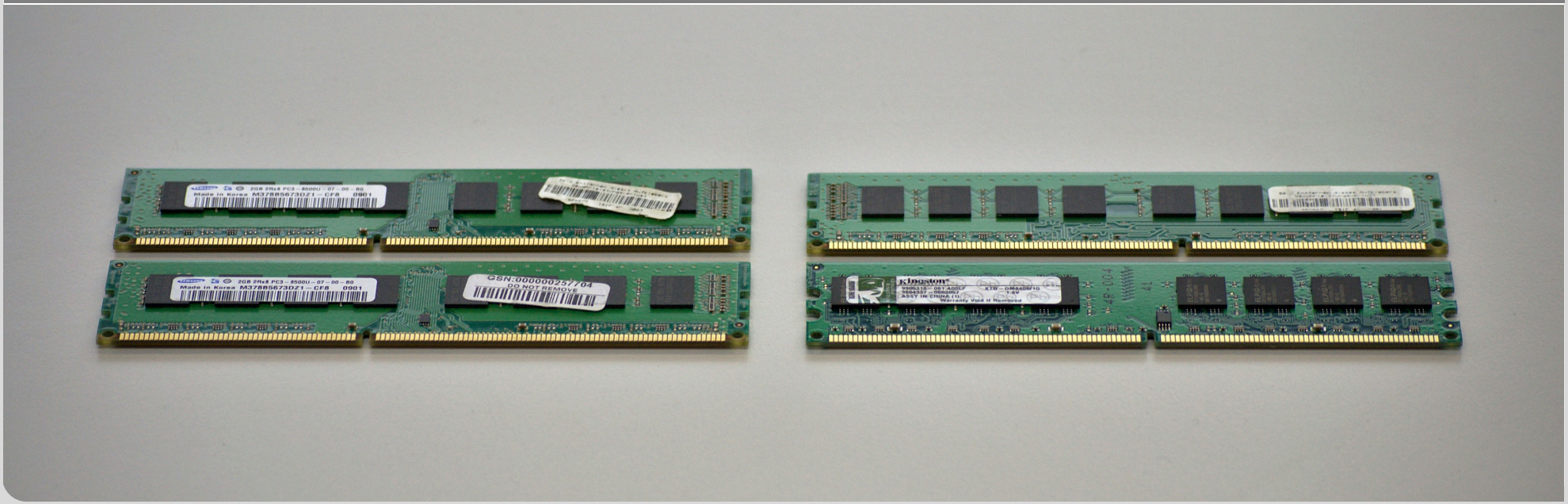
Karlsruhe Institute of Technology

Betriebssysteme

09. Memory Management Hardware

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP



We studied Processes and Address Spaces before

- Processes are a resource container for the OS
- The process feels alone in the world
  - It has its own memory (address space, AS) and only uses virtual addresses
  - The MMU relocates each load/store to physical memory
  - Processes never see physical memory and cannot address it directly
- The process AS layout is given by the ABI
- The layout is divided into sections
  - Statically allocated data (text, ro-data, data, bss)
  - Data that follows LIFO semantics (stack)
  - Data that can be allocated and free'd dynamically (heap)

Typical Process Address Space Layout

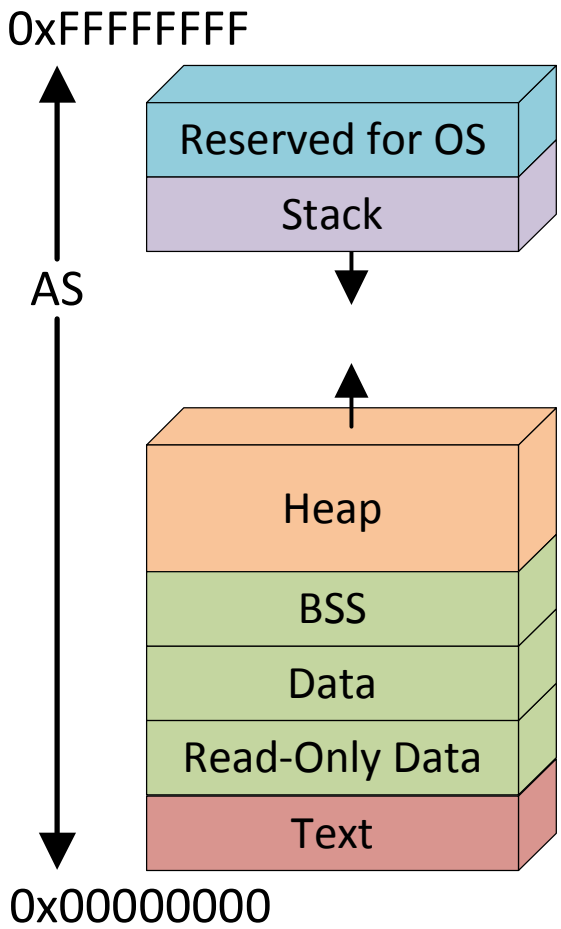
OS Addresses where the kernel is mapped (cannot be accessed by process)

Stack Local variables, function call parameters, return addresses

Heap Dynamically allocated data (`malloc`)

Data Static, constant, global variables

Text Program, machine code



- This week: How to translate between virtual and physical addresses
- Next week: How the OS organizes and provisions memory for multiple processes

Side note: Physical address translates to “physischer Speicher” (körperlich)

Do not say “physikalischer Speicher” (die Physik betreffend)

