CSCI 564 Advanced Computer Architecture

Lecture 07: Virtual Memory

Dr. Bo Wu (with modifications by Sumner Evans)

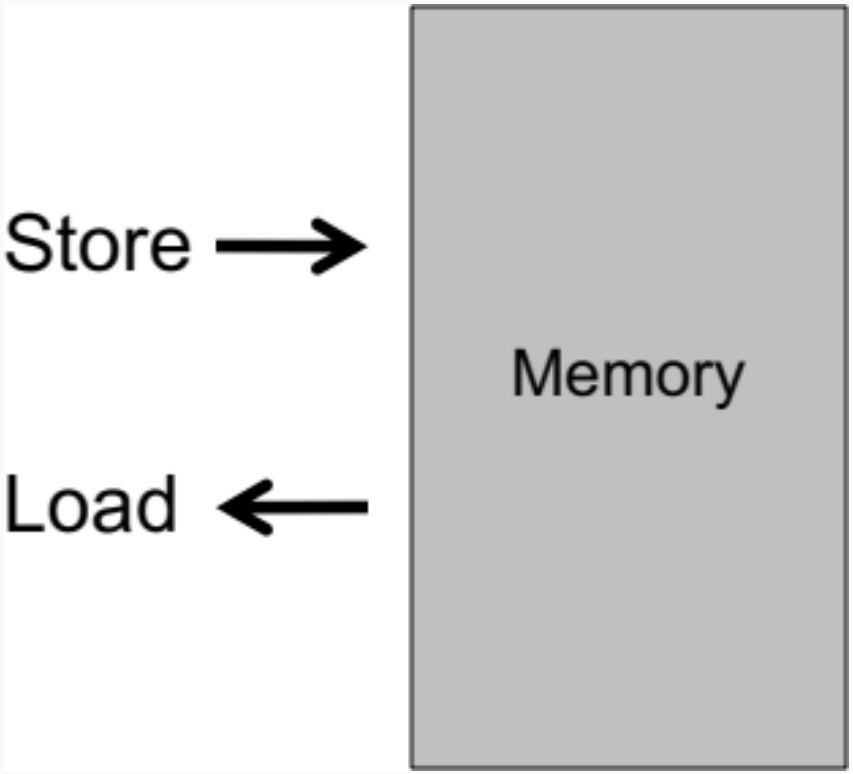
March 3, 2021

Colorado School of Mines

Why Do We Need Virtual Memory?

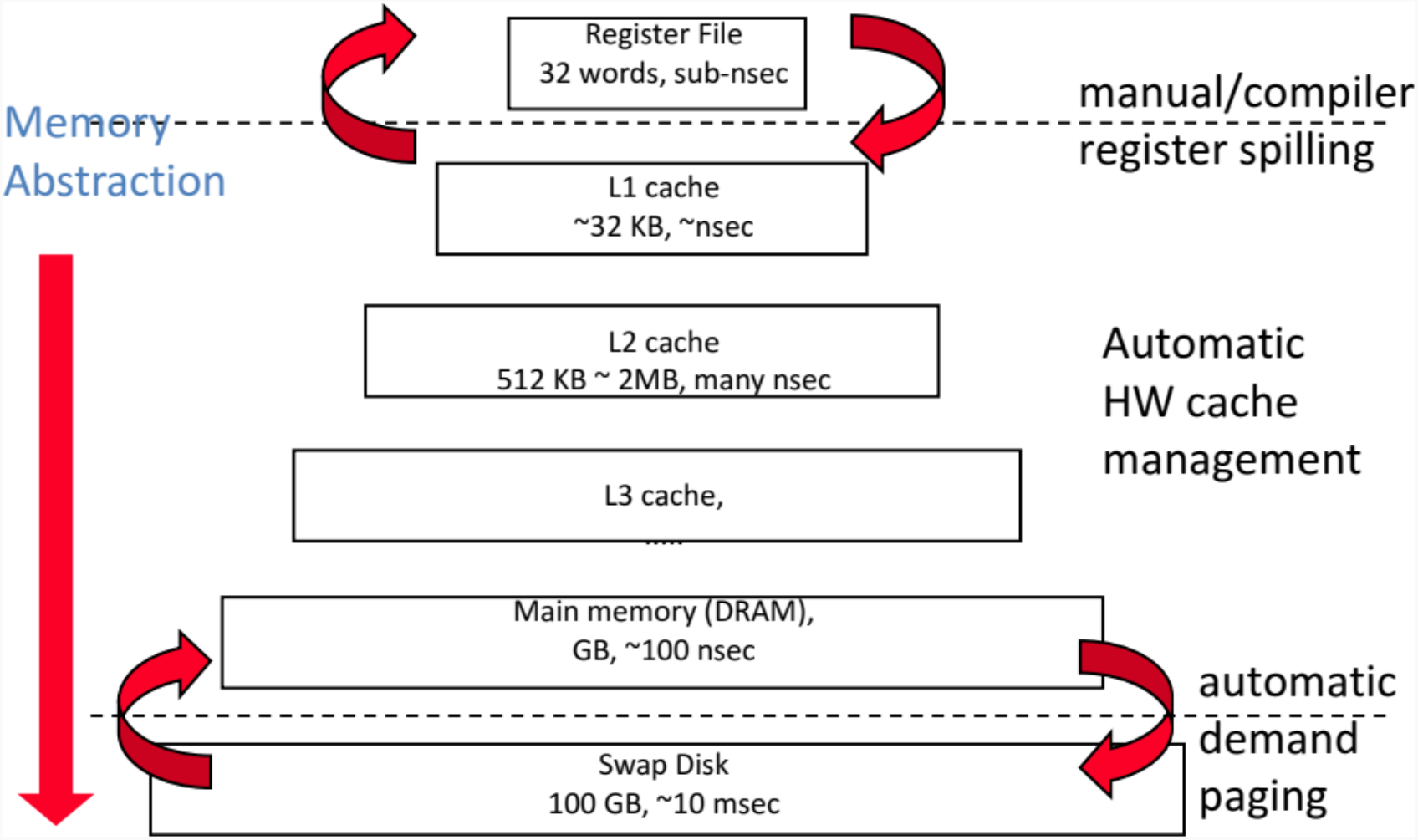
Memory: Ideal Programmer's View

To a programmer, memory should be a black-box.

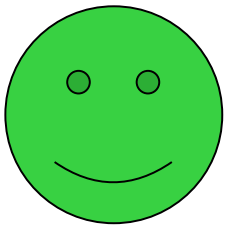


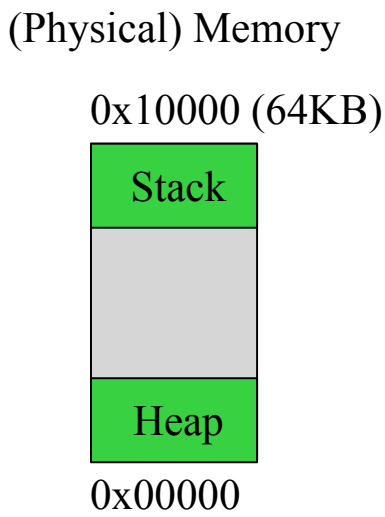
Ideally, memory should also be a zero-latency, infinite bandwidth, infinite capacity, linear memory space.

