CS 475
Operating Systems

Est. 1888
UNIVERSITY of
PUGET
SOUND

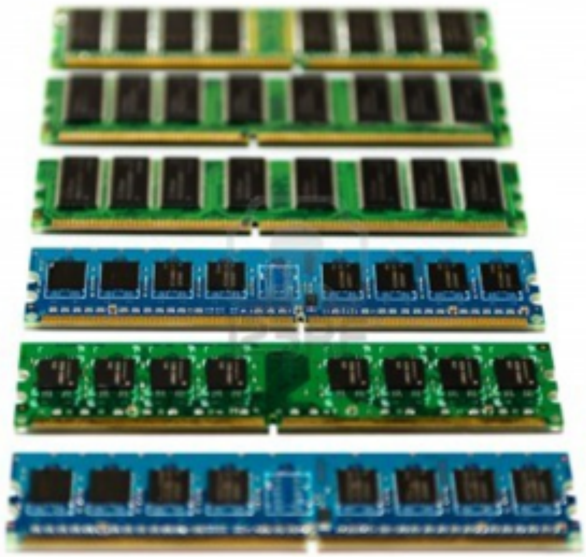
Department of Mathematics
and Computer Science

Lecture 8
Memory Management

Motivation: Virtualizing Memory

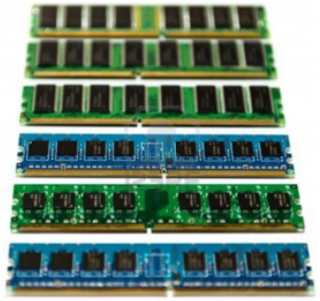
- Recall the goal of the OS: Timesharing
  - A program must be loaded into main memory before it can be run.
  - Reality: main memory is limited. How to run multiple processes?

- Policy: Want *Controlled Multiplexing* of main memory
  - Processes must be able to coexist securely
  - Each process thinks it has access to the entire main memory.



Memory Management Goals

- Need a new OS subsystem to deal with memory management
  - Memory Management Unit (MMU)



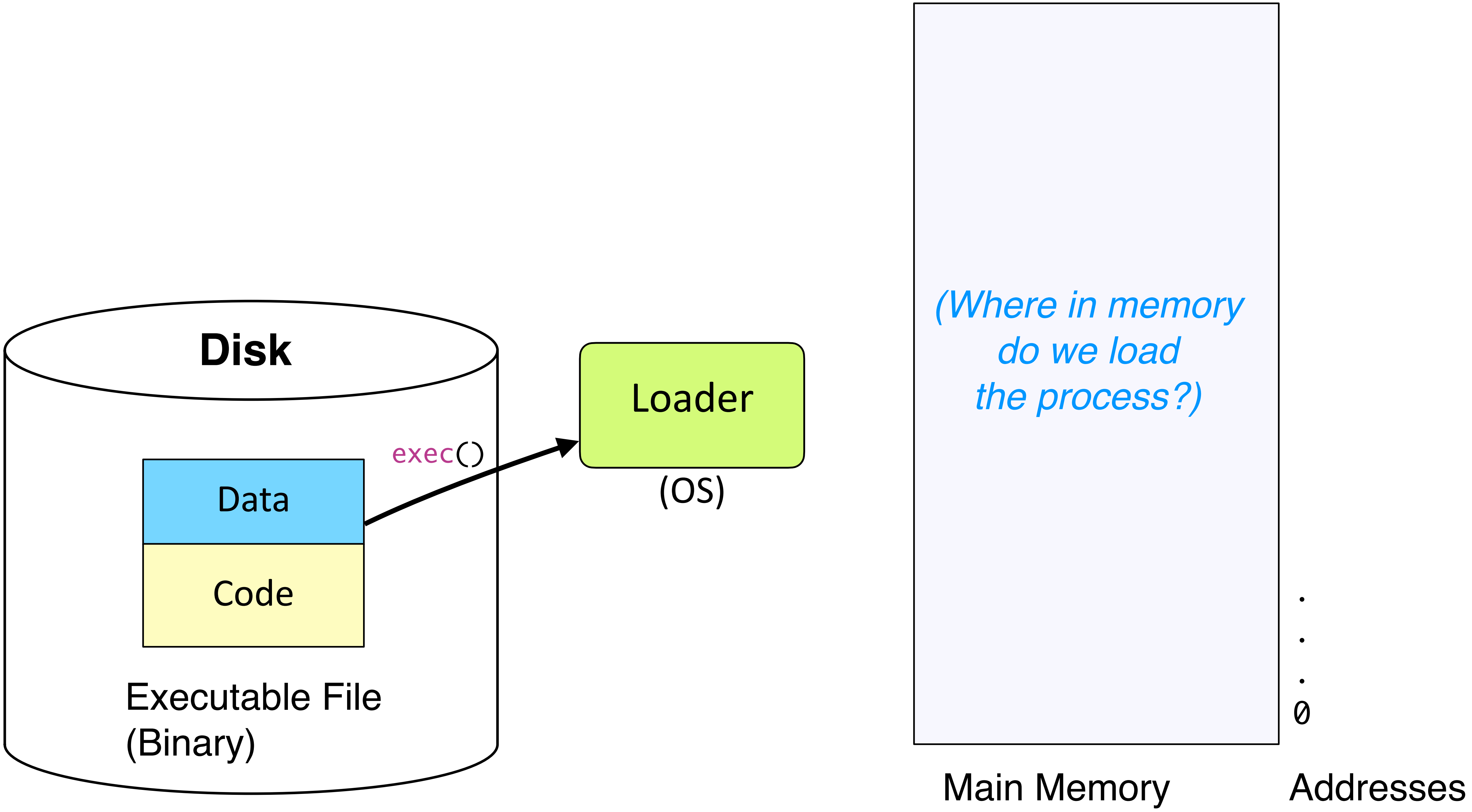
- An MMU must enable the following goals:
  - Process Relocation: OS can take process' address space out of memory and give it back later (*possibly* in a different physical location)
  - Protection of Address Spaces: Don't want different processes to have access to each other's memory.
  - Sharing: There are some cases when processes *do* want access to each other's memory (*e.g.*, share code across multiple processes)

Goals for This Lecture...

- Motivation and Goals
- Towards Virtual Addressing
  - Process Relocation
  - Address Translation
- Partitioning
- Segmentation
- Paging

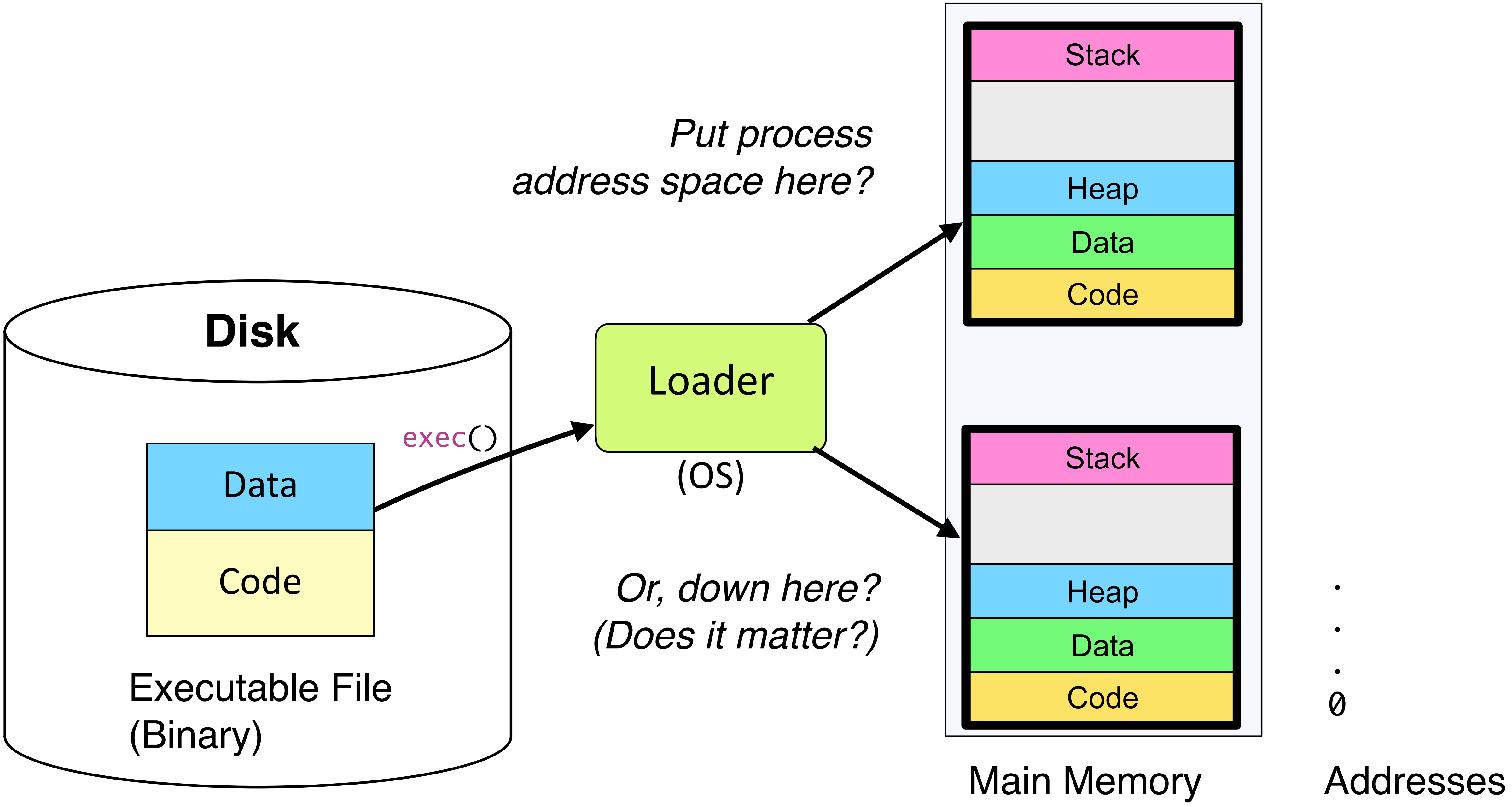
The Loader

- The `exec("/path/to/file")` system call invokes the OS's *Program Loader* to put an executable file into memory for execution.



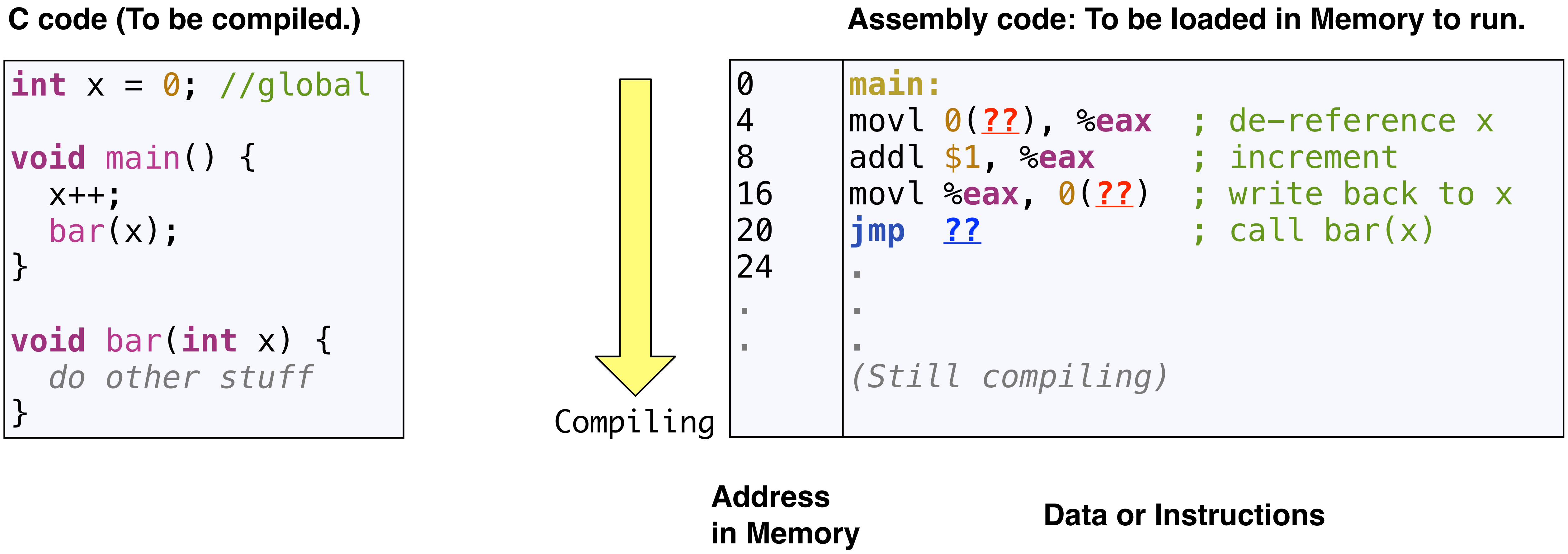
The Loader

- The `exec("/path/to/file")` system call invokes the OS's *Program Loader* to put an executable file into memory for execution.



Consider One of the Compiler's Jobs (1)

- Assume each instruction and data takes up 4 bytes of memory.



C code (To be compiled.)

```
int x = 0; //global

void main() {
  x++;
  bar(x);
}

void bar(int x) {
  do other stuff
}
```

Assembly code: To be loaded in Memory to run.

```
0      main:
4      movl 0(??), %eax   ; de-reference x
8      addl $1, %eax      ; increment
16     movl %eax, 0(??)   ; write back to x
20     jmp  ??            ; call bar(x)
24     .
.      .
.      .
       (Still compiling)
```

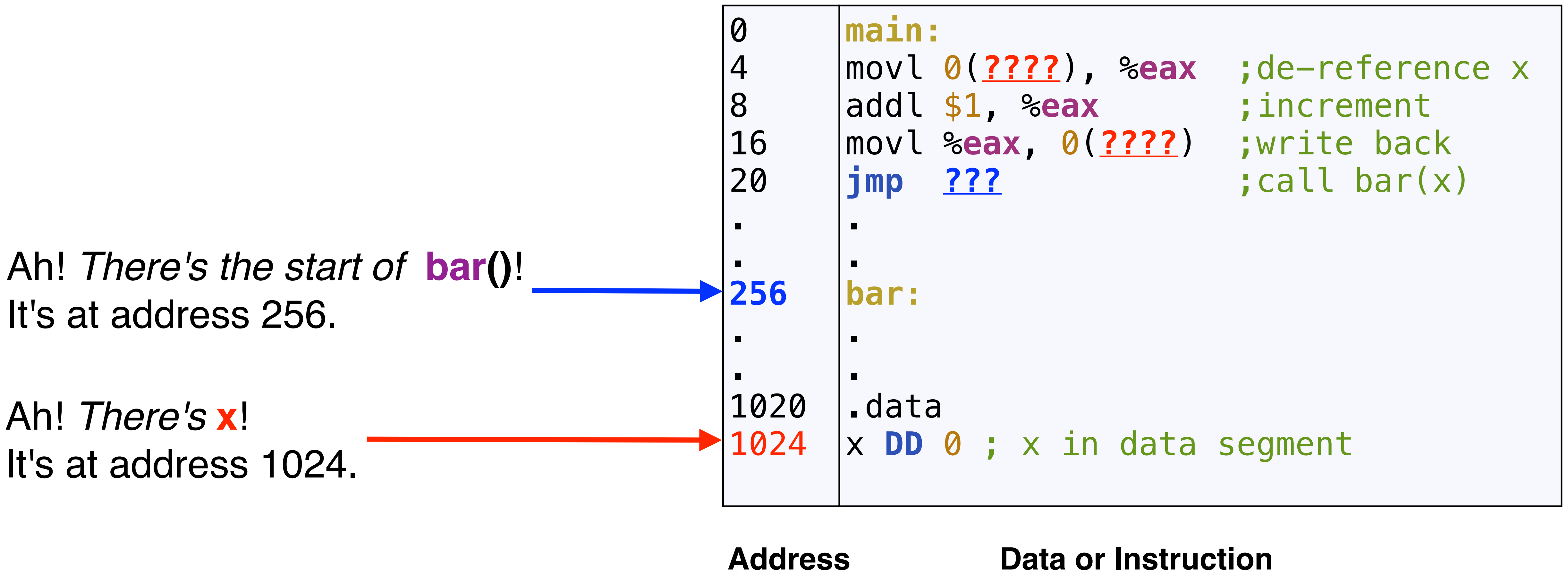
Address in Memory

Data or Instructions

- How does the compiler know *what address* ***(??)*** x will be placed?
  - *What address* ***(??)*** does `bar()` start at?

