Advance Caching

Today

- “quiz 5” recap
- quiz 6 recap
- advanced caching
- Hand a bunch of stuff back.

Speeding up Memory

- ET = IC * CPI * CT
- CPI = noMemCPI * noMem% + memCPI*mem%
- memCPI = hit% * hitTime + miss%*missTime

- Miss times:
  - L1 -- 20-100s of cycles
  - L2 -- 100s of cycles

- How do we lower the miss rate?

Know Thy Enemy

- Misses happen for different reasons
- The three C's (types of cache misses)
  - Compulsory: The program has never requested this data before. A miss is mostly unavoidable.
  - Conflict: The program has seen this data, but it was evicted by another piece of data that mapped to the same "set" (or cache line in a direct mapped cache)
  - Capacity: The program is actively using more data than the cache can hold.

- Different techniques target different C's

Compulsory Misses

- Compulsory misses are difficult to avoid
- Caches are effectively a guess about what data the processor will need.
- One technique: Prefetching (240A to learn more)

```
for(i = 0;i < 100; i++) {
   sum += data[i];
}
```

- In this case, the processor could identify the pattern and proactively "prefetch" data program will ask for.
- Current machines do this alot...
- Keep track of `delta= thisAddress - lastAddress`, it's consistent, start fetching `thisAddress + delta`.

Reducing Compulsory Misses

- Increase cache line size so the processor requests bigger chunks of memory.
- For a constant cache capacity, this reduces the number of lines.
- This only works if there is good spatial locality, otherwise you are bringing in data you don't need.
- If you are asking small bits of data all over the place (i.e., no spatial locality) this will hurt performance
- But it will help in cases like this

```
for(i = 0;i < 1000000; i++) {
   sum += data[i];
}
```

One miss per cache line worth of data

Reducing Compulsory Misses

- HW Prefetching

```
for(i = 0;i < 1000000; i++) {
   sum += data[i];
}
```

- In this case, the processor could identify the pattern and proactively "prefetch" data program will ask for.
- Current machines do this alot...
- Keep track of `delta= thisAddress - lastAddress`, it's consistent, start fetching `thisAddress + delta`.

Reducing Compulsory Misses

- Software prefetching
- Use register $zero!

```
for(i = 0;i < 1000000; i++) {
   sum += data[i];
   “load data[i+16] into $zero”
}
```

For exactly this reason, loads to $zero never fail (i.e., you can load from any address into $zero without fear)

Conflict Misses

- Conflict misses occur when the data we need was in the cache previously but got evicted.
- Evictions occur because:
  - Direct mapped: Another request mapped to the same cache line
  - Associative: Too many other requests mapped to the same cache line (N + 1 if N is the associativity)

```
while(1) {
  for(i = 0;i < 1024; i+=4096) {
     sum += data[i];
  } // Assume a 4 KB Cache
}
```

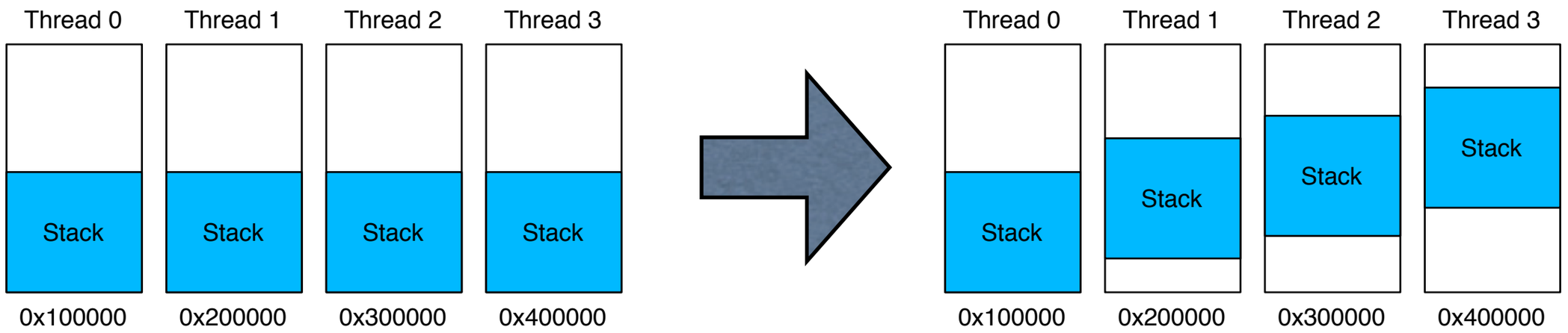
-

Reducing Conflict Misses

- Conflict misses occur because too much data maps to the same "set"
  - Increase the number of sets (i.e., cache capacity)
  - Increase the size of the sets (i.e., the associativity)
- The compiler and OS can help here too

Colliding Threads and Data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
  - Threads often run the same code in the same way
  - This means that thread stacks will end up occupying the same parts of the cache.
  - Randomize the base of each threads stack.
- Large data structures (e.g., arrays) are also often aligned. Randomizing malloc() can help here.



