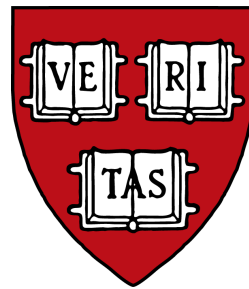


CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 12: The Linux Virtual Memory System
March 20, 2007

Today: The Linux VM System

As of kernel version 2.4.20

- 2.6 kernels have a very different VM implementation!

Focus on Linux x86 implementation

- Most parts are identical across architectures

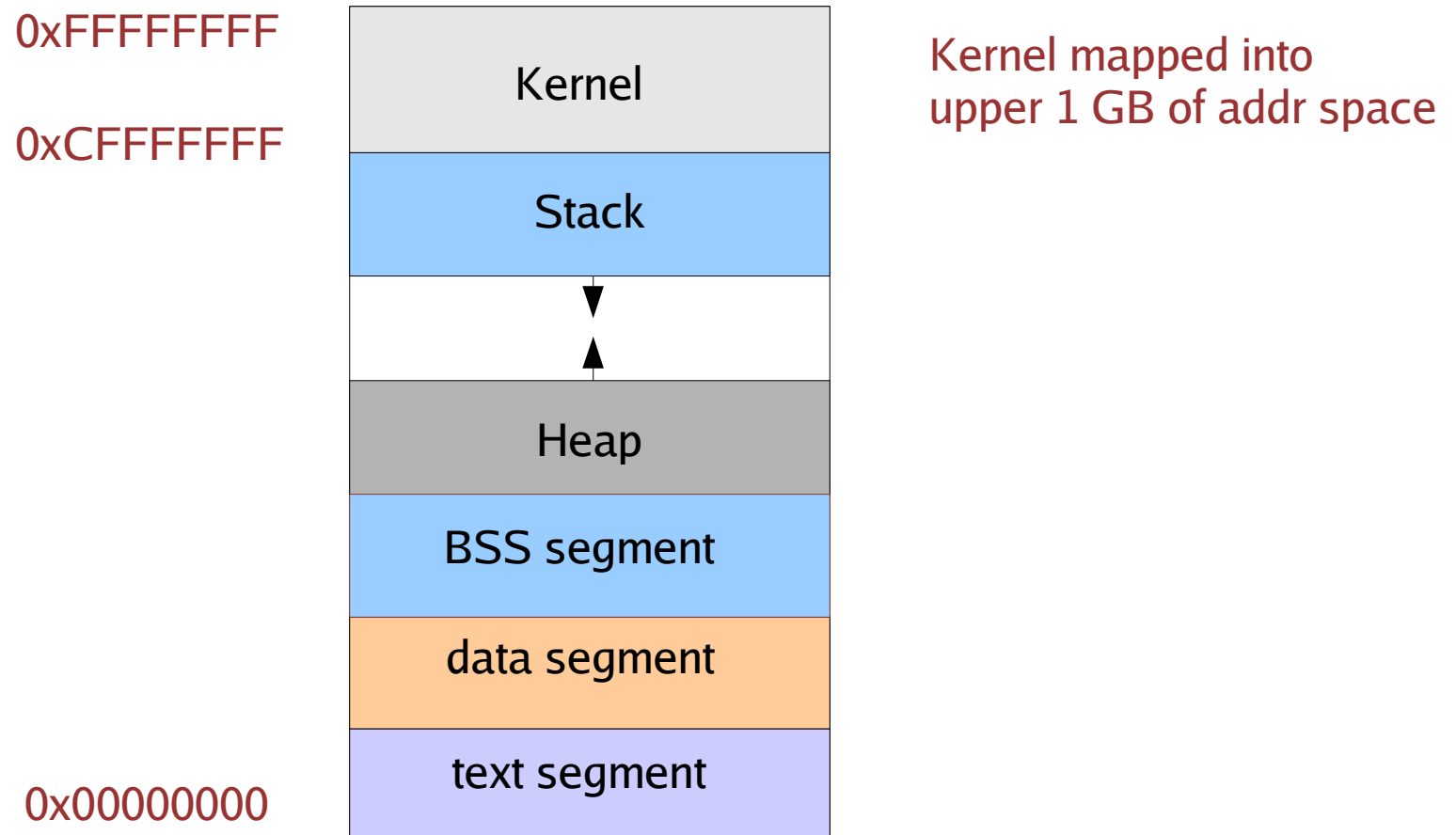
Overview of x86 virtual memory architecture

- Complex stuff ... combination of multi-level page tables and segmentation

Caveat: This is not definitive!!!

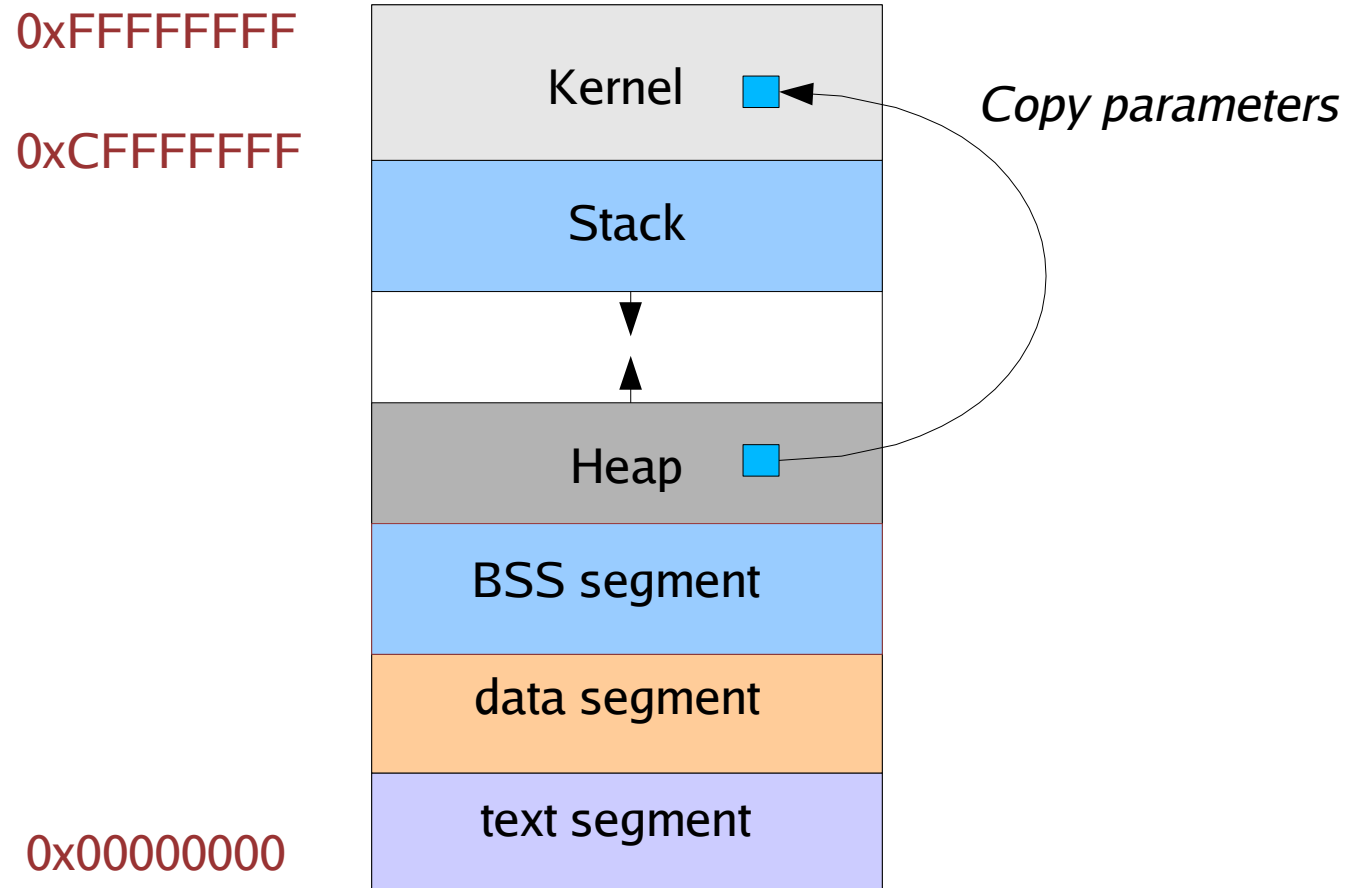
- I am glossing over many details
- I may have some things wrong

