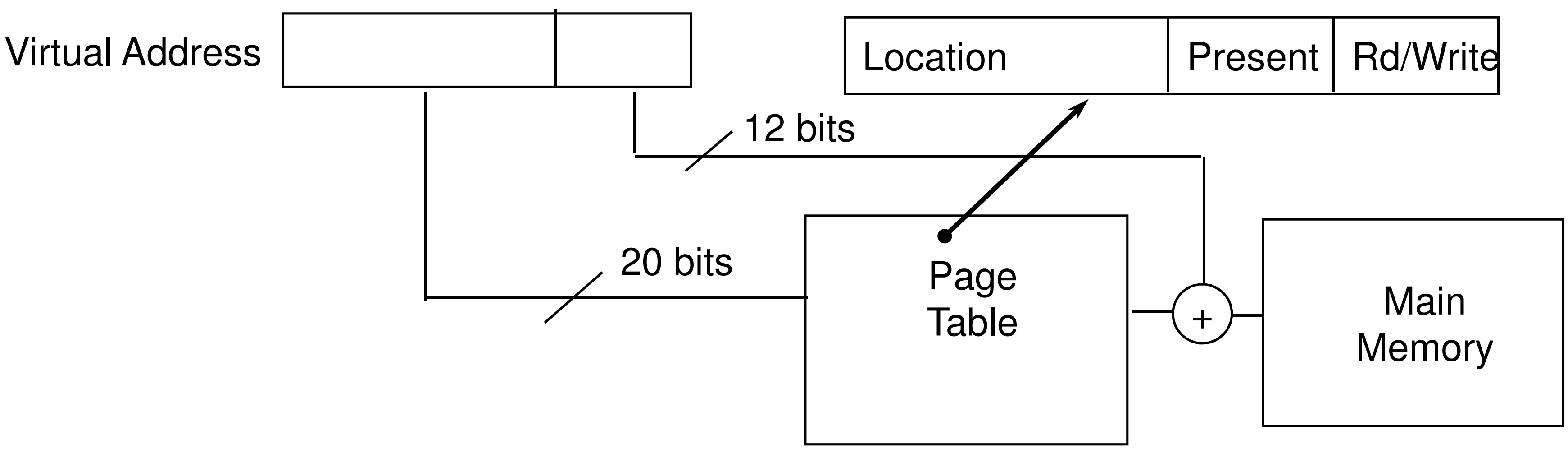
Announcements

- Reading
  - Last time : Chapter 8.1-8.5 (8th Ed)
  - Today: 8.6-8.8, 9.1-9.4
- Midterm #1
  - Thursday
- Project #3
  - Is on the web

