Sharing Main Memory: Segmentation (cont), Free Space Management, and Paging

ECE469, Feb 23

Yiying Zhang

Reading assignment

- Dinosaur Chapter 8
- Comet Chapters 16, 17, 18

Free Space Management

Dynamic memory allocation: two general ways

- Stack
  - Restricted
  - Simple and efficient
  - Easy to implement

- Heap
  - More general
  - Less efficient
  - More difficult to implement

Heap organization

- Allocation & freeing are unpredictable
  - For arbitrary, complex data structures
    - Example: payroll system
      - Don't know when employee will join and leave the company
      - Must keep track of all of them

- Memory consists of allocated areas and free areas (holes) → lots of holes inevitable

- Fragmentation problem
  - solution: keep # of holes small, size large

Heap organization

- *Fragmentation*: inefficient use of memory due to holes too small
  - What happens in stack?
- Typically, heap allocation uses a *free list* of holes
- Allocation algorithms differ in how to manage the free list

Implementation

- Bit map
  - For fixed-size chunks (e.g., disk blocks)

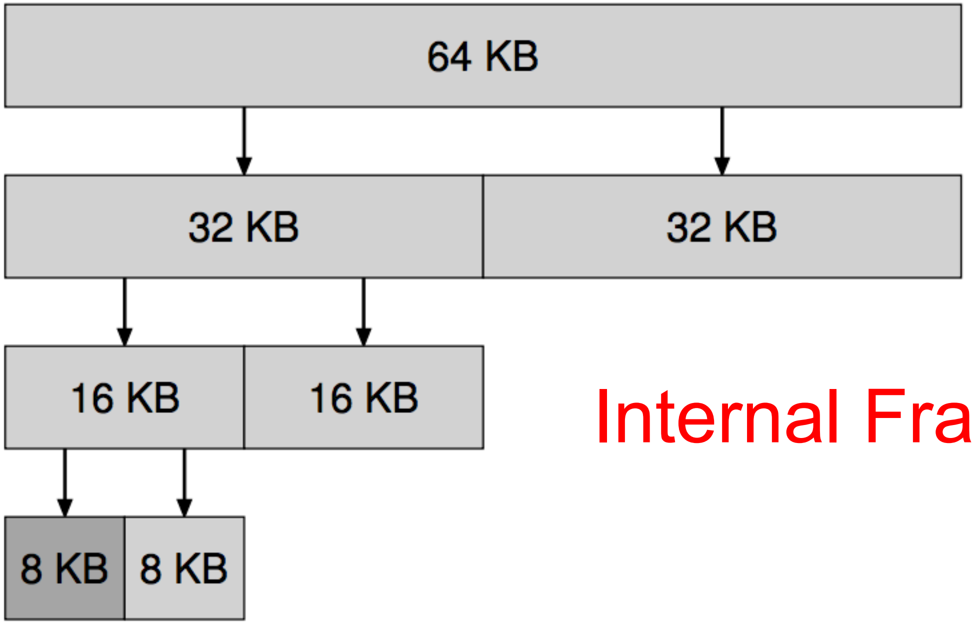
- Pools
  - A separate allocation pool for each popular size
  - Fast, no fragmentation
  - But some pools may run out faster than others

Implementation – Segregated Lists

- Basic motivation: applications may use certain types of objects often
  - These types have fixed sizes
- Always keep a few free objects (memory regions) of popular sizes
  - One list of free objects for one size
- Example: Linux kernel slab allocator

Implementation - Buddy Allocation

- Basic idea: coalescing free regions
  - Free space organized in “binary”
  - Only give out power-of-two-sized blocks
  - Neighboring free spaces form a bigger one



Internal Fragmentation!

Reclamation

- When can dynamically-allocated memory be freed?
  - Easy if a chunk is used in one place
  - Hard when a chunk is shared
  - Sharing is indicated by presence of pointers to the data

Reclamation techniques

- Reference counts:
  - Keep track of the number of outstanding pointers to each chunk of memory
  - When this goes to 0, free the memory
  - Example:
    - Memory region
    - File descriptors in UNIX
  - Works fine with hierarchical structures
  - What about circular structures?