Process's Address Space



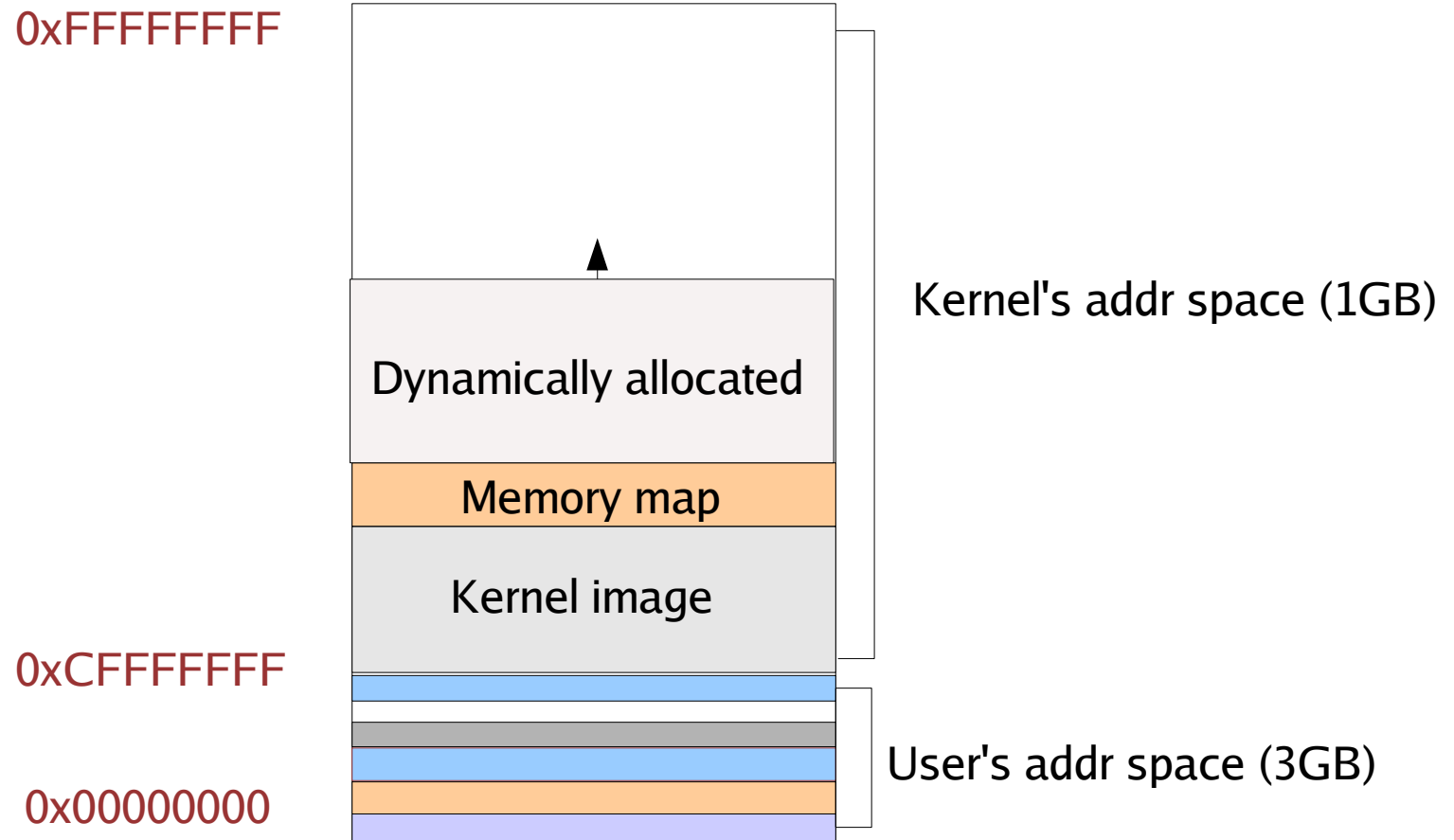
Why does the kernel appear in the process' address space??

Process's Address Space



- Kernel needs to access user data (e.g., syscall parameters)
- Idea: Kernel uses same address space as currently running user process.
- This allows kernel to read/write user data just using pointers.
 - *What is another benefit of this approach?*

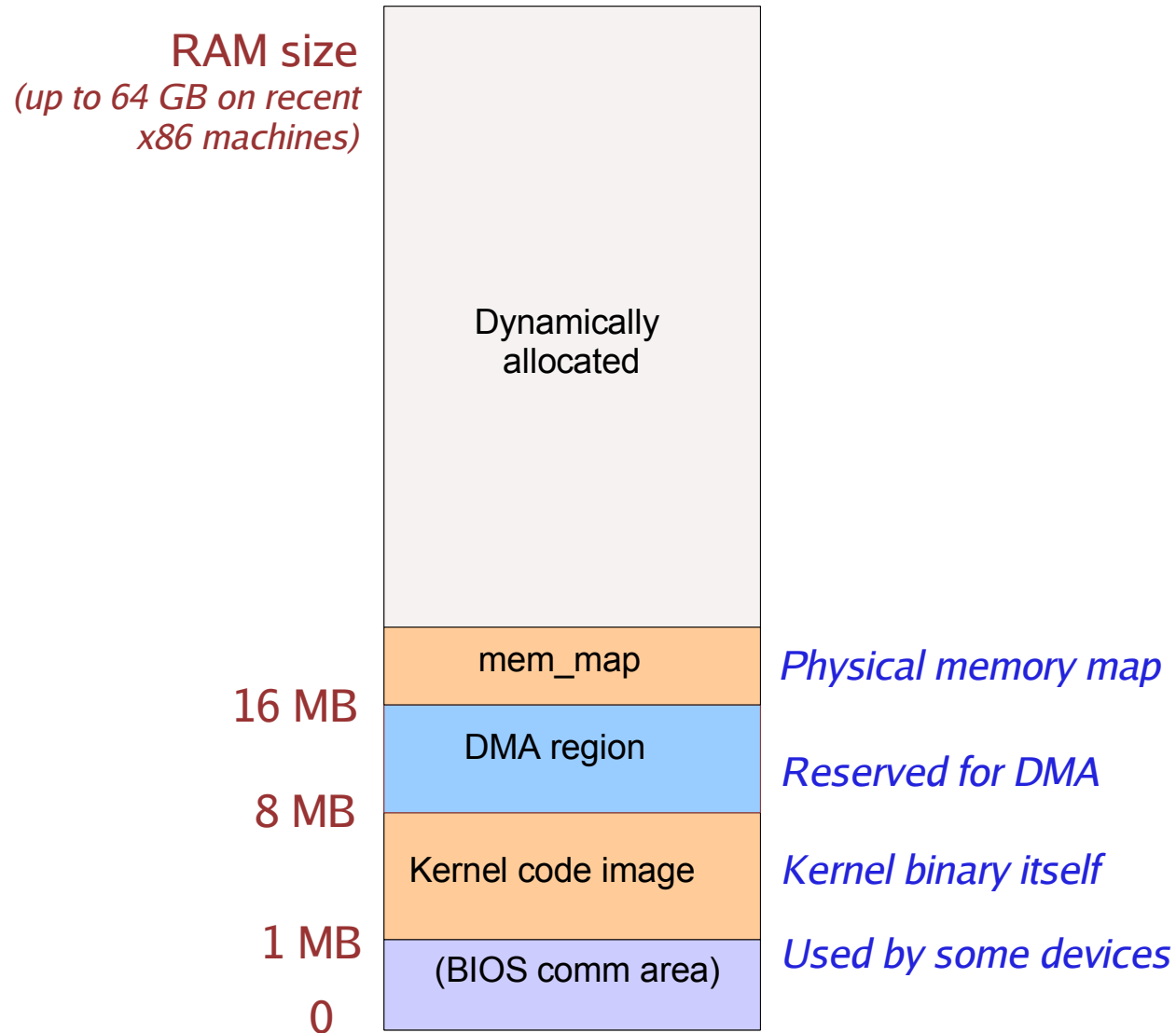
Kernel's view (not to scale...)



Of course, this means the kernel has its own page tables!

