

# CSC 369

## Operating Systems

---

### Lecture 6:

### Memory management

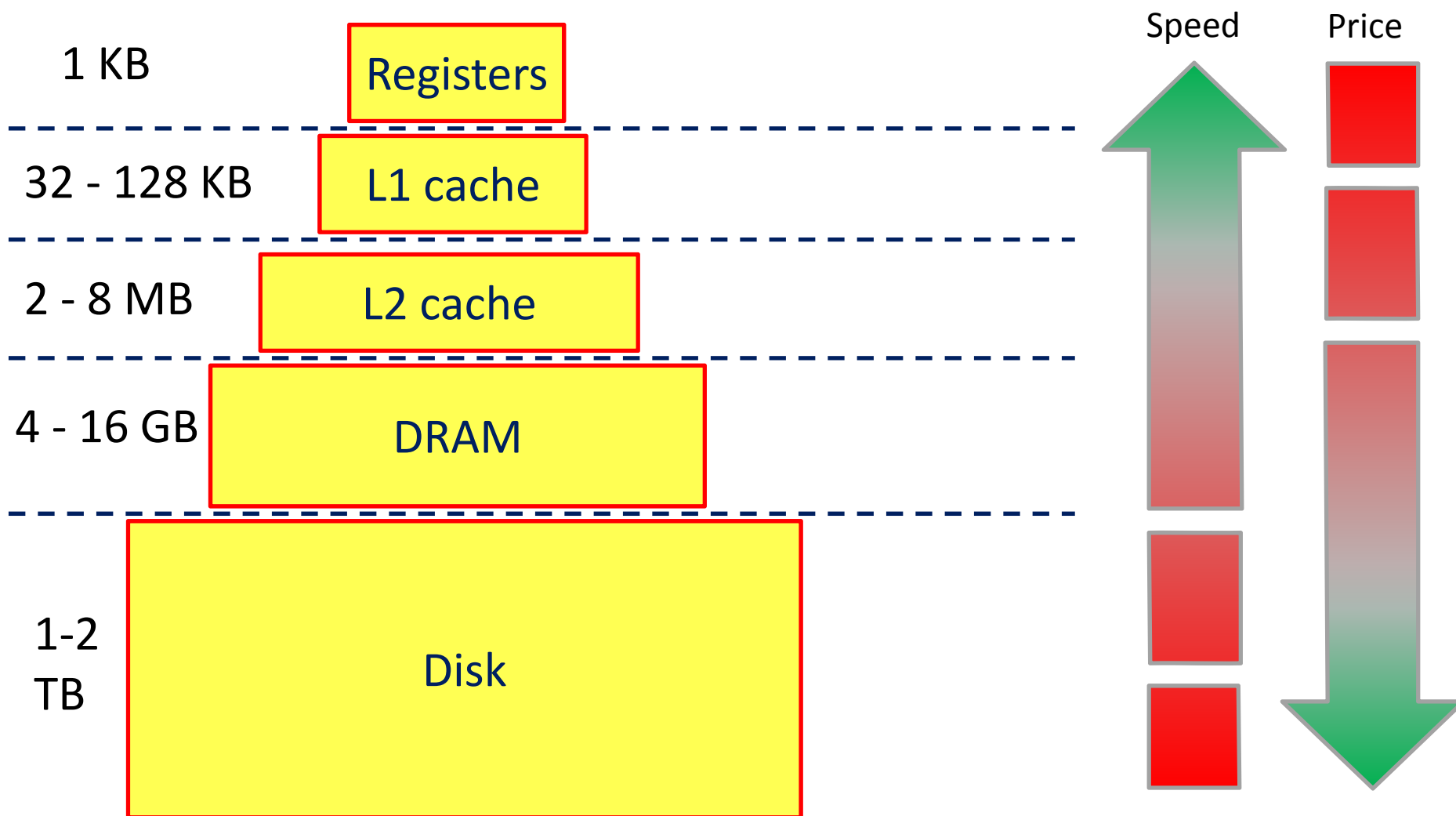
Bogdan Simion



University of Toronto, Department of Computer Science



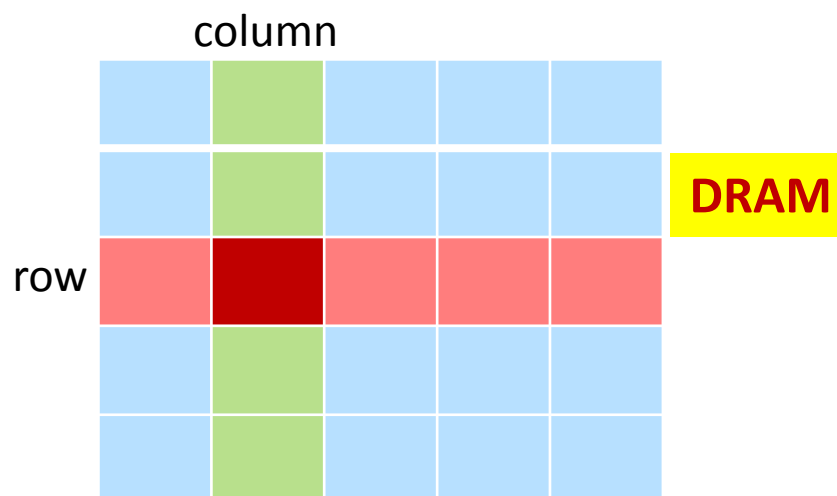
# Memory hierarchy





# But that's not all

- Sequential data access speeds on disk/DRAM differ



- Accessing the 1st byte:  $\sim 100,000$  times slower than the next bytes; DRAM only  $\sim 4$  times slower
- What does that mean?*
  - Better for large sequential data, bad for small random data



# Bottom line

- Disk is cheap, but waaaaay slow!
  - We want to avoid going to disk at all cost!
- DRAM acts like a cache for the disk
  - Performance depends on efficient use of DRAM!
- How can the OS help you use DRAM?
  - Virtual memory!



# Virtual memory

1. Only a limited amount of physical memory
  - Must use efficiently
2. Every active process needs memory
  - Must provide the illusion of “infinite” memory to each process
3. Physical memory is access by multiple processes
  - Must ensure data privacy! Might want to allow sharing!

=> Our goals:

- 1. Efficiency**
- 2. Transparency**
- 3. Protection and sharing**



# Goals: Efficiency

- Make use of memory wisely
- Basic idea:
  - Some portions are in DRAM
  - Some portions are stored on disk
  - Transfer data back and forth, “as needed”
- *Which portions should be in DRAM?*



# Goals: Transparency

- Data moves back and forth between RAM and disk
- Programmer should not worry if chunks of program/data are in memory or disk

**How do we create the “illusion” of having more memory than DRAM?**



# So, .. Transparency

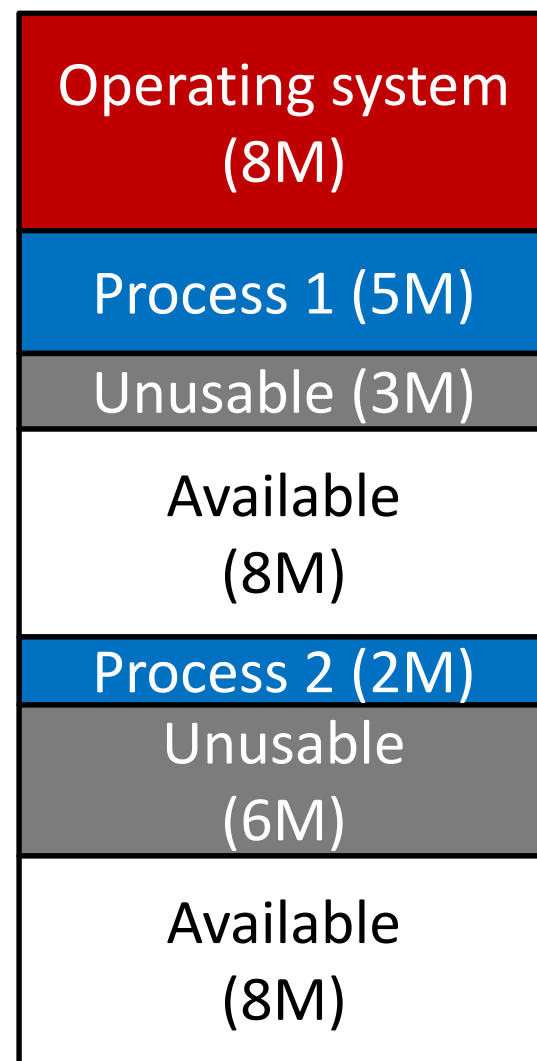
- Give each process its own view of memory
  - Large contiguous address space, starting at address 0
  - Simplifies memory allocation
- Decouple the data layout from where the data is actually stored in physical memory
- *Why do we want this separation?*
- *How would you manage physical memory?*
- *How do we find a location in physical memory to put our process's memory?*





