CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu

Lecture 9: Virtual Memory
March 1, 2007

Memory Management

Today we start a series of lectures on memory mangement

Goals of memory management

- Convenient abstraction for programming
- Provide isolation between different processes
- Allocate scarce physical memory resources across processes
    - *Especially important when memory is heavily contended for*
- Minimize overheads

Mechanisms

- Virtual address translation (today)
- Paging and TLBs (today)
- Page table management (next lecture)

Policies

- Page replacement policies (next Thursday)

Virtual Memory

The basic abstraction provided by the OS for memory management

VM enables programs to execute without requiring their entire address space to be resident in physical memory

- Program can run on machines with less physical RAM than it “needs”

Observation: Many programs don’t use all of their code or data

- e.g., branches they never take, or variables never accessed
- Observation: No need to allocate memory for it until it's used
- OS should adjust amount allocated based on its run-time behavior

Virtual Memory

Virtual memory also *isolates* processes from each other

- One process cannot access memory addresses in others
- Each process has its own isolated address space

VM requires **both** hardware and OS support

- Hardware support: *memory management unit* (MMU) and *translation lookaside buffer* (TLB)
- OS support: *virtual memory system* to control the MMU and TLB

Memory Management Requirements

Protection

- Restrict which addresses processes can use, so they can't stomp on each other

Fast translation

- Accessing memory must be fast, regardless of the protection scheme
  - *(Would be a bad idea to have to call into the OS for every memory access!!)*

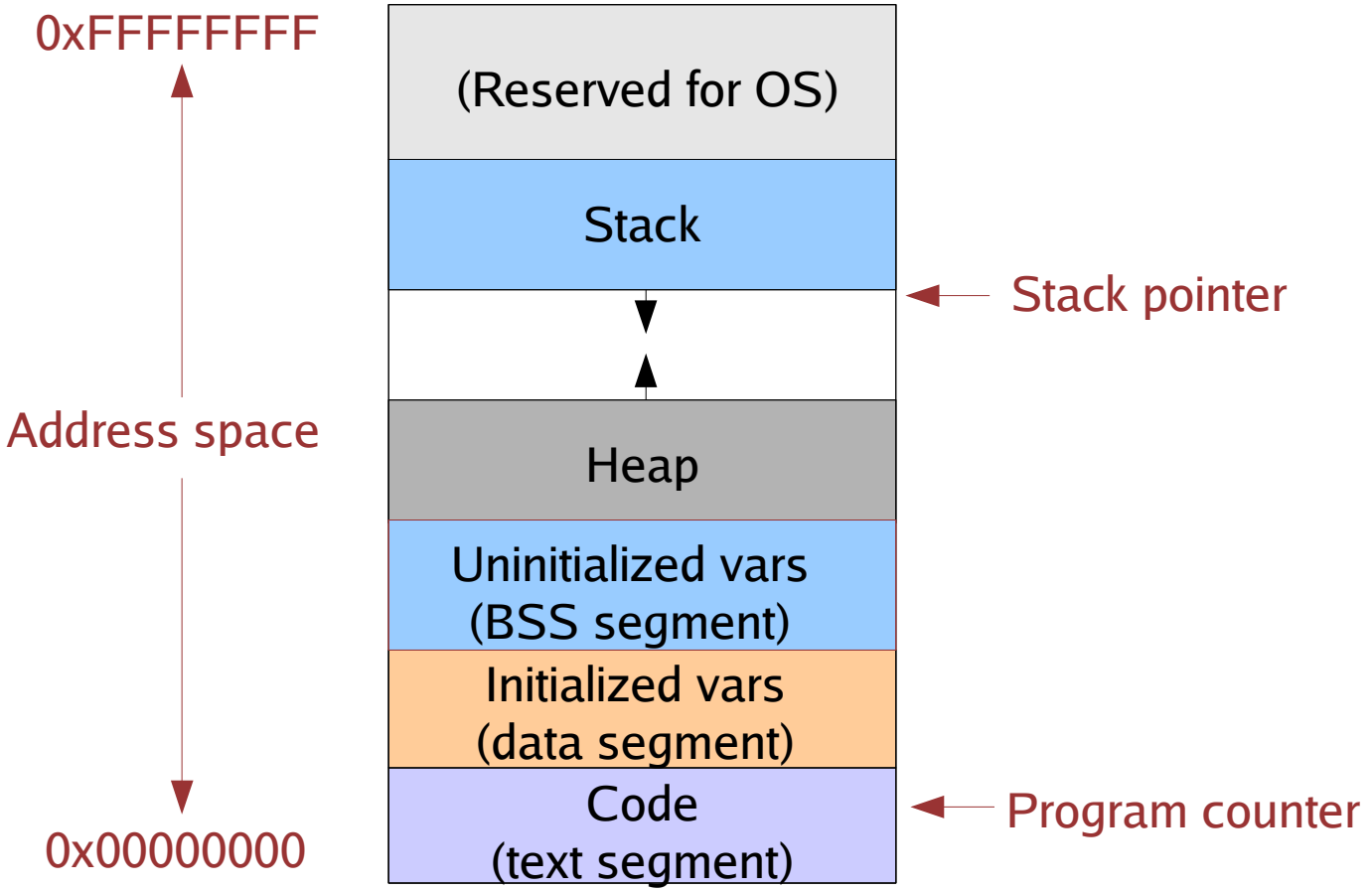
Fast context switching

- Overhead of updating memory hardware on a context switch must be low
  - *(For example, it would be a bad idea to copy all of a process's memory out to disk on every context switch.)*

Virtual Addresses

A *virtual address* is a memory address that a process uses to access its own memory