- The kernel uses *virtual* memory addresses just like user processes.
- But, most of the kernel's virtual address space is **pinned** to physical RAM

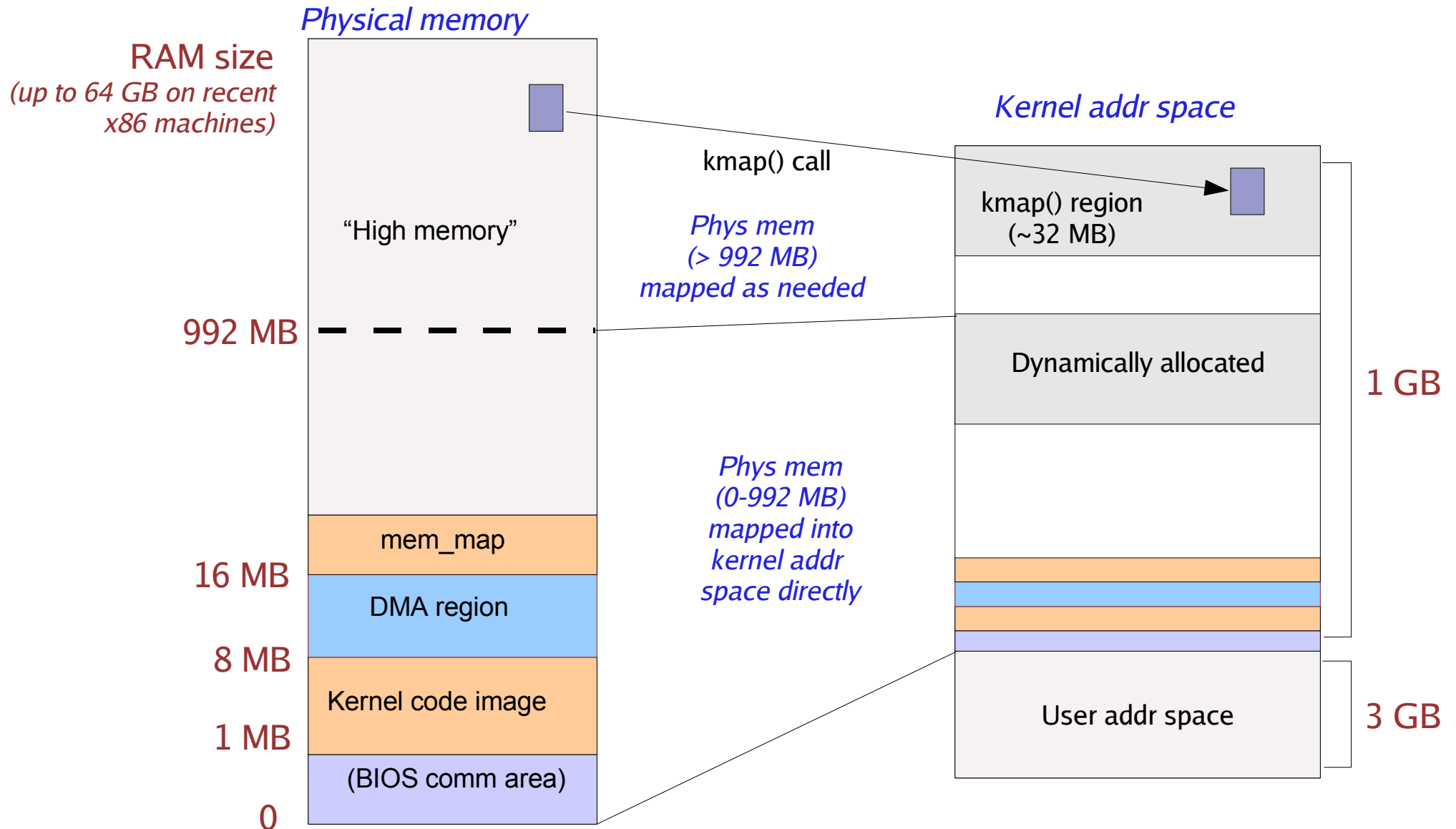
Physical Memory Layout



Recall: Kernel only has 1GB of addr space for itself.

- What does this imply??

High Memory Mapping

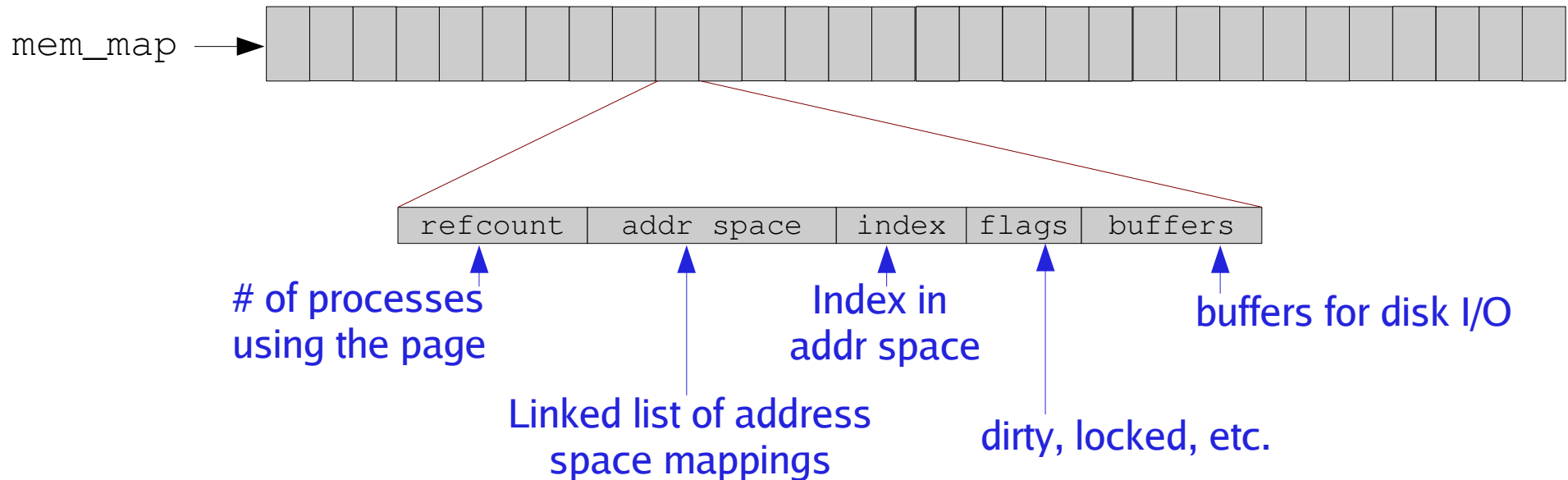


- Kernel must **map** any phys memory > 992 MB into it's addr space when needed!
- Very small amount of addr space (32MB) for mapping high memory ... means the kernel should not leave things mapped in there for long.

Physical Memory Map

Linux maintains a global array, `mem_map`, consisting of one entry for each **physical page** in the system

- Each page represented by a `struct page`



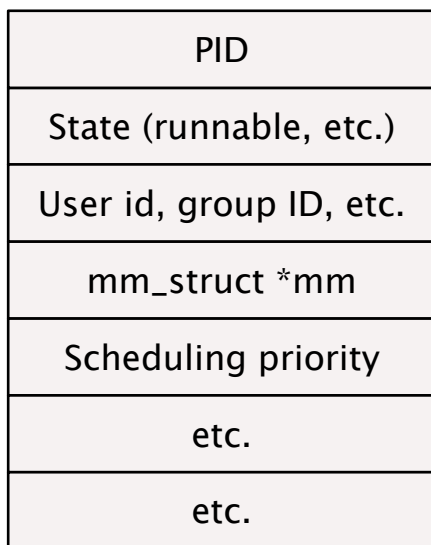
Lots of state maintained for each page!

- `struct page` is 44 bytes!
- So if we have 2GB of RAM, that's $(2\text{GB} / 4\text{KB}) = (512 \text{ K})$ pages, or **22 MB of space** just for the memory map!!

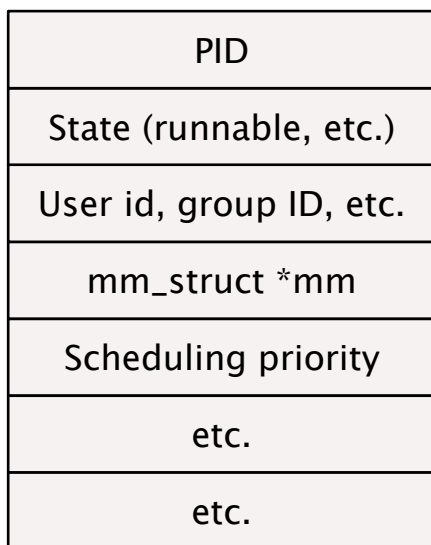
VM Structures Overview

One per thread

task_struct

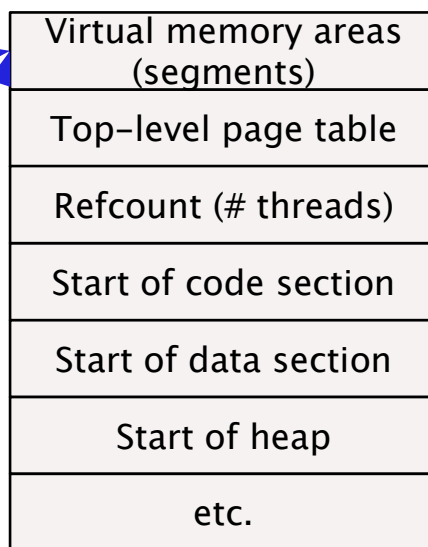


task_struct



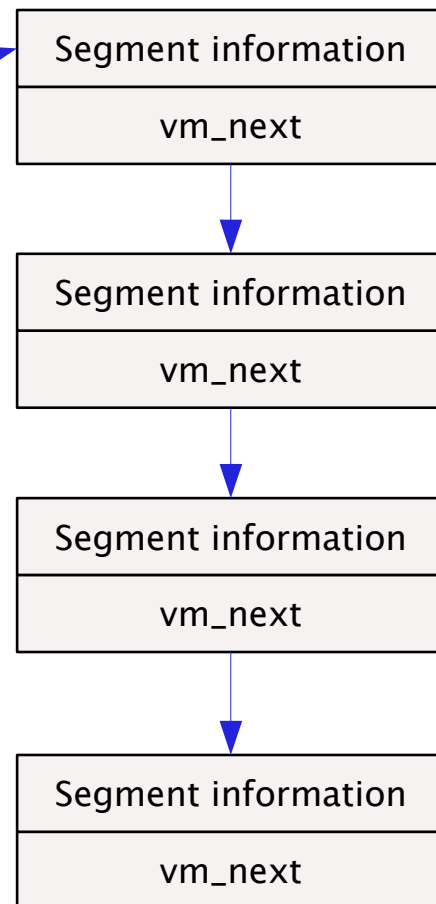
One per addr space

mm_struct



One per "segment"