A Modern Memory Hierarchy

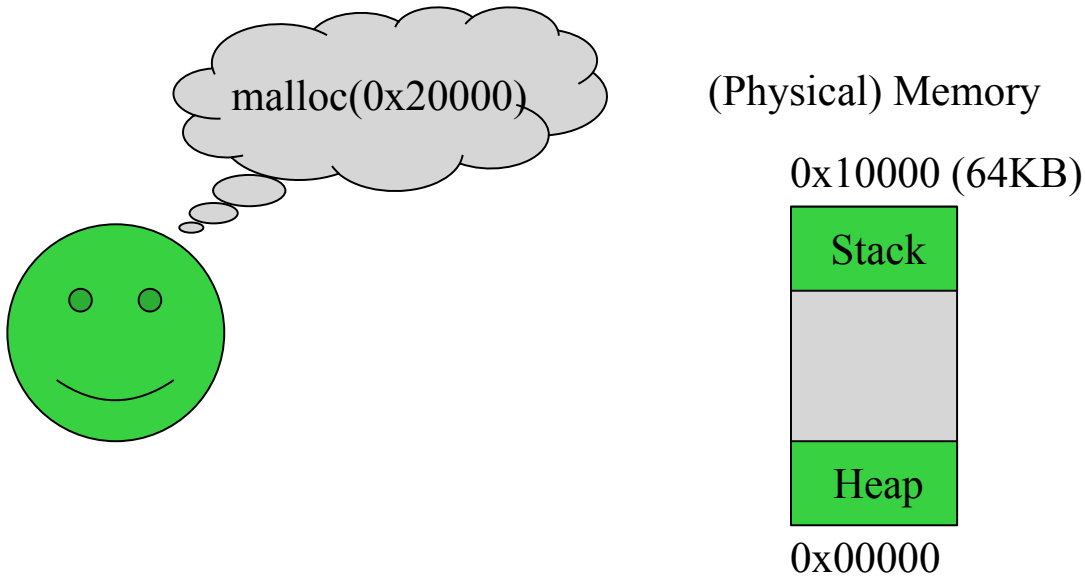


Learning to Play Well With Others

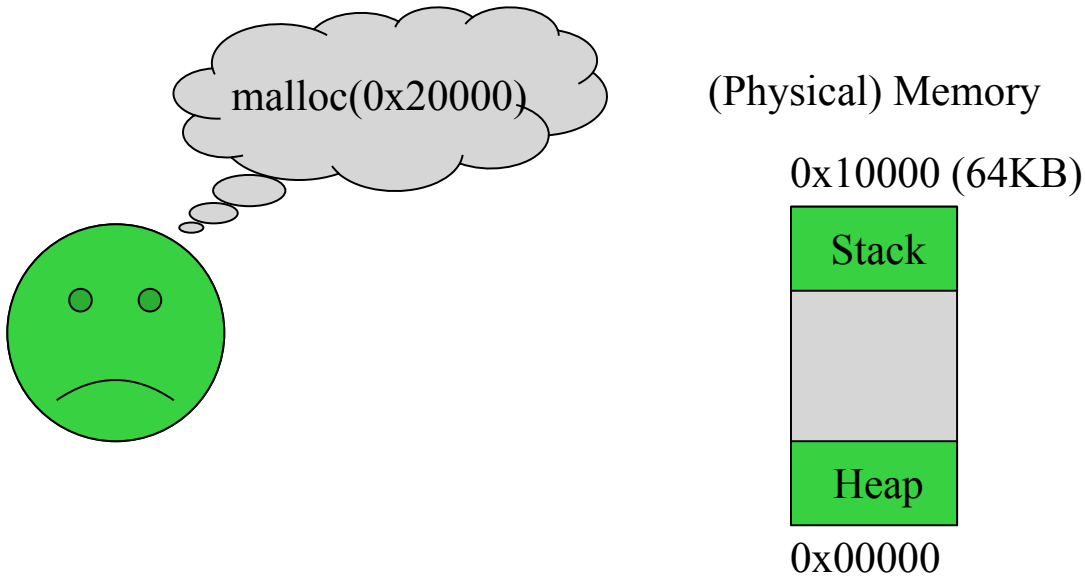




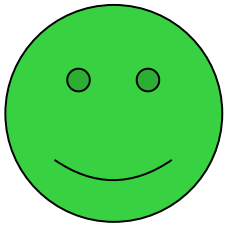
Learning to Play Well With Others

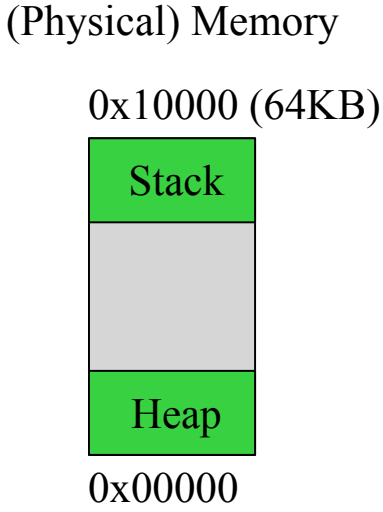


Learning to Play Well With Others

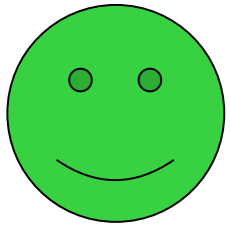


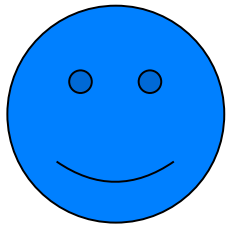
Learning to Play Well With Others

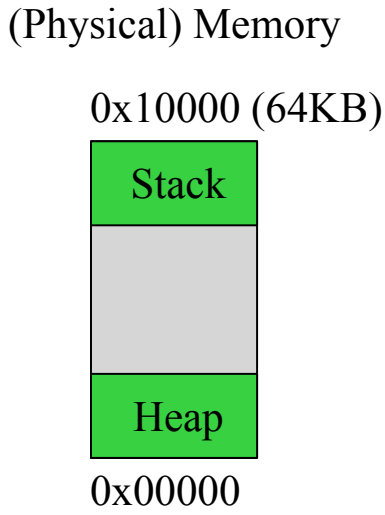




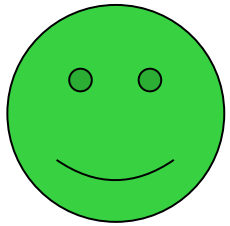
Learning to Play Well With Others

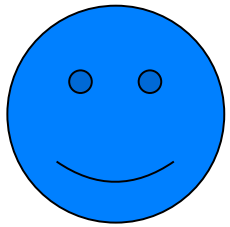


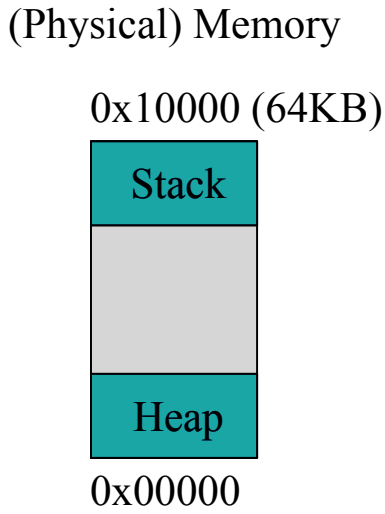




