Paging

Professor Hugh C. Lauer
CS-3013 — Operating Systems

(Slides include copyright materials from *Operating Systems: Three Easy Step*, by Remzi and Andrea Arpaci-Dusseau, from *Modern Operating Systems*, by Andrew S. Tanenbaum, 3rd edition, and from other sources)

Review – memory management

- **Allocation of physical memory to processes**

- ***Virtual Address***
  - Address space in which the process "thinks"
  - Each virtual address is translated "on the fly" to a physical address

- ***Fragmentation***
  - *Internal fragmentation* – unused space within an allocation
  - *External fragmentation* – unused space between allocations

Review – memory management (cont'd)

- ***Segmentation***
  - Recognition of different parts of virtual address space
  - Different allocation strategies

- **The rule of "no free lunch"**
  - Fixed size allocations ≡ no *external fragmentation*, more *internal fragmentation*
  - Variable allocations ≡ minimize *internal fragmentation*, more *external fragmentation*

Reading assignment

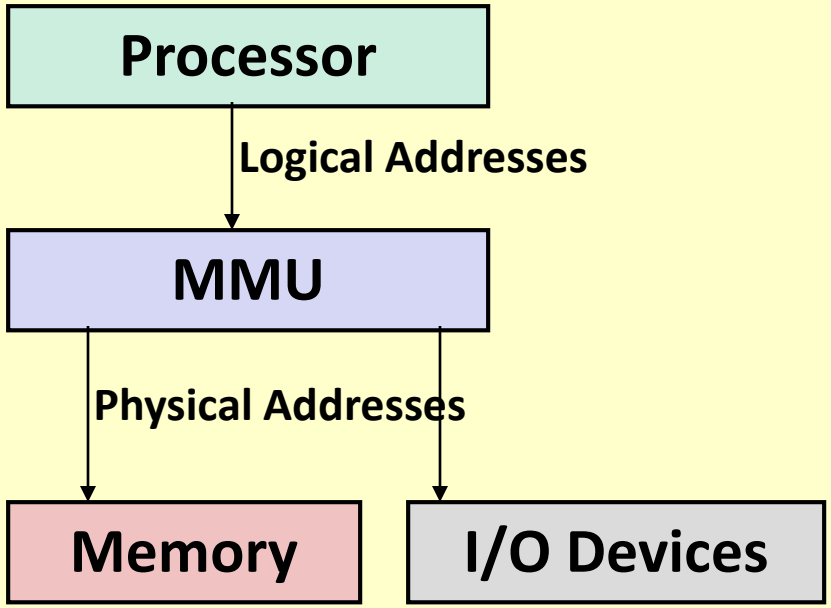
- **OSTEP**
  - §§ 12-17 (previous topic – *Memory Management*)
  - § 18 (this topic on *Paging*)

Paging

- A different approach
- Addresses all of the issues of previous topic
- Introduces new issues of its own

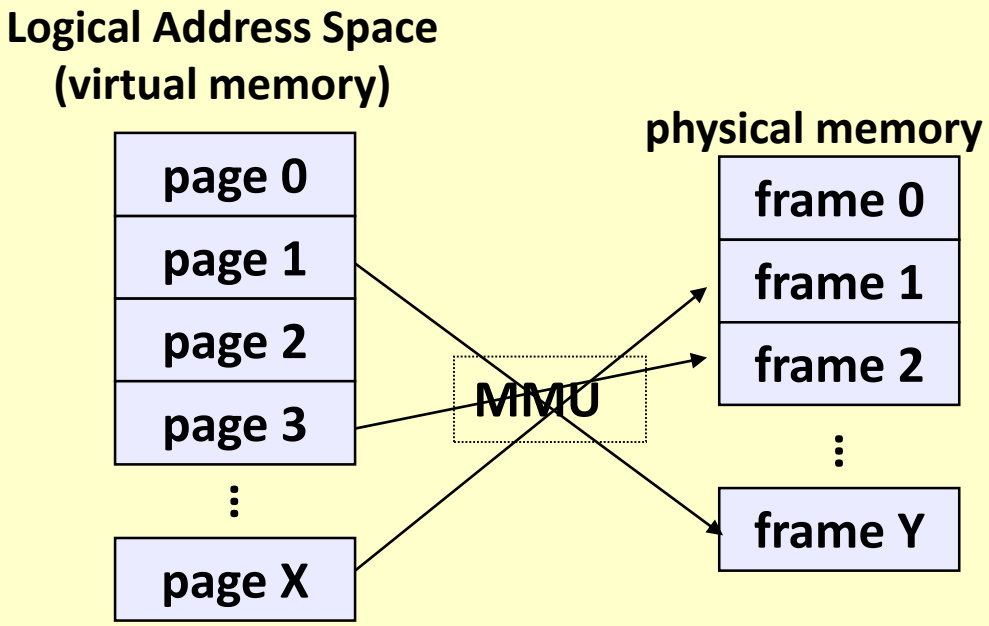
Memory management

- ***Virtual* (or logical) address *vs.* *Physical* address**
- ***Memory Management Unit* (MMU)**
  - Set of registers and mechanisms to translate *virtual* addresses to *physical* addresses
- **Processes (and processors) see virtual addresses**
  - Virtual address space is same for all processes, usually 0 based
  - Virtual address spaces are protected from other processes
- **MMU and devices see physical addresses**



Paging

- Use small *fixed size units* in both physical and virtual memory
- Provide sufficient *MMU hardware* to allow page units to be scattered across memory
- Make it possible to leave *infrequently used parts* of virtual address space out of physical memory
- Solve internal & external *fragmentation* problems



Paging

- **Processes see a large *virtual address space***
  - Either contiguous or segmented
- **Memory Manager divides the virtual address space into equal sized pieces called *pages***
  - Some systems support more than one page size