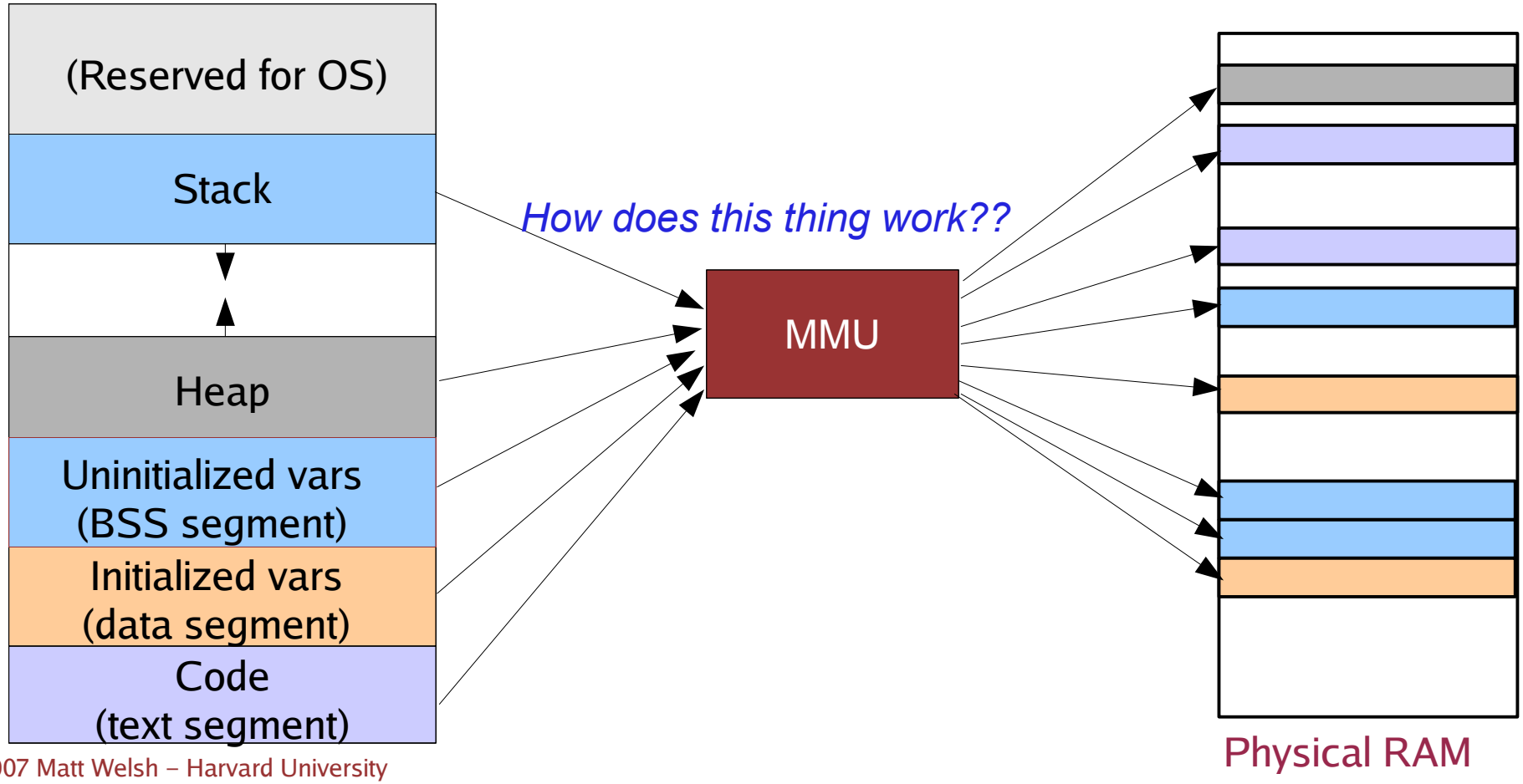
- The virtual address is *not the same* as the physical RAM address in which it is stored
- When a process accesses a virtual address, the MMU hardware *translates* the virtual address into a physical address
- The OS determines the *mapping* from virtual address to physical address



Virtual Addresses

A *virtual address* is a memory address that a process uses to access its own memory

- The virtual address is *not the same* as the actual physical RAM address in which it is stored
- When a process accesses a virtual address, the MMU hardware *translates* the virtual address into a physical address
- The OS determines the mapping from virtual address to physical address



Virtual Addresses

A *virtual address* is a memory address that a process uses to access its own memory

- The virtual address is *not the same* as the actual physical RAM address in which it is stored
- When a process accesses a virtual address, the MMU hardware *translates* the virtual address into a physical address
- The OS determines the mapping from virtual address to physical address

Virtual addresses allow *isolation*

- Virtual addresses in one process refer to **different** physical memory than virtual addresses in another
  - *Exception: shared memory regions between processes (discussed later)*

Virtual addresses allow *relocation*

- A program does not need to know which physical addresses it will use when it's run
  - *Obviously this would be a bad idea...*
- Compilers generate *relocatable code* – code that is independent of physical location in memory