# Fixed partitioning

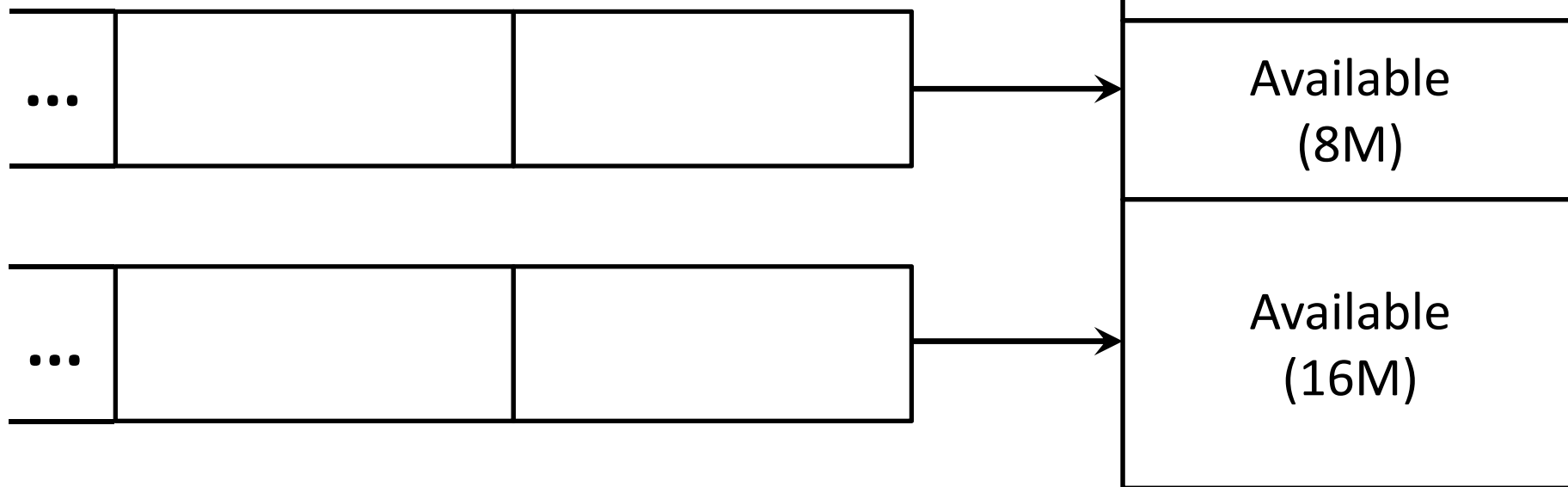
- Each process gets a fixed partition to use
  - Divide physical memory into 8M regions
  - OS occupies a separate partition
  - Each process is granted one of these
- See any problems?
  - If process is smaller than partition, this wastes memory (**internal fragmentation**)
  - If program needs more memory than the partition size, programmer has to deal with that (**overlays**)
  - Number of partitions is limited => limits the number of active processes!





# Fixed partitioning

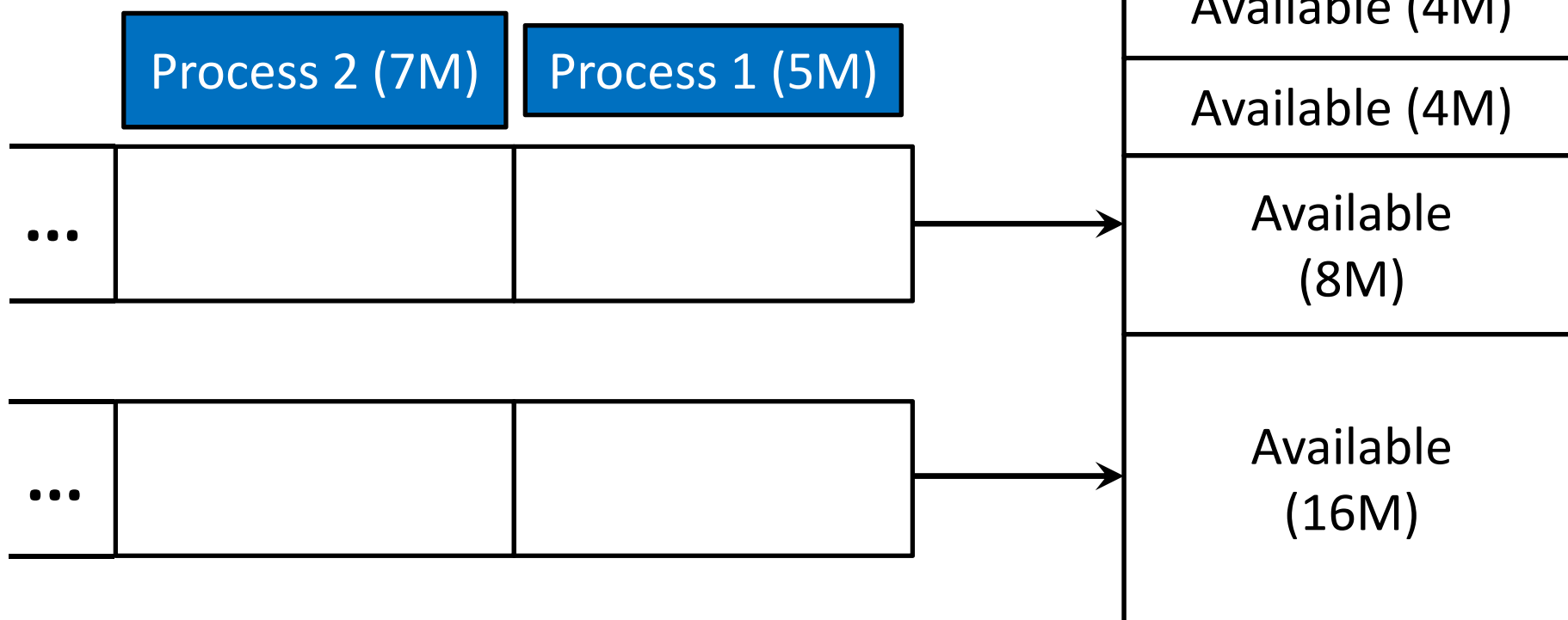
- Unequal-sized partitions:
  - Queue-per-partition
  - Enqueue process to *smallest partition* in which it will fit





# Placement example

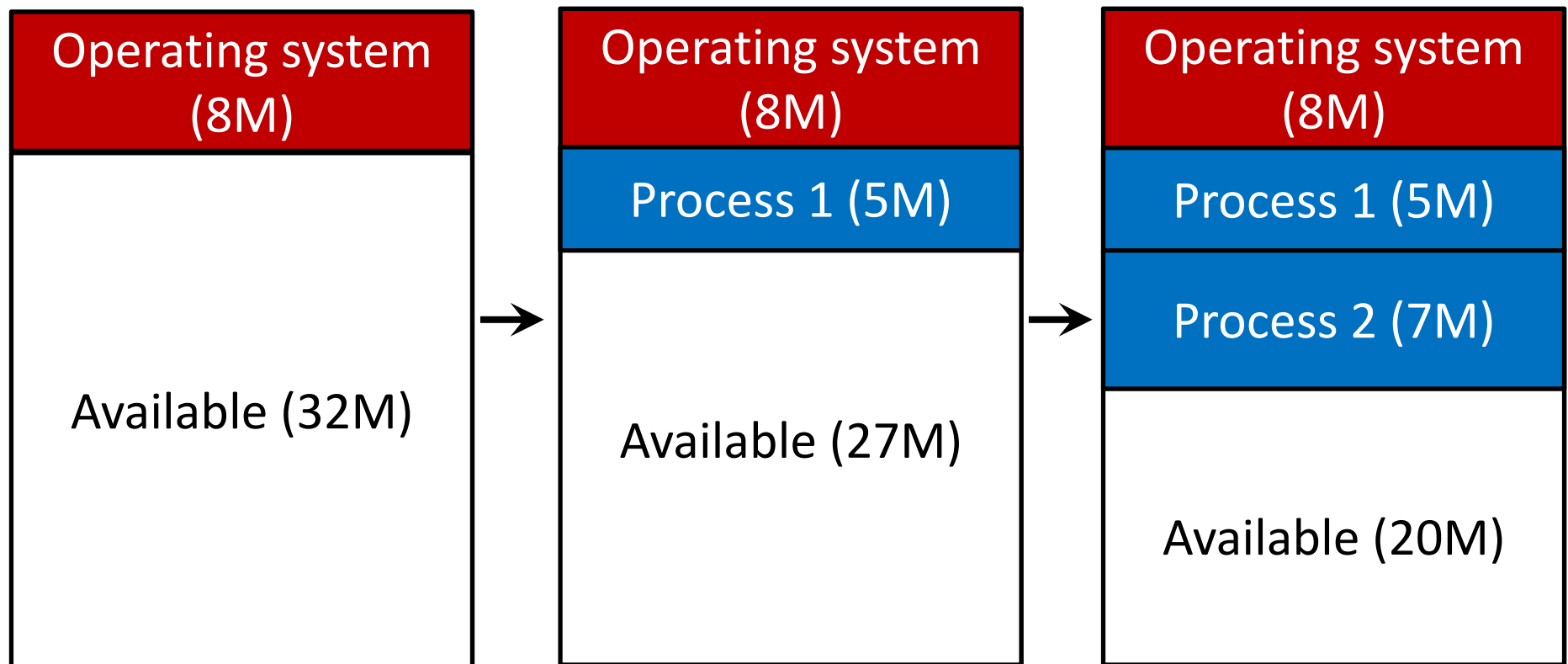
**Process 1 and Process 2 fit in the same partition. With smallest-partition policy, both must share the 8M partition while the 16M partition goes unused.**