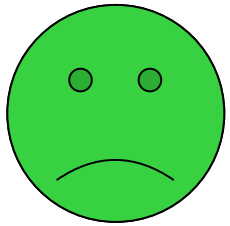
Learning to Play Well With Others

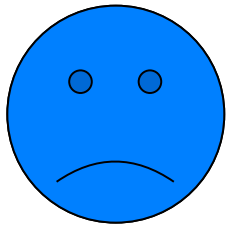


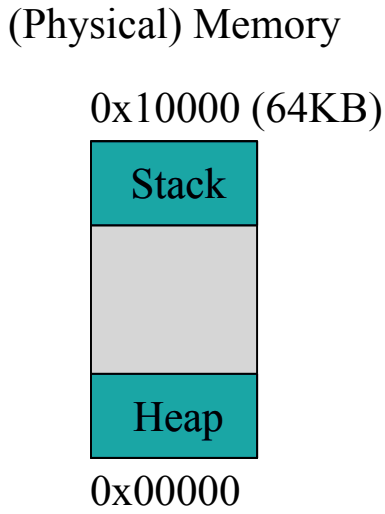


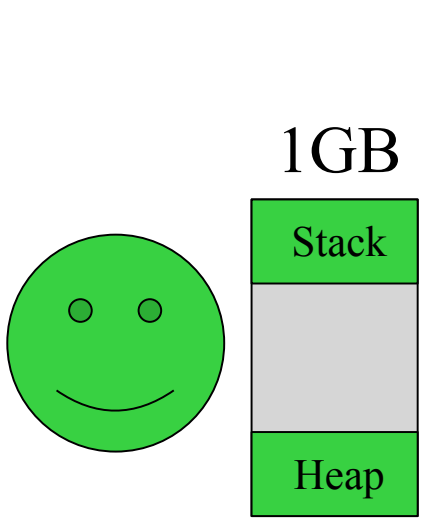


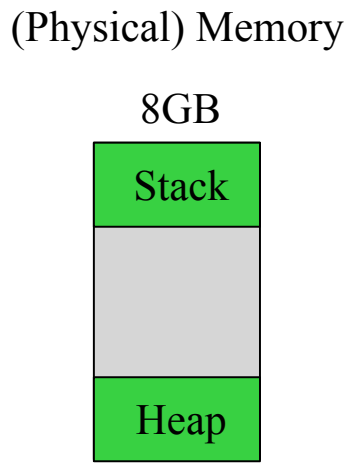
Learning to Play Well With Others

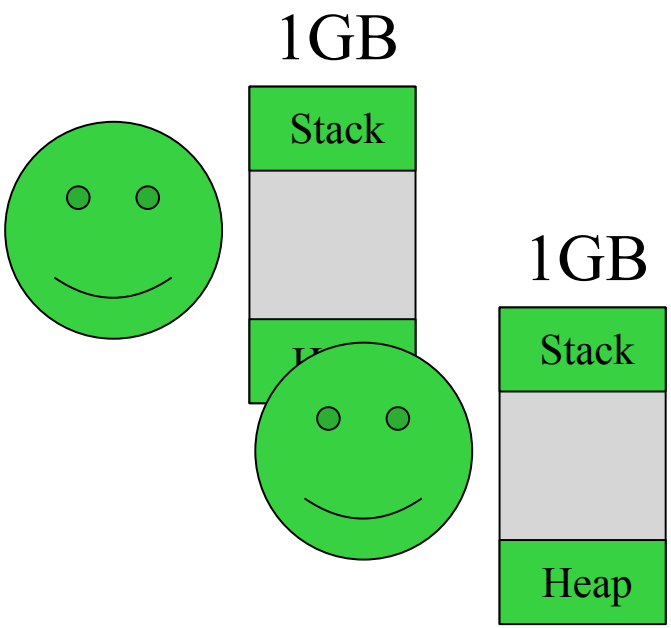




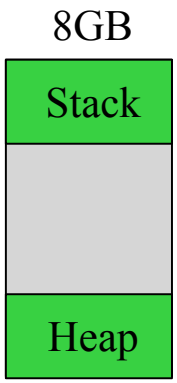


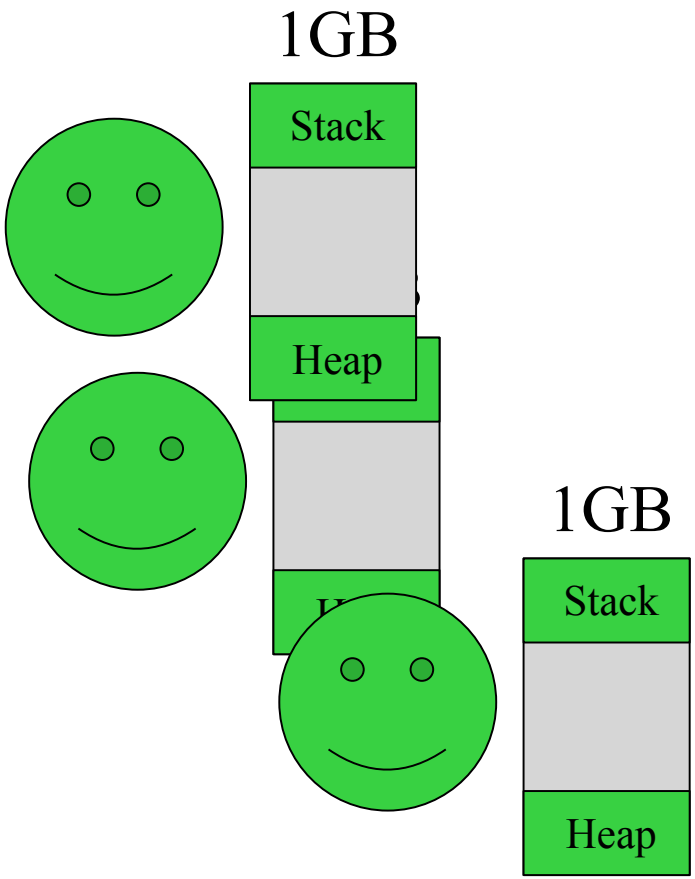

1GB
Stack
Heap


(Physical) Memory
8GB
Stack
Heap



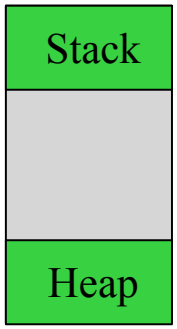
(Physical) Memory

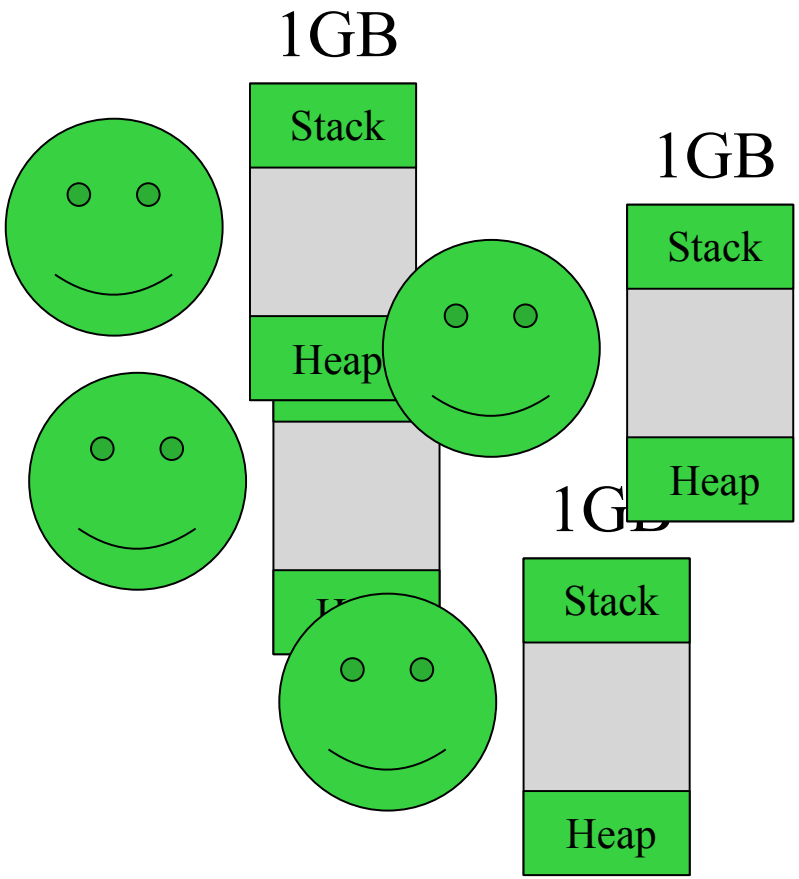




(Physical) Memory

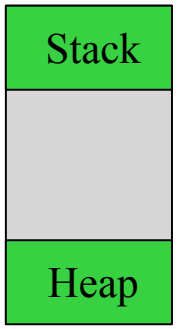
8GB

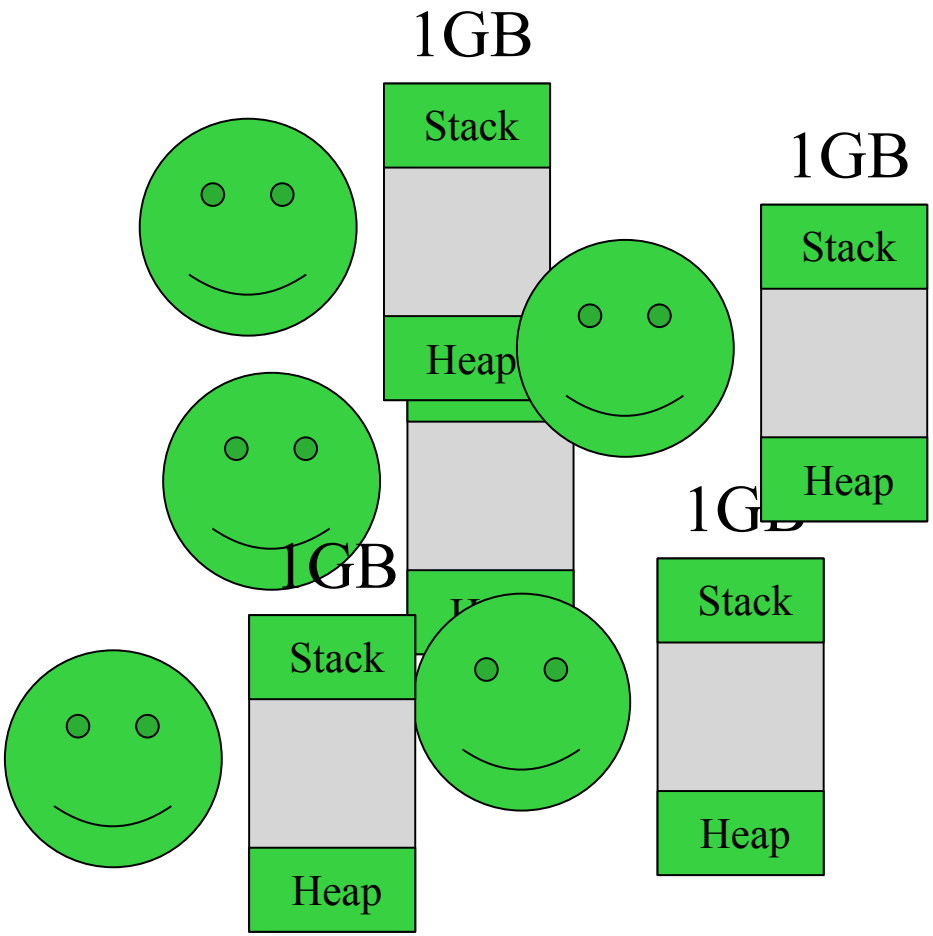




