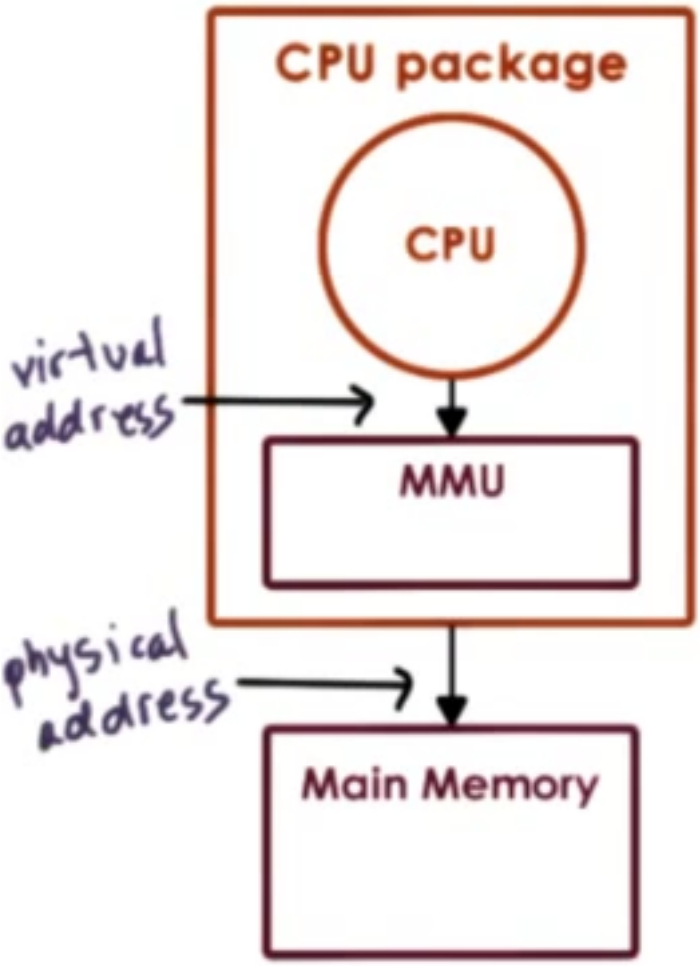
Memory Management

Goals of Memory Management

1. Operating system manages memory for all processes
    - Uses intelligently sized containers (memory pages or segments)
    - Processes operate on a subset of memory
    - Optimized for performance - Reduce time to access state in memory
2. Operating system manages physical resources (DRAM) for one or more processes
3. Processes use virtual address, which maps to a physical address
    - Virtual addresses » Physical addresses
    - Allocate physical memory and arbitrate how it's accessed
        - Allocate - Allocation, replacement
        - Arbitrate - Address translation and validation
    - OS must decide what should be moved from disk to memory and vice versa
4. Page-based memory management
    - Virtual address space partitioned into pages
    - Physical address space partitioned into page frames
    - Allocate: Pages -> Page frames
    - Arbitrate: Page tables
5. Segment-based memory management
    - Allocate: Segments (flexibly sized)
    - Arbitrate: Segment registers

