LECTURE 2.2
PAGING

COP4600

Dr. Matthew Gerber

3/2/2016

PAGING (8.5)

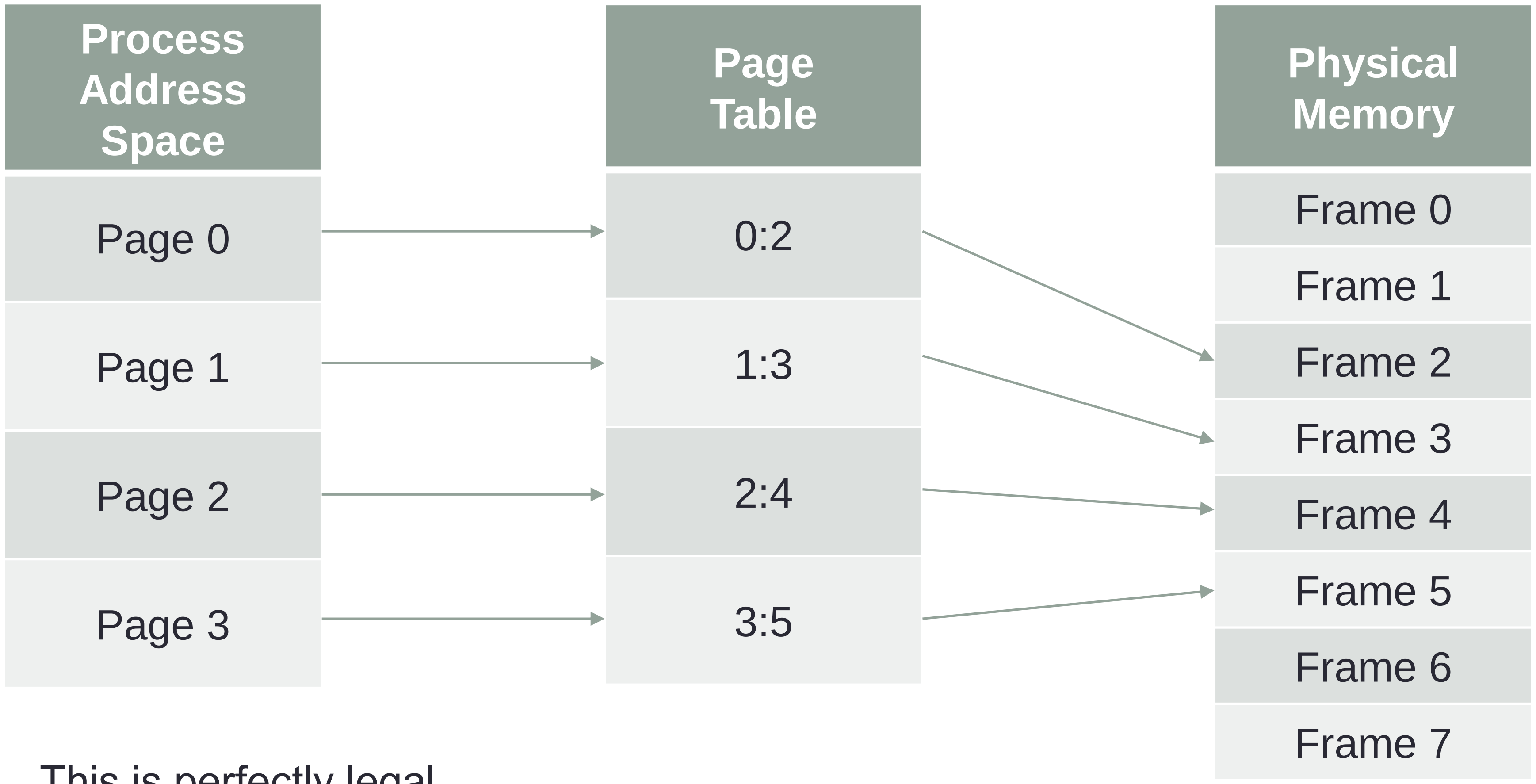
Paging (8.5.1)