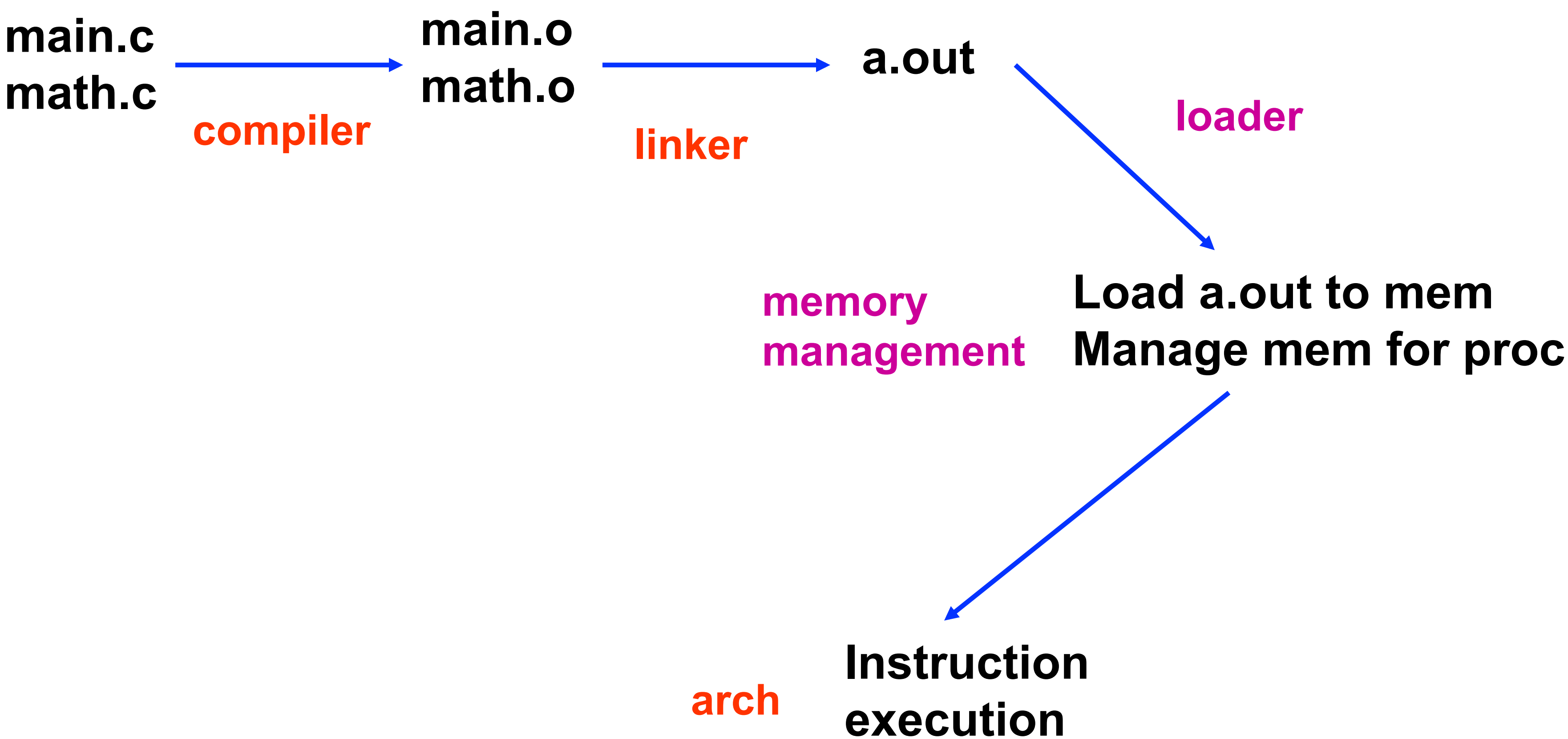
Reclamation techniques

- Garbage collection
  - Storage isn't freed explicitly (using free), rather implicitly, i.e., by deleting pointers
  - When the system needs storage, it scans through all pointers (all of them!!!) and collects things not used
  - For circular structures, this is the only way
  - Makes life easier on programmers, but GCs are hard to implement

Reclamation techniques

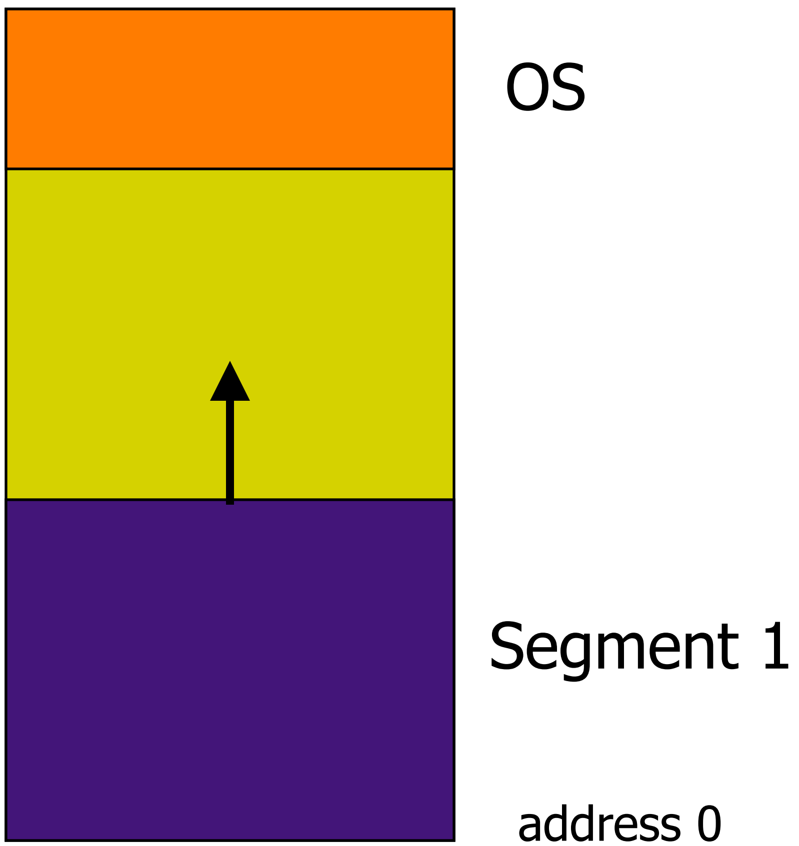
- Garbage collection – implementation
  - Must be able to find all objects
  - Must be able to find all pointers to objects
  - Pass1: mark
    - Go through all statically-allocated and procedure local variables, looking for pointers
    - Mark each obj pointed to, and recurs
    - Compiler has to help by saving info about pointers with structures
  - Pass 2: sweep
    - Go through all objs, free up those that aren't marked

[lec13] The big picture



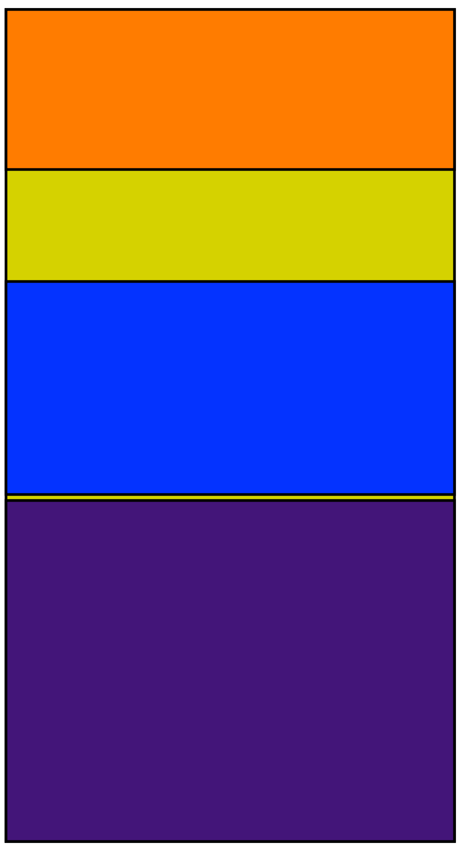
Review: managing memory

- Simple uniprogramming



Review: managing memory

- Simple multiprogramming



4 drawbacks?

Review: managing memory

- Simple multiprogramming – 4 drawbacks

1. No protection
2. Low utilization -- Cannot relocate dynamically
   - Cannot do anything about holes
3. No sharing -- Single segment per process
4. Entire address space needs to fit in mem
   - Need to swap whole, very expensive!

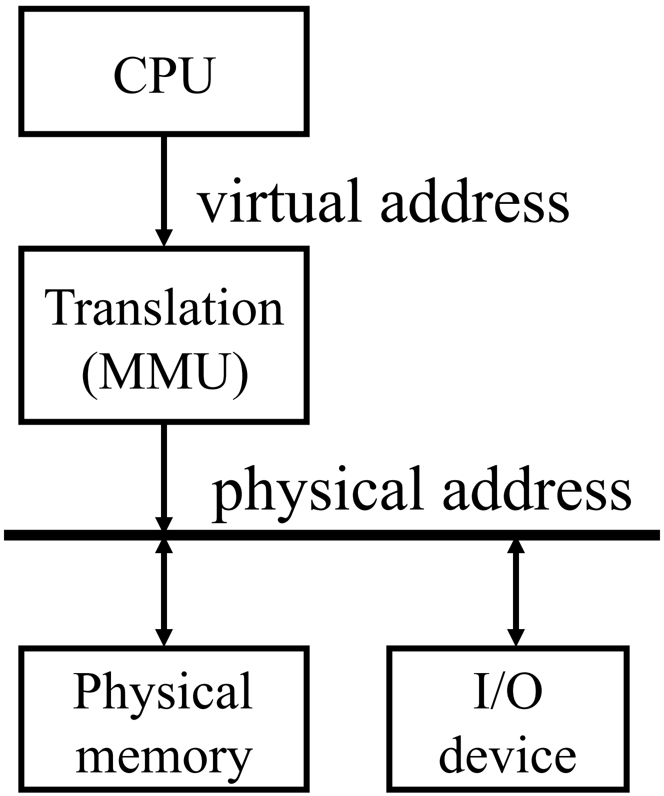
Review: Solution → relocation

- Because several processes share memory, cannot predict where a process will be loaded in memory
  - What's the analogy in compiler's job?
- Relocation adjusts a program to run in a different area of physical memory
  - Linker is an example of "static relocation" used to combine modules into programs

