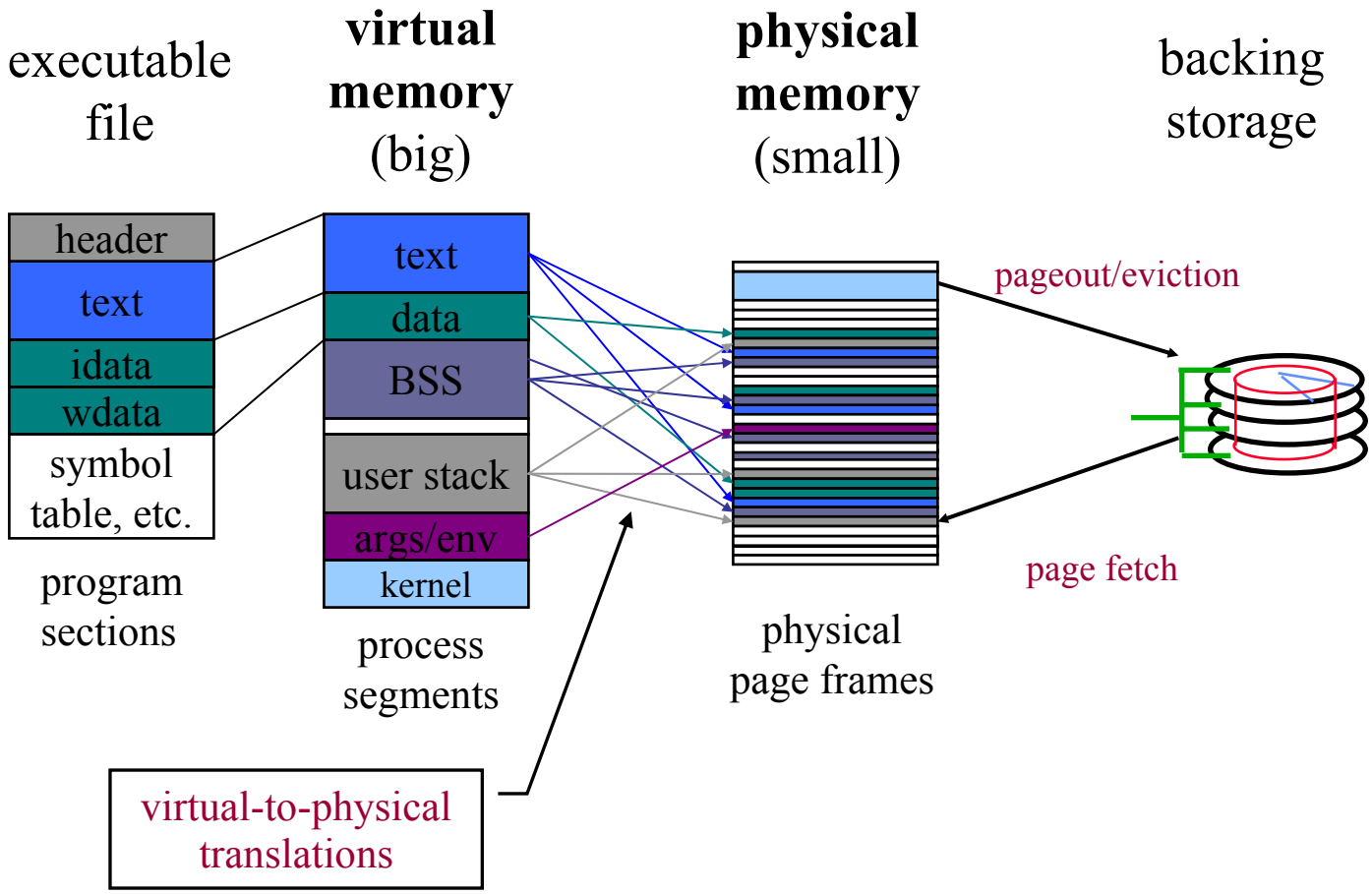
Virtual Memory and Demand Paging

Dr. Daniel Andresen

CIS520 – Operating Systems

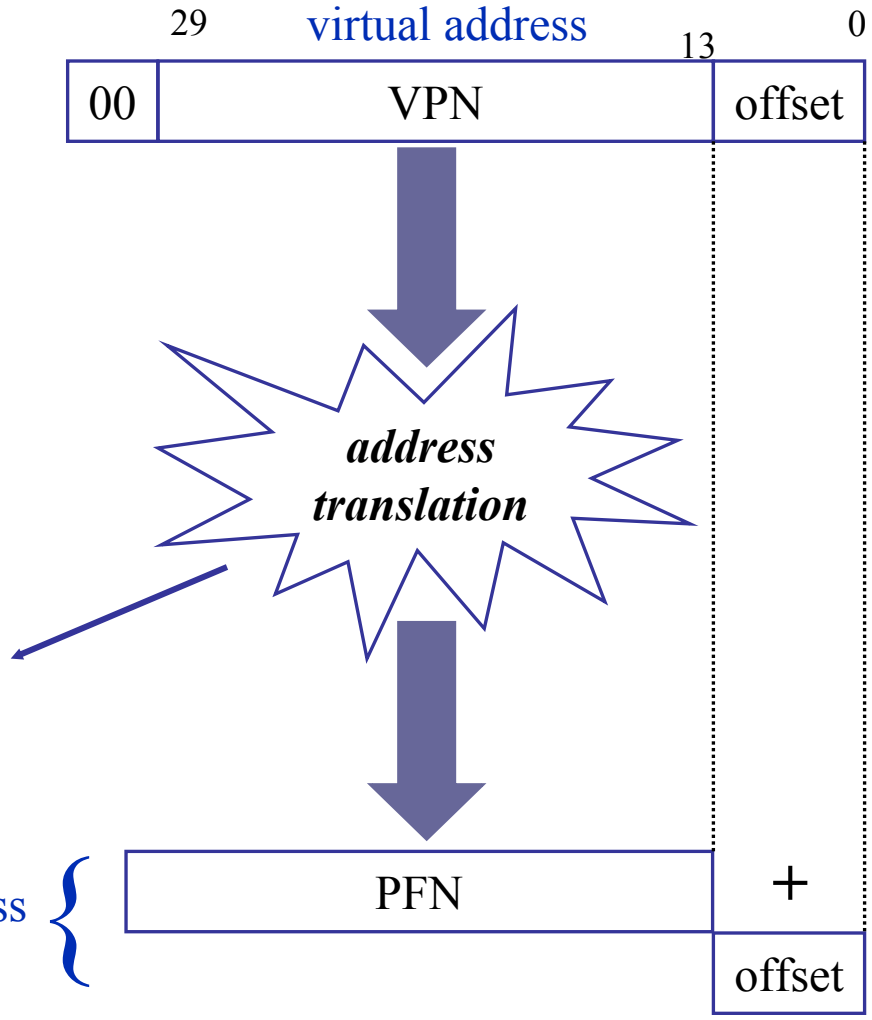
Virtual Memory Illustrated



Virtual Address Translation

Example: typical 32-bit architecture with 8KB pages.

Virtual address translation maps a *virtual page number* (VPN) to a physical *page frame number* (PFN): the rest is easy.



Role of MMU Hardware and OS

VM address translation must be very cheap (on average).

- Every instruction includes one or two memory references.

  (including the reference to the instruction itself)

VM translation is supported in hardware by a ***M***emory ***M***anagement ***U***nit or *MMU*.

- The addressing model is defined by the CPU architecture.