Capacity Misses

- Capacity misses occur because the processor is trying to access too much data.
  - Working set: The data that is currently important to the program.
  - If the working set is bigger than the cache, you are going to miss frequently.
- Capacity misses are bit hard to measure
  - Easiest definition: non-compulsory miss rate in an equivalently-sized fully-associative cache.
  - Intuition: Take away the compulsory misses and the conflict misses, and what you have left are the capacity misses.

Reducing Capacity Misses

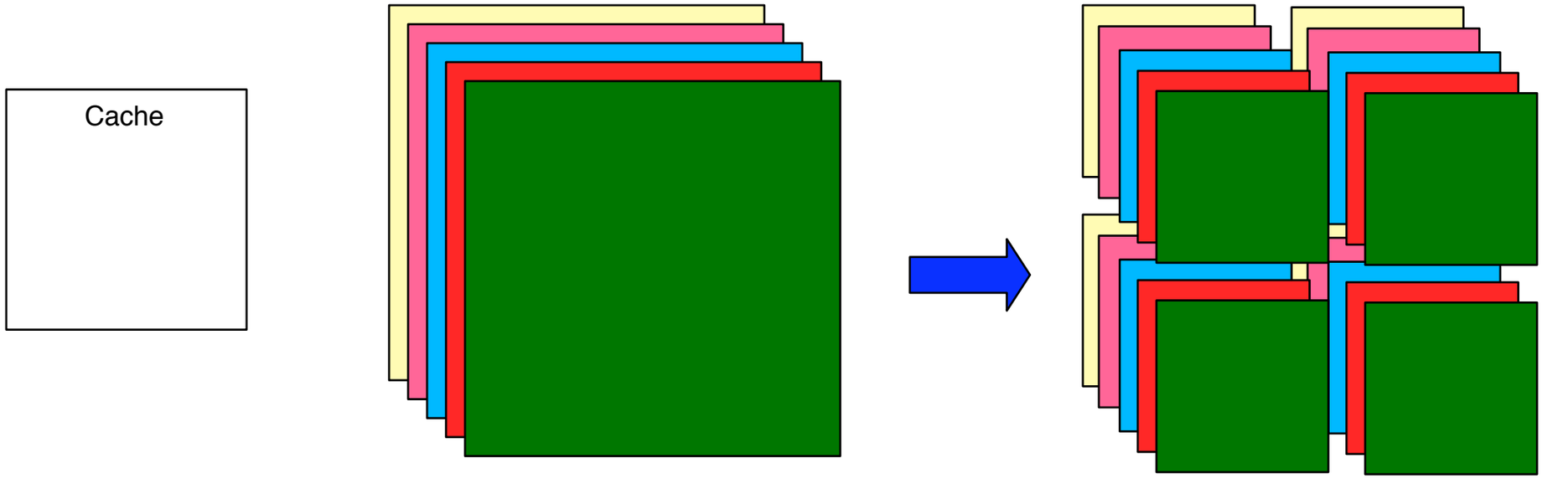
- Increase capacity!
- More associativity or more associative "sets"
- Costs area and makes the cache slower.
- Cache hierarchies do this implicitly already:
  - if the working set "falls out" of the L1, you start using the L2.
  - Poof! you have a bigger, slower cache.
- In practice, you make the L1 as big as you can within your cycle time and the L2 and/or L3 as big as you can while keeping it on chip.

Reducing Capacity Misses: The compiler

- The key to capacity misses is the working set
- How a program performs operations has a large impact on its working set.

Reducing Capacity Misses: The compiler

- Tiling
  - We need to makes several passes over a large array
  - Doing each pass in turn will "blow out" our cache
  - "blocking" or "tiling" the loops will prevent the blow out
  - Whether this is possible depends on the structure of the loop
- You can tile hierarchically, to fit into each level of the memory hierarchy.



Increasing Locality in the Compiler or Application

- Live Demo... The Return!

Capacity Misses in Action

- Live Demo... The return! Part Deux!

Cache optimization in the real world: Core 2 duo vs AMD Opteron (via simulation)