# Dynamic partitioning

- Partitions vary in length and number over time
- When a process is brought into memory, a partition of exactly the right size is created to hold it





# More dynamic partitioning

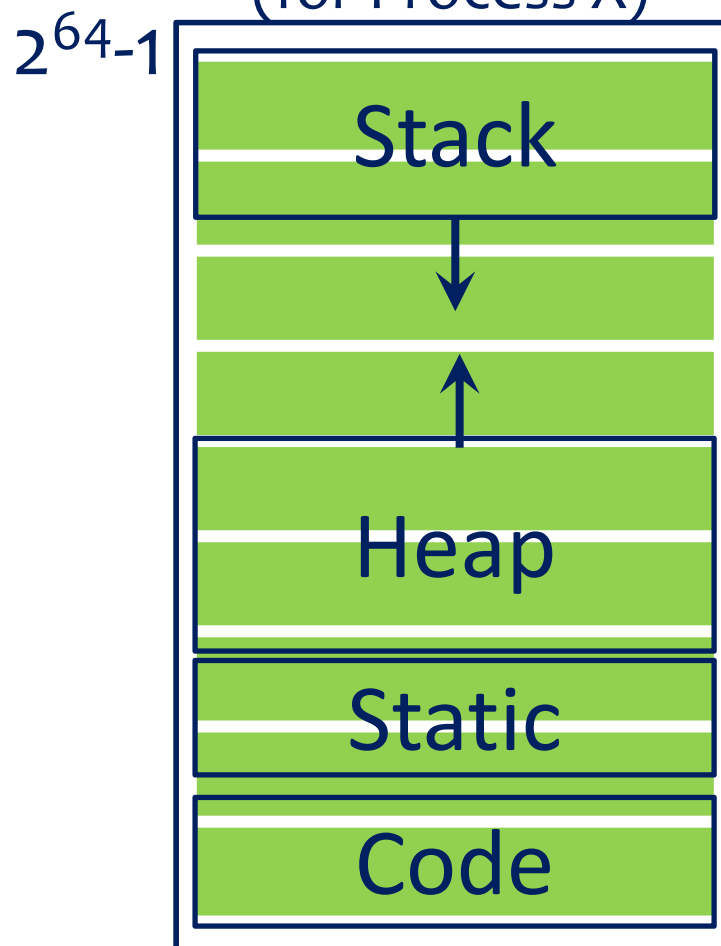
- P2 gone, “hole” is created
  - Some holes may be too small to be re-used
  - This is called **external fragmentation**
- OS may move processes around to create larger chunks of free space
  - E.g. Process 3 immediately after P1
  - This is called **compaction**
  - Requires processes to be **relocatable**
- Need to know maximum size of process at load time
  - Can we know?
  - Can we grow partitions at runtime?
  - Can we share data between processes?

Operating system (8M)
Process 1 (5M)
Process 2 (7M)
Process 3 (2M)
Process 4 (2M)
Available (16M)



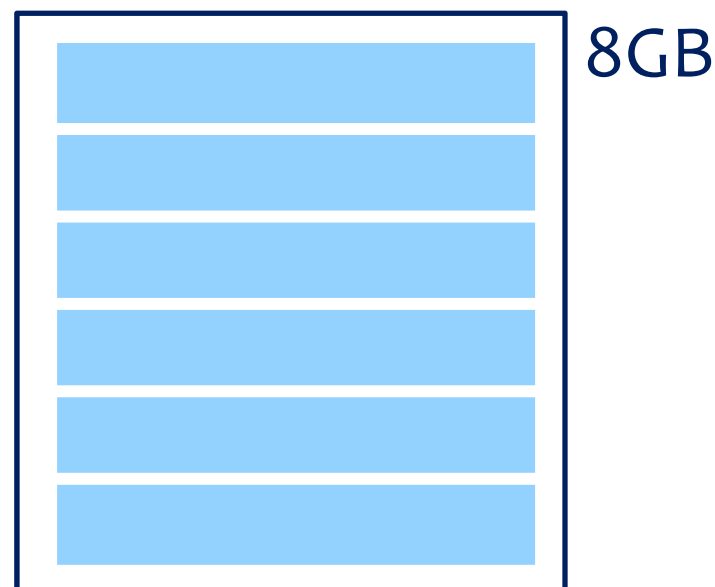
# VM to the rescue: use Paging instead!

Virtual Address Space  
(for Process X)



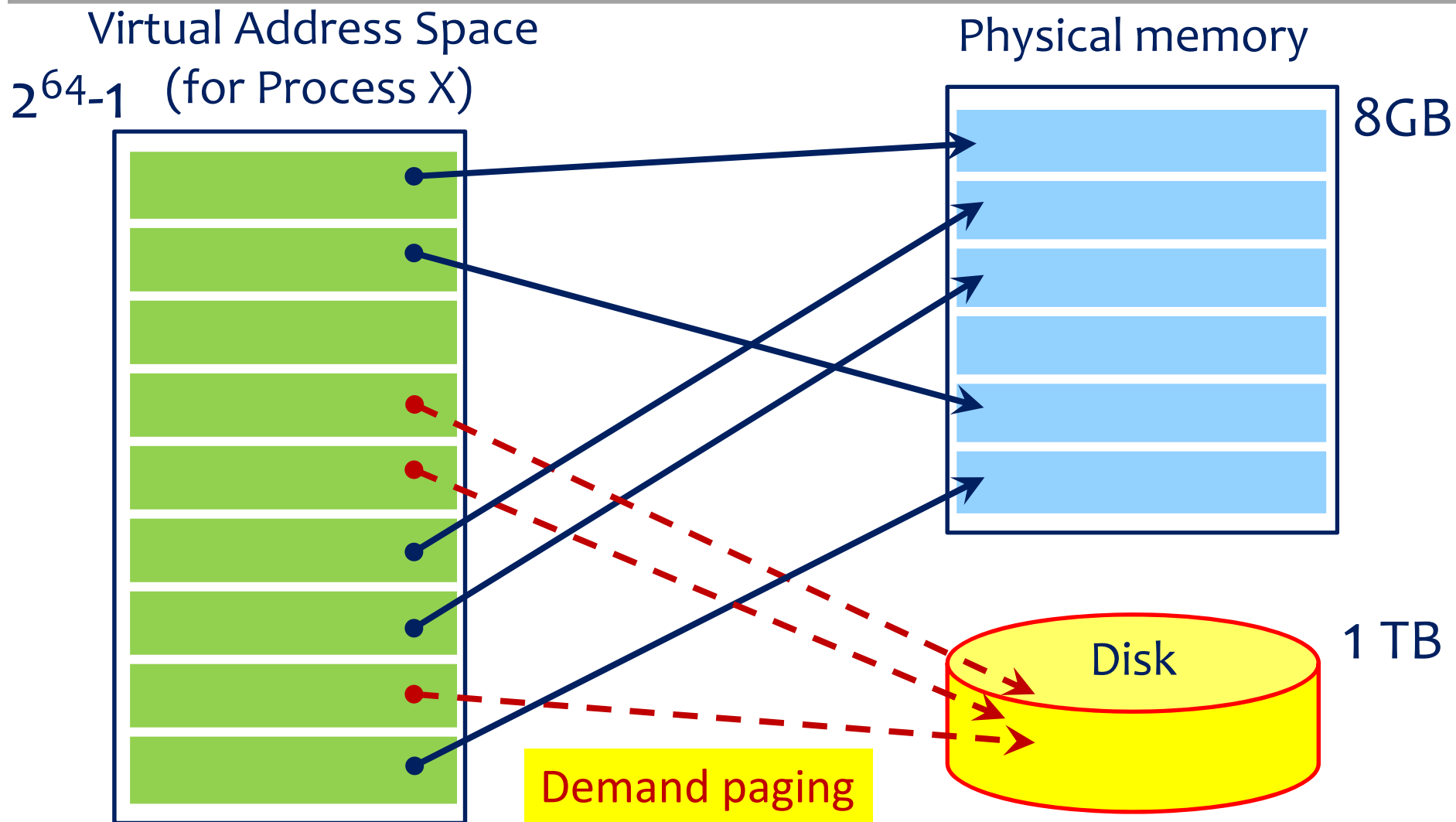
- Decouple address space completely from actual physical data location
- Split both virtual and physical memory in same-size chunks (**pages**)