Review: Fix drawback #1 & #2: Dynamic memory relocation

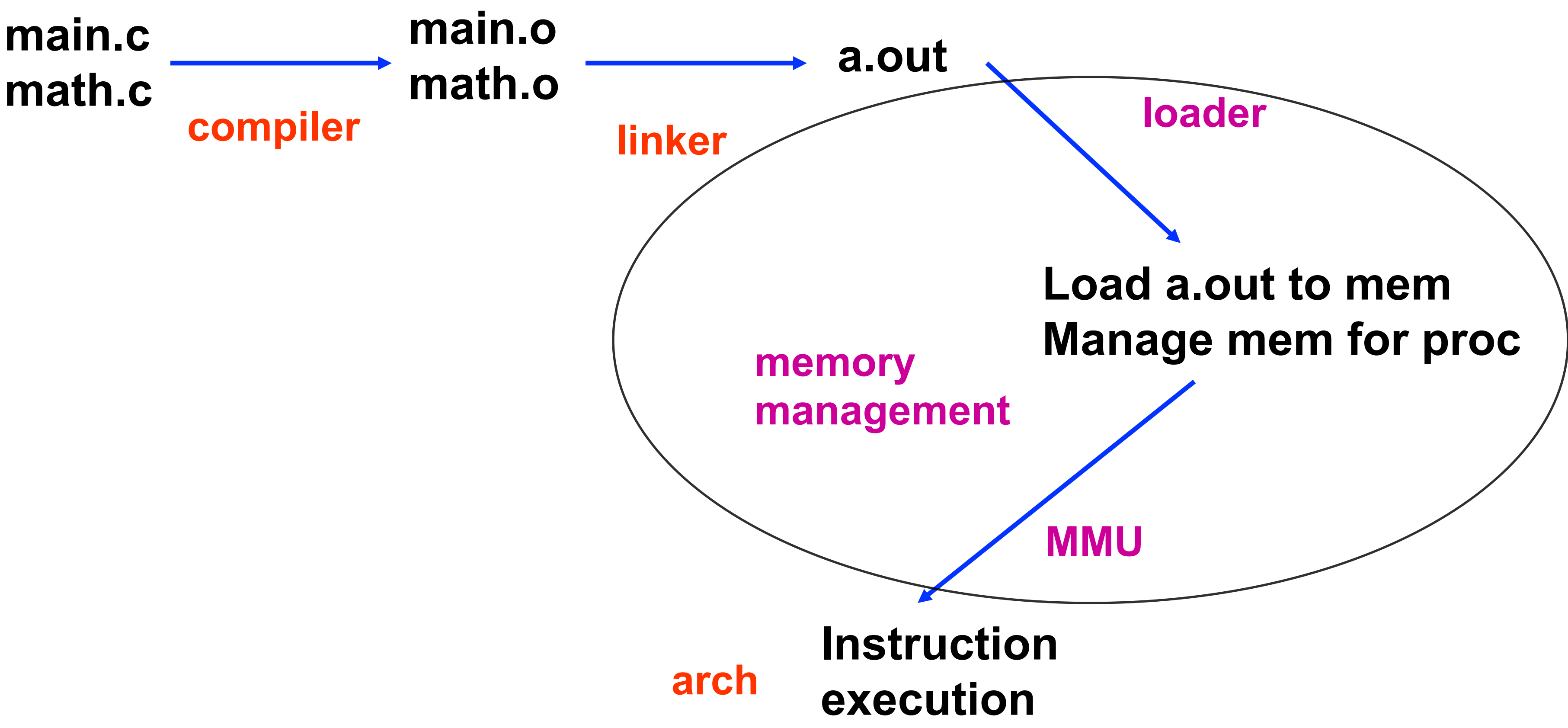
- Instead of changing the address of a program when it' s loaded, change the address dynamically *during every reference*
  - Under dynamic relocation, each program-generated address (called a *logical address* or *virtual address*) is translated in hardware to a *physical* or *real address*

Review: Dynamic memory relocation

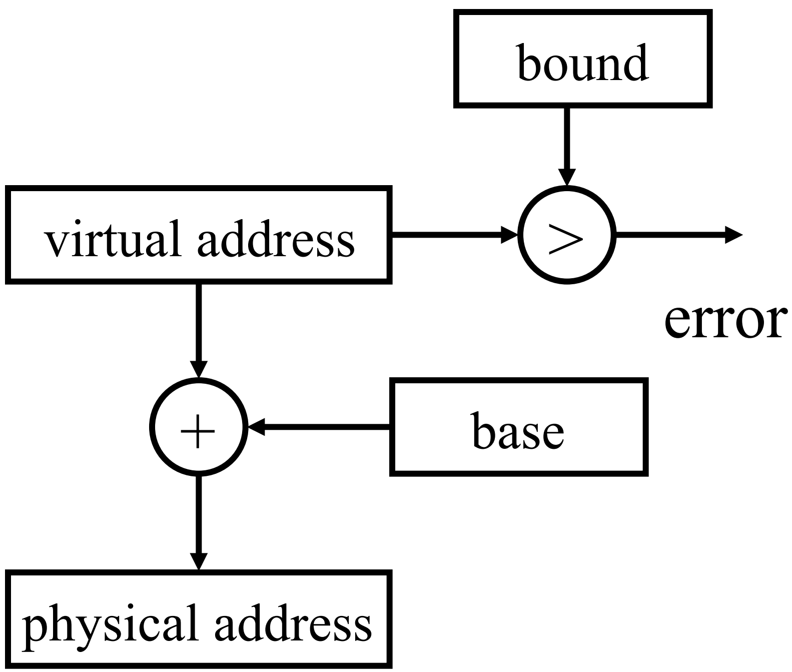


- Actual translation is in hardware (MMU)
  - why?
- Controlled in software
- CPU view
  - what program sees, virtual addresses
- Memory view
  - physical memory