(Physical) Memory

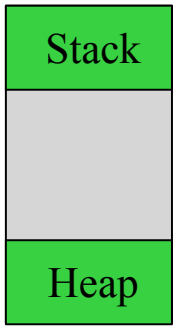
8GB

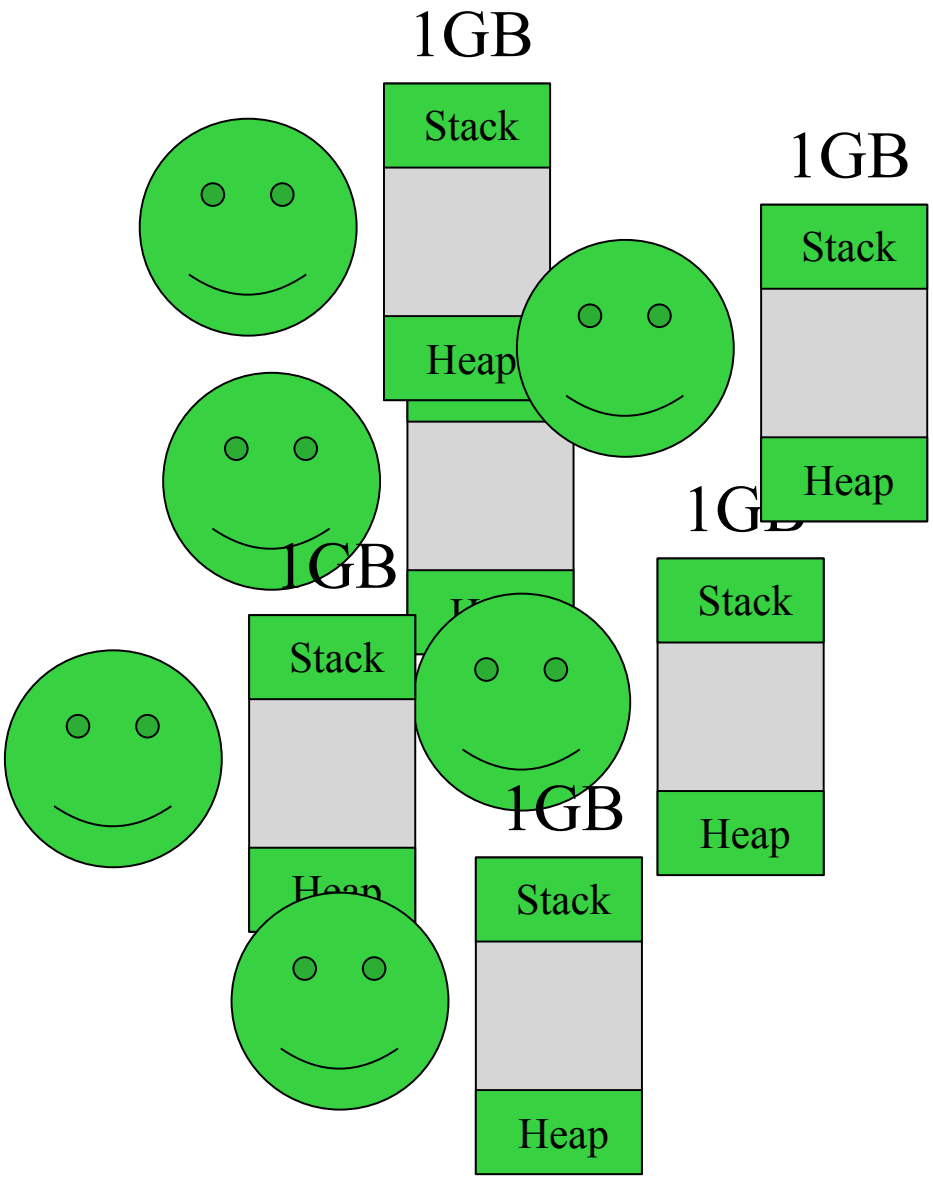




(Physical) Memory

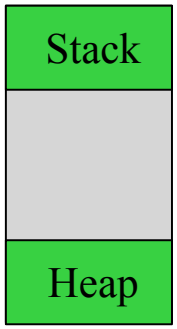
8GB

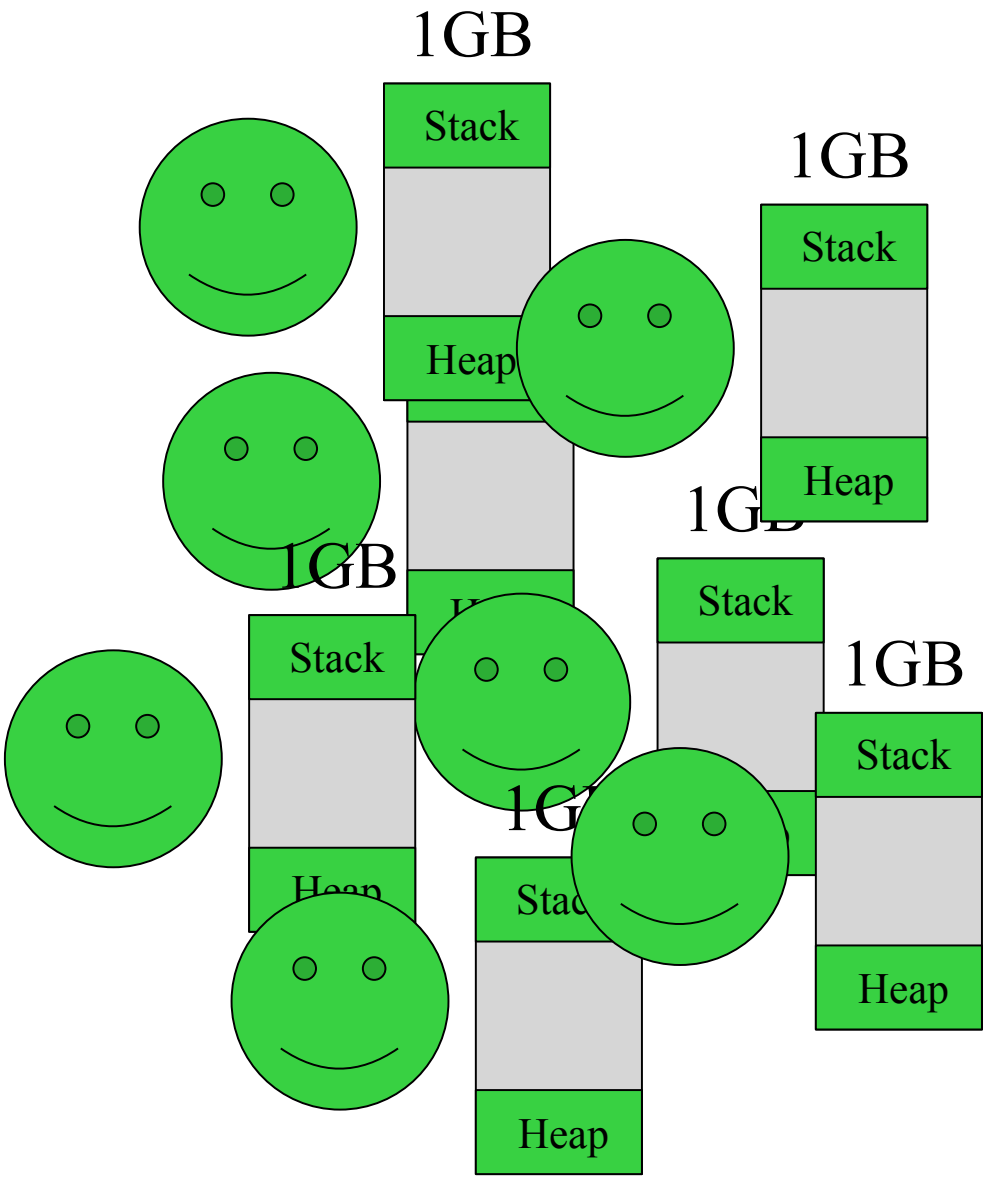




(Physical) Memory

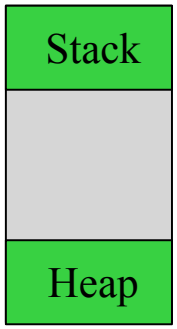
8GB

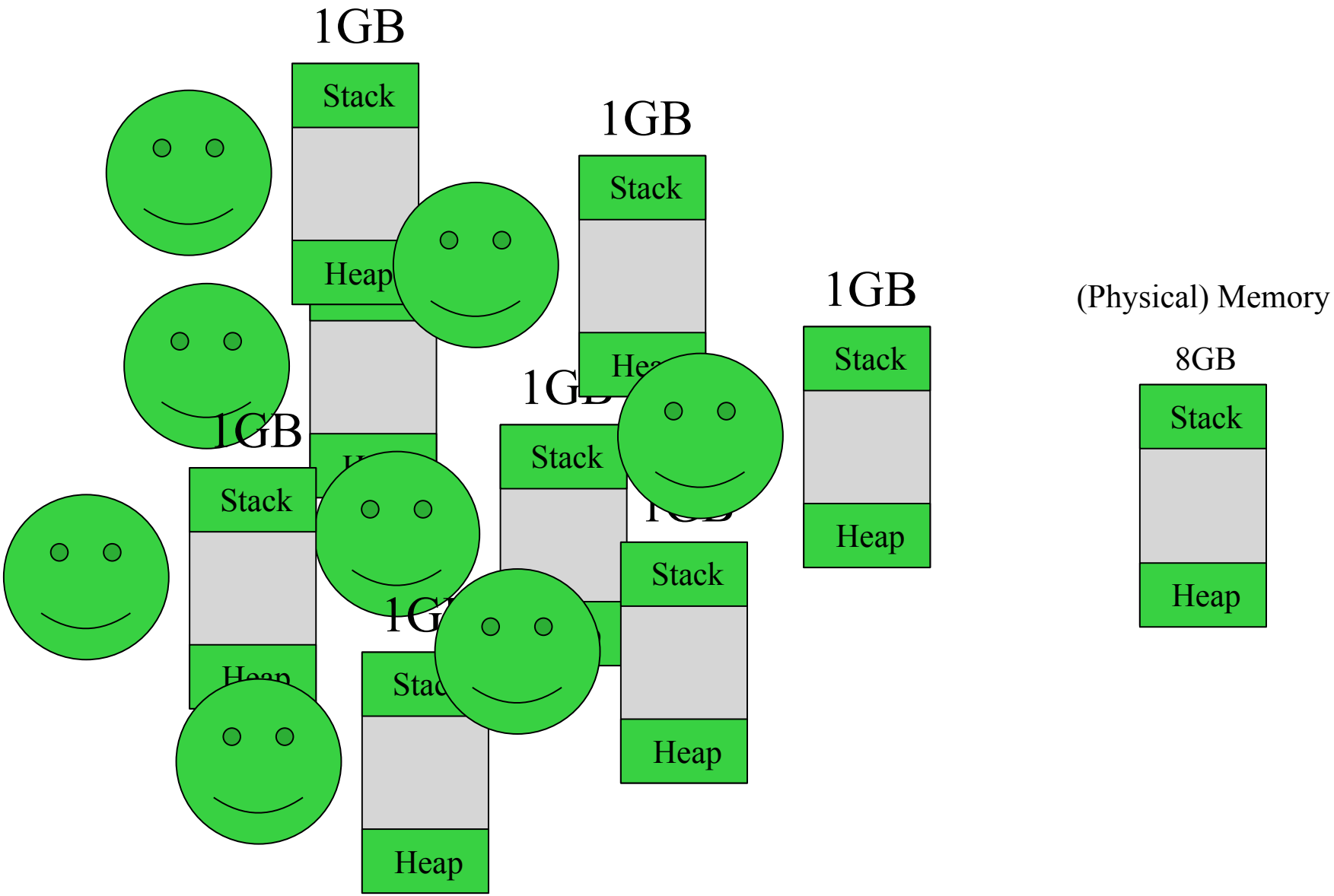


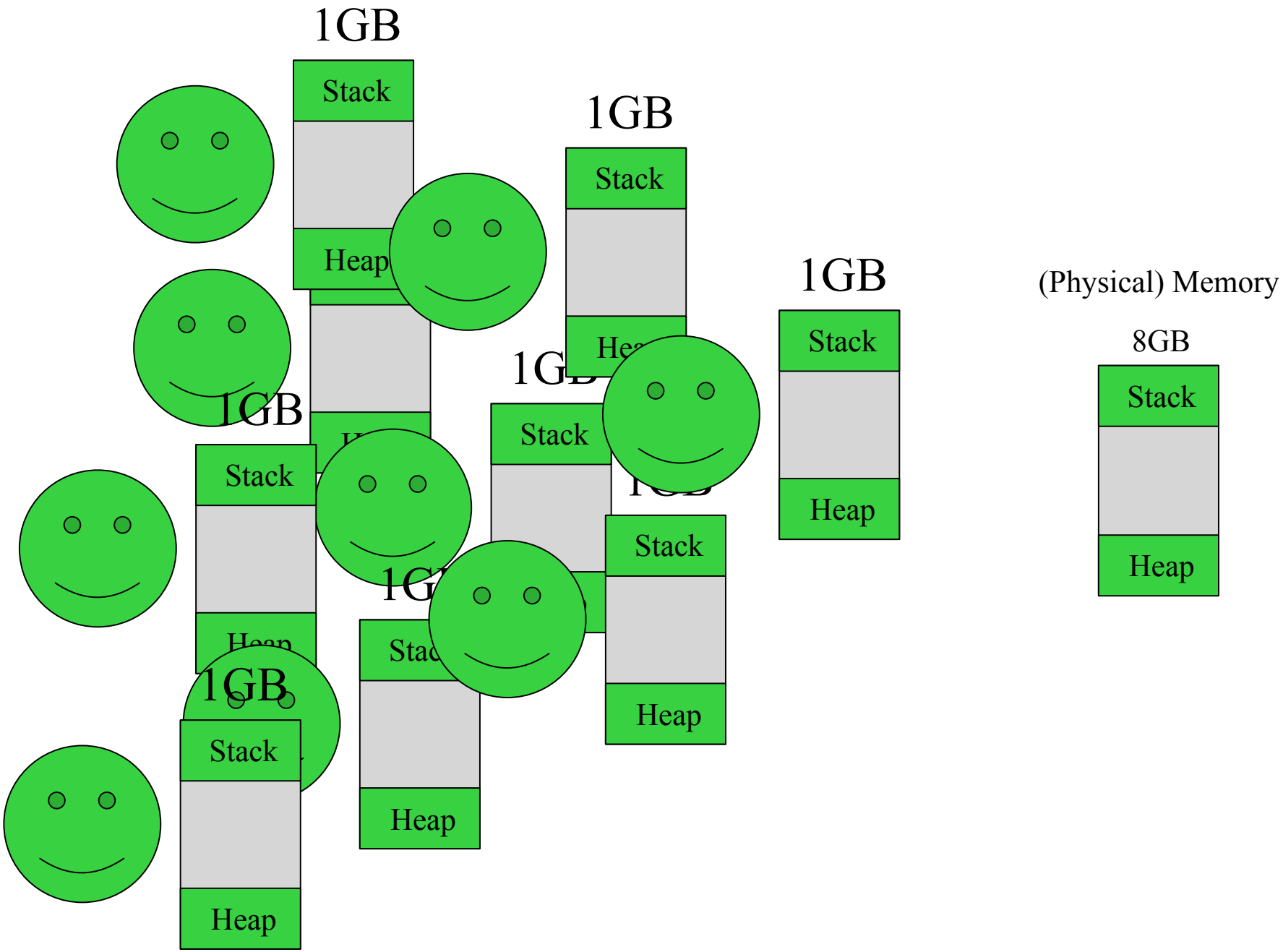


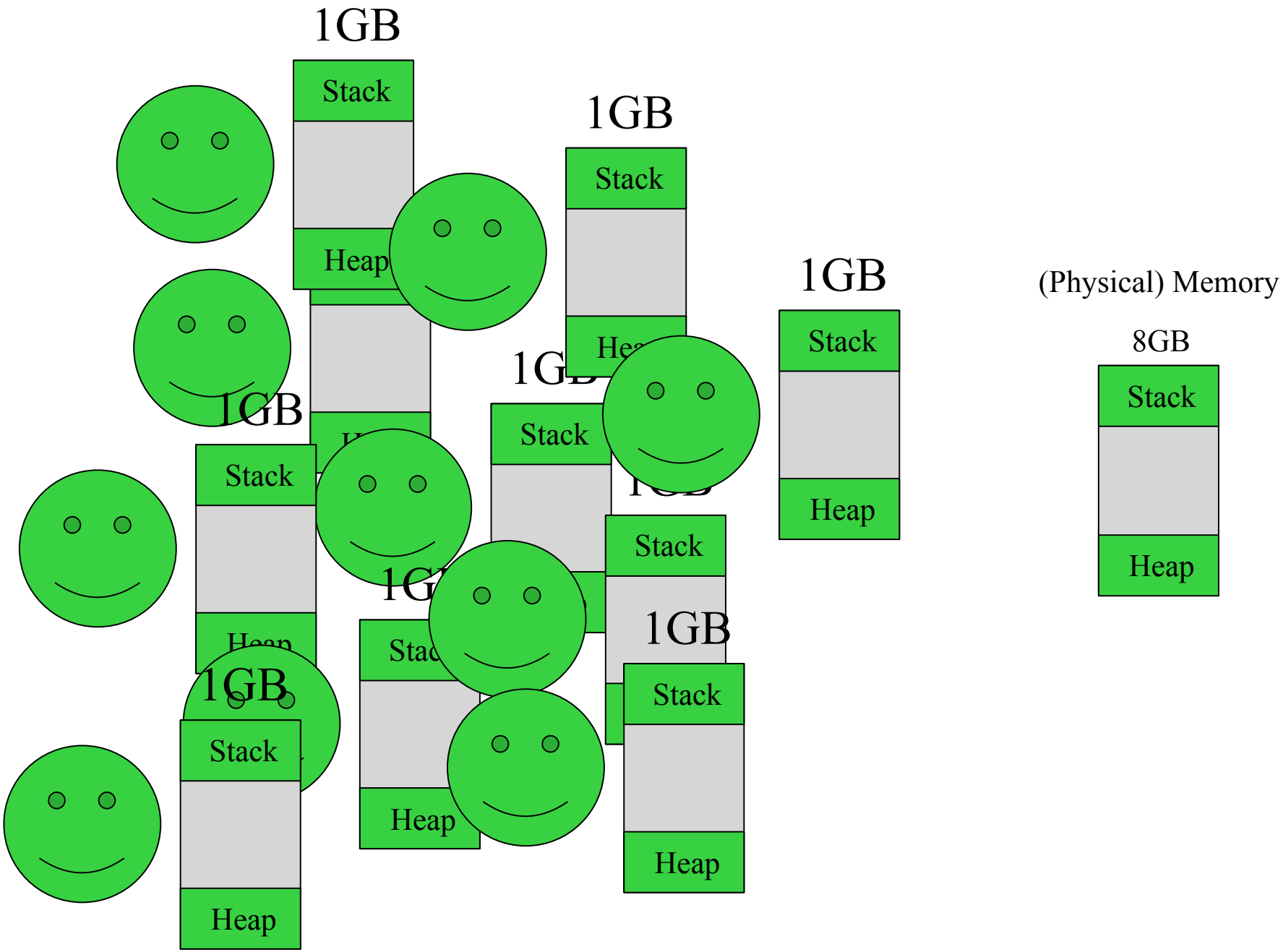
(Physical) Memory

8GB

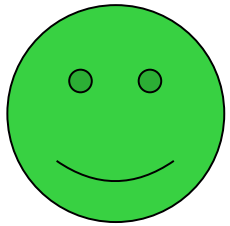



1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
(Physical) Memory
8GB
Stack
Heap


1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
(Physical) Memory