- The MMU itself is an integral part of the CPU.

The role of the OS is to install the virtual-physical mapping and intervene if the MMU reports that it cannot complete the translation.

The Translation Lookaside Buffer (TLB)

An on-chip hardware *translation buffer* (TB or TLB) caches recently used virtual-physical translations (ptes).

Alpha 21164: 48-entry fully associative TLB.

A CPU pipeline stage probes the TLB to complete over 99% of address translations in a single cycle (*hit rate*).

Like other memory system caches, replacement of TLB entries is simple and controlled by hardware.

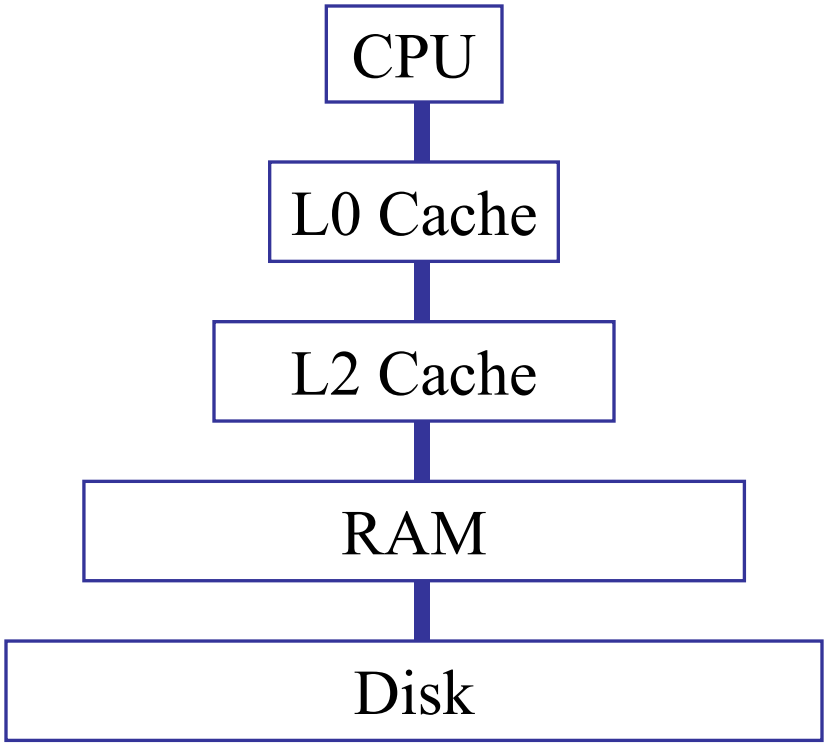
e.g., Not Last Used

If a translation misses in the TLB, the entry must be fetched by accessing the page table(s) in memory.

