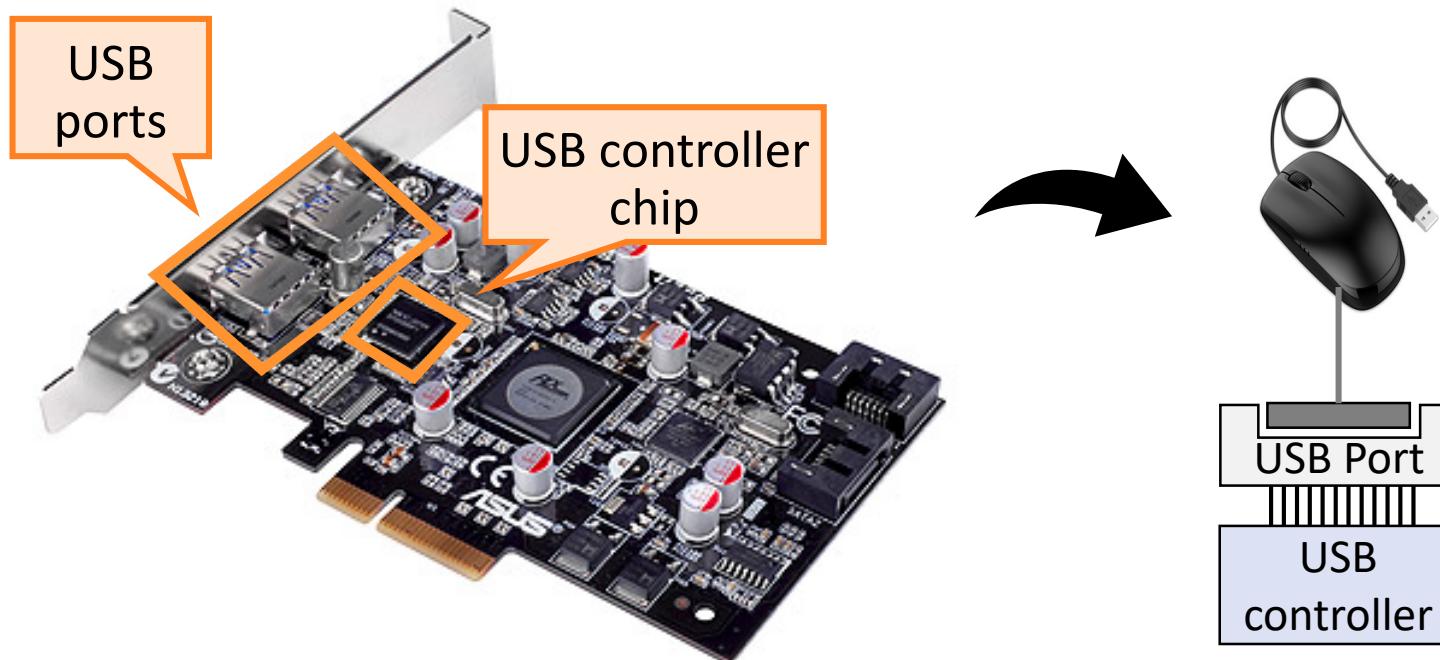


ASUS U3S6 2xUSB 3.0 / 2x SATA



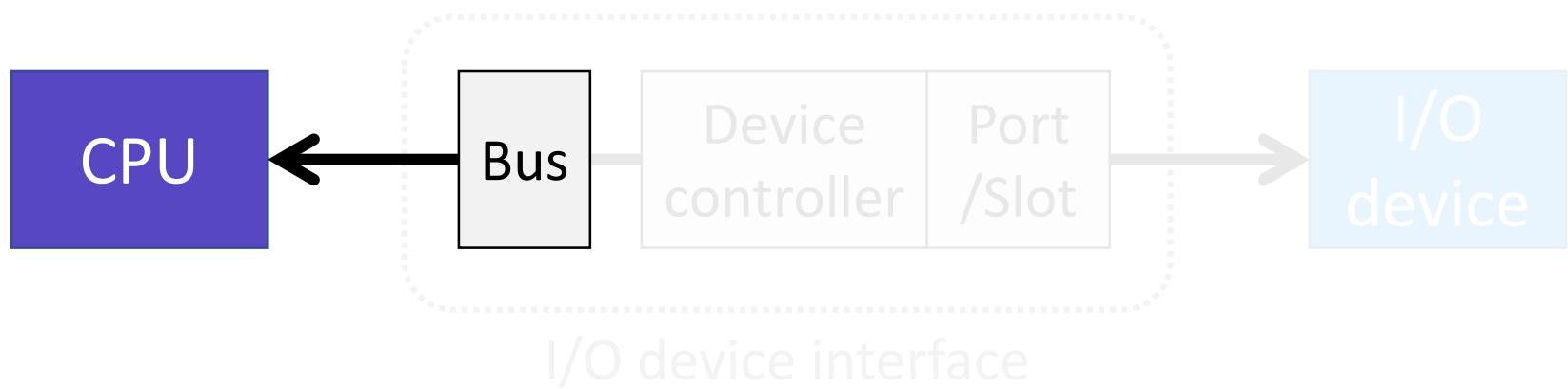
#2: Separated Circuit Board

- Sometimes I/O controller is separated as a extension card



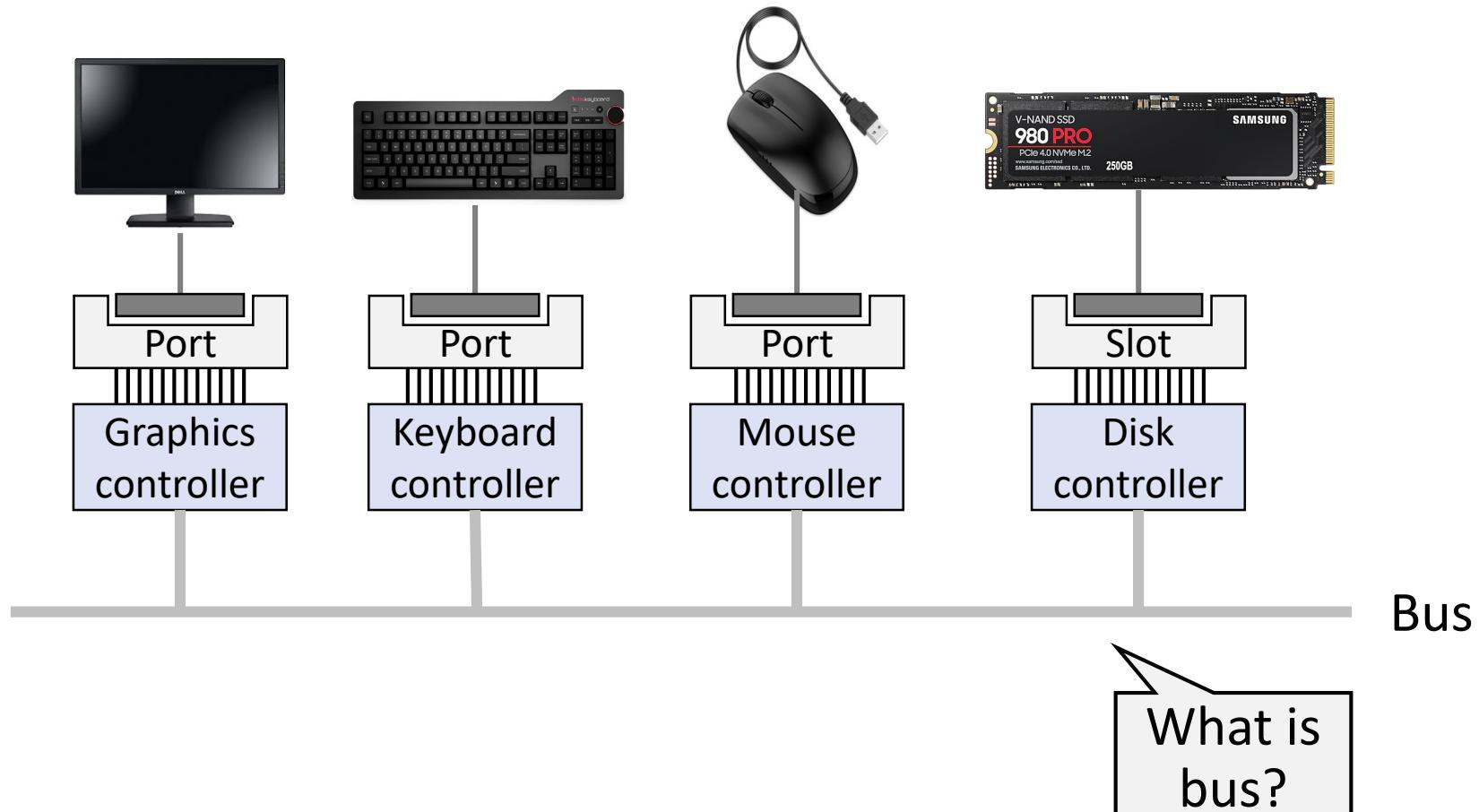
ASUS U3S6 2xUSB 3.0 / 2x SATA

Interface #3: Bus



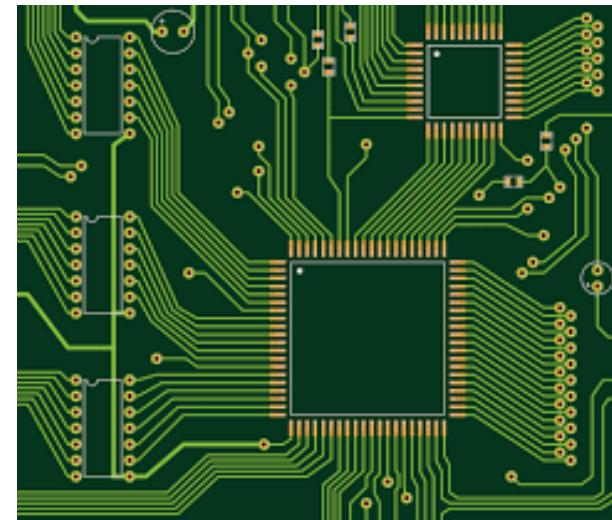
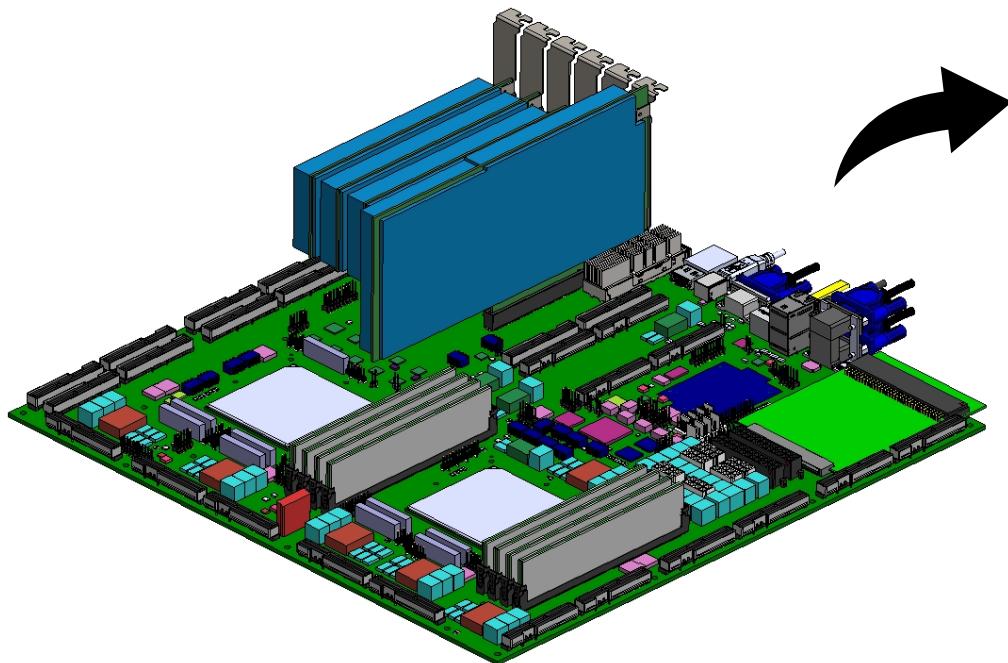


I/O Bus



Bus

- Motherboards require connecting all devices in a system

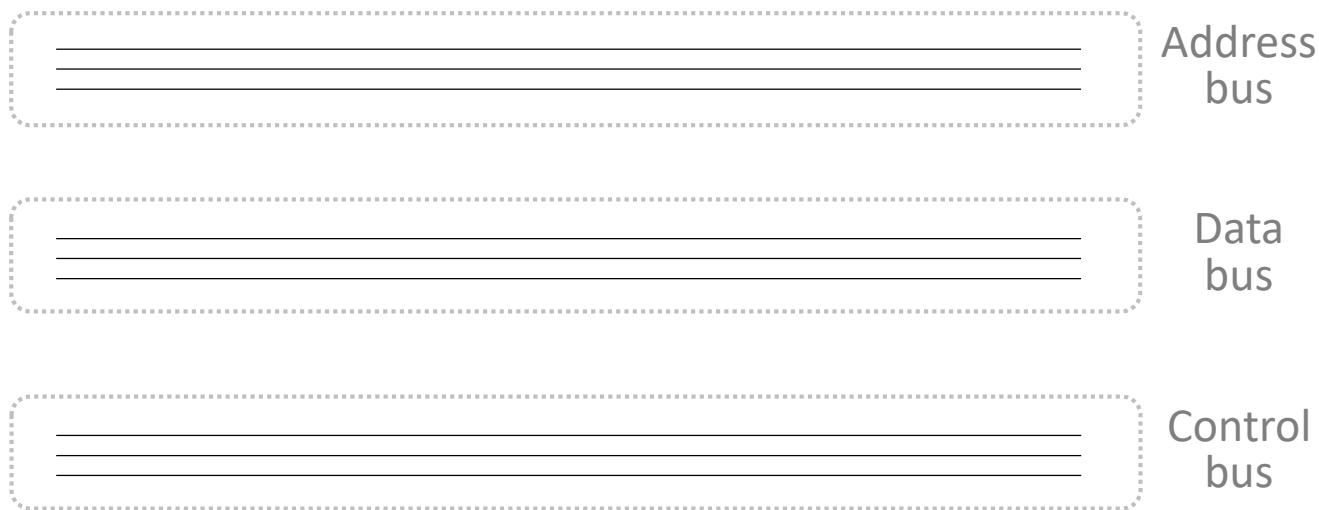


See the back of your mother board; there are circuit traces (wires)



Bus

- Comprised of many individual circuit traces grouped together to form a bus



Bus

- Comprised of many individual circuit traces forming busses

Used to carry the address of a device. Each device in the computer (memory and I/O devices) is uniquely identified by an index/number (address)

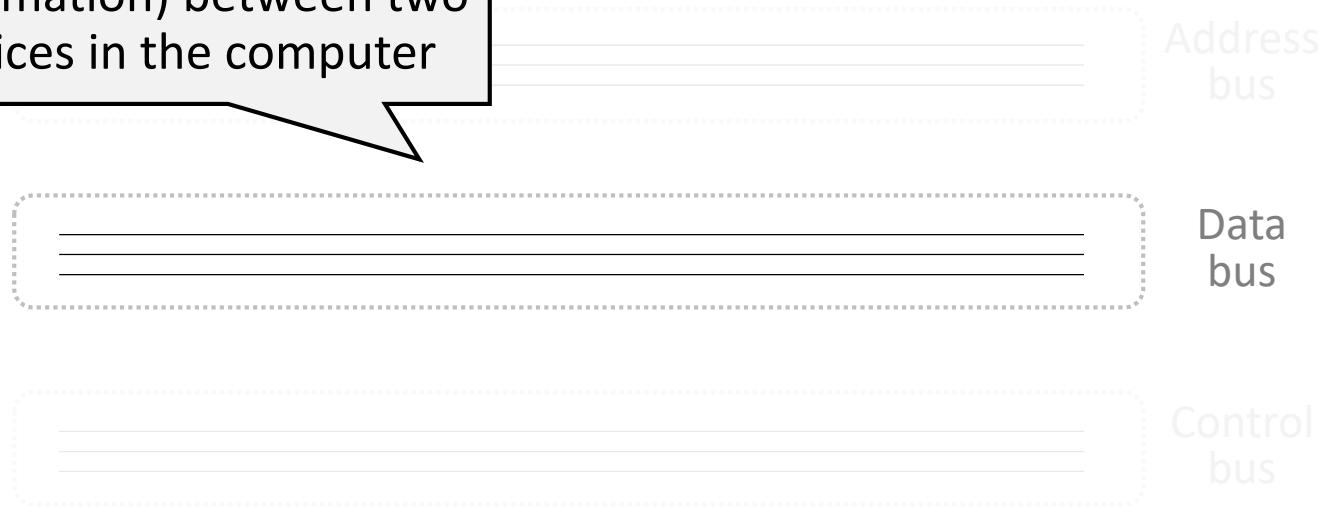




Bus

- Comprised of many individual circuit traces grouped together to form busses

Used to transfer the data (information) between two devices in the computer



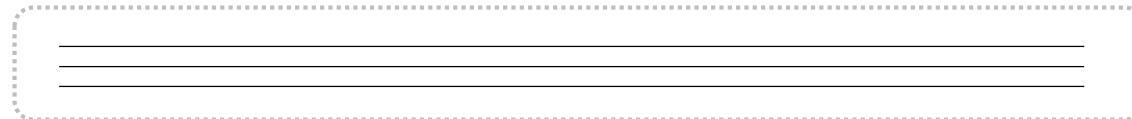


Bus

- Comprised of many individual circuit traces grouped together to form busses

Signal	
R/W	<p>Read vs. Write indicator signal</p> <ul style="list-style-type: none">- R/W = 1 → Wants to perform a READ operation- R/W = 0 → Wants to perform a WRITE operation
MEM	<p>Memory vs. IO device indicator signal</p> <ul style="list-style-type: none">- MEM = 1 → Wants to perform an operation w/ memory- MEM = 0 → Wants to perform an operation w/ IO device

Contain many signals

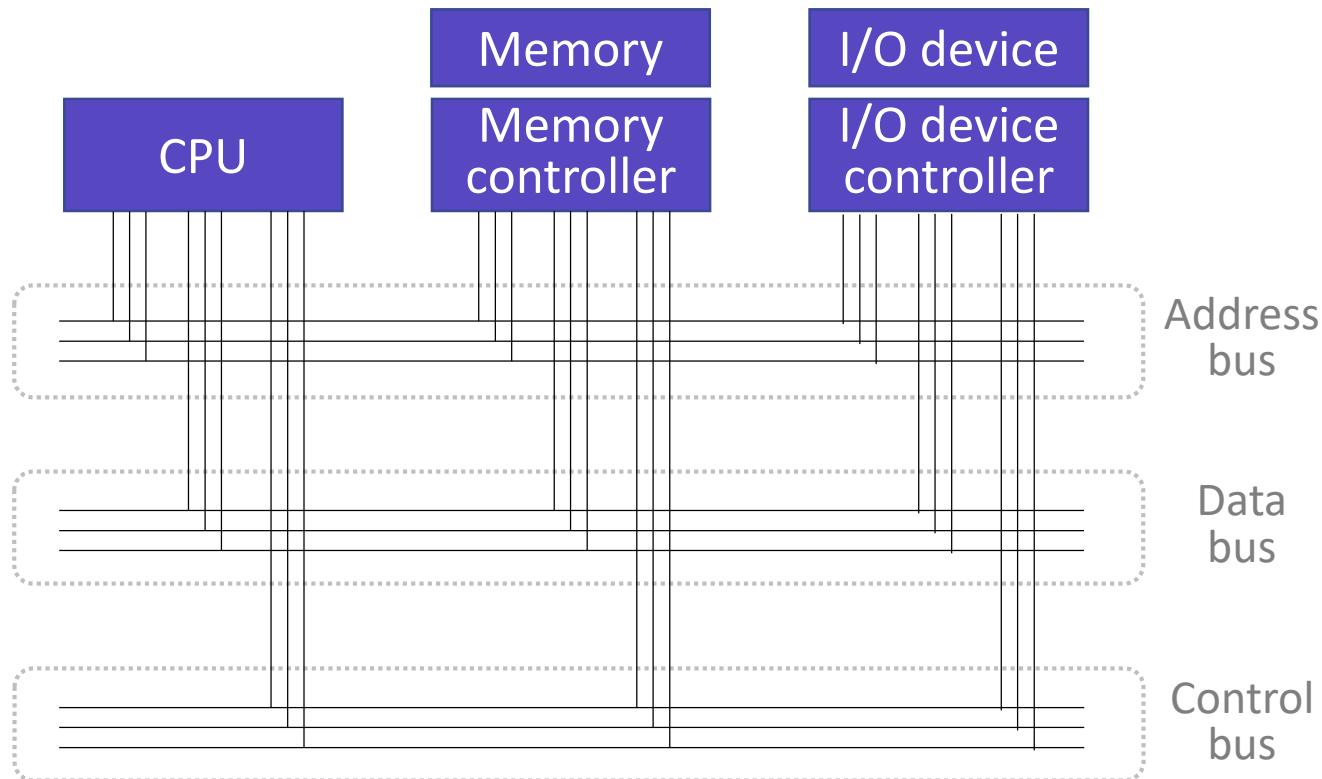


Control bus



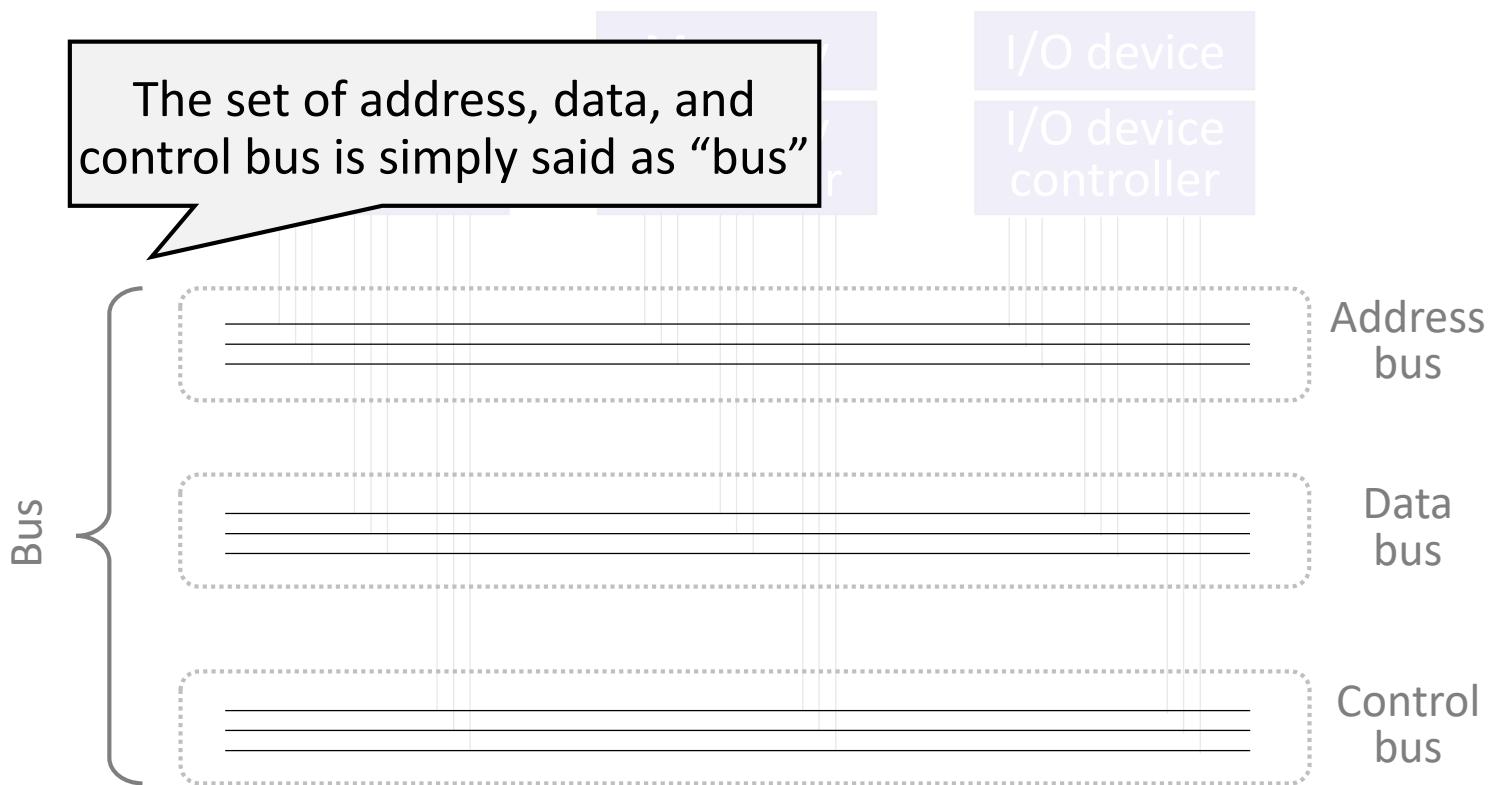
Bus

- Comprised of many individual circuit traces grouped together to form busses



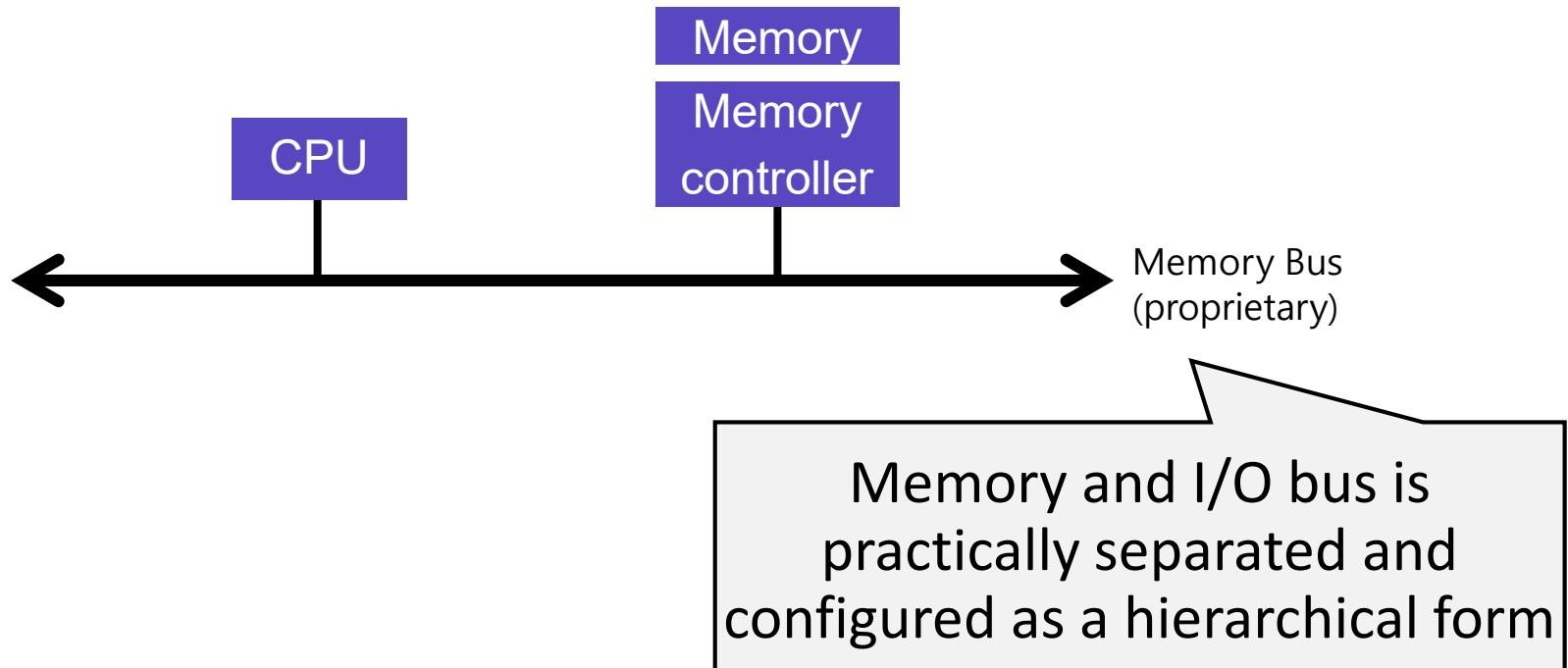
Bus

- Comprised of many individual circuit traces grouped together to form busses



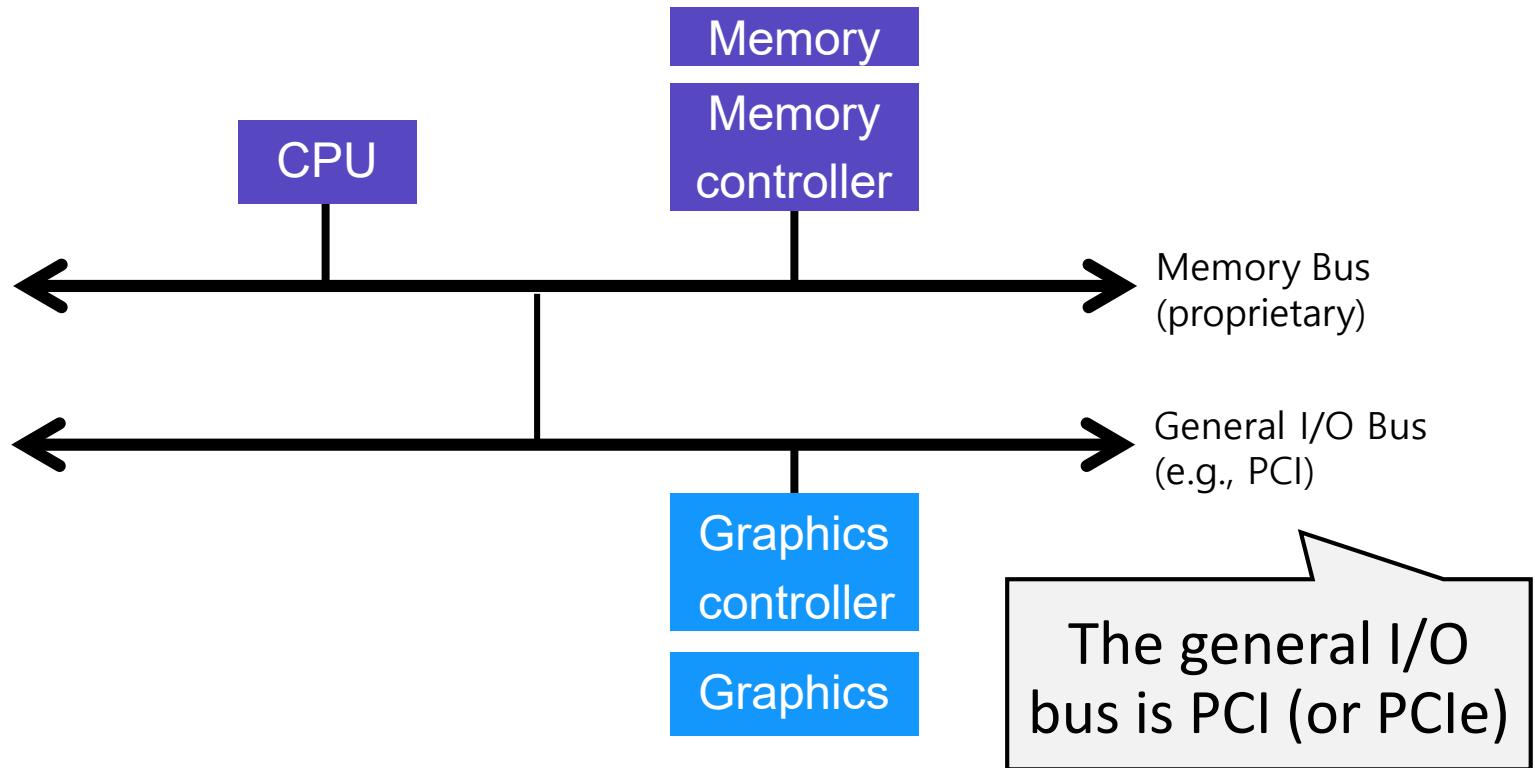


Hierarchical Bus Architecture



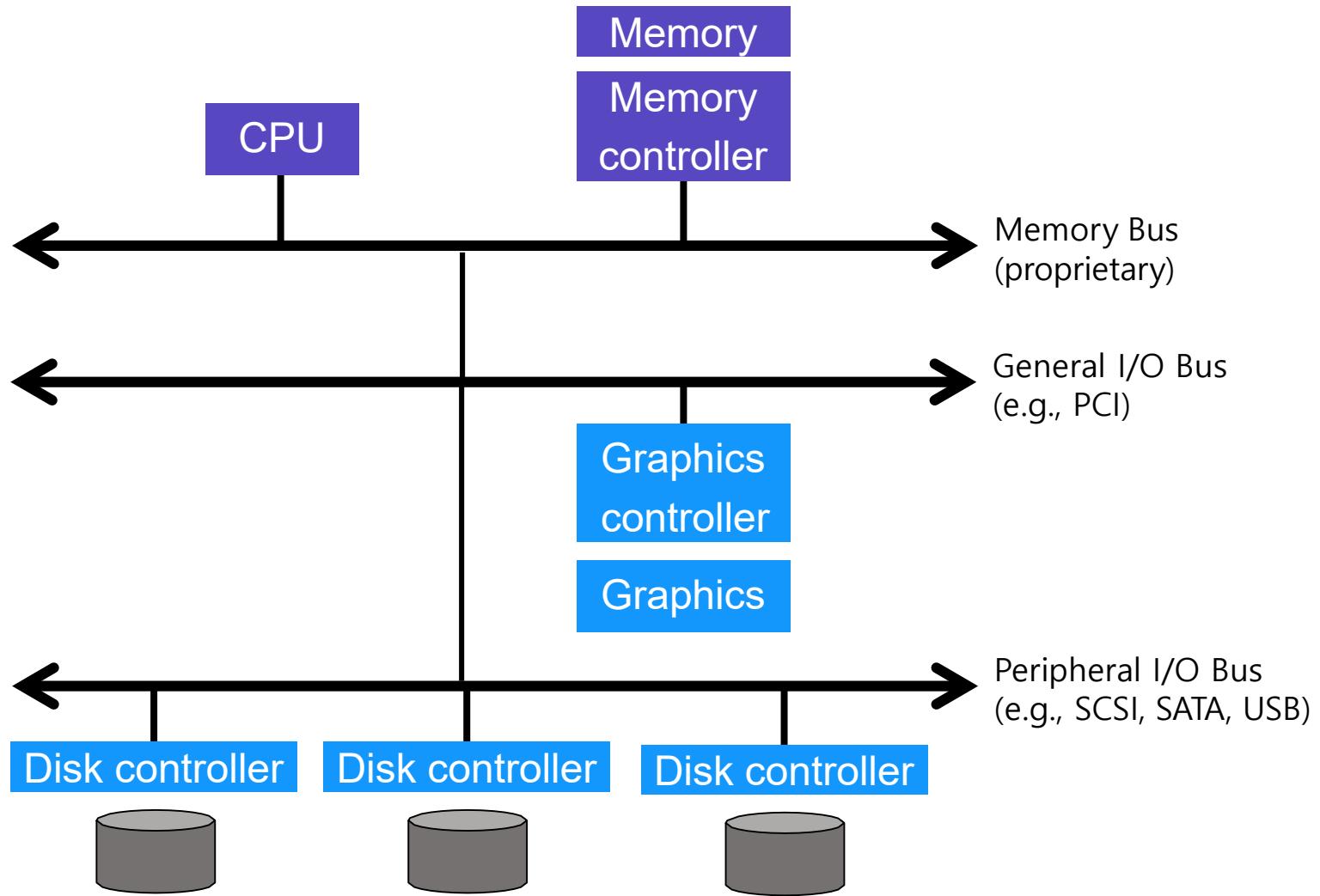


Hierarchical Bus Architecture



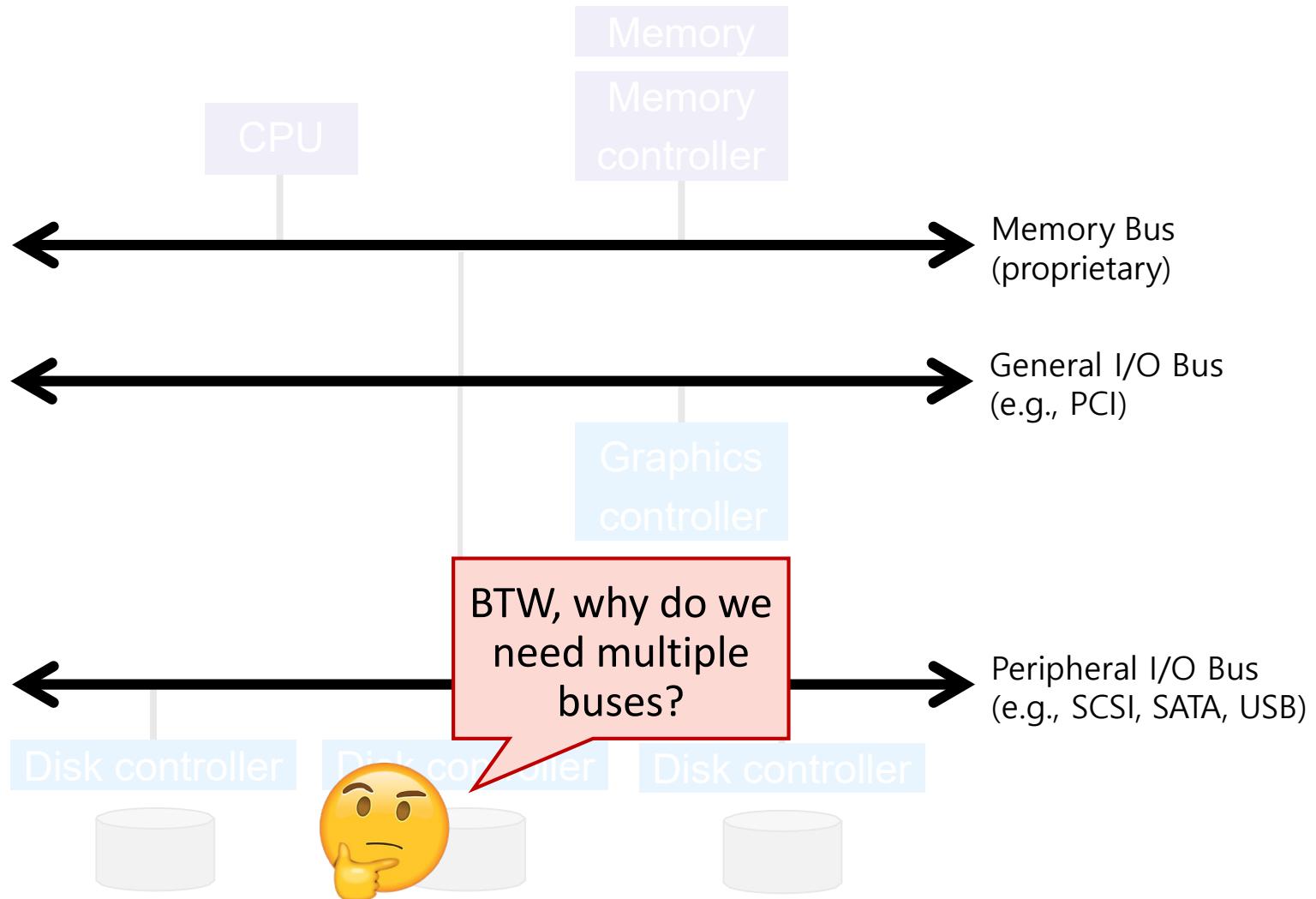


Hierarchical Bus Architecture





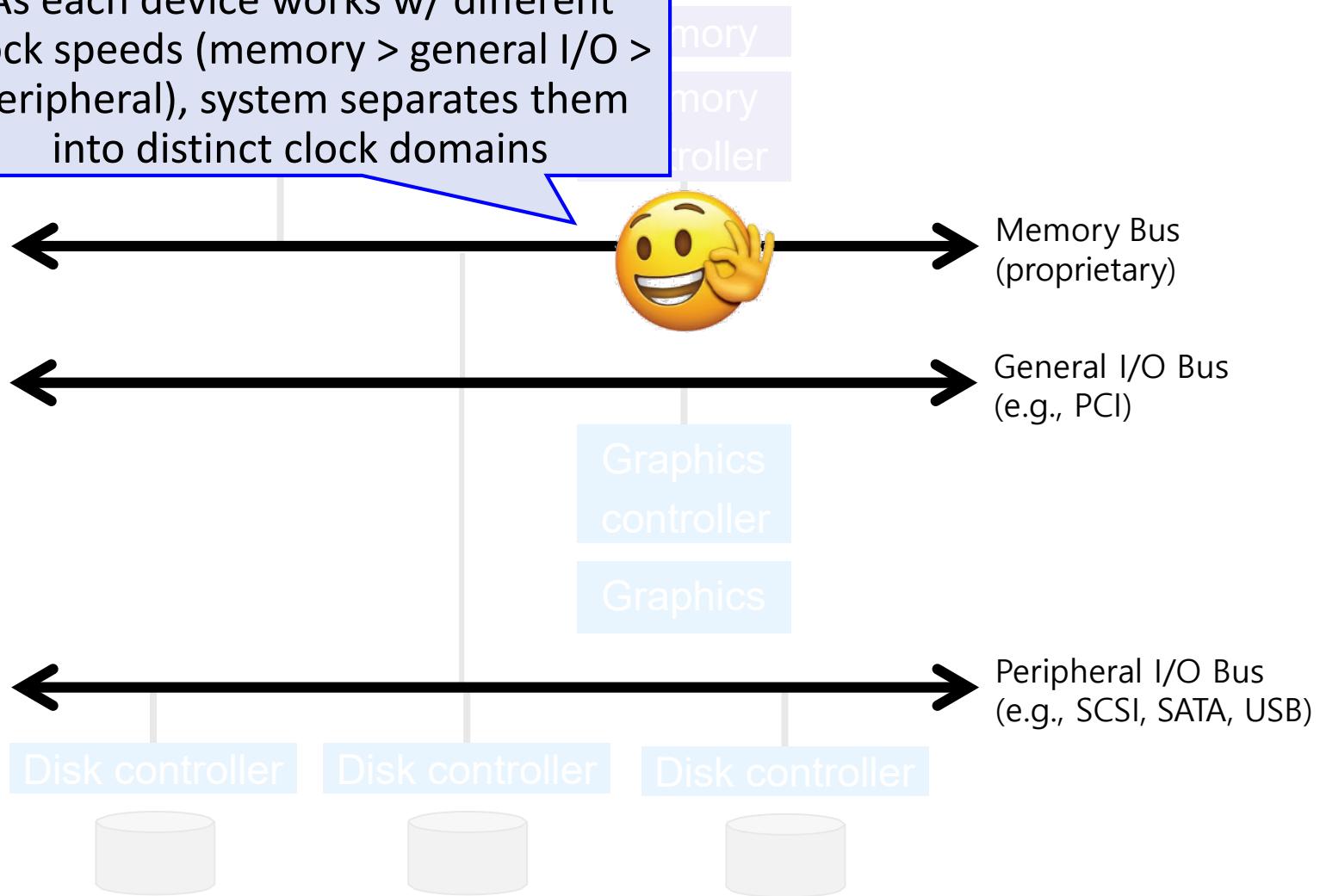
Hierarchical Bus Architecture



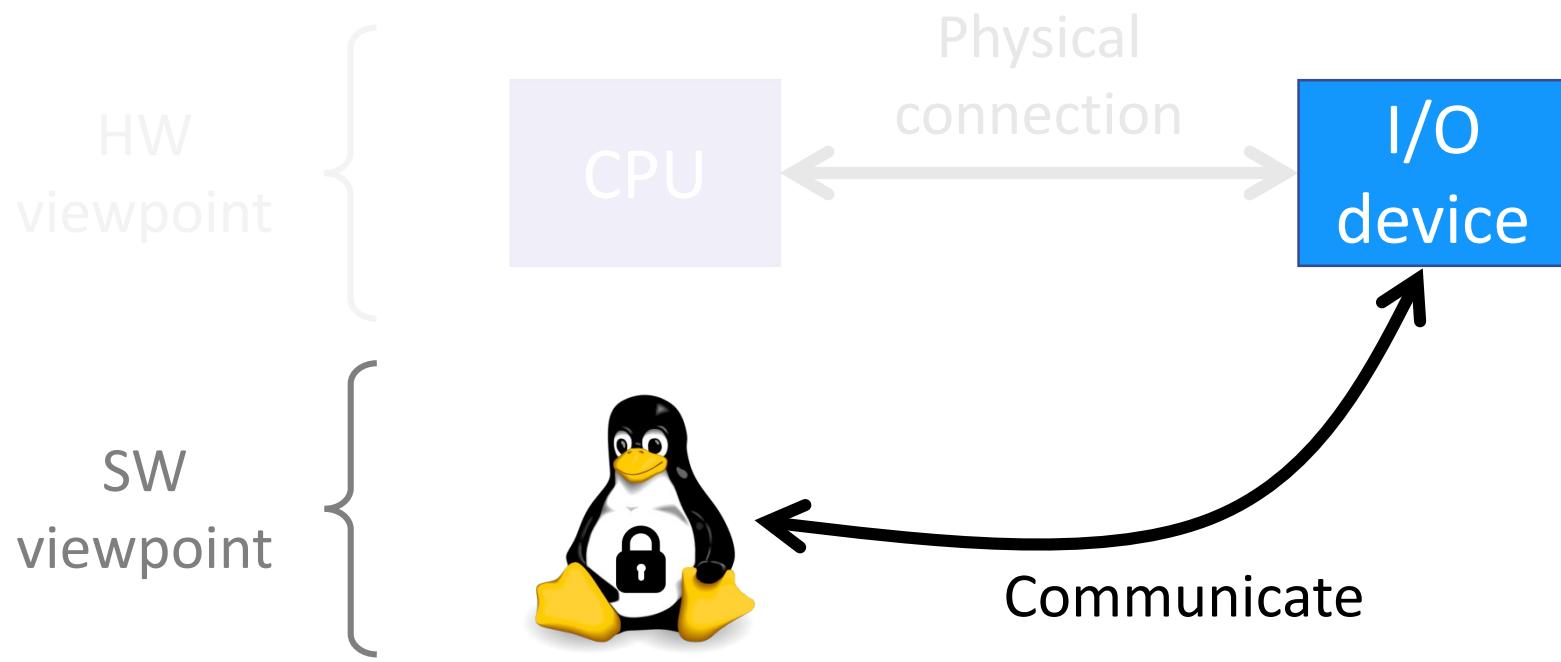


Hierarchical Bus Architecture

As each device works w/ different clock speeds (memory > general I/O > peripheral), system separates them into distinct clock domains

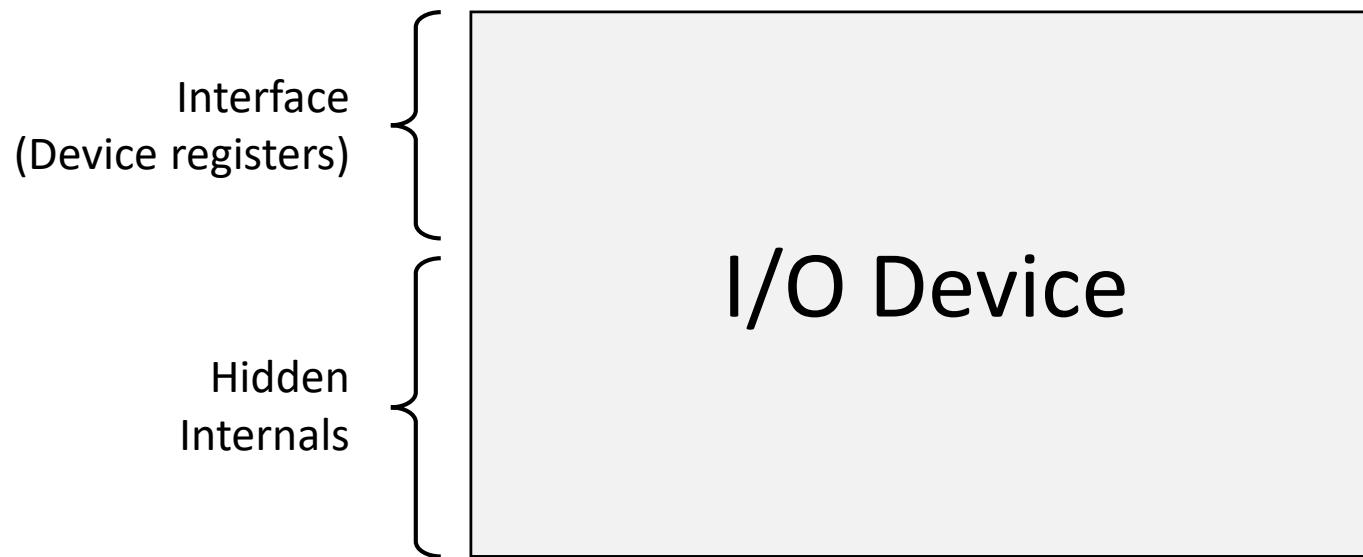


I/O Device Protocol



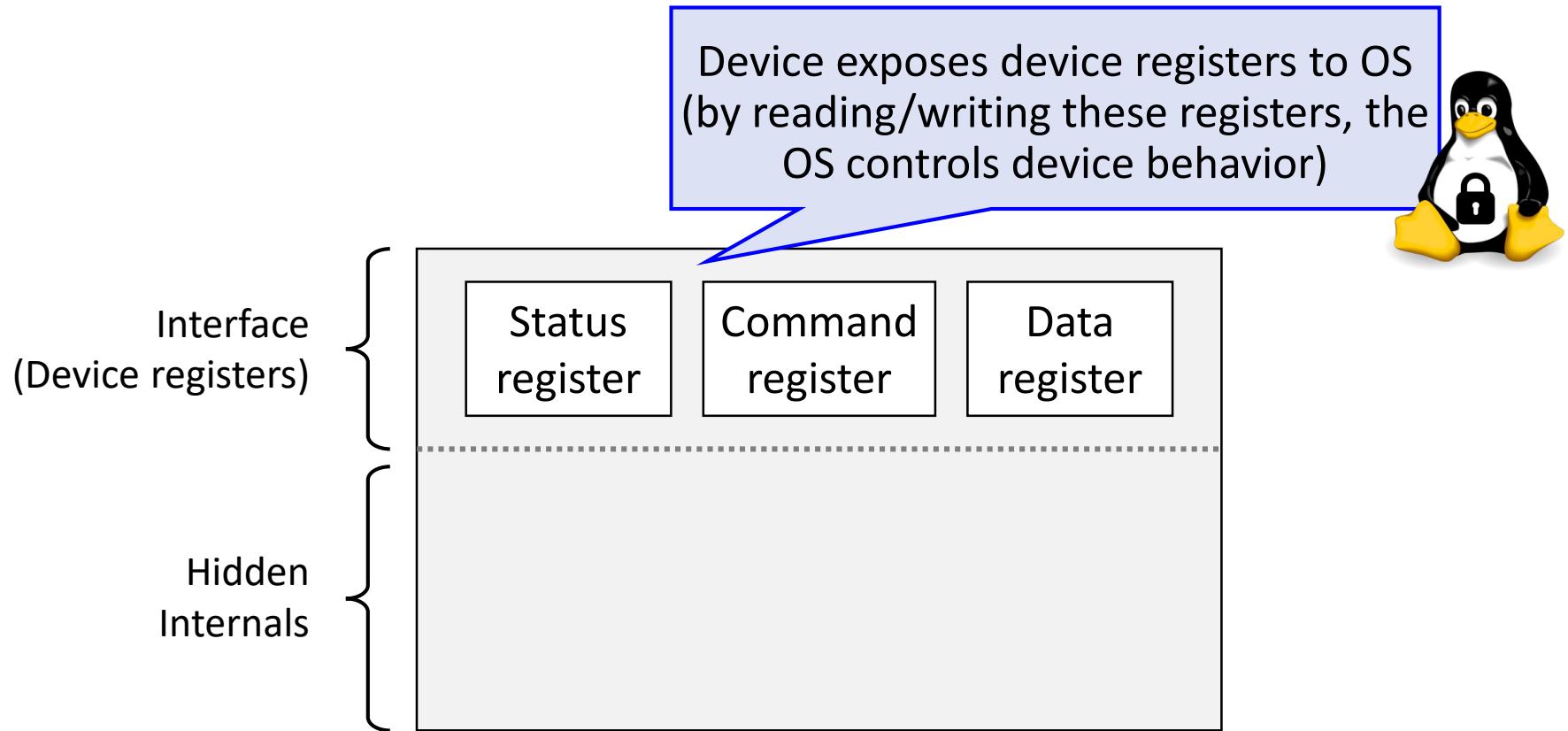
Canonical I/O Device

- Canonical I/O device consists of interface (device registers) and hidden internals



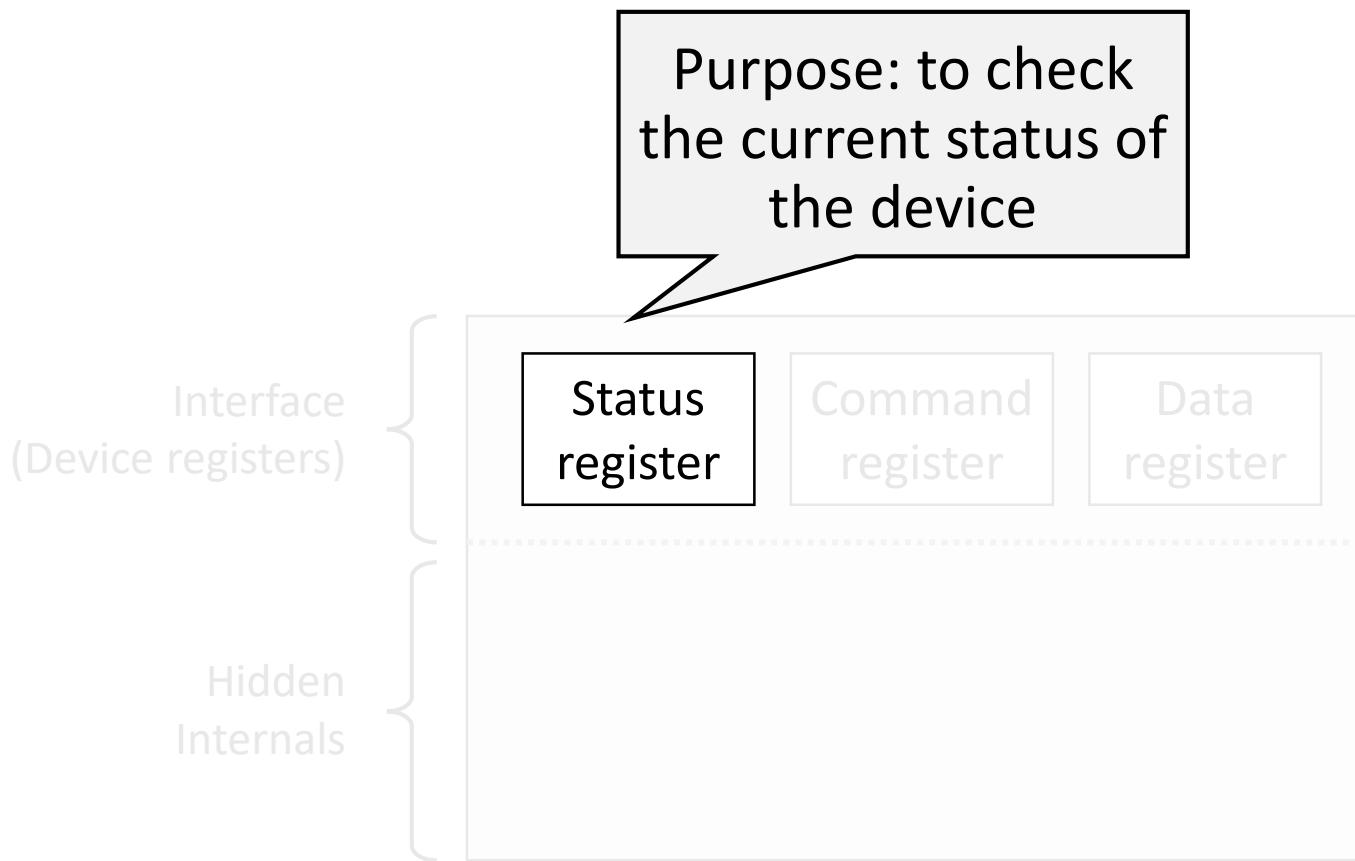
Canonical I/O Device

- Canonical I/O device consists of interface (device registers) and hidden internals



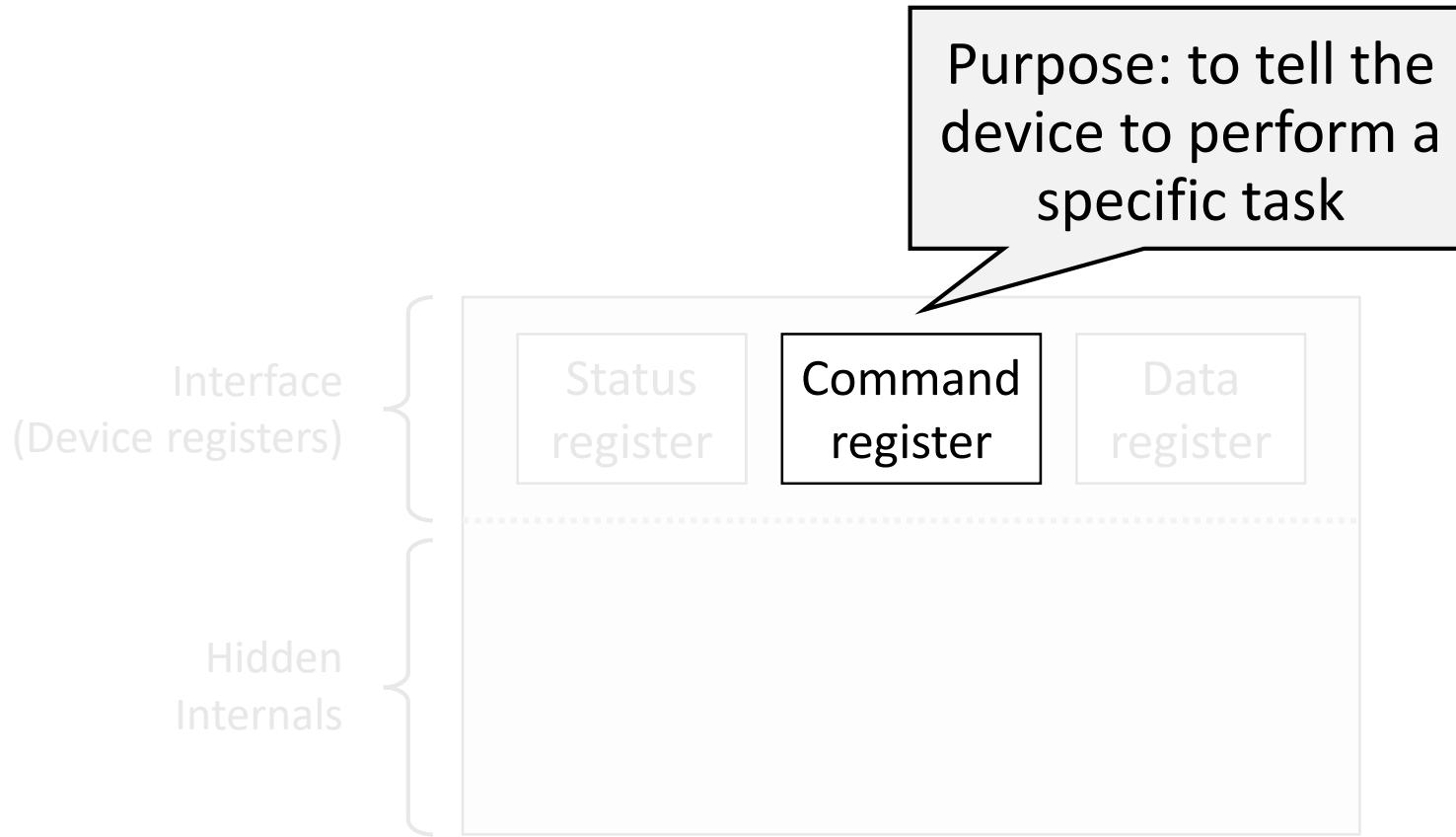
Canonical I/O Device

- Canonical I/O device consists of interface (device registers) and hidden internals



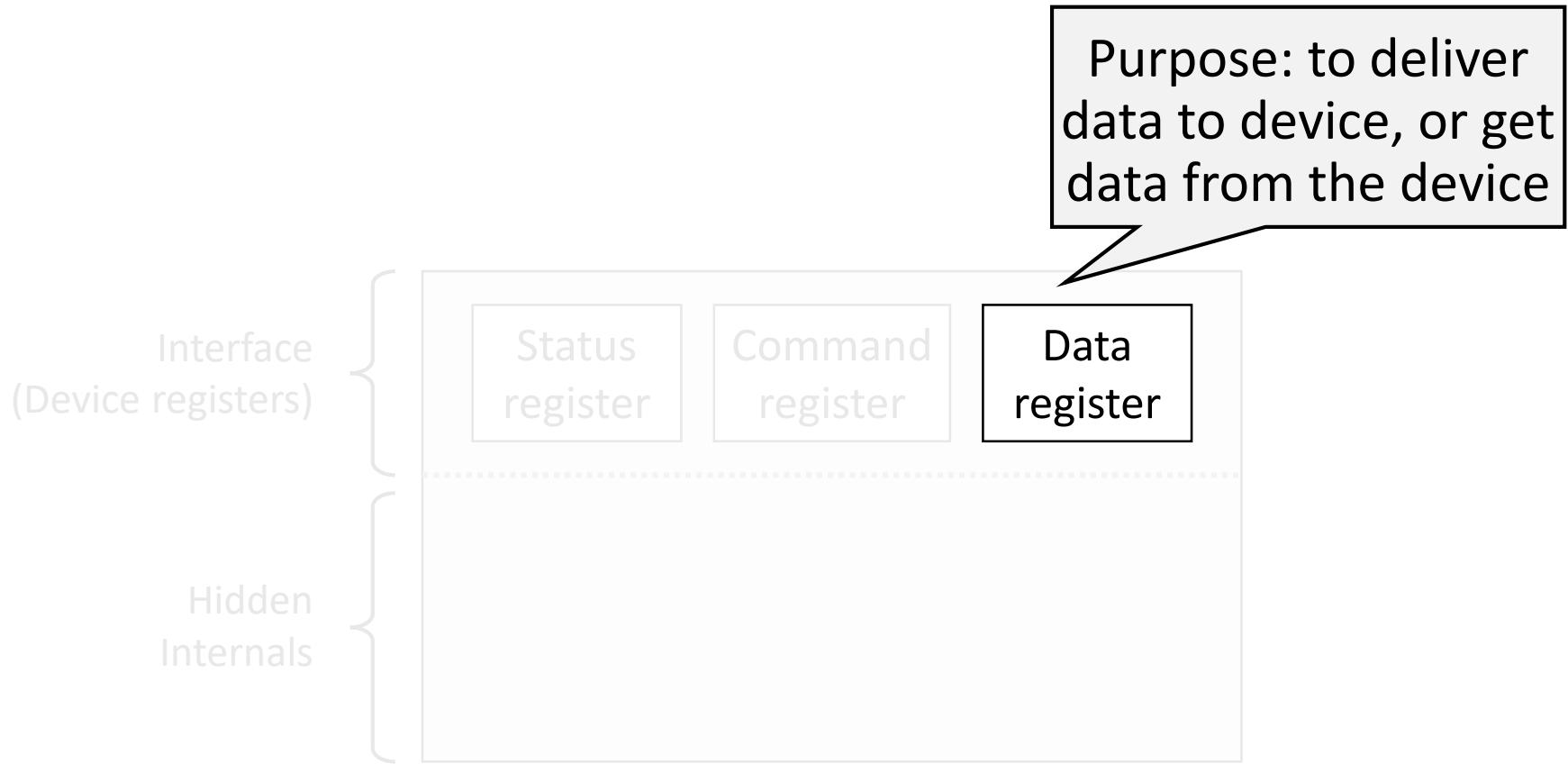
Canonical I/O Device

- Canonical I/O device consists of interface (device registers) and hidden internals



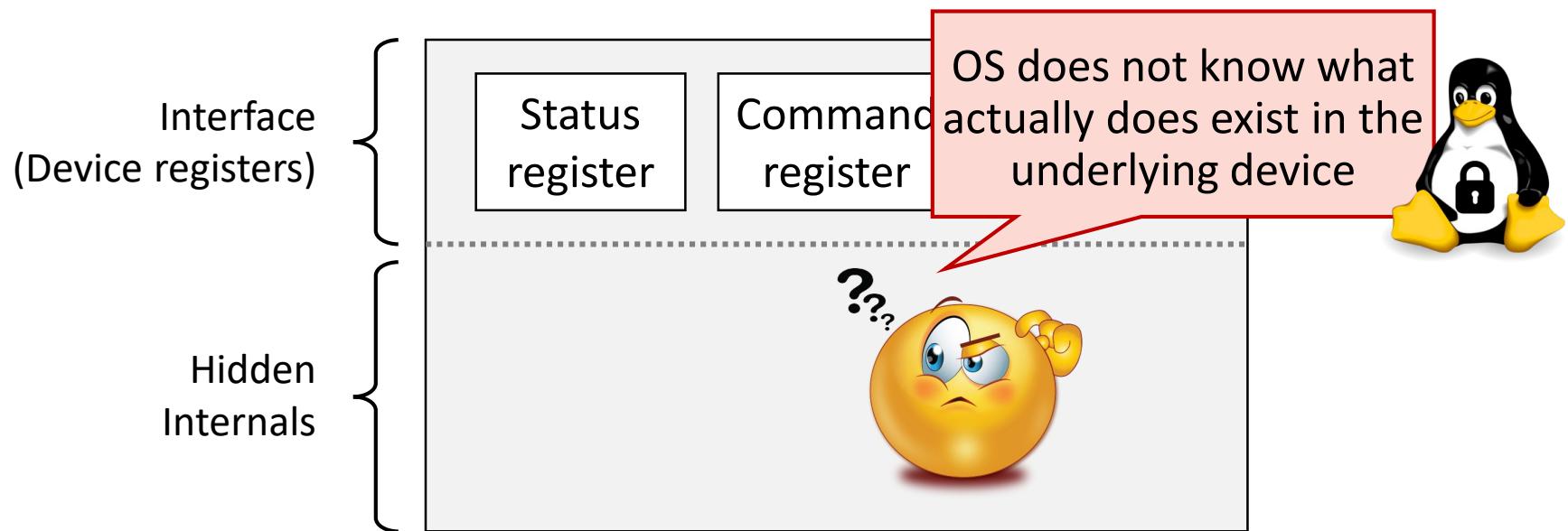
Canonical I/O Device

- Canonical I/O device consists of interface (device registers) and hidden internals



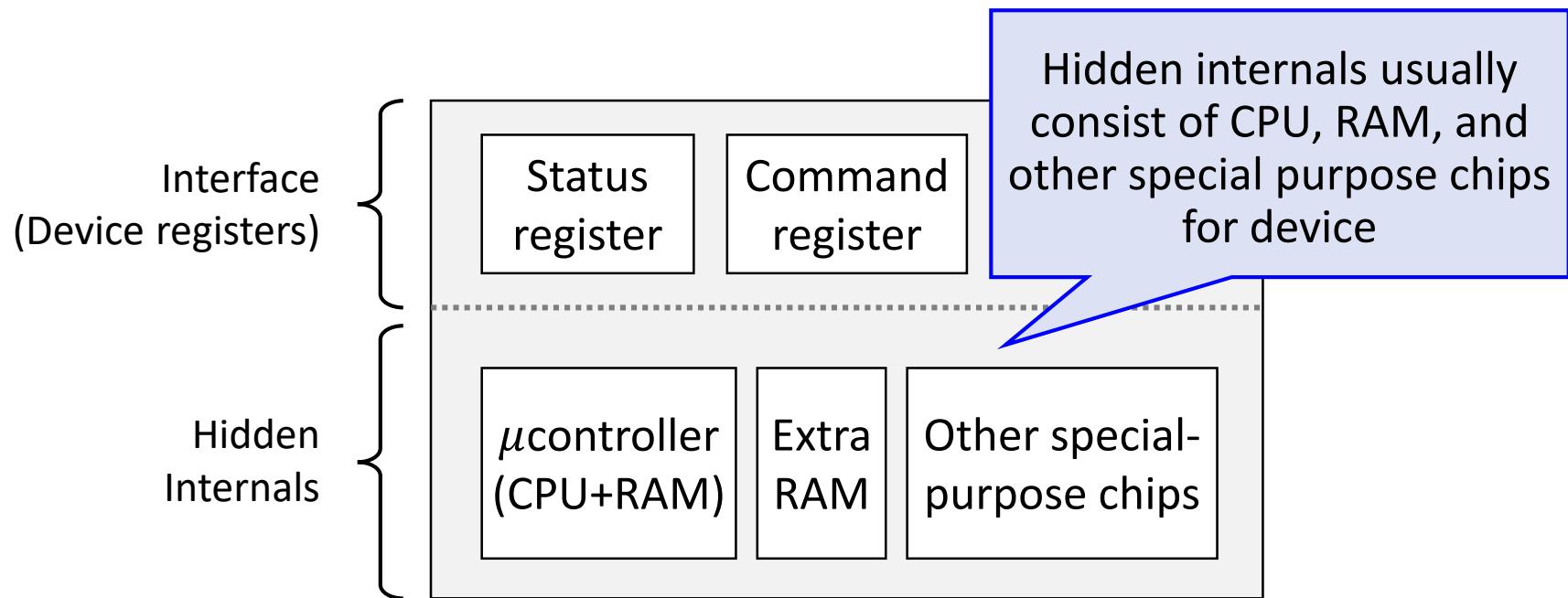
Canonical I/O Device

- Canonical I/O device consists of interface (device registers) and hidden internals



Canonical I/O Device

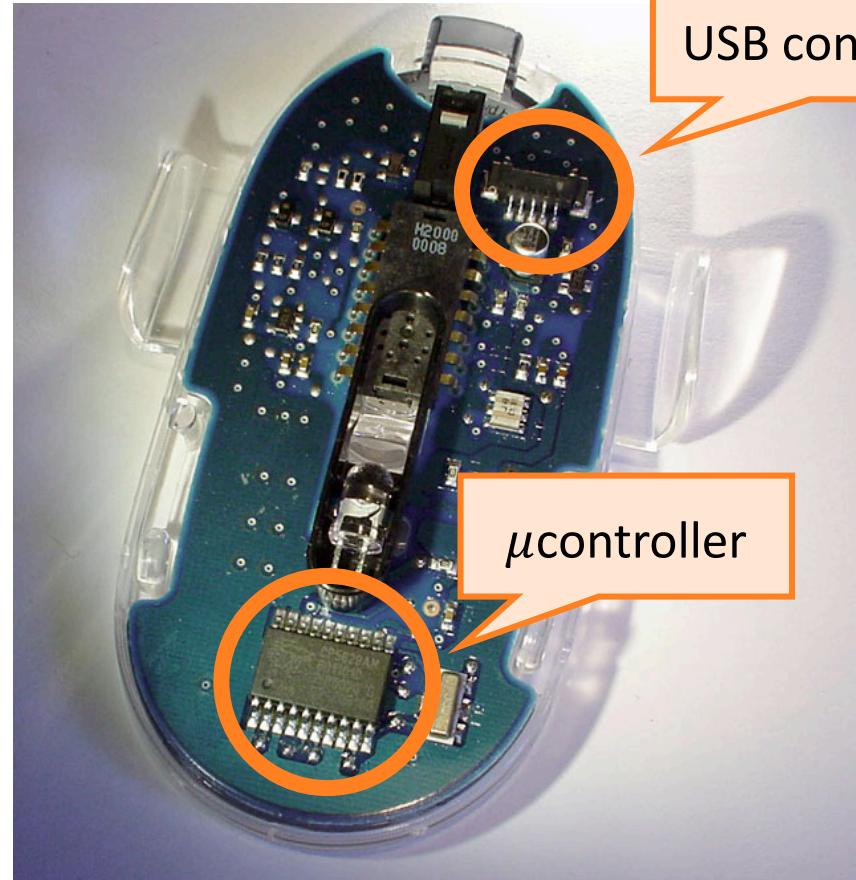
- Canonical I/O device consists of interface (device registers) and hidden internals



Example: Mouse

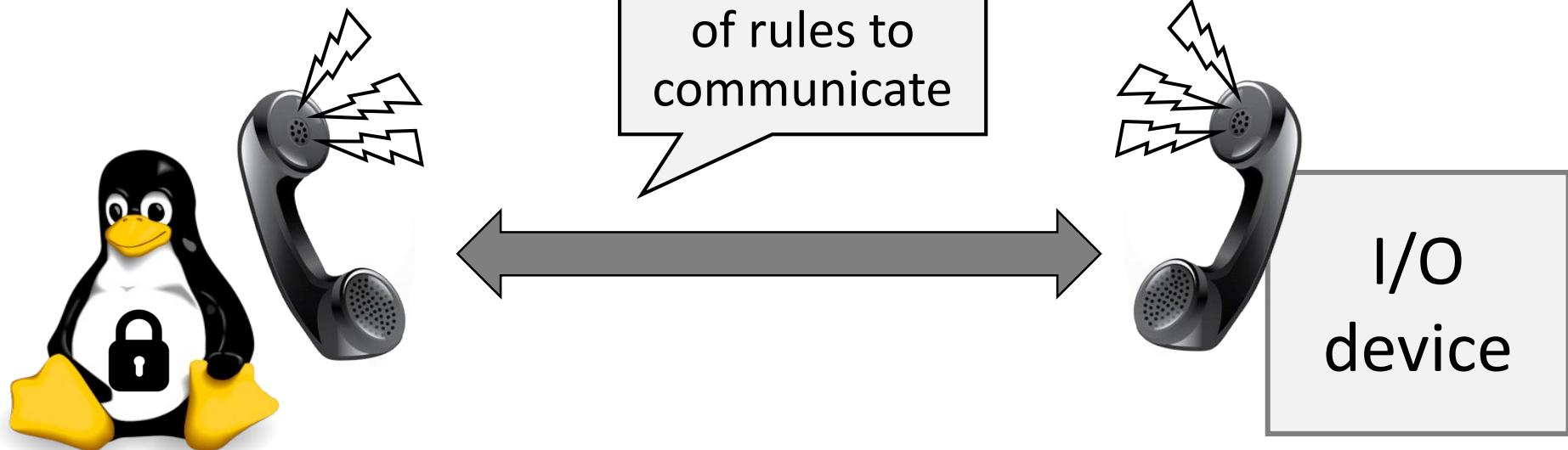


OS can control the mouse over a set of device registers exposed by its internals (e.g., status, data, and control)



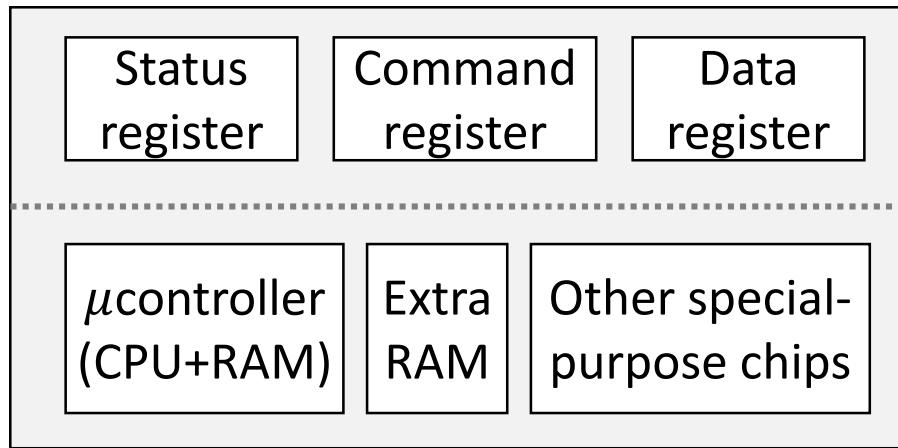
Apple Pro Mouse

Protocol





Example of Write Protocol

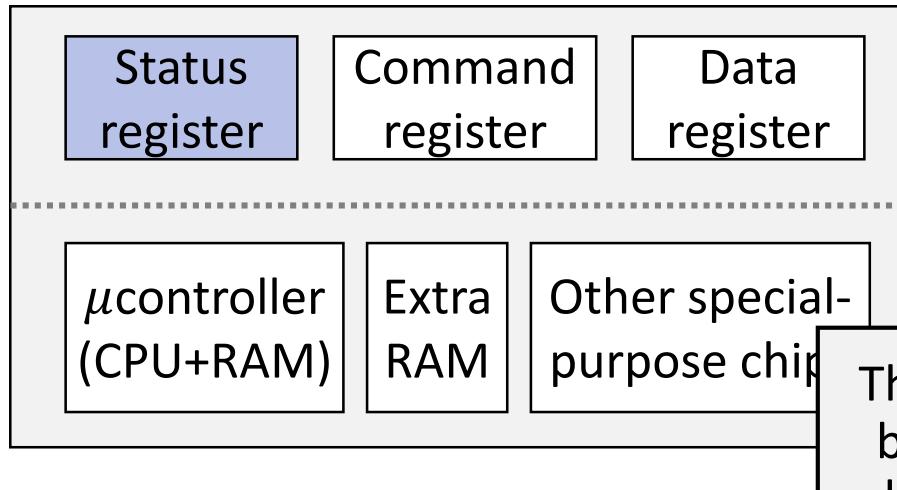


```
/* Write protocol */
```

Let's check how the OS can communicate with an I/O device (e.g., write)



Example of Write Protocol

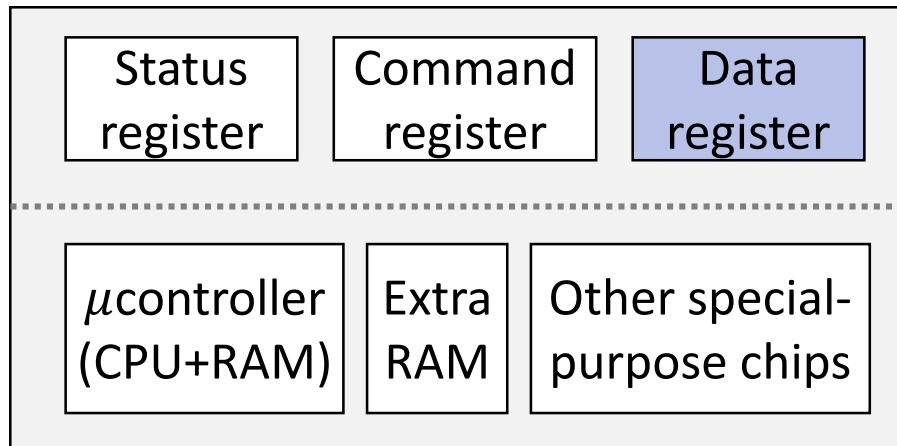


```
/* Write protocol */  
while (STATUS == BUSY)  
    // wait until device is not busy
```

The status register has busy bit, which show whether the underlying device is whether busy to handle something or not. OS can issue a request only if the target device is free (not busy)



Example of Write Protocol

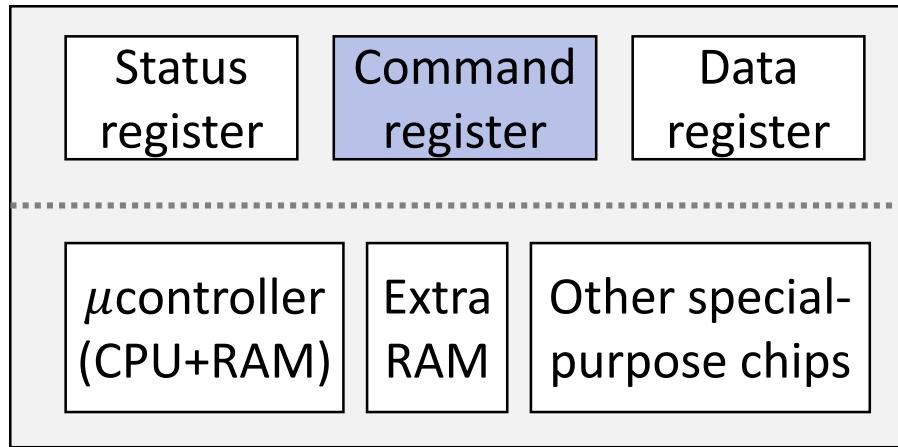


```
/* Write protocol */  
while (STATUS == BUSY)  
    // wait until device is  
Write data to DATA register
```

OS transfers data by
writing the data register



Example of Write Protocol

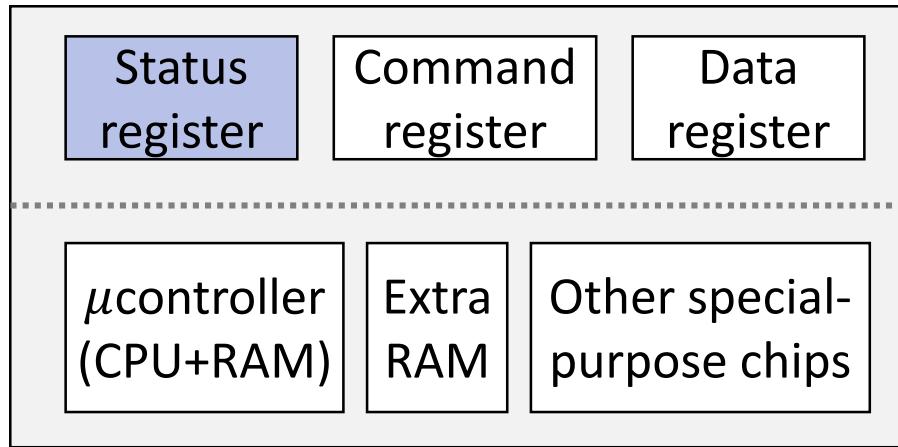


```
/* Write protocol */  
while (STATUS == BUSY)  
    // wait until device is not busy  
Write data to DATA register  
Write command to COMMAND register  
    // doing so starts the device and executes the command
```

Device starts to do something when the command register is set



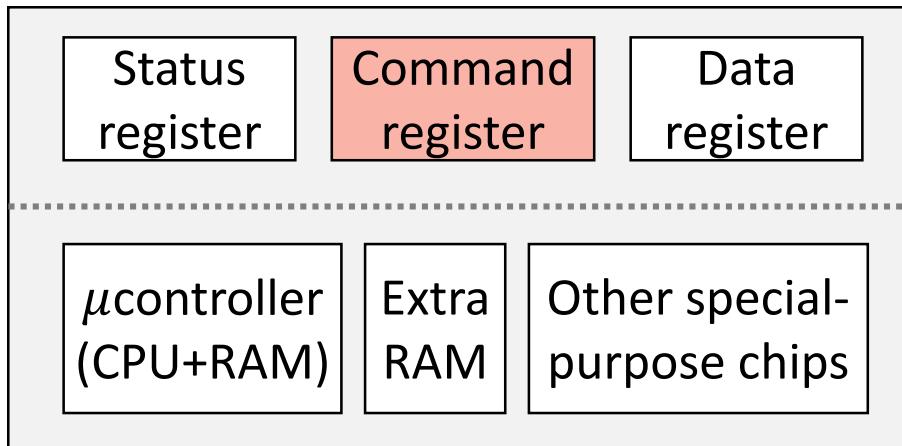
Example of Write Protocol



```
/* Write protocol */  
while (STATUS == BUSY)  
    // wait until device is done with your request  
    Write data to DATA register  
    Write command to COMMAND register  
    // doing so starts the process of executing the command  
    while(STATUS == BUSY)  
        // wait until device is done with your request
```

Device maintains busy bit as 1 (busy) for the service of such request

Device Control



```
/* Write protocol */
while (STATUS == BUSY)
    // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    // doing so starts the device and executes the command
while(STATUS == BUSY)
    // wait until device is done with your request
```



How can the OS manipulate the device registers?



I/O Ports

- I/O ports are configured by a set of the registers (each being associated with a distinct I/O controller)

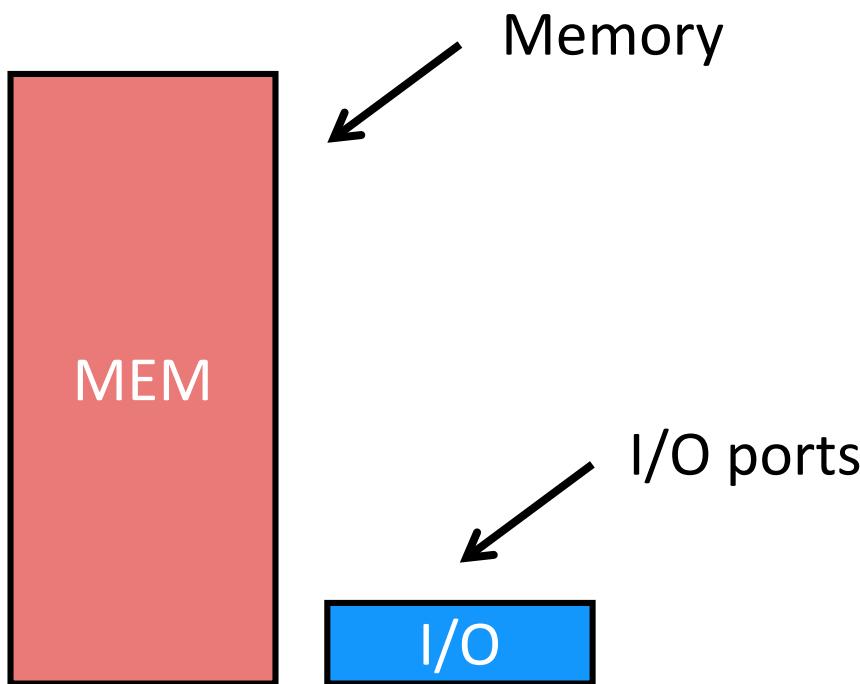
I/O address range (hexadecimal)	Device
000 – 00F	DMA controller
020 – 021	Interrupt controller
040 – 043	Timer
200 – 20F	Game controller
2F8 – 2FF	Serial port (secondary)
320 – 32F	Hard-disk controller
378 – 37F	Parallel port
3D0 – 3DF	Graphics controller
3F0 – 3F7	Diskette-drive controller
3F8 – 3FF	Serial port (primary)

Each device (device controller) has its own port address



Two Kinds of Address Mapping

#1: I/O address space (Two address spaces)



I/O ports are separated from the memory address space, and they can be accessed by using special I/O instructions



Special Instructions (for I/O)

- x86 provides 64KB of isolated I/O address space
- x86 provides two instructions to access I/O ports

- `in` instruction is used to read data from an I/O port
 - `in <register>, <port address>`
- `out` instruction is used to write data to an I/O port
 - `out <port address>, <register>`



Special Instructions (for I/O)

- x86 provides 64KB of isolated I/O address space
- x86 provides two instructions to access I/O ports

- `in` instruction is used to read data from an I/O port
 - `in <register>, <port address>`
- `out` instruction is used to write data to an I/O port
 - `out <port address>, <register>`



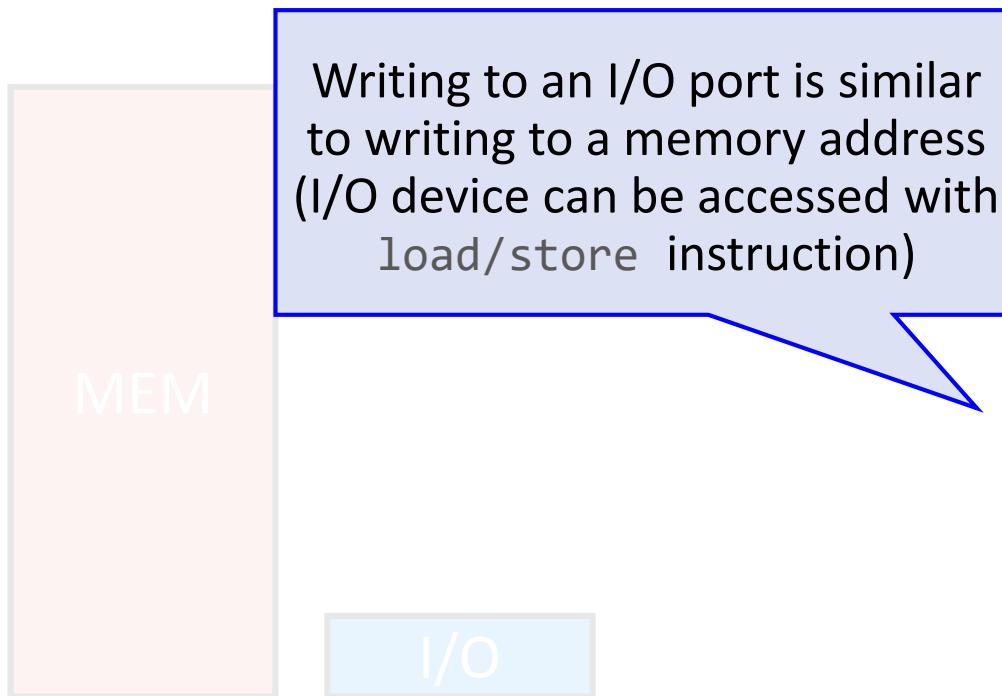
However, `in/out` instructions are **slow and clunky**

- Only allows 2^{16} different port numbers
- Instruction format restricts what portions of the registers you can use (register must be AL, AX, or EAX)

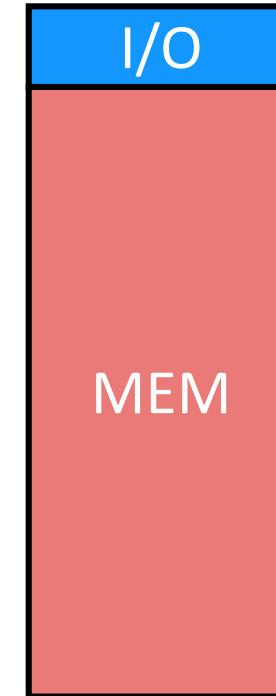


Two Kinds of Address Mapping

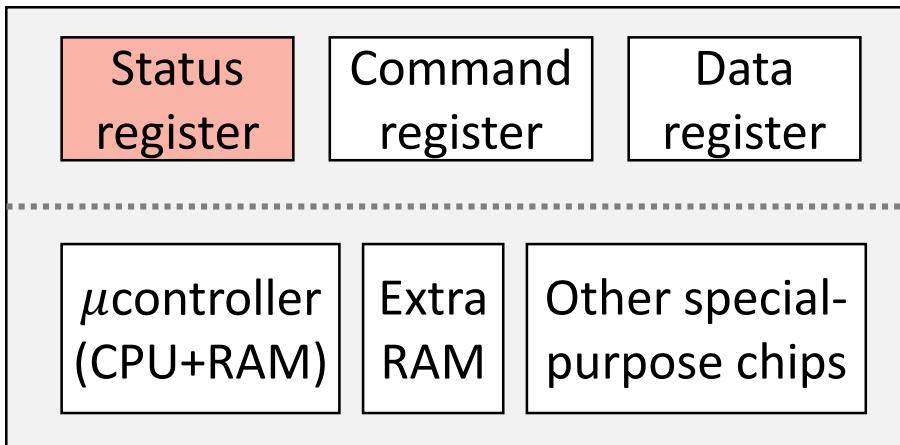
#1: I/O address space
(Two address spaces)



#2: Memory-mapped I/O
(One address space)



Status Checking



```
/* Write protocol */  
while (STATUS == BUSY)  
    // wait until device is not busy  
Write data to DATA register  
Write command to COMMAND register  
    // doing so starts the device and executes the command  
while(STATUS == BUSY)  
    // wait until device is done with your request
```

How can OS check the device's status?

A yellow thinking emoji with a hand on its chin is positioned near the red speech bubble, pointing towards it.



Polling and Interrupts



Polling

Imagine a phone **without a bell**. You would have to periodically answer the phone to see if anyone is there

Interrupt

Phone **with a bell**. You can do something else and stop and answer the phone when it rings

#1: Polling

- OS waits until the device is ready by repeatedly reading out the status register

Polling is simple, but it wastes CPU time to just wait for the device; Not that switching to another ready process is better to utilize the CPU

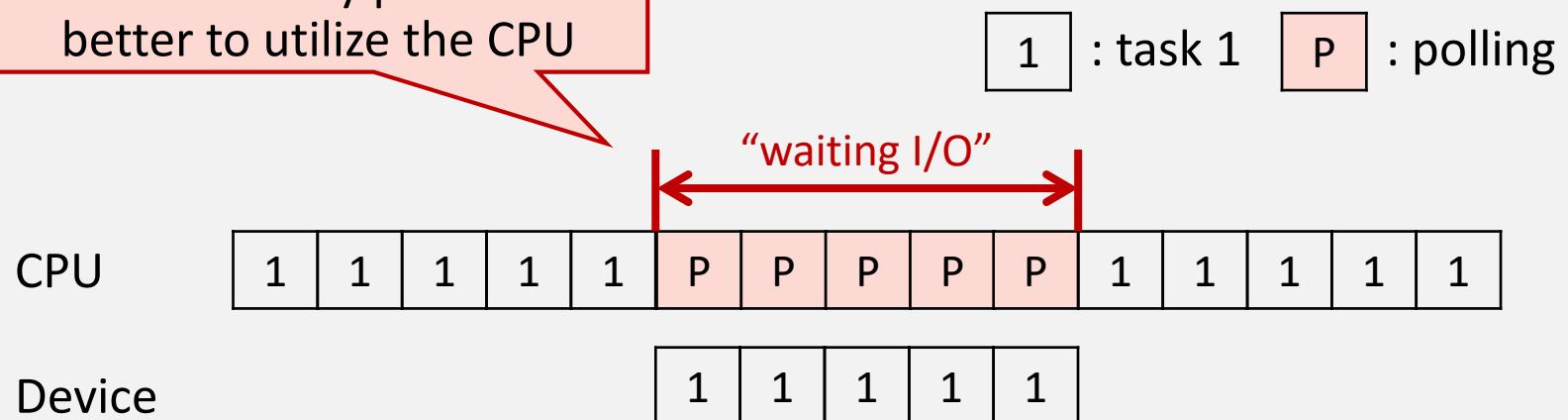


Diagram of CPU utilization by polling

#2: Interrupts

- Interrupt allow to CPU and the device are properly utilized

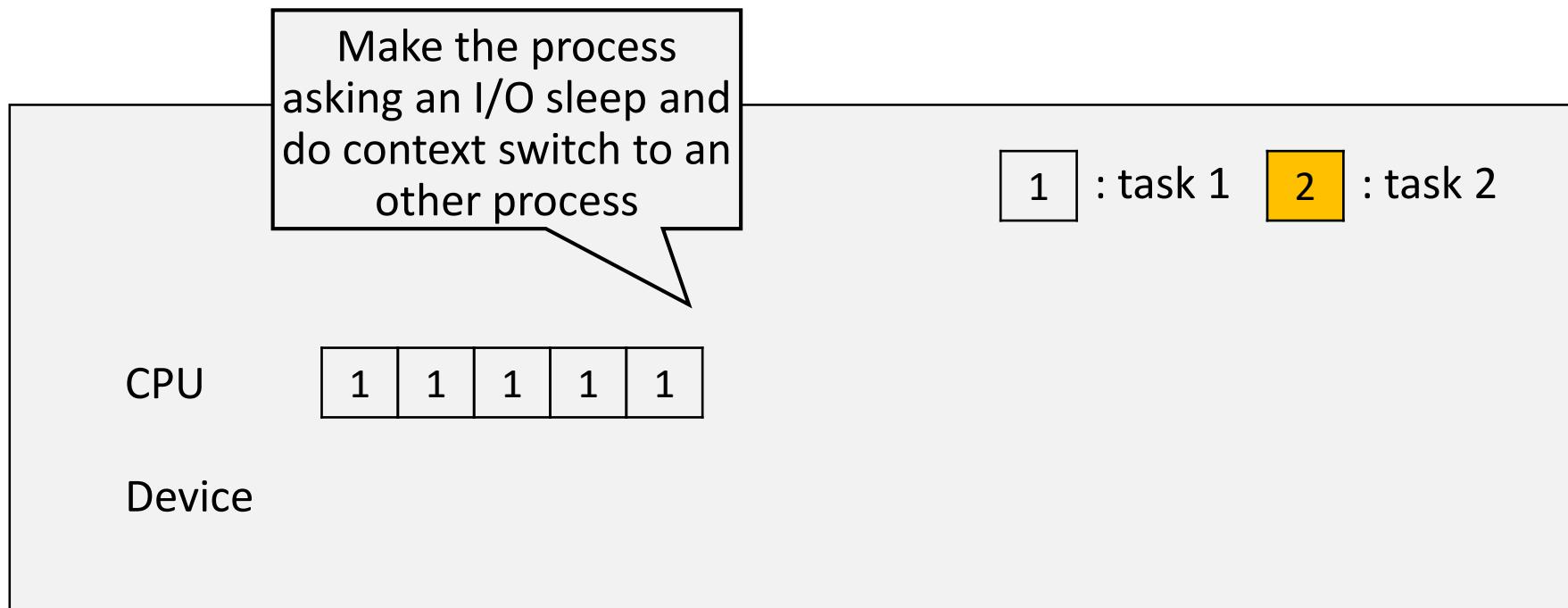


Diagram of CPU utilization by interrupt

#2: Interrupts

- Interrupt allow to CPU and the device are properly utilized

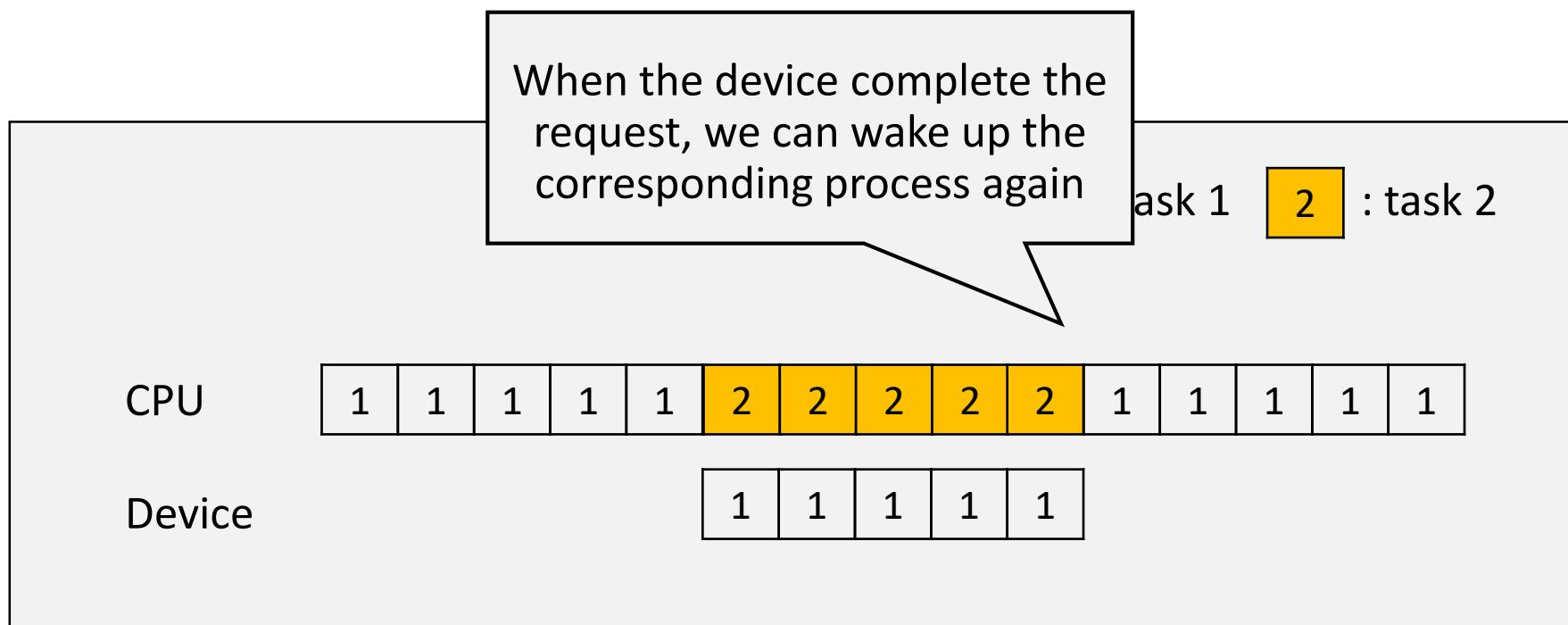


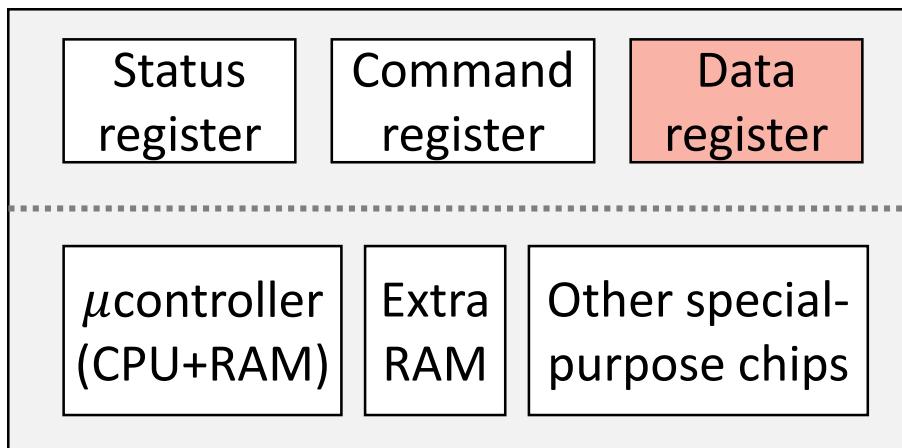
Diagram of CPU utilization by interrupt

Polling vs. Interrupt

- “Interrupt is not always the best solution”
 - If the underlying device performs I/O services very quickly, interrupt will “slow down” the target system



Data Transfer



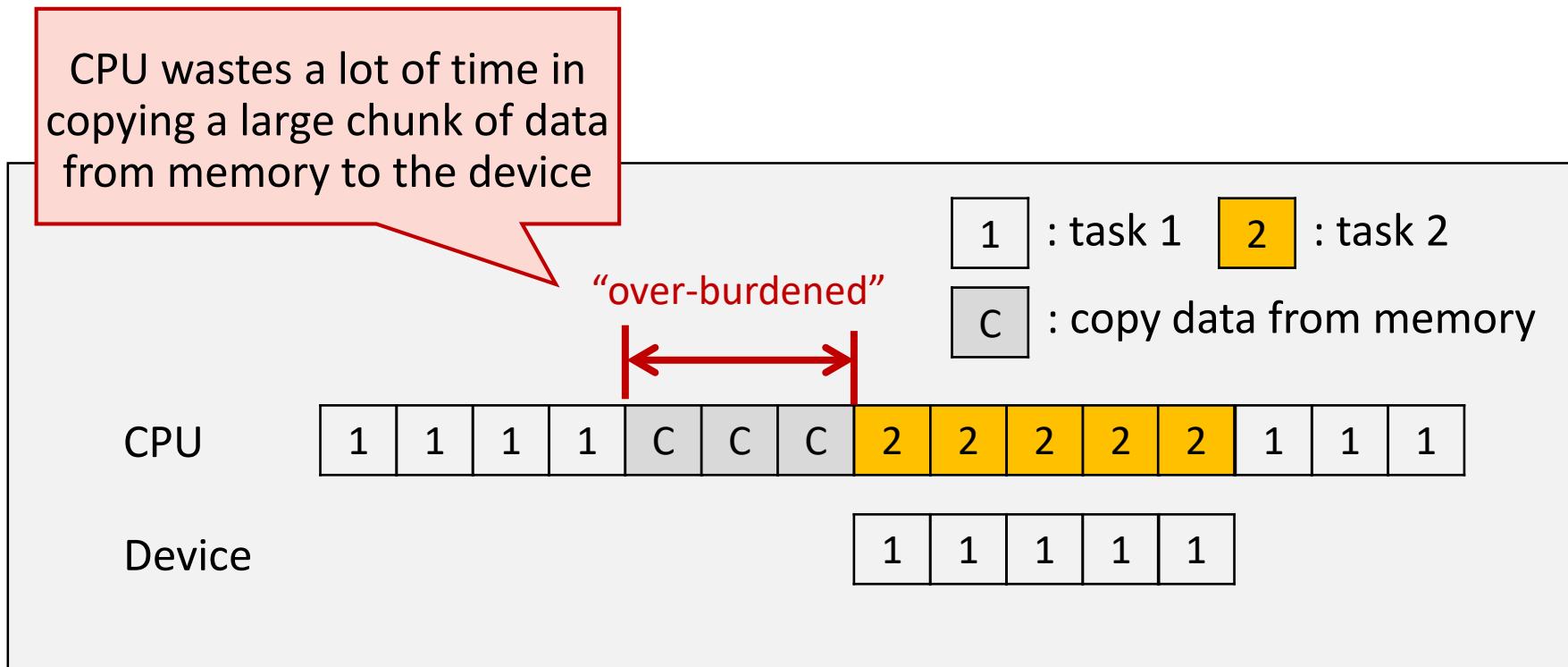
```
/* Write protocol */
while (STATUS == BUSY)
    // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    // doing so starts the device and executes the command
while(STATUS == BUSY)
    // wait until device is done with your request
```



How can OS send
data to I/O
device?

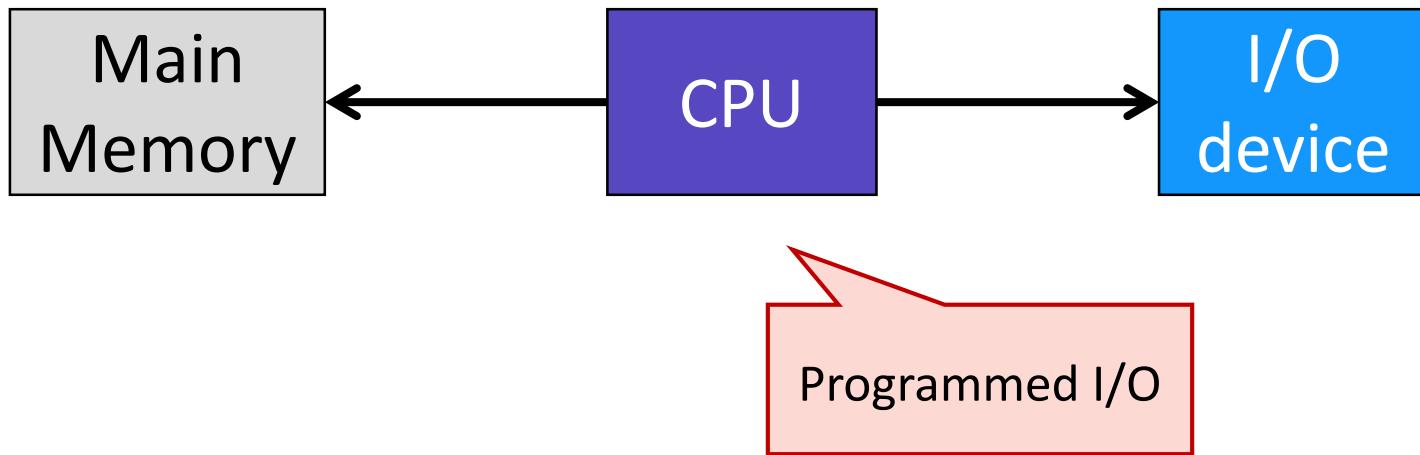
#1: Programmed I/O (PIO)

- Programmed I/O operations are the result of I/O instructions (in/out) written in computer program



#2: Direct Memory Access (DMA)

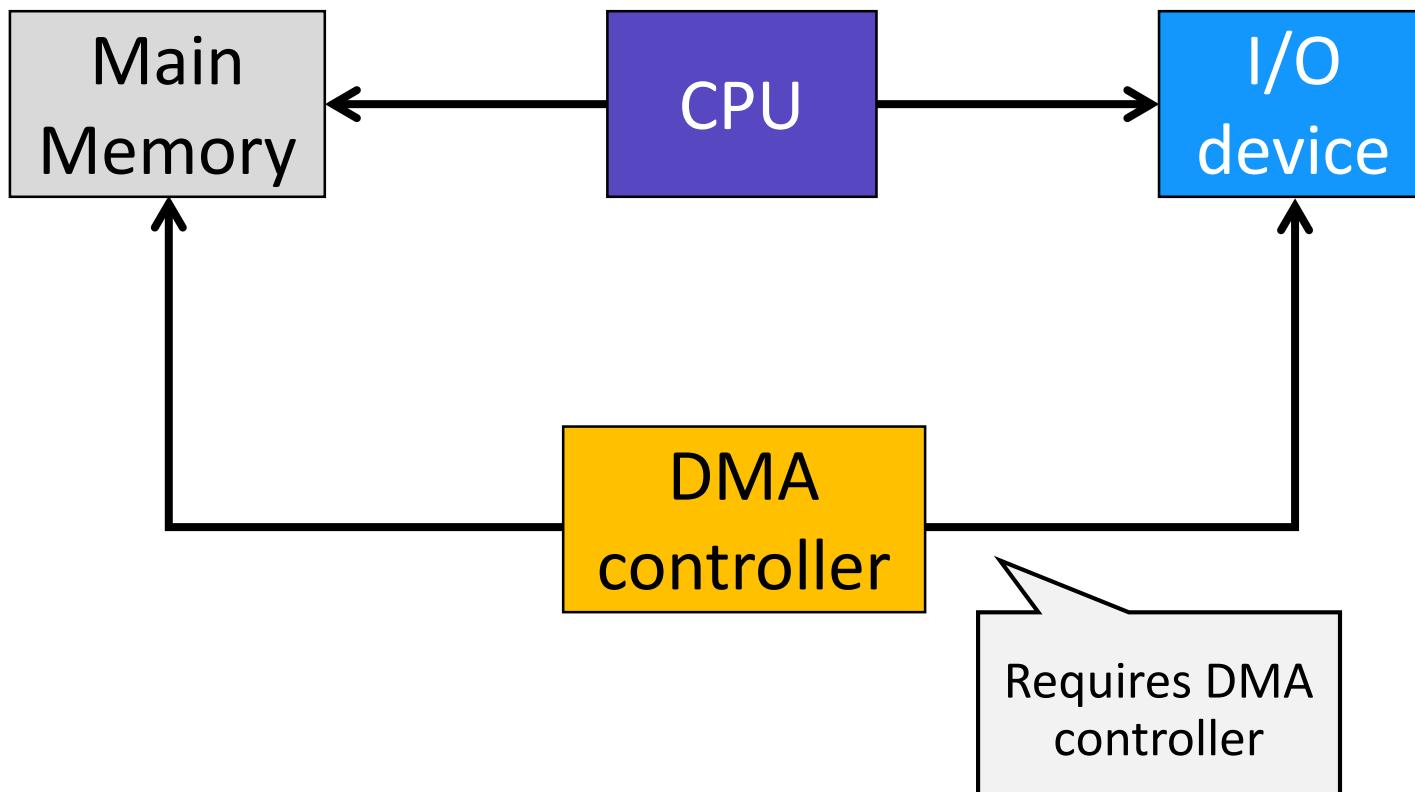
- **Key idea:** only use CPU to transfer control requests, not data





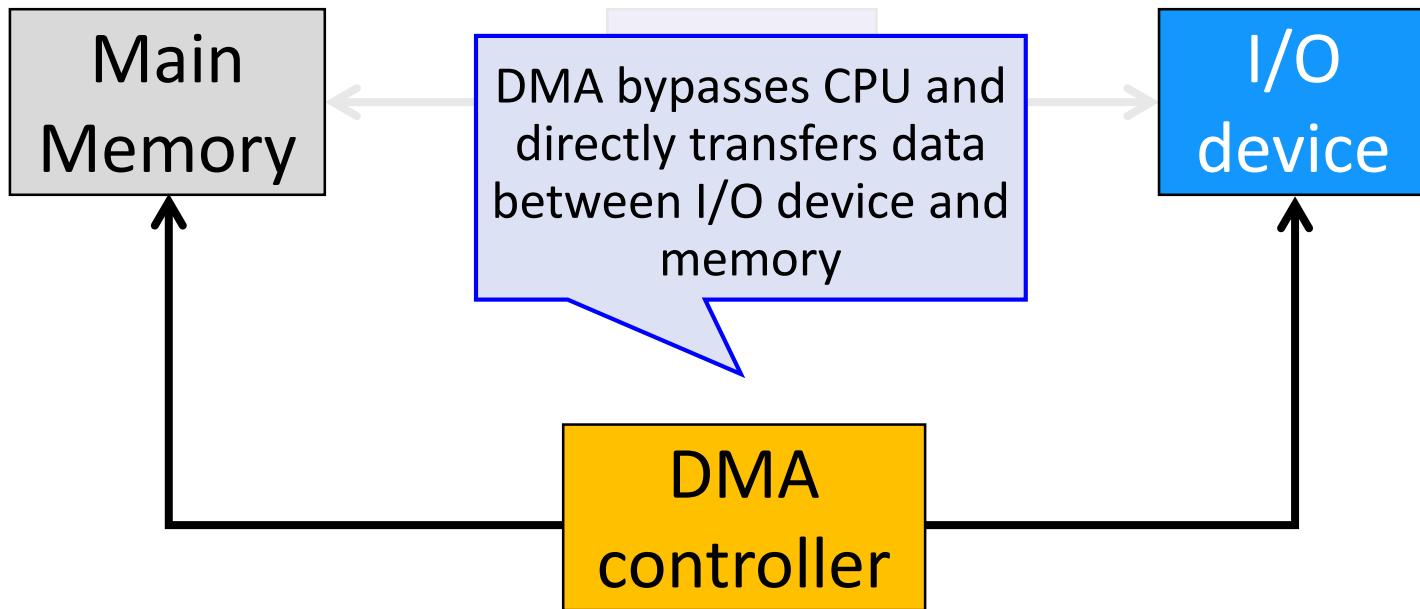
#2: Direct Memory Access (DMA)

- **Key idea:** only use CPU to transfer control requests, not data



#2: Direct Memory Access (DMA)

- **Key idea:** only use CPU to transfer control requests, not data



#2: Direct Memory Access (DMA)

- **Key idea:** only use CPU to transfer control requests, not data

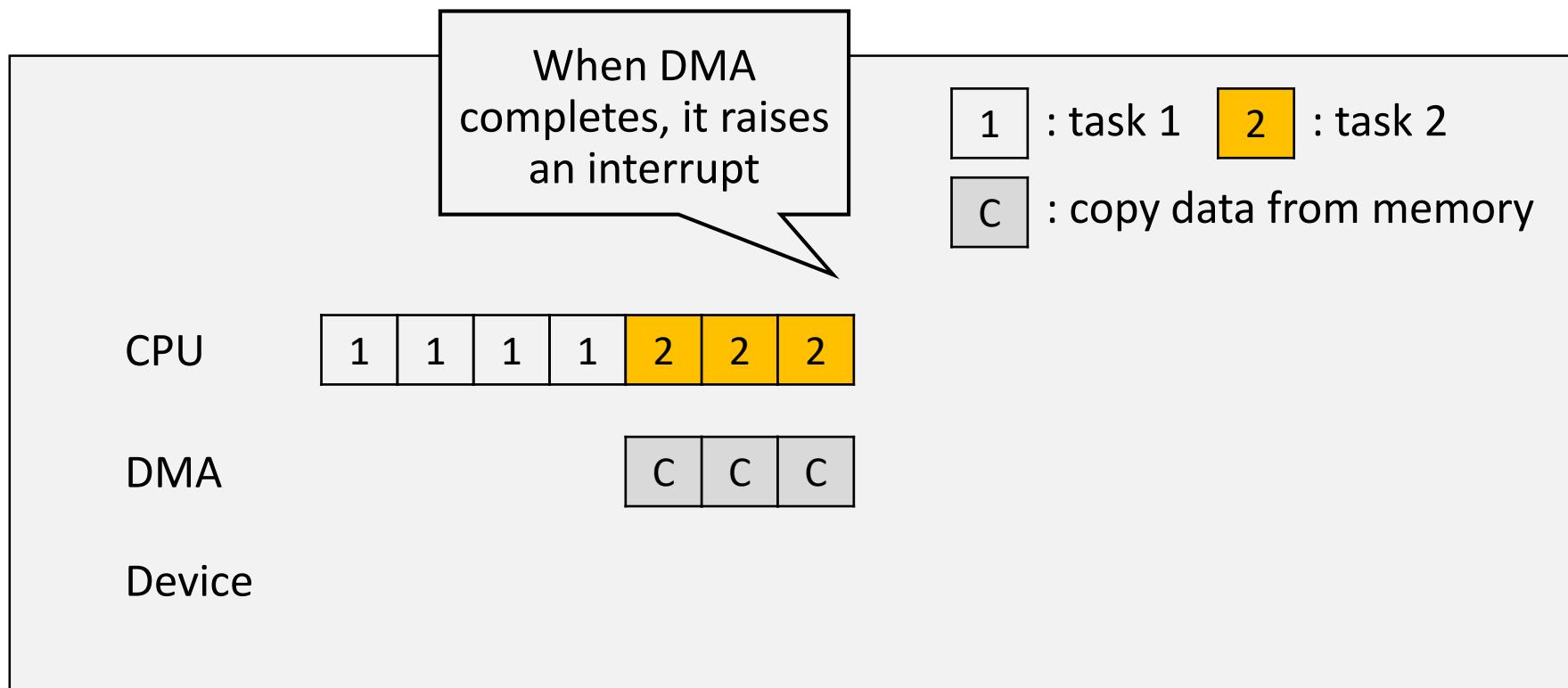


Diagram of CPU utilization by DMA

#2: Direct Memory Access (DMA)

- **Key idea:** only use CPU to transfer control requests, not data

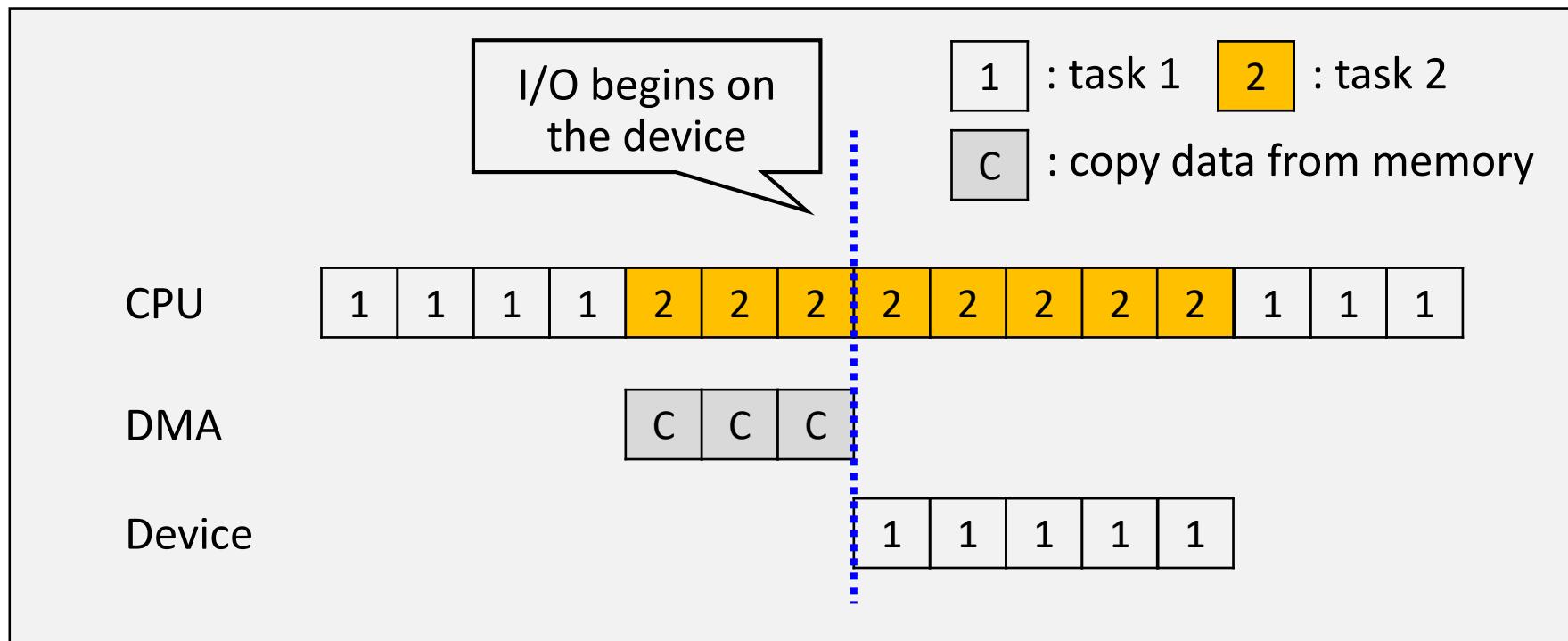
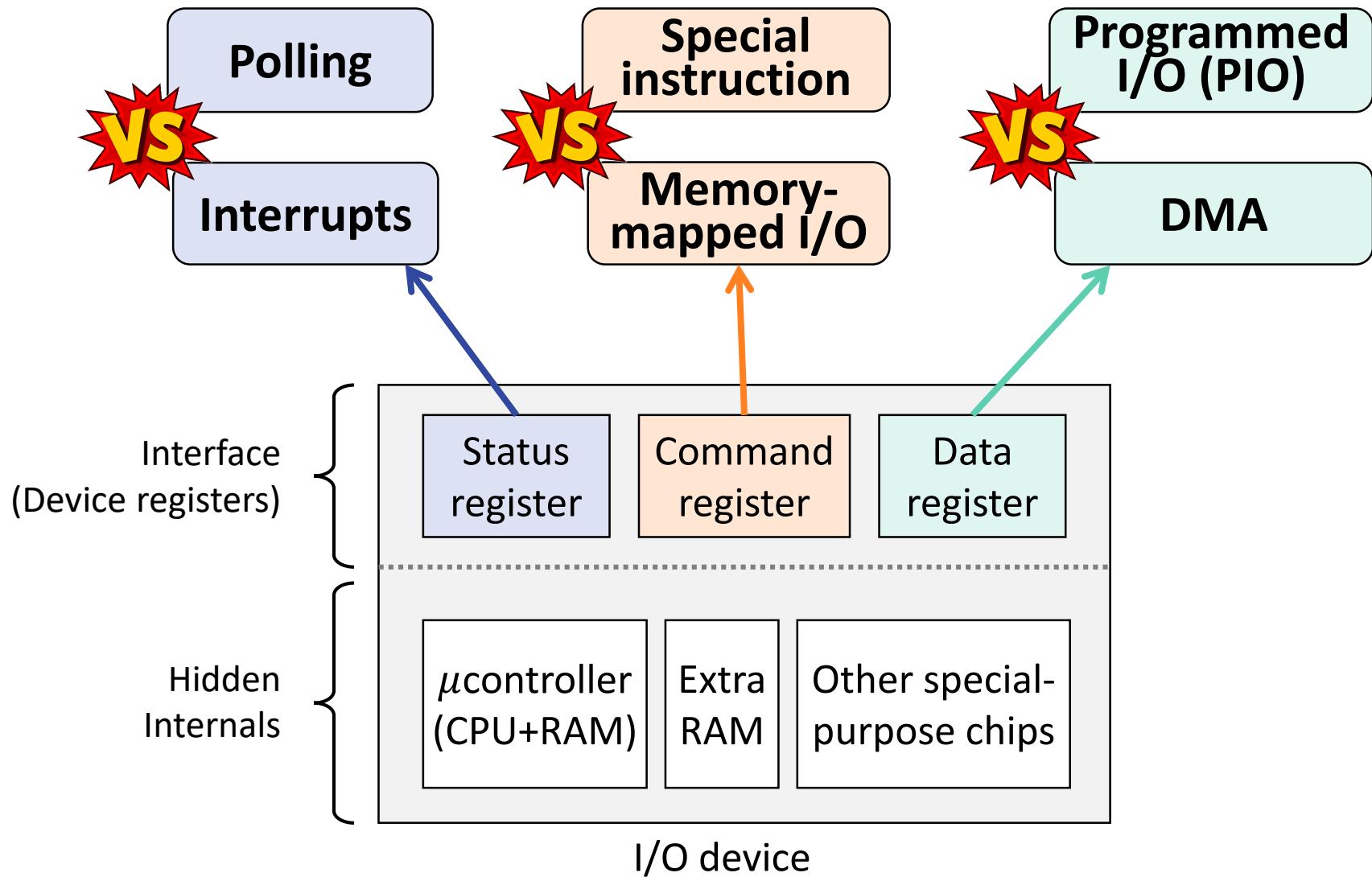


Diagram of CPU utilization by DMA

Protocol Variants

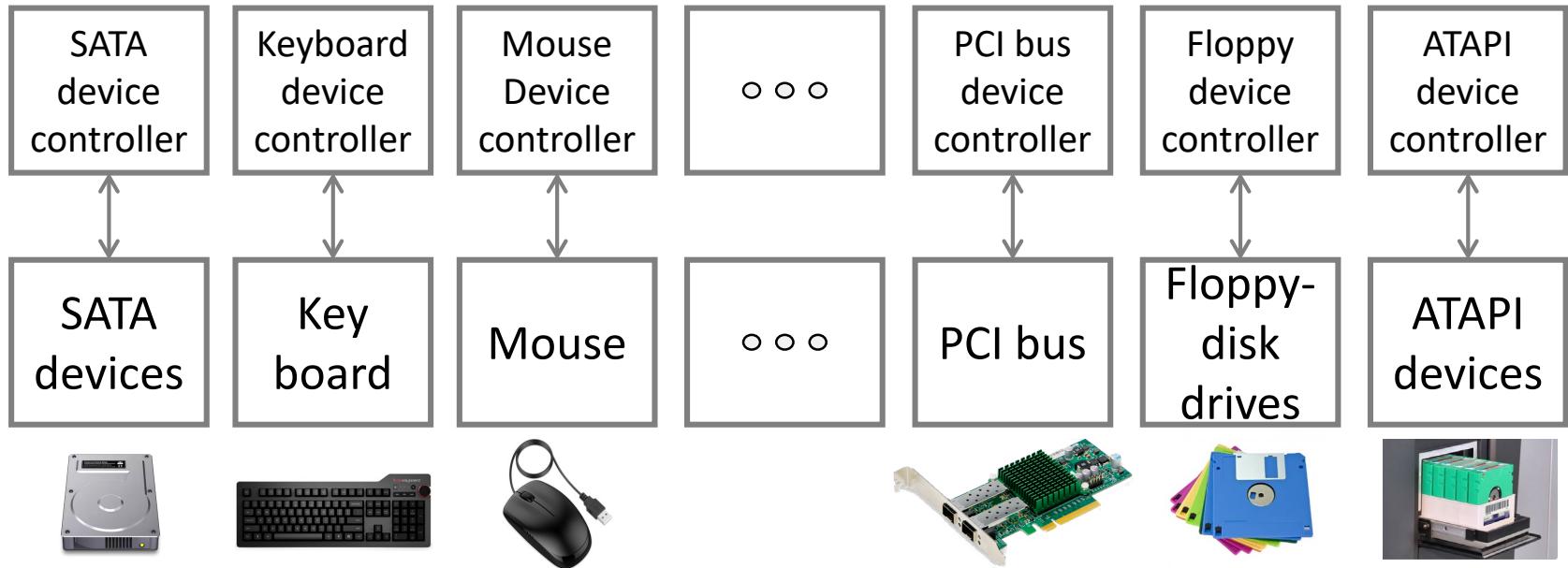




Variety Is a Challenge

There are many, many I/O devices and each has its own protocol

Hardware

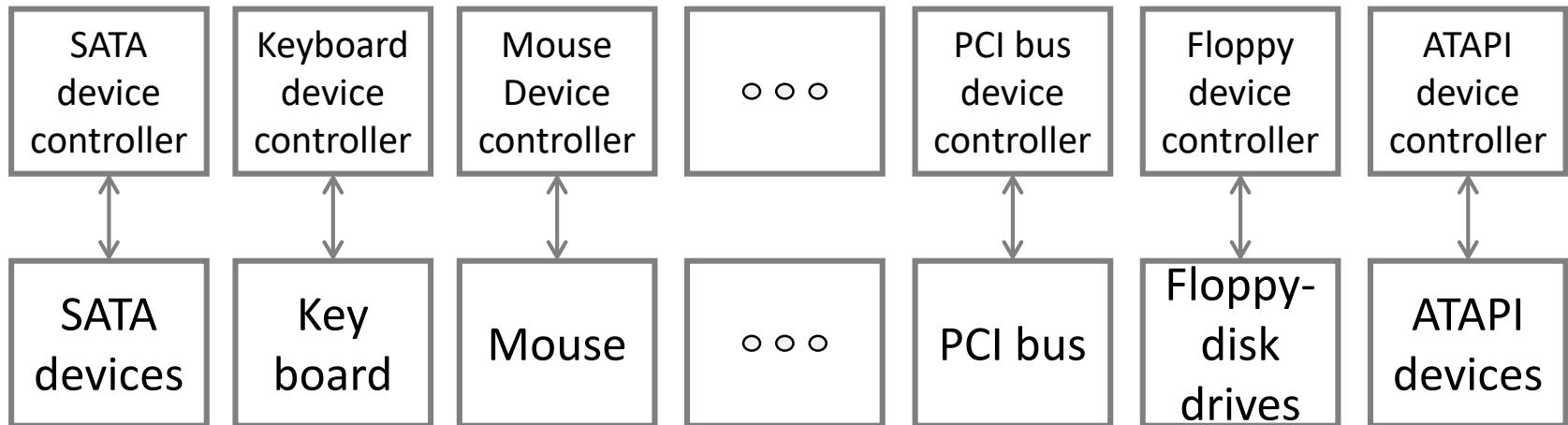


Variety Is a Challenge



Are we going to write up
and modify programs for
each HW combinations?

Hardware





Solution: Abstraction

Software (Kernel)

SATA
device
driver

Keyboard
device
driver

Mouse
Device
driver

o o o

PCI bus
device
driver

Floppy
device
driver

ATAPI
device
driver

Hardware

SATA
device
controller

Keyboard
device
controller

Mouse
Device
controller

o o o

PCI bus
device
controller

Floppy
device
controller

ATAPI
device
controller

Um... write a device driver for
each device (drivers account for
70% of Linux source code)





Solution: Abstraction

Software (Kernel)



Abstraction layer

But, build a common interface atop diverse drivers



SATA
device
driver

Keyboard
device
driver

Mouse
Device
driver

o o o

PCI bus
device
driver

Floppy
device
driver

ATAPI
device
driver

Hardware

SATA
device
controller

Keyboard
device
controller

Mouse
Device
controller

o o o

PCI bus
device
controller

Floppy
device
controller

ATAPI
device
controller



Example: Storage Stack

There are multiple HDD devices based on interface type (ATA, SATA, SCSI)

HW

Disk (Hard drive)

ATA device

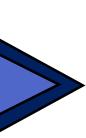


SATA device



SCSI device





Example: Storage Stack

KERNEL

Device Driver

ATA driver

SATA driver

SCSI driver

HW

Disk (Hard drive)

ATA device



SATA device



SCSI device



Thus, there are per-device (per-interface) device driver



Example: Storage Stack

USER

Application

KERNEL

Device Driver

ATA driver

SATA driver

SCSI driver

HW

Disk (Hard drive)

ATA device

SATA device

SCSI device



Don't worry. Users can access the underlying disks without the detailed knowledge of each device type

Example: Storage Stack

USER

Application

There is file system abstraction (and generic block abstraction) for storage device

File System

KERNEL

Generic Block Layer

Device Driver

ATA driver

SATA driver

SCSI driver

HW

Disk (Hard drive)

ATA device

SATA device

SCSI device



Example: Storage Stack

USER

Application

POSIX API [open, read, write, close, etc]

File System

KERNEL

Generic Block Layer

Device Driver

ATA driver

SATA driver

SCSI driver

HW

Disk (Hard drive)

ATA device

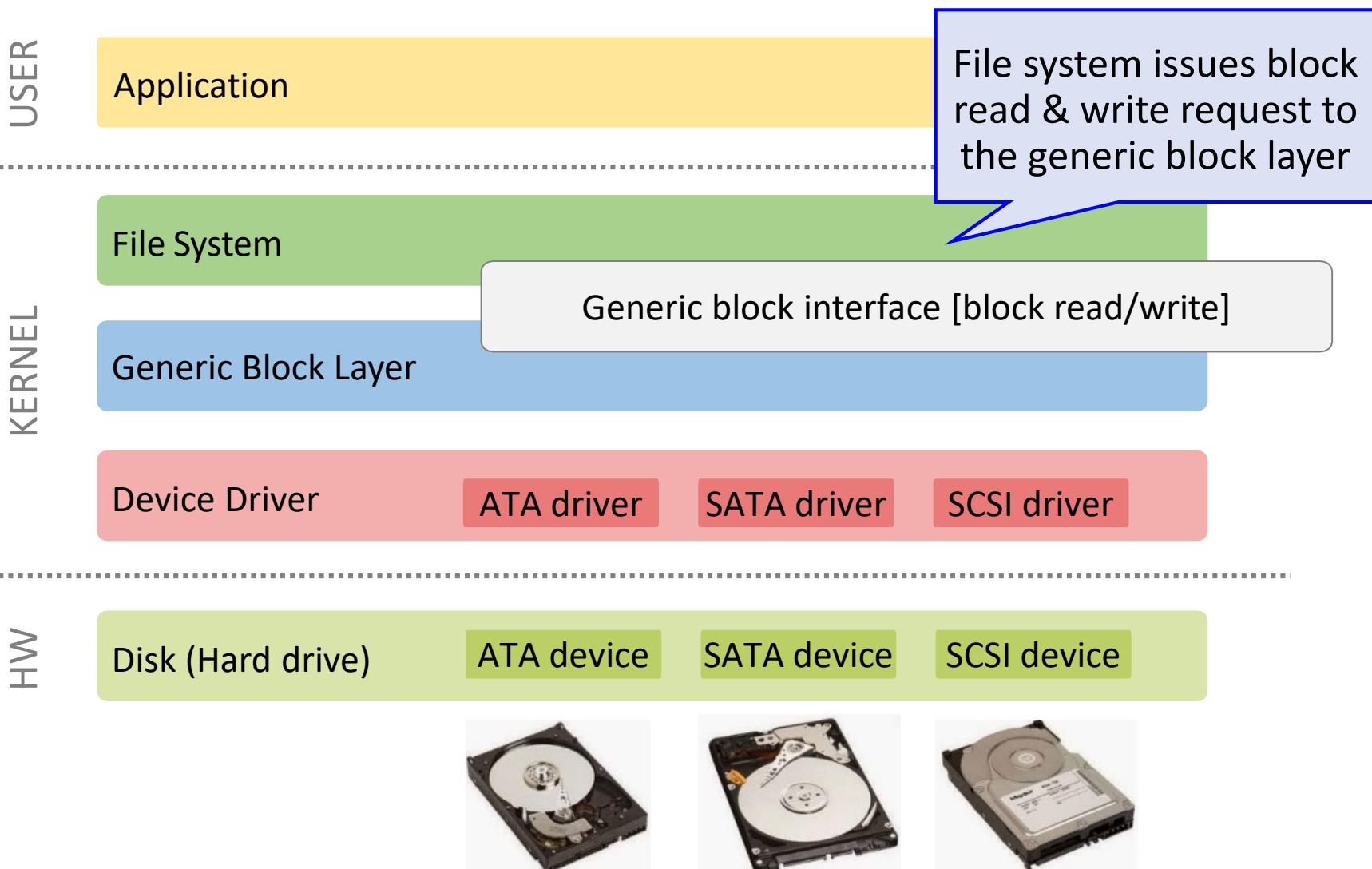
SATA device

SCSI device



File system exposes common API to user application

Example: Storage Stack



Example: Storage Stack

USER

Application

File System

KERNEL

Generic Block Layer

Then block layer issues
the specific request to
corresponding device
driver

Specific block interface [protocol-specific read/write]

Device Driver

ATA driver

SATA driver

SCSI driver

HW

Disk (Hard drive)

ATA device

SATA device

SCSI device





Example: Storage Stack

USER

Application

KERNEL

Generic Block Layer

Device Driver

ATA driver

SATA driver

HW

Disk (Hard drive)

ATA device

SATA device

SCSI device



Let's go over the details
of storage device first

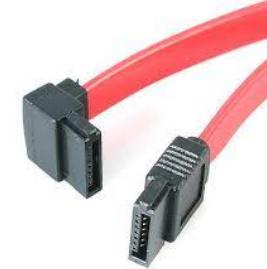
Devices: Magnetic Disks



Common Disk Interfaces

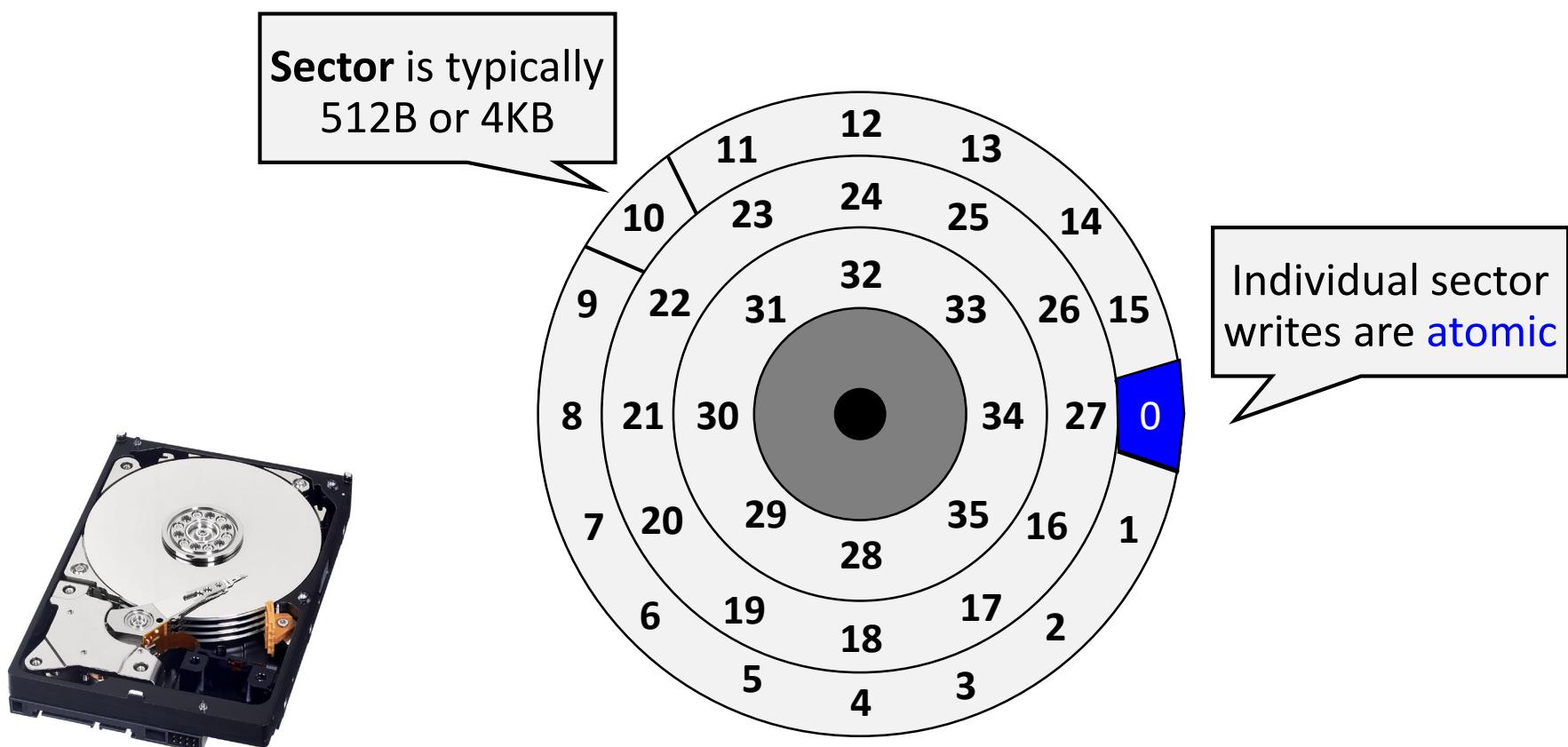
Commands (read/write) and logical block addresses (LBA) are placed in device registers

ST-506 → ATA → IDE → SATA → SCSI



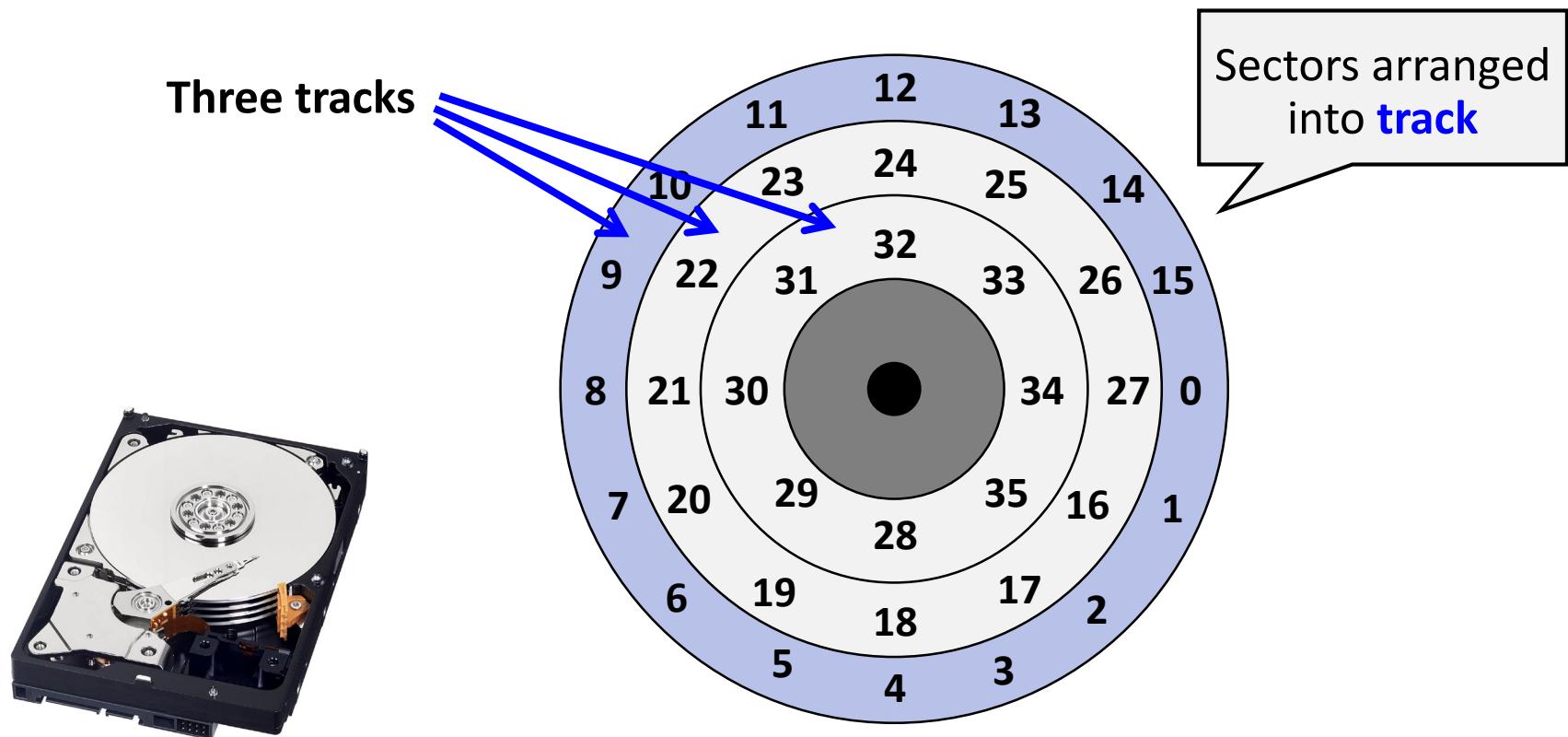
Addressing and Geometry

- Hard drives expose a large number of **sectors** (blocks)



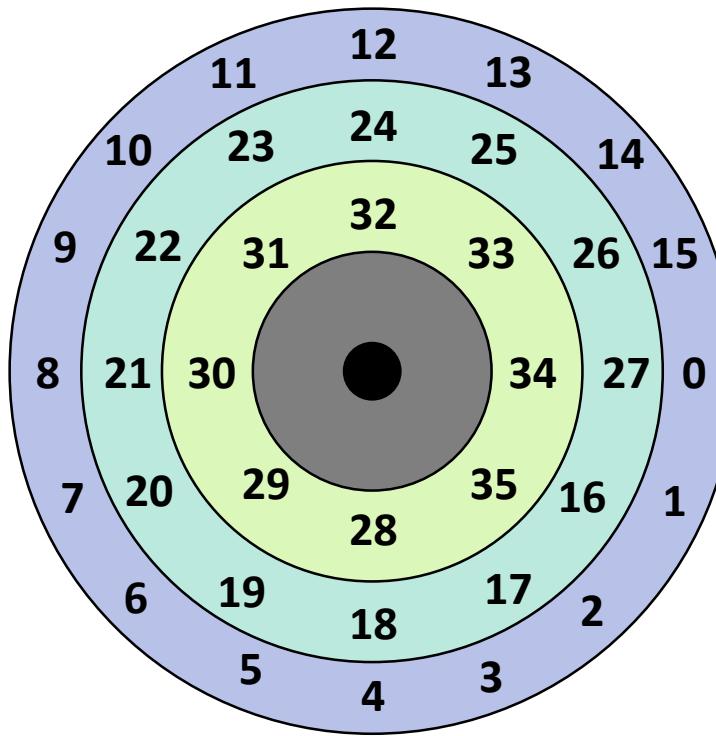
Addressing and Geometry

- Hard drives expose a large number of **sectors** (blocks)



Addressing and Geometry

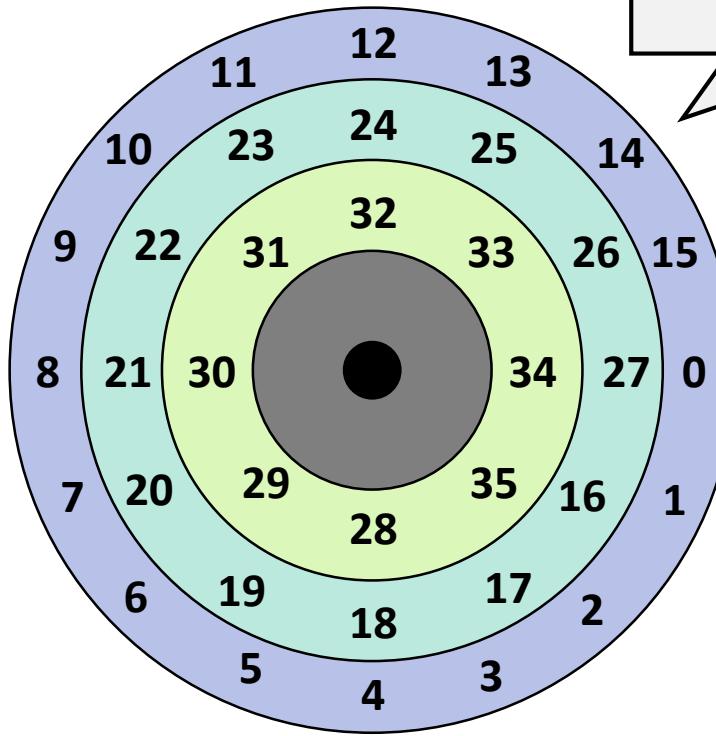
- Hard drives expose a large number of **sectors** (blocks)



Tracks are arranged in concentric circles on platters

Addressing and Geometry

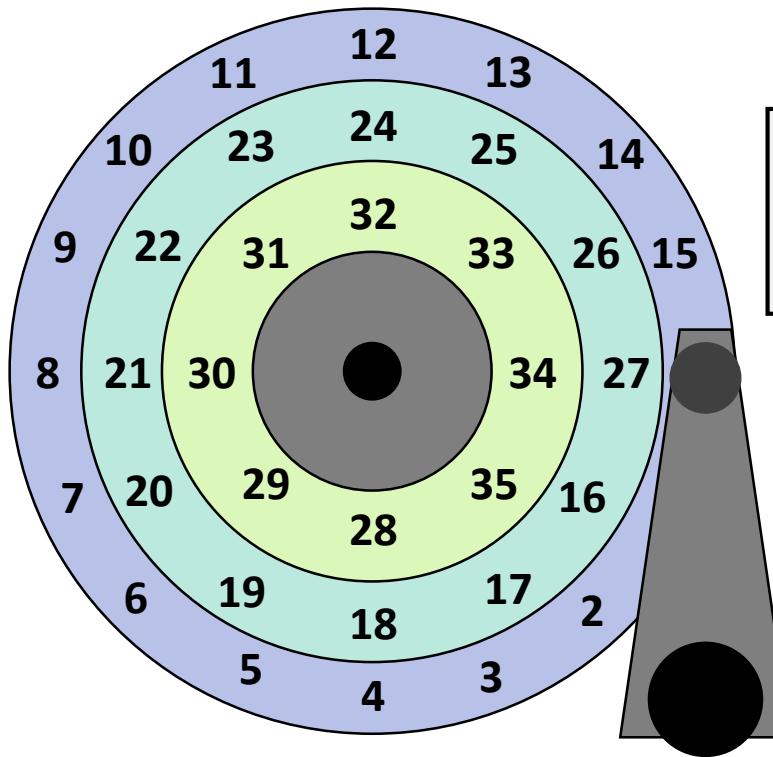
- Hard drives expose a large number of **sectors** (blocks)



Drive motor spins the platters at a constant rate (measured in revolutions per minute, RPM)

Addressing and Geometry

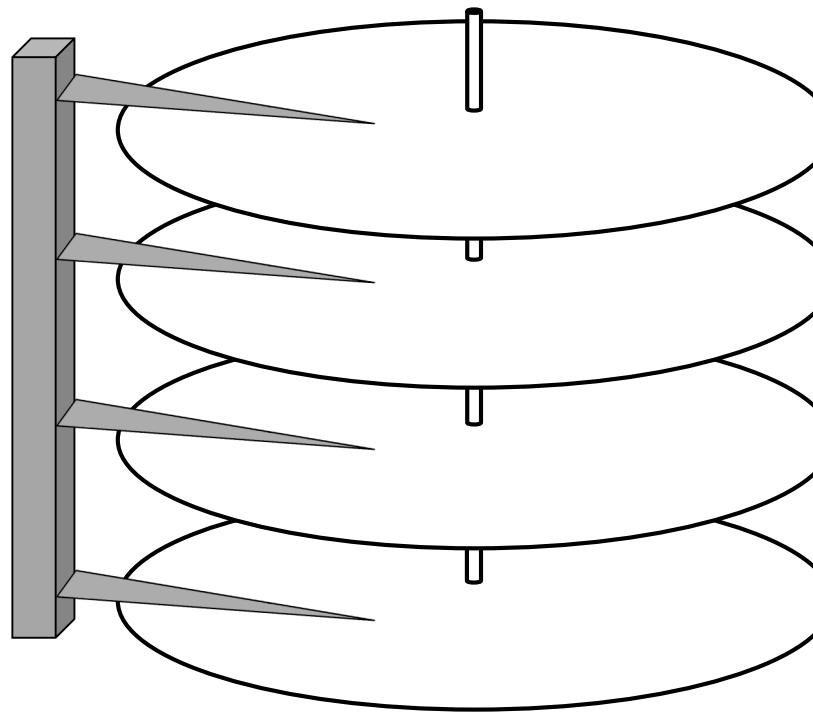
- Hard drives expose a large number of **sectors** (blocks)



Head seeks data across the various tracks

Addressing and Geometry

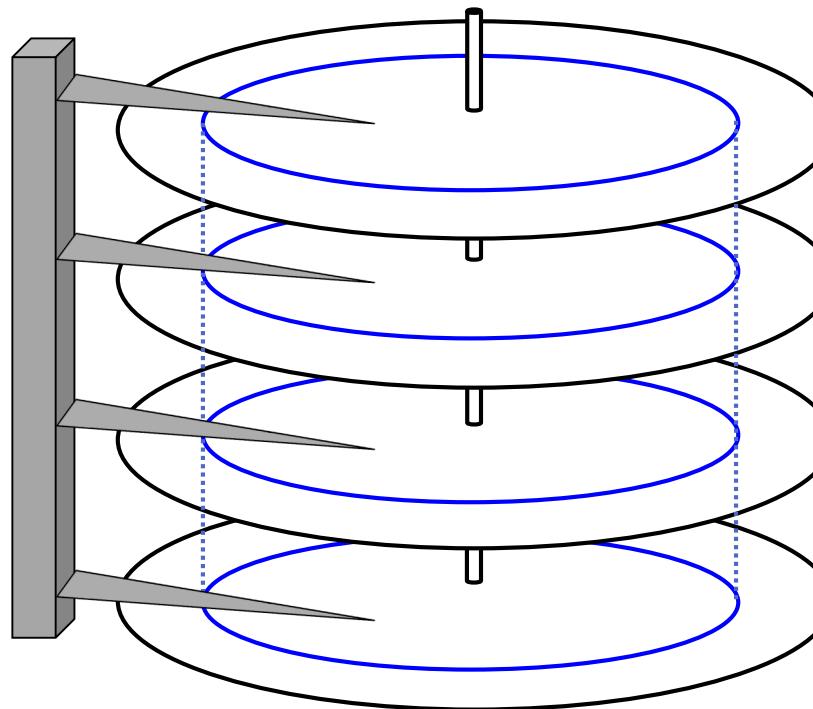
- Hard drives expose a large number of **sectors** (blocks)



A disk may have multiple, double-sided platters

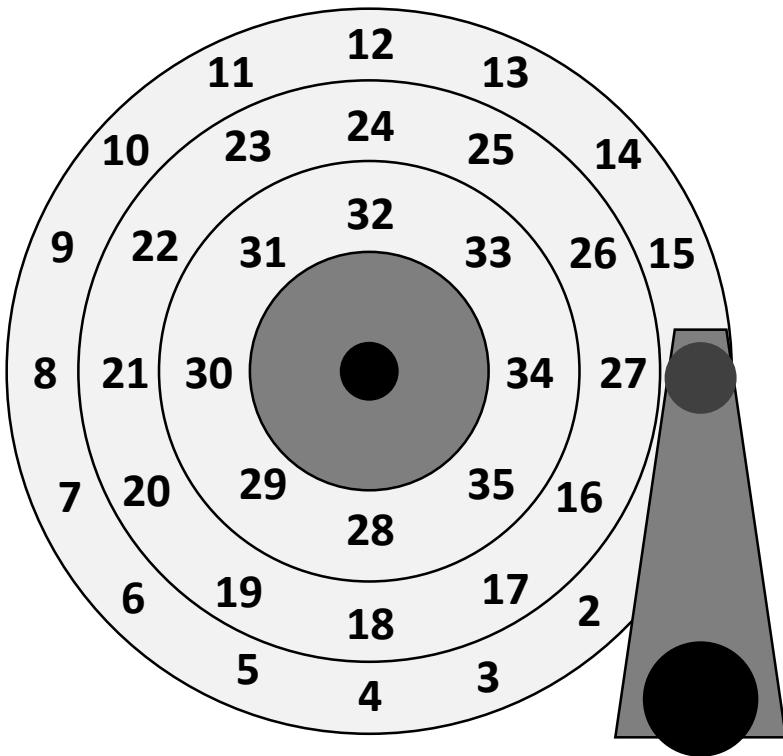
Addressing and Geometry

- Hard drives expose a large number of **sectors** (blocks)



A **cylinder** is a particular track on multiple platters

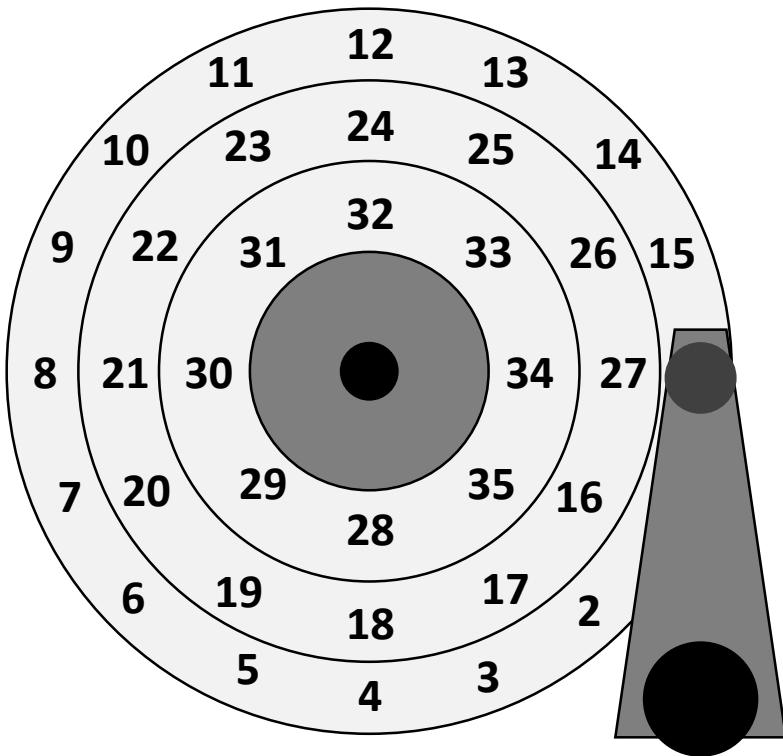
Types of Delay With Disks



Type #1: Rotational delay

- Time to rotate the desired sector to the read head
- Related to RPM

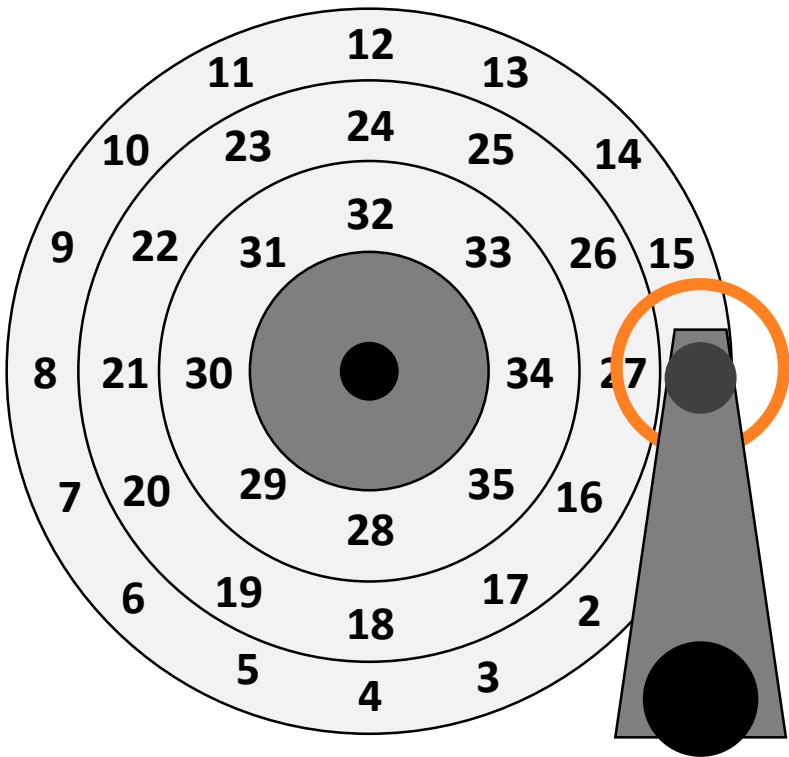
Types of Delay With Disks



Type #2: Seek delay

- Time to move the read head to a different track (seeking)

Types of Delay With Disks



Type #3: Transfer time

- Time to read or write bytes
(one sector)



Transfer Time Calculation



	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15000	7200
Avg. Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s

Transfer time

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

Cheetah

$$\begin{aligned} T_{I/O} &= 4 \text{ ms} + 1 / (15000 \text{ RPM} / 60 \text{ s/M} / 1000 \text{ ms/s}) / 2 \\ &\quad + (4096 \text{ B} / 125 \text{ MB/s} * 1000 \text{ ms/s} / 2^{20} \text{ MB/B}) \end{aligned}$$

$$T_{I/O} = 4 \text{ ms} + 2 \text{ ms} + 0.03125 \text{ ms} \approx 6 \text{ ms}$$

Barracuda

$$\begin{aligned} T_{I/O} &= 9 \text{ ms} + 1 / (7200 \text{ RPM} / 60 \text{ s/M} / 1000 \text{ ms/s}) / 2 \\ &\quad + (4096 \text{ B} / 105 \text{ MB/s} * 1000 \text{ ms/s} / 2^{20} \text{ MB/B}) \end{aligned}$$

$$T_{I/O} = 9 \text{ ms} + 4.17 \text{ ms} + 0.0372 \text{ ms} \approx 13.2 \text{ ms}$$



Transfer Time Calculation



	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15000	7200
Avg. Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s

Transfer time

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

Cheetah

$$T_{I/O} = 4 \text{ ms} + 1 / (15000 \text{ RPM} / 60 \text{ s/M} / 1000 \text{ ms/s}) / 2 \\ + (4096 \text{ B} / 125 \text{ MB/s} * 1000 \text{ ms/s} / 2^{20} \text{ MB/B})$$

→ 0.66MB/s

$$T_{I/O} = 4 \text{ ms} + 2 \text{ ms} + 0.03125 \text{ ms} \approx 6 \text{ ms}$$

Barracuda

$$T_{I/O} = 9 \text{ ms} + 1 / (7200 \text{ RPM} / 60 \text{ s/M} / 1000 \text{ ms/s}) / 2 \\ + (4096 \text{ B} / 105 \text{ MB/s} * 1000 \text{ ms/s} / 2^{20} \text{ MB/B})$$

→ 0.31MB/s

$$T_{I/O} = 9 \text{ ms} + 4.17 \text{ ms} + 0.0372 \text{ ms} \approx 13.2 \text{ ms}$$



Transfer Time Calculation

	Cheetah	Barracuda
Capacity	300 GB	7200 GB
RPM	15000	7200
Avg. Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s

This is maximum performance of sequential I/O

But, random I/O results in very poor disk performance

$$T_{I/O} = 4 \text{ ms} + 1 / (15000 \text{ RPM} / 60 \text{ s/M} / 1000 \text{ ms/s}) / 2 + (4096 \text{ B} / 125 \text{ MB/s} * 1000 \text{ ms/s} / 2^{20} \text{ MB/B})$$

0.66MB/s

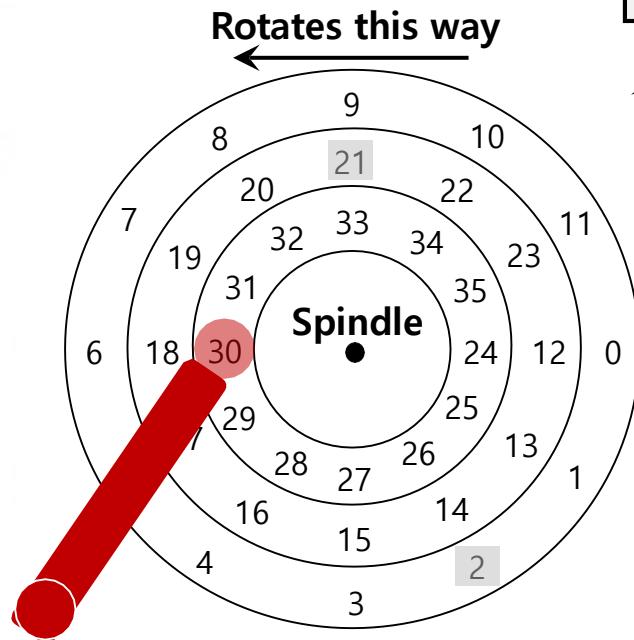
$$T_{I/O} = 4 \text{ ms} + 2 \text{ ms} + 0.03125 \text{ ms} \approx 6 \text{ ms}$$

$$T_{I/O} = 9 \text{ ms} + 1 / (7200 \text{ RPM} / 60 \text{ s/M} / 1000 \text{ ms/s}) / 2 + (4096 \text{ B} / 105 \text{ MB/s} * 1000 \text{ ms/s} / 2^{20} \text{ MB/B})$$

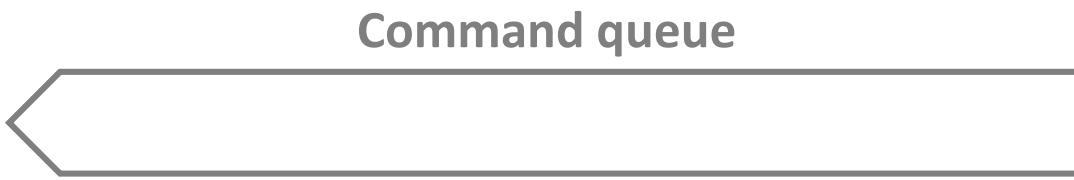
0.31MB/s

$$T_{I/O} = 9 \text{ ms} + 4.17 \text{ ms} + 0.0372 \text{ ms} \approx 13.2 \text{ ms}$$

Disk Head Scheduling

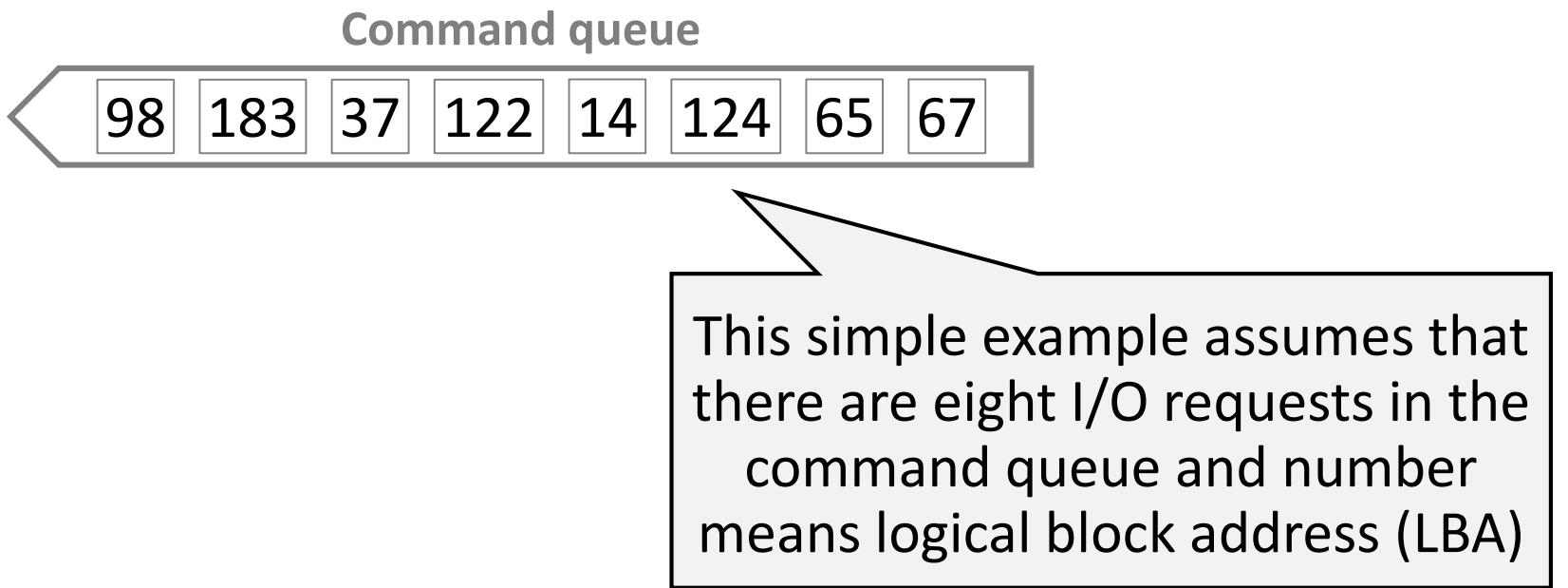


Simple Example

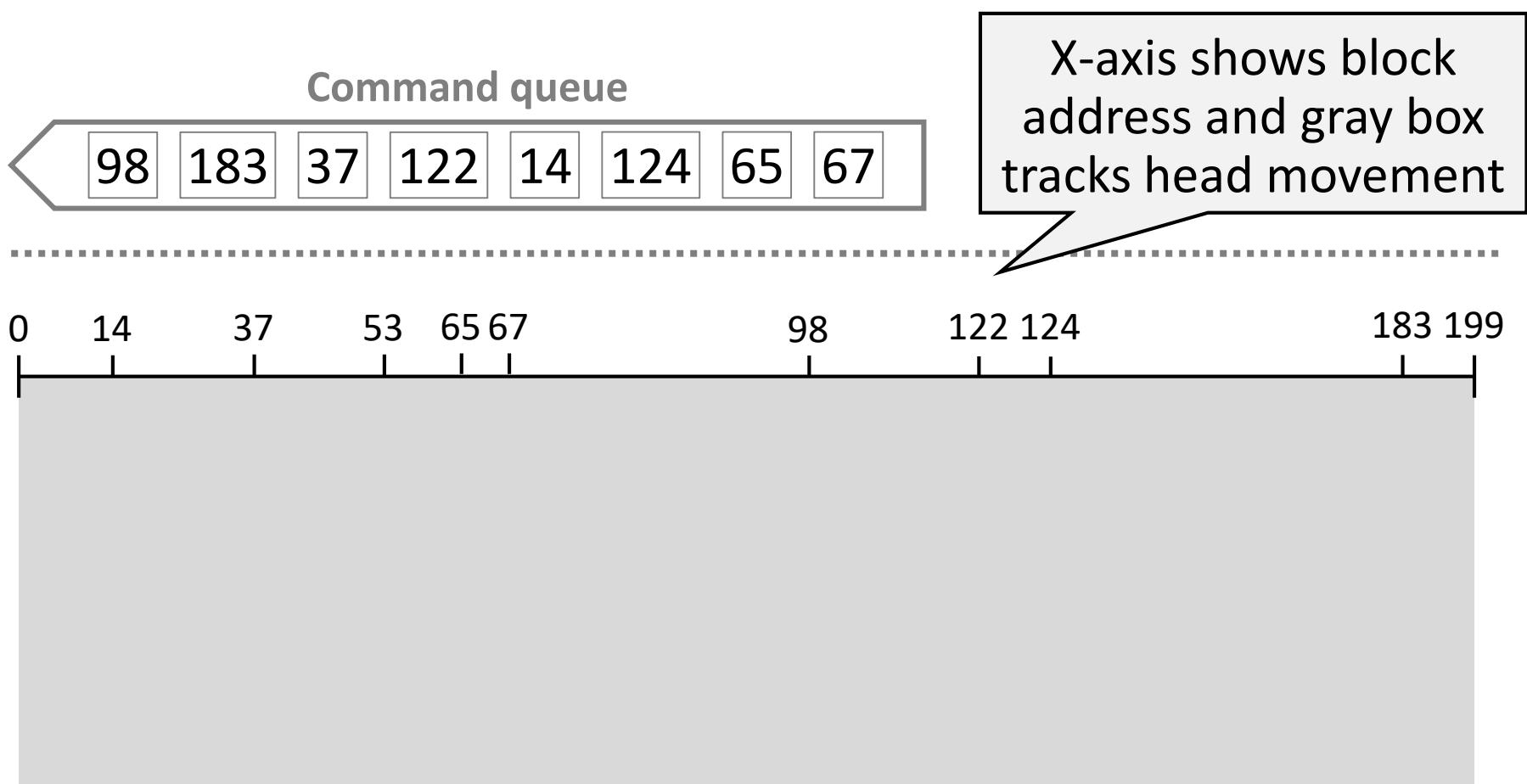


Key idea: If there is a queue of requests, the requests can be reordered to improve performance

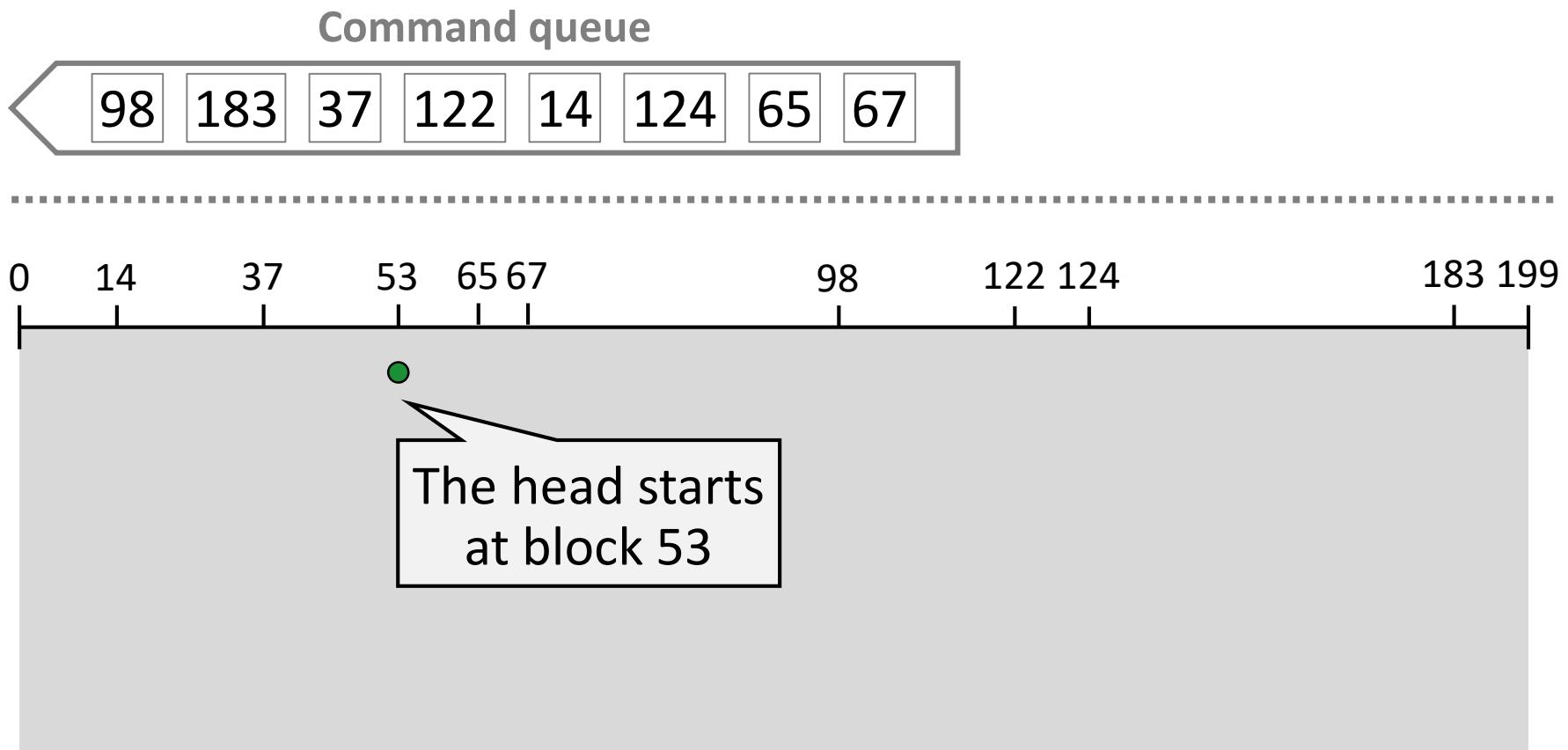
Simple Example



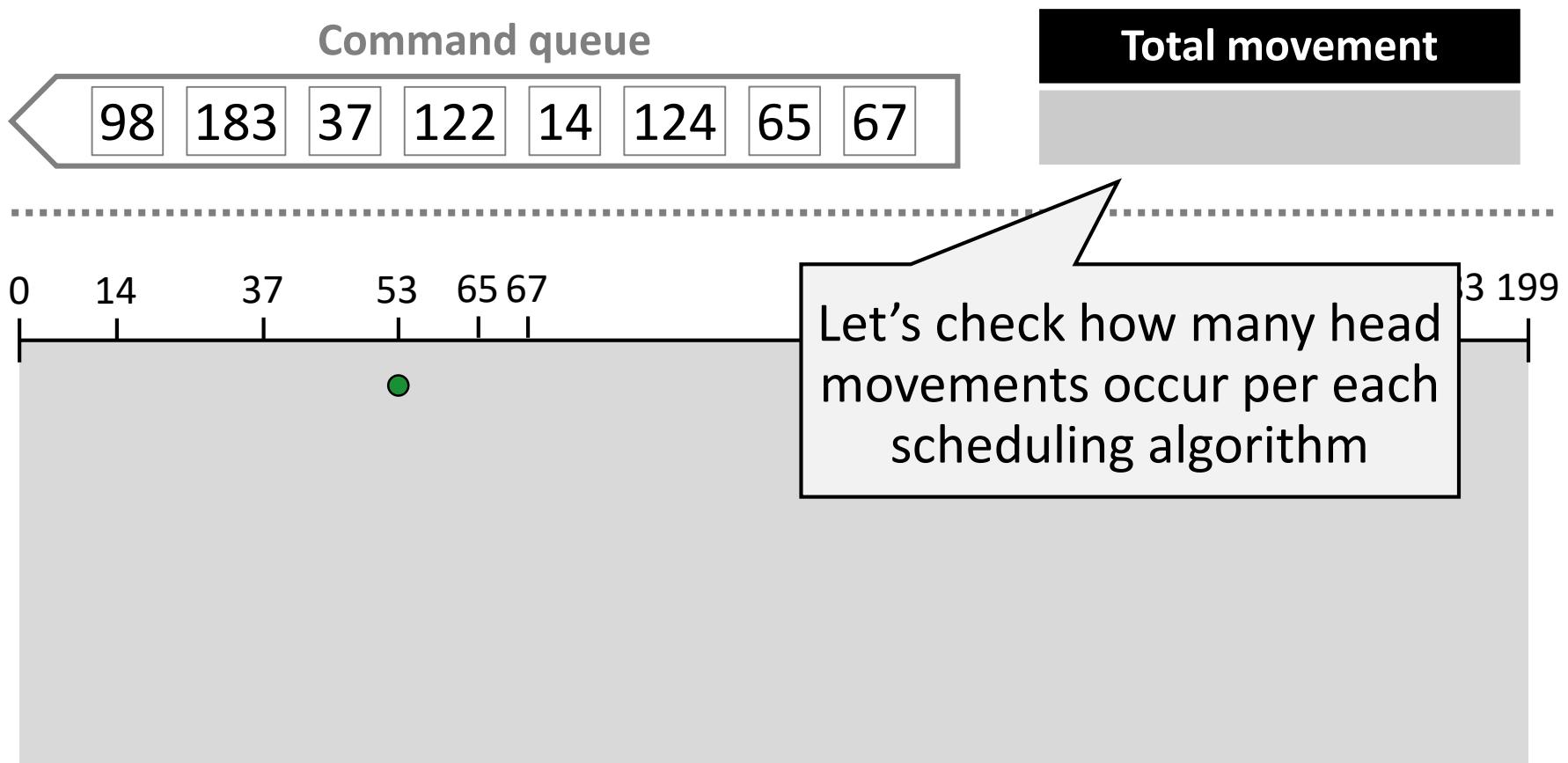
Simple Example



Simple Example

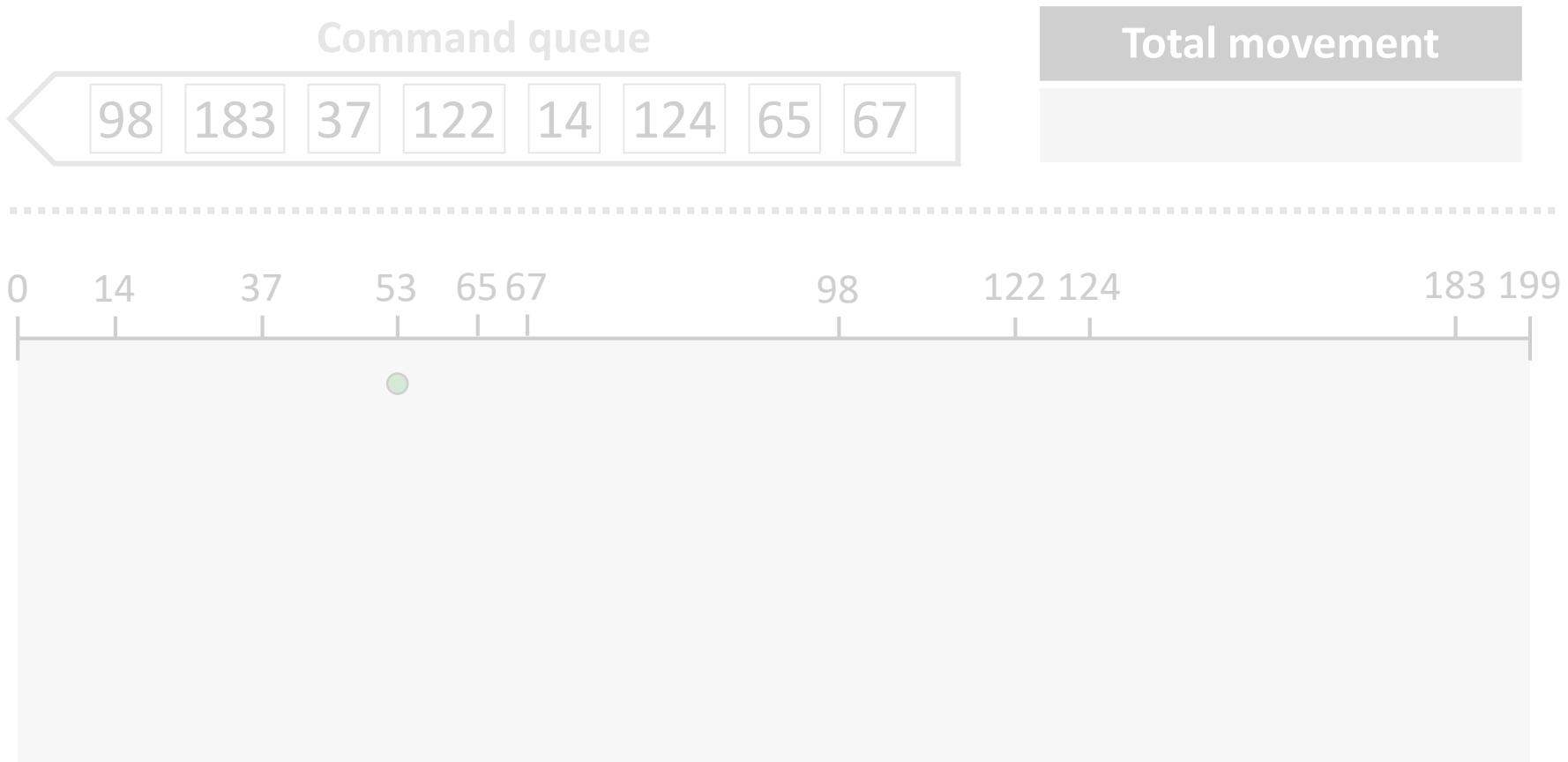


Simple Example



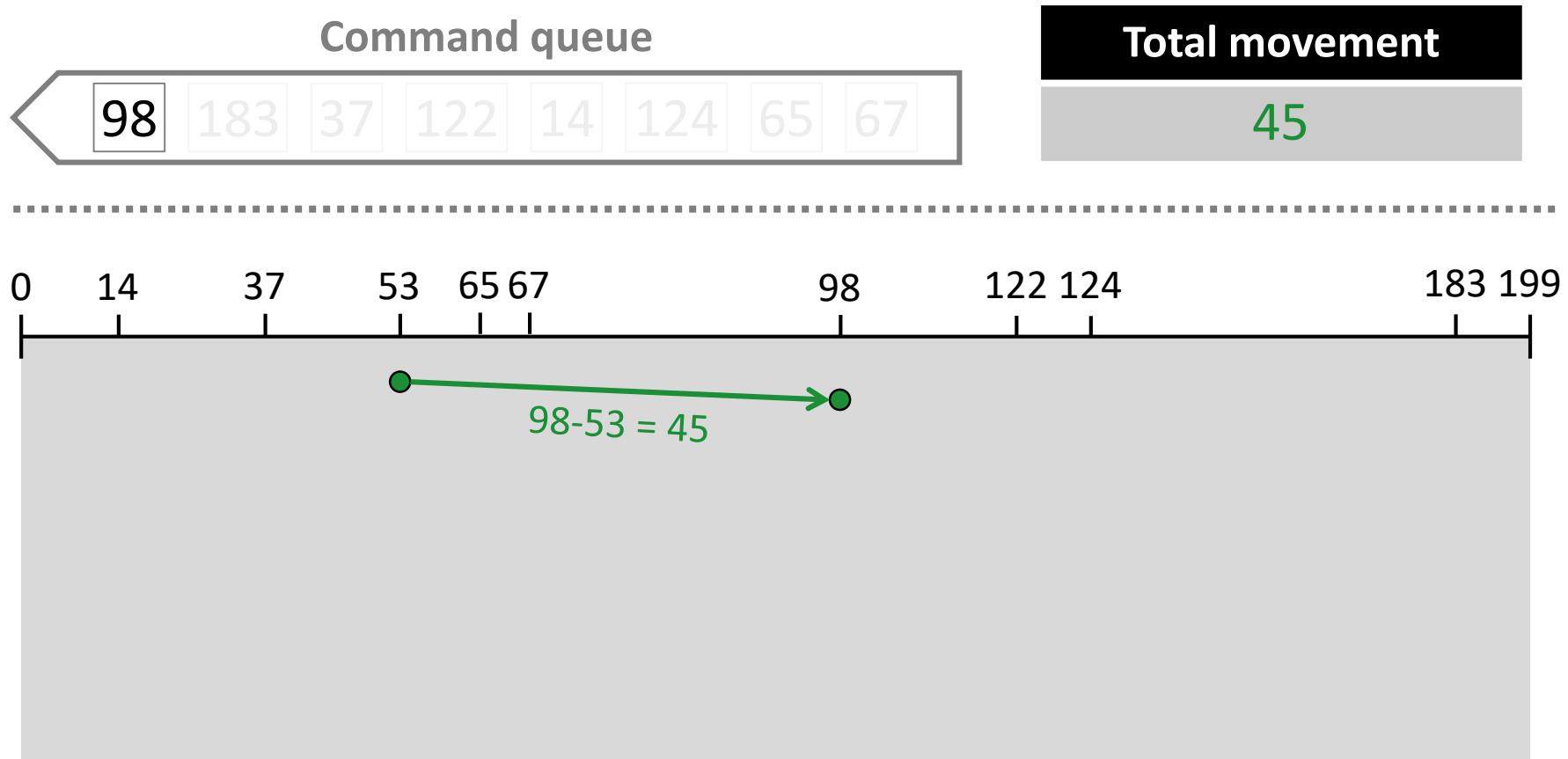
► #1: First Come, First Serve (FCFS)

- **Objective:** Process disk requests in the order they are received



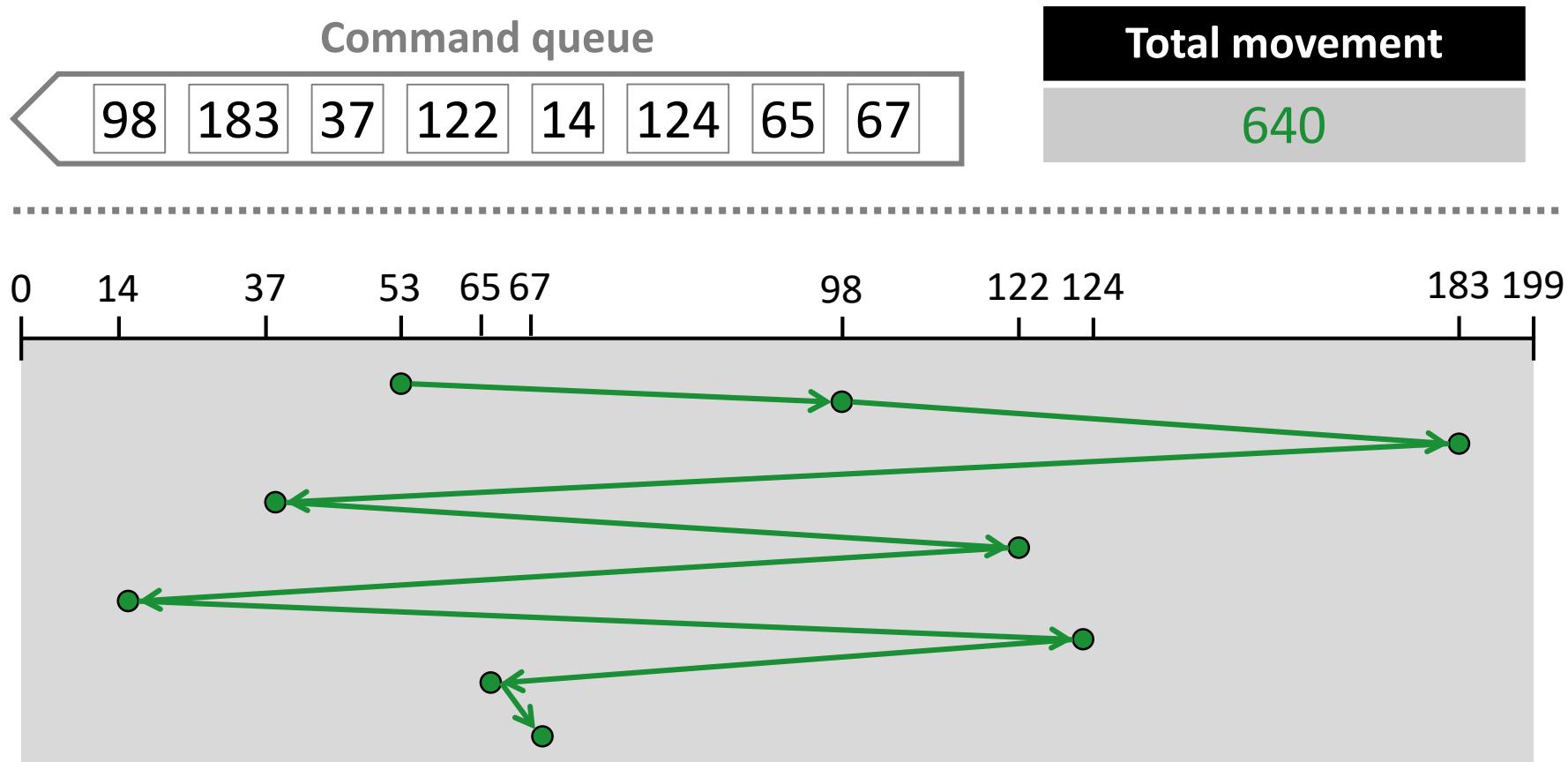
► #1: First Come, First Serve (FCFS)

- **Objective:** Process disk requests in the order they are received



#1: First Come, First Serve (FCFS)

- **Objective:** Process disk requests in the order they are received



#1: First Come, First Serve (FCFS)

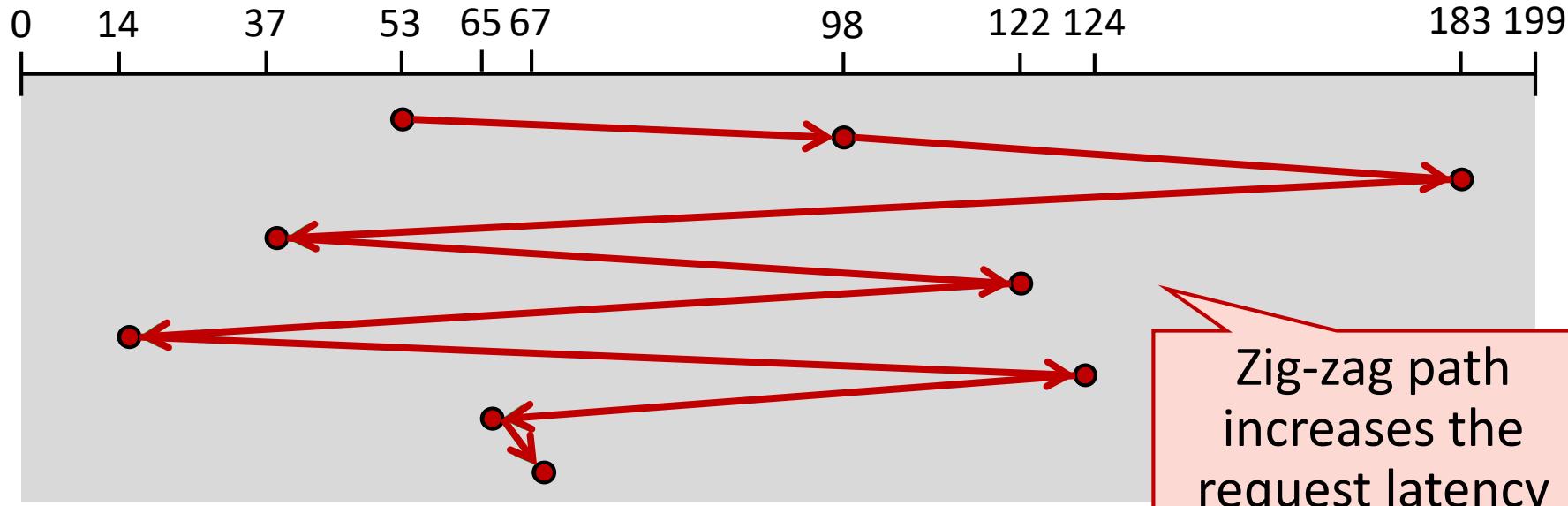
- Objective: Process disk requests in the order they arrive

The good

- Easy to implement
- Good fairness

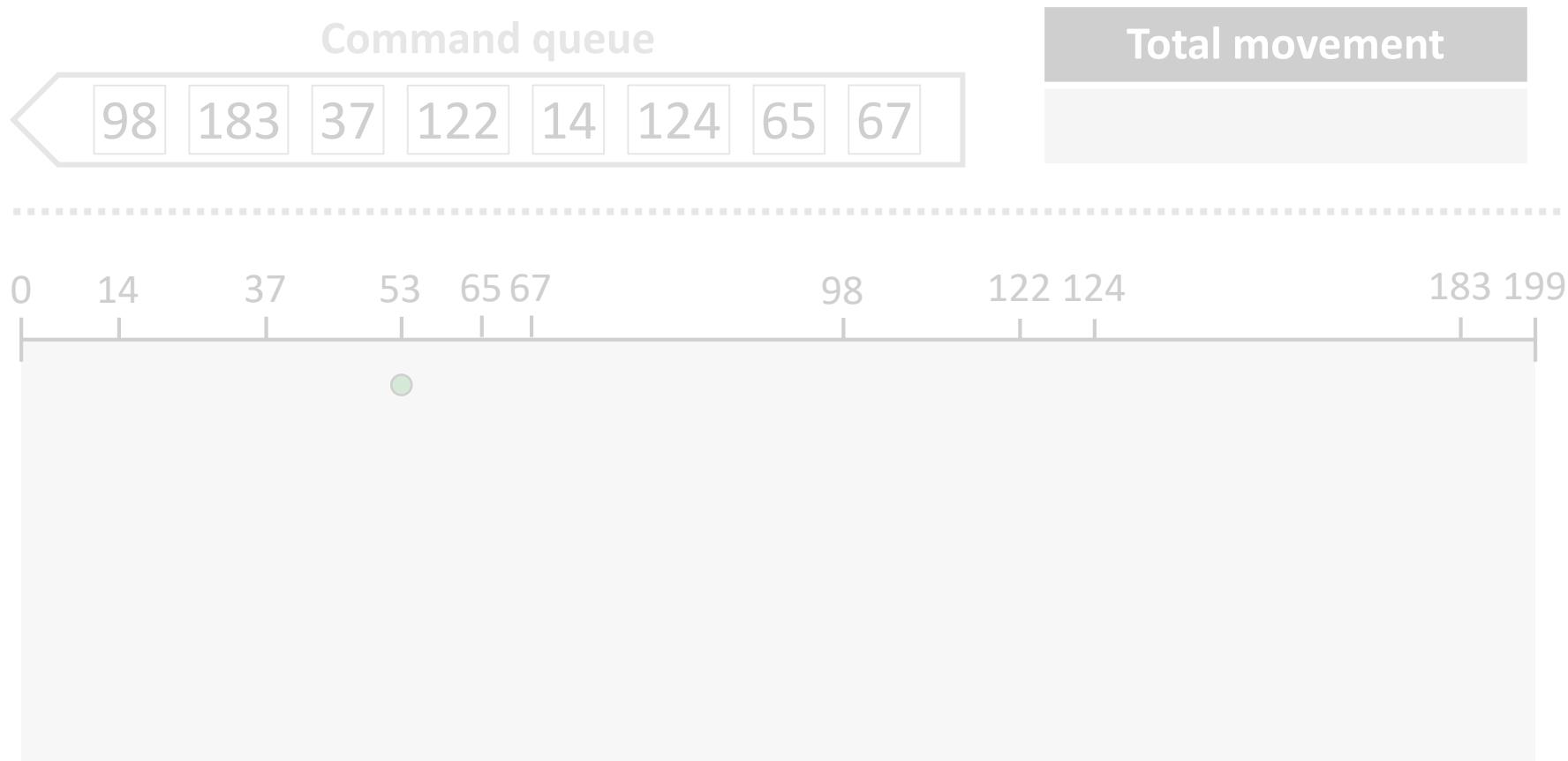
The bad

- Cannot exploit request locality
- Increases average latency, decreasing throughput



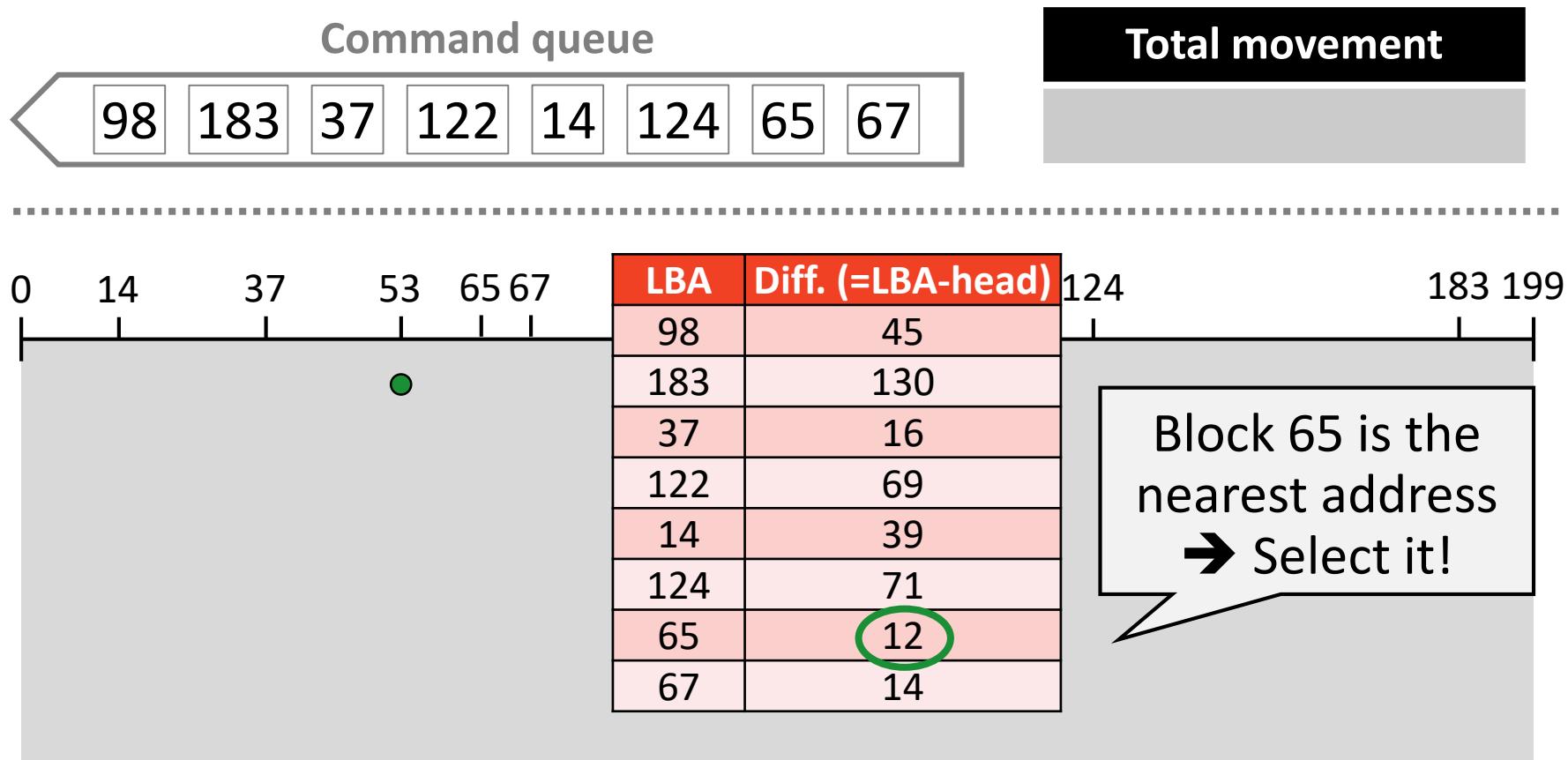
#2: Shortest Seek Time First (SSTF)

- **Objective:** minimize seek time by always selecting the block with the shortest seek time



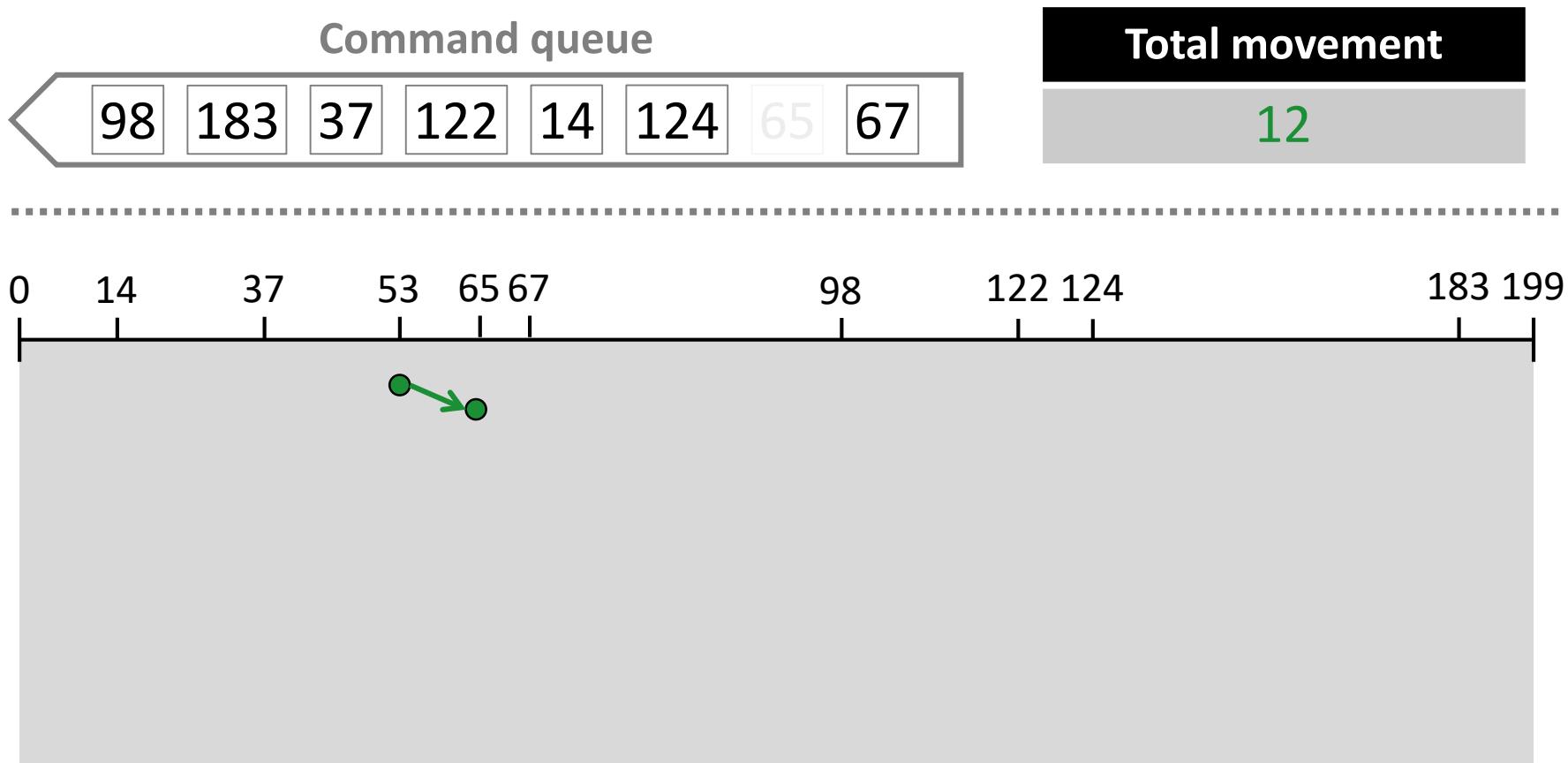
#2: Shortest Seek Time First (SSTF)

- **Objective:** minimize seek time by always selecting the block with the shortest seek time



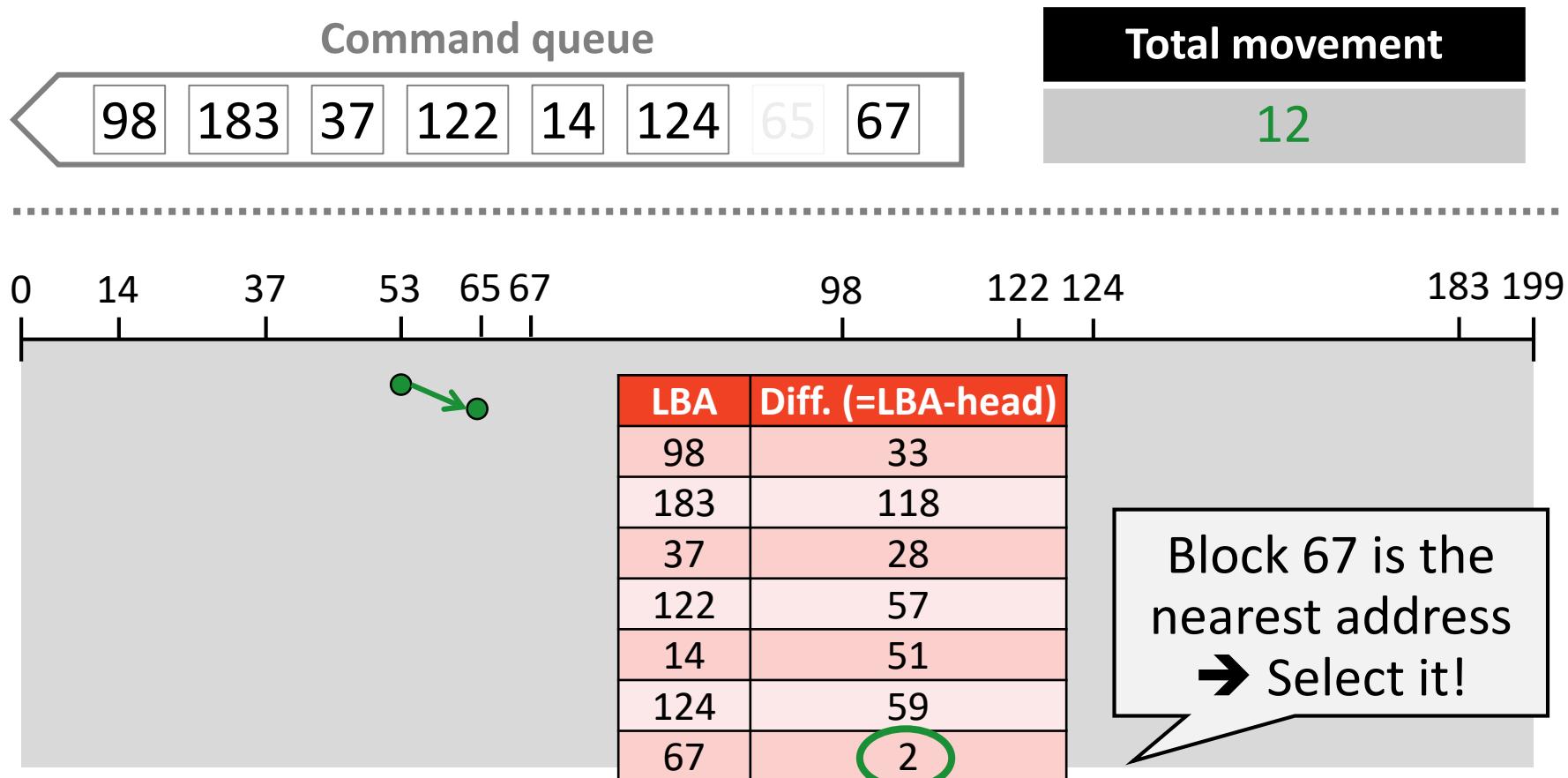
#2: Shortest Seek Time First (SSTF)

- **Objective:** minimize seek time by always selecting the block with the shortest seek time



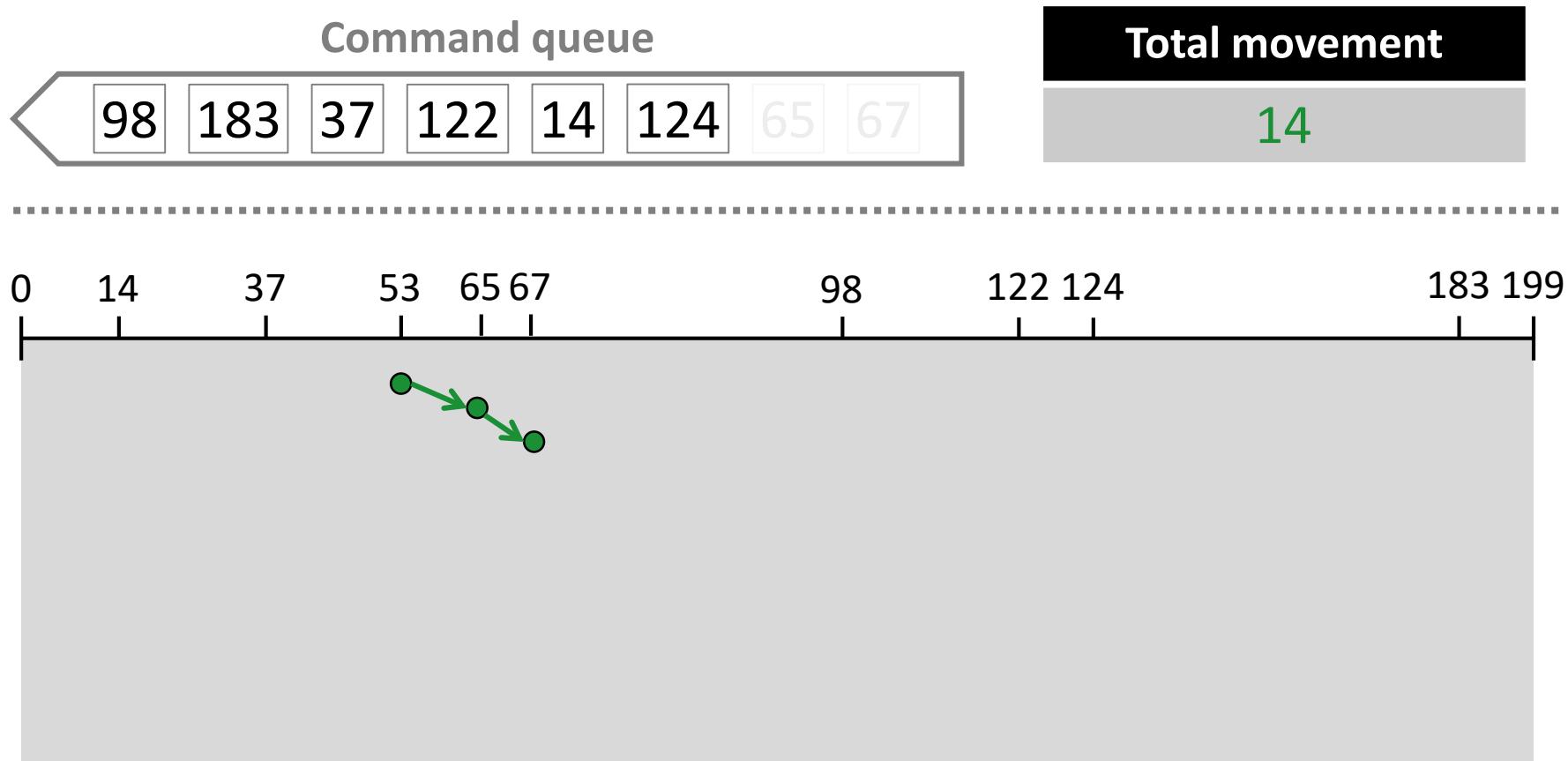
#2: Shortest Seek Time First (SSTF)

- **Objective:** minimize seek time by always selecting the block with the shortest seek time



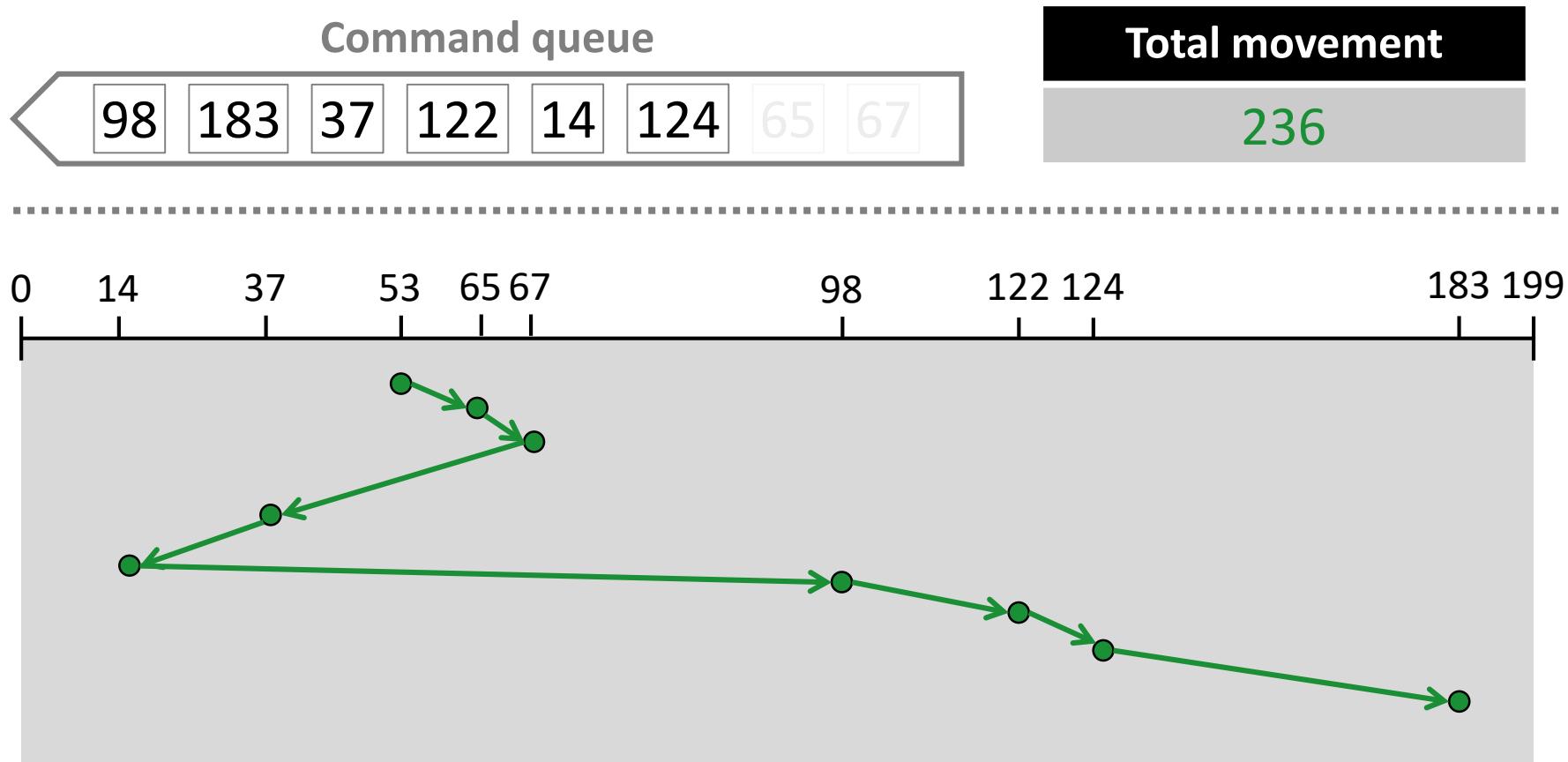
#2: Shortest Seek Time First (SSTF)

- **Objective:** minimize seek time by always selecting the block with the shortest seek time



#2: Shortest Seek Time First (SSTF)

- **Objective:** minimize seek time by always selecting the block with the shortest seek time





#2: Shortest Seek Time First (SSTF)

- Objective: minimize seek time by always selecting the shortest distance
- Pros:
 - t
 - SSTF is optimal
 - Can be easily implemented

The good

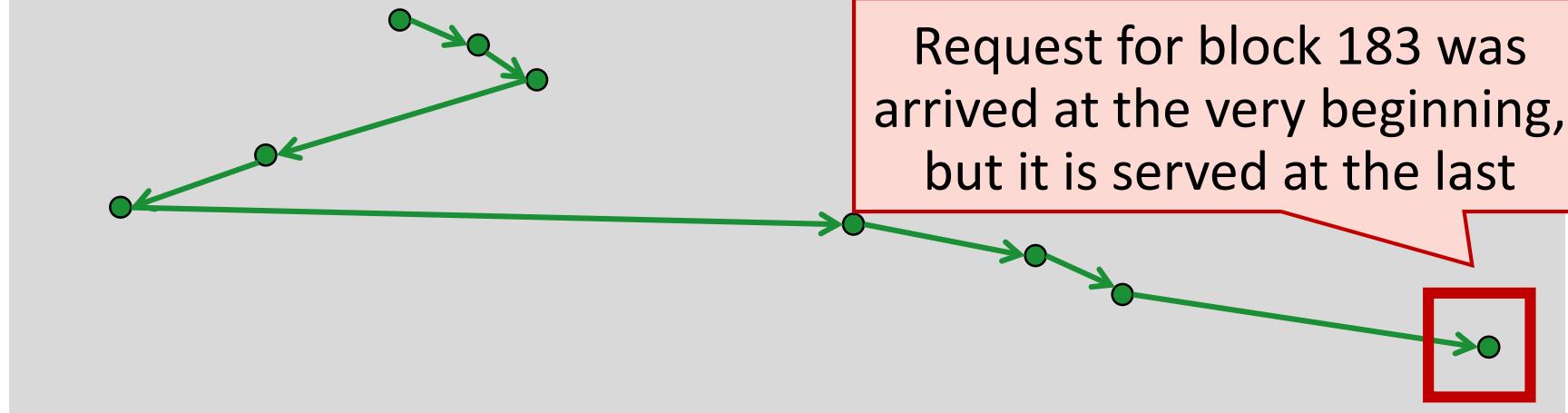
- SSTF is optimal
- Can be easily implemented

The bad

- Proven to starvation

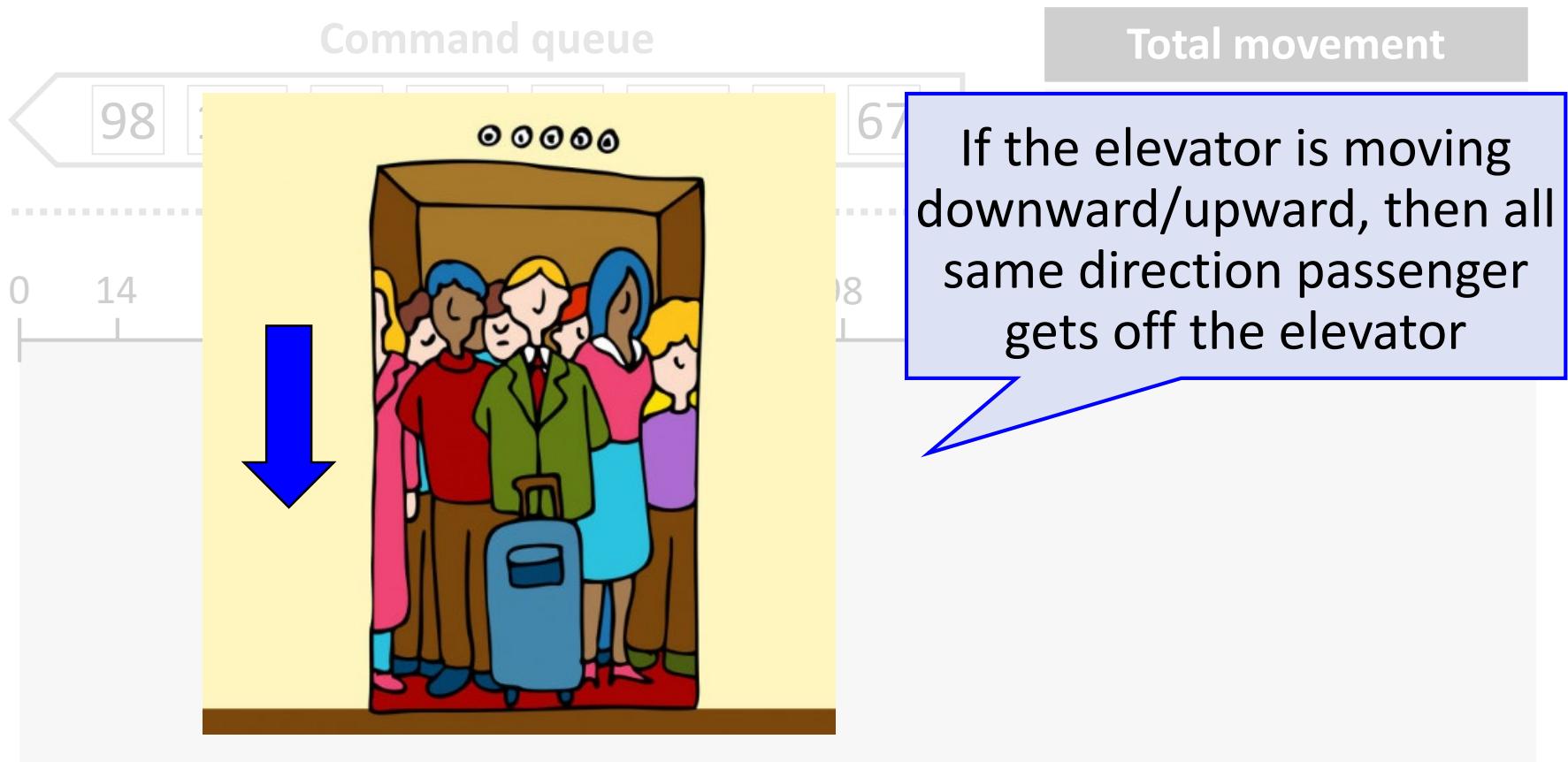


Request for block 183 was arrived at the very beginning, but it is served at the last



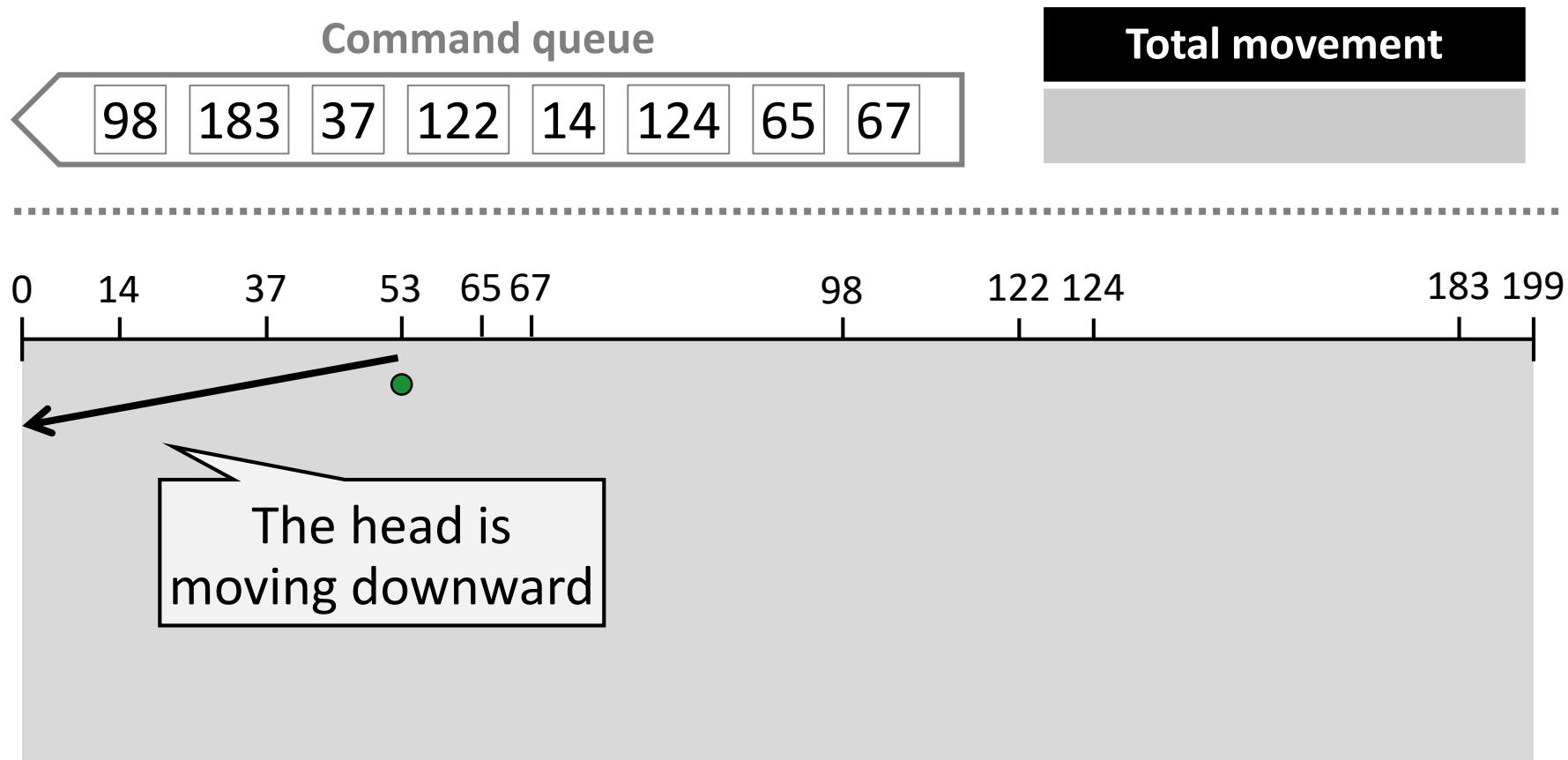
► #3: SCAN (a.k.a Elevator Algorithm)

- **Objective:** head sweeps across the disk, servicing all requests in the path



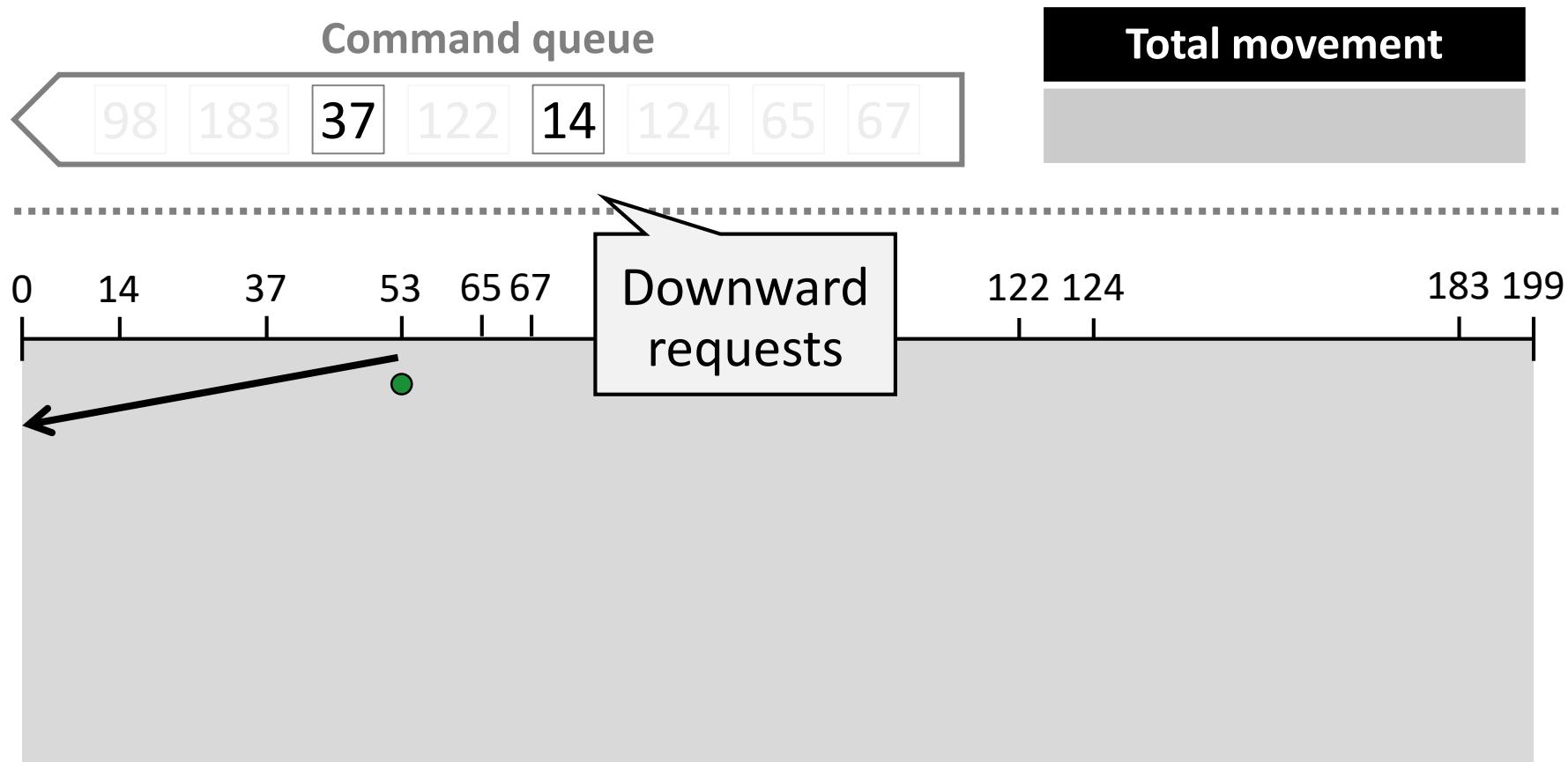
#3: SCAN (a.k.a Elevator Algorithm)

- **Objective:** head sweeps across the disk, servicing all requests in the path



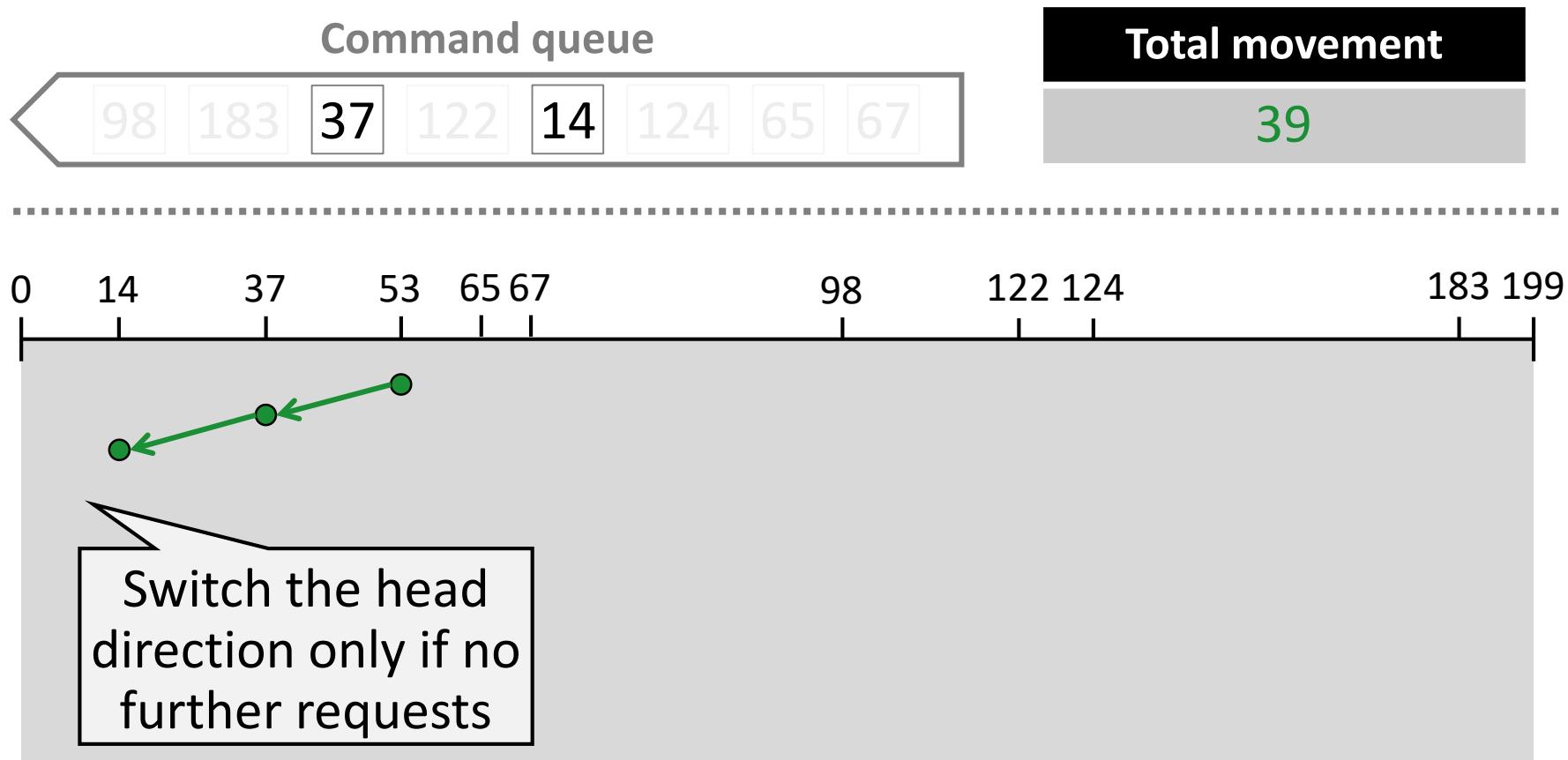
► #3: SCAN (a.k.a Elevator Algorithm)

- **Objective:** head sweeps across the disk, servicing all requests in the path



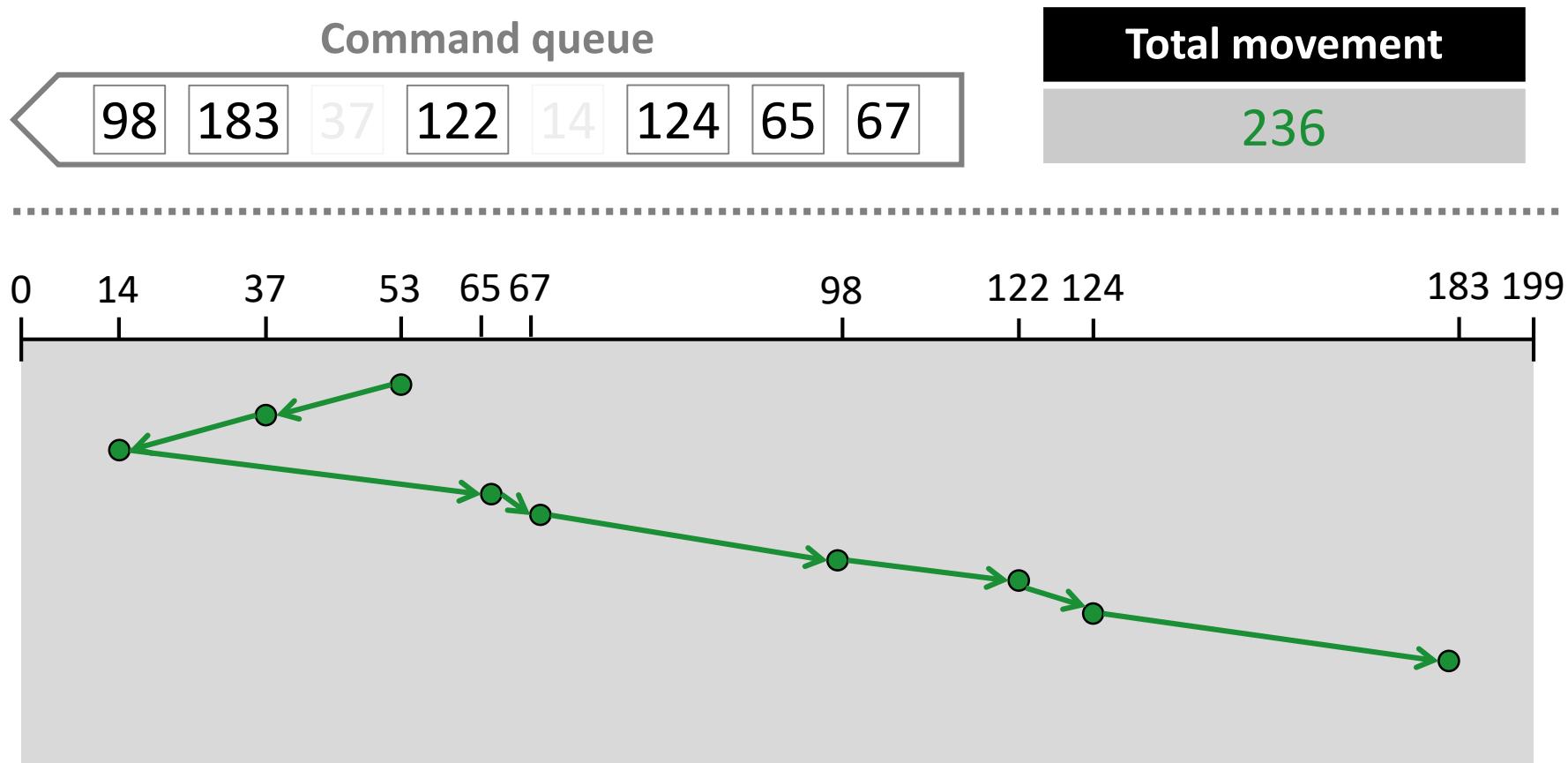
#3: SCAN (a.k.a Elevator Algorithm)

- **Objective:** head sweeps across the disk, servicing all requests in the path



#3: SCAN (a.k.a Elevator Algorithm)

- **Objective:** head sweeps across the disk, servicing all requests in the path



#3: SCAN (a.k.a Elevator Algorithm)

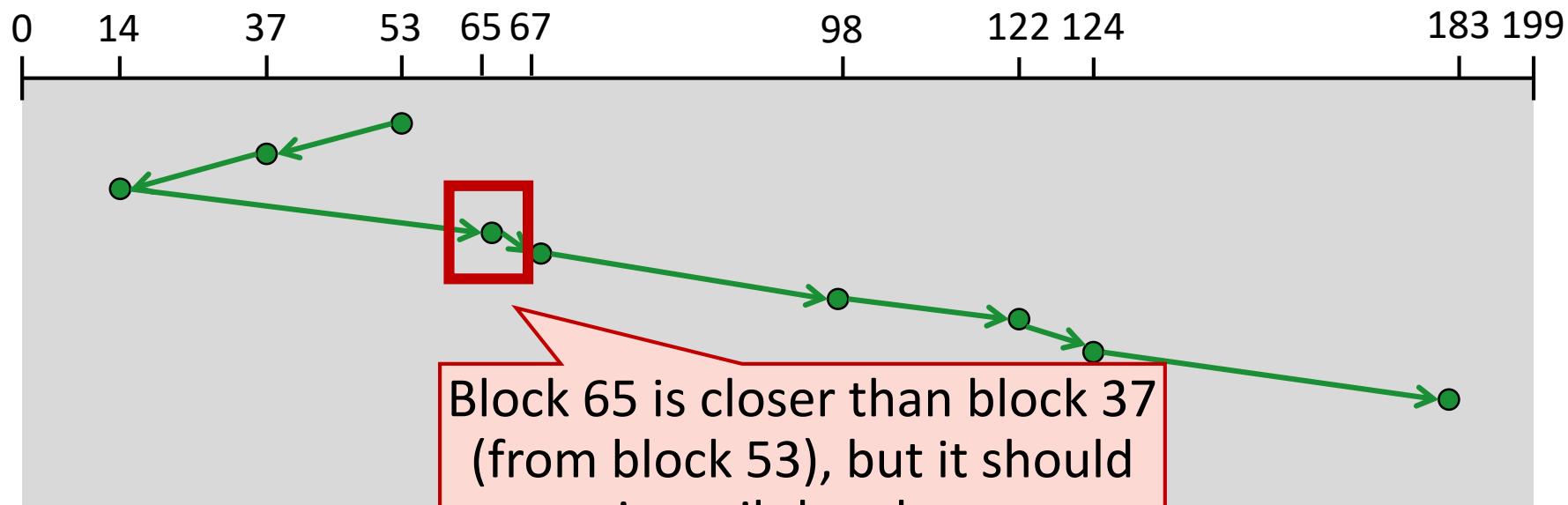
- Objective: head sweeps across the disk, servicing requests
- a
- 65

The good

- Reasonable performance
- No starvation (bounded waiting)

The bad

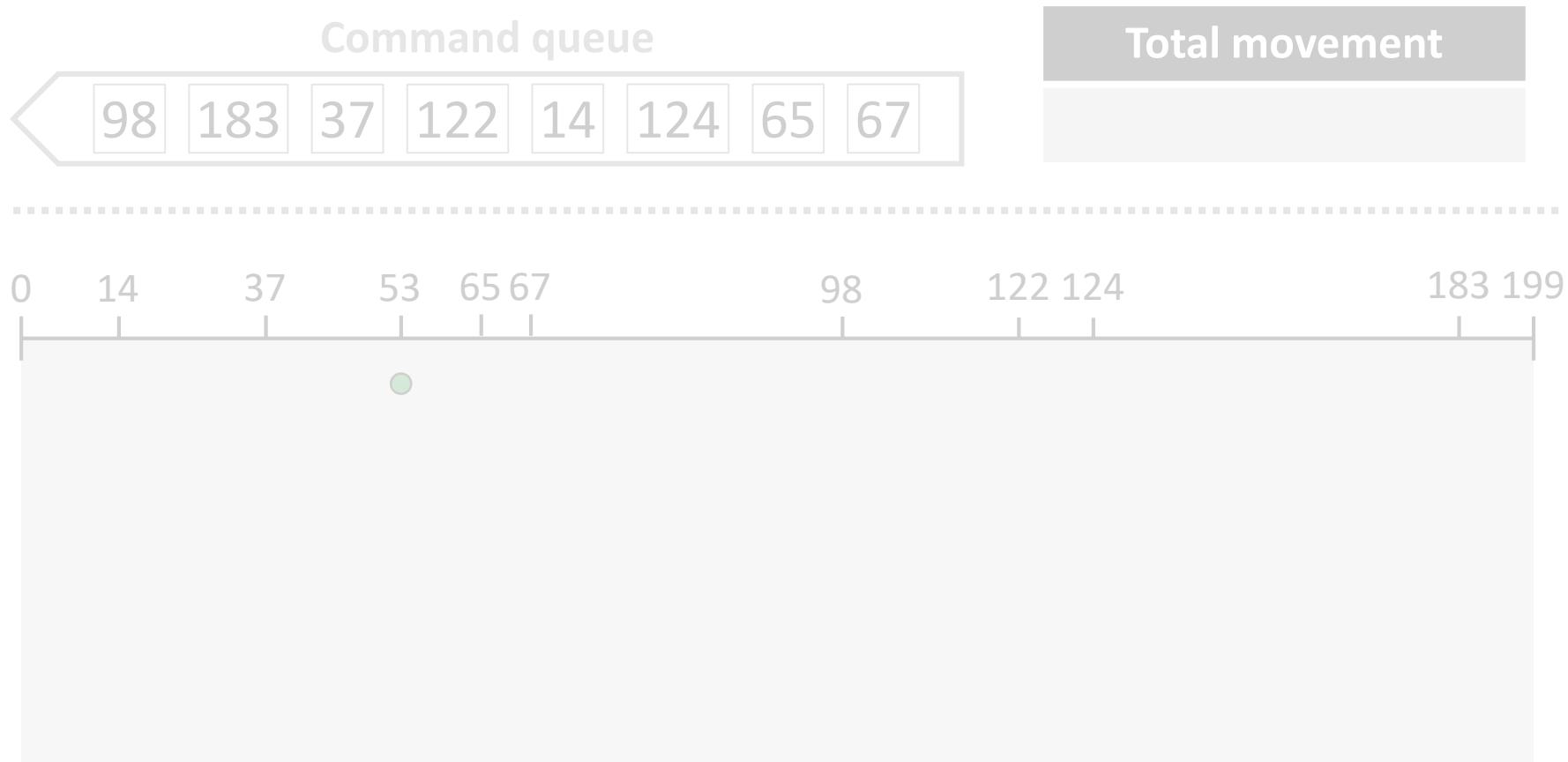
- Middle address get better service
- Might miss locality that SSTF could exploit





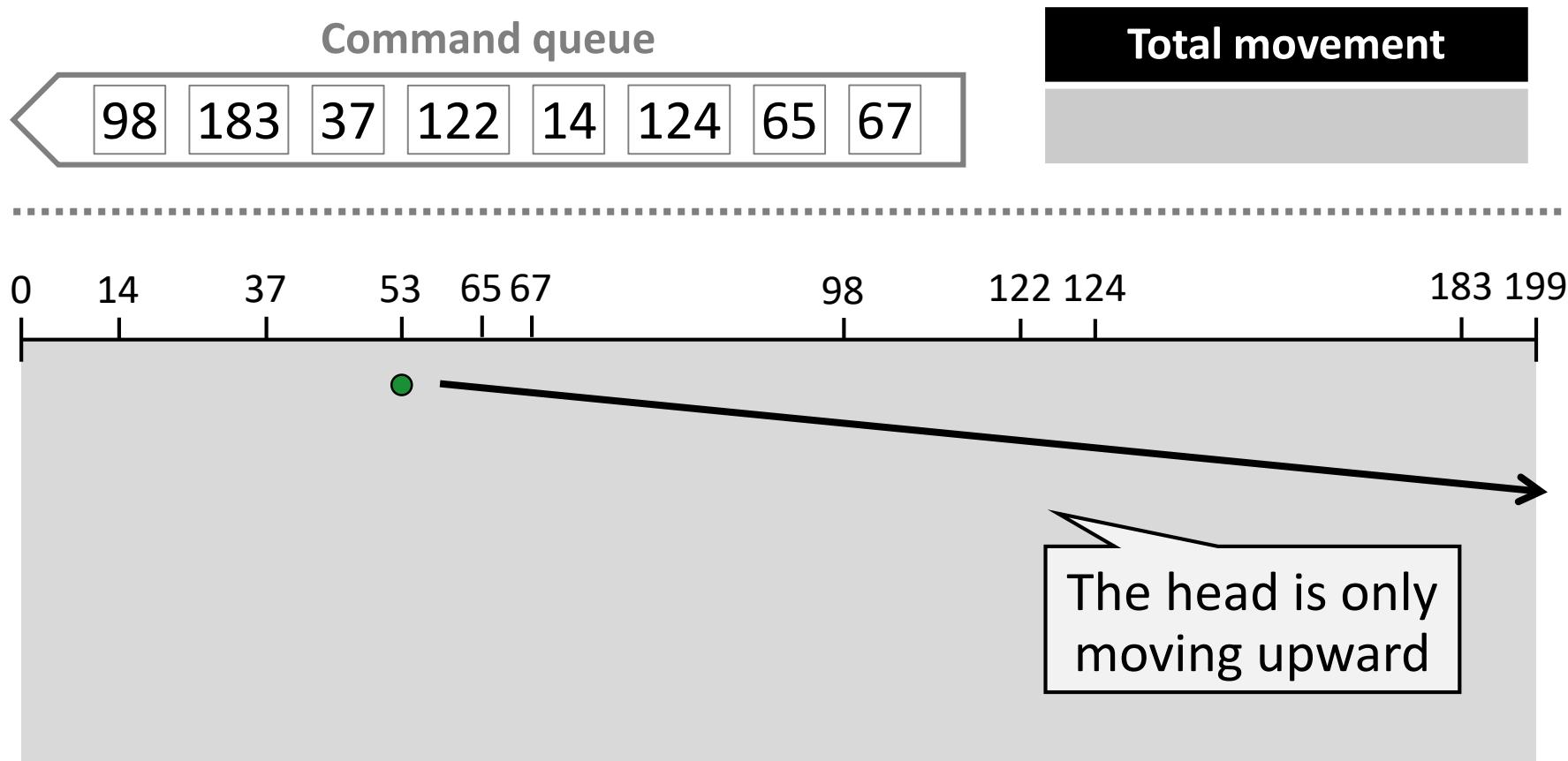
#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



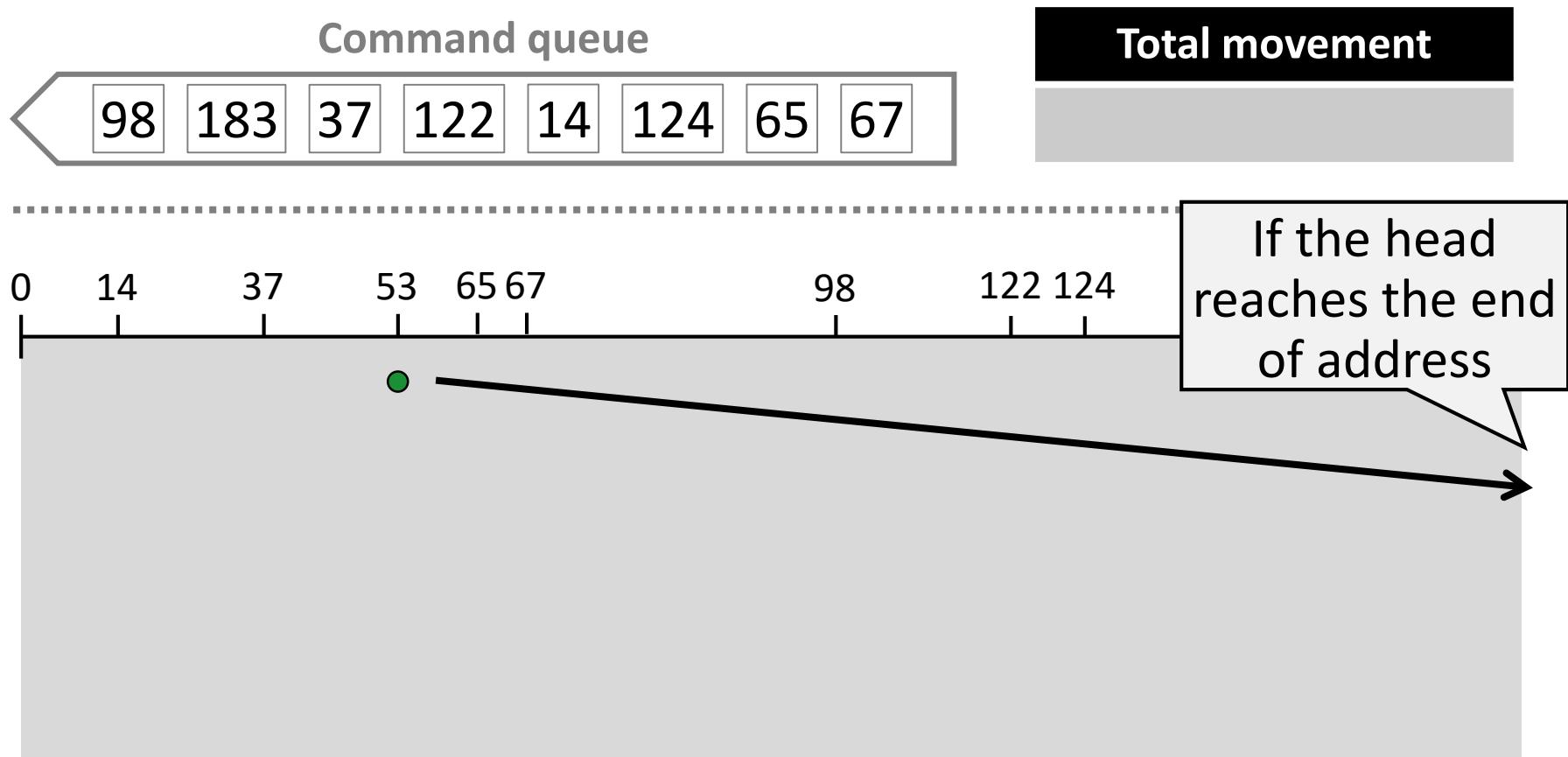
#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



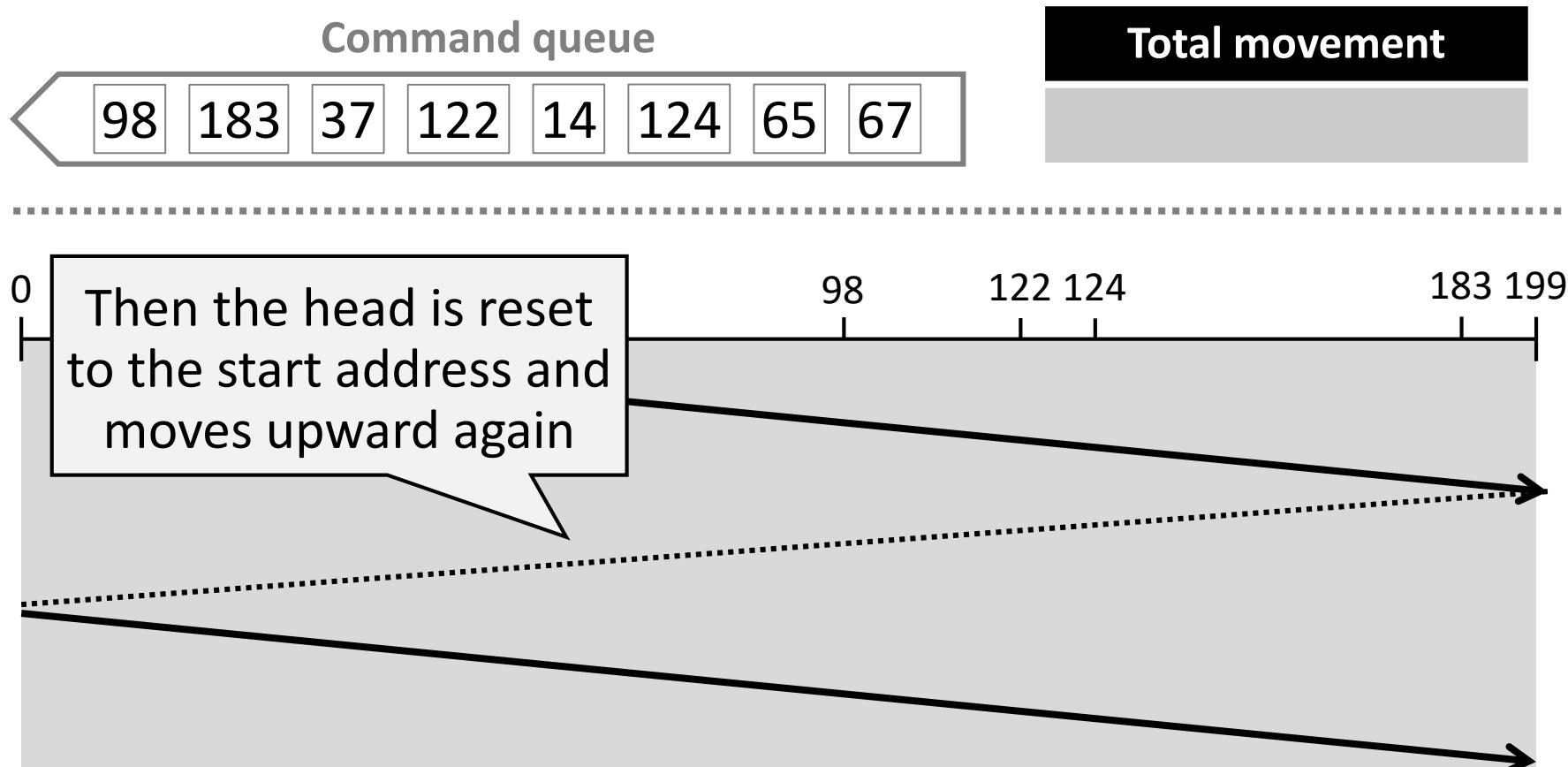
#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



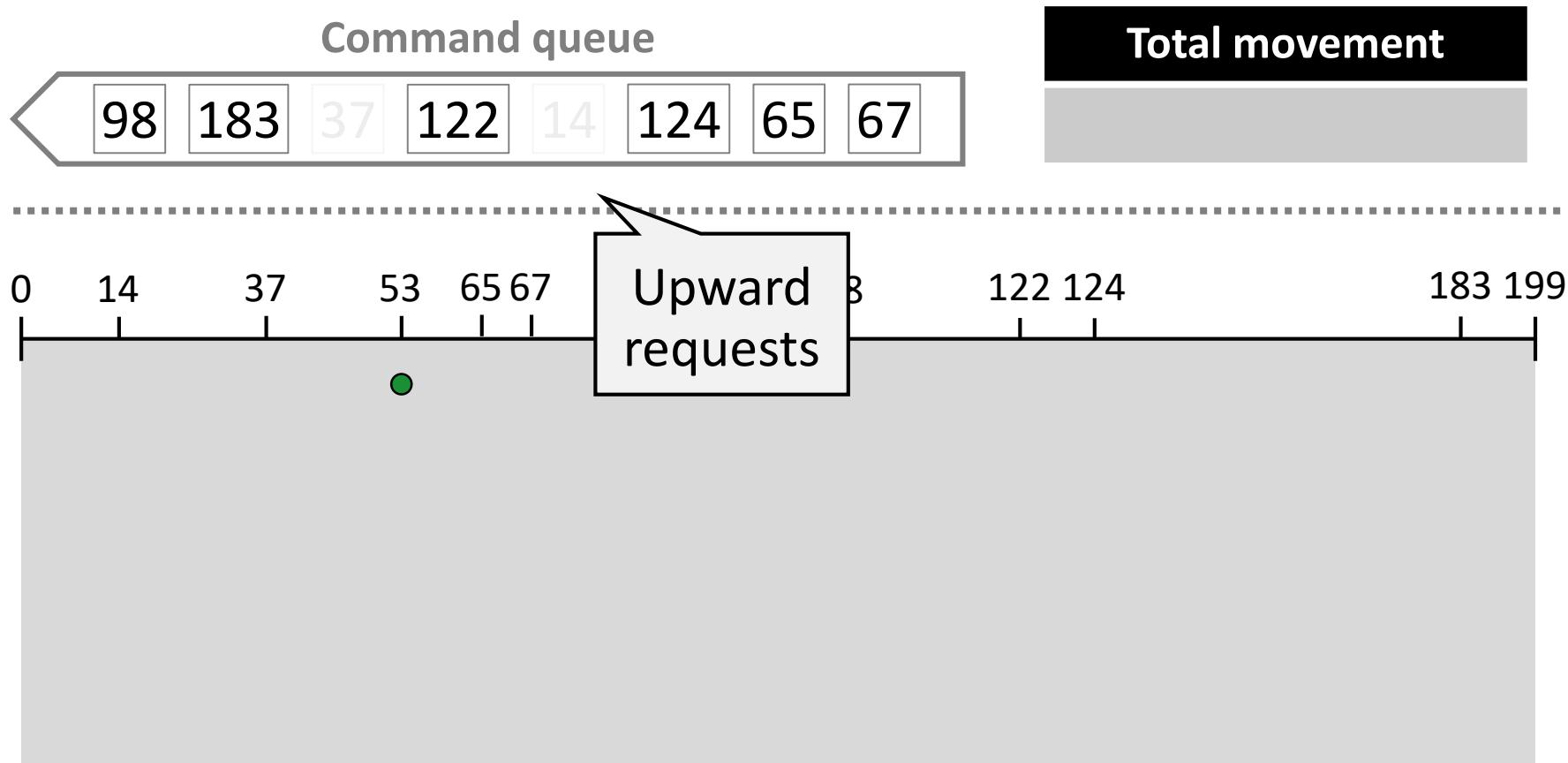
#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



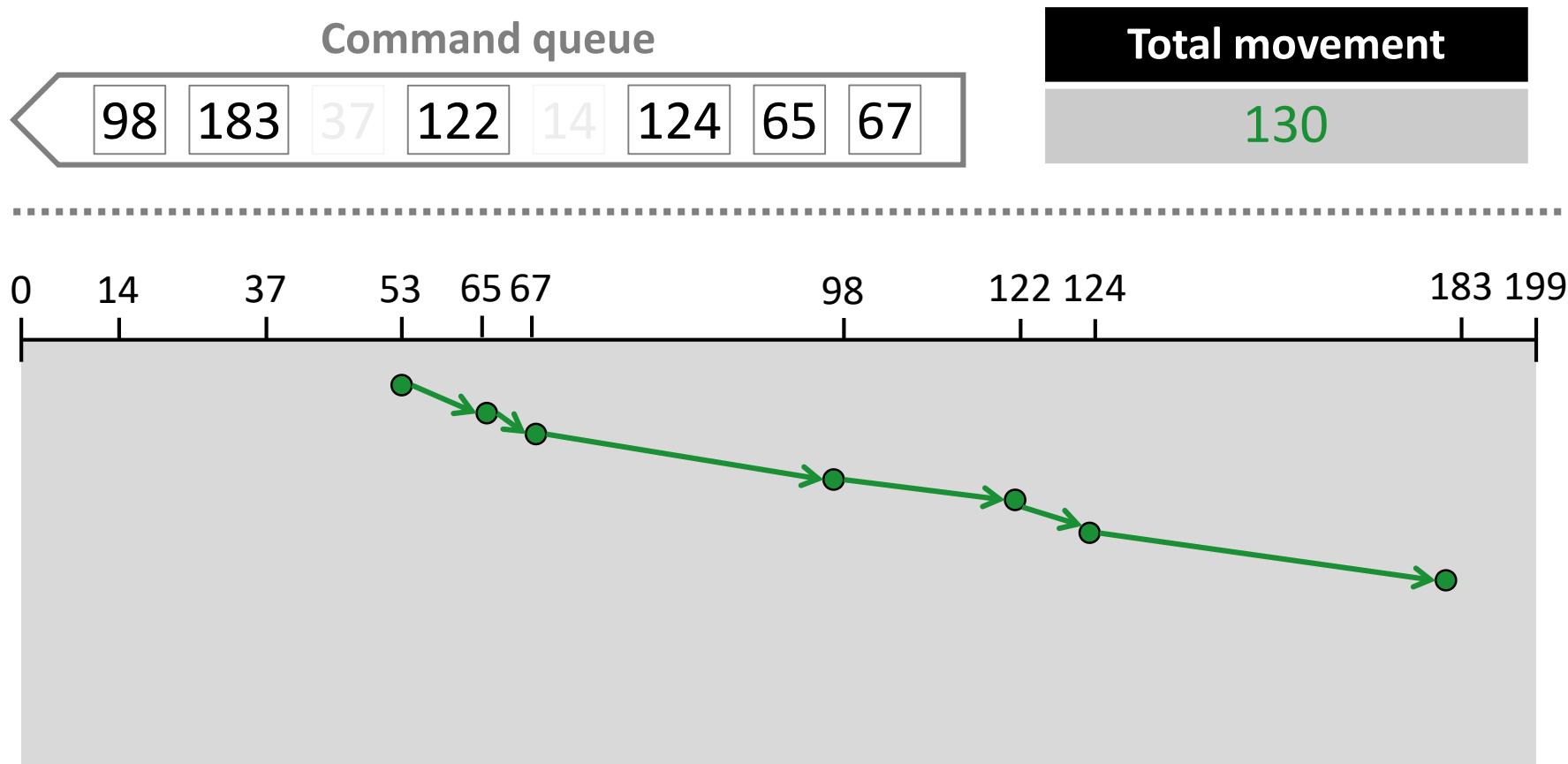
#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



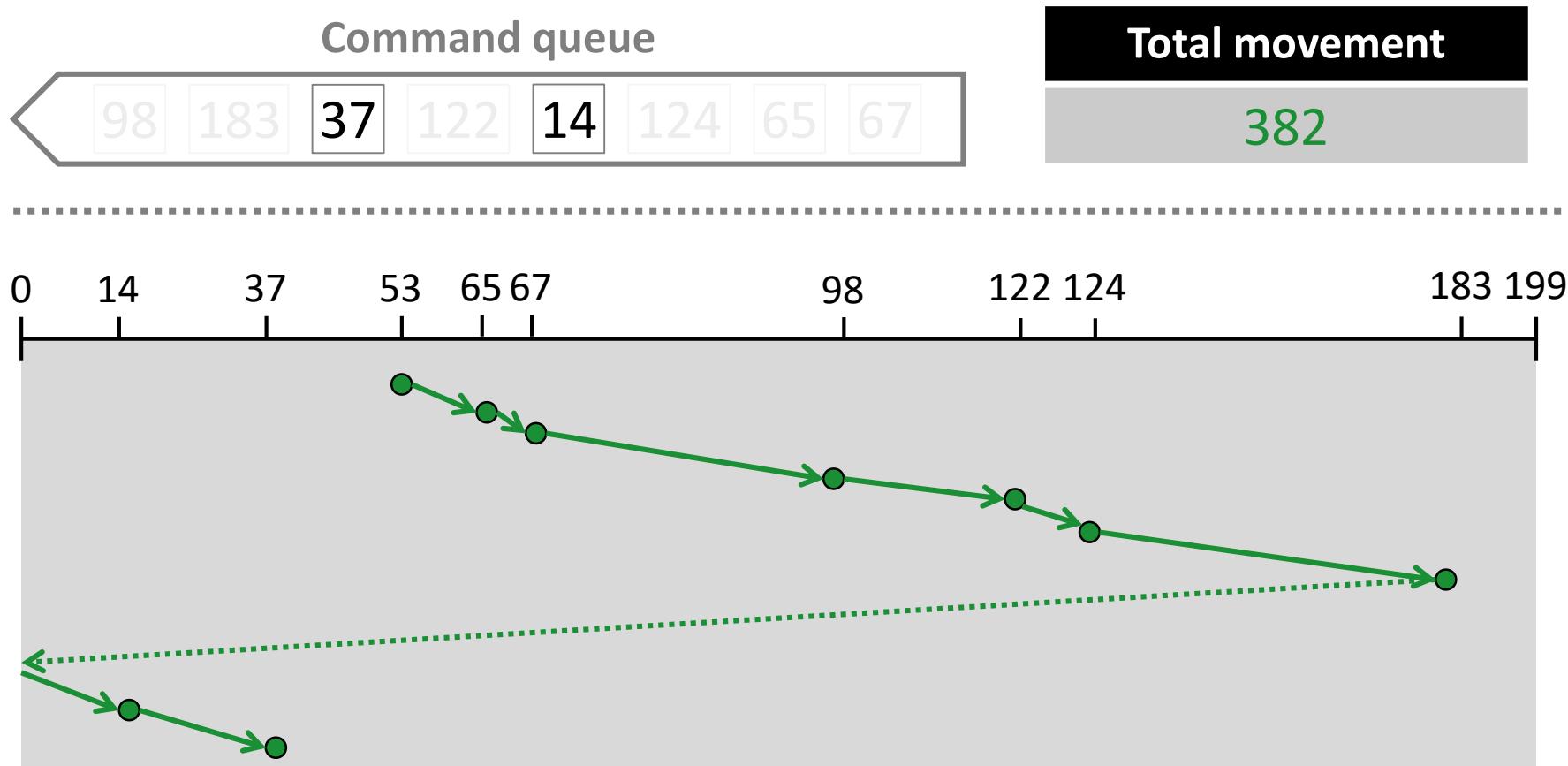
#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



#4: C-SCAN

- **Objective:** like SCAN, but only service requests in one direction



#4: C-SCAN

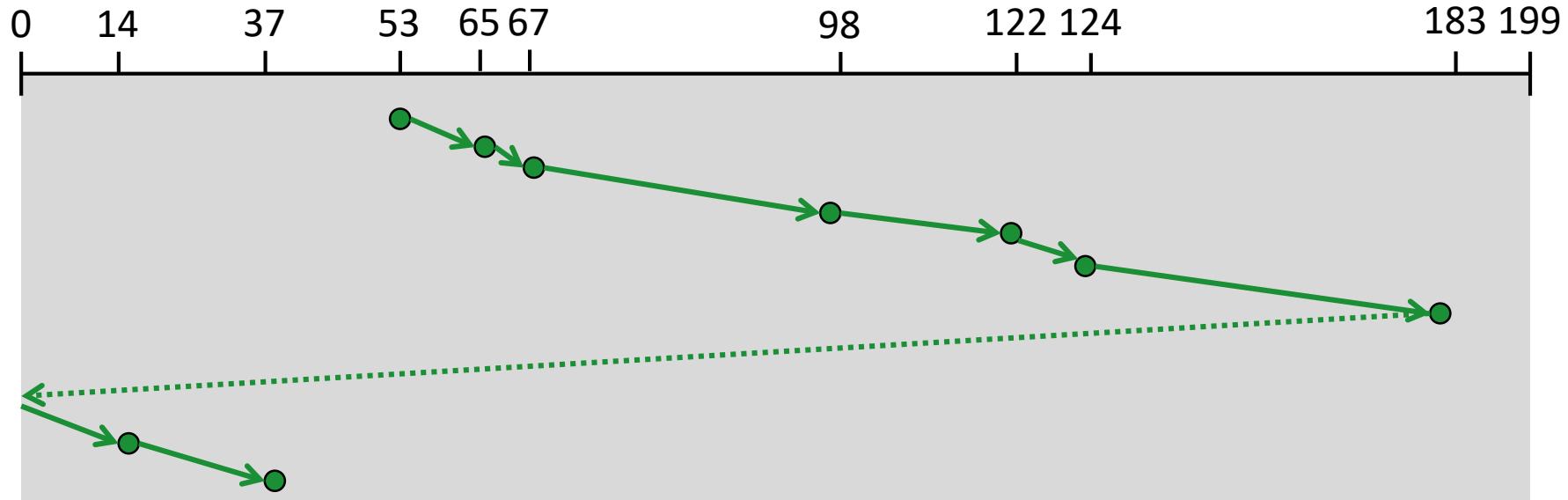
- Objective: like SCAN, but only service requests in one direction

The good

- Fairer than SCAN

The bad

- Worse performance than SCAN



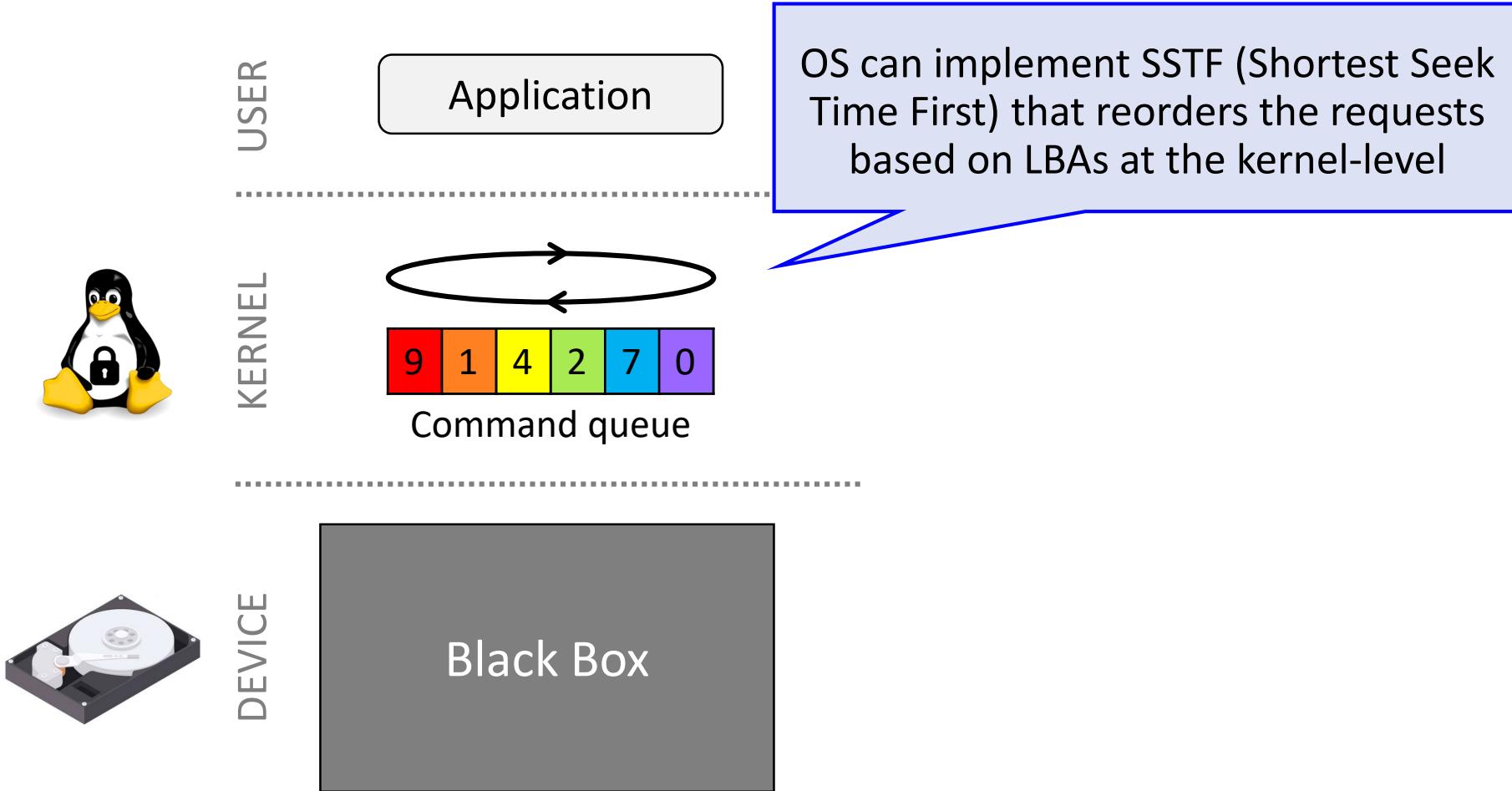
Implement Disk Scheduling

We have talked about several scheduling problems that take a place in the kernel (e.g., process scheduling, page swapping)

Where should be disk scheduling implemented?