[lec13] The big picture



Review: Fix drawback # 1 & #2 – base and bound



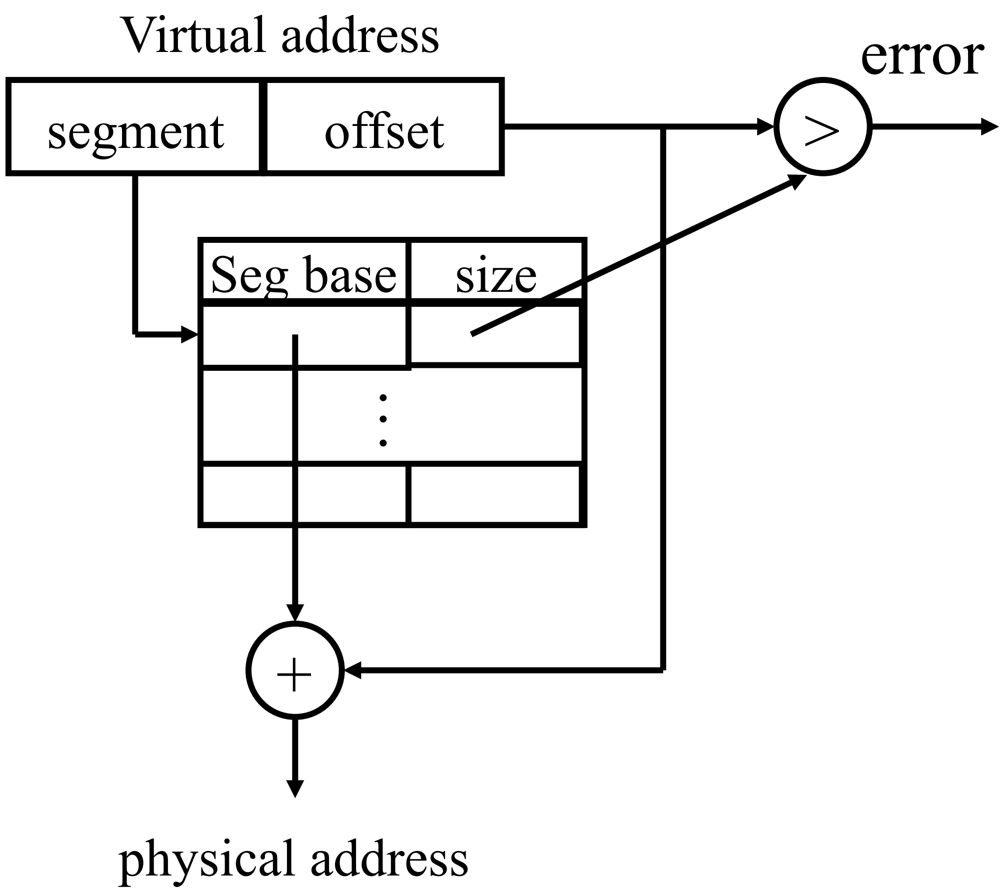
- The essence:
  - A level of indirection
  - Phy. Addr = Vir. Addr + base

Review: sharing main memory

- Simple multiprogramming – 4 drawbacks
  - No protection
  - Low utilization -- Cannot relocate dynamically
    - *Can relocate now; but is frequent relocation desirable?*

  - No sharing -- Single segment per process
  - Entire address space needs to fit in mem
    - Need to swap whole, very expensive!

Review: Fix drawback # 3 – Multiple Segments



- Have a table of (seg, size)
- Further protection: each entry has (nil, read, write, exec)
- On a context switch: save/restore the table (or a pointer to the table) in kernel memory

Indirection!

All problems in computer science can be solved by another level of indirection

Butler Lampson

All problems in computer science can be solved by another level of indirection

Butler Lampson

David Wheeler

but that usually will create another problem

– David Wheeler

Summary: Evolution of Memory Management (so far)

Scheme	How	Pros	Cons
Simple uniprogramming	1 segment loaded to starting address 0	Simple	1 process 1 segment No protection
Simple multiprogramming	1 segment relocated at loading time	Simple, Multiple processes	1 segment/process No protection External frag.
Base & Bound	Dynamic mem relocation at runtime	Simple hardware, Multiple processes Protection	1 segment/process, External frag.
Multiple segments	Dynamic mem relocation at runtime	More hardware, Protection, multi segs/process	External frag.

Break

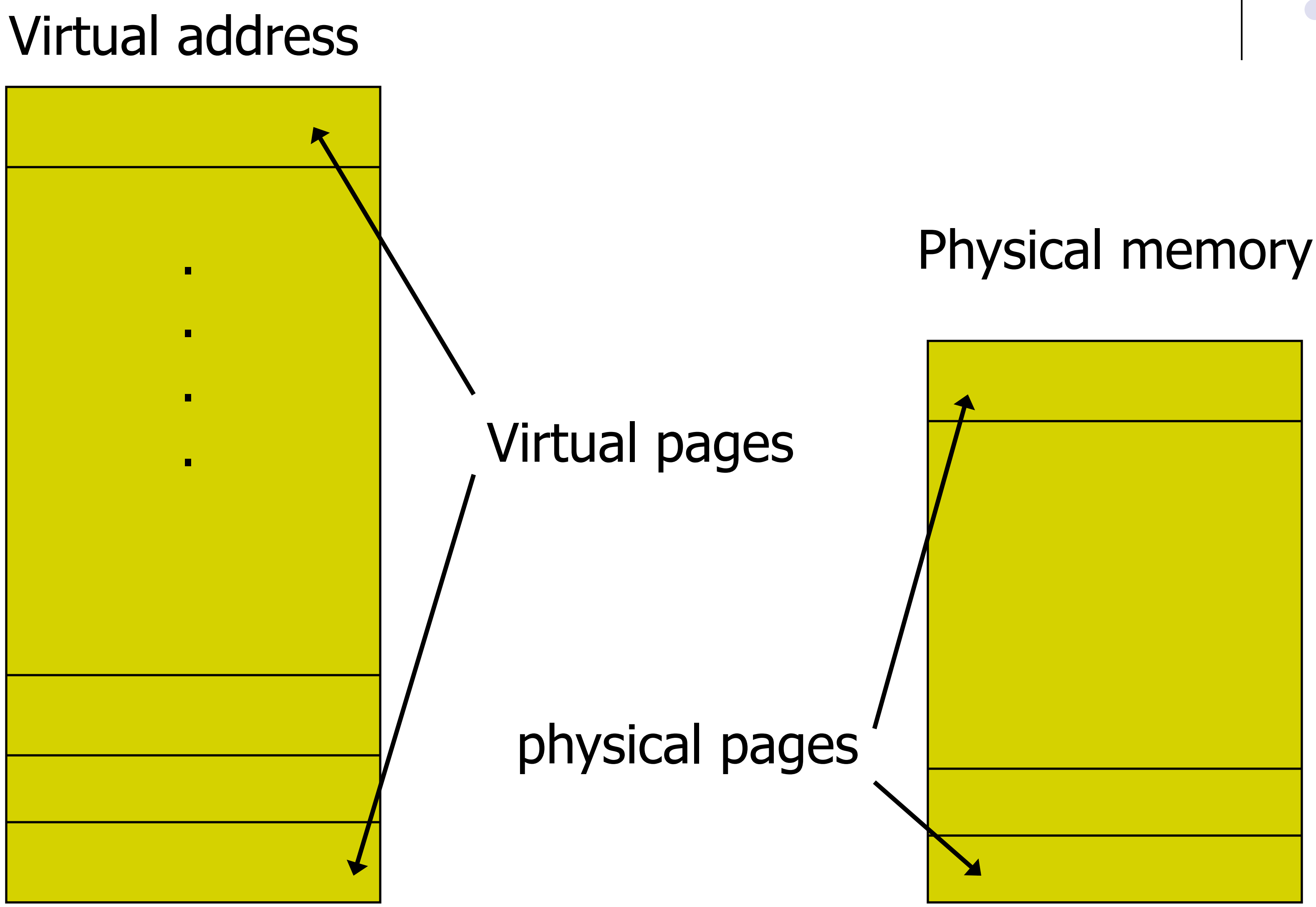
- You’re standing on the surface of the Earth. You walk one mile south, one mile west, and one mile north. You end up exactly where you started. Where are you?
  - Hint: more than one correct answer. Can you get them all?