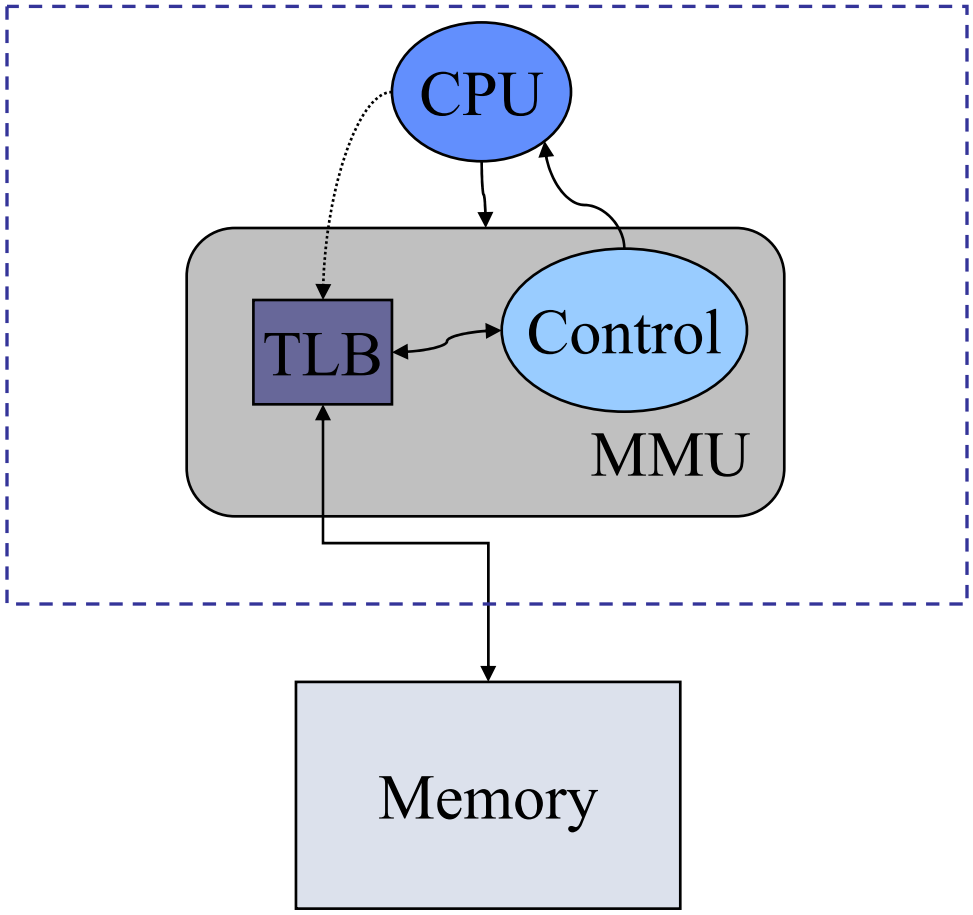
cost: 10-200 cycles

Aside: the Memory Hierarchy

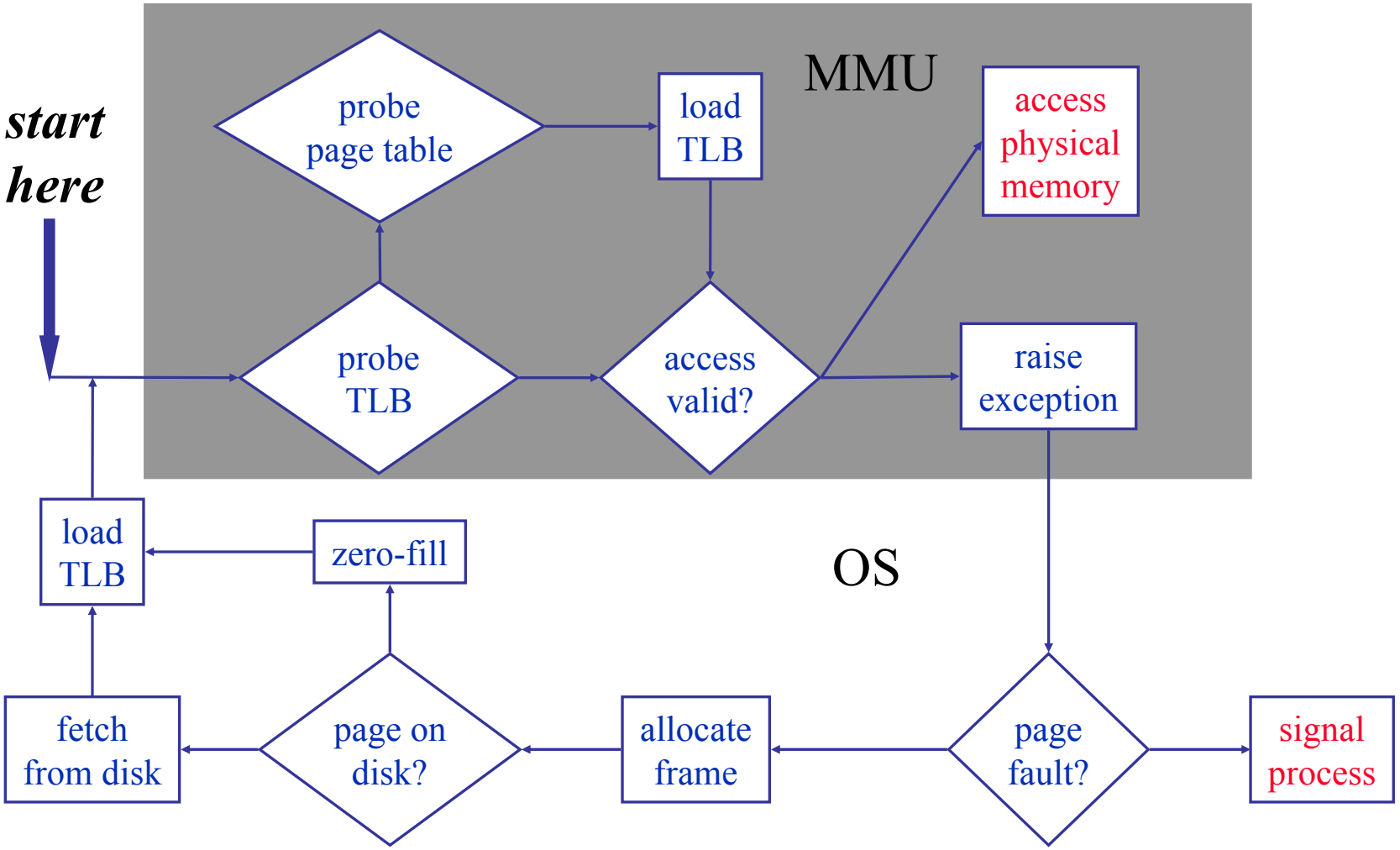
- "All of computer science is an exercise in cache management"
- Key is ***locality***
- TLB adds new dimension – locality of the page table
- What types of programs work best with these systems?
- Cache coherency becomes a big issue