Physical memory



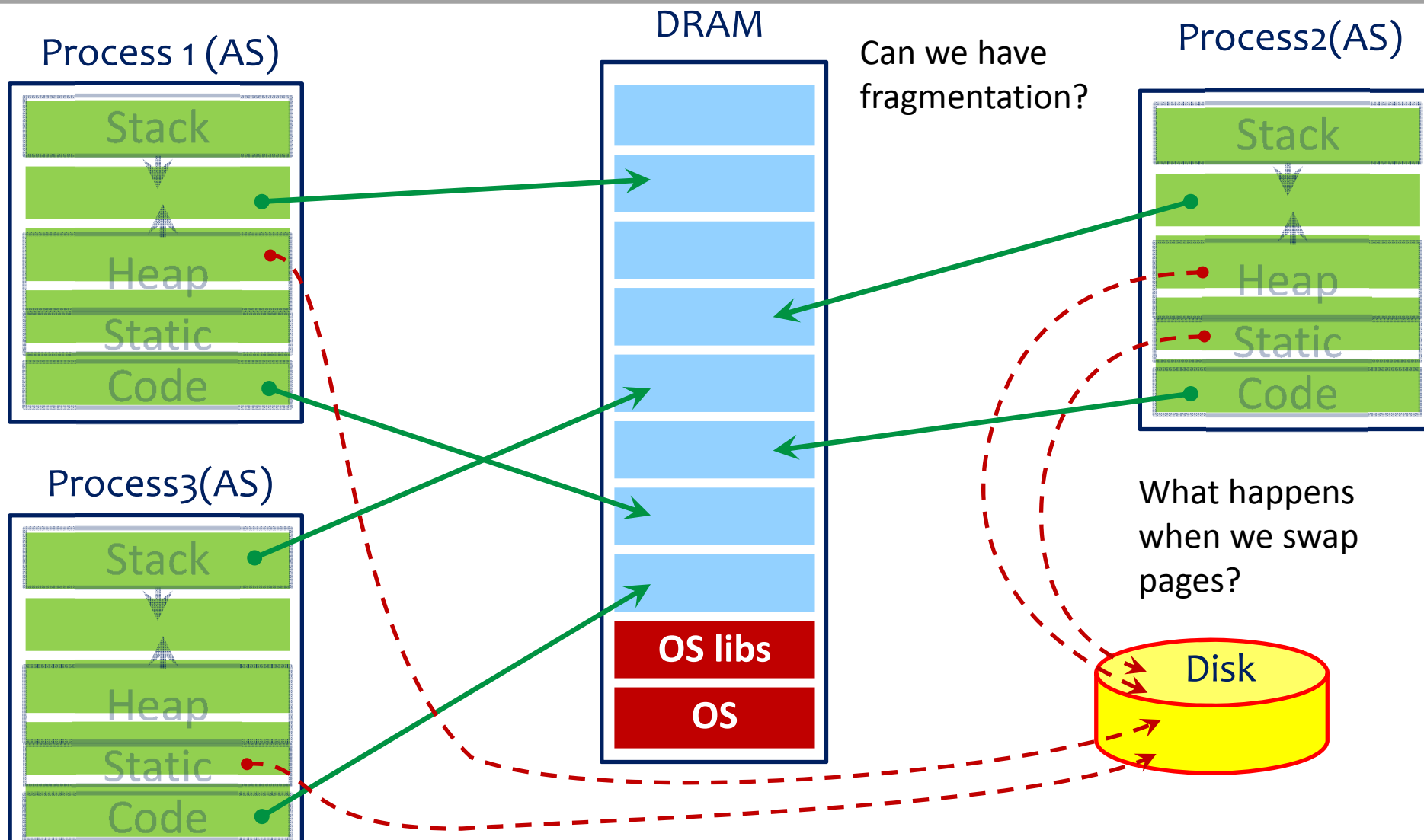


# Mapping virtual memory





# Processes co-exist in memory







# What we've learnt so far..

- Each process gets the illusion of its personal address space ( $0 \rightarrow 2^{64}-1$ )
- Each virtual page can be mapped to any physical page
- Data is either in memory or on disk
  - We must bring data in memory, as needed
  - Who keeps the correspondence between a virtual address and a physical address?

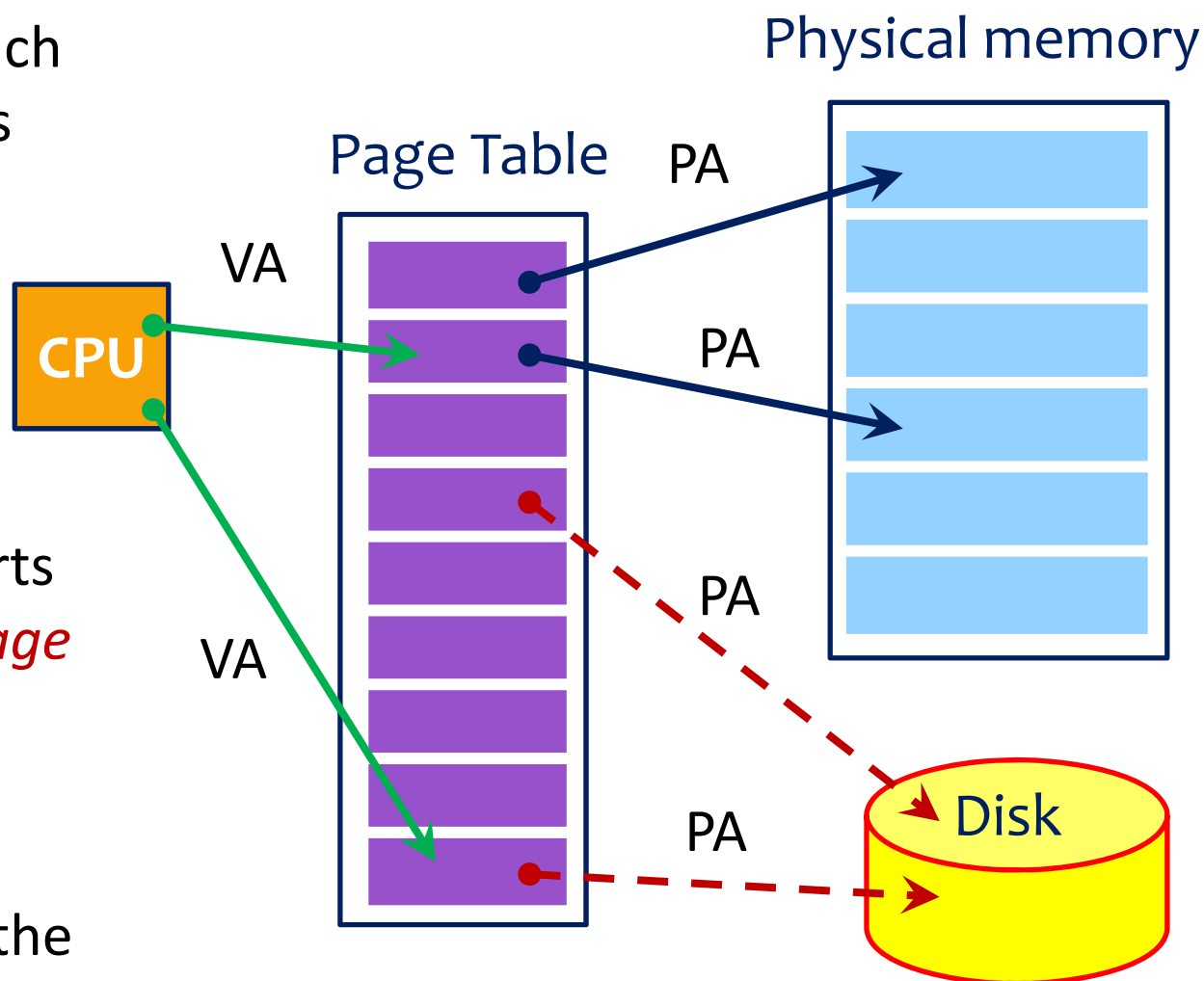


# Where does the translation happen?

Multiple processes => Each process needs to have its address space pages *translated* to “real” (physical) addresses

Hardware (MMU) converts VAs into PAs using the *Page Table* (an OS-managed lookup table)

Each process running in the OS has *its own* Page Table.





