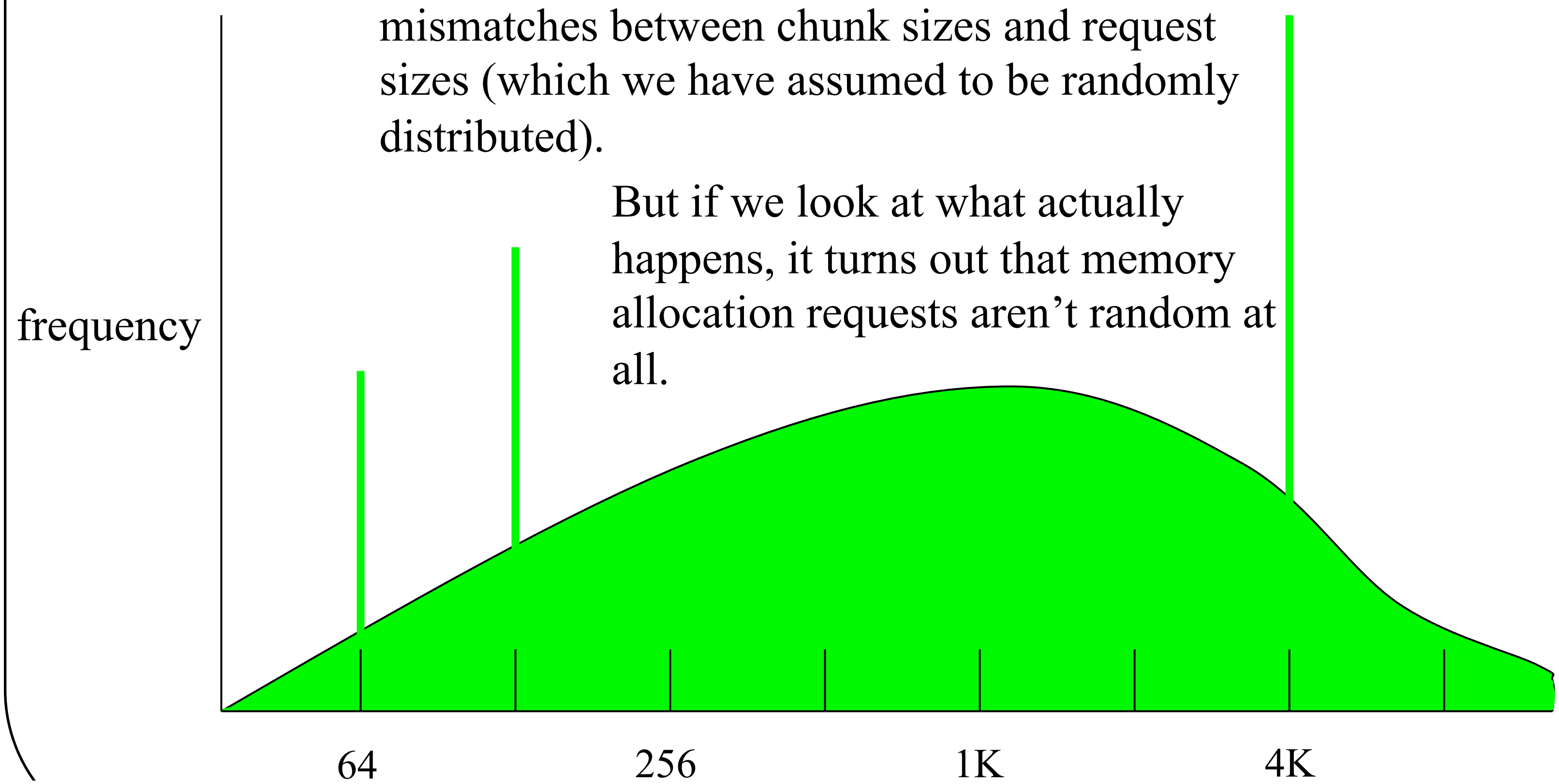
Another Option

- Fixed partition allocations result in internal fragmentation
  - Processes don't use all of the fixed partition
- Dynamic domain allocations result in external fragmentation
  - The elements on the memory free list get smaller and less useful
- Can we strike a balance in between?