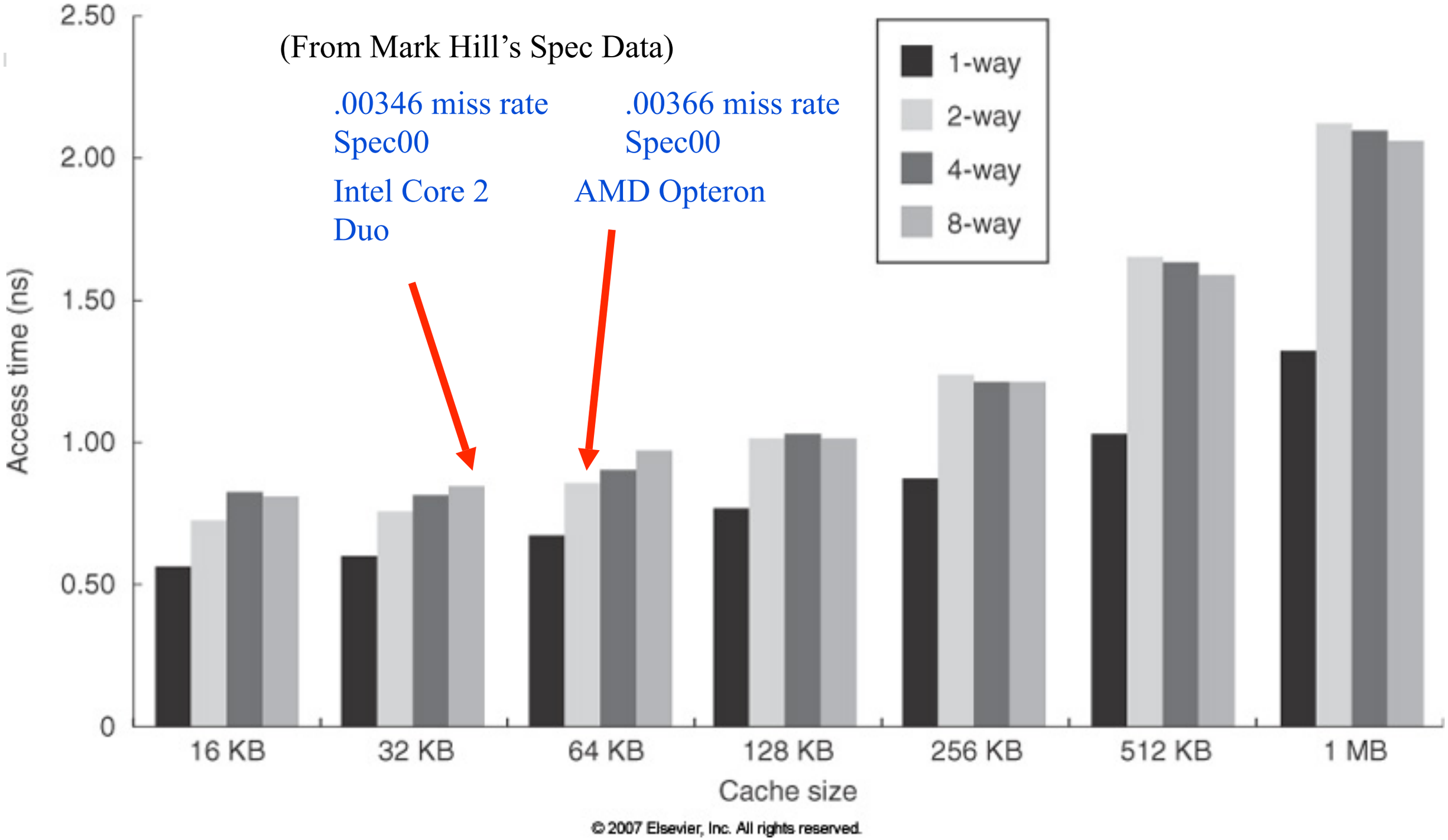
(From Mark Hill's Spec Data)

.00346 miss rate Spec00

Intel Core 2 Duo

.00366 miss rate Spec00

AMD Opteron



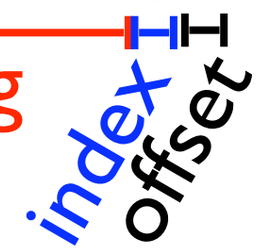
Intel gets the same performance for less capacity because they have better SRAM Technology: they can build an 8-way associative L1. AMD seems not to be able to.

A Simple Example

- Consider a direct mapped cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010

A Simple Example

- a direct mapped cache with 16 blocks, a block size of 16 bytes
  - 16 = 2^4 : 4 bits are used for the index
  - 16 = 2^4 : 4 bits are used for the byte offset
  - The tag is 32 - (4 + 4) = 24 bits
  - For example: 0x80000010



A Simple Example

	valid	tag	data
0	1	800000	
1	1	**800000**	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010 miss: conflict

0x80000018 hit!

A Simple Example: Increased Cache line Size

- Consider a direct mapped cache with 8 blocks, a block size of 32 bytes, and the application repeat the following memory access sequence:
  - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010

A Simple Example

- a direct mapped cache with 8 blocks, a block size of 32 bytes
  - 8 = 2^3 : 3 bits are used for the index
  - 32 = 2^5 : 5 bits are used for the byte offset
  - The tag is 32 - (3 + 5) = 24 bits
  - For example: 0x80000010 =
    - 01110000000000000000000000010000

A Simple Example

	valid	tag	data
0	1	800000	
1			
2			
3			
4			
5			
6			
7			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 miss: conflict

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

A Simple Example: Increased Associativity

- Consider a 2-way set associative cache with 8 blocks, a block size of 32 bytes, and the application repeat the following memory access sequence:
  - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010