Memory Managment Hardware

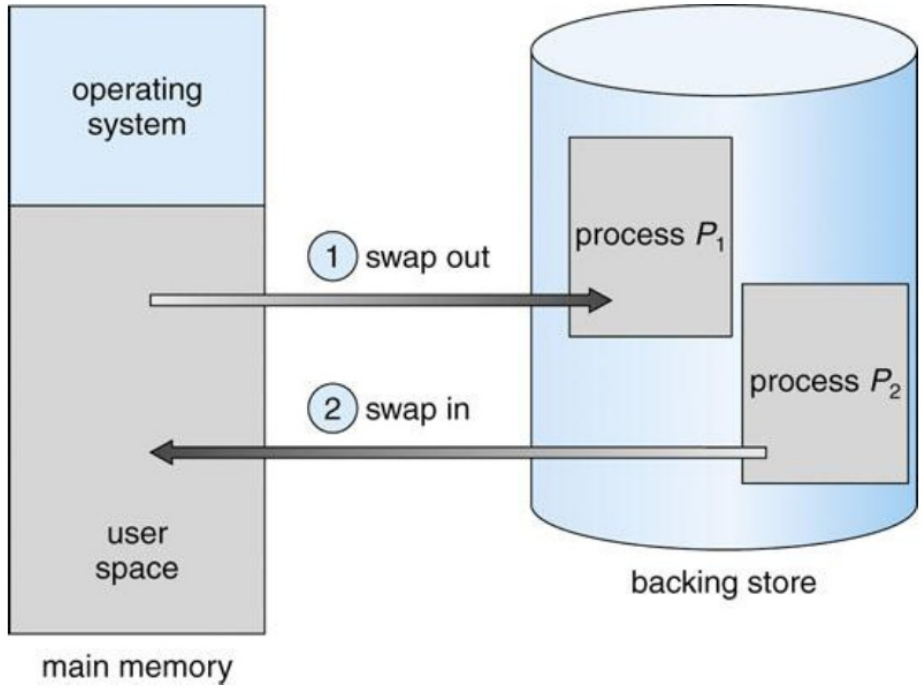
How to translate addresses

Main Memory

- Main memory and registers are the only storage that the CPU can access directly
- Program must be brought into memory from background storage and placed within a process' address space for it to be run
- Early computers had no memory abstraction
  - Programs accessed physical memory directly
- Multiple processes can be run concurrently even without memory abstraction
  - Swapping
  - Static Relocation

Swapping

- Swapping denotes
  - saving a program's state on background storage (roll-out)
  - replacing it with another program's state (roll-in)



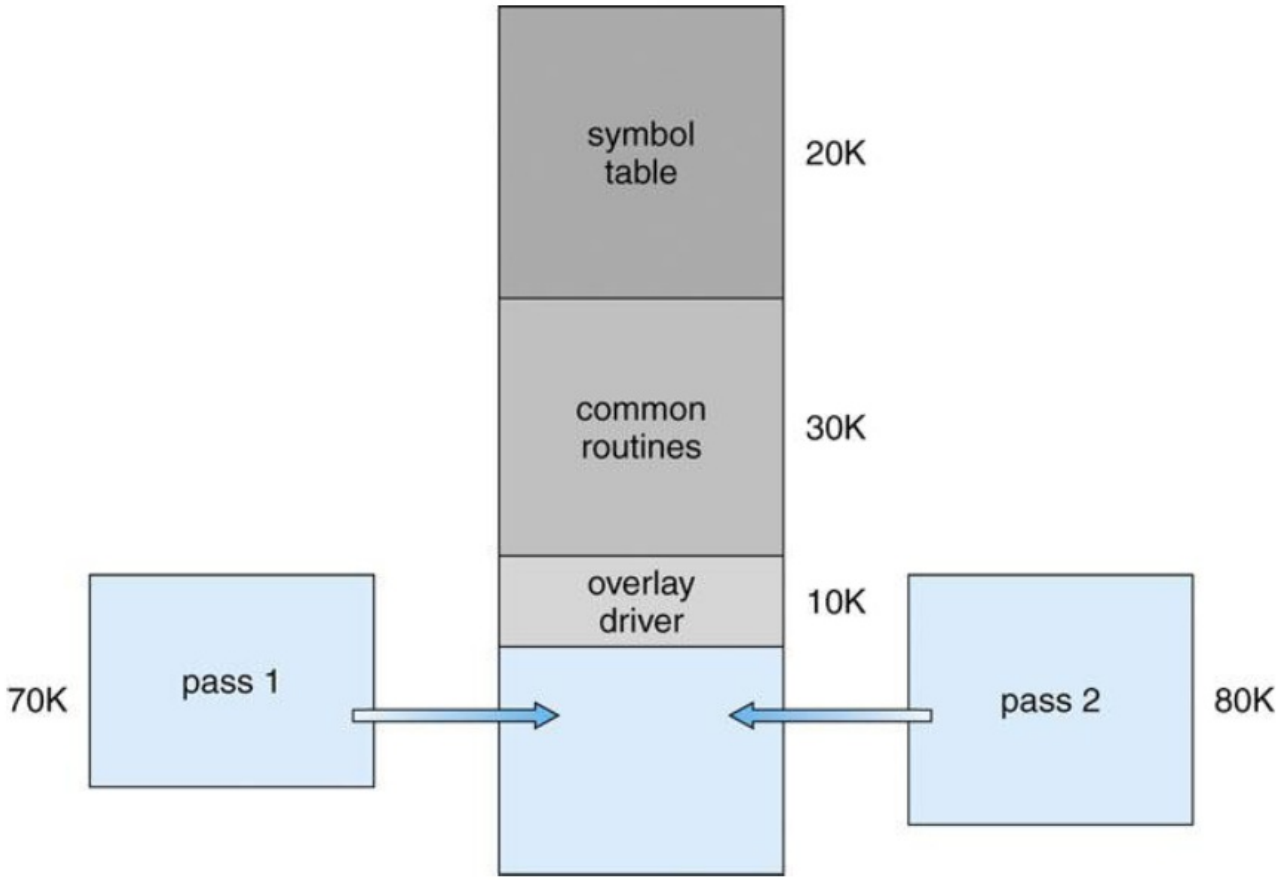
\+ Only needs hardware support to protect the kernel, but not to protect processes from one another

– Very slow
  - Major part of swap time is transfer time
  - Total transfer time is directly proportional to the amount of memory swapped

– At every point in time only one process runs: no parallelism
  - This process owns the entire physical address space

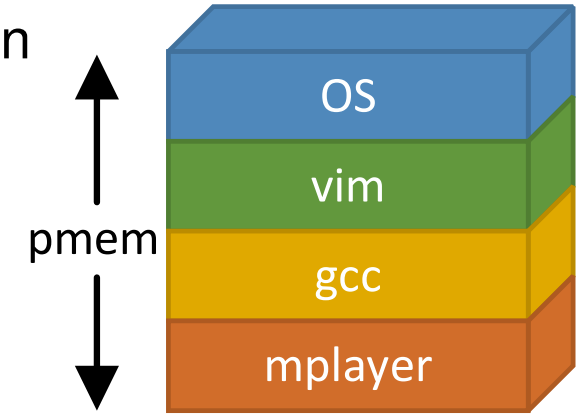
Overlays

- What if the process you want to run needs more memory than available?
- Need to partition program manually



Static Relocation

- Another possibility to solve address conflicts when loading multiple processes is static relocation
  - The OS adds a fixed offset to every address in a program when loading it and creating a process from it



- Same address space (physical addresses) for every process
  - No protection: Every program sees and can access every address!
  - What if gcc needs more memory for its abstract syntax tree?
  - What if mplayer is pausing playback and currently doesn't need memory? Can it be reused by other processes?
  - What if no contiguous free region fits program?