A Special Case for Fixed Allocations

Internal fragmentation results from mismatches between chunk sizes and request sizes (which we have assumed to be randomly distributed).

But if we look at what actually happens, it turns out that memory allocation requests aren't random at all.



Why Aren't Memory Request Sizes Randomly Distributed?

- In real systems, some sizes are requested much more often than others
- Many key services use fixed-size buffers
  - File systems (for disk I/O)
  - Network protocols (for packet assembly)
  - Standard request descriptors
- These account for much transient use
  - They are continuously allocated and freed
- OS might want to handle them specially

Buffer Pools

- If there are popular sizes,
  - Reserve special pools of fixed size buffers
  - Satisfy matching requests from those pools
- Benefit: improved efficiency
  - Much simpler than variable domain allocation
    - Eliminates searching, carving, coalescing
  - Reduces (or eliminates) external fragmentation
- But we must know how much to reserve
  - Too little, and the buffer pool will become a bottleneck
  - Too much, and we will have a lot of unused buffer space
- <u>Only</u> satisfy perfectly matching requests
  - Otherwise, back to internal fragmentation

How Are Buffer Pools Used?

- Process requests a piece of memory for a special purpose
  - E.g., to send a message
- System supplies one element from buffer pool
- Process uses it, completes, frees memory
  - Maybe explicitly
  - Maybe implicitly, based on how such buffers are used
    - E.g., sending the message will free the buffer "behind the process' back" once the message is gone

Dynamically Sizing Buffer Pools

- If we run low on fixed sized buffers
  - Get more memory from the free list
  - Carve it up into more fixed sized buffers
- If our free buffer list gets too large
  - Return some buffers to the free list
- If the free list gets dangerously low
  - Ask each major service with a buffer pool to return space
- This can be tuned by a few parameters:
  - Low space (need more) threshold
  - High space (have too much) threshold
  - Nominal allocation (what we free down to)
- Resulting system is highly adaptive to changing loads

Lost Memory

- One problem with buffer pools is memory leaks
  - The process is done with the memory
  - But doesn't free it
- Also a problem when a process manages its own memory space
  - E.g., it allocates a big area and maintains its own free list
- Long running processes with memory leaks can waste huge amounts of memory

Garbage Collection

- One solution to memory leaks
- Don't count on processes to release memory
- Monitor how much free memory we've got
- When we run low, start garbage collection
  - Search data space finding every object pointer
  - Note address/size of all accessible objects
  - Compute the compliment (what is inaccessible)
  - Add all inaccessible memory to the free list

How Do We Find All Accessible Memory?

- Object oriented languages often enable this
  - All object references are tagged
  - All object descriptors include size information
- It is often possible for system resources
  - Where all possible references are known
    - (E.g., we know who has which files open)
- How about for the general case?

General Garbage Collection

- Well, what would you need to do?
- Find all the pointers in allocated memory
- Determine "how much" each points to
- Determine what was and was not still pointed to
- Free what isn't pointed to
- Why might that be difficult?