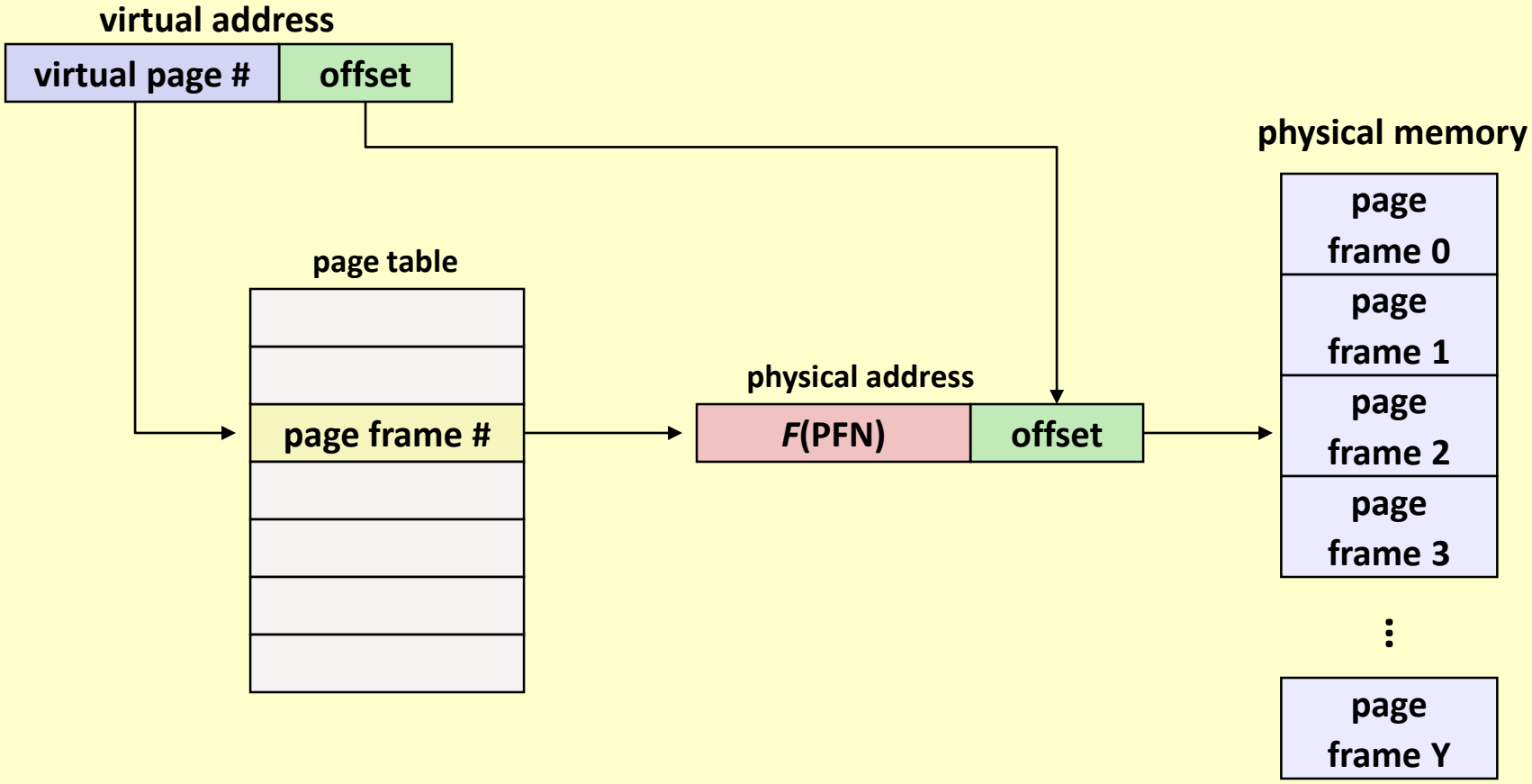
Paging (continued)

- **Memory Manager divides the physical address space into equal sized pieces called *frames***
  - Size usually a power of 2 between 512 and 8192 bytes
    - Some modern systems support 64 megabyte pages!
  - Frame table
    - One entry per frame of physical memory; each entry is either
      - Free
      - Allocated to one or more processes
- **`sizeof(page) == sizeof(frame)`**

Address translation for paging

- **Translating virtual addresses**
  - a virtual address has two parts: *virtual page number* & *offset*
  - virtual page number (VPN) is index into a *page table*
  - page table entry contains *page frame number* (PFN)
  - physical address is: **`startof(PFN) + offset`**
- **Page tables**
  - Supported by MMU hardware
  - Managed by the Memory Manager
  - Map virtual page numbers to page frame numbers
    - one *page table entry* (PTE) per page in virtual address space
    - i.e., one PTE per VPN

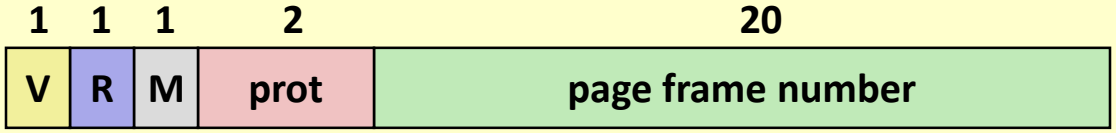
Paging translation



Page translation example

- **Assume a 32-bit contiguous address space**
  - Assume page size 4KBytes ($\log_2(4096) = 12$ bits)
  - For a process to address the full logical address space
    - Need 2^{20} PTEs – VPN is 20 bits
    - Offset is 12 bits
- **Translation of virtual address `0x12345678`**
  - Offset is `0x678`
  - Assume PTE(`0x12345`) contains `0x01010`
  - Physical address is `0x01010678`

Generic PTE structure



- ***Valid* bit gives state of this *Page Table Entry* (*PTE*)**
  - says whether or not its virtual address is valid – in memory and VA range
  - If not set, page might not be in memory or *may not even exist!*
- ***Reference* bit says whether the page has been accessed**
  - it is set by hardware *whenever* a page has been read or written to
- ***Modify* bit says whether or not the page is *dirty***
  - it is set by hardware during *every* write to the page
- ***Protection* bits control which operations are allowed**
  - read, write, execute, etc.
- ***Page* frame number (PFN) determines the physical page**
  - physical page start address
- ***Other* bits dependent upon machine architecture**