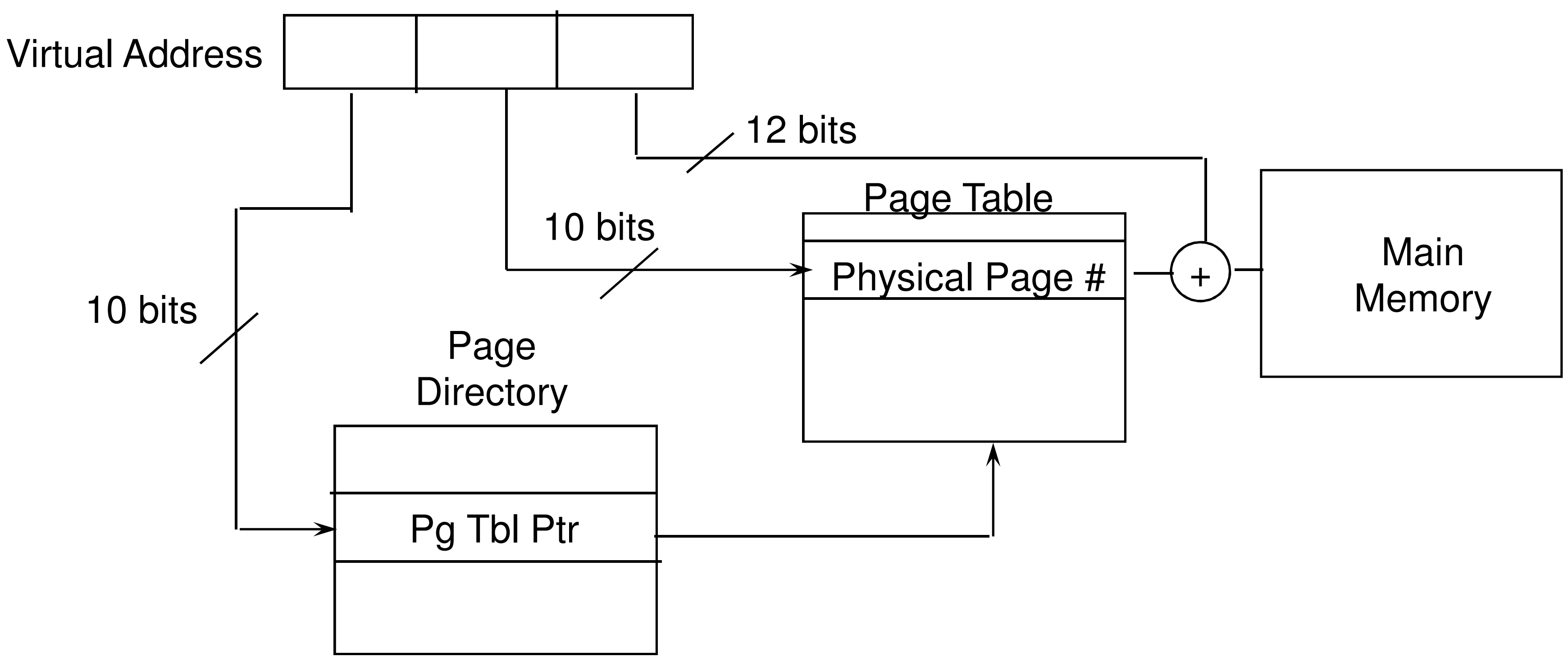
Paging

- Divide physical memory into fixed sized chunks called *pages*
  - typical pages are 512 bytes to a few megabytes
  - When a process is to be executed, load the pages that *are actually used* into memory
- Have a table to map virtual pages to physical pages
- Consider a 32 bit addresses
  - 4096 byte pages (12 bits for the page)
  - 20 bits for the page number



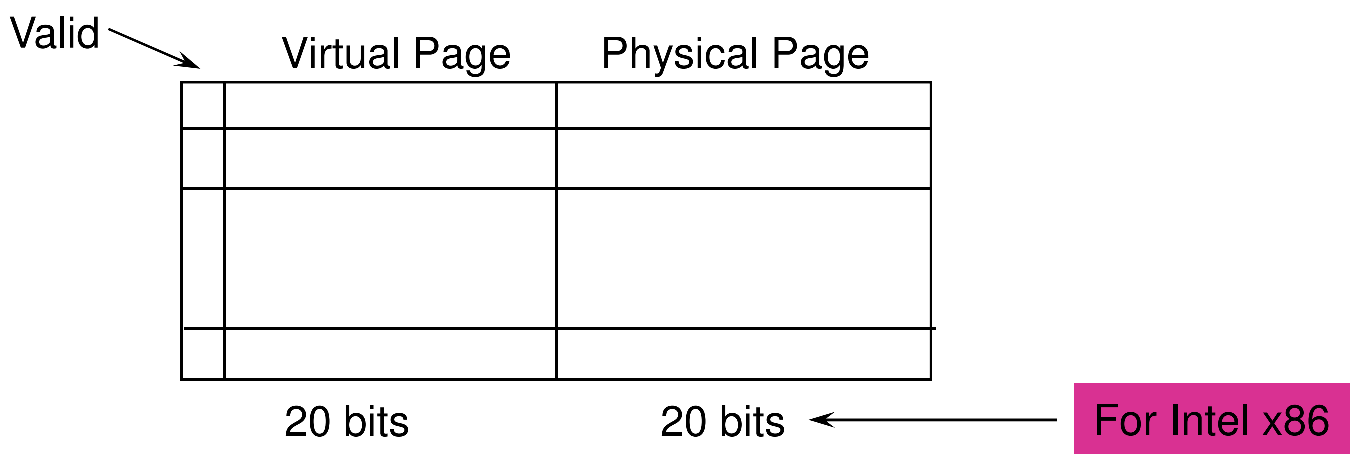
Problems with Page Tables

- One page table can get very big
  - 2^{20} entries (for most programs, most items are empty)
- solution1: use a hierarchy of page tables



Faster Mapping from Virtual to Physical Addresses

- need hardware to map between physical and virtual addresses
  - can require multiple memory references
  - this can be slow
- answer: build a cache of these mappings
  - called a translation look-aside buffer (TLB)
  - associative table of virtual to physical mappings
  - typically 16- 64 entries