Consider One of the Compiler's Jobs (2)

Took a while to resolve, but *now* we know where **bar()** and **x** are located in memory



Ah! *There's the start of* **bar()**! It's at address 256.

Ah! *There's* **x**! It's at address 1024.

```
0       main:
4       movl 0(????), %eax   ;de-reference x
8       addl $1, %eax        ;increment
16      movl %eax, 0(????)   ;write back
20      jmp  ???             ;call bar(x)
.       .
.       .
256     bar:
.       .
.       .
1020    .data
1024    x DD 0 ; x in data segment
```

Address **Data or Instruction**

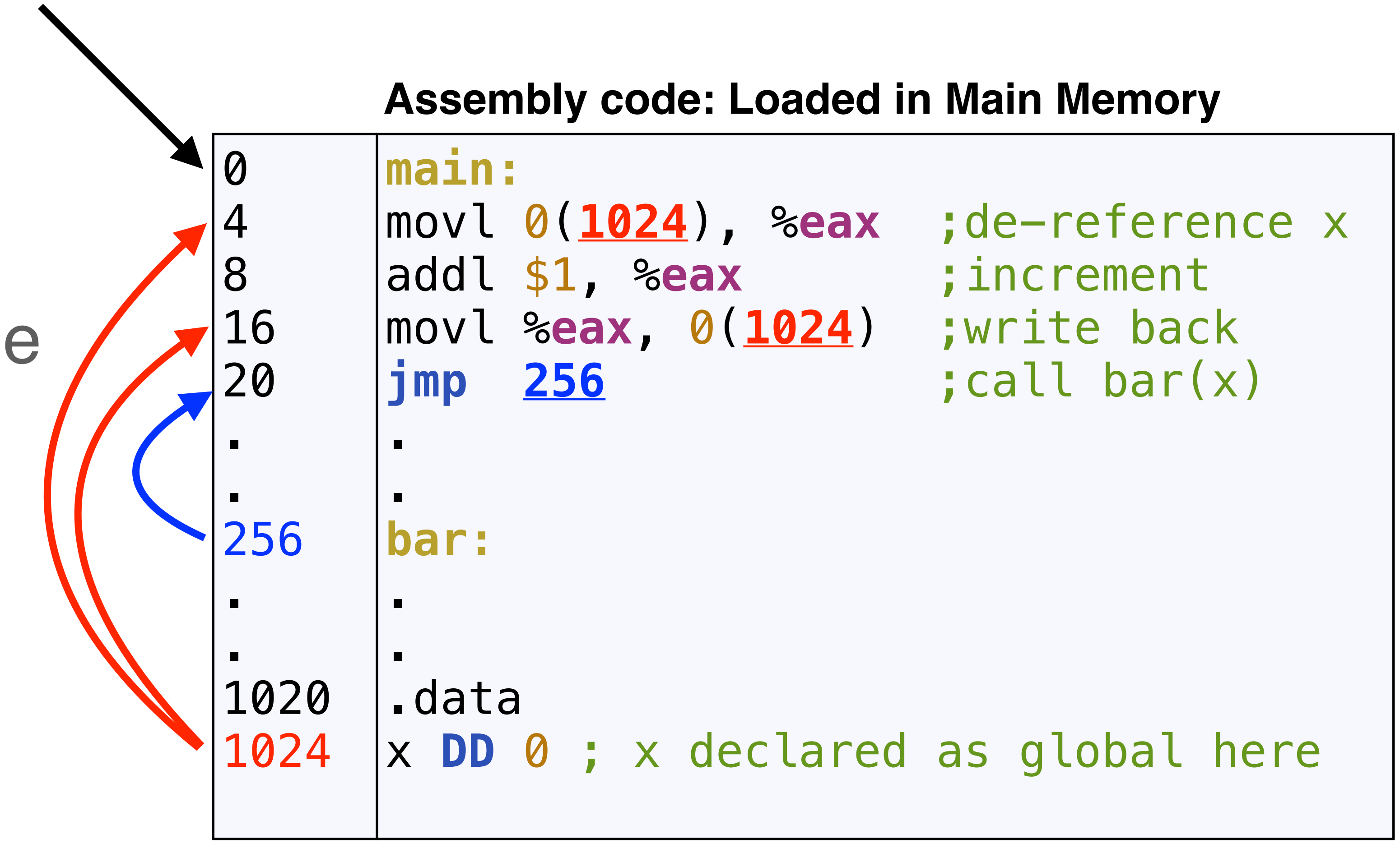
Address Binding

- *Address Binding:* Yay variables and functions can now be *"bound"* to addresses in memory.

Importance of zero as the starting (base) address!

- *Key Insight*
  - Address binding is easy when compiler can assume process address start from **0**
  - All other addresses are relative to **0**

Assembly code: Loaded in Main Memory



```
0      main:
4      movl 0(1024), %eax   ;de-reference x
8      addl $1, %eax        ;increment
16     movl %eax, 0(1024)   ;write back
20     jmp  256             ;call bar(x)
.      .
.      .
256    bar:
.      .
.      .
1020   .data
1024   x DD 0 ; x declared as global here
```

Addr **Data or Instruction**

Absolute Loader

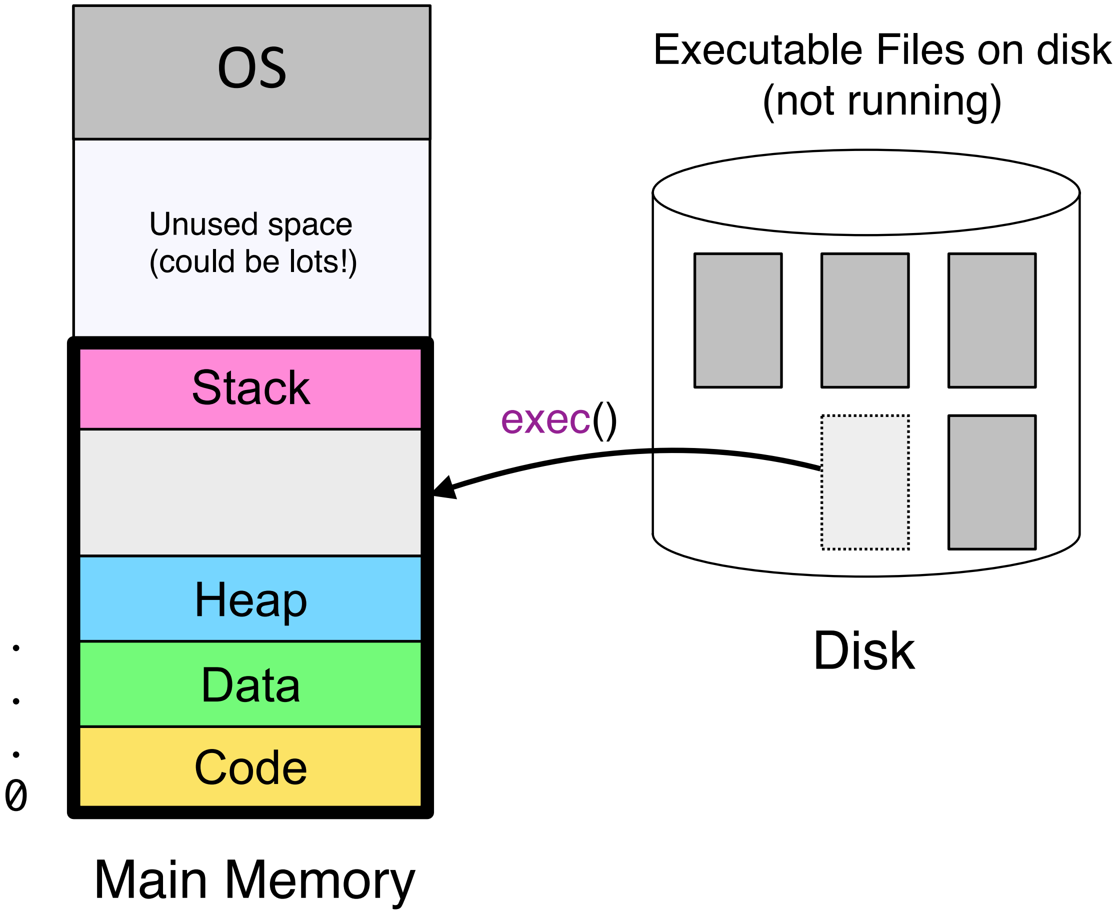
Absolute Loading

1. Pin the OS to highest memory location (far far away from user code)
2. **Always** load user process' address space starting from address **`0`**

   `(0 is called the "`*`base address`*`")`

- Only one user process can be in memory (*e.g.*, DOS)
- How does `contextSwitch()` work?
  - Well, *slowly!*

    (Two disk accesses)



Absolute Loading (Cont.)

- Used for batch-processing systems.
- Compiler binds variables and functions to physical addresses relative to the base address `0`
- Pros:
  - Only need to protect OS code from the user process
    - Raise a protection ("segmentation") fault and terminate user process on encroachment!
- Cons:
  - Lots of wasted space in main memory (gap between process and OS)
  - High overhead of context switching (disk involved)
  - Can't timeshare effectively if switching overhead is high

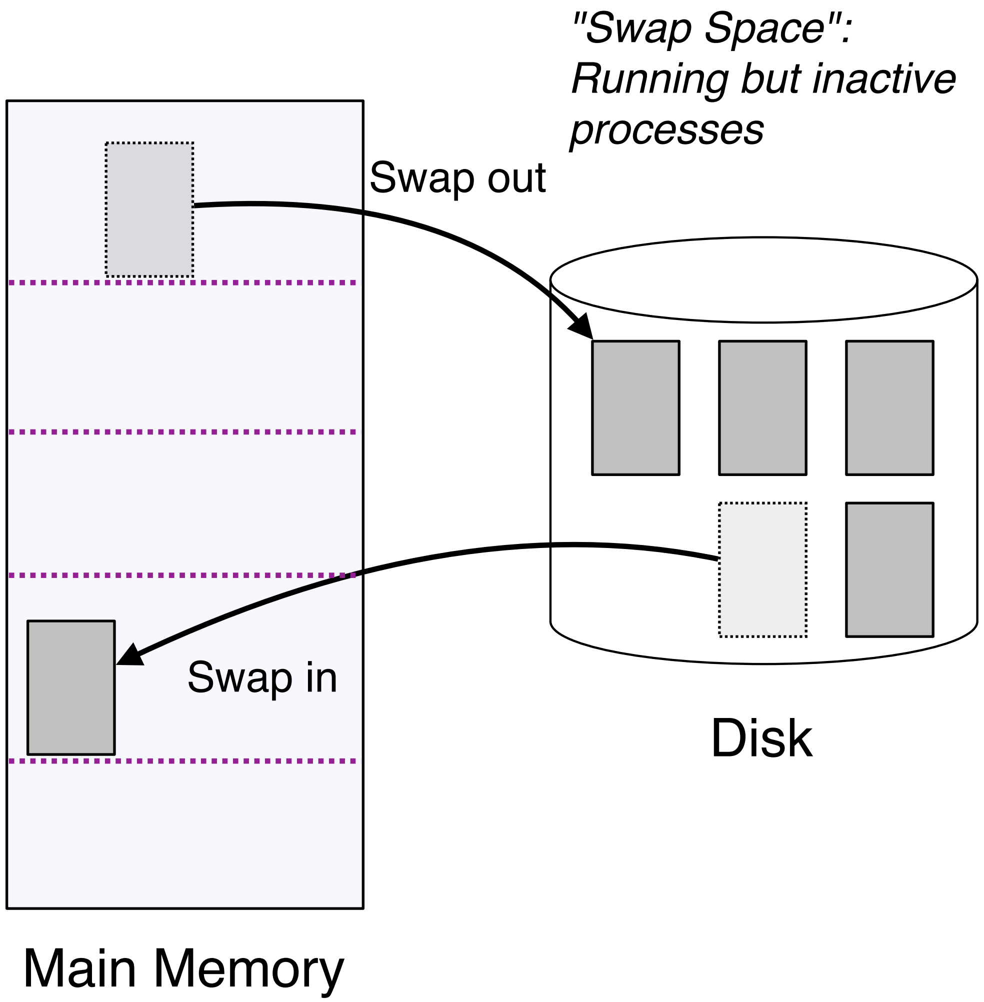
Goals for This Lecture...

- Motivation and Goals
- Towards Virtual Addressing
  - Process Relocation
  - Address Translation
- Partitioning
- Segmentation
- Paging

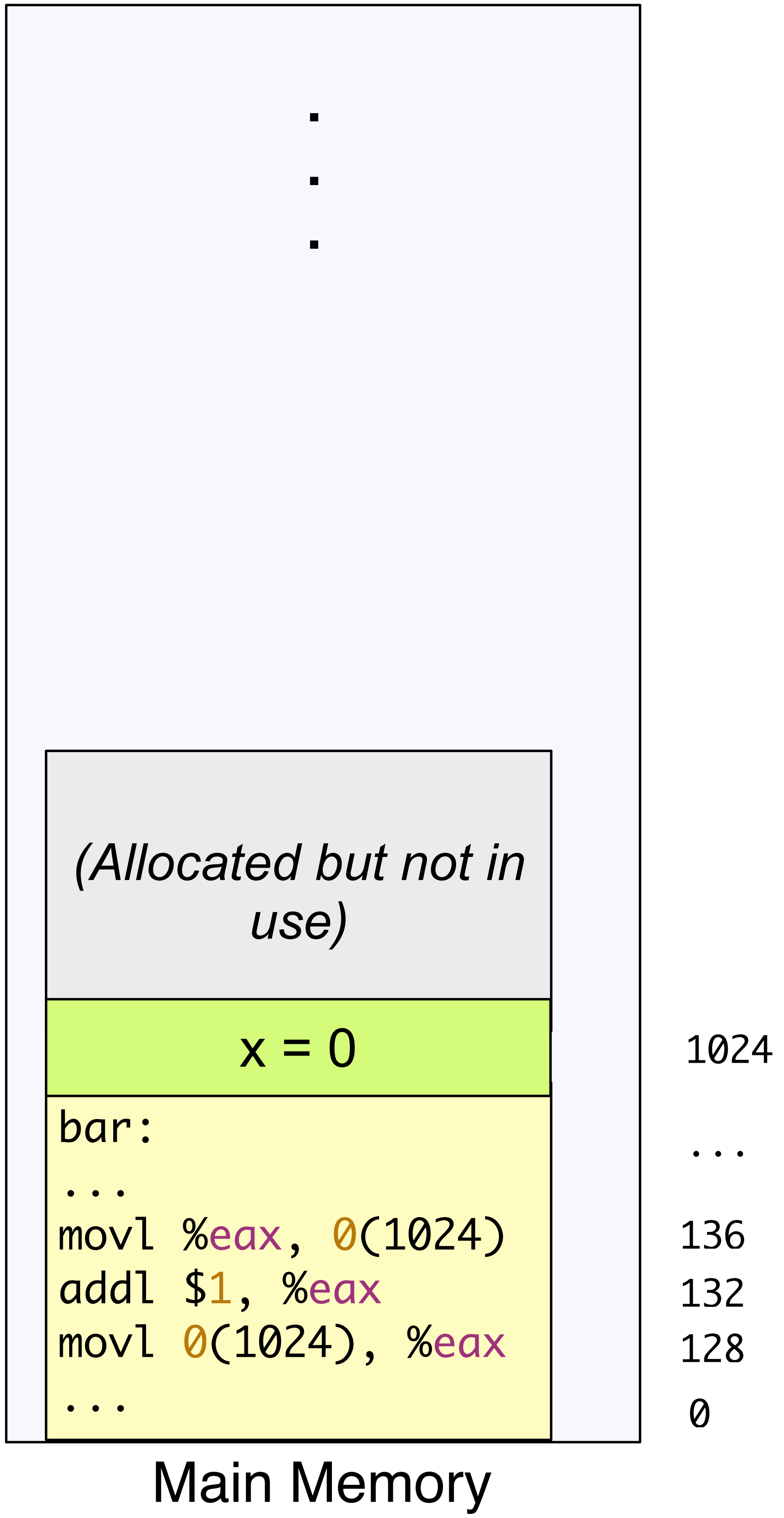
What If We Want Timesharing?

