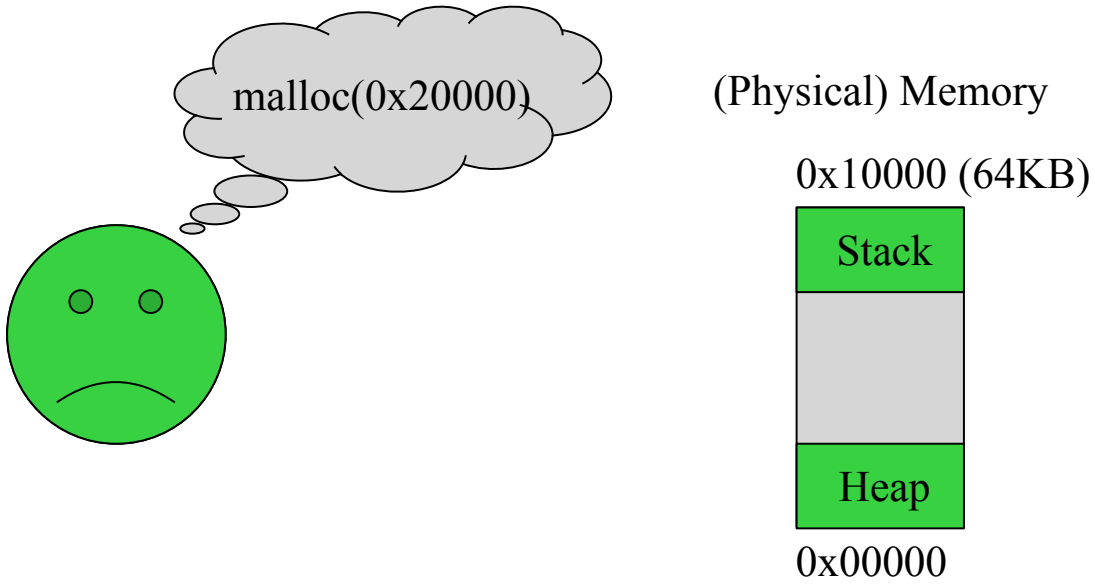
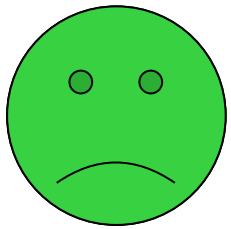
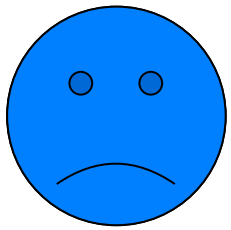
Virtual Memory