A View of the MMU and the TLB



Completing a VM Reference



The OS Directs the MMU

The OS controls the operation of the MMU to select:

(1) the subset of possible virtual addresses that are valid for each process (the process *virtual address space*);

(2) the physical translations for those virtual addresses;

(3) the modes of permissible access to those virtual addresses;

read/write/execute

(4) the specific set of translations in effect at any instant.

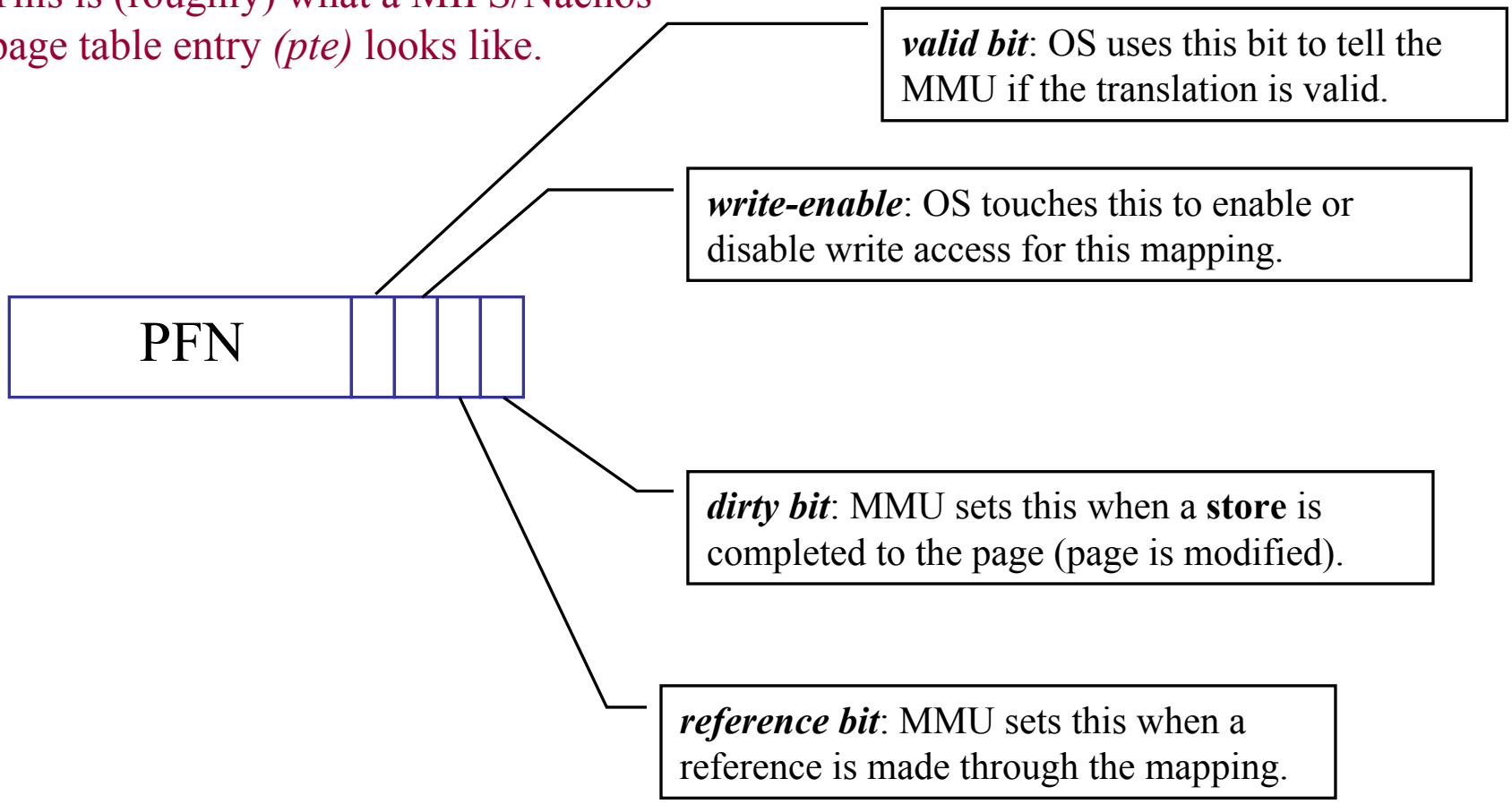
need rapid context switch from one address space to another