vm_area_struct

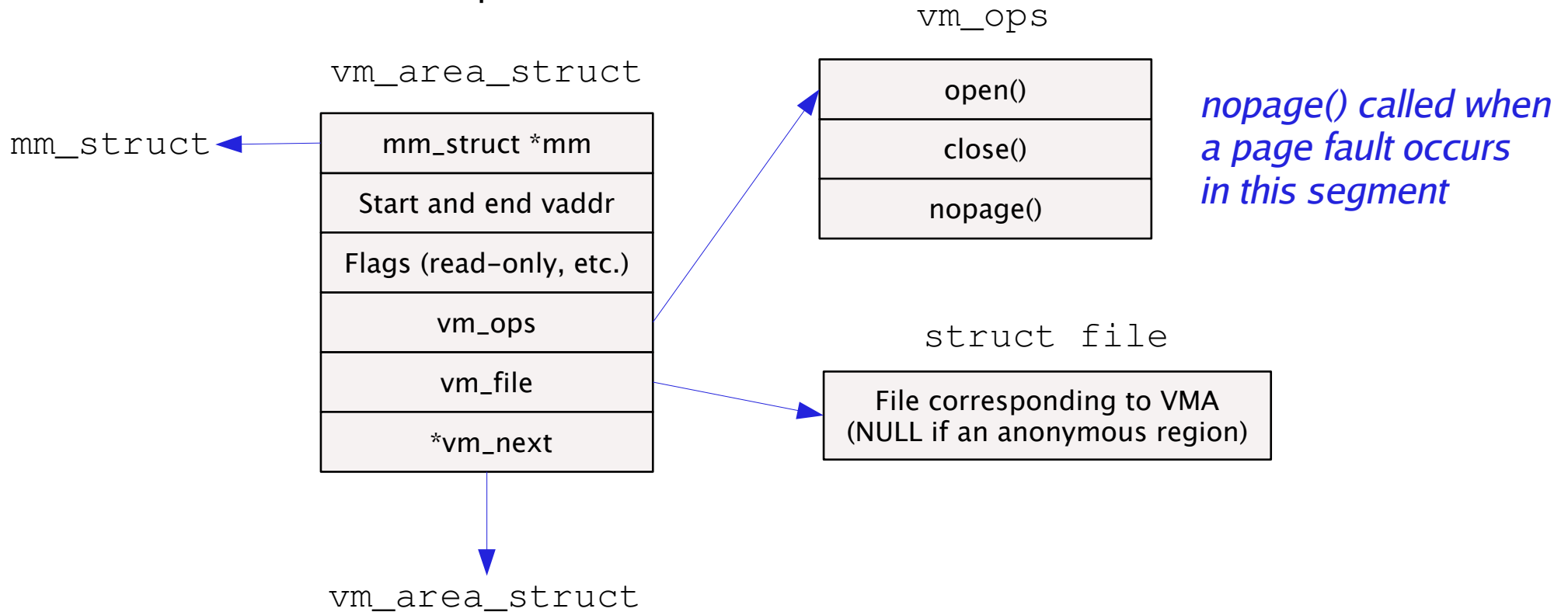


Two threads are in the same process if they have the same "mm" field in their task_struct!

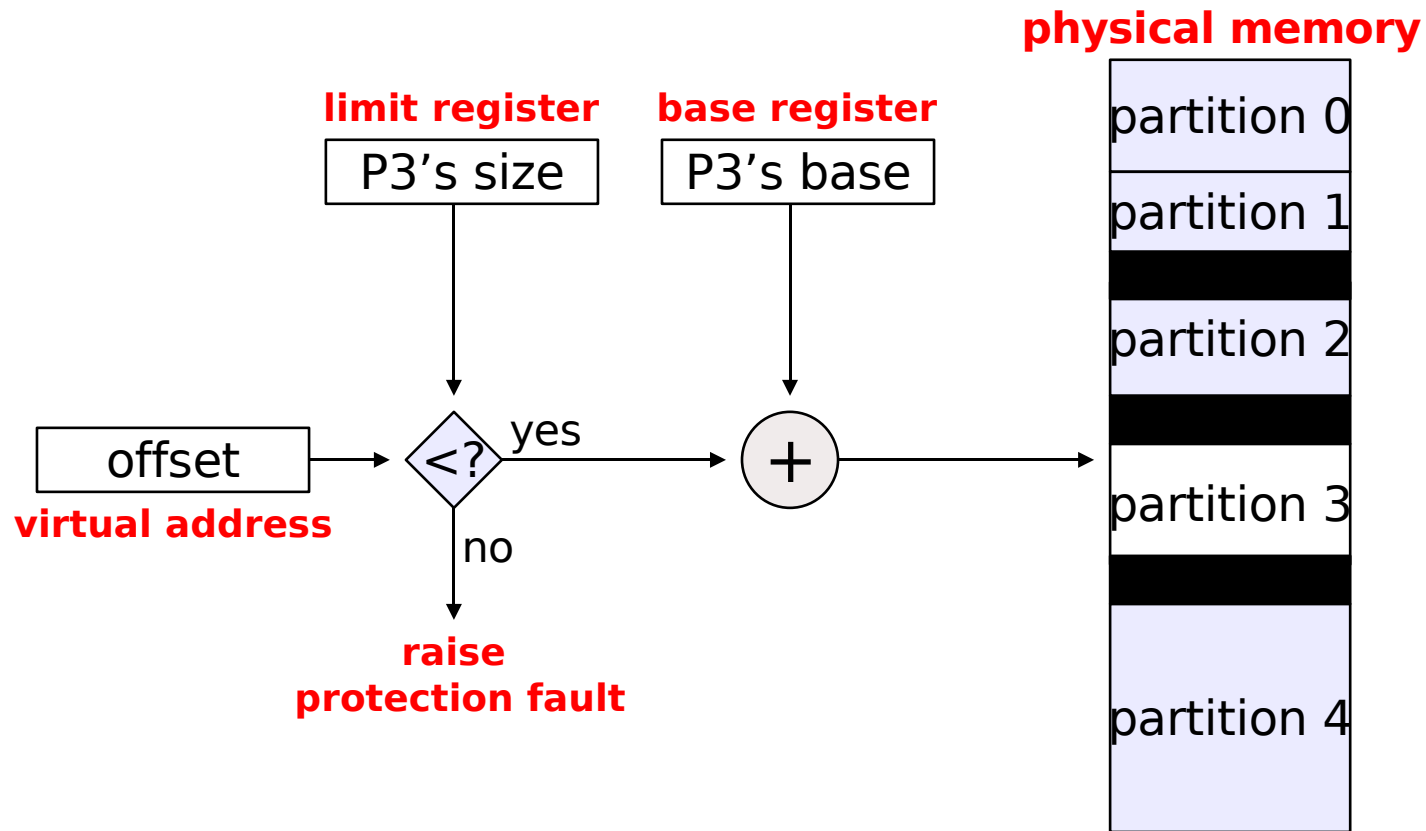
vm_area_struct

One vm_area_struct per segment in the address space

- The list of VMAs comprises the entire address space
- VMAs cannot overlap



Recall Segmentation...



Virtual address is $\langle \text{segment \#}, \text{offset} \rangle$ pair

- Each segment has a base address and total size (limit register)

Problem with segmentation: external fragmentation

- Holes left in physical memory when segments are destroyed

Segmentation and Paging

Can combine segmentation and paging!

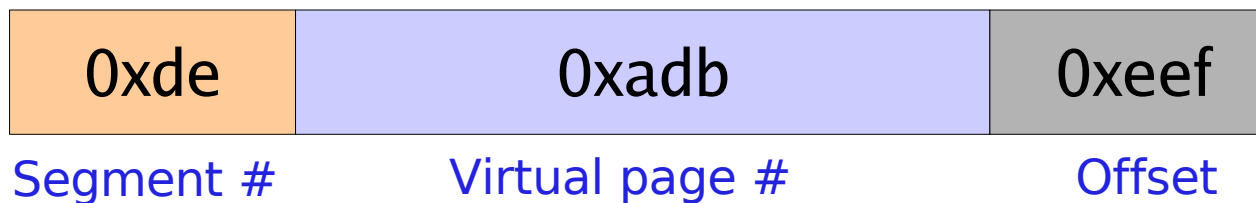
A segment is a contiguous span of *virtual addresses*

- ... rather than physical addresses, as on the previous slide
- Described by a *segment descriptor*
- Segment descr has total segment size, access rights, base virtual address

Each segment can have a corresponding page table!

