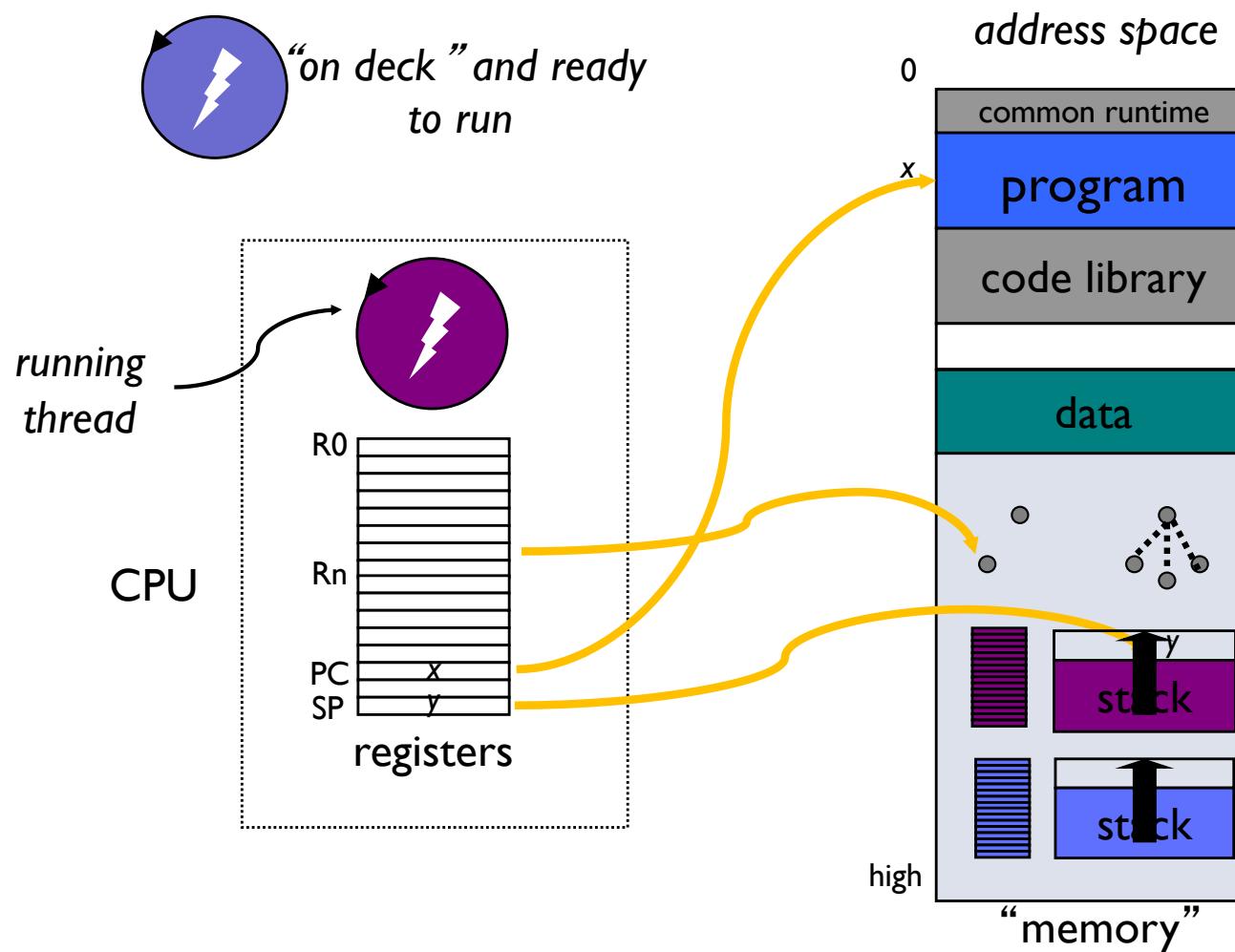


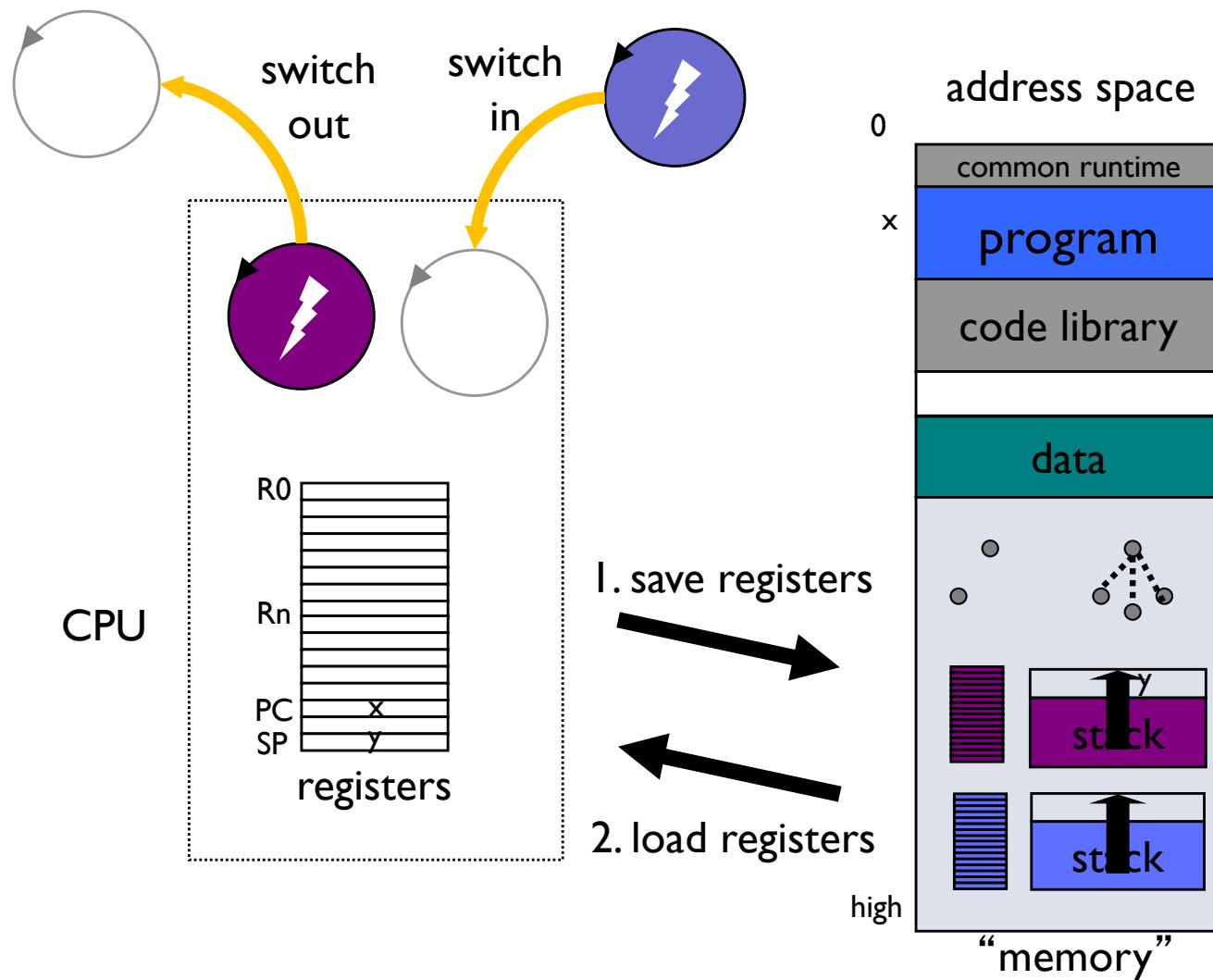
**ECE 570/670**

**David Irwin  
Final Exam Review**

# Program with two threads

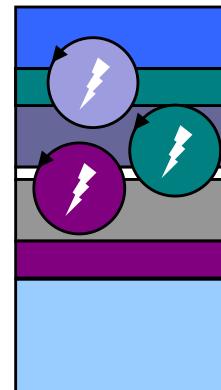
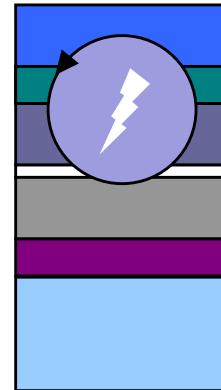


# Thread context switch



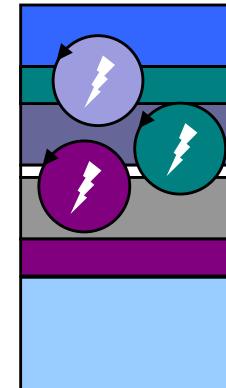
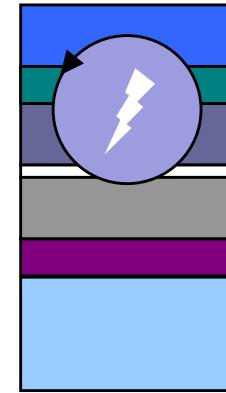
# Threads vs. Processes

- A **process** is an abstraction
  - “Independent executing program”
  - Includes at least one “thread of control”
- Also has a private address space (VAS)
  - Requires OS kernel support
  - To be covered in upcoming lectures



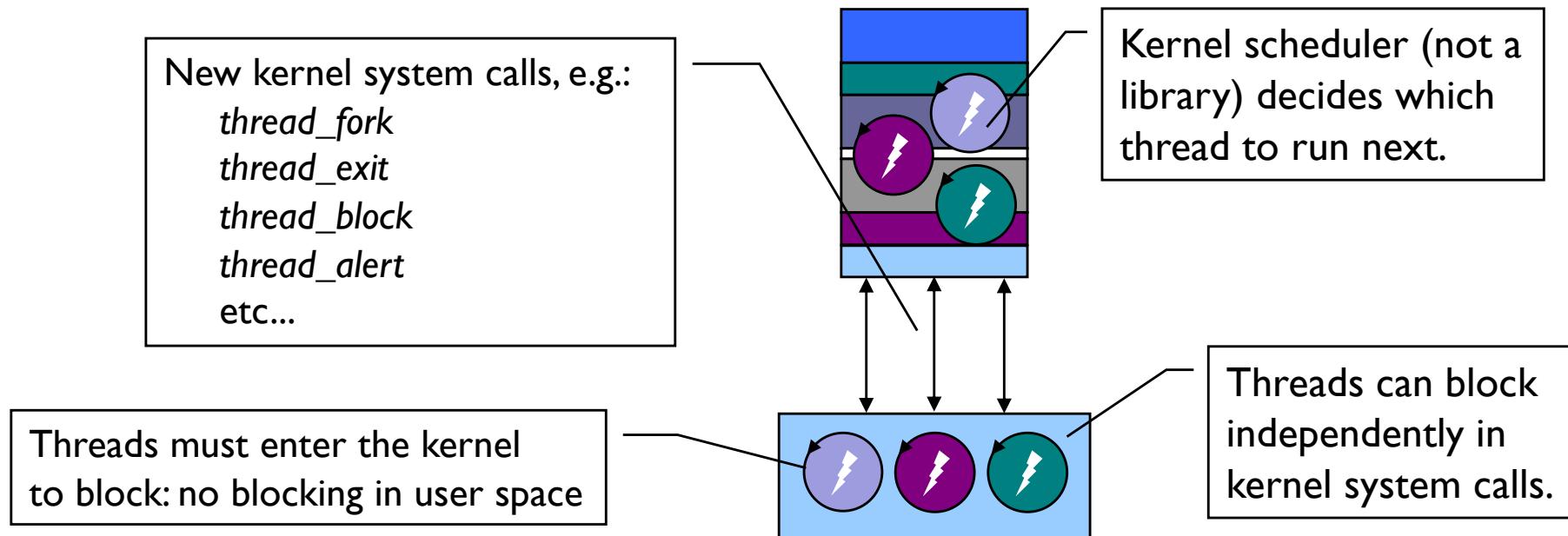
# Threads vs. Processes

- **Threads** may share an address space
  - Have “context” just like vanilla processes
  - Exist within some process VAS
  - Processes may be “multithreaded”
- Key difference
  - Thread context switch vs. process context switch



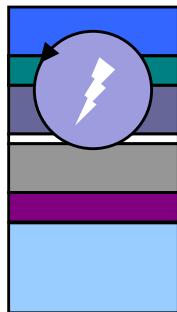
# Kernel-supported threads

- Most OS kernels have *kernel-supported threads*.
  - Thread model and scheduling defined by OS
    - Windows, Unix, etc.
  - Linux: threads are “lightweight processes”

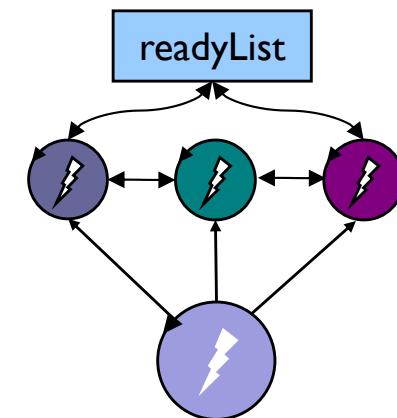


# User-level threads

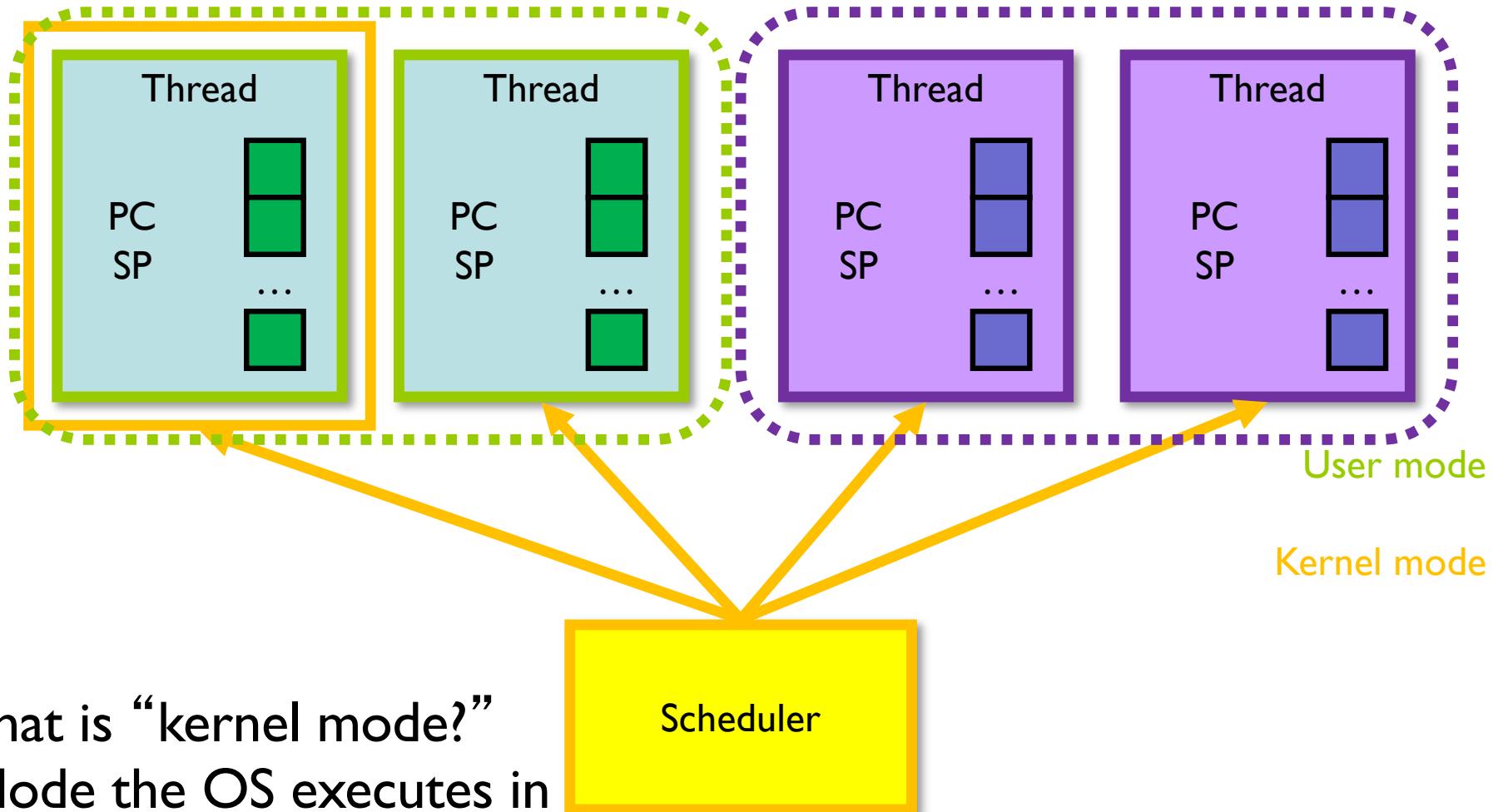
- Can also implement *user-level threads* in a library.
  - No special support needed from the kernel (use any Unix)
  - Thread creation and context switch are fast (no syscall)
  - Defines its own thread model and scheduling policies
  - Kernel only sees a single process
- **Project It**



```
while(1) {  
    t = get next ready thread;  
    scheduler->Run(t);  
}
```

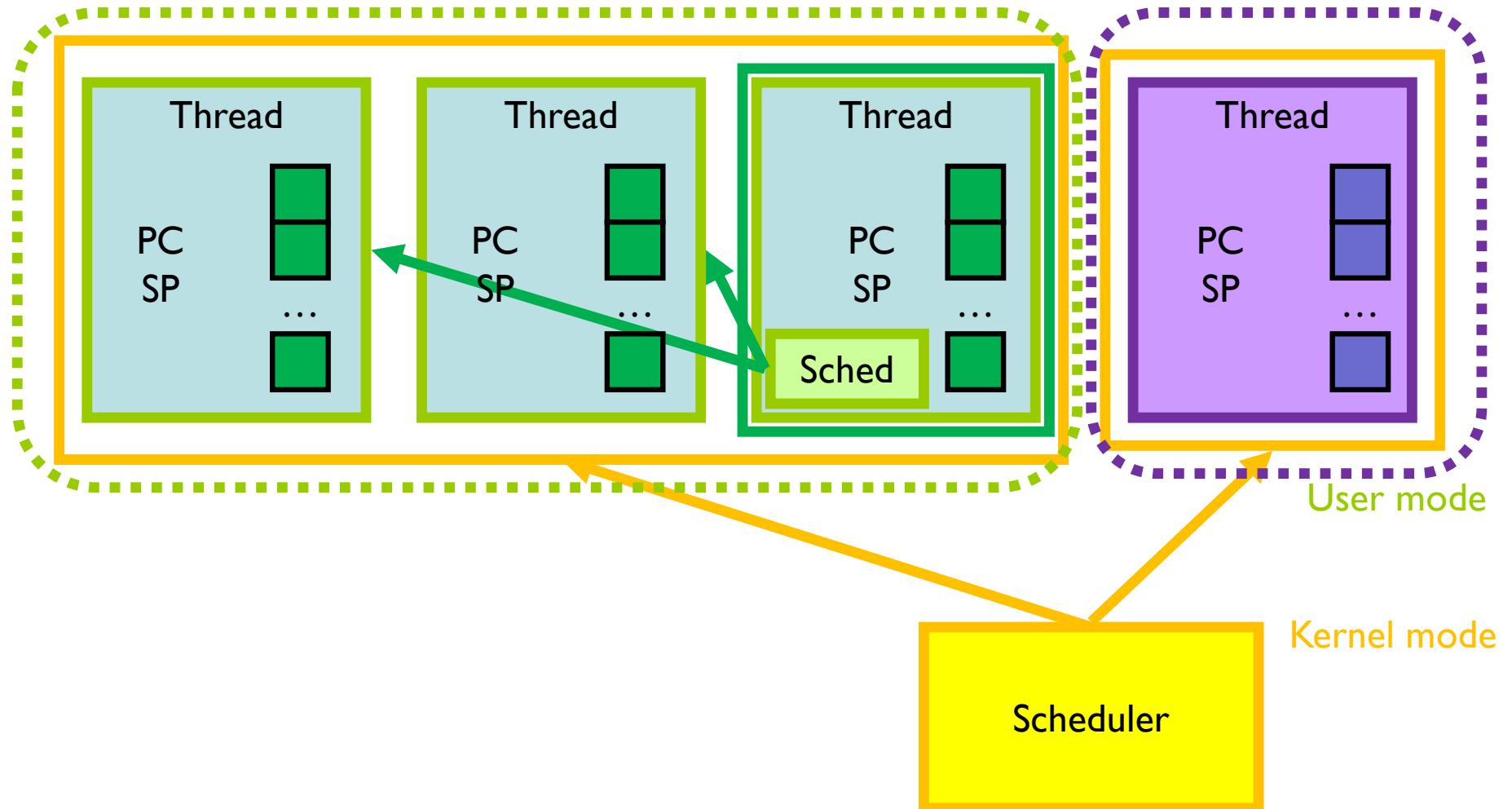


# Kernel threads



What is “kernel mode?”  
Mode the OS executes in  
heightened privileges  
(Will cover in detail soon...)

# User threads



# Kernel vs user tradeoffs

- Which do you expect to perform better?
  - User-threads should perform better
  - Synchronization functions are just a function call
  - Attractive in low-contention scenarios
  - Don't want to kernel trap on every lock acquire
- What is the danger of user-level threads?
  - If a user thread blocks, entire process blocks
  - Hard to take advantage of multiple CPUs
    - Kernel only knows about 1 entity (the process)

# Scheduler Activations

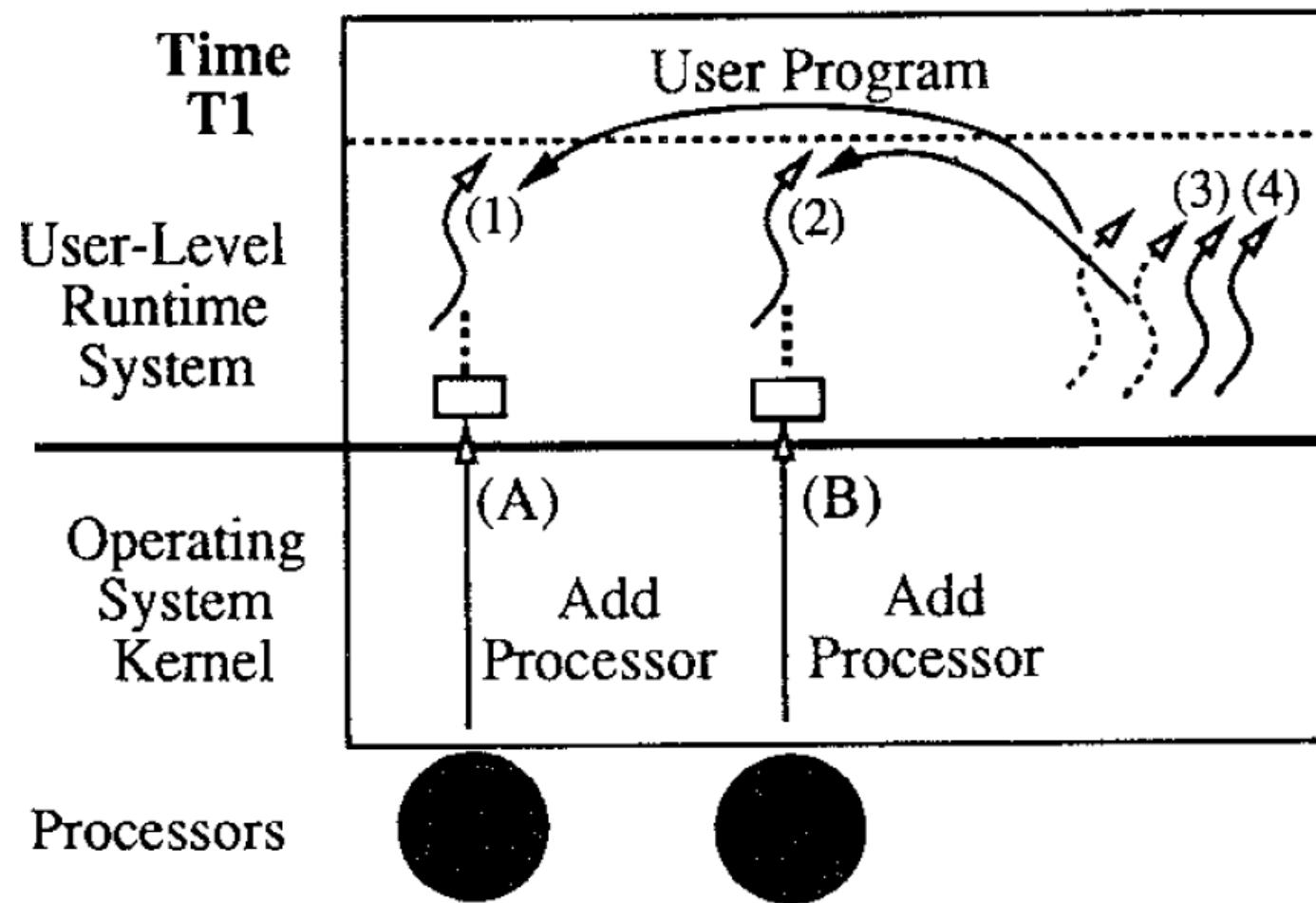
- User-level vs. kernel-level threads
  - Programmer must choose between performance/flexibility (user-level) and scheduling efficiency (kernel-level)
  - User-level = N user threads to 1 kernel thread
  - Kernel-level = 1 user thread to 1 kernel thread
- Scheduler Activation
  - Try to achieve best of both worlds
  - N:M model = N user threads to M kernel threads
  - Notify user thread of kernel changes
    - E.g., when one of M kernel thread gets processor
    - Enable user thread library to react

# Scheduler Activations

Each process supplied with a virtual multiprocessor

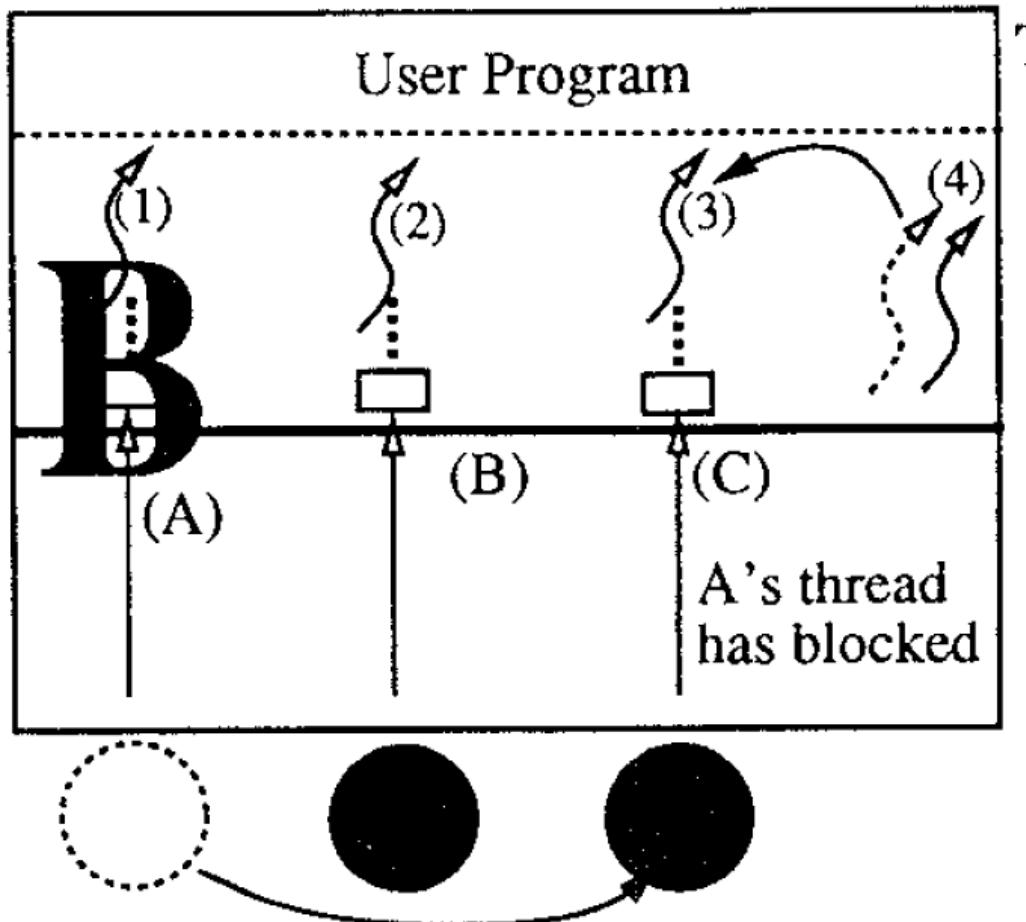
- Kernel allocates processors to address spaces
- User level thread system (**ULTS**) has complete control over scheduling on virtual multiprocessors
- Kernel notifies ULTS whenever it changes the number of actual processors, or a user thread blocks or unblocks
- ULTS notifies kernel when application needs more or fewer processors

# Example (T1)



At time  $T_1$ , the kernel allocates the application two processors. On each processor, the kernel upcalls to user-level code that removes a thread from the ready list and starts running it.

# Example (T2)



Time  
 $T_2$

At time  $T_2$ , one of the user-level threads (thread 1) blocks in the kernel. To notify the user level of this event, the kernel takes the processor that had been running thread 1 and performs an upcall in the context of a fresh scheduler activation. The user-level thread scheduler can then use the processor to take another thread off the ready list and start running it.

# Deadlock avoidance: Banker's algorithm

- Like acquiring all resources first
  - (but more efficient)

```
Phase 1a. state max resources needed  
Phase 1b. while (not done) {  
            get some resources (blocking if not SAFE)  
            work // assume finite  
        }  
Phase 2. release all resources
```

# Comparing banker's algorithm

- Original two-phase locking
  - Acquired resources if available (deadlock)
- Previous solution (no “hold-and-wait”)
  - Acquired all resources or none
- Key insight: Must be able to (eventually) acquire **max** resources to do work
  - All threads doing work can (eventually) get their max
  - Thus, all threads doing work can complete

# Banker's algorithm

- Give out resources at request time (1b)
- Request granted if it is **safe**
  - Can grant max requests of all threads in some sequential order
- **Sequential order**
  - One thread gets its max resources, finishes, and releases its resources
  - Another thread gets its max resources, finishes, and releases, etc

```
Phase 1a. state max resources needed
Phase 1b. while (not done) {
            get some resources (blocking if not SAFE)
            work // assume finite
        }
Phase 2. release all resources
```

# Example

- Bank
  - Has \$6000 to loan
- Customers
  - Establish credit limits (max resources)
  - Borrow money (up to their limit)
  - Return all money at the end

# Example: Solution I

- Bank gives money on request, if available
- For example
  - Ann asks for credit line of \$2000
  - Bob asks for credit line of \$4000
  - Cat asks for credit line of \$6000
- Can bank approve each of these lines?
  - (assuming it gives money, if available)

# Example: Solution I

- No. Consider:
  - Ann takes out \$1000 (bank has \$5000)
  - Bob takes out \$2000 (bank has \$3000)
  - Cat takes out \$3000 (bank is empty)
- Bank has no money
- Ann, Bob, and Cat could all ask for \$
  - None would be able to finish (deadlock)

# Example: Solution I

- This only works if we wait to approve credit lines
- For example
  - Ann asks for credit line of \$2000
    - Bank approves
  - Bob asks for credit line of \$4000
    - Bank approves
  - Cat asks for credit line of \$6000
    - Bank must make Cat wait until Ann or Bob finish before approving
- Sum of all max resource needs of all current threads
  - Must not exceed the total resources

# Example: Solution 2

- Bank says ok to all credit line requests
  - Customers may wait for resources
  - Bank ensures no deadlocks
- For example
  - Ann asks for credit limit of \$2000
  - Bob asks for credit limit of \$4000
  - Cat asks for credit limit of \$6000
  - Bank approves all credit limits

# Example: Solution 2

- Ann takes out \$1000 (bank has \$5000)
- Bob takes out \$2000 (bank has \$3000)
- Cat wants to take out \$2000
  - Is this allowed?
  - Bank would be left with \$1000
  - Ann would still be guaranteed to finish
    - (could take out another \$1000)
    - On finish, bank would have \$2k
    - This is enough to ensure that Bob can finish

## Example: Solution 2

- Ann takes out \$1000 (bank has \$5000)
- Bob takes out \$2000 (bank has \$3000)
- Cat wants to take out \$2500
  - Is this allowed?
  - Bank would be left with \$500
  - Can't guarantee that any threads will finish
  - Can't approve request

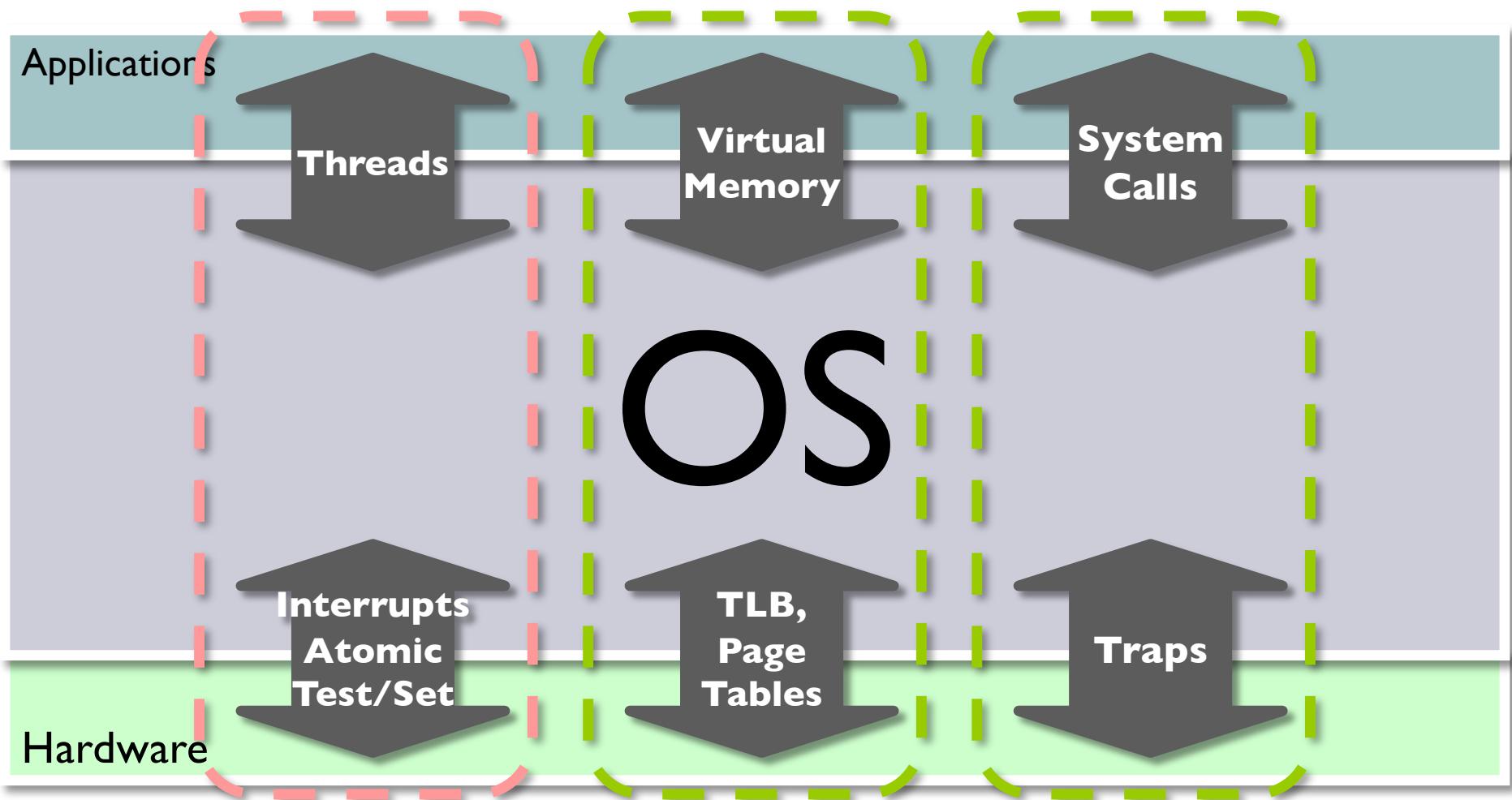
# Banker's algorithm

- Allows resources to be overcommitted
- How can we apply this to dining philosophers?
  - Put all chopsticks in the middle of the table
  - Max resource needs for each philosopher is 2
  - Grant all chopstick requests, unless
    - There is only one chopstick left and nobody has 2
- Nice algorithm, but has some limitations
  - You might not know what you'll need in advance
  - Doesn't really make sense for locks (no "generic lock")

# Threads/concurrency wrap-up

- **Concurrent programs help simplify task decomposition**
  - Relative to asynchronous events
  - Concentrate all the “messiness” in thread library
- **Cooperating threads must synchronize**
  - To protect shared state
  - To control how they interleave
- **We can implement the abstraction of many CPUs on one CPU**
- **Deadlock**
  - Want to make sure that one thread can always make progress
- **CPU scheduling**
  - Example of how policy affects how resources are shared
  - Crucial trade-off: efficiency vs. fairness

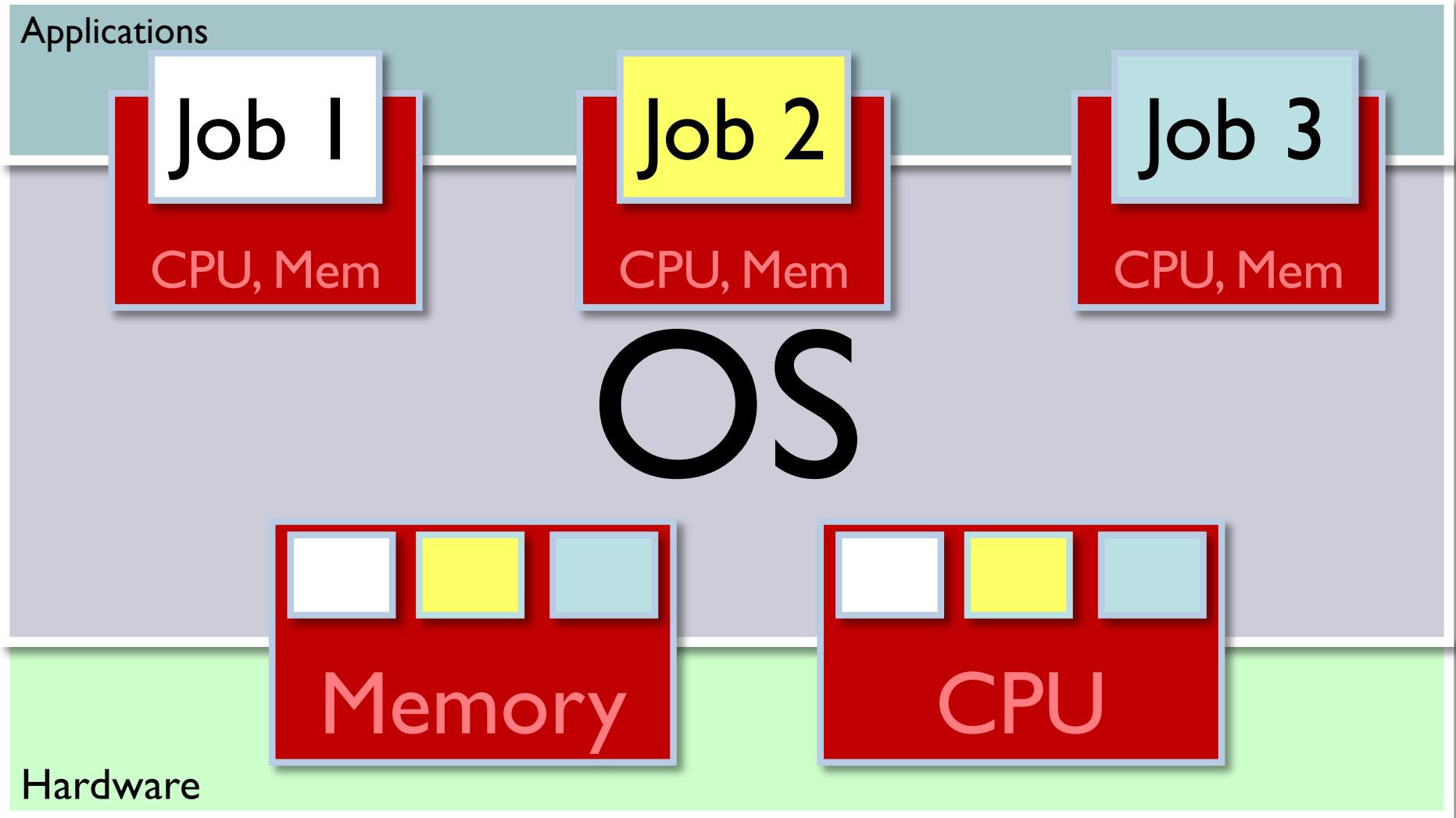
# Operating systems



# Address space abstraction

- Address space
  - All memory data process uses to run
  - Program code, stack, data segment
- **Hardware → software**
  - All processes share single small memory
- **OS → applications**
  - Each process has its own large memory

# Hardware, OS interfaces



# Illusions of the address space

## I. Address independence

- Can use same numeric address in 2 processes
- Each process has its own version of “address 4”
- Each “address 4” refers to different data items

## 2. Protection

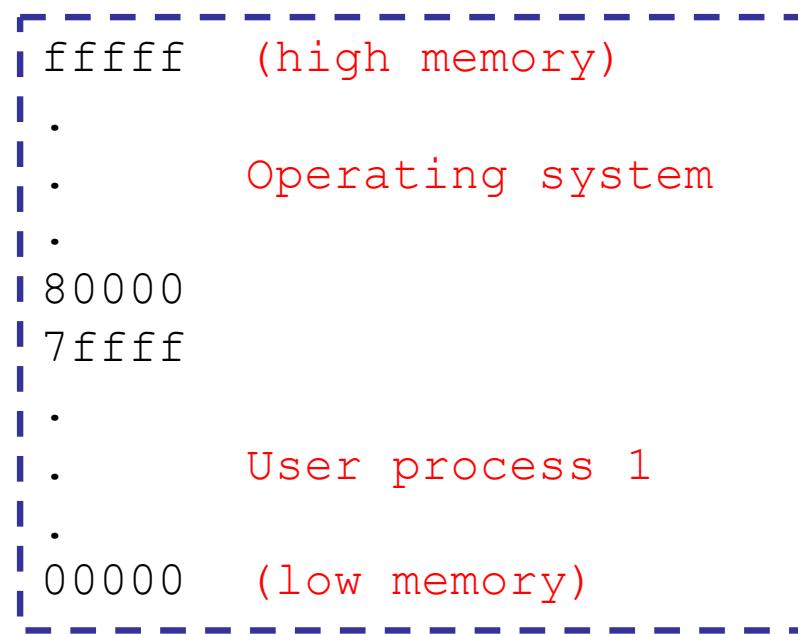
- One process cannot touch another’s data

## 3. Virtual memory

- Address space can be larger than physical memory

# Uni-programming

- One process occupies memory at a time
- Always load process into same spot
- Must reserve space for the OS



# Uni-programming

- Virtual address
  - Address programmer thinks he/she's accessing
- Physical address
  - Address from the view of the machine
  - (actual physical memory location)
- What is the V-to-P mapping in uni-programming?
  - Identity map (virtual A → physical A)

# Uni-programming

- How to run another process?
  - Swap user process 1 to disk
  - Bring user process 2 to memory
  - Kernel scheduler swaps address spaces
  - (when doing a context switch)
- This is how early PCs worked (badly)

# Uni-programming

- What features does this provide?
  - Address independence
  - Protection
  - No virtual memory
- Note sum of address spaces > phys mem.

# Uni-programming

- Problems with uni-programming?
  - Swapping to/from disk is sloooow
  - Moves more data than might be needed
- What does this imply about RR slice?
  - High overhead for swapping → large slice
  - Large slice → interactivity suffers

# Multi-programming

- More than one process in phys memory
  - Processes can have same virtual addrs
  - Cannot share physical addrs
- **So addresses must be translated!**
  - Statically: occurs before execution
  - Dynamically: occurs during execution
- Protection is harder (than uni-programming)

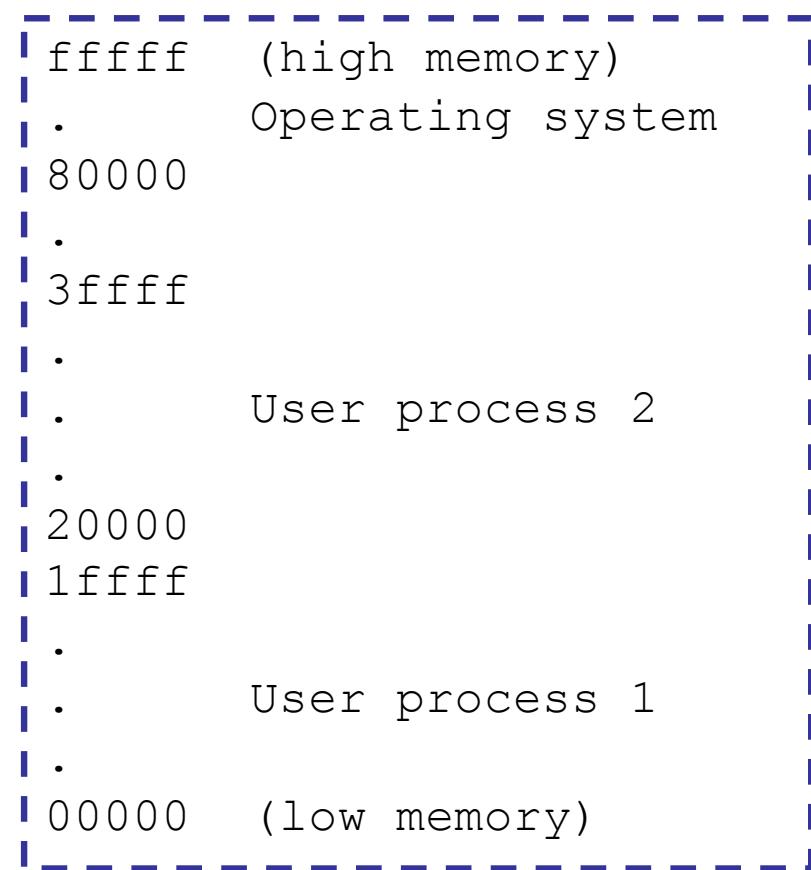
# Static address translation

- Want two processes in memory
  - With address independence
  - Without translating on-the-fly
  - Must use static translation
- Could you do this?