- But achieve high CPU utilization, we need to timeshare!
  - Memory is shared by *multiple* active process address spaces.
  - Split the main memory up into *partitions* where processes can live.

- But we still need space to put all the inactive processes: *"Swap Space"*
  - Reserve some space on disk.
  - Worst-case context switch?
    - Swap out to disk → Swap in from disk
  - Best case? Both already in memory

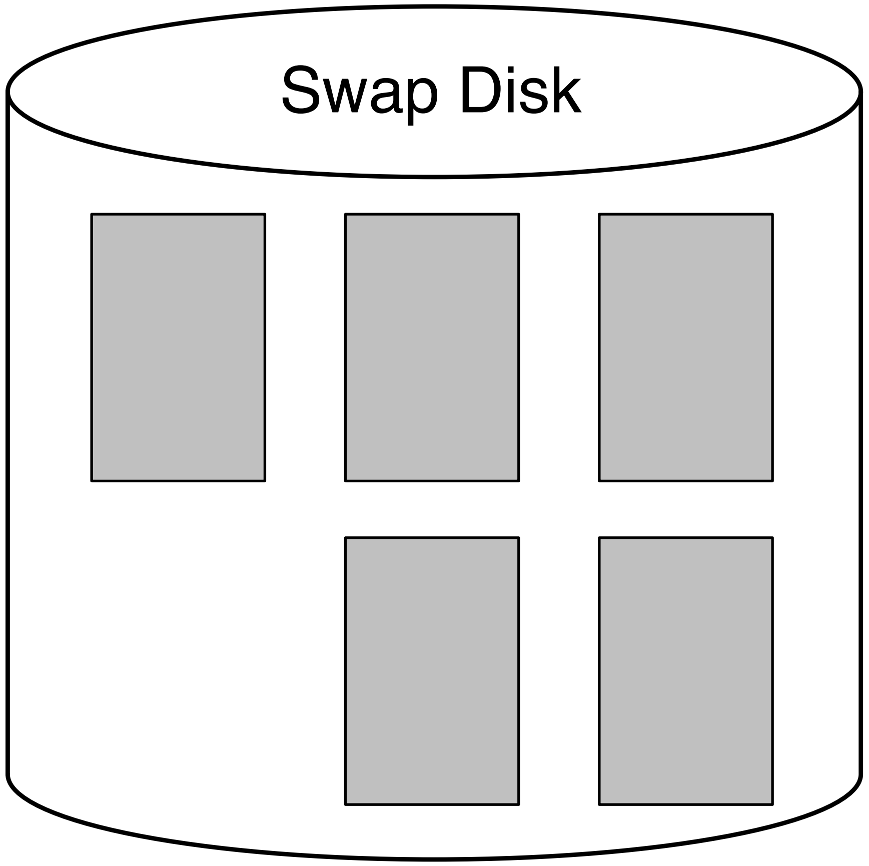


Process Relocation



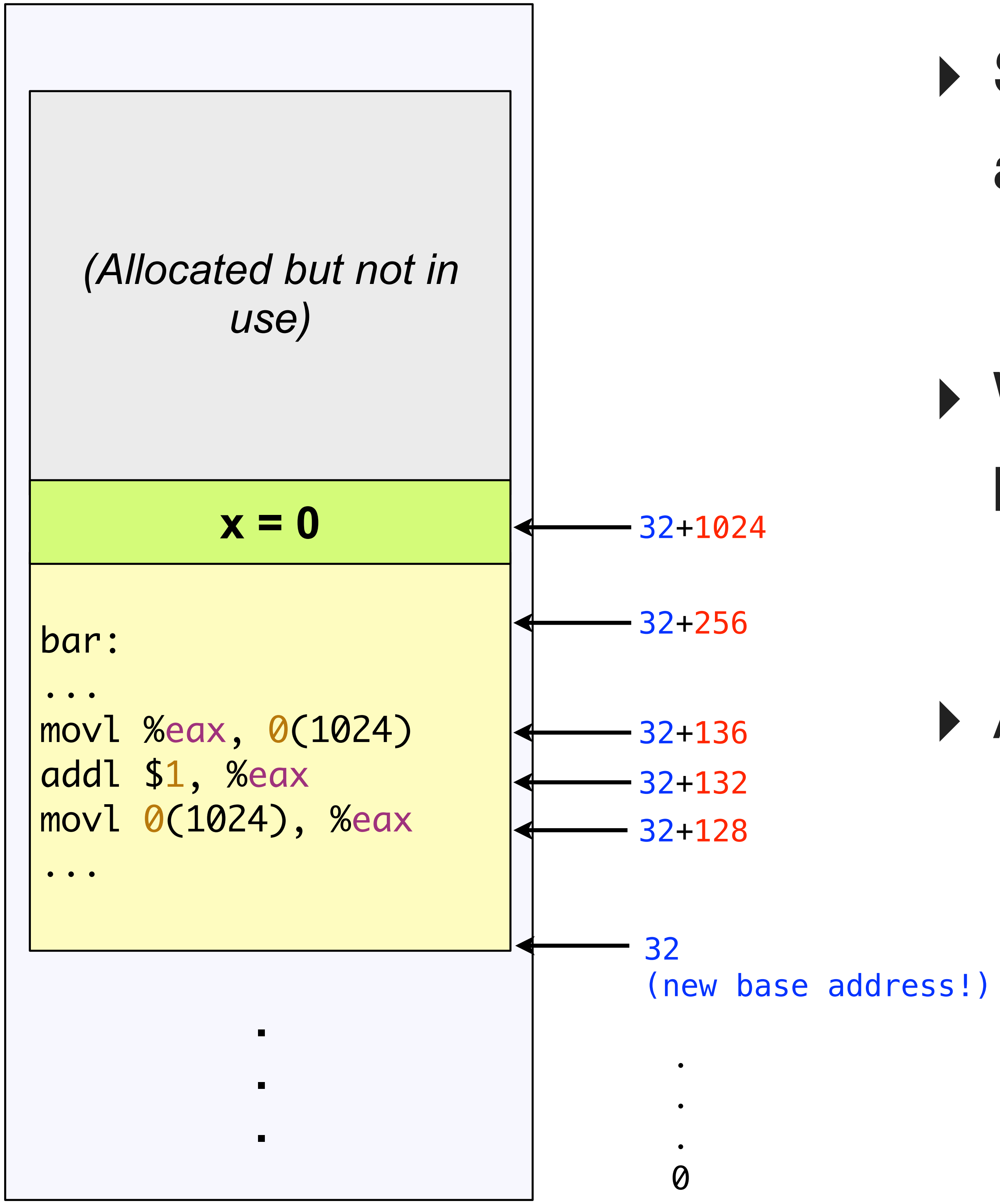
```
bar:
...
movl %eax, 0(1024)
addl $1, %eax
movl 0(1024), %eax
...
```

- Let processes swap into a potentially to a **different** location (i.e., "relocate")



- *But why is relocation a desirable policy to achieve higher CPU %? (The alternative is to always swap back into same location in RAM)*

How to Relocate? Need to Rebind Addresses!

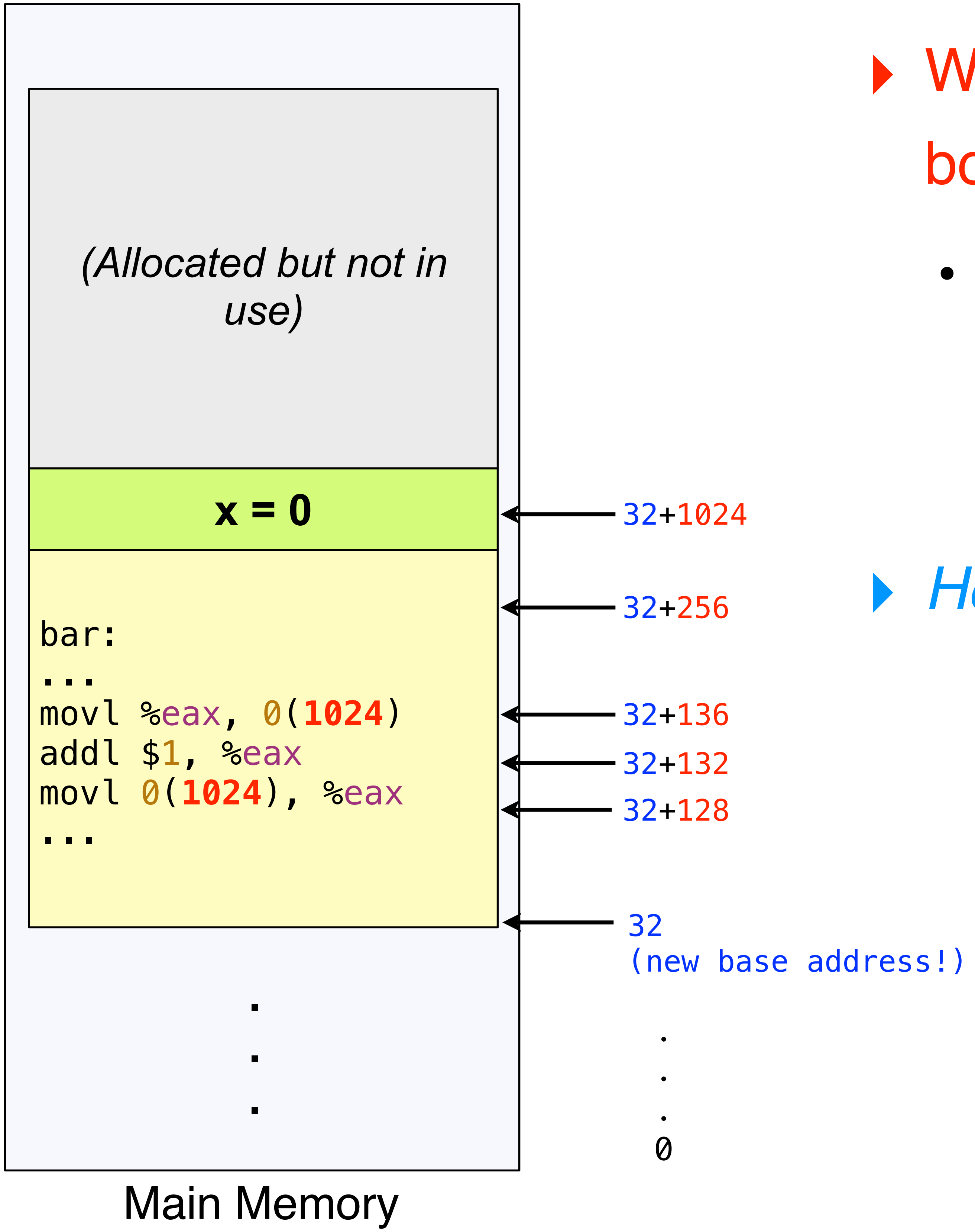


```
bar:
...
movl %eax, 0(1024)
addl $1, %eax
movl 0(1024), %eax
...
```

Main Memory

- Suppose this process began at address **0**, but gets swapped out.
- When it returns, it gets swapped back in at address **32**.
- After swapping back into memory,
  - Execution will raise a segmentation fault! *(Why?)*

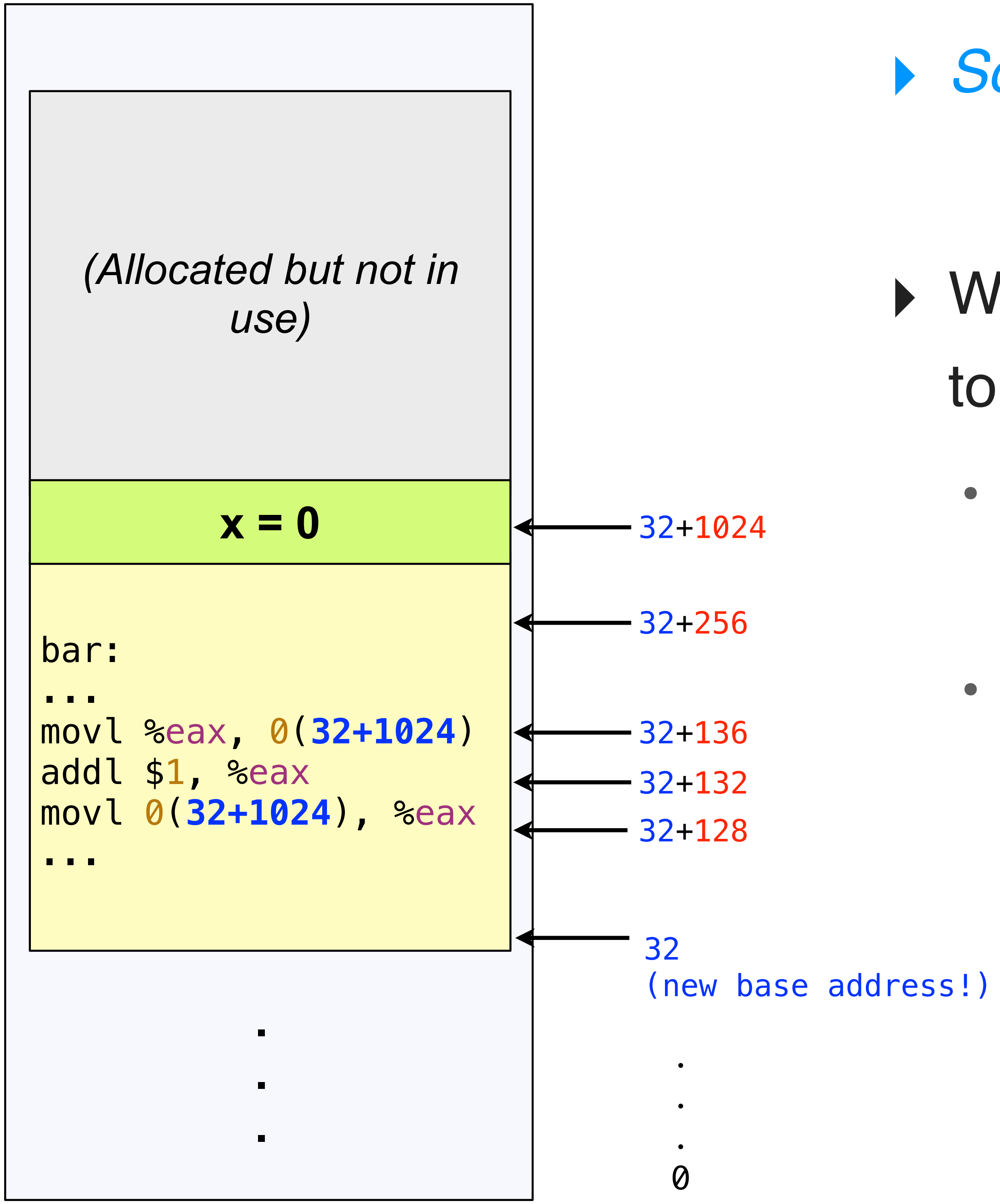
How to Relocate? Need to Rebind Addresses!



```
bar:
...
movl %eax, 0(1024)
addl $1, %eax
movl 0(1024), %eax
...
```

- Why does it segfault? All the compiler-bound addresses are now **wrong**!
  - All address references in the code now off by an offset of: **-32**
- *How do we fix the code's addresses?*

How to Relocate? Need to Rebind Addresses!



```
bar:
...
movl %eax, 0(32+1024)
addl $1, %eax
movl 0(32+1024), %eax
...
```

Main Memory

- *Solution: "Address Translation"*
- When loading a process to run, OS has to **re-bind** the addresses!
  - Add the *New Base Address* (**32**) to all the old address references in the code!
  - Oh no we have to retranslate (recompile) all references before swapping a process back in!