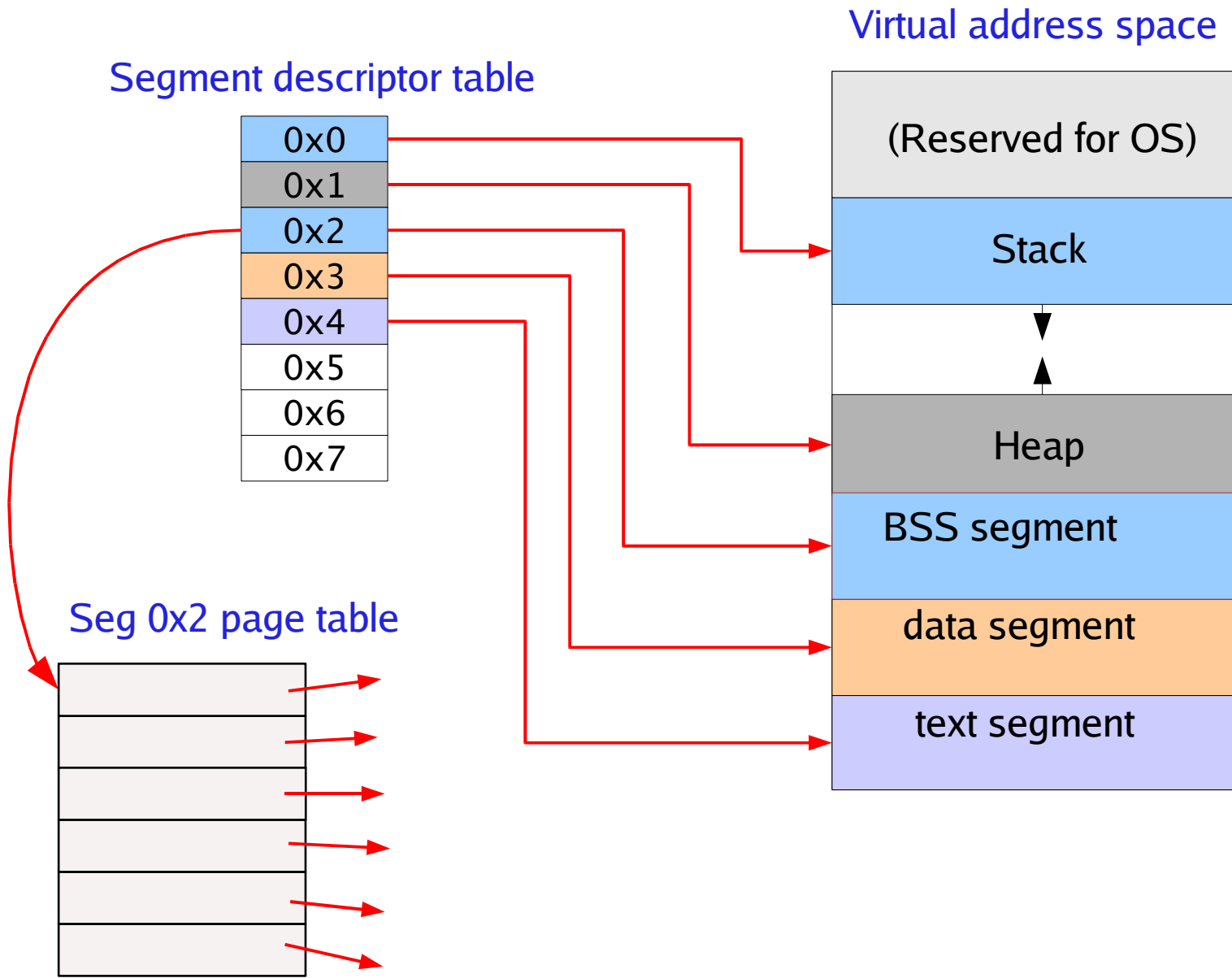
- Segment broken into pages internally
- Can use either one- or two-level paging on each segment

Virtual address now looks like:



- *Segment number* may be part of the address, or stored in a separate register
- Value of current segment register used to determine which segment to access

Virtual Address Space with Segments



Why use segments?

Segments cleanly separate different areas of memory

- e.g., Code, data, heap, stack, shared memory regions, etc.
- Use different segment registers to refer to each portion of the address space

Allows hardware to enforce protection on a segment as a whole

- e.g., Segment descriptor can mark the entire code segment as read only

Note that using page tables can accomplish the same result...

- But requires the OS to carefully maintain page table entries for entire “segments”

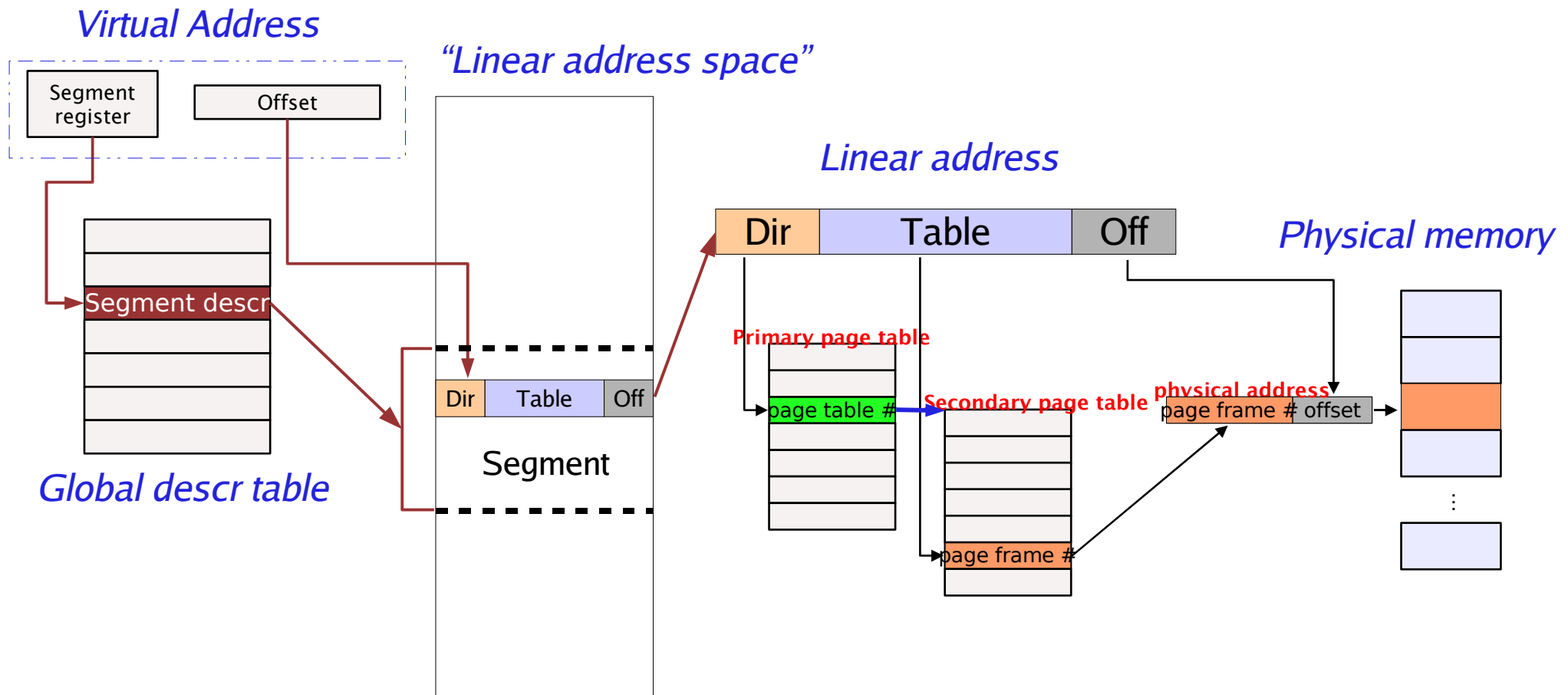
Most operating systems use “one big segment”

- Linux x86: One segment for user code, another segment for user data
- Both segments cover the same virtual address range! (0 ... 3GB)
- Another pair of segments for kernel virtual addresses (3GB ... 4GB)

Intel x86 VM Architecture

Really impressive, and complicated, memory model!