# Static address translation

- Adjust loads/stores when put in memory
  - This is called a linker-loader
- Programming
  - Assume memory starts at 0
- Linker-loader
  - Adds 0x20000 to offsets for P2
- Similar to linking phase of compile

```
load $1, $base + offset
```



# Linking your programs

- Compiler generates .o files + libraries
  - .o files all assume they start at address 0
- Linker puts everything together
  - Resolves cross-module references
    - (e.g. “how do I jump to `thread_create?`”)
  - Spits out single executable file

# Multi-programming abstractions

- Address independence?
  - Yes. (my address 4 is different from yours)
- Protection?
  - No.
  - Why not?

# Multi-programming protection

```
load $1, $base + offset
```

- Static translation changes the offset
- Process could still issue odd register value
- Problem
  - Buggy/malicious code can corrupt other processes
  - User (not system) gets last move

# Multi-programming features

- Address independence?
  - Yes. (my address 4 is different from yours)
- Protection?
  - No.
- Virtual memory?
  - No.

# Multi-programming virtual memory

- Virtual address space  $\leq$  phys mem
- Proof: static translation cannot provide VM
  - Each VA maps to a fixed PA
  - (one-to-one mapping)
  - Thus, # of VAs must equal # of PAs
  - (but we want more VAs than PAs)
- Our goals require dynamic translation

# Dynamic address translation



Will this allow us to provide protection?

Sure, as long as the translation is correct

# Dynamic address translation



This is an example of a systems service called “naming.”  
Can you think of other examples, that you use everyday?

Internet DNS (Domain Name Service)

File System (human-readable names to blocks on disk)

# Dynamic address translation

- Does this enable virtual memory?
  - Yes
- VA only needs to be in phys mem *when accessed*
- Provides flexibility
  - Translations can change on-the-fly
  - Different VAs can occupy different PAs
- Common technique
  - Add level of indirection to gain flexibility

# Hardware support

- Traditional view
  - Dynamic translation needs hardware support
- Basic requirement
  - Must translate each load/store
  - Incurs extremely high overhead!

# Dynamic address translation



- Translator: just a data structure
- Tradeoffs
  - Flexibility (sharing, growth, virtual memory)
  - Size of translation data
  - Speed of translation

# I. Base and bounds

- For each process
  - Single contiguous region of phys mem
  - Not allowed to access outside of region
  - Illusion of own physical mem [0, bound)

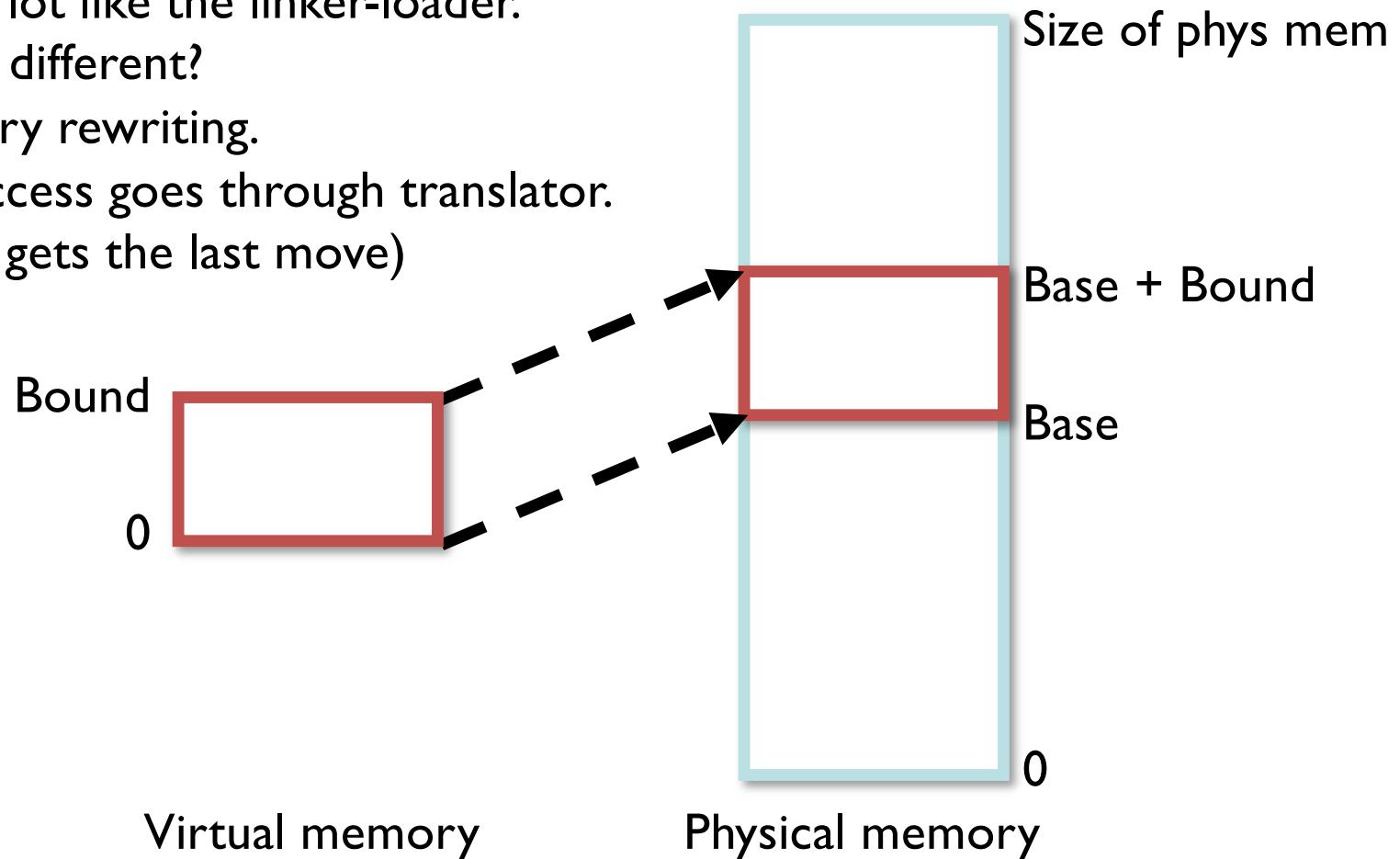
# I. Base and bounds

Looks a lot like the linker-loader.

What is different?

No binary rewriting.

Every access goes through translator.  
(system gets the last move)



# I. Base and bounds

- **Translator algorithm**

```
if (virtual address > bound) {  
    trap to kernel  
    kernel can kill process with segmentation fault  
} else {  
    physical address = virtual address + base  
}
```

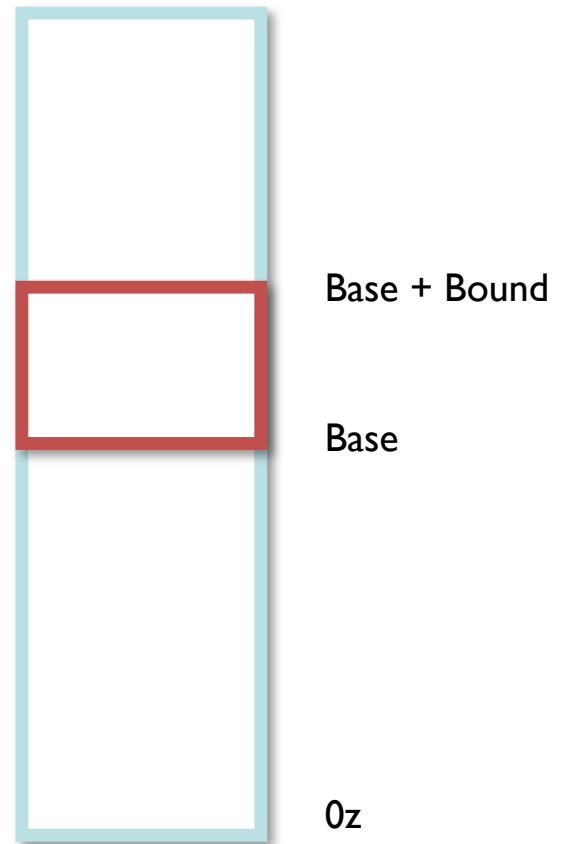
- **Only kernel can change base, bound**

# I. Base and bounds

- What happens on a context switch?
  - Must translate addresses differently
  - Load translation data for new process
  - Set base and bounds registers

# I. Base and bounds

- How can address spaces grow?
  - If memory above base+bound is free
    - Increase the bound
  - If memory above base+bound isn't free
    - Stop program
    - Copy data
    - Change base and bounds
    - Restart process
- Does being moved affect the program?
  - No, it still only knows about virtual addresses

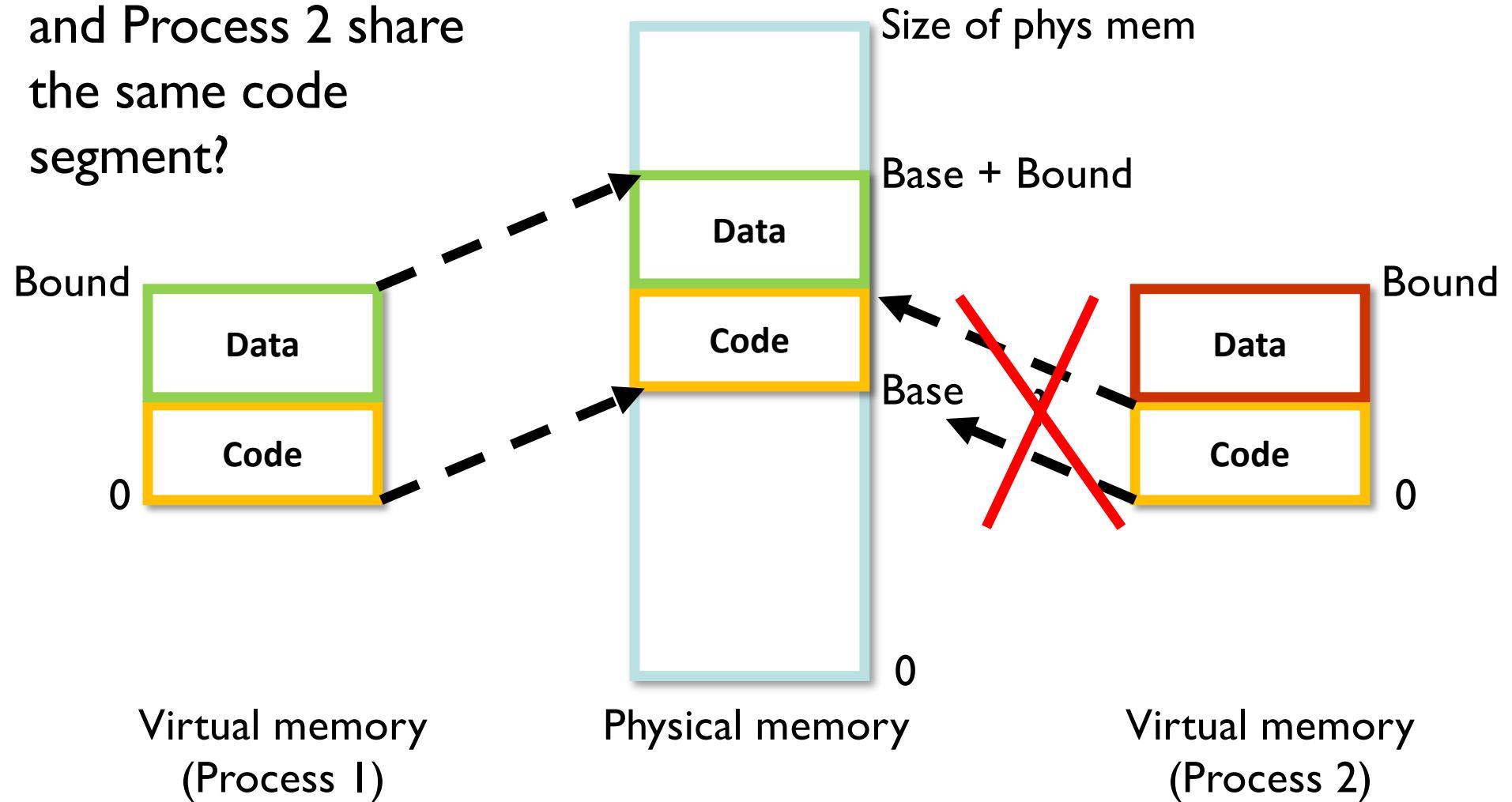


# Base and bounds pros and cons

- Pros
  - Low hardware costs (2 registers, adder, comparator)
  - Fast (only inserting addition and comparison)
- Cons
  - Single address space can't easily exceed size of phys mem
  - Growing virtual address space might be slow
  - **Sharing is difficult (must share all or nothing)**

# Base and bounds sharing

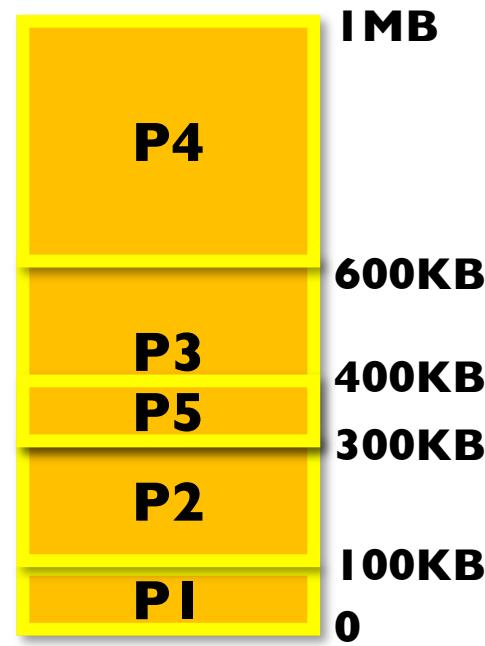
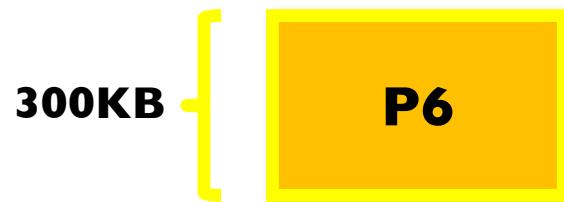
Why can't Process 1  
and Process 2 share  
the same code  
segment?



# Base and bounds pros and cons

- Pros
  - Low hardware costs (2 registers, adder, comparator)
  - Fast (only inserting addition and comparison)
- Cons
  - Single address space can't easily exceed size of phys mem
  - Growing virtual address space might be slow
  - Sharing is difficult (must share all or nothing)
  - **Waste memory due to external fragmentation**

# Base and bounds fragmentation



## Minimizing external fragmentation

Possible heuristics:

- Best fit  
(Allocate in smallest free region)
- First fit  
(Use first region that fits)

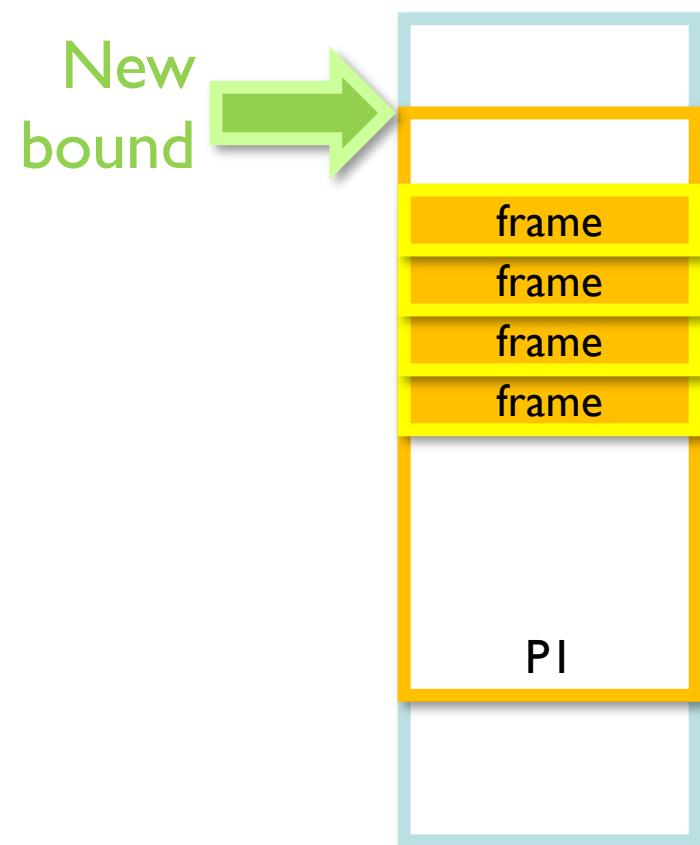
**Physical  
memory**

# Base and bounds pros and cons

- Pros
  - Low hardware costs (2 registers, adder, comparator)
  - Fast (only inserting addition and comparison)
- Cons
  - Single address space can't easily exceed size of phys mem
  - Sharing is difficult (must share all or nothing)
  - Growing virtual address space might be slow
  - Waste memory due to external fragmentation
  - **Hard to grow multiple regions of memory**

# Base and bounds growth

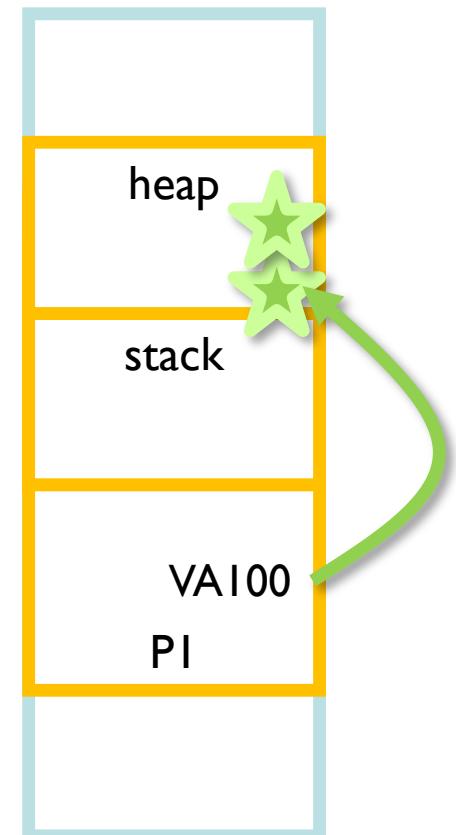
- What grows in a process?
  - The stack and heap



# Base and bounds growth

- What grows in a process?
  - The stack and heap
- How do you do this with both?
  - If heap grows, extend PI's bound
  - If stack grows, must shift heap up
    - What about pointers into heap?

New data on heap at VA 100  
(translated to PA 100+base)  
Store this address in a variable  
How can you shift VAs up?



Problem: must fit two growing data structures in one contiguous region

## 2. Segmentation

- **Segment**
  - Region of contiguous memory
  - (both virtually and physically)
- Idea
  - Generalize base and bounds
  - Create a table of base and bound pairs

## 2. Segmentation

Segment #	Base	Bound	
Segment 0	4000	700	Code
Segment 1	0	500	Data segment
Segment 2	Unused	Unused	
Segment 3	2000	1000	Stack segment

**Virtual info**      **Physical info**

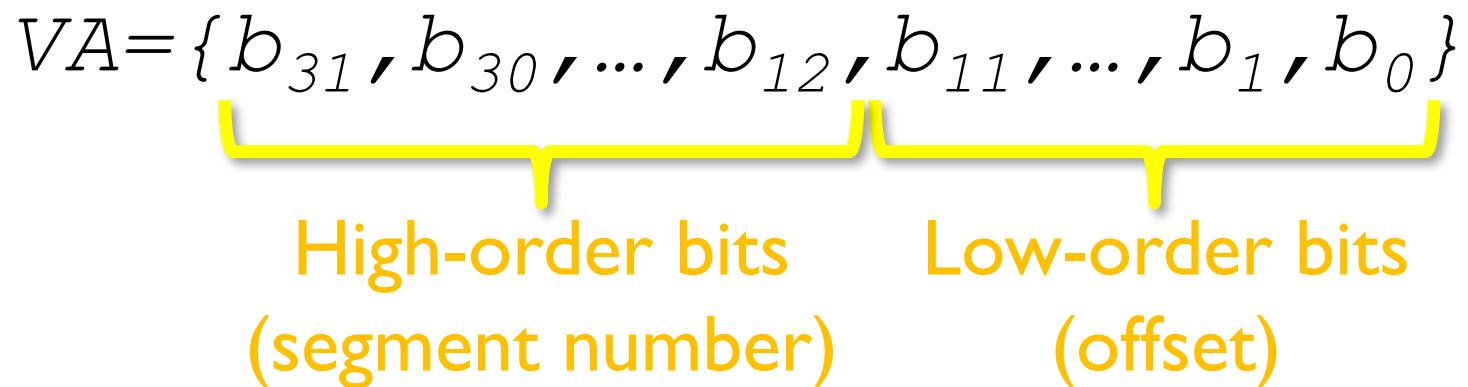
- Virtual address has two parts

1. Segment # (could be in high-order bits)

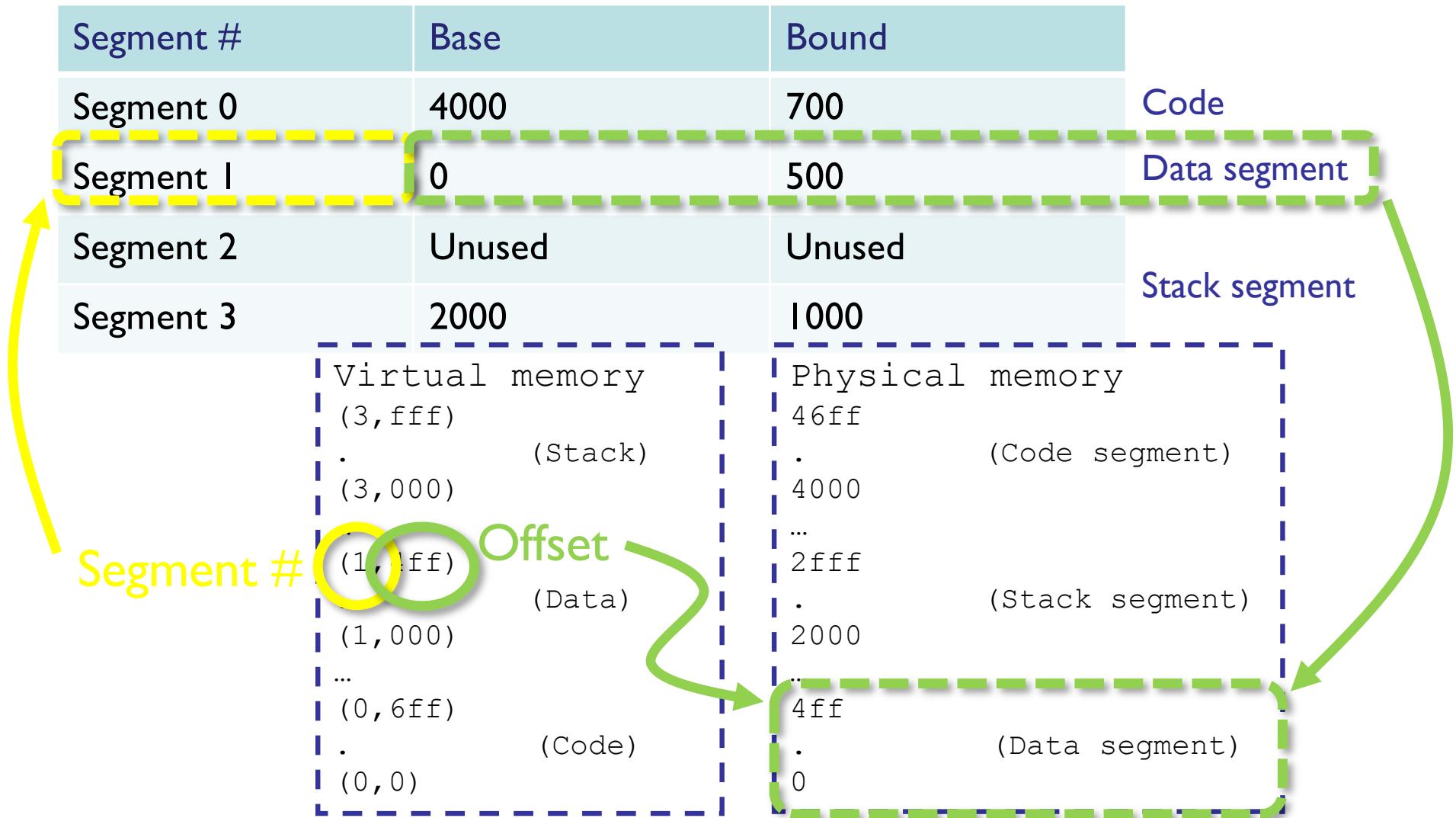
2. Offset (e.g. low-order bits of address)

Same for both

# Virtual addresses



## 2. Segmentation



## 2. Segmentation

Segment #	Base	Bound	
Segment 0	4000	700	Code
Segment 1	0	500	Data segment
Segment 2	Unused	Unused	Stack segment
Segment 3	2000	1000	

- Not all virtual addresses are valid
  - Nothing in segment 2
  - Nothing in segment 1 above 4ff
- Valid = part of process' s address space
- Accesses to invalid addresses are illegal
  - Hence a “segmentation fault”

The diagram illustrates the mapping of virtual memory into physical memory. A dashed blue rectangle represents the 'Virtual memory' space, spanning from address 3 (top) to 0 (bottom). Inside this space, four segments are defined by vertical dashed lines:

- Stack:** Addresses 3,000 to 4ff. Labeled '(Stack)'.
- Data:** Addresses 1,000 to 6ff. Labeled '(Data)'.
- Code:** Addresses 0 to 1000. Labeled '(Code)'.
- Unused:** Addresses 0 to 500. Labeled '(Unused)'.

Each segment's base address is aligned to the next power of 2 (4000 for code, 2000 for stack, etc.). The segments overlap, with the stack at the top and the code at the bottom.

## 2. Segmentation

Segment #	Base	Bound	
Segment 0	4000	700	Code
Segment 1	0	500	Data segment
Segment 2	Unused	Unused	Stack segment
Segment 3	2000	1000	

- Segments can grow
  - (can move to new physical location)
- Protection
  - Different protections for segments
    - E.g. read-only or read-write
  - B&B forced uniform protection

Virtual memory  
(3, fff) (Stack)  
.  
(3, 000)  
...  
(1, 4ff)  
.  
(1, 000)  
...  
(0, 6ff)  
. (Data)  
(0, 0) (Code)

# 2. Segmentation

Segment #	Base	Bound	
Segment 0	4000	700	Code
Segment 1	0	500	Data segment
Segment 2	Unused	Unused	Stack segment
Segment 3	2000	1000	

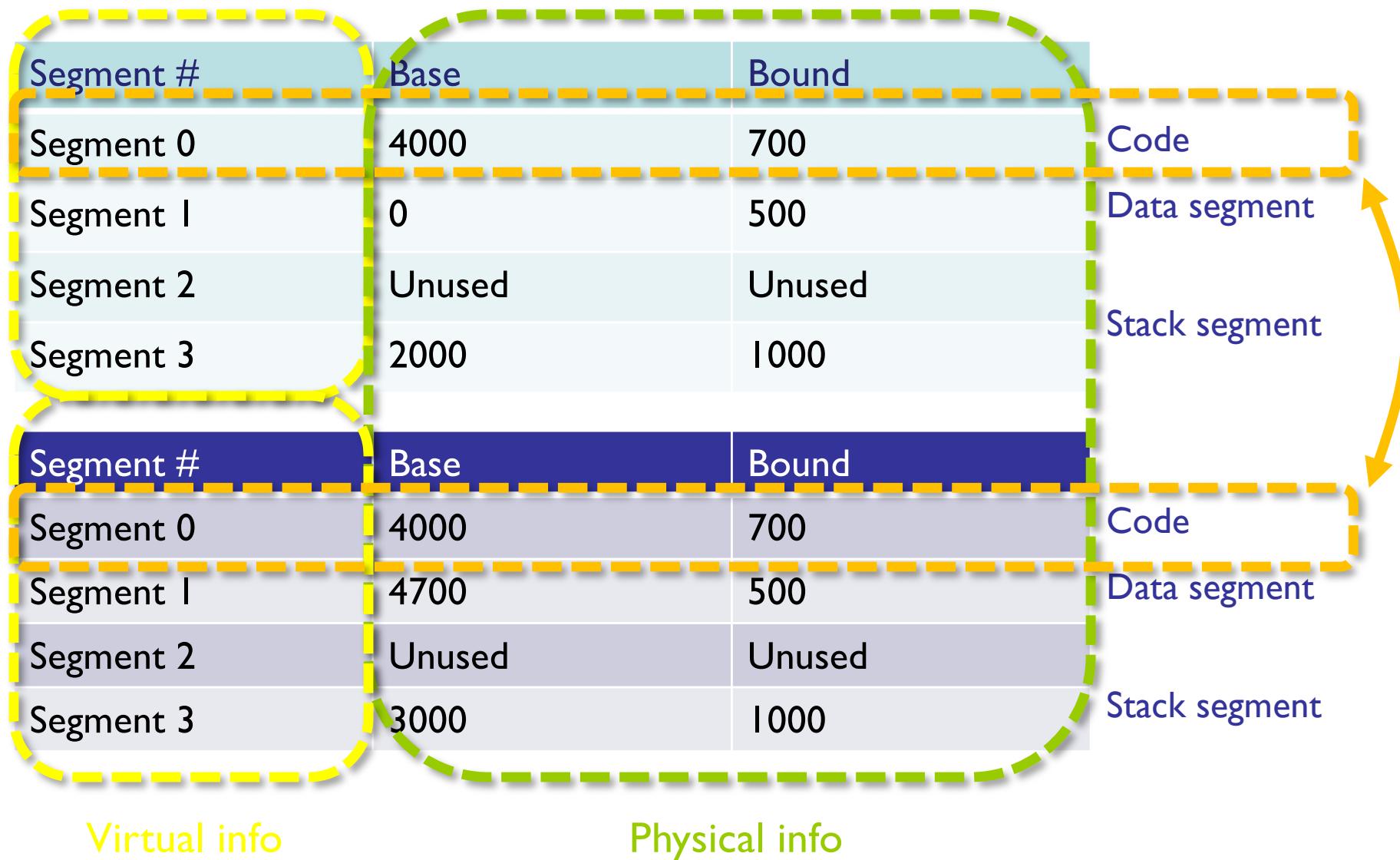
- What changes on a context switch?
  - Contents of segment table
  - Typically small (not many segments)

Virtual memory  
(3, ffff) (Stack)  
.  
(3, 000)  
...  
(1, 4ff)  
.  
(1, 000)  
...  
(0, 6ff)  
. (Code)  
(0, 0)

# Segmentation pros and cons

- Pros
  - Multiple areas of address space can grow separately
  - Easy to share parts of address space
  - (can share code segment)

# Segmentation sharing



# Segmentation pros and cons

- Pros
  - Multiple areas of address space can grow separately
  - Easy to share parts of address space
  - (can share code segment)
- Cons
  - Complex memory allocation
  - (still have external fragmentation)

## 2. Segmentation

- Do we get virtual memory?
- (can an address space be larger than phys mem?)
  - Segments must be smaller than physical memory
  - Address space can contain multiple segments
  - Can swap segments in and out
- What makes this tricky?
  - Performance (segments are relatively large)
  - Complexity (odd segment sizes → packing problem)
- Solution: fixed-size segments called **pages!**

# 3. Paging

- Very similar to segmentation
- Allocate memory in fixed-size chunks
  - Chunks are called **pages**
- Why fixed-size pages?
  - Allocation is greatly simplified
  - Just keep a list of free physical pages
  - Any physical page can store any virtual page

# 3. Paging

- Virtual addresses
  - Virtual page # (e.g. upper 20 bits)
  - Offset within the page (e.g. low 12 bits)
  - Each address refers to a byte of data
- For 12-bit offset, how big are pages?
  - 4KB ( $2^{12}$  different offsets)

# 3. Paging

- Translation data is a **page table**

Why no bound column?



Virtual page #	Physical page #
0	10
1	15
2	20
3	invalid
...	...
1048574	invalid
1048575	invalid

$2^{20} - 1$   
Why?



# 3. Paging

- Translation data is a **page table**

How does  
page table's  
size compare  
to segment  
table's?

Virtual page #	Physical page #
0	10
1	15
2	20
3	invalid
...	...
1048574	Invalid
1048575	invalid

Page table must reside in memory. Too large for registers alone.

# 3. Paging

- Translation process

```
if (virtual page # is invalid) {
    trap to kernel
} else {
    physical page = pagetable[virtual page].physPageNum
    physical address is {physical page}{offset}
}
```

# 3. Paging

- What happens on a context switch?
  - Copy out contents of entire page table
  - Copy new page table into translation box
    - (hideously slow)
- Instead, use indirection
  - Change page table base pointer (register)
  - Should point to new page table
- Does this look like anything else we've seen?
  - Stack pointer

# 3. Paging

- Pages can be in memory or swapped to disk
  - “Swapped out” = “paged out”
- How can the processor know?
  - Is it in physical memory or paged out to disk?
- Page table entry must have more info
  - **Resident bit**

# 3. Paging

- Resident bit in page table entry
- **If bit is 1 (vp is in physical memory)**
  - Page table entry has valid translation
  - Use to look up physical page
- **If bit is 0 (vp is swapped out)**
  - MMU causes a page fault to OS
  - (runs kernel's page fault handler)
  - (just like how timer interrupts are handled)

# 3. Paging

- On a page fault, what does the OS do?
  - OS finds free physical page
  - (may have to page out another virtual page)
  - OS does I/O to read virtual page into mem from disk
  - (or zero fills it, if it's a new page)
- What happens next?
  - OS restarts process at last faulting instruction
  - Essentially, replay the instruction
- How does it know the last instruction?
  - Hardware provides process' PC to handler on fault

# Valid vs. resident pages

- **Resident = page is in memory**
  - Accessing non-resident pages is not an error
  - Access triggers page fault, OS handles it
- **Valid = page is legal to address**
- Who makes a virtual page resident (or not)?
  - OS memory manager (allocating physical memory)

# Valid vs. resident pages

- **Resident = page is in memory**
  - Accessing non-resident pages is not an error
  - Access triggers page fault, OS handles it
- **Valid = page is legal to address**
- Who makes a virtual page valid (or not)?
  - User process controls this (with limits from OS)

# Valid vs. resident pages

- **Resident = page is in memory**
  - Accessing non-resident pages is not an error
  - Access triggers page fault, OS handles it
- **Valid = page is legal to address**
- Why would a process make a page invalid?
  - Make the process fail-stop
  - Accidental/buggy references kill the process
  - Same for process making pages read-only
  - (writes to code segment are probably bugs)

# Page size trade-offs

- If page size is too small
  - Lots of page table entries
  - Big page table
- If we use 4-byte pages with 4-byte PTEs
  - 4 byte pages use 2 bit offset, which leave 30 bits for virtual addresses
  - Num PTEs =  $2^{30}$  = 1 billion
  - 1 billion PTE \* 4 bytes/PTE = 4 GB
  - Would take up entire address space! For only 1 process!

# Page size trade-offs

- What if we use really big (1 GB) pages?
  - **Internal fragmentation**
  - Wasted space within the page
  - Recall external fragmentation
  - (wasted space between pages/segments)

# Page size trade-offs

- Compromise between the two
  - x86 page size = 4KB for 32-bit processor
- For 4KB pages, how big is the page table?
  - 1 million page table entries
  - 32-bits (4 bytes) per page table entry

→ 4 MB per page table

Still 4MB per process

If there are 1000 processes, could take up 4GB

207 processes were running on the autograder a few minutes ago

# Paging pros and cons

- + Simple memory allocation
- + Can share lots of small pieces
  - (how would two processes share a page?)
  - (how would two processes share an address space?)
- + Easy to grow address space
- Large page tables (even with large invalid section)

# Comparing translation schemes

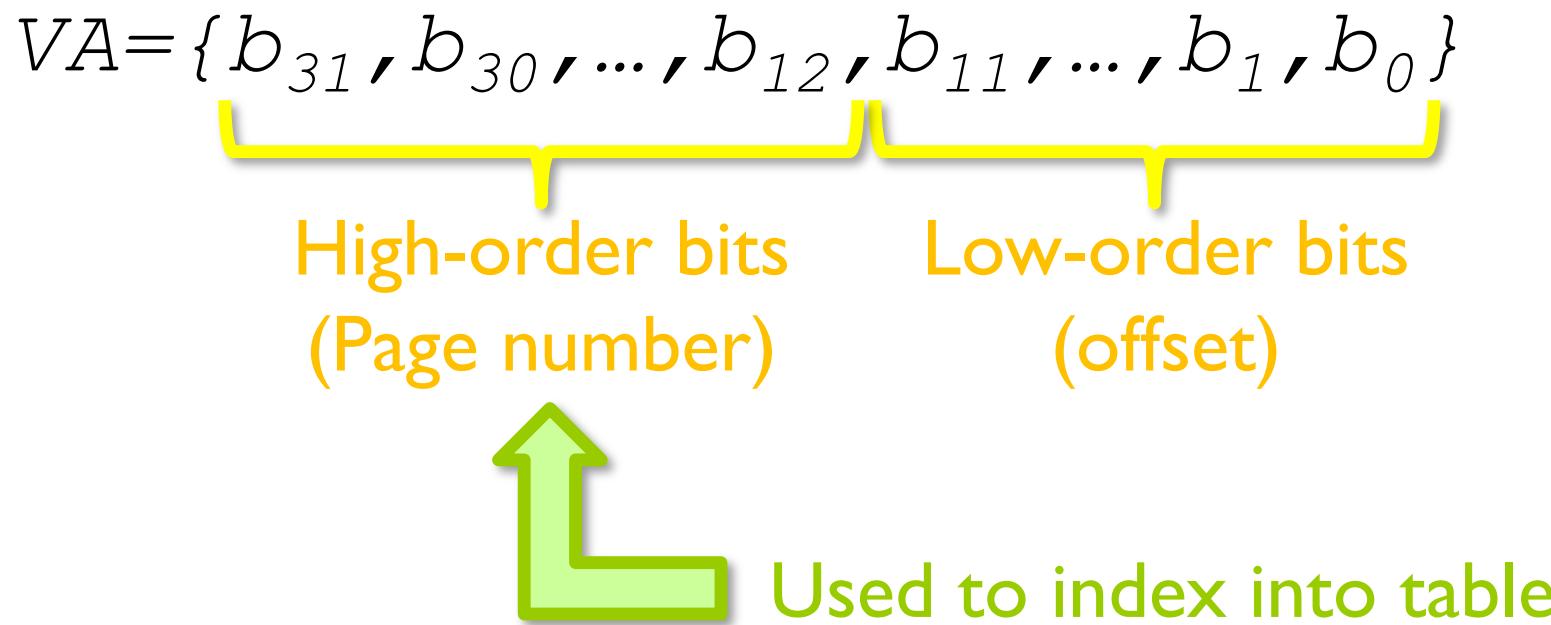
- Base and bounds
  - Unit of translation = entire address space
- Segmentation
  - Unit of translation = segment
  - A few large variable-size segments/address space
- Paging
  - Unit of translation = page
  - Lots of small fixed-size pages/address space

# What we want

- Efficient use of physical memory
  - Little external or internal fragmentation
  - Easy sharing between processes
  - **Space-efficient data structure**
    - Must handle sparse, widely distributed VAs
      - Text segment, stack, heap are widely separated across address space
    - Don't want to require large, physically-contiguous arrays
- How we'll do this
  - Modify paging scheme

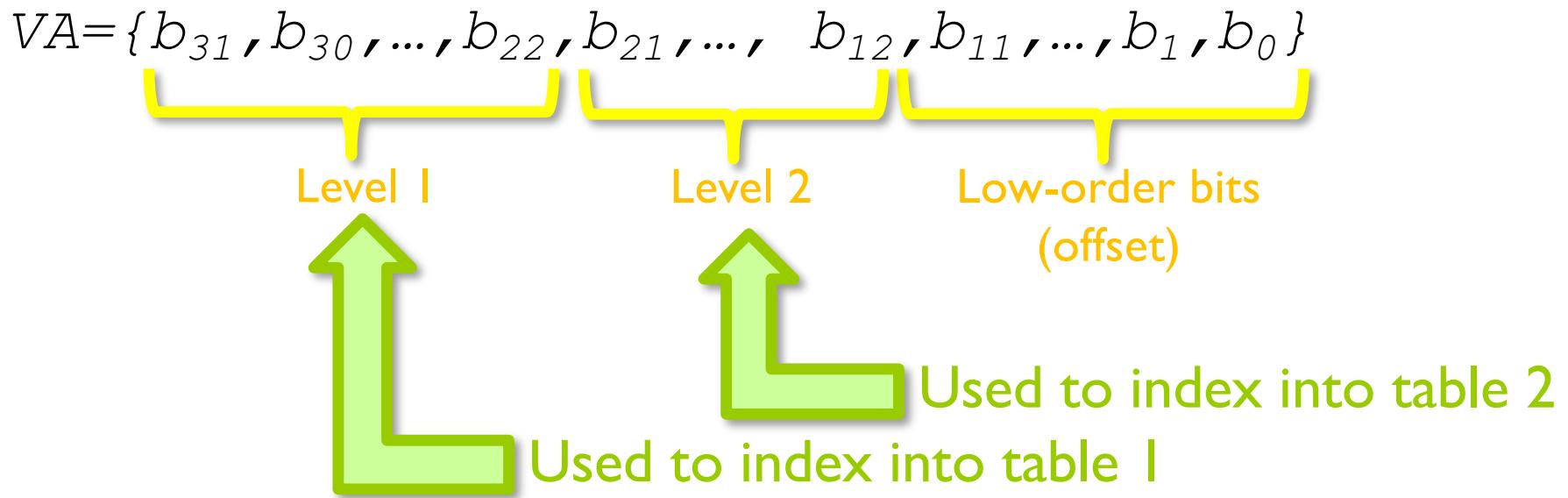
## 4. Multi-level translation

- Standard page table
  - Just a flat array of page table entries



# 4. Multi-level translation

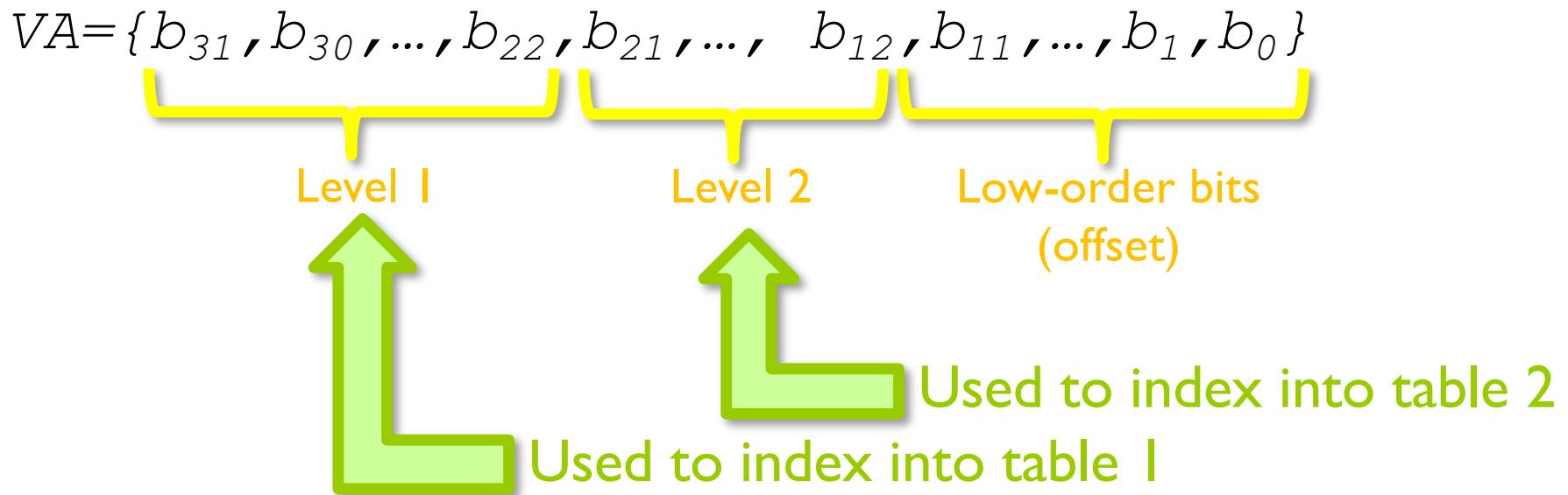
- Multi-level page table
  - Use a tree instead of an array
  - Linux actually uses a 4-level tree



What is stored in the level 1 page table? If valid? If invalid?