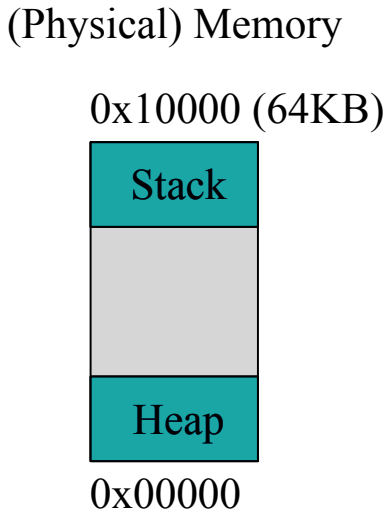
Learning to Play Well With Others



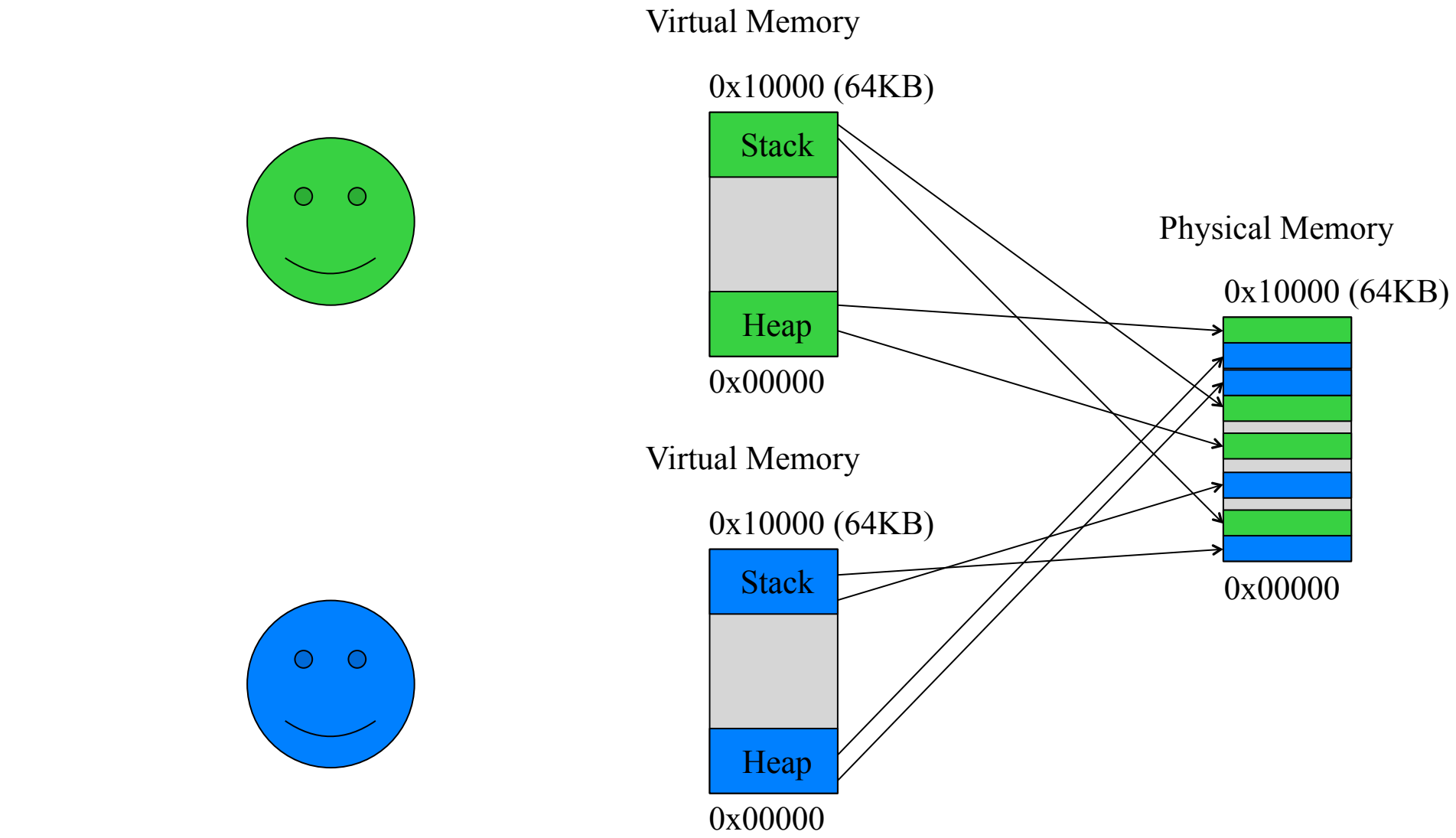
Learning to Play Well With Others



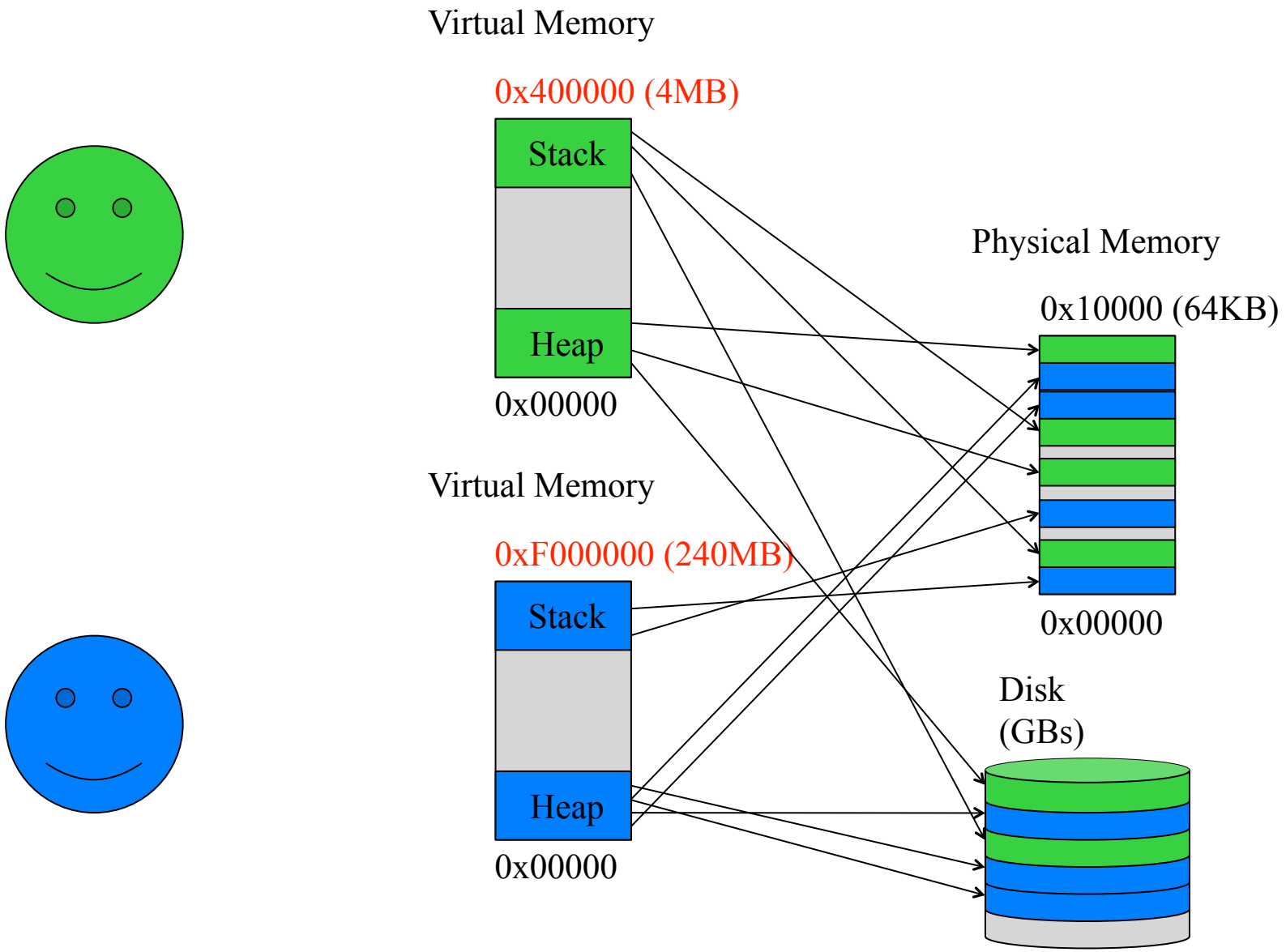




Learning to Play Well With Others



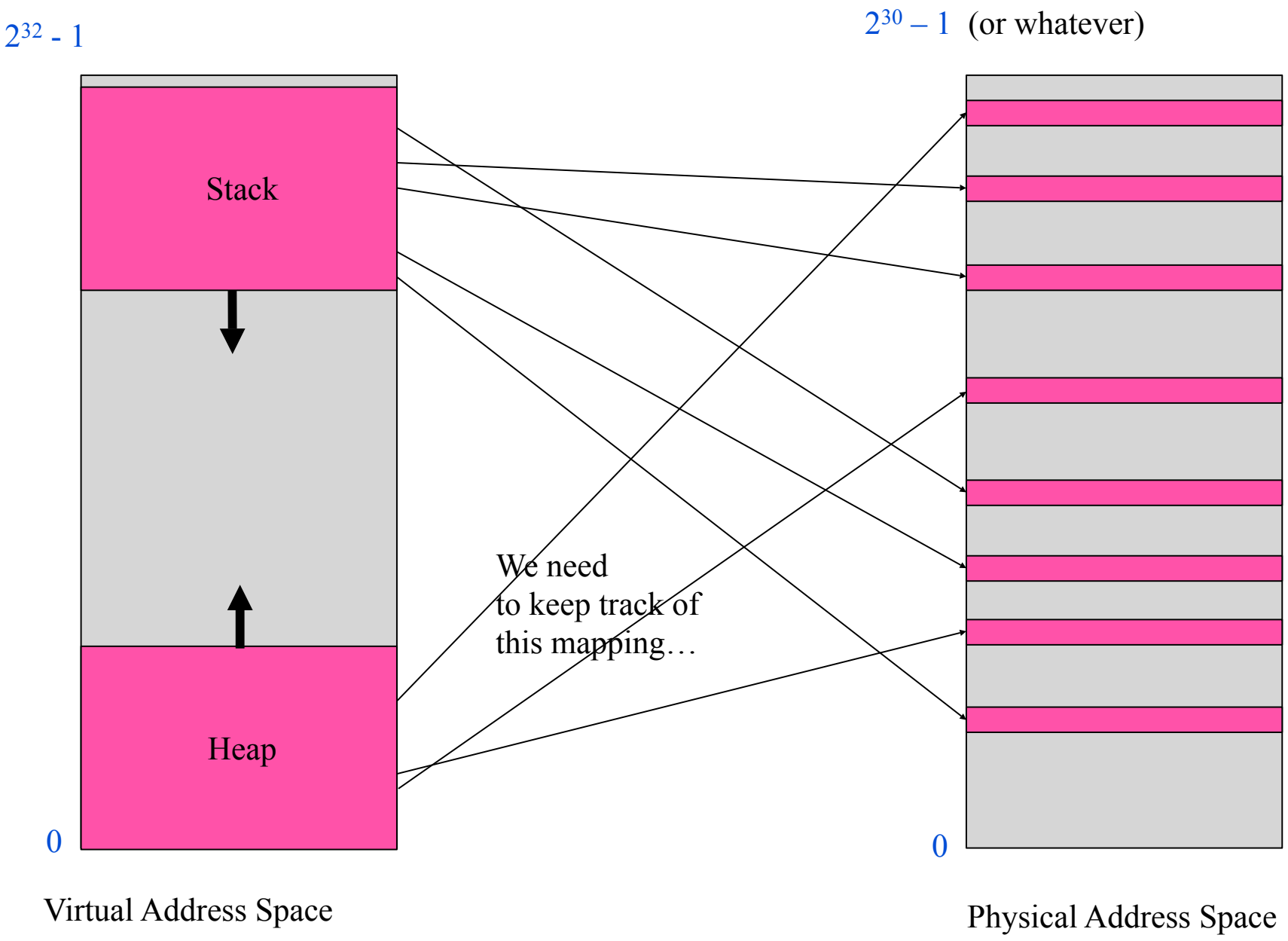
Learning to Play Well With Others