- Combines segments, multi-level paging, multiple protection rings, and more
- Anecdotally, x86 was designed to support a rich VM system like Multics



Intel x86 Segments

Multiple *segment registers*

- CS: Code Segment
- DS: Data Segment
- SS: Stack Segment
- Also ES, FS, and GS ... “other” segments

Each instruction uses one of these segment registers

- For example, instruction fetch implicitly uses segment pointed to by CS
- Push/pop instructions implicitly use segment pointed to by SS

Segment descriptor information:

- Virtual base address and size of segment
- Segment access rights (read, write, execute)
- Privilege level (0 through 3 – four separate “rings”)

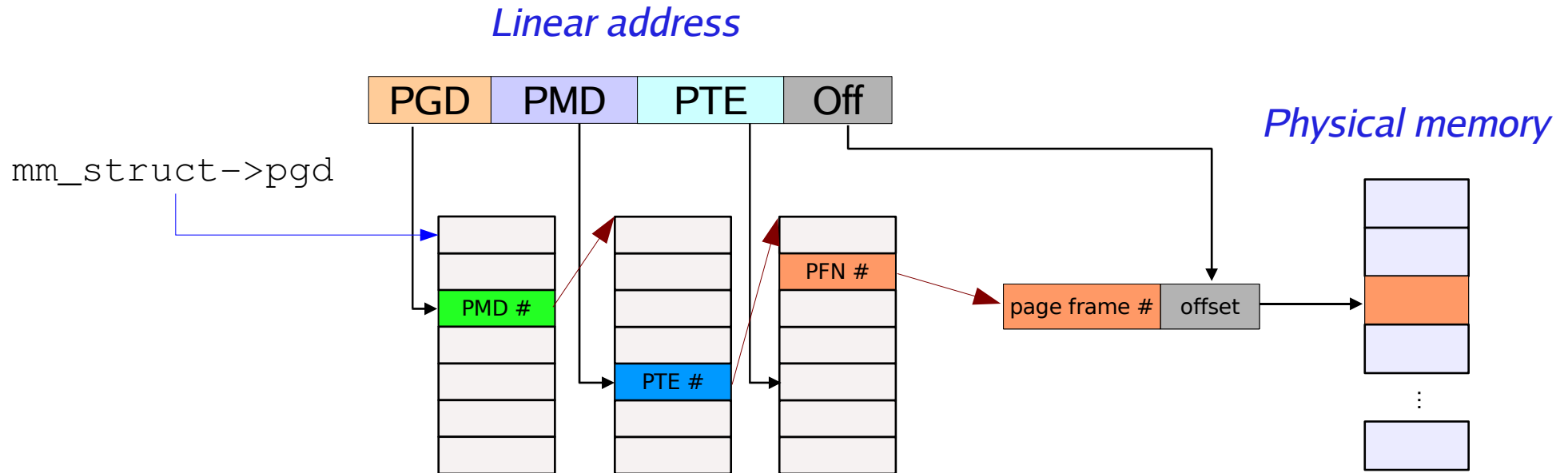
All segments share the same linear address space!

- This means there is **one set of page tables** for all segments in a process
- Segments can overlap in linear address space, too.

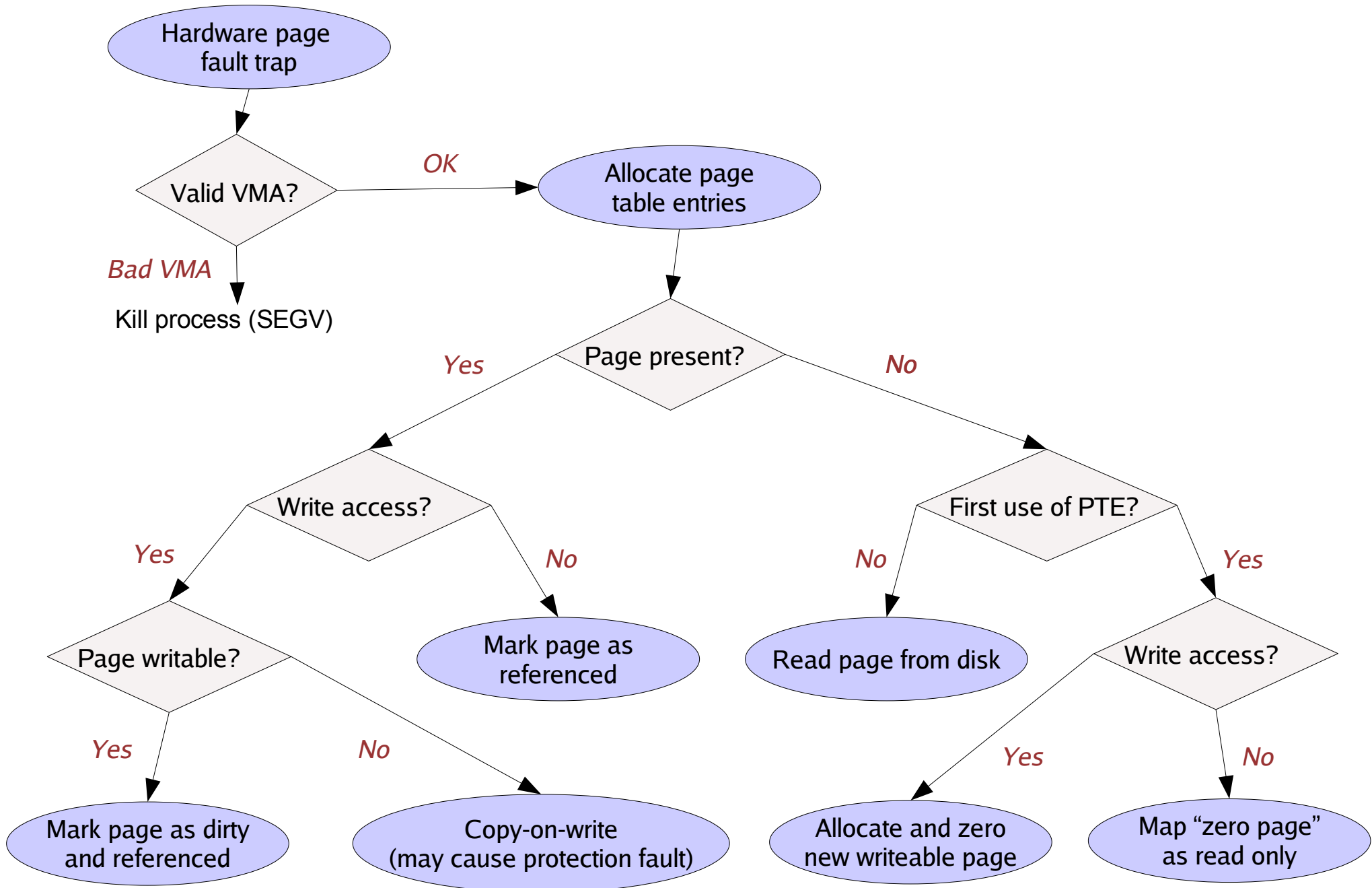
Linux page table structure

Linux supports **three-level** page tables in software

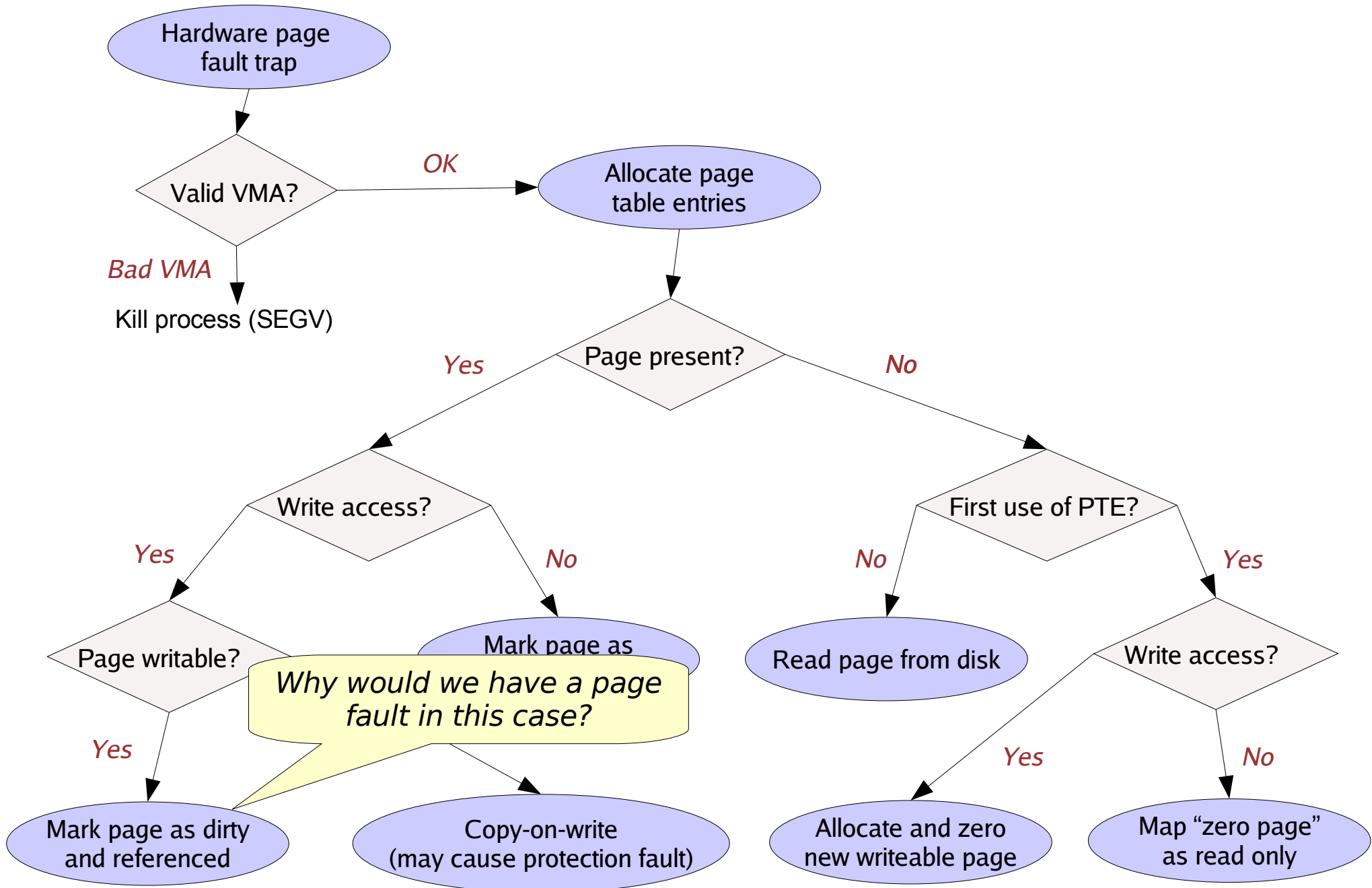
- These map onto two-level page tables on the x86 and most other architectures
- (The code for this is pretty ugly...)