#1: OS Scheduling



#1: OS Scheduling



USER

Application

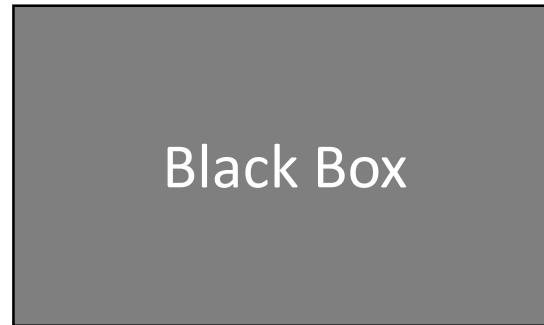
KERNEL



Command queue



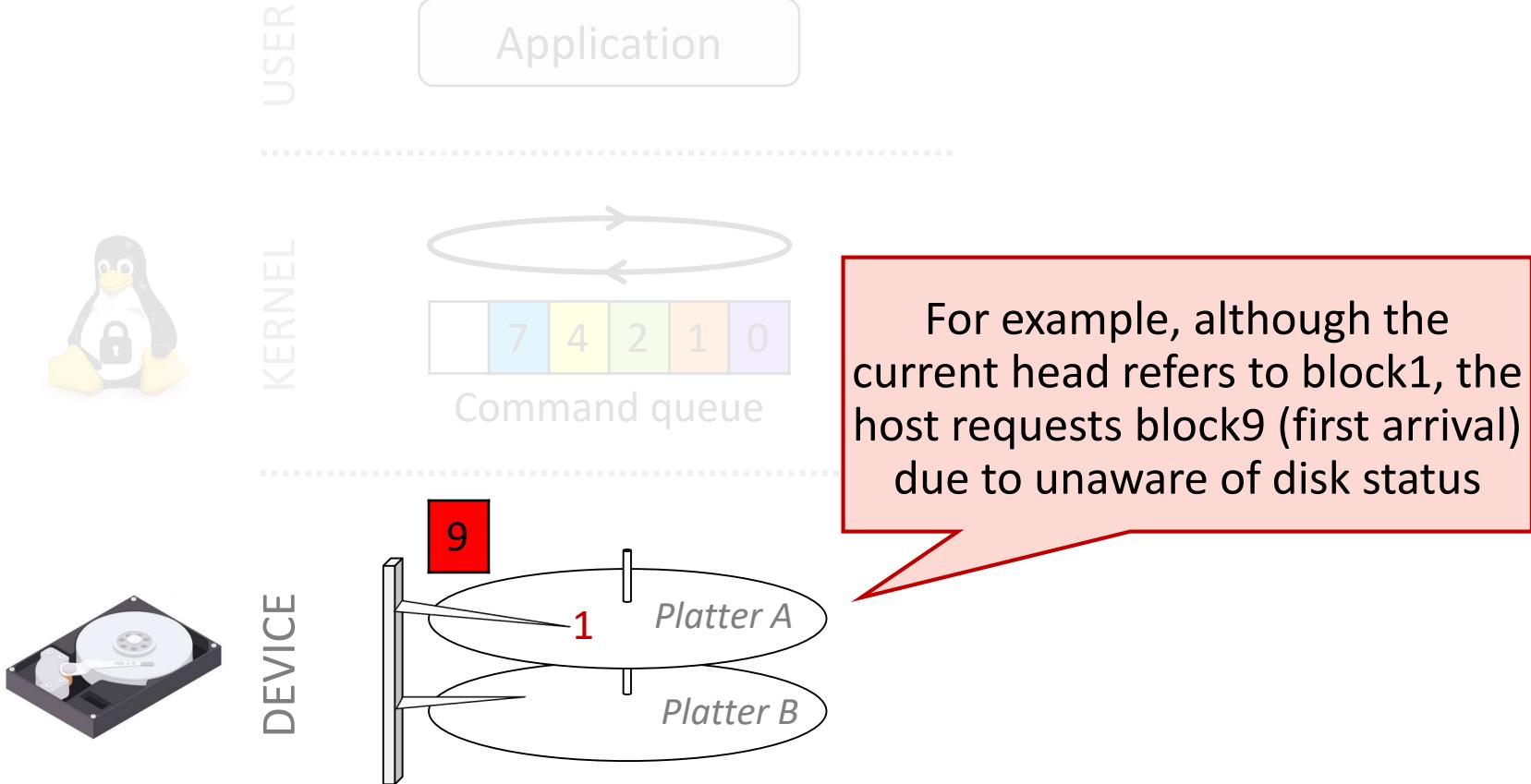
DEVICE



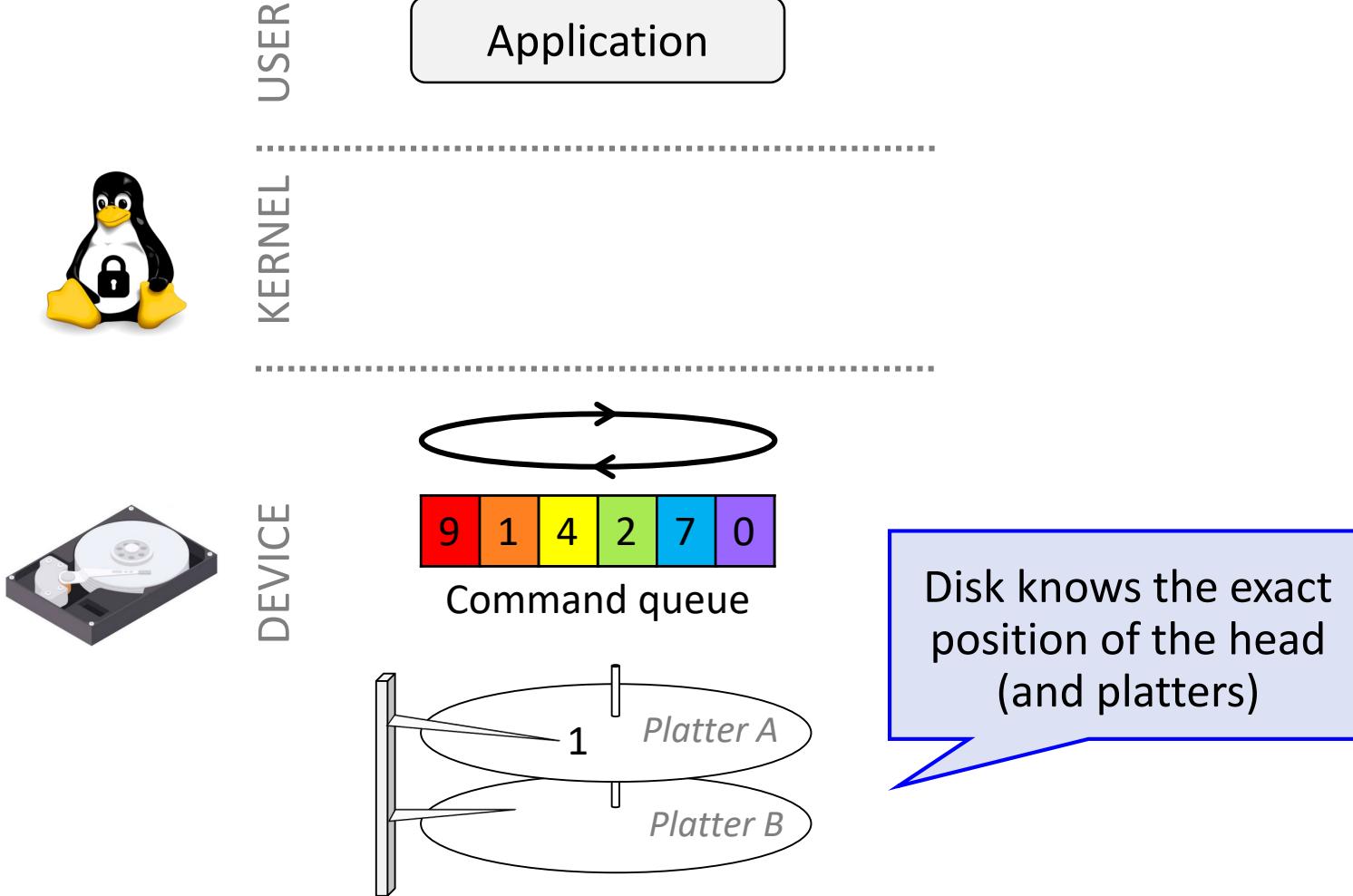
However, OS has no exact information of the underlying hardware (e.g., where is the current head)



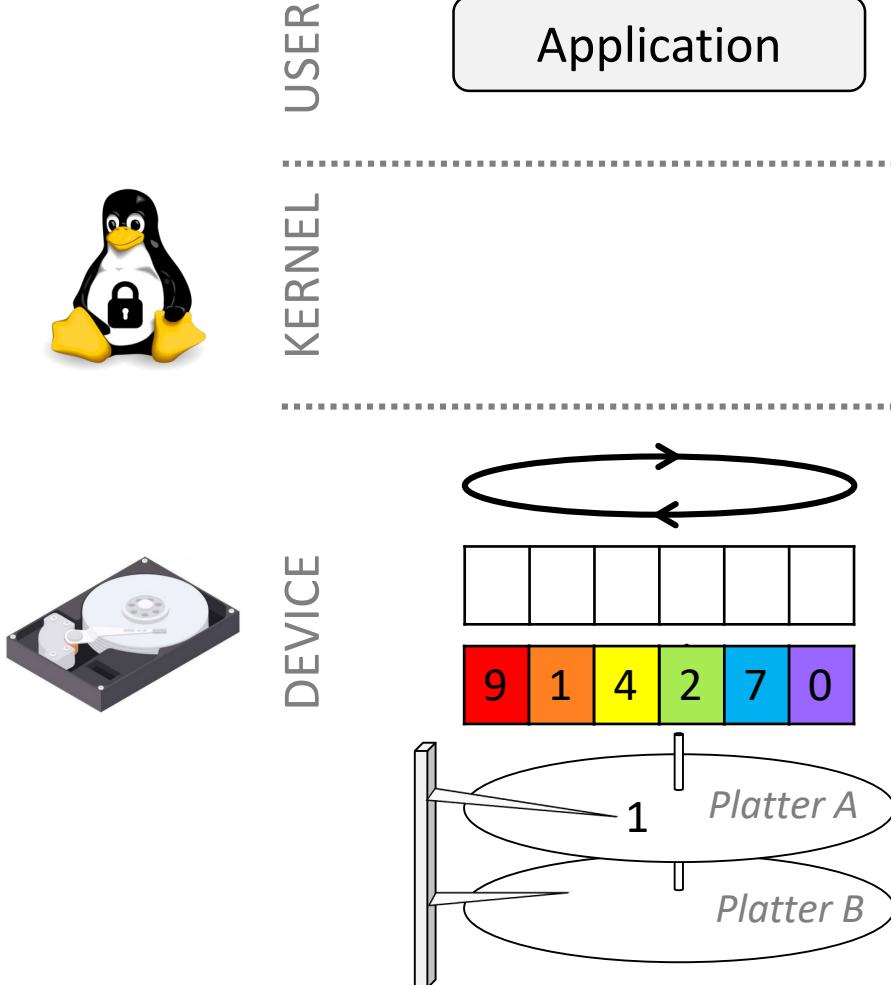
#1: OS Scheduling



#2: On-disk Scheduling



#2: On-disk Scheduling



Thus, an on-disk scheduler can have a more advanced algorithm (like a real SSTF)

But, it requires specialized hardware (command queue) and firmware

What's Next?

- Let's check more about disks (beyond single disk and beyond spinning disk)

21S EE415

IO & Disks II

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



IO & Disks II

Myoungsoo Jung
Computer Division

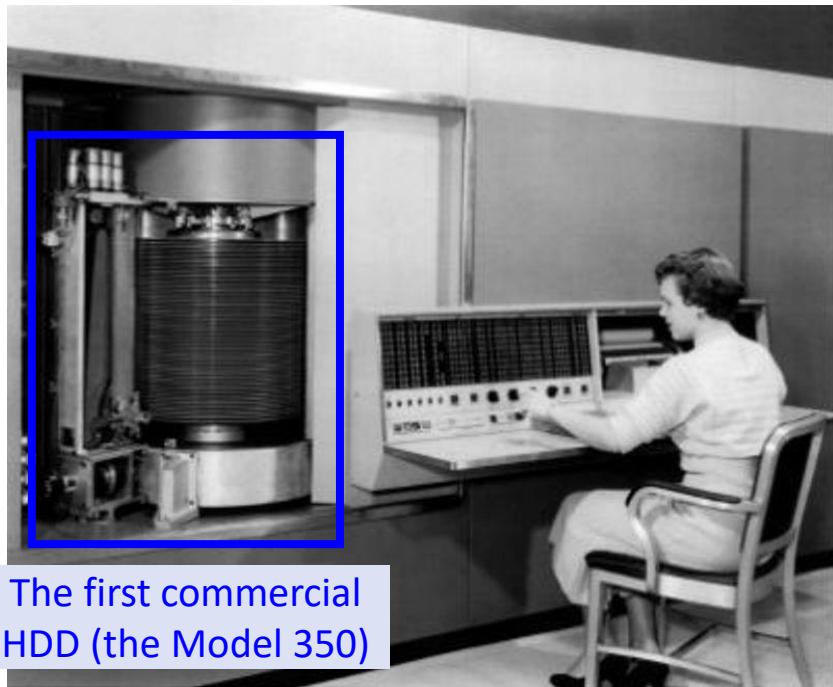
Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab* 

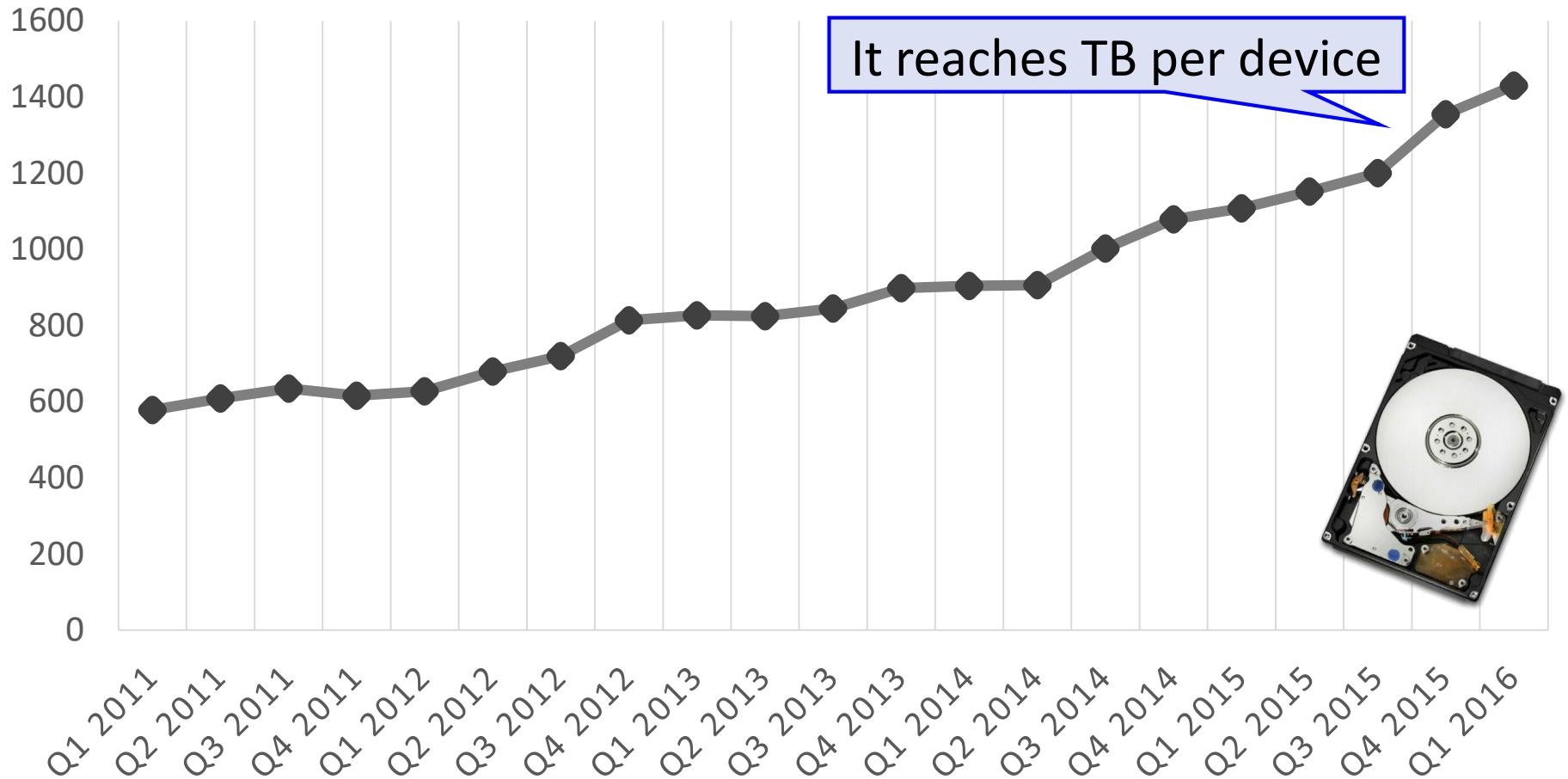
Pros of Spinning Disks

Spinning disks (hard drives) are popularly used in main computing systems since 1956; They are still widely employed as a great device

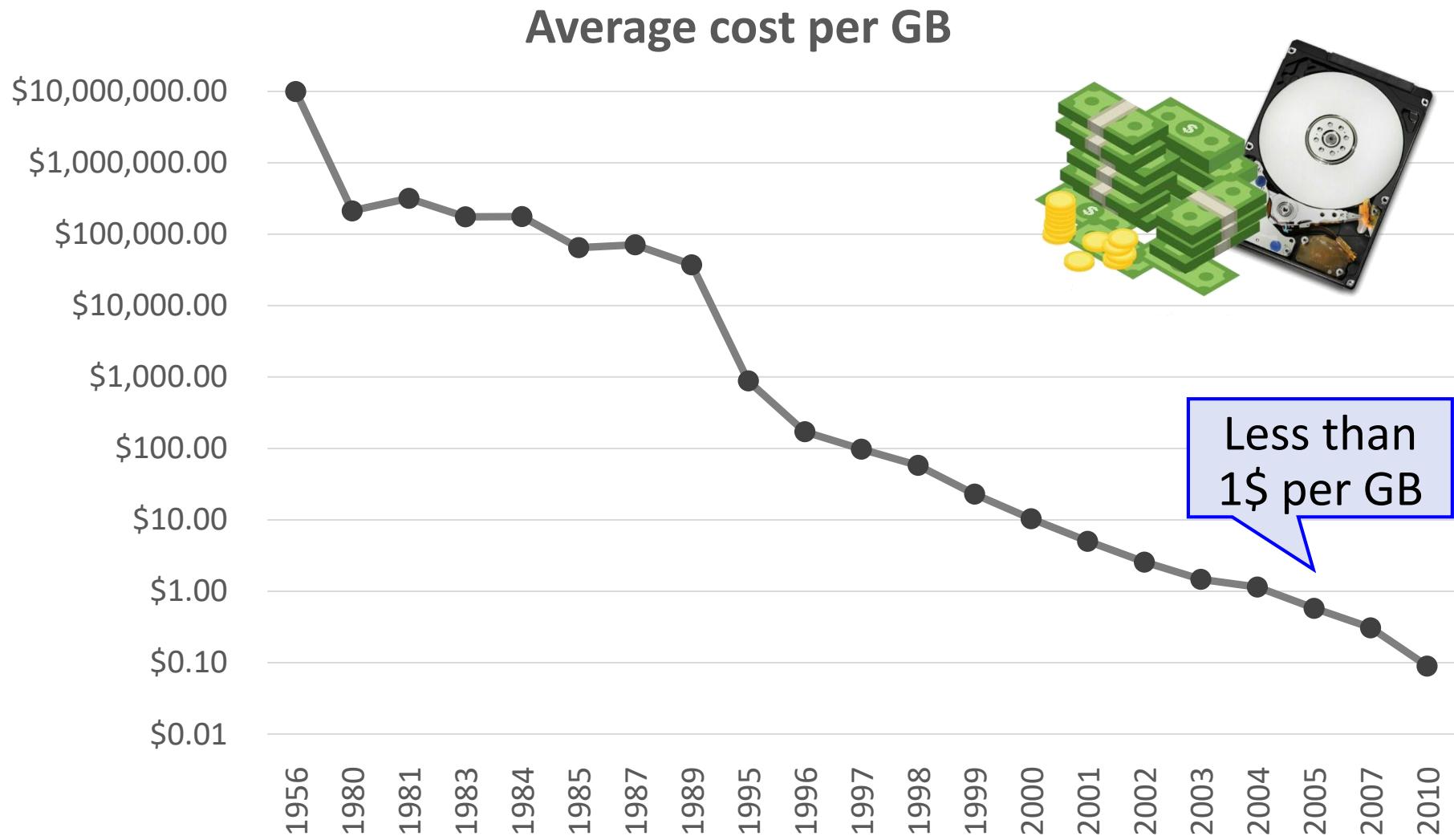


#1: Sizes Are Still Increasing

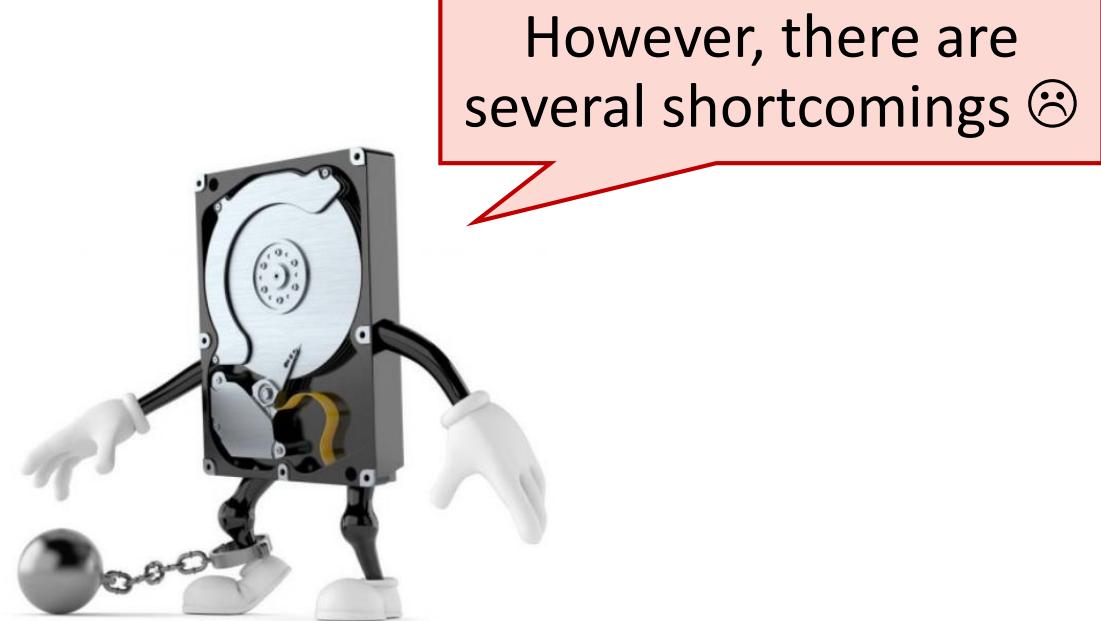
Average capacities of HDDs (GBs per drive)



#2: Costs Are Still Decreasing

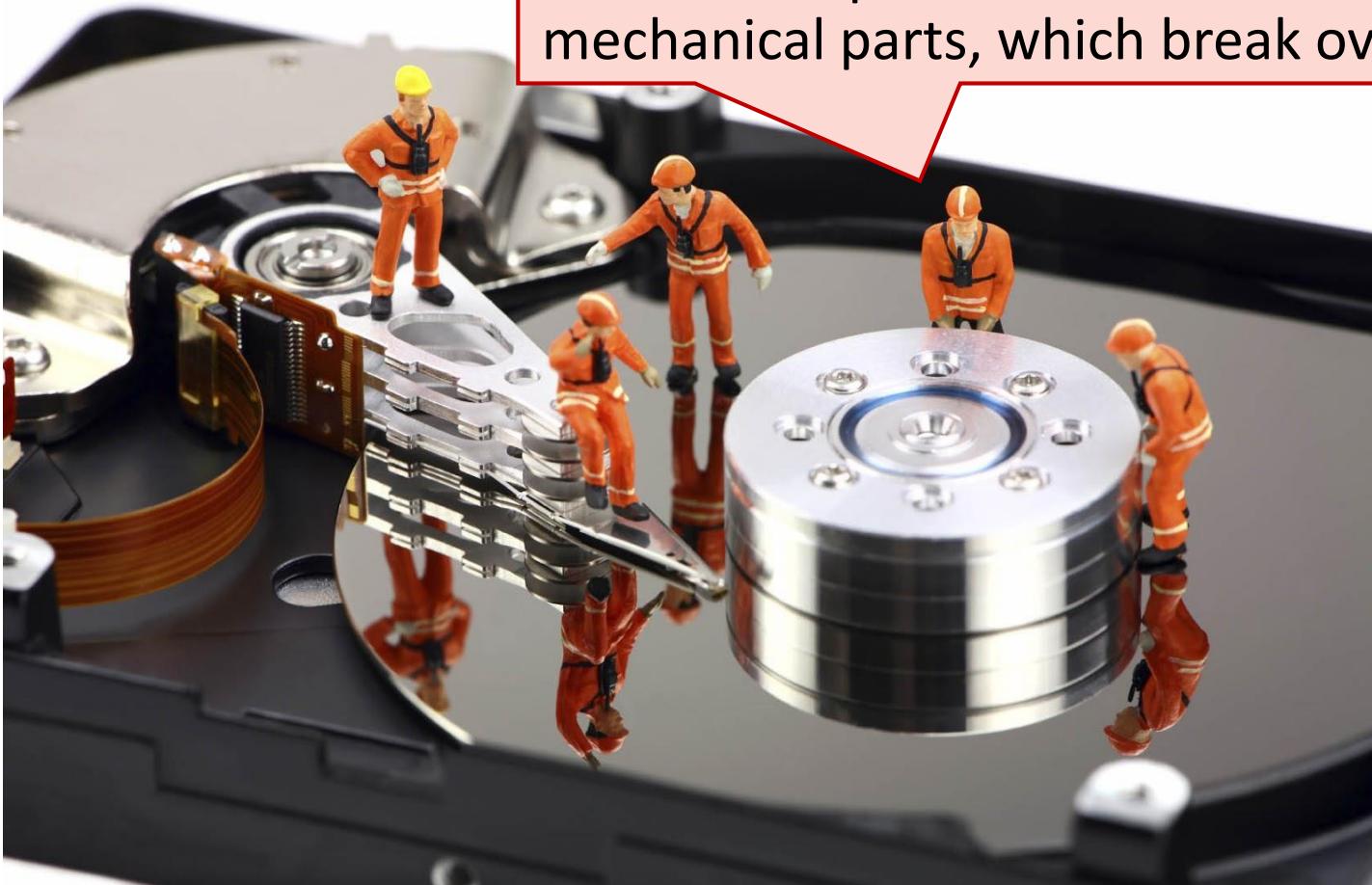


Shortcomings of Spinning Disks



#1: Disk Failure

In modern systems, hard drives are only the component that maintaining mechanical parts, which break over time



#1: Disk Failure

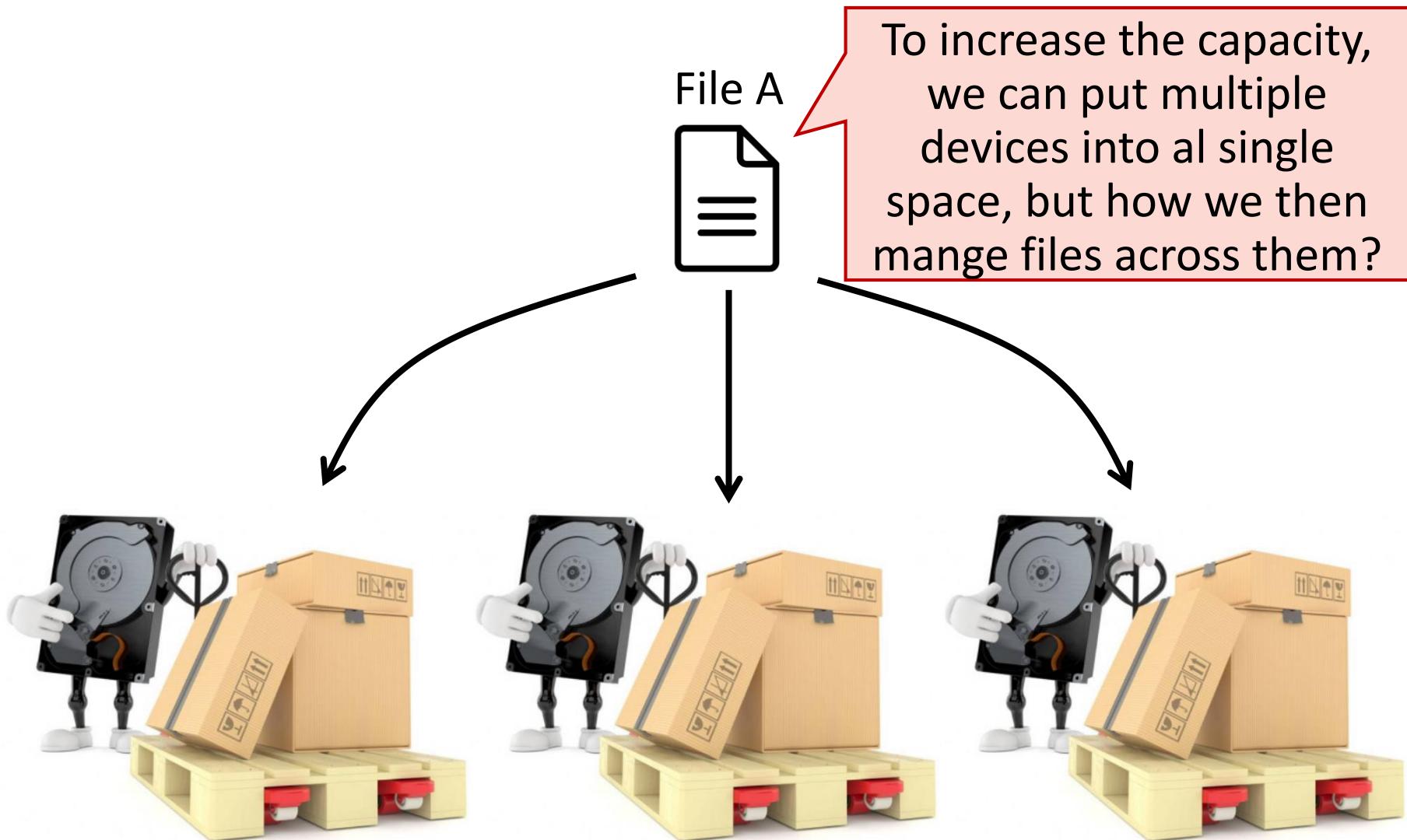


#2: Capacity is Limited

The capacity of each device
is yet not sufficient to
accommodate all the files
you have



#2: Capacity is Limited



Solution: Beyond Single Disks

RAID: Redundant Arrays of Inexpensive Disks



RAID

- RAID: Redundant Array of Inexpensive Disks

Use multiple “inexpensive” disks to create the illusion of a single **large, faster, more reliable disk**



A Case for Redundant Arrays of Inexpensive Disks (RAID)

David A. Patterson, Garth Gibson, and Randy H. Katz

Computer Science Division
Department of Electrical Engineering and Computer Sciences
571 Evans Hall
University of California
Berkeley, CA 94720
(pattrsn@ginger.berkeley.edu)

Abstract. Increasing performance of CPUs and memories will be squandered if not matched by a similar performance increase in I/O. While the capacity of Single Large Expensive Disks (SLED) has grown rapidly, the performance improvement of SLED has been modest. Redundant Arrays of Inexpensive Disks (RAID), based on the magnetic disk technology developed for personal computers, offers an attractive alternative to SLED, promising improvements of an order of magnitude in performance, reliability, power consumption, and scalability. This paper introduces five levels of RAIDs, giving their relative cost/performance, and compares RAID to an IBM 3380 and a Fujitsu Super Eagle.

ure of magnetic disk technology is the growth in the maximum number of bits that can be stored per square inch, or the bits per inch in a track times the number of tracks per inch. Called M.A.D., for maximal areal density, the "First Law in Disk Density" predicts [Frank87]:

$$MAD = 10(\text{Year}-1971)/10$$

Magnetic disk technology has doubled capacity and halved price every three years, in line with the growth rate of semiconductor memory, and in practice between 1967 and 1979 the disk capacity of the average IBM data processing system more than kept up with its main memory [Stevens81].



External Viewpoint of RAID

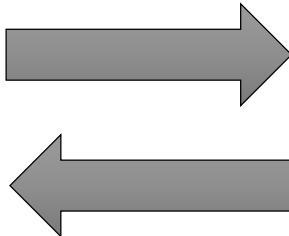
The main design policy of RAID is **transparency**; rather than managing files individually, making the disk bundles as a single disk





External Viewpoint of RAID

Data blocks can be read/written **as usual** (no need for SW to explicitly manage multiple disks or perform error checking/recovery)



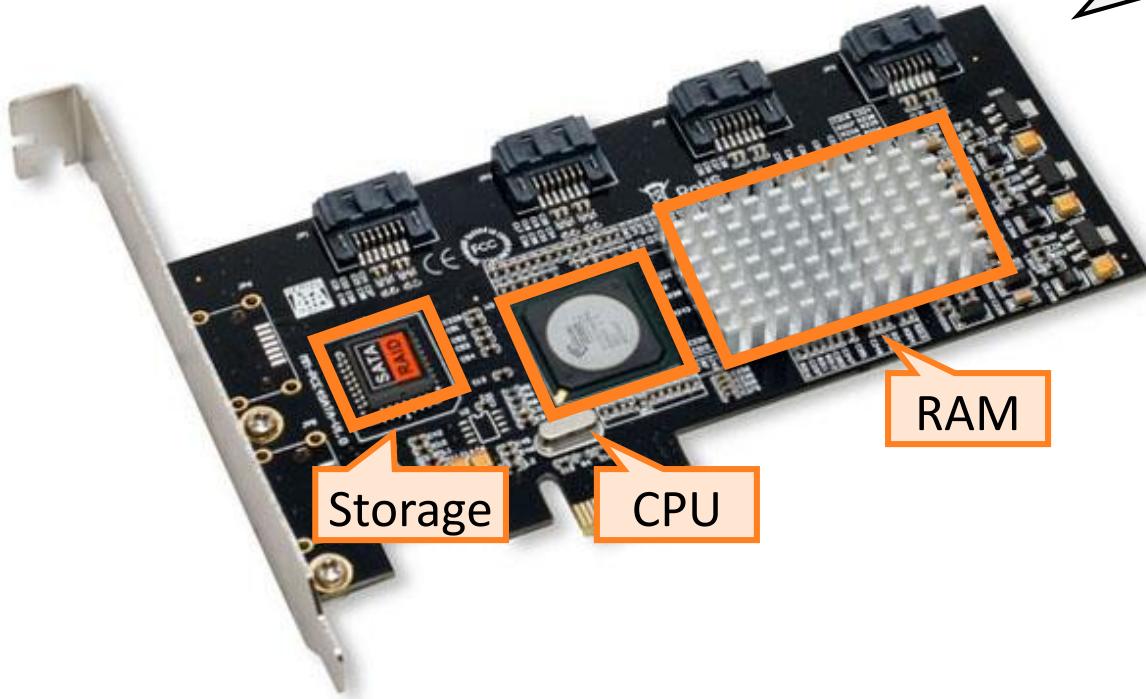
RAID





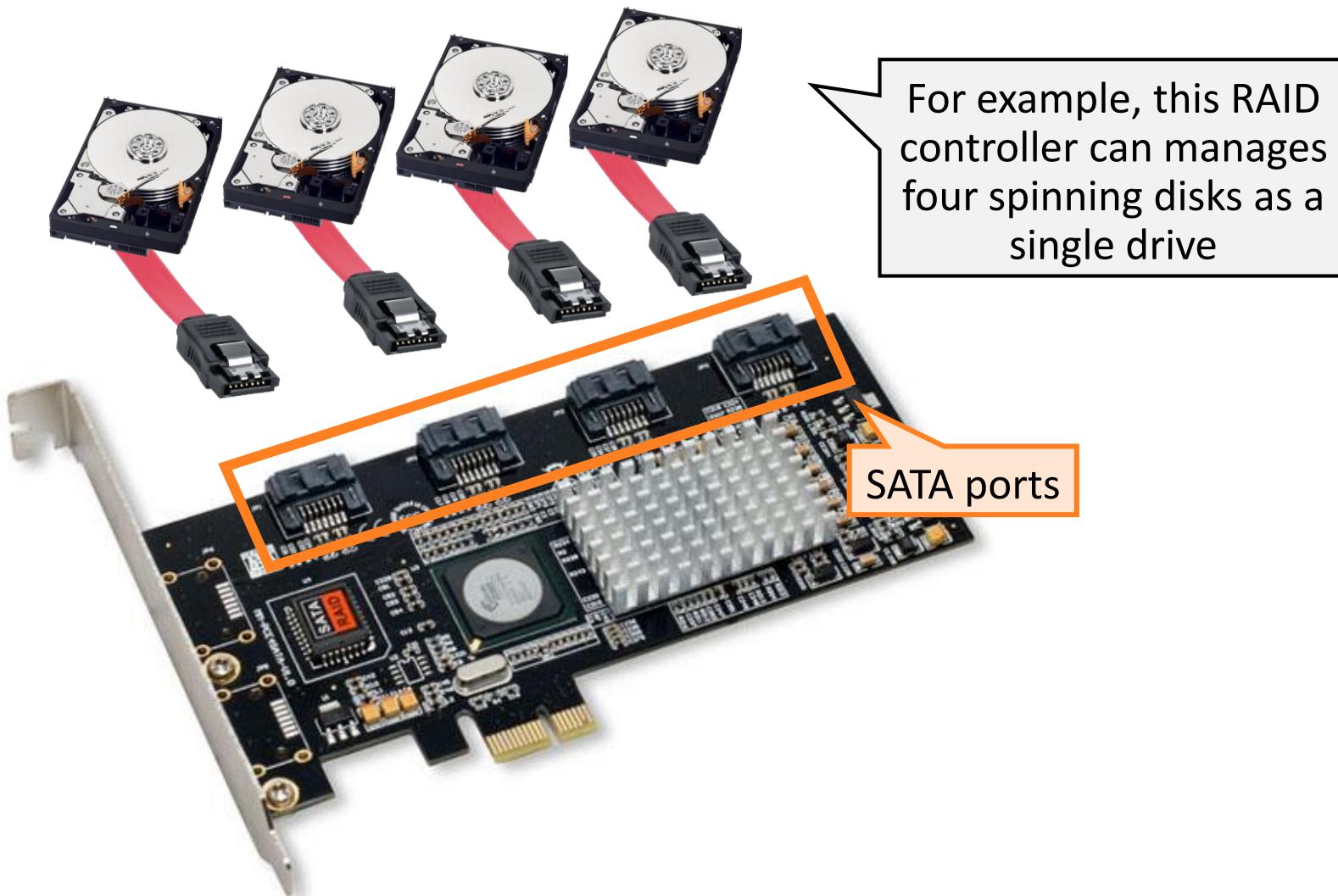
Internal Viewpoint of RAID

Internally, RAID is a **complex** computer system.
Multiple disks can be managed by a dedicated CPU





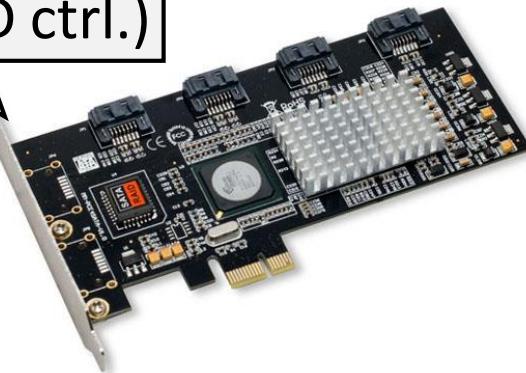
Internal Viewpoint of RAID



HW RAID vs. SW RAID

- RAID also can be implemented in software
(Implemented as OS service)

HW RAID
(RAID ctrl.)

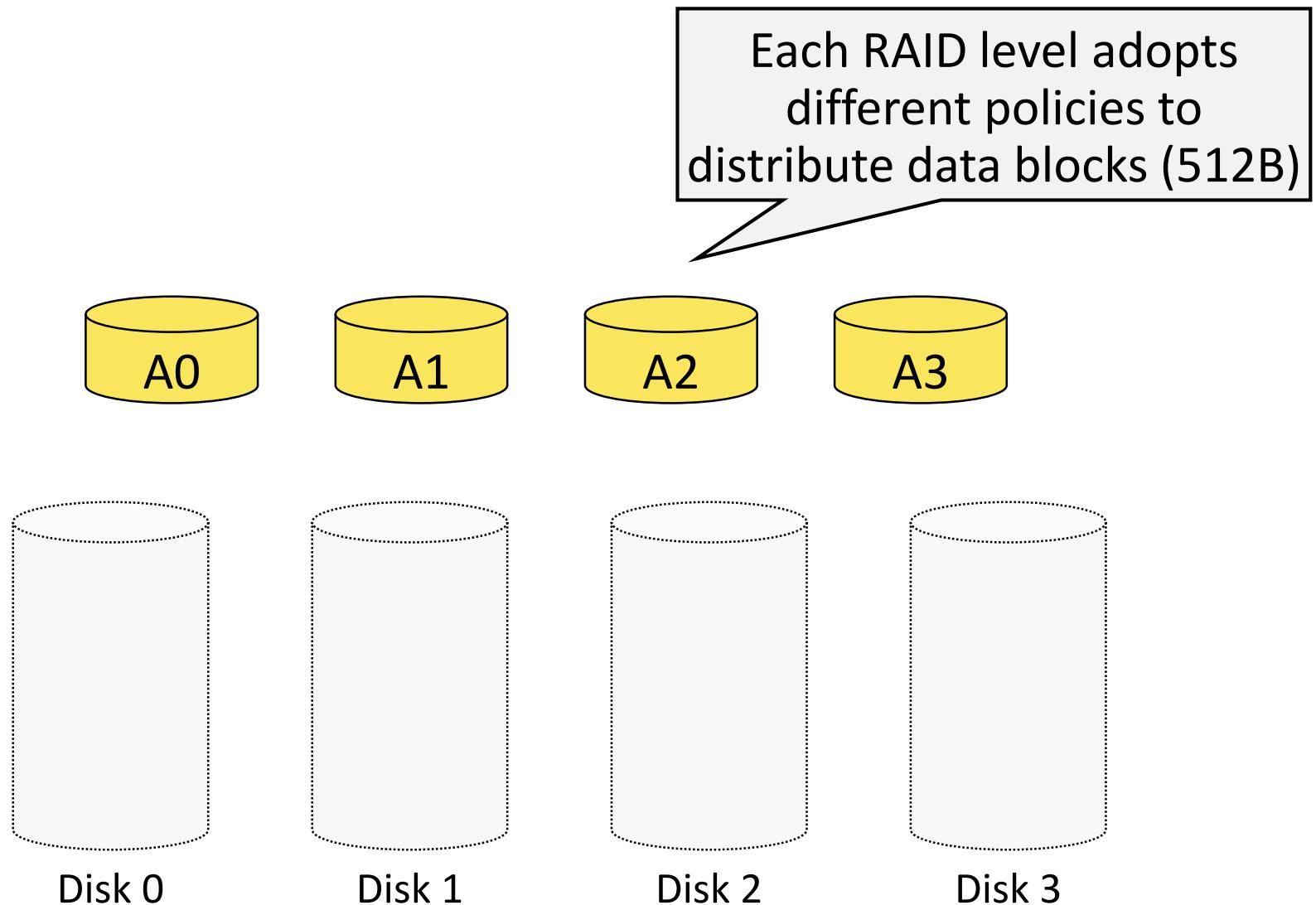


SW RAID
(mdraid)



Pros	<ul style="list-style-type: none">• Faster and more reliable than SW RAID	<ul style="list-style-type: none">• Simpler to migrate• Cheaper than HW-based RAID
Cons	<ul style="list-style-type: none">• Migrating a HW RAID array to a different HW controller almost never works	<ul style="list-style-type: none">• Worse performance and weaker reliability

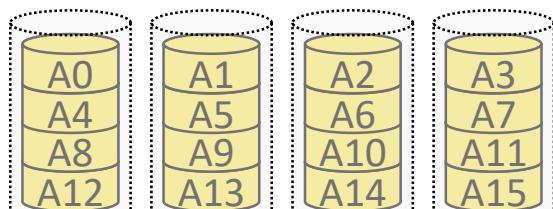
Configuration Options (RAID Levels)



Representative RAID Levels

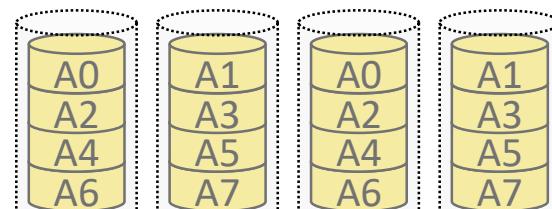
Let's check the different configurations of RAID one by one

RAID 0



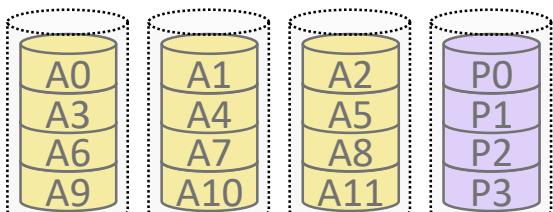
Disk 0 Disk 1 Disk 2 Disk 3

RAID 0+1



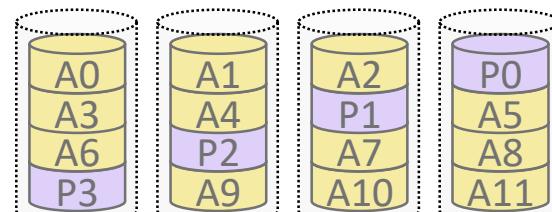
Disk 0 Disk 1 Disk 2 Disk 3

RAID 4



Disk 0 Disk 1 Disk 2 Disk 3

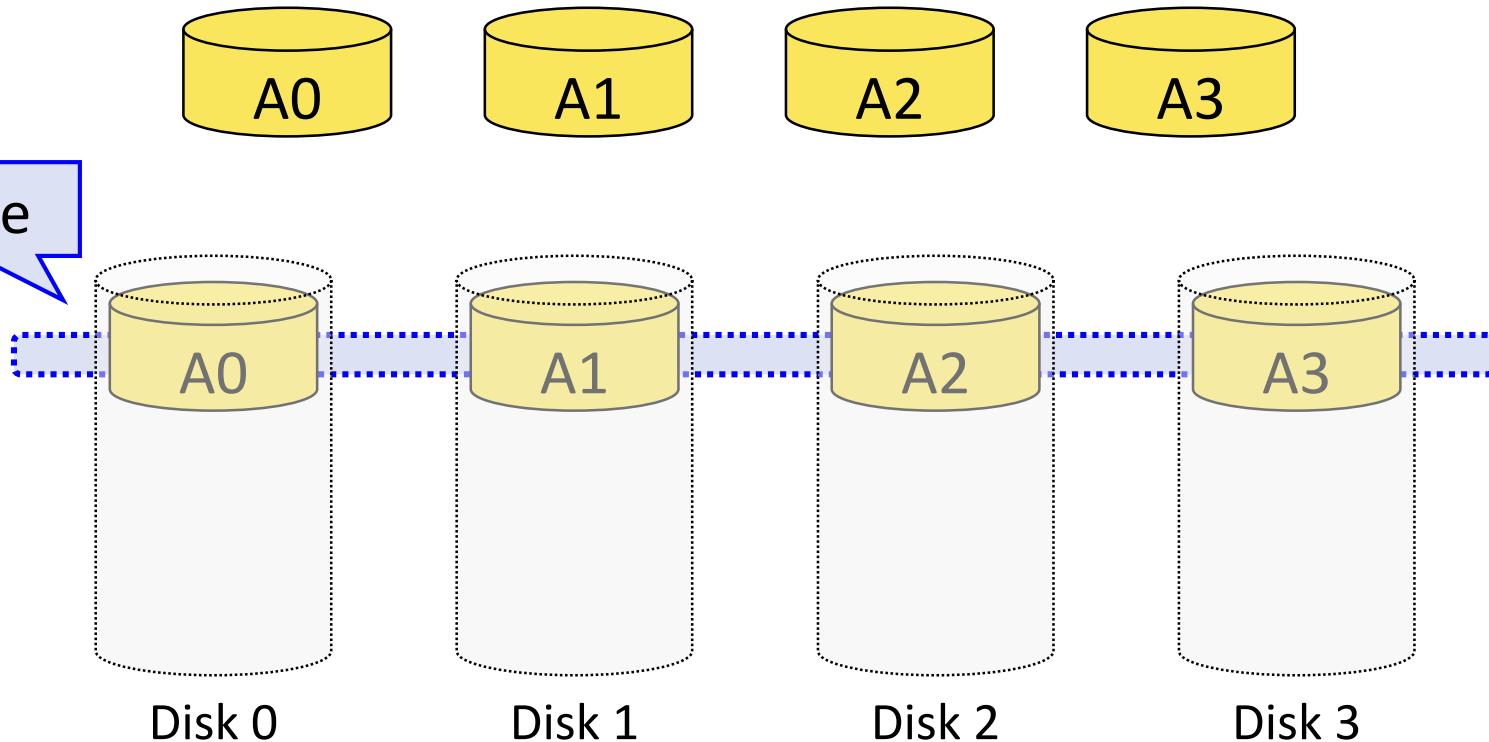
RAID 5



Disk 0 Disk 1 Disk 2 Disk 3

RAID 0 (Striping)

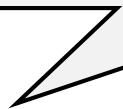
- **Key idea:** present a simple array of disks as a single large disk (nothing in there but can maximize parallelism)





Analysis of RAID 0

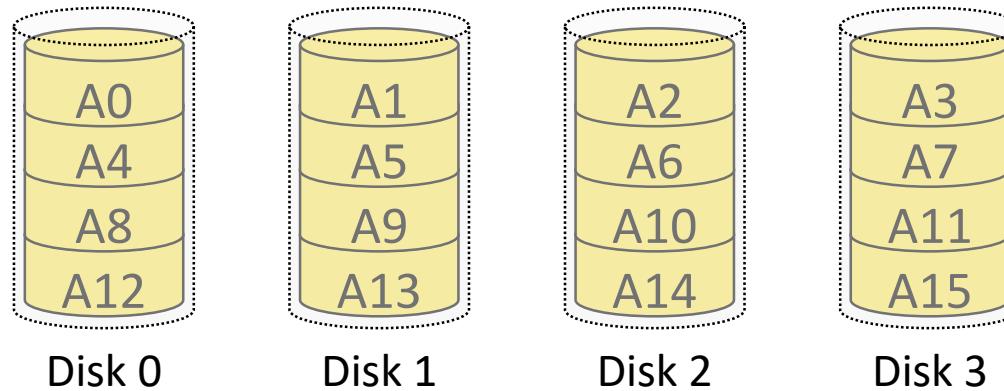
Let's analyze the different levels of RAID in terms of the capacity, reliability, and performance (throughput)



Capacity	
Reliability	
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0



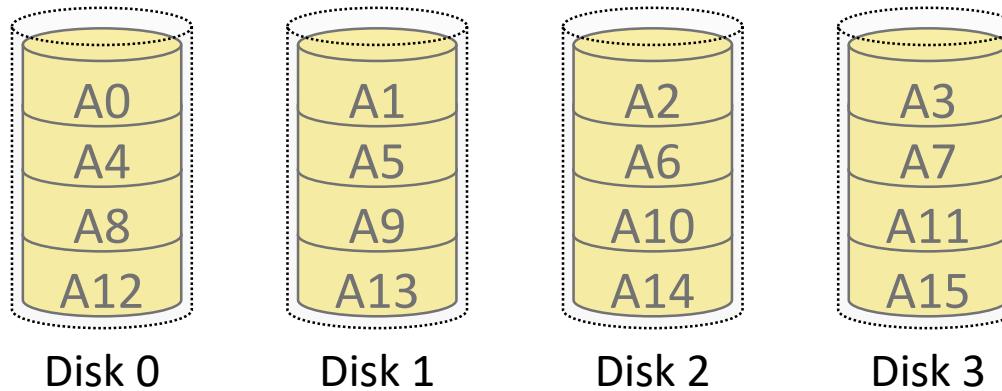
All storage spaces of the disks can be used for data

Capacity	N
Reliability	
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW



Analysis of RAID 0



How can we access data blocks with the configuration of multiple disks?

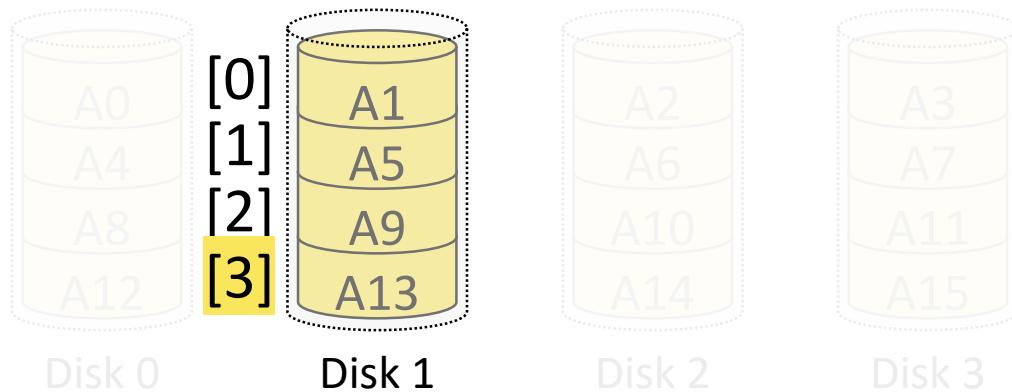


Disk = logical block number % number of disks
Offset = logical block number / number of disks

Capacity	N
Reliability	
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0



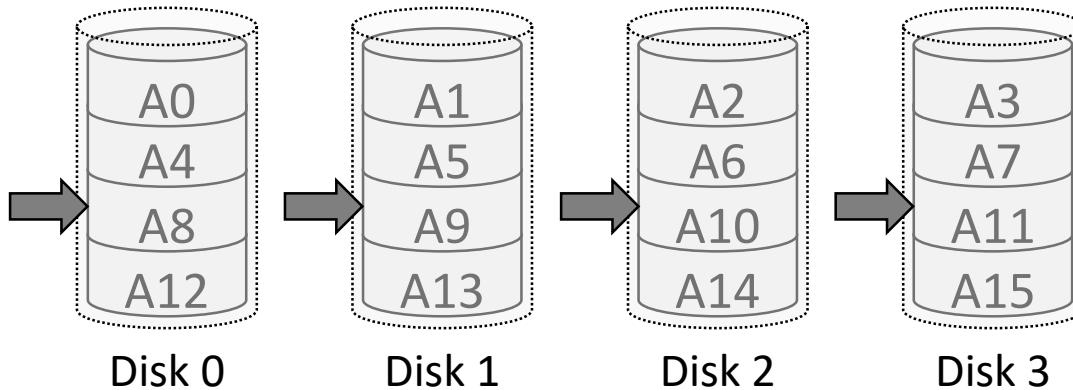
Disk = logical block number % number of disks
Offset = logical block number / number of disks

Capacity	N
R	
Sequential	
Random	

Example: read block 13 (A13)
→ $13 \% 4 = \text{Disk 1}$
→ $13 / 4 = \text{Physical block 3 (starting from 0)}$

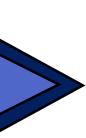
N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0

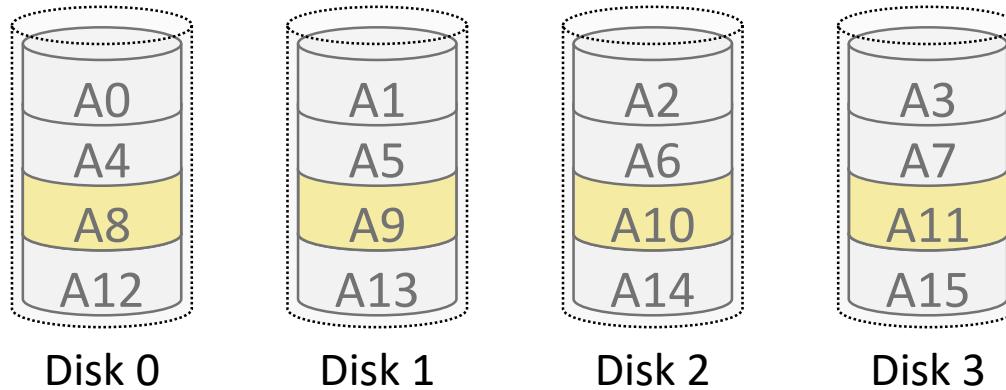


Capacity	What is the maximum bandwidth of RAID 0 with a sequential access pattern?	
Reliability		
Sequential read and write		
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW



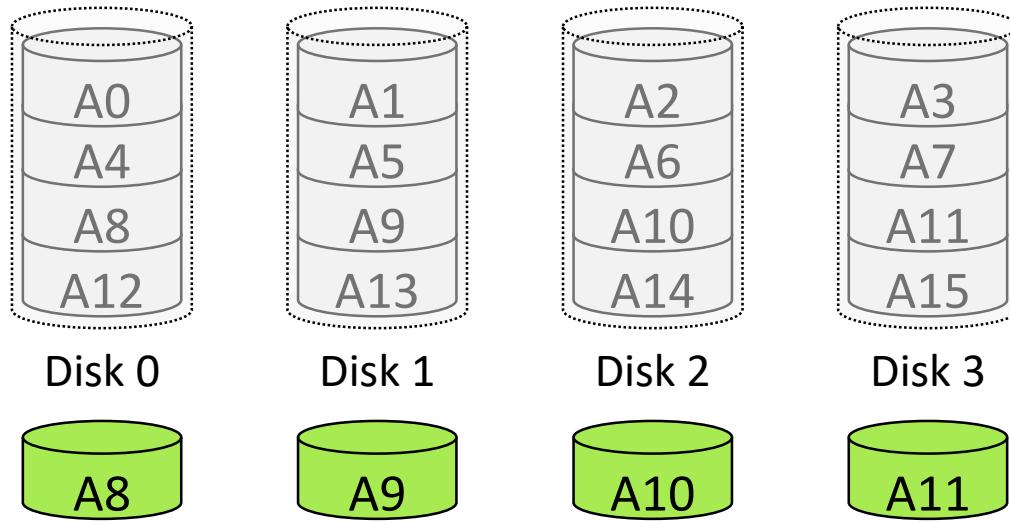
Analysis of RAID 0



Capacity	N	
Reliability		
Sequential read and write	$N * S$	
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0

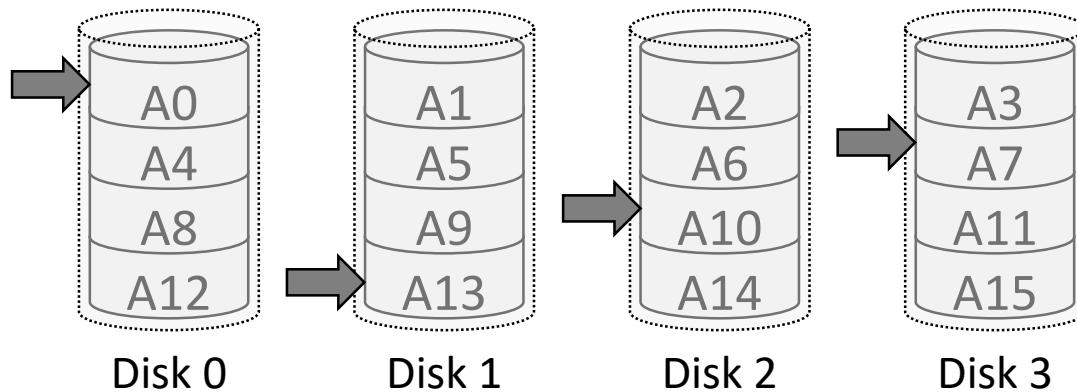


Capacity		
Reliability		
Sequential read and write	$N * S$	$N * S$
Random read and write		

Both sequential reads and writes can be fully parallelized across drives

N: number of disks, S: sequential access BW, R: random access BW

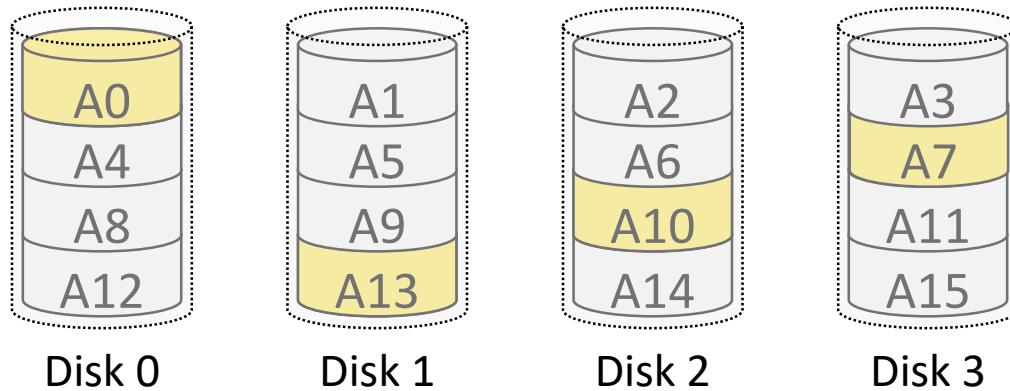
Analysis of RAID 0



Capacity	What is the maximum bandwidth of RAID 0 with a random access pattern?	
Reliability		
Sequential read and write		$N * S$
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

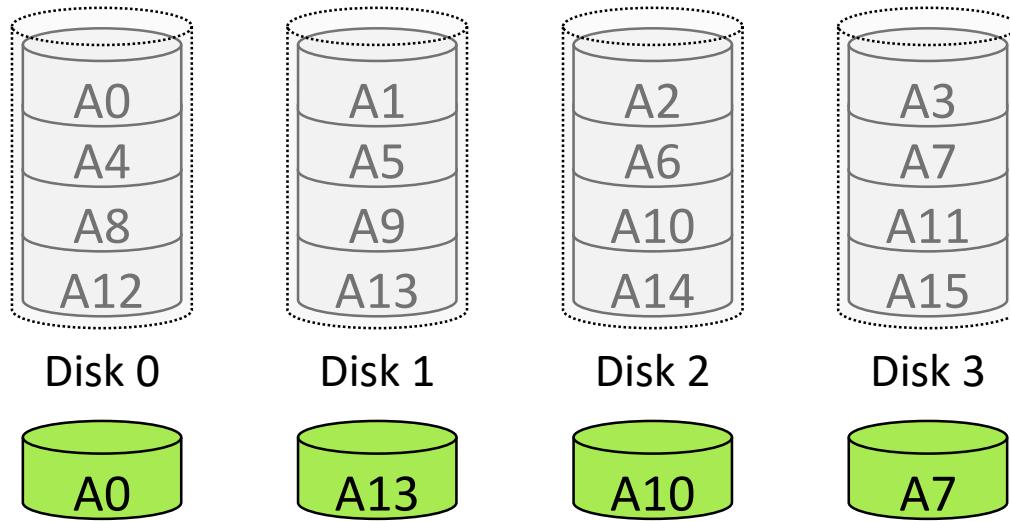
Analysis of RAID 0



Capacity	N	
Reliability		
Sequential read and write	$N * S$	$N * S$
Random read and write	$N * R$	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0



Capacity		
Reliability		
Sequential read and write	$N * S$	$N * S$
Random read and write	$N * R$	$N * R$

N: number of disks, S: sequential access BW, R: random access BW

Both random reads
and writes can be
naturally spread over
all drives

Analysis of RAID 0



Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate	50 MB/s

Let's calculate S and R with a real example

$$\text{Bandwidth} = \frac{\text{transfer_size}}{\text{time_to_access}} = \frac{\text{transfer_size}}{\text{seek_time} + \text{rotational_time} + \text{transfer_time}}$$

Analysis of RAID 0



Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate	50 MB/s

$$\text{Bandwidth} = \frac{\text{transfer_size}}{\text{time_to_access}} = \frac{\text{transfer_size}}{\text{seek_time} + \text{rotational_time} + \text{transfer_time}}$$

$$\text{transfer_time} = \frac{\text{transfer_size}}{\text{transfer_rate}}$$

Analysis of RAID 0



Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate	50 MB/s

$$\text{Bandwidth} = \frac{\text{transfer_size}}{\text{time_to_access}} = \frac{\text{transfer_size}}{\text{seek_time} + \text{rotational_time} + \text{transfer_time}}$$

Example: Access sequential **10MB** data

$$\text{transfer_time} = 10\text{MB}/(50\text{MB/s}) = 200\text{ms}$$

$$\begin{aligned}\text{Bandwidth}(S) &= \frac{10\text{MB}}{7\text{ms} + 3\text{ms} + 200\text{ms}} \\ &= 47.62 \text{ MB/s}\end{aligned}$$

Analysis of RAID 0



Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate	50 MB/s

$$\text{Bandwidth} = \frac{\text{transfer_size}}{\text{time_to_access}} = \frac{\text{transfer_size}}{\text{seek_time} + \text{rotational_time} + \text{transfer_time}}$$

Example: Access sequential 10MB data

$$\text{transfer_time} = 10\text{MB}/(50\text{MB/s}) = 200\text{ms}$$

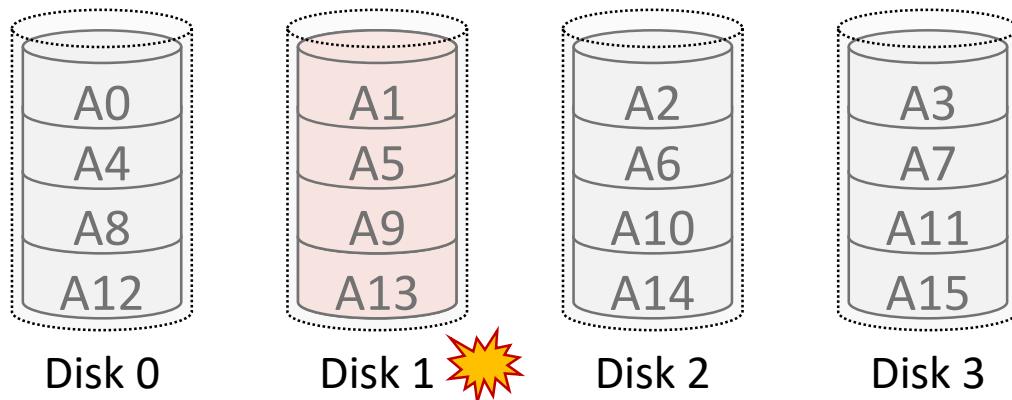
$$\begin{aligned}\text{Bandwidth}(S) &= \frac{10\text{MB}}{7\text{ms} + 3\text{ms} + 200\text{ms}} \\ &= 47.62 \text{ MB/s}\end{aligned}$$

Example: Access random 10KB data

$$\text{transfer_time} = 10\text{KB}/(50\text{MB/s}) = 0.2\text{ms}$$

$$\begin{aligned}\text{Bandwidth}(S) &= \frac{10\text{KB}}{7\text{ms} + 3\text{ms} + 0.2\text{ms}} \\ &= 0.98 \text{ MB/s}\end{aligned}$$

Analysis of RAID 0

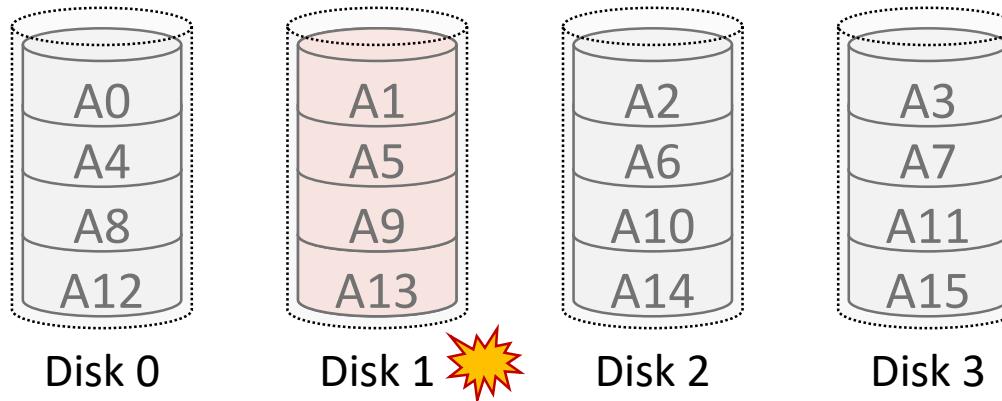


Capacity	N	
Reliability		
Sequential read and write	$N * S$	$N * S$
Random read and write	$N * R$	$N * R$

N: number of disks, S: sequential access BW, R: random access BW



Analysis of RAID 0



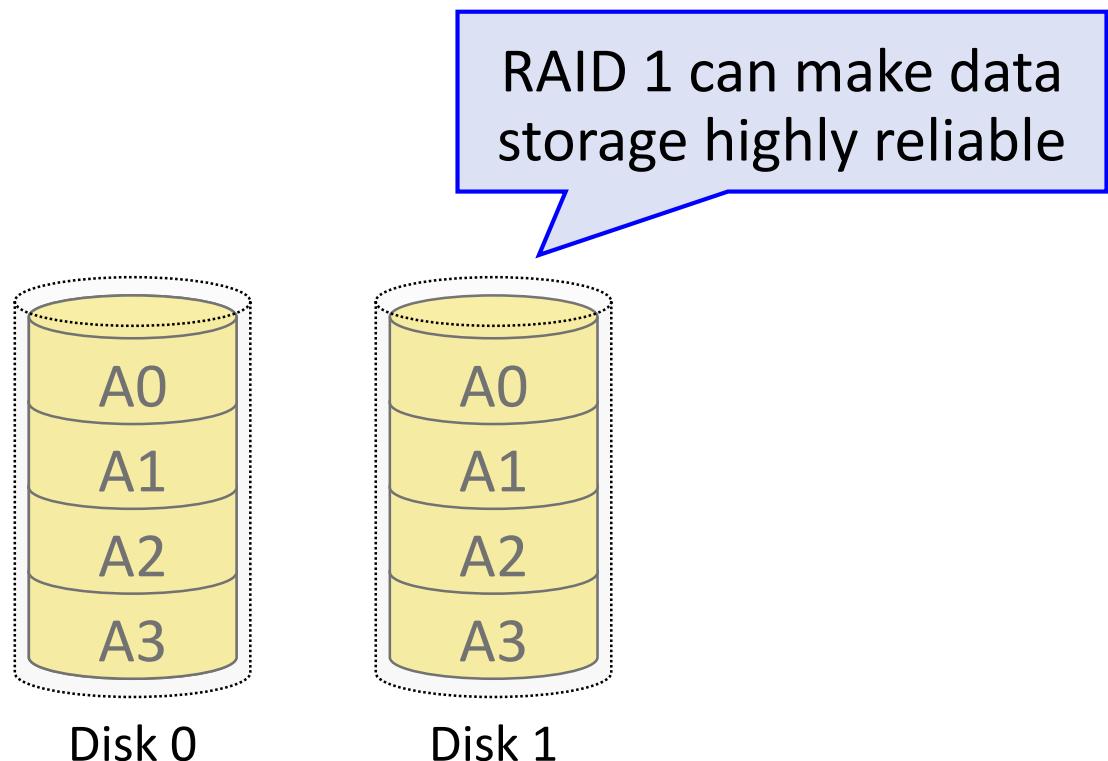
There is no way to recover the broken disk (**zero** error recovery)

Capacity	N	
Reliability	0	
Sequential read and write	$N * S$	$N * S$
Random read and write	$N * R$	$N * R$

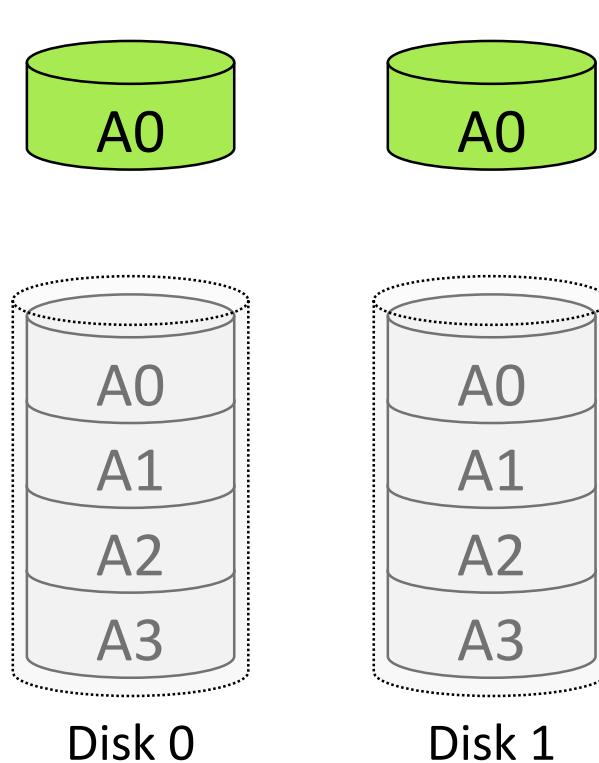
N: number of disks, S: sequential access BW, R: random access BW

RAID 1 (Mirroring)

- **Key idea:** make two copies of all data

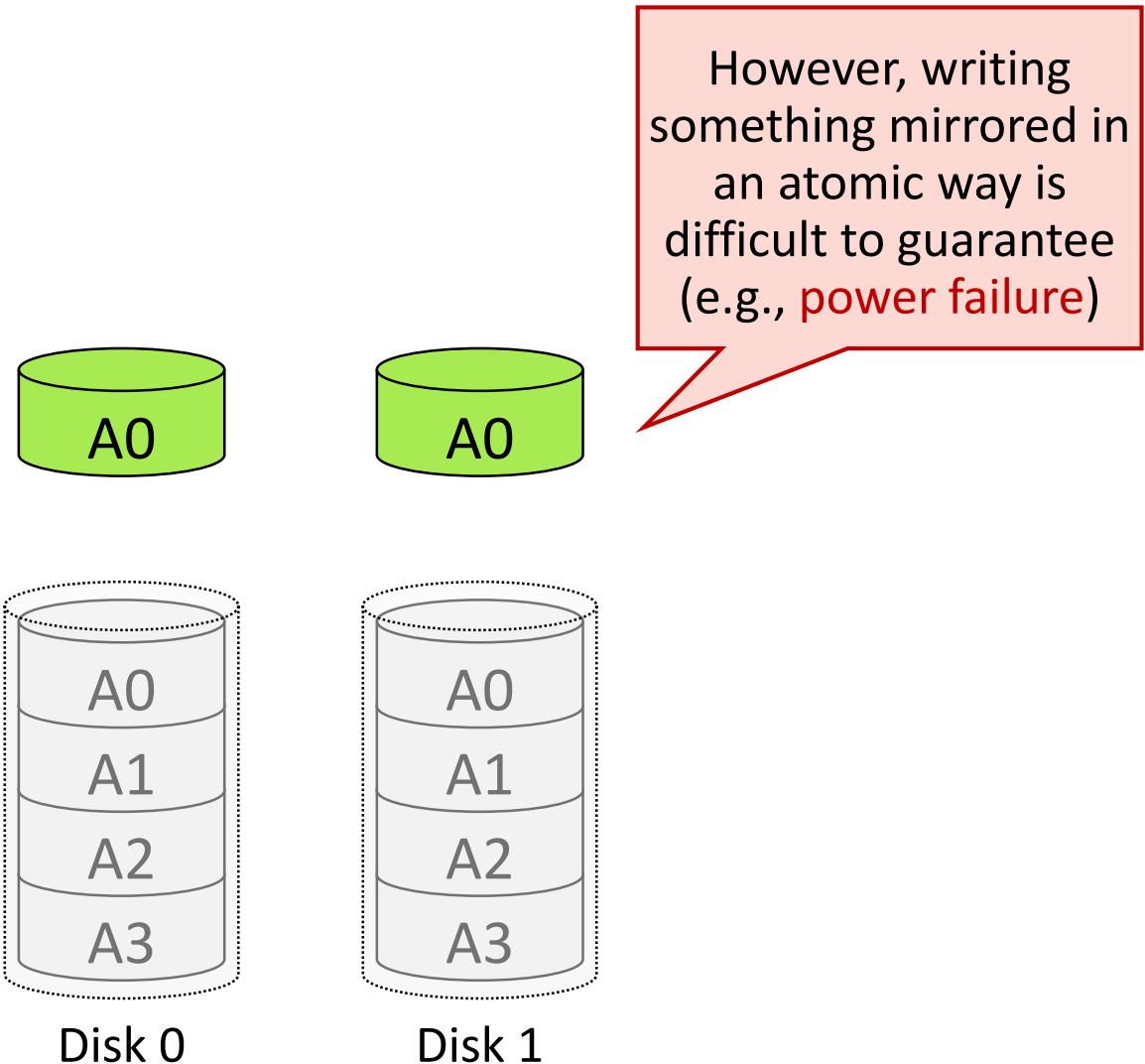


Consistent Update Problem

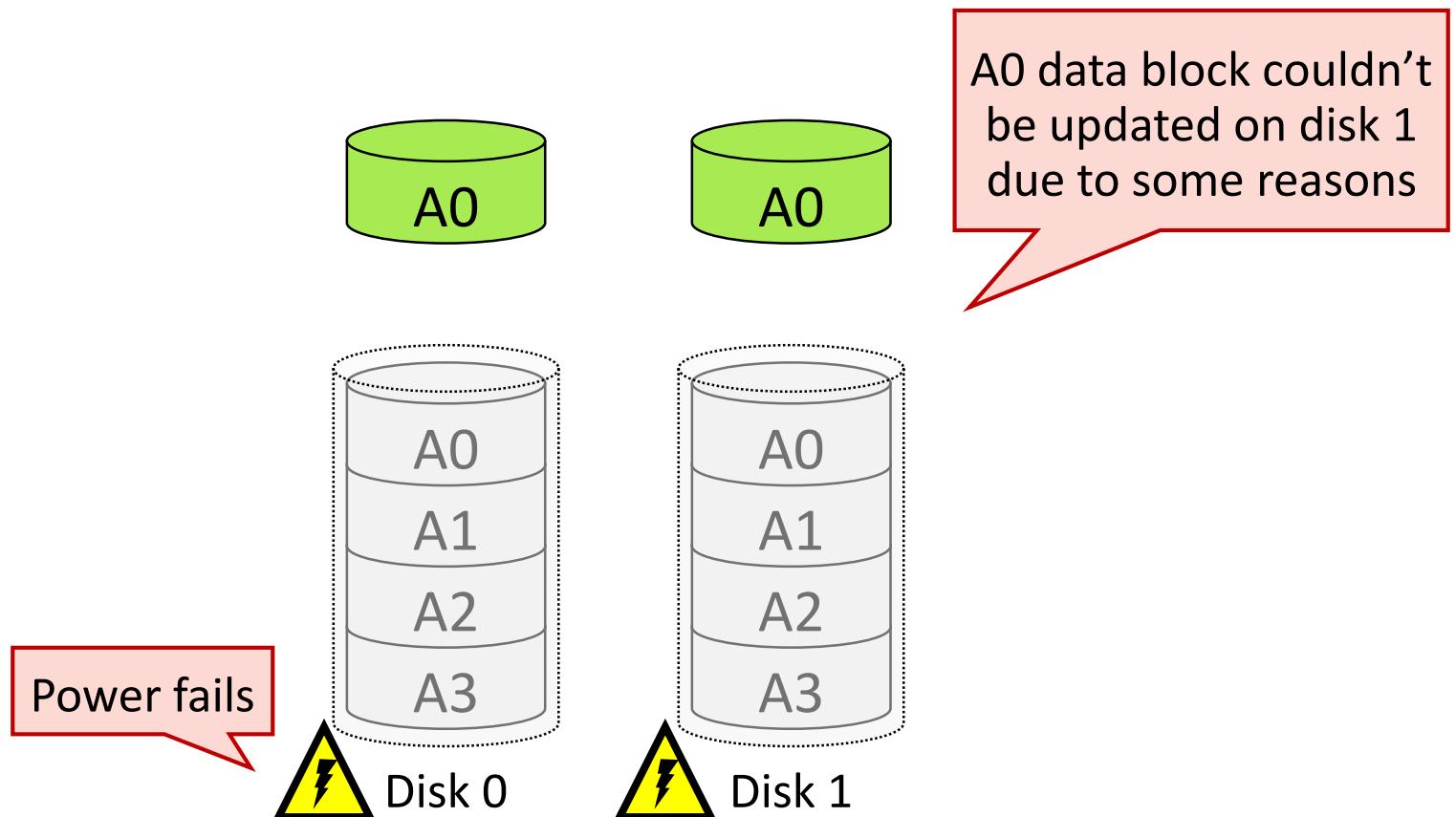


Mirrored writes should be **atomic** (all or nothing; all copies are written, or none are written)

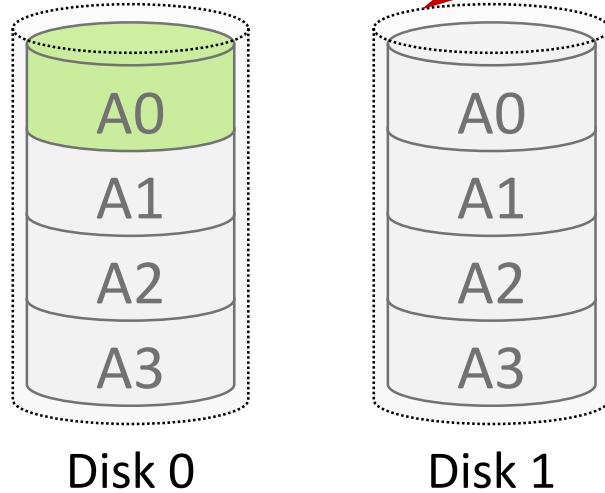
Consistent Update Problem



Consistent Update Problem

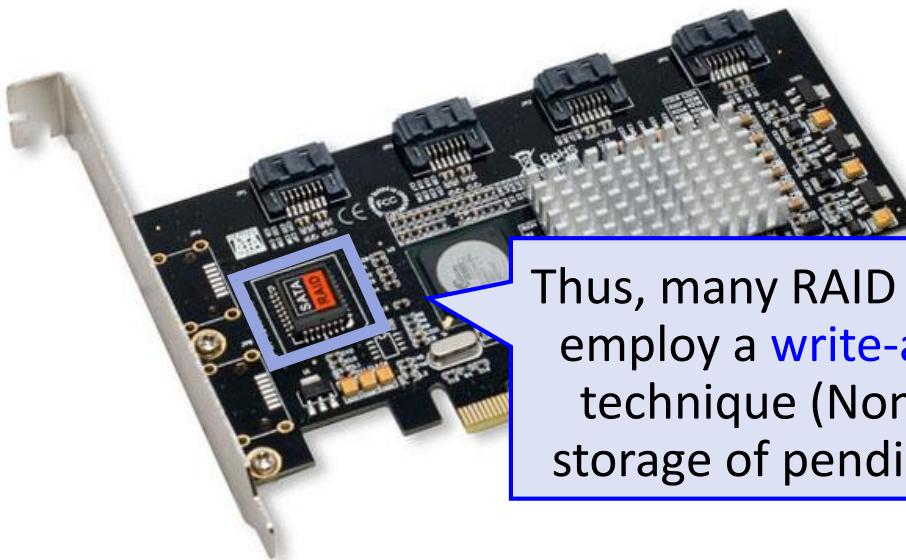


Consistent Update Problem

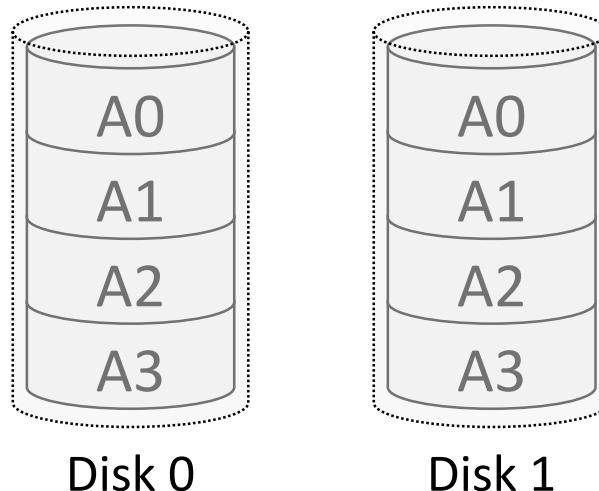


When a system reboots, it's going to get lost of A0. There is no A0 copy, but RAID cannot recognize this inconsistency

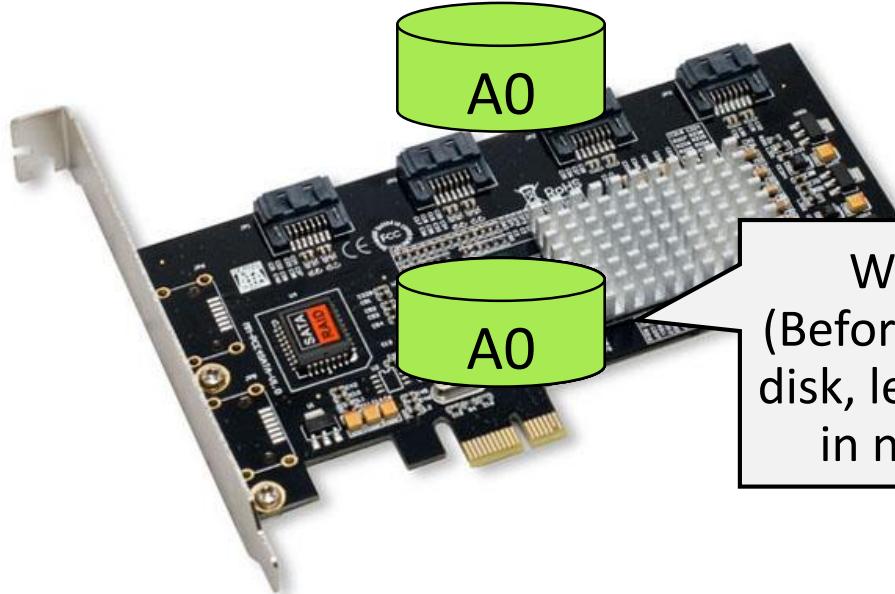
Consistent Update Problem



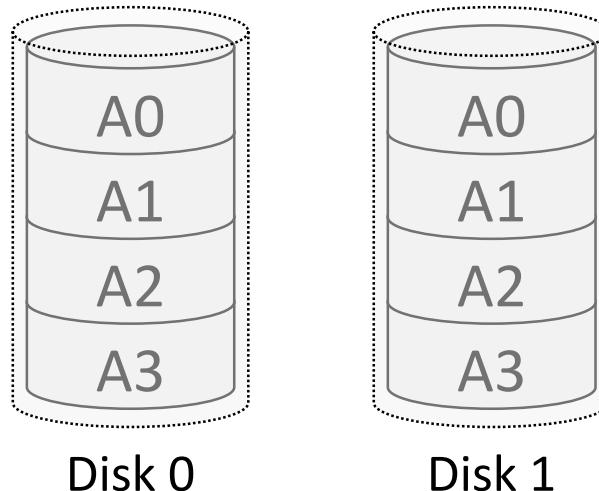
Thus, many RAID controllers employ a [write-ahead log](#) technique (Non-volatile storage of pending writes)



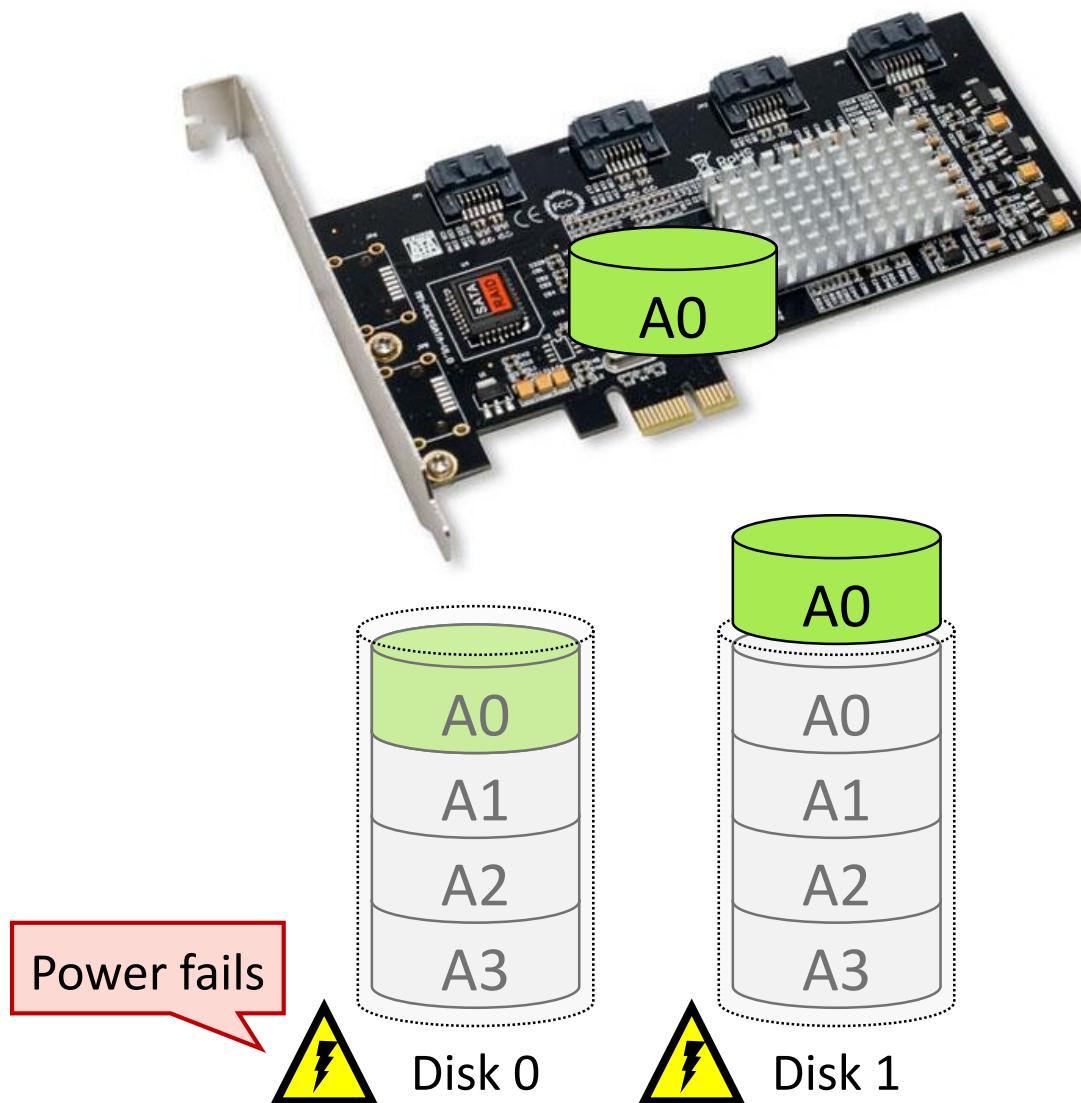
Consistent Update Problem



Write-ahead logging
(Before updating the data to disk, let's keep it in the built-in non-volatile storage)



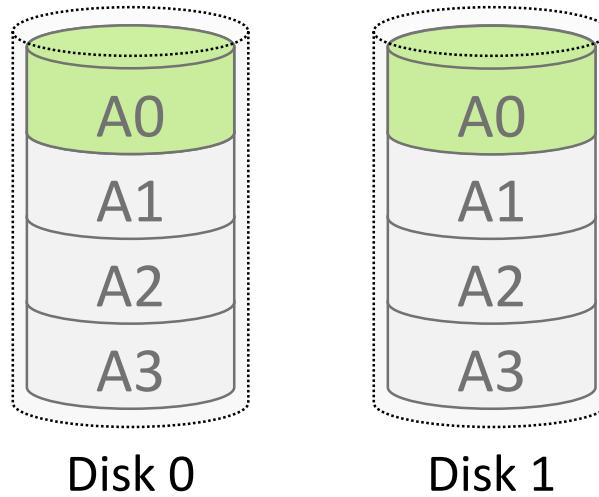
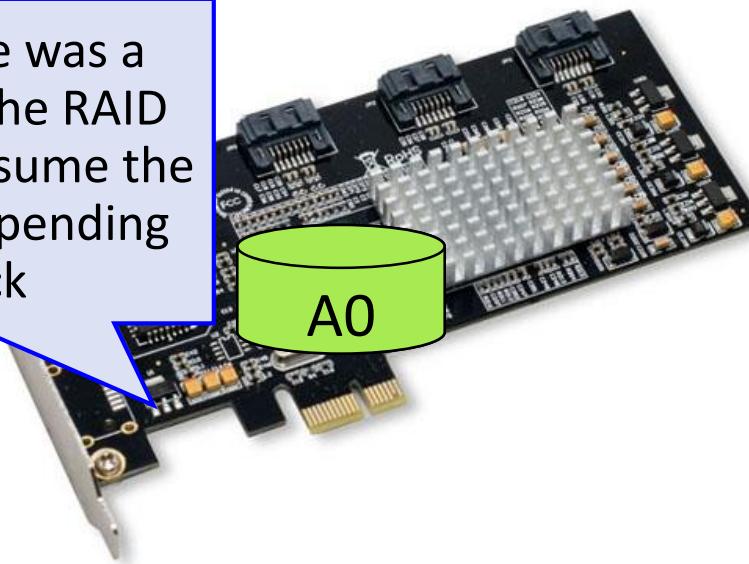
Consistent Update Problem





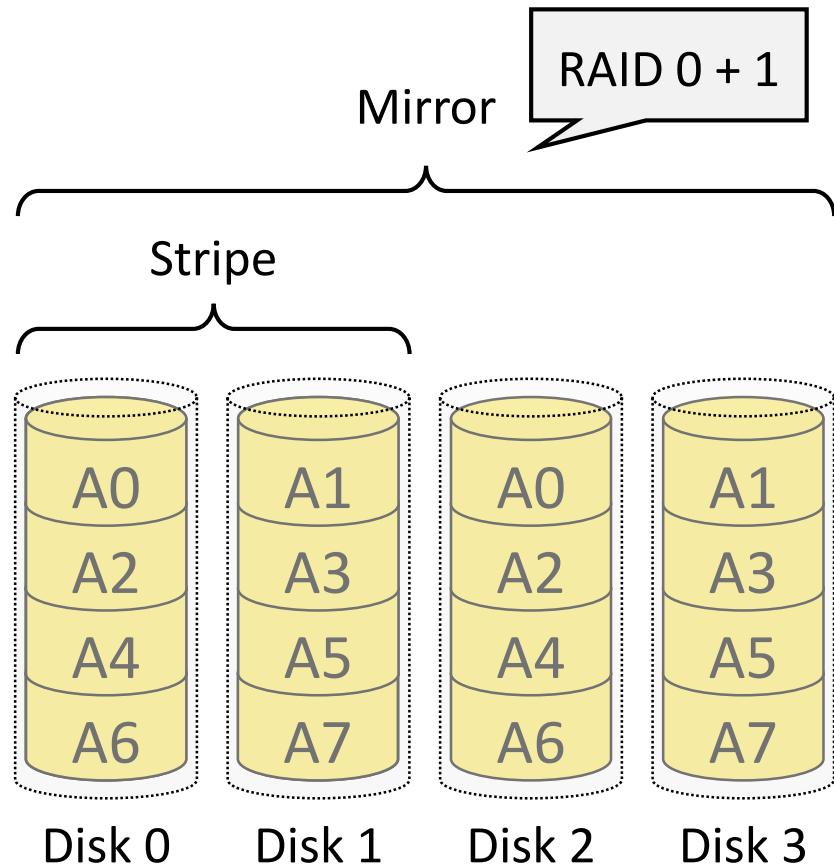
Consistent Update Problem

Although there was a power failure, the RAID controller can resume the update of such pending data block



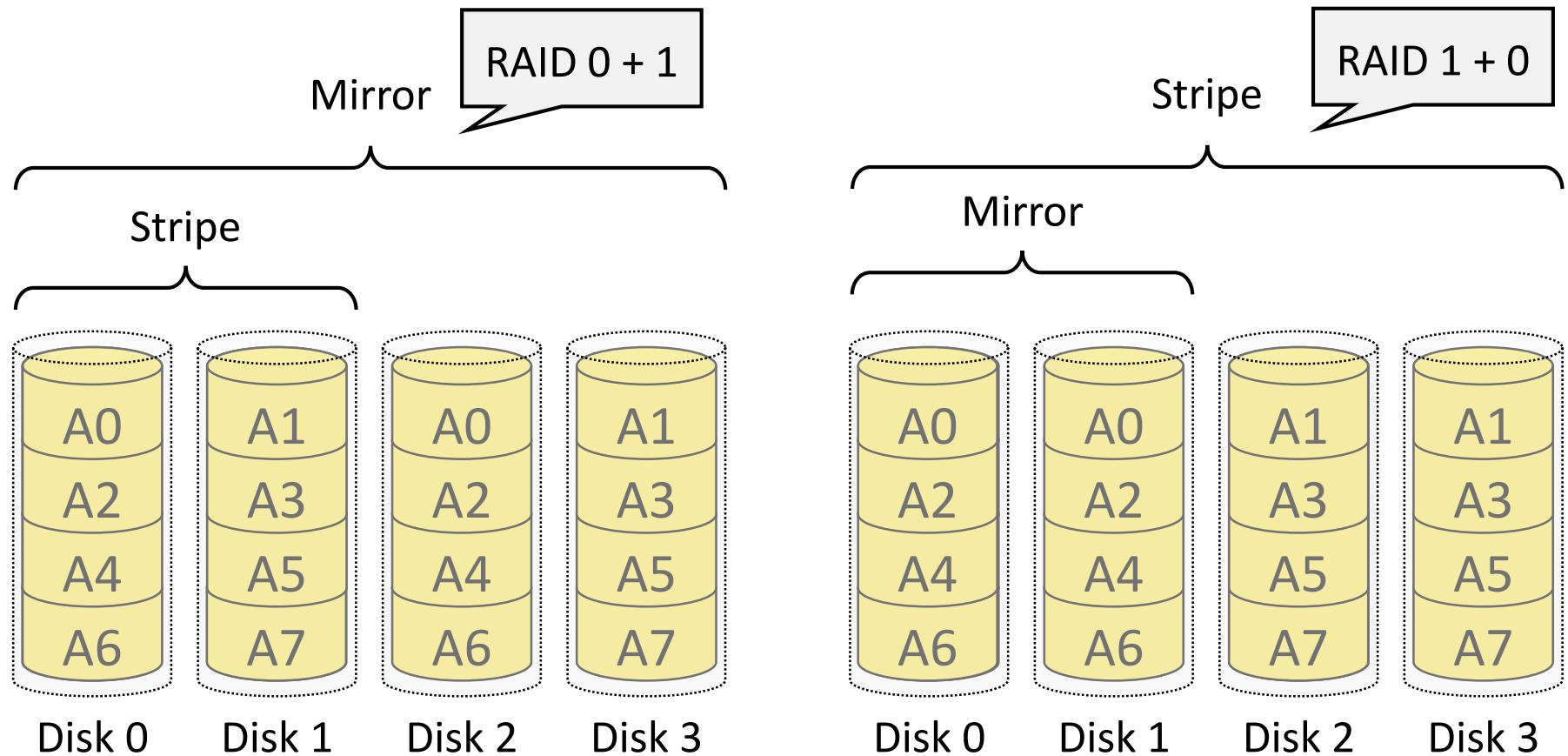
RAID 0+1 / RAID 1+0

- In many systems, striping (RAID 0) and mirroring (RAID 1) are often combined instead of mirroring only

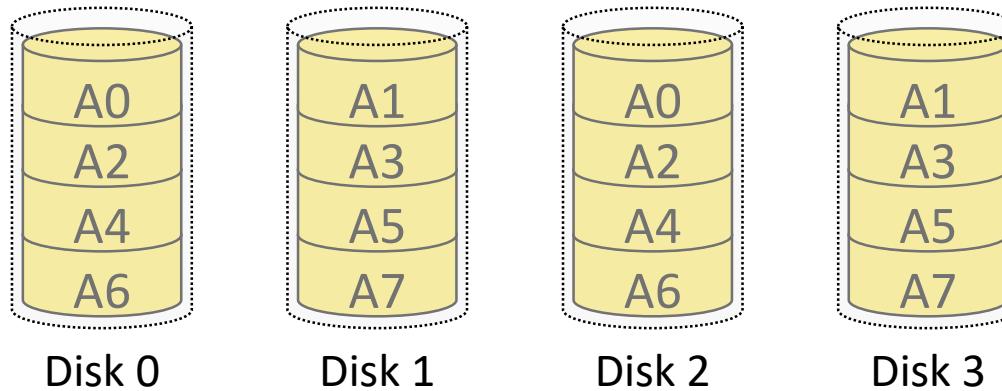


RAID 0+1 / RAID 1+0

- In many systems, striping (RAID 0) and mirroring (RAID 1) are often combined instead of mirroring only



Analysis of RAID 0+1

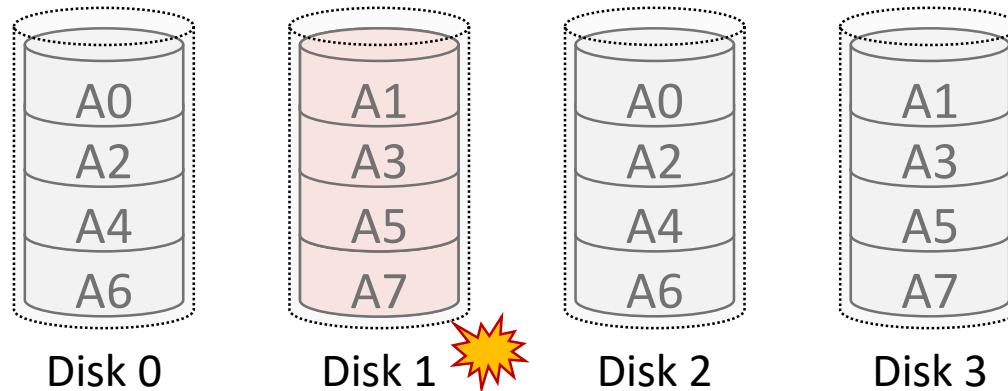


RAID 0+1 only can utilize half capacity

Capacity	$N/2$
Reliability	
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1



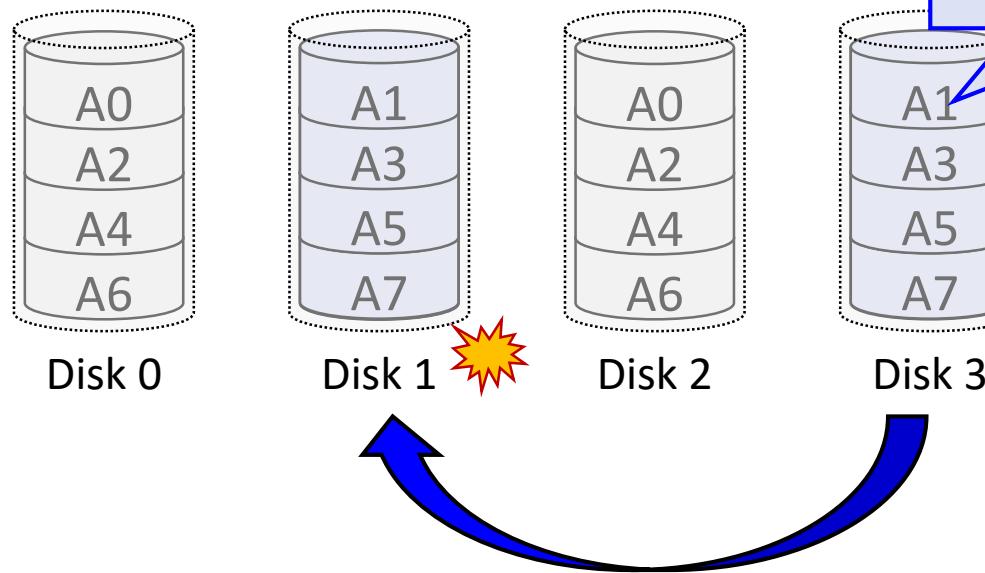
What happens
if **disk 1** is
broken?



Capacity	N/2	
Reliability		
Sequential read and write		
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1

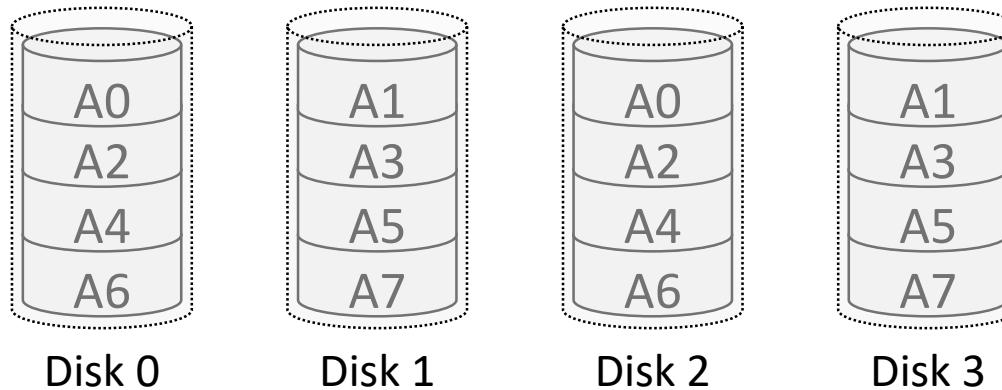


Capacity	$N/2$
Reliability	1 drive can fail
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW



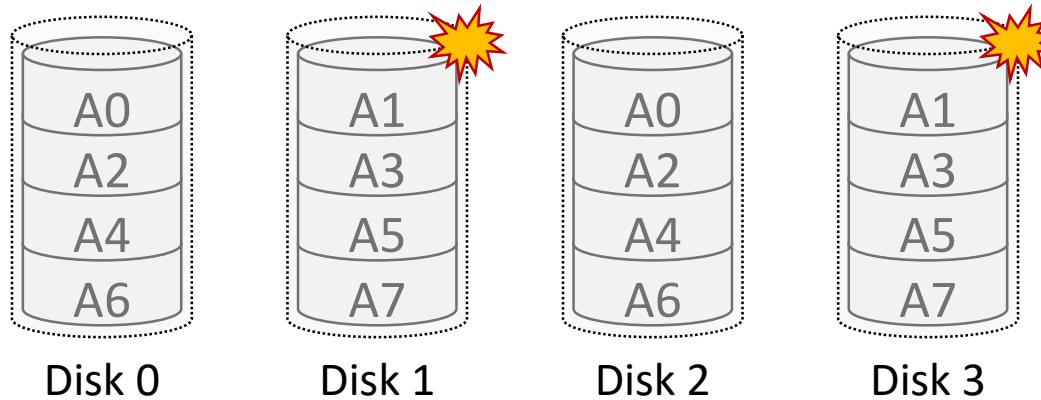
Analysis of RAID 0+1



Capacity	N/2	If you are lucky, N/2 drives can fail without a data loss
Reliability	1 drive can fail, sometime more	
Sequential read and write		
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1

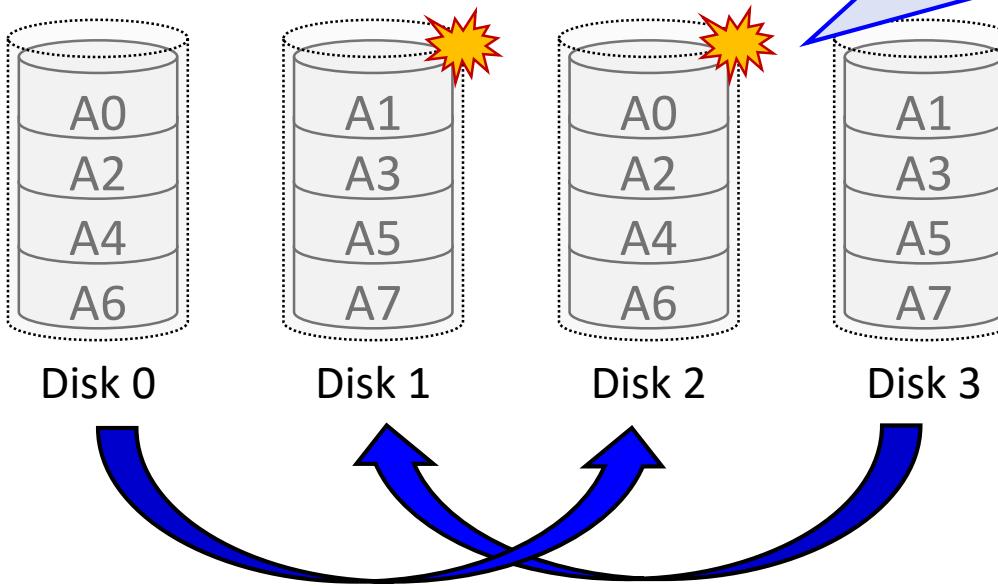


Capacity	N/2
Reliability	1 drive can fail, sometime more
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1

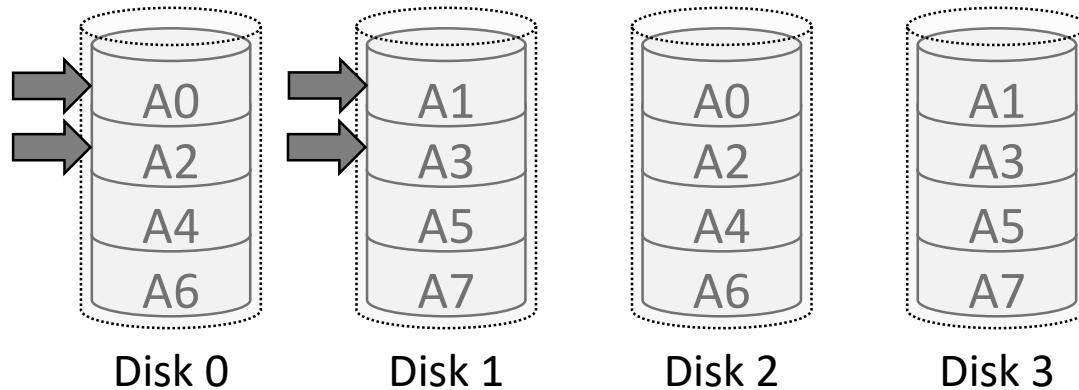
It's okay for Disk 1 & Disk 2 to fail in parallel though ($N/2$ drives)



Capacity	$N/2$
Reliability	1 drive can fail, sometime more
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

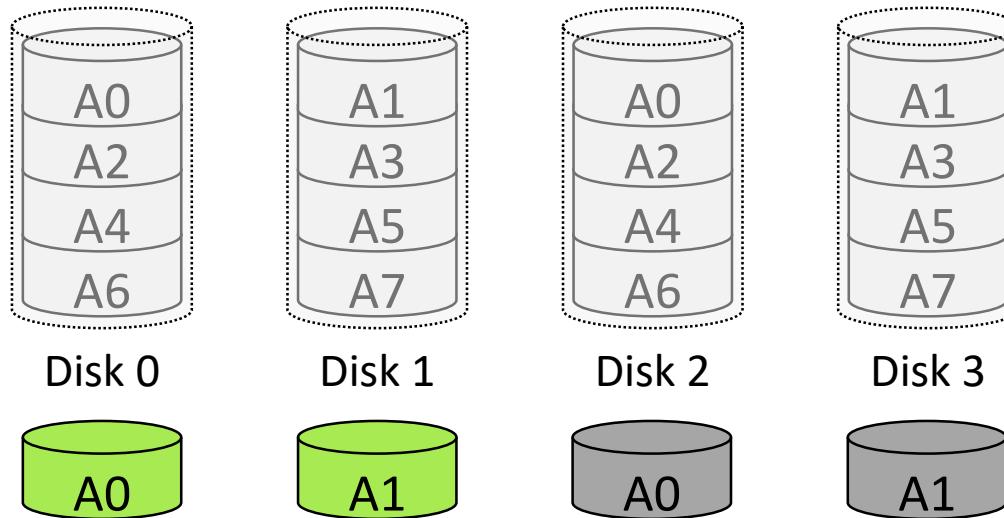
Analysis of RAID 0+1



Capacity	What is the maximum bandwidth of RAID 0+1 for sequential writes ?
Reliability	time more
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1

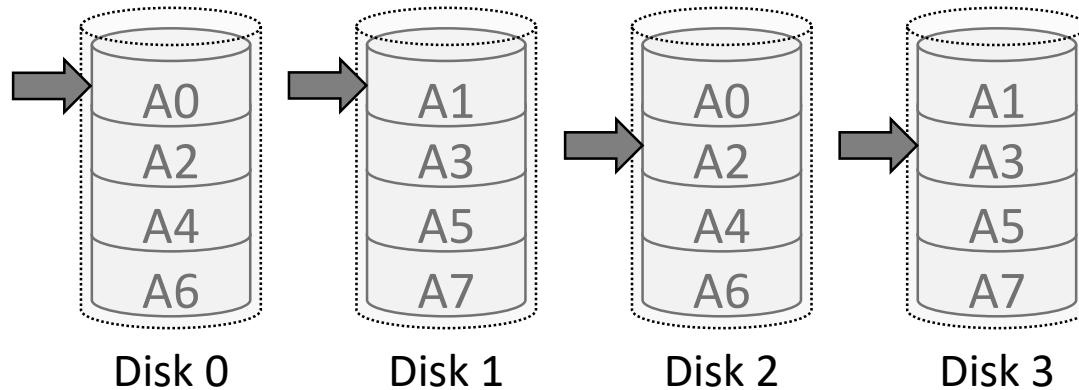


Capacity	N
Reliability	1 drive can fail,
Sequential read and write	$N/2 * S$
Random read and write	

$N/2$ drives update data block while other $N/2$ drives copy the data block (\therefore half throughput)

N: number of disks, S: sequential access BW, R: random access BW

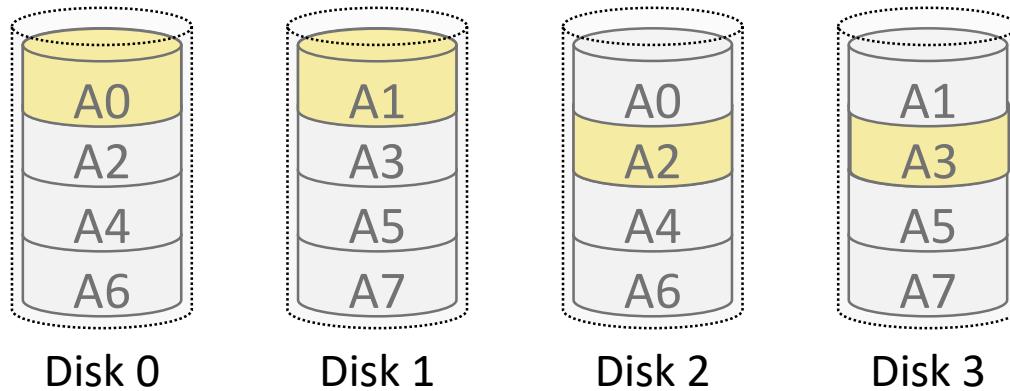
Analysis of RAID 0+1



Capacity	What is the maximum bandwidth of RAID 0+1 for sequential reads ?	
Reliability		time more
Sequential read and write		$N/2 * S$
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

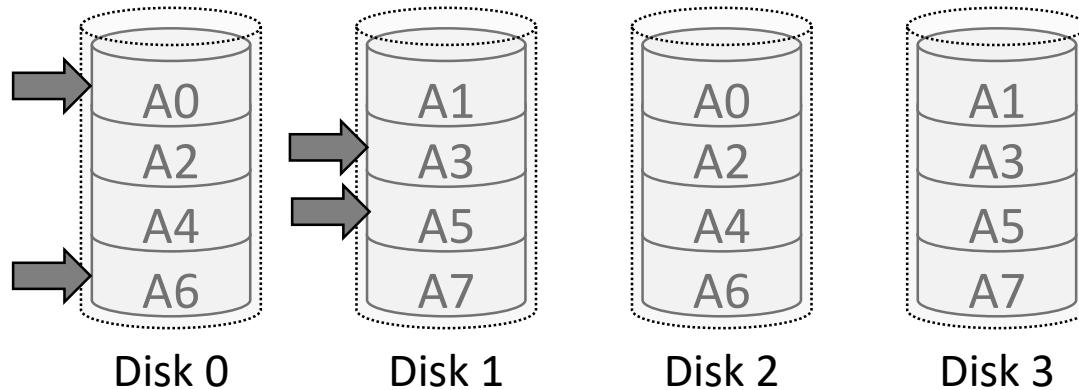
Analysis of RAID 0+1



Capacity		
Reliability	1 drive can fail, the more drives, the more reliable	
Sequential read and write	$N * S$	$N/2 * S$
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

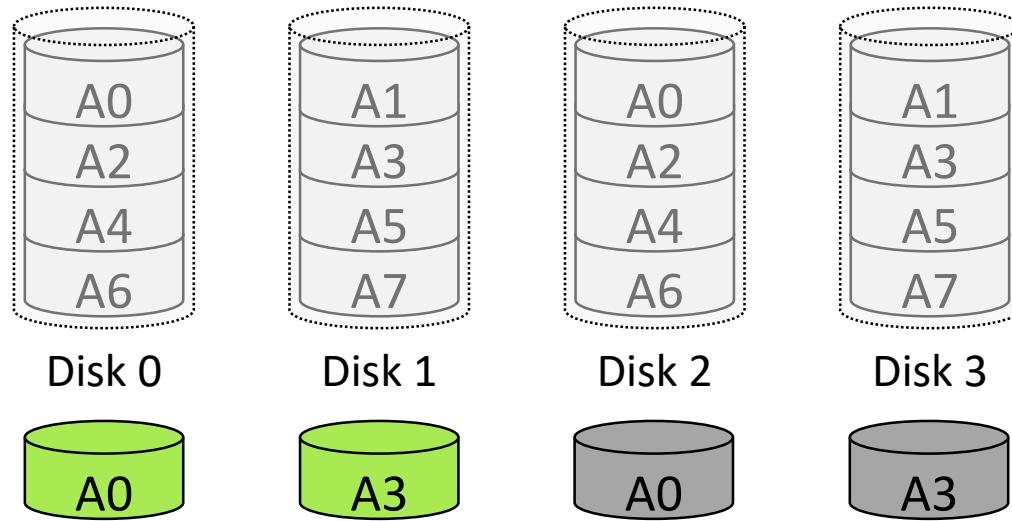
Analysis of RAID 0+1



Capacity	What is the maximum bandwidth of RAID 0+1 for random writes ?	
Reliability		Same more
Sequential read and write		$N * S$
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1

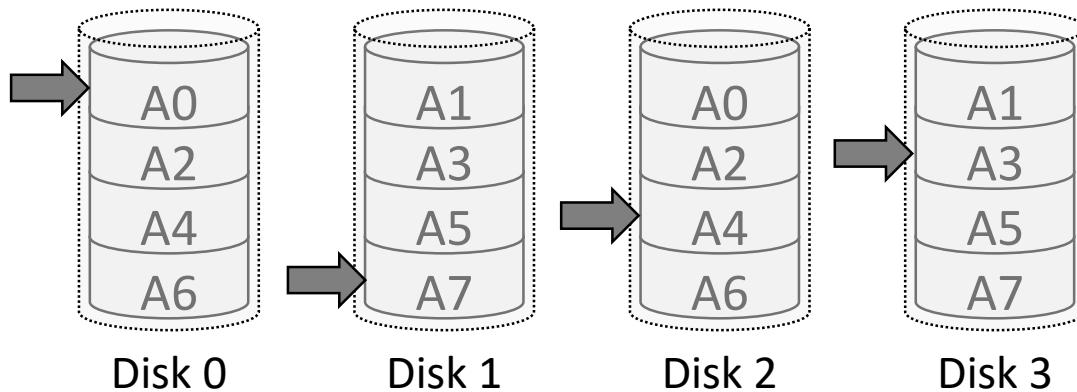


Capacity	
Reliability	1 drive can fail
Sequential read and write	$N * S$
Random read and write	$N/2 * R$

N/2 drives update data block while the remaining N/2 drives copy them as well (∵ half throughput)

N: number of disks, S: sequential access BW, R: random access BW

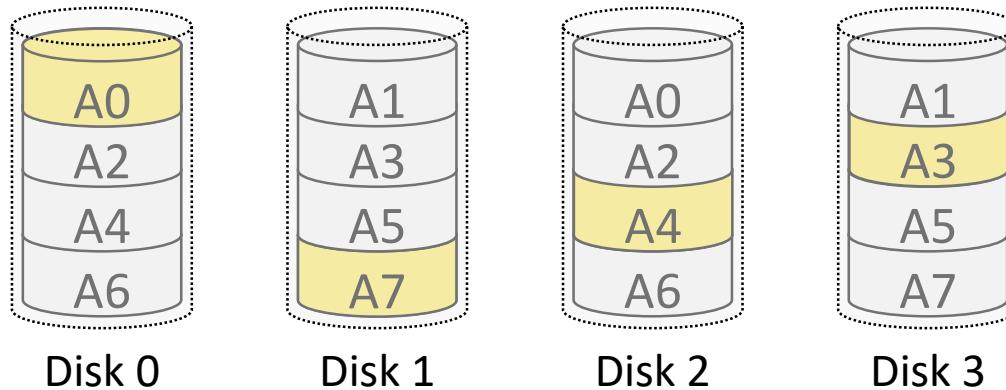
Analysis of RAID 0+1



Capacity	What is the maximum read bandwidth of RAID 0+1 for random reads ?	
Reliability		Same more
Sequential read and write		$N/2 * S$
Random read and write		$N/2 * R$

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1



Capacity		
Reliability	1 drive can fail	
Sequential read and write	$N * S$	
Random read and write	$N * R$	$N/2 * R$

Random reads can be parallelized across all the underlying disks (it's the best case scenario for RAID 0+1)

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1

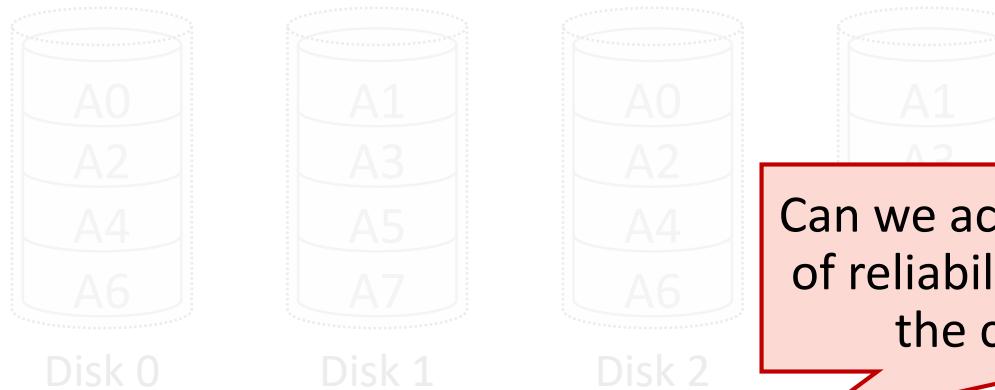


Mirroring is fault tolerant, but it can
lose $N/2$ of the array capacity

Capacity	$N/2$	
Reliability	1 drive can fail, sometime more	
Sequential read and write	$N * S$	$N/2 * S$
Random read and write	$N * R$	$N/2 * R$

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 0+1



Can we achieve the same level of reliability while minimizing the capacity losses?

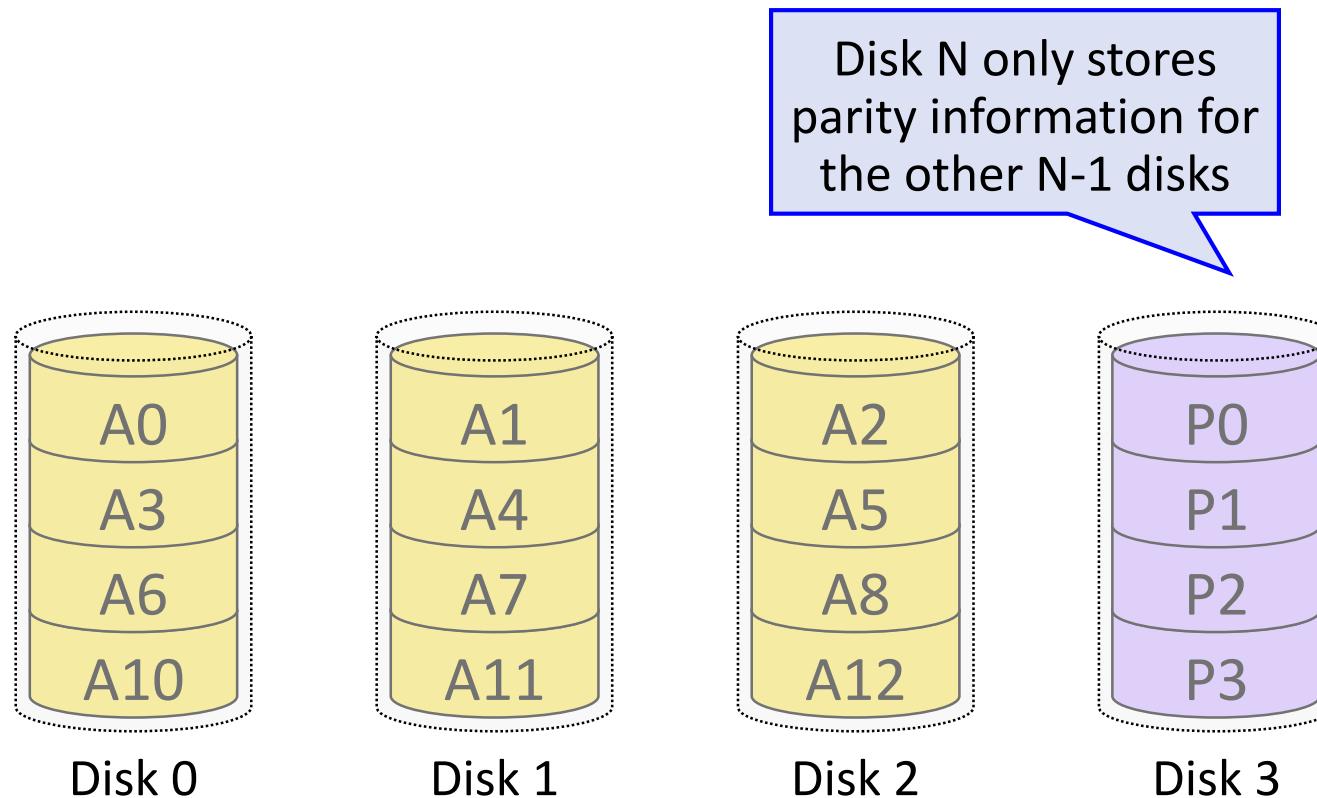


Capacity	N/2	
Reliability	1 drive can fail, sometime more	
Sequential read and write	N * S	N/2 * S
Random read and write	N * R	N/2 * R

N: number of disks, S: sequential access BW, R: random access BW

RAID 4 (Parity Drive)

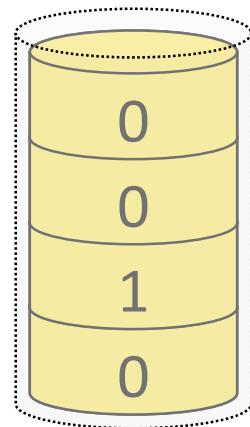
- **Key idea:** use information a coding technique as a light-weight error recovery mechanism



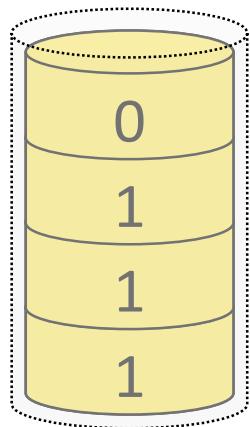


Parity Calculation

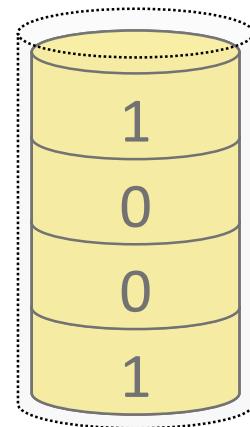
Let's check how the parity can be calculated with a simple example



Disk 0



Disk 1



Disk 2

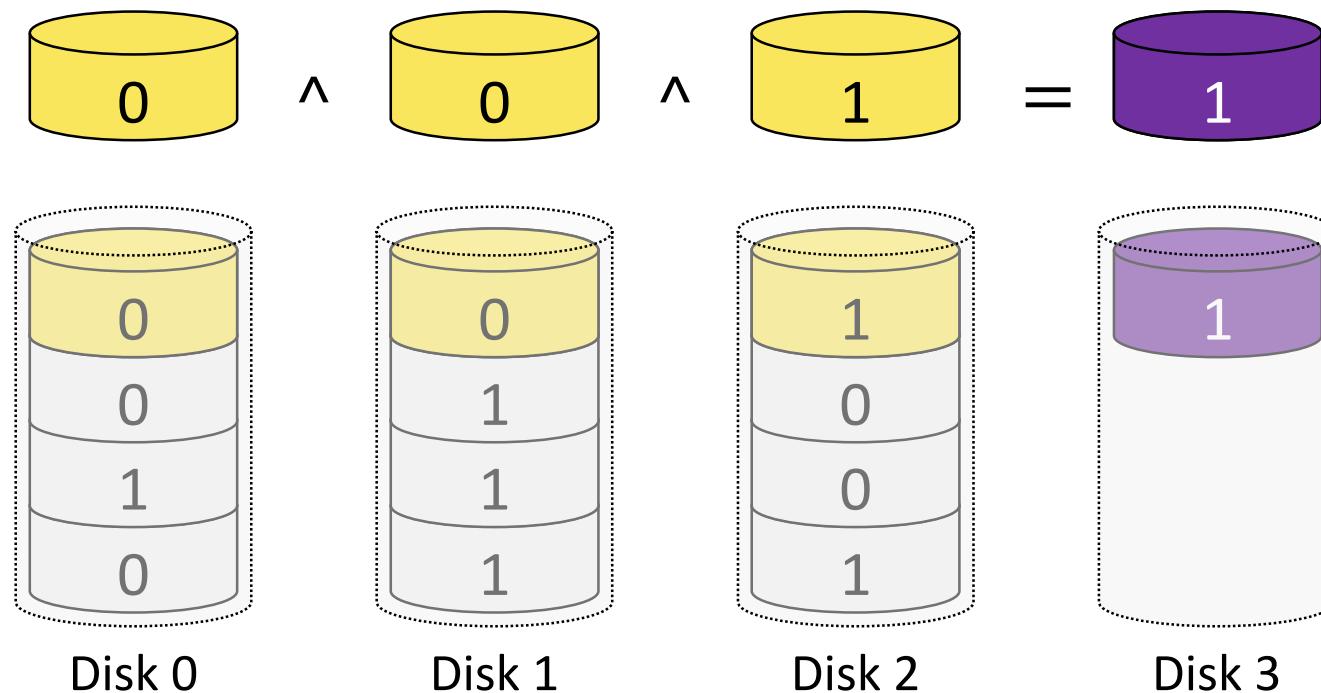


Disk 3



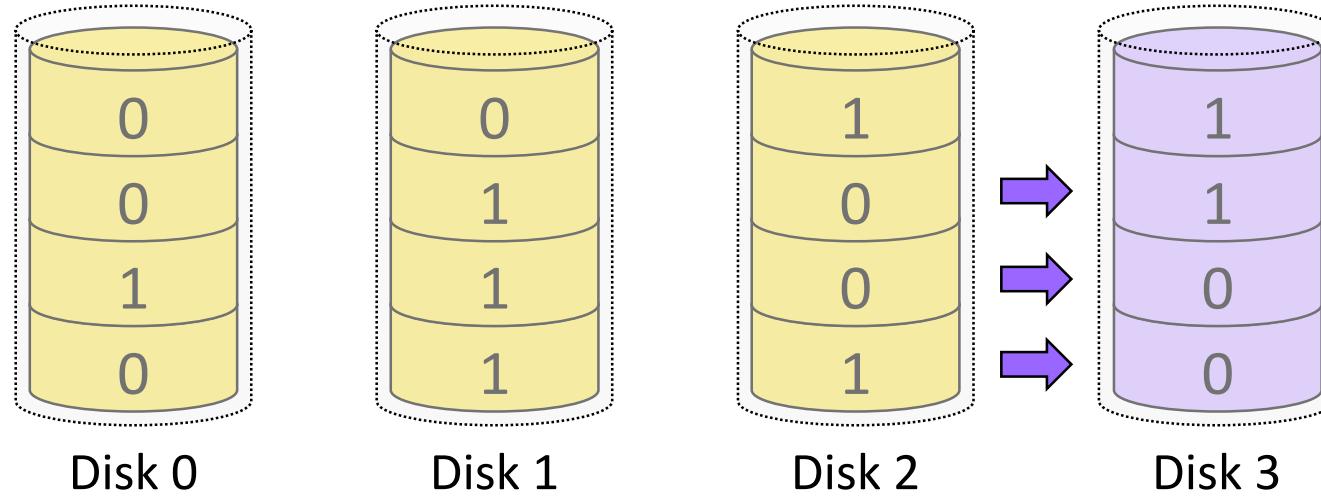
Parity Calculation

A **parity** block is calculated using XOR with the corresponding **data** blocks





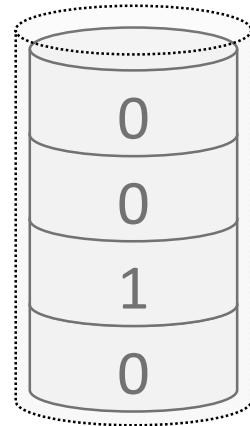
Parity Calculation



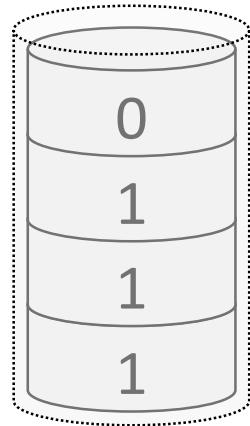


Updating Parity on Write

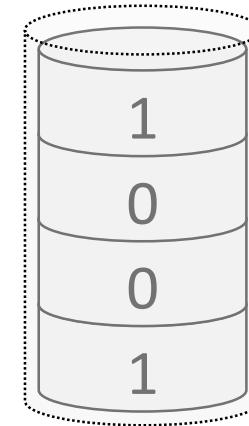
BTW, how can we update
the parity if there is a new
block update?



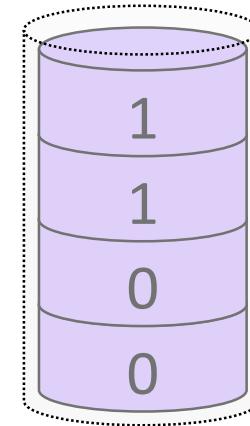
Disk 0



Disk 1



Disk 2

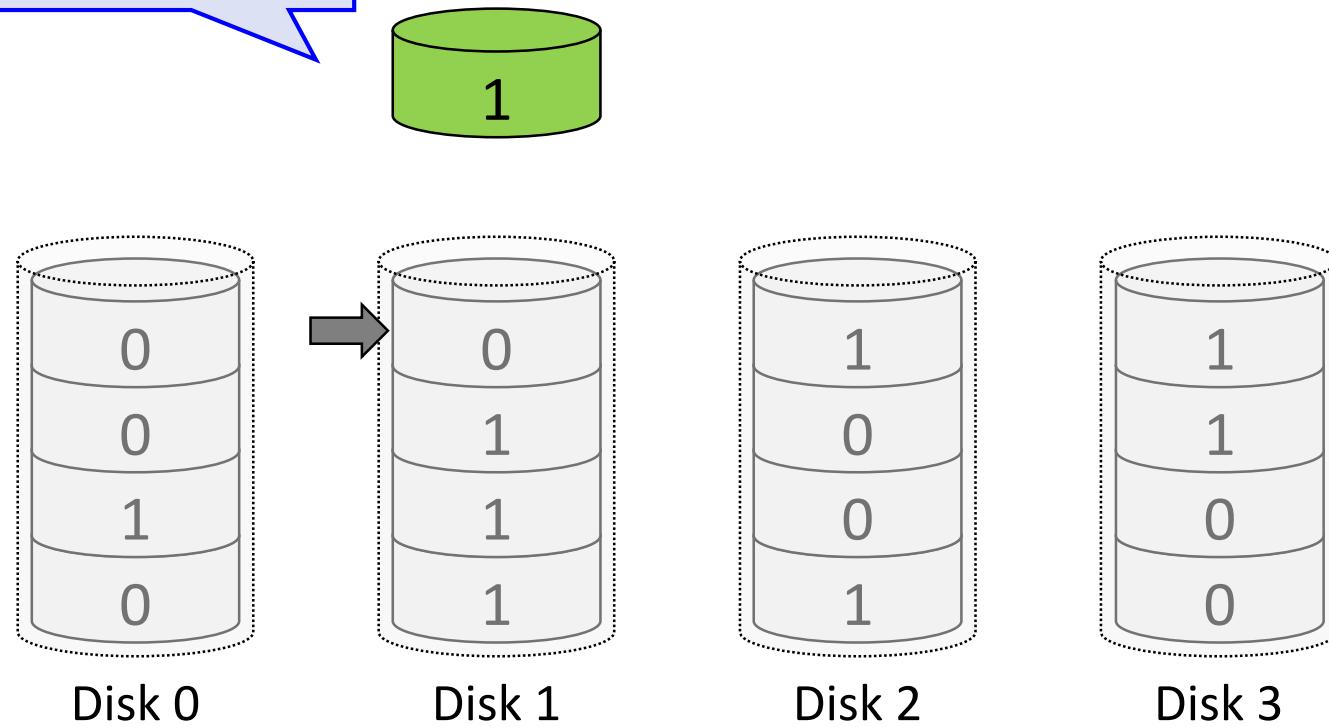


Disk 3



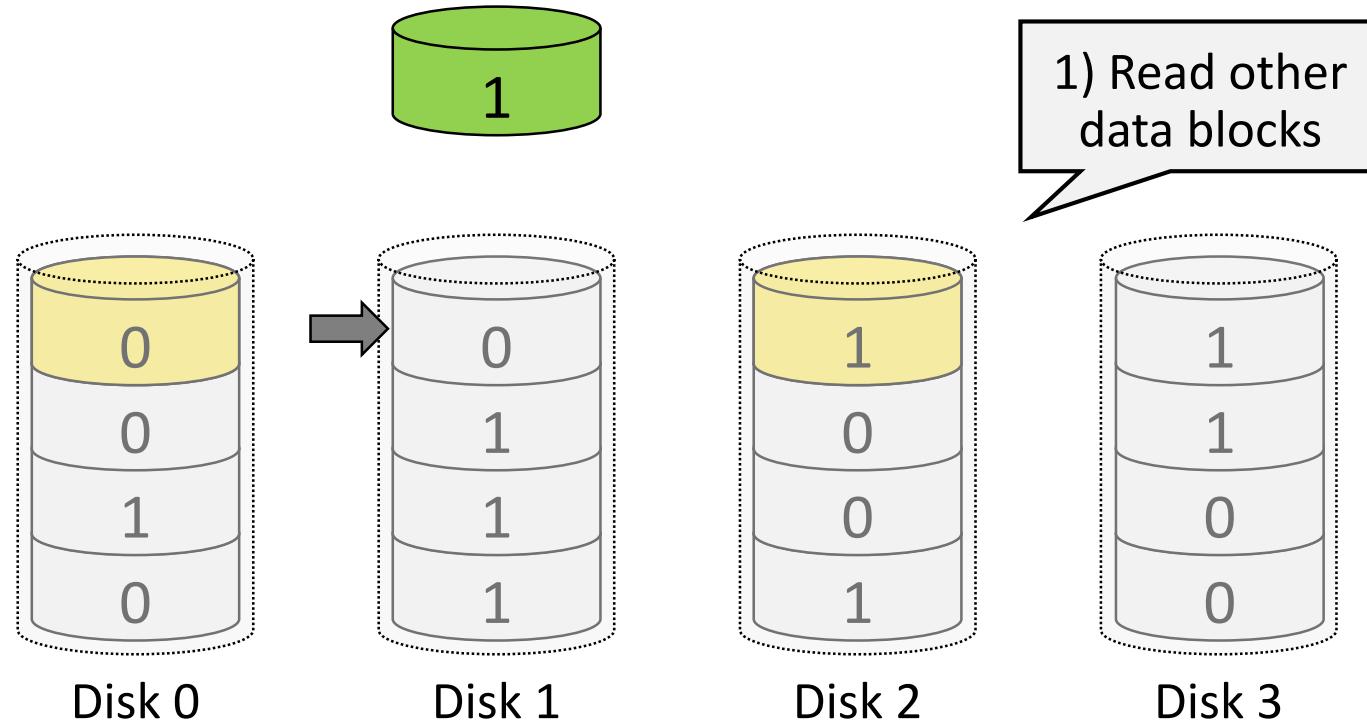
Updating Parity on Write

Case #1:
Additive parity





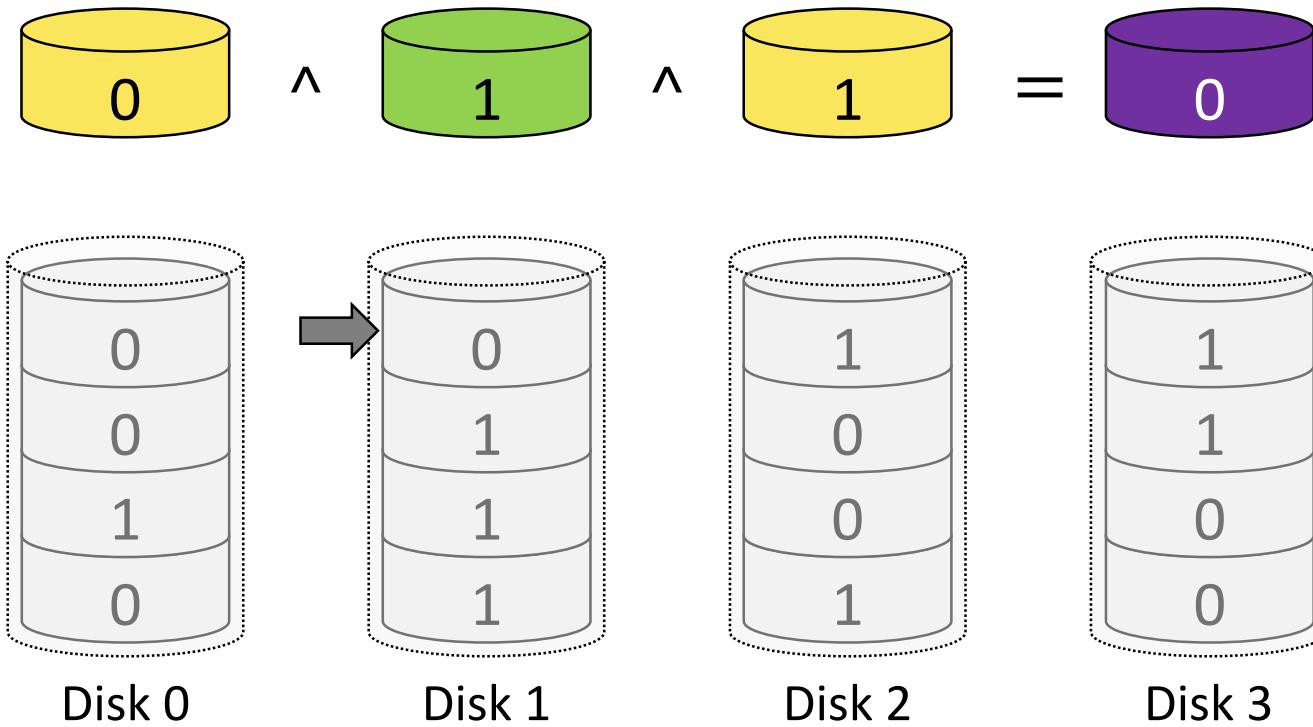
Updating Parity on Write





Updating Parity on Write

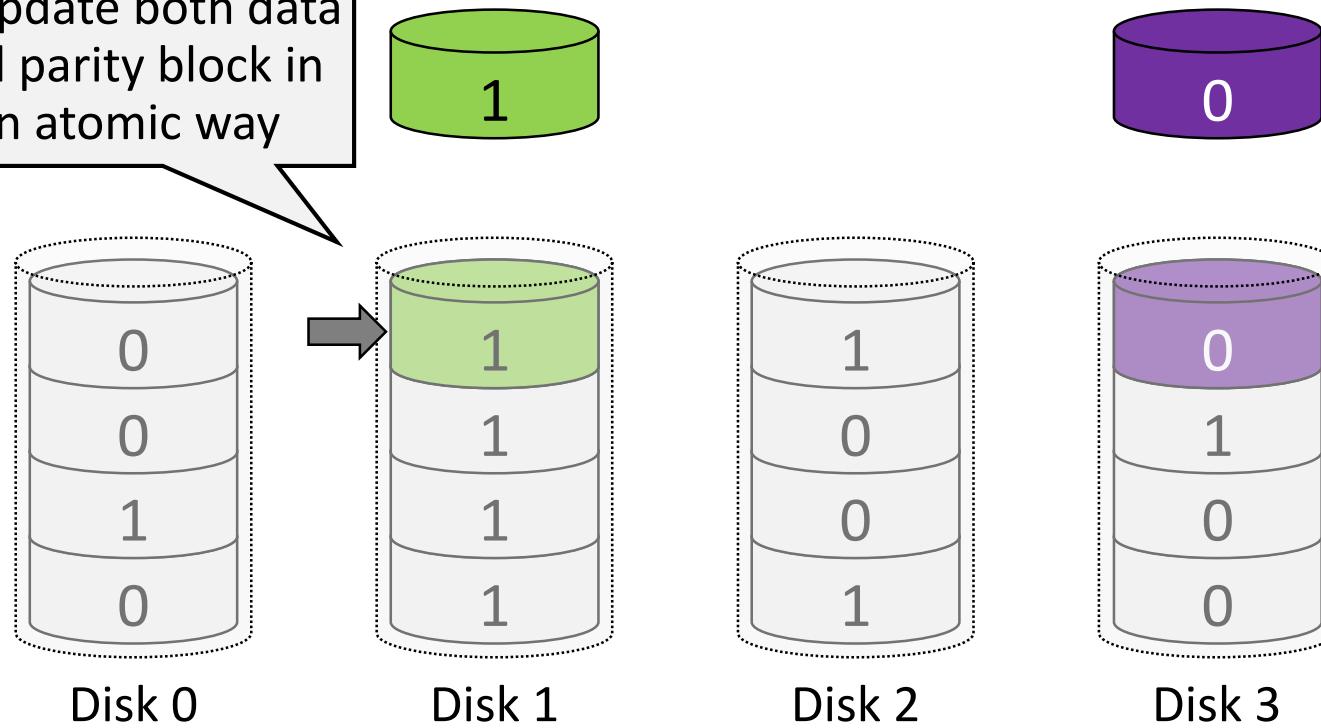
2) Recalculate
the parity block

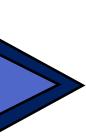




Updating Parity on Write

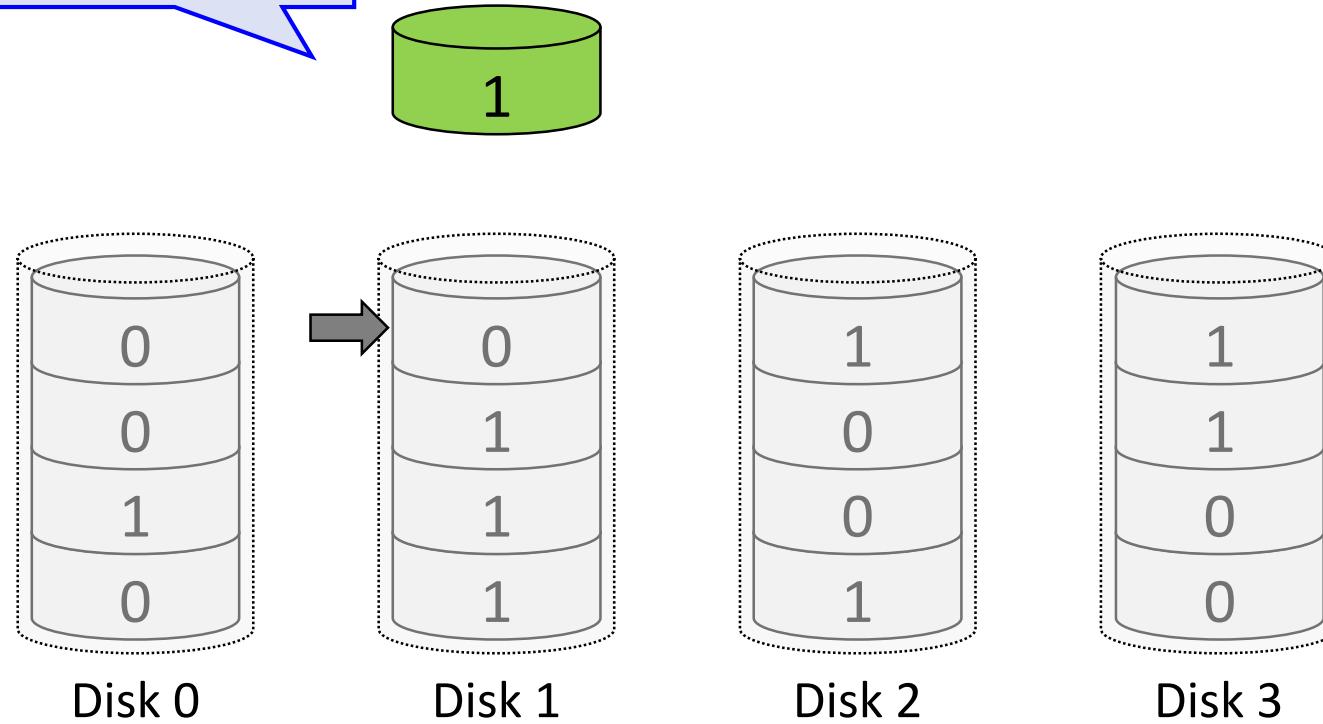
3) Update both data and parity block in an atomic way





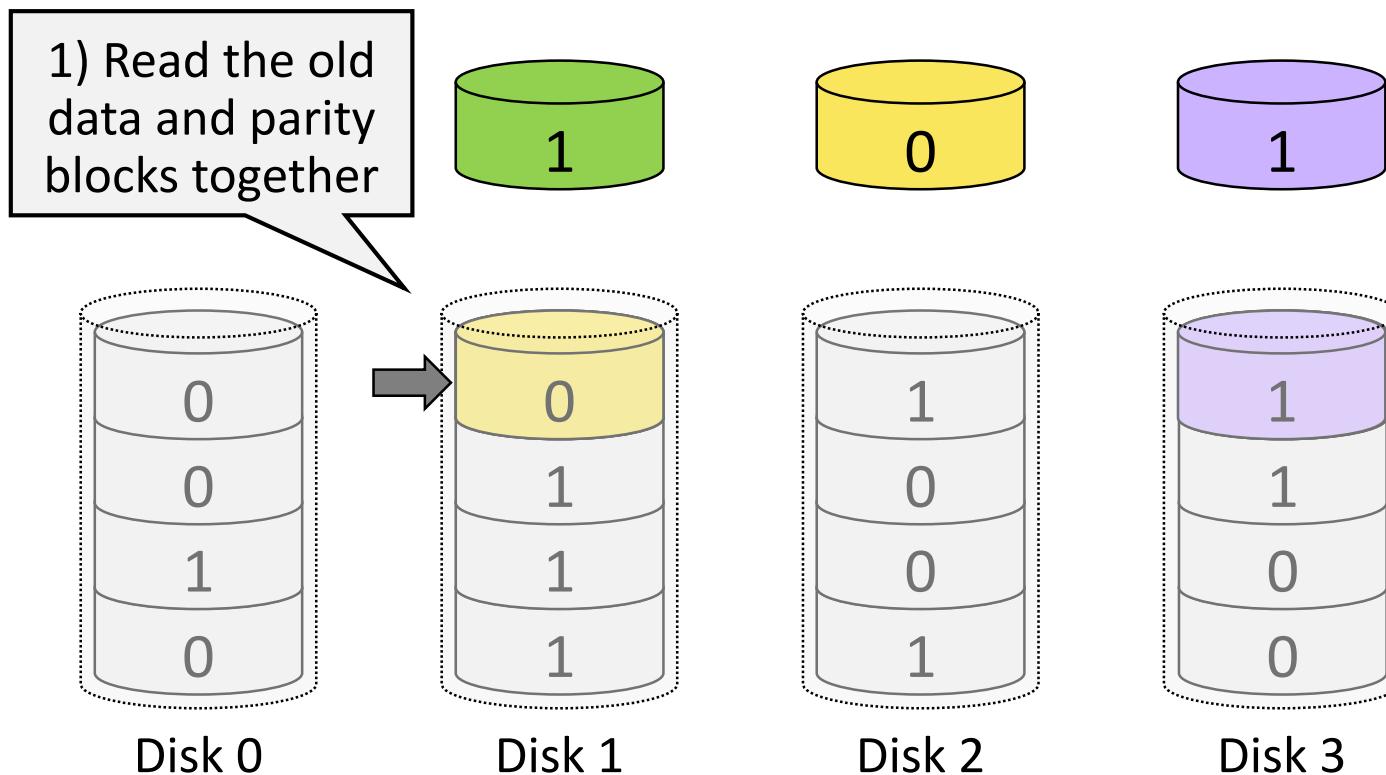
Updating Parity on Write

Case #2:
Subtractive parity





Updating Parity on Write

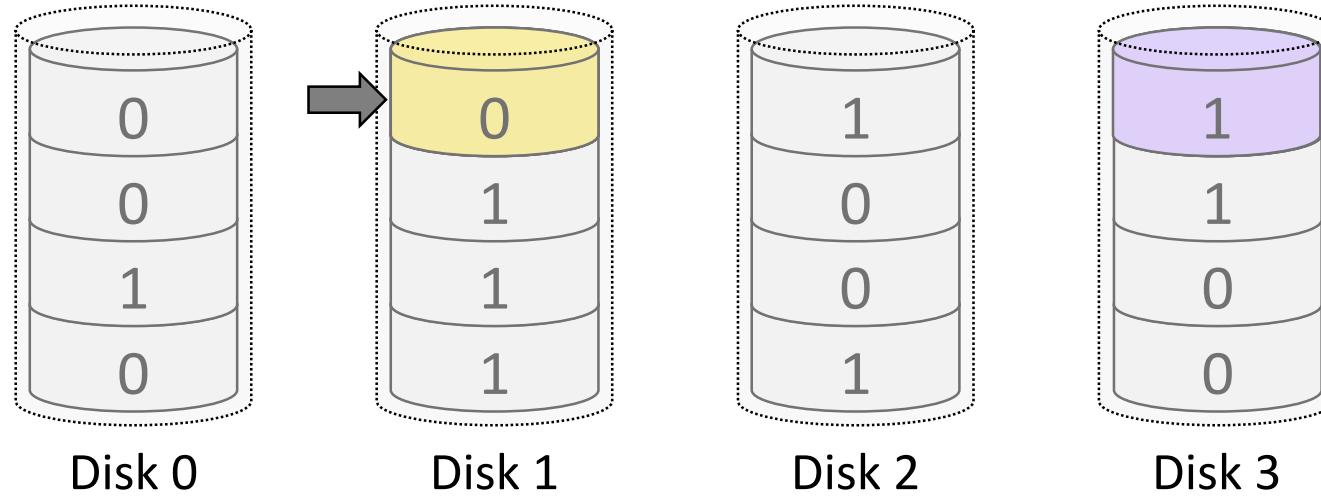


Updating Parity on Write

2) Adjust a parity difference of the updated data block to the parity block

$$P_{new} = C_{new} \wedge C_{old} \wedge P_{old}$$

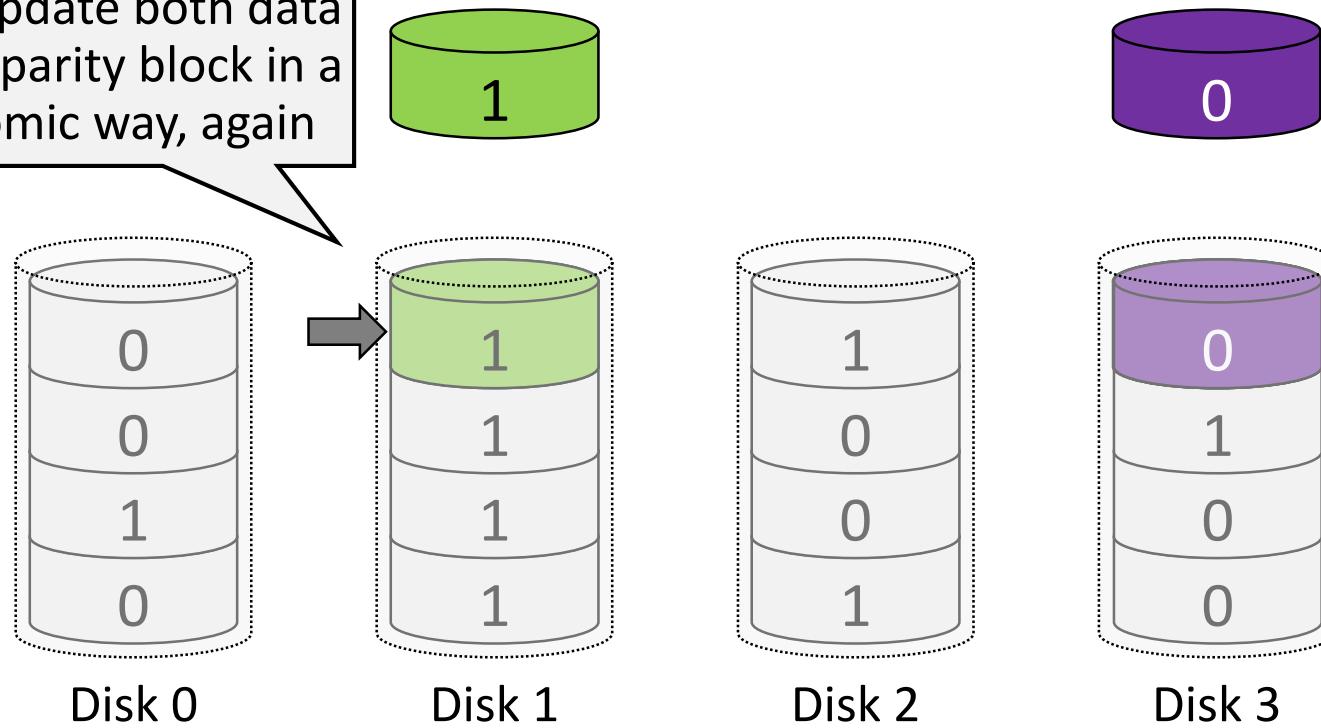
$$\begin{matrix} 1 \\ \wedge \\ 0 \\ \wedge \\ 1 \\ = \\ 0 \end{matrix}$$



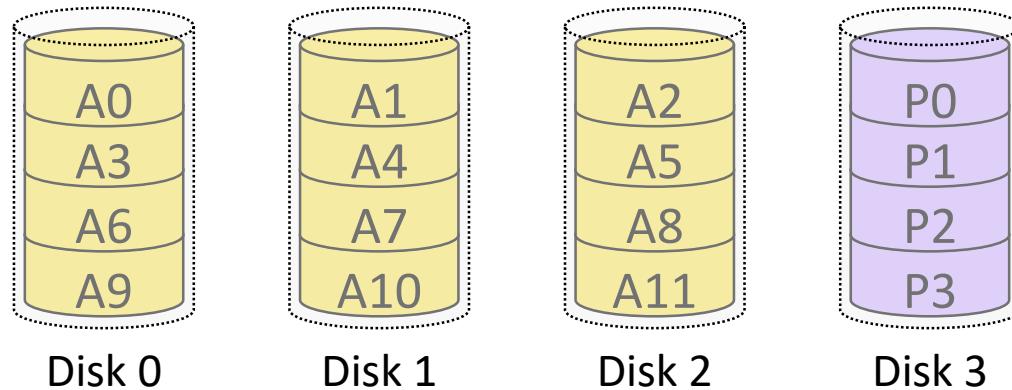


Updating Parity on Write

3) Update both data and parity block in a atomic way, again



Analysis of RAID 4

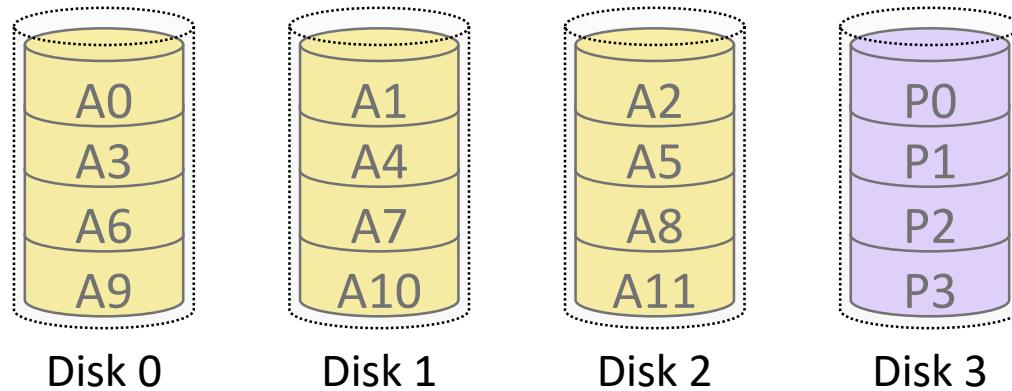


The space on the parity drive is not available for the data

Capacity	$N-1$
Reliability	
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

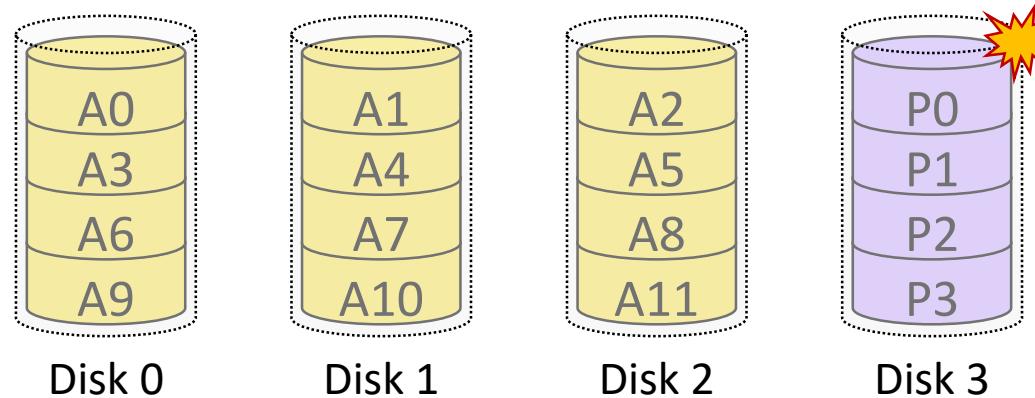
Analysis of RAID 4



Capacity	N-1
Reliability	1 drive can fail
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

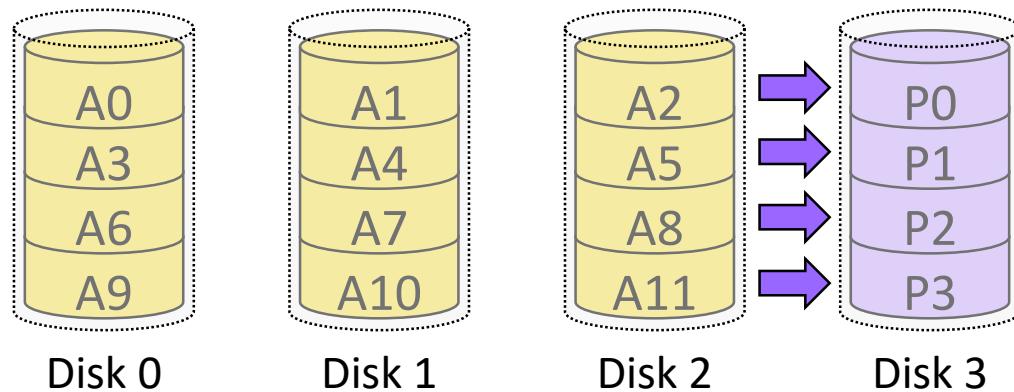
Analysis of RAID 4



Capacity	N-1
Reliability	1 drive can fail
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4



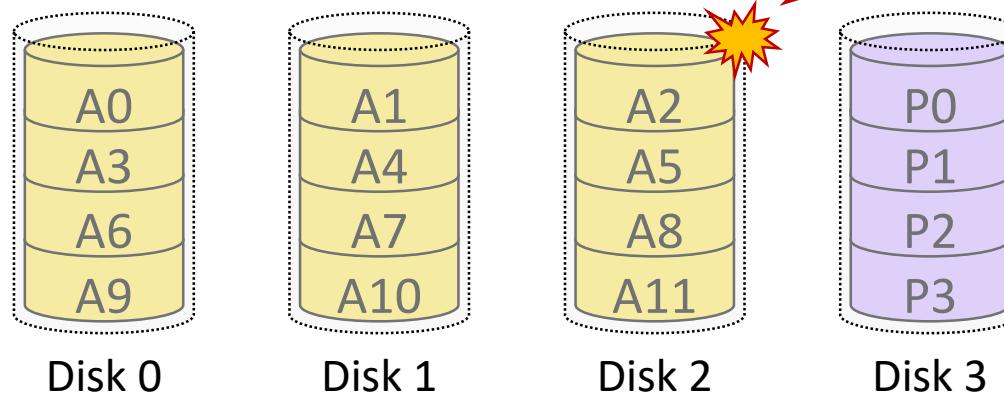
Recalculate the parity blocks with the existing data blocks

Capacity	N-1
Reliability	1 drive can fail
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4

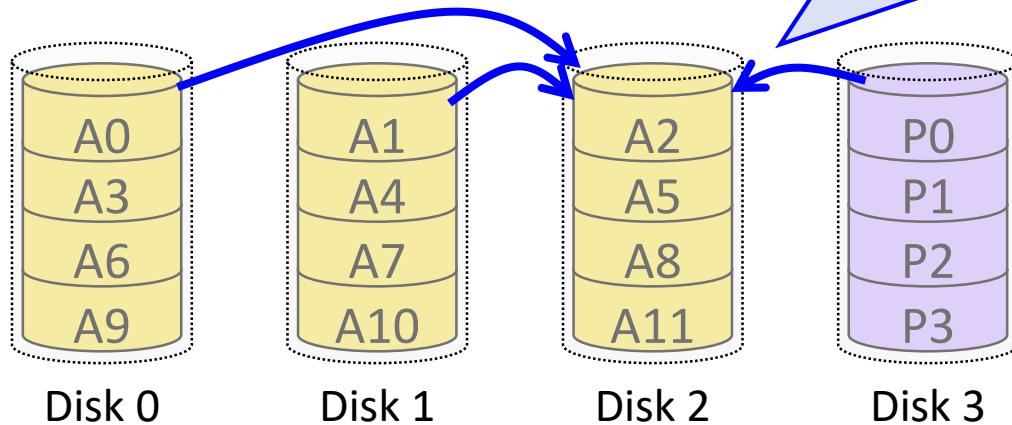
What if a data drive fails?



Capacity	N-1
Reliability	1 drive can fail
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

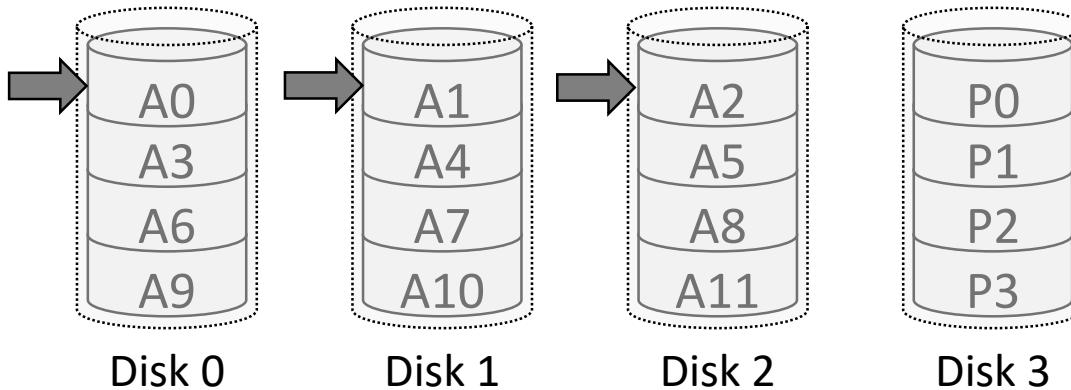
Analysis of RAID 4



Capacity	N-1
Reliability	1 drive can fail
Sequential read and write	
Random read and write	

N: number of disks, S: sequential access BW, R: random access BW

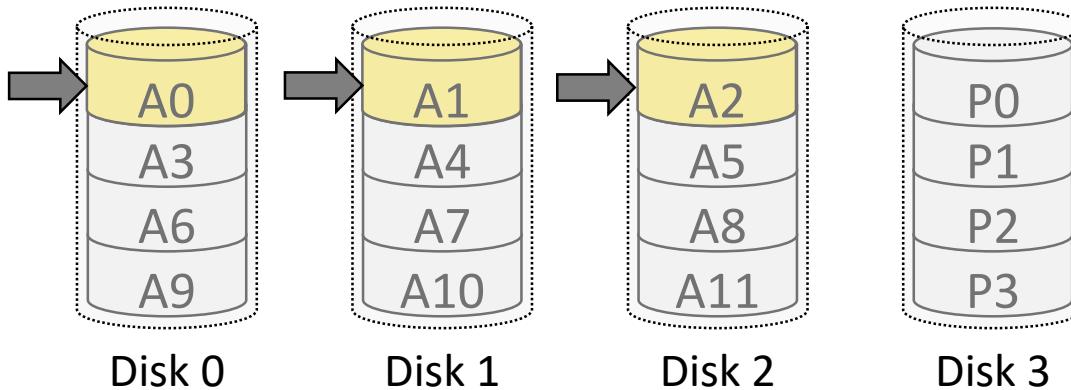
Analysis of RAID 4



Capacity	What is the maximum bandwidth of RAID 4 for sequential accesses ?	
Reliability	High	
Sequential read and write		
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

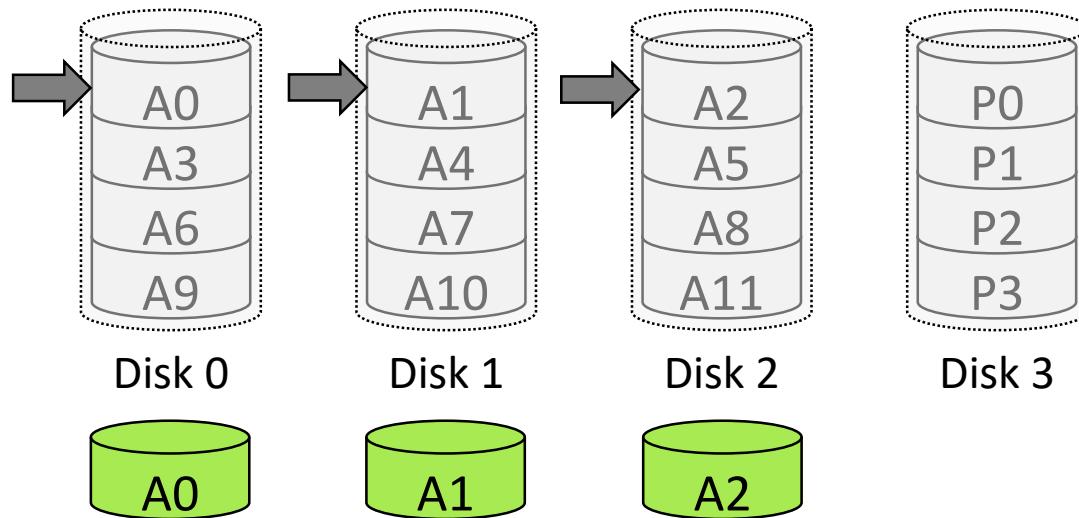
Analysis of RAID 4



Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4

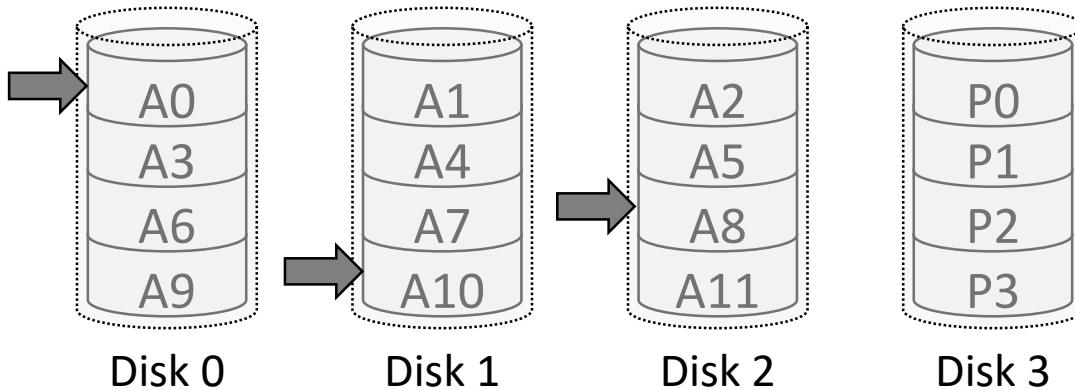


Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write		

Requests can be parallelized across all non-parity drives

N: number of disks, S: sequential access BW, R: random access BW

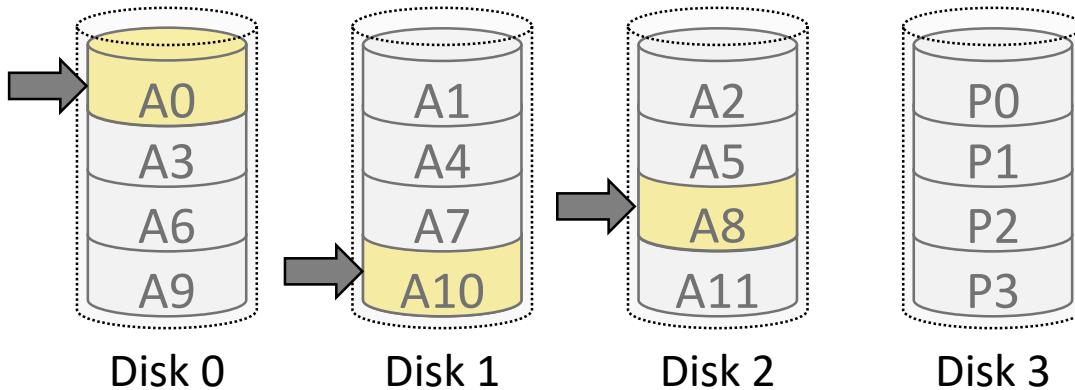
Analysis of RAID 4



Capacity	What is the maximum bandwidth of RAID 4 for random accesses ?	
Reliability		
Sequential read and write	$\frac{S}{N}$	$(N-1) * S$
Random read and write		

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4

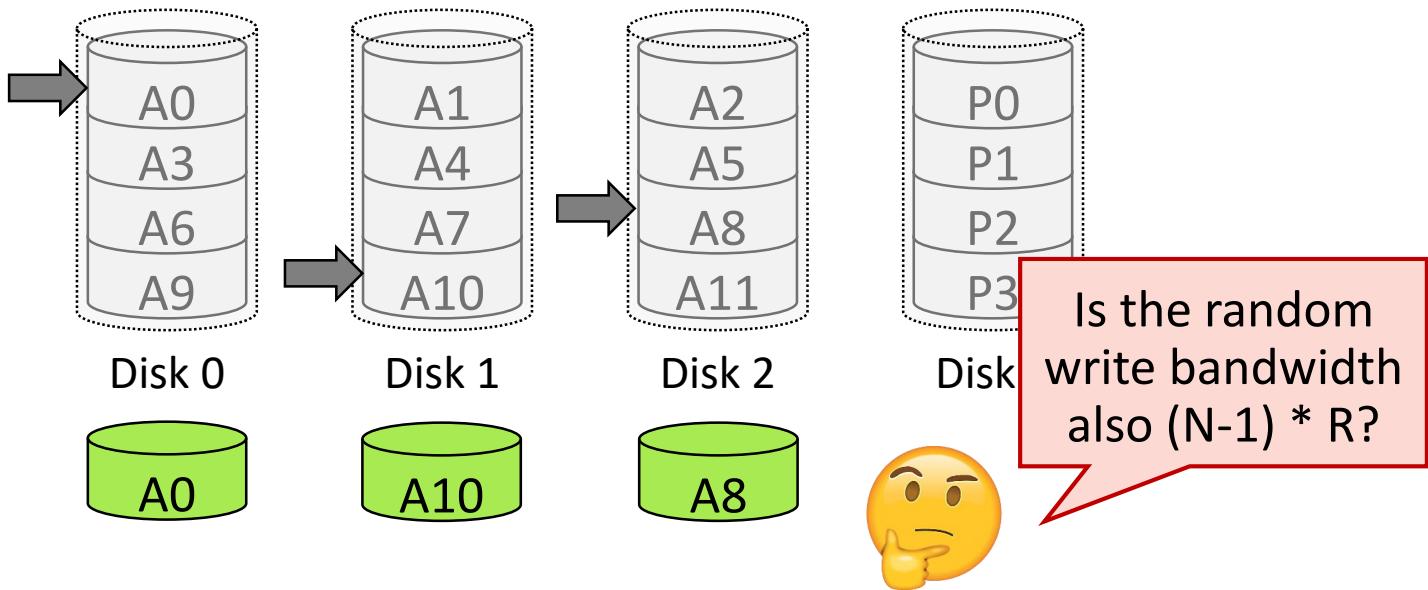


Capacity	N-1
Reliability	1 drive down
Sequential read and write	$(N-1) * S$
Random read and write	$(N-1) * R$

N: number of disks, S: sequential access BW, R: random access BW

Random reads can
be parallelized over
all but the parity
drive (best scenario)

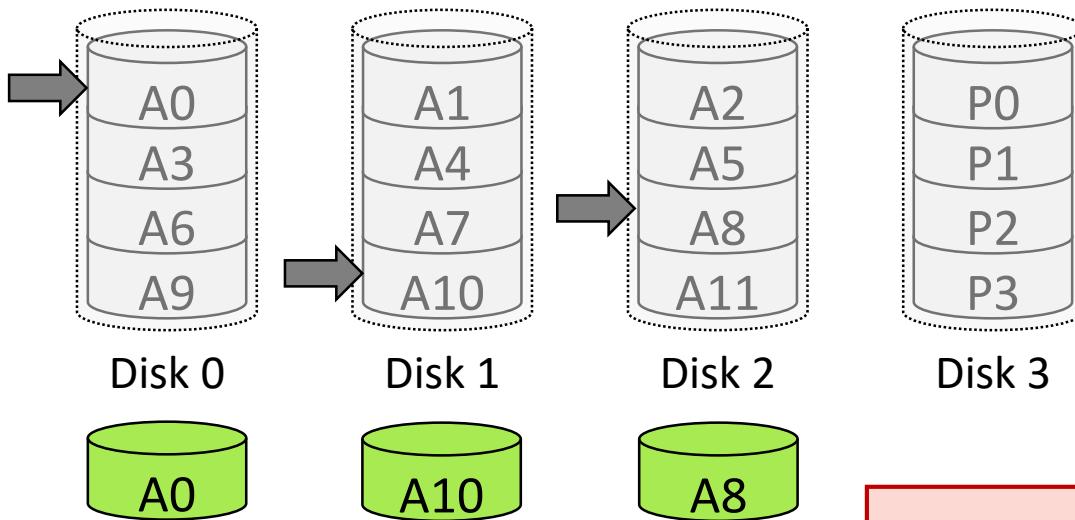
Analysis of RAID 4



Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$(N-1) * R$	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4



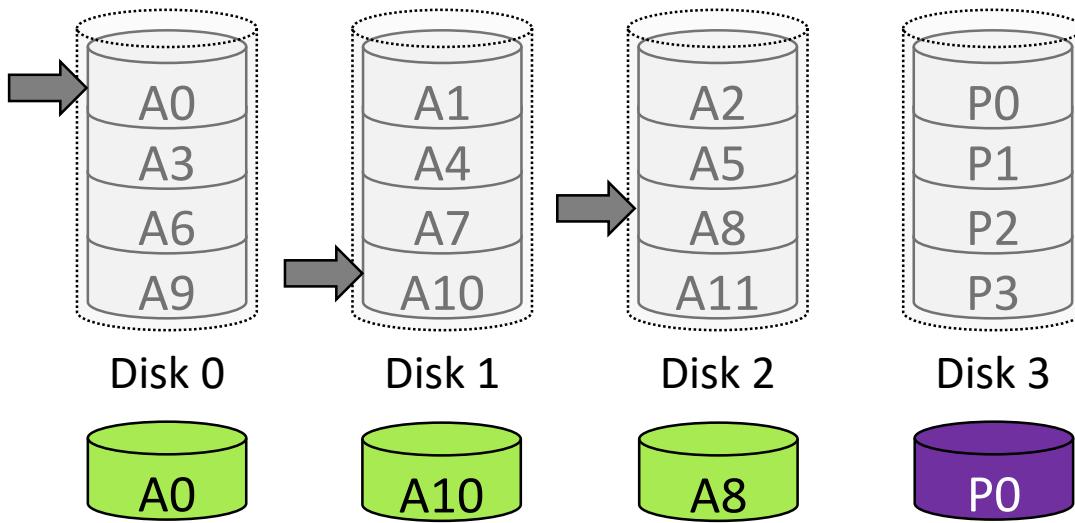
Capacity	(N-1) * S
Reliability	(R)
Sequential read and write	(N-1) * R
Random read and write	(N-1) * R



No! As all writes must update the parity drive, random writes causes serialization (Bottlenecked by the parity drive).

N: number of disks, S: sequential access BW, R: random access BW

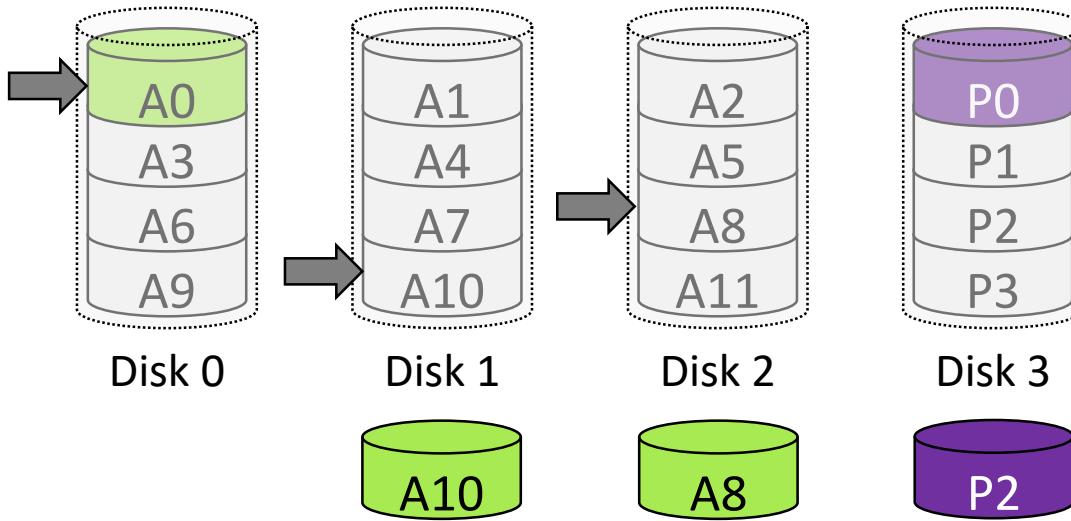
Analysis of RAID 4



Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$(N-1) * R$	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4

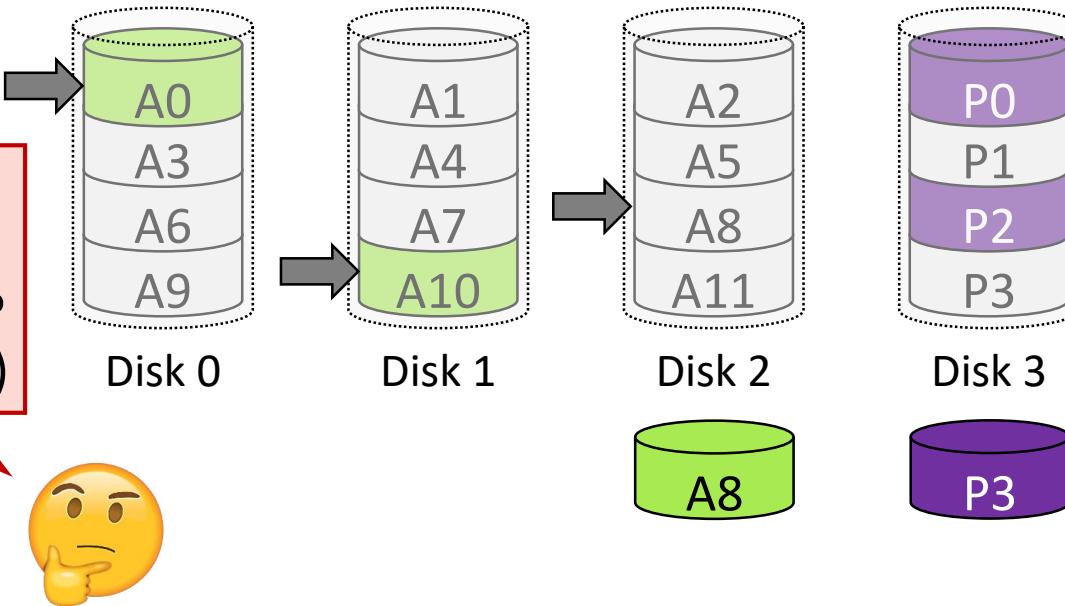


Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$(N-1) * R$	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4

Then, the random write bandwidth is **R**?
(Serialized perf.)



Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$(N-1) * R$	

N: number of disks, S: sequential access BW, R: random access BW

Analysis of RAID 4



Don't forget about how write works

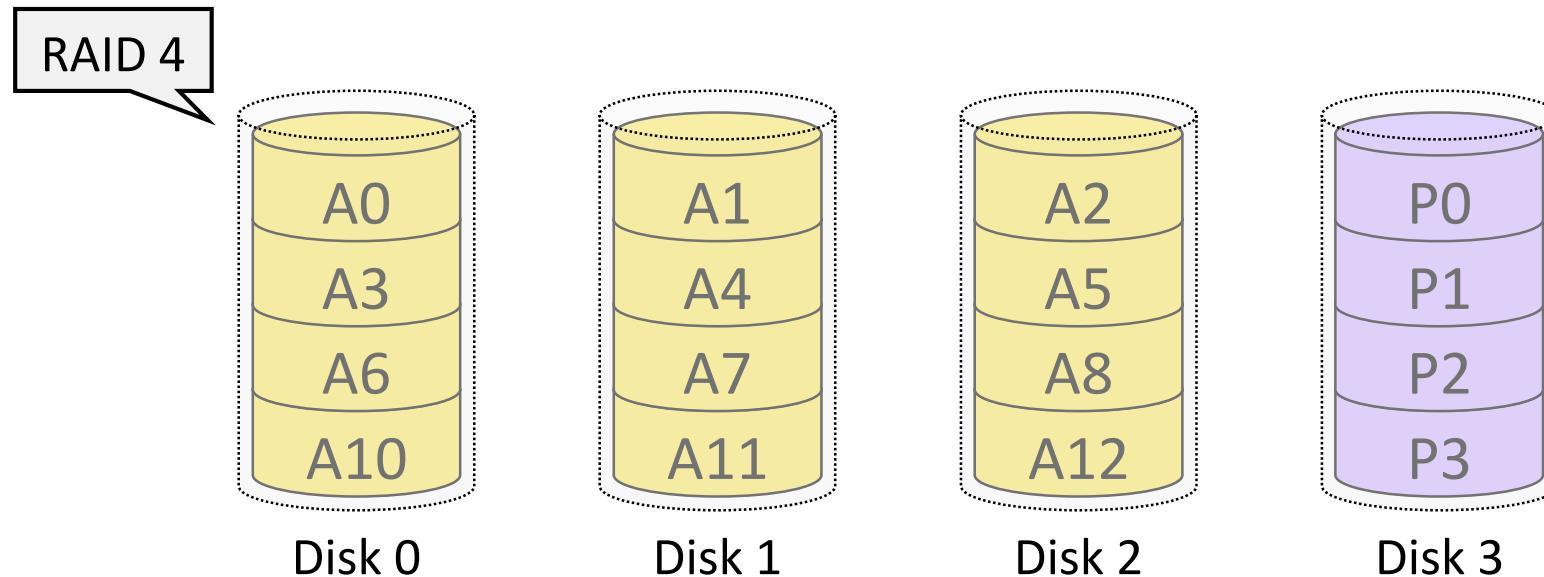
1. Read the target block and the parity block (**Read 1**)
2. Use subtraction to calculate the new parity block
3. Write the target block and the parity block (**Write 1**)

Capacity	N-1	
Reliability	1	Each write requests 1 read & 1 write, thus R/2
Sequential read and write	(N-1) * S	
Random read and write	(N-1) * R	R/2

N: number of disks, S: sequential access BW, R: random access BW

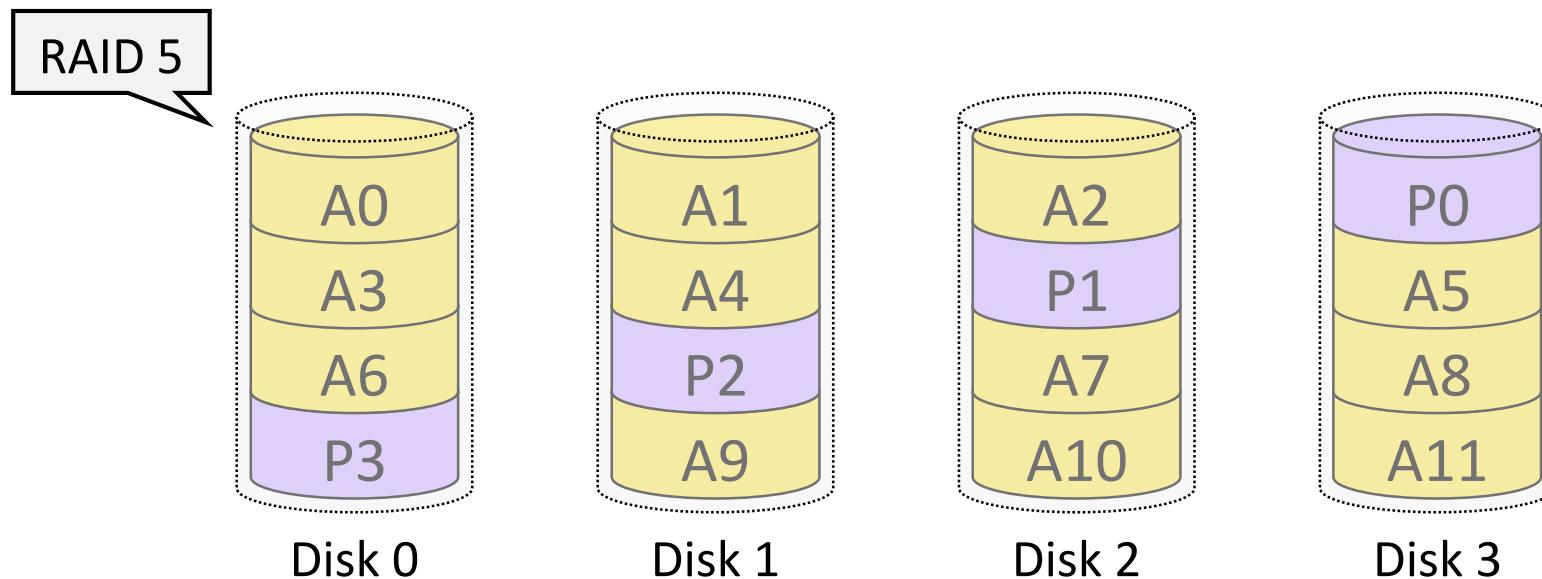
RAID 5 (Rotating Parity)

- **Key idea:** remove bottleneck of parity drive → spared parity blocks over all N disks

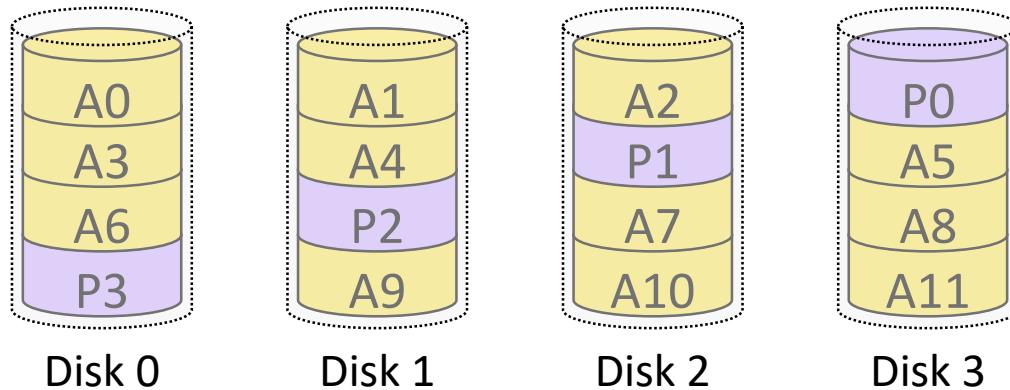


RAID 5 (Rotating Parity)

- **Key idea:** remove bottleneck of parity drive → spared parity blocks over all N disks



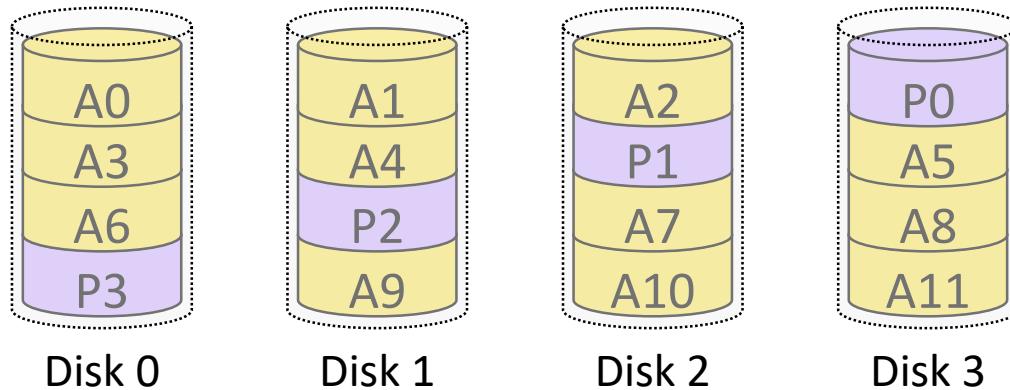
Analysis of RAID 5



Working behaviors of RAID 5 is the same with that of RAID 4

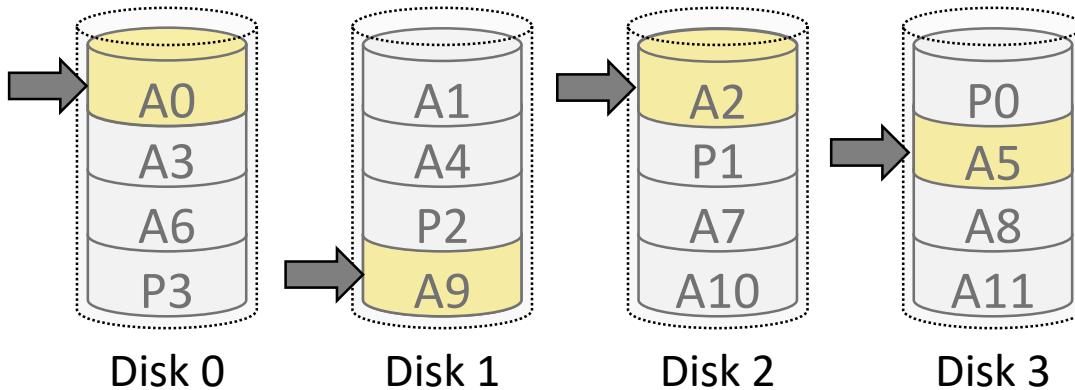
Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write		

Analysis of RAID 5



Capacity	But, RAID 5 exhibits better performance for both random reads & writes		
Reliability	fail		
Sequential read and write	$(N-1) * S$		
Random read and write			

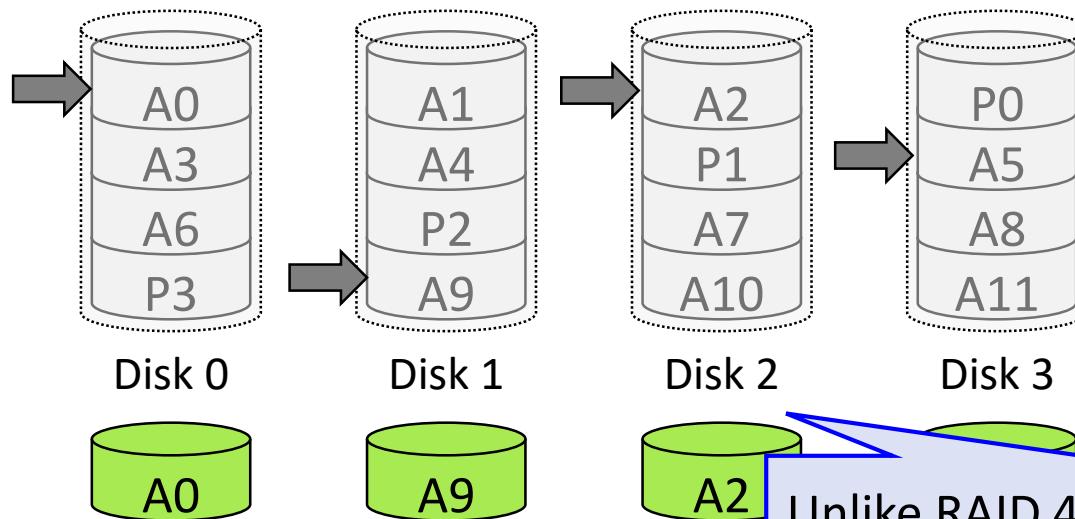
Analysis of RAID 5



Capacity	$N * S$
Reliability	1 drive failure tolerance
Sequential read and write	$(N-1) * S$
Random read and write	$N * R$

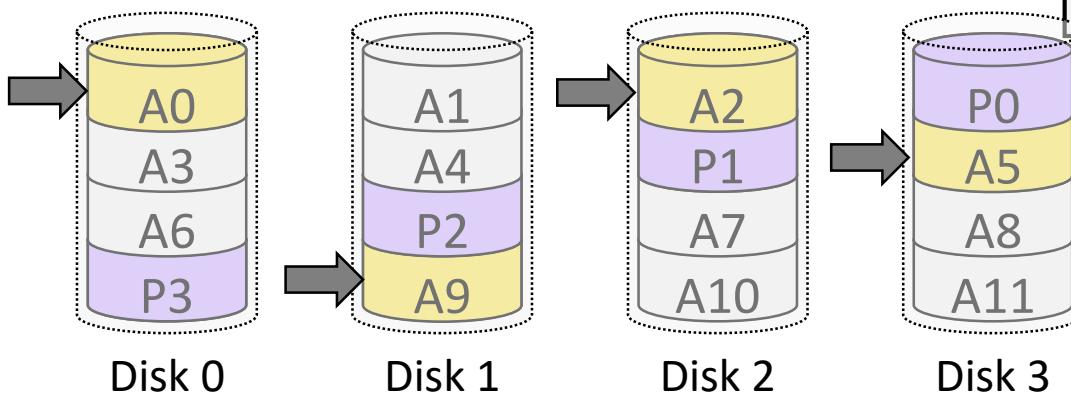
Unlike RAID 4, random reads can be parallelized over all the drives [vs. $(N-1) * R$]

Analysis of RAID 5

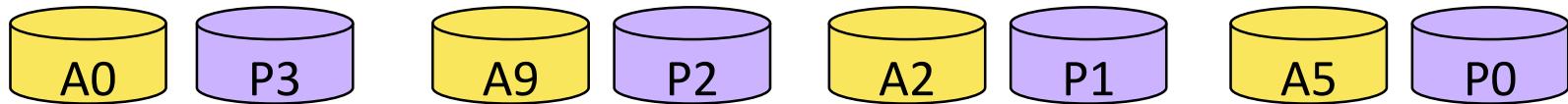


Capacity	$N-1$	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$N * R$	

Analysis of RAID 5



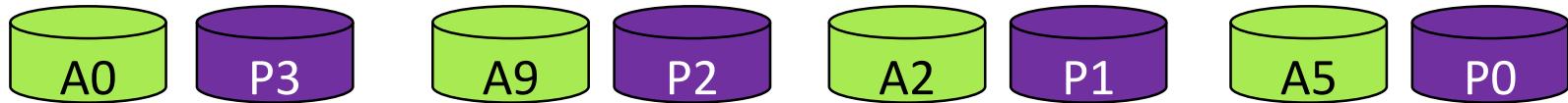
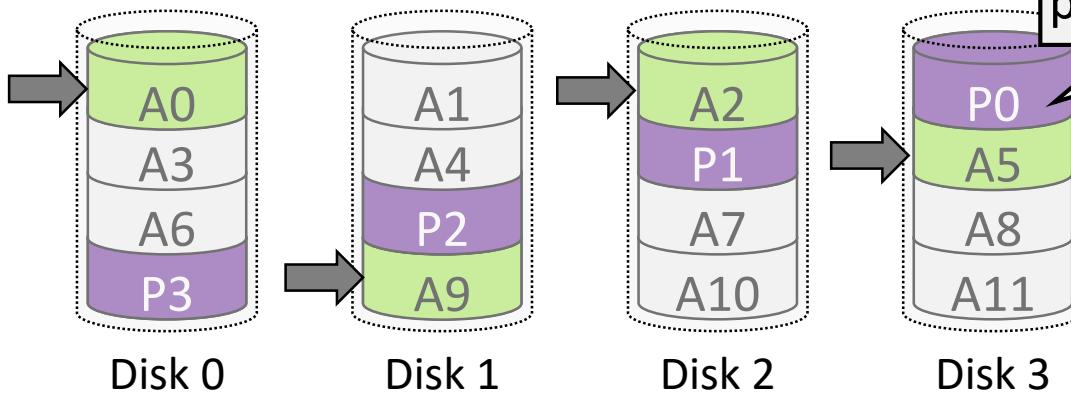
It requires reading the old data and parity blocks together (2 reads)



Capacity	$N-1$	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$N * R$	

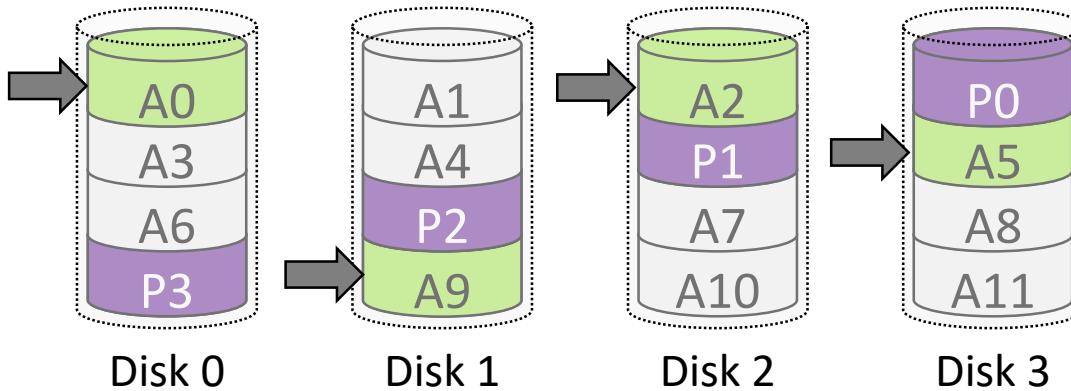
Analysis of RAID 5

It also need to update both new data block and new parity block (**2 writes**)



Capacity	N-1	
Reliability	1 drive can fail	
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$N * R$	

Analysis of RAID 5



Capacity		
Reliability		
Sequential read and write	$(N-1) * S$	$(N-1) * S$
Random read and write	$N * R$	$(N/4) * R$

A random write is composed by 4 I/O operations in total (2 reads + 2 writes), but it yet is better than RAID 4 [R/2]

[Summary] RAID Levels

		RAID 0	RAID 0+1	RAID 4	RAID 5
	Capacity	N	N / 2	N – 1	N – 1
	Reliability	0	1 (maybe N / 2)	1	1
Throughput	Sequential Read	N * S	N * S	(N – 1) * S	(N – 1) * S
	Sequential Write	N * S	(N / 2) * S	(N – 1) * S	(N – 1) * S
	Random Read	N * R	N * R	(N – 1) * R	N * R
	Random Write	N * R	(N / 2) * R	R / 2	(N / 4) * R
Latency	Read	D	D	D	D
	Write	D	D	2 * D	2 * D

N: number of disks

R: random access BW

S: sequential access BW

D: latency to access a single disk

[Summary] RAID Levels

		RAID 0	RAID 0+1	RAID 4	RAID 5
	Capacity	N	N / 2	N - 1	N - 1
	Reliability	0	1 (may lose 2)	1	1
Throughput	Sequential Read	N * S	N * S	(N - 1) * S	(N - 1) * S
	Sequential Write	N * S	(N / 2) * S	(N - 1) * S	(N - 1) * S
	Random Read	N * R	N * R	(N - 1) * R	(N - 1) * R
	Random Write	N * R	(N / 2) * R	R / 2	(N / 4) * R
Latency	Read	D	D	D	D
	Write	D	D	2 * D	2 * D

Why is the latency of RAID 5 not $4 * D$ (2 reads + 2 writes)?



N: number of disks

R: random access BW

S: sequential access BW

D: latency to access a single disk

[Summary] RAID Levels

		RAID 0	RAID 0+1	RAID 4	RAID 5
Capacity	N	N / 2		N - 1	N - 1
Reliability	2 * D	1 (maybe N / 2)			
Sequential Read	Data block read	Data block write			
Sequential Write	Parity block read	Parity block write			
Random Read	N * R	(N / 2) * R			
Random Write	N * R	(N / 2) * R	R / 2	(N / 4) * R	
Latency	Read	D	D	D	D
	Write	D	D	2 * D	2 * D

N: number of disks

R: random access BW

S: sequential access BW

D: latency to access a single disk

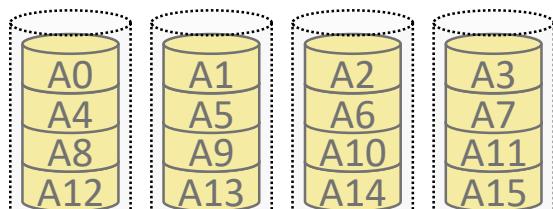
Managing the two blocks per I/O can be parallelized.



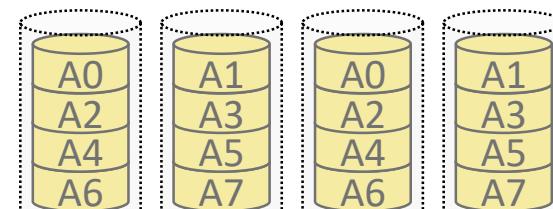
Choosing a RAID Level

Users can choose a RAID level based on their requirement

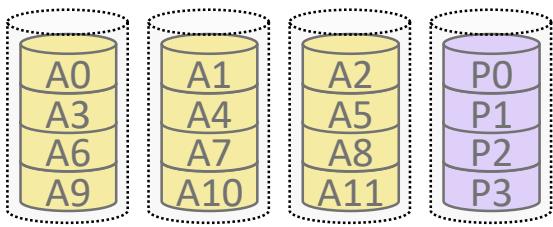
RAID 0



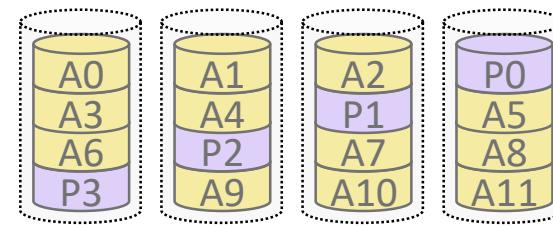
RAID 0+1



RAID 4

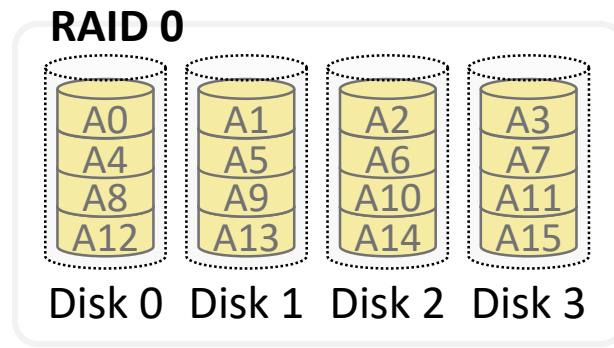


RAID 5

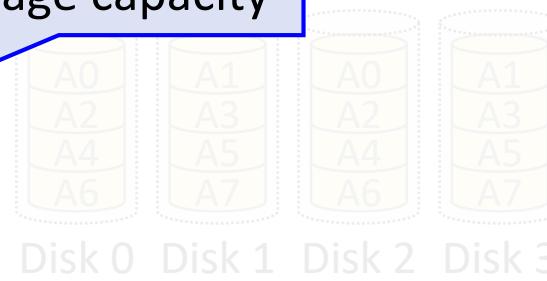




Need #1: Best Perf. & Most Capacity

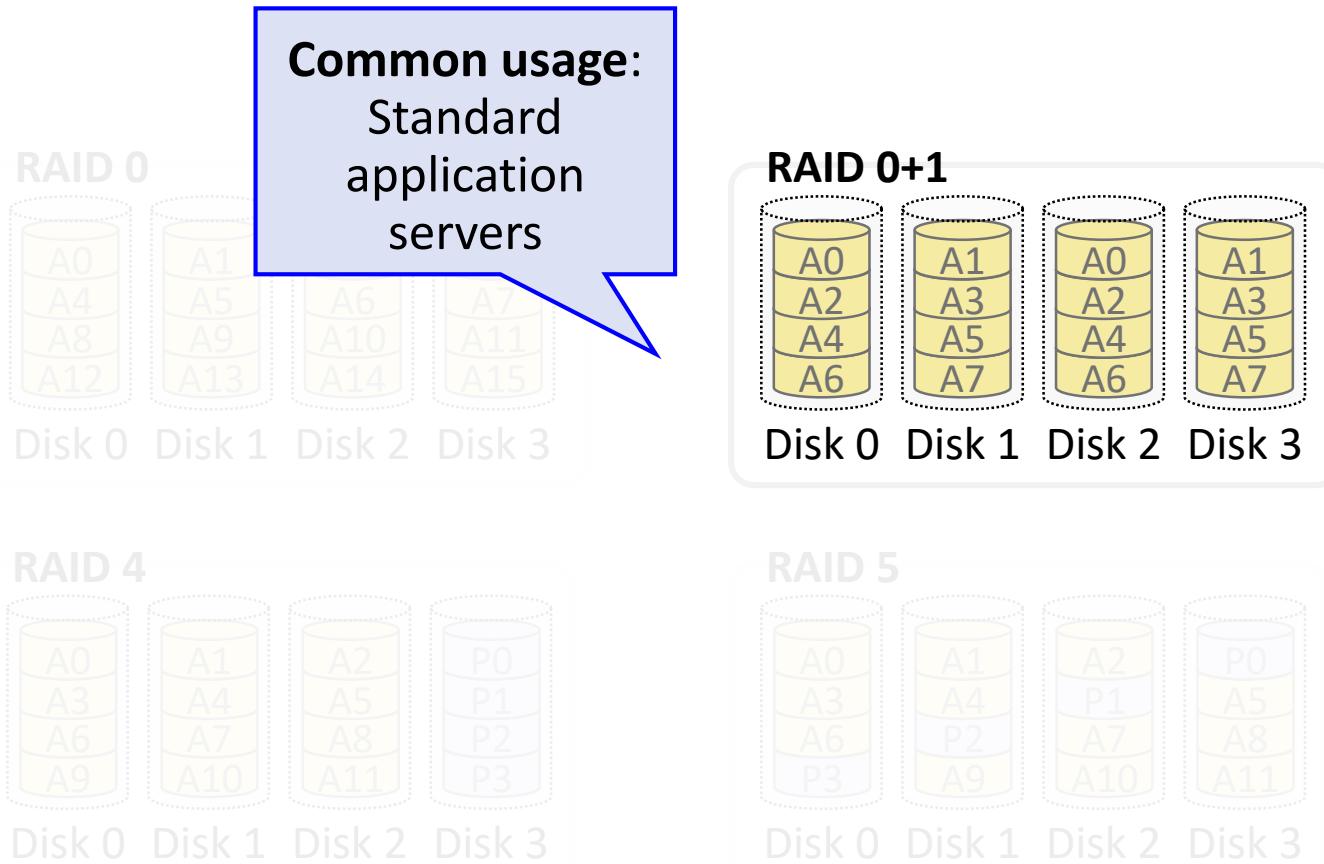


Common usage:
Systems require
maximizing the
storage capacity





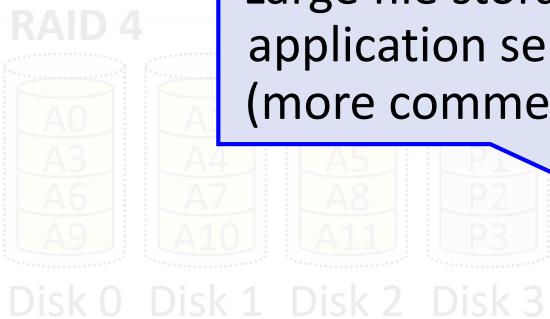
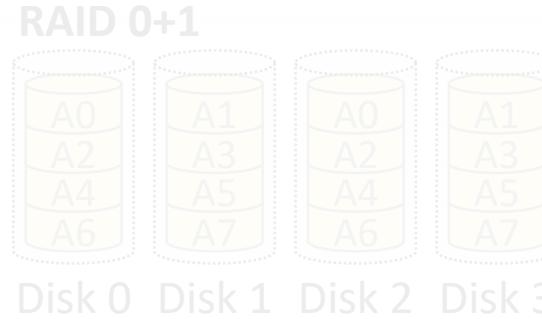
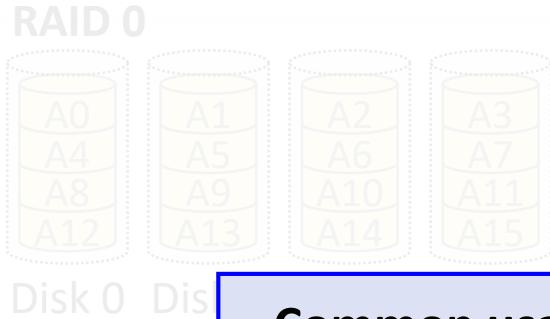
Need #2: Greatest Error Recovery



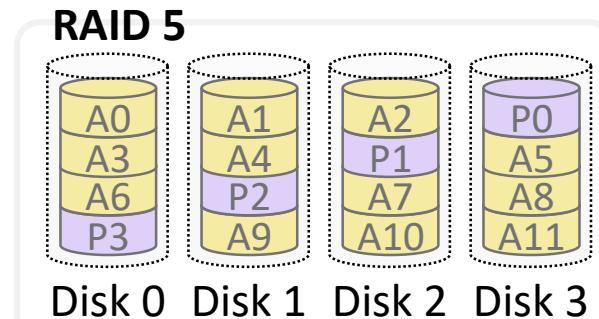


Need #3: Well Balanced

Space, performance, and recoverability



Common usage:
Large file storage &
application servers
(more commercial)

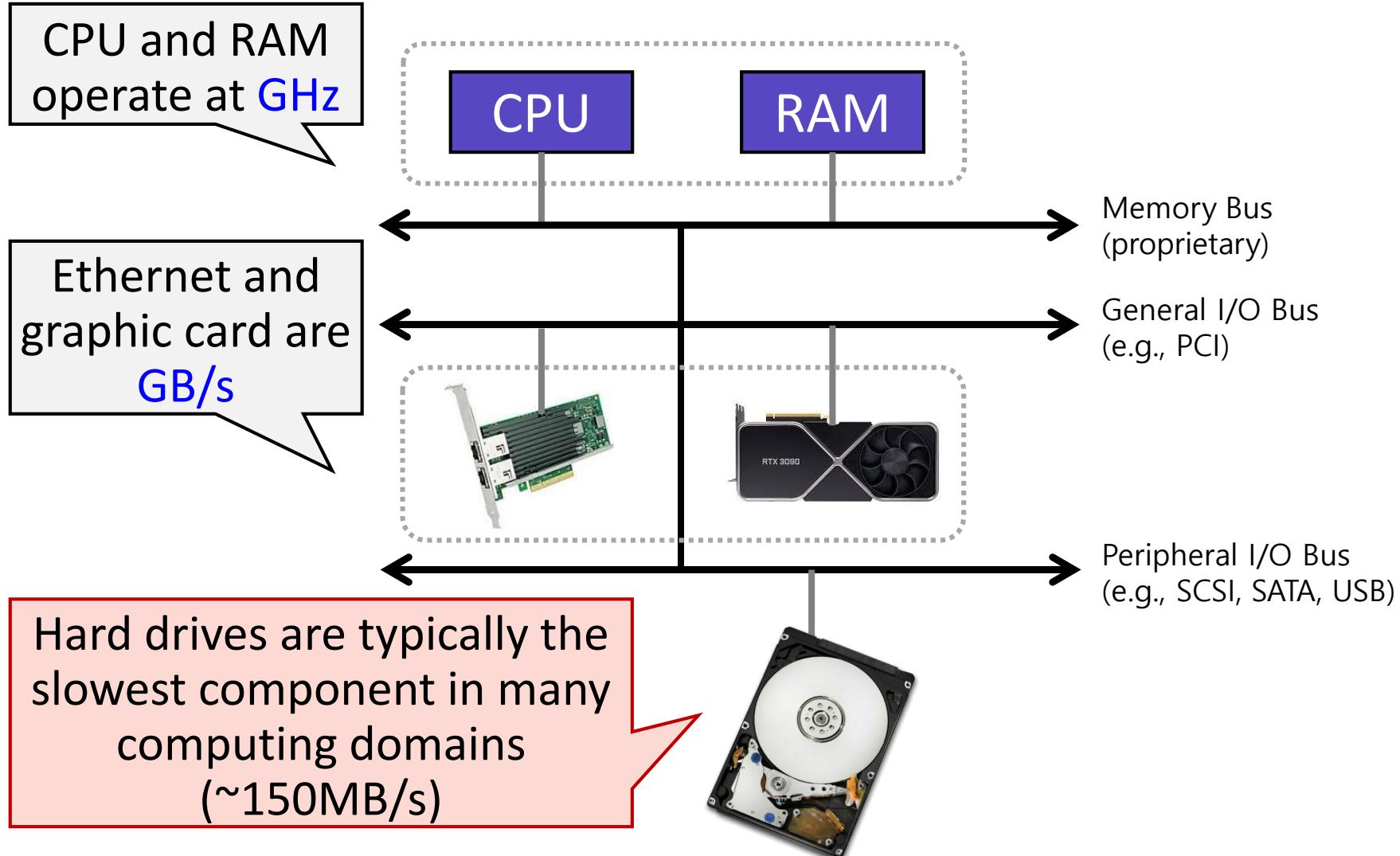


[Revisit] Shortcomings of Spinning Disk



BTW, can the RAID solve
all the shortcomings of
spinning disk?

#1: The Slowest Component



#2: Not Suitable for Mobile

What if you drop
your phone?

Fragile mechanical
components can break 😞



What about
power issue?



The disk motor is extremely
power hungry component 😞

Solution: **Beyond Spinning Disks**

We have another
option for
storage device ☺



Solid State Drives

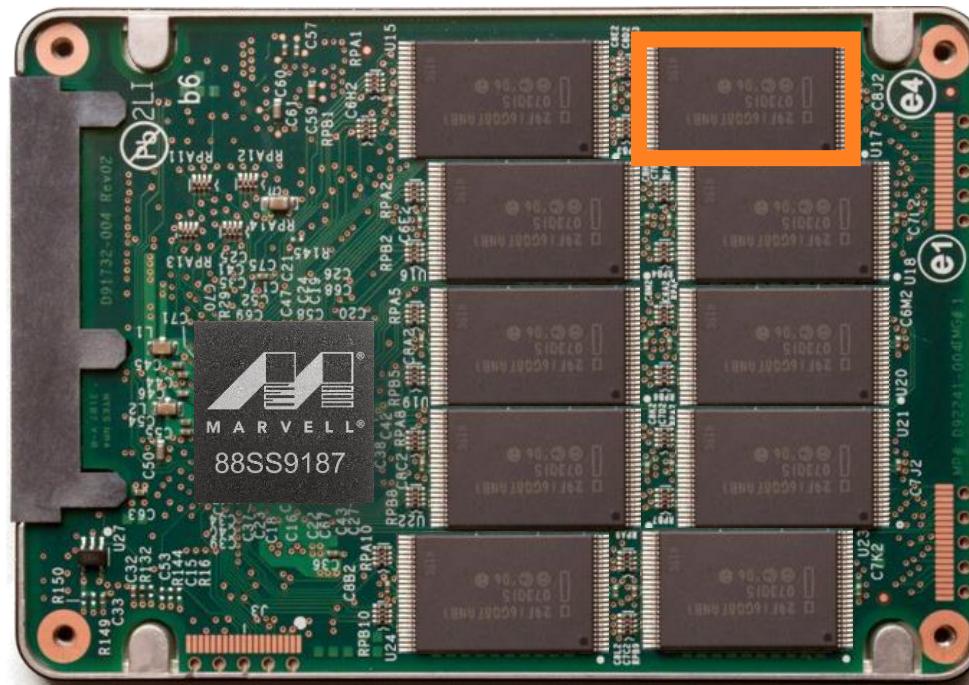
Solid State Drive (SSD) can address the shortcomings of hard drives (e.g., slow performance, high power consumption)



#1: No Mechanical, Moving Parts

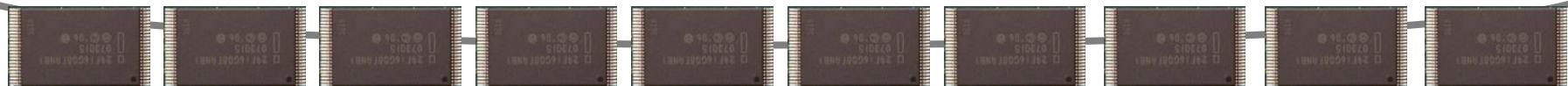
As there is no mechanically moving parts (read head or arm),
SSD is more **resilient** against physical damage and can **reduce power consumption**

NAND Flash memory chip (array of electrical switch)



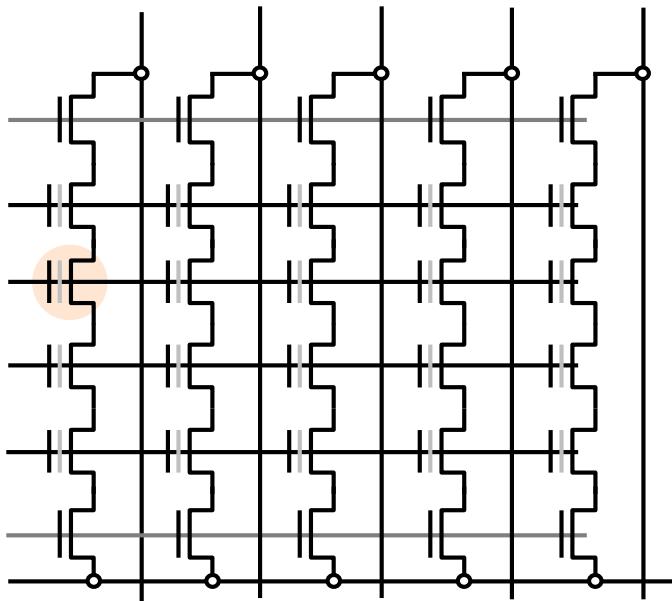
#2: Internal Parallelism

SSD is much **faster** than hard drives. Although each flash chip is slow, they are RAIDed (RAID 0)

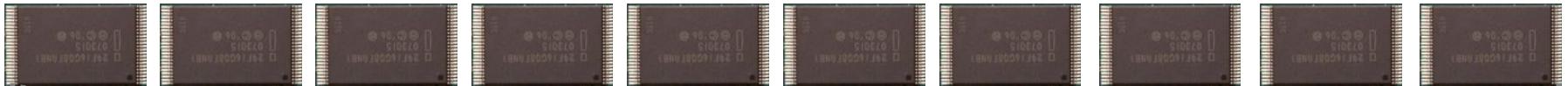


#3: No Penalty for Random Access

Each flash cell can store binary data (0 or 1)

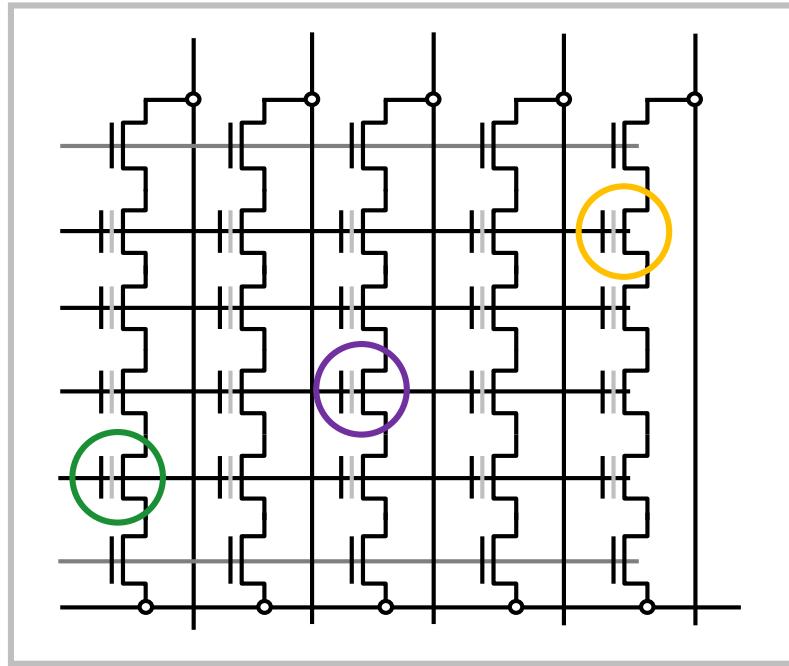


Array of electrical switches



#3: No Penalty for Random Access

How could we randomly access the flash cells?

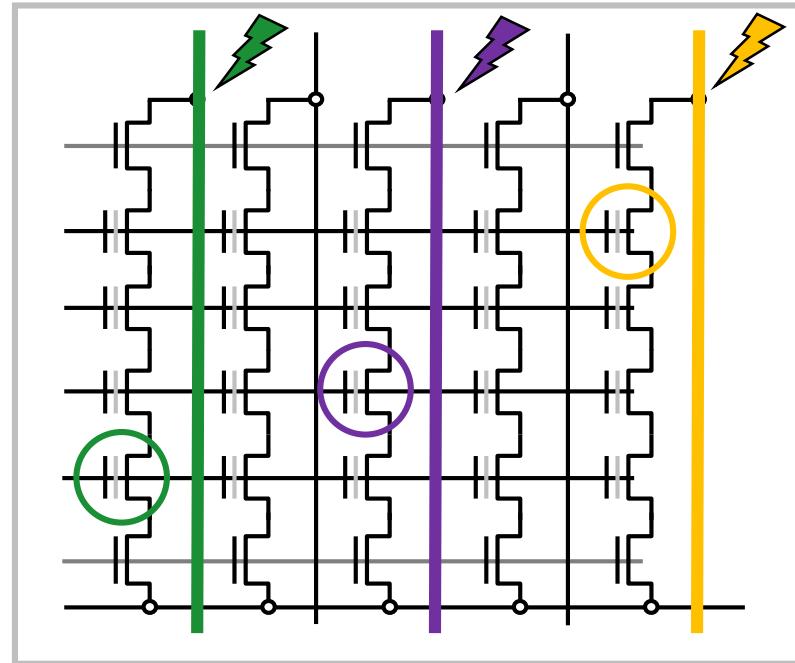


Array of electrical switches



#3: No Penalty for Random Access

Each flash cell can be addressed directly
(No need to rotate or seek like hard drives)



Array of electrical switches



Comparison w/ HDD?

+ No penalty for random accesses!

Everything work in parallel!

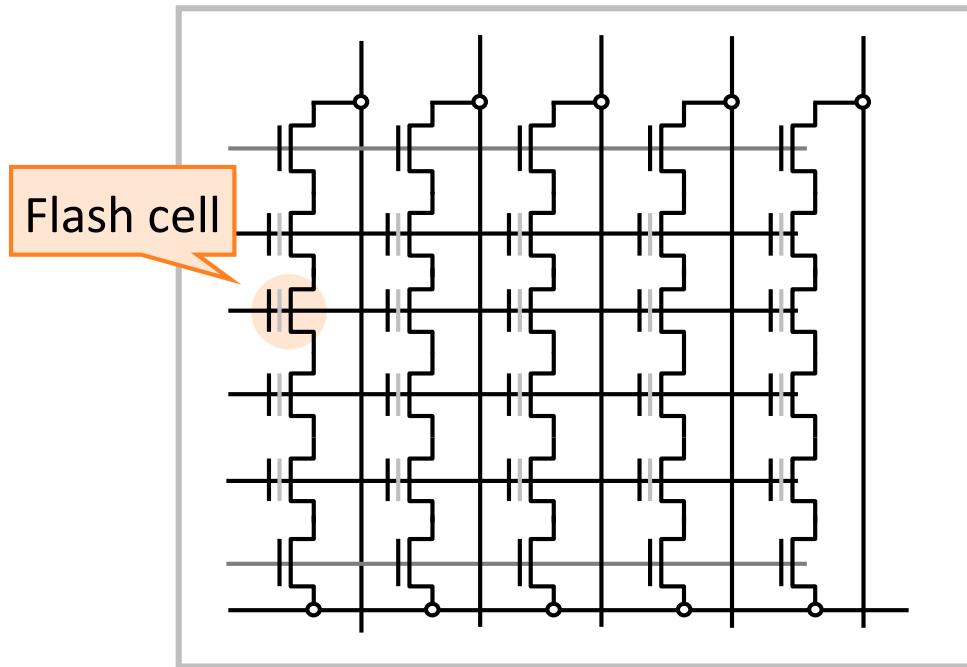
vs. HDD

Only 10,000 or 15,000 rpm SAS drives

SSD	HDD
0.1 ms	Access Times SSDs exhibit virtually no access time 5.5-8.0 ms
SSDs deliver at least 6000 io/s	Random I/O Performance SSDs are at least 15 times faster than HDDs 400 io/s
SSDs have a failure rate of less than 0.5%	Reliability This makes SSDs 4-10 times more reliable 2-5%
SSDs consume between 2 and 5 watts	Energy Savings This means that on a large server, approximately 100 watts are saved 6 and 15 watts
SSDs have an average I/O wait of 1%	CPU Power You will have an extra 6% of CPU power for other operations 7%
The average service for an I/O request during a backup remain below 20 ms	Input/Output Request Times SSDs allow for much faster data access 400-500 ms
SSD backups take about 6 hours	Backup Rates SSDs allow for 3-5 times faster backup for your data 20-24 hours

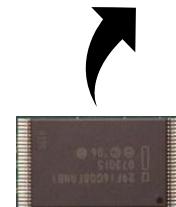
No moving parts!