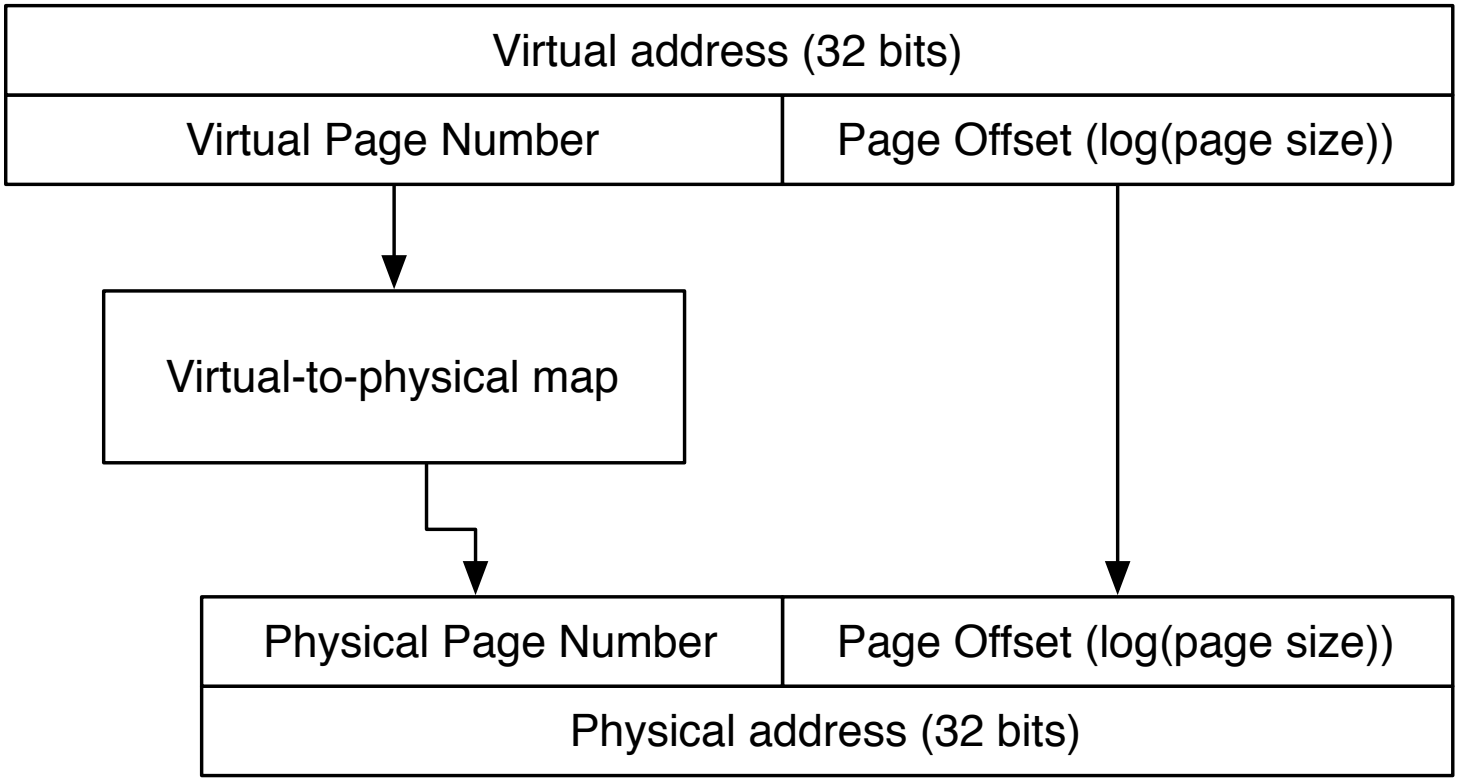
Mapping

- Virtual-to-physical mapping
  - Virtual --> "virtual address space"
  - physical --> "physical address space"
- We will break both address spaces up into "pages"
  - Typically 4KB in size, although sometimes large
- Use a "page table" to map between virtual pages and physical pages.
- The processor generates "virtual" addresses
  - They are translated via "address translation" into physical addresses.

Implementing Virtual Memory



The Mapping Process



Two Problems With VM

- How do we store the map compactly?
- How do we translation quickly?

How Big is the map?

- 32 bit address space:
  - 4GB of virtual addresses
  - 1MPages
  - Each entry is 4 bytes (a 32 bit physical address)
  - 4MB of map
- 64 bit address space
  - 16 exabytes of virtual address space
  - 4PetaPages
  - Entry is 8 bytes
  - 64PB of map