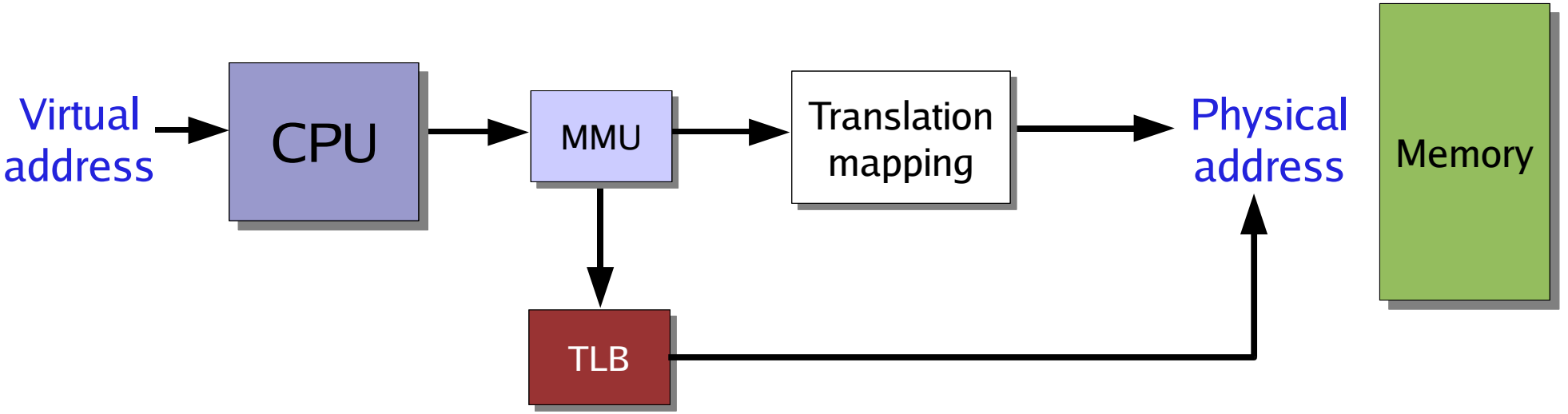
MMU and TLB

Memory Management Unit (MMU)

- Hardware that translates a virtual address to a physical address
- Each memory reference is passed through the MMU
- Translate a virtual address to a physical address
  - *Lots of ways of doing this!*

Translation Lookaside Buffer (TLB)

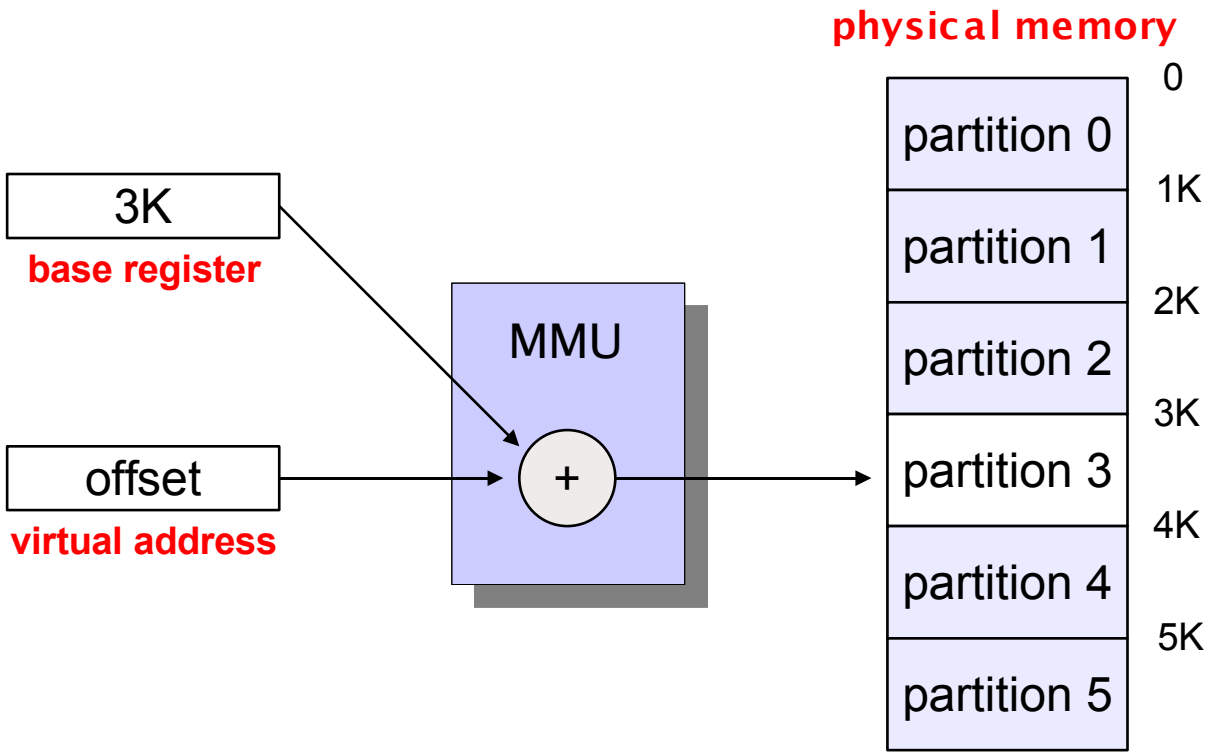
- **Cache** for MMU virtual-to-physical address translations
- Just an optimization – but an important one!



Simple approach: Fixed Partitions

Original memory management technique:
Break memory into fixed-size *partitions*

- Hardware requirement: *base register*
- Translation from virtual to physical address: simply add base register to vaddr



Advantages and disadvantages of this approach??

Simple approach: Fixed Partitions

Advantages:

- Fast context switch – only need to update base register
- Simple memory management code: Locate empty partition when running new process