Principle of Flash Memory



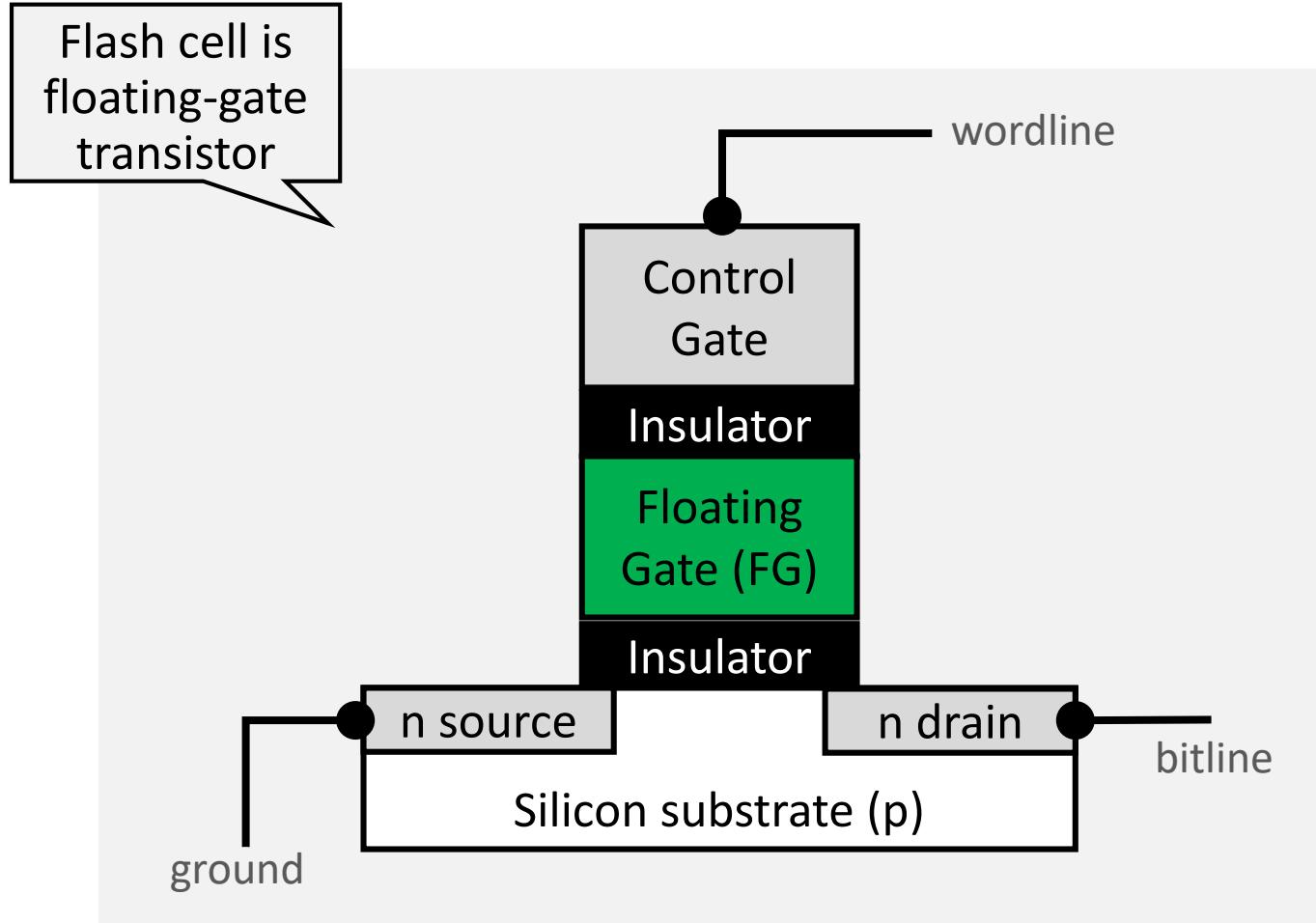
Let's check the principle
of flash cell and how it
can be configured as
flash memory

Array of electrical switches





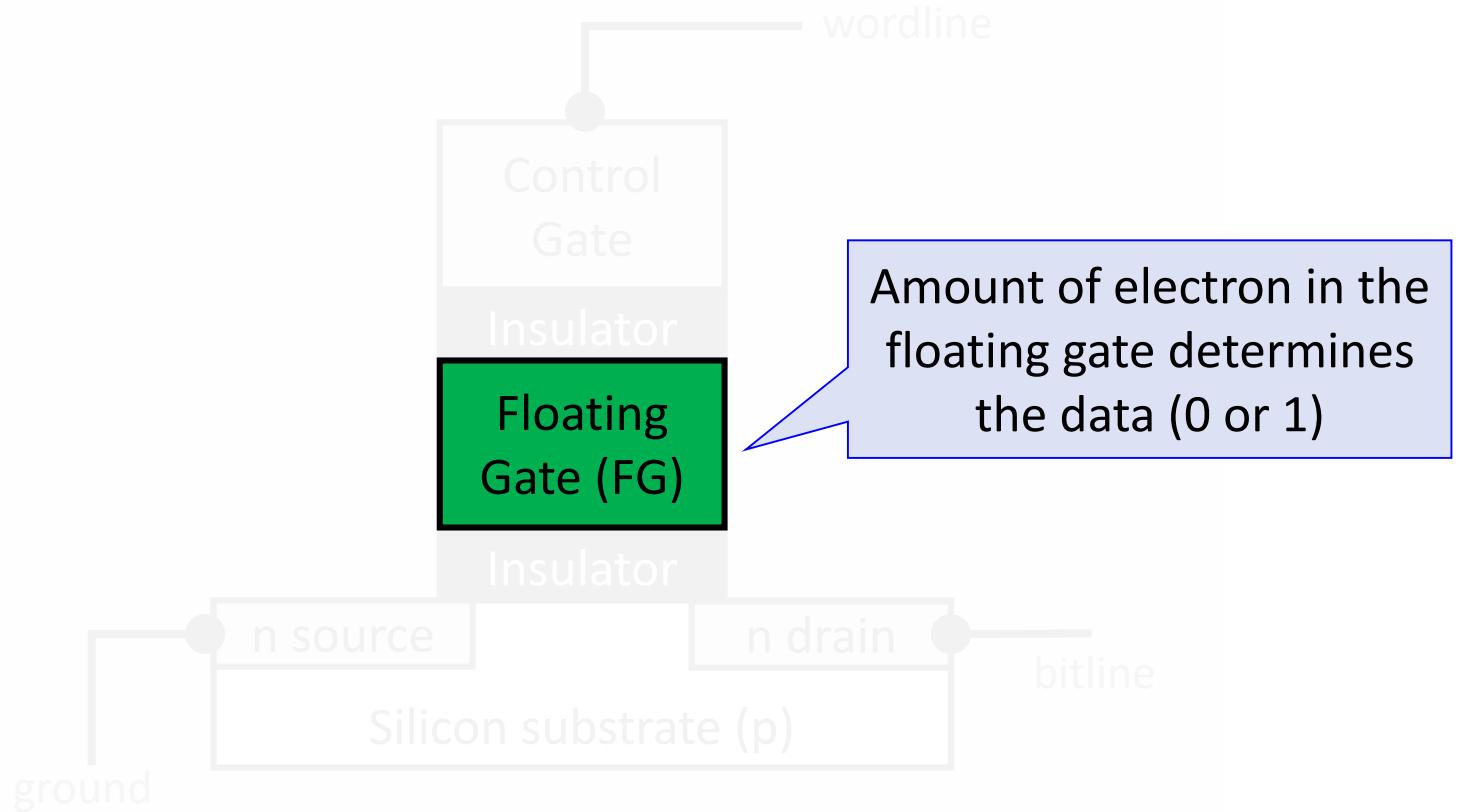
Flash Cell: Floating Gate Transistor





Flash Cell: Floating Gate Transistor

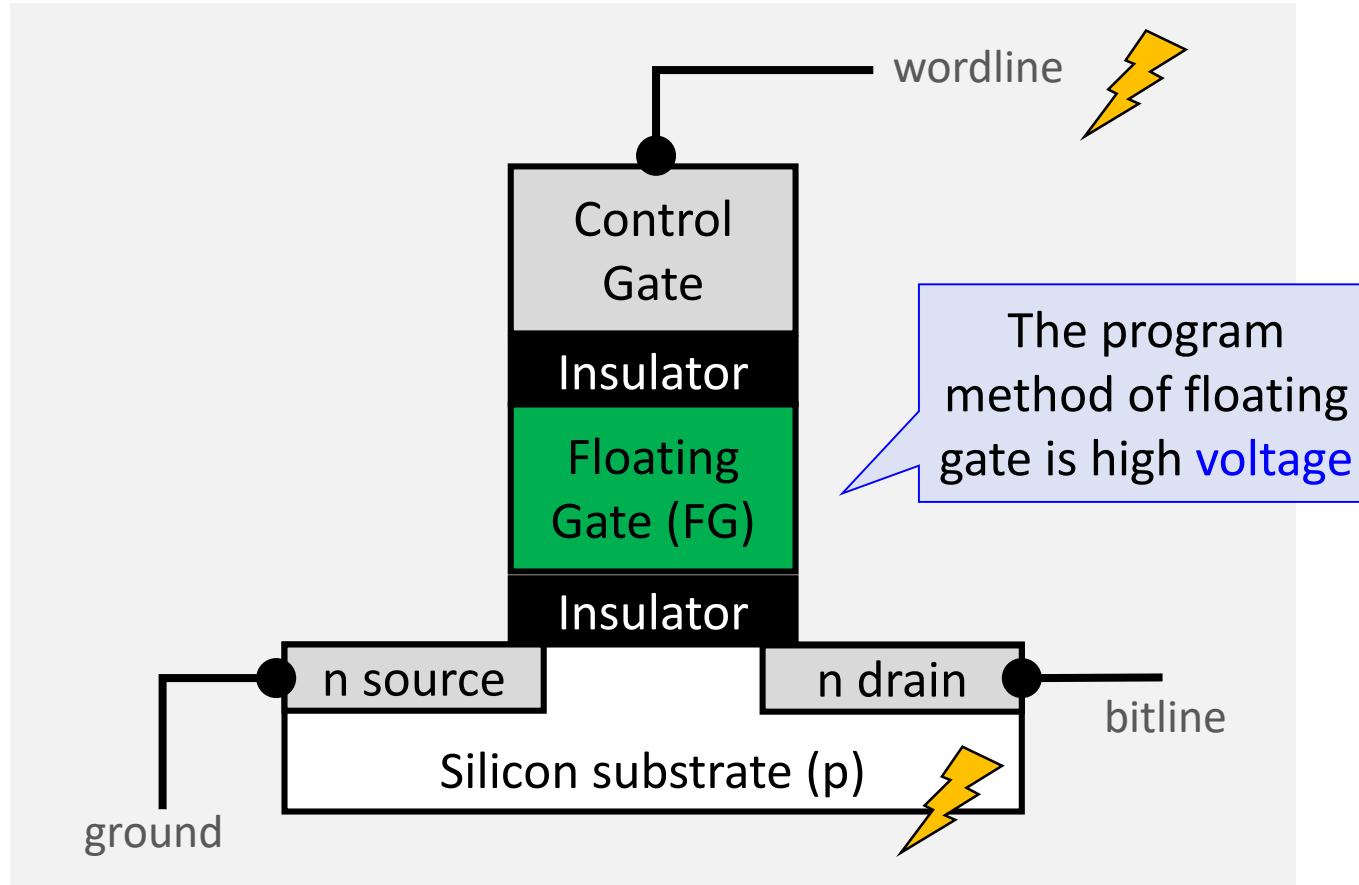
- State of the flash cell is interpreted as binary data





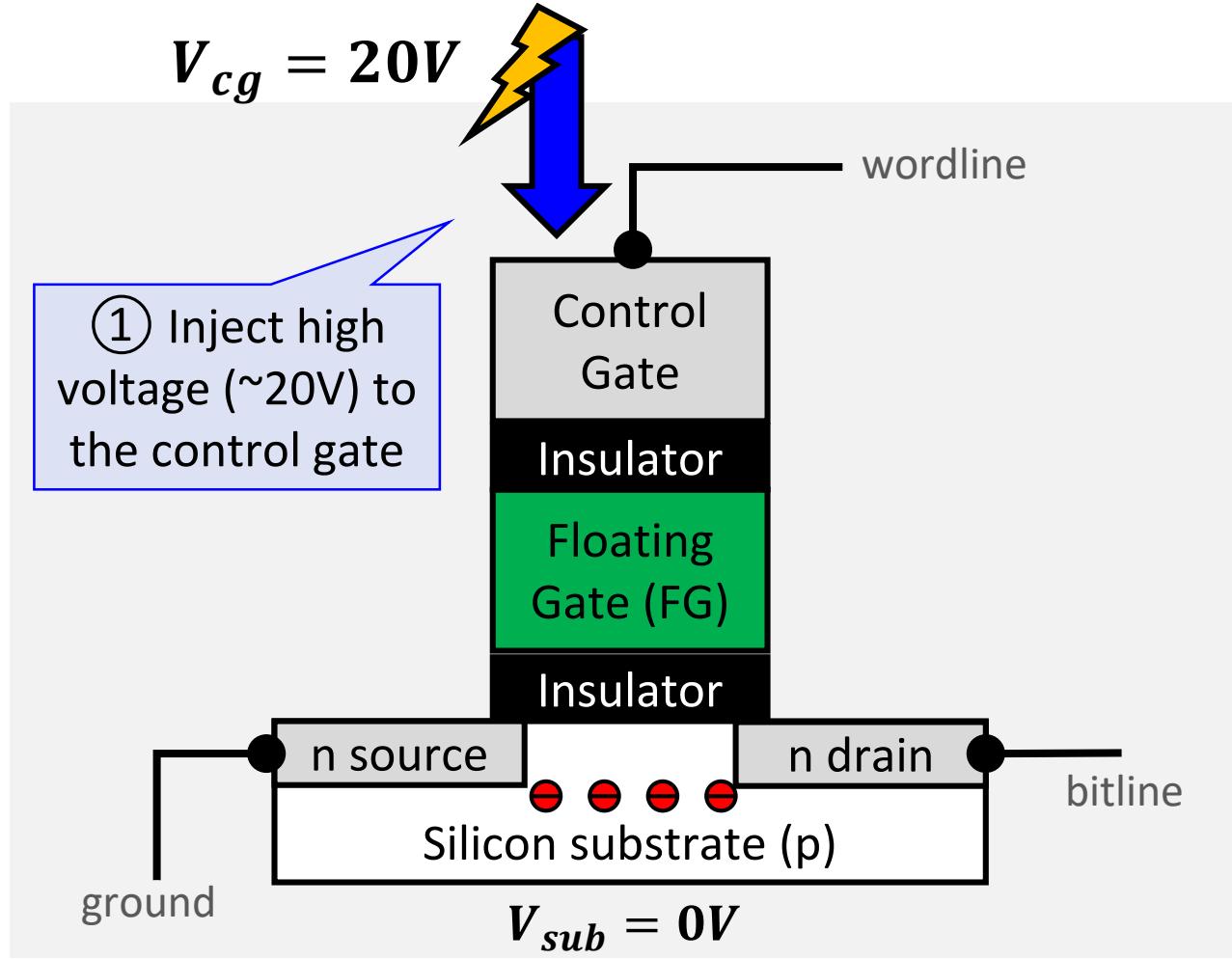
Flash Cell: Floating Gate Transistor

- State of the flash cell is interpreted as binary data



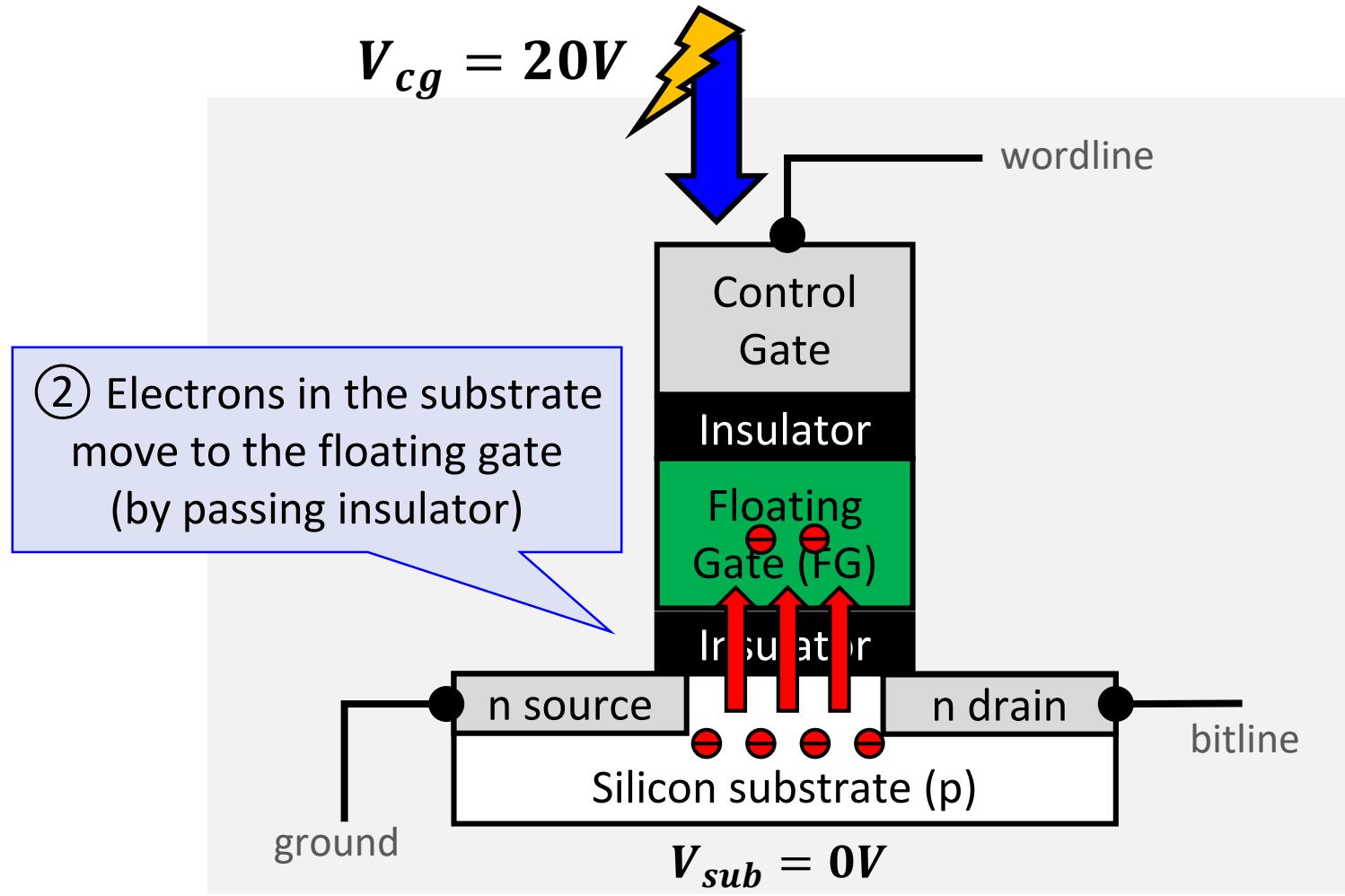


Principle of Flash Program



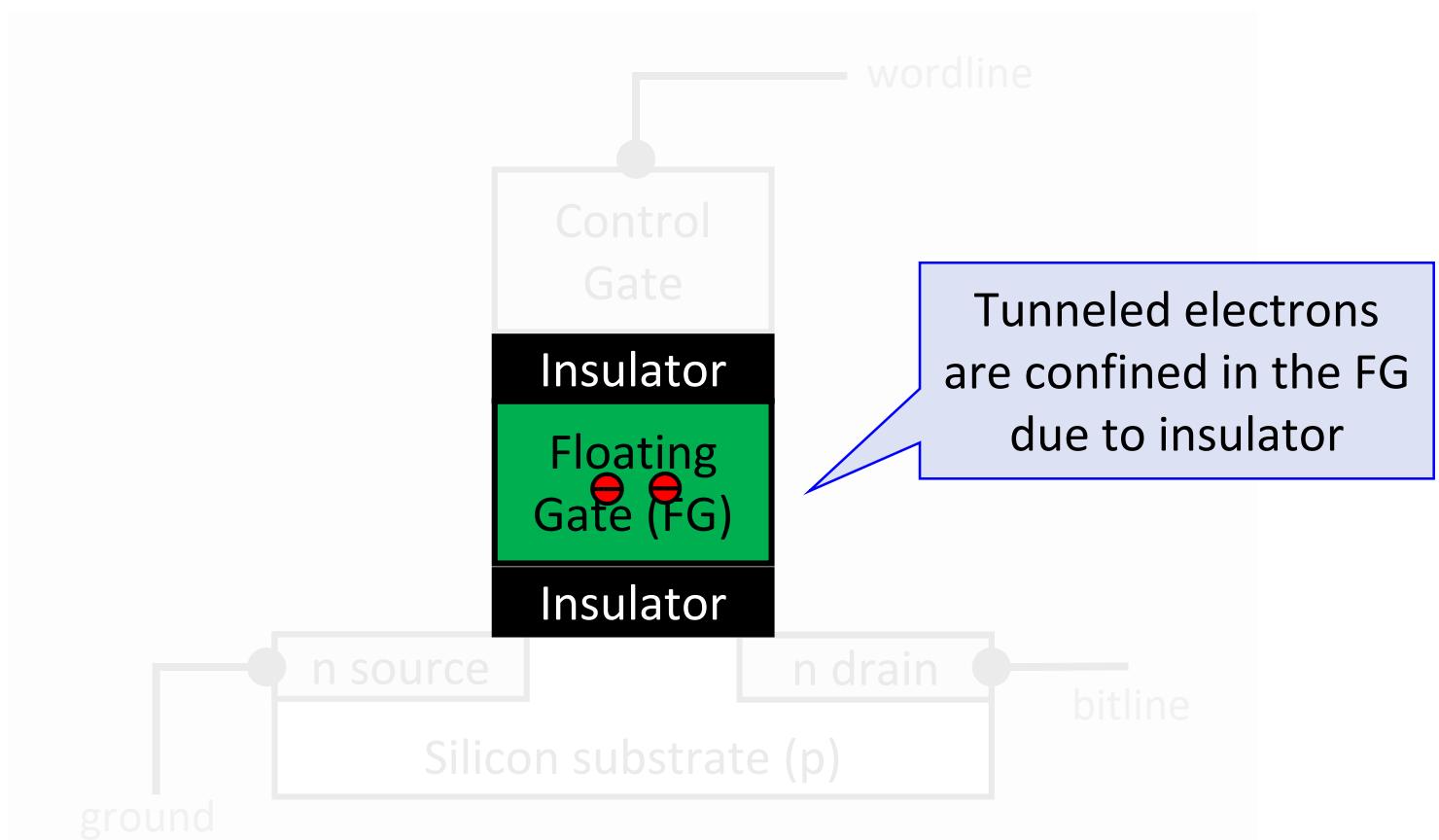


Principle of Flash Program



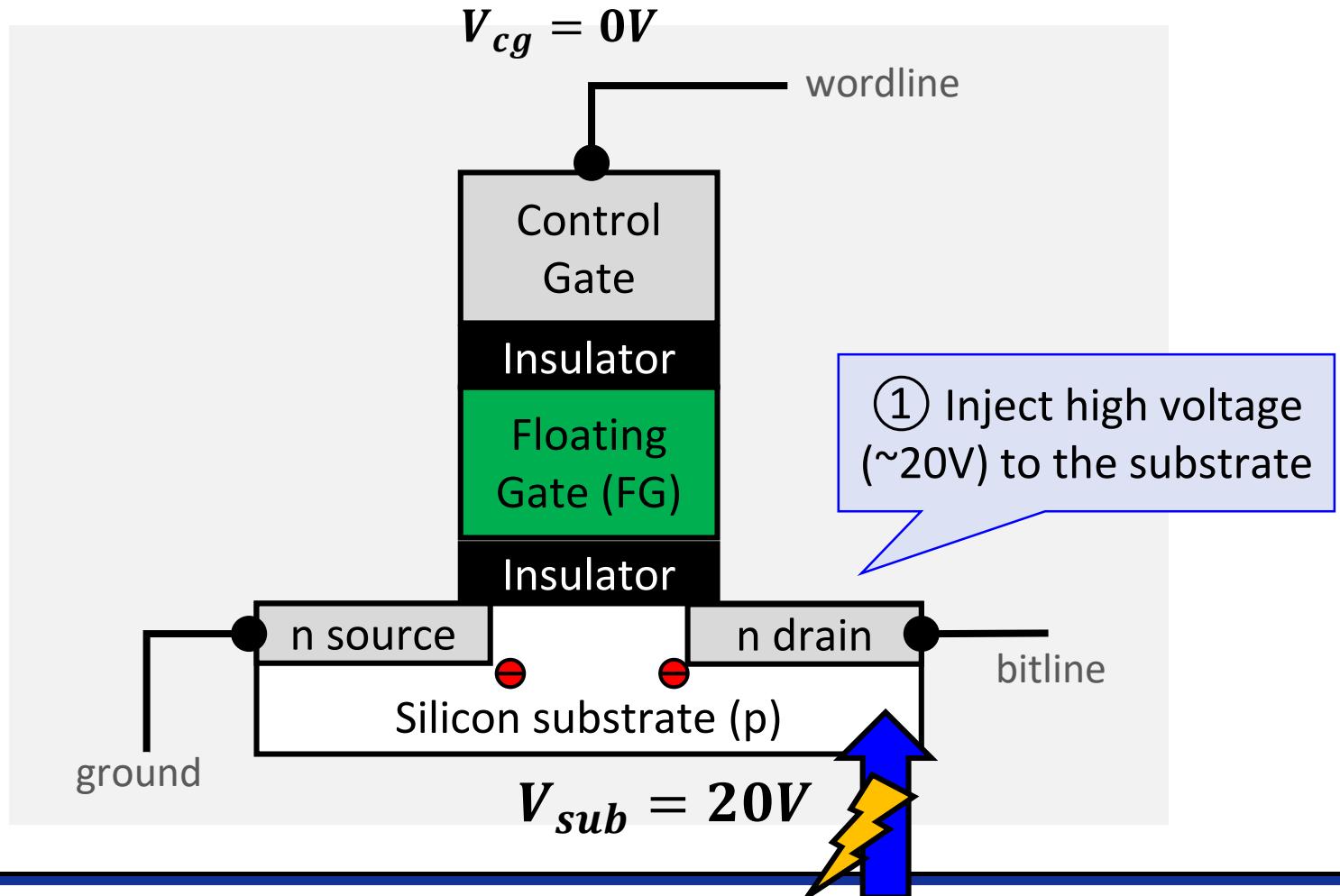


Principle of Flash Program



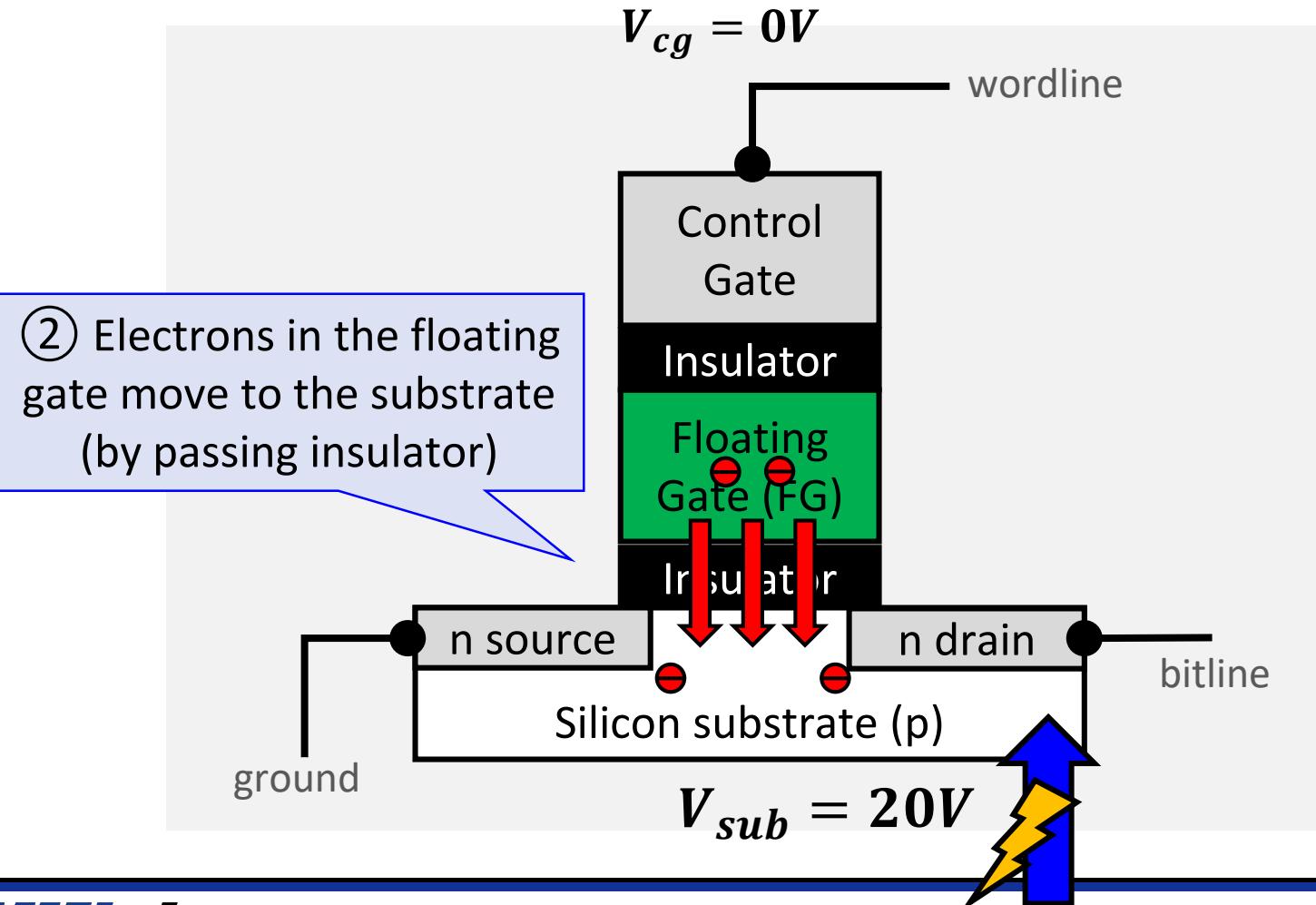


Principle of Flash Erase



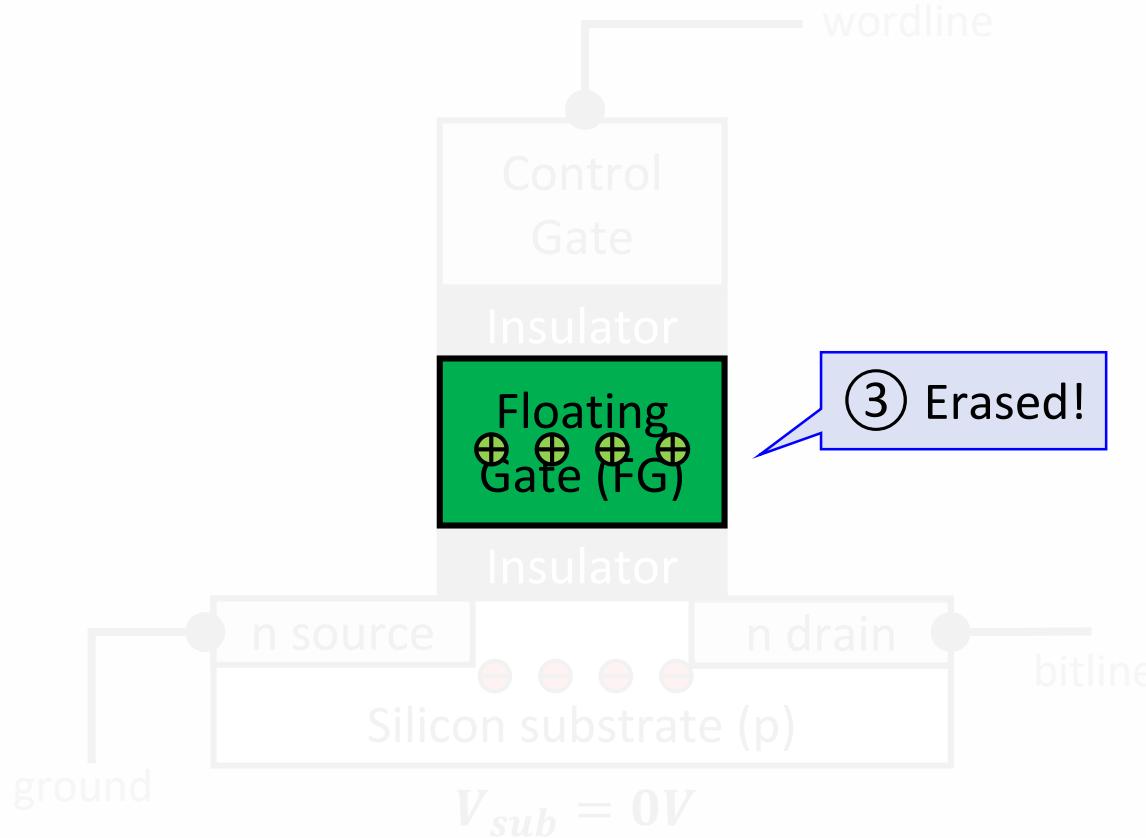


Principle of Flash Erase





Principle of Flash Erase

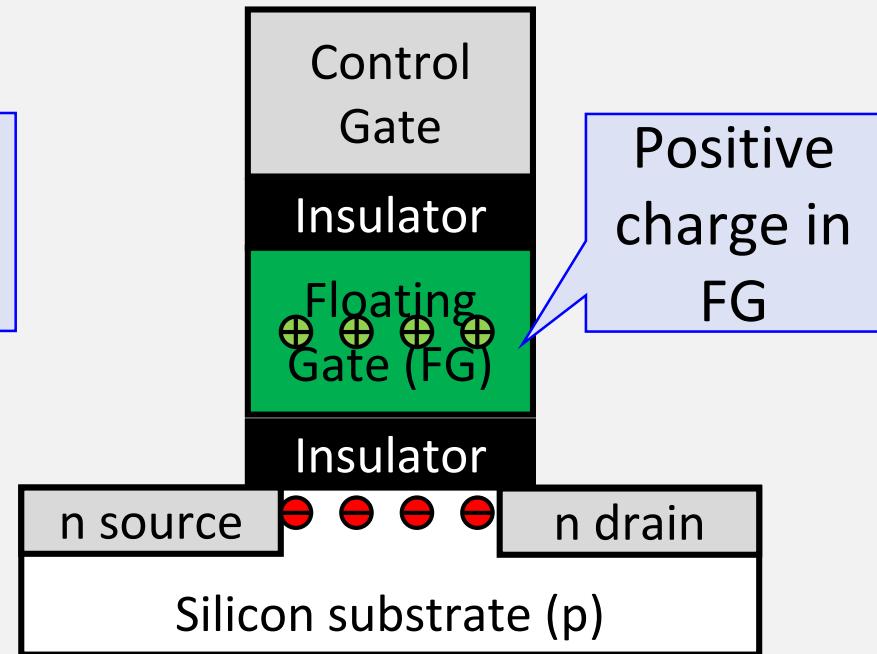
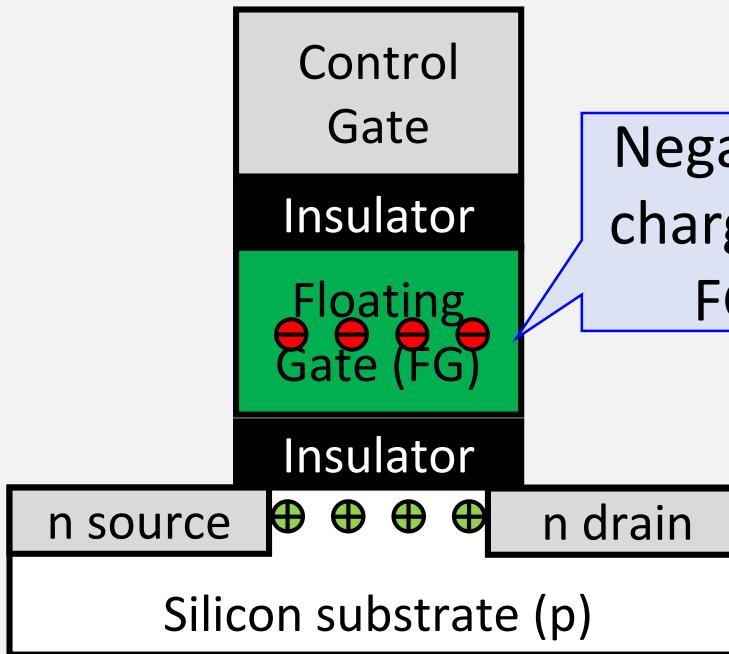




Principle of Flash Read

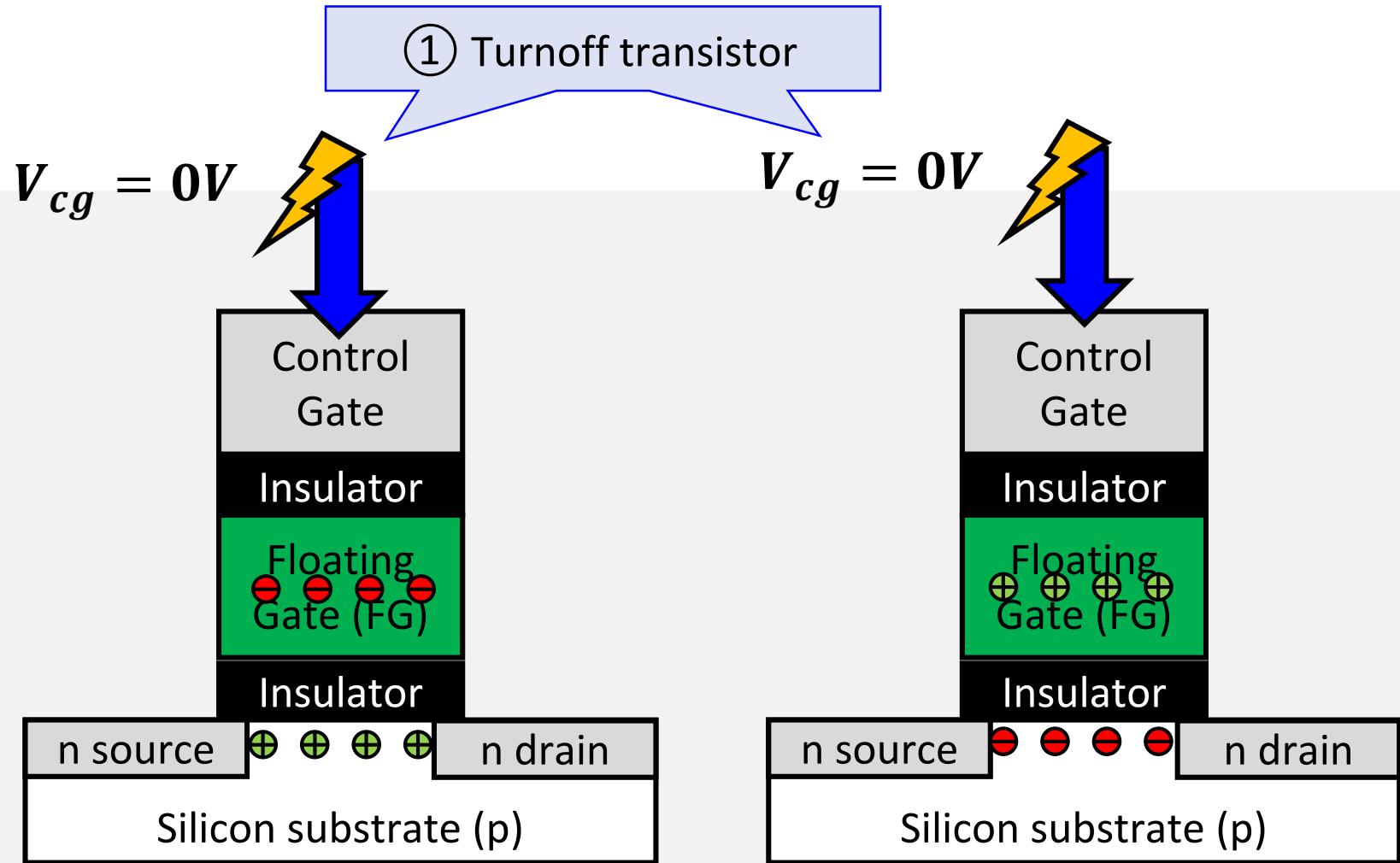
Programmed

Erased



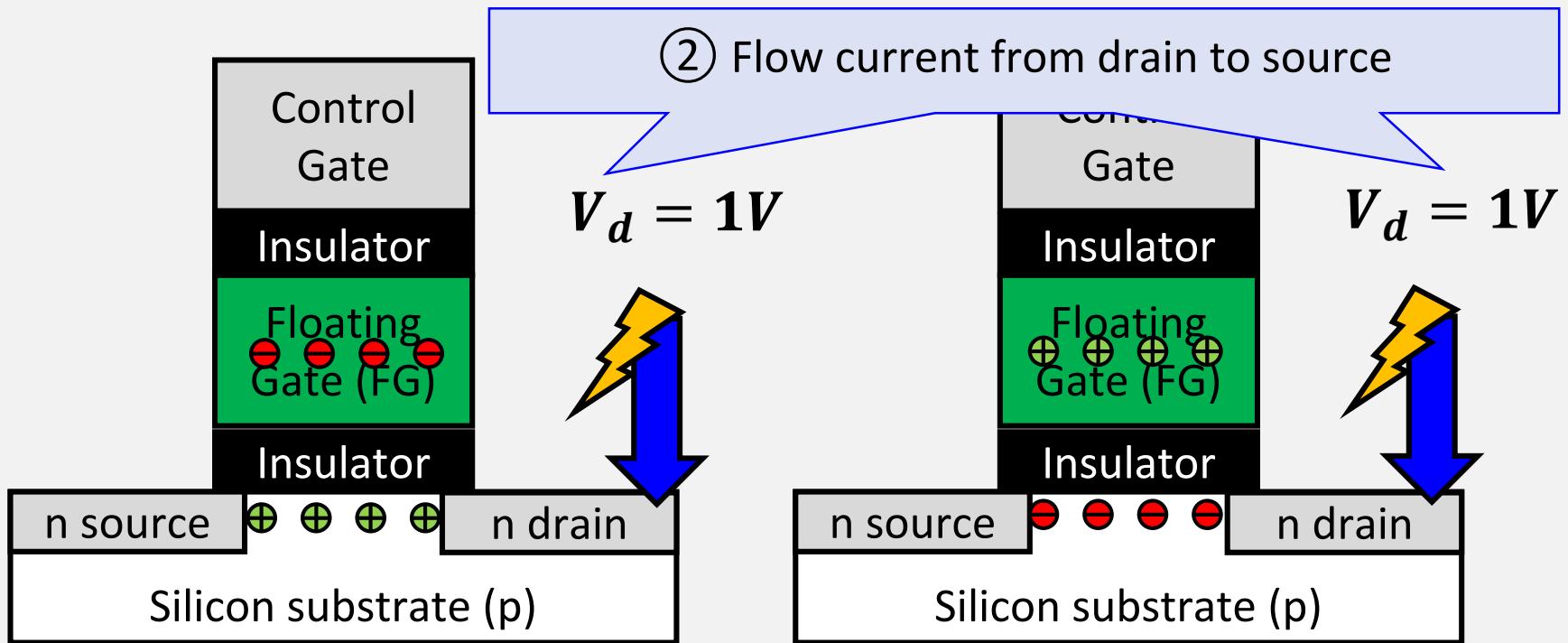


Principle of Flash Read





Principle of Flash Read

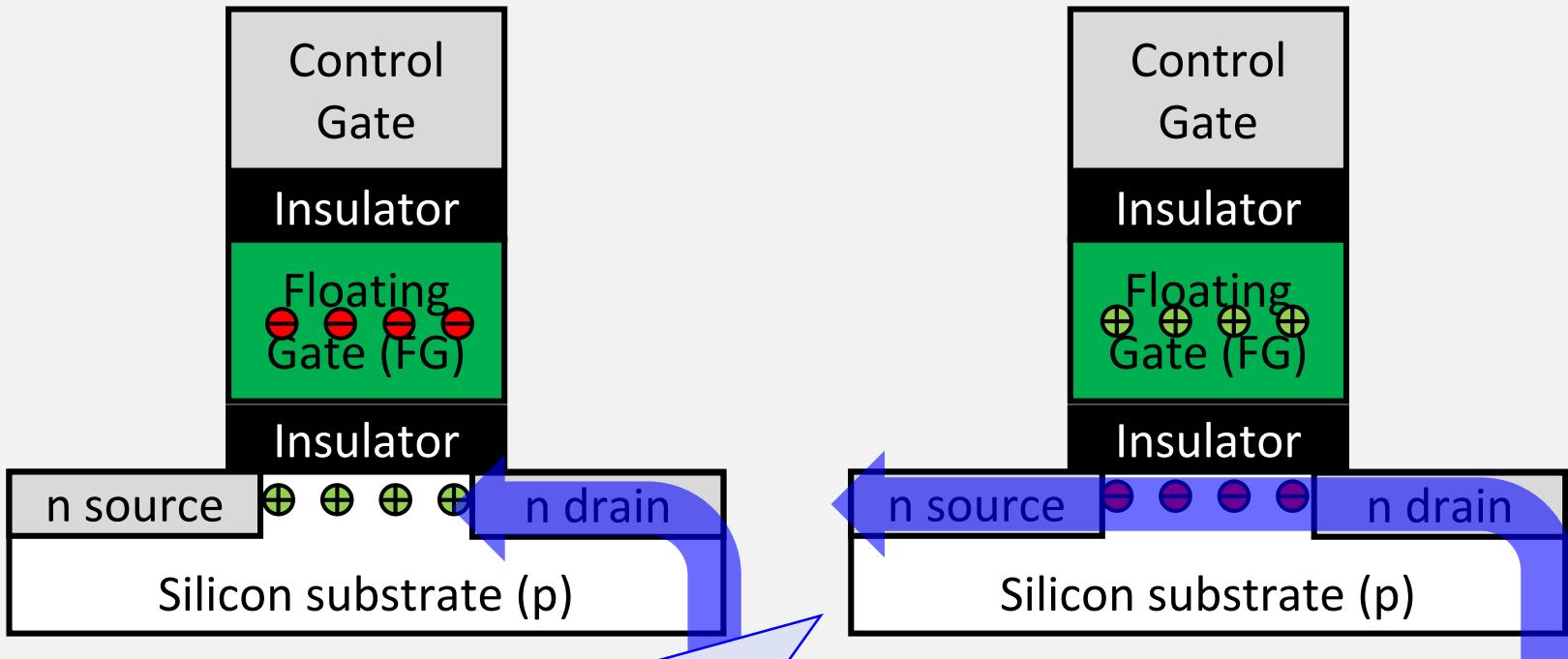




Principle of Flash Read

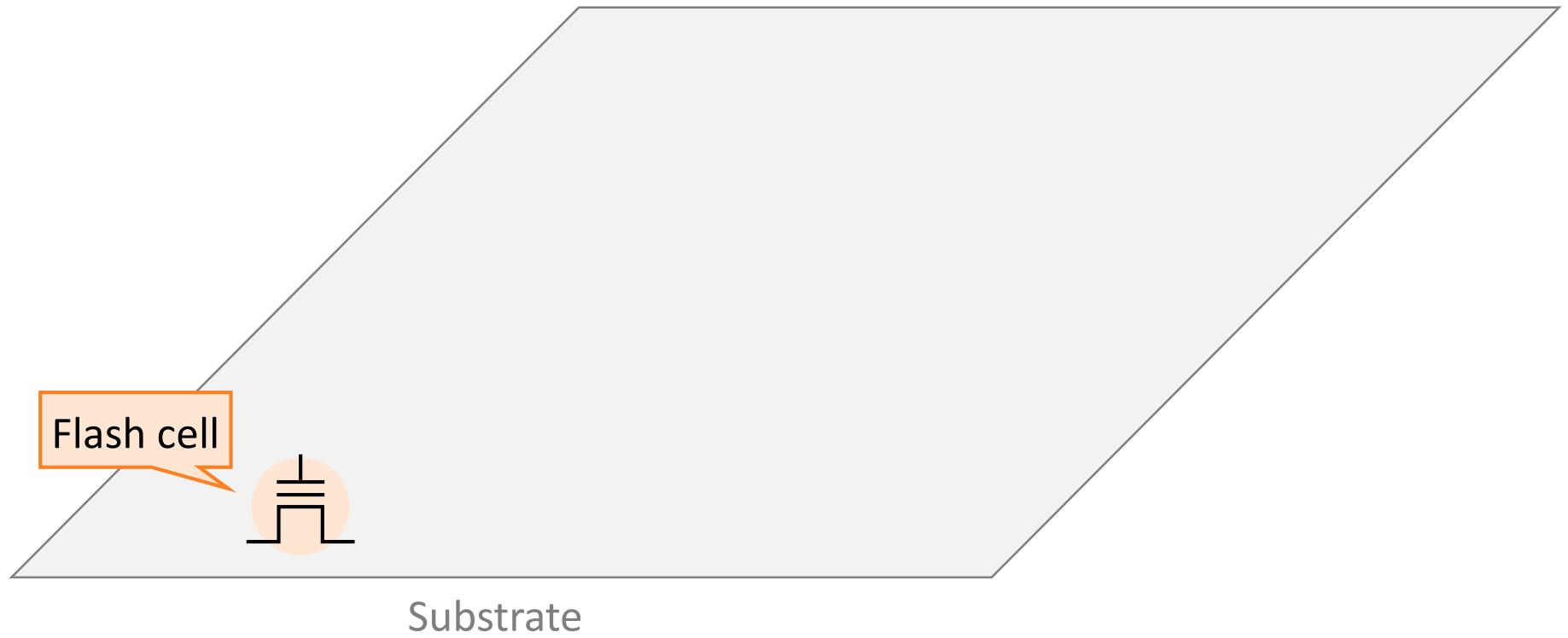
Programmed

Erased

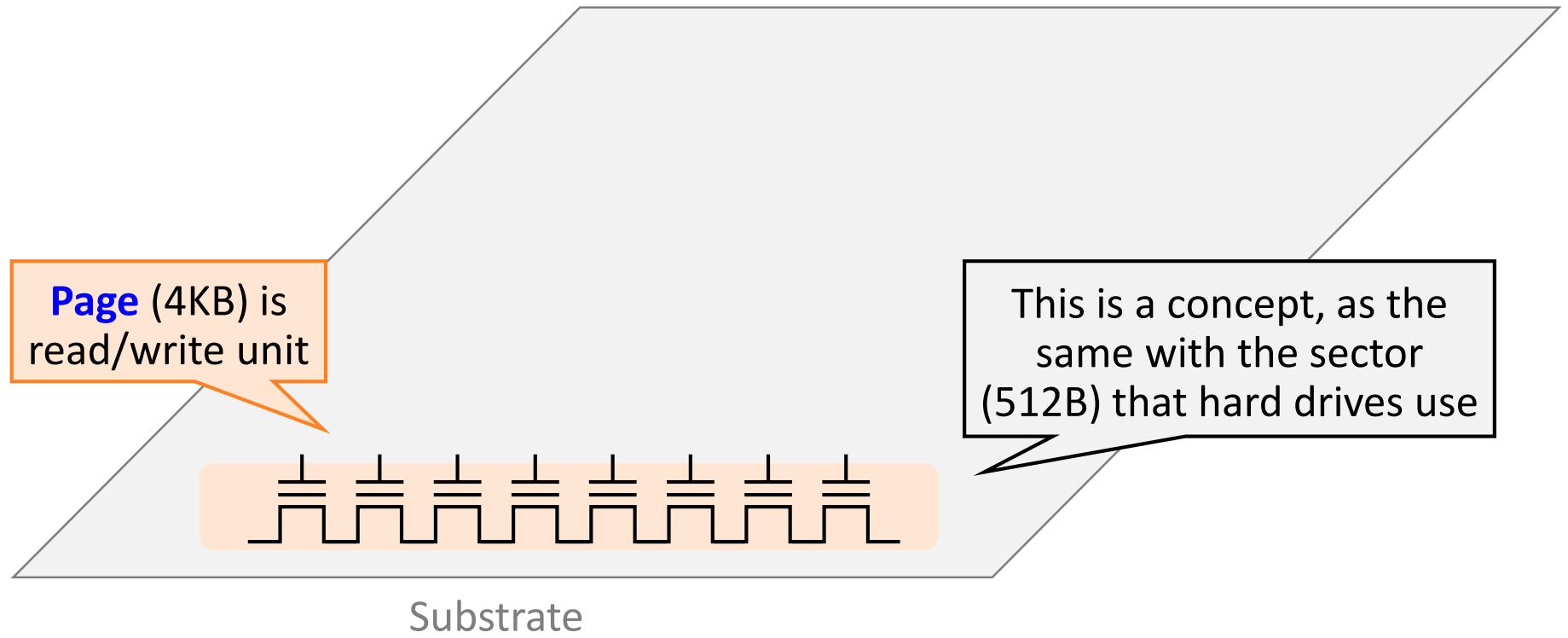


③ Erased cell → “0”,
Programmed cell → “1”

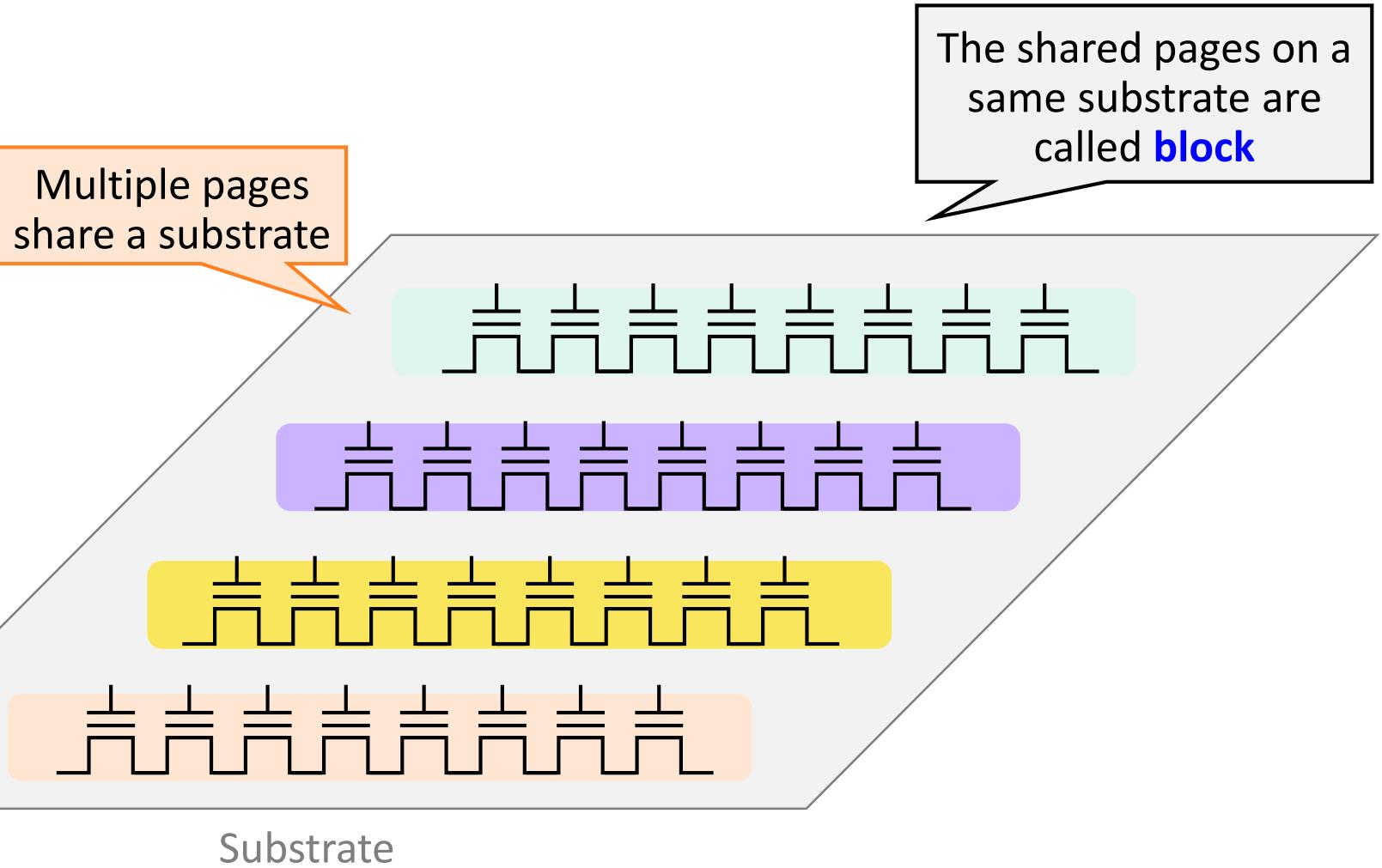
Array of Floating-gate Transistor



Array of Floating-gate Transistor

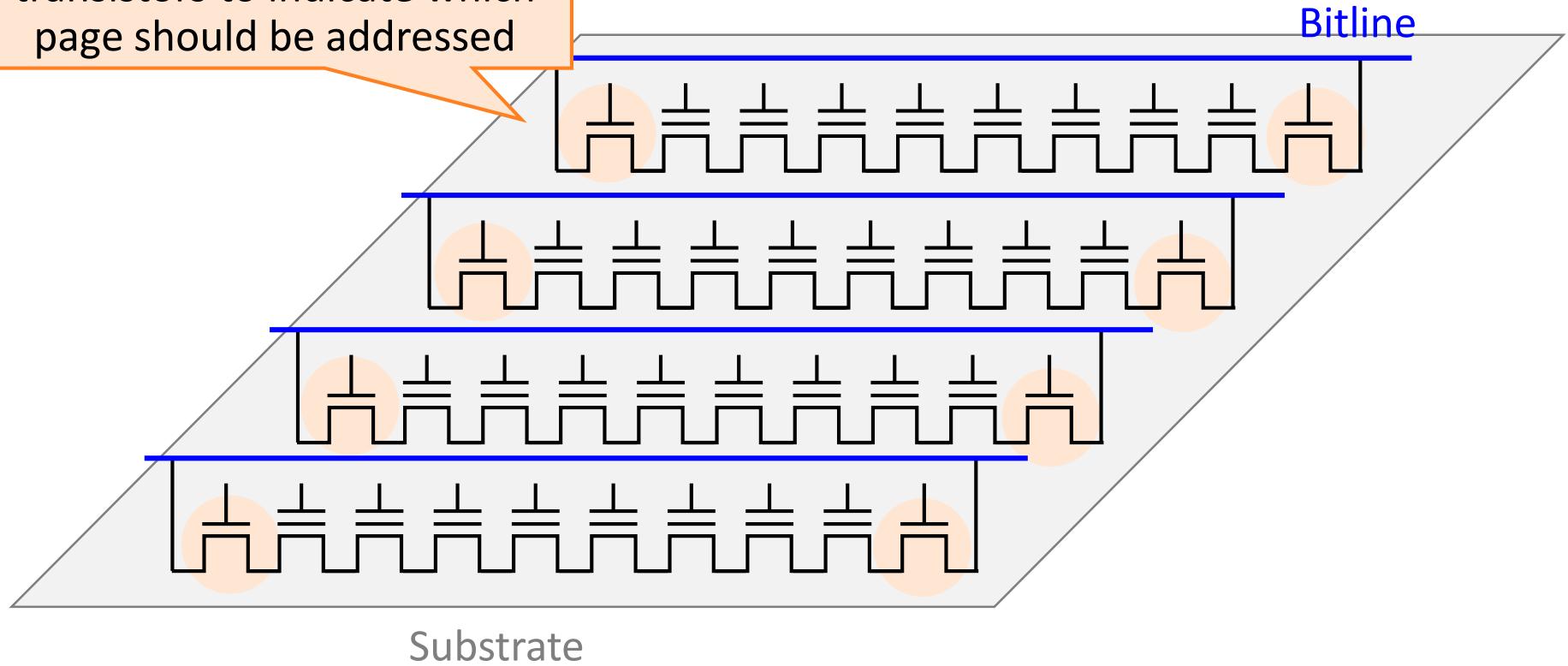


Array of Floating-gate Transistor



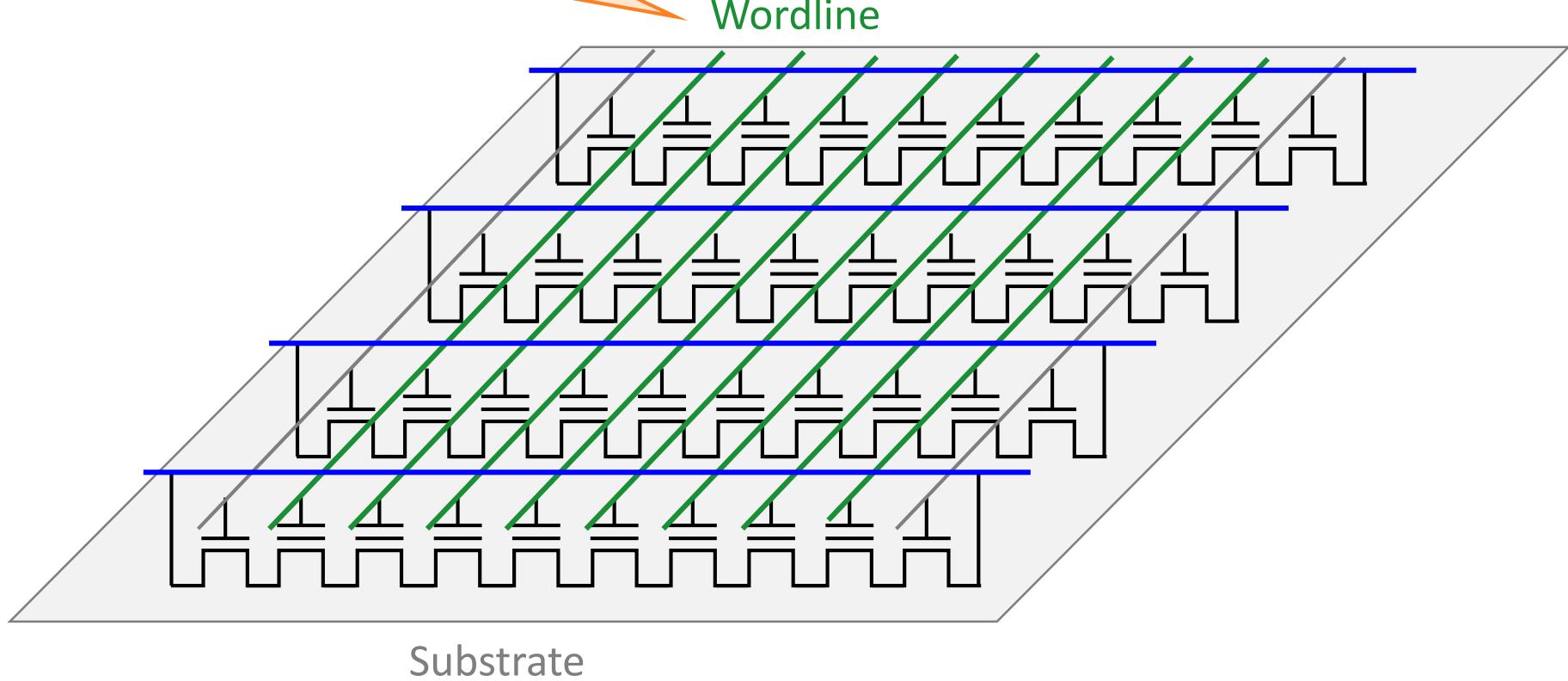
Array of Floating-gate Transistor

It requires selecting
transistors to indicate which
page should be addressed

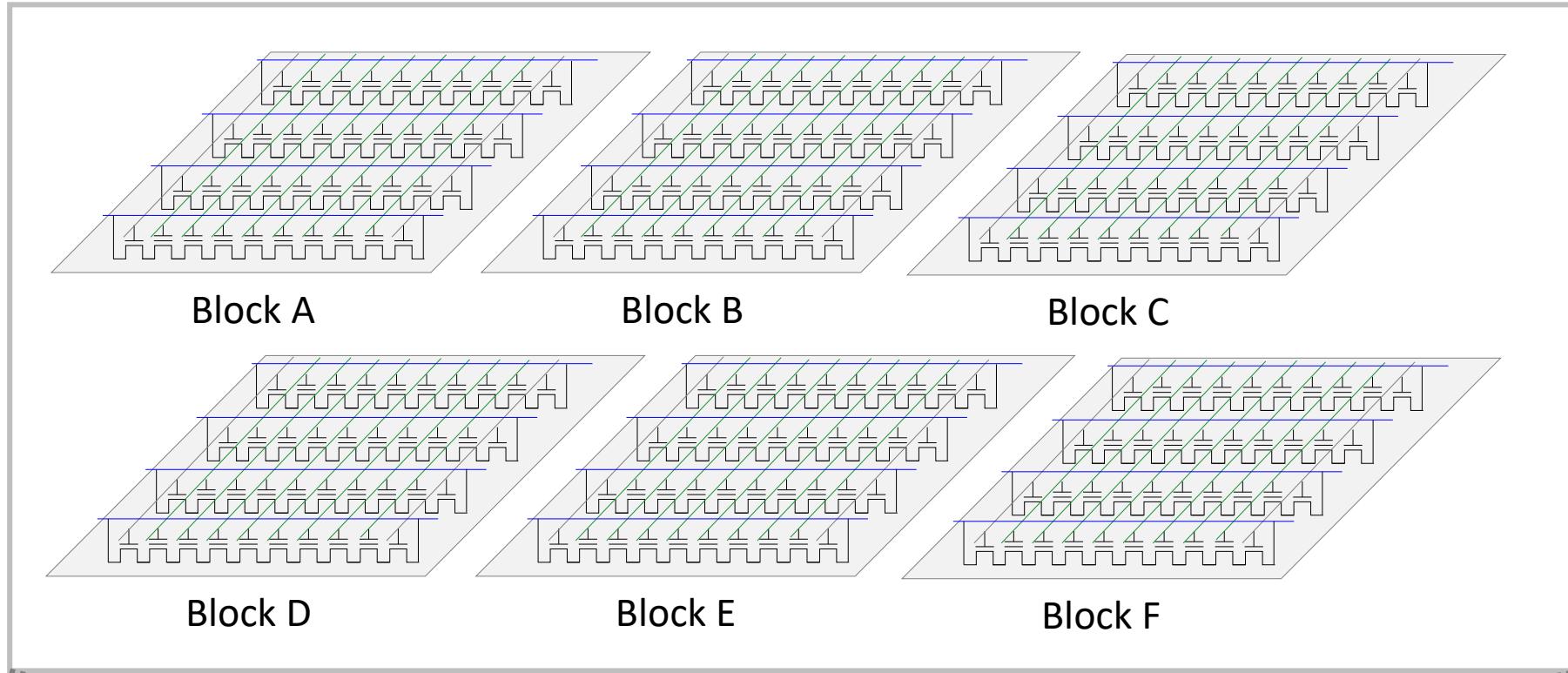


Array of Floating-gate Transistor

Each flash cell is programmed by enabling wordline (control gate)



Array of Floating-gate Transistor



Block E

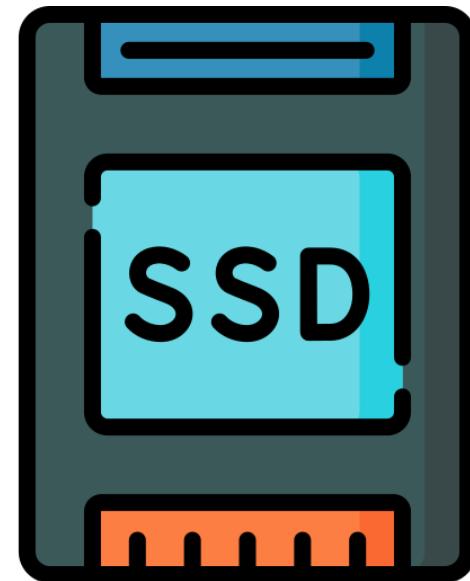
Block F



A flash memory
chip consists of
multiple blocks

What Does Principle of Flash Implies?

- SSD don't allow overwrites – erase-before-write characteristics
- I/O granularity of erases and writes are completely different (block vs. page)
- Employing many flash chips and blocks make SSD denser as well as faster



Many research components are here! If interested, take a advanced lecture for storage
<2021 Fall's EE817, advanced flash storage>

What's Next?

- We will address an important question regarding how to **manage storage** from the next lecture

21S EE415

File System I

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMELab 

File System I

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab* 

File System



HDDs/SSDs are just a kind of block storage; how can we manage them with meaningful data?

What is a File System?

All data are stored as binary digits ('1's and '0's) on storage media

How can we find/recognize all the data from the sea of binary data



File systems are originally designed toward organizing the data in an understandable and easy-to-access way

Files

A file is **named bytes** on disk





File Name

File names are often written in dotted notation, **extensions** (e.g., program.**exe**, image.**jpg**, music.**mp3**)



File Components

A file can contain anything and it is one or more blocks of binary data



A file is composed of two components

Metadata about the file (to support “Named” feature)

010101001
001010111
011110101
101010111
101111011

Blocks

- Name
- Total size
- Created/modified/ access time
- Owner, Permissions (read/write/execute)



More File Metadata



Files have additional metadata that is **not** typically shown to users



More File Metadata

myData.jpg



ID = 0

myData.zip



ID = 1

myData.pdf

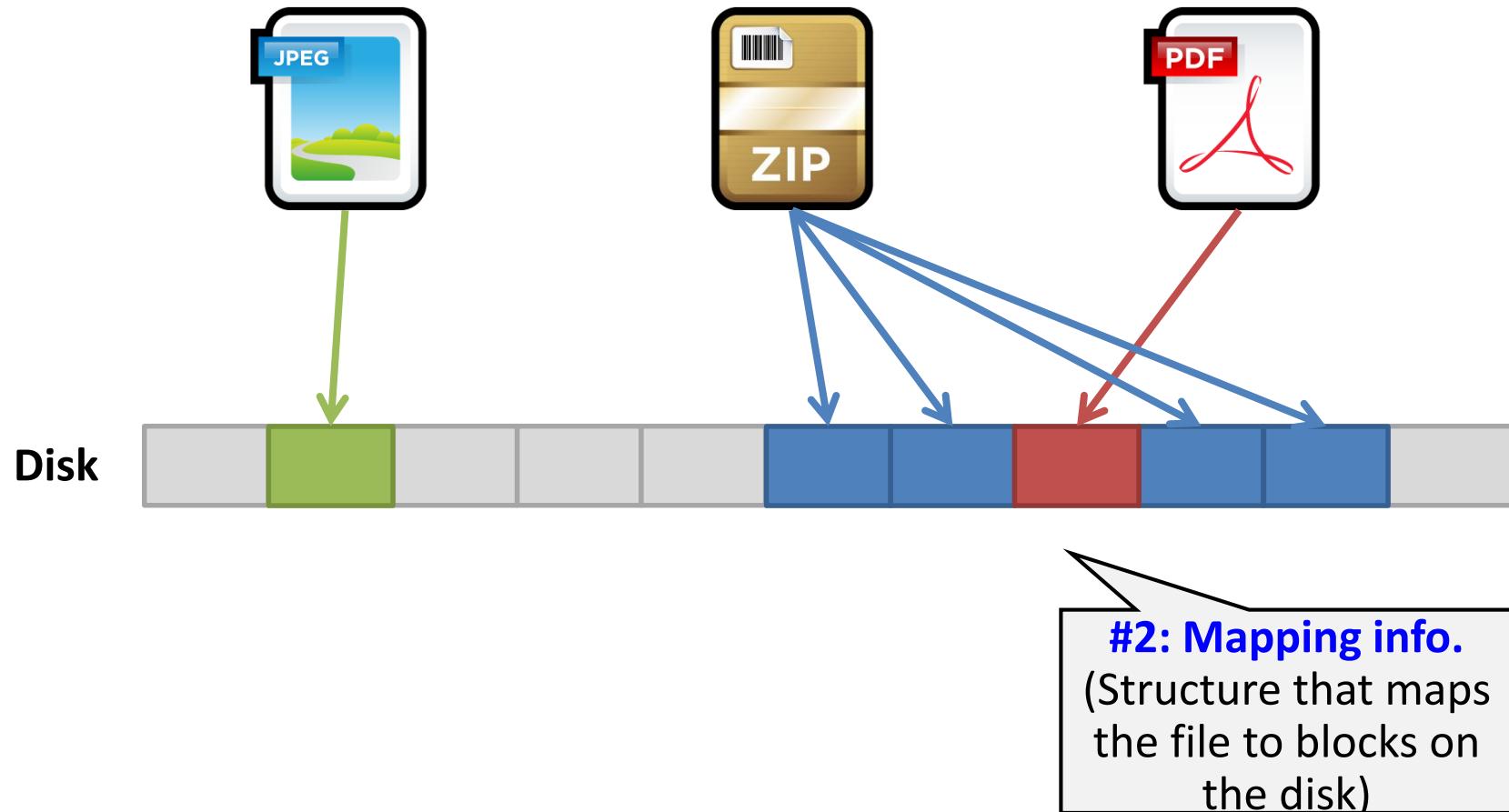


ID = 2

#1: Unique identifier
(file names may not
be unique)

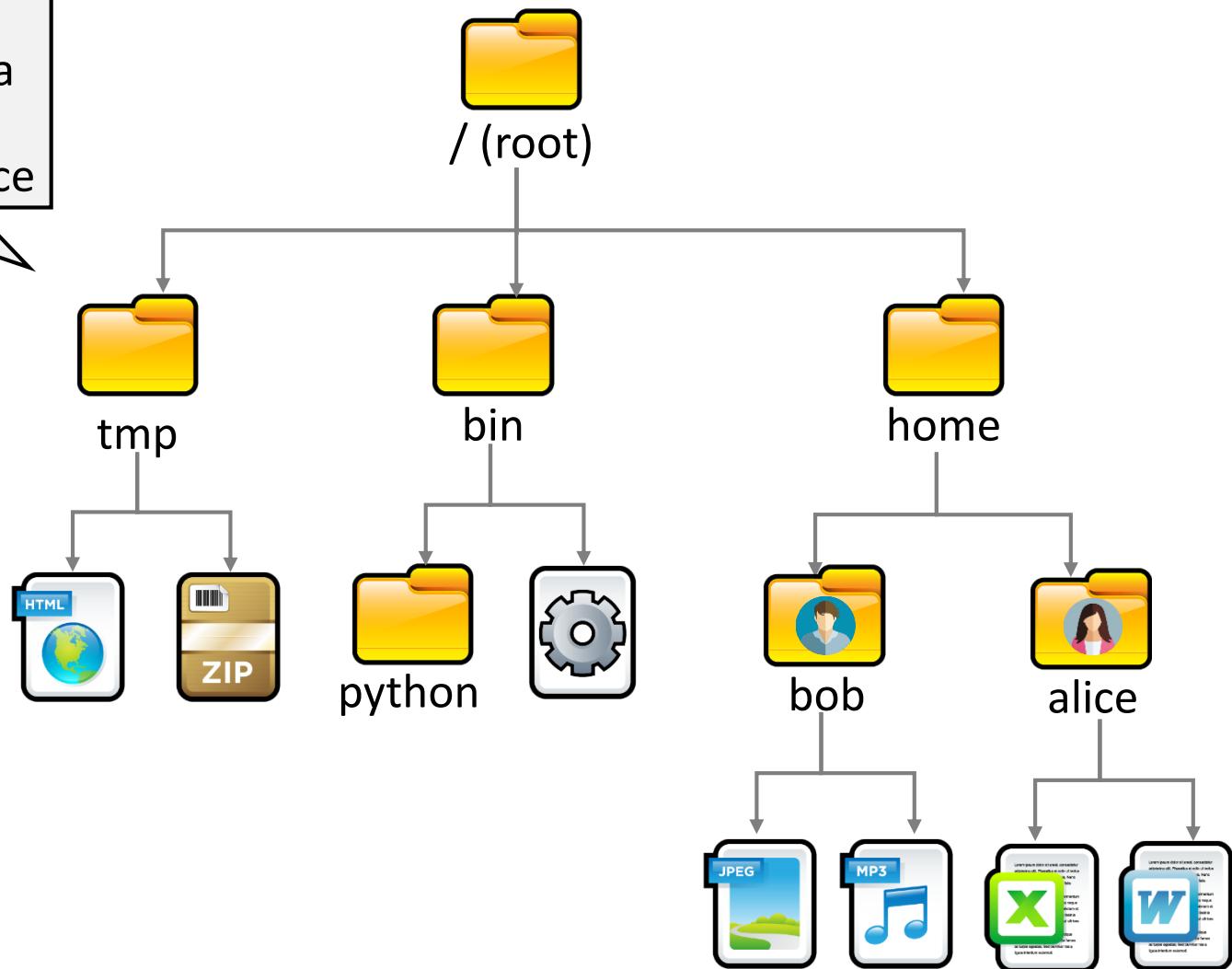


More File Metadata



File Hierarchy

Traditionally, file systems have used a **hierarchical**, tree-structured namespace





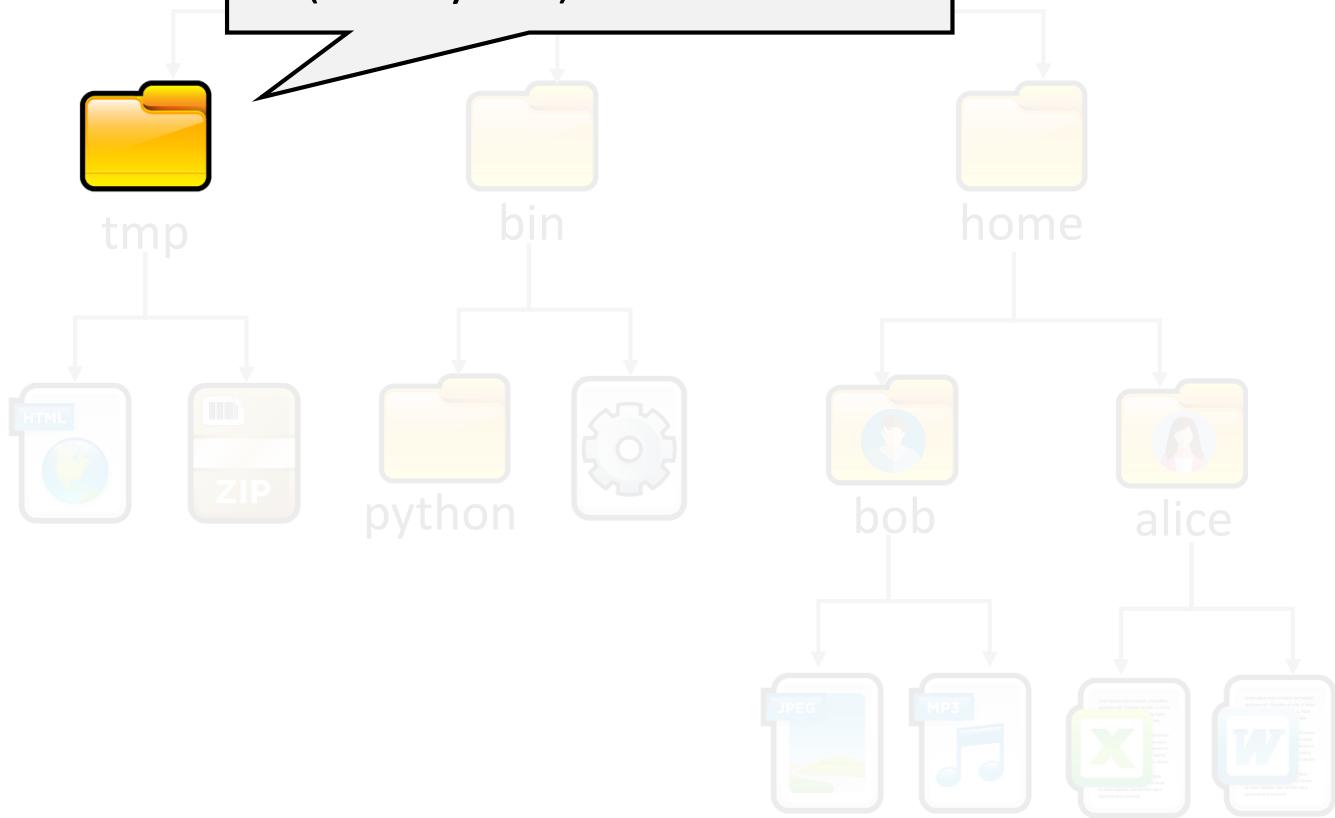
Files



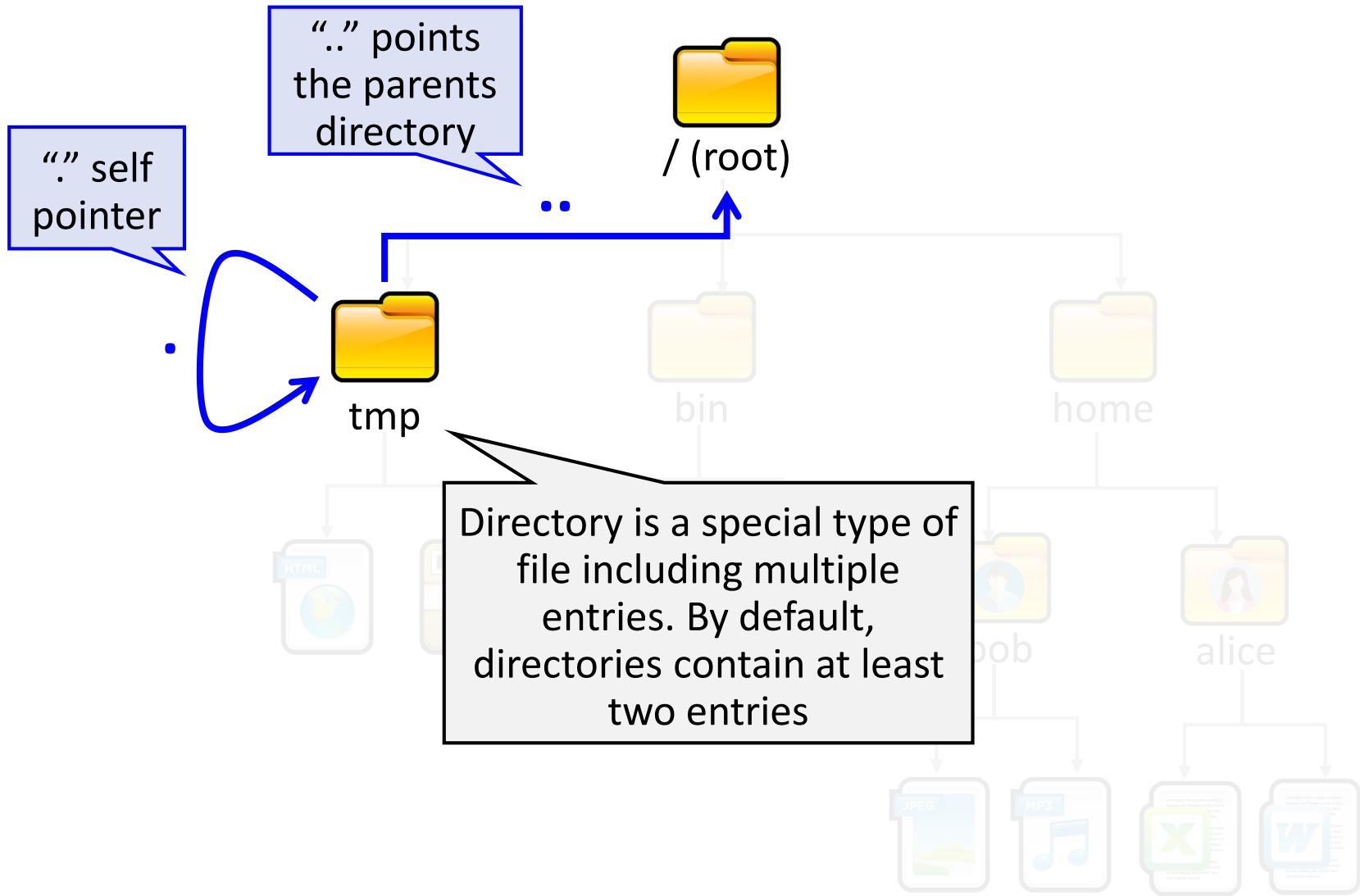


Directories

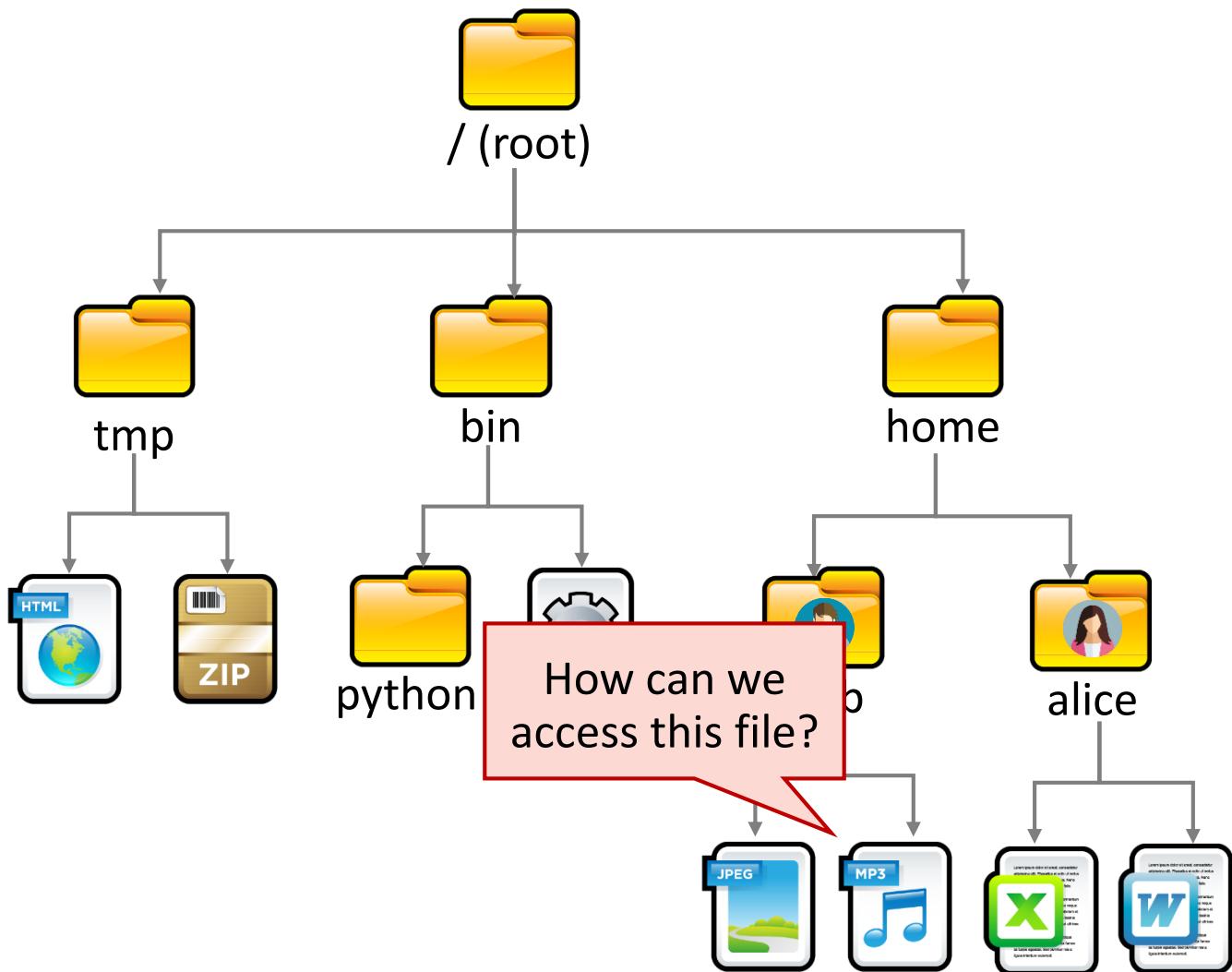
Directories are objects that contain other objects.
In other words, a directory may (or may not) have children



Directories

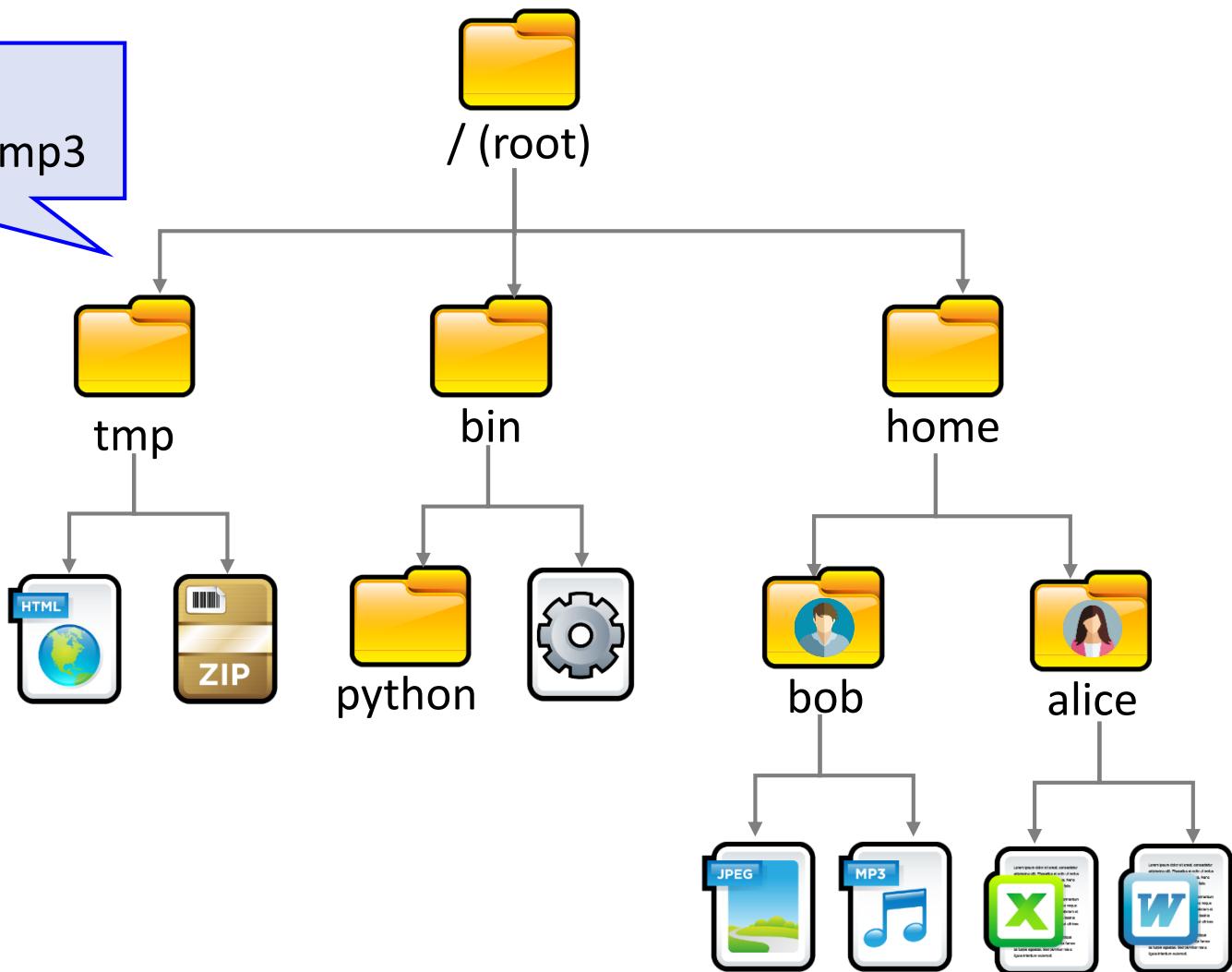


File Navigation

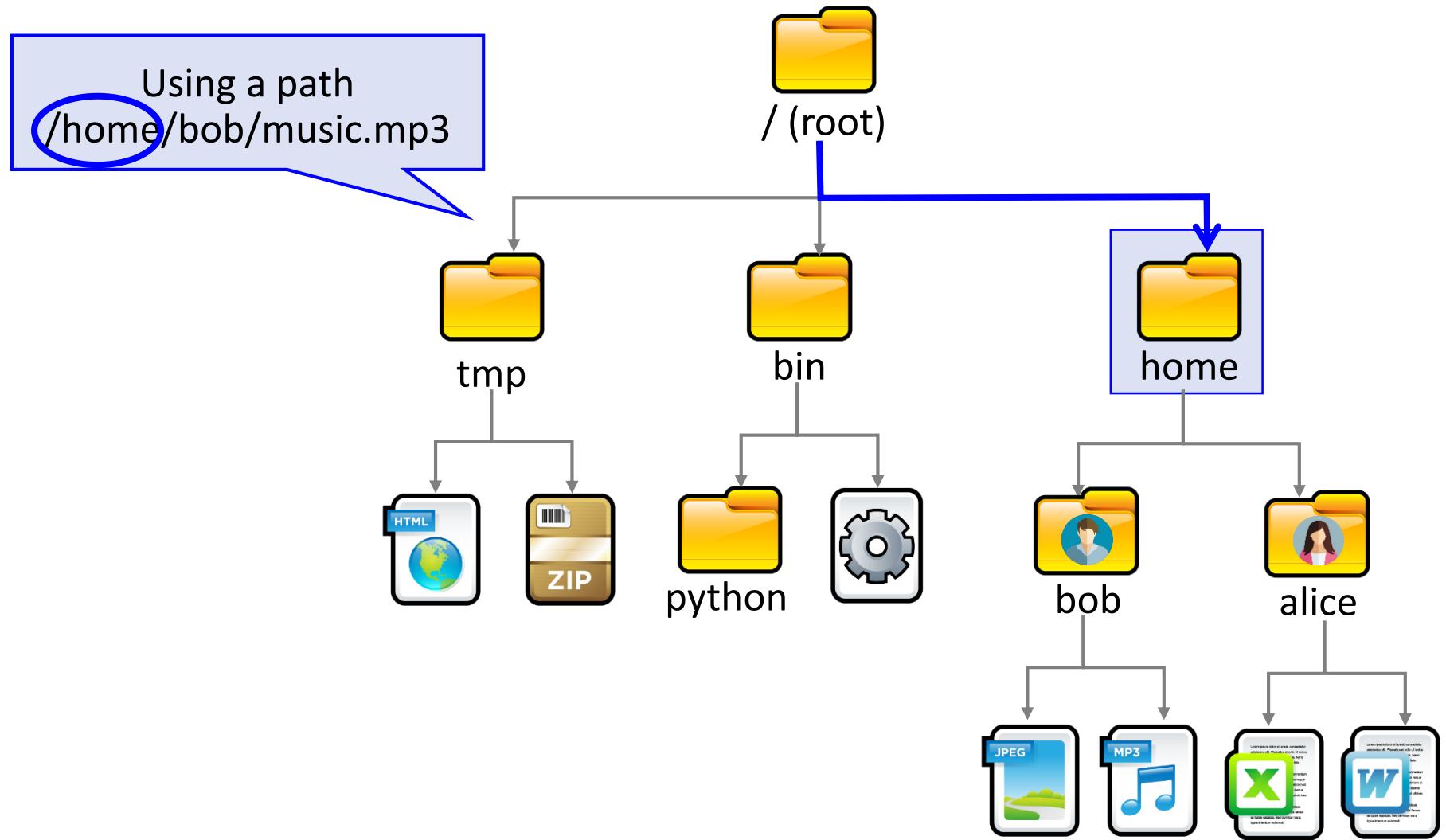


File Navigation

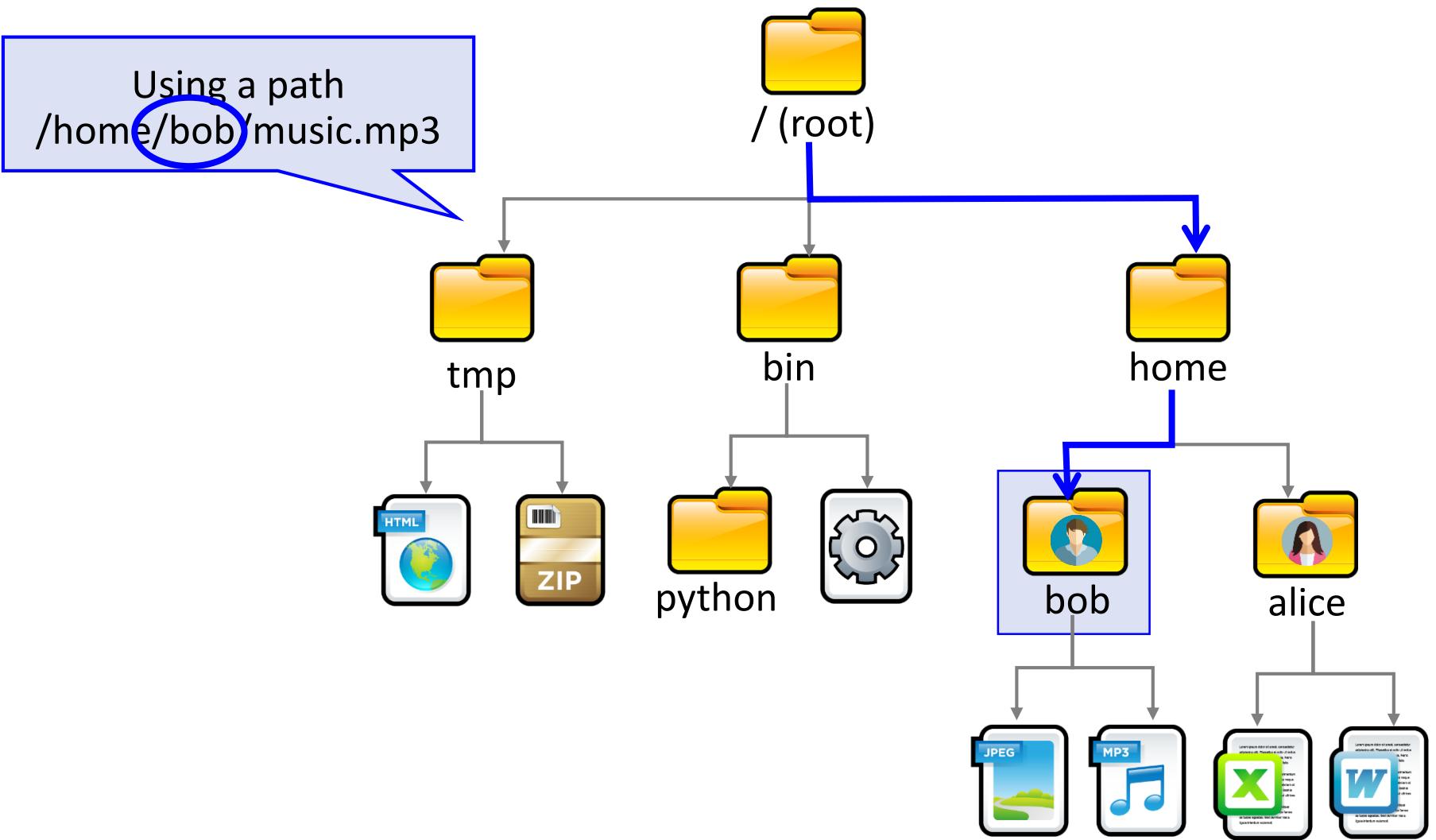
Using a path
/home/bob/music.mp3



File Navigation

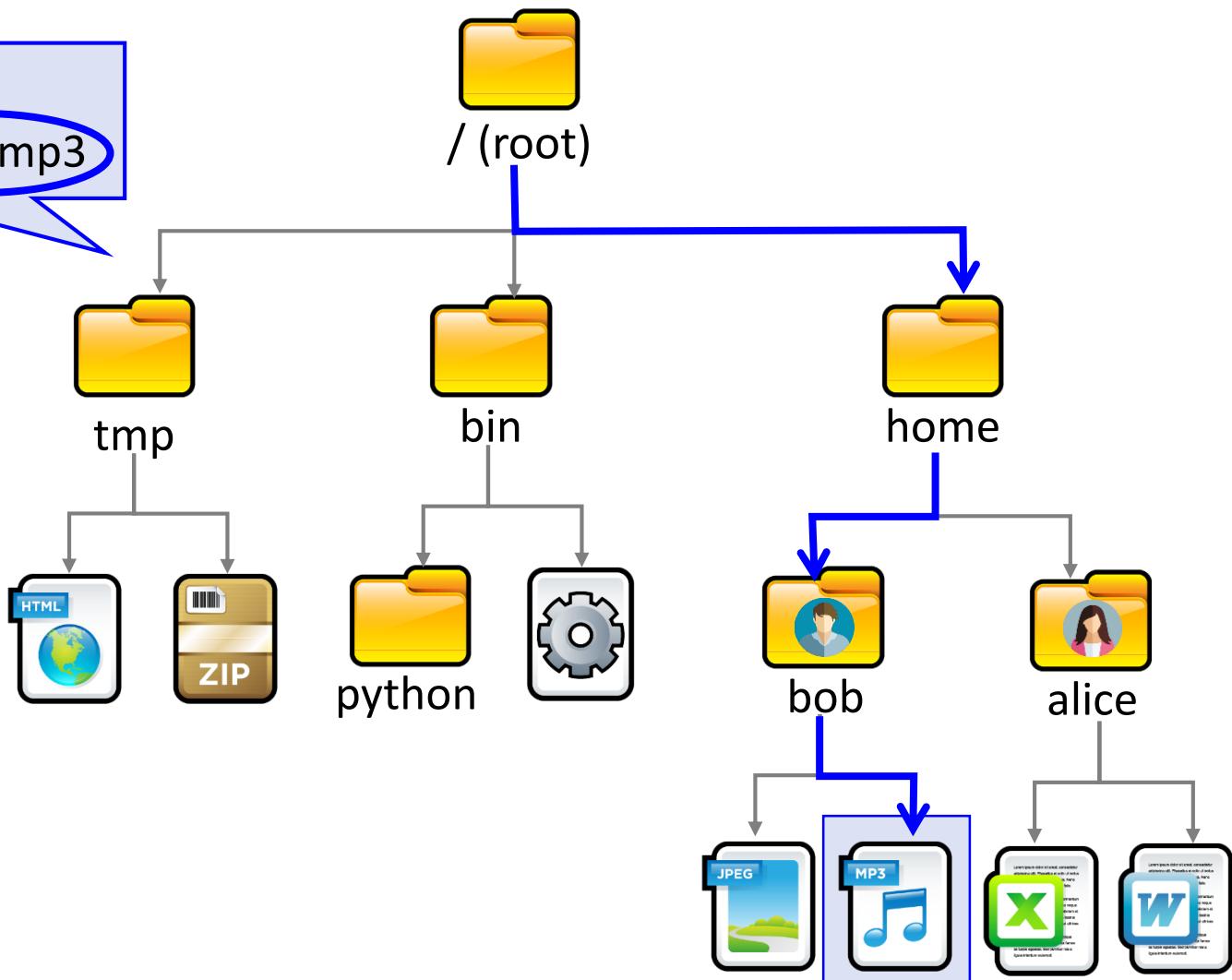


File Navigation



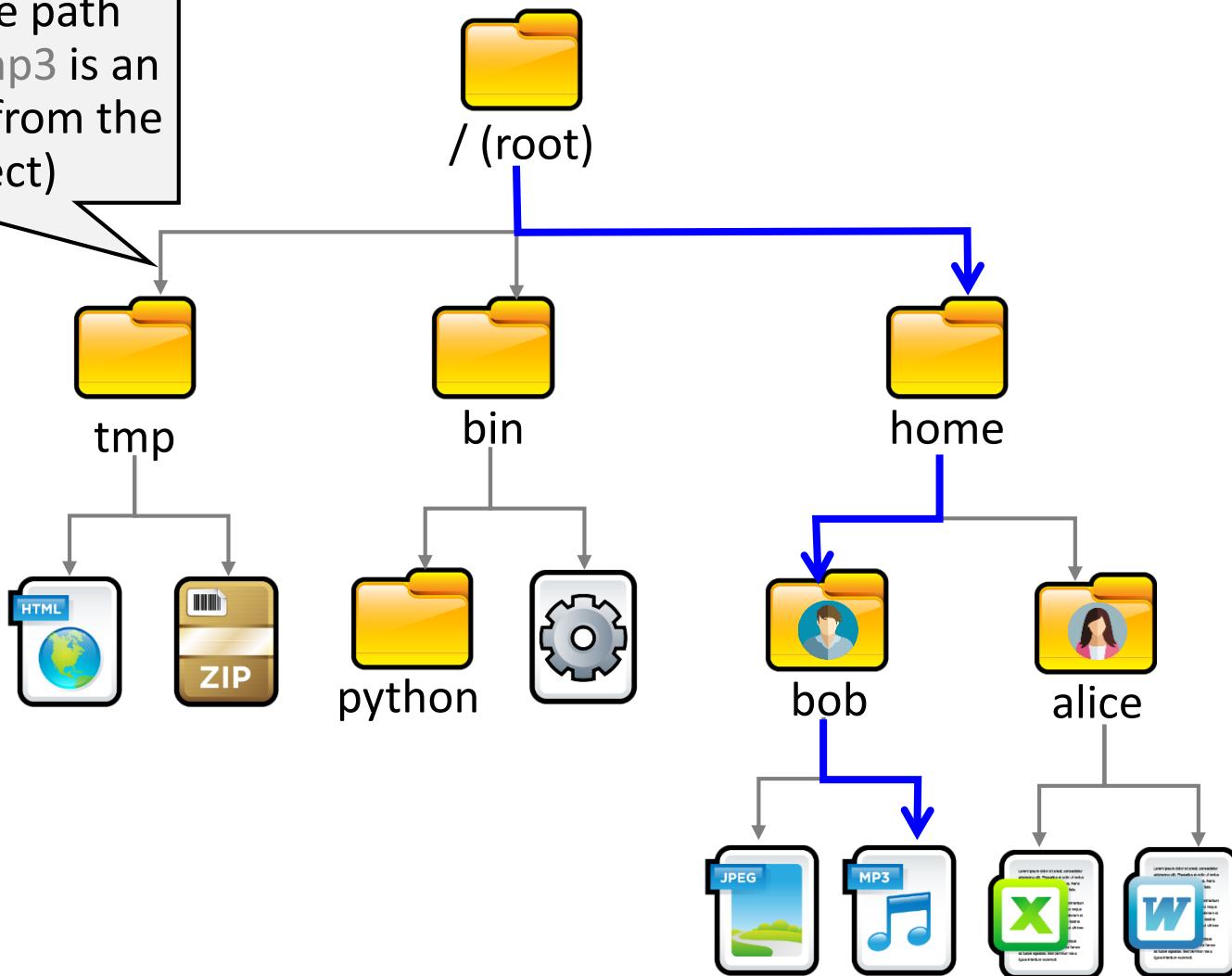
File Navigation

Using a path
/home/bob/music.mp3

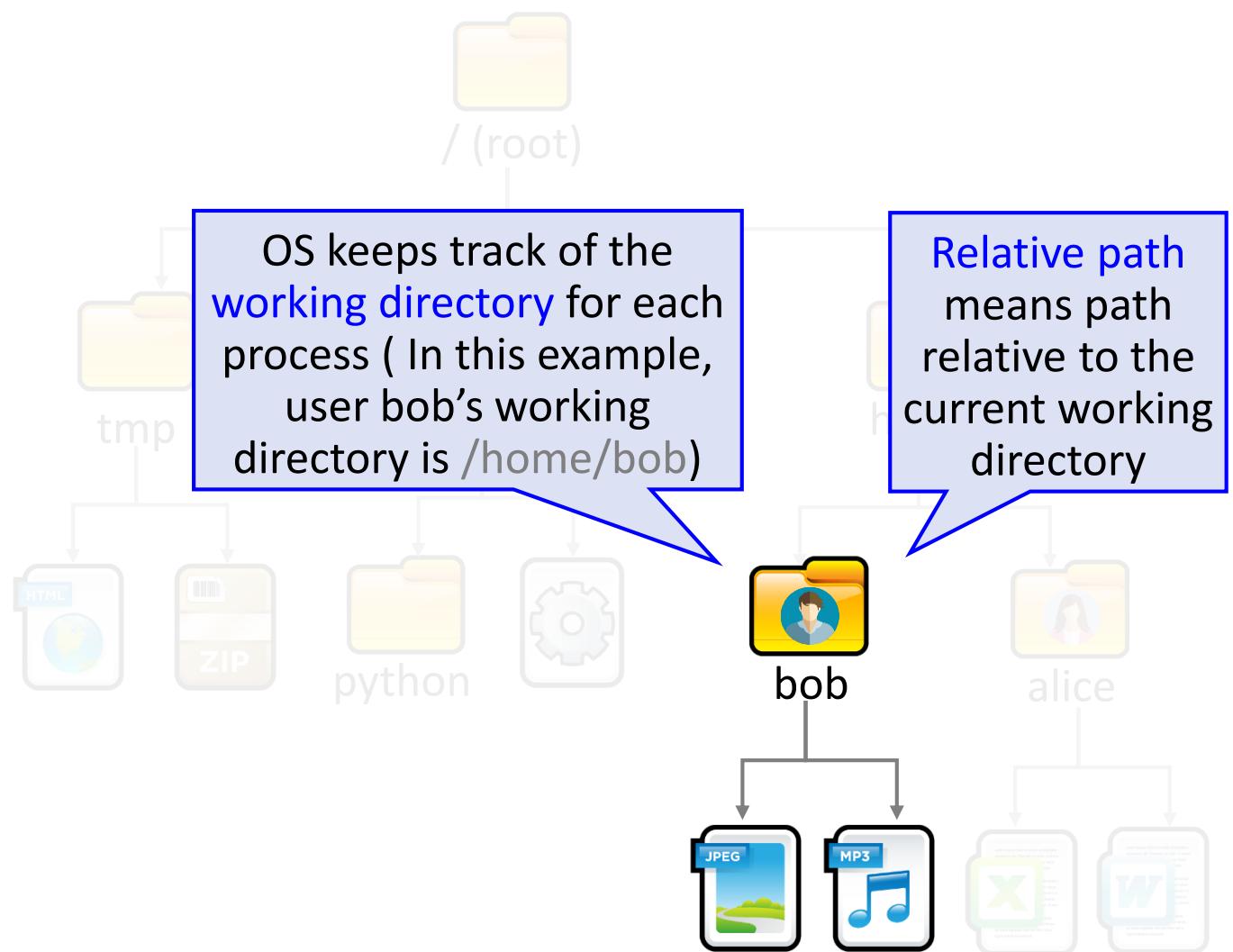


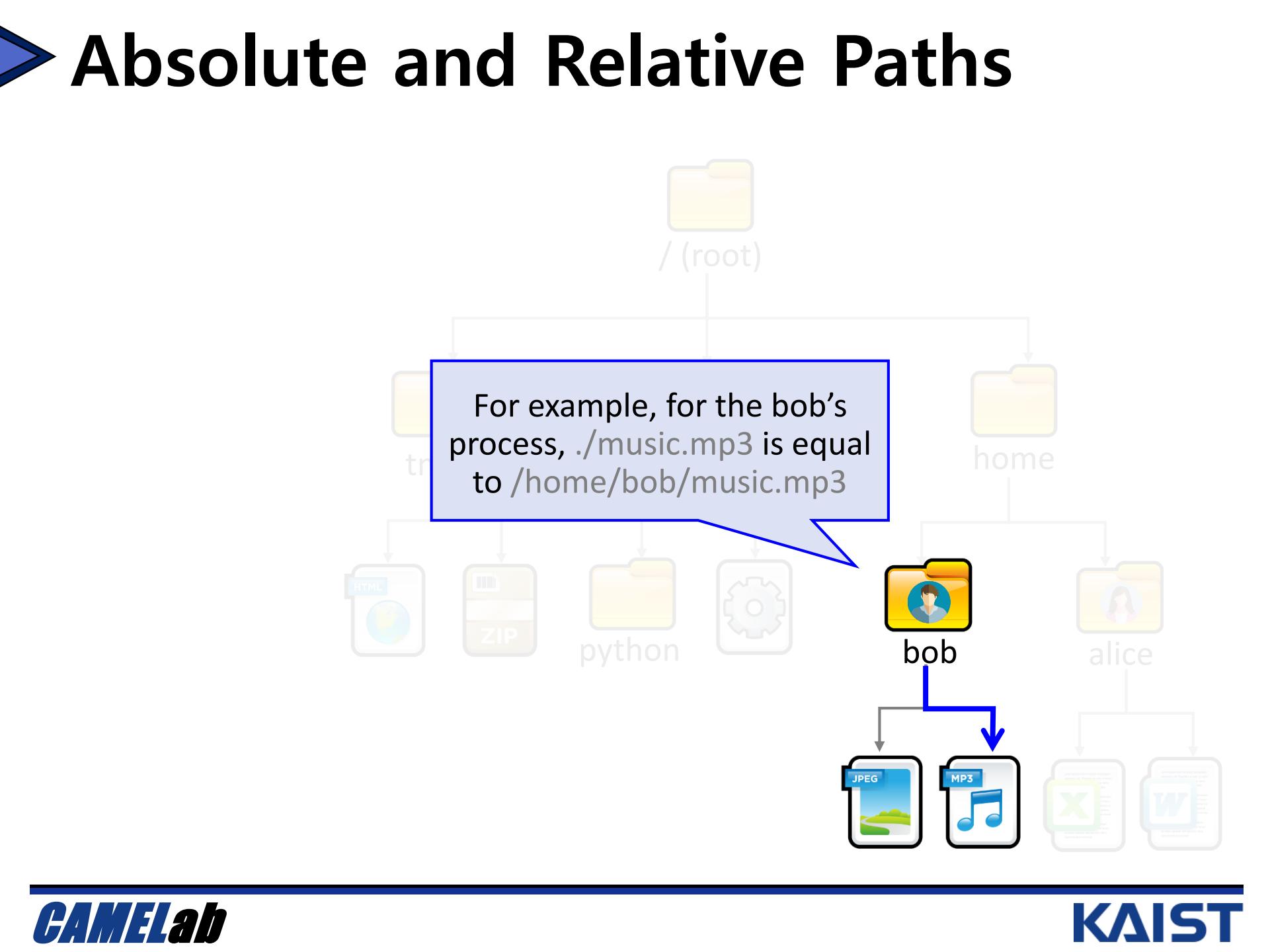
Absolute and Relative Paths

In this example, the path /home/bob/music.mp3 is an **absolute path** (path from the root to the object)



Absolute and Relative Paths





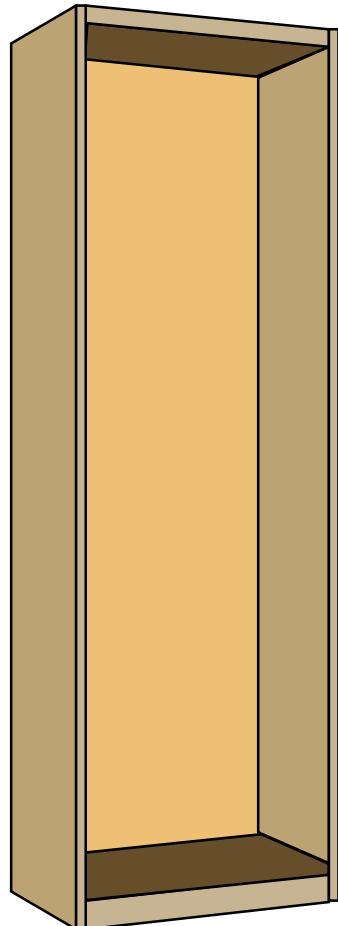
File System Initialization

Let's check how file system can be initialized with a simple analogy



[Analogy] File Cabinets

Step #1: Prepare
empty storage area



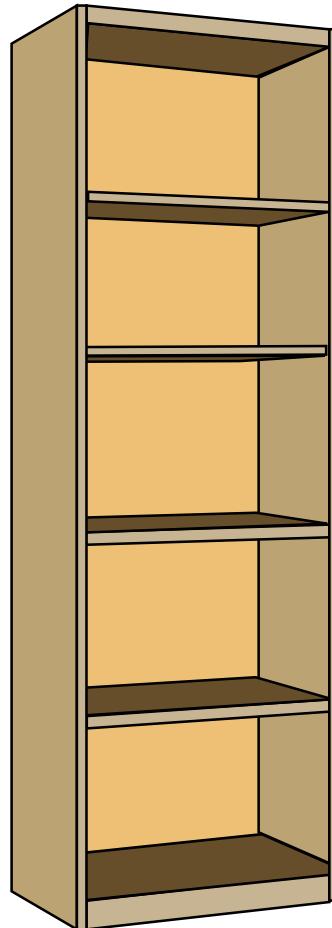
First of all, you need to
prepare an empty storage
area to store a bunch of files

[Analogy] File Cabinets

Step #1: Prepare
empty storage area



Step #2:
Partitioning



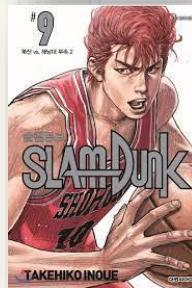
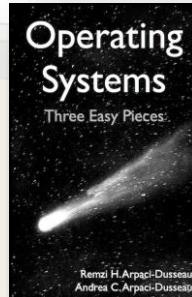
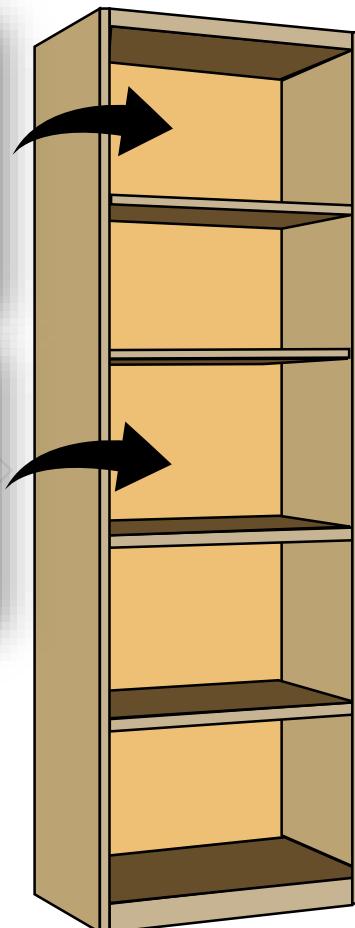
Divide then the storage area into
multiple partitions if you want to
separate files and manage them
for different purposes

[Analogy] File Cabinets

Step #1: Prepare
empty storage area



Step #2:
Partitioning



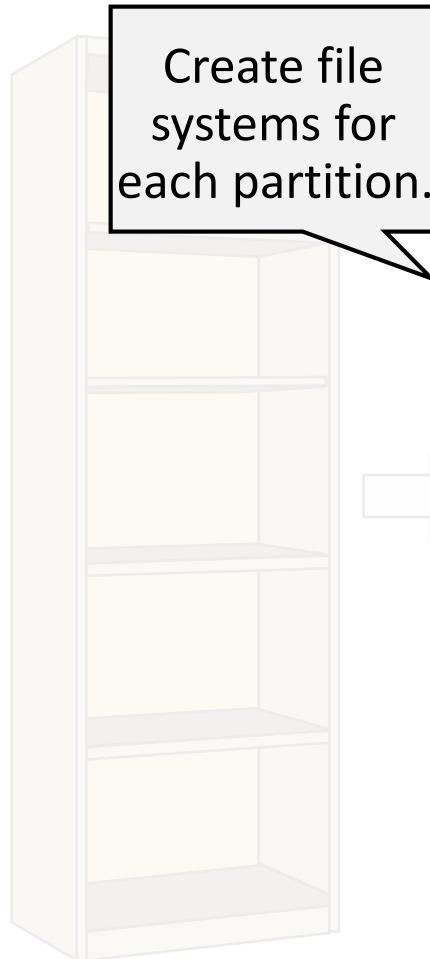
For example, you can
put OS textbooks and
comic books in
different spaces,
respectively 😊

[Analogy] File Cabinets

Step #1: Prepare
empty storage area

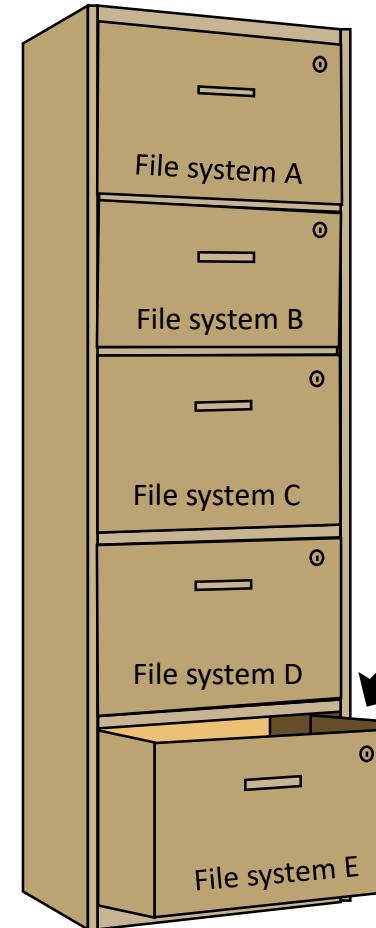


Step #2:
Partitioning



Create file
systems for
each partition.

Step #3: Build &
install FileSystem

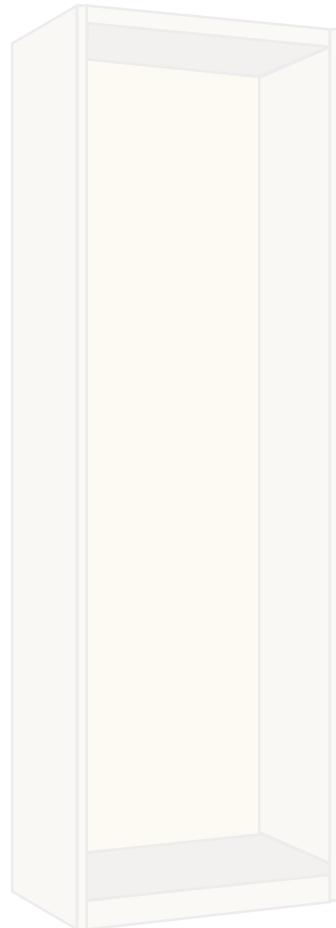


You can
categorize and
manage files
with different
file systems

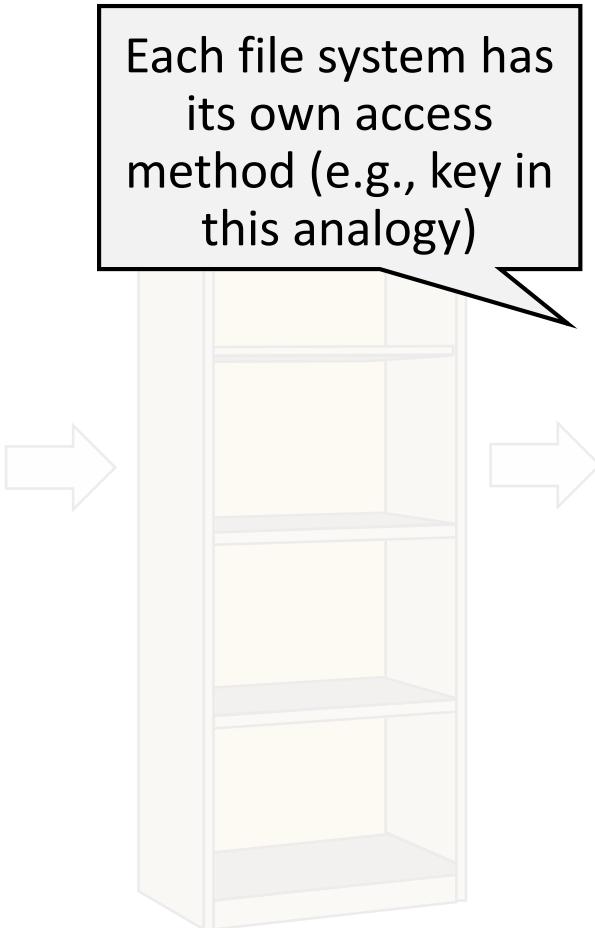


[Analogy] File Cabinets

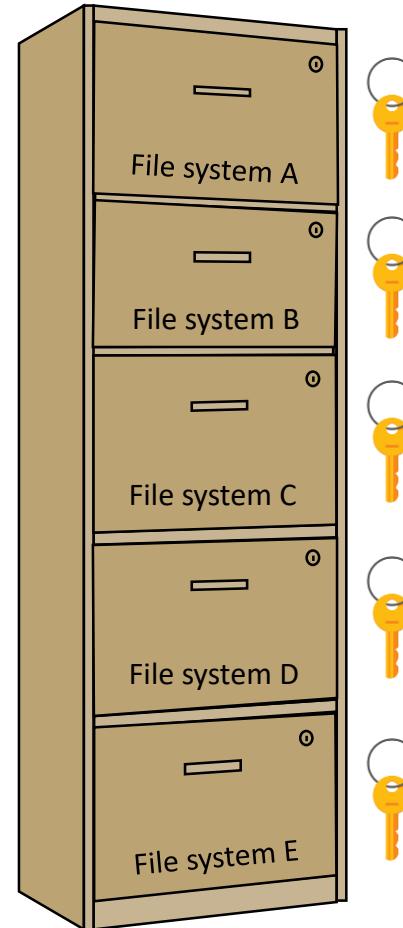
Step #1: Prepare
empty storage area



Step #2:
Partitioning



Step #3: Build &
install FileSystem

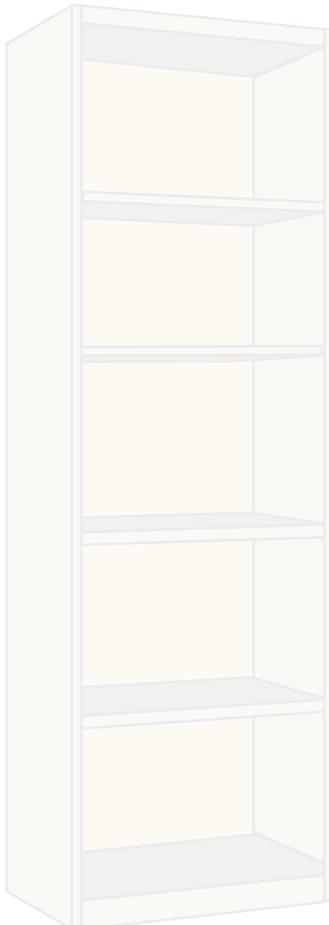


[Analogy] File Cabinets

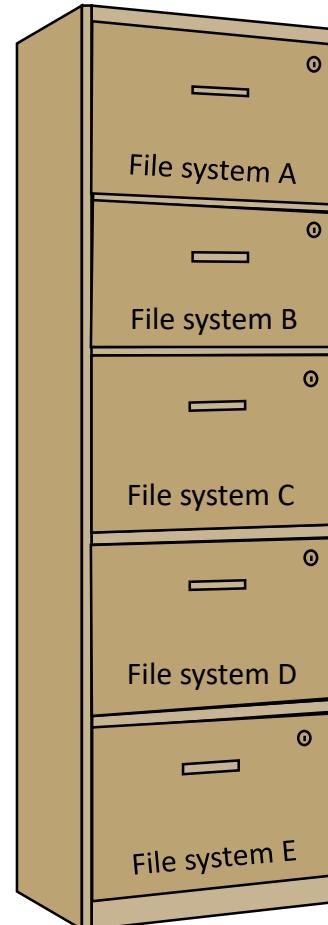
Step #1: Prepare
empty storage area



Step #2:
Partitioning



Step #3: Build &
install FileSystem



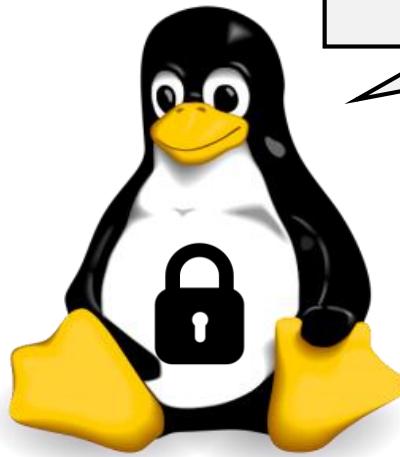
Step #4: Mount the Keys
in the desired place



Mounting* file
system; we can
place the keys on
the appropriate
place in a key
book

* In computer system, mount means
preparing something in an [available status](#)

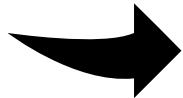
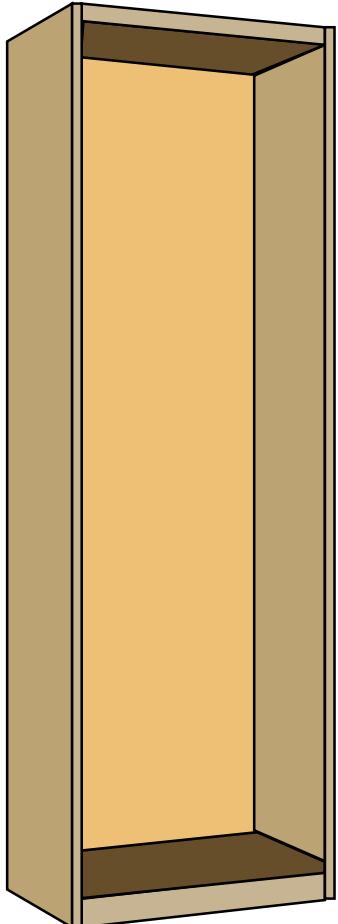
[OS] File System



Let's check how the initialization process of a real file system can be matched with what we showed before (analogy)

Step #1: Get a Storage Device

Step #1: Prepare
empty storage area





Storage Device Installation

Prepare and install
a pristine storage
device for your PC



Storage Device File

Userland

After the device installation,
OS kernel run an appropriate
device driver for the device
(e.g., SATA, SCSI, NVMe)

Kernel



Hardware

Block Device Driver



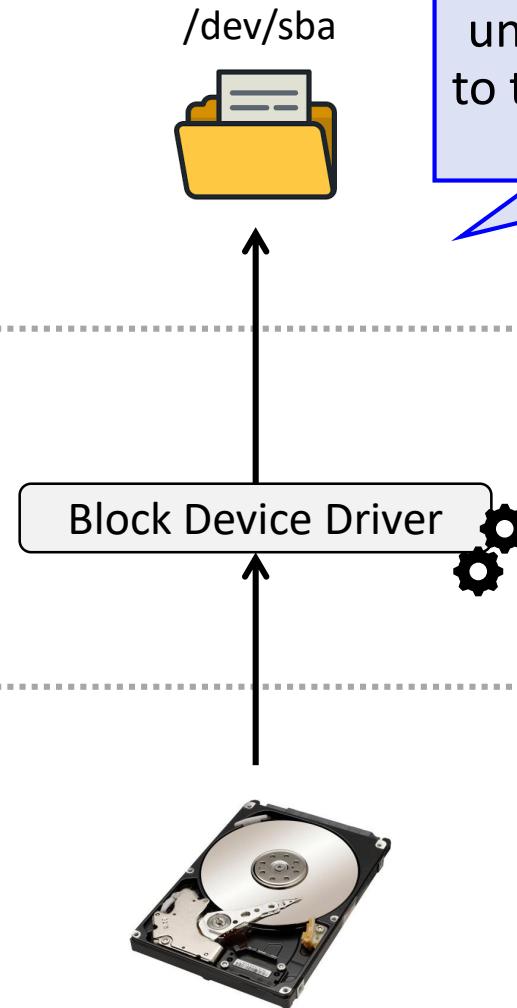
Storage Device File

Userland

Kernel



Hardware



The block device driver then exposes the underlying storage device to the userland as a special file (e.g., [device file](#))

Usable Command: lsblk

```
$ lsblk
```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE
sda	8:0	0	558.4G	0	disk
sdb	8:16	0	10T	0	disk
nvme0n1	259:0	0	372.6G	0	disk

lsblk lists information about all available block devices

We can know that there are two HDDs (sdX) and a SSD (nvmeX) in this system



/dev/sda
558.4 GB



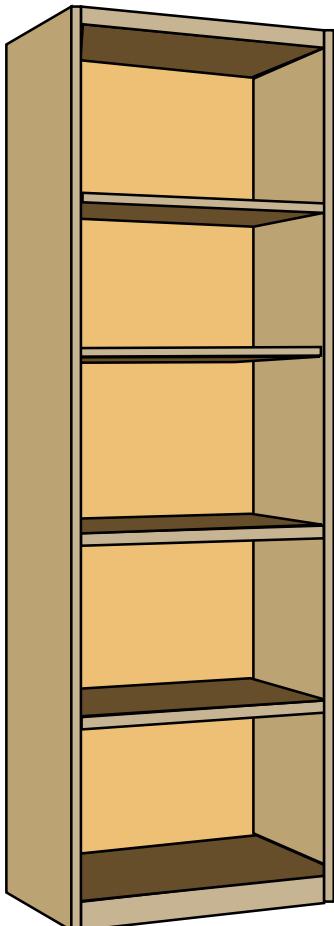
/dev/sdb
10 TB



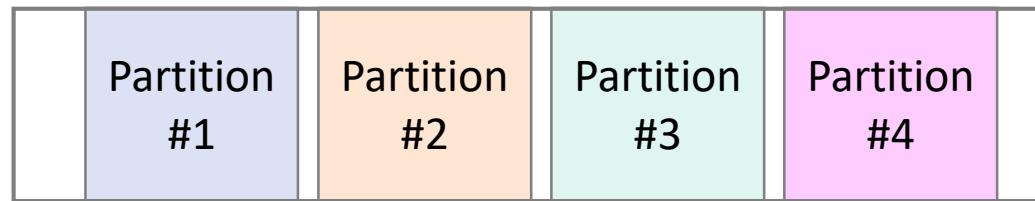
/dev/nvme0n1
372.6 GB

Step #2: Partition the Storage Device

Step #2: Partitioning



The disk can be segmented into one or more partitions, each occupying physically a part of the underlying disk storage area





Usable Command: fdisk



```
$ fdisk /dev/sdb
```

```
Command (m for help): m
```

```
Help:
```

```
Generic
```

- d delete a partition
- F list free unpartitioned space
- I list known partition types
- n add a new partition
- p print the partition table
- t change a partition type
- v verify the partition table
- i print information about a partition

```
Save & Exit
```

- w write table to disk and exit
- q quit without saving changes

You can create and manage the partition on a disk with fdisk command given with the path of device file



Usable Command: fdisk



```
$ fdisk /dev/sdb
```

```
Command (m for help): m
```

Help:

Generic

d delete a partition

F list free unpartitioned space

I list known partition types

n add a new partition

p print the partition table

t change a partition type

v verify the partition table

i print information about a partition

Save & Exit

w write table to disk and exit

q quit without saving changes

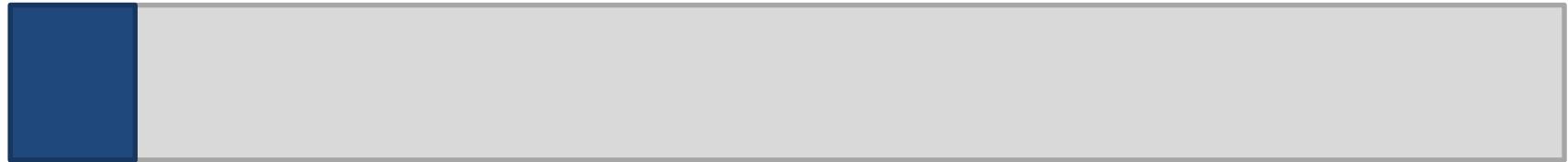
BTW, how can the file system know where a partition is in a disk and manage it addition/deletion)?

Partition Table

The partition table is a table located at a known place in the disk



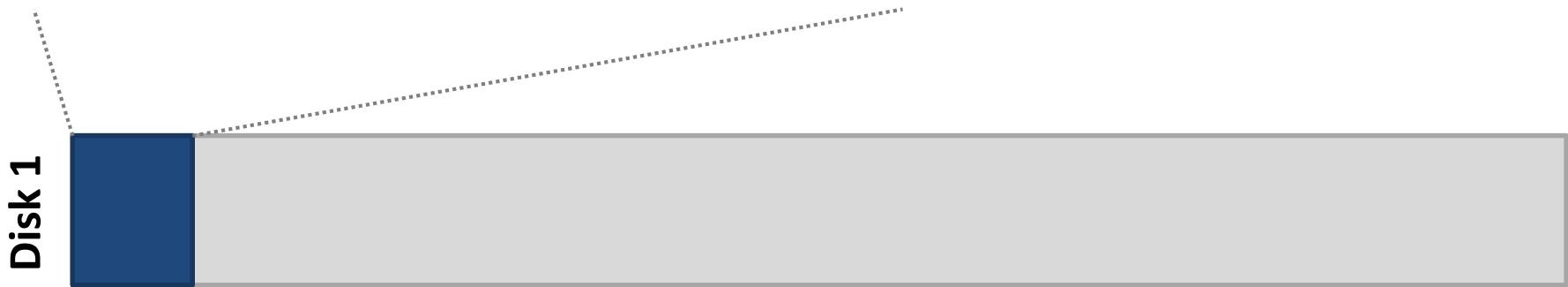
Disk 1



Partition Table

Address		Description	Size (Bytes)
Hex	Dec.		

The partition table includes the starting LBA and length of the partition



Partition Table

Address		Description	Size (Bytes)
Hex	Dec.		



There are multiple formats for the partition table, and the most popular name for the table is **Master Boot Record (MBR)**

Support disks up to **2TB**
Supports up to **4 partitions**



Partition Table

Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB



When a user sends a disk partition command, the kernel fills/retrieves the partition table information residing on the target disk

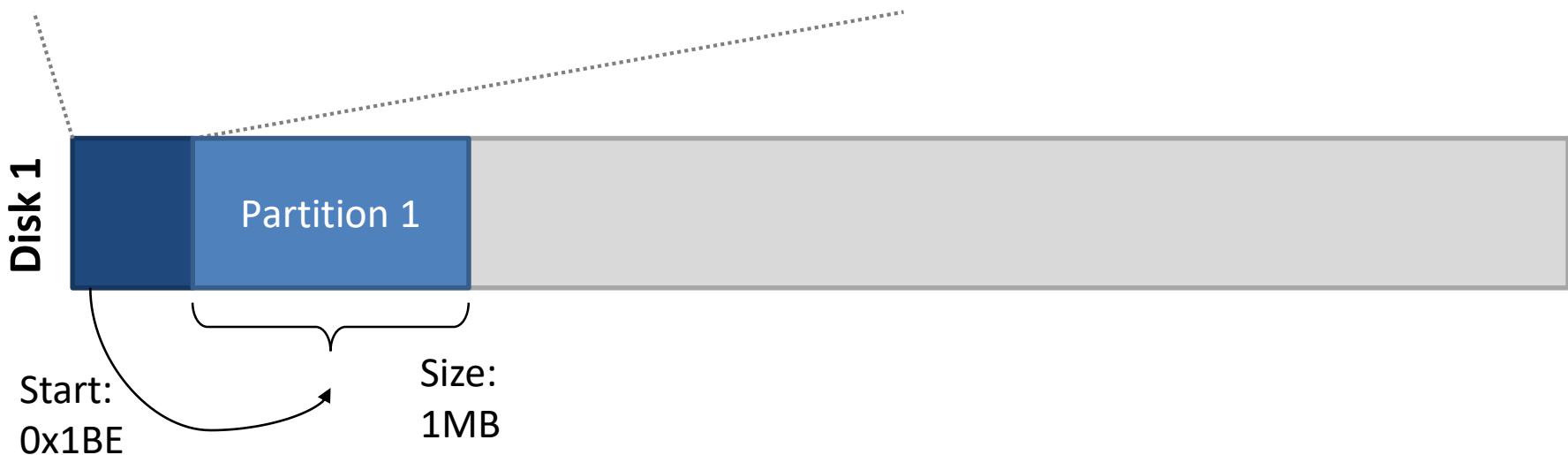


Partition Table

Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB



By referring the MBR (partition table), systems know where the target partition is and access it.

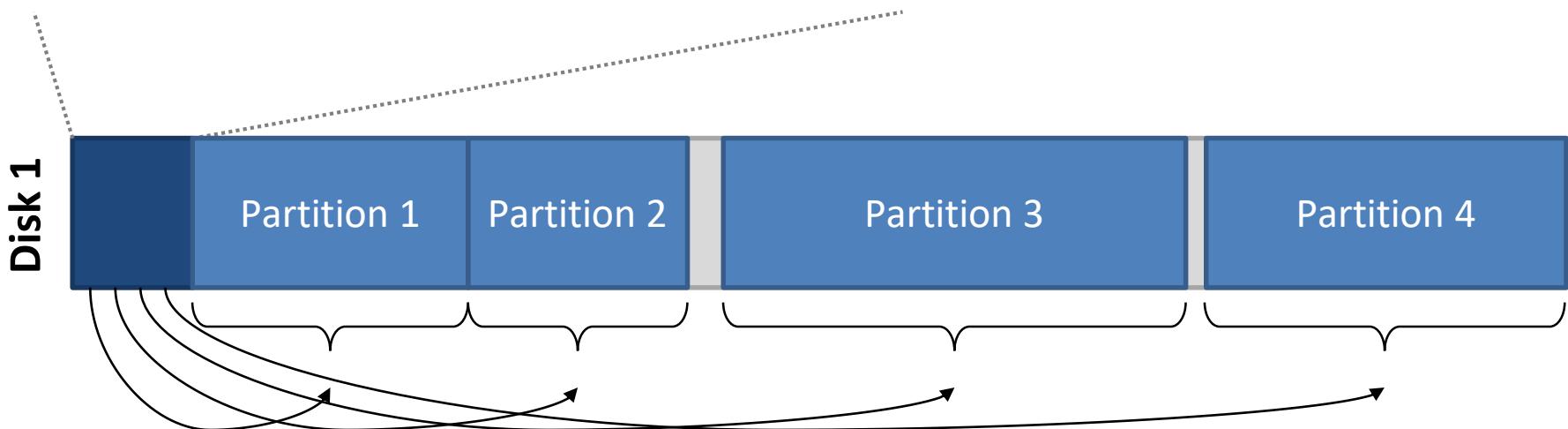


Partition Table

Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB
0x1CE	462	Partition Entry #2	500KB
0x1DE	478	Partition Entry #3	2MB
0x1EE	494	Partition Entry #4	1.5MB



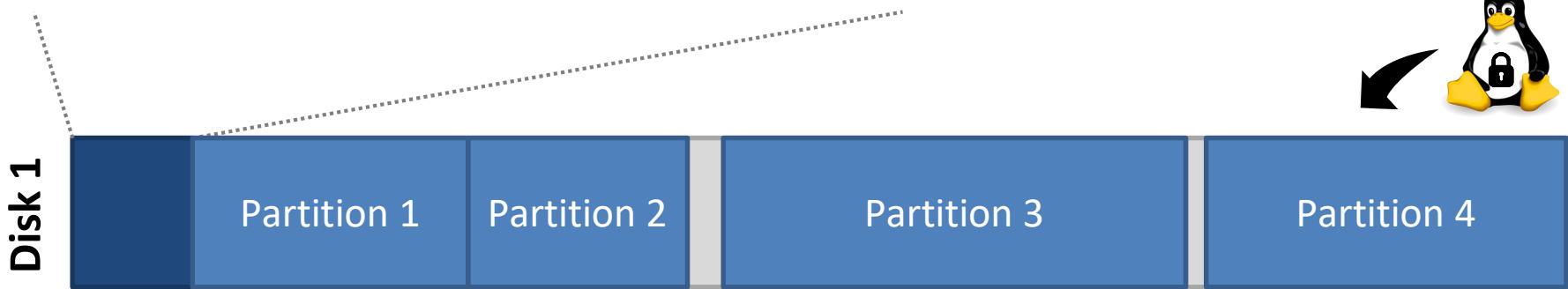
MBR can manage as many as 4 partitions



Partition Table

Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB
0x1CE	462	Partition Entry #2	500KB
0x1DE	478	Partition Entry #3	2MB
0x1EE	494	Partition Entry #4	1.5MB

These four partitions are called **primary partitions**, which can be set to be bootable (can install a bootable OS on it)



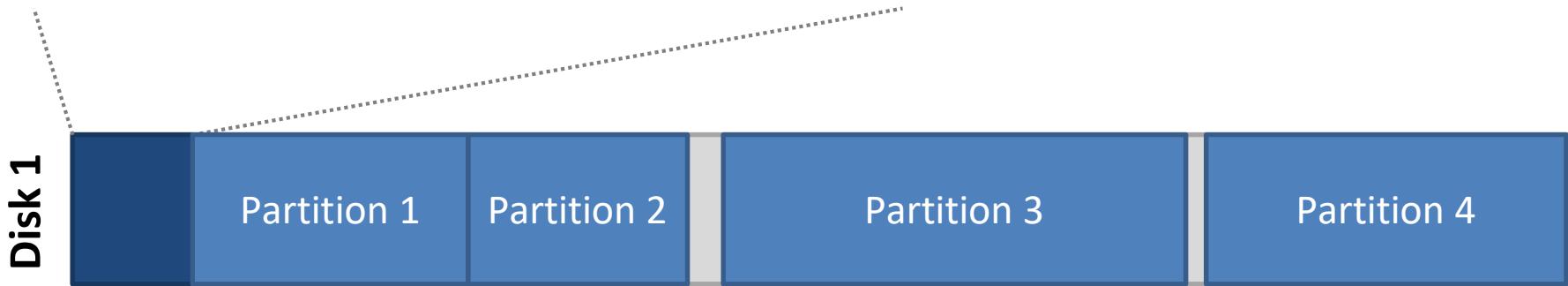


Challenge: Limited Partitions

Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB
0x1CE	462	Partition Entry #2	500KB
0x1DE	478	Partition Entry #3	2MB
0x1EE	494	Partition Entry #4	1.5MB



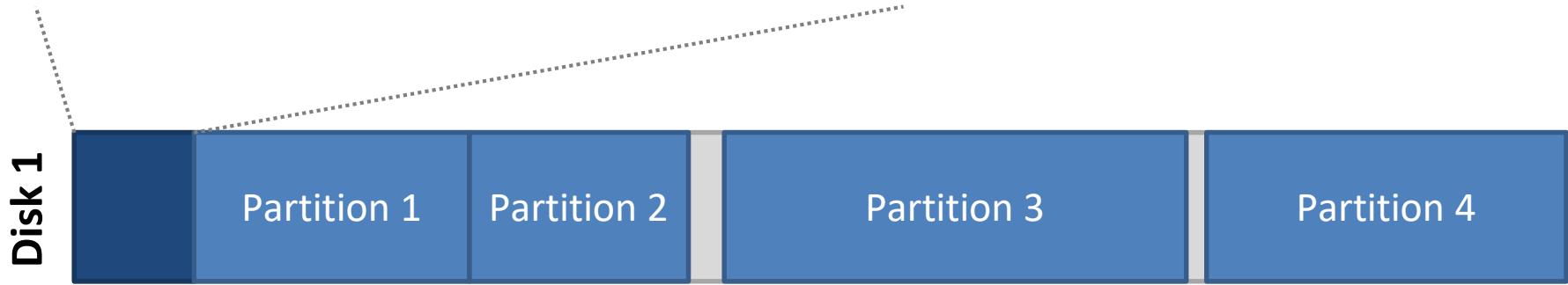
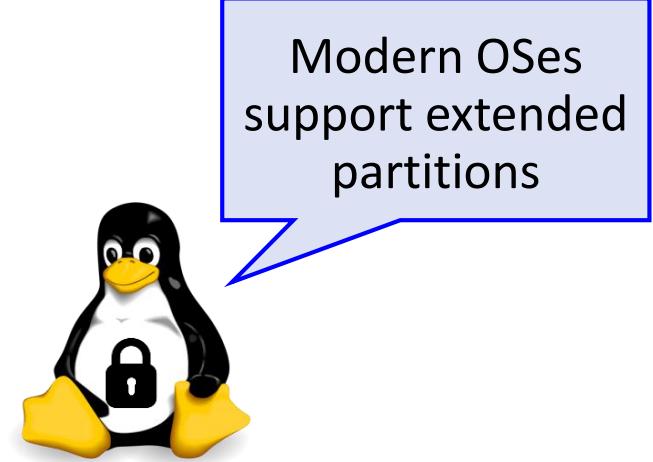
What about the case where you want more than 4 partitions?





Solution: Extended Partitions

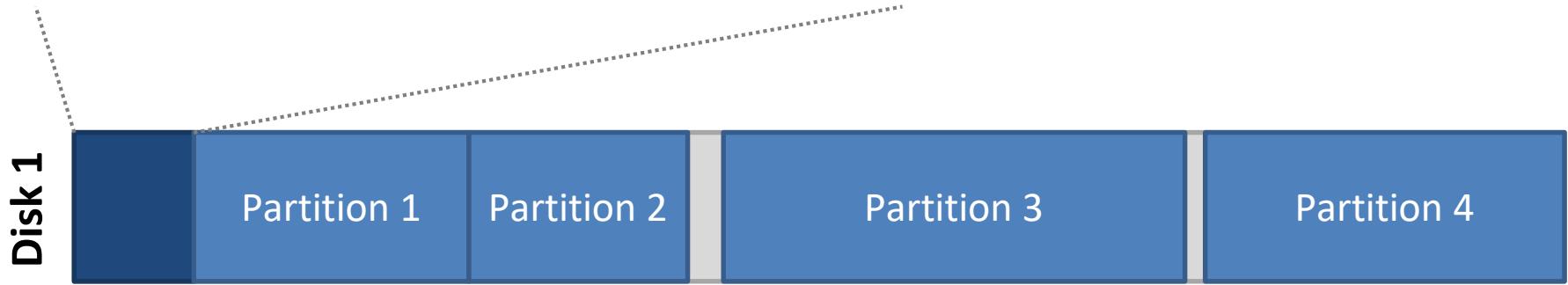
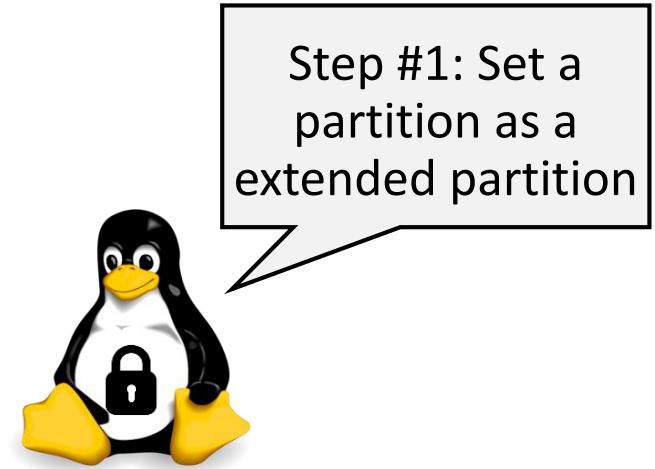
Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB
0x1CE	462	Partition Entry #2	500KB
0x1DE	478	Partition Entry #3	2MB
0x1EE	494	Partition Entry #4	1.5MB





Solution: Extended Partitions

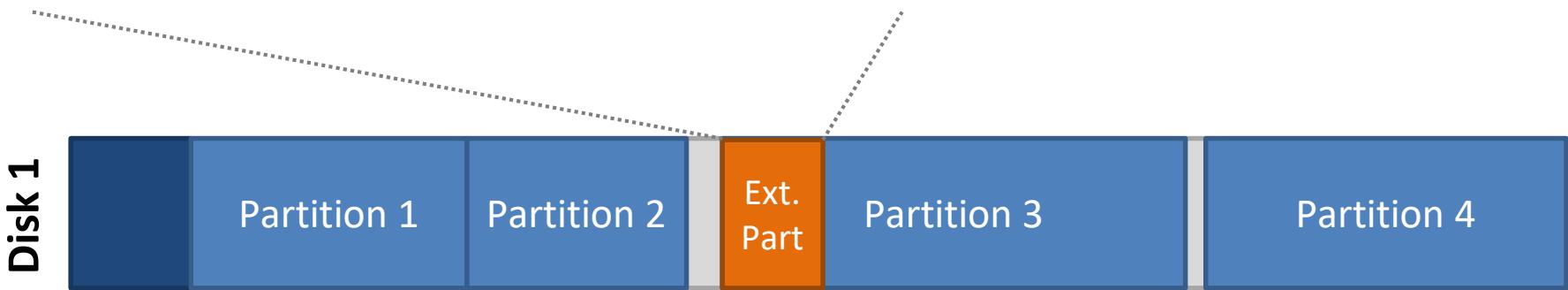
Address		Description	Size (Bytes)
Hex	Dec.		
0x1BE	446	Partition Entry #1	1MB
0x1CE	462	Partition Entry #2	500KB
0x1DE	478	Extended Partition	2MB
0x1EE	494	Partition Entry #4	1.5MB





Solution: Extended Partitions

Address		Description	Size (Bytes)
Hex	Dec.		



Step #2: Creates an extended partition table in the target extended partition (partition #3)

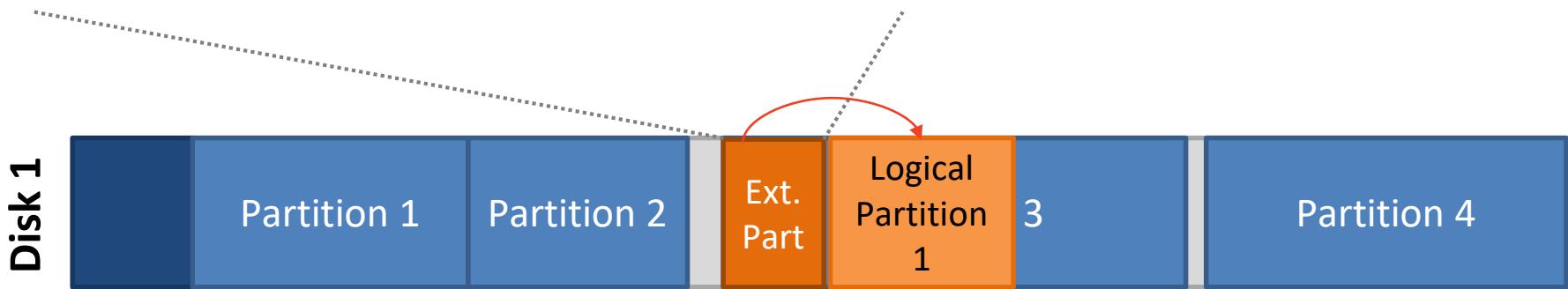


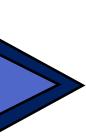


Solution: Extended Partitions

Address		Description	Size (Bytes)
Hex	Dec.		
0x1DF	479	Logical Partition #1	500KB

Step #3: Now fill the extended partition table for the information of logical partitions

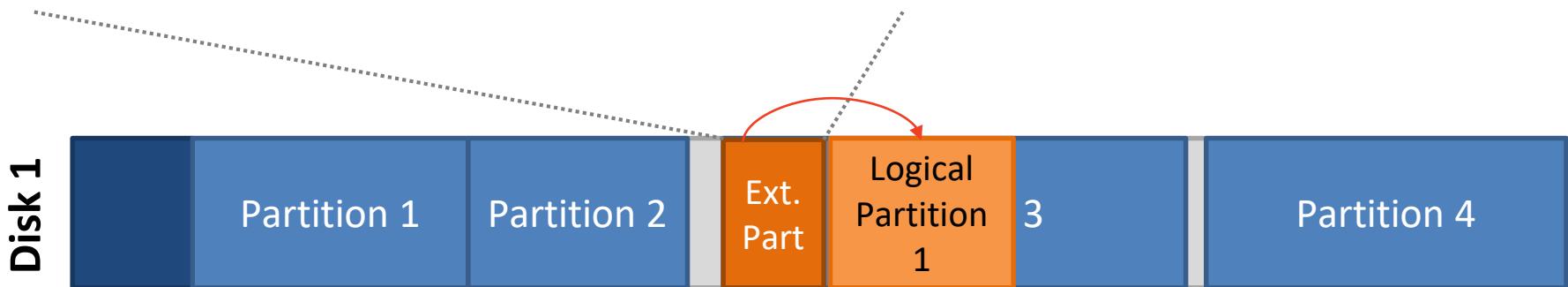




Solution: Extended Partitions

Address		Description	Size (Bytes)
Hex	Dec.		
0x1DF	479	Logical Partition #1	500KB

Note that, as the extended partition table follows OS-specific formats, other OSes cannot read the logical partitions



Partition Device Files

Userland

Kernel

Hardware



/dev/sba



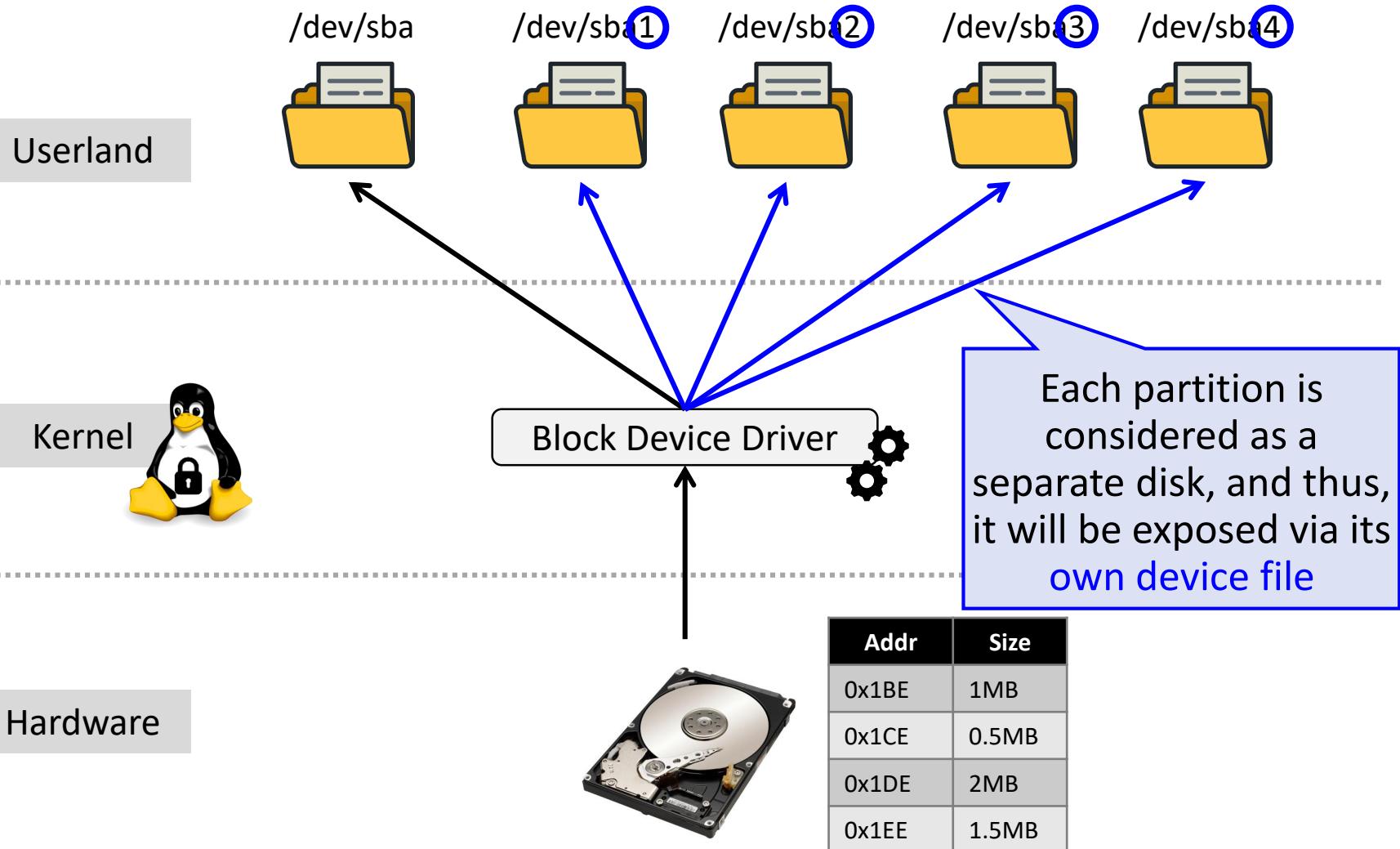
Block Device Driver



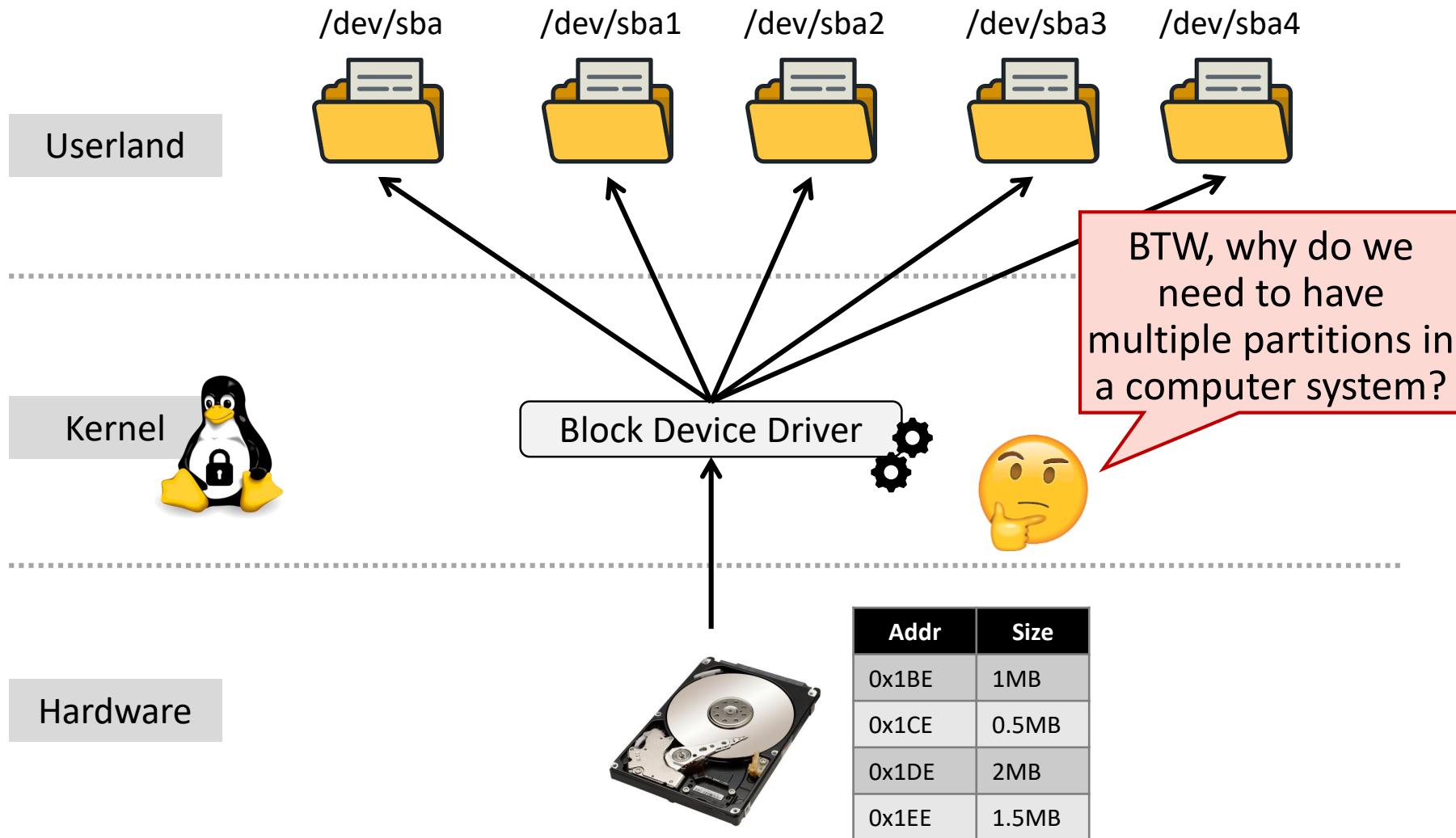
Addr	Size
0x1BE	1MB
0x1CE	0.5MB
0x1DE	2MB
0x1EE	1.5MB

Partition information
is managed by MBR
residing on a disk

Partition Device Files



Partition Device Files





Needs of Multiple Partitions



Partitions enable us to assign distinct characteristics (types) for each different storage area

Disk 1





Needs of Multiple Partitions

Case #1: Some partitions will be **read only**, others will be **writable**



Disk 1





Needs of Multiple Partitions

Case #2: Some partitions will **compress** data and their save space more, while others will have the data **without compression**



Disk 1





Needs of Multiple Partitions



Case #3: Some partitions will manage their data with **encryptions**, while others will have their files w/ **clear texts**

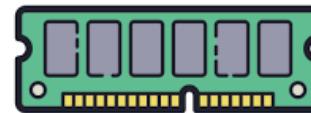
Disk 1





Needs of Multiple Partitions

Case #4: Some partitions will only handle **swap data** (part of memory), whereas others will have **normal files**



Disk 1





Needs of Multiple Partitions

This partition is referred as **swap partition**, and the size of swap partition is double amount of physical memory in general

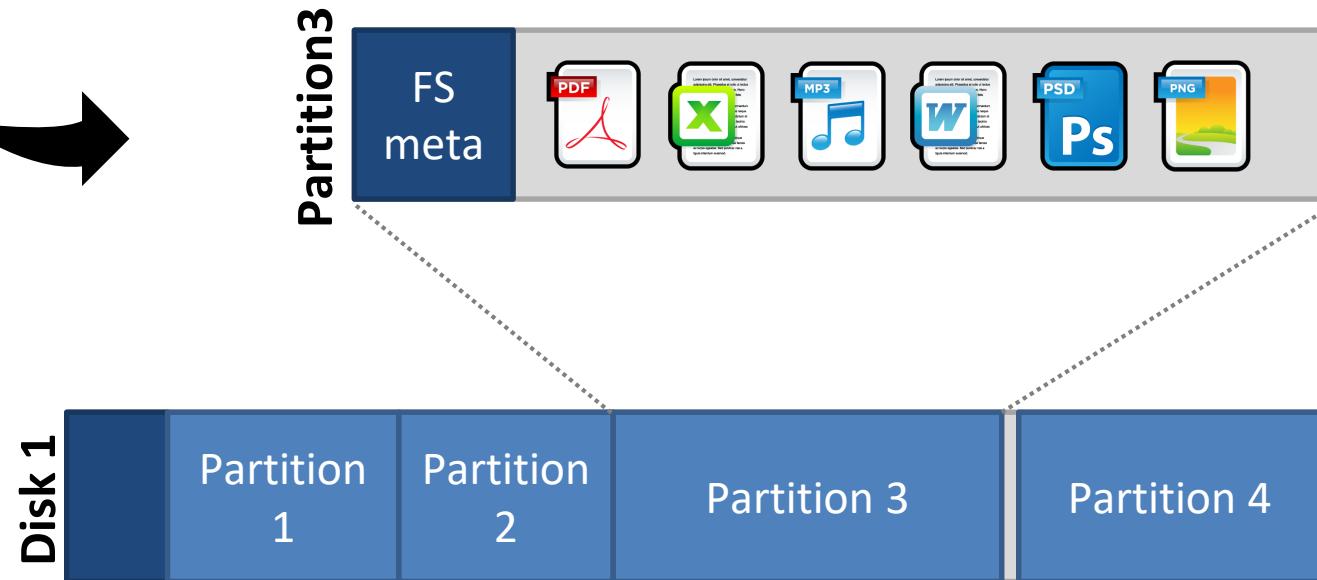
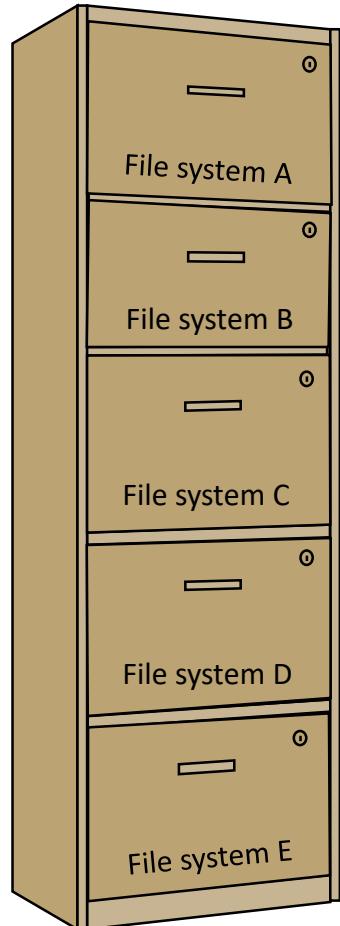


Disk 1



Step #3: Build & Install a FileSystem

Step #3: Build & install FileSystem



There should be a per-partition FS metadata to access all the files therein (metadata will be covered in the next lecture)



Usable Command: mkfs



```
$ mkfs
```

```
mkfs
```

```
mkfs.bfs
```

```
mkfs.cramfs
```

```
mkfs.ext2
```

```
mkfs.ext3
```

```
mkfs.ext4
```

```
mkfs.fat
```

```
mkfs.minix
```

```
mkfs.msdos
```

```
mkfs.ntfs
```

```
mkfs.vfat
```

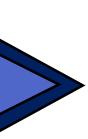
Filesystem can be built with
`mkfs` command; note that
there are many options to
pick your own filesystem

Usable Command: mkfs

```
$ mkfs
mkfs          mkfs.ext2        mkfs.fat        mkfs.ntfs
mkfs.bfs      mkfs.ext3        mkfs.minix     mkfs.vfat
mkfs.cramfs   mkfs.ext4        mkfs.msdos

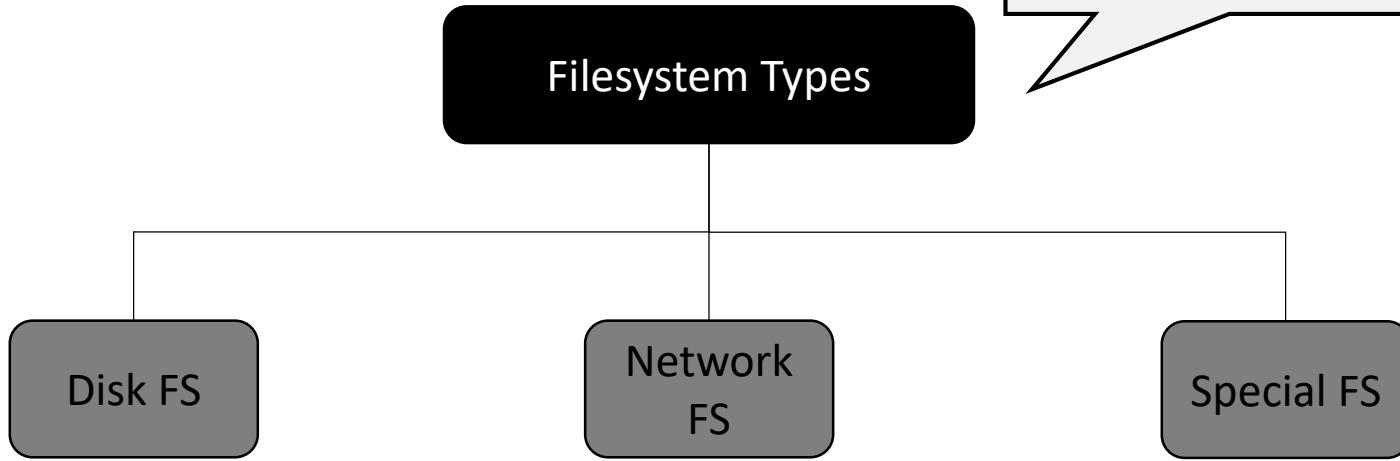
$ mkfs.ext4 /dev/sdb1
mke2fs
fs_types for mke2fs.conf resolution: 'ext4'
OS type: Linux
Block size = 4096
Fragment size = 4096
...
Writing superblocks and filesystem accounting information: done
```

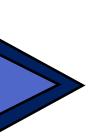
In this example, we build ext4 filesystem for the first partition of sdb



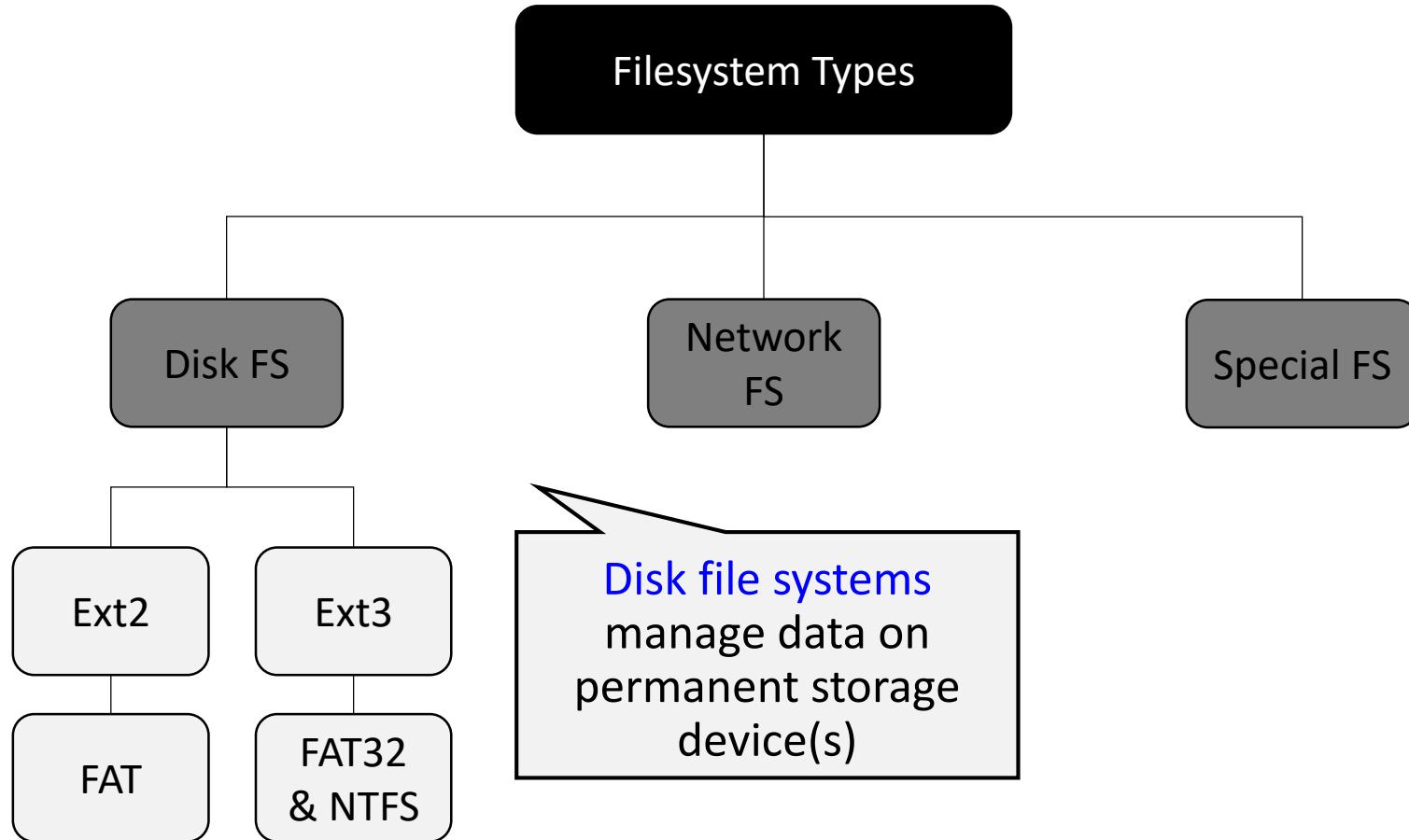
FileSystem Types

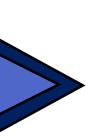
File system types can be largely classified into three groups; i) disk, ii) network, and iii) special-purpose FS



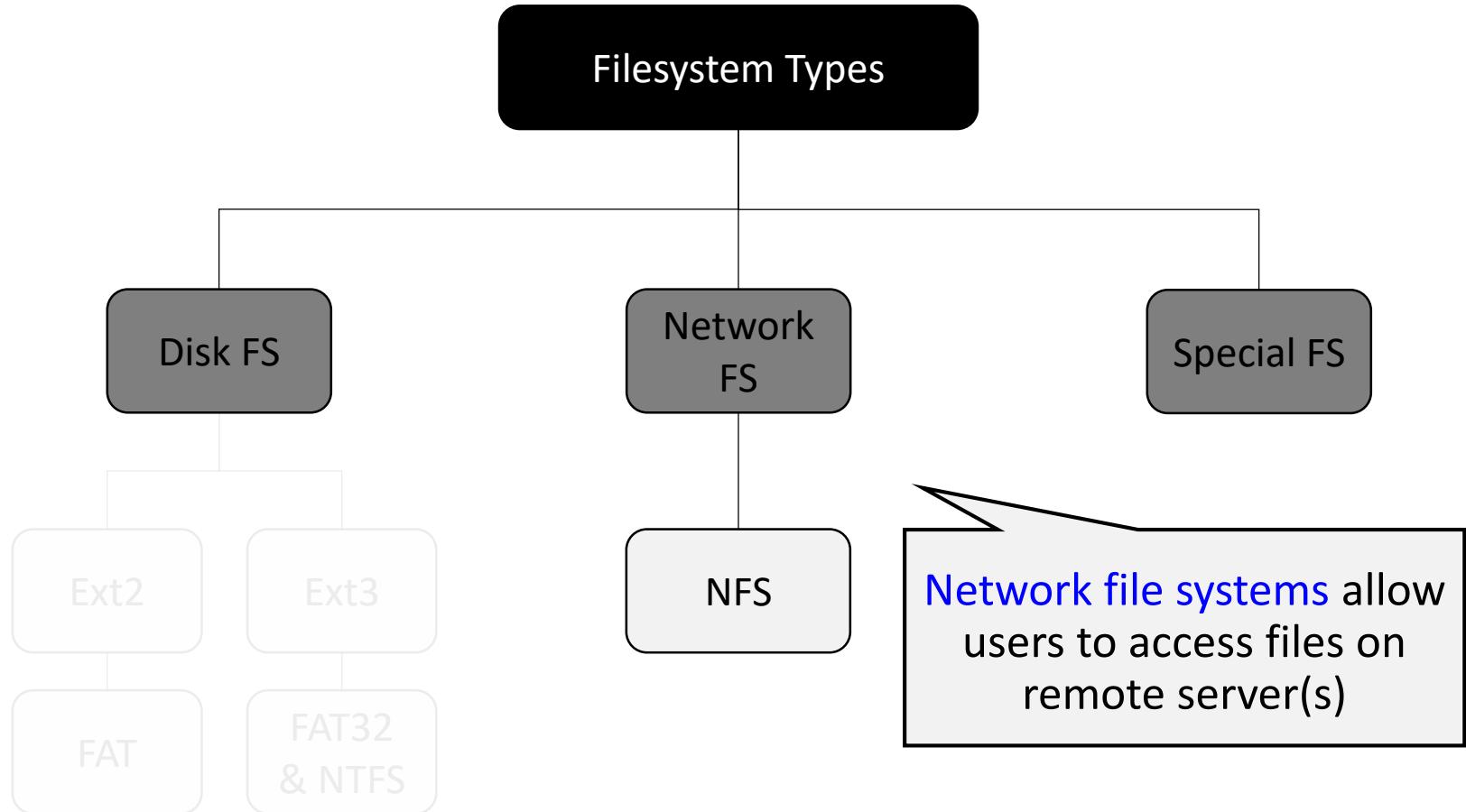


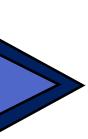
FileSystem Types



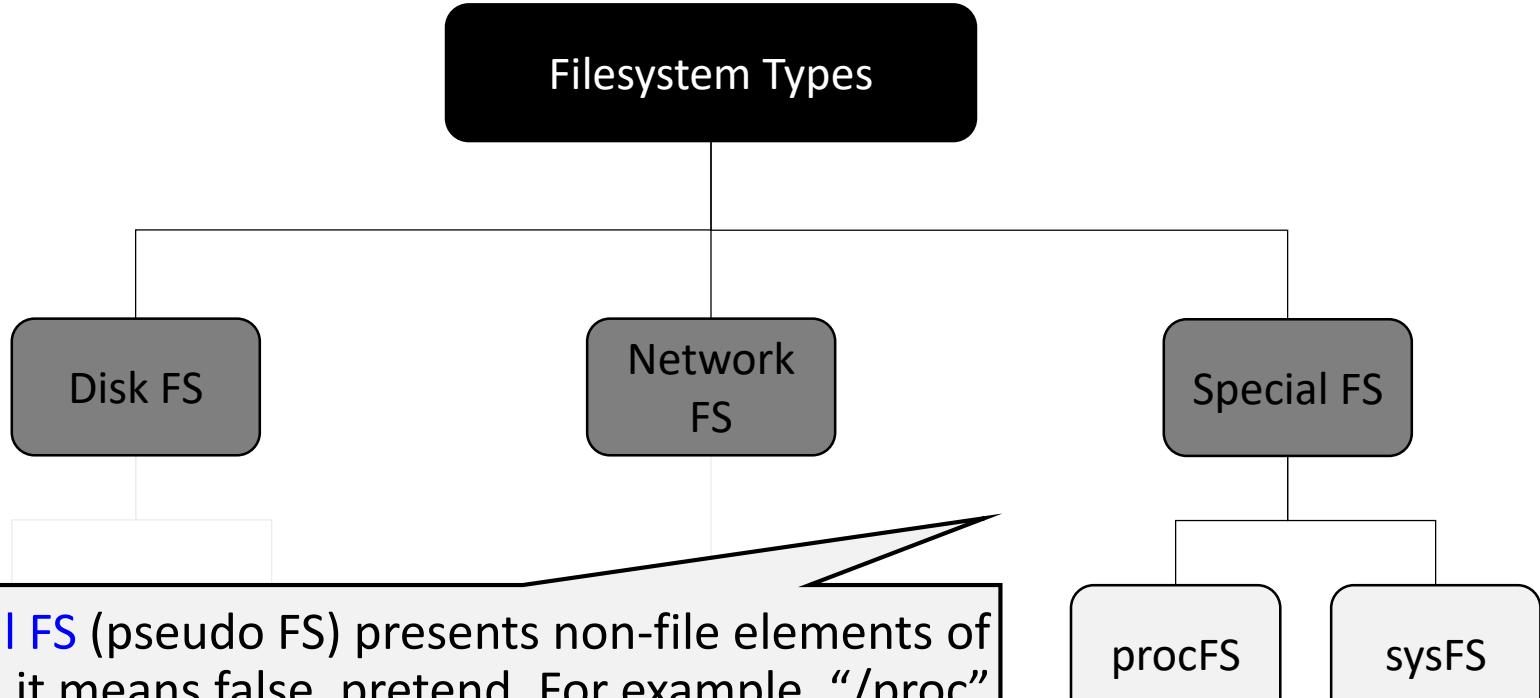


FileSystem Types





FileSystem Types



Special FS (pseudo FS) presents non-file elements of an OS; it means false, pretend. For example, “/proc” on many Oses is a “procfs” that dynamically generates directories for every process. “/sys” on Linux generates files to represent as well.

Multiple FileSystem Types in System

Linux kernel
provides all kinds of
filesystem driver

Kernel



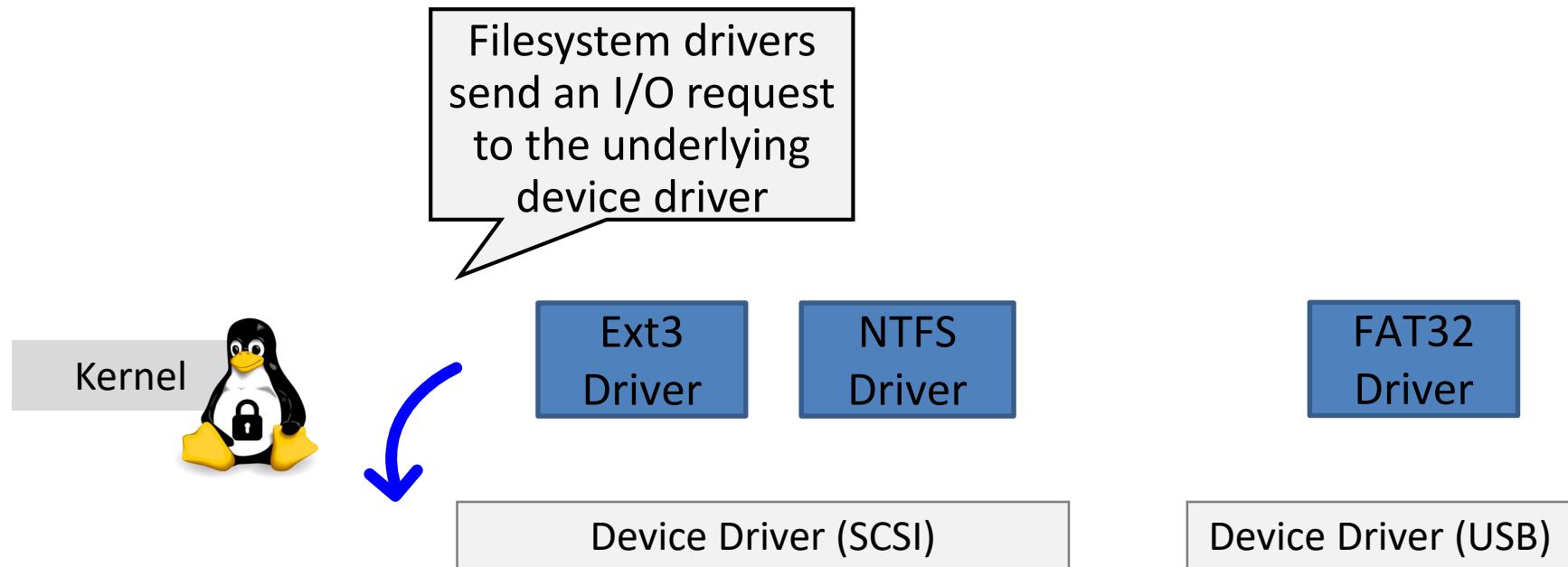
Ext3
Driver

NTFS
Driver

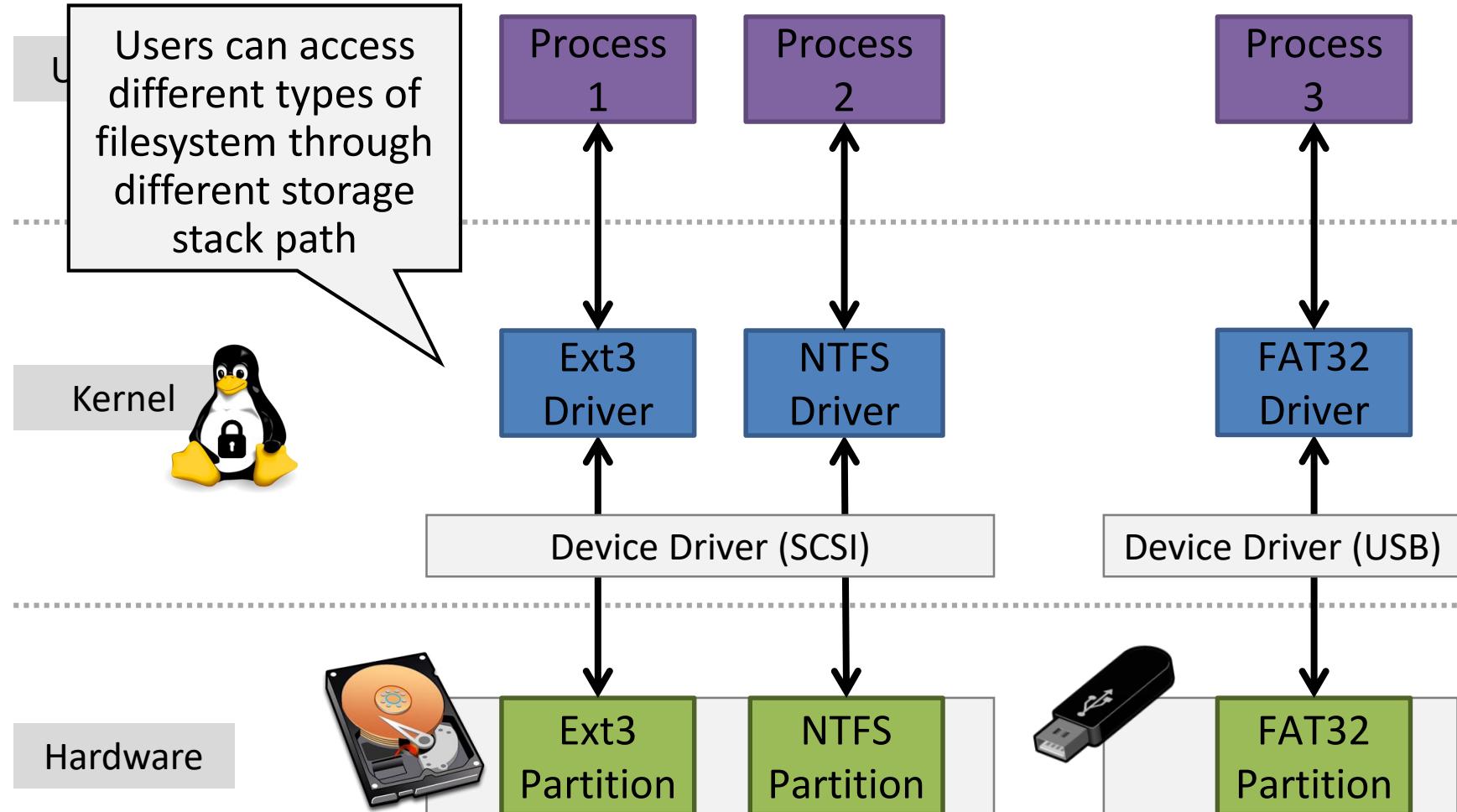
FAT32
Driver



Multiple FileSystem Types in System

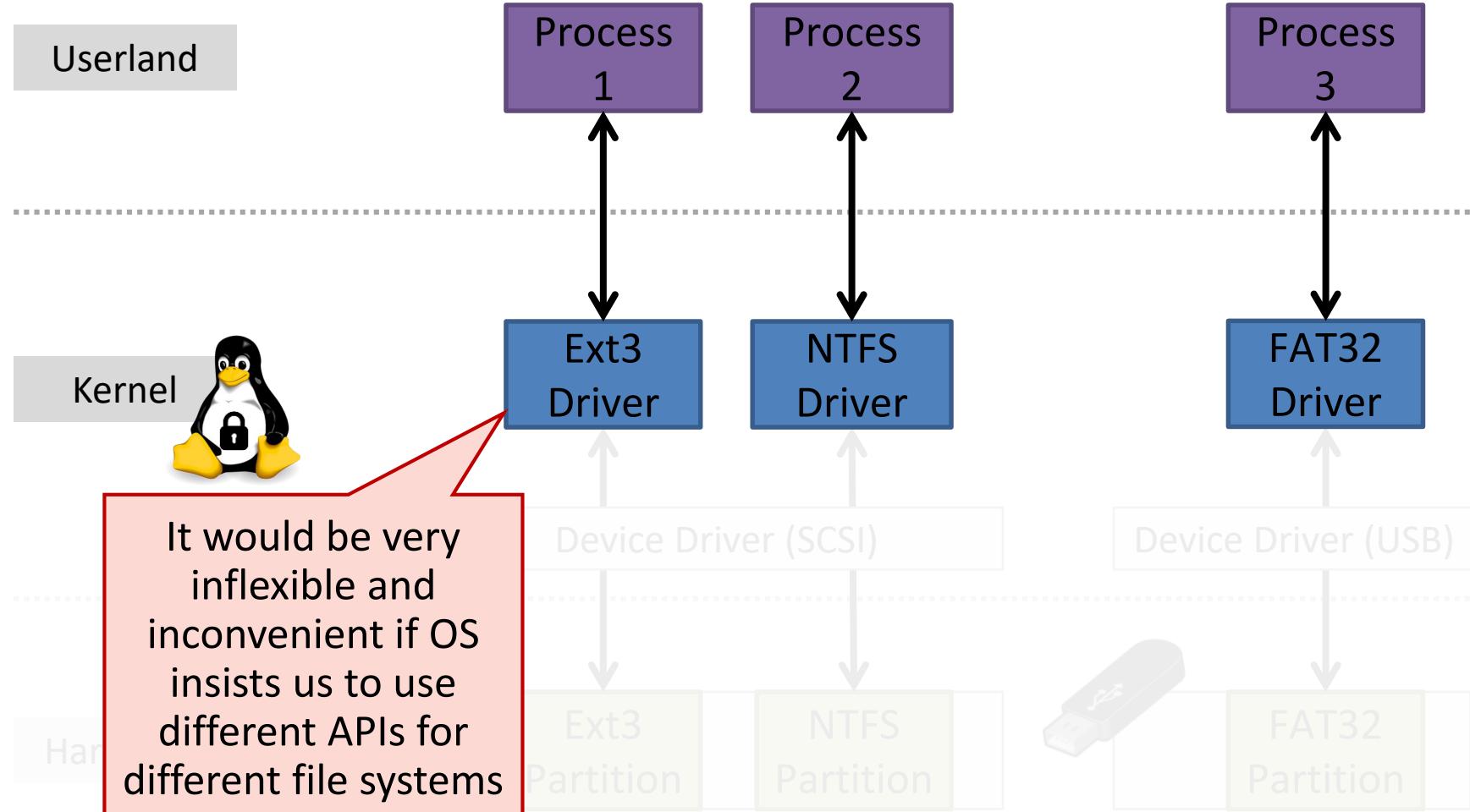


Multiple FileSystem Types in System

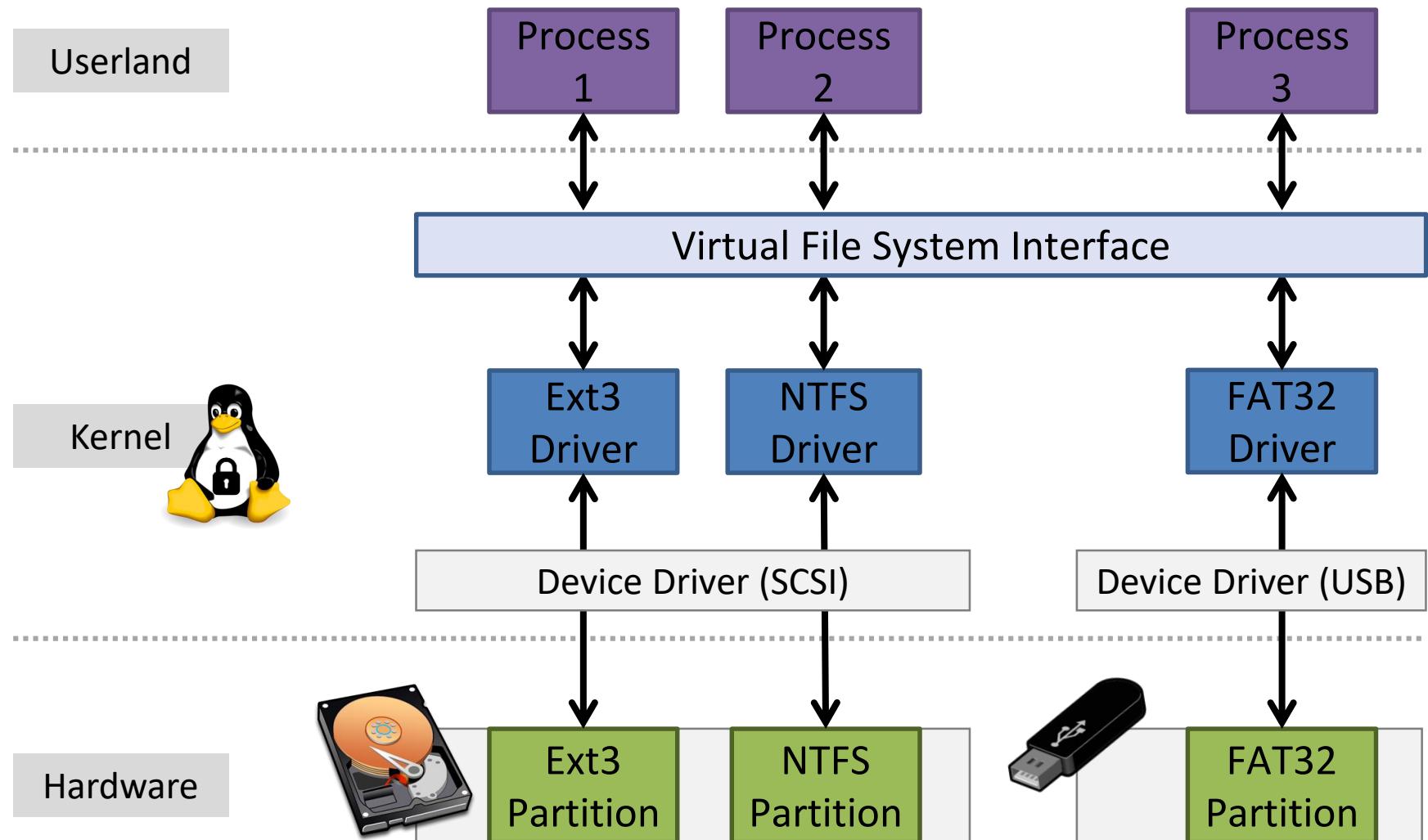




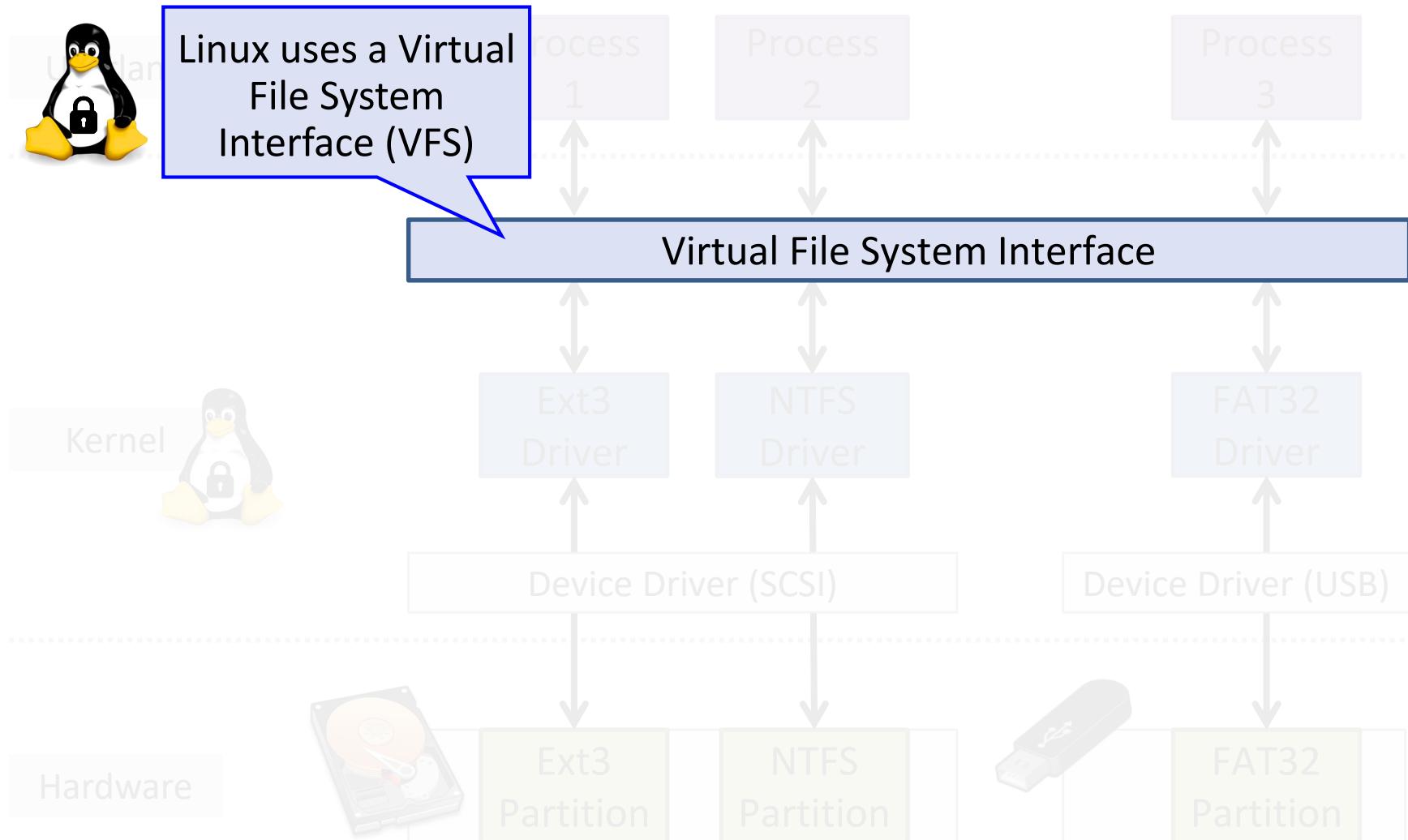
Problem: Burden to User



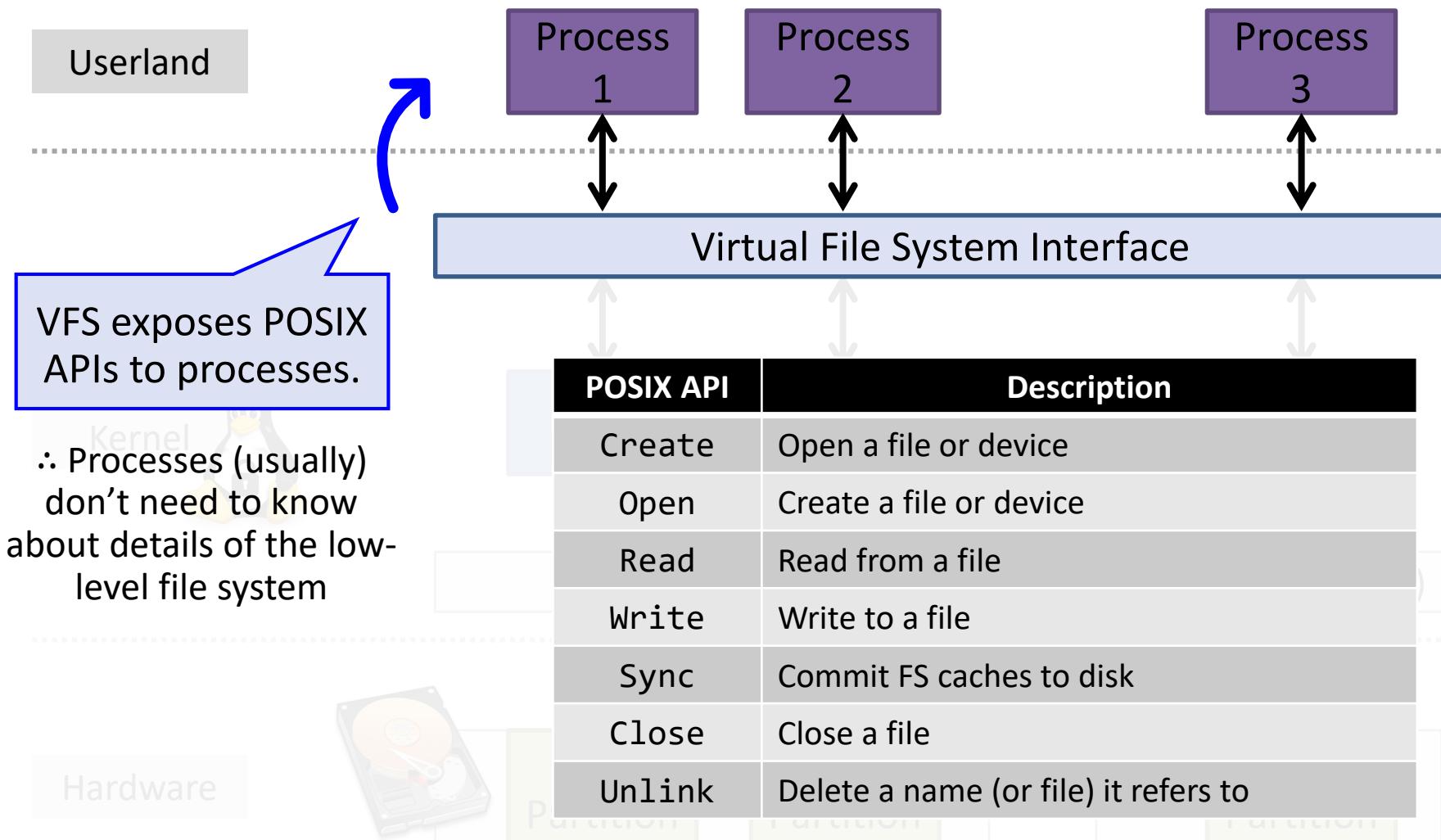
Solution: Virtual File System



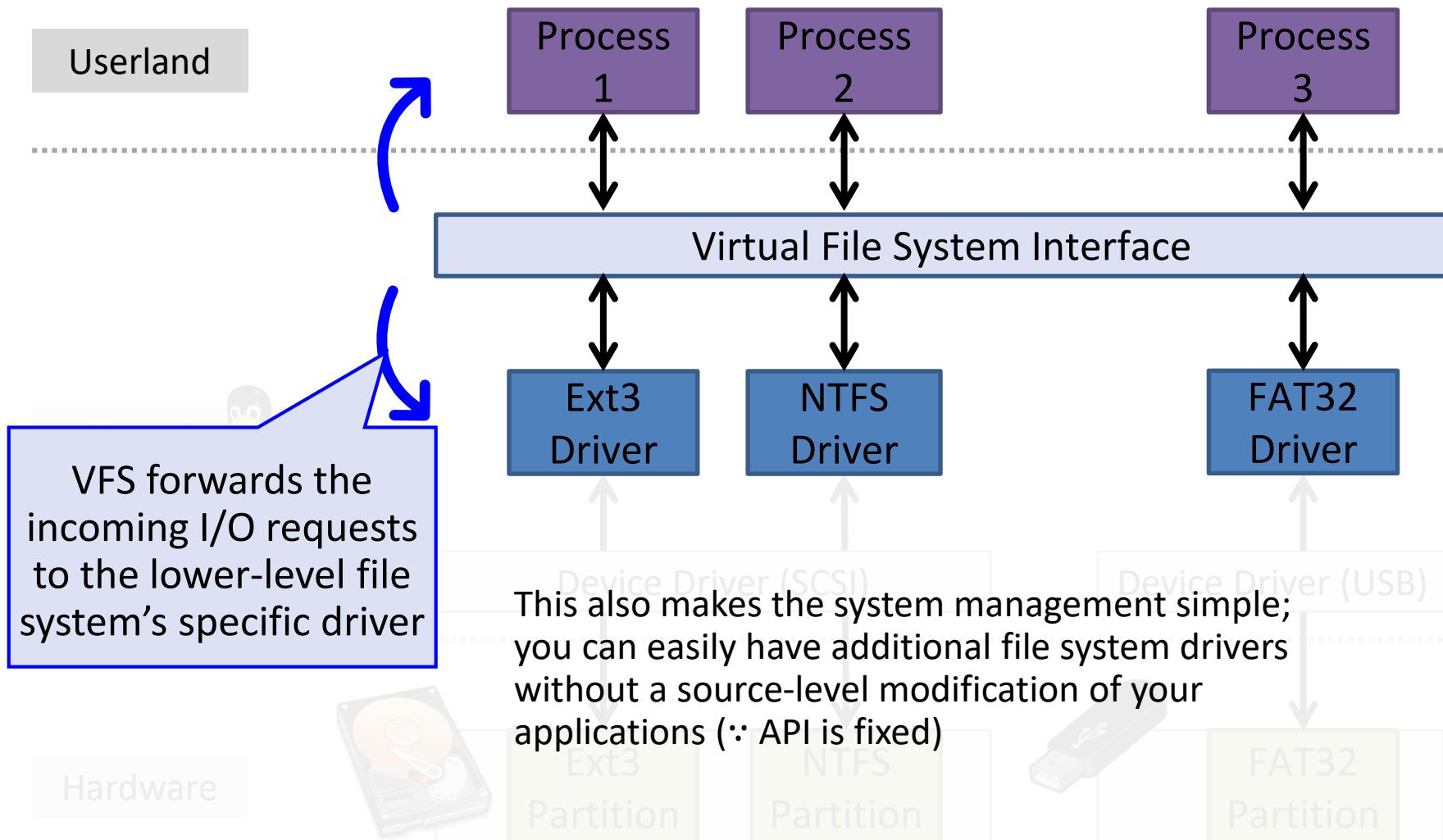
Solution: Virtual File System



Solution: Virtual File System

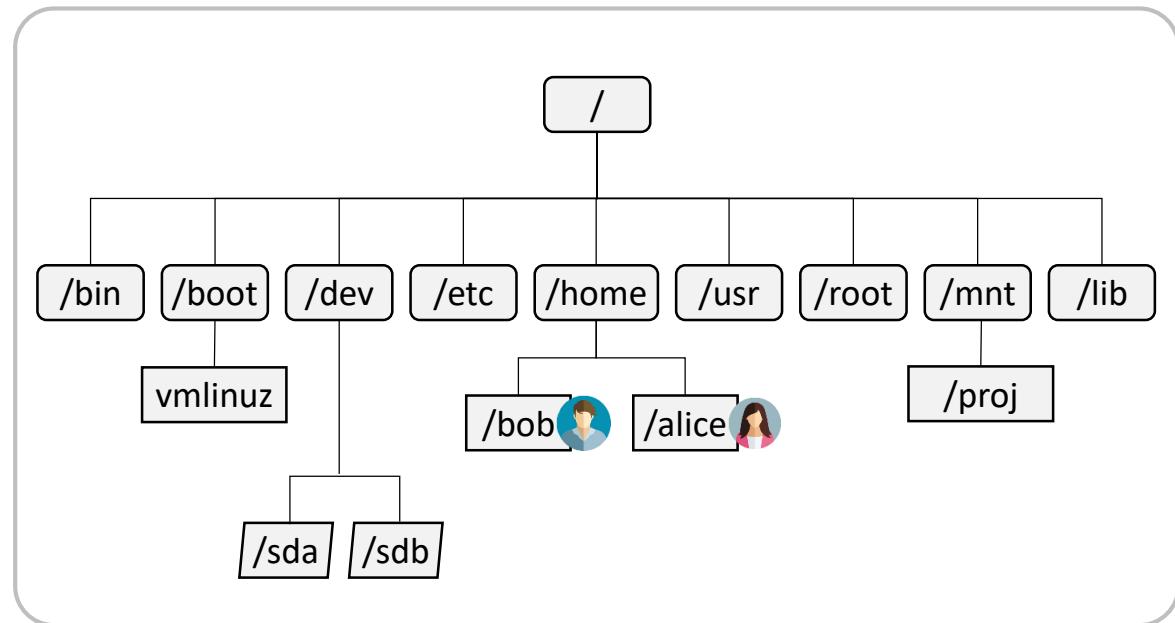


Solution: Virtual File System



Step #4: Mount the FileSystem

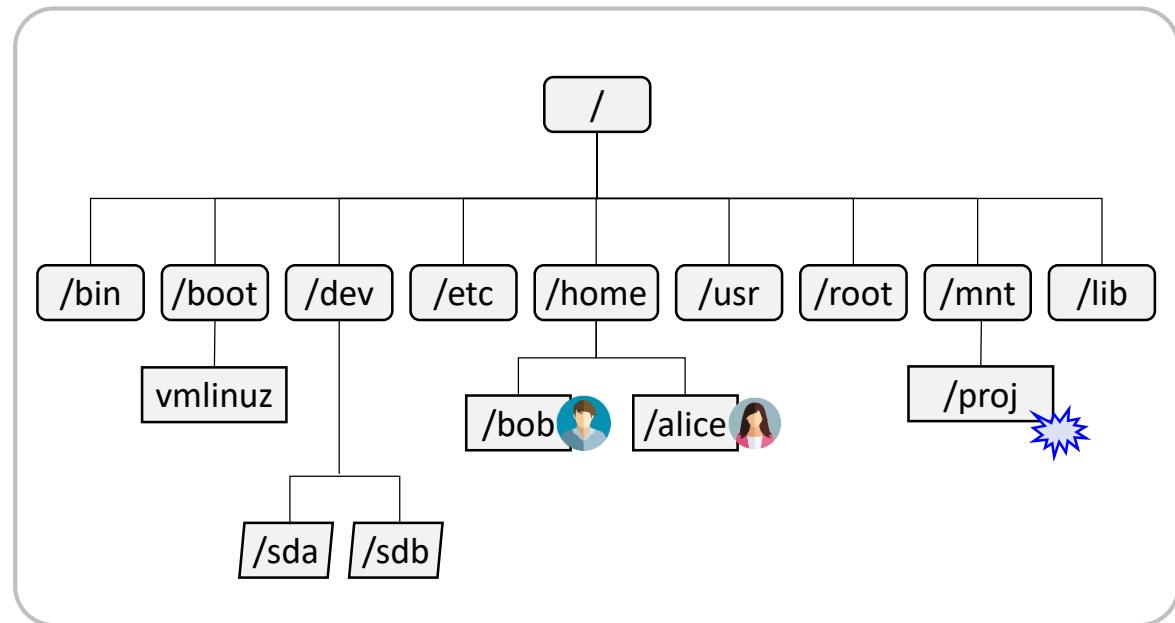
Step #4: Mount the Keys
in the desired place



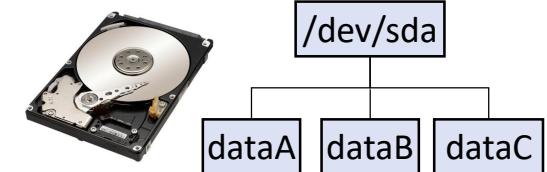
Each file system should be mounted into a unified file hierarchy before the system uses

Step #4: Mount the FileSystem

**Step #4: Mount the Keys
in the desired place**

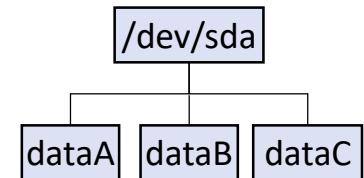
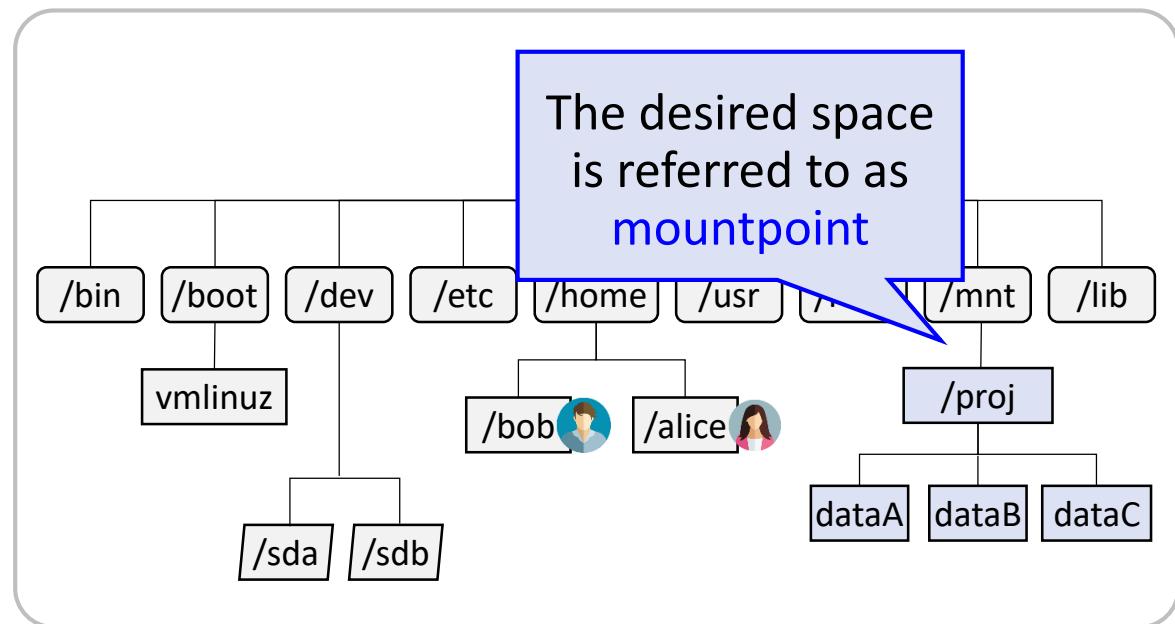


Let's suppose that you want to
access all the data of a
filesystem (/dev/sda) via /proj



Step #4: Mount the FileSystem

Step #4: Mount the Keys
in the desired place



Usable Command: mount/umount

```
$ mount /dev/sba /mnt/proj
```

<device> <mountpoint>

mount command mounts the
filesystem for a certain block
device file to a mountpoint

Usable Command: mount/umount

```
$ mount /dev/sba /mnt/proj
```

<device> <mountpoint>

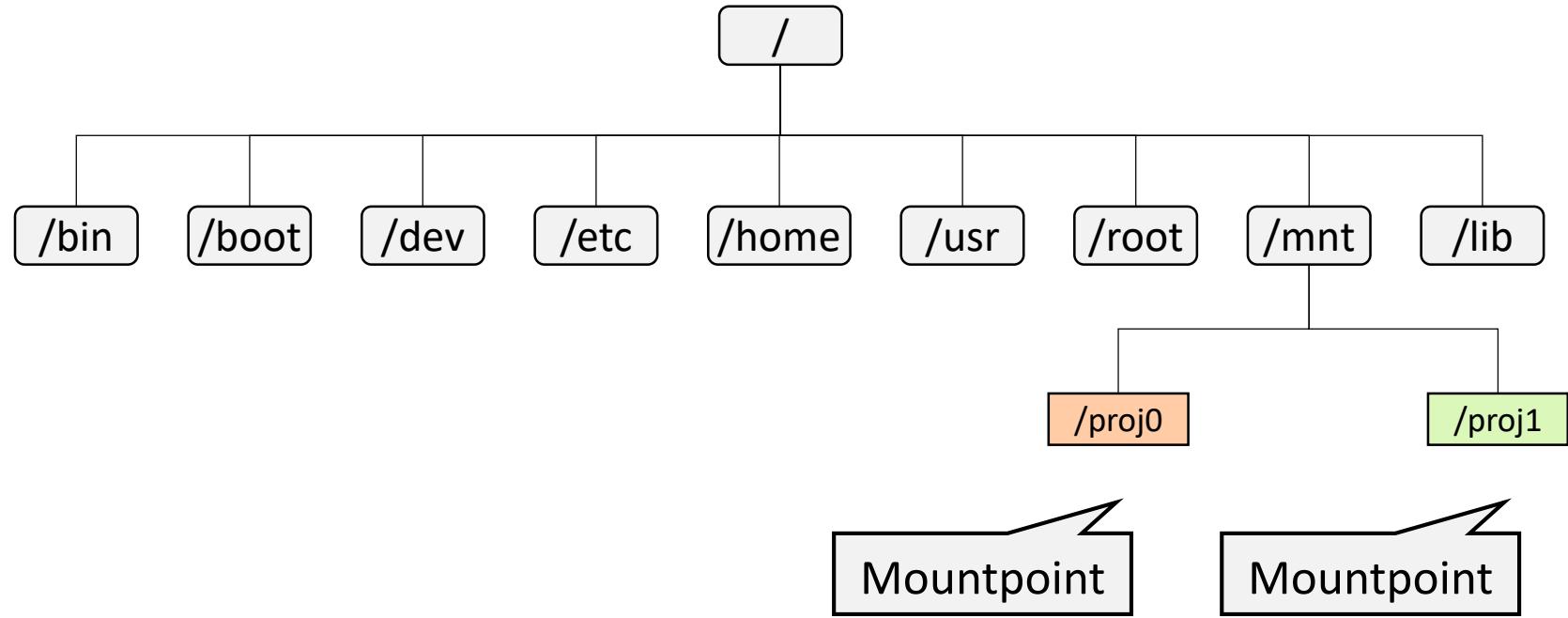
mount command mounts the filesystem for a certain block device file to a mountpoint

```
$ umount /dev/sba  
$ umount /mnt/proj
```

umount command unmounts the filesystem that was previously mounted. The file system can be identified by its associated device or mountpoint

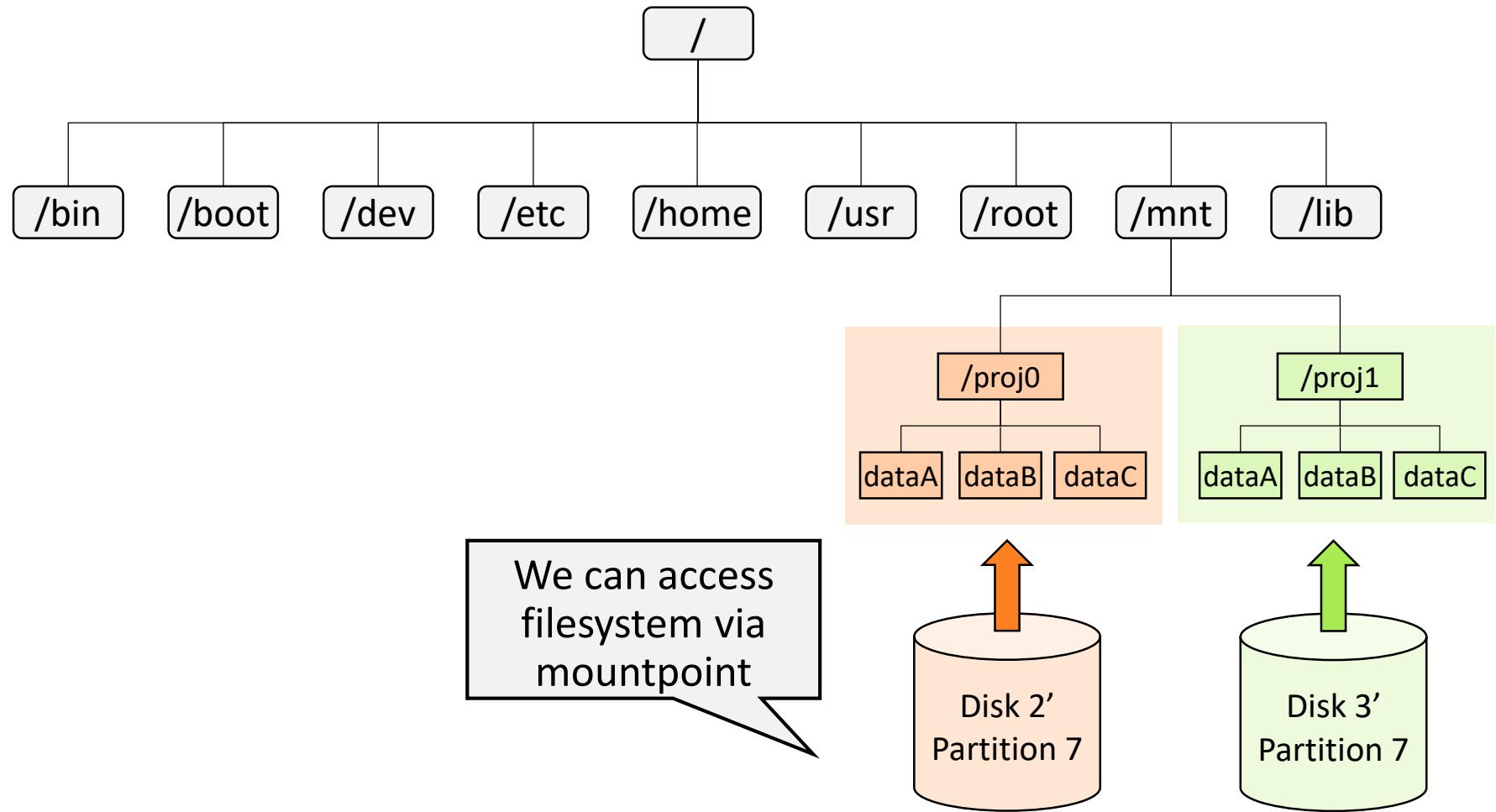


Mounting The FileSystem



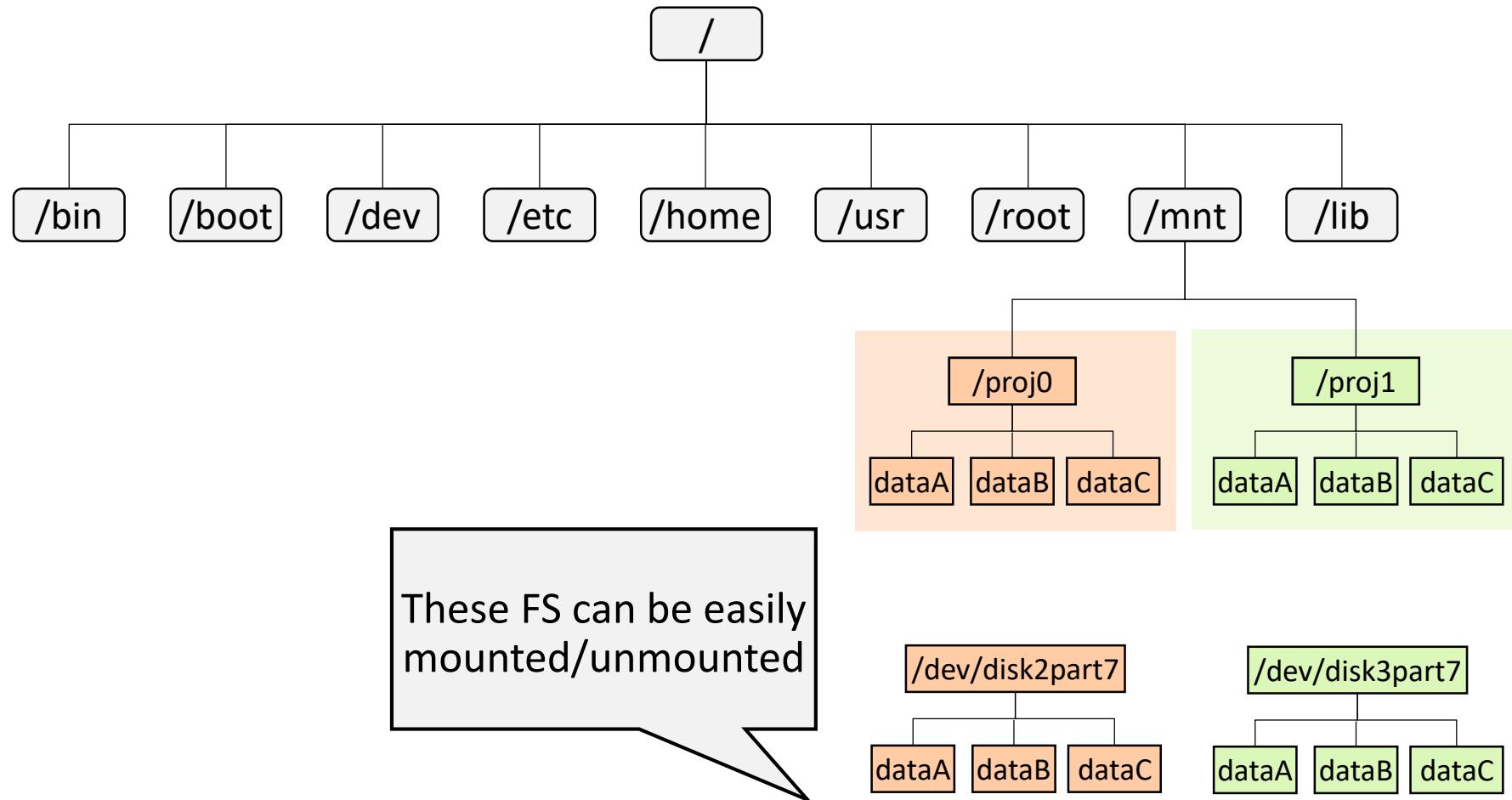


Mounting The FileSystem



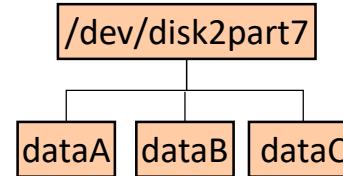
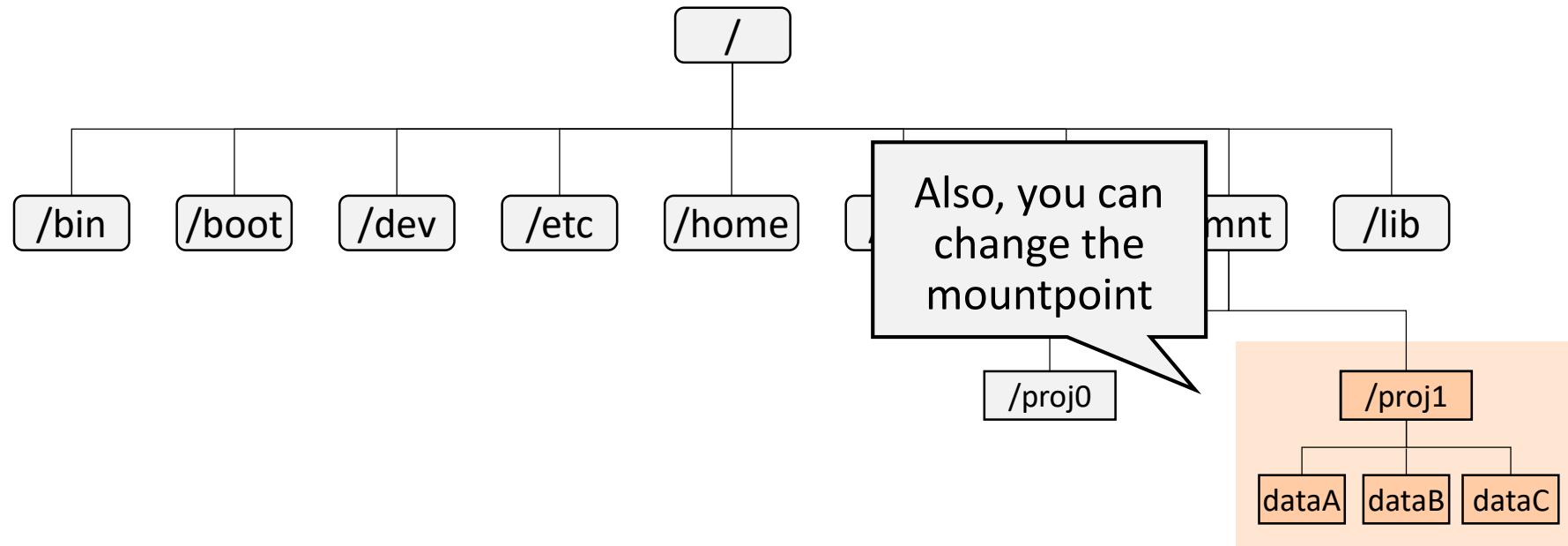


Mounting The FileSystem



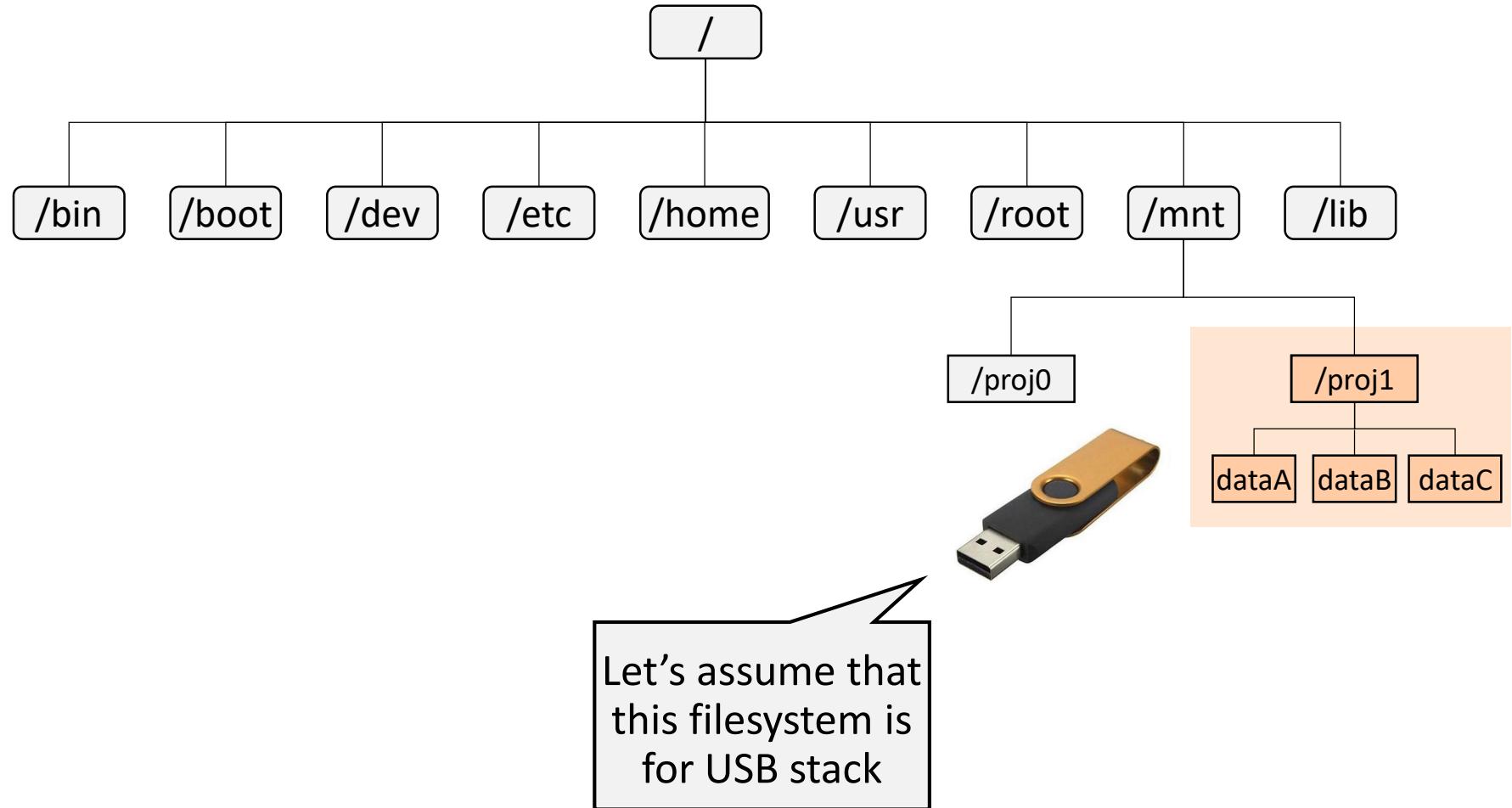


Mounting The FileSystem



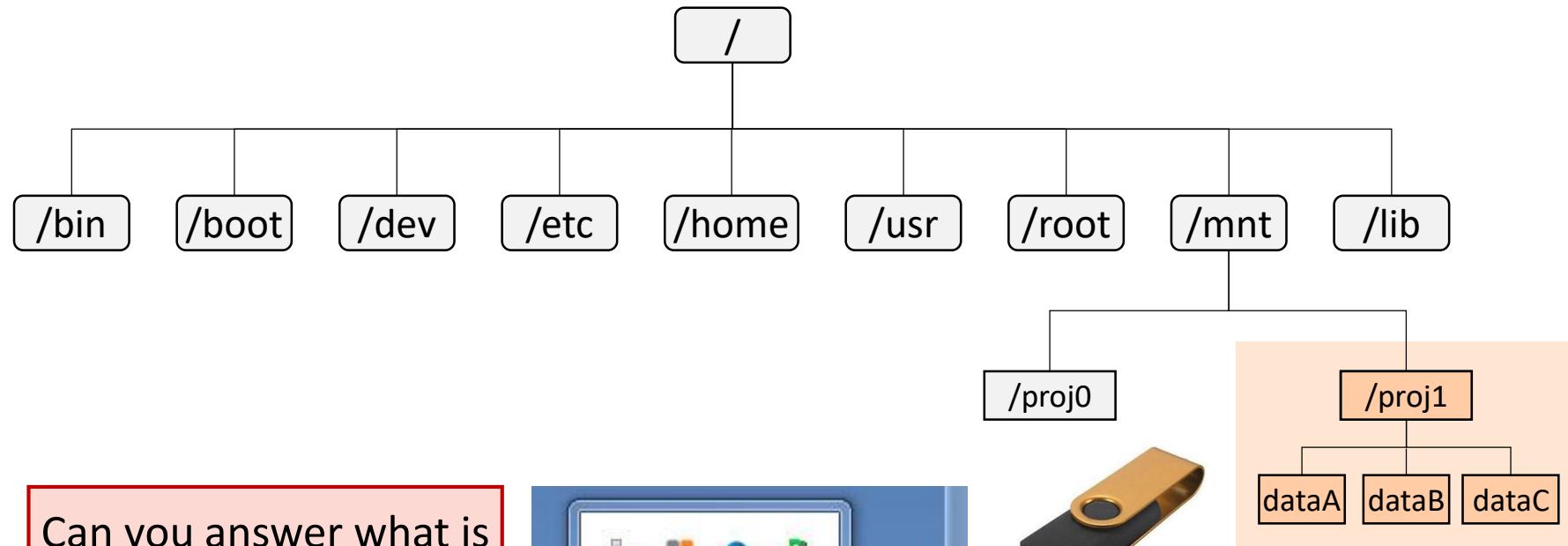


Mounting The FileSystem





Mounting The FileSystem

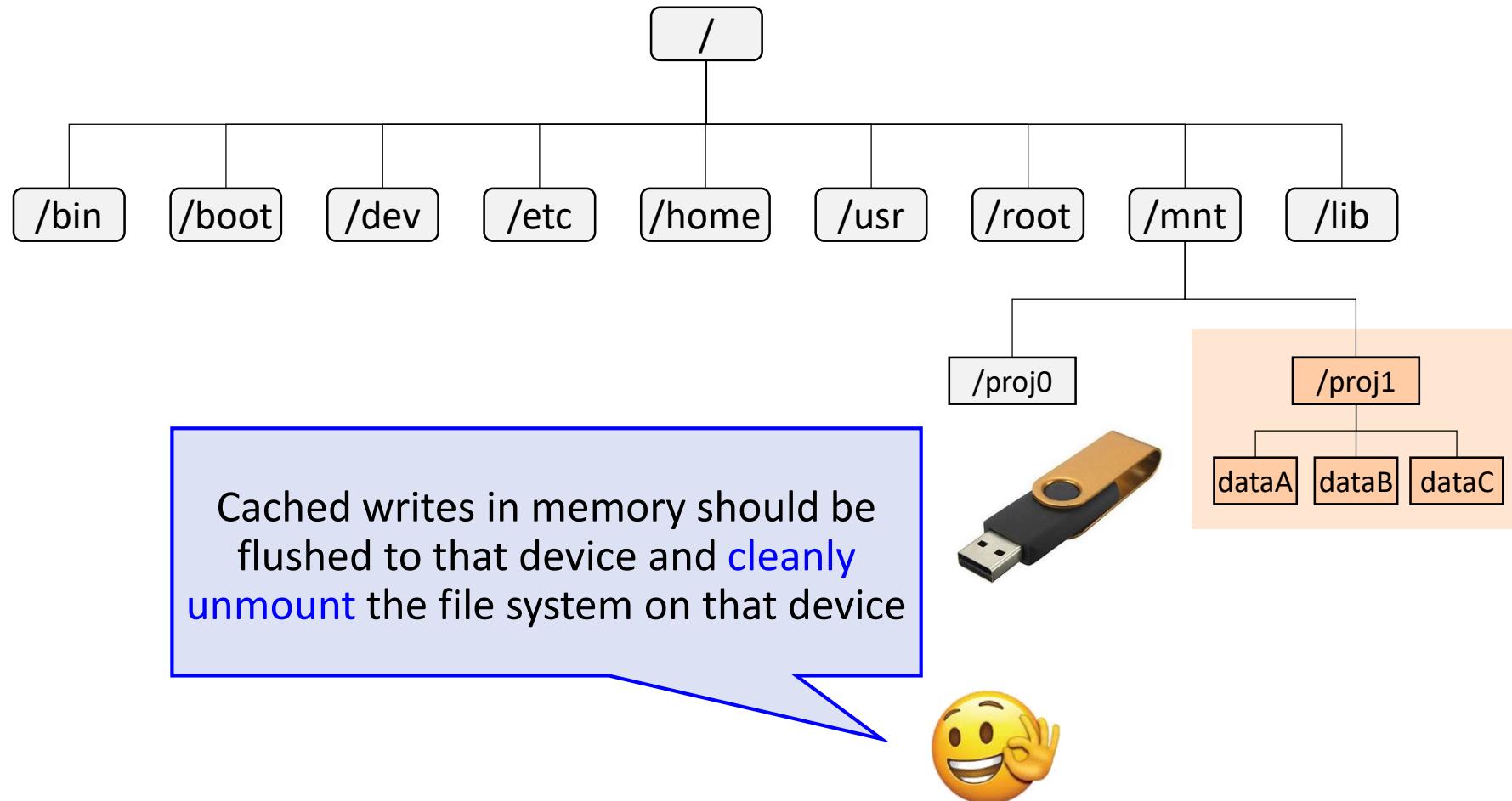


Can you answer what is the meaning of “safely eject” a device?



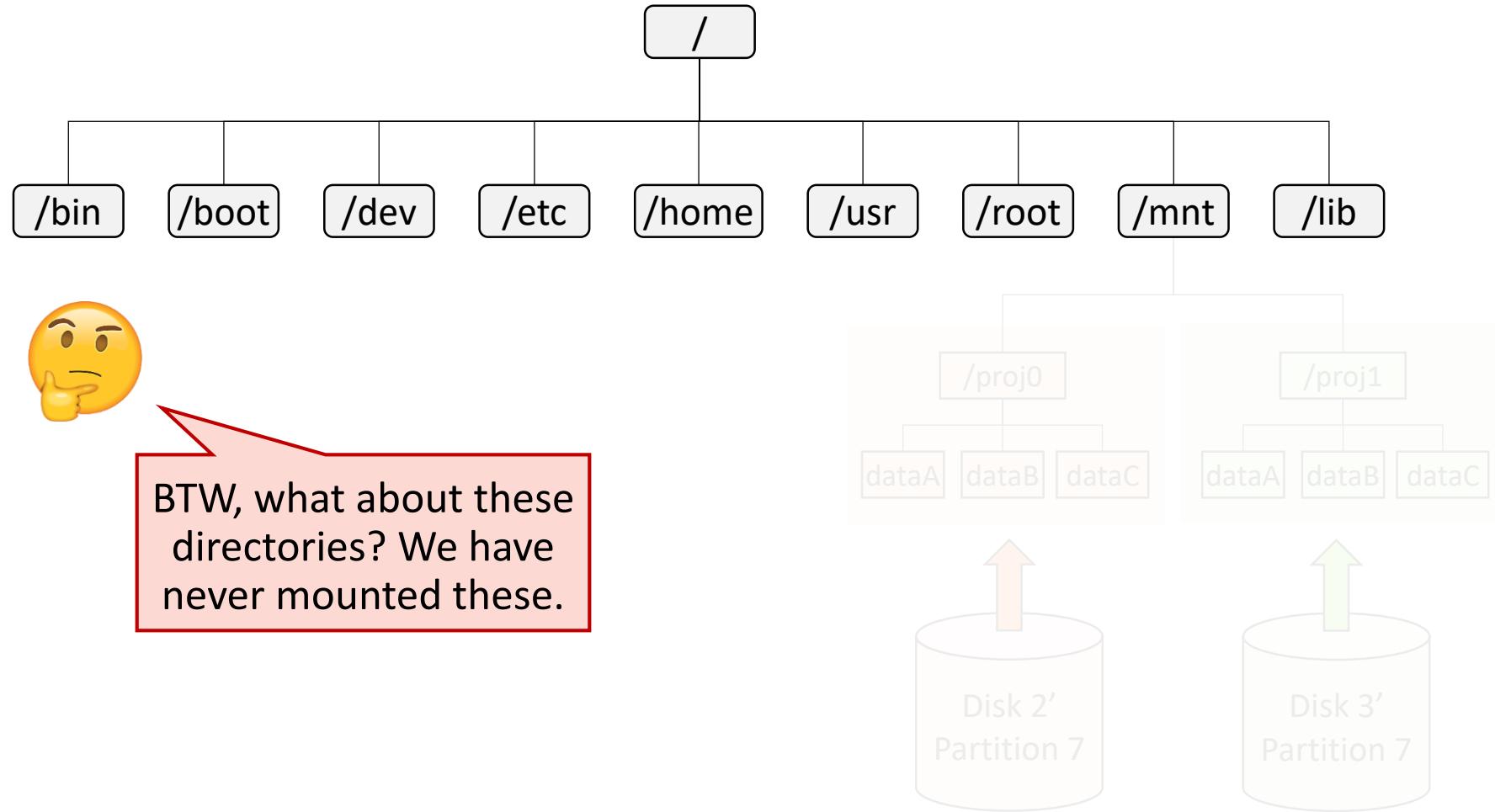


Mounting The FileSystem

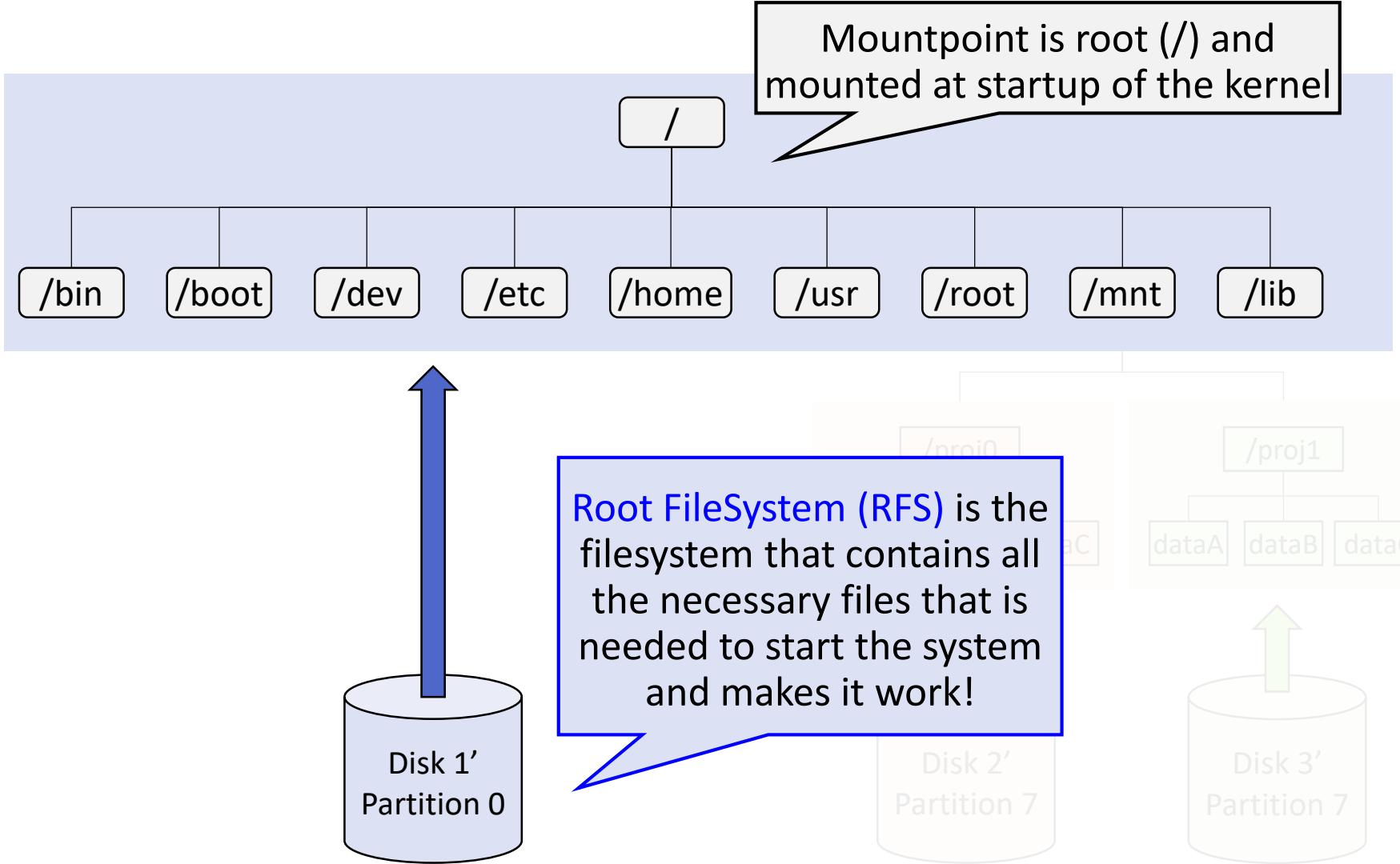




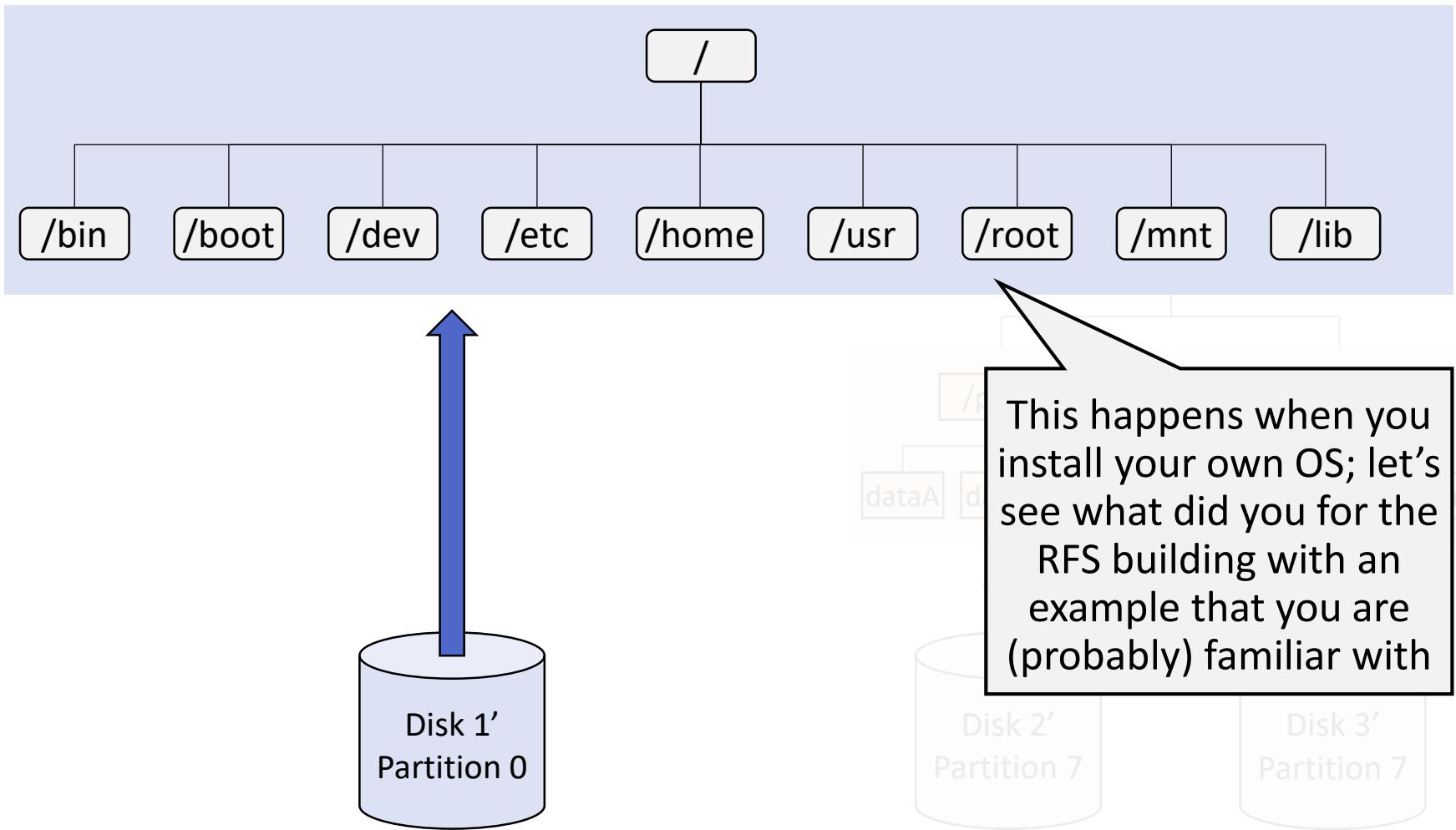
Mounting The FileSystem

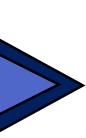


Root FileSystem

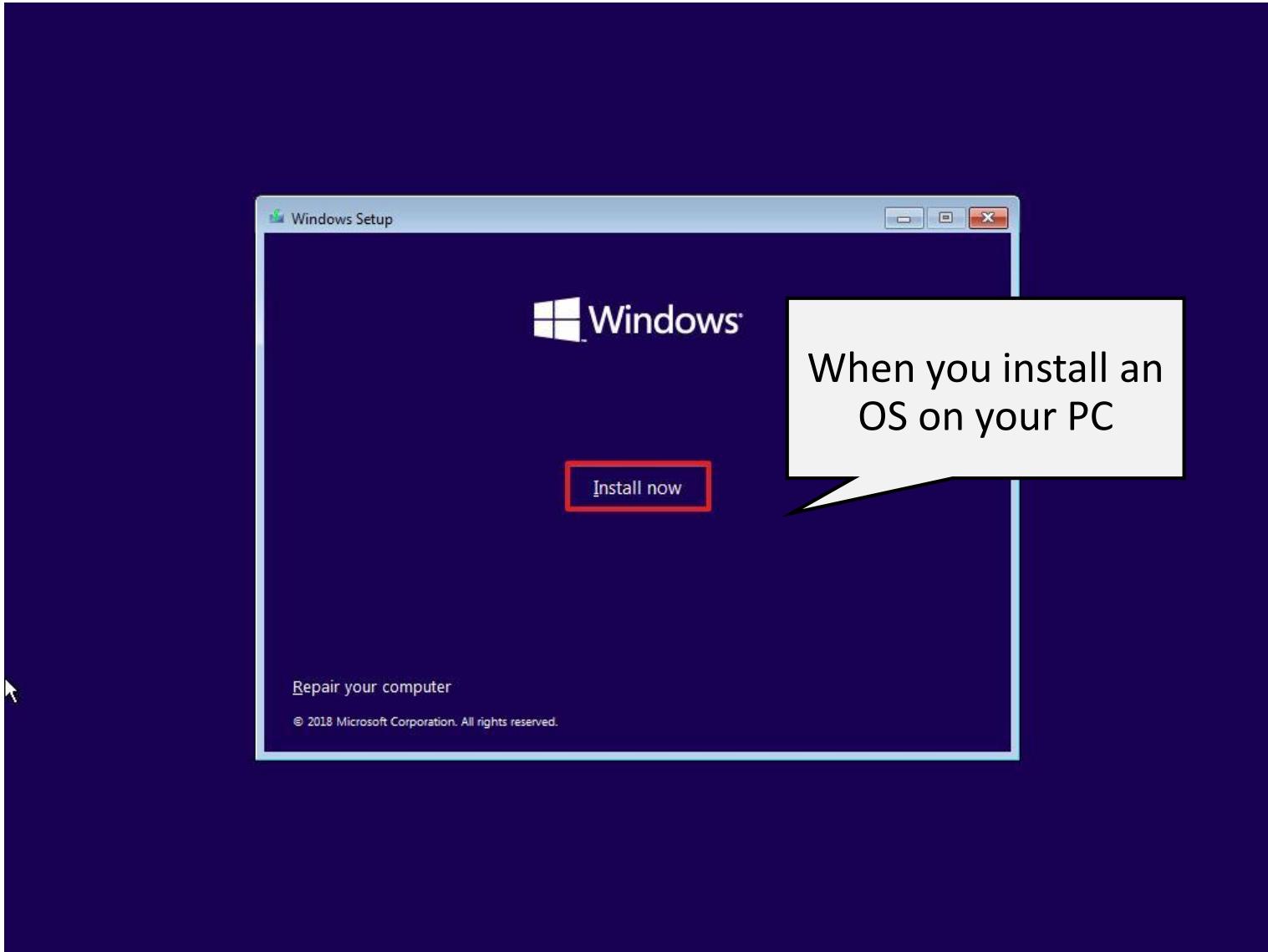


Root FileSystem



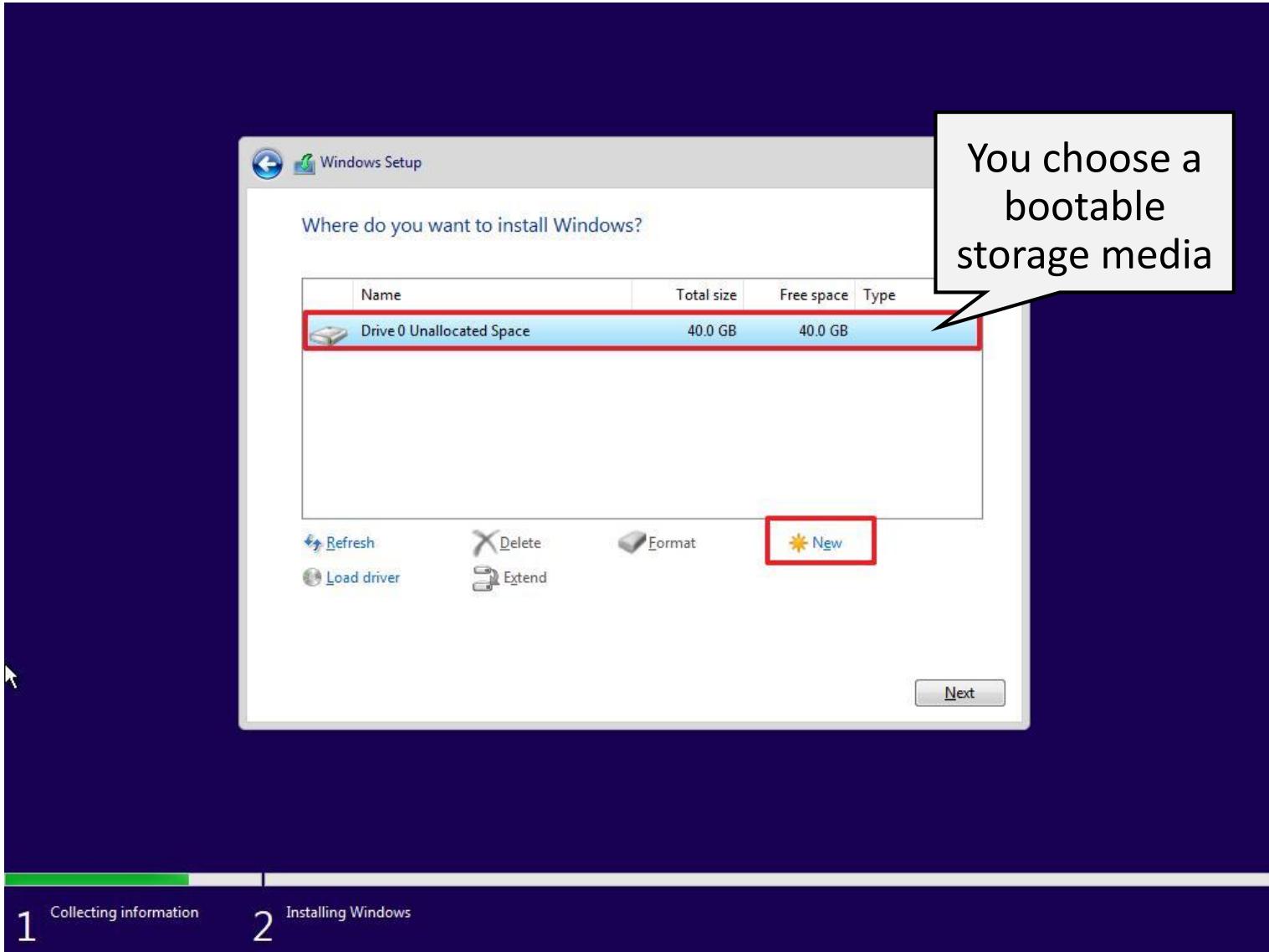


Building Root FileSystem

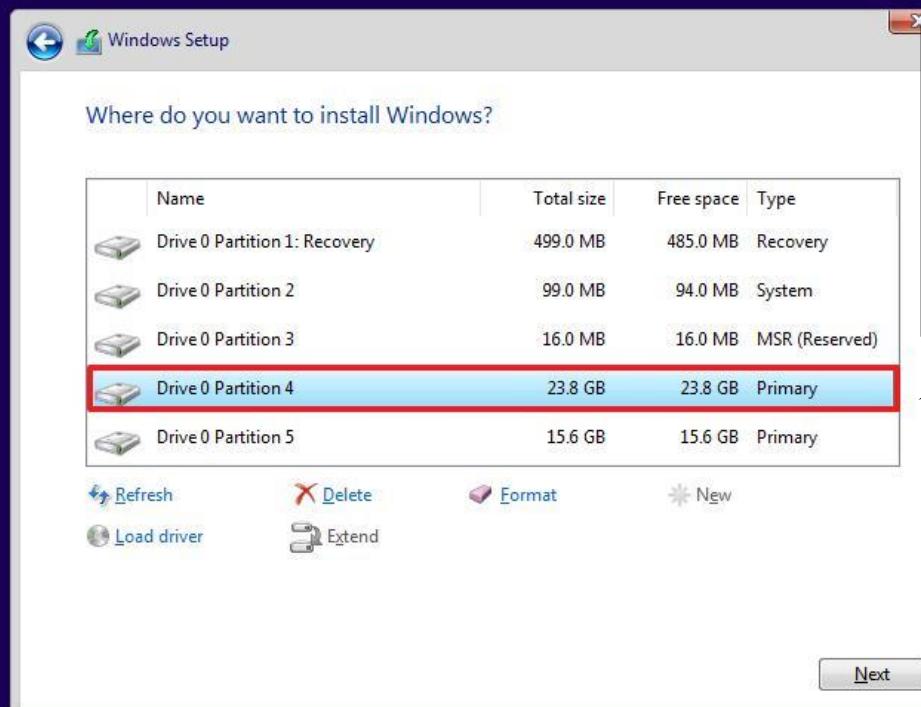




Building Root FileSystem



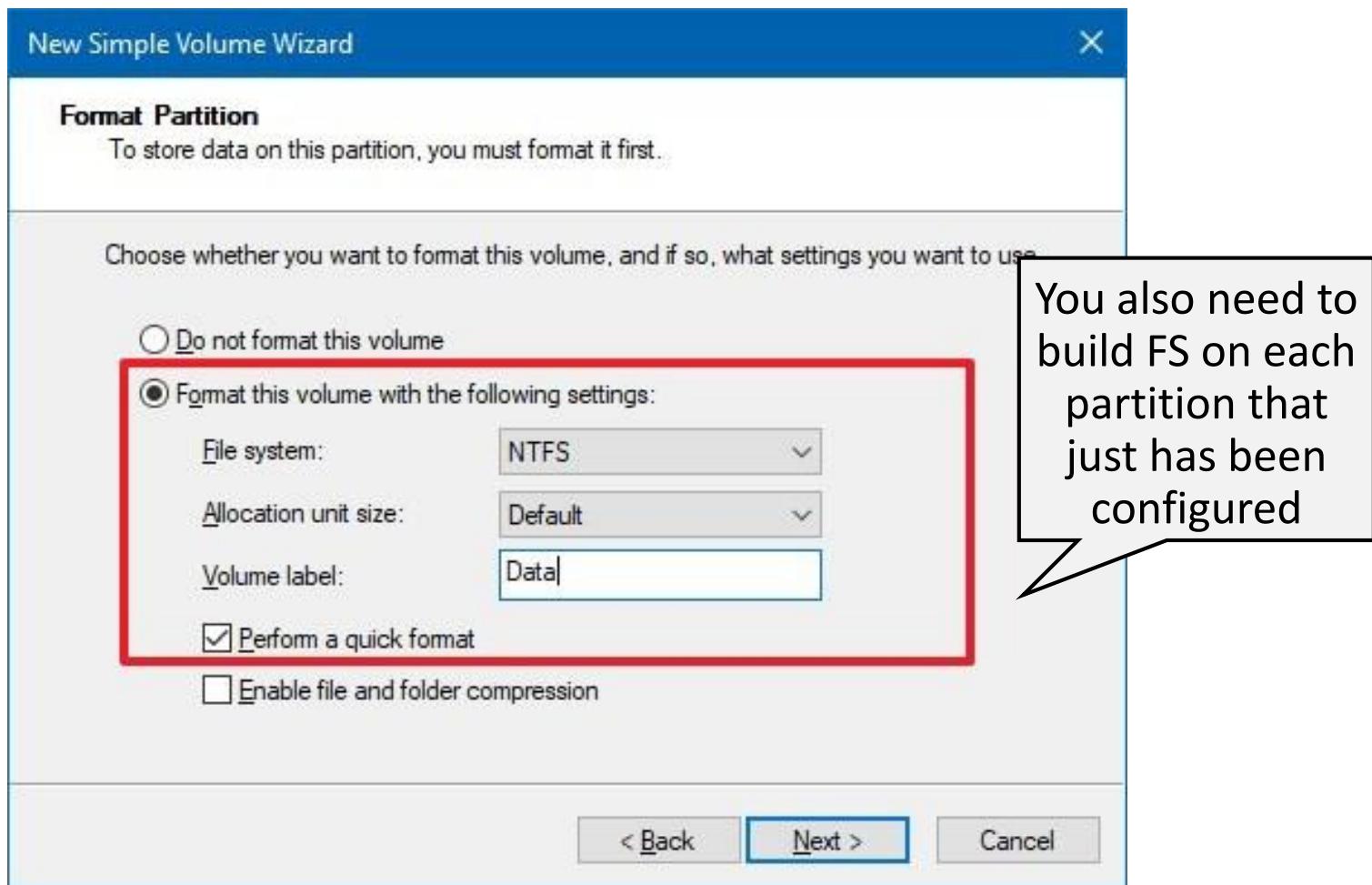
Building Root FileSystem



Make then multiple partitions if you need such as swap.



Building Root FileSystem



What's Next?