[review] What fundamentally causes external fragmentation?

- Segments of many different sizes
- Each has to be allocated contiguously

- “Million-dollar” question:
  Physical memory is precious.
  Can we limit the waste to a single hole of X bytes?

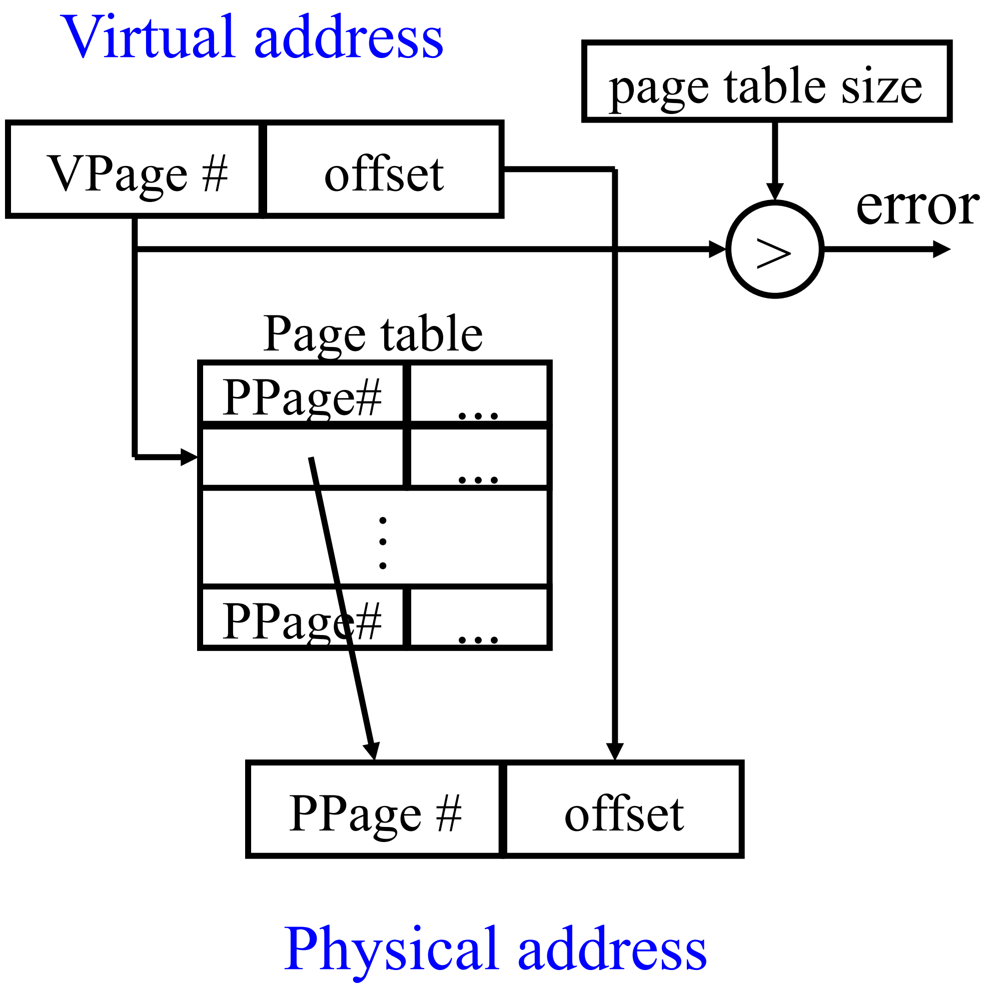
Virtual pages / physical pages



Paging

- Goal:
  - to make allocation and swapping easier (time)
  - to reduce memory fragmentation (space)
- Key idea:
  - Make all chunks of memory the same size, called *pages*
- Implementation:
  - For each process, a *page table* defines the base address of each of that process' pages along with existence and read/write bits
  - Translation?