- Take segmentation one step farther
- Break all of physical memory into fixed-sized blocks called *frames*
- Break all of logical memory into blocks of the same size, called *pages*
- When a process is to be executed, its *pages* are loaded into *frames*

Paged Addressing (8.5.1)

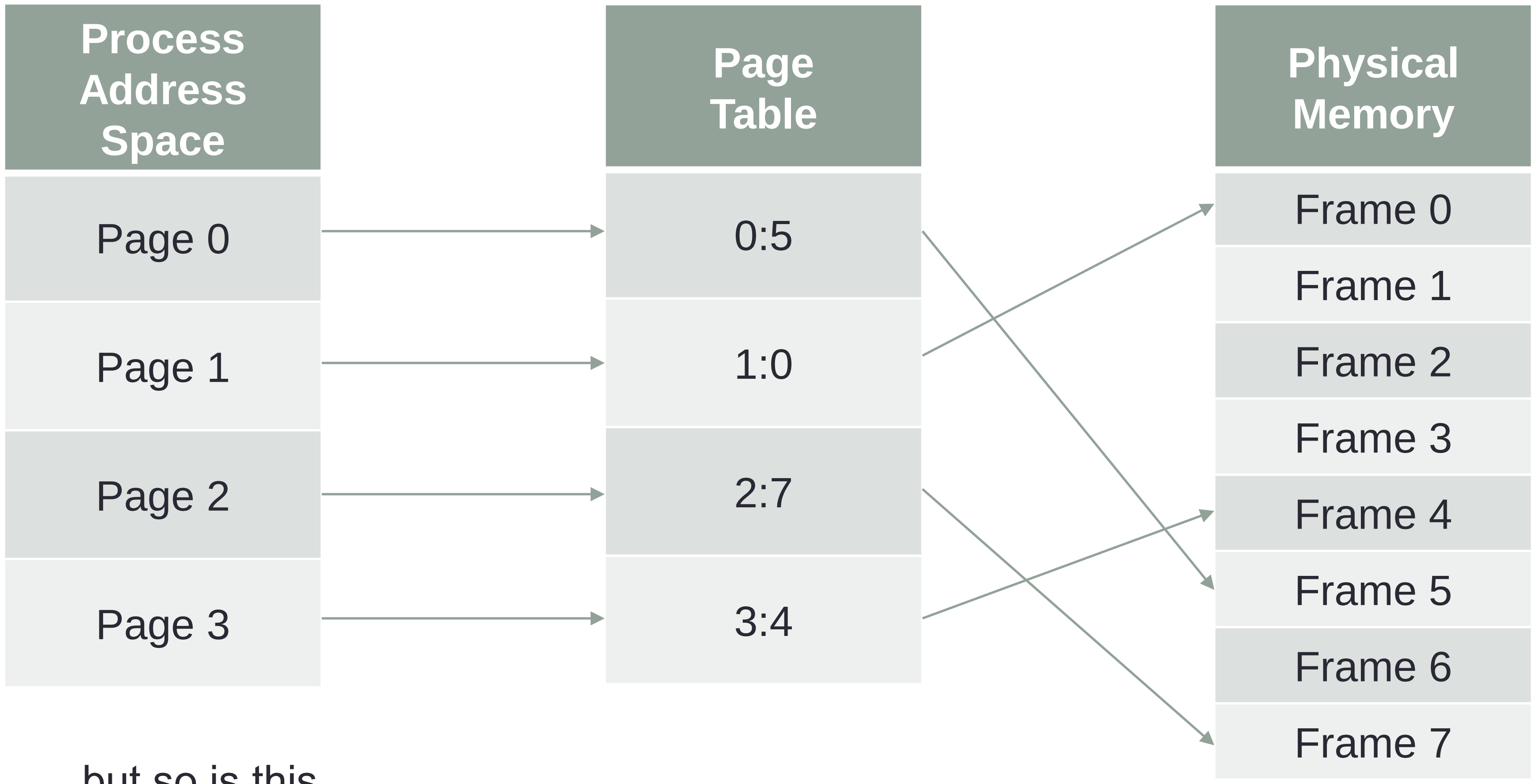
- Addresses again consist of two components: a *page number* and an *offset*
- The size of a page is invariably a power of 2, so that the address can be a single integer and we can extract the page number and offset using bitwise functions
- On every address access, translation is performed from the logical *page number* to the physical *frame*, and the offset is added to the result
- The translation is done by means of a (typically per-process) *page table*