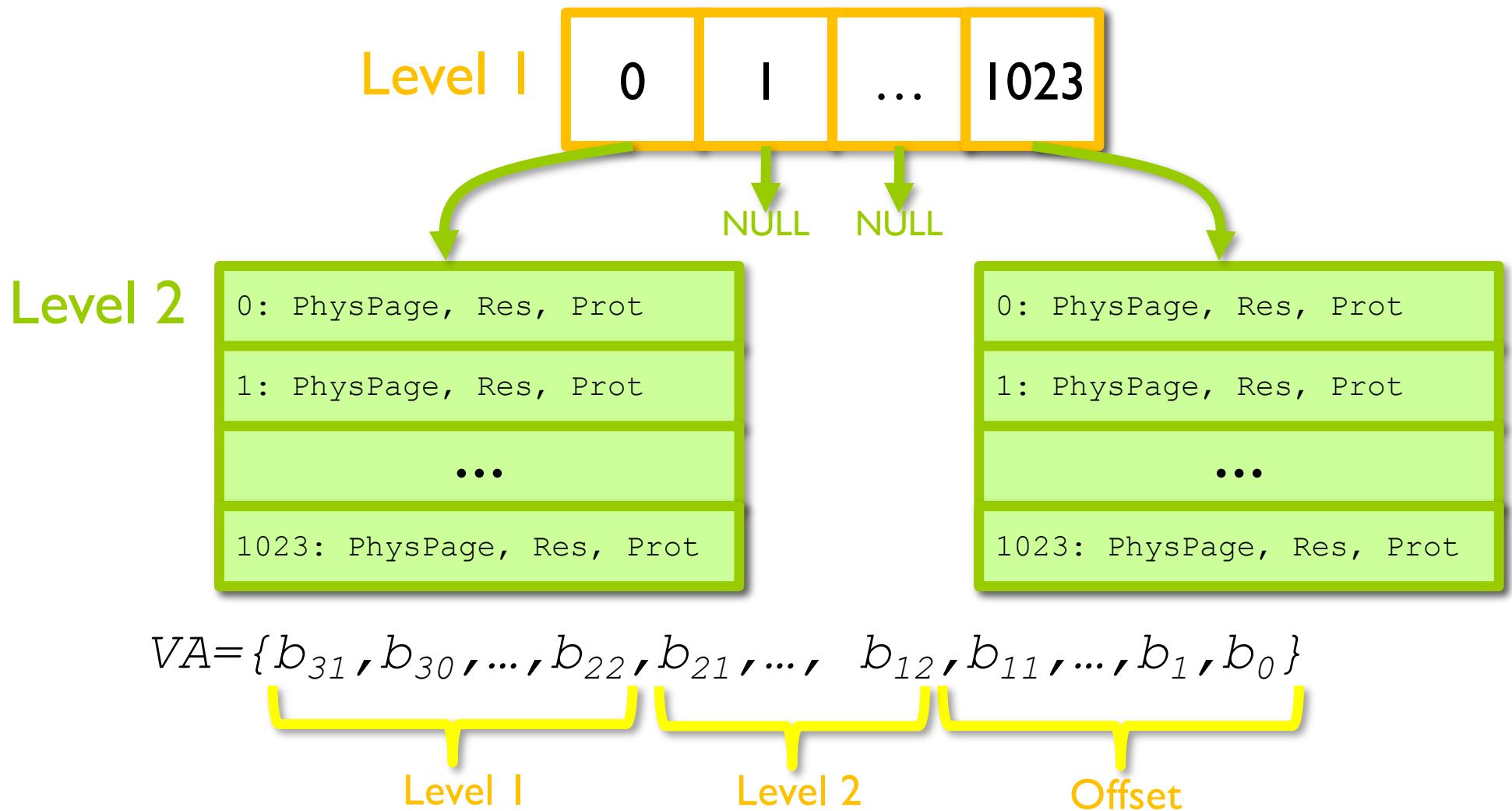
# 4. Multi-level translation

- Multi-level page table
  - Use a tree instead of an array
  - Linux actually uses a 4-level tree

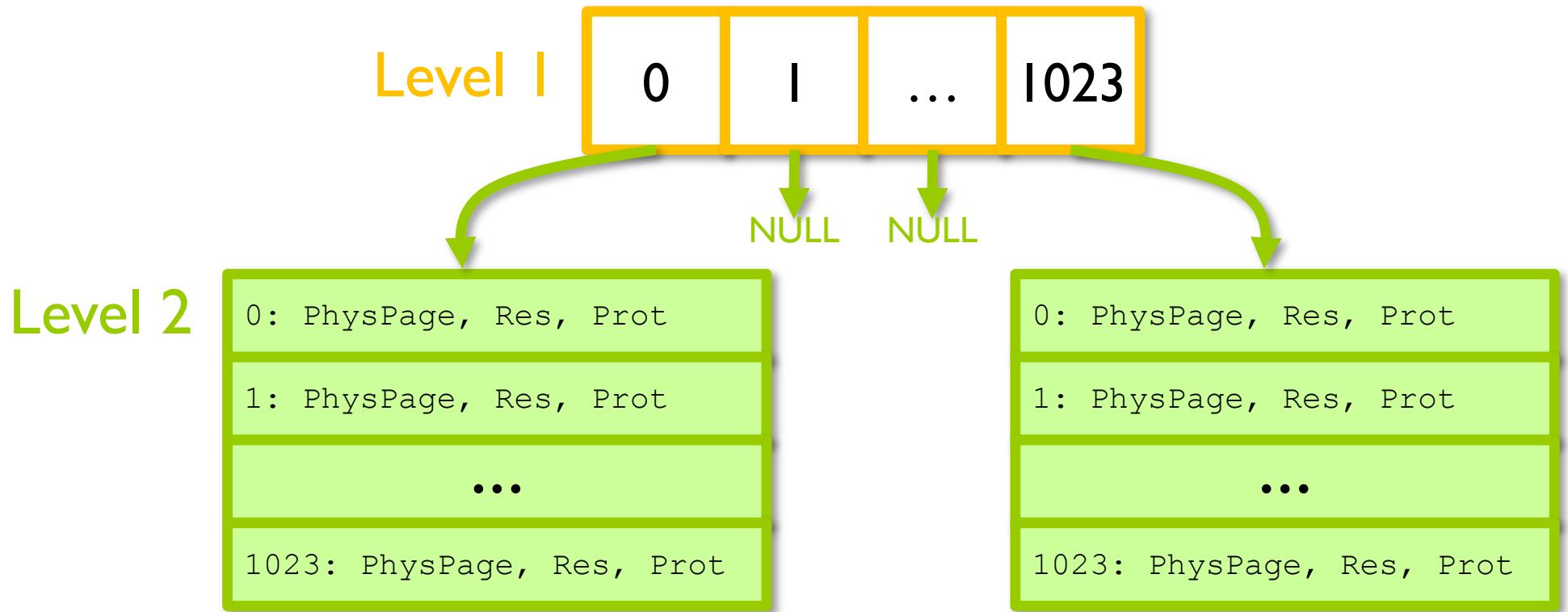


What is stored in the level 2 page table? If valid? If invalid?

# Two-level tree

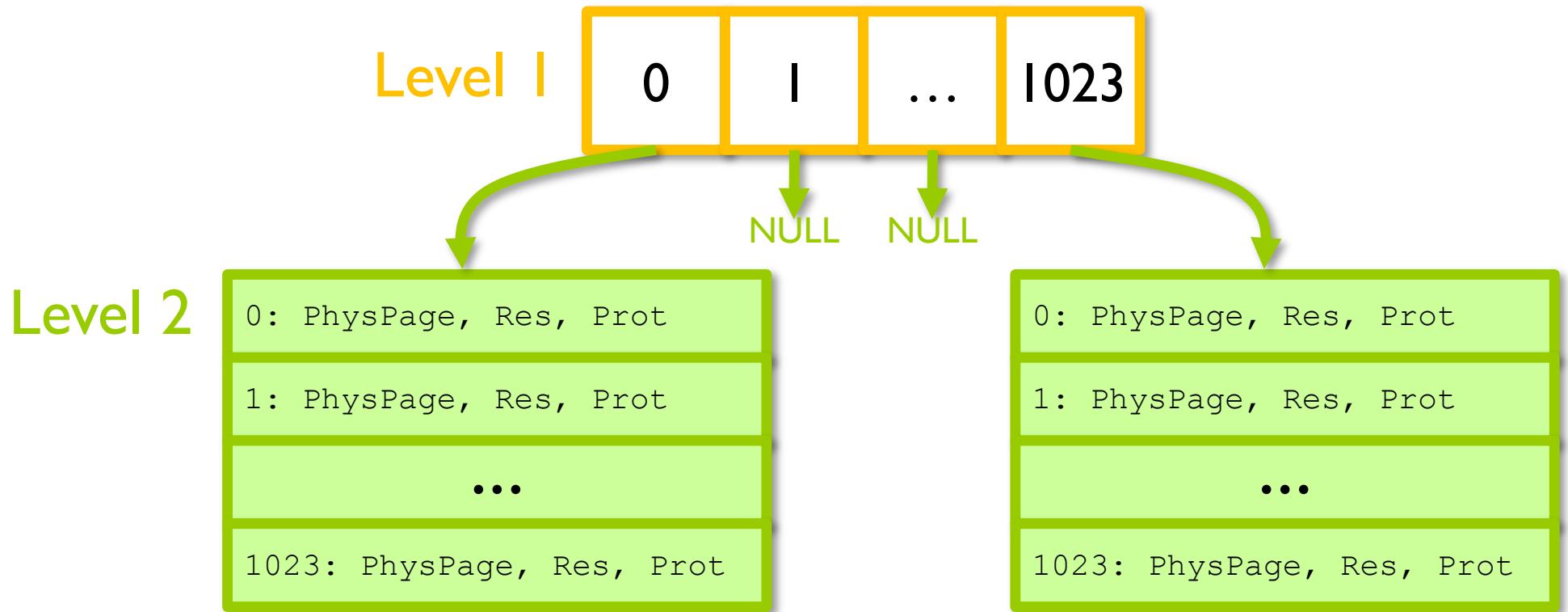


# Two-level tree



How does this save space?

# Two-level tree



What changes on a context switch?

# Multi-level translation pros and cons

- + Space-efficient for sparse address spaces
- + Easy memory allocation
- + Easy sharing
- What is the downside?

Two extra lookups per reference  
(read level 1 PT, then read level 2 PT)  
(memory accesses just got really slow)

\*Note Linux has 4 extra lookups per reference

# Translation look-aside buffer

- Aka the “TLB”
- Hardware cache
  - Maps virtual page #s to physical page #s
  - On cache hit, get PTE very quickly
  - On miss use page table, store mapping, restart instruction
- What happens on a context switch?
  - Have to flush the TLB
  - Takes time to “rewarm” the cache
  - Context switches still incur cost

# Page Replacement

- Think of physical memory as a cache
- What happens on a cache miss?
  - Page fault
  - Must decide what to evict
- Goal: reduce number of misses

# Replacement algorithms

## 1. Random

- Easy implementation, not great results

## 2. FIFO (first in, first out)

- Replace page that came in longest ago
- Popular pages often come in early
- **Problem: doesn't consider last time used**

## 3. OPT (optimal)

- Replace the page that won't be needed for longest time
- **Problem: requires knowledge of the future**

# Replacement algorithms

- LRU (least-recently used)
  - Use past references to predict future
  - Exploit “temporal locality”
  - **Problem: expensive to implement exactly**
  - Why?
    - Either have to keep sorted list
    - Or maintain time stamps + scan on eviction
    - **Update info on every access**

# LRU

- LRU is just an approximation of OPT
- Could try approximating LRU instead
  - Don't have to replace oldest page
  - Just replace an old page

# Clock

- Approximates LRU
  - Page replacement algorithm for Linux
- What can the hardware give us?
  - “Reference bit” for each PTE
  - Set each time page is accessed
- Why is this done in hardware?
  - May be slow to do in software

# Clock

- Approximates LRU
- What can the hardware give us?
  - “Reference bit” for each PTE
  - Set each time page is accessed
- What do “old” pages look like to OS?
  - Clear reference bit (haven’t been access recently)
  - Check later to which are set

# Clock

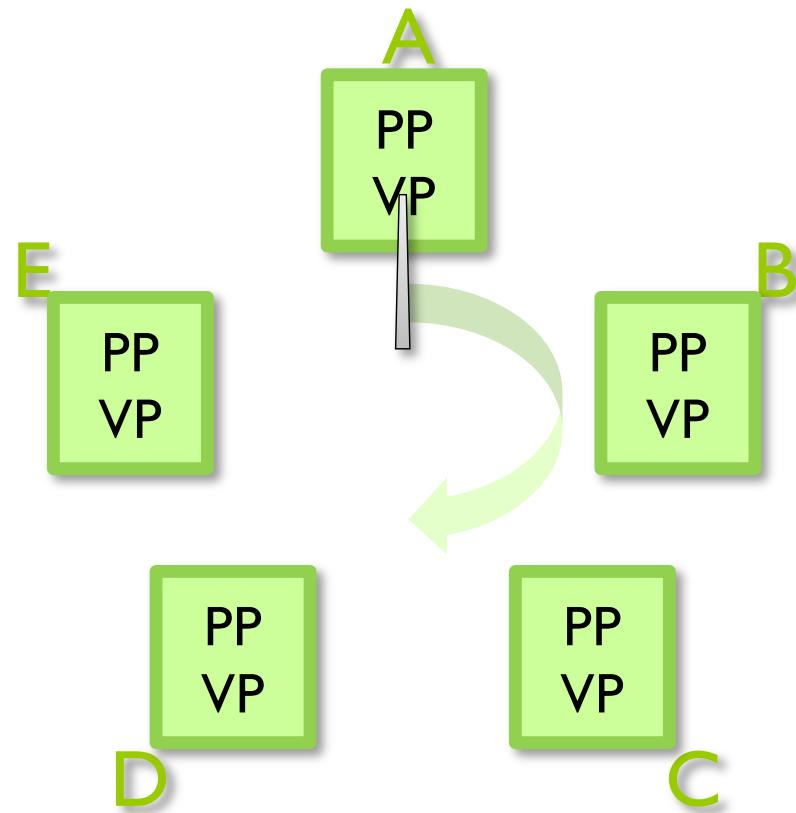
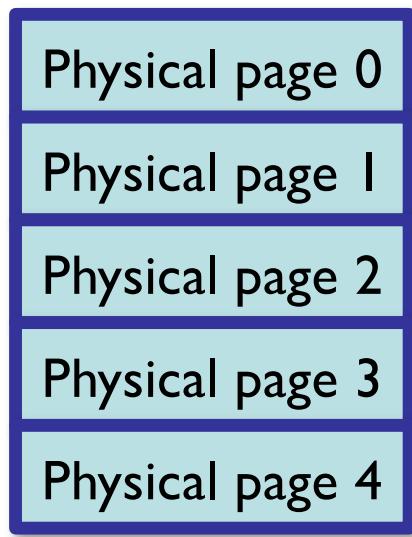
**Time 0: clear reference bit for page**

- 
- 
- 

**Time t: examine reference bit**

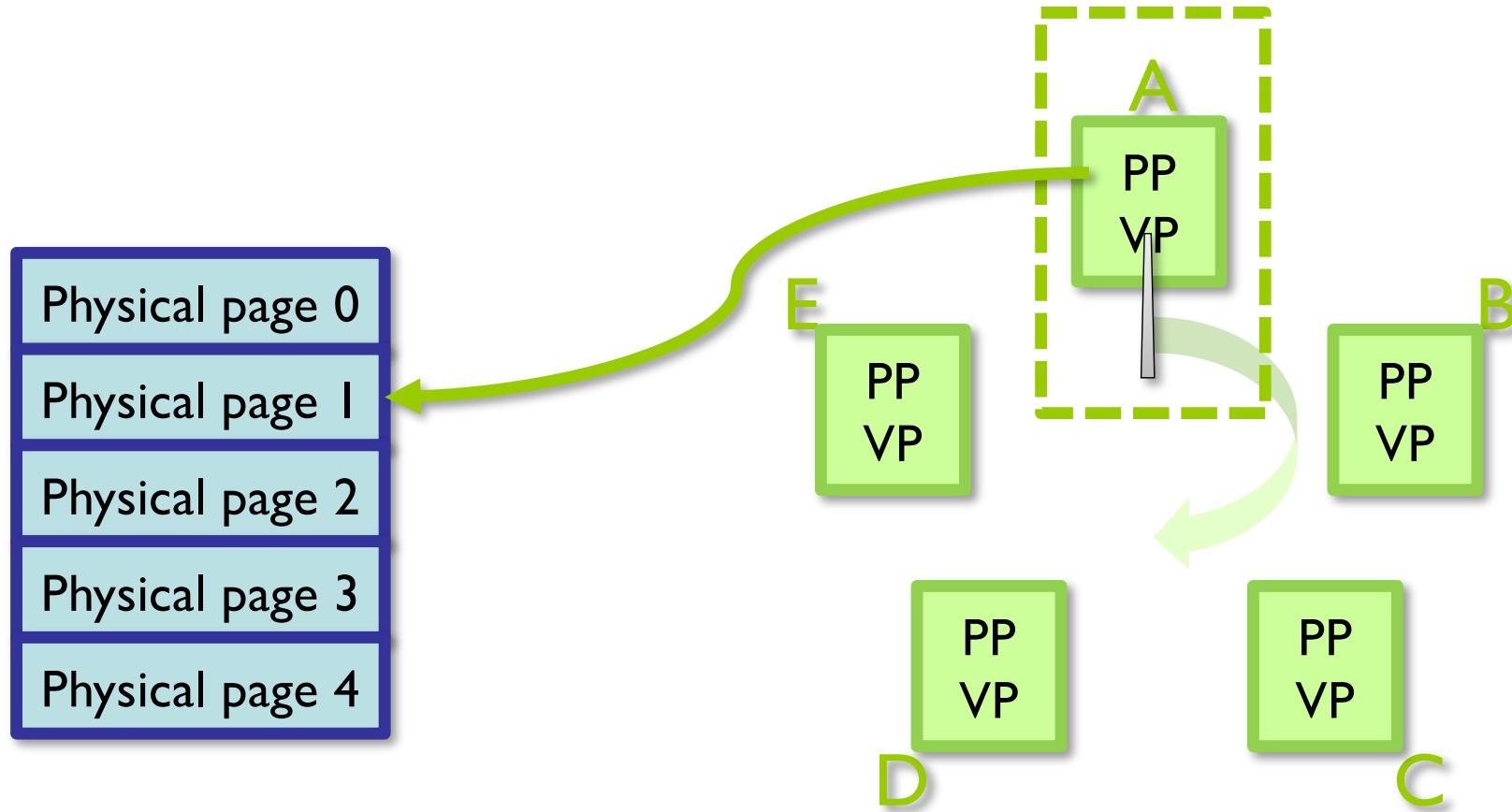
- Splits pages into two classes
  - Those that have been “touched” lately
  - Those that haven’t been “touched” lately
- Clearing all bits simultaneously is slow
- Try to spread work out over time

# Clock



= Resident virtual pages

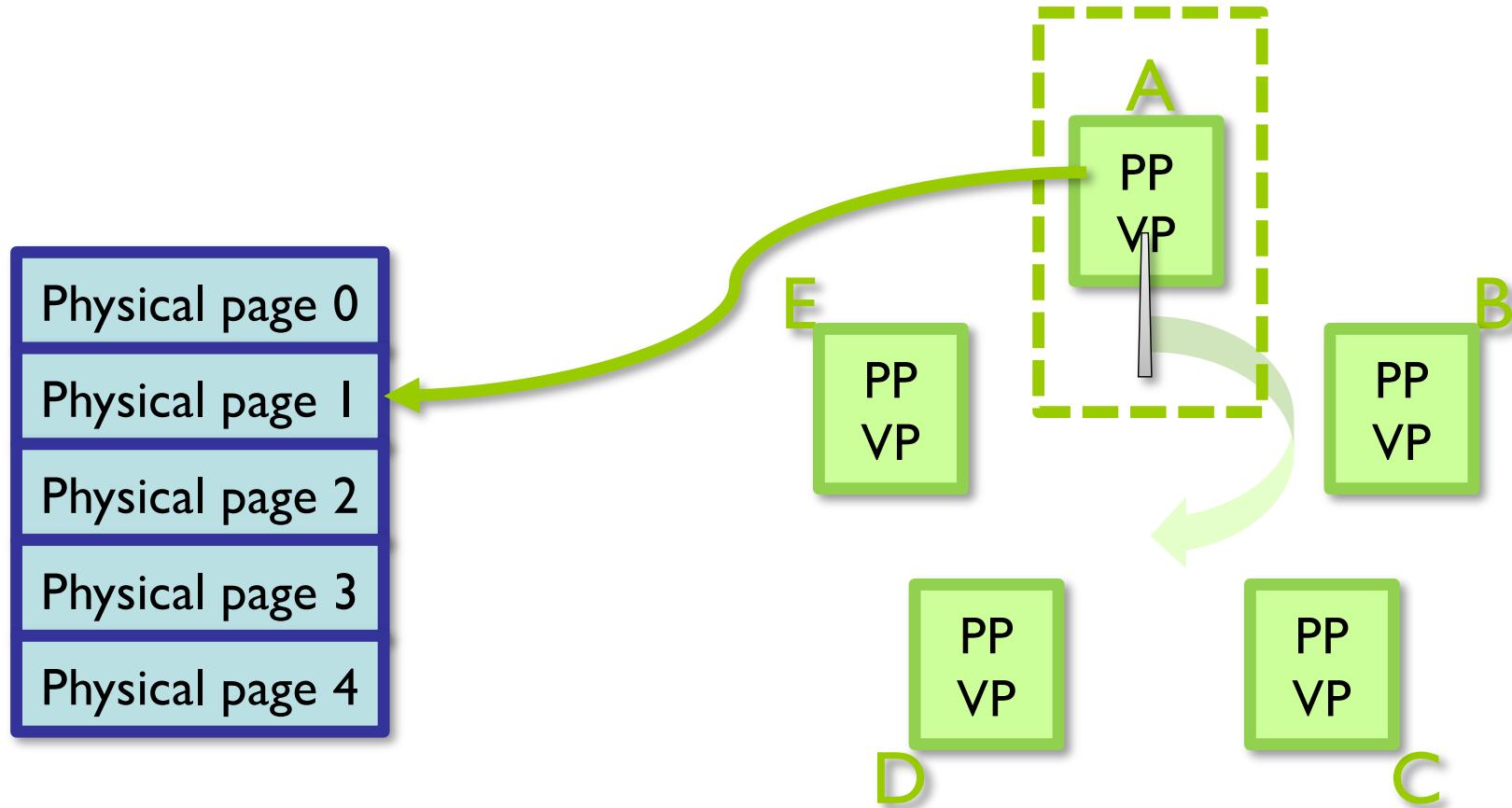
# Clock



When you need to evict a page:

- I) Check physical page pointed to by clock hand

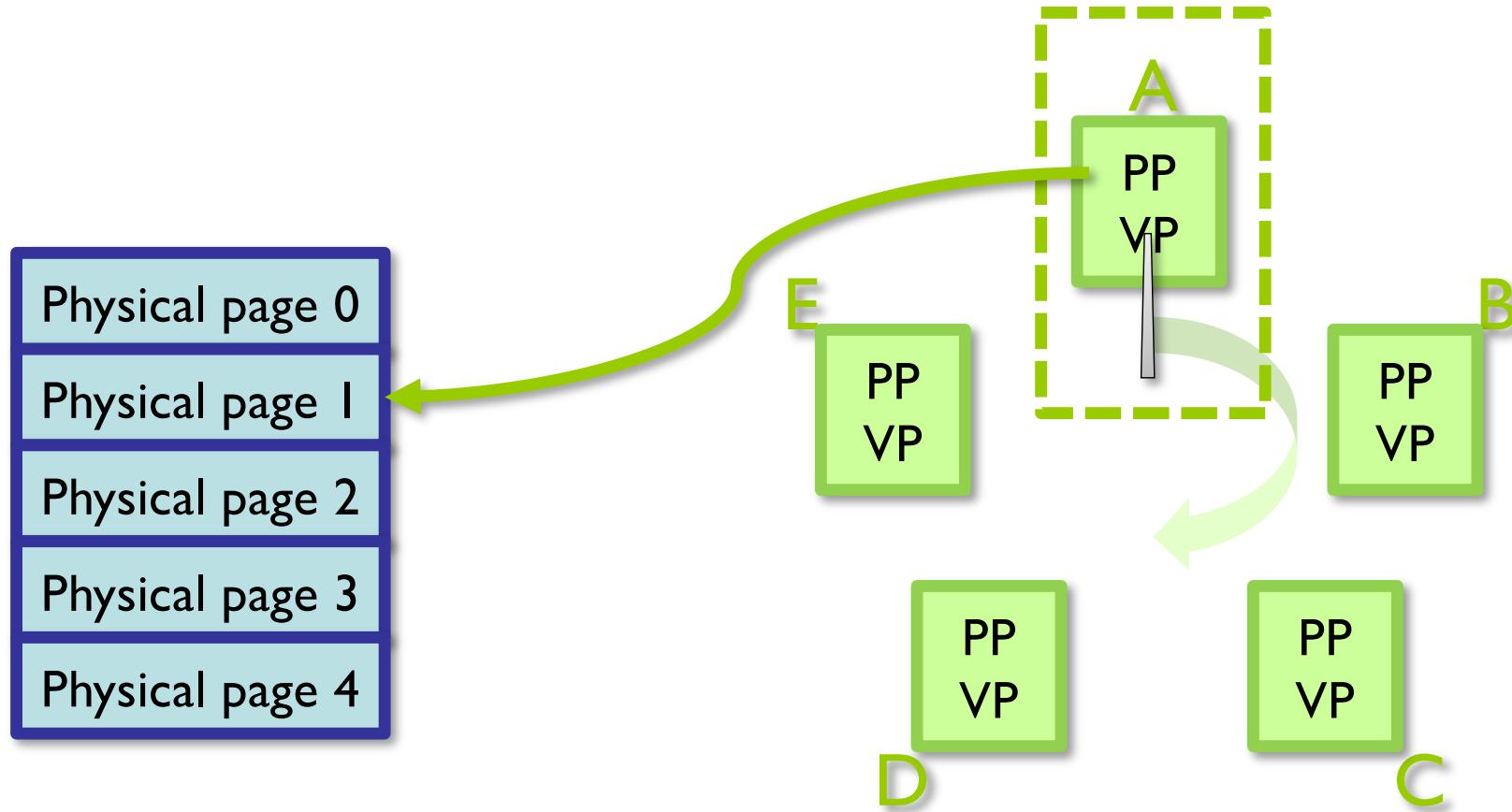
# Clock



When you need to evict a page:

- 2) If reference=0, page hasn't been touched in a while. Evict.

# Clock

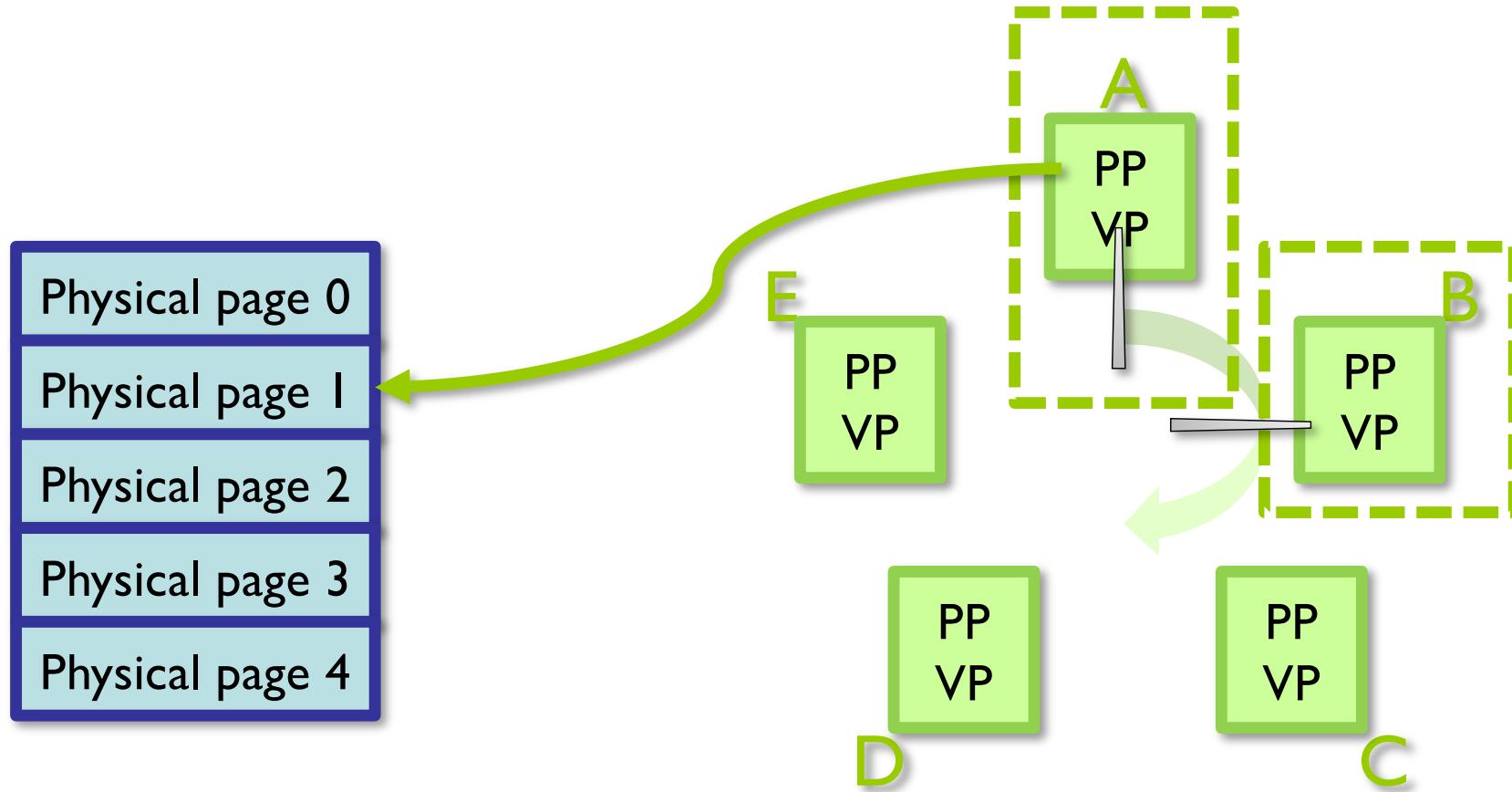


When you need to evict a page:

- 3) If reference=1, page has been accessed since last sweep.

What to do?

# Clock



When you need to evict a page:

- 3) If reference=1, page has been accessed since last sweep.  
Set reference=0. Rotate clock hand. Try next page.

# Clock

- Does this cause an infinite loop?
  - No.
  - First sweep sets all to 0, evict on next sweep
- What about new pages?
  - Put behind clock hand
  - Set reference bit to 1
  - Maximizes chance for page to stay in memory

# Paging out

- What can we do with evicted pages?
  - Write to disk
- When don't you need to write to disk?
  - Disk already has correct data (page is *clean*)
    - Need to maintain “dirty” bit, set on a write
  - Can recompute page content (zero page)

# Paging out

- Why not write to disk on each store?
  - Too slow
  - Better to defer work
  - You might not have to do it!

# Paging out

- When does work of writing to disk go away?
  - If you store to the page again
  - If the owning process exits before eviction
- Could also defer...
  - ...initializing newly allocated page with zeroes

# Paging out

- Faulted-in page must wait for the disk to write out the old page
- Can we avoid this work too?
  - Evict clean (non-dirty) pages, no need to write out
  - Write out pages during idle periods

# Hardware page table info

- What should go in a PTE?

Physical page #	Resident	Protection (read/write)	Dirty	Reference
Set by OS to control translation. Checked by MMU on each access.	Set by OS. Checked by MMU on each access.	Set by OS to control access. Checked by MMU on each access.	Set by MMU when page is modified. Used by OS to see if page is modified.	Set by MMU when page is used. Used by OS to see if page has been referenced.

What bits does a MMU need to make access decisions?  
MMU needs to know if resident, readable, or writable.  
Do we really need a resident bit? No, if non-resident, set R=W=0.

# MMU algorithm

```
if (VP # is invalid || non-resident || protected)
{
    trap to OS fault handler
}
else
{
    physical page = pageTable[virtual page].physPageNum
    physical address = {physical page}{offset}
    pageTable[virtual page].referenced = 1
    if (access is write)
    {
        pageTable[virtual page].dirty = 1
    }
}
```

**Assignment 2: infrastructure performs MMU functions**  
**Note: A2 page table entry definition has no dirty/reference bits**

# Hardware page table entries

- Do PTEs need to store disk block nums?
  - No
  - Only the OS needs this (the MMU doesn't)
- What per page info does OS maintain?
  - Which virtual pages are valid
  - On-disk locations of virtual pages

# Hardware page table entries

- Do PTEs need to store disk block nums?
  - No
  - Only the OS needs this (the MMU doesn't)
- What per page info does OS maintain?
  - Which virtual pages are valid
  - On-disk locations of virtual pages

# Hardware page table entries

- Do we really need a dirty bit?
  - Claim: OS can emulate at a reasonable overhead.
- How can OS emulate the dirty bit?
  - Keep the page read-only
  - MMU will fault on a store
  - OS/you now know that the page is dirty
- Do we need to fault on every store?
  - No. After first store, set page writable
- When do we make it read-only again?
  - When it's clean (e.g. written to disk and paged in)

# Hardware page table entries

- Do we really need a reference bit?
  - Claim: OS can emulate at a reasonable overhead.
- How can OS emulate the reference bit?
  - Keep the page unreadable
  - MMU will fault on a load/store
  - OS/you now knows that the page has been referenced
- Do we need to fault on every load/store?
  - No. After first load/store, set page readable

# Hardware page table info

- What should go in a PTE?

Physical page #	Resident	Protection (read/write)	Dirty	Reference
Set by OS to control translation. Checked by MMU on each access.	Set by OS. Checked by MMU on each access.	Set by OS to control access. Checked by MMU on each access.	Set by MMU when page is modified. Used by OS to see if page is modified.	Set by MMU when page is used. Used by OS to see if page has been referenced.

# Hardware page table info

- What should go in a PTE?

Physical page #	Resident	Protection (read/write)	Dirty	Reference
Set by OS to control translation. Checked by MMU on each access.	R=W=0 Fault on Read or Write	Set by OS to control access. Checked by MMU on each access.	W=0 Fault on any Write	R=0 Fault on any Read/Reference

Use minimum hardware bits to transfer control to the OS

OS maintains its own list of the state of each virtual page  
Once it has control, it can update/change virtual page state

# Application's perspective

- VM system manages page permissions
  - Application is totally unaware of faults, etc
- Most OSes allow apps to request page protections
  - E.g. make their code pages read-only
- Looking ahead to Assignment 2 (after Spring Break)
  - Application has no control over page protections
  - Application assumes all pages are readable/writable
  - VM system manipulates page protections to emulate resident, dirty, and reference bits

# Where to store translation data?

1. Could be kept in physical memory
  - How the MMU accesses it
  - E.g. PTBR (page table base register) points to physical memory
2. Could be in *kernel* virtual address space
  - A little weird, but what is often done in practice
  - How to keep translation data in a place that must be translated?
    - Translation for user address space is in kernel virtual memory
    - Kernel's translation data stays in physical memory (**pinned**)
  - Does anything else need to be pinned (and why)?
    - Kernel's page fault handler code also has to be pinned.

# Where to store translation data?

- How does kernel access data in user's address space?
  - Kernel can't issue a "user virtual address" (wrong page table)
  - Kernel knows physical location of user's data
  - Allows the kernel to translate manually, then access physical address

# Kernel vs. user mode

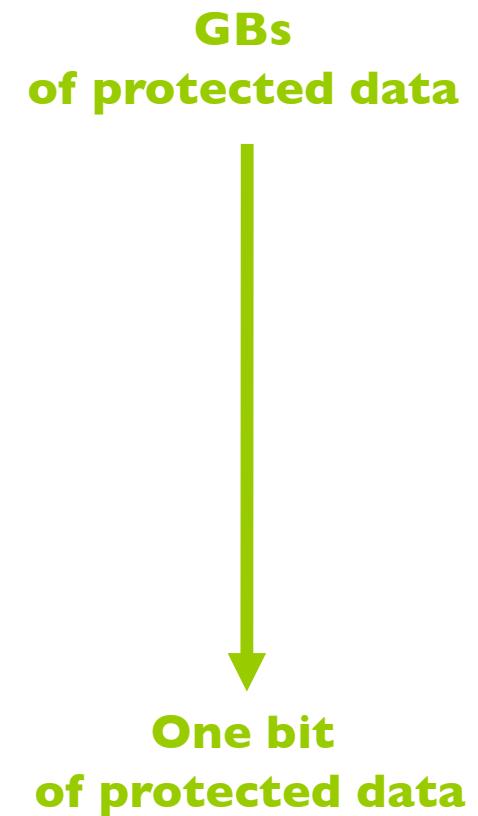
- Who sets up the data used by the MMU?
  - Can't be the user process
  - Otherwise could access anything
  - Only kernel is allowed to modify any memory
- Processor must know to allow kernel to:
  - Update the translator
  - Execute privileged instructions (halt, do I/O)
  - **Processor cannot allow process to do these things**

# Kernel vs. user mode

- How does machine know the kernel is running?
  - This requires hardware support
  - Processor supports two modes, kernel and user
  - Mode is indicated by a hardware register
- **Mode bit**

# Protection recap

1. All memory accesses go through a translator
  - Who can modify translator's data?
2. Only kernel can modify translator's data
  - How do we know if kernel is running?
3. Mode bit indicates if kernel is running
  - Who can modify the mode bit?



Making progress: the amount of protected data is down to a bit

# Protecting the mode bit

- Can kernel change the mode bit?
  - Yes. Kernel is completely trusted.
- Can user process change the mode bit?
  - Not **directly**
  - User programs need to invoke the kernel
  - Must be able to initiate a change

# When to transition from user to kernel?

## I. Exceptions (interrupts, traps)

- Access something out of your valid address space
  - (e.g. segmentation fault)
- Disk I/O finishes, causes interrupt
- Timer pre-emption, causes interrupt
- Page faults

## 2. System calls

- Similar in purpose to a function call

# Example system calls

- Process management
  - Fork/exec (start a new process), exit, getpid
- Signals
  - Kill (send a signal), sigaction (set up handler)
- Memory management
  - Brk (extend the valid address space), mmap
- File I/O
  - Open, close, read, write
- Network I/O
  - Accept, send, receive
- System management
  - Reboot

# System call implementation

- Syscalls are like functions, but different
- Implemented by special instruction
  - `syscall`
- Execution of `syscall` traps to kernel
  - Processor safely transfers control to kernel
    - Changes the mode bit
  - Hardware invokes kernel's `syscall` trap handler

# System call implementation

- Libc wraps systems calls
- C++/Java actually make calls into libc

# Tracing through “cin >> a;”

C++: `cin >> a`

Java: `in.read()`

C: `read ()`

libc: `syscall(SYS_read,filenum,offset,numbytes)`

Processor: kernel's `syscall handler`

kernel: `sys_read`

kernel: // perform I/O

Crucial  
step

# Kernel trap details

- Hardware must **atomically**:
  1. Set processor mode bit to “kernel”
  2. Save current registers (SP, PC, general purpose)
  3. Change execution stack to kernel (SP)
  4. Change to use kernel’s address space (PTBR)
  5. Jump to exception handler in kernel (PC)
- What does this look a lot like?
  - Swapcontext from Assignment 1

# Kernel trap details

- User process can initiate mode switch
  - But only transfers control in limited way
  - (i.e. to predefined kernel code)
- How does processor know where to jump?
  - Stored in hardware “interrupt vector table”
- Who can update the interrupt vector table?
  - Kernel does this at startup
  - Code that runs first gets control of machine

# Syscall arguments, return values

- For arguments passed in memory
  - In which address space do they reside?
  - User (programs can't access kernel's address space)
  - Parameters pushed onto user stack before `syscall`
- Kernel knows location of user stack
- Kernel can manually translate to physical address
- You'll do this in Assignment 2 for `vm_syslog`

# Tracing through read()

C: read (int fd, void \*buf, size\_t size)

libc: push fd, buf, size onto user's stack

Note: pointer!

```
kernel: sys_read () {  
    verify fd is an open file  
    verify [buf – buf+size] valid, writable  
    read file from data (e.g. disk)  
    find physical location of buf  
    copy data to buf's physical location  
}
```

} Verify arguments!

# Creating, starting a process

1. Allocate process control block
2. Read program code from disk
3. Store program code in memory (could be demand-loaded too)
  - Physical or virtual memory?
  - (Physical: no virtual memory exists for process yet)
4. Initialize machine registers for new process
5. Initialize translator data for new address space
  - E.g. page table and PTBR
  - Virtual addresses of code segment point to correct physical locations
6. Set processor mode bit to “user”
7. Jump to start of program

# Multi-process issues

- Can think of physical memory as a cache
  - Cache of what?
  - Virtual pages
- How do I allocate pmem among different processes?
  - Another term that gets used: “multiplex”
  - Allocation policy is always an issue for shared resources
  - Often a tradeoff between global optimality and fairness

# Global vs local replacement

- **Global replacement**
  - All pages are equal when looking to evict
  - Process grows if another's pages are evicted
- **Local replacement**
  - Only consider pages of the same process
  - Still have to decide an initial allocation
- How should we compare the two?

# Thrashing

- **What happens if we have lots of big processes?**
  - They all need a lot of physical memory to run
  - Switching between processes will cause lots of disk I/O
- **For most replacement algorithms**
  - Performance degrades rapidly if not everything fits
  - OPT degrades more gracefully
- This is called “thrashing”

# Why thrashing is so bad

- **Average access time**
  - $(\text{hit rate} * \text{hit time}) + (\text{miss rate} * \text{miss time})$
  - Hit time = .0001 milliseconds (100 ns)
  - Miss time = 10 milliseconds (Disk I/O)
- **What happens if hit rate goes from 100% → 99%?**
  - 100% hit rate: avg access time = .0001 ms
  - 99% hit rate: avg access time =  $.99 * .0001 + .01 * 10 = .1$  ms
  - 90% hit rate: avg access time =  $.90 * .0001 + .1 * 10 = 1$  ms
- **1% drop in hit rate leads to 1000 times increase in access time!**

# Why thrashing is so bad

- **Easy to induce thrashing with LRU and clock**
  - Even if they are 99% accurate, thrashing kills
- **Another way to look at it**
  - 1ms per instruction
  - 1 cycle per instruction
  - Every few instructions access memory
  - End up with a 3GHz processor running at 1KHz

# What to do about thrashing

- Best solution
  - Throw money at the problem (buy more RAM)
- No solution
  - One process “actively uses” more pages than fit
- What if only the sum of pages doesn’t fit?
  - Run fewer at a time (i.e. swap processes out)

# What to do about thrashing

- Example
  - Four processes: A, B, C, and D
    - Each uses 500 MB of memory
    - System has 1GB of physical memory
    - What can we do?
  - Don't run A, B, C and D together!
    - Run A and B together
    - Then run C and D together
  - How would this help?

# What to do about thrashing

- Another solution
  - Run each process for a longer slice
  - Amortize cost of initial page faults
- Extend the fast phase of execution
  - i.e. the part after its pages are in pmem

# Working-Set Model

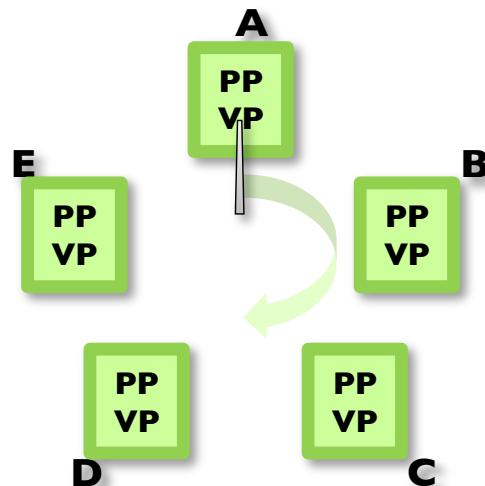
- **Idea:** For all active processes, only keep pages that we are “actively using” in memory
- Working set ( $T$ )
  - All pages used in last  $T$  seconds
  - Larger working sets require more pmem
- Sum of all working sets for all processes must fit in pmem
  - Otherwise you’ll have thrashing

# Working sets

- Want to run a “balance set” of processes
- **Balance set**
  - Subset of all processes
  - Sum of working sets fits in physical memory
- **Intuition**
  - Run processes whose T-working sets fit
  - Suspend other processes
  - Can go T seconds without getting a fault

# Working sets

- **How do we know the size of the working sets?**
  - Updating working set on each mem reference is slow
  - Can approximate using modified clock algorithm
  - Sweep at fixed intervals (instead of on faults)
  - On sweep, see if page was referenced
  - If page has been used in last T seconds, it's in working set
  - Reset all reference bits on sweep



# Putting it all together

- Unix fork, exec, and shell
- How does Unix create processes?

## 1. Fork

- New (child) process with one thread
- Address space is an exact copy of parent (file descriptors, stack, ..., everything!)

## 2. Exec

- Address space is initialized from new program

# Fork

- Problem with exactly copying an address space?
  - How does child know it is the child?
  - (child should probably do something different than parent)
- So it can't be an **exact** copy
- Child is distinguished by one variable
  - The return value of fork

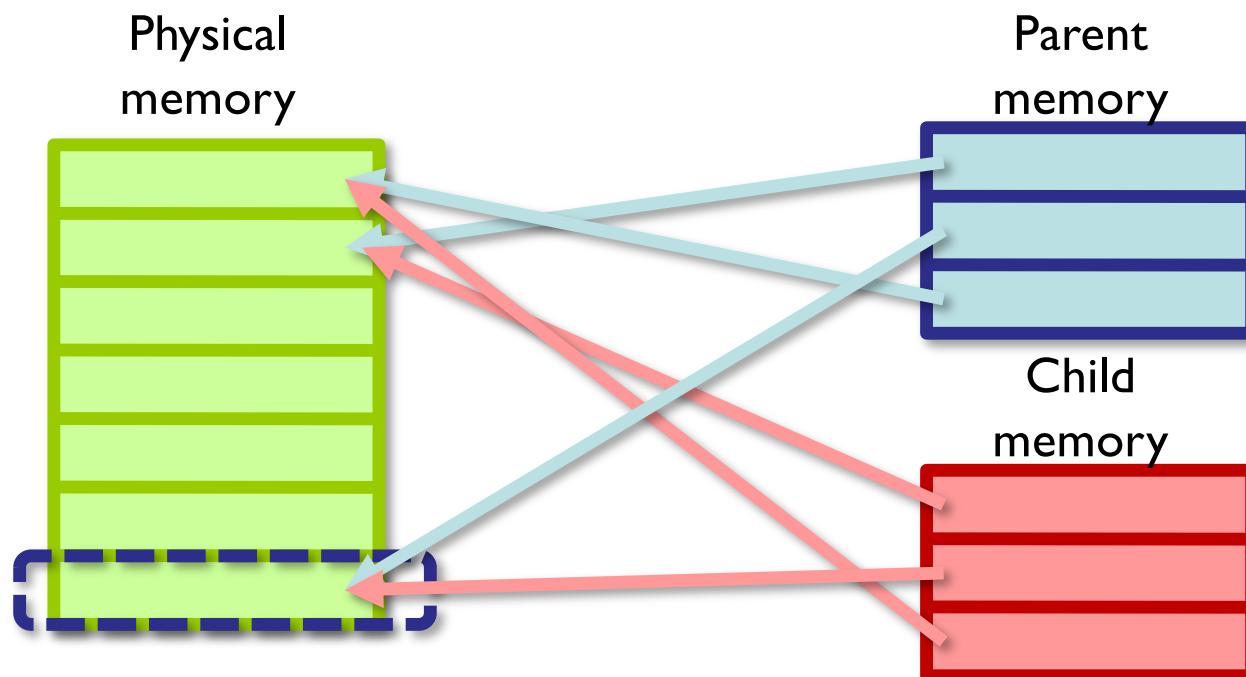
```
if (fork () == 0) {  
    /* child */  
    execute new program  
} else {  
    /* parent */  
    carry on  
}
```

# Fork

- How can we efficiently create the “copy” address space?