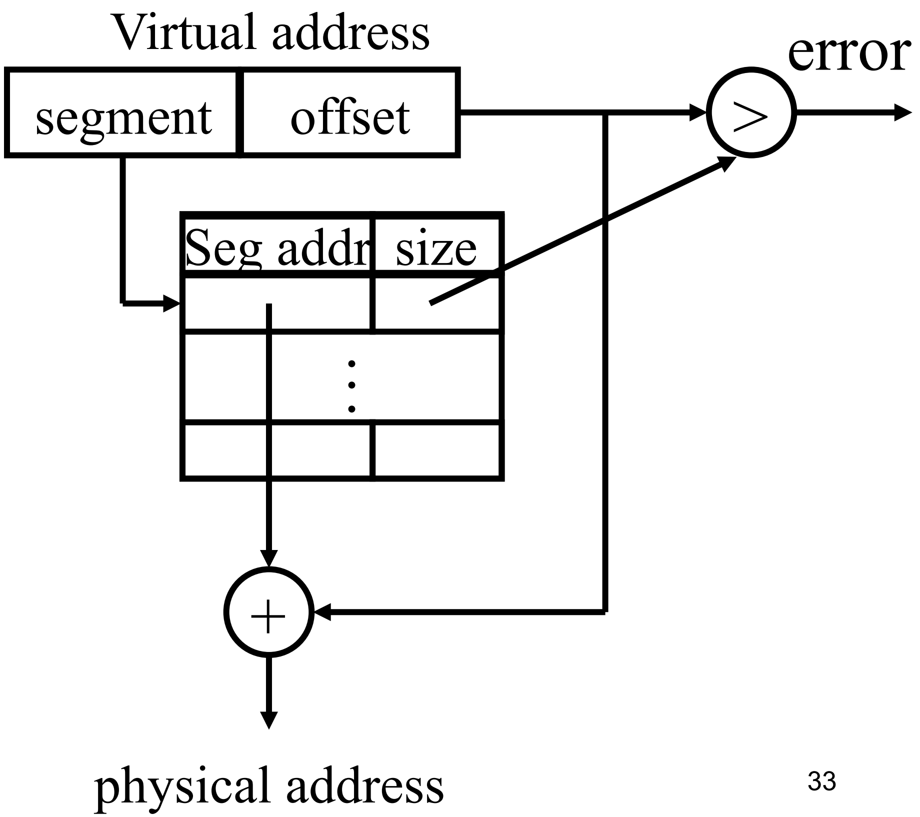
Paging



- Context switch
  - similar to the segmentation scheme
- Pros:
  - easy allocation, keep a free list
  - easy to swap
  - easy to share

Deek thinking: Paging implementation

- Translation: table lookup and bit substitution
- Why is this possible?
- Why cannot we do the same in segmentation?

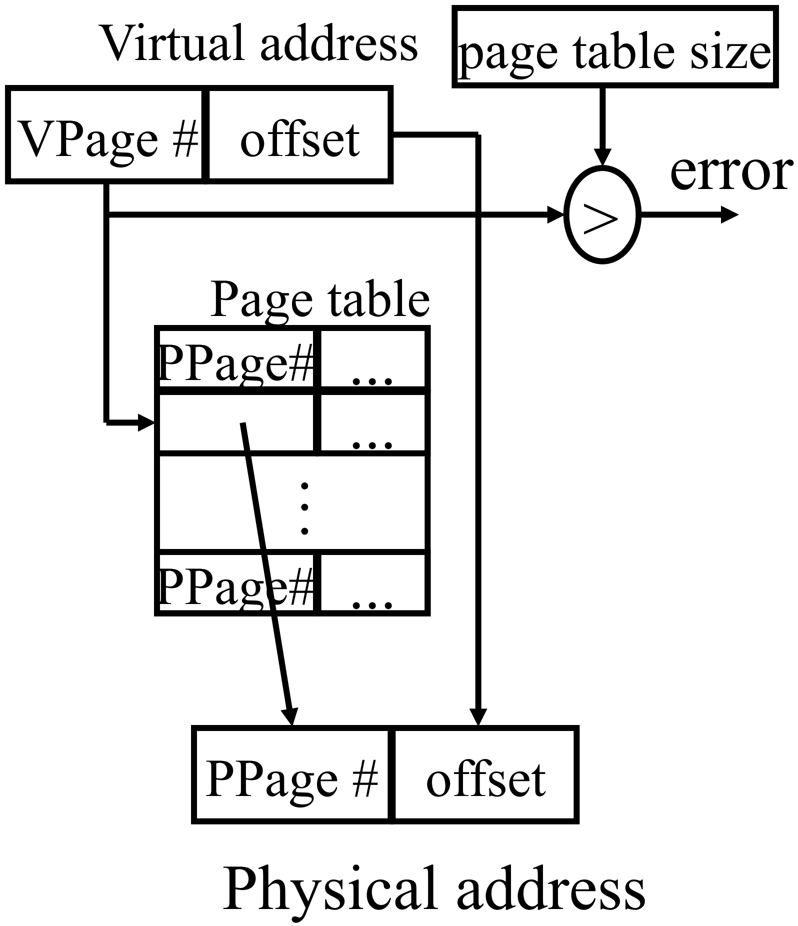


How many PTEs do we need?
(assume page size is 4096 bytes)

- Worst case for 32-bit address machine?

- What about 64-bit address machine?

Paging implementation – how does it really work?



- Where to store page table?
- How to use MMU?
  - Even small page tables too large to load into MMU
  - Page tables kept in mem and MMU only has their base addresses
    - What does MMU have to do?
- Page size?
  - Small page -> big table
    - 32-bit with 4k pages
  - Large page ->small table but large internal fragmentation

Paging vs. segmentation

- Segmentation:
  - External fragmentation
  - Complicated allocation, swapping
  + Small segmentation table

- Paging
  - Internal fragmentation
  + Easy allocation, swapping
  - Large page table

Deep thinking

- Why does the page table have to be contiguous in the physical memory?
  - Why did a segment have to be contiguous in memory?
- For a 4GB virtual address space, we just need 1M PTE (~4MB), what is the big deal?
- My PC has 2GB, why do we need PTEs for the entire 4GB address space?