Static Relocation (Load-Time Binding)

Static Relocation

1. Split physical memory into partitions
2. Put OS in one of these partitions
3. Compiler ***still binds*** user code's addresses relative to address `0`
4. But at *Load Time*, the OS:
   (a) Assigns the process to an appropriate partition in memory
   (b) Loader binds the compiler-generated *addresses* to the OS-assigned base address to reflect assigned location

Compiler-generated addresses aren't fixed (or "real") anymore

These addresses *are* real!

Loading is really slow (Need to translate every address in the program first!)

Evaluation of Static Relocation

- Pros:
  - Physical memory can be multiplexed across different processes
  - Totally software-based
    - Translate all addresses directly within the binary executable file at load time
  - No changes to compiler technology
    - Compiler still *thinks* that all user program's addresses start at `0` unless told otherwise
- Cons:
  - No protection of address spaces
    - A process could still refer to an illegal address (un)intentionally.
  - Very poor performance
    - Swap-in times suffer since the *loader* must re-bind every address reference to a potentially new **base** address!

Goals for This Lecture...

- Motivation and Goals
- Towards Virtual Addressing
  - Process Relocation
  - Address Translation
- Partitioning
- Segmentation
- Paging

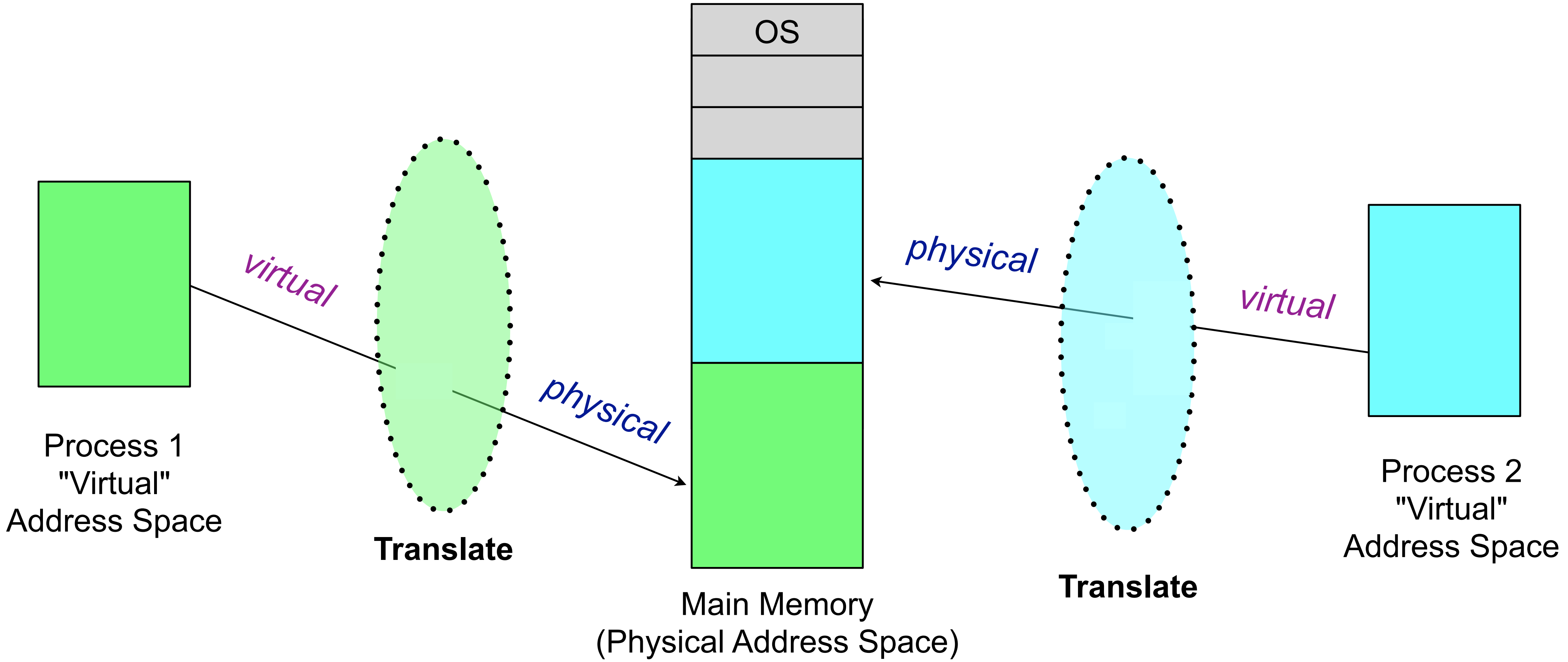
Virtual Addressing

- *Virtual Addressing* is considered one of the biggest breakthroughs in computer science. (First appeared in Multics)

- Ren and Stimpy go to the post office.
  - Post office has many physical mailboxes to rent out.
  - Adam wants PO Box 1, 2, and 3.
  - America wants PO Box 1, 2, 3, and 4.
  - Can the post office satisfy both customers?

Virtual Addressing & Address Translation

- *Virtual Address:* an address that a process uses to access its own address space (Processes know nothing about real addresses!)
- *Physical Address:* the address in physical memory
  - Address translation done by the loader (for now...)



Dynamic Relocation (Runtime Binding)

Dynamic Relocation (Hardware-Supported)

1. Split physical memory up into partitions
2. Put the OS in one of these partitions.
3. Compiler *still binds* user code's addresses relative to base `0`
4. Translate each virtual address to a physical address *as the process runs.*

- Implementing this mechanism will be our focus
  - *How does address translation at runtime (vs. at load-time) help us multiprogram?*

Goals for This Lecture...

- Motivation and Goals
- Towards Virtual Addressing
- Memory Partitioning
  - Fixed Partitioning
  - Dynamic Partitioning (Base and Bounds)
- Segmentation
- Paging

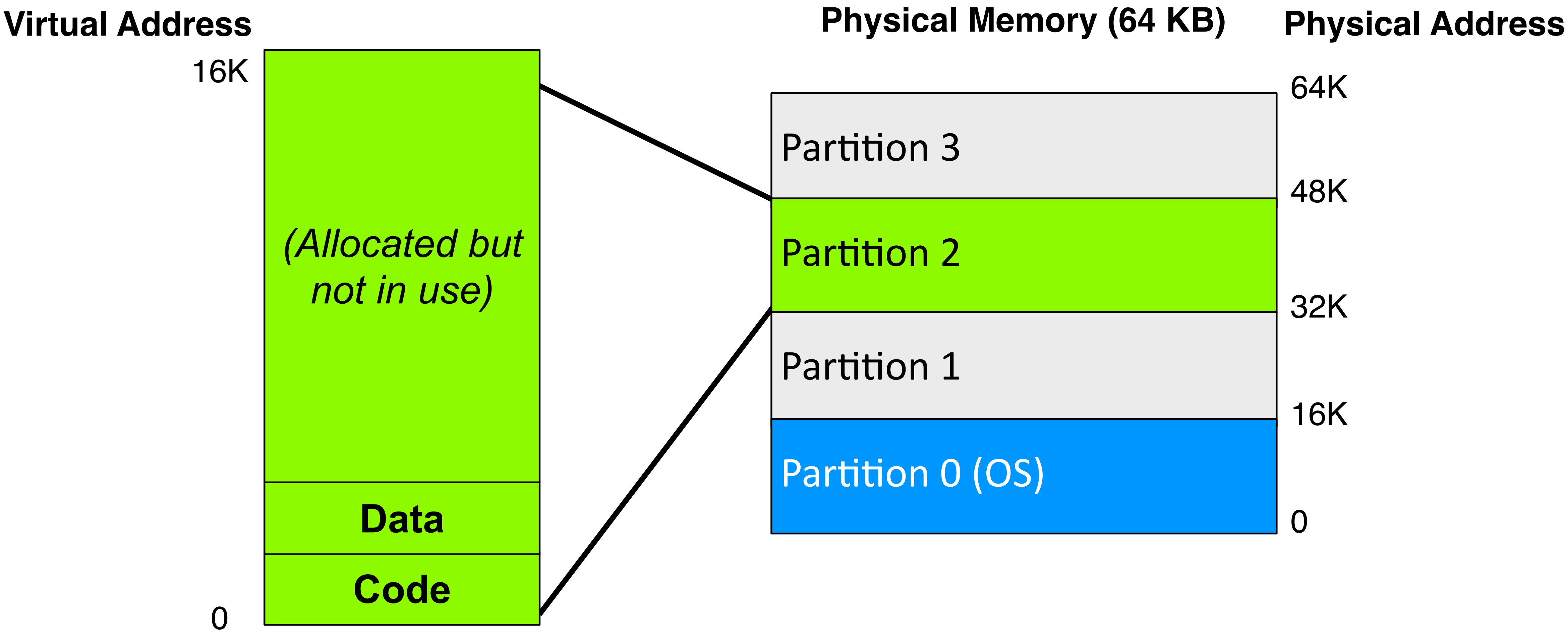
Some Assumptions

- Let's make some simplifying assumptions:
  1. Size of process address space $\leq$ available physical memory
  2. The address space is monolithic and must be placed *contiguously* in physical memory.
     - (This *will* change later...)
  3. Specifically for this lecture:
     - 64 KB available physical memory
     - Process' address spaces are allowed a maximum of 16 KB
     - Therefore, 64 KB / 16 KB = 4 processes can be loaded in memory simultaneously

Fixed Partitioning

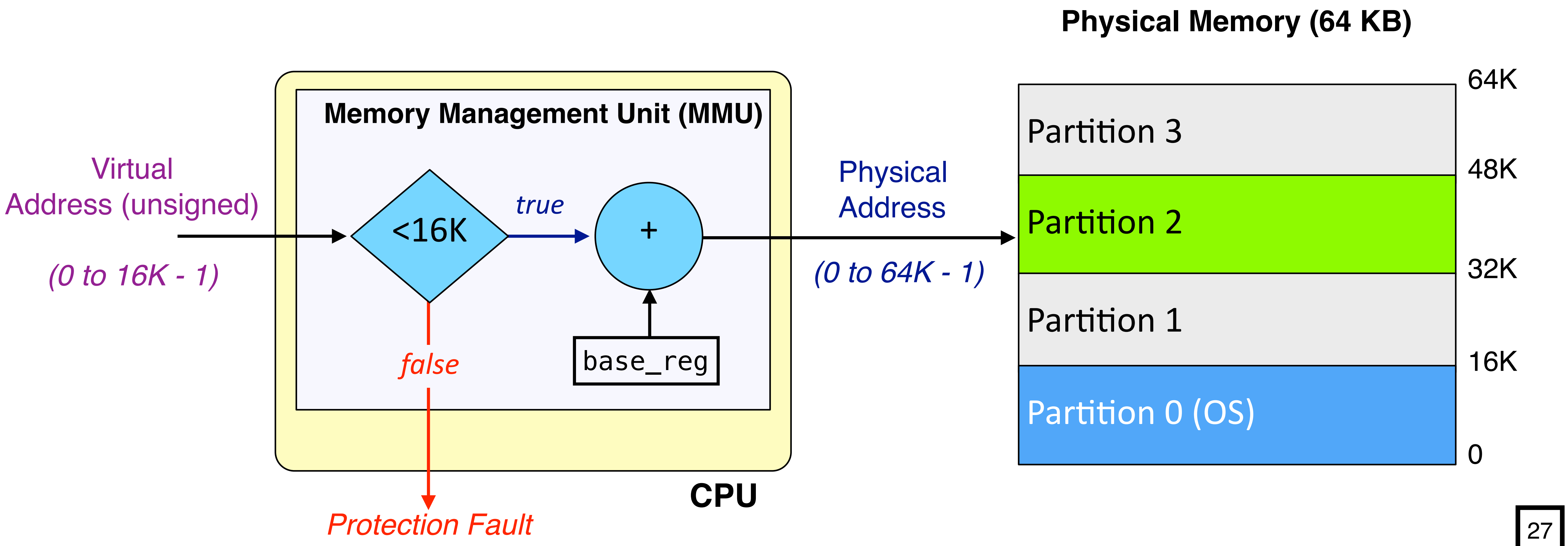
- *Fixed Partitioning:* The original dynamic-relocation technique
  - Break physical memory into equal-sized partitions

- A process' entire address space is loaded into a partition



Implementing Fixed Partitioning

- Address translation (base address of process stored in process' PCB)
  - *physical addr = virtual addr + base addr*
- Memory protection is now possible!
  - If *virtual addr ≥ 16K*, generate a segmentation fault and terminate
- Updated hardware (MMU):



Implementing Fixed Partitioning (cont.)

- How to context switch now?
  - Store a process' **base address** in its PCB
    - Holds the *base address* of the partition to which a process is assigned
  - Add a `base register` to the CPU hardware
    - Holds the *base address* of the currently running process

- A chance that the scheduled process was out on disk (or was new)
  - Loader looks for a free partition and loads the process from disk
    - Update the *base address* in the process PCB
    - Update the *base register* on the CPU
    - (If no empty partitions exist, then OS first swaps out another process)

Evaluation of Fixed Partitioning

- Pros:
  - Enables code relocation and basic memory protection!
  - Address translation is fast and with minimal hardware changes
  - Switching doesn’t slow badly (better than static relocation)
  - Partitions are of equal size: Don't need a complicated algorithm to find a "gap" big enough to fit the process' entire address space.