# Fork

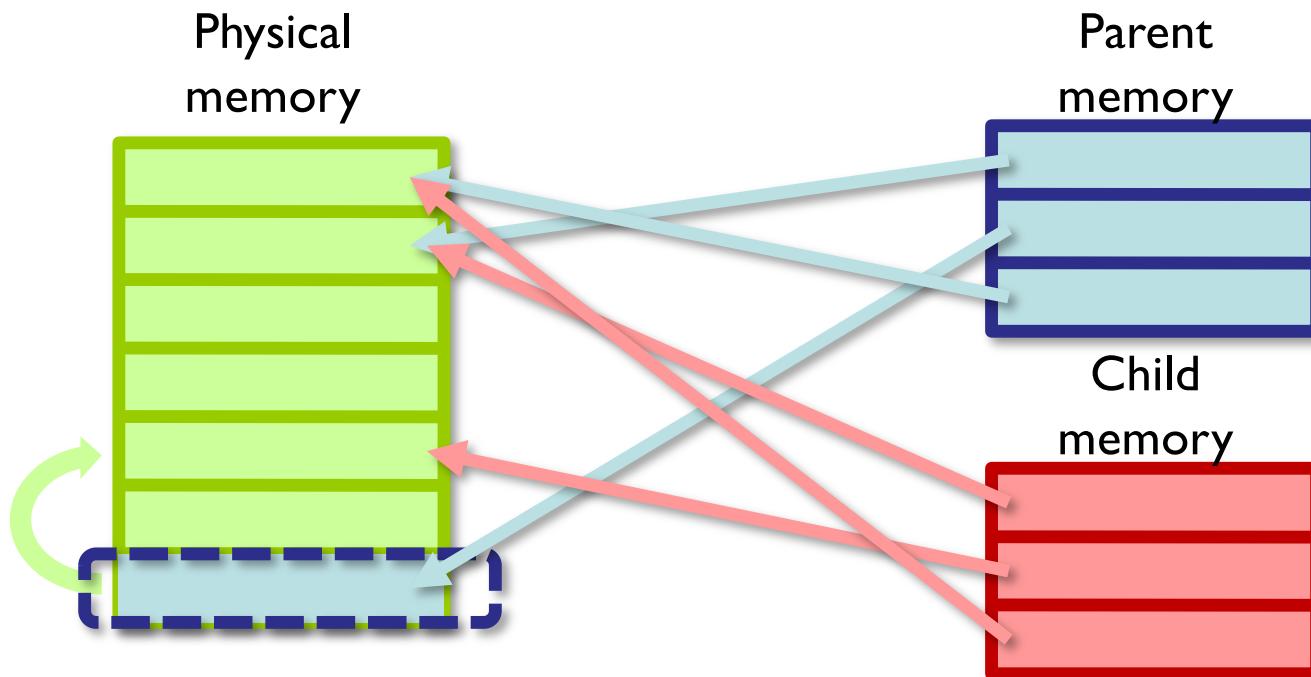
- How can we efficiently create the “copy” address space?
  - Copy-on-write



What happens if parent writes to a page?

# Fork

- How can we efficiently create the “copy” address space?
  - Copy-on-write



What happens if parent writes to a page?  
Have to create a copy of pre-write page for the child.

# Why use fork?

- Why copy the parent's address space?
  - You're probably just going to throw it out anyway with exec...
- Sometimes you want a new copy of same process
- Separating fork and exec provides flexibility
- Also, the child can inherit some kernel state
  - E.g. open files, stdin, stdout
  - Used by shells to redirect input/output

# Shells (bash, windows explorer)

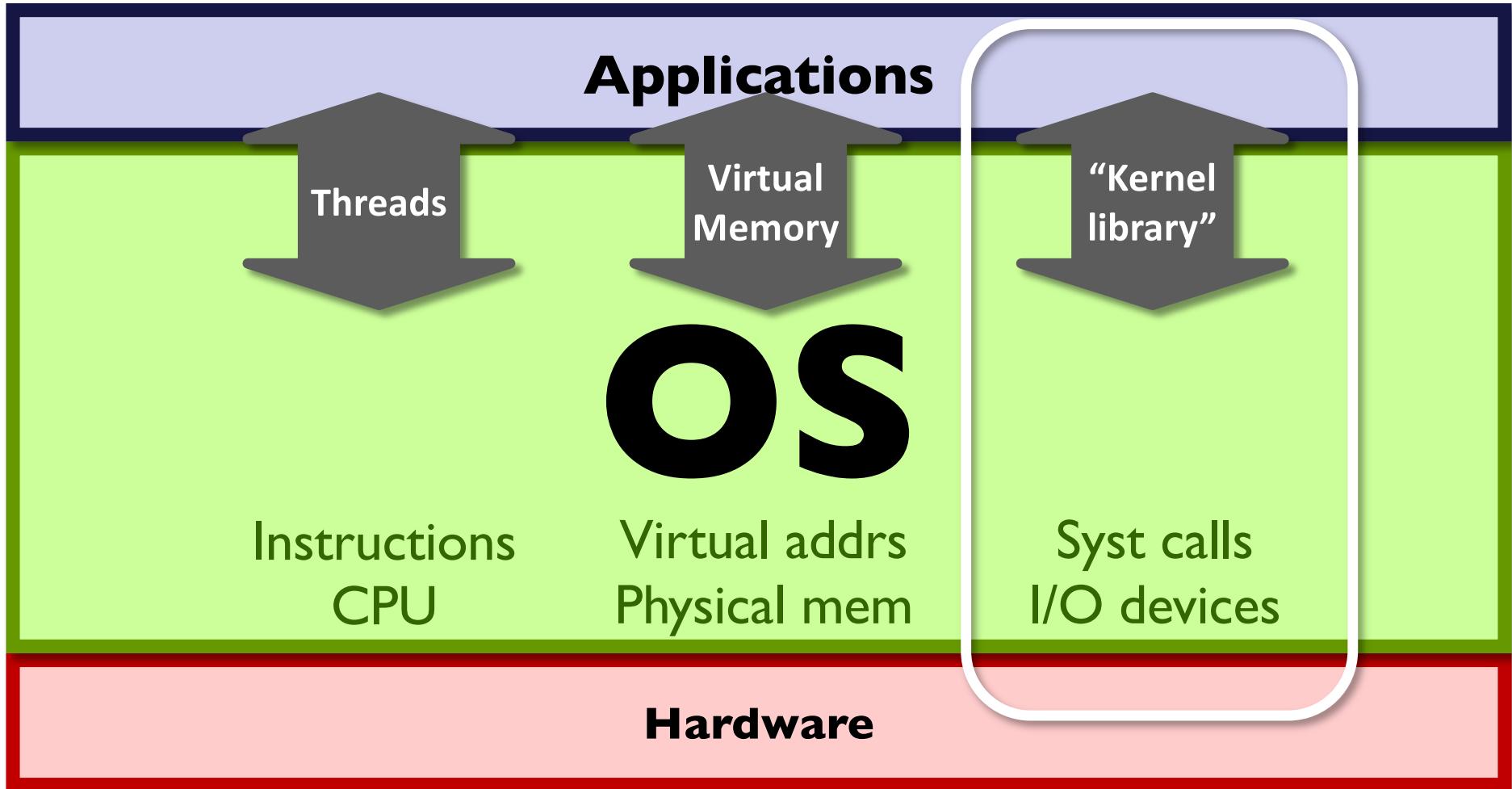
- Shells are normal programs
  - Though they look like part of the OS
- How would you write one?

```
while (1) {  
    print prompt ("bash% ")  
    ask for input (cin)  
    // e.g. "ls /tmp"  
    first word of input is command  
    // e.g. ls  
    fork a copy of the current process (shell)  
    if (child) {  
        redirect output to a file, if requested (or a pipe)  
        exec new program (e.g. with argument "/tmp")  
    } else {  
        wait for child to finish  
        or can run child in background and ask for another command  
    }  
}
```

Moving on...

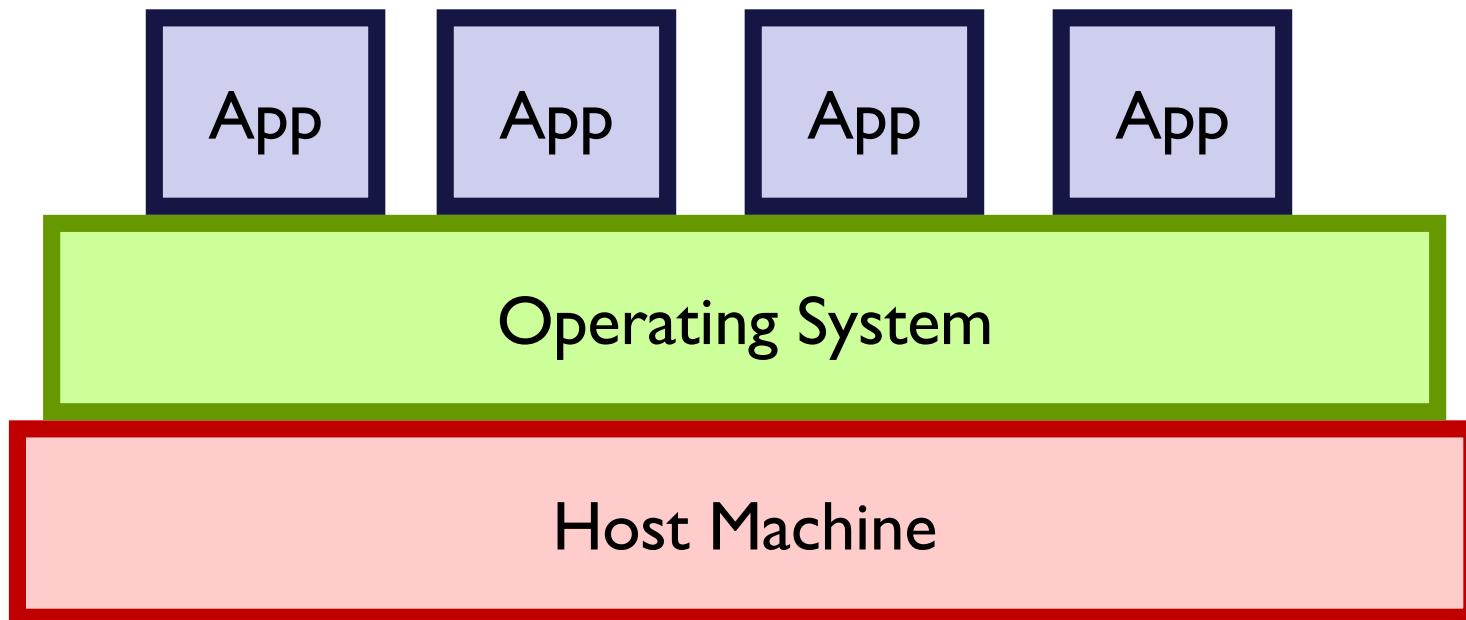
# OS abstractions

Last month of class



What are the interfaces and the resources?  
What is being virtualized?

# Traditional OS structure



# Coarser abstraction: virtual machine

- We've already seen a kind of virtual machine
  - OS gives processes virtual memory
  - Each process runs on a virtualized CPU
- **Virtual machine**
  - An execution environment
  - May or may not correspond to physical reality

# Virtual machine options

- How to implement a virtual machine?

## I. Interpreted virtual machines

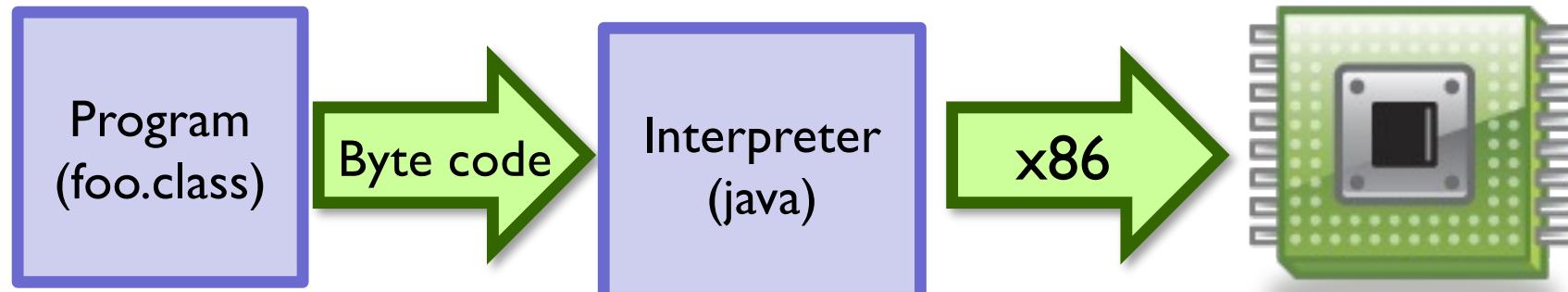
- *Translate* every VM instruction
- Kind of like on-the-fly compilation
- VM instruction → HW instruction(s)

## 2. Direct execution

- Execute instructions directly
- Emulate the hard ones

# Interpreted virtual machines

- Implement the machine in software
- Must translate emulated to physical
  - Java: byte codes → x86, PPC, ARM, etc
- Software fetches/executes instructions



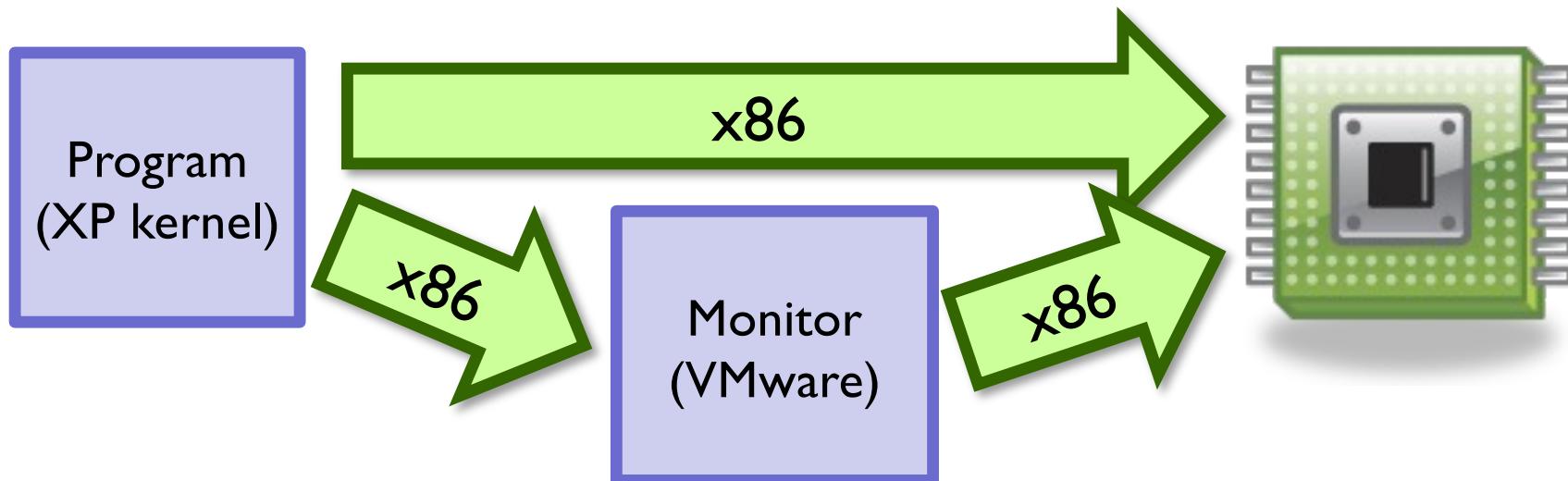
What does this picture look like?  
Dynamic virtual memory translator

# Java virtual machine

- **What is the interface?**
  - Java byte-code instructions
- **What is the abstraction?**
  - Stack-machine architecture
- **What are the resources?**
  - CPU, physical memory, disk, network
- **The Java programming language**
  - High-level language compiled into byte code
  - Library of services (kind of like a kernel)
  - Like C++/STL, C#

# Direct execution

- **What is the interface?**
  - Hardware ISA (e.g. x86 instructions)
- **What is the abstraction?**
  - Physical machine (e.g. x86 processor)
- **What are the resources?**
  - CPU, physical memory, disk, network



# Different techniques

- Emulation
  - Bochs, QEMU
- Full virtualization
  - **VMware**
- Para-virtualization
  - **Xen**
- Dynamic recompilation
  - Virtual PC

# Virtual machine options

- How to implement a virtual machine?

## I. Interpreted virtual machines

- *Translate* every VM instruction
- Kind of like on-the-fly compilation
- VM instruction → HW instruction(s)

## 2. Direct execution

- Execute instructions directly
- Emulate the hard ones

# Singularity

Microsoft (finally) releases Singularity, the research oriented operating system – Engadget

<http://www.engadget.com/2008/03/05/microsoft-finally-releases-singularity-the-res>

Home Smart Bookmarks ▾ Places ▾ Getting Started Latest Headlines ▾ Apple ▾ Amazon eBay Yahoo! News ▾ Details of last 100 v...

**engadget**

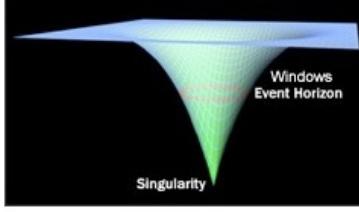
## Microsoft (finally) releases Singularity, the research oriented operating system

Posted Mar 5th 2008 2:28AM by Ryan Block  
Filed under: Desktops, Laptops

It's been in development for nearly half a decade, but this year at Microsoft's R&D extravaganza TechFest, the company finally lifted the curtain on its research-oriented Singularity OS. Let's just be clear from the get-go, though: while it's available for immediate use, Singularity is nowhere near anything you'd replace your desktop OS with.

The sole intention here is to test out futuristic new concepts in application interaction, microkernel architecture, and so on, so don't expect to hear that Microsoft is hanging up the Vista apron or anything. But for the turbo-geeks in the crowd, the Singularity Research Development Kit (RDK) 1.1 is now available for download for academic non-commercial use. And for the rest of us, well, we'll just see what the year 2011 holds in store.

[Thanks to everyone who sent this in]



**BREAKING NEWS ▾**

-  iPhone 2.0 software update hits in June
-  Apple announces App Store for iPhone, iPod touch
-  iPhone SDK gets real, available today
-  iPhone heads to Enterprise-ville
-  Samsung kills BD-UP5500 combo player before it ever truly lived

**FEATURED STORIES ▾**

-  Live from Apple's iPhone SDK press conference
-  Hands on Clevo's OEM-ready 15- and 17-inch gaming rigs
-  Eyes on NVIDIA's GeForce 9800 GX2

**Tip us on news!**  
[RSS Feed](#) | [Category Feeds](#)  
Engadget in: [Español](#) | [繁體字](#) | [简体字](#) | [日本語](#)

**Current Offers from LendingTree®**  
Refinance Loan \$400,000 for \$2,272/month\*  
Refinance Loan \$300,000 for \$1,646/month\*  
Refinance Loan \$200,000 for \$1,090/month\*

\*Terms & conditions apply

**LendingTree**

latest cellphone and mobile news from **engadget:mobile**

Can you do that? Verizon renames year-old Samsung U740 to "Alias"

Everyone else sued over picture caller ID

Padded lampposts for distracted texters being tested in London

Flash on the iPhone: Apple has Goldilocks syndrome

VODIA Y1 shows AT&T graphic at CNET what

Done

# Singularity

- “Microkernel” OS
- Components execute in same address space
  - Processes are “software-isolated”
  - Avoids cost of context switch
  - Protection ensured by language and static analysis
    - Written in memory-managed language (like Java)
    - No pointers or pointer arithmetic allowed
  - Protection invariants checked at install time

# Full vs interpreted

- Why would I use VirtualBox instead of Java?
  - Support for legacy applications
  - Do not force users to use a particular language
  - Do not force users to use a particular OS
- Why would I use Java instead of VirtualBox?
  - Lighter weight
  - Nice properties of type-safe language
  - Can prove safety at compile time

# Full vs interpreted

- What about protection?
- What does Java use for protection? VirtualBox?
  - Java relies on language features (cannot express unsafe computation)
  - VirtualBox relies on the hardware to enforce protection (like an OS)
- What are the trade-offs? Which protection model is better?
  - Java gives you stronger (i.e. provable) safety guarantees
  - Hardware protection doesn't constrain programming expressiveness
- What about sharing (kind of the opposite of protection)?
  - Sharing among components in Java is easy
  - (call a function, compiler makes sure it is safe)
  - Sharing between address spaces is more work, has higher overhead
  - (use sockets, have to context switch, flush TLB, etc)

# Traditional Virtualization

Each option below provides abstraction of  
**isolated/customizable OS** and **isolated resources**

- Level of isolation varies from weak to strong
- Resource Containers
  - LXC and Docker
- Emulate virtual hardware
  - Bochs, QEMU
- Full virtualization
  - **VMware, VirtualBox**
- Para-virtualization
  - **Xen**

# Traditional Virtualization

Don't confuse with software that provides an OS abstraction, but not any resource isolation

- **Cygwin** – expose Unix system call interface in Windows
  - Translates Unix system calls to Windows system calls
- **Wine** – expose Windows system call interface in Linux
  - Translates Windows system calls to Unix system calls
- “Virtualize” at the system call layer, enabling applications native to one OS to run on another OS
  - Hard to get right b/c system call interface is large and a moving target
  - There is not always a one-to-one mapping between Linux and Windows

# Foundation of “cloud”

San Jose Mercury News – VMware IPO: Silicon Valley giant is born

http://www.mercurynews.com/business/cl\_6626949?nclick\_check=1

Getting Started Latest Headlines Apple Amazon eBay Yahoo! News

**MercuryNews.com**  
The Mercury News

home news business tech sports entertainment life & style opinion my city help jobs cars real estate classifieds shopping place an ad

real estate | green energy | special reports | financial markets | personal finance | venture capital | drive / automotive

Most Viewed Most Emailed (From the last 12 hours) RSS

del.icio.us Digg Reddit YahooMyWeb Google Facebook What's this?

**VMware IPO: Silicon Valley giant is born AS SHARES ZOOM 76%, VMWARE JOINS RANKS OF ORACLE, ADOBE**

By Scott Duke Harris Mercury News Article Launched: 08/15/2007 01:34:14 AM PDT

VMware, a company recently known only to hard-core technologists, debuted as the darling of Wall Street on Tuesday, with an opening-day surge that exceeded even Google's historic 2004 launch.

Stock of the Palo Alto maker of "virtualization" software soared 76 percent, eclipsing Google's 18 percent first-day gain. VMware's value at closing was \$19.1 billion - ranking it as Silicon Valley's third-largest home-grown software company after Oracle and Adobe Systems. It was the largest initial public offering since Google achieved a \$27 billion valuation.

Time will tell if VMware can sustain its appeal. But Tuesday's rocket ride, as chief executive Diane Greene put it, "was a little heady" for founders who rang the opening bell at the New York Stock Exchange. It also touched off an early-morning champagne celebration at VMware's campus in Palo Alto.

An estimated 1,000 employees - including a small number who had stayed overnight at an office slumber party - gathered for a special 6 a.m. breakfast in the company cafeteria and other rooms to watch a live video feed from the NYSE floor. Employees, who were served scrambled eggs, pastries,

Click photo to enlarge

Diane Greene, president and CEO of VMware Inc., listens to NYSE CEO John Thain... (RICHARD DREW)

1 2 >

Now: Fair and 76°F Today: 73°F Fri: 63°F Sat: 66°F

Done

Xen sale: \$500 million

## VMware IPO: \$19.1 billion

San Jose Mercury News – Palo Alto-based Xensource bought for \$500 million

http://www.mercurynews.com/search/cl\_6630304

Apple Amazon eBay Yahoo! News

WS.com

sports entertainment life & style opinion my city help jobs cars real estate classifieds shopping place an ad apartments | real estate videos | commercial properties | showcase | open homes | list a home

del.icio.us Digg Reddit YahooMyWeb Google Facebook What's this?

**Palo Alto-based Xensource bought for \$500 million**

By Niraj Sheth Mercury News Article Launched: 08/15/2007 12:21:35 PM PDT

Citrix Systems today announced an agreement to buy XenSource, a privately owned open-source virtualization firm based in Palo Alto, for \$500 million in cash and stock. The move brings Citrix into direct competition with VMware, whose IPO surge Tuesday signaled a growing tide of interest in virtualization software.

In making the deal, the Florida-based developer of access infrastructure software said that it expects the server and desktop virtualization market to balloon to \$5 billion by 2011.

Both companies have cultivated strong partnerships with Microsoft, which is planning to introduce virtualization software codenamed "Viridian" in the future. Citrix said that it expects its relationship with the Redmond company to become even stronger.

The acquisition, which includes the assumption of approximately \$107 million in unvested stock options by Citrix, is expected to close in the fourth quarter.

Related Stories

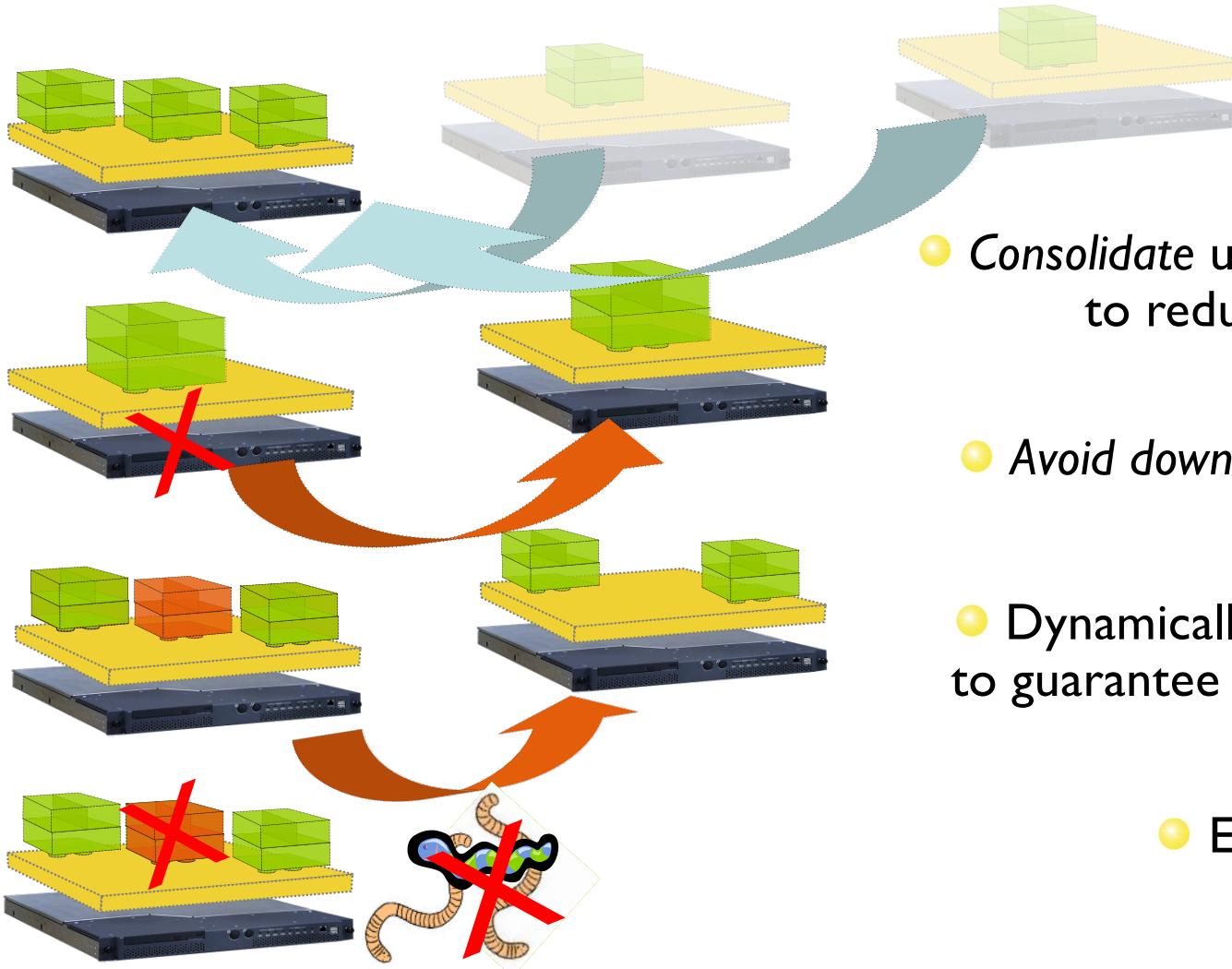
Aug 15:

- Wall Street retreats as credit concerns grow
- VMware IPO: Silicon Valley giant is born

Now: Fair and 76°F Today: 73°F Fri: 63°F Sat: 66°F

Done

# Benefits of VMs to Cloud



- *Consolidate under-utilized servers to reduce CapEx and OpEx*
- *Avoid downtime with relocation*
- *Dynamically re-balance workload to guarantee application quality of service*
- *Enforce security policy*

# Isolating Cloud Users

- Option I: Unmodified OS
  - Each app gets a login username
  - Access resources through system calls
  - Users can see other users' files, processes
  - Drawbacks of this approach?
    - Administration, configuration headaches
    - (e.g. which libraries are installed?)
    - No performance isolation
    - (one process can dominate CPU, buffer cache, bandwidth)

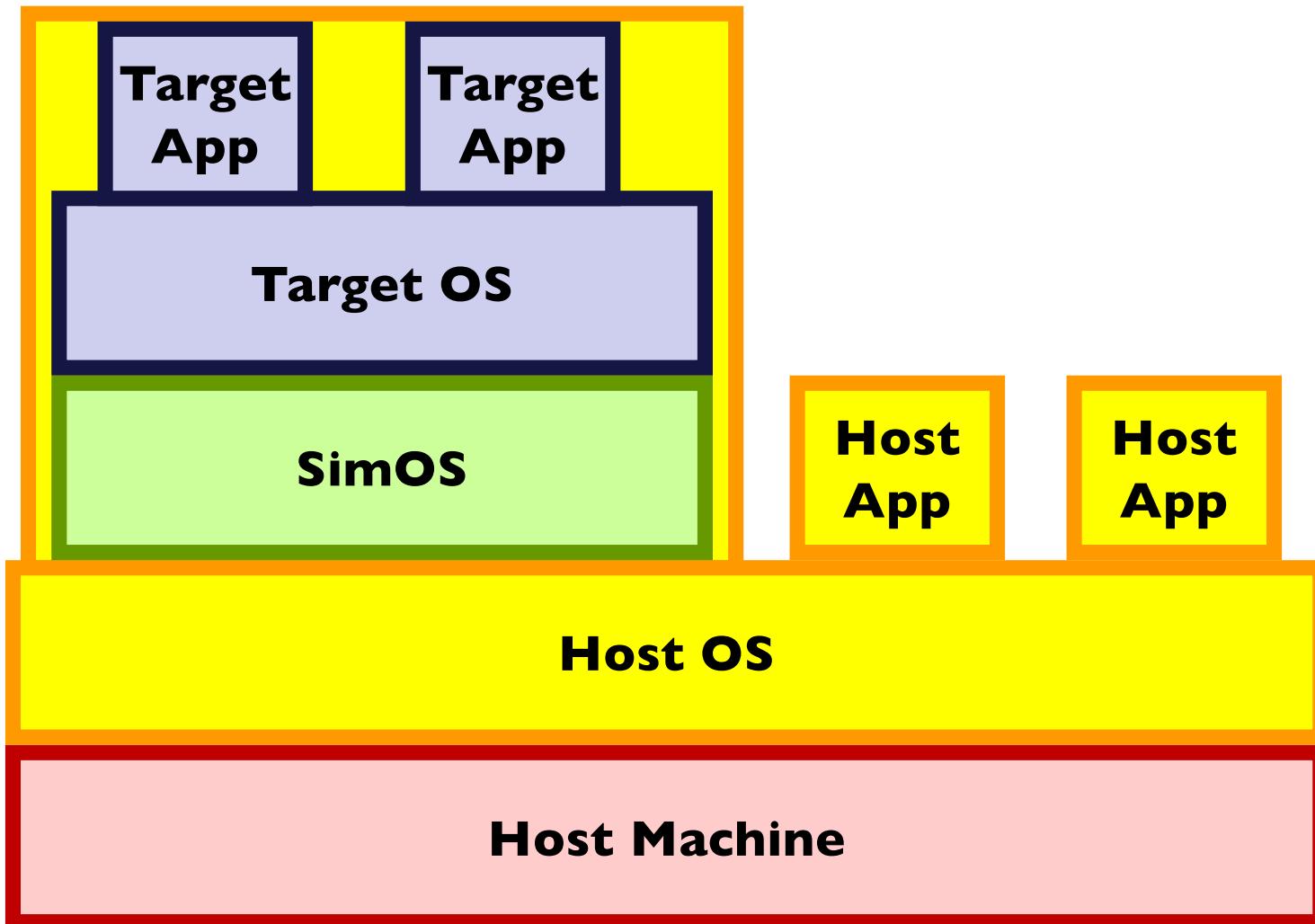
# Isolating Cloud Users

- Option 1: Unmodified OS
- Option 2: Retrofit resource accounting into OS
  - Resource containers (LXC/Docker) - access resources through system calls
  - Virtualize some resources
    - Each app has own process table, file system (with own libraries), etc.
- Benefits
  - Much less overhead than virtualization – fast boot-up times, more scalable
- Drawbacks of this approach?
  - How do you know that you've virtualized everything you need to
  - Especially hard for **software** resources
    - What about entries in the file descriptor table? Open ports?
    - Who gets charged on a page fault?
  - Container migration hard to get right
  - Everyone must use same OS (w/same kernel version, modules, etc.)

# Isolating Cloud Users

- Option 1: Unmodified OS
- Option 2: Retrofit resource accounting into OS
- Option 3: Virtual machines (SimOS/Bochs, Xen, VMware)
  - Virtualize hardware interface
  - Each app gets to choose its own OS
    - Apps have their own virt. CPU, physical memory, disk
    - Strongest isolation possible
    - Migration is simpler
  - Drawbacks of this approach?
    - Very heavy-weight, lower performance
    - A lot of redundant state (e.g. kernel, libraries, executables)
    - Might not scale well, server can support fewer VMs

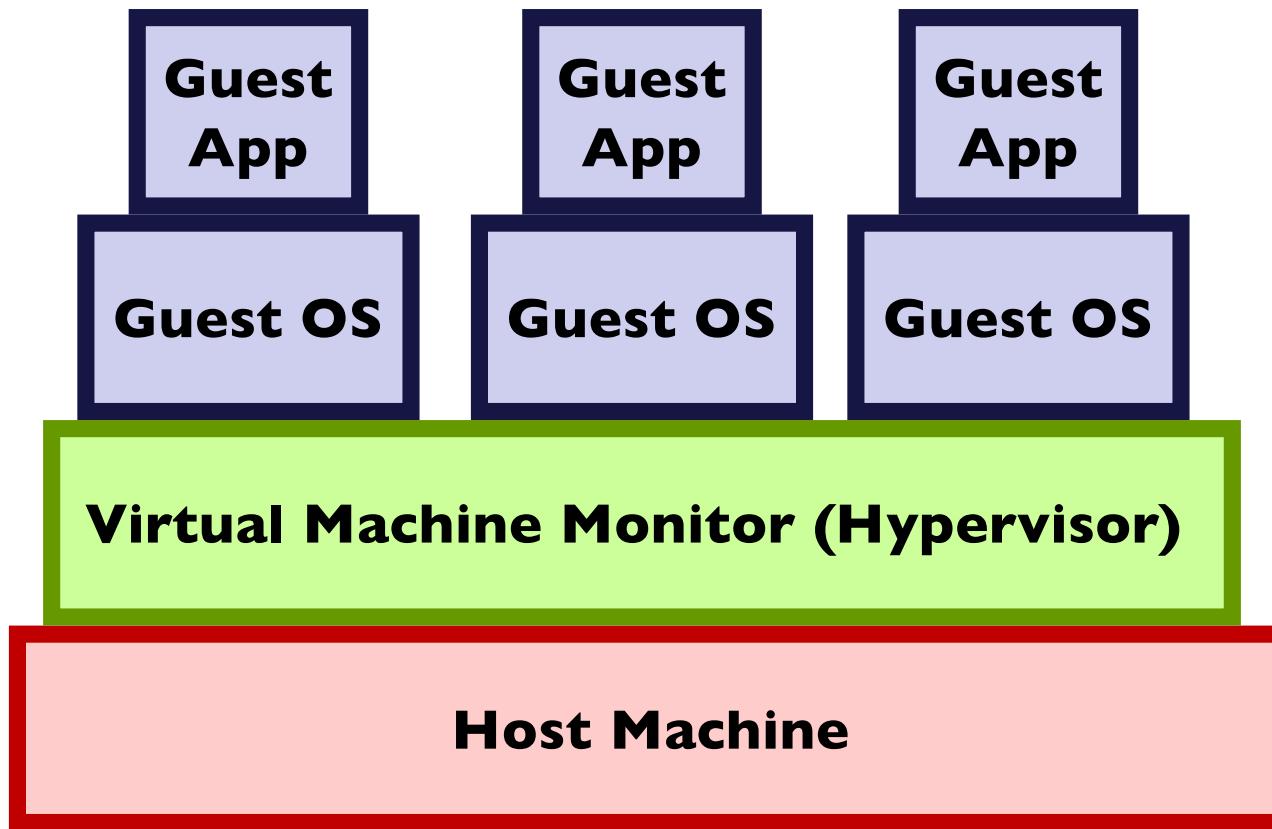
# SimOS (proto-VMware) arch.



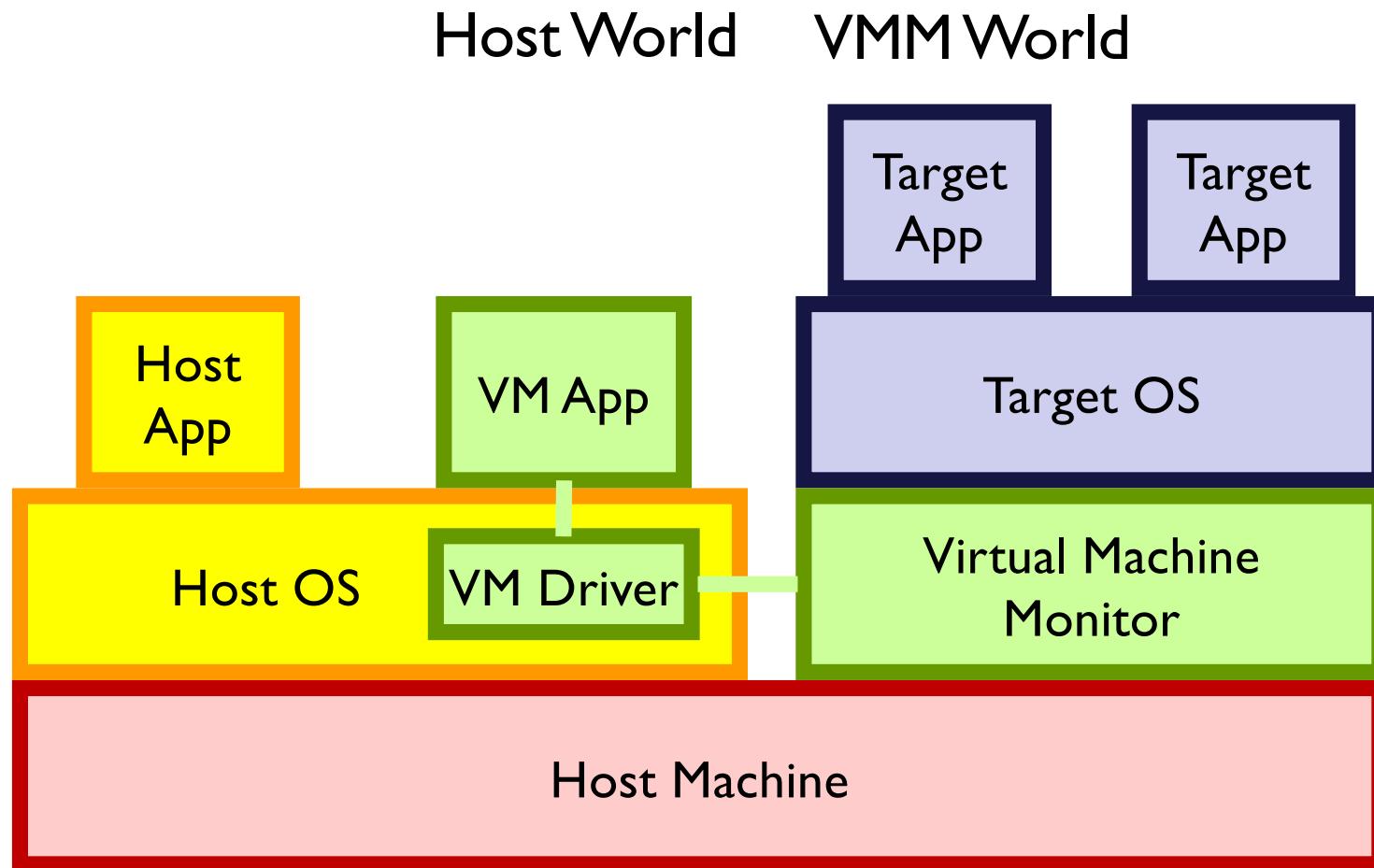
# SimOS (proto-VMware) arch.

- Similar to current emulation architectures
  - E.g., Bochs, QEMU
  - Present virtual hardware to OS
    - Differs from physical hardware
    - Maybe different CPU type (ISA), devices, etc.
  - Translate virtual hardware instructions to perform same operations on the actual physical hardware
    - Translation degrades performance
    - Much faster to directly execute instructions (no translation)

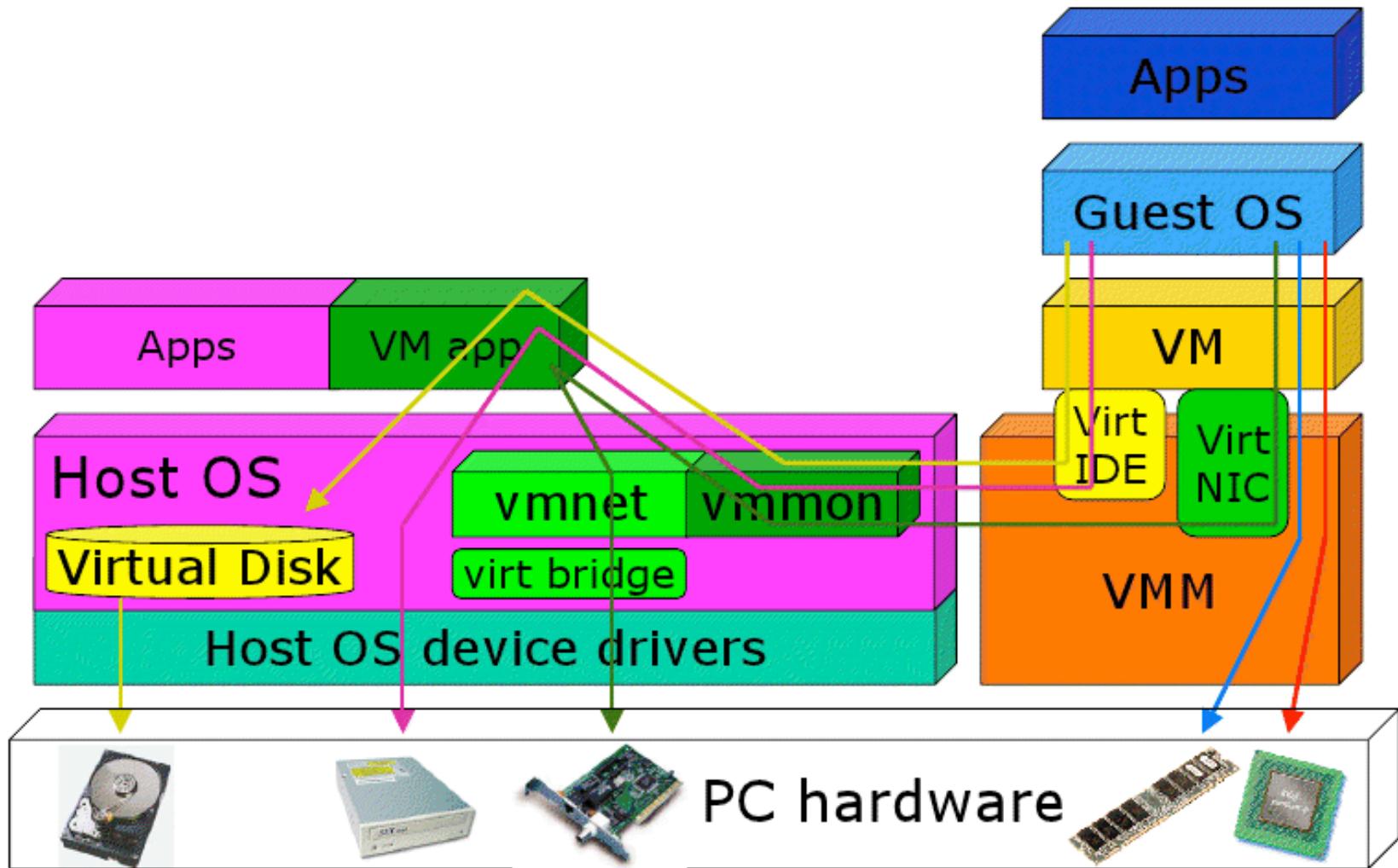
# Ideal virtual machine structure



# VMware Architecture



# Hosted VM I/O Virtualization



# Hypervisor Benefits

- **Code reuse**
  - Can run old operating systems + apps on new hardware
  - Original purpose of VMs by IBM in the 60s
- **Encapsulation**
  - Can put entire state of an “application” in one thing
  - Move it, restore it, copy it, etc
- **Isolation, security**
  - All interactions with hardware are mediated
  - Hypervisor can keep one VM from affecting another
  - Hypervisor cannot be corrupted by guest operating systems
- **Aids code development and debugging**

# Encapsulation

- Say I want to suspend/restore/migrate an application
- I decide to write the process mem + PCB to disk
- I reboot my kernel and restart the process
- Will this work?
  - No, application state is spread out in many places
  - Application might involve multiple processes
  - Applications have state in the kernel (lost on reboot)
  - (e.g. open files, locks, process ids, driver states, etc)
    - Called *residual dependencies*

# Encapsulation

- Virtual machines capture all of this state
- Can suspend/restore an application
  - On same machine between boots
  - On different machines
- Migration **much harder** for containers
  - Kernel is a moving target
    - Doesn't explicitly isolate and track dependent state
  - Every version might be different

# Security

- Can user processes corrupt the kernel?
  - Can overwrite logs
  - Overwrite kernel file
  - Can boot a new kernel
  - Exploit a bug in the system call interface
- Ok, so I'll use a hypervisor. Is my data any less vulnerable?
  - All the state in the guest is still vulnerable (file systems, etc)
- So what's the point?
  - Hypervisors can observe the guest OS
  - Security services in hypervisor are safe, makes **detection** easier

# Security

- Hypervisors buggy too, why trust them more than kernels?
  - Narrower interface to malicious code (no system calls)
  - No way for kernel to call into hypervisor
  - Smaller, (hopefully) less complex codebase
  - Should be fewer bugs
- Anything wrong with this argument?
  - Hypervisors are still complex
  - May be able to take over hypervisor via non-syscall interfaces
    - E.g. what if hypervisor is running IP-accessible services?
  - Paravirtualization (in Xen) may compromise this

# Virtual Machine challenges

- **Privilege modes** – guest OS think it runs in kernel mode
- **Memory management** – guest OS thinks it has access to physical memory
- **Performance** – two levels of context switching
- **Protection** – interference between guests may violate isolation

# Views of the CPU

- How is a process' view of the CPU different than OS's?
  - Kernel mode
  - Access to physical memory
  - Manipulation of page tables
  - Other “privileged instructions”
  - Turn off interrupts
  - Traps
- Keep these in mind when thinking about virtual machines

# Xen Approach

- Kind of the opposite approach of containers
  - Containers: start with OS, virtualize as needed
  - Xen: start with VMM, “para-virtualize”
- **Goals**
  - Performance isolation
  - Support many operating systems
  - Reduce performance overhead of virtualization

# Para-virtualization

- Full virtualization
  - Fool OS into thinking it has access to hardware
- Para-virtualization
  - Expose real and virtual resources to OS
- Why do we need para-virtualization?
  - Mostly because x86 architectures (used to) make full virtualization hard

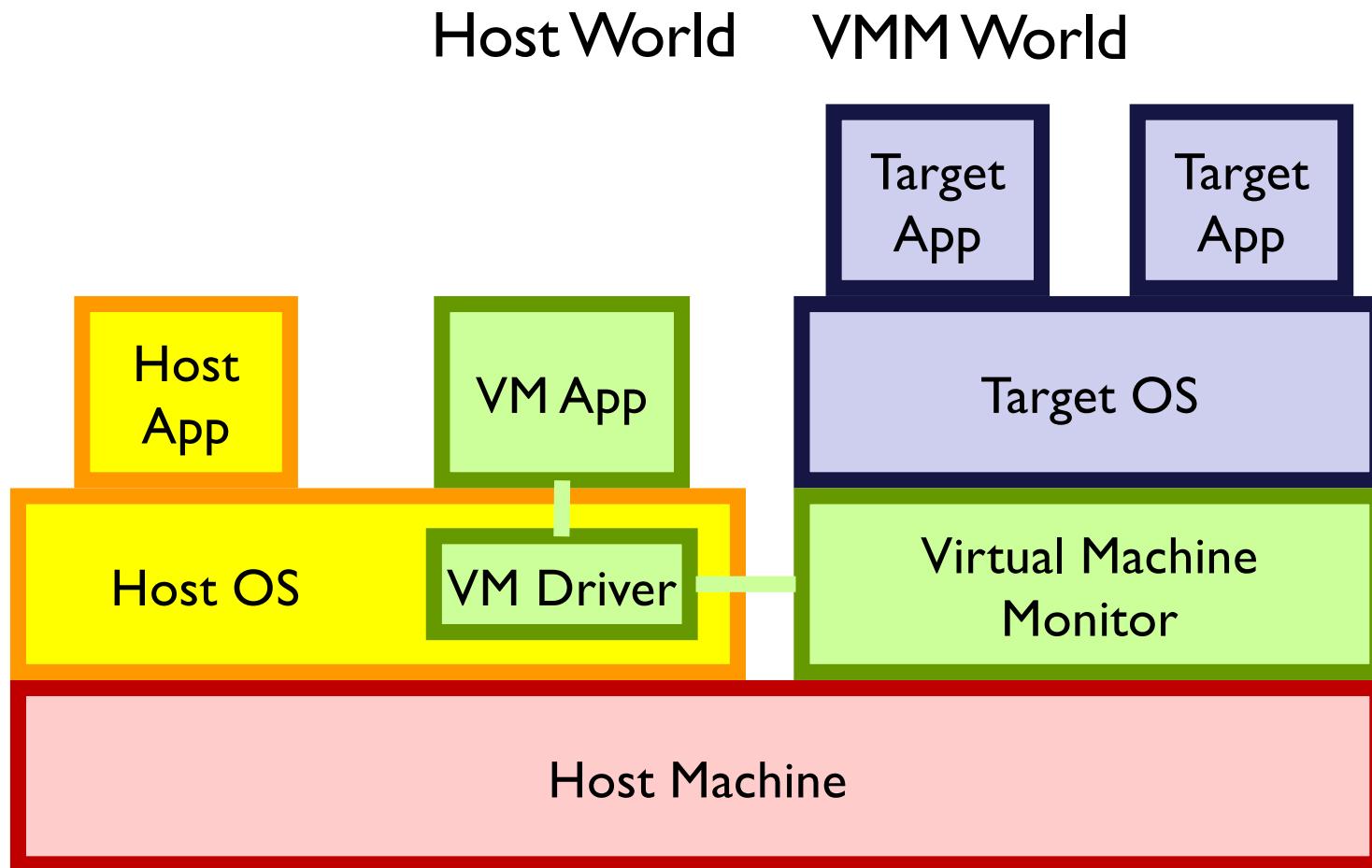
# Why para-virtualize?