Paging – advantages

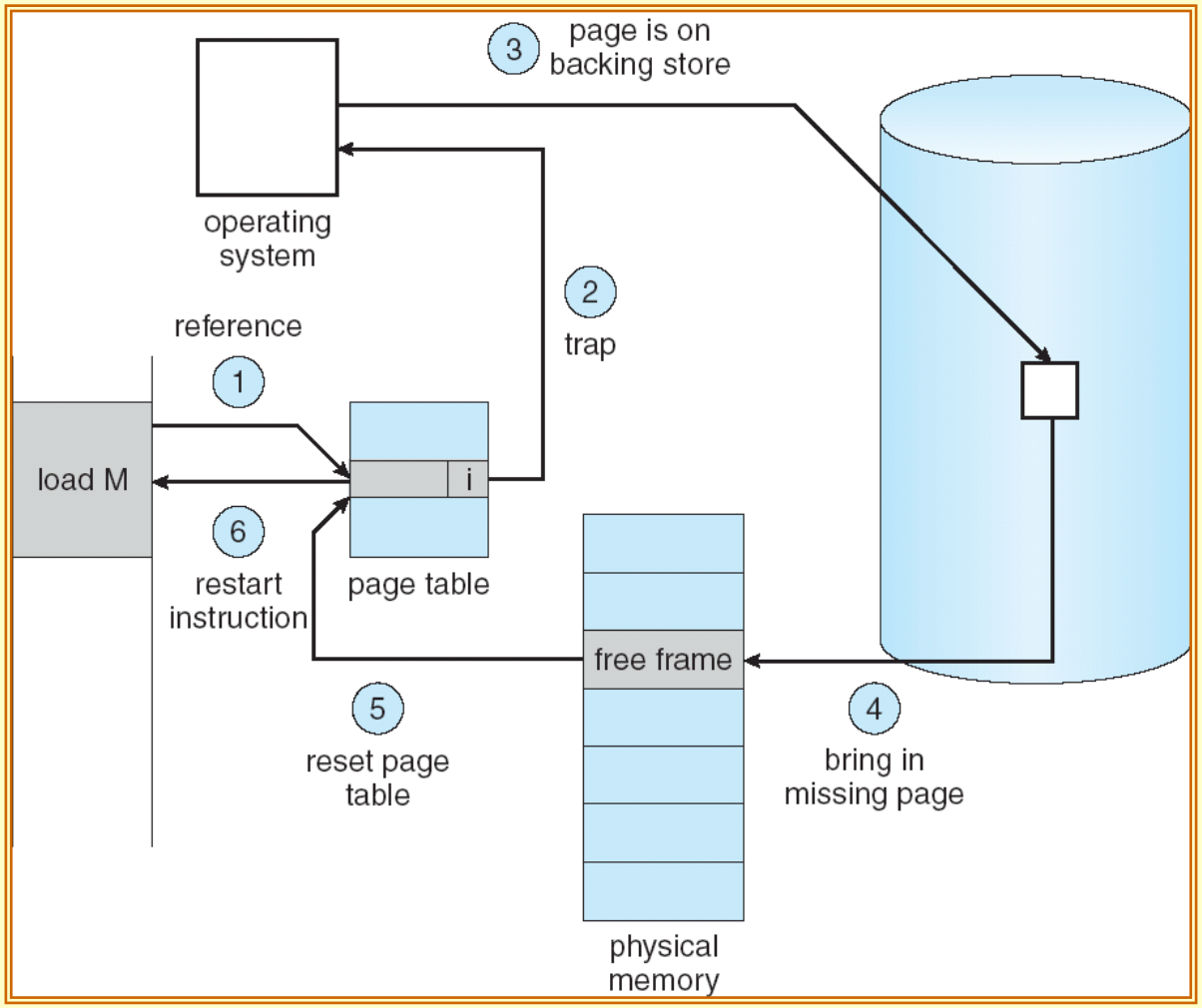
- **Easy to allocate physical memory**
  - pick (almost) any free frame
- **No external fragmentation**
  - All frames are equal
- **Minimal internal fragmentation**
  - Bounded by page/frame size
- **Easy to swap out pages (called *pageout*)**
  - Size is usually a multiple of disk blocks
  - PTEs may contain info that help reduce disk traffic
- **Processes can run with not all pages swapped in**

Definition — *page fault*

- ***Trap* when process attempts to reference a virtual address in a page with *Valid* bit in PTE set to *false***
  - E.g., page not in physical memory
  - Or some other reason!
- **If page exists on disk:–**
  - Suspend process
  - If necessary, throw out some other page (& update its PTE)
  - Swap in desired page, resume execution
- **If page does not exist on disk:–**
  - Return program error

    or
  - Conjure up a new page and resume execution
    - E.g., for growing the stack or heap!

Steps in handling a page fault



Definition — *backing storage*

Also called backing store

- A place on external storage medium where copies of virtual pages are kept
  - Usually a hard disk (locally or on a server)
- Place from which contents of pages are read after page faults
- Place where contents of pages are written if page frames need to be re-allocated

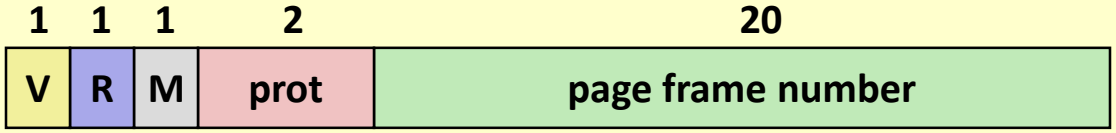
Backing storage

- **Executable code and constants:–**
  - The loadable image file produced by compiler
- **Working memory (stack, heap, static data)**
  - Special *swapping* file (or partition) on disk
- **Persistent application data**
  - Application data files on local or server disks

Requirement for paging to work!

- **Machine instructions must be capable of *restarting***
- **If execution was interrupted during a partially completed instruction, need to be able to**
  - continue or
  - redo without harm
- **This is a property of all modern CPUs …**
  - … but not of some older CPUs!

Protection bits



- **Protection bits are important part of paging**
- **A process may have valid pages that are**
  - not writable
  - execute only
  - etc.
- **E.g., setting PTE protection bits to prohibit certain actions is a legitimate way of detecting the first action to that page.**

Protection bits (continued)

- Definition of *Page Fault* is extended to include both:–

- *Trap* when process attempts to reference a virtual address in a page with *Valid* bit in PTE set to *false*

OR

- Access inconsistent with protection setting

Example

- **A page may be logically writable, as required by the application, but …**
- **… setting PTE bits to disallow writing is a way of detecting the first attempt to write**
  - Take some action
  - Reset PTE bit
  - Allow process to continue as if nothing happened

Questions?

Observations and definitions

- **Recurring themes in paging**
  - *Temporal Locality* – locations referenced recently tend to be referenced again soon
  - *Spatial Locality* – locations near recent references tend to be referenced soon
- **Definitions**
  - *Working set:* The set of pages that a process needs to run without frequent page faults
  - *Thrashing:* Excessive page faulting due to insufficient frames to support working set