Want programs to co-exist peacefully.
Need to provide dynamic allocation and mutual protection!

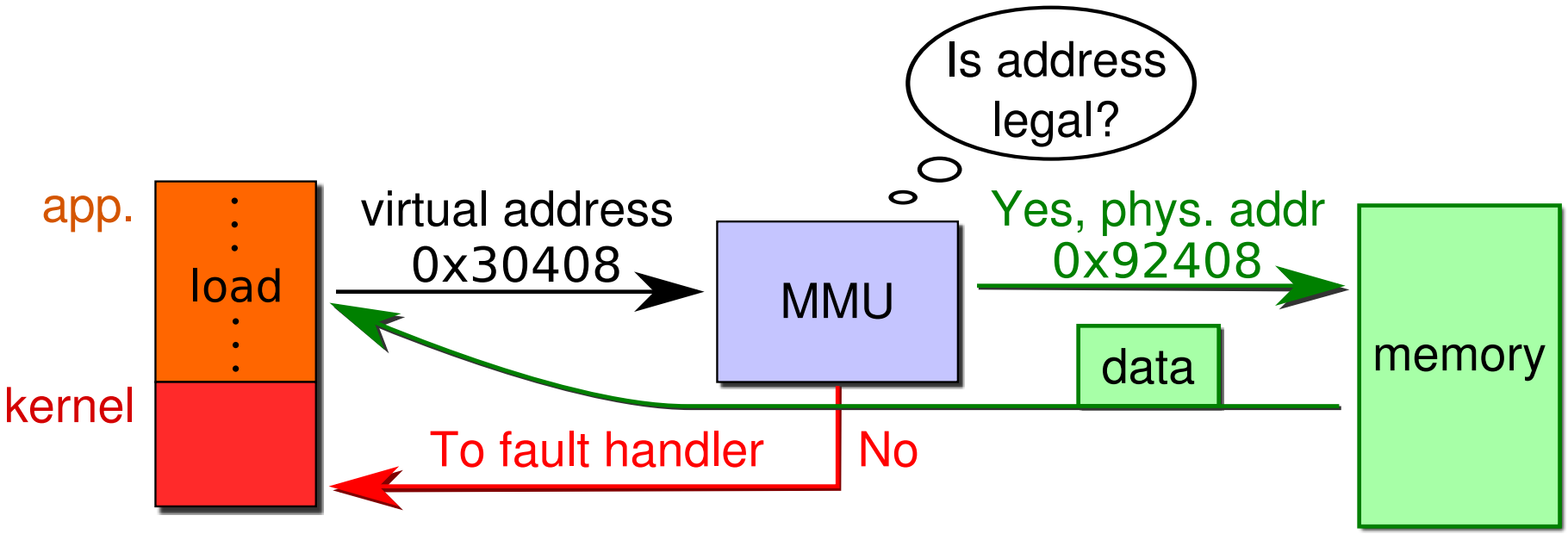
Desired properties when sharing physical memory

- Protection
  - A bug in one process must not corrupt memory in another
  - Don't allow processes to observe other processes' memory (pgp/ssh-agent)
- Transparency
  - A process shouldn't require particular physical memory addresses
  - Processes should be able to use large amounts of contiguous memory
- Resource exhaustion
  - Allow that the sum of sizes of all processes is greater than physical memory

Need protection and (dynamic) relocation

Memory-Management Unit (MMU)

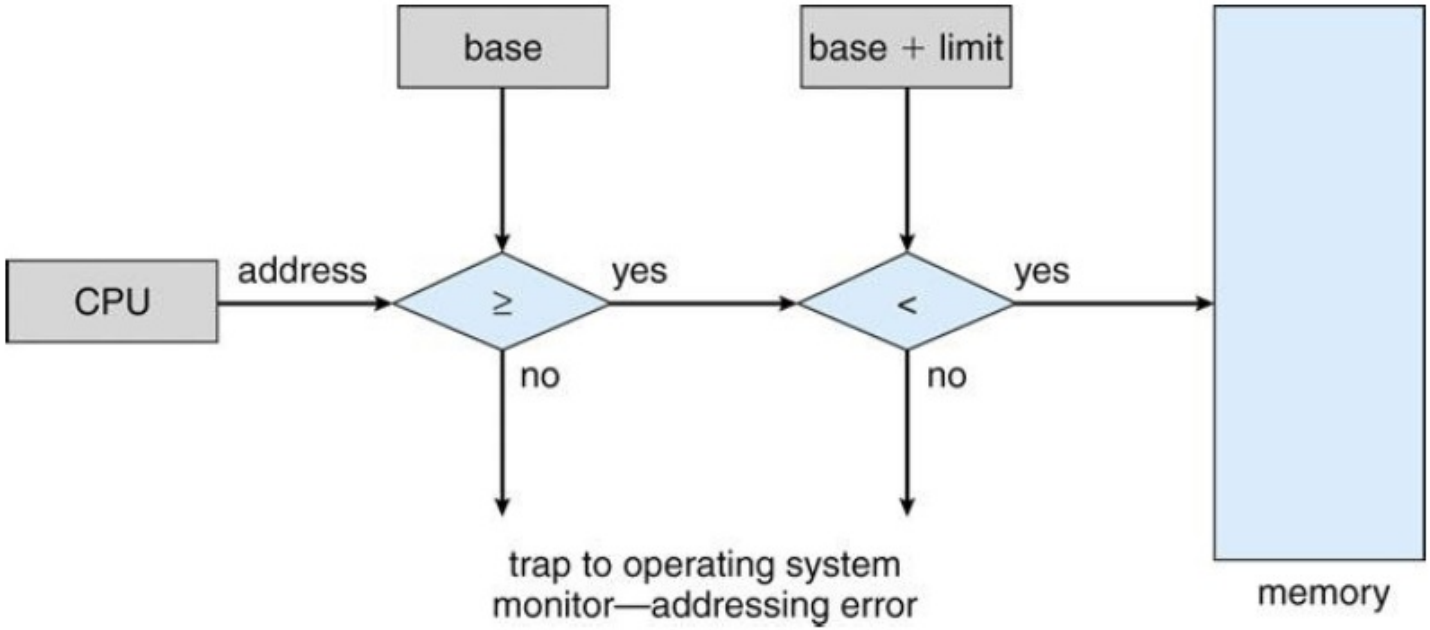
- Need hardware support to achieve safe and secure protection
- Hardware device maps virtual to physical address
- The user program deals with virtual addresses
  - It never sees the real physical addresses



Next: How does an MMU work?