MMU completes a reference only if the OS "says it's OK".

MMU raises an exception if the reference is "not OK".

A Page Table Entry (PTE)

This is (roughly) what a MIPS/Nachos page table entry *(pte)* looks like.



Paged Virtual Memory

Like the file system, the paging system manages physical memory as a *page cache* over a larger virtual store.

- Pages not resident in memory can be zero-filled or found somewhere on secondary storage.
- MMU and TLB handle references to resident pages.
- A reference to a non-resident page causes the MMU to raise a *page fault* exception to the OS kernel.

  Page fault handler validates access and services the fault.

  Returns by restarting the faulting instruction.

- Page faults are (mostly) transparent to the interrupted code.

Care and Feeding of TLBs

The OS kernel carries out its memory management functions by issuing *privileged* operations on the MMU.

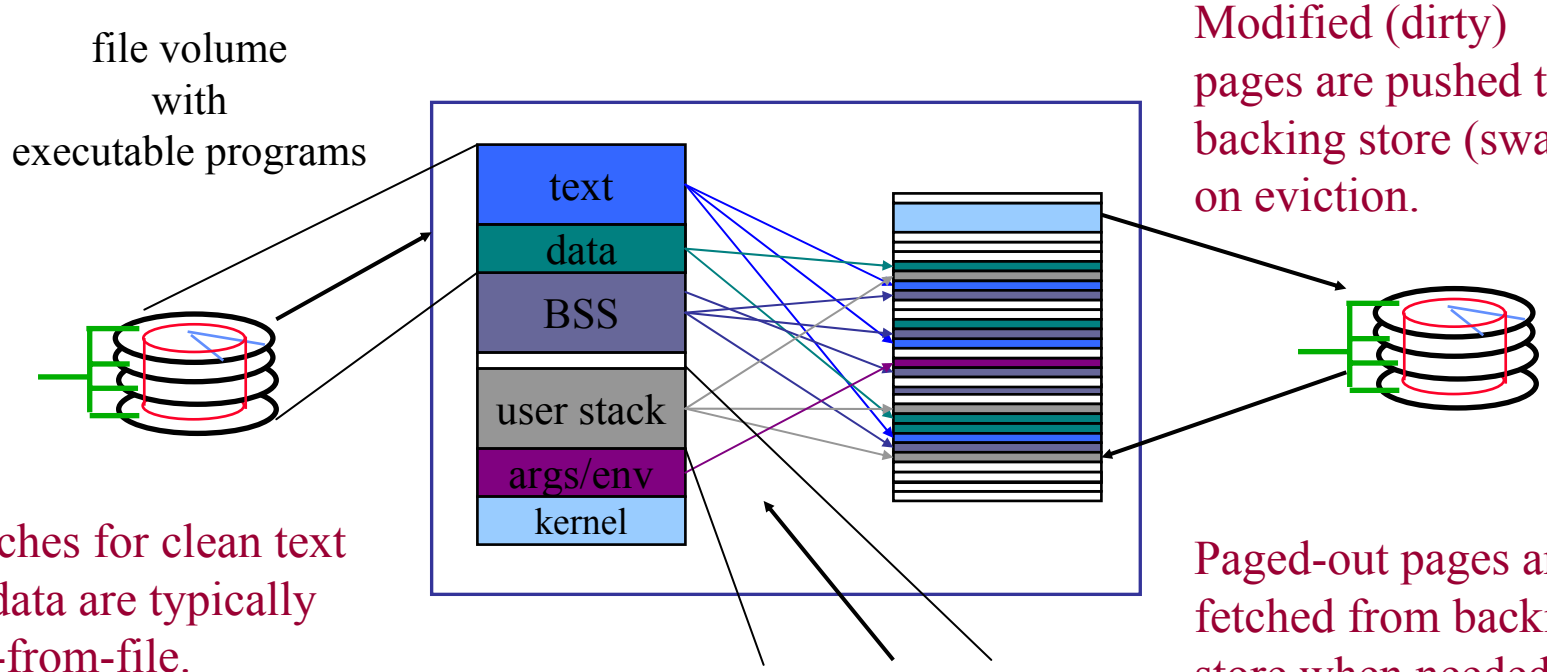
Choice 1: OS maintains page tables examined by the MMU.

- MMU loads TLB autonomously on each TLB miss
- page table format is defined by the architecture
- OS loads page table bases and lengths into privileged *memory management registers* on each context switch.