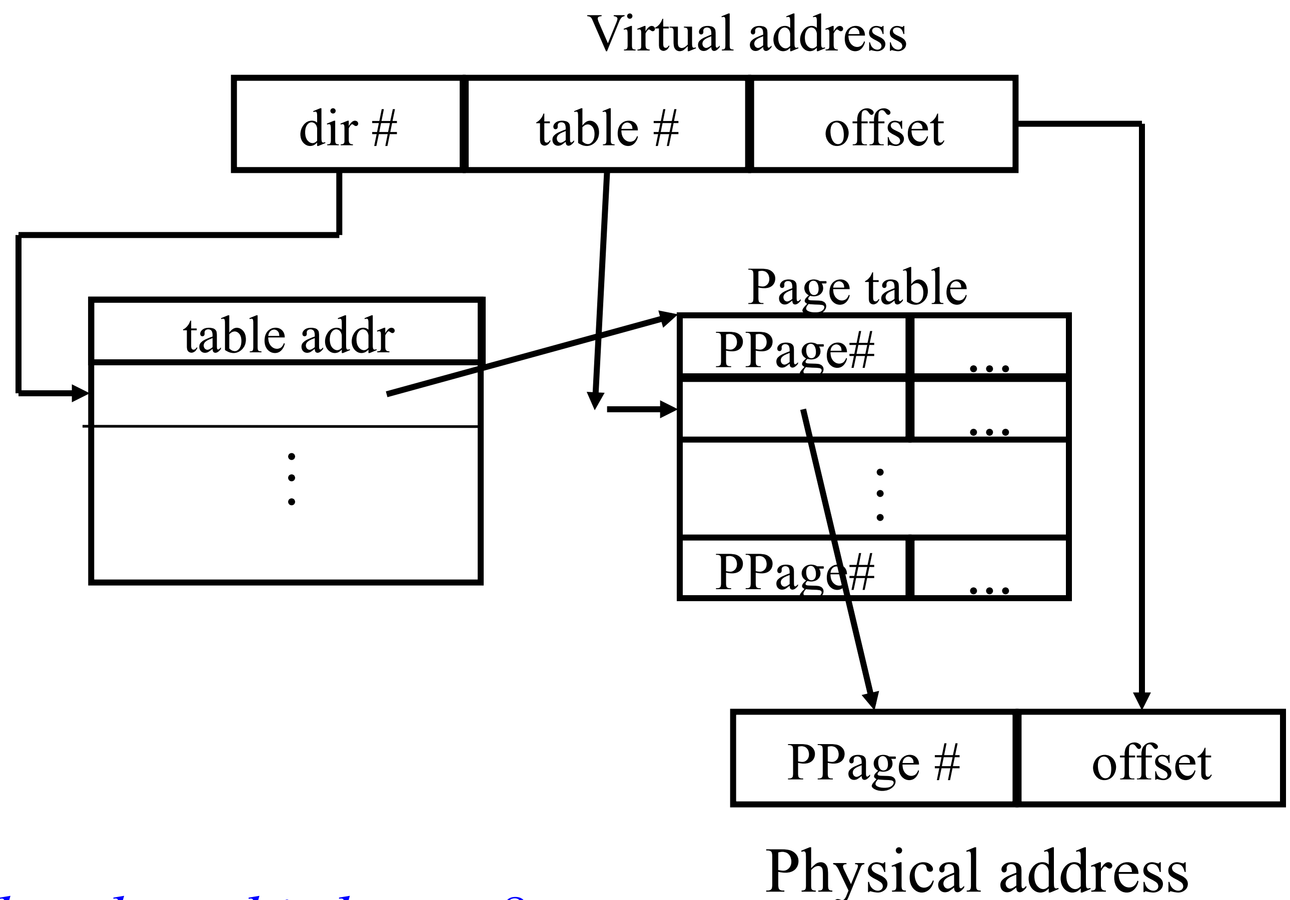
Page table

- The page table has to be consecutive in mem
  - Potentially large
  - Consecutive pages in mem hard to find

- How can we be flexible?

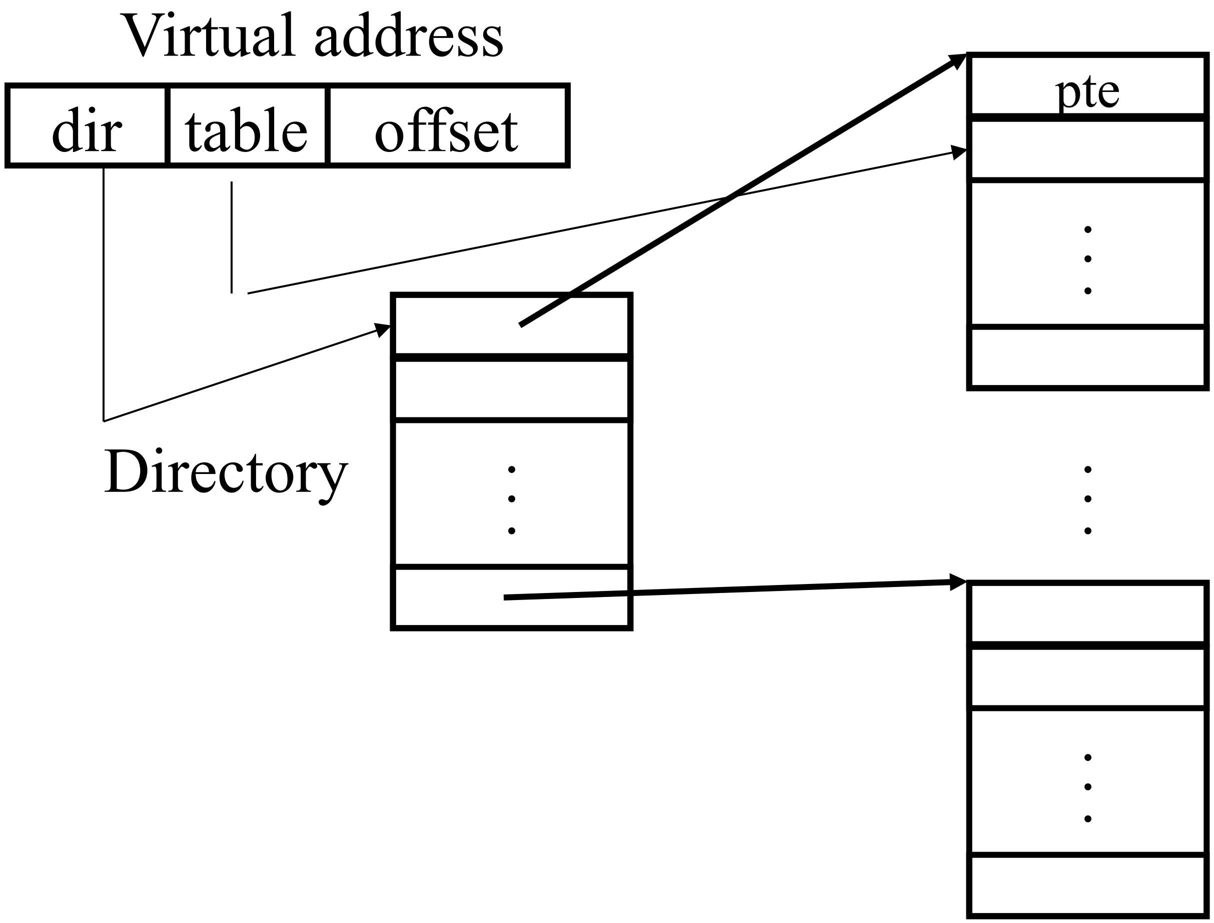
"All computer science problems can be solved with an extra level of indirection."

Two-level page tables



What does this buy us?