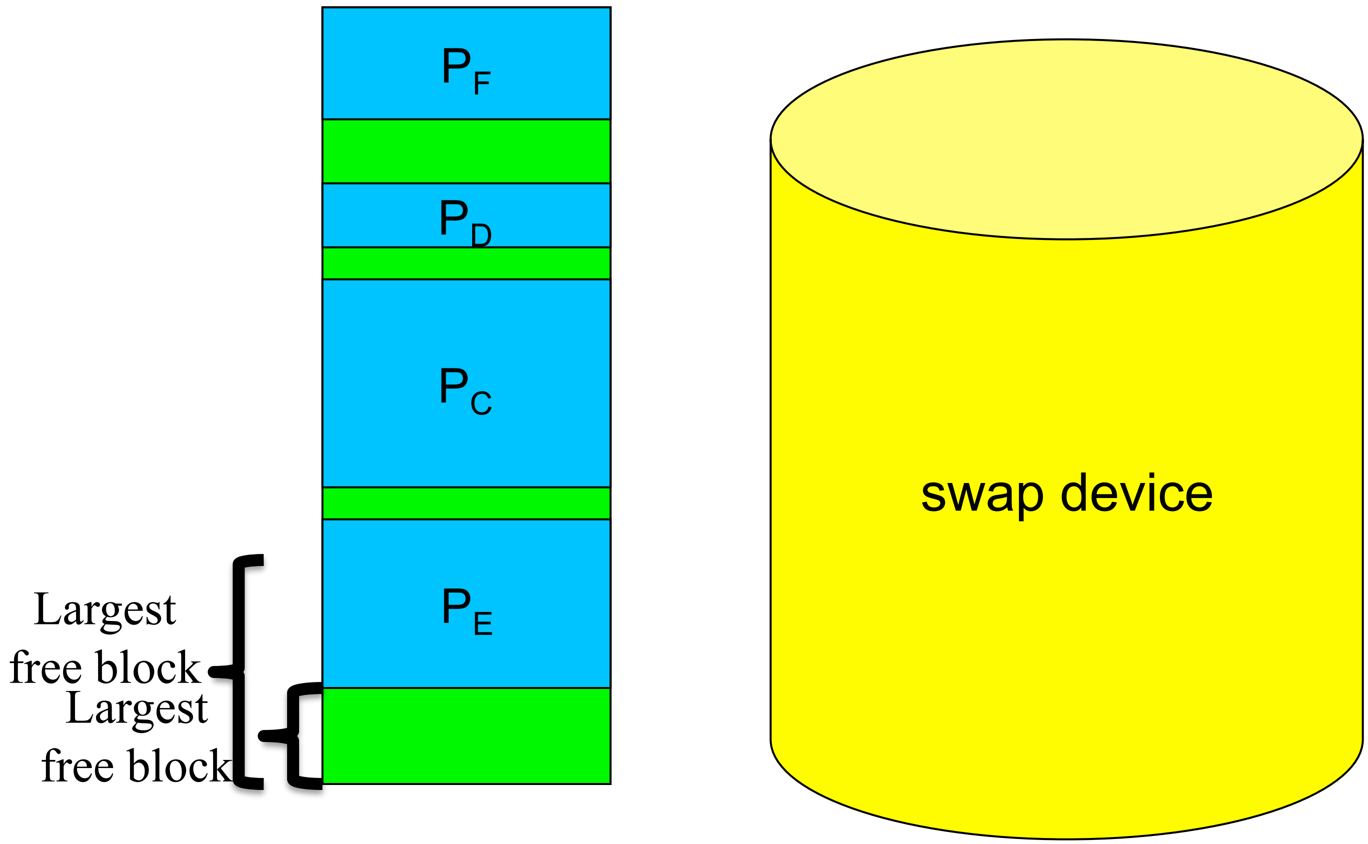
Problems With General Garbage Collection

- A location in the data or stack segments might <u>seem</u> to contain addresses, but ...
  - Are they truly pointers, or might they be other data types whose values happen to resemble addresses?
  - Even if they are truly pointers, are they themselves still accessible?
  - We might be able to infer this (recursively) for pointers in dynamically allocated structures …
  - But what about pointers in statically allocated (potentially global) areas?
- And how much is "pointed to," one word or a million?

Compaction and Relocation

- Garbage collection is just another way to free memory
  - Doesn't greatly help or hurt fragmentation
- Ongoing activity can starve coalescing
  - Chunks reallocated before neighbors become free
- We could stop accepting new allocations
  - But resulting convoy on memory manager would trash throughput
- We need a way to rearrange active memory
  - Re-pack all processes in one end of memory
  - Create one big chunk of free space at other end

Memory Compaction



Now let's compact!

An obvious improvement!