Hardware Support

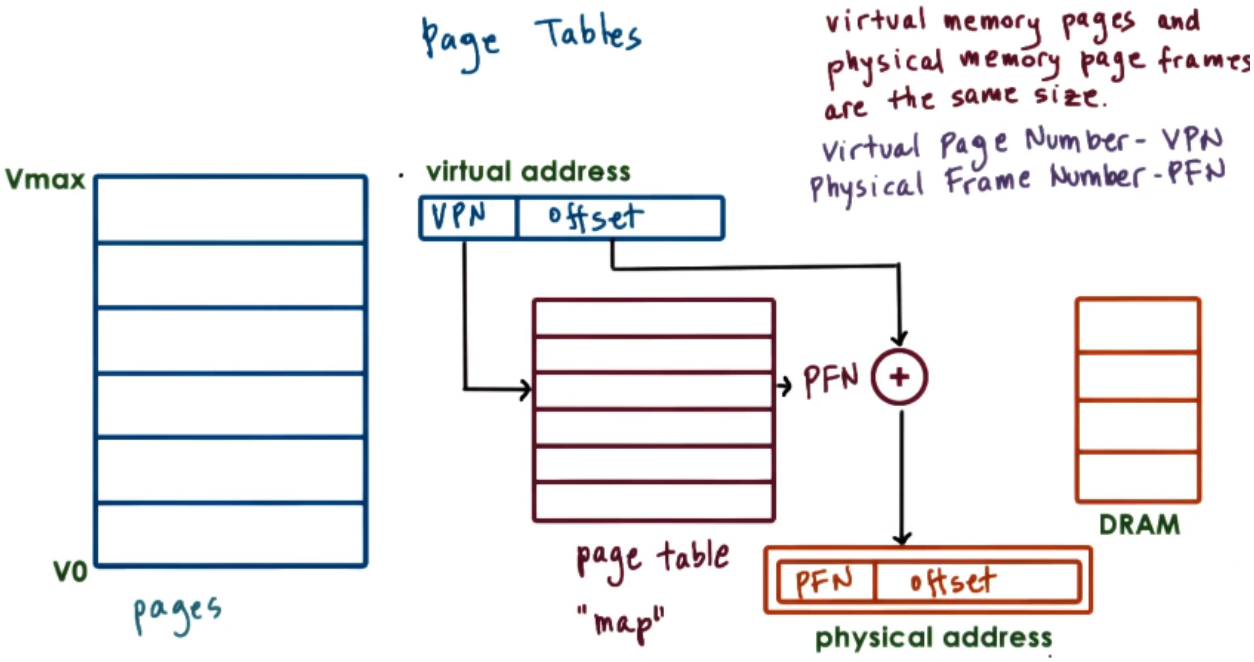
1. Memory management handled by hardware in addition to operating system
2. CPU contains memory management unit (virtual -> physical addresses)
    - Reports faults: Memory address requested hasn't been allocated
    - Permission: Not allowed to access memory (no write permissions)
    - Page fault: Page not present in memory, fetch from disk
3. Registers for address translation
    - Page-based: Pointers to page table
    - Segment-based: Base address and limit size, number of segments
4. Cache: Translation Lookaside Buffer (TLB)
    - Cache of valid VA-PA translations
5. Translation: Actual PA generation done in hardware



Memory Management Unit

Page Tables

1. Pages are more popular method of memory management
2. Page tables convert virtual memory addresses to physical memory addresses
3. "Map" of where in memory to find virtual memory addresses
4. Virtual memory pages and physical memory page frames are the same size
   - Allows us to only translate the first address, others are just offset
5. Only the first portion of the virtual address corresponds to the page number
   - Virtual address: Virtual page number (VPN) + offset
   - Page table maps virtual page number to physical frame number (PFN)
   - Physical address: Physical frame number + offset
   - Valid bit (in memory = 1, not in memory = 0)
6. Allocation on first touch: Only allocate physical memory when accessed, not when allocated
7. Unused pages can be reclaimed by the OS after some duration
8. If a mapping is invalid (valid bit == 0), TRAP occurs and OS intervenes
   - Is access permitted? Where is the page located? Where should it be brought into DRAM?
9. Each process has a page table
   - Every context switch requires switching to valid page table
   - Hardware assists: Register pointing to active page table (CR3 on x86)



Page Table

Page Table Entry

1. Page Frame Number
2. Flags
   - Present (valid/invalid)
   - Dirty (set when written to, indicates cached file should be updated on disk)
   - Accessed (has been accessed for read or write)
   - Protection bits (read, write, execute)
3. Page Table Entry on x86
   - P: Present
   - D: Dirty
   - A: Accessed
   - R/W: Permission bit (0 -> Read only, 1 -> Read/write)
   - U/S: Permission bit (0 -> usermode, 1 -> Supervisor mode)
   - Others: Caching related info (write through, caching disabled)
   - Unused: For future use
4. If hardware determines physical memory access can't be performed using the permission bits, it causes a page fault (error code on kernel stack)
   - CPU generates a TRAP into the OS kernel
   - This generates a page fault handler
     - Determines action based on error code and faulting address
     - Can bring page from disk into memory
     - Protection error (SIGSEGV)
   - On x86
     - Error code from PTE flags
     - Faulting address in CR2

Page Table Entry

Page Table Size

1. 32-bit architecture
   - Page Table Entry (PTE): 4 bytes, including PFN + flags
   - Virtual Page Number (VPN): 2^32 / Page size
   - Page Size: 4 kB
   - (2^32 / 2^12) = 4 MB per process
2. Process doesn't use entire address space
3. Even on 32-bit architecture, won't always use all of 4 MB
4. Page table assumes an entry per VPN, regardless of whether corresponding virtual memory is needed or not