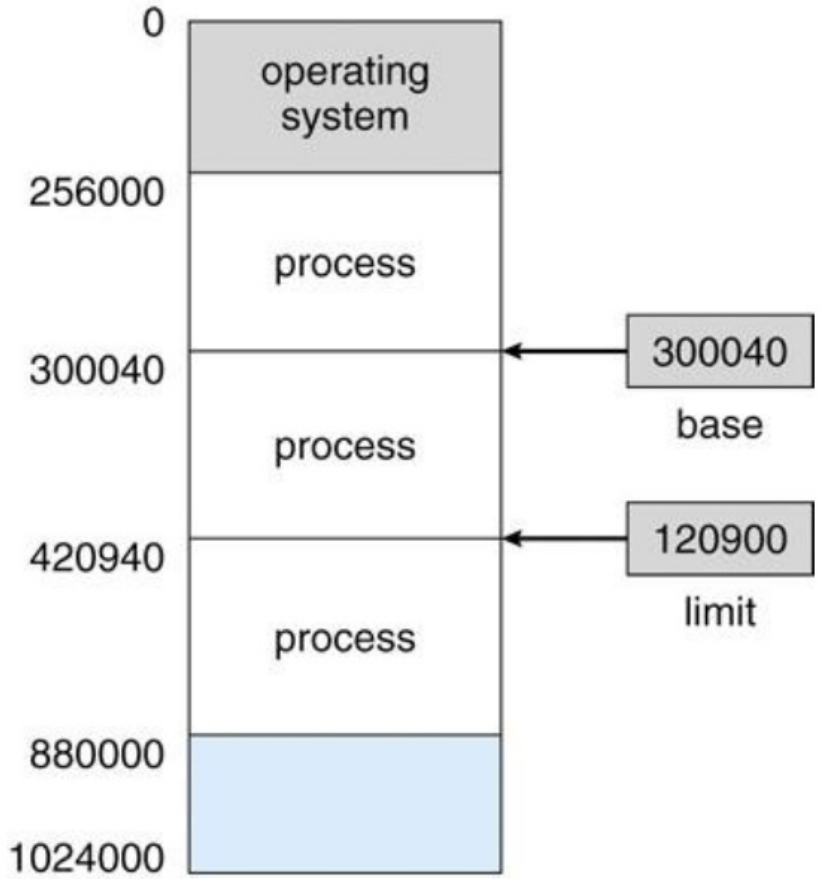
Base and Limit Registers

- Idea: Provide protection and dynamic relocation in the MMU
  - Introduce special base and limit registers (e.g., Cray-1 did this)
- On every load/store the MMU
  - Checks if the virtual address is larger or equal to **`base`**
  - Checks if the virtual address is smaller than **`base`** + **`limit`**
  - Use the virtual address as the physical address in memory



Protecting the Kernel with Base and Limit Registers

- Need to protect OS from processes
- Main memory split into two partitions
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes held in high memory
- OS can access all process partitions e.g., to copy syscall parameters
- MMU denies processes access to OS memory



Base and Limit Registers

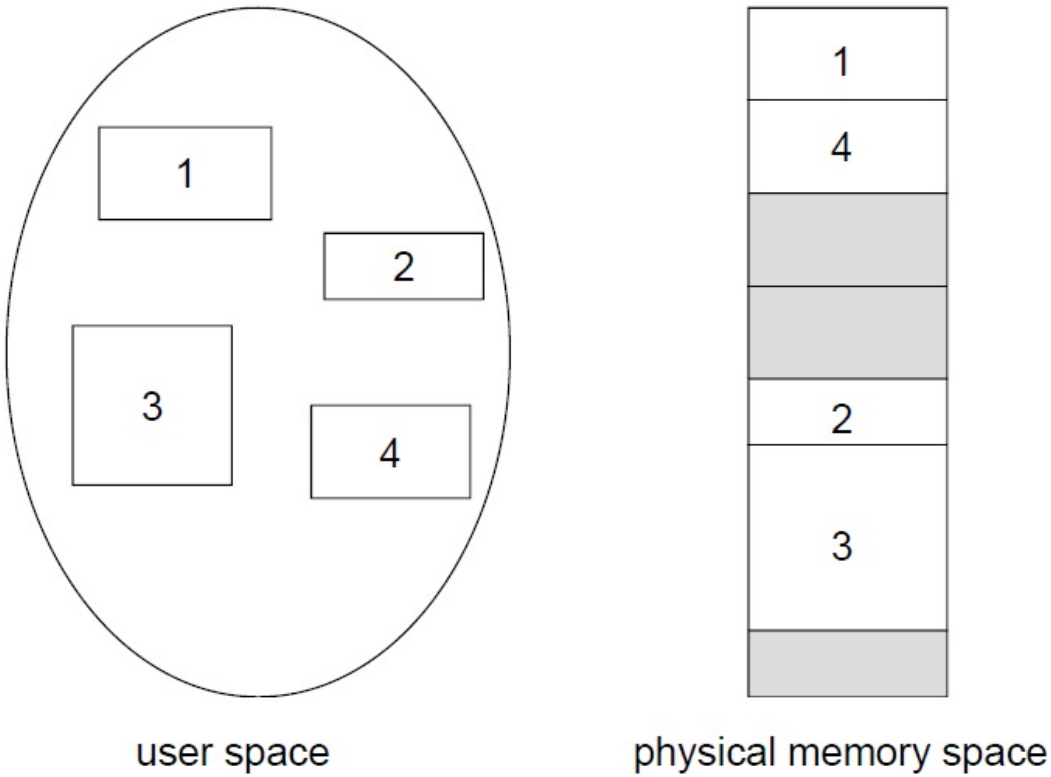
+ Straight forward to implement MMU
  - Only need to load new base and limit registers to switch address space
+ Very quick at run-time
  - Only two comparisons (can do both in parallel)
  - Compute **`base`** + **`limit`** in advance

– How do you grow a process' address space?