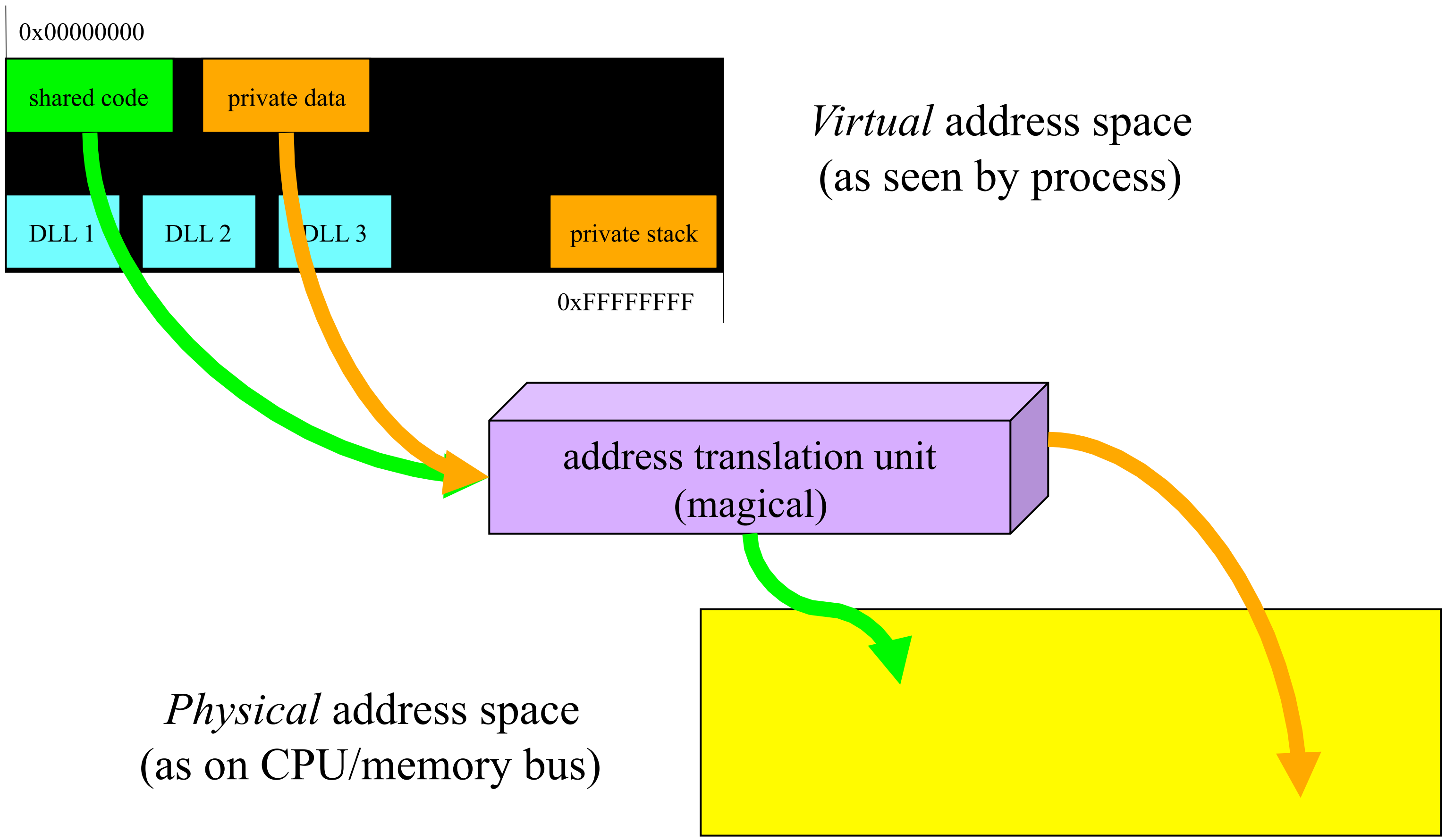
All This Requires Is Relocation . . .

- The ability to move a process
  - From region where it was initially loaded
  - Into a new and different region of memory
- What's so hard about that?
- All addresses in the program will be wrong
  - References in the code segment
    - Calls and branches to other parts of the code
    - References to variables in the data segment
  - Plus new pointers created during execution
    - That point into data and stack segments

The Relocation Problem

- It is not generally feasible to re-relocate a process
  - Maybe we could relocate references to code
    - If we kept the relocation information around
  - But how can we relocate references to data?
    - Pointer values may have been changed
    - New pointers may have been created
- We could never find/fix all address references
  - Like the general case of garbage collection
- Can we make processes location independent?