Choice 2: OS controls the TLB directly.

- MMU raises exception if the needed pte is not in the TLB.
- Exception handler loads the missing pte by reading data structures in memory (*software-loaded TLB*).

Where Pages Come From



Demand Paging and Page Faults

OS may leave some virtual-physical translations unspecified.

- mark the pte for a virtual page as *invalid*

If an unmapped page is referenced, the machine passes control to the kernel exception handler (page fault).

- passes faulting virtual address and attempted access mode

Handler initializes a page frame, updates pte, and restarts.

- If a disk access is required, the OS may switch to another process after initiating the I/O.
- Page faults are delivered at IPL 0, just like a system call trap.
- Fault handler executes in context of faulted process, blocks on a semaphore or condition variable awaiting I/O completion.

Issues for Paged Memory Management

The OS tries to minimize page fault costs incurred by all processes, balancing fairness, system throughput, etc.

(1) fetch policy: When are pages brought into memory?

prepaging: reduce page faults by bring pages in before needed

clustering: reduce seeks on backing storage

(2) replacement policy: How and when does the system select victim pages to be evicted/discarded from memory?

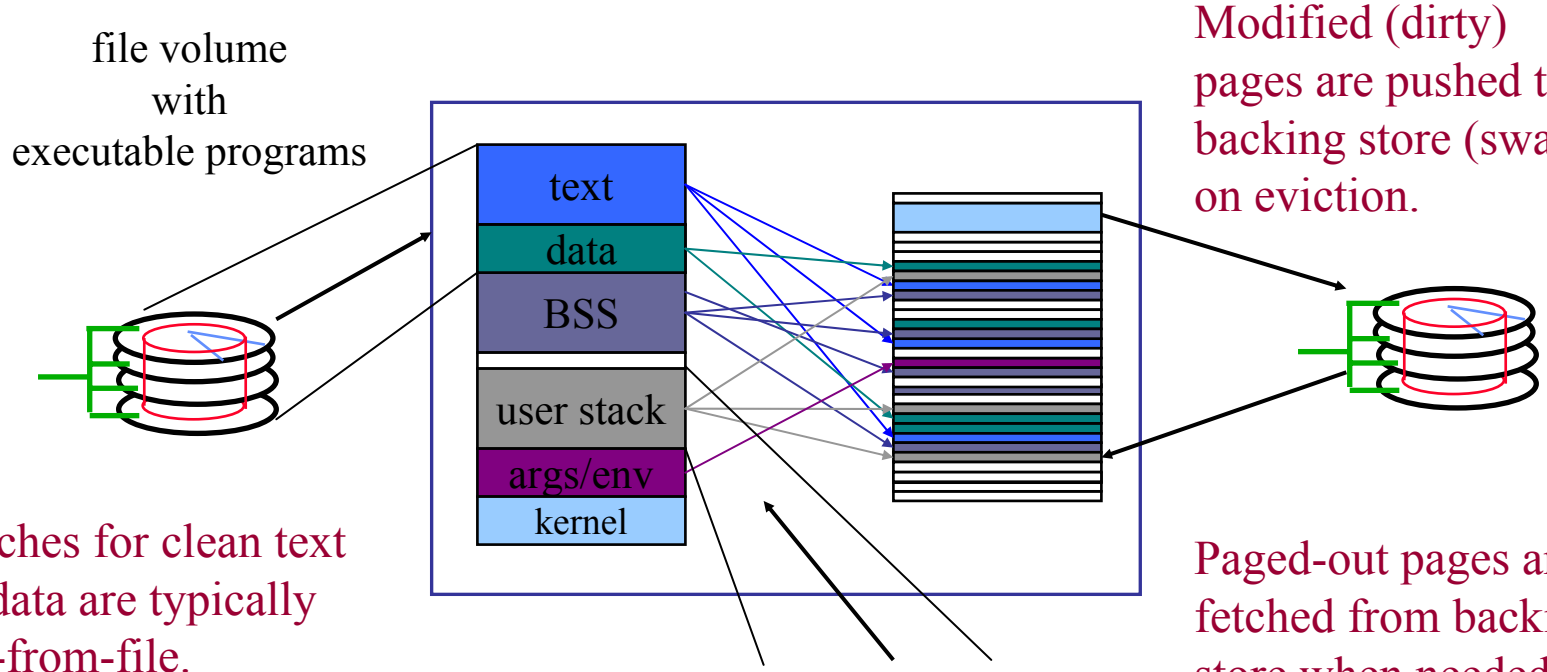
(3) backing storage policy:

Where does the system store evicted pages?

When is the backing storage allocated?

When does the system write modified pages to backing store?

Where Pages Come From



Questions for Paged Virtual Memory

1. How do we prevent users from accessing protected data?
2. If a page is in memory, how do we find it?

   Address translation must be fast.
3. If a page is not in memory, how do we find it?
4. When is a page brought into memory?
5. If a page is brought into memory, where do we put it?
6. If a page is evicted from memory, where do we put it?
7. How do we decide which pages to evict from memory?

   Page replacement policy should minimize overall I/O.
8. How big should pages be?

Mapped Files

With appropriate support, virtual memory is a useful basis for accessing file storage (vnodes).