– How do you share code or data?

Segmentation

- Possible solution for shortcomings of Base + Limit approach:
  - Use **multiple** Base+Limit register pairs **per process**



→ Can keep some segments private, share others

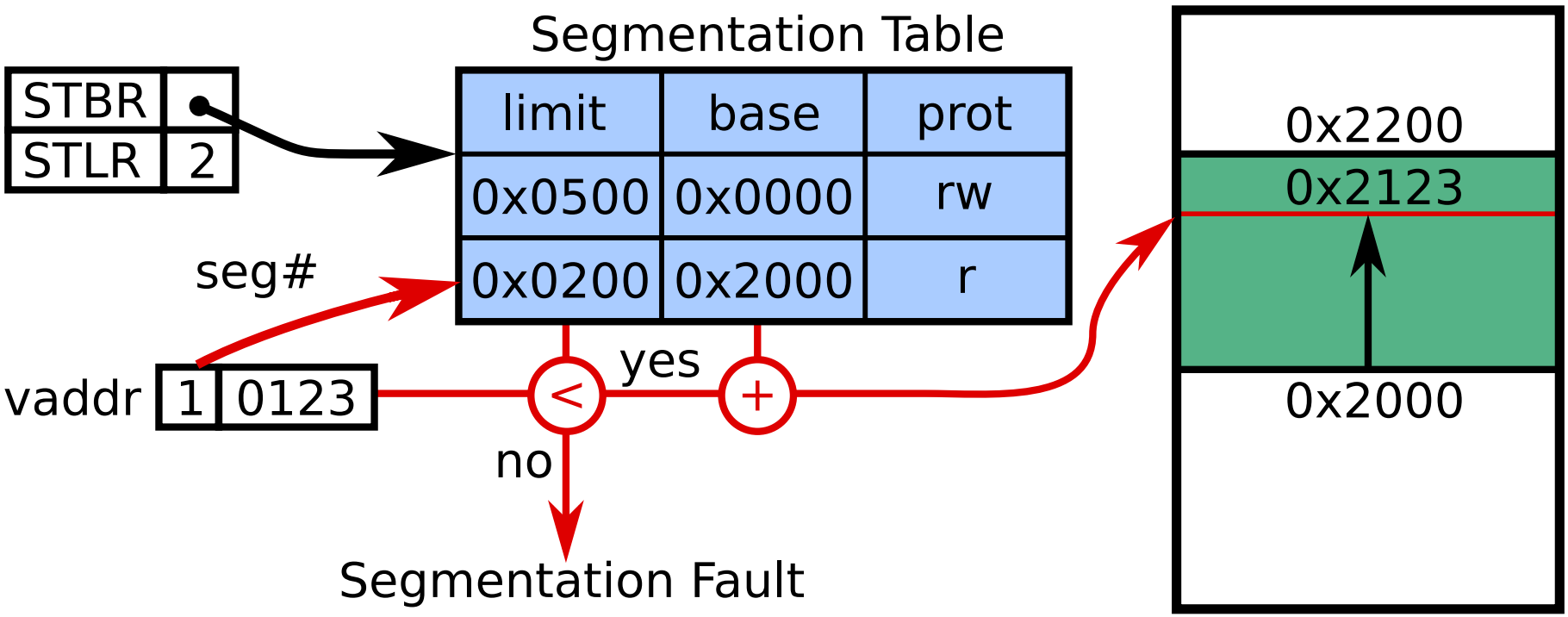
Segmentation Architecture

- Virtual address consists of a tuple: <segment #, offset>
  - Can be encoded in the address (PDP-10 – seg #: high bits, offset: low bits)
  - Can be selected by an instruction or an operand

- Each process (address space) has a segment table that maps virtual address to physical addresses in memory
  - Base Starting physical address where the segment resides in memory
  - Limit Length of the segment
  - Protection Access restriction (read/write) to make safe sharing possible

- The MMU has two registers that identify the current address space
  - Segment-table base register (STBR) points to the segment table location of the current process
  - Segment-table length register (STLR) indicates number of segments used by the process (segment # is legal if it is $<$ STLR)

Segmentation Mechanics



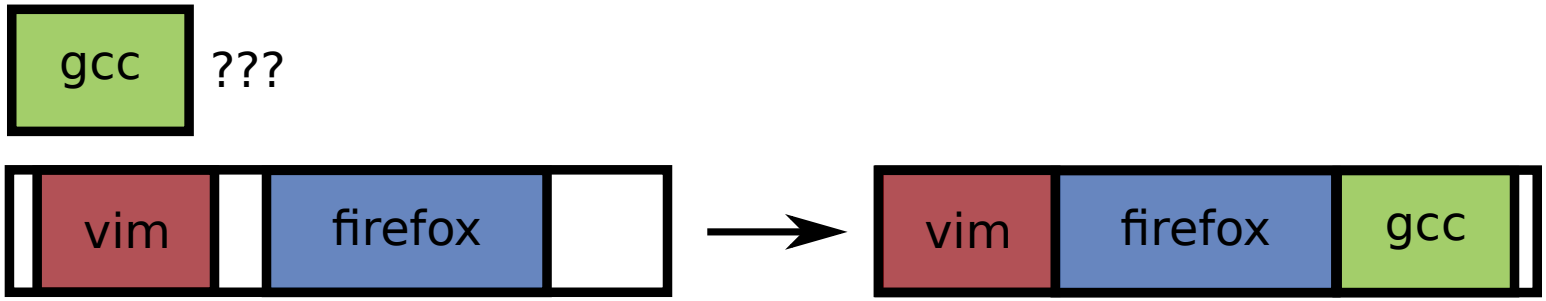
Segmentation Trade-offs

+ Makes data/code sharing between processes possible without compromising confidentiality and safety/security
+ Process doesn't need large contiguous physical memory area
  → easier placement
+ Don't need entire process in memory
  → Can overcommit memory

– Segments need to be kept contiguous in physical memory
– Fragmentation of physical memory

External Fragmentation

- Fragmentation ≡ The inability to use free memory
- External Fragmentation Sum of free memory satisfies requested amount of memory. Contiguous memory required, though.
- Can reduce external fragmentation through compaction
  - Close gaps by moving allocated memory in one direction (e.g., towards 0)
  - Results in a large free block on the other side
  - Compaction is possible only if relocation is dynamic, and can be done at execution time.