A Simple Example

- a 2-way set-associative cache with 8 blocks, a block size of 32 bytes
  - The cache has 8/2 = 4 sets: 2 bits are used for the index
  - 32 = 2^5 : 5 bits are used for the byte offset
  - The tag is 32 - (2+ 5) = 25 bits
  - For example: 0x80000010 =
    - 01110000000000000000000010000

A Simple Example

	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

0x30000010 miss: compulsory

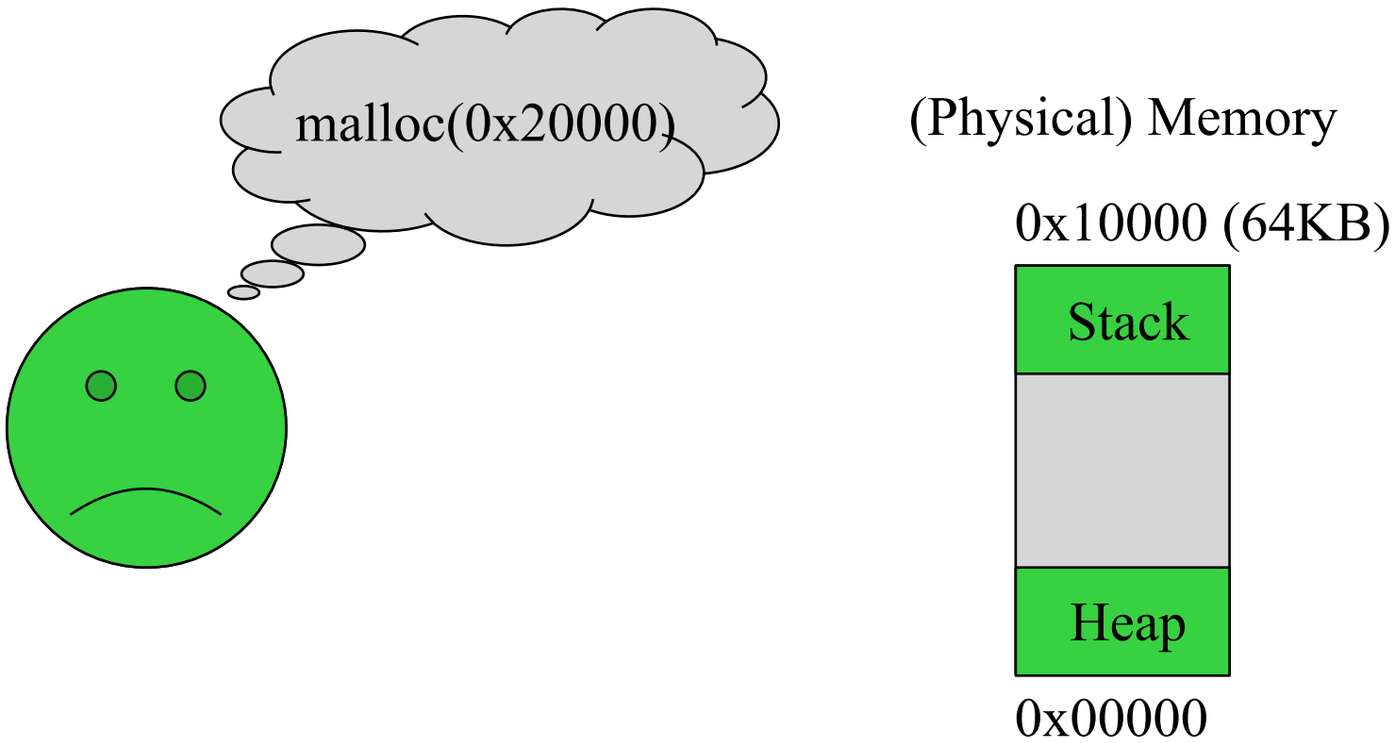
0x80000000 hit!

0x80000008 hit!

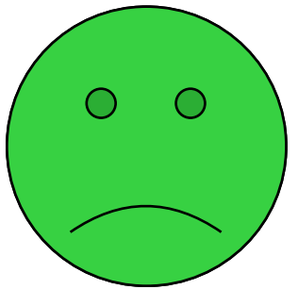
0x80000010 hit!

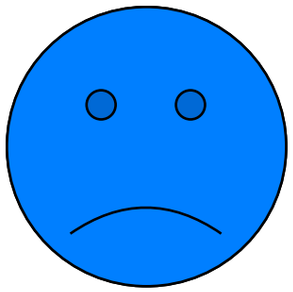
0x80000018 hit!