Evaluation of Fixed Partitioning

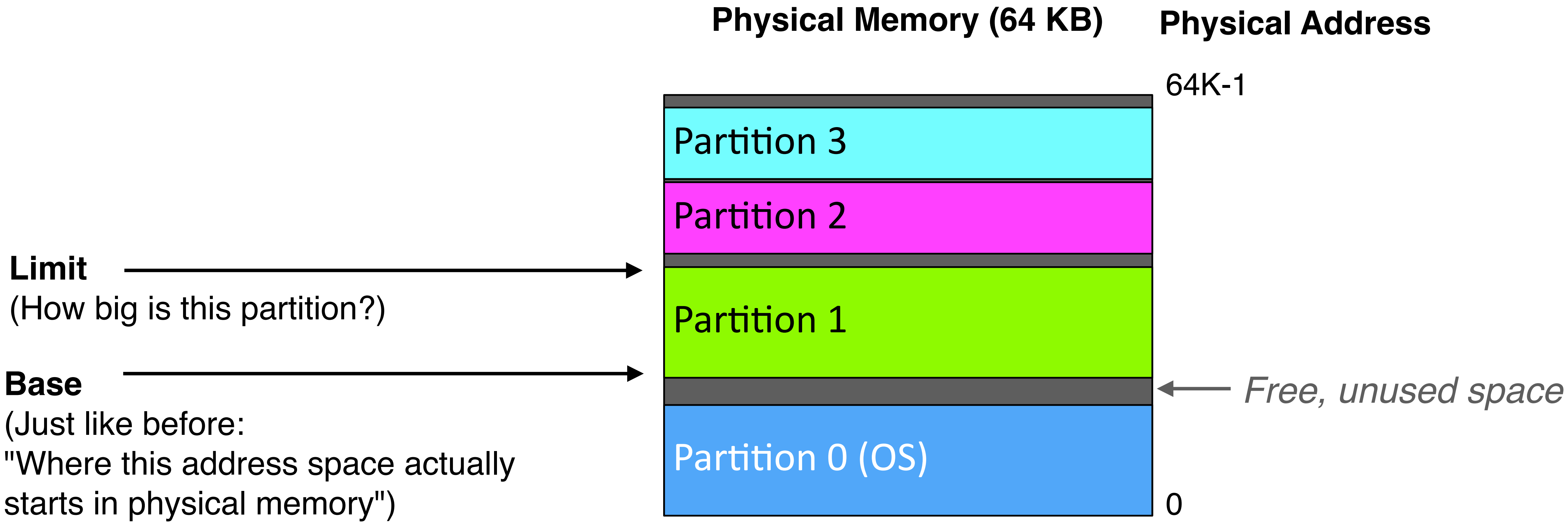
- Cons:
  - The fixed number of partitions set by the OS **limits** the degree of multiprogramming
  - Memory partitions are 1-size-fits-all, but processes vary wildly in actual memory usage
    - Processes *can't grow* beyond the OS-defined partition size

- Susceptible to *"internal fragmentation"*
  - That is, unused space *within* memory partitions

Variable Partitioning (Base and Bounds)

Variable Partitioning

- Seeks to fix internal fragmentation and to remove limit on number of partitions OS can have.
- When a process gets brought into memory, it gets a partition of exactly the right size.



Limit
(How big is this partition?)

Base
(Just like before:
"Where this address space actually
starts in physical memory")

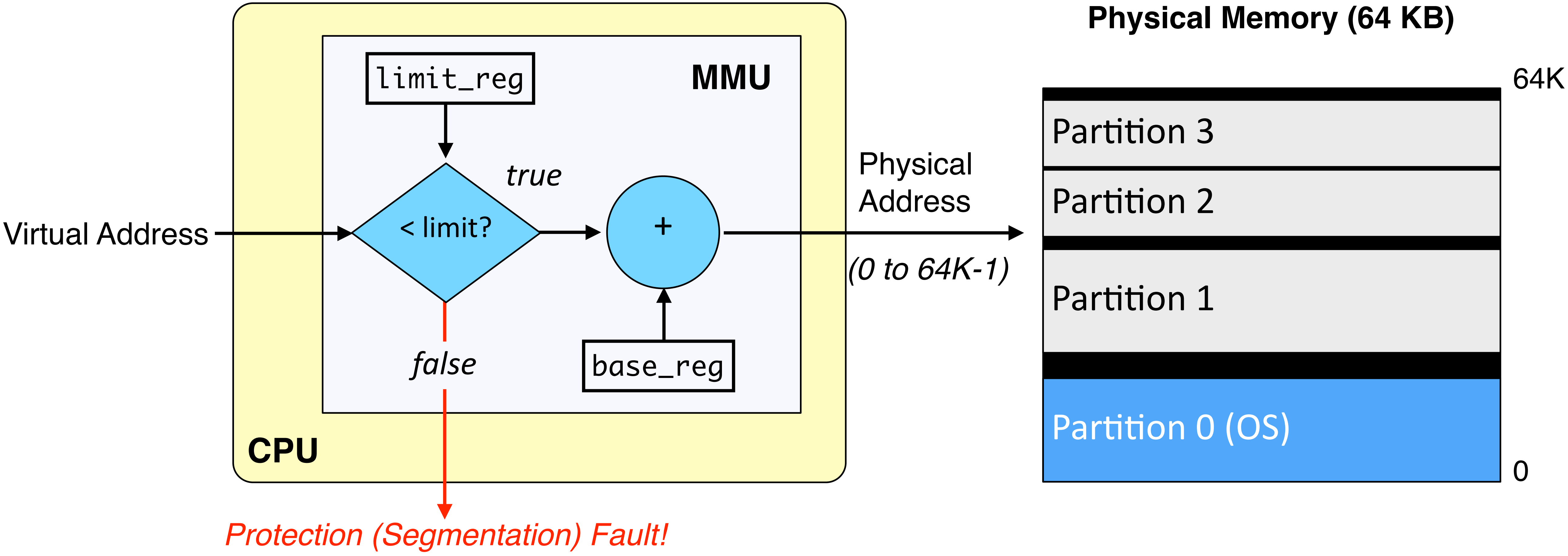
Variable Partitioning (Cont.)

- Each process (PCB) now has to remember:
  - `base_addr` - Where does process address space actually start in physical memory?
  - `limit` - How big is your partition?

- Hardware requirements:
  - `base_reg` and `limit_reg` added to the CPU that will get populated when the process PCB is loaded or switched in.

Variable Partitioning (Cont.)

- Translation logic (same as fixed partitioning)
  - *physical addr = virtual addr + base addr*
- Memory Protection:
  - If *virtual addr ≥ limit*, generate segfault and terminate process



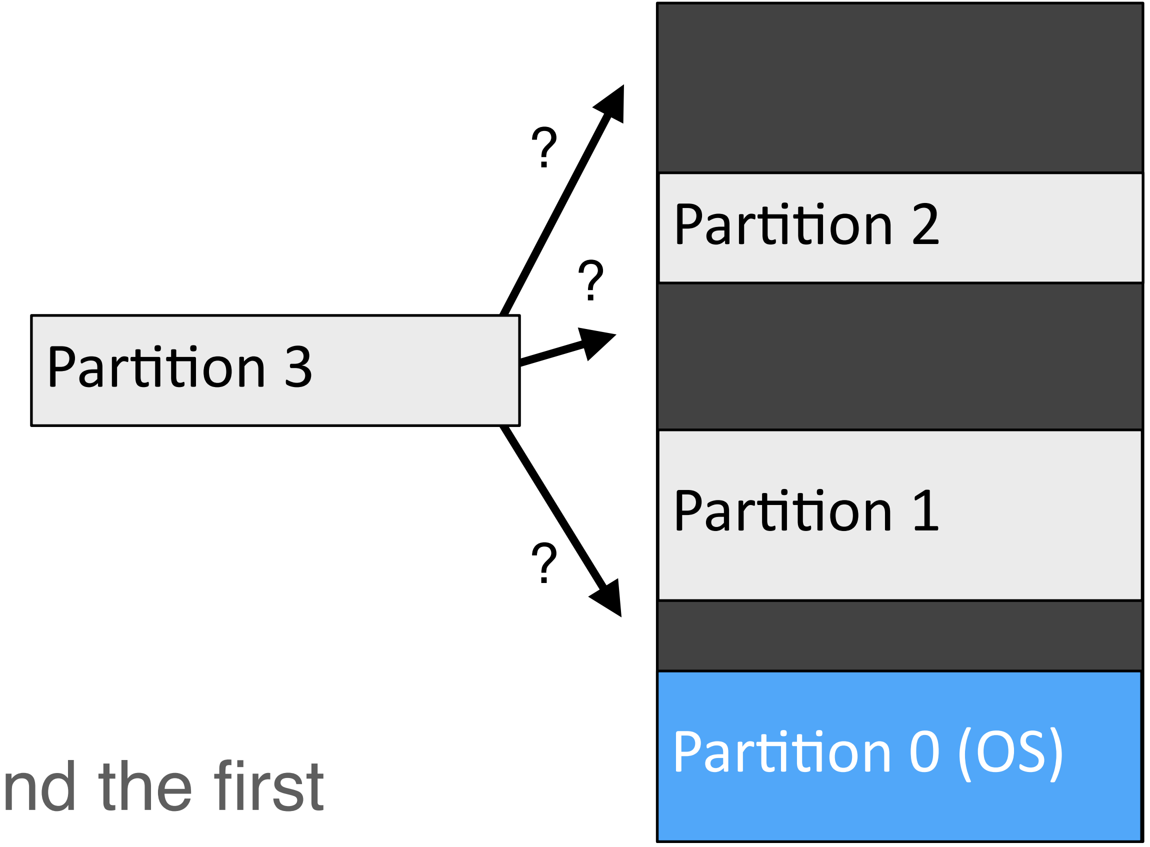
Evaluation of Variable Partitioning

- Pros:
  - No more internal fragmentation!
  - Degree of multiprogramming no longer confined by no. of partitions
  - Processes can even grow in size during runtime (i.e. malloc)
    - Swap address space out to disk, increase *limit*, swap back in.
  - Minimal hardware changes

- Cons:
  - External fragmentation is now possible!
  - Need a memory allocation algorithm (next slide)
    - Adds overhead to context switching

Memory Allocation Policy Needed!

- *Memory allocation:* With variable partitions, there are differently sized gaps in memory.
  - What's the right *gap* to assign a new process?



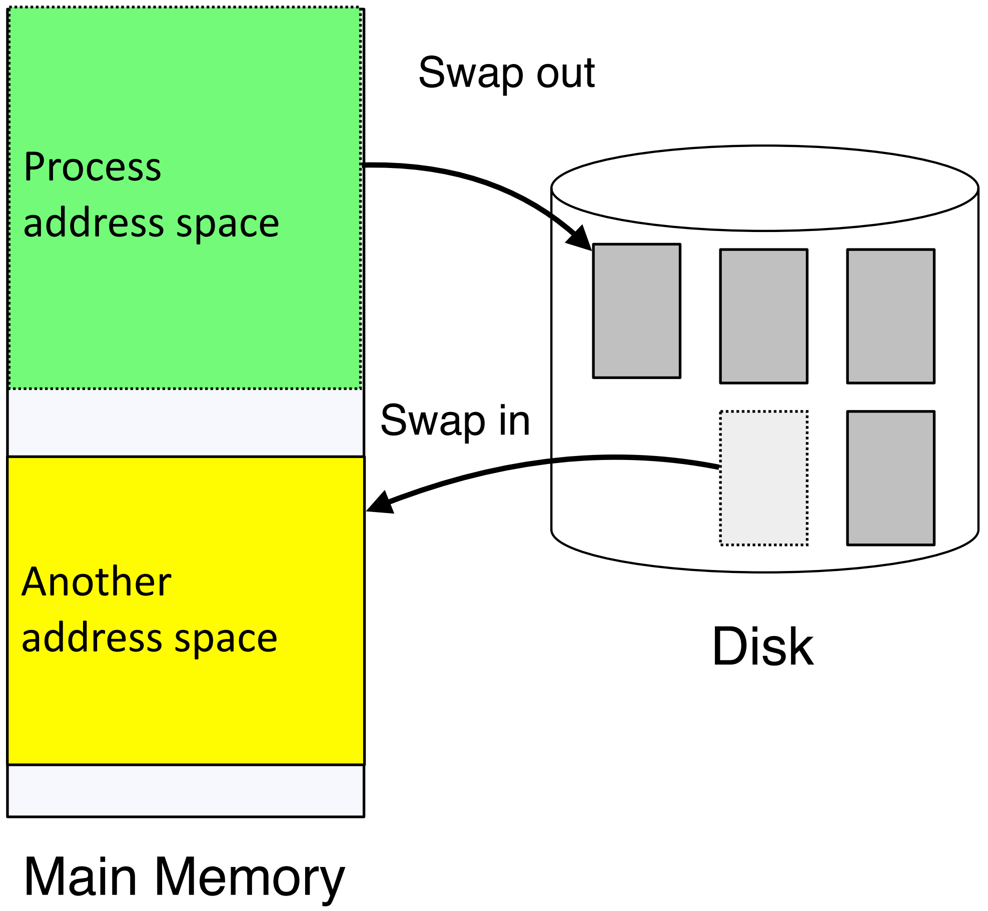
- Policies:
  - First Fit: Starting from physical address `0`, find the first gap that is wide enough to fit the process.
  - Best Fit: Find the best-fitting gap.
    - Best-Fit usually the worst.
    - Gives rise to Worst-Fit usually being the best -- *wait what?*

Goals for This Lecture...

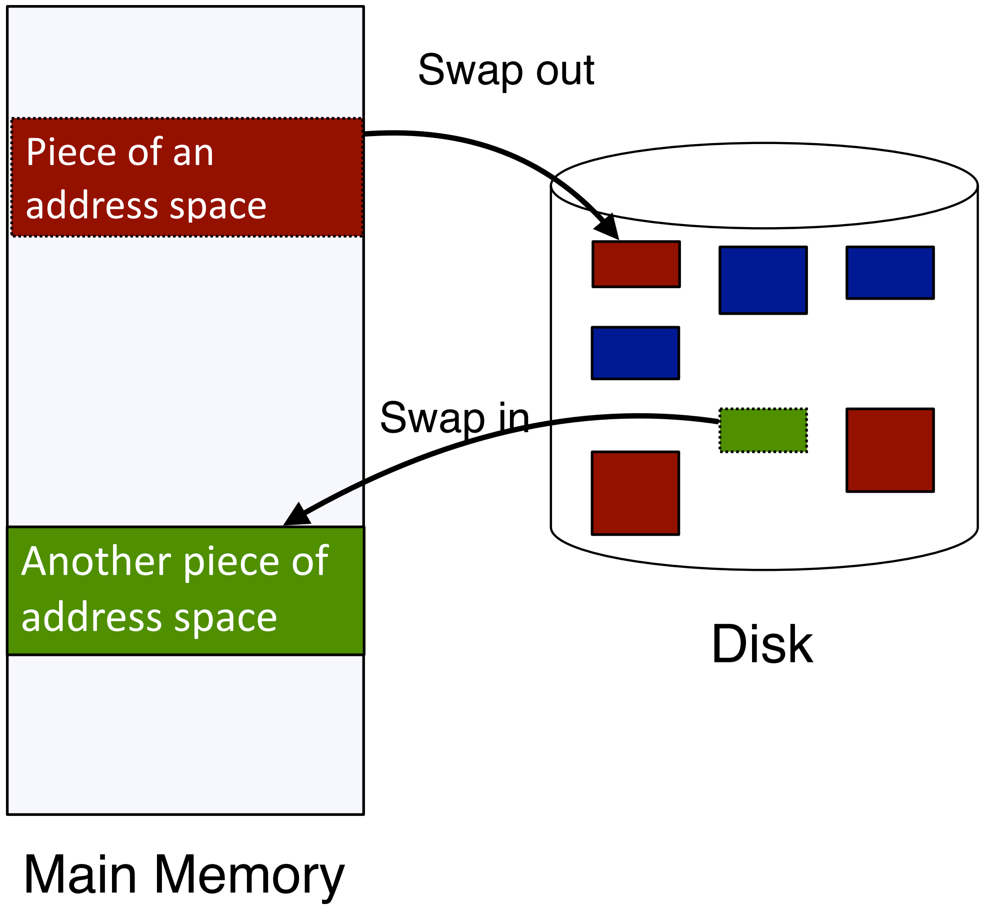
- Motivation and Goals
- Towards Virtual Addressing
- Partitioning
  - Fixed Partitioning
  - Variable Partitioning (Base and Bounds)
- Segmentation
- Paging

Partitions Are Monolithic and Restrictive

- Currently, the **whole** address space must fit into one partition.
  - Context switching can be a pain if a partition isn't available.