Hierarchical Page Tables

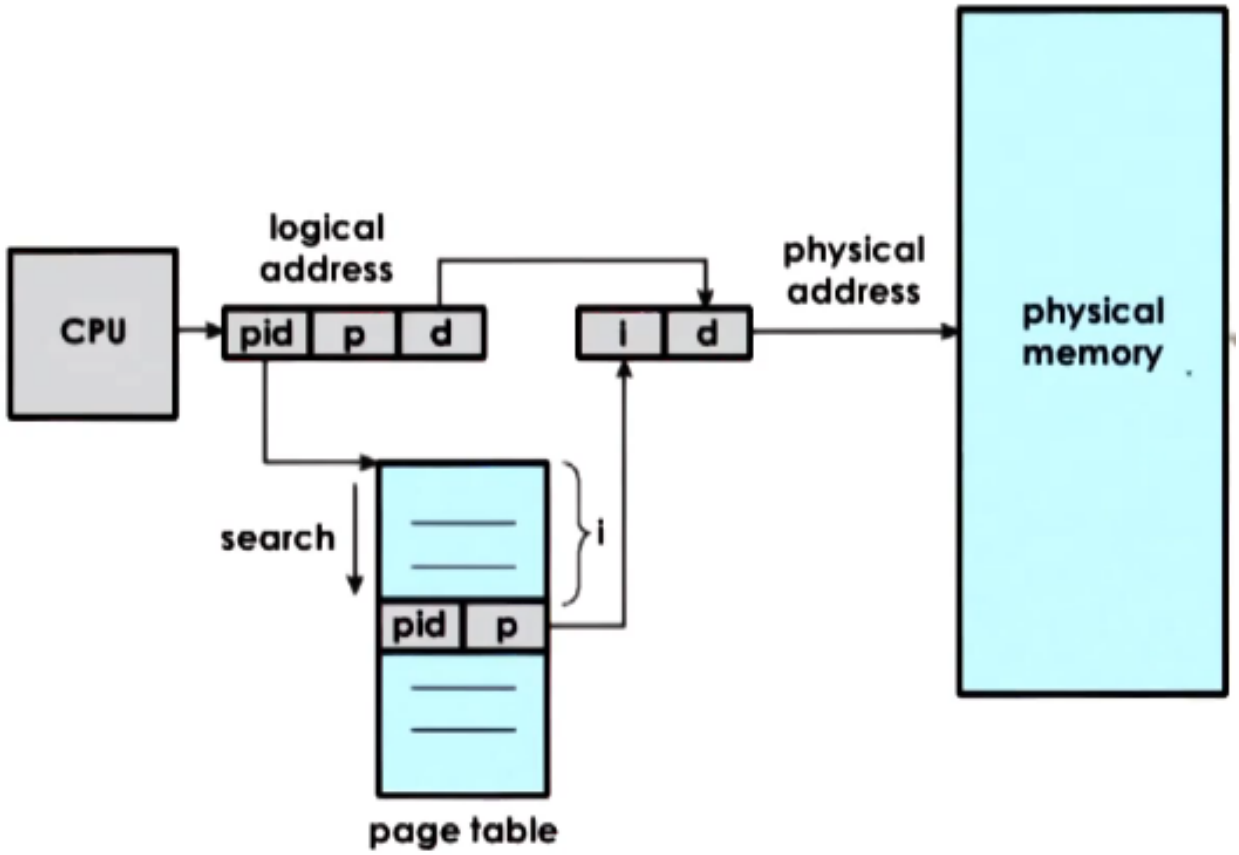
1. Outer Page Table: Page table directory (pointers to page tables)
2. Internal Page Table: Only for valid virtual memory regions
3. On malloc, a new internal page table may be allocated
4. 12 bits for outer page table, 10 bits for internal page table, 10 bits offset
   - 2^10 * page size = 2^10 * 2^10 = 1MB
   - Don't require an inner table for each 1MB virtual memory gap
   - Solves the problem of having to store so much in memory
5. Additional layers
   - 3rd level: Page table directory pointer
   - 4th level: Page table directory pointer map
   - Important on 64 bit architectures, larger and more sparse
     - Larger gaps -> Could save more internal page table components
6. Tradeoffs
   - Pro: More levels means smaller internal page tables/directories and improved granularity of coverage
     - Reduced page table size
   - Con: More memory accesses required for translation
     - Increased translation latency

Translation Lookaside Buffer

1. Single level page table
   - 1x access to page table entry
   - 1x access to memory
2. Four level page table (slower)
   - 4x accesses to page table entry
   - 1x access to memory
3. Avoid repeated access to memory by caching address
4. Translation Lookaside Buffer (TLB)
   - MMU-level address translation cache

- On TLB miss -> Page table access from memory
- Has protection validity bits
- Small number of cached addresses can still result in high TLB hit rate
  - Due to temporal and spatial locality

5. x86 core i7
   - Per core: 64-entry data TLB, 128-entry instruction TLB
   - 512-entry shared second-level TLB



Inverted Page Table

Inverted Page Tables

1. Use pid to index into page table; index + offset gets physical memory address
2. Must perform linear search of pids (table isn't ordered)
3. TLB catches most of the memory references, so search is infrequent
4. Hashing page tables can help solve this problem
   - Hash points to a linked list
5. Speeds up address translation