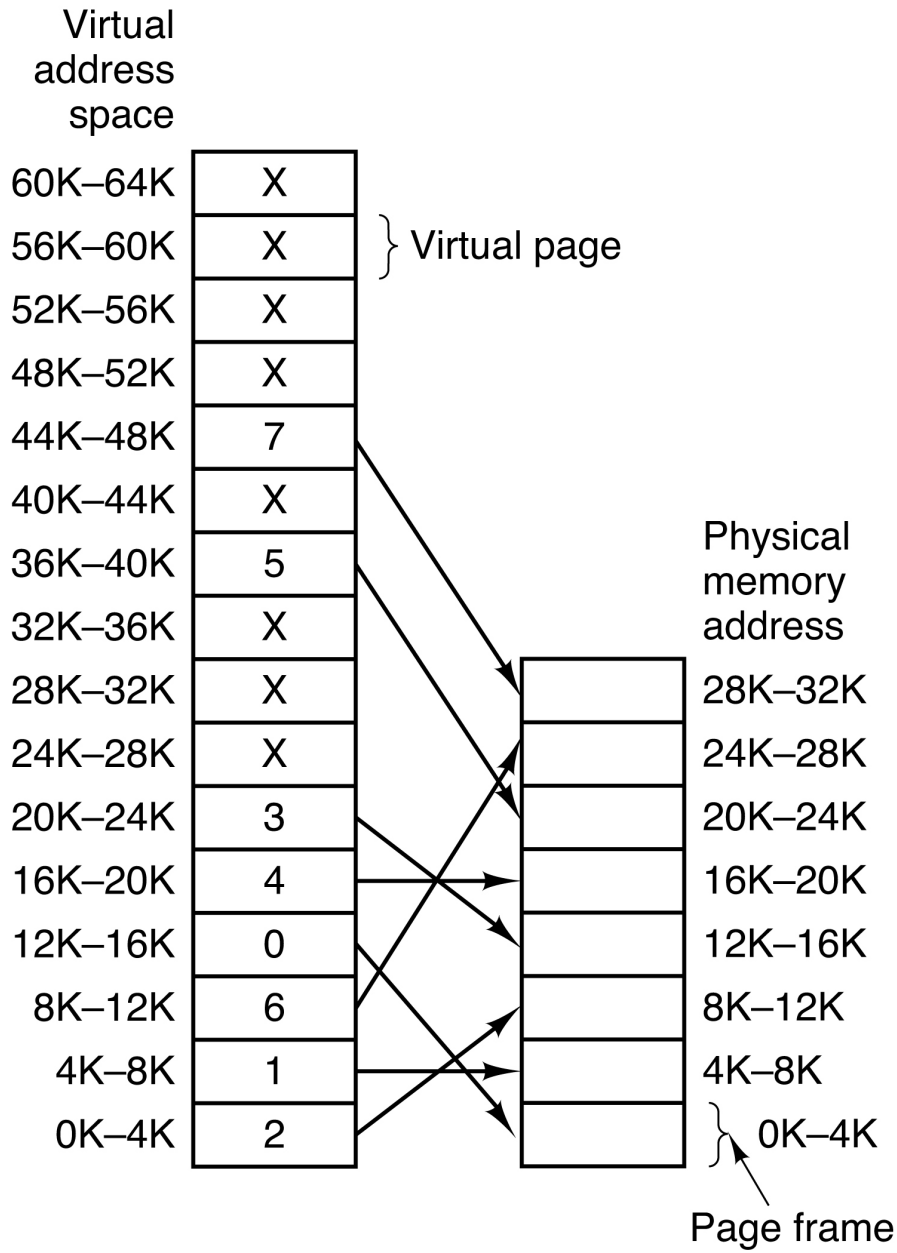
– Expensive: Need to halt process while moving data and updating tables. Need to reload caches afterwards → Should be avoided

Paging

- Divide physical memory into fixed-sized blocks called page frames
  - Size is power of 2 Bytes
  - Typical frame sizes: 4 KiB, 2 MiB, 4 MiB
- Divide virtual memory into blocks called pages
  - Same sizes available as for frames
- OS keeps a page table that stores mappings between virtual page numbers (vpn) and page frame numbers (pfn) for each AS
- OS keeps track of all free frames and modifies page tables as needed
  - To run a program of size n pages, need to find n free frames and load program