8GB
Stack
Heap

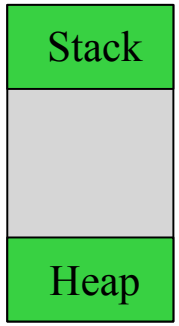

1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
1GB
Stack
Heap
(Physical) Memory
8GB
Stack
Heap

Learning to Play Well With Others

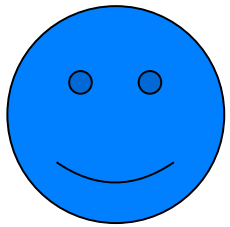


Virtual Memory

0x10000 (64KB)

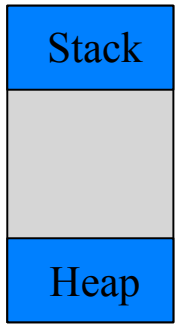


0x00000



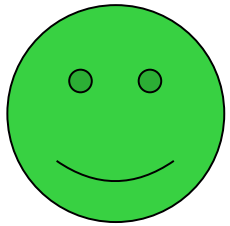
Virtual Memory

0x10000 (64KB)



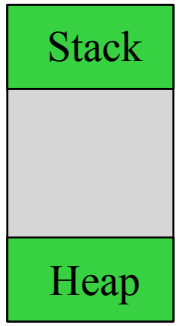
0x00000

Learning to Play Well With Others



Virtual Memory

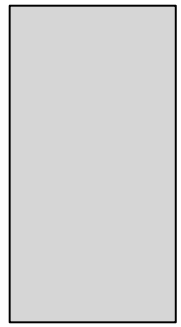
0x10000 (64KB)



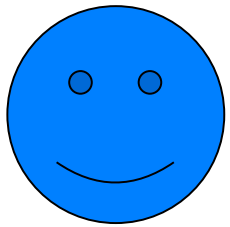
0x00000

Physical Memory

0x10000 (64KB)