Page Fault Handling

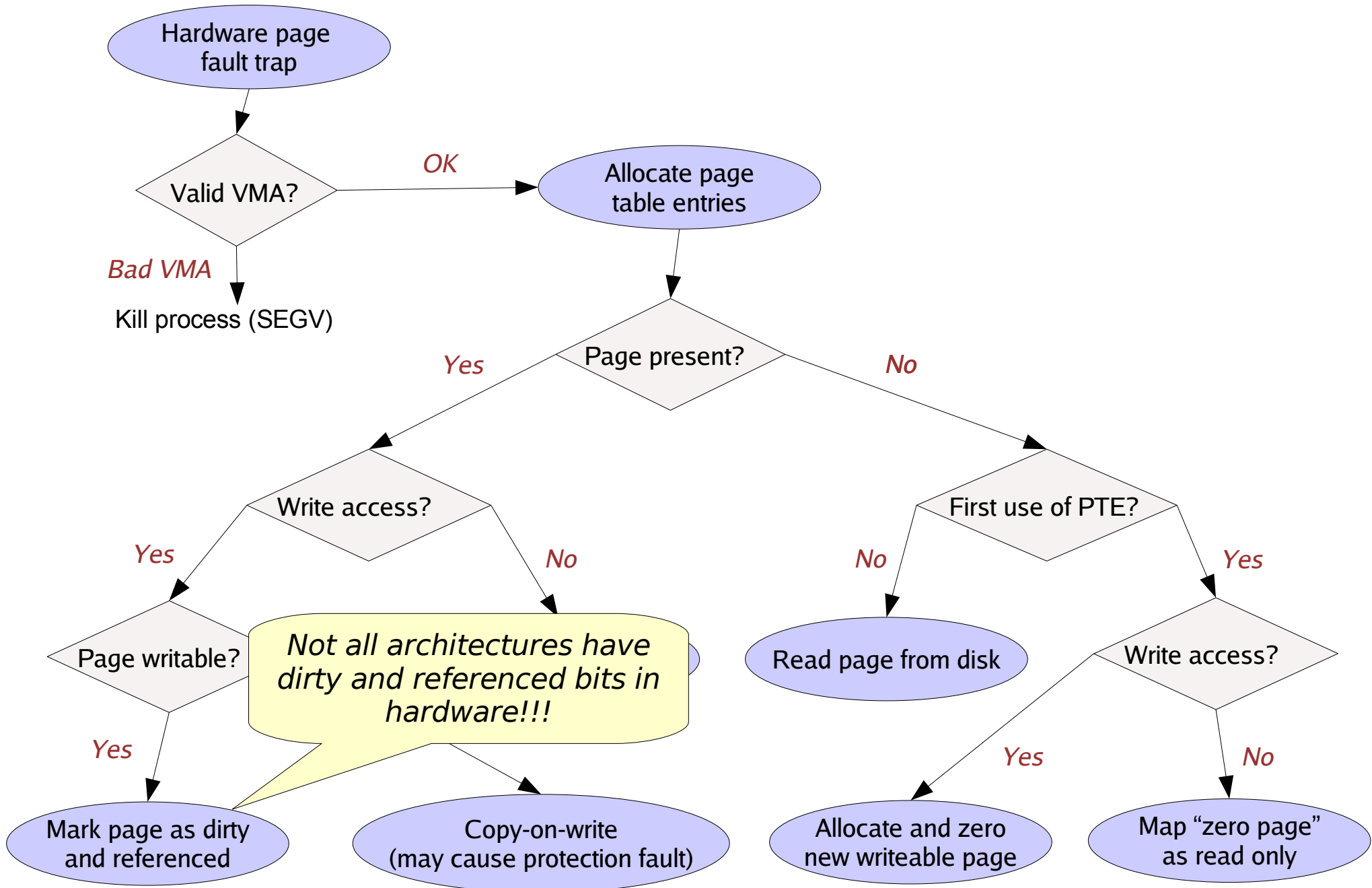


Page Fault Handling



Why would we have a page fault in this case?

Page Fault Handling



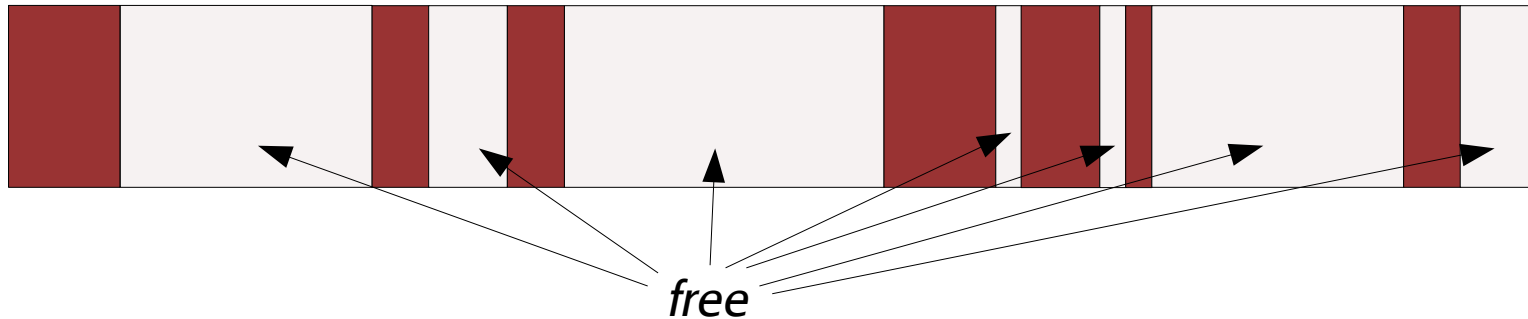
Reclaiming Page Frames

kswapd: Kernel pageout daemon

- Thread that executes in the kernel to free up physical pages

Each physical memory “zone” maintains a count of free pages

- Zones correspond to DMA memory, “main” memory (up to 1GB), and “high” memory (above 1GB)
- When free page count falls below a threshold, kswapd is woken up