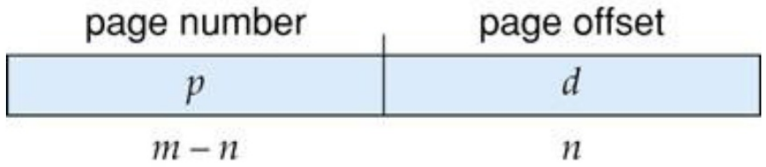
Paging

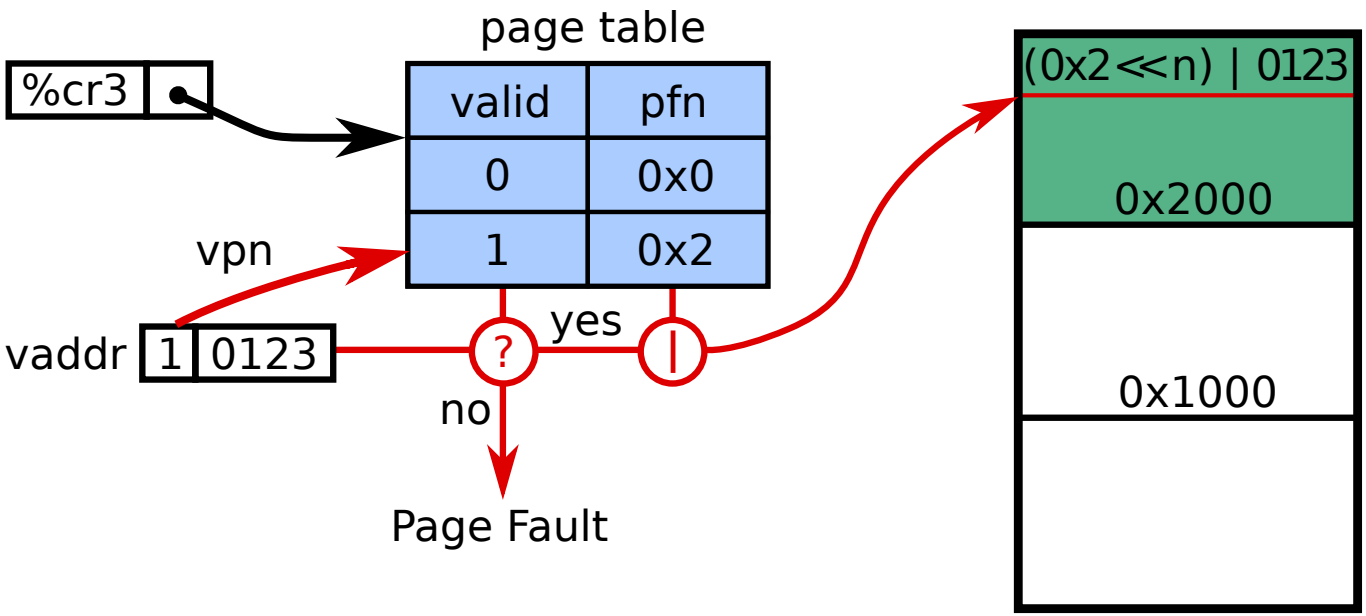
- A Present Bit in the page table indicates if a virtual page is currently mapped to physical memory
- MMU reads the page table and autonomously translates valid mappings
- If a process issues an instruction to access a virtual address that is currently not mapped, the MMU calls the OS to bring in the data (page fault)



Address Translation Scheme

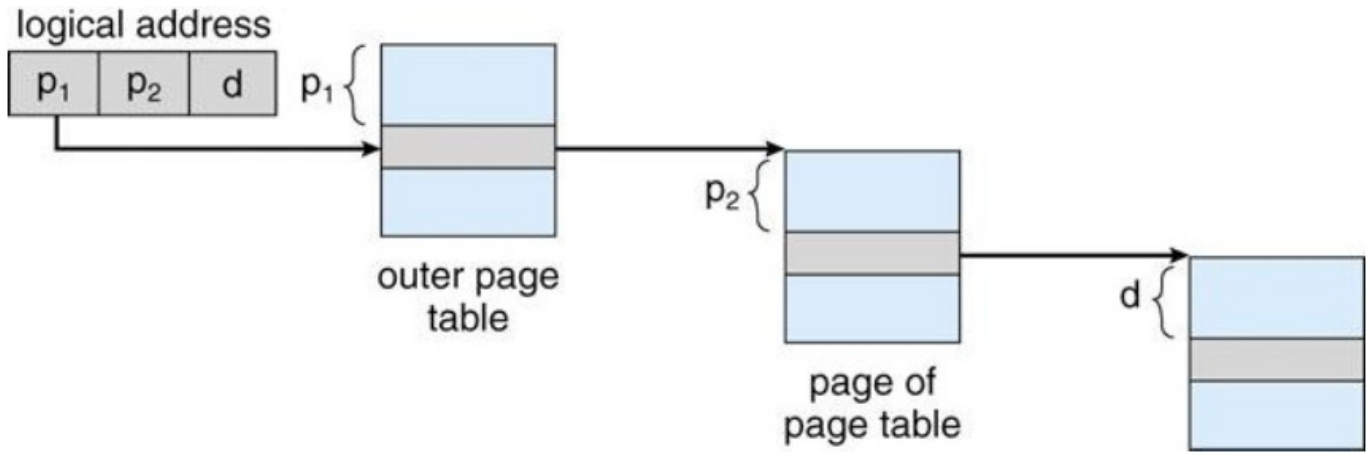
- Virtual address is divided into:
  - Virtual page number: Index into the page table which contains base address of each page in physical memory
  - Page offset: Concatenated with base address results in physical address





Hierarchical Page Table

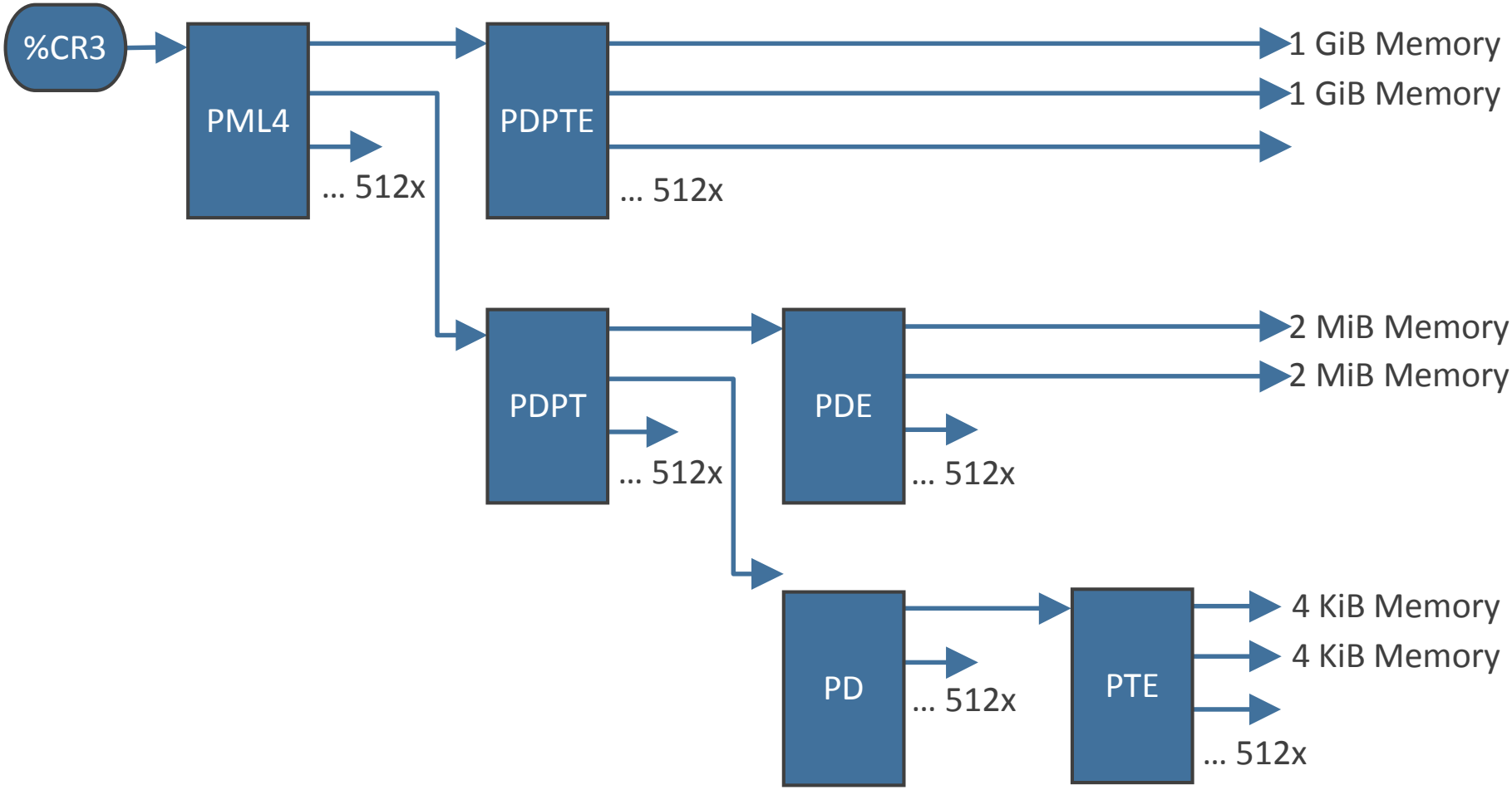
- Problem: For every address space, need to keep complete page table in memory that can map all **virtual** page numbers
- Idea: Don't need complete table, most virtual addresses are not used by process
- Again, another level of indirection saves the day:
  - Subdivide the virtual address further into multiple page table indices p_n forming a hierarchical page table