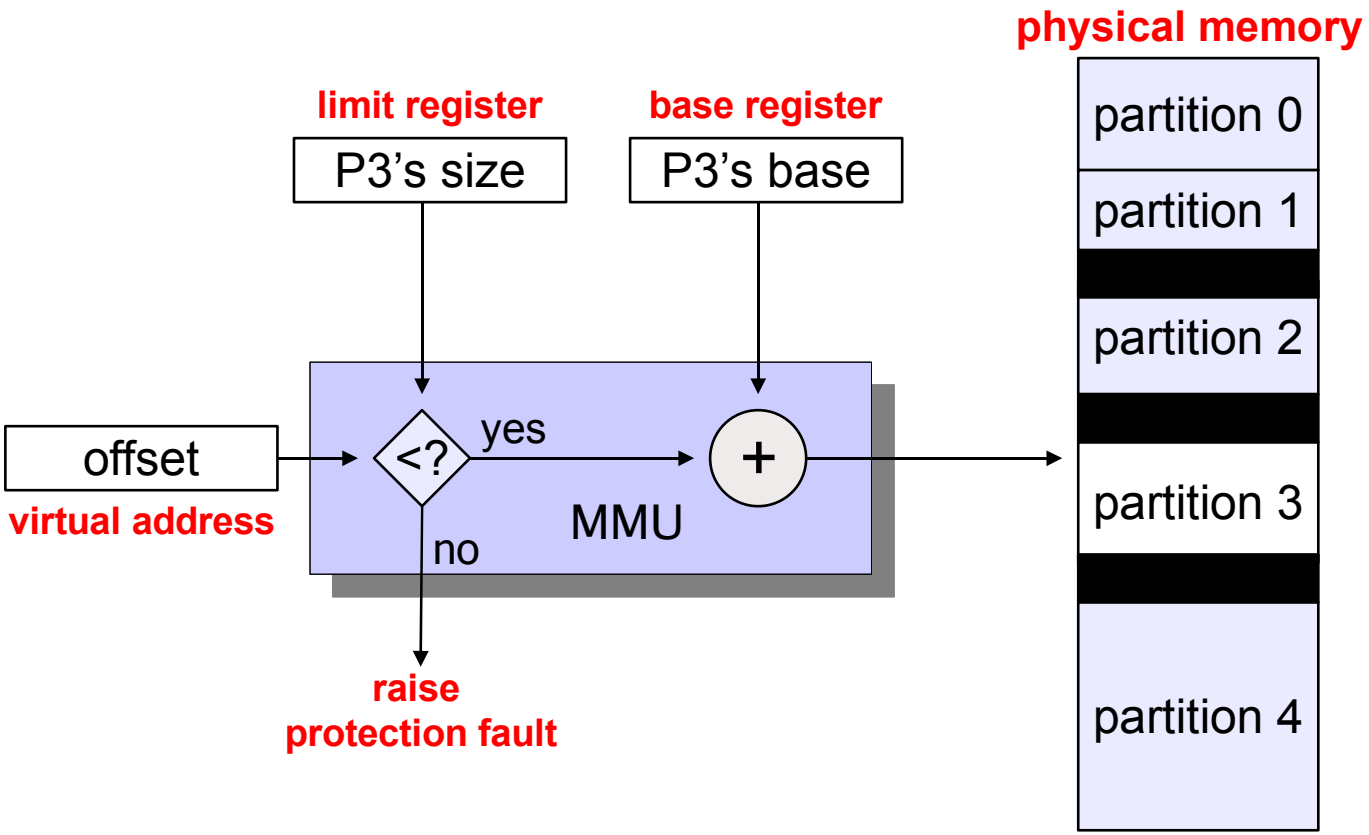
Disadvantages:

- Internal fragmentation
  - *Must consume entire partition, rest of partition is "wasted"*
- Static partition sizes
  - *No single size is appropriate for all programs!*

Variable Partitions

Obvious next step: Allow variable-sized partitions

- Now requires both a *base register* and a *limit register* for performing memory access
- Solves the internal fragmentation problem: size partition based on process needs