Paging issues

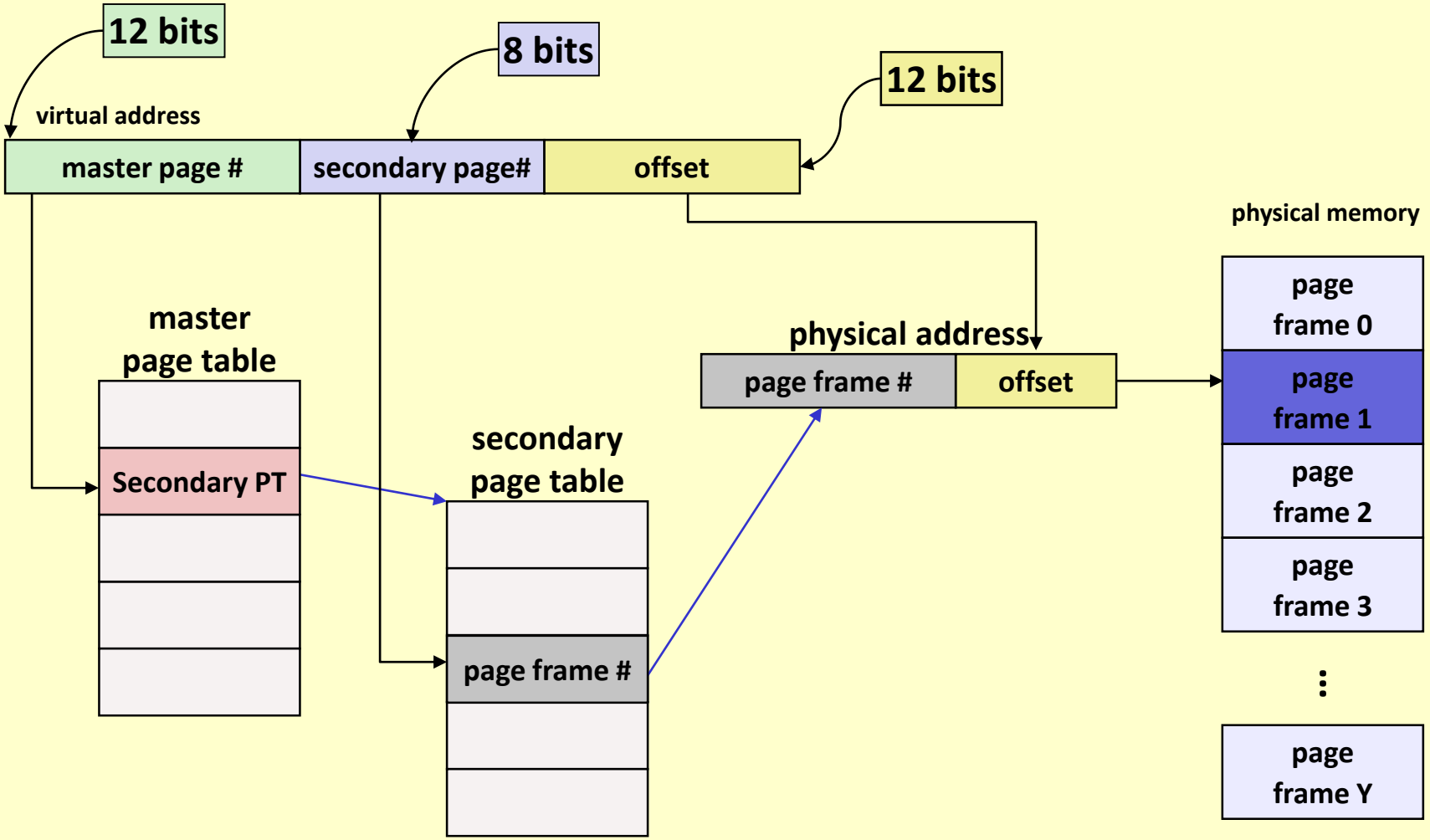
- **#1 — Page Tables can consume large amounts of space**
  - If PTE is 4 bytes, and use 4KB pages, and have 32 bit VA space ⇒ 4MB for each process's page table
    - Stored in main memory!
  - What happens for 64-bit logical address spaces?
- **#2 — Performance Impact**
  - Converting virtual to physical address requires multiple operations to access memory
    - Read Page Table Entry from memory!
    - Get page frame number
    - Construct physical address
    - Assorted protection and valid checks
  - Without fast hardware support, requires multiple memory accesses and a lot of work per logical address

I.e., must read memory to read memory!

Issue #1: page table size

- **Process Virtual Address spaces**
  - Not usually full – don't need every PTE
  - Processes do exhibit locality – only need a subset of the PTEs
- **Solution — two-level page tables**
- **I.e., Virtual Addresses have 3 parts**
  - Master page number – points to secondary page table
  - Secondary page number – points to PTE containing page frame #
  - Offset
- **Physical Address = offset + startof (PFN)**

Two-level page tables

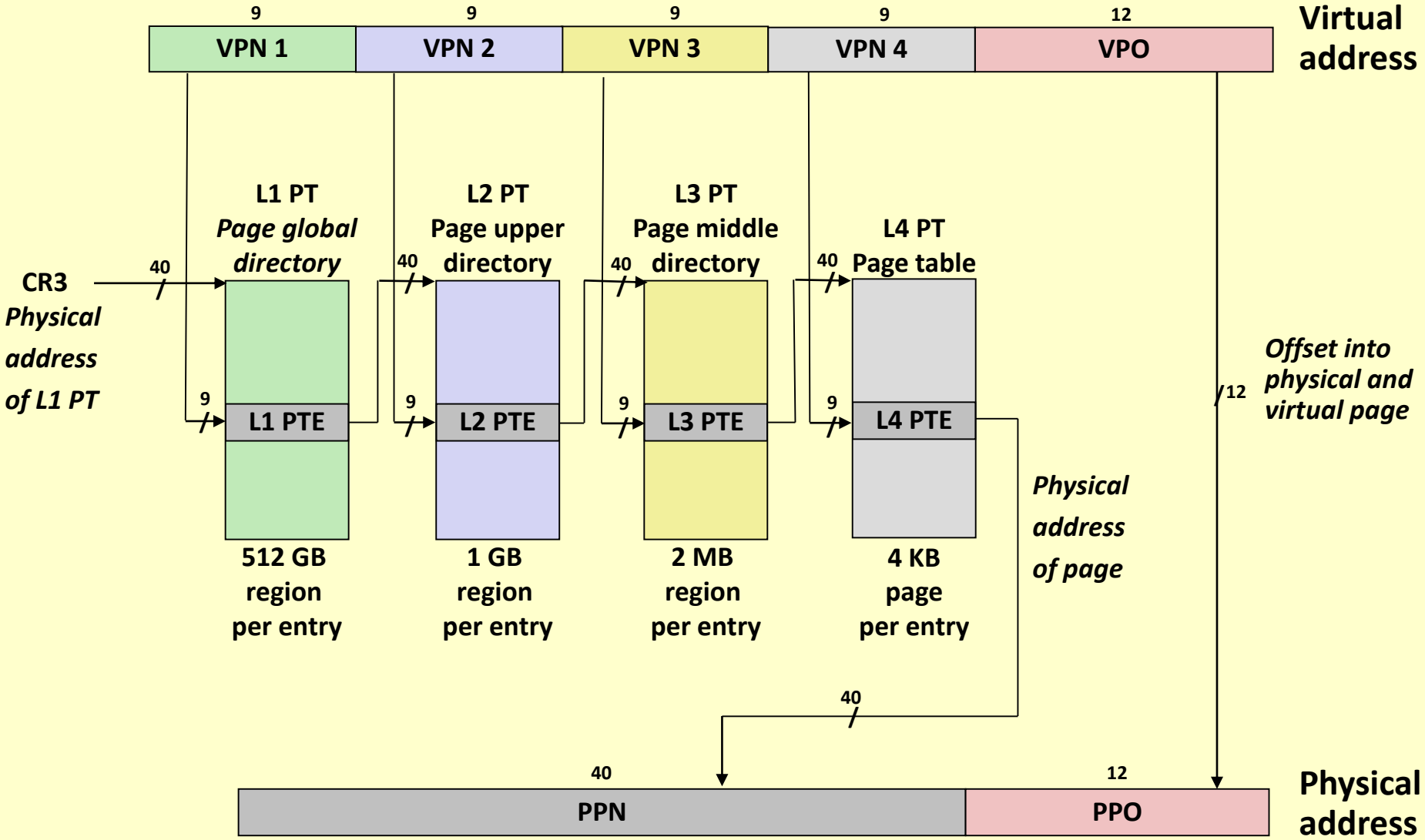


Note

- **Note: *Master page number* can function as *Segment number***
  - previous topic