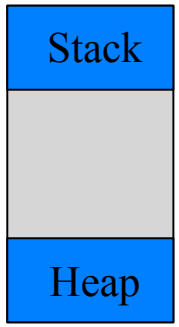


0x00000



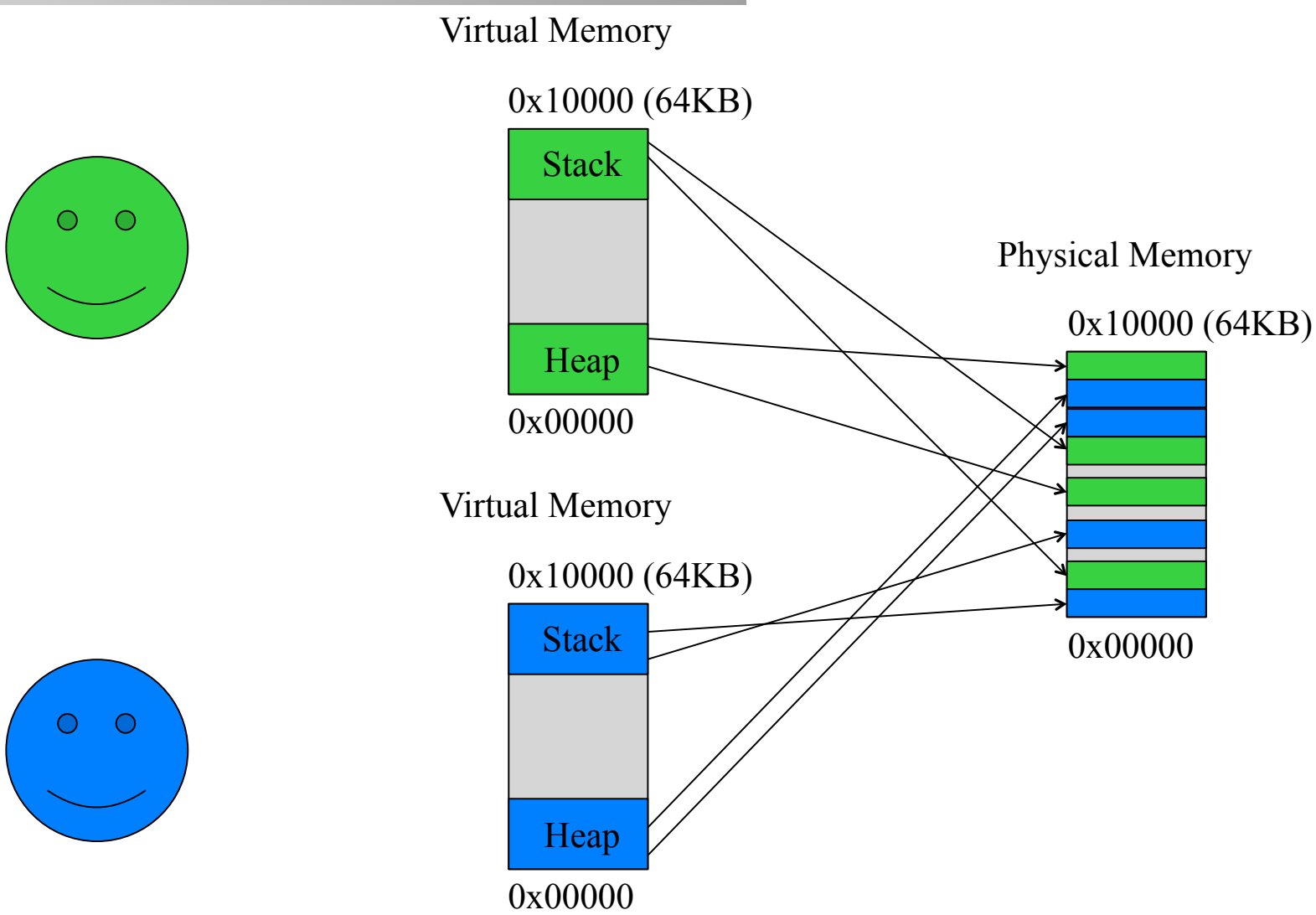
Virtual Memory

0x10000 (64KB)

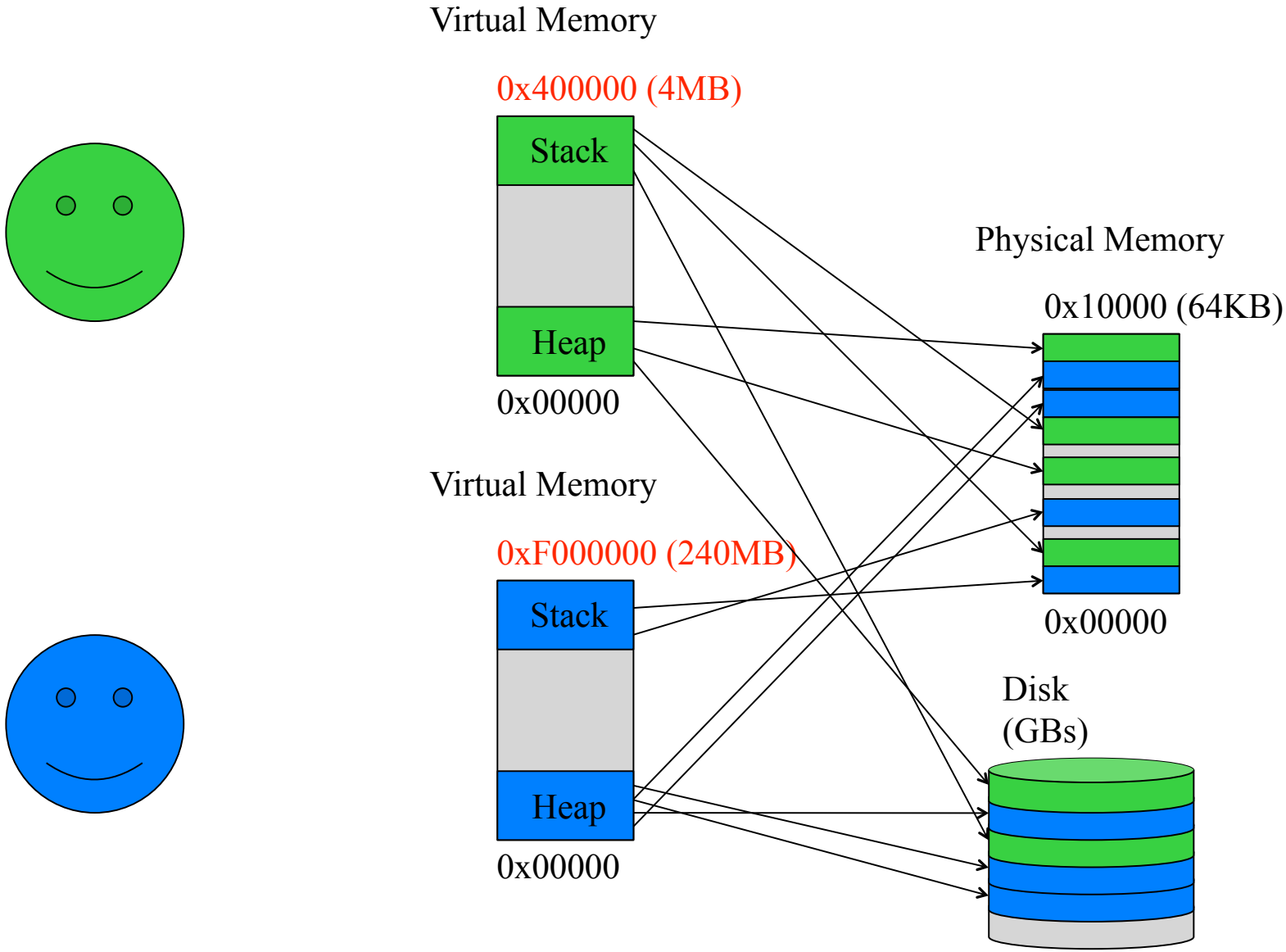


0x00000

Learning to Play Well With Others



Learning to Play Well With Others



Virtual Memory to the Rescue

More Memory Abstractions

In addition to the memory hierarchy, we will create a *virtual address space* further disconnecting the program from the physical memory.

We need a *virtual-to-physical mapping* to facilitate this.

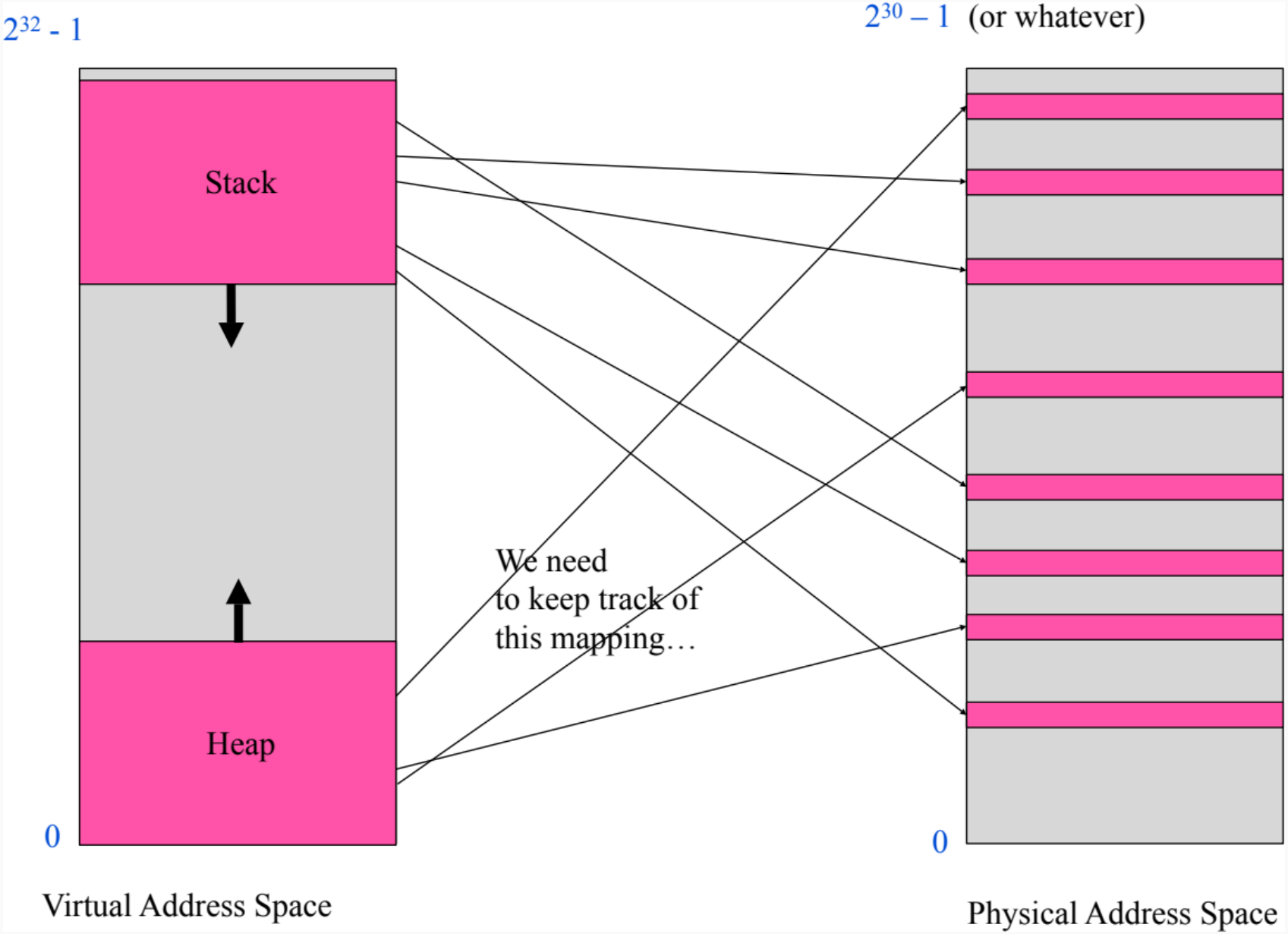
We will break both address spaces up into *pages*. Typically, 4KiB in size, although sometimes larger.