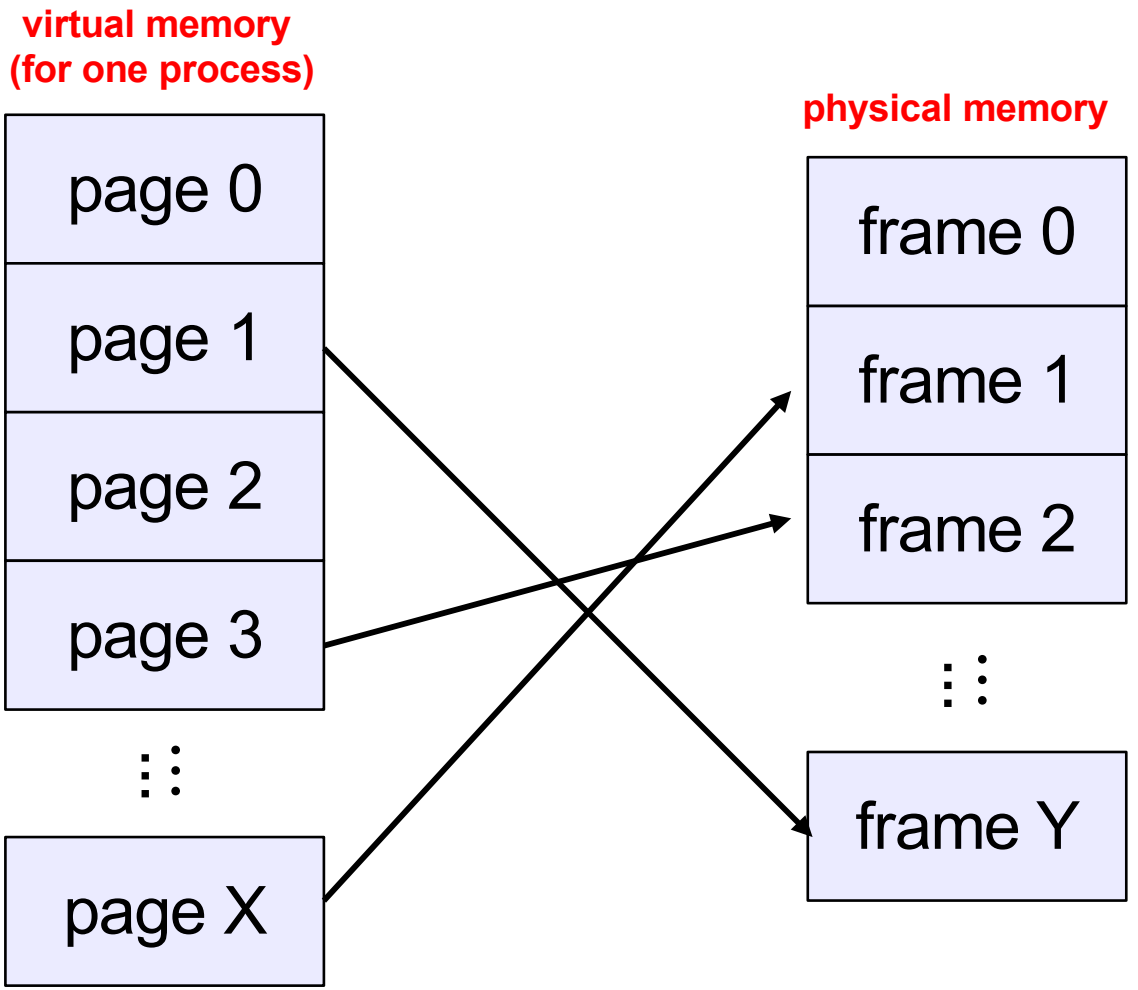
New problem: *external fragmentation*

- As jobs run and complete, holes are left in physical memory

Modern technique: Paging

Solve the external fragmentation problem by using fixed-size chunks of virtual and physical memory

- Virtual memory unit called a *page*
- Physical memory unit called a *frame* (or sometimes *page frame*)



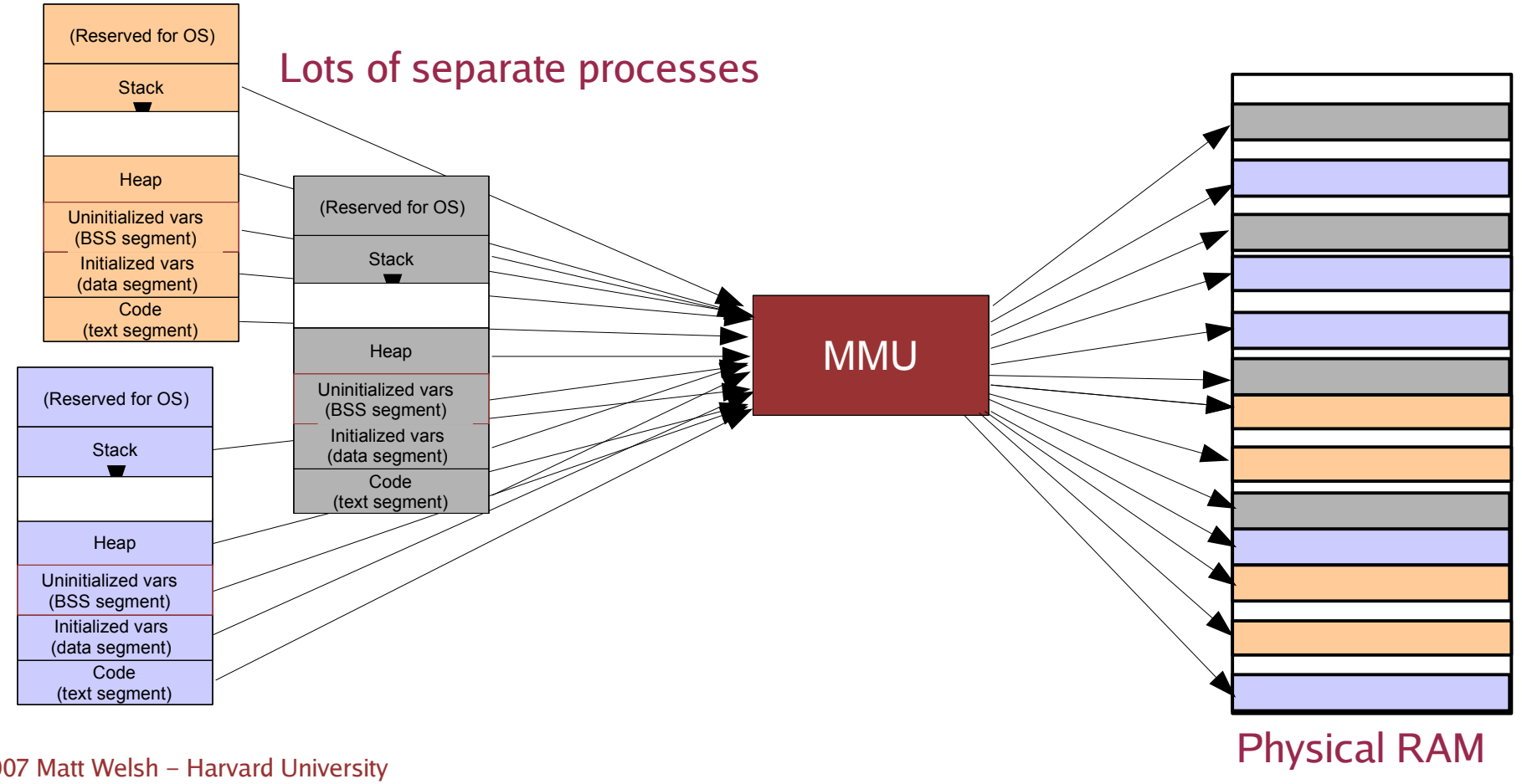
Application Perspective

Application believes it has a single, contiguous address space ranging from 0 to $2^P - 1$ bytes

- Where P is the number of bits in a pointer (e.g., 32 bits)

In reality, virtual pages are scattered across physical memory

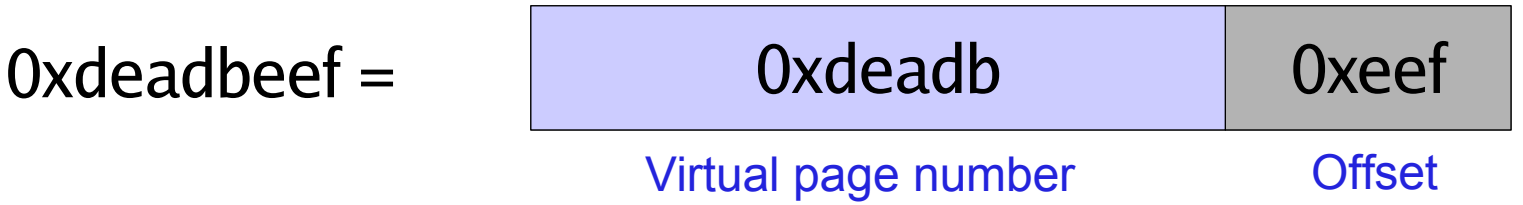
- This mapping is invisible to the program, and not even under it's control!