- bind file to a region of virtual memory with *mmap* syscall.

  e.g., start address *x*

  virtual address *x*+*n* maps to offset *n* of the file

- several advantages over stream file access

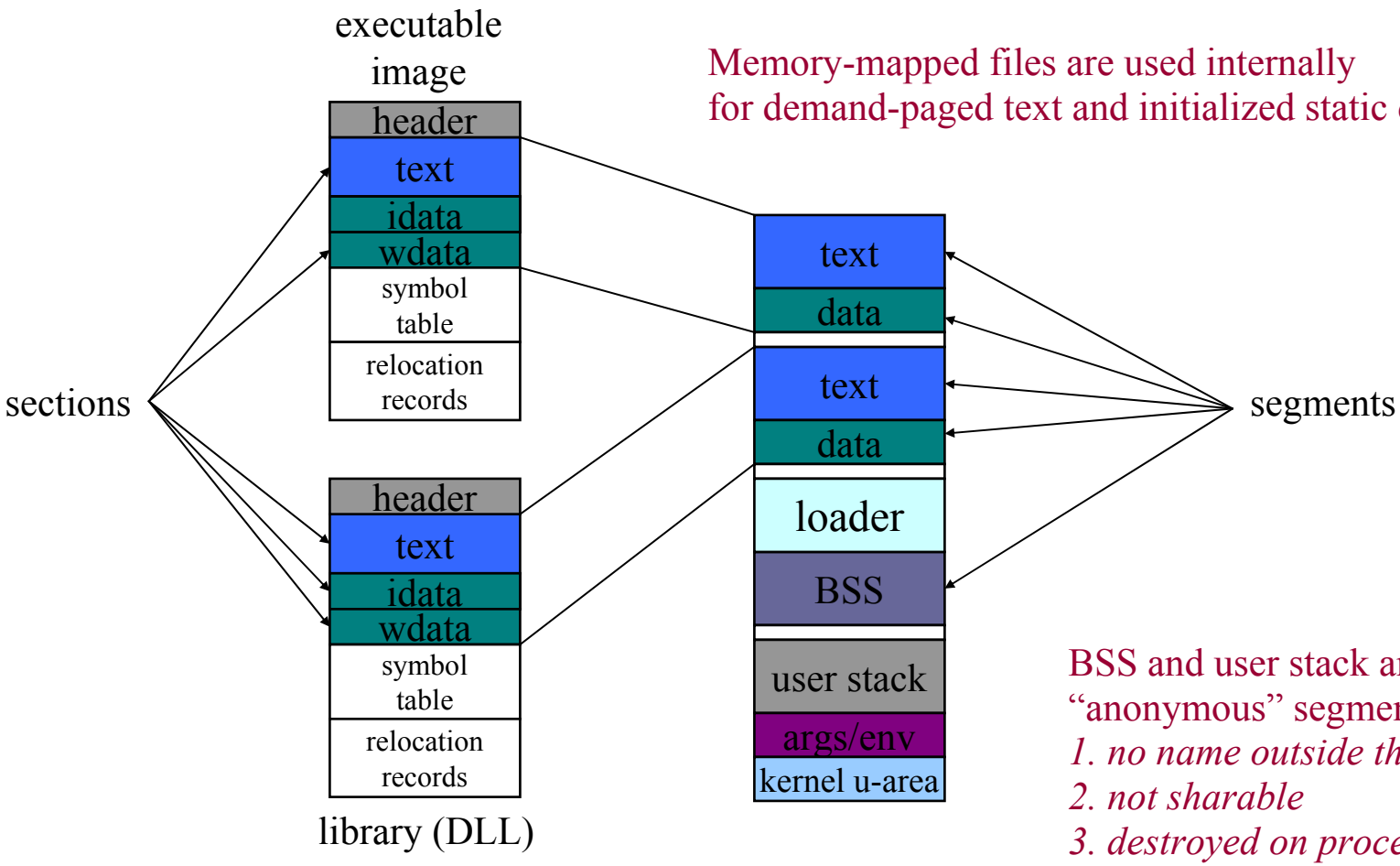
  uniform access for files and memory (just use pointers)

  performance: zero-copy reads and writes for low-overhead I/O

  but: program has less control over data movement

  style does not generalize to pipes, sockets, terminal I/O, etc.

Using File Mapping to Build a VAS



Memory-mapped files are used internally
for demand-paged text and initialized static data.