Learning to Play Well With Others



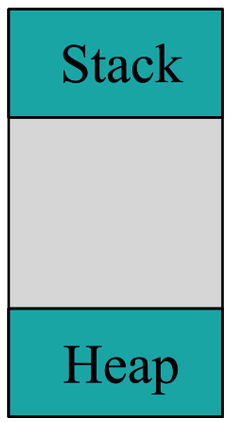
Learning to Play Well With Others





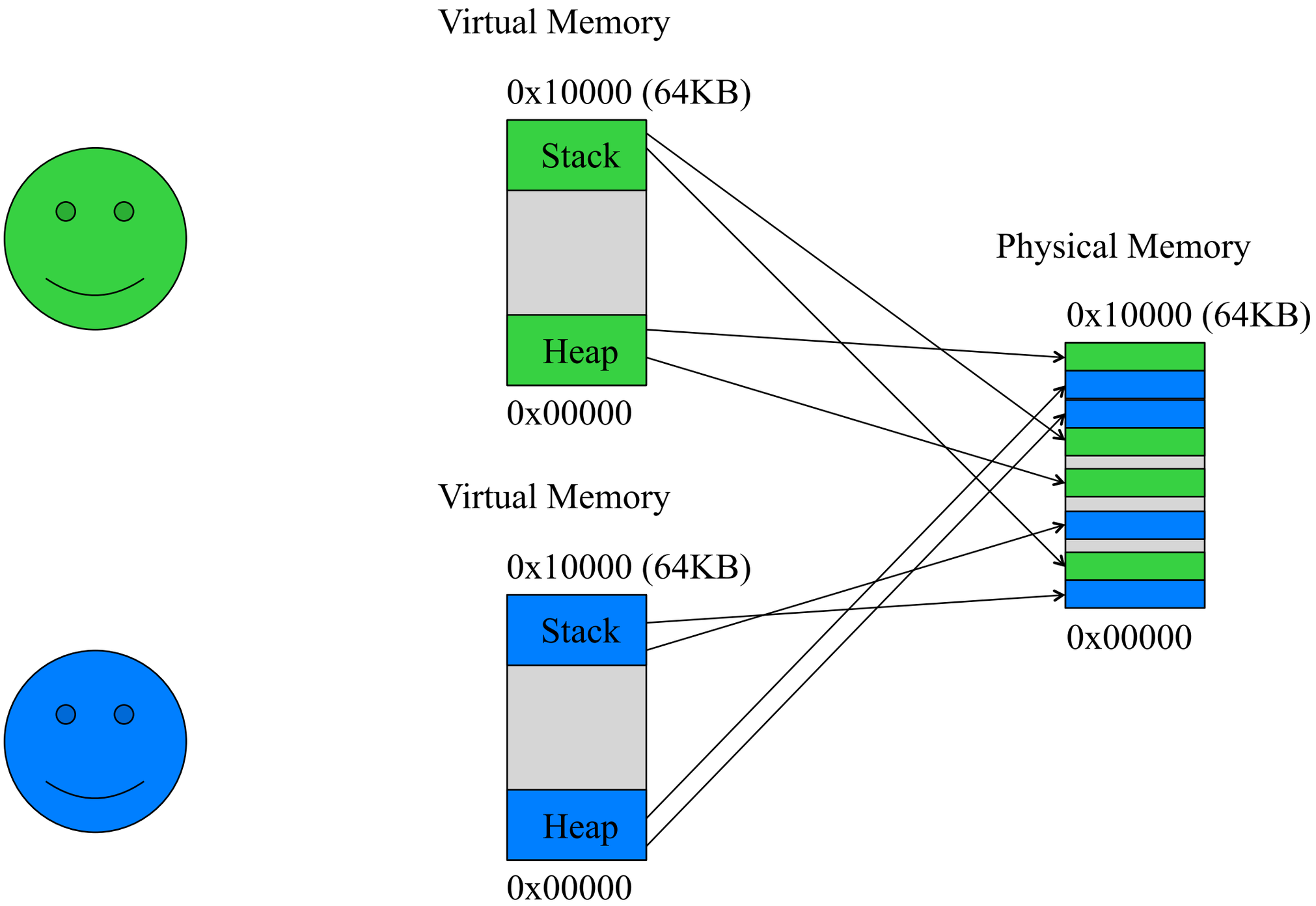
(Physical) Memory

0x10000 (64KB)