Virtual Address Translation

Virtual-to-physical address translation performed by MMU

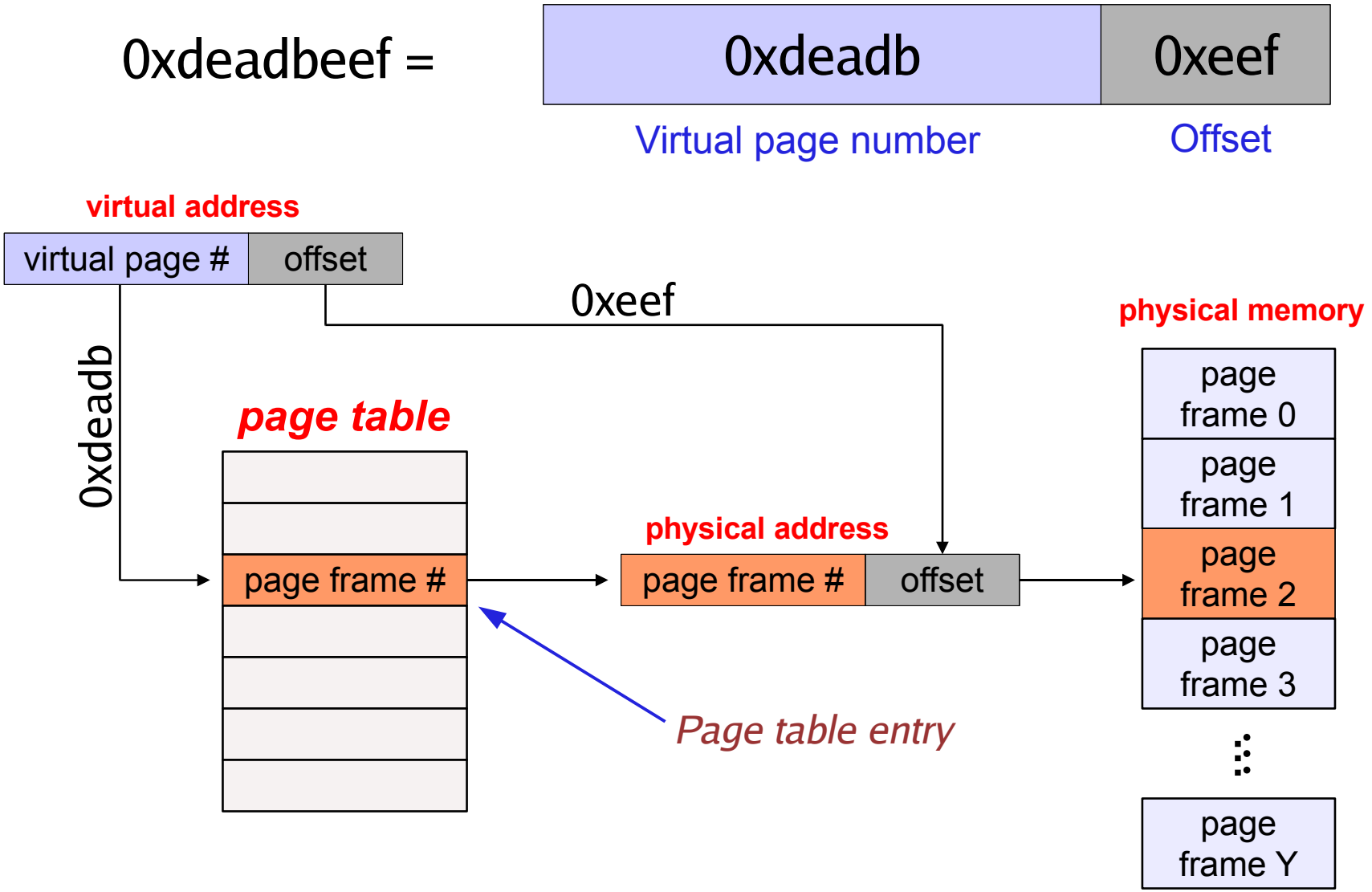
- Virtual address is broken into a *virtual page number* and an *offset*
- Mapping from virtual page to physical frame provided by a *page table*



Virtual Address Translation

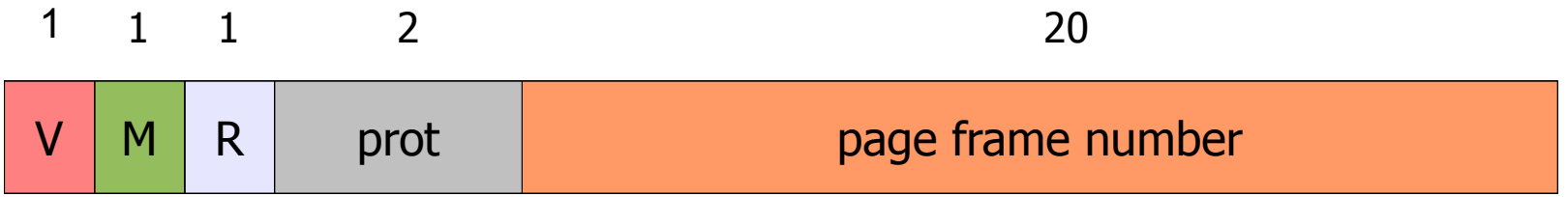
Virtual-to-physical address translation performed by MMU

- Virtual address is broken into a *virtual page number* and an *offset*
- Mapping from virtual page to physical frame provided by a *page table*



Page Table Entries (PTEs)

Typical PTE format (depends on CPU architecture!)