- Previous limitations of x86
  - *Privileged instructions fail silently*
  - VMM must execute these instructions
  - Cannot rely on traps to VMM
  - How does VMware deal with this?
    - At run-time rewrite guest kernel binary
    - Insert traps into the VMM, when necessary

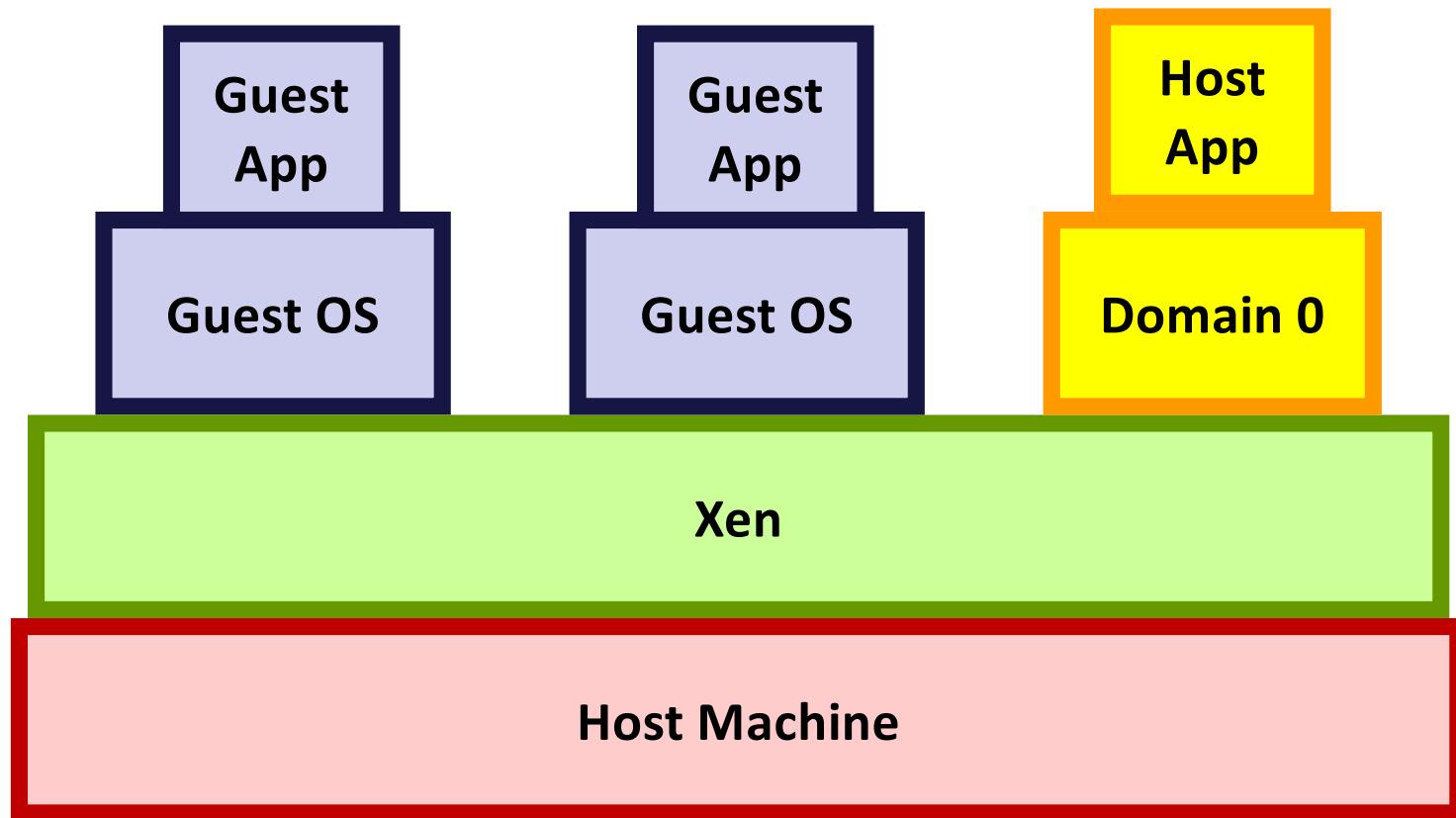
# Why para-virtualize?

- Previous limitations of x86
- Timing issues
  - May want to expose “real time” to OS
  - TCP time outs, RTT estimates
- Support for performance optimizations
  - Superpages (supporting huge pages in memory; increases TLB coverage)

# VMware architecture

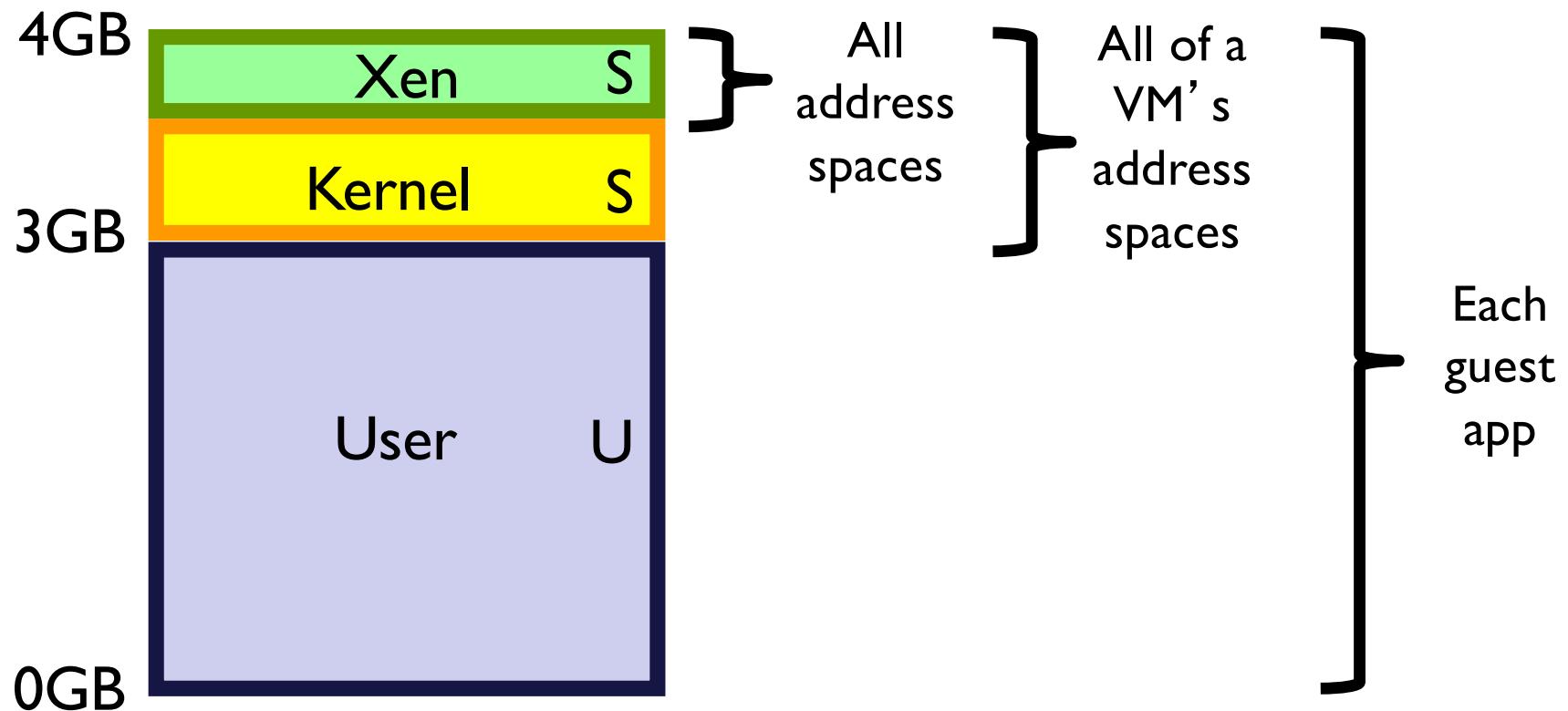


# Xen architecture



# X86\_32 address space

When are each set of virtual addresses valid?



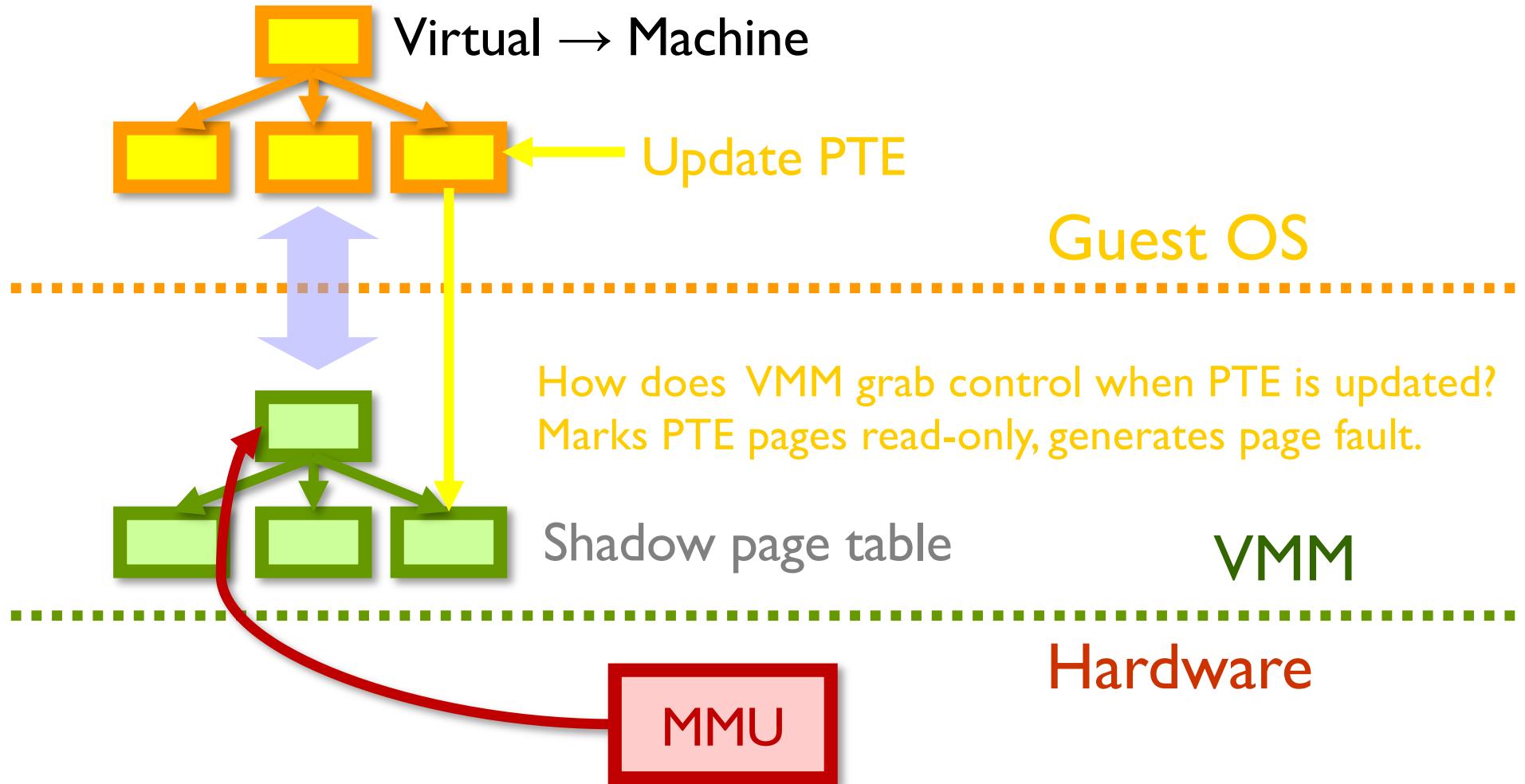
When does the hypervisor need to flush the TLB?

When a new guest VM or guest app needs to be run.

# Xen physical memory

- Allocated by hypervisor when VM is created
- Why can't we allow guests to update PTBR?
  - Might map virtual addrs to physical addrs they don't own
- VMware and Xen handled this differently
  - VMware maintains “shadow page tables”
  - Xen uses “hypercalls”
  - (Xen and VMware actually support both mechanisms now)

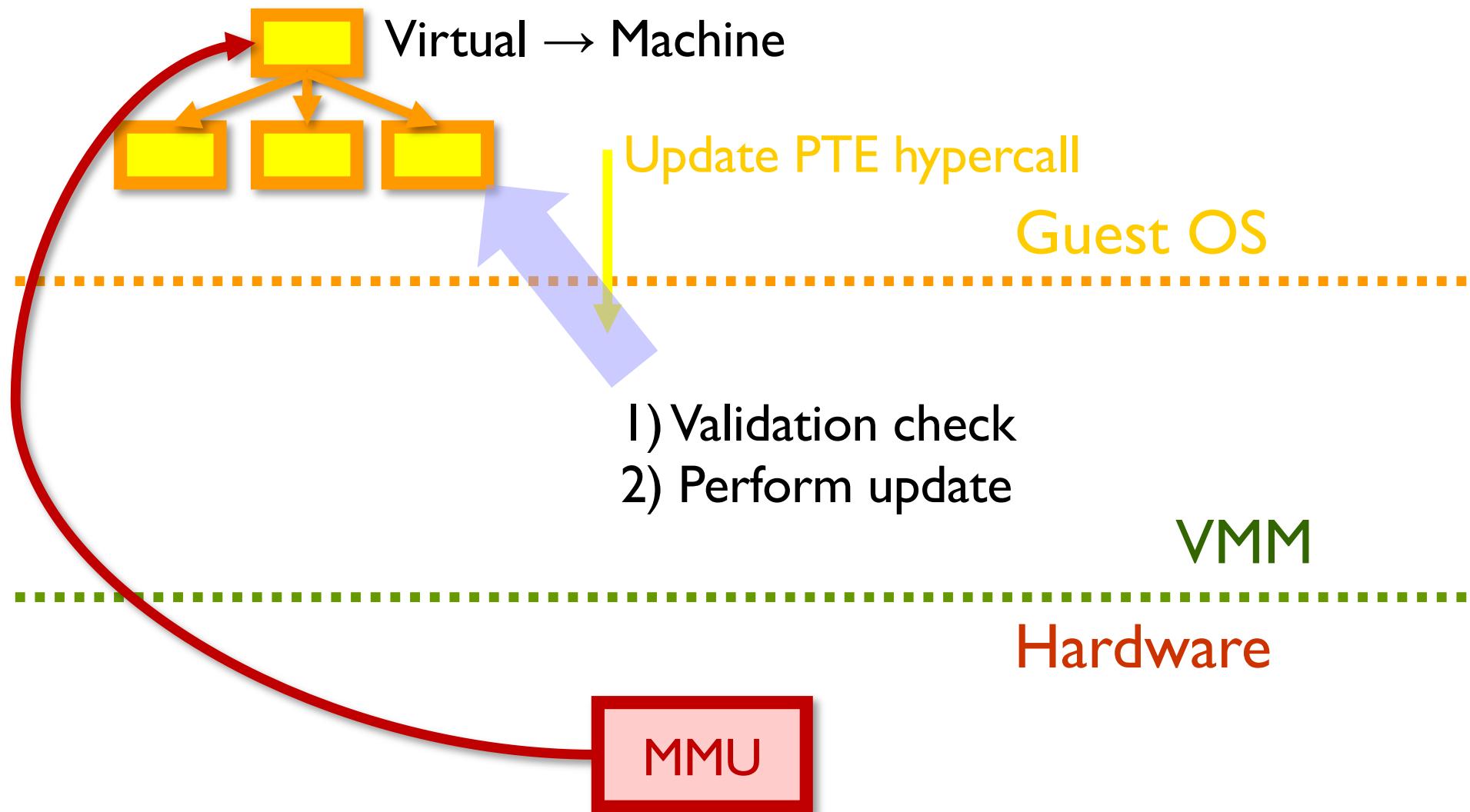
# VMware guest page tables



# Xen physical memory

- Guest OSes allocate and manage own PTs
  - “Hypervisor call” to change PT base register
  - Like a system call between guest OS and Xen
- Xen must validate PT updates before use
- What are the validation rules?
  1. Guest may only map phys. pages it owns
  2. PT pages may only be mapped read only

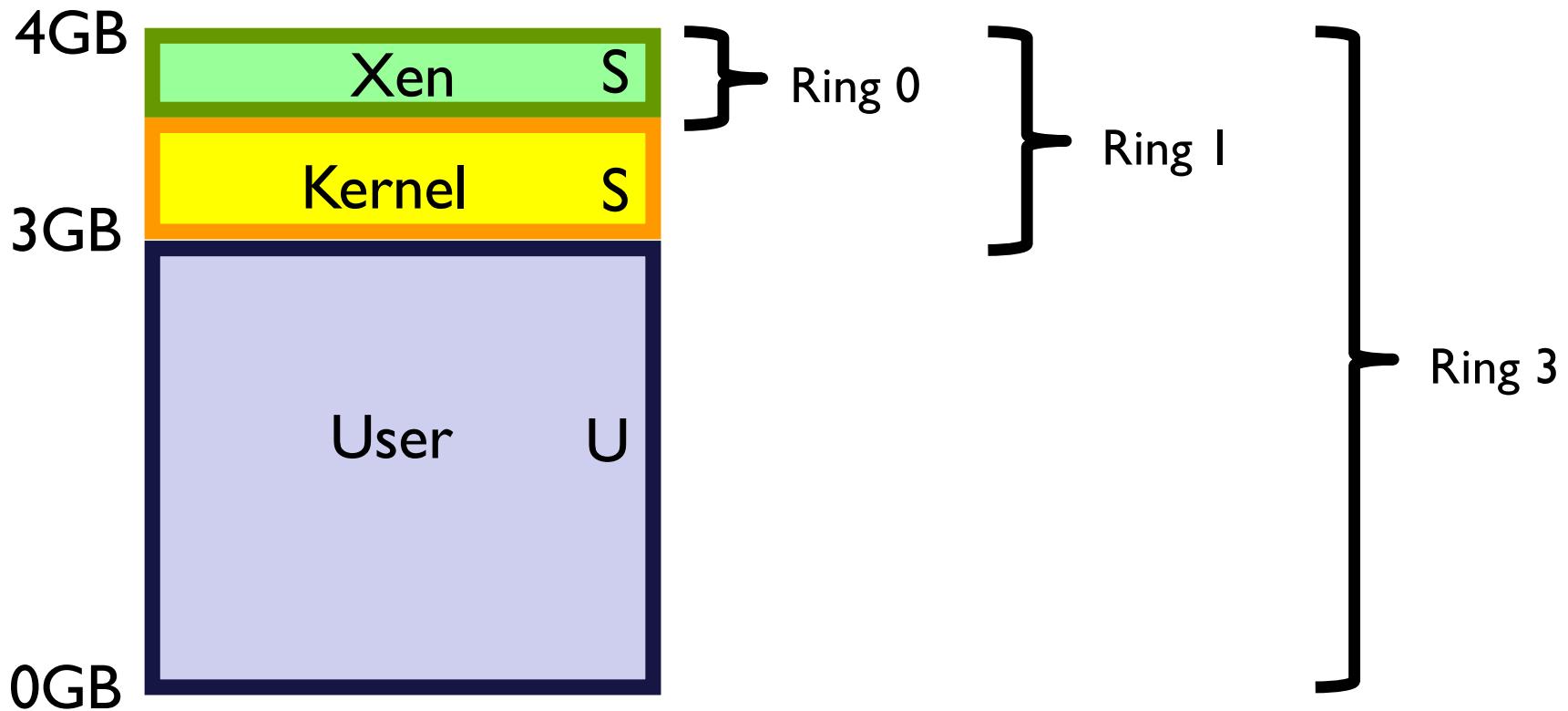
# Xen guest page tables



# Para-virtualized CPU

- Hypervisor runs at higher privilege than guest OS
- Why is having only two levels a problem?
  - Guest OSes must be protected from guest applications
  - Hypervisor must be protected from guest OS
- What do we do if we only have two privilege levels?
  - OS shares lower privilege level with guest applications
  - Run guest apps and guest OS in different address spaces
- Why would this be slow?
  - VMM must flush the TLB on all system calls, page faults

# X86\_32 address space



What does this assume?

Guest OS doesn't need ring 2; OS/2 used it (ring 0 for kernel, ring 1 for device drivers, ring 2 for user programs with I/O permissions, ring 3 for applications)

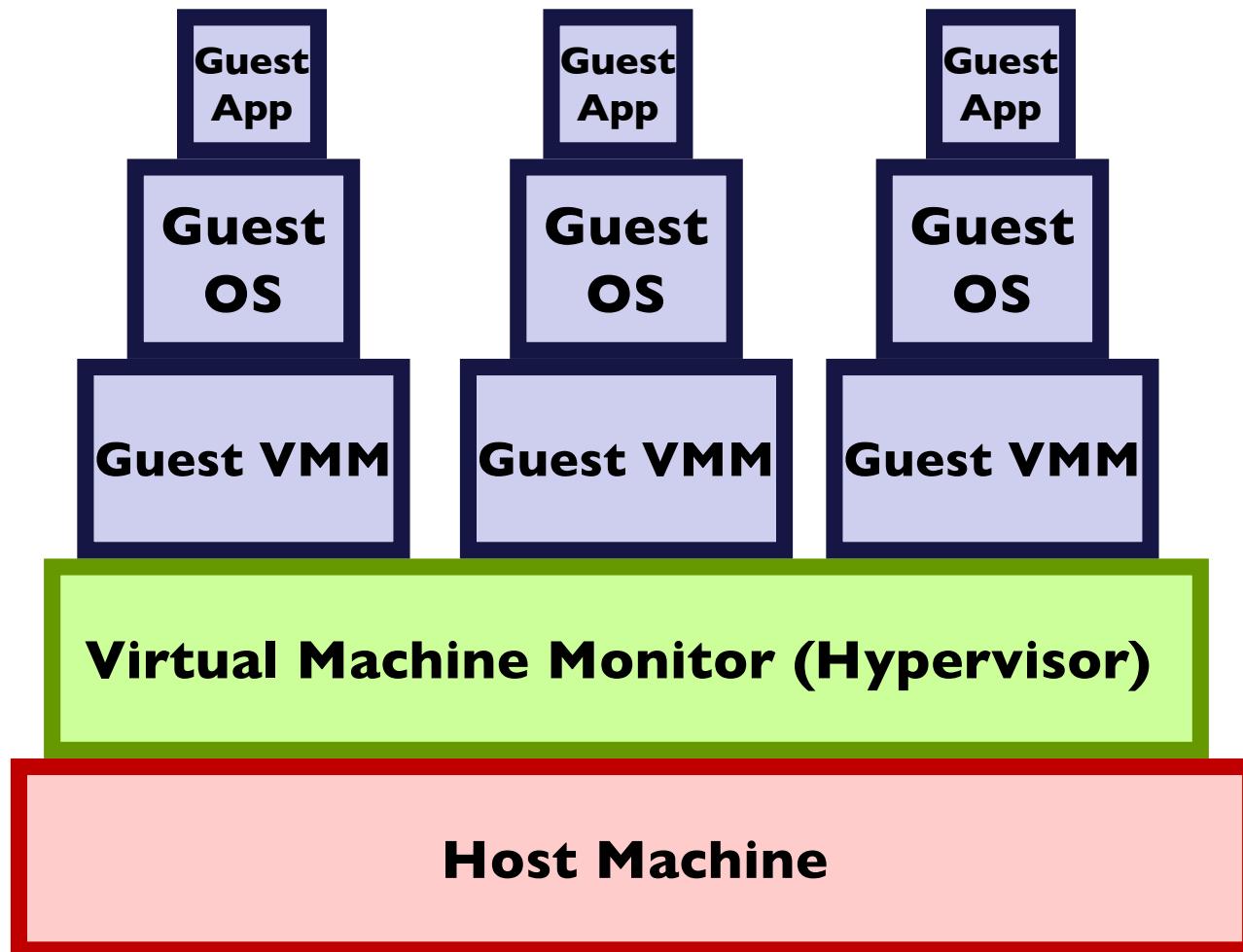
# Para-virtualized CPU

- Hypervisor runs at higher privilege than guest OS
  - Ring 0 → hypervisor, ring 1 → guest OS, ring 3 → guest apps
- Handling exceptions
  - Guest registers handlers with Xen (must modify guest OS!)
- System calls
  - Guests register "fast" handler with Xen
  - Xen validates handler, inserts in CPU's handler table
  - No need to go to ring 0 to execute
- What handler cannot be executed directly and why?
  - Page fault handler must read register CR2 (only allowed in ring 0)
  - CR2 is where the fault-generating address is stored

# Para-virtualization

- Pros
  - Better performance
  - Scales better than full virtualization
- Cons
  - OS needs (minor) changes
  - How well does it actually scale? Depends...
  - **Impure abstractions**
  - Is it important to provide good abstractions?
    - I say yes
    - Bad interfaces lead to code complexity, maintainability issues

# Nested VM structure?



# Live Migration of VMs

- Move VM from one physical server to another
  - Copy all state – local memory and local disk
  - VM continues running on new machine
- Possible because VM has few residual dependencies on hypervisor
  - Hypervisor decoupled from VM
  - Much harder to migrate processes and containers

# Live Migration of VMs

- How long to migrate a VM's state?
  - Minimum = time to transfer memory/disk state over network
    - Called a **stop-and-copy** migration
  - Assume 10Gbps network link (fully utilized) and 64GB memory
    - Stop application, and copy its memory state over
    - Would take 51.2 seconds to copy just the memory
    - Much longer in reality as links not fully utilized (not all are 10Gbps either)
    - Didn't even discuss disk state, which may be larger
- Causes significant application downtime
  - Not transparent
  - Will break network connections, which breaks networked applications

# Live Migration of VMs

- **Goals of Live VM Migration**

- Minimize application downtime and disruption
  - Reduce from many minutes to near 0
- Minimize downtime perceptible by external observer
  - Make downtime imperceptible to others on network
- Minimize migration time (and make it consistent)
  - Keep total migration time close to stop-and-copy (and minimize variability)
- Minimize network overhead
  - Keep the number of bytes sent similar to stop-and-copy (or even less)
  - Use optimizations similar your pager
    - Prevent sending pages multiple times
    - Don't send zero pages
    - Compress memory pages

# Migration Concerns

- **Maintaining network connectivity**
  - IP address remains the same and is routable at the new server
    - Changing IP would break existing network connections
  - Simple in a LAN (e.g., same ethernet segment)
    - Just send out a new ARP request – informs routers of new location
    - Few packets may be lost, but TCP will correct for it
  - Hard in a WAN (e.g., from UMass to Boston)
    - IP addresses are coupled with physical location
    - Solutions – protocols to change IPs with clients, use of proxies, use of virtual LANs
    - Also WAN bandwidth is much lower (at least over public Internet)
  - Migration over WAN still not common

# Migration Concerns

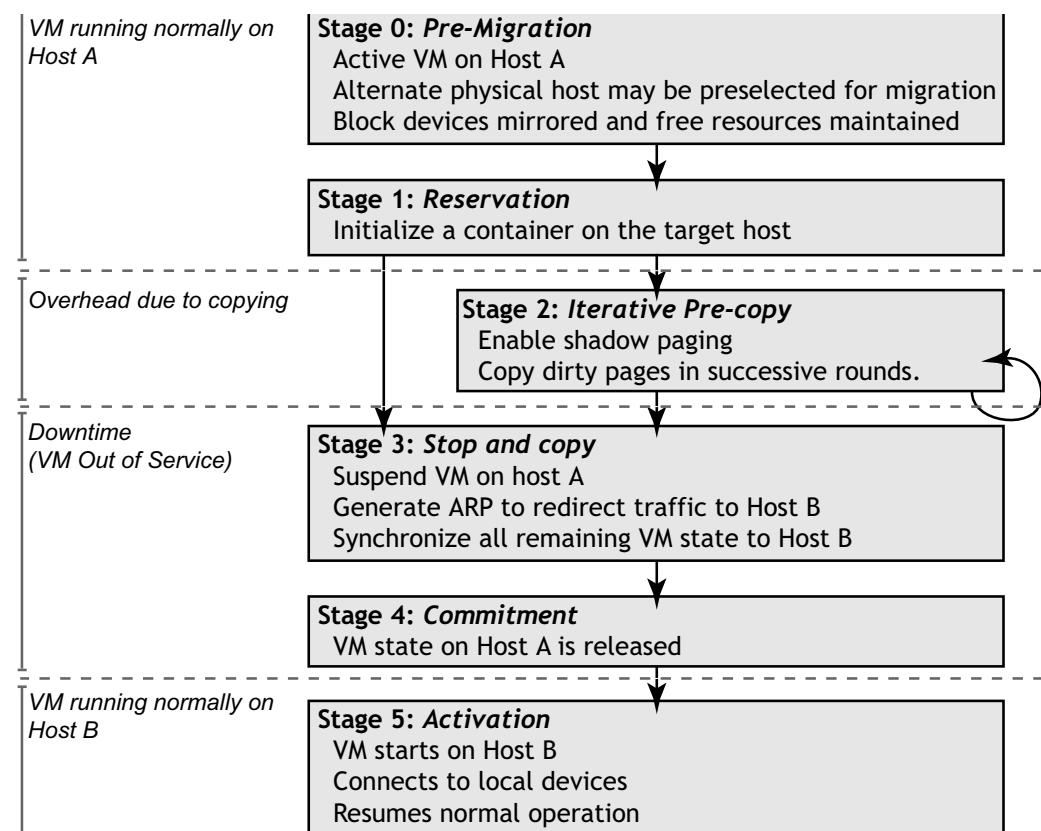
- **Migrating local disk state**
  - Much larger than memory state – many GBs
  - Could apply same techniques to disk as memory
    - Minimize application downtime
    - But total migration time would be large (still must copy all disk state)
  - More common
    - Use little or no local disk state
    - Remote mount all disk from a networked storage server
    - Storage server is accessible from all physical servers in LAN
  - Another reason why WAN migration is hard
    - Requires migrating disk state across WAN as well

# Migration Techniques

- **General Migration Phases**
  - *Push phase* – source VM continues running while memory pages are pushed across network to new physical server
  - *Stop-and-copy phase* – source VM is stopped, additional pages are copied to destination, and new VM at destination is started
  - *Pull phase* – new VM executes and if it accesses a page not yet copied are “faulted in” or pulled across the network from the source VM
- Can adjust number of pages sent in each phase
  - *Pre-copy migration* – push phase sends most pages
  - *Stop-and-copy migration* – stop-and-copy phase sends most pages
  - *Post-copy migration* – pull phase sends most pages

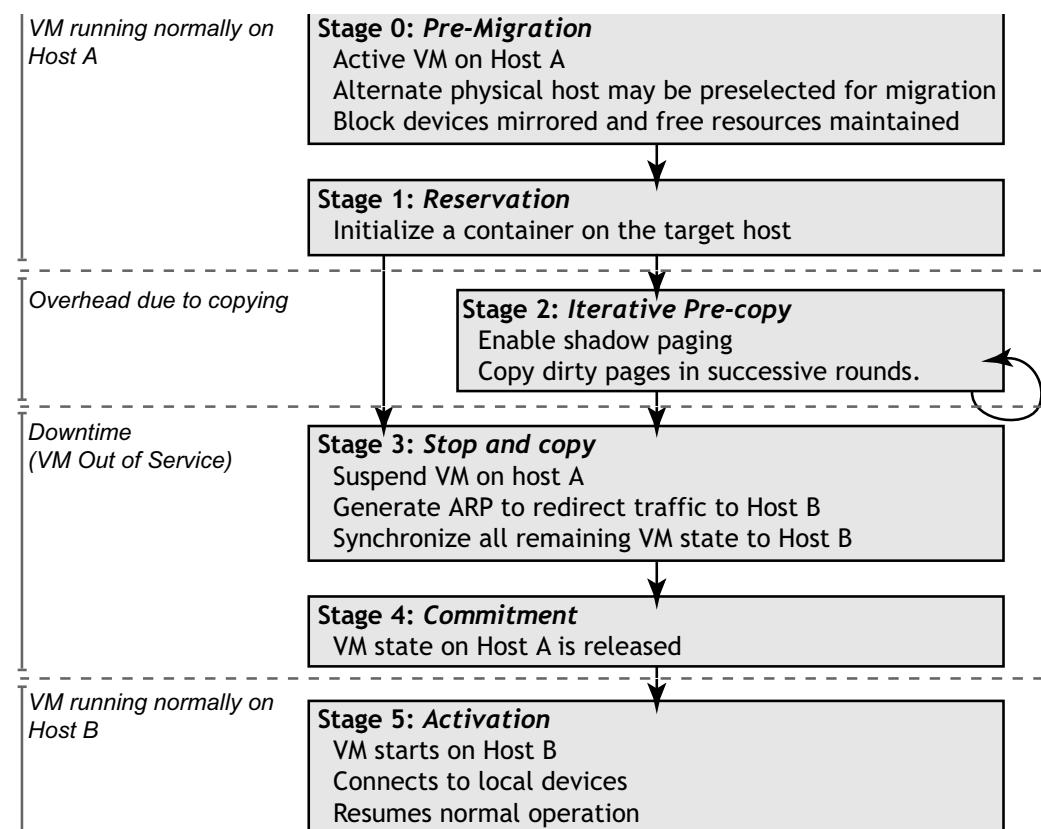
# Xen Live Migration

- Uses pre-copy technique
  - While VM is running...
    - ...copy all pages over
  - But pages may have changed...
    - ...so track those changes
    - Use your dirty bit
  - After you finish 1<sup>st</sup> round...
    - ...copy the newly dirty pages
  - But pages may have changed again...
    - ...so keep tracking changes
  - After you finish 2<sup>nd</sup> round...
    - ...copy dirty pages since 1<sup>st</sup> round
  - Size of pages copied in each round gets progressively smaller
    - Length of round gets smaller

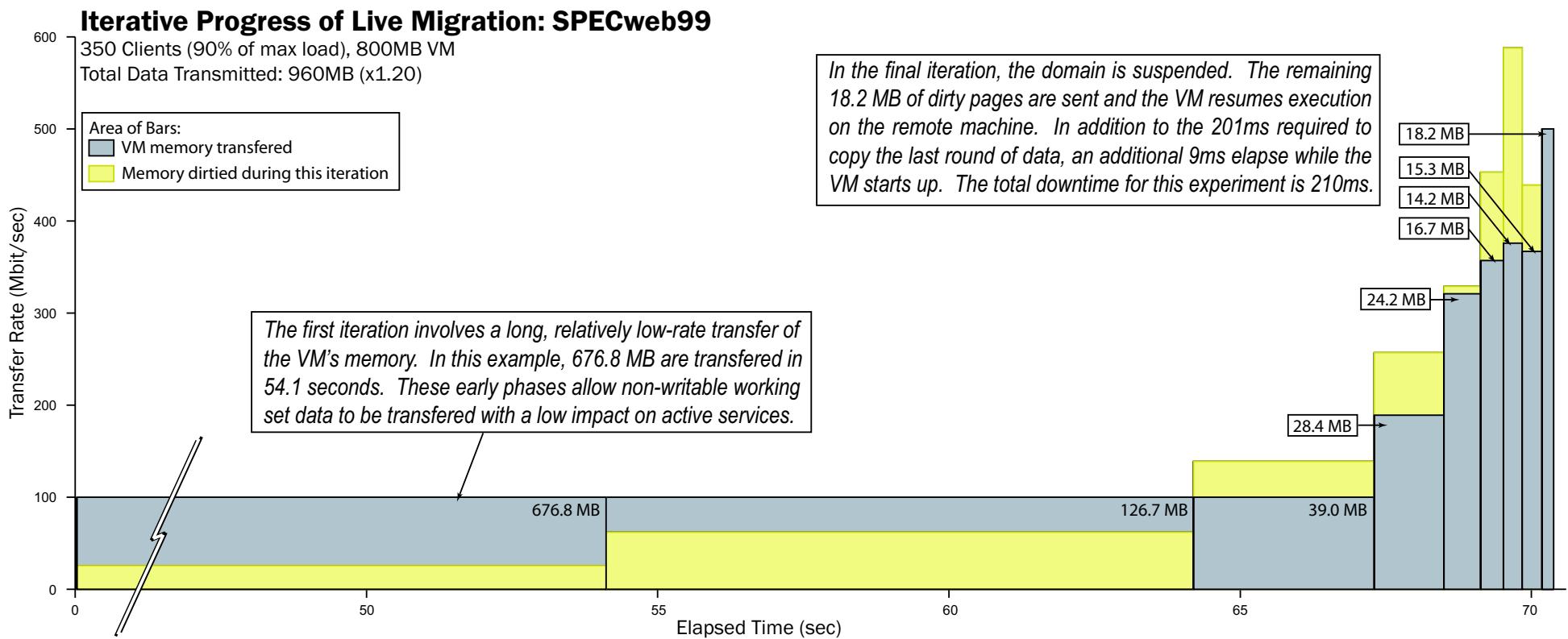


# Xen Live Migration

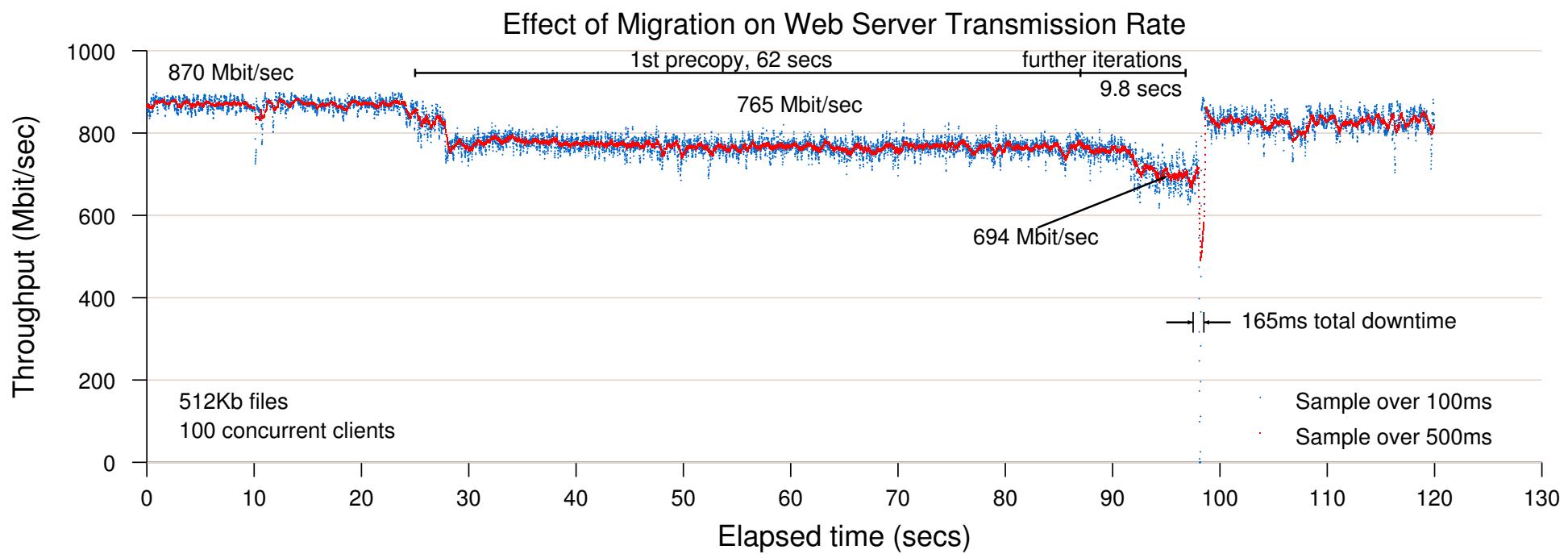
- Uses pre-copy technique
  - Once size stays the same per round
  - Stop VM and copy few remaining pages over
  - Release VM state on source host
  - Start VM on new host