# Page faults

- What if the page we want is not in memory?
  - Page table entry indicates that the page is not in memory
  - Causes a **page fault** (basically, like a cache miss)
- How do we handle a page fault?
  - OS is responsible for loading page from disk
  - Process stops until the data is brought into memory
  - Page replacement policy is up to the OS (next time..)



# How come this is not way too slow?

- Processes reference pages in **localized patterns!**
- **Temporal locality**
  - Locations referenced recently likely to be referenced again. Examples?
- **Spatial locality**
  - Locations near recently referenced locations are likely to be referenced soon. Examples?
- Although cost of paging is high, if it's infrequent enough (due to locality), it's acceptable
- **What's the worst we can do in our programs?**



# Exercise 1

```
#define N 409600000
#define M 100000
#define PAGE_SIZE 4096
```

```
int main(int argc, char **argv)
{
    int i;

    int *X = malloc(N * sizeof(int));
    if(!X) return 1;

    for (i = 0; i < M; i++)
        X[i] = 0;

    return 0;
}
```

```
int main(int argc, char **argv)
{
    int i;

    int *X = malloc(N * sizeof(int));
    if(!X) return 1;

    for (i = 0; i < M; i++)
        X[i * PAGE_SIZE] = 0;

    return 0;
}
```

**Try timing each one:**

**\$ time ./program**

**Make sure you check that your system's page size is 4K too:**

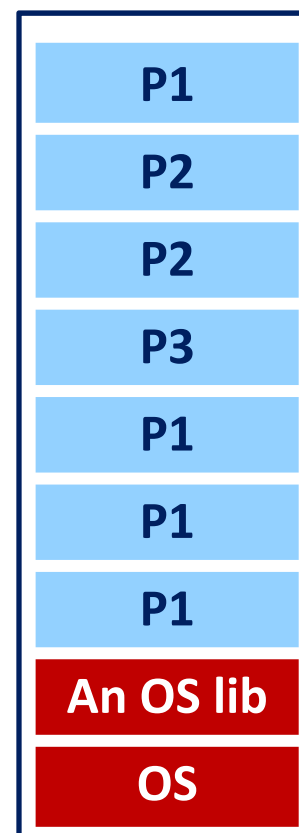
**\$ getconf PAGE\_SIZE**



# Goals: Protection and sharing

- Processes co-exist in memory
- Processes should not access other processes' memory
  - Must protect process address spaces!
  - Implies we need access rights for pages (like file permissions – R/W/X)
- Privileged OS data shouldn't be accessible to users
- In some cases, **sharing** may be desirable
  - Need to control what sharing is allowed

Physical  
memory





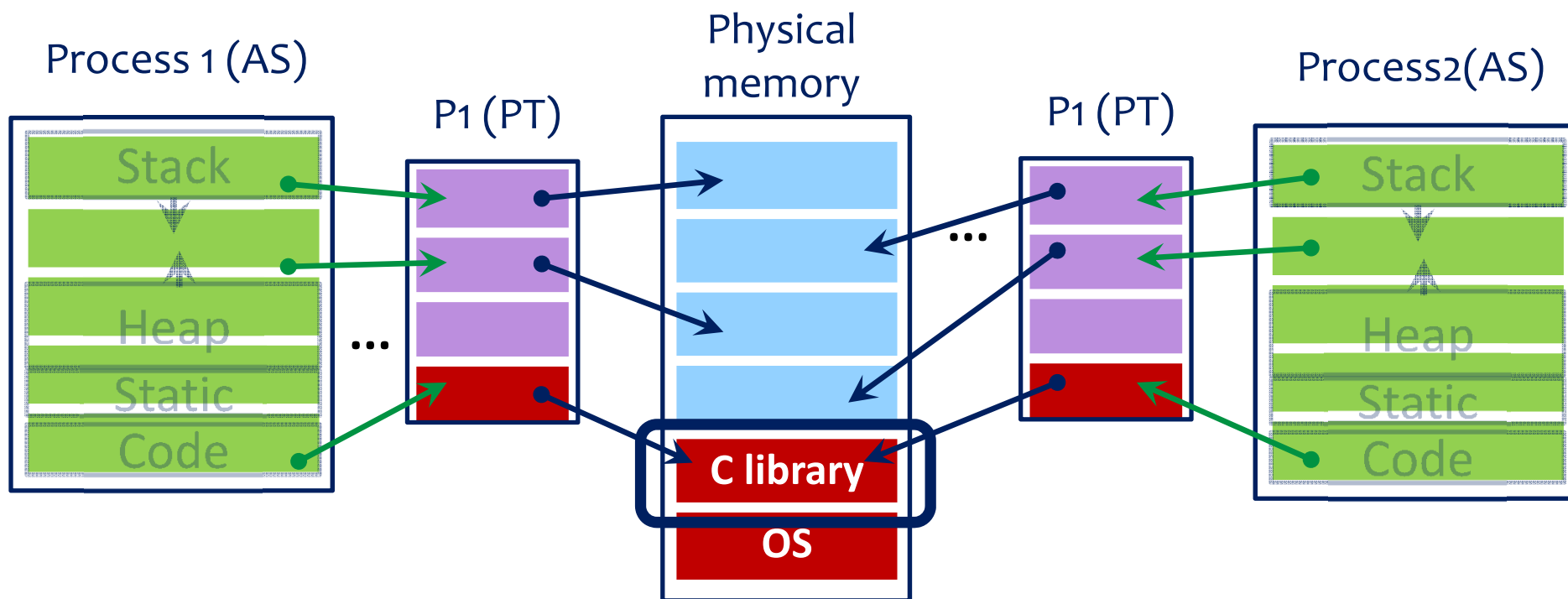
# VM enforces protection

- Address spaces are per-process, completely isolated
- Access rights are kept in the page tables
  - Hardware enforces protection, OS is called when violation happens
  - Can I modify my own page tables?
    - No! Page tables are in protected OS memory (only OS can modify them)
- Avoid leaked information from deallocated pages
  - Programmer should not have to zero out each page on dealloc
  - OS ensures that newly allocated pages are zero-ed out. How?
    - Use a “zero-ed” page and give the VA of that page (*Copy-on-Write*)
    - Where is this COW concept also being used?



# Sharing pages

Allow sharing pages, like  
read-only library code



**VM simplifies sharing code and data  
between multiple processes in the system!**





# Summary

- Virtual memory allows mechanisms for:
  - *Efficient* use of physical memory
  - *Transparent* use of physical memory
  - *Protection* and *sharing* of physical data between processes