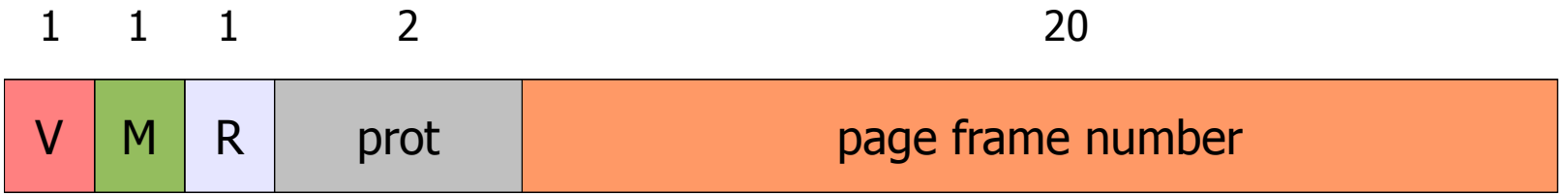


Various bits accessed by MMU on each page access:

- Valid bit (V): Whether the corresponding page is in memory
- Modify bit (M): Indicates whether a page is "*dirty*" (modified)
- Reference bit (R): Indicates whether a page has been accessed (read or written)
- Protection bits: Specify if page is readable, writable, or executable
- Page frame number: Physical location of page in RAM
  - ***Why is this 20 bits wide in the above example???***

Page Table Entries (PTEs)

What are these bits useful for?



The R bit is used to decide which pages have been accessed recently.

- Next lecture we will talk about swapping "old" pages out to disk.
- Need some way to keep track of what counts as an "old" page.
  - *"Valid" bit will not be set for a page that is currently swapped out!*

The M bit is used to tell whether a page has been modified

- Why might this be useful?

Protection bits used to prevent certain pages from being written.

- Why might this be useful?

How are these bits updated?

Advantages of paging

Simplifies physical memory management

- OS maintains a free list of physical page frames
- To allocate a physical page, just remove an entry from this list

No external fragmentation!

- Virtual pages from different processes can be interspersed in physical memory
- No need to allocate pages in a contiguous fashion

Allocation of memory can be performed at a fine granularity

- Only allocate physical memory to those parts of the address space that require it
- Can swap unused pages out to disk when physical memory is running low
- Idle programs won't use up a lot of memory (even if their address space is huge!)

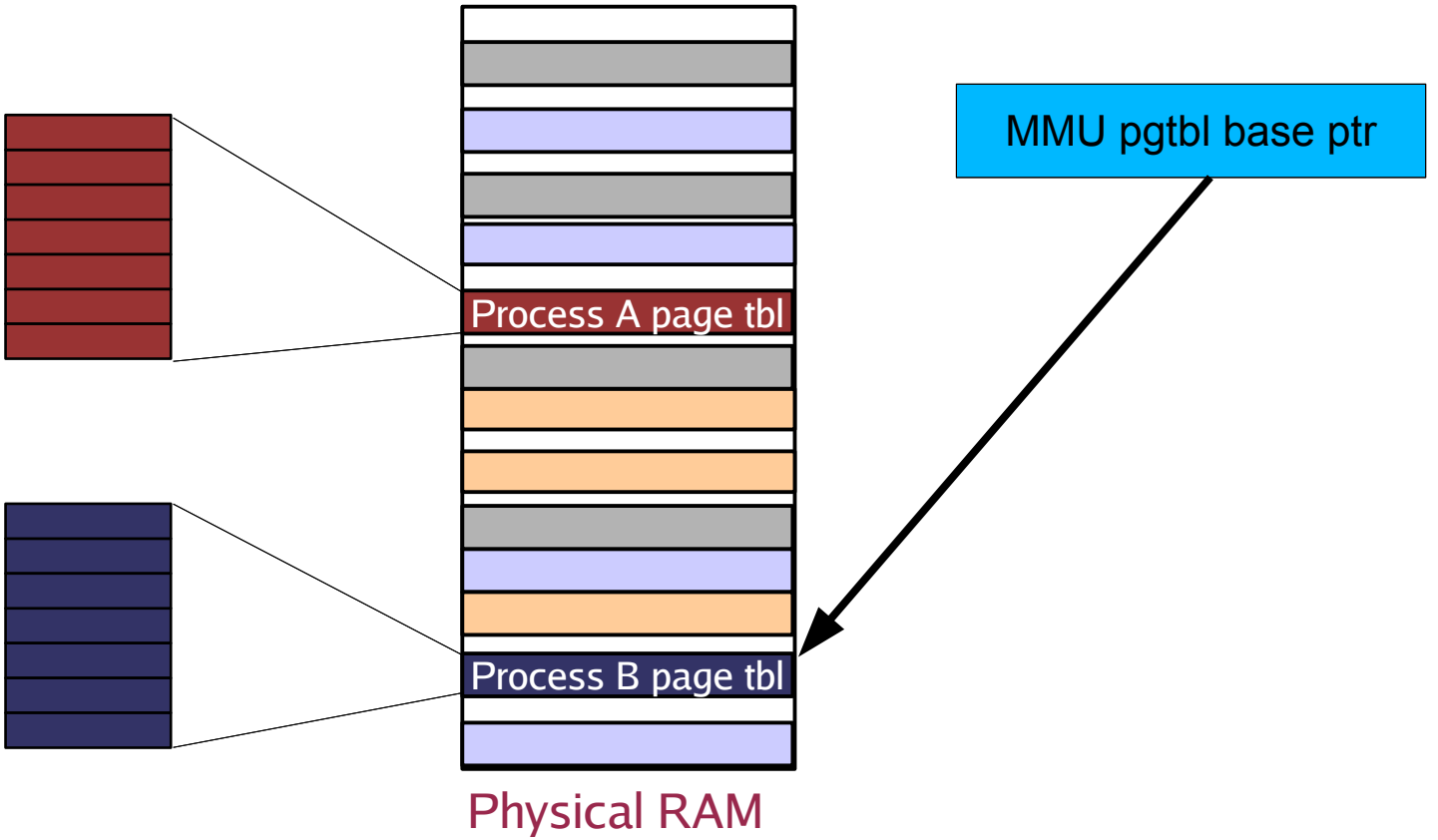
Page Tables

Page Tables store the virtual-to-physical address mappings.