- ***n*-level page tables are possible**
  - Rare in 32-bit systems
  - Common in 64-bit systems

Core i7 page translation



Multilevel page tables

- **Sparse Virtual Address space – very few secondary PTs ever needed**
- **Process Locality – only a few secondary PTs needed at one time**
- **Can page out secondary PTs that are not needed now**
  - Don't page Master Page Table
  - Save physical memory

However

- **Performance is worse**
  - Now have 3 (or more) memory accesses per virtual memory reference or instruction fetch
- **How do we get back to about 1 memory access per VA reference?**
  - Problem #2 of "Paging Issues" slide

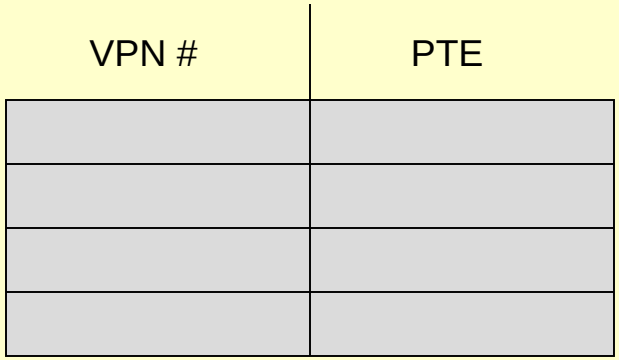
Paging issues

- **#1 — Page Tables can consume large amounts of space**
  - If PTE is 4 bytes, and use 4KB pages, and have 32 bit VA space ⇒ 4MB for each process's page table
    - Stored in main memory!
  - What happens for 64-bit logical address spaces?
- **#2 — Performance Impact**
  - Converting virtual to physical address requires multiple operations to access memory
    - Read Page Table Entry from memory!
    - Get page frame number
    - Construct physical address
    - Assorted protection and valid checks
  - Without fast hardware support, requires multiple memory accesses and a lot of work per logical address

Solution — associative memory

aka *dynamic address translation* — DAT

aka *translation lookaside buffer* — TLB

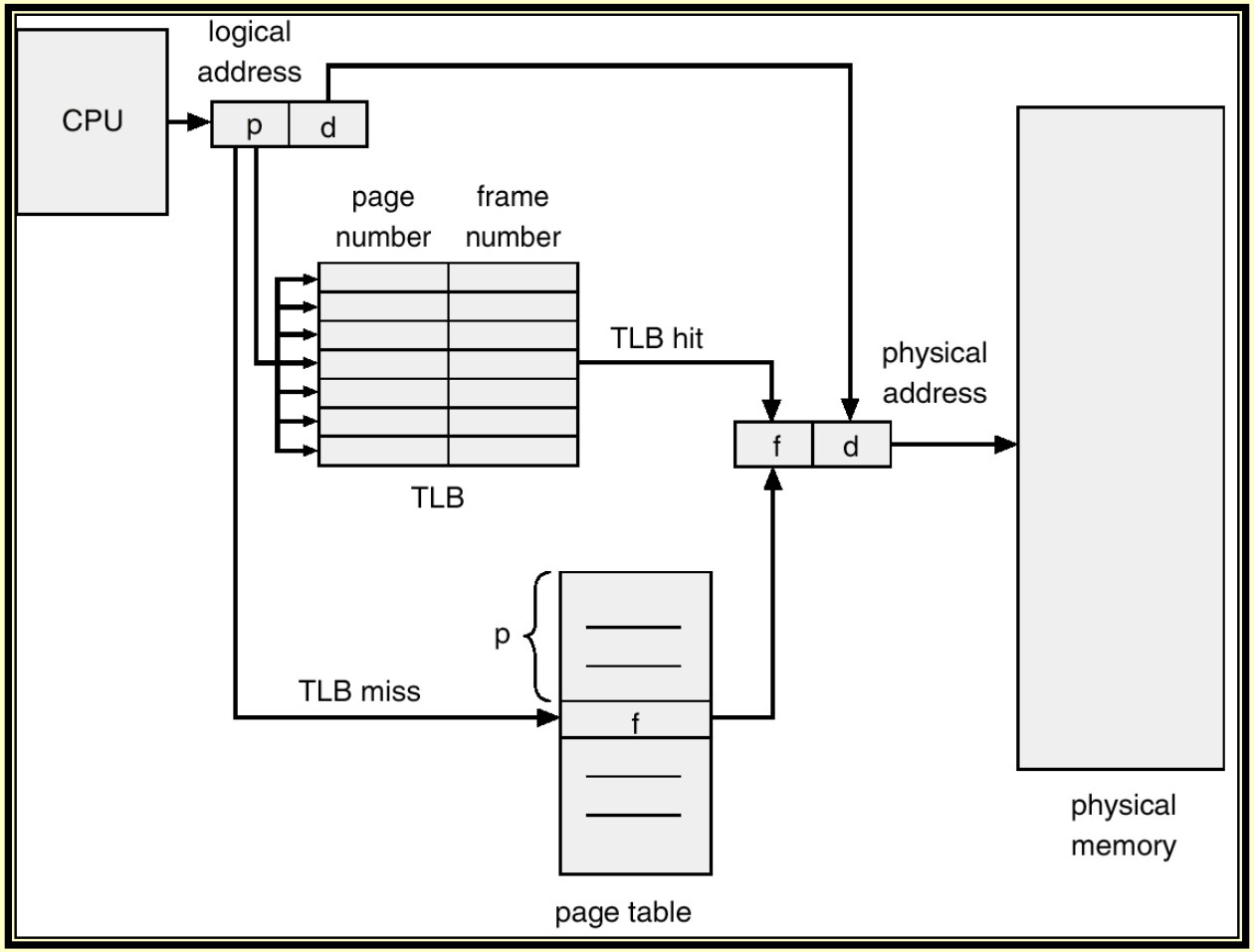


- **Do fast hardware search of all entries in parallel for VPN**
- **If present, use *page frame number* from PTE directly**
- **If not,**
  a) Look up in page table (multiple accesses)
  b) Load VPN and PTE into Associative Memory (throwing out another entry as needed)