Segmentation

1. Segments == arbitrary granularity
   - Correspond to code, heap, data, stack....
   - Address == segment selector + offset
2. Segment == continuous physical memory
   - Segment size == segment base + limit registers

3. Segmentation is used with paging; address passed to paging unit to compute physical address
4. Intel x86 (32 bit) -> Segmentation and paging supported
    - Linux: Up to 8K per process/8K per global process
5. Intel x86 (64 bit) -> Only paging

Page Size

1. 10-bit offset -> 1kB page size
2. 12-bit offset -> 4kB page size
3. Systems generally support different page sizes
    - Linux/x86 - 4kB, 2MB, 1GB
    - Solaris/SPARC - 8kB, 4MB, 2GB
4. Larger pages:
    - Pros: Fewer page table entries, smaller page tables, more TLB hits
    - Cons: Page size -> internal fragmentation, wastes memory

	large	huge
page size	2 MB	1 GB
offset bits	21 bits	30 bits
Reduction	x512	x1024

Memory Allocation

1. Memory allocator: Determines VA to PA mapping
    - Address translation, page tables, etc. used to simply determine PA from VA and check validity/permissions
2. Kernel-level allocators
    - Kernel state, static process state
3. User-level allocators
    - Dynamic process state (heap); malloc/free

Linux Kernel Allocators

1. A memory allocator must limit fragmentation and permit the coalescing and aggregation of adjacent free areas
2. Buddy Allocator: Start with a 2^x area
    - On request: Subdivide into 2^x chunks and find smallest 2^x chunk that can satisfy the request
    - On free: Check buddy to see if you can aggregate into a larger chunk
        - Aggregate more up the tree
        - Power of 2 means addresses only differ by 1 bit
    - Pros: Aggregation works well and fast
    - Cons: Some fragmentation still exists
3. Slab Allocator: Caches for common object types/sizes, on top of contiguous memory
    - Slab: Continuous allocated physical memory
    - Cache based on size of object (i.e., task struct)
        - If a cache has space, add an object to it
        - Else, allocate more pages to make space (this is within a page)
    - Pros: Avoids internal and external fragmentation

Demand Paging

1. Virtual memory » Physical memory
    - Virtual memory page not always in physical memory

- Physical page frame saved and restored to/from secondary storage
- This is referred to as demand paging (swapping pages in/out of memory and a swap partition (on disk))

2. If the page is not present (present bit == 0 in page table), trigger a page fault
   - Operating system will go to disk, retrieve the page, and swap it into physical memory in a free frame
   - Then, reset the page table and restart the instruction
3. Can "pin" a page, meaning constantly present in memory (disable swapping)

Page Replacement (Freeing Physical Memory)

1. When should pages be swapped out? Page(out) daemon should run when...
   - Memory usage is above threshold (high watermark)
   - CPU usage is below threshold (low watermark)
2. Which pages should be swapped out?
   - Pages that won't be used; history-based prediction
   - Least-recently Used (LRU policy) uses access bit to track if a page is referenced
   - Pages that don't need to be written out to disk (slow) - Can use the dirty bit to track modified pages
   - Avoid non-swappable pages
3. Parameters to tune thresholds for swapping in Linux
   - Target page count
   - Memory usage
   - CPU usage
4. Categorize pages into different types (claimable, swappable...)
5. "Second chance" variation of LRU - Only swap if not used twice

Copy on Write

1. On process creation, the entire parent process address space is copied
   - However, don't need to track multiple copies of static pages
   - On create, map new VA to original page and write protection this page
2. On write, page fault and make a copy
   - Pays the cost of copying only if absolutely necessary

Failure Management Checkpointing

1. Checkpointing: Failure and recovery management technique
   - Periodically save process state
   - Failure may be unavoidable, but can restart from checkpoint so recovery is much faster
2. approach: Pause and copy
3. Better approach: Write-protect entire address space and copy everything
   - However, process will continue to execute, writing to pages
   - Copy diffs of "dirtied" pages for incremental checkpoints
   - Rebuild from multiple diffs, or in background
4. Debugging (rewind-replay)
   - Rewind: Restart from last checkpoint
   - Replay: Run from there
   - Gradually go back to older checkpoints until the error is found
5. Migration
   - Checkpoint the process to another machine and continue there
   - Disaster recovery
   - Consolidation (migrate load to as few machines as possible)
   - Repeated checkpoints in a fast loop until pause-and-copy becomes acceptable (or unavoidable)

6. The more frequently you checkpoint...
    - The more state you will checkpoint (observe multiple writes)
    - The higher the overheads of the checkpointing process
    - The faster you will be able to recover from a fault