The Page Table (8.5.1)



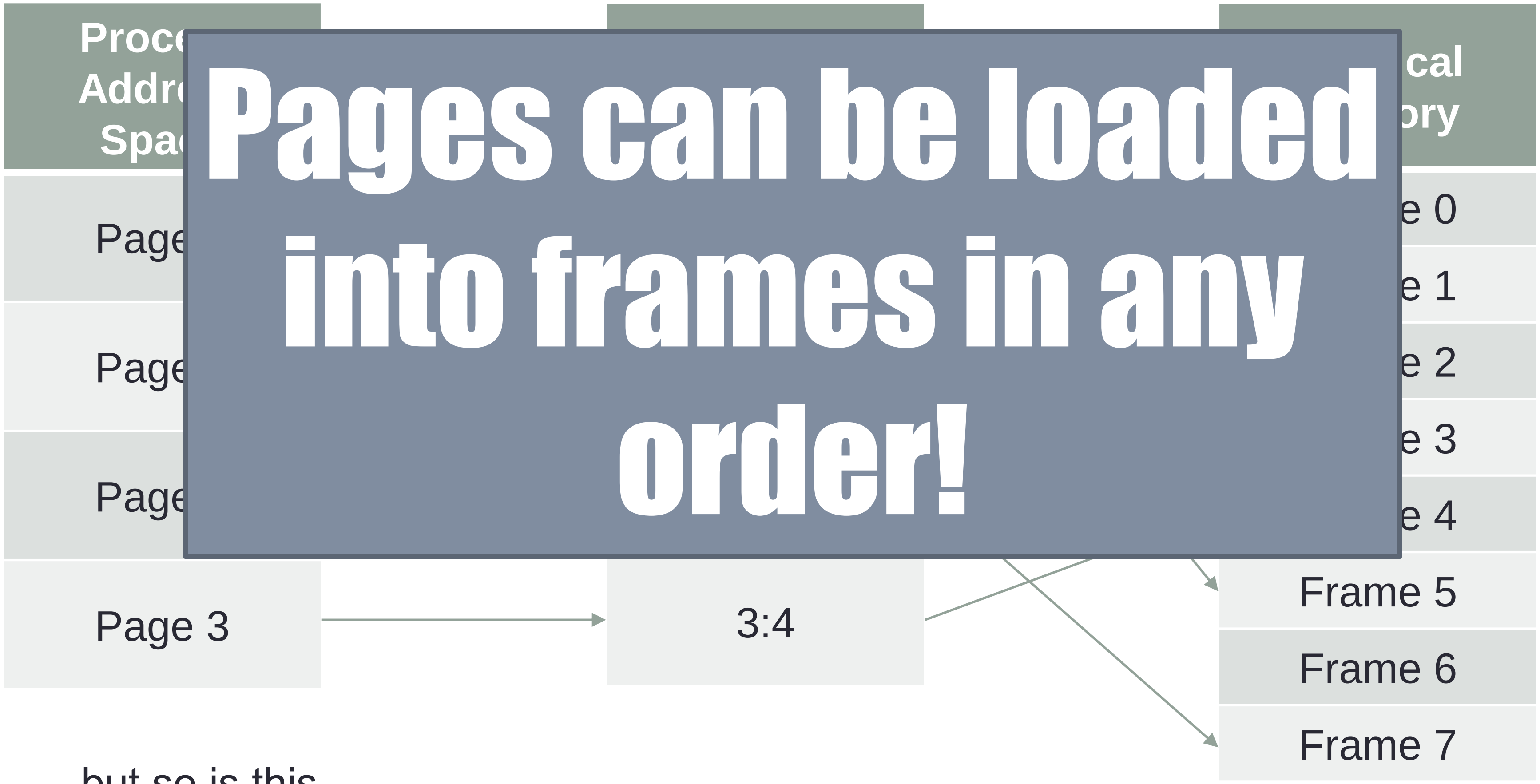
This is perfectly legal…

The Page Table (8.5.1)



…but so is this

The Page Table (8.5.1)



Pages can be loaded into frames in any order!