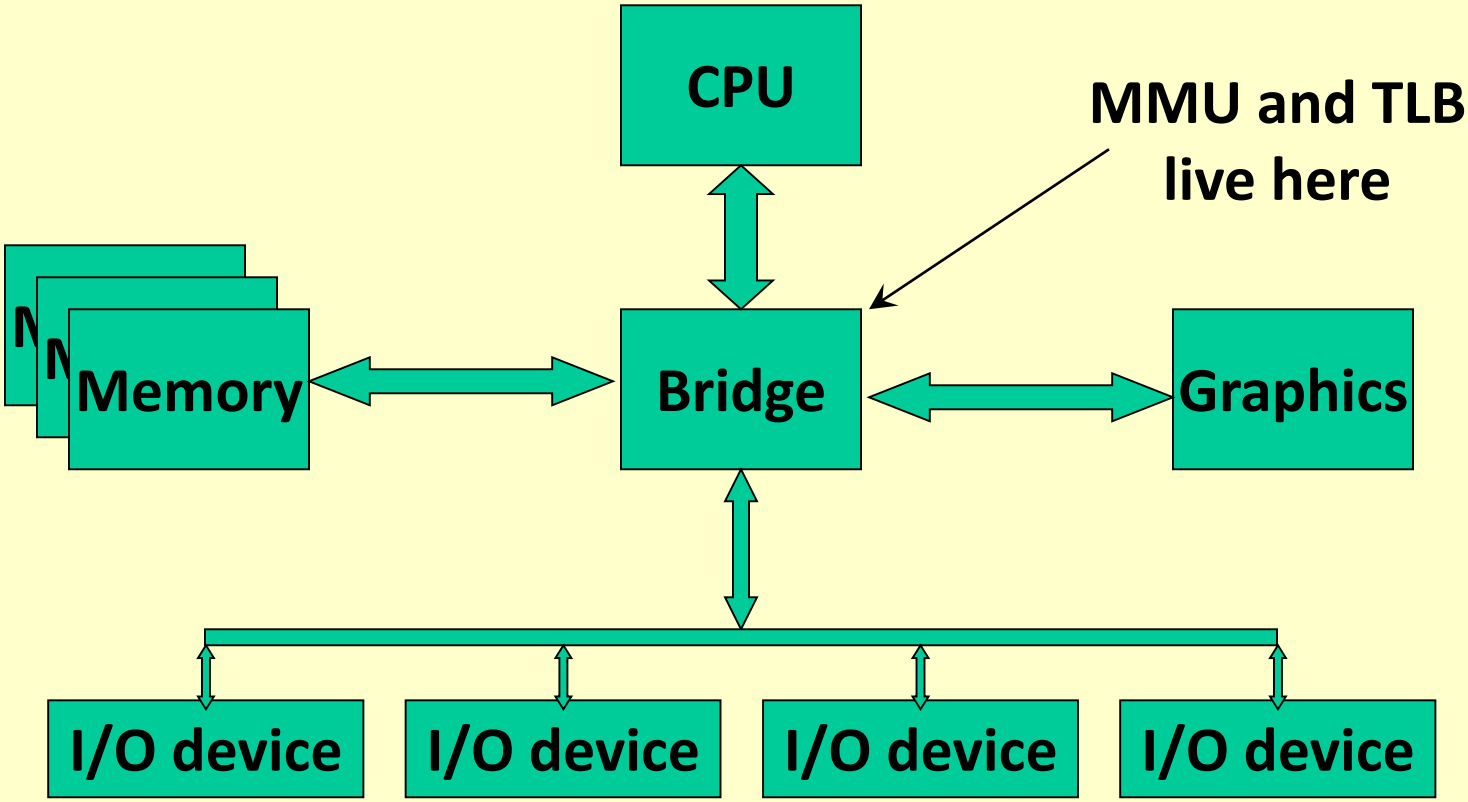
Translation lookaside buffer (TLB)

- **Associative memory implementation in hardware**
  - Translates VPN to PTE (containing PFN)
  - Done in single machine cycle
- **TLB is hardware assist**
  - Fully or partially *associative* – entries searched in parallel with VPN as index
  - Returns page frame number (PFN)
  - MMU use PFN and offset to get Physical Address
- ***Locality* makes TLBs work**
  - Usually have 8–1024 TLB entries
  - Sufficient to deliver 99%+ hit rate (in most cases)
- **Works especially well with multi-level page tables**

MMU with TLB


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Traditional machine architecture



TLB-related Policies

- **OS must ensure that TLB and page tables are consistent**
  - When OS changes bits (e.g. protection) in PTE, it must invalidate TLB copy
  - *Dirty* bit and reference bits in TLB must be written back to PTE
- **TLB replacement policies**
  - Random
  - Least Recently Used (LRU) – with hardware help
- **What happens on context switch?**
  - Each process has own page tables (possibly multi-level)
  - Must invalidate *all* TLB entries
  - Then TLB fills up again as new process executes
  - Expensive context switches just got more expensive!
    - Note benefit of Threads sharing same address space

Questions?