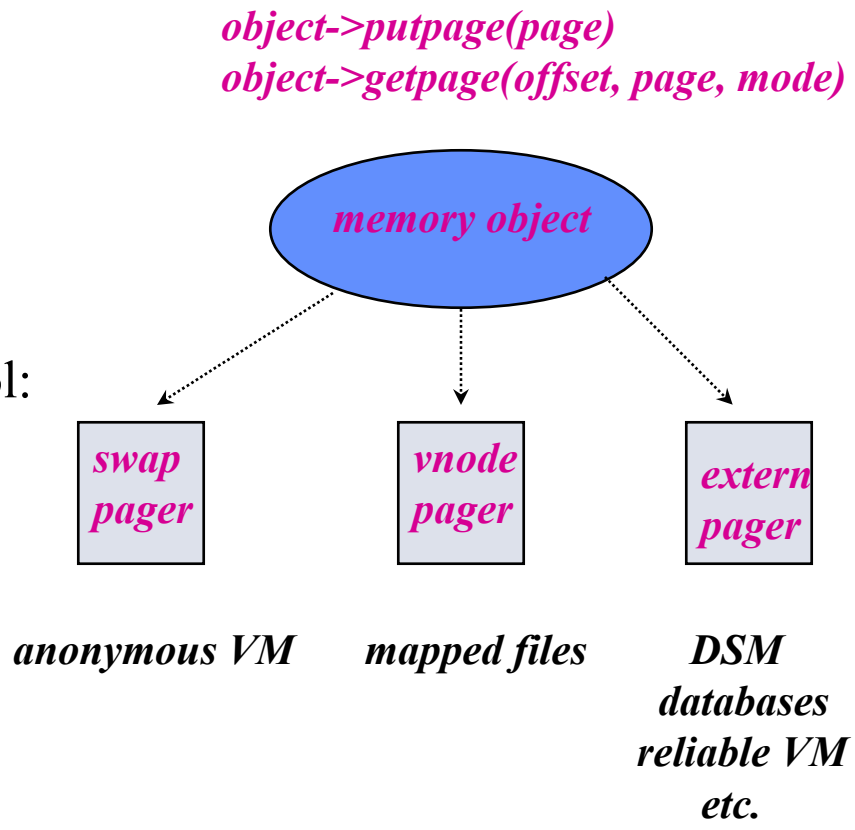
BSS and user stack are
"anonymous" segments.

1. no name outside the process
2. not sharable
3. destroyed on process exit

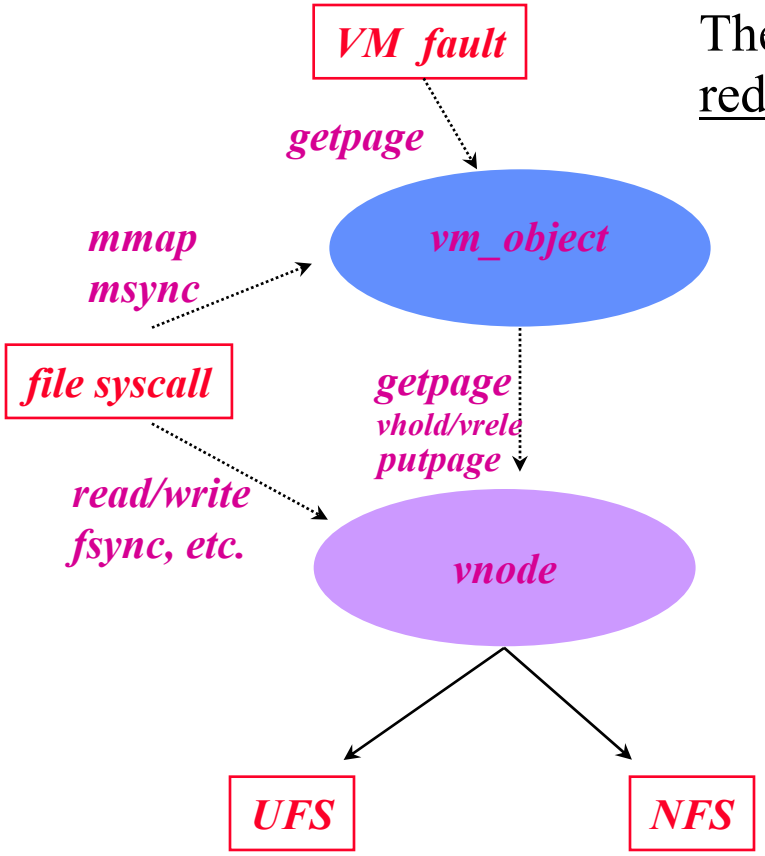
Memory Objects

Memory objects "virtualize" VM backing storage policy.

- source and sink for pages
  - triggered by faults
  - ...or OS eviction policy
- manage their own storage
- external pager has some control:
  - prefetch
  - prewrite
  - protect/enable
- can be shared via *vm_map()*
  - (Mach extended *mmap syscall)*



The Block/Page I/O Subsystem



The VFS/memory object/pmap framework reduces VM and file access to the central issue:

How does the system handle a stream of get/put block/page operations on a collection of vnodes and memory objects?

- executable files
- data files
- anonymous paging files (swap files)
- reads on demand from file syscalls
- reads on demand from VM page faults
- writes on demand

To deliver good performance, we must manage system memory as an I/O *cache* of pages and blocks.

So we have a conflict – how much memory should we devote to either kind of data?

Shadow Objects and Copy-on-Write

Operating systems spend a lot of their time copying data.

- particularly Unix operating systems, e.g., *fork()*
- cross-address space copies are common and expensive

Idea: defer big copy operations as long as possible, and hope they can be avoided completed.

- create a new *shadow* object backed by an existing object
- shared pages are mapped readonly in participating spaces

  read faults are satisfied from the original object (typically)

  write faults trap to the kernel

    make a (real) copy of the faulted page

    install it in the shadow object with writes enabled

A Copy-on-Write Mapping