Multiple-level page tables

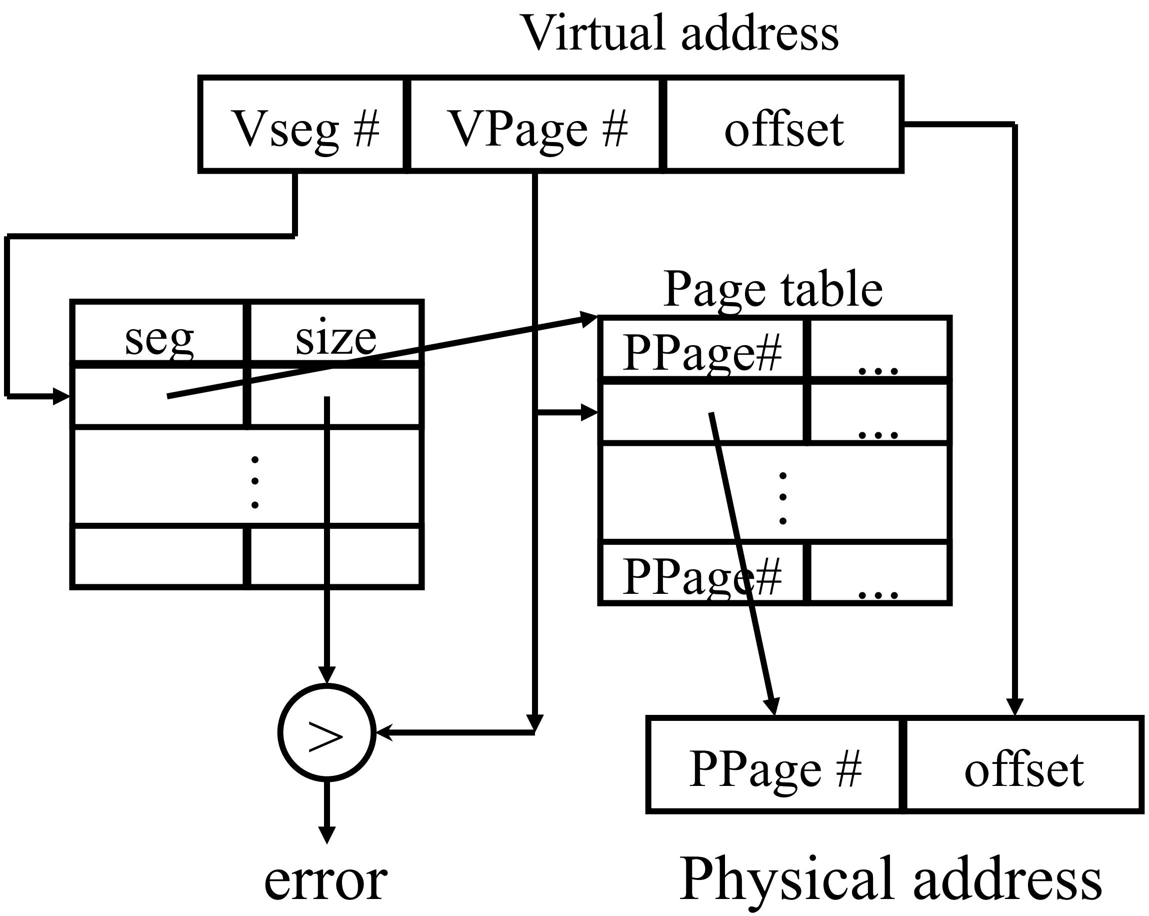


Multi-level page tables

- 3 Advantages?
  - L2 page tables do not have to consecutive
  - They do not have to be allocated before use!
  - They can be swapped out to disk!

The power of an extra level of indirection!

Segmentation with paging



Ex: IBM System 370 (24-bit, 4-bit segment #, 8-bit page #)

Segmentation + paging

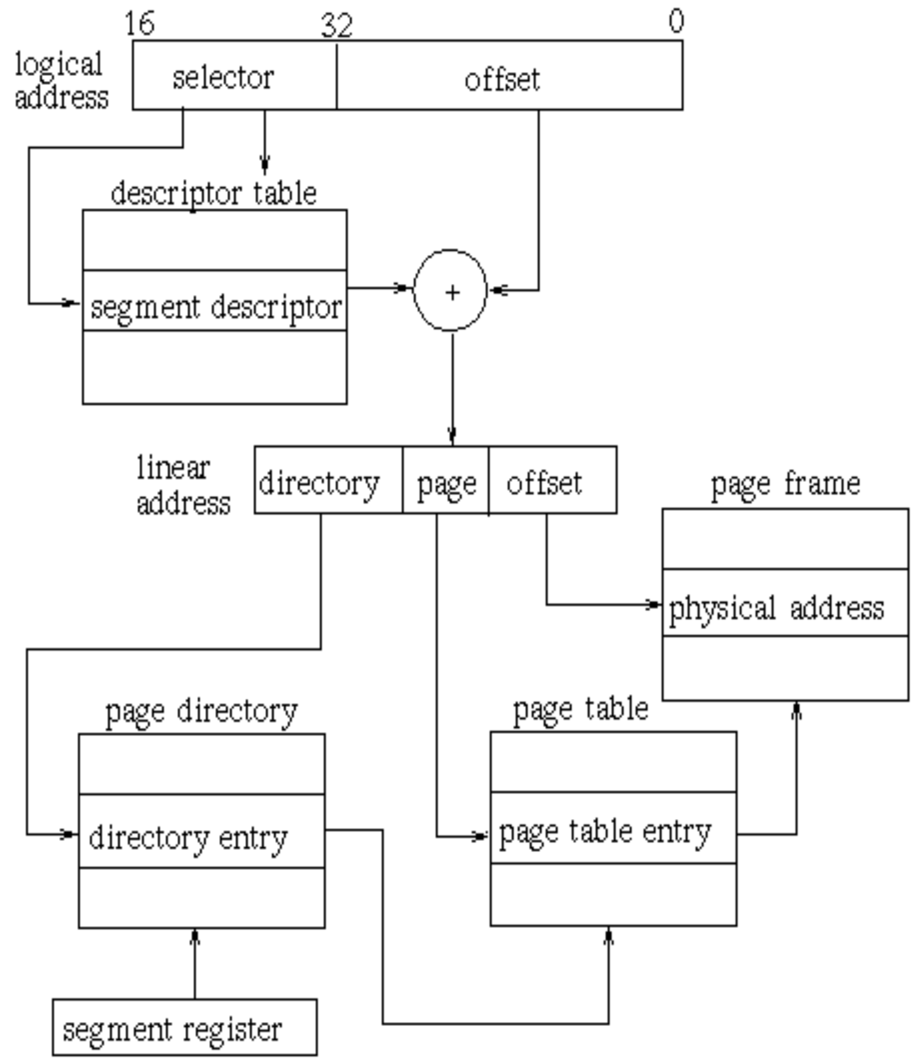
- Use two levels of mapping to make tables manageable:
  - Each segment contains one or more pages
  - Segments correspond to logical units: code, data, stack
  - Segments vary in size and are often large
  - Pages are for easy of management by OS: fixed size -> easy to allocate/free

  - Going from P to P+S is like going from single segment to multiple segments, except at a high level
    - One page table -> many page tables with bases/bounds

The Intel Pentium (1993) (pro, II, III, 4) (Ch 8.7, fig 8.22, 8.23)

- Supports both pure segmentation and segmentation with 1-level paging (page size=4M) or 2-level paging (page size=4k)

- CPU generates logical addresses
    - (selector, offset), 16 bits and 32 bits
    - As many as 16K segments
    - Up to 4GB per segment



Linux on Pentium

- Linux uses 3-level paging
  - For both 32-bit and 64-bit architectures
- On Pentium, degenerates to 2-level paging
  - Middle-level directory has zero bits