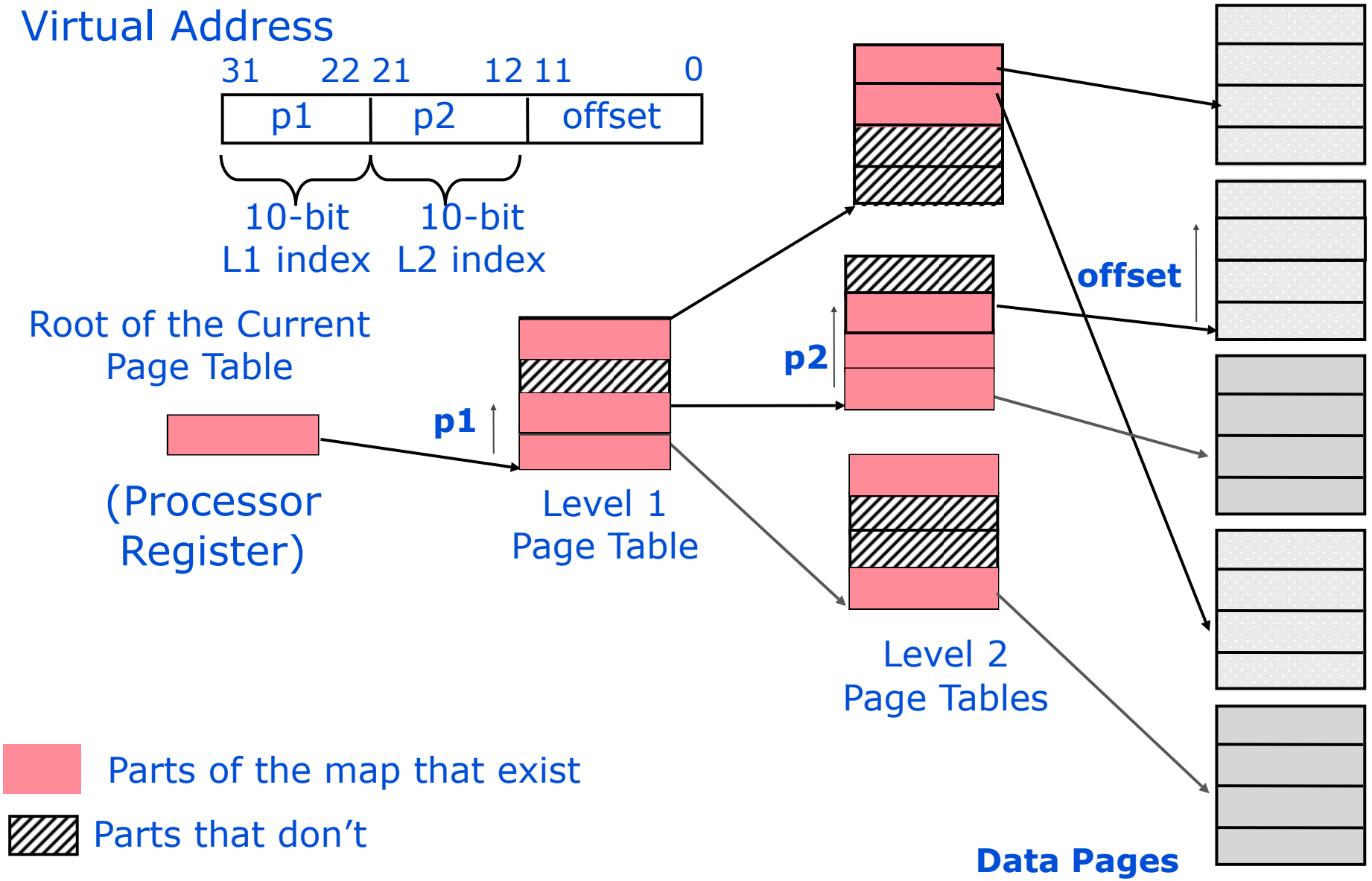
Shrinking the map

- Only store the entries that matter (i.e.,. enough for your physical address space)
- 64GB on a 64bit machine
  - 16M pages, 128MB of map
- This is still pretty big.
- Representing the map is now hard because we need a "sparse" representation.
  - The OS allocates stuff all over the place.
  - For security, convenience, or caching optimizations
  - For instance: The stack is at the "top" of memory. The heap is at the "bottom"
- How do you represent this "sparse" map?

Hierarchical Page Tables

- Break the virtual page number into several pieces
- If each piece has N bits, build an 2^N-ary tree
- Only store the part of the tree that contain valid pages
- To do translation, walk down the tree using the pieces to select with child to visit.