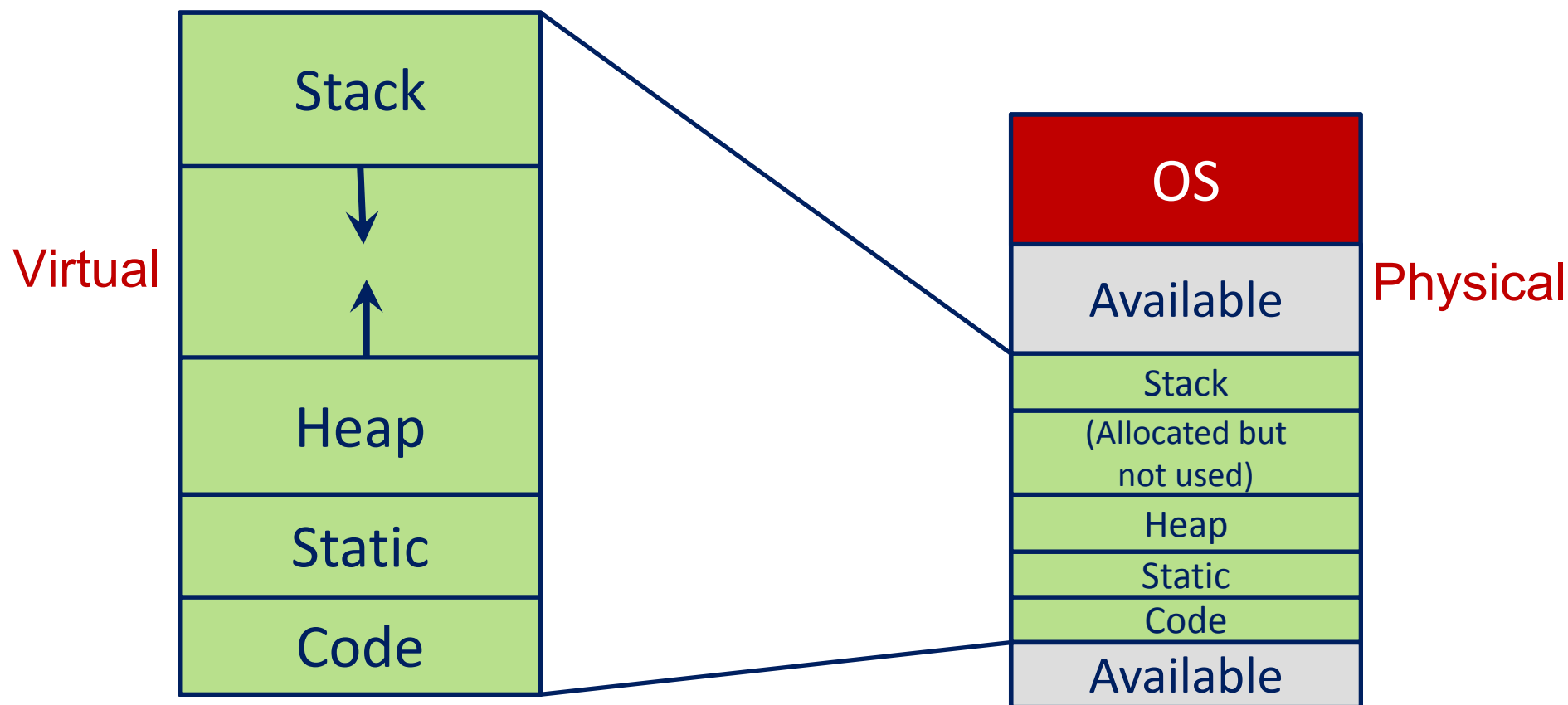
# Recap

- Addresses in program must be translated (**mapped, bound**) to real (**physical**) addresses
- Looked at different ways of allocating physical memory to processes
  - Fixed partitioning
  - Dynamic partitioning
  - Paging
- Keep in mind the goals:
  - **Efficiency,**
  - **Transparency,**
  - **Protection and Sharing**



# Key theme: how to translate!

- How does the OS translate **the programmer's view** to (real) **physical addresses**?
- Let's assume, for now, that we can actually fit the address space in physical memory!





# Where does my program go?





# Overview of requirements

- **Relocation**

- Programmers don't know what physical memory will be available when their programs run
- Medium-term scheduler may swap processes in/out of memory; need to be able to bring it back in to a different region of memory
- This implies some sort of address translation

- **Logical organization**

- Machine accesses/addresses the memory as a one-dimensional array of bytes
- Programmers organize code in modules
- Need to map between these views

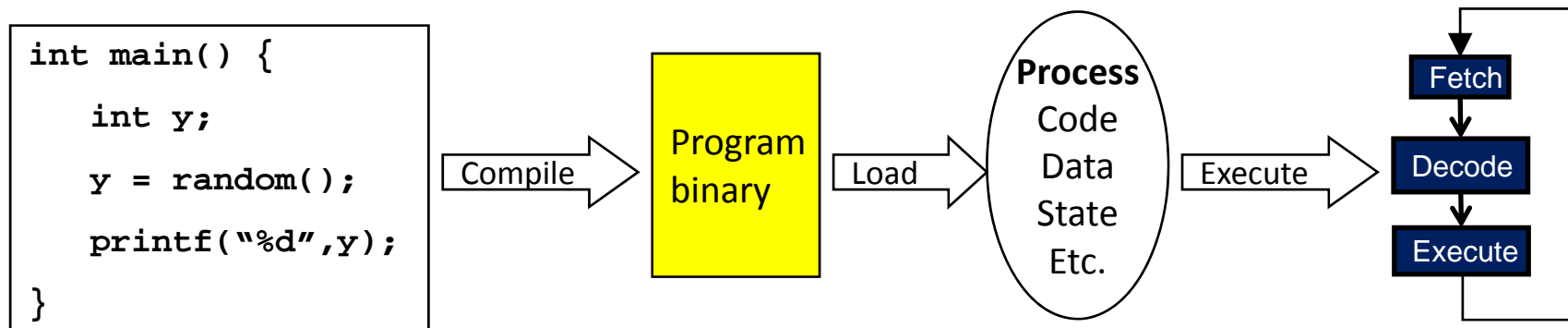
- **Physical organization**

- Memory and Disk form a two-level hierarchy, flow of information between levels must be managed
- Recall that CPU can only access data in registers or memory, not disk



# Address binding

- Programs must be in memory to execute
  - Program binary is **loaded** into a process's **address space**
  - Needs memory for code (instructions) & data
- Addresses in program must be translated (mapped, bound) to real (physical) addresses
  - Programmers use **symbolic** addresses (i.e., variable names) to refer to memory locations
  - CPU fetches from, and stores to, real memory addresses
- **Address translation is the process of linking variable names to physical locations**





# When are addresses bound?

- Option 1: **Compile time**
  - Must know what memory the process will use, during compilation
  - Called *absolute code* since binary contains real addresses
  - No relocation is possible
  - e.g., old MS-DOS .COM programs, simple embedded systems
- Option 2: **Load time** (aka *static relocation*)
  - Compiler: translates (binds) symbolic addresses to *logical, relocatable* addresses within compilation unit (source file)
  - Linker: translates addresses from obj files to *logical, absolute addresses* within executable
    - Takes care of references to symbols from other files/modules
  - Loader translates logical absolute addresses to *physical* addresses when program is loaded into memory



# Load-Time Binding Example

program1

4092	...
...	...
12	ADD
8	MOV
4	...
0	JMP 12

program2

4092	...
...	...
12	SUB
8	CMP
4	...
0	JMP 8

Content of binary  
file on disk

- Programs can be loaded to different address when they start, but cannot be relocated later. Why not?

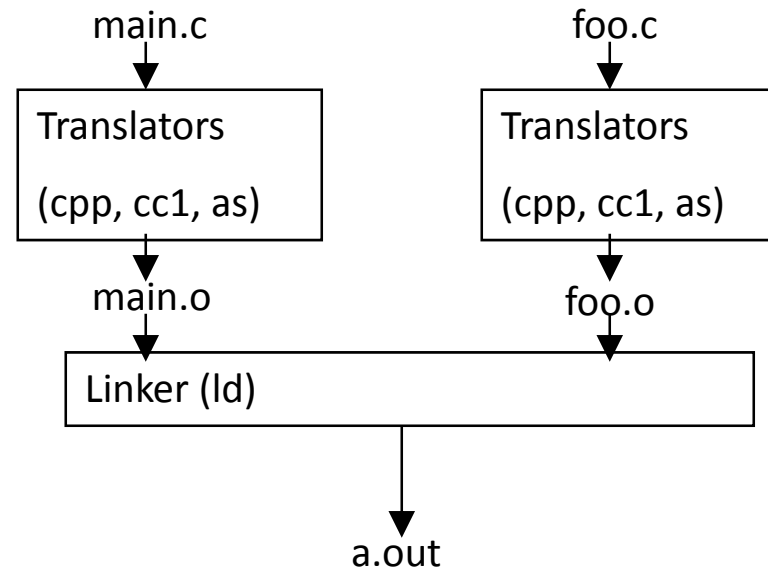
program1 (in memory)	16380	...
	...	...
	8204	ADD
	8200	MOV
	8196	...
program2 (in memory)	8192	JMP 8204
	8188	...
	...	...
	4108	SUB
	4104	CMP
	4100	...
	4096	JMP 4104





# A better plan

- Bind addresses at execution time (*dynamic relocation*)



- Executable file, `a.out`, contains **logical addresses** for entire program
  - Translated to **a real, physical address during execution**
  - Flexible, but **requires special hardware** (as we will see)