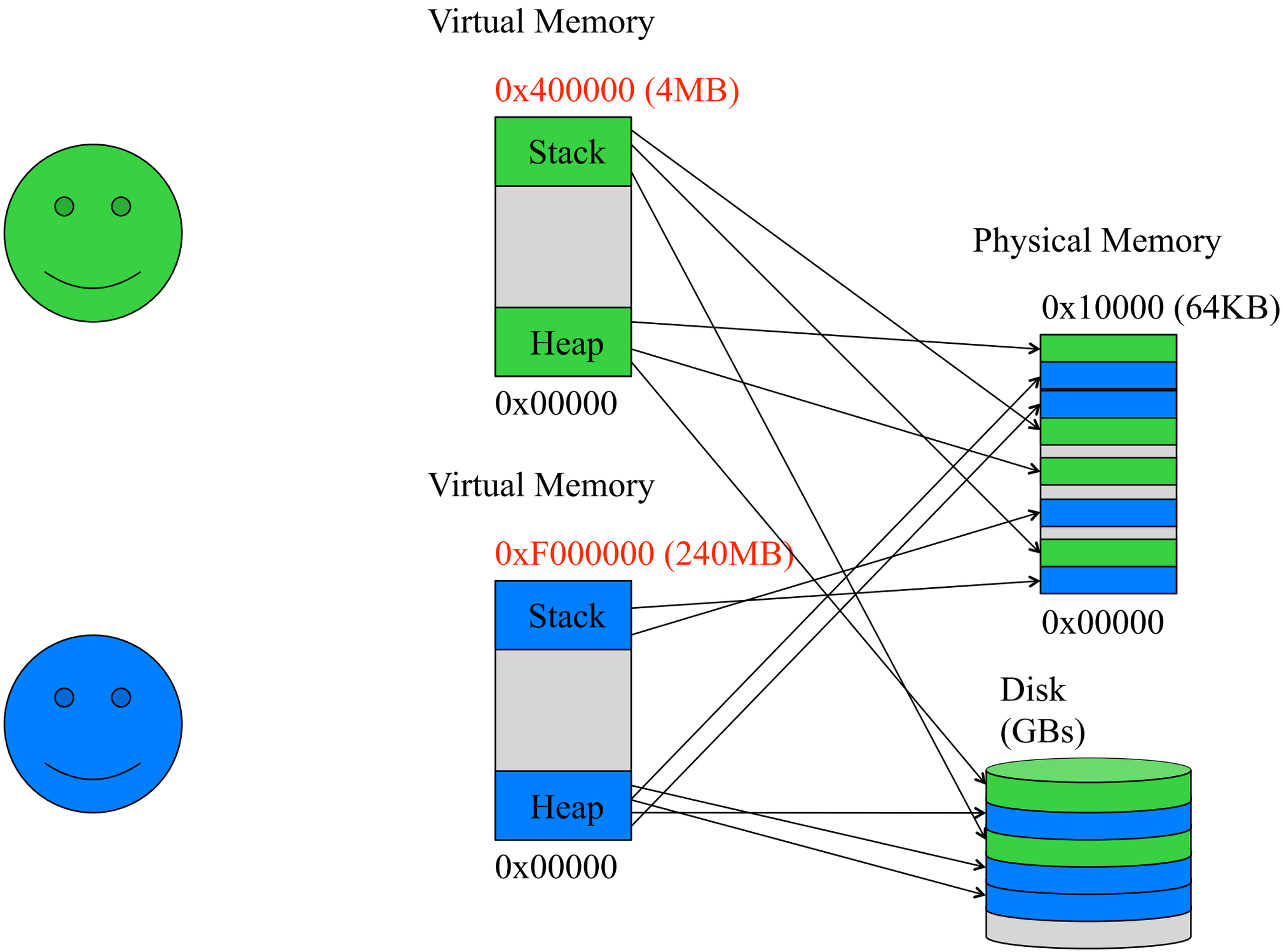


0x00000

Learning to Play Well With Others



Learning to Play Well With Others



Virtual Memory

- The games we play with addresses and the memory behind them

Address translation

- decouple the names of memory locations and their physical locations
- maintains the address space ABSTRACTION
- enable sharing of physical memory (different addresses for same objects)
  - shared libraries, fork, copy-on-write, etc

Specify memory + caching behavior

- protection bits (execute disable, read-only, write-only, etc)
- no caching (e.g., memory mapped I/O devices)
- write through (video memory)
- write back (standard)

Demand paging

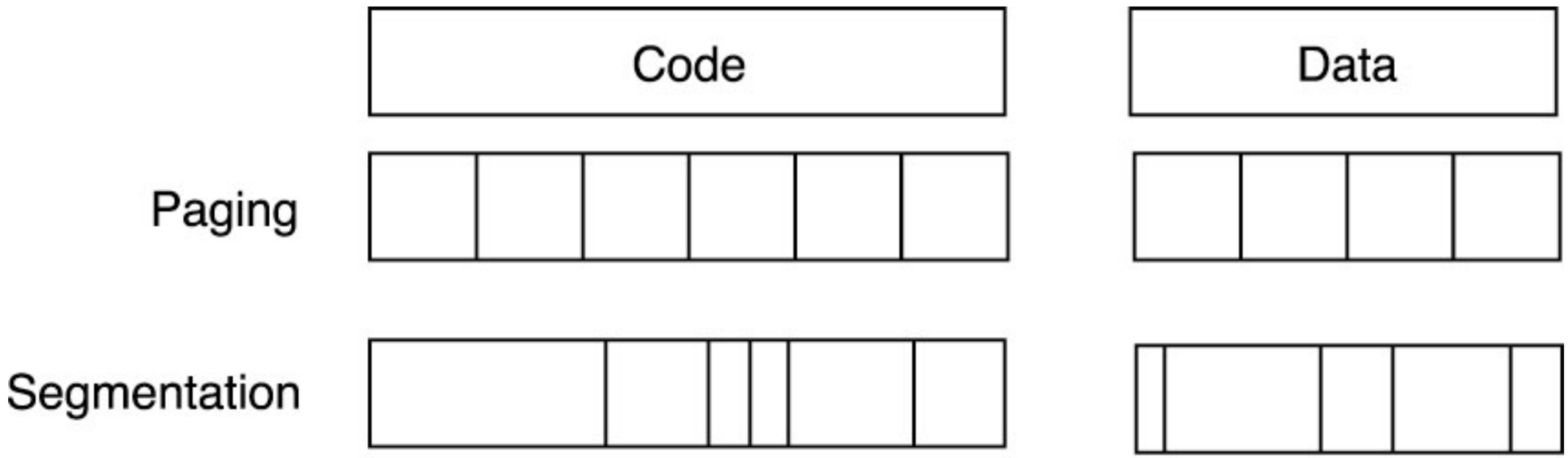
- use disk (flash?) to provide more memory
- cache memory ops/sec: 1,000,000,000 (1 ns)
- dram memory ops/sec: 20,000,000 (50 ns)
- disk memory ops/sec: 100 (10 ms)
  - demand paging to disk is only effective if you basically never use it
    not really the additional level of memory hierarchy it is billed to be