- Today we understand the basics of file system
- In the next lecture, let's check how FS can be implemented

21S EE415

File System II

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



File System II

Myoungsoo Jung
Computer Division

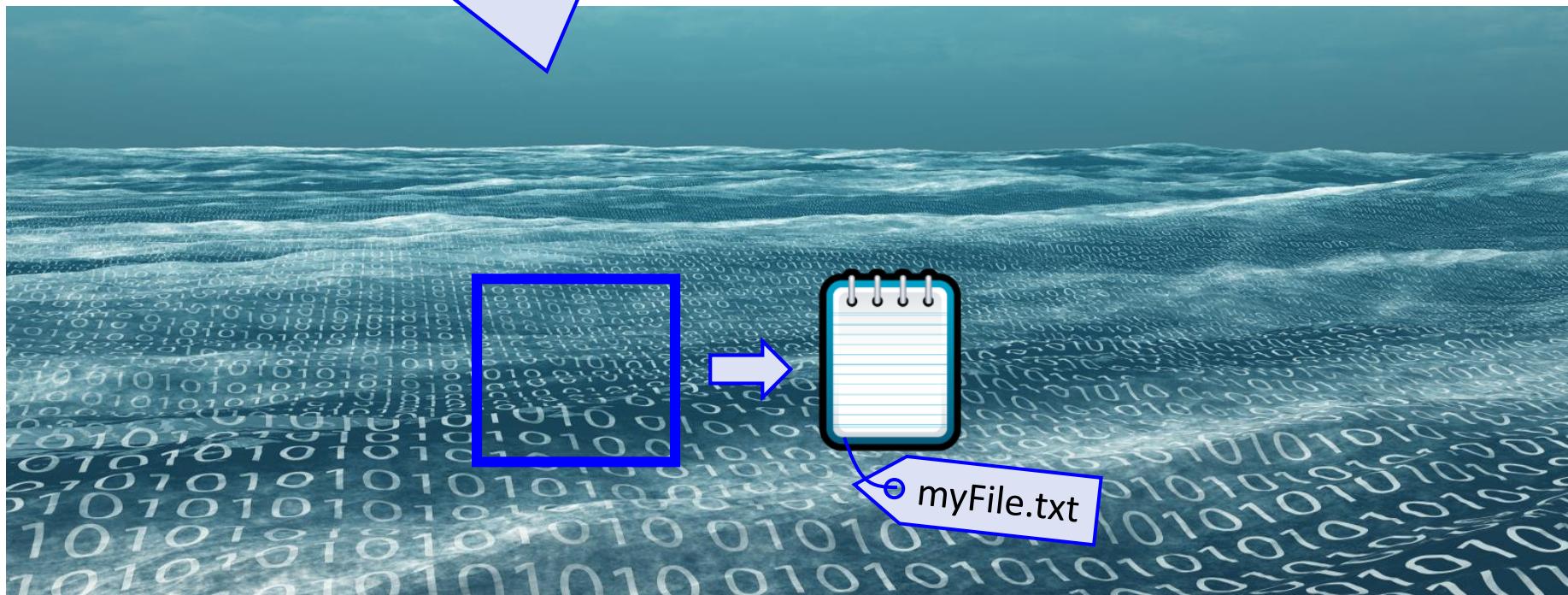
Computer Architecture and Memory systems Laboratory

KAIST EE

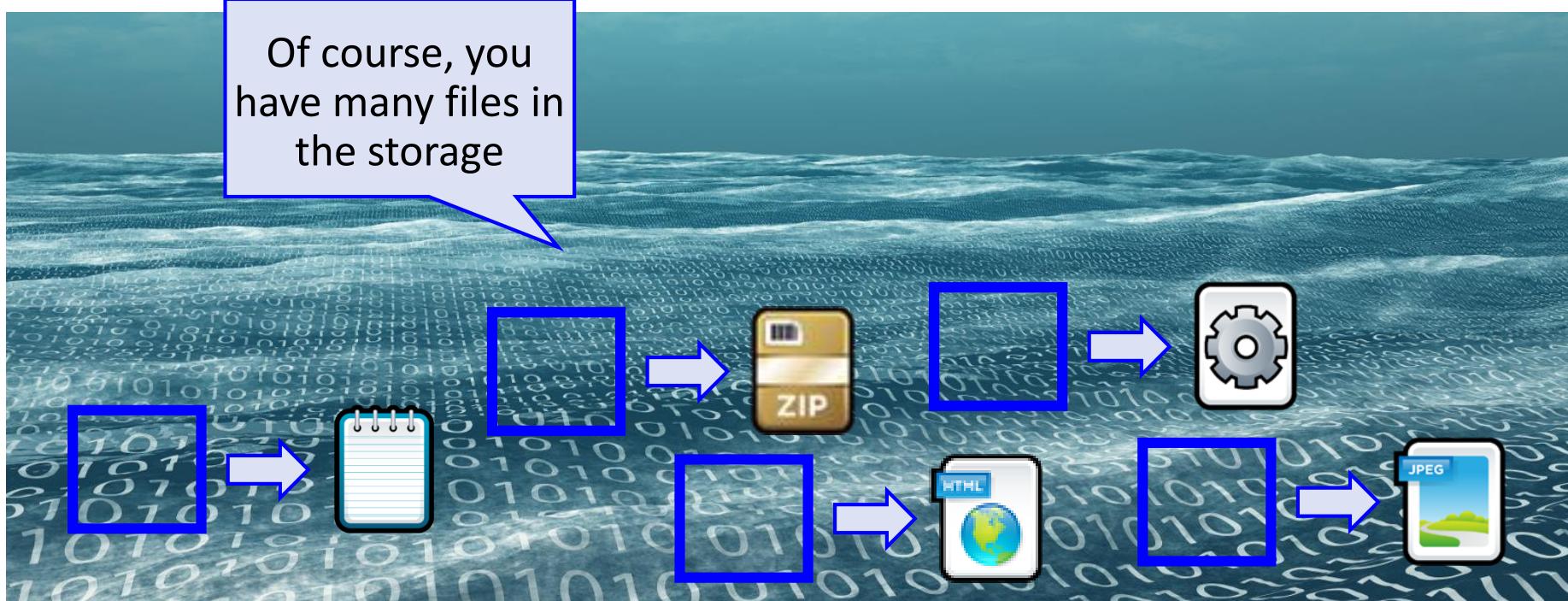
CAMEL*ab* 

[Recap] Bytes → File → Name

Storage has no mechanism to classify your data and structure them appropriately, **File-System** groups a set of meaning bytes as a “File” and names it for you ☺

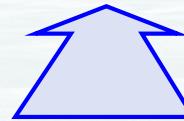
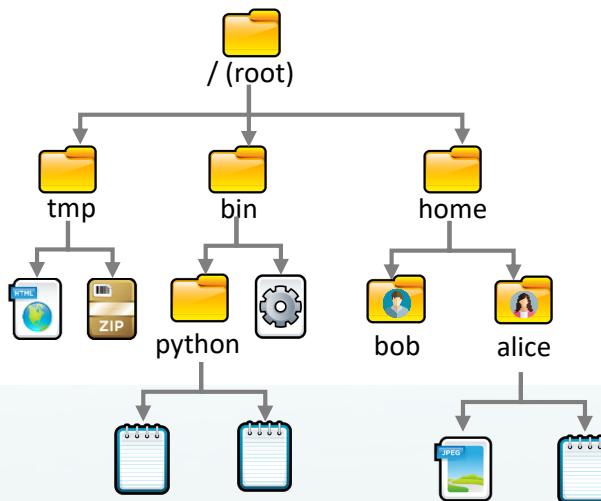


[Recap] Bytes → File → Name

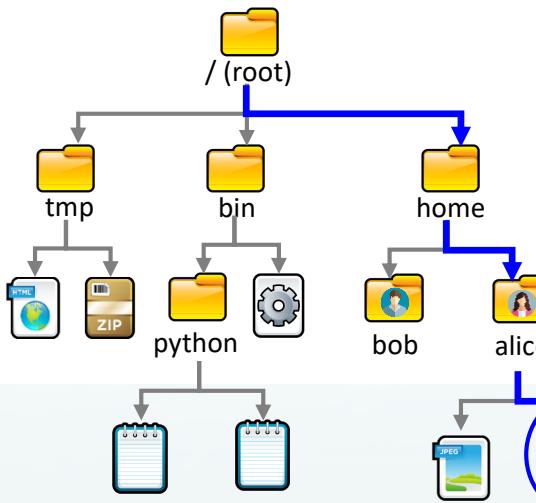


[Recap] Bytes → File → Name

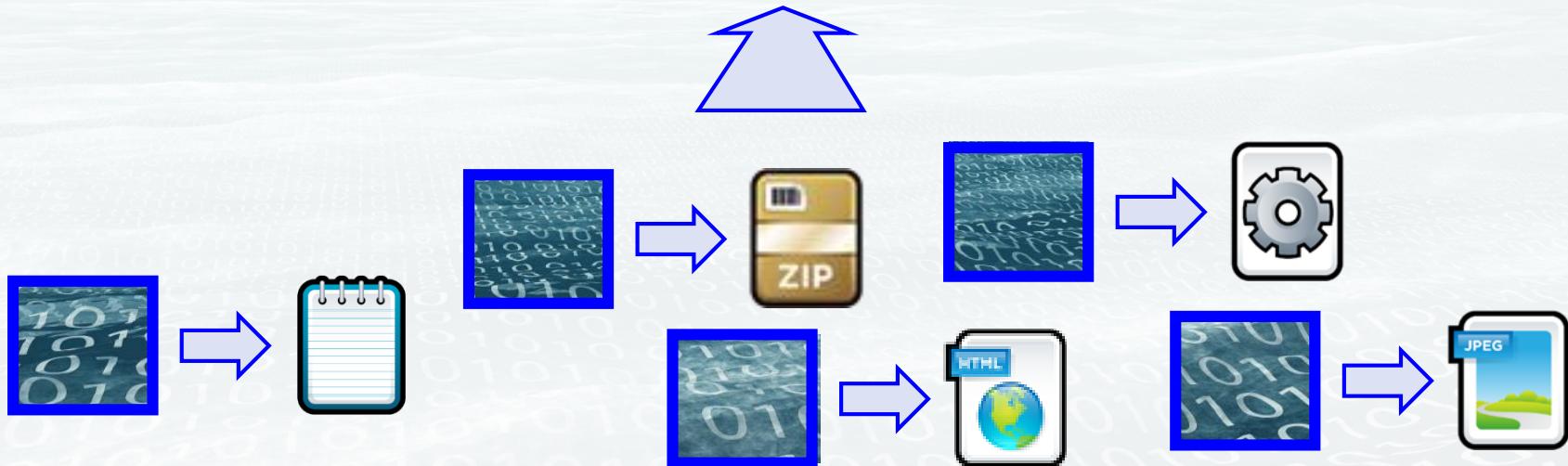
Files are managed by the **directory tree** (leaves are file)



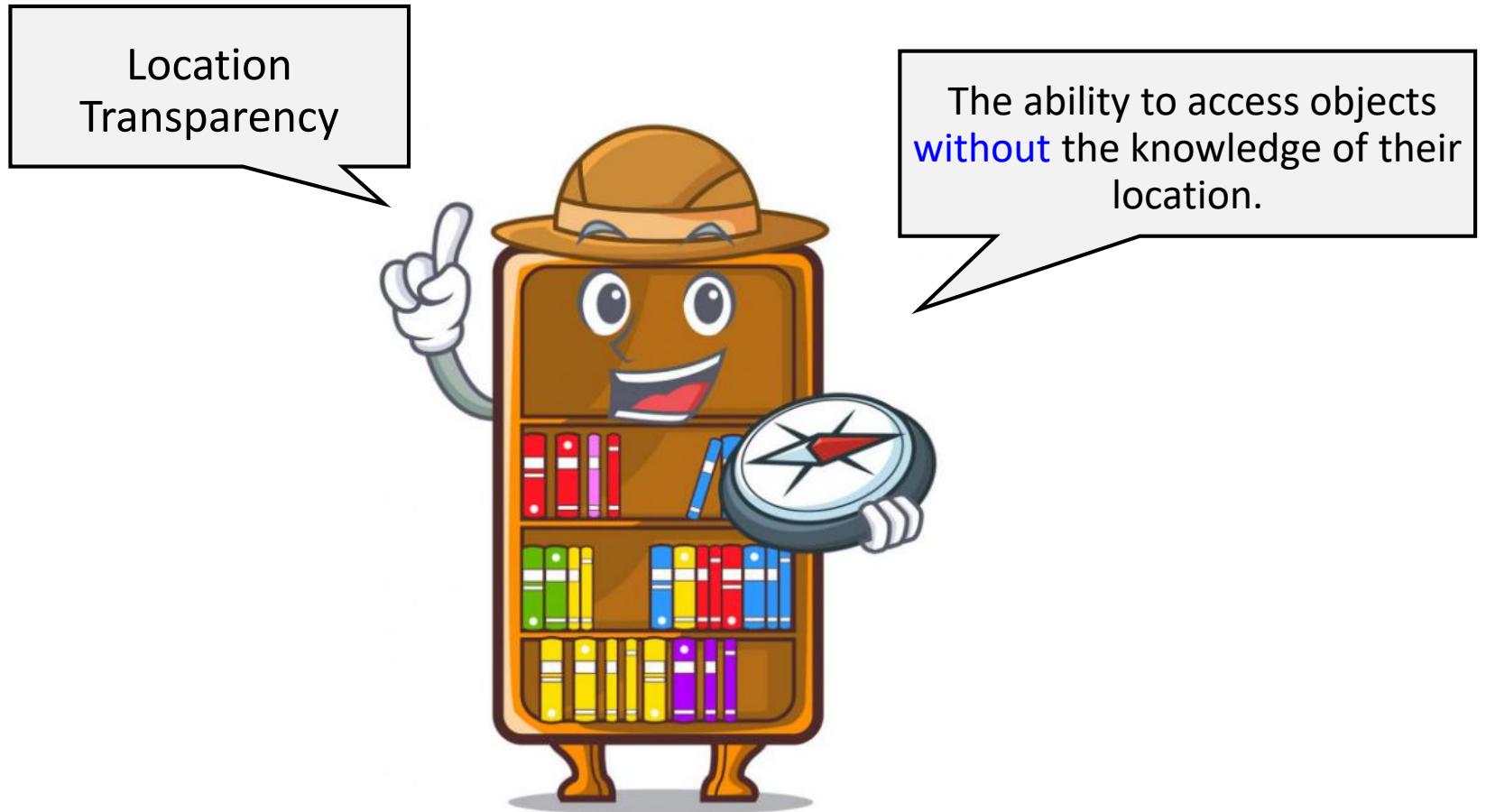
[Recap] Bytes → File → Name

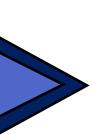


In other words, this file can be accessed with the path
(/home/alice/myFile.txt)



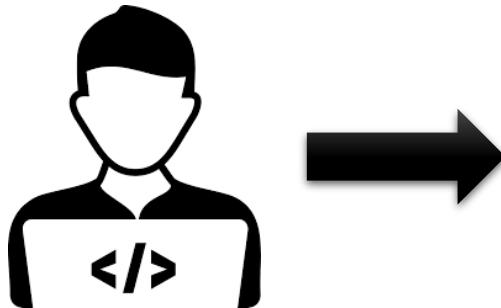
An important Role of File-System



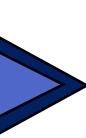


Location Transparency in FS

Use of **names** to identify resources rather than their actual locations

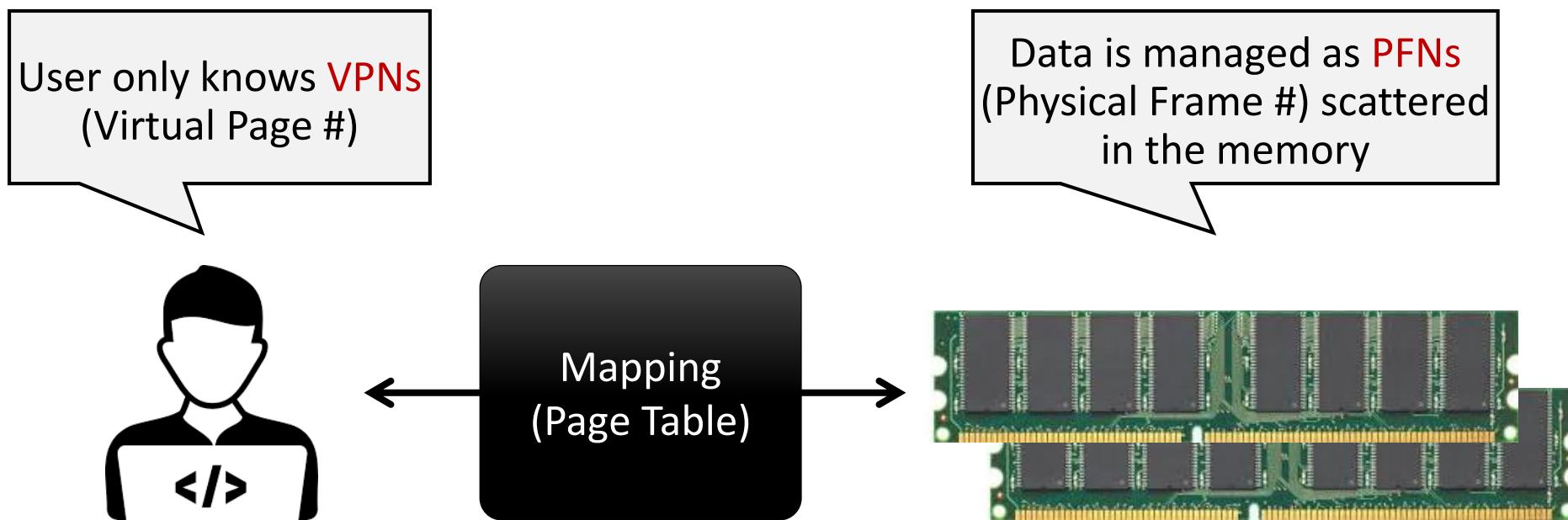


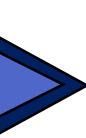
Files are accessed by a unique file name, but the actual data is stored in **physical sectors** scattered around a disk



Similarity w/ Virtual Memory

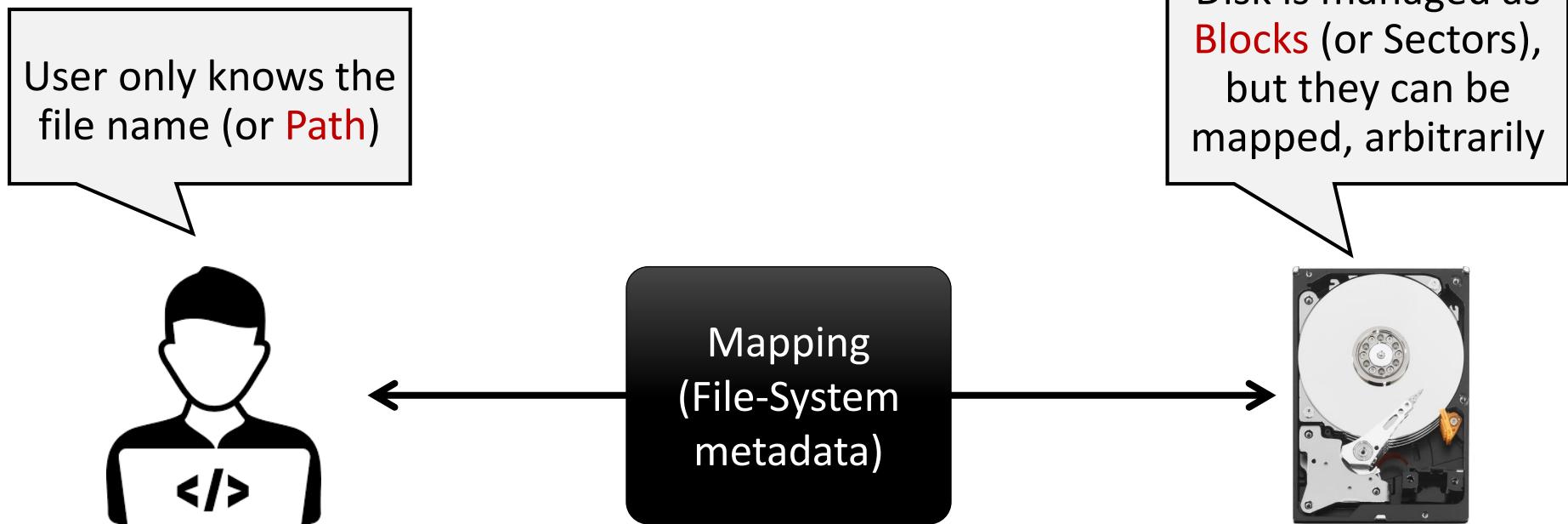
- We already reviewed the location transparency concept in the Virtual Memory (VM)

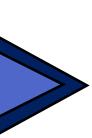




Transparency of File-System

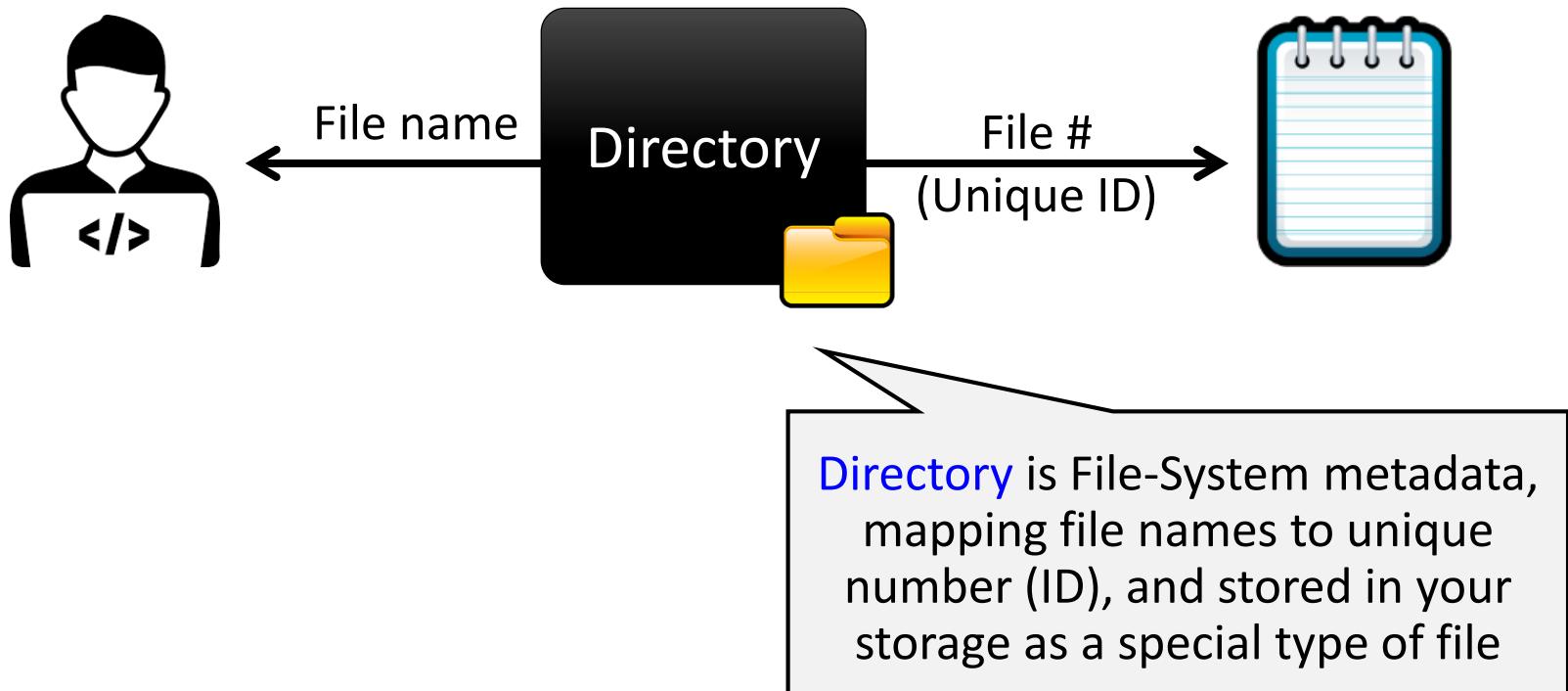
- Similar with VM's page tables, File-System requires mapping information





File-System Metadata

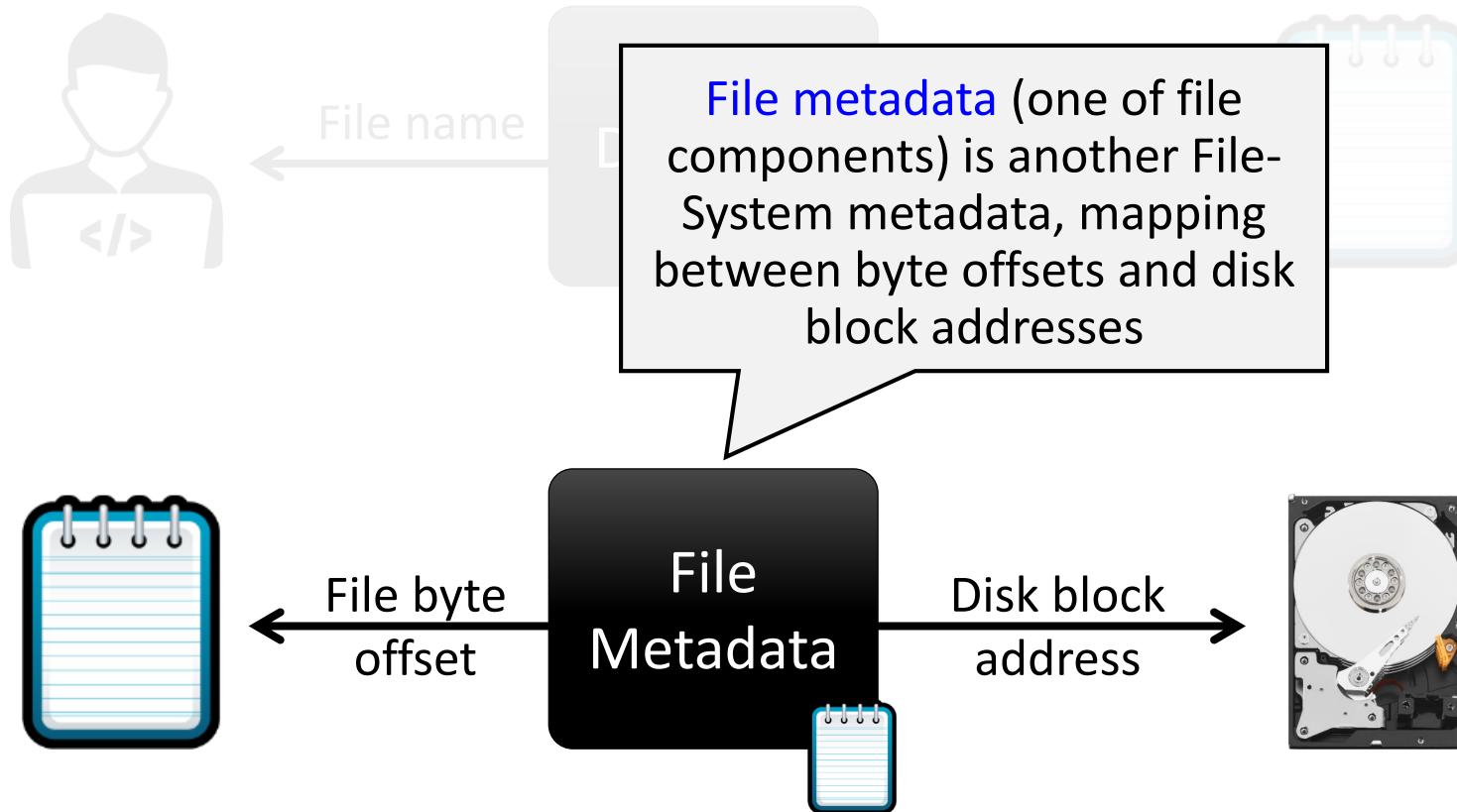
- File-System metadata can be largely classified into two

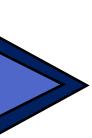




File-System Metadata

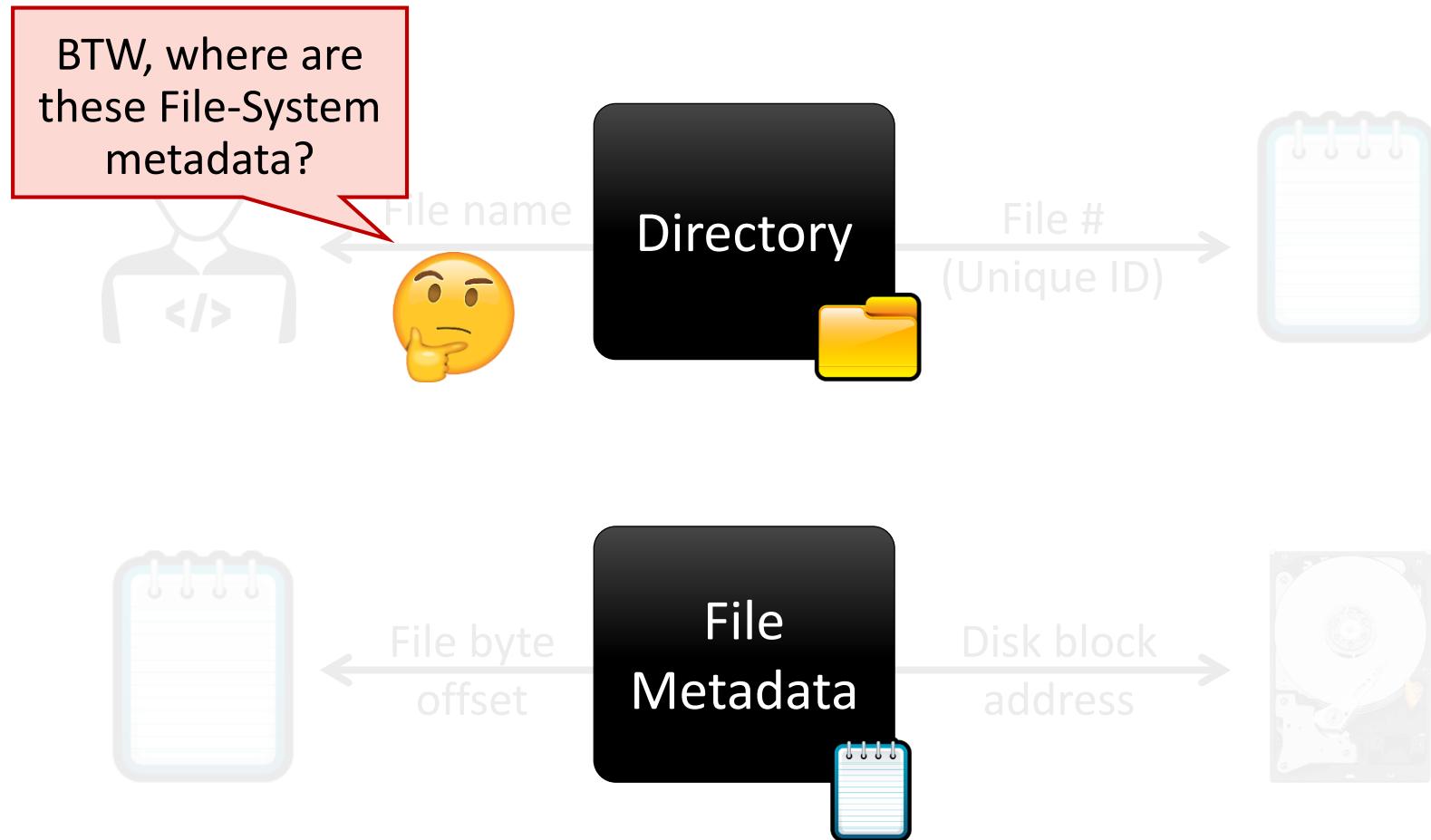
- File-System metadata can be largely classified into two





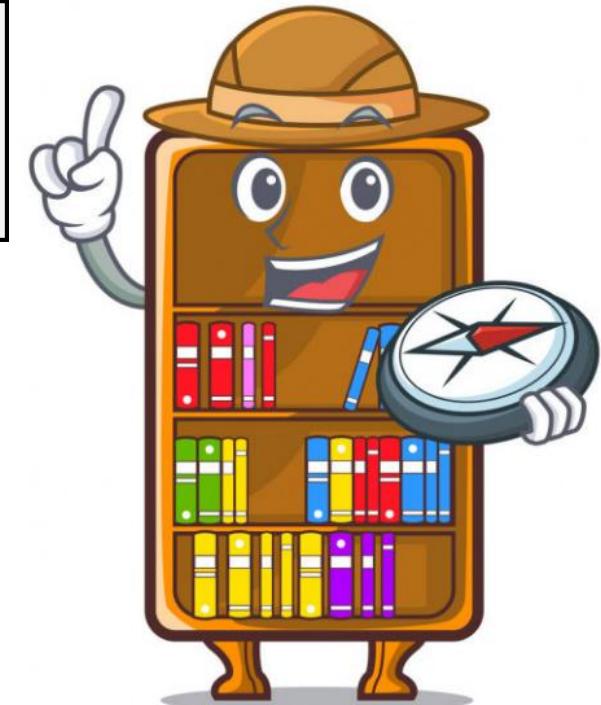
File-System Metadata

- File-System metadata can be largely classified into two



FS Metadata: On-disk Layout

File-System
metadata are
structured on
your storage



Example: USB

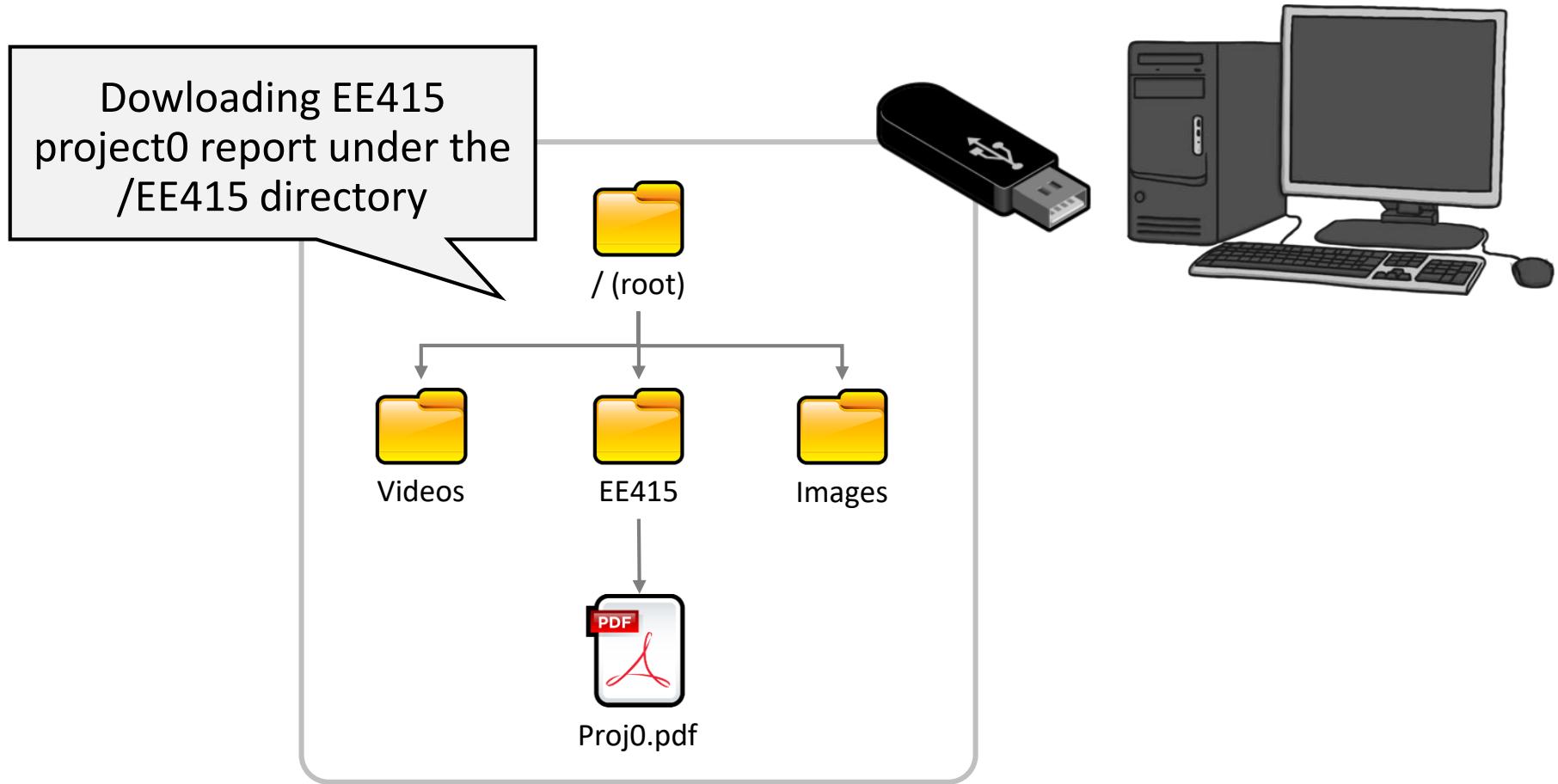
- Think about a USB

You plug a USB stick into your PC



Example: USB

- Think about a USB



Example: USB

- Think about a USB

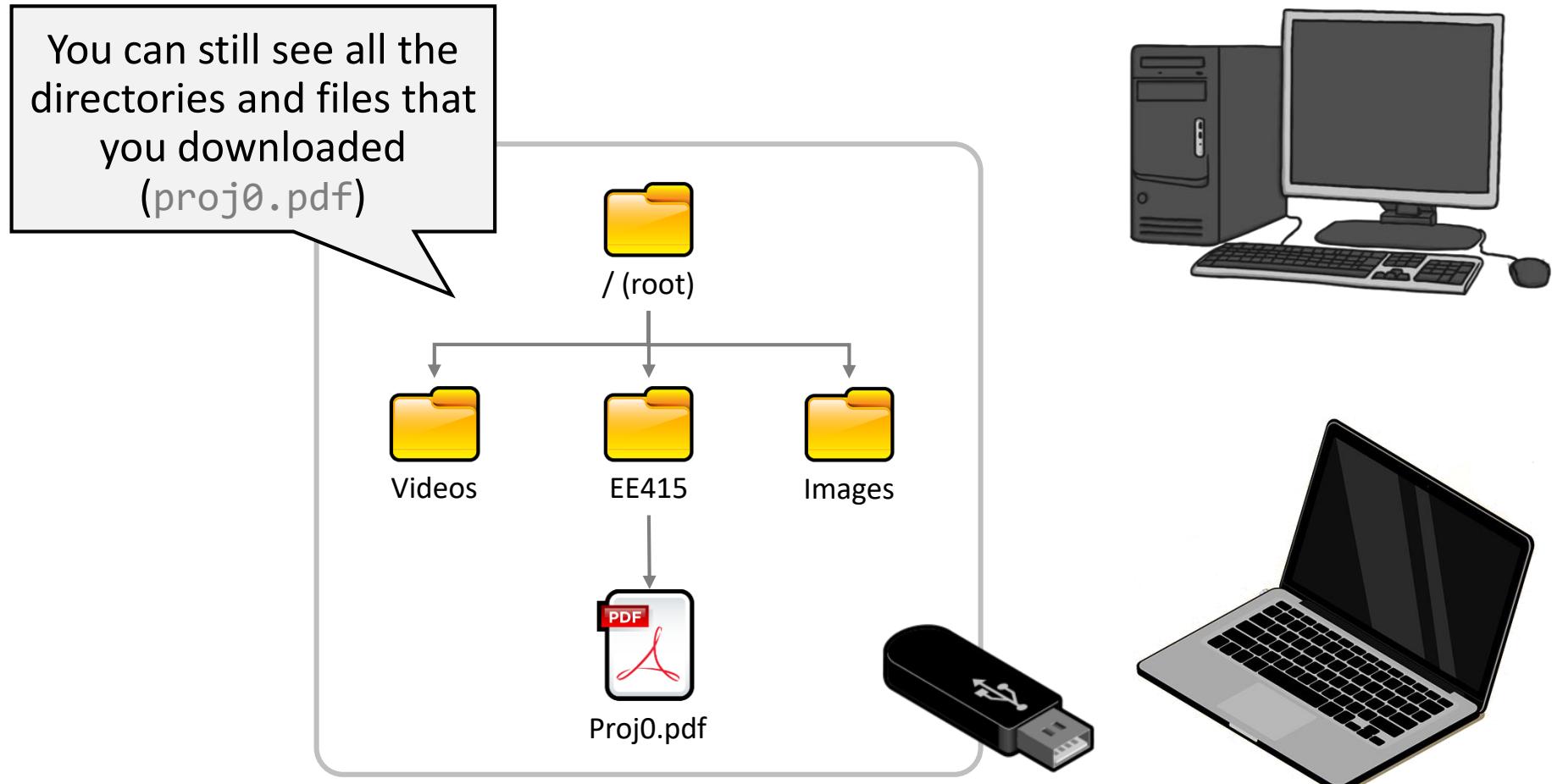


What if you plug the USB out
from the PC and plug it in into a
different laptop system?



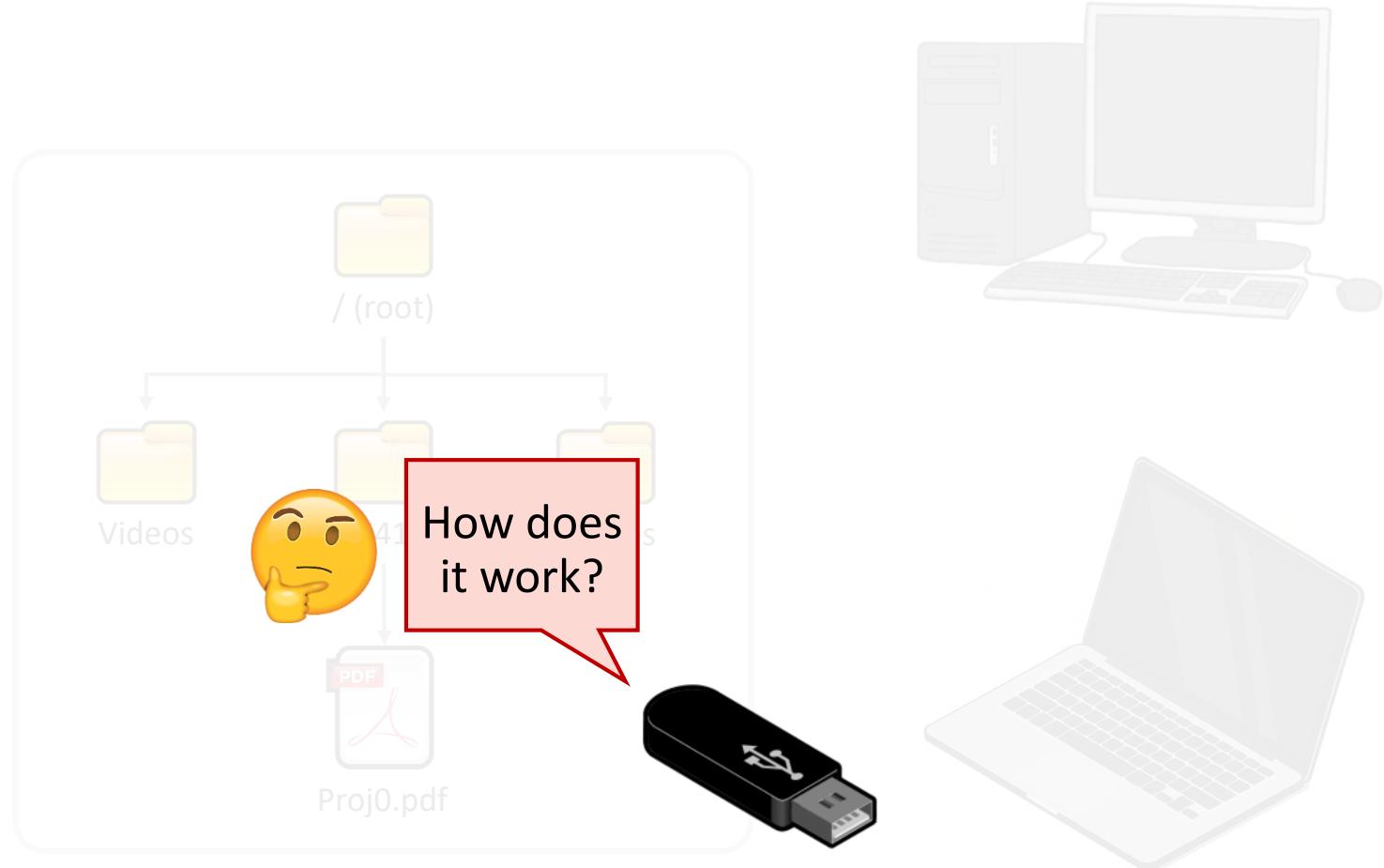
Example: USB

- Think about a USB



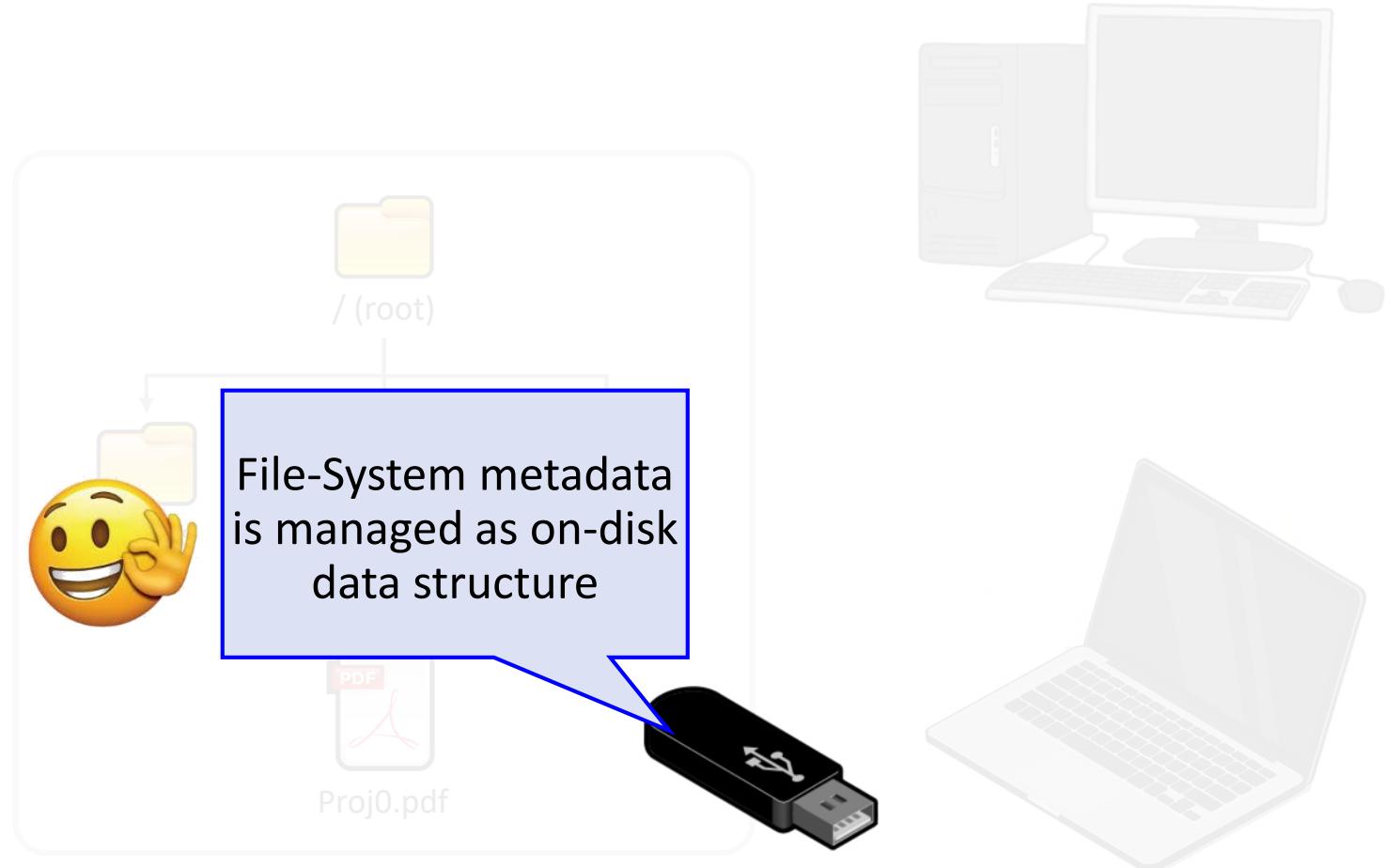
Example: USB

- Think about a USB

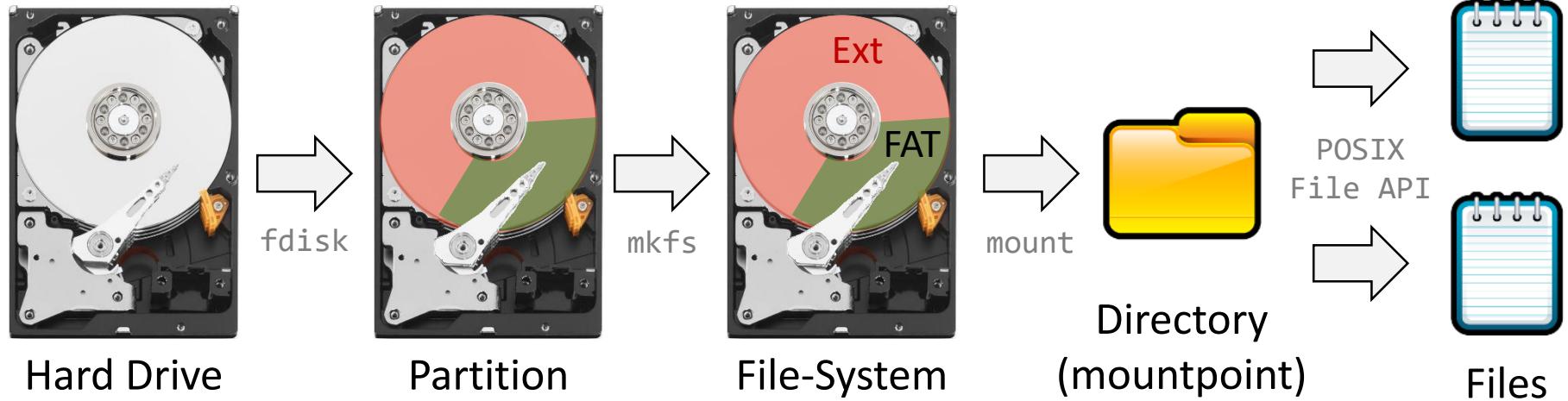


Example: USB

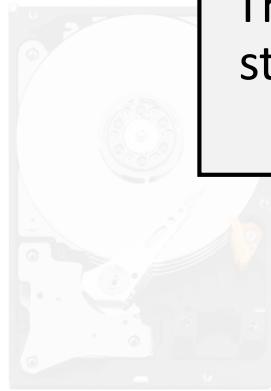
- Think about a USB



[Recap] Linux File-System Workflow



[Recap] Linux File-System Workflow



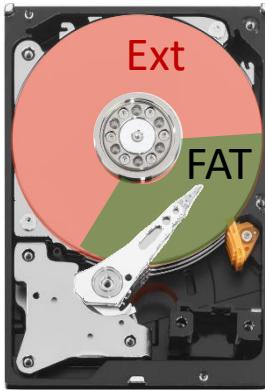
Hard Drive

The “FS build and install step” **initializes** the disk layout of File-System

fdisk

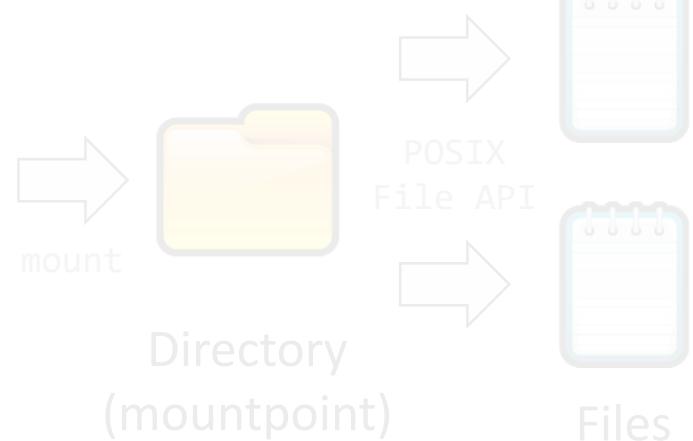


Partition



File-System

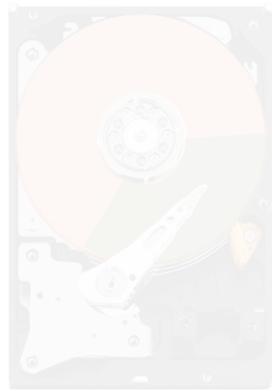
mkfs



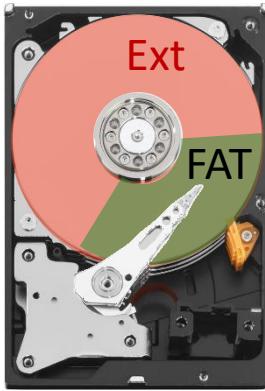
[Recap] Linux File-System Workflow



Hard Drive



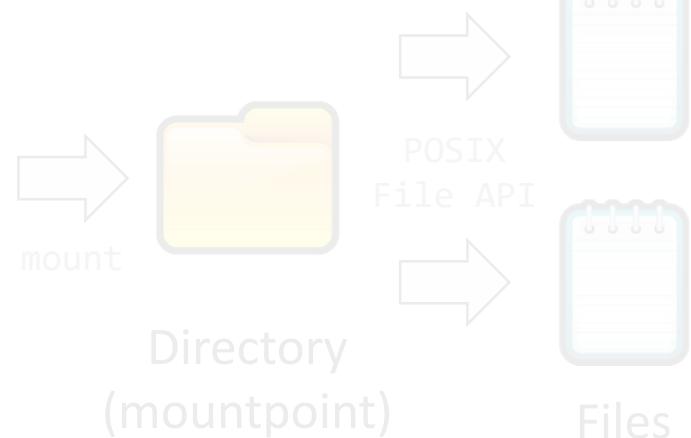
Partition



File-System



FS (FAT, EXT, VFS, etc.) have its own disk layout. Let's check how they look like



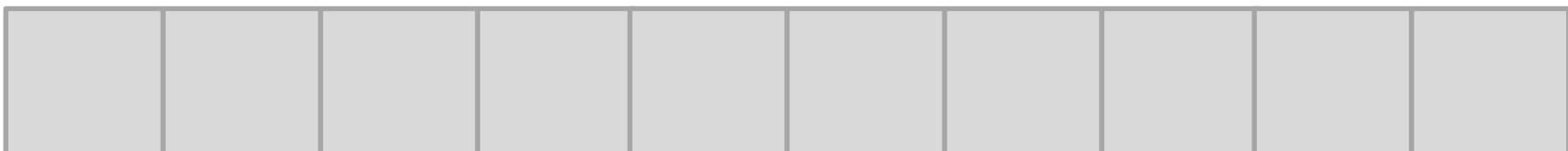
On-Disk Layout #0: Basics

Simple Example

Here is a simple disk. Each gray box means a “disk block”, which can be one or more sectors.



0 1 2 3 4 5 6 7 8 9

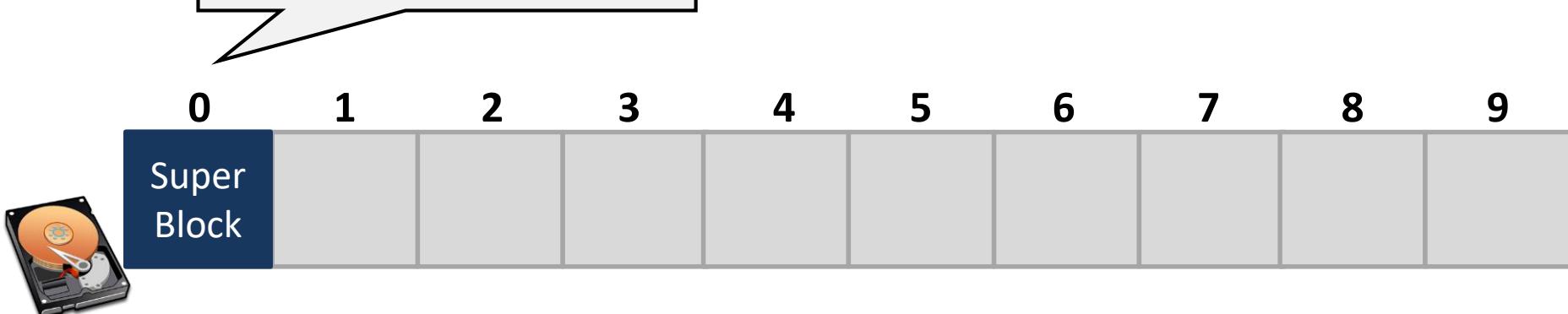


Simple Example

When you build and install FS with `mkfs` command, **superblock*** is created on the first block

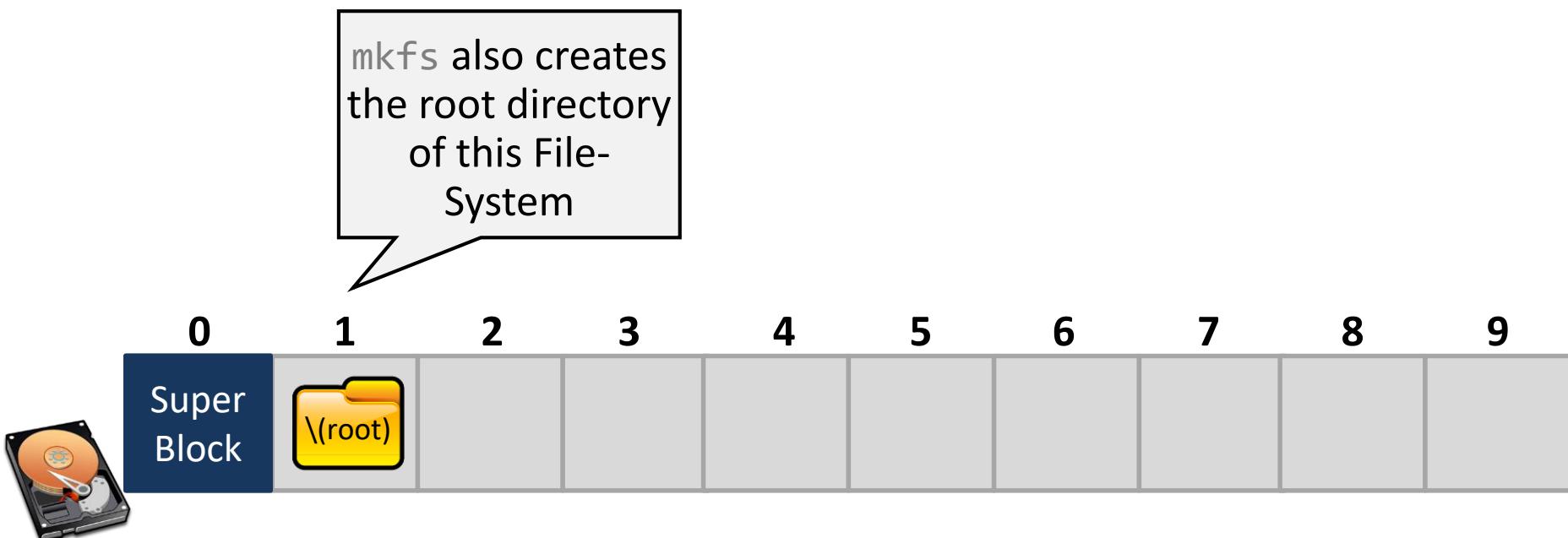
* **Metadata of File-System itself**

- Version of FS
- Number of blocks in FS
- Counts of maximum # of files



Simple Example

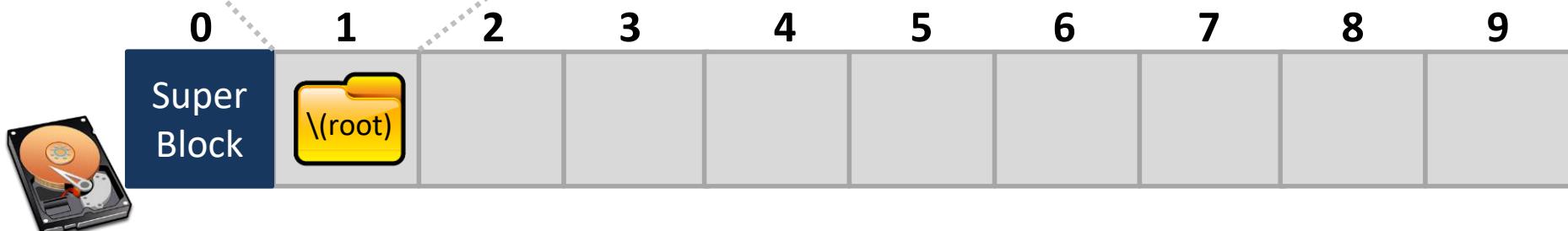
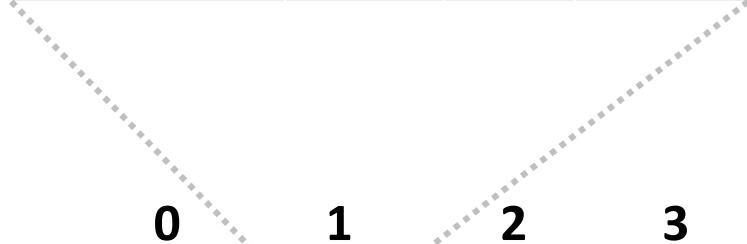
mkfs also creates
the root directory
of this File-
System



Simple Example

Name	Index	Dir?	Perms

Directory file contains a list of **entries inside the directory**, and some **metadata** for each entry

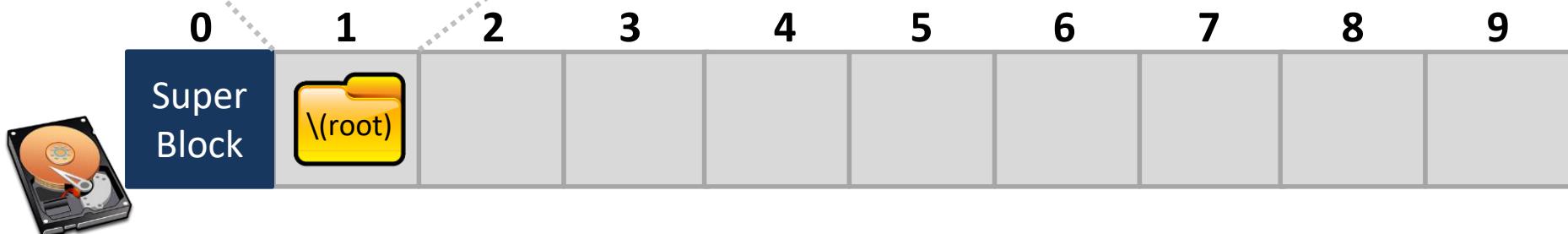


Simple Example

Metadata

Name	Index	Dir?	Perms

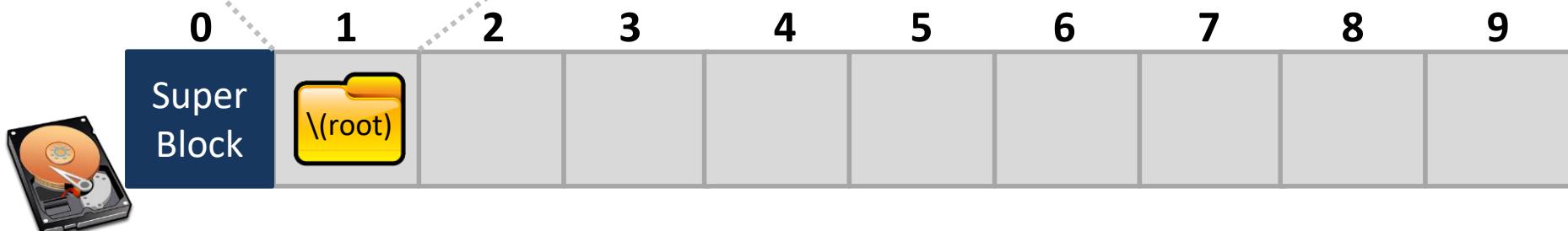
There are many metadata such as created time, modified time, accessed time, but in this simple example, we only highlight disk block index, directory flag, and permissions



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx

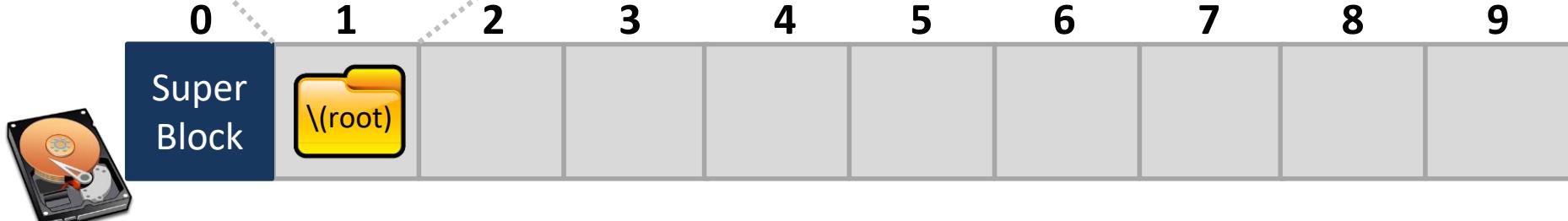
By default, directories contain at least two entries “.” for self pointer and “..” for the parents directory



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx

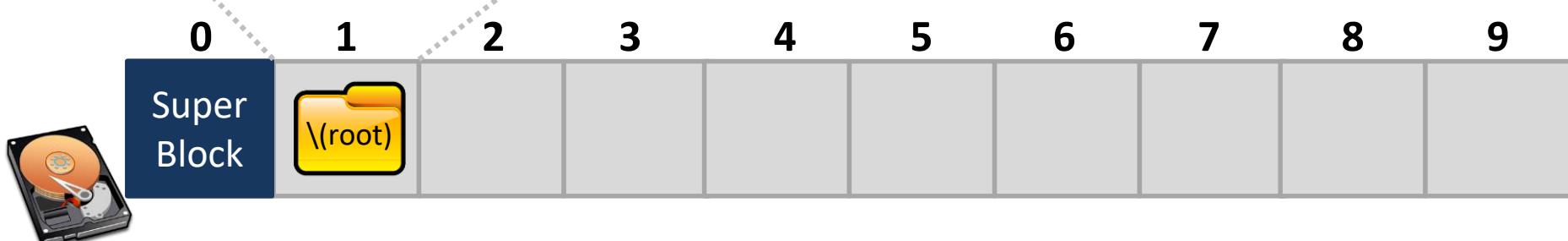
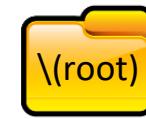
For my self (root),
* I reside on the block 1
* I am a directory
* I am readable,
writable, and executable



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx

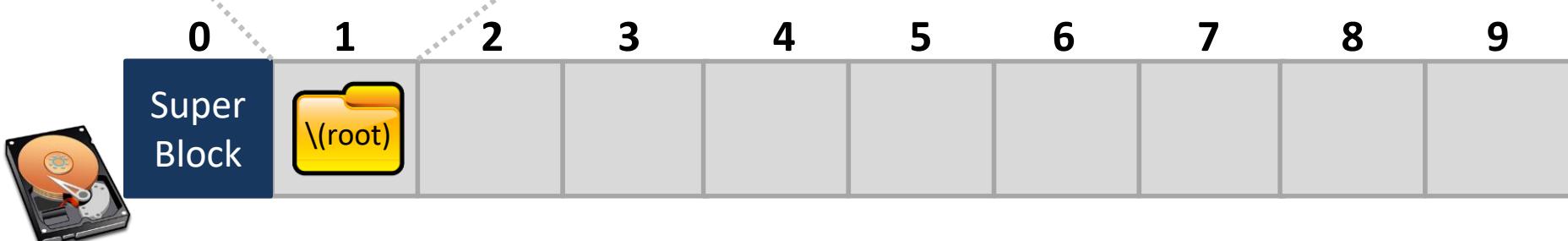
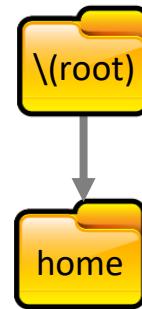
This is our system's directory tree



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx

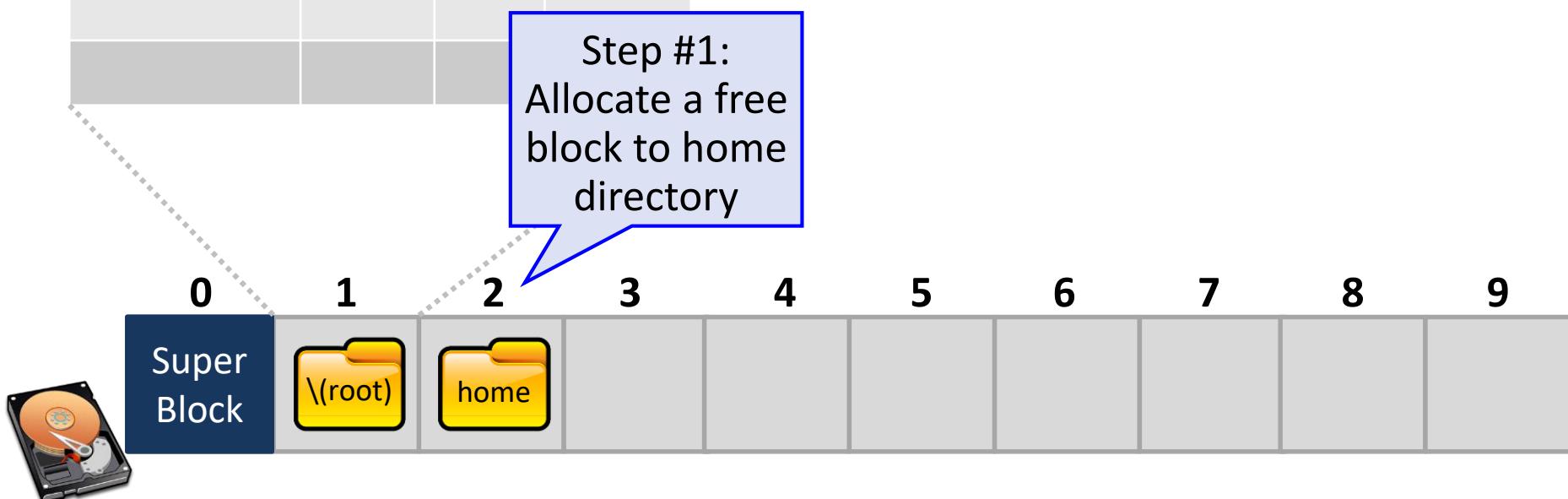
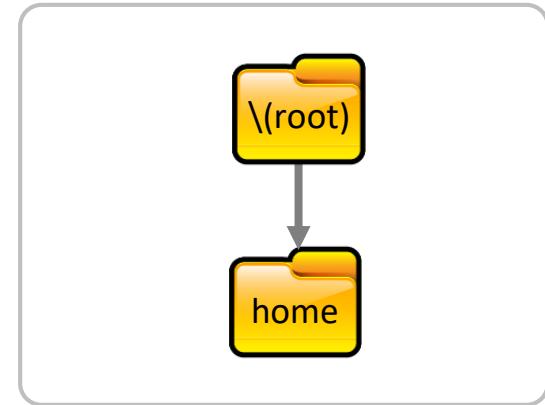
What if we create
a directory,
named home?



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx

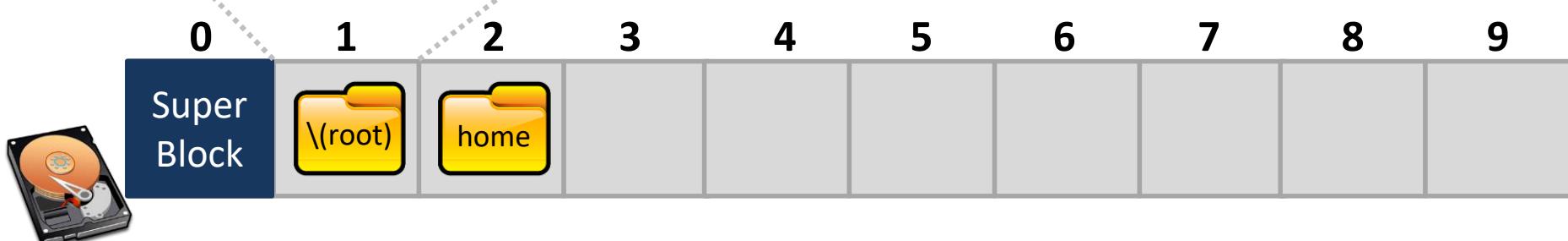
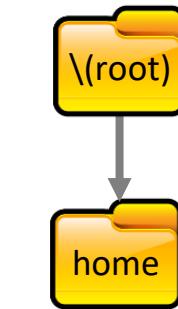
Step #1:
Allocate a free
block to home
directory



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx
home	2	Y	rmw

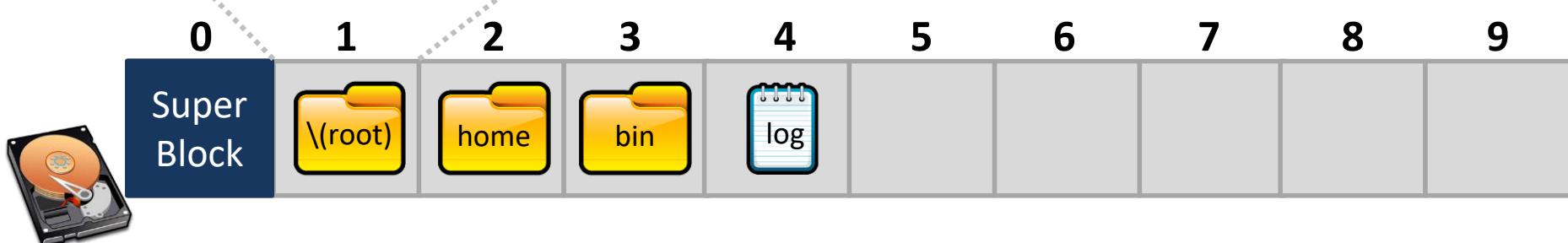
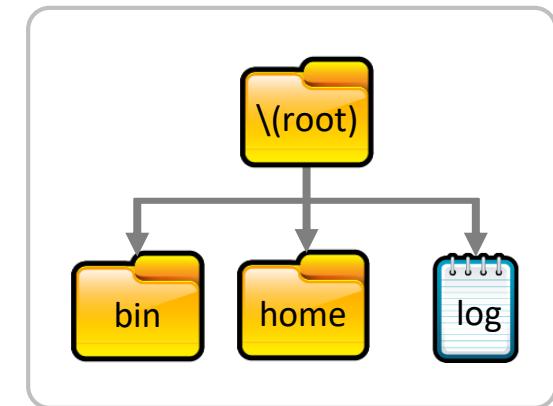
Step #2:
Update root
directory



Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx
home	2	Y	rmw
bin	3	Y	rmw
log	4	N	r

Do the same
for the another



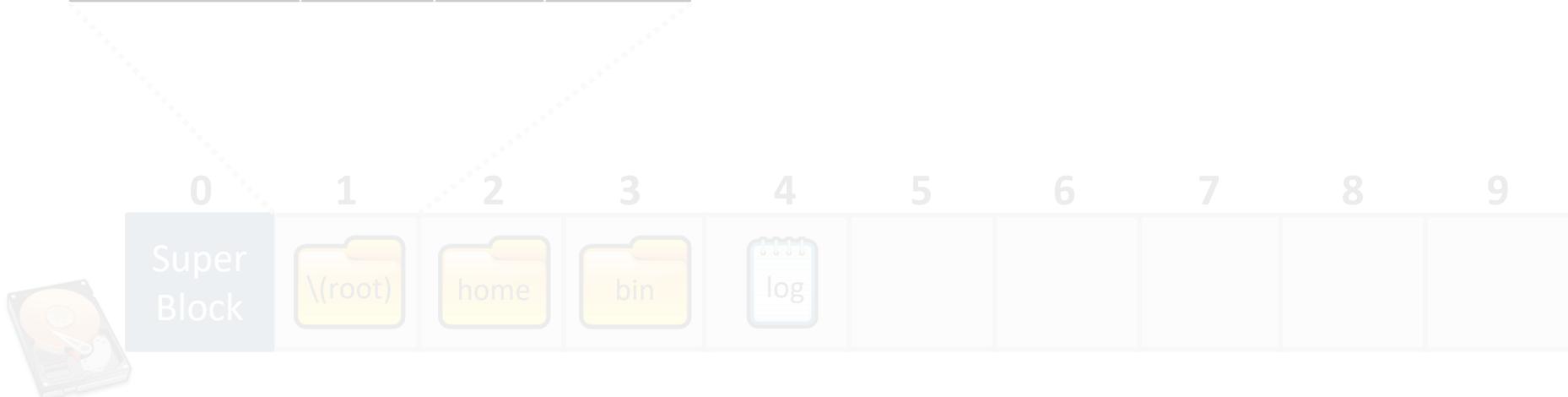
Simple Example

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx
home	2	Y	rmw
bin	3	Y	rmw
log	4	N	r

BTW, a directory (file) will have many entries. How can you **encode*** the entries?



*to know which entry is for our target file





Directory File Implementation

Unordered List of Entries

Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rmw
home	2	Y	rmw
bin	3	Y	rmw
log	4	N	r

Good: O(1) to **add** new entries
(Just append to the directory file)

Bad: O(n) to **search** for an entry

Sorted List of Entries

Name	Index	Dir?	Perms
.	1	Y	rmw
..	-	-	-
bin	3	Y	rmw
home	2	Y	rmw
log	4	N	r

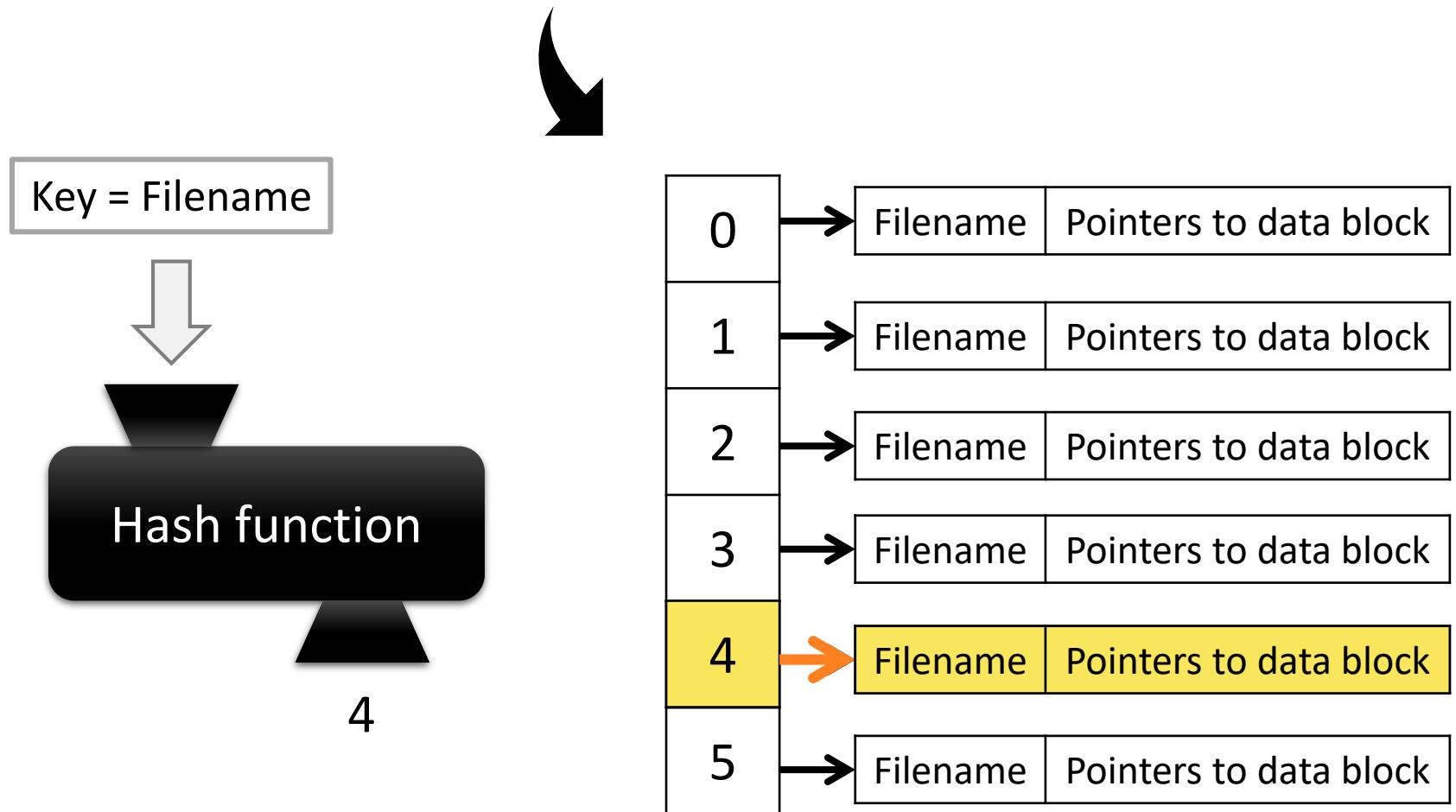


Good: O(log n) to **search** for an entry (e.g., binary search)

Bad: O(n) to add new entries (& Entire file has to be **rewritten**)

Directory File Implementation

- Other alternatives: hash tables, B-trees

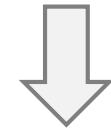




Directory File Implementation

- Other alternatives

Key = Filename

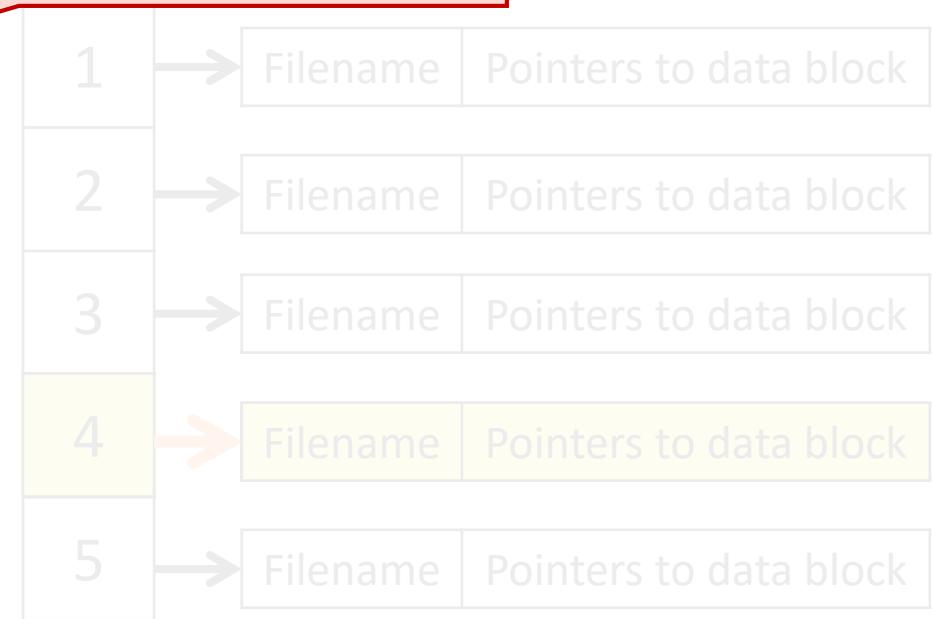


Hash function

4

Implementing directory files is not that easy peasy lemon squeezy.

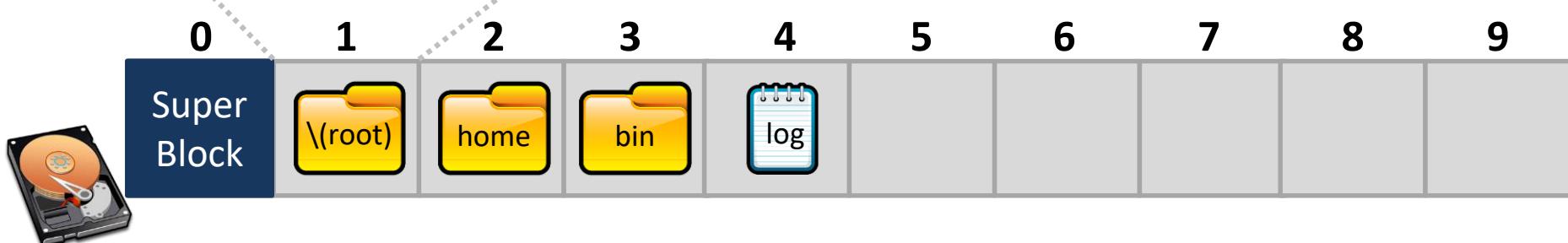
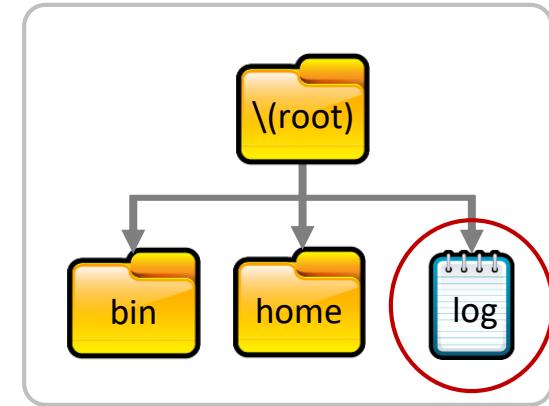
For example, each file can have variable length, and can be updated in anytime while your data block has a fixed size.



Simple Example

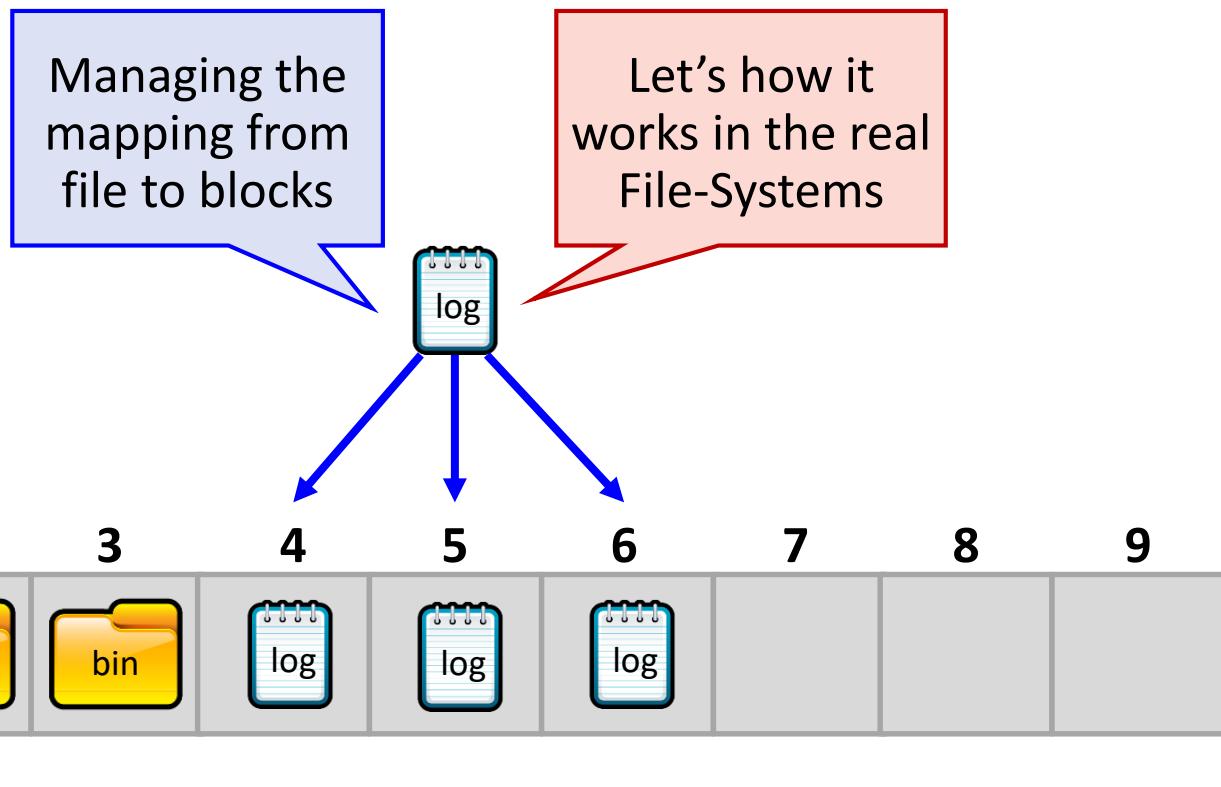
Name	Index	Dir?	Perms
..	-	-	-
.	1	Y	rwx
home	2	Y	rmw
bin	3	Y	rmw
log	4	N	r

What if the “log” file get bigger?





File Metadata Implementation



On-Disk Layout #1: File Allocation Table

File Allocation Tables (FAT)

- Simple File-System popularized by MS-DOS

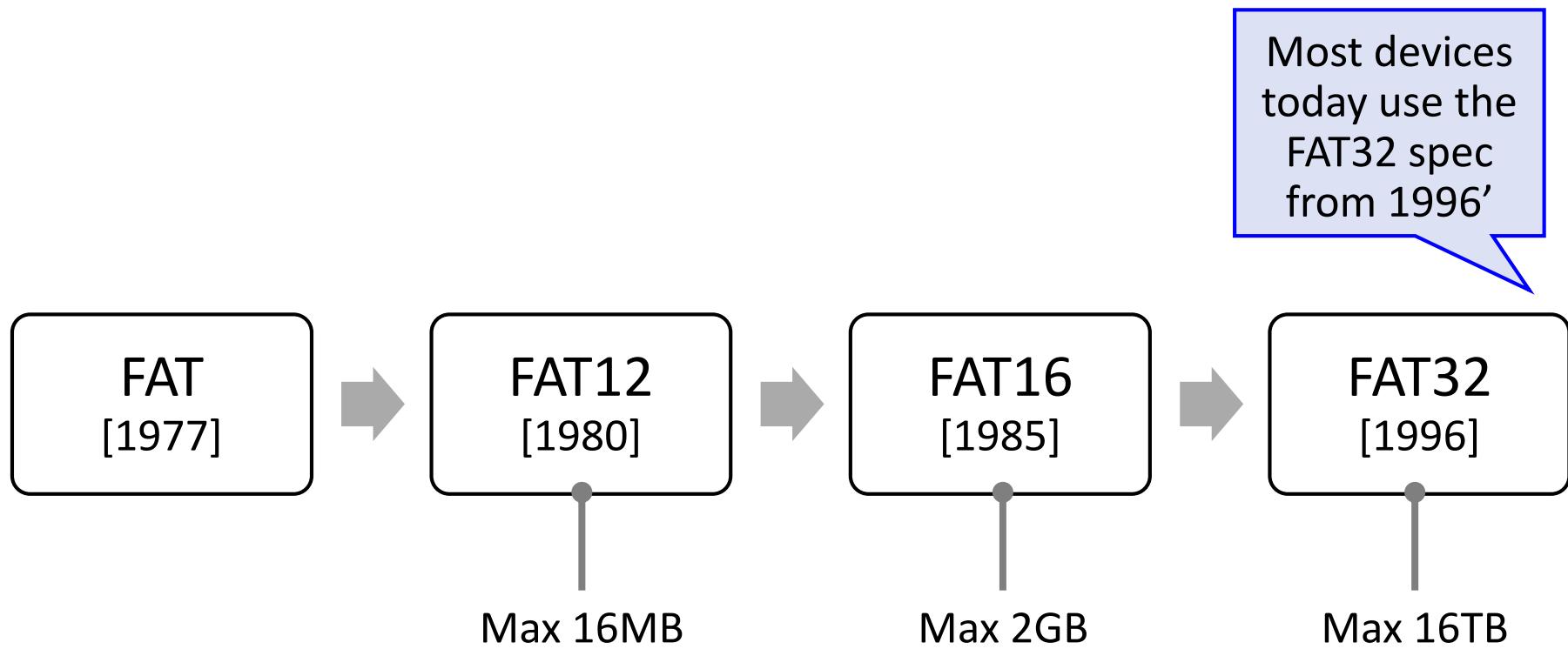


First
introduced
in 1977

FAT
[1977]

File Allocation Tables (FAT)

- Simple File-System popularized by MS-DOS



File Allocation Tables (FAT)

- Still quite popular today

Default format for
USB sticks and
memory cards



Also used for EFI
boot partitions



On-disk Layout of FAT

How does FAT
FS look like?



Super
Block



Superblock of FAT FS

Stores basic information about the file system

- FAT version, location of boot files
- Total number of blocks
- Index of the root directory in the FAT

Super
block



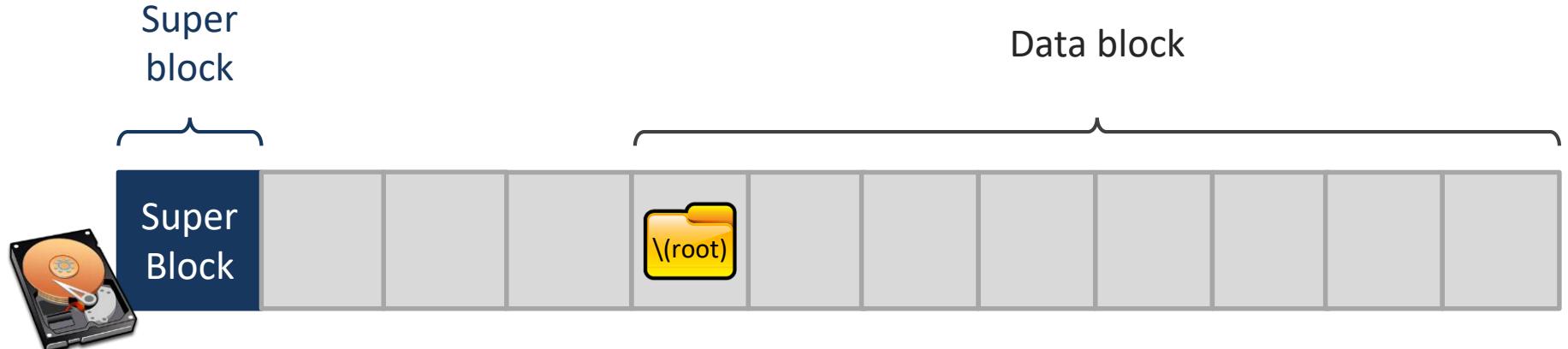
Super
Block



Data Block of FAT FS

※ No difference with basic concept

Store file and directory **data**
- Each block is a fixed size (4KB – 64KB)
- Files may span multiple blocks





File Allocation Table of FAT FS

File allocation table (FAT)

- Marks which blocks are free or in-use
- **Linked-list structure** to manage large files

Super
block

FAT

Disk block



Super
Block





File Allocation Table of FAT FS

Possible values for FAT entries:

- 0: entry is empty
- 1: reserved by the OS
- $1 < N < 0xFFFF$: next block in a chain
- 0xFFFF: end of a chain



Super Block

0xFFFF





File Allocation Table of FAT FS

$\text{Len(FAT)} ==$
Number of clusters
on the disk

Max number of
files/directories is
bounded (decided when
you format the partition)



Super
Block

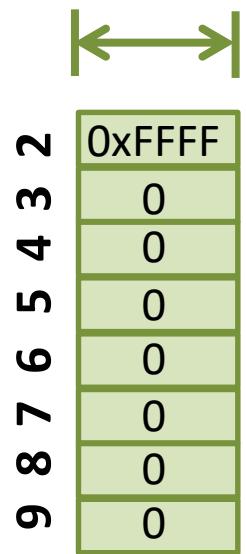
0xFFFF

2 3 4 5 6 7 8 9 2 3 4 5 6 7 8 9

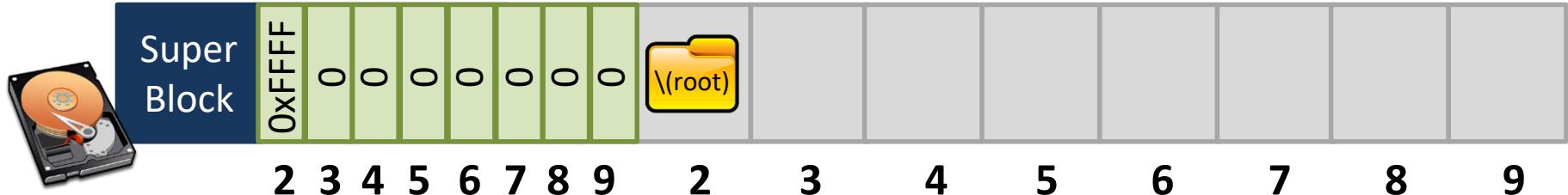


File Allocation Table of FAT FS

The FAT version roughly corresponds to the size in bits of each FAT entry

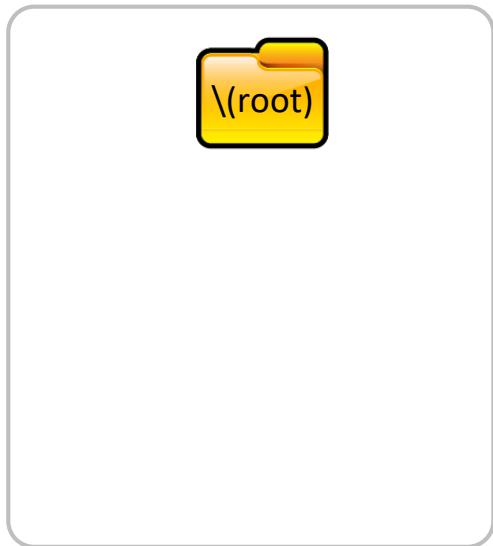


For example, **FAT16** →
each FAT entry is 16 bits
(∴ more bits → larger disks are supported)

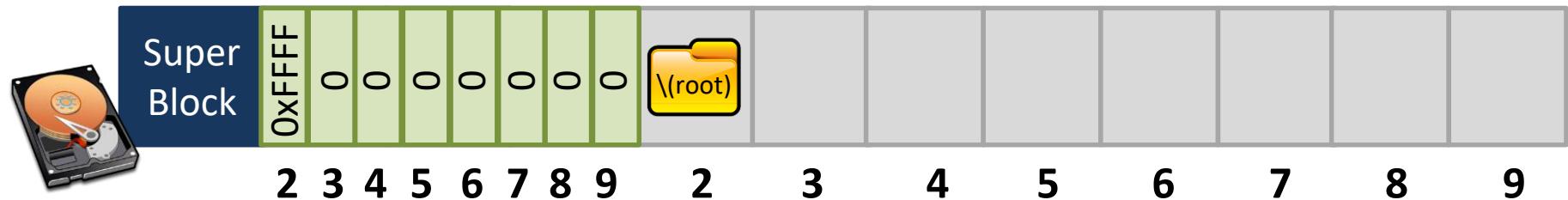




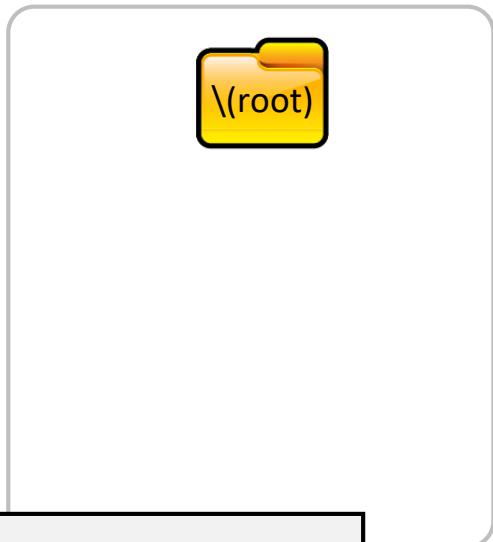
Example of FAT FS



This is our system's directory tree. Where is the root directory?



Example of FAT FS



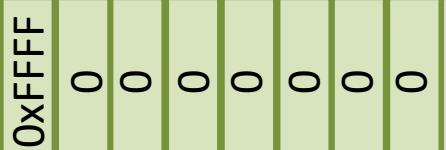
Block 2

Name	Index	Dir?	Perms
.	2	Y	rwx

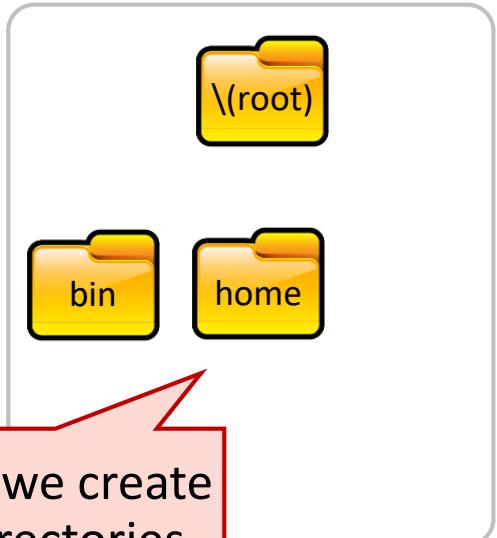
The root directory index is recorded in the super block (==2)



Super Block



Example of FAT FS

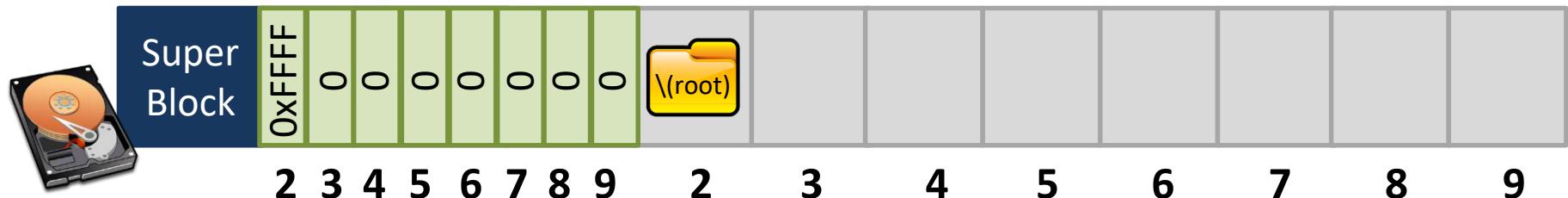


What if we create
two directories
under the root?

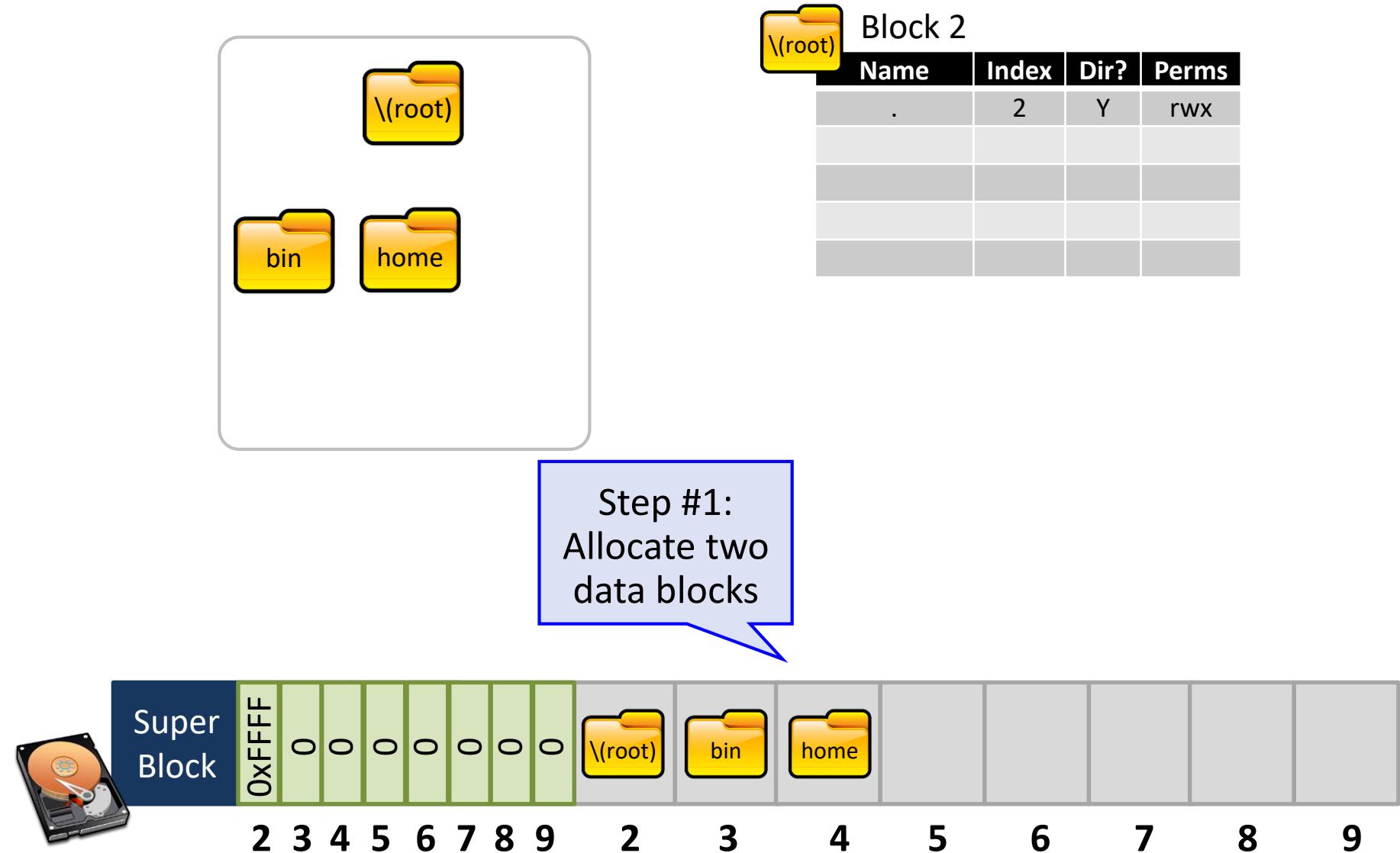


\(root) Block 2

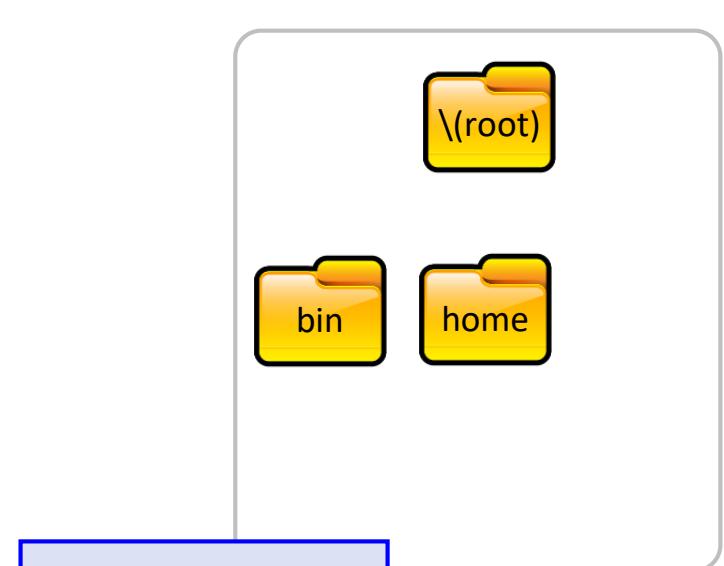
Name	Index	Dir?	Perms
.	2	Y	rwx



Example of FAT FS



Example of FAT FS

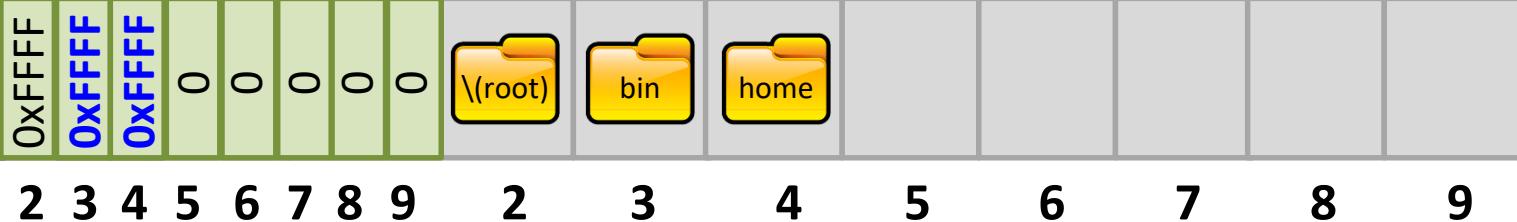


Block 2				
\(root)	Name	Index	Dir?	Perms
.	2	Y	rwx	

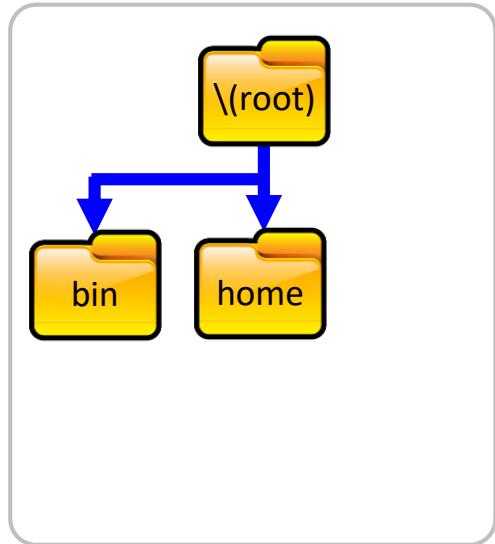
Step #2:
Update the
corresponding
FAT entries



Super Block

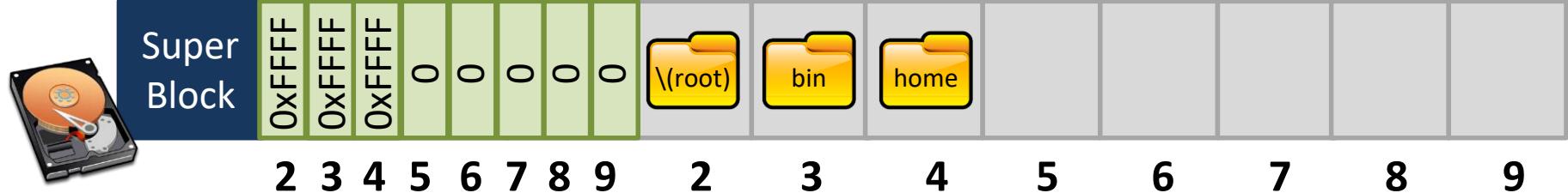


Example of FAT FS

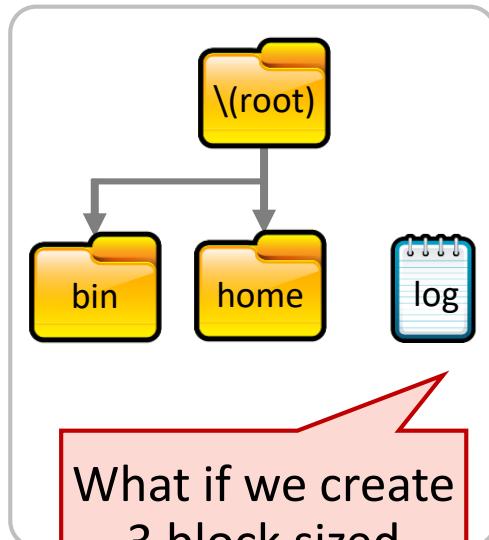


Name	Index	Dir?	Perms
.	2	Y	rwx
bin	3	Y	rmw
home	4	Y	rmw

Step #3:
Update parent
directory file



Example of FAT FS



What if we create
3 block sized
“log” file?

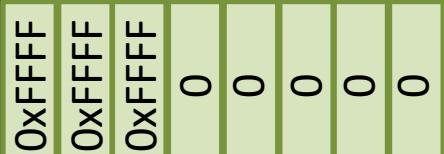


Block 2

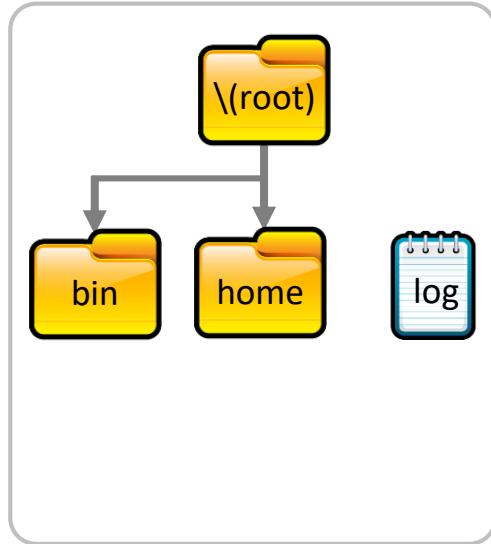
Name	Index	Dir?	Perms
.	2	Y	rwx
bin	3	Y	rmw
home	4	Y	rmw



Super Block



Example of FAT FS

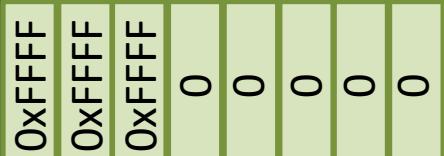


Name	Index	Dir?	Perms
.	2	Y	rwx
bin	3	Y	rmw
home	4	Y	rmw

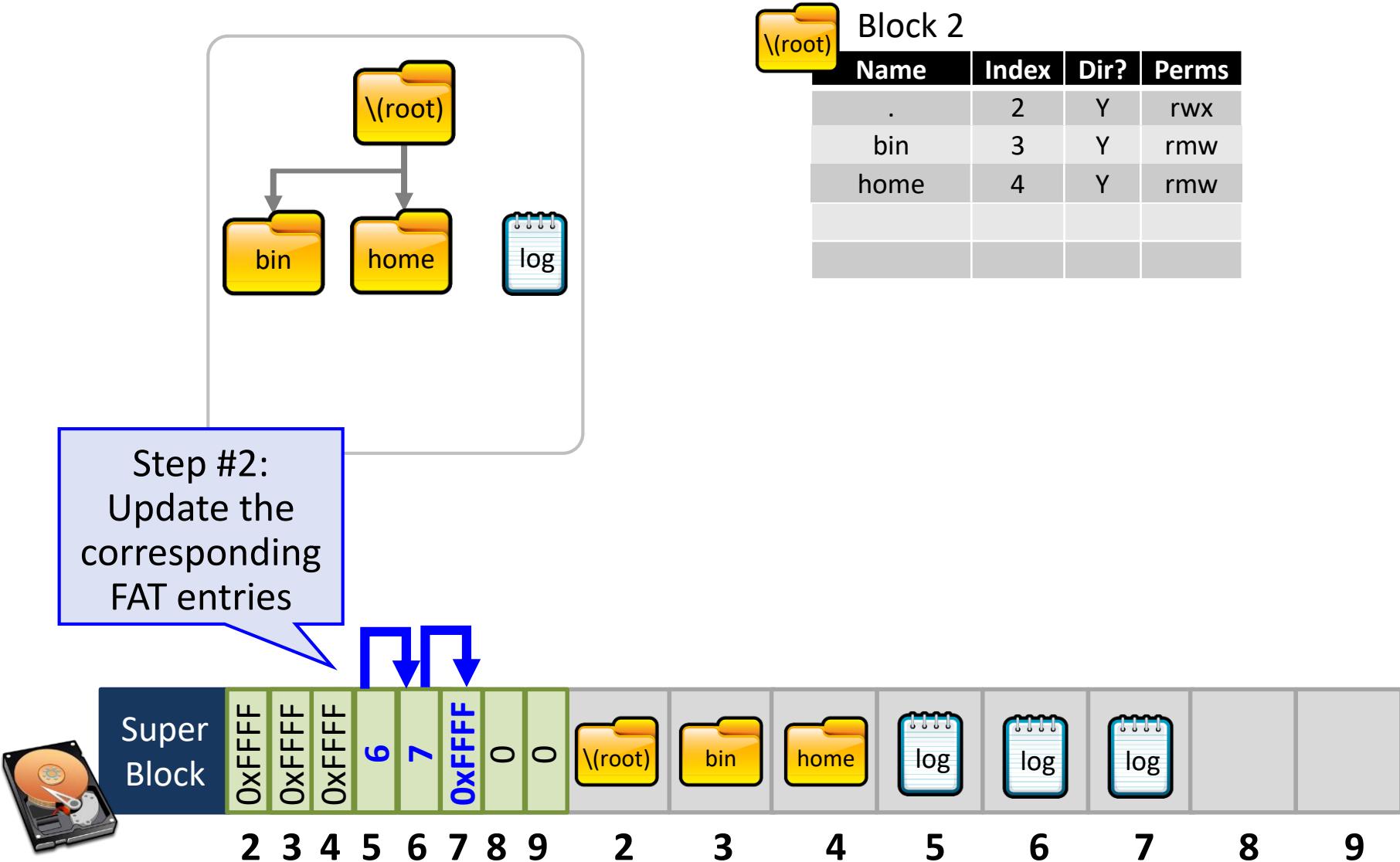
Step #1:
Allocate three
data blocks



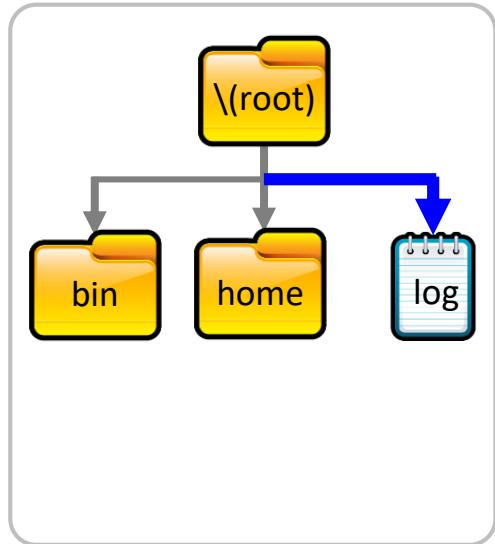
Super Block



Example of FAT FS

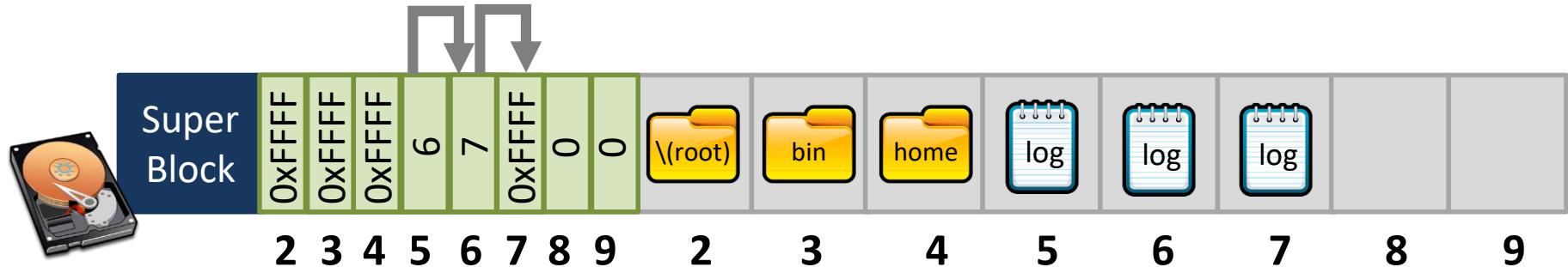


Example of FAT FS

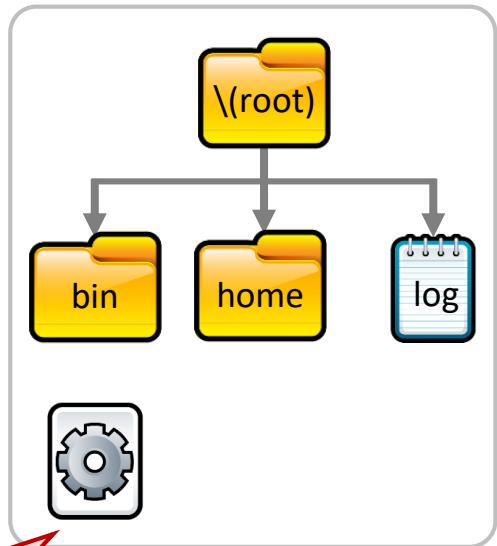


Name	Index	Dir?	Perms
.	2	Y	rwx
bin	3	Y	rmw
home	4	Y	rmw
log	5	N	r

Step #3:
Update parent
directory file



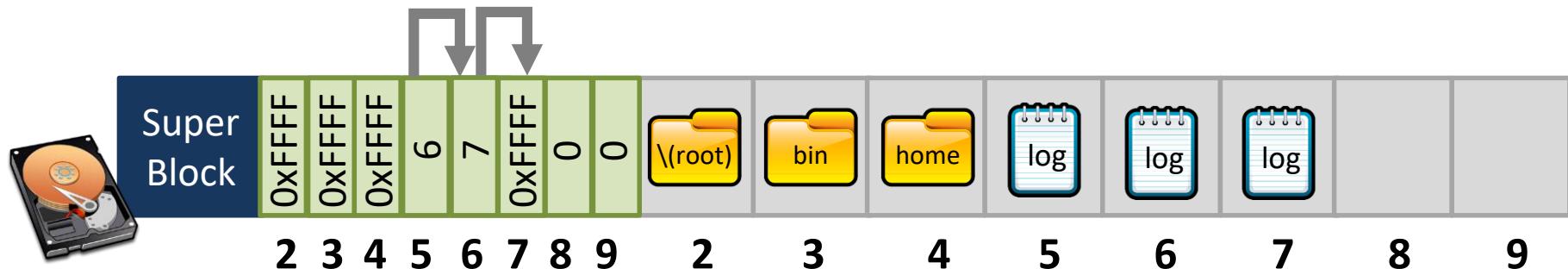
Example of FAT FS



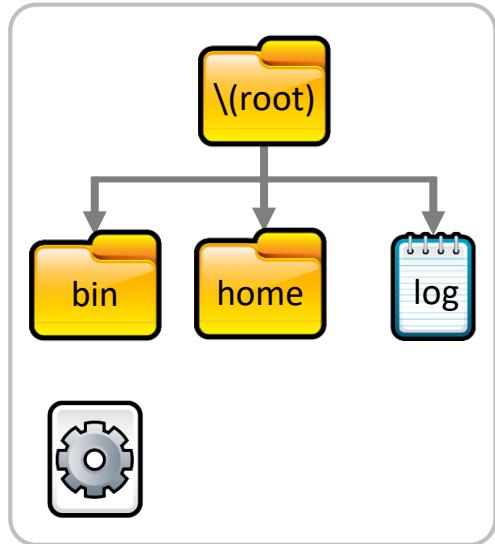
2 block sized
“program”

\(root) Block 2

Name	Index	Dir?	Perms
.	2	Y	rwx
bin	3	Y	rmw
home	4	Y	rmw
log	5	N	r



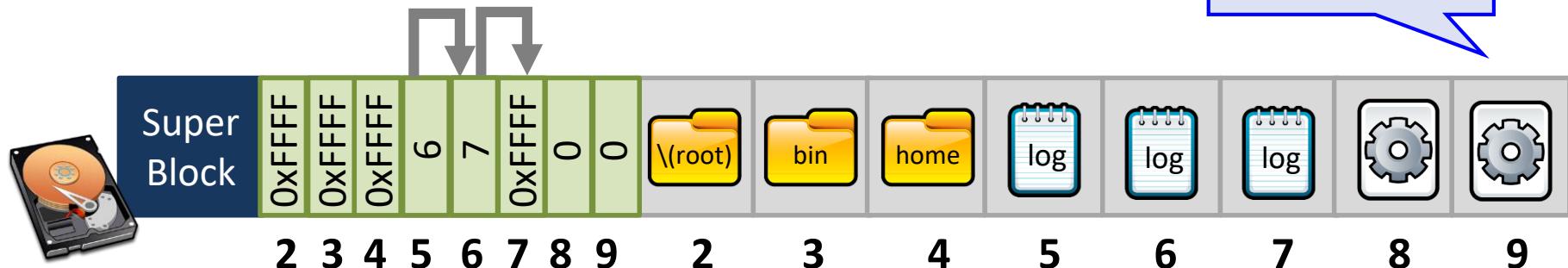
Example of FAT FS



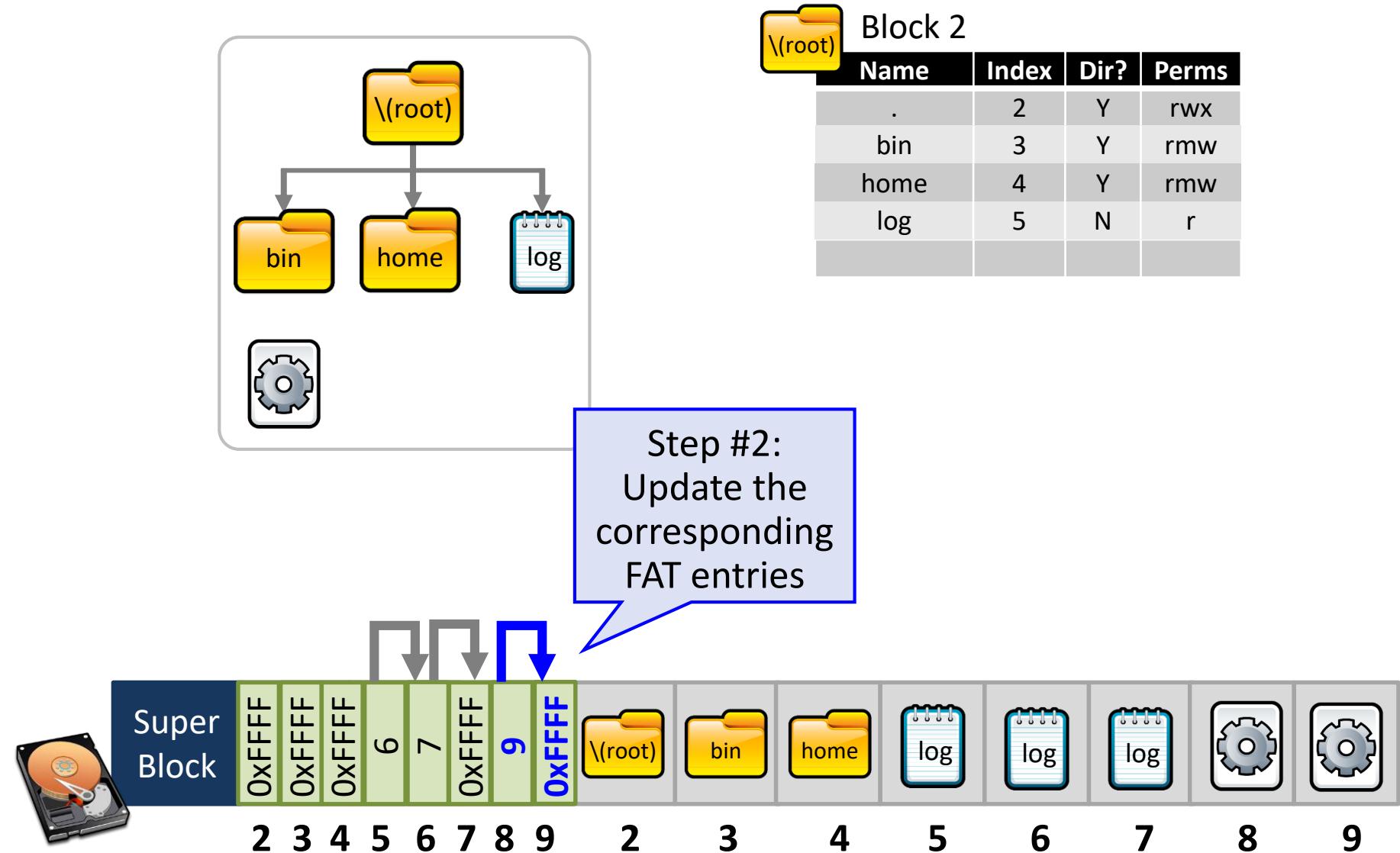
Block 2

Name	Index	Dir?	Perms
.	2	Y	rwx
bin	3	Y	rmw
home	4	Y	rmw
log	5	N	r

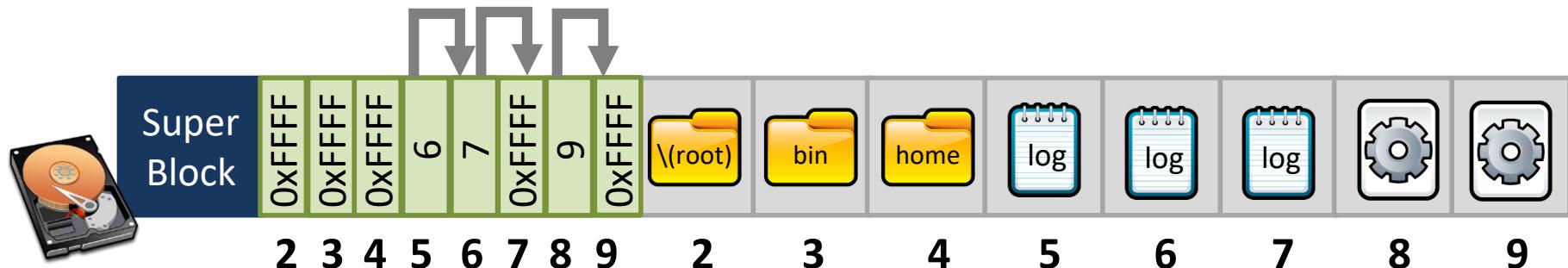
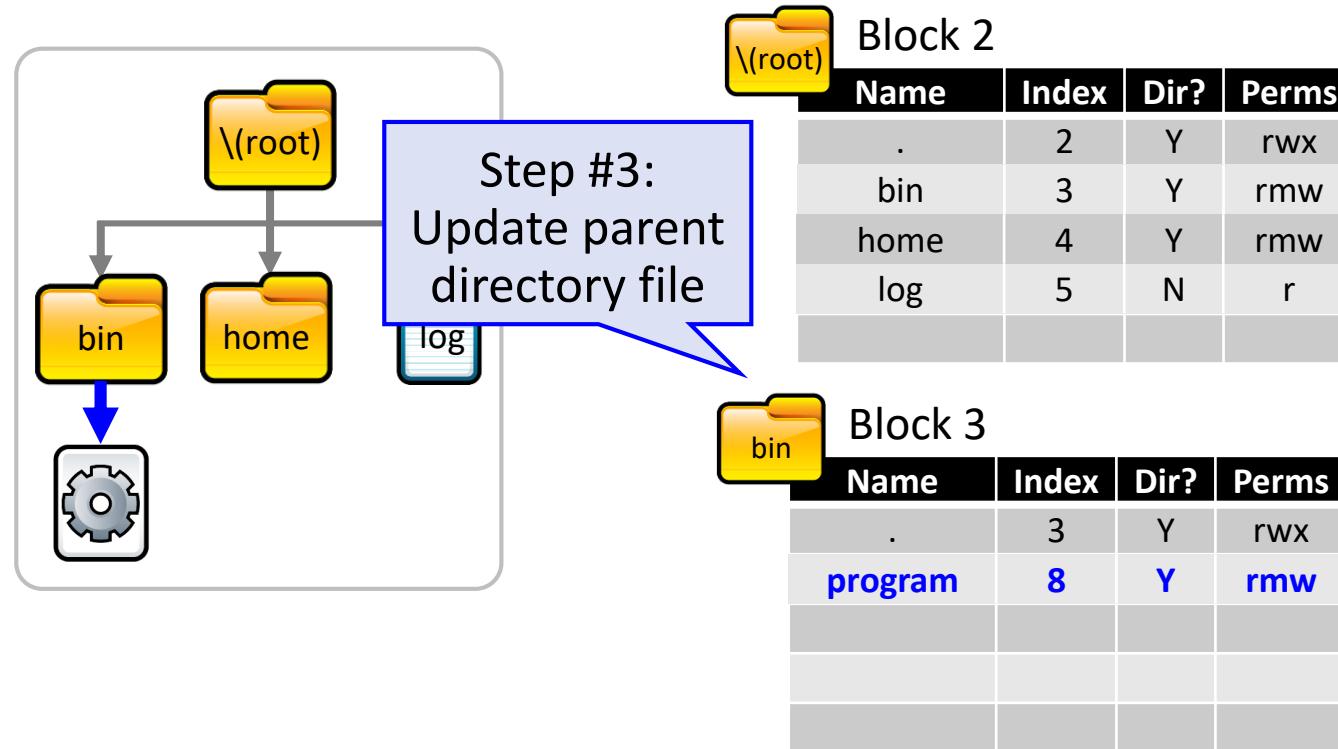
Step #1:
Allocate two data blocks



Example of FAT FS



Example of FAT FS



Limitations of FAT

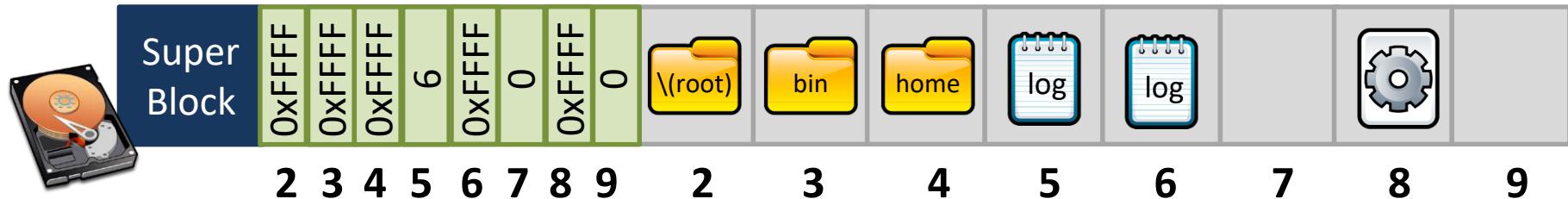
FAT is **good**, as it supports hierarchical tree of directories and variable length files

However, there are several **drawbacks** ☹

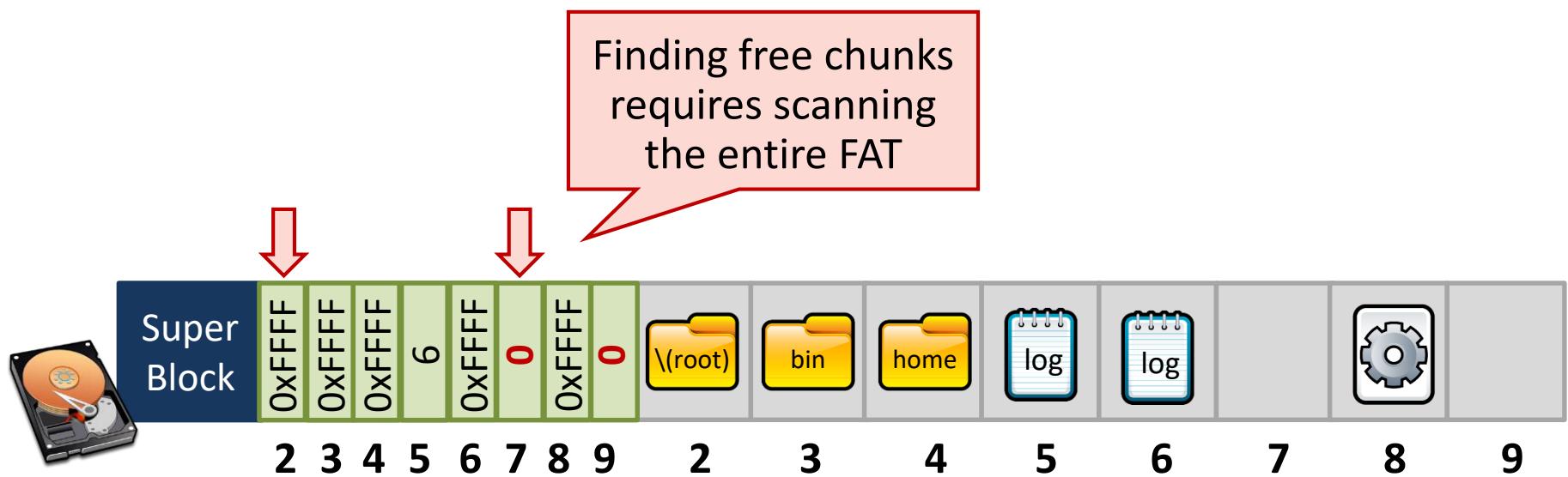


#1: Overheads of Block Allocation

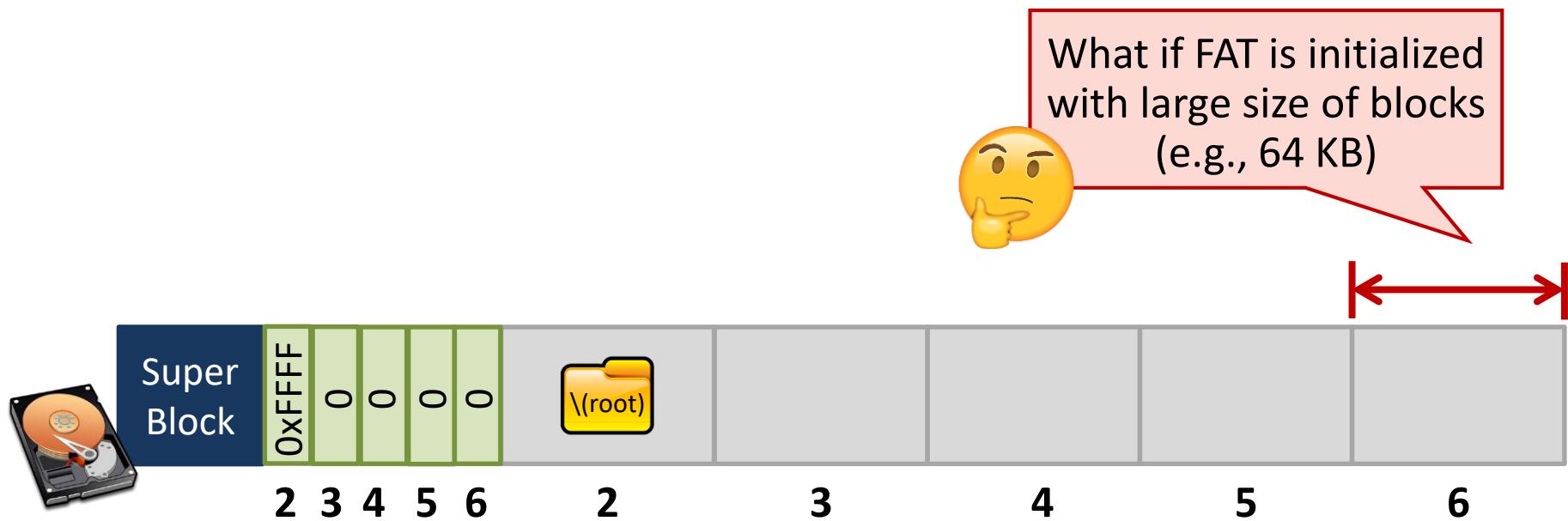
How could we
find free blocks in
this disk?



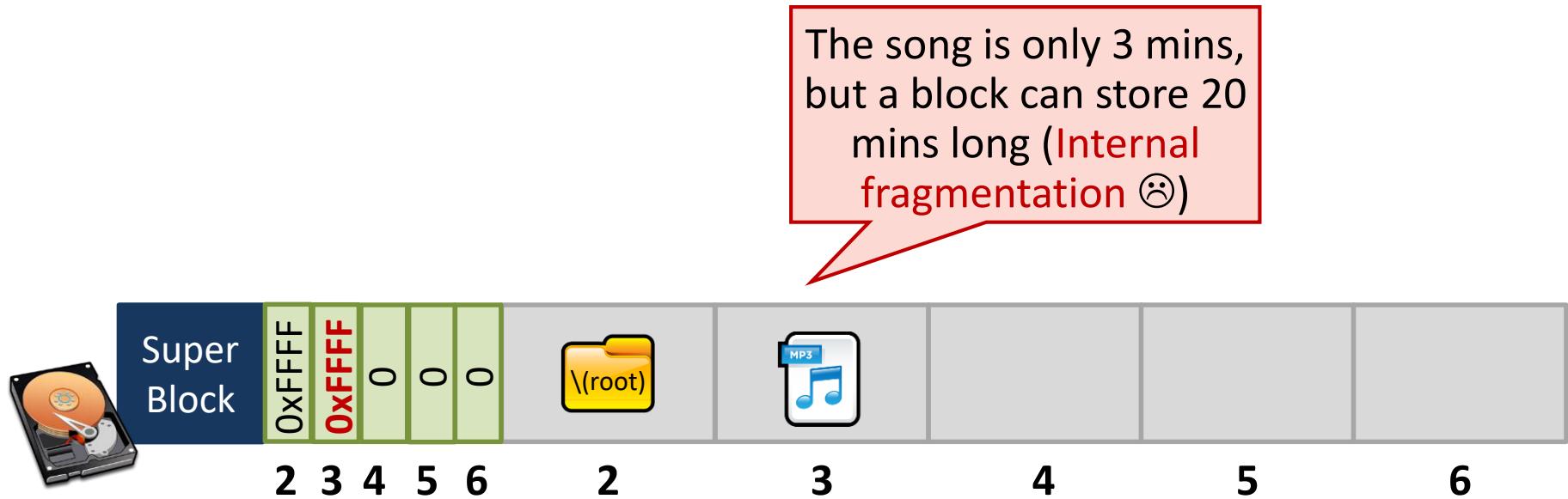
#1: Overheads of Block Allocation



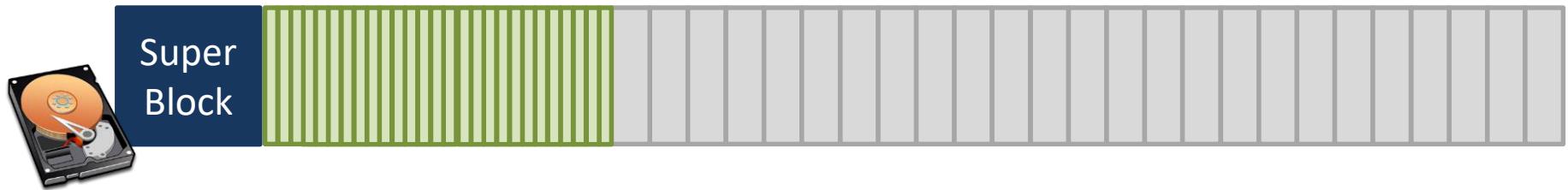
#2: Internal Fragmentation



#2: Internal Fragmentation



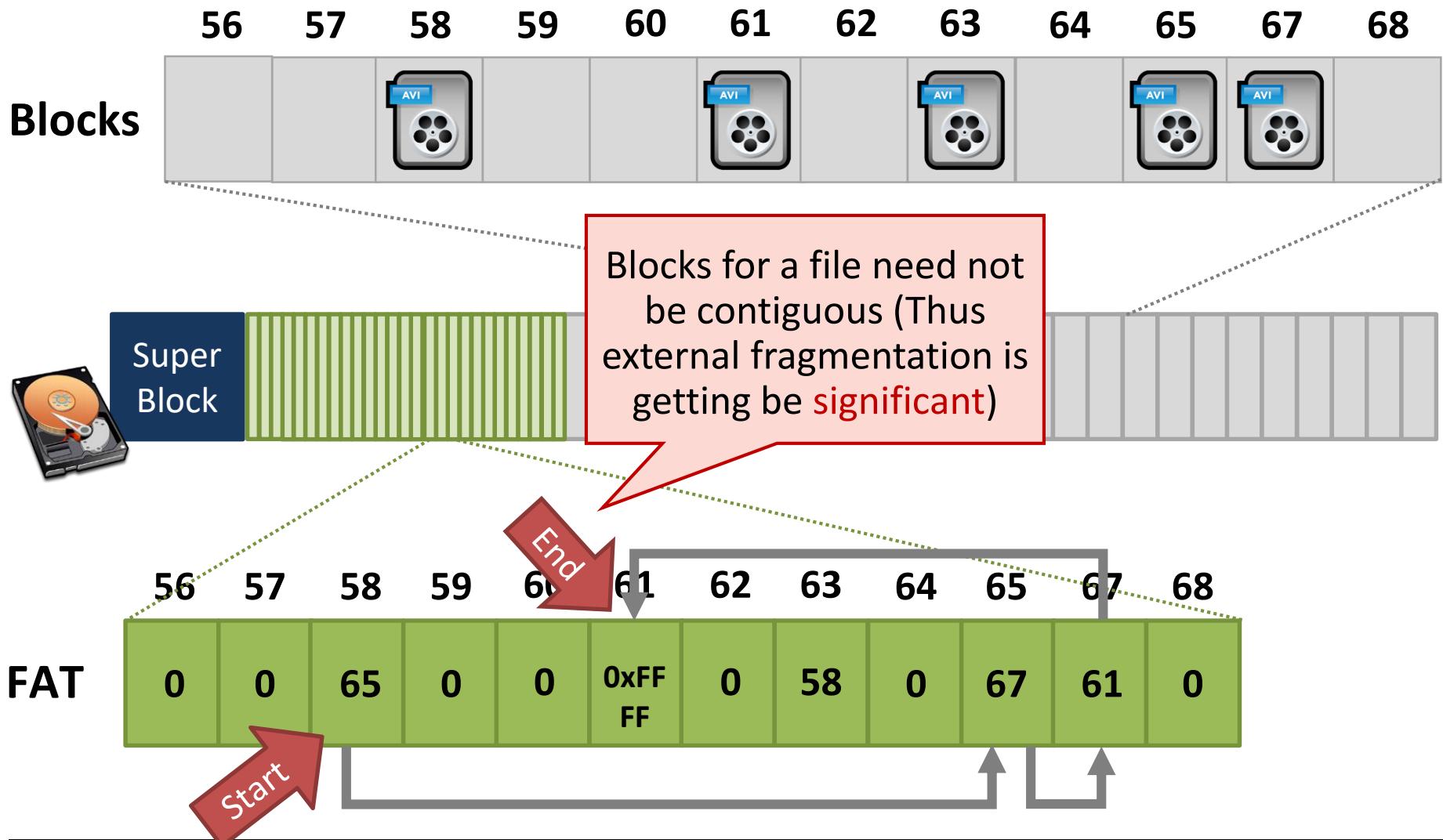
#3: External Fragmentation



Then what if FAT is initialized with small size of blocks (e.g., 4 KB)



#3: External Fragmentation



#4: Overhead of Reads

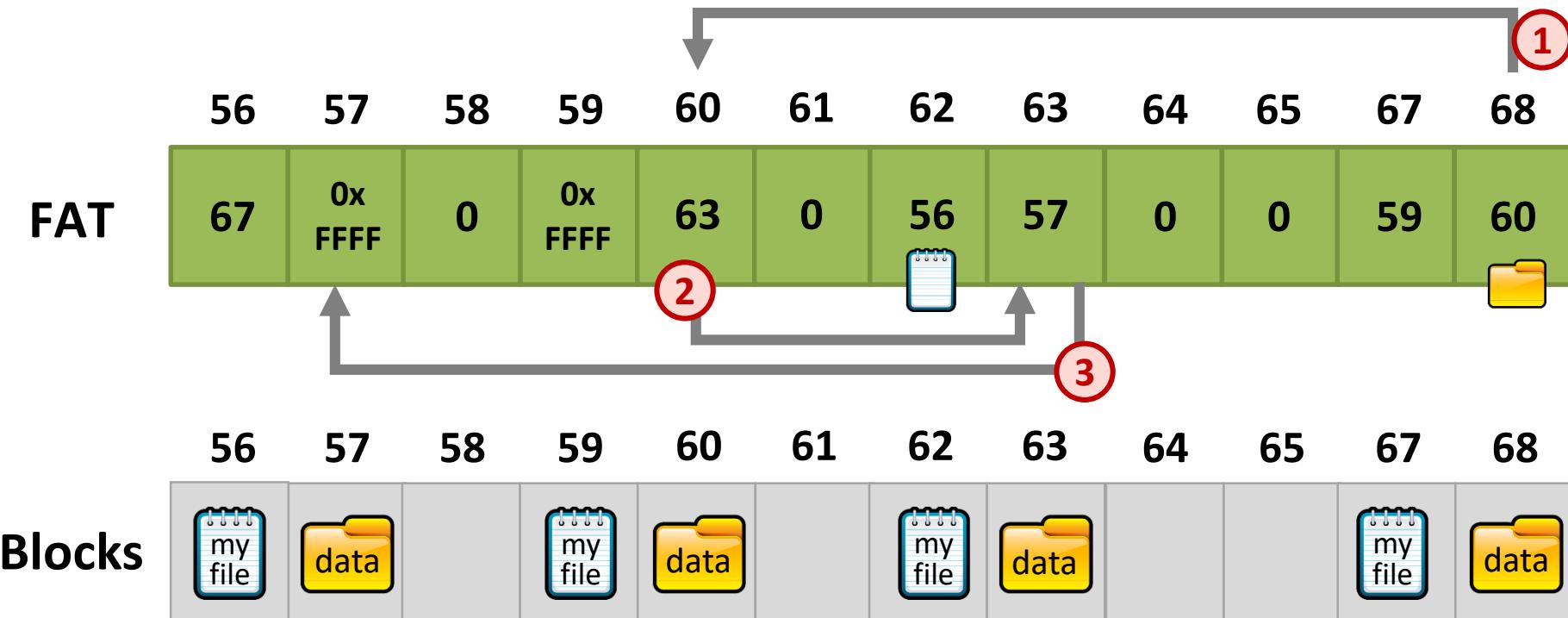
```
int fd = open("/data/my_file.txt", "r");  
int r = read(fd, buffer, 1024 * 4 * 4); // 4 4KB blocks
```

Consider the
following code

#4: Overhead of Reads

```
int fd = open("/data/my_file.txt", "r");  
int r = read(fd, buffer, 1024 * 4 * 4); // 4
```

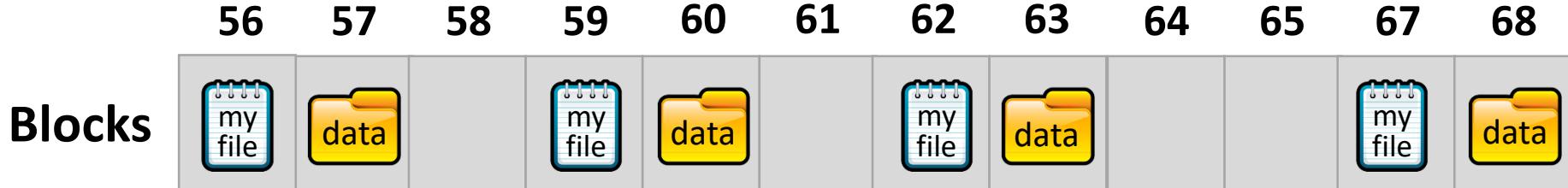
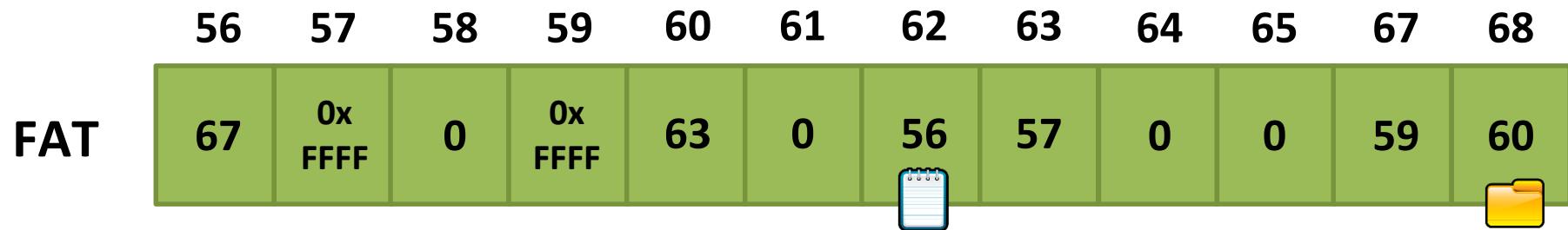
Check data directory file



#4: Overhead of Reads

```
int fd = open("/data/my_file.txt", "r");  
int r = read(fd, buf, 4 * 4); // 4 4KB blocks
```

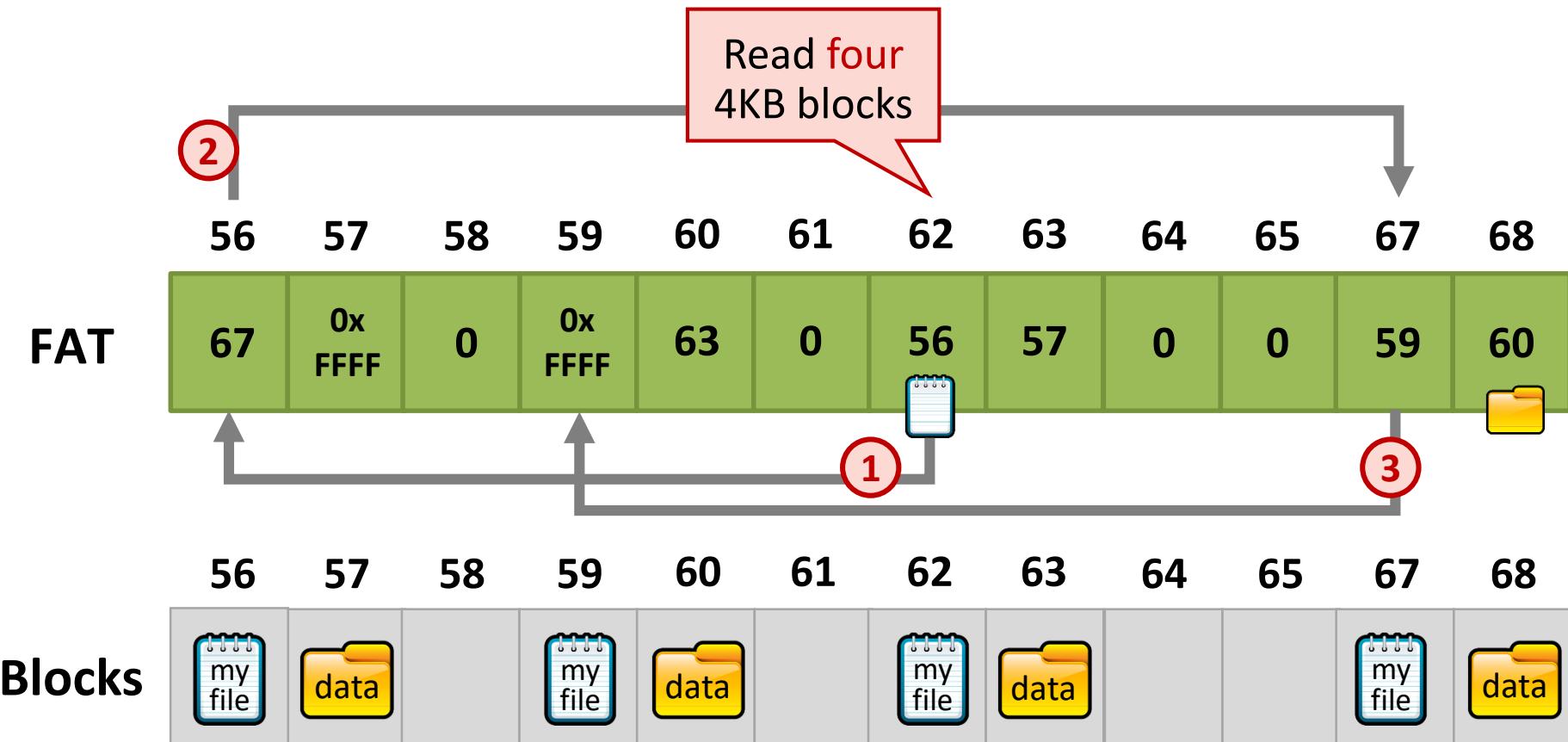
From the directory (data), we can know that my_file.txt exists at block 62



#4: Overhead of Reads

```
int fd = open("/data/my_file.txt", "r");
```

```
int r = read(fd, buffer, 1024 * 4 * 4); // 4 4KB blocks
```

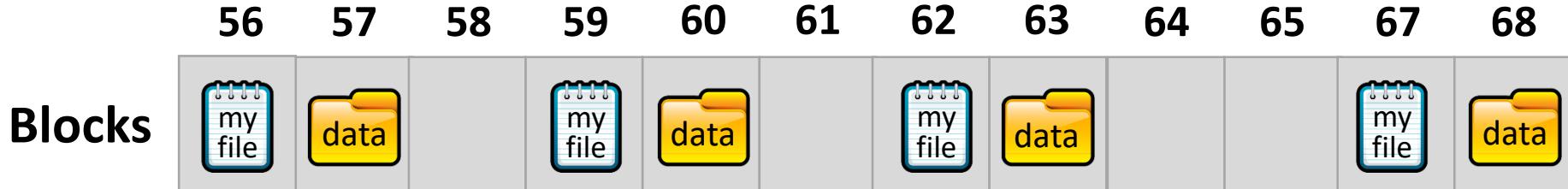
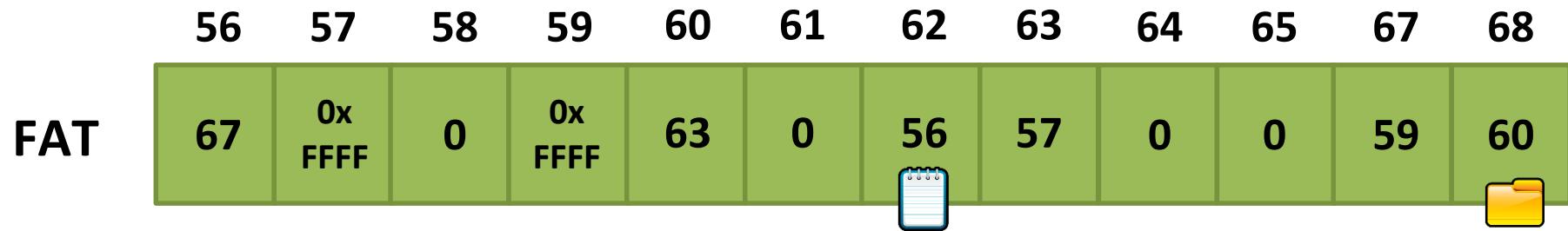


#4: Overhead of Reads

```
int fd = open("/data/my_file.txt", "r");
```

```
int r = read(fd, buffer, 1024 * 4 * 4);
```

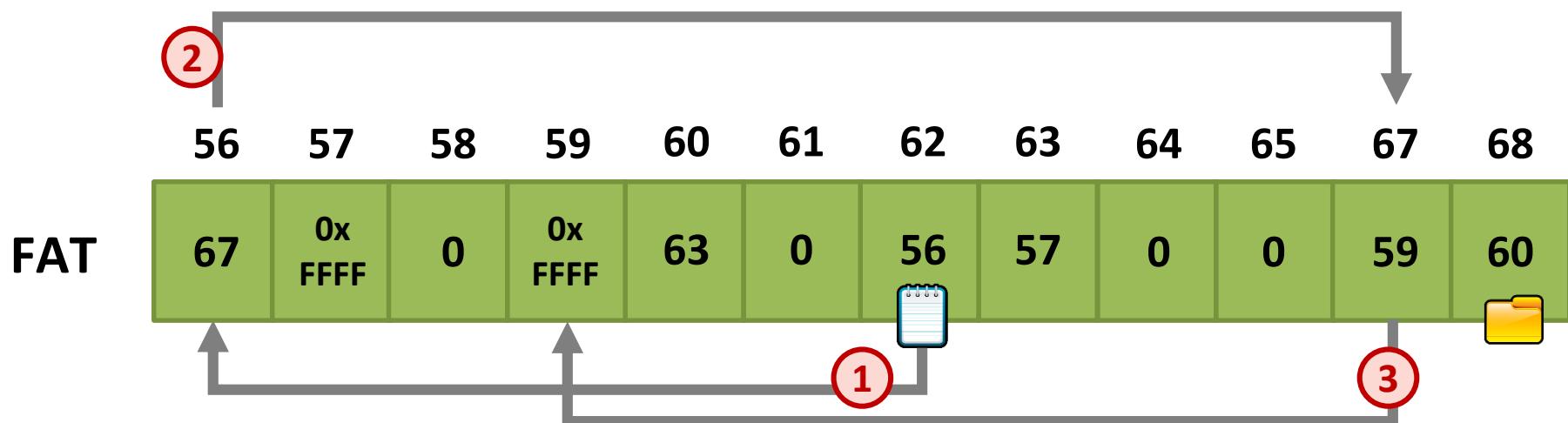
FAT may have **very low spatial locality**, thus there are a lot of random seeking



Linked-list based File

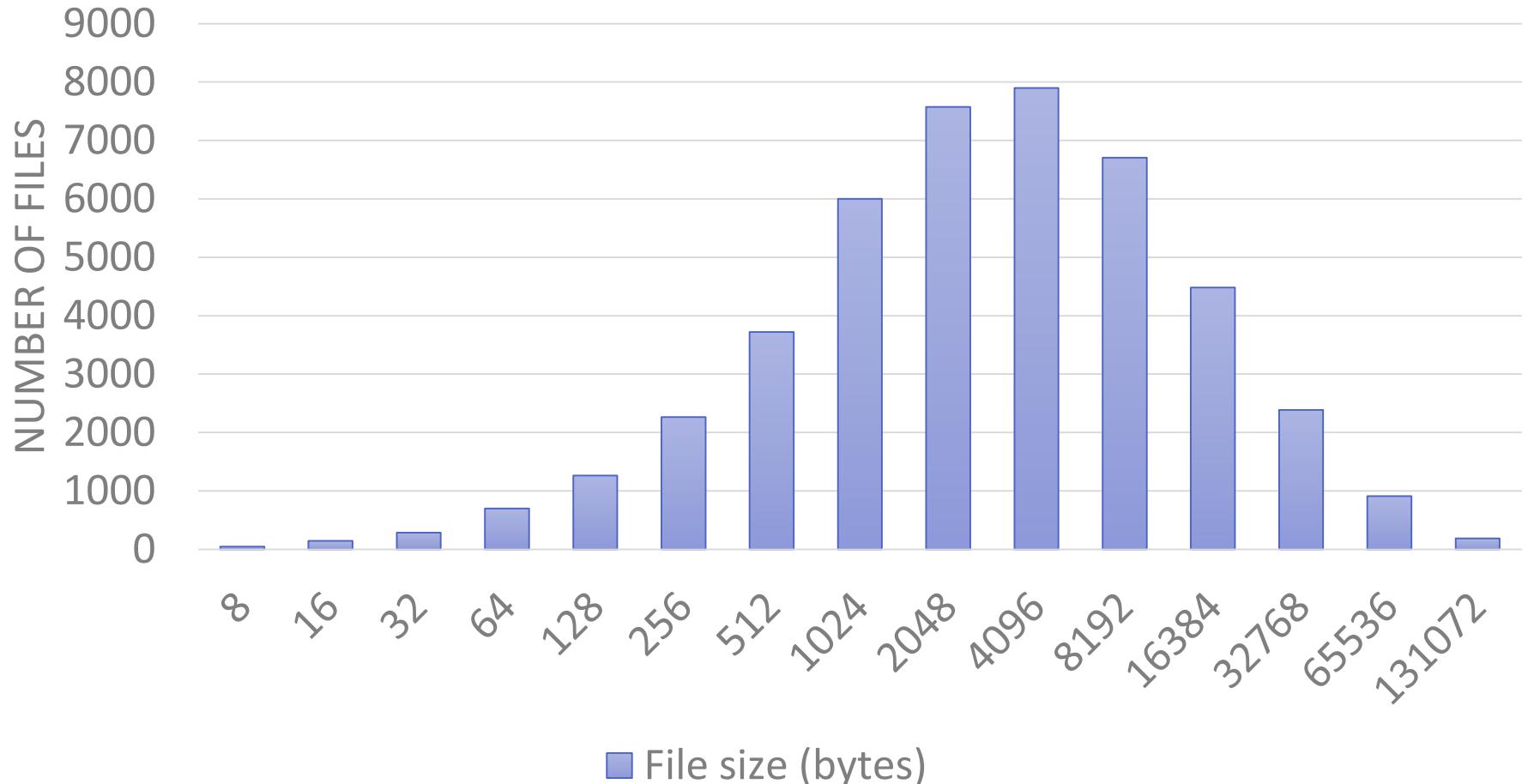
FAT uses a linked list for all files.
It is simple and uniform mechanism. But, it is not optimized for short or long files

Which files are more common: short vs long



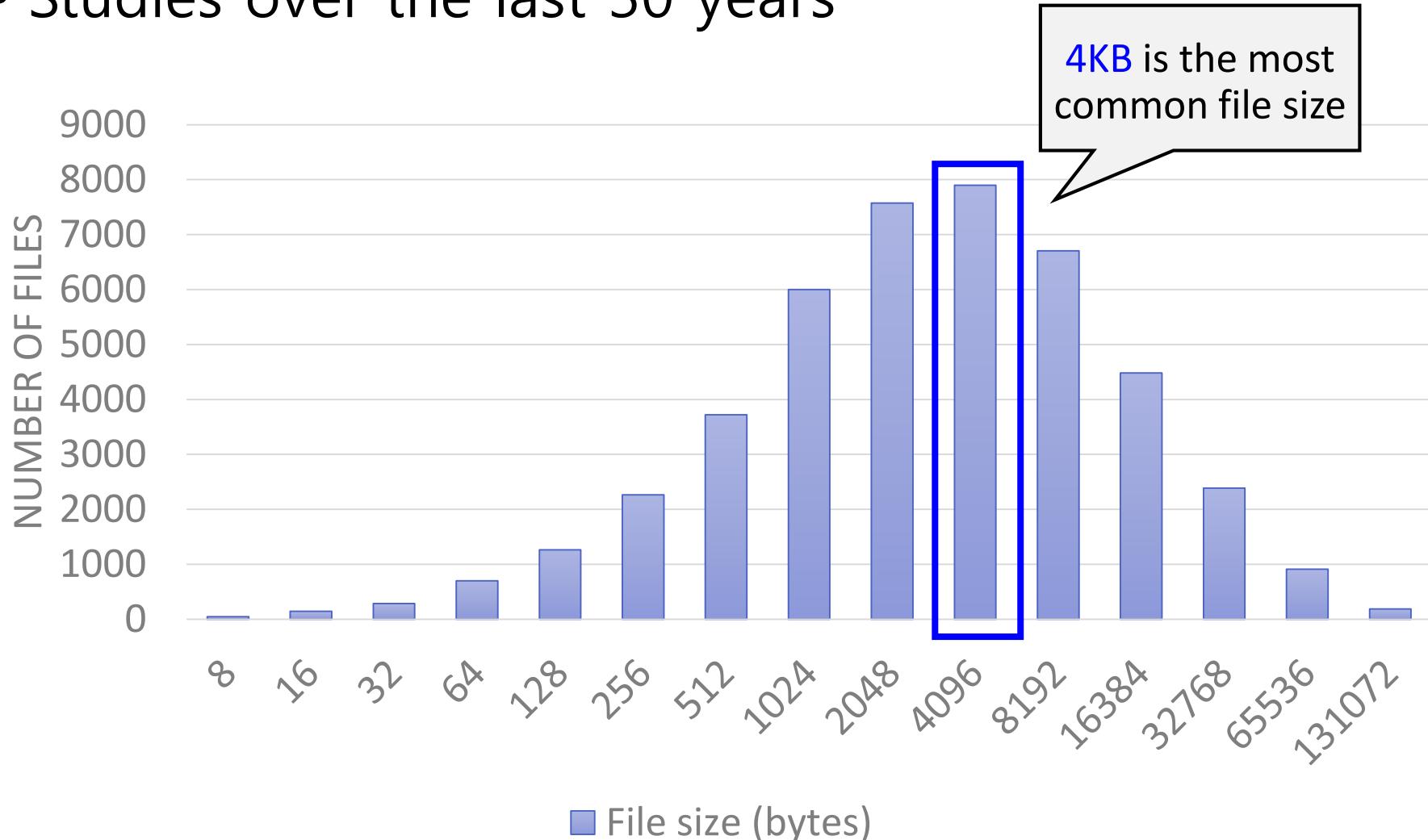
Size Distribution of Files

- Studies over the last 30 years



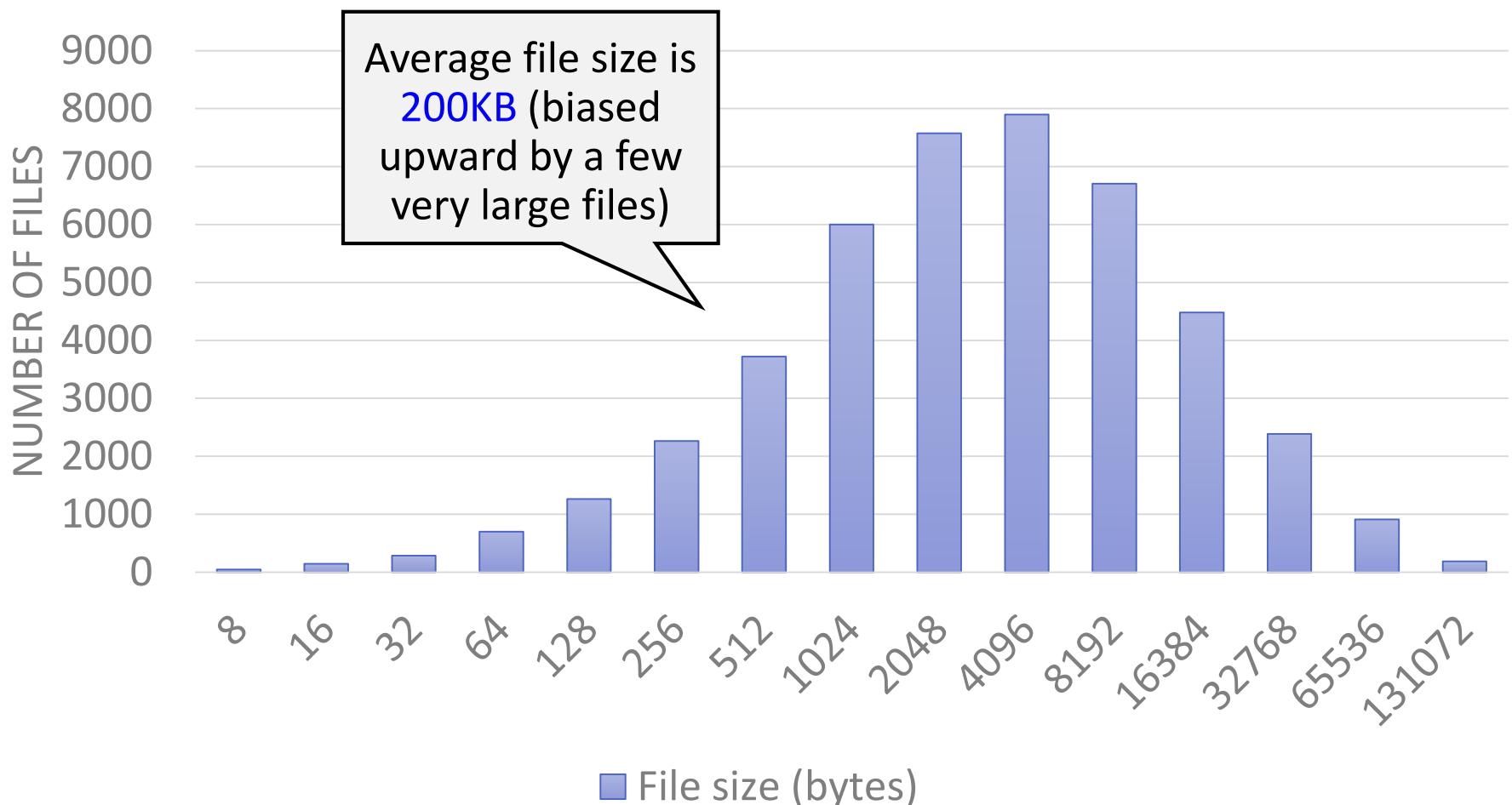
Size Distribution of Files

- Studies over the last 30 years



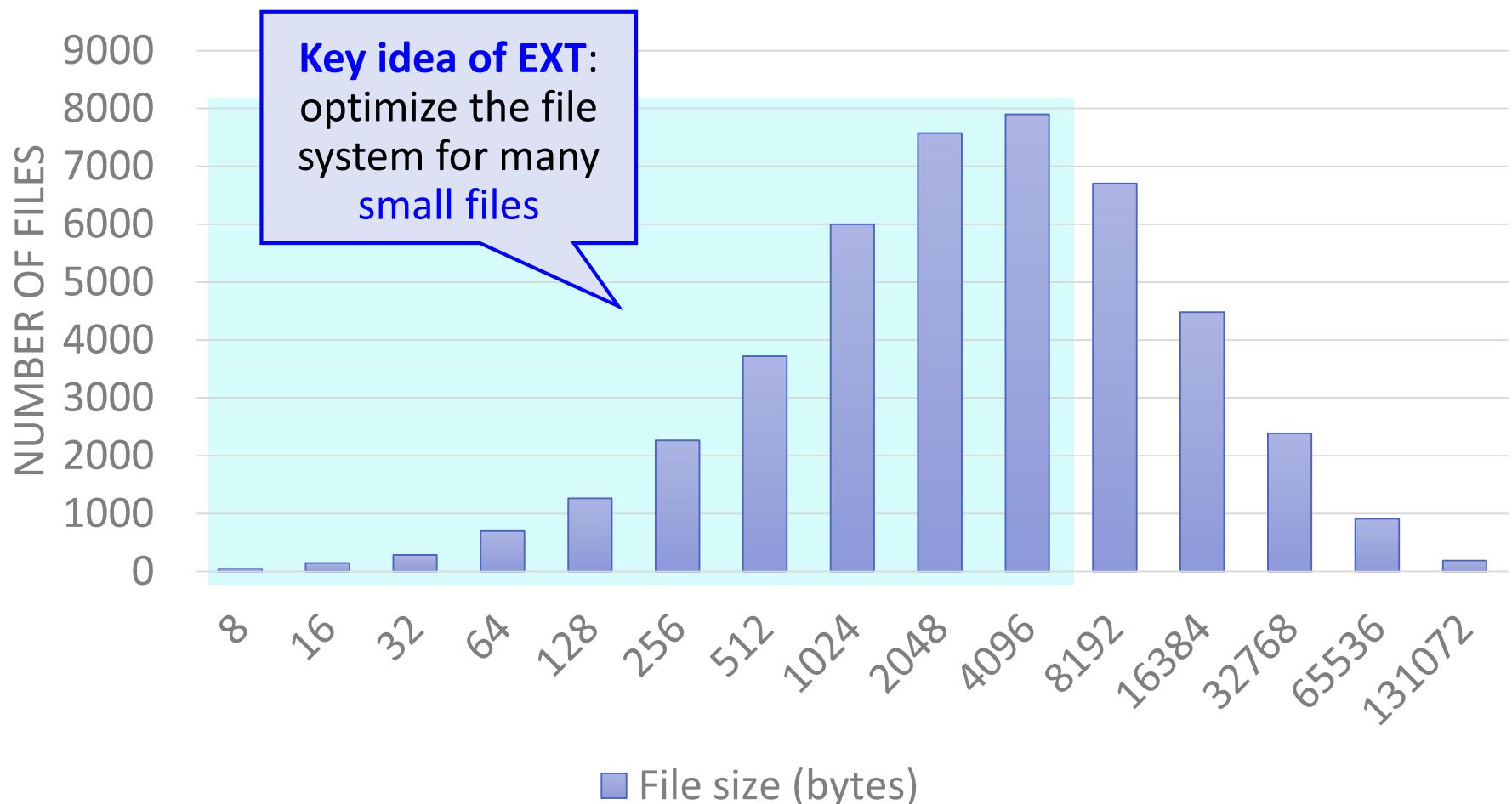
Size Distribution of Files

- Studies over the last 30 years



Size Distribution of Files

- Studies over the last 30 years



On-Disk Layout #2: Inodes and Blocks

Extended File System (EXT)

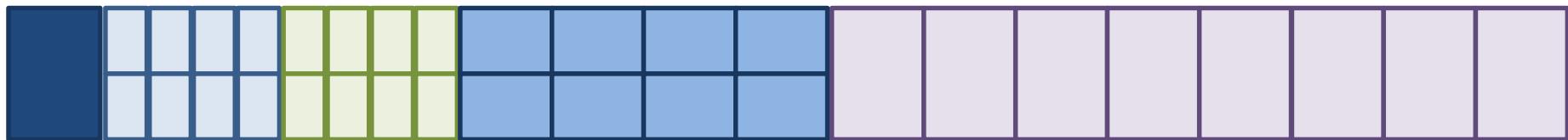
Linux file system
uses more efficient
structures

EXT uses index nodes
([inodes](#)) to track files
and directories

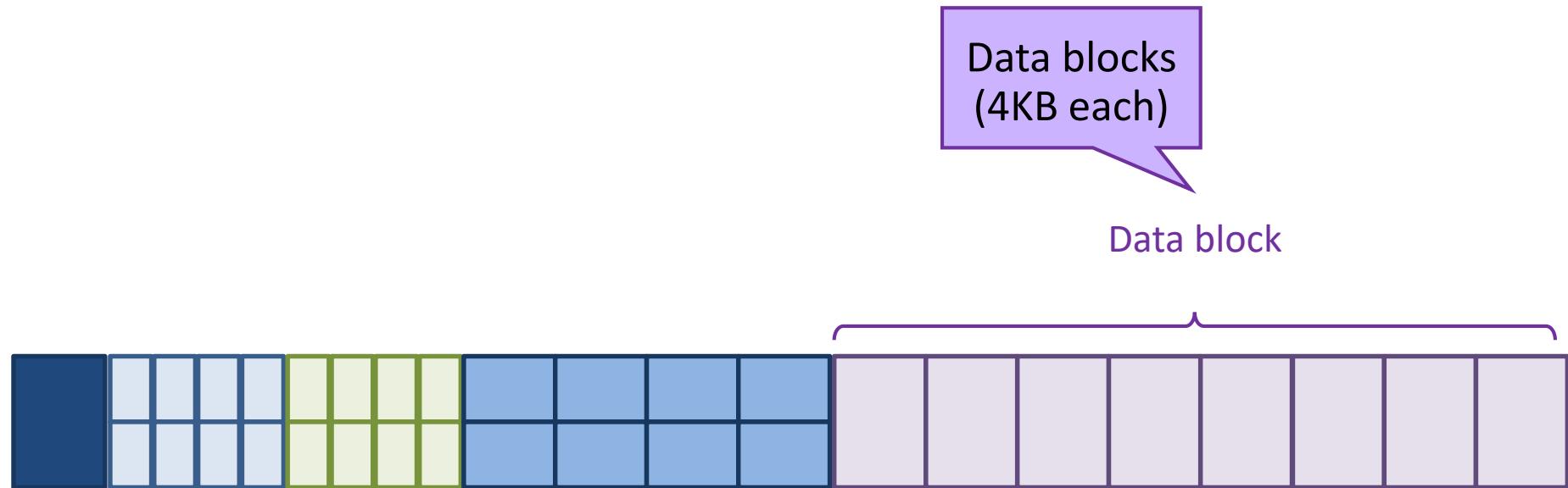


On-disk Layout of EXT

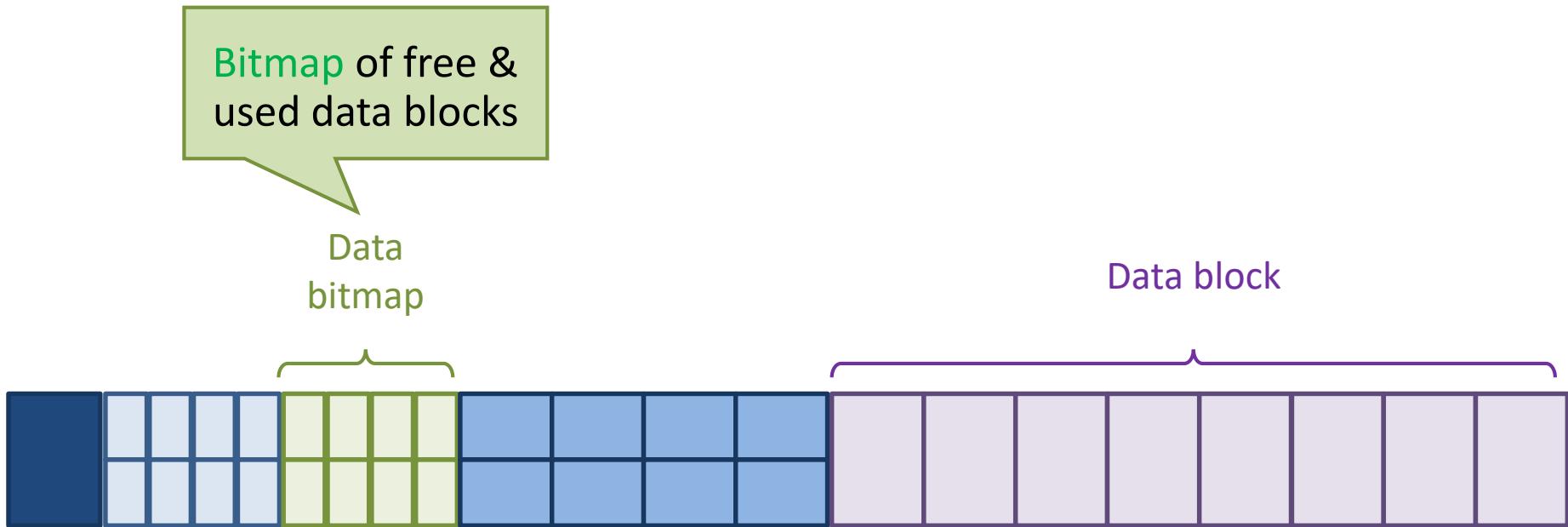
How does EXT
FS look like?



Data block of EXT FS

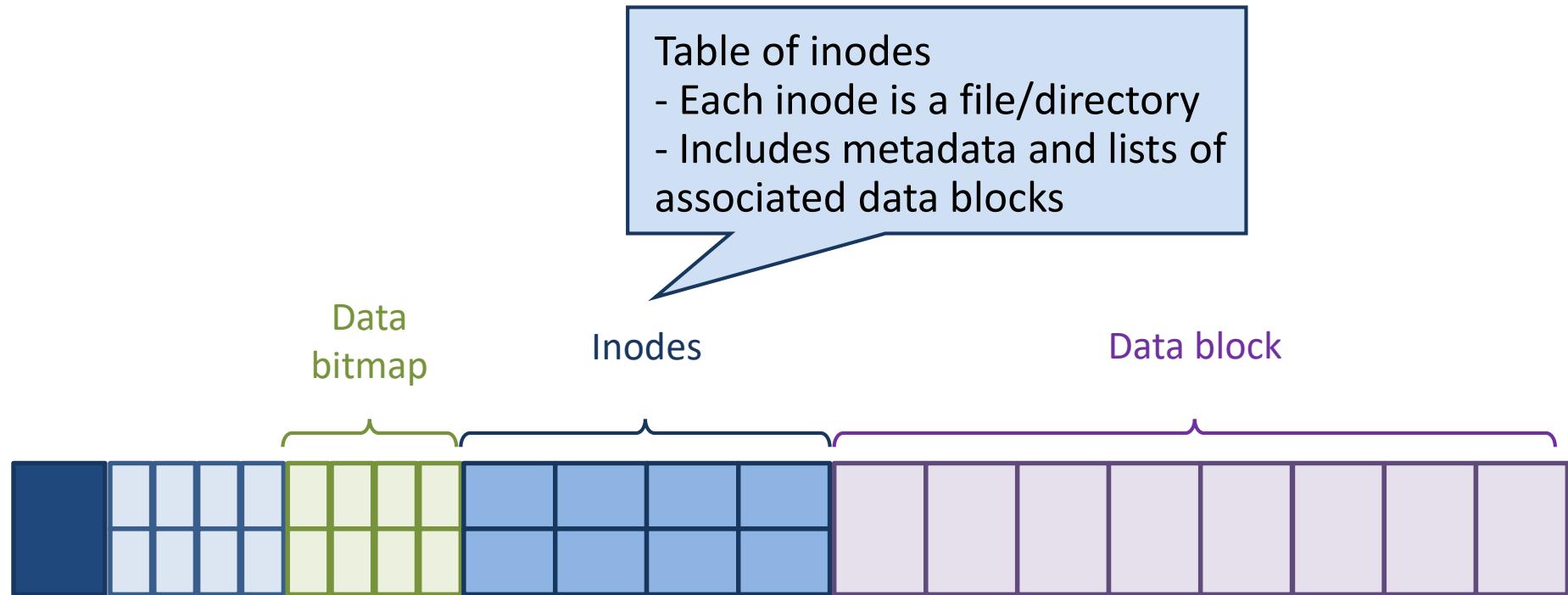


Data bitmap of EXT FS





Inodes of EXT FS



Inodes of EXT FS

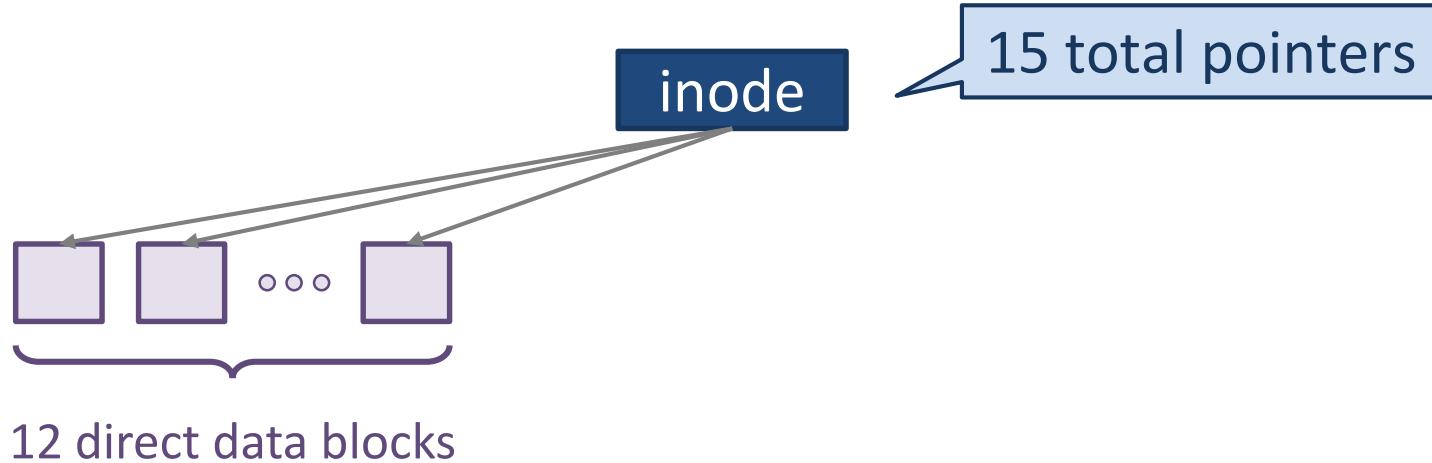
Size (bytes)	Name	What is this field for?
2	mode	Read/write/execute?
2	uid	User ID of the file owner
4	size	Size of the file in bytes
4	time	Last access time
4	ctime	Creation time
4	mtime	Last modification time
4	dtime	Deletion time
2	gid	Group ID of the file
2	links count	How many hard links point to this file?
4	blocks	How many data blocks are allocated to this file?
4	flags	File or directory? Plus, other simple flags
60	block	15 direct and indirect pointers to data blocks





Inode' Block Pointers

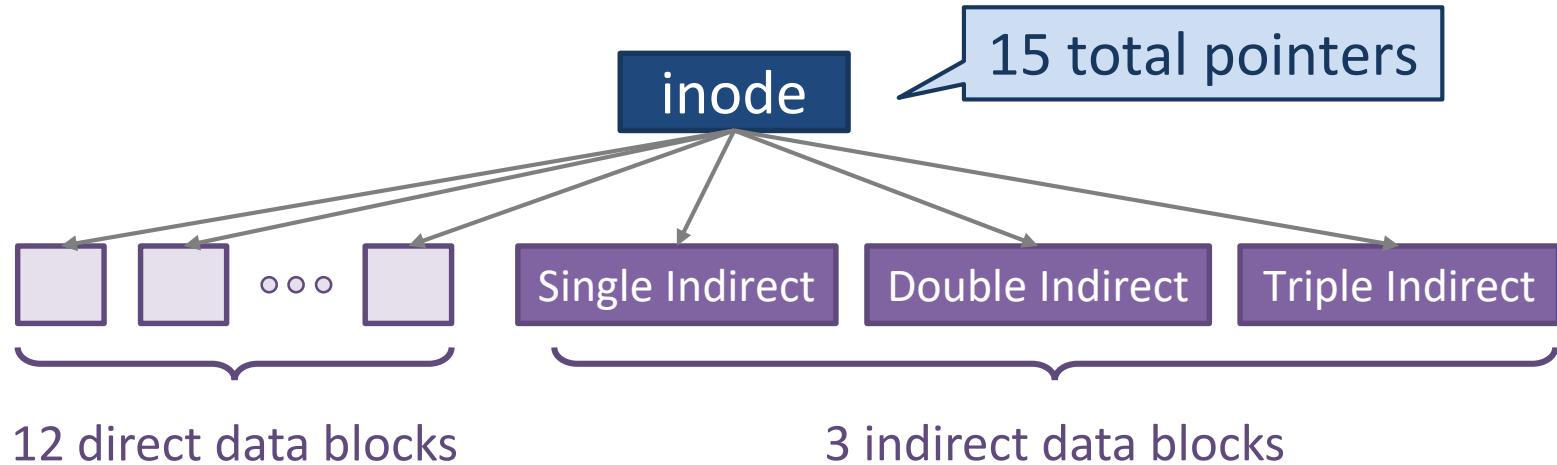
- Each **inode** is the root of an unbalanced tree of data blocks





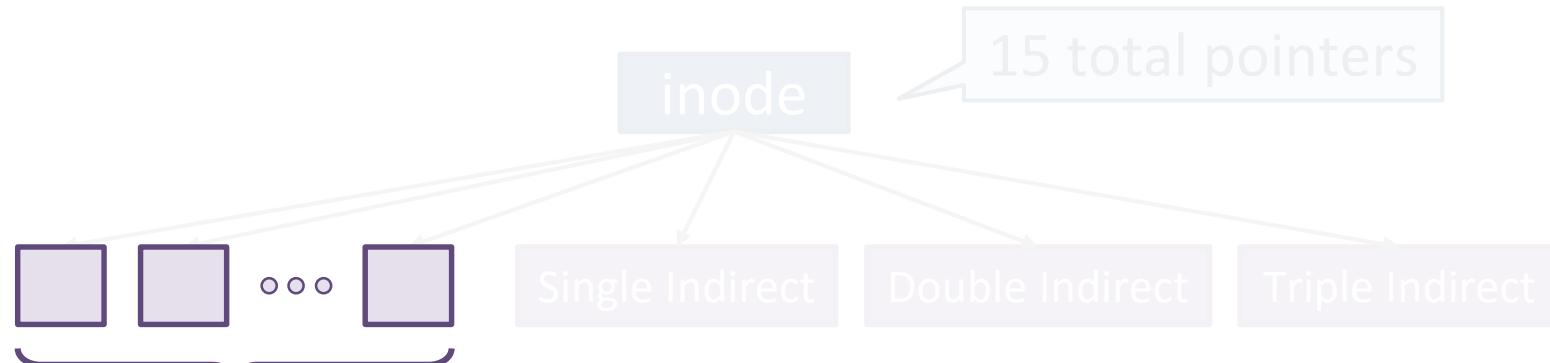
Inode' Block Pointers

- Each **inode** is the root of an unbalanced tree of data blocks



Inode' Block Pointers

- Each **inode** is the root of an unbalanced tree of data blocks



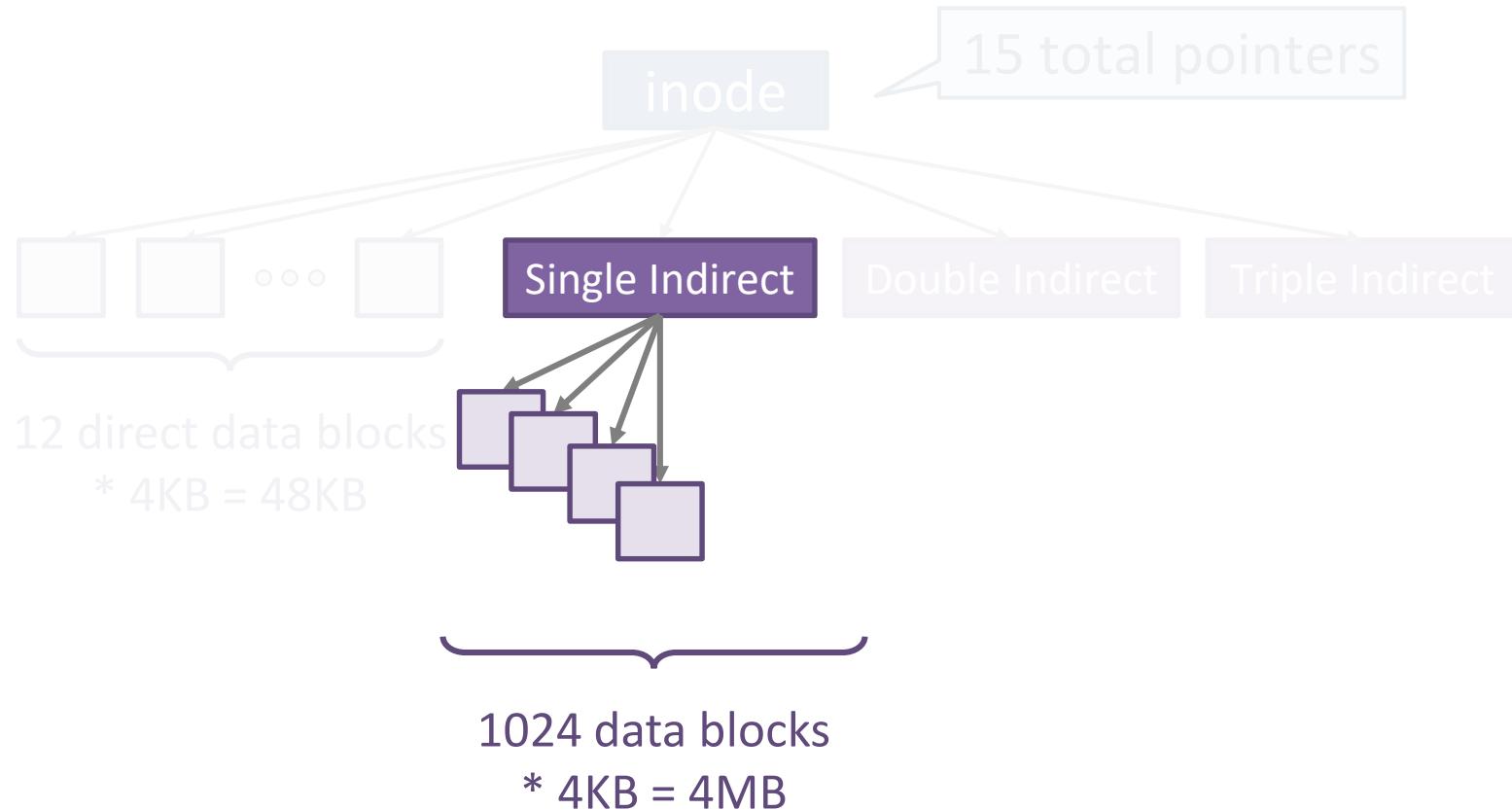
12 direct data blocks
* 4KB = 48KB

As small files are common,
almost inode only indicates a
single direct data blocks ☺



Inode' Block Pointers

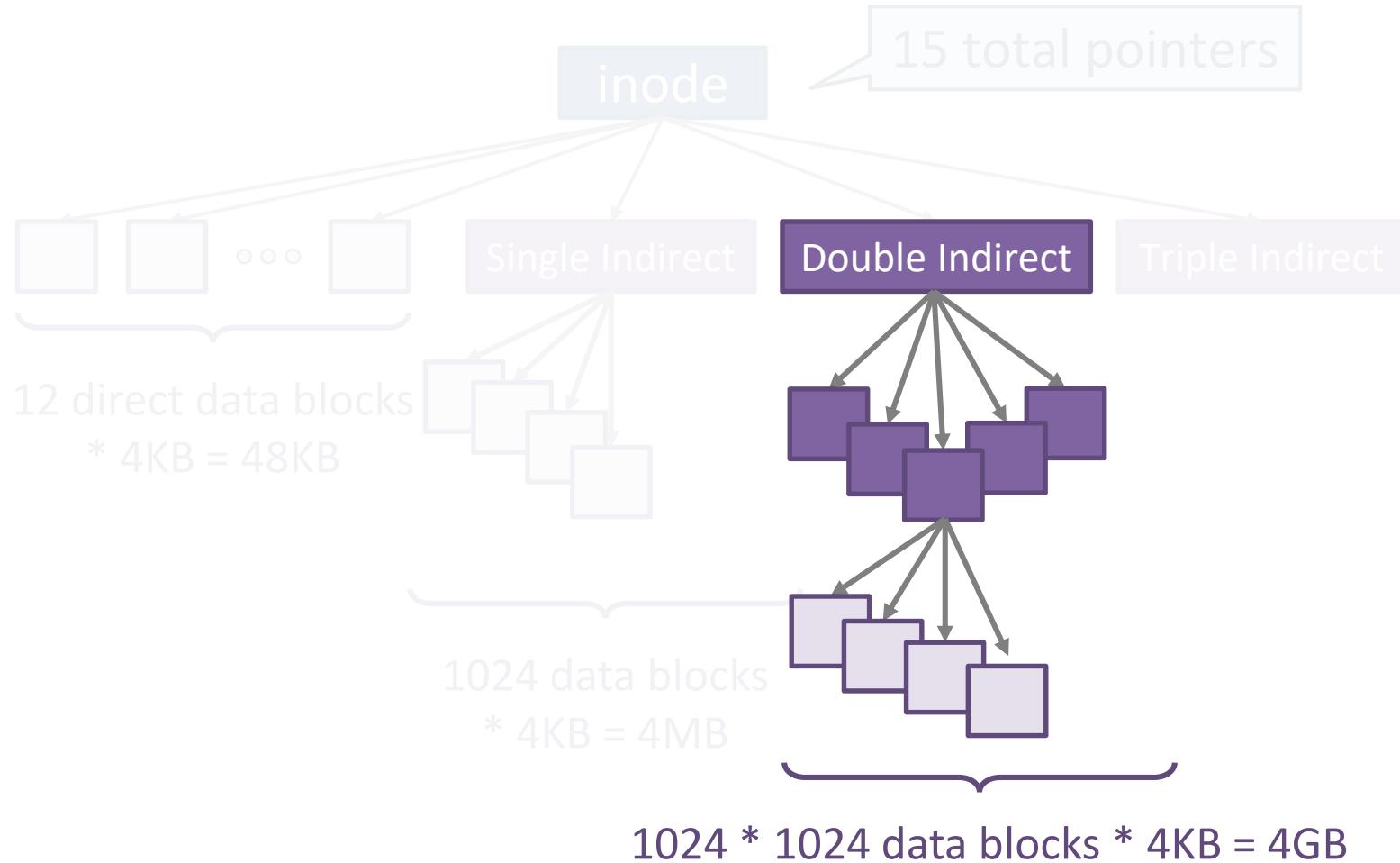
- Each **inode** is the root of an unbalanced tree of data blocks





Inode' Block Pointers

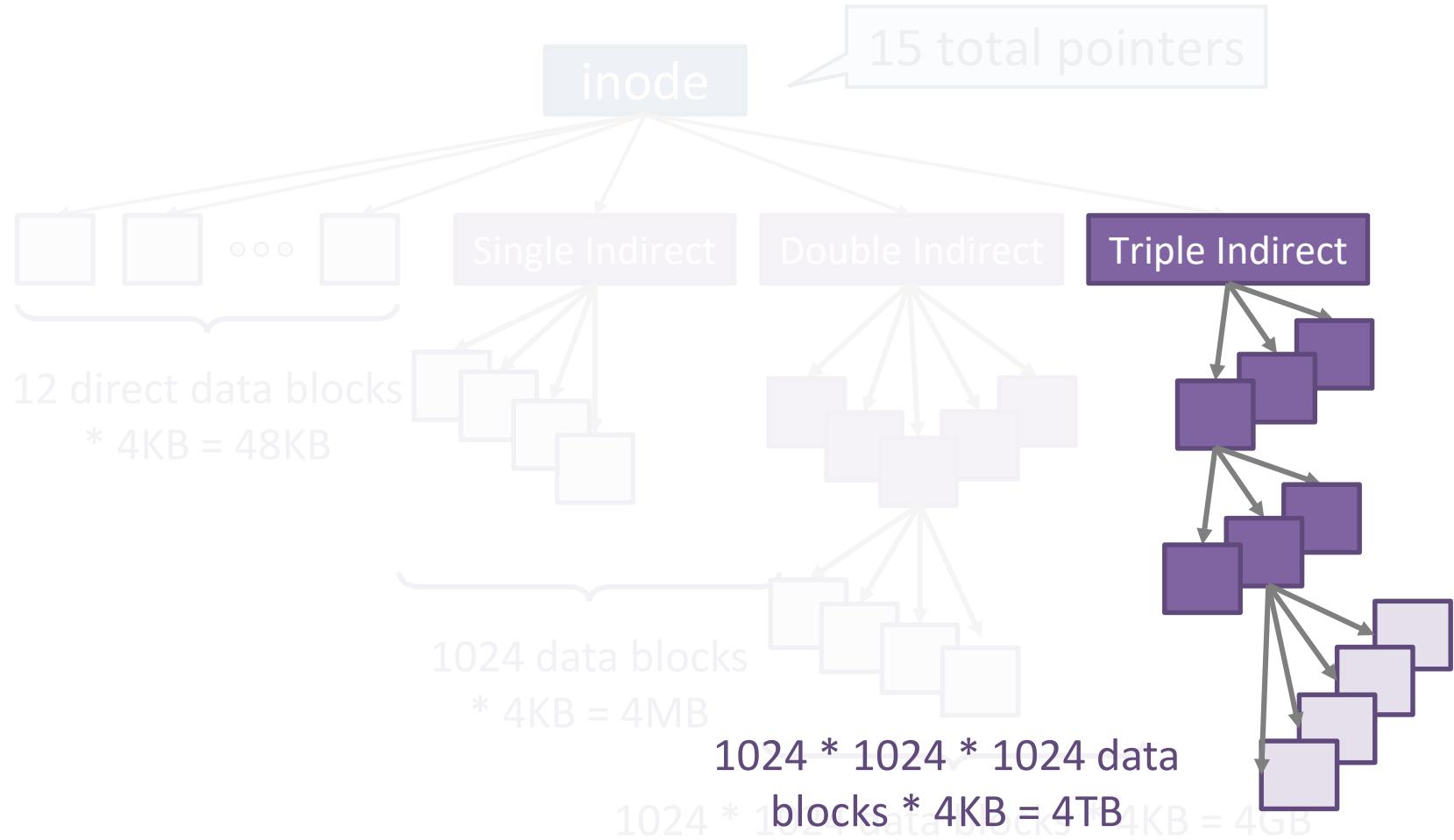
- Each **inode** is the root of an unbalanced tree of data blocks





Inode' Block Pointers

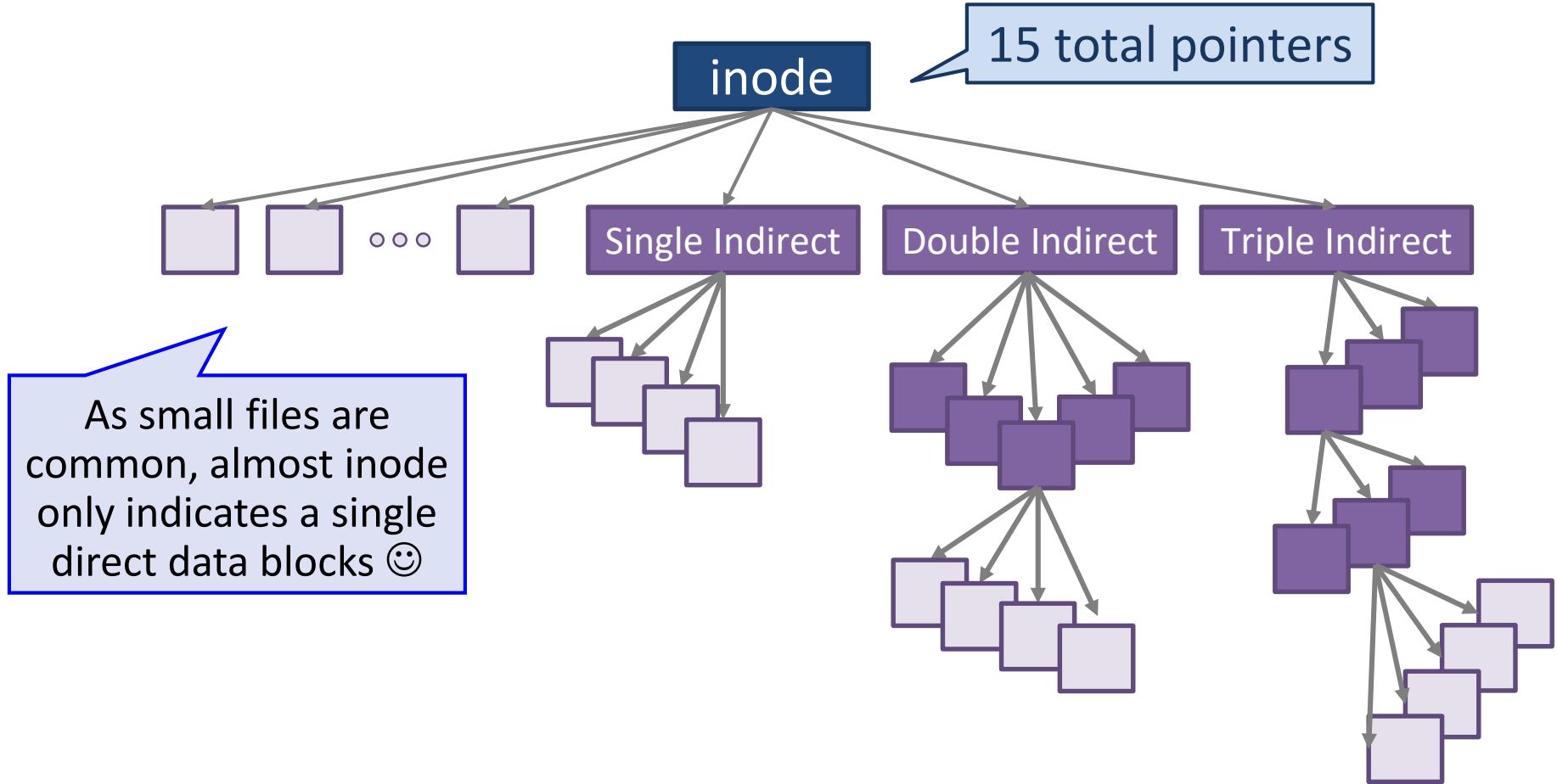
- Each **inode** is the root of an unbalanced tree of data blocks



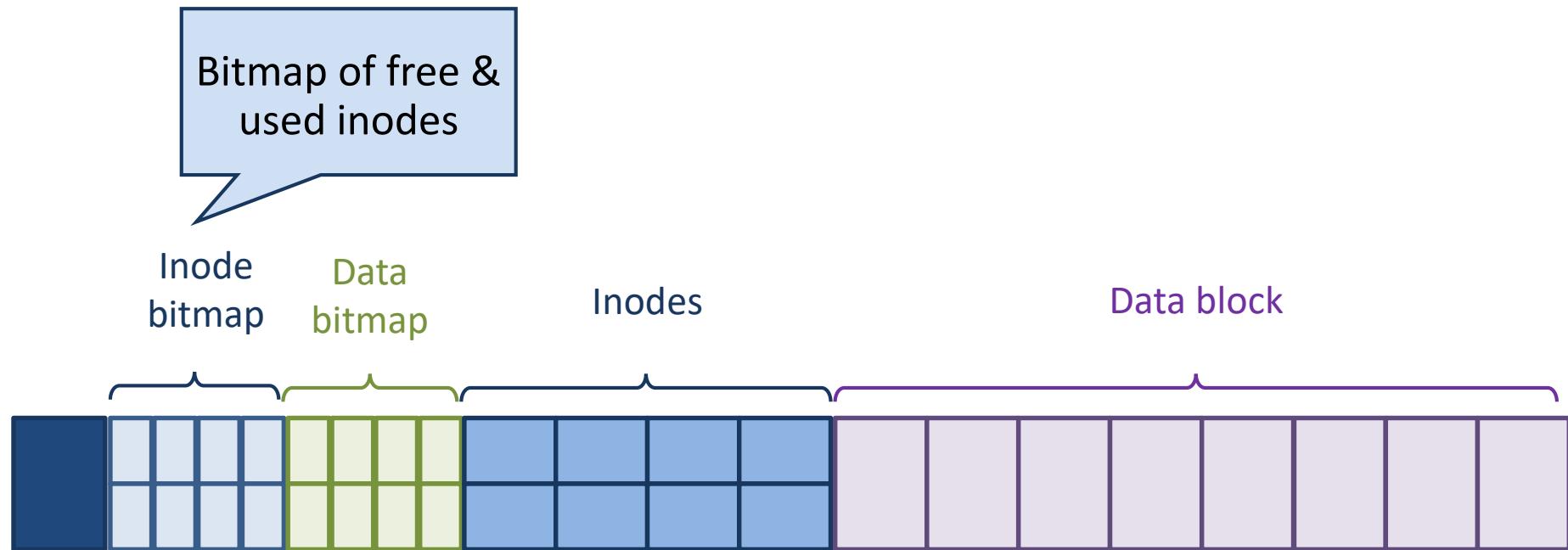


Inode' Block Pointers

- Each **inode** is the root of an unbalanced tree of data blocks



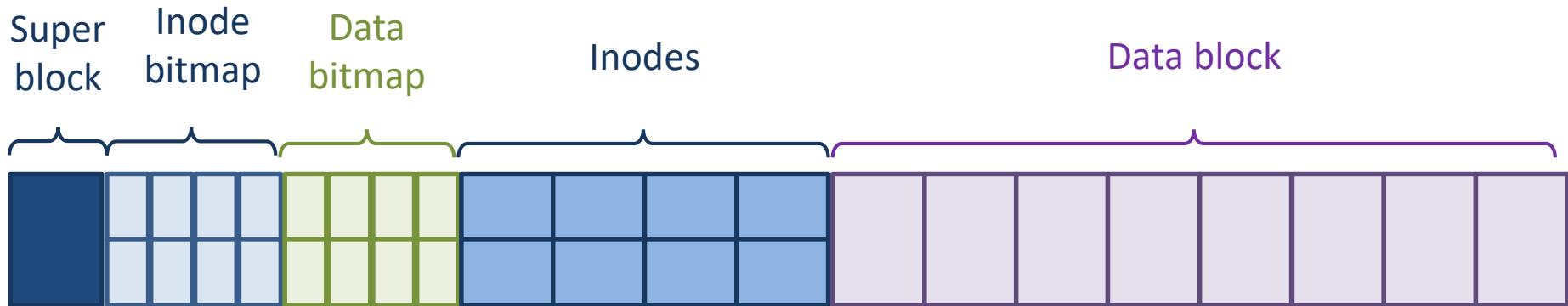
Inode Bitmap of EXT FS



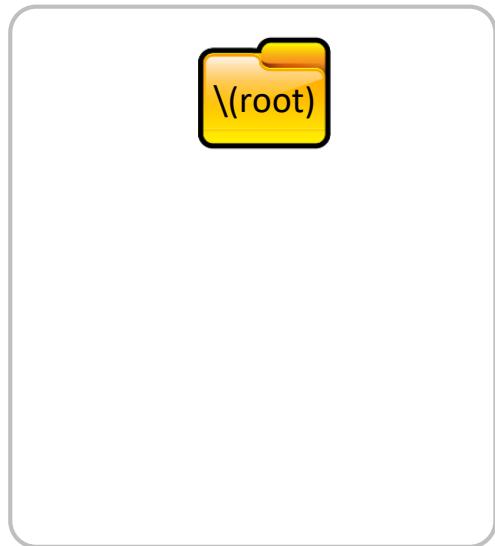
Superblock of EXT FS

Superblock storing:

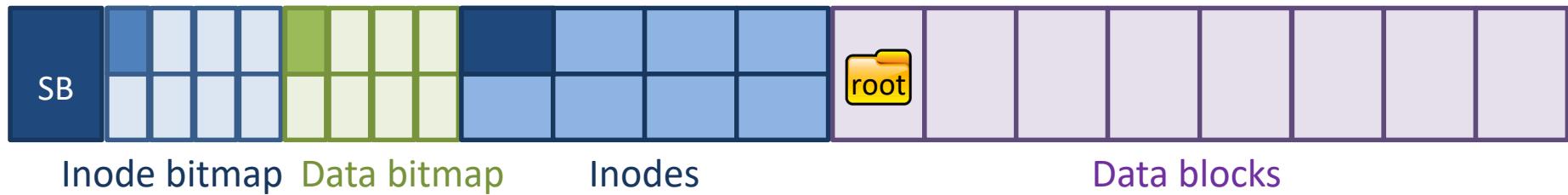
- Size and location of bitmaps
- Number and location of inodes
- Number and location of data blocks
- Index of root inodes



Example of EXT FS

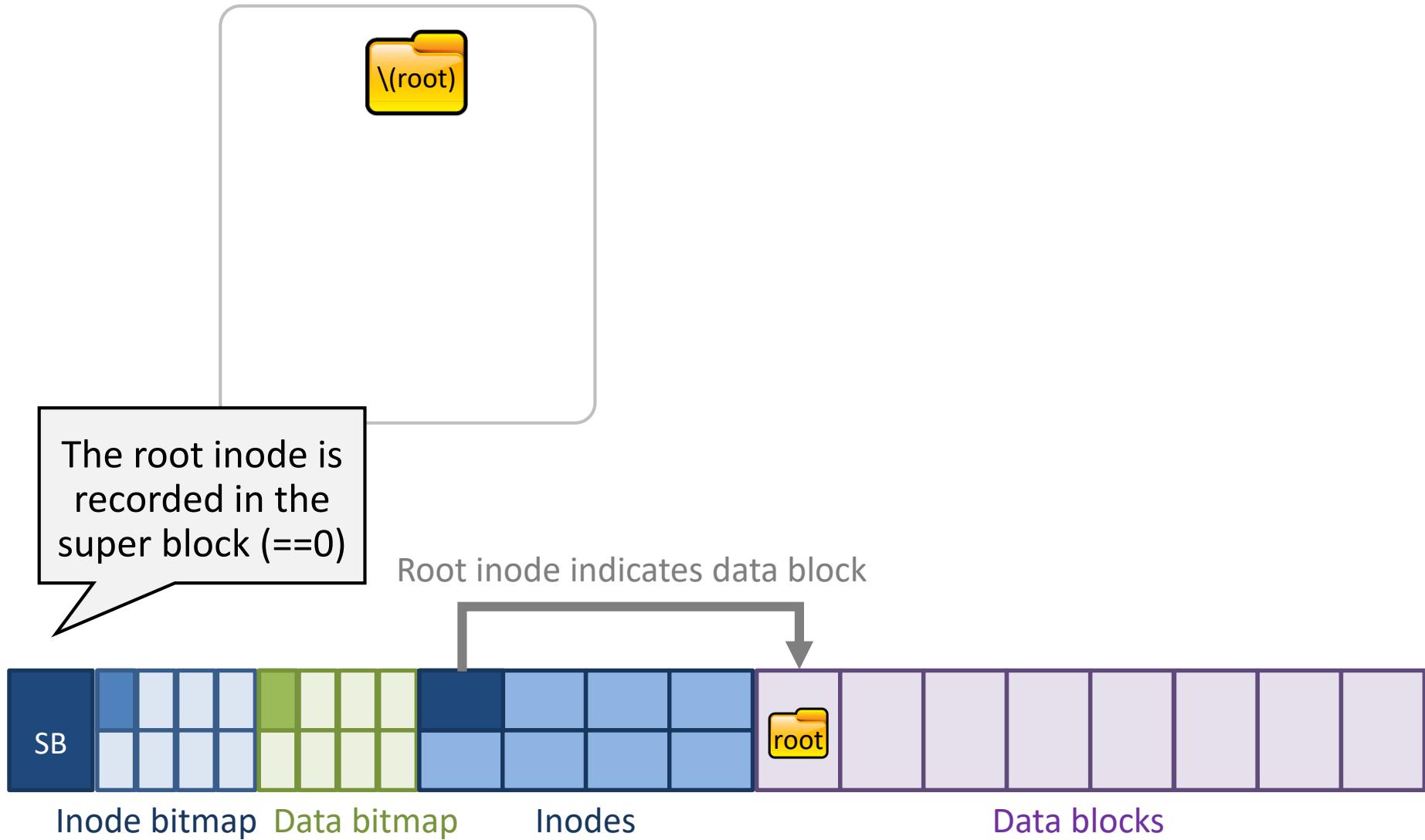


This is our system's
directory tree. Where
is the root **inode**?

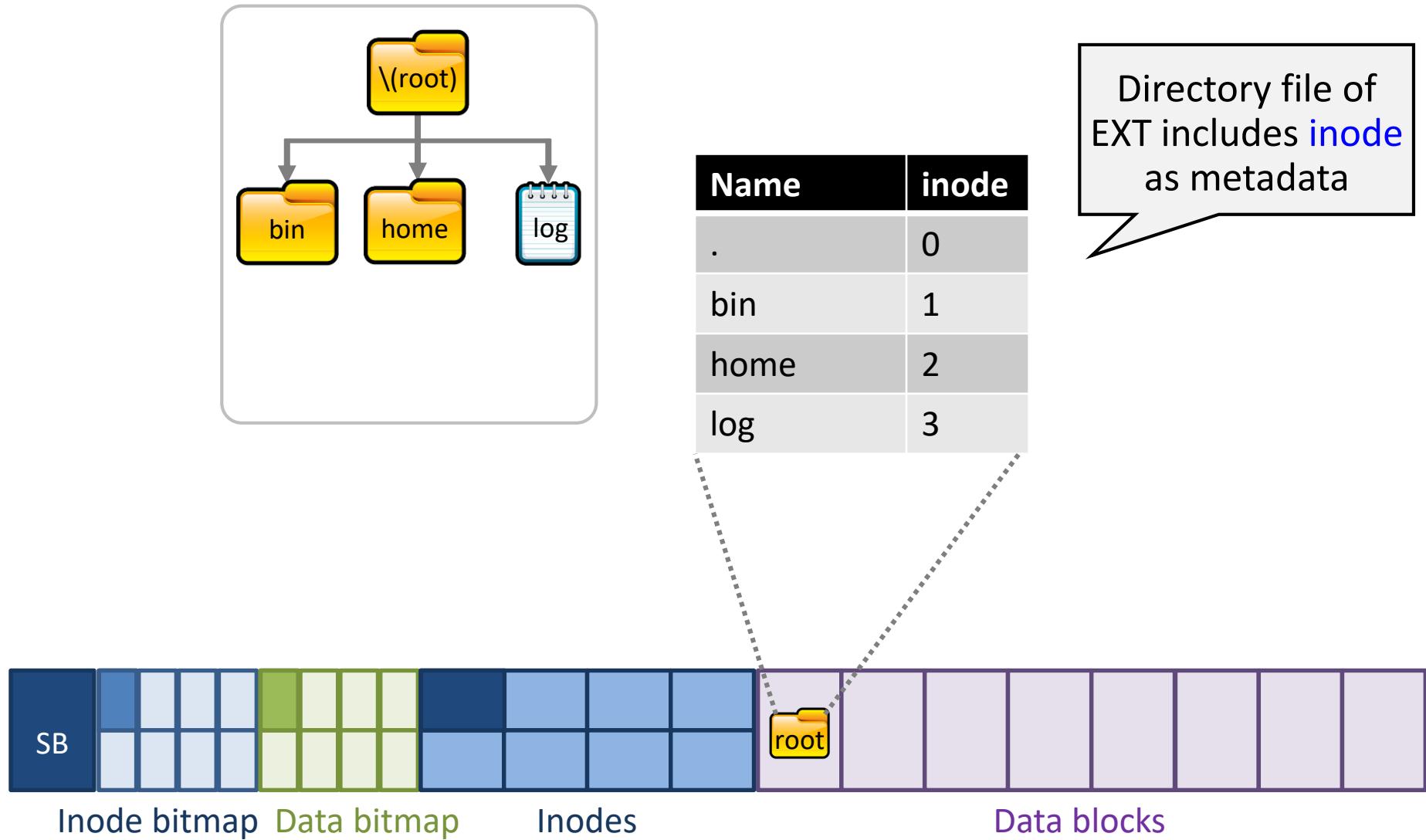




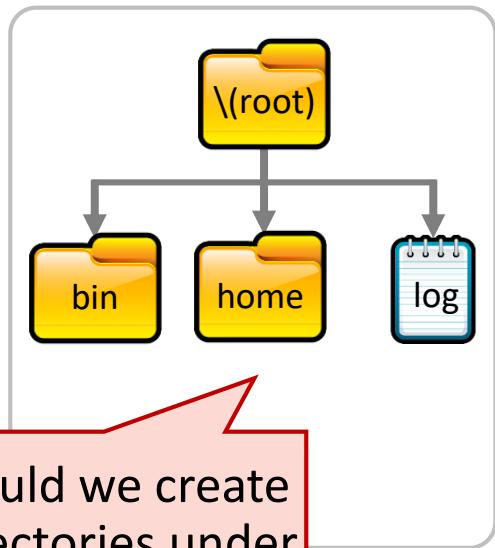
Example of EXT FS



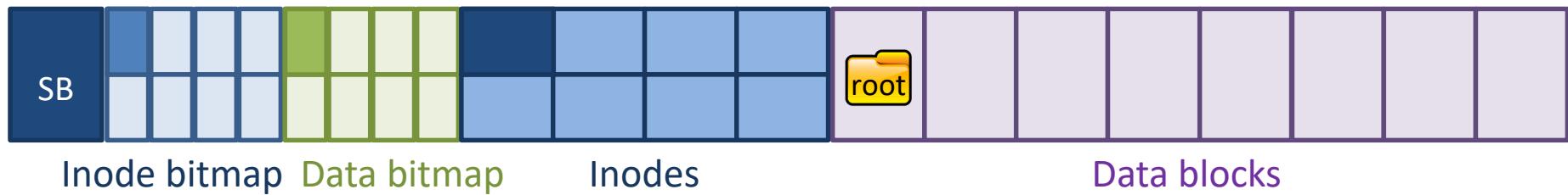
Example of EXT FS



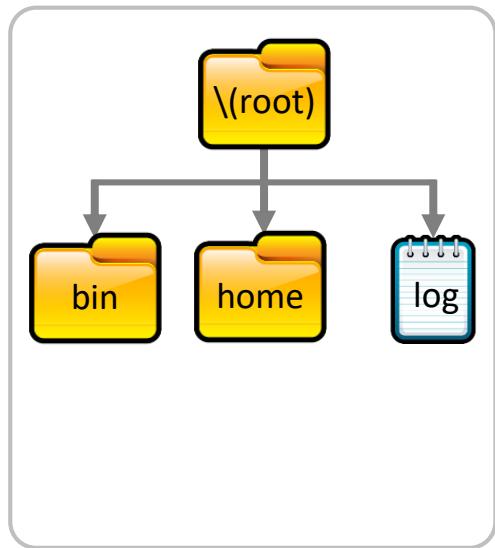
Example of EXT FS



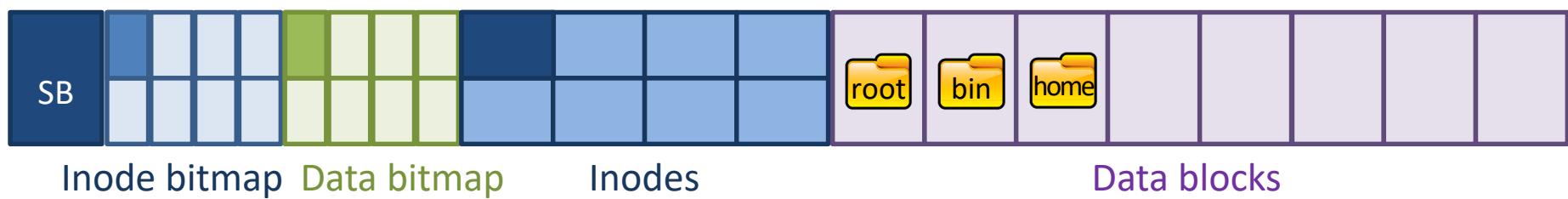
How could we create
two directories under
the root?



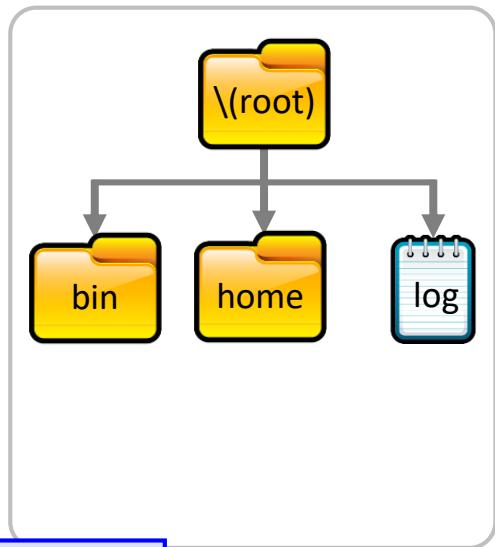
Example of EXT FS



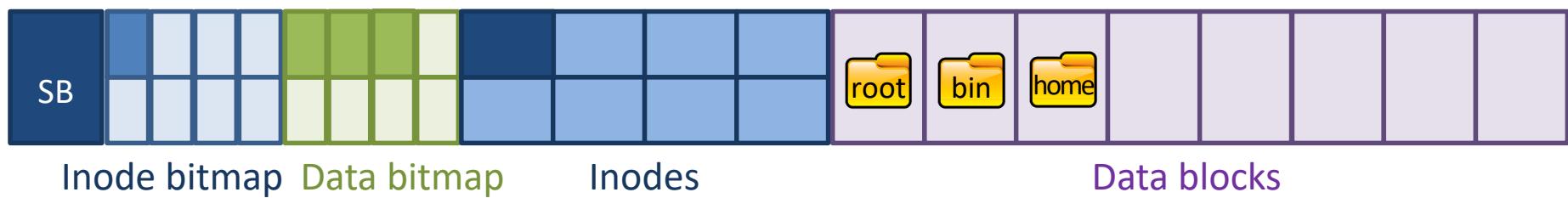
Step #1:
Allocate two
data blocks



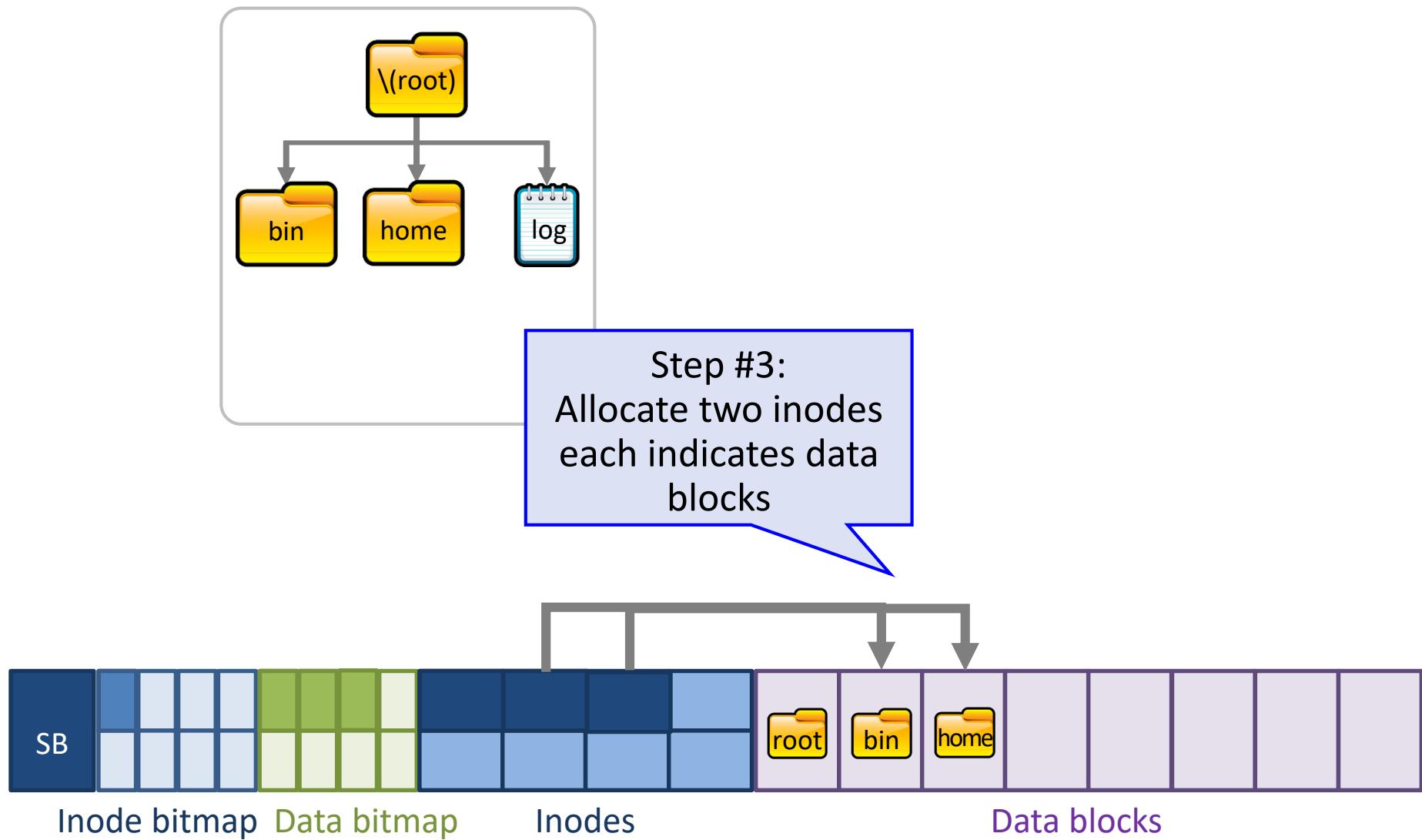
Example of EXT FS



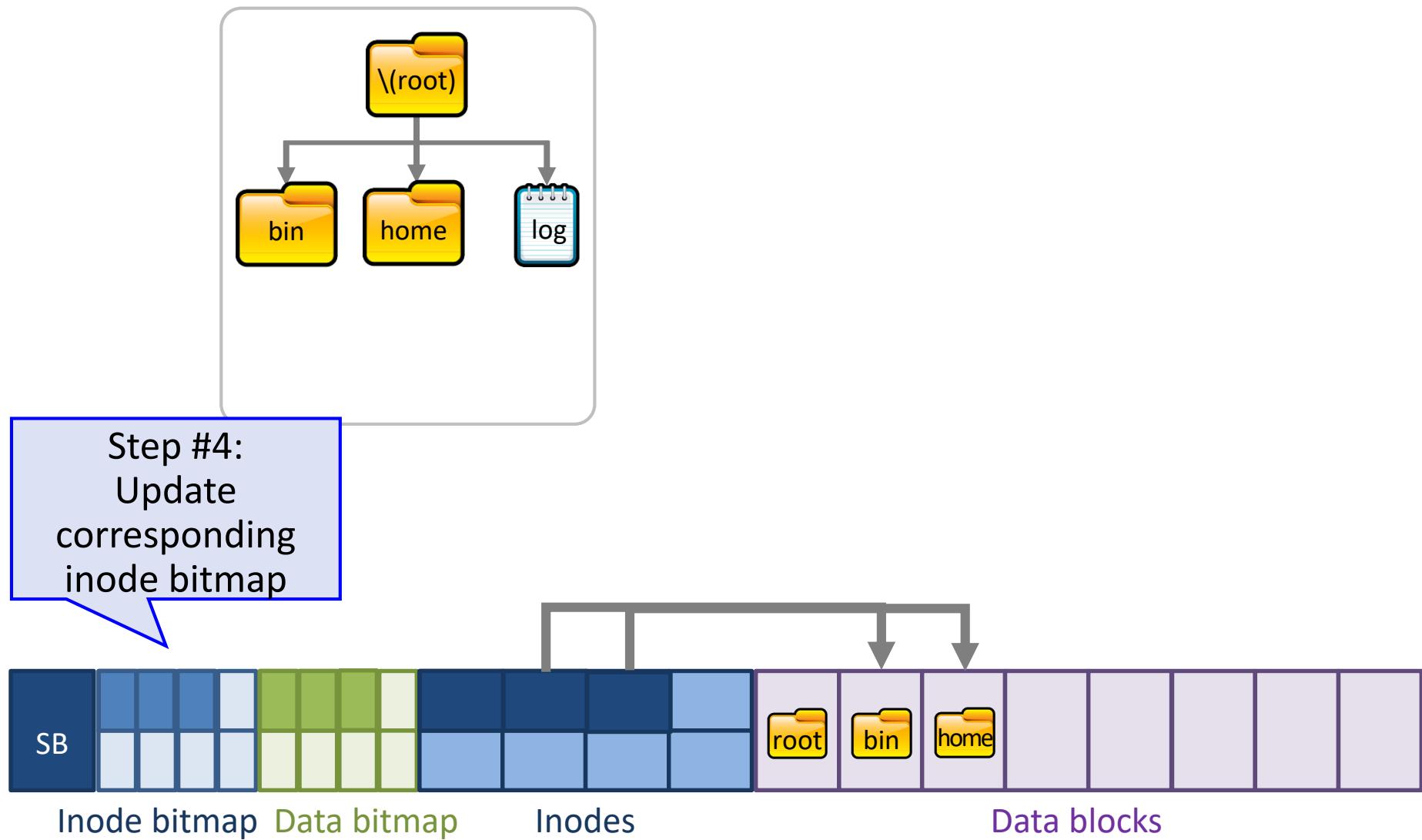
Step #2:
Update
corresponding
data bitmap



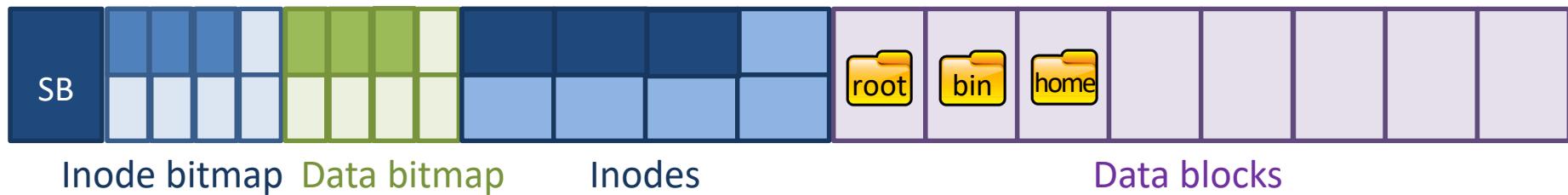
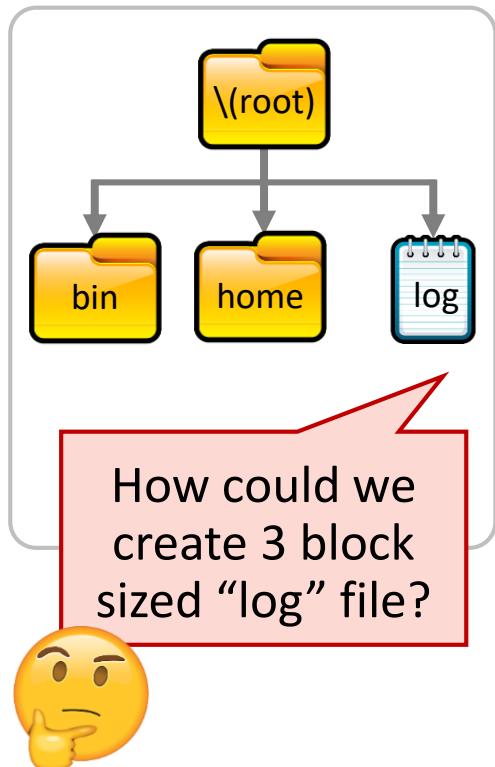
Example of EXT FS



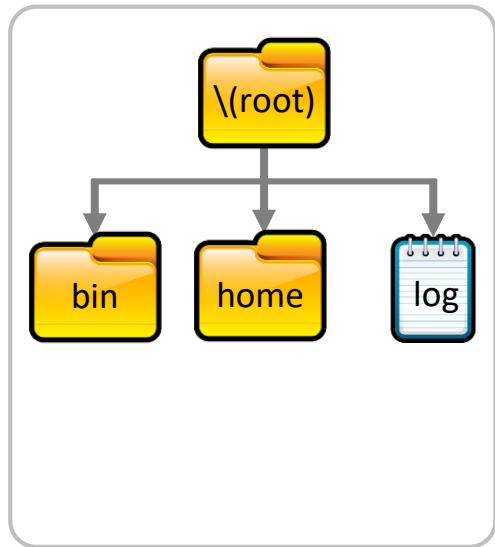
Example of EXT FS



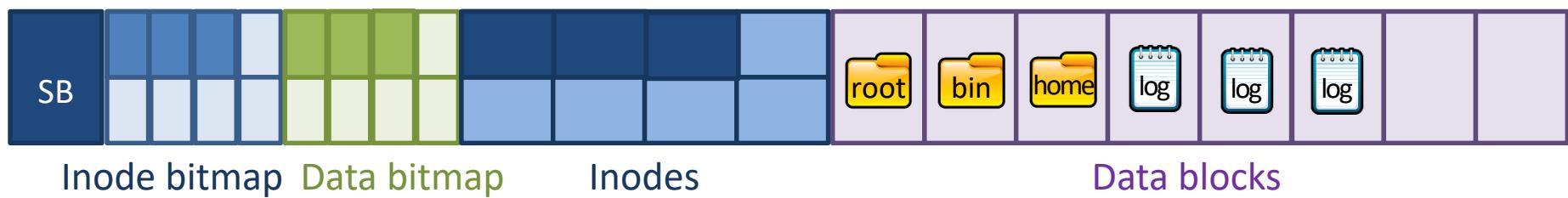
Example of EXT FS



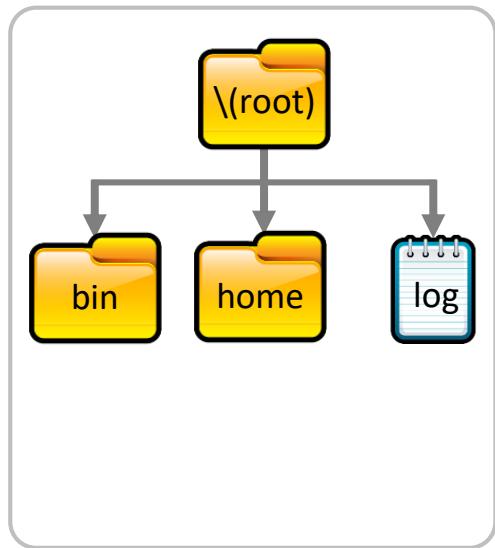
Example of EXT FS



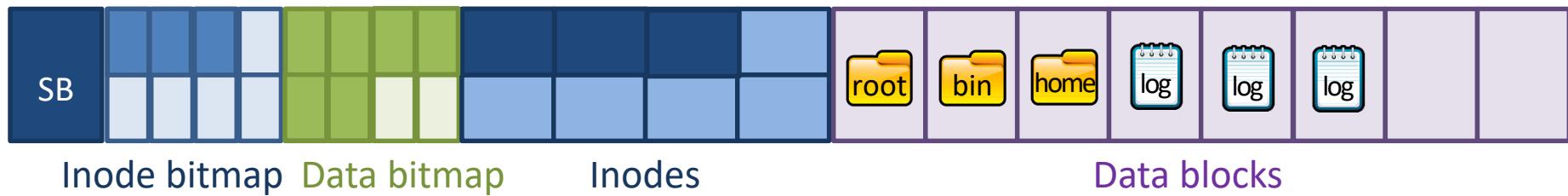
Step #1:
Allocate three
data blocks



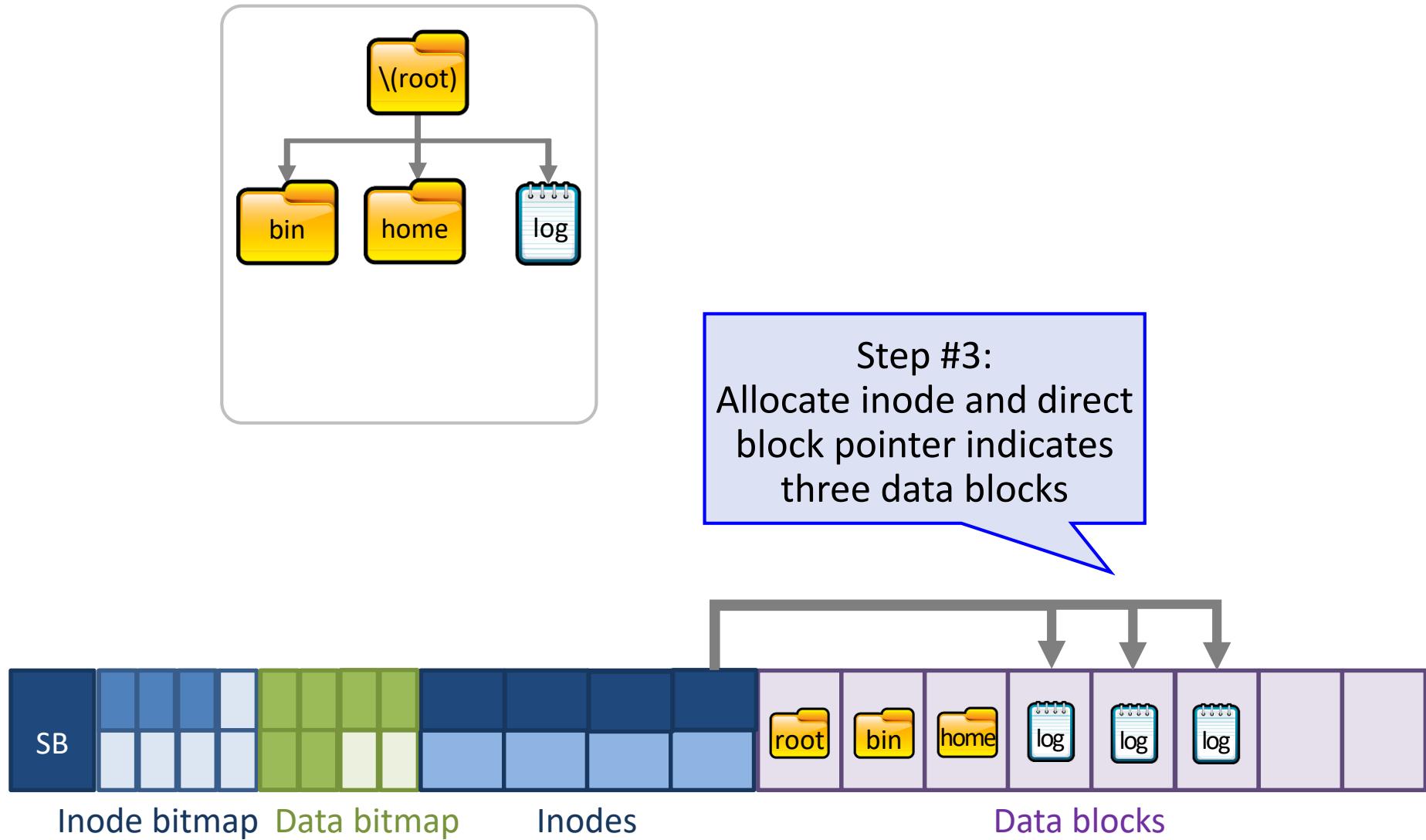
Example of EXT FS



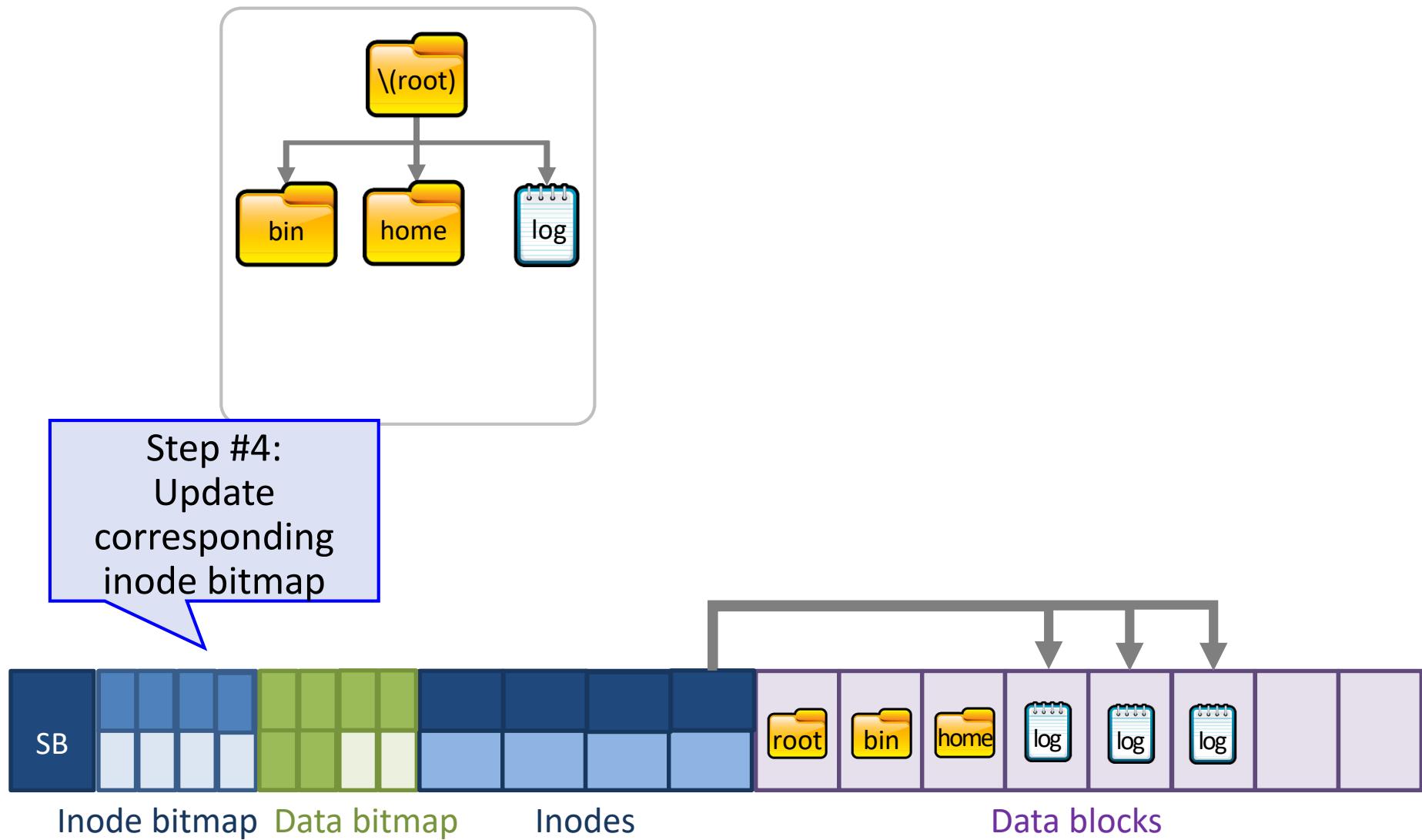
Step #2:
Update corresponding
data bitmap



Example of EXT FS



Example of EXT FS



EXT Inodes, Again

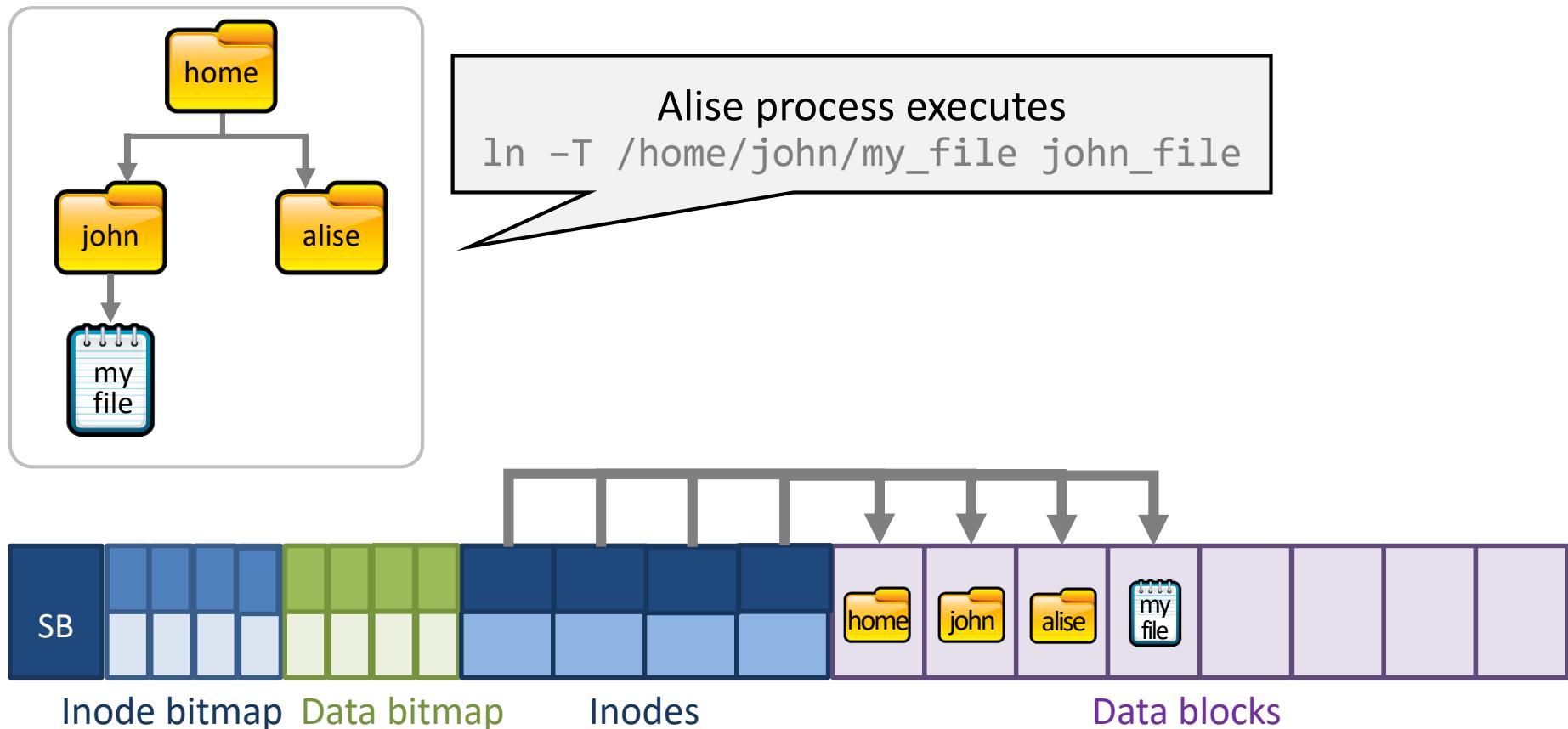
Size (bytes)	Name	What is this field for?
2	mode	Read/write/execute?
2	uid	User ID of the file owner
4	size	Size of the file in bytes
4	time	Last access time
4	ctime	Creation time
4	mtime	Last modification time
4	dtime	Deletion time
2	gid	Group ID of the file
2	links_count	How many hard links point to this file?
4	blocks	How many data blocks are allocated to this file?
4	flags	File or directory? Plus, other simple flags
60	block	15 direct and indirect pointers to data blocks





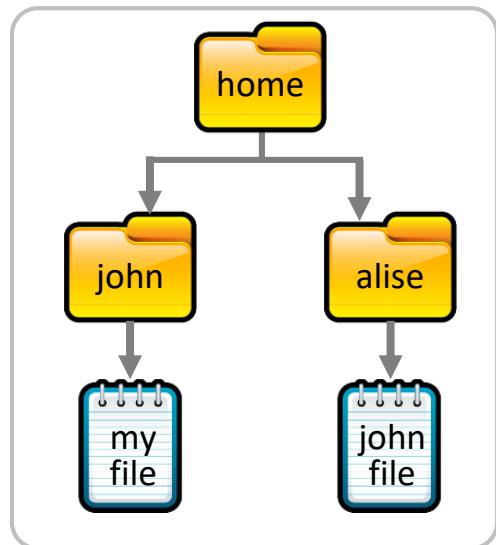
Hard Link

- Multiple directory entries may point to the same inode



Hard Link

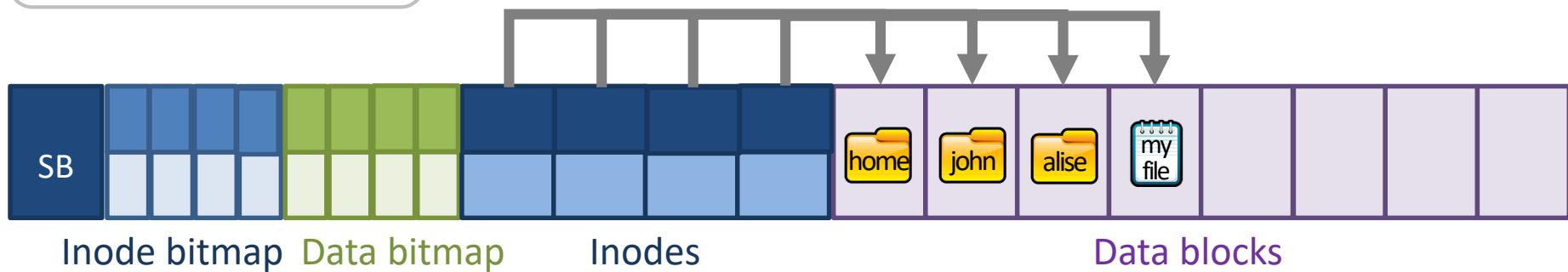
- Multiple directory entries may point to the same inode



Step #1:
Add an entry to the
“alise” directory

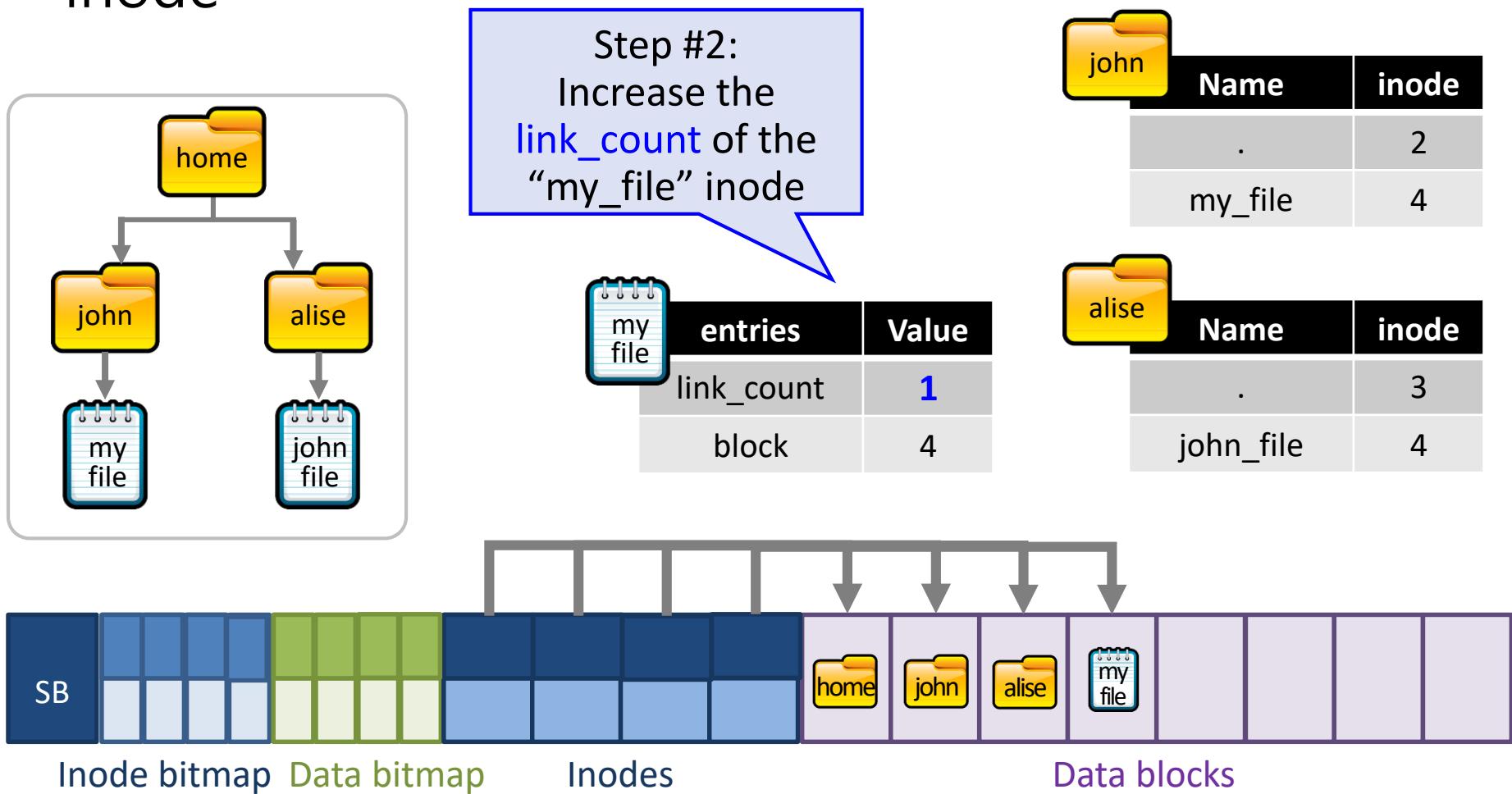
Name	inode
john	2
my_file	4

Name	inode
alise	3
john_file	4



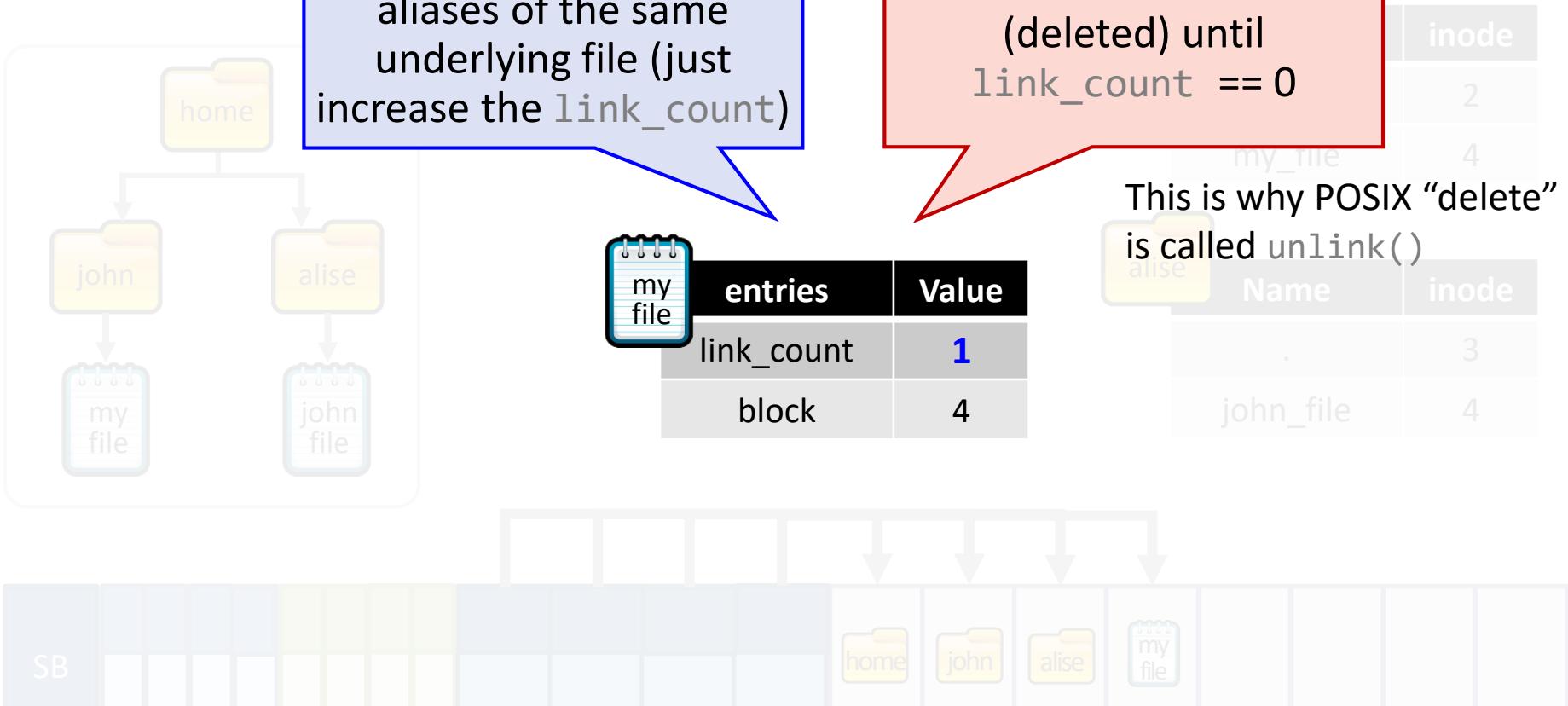
Hard Link

- Multiple directory entries may point to the same inode



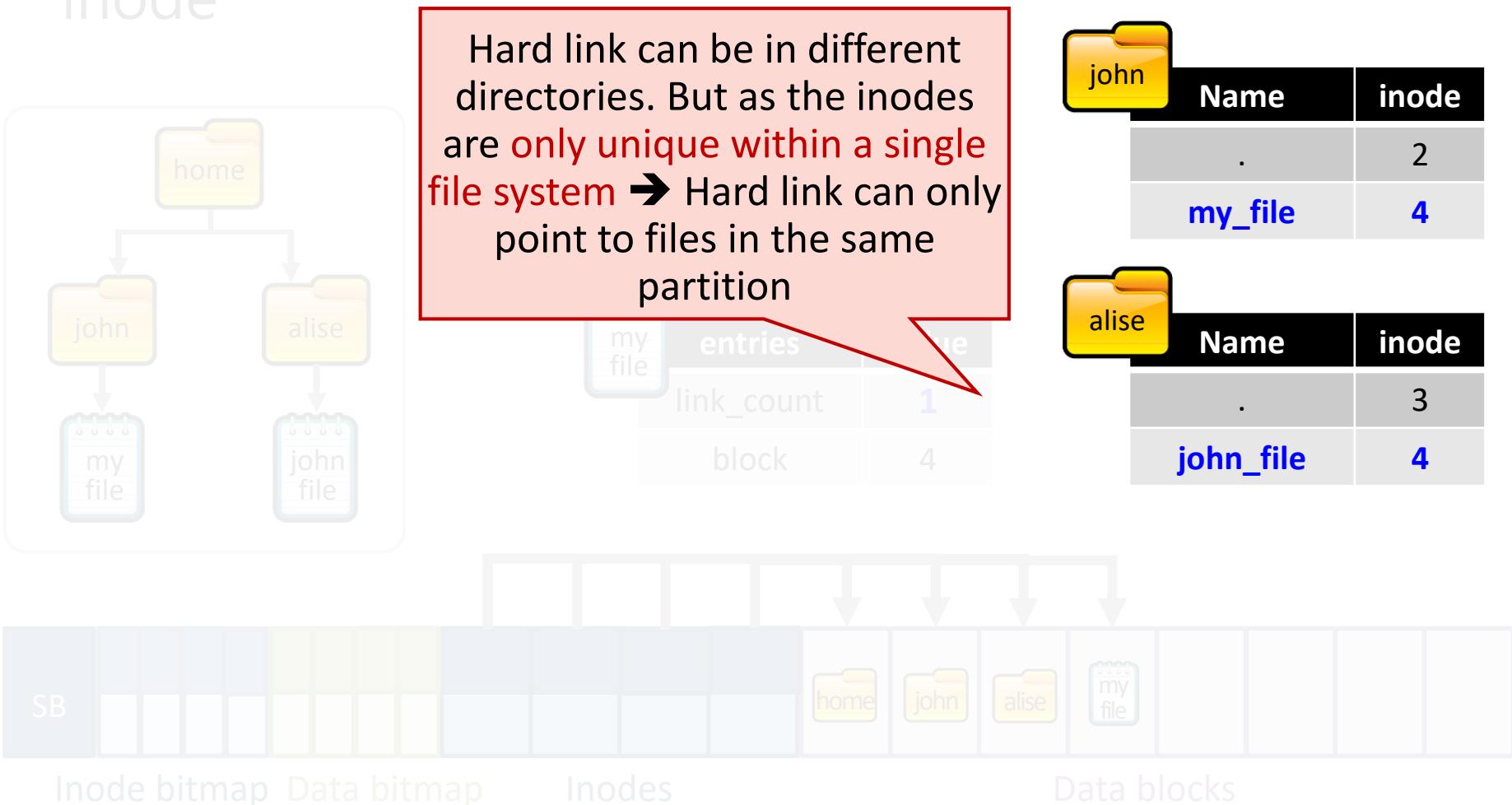
Hard Link

- Multiple directory entries may point to the same inode



Hard Link

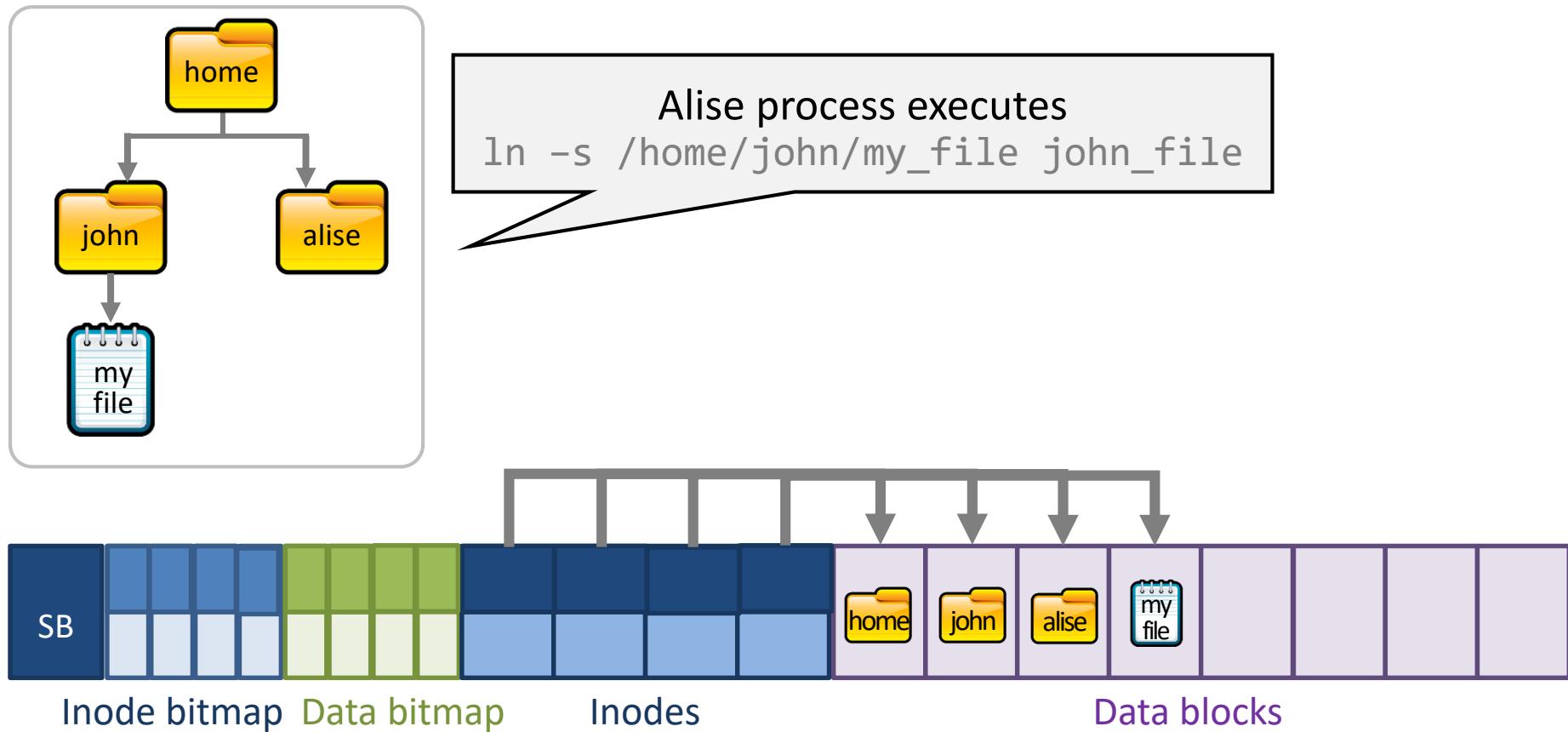
- Multiple directory entries may point to the same inode





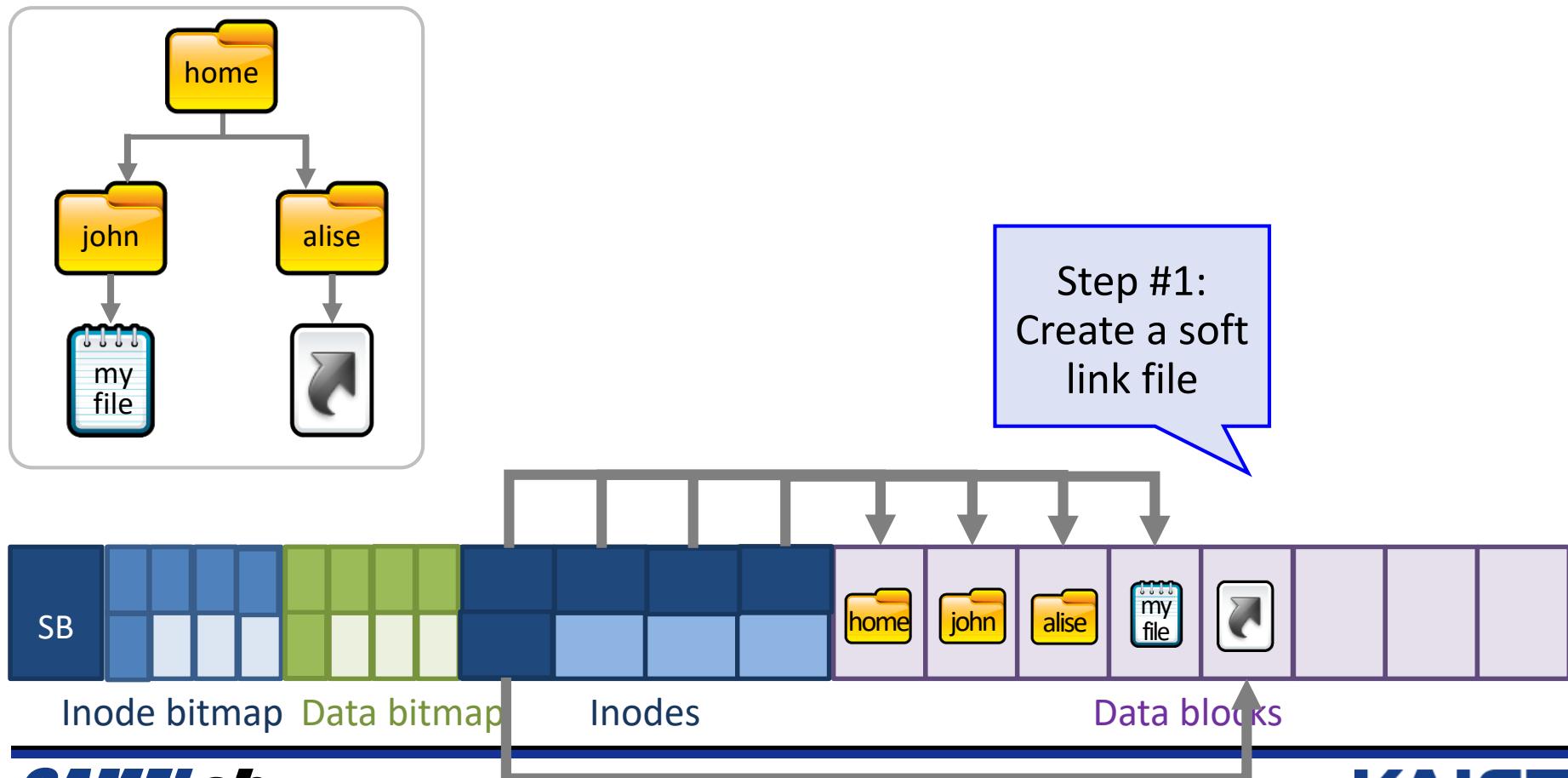
Soft Link

- Soft links are **special files** that include the path to another file (a.k.a, symbolic links, shortcuts)



Soft Link

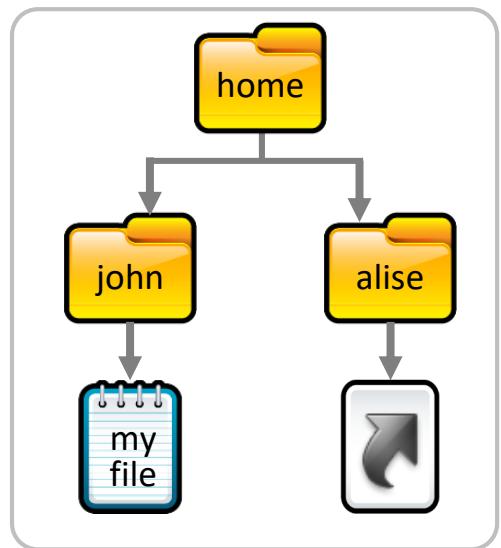
- Soft links are **special files** that include the path to another file (a.k.a, symbolic links, shortcuts)





Soft Link

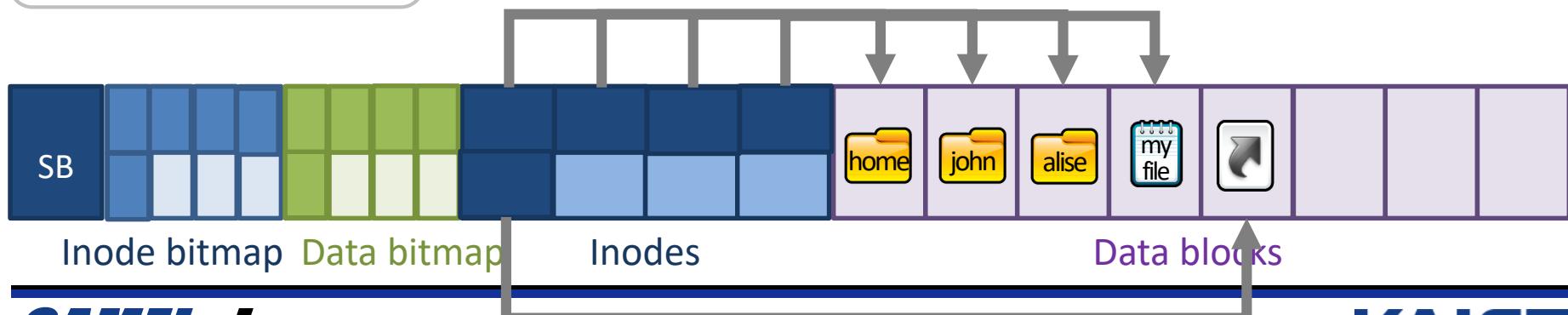
- Soft links are **special files** that include the path to another file (a.k.a, symbolic links, shortcuts)



Step #2:
Add it to the
alise directory

Name	inode
john	2
my_file	4

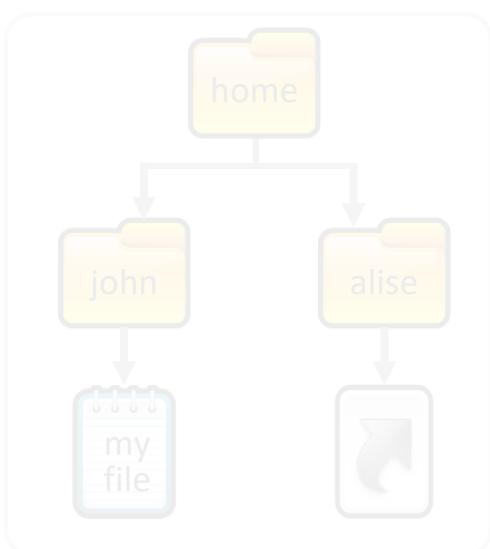
Name	inode
alise	3
john_file	5





Soft Link

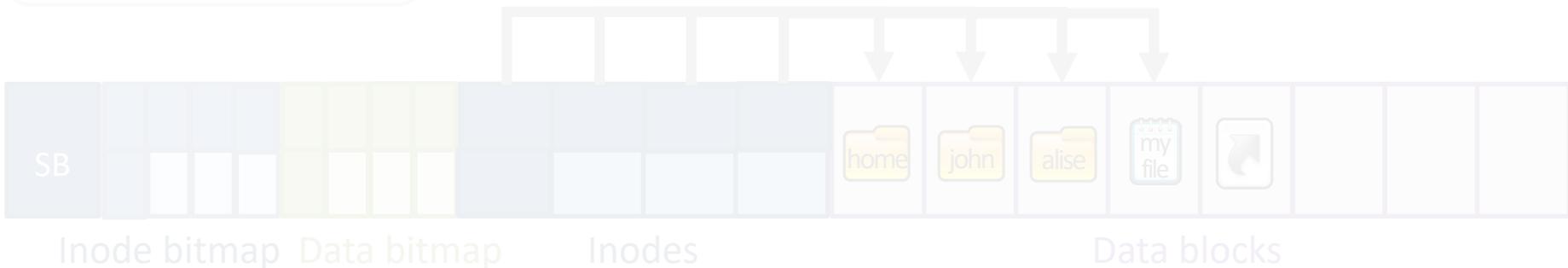
- Soft links are special files that include the path to another file (a.k.a, symbolic links, shortcuts)



As soft link includes the path of linked file, it can support different partition file as well

Name	inode
john	2
my_file	4

Name	inode
alise	3
john_file	5



What's Next?