Old (Partitioned) Memory Model



New (Segmented) Memory Model

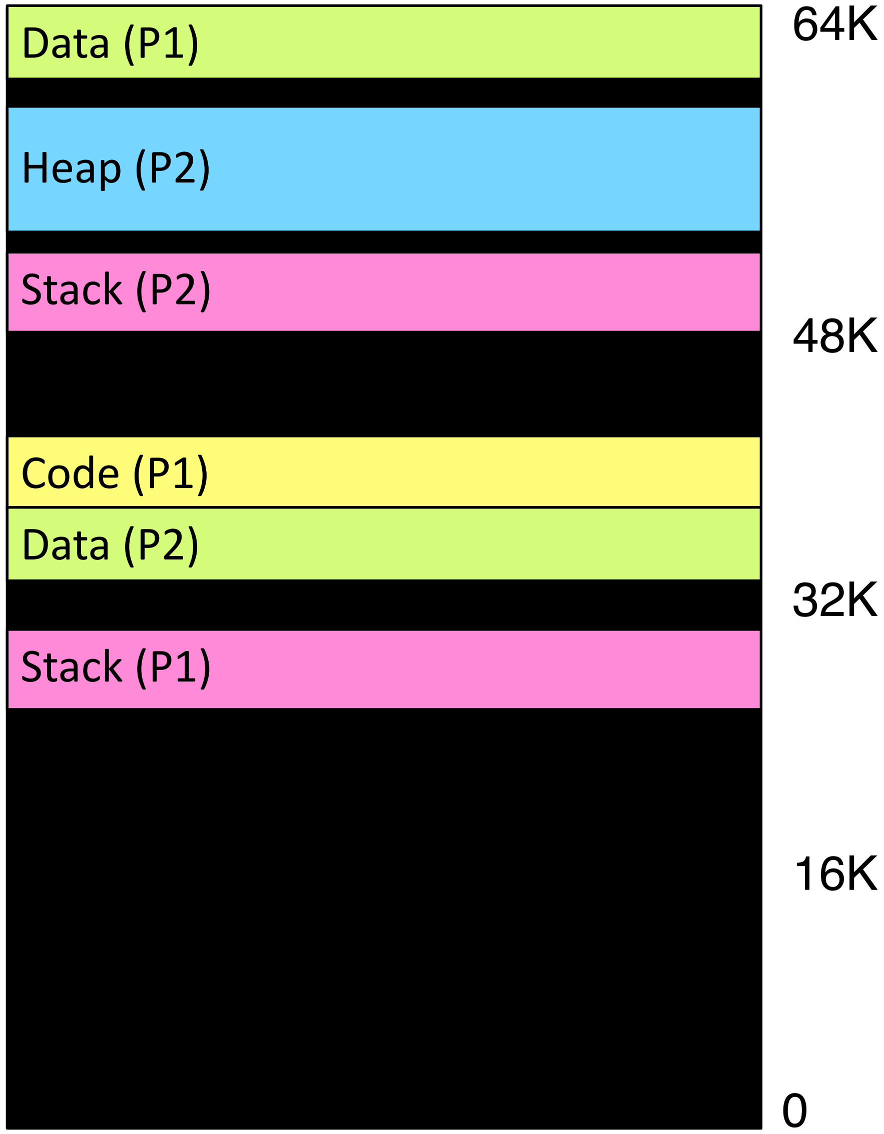
Memory Segmentation

- New Idea: Segmentation
  - Break each address space down into: *code, data, heap, stack*
  - Each segment gets its own "**base**" and "**limit**"

Key difference from Variable Partitioning:

- Each segment is independently mapped to physical memory.
- Flexibility for multiprogramming: Not *all* segments need to be loaded in order for process to run.
- Requires a "segment table" for each process.

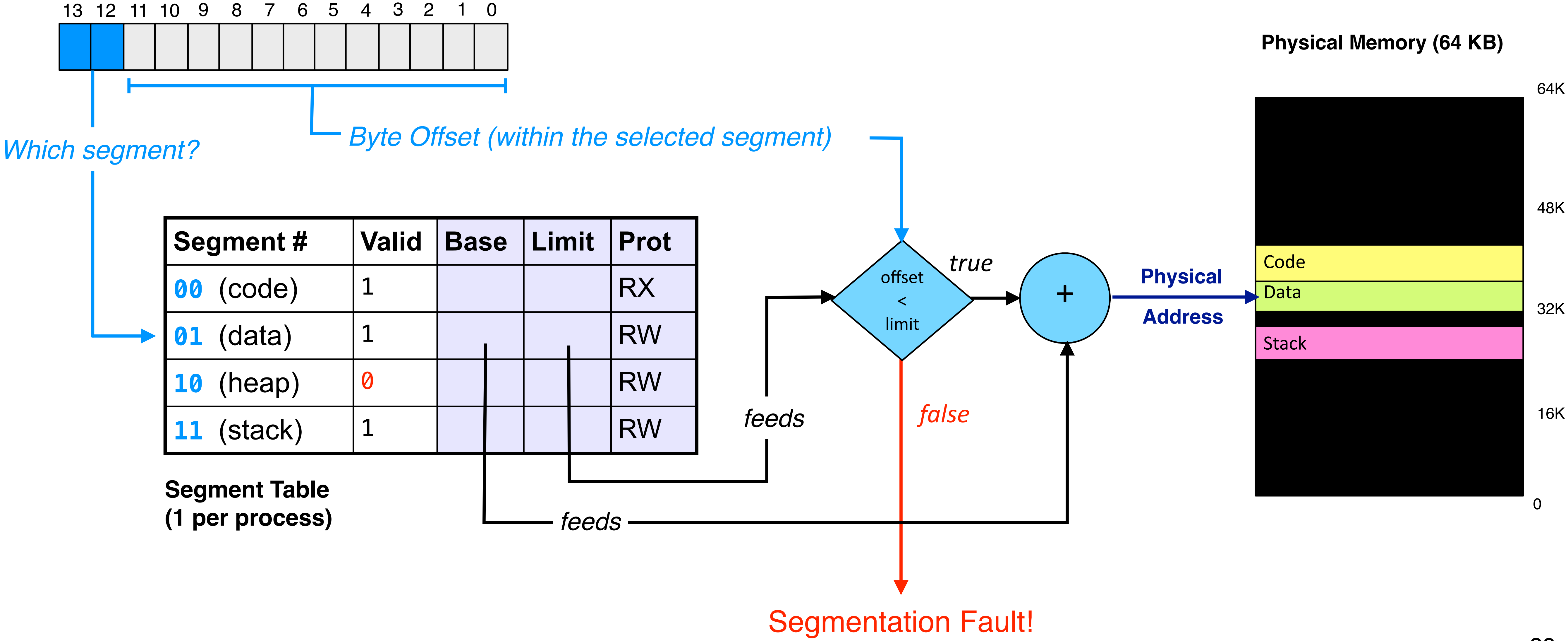
Physical Memory (64 KB)



Segmentation: Address Translation

- Each process now has a *Segment Table (or "Segment Map")*
  - What's the max size (limit) of each segment in this scheme below?

Virtual Address for a byte in process space:
(Assuming 16 KB address space for processes, the virtual address requires $\log_2 16K = 14$ bits to address each byte)



Segment #	Valid	Base	Limit	Prot
00 (code)	1			RX
01 (data)	1			RW
10 (heap)	0			RW
11 (stack)	1			RW

Segment Table (1 per process)

Sanity Check: Segmentation

- Consider a system that supports 32-bit addressing.
  - How much physical memory (RAM) can this system handle?
  - What is the possible range of addresses per process?
    - (I.e., how many bytes can each process have access to?)
  - What is the maximum segment size?

Sanity Check: Segmentation (Sol'n)

- Consider a system that supports 32-bit addressing.
  - How much physical memory (RAM) can this system handle?
    - Solution: 1 address per byte, so there are 2^{32} possible addresses.
    - So, $2^{32}\ bytes = 4GB$. Anything over that would be un-addressable with only 32 bits.
  - What is the possible range of addresses per process?
    - (I.e., how many bytes can each process have access to?)
    - Solution: addressable range: 0 to $2^{32} - 1$ ... so yes one process address space could take up all the physical memory!
  - What is the maximum segment size?
    - Solution: You need the left-most two bits to address each segment, leaving 30 bits for the byte offset.
    - Therefore, 2^{30} bytes can be addressed per segment, so the segment size is $2^{30}\ bytes = 1GB$

Segmentation Summary

- Pros:
  - Better space utilization and flexibility than partitions since segments can be placed (re-located) anywhere in physical memory.
  - Segments can be swapped out independently of each other
    - "Valid Bit" in the segment table in tells us whether a segment is loaded in memory or is out on disk
  - Segment table allows for easy memory sharing
    - Example: Two processes can use the same `(base, limit)` for the code segment
  - Segments can grow + shrink during runtime
    - Swap segment out to disk, change its "limit" and swap back in

Segmentation Summary (Cont.)

- Cons:
  - Overhead in OS
    - A segment table needs to be set up for each process
    - (Process creation time is now slower)
  - Address translation is much slower than memory partitioning
    - Each memory access needs to first consult segment table instead of just adding the offset to a base register
  - External fragmentation of segments *still* possible
    - (Holes *between* segments get smaller and smaller and will add up and make it hard to swap segments in)

Goals for This Lecture...

- Motivation and Goals
- Towards Virtual Addressing
- Partitioning
- Segmentation
- Paging
    - Page Tables
    - Translation Lookaside Buffer (TLB)

Segmentation Review

- Segmentation offers...
  - Dynamic relocation and protection ✓
    - Virtual addressing and translation
  - Code sharing ✓
    - Via segment tables
    - Just map a segment from different processes to the same base/limit

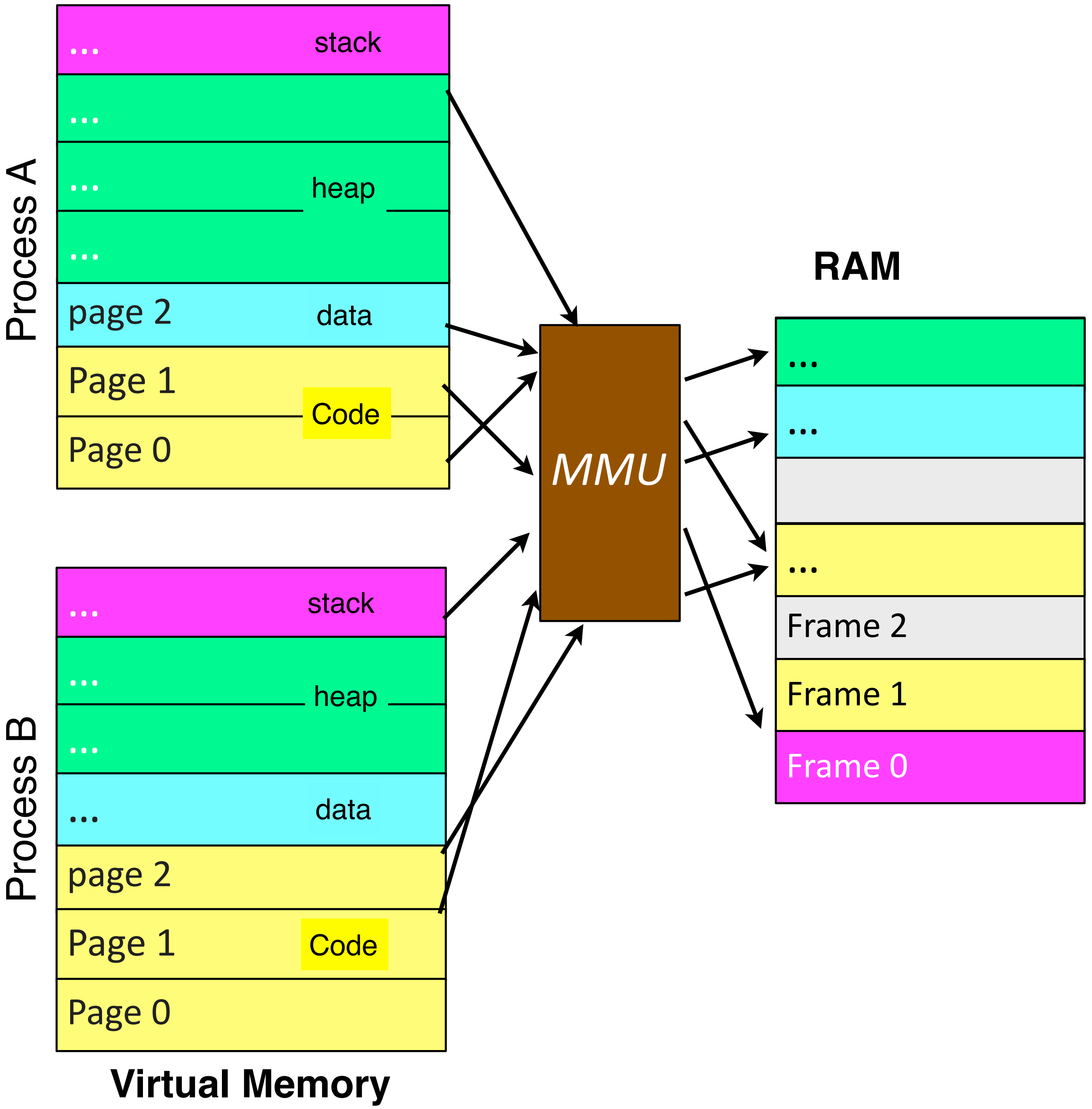
- One major problem remains:
  - **External Fragmentation of Segments**

We need to focus on eliminating **External Fragmentation.**

* They arise from being variable length.
* We should return to fixed-length chunks!
* Which means, we will *allow* internal fragmentation!

