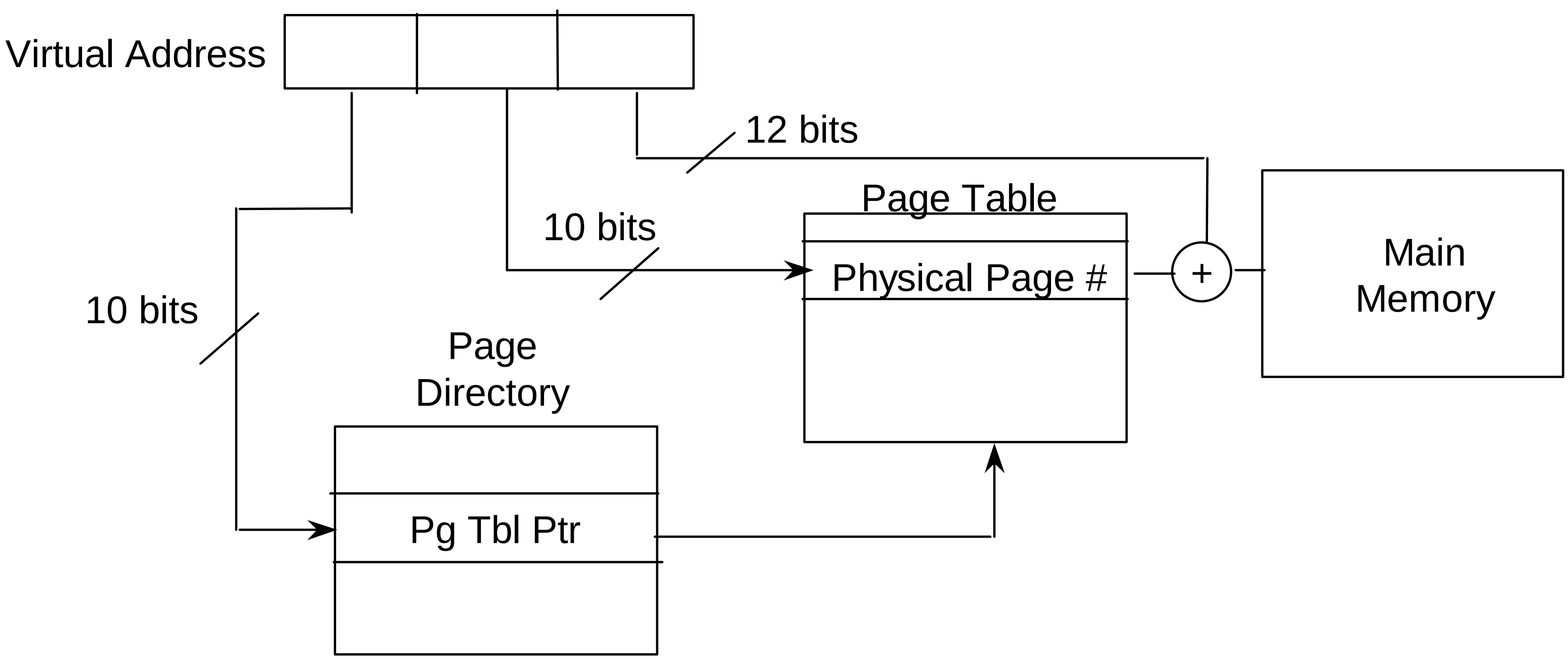
Announcements

- Reading 8.7, 9.1-9.4
- Suggested problems
  - 8.10, 8.12, 8.17
- Midterm #1 is on Tuesday

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



Inverted Page Tables

- Solution to the page table size problem
- One entry per page frame of physical memory

    <process-id, page-number>

  – each entry lists process associated with the page and the page number

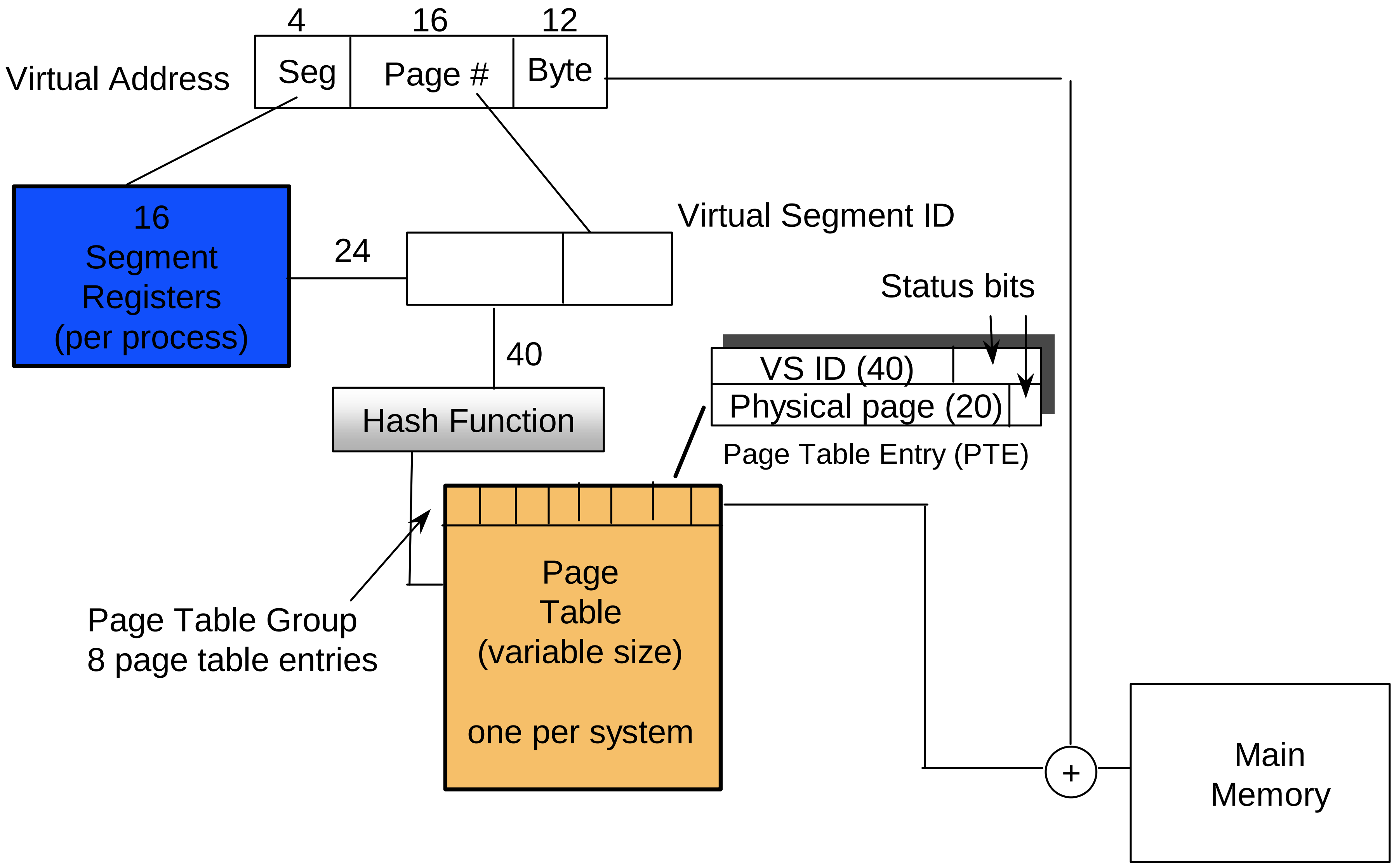
  – when a memory reference:

    - **<process-id,page-number,offset>**occurs, the inverted page table is searched (usually with the help of a hashing mechanism)
    - if a match is found in entry ***i*** in the inverted page table, the physical address **<i,offset>** is generated

  – The inverted page table does not store information about pages that are not in memory

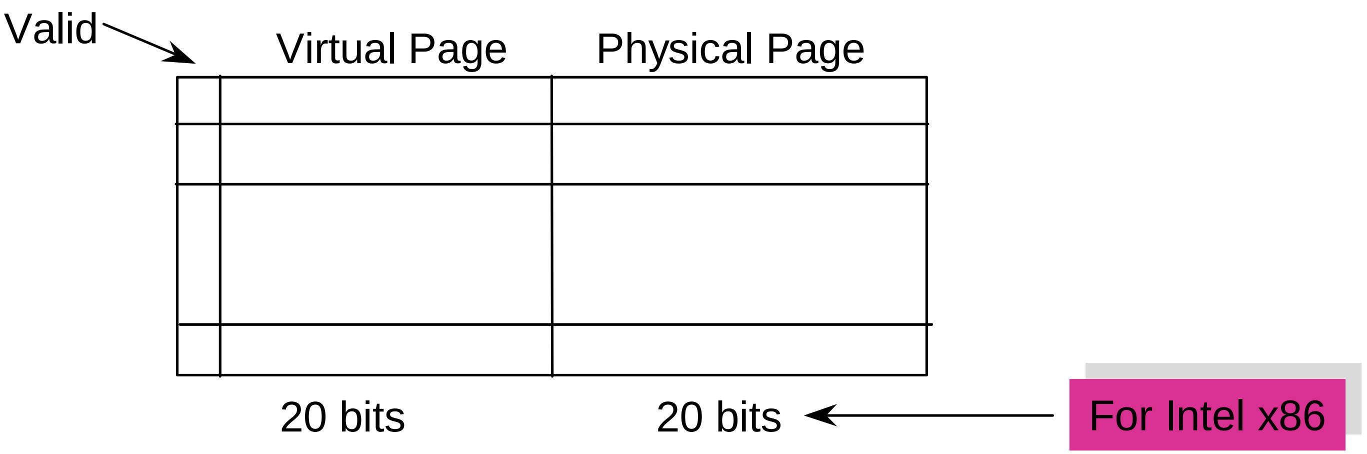
    - page tables are used to maintain this information
    - page table need only be consulted when a page is brought in from disk

Inverted Page Table Example (PPC)



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



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

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

Page State (hardware view)

- Page frame number (location in memory or on disk)
- *Valid Bit*
  - indicates if a page is present in memory or stored on disk
- A *modify* or *dirty* bit
  - set by hardware on write to a page
  - indicates whether the contents of a page have been modified since the page was last loaded into main memory
  - if a page has not been modified, the page does not have to be written to disk before the page frame can be reused
- *Reference bit*
  - set by the hardware on read/write
  - cleared by OS
  - can be used to approximate LRU page replacement
- Protection attributes
  - read, write, execute

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.

What happens when we fault and there are no more physical pages?

- Need to remove a page from main memory
  - if it is “dirty” we must store it to disk first.
    - dirty pages have been modified since they were last stored on disk.
- How to we pick a page?
  - Need to choose an appropriate algorithm
    - should it be global?
    - should it be local (one owned by the faulting process)

Page Replacement Algorithms

- FIFO
  - Replace the page that was brought in longest ago
  - However
    - old pages may be great pages (frequently used)
    - number of page faults may increase when one increases number of page frames (discouraging!)
      - called belady's anomaly
      - 1,2,3,4,1,2,5,1,2,3,4,5 (consider 3 vs. 4 frames)
- Optimal
  - Replace the page that will be used furthest in the future
  - Good algorithm(!) but requires knowledge of the future
  - With good compiler assistance, knowledge of the future is sometimes possible