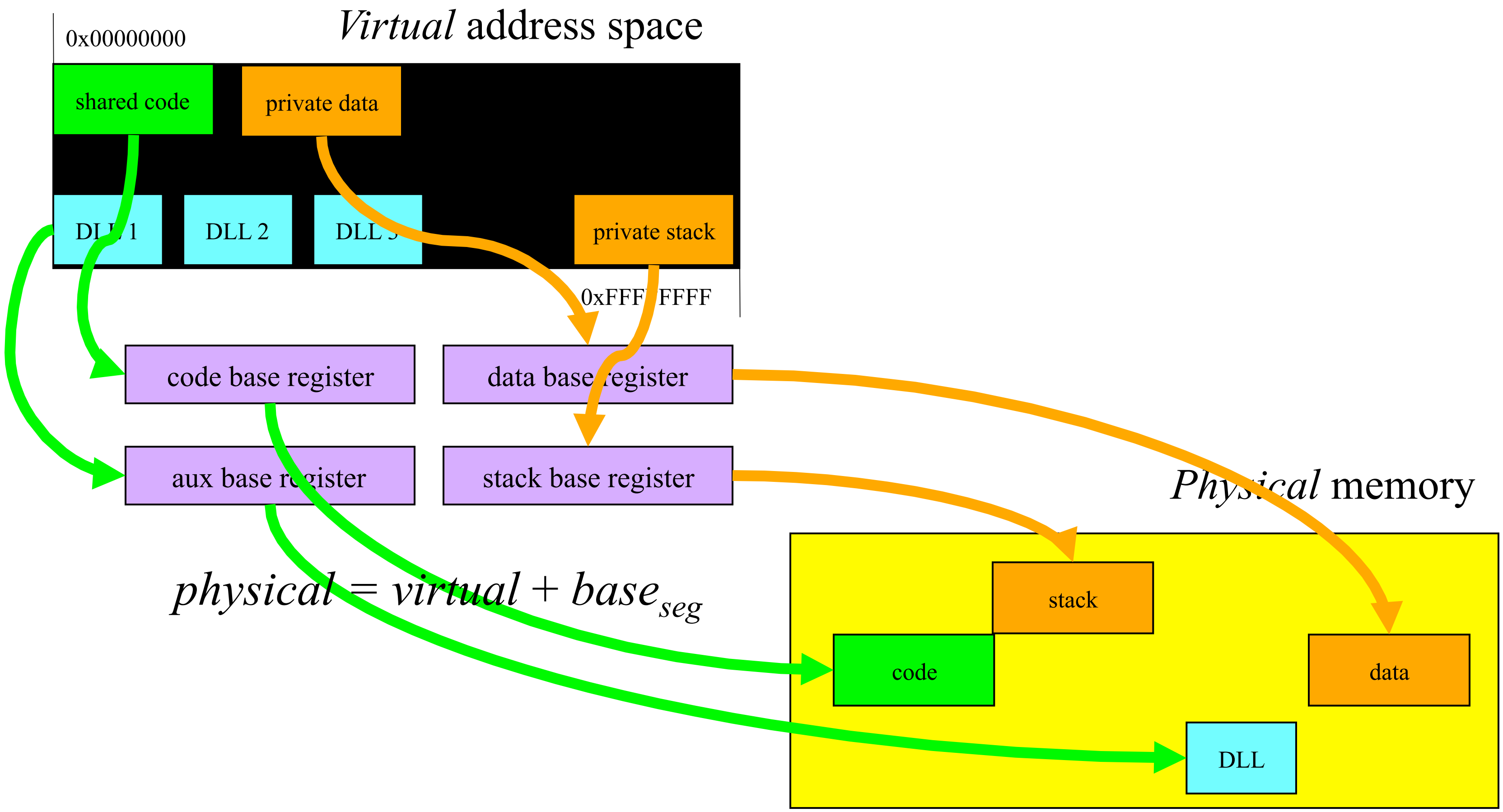
Virtual Address Spaces



Memory Segment Relocation

- A natural model
  - Process address space is made up of multiple segments
  - Use the segment as the unit of relocation
  - Long tradition, from the IBM system 360 to Intel x86 architecture
- Computer has special relocation registers
  - They are called segment base registers
  - They point to the start (in physical memory) of each segment
  - CPU automatically adds base register to every address
- OS uses these to perform virtual address translation
  - Set base register to start of region where program is loaded
  - If program is moved, reset base registers to new location
  - Program works no matter where its segments are loaded