Paged vs Segmented Virtual Memory

- Paged Virtual Memory
  - memory divided into fixed sized pages

    → each page has a base physical address
- Segmented Virtual Memory
  - memory is divided into variable length segments

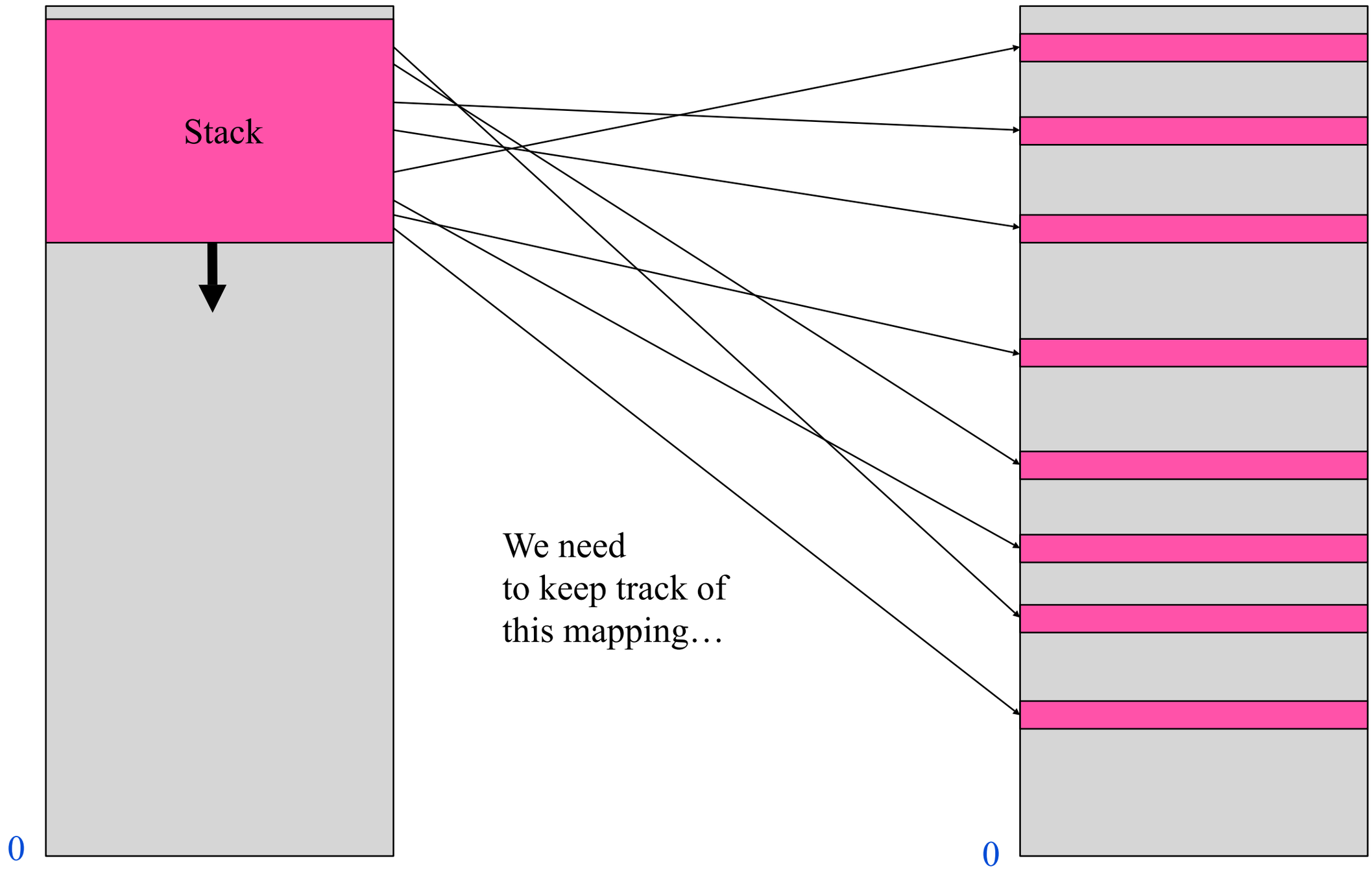
    → each segment has a base physical address + length