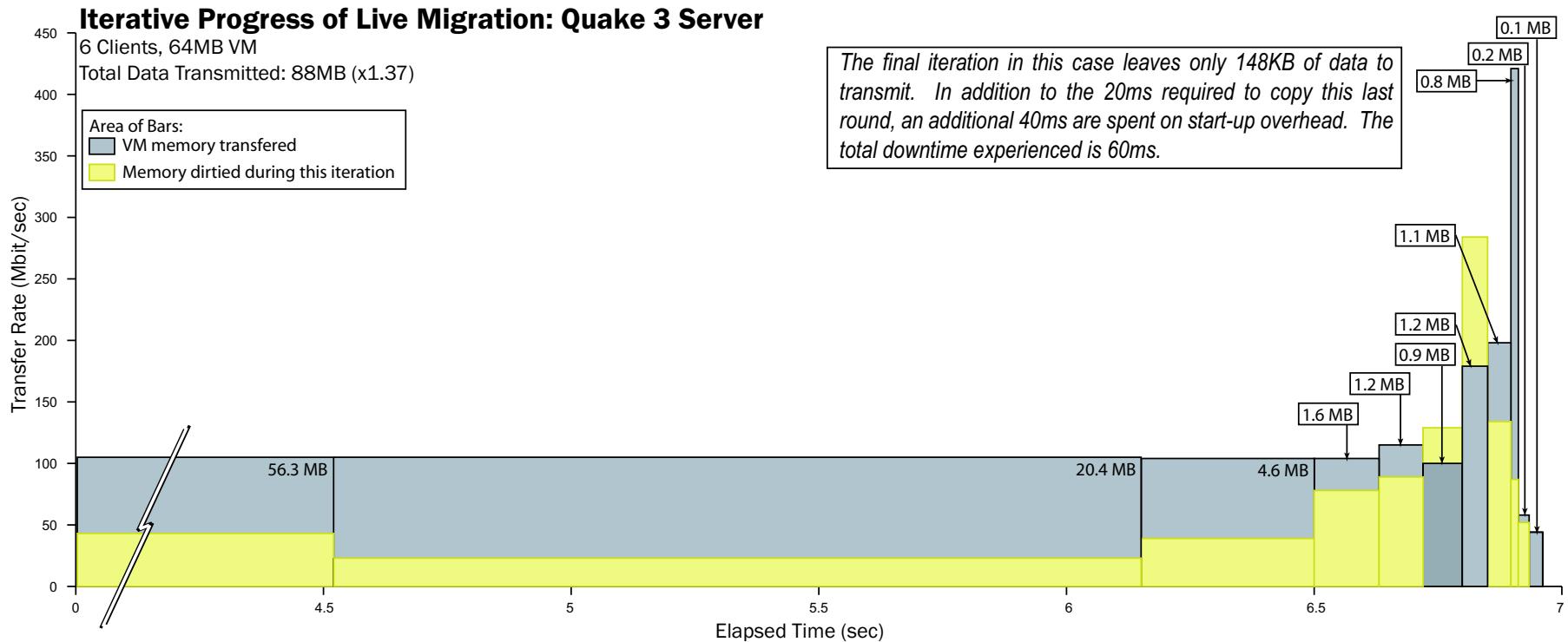
# Xen Live Migration



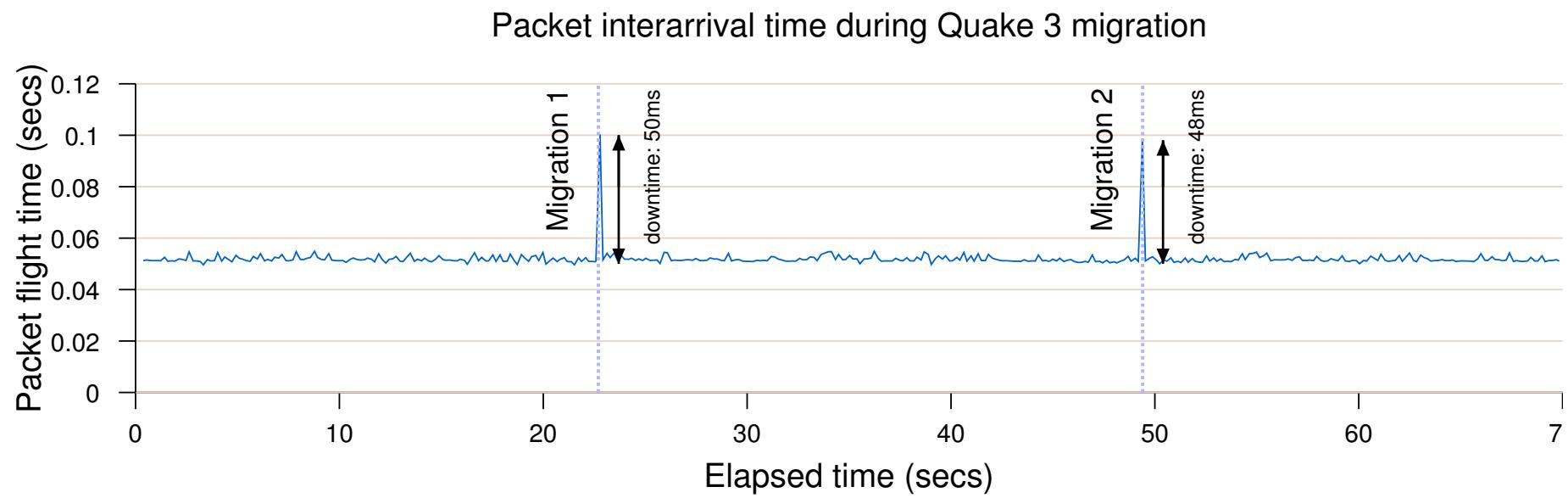
# Xen Live Migration



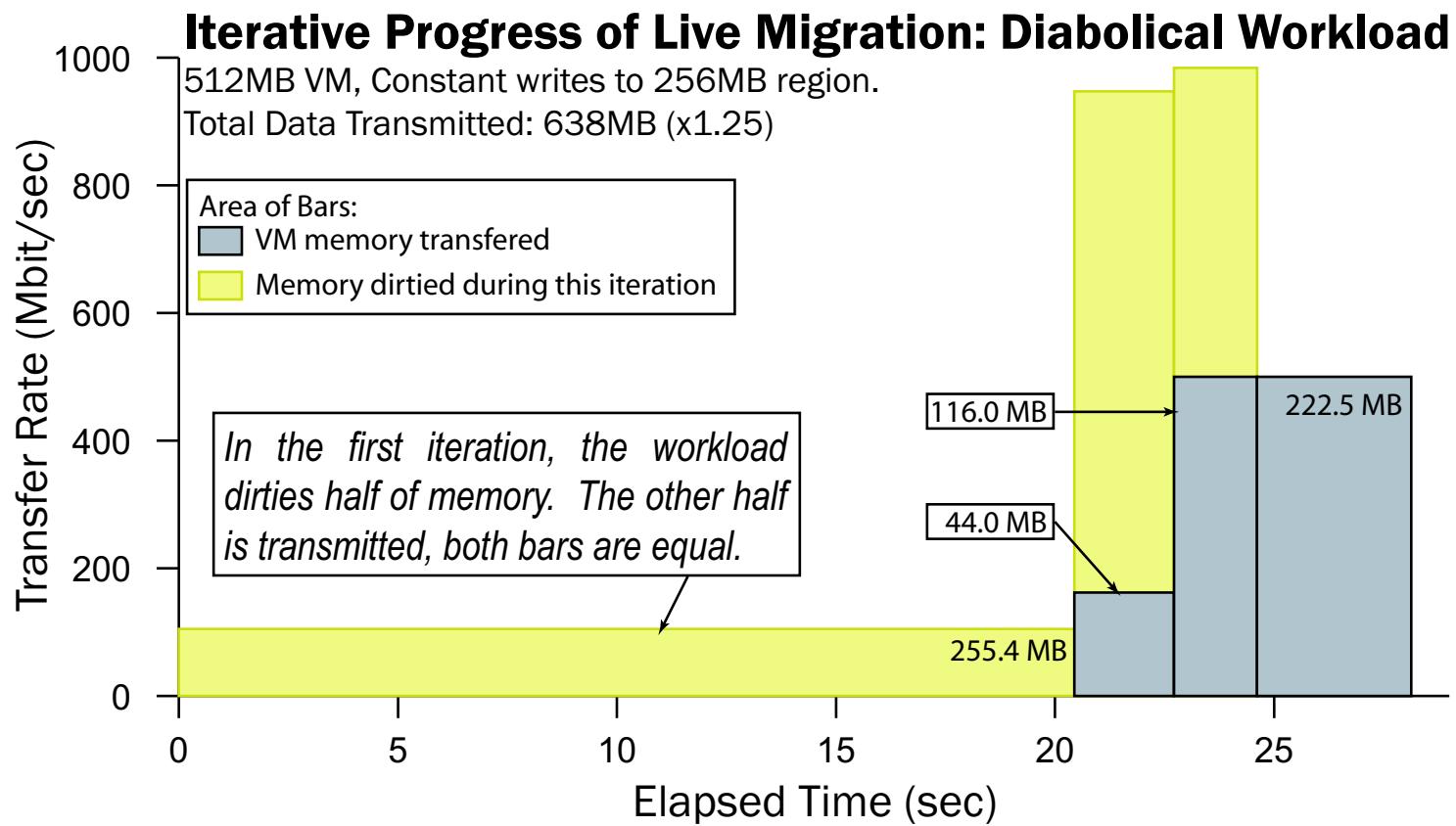
# Xen Live Migration



# Xen Live Migration



# Xen Live Migration



# Xen Live Migration

- Many optimizations possible
  - **Rate-limiting network usage**
    - Don't want to interfere with other applications on network
    - Adapt network bandwidth limit to reduce network interference on earlier rounds
    - Increase limit for later rounds that affect perceived downtime
  - **Stunning rogue processes**
    - Forcibly put processes within a VM in a wait if they have a high dirtying rate
    - Can only do with paravirtualization
  - **Free page cache pages**
    - Give these back to the hypervisor to reduce memory footprint
    - Not necessary for correctness
  - **Compress memory**
    - Don't send zero pages
    - Compress memory pages
  - **Shared memory pages on remote host**
    - Don't send memory pages that already exist on remote host
    - Libraries for common operating systems have similar pages

# Xen Live Migration

- **Advantages** of pre-copy technique
  - Small application downtime in best case - ~200ms
    - Very little impact on application performance
  - Imperceptible by external entities
    - Clients with open network connections
    - Users logged in via ssh
- **Disadvantages** of pre-copy technique
  - Increases total length of the migration
    - May be much longer than stop-and-copy
    - Must go through multiple rounds of copying pages
  - Migration performance highly variable
    - Depends on how write-intensive the VM is
    - Increases the size of each round
    - Worst case – perform multiple stop-and-copy migrations, and still have stop-and-copy downtime
  - High network traffic
    - Copy many more bytes over the network than with stop-and-copy

# Xen Live Migration

- **Advantages** of post-copy technique
  - Small application downtime in all cases – shorter than pre-copy
    - Always move smallest data possible and start application
  - Imperceptible by external entities
    - Clients with open network connections
    - Users logged in via ssh
  - Low network traffic
    - Transfer each page at most once; same network impact as stop-and-copy
- **Disadvantages** of post-copy technique
  - Highly variable application performance
    - Each page fault on a new page requires fetching page across network
  - Long migration time – based on when page faults occur
    - Will slow application down, increasing overall migration time

# Memory Resource Management in VMWare ESX Server

- **Example Situation**
  - Physical machine has 8GB of memory running two VMs
  - Each VM has 4GB allocated to it and is using 4GB
  - How do we add another VM with 2GB memory?
  - Must force both physical machines to reduce to 3GB memory
    - How do we force the OS to reduce its memory usage?
- Introduces **balloon driver**
  - Runs inside of each guest VM (no interface to outside)
  - Inflates by allocating pinned memory pages within VM
    - Forces the use of swap
  - Deflates by deallocating pinned memory pages

# Memory Resource Management in VMWare ESX Server

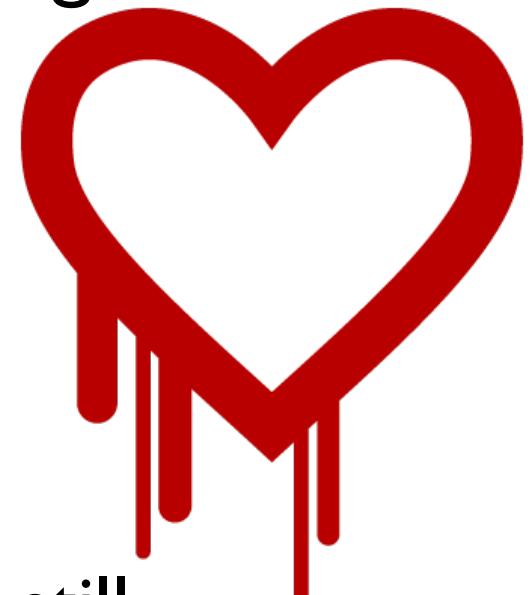
- **Share-based allocation of memory**
  - Similar to Lottery Scheduling (by same author)
  - Each VM given some shares
    - Entitled to fraction of memory equal to their fraction of total shares
    - Shares are equally distributed across pages (VMs pay page per share)
  - Normal policy is to give pages to VMs with the highest shares per page
- **Normal policy enables hoarding**
  - Solve using *idle memory tax*
  - Charge VMs more for idle pages than for active pages
    - Changes shares per page ratio
  - Causes idle memory to be reclaimed first

# Memory Resource Management in VMWare ESX Server

- **Content-based page sharing**
  - Only keep one copy of multiple pages with same value
  - Doesn't matter how page contents was generated
    - Could be in different VMs
  - Prohibitively expensive to compare memory for sharing opportunities
    - $O(n^2)$  – compare every page to every other page
- **Optimization**
  - Hash page contents – convert to small value that identifies contents
  - Use hash in lookup table – stores other pages with same hash
  - Use copy-on-write techniques if a match is found

# OS Security

- Motivating example from a few years ago
  - Heartbleed bug
    - <http://heartbleed.com/>
  - Exploit of OpenSSL (everyone uses it)
    - ~66% of websites exposed
    - Present for around 2 years
  - Even after patching code websites were still exposed



# Asymmetric Keys

- Two different keys – **a** and **b**
  - Encrypt with **a**, can decrypt with **b**
  - Or, encrypt with **b**, can decrypt with **a**
- **Public/private key-pairs**
  - Make **a** public (post on web), but keep **b** private
  - Anyone can encrypt message using **a** and send to you
  - Only you can decrypt it, because only you have **b**

# Asymmetric Keys

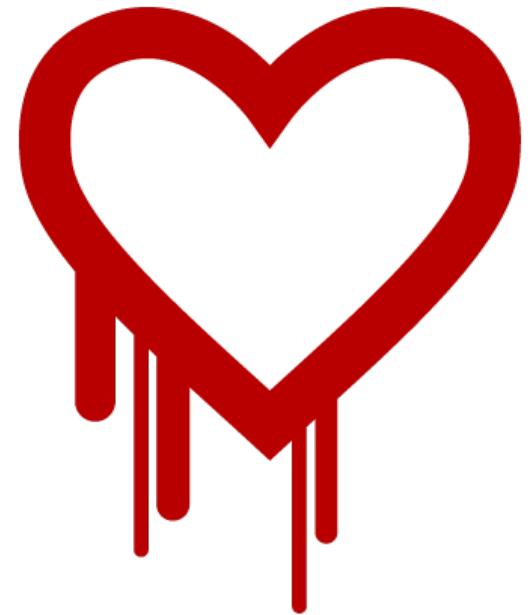
- Two different keys – **a** and **b**
  - Encrypt with **a**, can decrypt with **b**
  - Or, encrypt with **b**, can decrypt with **a**
- **Digital Signatures**
  - Encrypt message with your private key **b**
  - Anyone can decrypt with your public key **a**
  - Everyone can be sure you sent the message
    - Authenticated – you sent it
    - Non-repudiable – you cannot deny sending it
    - Integrity – the message is not corrupted

# Problem

- **OpenSSL Heartbeat**
  - Client sends message of length x
  - Server echoes message back
- **Bug**
  - Client says it is sending message of length 65k
    - Only sends message of size <65k (e.g., “foo”)
    - Fills rest of buffer with whatever is on the heap
  - Things possibly/probably on the heap
    - Username, encrypted password, session key, and OpenSSL private key (if it is near the first request to the server)
    - Also instant messages, emails and business critical documents

# Impact of Heartbleed

- I needed to do my taxes on TurboTax
  - Before April 15<sup>th</sup>
- Turbotax said they had patched bug
- Was it ok?



# Authenticating SSL public keys

- I want to send my credit card # to Turbotax
  - No one but Turbotax should see my message
  - I want to make sure it is really Turbotax
  - Turbotax wants to know it's really me
- **Step 1:** authenticate Turbotax to me
- **Step 2:** authenticate me to Turbotax

# Step I: authenticating Turbotax

- Turbotax has a public key
- How do you learn this public key?
  - **Web solution:** someone else vouches for key
  - Use a certificate authority
    - E.g. Verisign
- Turbotax sends you their public key
  - Public key is digitally signed by Verisign
  - Called a certificate --- has an expiration

{“turbotax’s public key is Turbotax-public”}<sub>verisign-private</sub>

# Step I: authenticating Turbotax

- Turbotax has a public key

{“turbotax’ s public key is Turbotax-public”}<sub>verisign-private</sub>

- Decrypt using Verisign’ s public key
  - I see that Verisign vouches for Turbotax-public
- Once talking to Turbotax
  - Establish session key

{“use session key K-sec”}<sub>Turbotax-public</sub>

# Step I: authenticating Turbotax

- Once talking to Turbotax, establish session key
- Any problems with this?
  - How do you know Verisign's public key?
  - Hard-coded into Firefox/IE binary
- How to trust Firefox binary?
  - Arrives in sealed CD package or pre-installed
  - Could download from the Internet
- Why trust this?
  - At some point you have to trust something

# Step 2: authenticate you to Turbotax

- Turbotax must know it is talking to you
- Use a password
  - Can be sent using session key from step 1
  - Only Turbotax can see password  
 $\{\text{"user david, password pass"}\}_{K-\text{sec}}$
  - Turbotax knows message came from session key sender
- Turbotax decrypts to check password
  - Stored in memory on heap (not on disk)

# Is Turbotax ok?

- What could happen?
  - Server's private keys could be in heap
  - User session keys and cookies could be in heap
  - Verisign root private keys could have been in heap on Verisign server
    - Enable anyone to “vouch” for a public key
    - Could enable man-in-the-middle attacks
    - I create a website and act like TurboTax
      - I send you my own fake public key
      - Sign it with the Verisign private key I stole

# What was the result?

- Shouldn't have logged into any websites using https until they updated their certificate and updated OpenSSL software
  - Certificates should not be valid before April 7<sup>th</sup>, 2014
  - Everyone needed new public/private key-pair
- After you log on, change your password
- Evidence bots had been exploited bug since at least November 2013 and possibly earlier

# OS security

- Access control
- Buffer overflows
- Hidden channels
- Trojan horses
- Viruses and other malware

# Security abstractions

- **What HW primitives exist for access control?**
  - Processor mode bit (kernel/user mode)
  - Protected instructions (I/O instructions, halt)
  - Protected data (page tables, interrupt vector)
- **Want to build two abstractions on these:**
  - Identity (authentication)
    - Who are you?
  - Security policy (authorization)
    - What are you allowed to do?

# Authentication: who are you?

- Prove your identity to the OS
- Many ways to authenticate
  - Passwords
  - Physical tokens
  - Biometrics

# Password authentication

- **Password**
  - Shared secret between you and the OS
- **Several weaknesses**
  - People are the weakest link
- **How should we store passwords?**
  - Cryptographic hashes (MD5, SHA)
  - Hashes are one-way functions
  - Check that  $\text{hash}(\text{input}) = \text{hash}(\text{password})$

# Weak passwords

- Consider an 8-character password
- $256^8$  possible passwords
  - $2^{64} \sim 16 * \text{billion} * \text{billion}$
- If you only choose lower-case letters
  - $26^8 \sim 200 \text{ billion}$
- If you choose a word from the dictionary
  - 320,000 words in Webster's unabridged
  - 1000/second → find a password in 5 minutes
- Researchers ran an attack on Michigan passwords
  - Recovered thousands; some of the most popular: "beer", "hockey"
  - Similar result at Berkeley CS in the 80s: "gandolph"

# Physical token authentication

- **Require something physical**
  - E.g. a ticket to a dance performance
- **What if your token is stolen/forged?**
  - Use a physical token + password
  - E.g. your ATM card + PIN

# Biometric authentication

- **Essentially a physical token**
  - One that should be hard to steal/forge
- **Examples**
  - Thumbprint, retina scan, signature
- **Most have accuracy problems**
  - Gummy bears?
  - False positives (accept invalid person)
  - False negatives (reject valid person)

# Real-world authentication

- How to authenticate to your credit card company?
  - Give them your social security number
  - They ask over the phone and check
- How is this vulnerable?
  - Company has my SSN, so they can pretend to be me
  - (or someone who breaks into them can pretend to be me)
  - Phone line snoop can pretend to be me

# Authorization: what can you do?

- **Fundamental structure**

- Access control matrix
- Rows = authentication domains
- Columns = objects

	<b>File 1</b>	<b>File 2</b>	<b>File 3</b>
User 1	RW	RW	RW
User 2	--	R	RW

- Two approaches to storing data:
  - **Access control lists** and **capabilities**

# Access control lists

- Standard for file systems
- **At each object**
  - Store who can access the object
  - Store how the object can be accessed
- **On each access**
  - Check that the user has proper permissions
- **Use groups to make things more convenient**
  - E.g. mike, david, and tilman are all in group “faculty”

# Access control lists

- **How can you defeat ACL authorization?**
  - Villain convinces the OS it is someone else
- **Example**
  - Sendmail runs as root (full privileges)
  - Attacker compromises sendmail process
  - Can now run arbitrary code under root ID
  - This is what you are doing in Assignment 3!
- **What if you get the ACL wrong?**

# Capabilities

- **At each user**
  - Store a list of objects they may access
  - Store how those objects may be accessed
- **Example**
  - At user2, store <file2, r: file3, rw>
- **On each access**
  - User presents capability
  - Capability proves they are authorized

# Capabilities

- **Possession of capability provides authorization**
  - Like car keys
  - If you have the door key, you can get in the car
  - If you have the ignition key, you can drive it
- **How can you defeat a capability system?**
  - Forge an authorized user's capabilities
  - Like making a copy of someone's car keys

# Revocation

- **How to revoke permissions with an ACL?**
  - Just delete/change the ACL entry
- **How to revoke permissions with capabilities?**
  - Not nearly as easy
  - Could change the car door's lock
  - This revokes access for everyone with the key

# Dealing with security

- Principle of least privilege
  - Users get minimum privilege to do their work
- Defense in depth
  - Create multiple levels
  - Lower-levels monitor higher ones
- Minimize trusted computing base
  - Every system has a trusted part
  - (e.g. the thing that checks the ACL)
  - This part has special privileges
  - It cannot be subject to the security mechanism

# Common attacks

- Hidden channels
- Buffer overflow
- Trojan horse

# Hidden channels

- Get system to communicate in unintended ways
- Example: tenex (supposedly secure OS)
  - Created a team to break in
  - Team had all passwords within 48 hours ... oops.

Password checker

```
for (i=0; i<8; i++) {  
    if (input[i] != password[i]) {  
        break;  
    }  
}
```

- Goal: require  $256^8$  tries to see if password is right

# Hidden channels: tenex

Password checker

```
for (i=0; i<8; i++) {  
    if (input[i] != password[i]) {  
        break;  
    }  
}
```

- How to break? (hint: virt mem faults are visible)
  - Specially arrange the input's layout in memory
  - Force a page fault if second character is read
  - If you get a fault, the first character was right
  - Do again for third, fourth, ... eighth character
- Can check the password in  $256^8$  tries

# Buffer overflow

- **Suppose you have an on-stack buffer**
  - You read input into the buffer
  - You don't check the length of the input
- **A villain's long input can corrupt the stack**
  - If they're smart, they can corrupt the return value
  - Allows villain to jump to their own code
- **First Internet worm did this**
  - Creator was a Cornell grad student, Robert Morris
  - Son of a famous computer security guru
  - He is now a professor at MIT

# Trojan horse

- **Basic idea**
  - Give somebody something useful
  - Make it do something evil
- **Examples**
  - Replace login with something that emails passwords
  - Send a Word document with a malicious macro
  - Download Kazaa and install spyware
  - Install a Sony CD and have it install a buggy DRM root-kit

# Why security is so hard

- Ken Thompson's self-replicating code
  - Goal: put a backdoor into login
    - Allow "ken" to login as root w/o password
1. Make it possible (easy)
  2. Hide it (tricky)

# Login backdoor

- Step 1: modify login.c
  - (code A) if (name == “ken”) login as root
  - **This is obvious so how do we hide it?**
- Step 2: modify C compiler
  - (code B) if (compiling login.c) compile A into binary
  - Removes code A from login.c, keeps backdoor
  - **This is now obvious in the compiler, how do we hide it?**
- Step 3: distribute a buggy C compiler binary
  - (code C) if (compiling C compiler) compile code B into binary
  - No trace of attack in any source code (only in C compiler’s binary)
- Upshot: everything is bootstrapped, everything trusts something

# Failure-Oblivious Computing

- Rinard, Cadar, Dumitran, Roy, Leu, Beebee
-  Main Idea:
  - Detect invalid memory accesses
    - i.e, Invalid array access or pointer access
    - Not needed in Java. Why?
  - Rather than terminate (w/seg faults), simply discard invalid writes, and make up values for reads

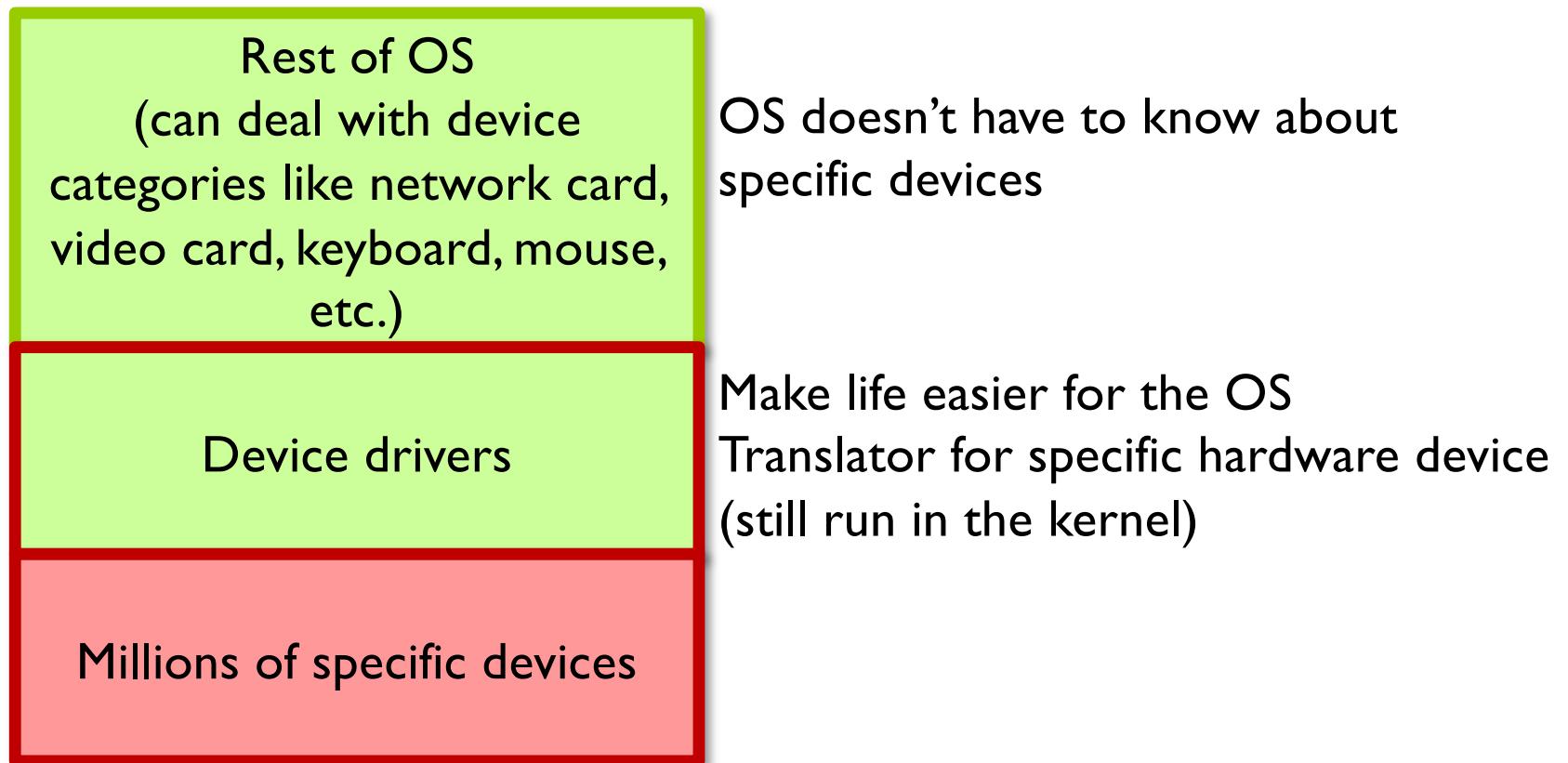
# Big Picture

- The main job of an OS is to facilitate the sharing of resources in a computer
- What resources have we discussed so far?
  - CPU: Threads, processes
    - Challenges?
    - Synchronization, protection, fairness, preventing deadlock
  - Memory: virtual address spaces, paging (related to security)
    - Challenges?
    - Thrashing, protection, page eviction fairness
  - What resources are missing from this list?
    - IO devices (network, disk, etc)

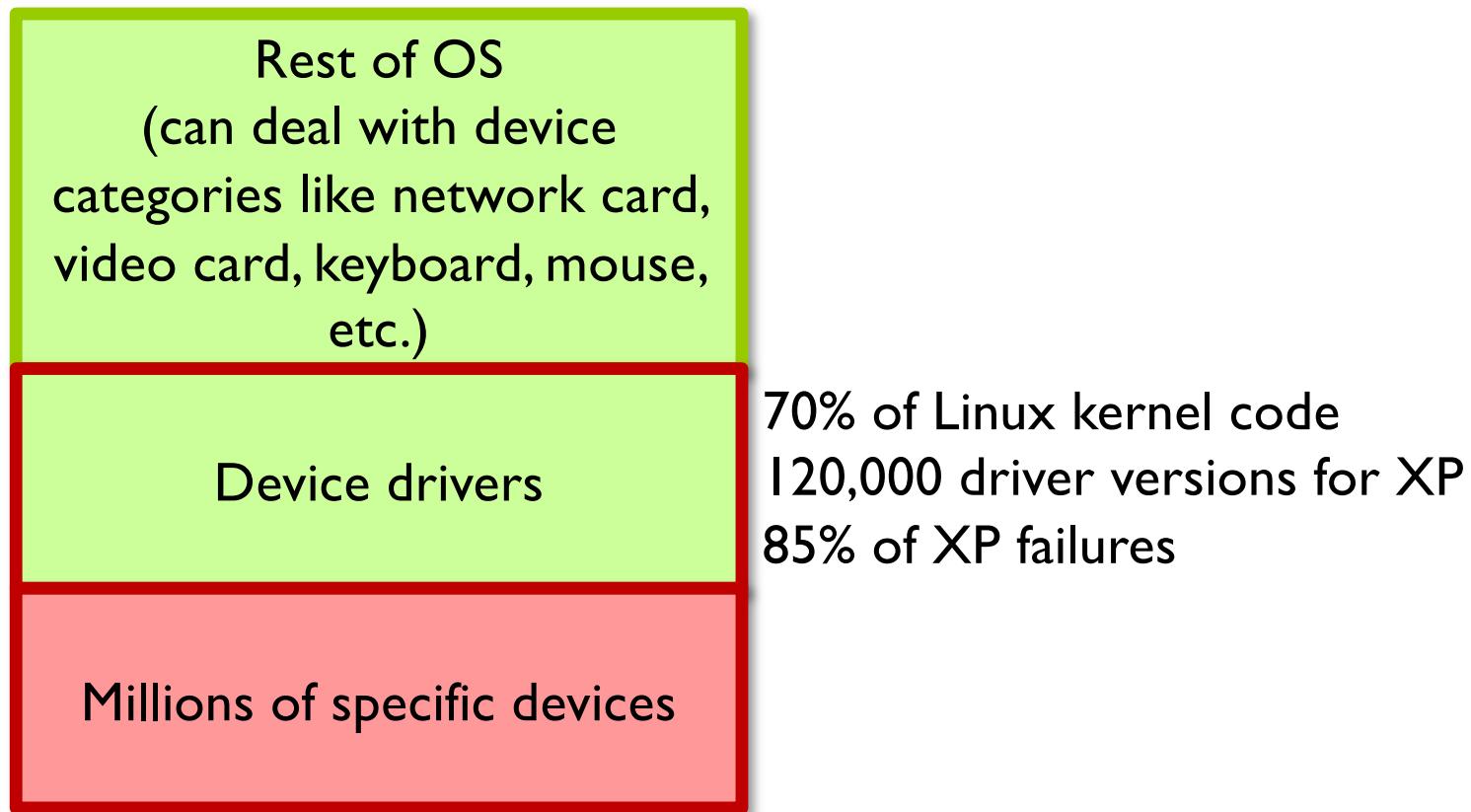
# Device independence

- Problem: many different devices
  - Also interfaces, controllers, etc
  - Each has their own custom interface
- How do we manage this diversity?
  - Add an abstraction **inside** the OS

# Device drivers



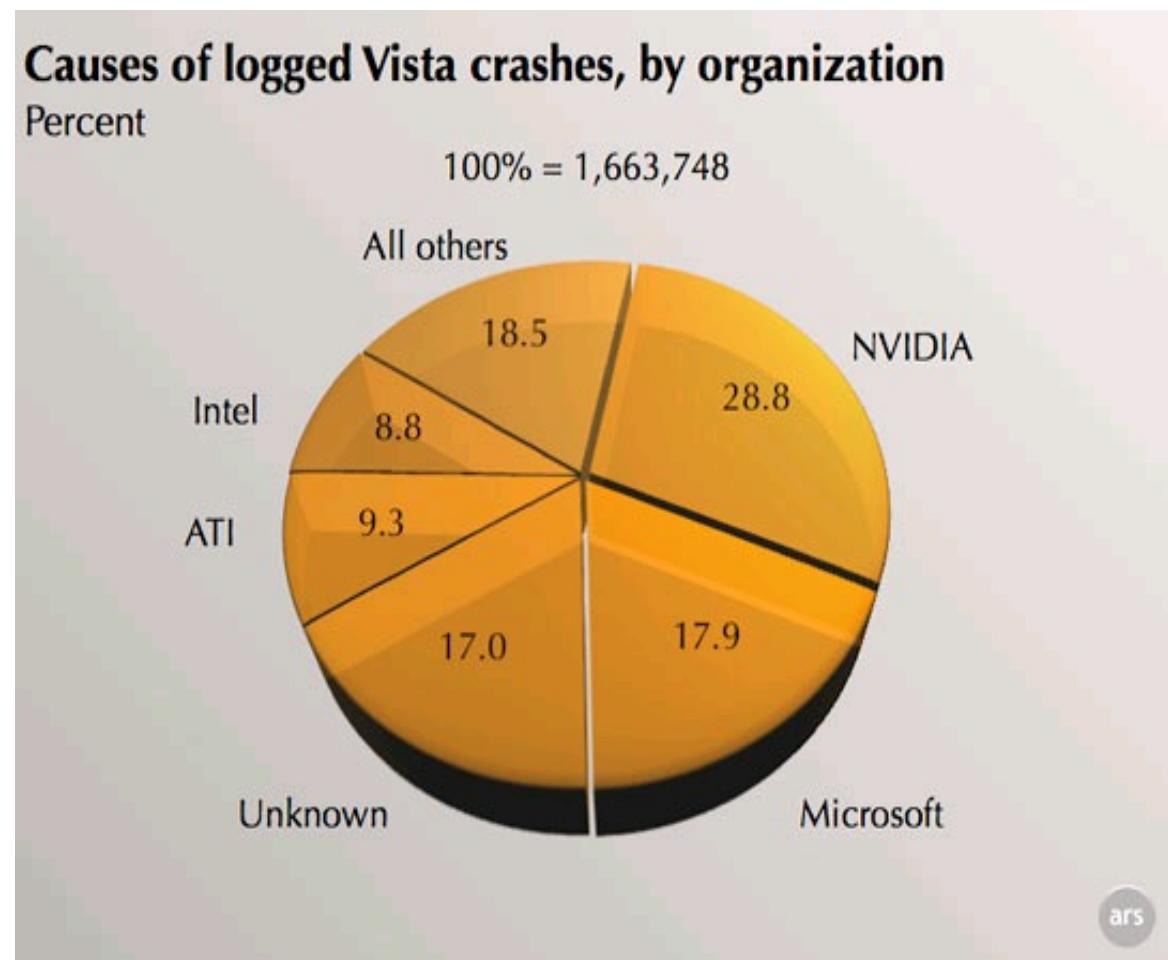
# Device drivers



Drivers in Windows vs Mac?  
In Vista, many drivers run in user space. Why?

# Crashes in Vista

“The data clearly indicates that NVIDIA had a driver problem. Driver problems are nothing new, particularly during the launch of a new OS, but the high incidence of NVIDIA driver crashes may have fueled public perception of Vista as an unstable and . . .”



ars

<http://arstechnica.com/hardware/news/2008/03/vista-capable-lawsuit-paints-picture-of-buggy-nvidia-drivers.ars>

# OS Supports Many Device Classes

- Has internal subsystem that devices plug into
- NICs (sockets)
- File systems (read/write)
  - Disks are underneath file systems
- Audio subsystem
- Keyboard subsystem
- Video subsystem
- USB device subsystems

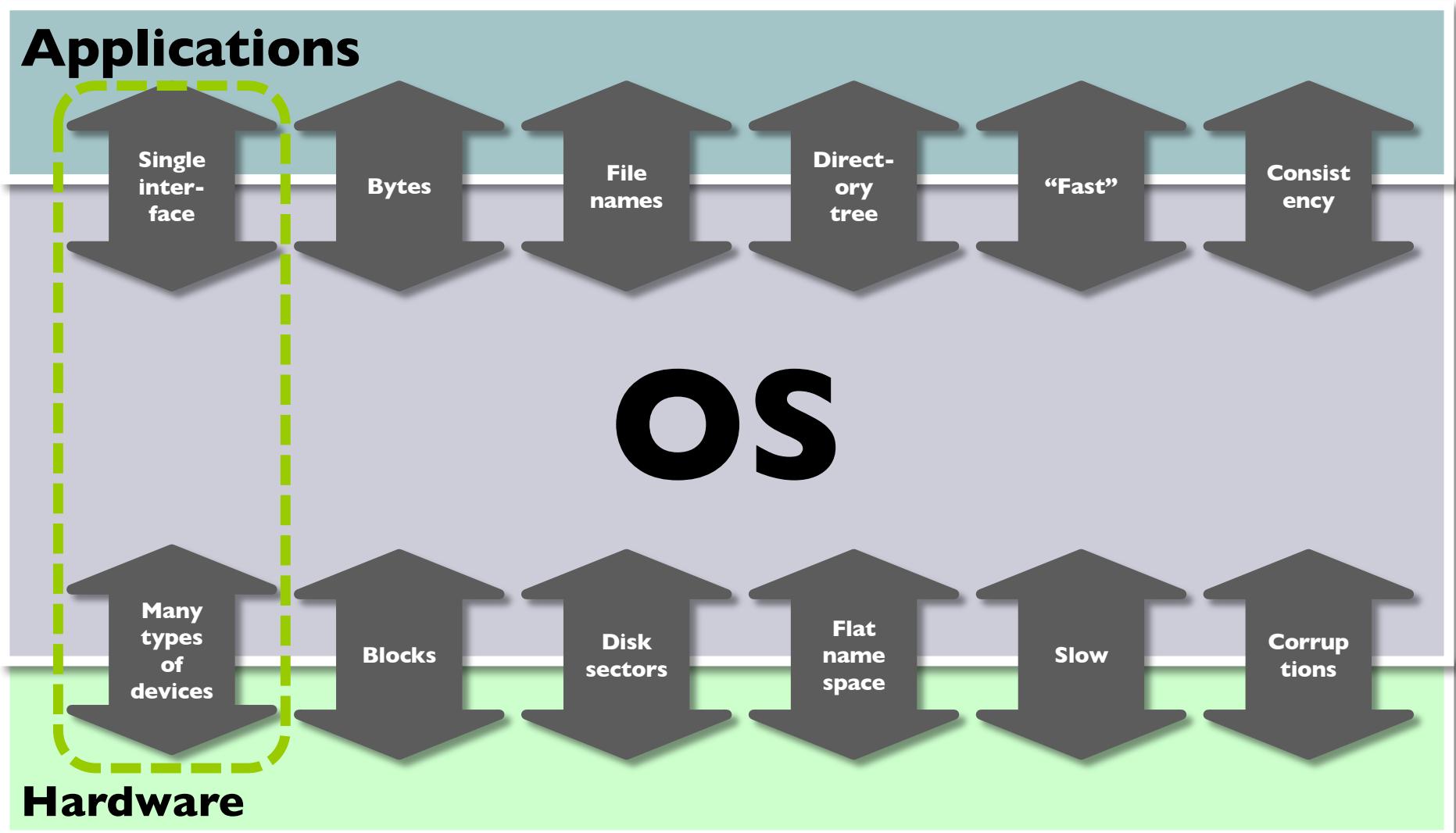
# Generic Interface to I/O devices

- read/write - transfer data to and from device
  - Communicate through virtual files in /dev/ directory
- ioctl – generic system call to control I/O devices
  - virtual file descriptor id (int)
  - device-specific request code (int)
  - untyped pointer to memory (char\*)
- Is all of this complexity good?