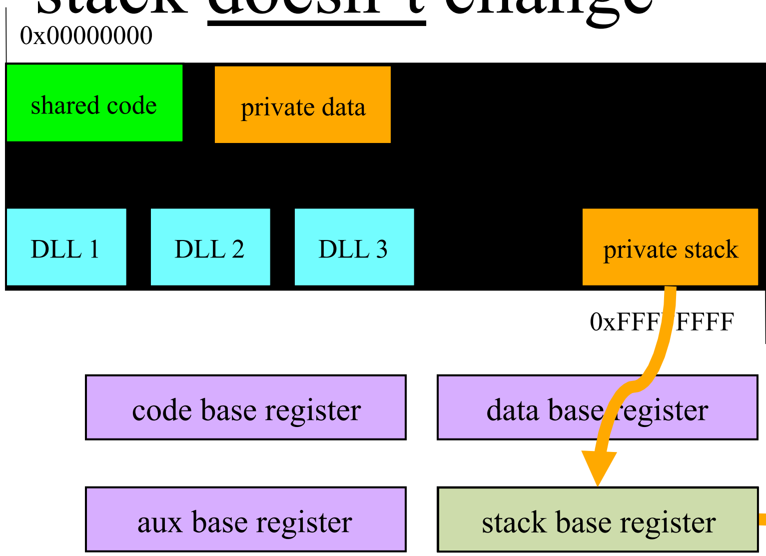
How Does Segment Relocation Work?



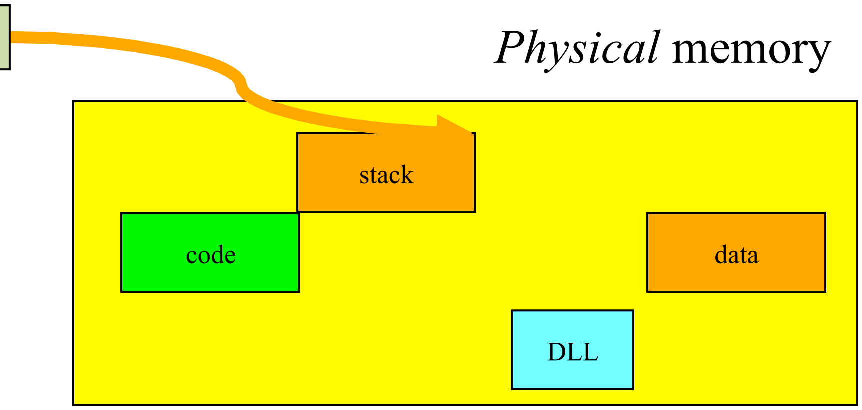
$$physical = virtual + base_{seg}$$

Relocating a Segment

The virtual address of the stack <u>doesn't</u> change



Let's say we need to move the stack in physical memory



$$physical = virtual + base_{seg}$$

We just change the value in the stack base register

Relocation and Safety

- A relocation mechanism (like base registers) is good
  - It solves the relocation problem
  - Enables us to move process segments in physical memory
  - Such relocation turns out to be insufficient
- We also need protection
  - Prevent process from reaching outside its allocated memory
    - E.g., by overrunning the end of a mapped segment
- Segments also need a length (or limit) register
  - Specifies maximum legal offset (from start of segment)
  - Any address greater than this is illegal (in the hole)
  - CPU should report it via a <u>segmentation</u> exception (trap)

How Much of Our Problem Does Relocation Solve?

- We can use variable sized domains
  - Cutting down on internal fragmentation
- We can move domains around
  - Which helps coalescing be more effective
  - But still requires contiguous chunks of data for segments
  - So external fragmentation is still a problem
- We need to get rid of the requirement of contiguous segments