- At this point, we have a full featured File-System.
- Then what's next? – Let's check **consistency** and **reliability** issue of File-System

21S EE415

File System III

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMELab 

File System III

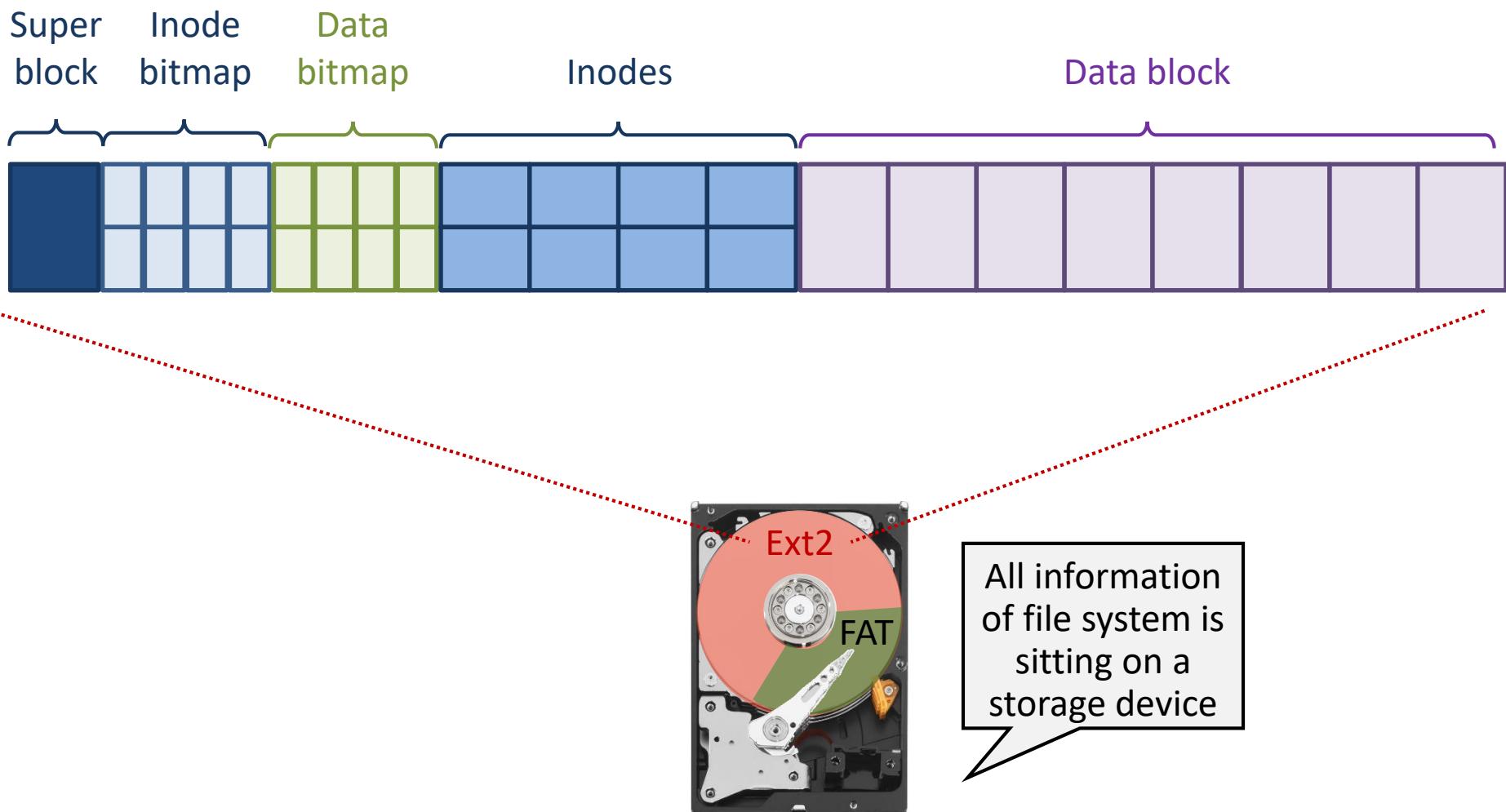
Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

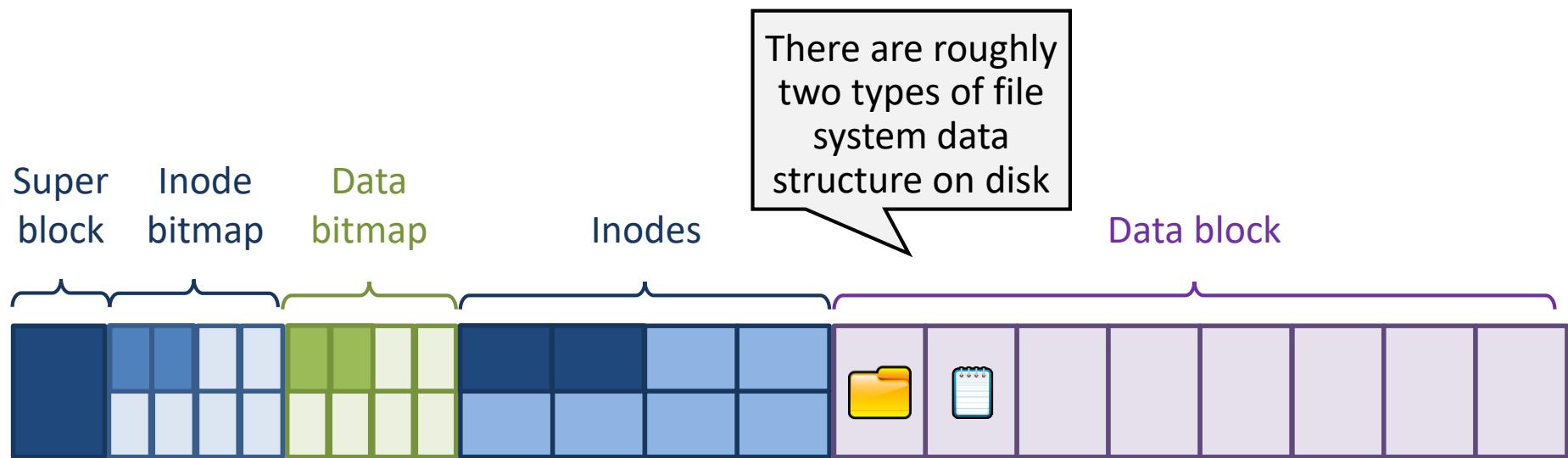
CAMEL*ab* 

[Recap] file system On Disk

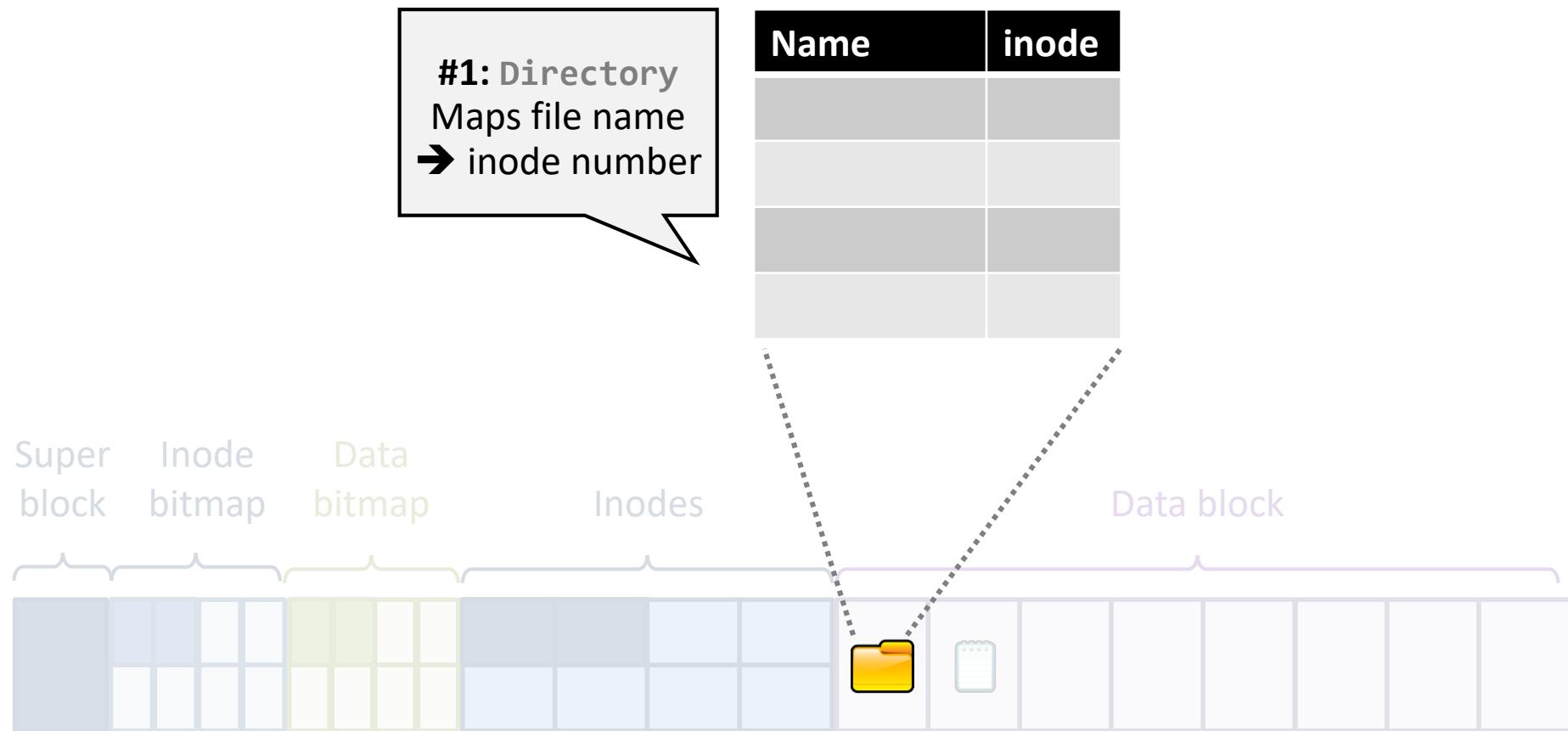




[Recap] On-Disk FS Data Structure

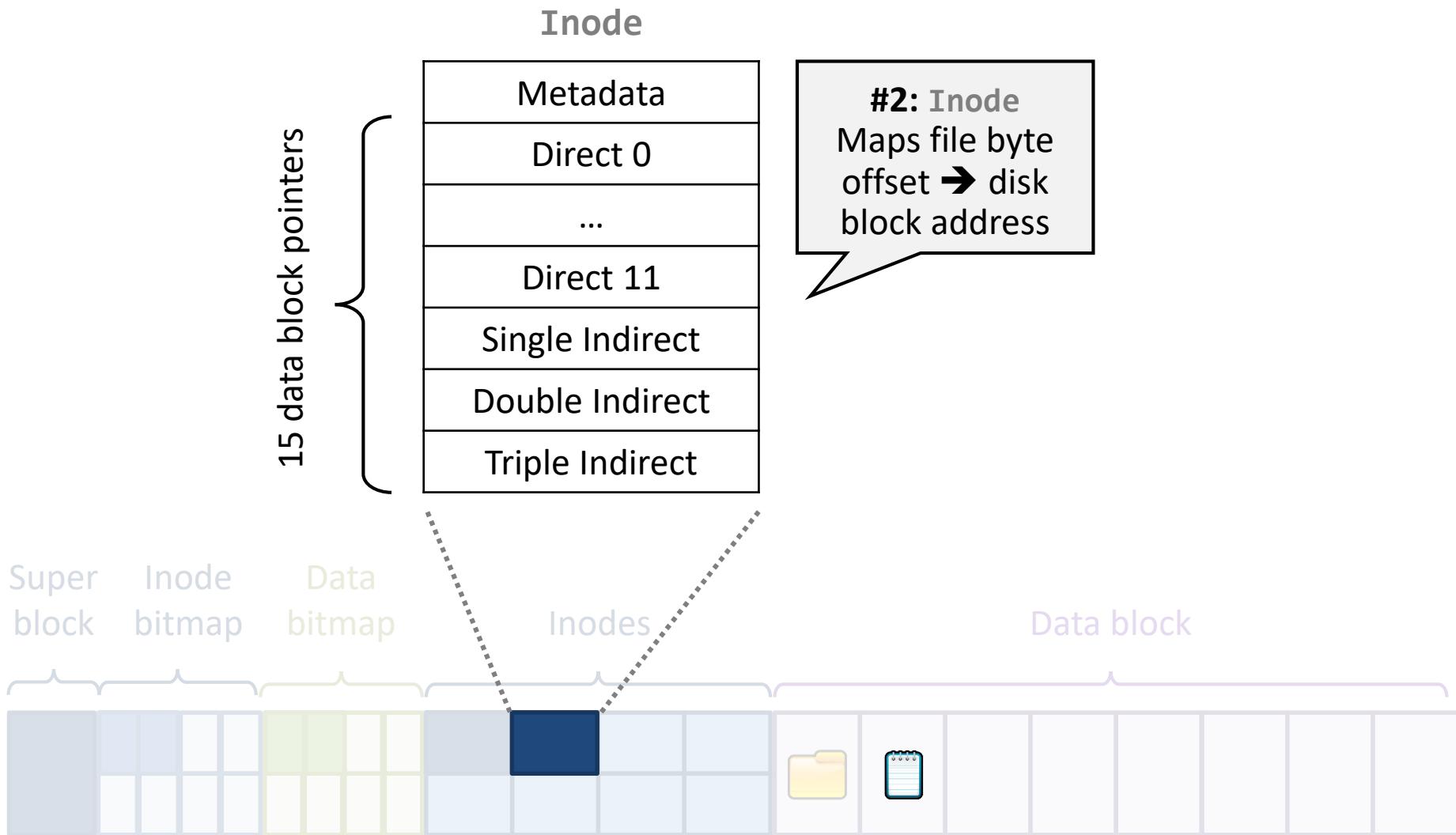


[Recap] On-Disk FS Data Structure

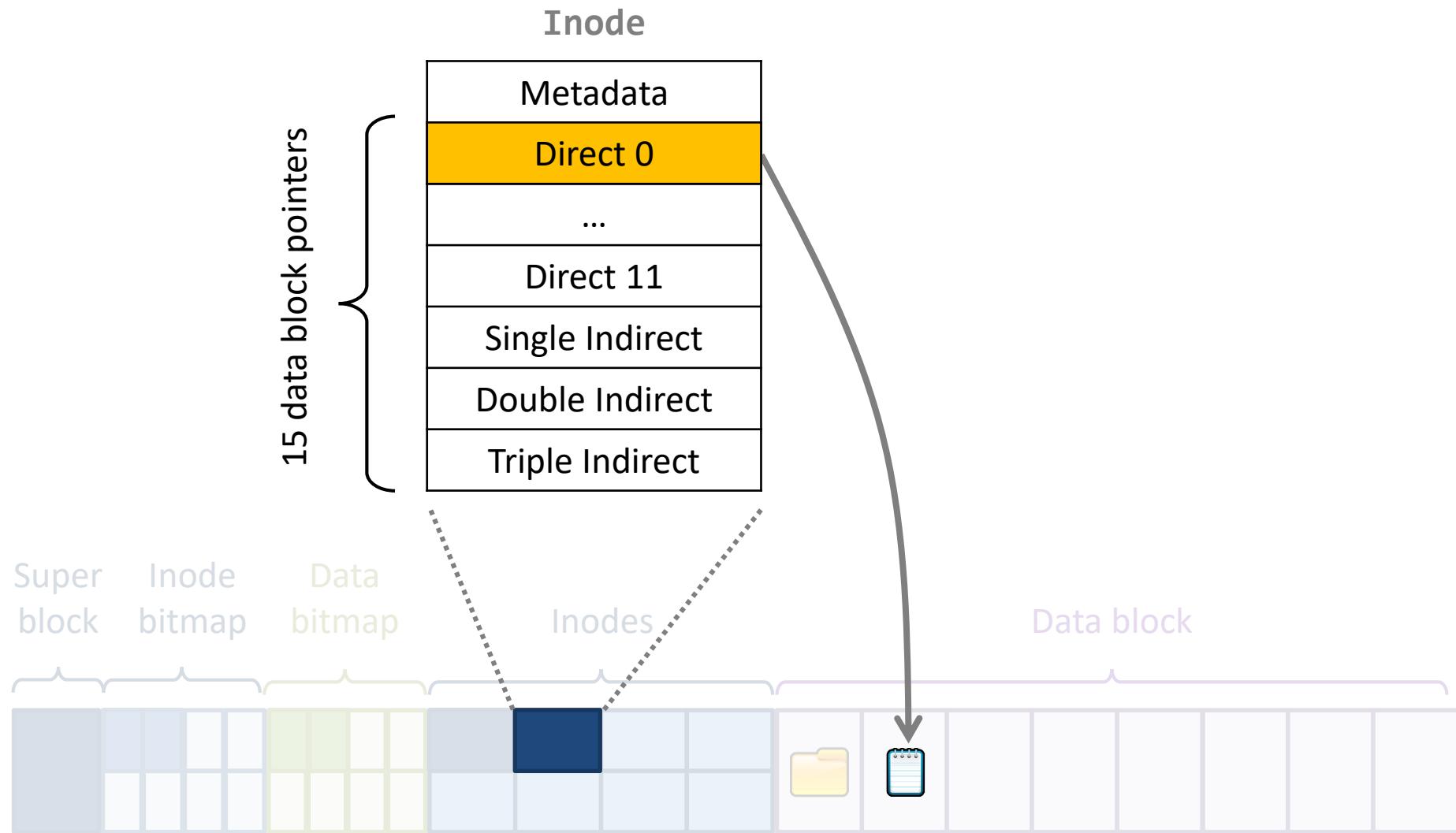




[Recap] On-Disk FS Data Structure

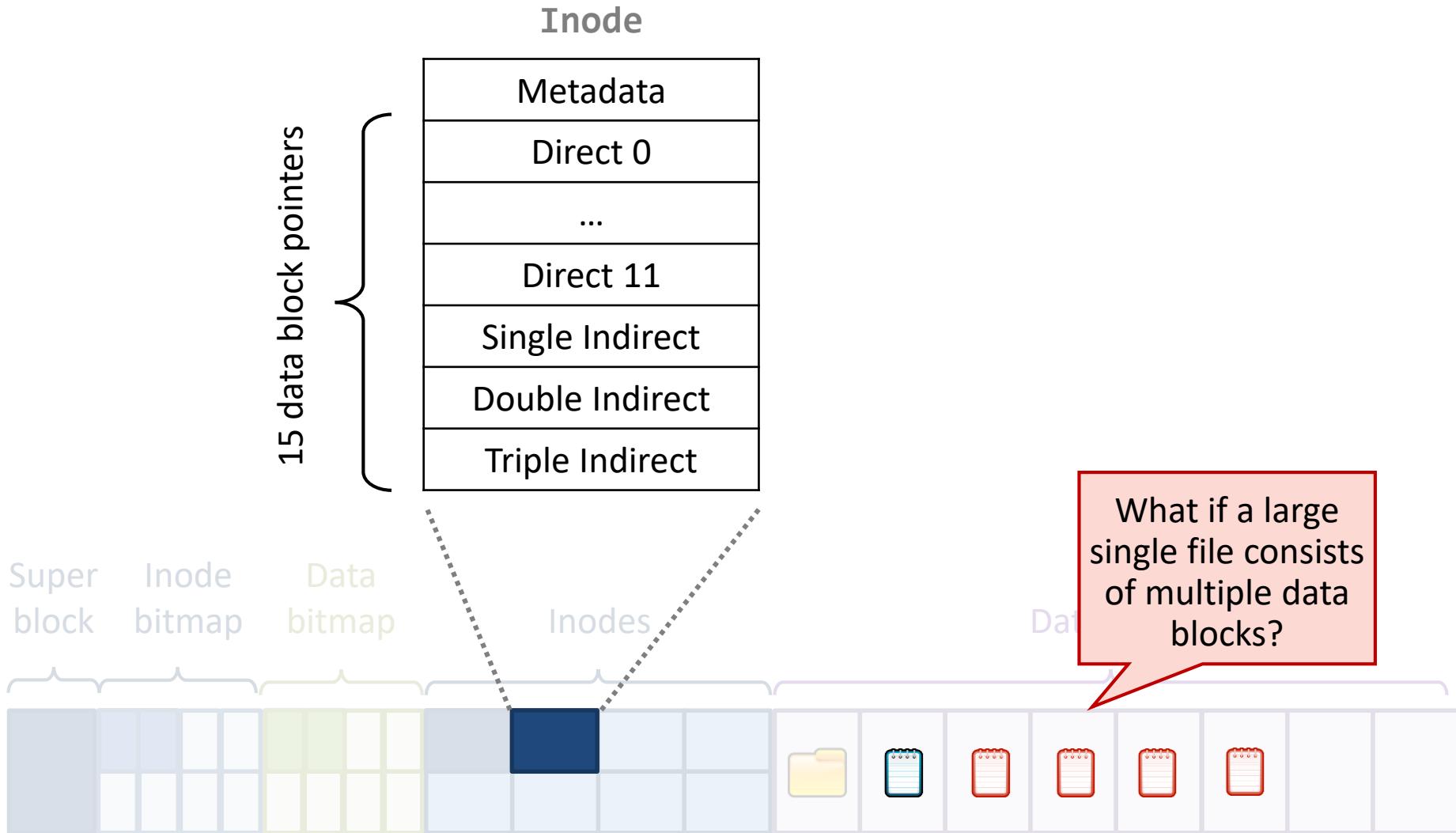


[Recap] On-Disk FS Data Structure



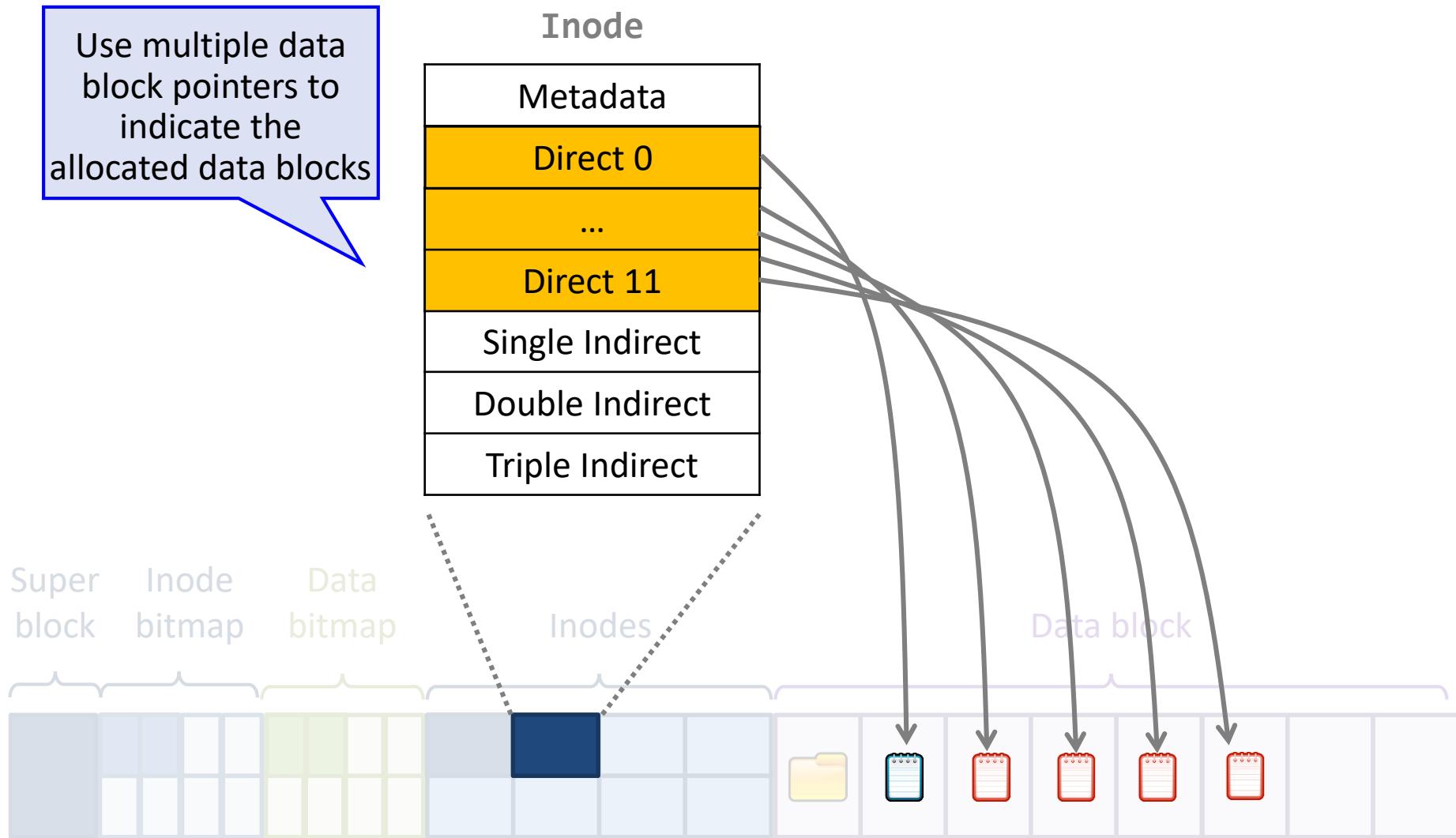


[Recap] On-Disk FS Data Structure





[Recap] On-Disk FS Data Structure



[Recap] file systemOperation

Using a file write (`fwrite`) operation, you can update the essay as many as you want

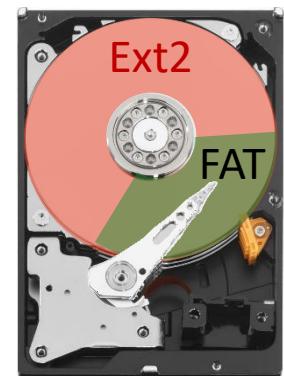
`fwrite`



Will a function call be handle by a single write to the underlying file system?

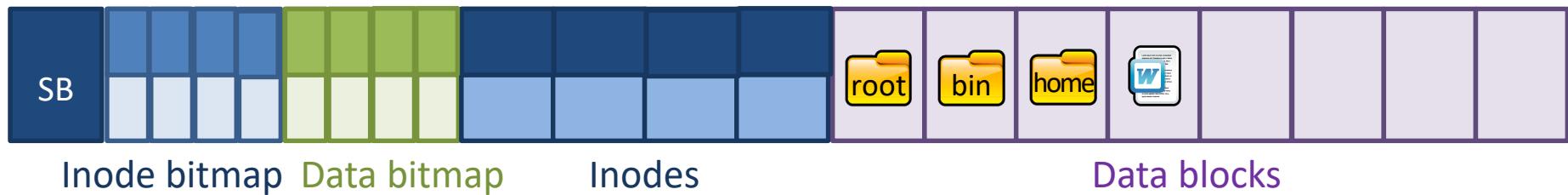
Ext2

FAT



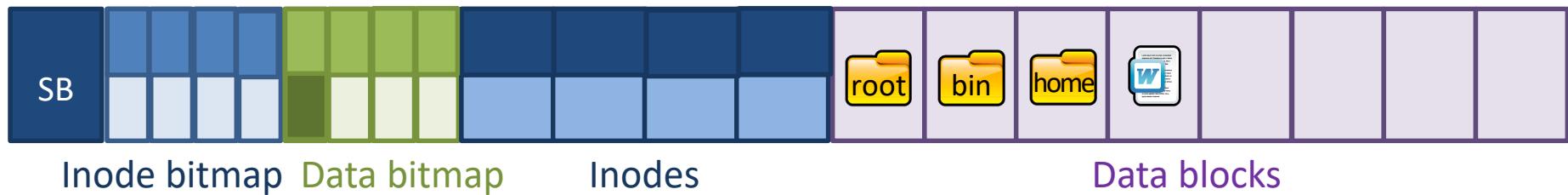
Example: File Append

No! A single `fwrite` call results in multiple, independent writes to the File-System

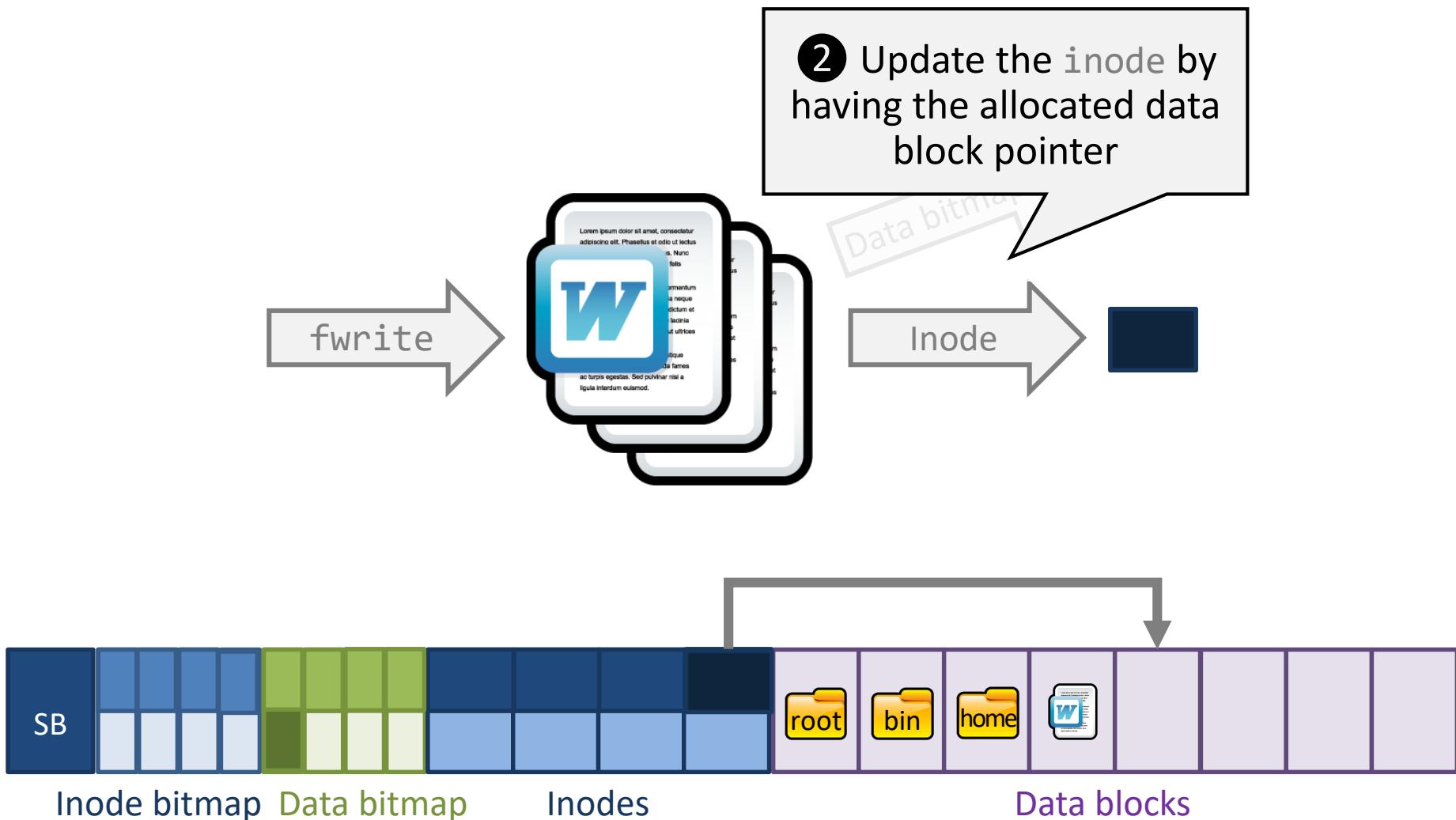


Example: File Append

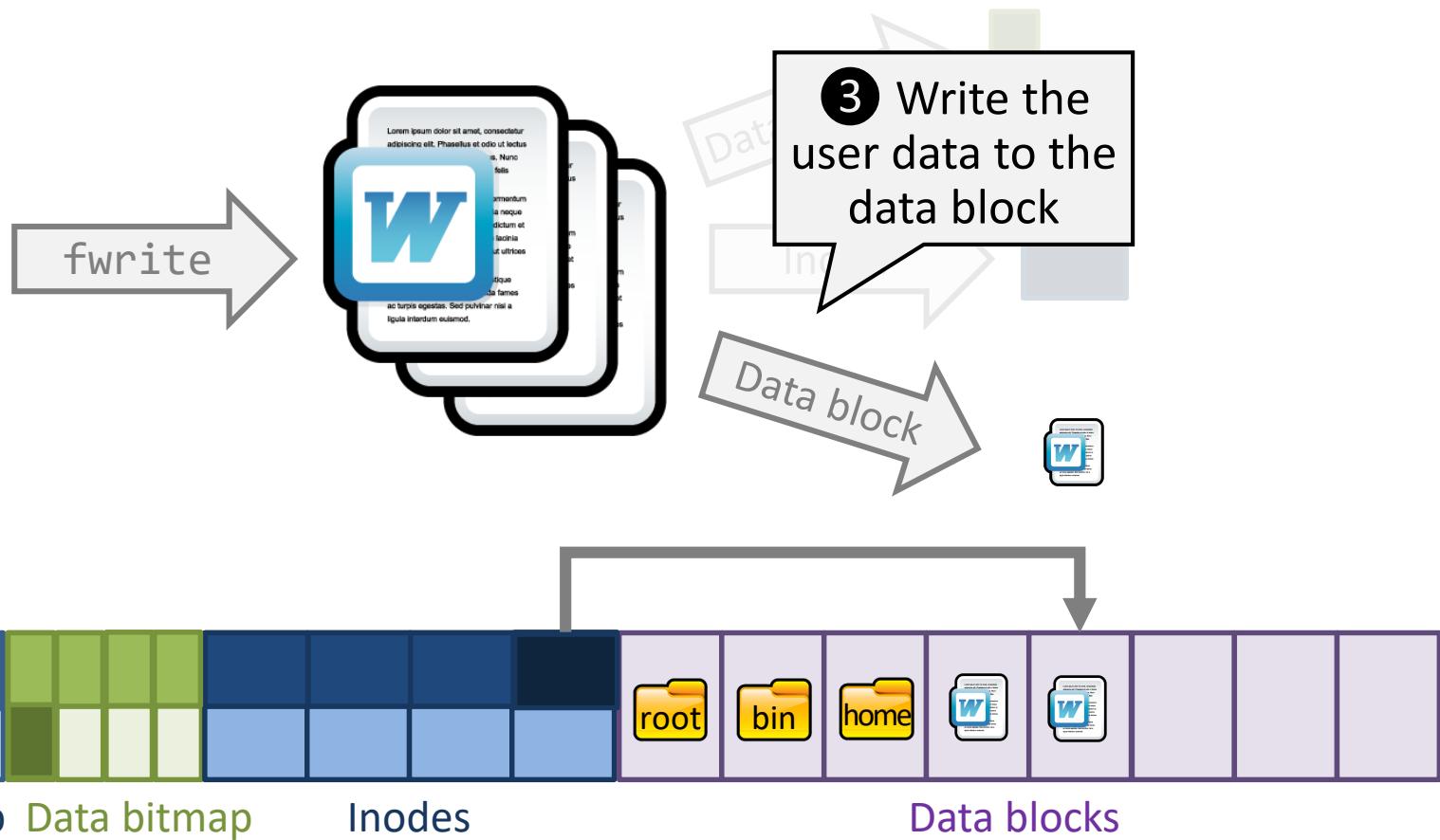
- ① (Check data bitmap and) allocate a free block with an update of the corresponding data bitmap 's bit



Example: File Append



Example: File Append

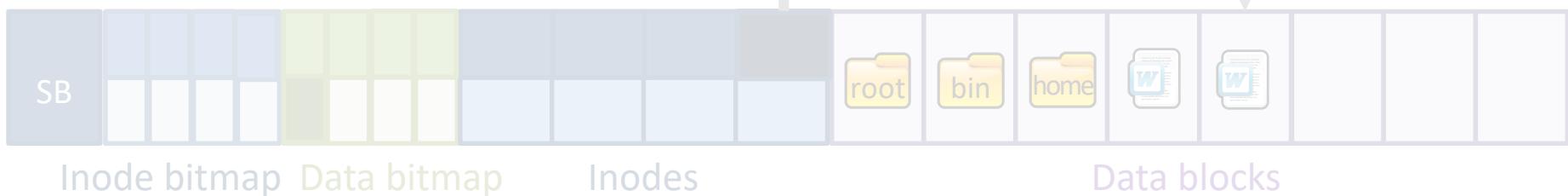
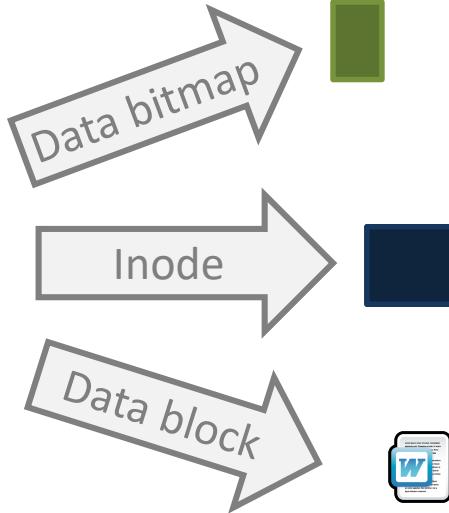


Example: File Append

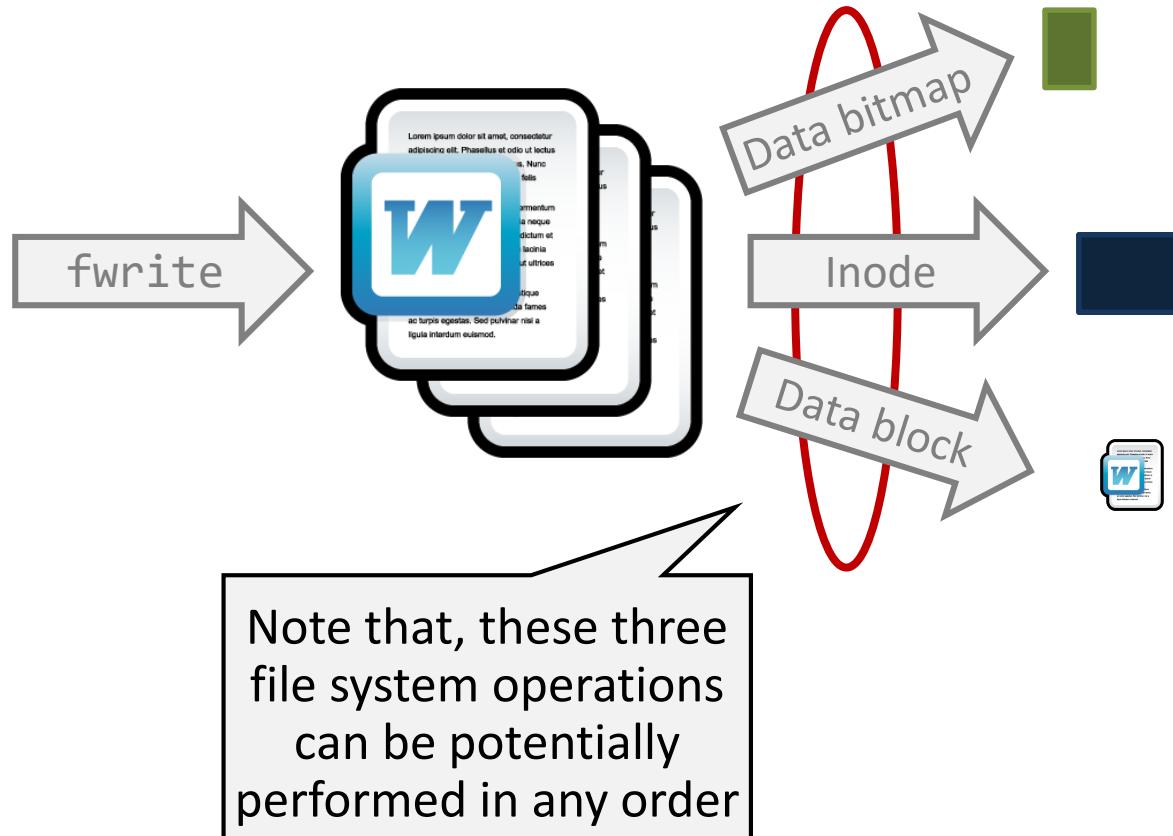
What happens if the computer **crashes** in the middle of this file system writes?



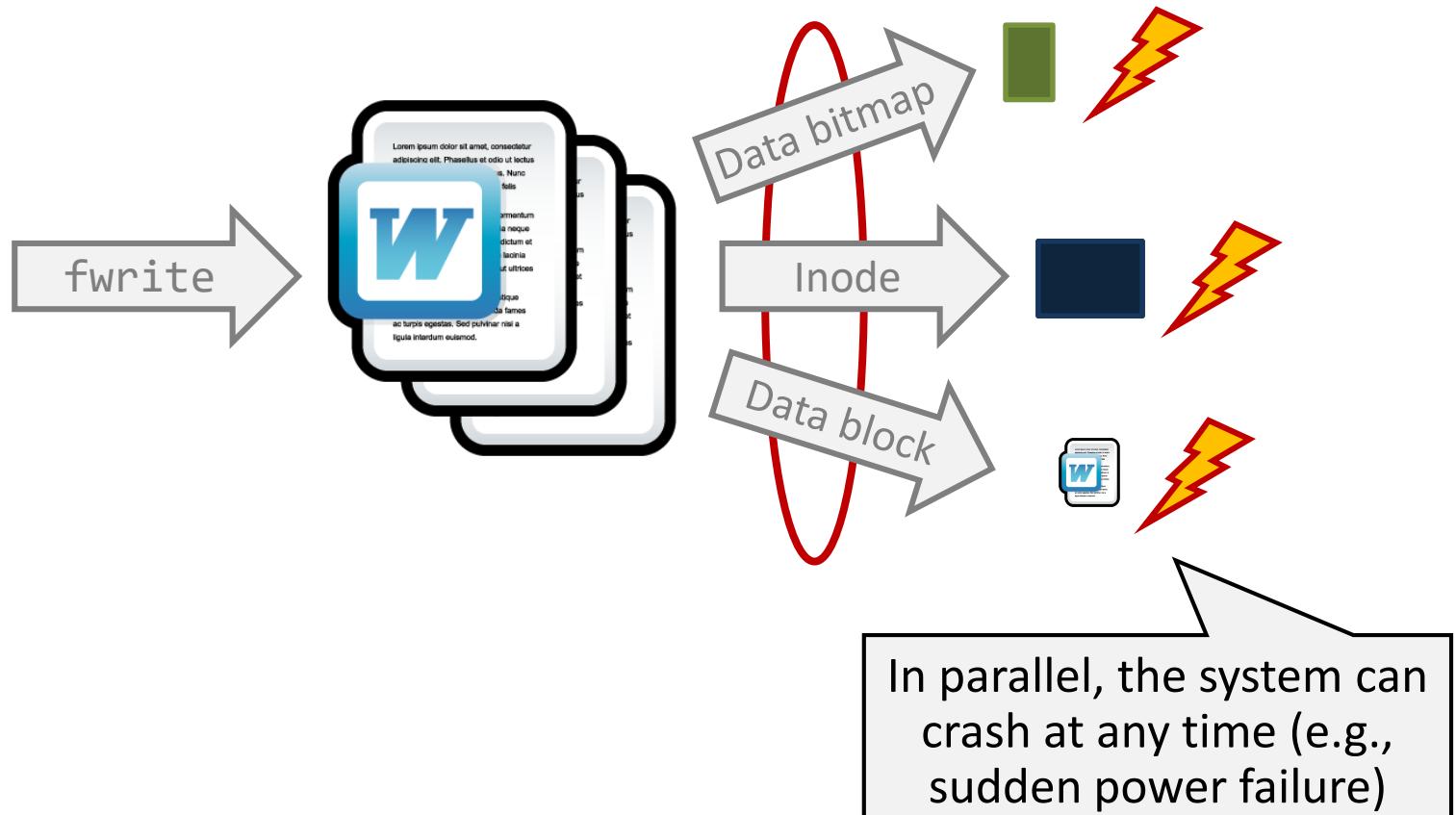
fwrite



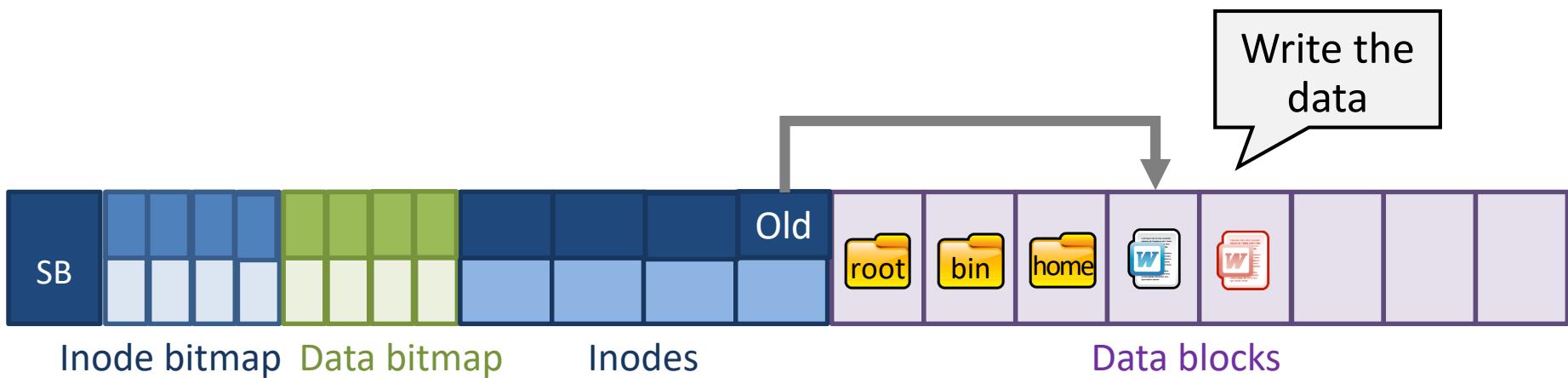
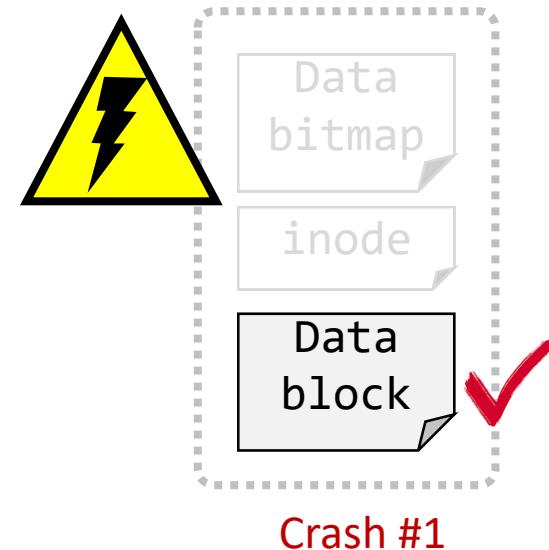
The Crash Consistency Problem



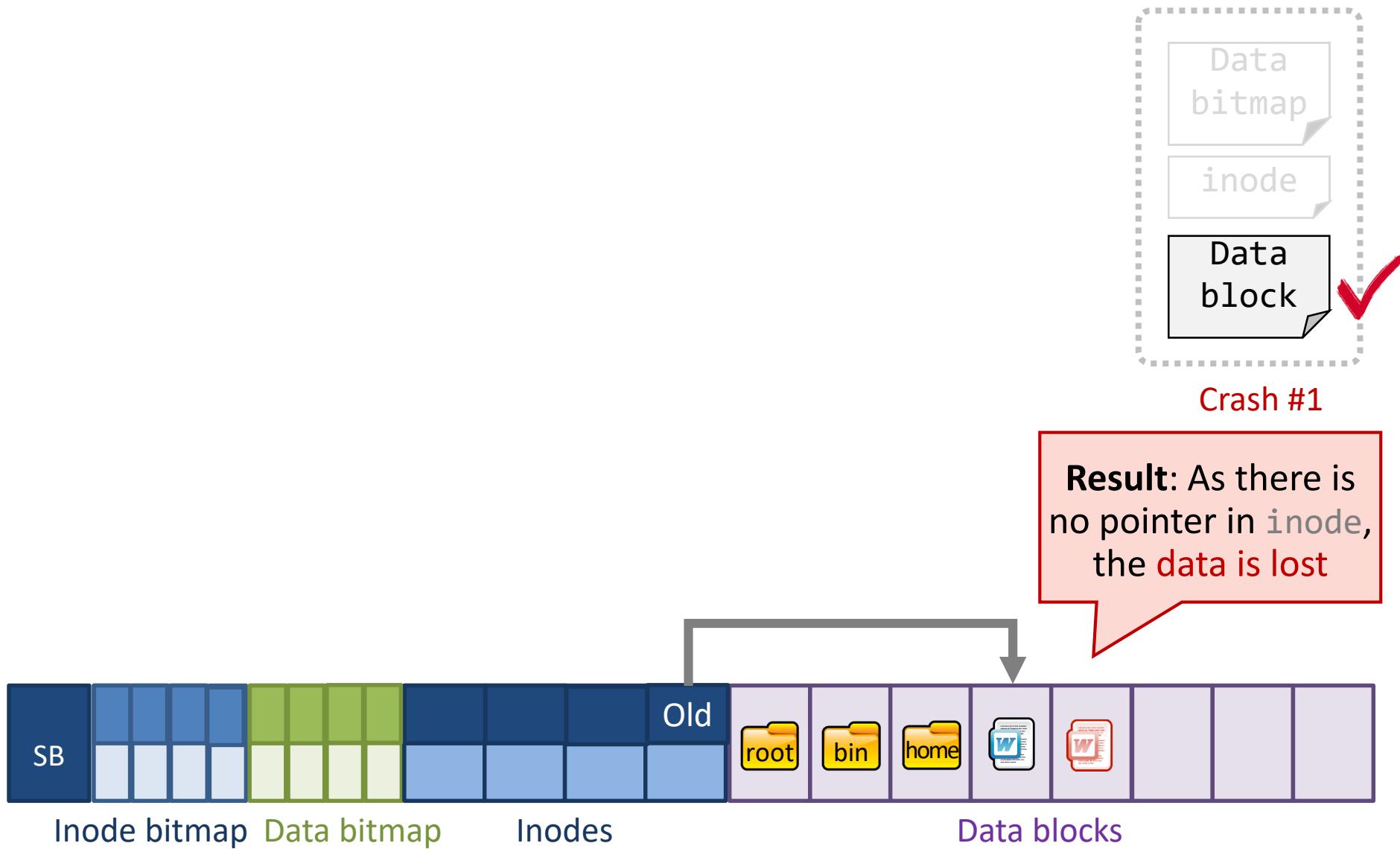
The Crash Consistency Problem



Crash #1: Crash After data block Update

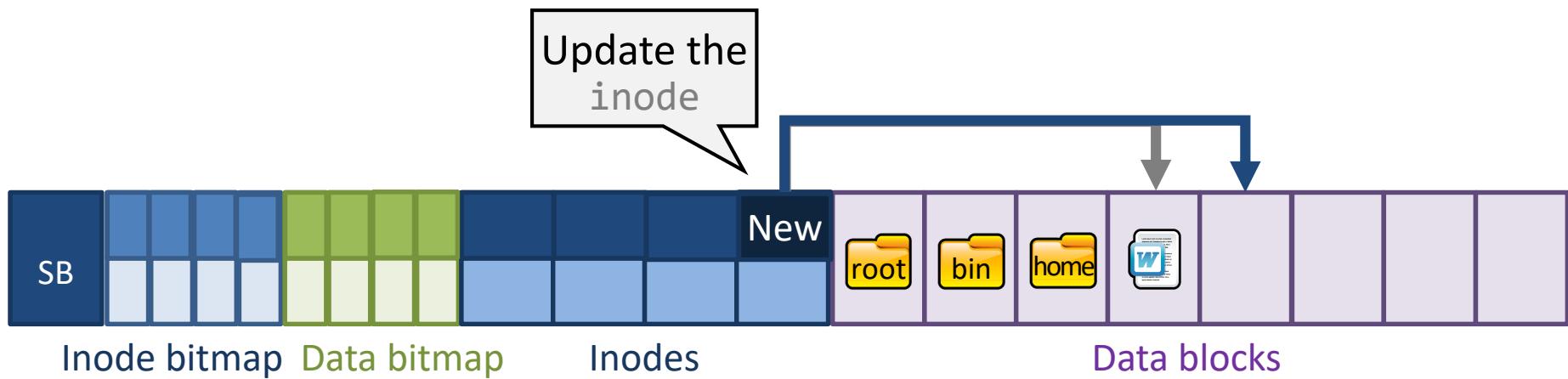
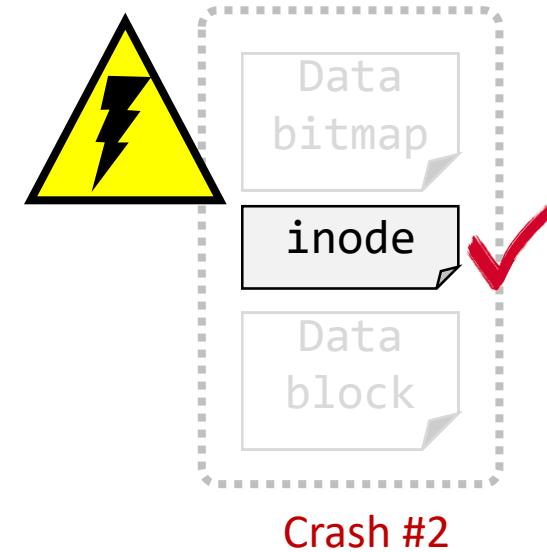


Crash #1: Crash After data block Update



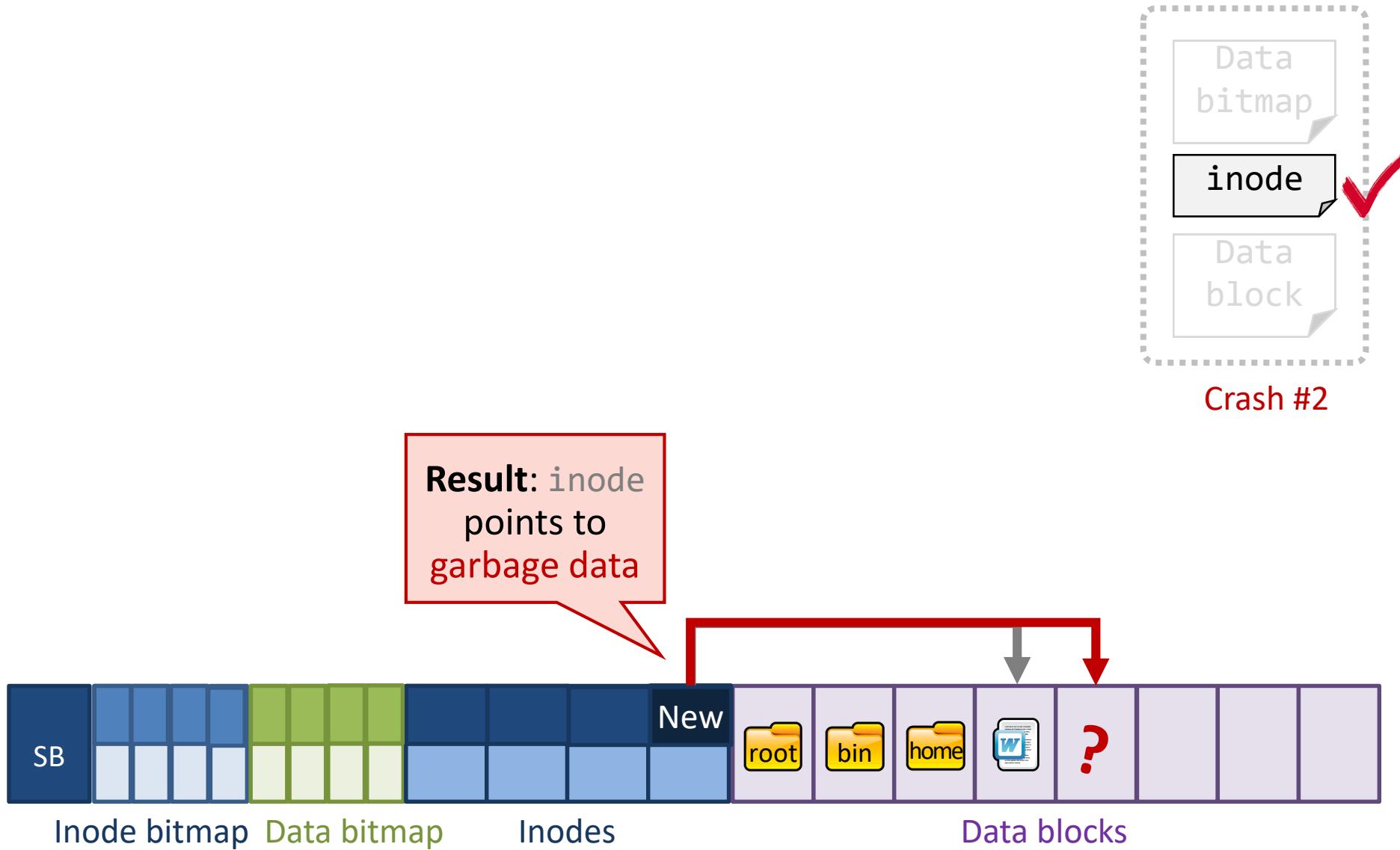


Crash #2: Crash After inode Update

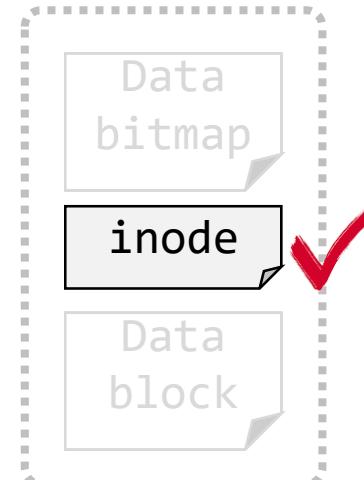




Crash #2: Crash After inode Update



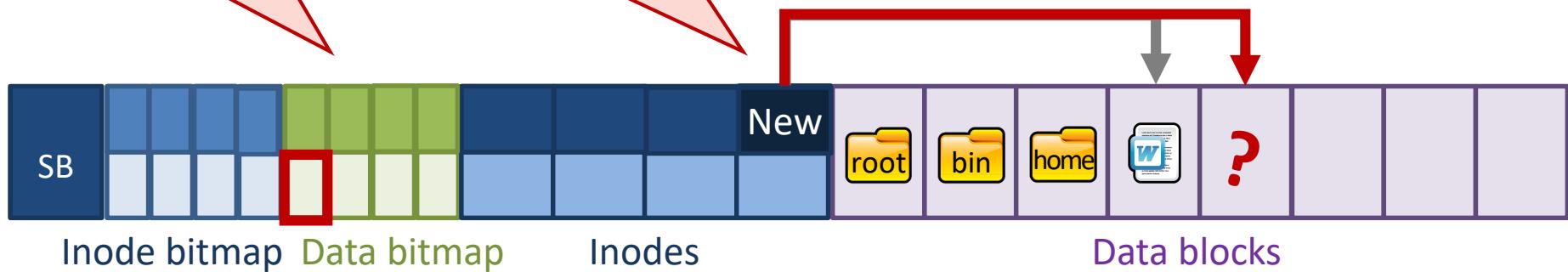
Crash #2: Crash After inode Update



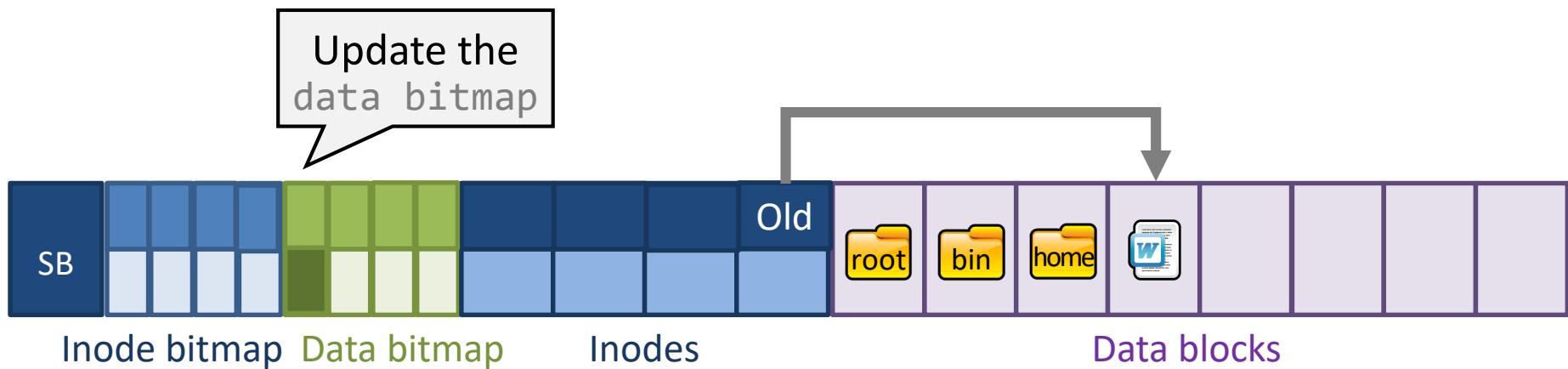
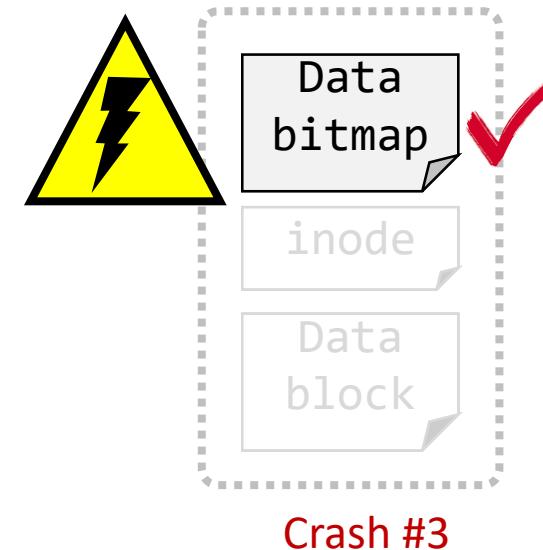
data bitmap  inode

However, the
data block 4 isn't
yet allocated in
data bitmap

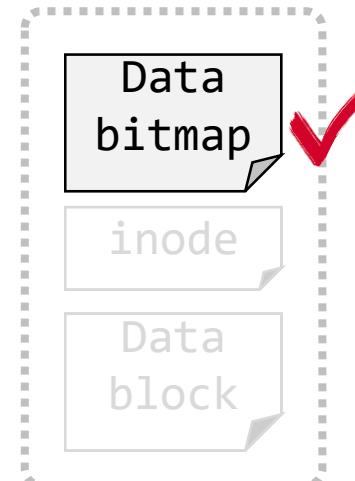
File system falls in
inconsistent state!
inode points data
block 4 (allocated)



Crash #3: Crash After data bitmap Update

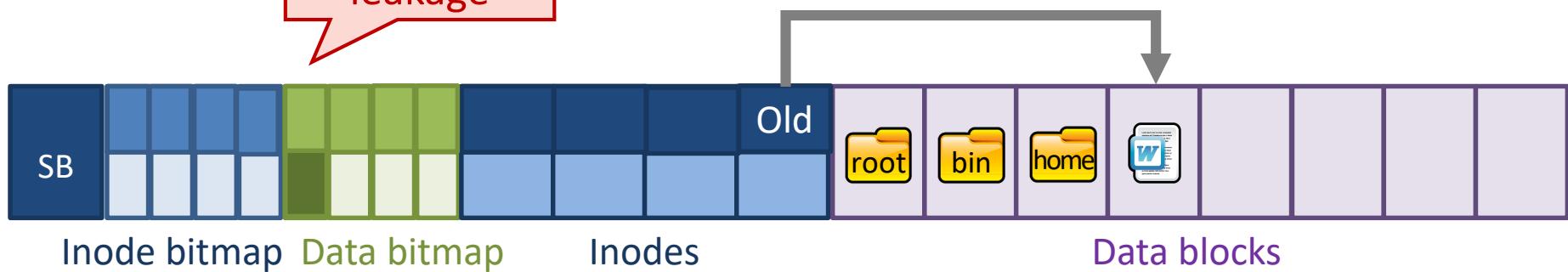


Crash #3: Crash After data bitmap Update

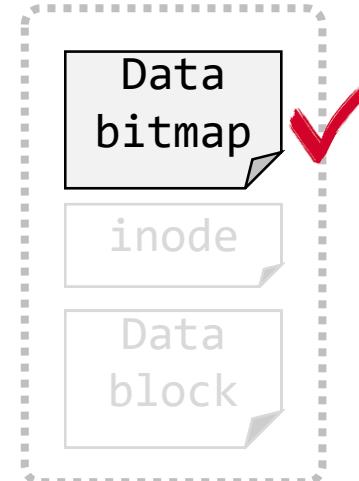


Crash #3

Result: space leakage



Crash #3: Crash After data bitmap Update

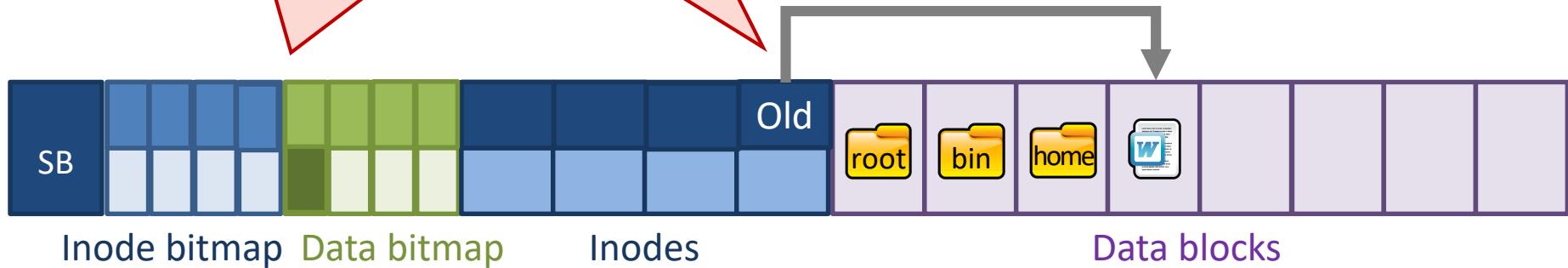


data bitmap  inode

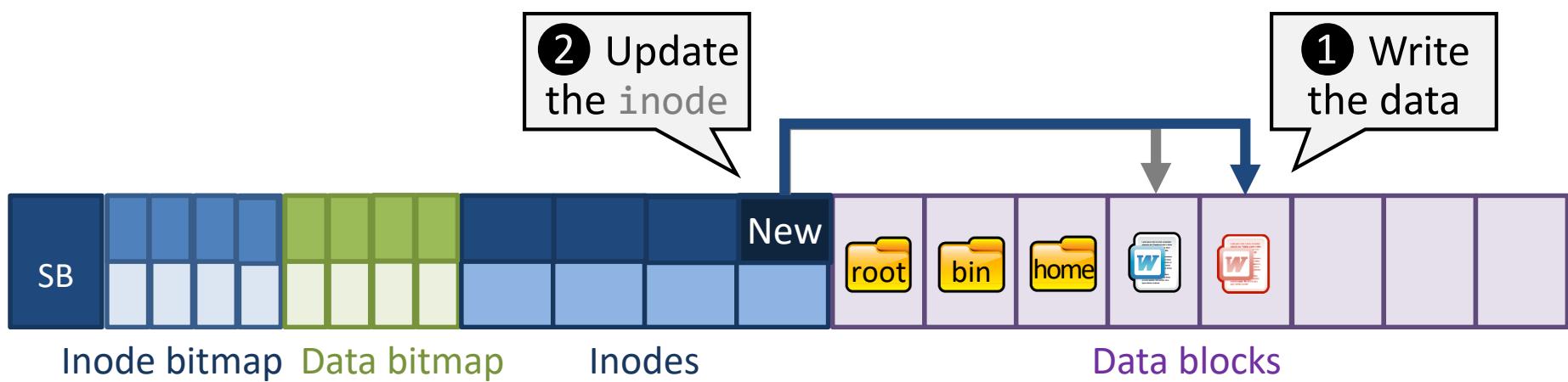
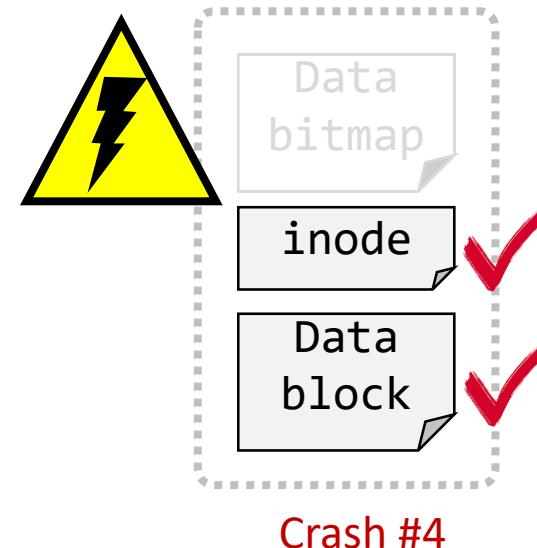
File system becomes inconsistent! The data block 4 for data bitmap is allocated

However, inode
Has no pointer to the block 4

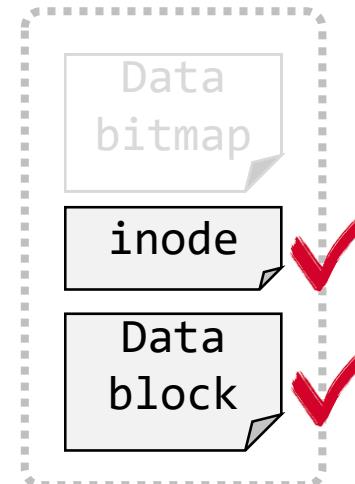
Crash #3



Crash #4: Crash Before data bitmap Update

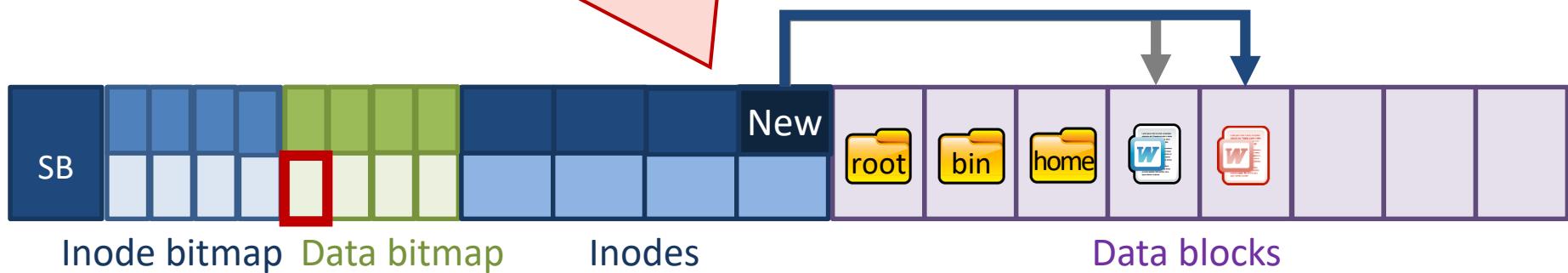


Crash #4: Crash Before data bitmap Update

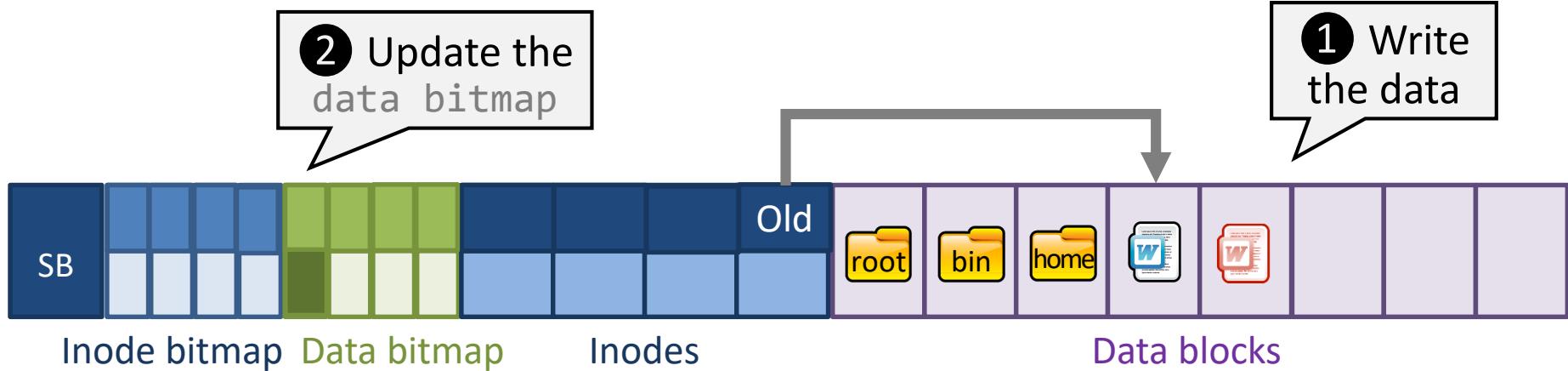
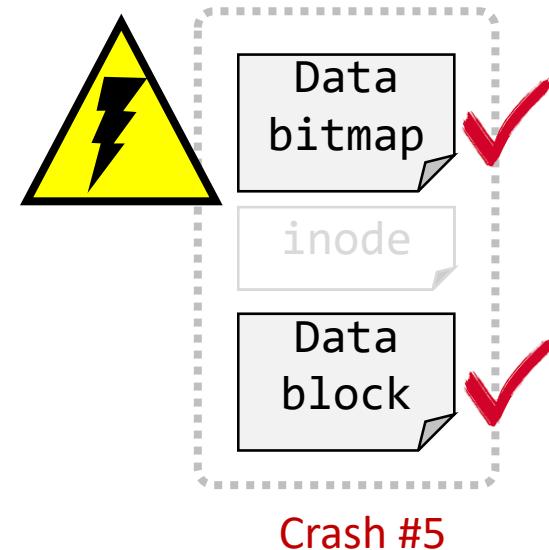


Crash #4

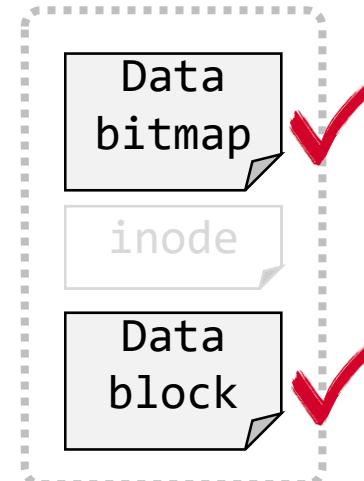
Result: inode points to the new data, but file system is **inconsistent**. Note that data bitmap has no idea of data block 4, and thus it can be overwritten in soon or later



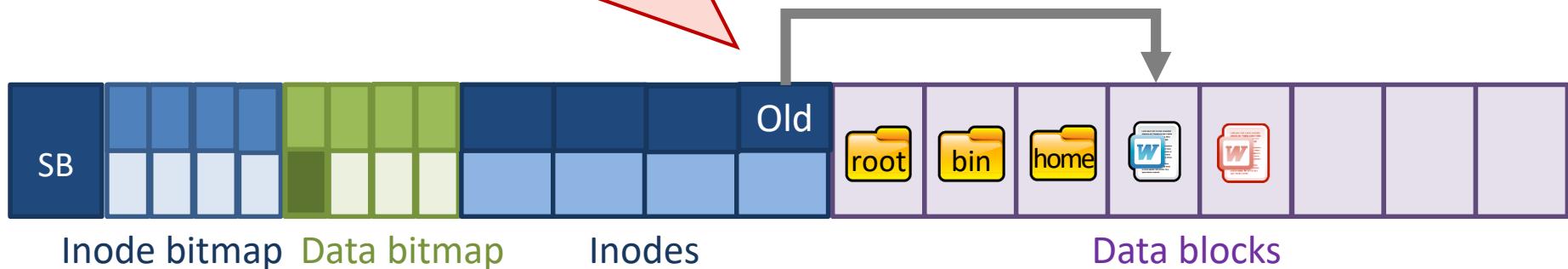
Crash #5: Crash Before inode Update



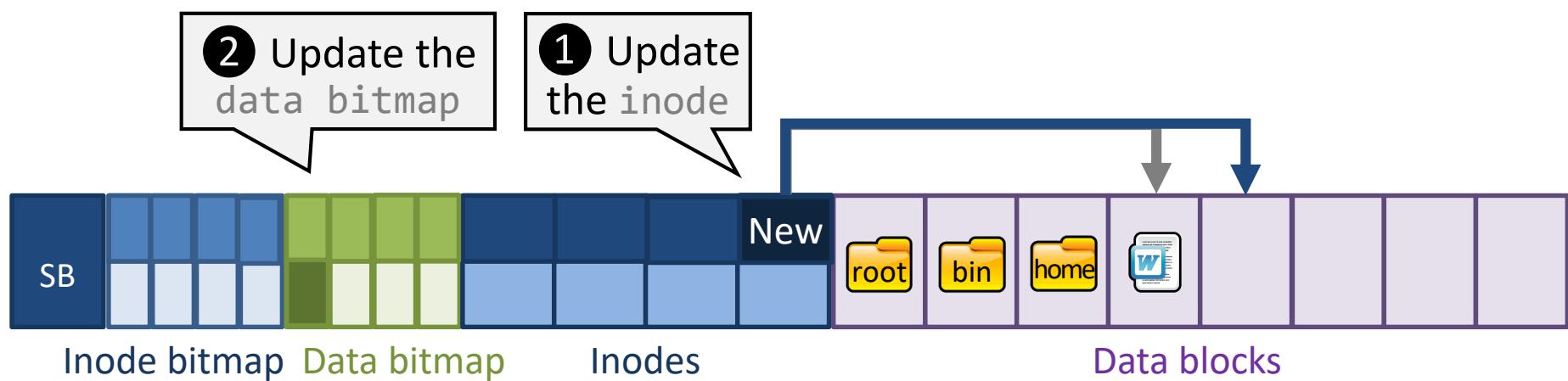
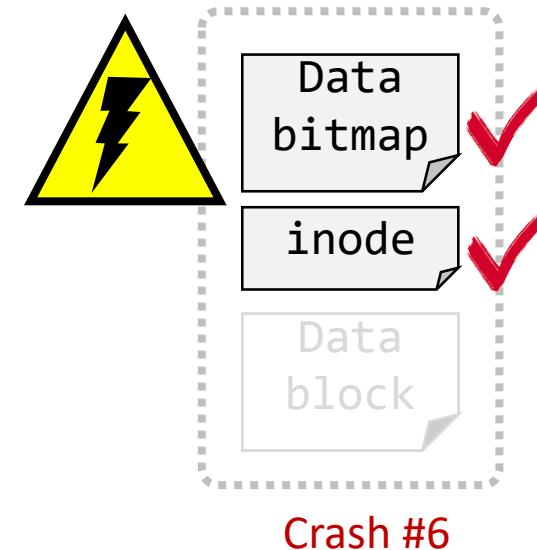
Crash #5: Crash Before inode Update



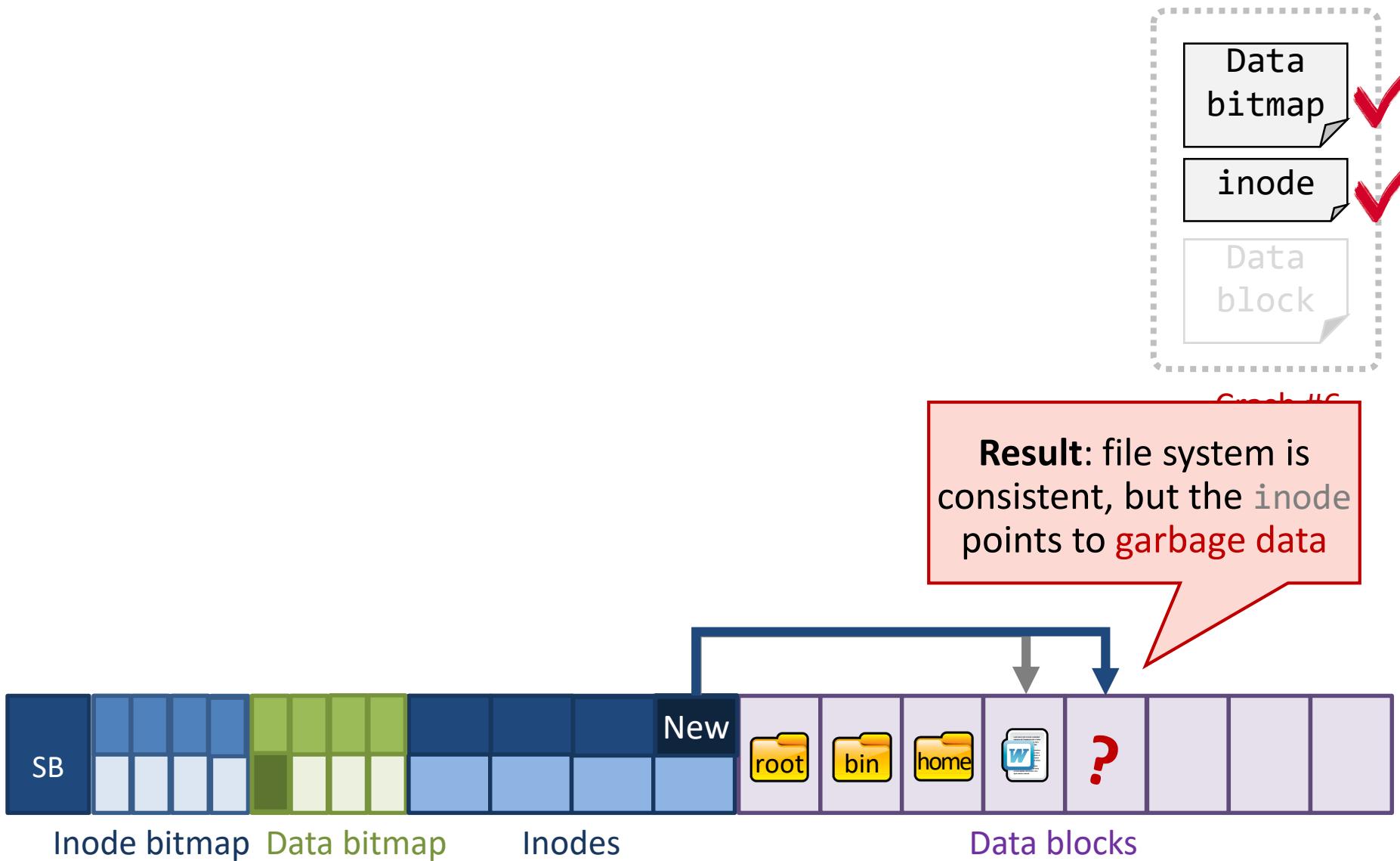
Result: file system is **inconsistent**,
and the data is dangling as **inode**
isn't have reference information
about the block



Crash #6: Crash Before data block Update



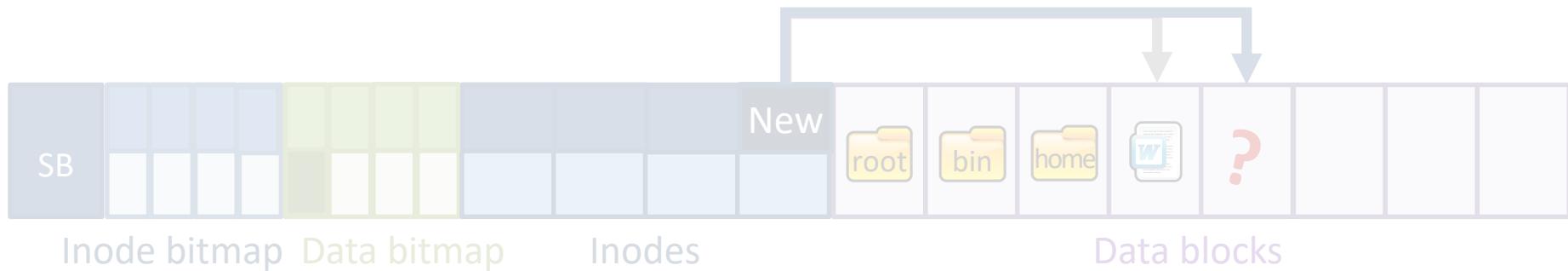
Crash #6: Crash Before data block Update



Crash #6: Crash Before data block Update

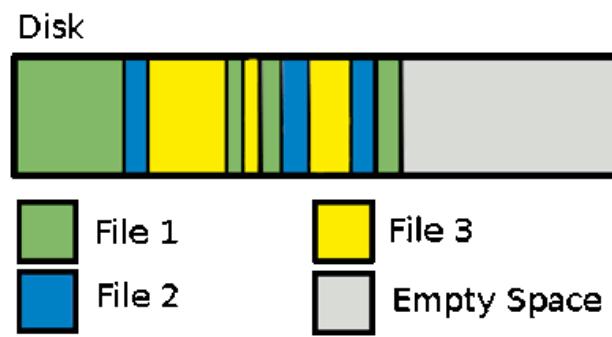
There would be more than complicated situation like a crash happens in between something two

How can we then ensure the file system consistency after the system sees a power failure?



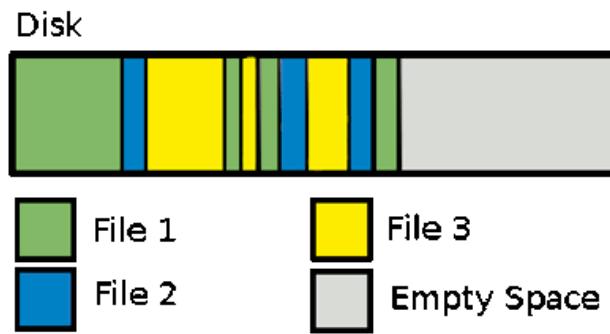
Solution #1: File System Checker (fsck)

Let's accept that the file system can be inconsistent after a crash



Key Idea: Fix During Bootup

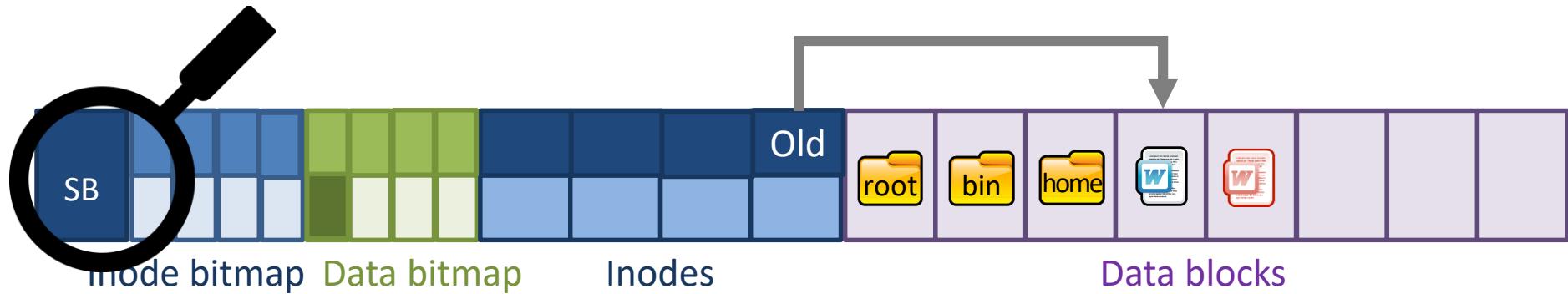
Run a program to fix
the file system during
bootup



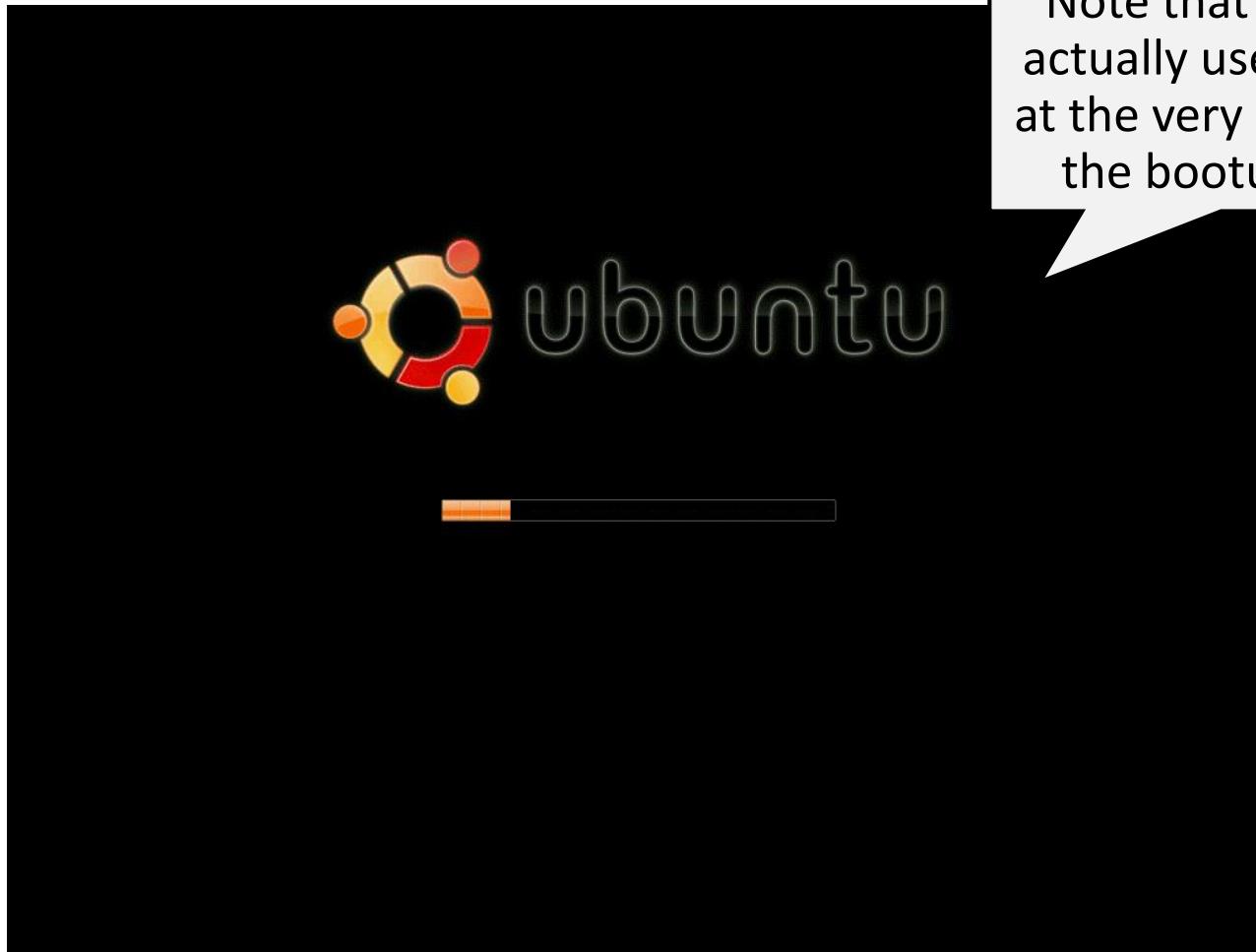


Utility: fsck (Chkdsk on Windows)

During bootup, fsck scans the entire file system “multiple times” to identify and correct the inconsistencies

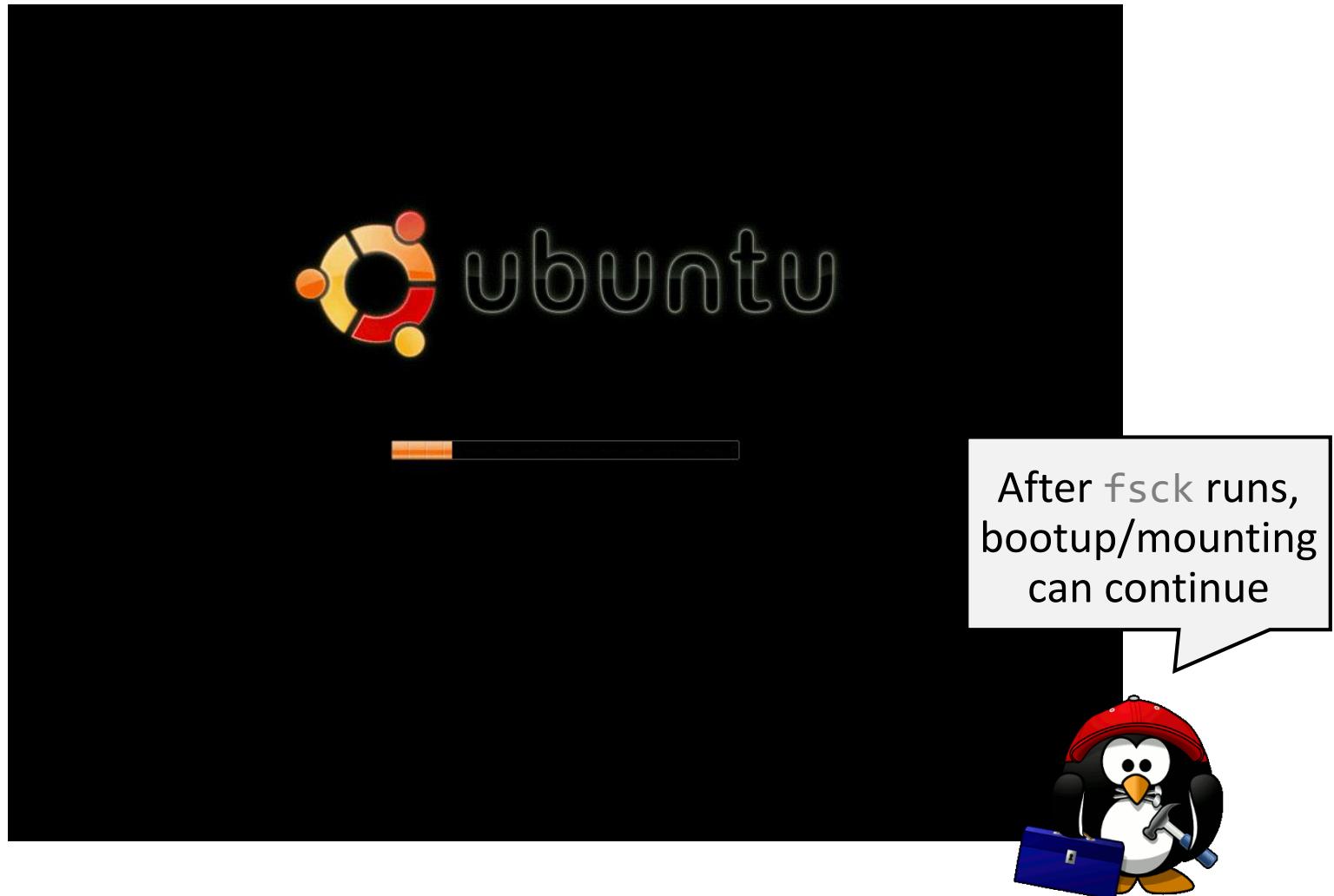


Why During Bootup?



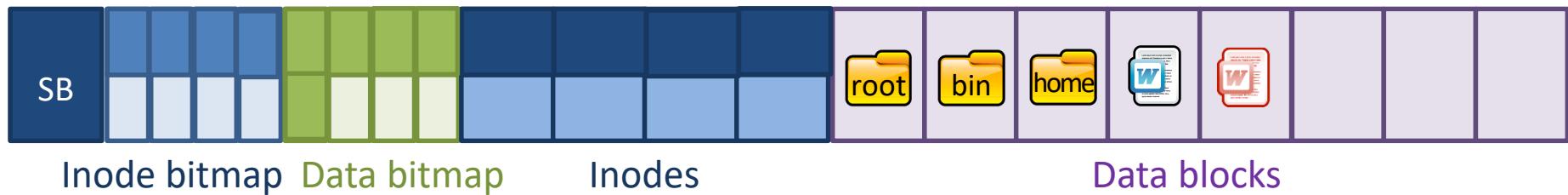
Note that we are not actually use file system at the very beginning of the bootup process

Why During Bootup?



fsck Tasks

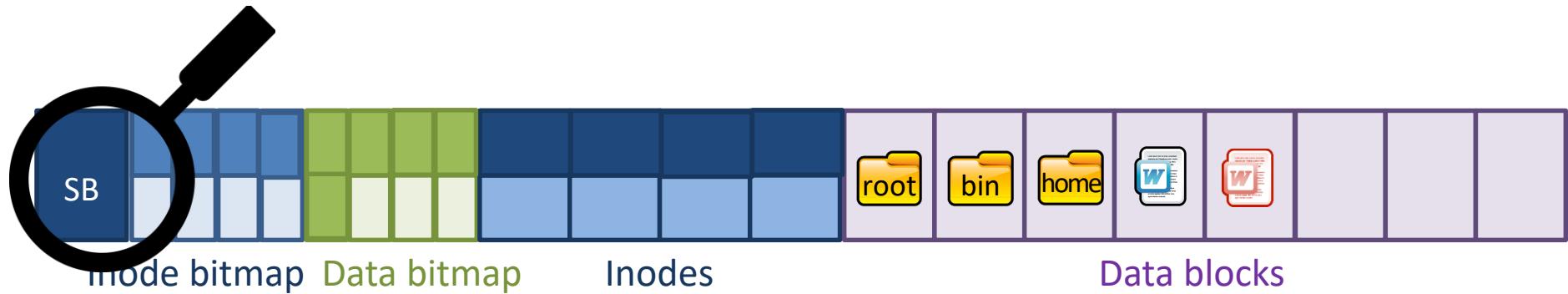
fsck is quite complicated process, but let's classify its tasks into five tasks





fsck Task #1: Superblock

Validate the superblock,
replace it with a backup if
it is corrupted



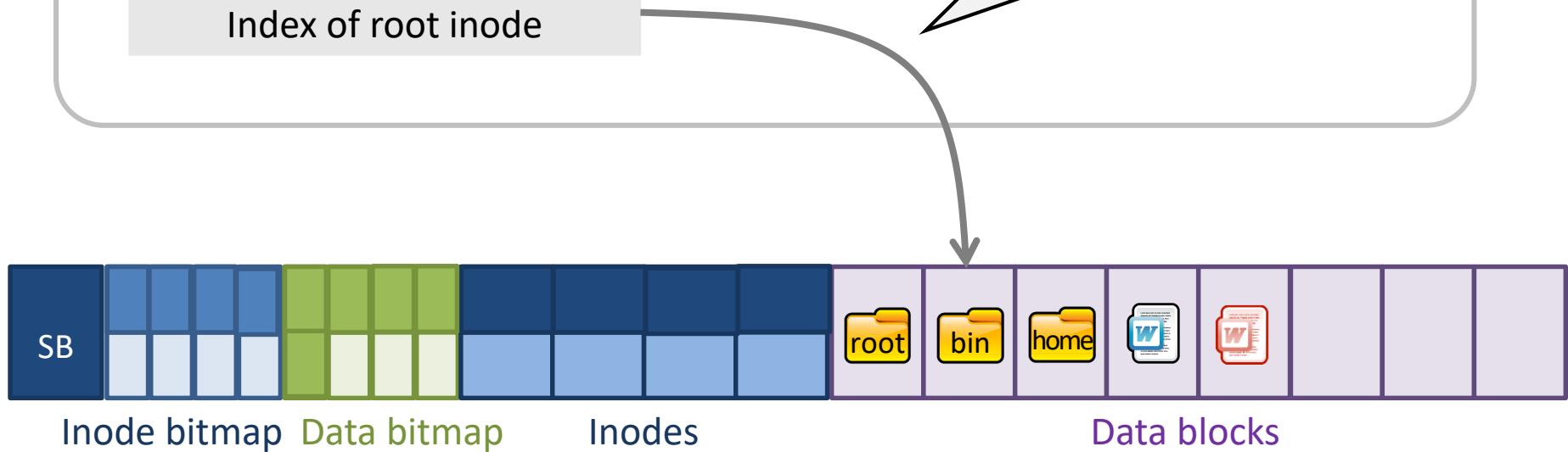


fsck Task #1: Superblock



Superblock
file systemversion
Size of inode bitmaps
Location of inode bitmaps
Size of data bitmaps
Location of data bitmaps
Index of root inode

fsck chases the index
of root inode and
checks the correctness



fsck Task #1: Superblock



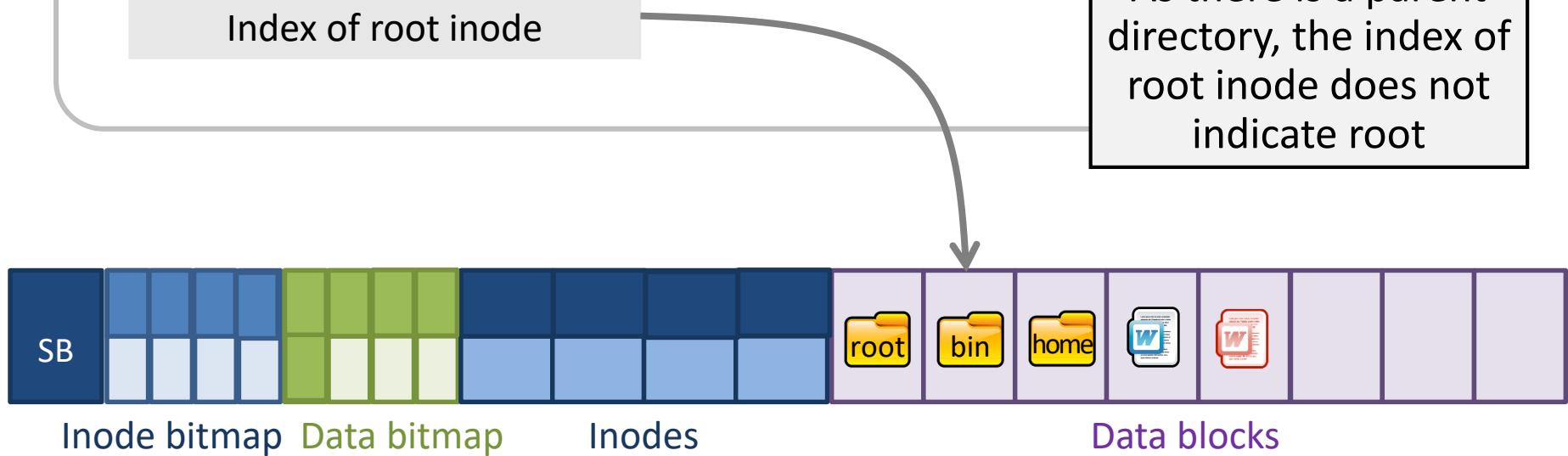
Superblock
file systemversion
Size of inode bitmaps
Location of inode bitmaps
Size of data bitmaps
Location of data bitmaps
Index of root inode



Directory

Name	inode
..	0
.	1

As there is a parent directory, the index of root inode does not indicate root



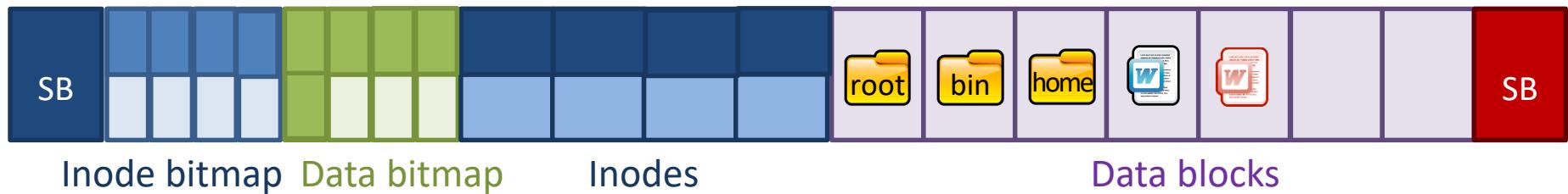


fsck Task #1: Superblock



Superblock
file systemversion
Size of inode bitmaps
Location of inode bitmaps
Size of data bitmaps
Location of data bitmaps
Index of root inode

Most file systems keep the backup version of the superblock



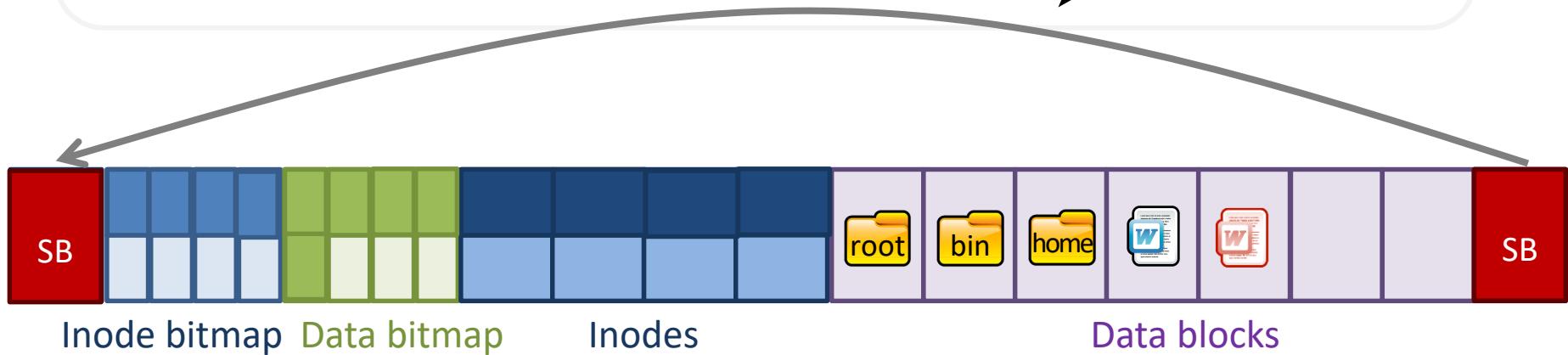


fsck Task #1: Superblock



Superblock
file systemversion
Size of inode bitmaps
Location of inode bitmaps
Size of data bitmaps
Location of data bitmaps
Index of root inode

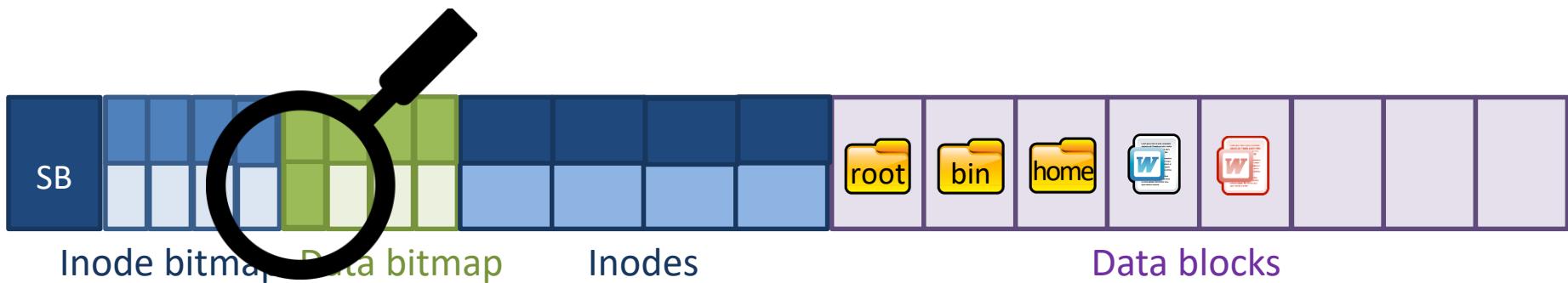
Replace superblock
with the backup
version



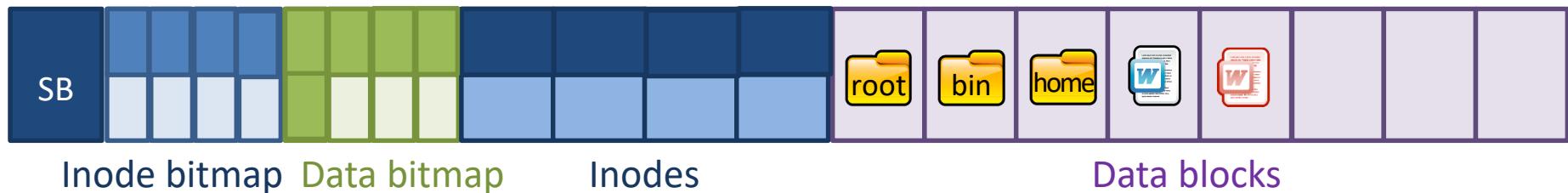
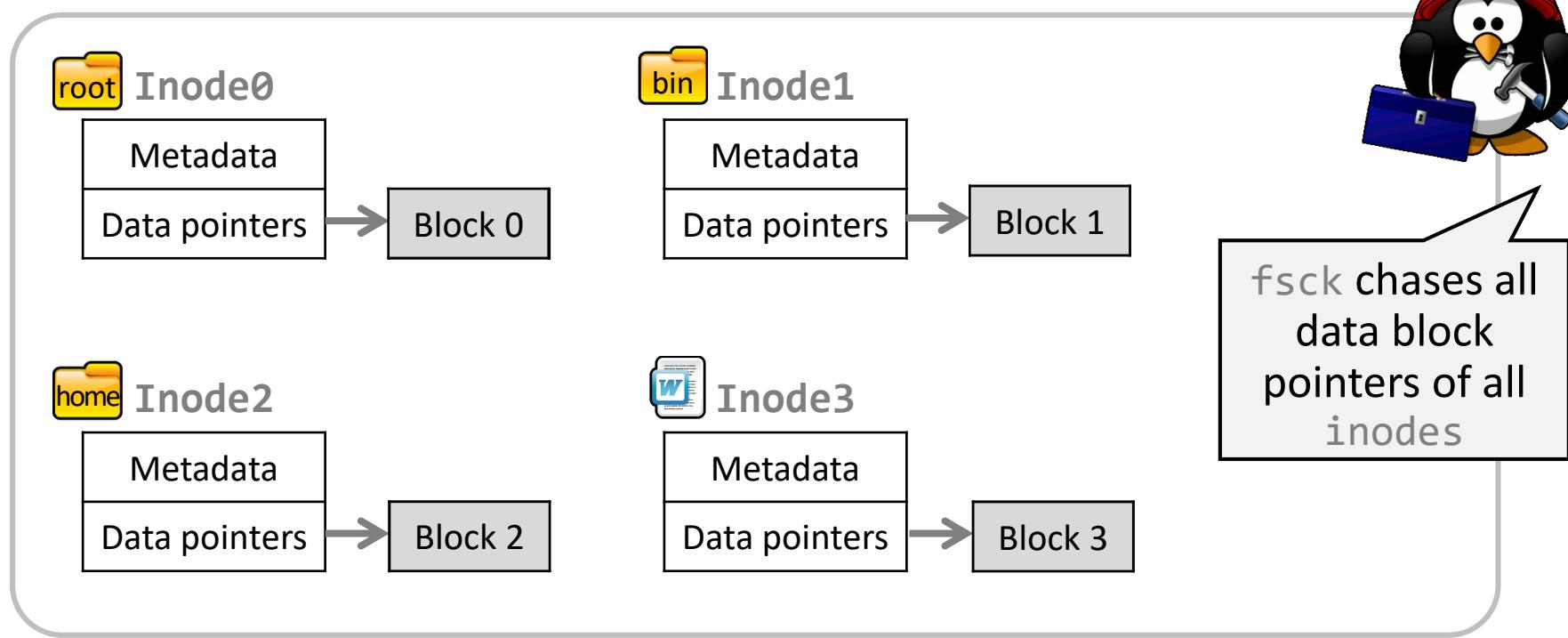


fsck Task #2: Free blocks

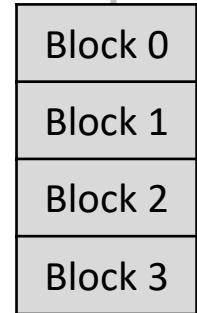
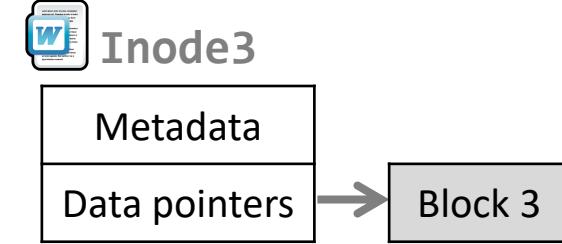
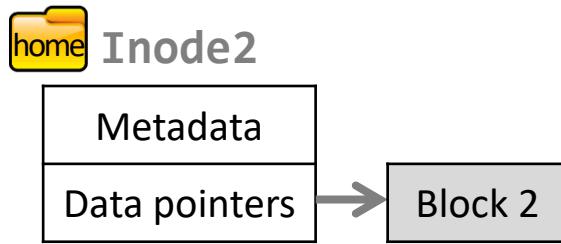
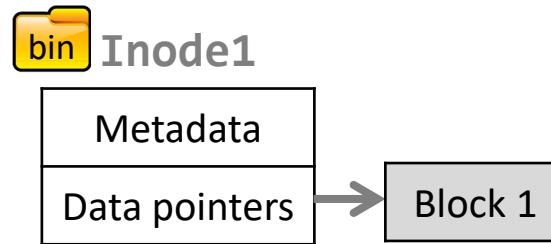
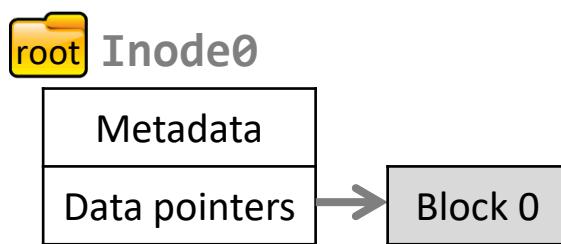
Rebuild the bitmaps
by scanning all
inodes



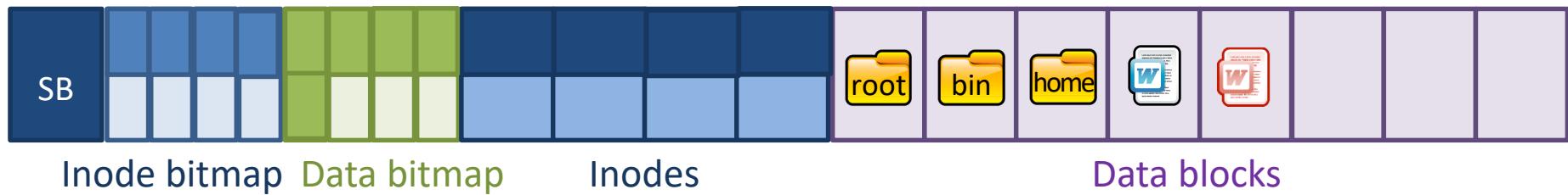
fsck Task #2: Free blocks



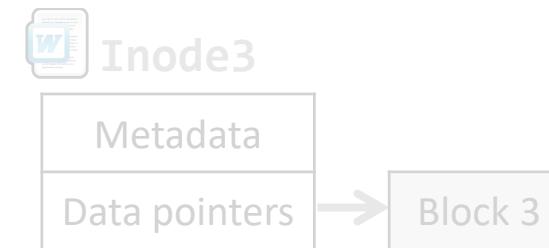
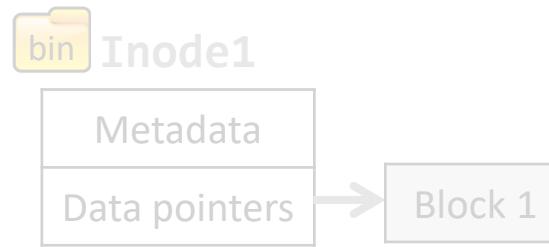
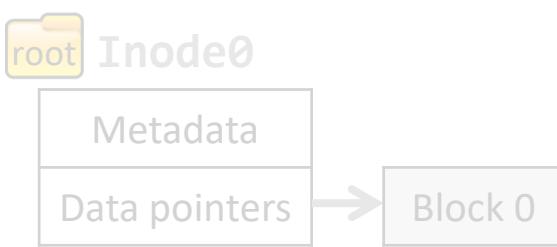
fsck Task #2: Free blocks



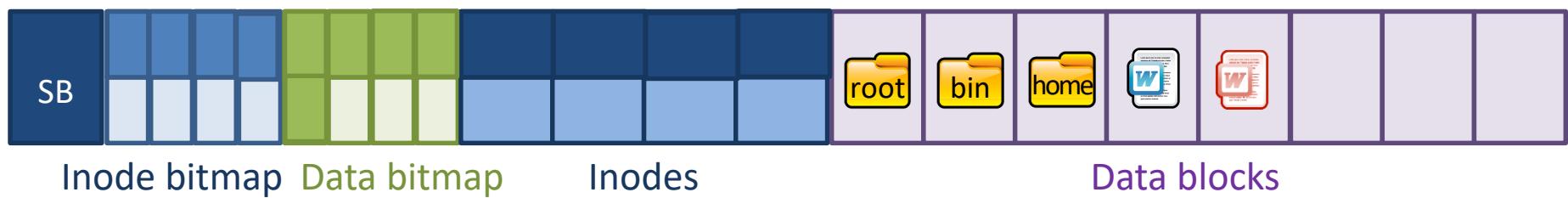
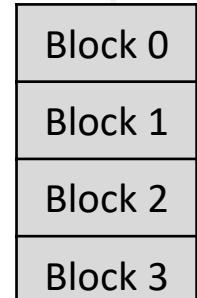
These are valid data blocks (which can be also updated in data bitmap)



fsck Task #2: Free blocks



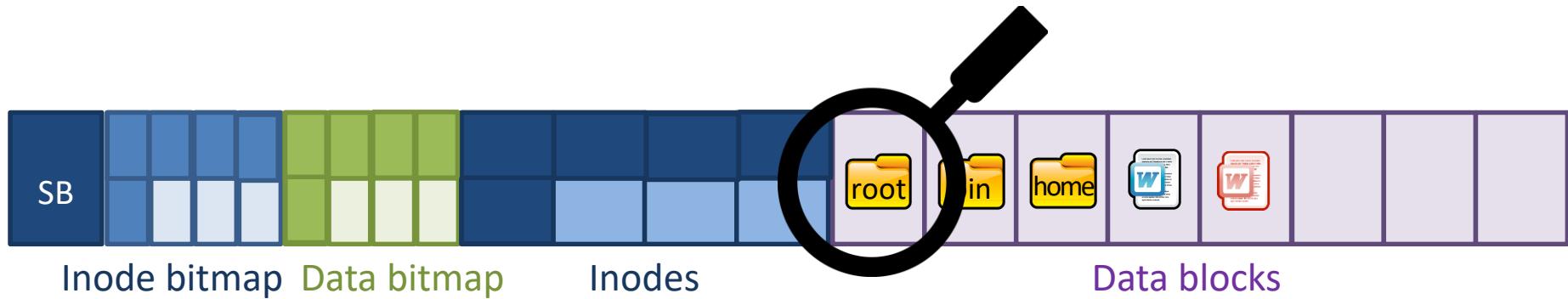
Fix data bitmap
(remove the data block 4's flag)





fsck Task #3: Reachability

Make sure all `inodes` are
reachable from the root
of the file system





fsck Task #3: Reachability



The diagram shows three directory structures:

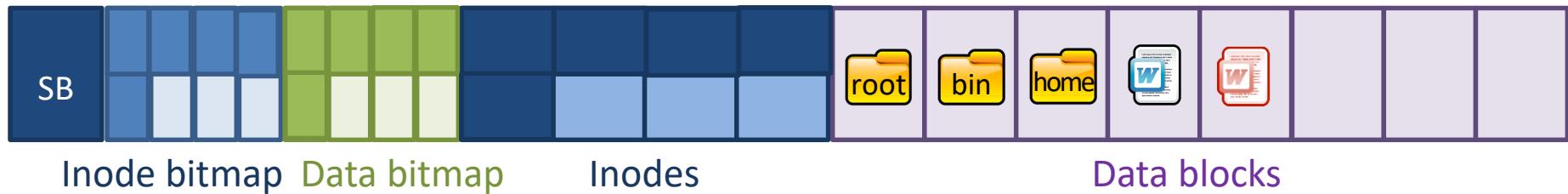
- root** Directory:

Name	inode
.	0
bin	1
home	2
essay	3
- bin** Directory:

Name	inode
..	0
.	1
- home** Directory:

Name	inode
..	0
.	2

A callout box points to the 'home' directory with the text: "fsck chases all directories from the root".





fsck Task #3: Reachability



root **Directory**

Name	inode
.	0
bin	1
home	2
essay	3

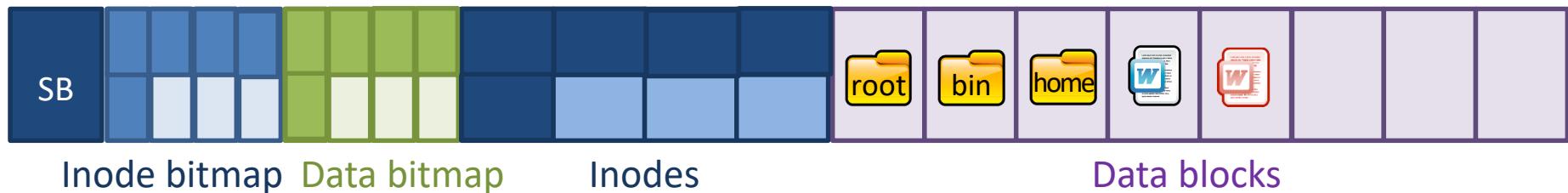
bin **Directory**

Name	inode
..	0
.	1

home **Directory**

Name	inode
..	0
.	2

These are valid
inodes (which
could be set in
inode bitmap)



fsck Task #3: Reachability



Directory



Directory



Directory

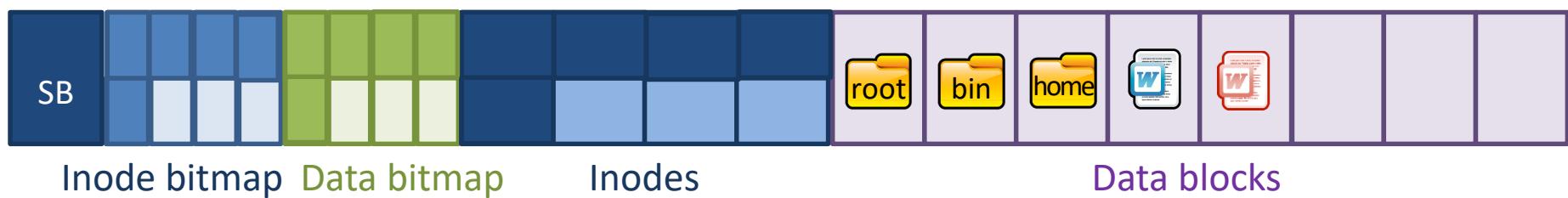
Name	inode
.	0
bin	1
home	2

Name	inode
..	0
.	1

Name	inode
..	0
.	2

0
1
2
3

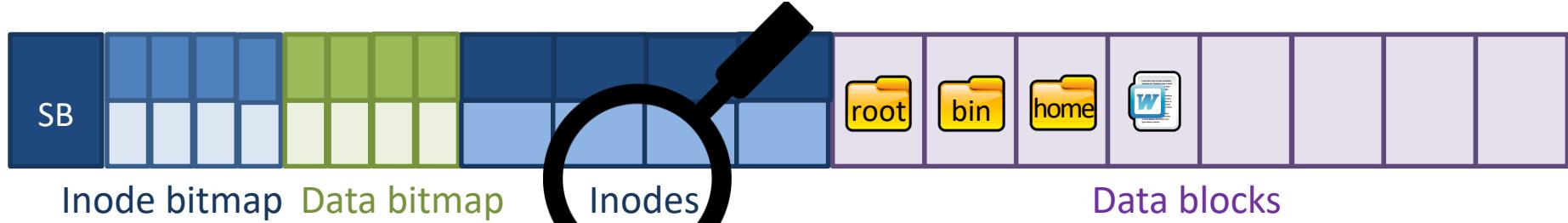
Fix inode bitmap
(remove data
inode 4 flag)





fsck Task #4: inode

Delete all corrupted
inodes, and rebuild their
link counts by walking the
directory tree



fsck Task #4: inode

root Inode0

Link_count	0
Data pointers	

home Inode2

Link_count	0
Data pointers	

bin Inode1

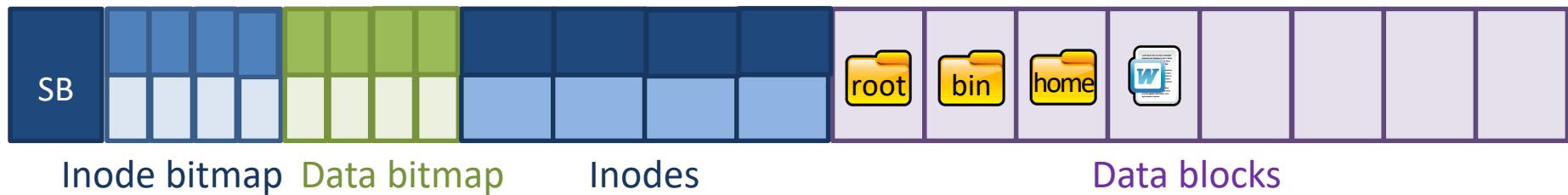
Link_count	0
Data pointers	

Inode3

Link_count	3
Data pointers	



fsck chases all
data block
pointers of all
inodes



fsck Task #4: inode

root Inode0

Link_count	0
Data pointers	

bin Inode1

Link_count	0
Data pointers	

home Inode2

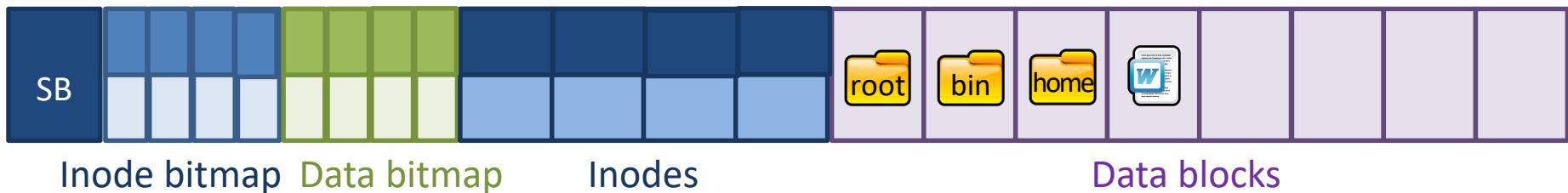
Link_count	0
Data pointers	

W Inode3

Link_count	3
Data pointers	



Check the correctness
of inode 3's link_count



fsck Task #4: inode



root Inode0

Link_count	0
Data pointers	

bin Inode1

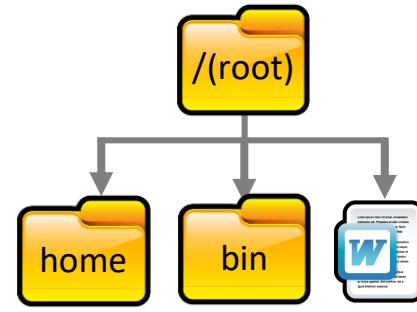
Link_count	0
Data pointers	

home Inode2

Link_count	0
Data pointers	

Inode3

Link_count	0
Data pointers	



There is no link for
inode 3. Change
link_count to 0

SB

Inode bitmap Data bitmap

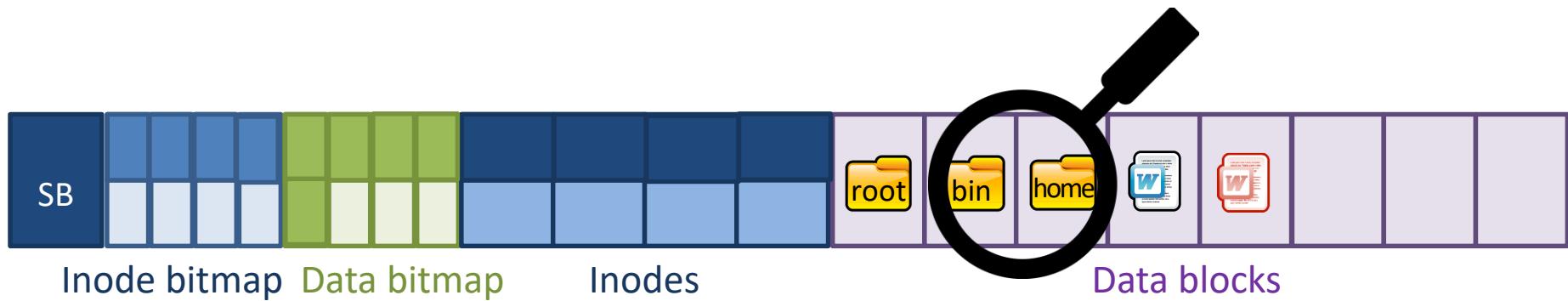
Inodes

Data blocks



fsck Task #5: directories

Verify the
integrity of all
directories





fsck Task #5: directories



Directory

Name	inode
.	0
bin	1
home	2
essay	3



Directory

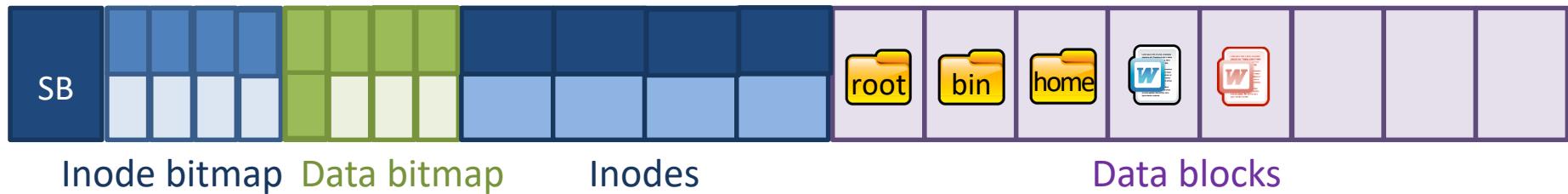
Name	inode
..	0
.	1



Directory

Name	inode
..	
.	2

fsck chases all
directories from
the root





fsck Task #5: directories



Directory

Name	inode
.	0
bin	1
home	2
essay	3



Directory

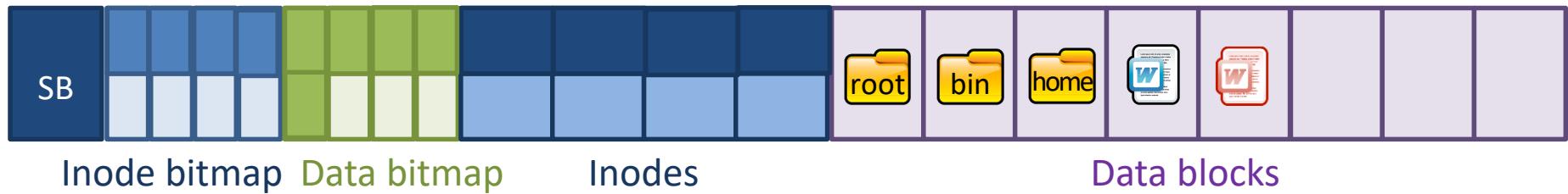
Name	inode
..	0
.	1



Directory

Name	inode
..	
.	2

Although home is
not a root directory,
there is no parent
directory





fsck Task #5: directories

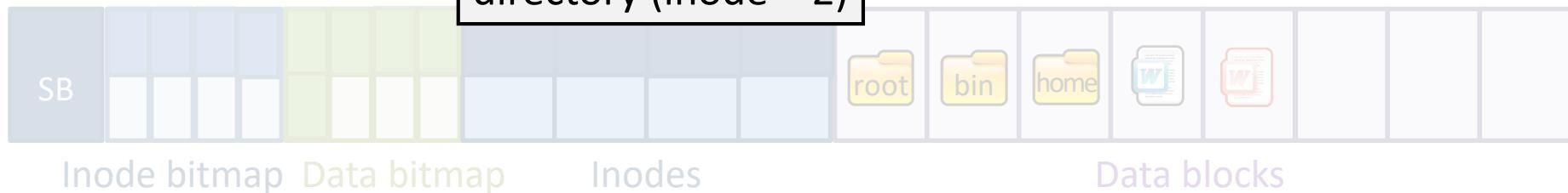


Directory	
Name	inode
.	0
bin	1
home	2
essay	3

Directory	
Name	inode
..	0
.	1

Directory	
Name	inode
..	
.	2

Find out the directory, which includes the home directory (inode = 2)





fsck Task #5: directories



Directory

Name	inode
.	0
bin	1
home	2
essay	3



Directory

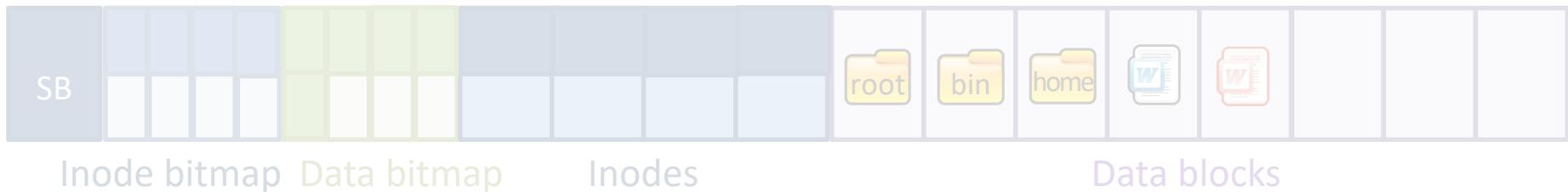
Name	inode
..	0
.	1



Directory

Name	inode
..	0
.	2

Update the
parent
directory



Problem of fsck

fsck doesn't require
the file system to do
any work to ensure
consistency

However, there are
several critical
problems of fsck if we
only rely on fsck



#1: Hard to Implement fsck

Many possible inconsistencies that **must be detected** to recover. Are all these really the tasks that we should rely on `fsck`?

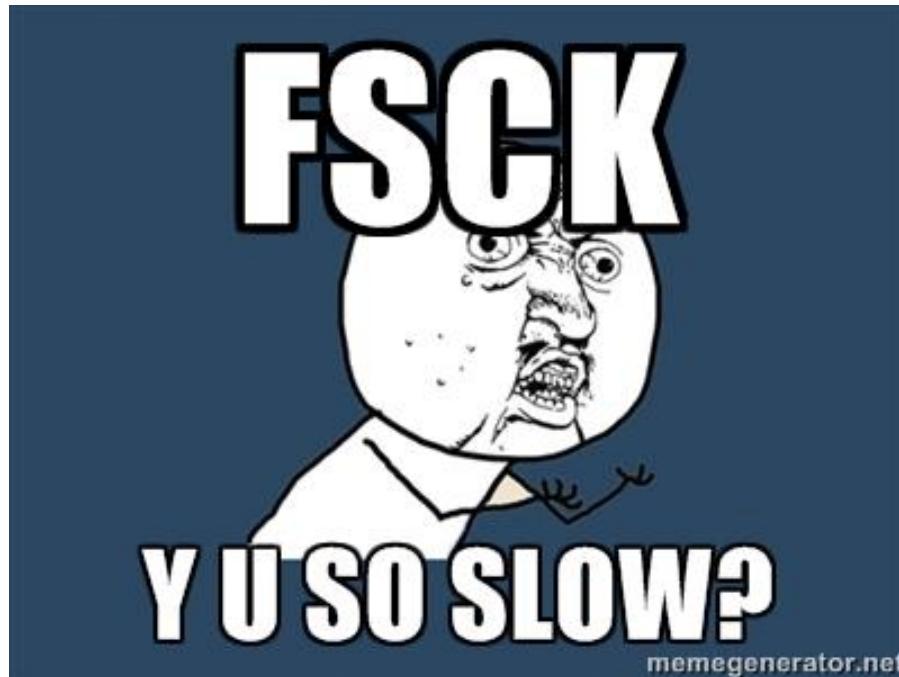
Many difficult corner cases to consider and handle

Tasks	Detail task
Task #1: Superblock	Validate the superblock, replace it with a backup if it is corrupted
Task #2: Free blocks	Rebuild the bitmaps by scanning all inodes
Task #3: Reachability	Make sure all inodes are reachable from the root of the File-System
Task #4: Inodes	Delete all corrupted inodes, and rebuild their link counts by walking the directory tree
Task #5: Directories	Verify the integrity of all directories

#2: Super Slow

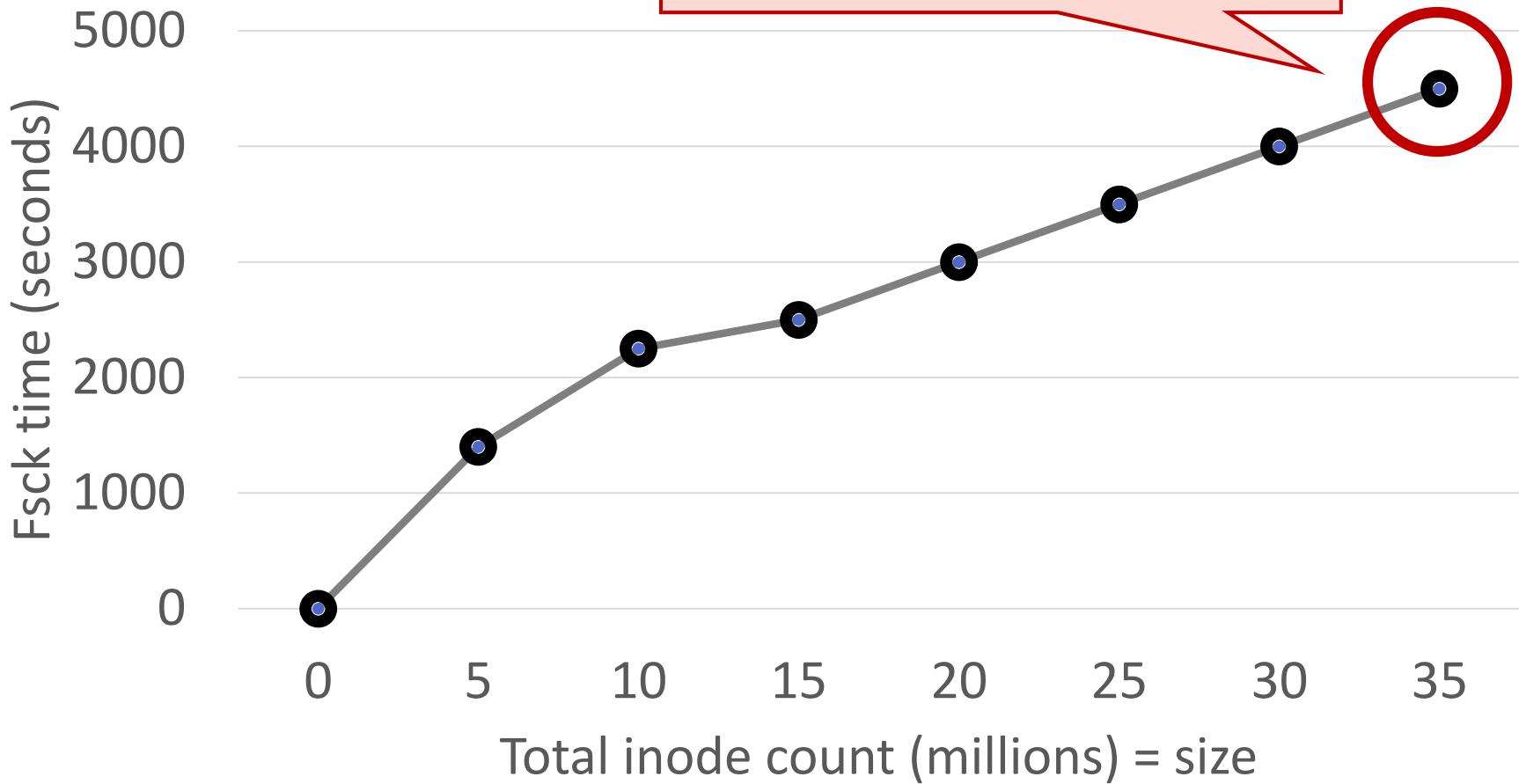
Fsck scans the entire file system multiple times →
Super slow 😞

Imagine how long it would take fsck for a **40TB** RAID array

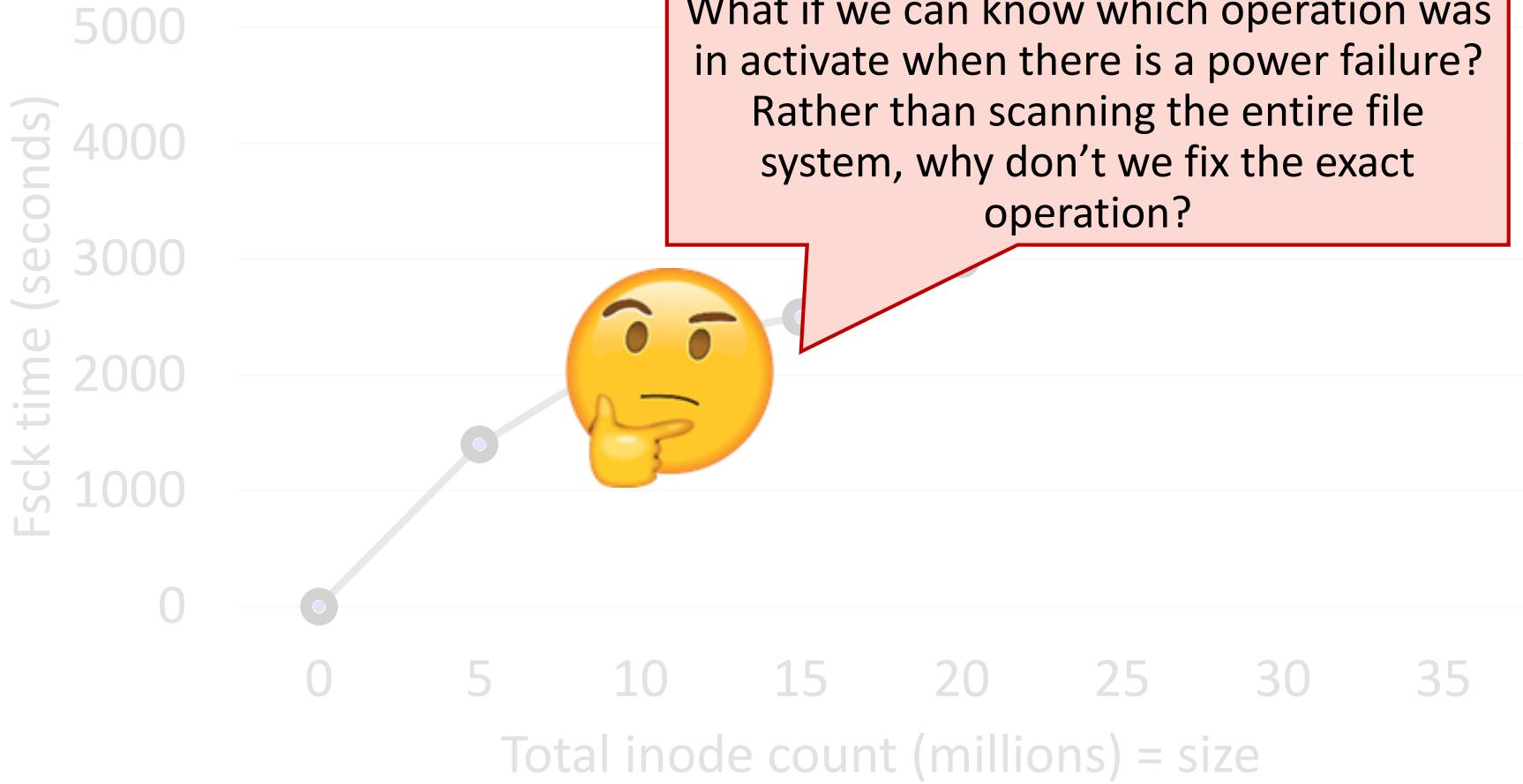


#2: Super Slow

If there are more than 35M
inodes, it will take **1.25 hours** to
recover the underlying file system

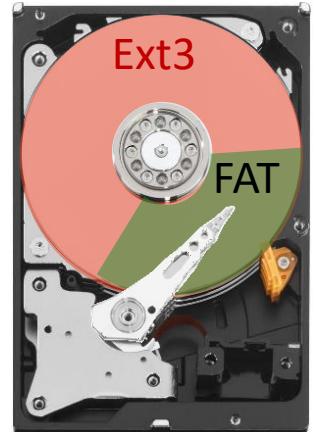


#2: Super Slow



Solution #2: Journaling File-System

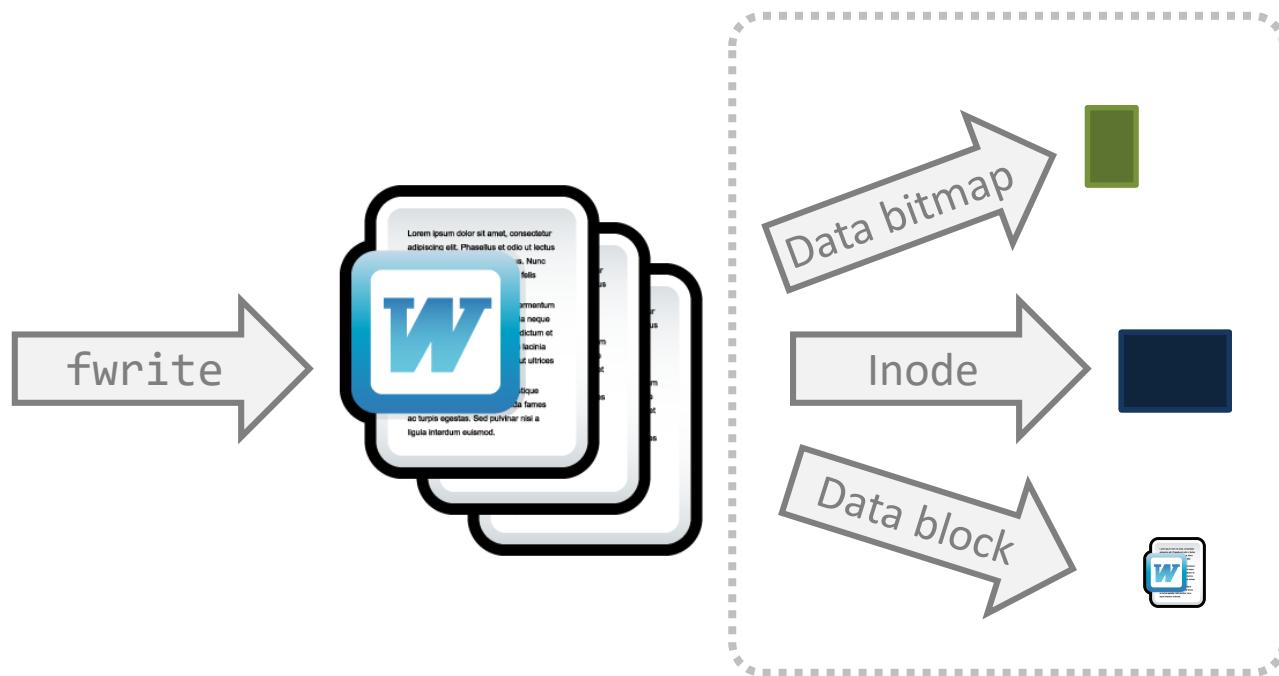
Let's modify file system
to keep the history of all
writes to the disk by
having a log (a.k.a,
journal)



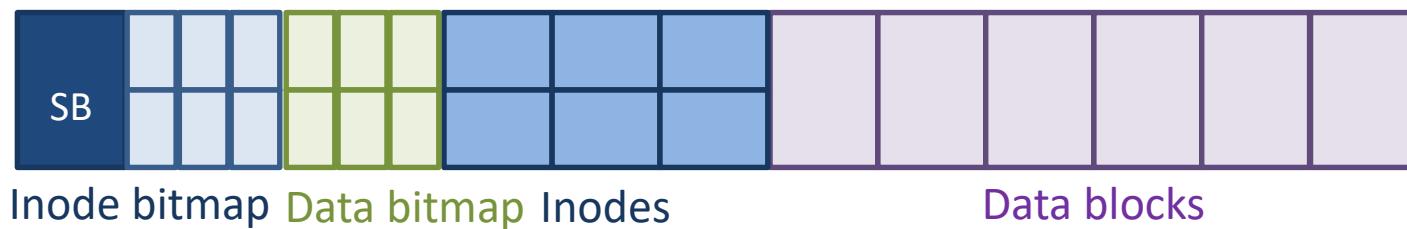
Key Idea: Make Writes Transaction

- Revisited example: file append

Journaling file system can make multi-writes as **atomic** by using a write-ahead log

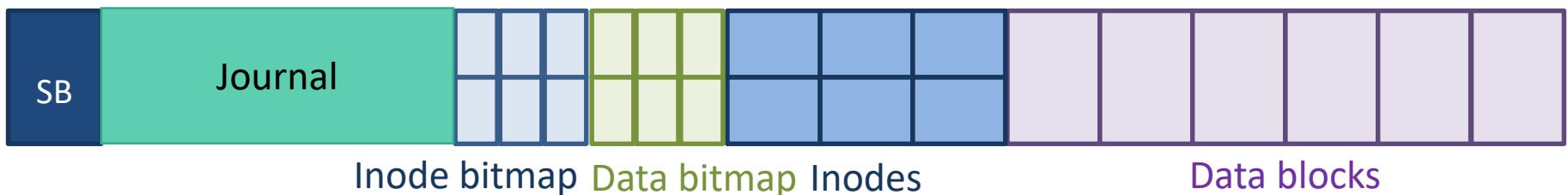


File System Modification

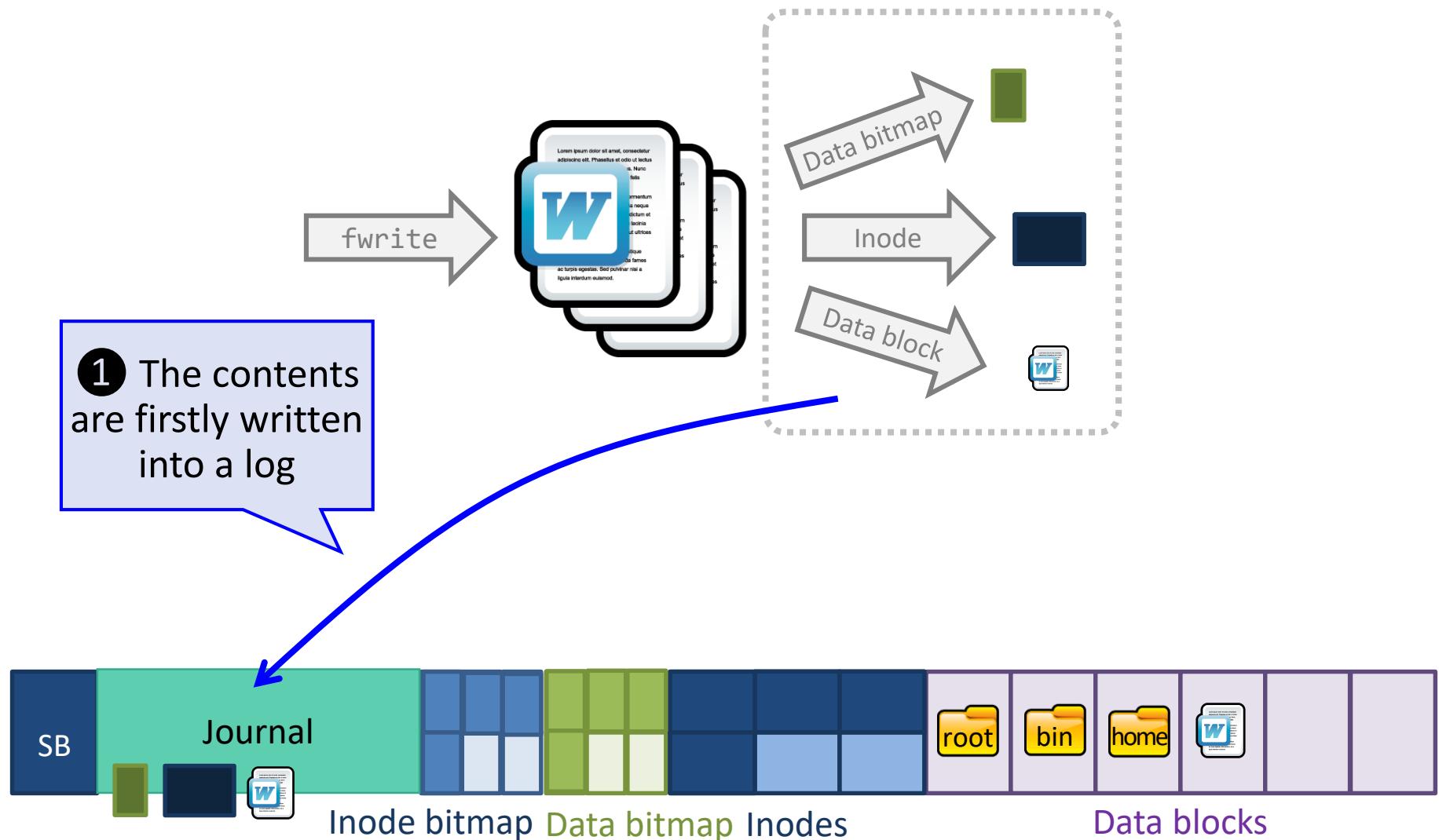


File System Modification

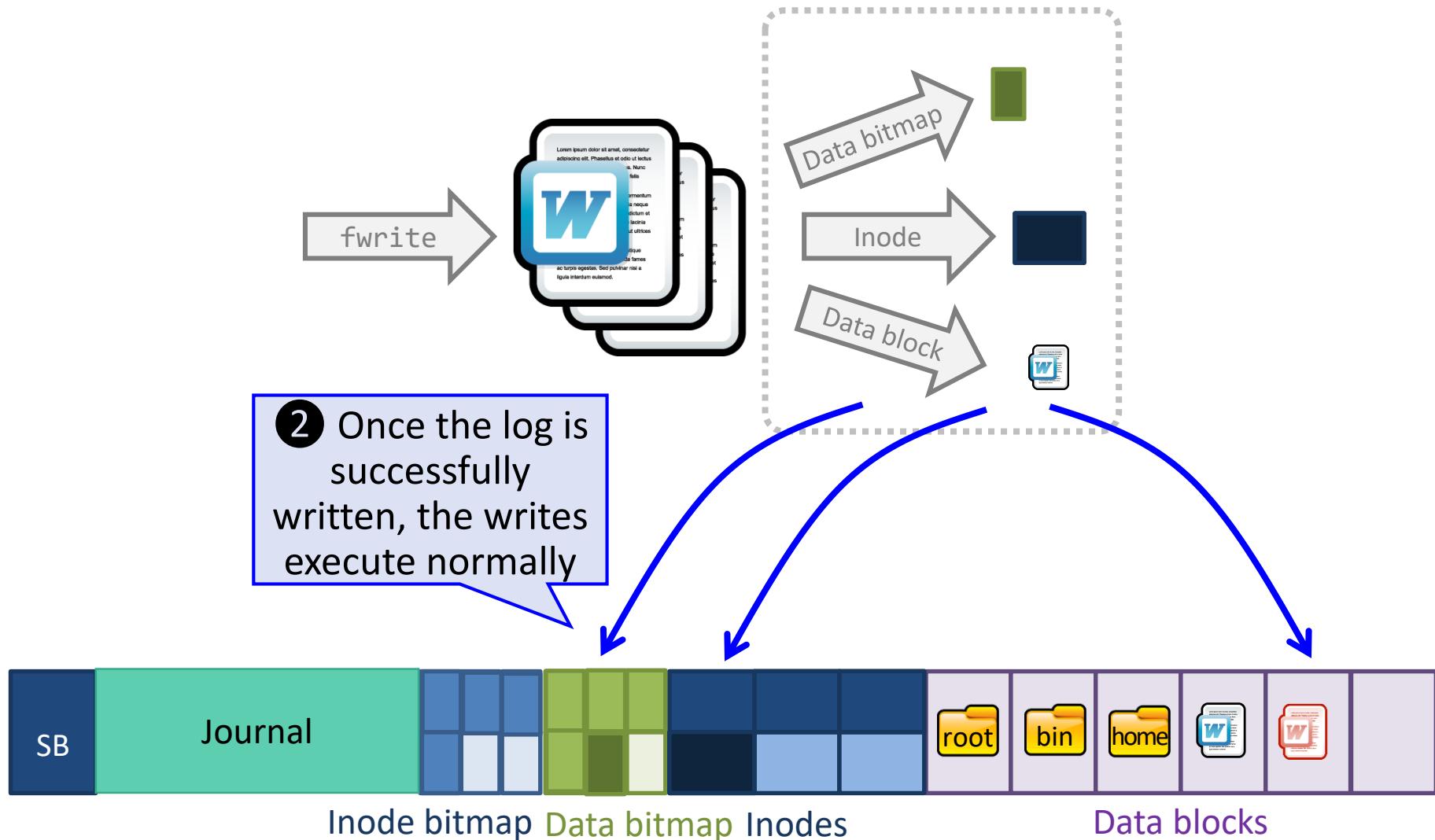
Journaling file system has an additional on-disk data structure ([journal/write-ahead log](#))



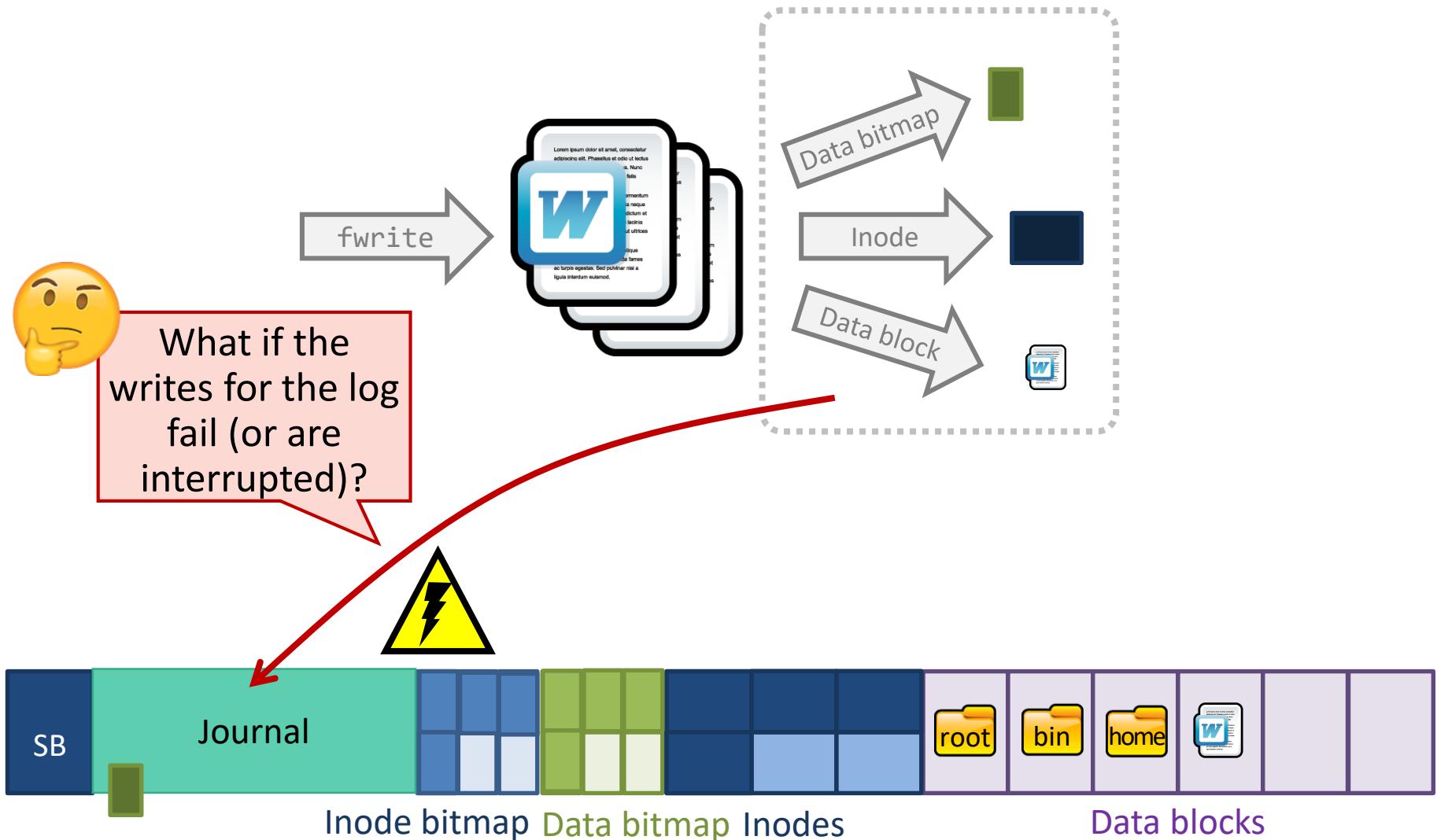
Write-Ahead Log



Write-Ahead Log

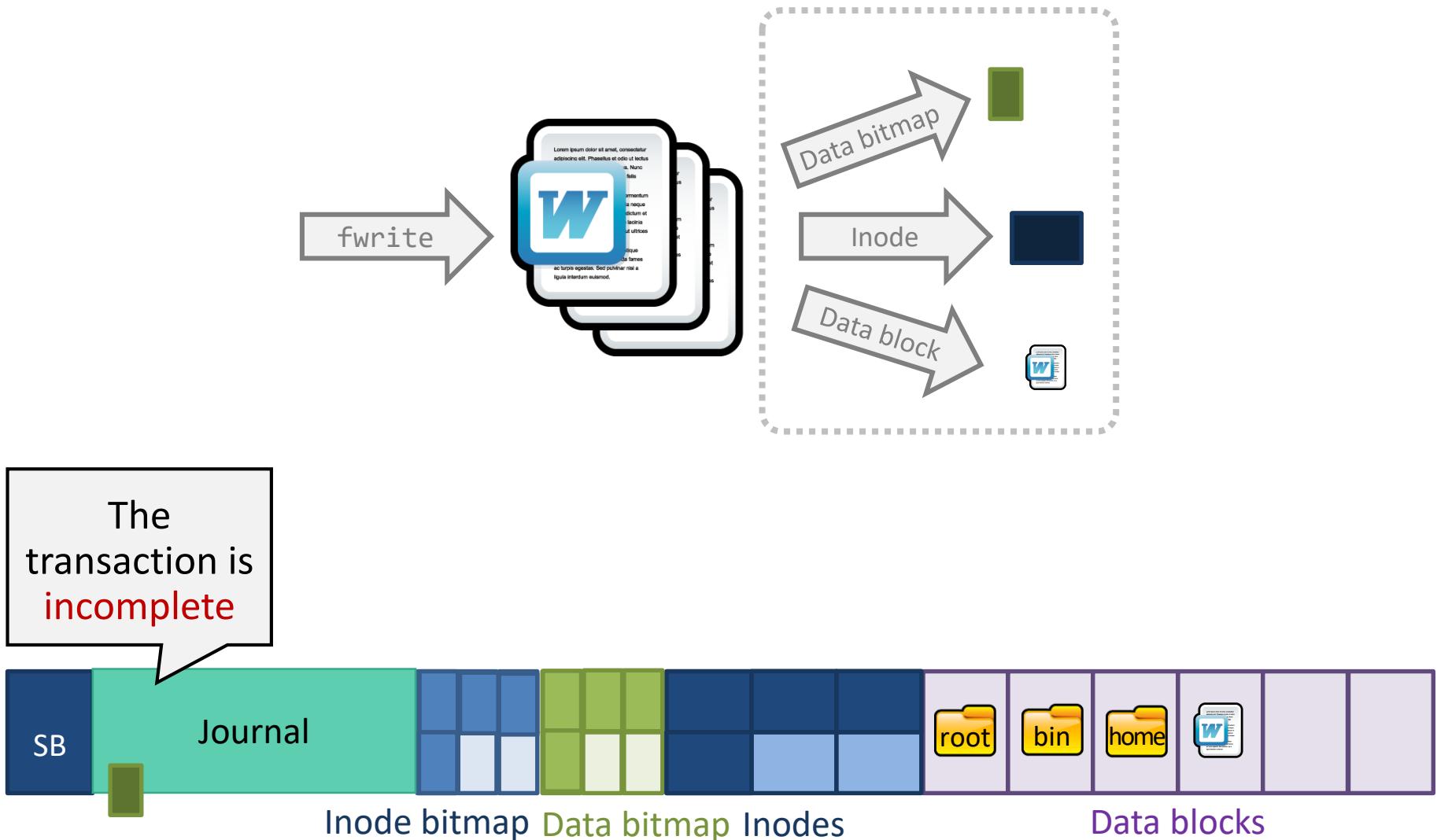


Crash with Write-Ahead Log

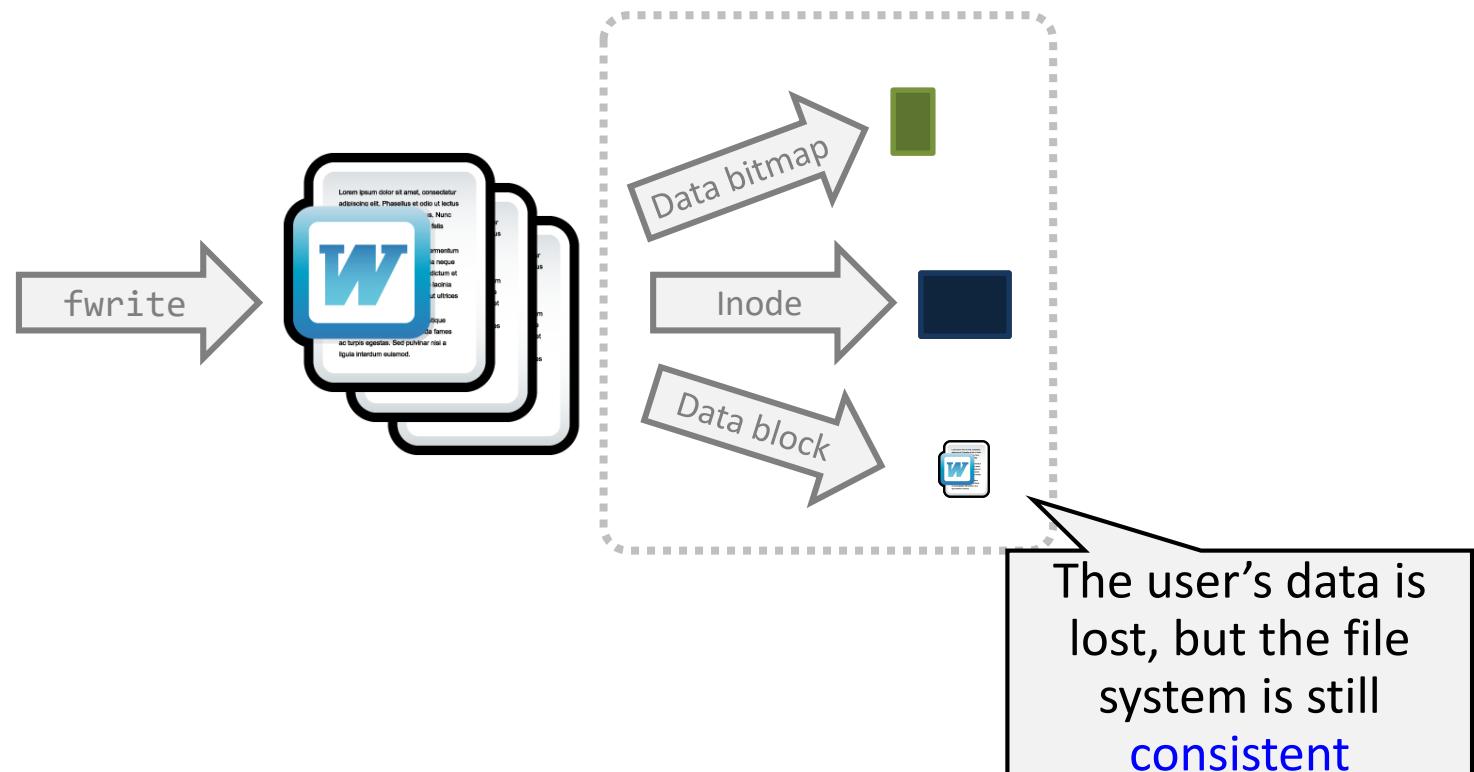




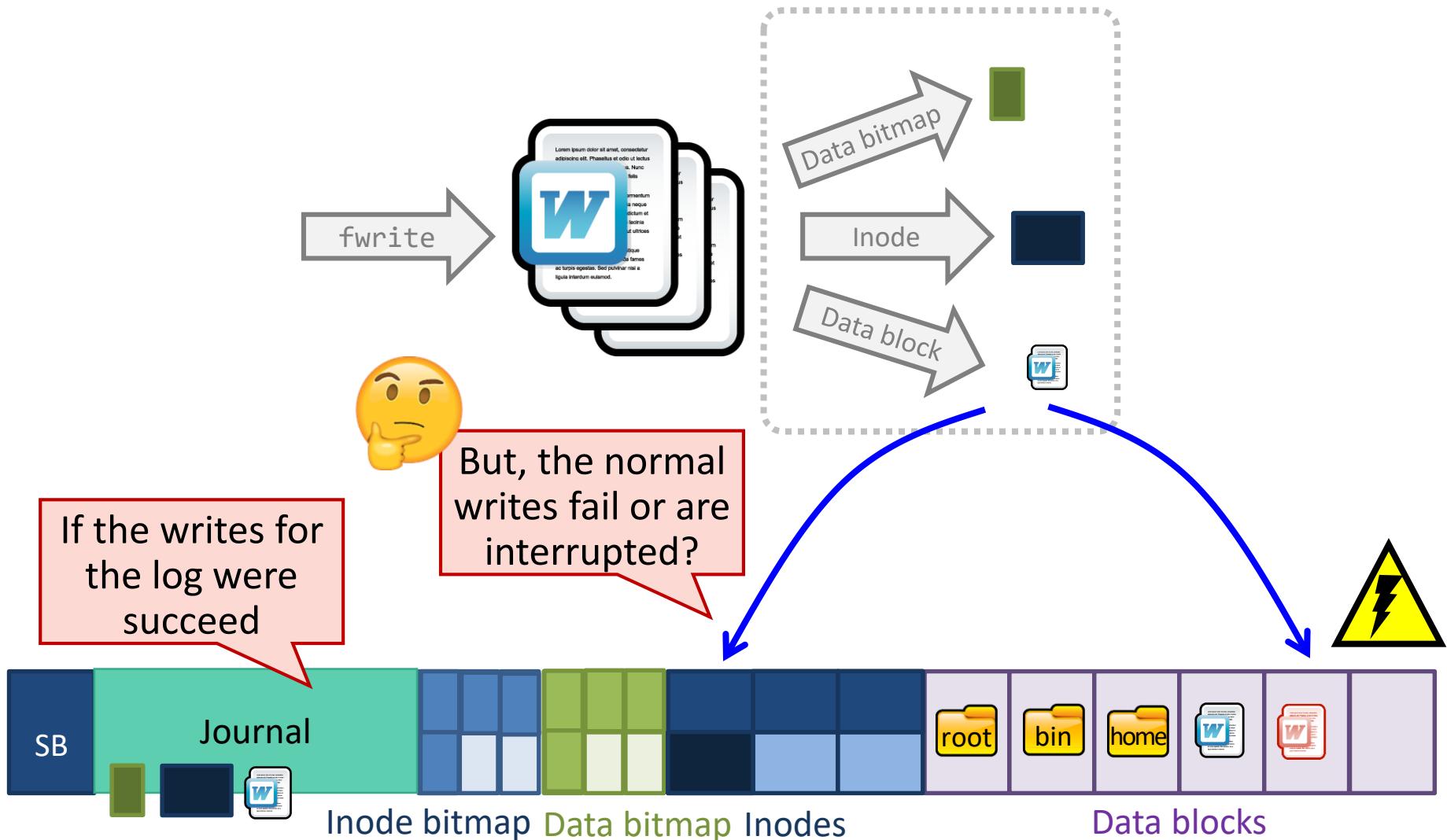
Crash with Write-Ahead Log



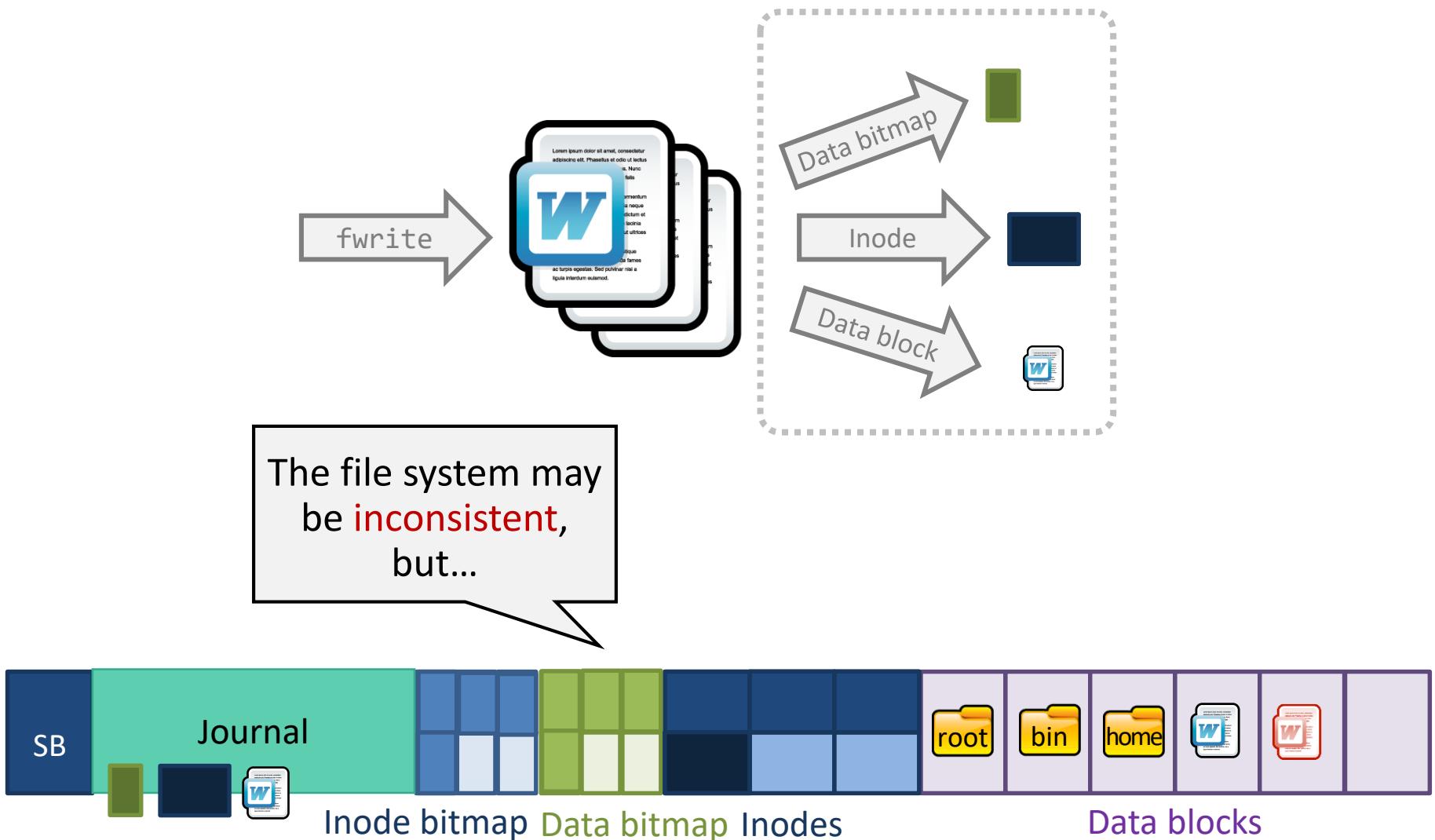
Crash with Write-Ahead Log



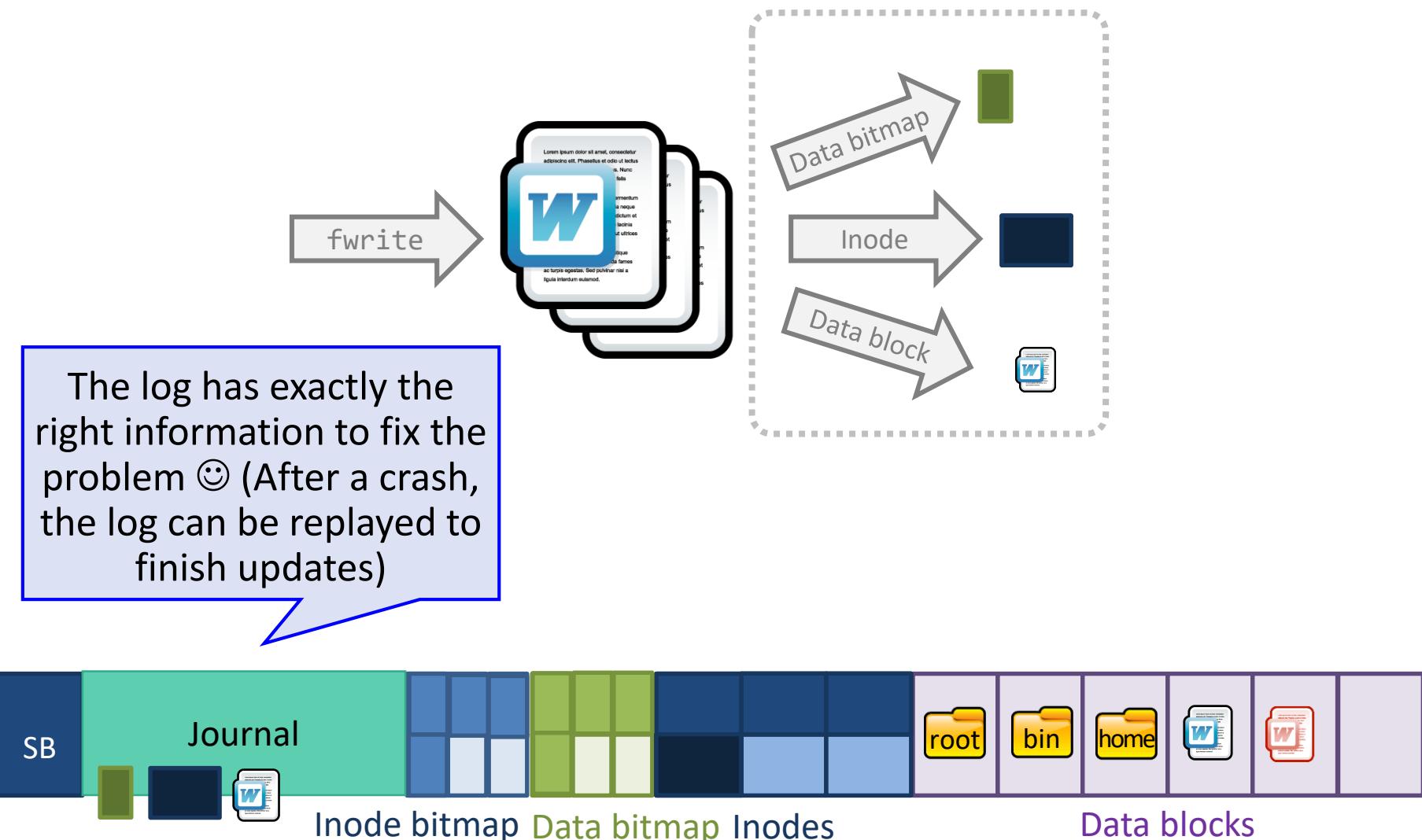
Crash with Write-Ahead Log



Crash with Write-Ahead Log



Crash with Write-Ahead Log



Example of Data Journaling

- Assume we are appending to a file
 - Three writes: inode v2, data bitmap v2, data D2

Before executing
these writes, log
these, first

Journal

Step #1: Being a New Transaction

* TxB: Transaction Begin

Begin a new
transaction with a
unique ID= k

Journal

TxB
ID=1

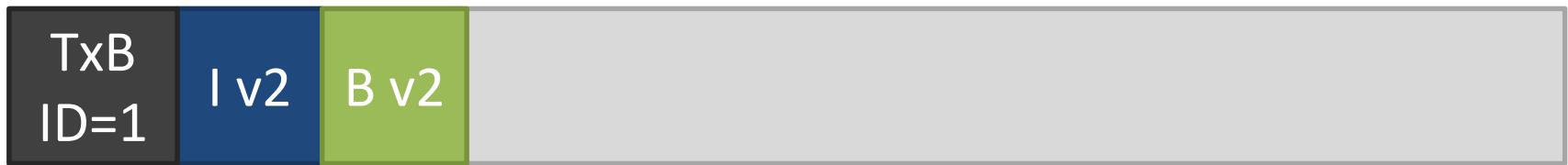


Step #2: Update Meta-data

Write the updated
meta-data block(s)
*(e.g., inode and
data bitmap)*



Journal





Step #3: Update Data

Journal



Write the file
data block(s)





Step #4: End of Transaction

* TxE: Transaction End

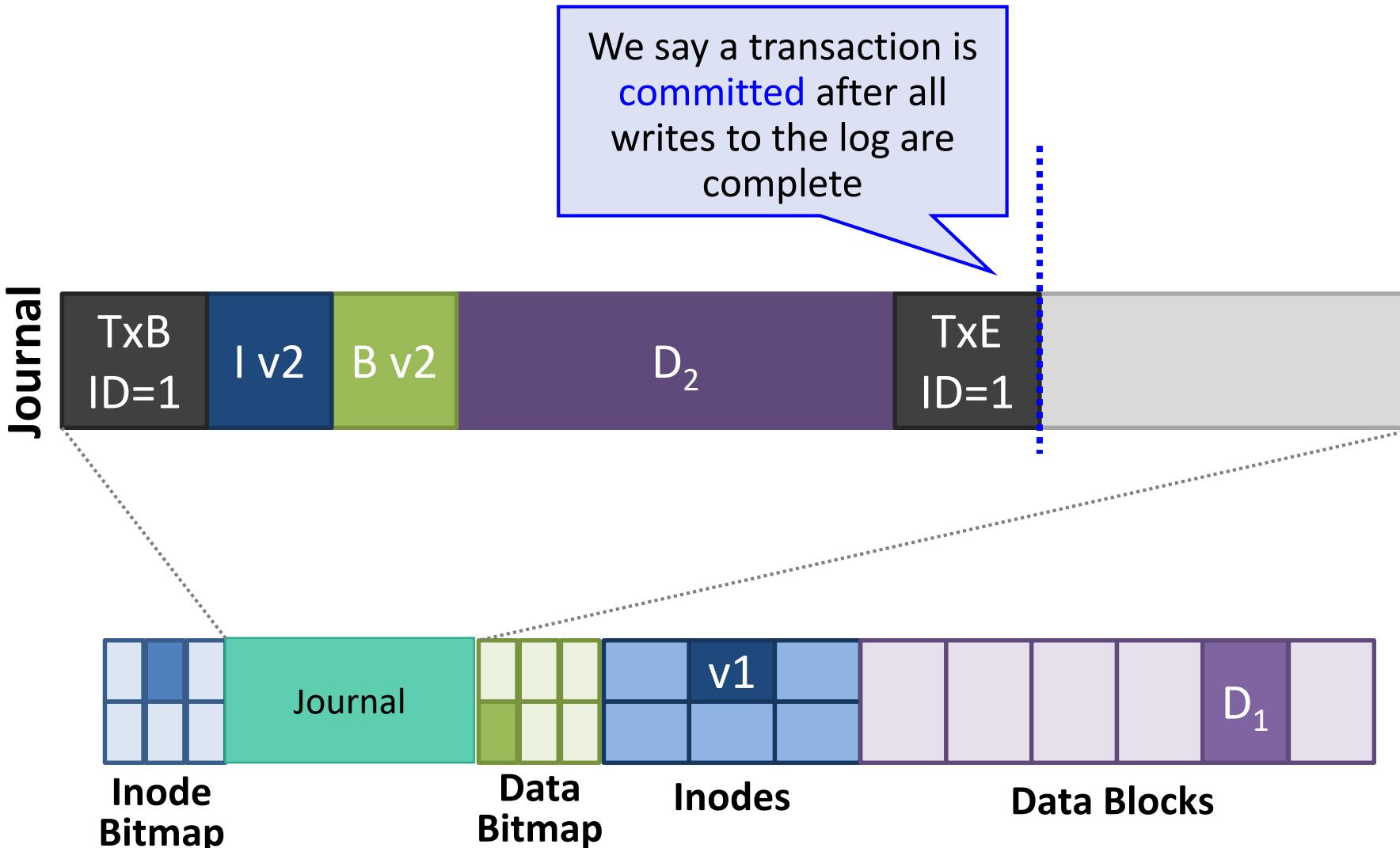
Write an end-of-transaction with ID= k

Journal



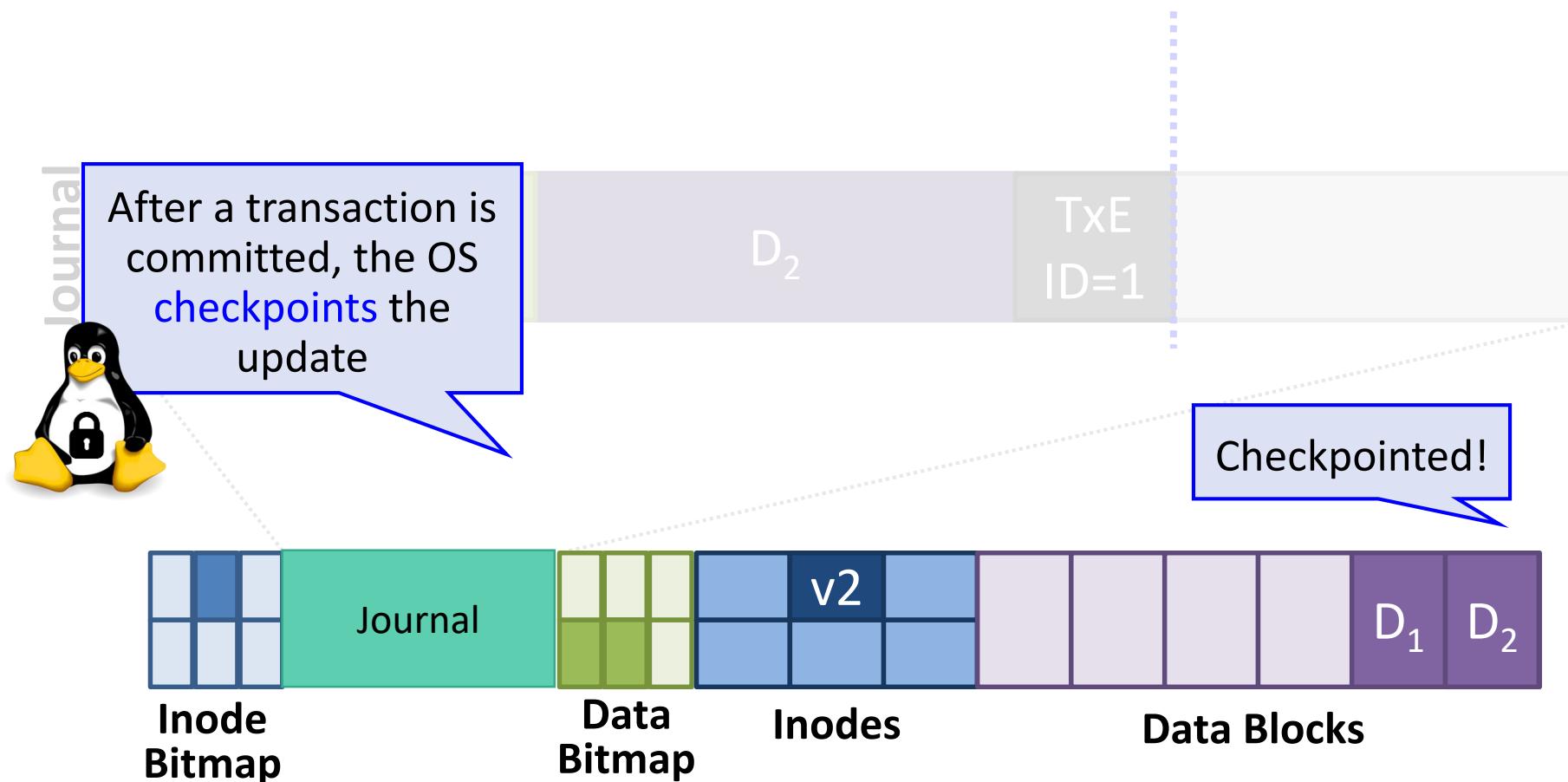


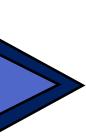
Step #5: Commits





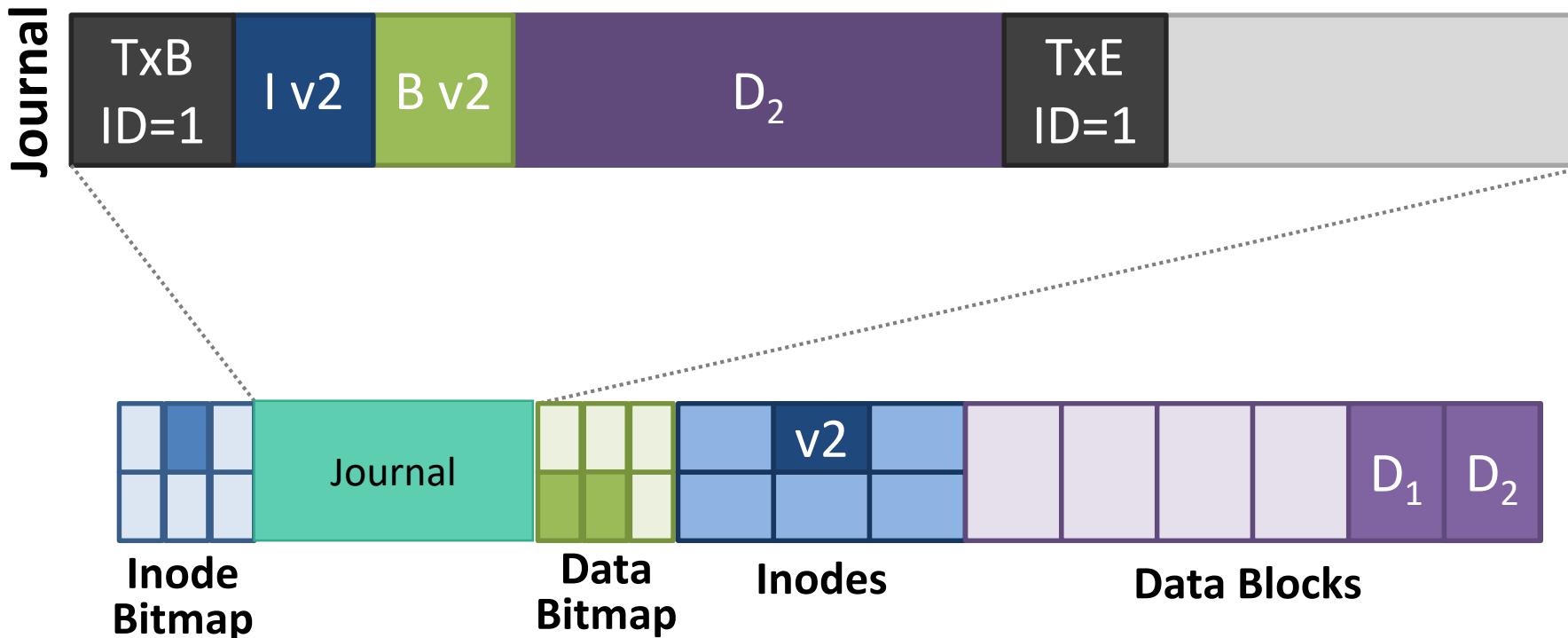
Step #6: Checkpoints





Step #7 (Final): Free The Transaction

After a checkpoint, free the checkpointed transaction from the journal region



Journal Implementation



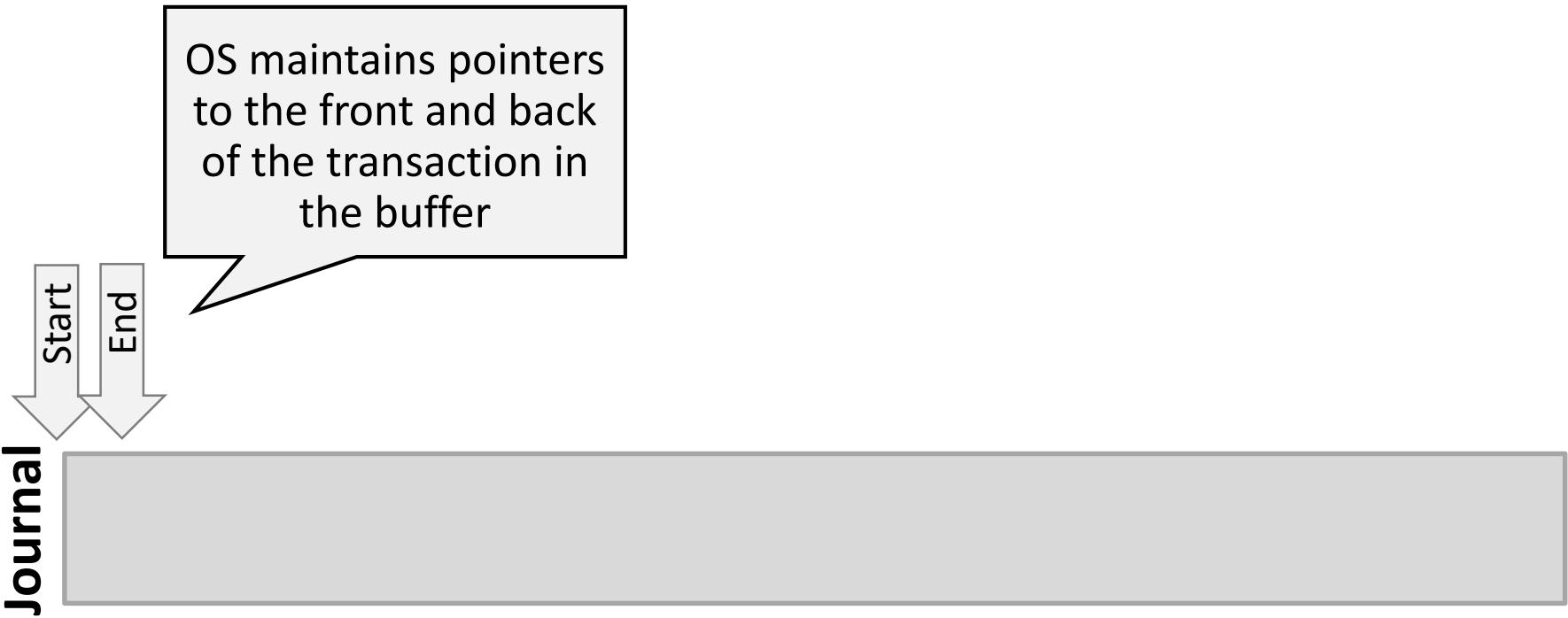
How can we
implement a journal?

Journal Implementation

Journal

Journals are typically implemented as a circular buffer
(append-only)

Journal Implementation



Journal Implementation



Journal Implementation



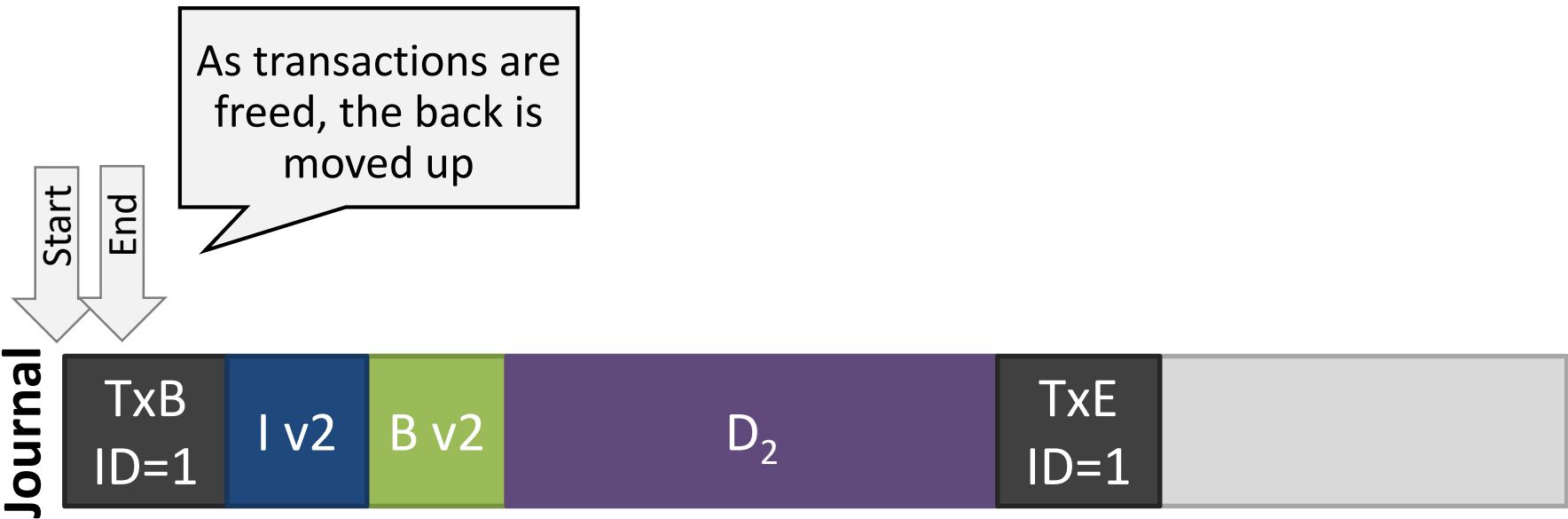
Journal Implementation



Journal Implementation

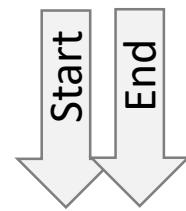


Journal Implementation



Journal Implementation

As the journal is managed by pointers, the contents of the log actually **never be deleted**, they are just **overwritten** as writes are progressed



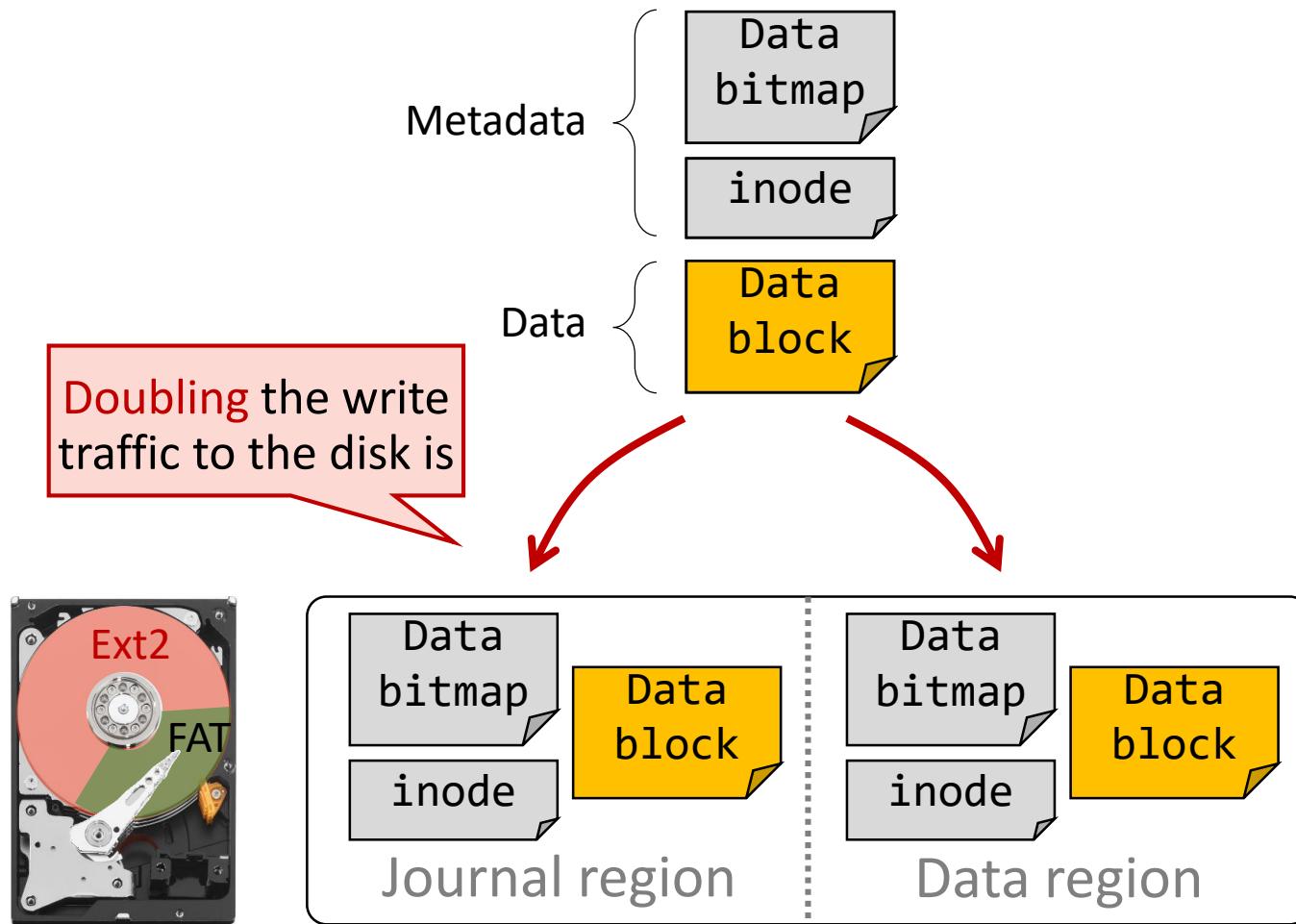
Problems of Journaling

Journaling is **robust** and **fast** to remedy file system's inconsistency

However, there are several problems of journaling, so we need **extra supports**



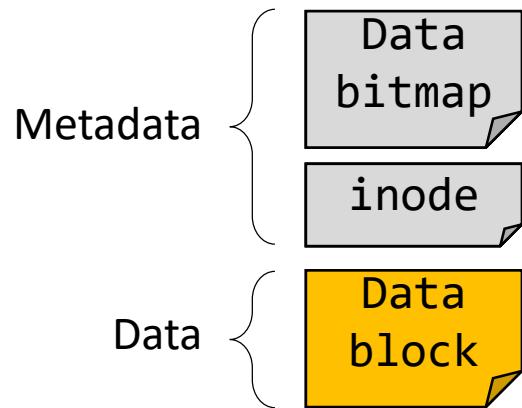
Problem #1: Doubled Write Traffic





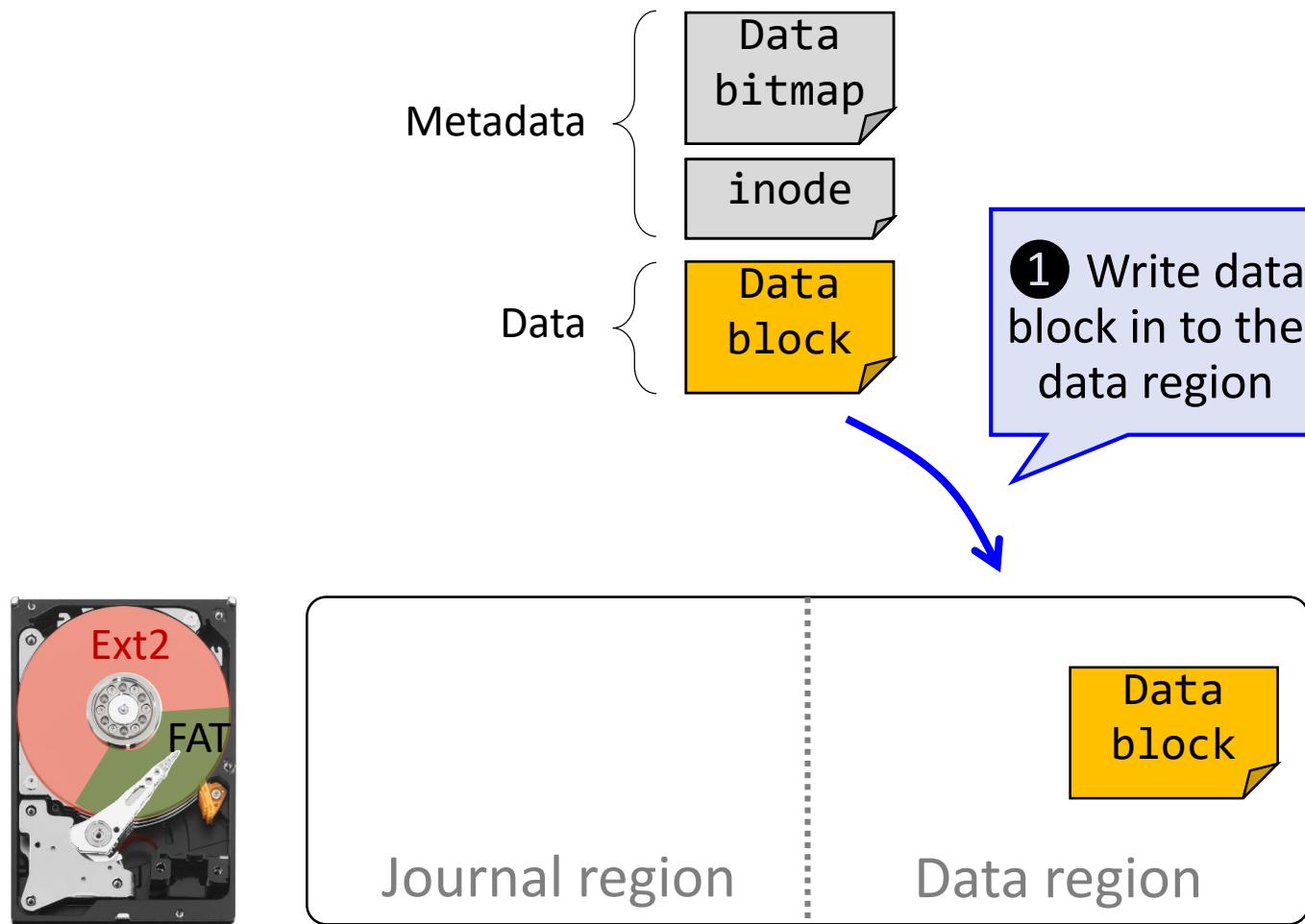
Solution: Meta-Data Journaling

Key idea: the most expensive part of data journaling is writing the file data twice
(meta-data is small: ~1 sector, file data is large)



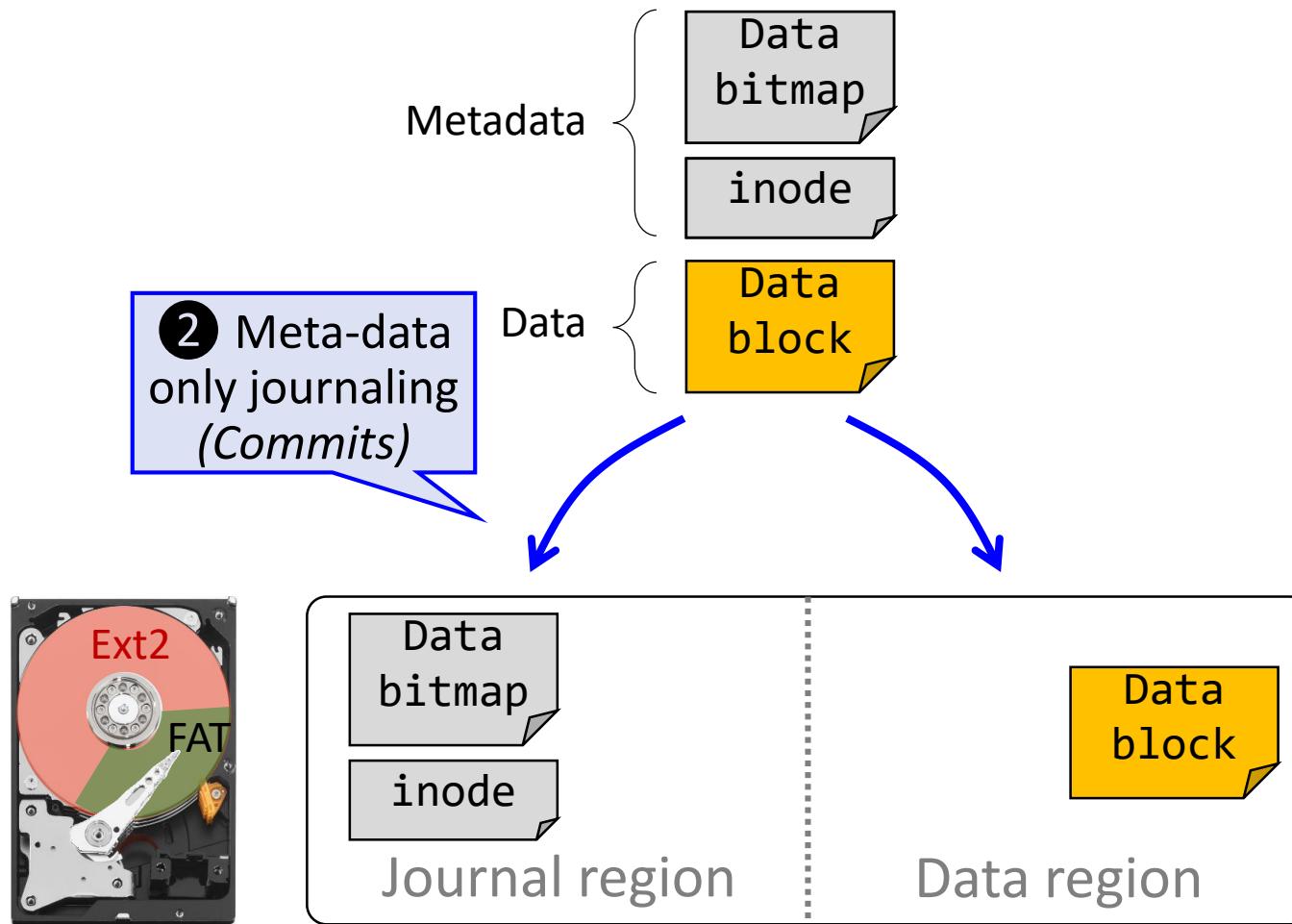


Solution: Meta-Data Journaling



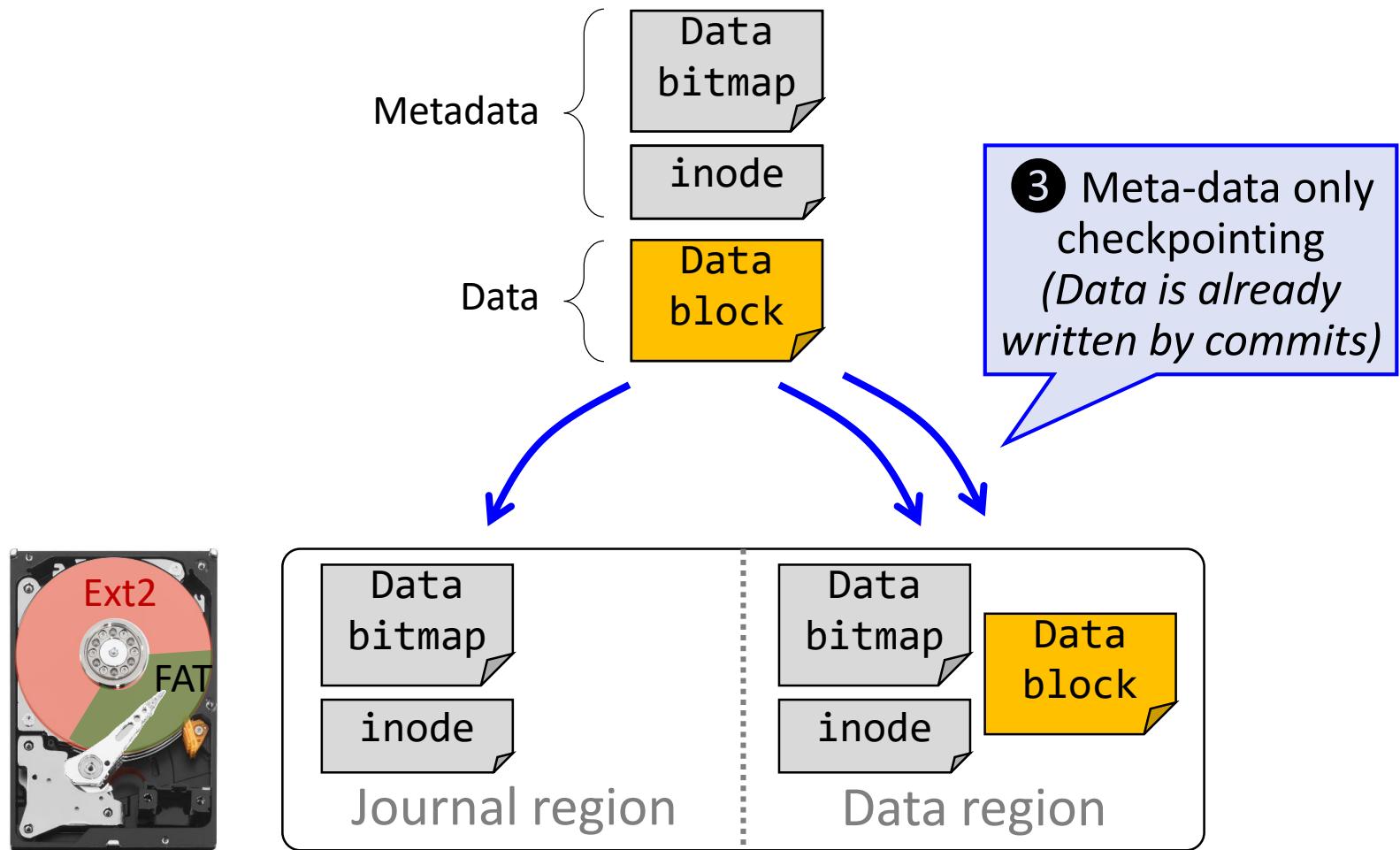


Solution: Meta-Data Journaling



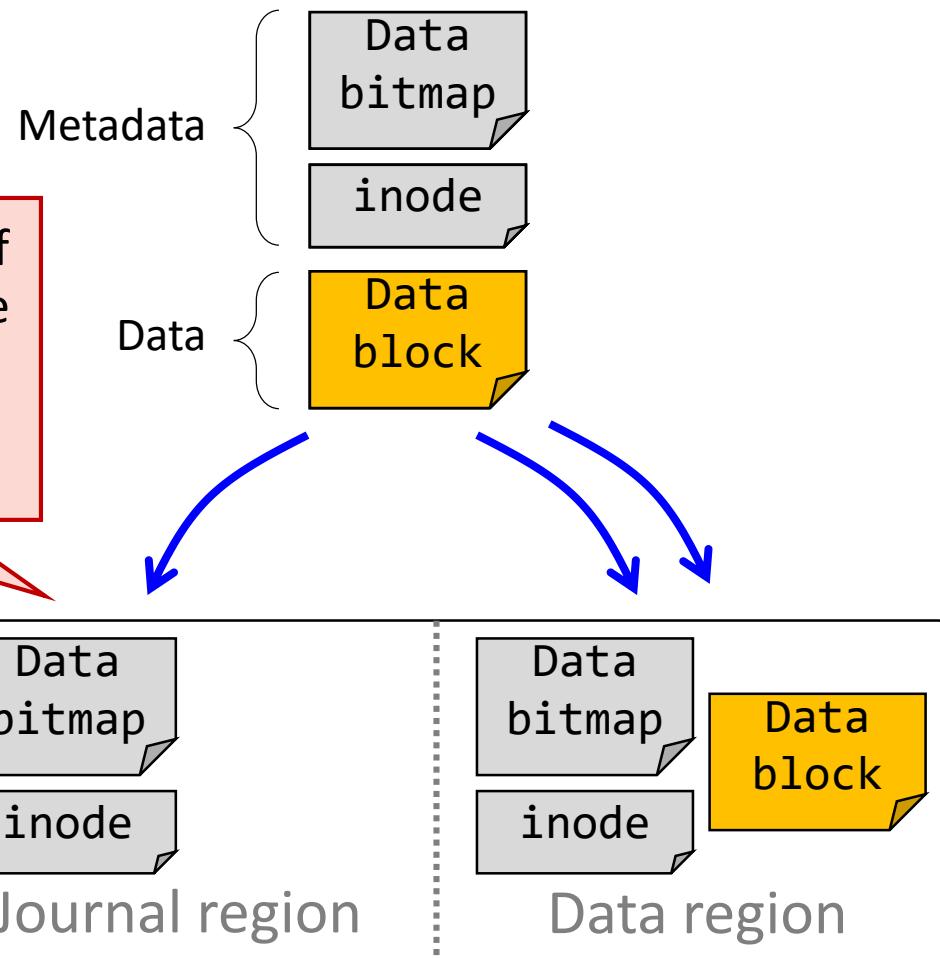


Solution: Meta-Data Journaling



Problem #2: Journaling Overhead

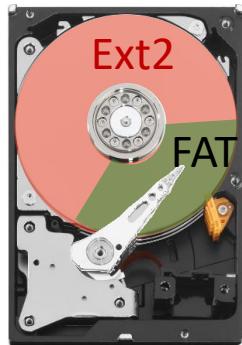
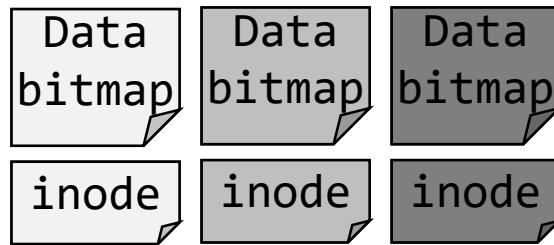
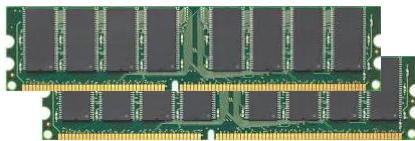
We reduce the amount of journaling by skipping the data block. However, still journaling adds a lot of write overhead





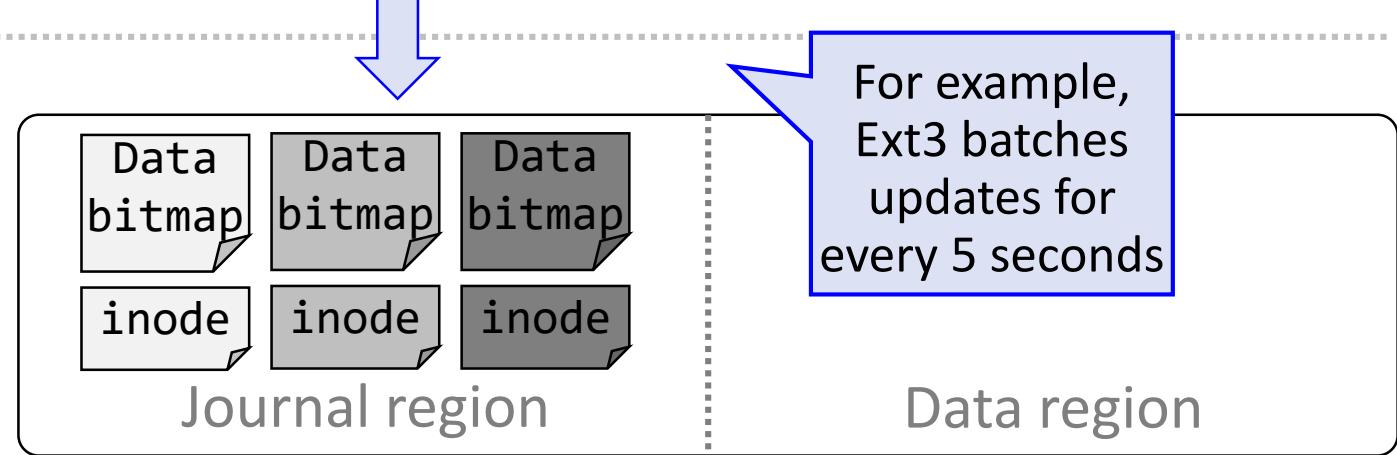
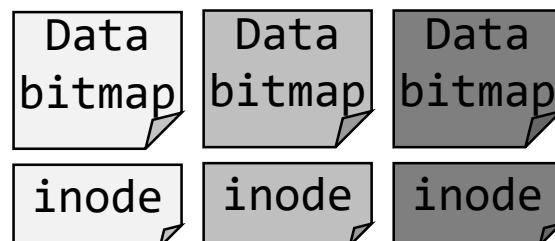
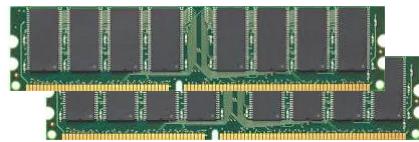
Solution: Batch Updates

OSes typically **batch updates** to the journal by buffering writes in memory and flushing/issuing them with a single large write to the log



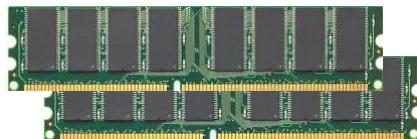


Solution: Batch Updates

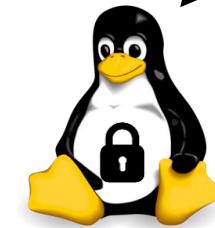




Solution: Batch Updates



There is tradeoff
between performance
and persistence

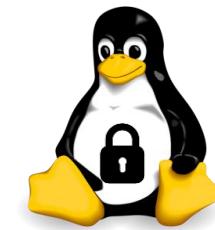
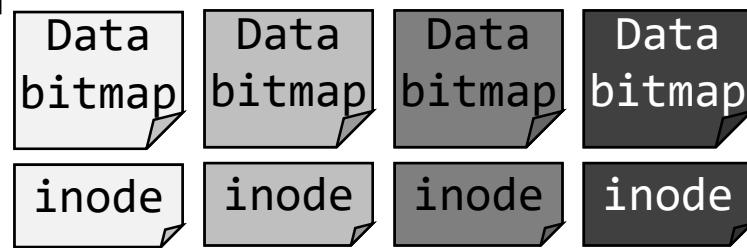
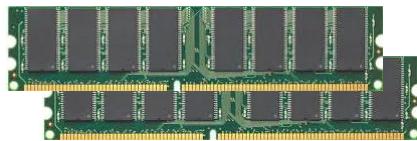


Journal region

Data region

Solution: Batch Updates

Long batch interval
means fewer, but
larger writes to the log

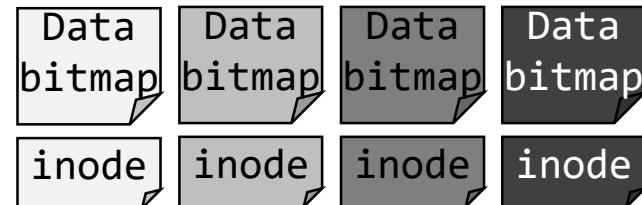
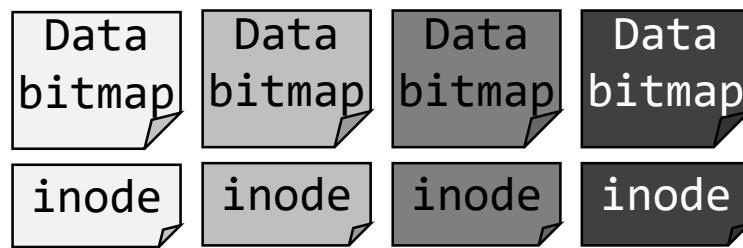
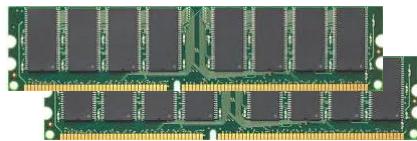


Journal region

Data region



Solution: Batch Updates



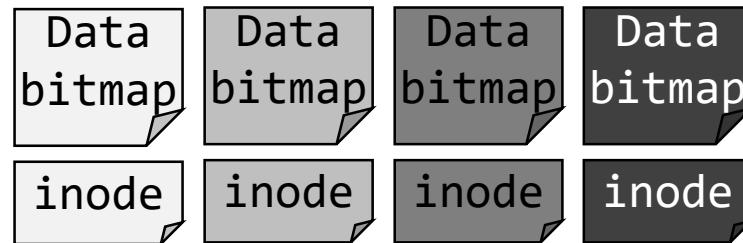
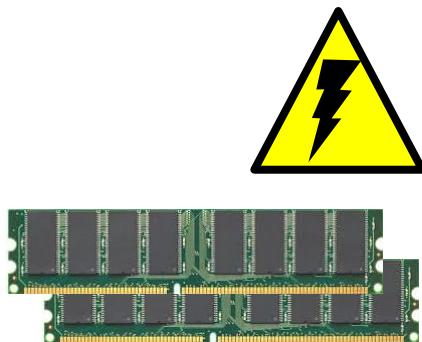
Journal region

Improved
performance due to
large sequential writes

Data region



Solution: Batch Updates



But, if there is a crash,
everything in the
buffer will be **lost**



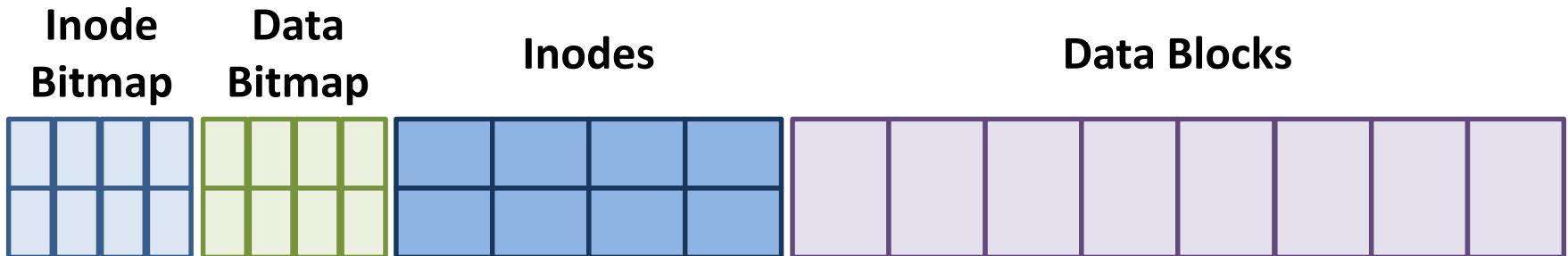
Problem #3: Hard to Log the “Delete”

User scenario

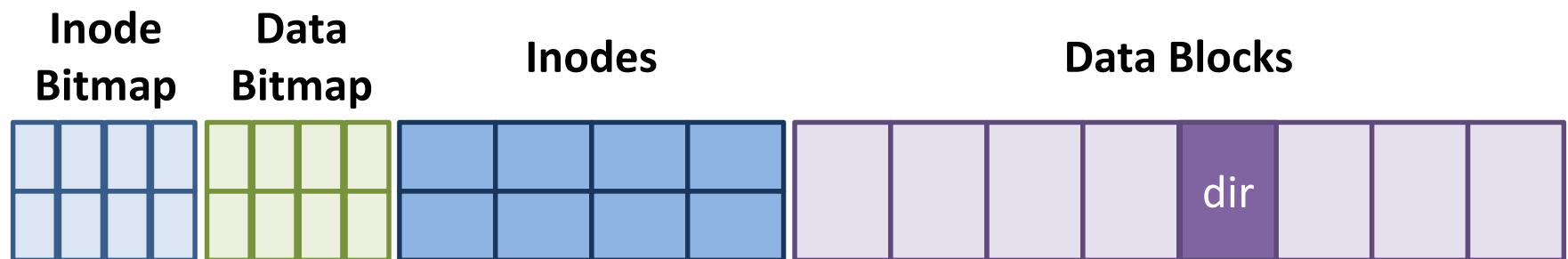
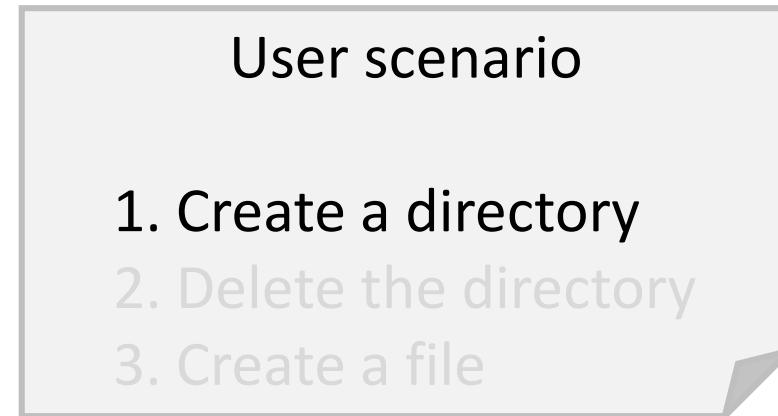
1. Create a directory
2. Delete the directory
3. Create a file

Let's think
about this
user scenario

Journal



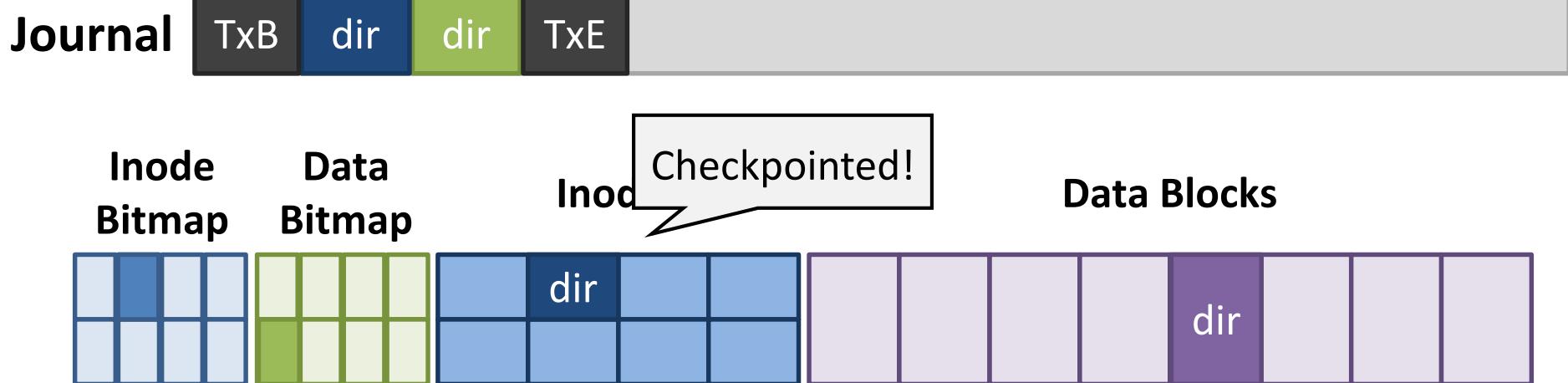
Problem #3: Hard to Log the “Delete”



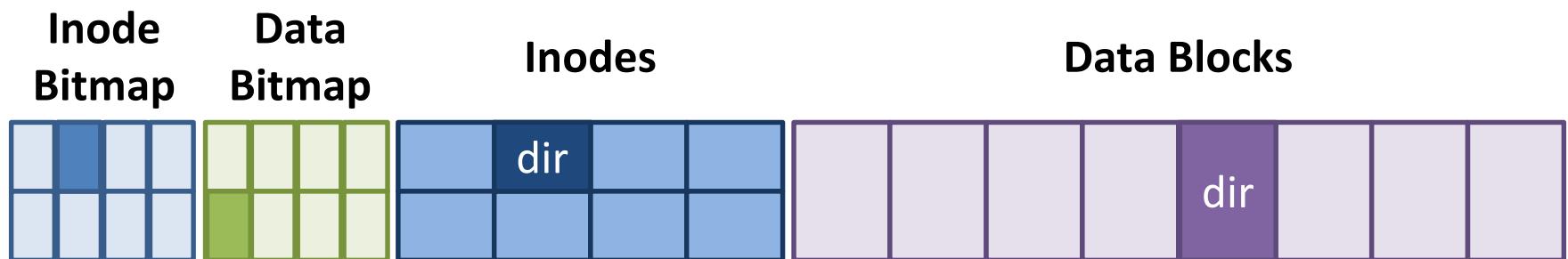
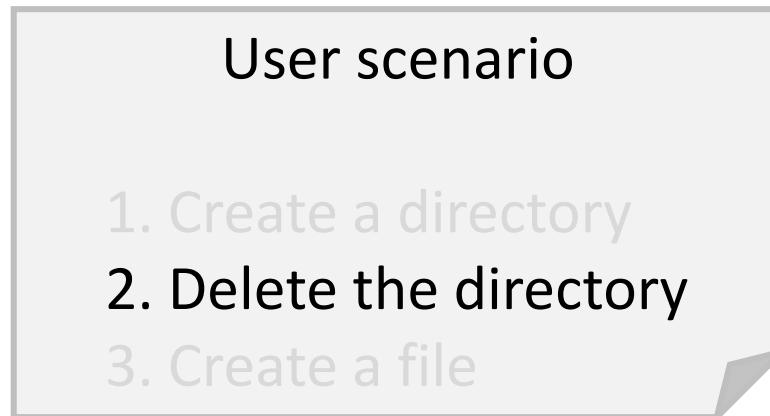
Problem #3: Hard to Log the “Delete”

User scenario

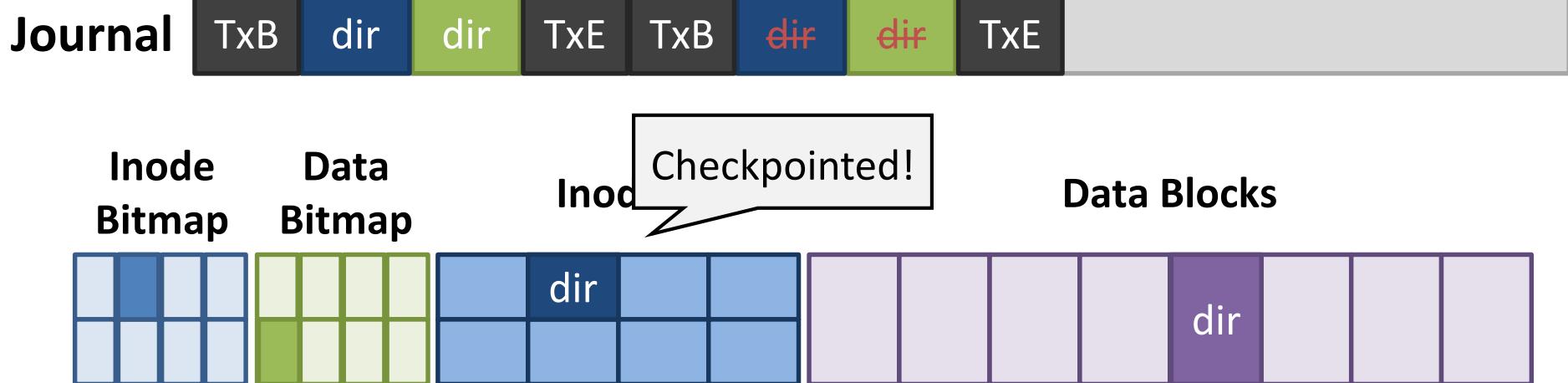
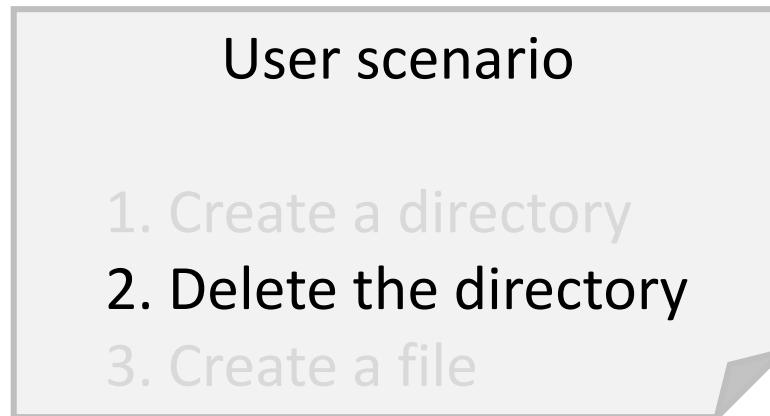
1. Create a directory
2. Delete the directory
3. Create a file



Problem #3: Hard to Log the “Delete”



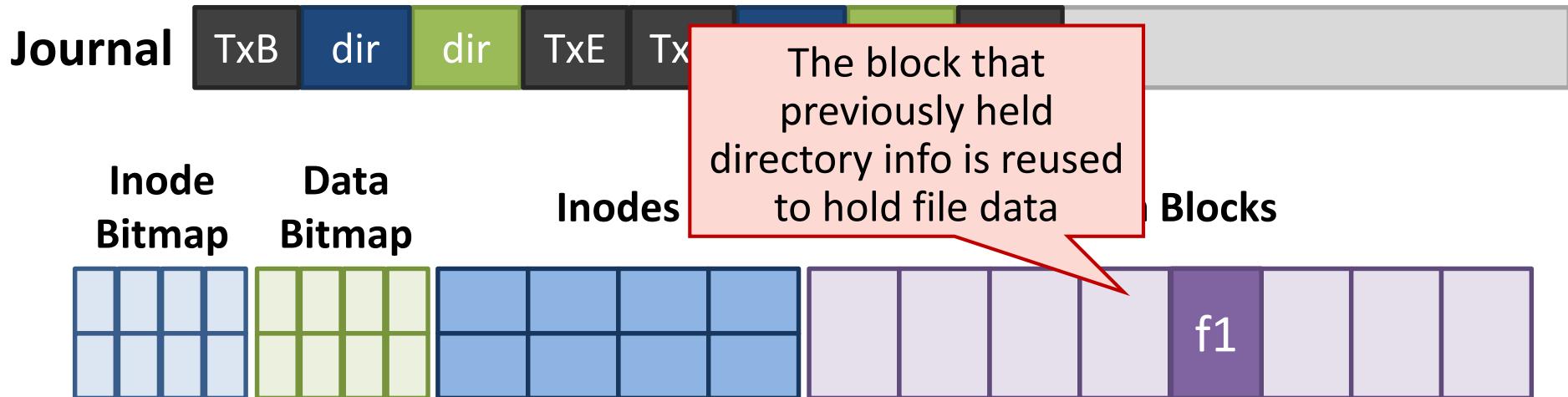
Problem #3: Hard to Log the “Delete”



Problem #3: Hard to Log the “Delete”

User scenario

1. Create a directory
2. Delete the directory
3. Create a file



Problem #3: Hard to Log the “Delete”

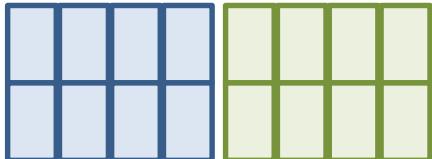
User scenario

1. Create a directory
2. Delete the directory
3. Create a file

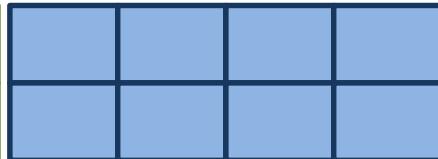
Committed!



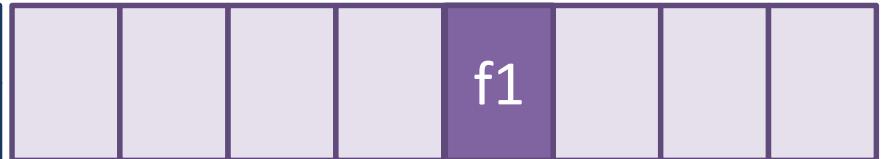
Inode
Bitmap Data
Bitmap



Inodes



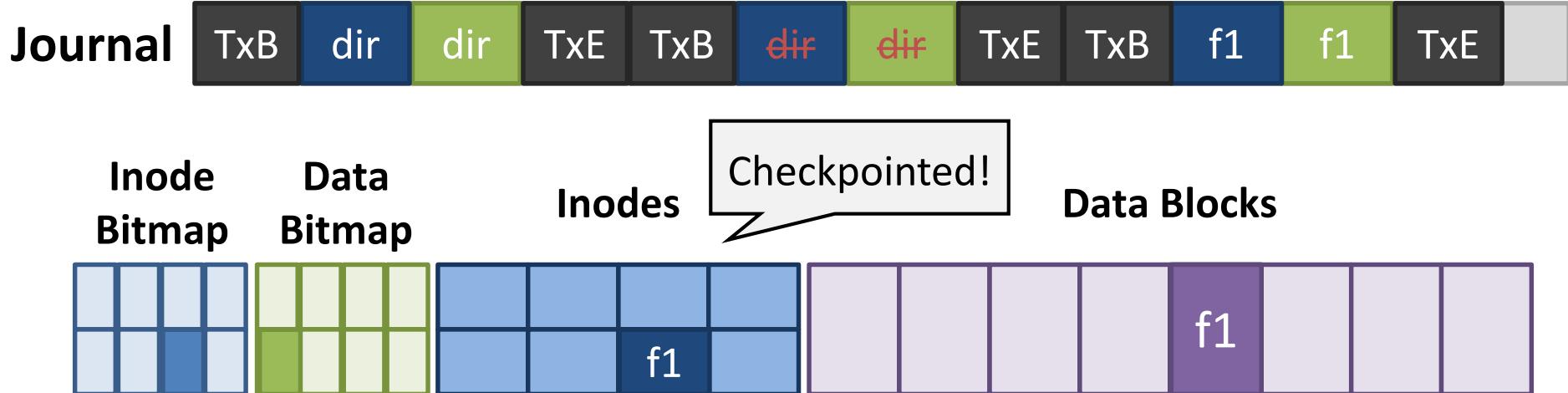
Data Blocks



Problem #3: Hard to Log the “Delete”

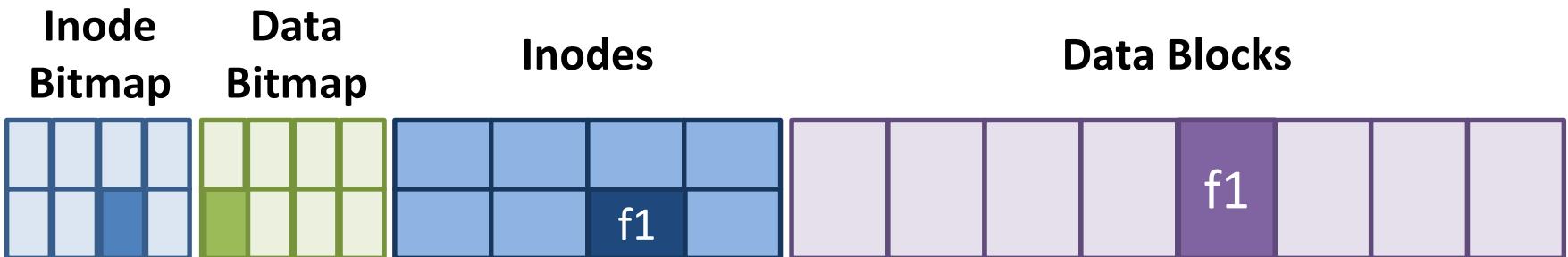
User scenario

1. Create a directory
2. Delete the directory
3. Create a file

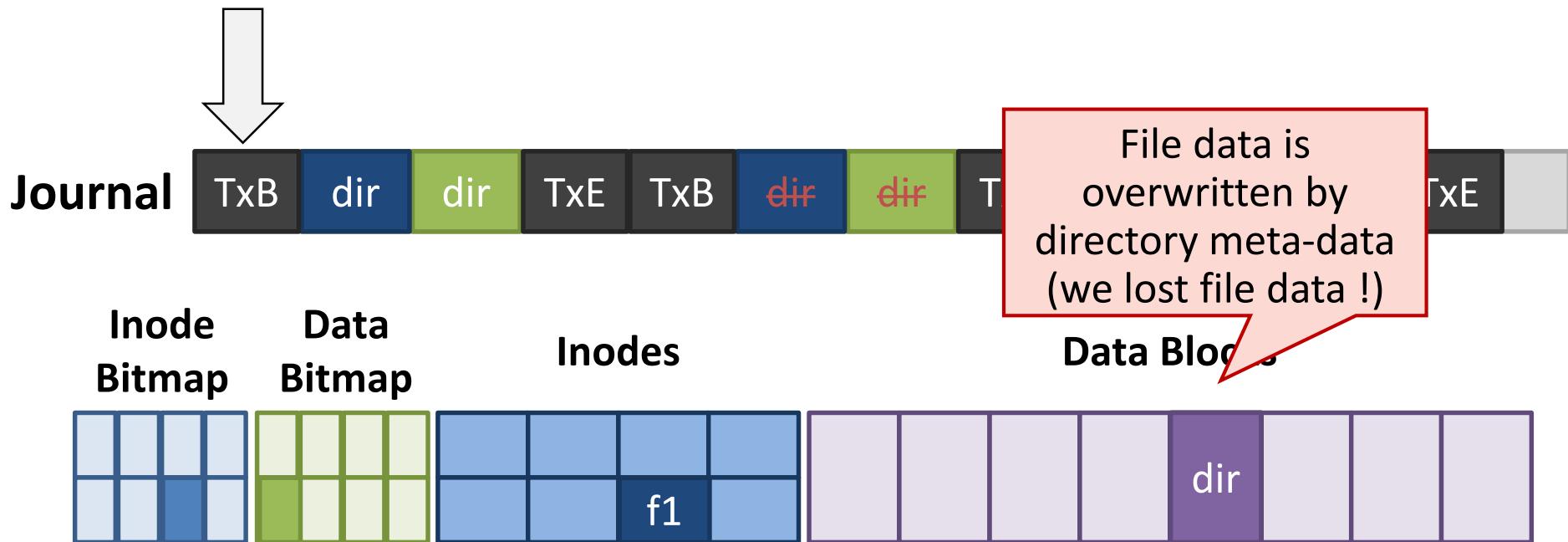


Problem #3: Hard to Log the “Delete”

What happens
when this log is
replayed?



Problem #3: Hard to Log the “Delete”





Solution: Add a Revoke Record

- Ext3 used revoke records

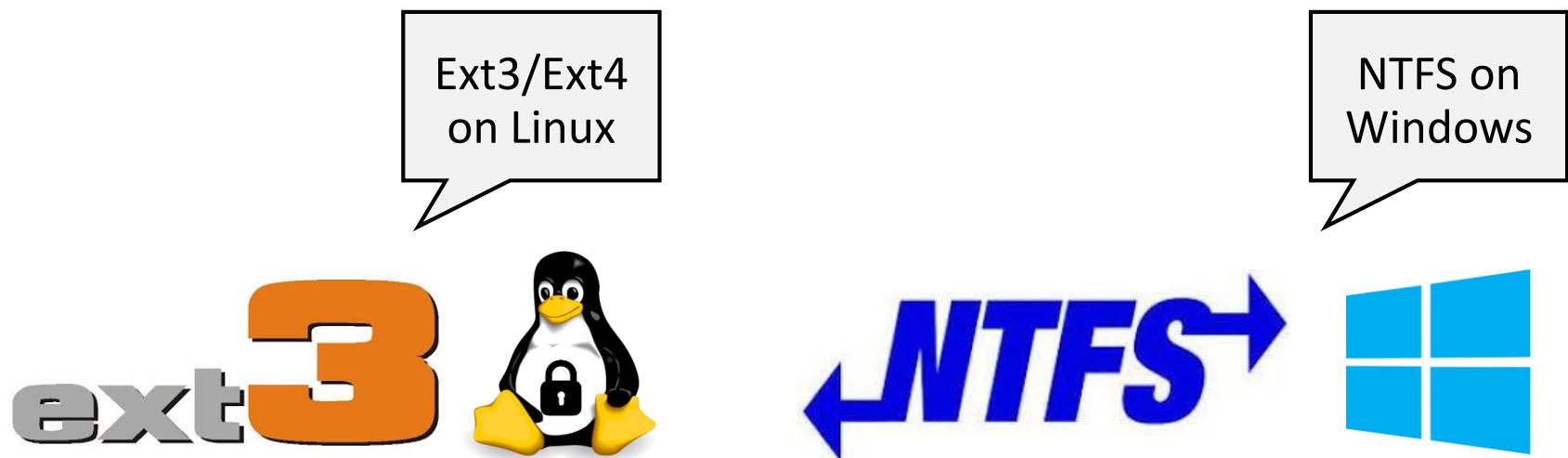
Journal



If the log is
replayed, ignore
transaction ID=1

[Wrapup] Journaling File-System

- Today, most OSes use journaling File-Systems and provides excellent **crash recovery** with relatively low space and performance overhead



What's Next?

- Go back to the drawing board and reevaluated from first-principals (consideration of sequential writes?)

21S EE415

File System IV

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMELab 

File System IV

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab* 

NOTICE: Project#4 is Ready

- Available at KLMS
- Project topic: [File-System](#)
- Deadline: 06/18 (Friday) 23:59:59

전자공학을 위한 운영체제 및 시스템 프로그래밍

Home / Courses / EE415_2021_1 / week 12 / Project 4

Project 4

fspatch.patch	26 May 2021, 5:22 PM
Project4.pdf	26 May 2021, 5:22 PM

Grading summary

Hidden from students	No
Participants	25
Submitted	4
Needs grading	4
Due date	Saturday, 19 June 2021, 12:00 AM
Time remaining	14 days 10 hours

[View all submissions](#) [Grade](#)

NOTICE: Homework #2 is Ready

- Available at KLMS
- Suggestion: Do it before the final exam
- Deadline: 06/16 (Wednesday) 23:59:59

전자공학을 위한 운영체제 및 시스템 프로그래밍

Home / Courses / EE415_2021_1 / week 14 / Homework 2

Homework 2

TBD

Grading summary

Hidden from students	No
Participants	25
Submitted	0
Needs grading	0
Due date	Wednesday, 16 June 2021, 11:59 PM
Time remaining	12 days 10 hours

[View all submissions](#) [Grade](#)

NOTICE: Final Exam

- Untact (non-face-to-face) Exam
- **Schedule:** Wednesday (**6/16**) 1:00 PM – 3:45 PM (this is what the school *officially* assigned for this lecture. No change or exception)
- **Scope:**
Lecture 13 ~ Lecture 25 (File System)

Notice

Please check the detailed guidelines at KLMS

Guideline for Untact (non-face-to-face) Exam

작성자 : 권미령

작성일 : 2021년 4월 12일(월요일) 오후 3:23 조회수 : 95

Schedule

Wednesday (6/16) 1:00 PM - 3:45 PM

- Closed book
- No early answer sheet submission (You should stay Zoom until exam end)
- If you have any questions on the Exam, please type your question in English via Zoom chatting. Korean does not allow.

NOTICE: Final Exam

Student ID	Name	TA
20110273	김성구	Hanyeoreum Bae
20140300	송영준	Hanyeoreum Bae
20150114	김동언	Hanyeoreum Bae
20150816	홍지승	Hanyeoreum Bae
20160265	박종회	Hanyeoreum Bae
20160466	이선호	Hanyeoreum Bae
20170327	서준석	Hanyeoreum Bae
20170353	신경섭	Hanyeoreum Bae
20170413	우상민	Hanyeoreum Bae
20170534	임석빈	Hanyeoreum Bae
20170628	조우석	Hanyeoreum Bae
20170651	자승훈	Hanyeoreum Bae
20180046	김경민	Hanyeoreum Bae
20180498	이준	Miryeong Kwon
20180545	임진영	Miryeong Kwon
20180662	자준서	Miryeong Kwon
20180710	한승윤	Miryeong Kwon
20190028	고형석	Miryeong Kwon
20203234	박상수	Miryeong Kwon
20204409	오동석	Miryeong Kwon
20208252	JULIAN TROY VALDEZ	Miryeong Kwon
20213007	강승관	Miryeong Kwon
20213408	윤정한	Miryeong Kwon
20214207	RAJENDRAN PRAVEEN KUMAR	Miryeong Kwon
20215356	최진수	Miryeong Kwon

Please double check your TA
for as follows and use your
TA's Zoom link for the exam

(All these have been also
announced in KLMS)

<https://docs.google.com/spreadsheets/d/1p3QuNjpd-2NeGyPGCCgh1on2TDPCAjY-mGThMXeGzI8/edit#gid=0>

[Recap] File System by Far

We can easily find out all the location and necessary information thanks for FS's metadata `inode` and `directory`

The file system provides an excellent level of **ACID (atomicity, consistency, isolation, durability)** and resilient against corruption; thanks for “**journaling**”



Revisiting File System

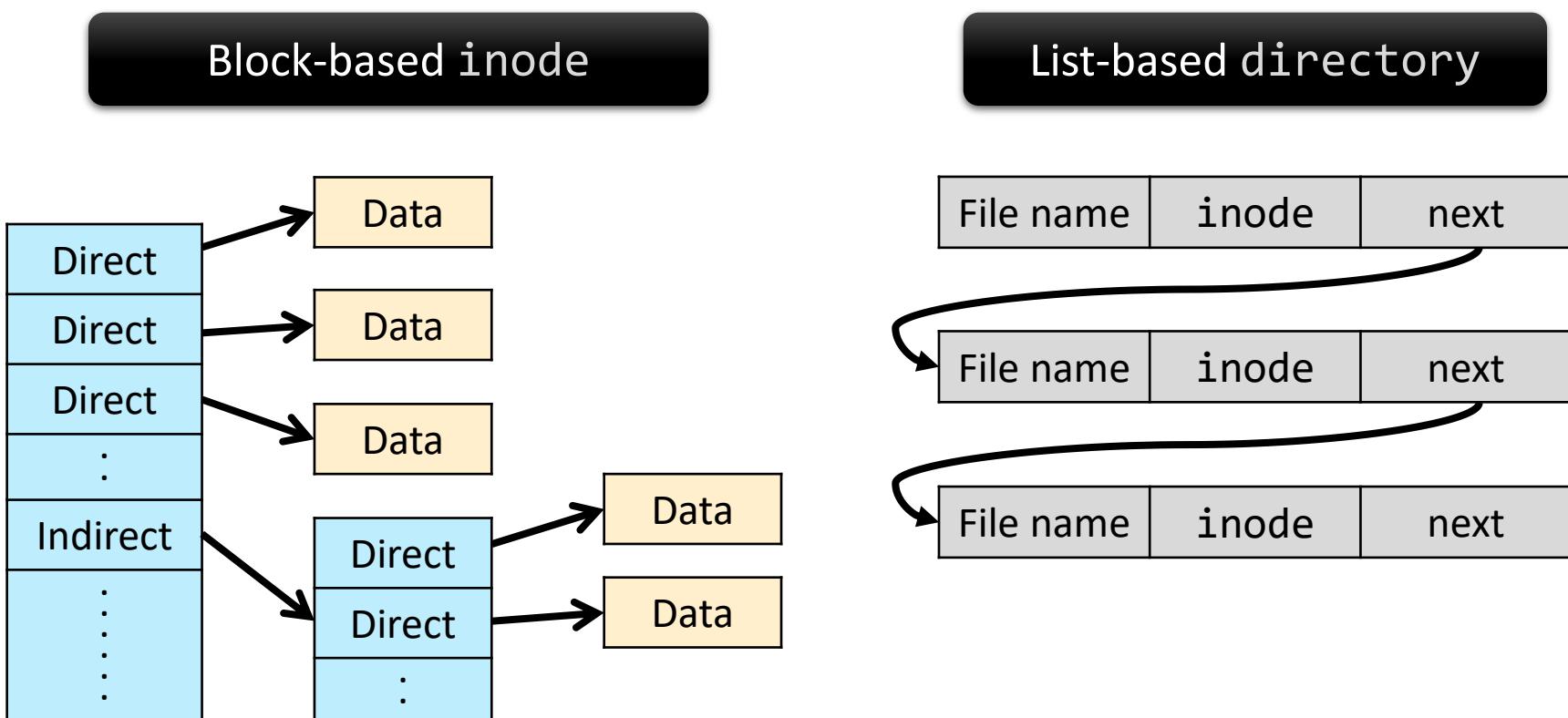
Wait.. Are we done with all the things for file system?

The concept of file system that we learned by far is FAT and EXT3. Is there something that we can improve further?



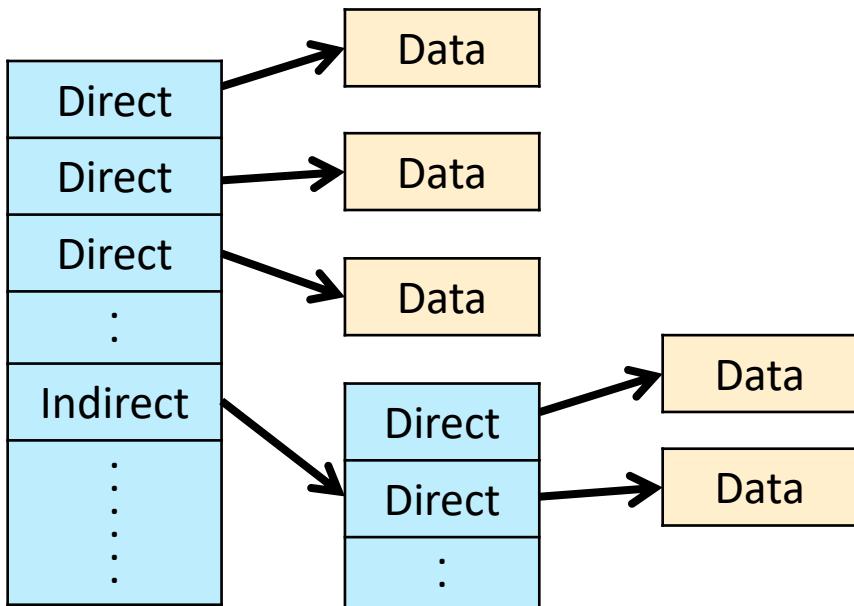
Revisiting File System

- The current design of file system metadata (inode and directory)



Revisiting #1: inodes

Block-based inode



[Recall]: inodes use
indirections to
manage additional
blocks of a file
(greater than $4KB * 12$)

linked directory





Problem: Not Efficient for Large Files

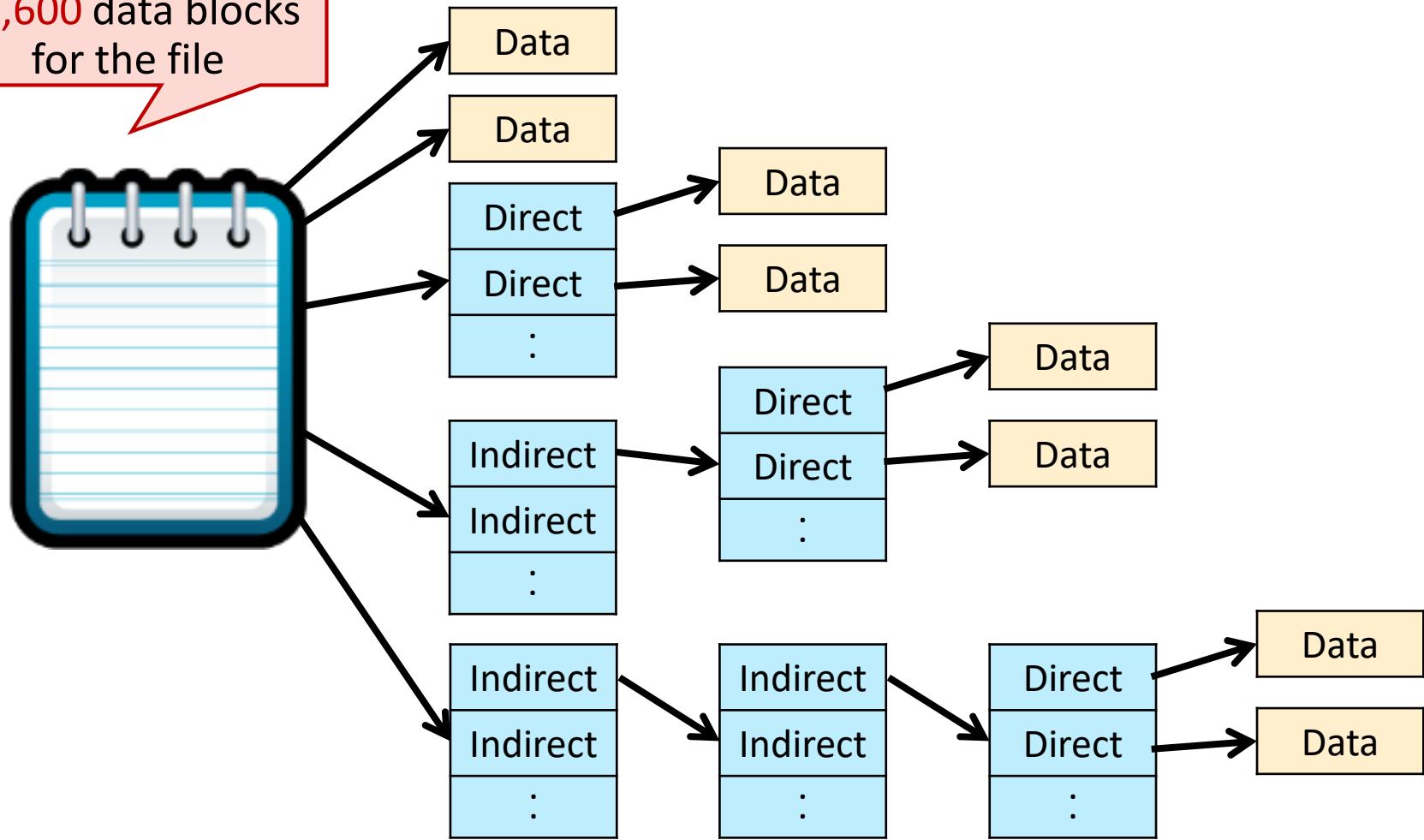


How many blocks do we
need to handle a 100MB file?
-- Let's assume that the size
of data block is 4KB.