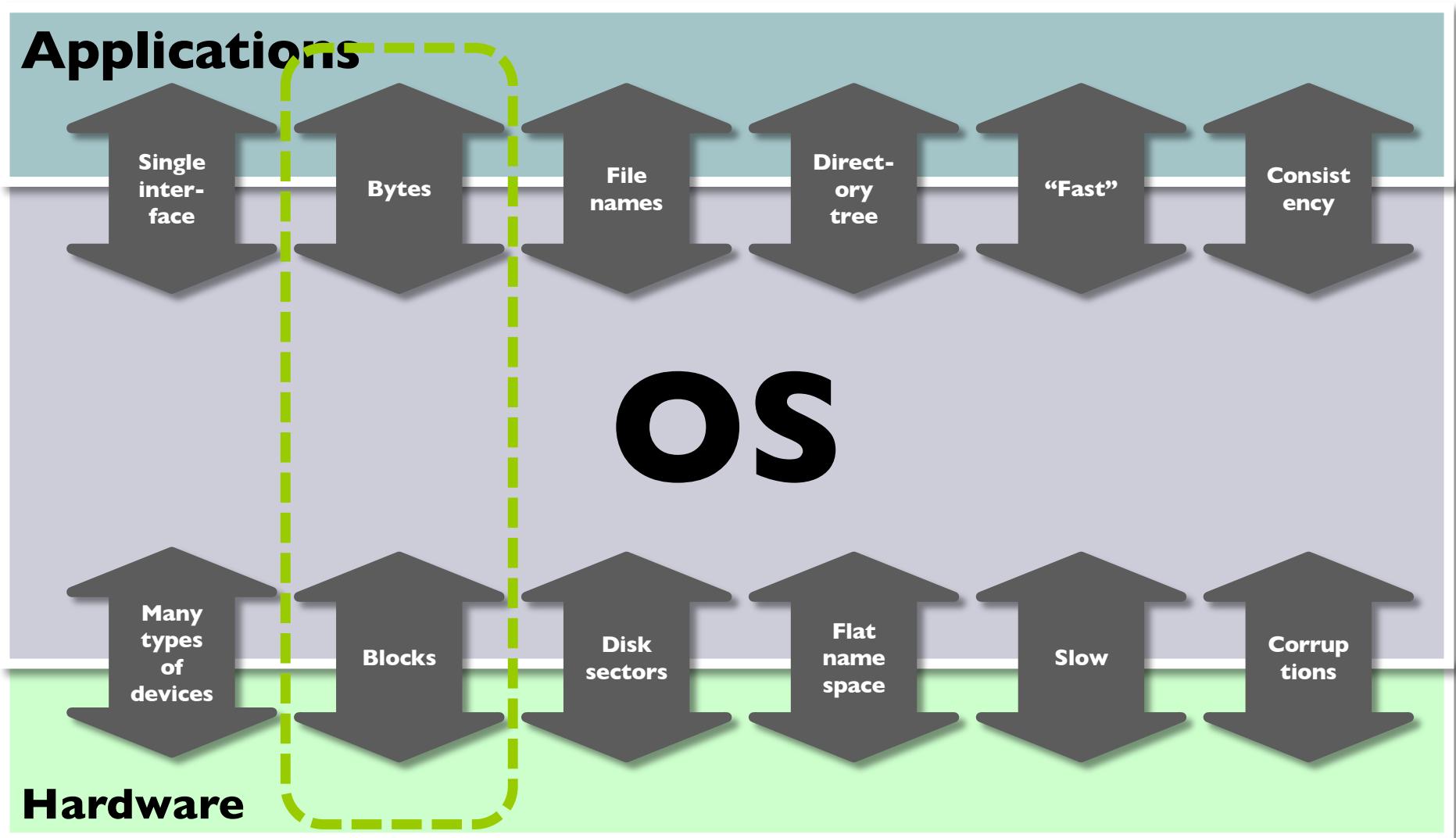
# Other abstractions we've seen

- Abstractions simplify things for their users
- **Threads**
  - Programmers don't have to worry about sharing the CPU
- **Address spaces**
  - Applications don't have to worry about sharing phys mem
- **TCP**
  - Network code needn't worry about unreliable network
- **Device drivers**
  - Parts of OS don't have to worry about device variations

# Virtual/physical interfaces



# File Systems



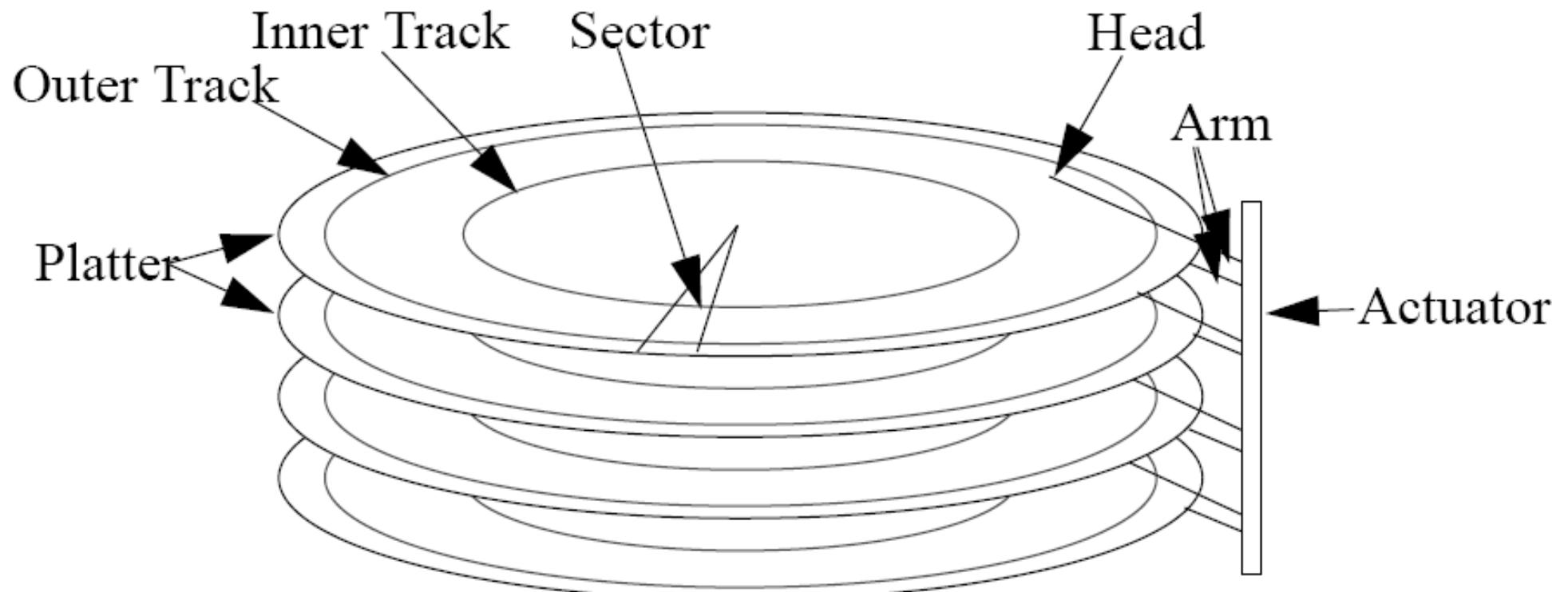
# Block-oriented vs byte-oriented

- Disks are accessed in terms of **blocks**
  - Also called **sectors**
  - Similar idea to memory pages
  - E.g. 4KB chunks of data
- Programs deal with bytes
  - E.g. change ‘J’ in “Jello world” to ‘H’

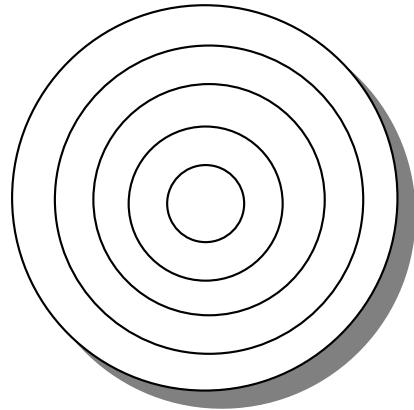
# Block-oriented vs byte-oriented

- How to read less than a block?
  - Read entire block
  - Return the right portion
- How to write less than a block?
  - Read entire block
  - Modify the right portion
  - Write out entire block
- Nothing analogous to byte-grained load/store
- Flash devices are even more complicated

# Disk geometry

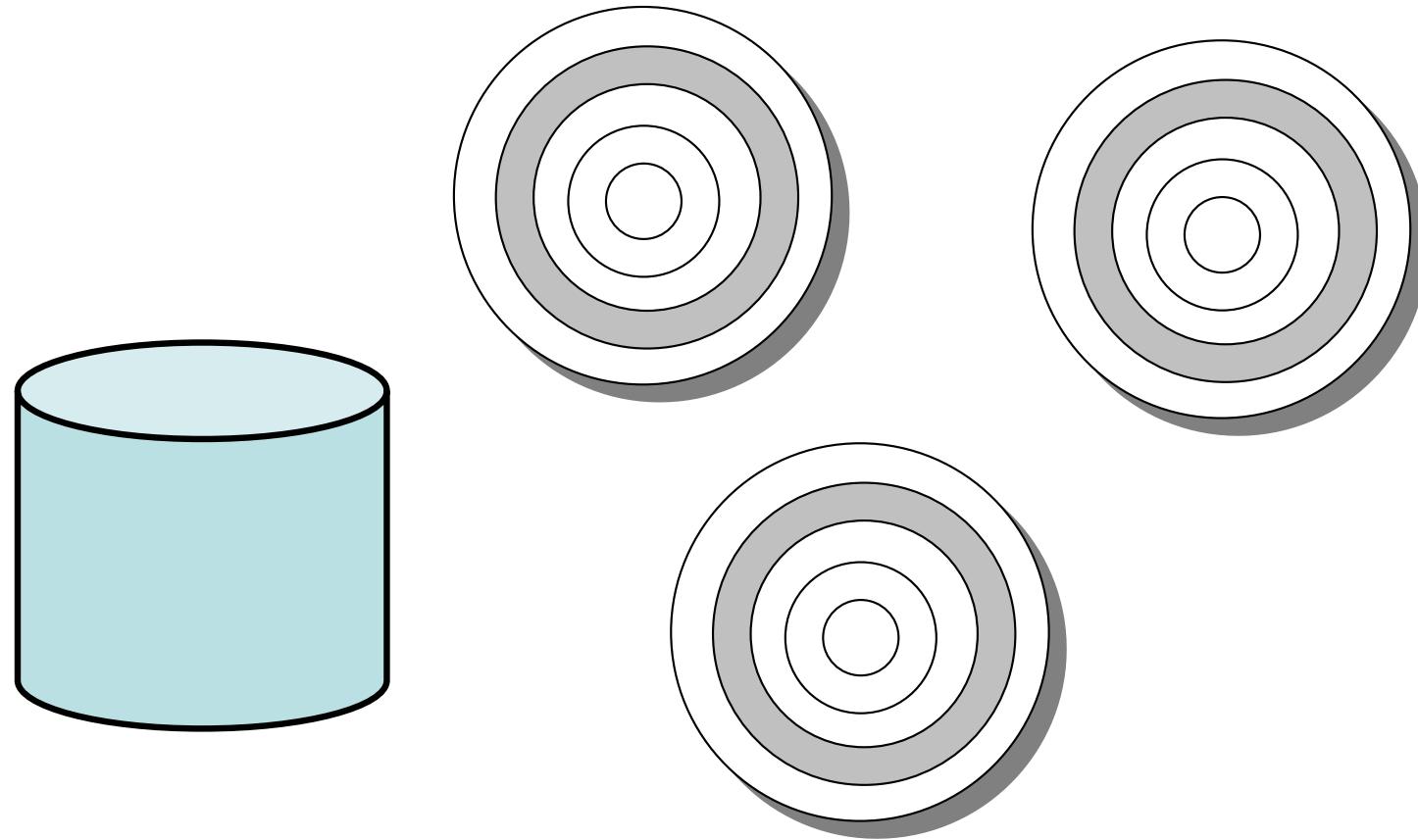


# Surface organized into tracks



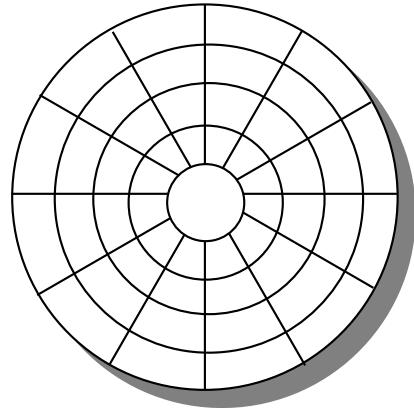
“The tracks are the thin concentric circular strips on a floppy medium or platter surface which actually contain the magnetic regions of data written to a disk drive. They form a circle and are (therefore) two-dimensional. At least one head is required to read a single track.”  
(wikipedia)

# Parallel tracks form cylinders



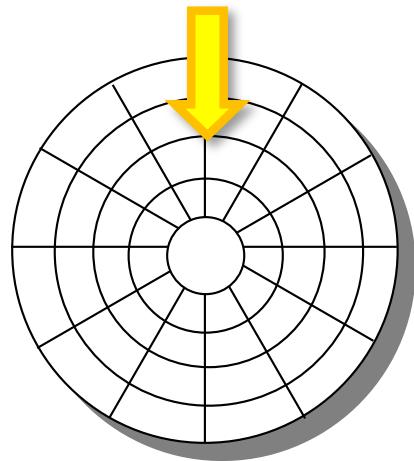
“A cylinder comprises the same track number but spans all such tracks across each platter surface that is able to store data (without regard to whether or not the track is "bad"). Thus, it is a three-dimensional object.”

# Tracks broken up into sectors



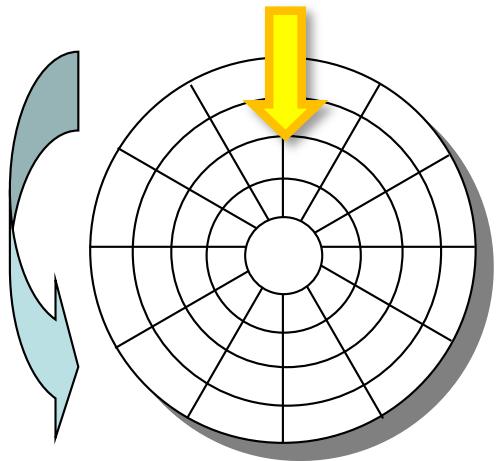
“Each usable side of a platter can also be thought of as a collection of slices called sectors.”  
Sometimes called blocks (512 bytes in size)

# Disk head position

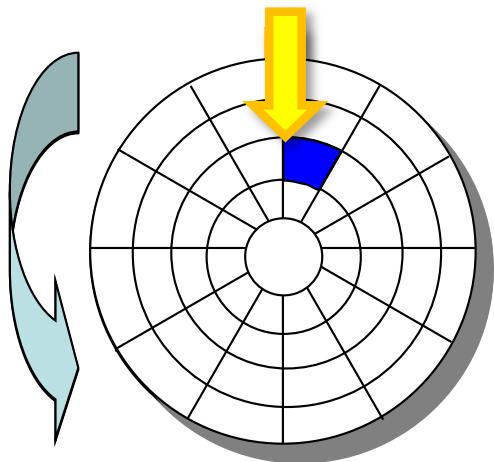


“Data is written to and read from the surface of a platter by a device called a head. Naturally, a platter has 2 sides and thus 2 surfaces on which data could be manipulated; usually there are 2 heads per platter--one on each side, but not always.”

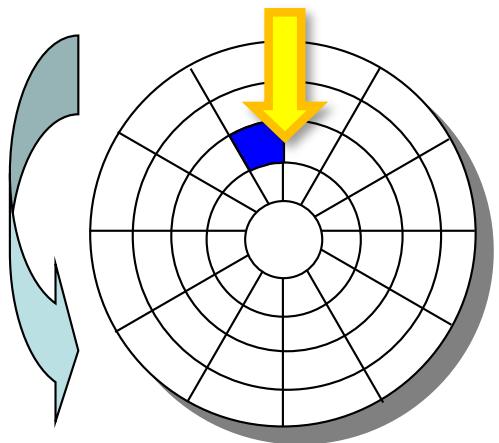
# Rotation is counter-clockwise



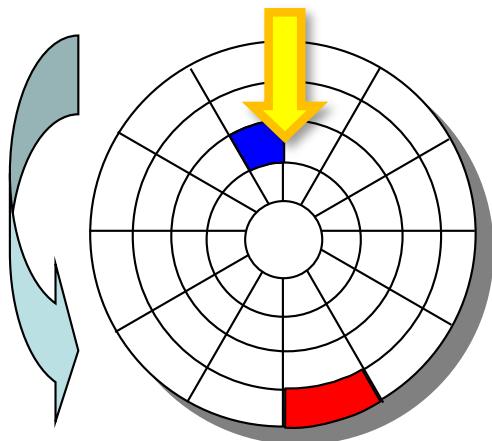
# About to read a sector



# After reading blue sector



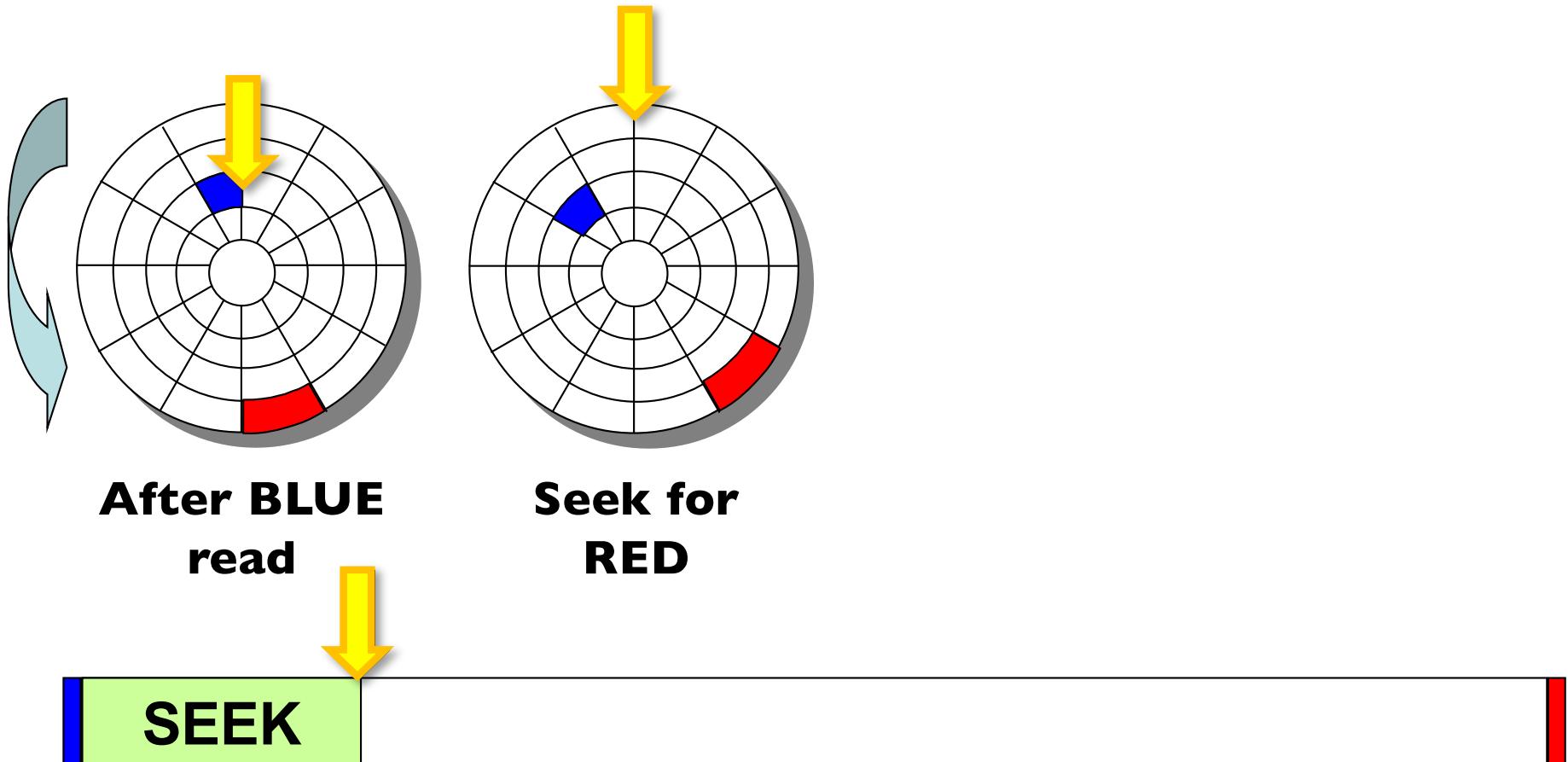
# Red request scheduled next



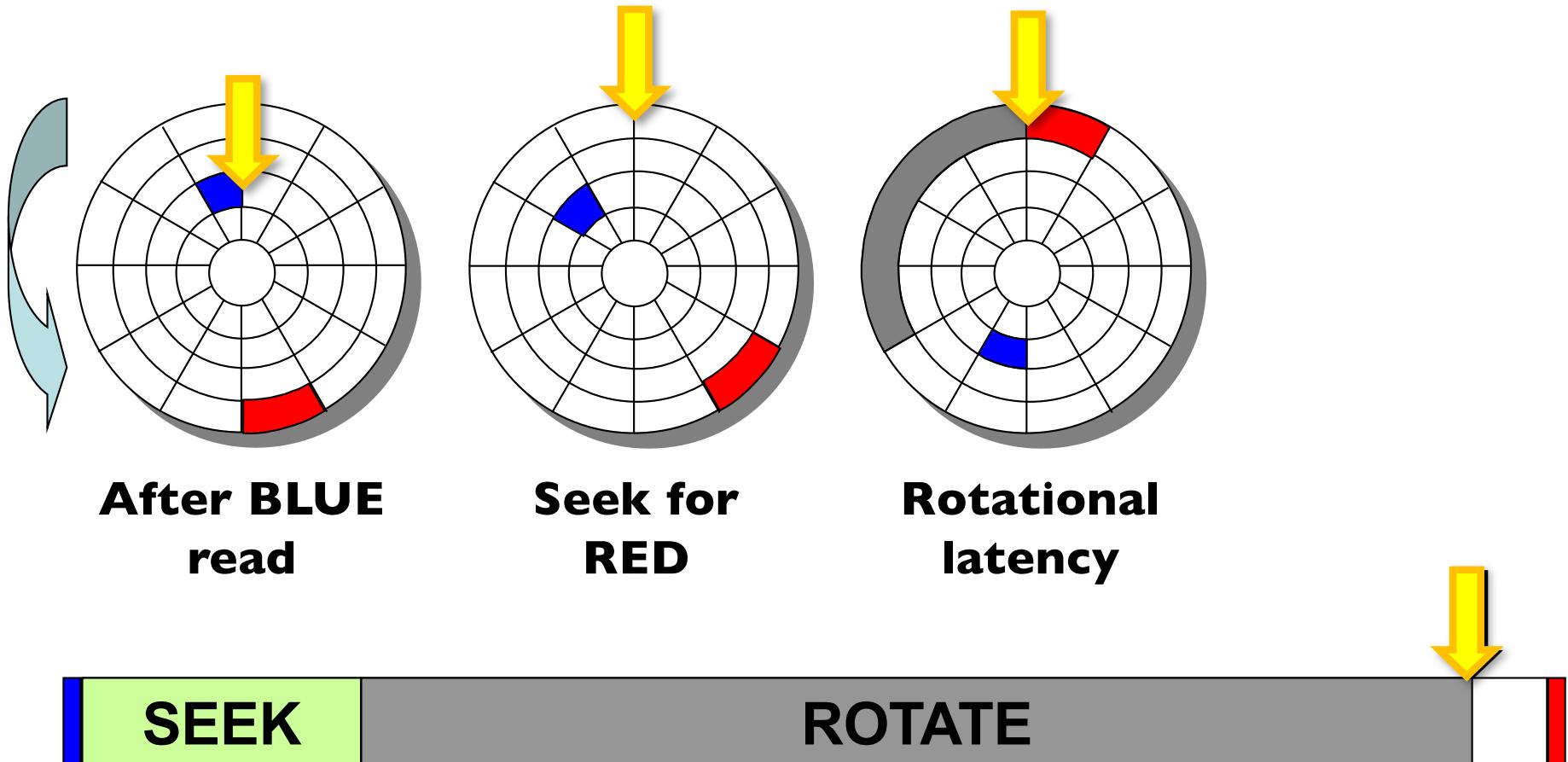
**After BLUE  
read**



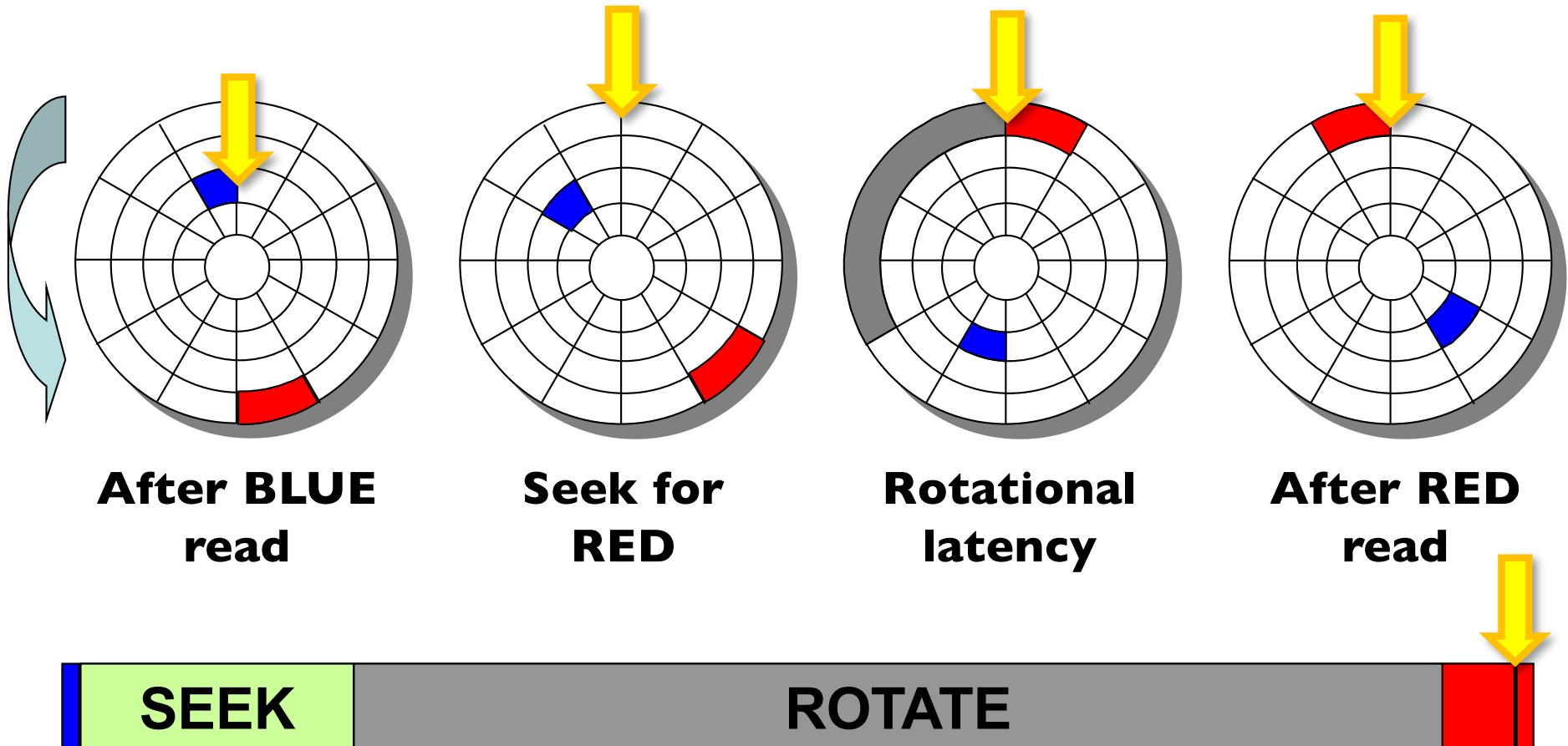
# Seek to red's track



# Wait for red sector to reach head



# Read red sector



# To access a disk

1. Queue (wait for disk to be free)
  - 0-infinity ms
2. Position disk head and arm
  - Seek + rotation
  - 0-12 ms
  - Pure overhead
3. Access disk data
  - Size/transfer rate (e.g. 49 MB/s)
  - Useful work

# Goals for optimizing performance

- Disk is really slow
  - In the time to do a disk seek ( $\sim 6$  ms)
  - 3GHz processor executes 18 million cycles!
- $Efficiency = transfer\ time / (positioning + transfer\ time)$ 
  - a) Best option is to eliminate disk I/O (caching)
  - b) Then try to minimize positioning time (seeks)
  - c) Then try to amortize positioning overhead

# Goals for optimizing performance

- For each re-positioning, transfer a lot of data
  - True of any storage device
  - Particularly true for disks
  - Fast transfer rate, long positioning time
- For Seagate Barracuda 1181677LW
  - Can get 50% efficiency by...
  - ...transferring 300KB, which takes 6 ms

# How to reduce positioning time

- Can optimize when writing to disk
  - Try to reduce future positioning time
  - Put items together that will be accessed at the same time
  - How can we know this?
  - Use general usage patterns (spatial locality)
    - Blocks in same files usually accessed together
    - Files in a directory tend to be accessed together
    - Files tend to be accessed with the containing directory
  - Learn from past accesses to specific data
- Can also optimize when reading from disk

# Sequential and Random Access

- Sequential Access
  - Read/write disk blocks that are next to each other
  - This is (relatively) fast
    - $\sim 100\text{MB/s}$
- Random Access
  - Read/write disk blocks that are not next to each other
  - This is (very) slow
    - $\sim 1\text{MB/s}$

# Disk scheduling

- Idea
  - Take a bunch of accesses
  - Re-order them to reduce seek time
  - Kind of like traveling salesman problem
- Can do this either in OS or on disk
  - Disk knows more about disk geometry
  - OS knows more about the requests

# First-come, first-served (FCFS)

- Disk scheduling
  - Goal: reorder disk access to decrease total positioning time
- E.g. start at track 53
- Other queued requests
  - 98, 183, 37, 122, 14, 124, 65, 67
- Total head movement
  - 640 tracks
  - Average 80 tracks per seek

# Shortest seek time first (SSTF)

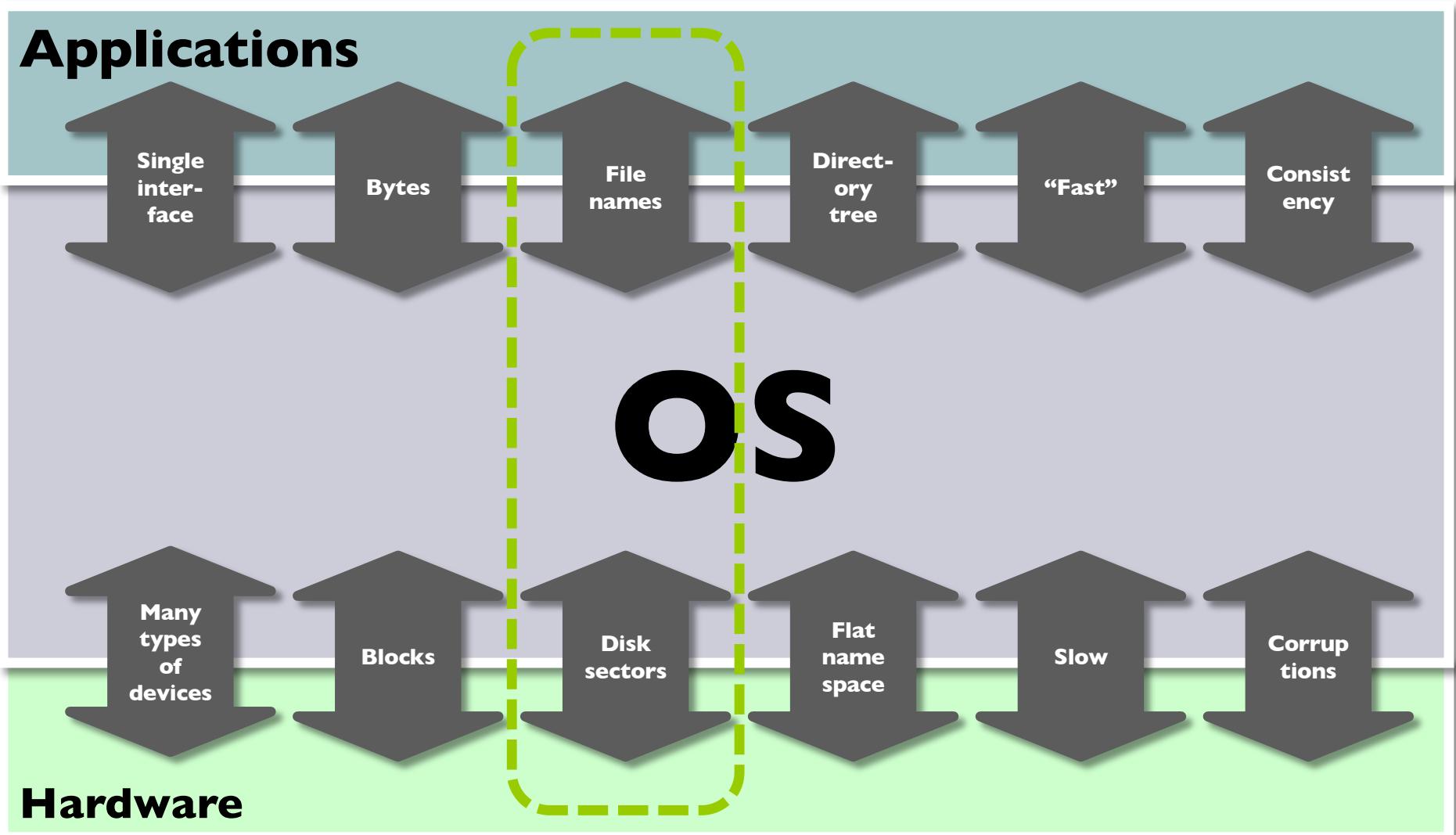
- E.g. start at track 53
- Other requests
  - 53, 65, 67, 37, 14, 98, 122, 124, 183
- Total head movement
  - 236 tracks
  - Average 30 tracks per seek

**Any problems with this?**

# Anticipatory Scheduling

- Want to maximize sequential reads
  - *Typical pattern:* read, compute, read, compute..
  - During compute, the I/O is “idle”
    - I/O scheduler would normally switch to other I/O
  - However, idleness **“deceptive”** since process will read again
- Solution: wait briefly (a few ms) after synchronous read, in case process reads again from same spot
- Big improvement for some applications
  - E.g., apache web server gets 71% more disk throughput
  - Built into default Linux kernel disk scheduler

# Virtual/physical interfaces



# File systems

- What is a file system?
  - OS abstraction that makes disks easy to use
- File system issues
  1. How to map file space onto disk space
    - Structure and allocation: like memory management
  2. How to use names instead of sectors
    - Naming and directories: not like memory

# Intro to file system structure

- Overall question:
  - How do we organize things on disk?
- Really a data structure question
  - What data structures do we use on disk?

# Intro to file system structure

- Need an initial object to get things going
  - In file systems, this is a **file header**
  - Unix: this is called an **inode** (indexed node)
- File header contains info about the file
  - Size, owner, access permissions
  - Last modification date
- Many ways to organize files and headers on disk
  - Use actual usage patterns to make good decisions

# File system usage patterns

1. 80% of file accesses are reads (20% writes)
  - Ok to save on reads, even if it hurts writes
2. Most file accesses are sequential
  - Form of **spatial locality**
  - Put sequential blocks next to each other
  - Can pre-fetch blocks next to each other
3. Most files are small
4. Most bytes are consumed by large files

# I) Contiguous allocation

- Many ways to organize data on disk
  - **Option I:** Contiguous allocation
- Store a file in one contiguous segment
  - Sometimes called an “extent”
- Reserve space in advance of writing it
  - User could declare in advance
  - If grows larger, move it to a place that fits

# I) Contiguous allocation

- File header contains
  - Starting location (block #) of file
  - File size (# of blocks)
  - Other info (modification times, permissions)
- Exactly like base and bounds memory

# I) Contiguous allocation

- Pros
  - Fast sequential access
  - Easy random access
- Cons
  - External fragmentation
  - Internal fragmentation
  - Hard to grow files

## 2) Indexed files

- File header

<b>File block #</b>	<b>Disk block #</b>
0	18
1	50
2	8
3	15

- Looks a lot like a page table

## 2) Indexed files

- Pros
  - Easy to grow (don't have to reserve in advance)
  - Easy random access
- Cons
  - How to grow beyond index size?
  - Sequential access may be slow. Why?
    - May have to seek after each block read

Why wasn't sequential access a problem with page tables?  
Memory doesn't have seek times.

## 2) Indexed files

- Pros
  - Easy to grow (don't have to reserve in advance)
  - Easy random access
- Cons
  - How to grow beyond index size?
  - Lots of seeks for sequential access

How to reduce seeks for sequential access?

When you allocate a new block, choose one near the one that precedes it. E.g. blocks in the same cylinder.

# What about large files?

- Could just assume it will be really large
- Problem?
  - Wastes space in header if file is small
  - Max file size is 4GB (on 32 bits)...
    - ...and File block is 4KB → 1 million pointers
    - 4 MB header for 4 byte pointers
- Remember most files are small
  - 10,000 small files → 40GB of headers

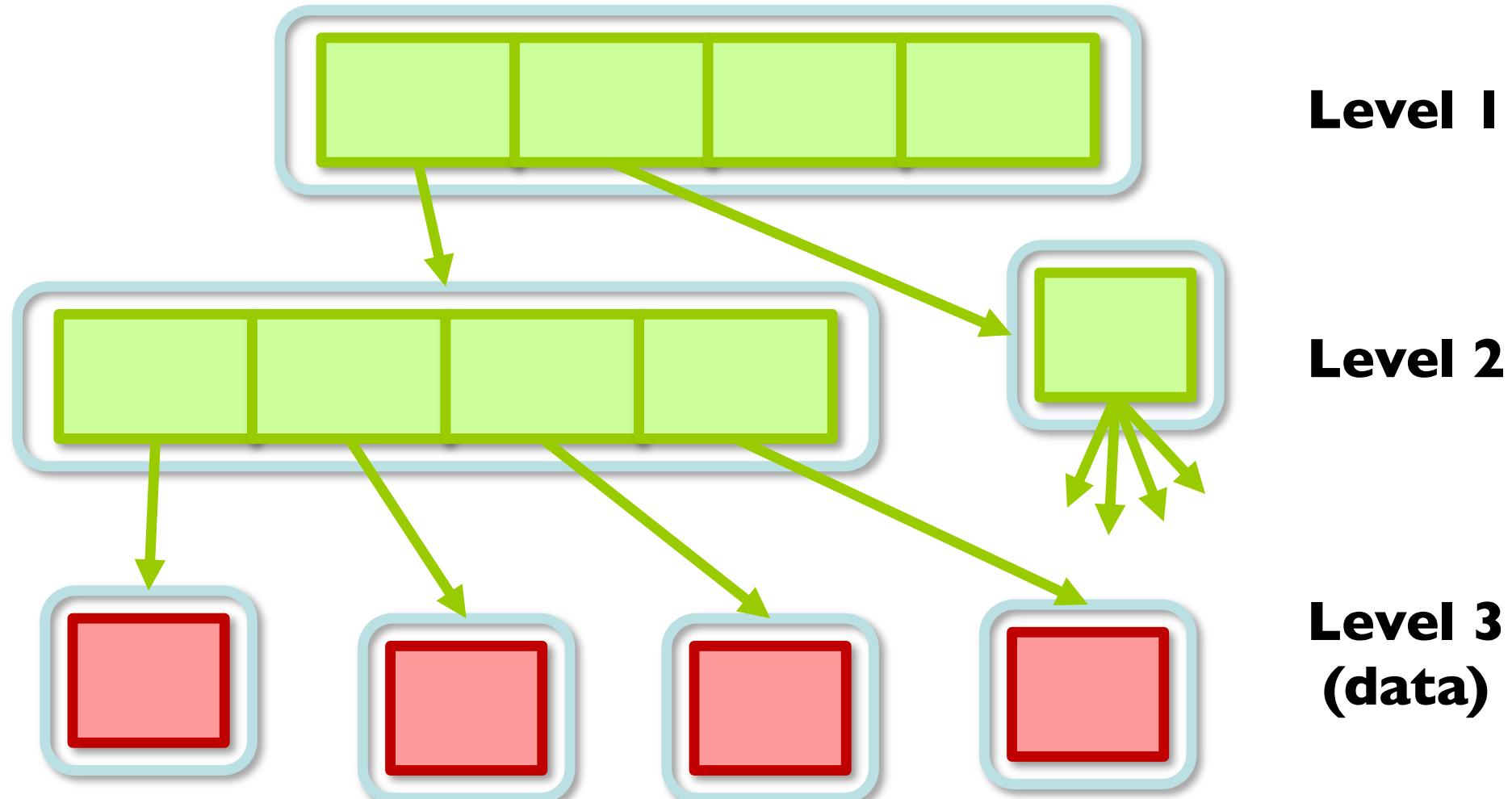
# What about large files?

- Could use a larger block size
- Problem?
  - Internal fragmentation (most files are small)
- Solution
  - Use a more sophisticated data structure

### 3) Multi-level indexed files

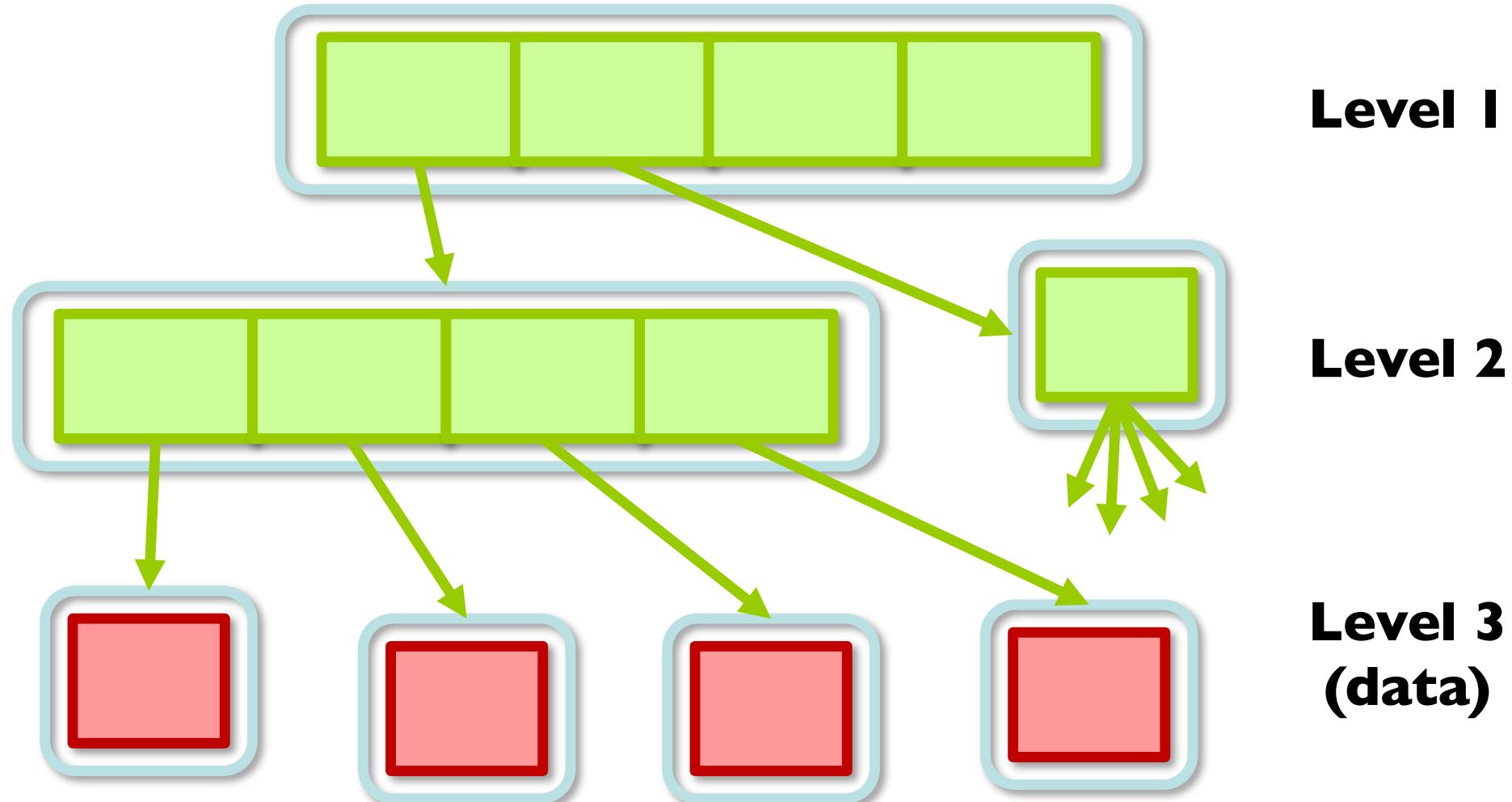
- Think of indexed files as a shallow tree
- Instead could have a multi-level tree
  - Level 1 points to level 2 nodes
  - Level 2 points to level 3 nodes
- Gives us big files without wasteful headers
  - How?
    - Store “indirect blocks” in file data blocks
    - Only level 1 is in inode

### 3) Multi-level indexed files



How many access to read one block of data?  
3 (one for each level)

### 3) Multi-level indexed files

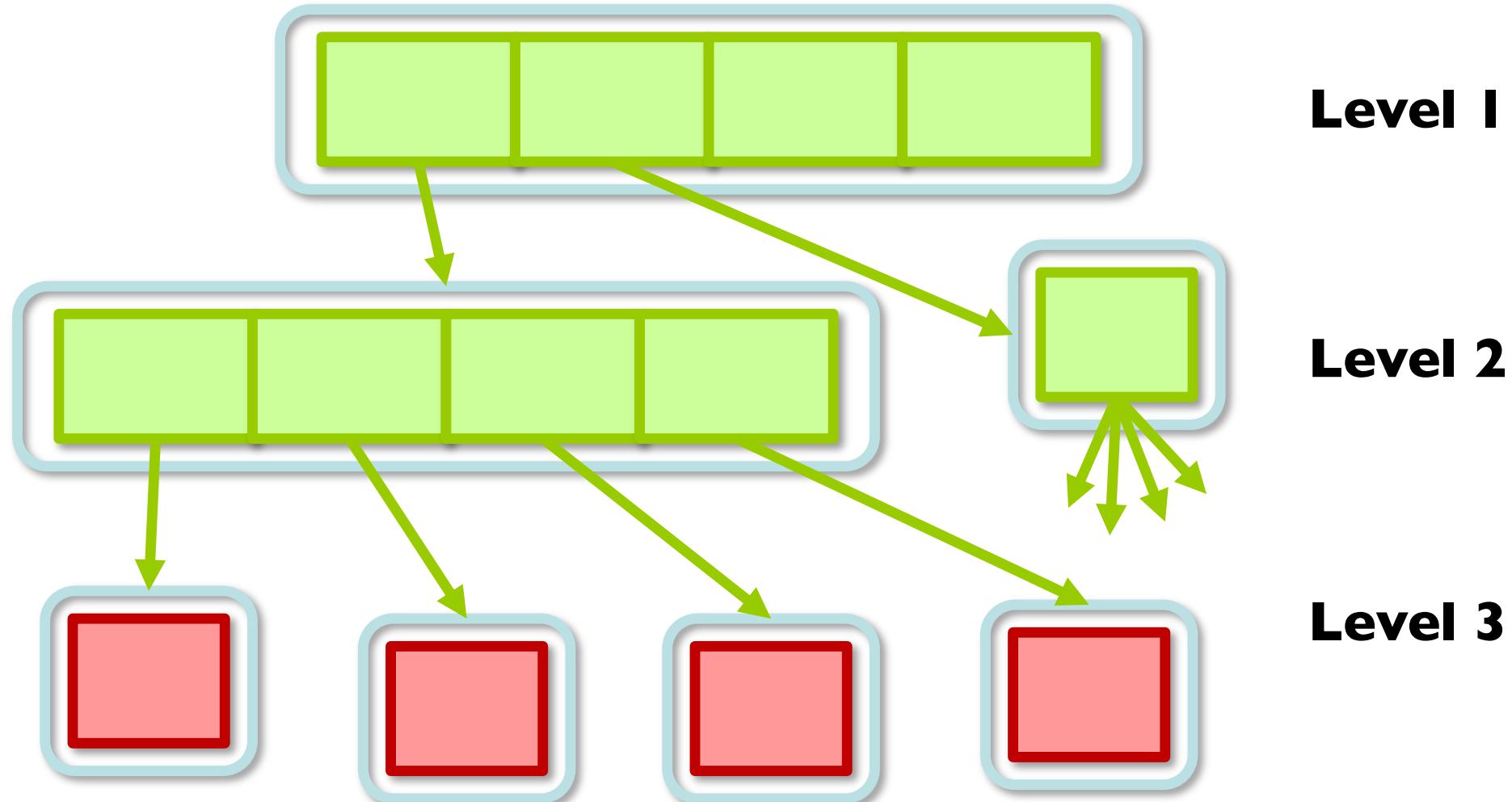


How to improve performance?  
Caching.

### 3) Multi-level indexed files

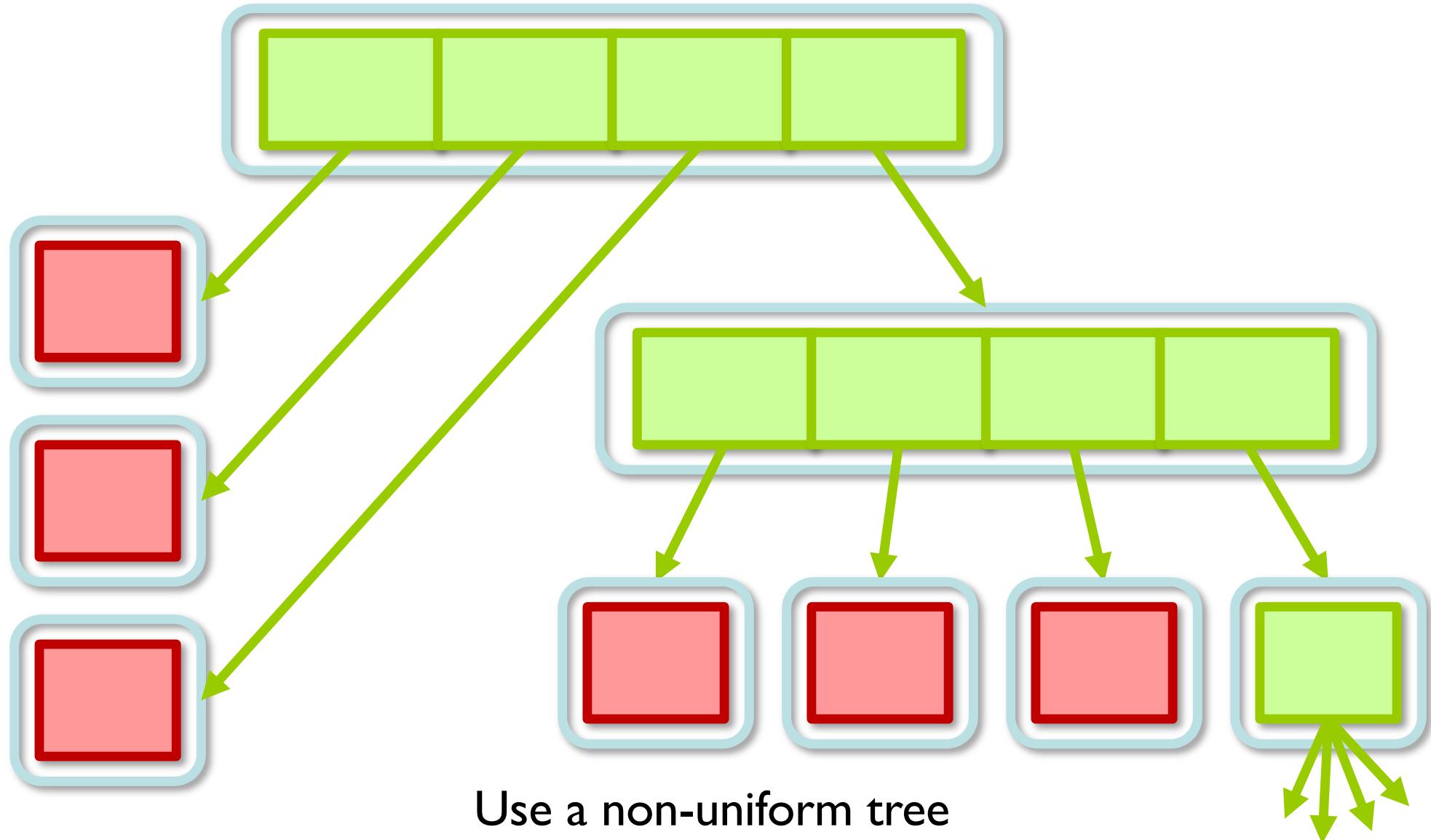
- To reduce number of disk accesses
  - Cache level 1 and level 2 nodes
- Often a useful combination
  - Indirection for flexibility
  - Caching to speed up indirection
  - Can cache lots of small pointers

### 3) Multi-level indexed files



What about small files (i.e. most files)?