We use a *page table* to map between virtual and physical pages.

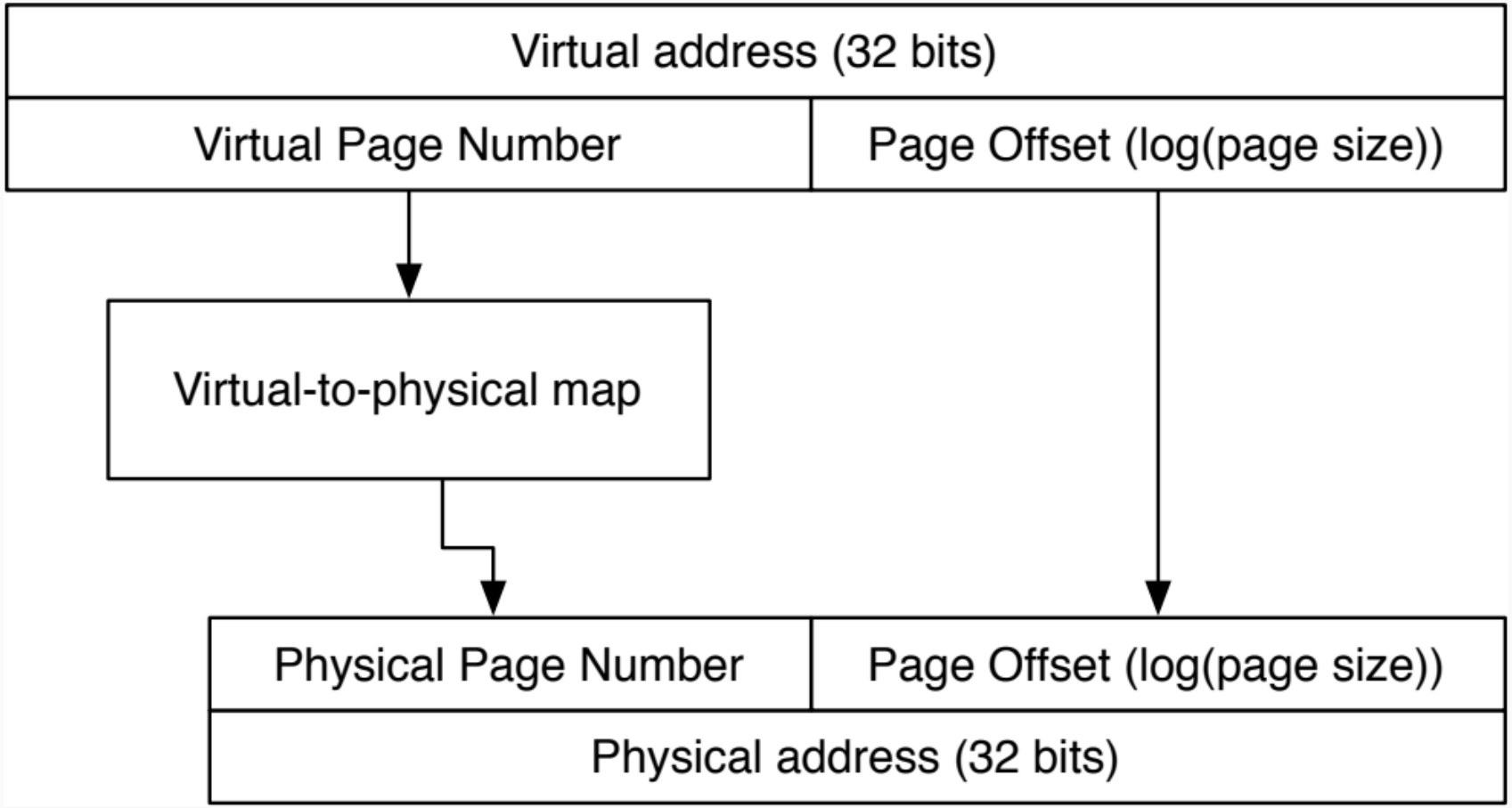
The processor generates *virtual* addresses which are translated via *address translation* into physical addresses.

Implementing Virtual Memory

Keeping Track of the Mapping



The Mapping Process



Two Problems with Virtual Memory

1. How do we store the map compactly?
2. How do we perform translation quickly?

1. Storing the Map Compactly: How Big is it?

- For a **32-bit address space**
  - 4 GiB of virtual addresses
  - 1 million pages
  - Each entry is 4 bytes (a single 32-bit physical address)
  - **4 MiB of map**
- For a **64-bit address space**
  - 16 exabytes of virtual address space
  - 4 peta-pages
  - Each entry is 8 bytes (a single 64-bit physical address)
  - **64 PiB of map!**
  - For context, all of Wikimedia Commons (including images, videos, etc.) was approximately 200 TiB of data in 2018.

1. Storing the Map Compactly: Shrinking the Map

Clearly we need a way to shrink the map!

Try just storing the entries that matter (enough for your physical address space).

- A 64 GiB, 64-bit machine
- 16 MiB pages $\rightarrow$128 MiB of map!
- Still pretty big (even L3 cache couldn't contain the entire page table)!

What we really need is a *sparse* representation.

- The OS allocates stuff all over the place.
- For security, convenience, or caching optimizations
- Example: stack is at the "top" of memory and the heap is at the "bottom'.

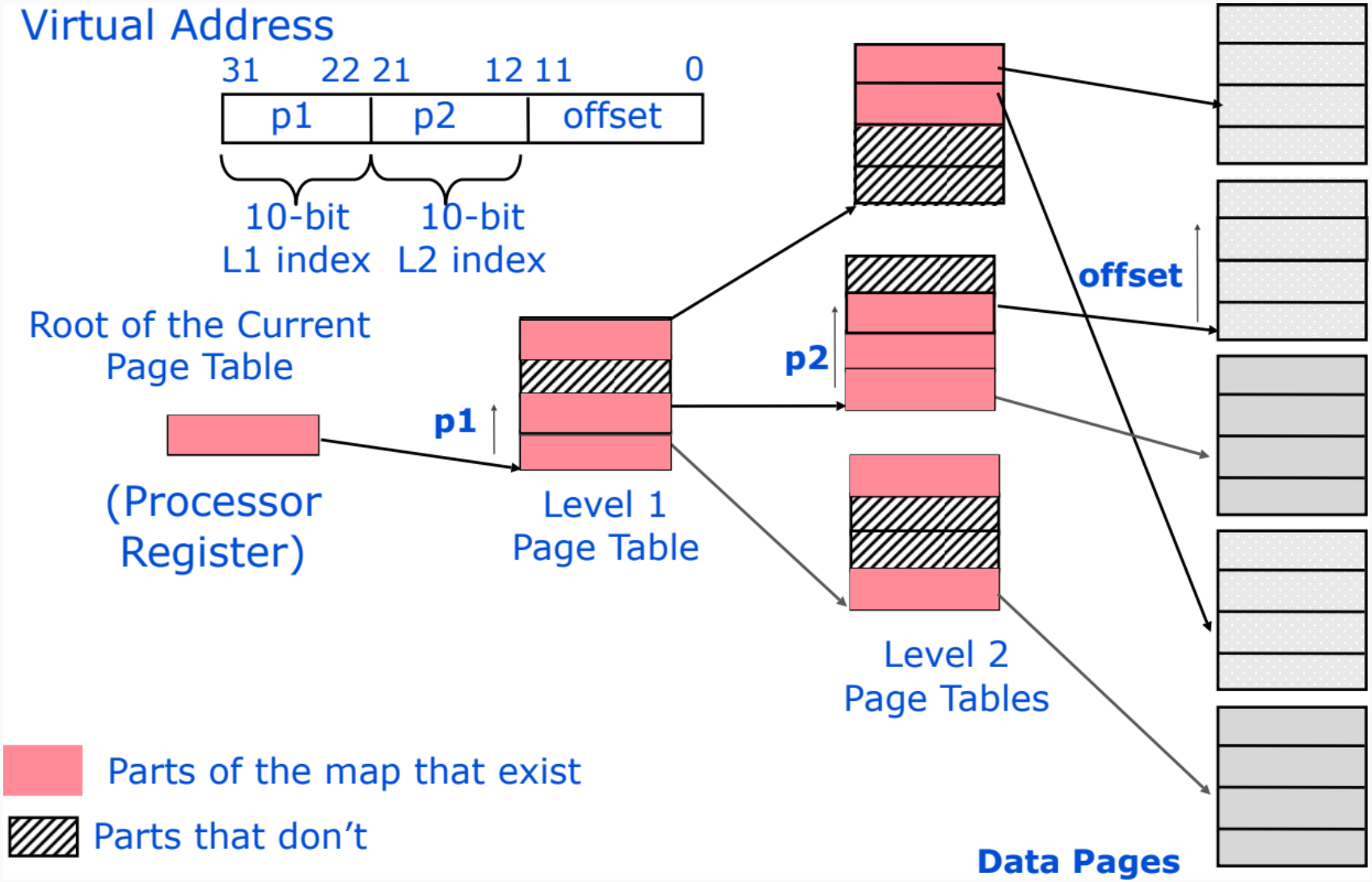
How do you represent this sparse map?

1. Storing the Map Compactly: Hierarchical Page Tables

We can use a tree to hold the page table!