Implementing Virtual Memory

$2^{64} - 1$

$2^{40} - 1$ (or whatever)



We need
to keep track of
this mapping…

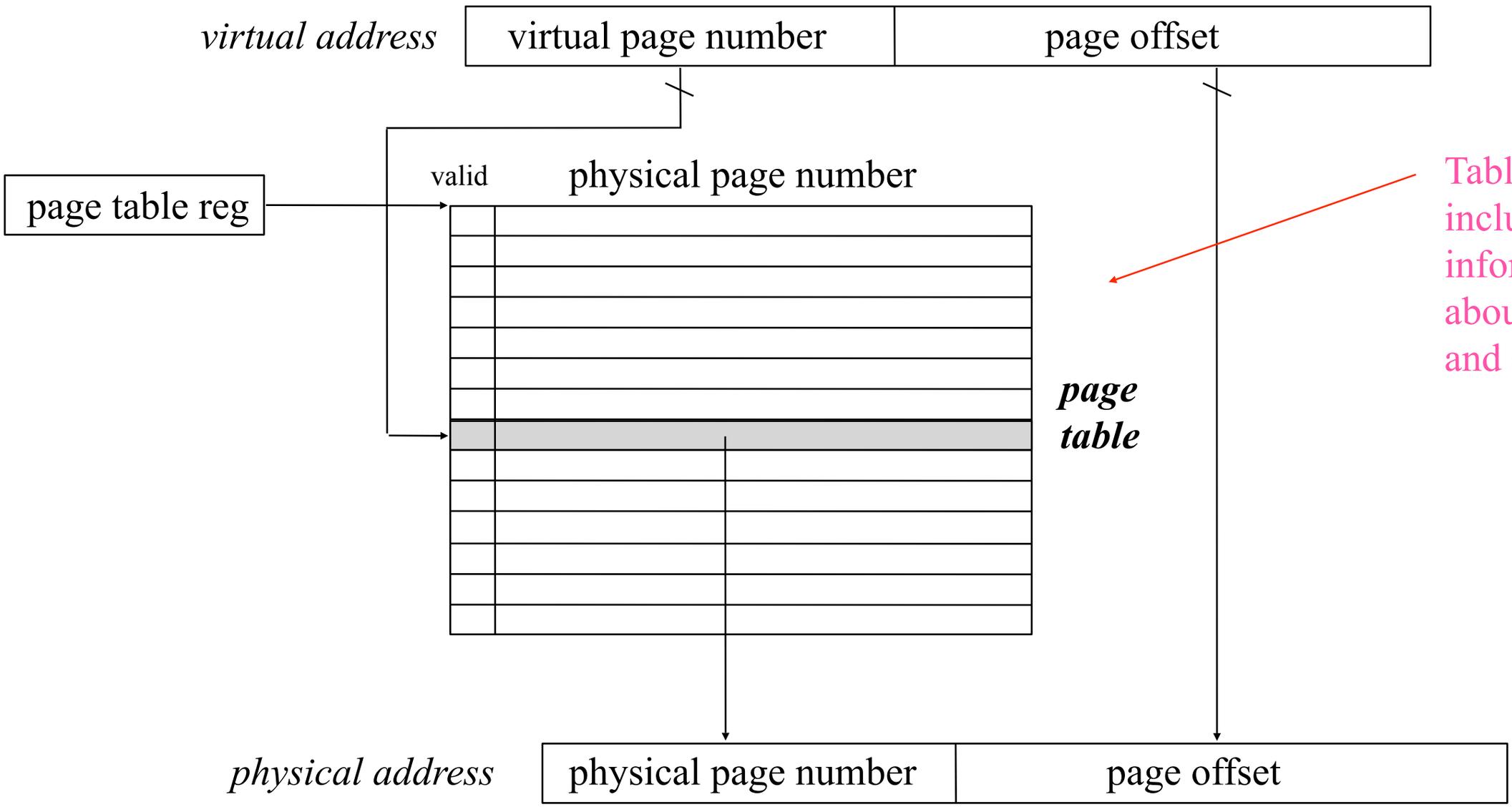
0

Virtual Address Space

0

Physical Address Space

Address translation via Paging



- all page mappings are in the page table, so hit/miss is determined solely by the valid bit (i.e., no tag)

Paging Implementation

Two issues; somewhat orthogonal

- specifying the mapping with relatively little space
  - the larger the minimum page size, the lower the overhead
    1 KB, 4 KB (very common), 32 KB, 1 MB, 4 MB …
  - typically some sort of hierarchical page table (if in hardware)
    or OS-dependent data structure (in software)

- making the mapping fast
  - TLB
    - small chip-resident cache of mappings from virtual to physical addresses

Hierarchical Page Table