Modern Memory Management: Paging

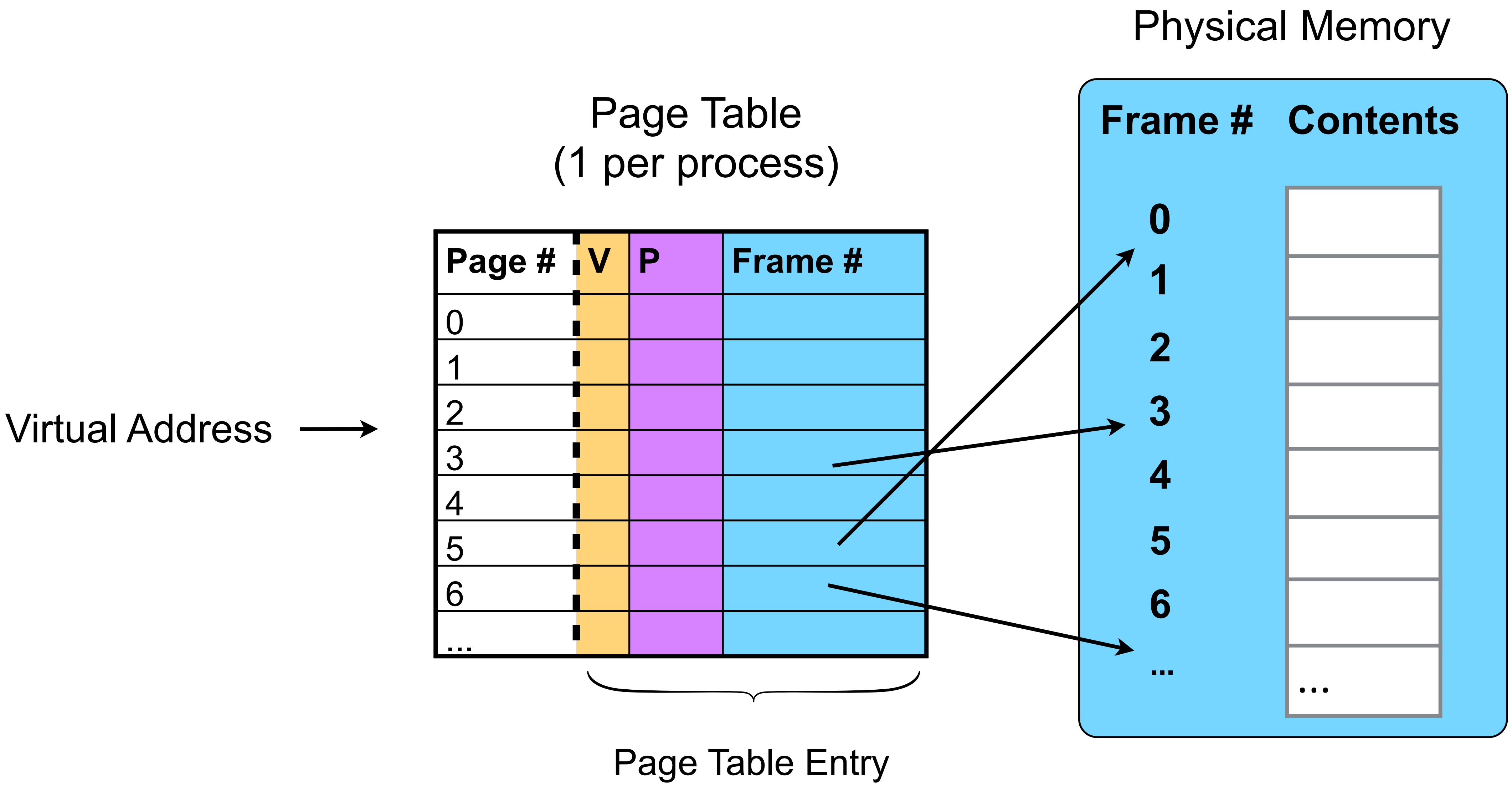
Lesson learned from segmentation: Processes don’t need a contiguous, monolithic view of its own address space. **We can break it up!**



- Use **small, fixed-sized** chunks of virtual and physical memory
  - *Page*: a virtual memory chunk
  - *Frame*: a physical memory chunk
  - *Page Size = Frame Size*
    - This constraint *must* be satisfied!
    - Anywhere from 512 bytes to 16 KB in practice.

Page Tables

- How does the OS keep track of pages for a process?



Physical Memory

Page Table
(1 per process)

Virtual Address →

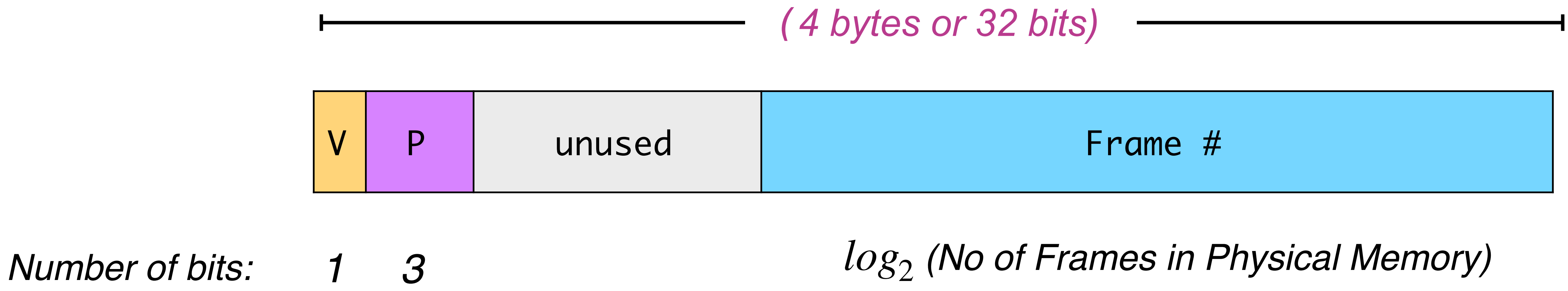
Page #	V	P	Frame #
0			
1			
2			
3			
4			
5			
6			
...			

Page Table Entry

Frame #	Contents
0	
1	
2	
3	
4	
5	
6	
...	...

Page Table Entries

- A "page table entry" is really just 32-bit sequence (i.e., an int)



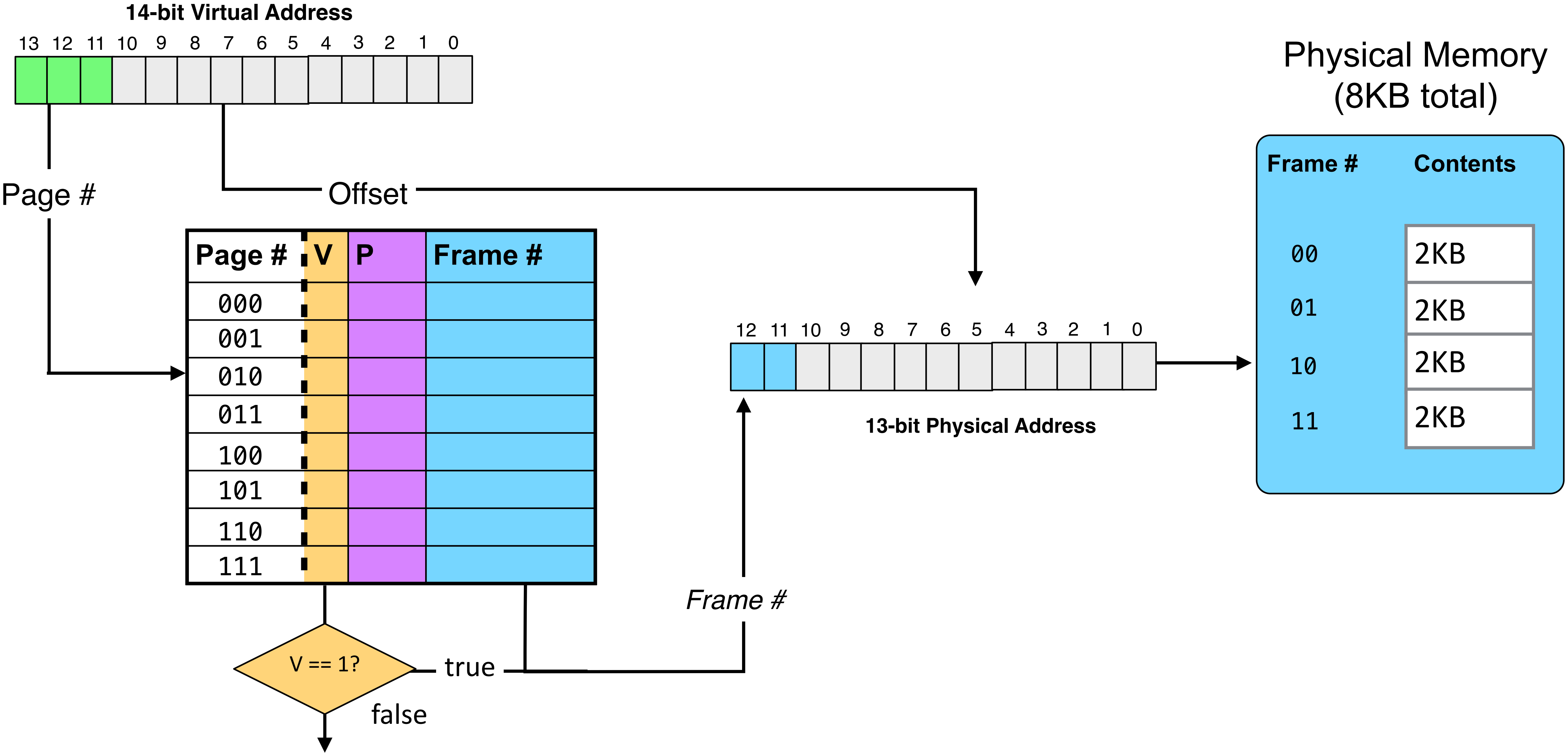
(4 bytes or 32 bits)

	V	P	unused	Frame #
Number of bits:	*1*	*3*		log_2 *(No of Frames in Physical Memory)*

- What do these bits mean?
  - **Valid Bit (V):** Is the page present in memory (1) or out on disk (0)?
  - **Protection Bits (P)**: Read-Only? Read-Write? Read-Execute? ...
  - **Frame #**: Which frame in physical memory does this page map to?
  - **Unused:** any remaining bits might be useful in the future.

Address Translation Logic (Paging)

- Example: 8 pages; 4 frames in physical memory; page size = 2KB
  - (In paging, processes can address more memory than available physical memory!)



Page #	V	P	Frame #
000			
001			
010			
011			
100			
101			
110			
111			

Frame #	Contents
00	2KB
01	2KB
10	2KB
11	2KB

Trap a Page Fault! (Need to load a page/frame into memory from disk)

Paging Example

- Consider the following scheme:
  - 8 frames and 8 pages (Size = 8 KB)
    - Addresses need at least 16 bits *(How do I know that? What's the address-format?)*
  - Four processes: A, B, C, and D.
  - Show the page tables after this schedule is run:
    - Load(A); Load(B); Load(C); SwapOut(B); Load(D); Load(B)
    - For now, assume ***all*** pages of a process get swapped in and out

- Say D's address space is currently 20,000 bytes.
  - What actually happens when D accesses its own (virtual) address, **`9217`**(= **`0010 0100 0000 0001`**)?

Paging Example

PageTable: Process **A**

Page #	V	P	Frame #
0 (000)			
1 (001)			
2 (010)			
3 (011)			
4 (100)			
5 (101)			
6 (110)			
7 (111)			

PageTable: Process **B**

Page #	V	P	Frame #
0 (000)			
1 (001)			
2 (010)			
3 (011)			
4 (100)			
5 (101)			
6 (110)			
7 (111)			

PageTable: Process **C**

Page #	V	P	Frame #
0 (000)			
1 (001)			
2 (010)			
3 (011)			
4 (100)			
5 (101)			
6 (110)			
7 (111)			

PageTable: Process **D**

Page #	V	P	Frame #
0 (000)			
1 (001)			
2 (010)			
3 (011)			
4 (100)			
5 (101)			
6 (110)			
7 (111)			

Physical Memory

Frame #	Content
0	
1	
2	
3	
4	
5	
6	
7	

Solution: After Running the Schedule

PageTable: Process A

Page #	V	P	Frame #
0 (000)	1		0
1 (001)	1		1
2 (010)	0		-
3 (011)	0		-
4 (100)	0		-
5 (101)	0		-
6 (110)	0		-
7 (111)	0		-

PageTable: Process B

Page #	V	P	Frame #
0 (000)	1		6
1 (001)	1		7
2 (010)	0		-
3 (011)	0		-
4 (100)	0		-
5 (101)	0		-
6 (110)	0		-
7 (111)	0		-

PageTable: Process C

Page #	V	P	Frame #
0 (000)	1		4
1 (001)	0		-
2 (010)	0		-
3 (011)	0		-
4 (100)	0		-
5 (101)	0		-
6 (110)	0		-
7 (111)	0		-

PageTable: Process D

Page #	V	P	Frame #
0 (000)	1		2
1 (001)	1		3
2 (010)	1		5
3 (011)	0		-
4 (100)	0		-
5 (101)	0		-
6 (110)	0		-
7 (111)	0		-

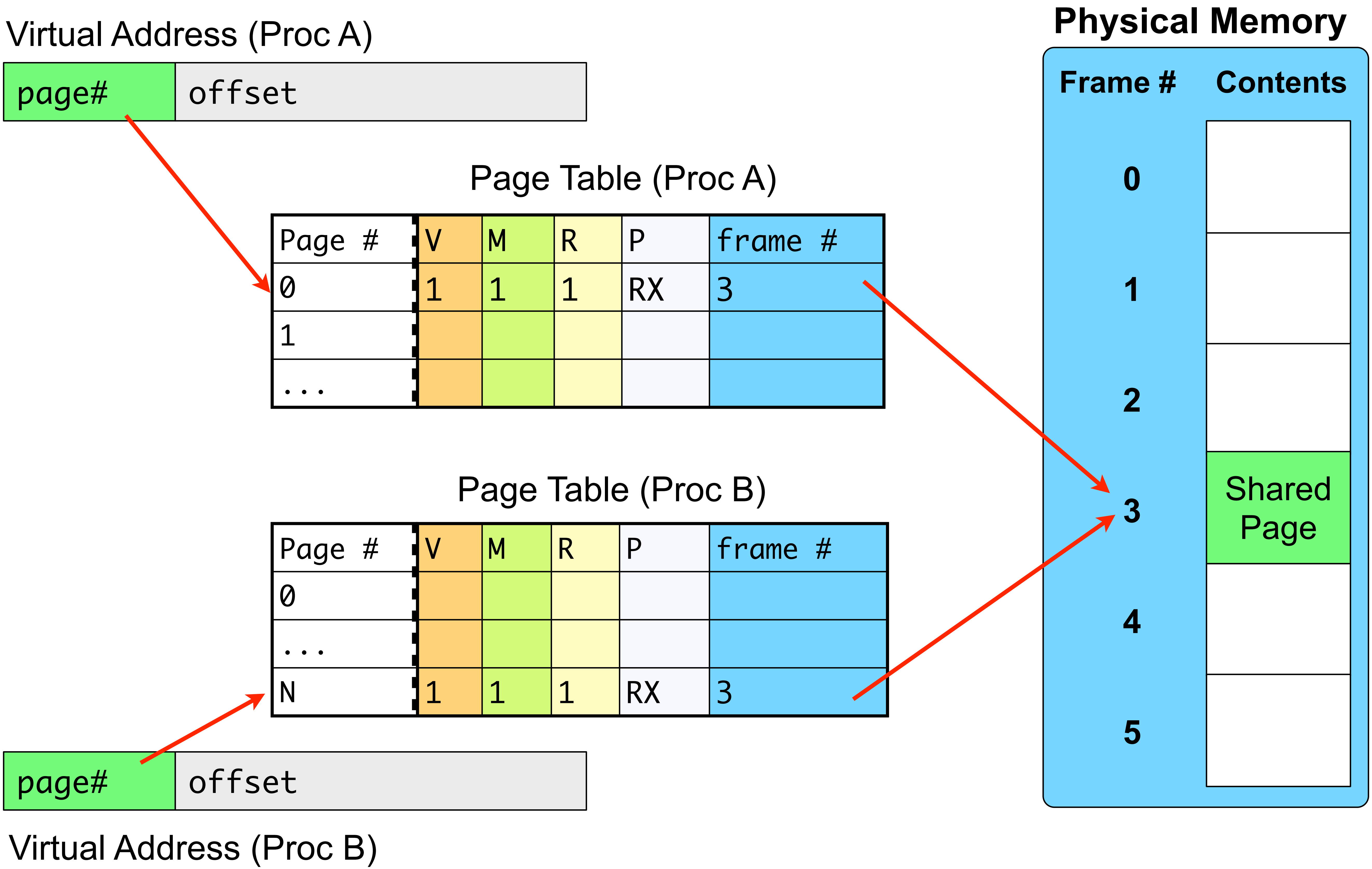
Physical Memory

Frame #	Content
0	A[0]
1	A[1]
2	D[0]
3	D[1]
4	C[0]
5	D[2]
6	B[0]
7	B[1]

Evaluation of Paging

- Pros
  - No External Fragmentation!
  - Separation of address spaces: Virtual vs. Physical
    - Relocatability!
    - Processes can address **more memory** than is *physically* available!
      - (How? Multiple pages can map to the same frame, and only one page is **valid**)
  - Memory Protection
    - Page tables enable controlled multiplexing of processes in RAM
  - Memory Sharing
    - Map pages from different processes to the same frame in RAM

Paging Makes Memory Sharing Easy



Virtual Address (Proc A)

page#	offset

Page Table (Proc A)

Page #	V	M	R	P	frame #
0	1	1	1	RX	3
1					
...					