# Dynamic relocation

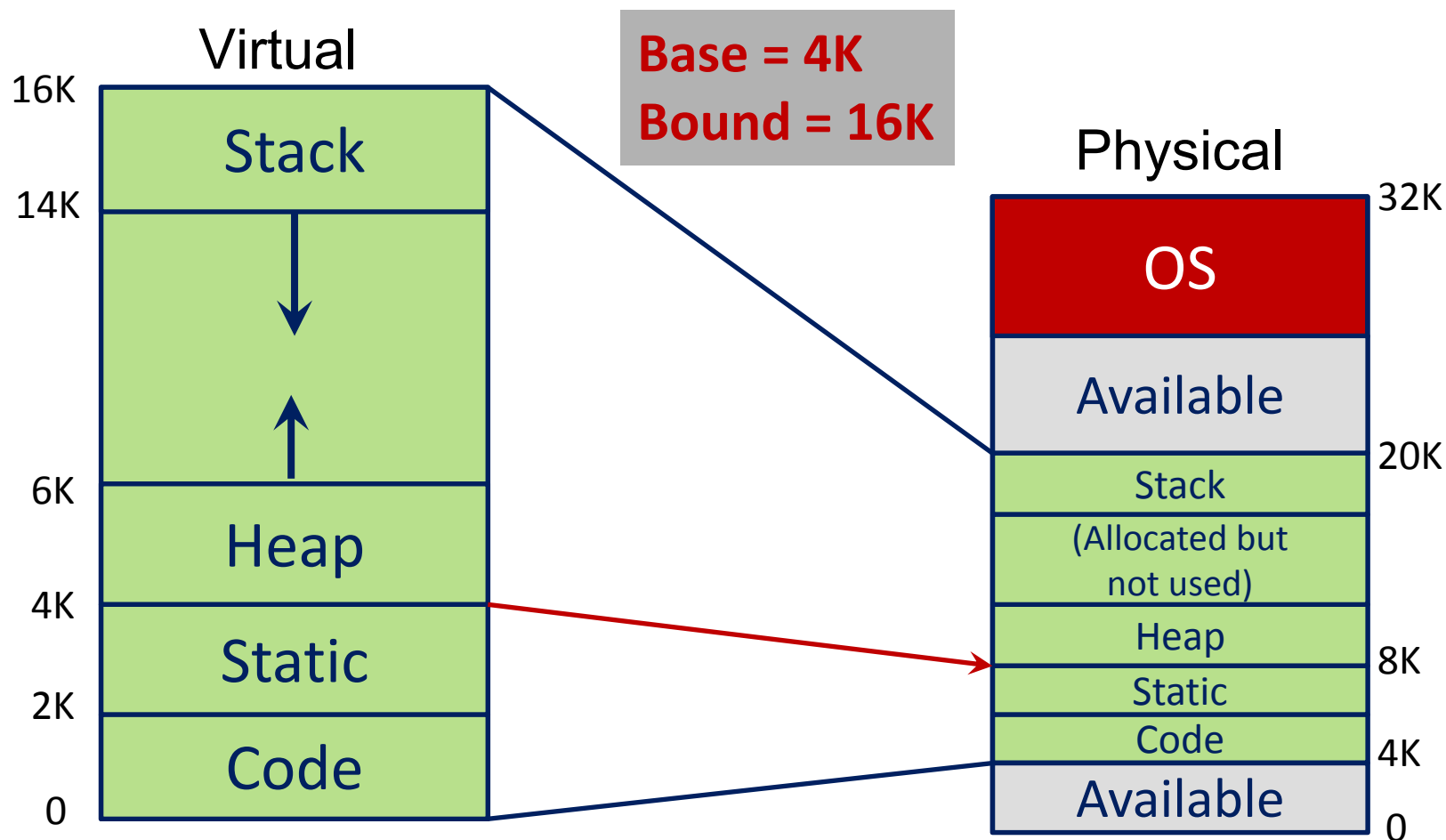
- Hardware support: 2 registers called **base** and **bound** on the MMU
  - The executable simply stores virtual addresses (starting at 0)
  - When a program starts to run, the OS decides where in physical memory to place its address space
    - Step 1: Set the base register
    - Step 2: Physical address can be translated as the following
      - $\text{physical address} = \text{base} + \text{virtual address}$
    - Step 3: Profit!
- => Instructions in the executable do NOT need to be modified!
- Why do we need the bound register?
    - Ensures that we don't access outside a process' address space
  - MMU only has one base and bound register
    - But ... we have N processes
    - Base and bound get saved in the PCB when we do a context switch





# Example

- Virtual address space relocated into physical memory





# Problems?

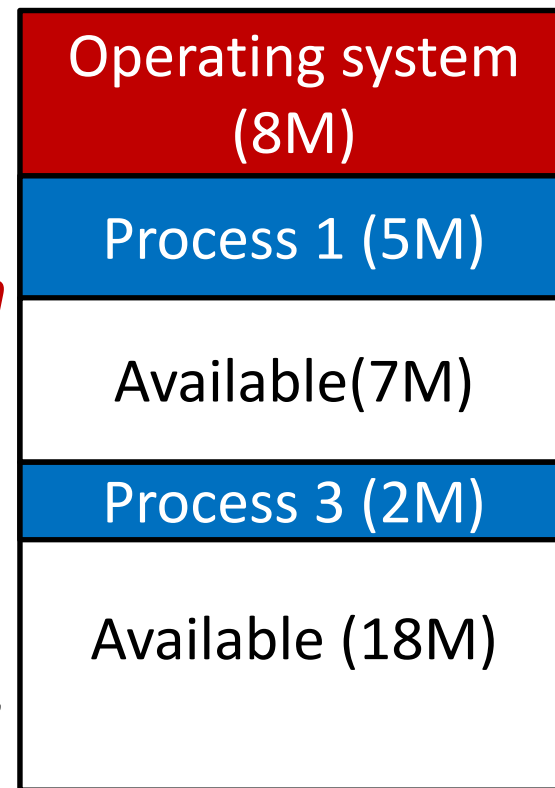
- OS must be able to find free space to relocate new processes' address spaces
  - Possible problems?







# Recall dynamic partitioning?

- As processes come and go, “holes” are created
  - Problem: Some holes may be too small to be re-used, aka **external fragmentation**
- Solution: OS may move processes around to create larger chunks of free space
  - Aka **compaction**
  - Expensive to shuffle many of them around, in order to find space for a new process





# Placement Heuristics

- Compaction is time-consuming and not always possible
- Can we reduce the need for it? 
- How about being careful about how memory is allocated to processes over time?
- Smart freelist management algorithms! 
  - **First-fit** - choose first block that is large enough; search can start at beginning, or where previous search ended (called next-fit)
  - **Best-fit** - choose the block that is closest in size to the request
  - **Worst-fit** – choose the largest block
  - **Quick-fit** – keep multiple free lists for common block sizes



# Comparing Placement Algorithms

- **Best-fit**
  - left-over fragments tend to be small (unusable)
  - In practice, similar storage utilization to first-fit
- **First-fit**
  - Simplest, and often fastest and most efficient
  - May leave many small fragments near start of memory that must be searched repeatedly
  - Next-fit variant tends to allocate from end of memory
    - Free space becomes fragmented more rapidly
- **Worst-fit**
  - Not as good as best-fit or first-fit in practice
- **Quick-fit**
  - Great for fast allocation, generally harder to coalesce



# Going back to Relocation

- Swapping and compaction require a way to change the physical memory addresses a process refers to
  - can we repeat address translation as done at initial load?
- Really, need **dynamic relocation** (aka *execution-time binding of addresses*)
  - process refers to *relative* addresses, hardware translates to physical address as instruction is executed
- Let's recall minimum hardware requirements
  - All memory used by process is *contiguous* in these methods





# Recap: Hardware for Relocation

- Basic idea: add relative address to the process's starting address (base address) to form real, or physical, address
  - check that address generated is within process's space
- 2 registers, "base" and "limit"
  - When process is assigned to CPU (i.e., set to "Running" state), load base register with starting address of process
  - Load limit register with last address of process
  - On memory reference instruction (load, store) add base to address and compare with limit
  - If compare fails, trap to operating system
    - if ( $\text{addr} < \text{base} \mid \mid \text{addr} \geq (\text{base} + \text{limit})$ ) then trap
    - This is an **Illegal address exception**



# Other problems?

- Basic problem is that processes must be allocated to contiguous blocks of physical memory
  - Hard to figure out how to size these blocks given that processes are not all the same

- Now what?



- *Paging!*



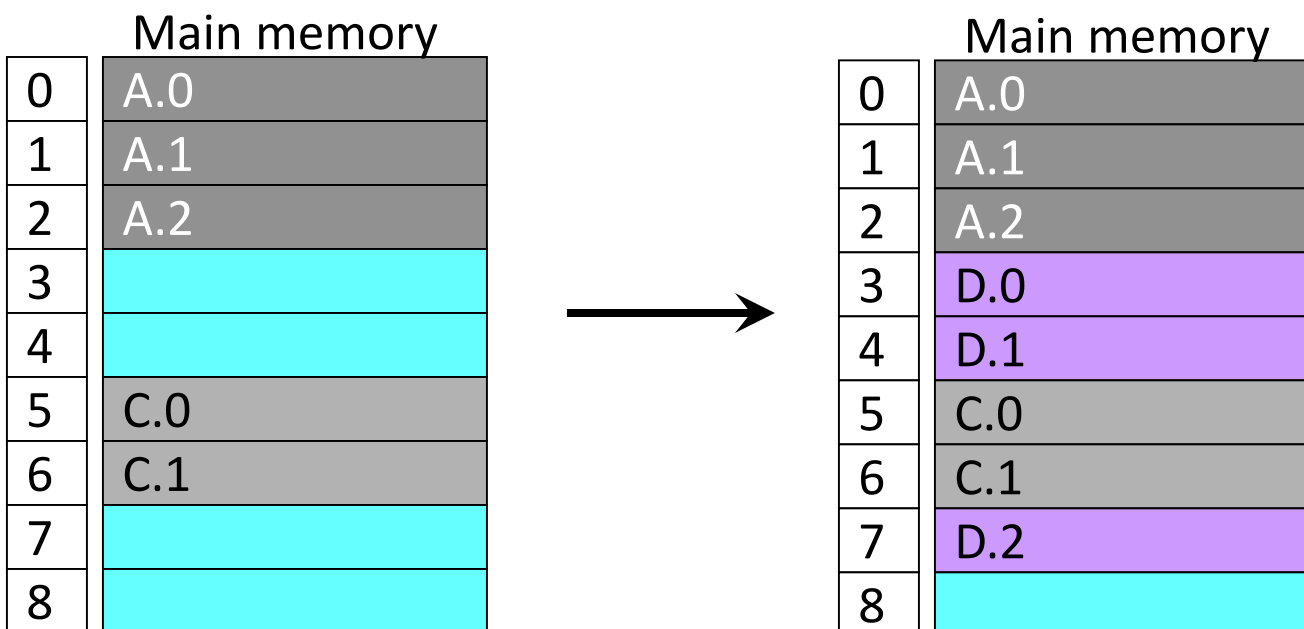
# Paging

- Logically partition physical memory into equal, fixed-size chunks ✓
  - These are called *page frames* or simply *frames*
- Divide processes' memory into chunks of the same size ✓
  - These are called *virtual pages* or just *pages*
- Any page can be assigned to any free page frame ✓
  - External fragmentation is eliminated
  - Internal fragmentation is at most a part of one page per process
- Possible page frame sizes are restricted to powers of 2 to simplify address translation ✓