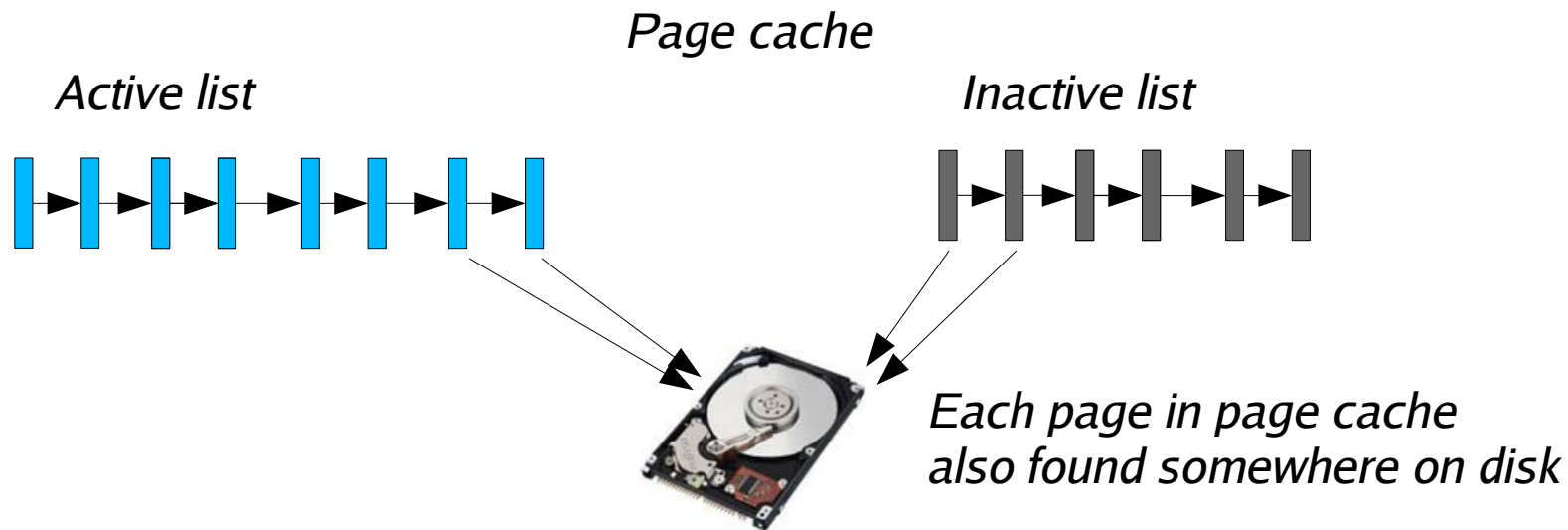
Reclaiming Page Frames

Kernel maintains a “page cache”

- Set of physical pages corresponding to blocks on disk
 - *Either files or swap space*
- Before doing any disk I/O, always check page cache for the page

Page cache has two lists of pages: “active” and “inactive”

- “Active” pages have been used recently
- “Inactive” pages have not been used recently and may be swapped out



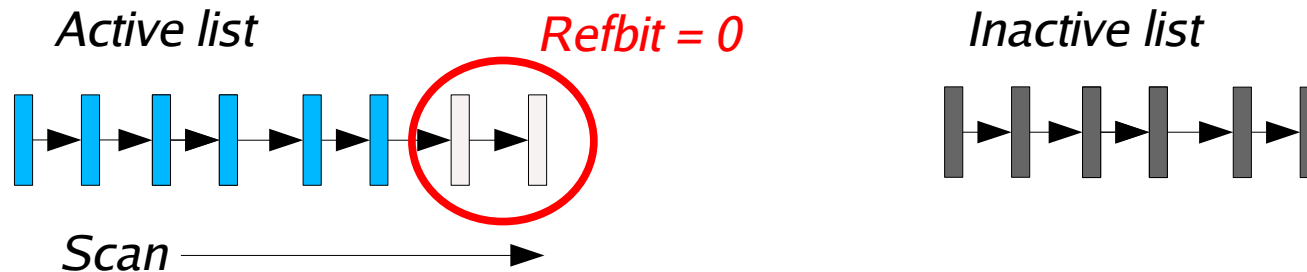
Shrinking the page cache

To reclaim physical memory, first try to shrink the page cache

- kswapd has a target for the number of pages to free up
- If page has reference bit set to 0, move it to the “inactive” list
- Otherwise, move it to the front of the “active” list
 - *This is essentially the Clock algorithm!*

Next step: Decide which pages in the inactive list to swap out

- Tries to tune the number of pages to swap out to minimize disk I/O
- If not enough pages freed in one call, try again with a higher target
- Likewise, if enough pages freed in one call, lower the target



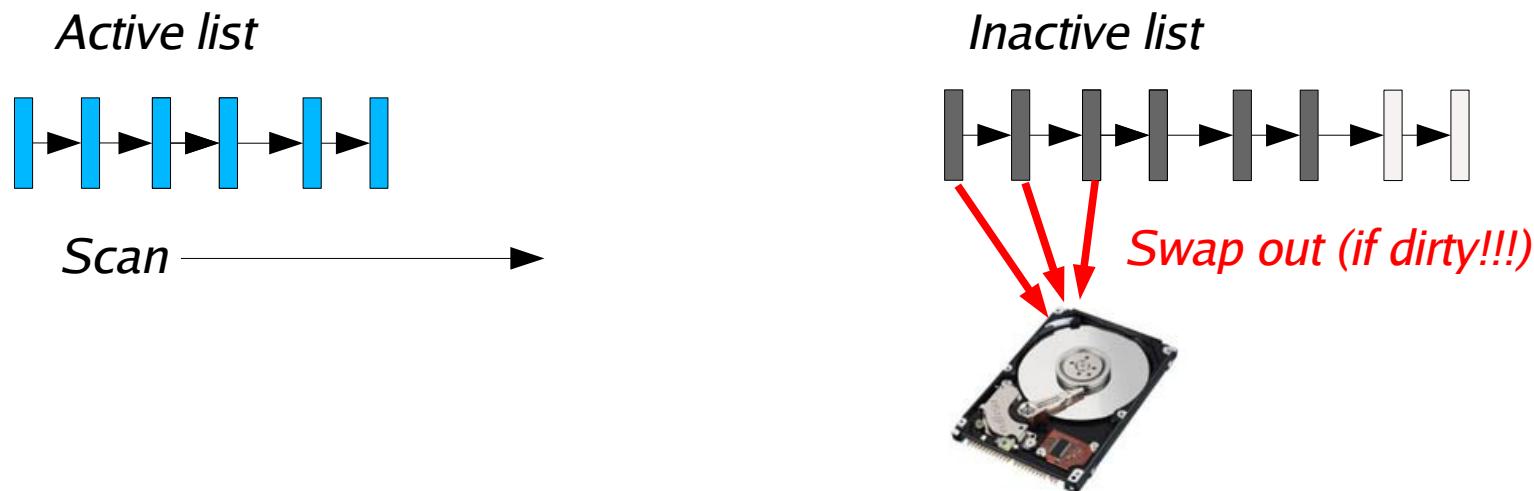
Shrinking the page cache

To reclaim physical memory, first try to shrink the page cache

- kswapd has a target for the number of pages to free up
- If page has reference bit set to 0, move it to the “inactive” list
- Otherwise, move it to the front of the “active” list
 - *This is essentially the Clock algorithm!*

Next step: Decide which pages in the inactive list to swap out

- Tries to tune the number of pages to swap out to minimize disk I/O
- If not enough pages freed in one call, try again with a higher target
- Likewise, if enough pages freed in one call, lower the target



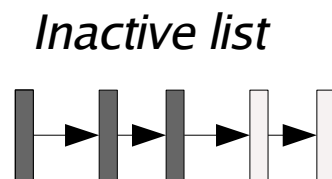
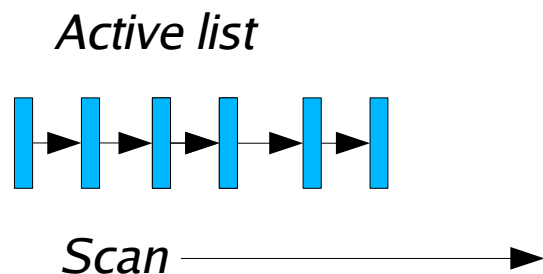
Shrinking the page cache

To reclaim physical memory, first try to shrink the page cache

- kswapd has a target for the number of pages to free up
- If page has reference bit set to 0, move it to the “inactive” list
- Otherwise, move it to the front of the “active” list
 - *This is essentially the Clock algorithm!*

Next step: Decide which pages in the inactive list to swap out

- Tries to tune the number of pages to swap out to minimize disk I/O
- If not enough pages freed in one call, try again with a higher target
- Likewise, if enough pages freed in one call, lower the target



Reclaiming non-page-cache memory

Page cache is always the first to get swapped out

- In some sense, it's just an optimization...

What happens if the page cache is empty?

First, shrink other kernel memory in a similar manner

- e.g., Special caches for file pages, kernel VM allocator, and others
- Again, these are all mainly optimizations.

Only if we can't free enough memory from these caches is process memory freed up!

- Scan over process page tables and try to free up inactive pages

Out of memory management

Even after all of this, it is still possible we need to free some memory.

- What can we do?

Out of memory management

Even after all of this, it is still possible we need to free some memory.

- What can we do?

Linux resorts to killing processes until it can free some memory.

`select_bad_process()`: Find a “bad” process and kill it

- For each process, calculate:

$$\text{Badness} = \frac{\text{total_vm_size}}{\sqrt{\text{cpu_time_used in seconds}} + \sqrt{\sqrt{\text{wall_clock_time in minutes}}}}$$

Kill process with highest badness score.

- “nice” processes are given double badness points
- Procs run by root are given $\frac{1}{4}$ badness points
- Pick procs with a lot of memory but not long-lived ones (examples????)

Lesson: Seemingly important OS policies are often black magic.