Page Table (Proc B)

Page #	V	M	R	P	frame #
0					
...					
N	1	1	1	RX	3

page#	offset

Virtual Address (Proc B)

Physical Memory

Frame #	Contents
0	
1	
2	
3	Shared Page
4	
5	

Evaluation of Paging (Cont.)

- Cons
  - Still some internal fragmentation
    - Worse for larger sized pages
    - Not a huge deal… if we just keep pages small (4 KB - 16 KB in practice)
      - Page size is an OS configuration parameter

        (What’s the tradeoff between larger vs. smaller pages?)
  - In the best case, paging slows down memory access by a factor of 2!
    - Page tables are too large to fit into CPU registers.
    - *You effectively need **two** memory accesses to resolve each memory reference:*
    - *(1 to lookup the page in the page table + 1 to reference physical memory)*

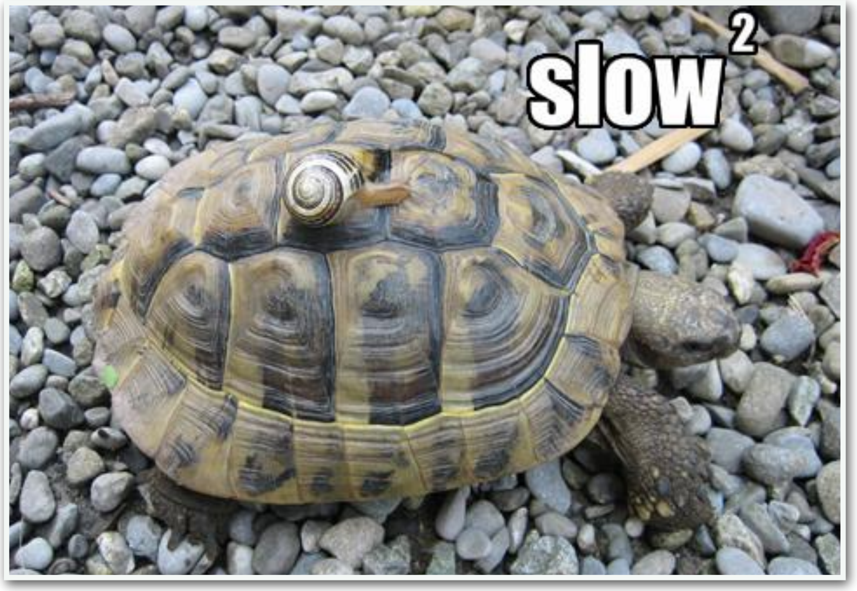
Goals for This Lecture...

- Motivation and Goals
- Towards Virtual Addressing
- Partitioning
- Segmentation
- Paging
  - Page Tables
  - Translation Lookaside Buffer (TLB)

Problem: Address Translation Is Slow!

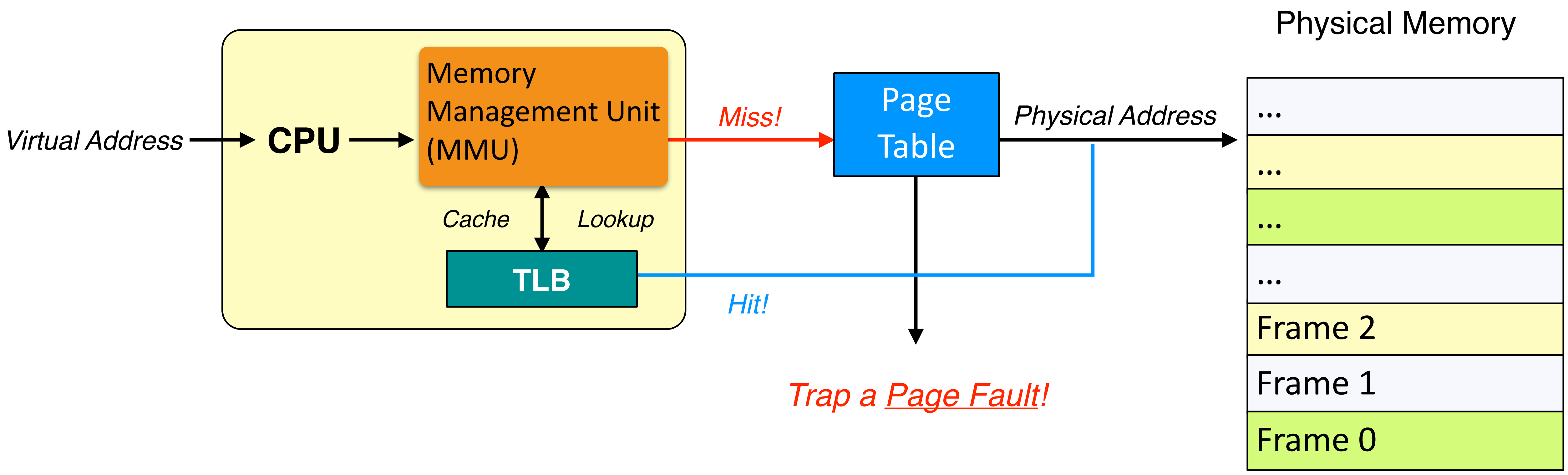
- New Problem: **S-L-O-W** Address Translations

- Every memory reference is at least twice as slow as before:
  - Given a virtual address:
    - Page Table lookup (1 memory access)
    - Retrieve the byte in physical memory (1 memory access)
    - If the page-table entry was invalid, then you have to first swap the page in from disk (even slower!)



Updated Translation Hardware

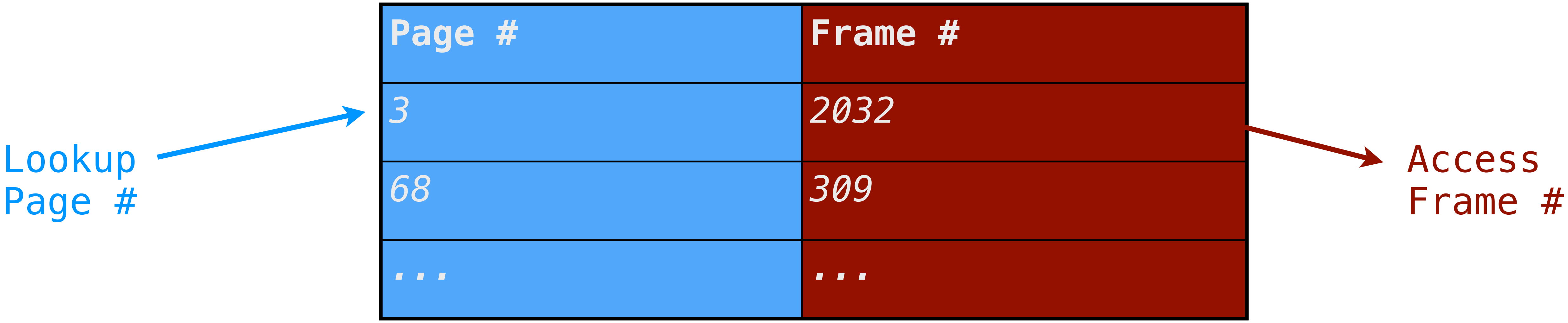
- Cache the address translations!
  - The *Translation Lookaside Buffer (TLB)* first appeared in IBM System 370 (just a fancy name for an "translation cache")



Translation Lookaside Buffer (TLB)

- TLBs are used to "cache" the address translations!
  - It lives on the CPU (the fastest cache on chip ~ faster than L1 cache!)
    - Its sole purpose is to map virtual addresses to physical addresses
  - A typical TLB stores 64 to 512 entries

Translation Lookaside Buffer (TLB)



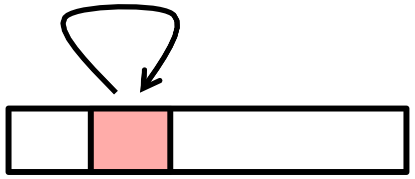
Lookup Page #

Page #	Frame #
3	*2032*
68	*309*
...	...

Access Frame #

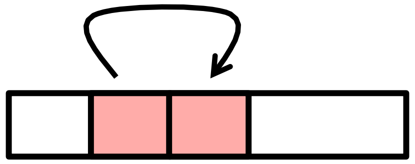
Would a TLB Really Work?

- Principle of Locality?
  - Do programs tend to use data and instructions with addresses near or equal to those used recently?



- "Temporal Locality"
  - Recently-used elements will likely be used again in the near future.

- "Spatial Locality"
  - Elements with nearby addresses will tend to be referenced in the near future.



Principle of Locality Example

- For translation caching to work, both **code** and **data** references need to observe the principle of locality.

- Both **temporal** and **spatial** locality of **code (instructions)** and **data** exhibited in this simple example.

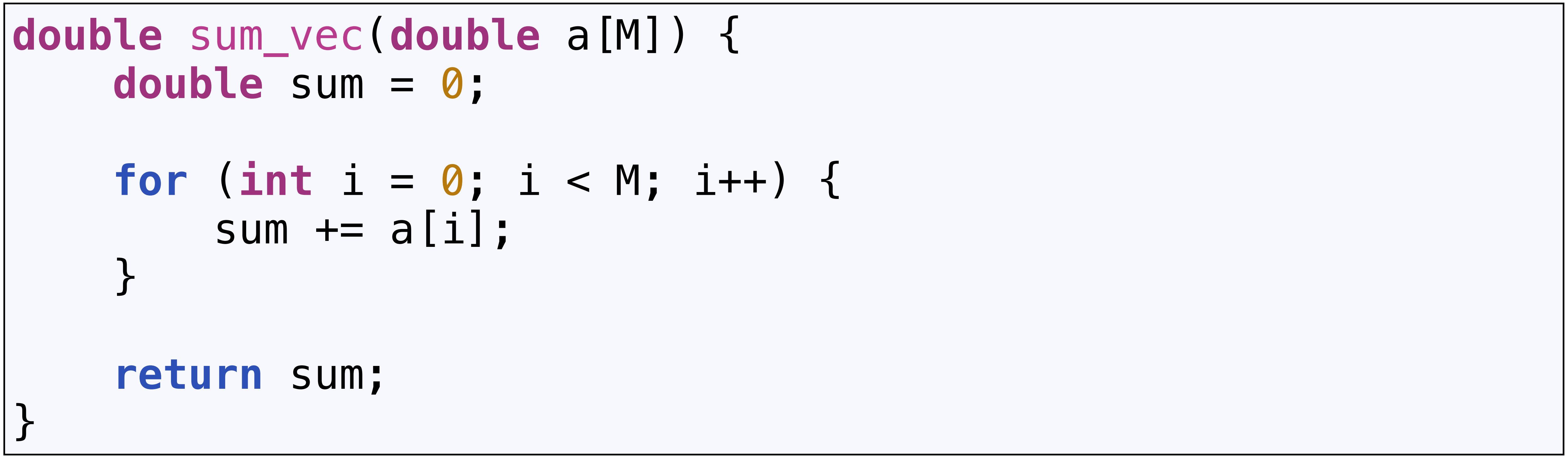
```c
double sum_vec(double a[M]) {
    double sum = 0;

    for (int i = 0; i < M; i++) {
        sum += a[i];
    }

    return sum;
}
```

Locality Example: Code

- Instruction (Code) Locality
  - When loaded, the compiled binary code gets placed inside pages
  - Repeated references of instructions in the *same page* is very likely.
    - e.g., Loops tend to dominate code
    - e.g., Calls to popular functions too



```
double sum_vec(double a[M]) {
    double sum = 0;

    for (int i = 0; i < M; i++) {
        sum += a[i];
    }

    return sum;
}
```

Locality Example: Data

- Data Locality
  - We know sizeof(double) is 8 bytes
  - If page size = 4KB, then a page can store 4K/8 = 512 doubles:
    - a[0] ... a[511] fit in one page
    - a[512] ... a[1023] fit in another page, and so on.
    - *Repeated references to sequential array elements are to the same page!*

```
double sum_vec(double a[M]) {
    double sum = 0;

    for (int i = 0; i < M; i++) {
        sum += a[i];
    }

    return sum;
}
```

Handling Context Switches

- Remember, due to virtual addressing,
  - Data at address X for Process A is different than for Process B
    - (Unless the page containing address X is shared)
  - **Problem:** TLB entries would be invalid after a context switch!

- Two options:
  - Small TLB: Just clear it out on a context switch
  - Larger TLB can store more entries. May not want to flush all.
    - Add address-space identifiers (*e.g.*, PID) to the TLB.

Measuring Memory Access Performance

- We have two access paths for an address reference:
  1. Address translation is found in TLB *(TLB hit)*
  2. Or, not found in TLB, need a Page-Table Walk *(TLB miss)*

- Definition: *Effective Memory Access Time (EAT)*
  - $EAT = hT_{hit} + (1 - h)T_{miss} + mem\_access\_time$
    - h = TLB hit rate
    - T_{hit} = TLB lookup time
    - T_{miss} = TLB lookup time + page-table lookup time

(How much can a TLB improve EAT?)

Example

- Assume:
  - Memory Access Time = 200 ns
  - Page Table lookup requires 1 additional memory access per translation

- What's the EAT without a TLB?
- What's the EAT with a TLB with following specs?
  - TLB hit rate (h) = 0.75
  - TLB lookup time (T_{hit}) = 25 ns
- If we can only absorb a 20% slowdown of memory access, what does the TLB hit rate have to be?

In Summary... (Cont.)

- Limitation: We have less physical memory than virtual address space
  - Currently: *All* segments or pages of a process must be in physical memory for the process to execute
  - Context switching and swapping slow because of it!

- Do all pages need to be in physical memory for process to run?
  - At any point in time, only part of the code is actually executing
  - *Next time: Virtual Memory*

Administrivia 4/14

- Reminders:
  - Hwk 7 (Due 4/18 Friday)
- Last time ...
  - Banker's algorithm
- Today
  - Start memory management (Chap 9)
  - Desirable features of memory management
  - Loaders: Absolute loading; Static Relocation; Virtual Addresses

Administrivia 4/18

- Reminders:
  - Hwk 7 (extension - due Monday)
- Last time...
  - Virtual addressing *(Why do we even need this?)*
  - Fixed partitioning
  - Variable partitioning ("Base & Bounds")
  - Segmentation
- Today
  - The modern approach: Paging

Administrivia 4/21

- Reminders
  - Homework 8 posted (due 5/9)
- Last time...
  - Paging: the modern memory management solution
  - Split address space up into fixed size chunks (called "pages")
- Today:
  - Review of paging
  - Updated address translation scheme
  - Translation lookaside buffer (TLB)