Intel x86-64 Page Table Hierarchy

- x86-64 long mode: 4-level hierarchical page table
- Page directory base register (Control Register 3, %CR3) stores the starting physical address of the first level page table
- For every address space, the page-table hierarchy goes as follows
  - Page map level 4 (PML4)
  - Page directory pointers table (PDPT)
  - Page directory (PD)
  - Page table entry (PTE)
- At each level, the respective table can either point to a directory in the next hierarchy level, or to an entry containing actual mapping data.
- Depending on the depth of the entry, the mapping has different sizes
  - PDPTE: 1 GiB page
  - PDE: 2 MiB page
  - PTE: 4 KiB page

Intel x86-64 Page Table Hierarchy



Page Table Entry Content

- Valid Bit: Whether the page is currently available in memory or needs to be brought in by the OS, via a page-fault, before accessing it (a.k.a. Present Bit)
- Page Frame Number: If the page is present, at which physical address the page is currently located
- Write Bit: If the page may be written to. When a process writes to a page with a clear write bit, the MMU halts the operation and raises a page-fault
- Caching: If this page should be cached at all and with which policy
- Accessed Bit: Set by the MMU if page was touched since the bit was last cleared by the OS
- Dirty Bit: Set by the MMU if this page was modified since the bit was last cleared by the OS

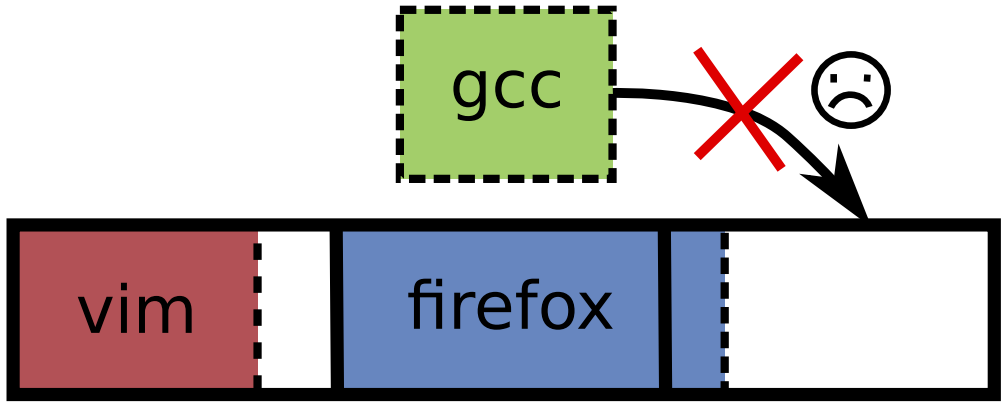
The OS's Involvement in Paging

The OS performs all operations that require semantic knowledge:

- Page allocation/bringing data into memory
  - The OS needs to find a free page frame for new pages and set up the mapping in the page table of the affected address space
- Page replacement
  - When all page frames are in use, the OS needs to evict pages from memory to make room for new pages
  - e.g., code sections can be dropped and re-read from disk on their next use
  - e.g., heap memory has to be saved to a pagefile or swap area before the frame can be evicted
- Context switching
  - The OS sets the MMU's base register (`%CR3` on x86) to point to the page hierarchy of the next process's address space

Internal Fragmentation

- Paging eliminates external fragmentation due to its fixed size blocks
- With paging, however, internal fragmentation becomes a problem
  - As memory can only be allocated in course grained page frame sizes
  - An allocated virtual memory area will generally not end at a page boundary
  - The unused rest of the last allocated page cannot be used by other allocations and is lost



Page Size Trade-Offs

- Fragmentation:
  - Larger pages → More memory wasted due to internal fragmentation for every allocation
  - Small pages → On average only half a page wasted for every allocation
- Table size:
  - Larger pages → Fewer bits needed for pfn (more bits in the offset)
  - Larger pages → Fewer PTEs
  - Smaller pages → More and larger PTEs
  - Note: Page table hierarchies support multiple page sizes with uniform entries, larger pages need fewer page tables (e.g., x86-64)
- I/O:
  - Larger pages → More data needs to be loaded from disk to make page valid
  - Smaller pages → Need to trap to OS more often when loading large program

Summary

- Need to place process in memory to run
- Want to place multiple processes in memory at the same time to run them concurrently/in parallel
- Virtual memory enables protection, transparency, and overcommitting memory at the cost of adding hardware (MMU) to translate memory addresses at every load and store
- Different MMUs have been invented in the past
  - Base + Limit
  - Segmentation
  - Paging
- Paging is supported by all contemporary MMUs
  - Some also support segmentation (e.g., x86, limited in x86-64)
  - Most OS's favor paging over segmentation