…but so is this

Addressing Again (8.5.1)

- This is dynamic relocation taken to its logical conclusion
- In the second example, assume our page size was 512 bytes
- Logical address 511 would map to the last byte of page 0, and be in frame 5
- Logical address 512 would map to the first byte of page 1, and be in frame 0
- *The program never has to care!*

Fragmentation and Page Size (8.5.1)

- Paging completely eliminates external fragmentation and the entire memory hole problem
- Paging does induce *internal* fragmentation
- We can easily observe that every process will waste an average of half the page size
- That’s a good reason to have a small page size…
- …but the management isn’t free
- The good news is, since user-level programs simply don’t care, we can let page sizes slowly grow over time without breaking anything

The Frame Table (8.5.1)

- As well as keeping track of pages per process, we also need to keep track of them system wide
- For every process, we have a page table
- For the system, we have a *frame table*
- The frame table has an entry for every physical frame that indicates:
  - Whether it is allocated, and if so
  - To which process, and
  - To which page of that process

Paging: Hardware Support (8.5.2)

- Even more so than simple base and limit registers, the hardware *must* support this
- Since we (usually) use a page table per process, context switch time is increased
- The page table(s) also have to be kept somewhere
- They’re large enough that they need to be kept in main memory, but that can theoretically double memory access time
- Guess what happens next?

The Translation Look-Aside Buffer (8.5.2)

- A *translation look-aside buffer (TLB)* is a cache for page table entries
- With a TLB available, we only have to go to main memory to get a frame number when the page table entry isn't already in the TLB – this is a *TLB miss*
- TLBs may contain page table entries for only one process or may be able to contain entries for an arbitrary number of processes
- As with any cache, the TLB has a *hit ratio*

Paging: Protection (8.5.3)

- As well as having frame numbers, page table entries can simply be marked as not allocated
- If a process tries to access a non-allocated page, it incurs a protection violation
- A process's page table obviously does not always need to map the entire address space; if it doesn't, access to space that isn't in the page table can be presumed to be non-allocated
- We can also define specific pages as read-only, read-write, or even execute-only

Paging: Page Sharing (8.5.4)

- If more than one process uses identical pages for code (as opposed to data), and those code pages *never modify themselves*, then there’s no reason those pages need to be unique for each process
- Useful for programs that need to be run more than once on servers
- Also useful for shared libraries
- The read-only property of shared code *must* be enforced by the OS

Hierarchical Page Tables (8.6.1)

- 32-bit addresses allow for a pretty big address space: 4,294,967,296 bytes
- With small pages, if we use a nontrivial amount of the space, the page table itself can reach into the megabytes
- One solution to this is to page the page table
- By the time we get to 64-bit addresses and the resulting 18,446,744,073,709,551,616-byte address space, that isn’t enough to make the size manageable…
- …but as it turns out, we aren’t *actually* using 64-bit addresses now

Case Study: IA32 (8.7.1)

- Intel's legacy IA-32 architecture uses a two-level paging table for 4KB pages
- (It also still allows for old-school segmented addresses)
- IA-32 also supports a *page address extension* scheme that effectively expands paging by process
  - *Processes* are still limited to 4GB
  - The *system* can address up to 64GB
  - This *is* actually supported on some operating systems

Case Study: x86-64 (8.7.2)

- Intel's (really, AMD's) x86-64 architecture provides a four-level paging table, with page sizes of 4KB, 2MB, and 1GB
- Only 48 bits are actually used
- That's enough for 262,144GB, so it's still probably going to be pretty safe for a while
  - PAE is also supported if we're really worried
  - That brings the total system-addressable memory to 4,096TB

What About Swapping Pages?

- Remember that we advanced the idea of swapping out processes
- This isn’t actually practical in most cases because of the sheer overhead involved in swapping them out or bringing them back in
- But what if we could swap out only certain *pages* of a process?
  - Initialization or rarely-used code fragments
  - Historical data that may not need to be referenced for a long time
  - And so on…

NEXT TIME: VIRTUAL MEMORY