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

1. Storing the Map Compactly: Hierarchical Page Tables



2. Performing Translation Quickly: Why?

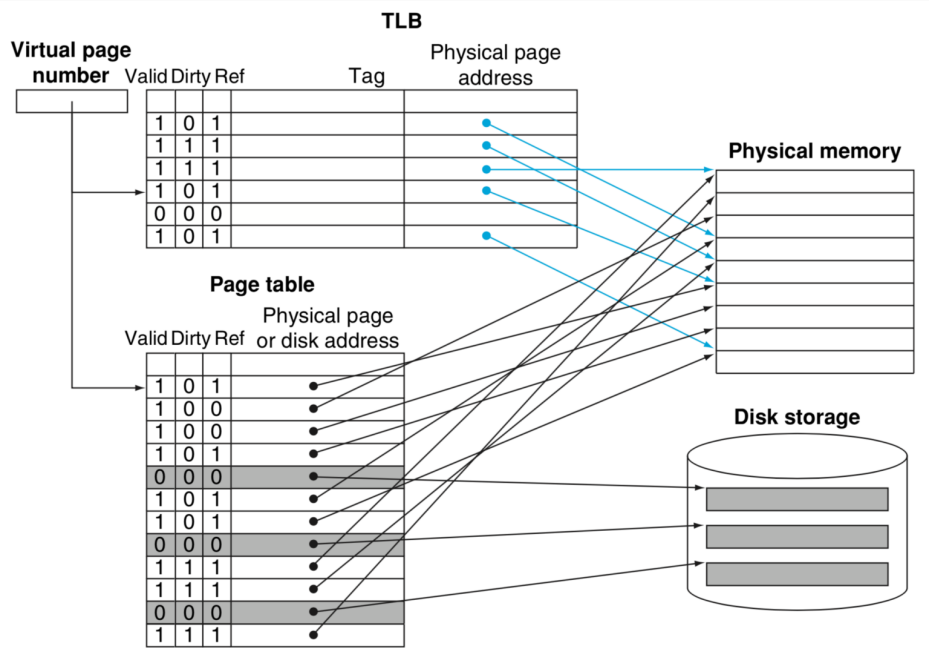
- Address translation has to happen for each memory access.
- This puts it squarely on the *critical path* for memory operations (which are already slow).

2. Performing Translation Quickly: Page Table is Too Slow!

- We could walk the page table on every memory access.
- Result: every load or store operation requires an additional 3-4 loads to walk the page table!
- This is an unacceptable performance hit.

2. Performing Translation Quickly: Solution: TLBs!

- We have a lot of data (the page table) that we want to access very quickly (in a single clock cycle).
- **Solution: add a cache!**
- This cache holds page mappings and is called the *translation lookaside buffer (TLB)*



http://thebeardsage.com/virtual-memory-translation-lookaside-buffer-tlb/

2. Performing Translation Quickly: TLB Details

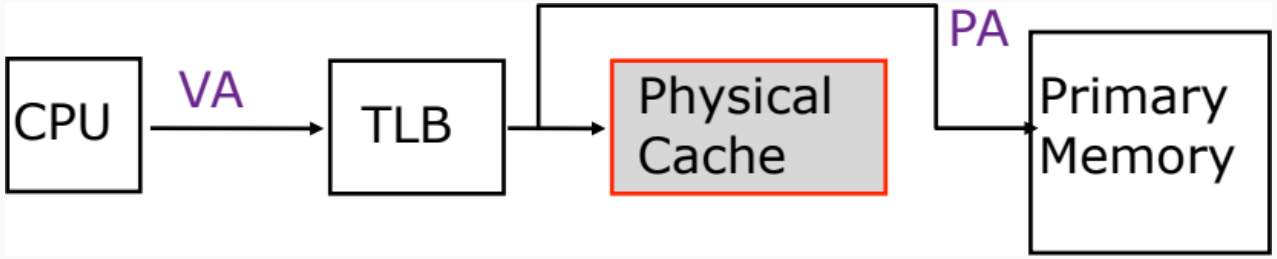
- TLBs are small (16-512 entries), highly associative (often fully associative) caches for page table entries.
- Since it is a cache, there is a possibility of a *TLB miss* which can be expensive.
  - To make them less expensive, there are *hardware page table walkers* which are specialized state machines which can load page table entries into the TLB without OS intervention.
  - This means that the page table format is now part of the architecture.
  - Typically, the OS can disable the walker and implement its own format.

When to Translate?

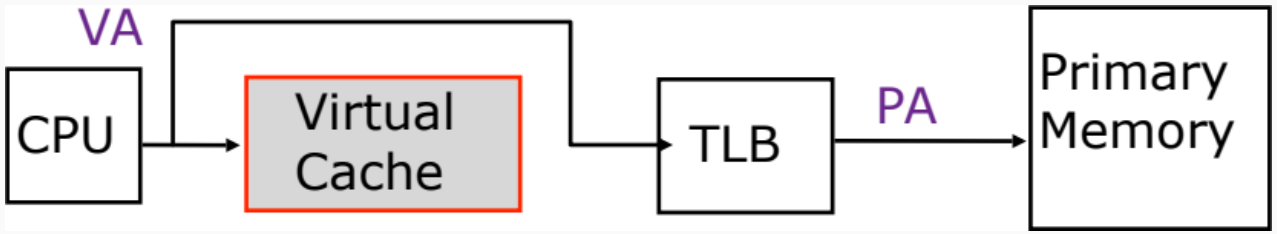
Should we Translate Before or After Cache Access?

If we translate *before* we go to the cache we have a *physical cache*. That is, a cache that works on physical addresses.

Critical Path: TLB access time + Cache access time



Alternatively, we could translate *after* the cache. Now translation is only required on miss! This would be a *virtual cache* since it caches according to virtual addresses.



Dangers of Virtual Cache: Context Switches

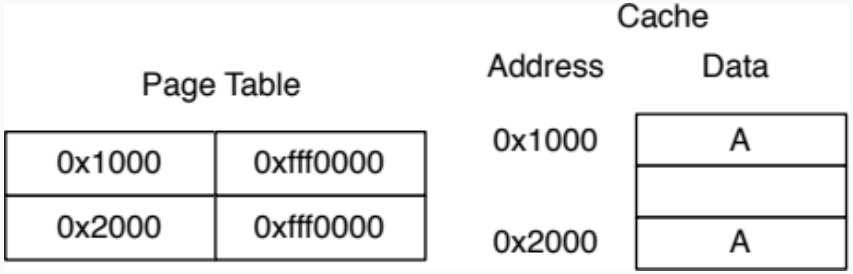
Consider the following situation:

1. Process A is running. It issues a memory request to address `0x10000`.
   - It is a miss, and `0x10000` is brought into the virtual cache.
2. A context switch occurs
3. Process B starts running. It issues a request for `0x10000`.
   - **Will Process B get the right data?**
   - **No! We must flush virtual caches on context switch.**

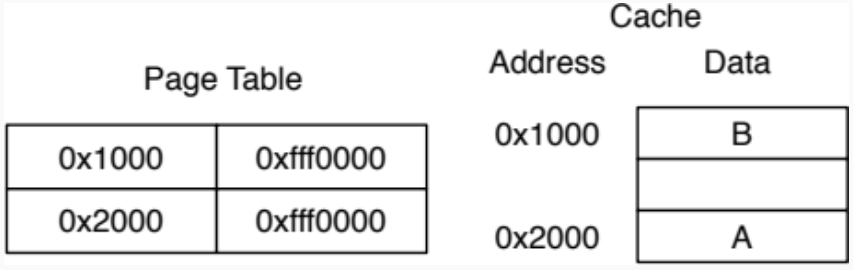
Dangers of Virtual Cache: Aliasing

There is no rule that says each virtual address maps to different physical address.

- When this occurs, it is called "aliasing".
- **Example:**



- Store `B` to `0x1000`

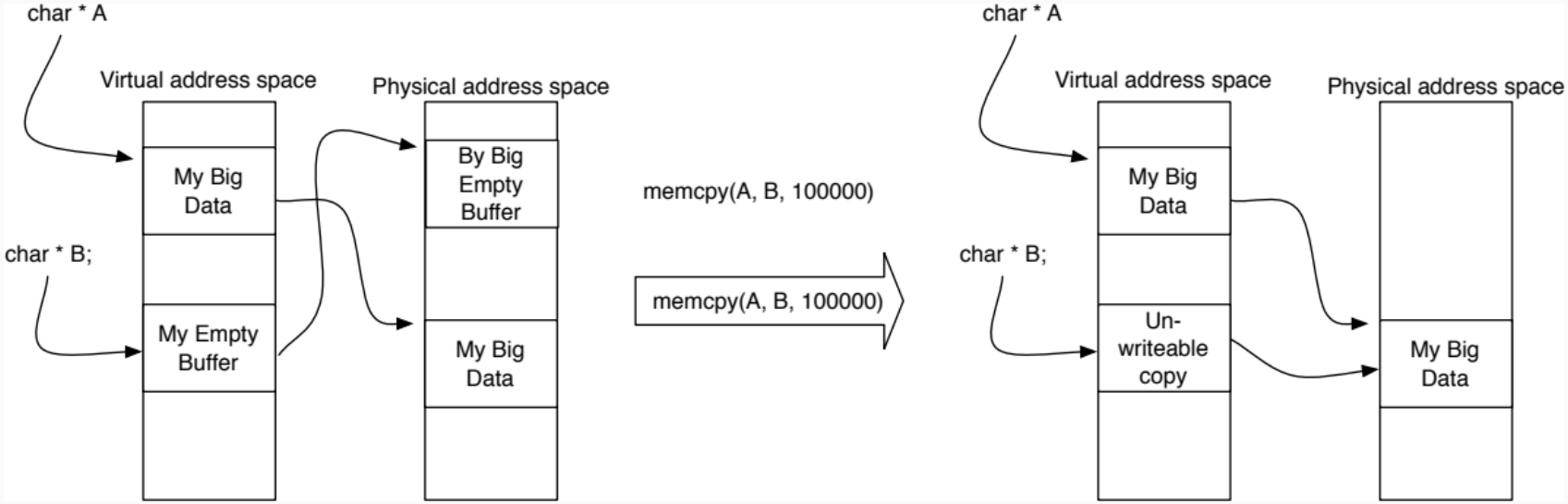


- Now, a load from `0x2000` will return the wrong value!

Dangers of Virtual Cache: Aliasing: Why It's Useful

Aliases can be useful for implementing *copy-on-write* memory.

Consider a situation where you have to copy a large chunk of memory: `memcpy(A, B, 100000)`



- Two virtual addresses pointing to the same physical address!
- Adjusting the page table is much faster for large copies.
- The initial copy is free, and the OS will catch attempts to write to the copy, and perform the actual copy lazily.
- There are also system calls that let you do this arbitrarily.