Where are they located? *In memory!*

OK, then. How does the MMU access them?

- The MMU has a special register called the *page table base pointer.*
- This points to the **physical memory address** of the top of the page table for the currently-running process.



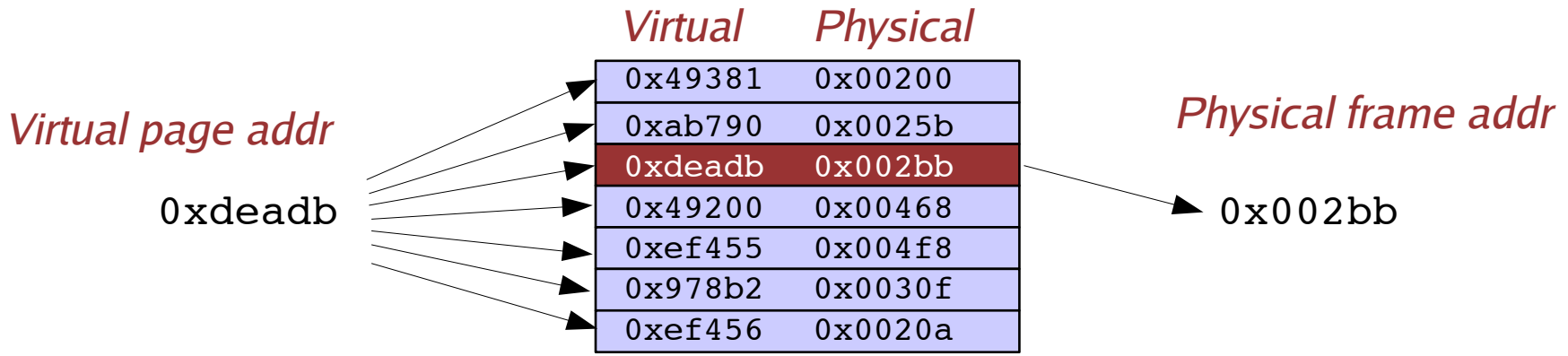
The TLB

Now we've introduced a high overhead for address translation

- On every memory access, must have a *separate* access to consult the page tables!

Solution: *Translation Lookaside Buffer (TLB)*

- Very fast (but small) cache directly on the CPU
  - *Pentium 6 systems have separate data and instruction TLBs, 64 entries each*
- Caches most recent virtual to physical address translations
- Implemented as fully associative cache
  - *Any address can be stored in any entry in the cache*
  - *All entries searched "in parallel" on every address translation*
- A *TLB miss* requires that the MMU actually try to do the address translation



Loading the TLB

Two ways to load entries into the TLB.

1) MMU does it automatically

- MMU looks in TLB for an entry
- If not there, MMU handles the TLB miss directly
- MMU looks up virt->phys mapping in page tables and loads new entry into TLB

2) Software-managed TLB

- TLB miss causes a trap to the OS
- OS looks up page table entry and loads new TLB entry

Why might a software-managed TLB be a good thing?

Loading the TLB

Two ways to load entries into the TLB.

1) MMU does it automatically

- MMU looks in TLB for an entry
- If not there, MMU handles the TLB miss directly
- MMU looks up virt->phys mapping in page tables and loads new entry into TLB

2) Software-managed TLB

- TLB miss causes a trap to the OS
- OS looks up page table entry and loads new TLB entry

Why might a software-managed TLB be a good thing?

- OS can dictate the page table format!
  - *MMU does not directly consult or modify page tables.*
- Gives a lot of flexibility for OS designers to decide memory management policies

Page Table Size

How big are the page tables for a process?

Well ... we need one PTE per page.

Say we have a 32-bit address space, and the page size is 4KB

How many pages?

Page Table Size

How big are the page tables for a process?

Well ... we need one PTE per page.

Say we have a 32-bit address space, and the page size is 4KB

How many pages?

- 2^32 == 4GB / 4KB per page == 1,048,576 (1 M pages)

How big is each PTE?

- Depends on the CPU architecture ... on the x86, it's 4 bytes.

So, the total page table size is: 1 M pages * 4 bytes/PTE == 4 Mbytes

- And that is *per process*
- If we have 100 running processes, that's over 400 Mbytes of memory just for the page tables.

Solution: Swap the page tables out to disk!

- My brain hurts ... more on this next lecture

Application Perspective

Remember our three requirements for memory management:

Isolation

- One process cannot access another's pages. Why?
  - *Process can only refer to its own virtual addresses.*
  - *O/S responsible for ensuring that each process uses disjoint **physical** pages*

Fast Translation

- Translation from virtual to physical is fast. Why?
  - *MMU (on the CPU) translates each virtual address to a physical address.*
  - *TLB caches recent virtual->physical translations*

Fast Context Switching

- Context Switching is fast. Why?
  - *Only need to swap pointer to current page tables when context switching!*
  - *(Though there is one more step ... what is it?)*

Next Lecture

More on paging mechanisms, multi-level page tables, and TLBs