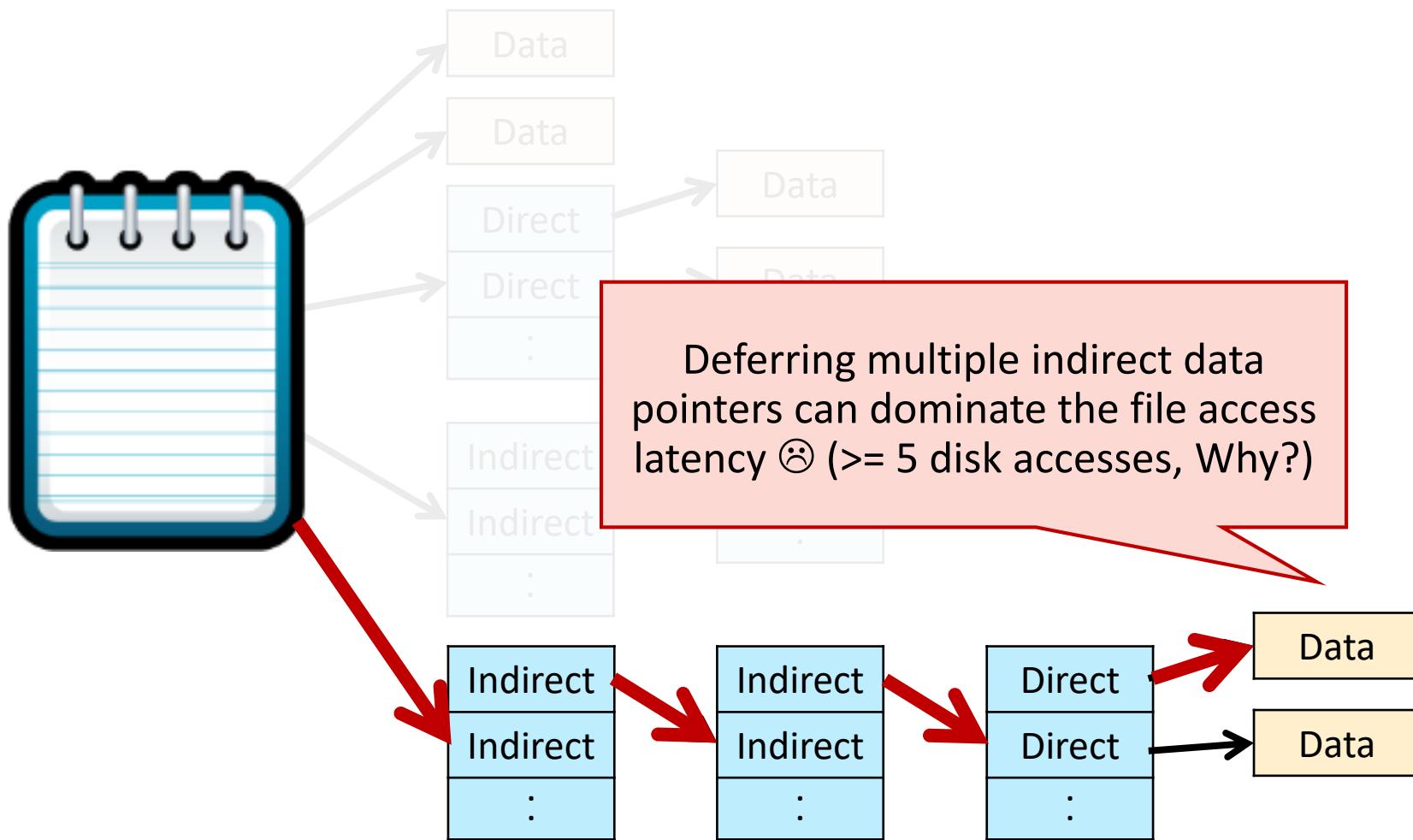
Problem: Not Efficient for Large Files

You need to handle
25,600 data blocks
for the file

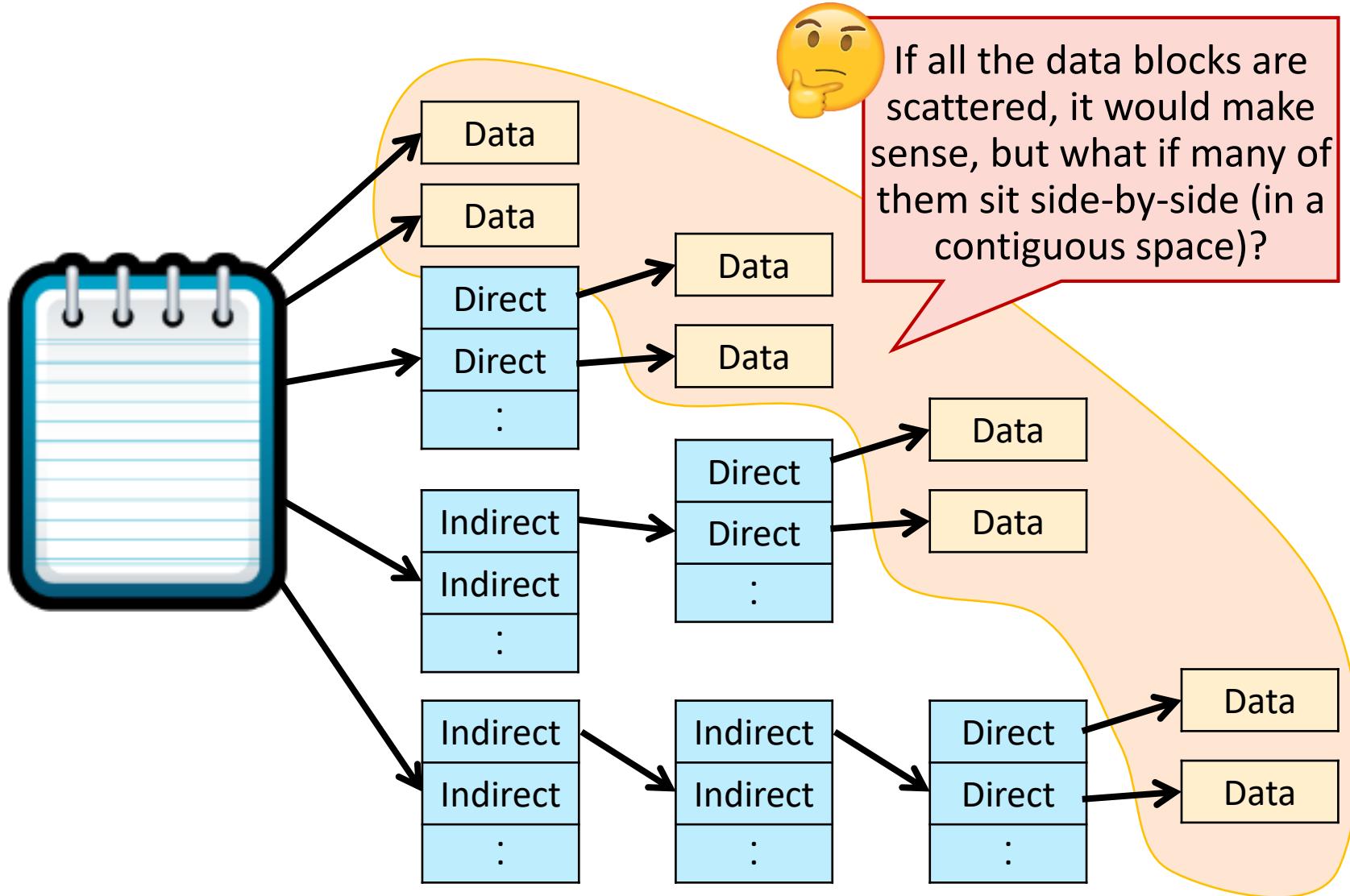




Problem: Not Efficient for Large Files



Problem: Not Efficient for Large Files



➤ Solution: From Pointers to Extents

Ext4 (Linux) and NTFS (Windows) use [extents](#) instead of the data block pointers

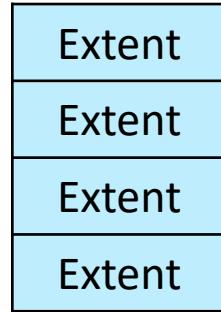


Extent-based inode

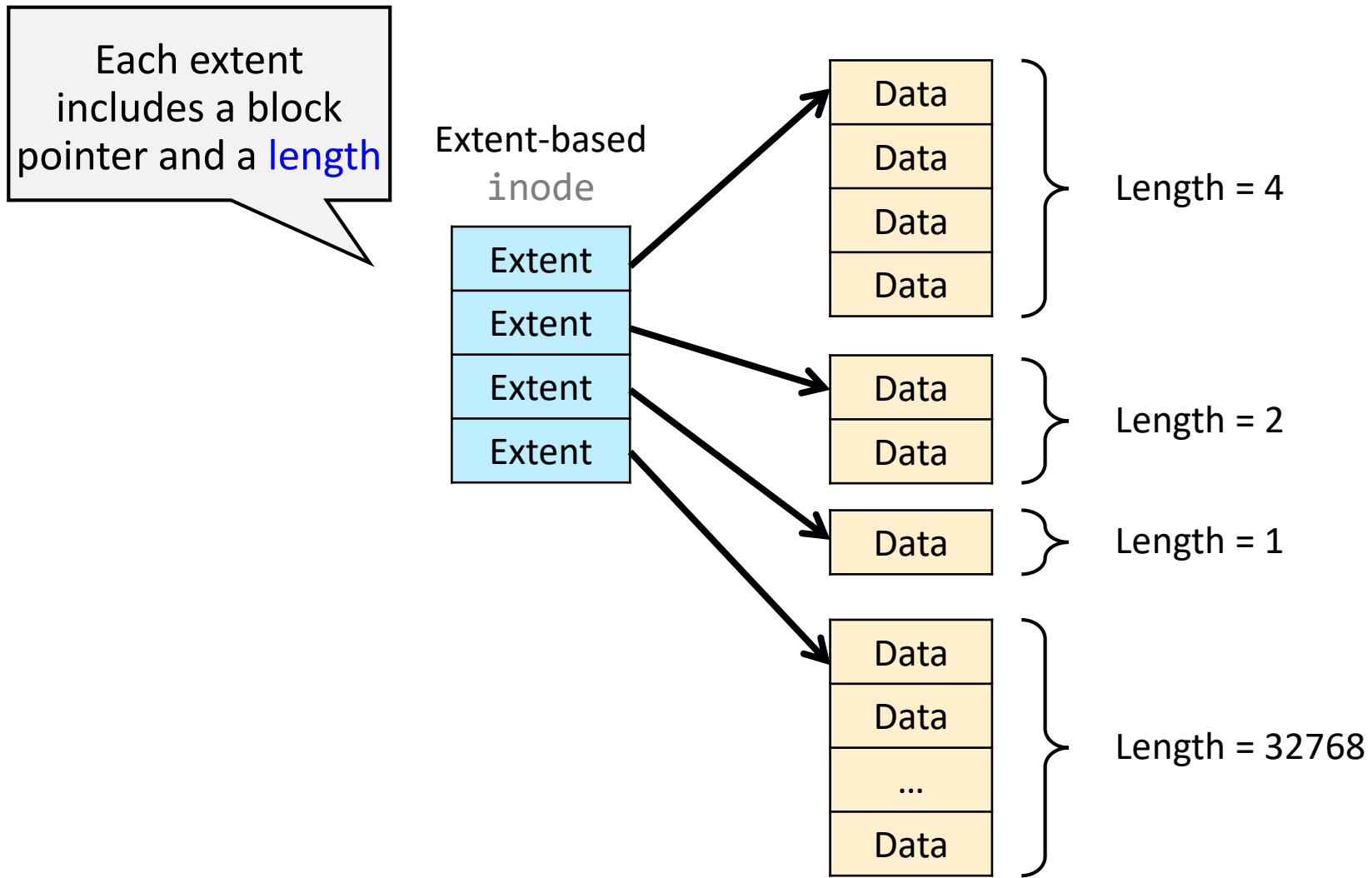
Ext4 inodes consist of **4 extents** instead of 15 block pointers that block-based inodes use



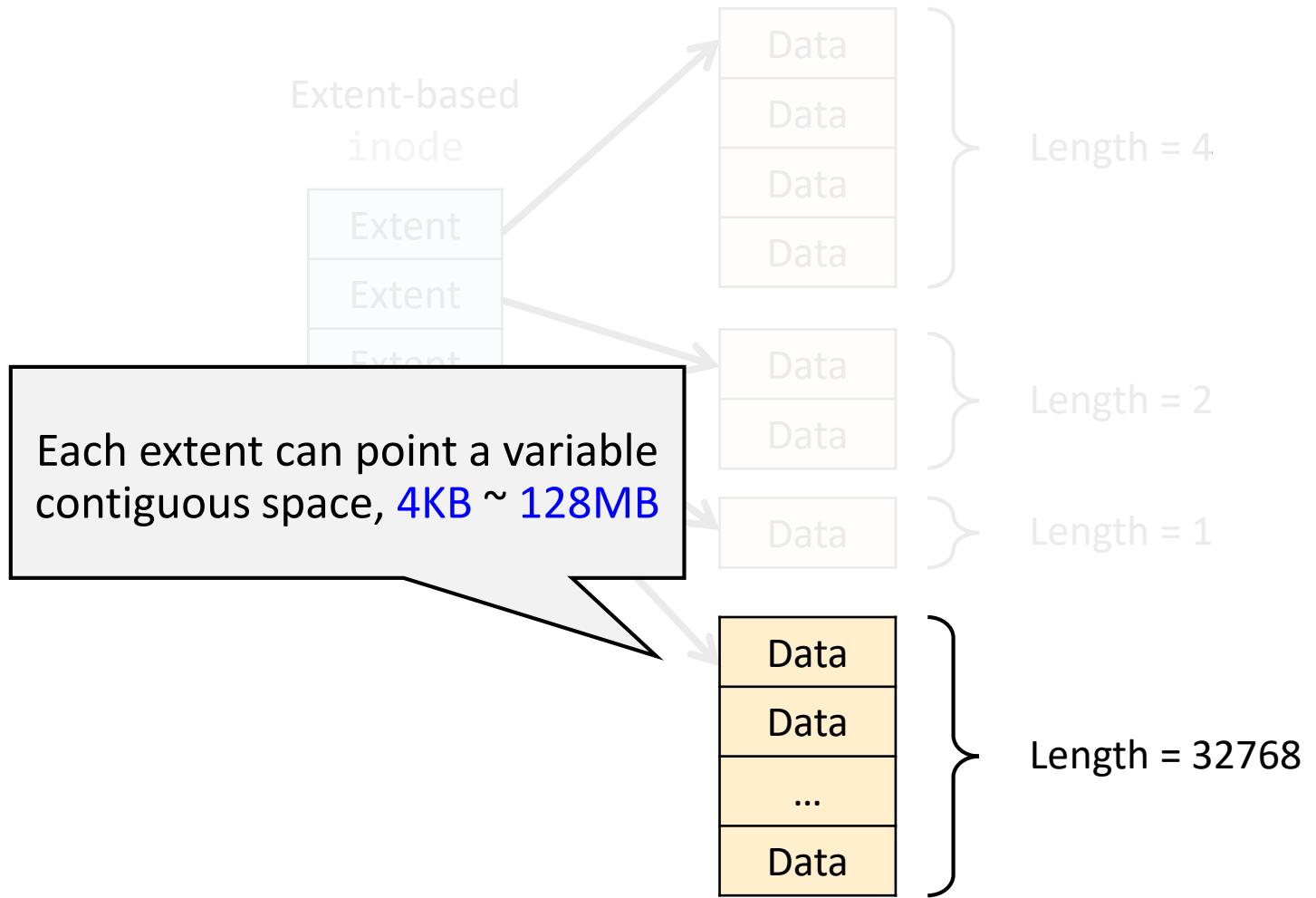
Extent-based
inode



Extent-based inode



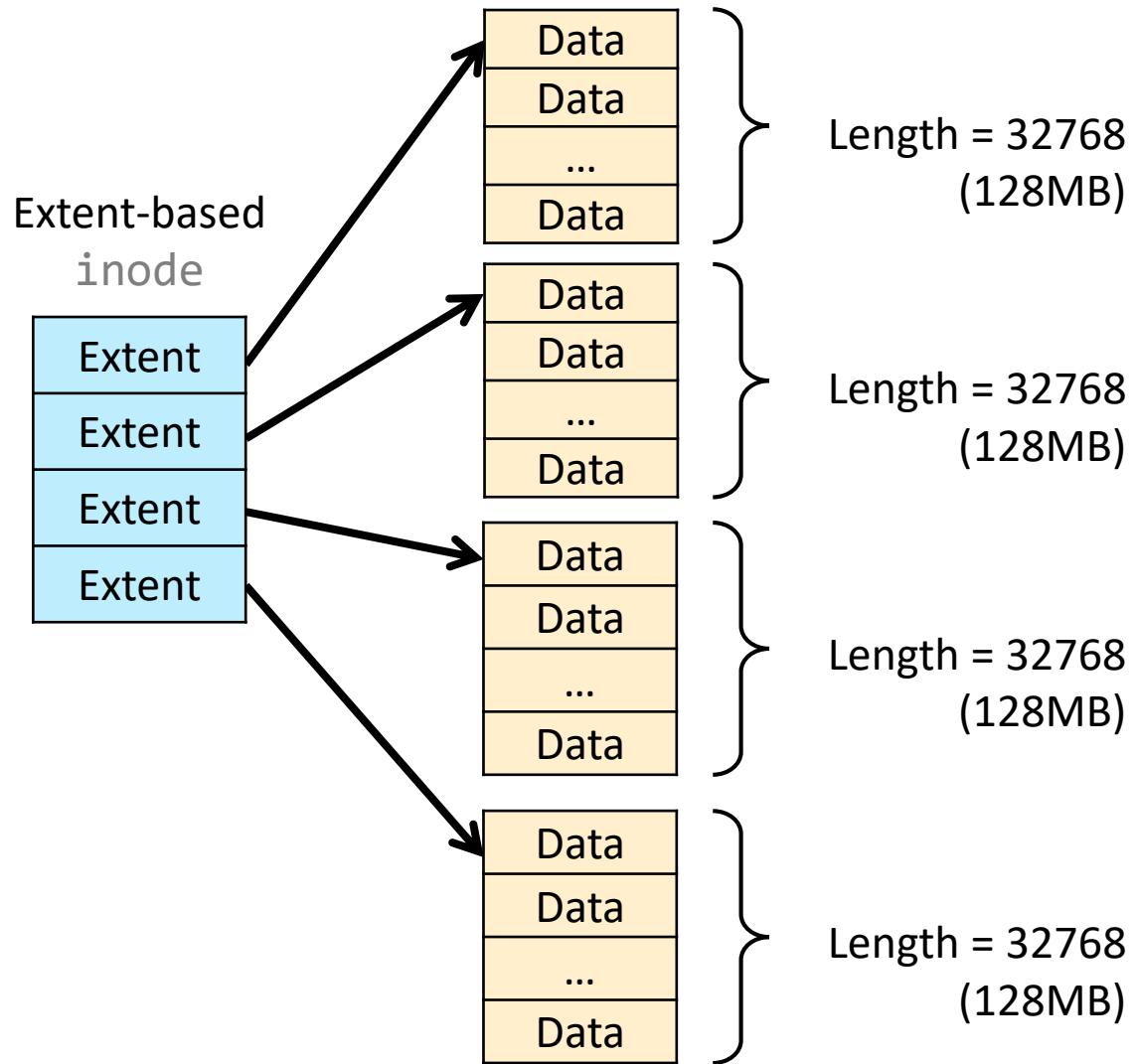
Extent-based inode



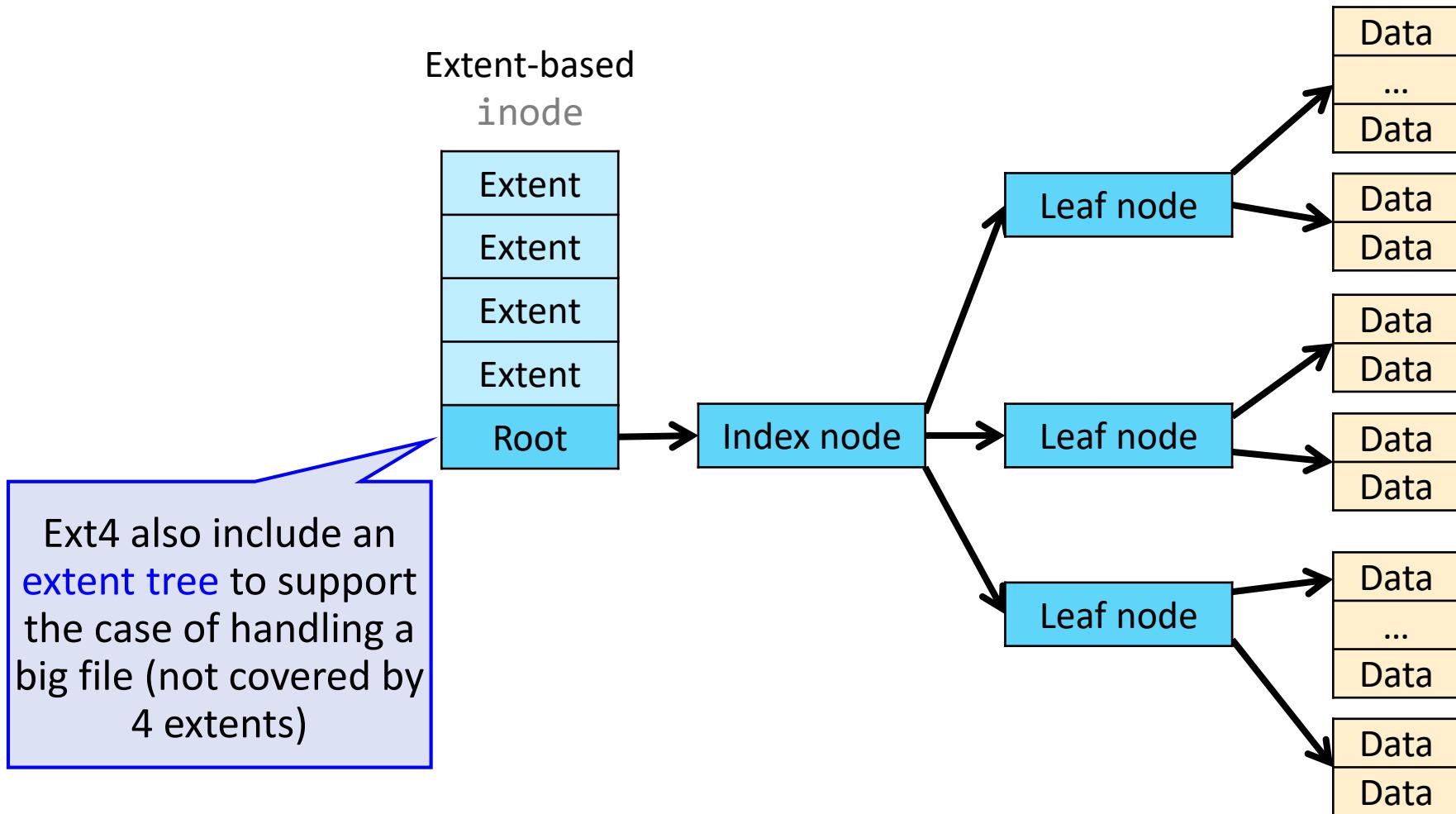
Extent-based inode



Hmm... 4 extents
only can support
512MB file at most

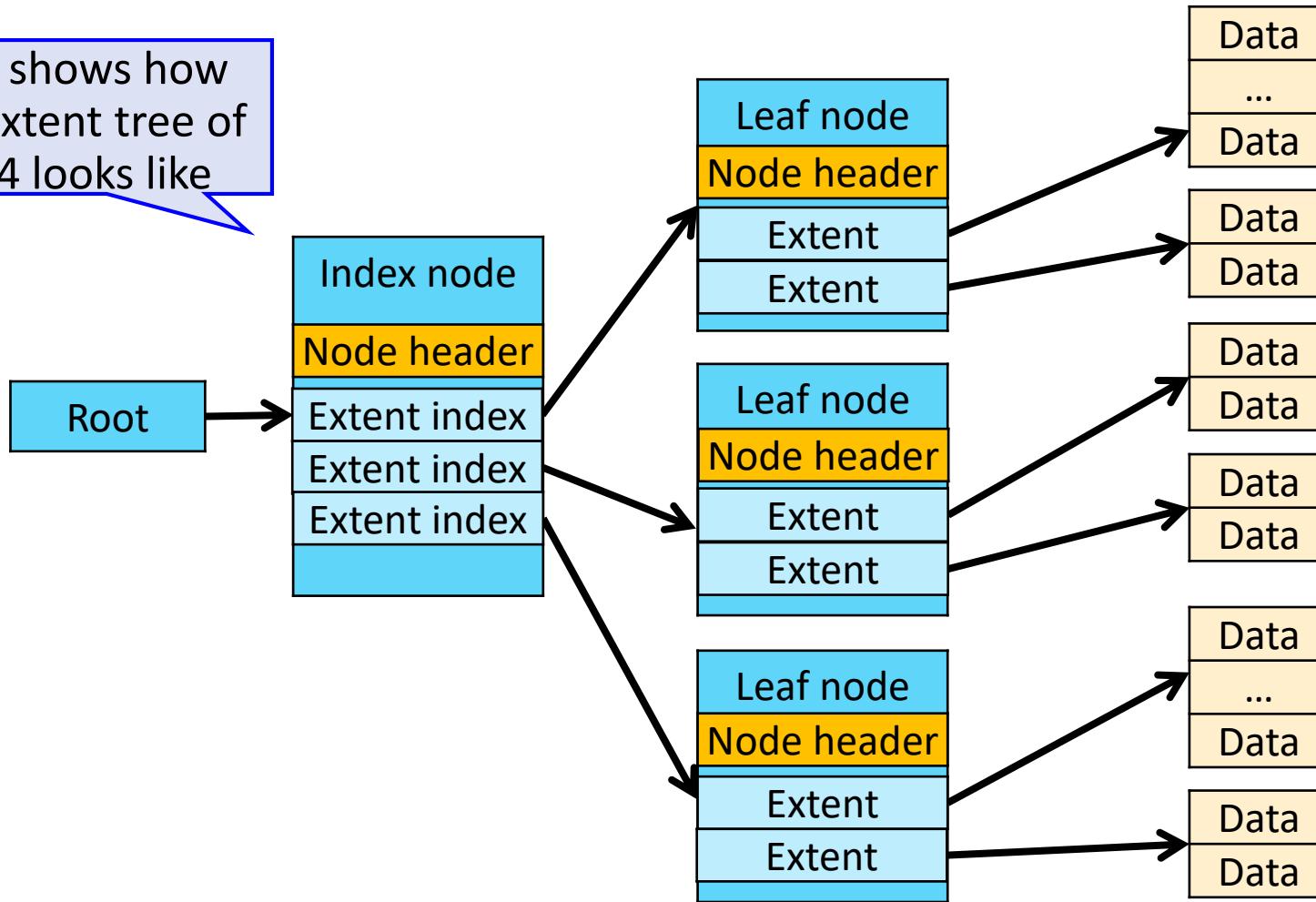


Extent-based inode

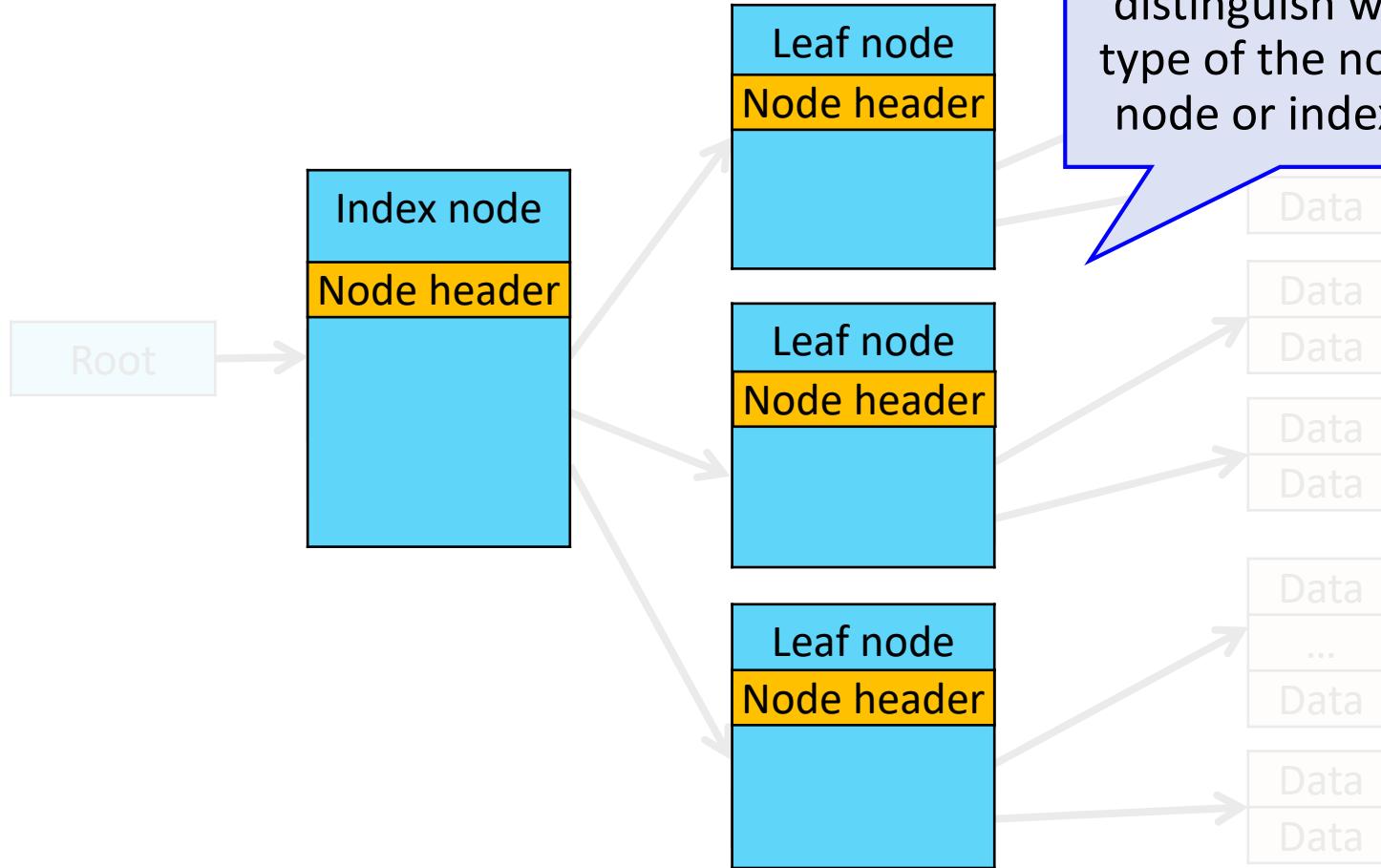


Extent-based inode

This shows how
the extent tree of
Ext4 looks like

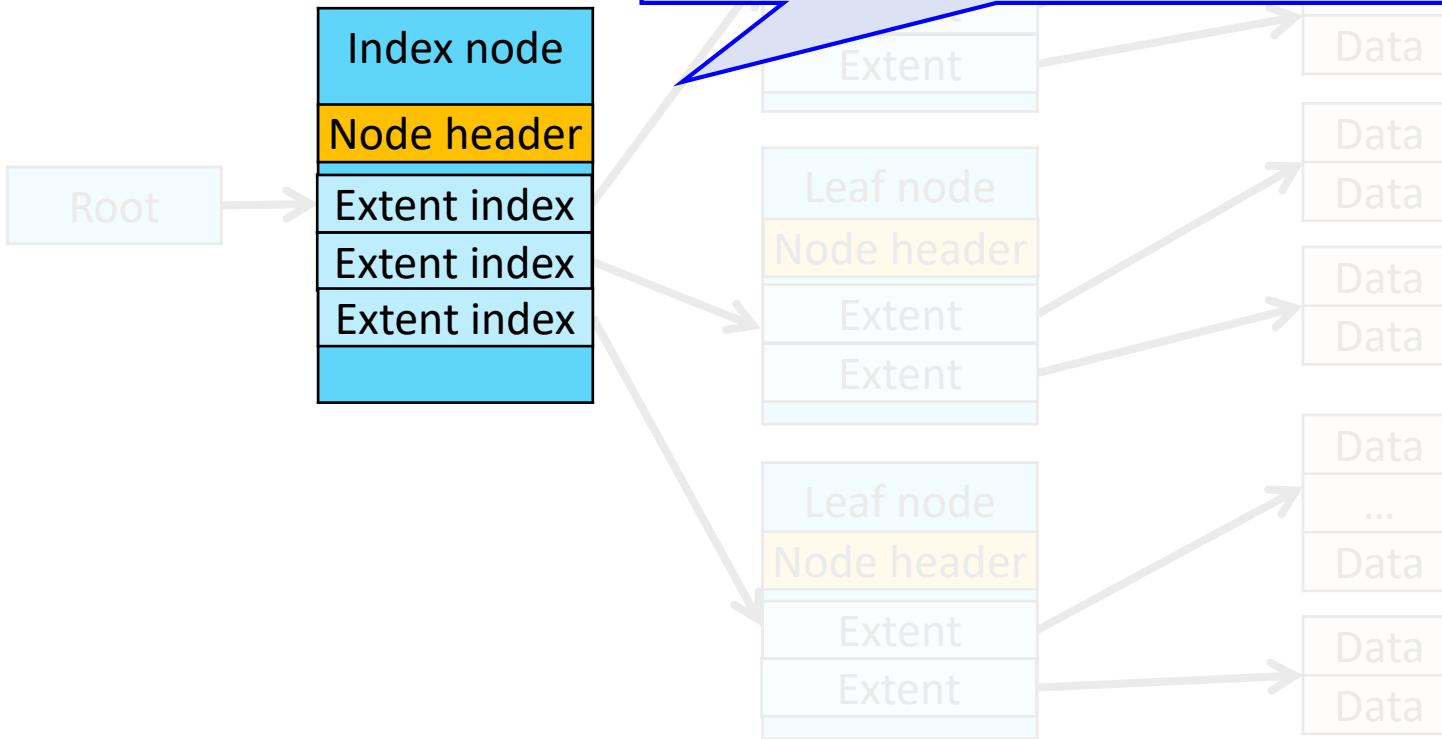


Extent-based inode

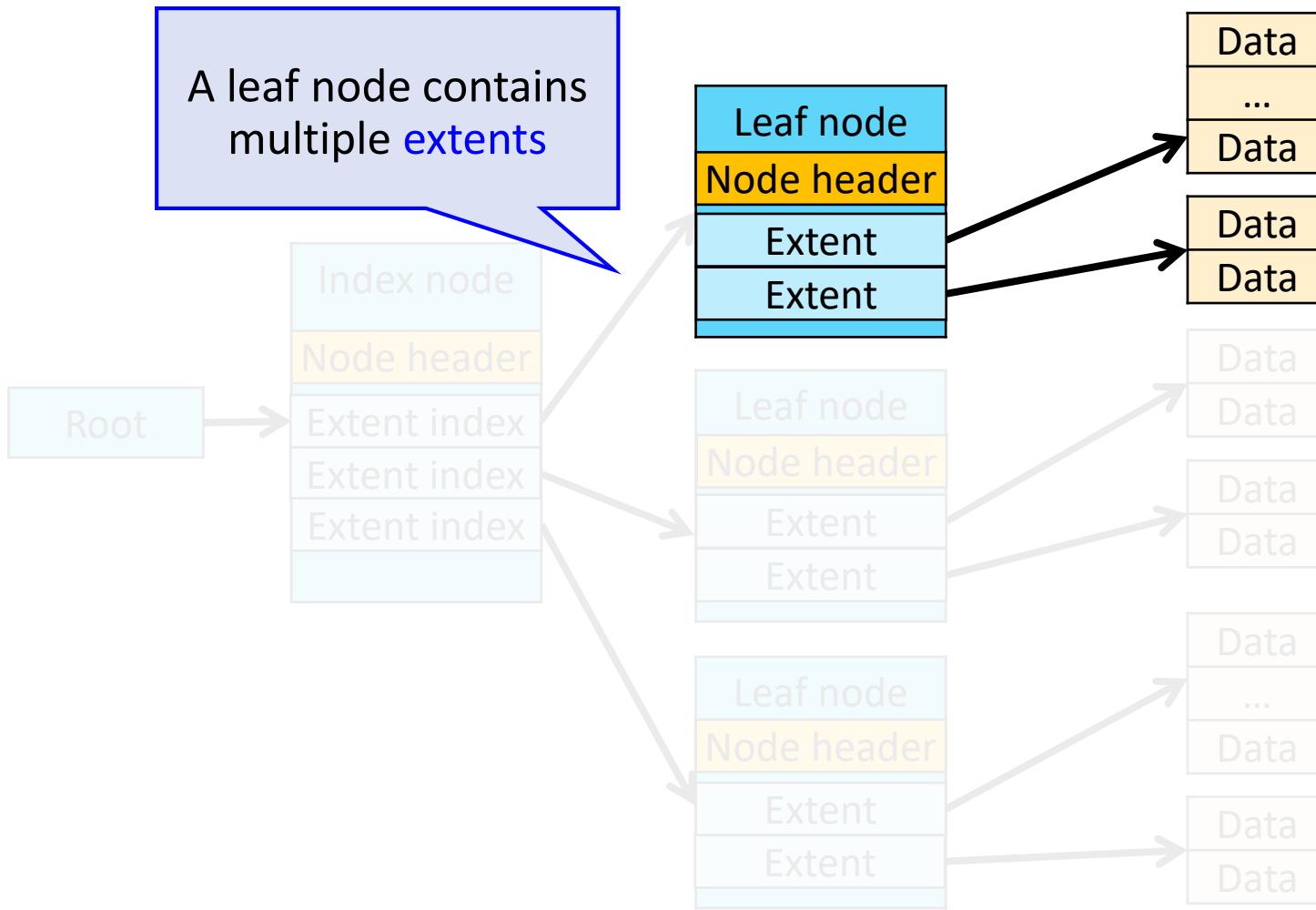


Extent-based inode

The index node consists of multiple **extent indices**, each containing the address of the target **leaf node**



Extent-based inode



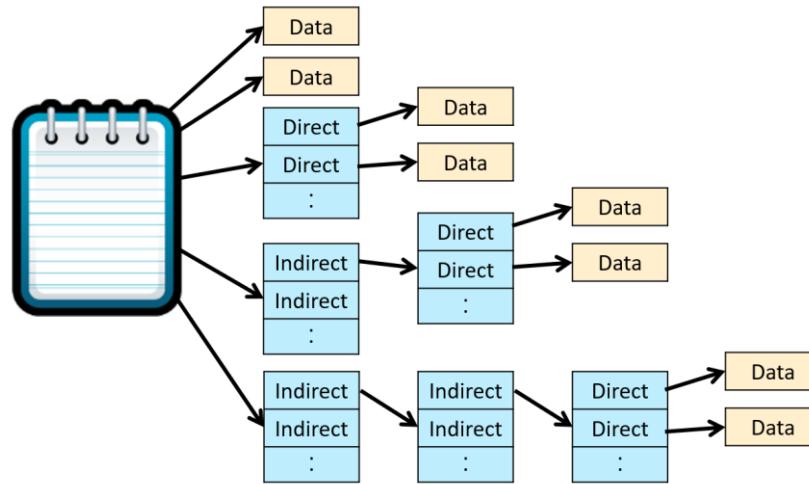


Indirect Mapping (Ext3) vs. Extent (Ext4)

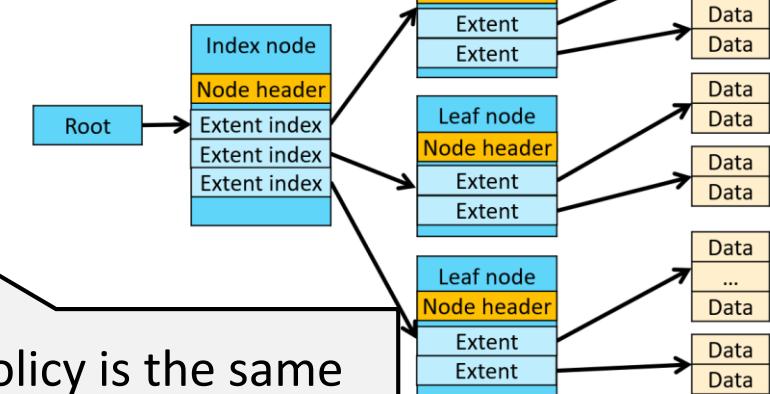
ext3



Ext4



The policy is the same between Ext3 and Ext4, but give a bit more shot to a large file by changing the inode structure

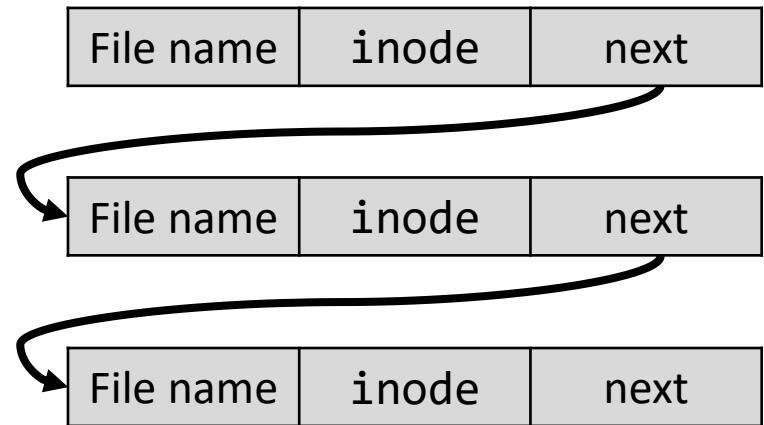
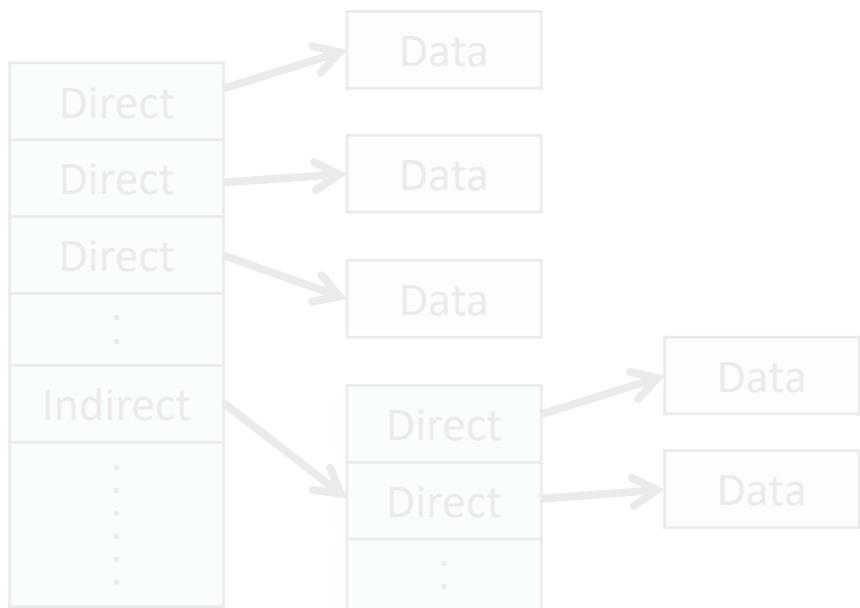


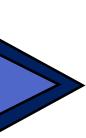
Revisiting #2: Directories

B

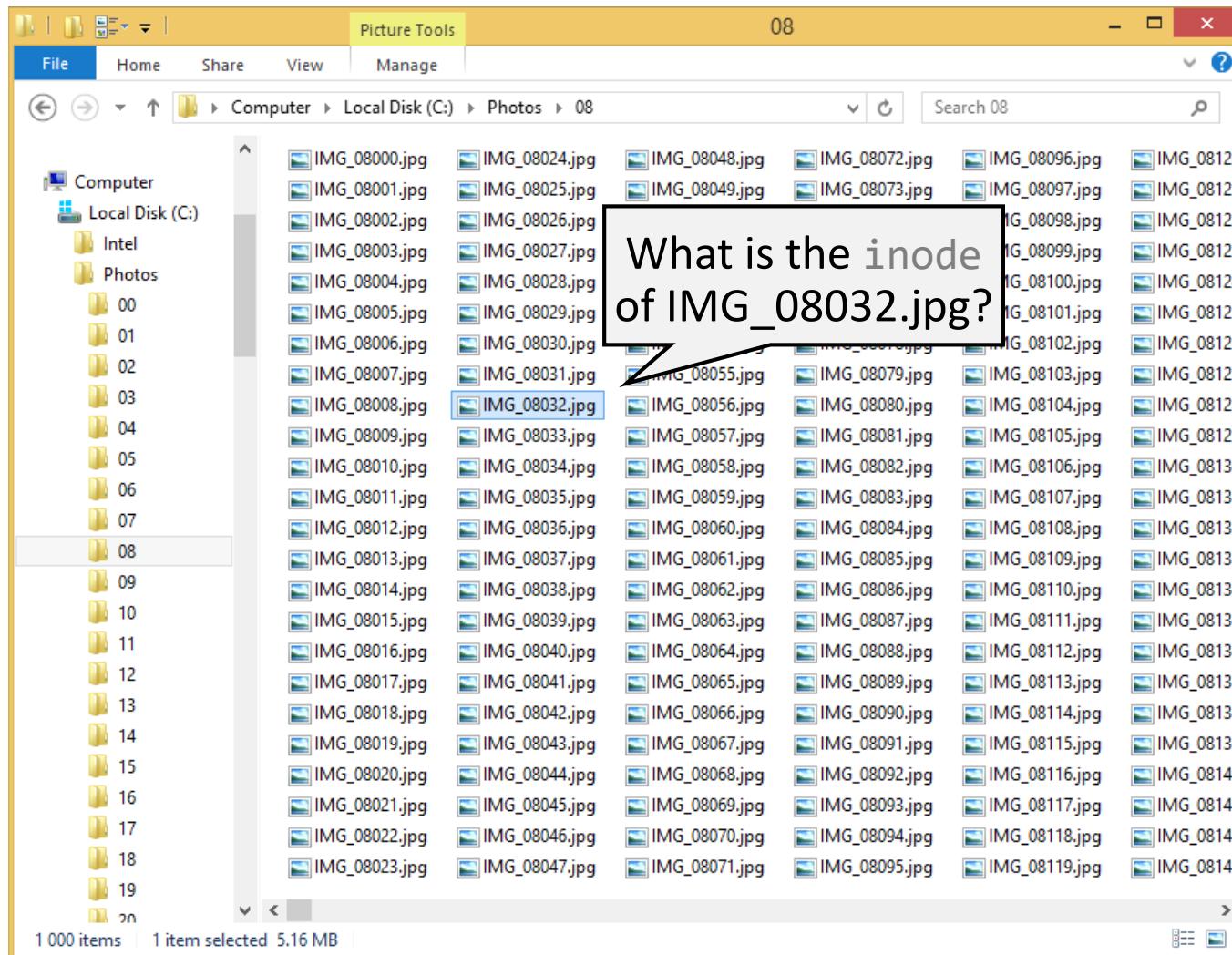
[Recall]: A directory is a file including a list of entries, which contain the file name and inode in an unordered manner.

List-based directory



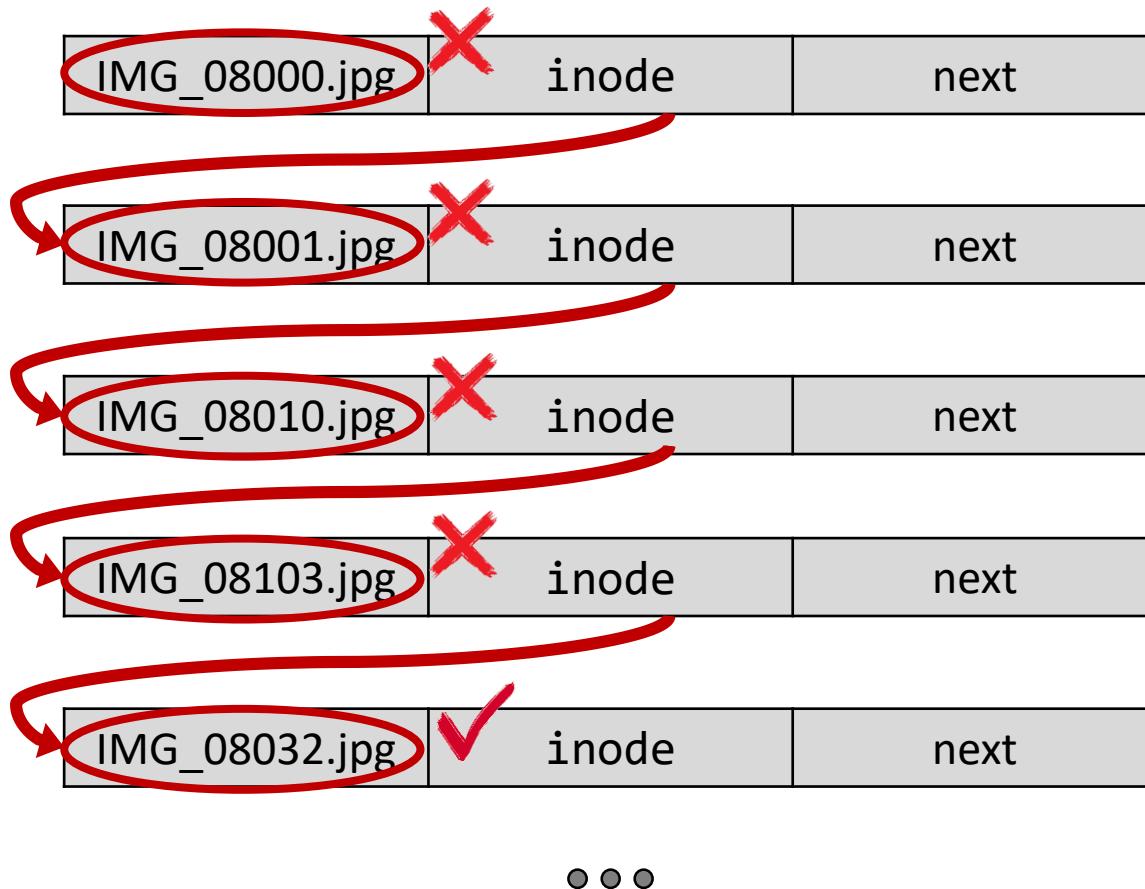


Problem: $O(n)$ File Searching Time





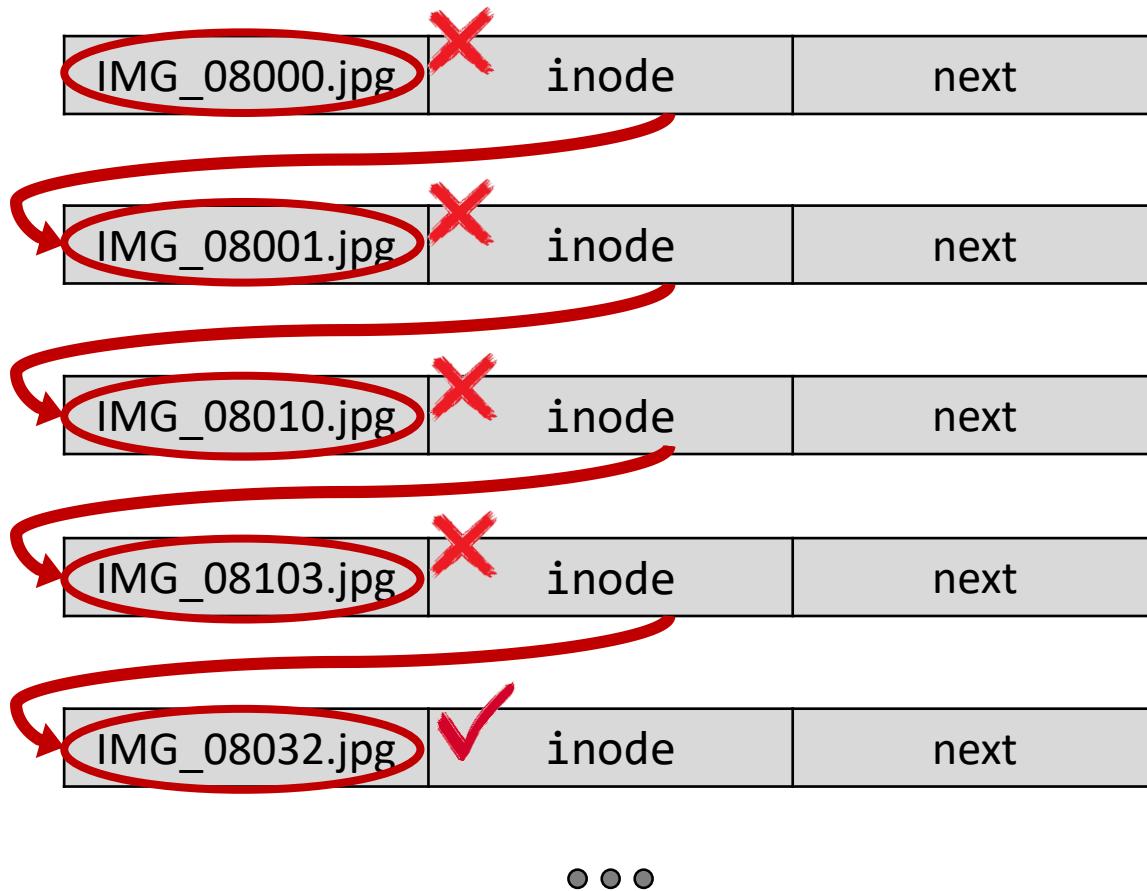
Problem: O(n) File Searching Time



Let's search the
C:/Photos/08
directory



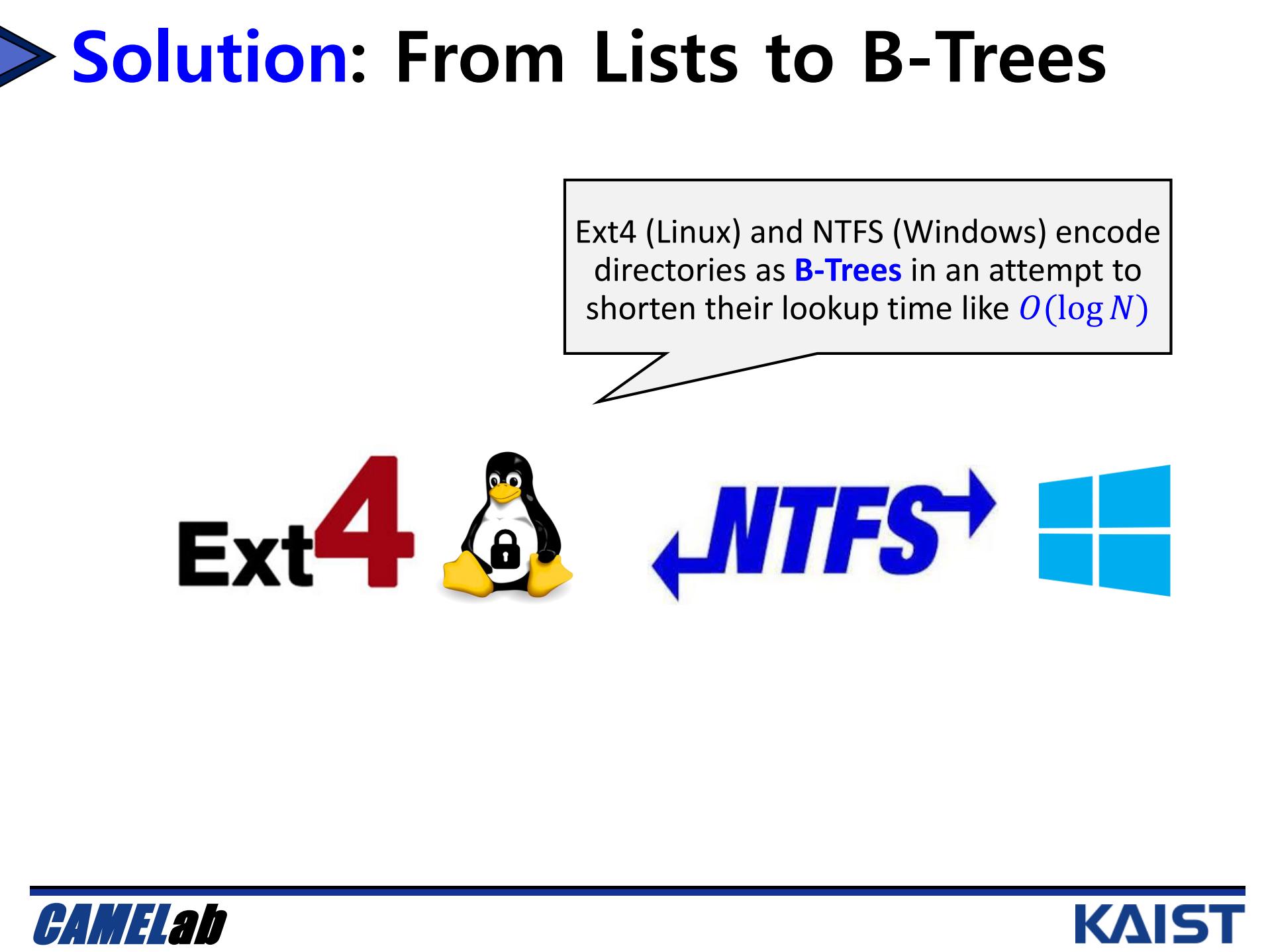
Problem: O(n) File Searching Time



Based on the file name rule, **10K files** may be in a directory. What if **IMG_08032.jpg** exists at the end of the list?

It takes way too long to locate and open files 😞



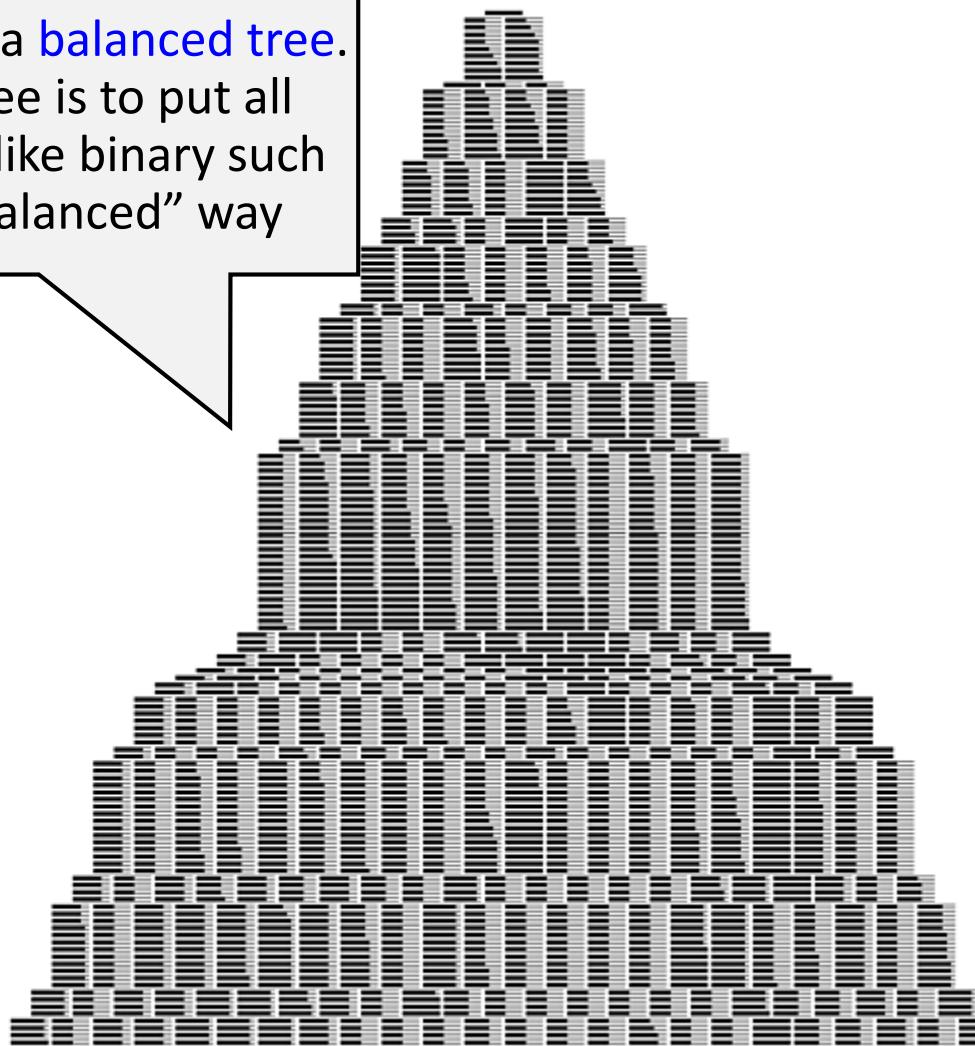




B-Tree based Directory

B-Tree is a type of a **balanced tree**.

The goal of B-tree is to put all items in the tree like binary such tree but in a “balanced” way

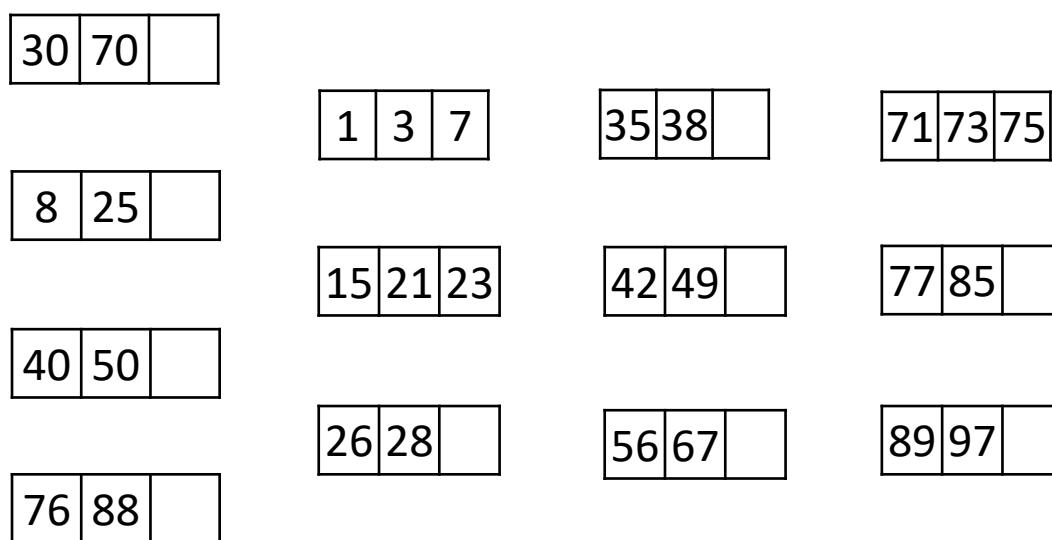


B-Tree based Directory

- **Order (m)**: the upper bound on the number of children. i.e., the maximum number possible.
- **Degree**: the number of children it has

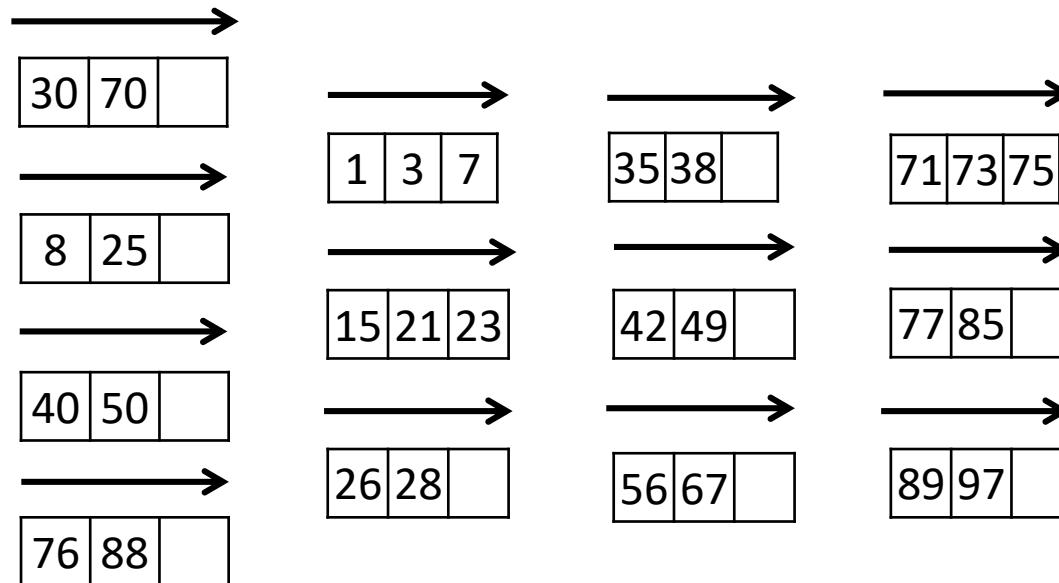
In this example, the order of B-tree is 3 ($m = 3$)

Each node should have more than $m/2$ items ($\lfloor 3/2 \rfloor = 1$)



B-Tree based Directory

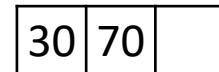
In addition, all items per node are sorted



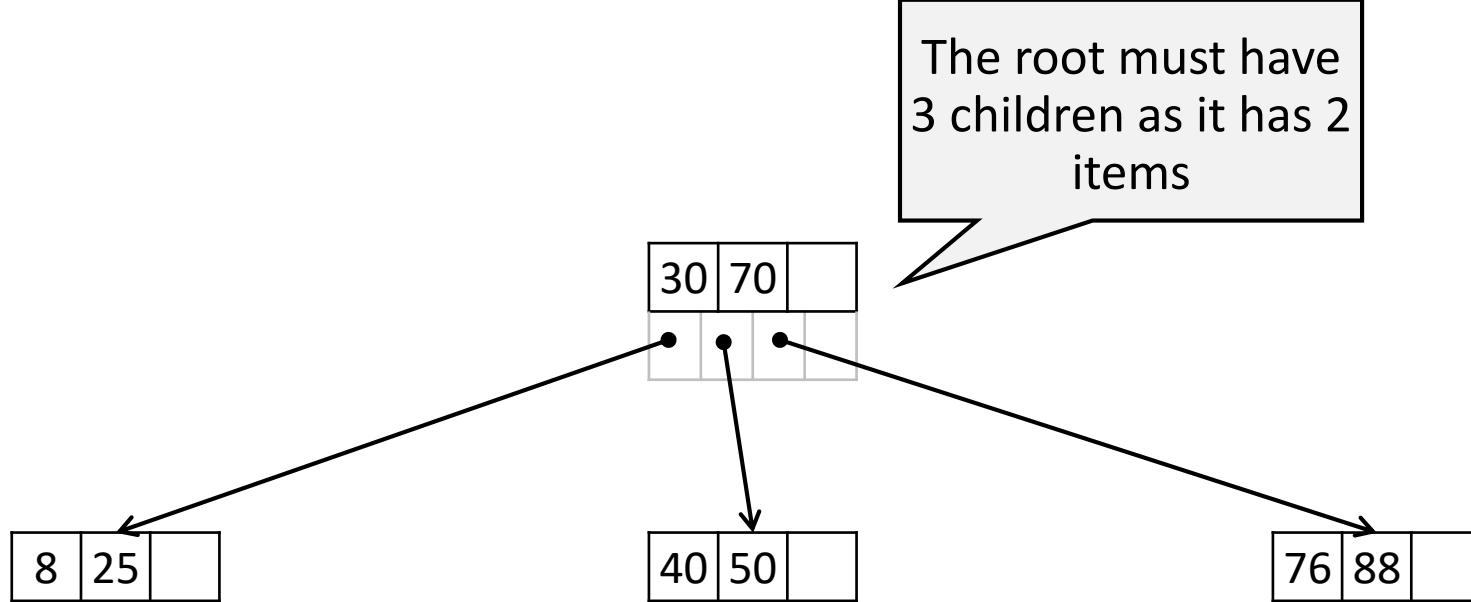


B-Tree based Directory

Suppose that items 30 and 70
are in the **root** of the tree



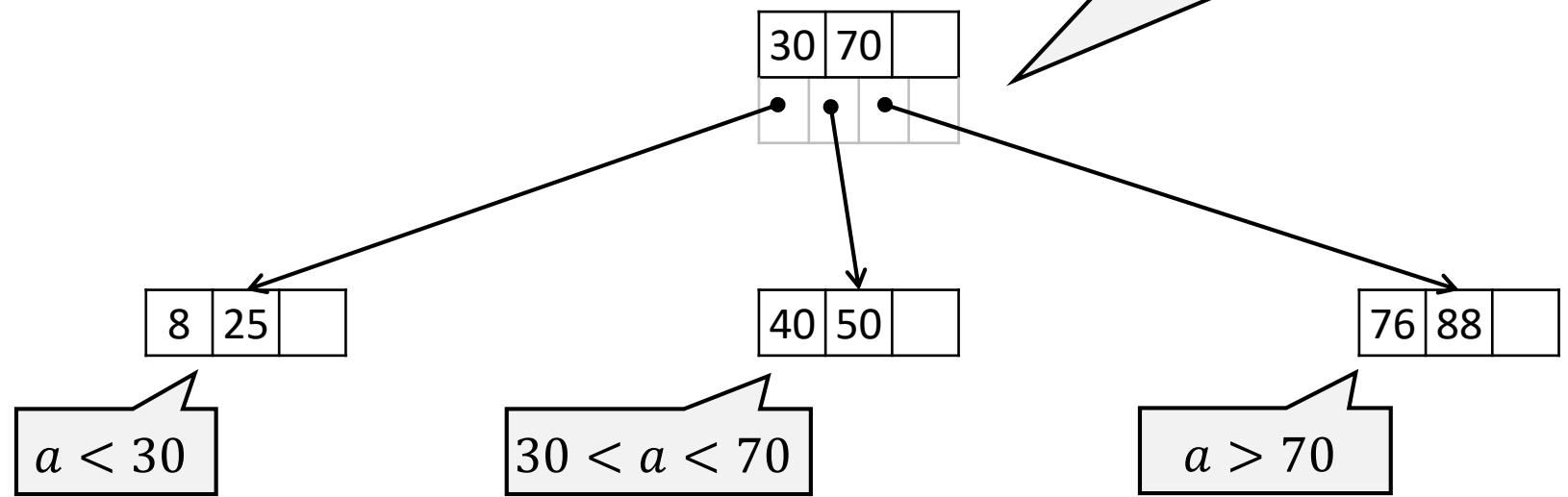
B-Tree based Directory



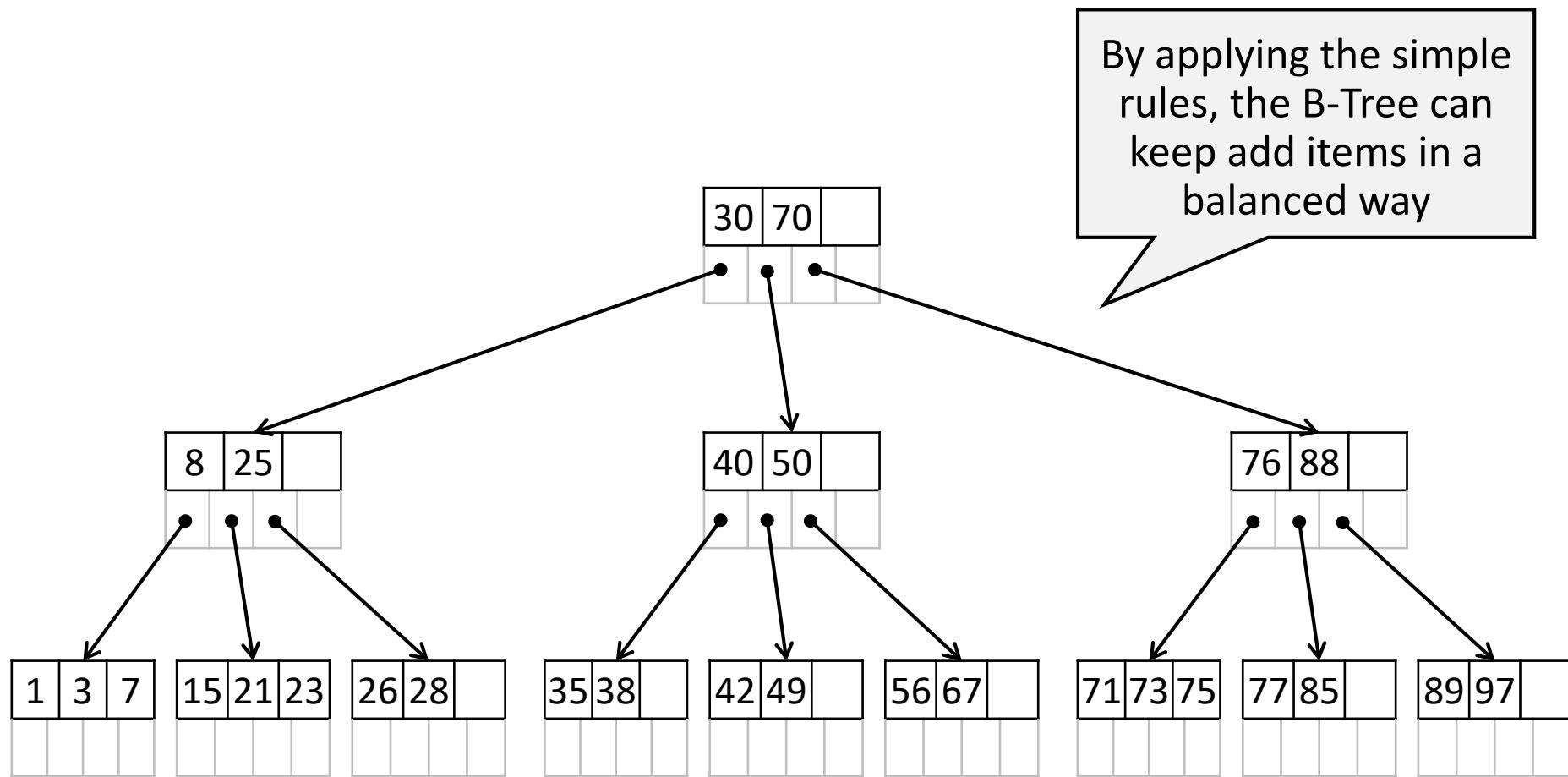
B-Tree based Directory

The three child groups are indicated by the items

- $a < 30$
- $30 < a < 70$
- $a > 70$



B-Tree based Directory



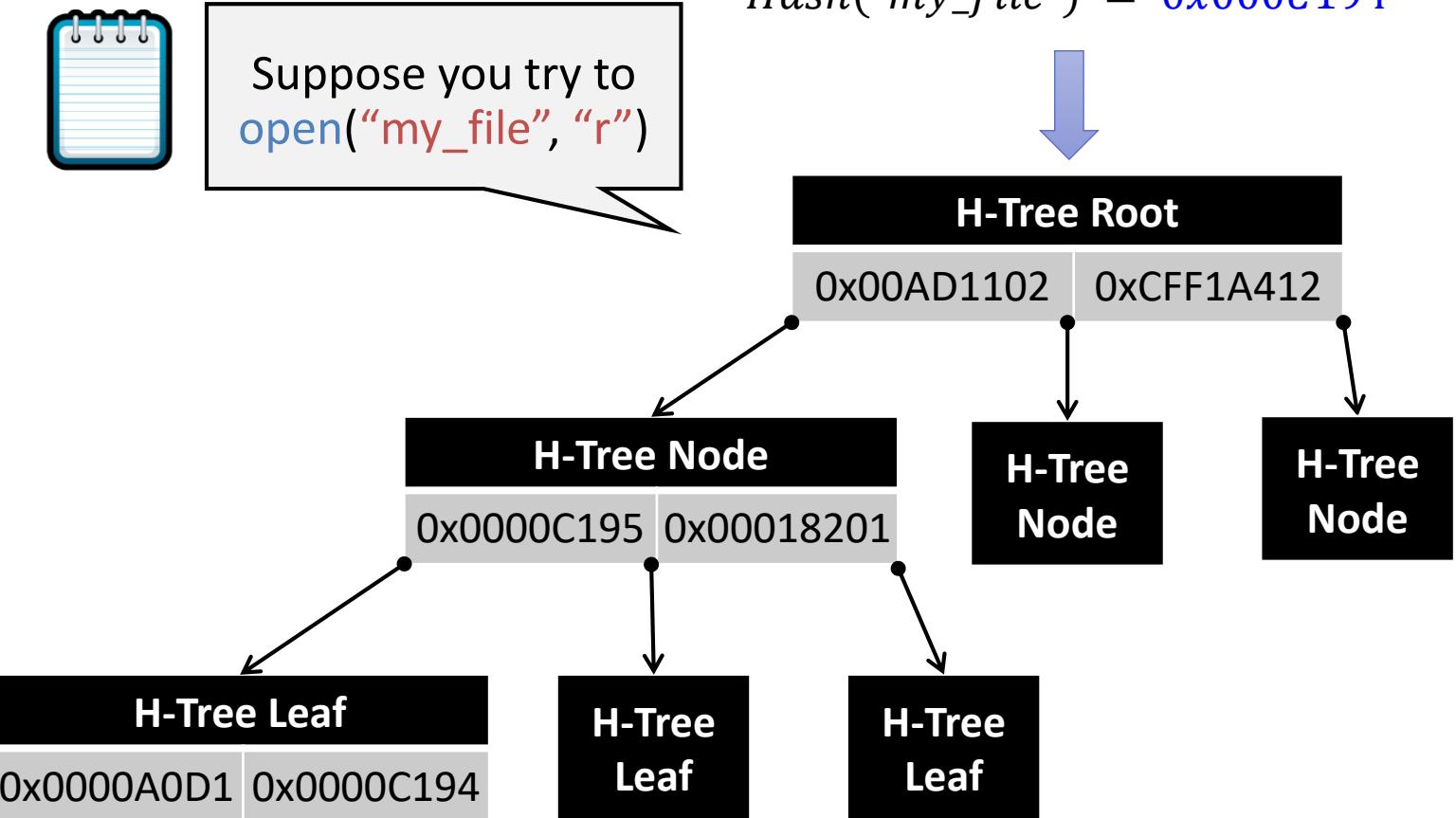
B-Tree based Directory

Ext4 employs a B-Tree variant,
H-Tree (similar to B+Tree).
The H stands for *hash*

Ext**4**

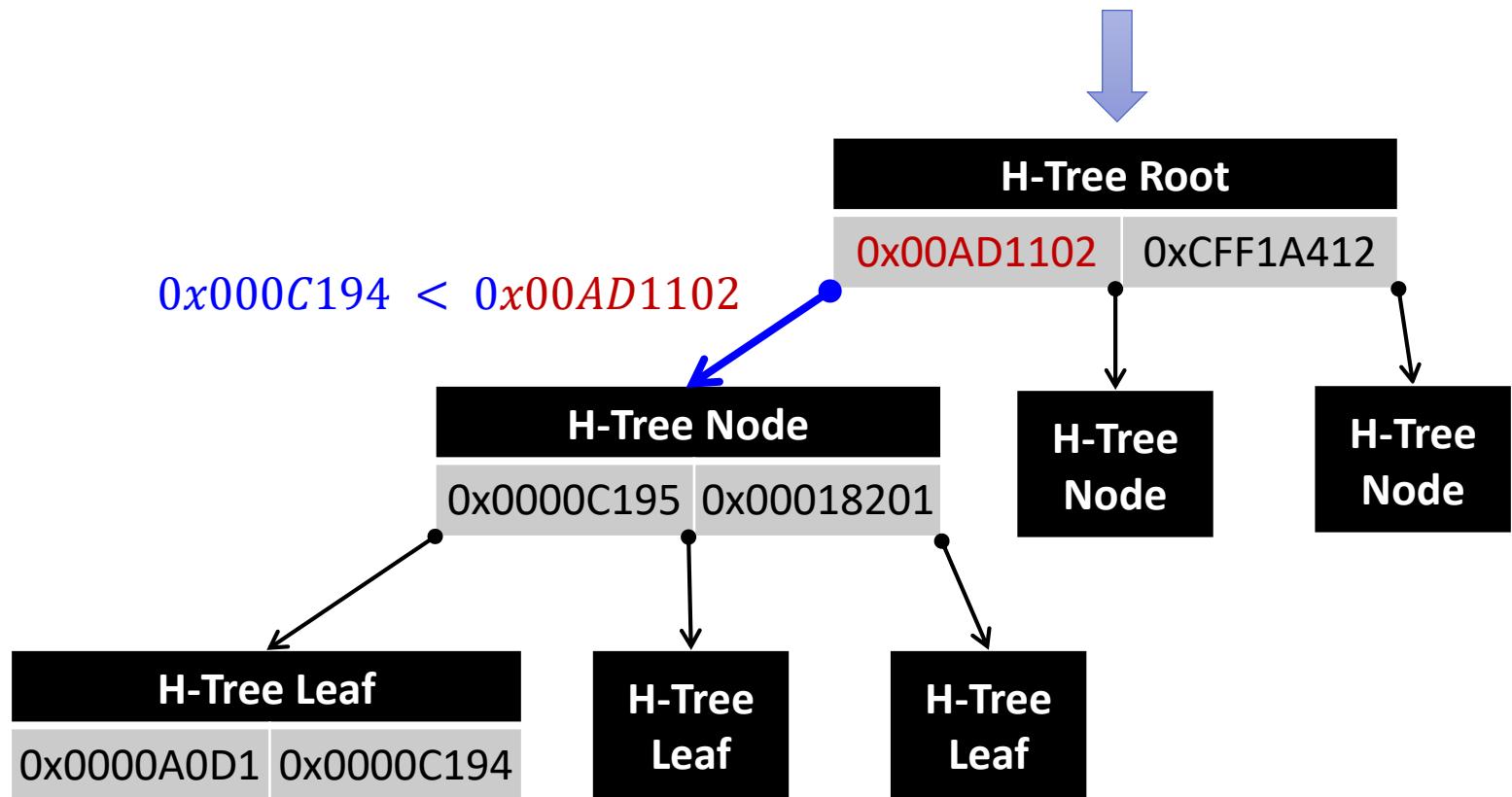


Example: Ext4



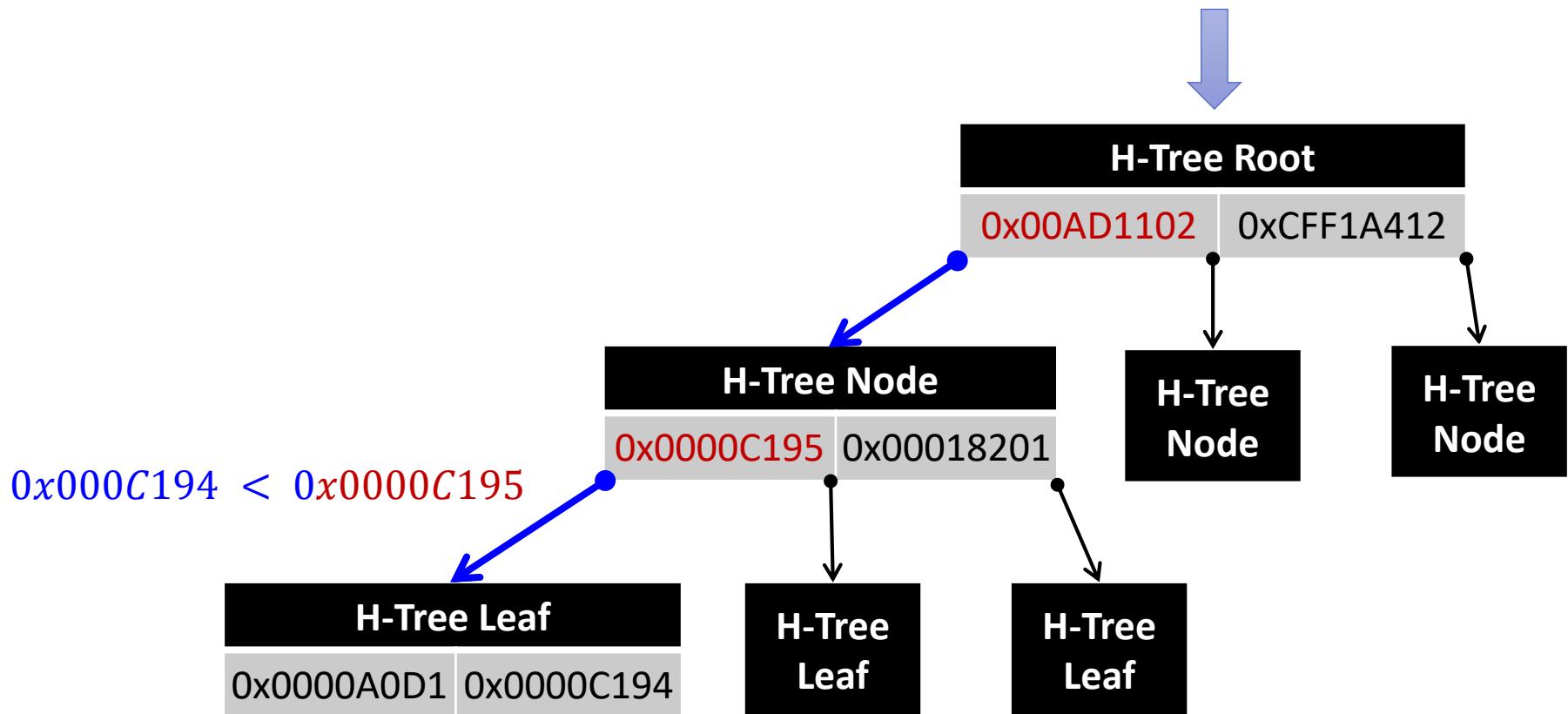
Example: Ext4

$\text{Hash}(\text{"my_file"}) = 0x000C194$



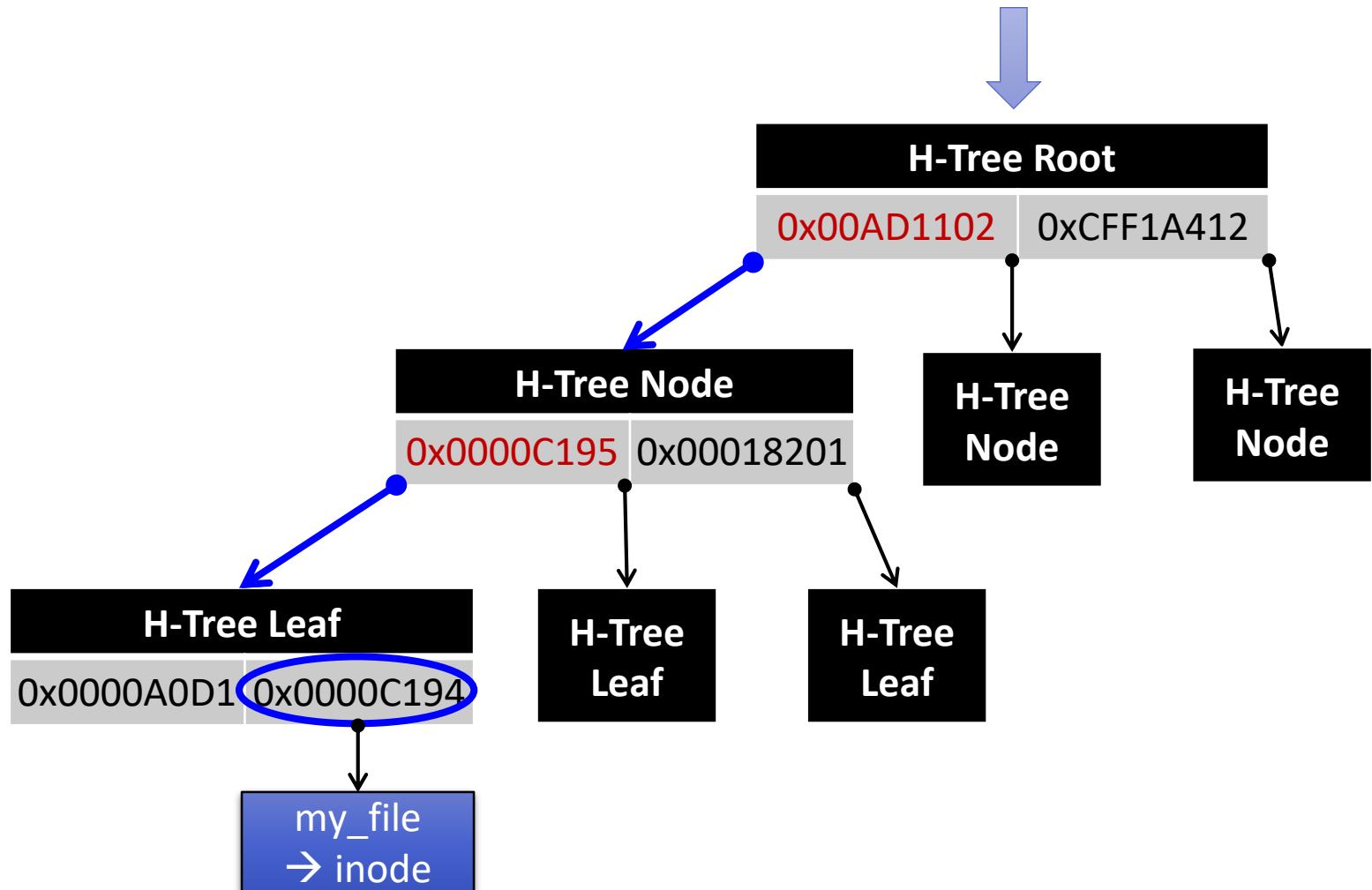
Example: Ext4

$\text{Hash}(\text{"my_file"}) = 0x000C194$



Example: Ext4

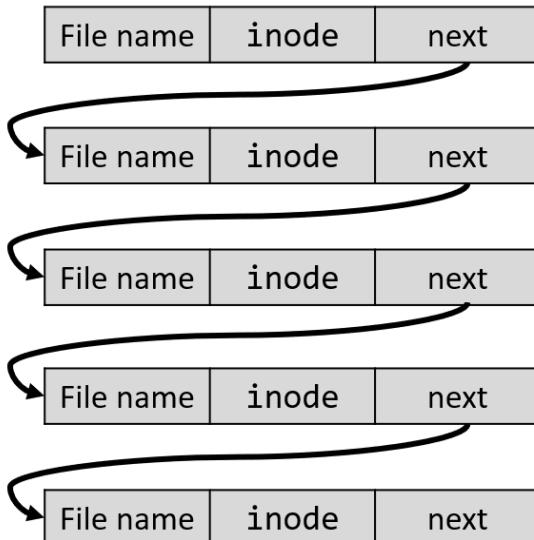
$\text{Hash}(\text{"my_file"}) = 0x000C194$



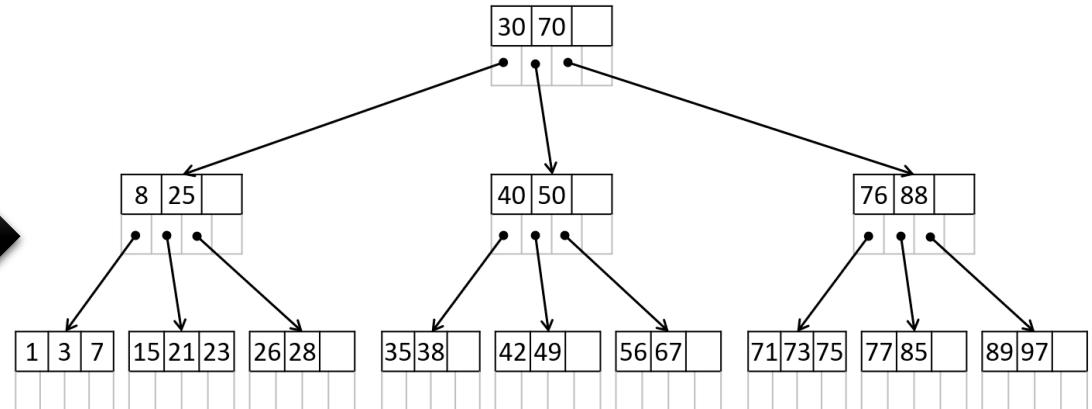


Linear List (Ext3) vs. B-tree (Ext4)

ext3 



Ext4 



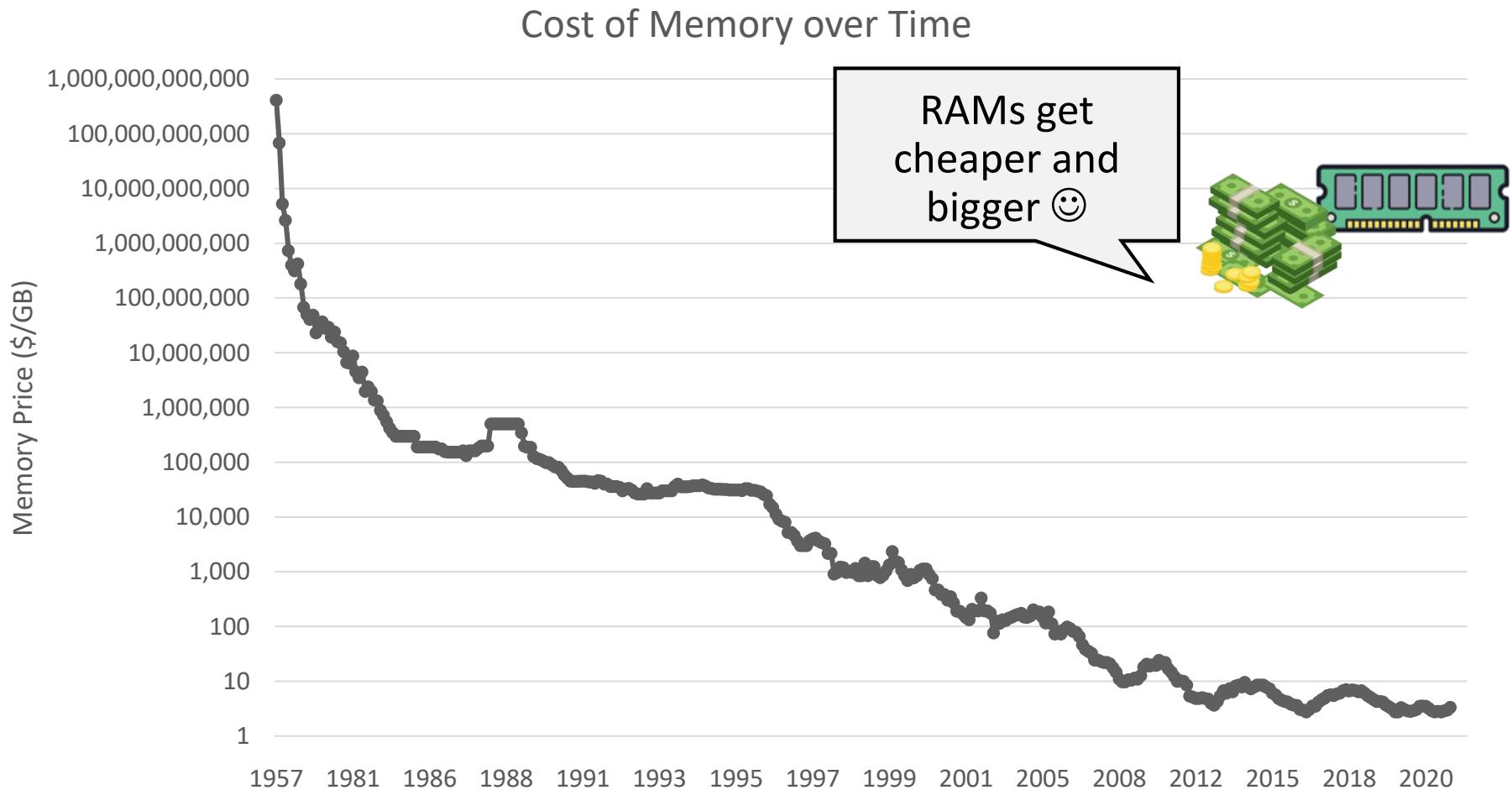
Ext4 shortens the algorithm complexity for the directory's inode lookup
(e.g., $O(n) \rightarrow O(\log n)$)

Reevaluating Disk Performance

We reviewed the most state-of-the art, and popular FS, Ext4. But let's go back to the drawing board and **reevaluate** from first-principals

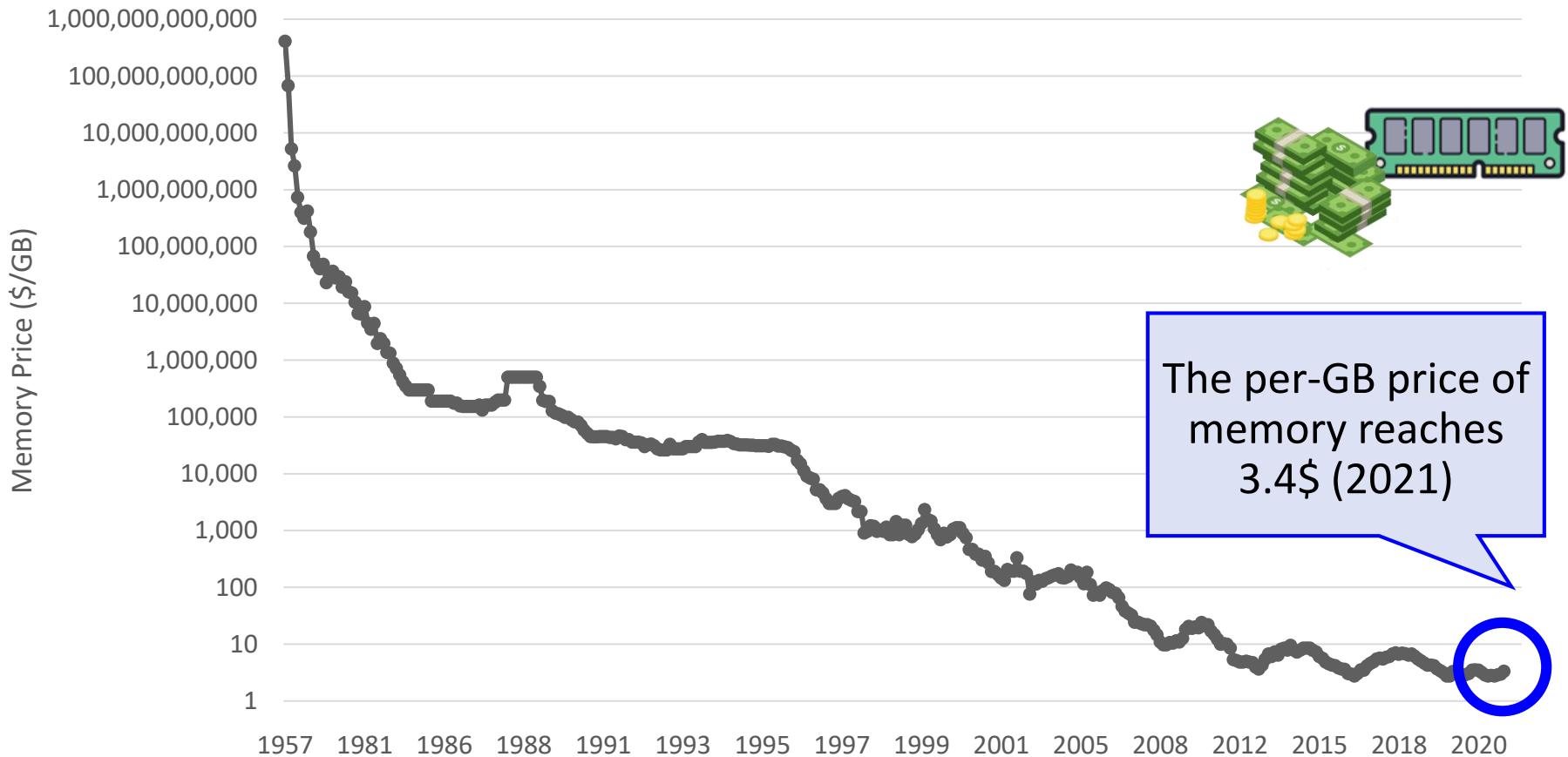


Evolution of Computer Hardware



Evolution of Computer Hardware

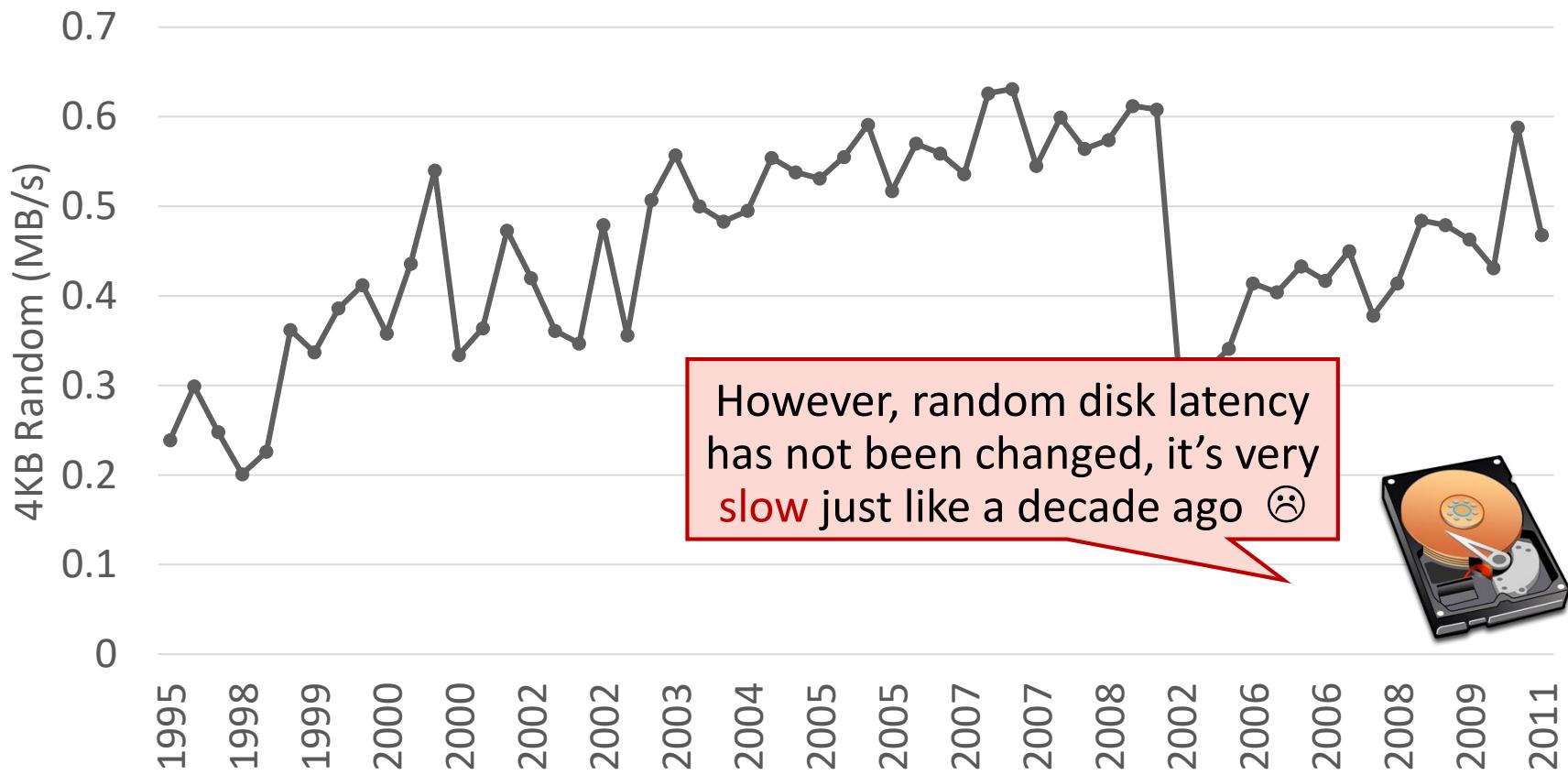
Cost of Memory over Time



The per-GB price of
memory reaches
3.4\$ (2021)

Evolution of Computer Hardware

Random Performance over Time



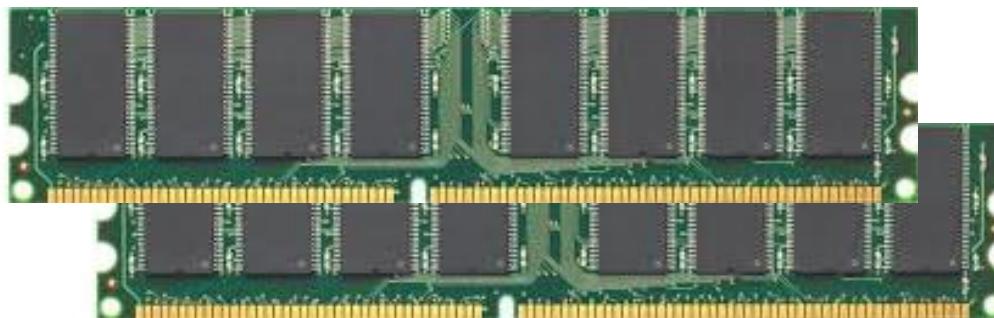
However, random disk latency
has not been changed, it's very
slow just like a decade ago 😞



Performance Angle

Data with sequential accesses can be cached in DRAM. Caching reads doesn't hurt durability ☺

But, all writes should be persistently stored back, and the random writes can move your storage performance back to a decade ago ☹



Log-Structured file system (LFS)

LFS is designed towards increasing the disk write performance, in particular, random writes

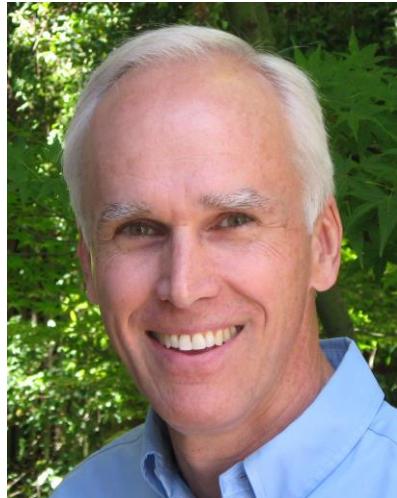


Log-Structured file system

The design of LFS was first proposed in 1988 by John K. Ousterhout and Fred Douglis



What is the main features of LFS?





#1: Treating the Disk as a Log

Feature #1: Disk is considered as a circular buffer (called a **log**)

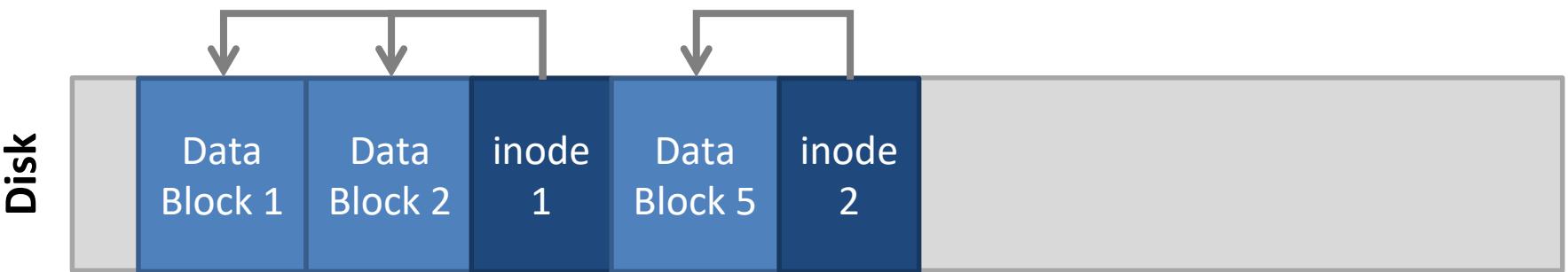
Disk

Giant Log



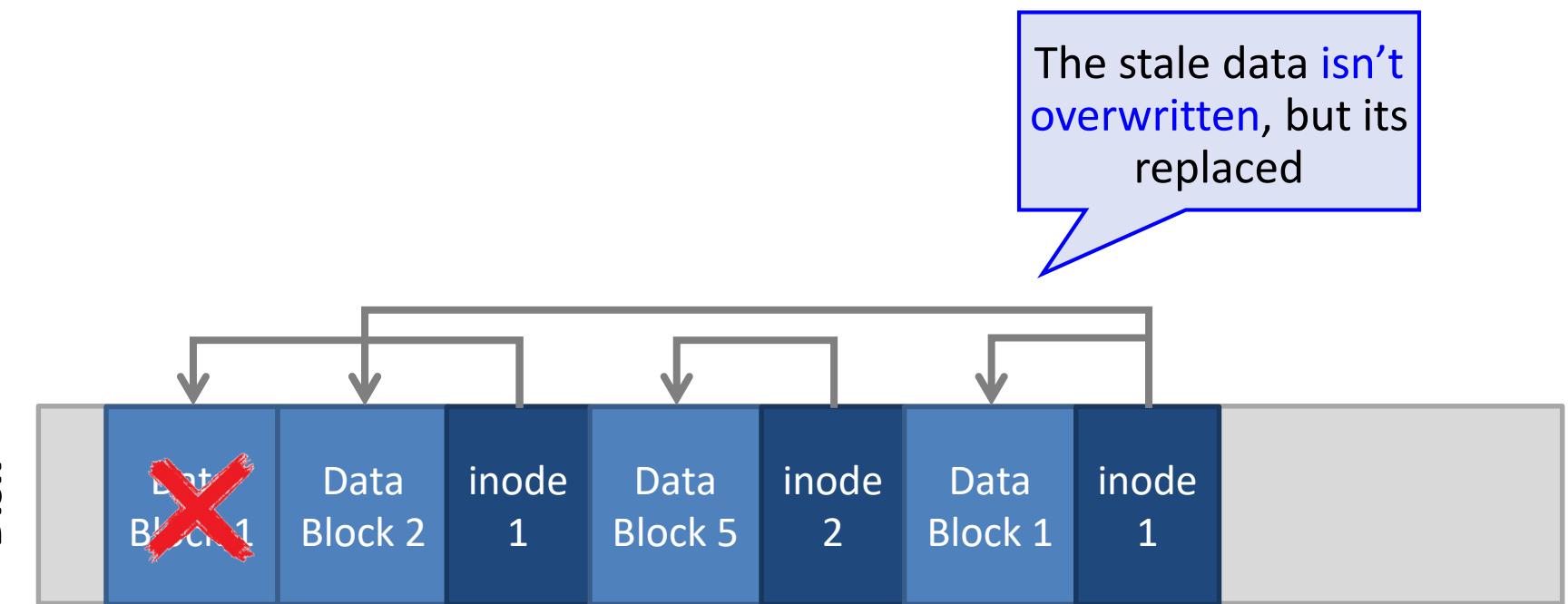
#1: Treating the Disk as a Log

The main concept of LFS is the same as data journaling. Data and metadata get appended to a log



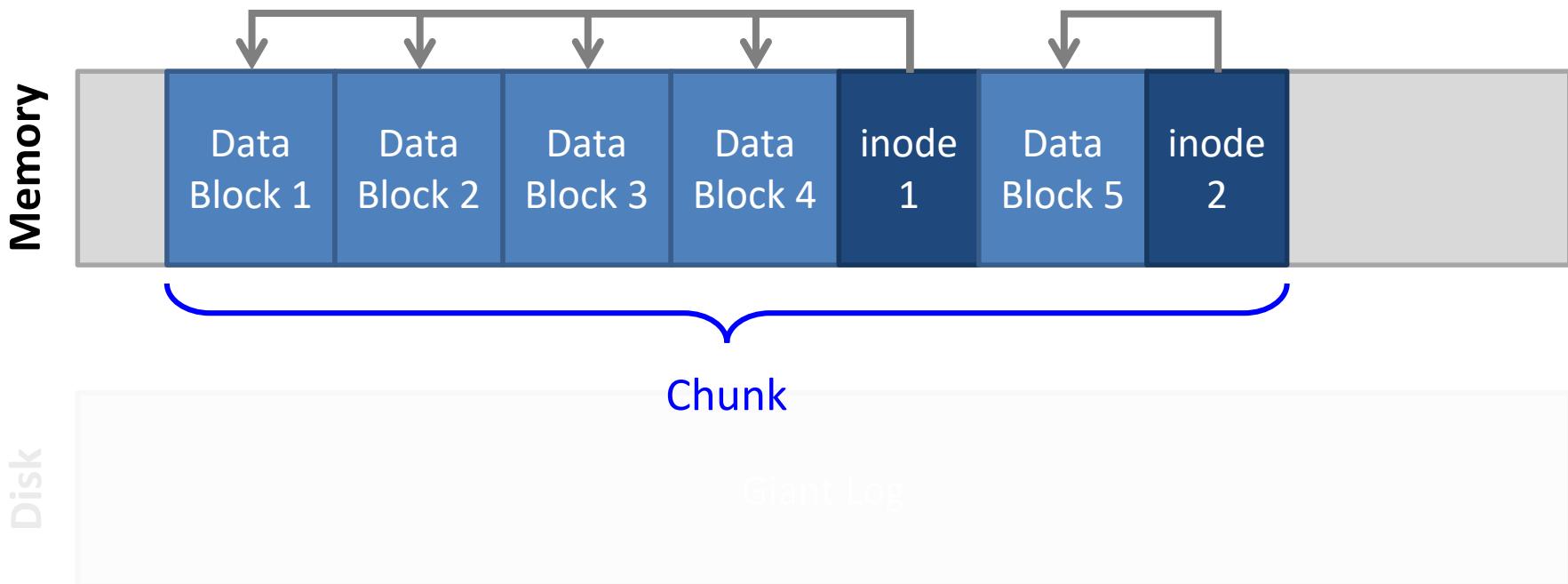


#1: Treating the Disk as a Log



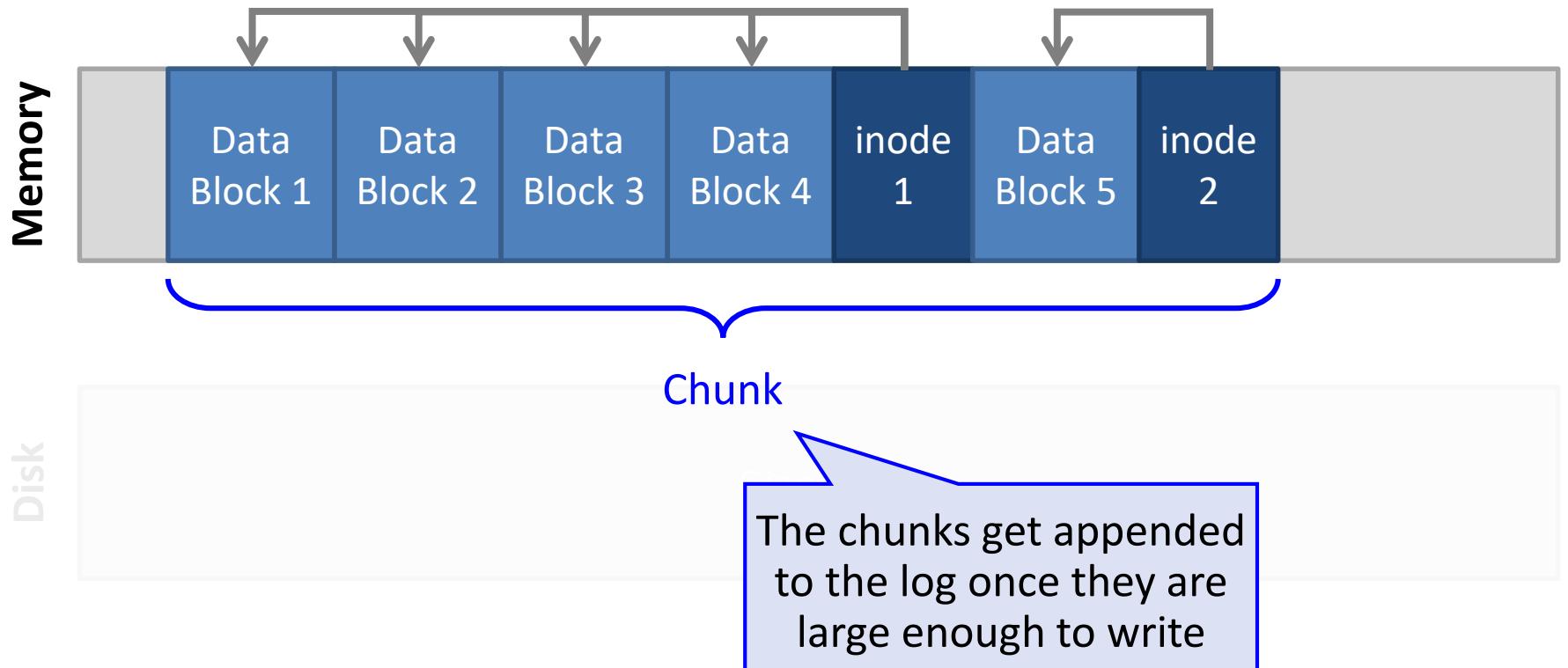
#2: Buffering Writes in Memory

Feature #2: LFS buffers all writes (including meta-data) in-memory into chunks





#2: Buffering Writes in Memory



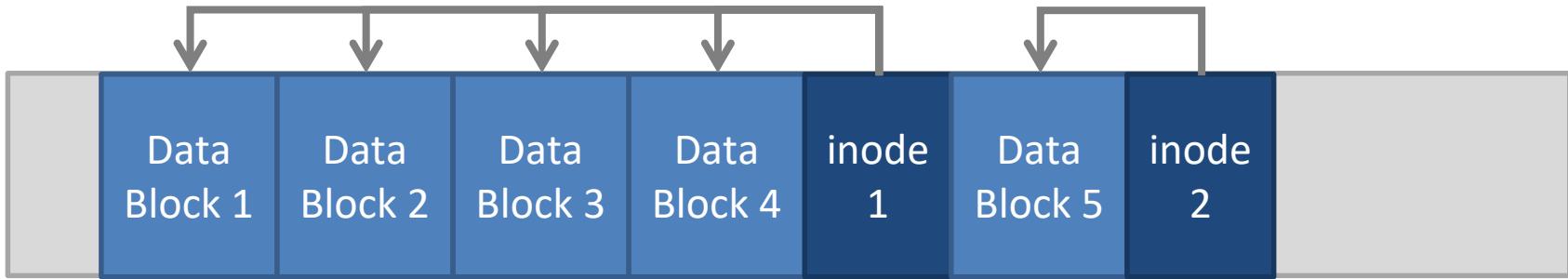


#2: Buffering Writes in Memory

Memory

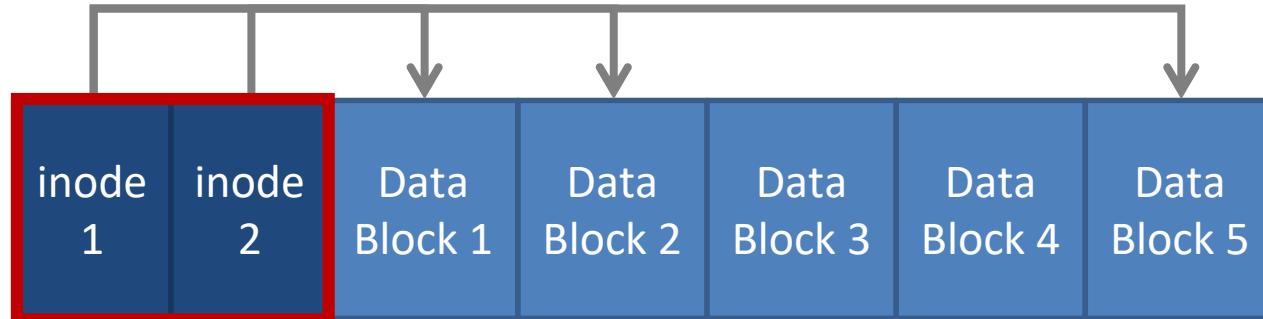


Disk



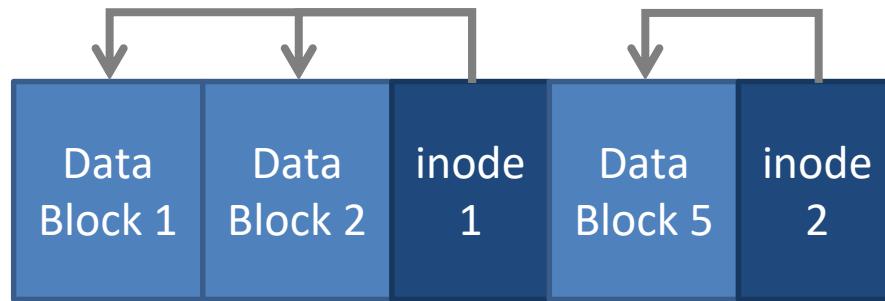
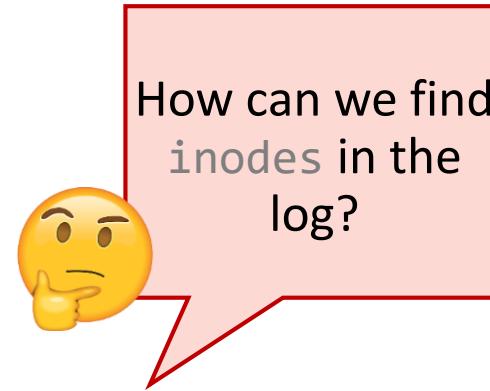
How to Find inodes

In a typical file system,
the **inodes** are stored at
fixed locations (relatively
easy to find)



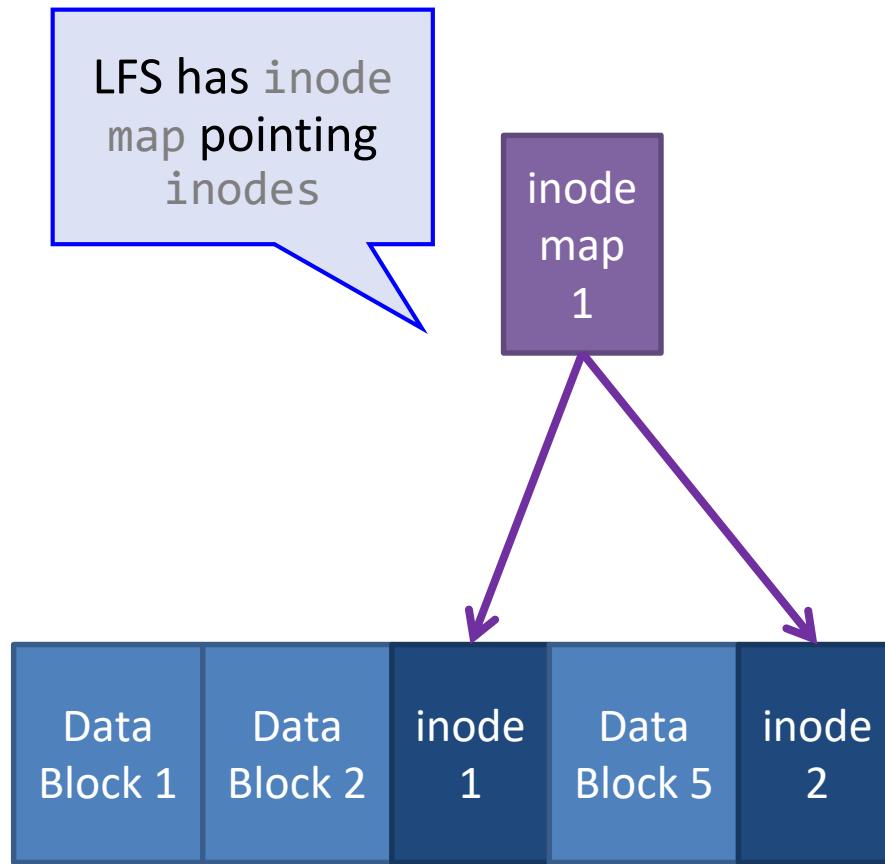
Typical file system (e.g., Ext4)

How to Find inodes



Log-Structured file system

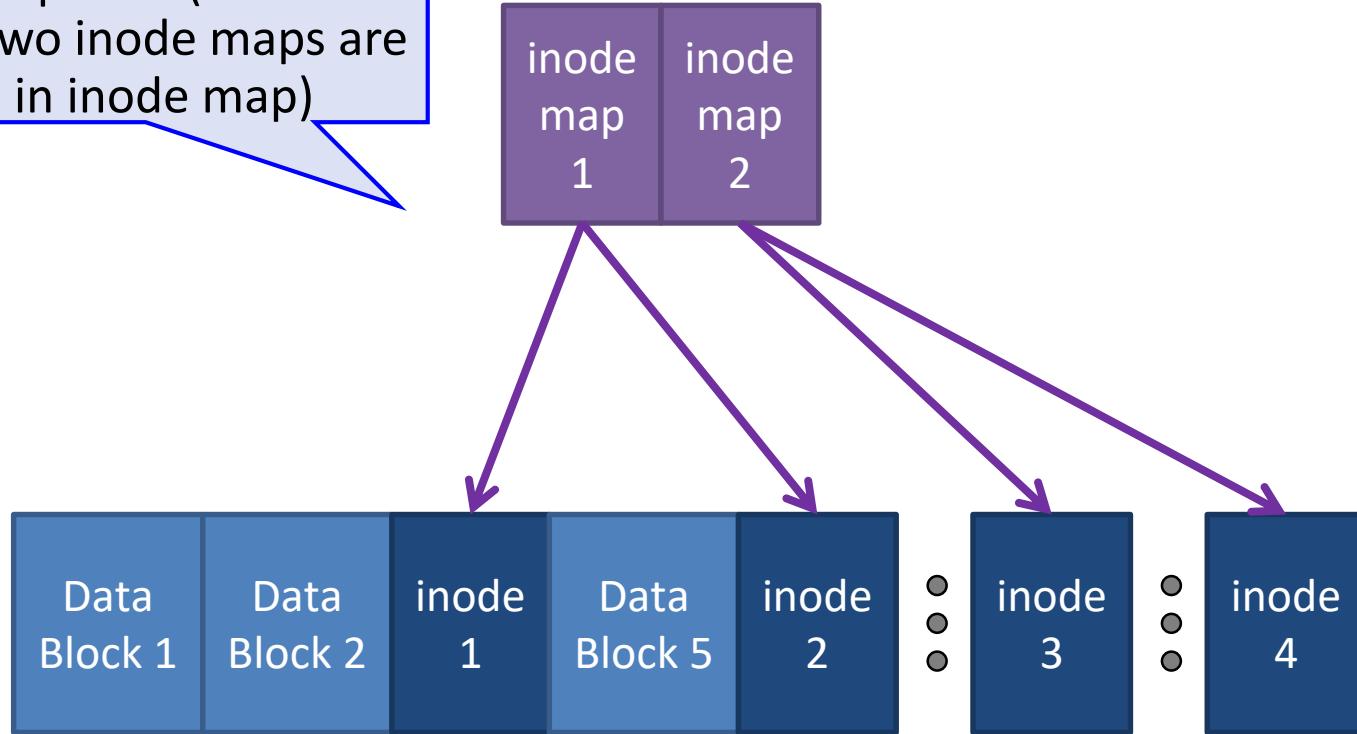
Solution: Add a Level of Indirection





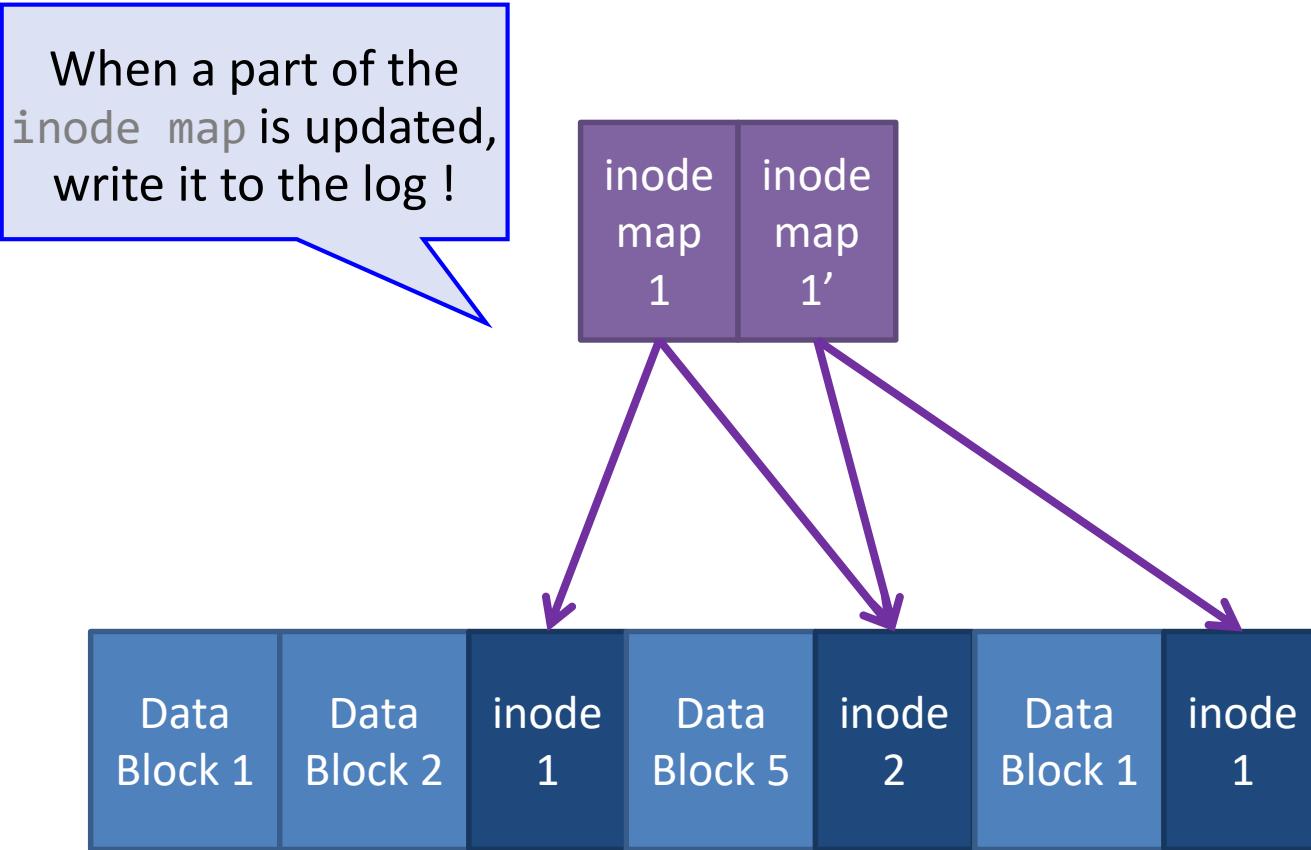
Solution: Add a Level of Indirection

inode map can be broken into multiple pieces (in this example, two inode maps are stored in inode map)

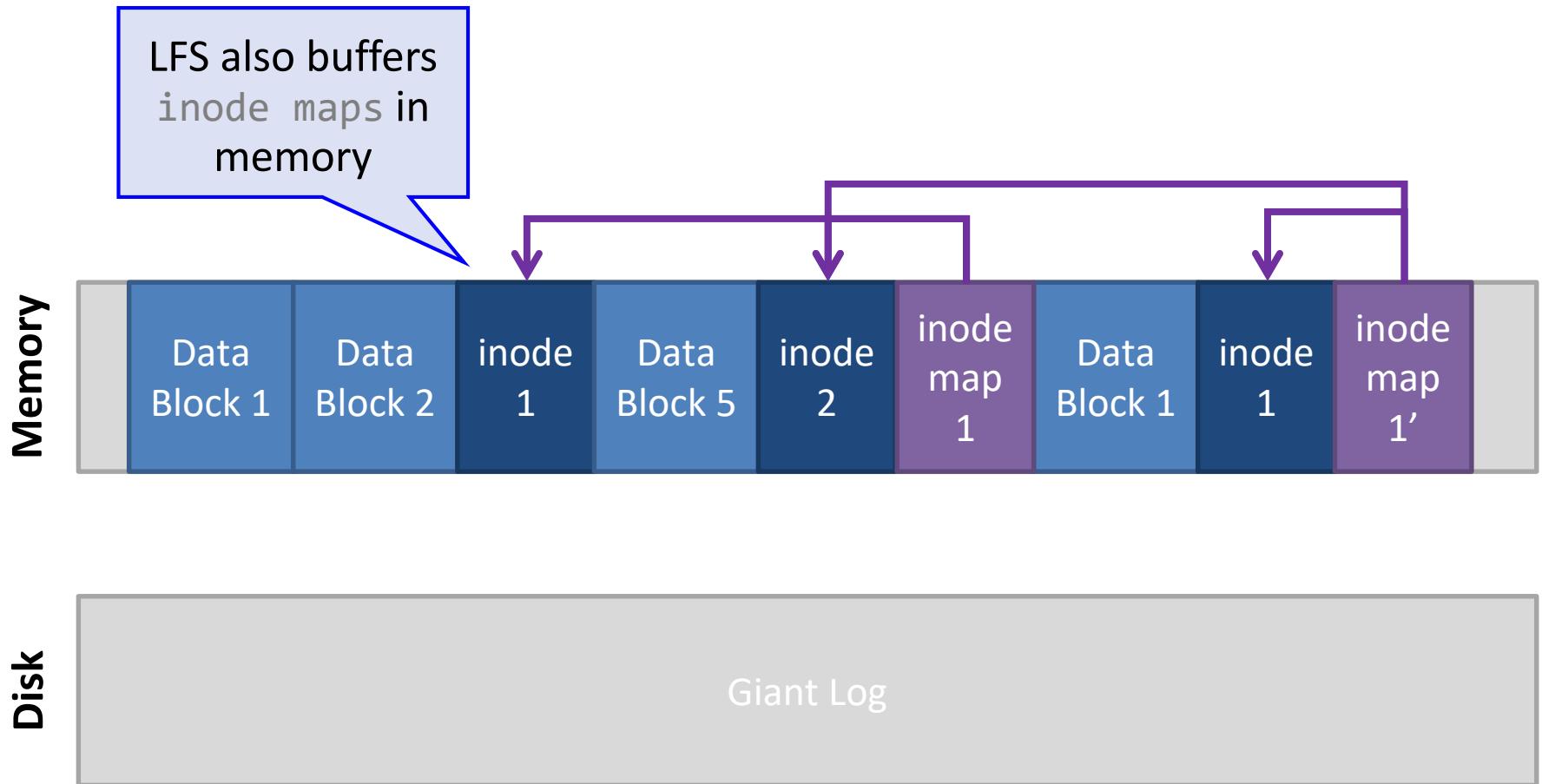




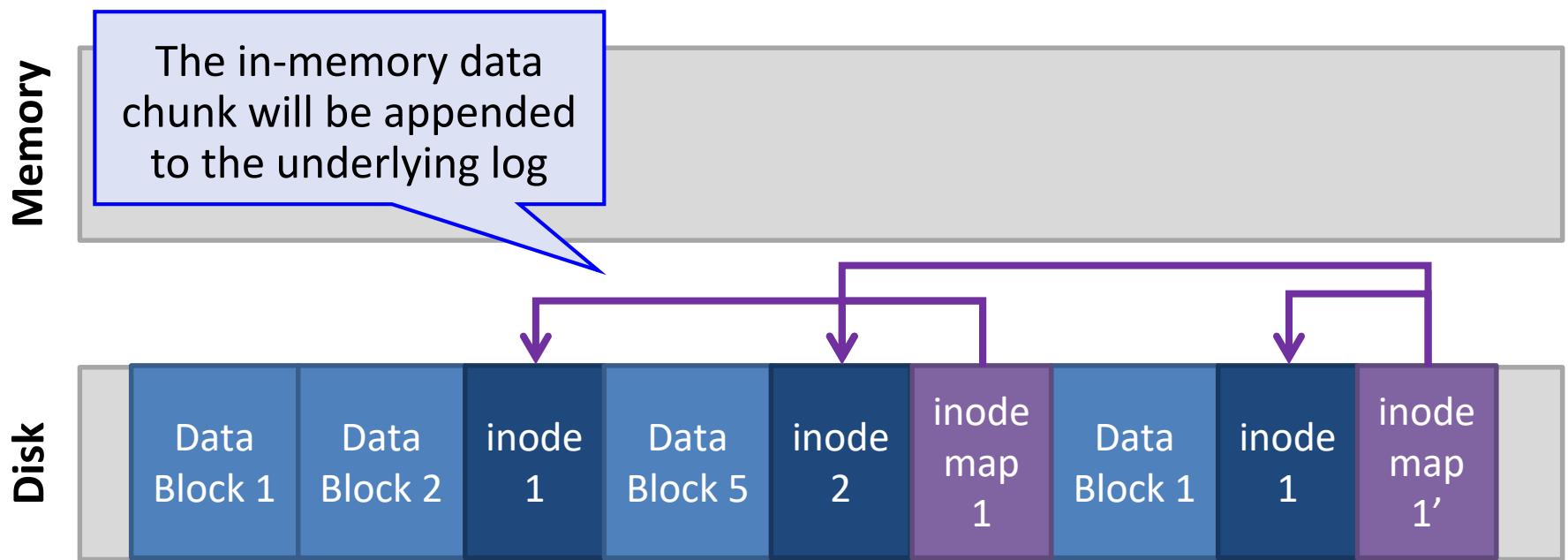
Solution: Add a Level of Indirection



Inode maps



Inode maps



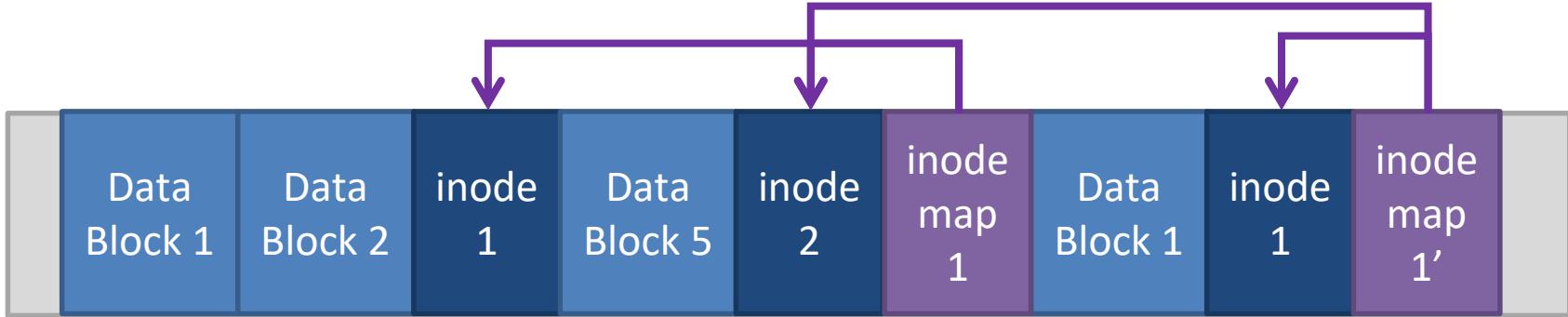
Inode maps

Memory



We didn't yet answer the question. How can we find (the most up-to-date) inode maps in the log then?

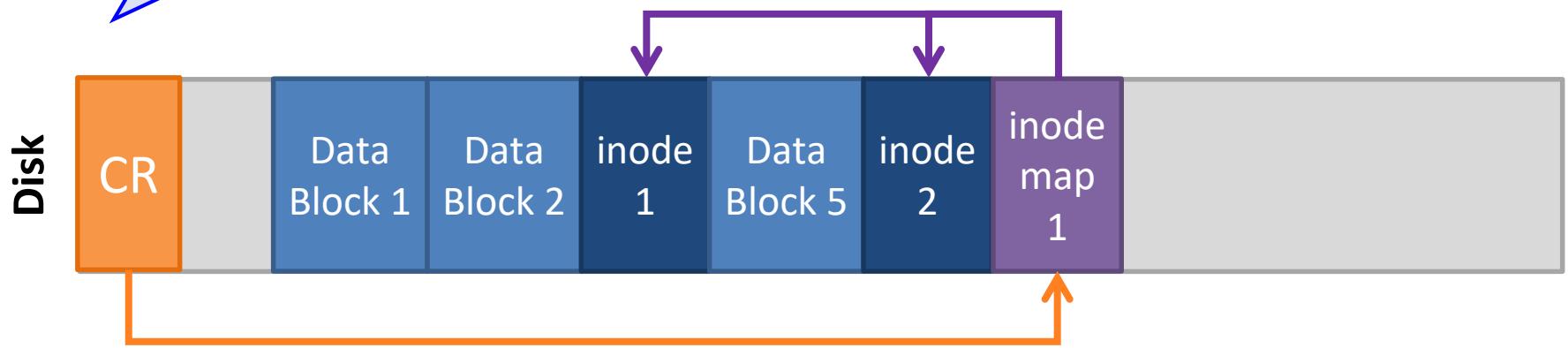
Disk





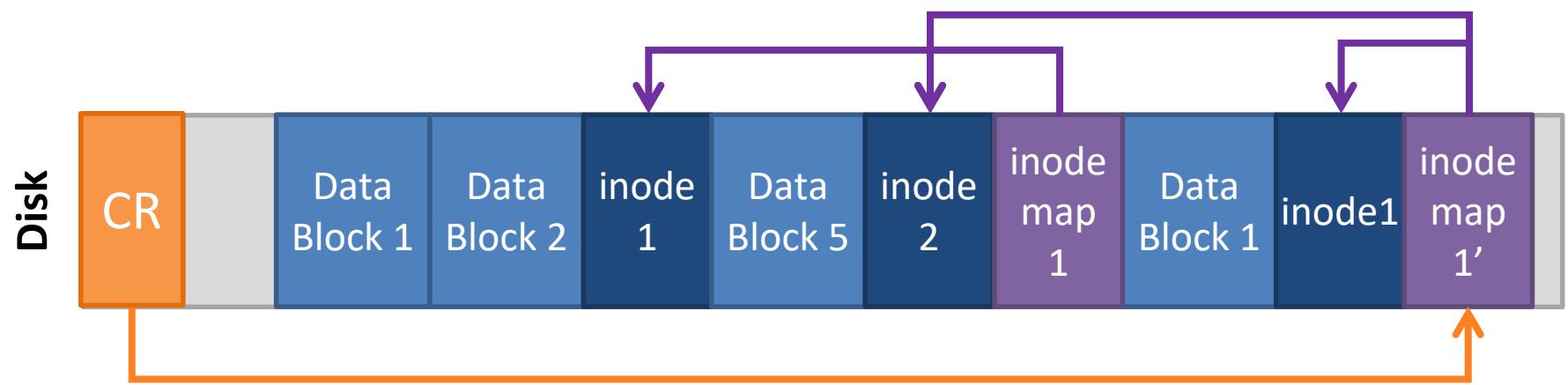
Solution: The Checkpoint Region

The superblock in LFS contains pointers to all of the up-to-date inode maps

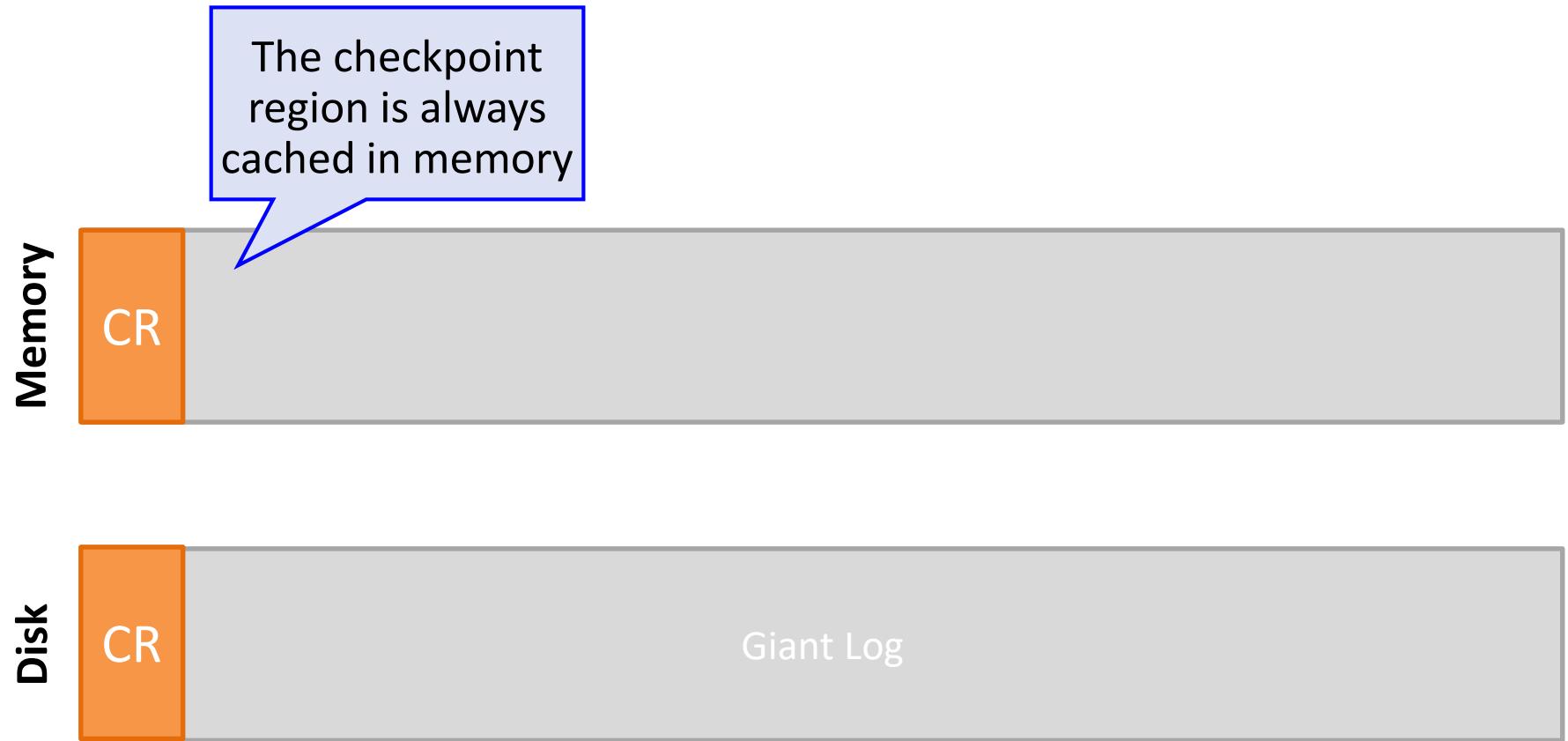




Solution: The Checkpoint Region

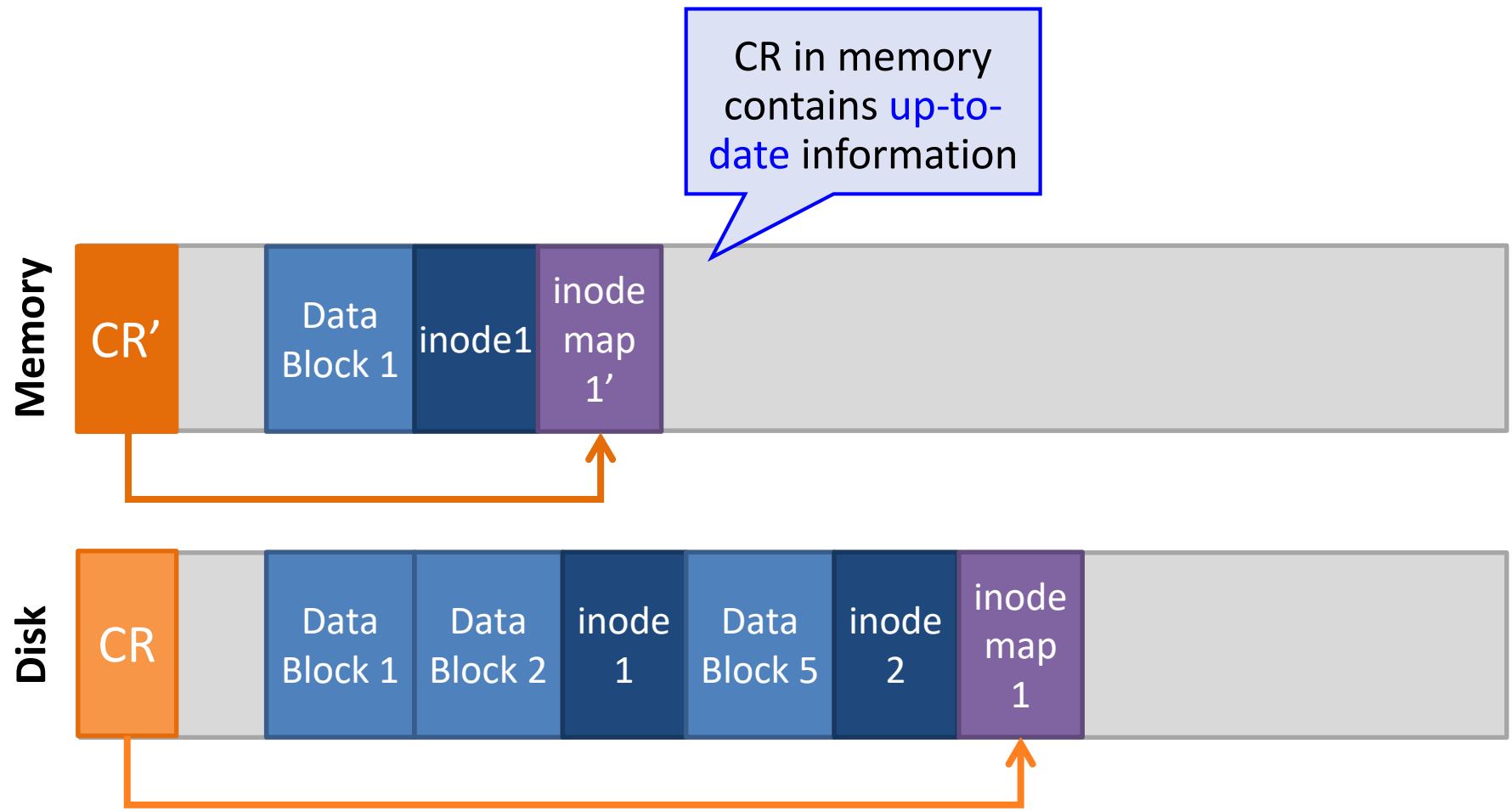


Solution: The Checkpoint Region



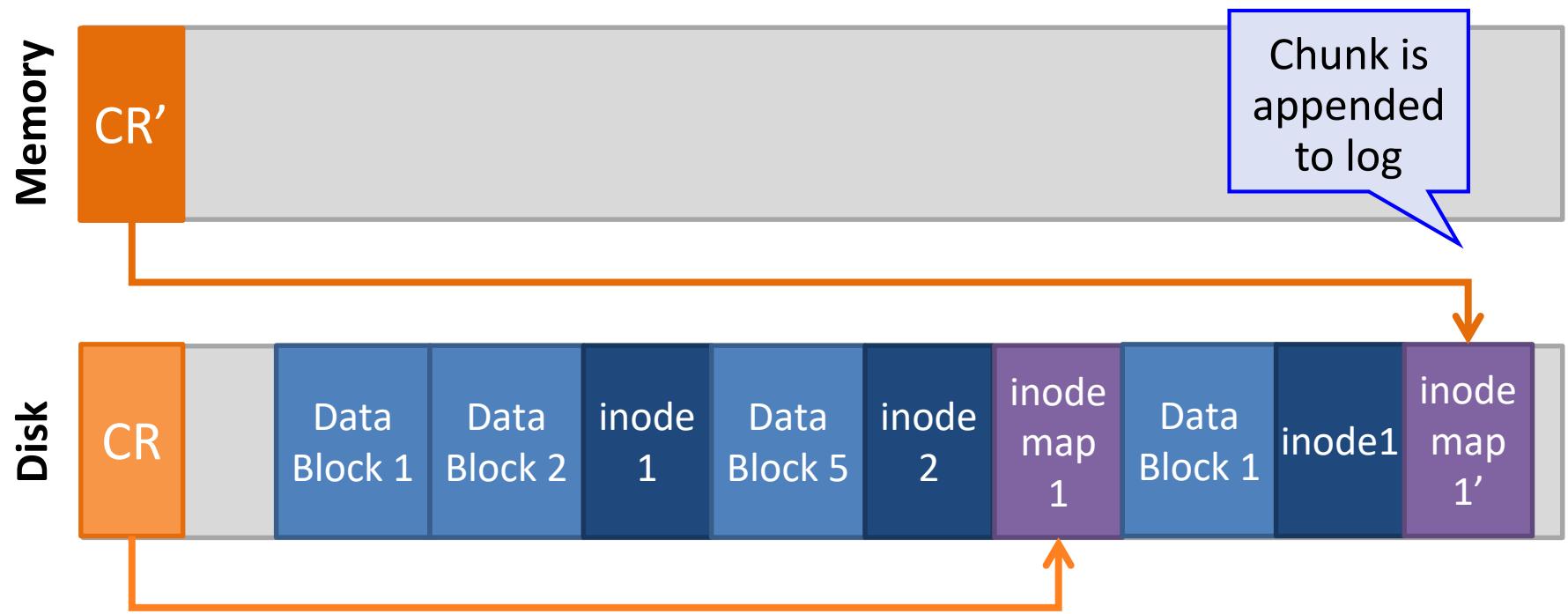


Solution: The Checkpoint Region



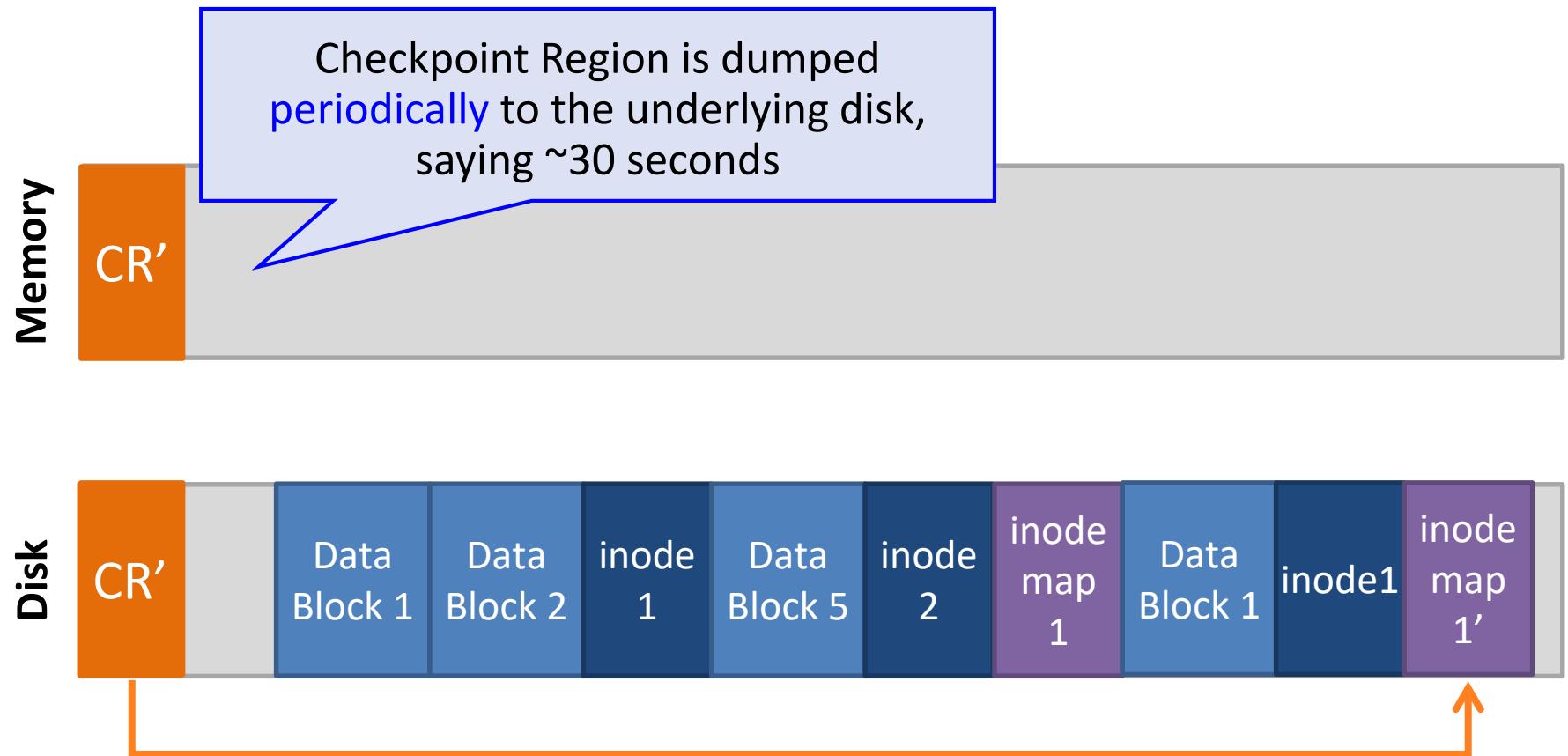


Solution: The Checkpoint Region





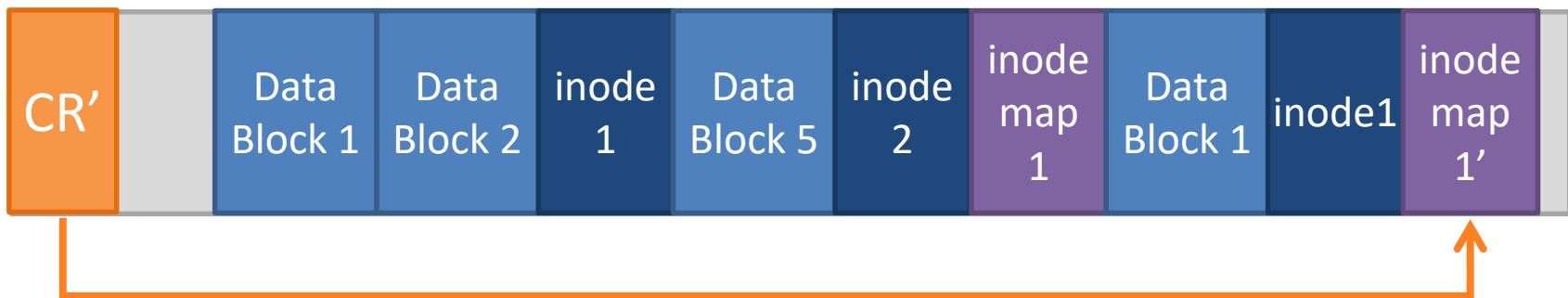
Solution: The Checkpoint Region



How to Read a File in LFS

Suppose that
you want to
read inode 1

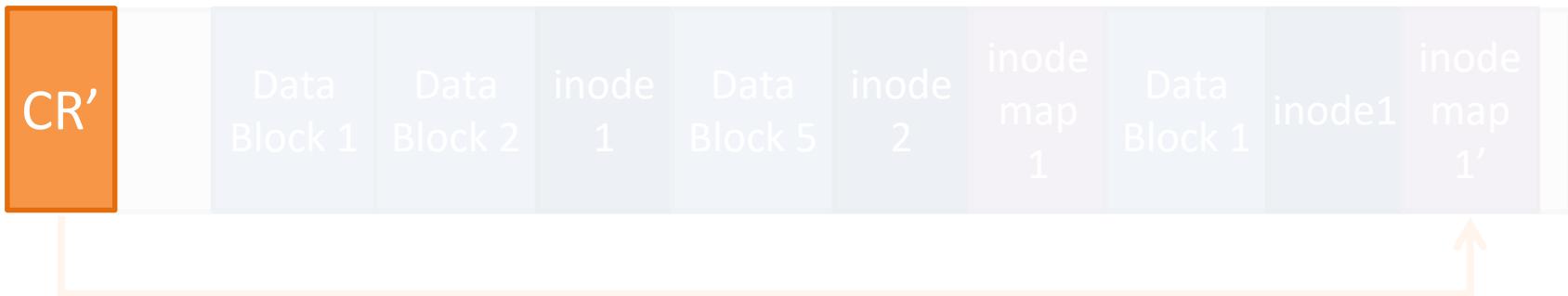
Disk



Step #1: Lookup Checkpoint Region

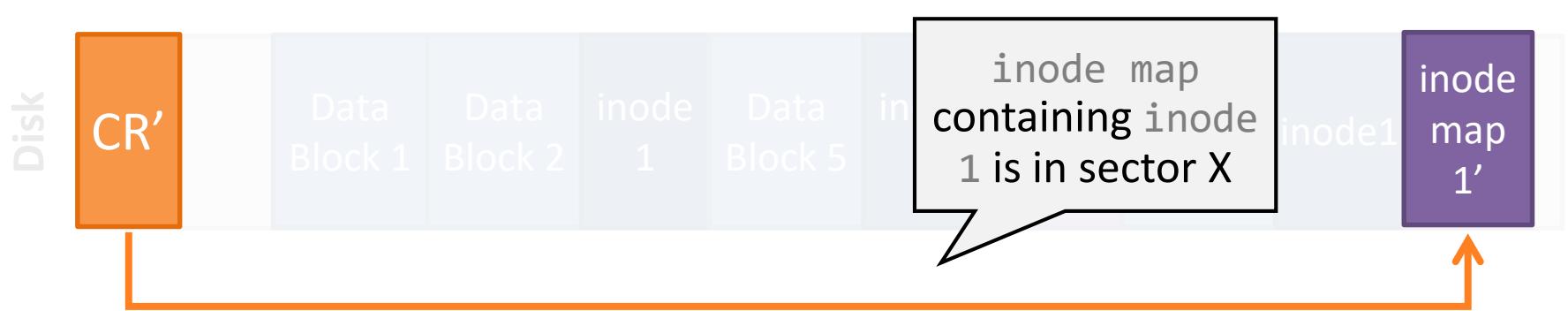
- ① Lookup inode 1 in the checkpoint region

Disk





Step #1: Lookup Checkpoint Region





Step #2: Read inode map

② Read the inode map at *sector X*

Disk

CR'

Data Block 1

Data Block 2

inode 1

Data Block 5

inode 2

inode map 1

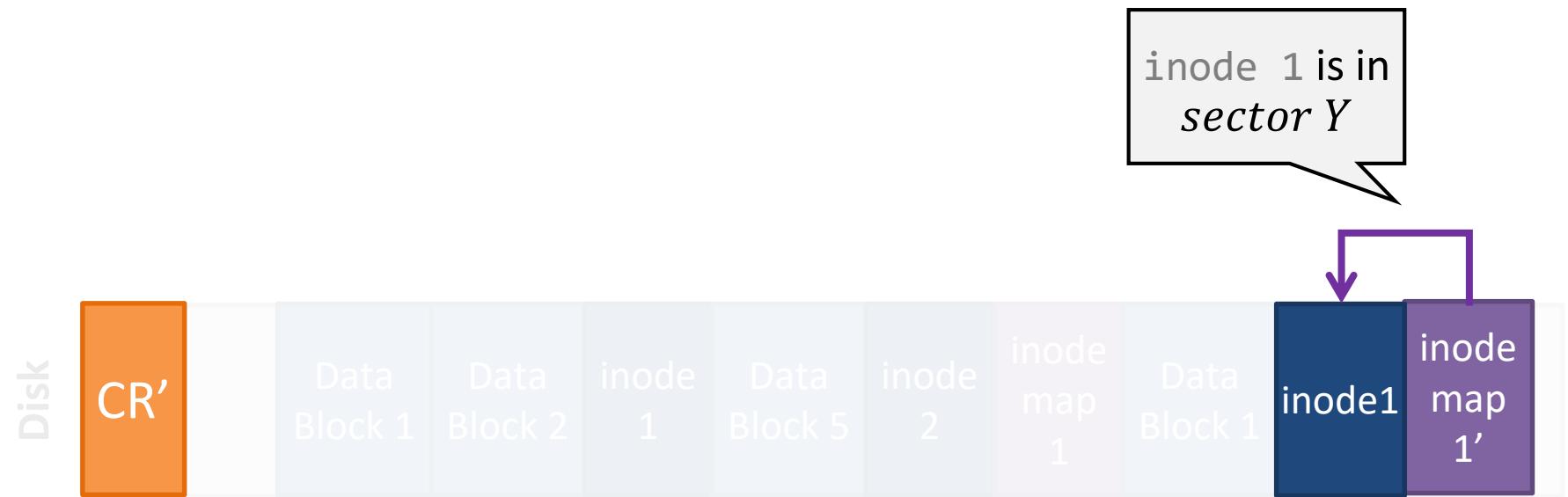
Data Block 1

inode1

inode map 1'

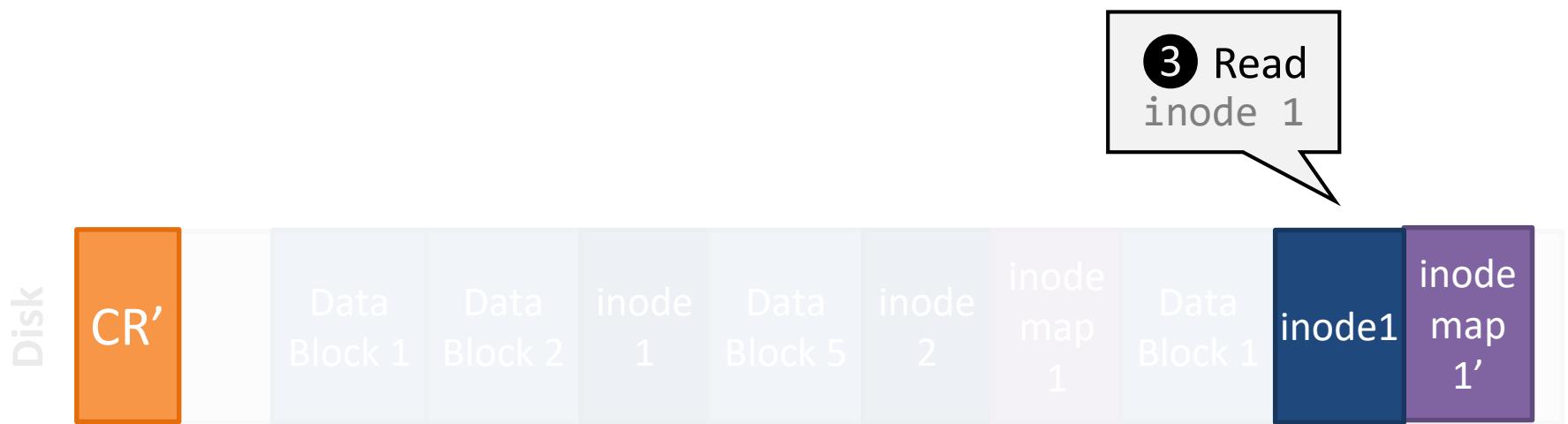


Step #2: Read inode map

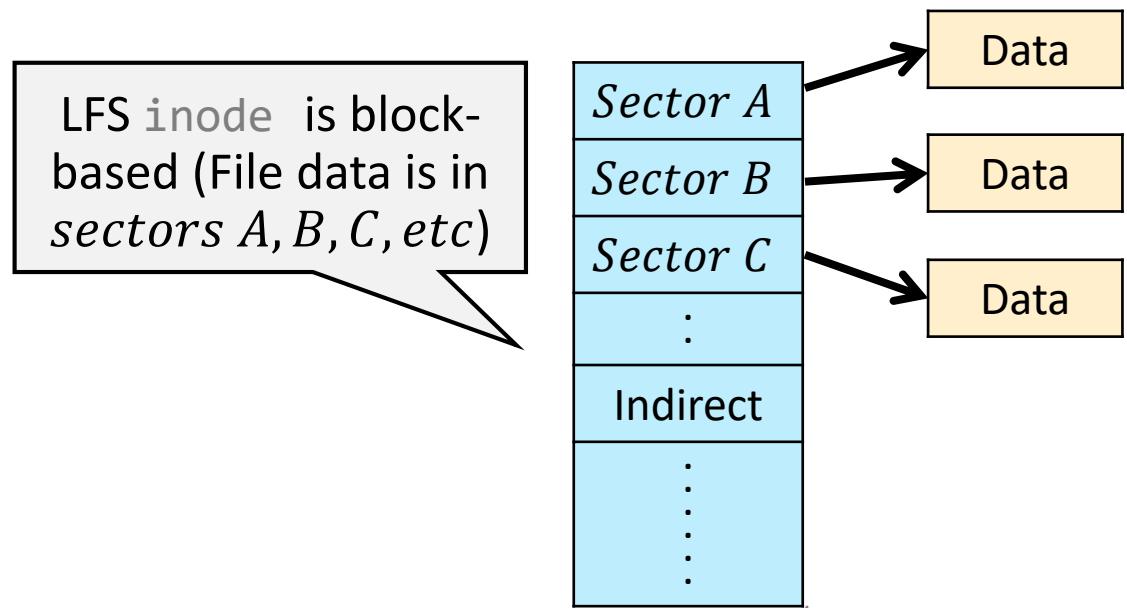




Step #3: Read inode

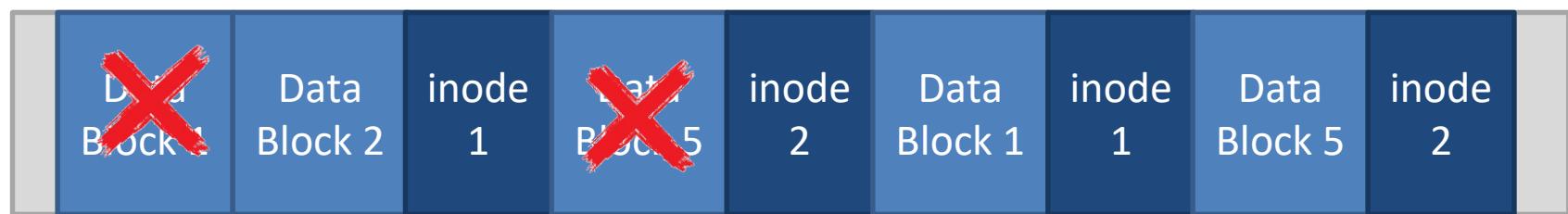


Step #3: Read inode



What about Overwritten Data?

As the writes are processed, the log is going to be filled up with stale data



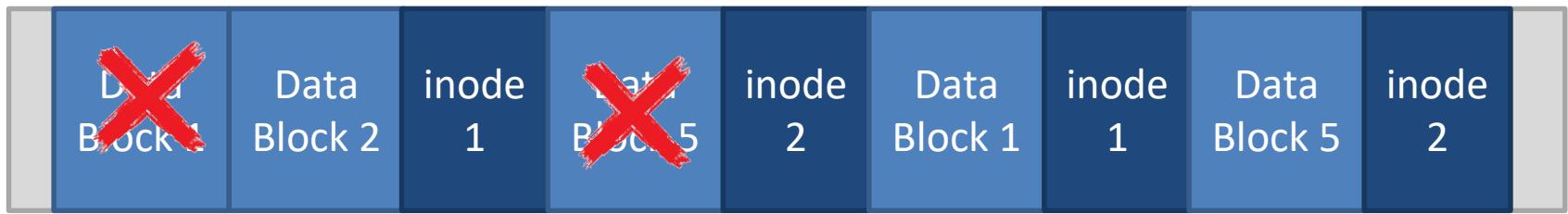


Problem: Highly Fragmented

Your data are all
intermixed with
stale data



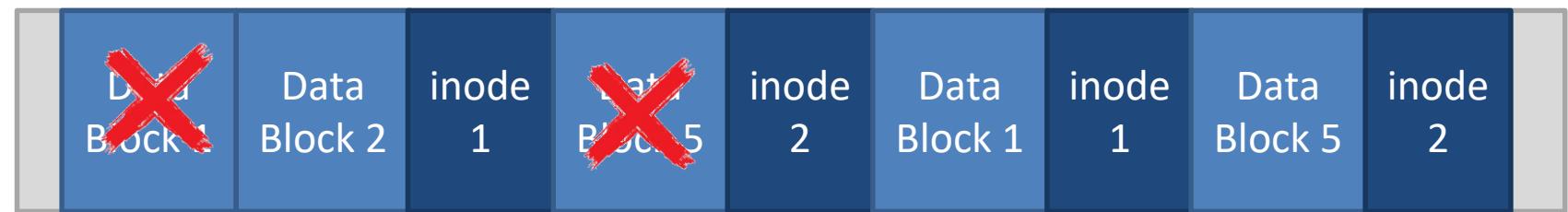
Disk





Solution: Garbage Collection

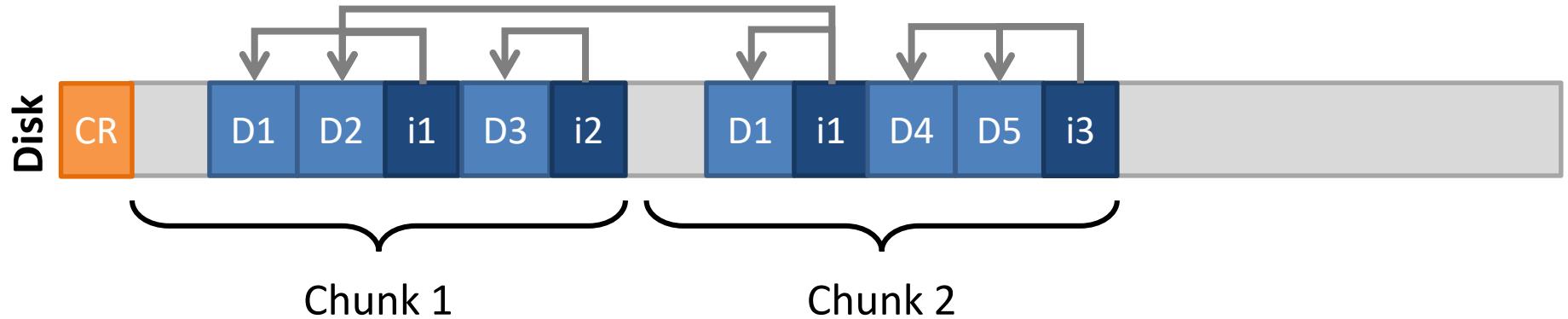
LFS periodically collects garbage blocks from the log





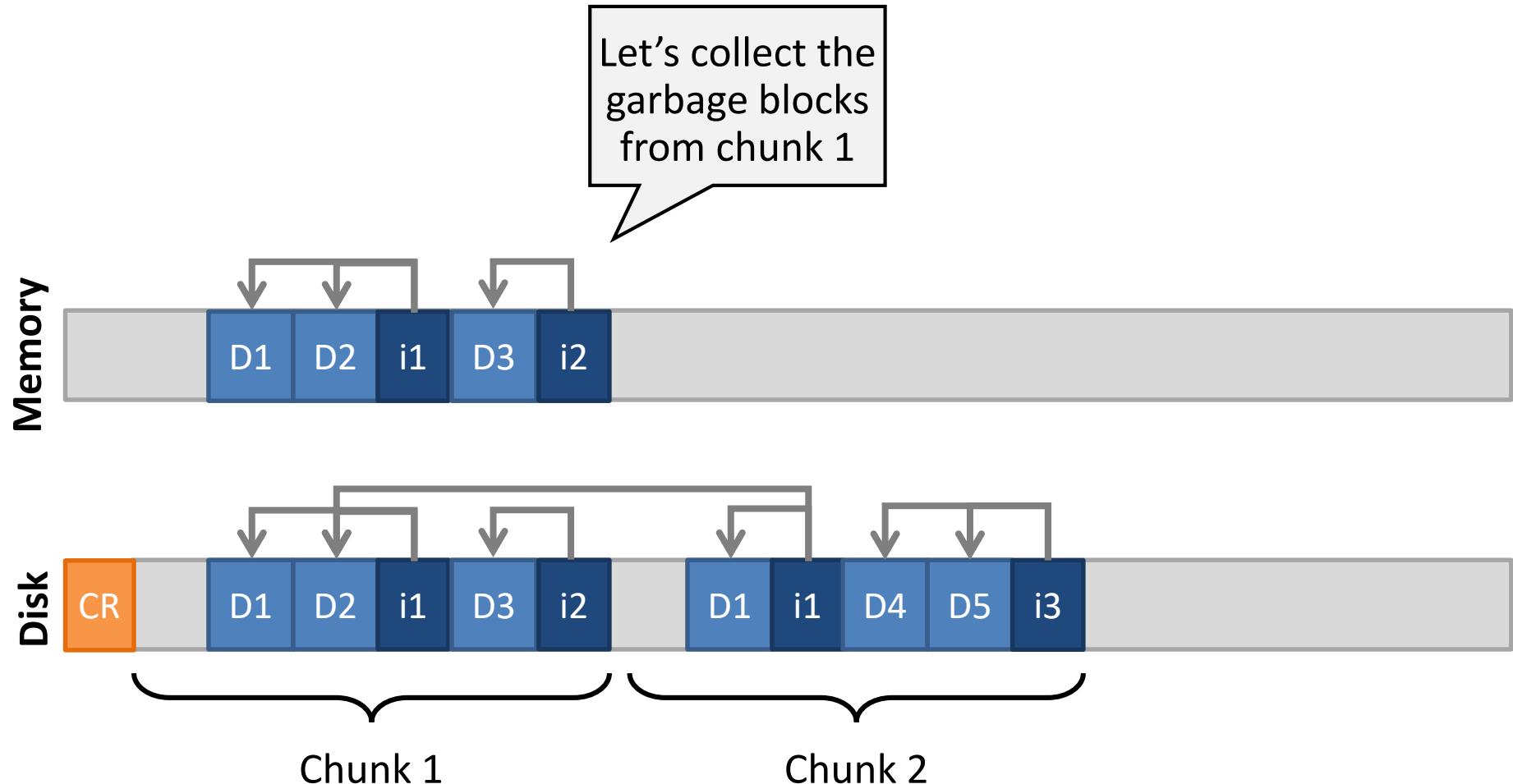
Solution: Garbage Collection

Let's check how GC works in LFS





Solution: Garbage Collection





Solution: Garbage Collection

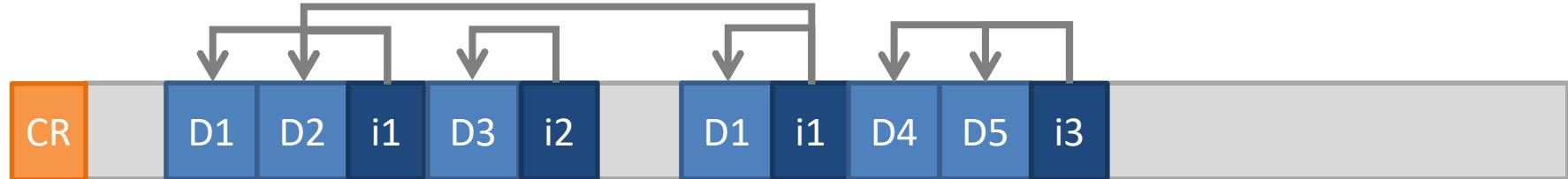
How could we know which blocks are **staled**?

The pointers residing at other chunks are in a **unknown state** from the chunk 1's viewpoint

Memory



Disk



Chunk 1

Chunk 2



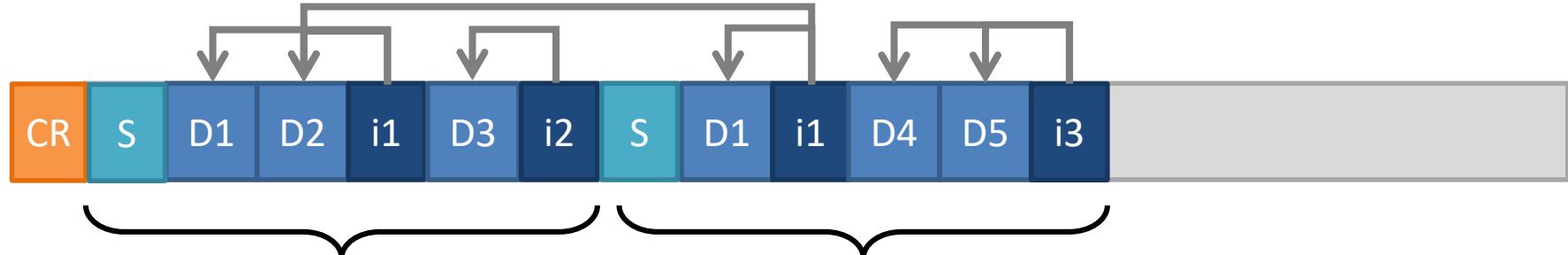
Solution: Garbage Collection

All chunks have their own **summary block**, which contains the {data block → inode} mapping for its each data block

Memory



Disk



Chunk 1

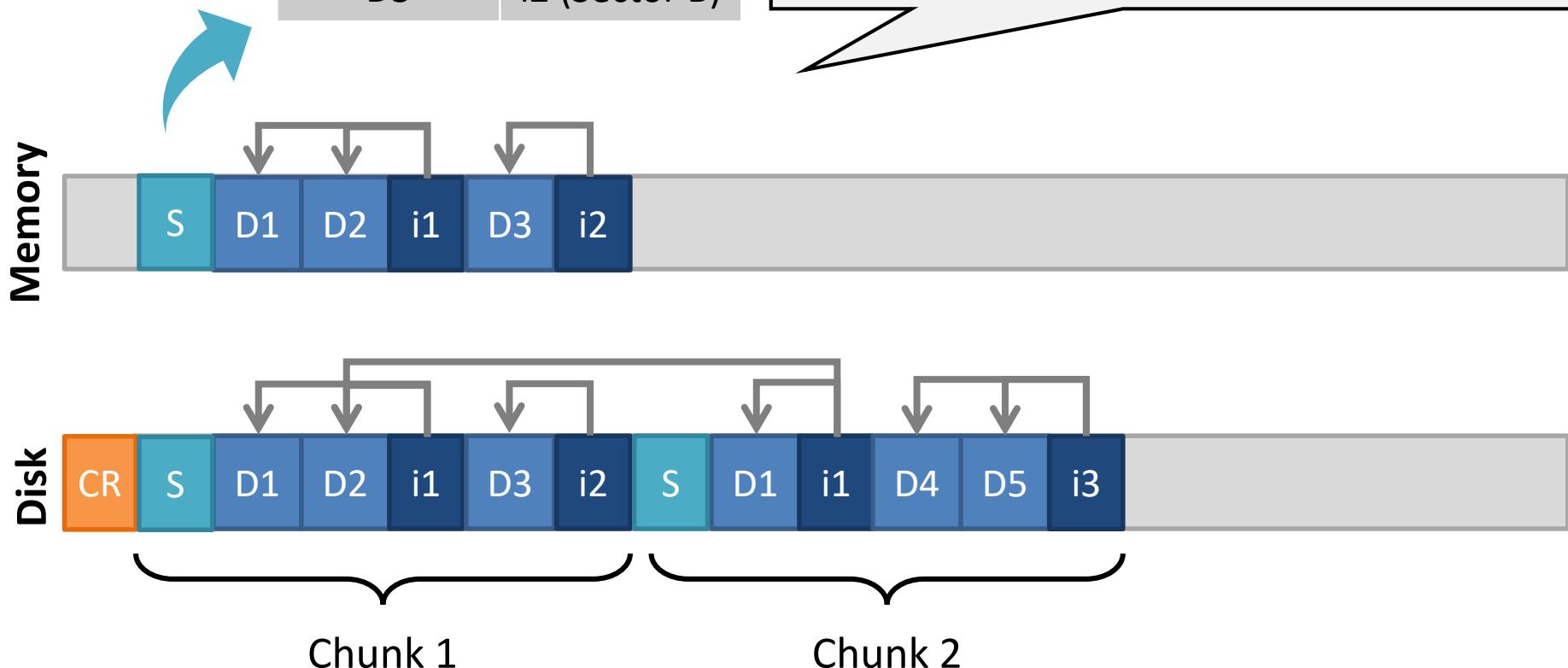
Chunk 2



Solution: Garbage Collection

Data block	inode#
D1	i1 (Sector A)
D2	i1 (Sector A)
D3	i2 (Sector B)

To validate the consistency, the garbage collector of LFS check all the state-of-the art information by reading out the inodes corresponding with the target chunk

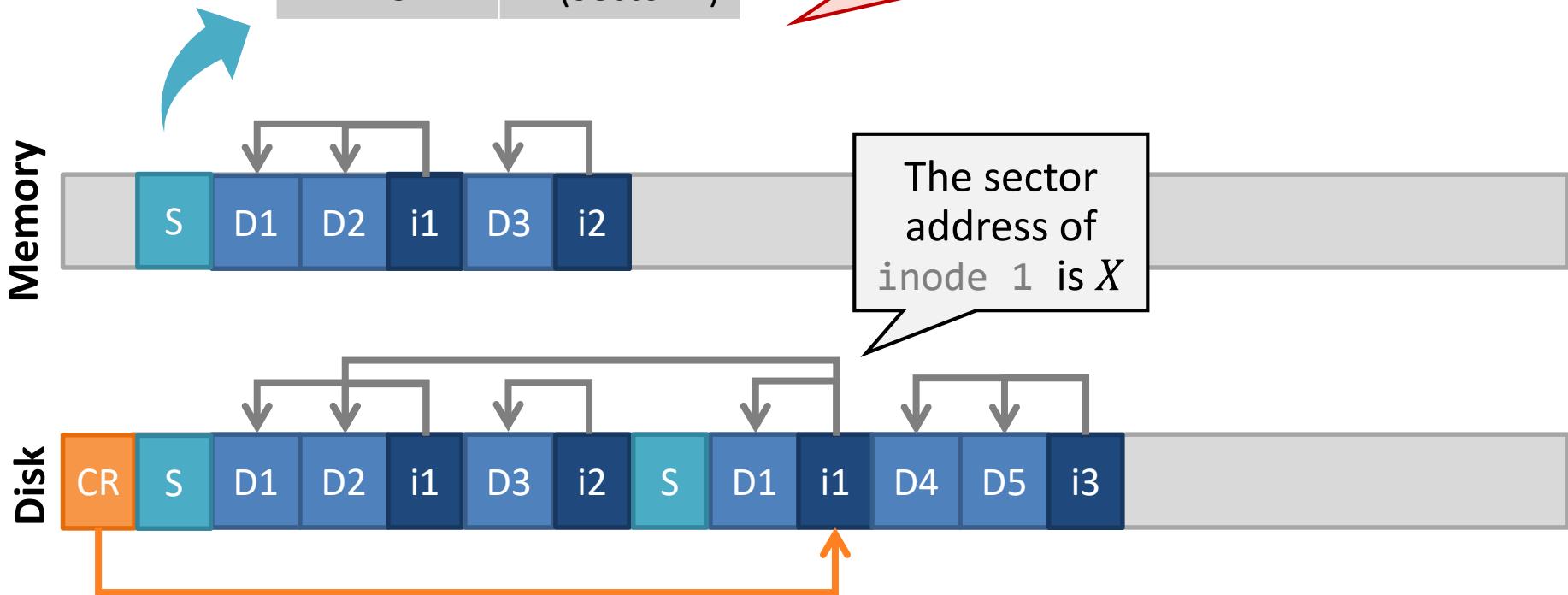




Solution: Garbage Collection

Data block	inode#
D1	i1 (Sector A)
D2	i1 (Sector A)
D3	i2 (Sector B)

However, the summary block in the target chunk records sector address of inode 1 as A





Solution: Garbage Collection

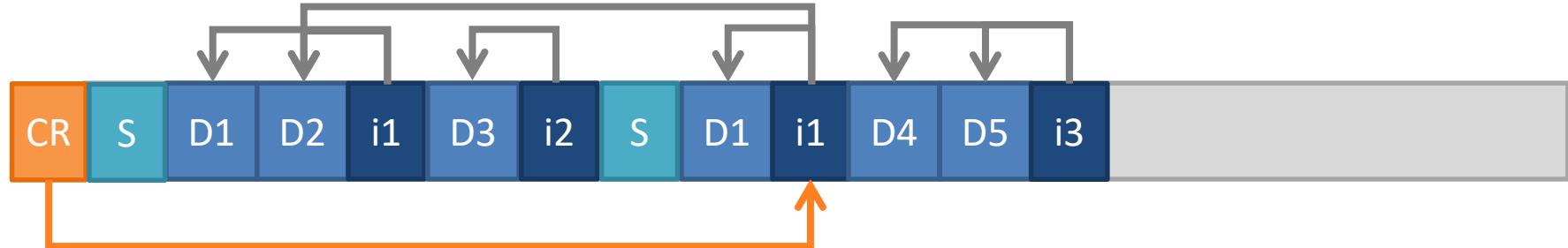
Data block	inode#
D1	i1 (Sector A)
D2	i1 (Sector A)
D3	i2 (Sector B)

Memory



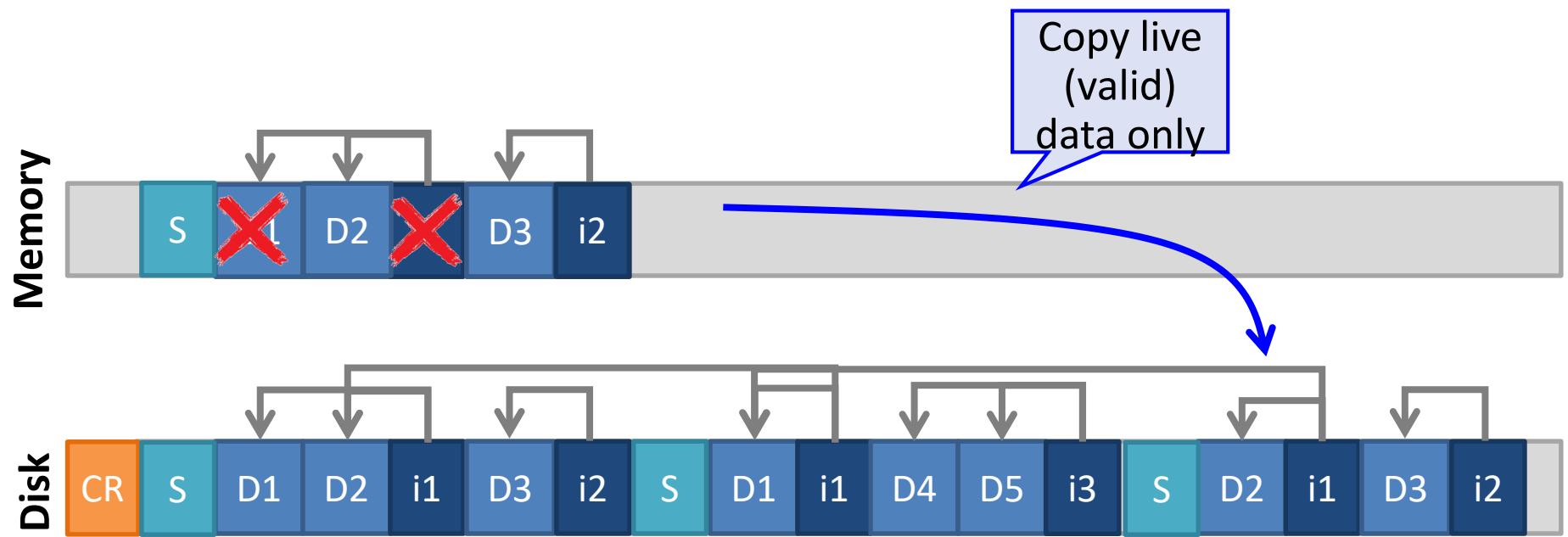
If the current info doesn't match with the summary block, it means that blocks are **staled**

Disk



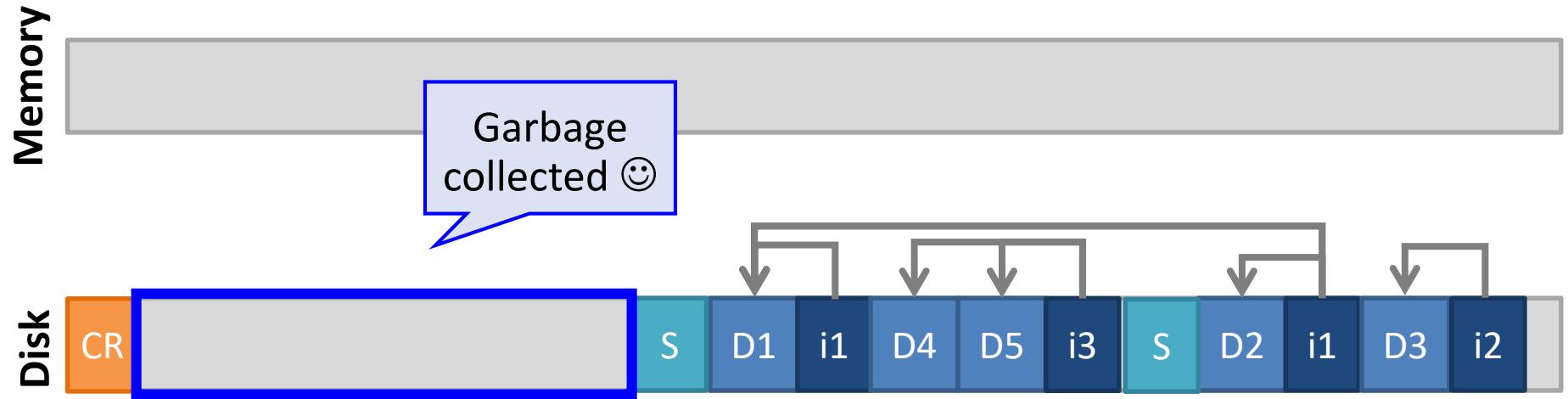


Solution: Garbage Collection





Solution: Garbage Collection



LFS in Practice

LFS seems like a very **strange** design, as it is different with what we learn for FAT/EXT (e.g., small writes, bitmap, etc.)

However, today LFS's features are **widely leveraged** in many practical file systems



#1: Versioning file systems

Many modern file systems, including MacOS, are **versioned**



MacTMOS



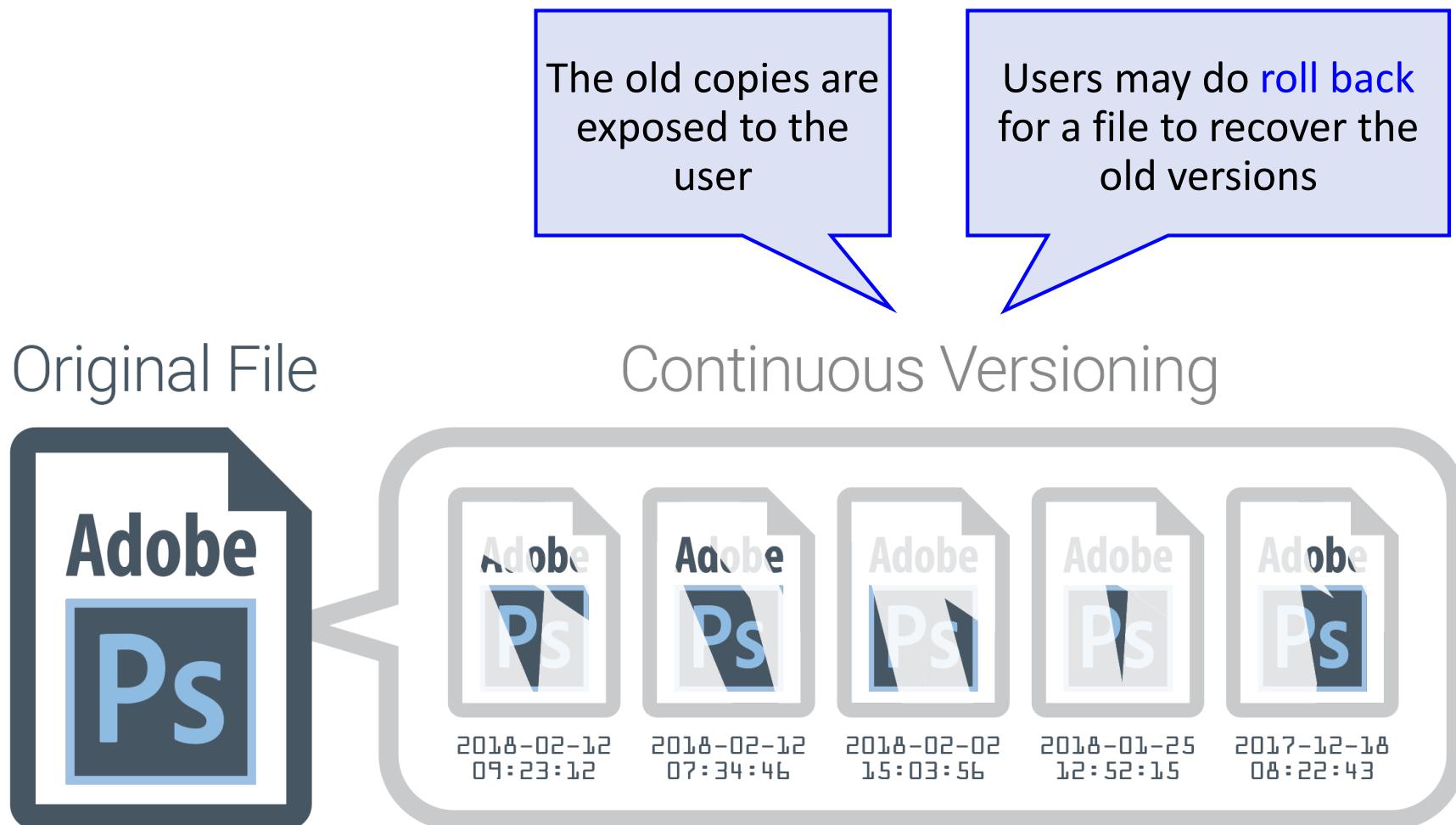
Motivation: Useful Old Versions

Old versions of files may be useful 😊

You often **delete** a file without a careful attention.



Leveraged Feature: Keep Old Copy



#2: File Systems for SSD

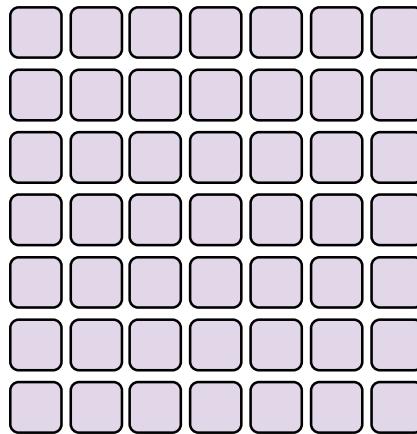


LFS is well
harmonized with
SSDs



Motivation: SSD's Constraints

Periodically, old blocks need to be **reclaimed as garbage**, which increases P/E cycles.



High P/E Cycles



Medium P/E Cycles



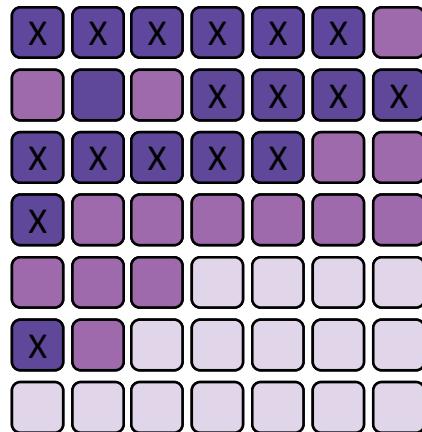
Low P/E Cycles



Motivation: SSD's Constraints

For wear-leveling,
writes must be
spread across the
blocks of flash

Without Wear Leveling



High P/E Cycles



Medium P/E Cycles



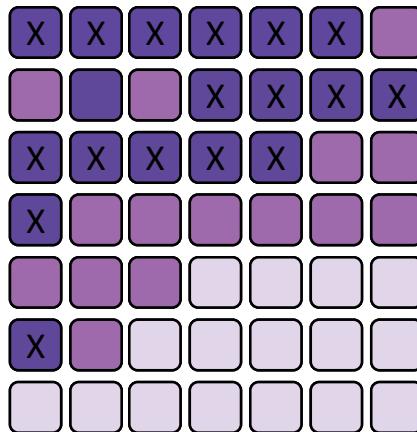
Low P/E Cycles



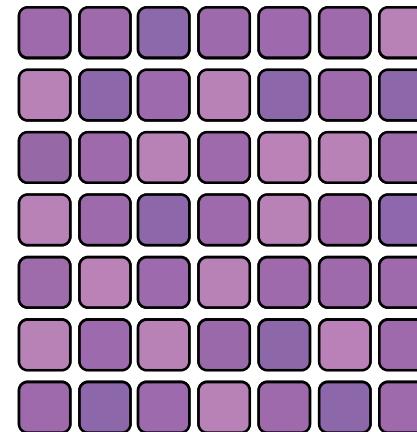
Leveraged Feature: Append-Only

SSD internally manage all files in a LFS (transparent to the OS and end-users)

Without Wear Leveling



With Wear Leveling



High P/E Cycles



Medium P/E Cycles



Low P/E Cycles

Network FS

Quick
introduction of
Network FS



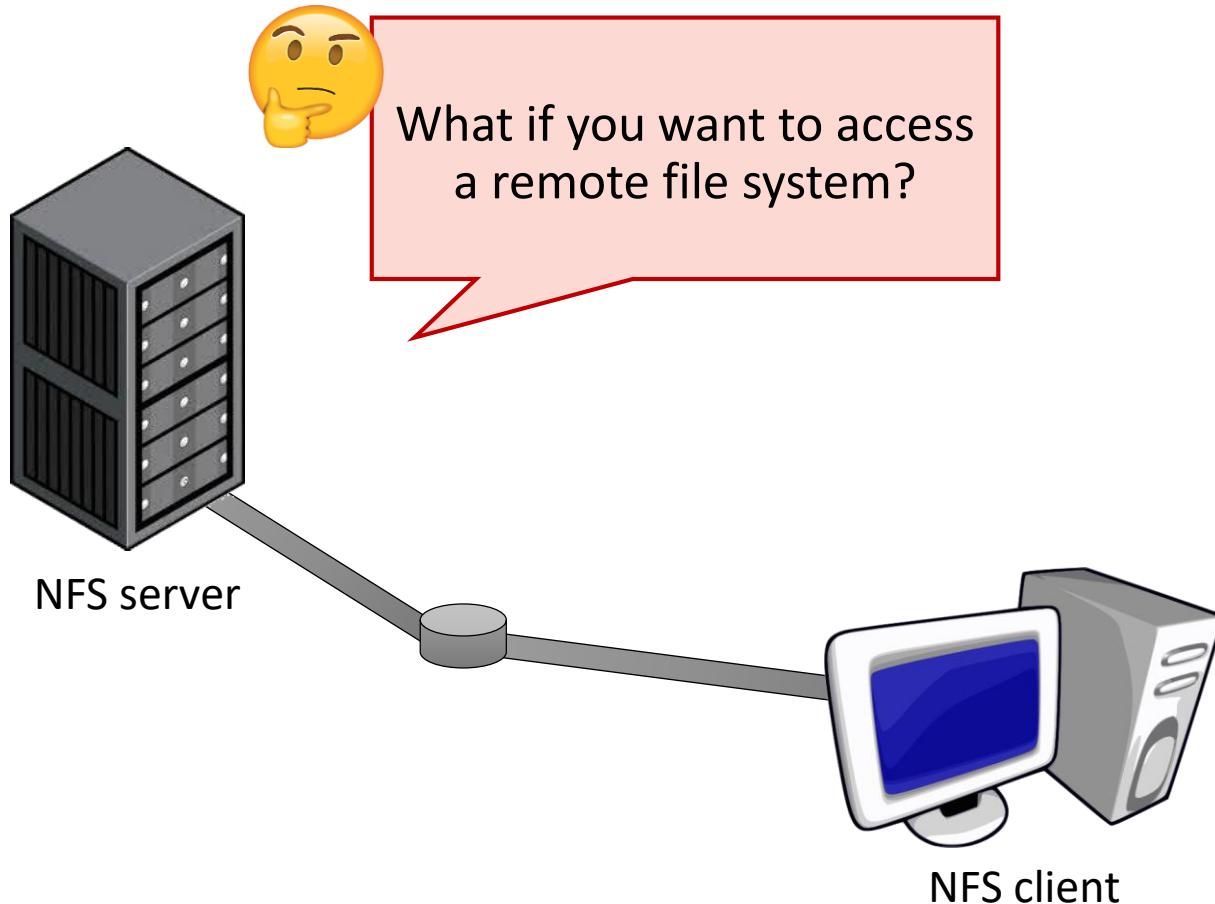
Local file system



Until now, we learned about **local file systems** which are built/installed on your local PC

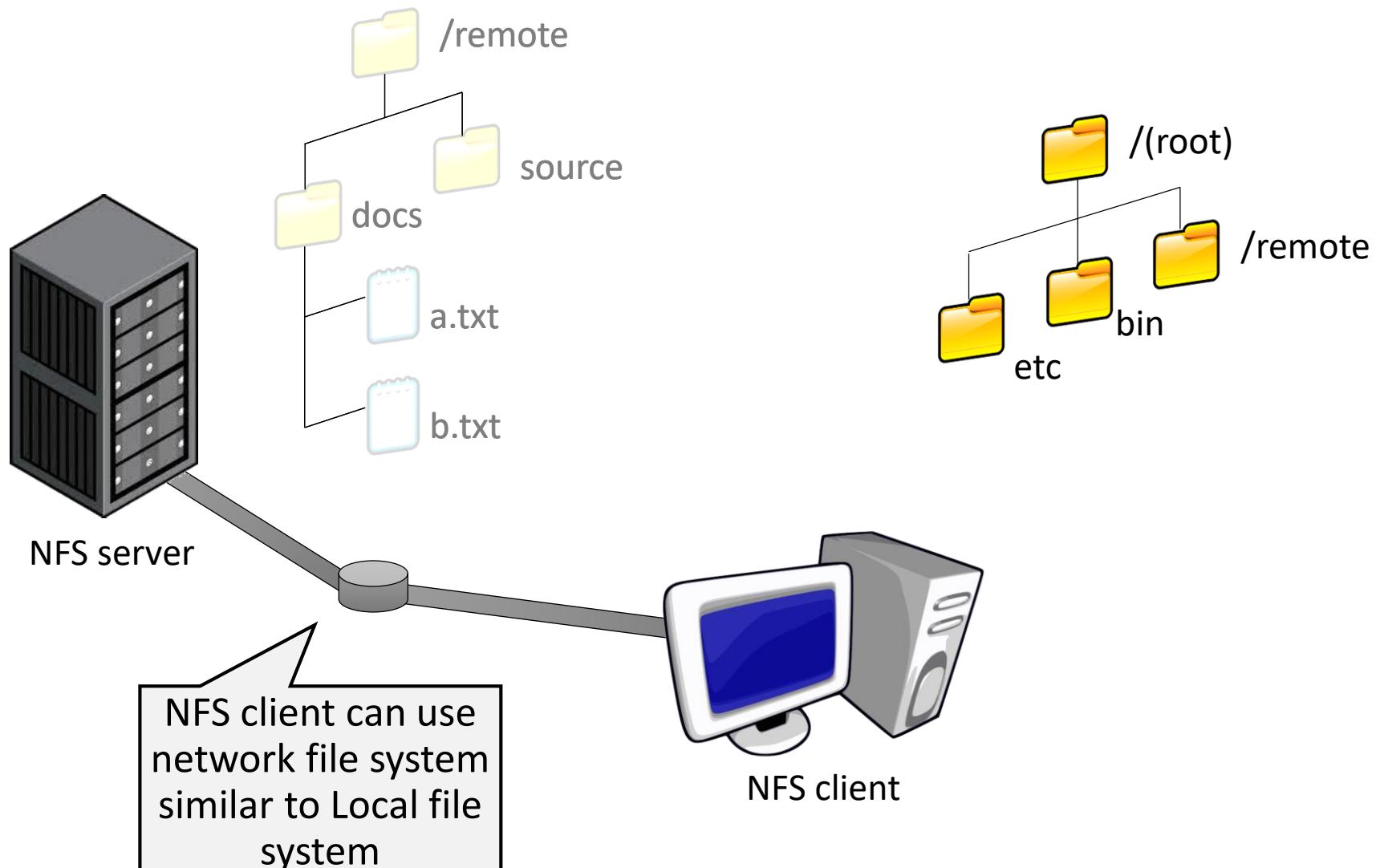


Networked File System



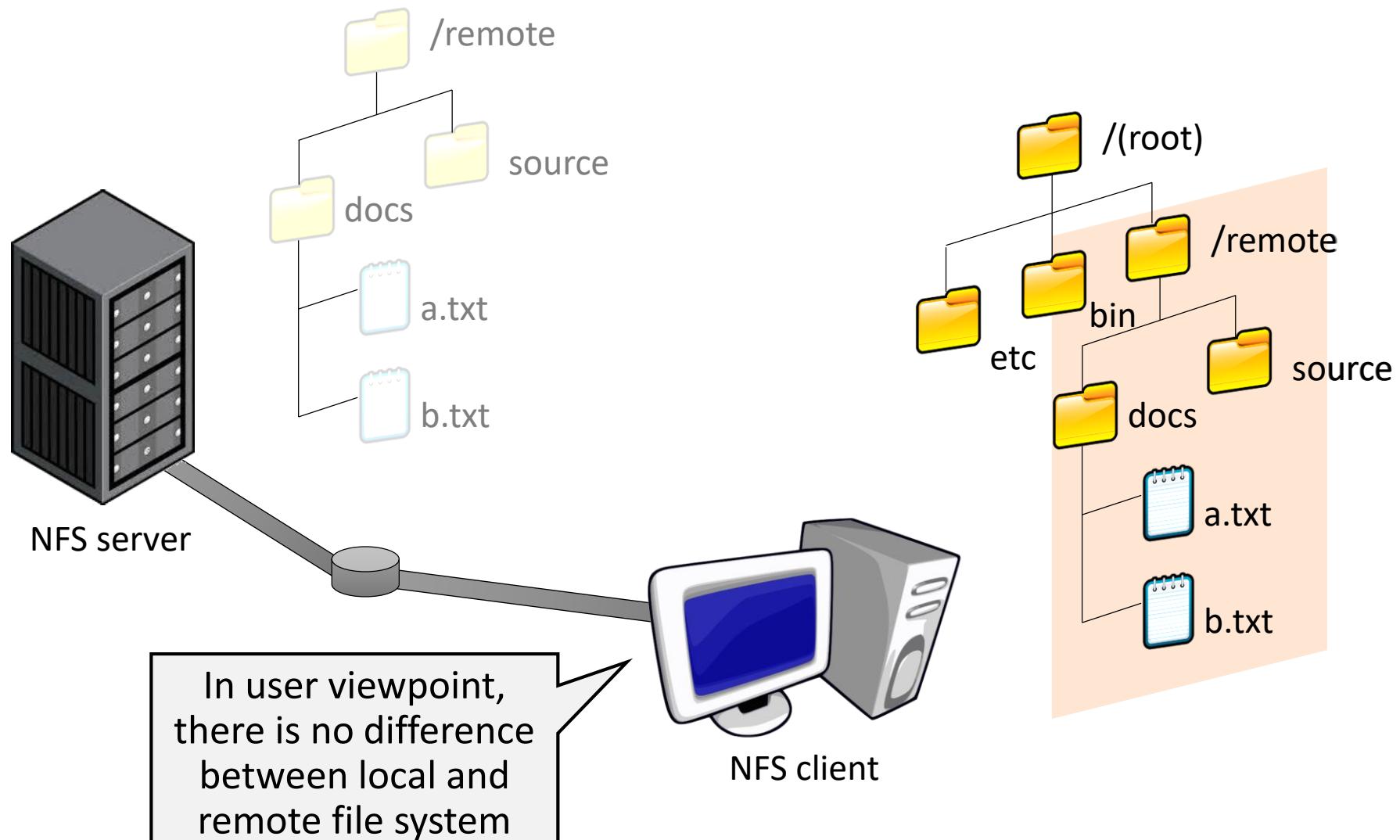


Mounting Network File System



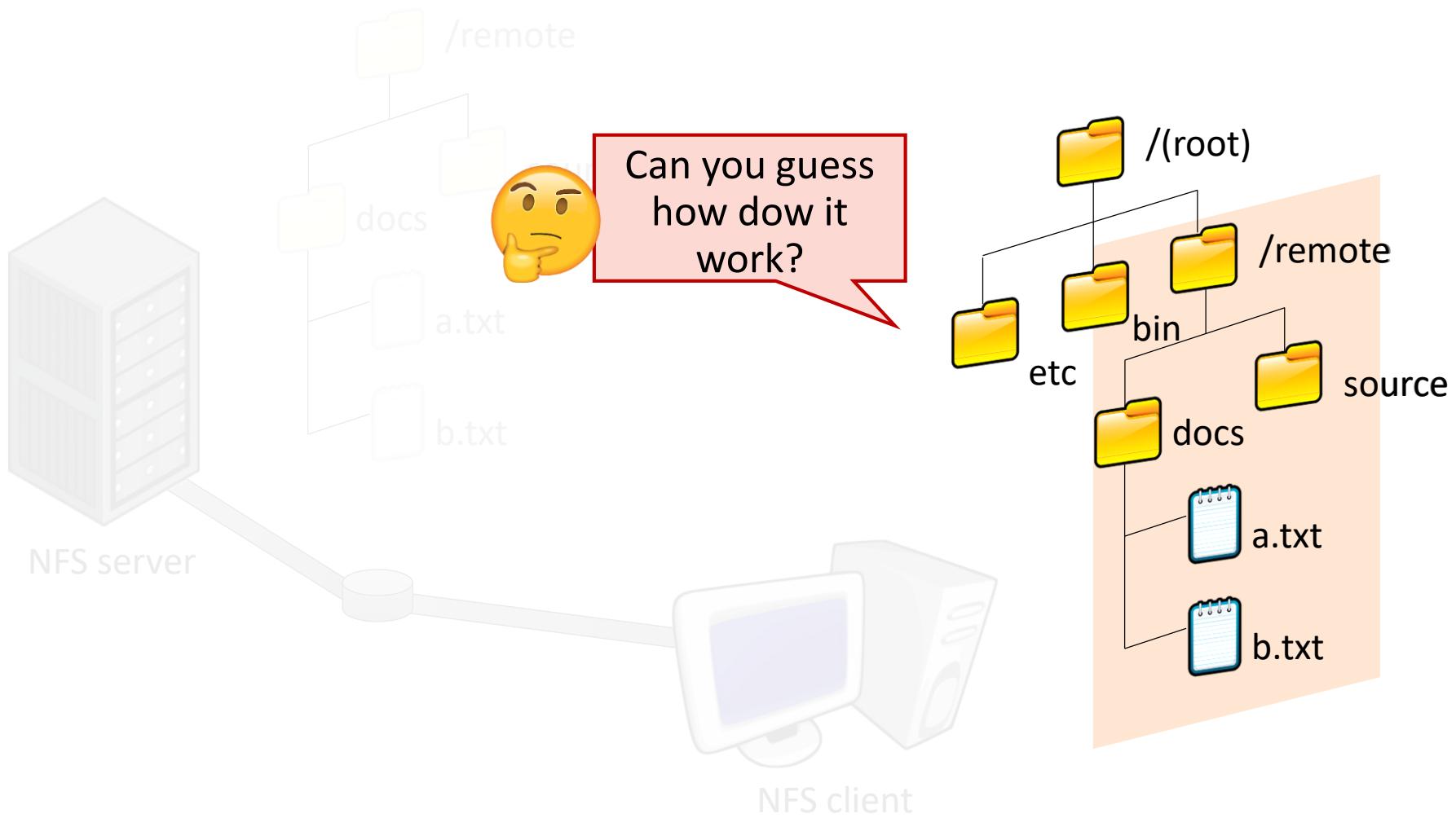


Mounting Network File System





Mounting Network File System



What's Next?

- Let's go over how we can access the storage device in a remote machine 😊

21S EE415

Network File-System I

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



Distributed File-System I

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab* 

Local FS vs. Distributed FS



Hardware View

Let's check the major differences between a **local FS** and a **distributed FS** from hardware and software angles

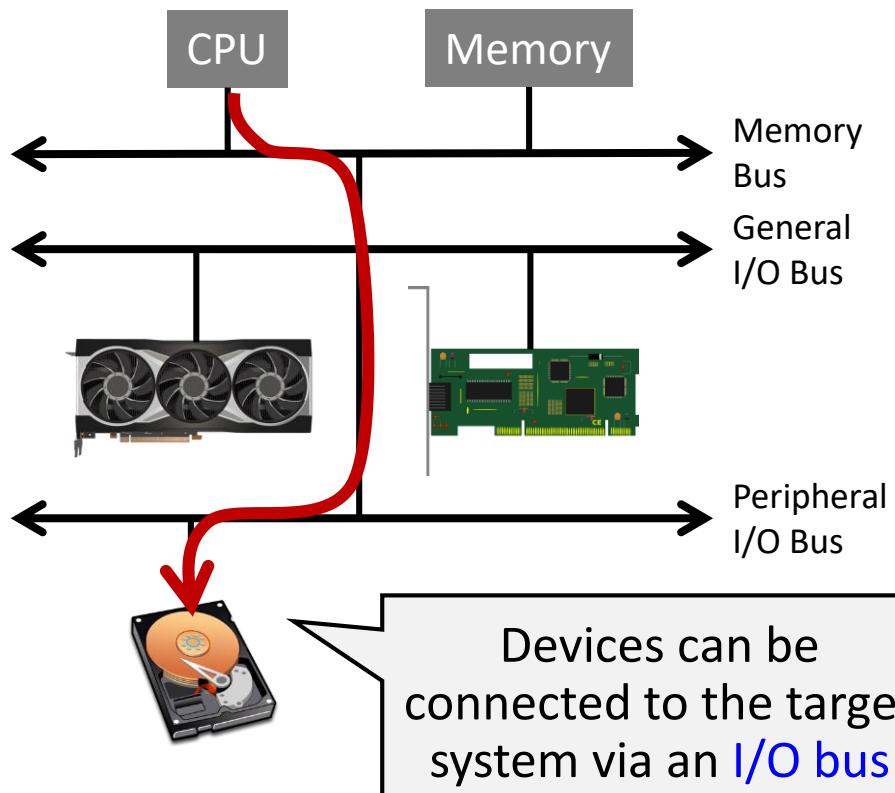


Software (OS) View

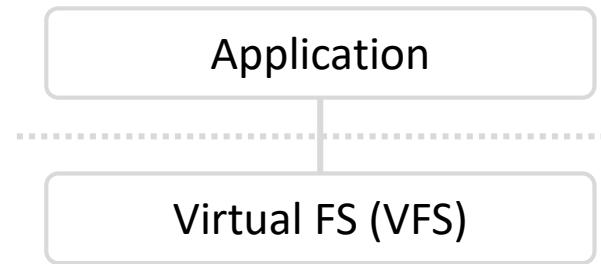
Local FS vs. Distributed FS



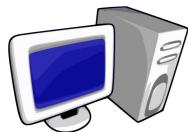
Hardware View



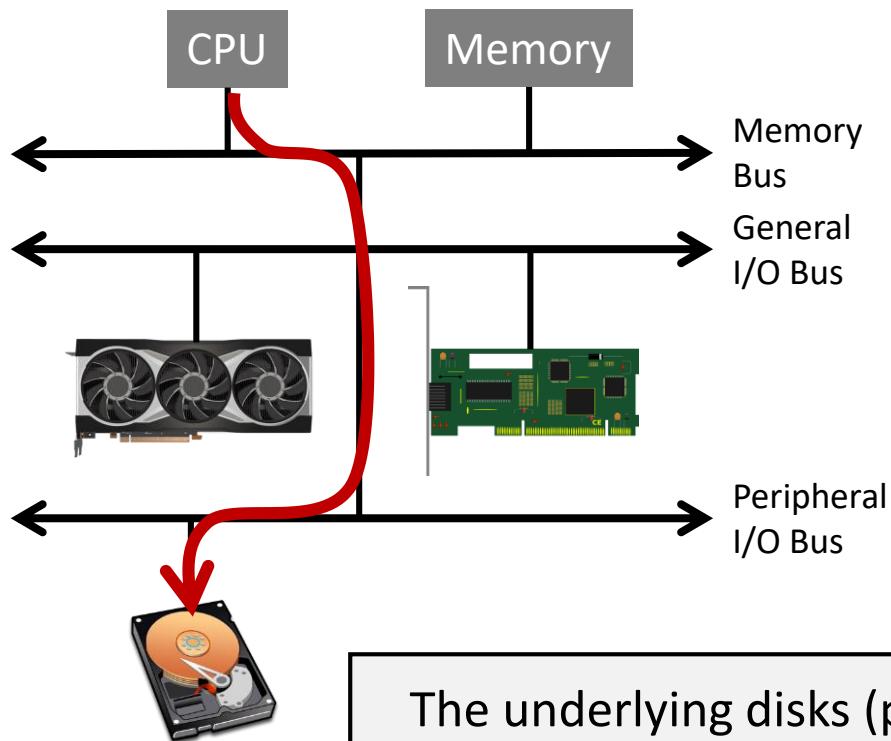
Software (OS) View



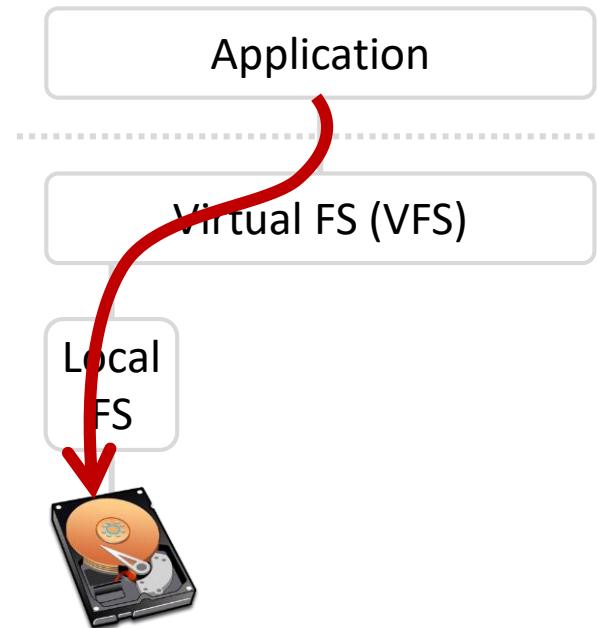
Local FS vs. Distributed FS



Hardware View



Software (OS) View

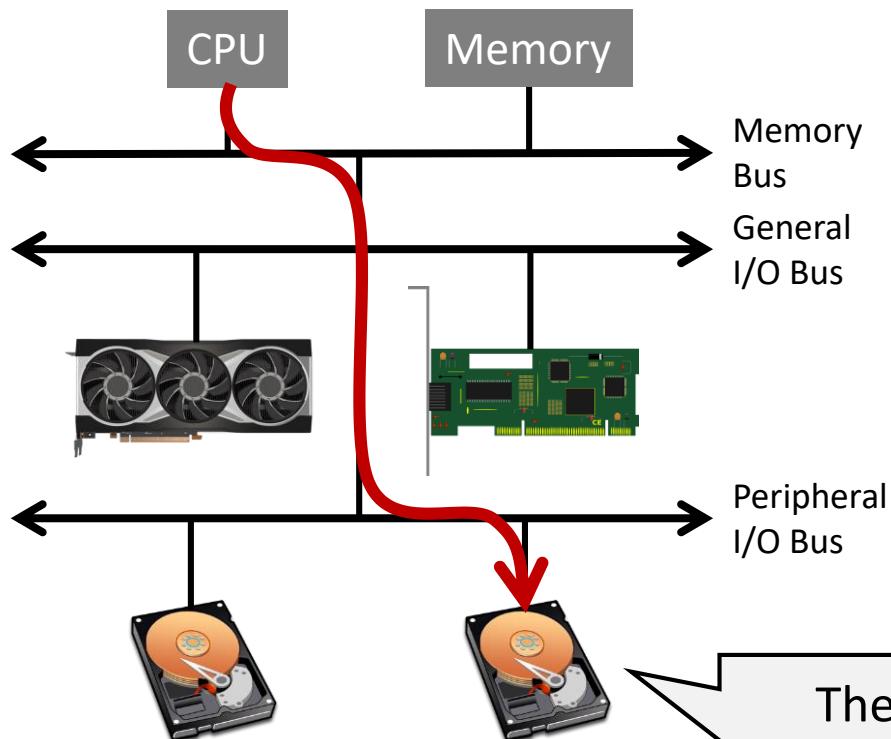


The underlying disks (physically connected to the system's I/O bus) can be accessed via a **local FS**

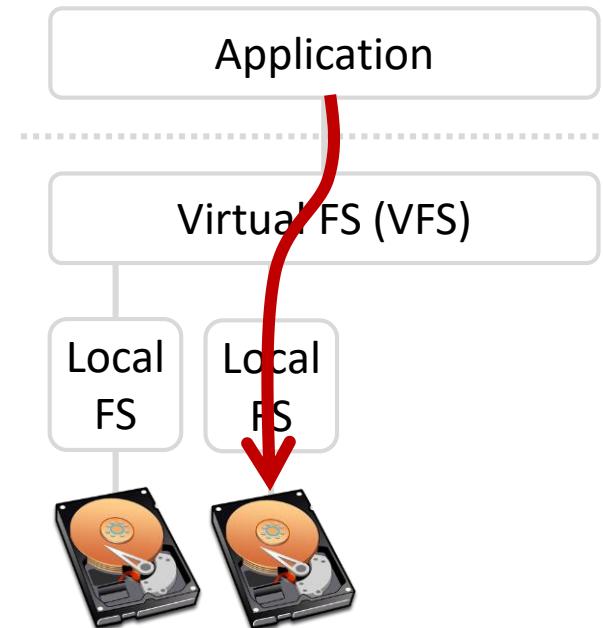
Local FS vs. Distributed FS



Hardware View



Software (OS) View

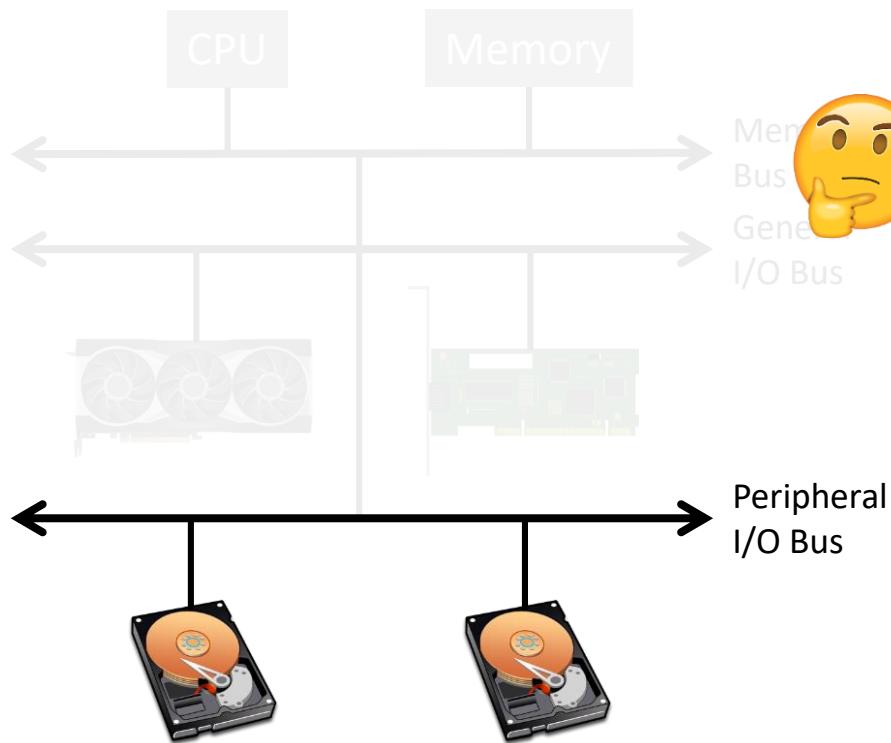


The storage can be extended by attaching more disks to the bus

Local FS vs. Distributed FS



Hardware View



Software (OS) View

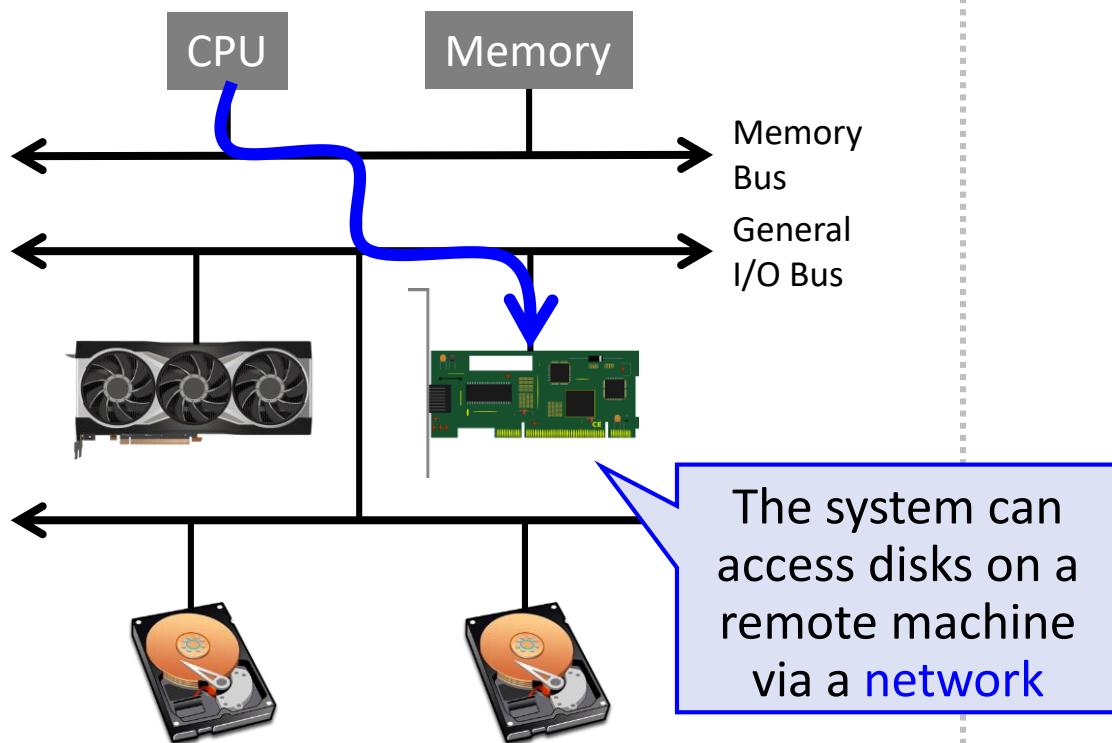
What if the I/O bus has no available port/slot for the storage expansion?



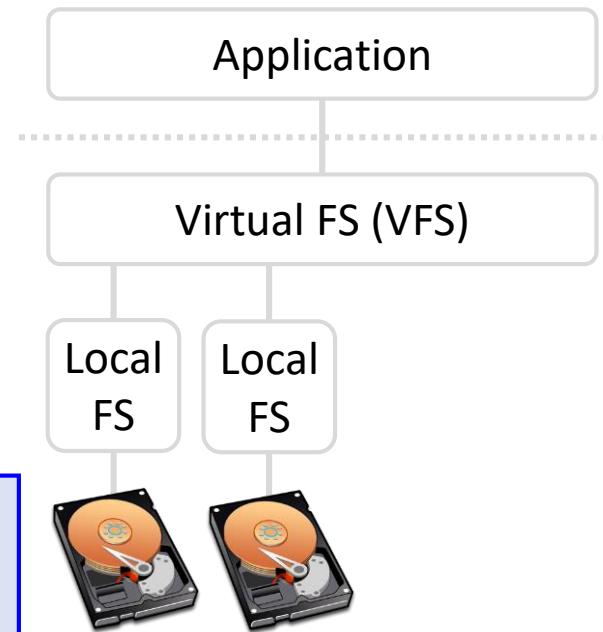
Local FS vs. Distributed FS



Hardware View



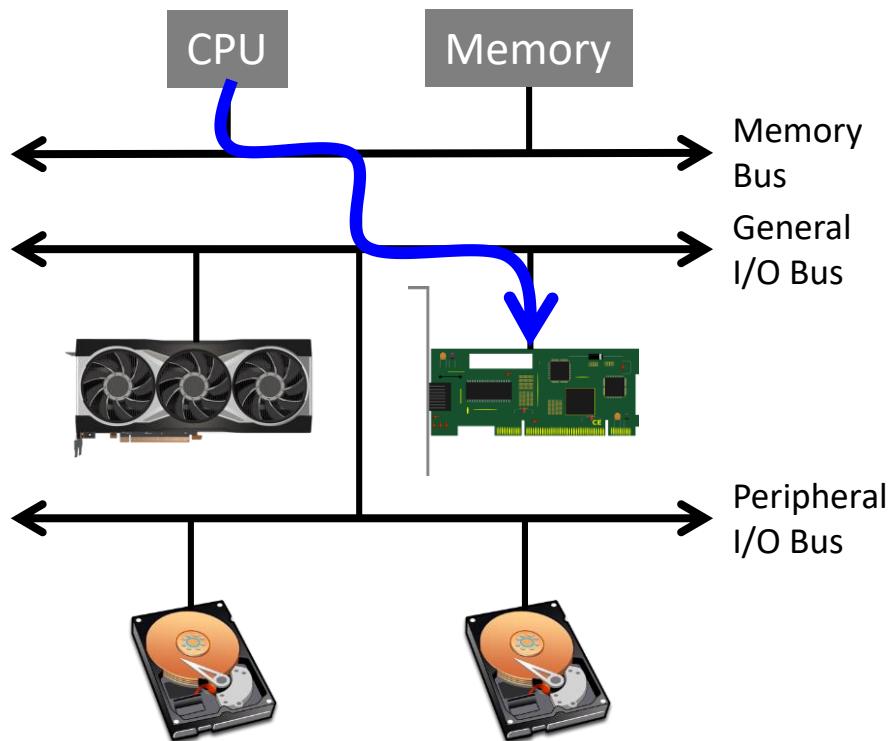
Software (OS) View



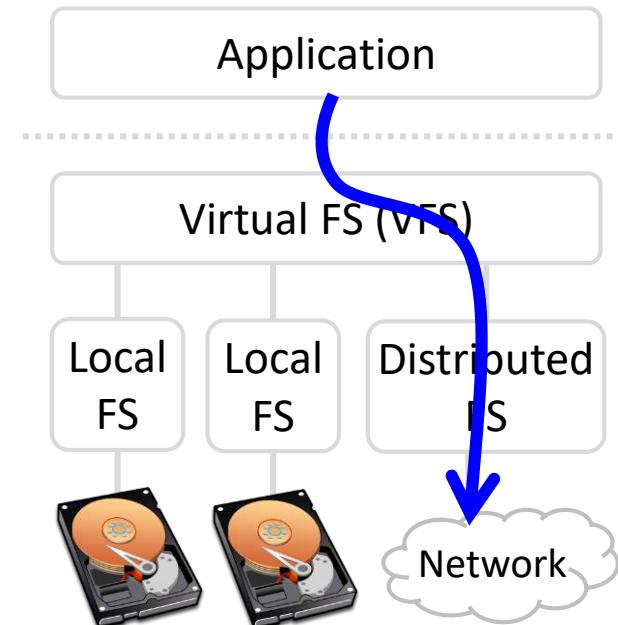
Local FS vs. Distributed FS



Hardware View



Software (OS) View

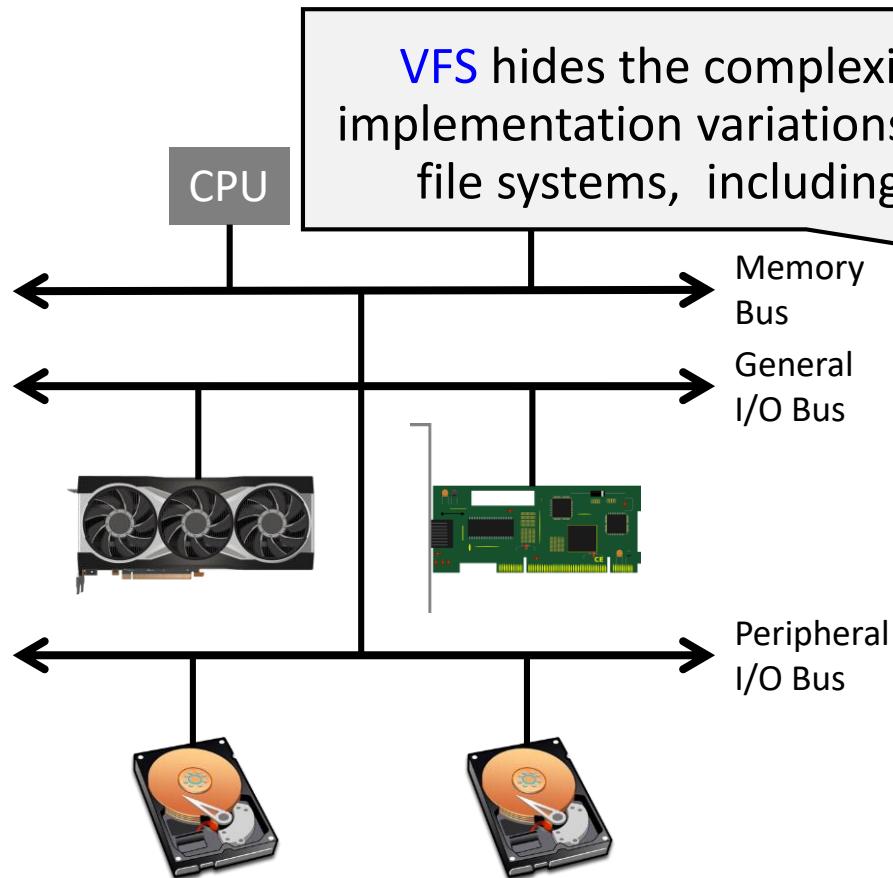


The remote disks can
be accessed by
distributed FS

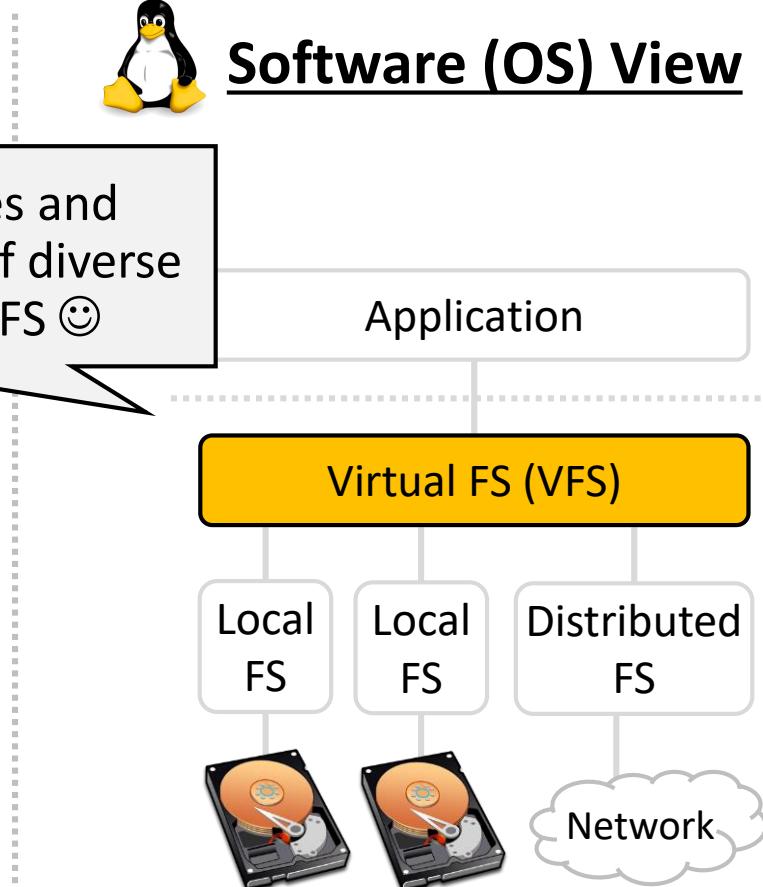
Local FS vs. Distributed FS



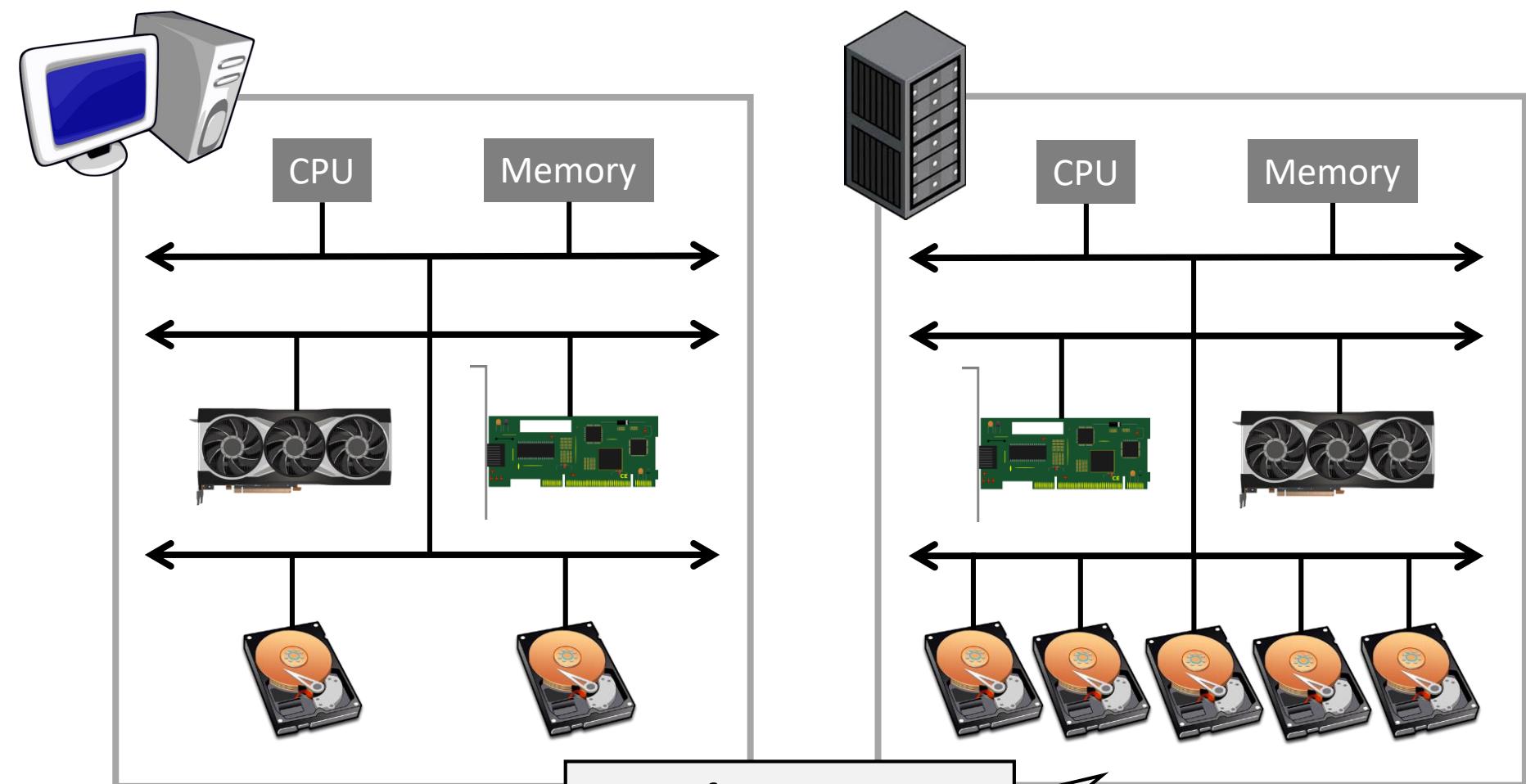
Hardware View



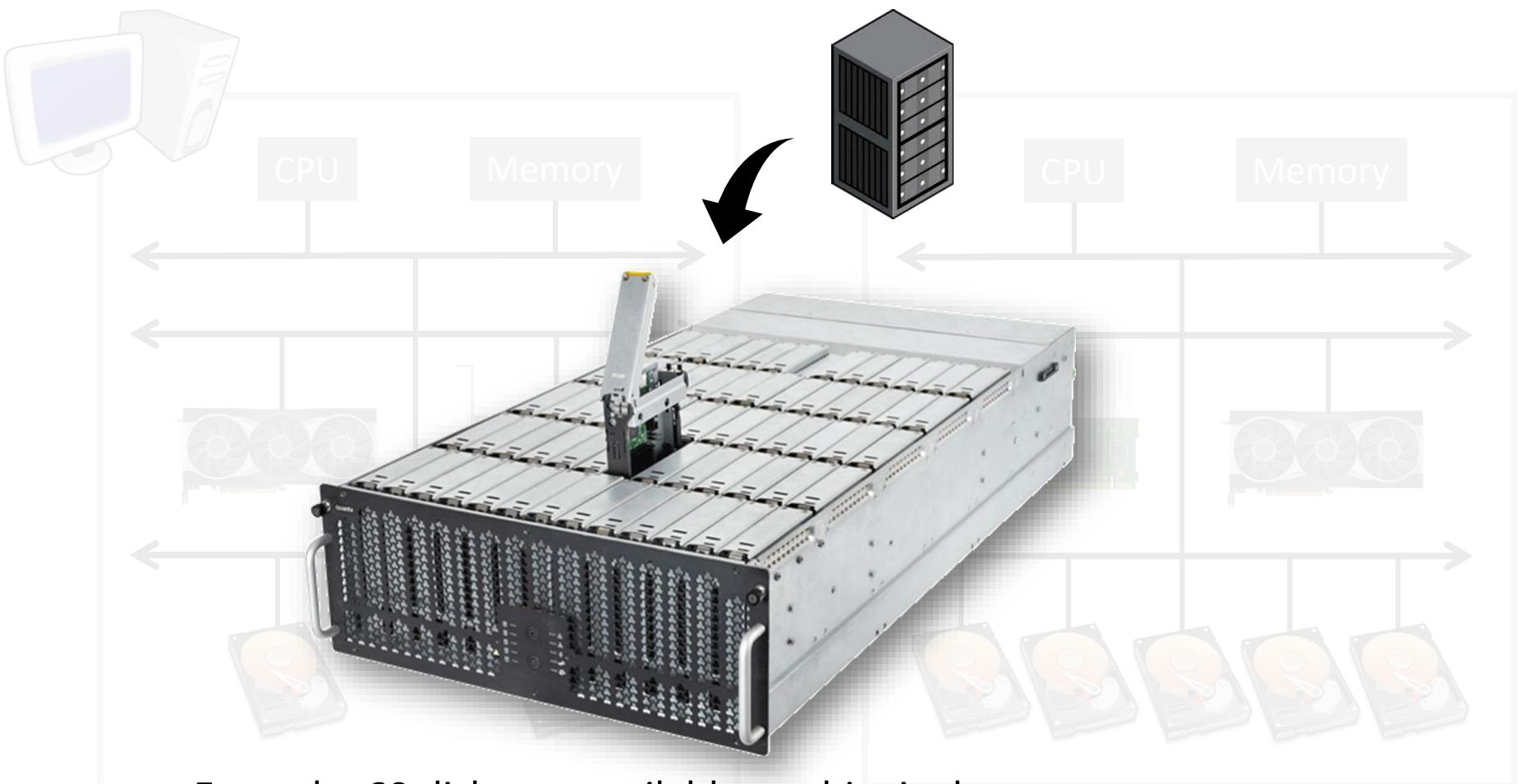
Software (OS) View



Local FS vs. Distributed FS

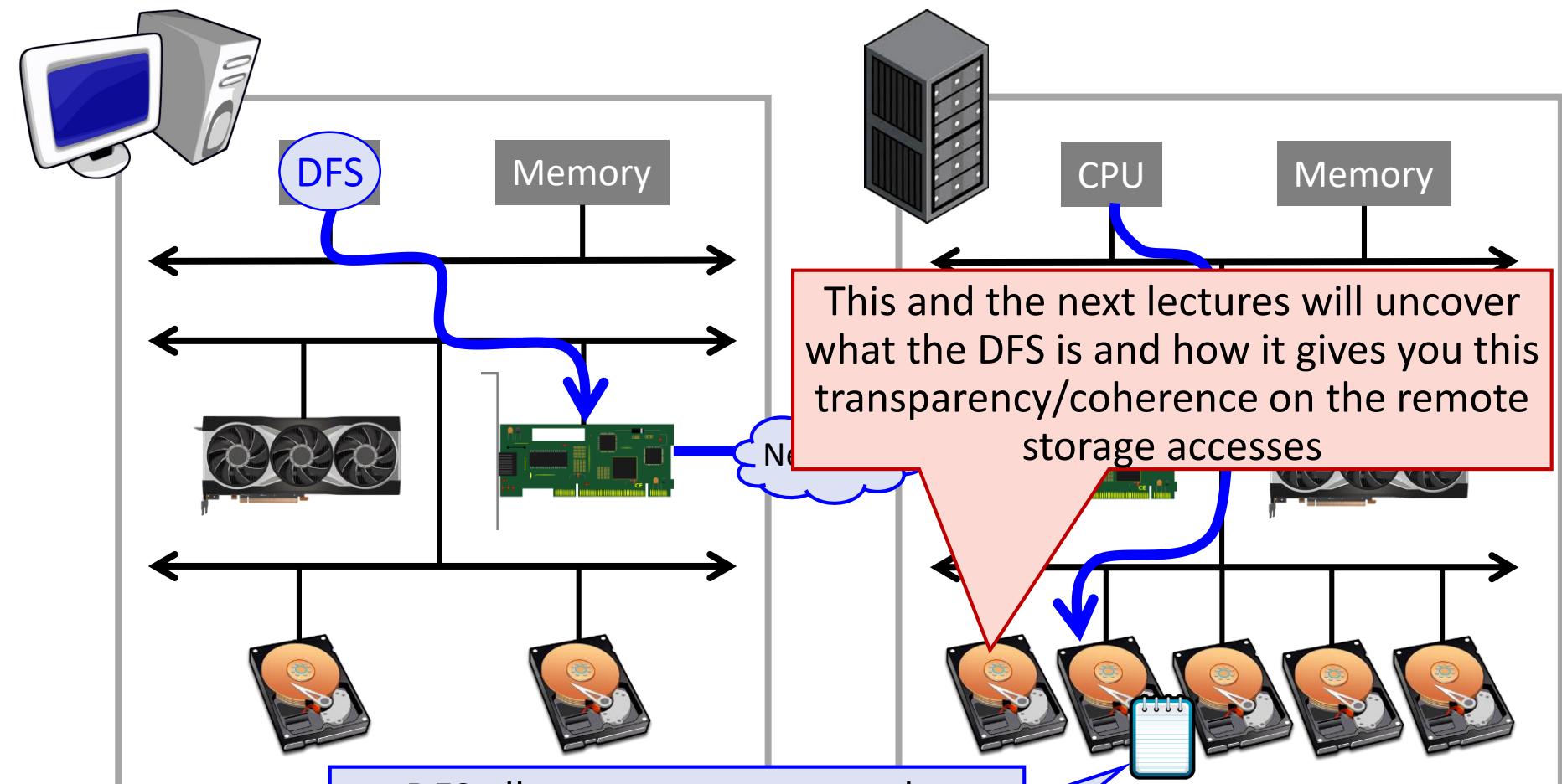


Local FS vs. Distributed FS

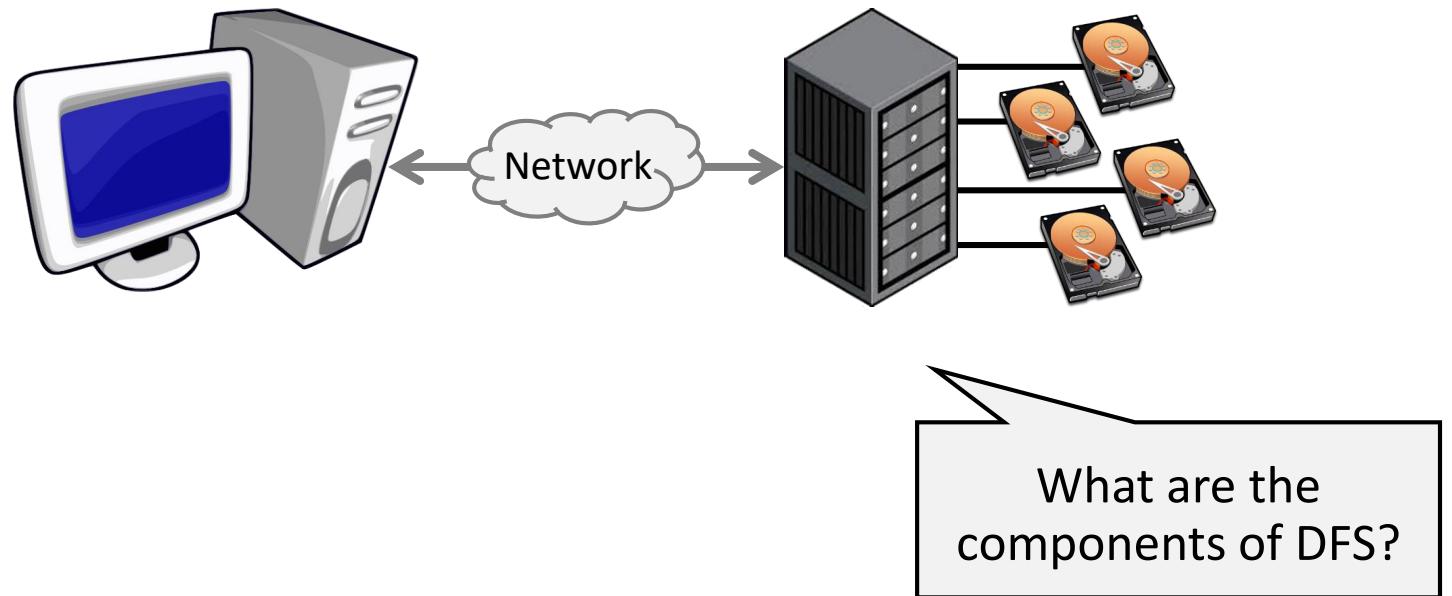


Example: 60 disks are available on this single server

Local FS vs. Distributed FS

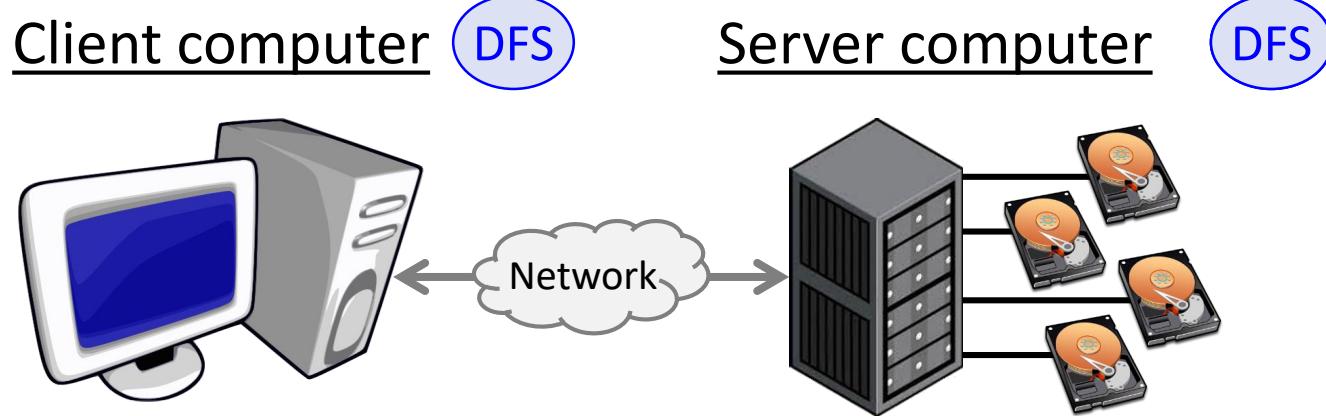


Basic Components of DFS



Terminologies of DFS

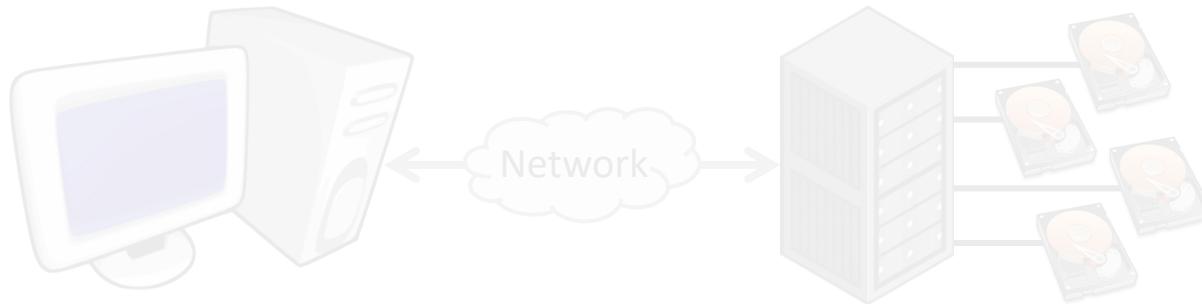
A distributed file system (DFS) can be considered as a *service*, which is offered by one or more *servers* to multiple *clients* (e.g., a server <-> many clients conflagration)



#1: DFS Service

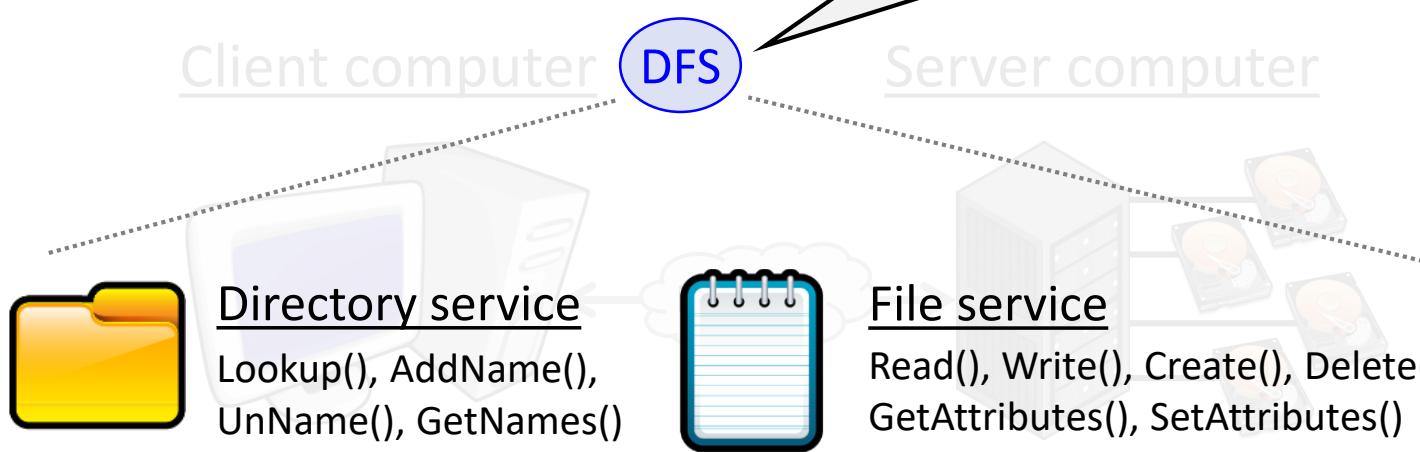
Service consists an interface and associated semantic definitions (behaviors)

Client computer DFS Server computer



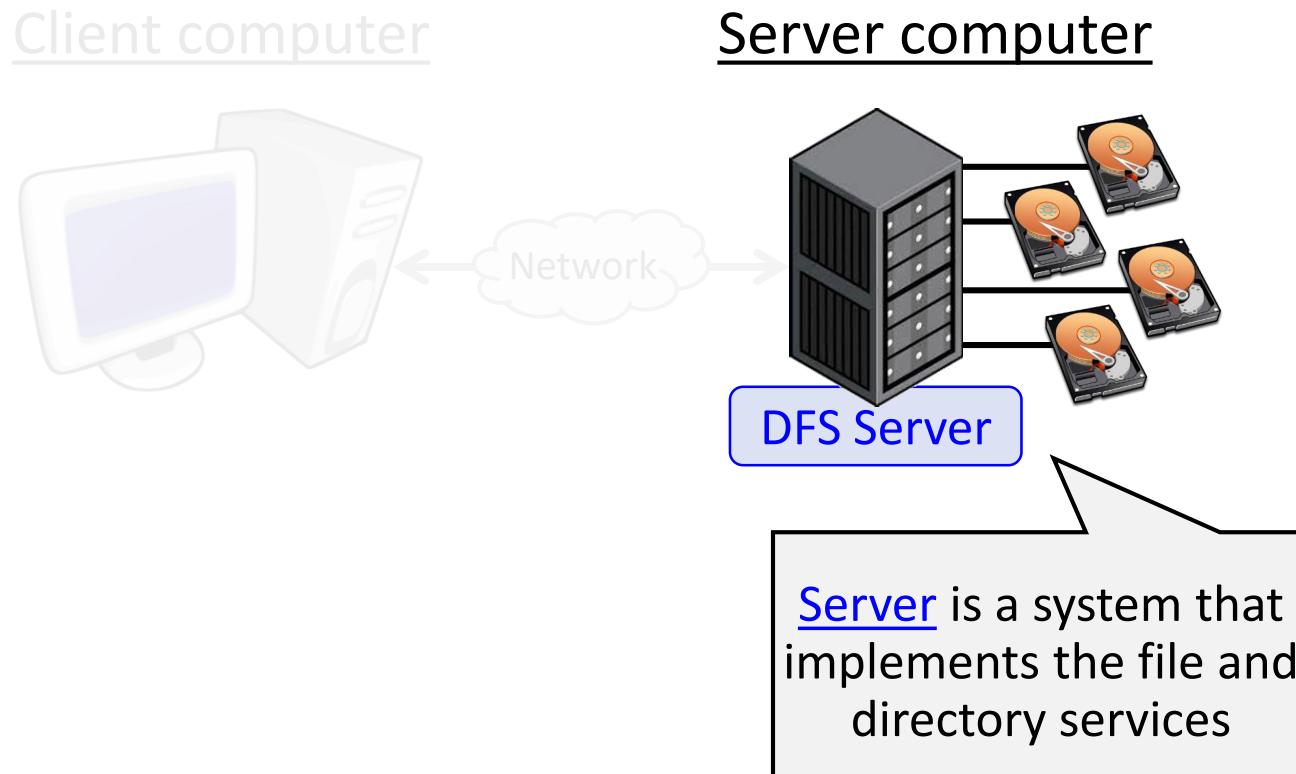
#1: DFS Service

We can classify DFS *services* into two: i) directory and ii) file services; the detailed information will be covered in the next lecture.