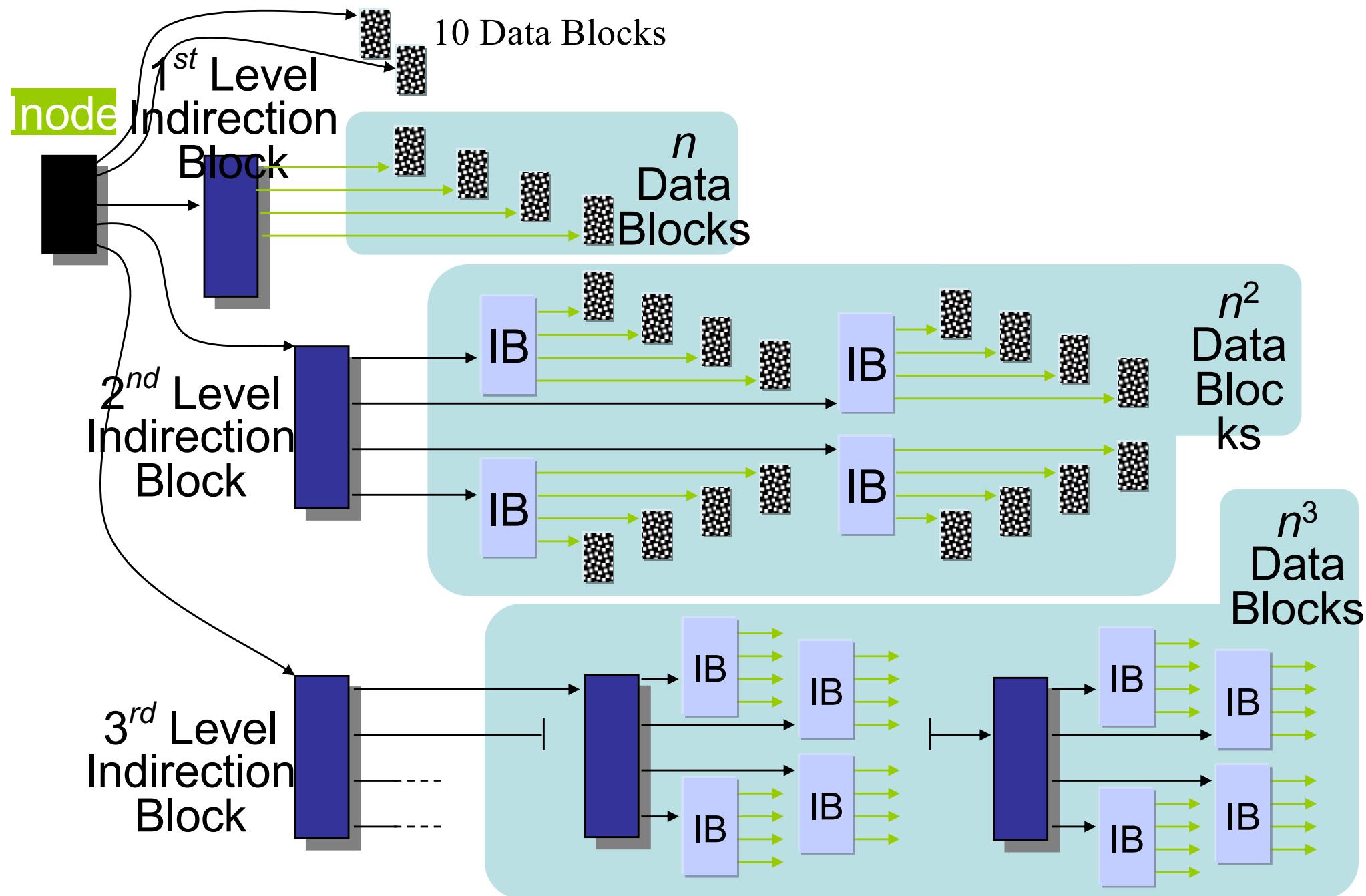
### 3) Multi-level indexed files



# Multi-level Indirection in Linux

- File header contains 13 pointers
  - 10 pointers to 4kb data blocks; 11th pointer → indirect block; 12th pointer → doubly-indirect block; and 13th pointer → triply-indirect block
- Implications
  - Upper limit on file size (~2 TB)
  - Blocks are allocated dynamically (allocate indirect blocks only for large files)
- 
-

# UNIX Depiction: Multilevel, indirection, index blocks



### 3) Multi-level indexed files

- Pros
  - Simple
  - Files can easily expand
  - Small files don't pay the full overhead
- Cons
  - Large files need lots of indirect blocks (slow)
  - Could have lots of seeks for sequential access

# Free List Allocation

- Need a data block
  - Take one from list of free data blocks
- Need an inode
  - Take one from list of free inodes
- Why do inodes have their own free list?
  - They are fixed (small) size
  - They exist at fixed locations
  - There are a fixed number of them (limits number of files)

# Free list representation

- Represent free list as bit vector:

1111111111111100111010101110111...

- If bit  $i = 0$  then block  $i$  is *free*, if  $i = 1$  it is *allocated*

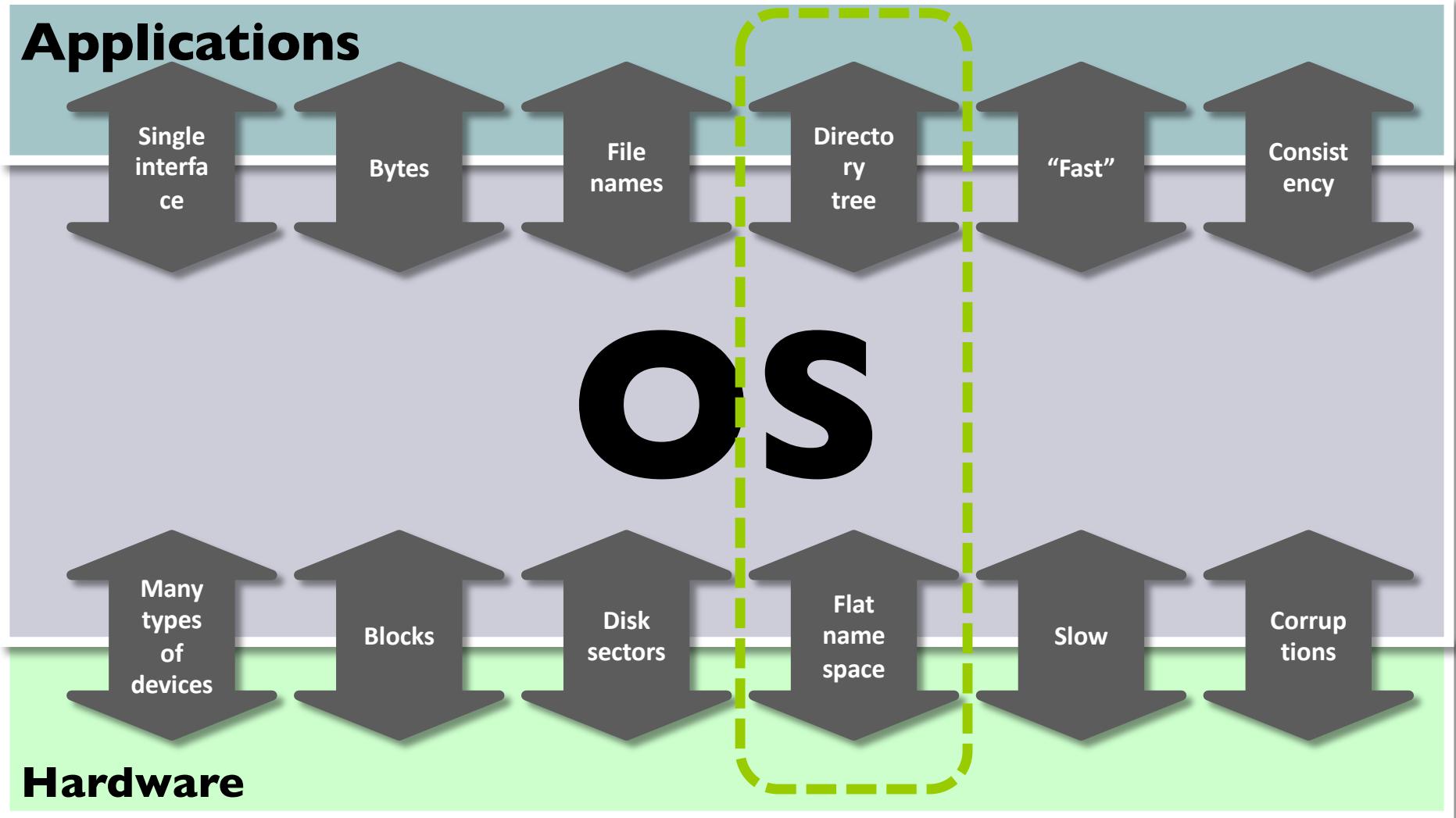
- Simple to use and vector is compact:

- 1TB disk with 4KB blocks is  $2^{28}$  bits or 32MB
- If free sectors uniformly distributed then...
- ...expected number of scanned bits before finding 0 is
  - $n/r$  where  $n$  is number blocks on disk, and  $r$  is number of free blocks
- If disk 90% full, average number of scanned bits is only 10
  - Independent of disk size

# Deleting a File

- Data blocks go back on free list
- Indirect blocks go back on free list
  - Expensive for large files
  - Possibly clear inodes (make data blocks “dead”)
- Inode free list written
- Directory updated
- Order matters:
  - Can’t put block on free list until no inode points to it

# Virtual/physical interfaces

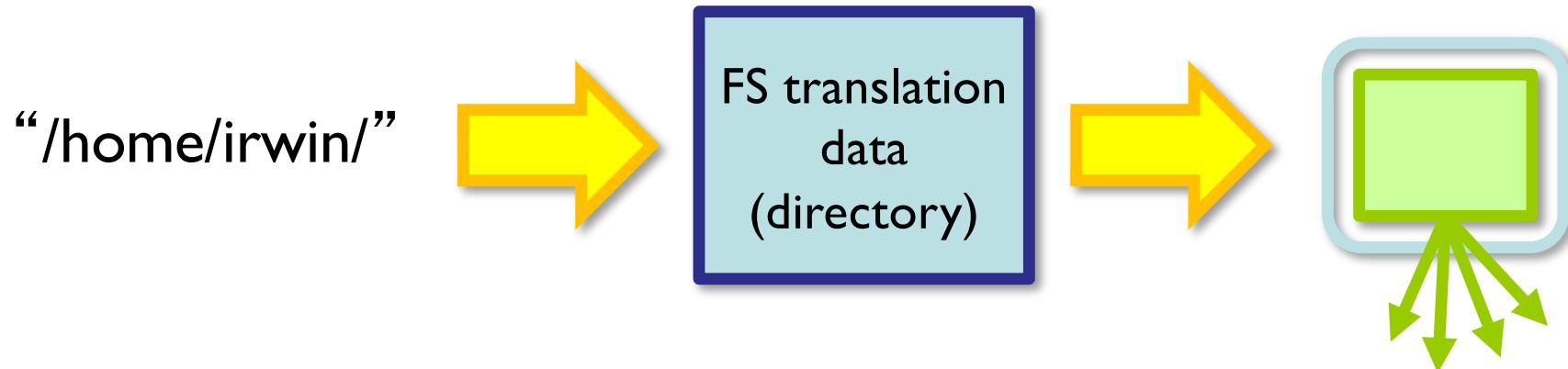


# Review: Naming and directories

- How to locate a file?
  - Need to tell OS which file header
  - Use a symbolic name or click on an icon
- Could also describe contents
  - Like Google desktop and Mac Spotlight
  - Naming in databases works this way

# Review: Name translation

- User-specified file → on-disk location
  - Lots of possible data structures
  - (hash table, list of pairs, tree, etc)
- Once you have the header, the rest is easy



# Review: Directories

- **Directory**
  - Contains a mapping for a set of files
  - Name → file header's disk block #
- Often just table
  - <file name, disk block #> pairs

# Directories

- Typically a hierarchical structure
  - Directory A has mapping to files and directories
- `/irwin/ece570/names`
  - `/` is the root directory
  - `/irwin` is a directory within the `/` directory
  - `/irwin/ece570` is a directory within the `/irwin` directory

# How many disk I/Os?

- Read first byte of /irwin/ece570/names?
  1. File header for / (at a fixed spot on disk)
  2. Read first data block for /
  3. Read file header for /irwin
  4. Read first data block for /irwin
  5. Read file header for /irwin/ece570
  6. Read first data block for /irwin/ece570
  7. Read file header for /irwin/ece570/names
  8. Read first data block for /irwin/ece570/names

# How many disk I/Os?

- Caching is only way to make this efficient
  - If file header block # of /irwin/ece570 is cached
  - Can skip steps 1-4
- Current working directory
  - Allows users to specify file names instead of full paths
  - Allows system to avoid walking from root on each access
  - Eliminates steps 1-4

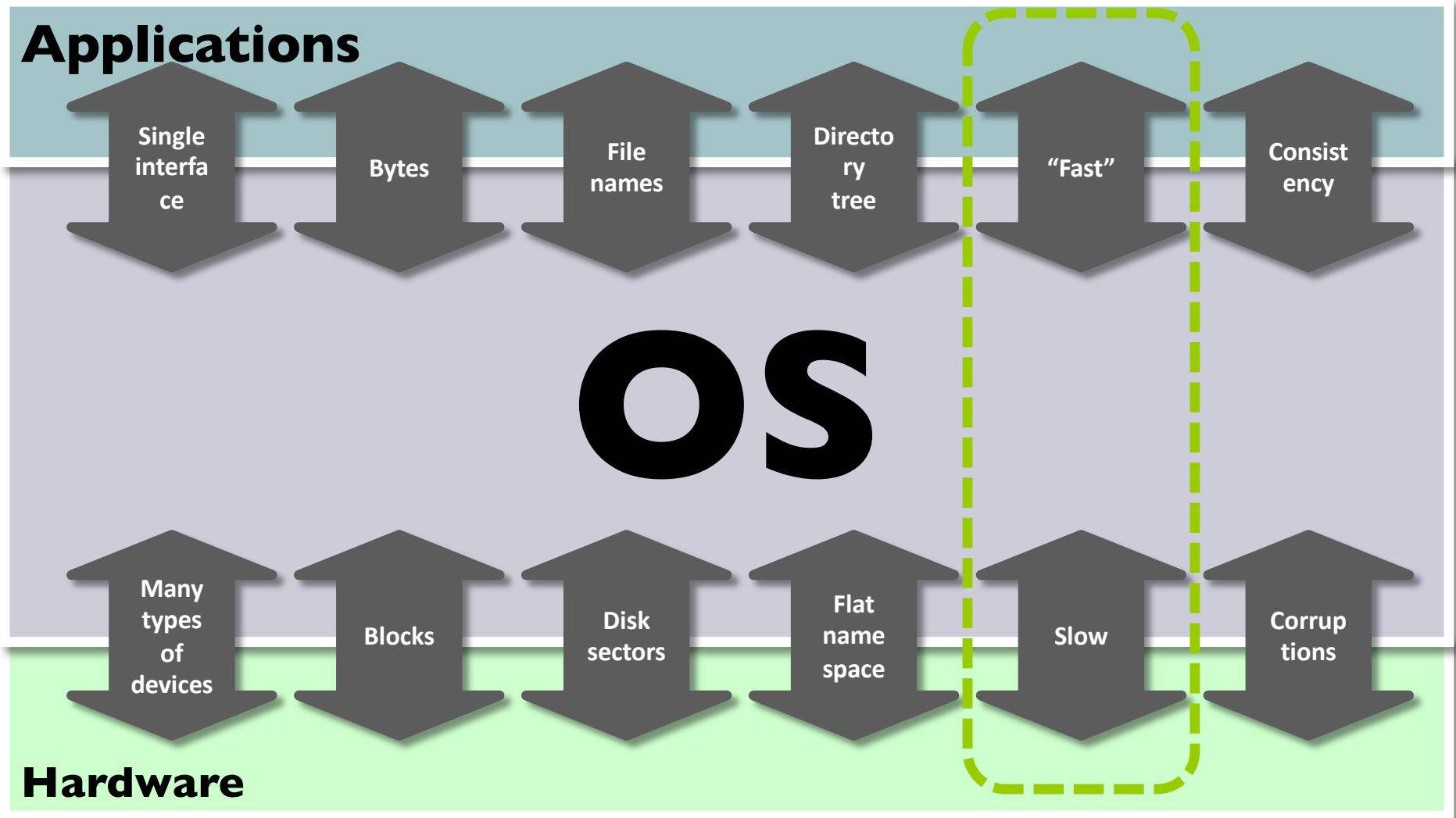
# Mounting multiple disks

- Can easily combine multiple disks
- Basic idea:
  - Each disk has a file system (a / directory)
  - Entry in one directory can point to root of another FS
  - This is called a “mount point”
- For example, on my machine
  - /bin is part of one local disk
  - /tmp is part of another local disk
  - /nfs is part of the distributed file system NFS

# Mounting multiple disks

- Requires a new mapping type
  - Name → file's disk header block #
  - Name → directory's disk header block #
  - Name → device name, file system type
- Use drivers to walk multiple file systems
  - Ext3, ext4, vfat, ntfs, etc.
- Windows: disks visible under MyComputer/{C,D,E}
- Unix: disks visible anywhere in tree
  - Can view using “df” command

# Virtual/physical interfaces



# File caching

- File systems have lots of data structures
  - Data blocks
  - Meta-data blocks
    - Directories
    - File headers
    - Indirect blocks
    - Bitmap of free blocks
- Accessing all of these is really slow

# File caching

- Caching is the main thing that improves performance
  - Random → sequential I/O
  - Accessing disk → accessing memory
- Should the file cache be in virtual or physical memory?
  - Should put in physical memory
  - Could put in virtual memory, but might get paged out
  - Worst-case: each file is on disk twice
- Could also use memory-mapped files
  - This is what Windows does

# Memory-mapped files

- Basic idea: use Virt Mem system to cache files
  - Map file content into virtual address space
  - Set the backing store of region to file
    - Instead of swap
  - Can now access the file using load/store
- When memory is paged out
  - Updates go back to file instead of swap space

# File caching issues

- Normal design considerations for caches
  - Cache size, block size, replacement, etc.
- Write-back or write-through?
  - **Write-through:** writes to disk immediately
    - Slow, loss of power won't lose updates
  - **Write-back:** delay for a period
    - Fast, loss of power can lose updates
- Most systems use a 30 sec write-back delay

# Distributed file systems

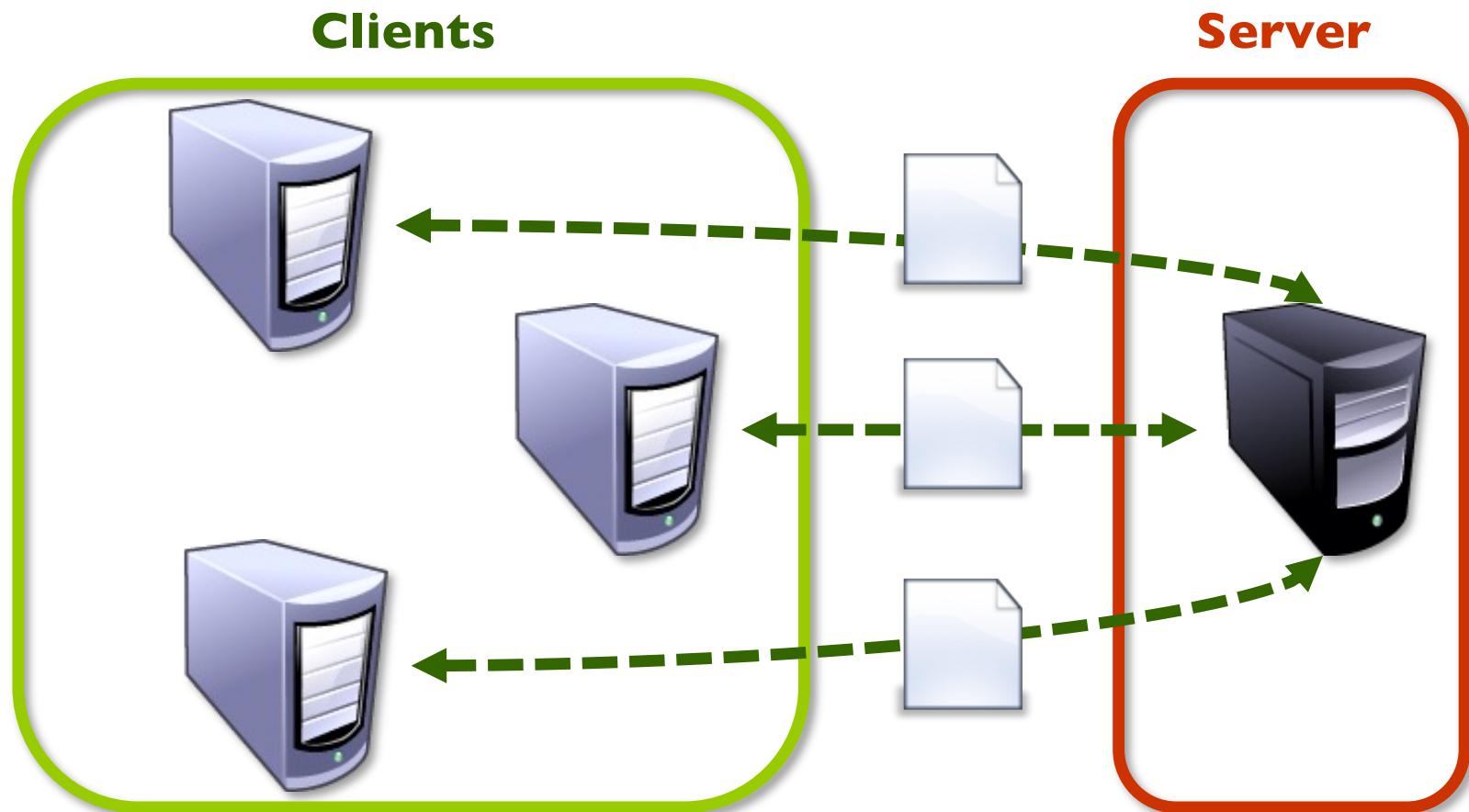
- Distributed file system
  - Data stored on many different computers
  - One, unified view of all data
- Same interface to access
  - Local files
  - Remote files

# Distributed file systems

- Examples: AFS, NFS, Samba, web(?)
- **Why use them?**
  - Easier to share data among people
  - Uniform view of files from different machines
    - Enables live VM migration
  - Easier to manage (backup, etc)

# Basic implementation

- Classic client/server model



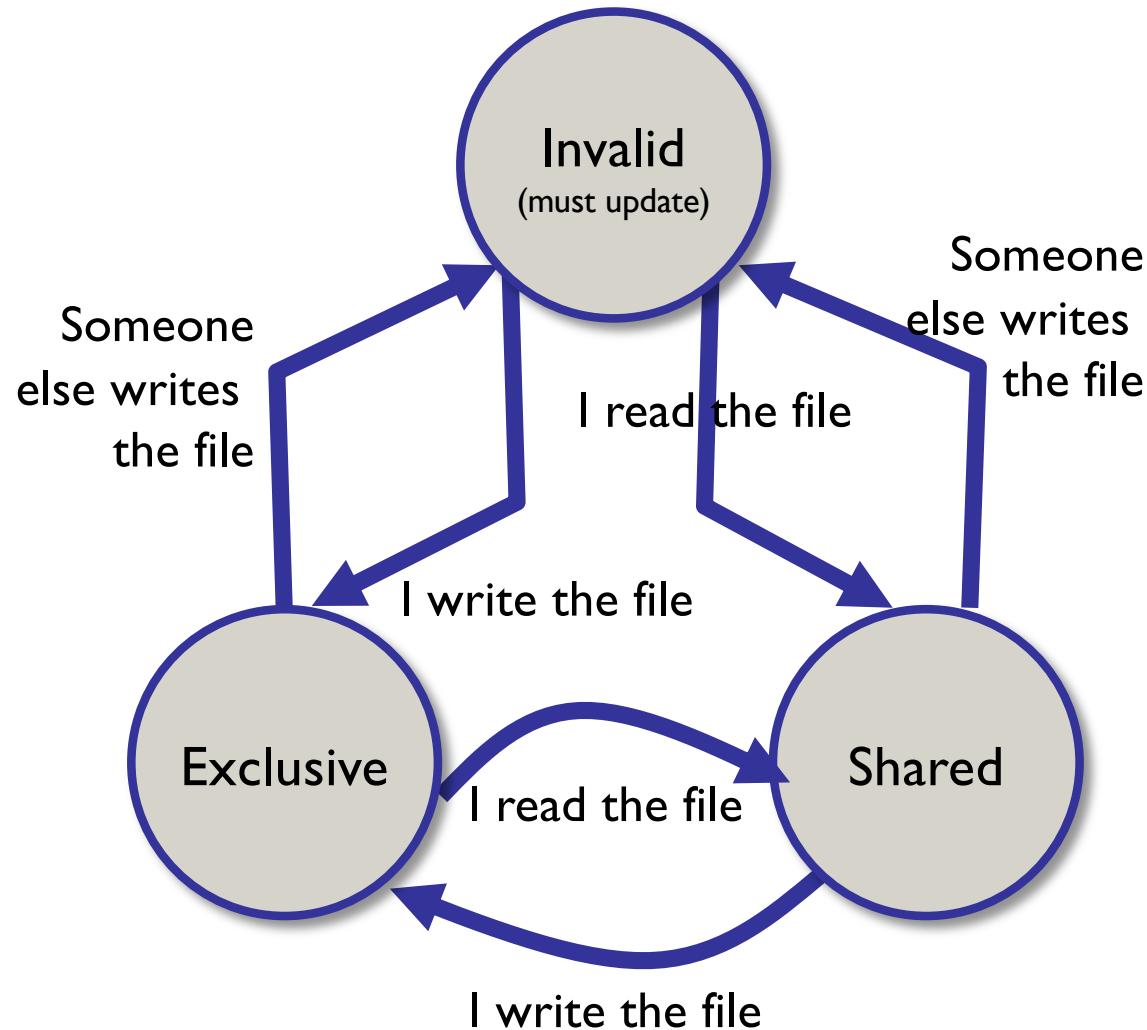
# Caching

- I'm sitting at a machine and want to access
  - /usr/ece-local/ece570/bin/submit570
  - Which is stored at autograder.ecs.umass.edu
- What should happen to the file?
  - Transfer sole copy from server to client?
  - Make a copy of the file instead (replication)

# Caching

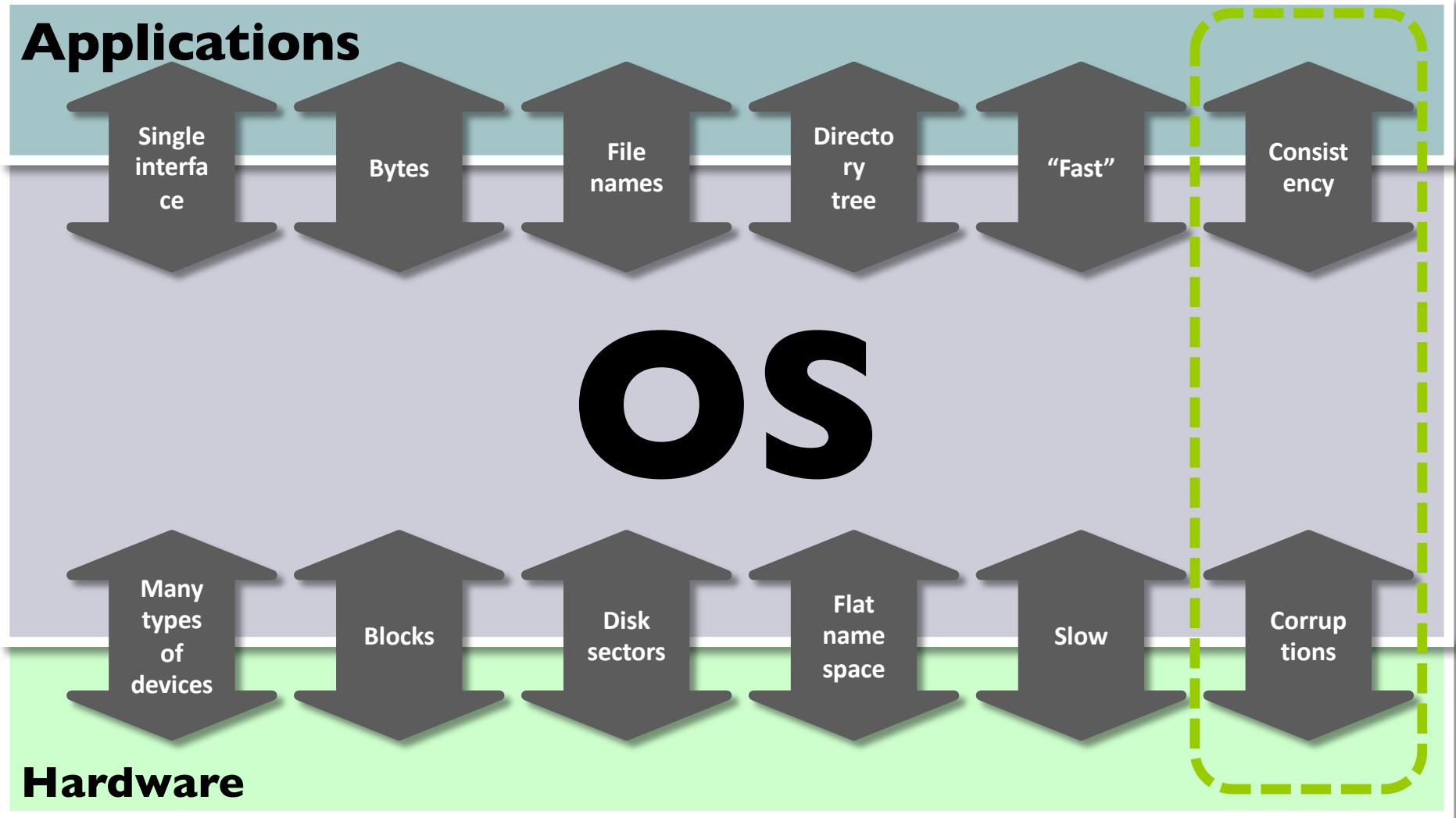
- What happens if I modify the file?
  - Other clients' copy will be out of date
- All copies must be kept consistent
  - 1. Invalidate all other copies
  - 2. Force other clients to update their copies

# Cached file states



What else from 570/670 does this remind you of?  
Reader-writer locks: allow parallel reads of shared data, one writer

# Virtual/physical interfaces



# Multiple updates and reliability

- Reliability is only an issue in file systems
  - Don't care about losing address space after crash
  - Your files shouldn't disappear after a crash
  - Files should be **permanent**
- Multi-step updates cause problems
  - Can crash in the middle

# Multi-step updates

- Transfer \$100 from your account to mine
  - 1. Deduct \$100 from your account
  - 2. Add \$100 to my account
- Crash between 1 and 2, we lose \$100
  - :-(

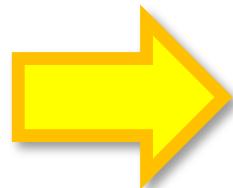
# Multiple updates and reliability

- This is a well known problem
  - No modern OS would make this mistake, right?
- Evidence suggests otherwise
  - Famous Mac OSX bug from a couple years ago
    - Directory with 3 files
    - Want to move them to external drive
    - Drive “fails” during move (unplugged)
    - Loses all data

# Multi-step updates

- Move file from one directory to another
  - I. Delete from old directory
  - 2. Add to new directory
- Crash between I and 2, we lose a file

“/home/irwin/names”



“/home/you/names”

# Multi-step updates

- Create an empty new file
  - 1. Point directory to new file header
  - 2. Create new file header
- What happens if we crash between 1 and 2?
  - Directory will point to uninitialized header
  - Kernel will crash if you try to access it
- How do we fix this?
  - Re-order the writes

# Multi-step updates

- Create an empty new file
  - 1. Create new file header
  - 2. Point directory to new file header
- What happens if we crash between 1 and 2?
  - File doesn't exist
  - File system won't point to garbage

# Multi-step updates

- What if we also have to update a map of free blocks?
  1. Create new file header
  2. Point directory to new file header
  3. Update the free block map
- Does this work?
  - Bad if crash between 2 and 3
  - Free block map will still think new file header is free

# Multi-step updates

- What if we also have to update a map of free blocks?
  1. Create new file header
  2. Update the free block map
  3. Point directory to new file header
- Does this work?
  - Better, but still bad if crash between 2 and 3
  - Leads to a disk block leak
  - Could scan the disk after a crash to recompute free map
  - Older versions of Unix and Windows do this
  - (now we have journaling file systems ...)

# Careful ordering is limited

- Transfer \$100 from your account to mine
  - I. Deduct \$100 from your account
  2. Add \$100 to my account
- Crash between I and 2, we lose \$100
- Could reverse the ordering
  - I. Add \$100 to my account
  2. Deduct \$100 from your account
- Crash between I and 2, we gain \$100
- What does this remind you of?

# Atomic actions

- A lot like pre-emptions in a critical section
- Race conditions
  - Allow threads to see data in inconsistent state
- Crashes
  - Allow OS to see file system in inconsistent state
- Want to make actions **atomic**
  - All operations are applied or none are

# Atomic actions

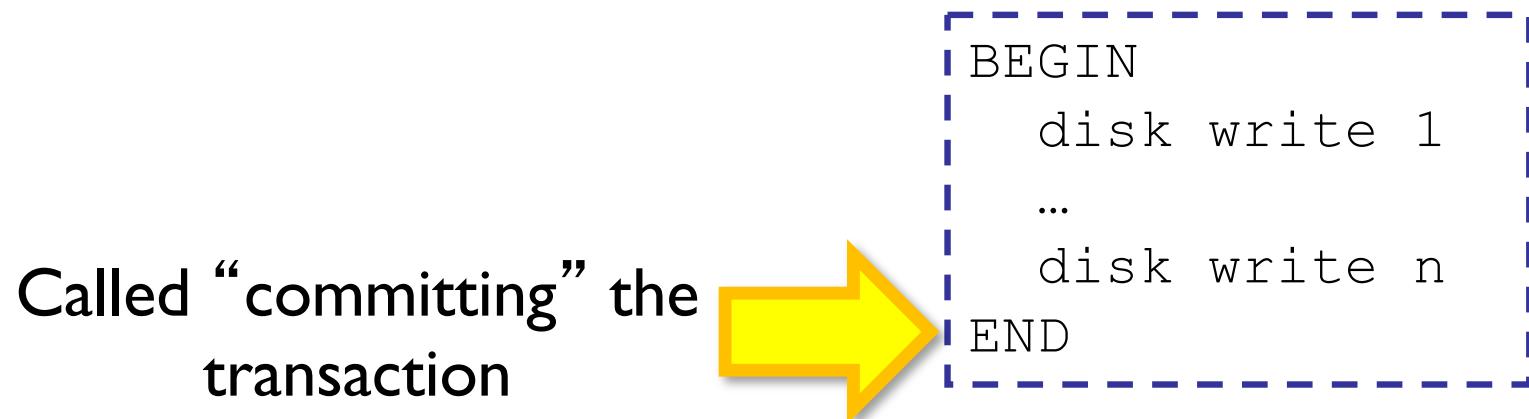
- With threads
  - Built larger atomic operations
    - (lock, wait, signal)
  - Using atomic hardware operations
    - (test and set, interrupt enable/disable)
- Same idea for persistent storage
  - **Transactions**

# Transactions

- Fundamental to databases
- Several important properties
  - “ACID” (atomicity, consistent, isolated, durable)
    - **Atomicity:** all or nothing
    - **Consistency:** DB is in valid state afterwards
    - **Isolated:** concurrent execution of transactions ok
    - **Durable:** persists after crashes

# Transactions

- Fundamental to databases
- Several important properties
  - “ACID” (atomicity, consistent, isolated, durable)
  - We will only talk about atomicity (all or nothing)



# Transactions

- Basic atomic unit provided by the hardware
  - Writing to a single disk sector/block
  - (not actually atomic, but close)
- How to make sequences of updates atomic?
- Two styles
  - Shadowing
  - Logging

# Transactions: shadowing

- Easy to explain, not widely used
  1. Update a copy version
  2. Switch the pointer to the new version
- Each individual step is atomic
- Step 2 commits the transaction
- Why doesn't anyone do this?
  - Double the storage overhead

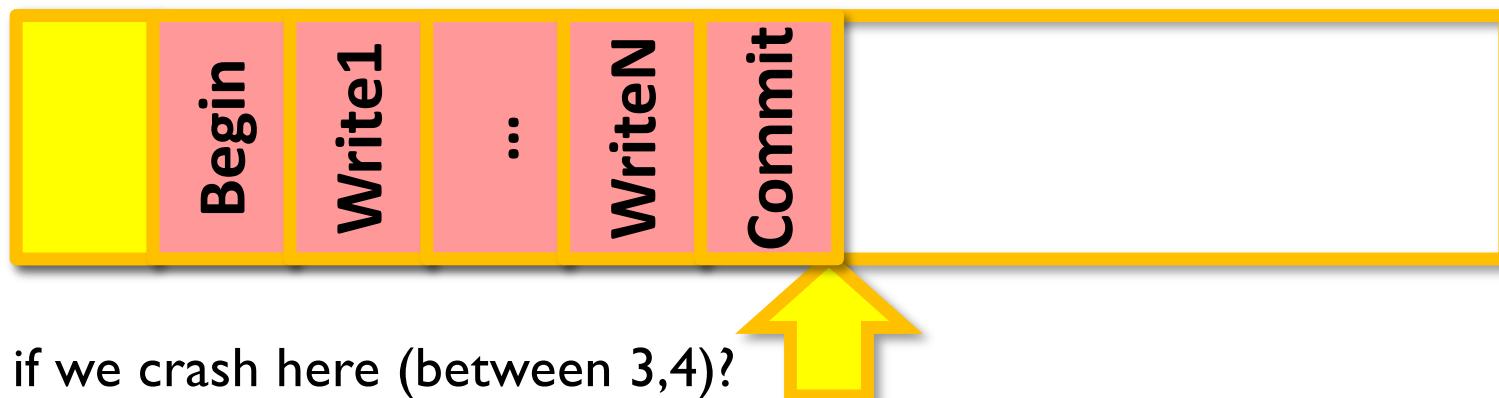
# Transactions: logging

1. Begin transaction
2. Append info about modifications to a log
3. Append “commit” to log to end x-action
4. Write new data to normal database
- Single-sector write commits x-action (3)



# Transactions: logging

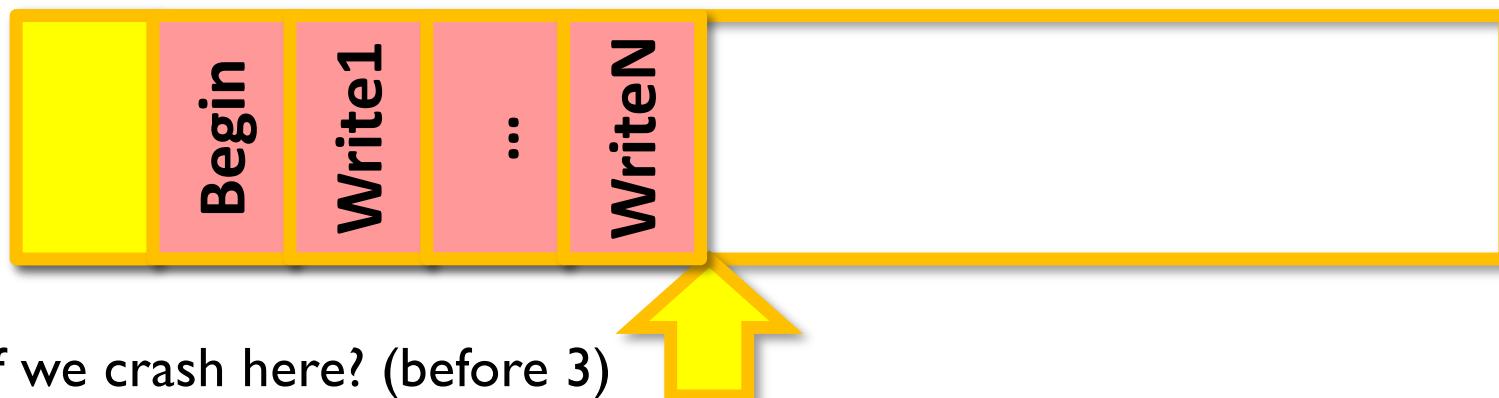
1. Begin transaction
  2. Append info about modifications to a log
  3. Append “commit” to log to end x-action
  4. Write new data to normal database
- Single-sector write commits x-action (3)



What if we crash here (between 3,4)?  
On reboot, reapply committed updates in log order.

# Transactions: logging

1. Begin transaction
  2. Append info about modifications to a log
  3. Append “commit” to log to end x-action
  4. Write new data to normal database
- Single-sector write commits x-action (3)



What if we crash here? (before 3)  
On reboot, discard uncommitted updates.

# Transactions

- Most file systems
  - Use transactions to modify only meta-data
- Why not use them for data?
  - Related updates are program-specific
  - Would have to modify programs
  - OS doesn't know how to group updates

# One last thing

- We have looked at the Unix file system design
- Many other designs possible
  - Example: log-structured file system

# Basic Problem

- Most file systems now have large memory caches (buffers) to hold recently-accessed blocks
  - Most reads are thus satisfied from this buffer cache
  - From the point of view of the disk, most traffic is write traffic...
- To speed up disk I/O, we need to make writes go faster
  - But disk performance is limited ultimately by disk head movement
  - With current file systems, adding a block takes several writes (to the file and to the metadata), requiring several disk seeks

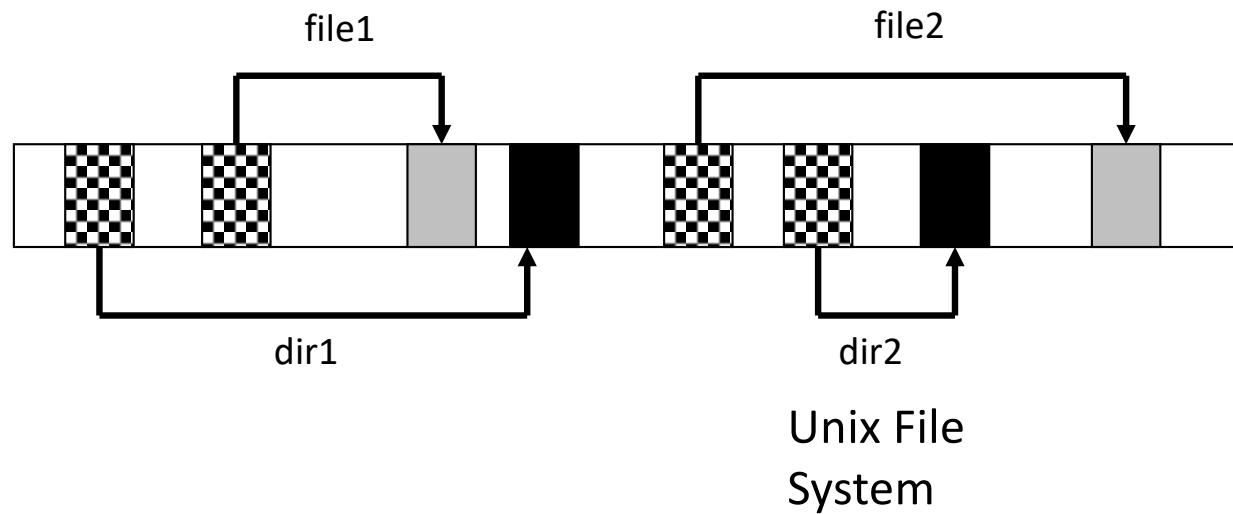
# LFS: Basic Idea

- An alternative is to use the disk as a *log*
  - A **log** is a data structure that is written only at the head
  - If the disk were managed as a log, there would be effectively no head seeks
  - The “file” is always added to sequentially
- New data and metadata (inodes, directories) are accumulated in the buffer cache, then written all at once in large blocks (e.g., segments of .5M or 1M)
  - This would greatly increase disk throughput
  - How does this really work? How do we read? What does the disk structure look like? etc.?

# LFS Data Structures

- **inodes:** as in Unix, contain physical block pointers for files
- **inode map:** table indicating where each inode is on disk
  - inode map blocks are written as part of the **segment**; a table in a fixed checkpoint region on disk points to those blocks
- **segment summary:** info on every block in a segment
- **segment usage table:** info on the amount of “live” data in a block

# LFS vs. Unix



# LFS: read and write

- Every write causes new blocks to be added to the current segment buffer in memory; when that segment is full, it is written to the disk
- Reads are no different than in Unix File System, once we find the inode for a file (in LFS, using the inode map, which is cached in memory)
- Over time, segments in the log become fragmented as we replace old blocks of files with new block
- **Problem:** in steady state, we need to have contiguous free space in which to write

# Cleaning

- The **major problem** for a LFS is ***cleaning***, i.e., producing contiguous free space on disk
  - A cleaner process “cleans” old segments, i.e., takes several non-full segments and compacts them, creating one full segment, plus free space
- The cleaner chooses segments on disk based on:
  - **utilization:** how much is to be gained by cleaning them
  - **age:** how likely is the segment to change soon anyway
- Cleaner cleans “cold” segments at 75% utilization and “hot” segments at 15% utilization (because it’s worth waiting on “hot” segments for blocks to be rewritten by current activity)

# LFS Summary

- Basic idea is to handle reads through caching and writes by appending large segments to a log
- Greatly increases disk performance on writes, file creates, deletes, ....
- Reads that are not handled by buffer cache are same performance as normal file system
- Requires cleaning daemon to produce clean space, which takes additional CPU and I/O

- **Final Exam on May 25<sup>th</sup> (Thursday)**
- Covers lecture 12 and after
- Address spaces and multiprogramming
  - User vs. kernel threads
  - Dynamic address translation
  - Page replacement algorithms
  - Hardware page tables
  - Kernel/user mode and system calls
  - Thrashing and working sets
  - Virtual machines
  - Capabilities and access control lists
  - Common attacks (buffer overflow, hidden channel, trojan horse)
  - I/O device interfaces
  - Disk scheduling and file systems