Paging – some tricks

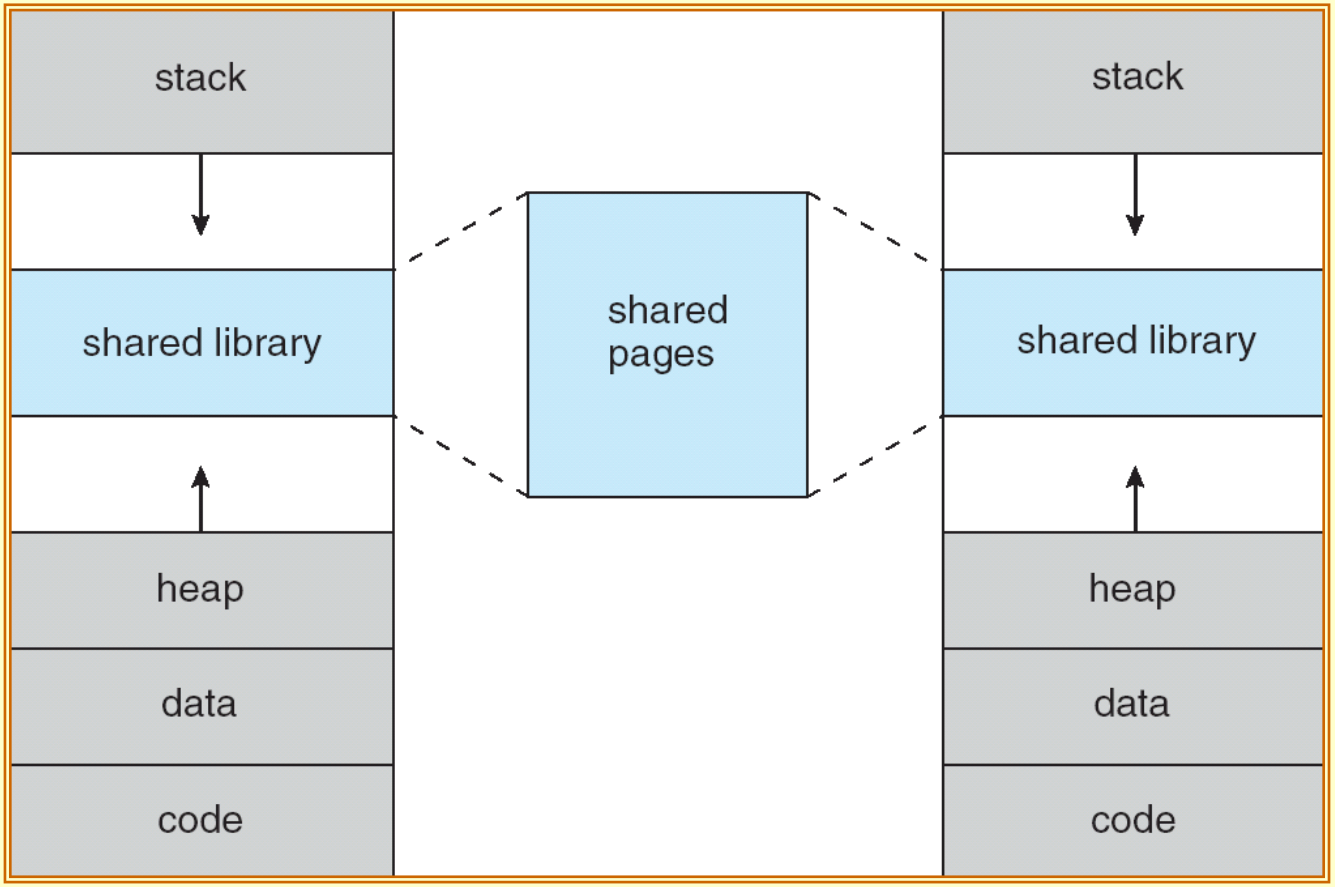
Paging – some tricks

- Shared Memory
- Copy on Write
- Mapping Files to Virtual Memory
- Guard pages, segments, and Virtual Memory Organization

Shared memory

- **Part of virtual memory of two or more processes *map* to same frames**
  - Finer grained sharing than segments
  - Data sharing with Read/Write
  - Shared libraries with eXecute
- **Each process has own PTEs – different privileges**

Shared pages (illustration)



Shared pages (continued)

- **Widely used for libraries of code**
  - Supported by most modern operating systems
- **Less widely used for sharing data memory**
  - **`shmget()`**, etc.
  - Not always as same virtual address in each virtual memory — causes complications
  - Old gimmick serving same purpose as threads today

Questions?

Paging – some tricks

- Shared Memory
- Copy on Write
- Mapping Files to Virtual Memory
- Guard pages, segments, and Virtual Memory Organization

Paging trick – *copy-on-write*

- **During `fork()`**
  - Don't copy individual pages of virtual memory
  - Just copy page tables; *all* pages start out shared
  - Set *all* PTEs to read-only in both processes
- **When either process hits a write-protection fault on a valid page *that should be writable***
  - Make a copy of that page, set write protect bit in PTE to allow writing
  - Resume faulting process
- **Saves a lot of unnecessary copying**
  - Especially if child process will delete and reload all of virtual memory with call to **`exec()`**

Paging – some tricks

- Shared Memory
- Copy on Write
- Mapping Files to Virtual Memory
- Guard pages, segments, and Virtual Memory Organization

Definition — *mapping*

- ***Mapping* (part of) *virtual memory to a file***

≡

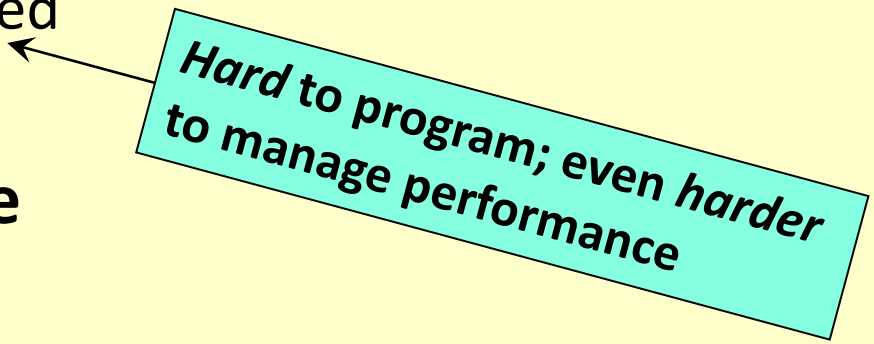
Assigning blocks of file as *backing store* for the virtual memory pages, so that

- *Page faults* in (that part of) virtual memory resolve to the blocks of the file
- *Dirty pages* in (that part of) virtual memory are written back to the blocks of the file

(Almost) always used for executable code (read-only)

Warning — memory-mapped files

- **Tempting idea:–**
  - Instead of standard I/O calls (`read()`, `write()`, etc.) *map* file starting at virtual address X
  - Access to "X + *n*" refers to file at offset *n*
  - Use paging mechanism to read file blocks into physical memory on demand …
  - … and write them out as needed
- **Has been tried many times**
- **Rarely works well in practice**

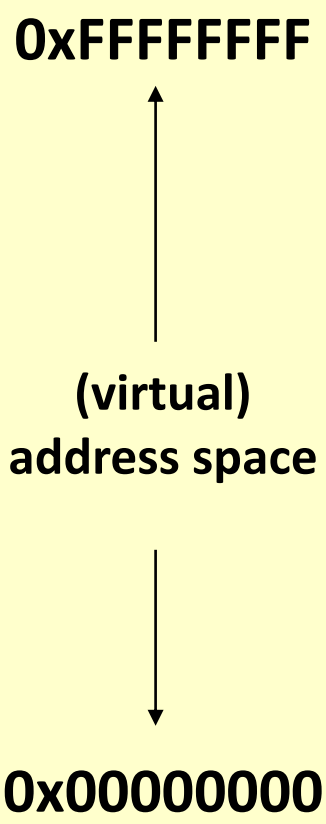


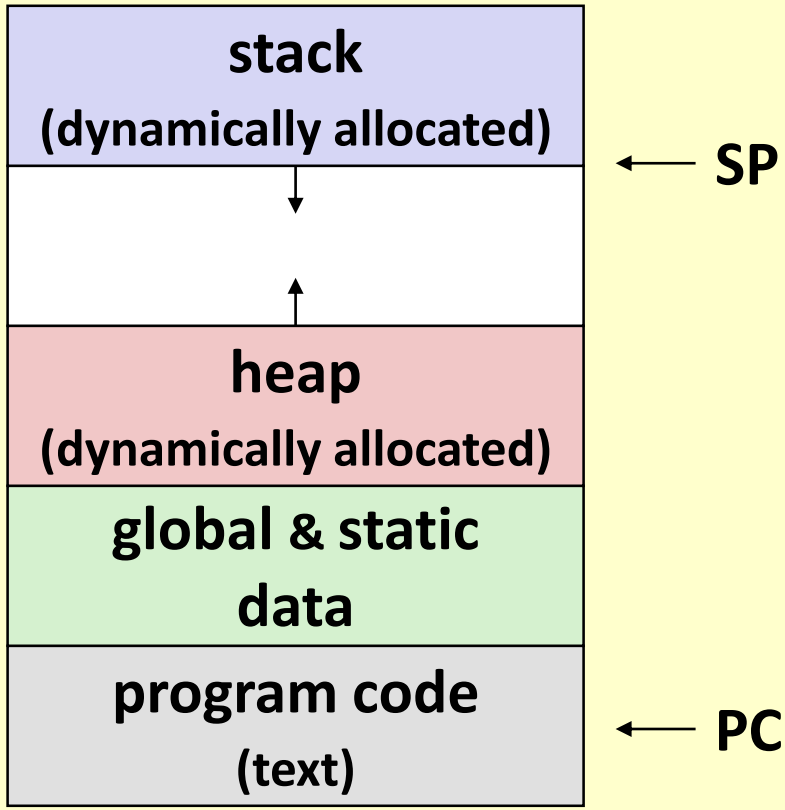
Paging – some tricks

- Shared Memory
- Copy on Write
- Mapping Files to Virtual Memory
- **Guard pages, segments, and Virtual Memory Organization**