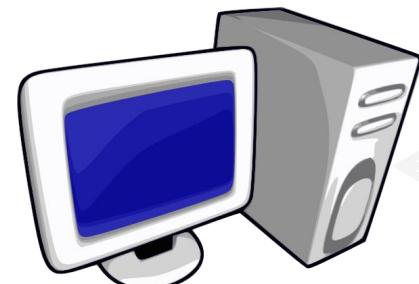
Similar to any of local FS, it provides a set of file access interfaces, but adds more network properties (e.g., AddName, UnName, GetNames, etc.)

#2: DFS Server



#3: DFS Client

Client computer



DFS Client

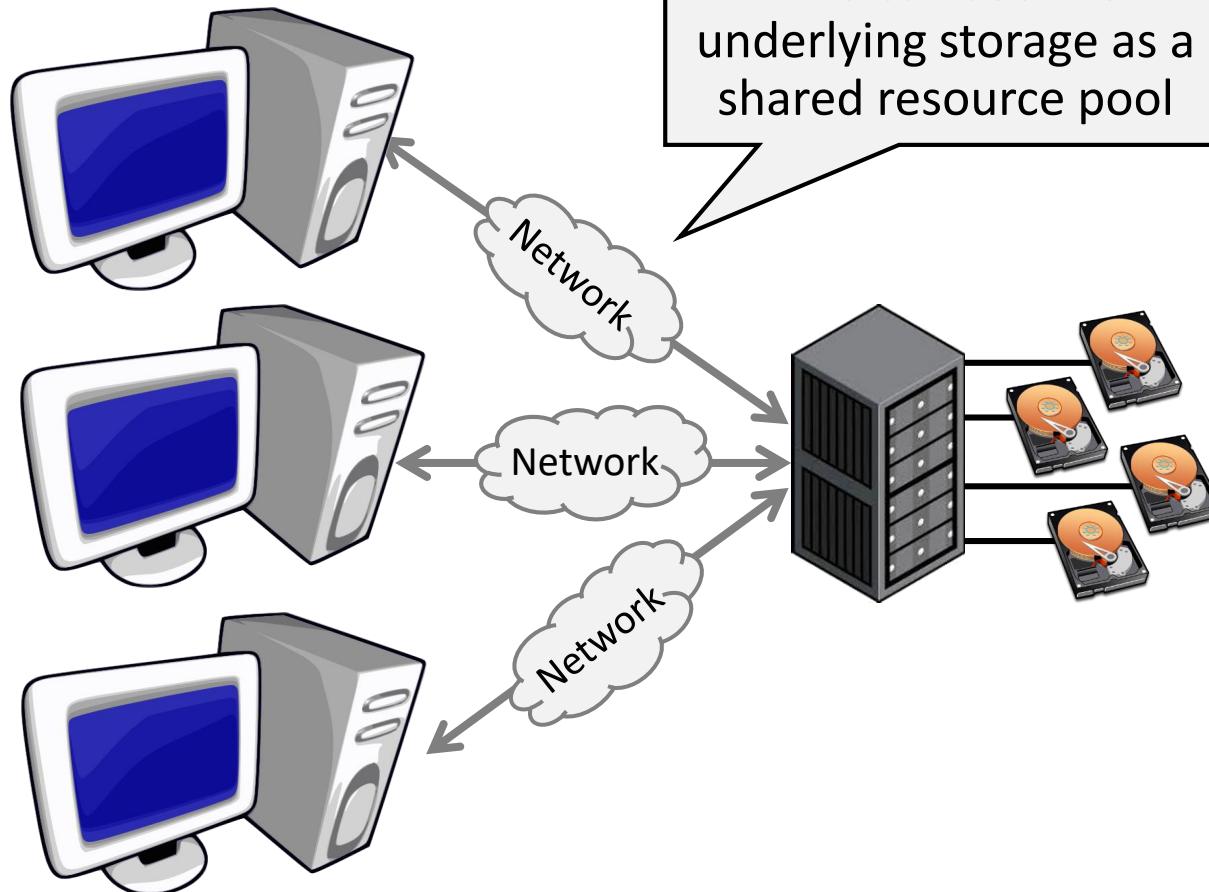
Server computer



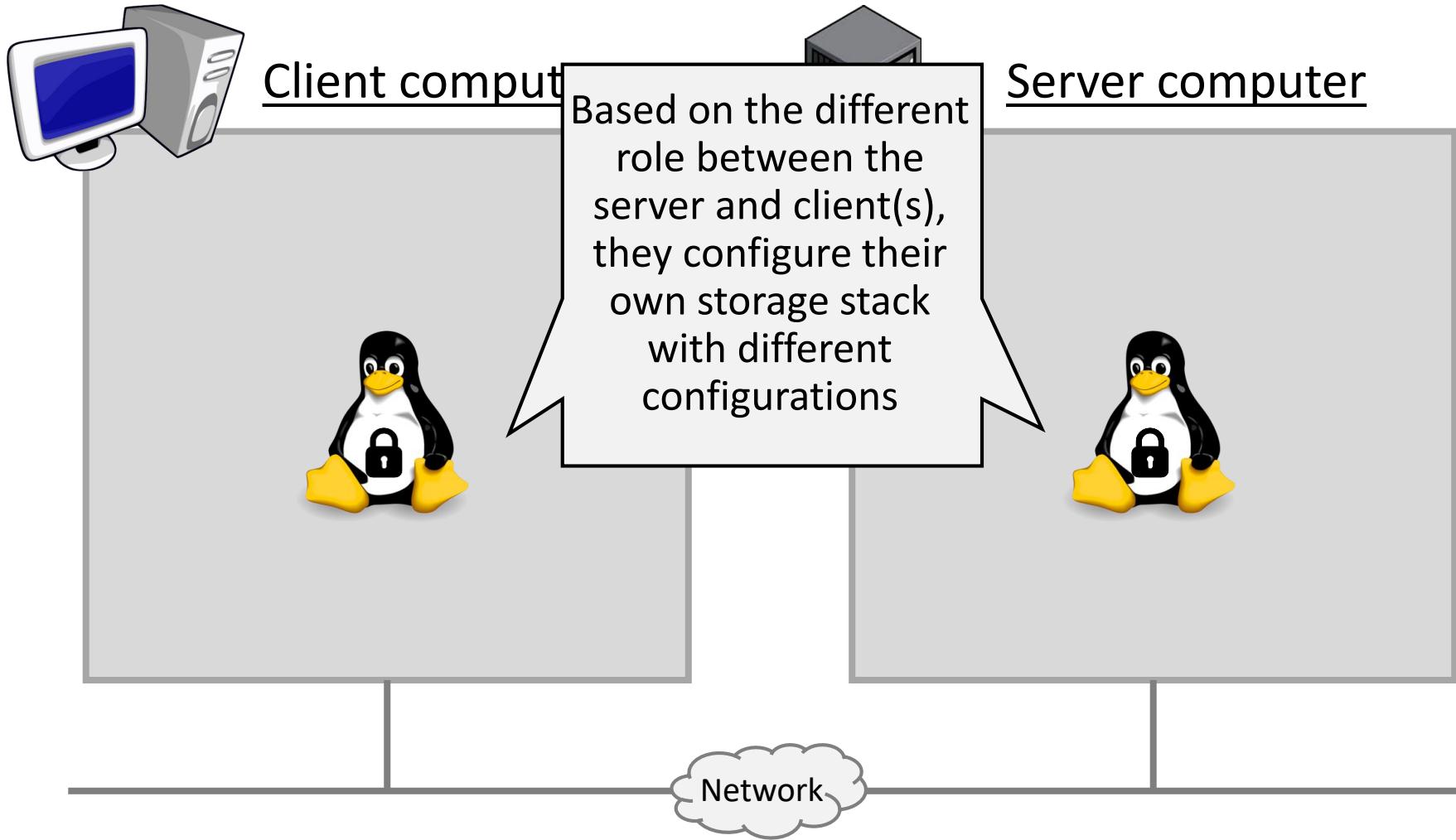
DFS Server

Client is a process, which provides remote storage accessibility to the users by dealing with DFS's service operations.

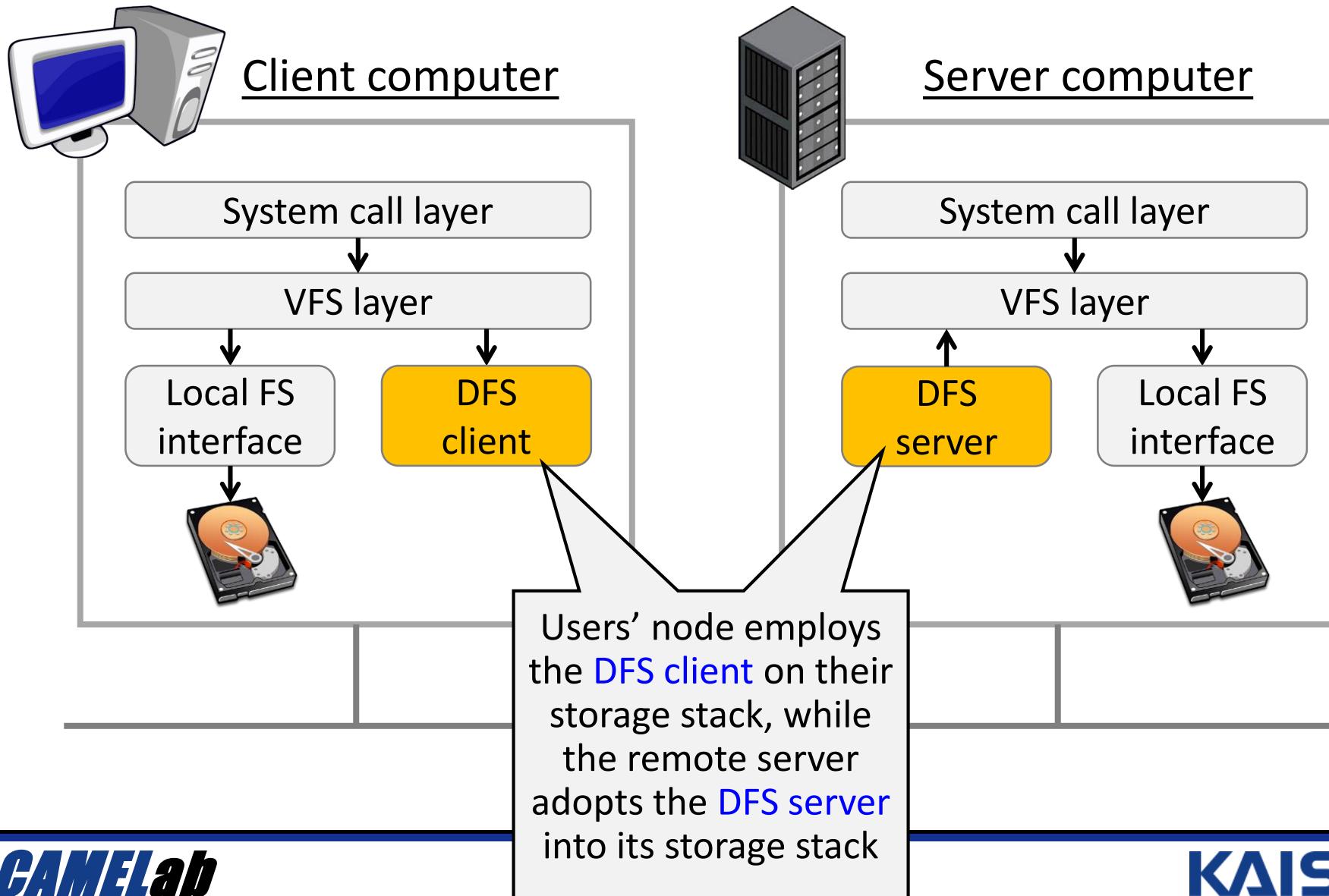
#3: DFS Client



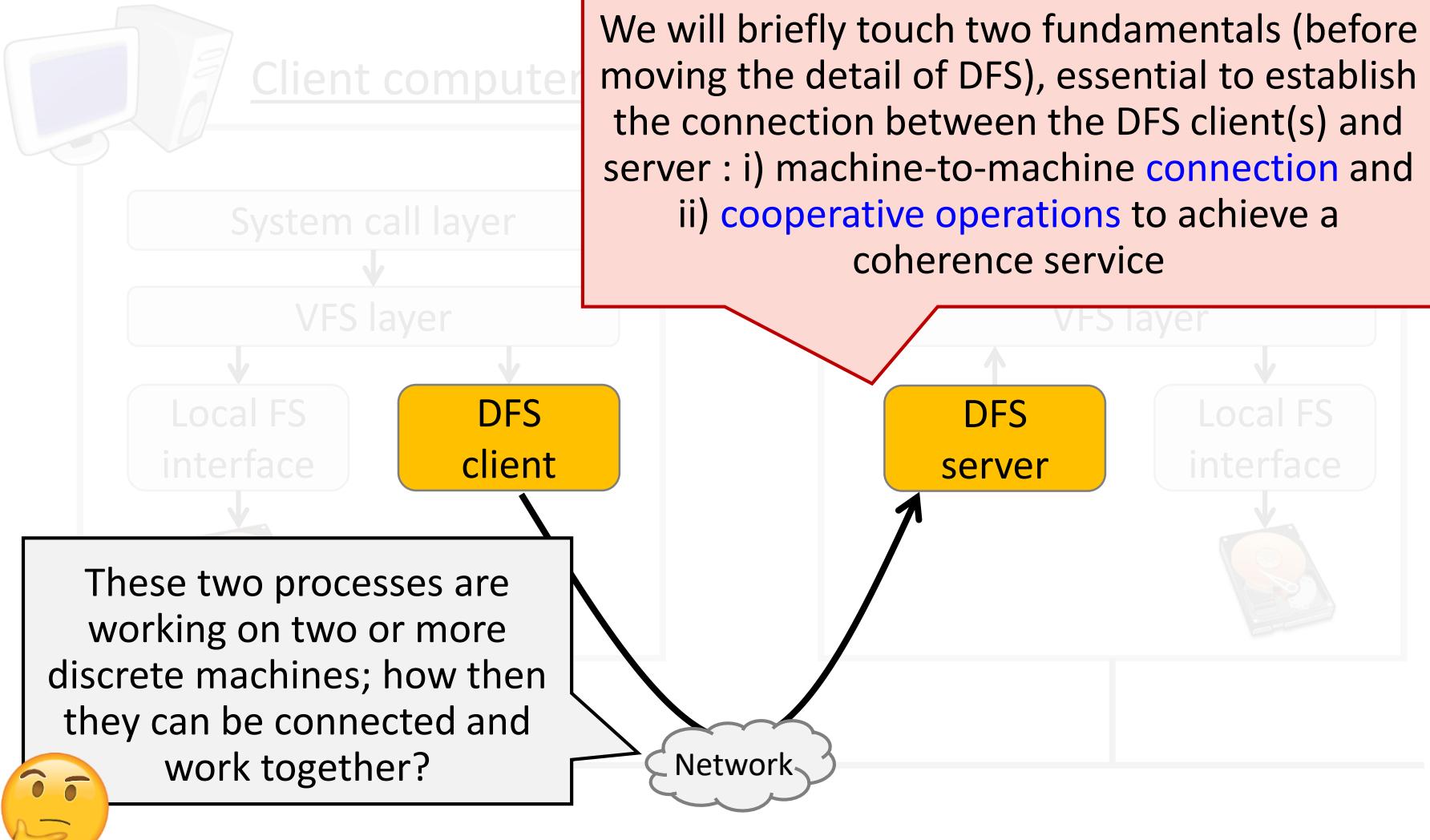
Distributed File-System Stack



Distributed File-System Stack



Distributed File-System Stack



Machine Communication



The Case Study: The Internet

For the machine-to-machine communication, Internet is a de facto protocol, which is widely also used in DFS



Internet protocol basically enables the machine-to-machine communication by handshaking a message among multiple nodes, which is referred to as “packet”



Various Supports

There are many different applications to be connected with

Application





Various Supports

Application



Transmission
Media



Coaxial cable



Fiber optic



Packet radio

The system also can have a wide spectrum of physical connection methods

Various Supports

Your DFS-enabled system may have any of these



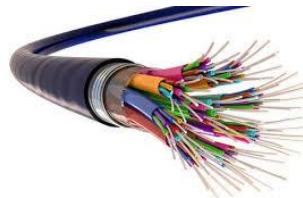
Application



Transmission
Media



Coaxial cable



Fiber optic



Packet radio



Various Supports



How can we cover these extreme diversities and connect them all together for the coherent services?



Application



Transmission
Media



Coaxial cable



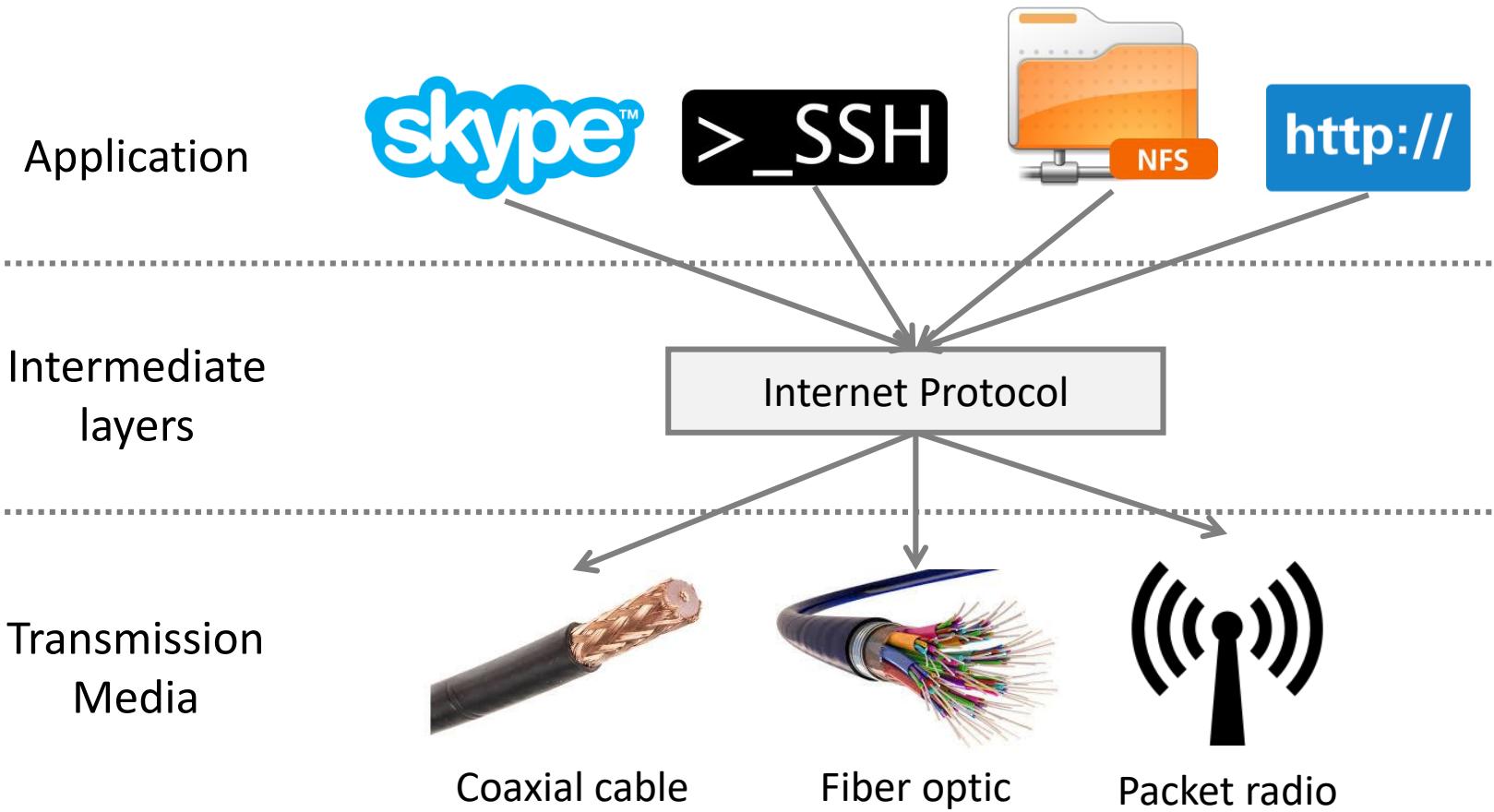
Fiber optic



Packet radio



Solution: Narrow Waist





Solution: Narrow Waist

Application



Intermediate
layers

Transmission
Media

At the very high-level viewpoint, we can hide such complexity and diversity on the network by introducing an intermediate component, **Internet**. It provides an **illusion** for the coherent network service across different technology domains.

Internet Protocol



Coaxial cable

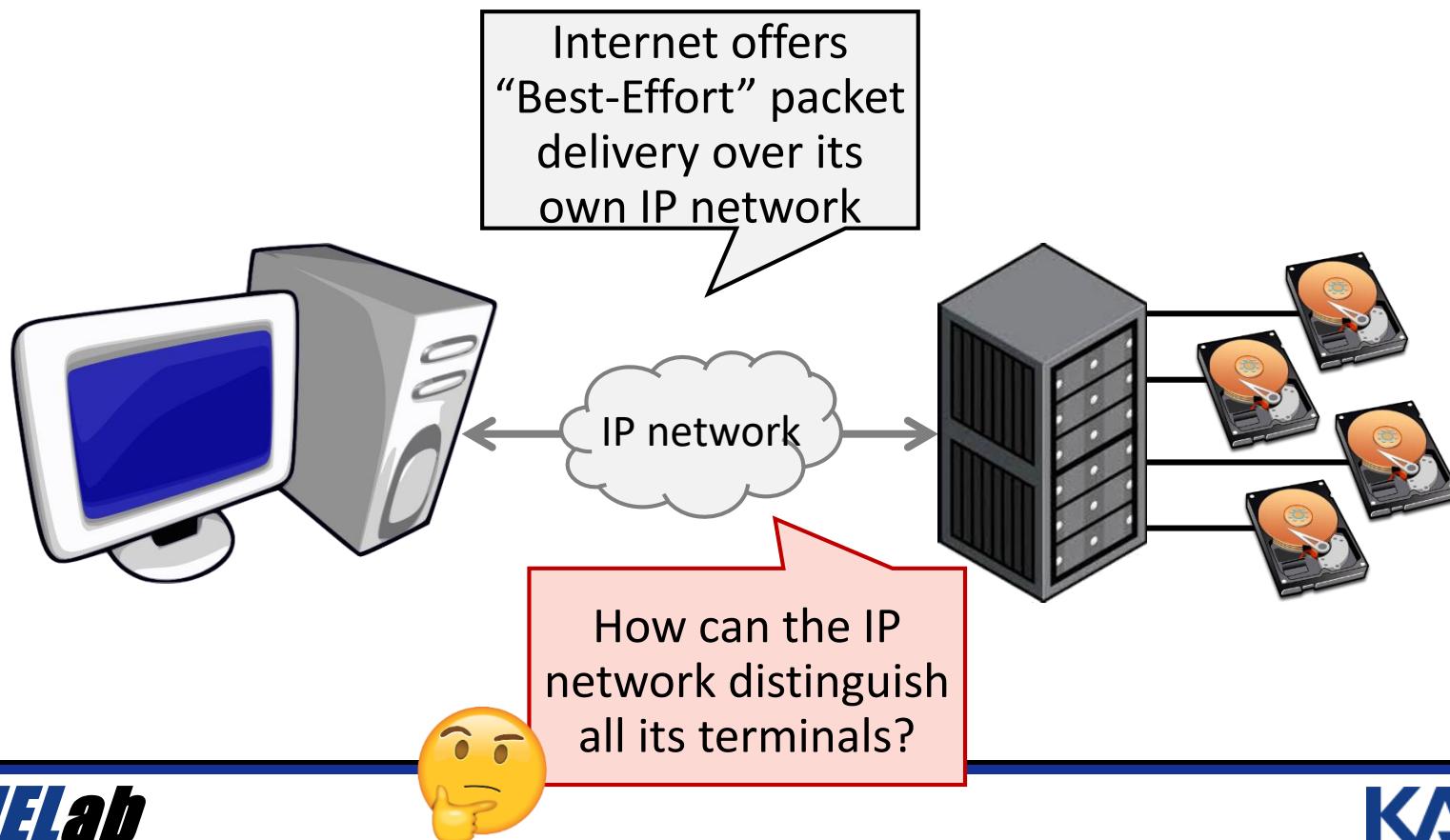


Fiber optic



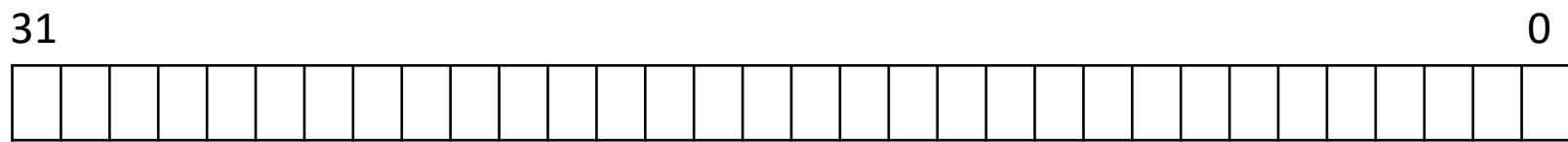
Packet radio

The Internet Protocol (IP)





IP Address (e.g., IPv4)



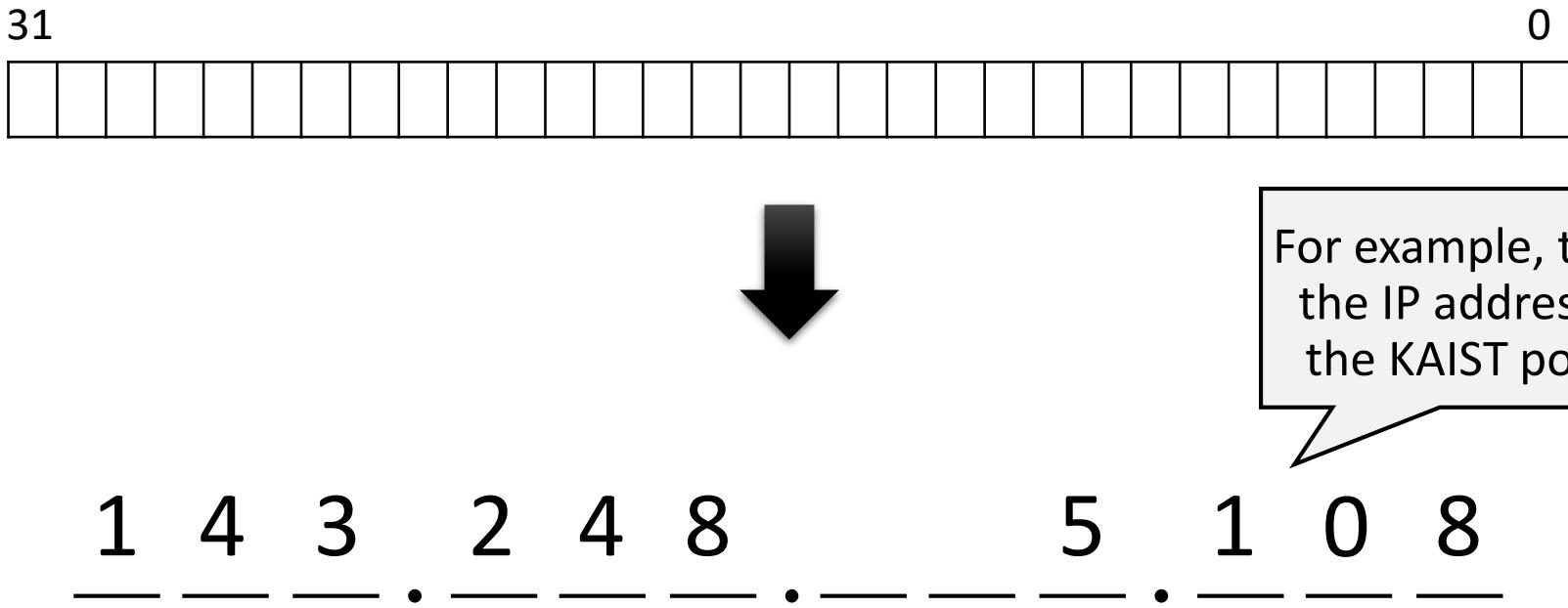
IP packet needs an address indicating where it should go; IPv4 uses a 32-bit integer for the address



— — — • — — — • — — — • — — —

The address is often written with four dot-separated integers

IP Address





IP Address

31

0



BTW, do you
remember all IP
address to access a
server?

1 4 3 . 2 4 8 . 5 1 0 8



Domain Name System

`https://portal.kaist.ac.kr`

No! an IP address is mapped to a **human-readable name** that you are familiar with

DNS is a hierarchical mechanism for the naming; the name divided in domains, (right to left)



Domain Name System

`https://portal.kaist.ac.kr`

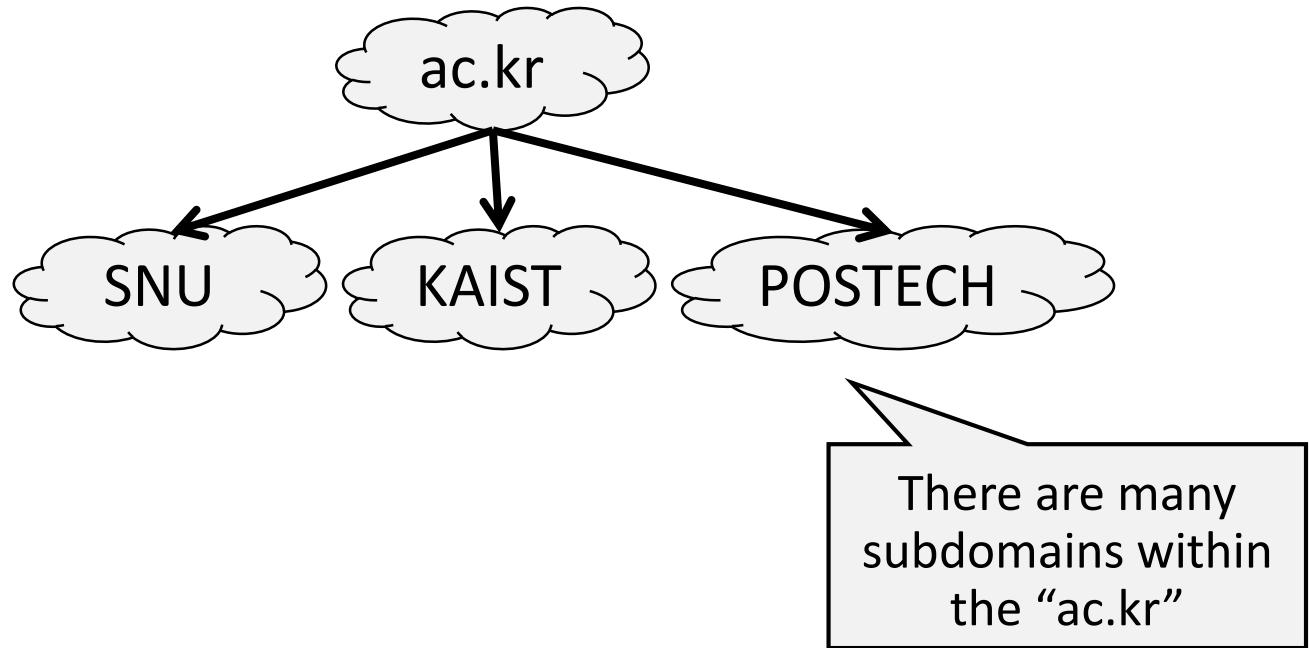


The top-level of the target network is “*ac.kr*”



Domain Name System

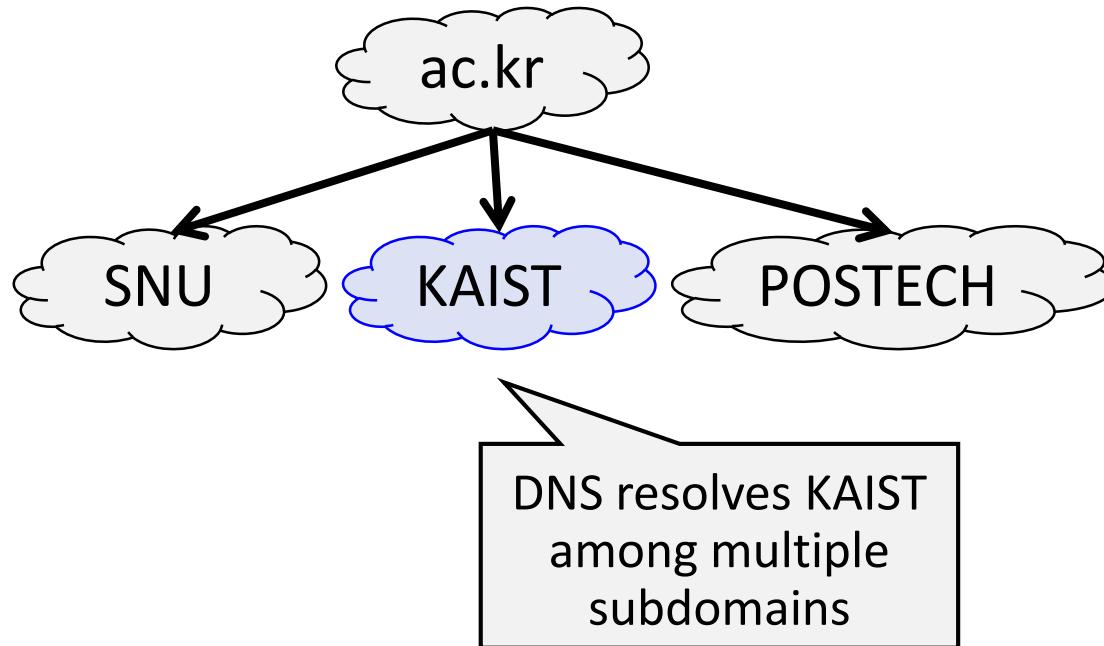
`https://portal.kaist.ac.kr`





Domain Name System

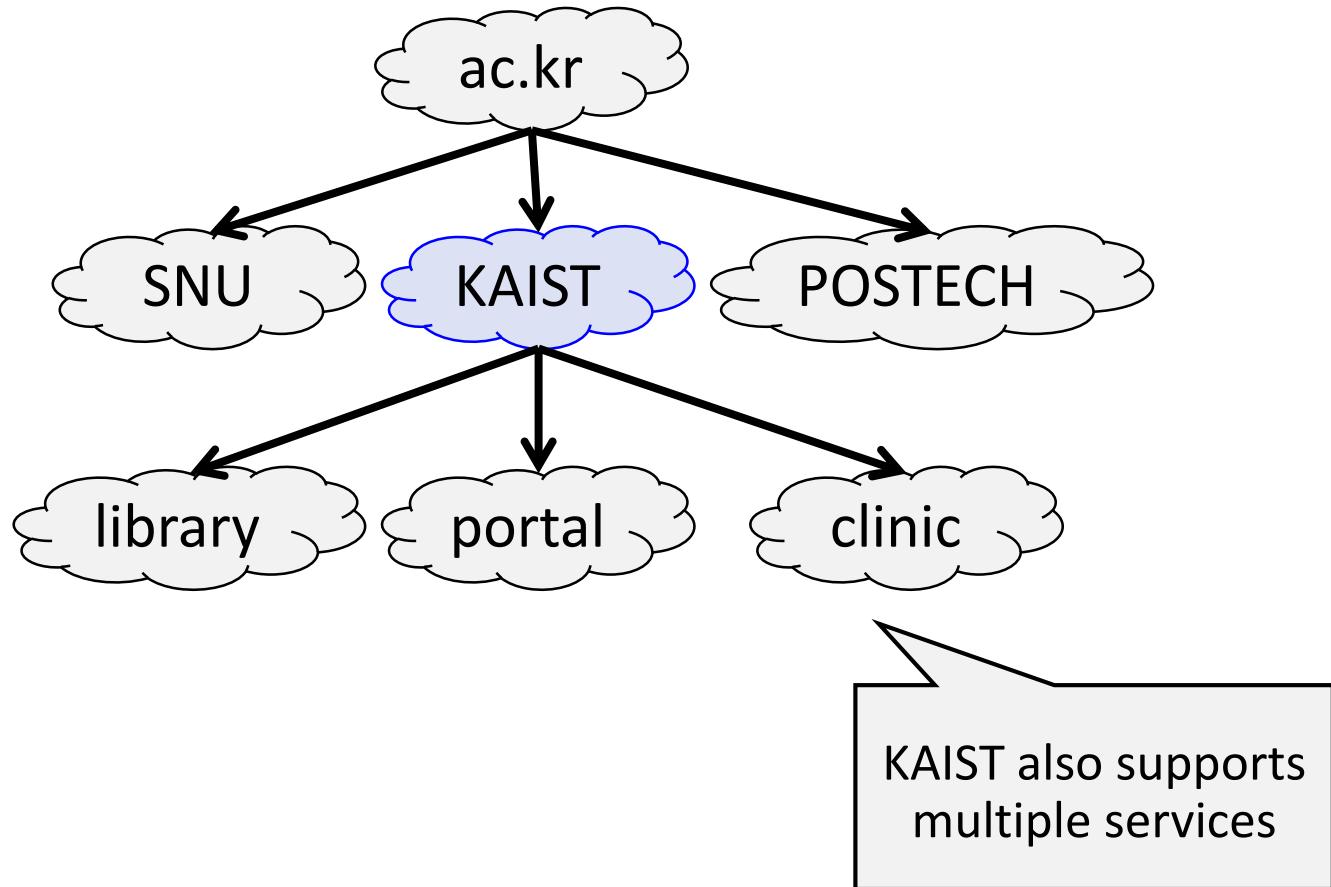
`https://portal.kaist.ac.kr`





Domain Name System

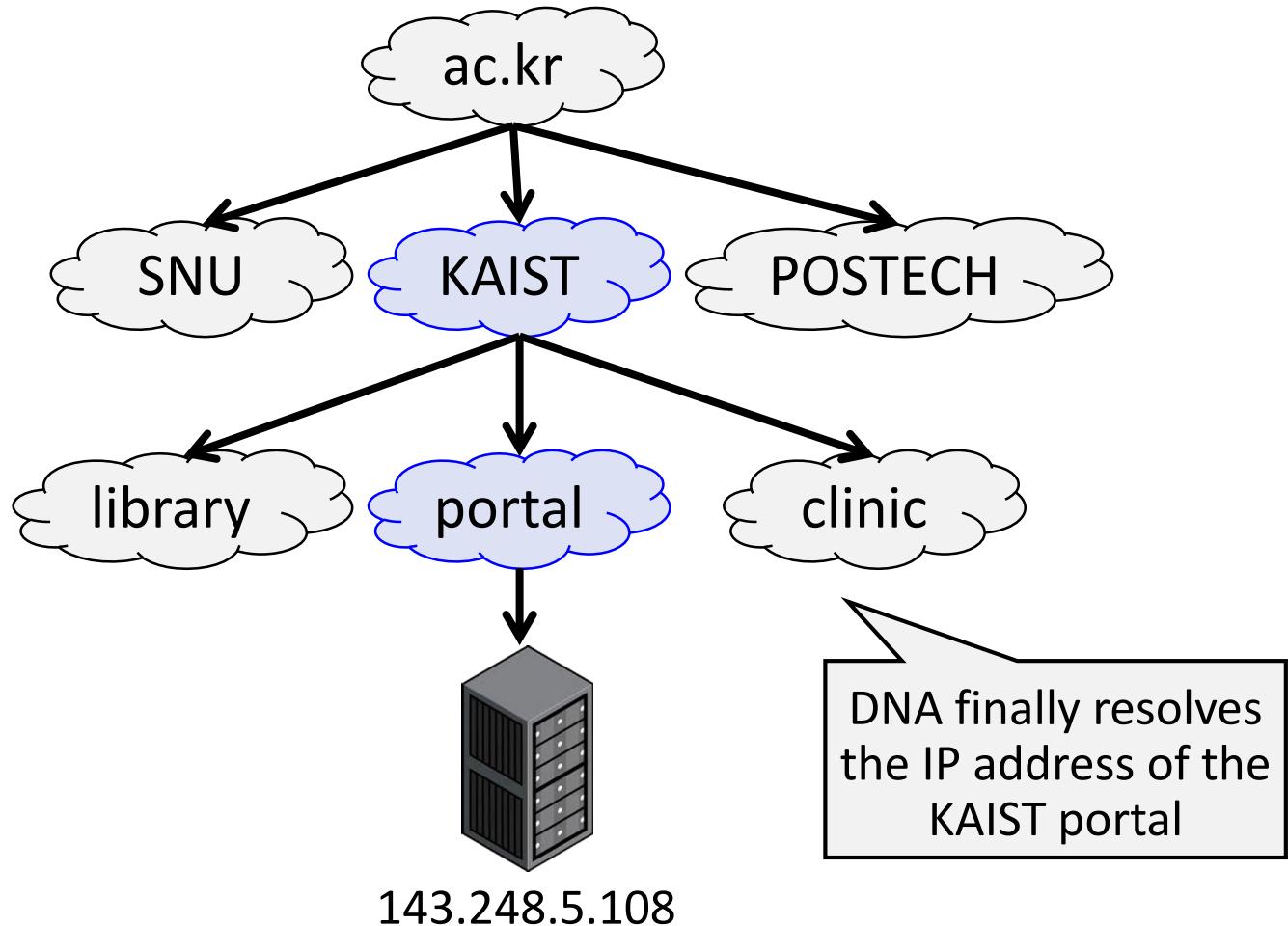
`https://portal.kaist.ac.kr`





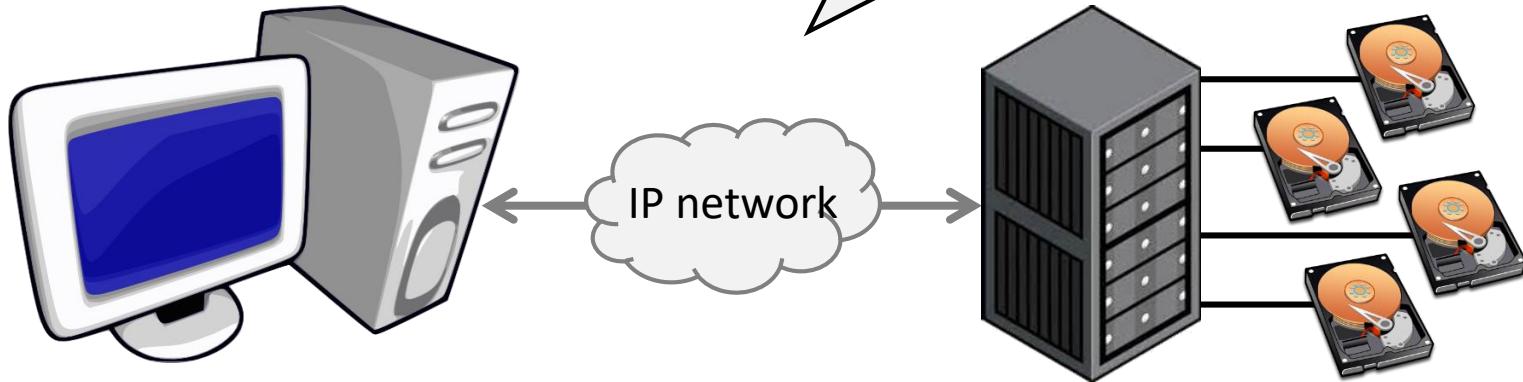
Domain Name System

`https://portal.kaist.ac.kr`



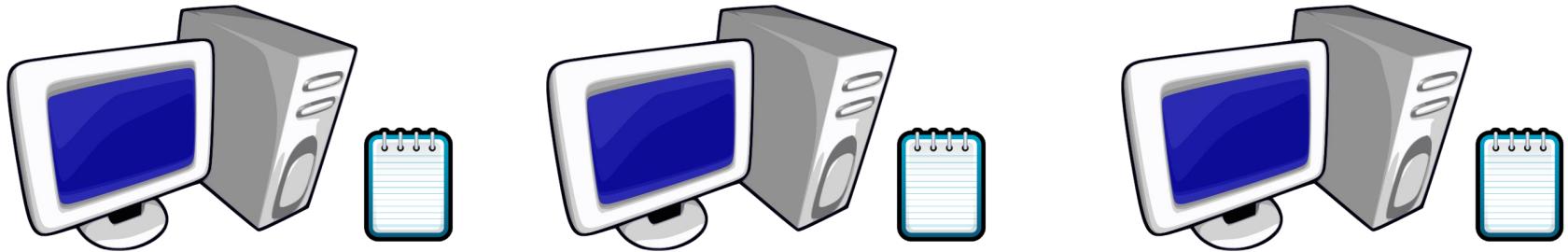
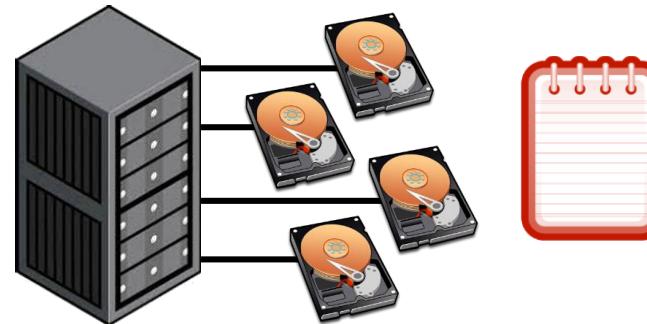
Machine Coordination

The machines can communicate each other via internet protocol, but how can we make all of them **consensus** for an event (in order to make the service **coherent**)

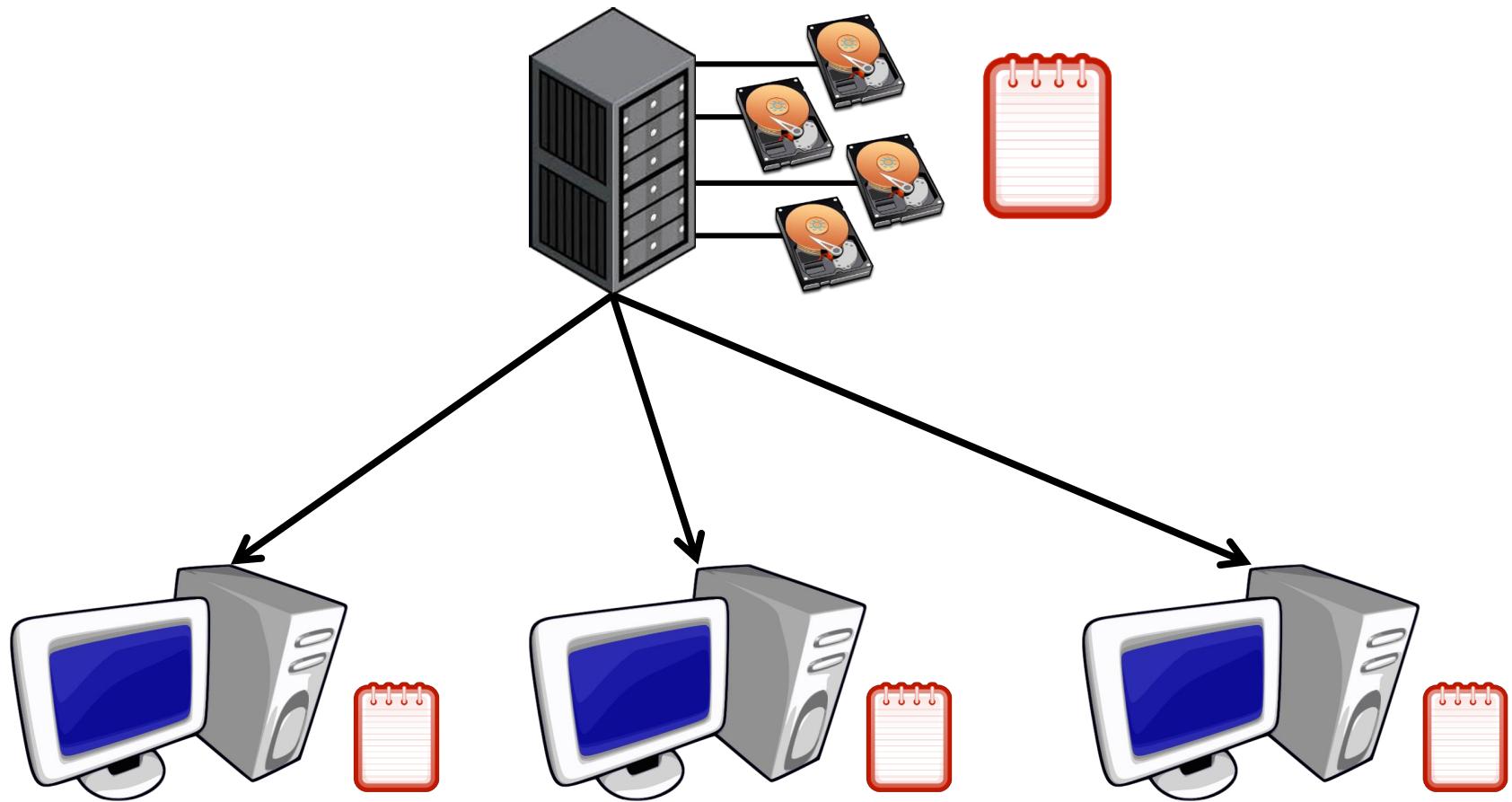


Coordination

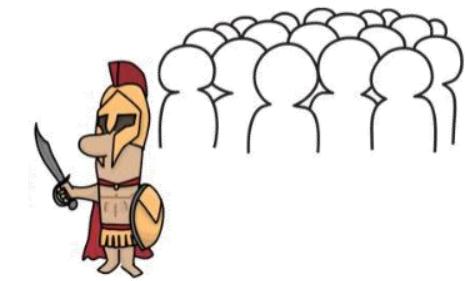
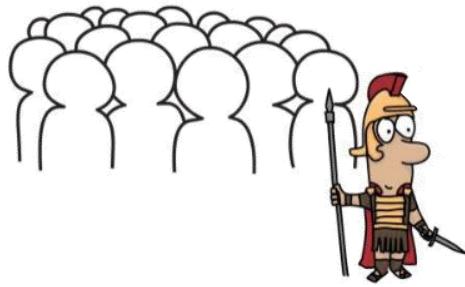
If a file is updated, the server needs to let all clients know the update



Coordination



Two Generals' Problem

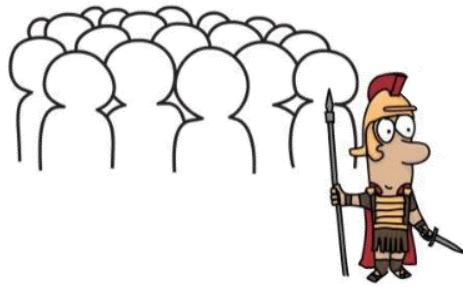


 Can messages over an **unreliable network** be used to guarantee two entities do something simultaneously?

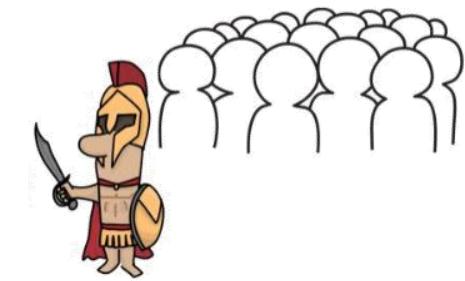
Two Generals' problem, also known as the Two Armies Problem, is a classic computer science and computer communication thought experiment



The Story of The Two Generals



Let's imagine two armies,
led by two generals,
planning an attack on a
common enemy





The Story of The Two Generals



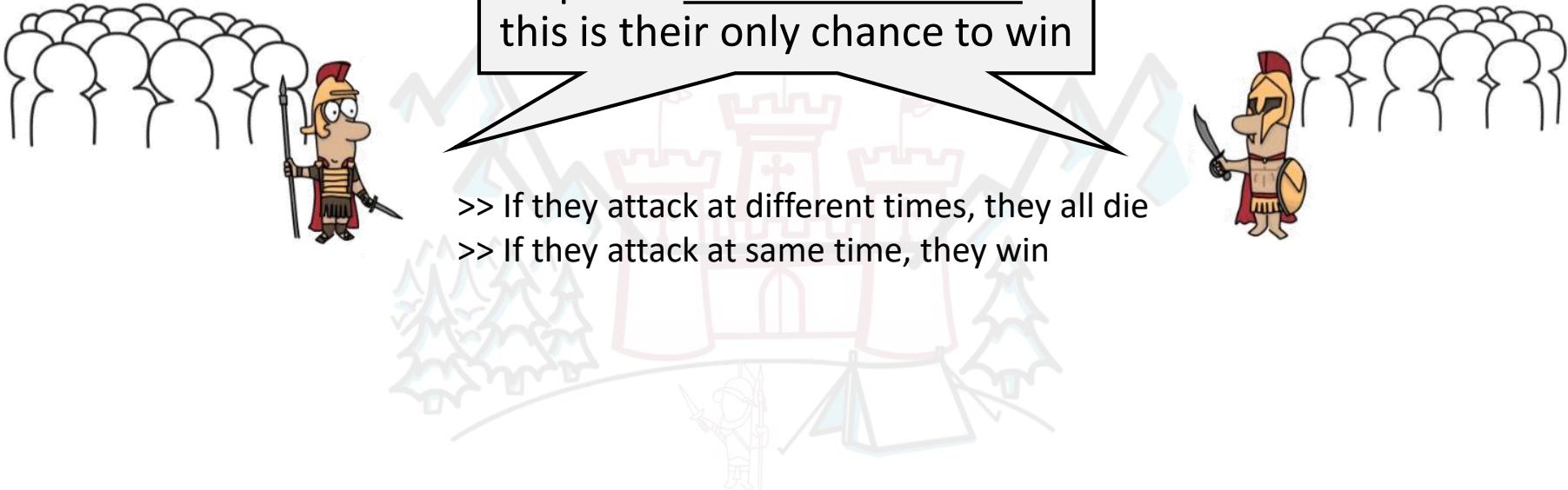
The enemy's city is in a valley and has a strong defense that can easily fight off a single army



The Story of The Two Generals

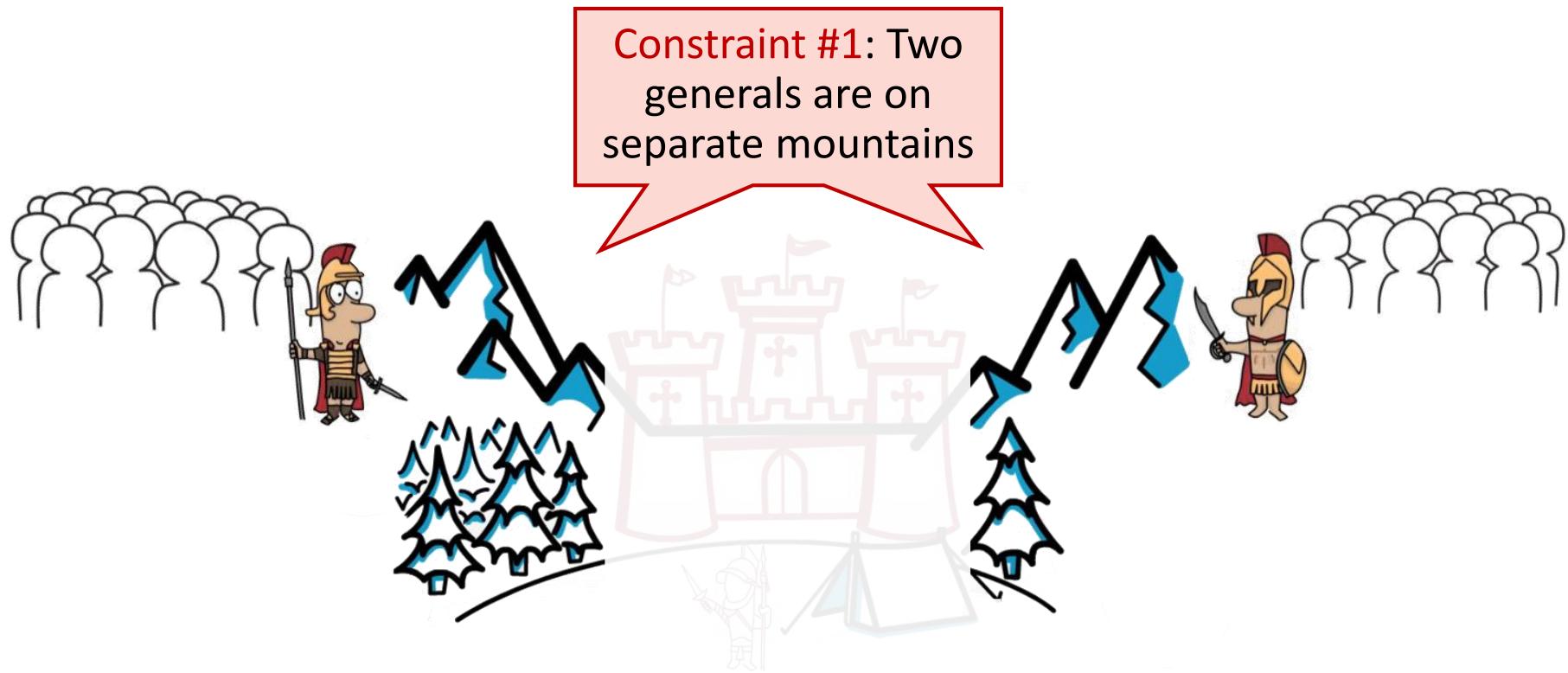
“Two Generals’ Problem”

Thus, two generals have to communicate with each other to plan a coordinate attack as this is their only chance to win

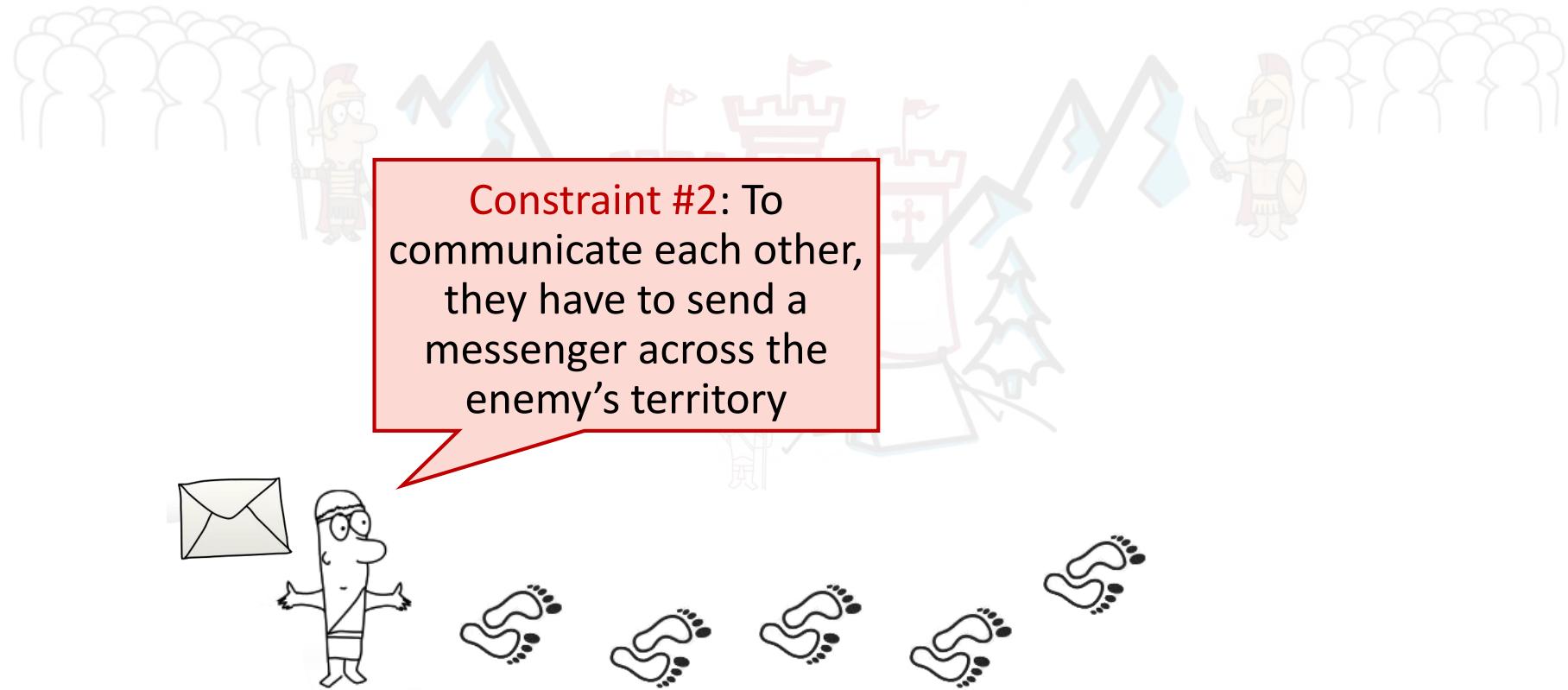


There are several **constraints** of Two General’s Problem that we have to put into as well

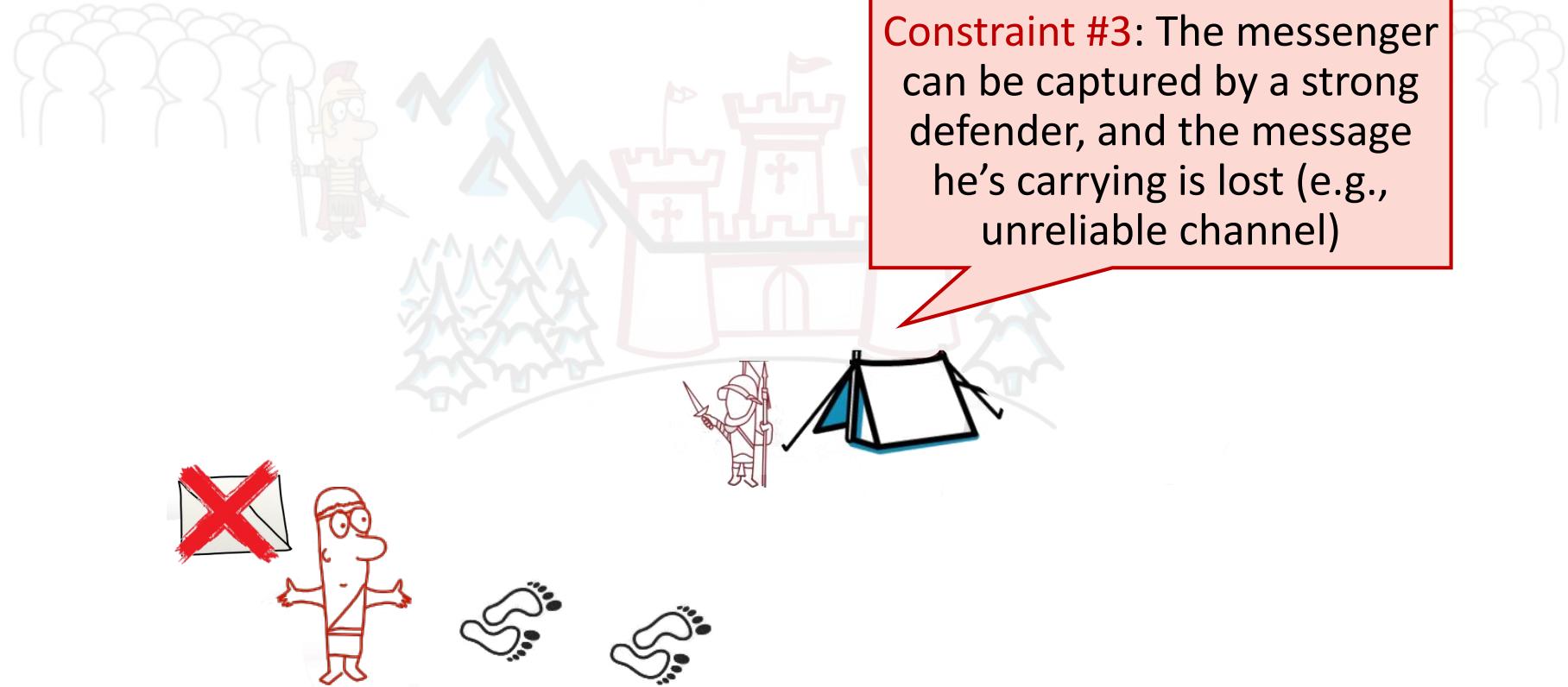
► Constraints of Two Generals' Problem



► Constraints of Two Generals' Problem

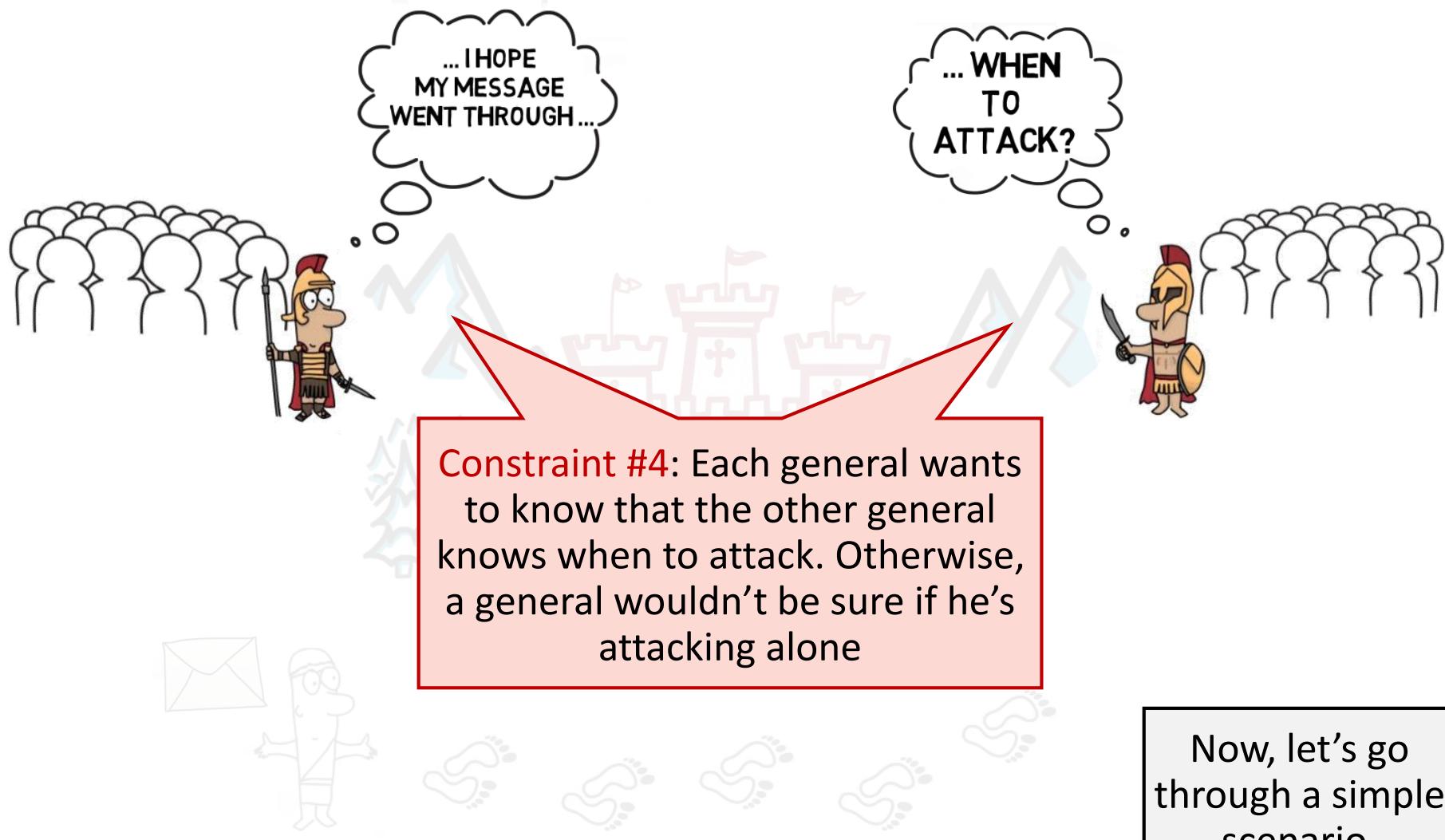


► Constraints of Two Generals' Problem





Constraints of Two Generals' Problem





Simple Scenario #1

A

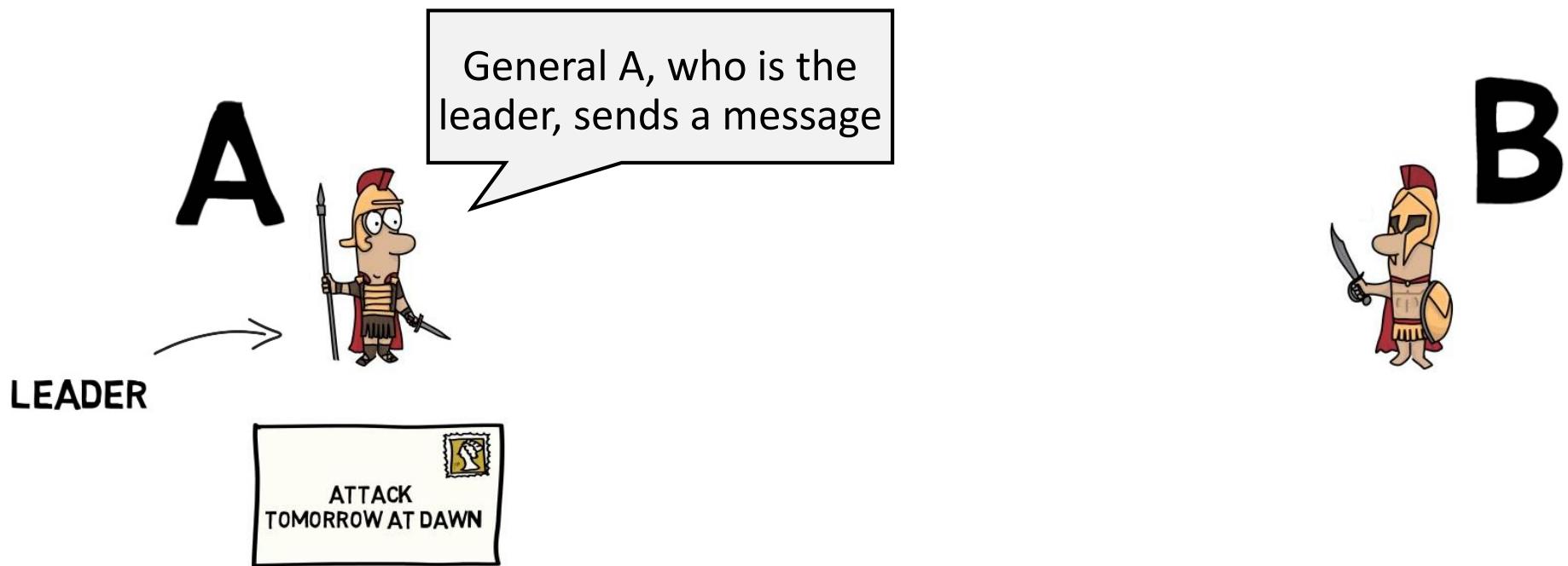


Let's call our generals A and B,
and let's assume everything goes
perfectly fine, saying that "**no
steal by a defender**"

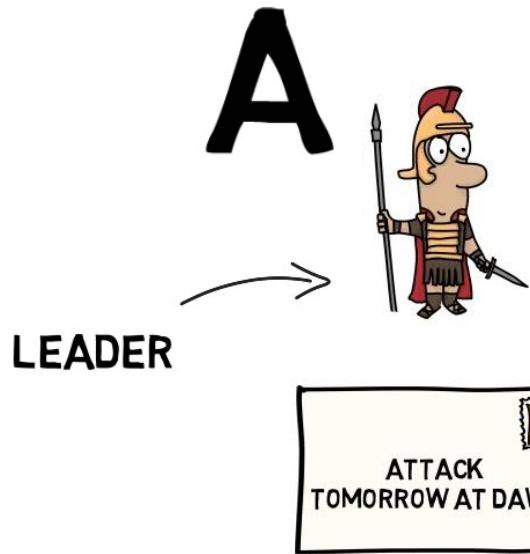
B



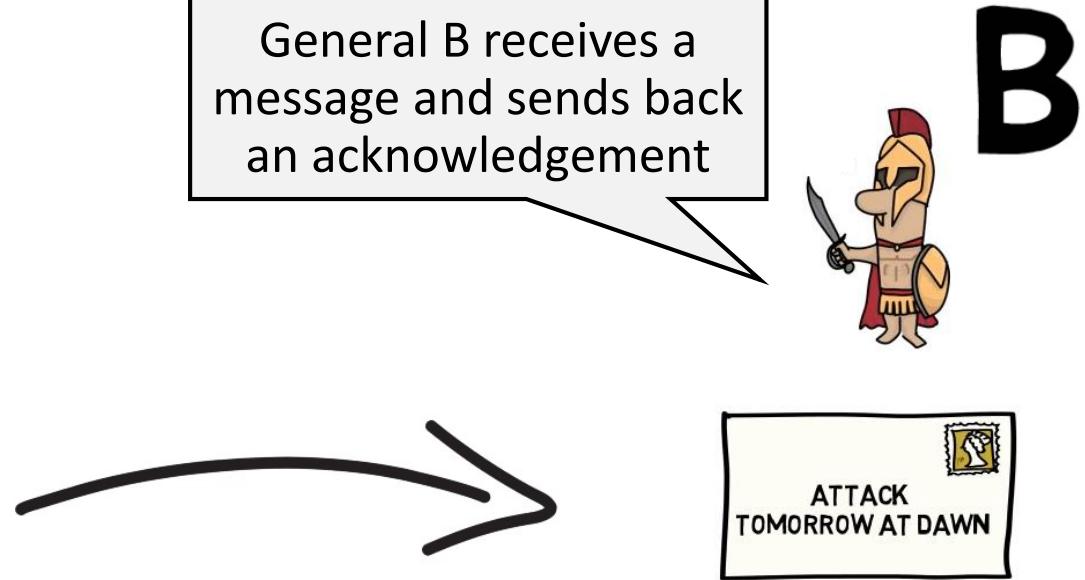
Simple Scenario #1



Simple Scenario #1



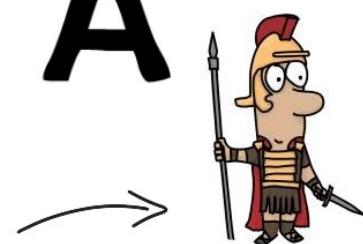
General B receives a message and sends back an acknowledgement





Simple Scenario #1

A



LEADER



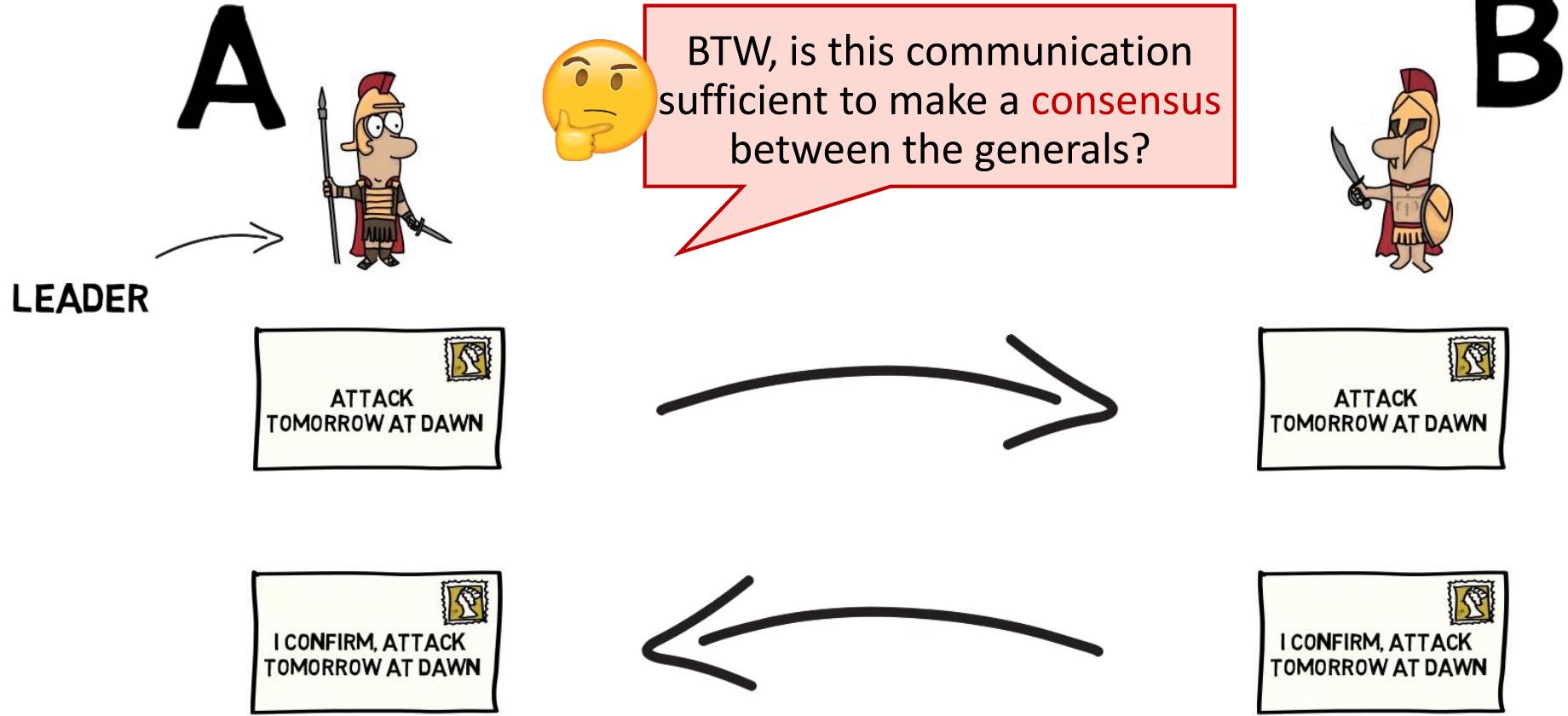
General B receives a message and sends back an acknowledgement

B

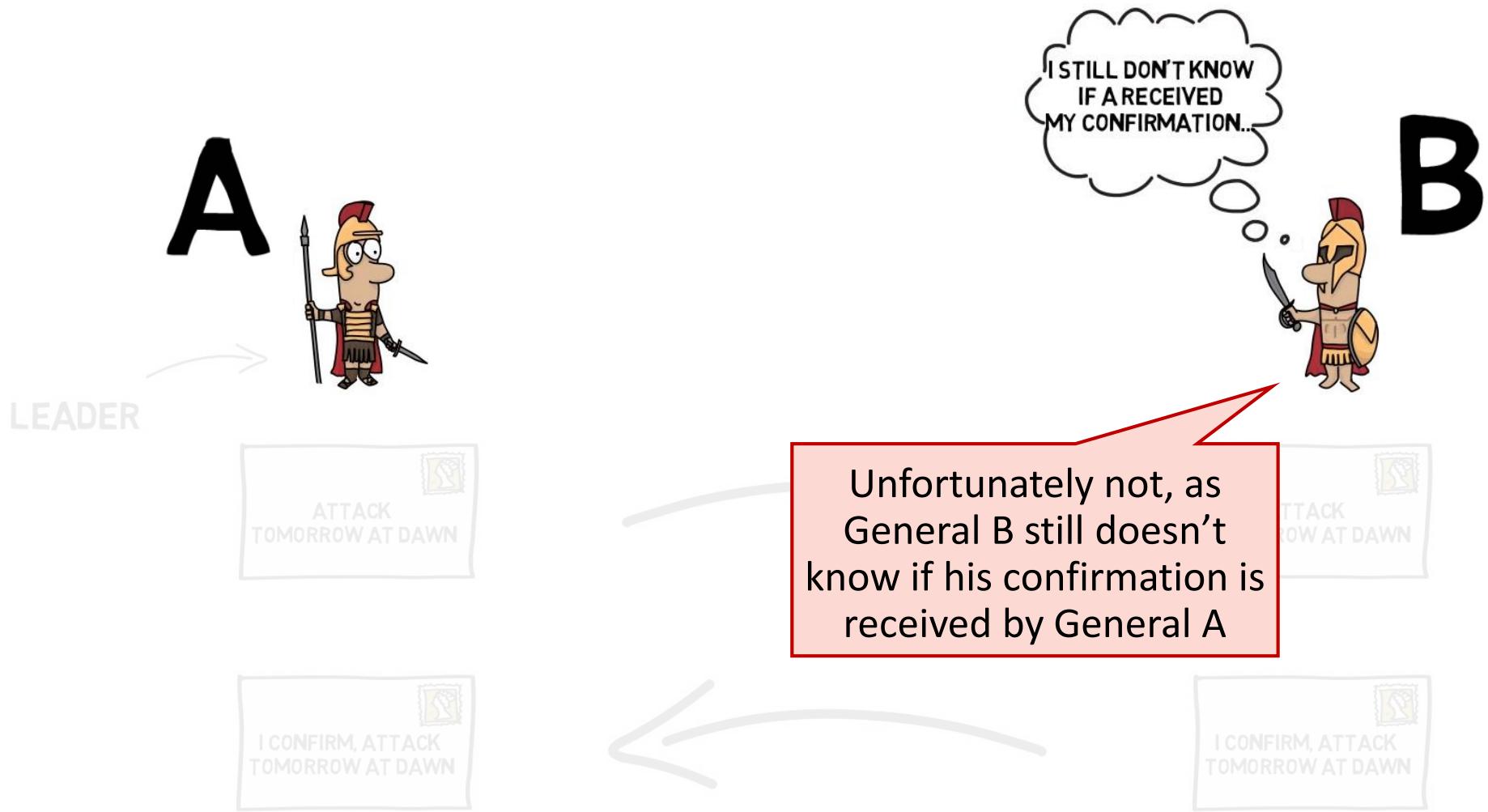




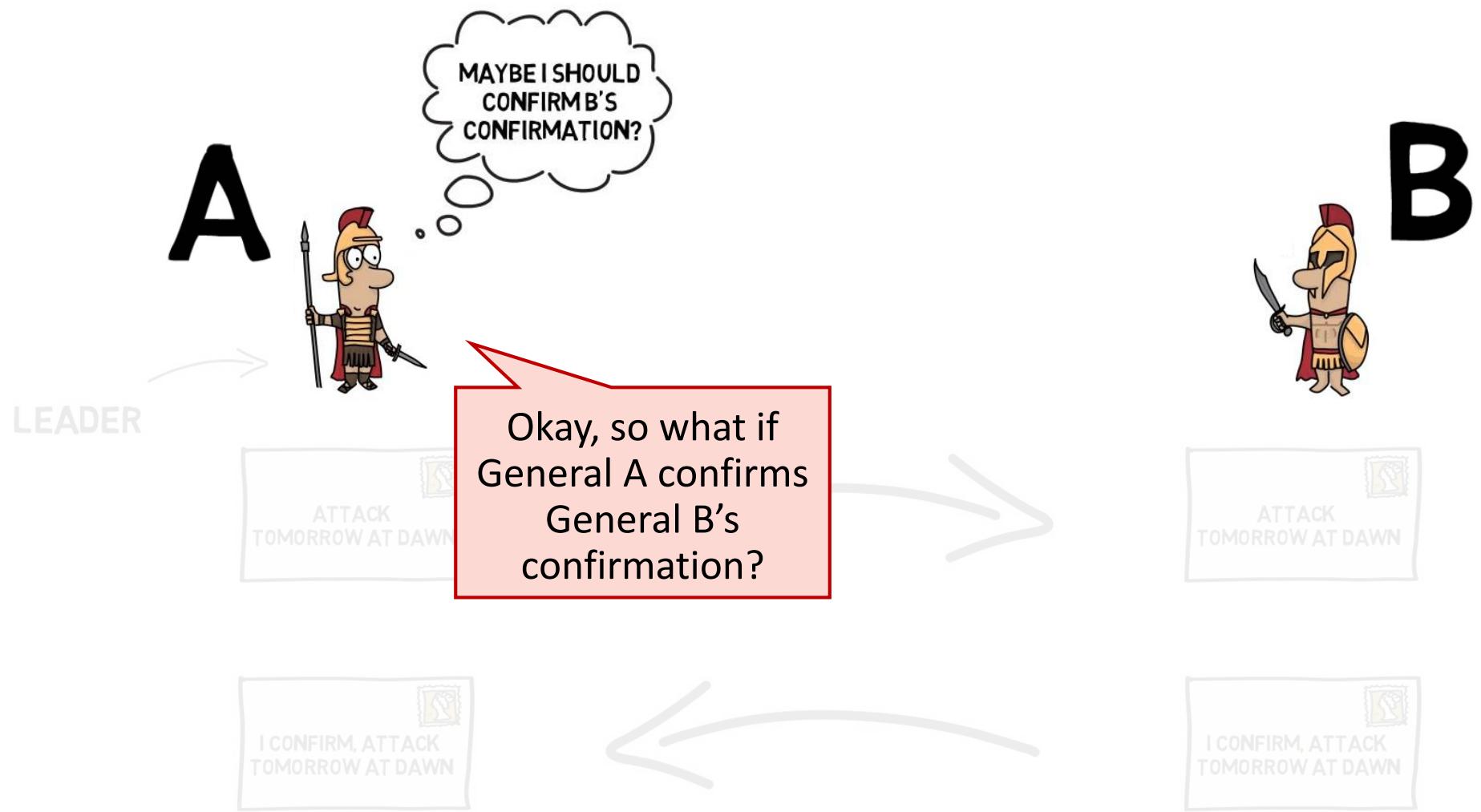
Simple Scenario #1



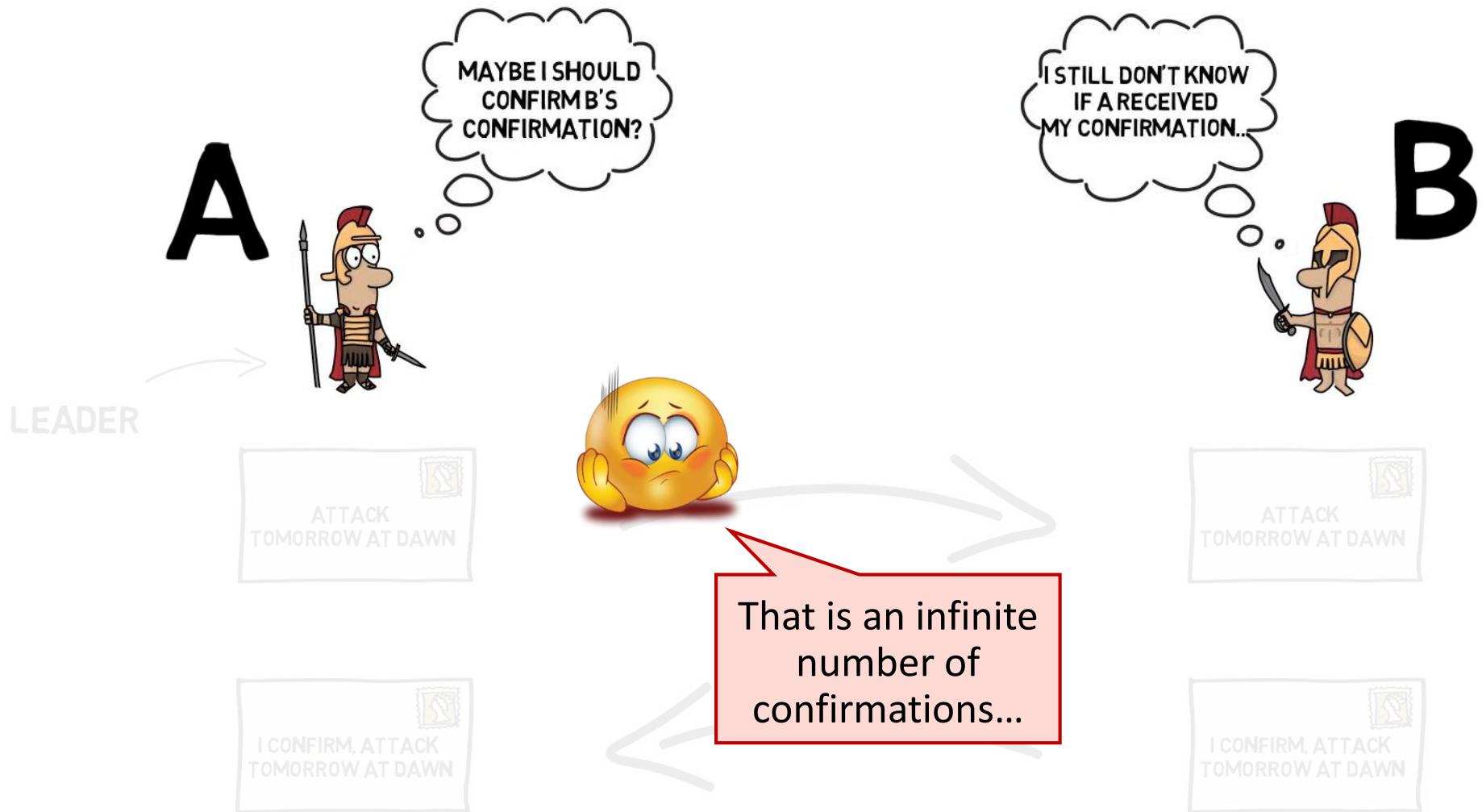
Simple Scenario #1



Simple Scenario #1



Simple Scenario #1





Simple Scenario #2

A



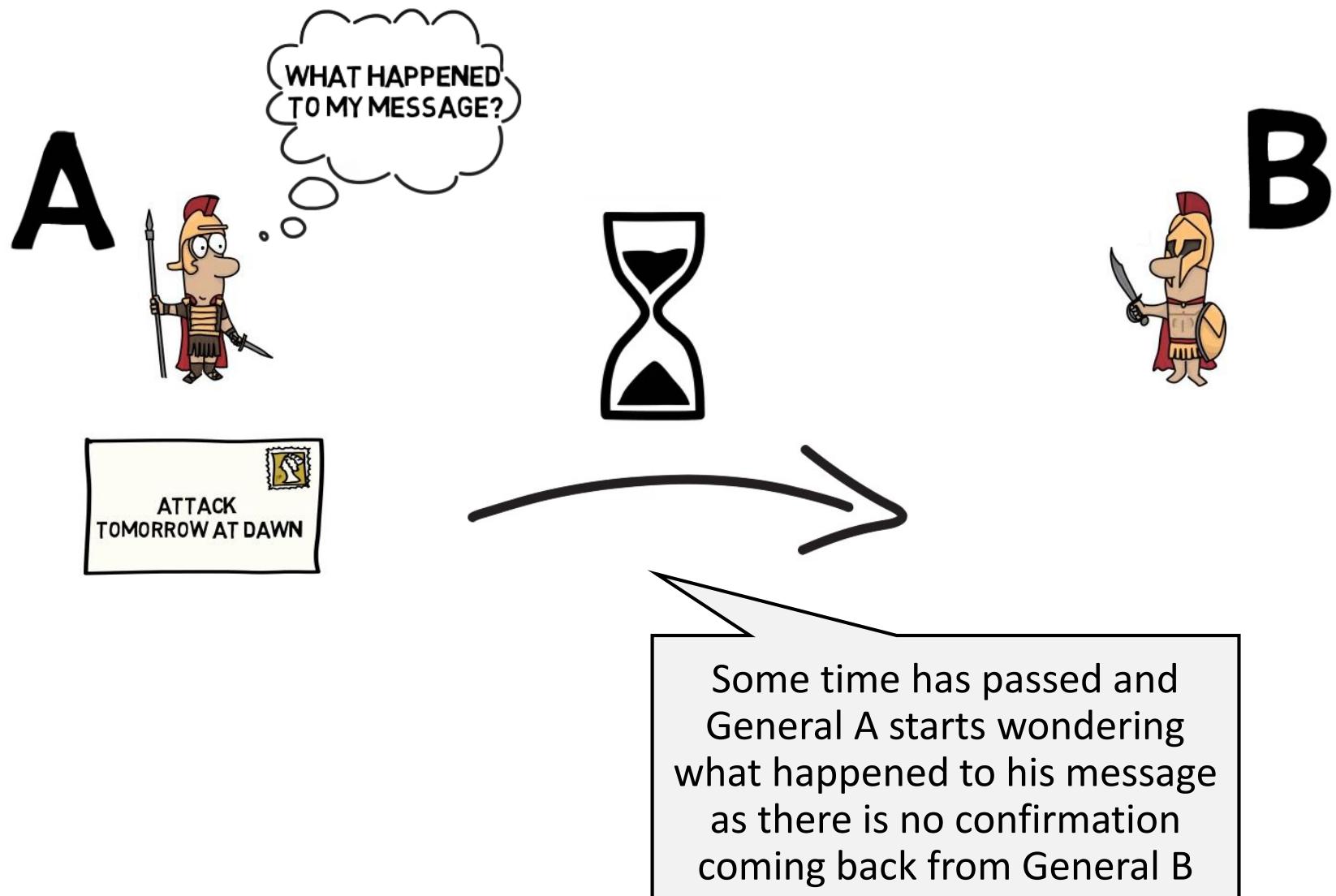
In the second scenario,
let's assume that
General A sends a
message to General B

B





Simple Scenario #2





Simple Scenario #2

A



There are two possibilities here

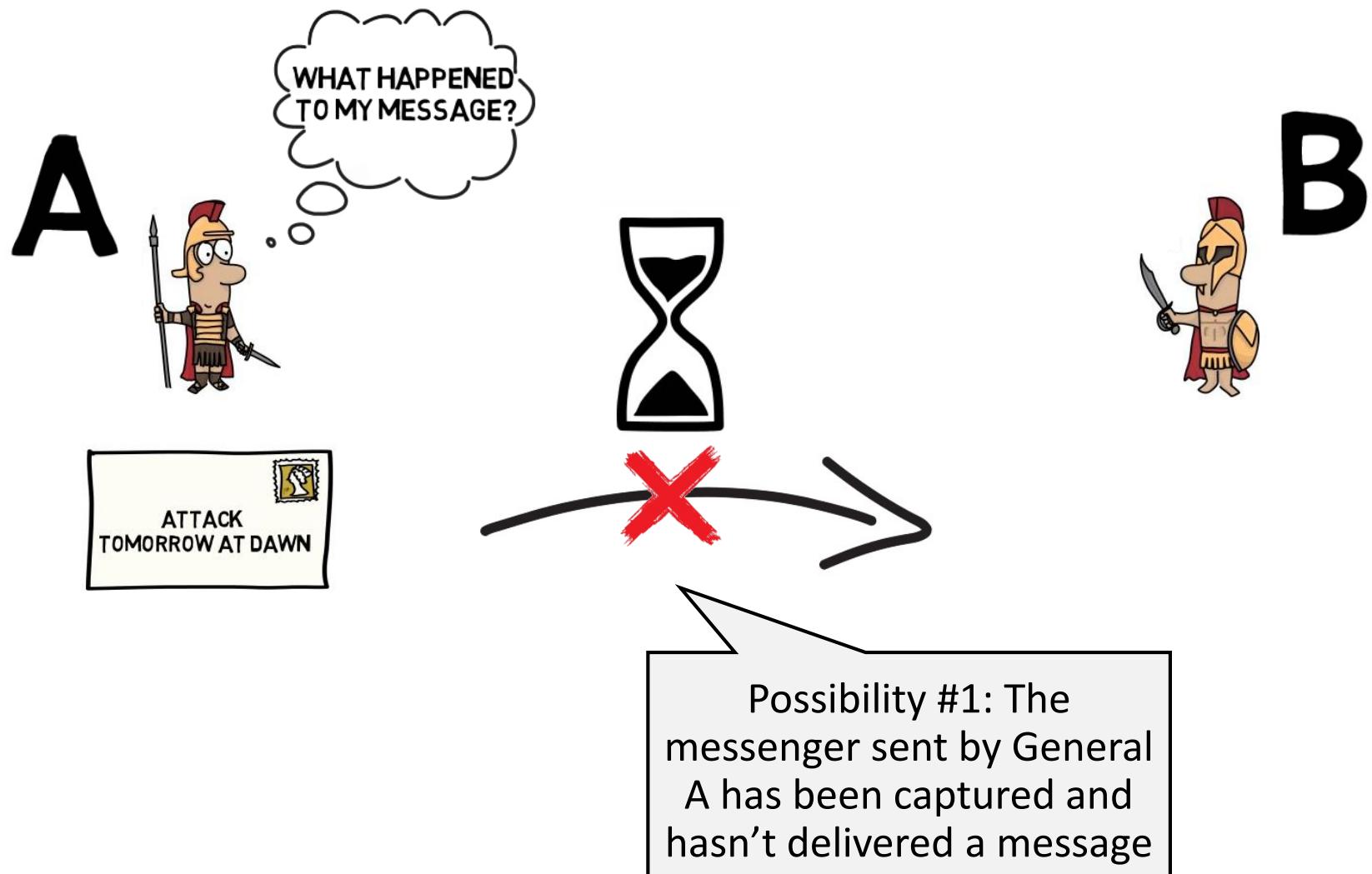


B

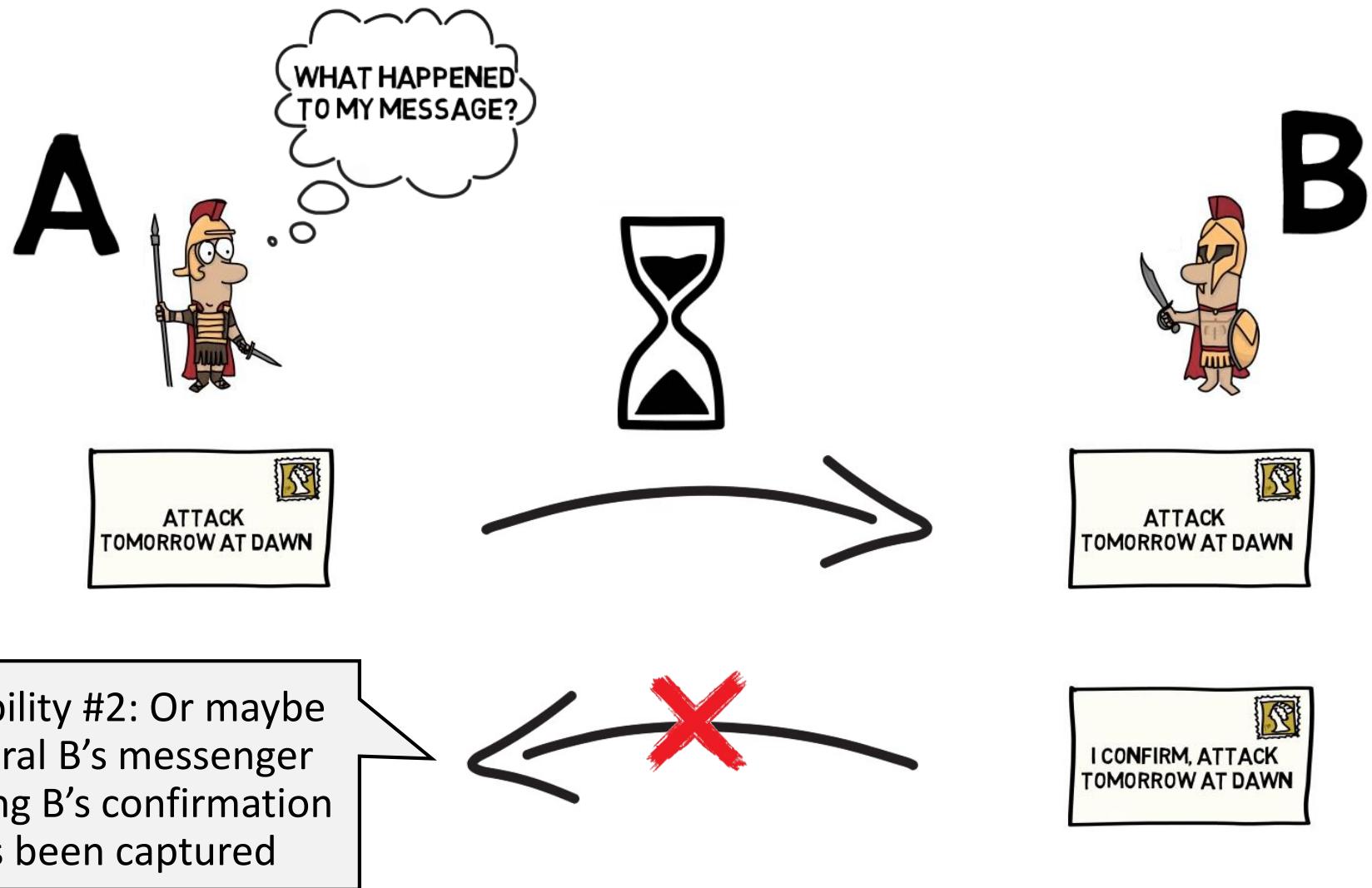




Simple Scenario #2



Simple Scenario #2



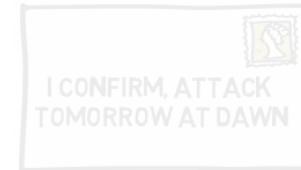
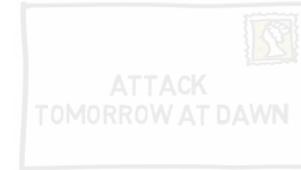
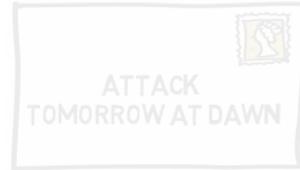


Simple Scenario #2

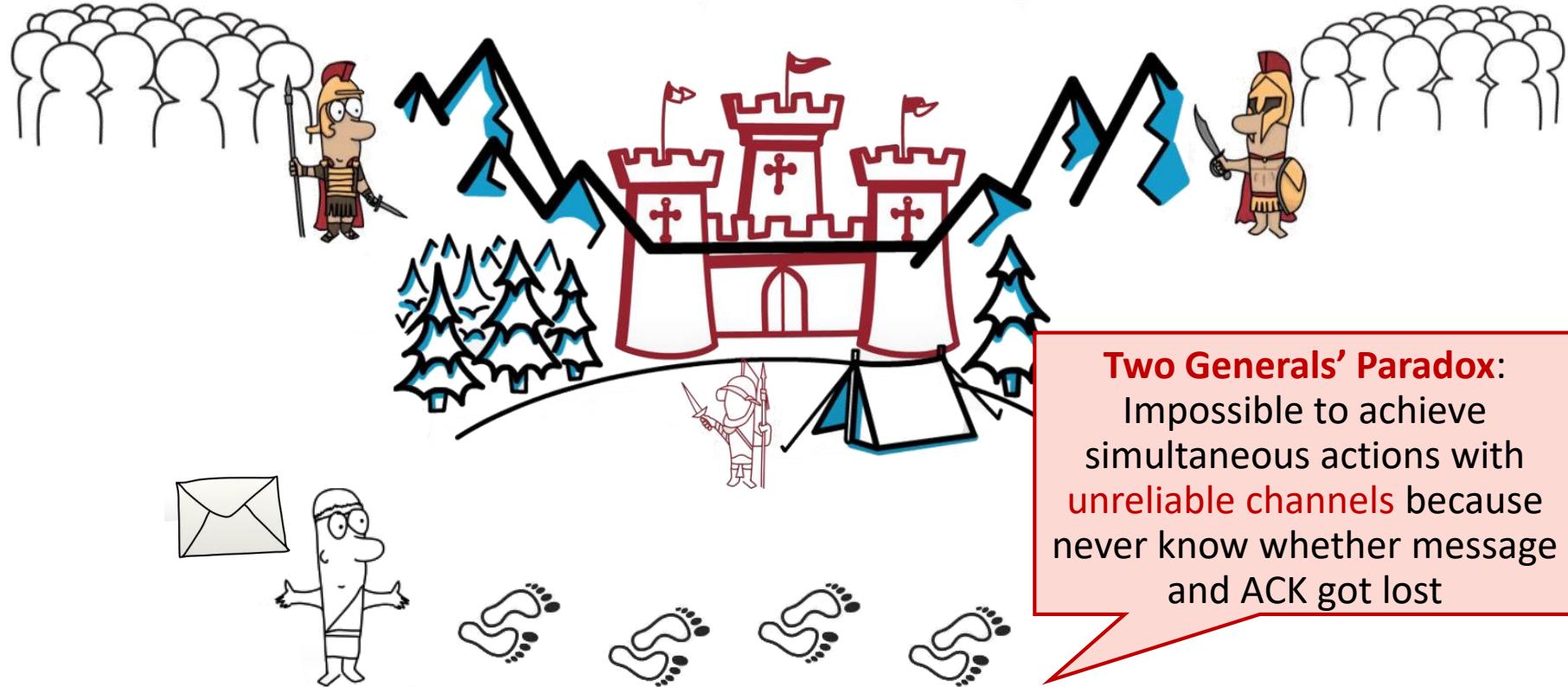


In both possibilities, they cannot come to a **consensus** again as A is not able to tell if his message was lost or if it was B's confirmation that didn't get through

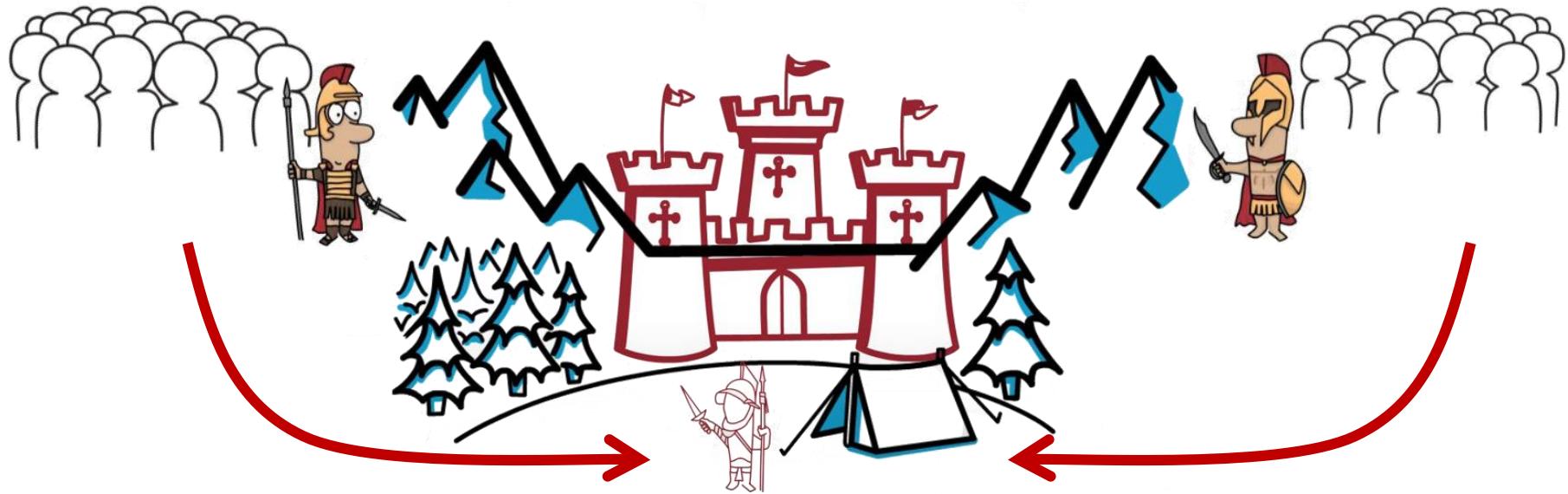
Again, we ended up in an **inconsistent state**, which would result in either General A or B attacking by himself



Two Generals' Paradox

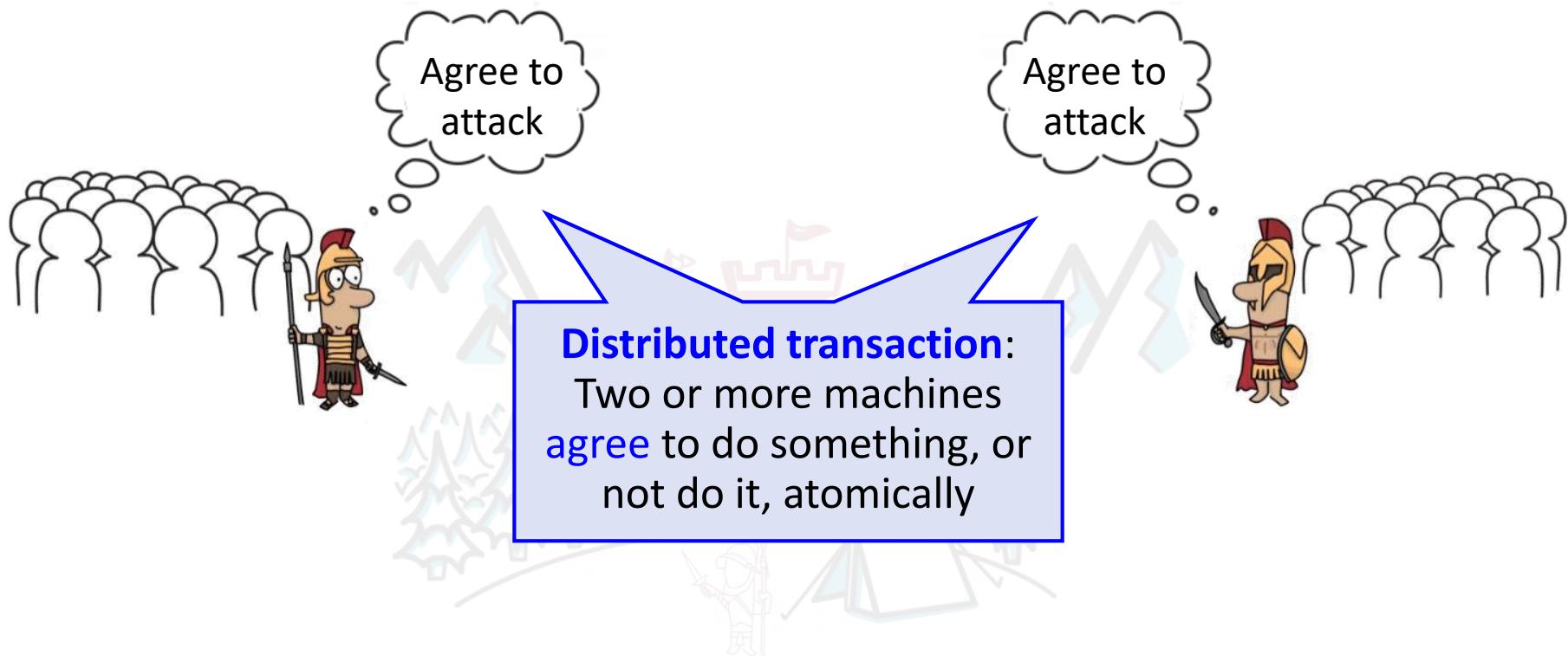


Distributed Transactions

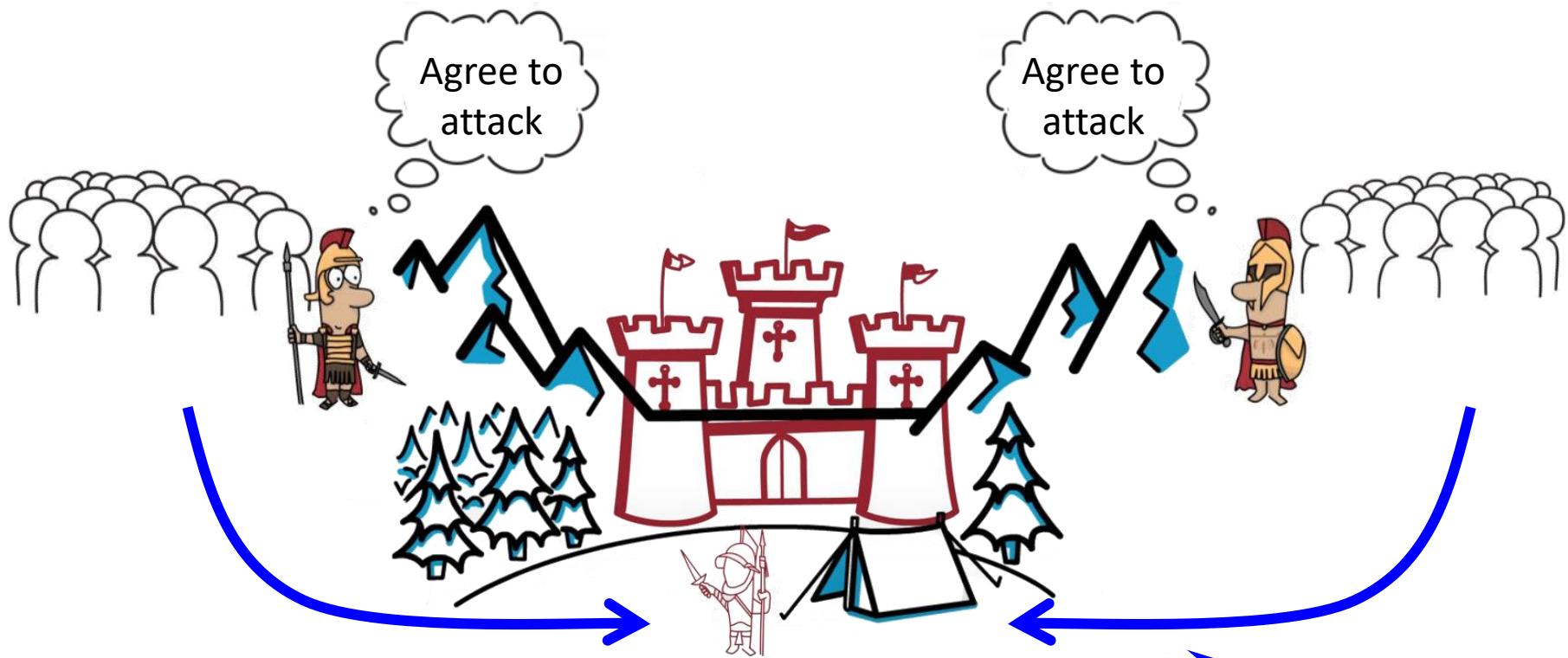


Since we can't solve the General's Paradox (i.e., simultaneous attack), let's solve a **related problem**

Distributed Transactions



Distributed Transactions

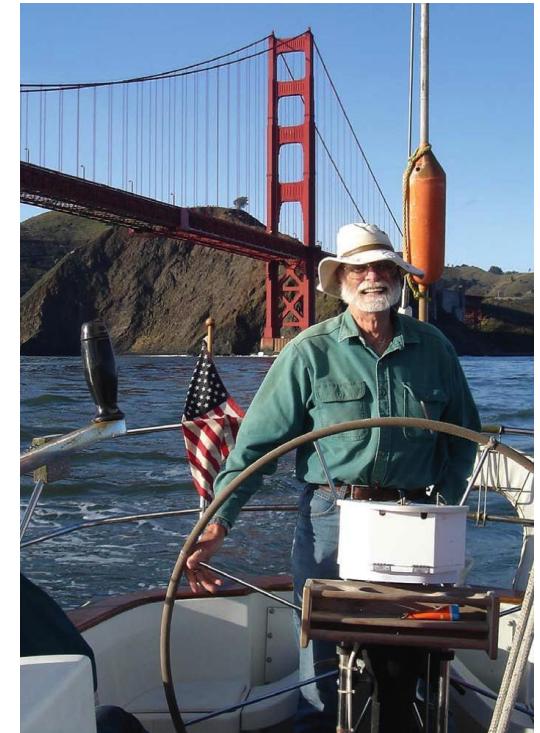


In other words, there is no constraints on time; just that it will eventually happen

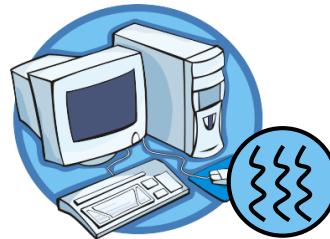
Coordination Protocols: Two-Phase Commit

Two-Phase Commit protocol
(a.k.a., 2PC) is developed by
Turing award winner Jim Gray
for distributed transaction

2PC is used in most modern
distributed systems.
Representative of other
coordination protocols



2PC Terminology

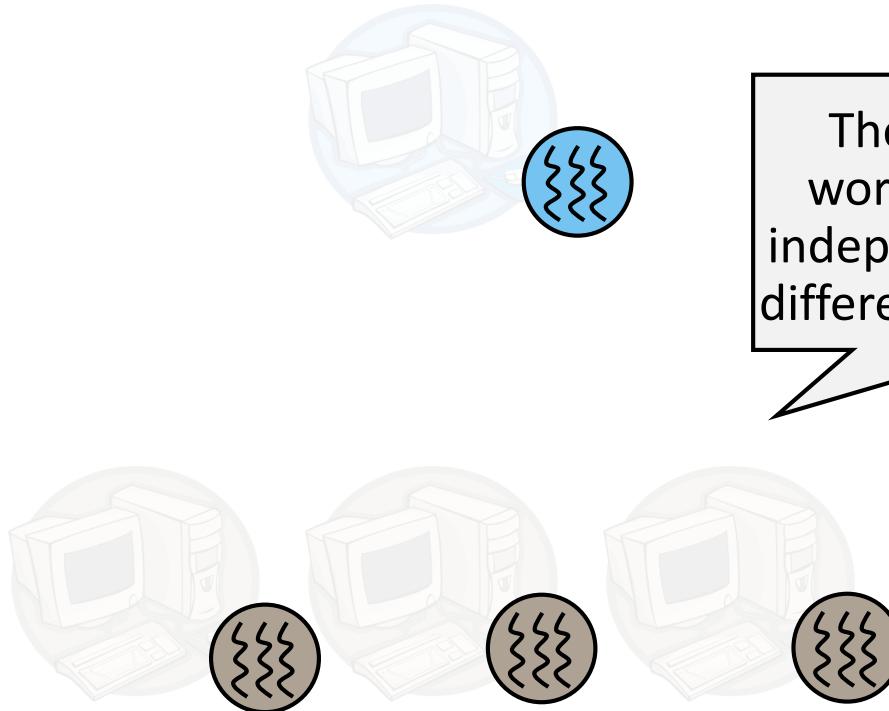


There is one
coordinator



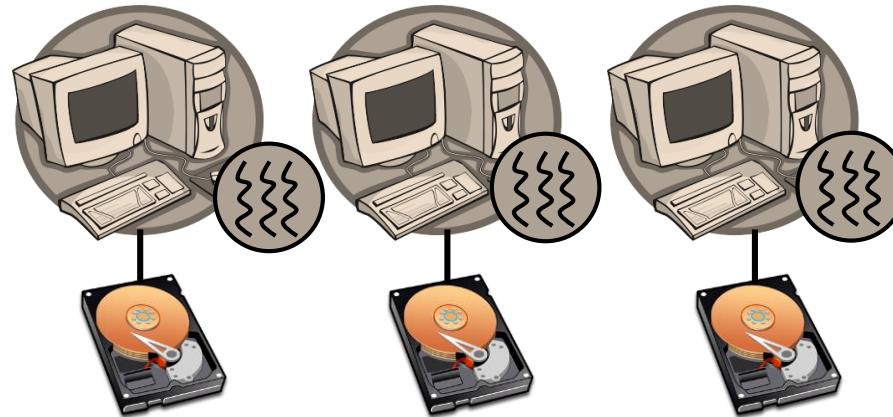
There is a set
of *workers*

2PC Terminology



2PC Terminology

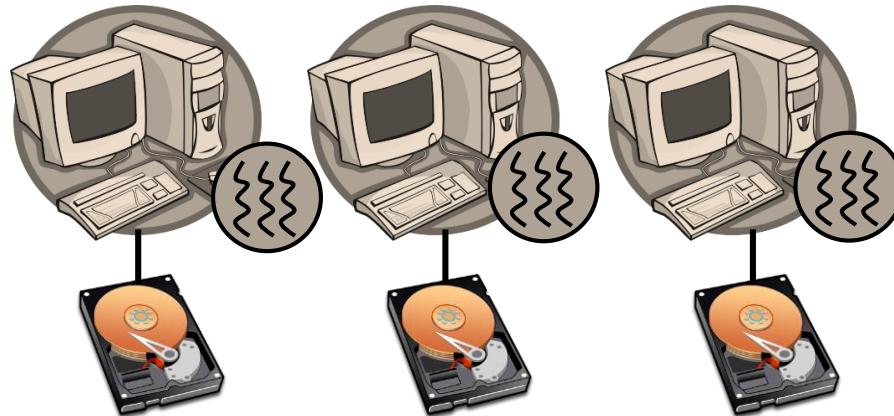
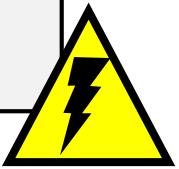
All nodes use their own stable (local) storage to keep the status of coordinator/workers
(persistent log)



2PC Terminology

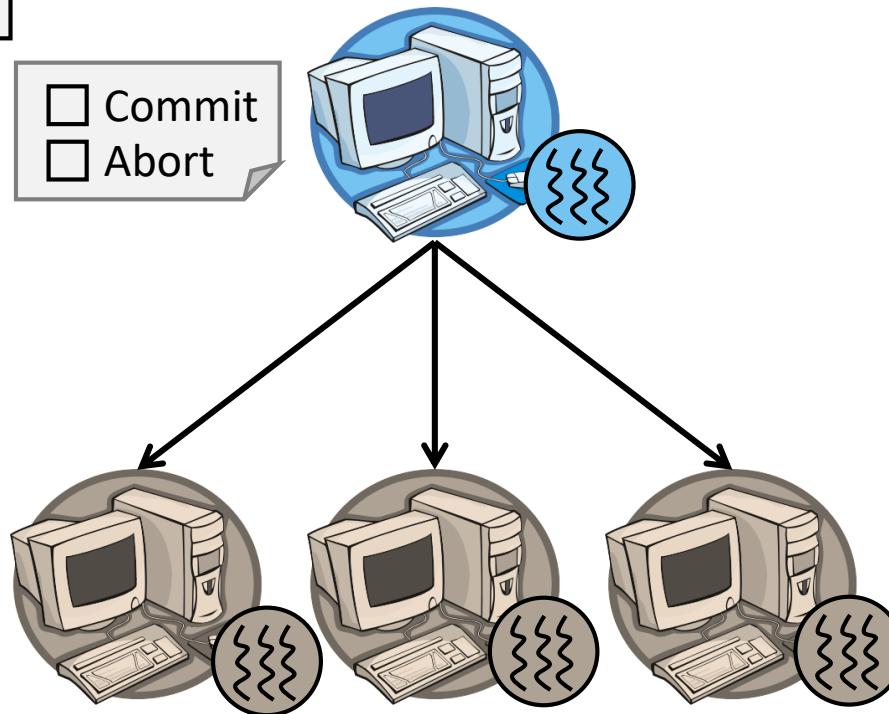


The status information
of the log will be persist
for any of machine
crashes 😊

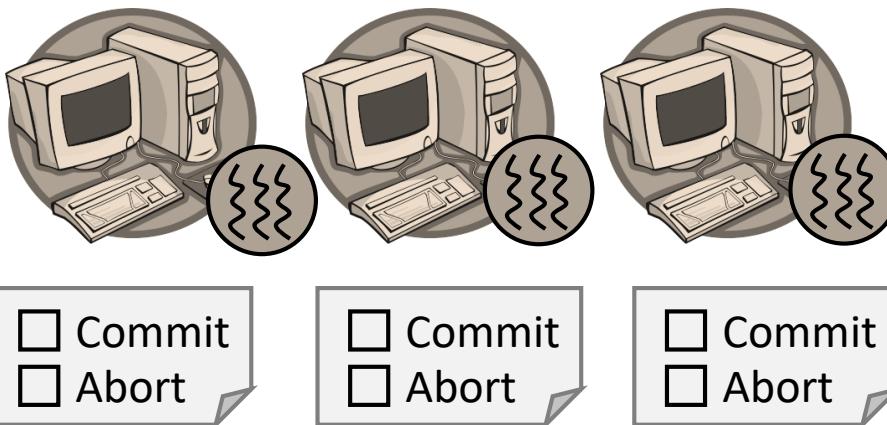
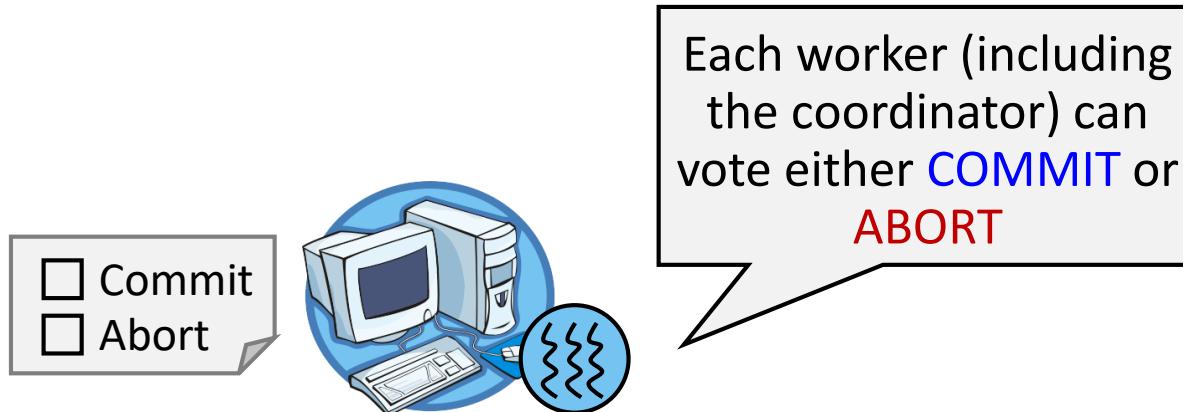


2PC Terminology

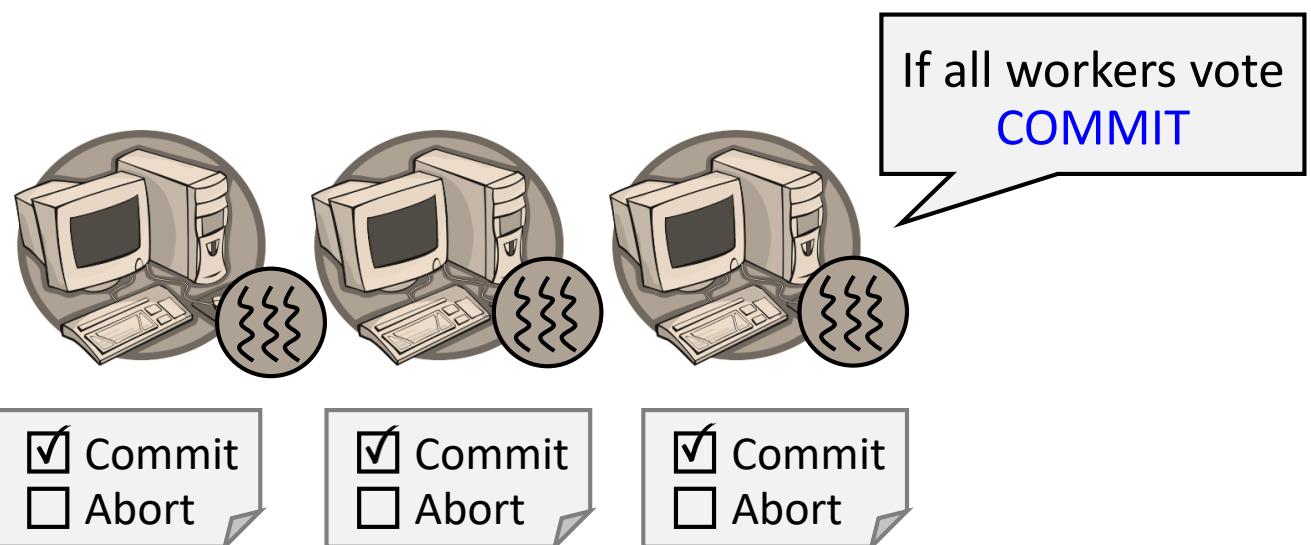
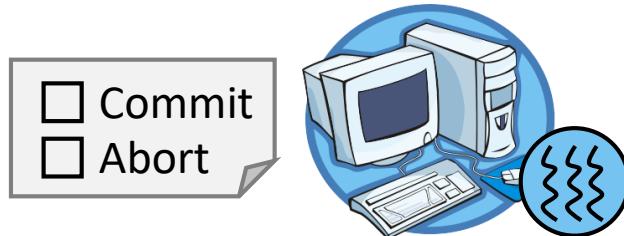
The coordinator asks all processes to vote



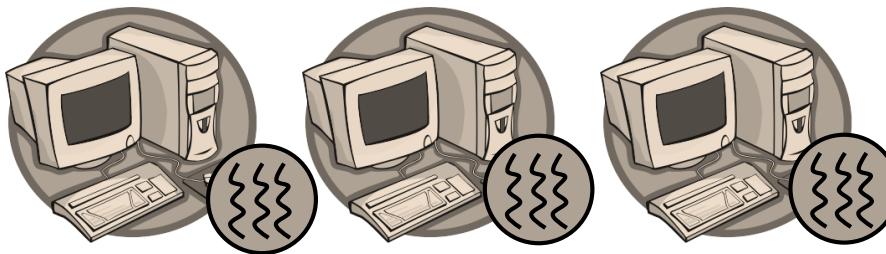
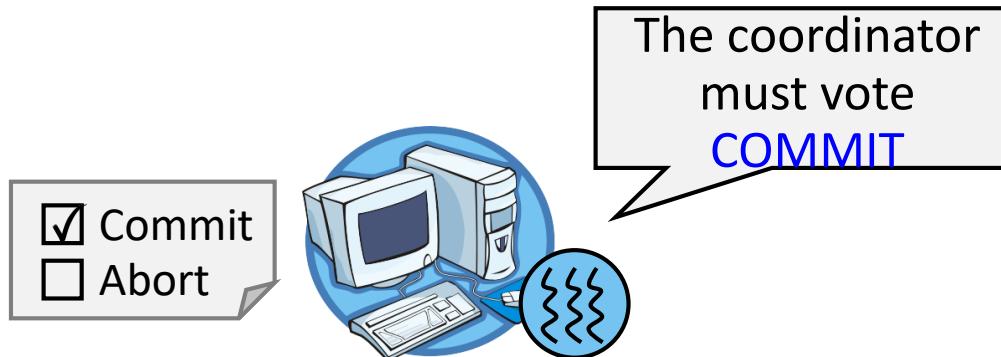
2PC Terminology



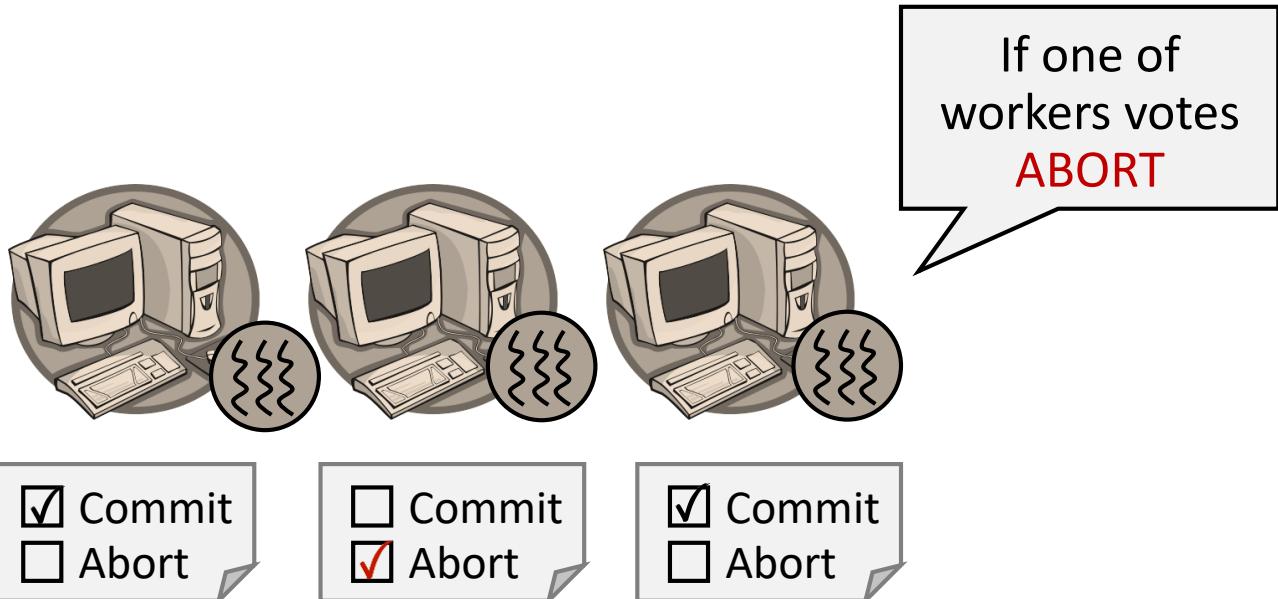
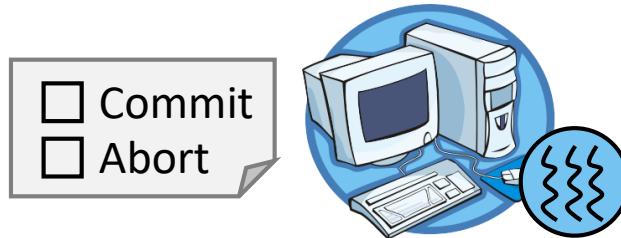
2PC Terminology



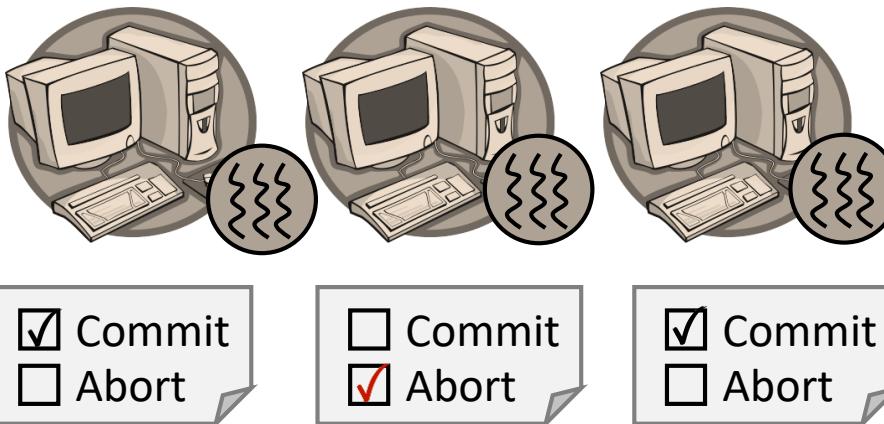
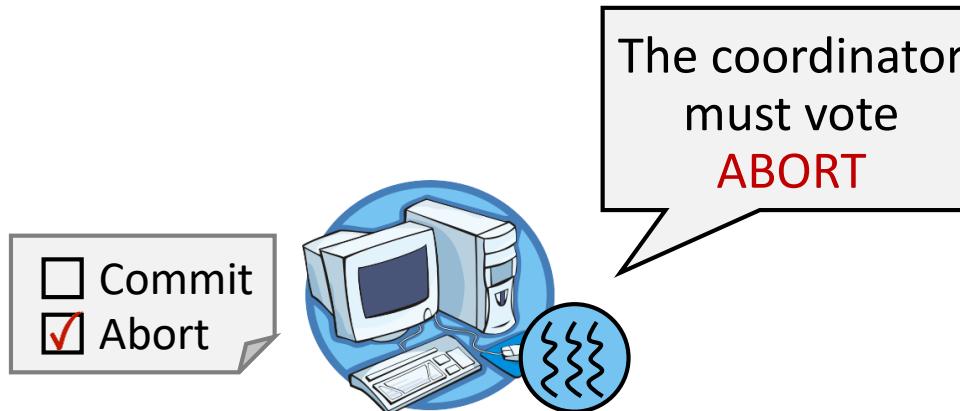
2PC Terminology



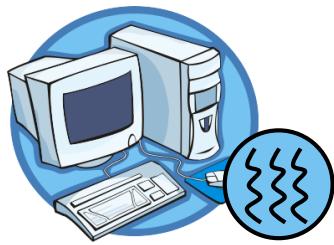
2PC Terminology



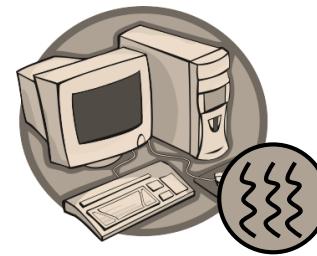
2PC Terminology



2PC: The Easy Case (No Failures)



Coordinator Algorithm



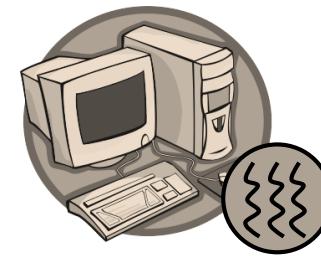
Worker Algorithm

Let's check the simple
2PC algorithm (no
failure consideration)

2PC: The Easy Case (No Failures)



Coordinator Algorithm



Worker Algorithm

1. Request votes

- Sends VOTE-REQ to all the workers

2. Voting

- Send VOTE-COMMIT or VOTE-ABORT to the coordinator
- If the work sent VOTE-ABORT back to the coordinator, it immediately aborts the request/action

3. Collect votes

- If all the workers agree with COMMIT, the coordinator broadcasts GLOBAL-COMMIT to all the workers
- If there is any pending (thus, the coordinator couldn't receive VOTE-COMMIT from all N workers), it sends GLOBAL-ABORT to all the workers

4. Finalization

- If GLOBAL-COMMIT arrives, the work commits the request
- Otherwise (i.e., GLOBAL-ABORT) it aborts the request

Failure Free Execution



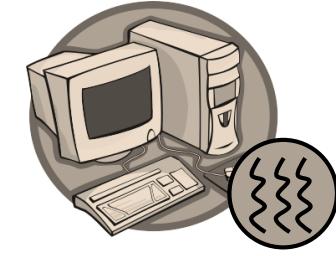
Coordinator

Time
→



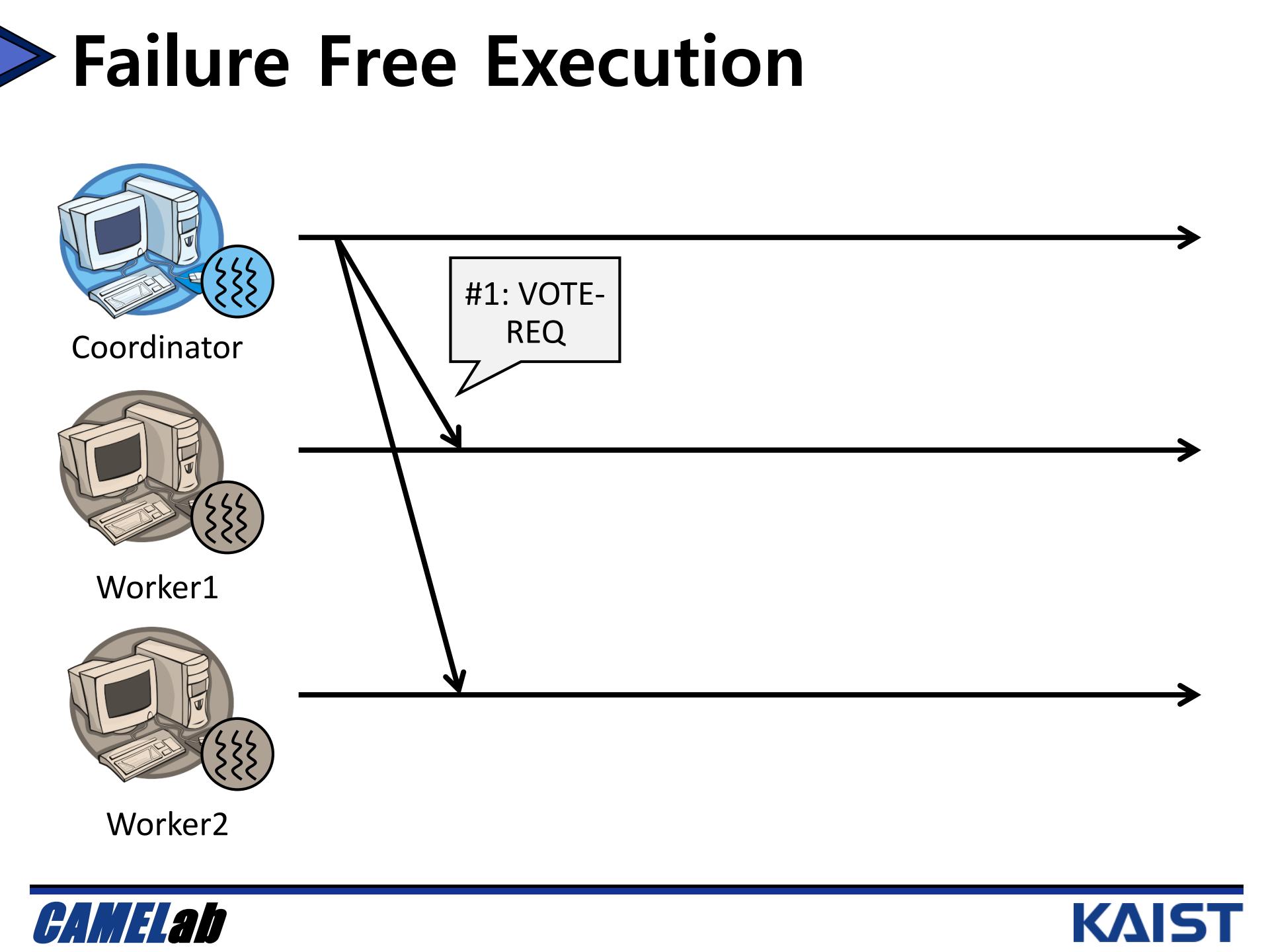
Worker1

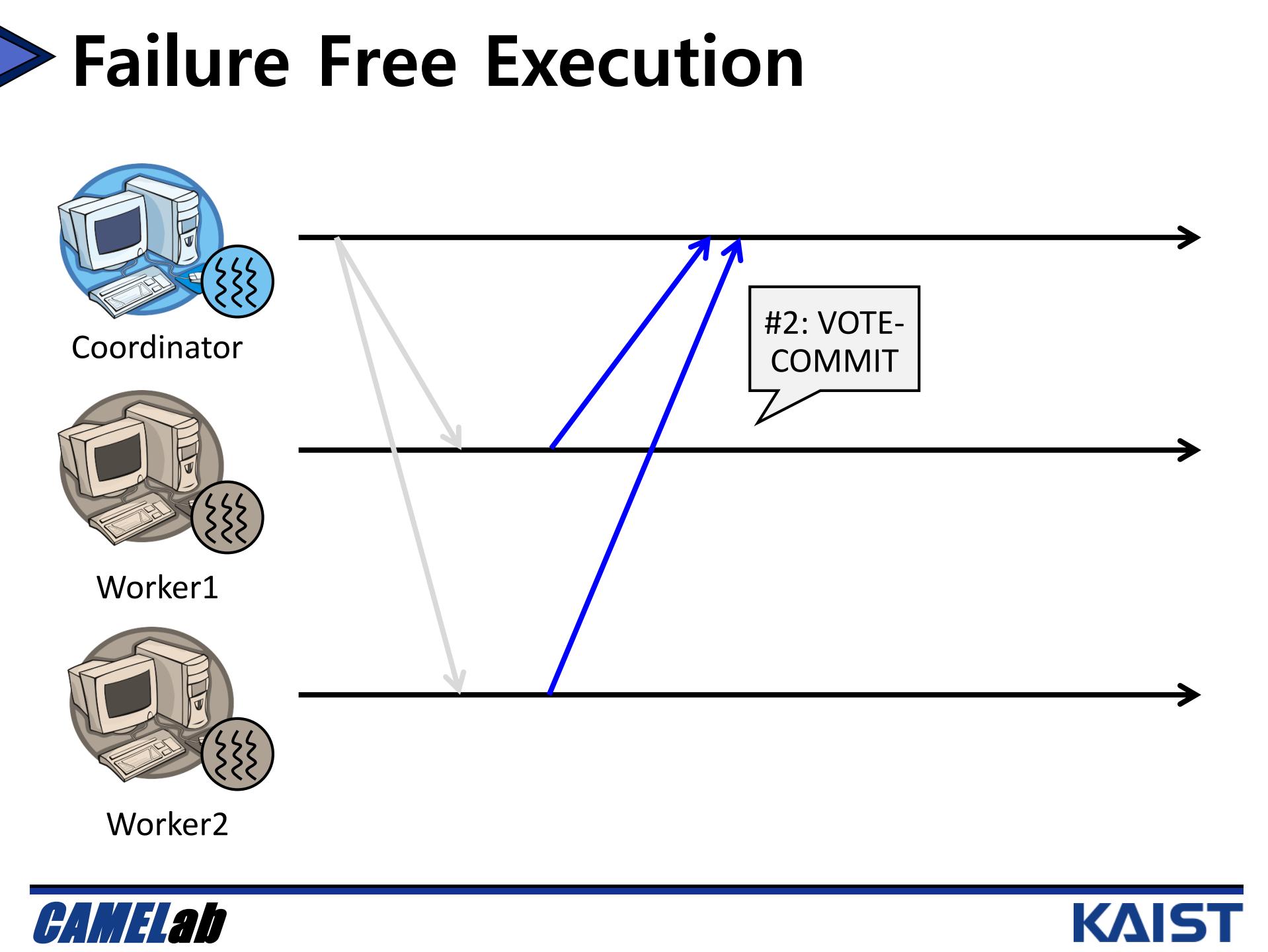
→

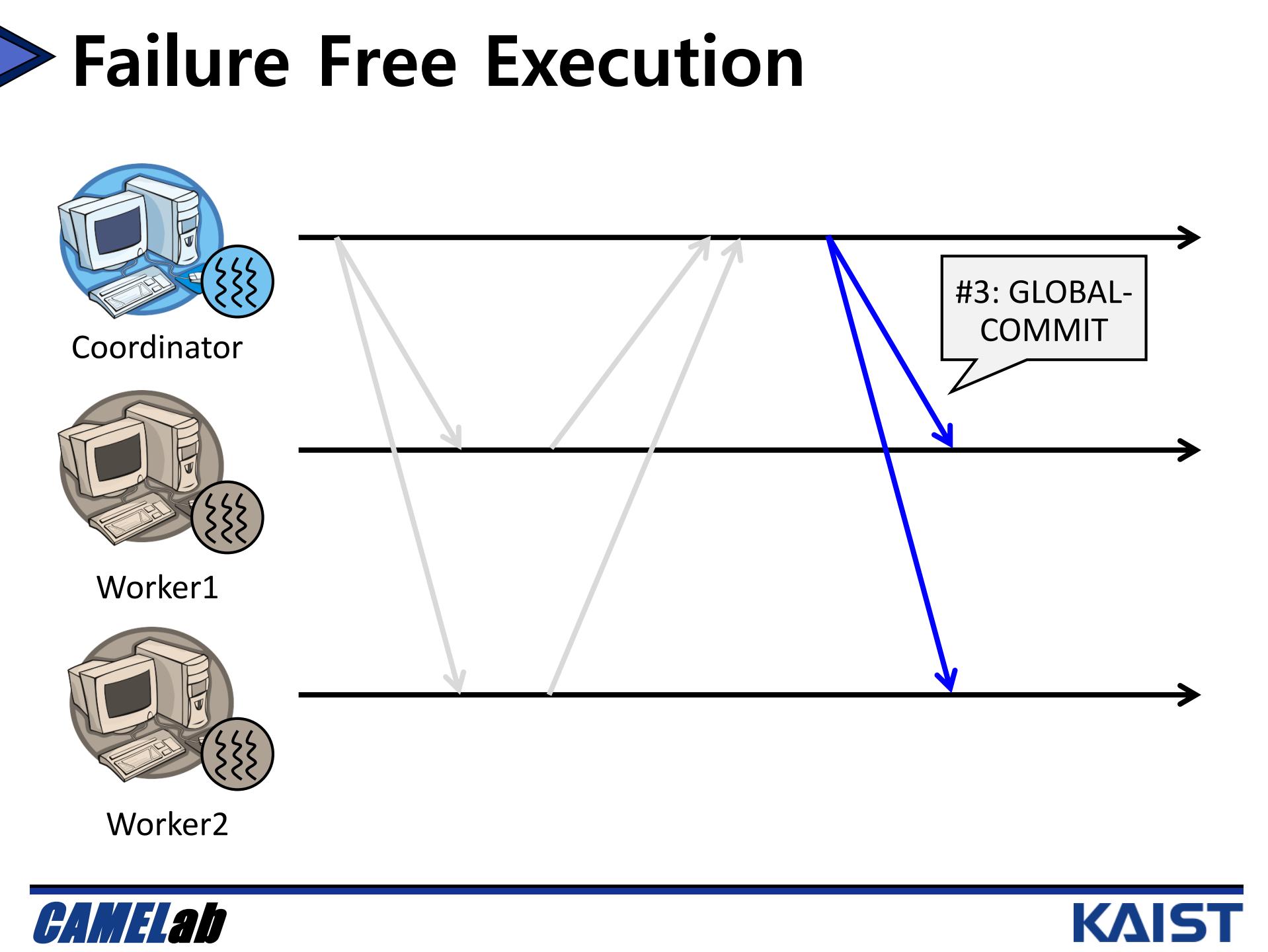


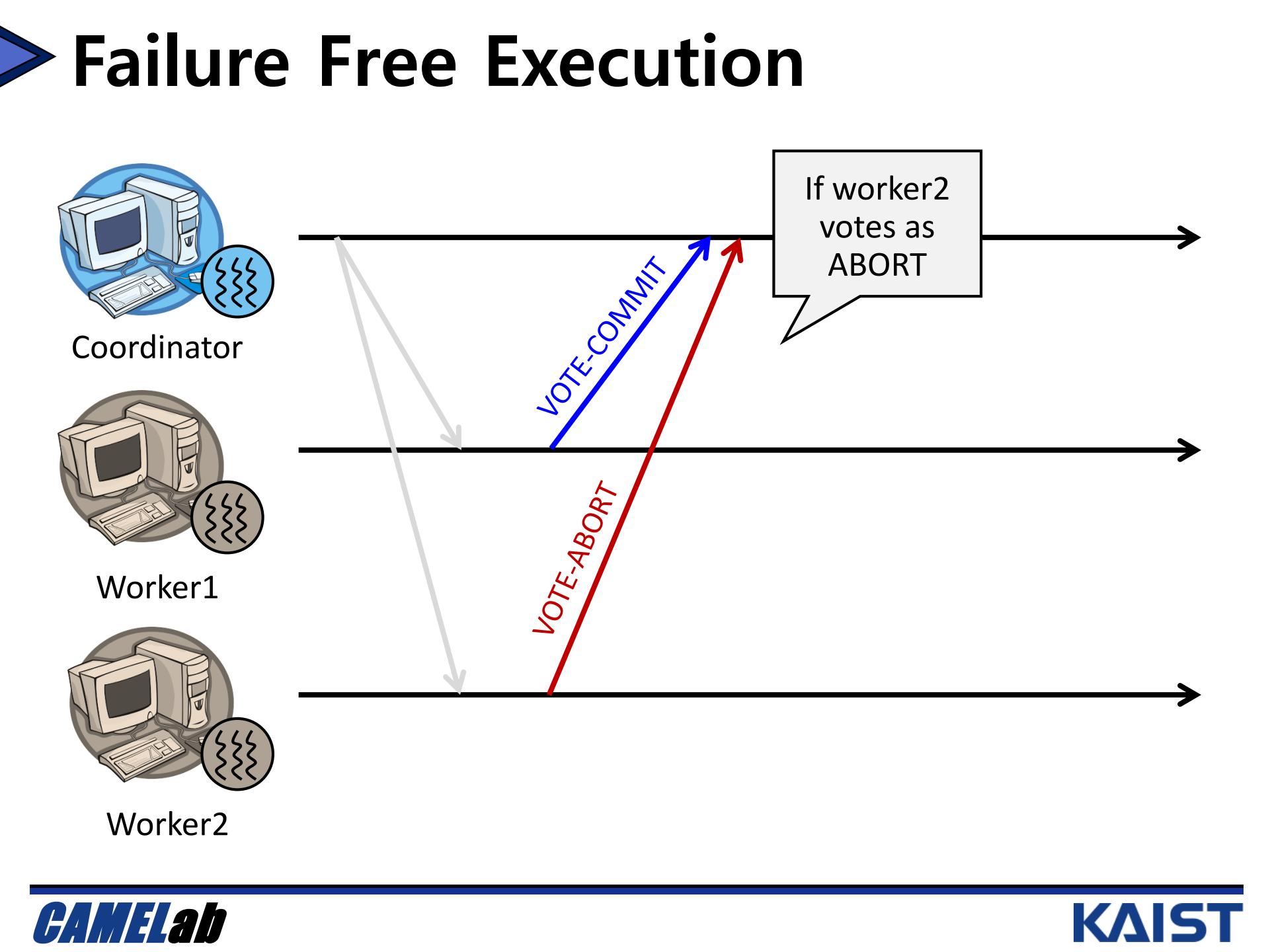
Worker2

→





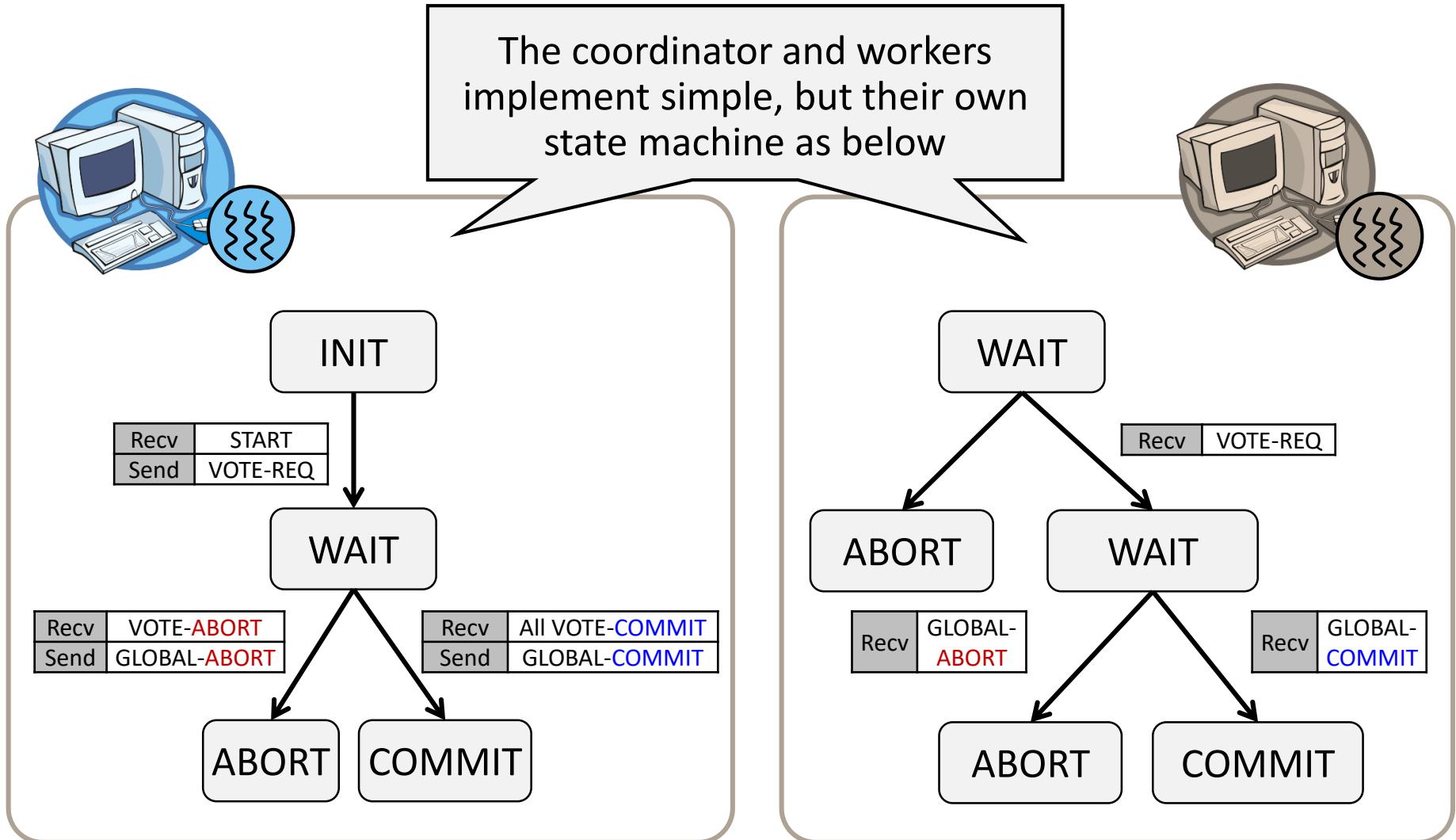




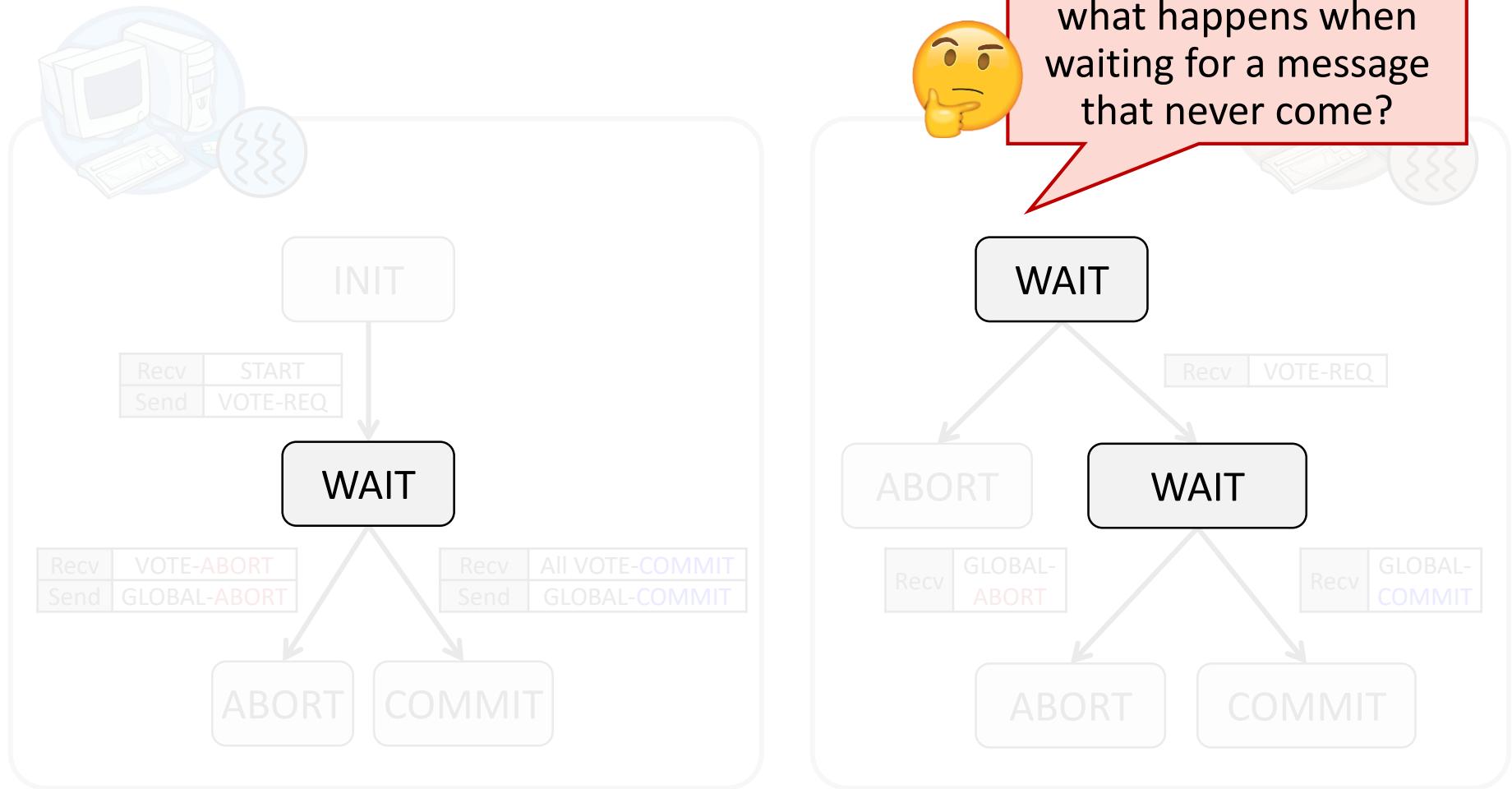
Failure Free Execution



State Machine (for a failure free)

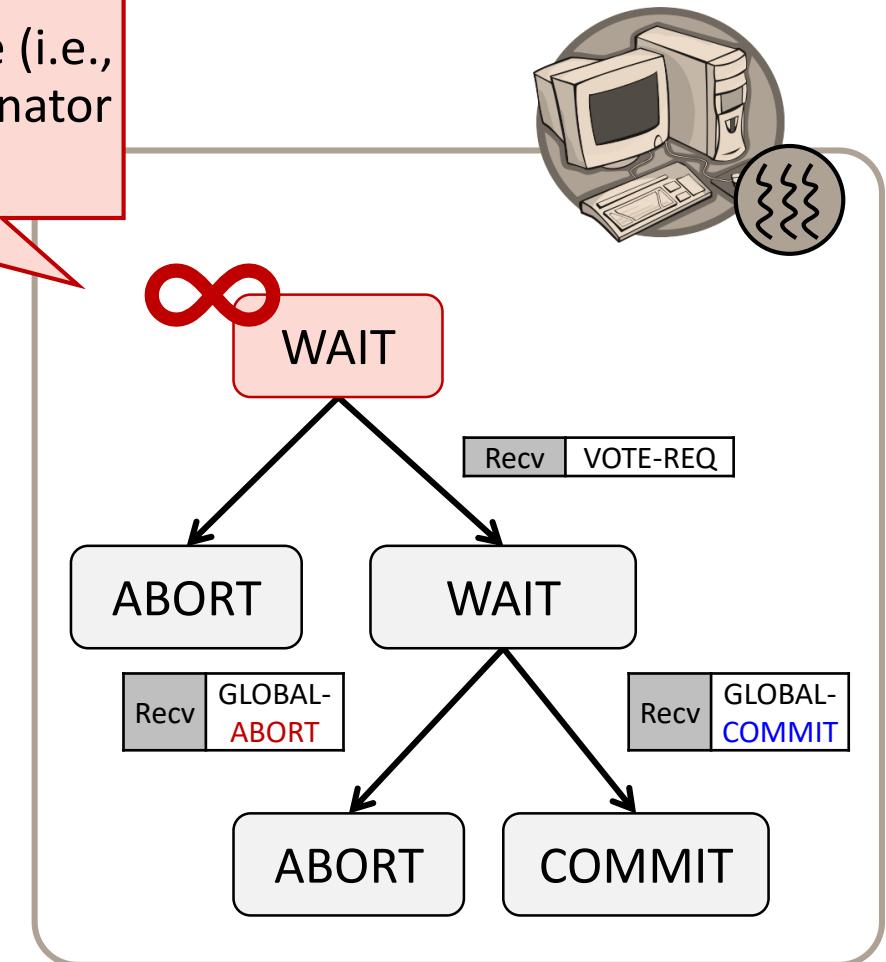


State Machine



Failure #1: VOTE-REQ Waiting

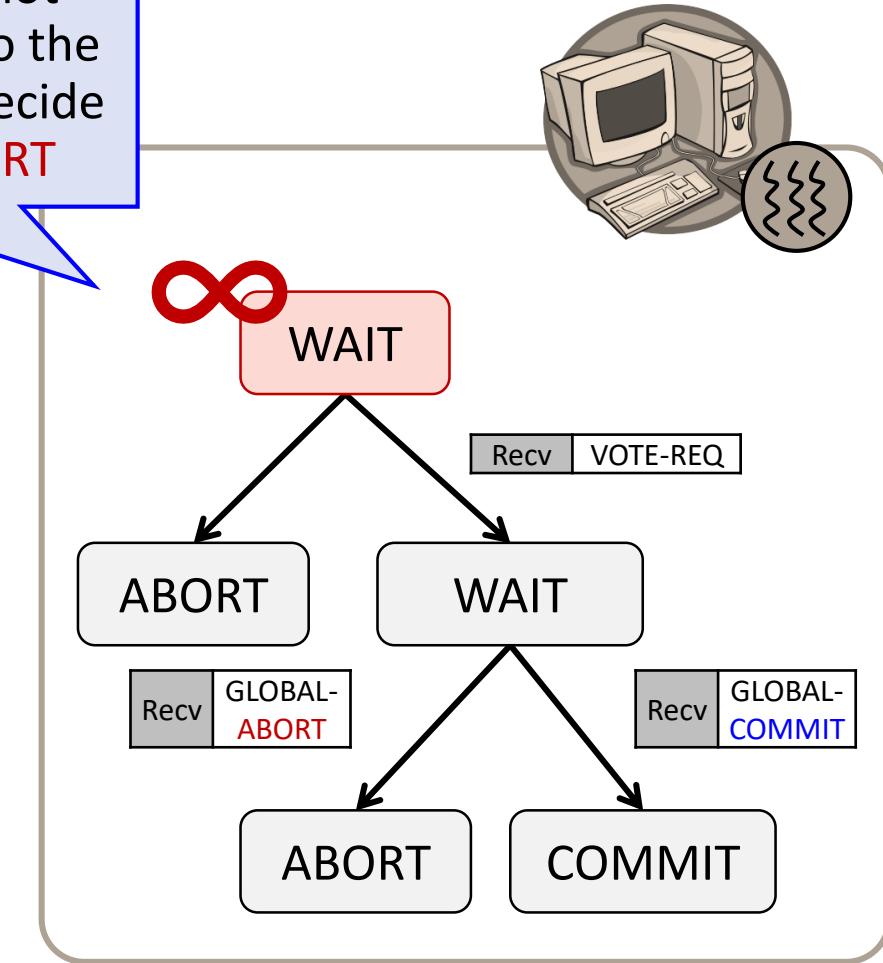
What if there is no message (i.e., VOTE-REQ) from the coordinator until the time is out?



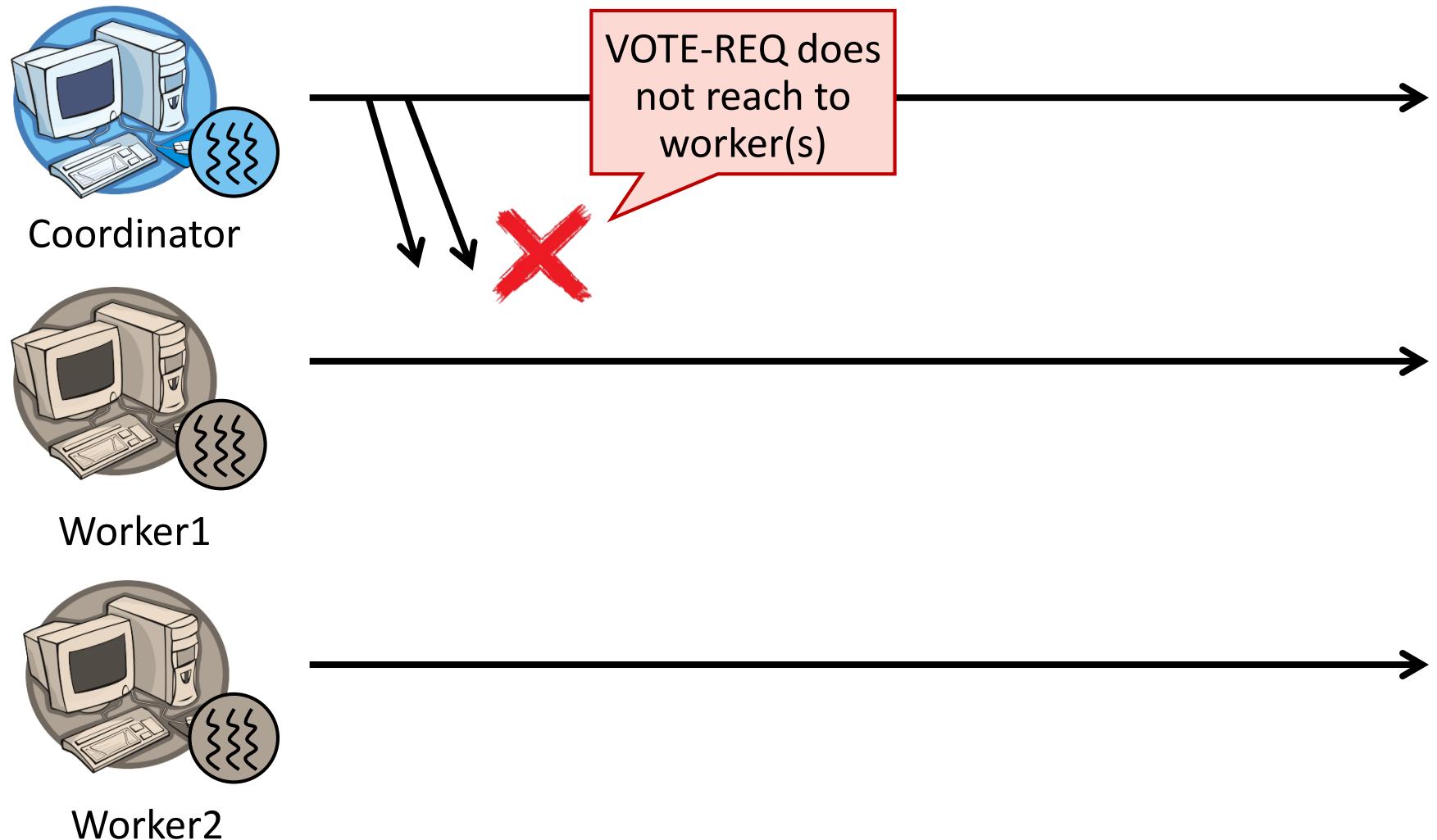


Solution: Timeout & VOTE-ABORT

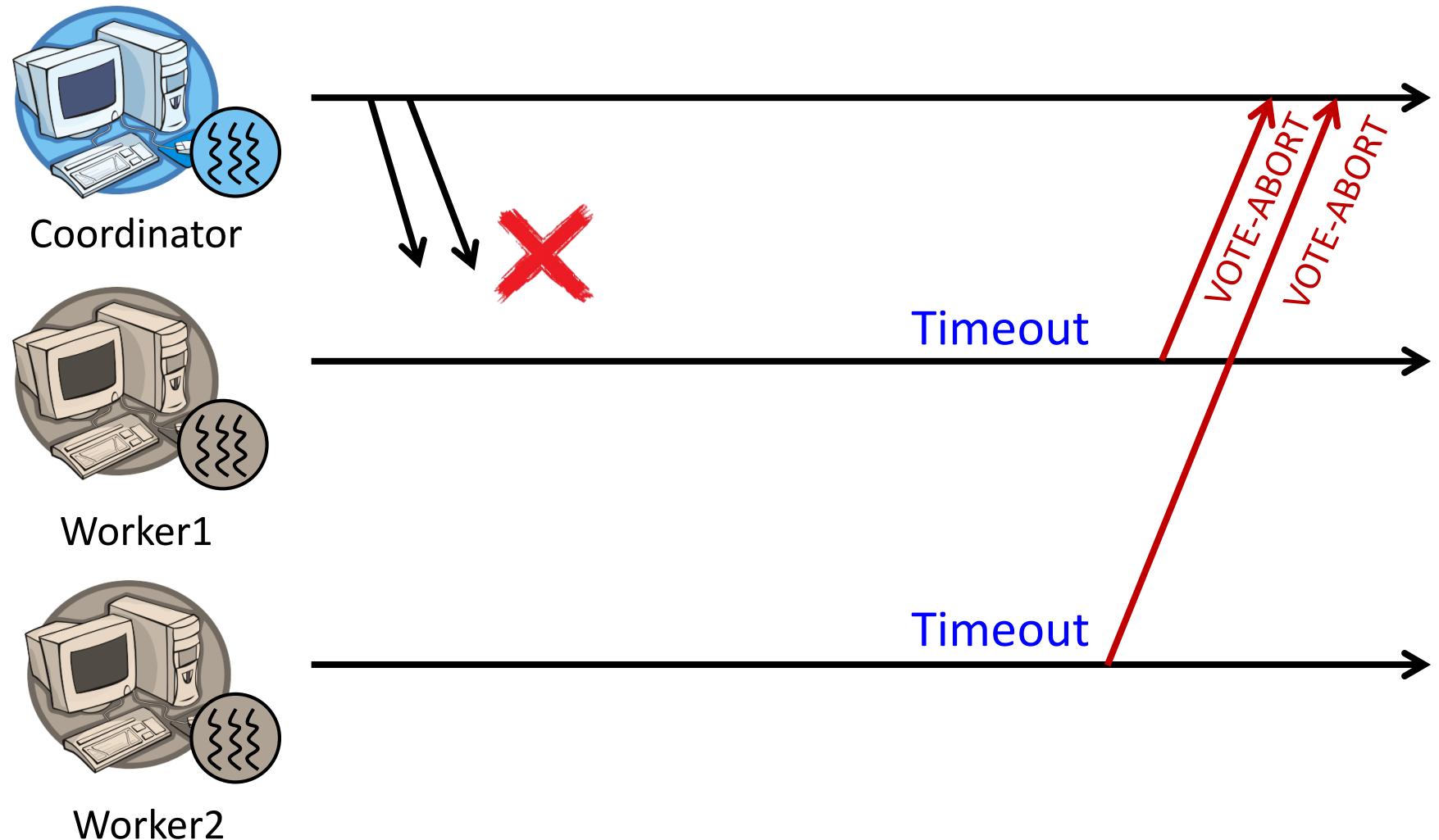
Since this worker has not issued a vote message to the coordinator yet, it can decide (e.g., send) VOTE-ABORT



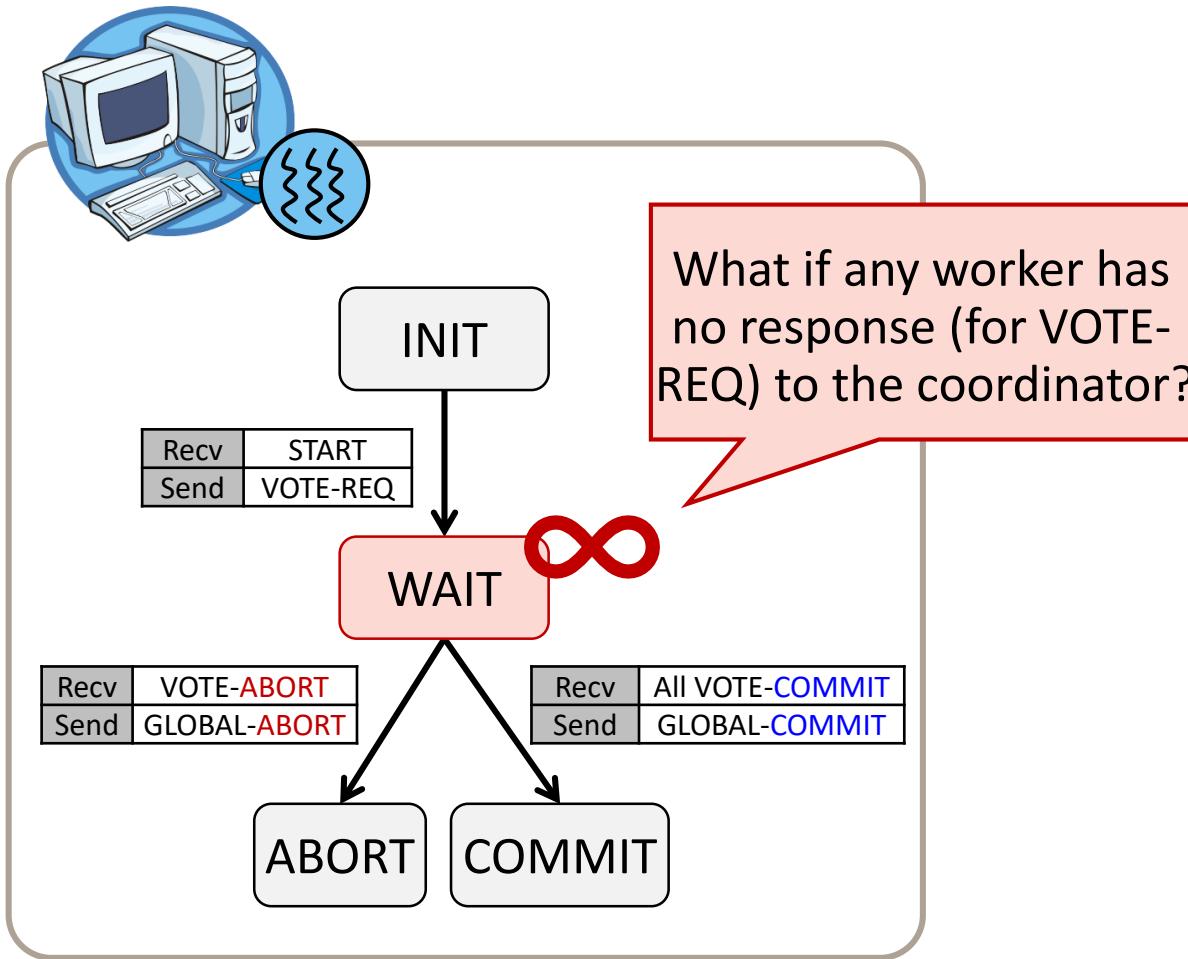
Solution: Timeout & VOTE-ABORT



> Solution: Timeout & VOTE-ABORT

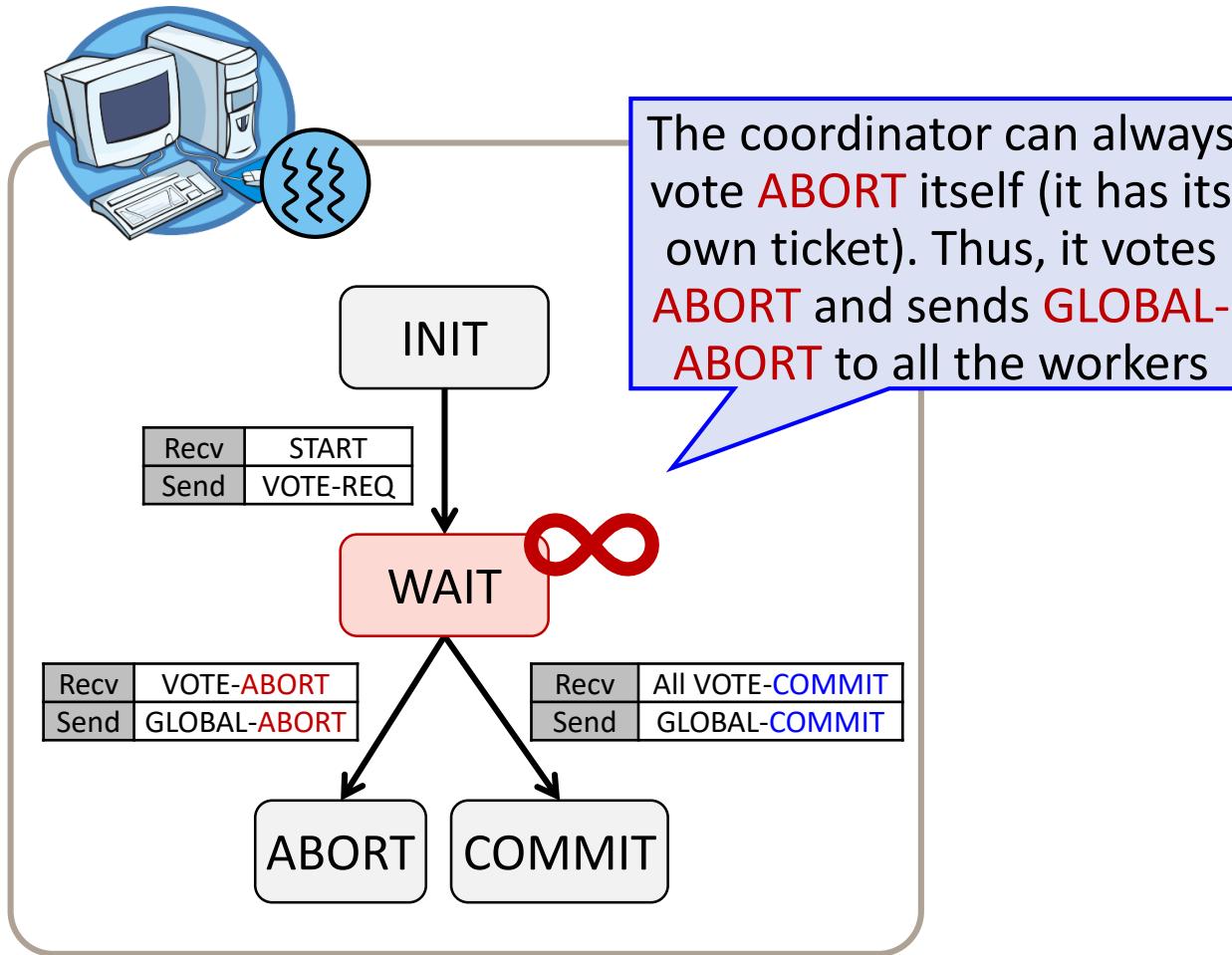


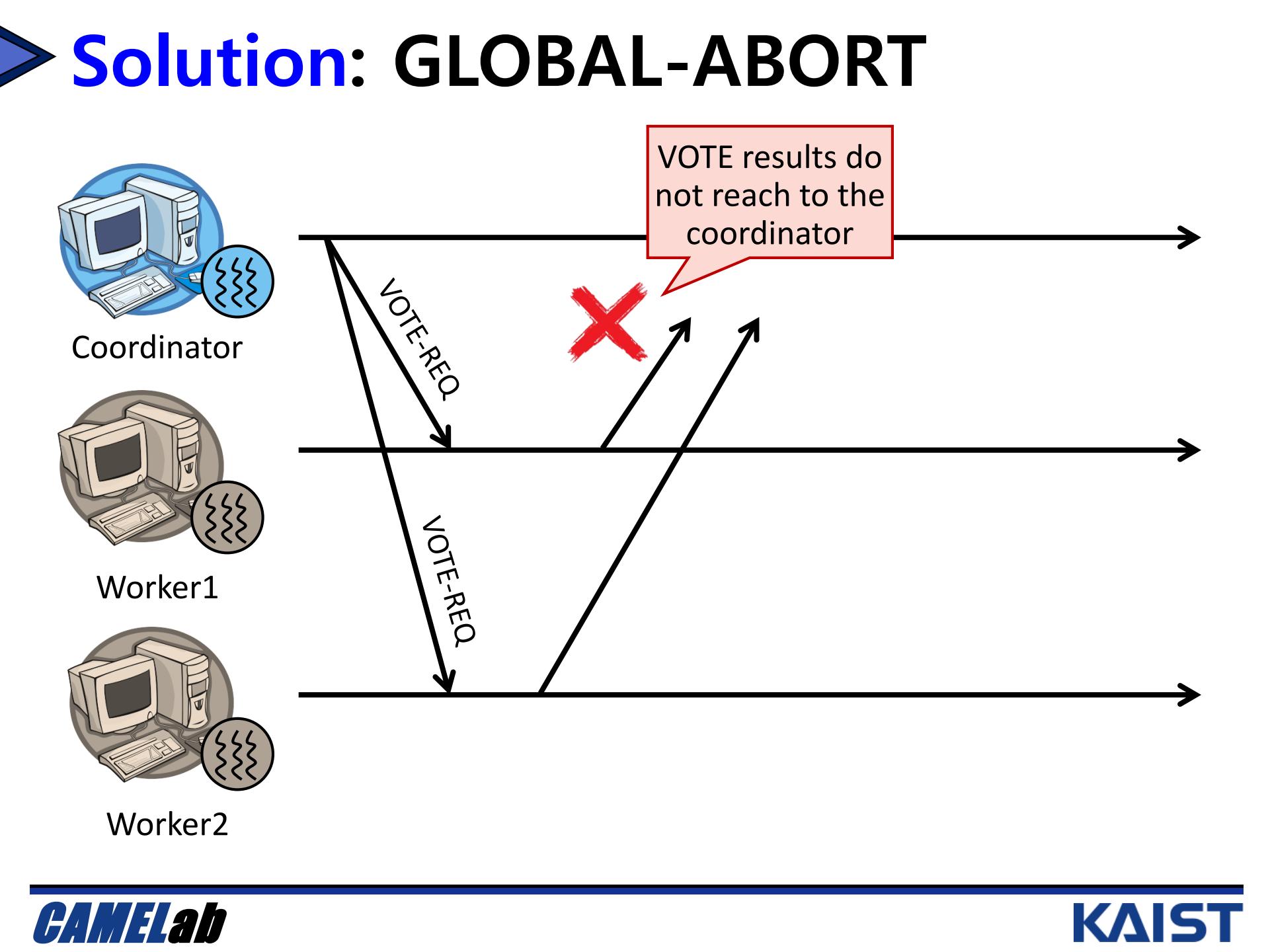
Failure #2: VOTE Waiting

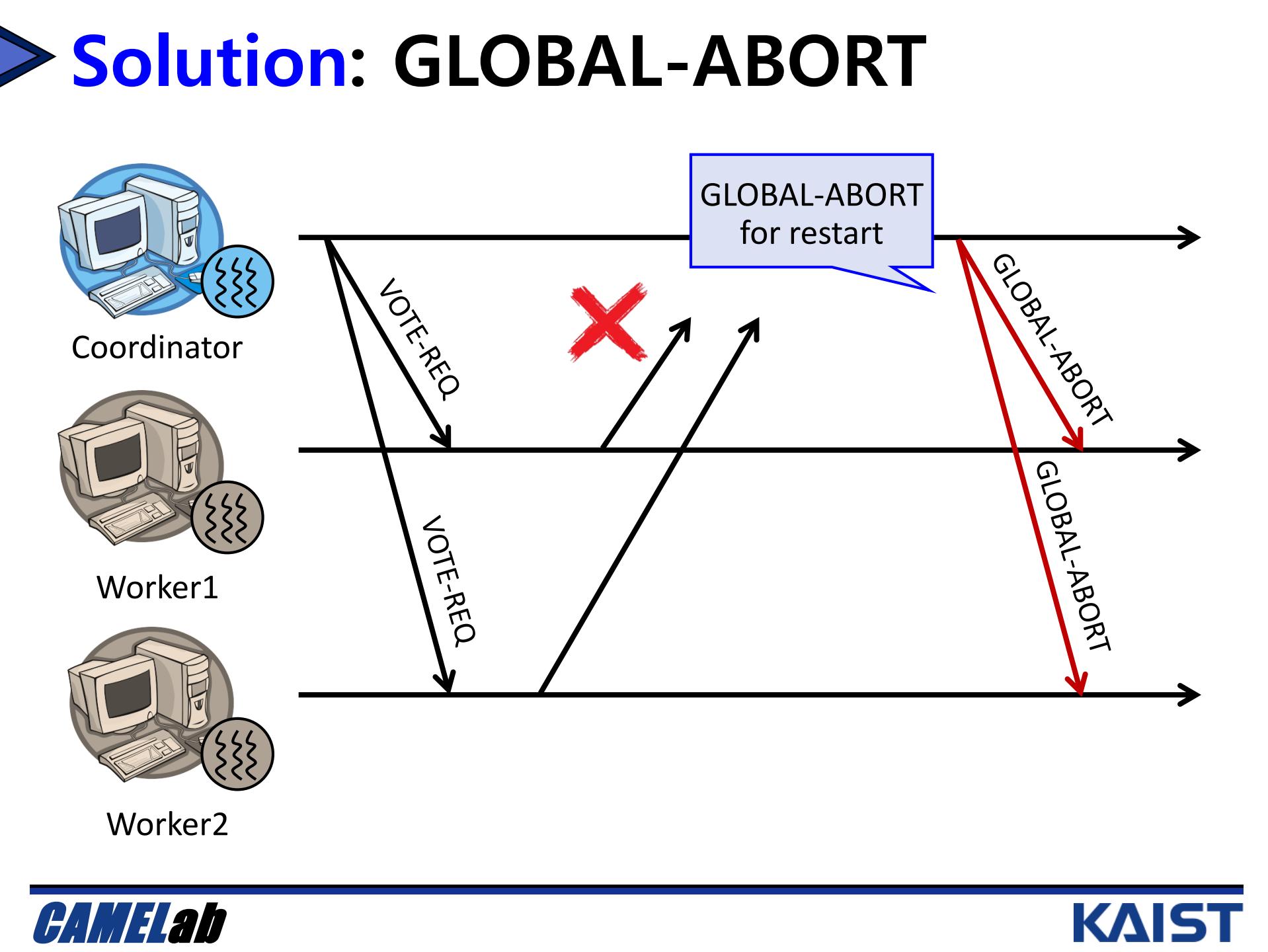




Solution: GLOBAL-ABORT

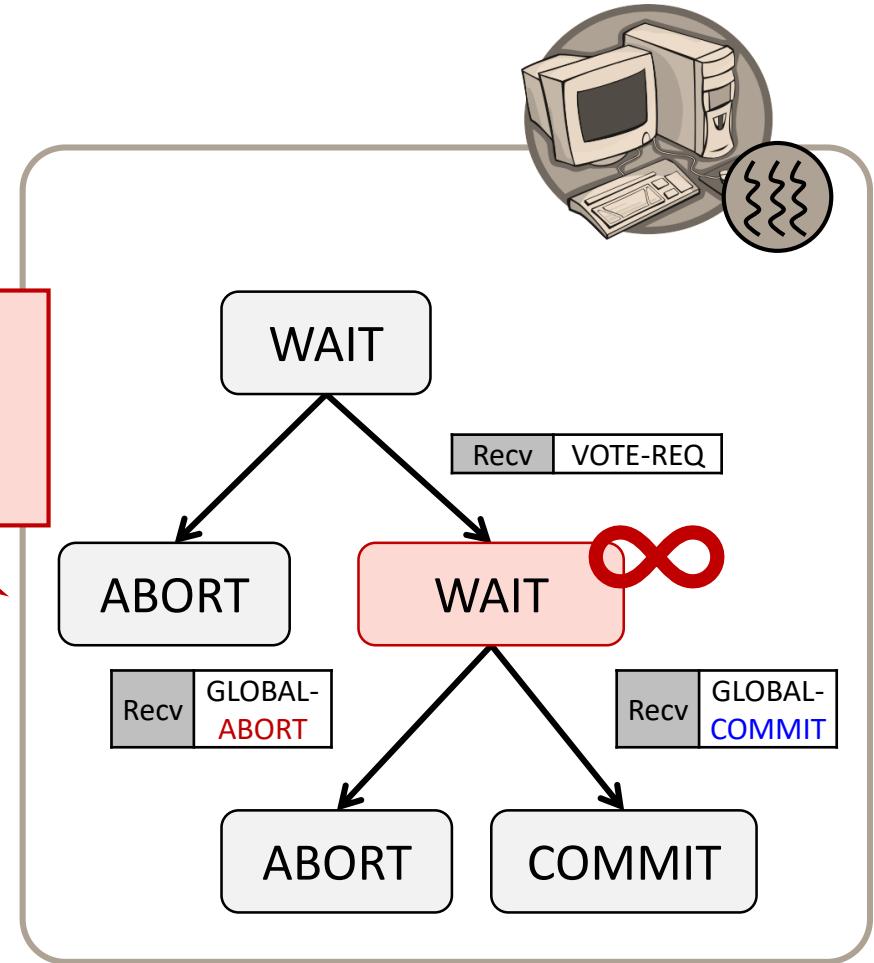






Failure #3: GLOBAL Decision Waiting

What if the coordinator never share its GLOBAL decision with the underlying workers?

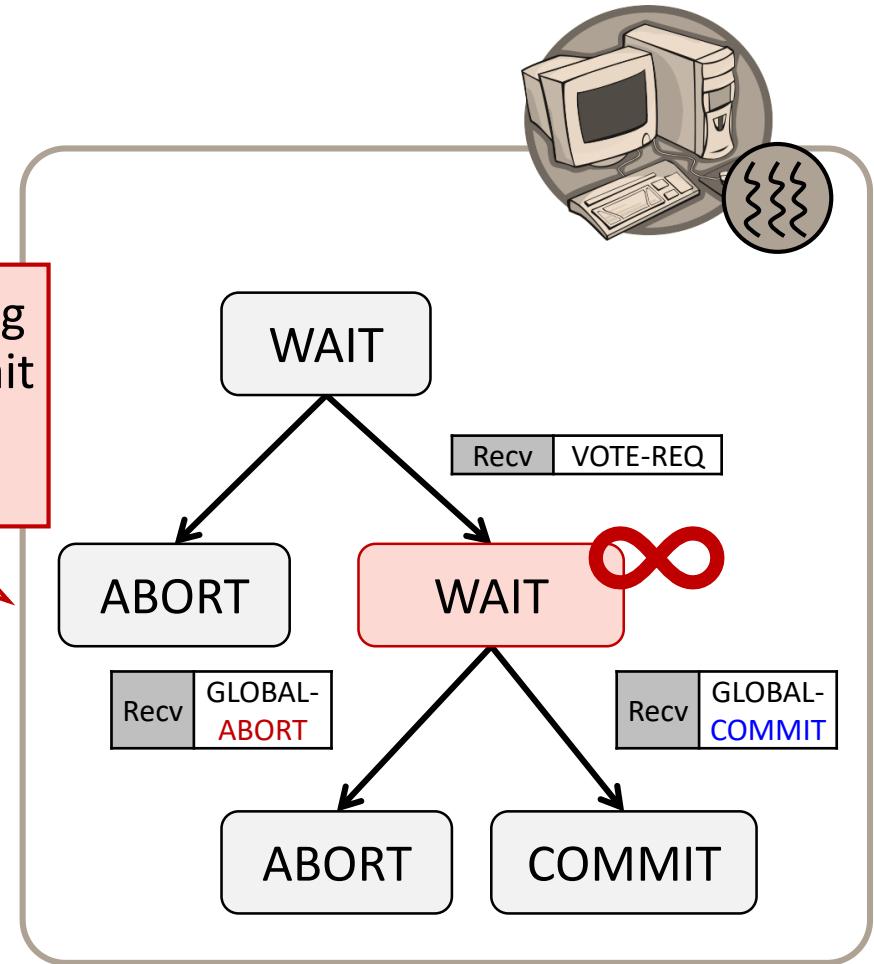




Solution: Wait for Recovery



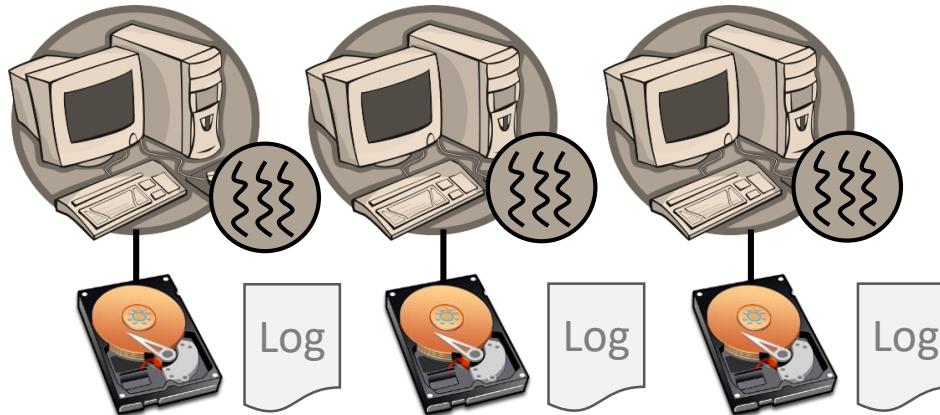
Worker cannot do anything at this juncture. It must wait for the coordinator to recover the request



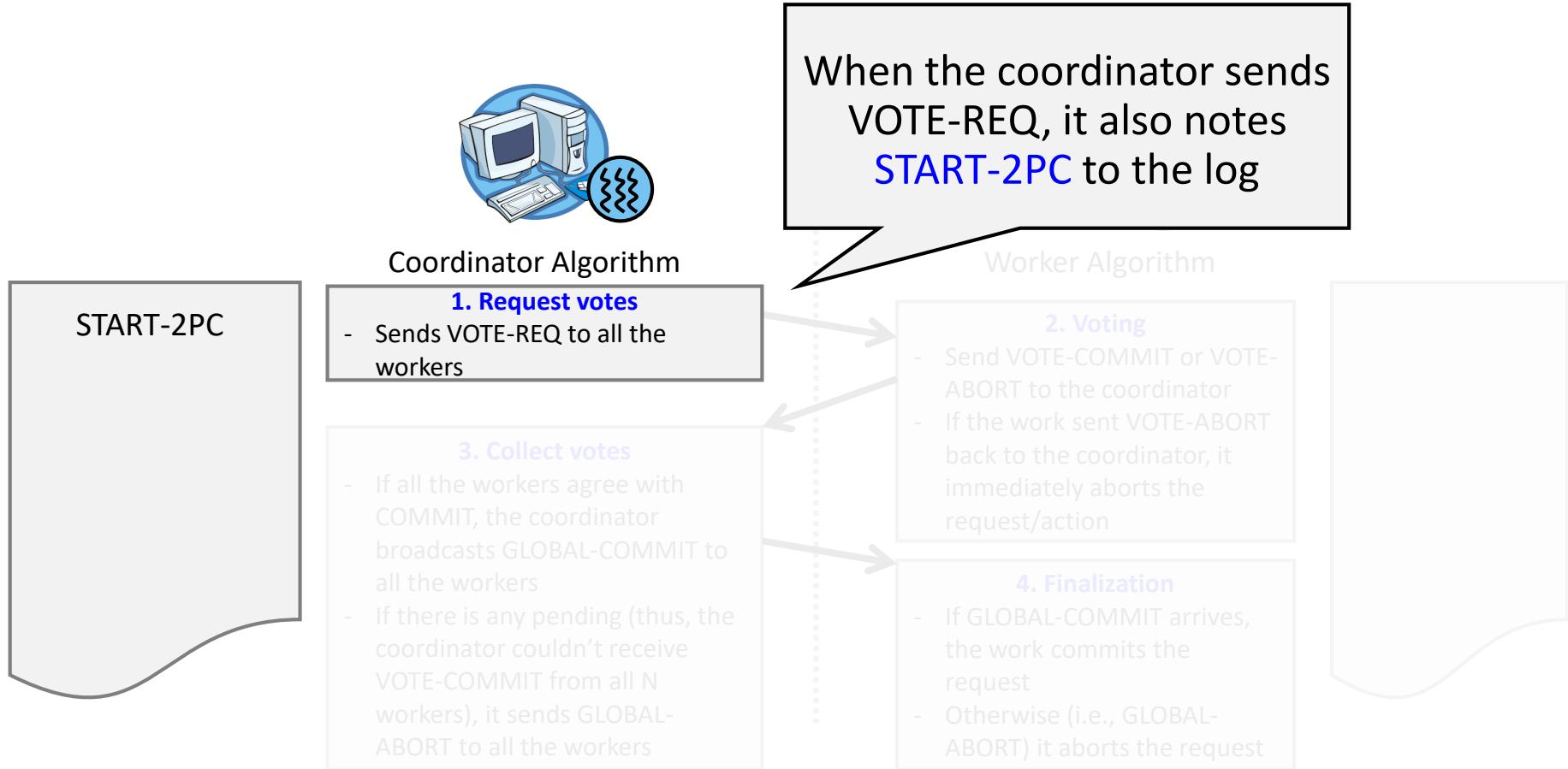
Machine Recovery



For the purpose of recovery management, all the nodes in this network can restore its own state and resume an action based on the persistent log



Status Logging



Status Logging

Before voting, the worker writes **VOTE-COMMIT** or **ABORT** to the log, and then, it sends its vote result to the coordinator



Worker Algorithm

START-2PC

- Sends VOTE-REQ to all the workers

3. Collect votes

- If all the workers agree with COMMIT, the coordinator broadcasts GLOBAL-COMMIT to all the workers
- If there is any pending (thus, the coordinator couldn't receive VOTE-COMMIT from all N workers), it sends GLOBAL-ABORT to all the workers

2. Voting

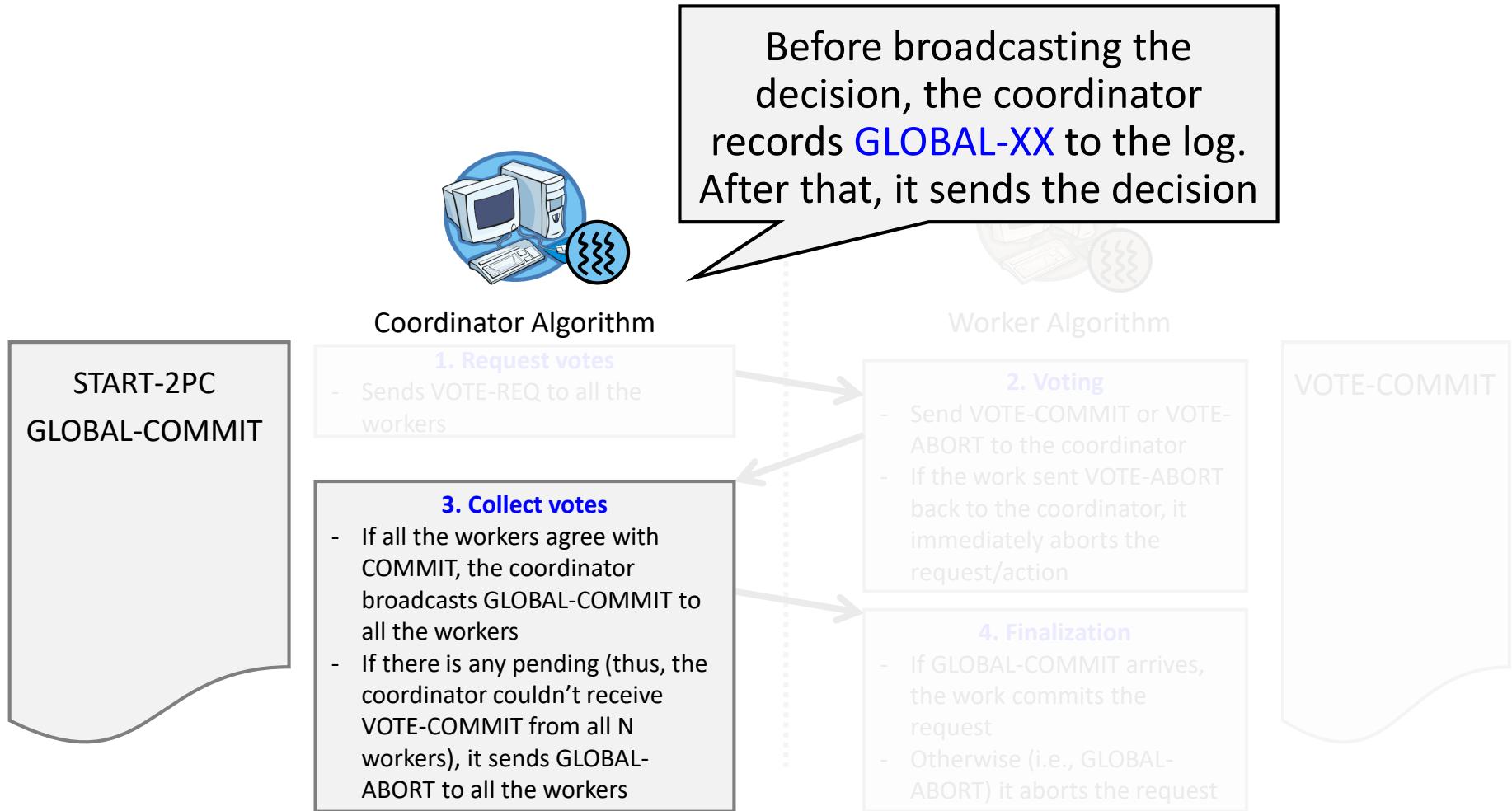
- Send VOTE-COMMIT or VOTE-ABORT to the coordinator
- If the work sent VOTE-ABORT back to the coordinator, it immediately aborts the request/action

4. Finalization

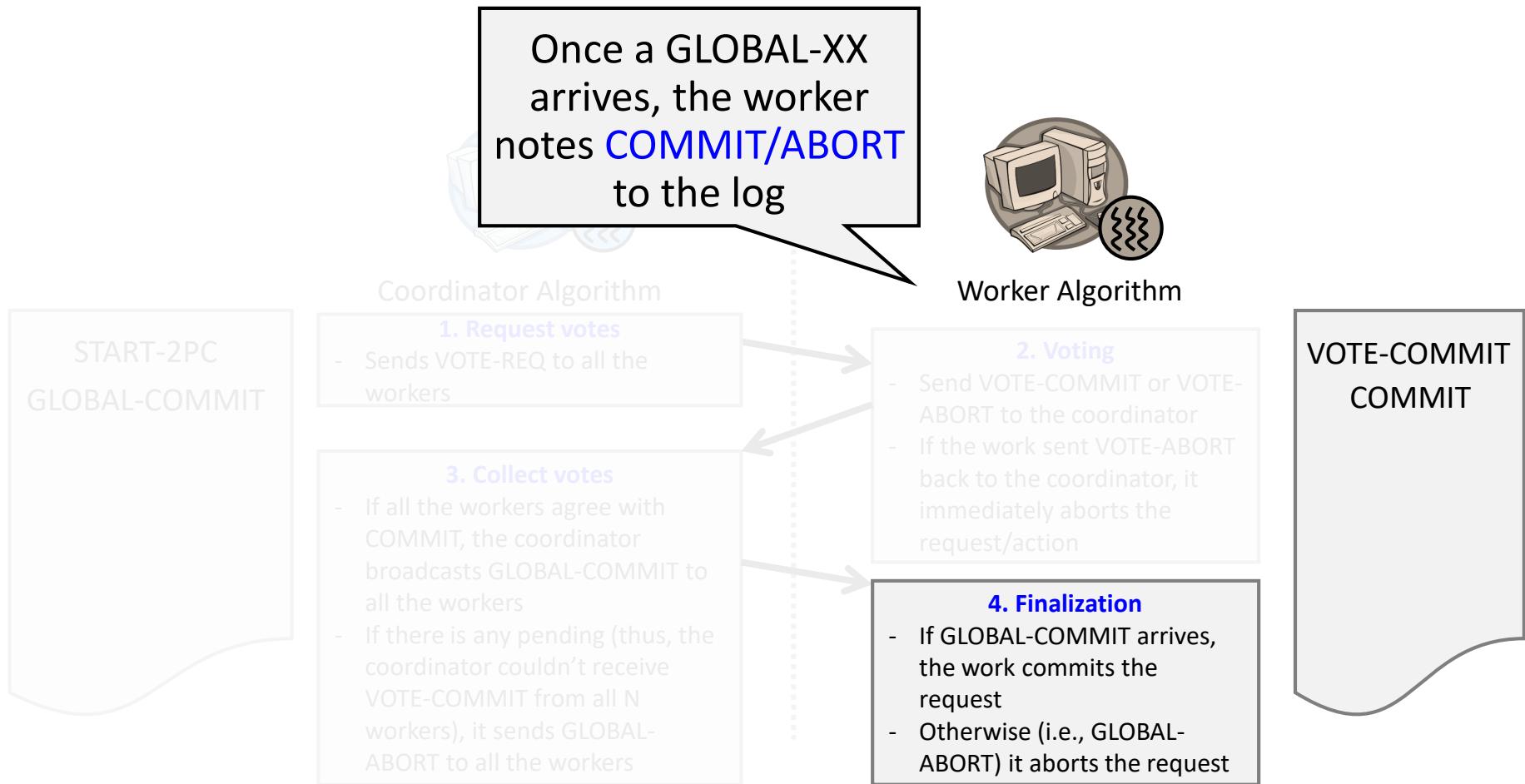
- If GLOBAL-COMMIT arrives, the work commits the request
- Otherwise (i.e., GLOBAL-ABORT) it aborts the request

VOTE-COMMIT

Status Logging

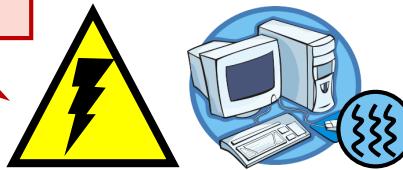


Status Logging



Recovery Scenario #1

What if the coordinator faces a power failure?



Coordinator Algorithm

START-2PC

1. Request votes

- Sends VOTE-REQ to all the workers

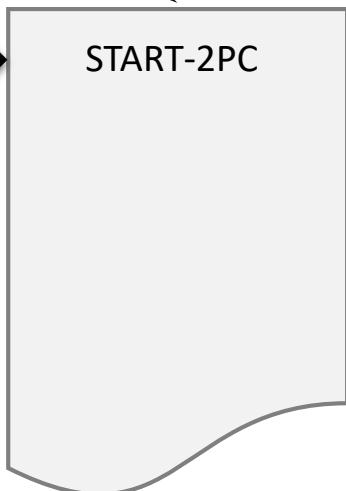
3. Collect votes

- If all the workers agree with COMMIT, the coordinator broadcasts GLOBAL-COMMIT to all the workers
- If there is any pending (thus, the coordinator couldn't receive VOTE-COMMIT from all N workers), it sends GLOBAL-ABORT to all the workers



Unilaterally ABORT

The coordinator reads its log. If it sees START-2PC but there is no record for a decision, the coordinator enforces ABORT right away



Coordinator Algorithm

1. Request votes

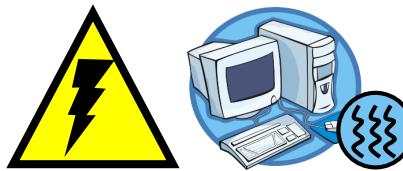
- Sends VOTE-REQ to all the workers

3. Collect votes

- If all the workers agree with COMMIT, the coordinator broadcasts GLOBAL-COMMIT to all the workers
- If there is any pending (thus, the coordinator couldn't receive VOTE-COMMIT from all N workers), it sends GLOBAL-ABORT to all the workers

Recovery Scenario #2

What if the coordinator faces a power failure, here?



Coordinator Algorithm

1. Request votes

- Sends VOTE-REQ to all the workers

3. Collect votes

- If all the workers agree with COMMIT, the coordinator broadcasts GLOBAL-COMMIT to all the workers
- If there is any pending (thus, the coordinator couldn't receive VOTE-COMMIT from all N workers), it sends GLOBAL-ABORT to all the workers

START-2PC
GLOBAL-COMMIT

Resending Decision

If the coordinator sees any of GLOBAL-XX from the log, it resends the recorded decision



Coordinator Algorithm

1. Request votes

- Sends VOTE-REQ to all the workers

3. Collect votes

- If all the workers agree with COMMIT, the coordinator broadcasts GLOBAL-COMMIT to all the workers
- If there is any pending (thus, the coordinator couldn't receive VOTE-COMMIT from all N workers), it sends GLOBAL-ABORT to all the workers

Recovery Scenario #3



Worker Algorithm



What if a worker faces a power failure, here?

2. Voting

- Send VOTE-COMMIT or VOTE-ABORT to the coordinator
- If the work sent VOTE-ABORT back to the coordinator, it immediately aborts the request/action

4. Finalization

- If GLOBAL-COMMIT arrives, the work commits the request
- Otherwise (i.e., GLOBAL-ABORT) it aborts the request



VOTE-ABORT



Worker Algorithm

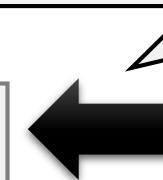
2. Voting

- Send VOTE-COMMIT or VOTE-ABORT to the coordinator
- If the work sent VOTE-ABORT back to the coordinator, it immediately aborts the request/action

4. Finalization

- If GLOBAL-COMMIT arrives, the work commits the request
- Otherwise (i.e., GLOBAL-ABORT) it aborts the request

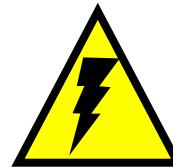
The worker reads log. If it doesn't see any note from the log, the worker sends **VOTE-ABORT**



Recovery Scenario #4



Worker Algorithm



What if a worker faces a power failure, here?

2. Voting

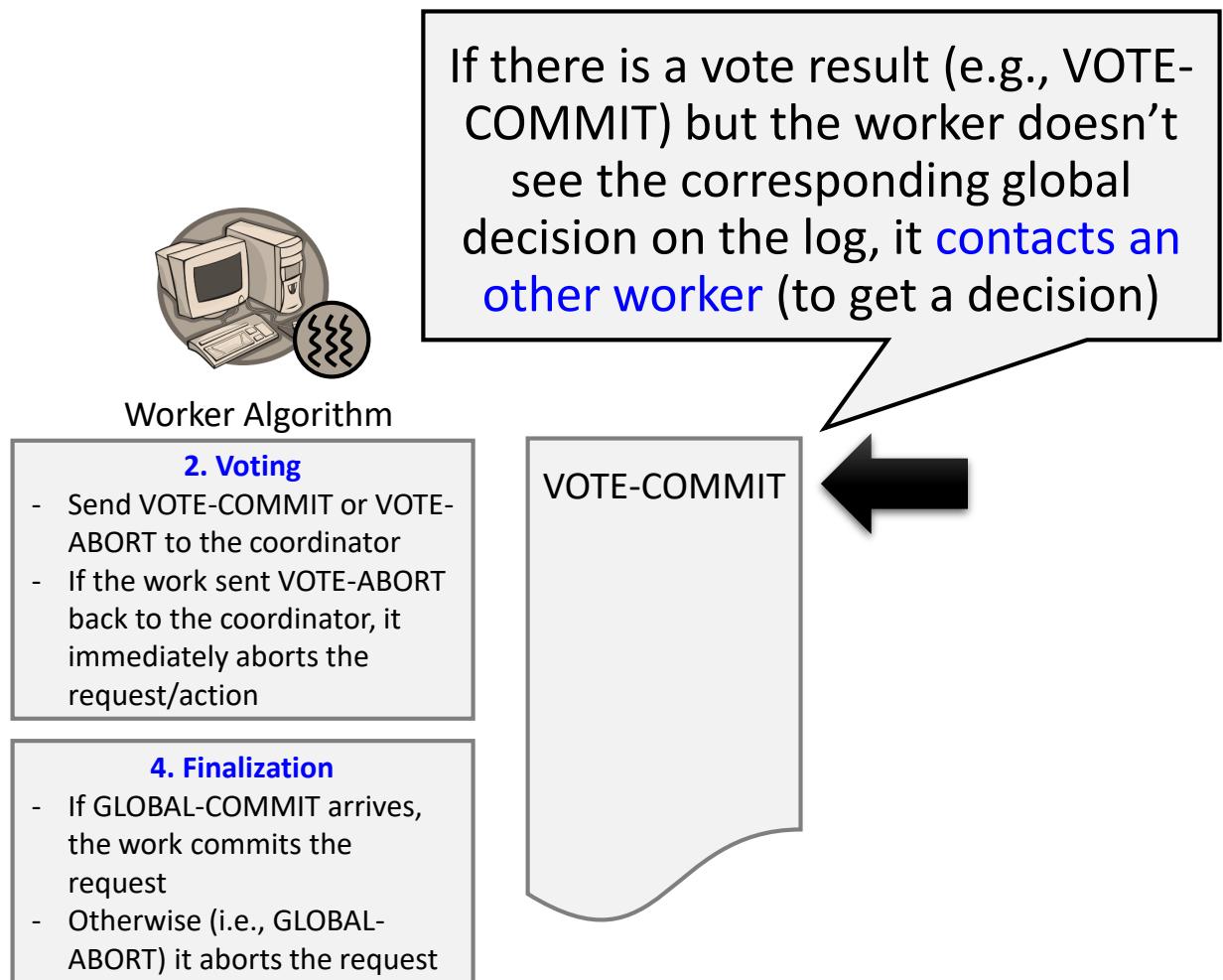
- Send VOTE-COMMIT or VOTE-ABORT to the coordinator
- If the work sent VOTE-ABORT back to the coordinator, it immediately aborts the request/action

4. Finalization

- If GLOBAL-COMMIT arrives, the work commits the request
- Otherwise (i.e., GLOBAL-ABORT) it aborts the request

VOTE-COMMIT

Contact Friend Worker



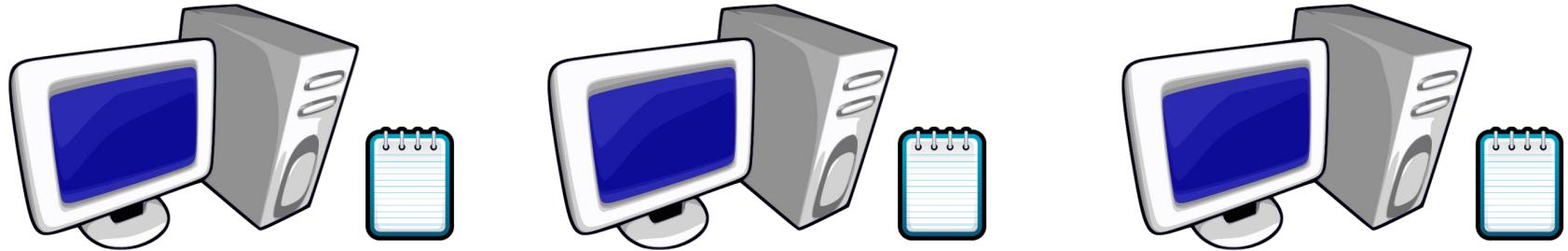
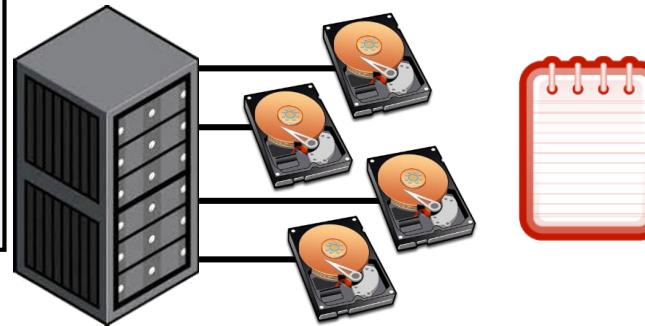
[Summary] 2PC

Two-Phase Commit (2PC) largely supports 4 properties for a distributed decision making (distributed transaction)

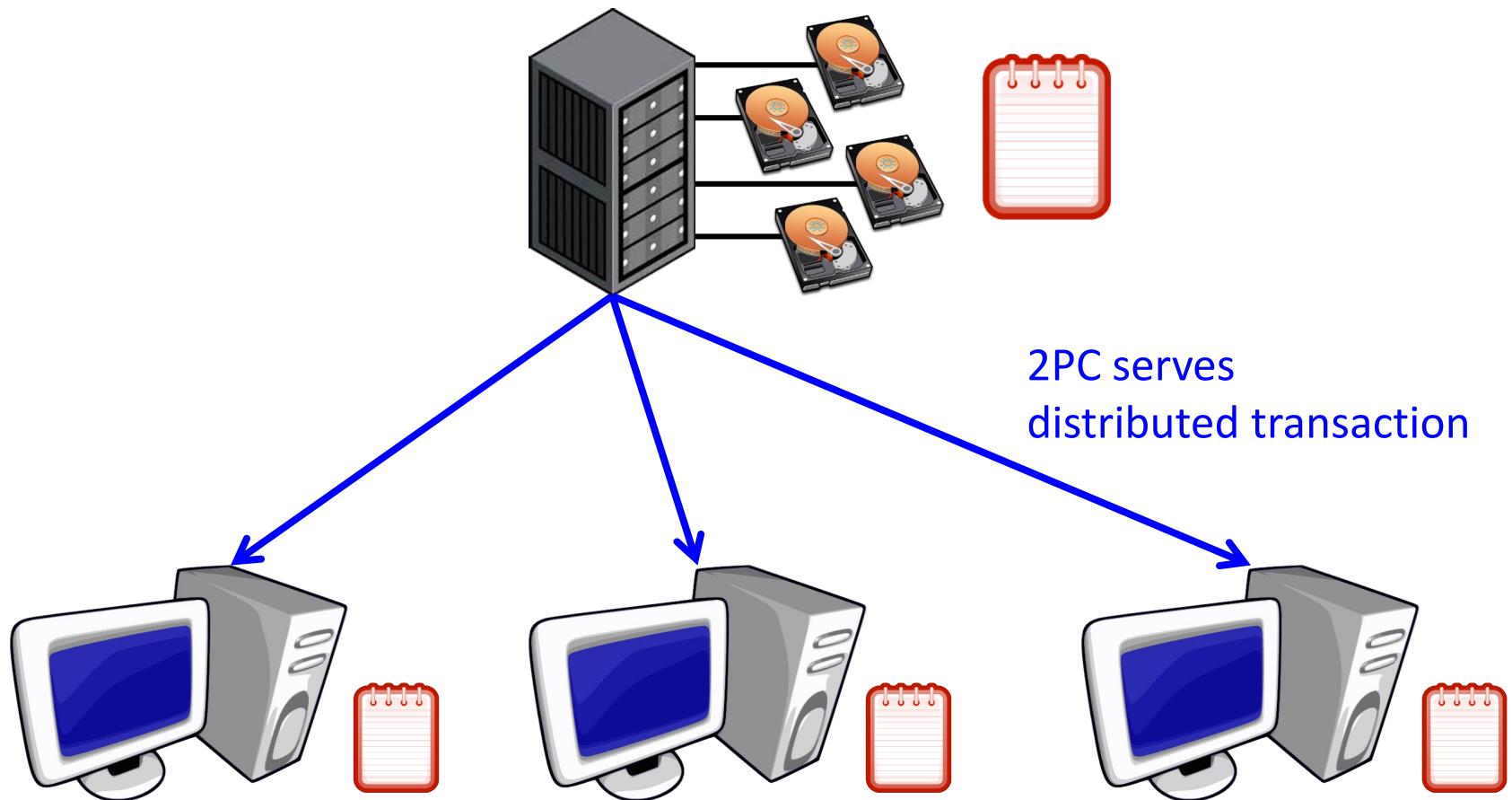
Properties	
Agreement	All processes that reach a decision should have the same state
Finality	A process cannot reverse its decision after the process has reached one
Consistency	If there is no failure, and every process votes COMMIT, the decision for the system will be with COMMIT (global)
Termination	If all failures are repaired, and there is no failure any more, all processes will eventually reach a consensus on COMMIT or ABORT.

[Summary] 2PC

If a file is updated, we can let multiple clients for the update and make them consensus on the update by using the 2PC protocol



[Summary] 2PC



What's Next?

- Let's check more about Distributed File System (e.g., mounting and file/directory services with distributed transaction)

21S EE415

Distributed File-System II

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE



Distributed File-System II

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMEL*ab*



REMIND: Final Exam

- Untact (non-face-to-face) Exam
- **Schedule:** Wednesday (**6/16**) 1:00 PM – 3:45 PM (this is what the school *officially* assigned for this lecture. No change or exception)
- **Scope:**
Lecture 13 ~ Lecture 25

Notice

Please check the detailed guidelines at KLMS

Guideline for Untact (non-face-to-face) Exam

작성자 : 권미령

작성일 : 2021년 4월 12일(월요일) 오후 3:23 조회수 : 95

Schedule

Wednesday (6/16) 1:00 PM - 3:45 PM

- Closed book
- No early answer sheet submission (You should stay Zoom until exam end)
- If you have any questions on the Exam, please type your question in English via Zoom chatting. Korean does not allow.

REMIND: Final Exam

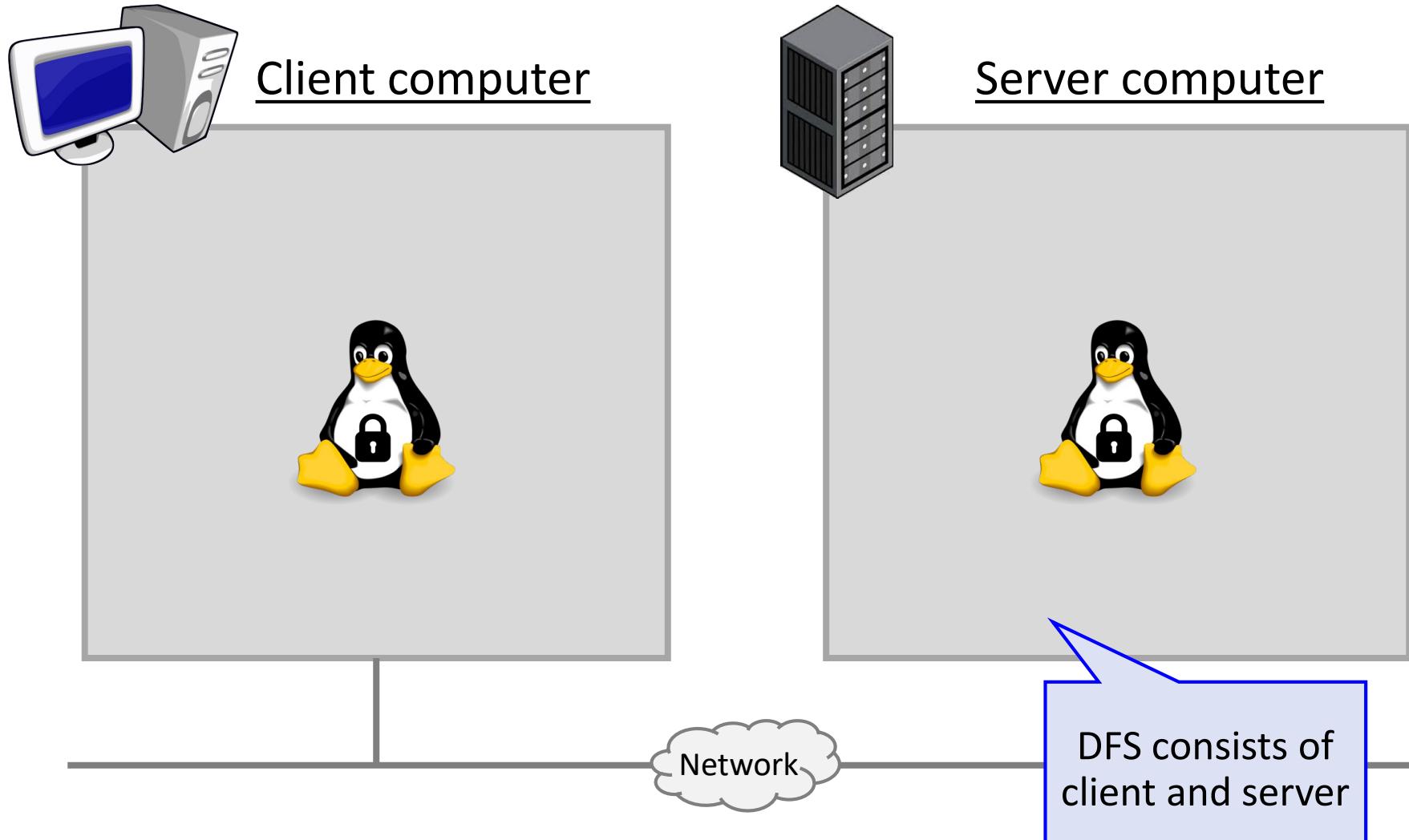
Student ID	Name	TA
20110273	김성구	Hanyeoreum Bae
20140300	송영준	Hanyeoreum Bae
20150114	김동언	Hanyeoreum Bae
20150816	홍지승	Hanyeoreum Bae
20160265	박종회	Hanyeoreum Bae
20160466	이선호	Hanyeoreum Bae
20170327	서준석	Hanyeoreum Bae
20170353	신경섭	Hanyeoreum Bae
20170413	우상민	Hanyeoreum Bae
20170534	임석빈	Hanyeoreum Bae
20170628	조우석	Hanyeoreum Bae
20170651	자승훈	Hanyeoreum Bae
20180046	김경민	Hanyeoreum Bae
20180498	이준	Miryeong Kwon
20180545	임진영	Miryeong Kwon
20180662	자준서	Miryeong Kwon
20180710	한승윤	Miryeong Kwon
20190028	고형석	Miryeong Kwon
20203234	박상수	Miryeong Kwon
20204409	오동석	Miryeong Kwon
20208252	JULIAN TROY VALDEZ	Miryeong Kwon
20213007	강승관	Miryeong Kwon
20213408	윤정한	Miryeong Kwon
20214207	RAJENDRAN PRAVEEN KUMAR	Miryeong Kwon
20215356	최진수	Miryeong Kwon

Please double check your TA
for as follows and use your
TA's Zoom link for the exam

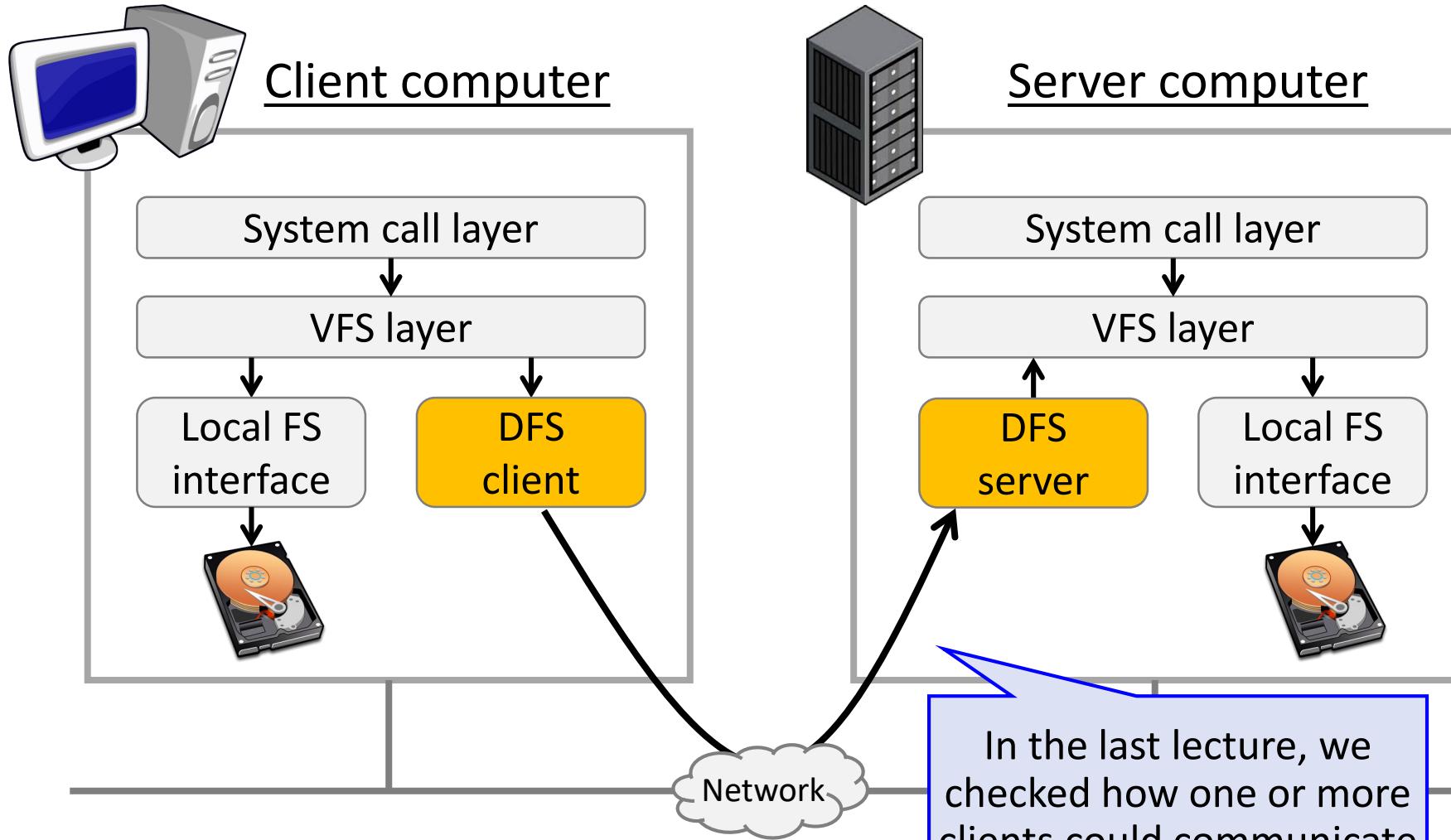
(All these have been also
announced in KLMS)

<https://docs.google.com/spreadsheets/d/1p3QuNjpd-2NeGyPGCCgh1on2TDPCAjY-mGThMXeGzI8/edit#gid=0>

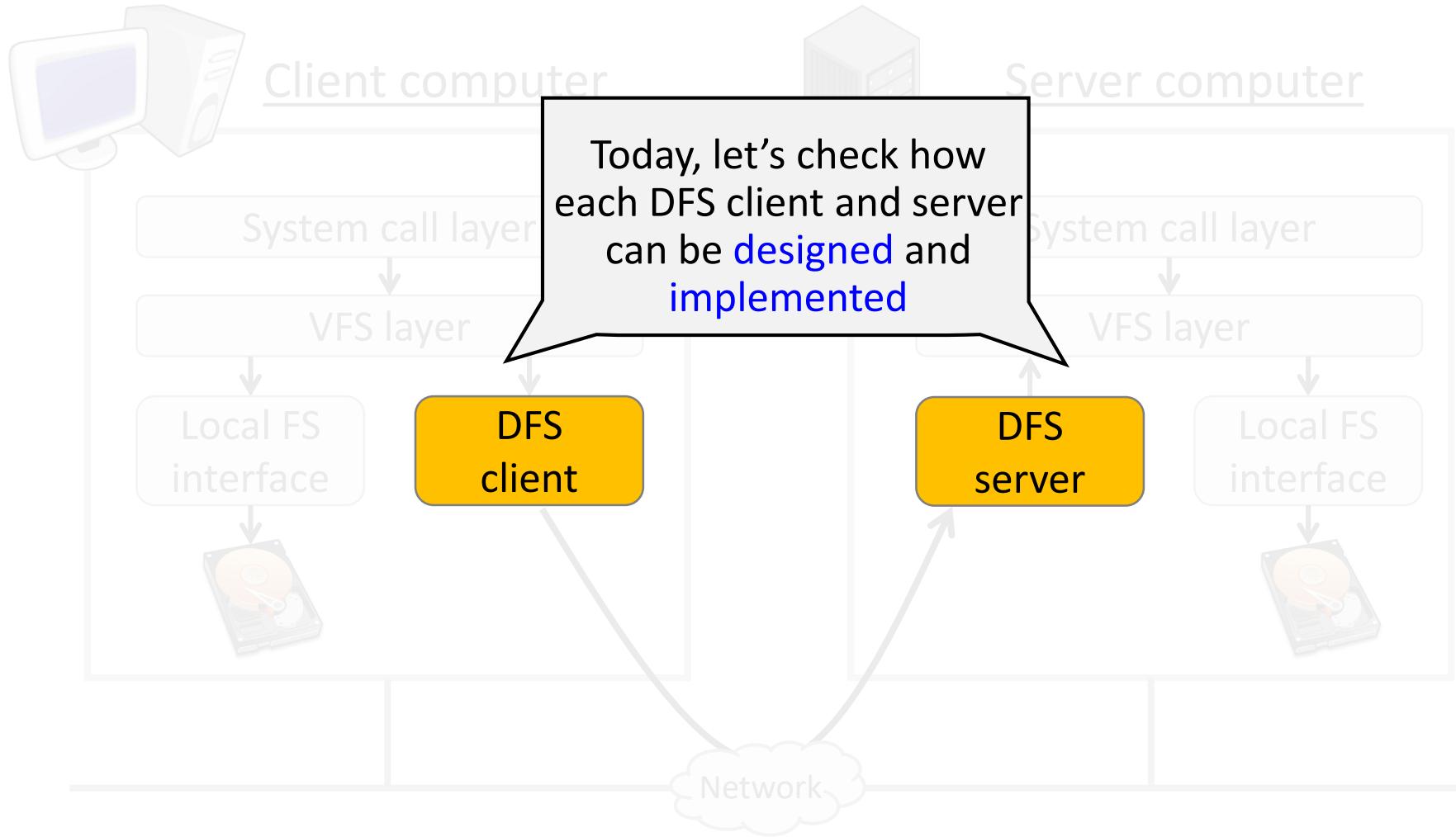
[Recap] Distributed File-System Stack



[Recap] Distributed File-System Stack

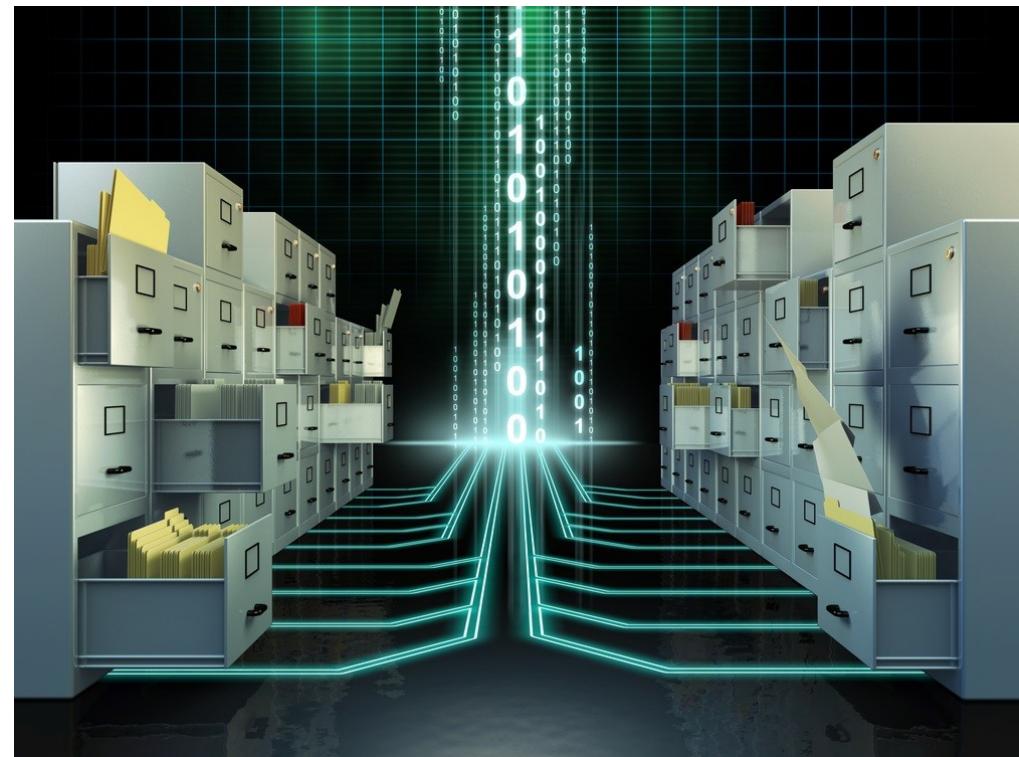


[Recap] Distributed File-System Stack



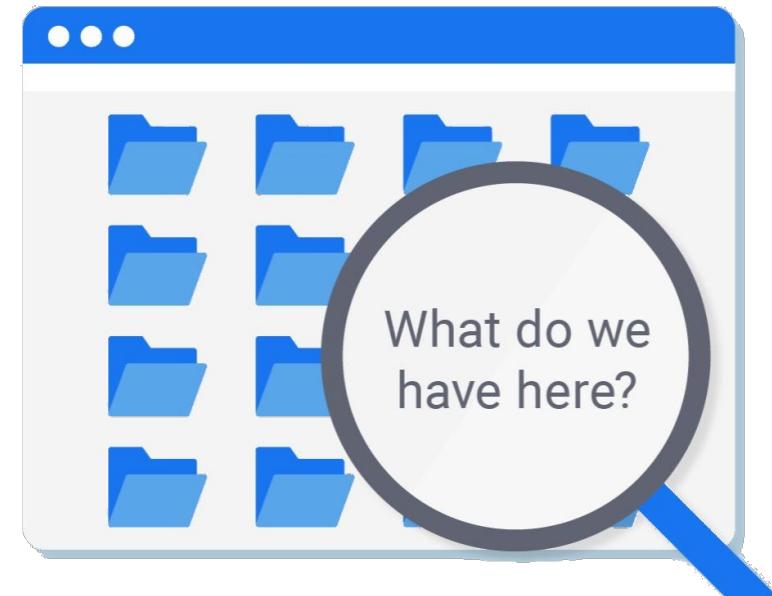
DFS Design

DFS should support both
file and directory services.
Let's design them

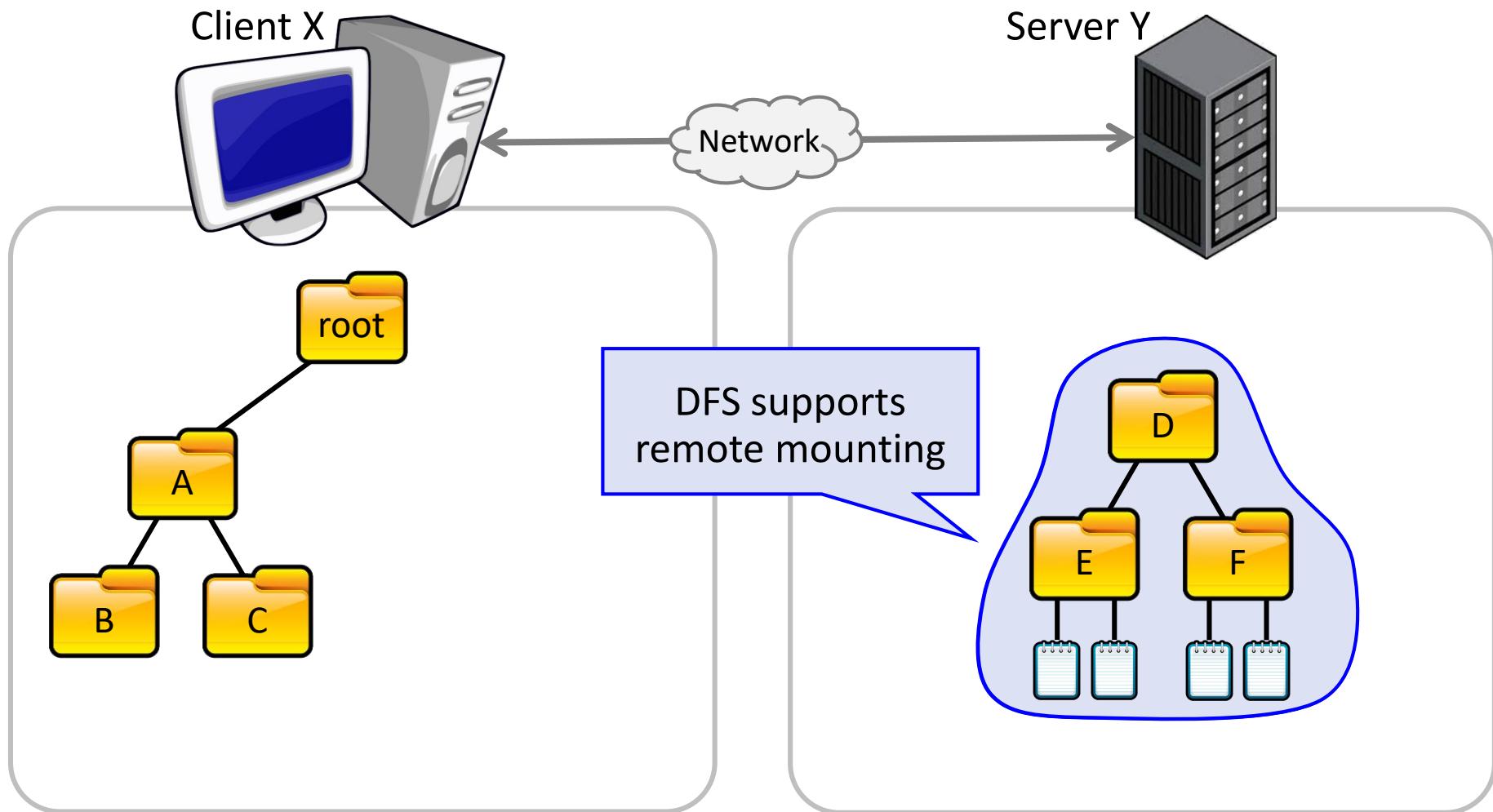


Directory Service

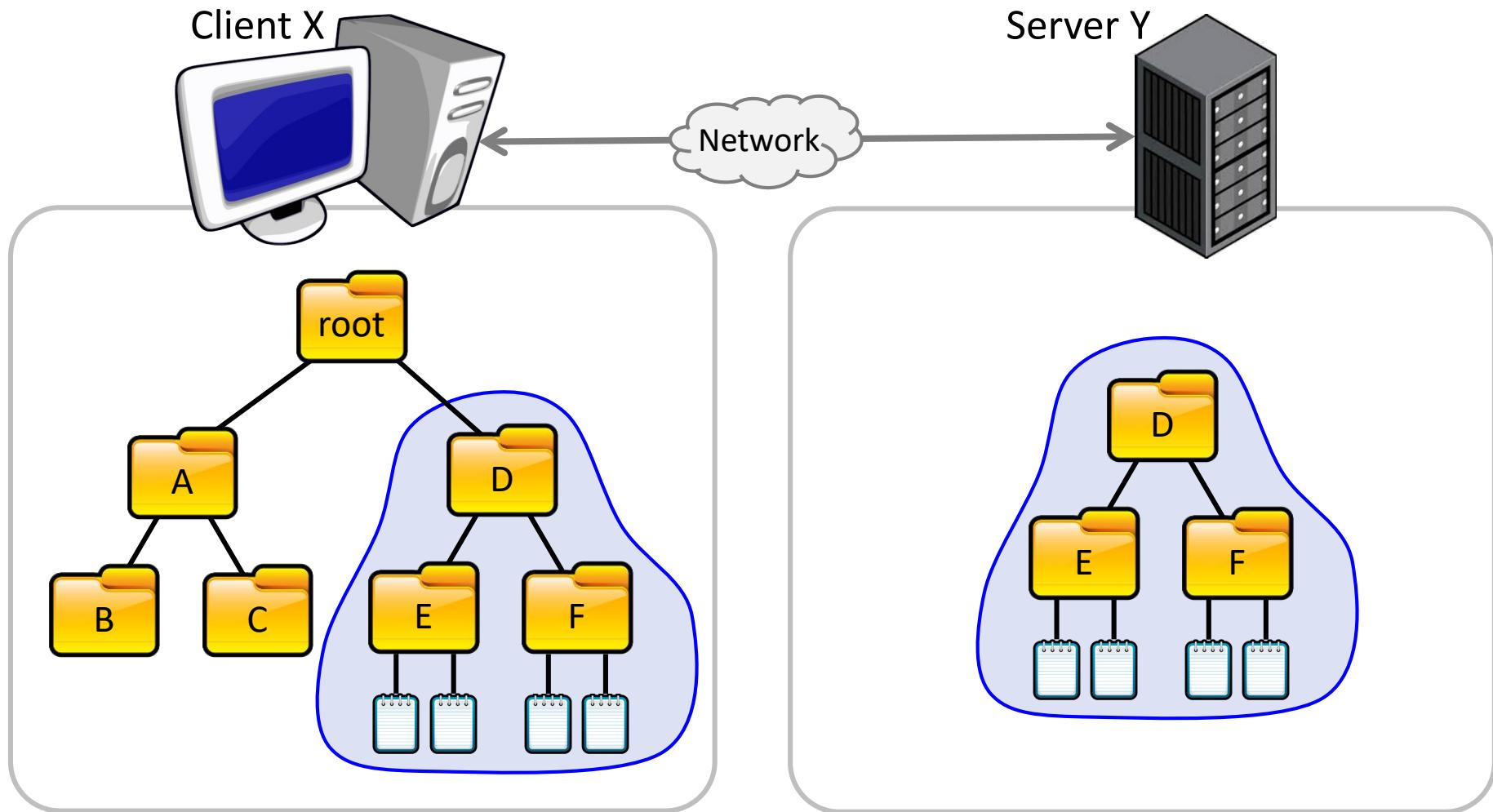
How could the client
know where the
remote file is?



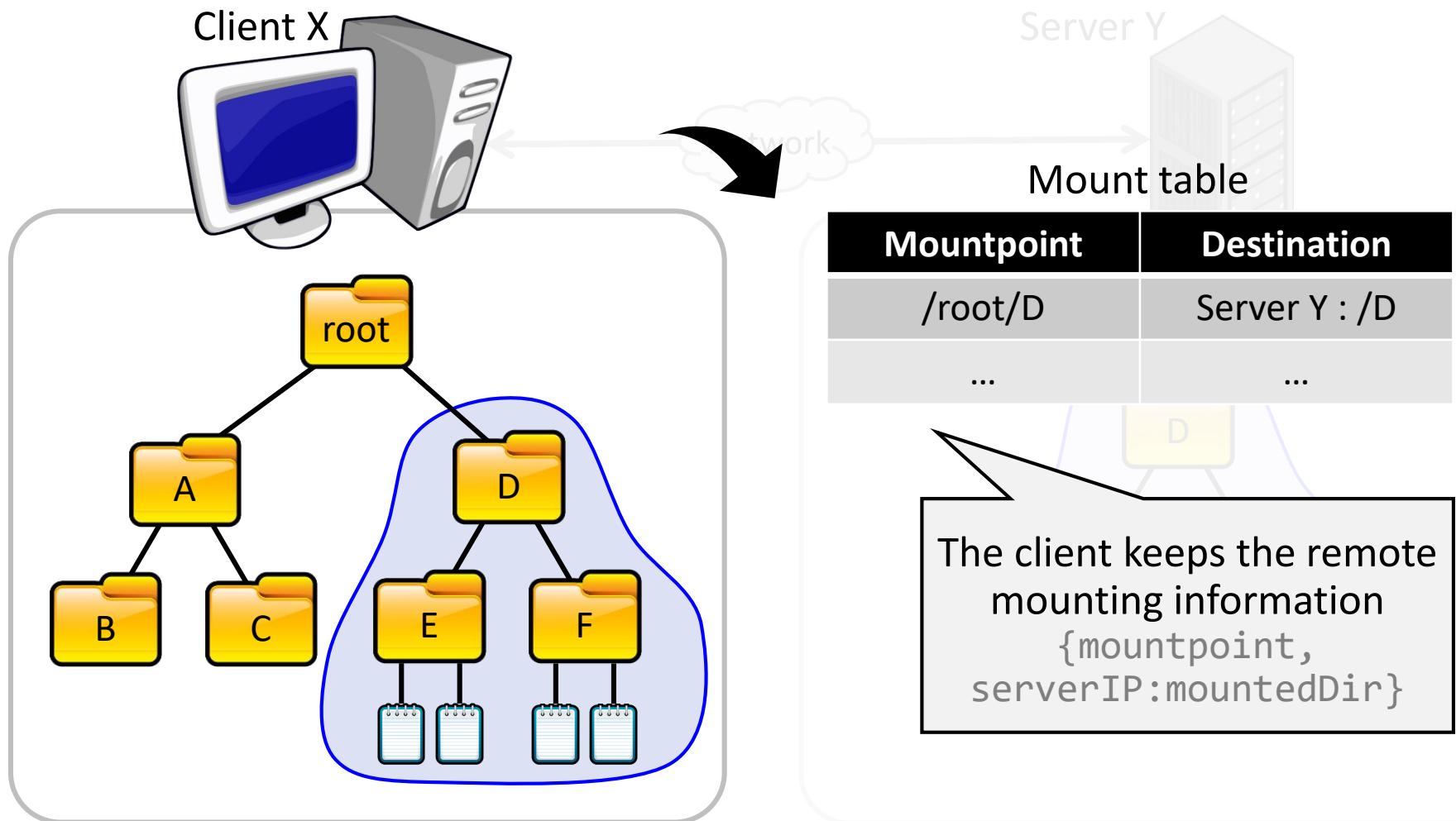
Remote Mounting



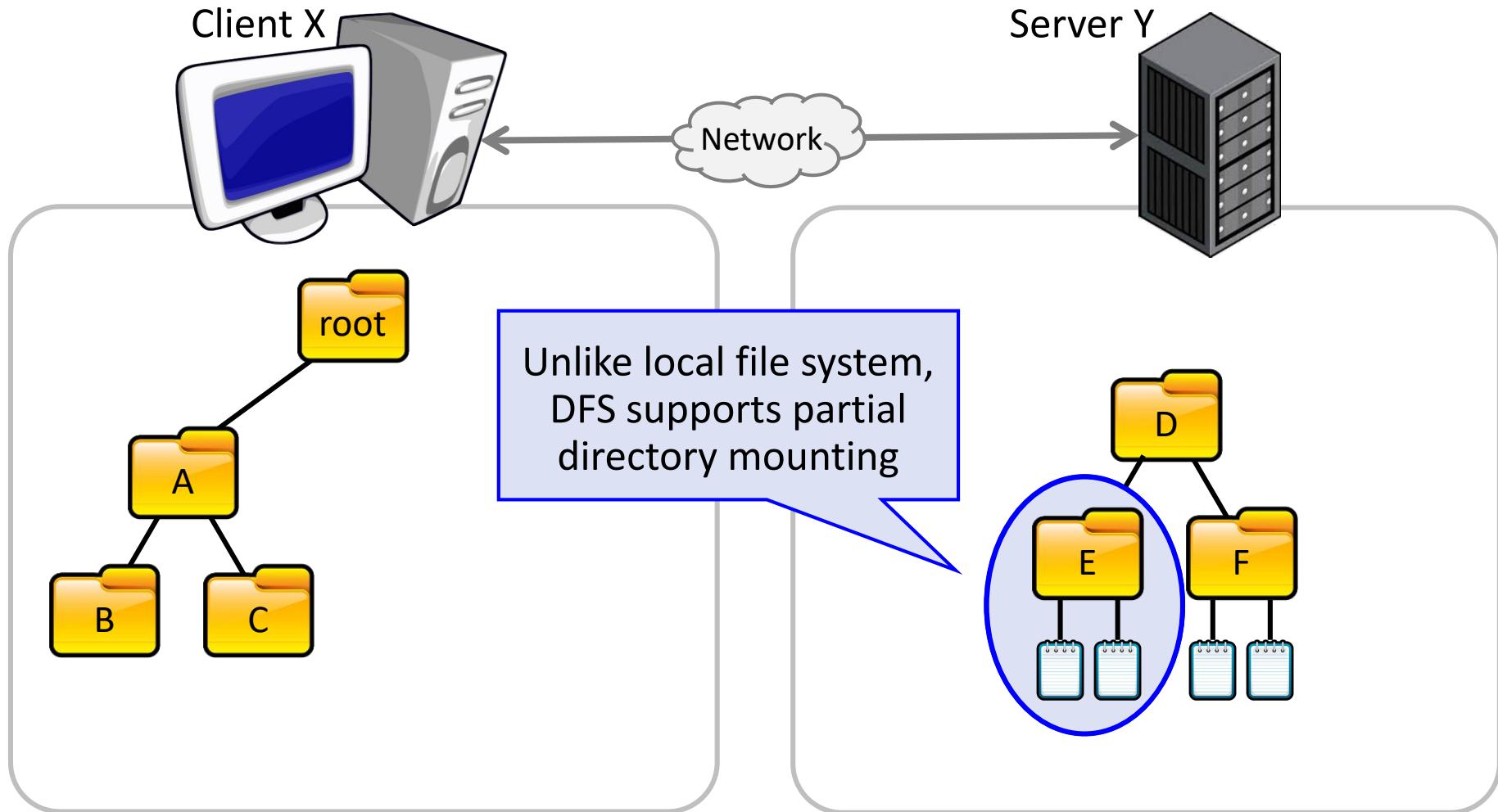
Remote Mounting



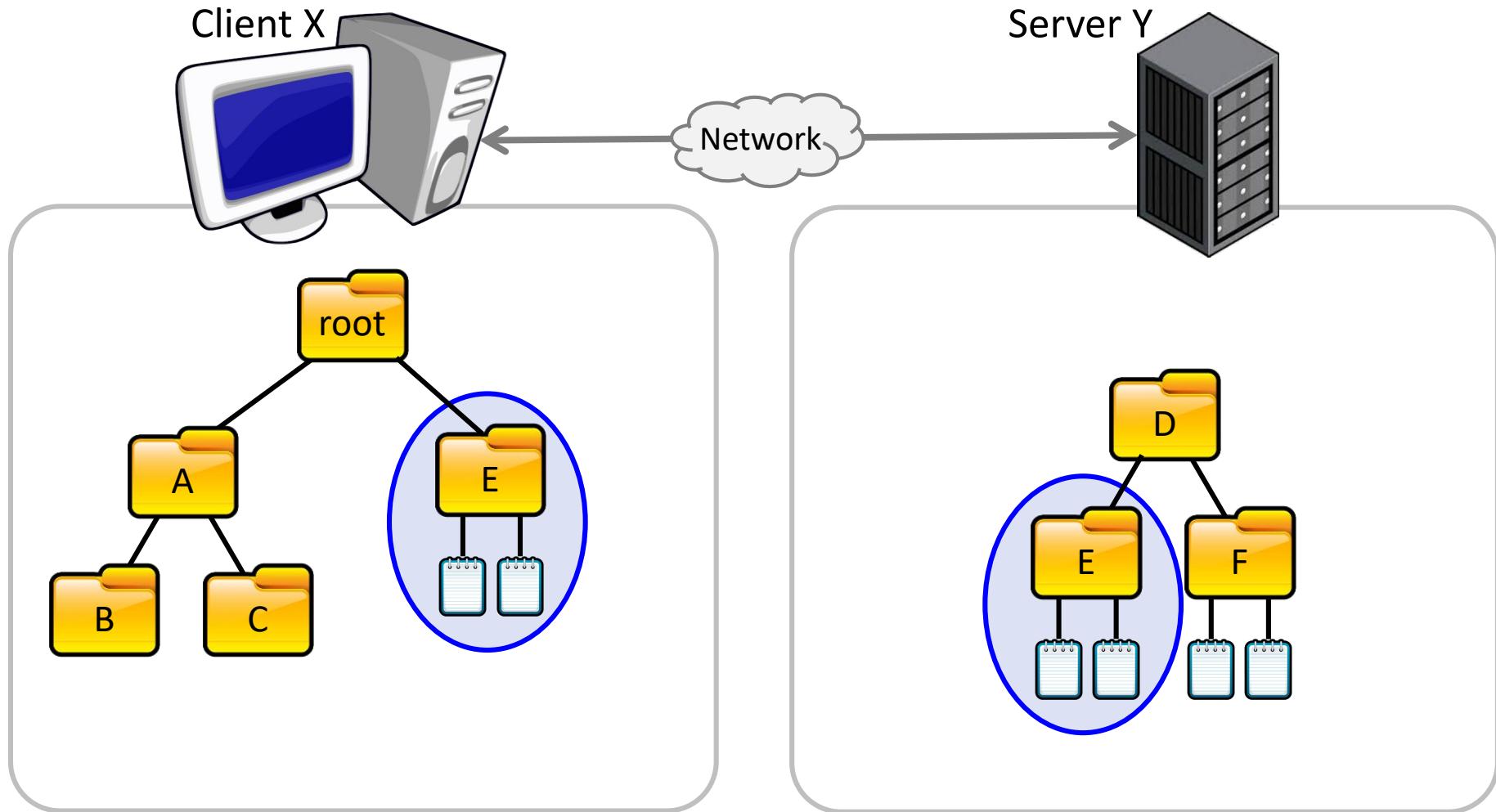
Remote Mounting



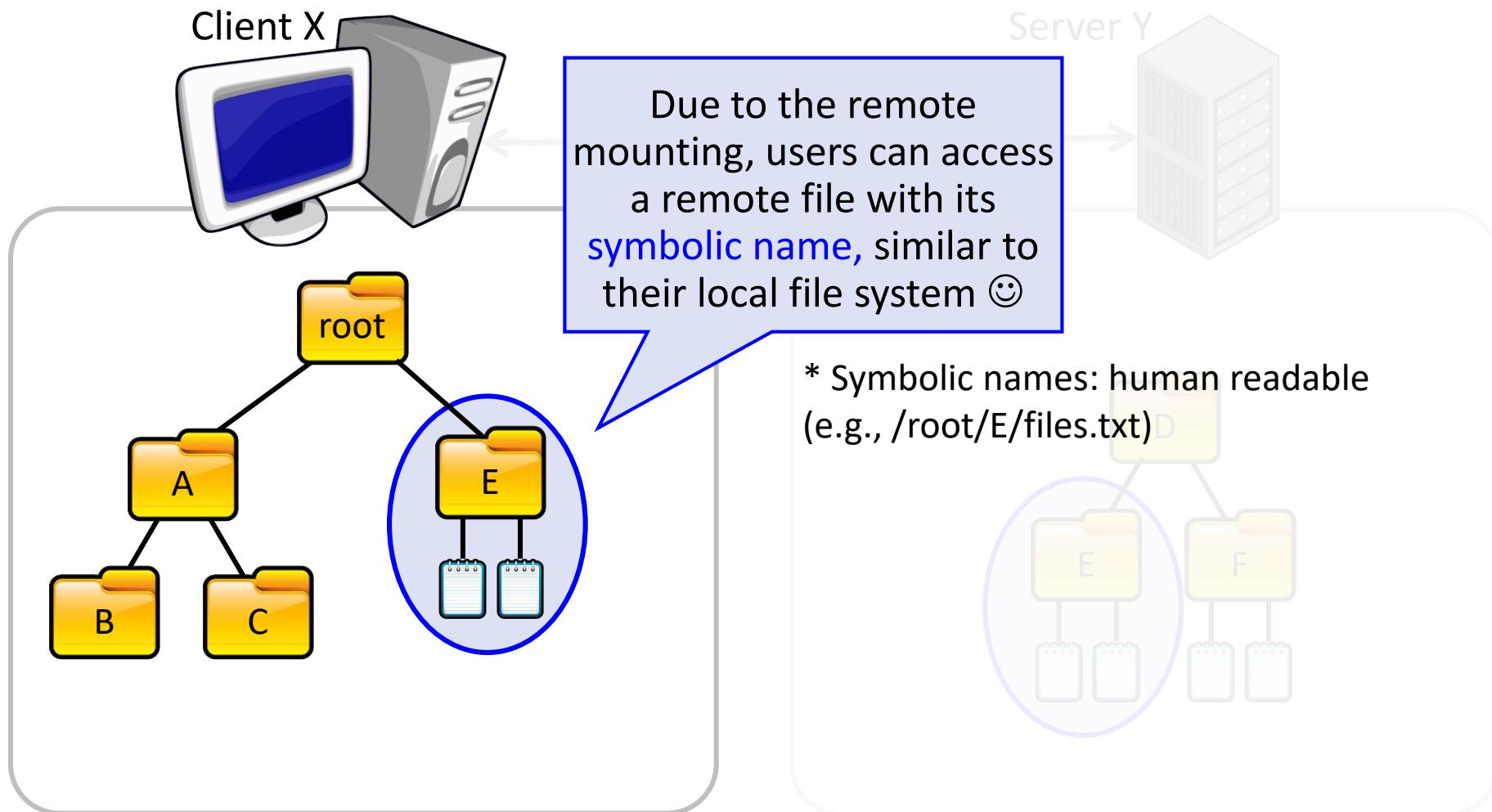
Remote Mounting



Remote Mounting

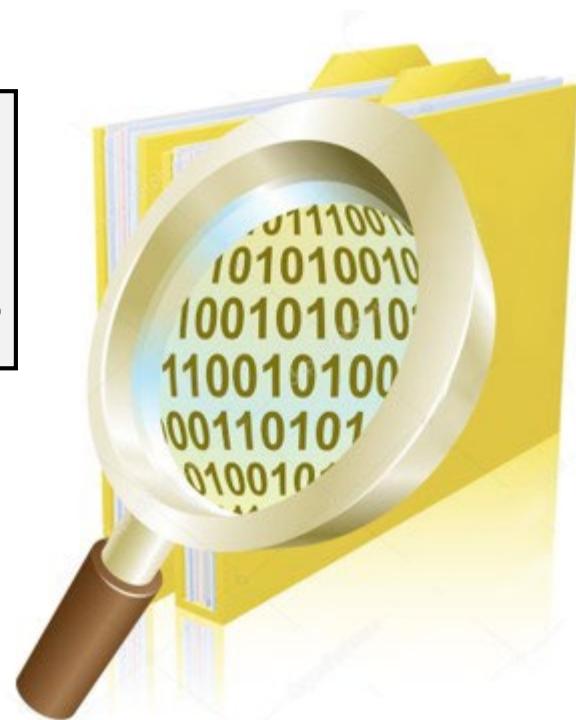


Remote Mounting

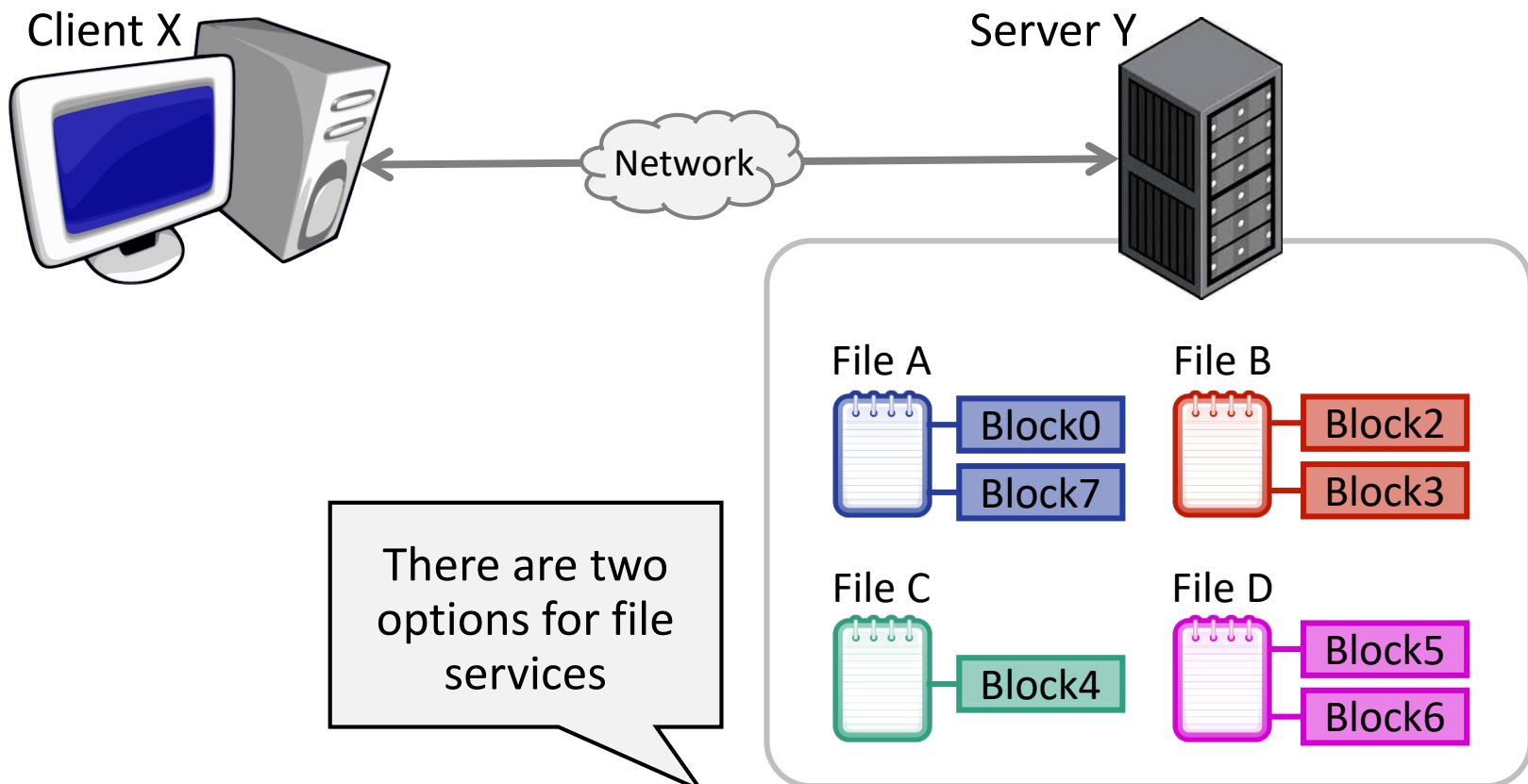


File Service

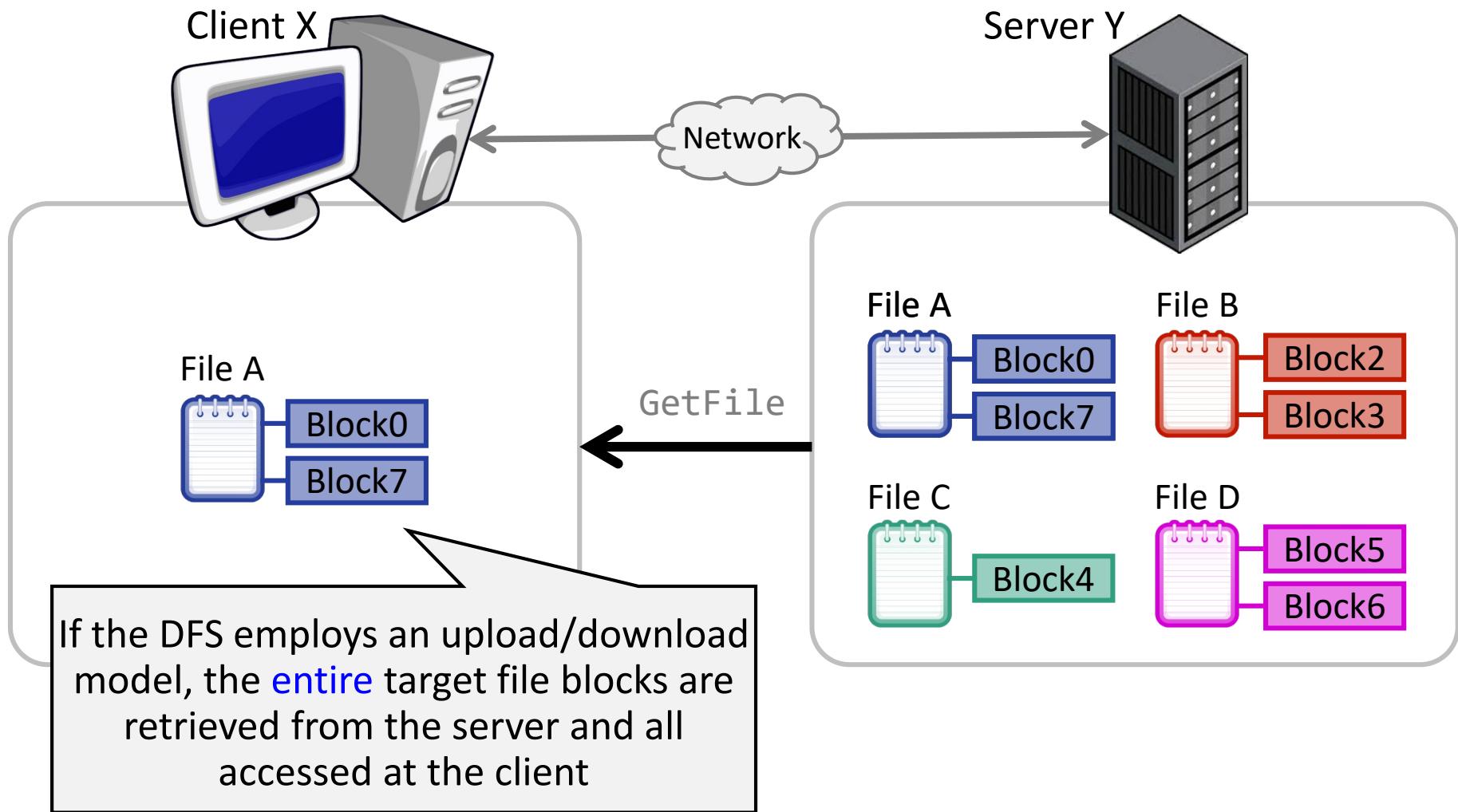
After retrieving the target file location with the assistance of directory service, how does the client actually read/write the file?