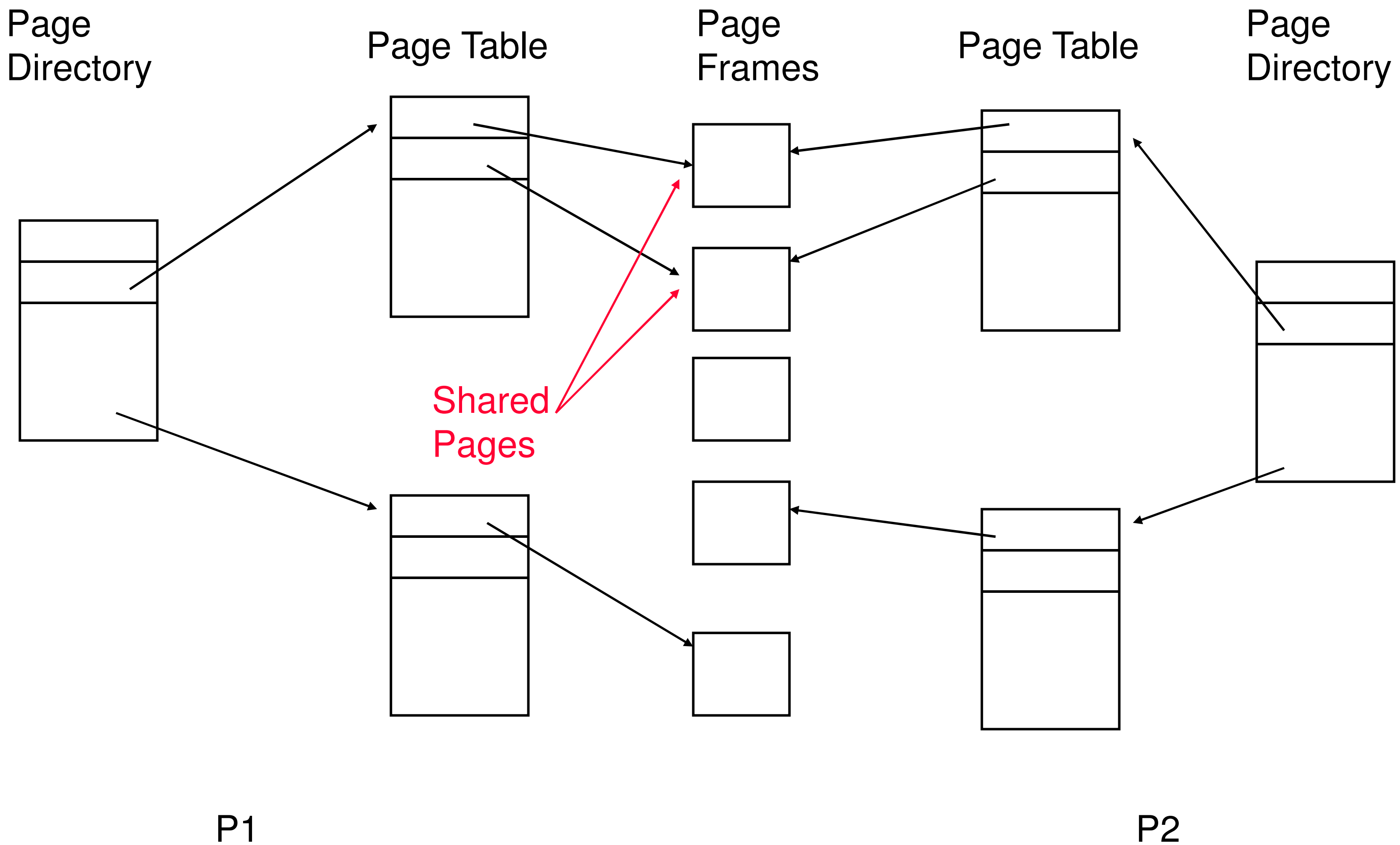
Super Pages

- TLB Entries
  - Tend to be limited in number
  - Can only refer to one page
- Idea
  - Create bigger pages
  - 4MB instead of 4KB
  - One TLB entry covers more memory

Sharing Memory

- Pages can be shared
  - several processes may share the same code or data
  - several pages can be associated with the same page frame
  - given read-only data, sharing is always safe
- when writes occur, decide if processes share data
  - operating systems often implement “copy on write” - pages are shared until a process carries out a write
    - when a shared page is written, a new page frame is allocated
    - writing process owns the modified page
    - all other sharing processes own the original page
  - page could be shared
    - processes use semaphores or other means to coordinate access

Page Sharing



What Happens when a virtual address has no physical address?

- called a *page fault*
  - a trap into the operating system from the hardware
- caused by: the first use of a page
  - called *demand paging*
  - the operating system allocates a physical page and the process continues
  - read code from disk or init data page to zero
- caused by: a reference to an address that is not valid
  - program is terminated with a "segmentation violation"
- caused by: a page that is currently on disk
  - read page from disk and load it into a physical page, and continue the program
- causde by: a copy on write page

OS Protection attributes (Win32)

- NOACCESS: attempts to read, write or execute will cause an access violation
- READONLY: attempts to write or execute memory in this region cause an access violation
- READWRITE: attempts to execute memory in this region cause an access violation
- EXECUTE: Attempts to read or write memory in this region cause an access violation
- EXECUTE_READ: Attempts to write to memory in this region cause an access violation
- EXECUTE_READ_WRITE: Do anything to this page
- WRITE_COPY: Attempts to write will cause the system to give a process its own copy of the page. Attempts to execute cause access violation
- EXECUTE_WRITE_COPY: Attempts to write will cause the system to give a process its own copy of a page. Can't cause an access violation

Handling a page fault

1) Check if the reference is valid
   - if not, terminate the process
2) Find a page frame to allocate for the new process
   - for now we assume there is a free page frame.
3) Schedule a read operation to load the page from disk
   - we can run other processes while waiting for this to complete
4) Modify the page table entry to the page
5) Restart the faulting instruction
   - hardware normally will abort the instruction so we just return from the trap to the correct location.

Page Fault – Page is Paged out