Hierarchical Page Table



Making Translation Fast

- Address translation has to happen for *every* memory access
- This potentially puts it squarely on the critical for memory operation (which are already slow)

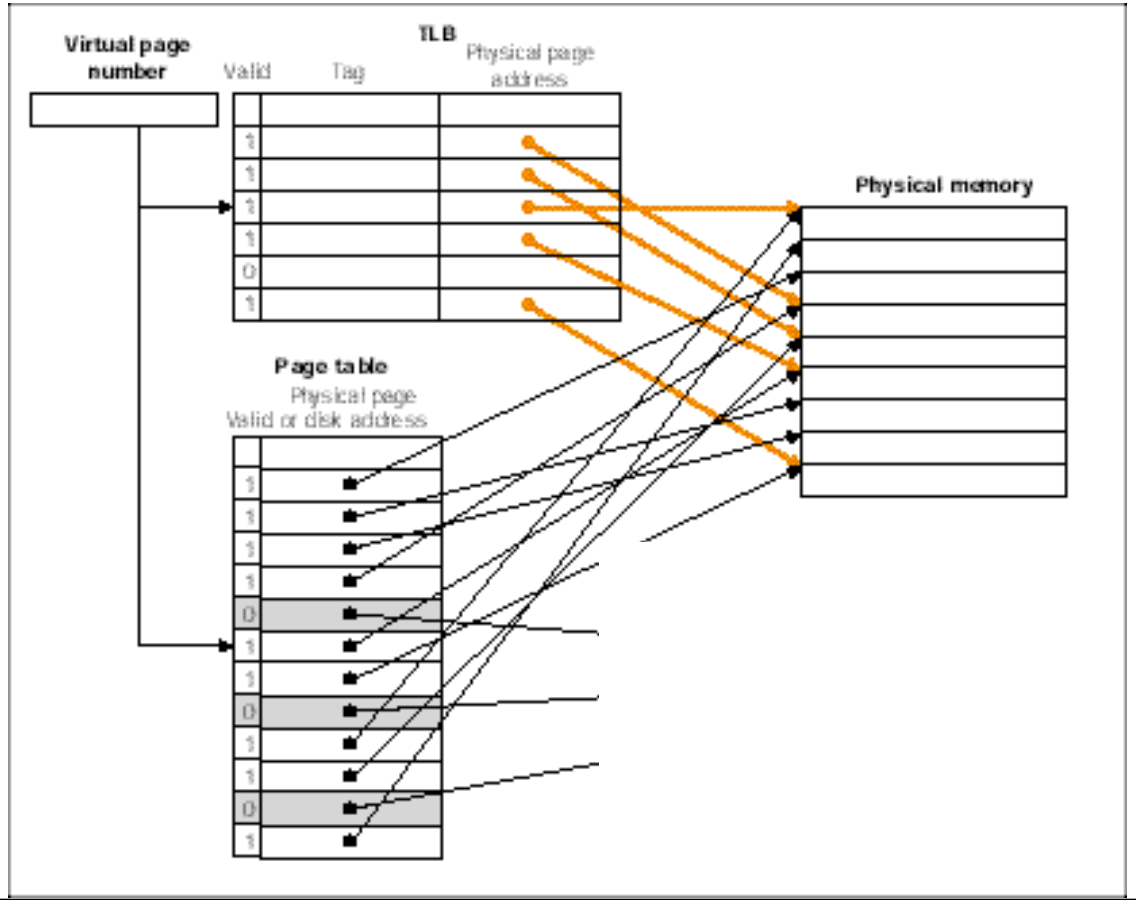
"Solution 1": Use the Page Table

- We could walk the page table on every memory access
- Result: every load or store requires an additional 3-4 loads to walk the page table.

- Unacceptable performance hit.

Solution 2: TLBs

- We have a large pile of data (i.e., the page table) and we want to access it very quickly (i.e., in one clock cycle)
- So, build a cache for the page mapping, but call it a "translation lookaside buffer" or "TLB"



TLBs

- TLBs are small (maybe 128 entries), highly-associative (often fully-associative) caches for page table entries.
- This raises the possibility of a TLB miss, which can be expensive
  - To make them cheaper, there are "hardware page table walkers" -- specialized state machines that can load page table entries into the TLB without OS intervention
  - This means that the page table format is now part of the big-A architecture.
  - Typically, the OS can disable the walker and implement its own format.