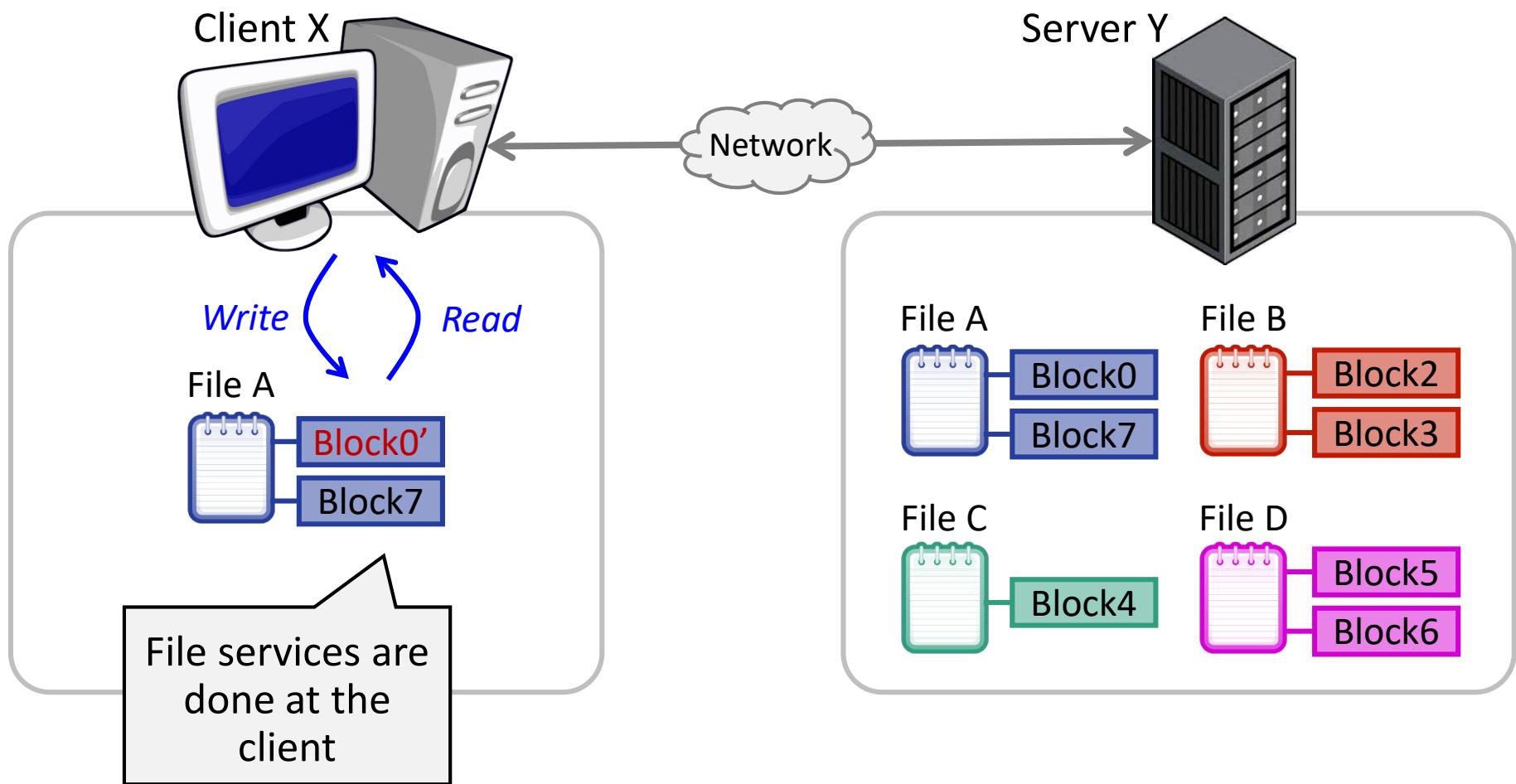
File Service Interface



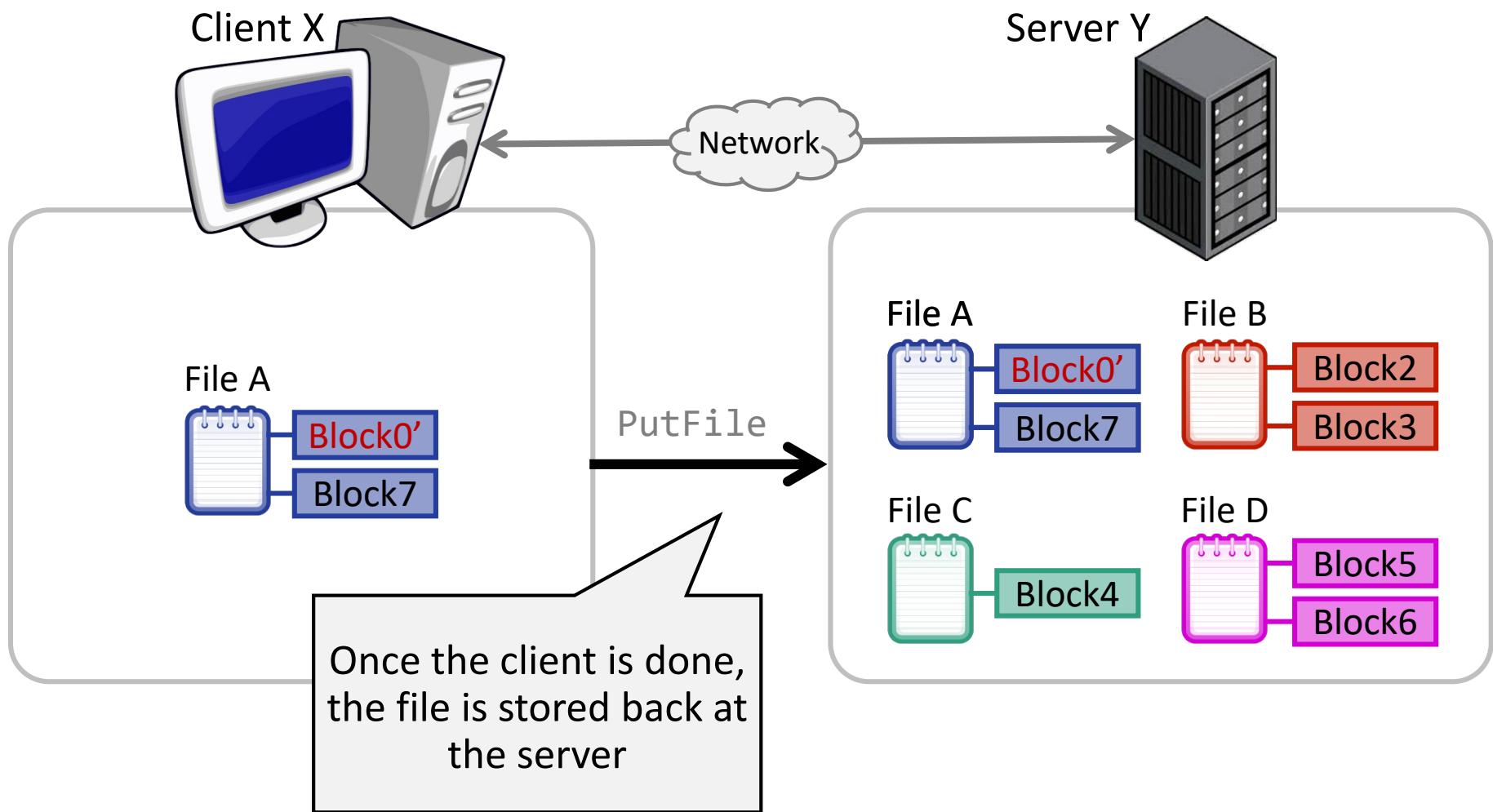
Option #1: Upload/Download Model



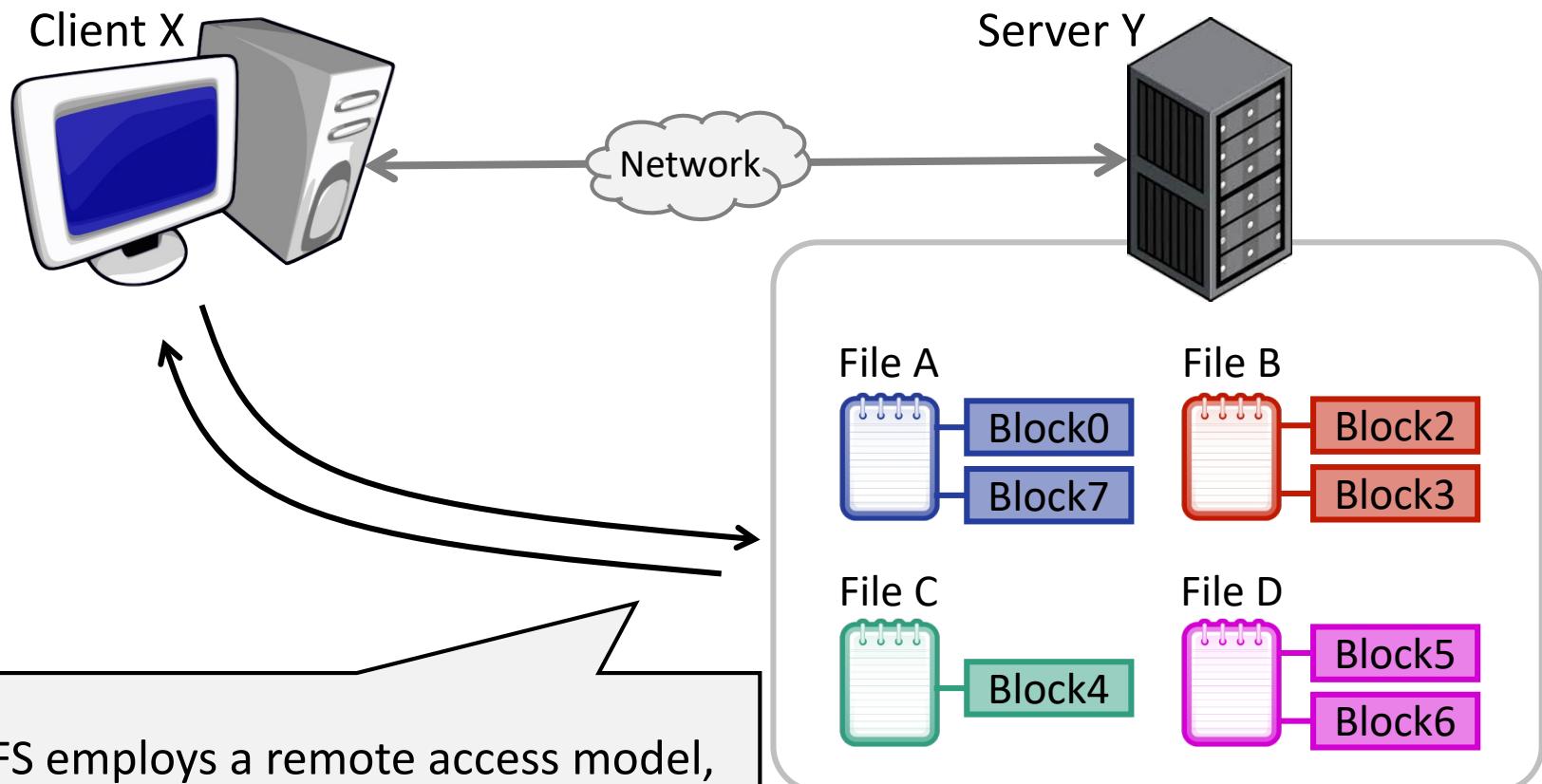
Option #1: Upload/Download Model



Option #1: Upload/Download Model

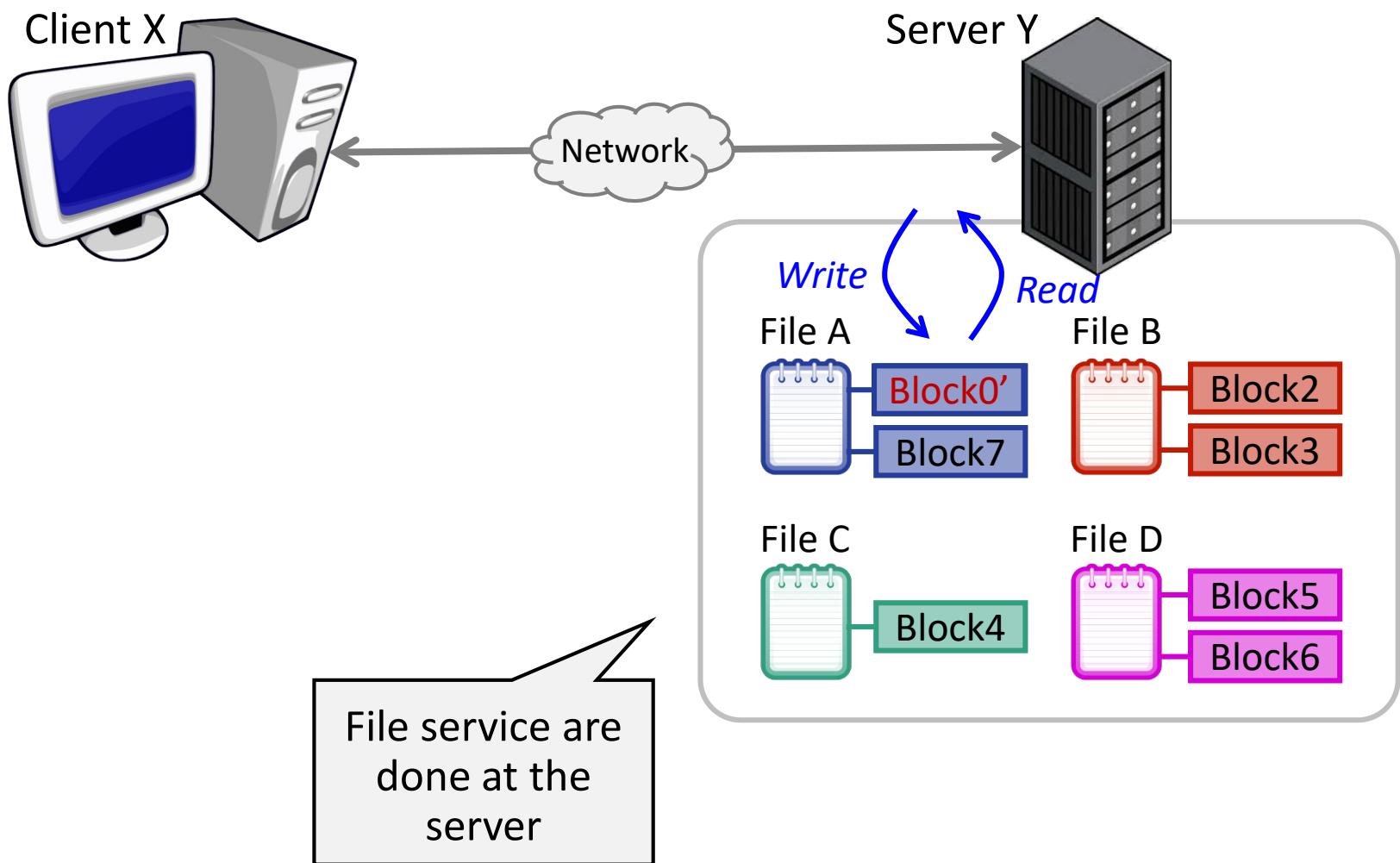


Option #2: Remote Access Model



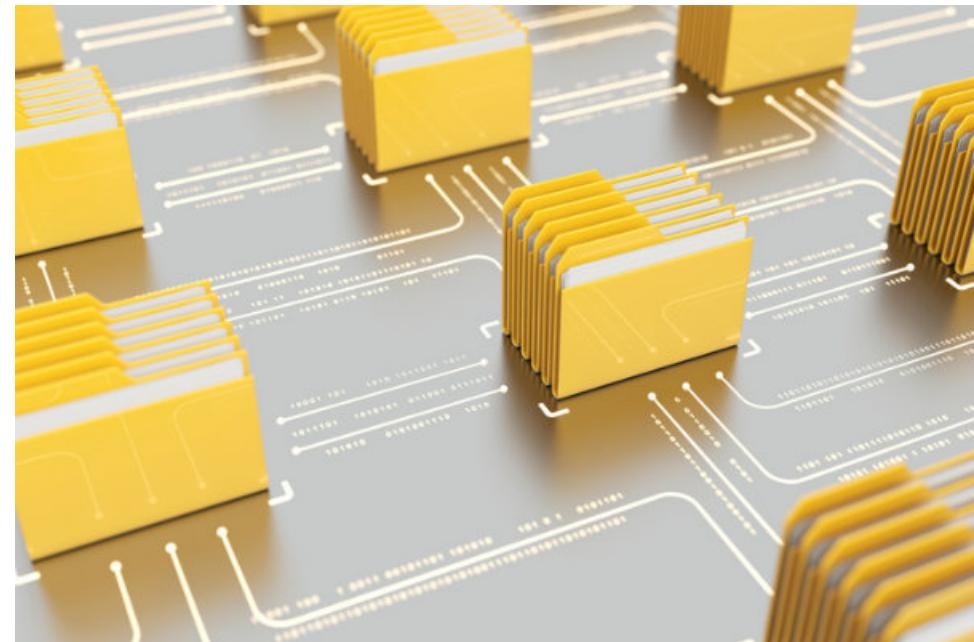
If the DFS employs a remote access model,
the client need to send requests to access the
target file from the server

Option #2: Remote Access Model



DFS Implementation

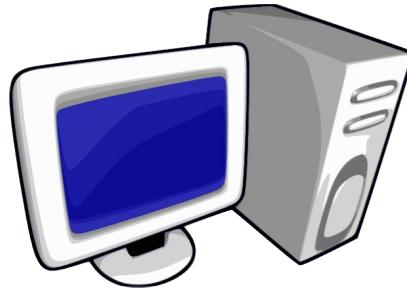
Let's implement DFS from now on. Note that, there are several **requirements** that DFS has to satisfy



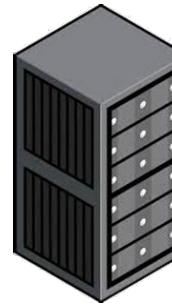
DFS Requirements

We can classify them into two: i) client-side and ii) server-side requirements

Client-side

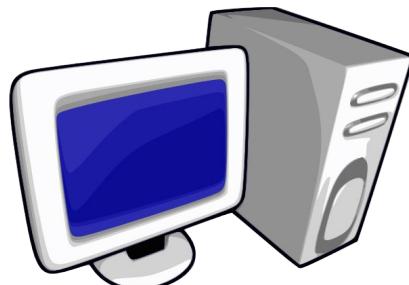


Server-side

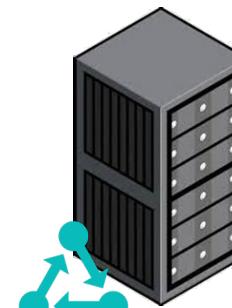


DFS Requirements

Client-side



Server-side

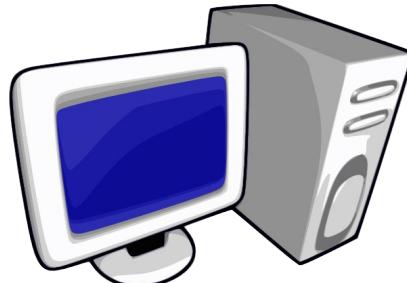


Server' Connectivity: The server should support multiple clients and manage their information (states)

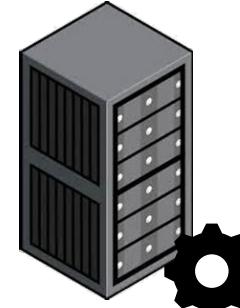
Connectivity

DFS Requirements

Client-side



Server-side

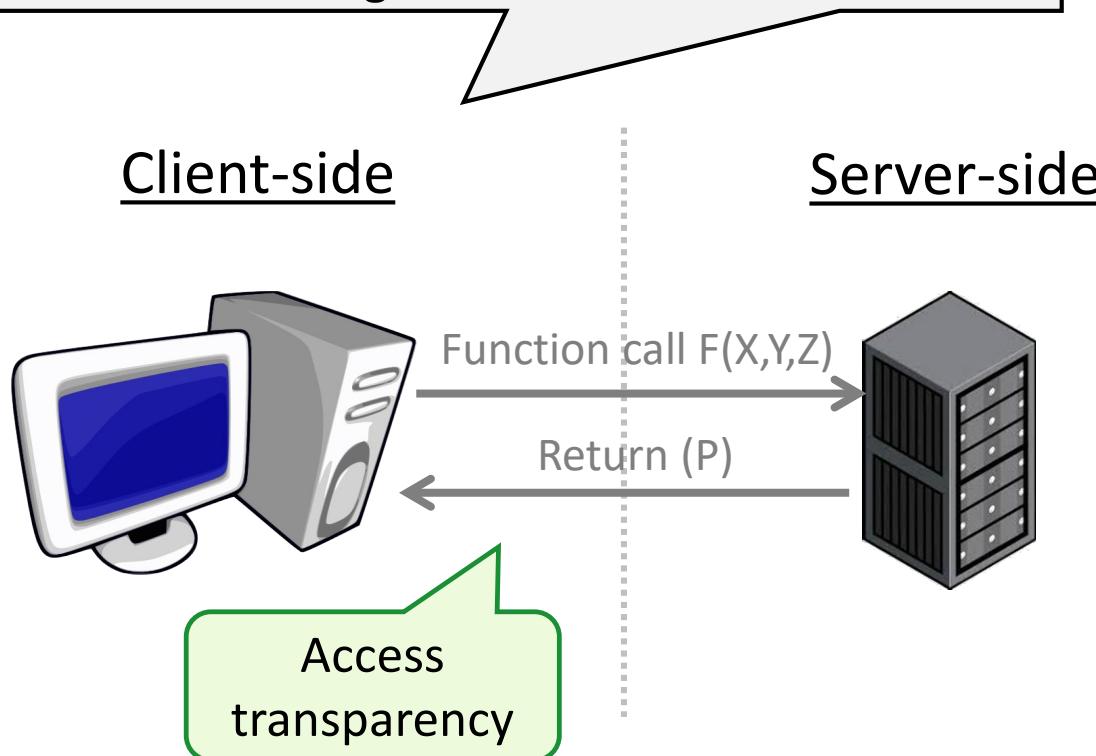


Server' Availability: The server and data (files) therein should be always reachable and accessible

Availability

DFS Requirements

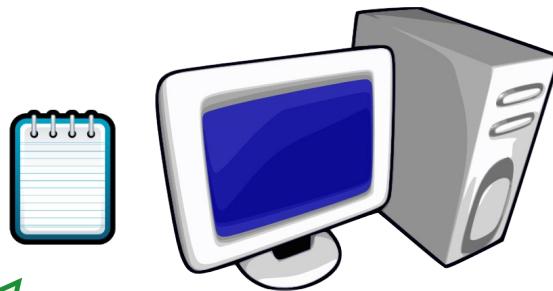
Client' Access Transparency: The client provides a set of interfaces (APIs) to users, which can hide the detail of the remote (network) accesses – the interface logic is the same as that of the local accesses. The users request something over a function call and get the results as its return



DFS Requirements

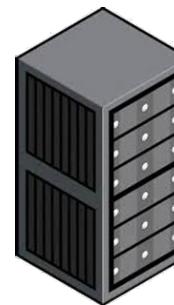
Client' Performance Transparency: There is no difference between local file accesses and remote (network) file accesses

Client-side



Performance transparency

Server-side



DFS Requirements

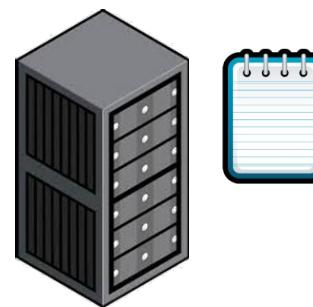
Client' Consistency: A file modification should be visible to all clients sharing the target file

Client-side

Consistency



Server-side

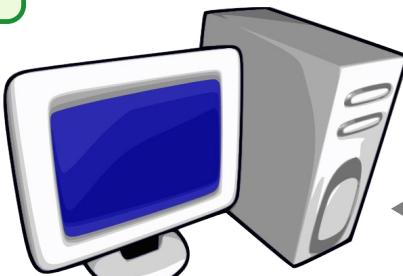


DFS Requirements

Let's check how DFS can satisfy these **requirements**, one by one

Client-side

Consistency



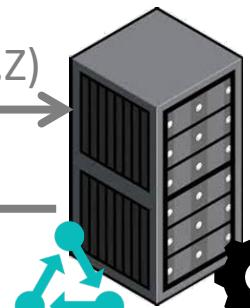
Performance transparency

Access transparency

Server-side

Function call $F(X,Y,Z)$

Return (P)

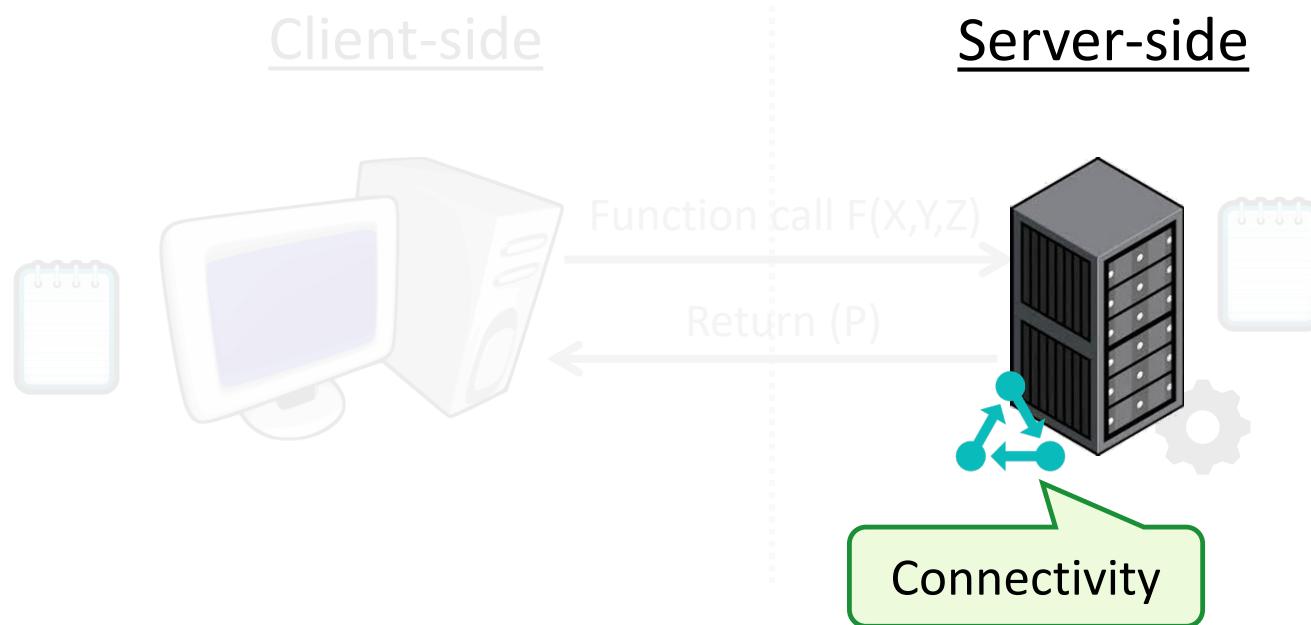


Connectivity

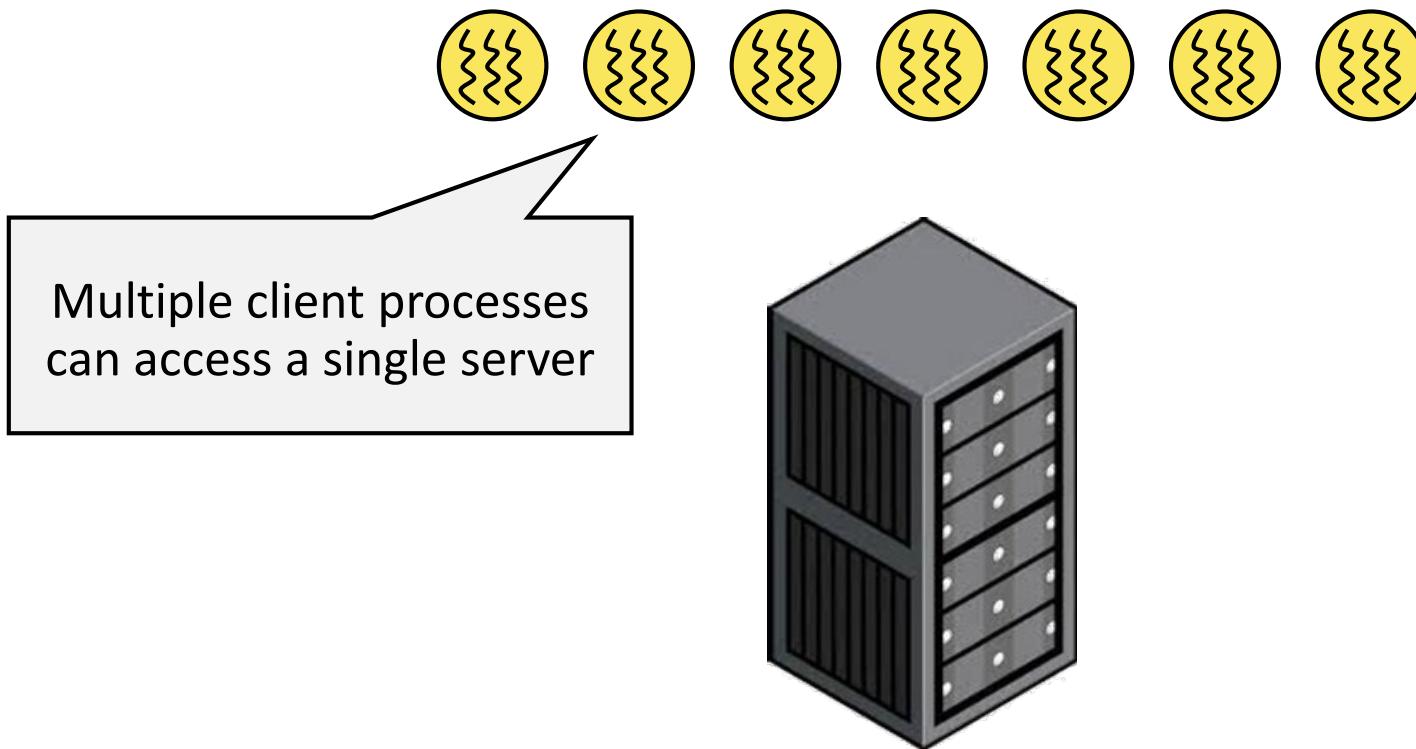


Availability

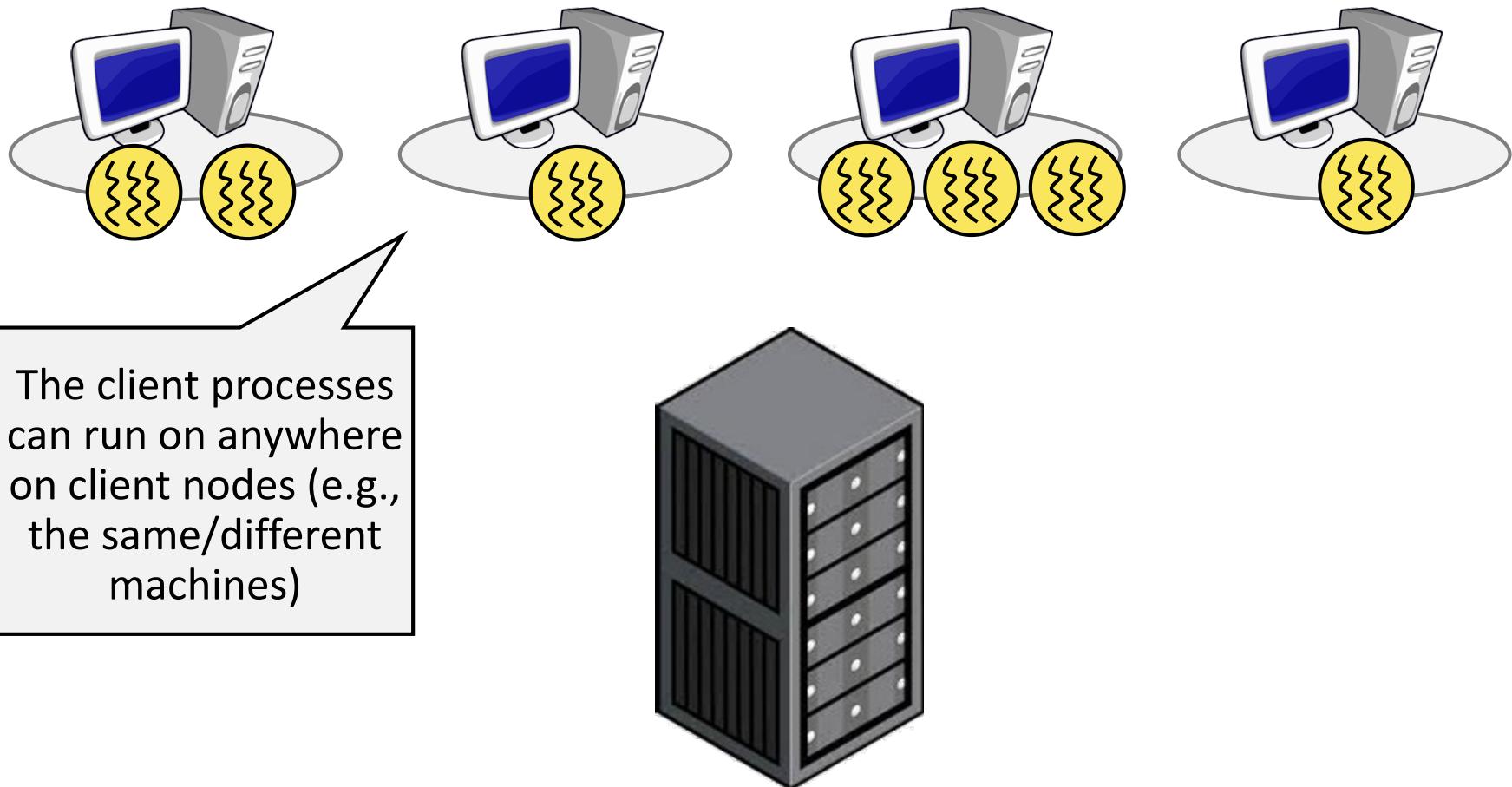
Server-Side #1: Connectivity



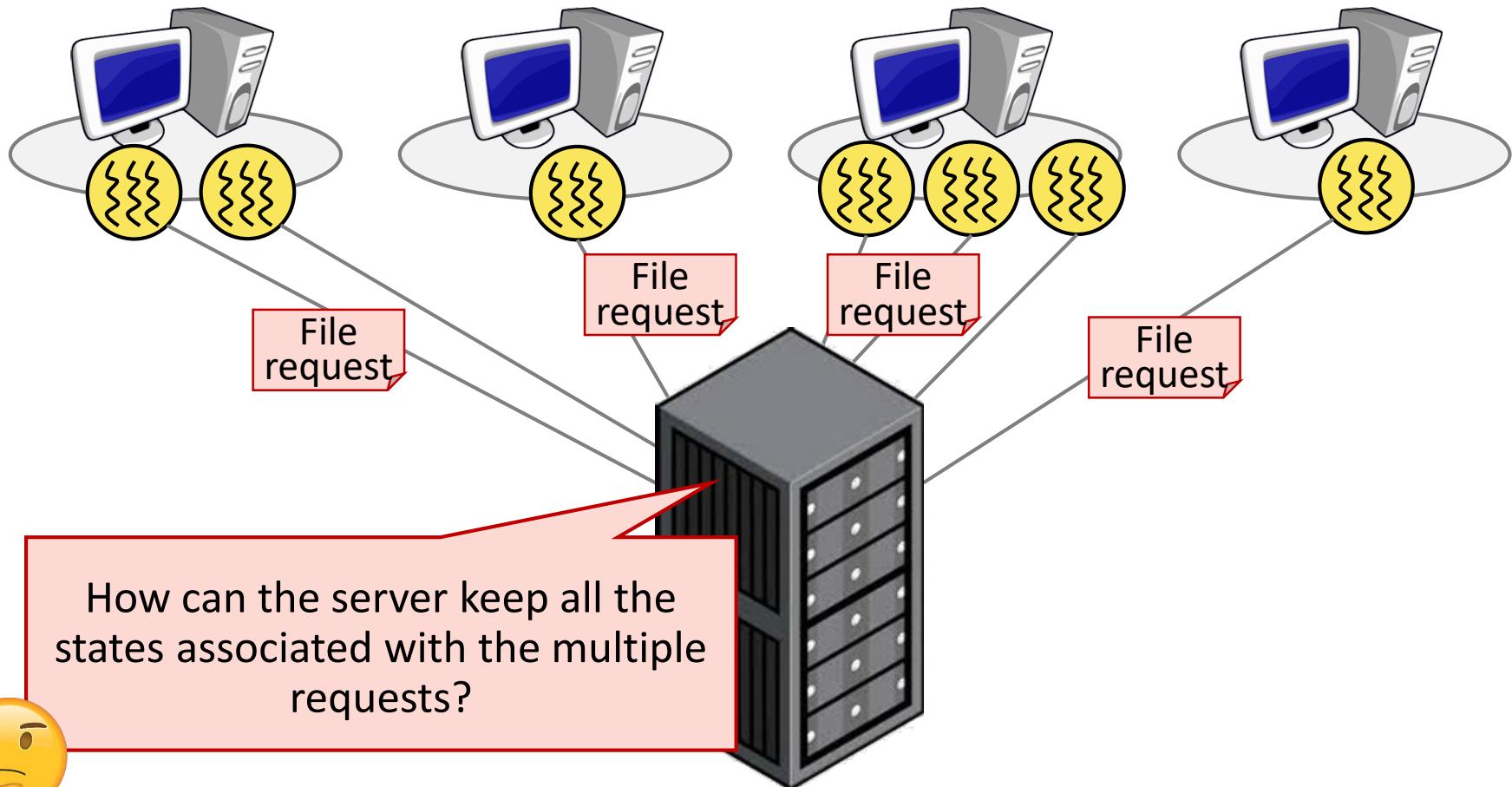
Multiple Clients (Connections)



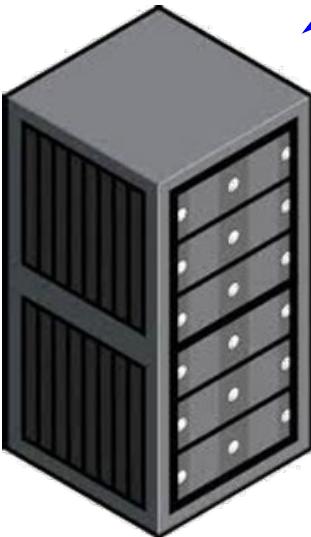
Multiple Clients (Connections)



Multiple Clients (Connections)



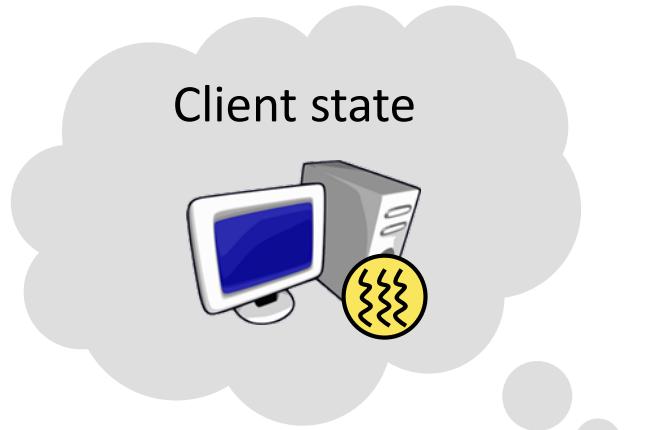
Implementation: State Info. Management



DFS server should have its own strategy for the client state management to support multiple clients (connections)

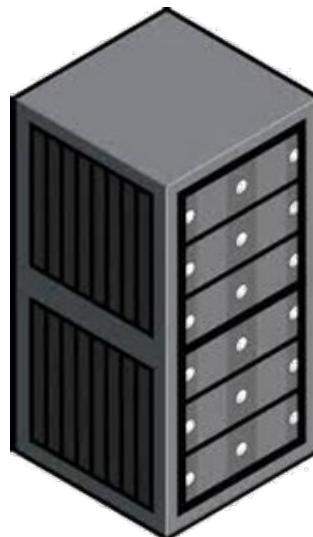
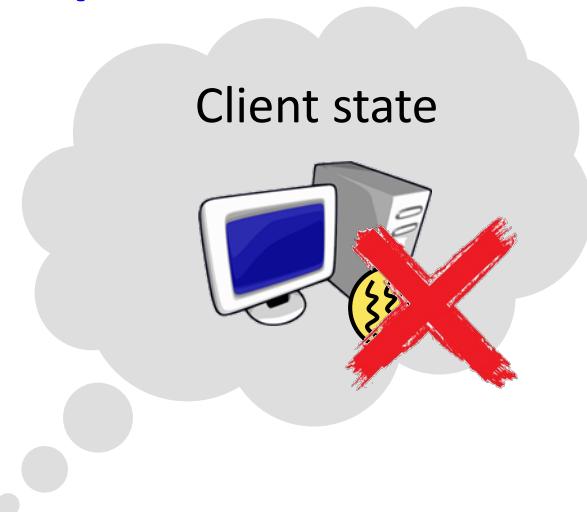
Implementation: State Info. Management

Option #1: Stateful



We can classify the DFS server types based on how its client states are managed

Option #2: Stateless



Stateful

- Long-term services
 - Online banking, email, and applications need to track history such as setting preferences, window location.
- Stateful processes' communication can be returned to again and again
 - They are performed with the context of previous transactions **and** the current transactions, which may be affected by what happened during previous transactions
 - If a stateful transaction is interrupted, the context and history **have been stored** so you can more or less pick up where you left off.

Stateless

- Short-term services
 - Content delivery network (CDN), web, or print servers
- A stateless process or application can be understood in isolation.
 - There is **no stored knowledge** of or reference to past transactions.
 - Each transaction is made as if **from scratch** for the first time.
 - Stateless applications provide one service or function

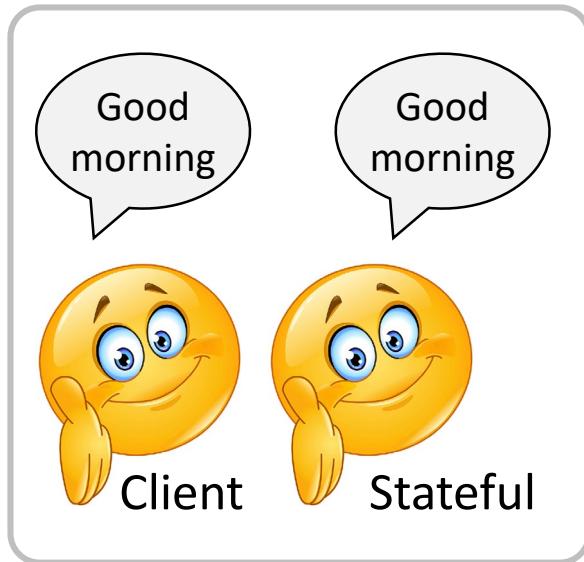
Option #1: Stateful Servers

We can simply understand what “stateful” does mean by with this good morning example



Option #1: Stateful Servers

When the client says “Good morning”, the stateful server responds w/ “Good morning” and keeps the client state as “**after initialization**”



Option #1: Stateful Servers

The stateful server can respond something more by considering the current client' state



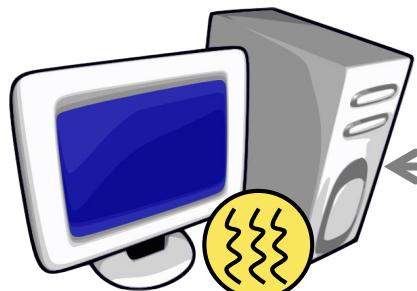
Stateful Server in DFS

When the client request is arrived,
the stateful server performs
directory and file services

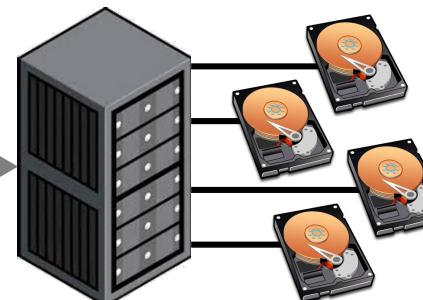
The /a/b/c/data.txt is at
block10 and inode is 100

Open /a/b/c/data.txt

Open /a/b/c/data.txt



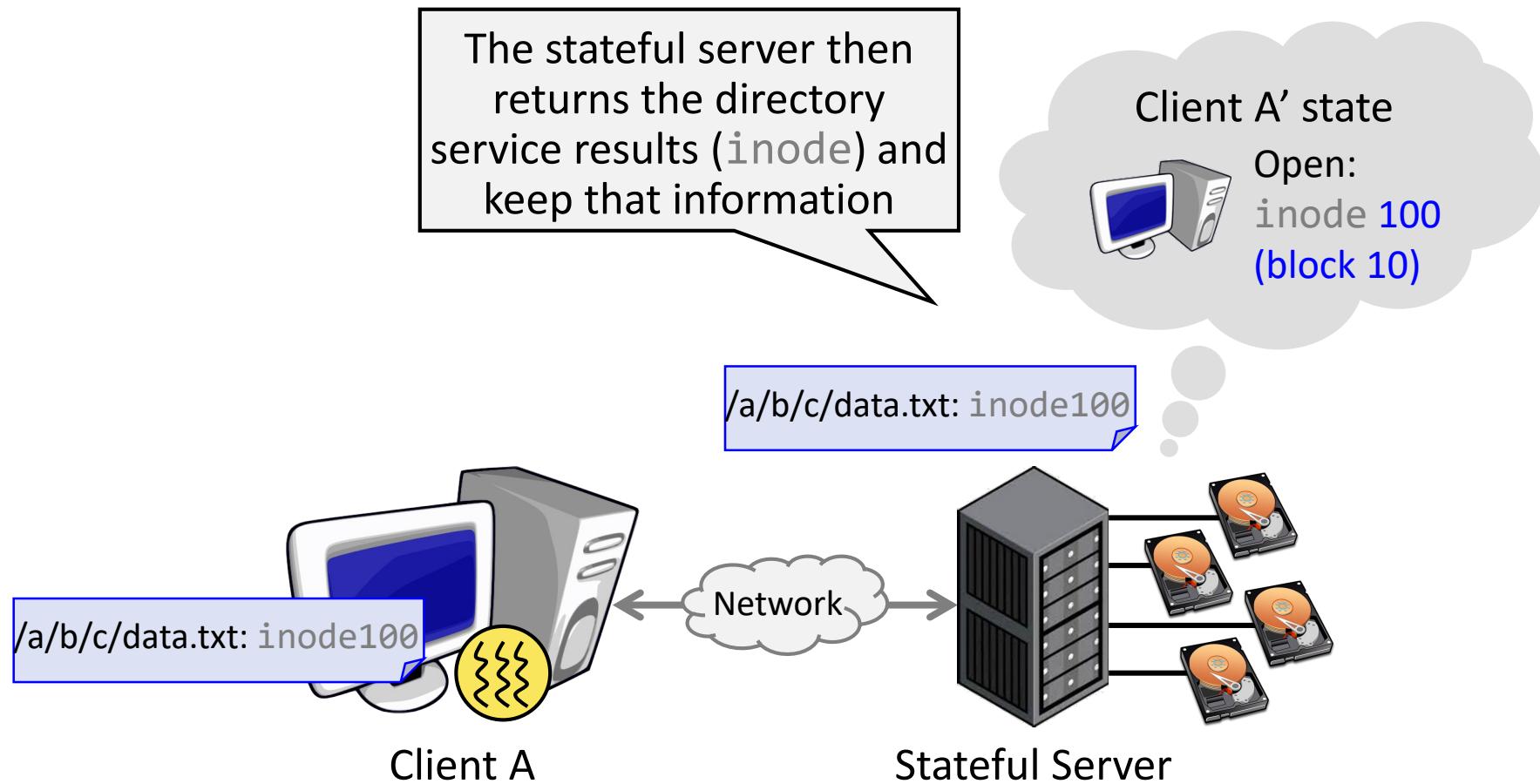
Client A



Stateful Server



Stateful Server in DFS



Stateful Server in DFS

Except for the “file open”,
the other requests
associated with the file (e.g.,
read/write) need only **inode**

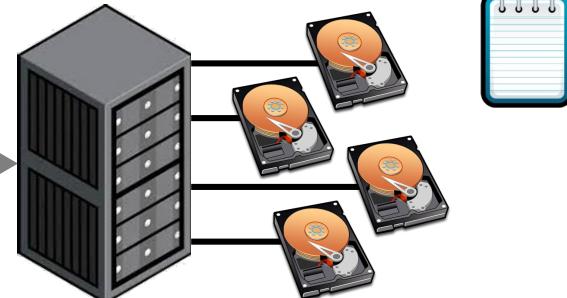
Read **inode 100**

/a/b/c/data.txt: inode100



Client A

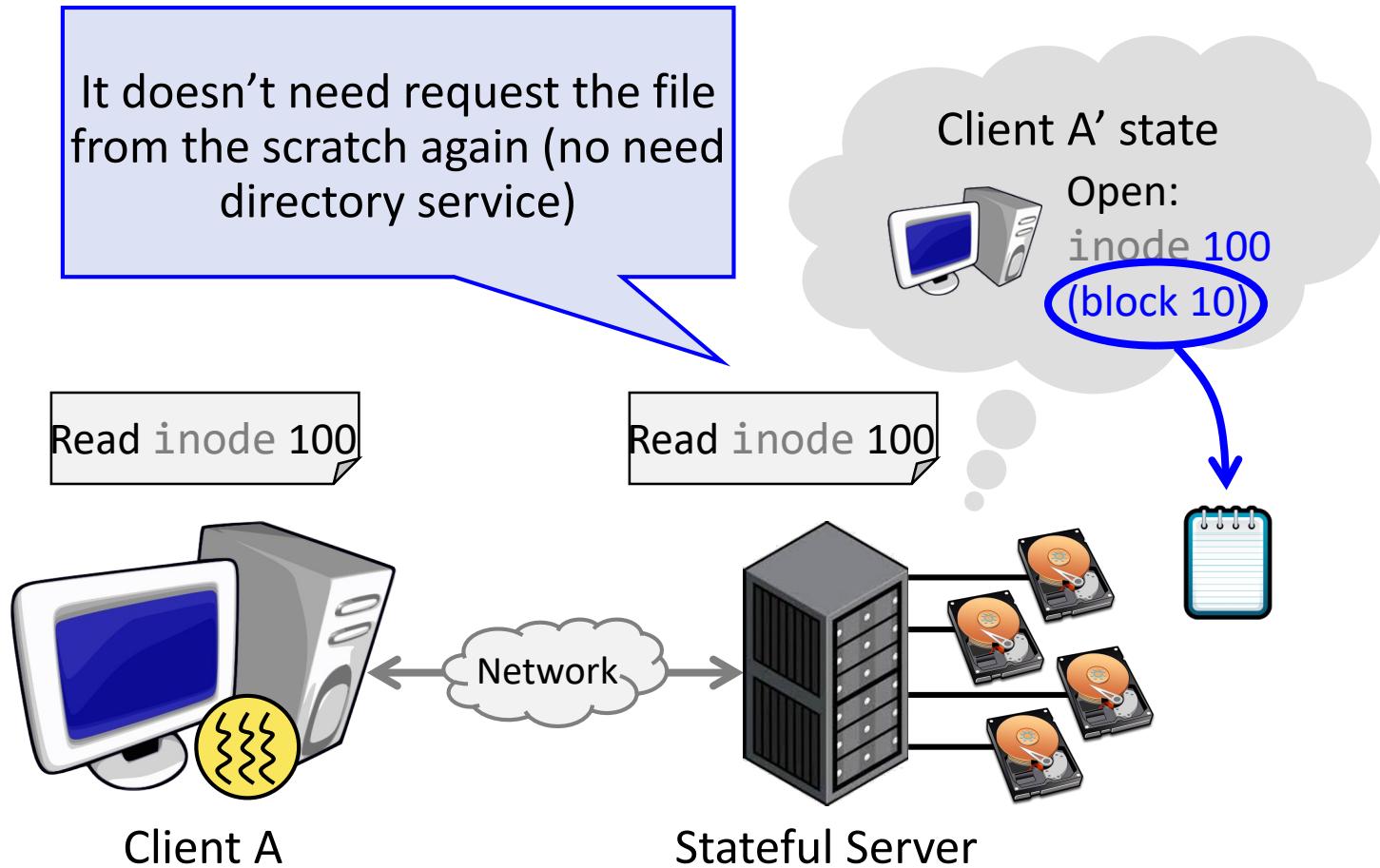
Network



Stateful Server

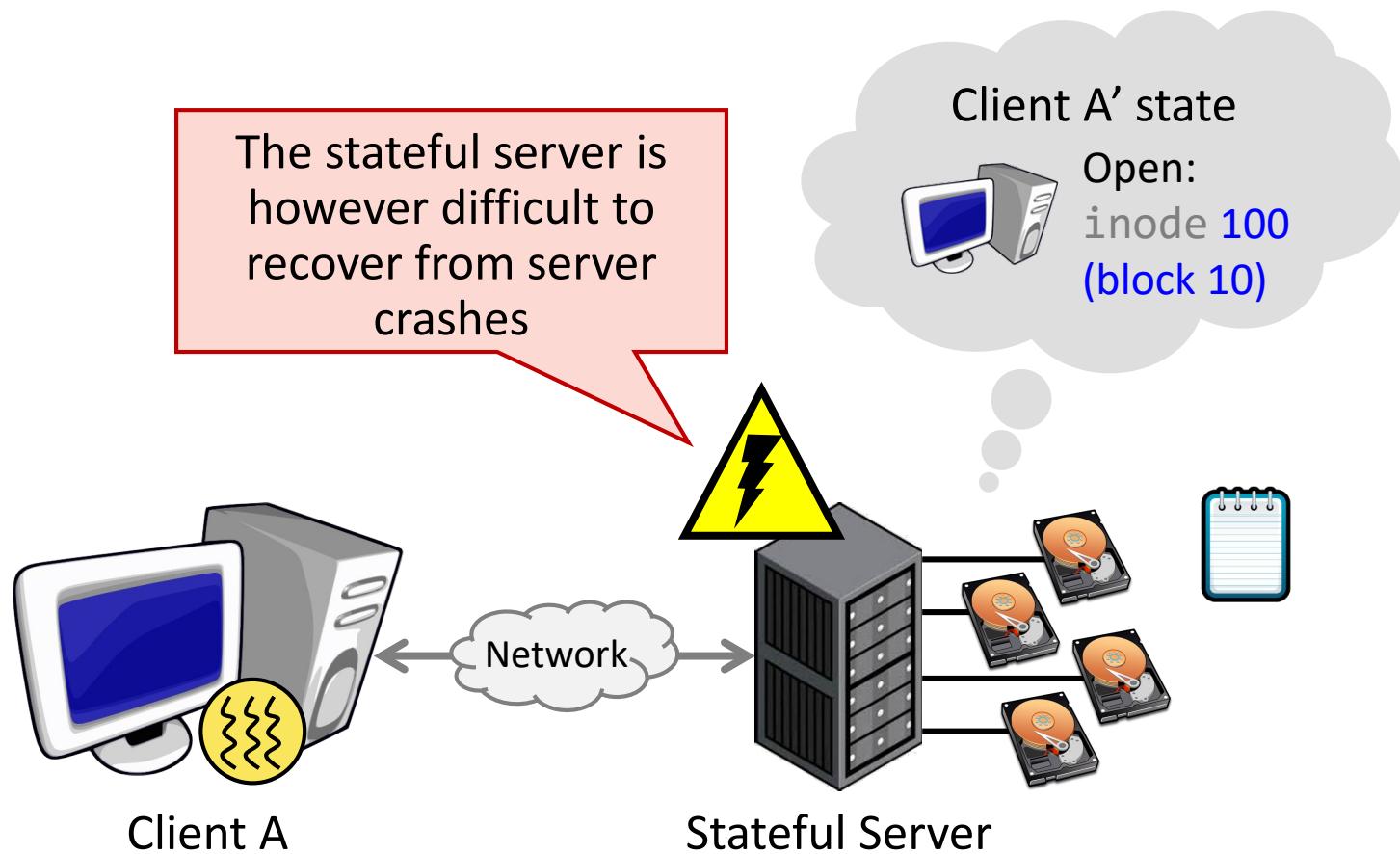
Client A' state
Open:
inode 100
(block 10)

Stateful Server in DFS



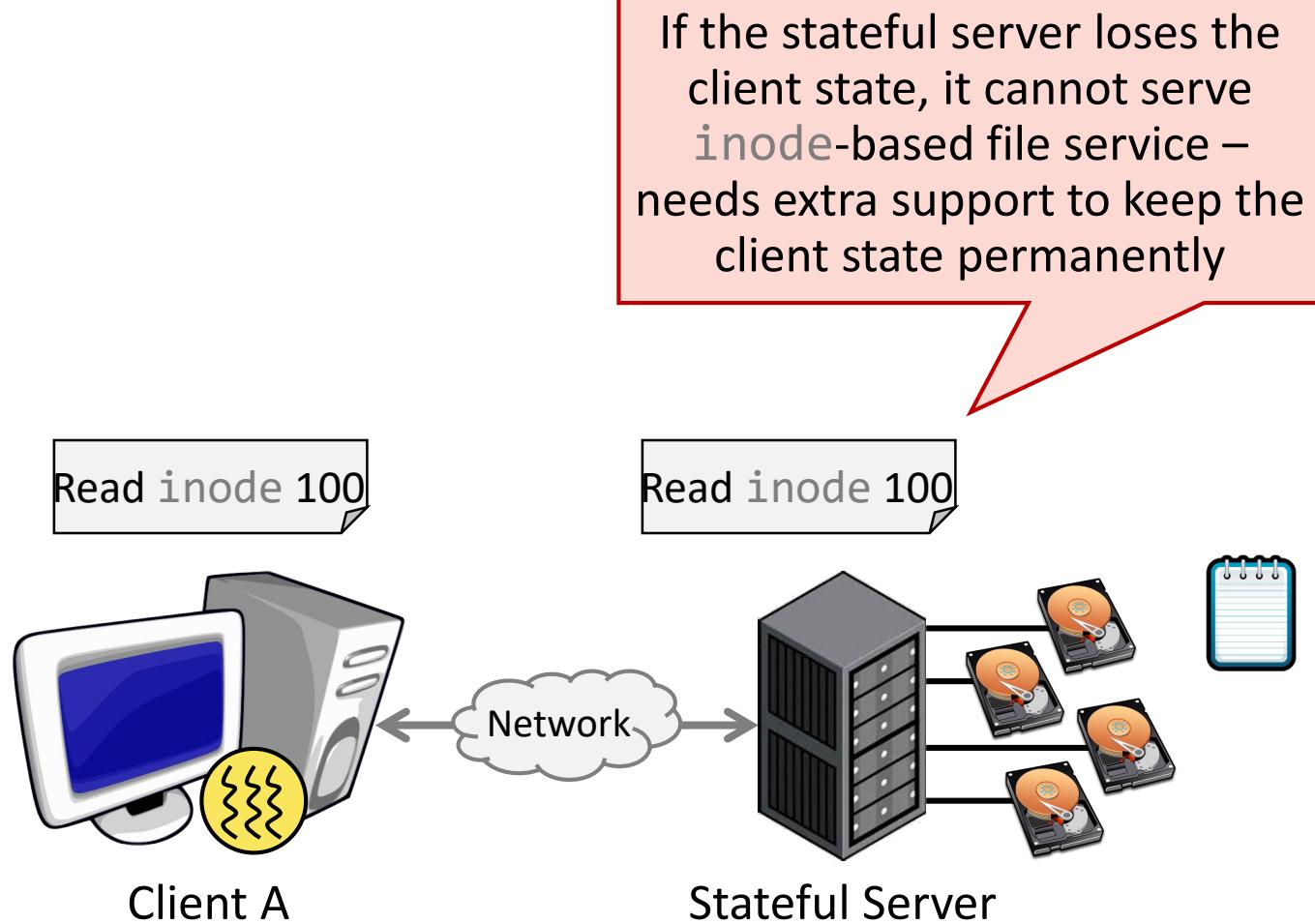


Stateful Server in DFS



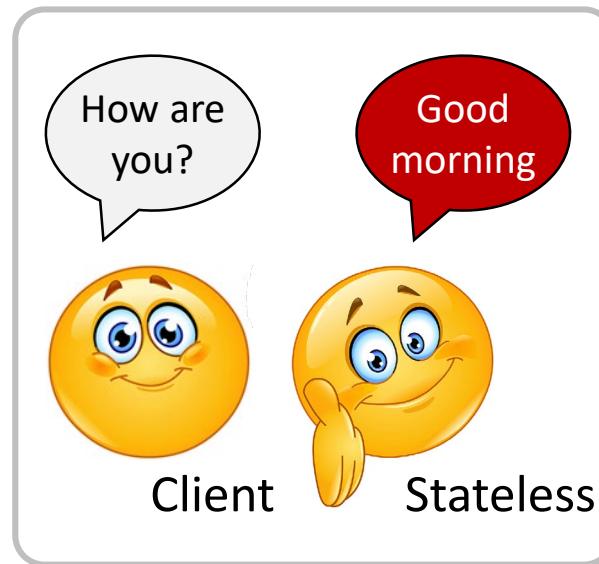
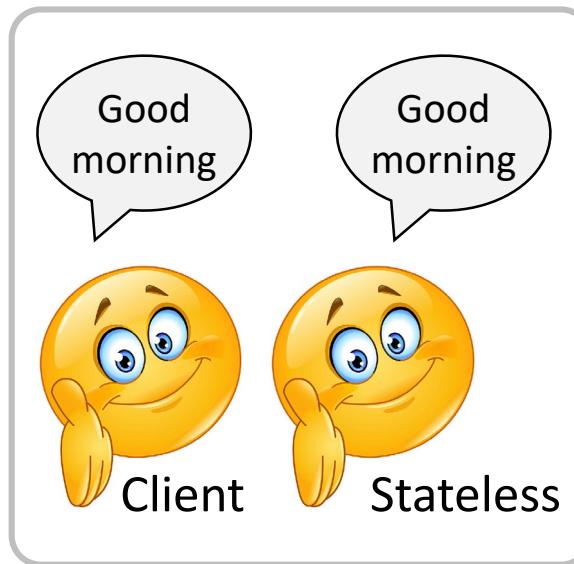


Stateful Server in DFS



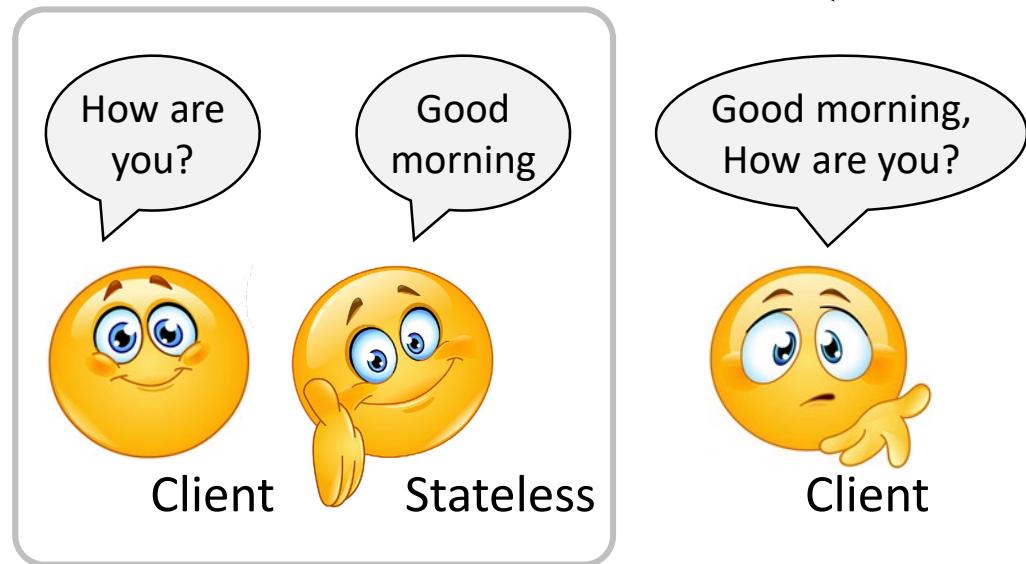
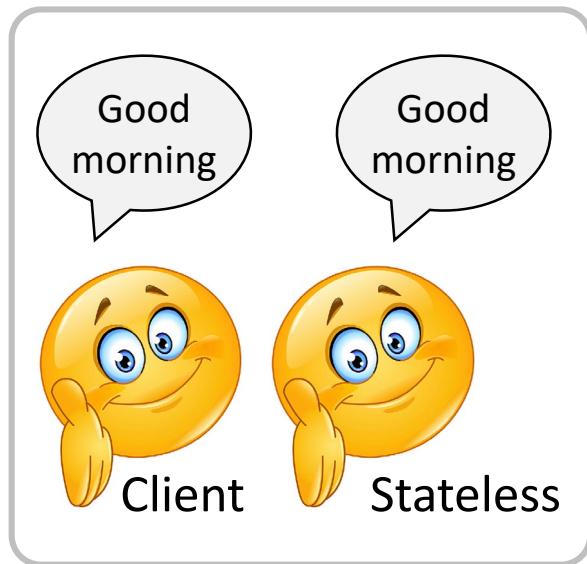
Option #2: Stateless Servers

Unlike the stateful server, as the stateless server does not keep the client's state. It just repeatedly says "Good morning" (initialized)



Option #2: Stateless Servers

Thus, the client should send a **complete request** (containing all information everytime)

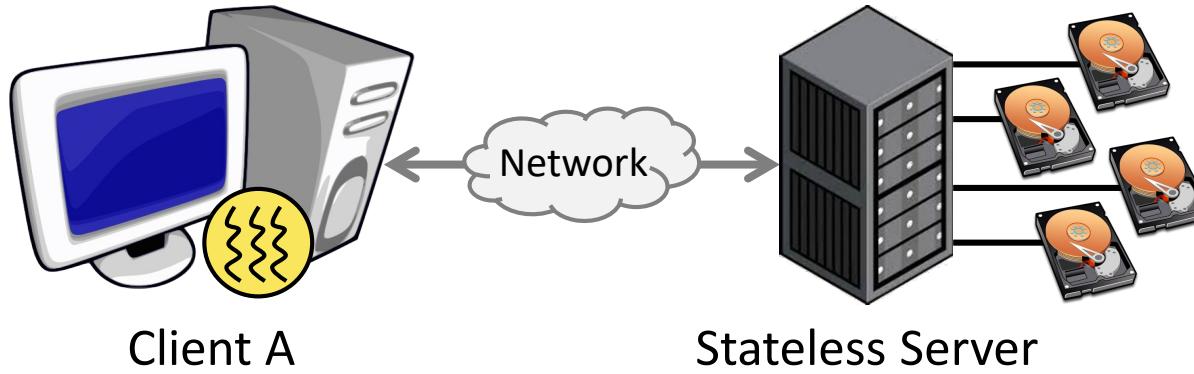




Stateless Server in DFS

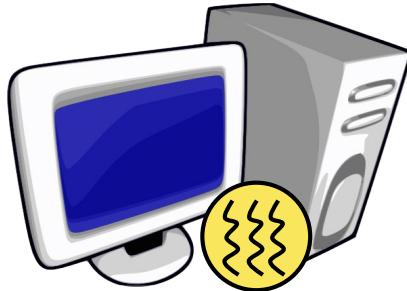
As no information kept by the stateless sever, the client should send a request containing **complete file names**

Open & Write /a/b/c/data.txt



Stateless Server in DFS

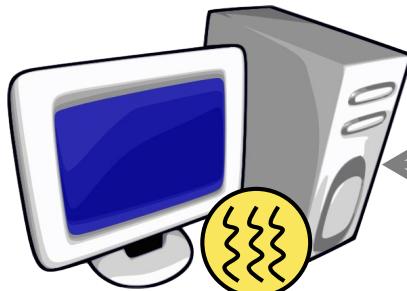
Open & Write /a/b/c/data.txt



Client B

As there is nothing to record on the server-side, there is **no limit** on the number of clients

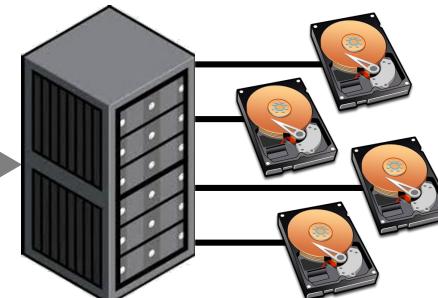
Open & Write /a/b/c/data.txt



Client A

Open & Write /a/b/c/data.txt

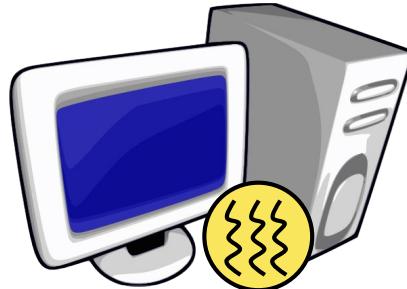
Open & Write /a/b/c/data.txt



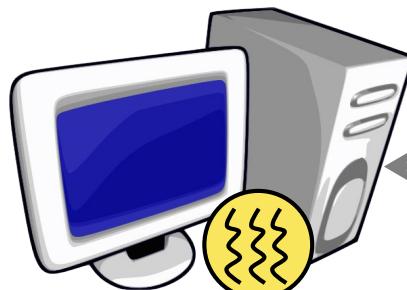
Stateless Server



Stateless Server in DFS



Client B

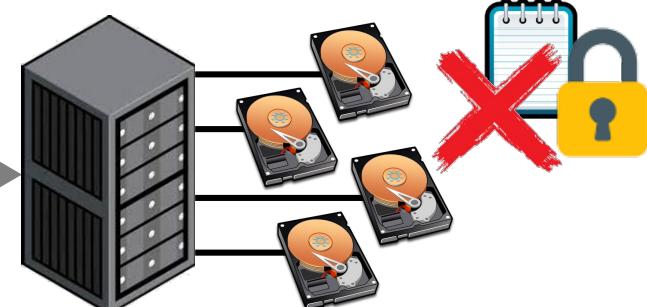


Client A

The stateless server has better tolerance of server crashes than a stateful server. However, **file locking is not supported in this design** – needs extra file locking server to manage the file sharing

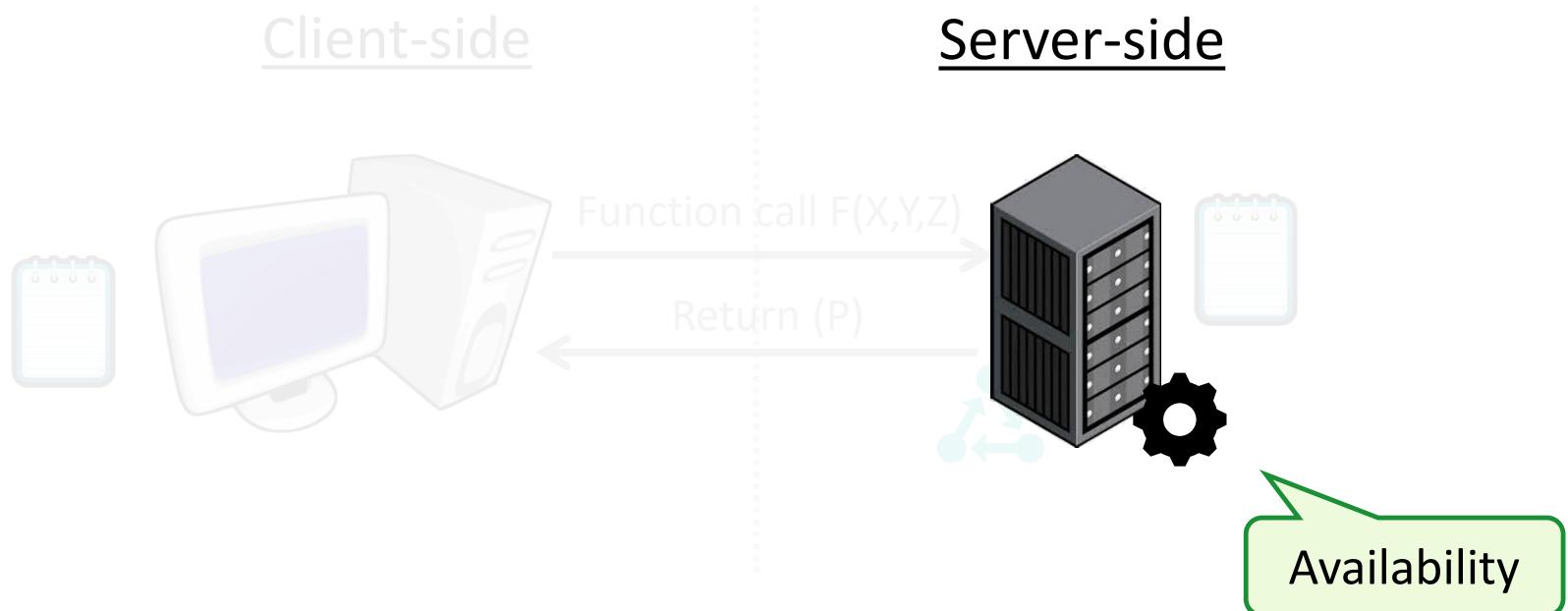
Open & Write /a/b/c/data.txt

Open & Write /a/b/c/data.txt

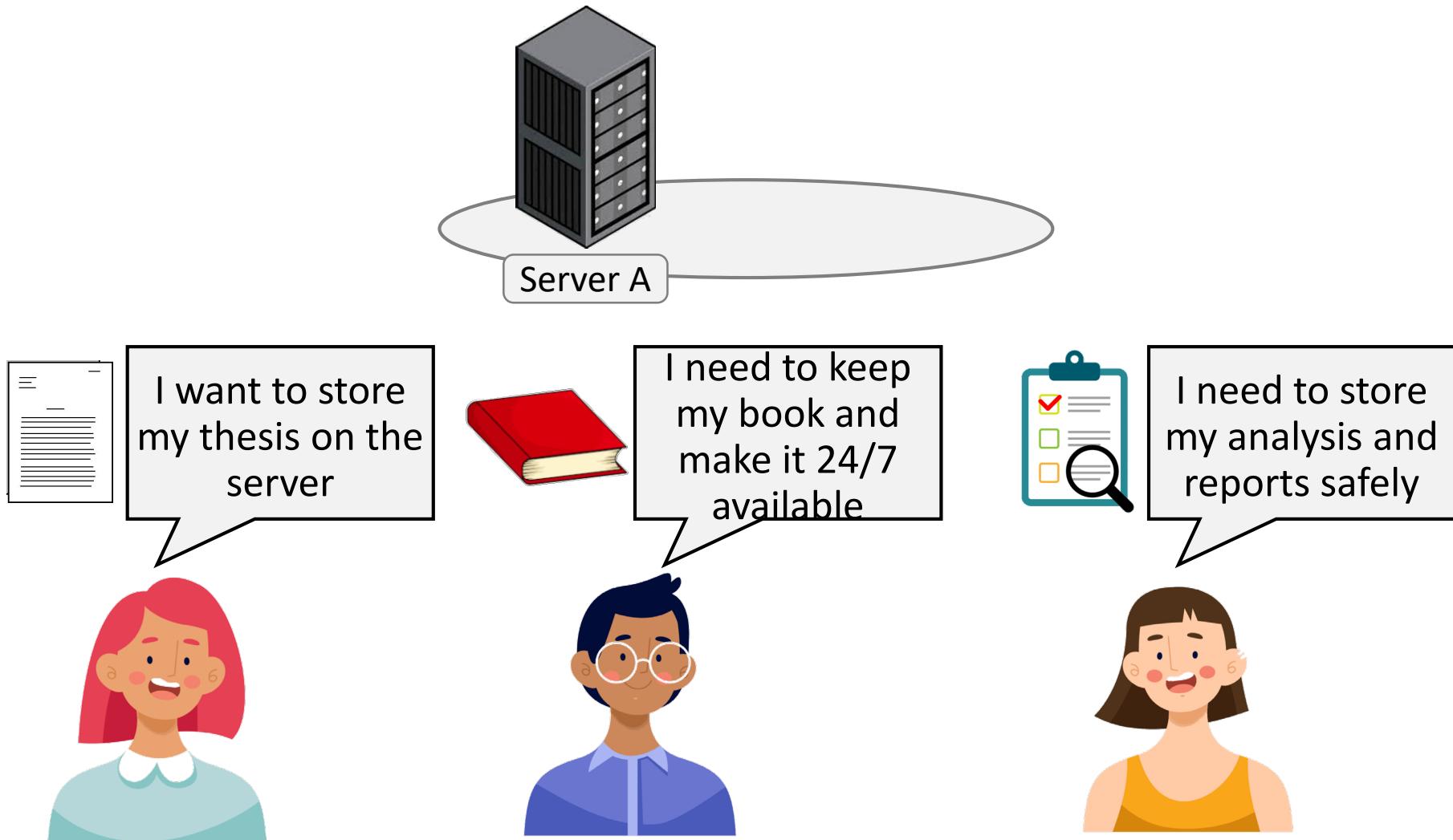


Stateless Server

Server-Side #3: Availability



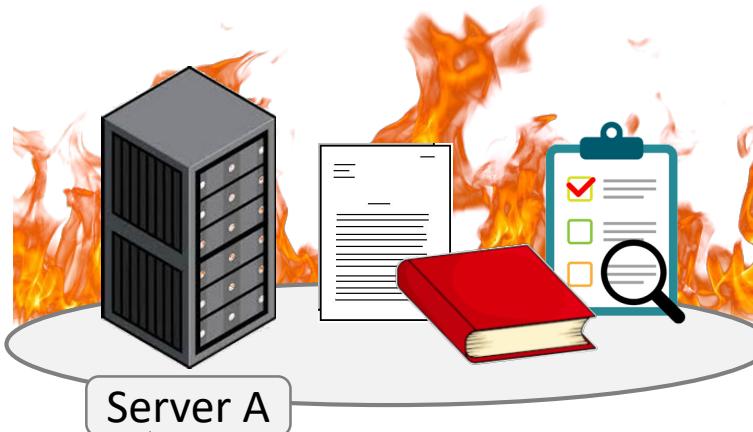
A Simple Unavailability Case for DFS



A Simple Unavailability Case for DFS

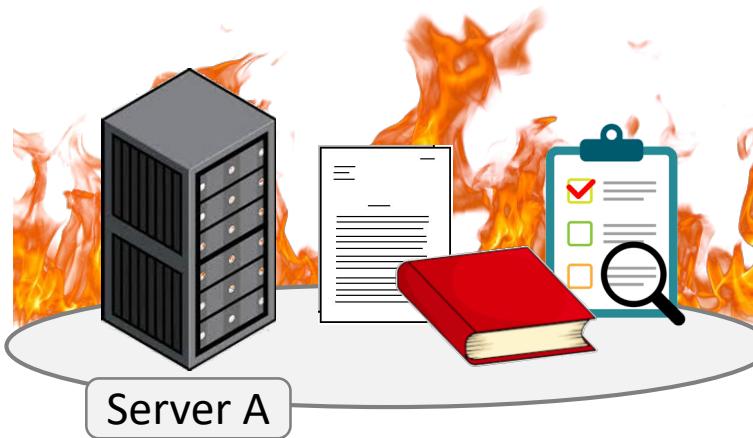


A Simple Unavailability Case for DFS



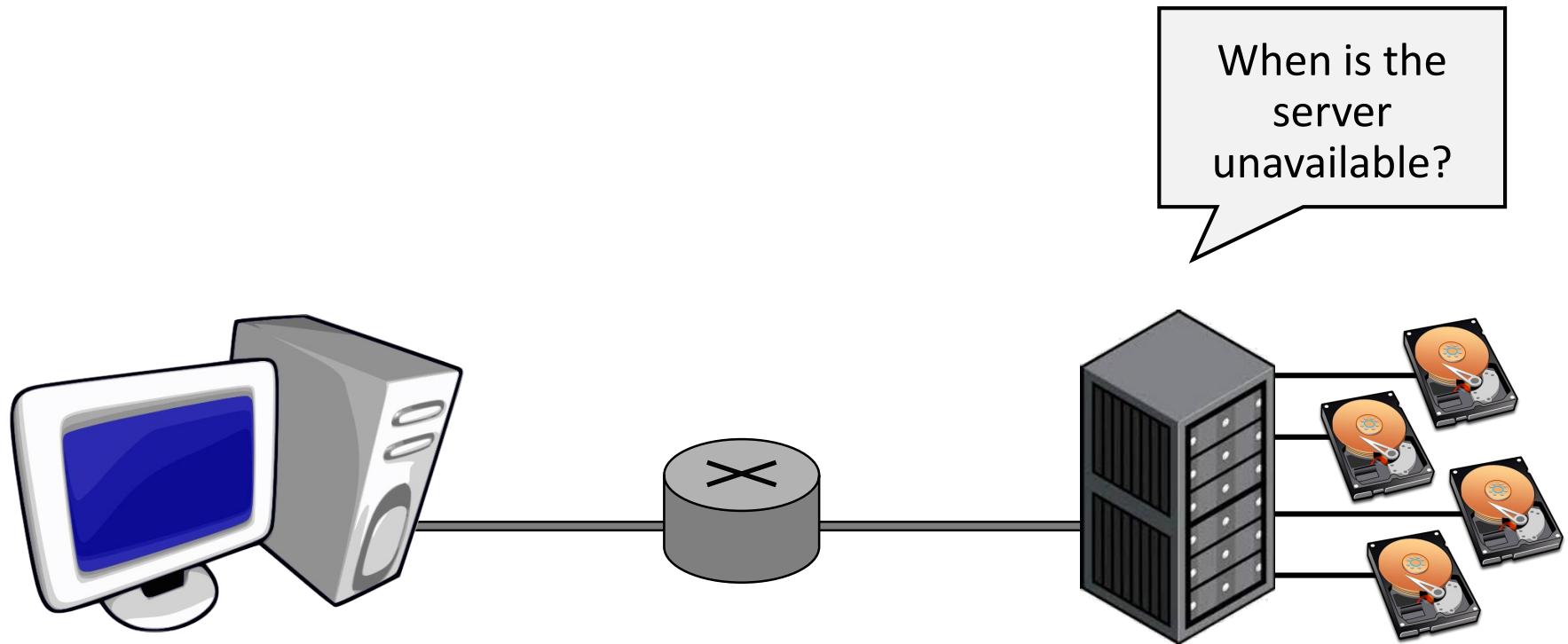
There is always possible failure,
which make the server A
unavailable for services and/or it
permanently lose clients data

A Simple Unavailability Case for DFS



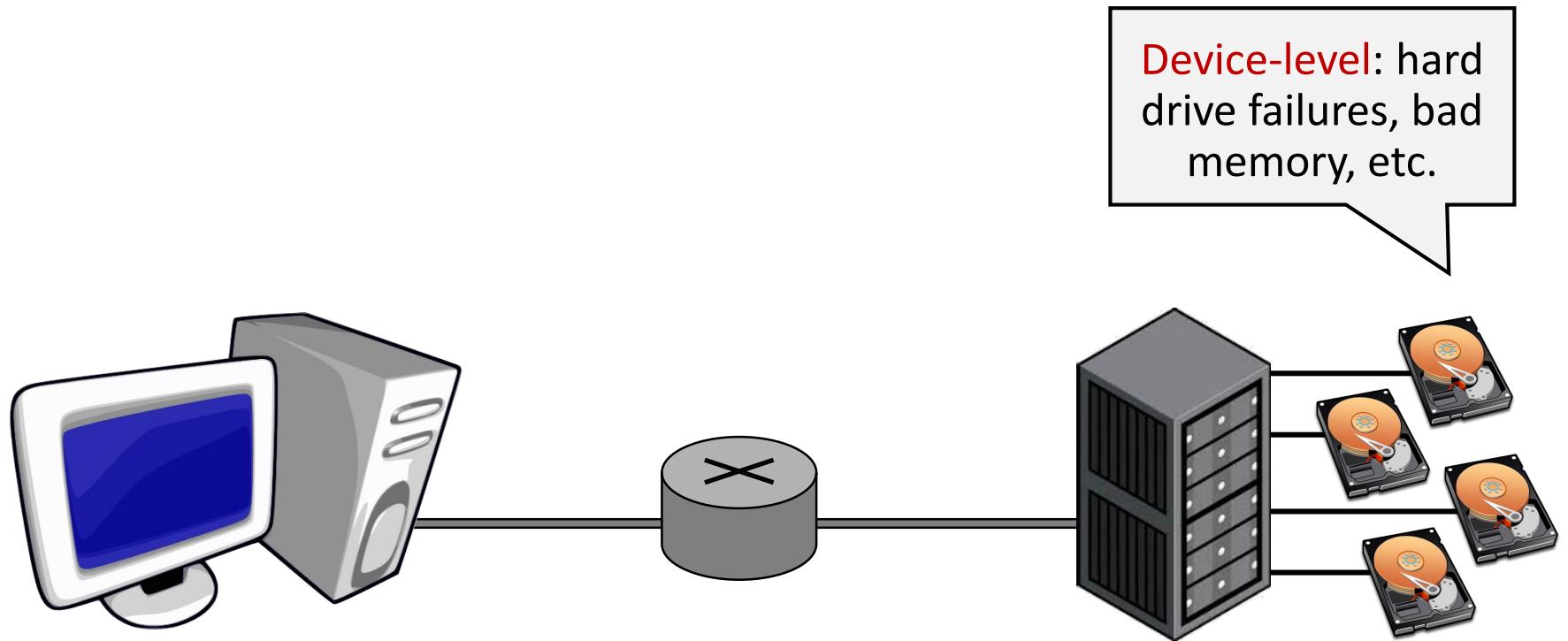
Why then we don't face
this situation (e.g., when
you use Cloud)?

Unavailability of Real Hardware



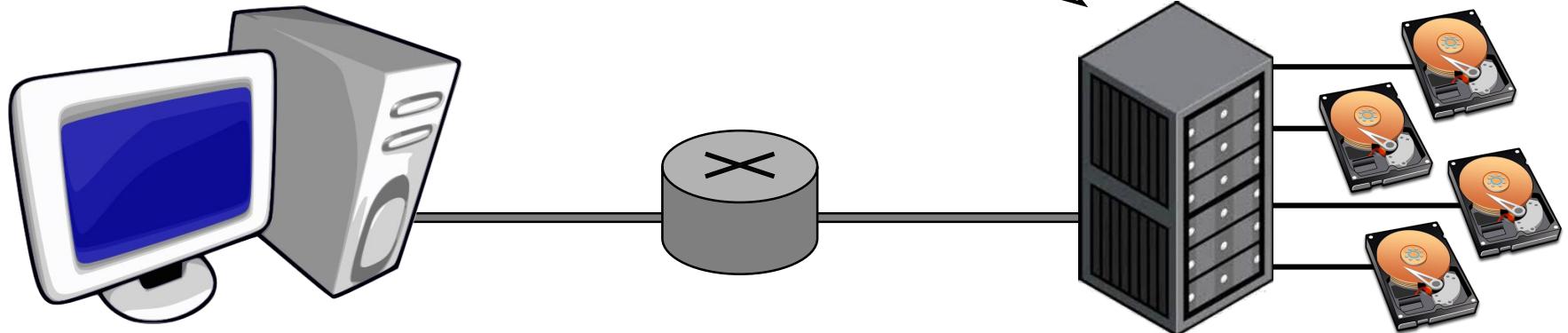


Device-Level



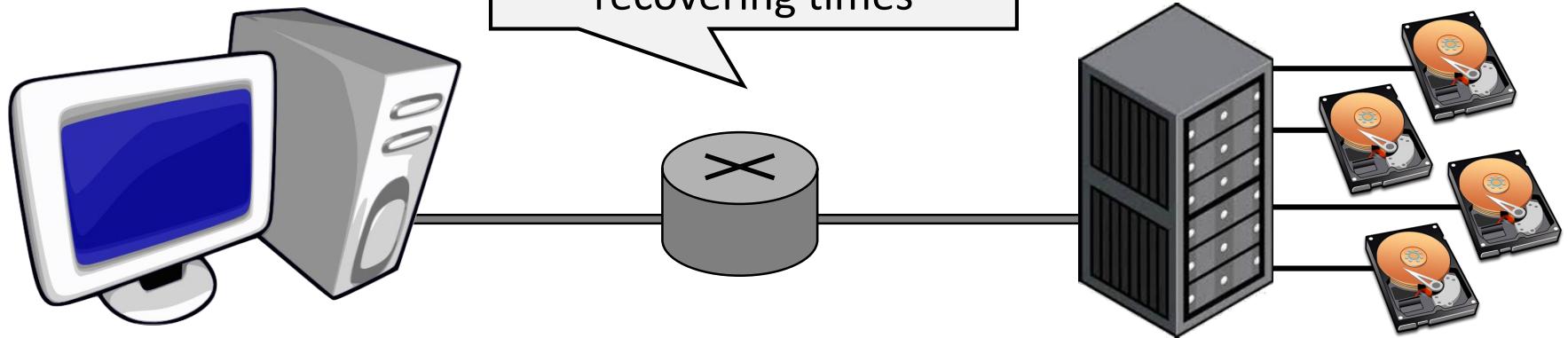
Machine-Level

Machine-level: Thermal issues (e.g., overheating) make the server broken.
Even with a short power failure (<5 mins), it takes 1-2days to recover

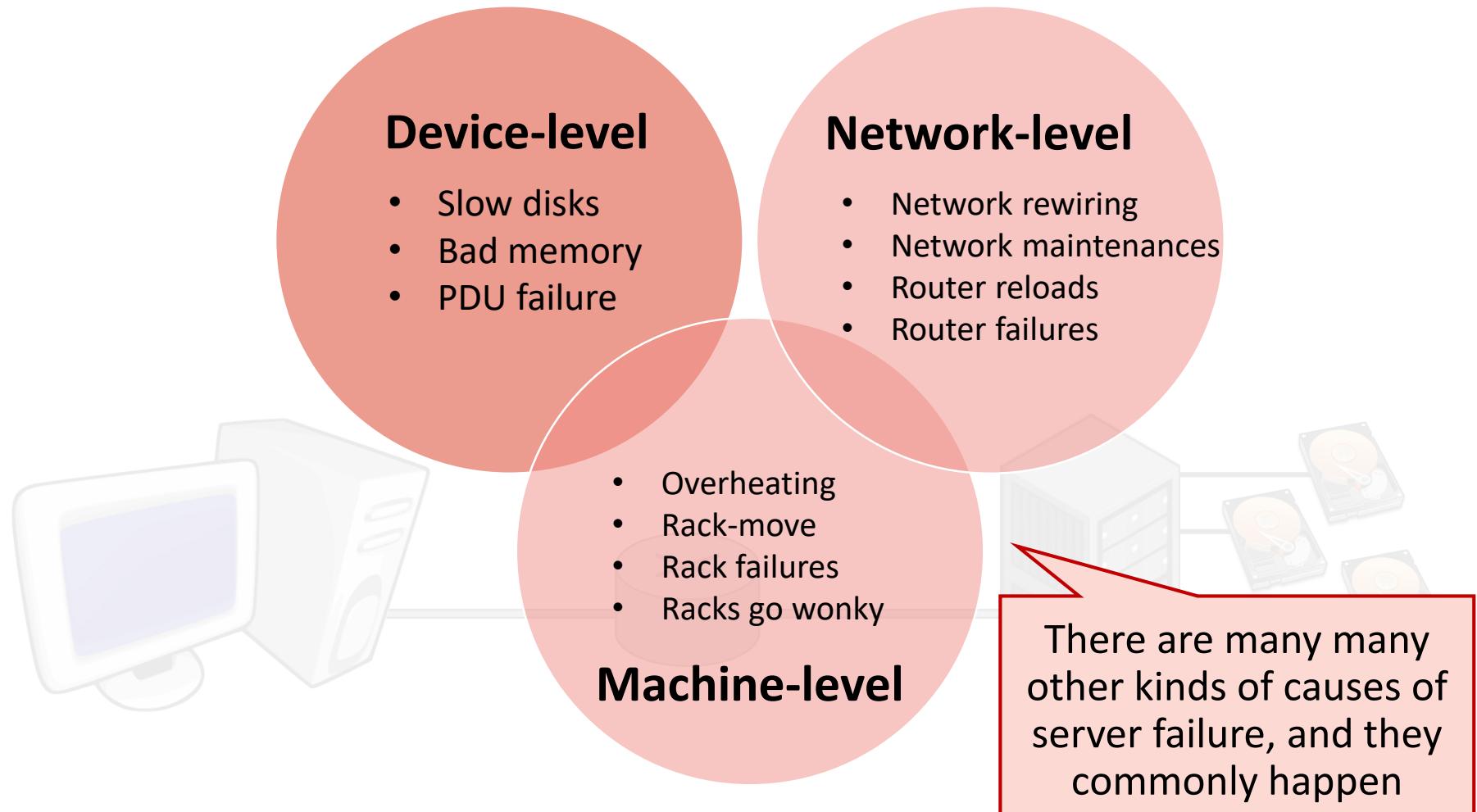


Network-Level

Network-level: 5% of network often be rewired, which takes 2-day recovering times

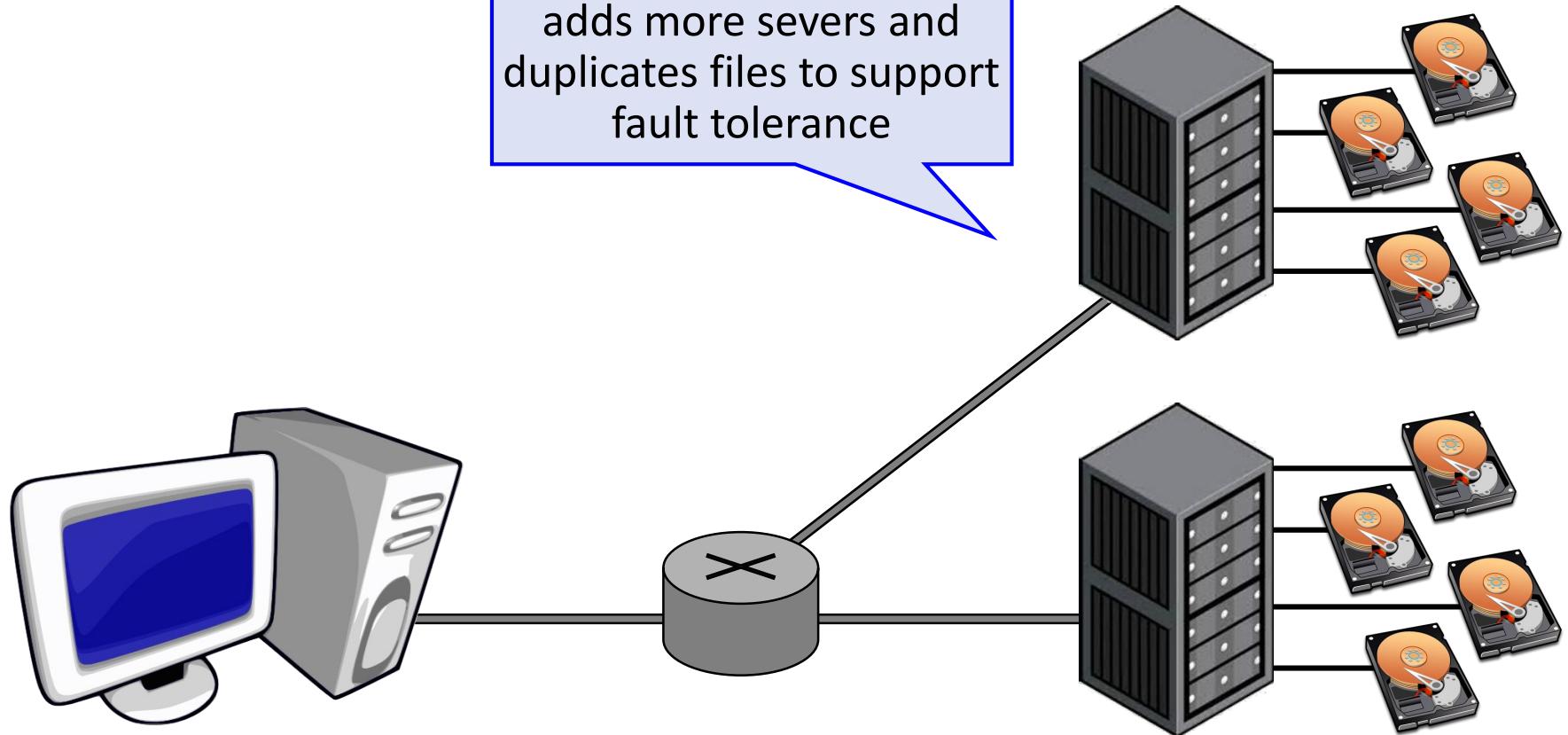


Network-Level

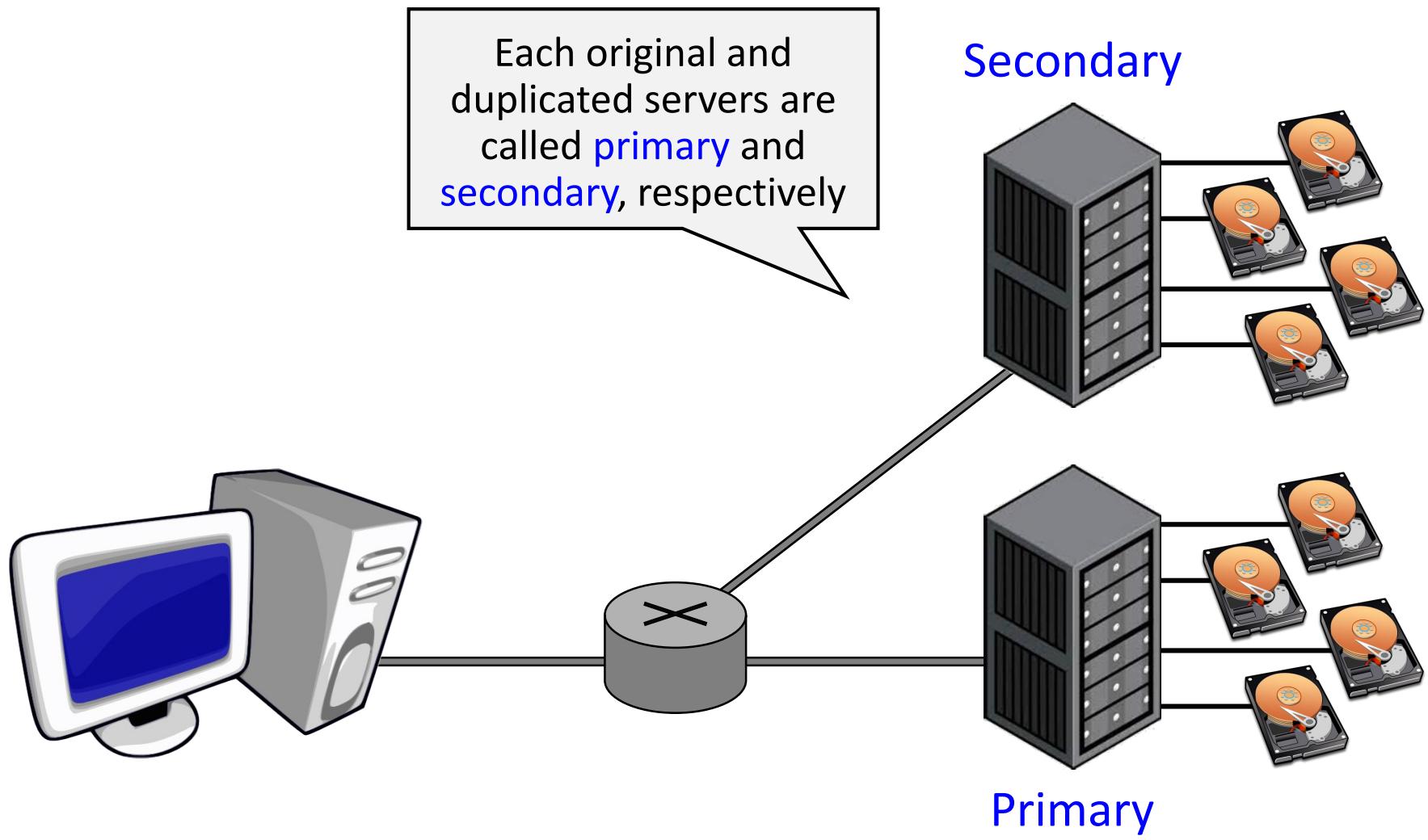


Implementation: Duplication

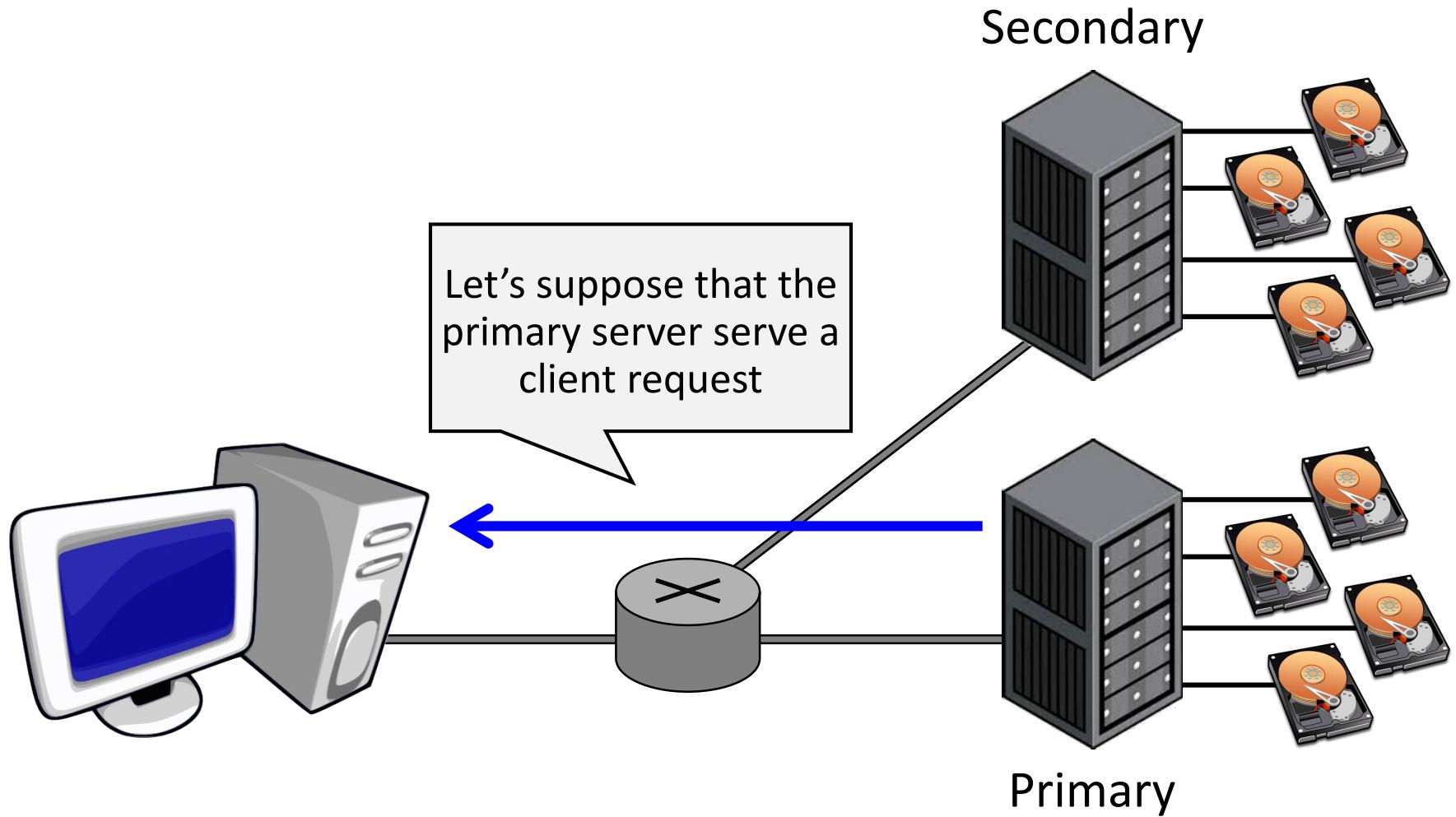
Like RAID-0 (mirroring),
adds more servers and
duplicates files to support
fault tolerance



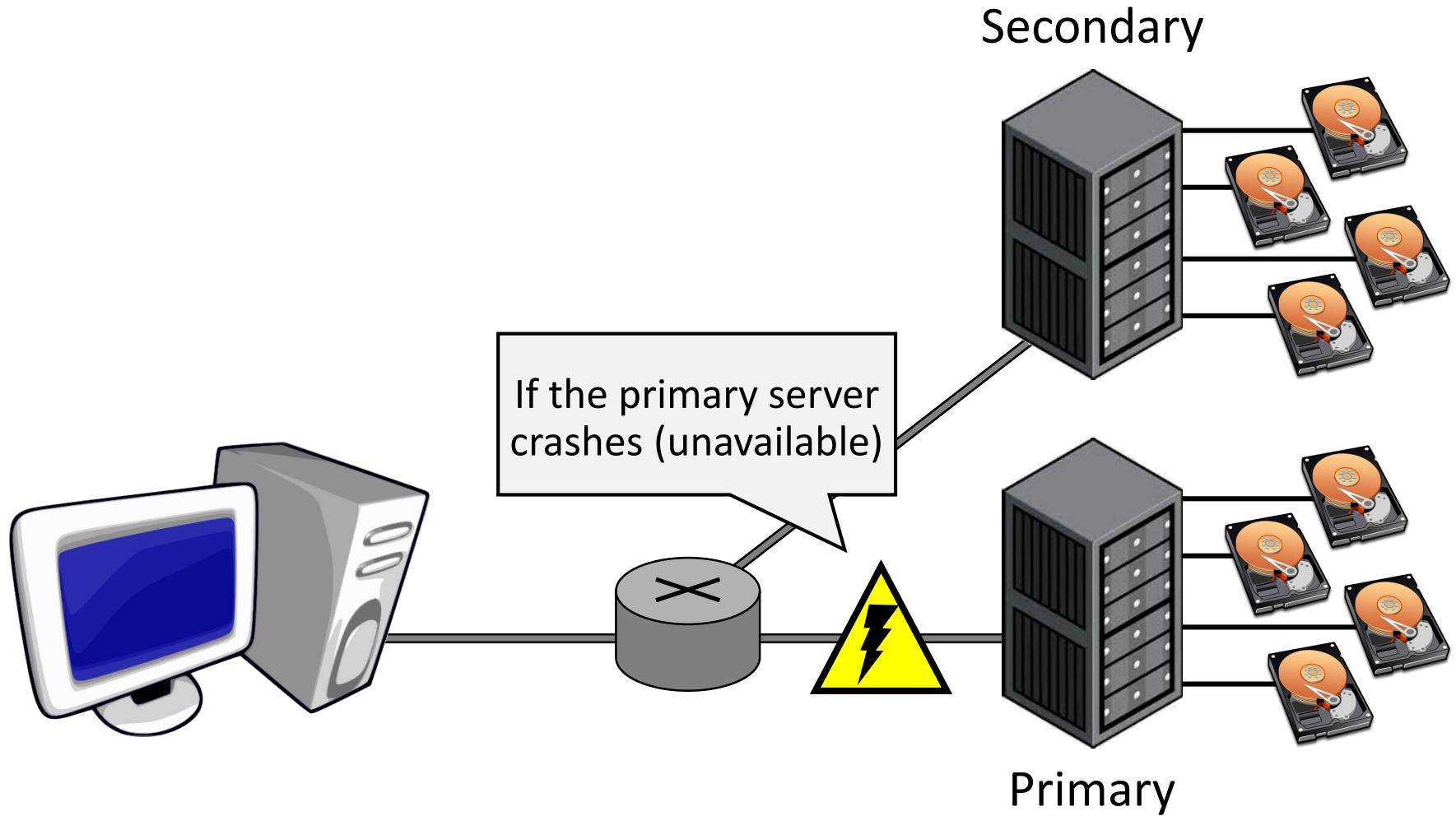
Implementation: Duplication



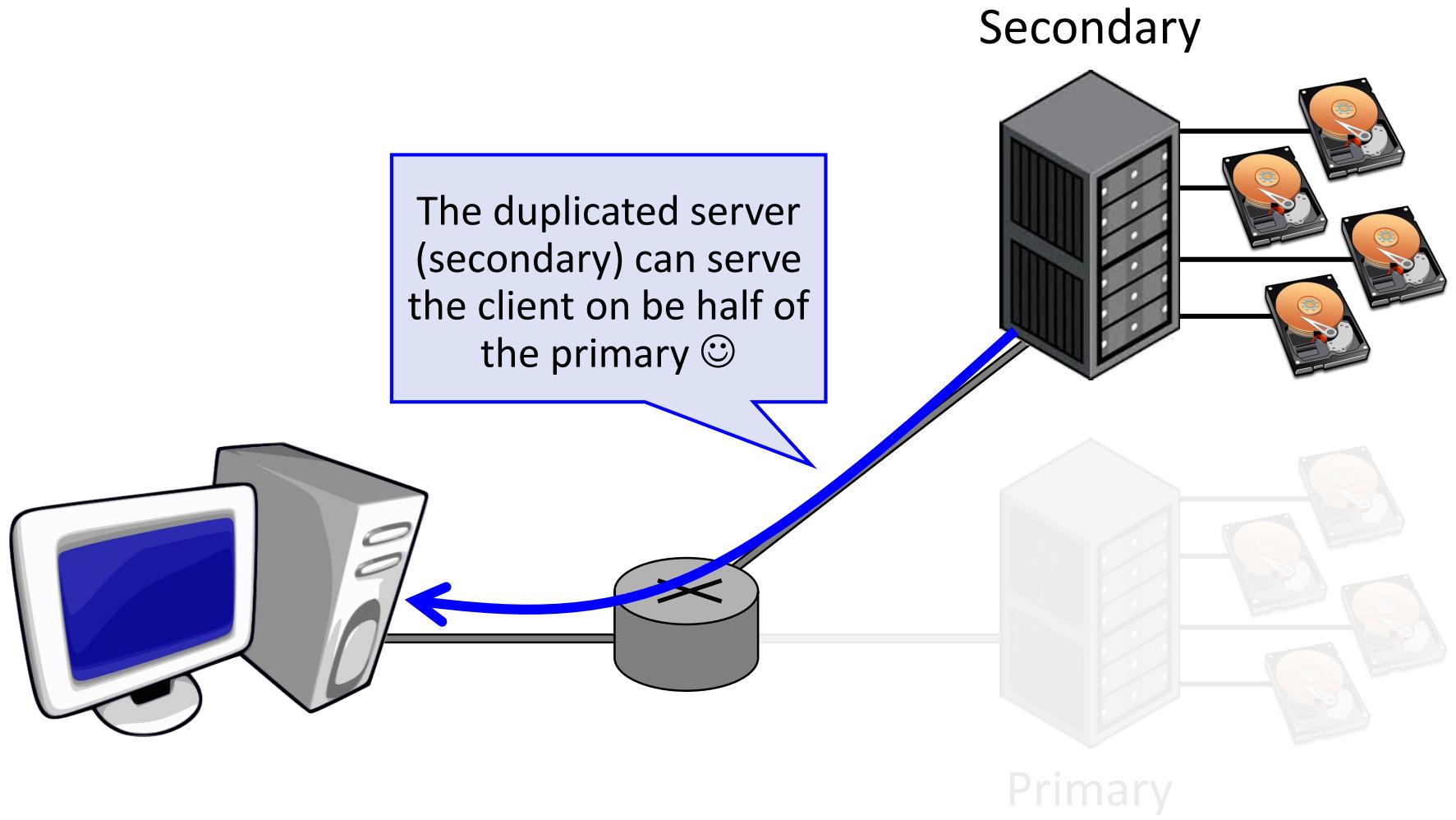
Implementation: Duplication



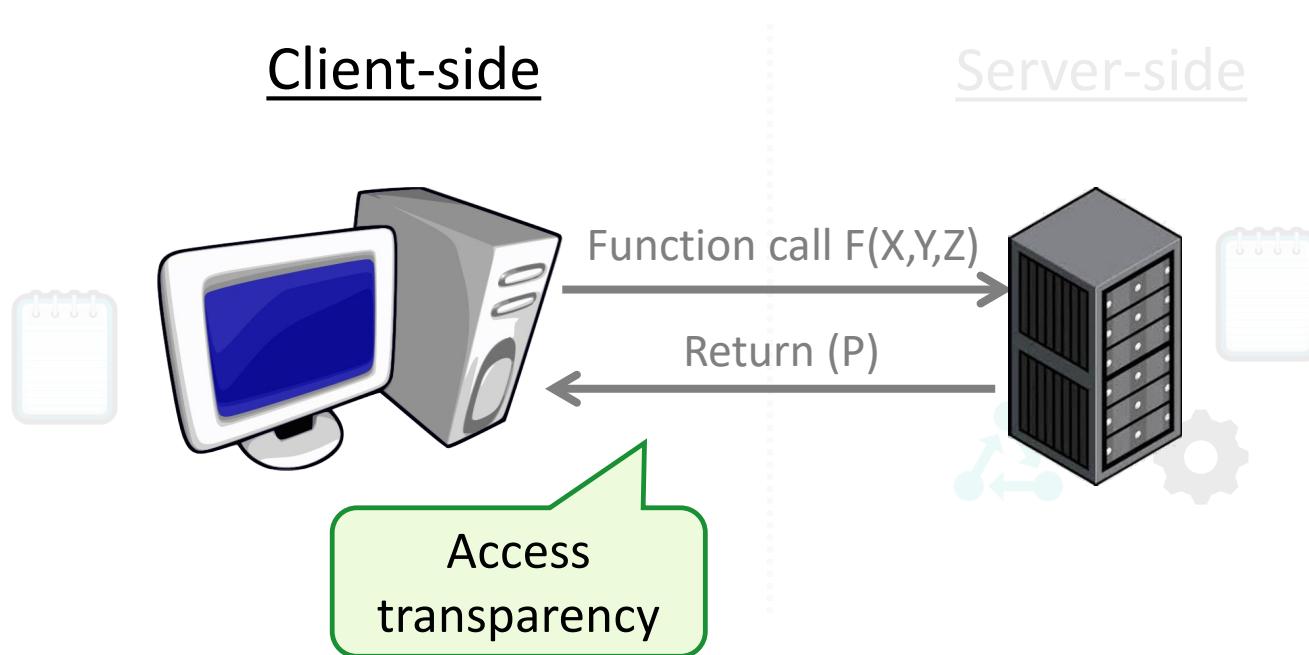
Implementation: Duplication



Implementation: Duplication

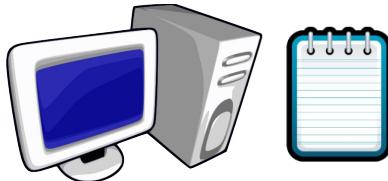


Client-Side #1: Access Transparency



Local Access vs. Remote Access

Local file access



```
FILE * f = fopen("foo.txt", "w");
fwrite(&x, sizeof(uint32_t), 1, f);
```

A local file can be accessed over the POSIX file APIs (e.g., `fopen`, `fwrite`)

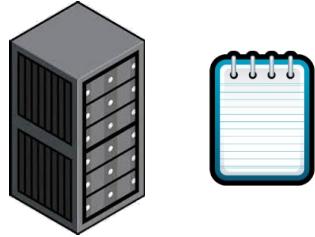
Local Access vs. Remote Access



Local file access

```
FILE * f = fopen  
fwrite(&x, size,
```

But, a remote file is not



Remote file access

```
serverIP, remoteDir  
= remotefopen("foo.txt", "w");  
message = strncat(data, remoteDir, "w");  
send(serverIP, message);
```

Local Access vs. Remote Access



Local file access

```
FILE * f = fo  
fwrite(&x, si
```

As the client and the server are connected via a network, they send/receive a message by giving the target IP address

Remote file access

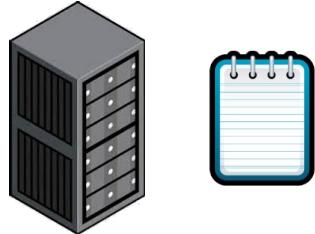
```
serverIP, remoteDir  
= remotefopen("foo.txt", "w");  
message = strncat(data, remoteDir, "w");  
send(serverIP, message);
```

Local Access vs. Remote Access



Local file access

```
FILE * f = fopen("foo.txt", "w");
fwrite(&x, sizeof(uint32_t), 1, f);
```



Remote file access

```
serverIP, remoteDir
= remotefopen("foo.txt", "w");
message = strncat(data, remoteDir, "w");
send(serverIP, message);
```

Thus, you first need to get the serverIP and remoteDir for the remote file access

Local Access vs. Remote Access

Should the client distinguish local and remote file access and implement them differently? – **not easy to use**



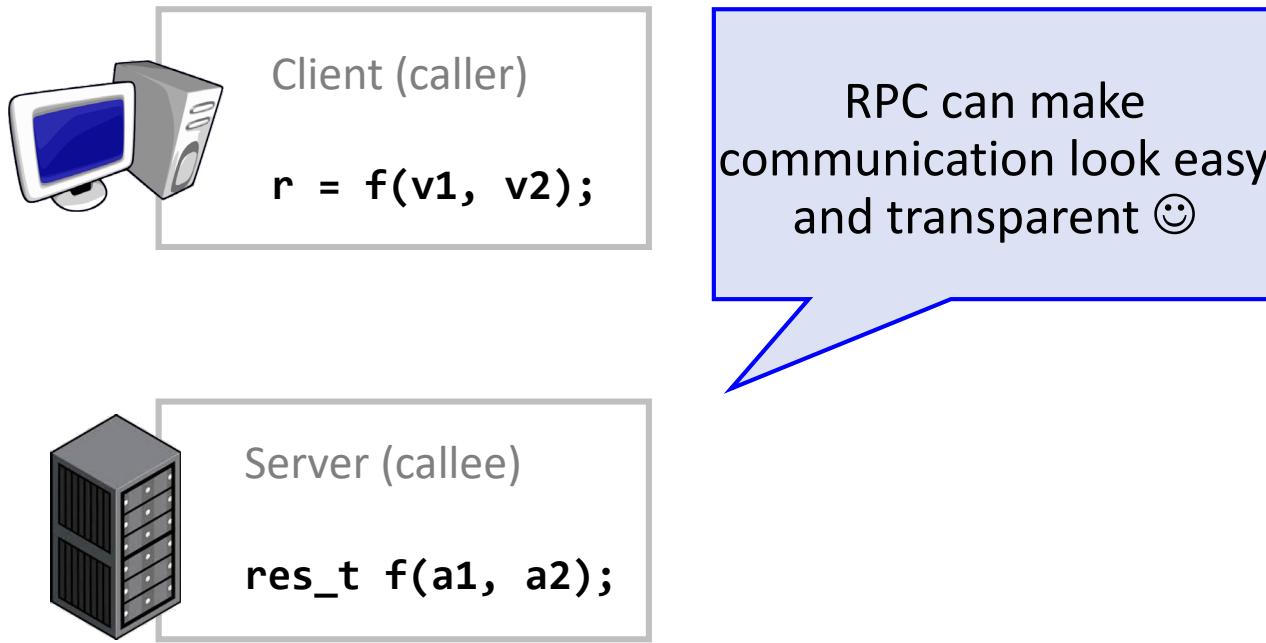
Local file access

```
FILE * f = fopen("foo.txt", "w");
fwrite(&x, sizeof(uint32_t), 1, f);
```

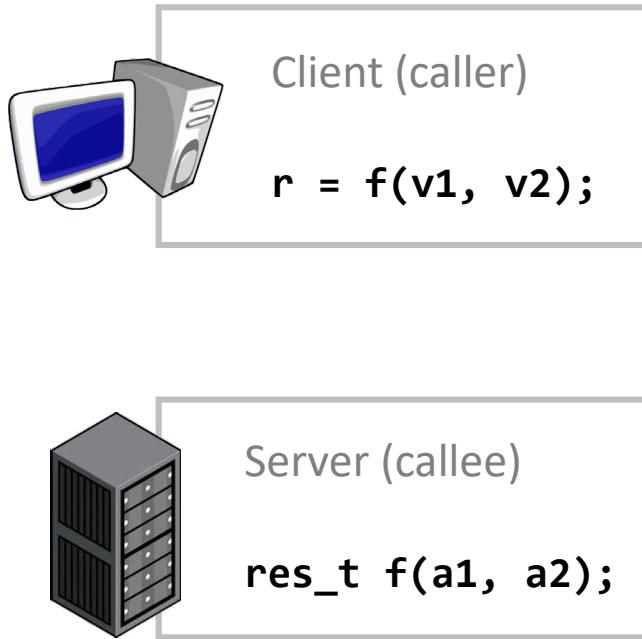
Remote file access

```
<serverIP, remoteDir>
    = remotefopen("foo.txt", "w");
message = strncat(data, remoteDir, "w");
send(serverIP, message);
```

Implementation: Remote Procedure Call



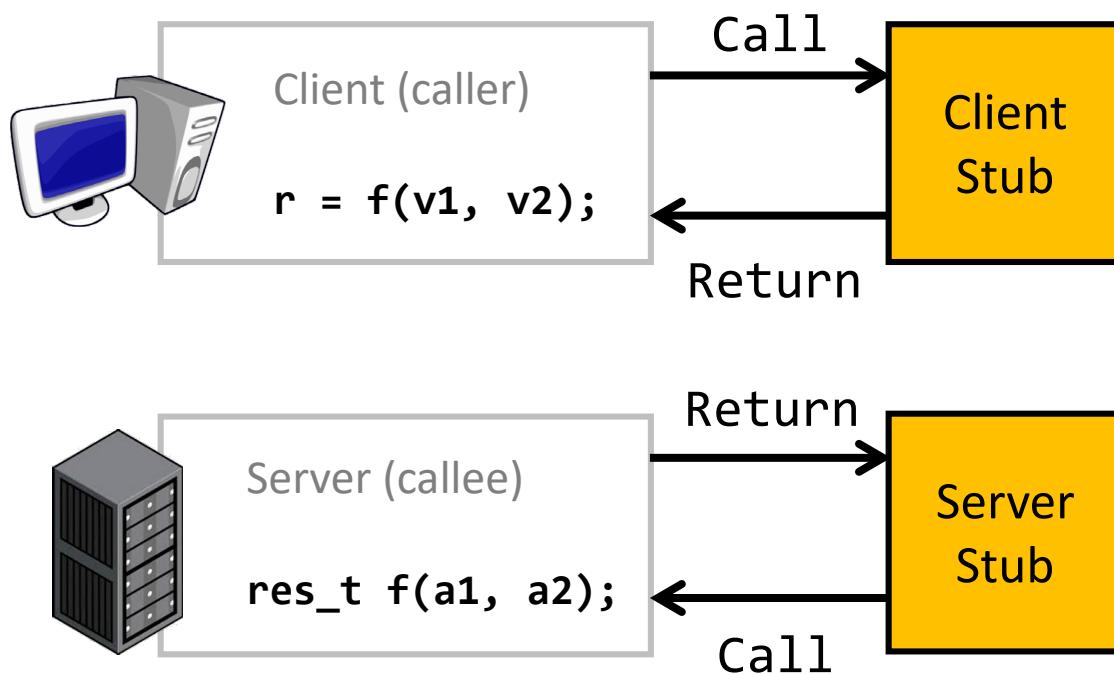
Implementation: Remote Procedure Call



1) When the client calls the function `f()`

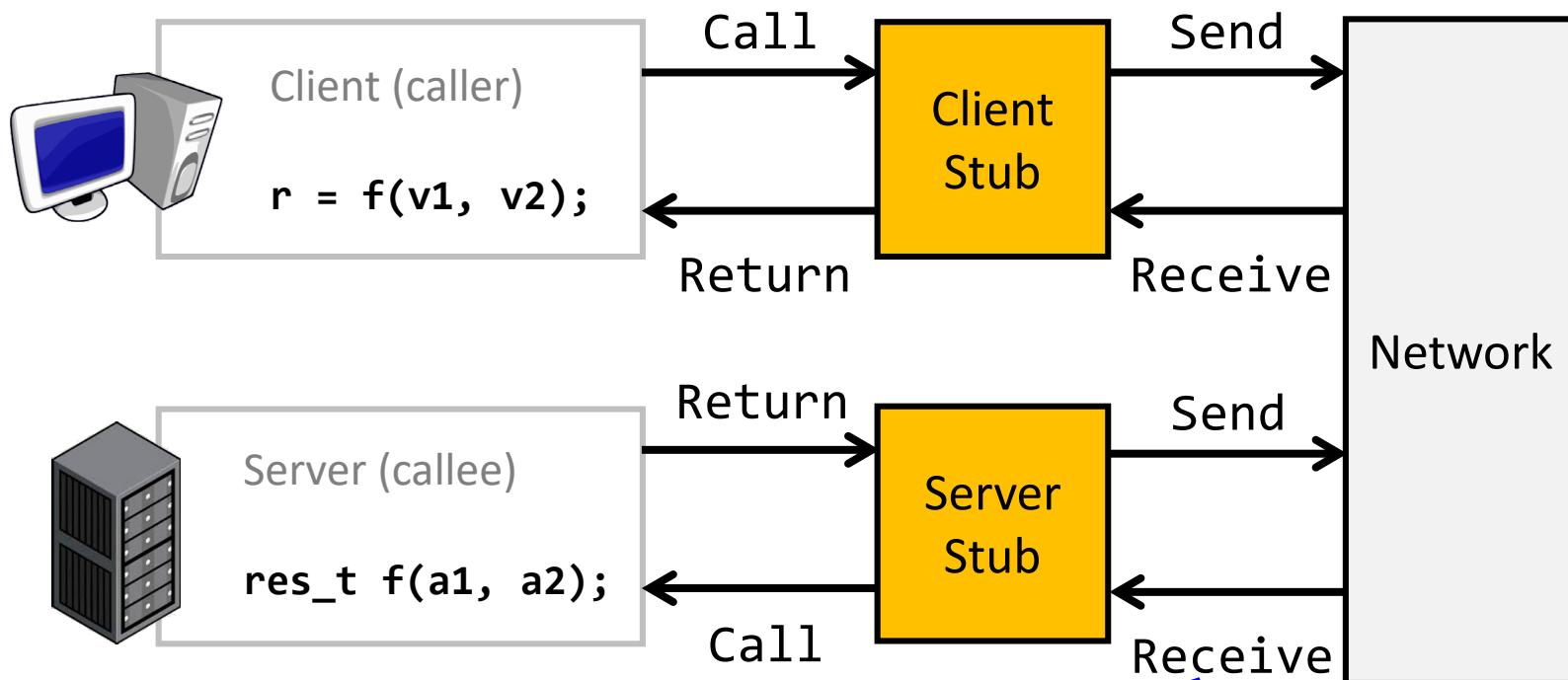
2) The server executes the function `f()` as well and returns the results to the client

Implementation: Remote Procedure Call



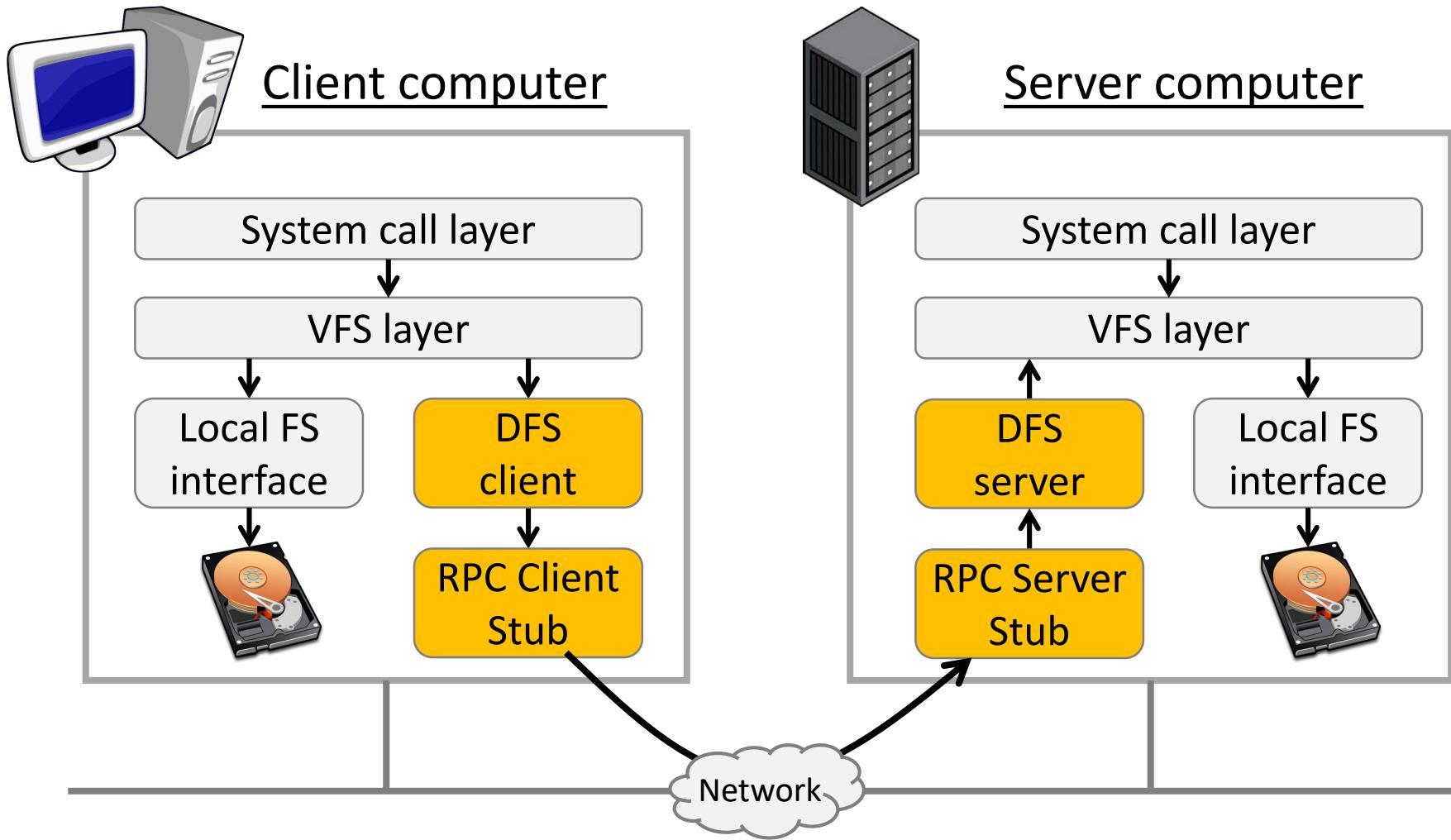
RPC provides “Stub” which can connect the client/server in a seamless manner

Implementation: Remote Procedure Call

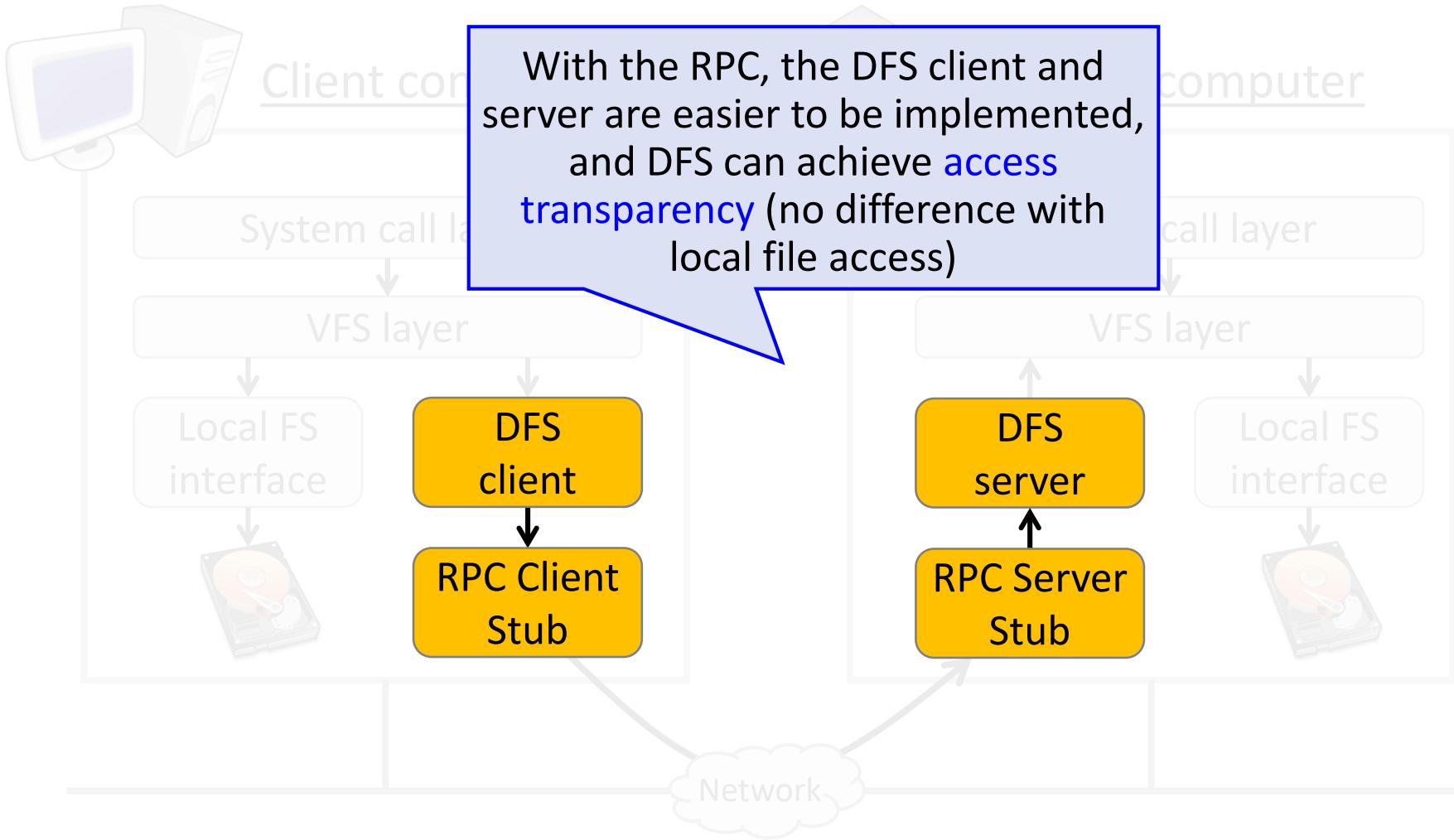


Stubs send/receive network messages (on behalf of DFS client/server processes). They convert this network semantic to the function call semantic

Modified Distributed File-System Stack



Modified Distributed File-System Stack



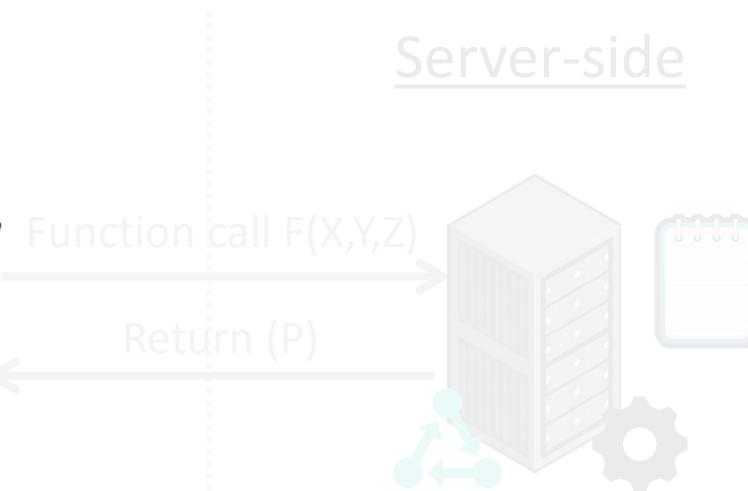
Client-Side #2: Perf. Transparency

Client-side



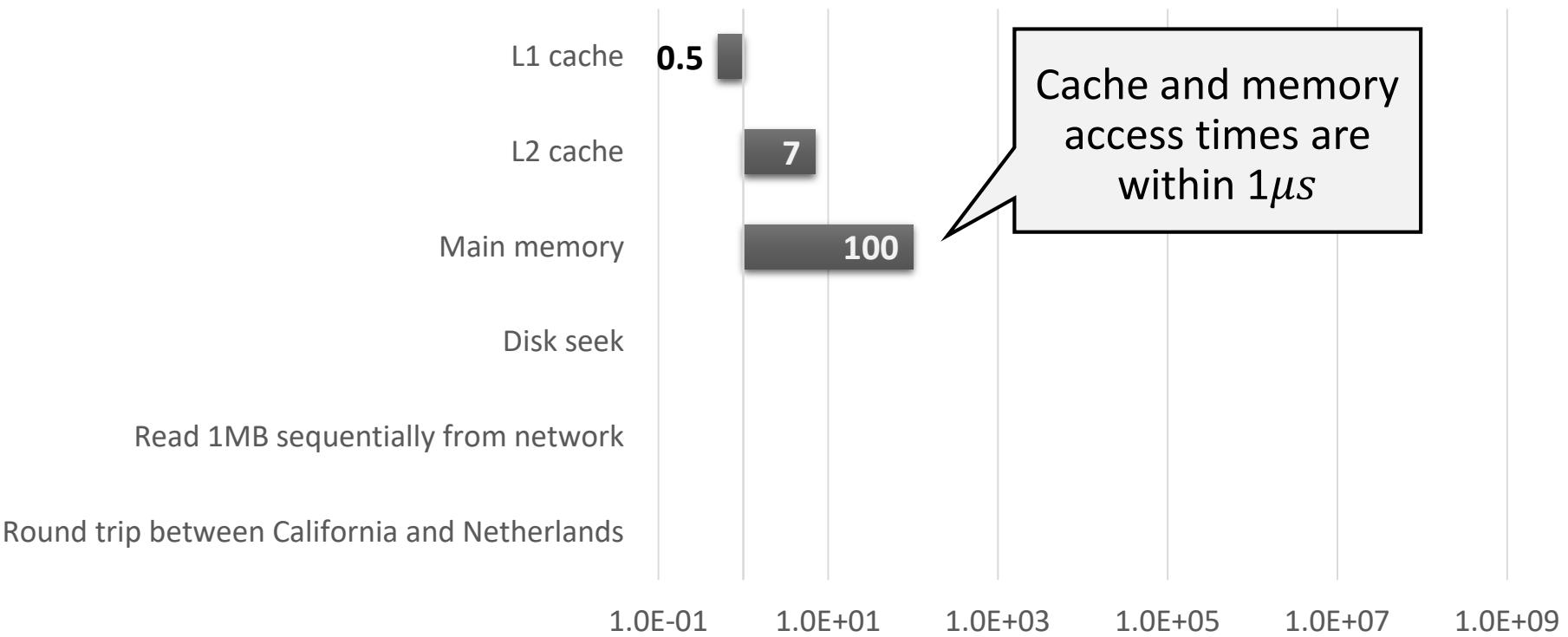
Performance transparency

Server-side



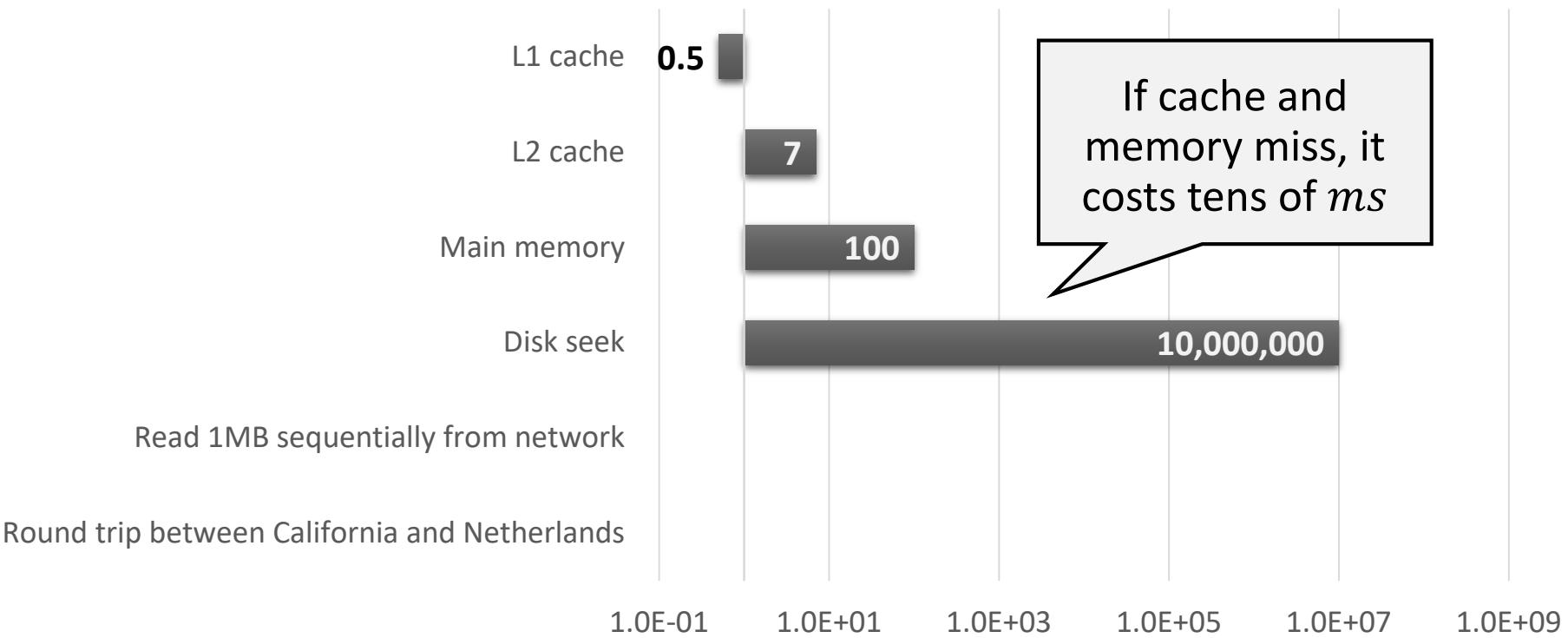
Network Is Slow

Numbers Real Word Engineers **SHOULD** Know (ns)



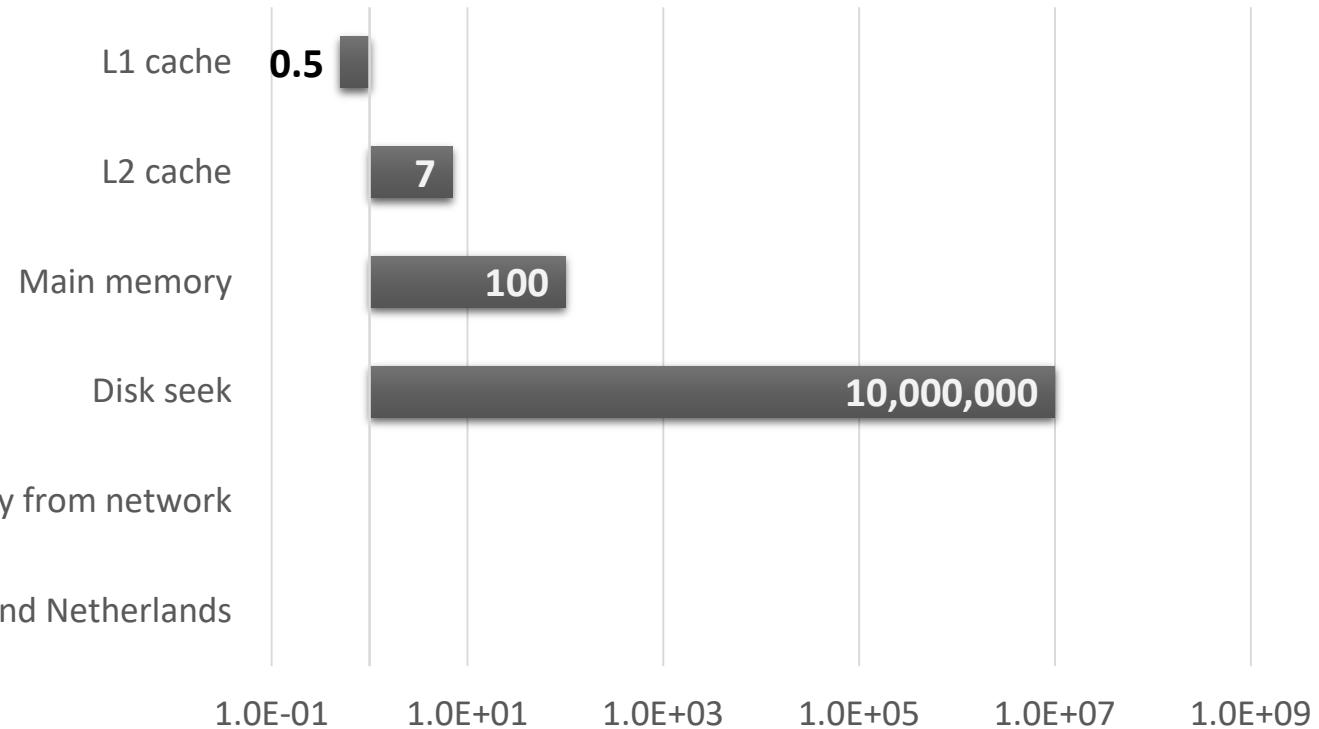
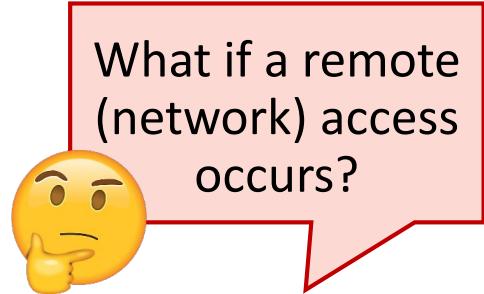
Network Is Slow

Numbers Real Word Engineers **SHOULD** Know (ns)



Network Is Slow

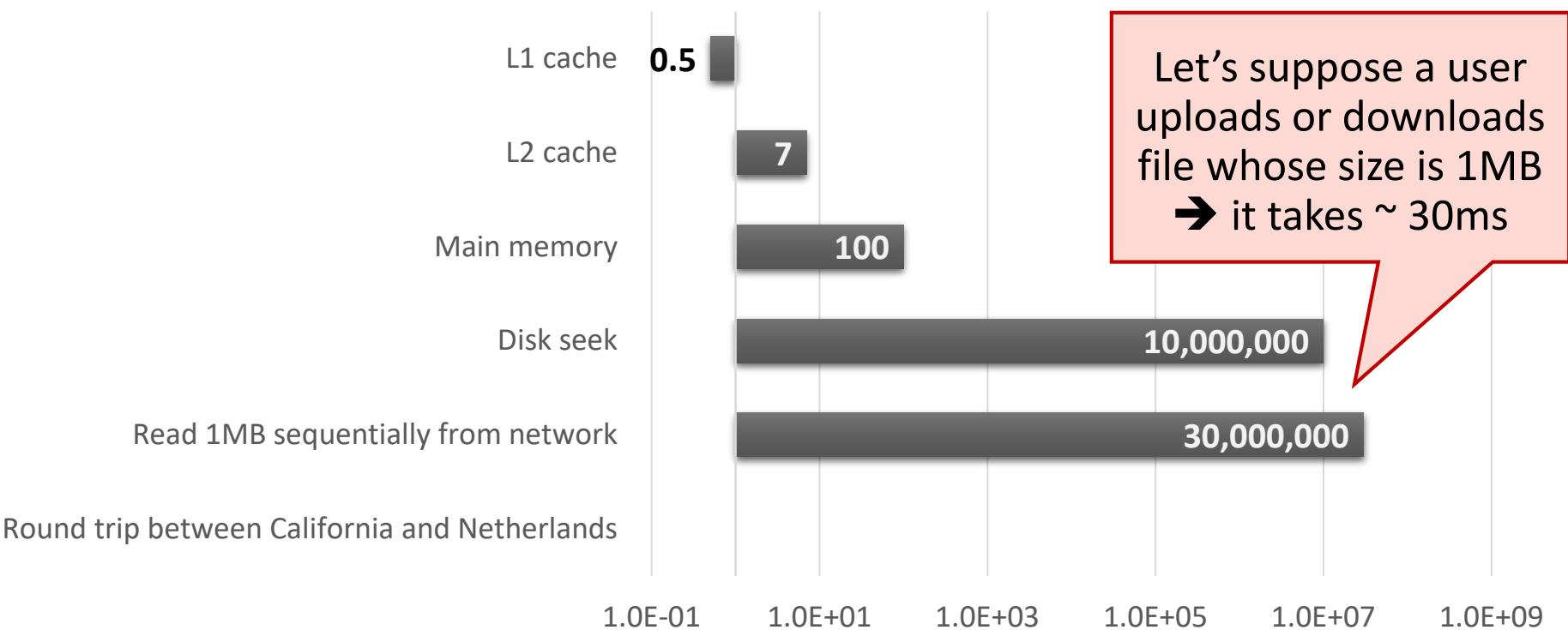
Numbers Real Word Engineers SHOULD Know (ns)



Read 1MB sequentially from network
Round trip between California and Netherlands

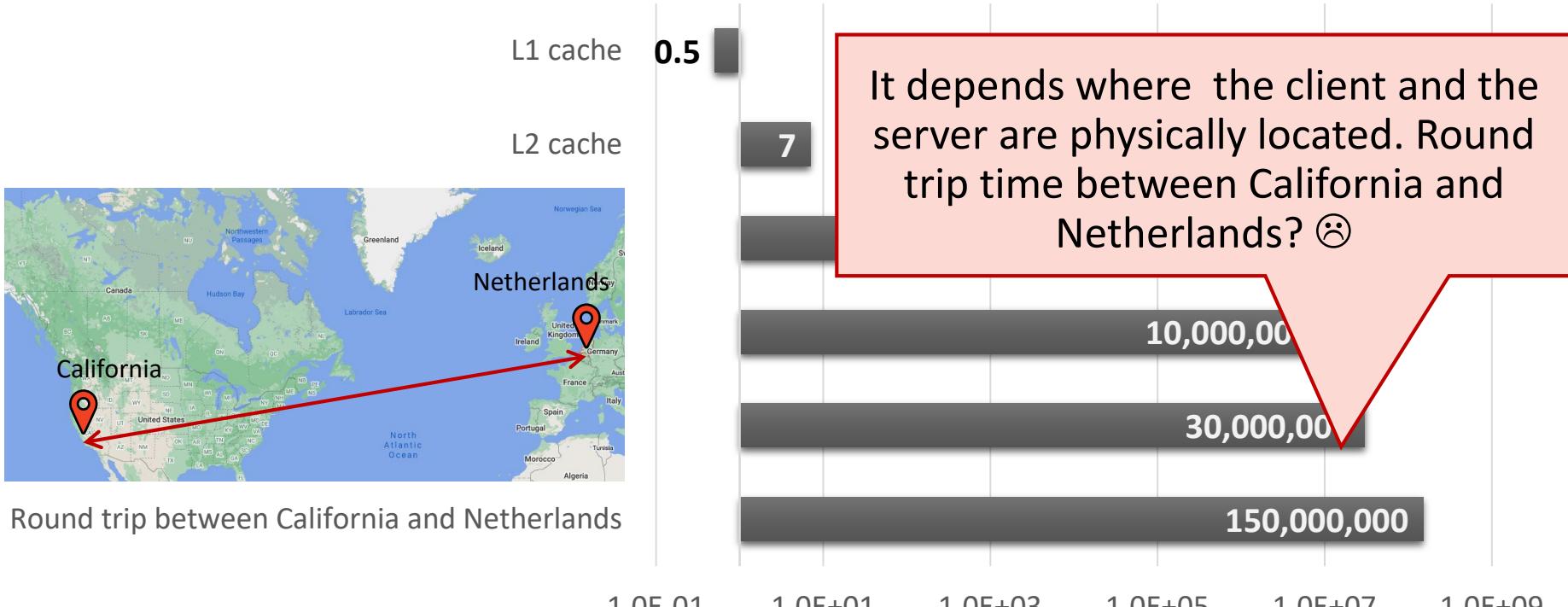
Network Is Slow

Numbers Real Word Engineers SHOULD Know (ns)

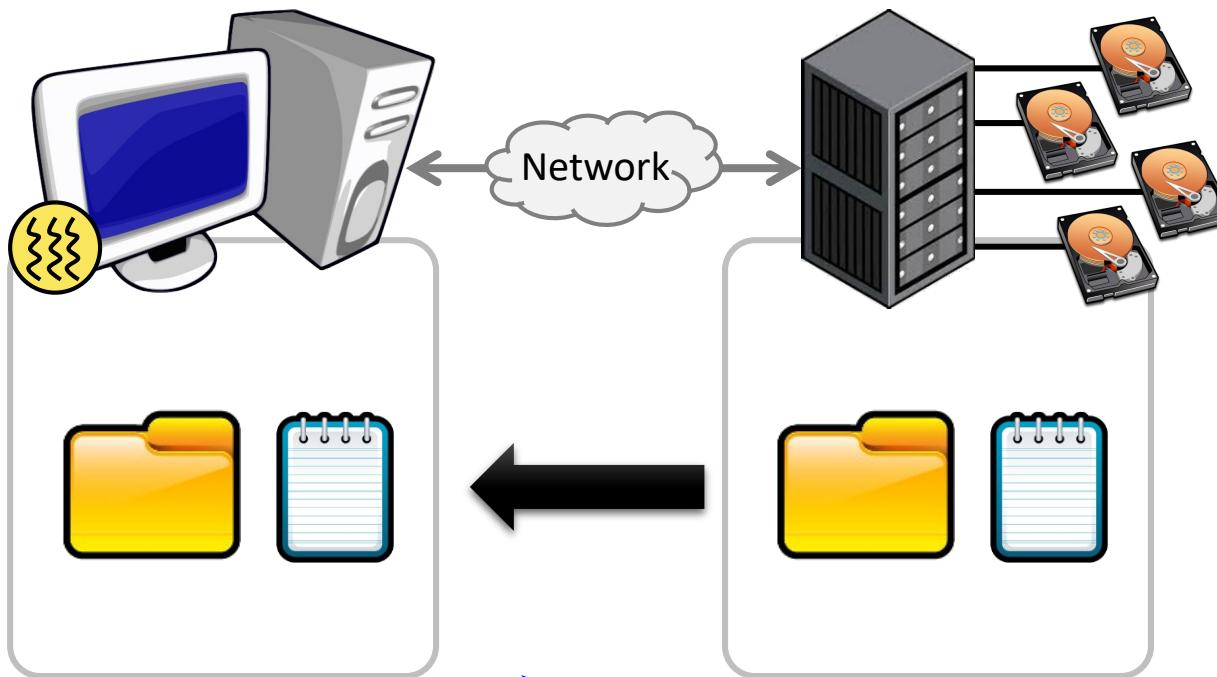


Network Is Slow

Numbers Real Word Engineers **SHOULD** Know (ns)



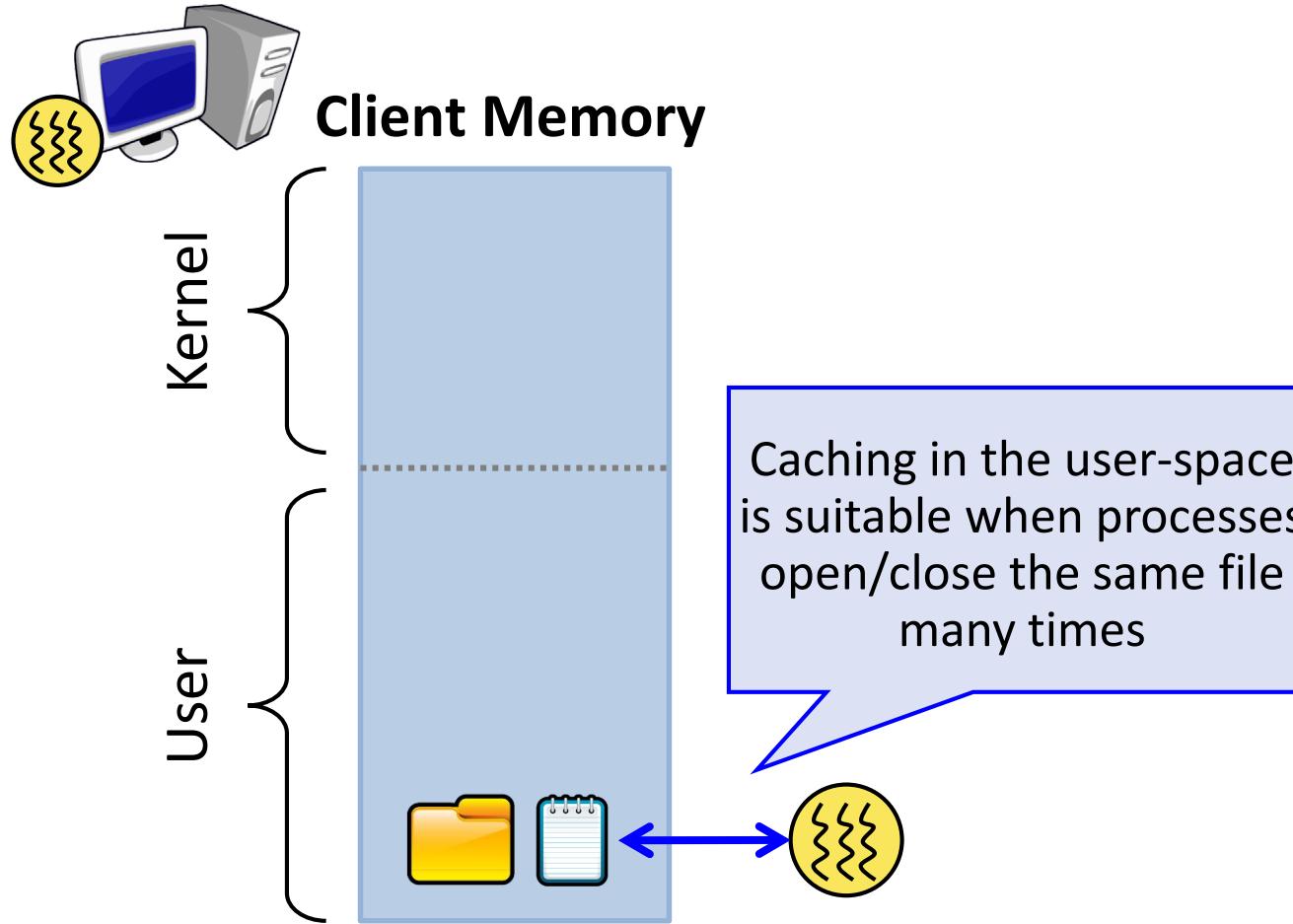
Implementation: Caching



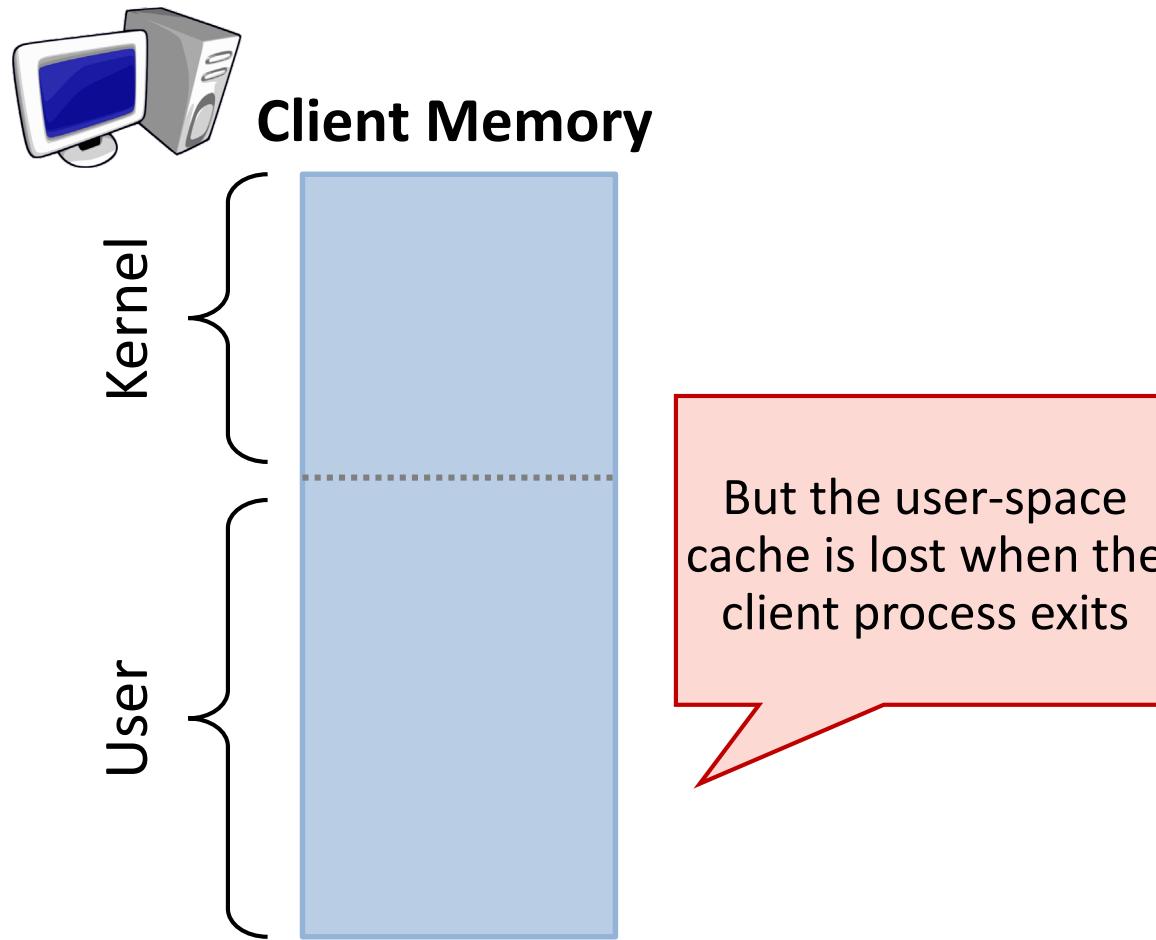
Client caching can
reduce server I/O and
network traffic ☺



Option #1: Cache Within User Process

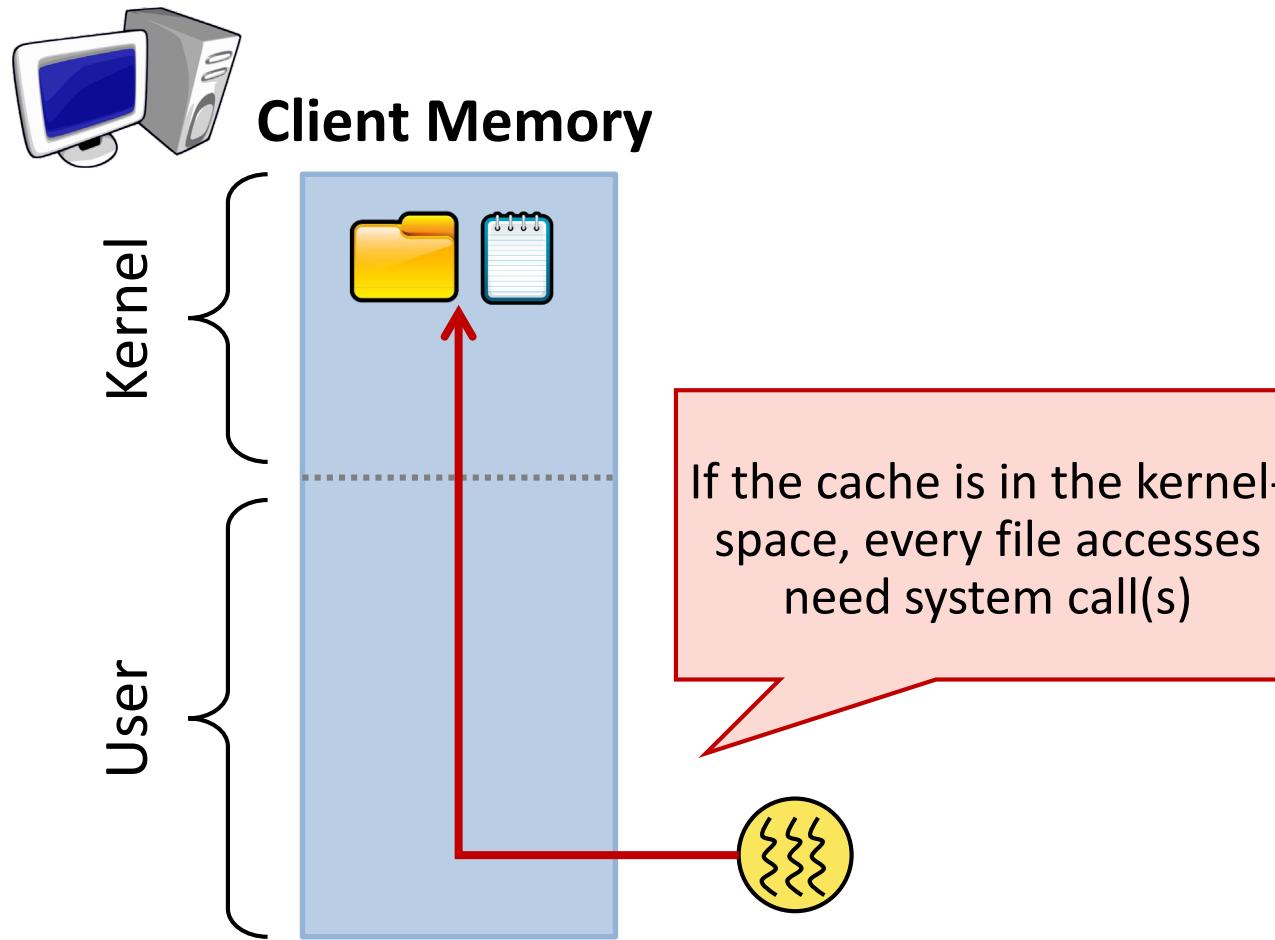


Option #1: Cache Within User Process



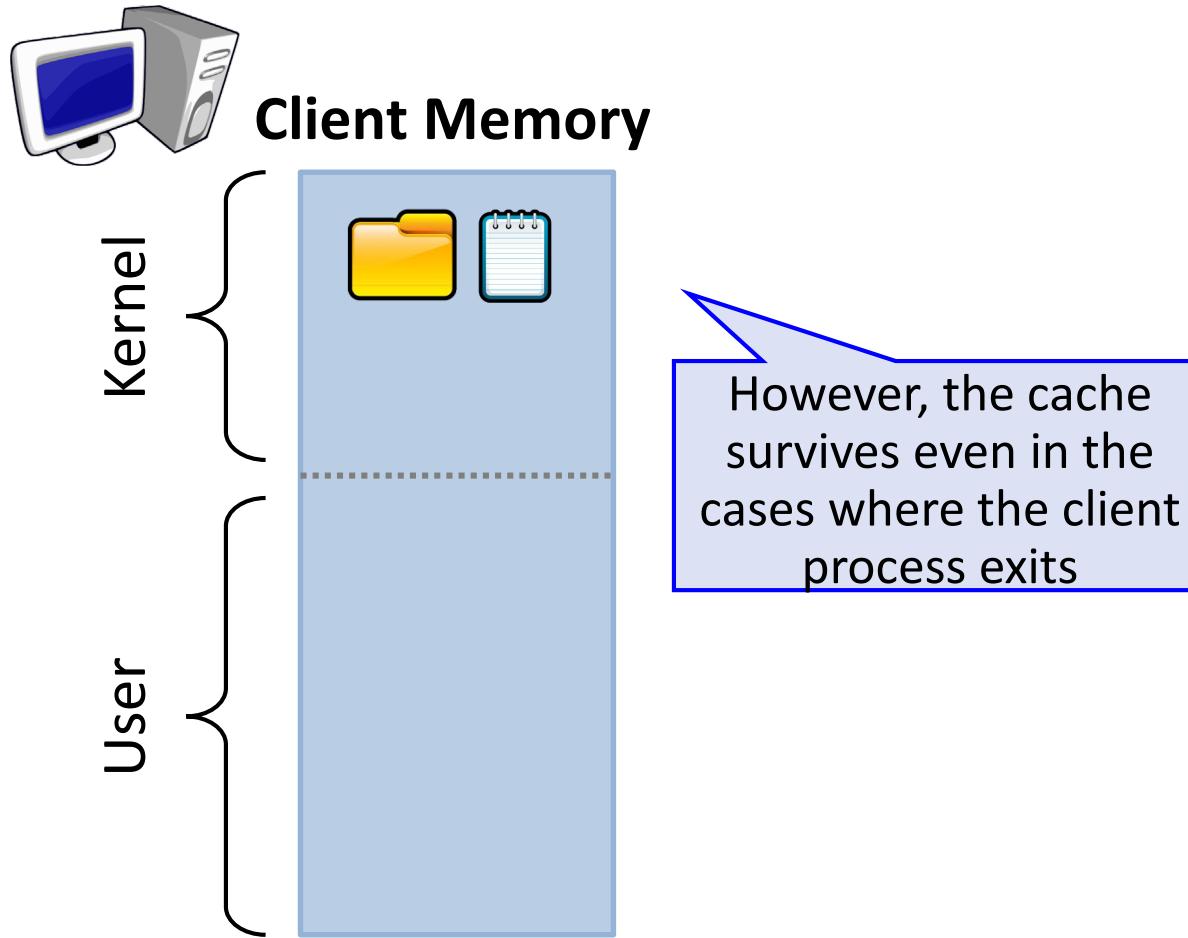


Option #2: Cache In the Kernel

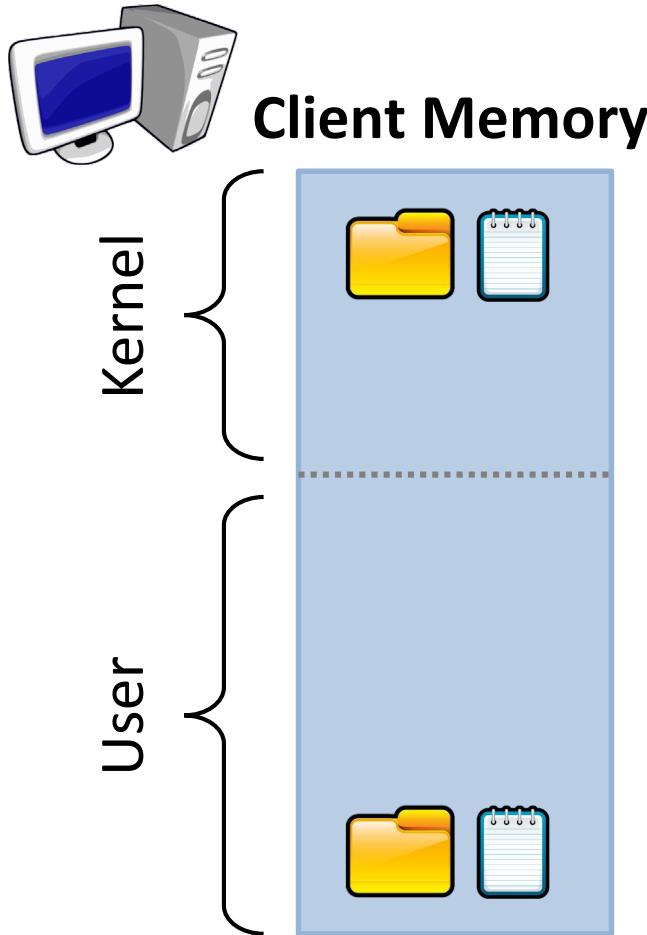




Option #2: Cache In the Kernel



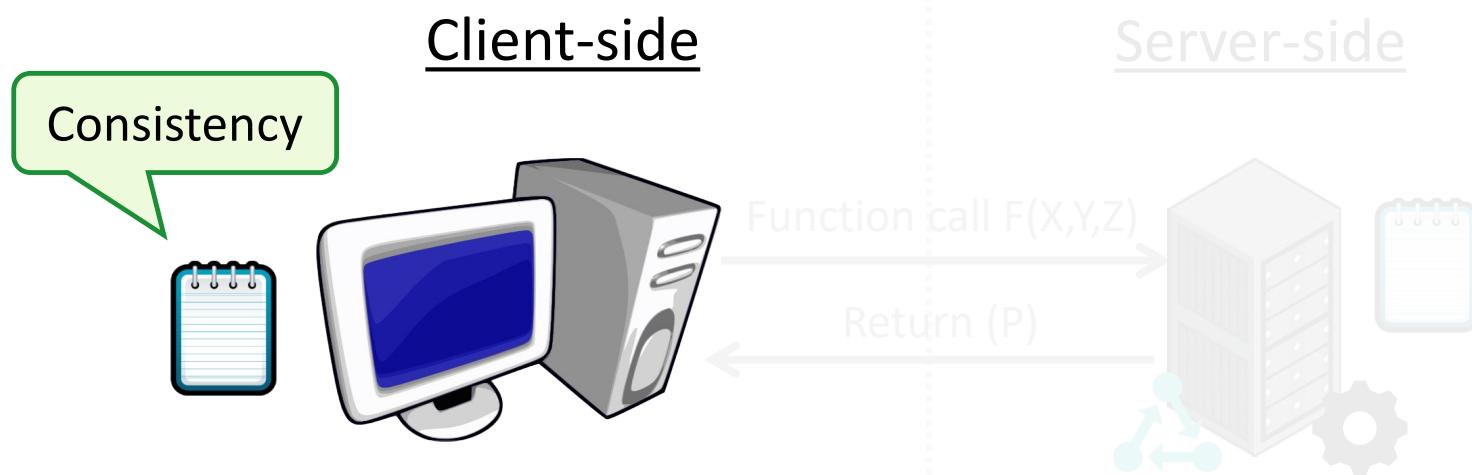
Option #2: Cache In the Kernel



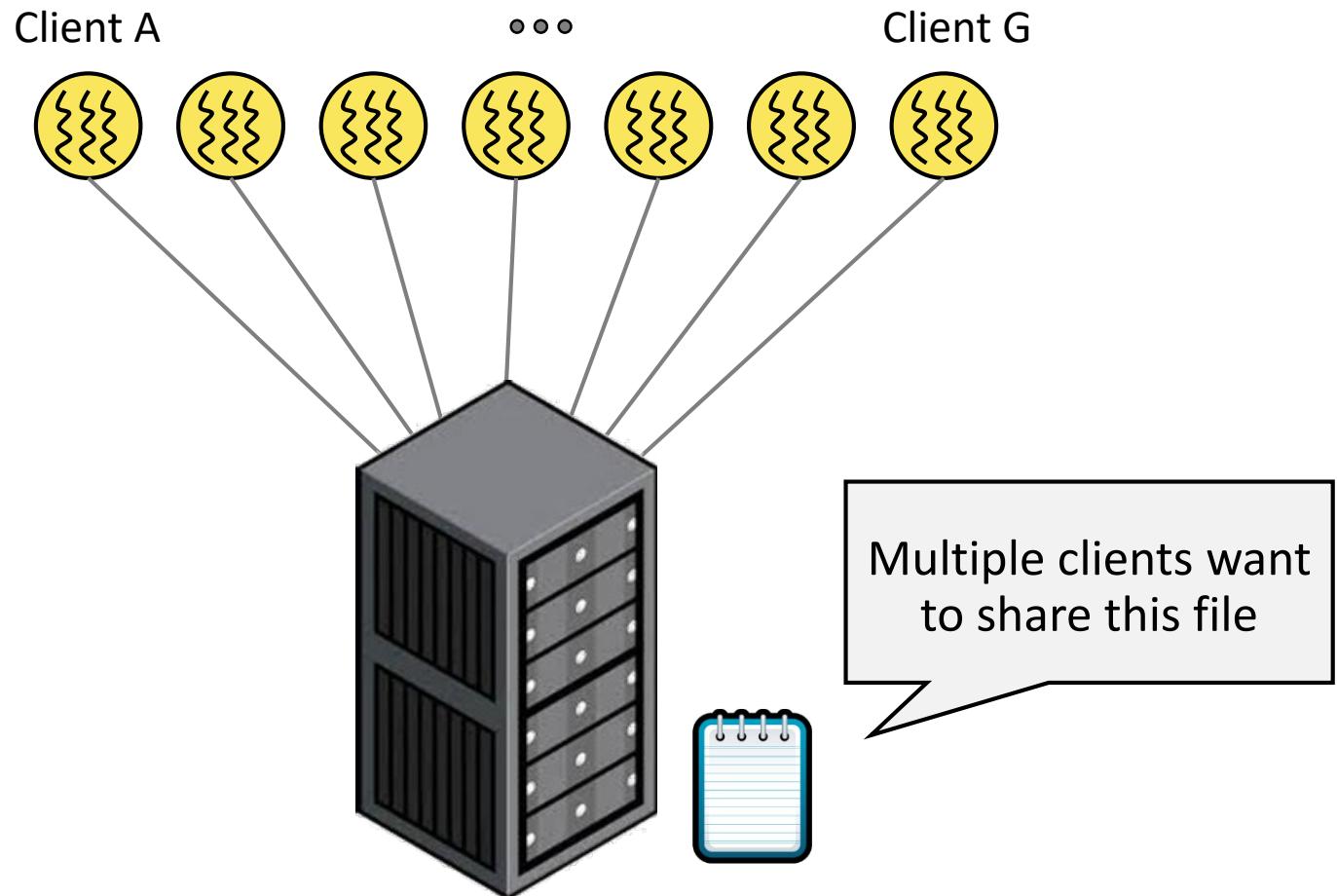
There is trade-off between user-space and kernel-space caching. It thus require understanding the workload and runtime behaviors that you consider for your DFS (rarely/frequently access the same file)

With this client caching, user can achieve **performance transparency** 😊

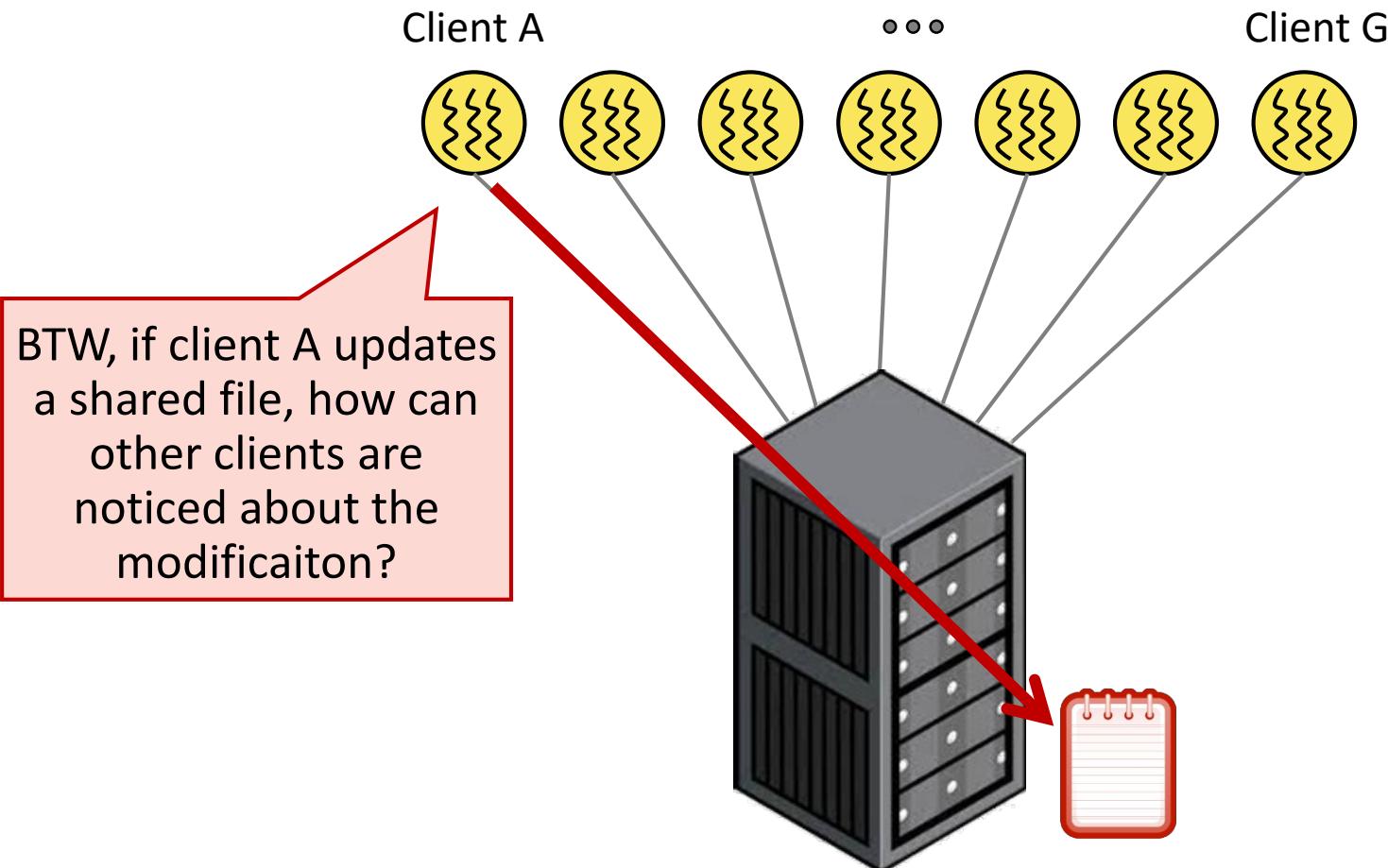
Client-Side #3: Consistency



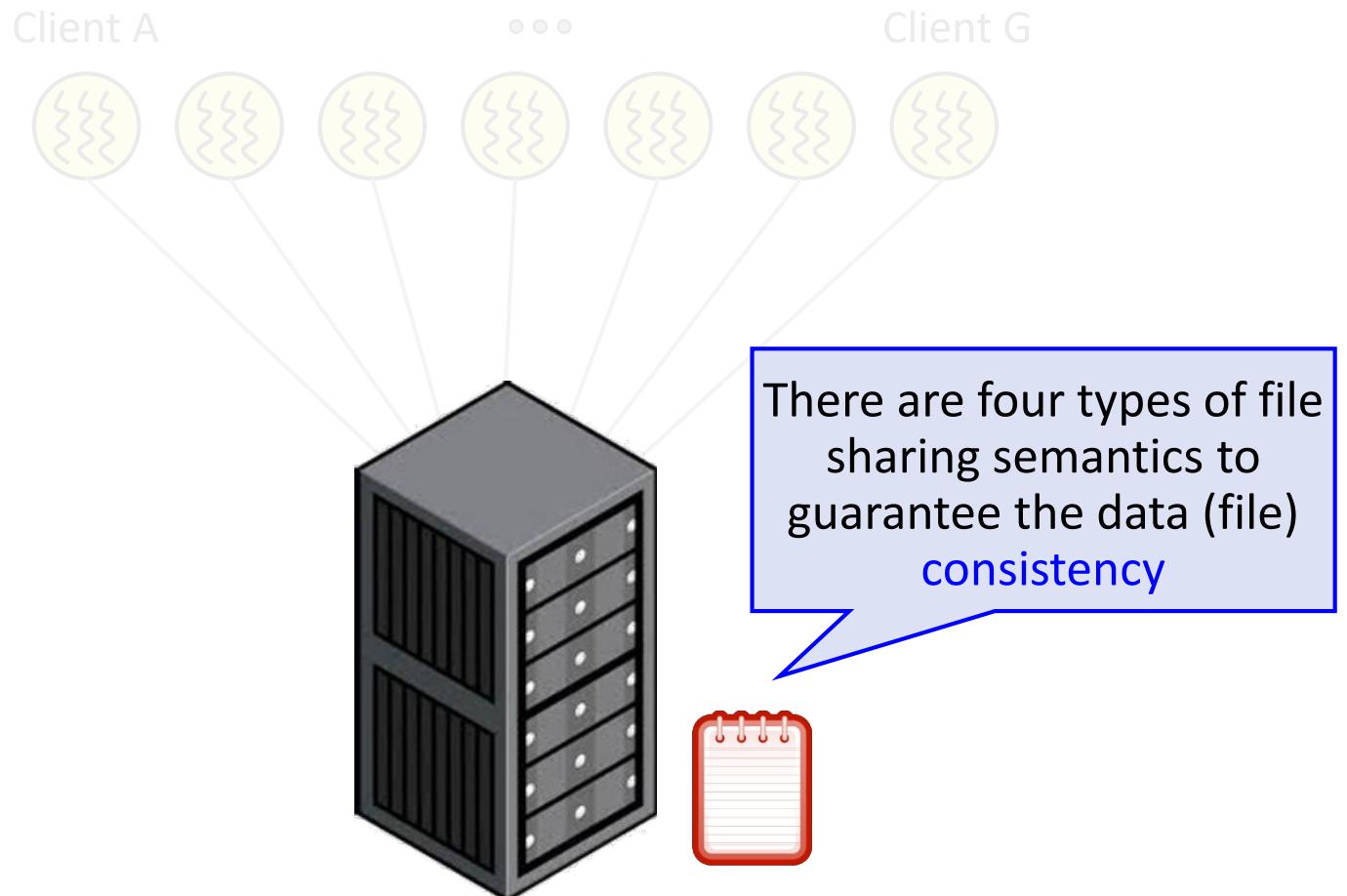
File Sharing



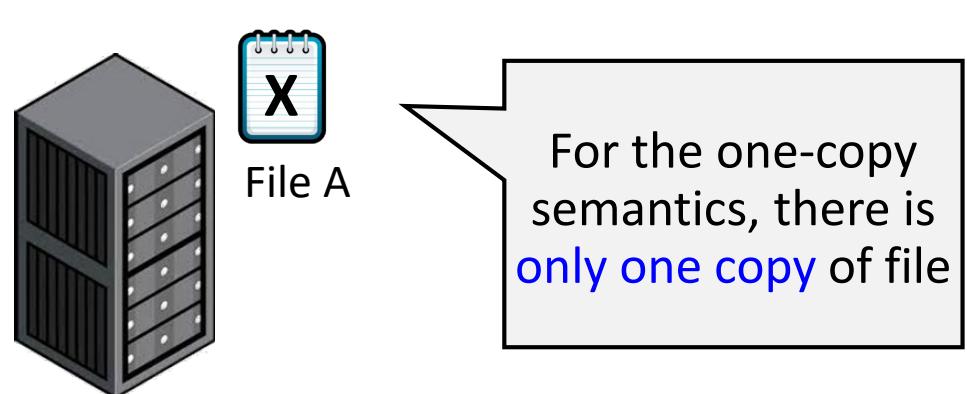
File Sharing



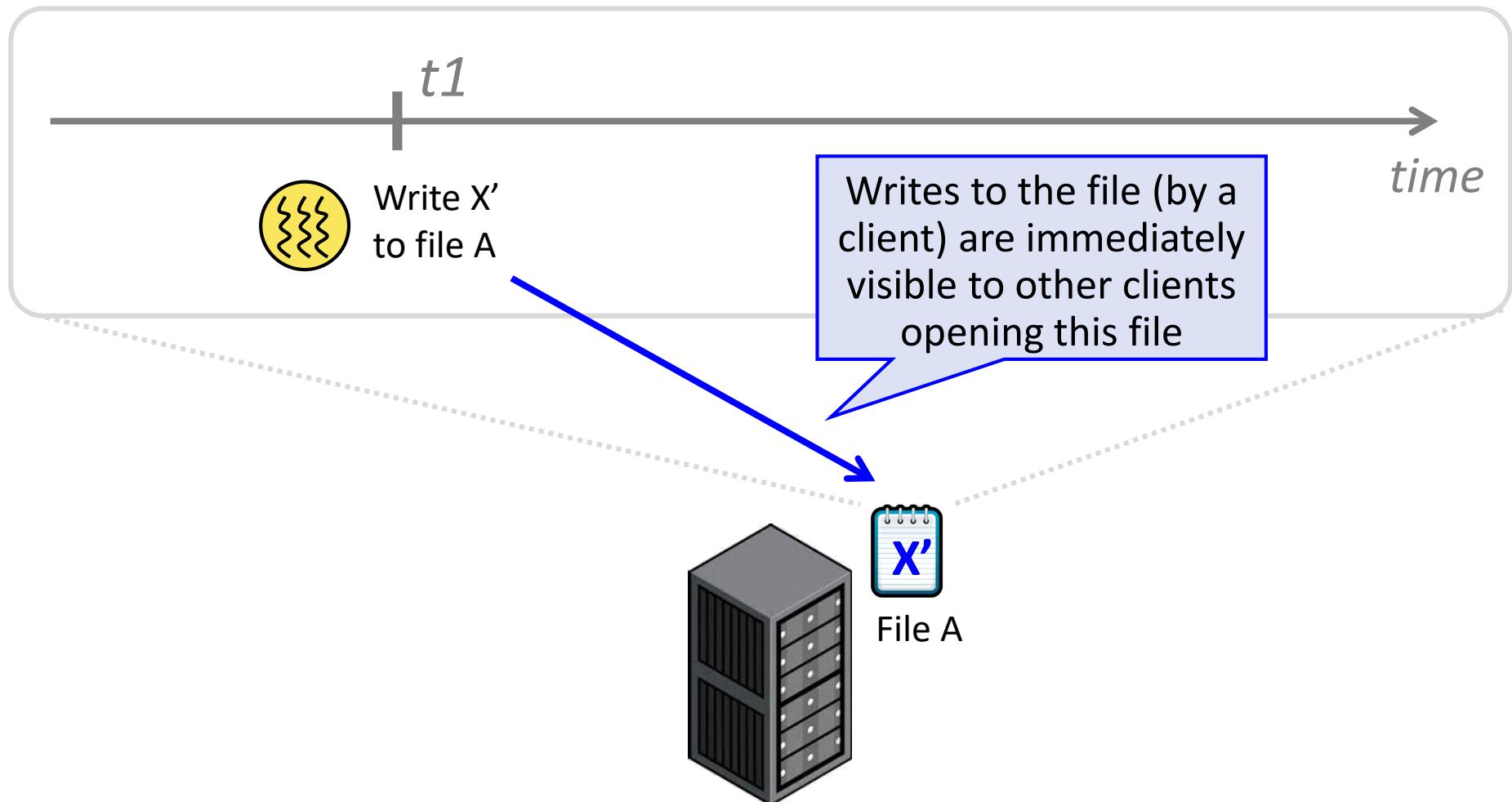
Implementation: File Sharing Semantics



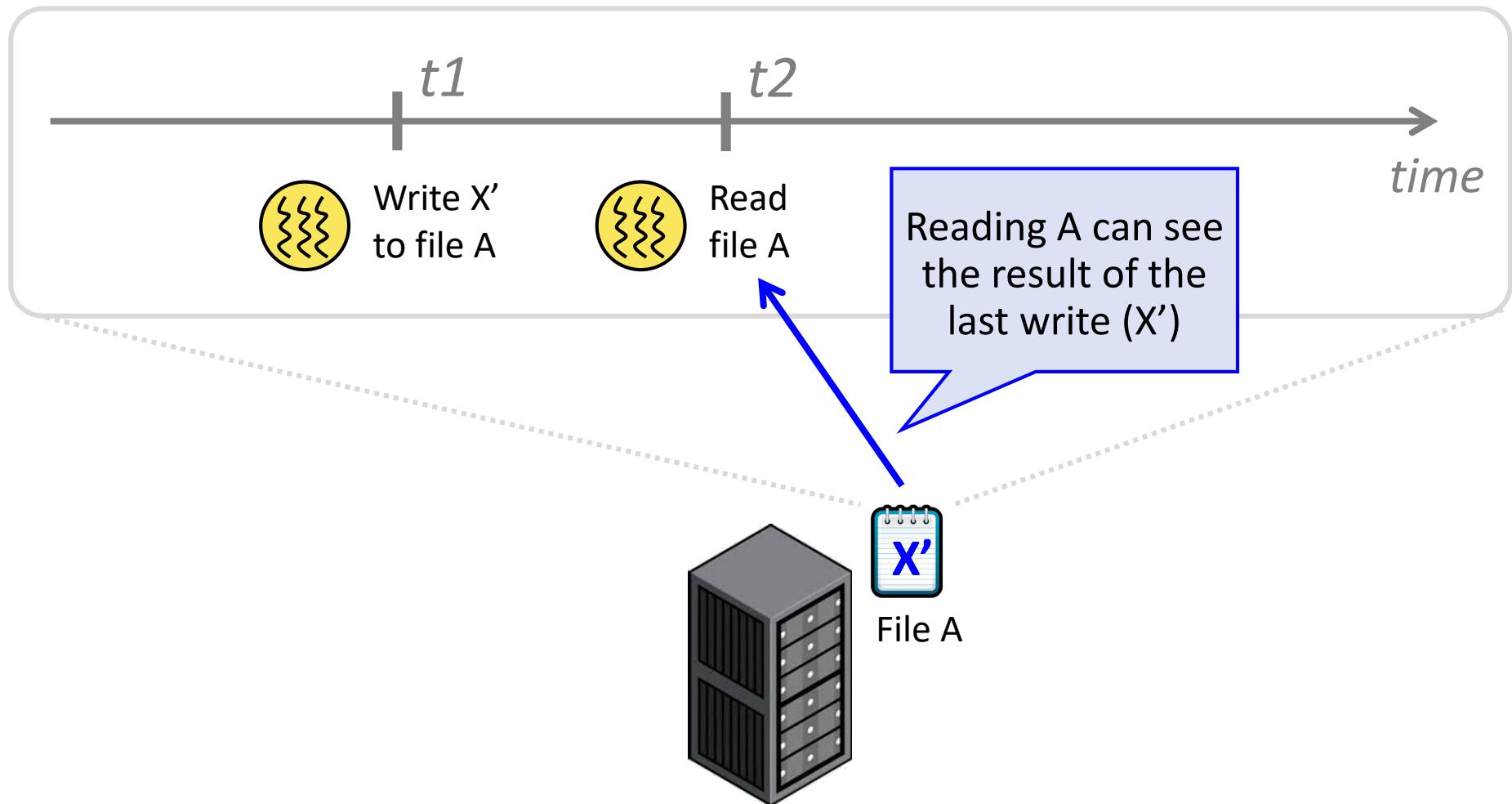
Option #1: One-Copy Semantics



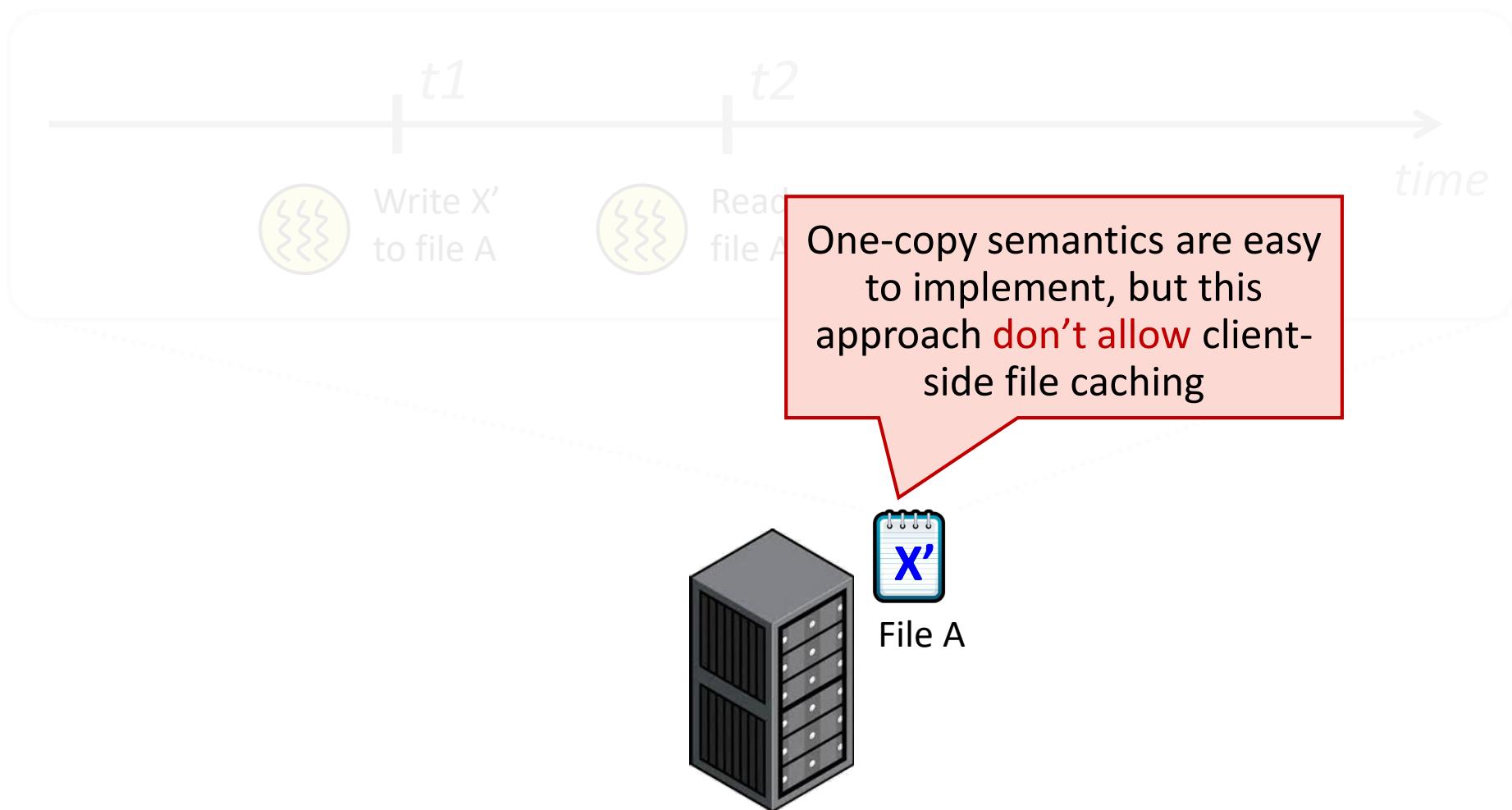
Option #1: One-Copy Semantics



Option #1: One-Copy Semantics

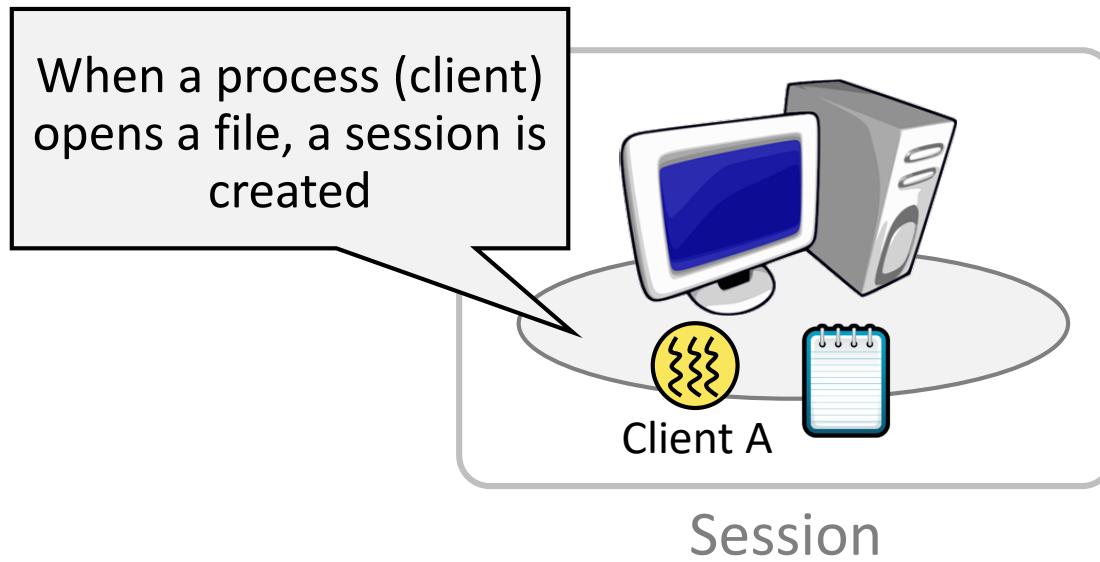


Option #1: One-Copy Semantics

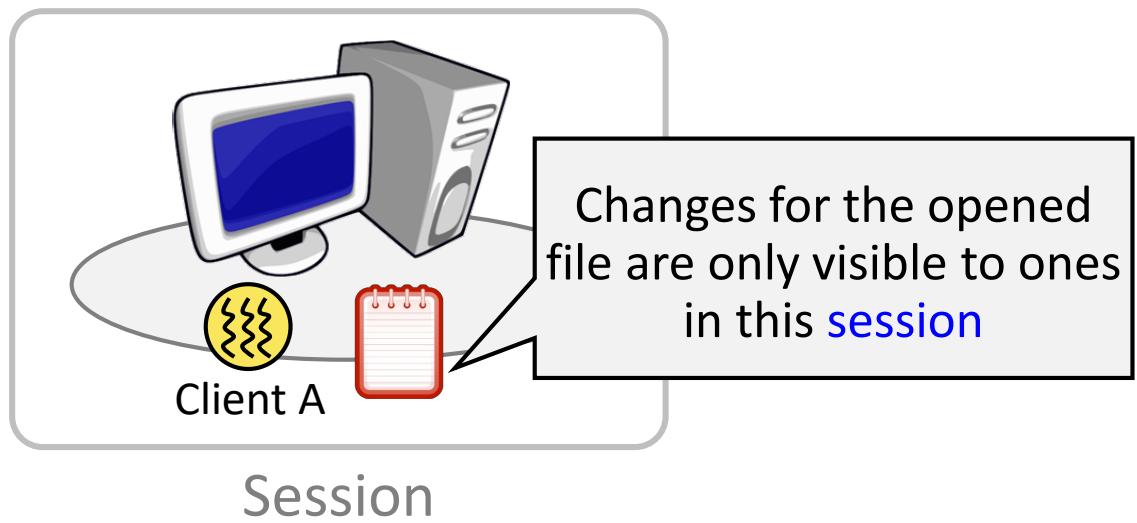




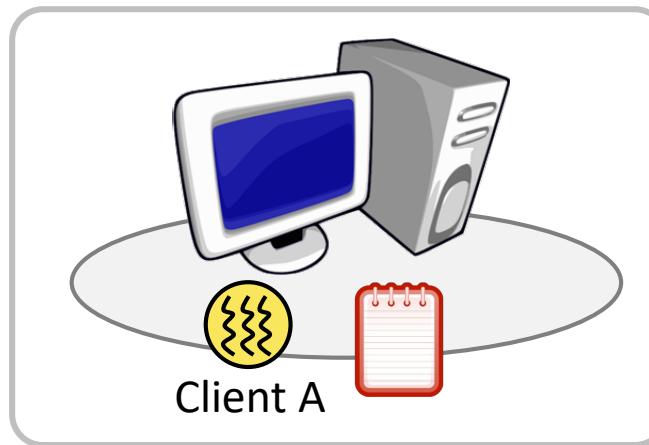
Option #2: Session Semantics



Option #2: Session Semantics



Option #2: Session Semantics

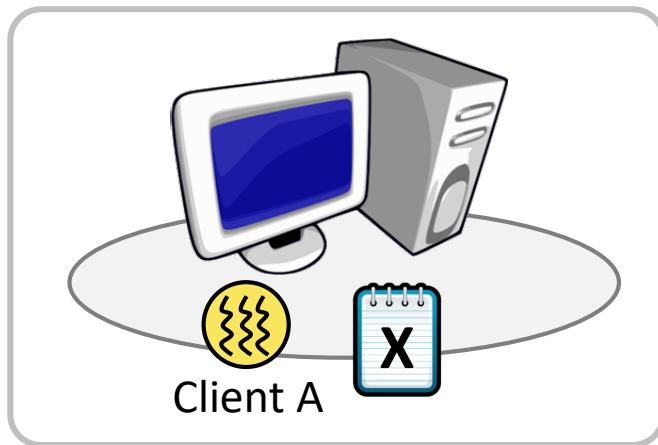


Session

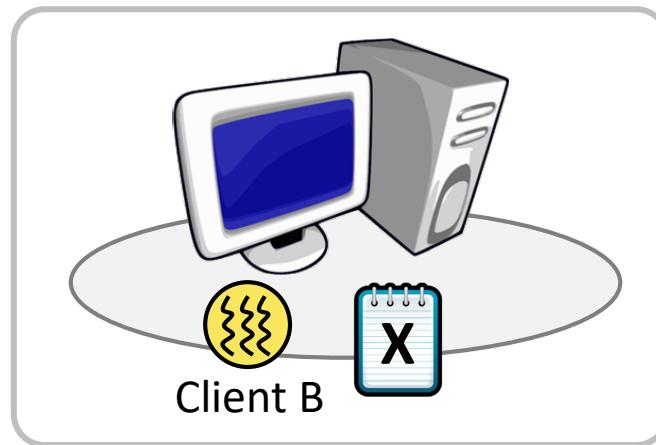


When the file is closed, changes are visible to other clients (processes); the session sends the closed file back to the server

Option #2: Session Semantics

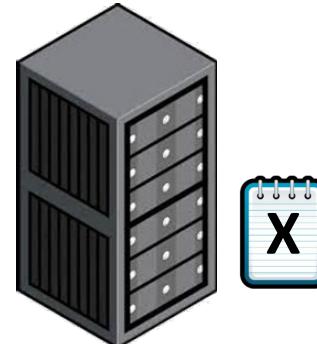


Session

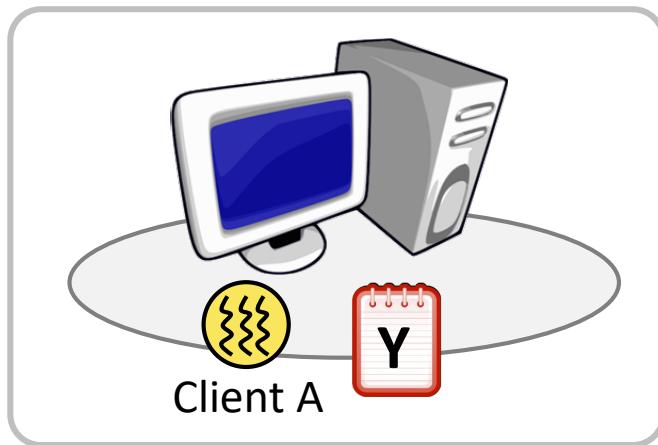


Session

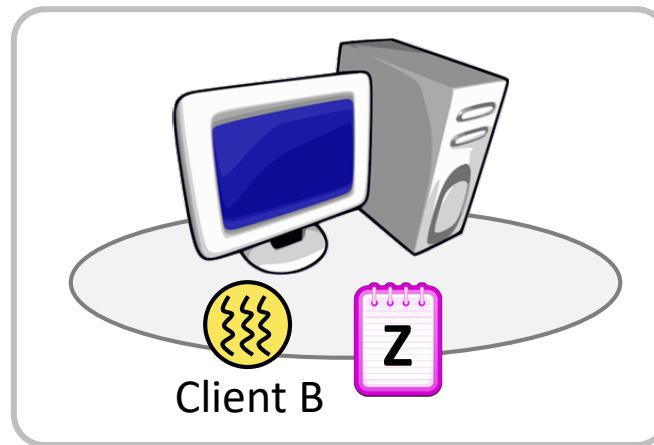
What if two or more clients are caching and modifying a file?



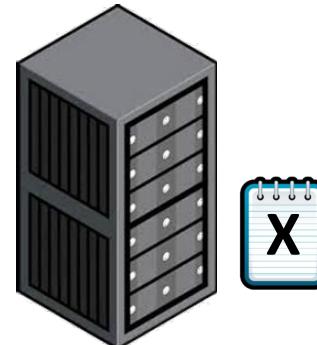
Option #2: Session Semantics



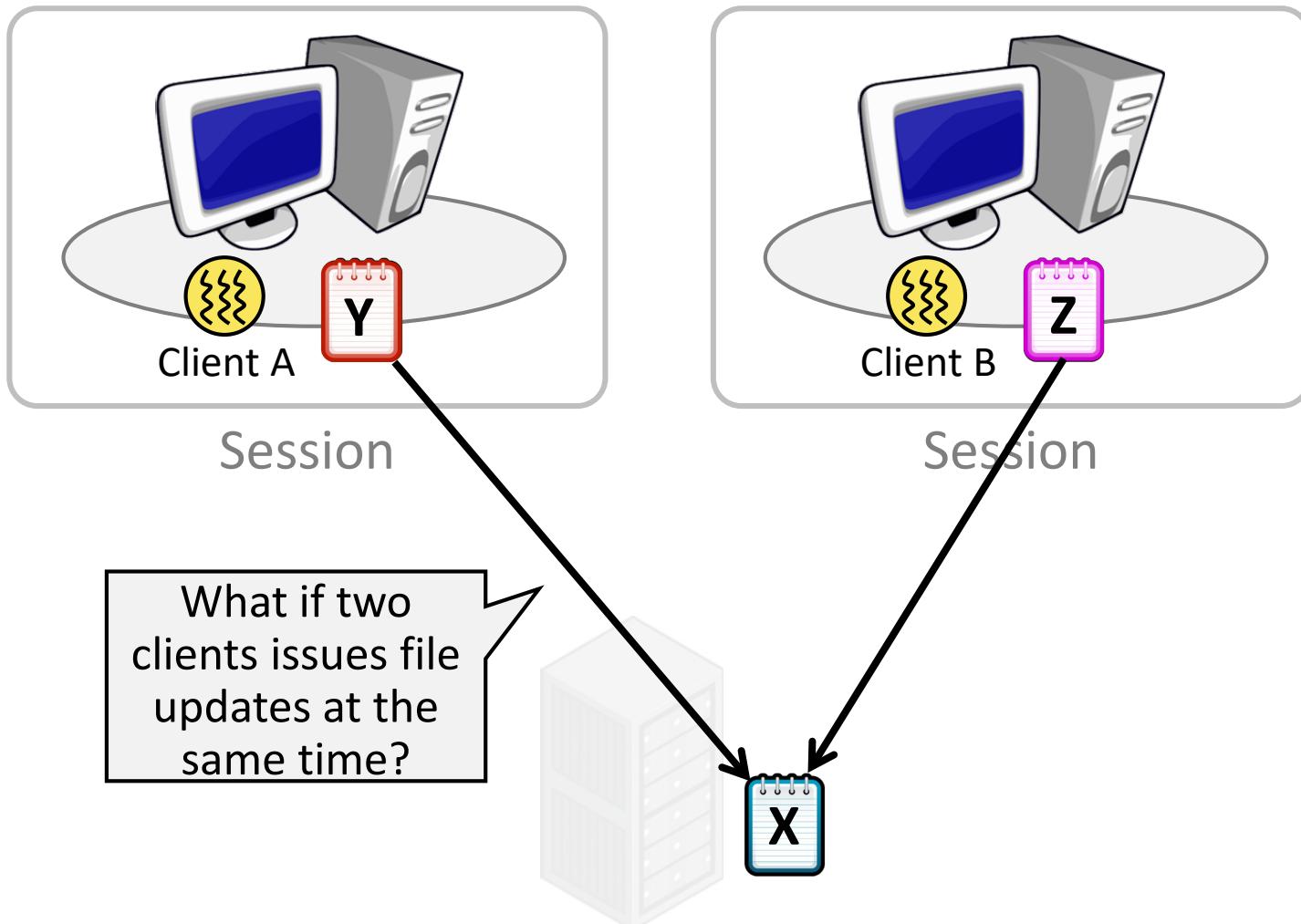
Session



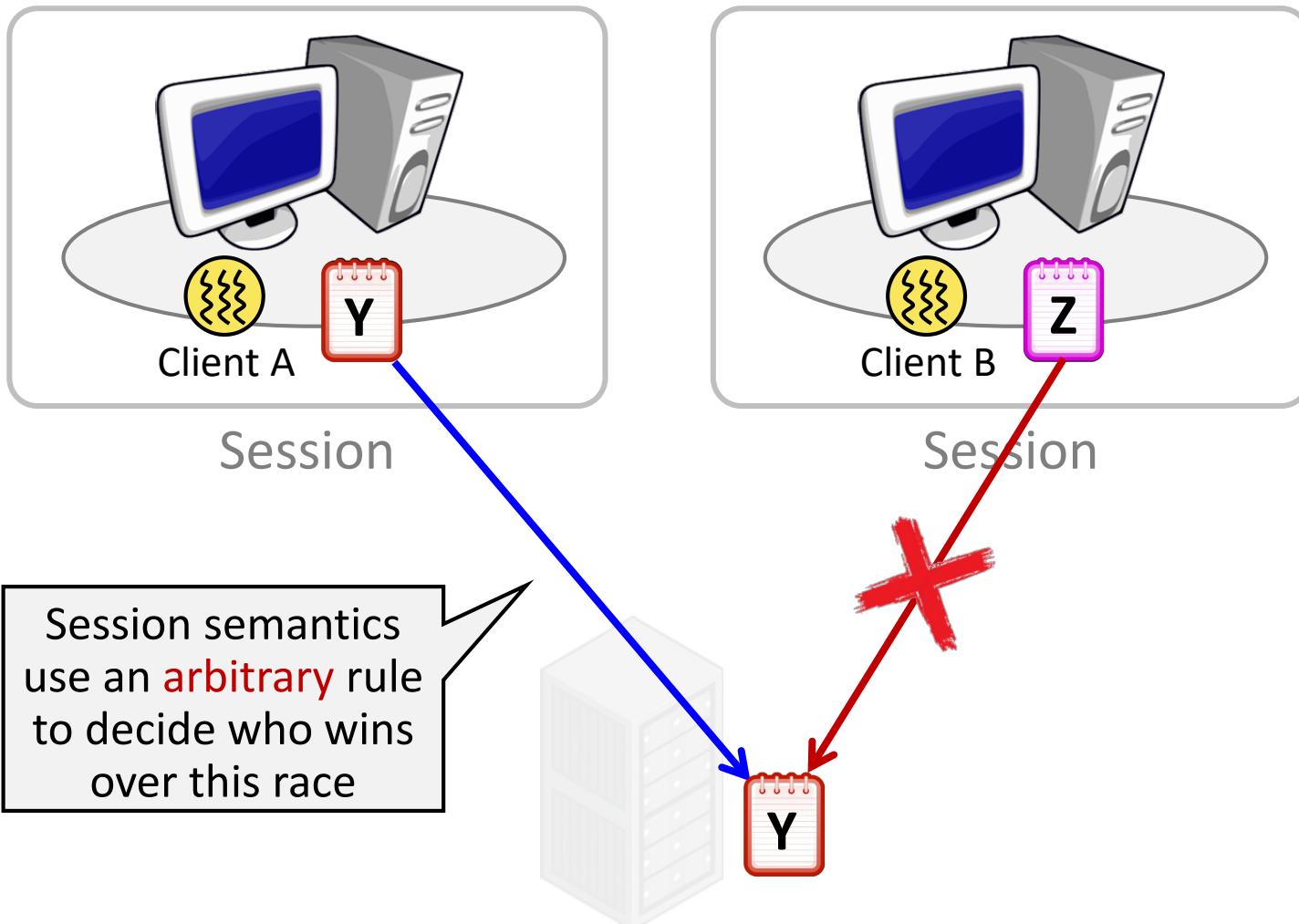
Session



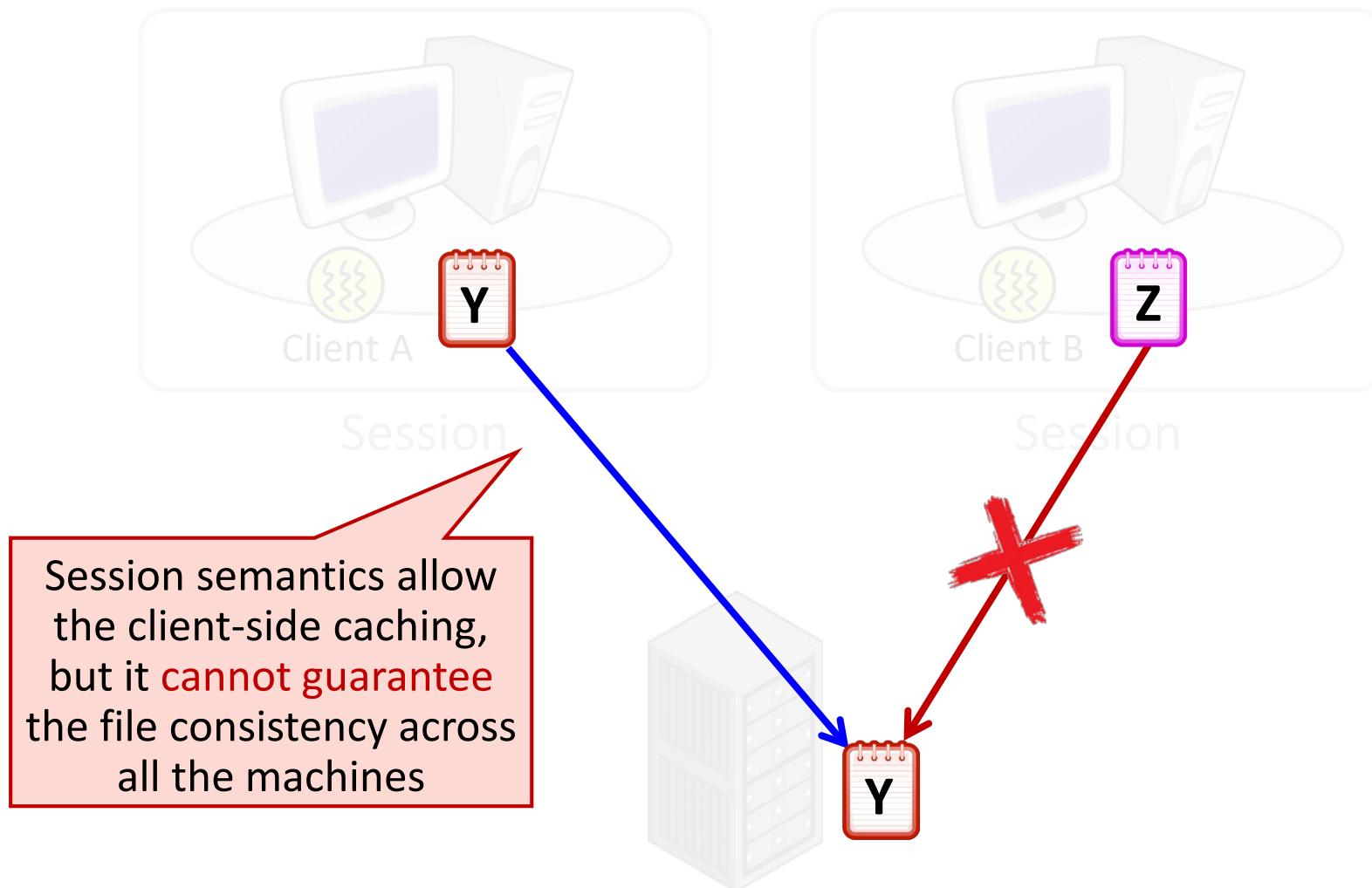
Option #2: Session Semantics



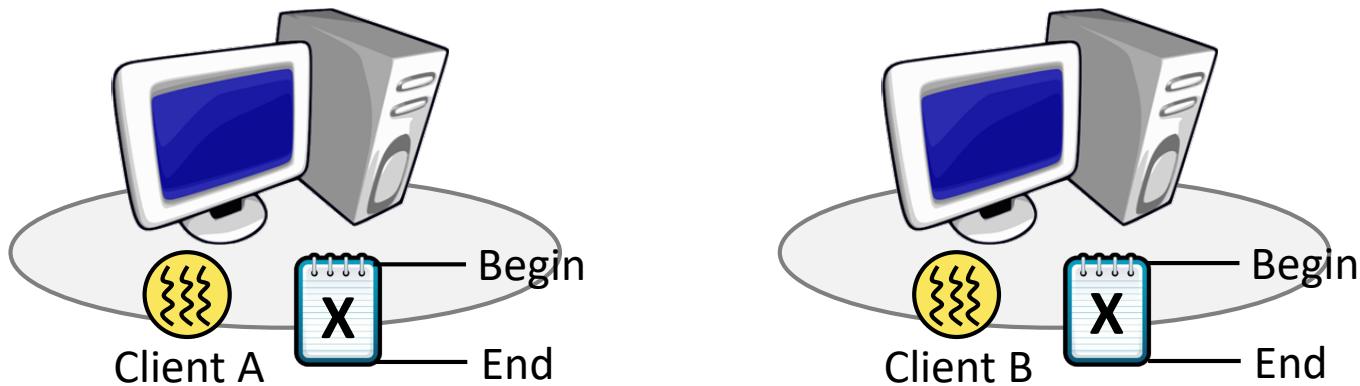
Option #2: Session Semantics



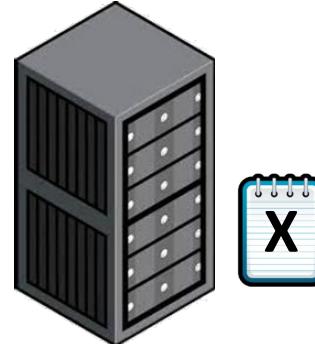
Option #2: Session Semantics



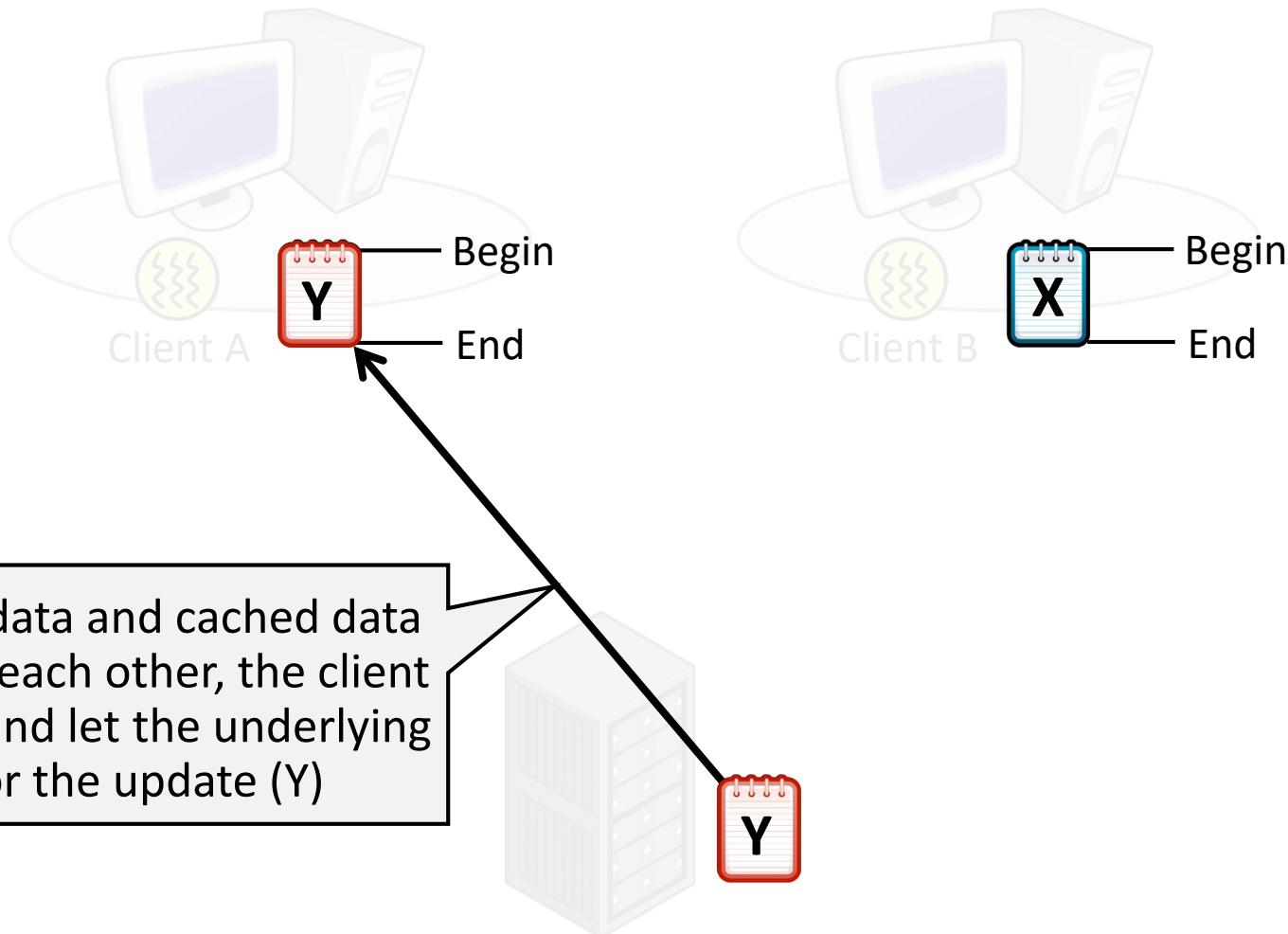
Option #3: Transaction Semantics



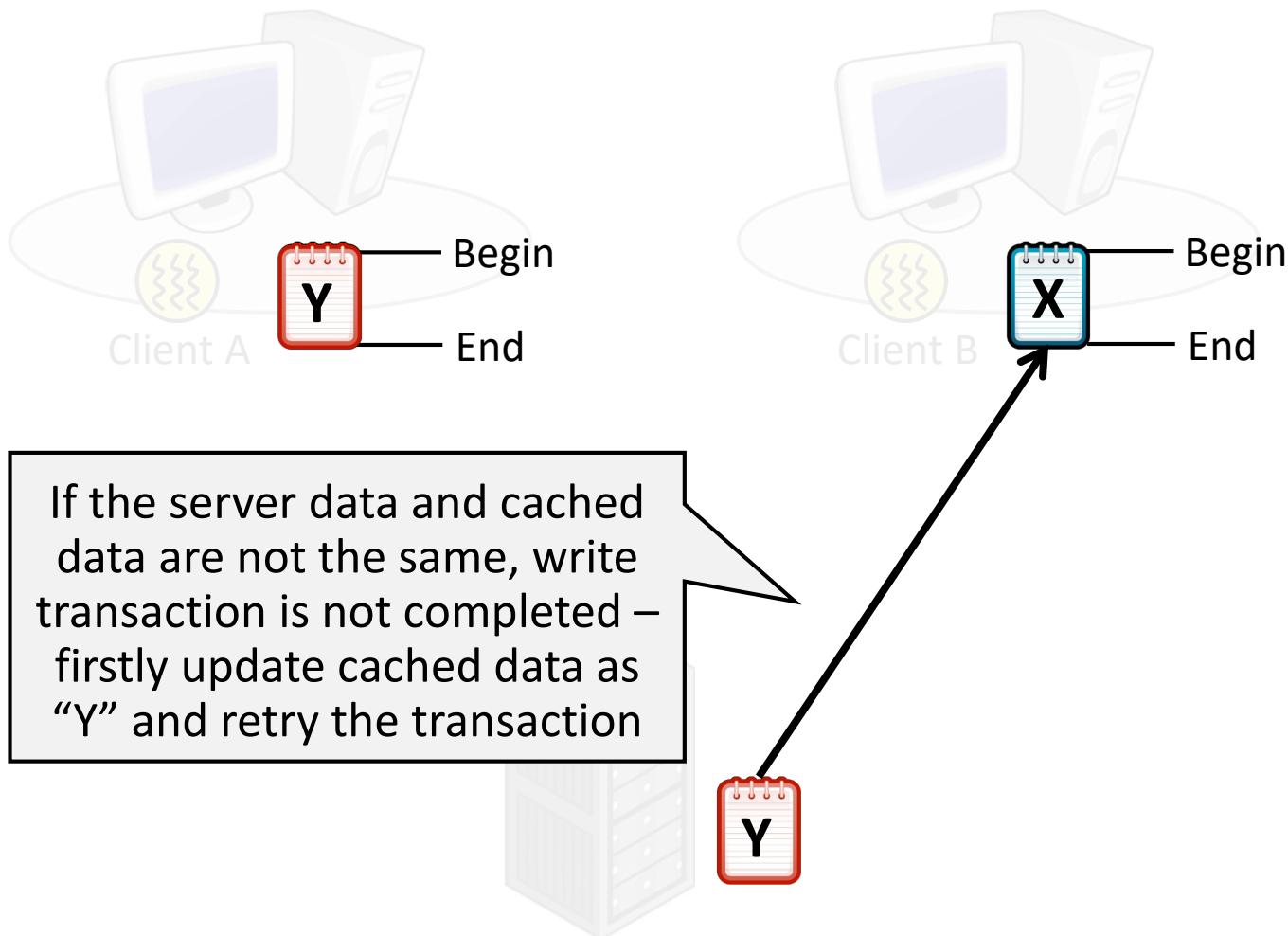
Unlike session semantics, all file changes in the transaction semantics are distinguished by a Begin-and-End delimiter.



Option #3: Transaction Semantics

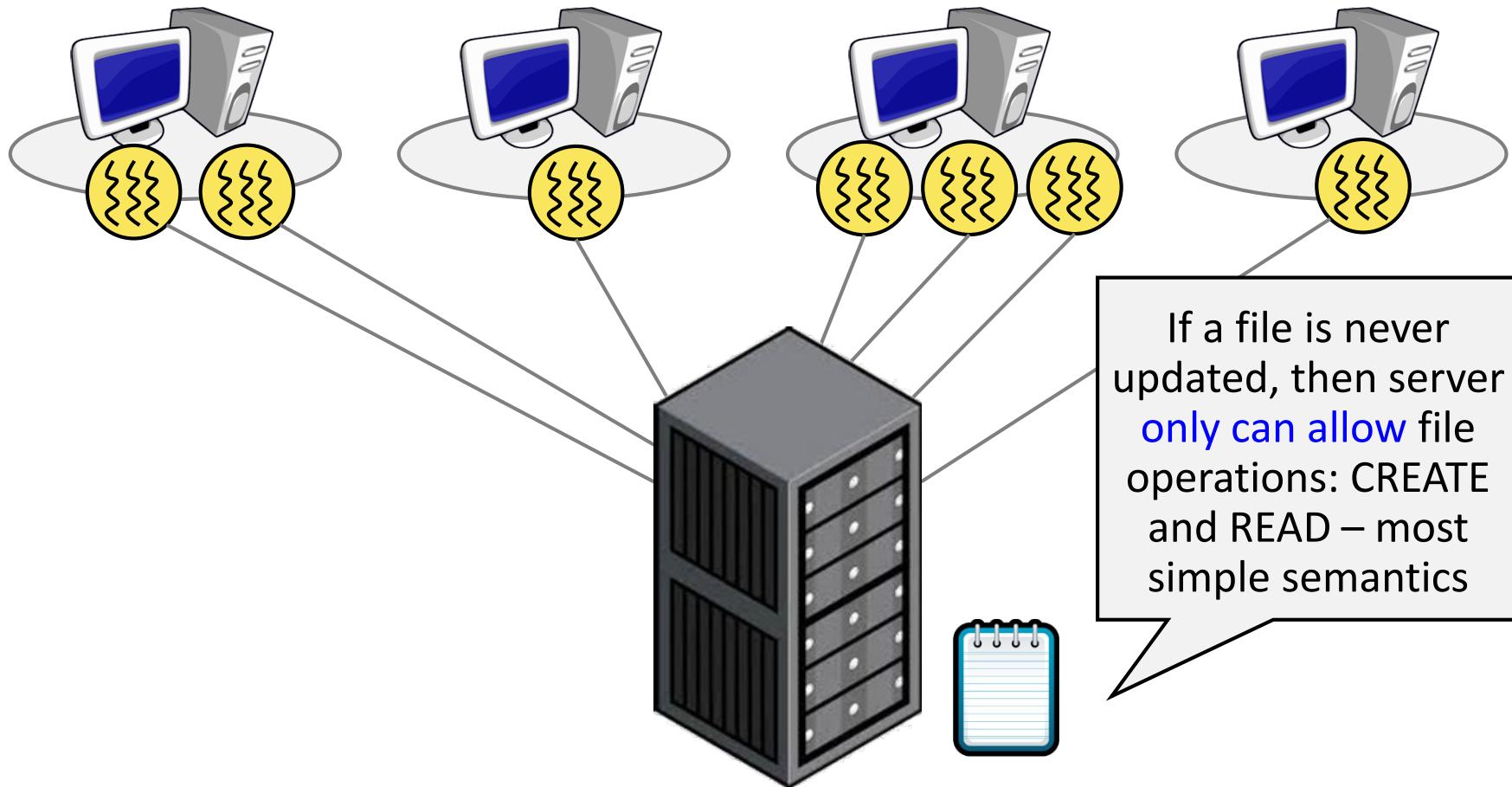


Option #3: Transaction Semantics





Option #4: NO File Updates Semantics



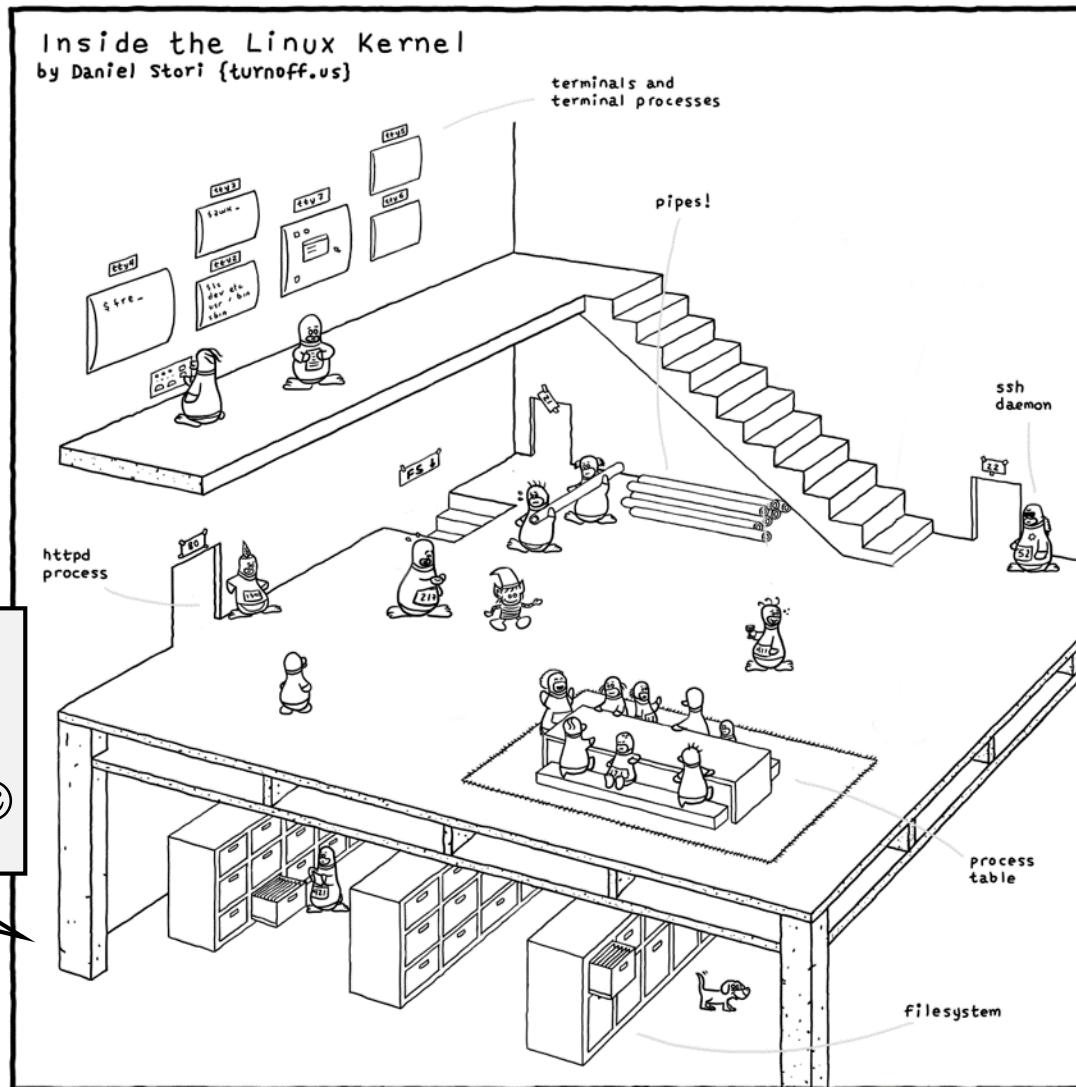


Summary of File Sharing Semantics

DFS can choose the appropriate file sharing semantics based on the file usage pattern (e.g., read-only, modifiable)

File Sharing Semantics	
One-copy Semantics (UNIX Semantics)	Every operation is instantly visible to others
Session Semantics	No changes are visible until the target file is closed
Transactions	Atomic updates
No Updates Semantics	No file updates are allowed

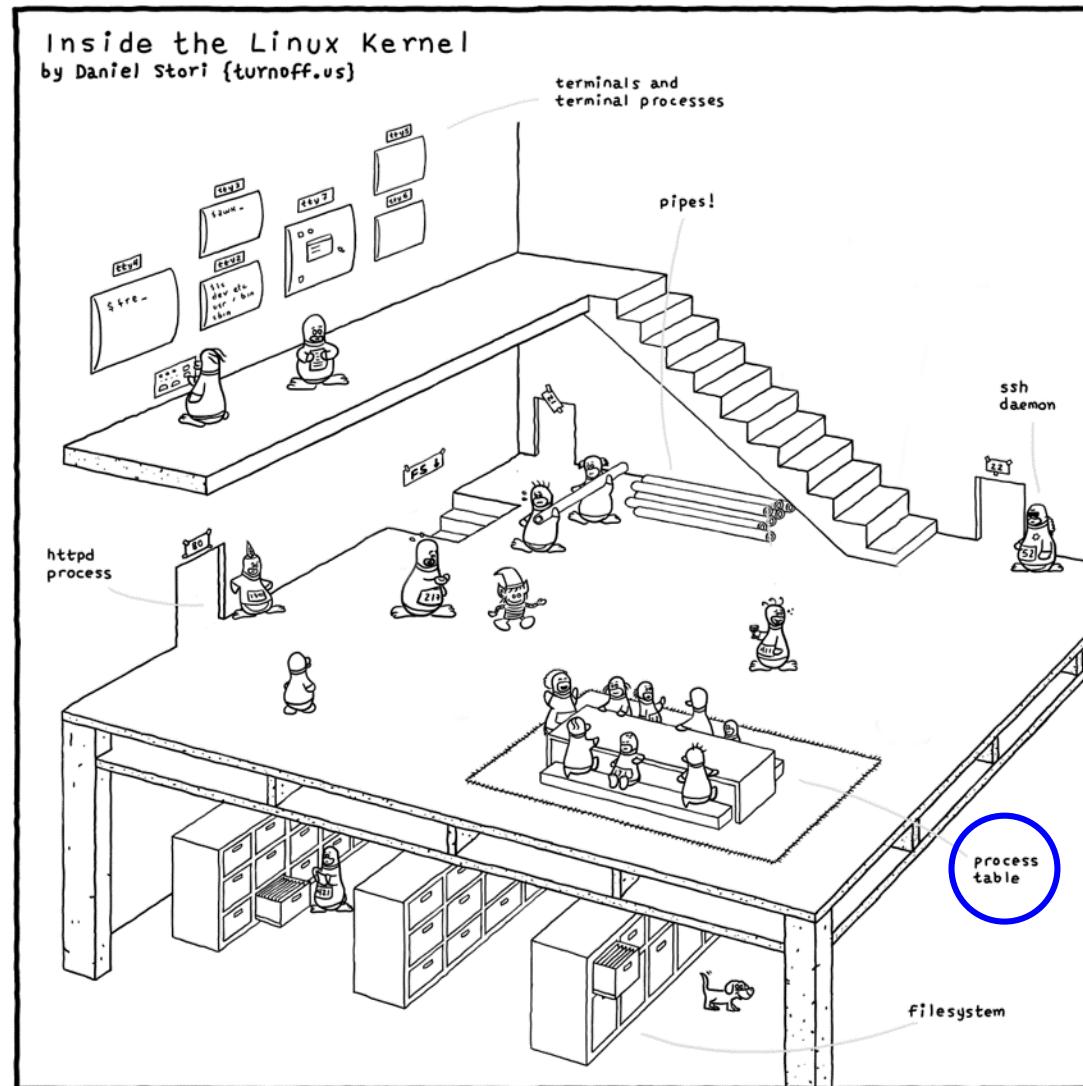
Inside The Linux Kernel



We finally understand the underground of operating system (archiving system) ☺

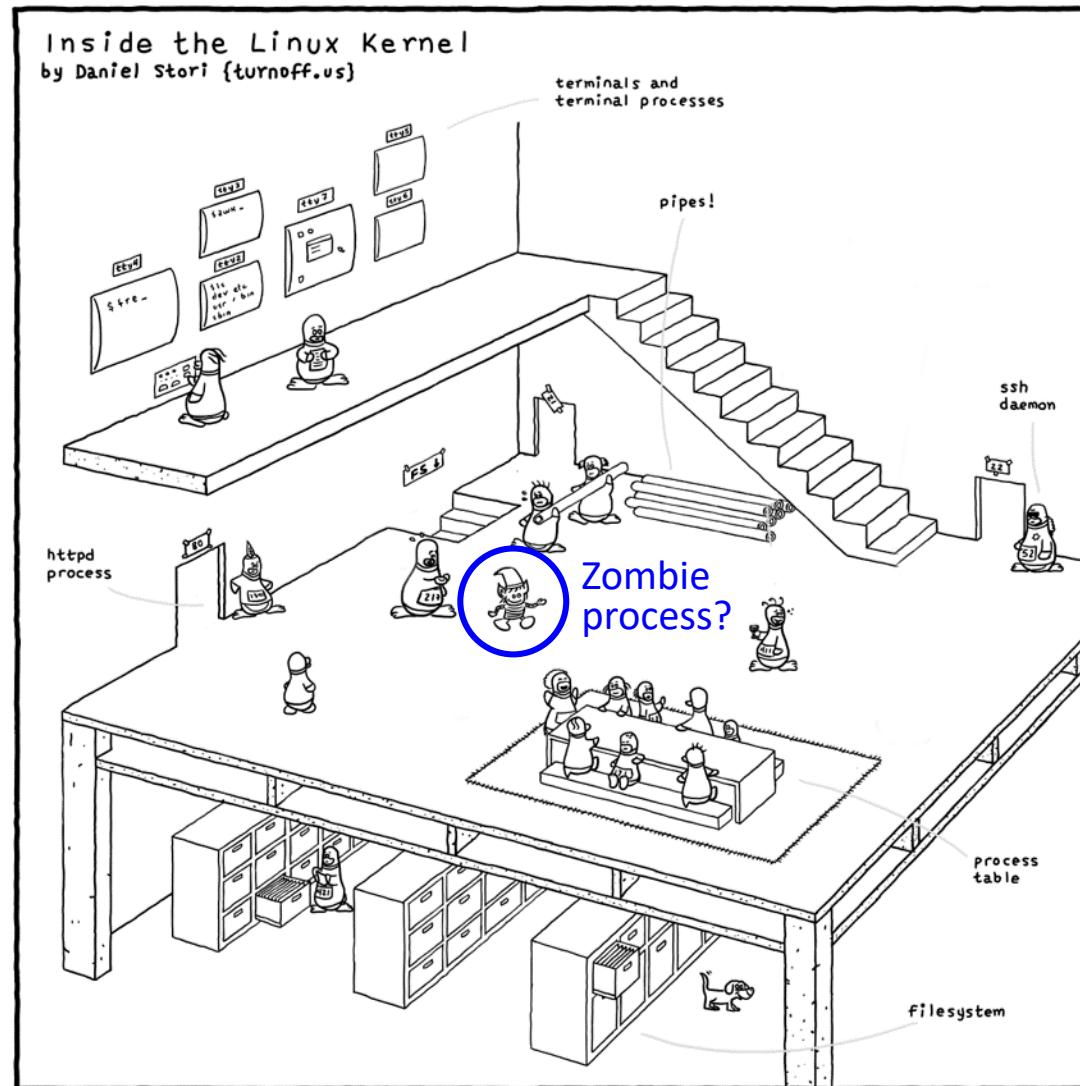
Inside The Linux Kernel

Do you remember
other parts of
operating system ?



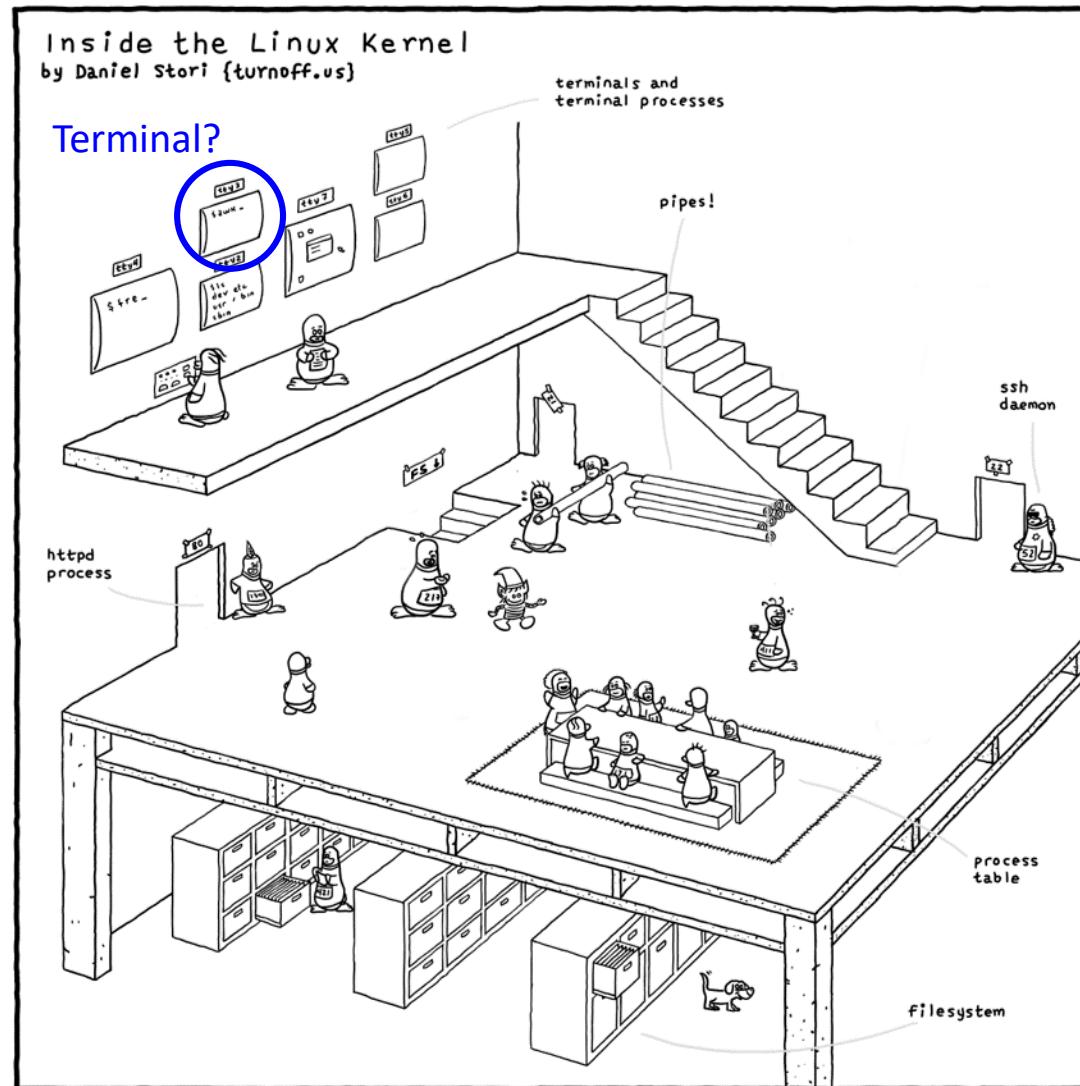
Inside The Linux Kernel

Do you remember
other parts of
operating system ?



Inside The Linux Kernel

Do you remember
other parts of
operating system ?

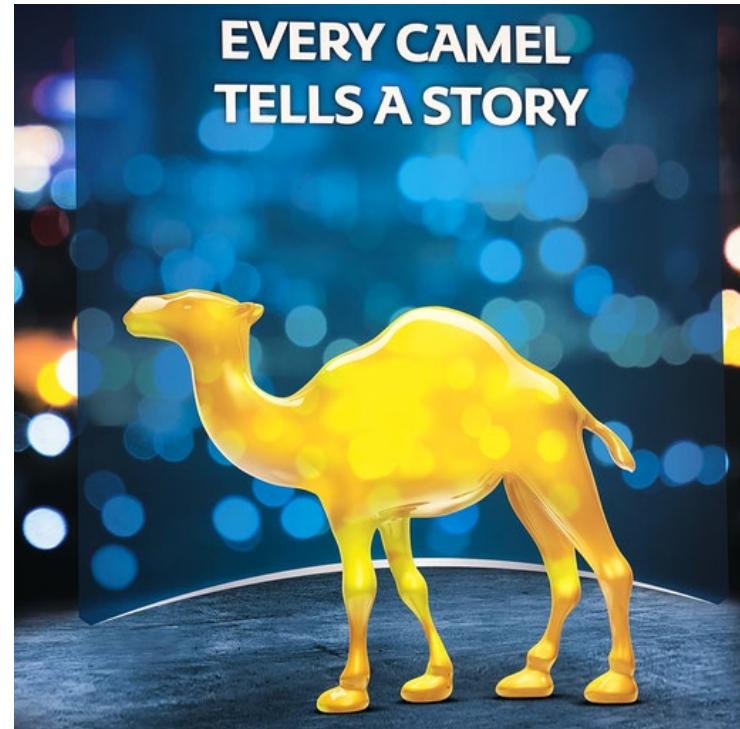


Don't forget TA



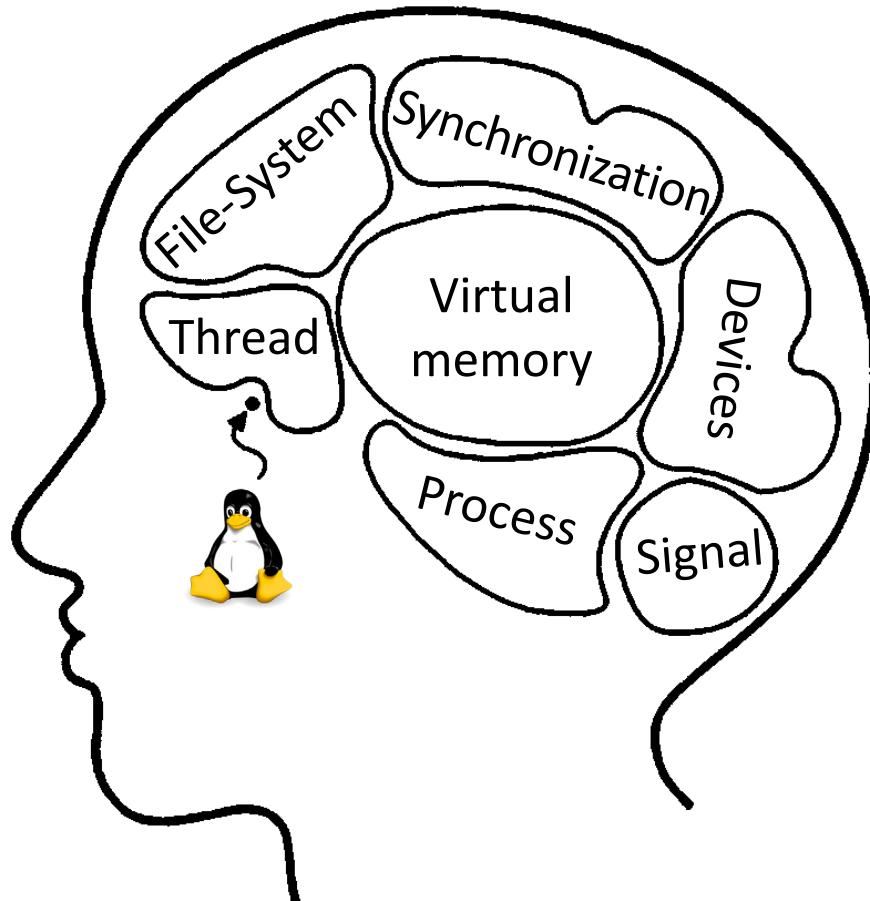
Miryeong Kwon
[\(mkwon@camelab.org\)](mailto:mkwon@camelab.org)

Hanyeoreum Bae
[\(hyr.bae@camelab.org\)](mailto:hyr.bae@camelab.org)



The courses are done w/ a significant help of our TAs!
All feedbacks are welcome, which will make your juniors and future students fruitful

What's Next?



Brain of EE415 students 😊

Don't forget the key concepts of the operating system and try to explore the more advanced topic