Process address space (traditional unix)

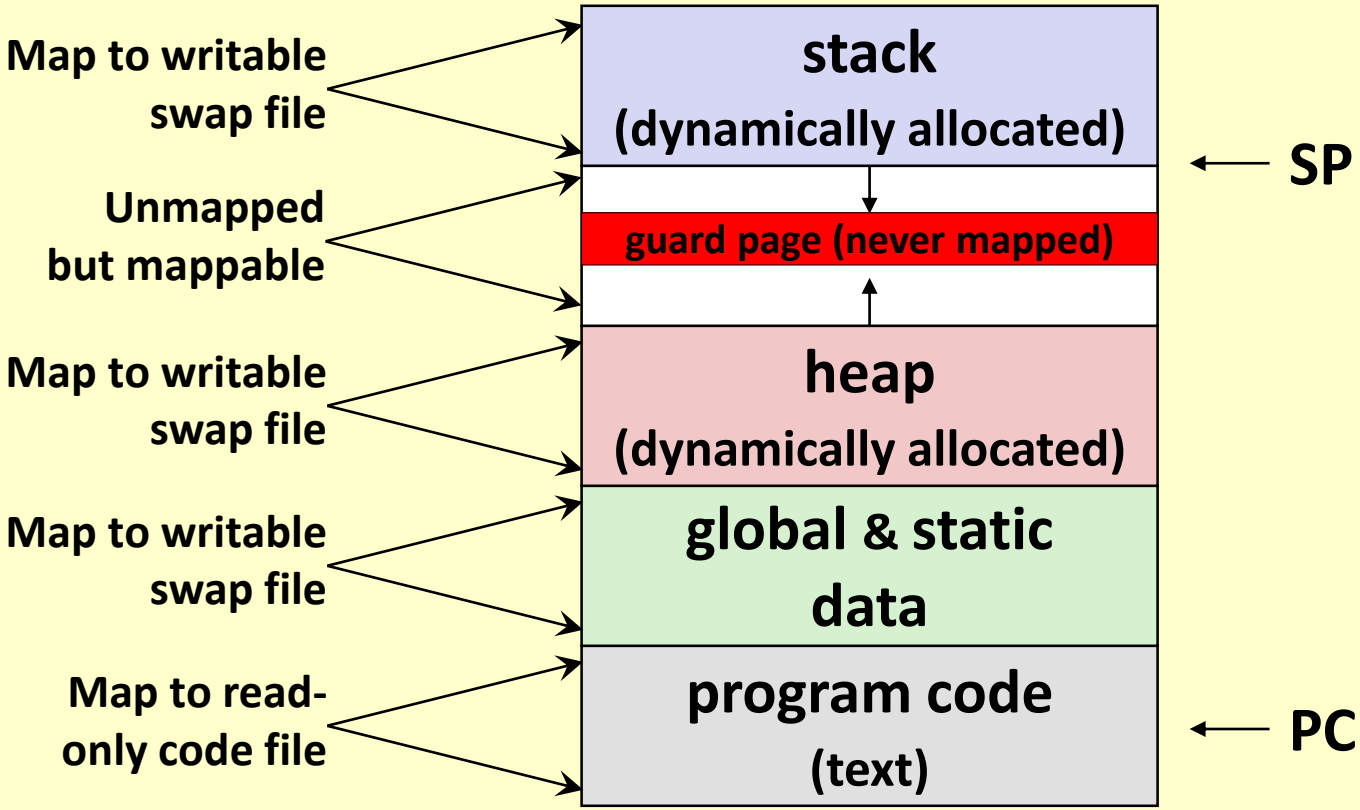
From an earlier topic



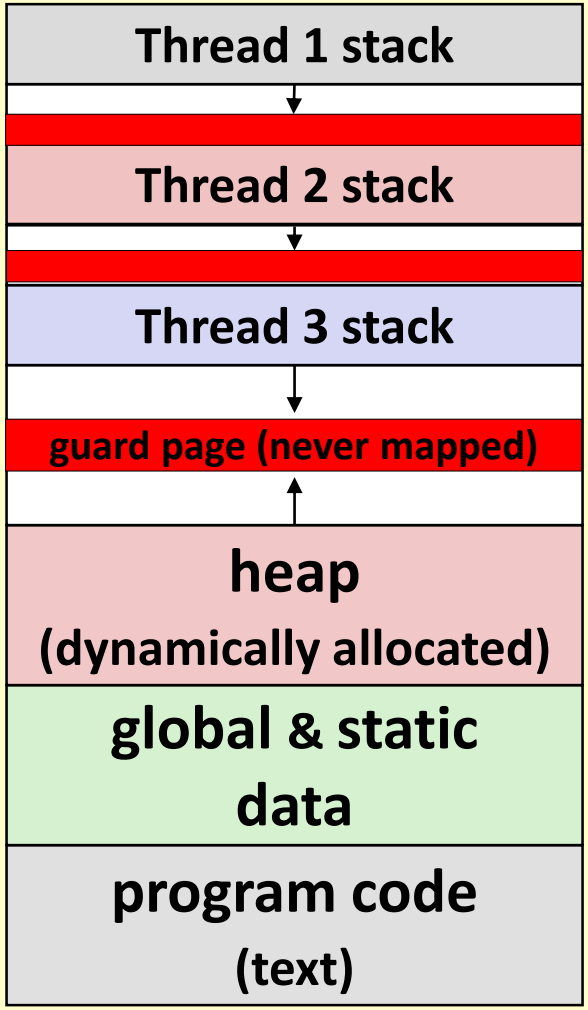


Process address space (traditional unix)

From an earlier topic



Address space for multiple threads



Virtual memory organization (continued)

- **Each area of process VM is its own segment (or sequence of segments)**
  - Executable code & constants – fixed size
  - Shared libraries
  - Static data – fixed size
  - Heap – open-ended
  - Stacks – each open-ended
  - Other segments as needed
    - E.g., symbol tables, loader information, etc.

Summary

Virtual memory – a major OS abstraction

- **Processes execute in *virtual memory*, not physical memory**
- **Virtual memory may be very large**
  - Much larger than physical memory
- **Virtual memory may be organized in interesting & useful ways**
  - Open-ended segments
- ***Virtual memory is where the action is!***
  - *Physical memory is just a* cache *of virtual memory*

Reading assignment

- **OSTEP**
  - §§ 13 – 17 (previous topic – *Memory Management*)
  - §§ 18 – 19 (this topic on *Paging*)

Questions?

Related topic – *Caching*