# Example of Paging

Suppose a new process, D, arrives needing 3 frames of memory



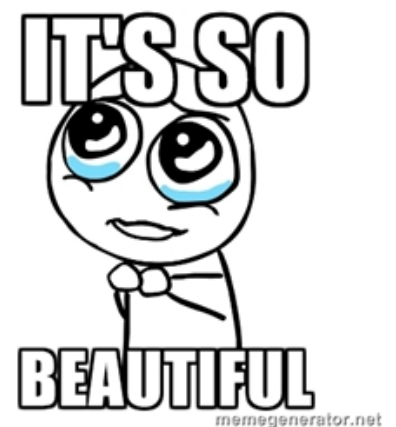
- We can fit Process D into memory, even though we don't have 3 contiguous frames available!





# Paging simplifies things

- Good virtualization of memory
  - No more base and bound registers
  - Just need to translate virtual pages to physical pages
- All we need to know is where does the page table start: PTBR (Page Table Base Register)





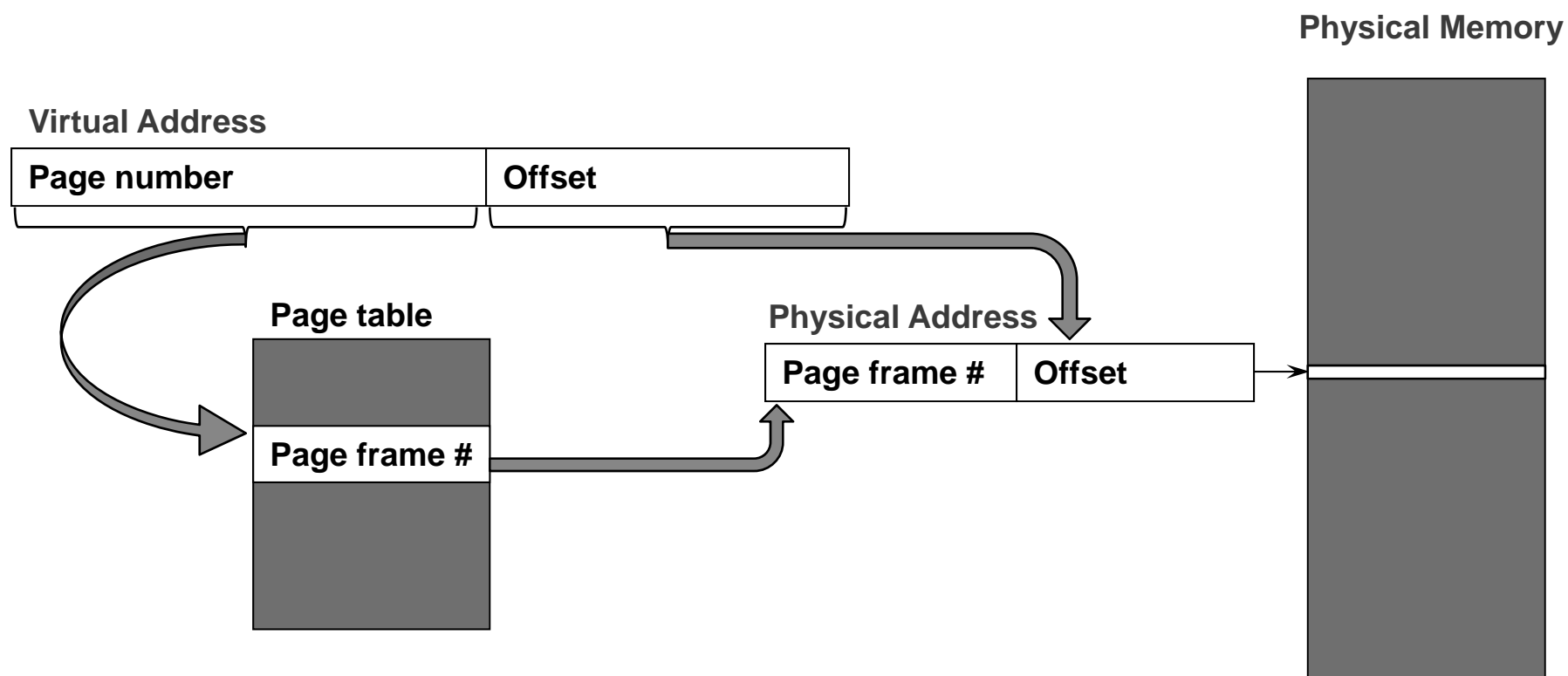
# How do we support paging?



- Recall that OS maintains a *page table* for each process:
  - Software data structure, stored in OS memory
  - Page table - records which physical frame holds each page
  - Virtual addresses now interpreted as *page number + page offset*
    - *page number = vaddr / page\_size*
    - *page offset = vaddr % page\_size*
    - Simple to calculate if page size is power-of-2
  - On each memory reference, processor MMU translates page# to frame# and adds offset to generate a physical address
  - Hence why a hardware “page table base register” (PTBR) to quickly locate the page table for the running process
    - Loaded when process runs, just like base/limit registers



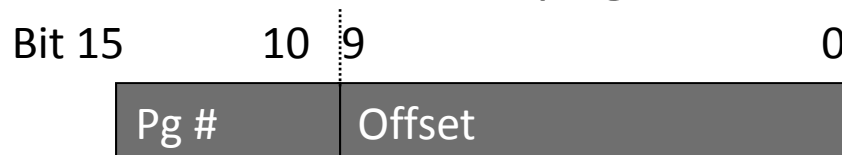
# Paged Address Translation





# Example Address Translation

- Suppose addresses are 16 bits, pages are 1024 bytes



- Least significant 10 bits of address provide offset within a page ( $2^{10} = 1024$ )
- Most significant 6 bits provide page number
- Maximum number of pages =  $2^6 = 64$

- To translate virtual address: 0x0DDE

0000 1101 1101 1110  
0    D    D    E

- Extract page number (high-order 6 bits)
  - $\rightarrow \text{pg} = \text{vaddr} \gg 10 \quad (== \text{vaddr}/1024) == 3$
- Get frame number from page table
- Combine frame number with page offset
  - $\text{offset} = \text{vaddr} \& 0x3FF \quad (== \text{vaddr} \% 1024) == 478$
  - $\text{paddr} = (\text{frame} \ll 10) \mid \text{offset}$ 
    - Equivalent to  $\text{paddr} = \text{frame} * 1024 + \text{offset}$

Logical address

Page#	Offset
3	478

Page Table

Page#	Frame#
0	11
1	16
2	7
3	28

28	478
Frame#	Offset

Physical address





- 
- The diagram illustrates the MMU address translation process. On the left, a yellow box represents the CPU/MMU. The CPU sends a virtual address  $0x7468$  to the MMU. The MMU contains a table with columns for VPN (Virtual Page Number) and OFF (Offset). The VPN is  $0x7$  and the OFF is  $0x468$ . The MMU also contains a PTBR (Page Table Base Register) which points to the base address of the page table in Main Memory. The binary representation of the virtual address  $0x7468$  is shown as  $0111\ 0100\ 0110\ 1000$ . The first four bits,  $0111$ , represent the VPN. The remaining bits,  $0100\ 0110\ 1000$ , represent the OFF. The MMU uses the VPN to find the corresponding page table entry in Main Memory. The page table entry for VPN  $0x7$  is shown as a red box containing the value  $42$ . This value represents the physical address of the page. The MMU then combines this physical address with the OFF to find the final byte in Main Memory. The final byte is shown as a yellow box in Main Memory, labeled with the offset  $468$ . The entire process is connected to a blue bar at the bottom labeled "Memory Bus".



# Details of calculation

- Program generates virtual address 0x7468
  - CPU and MMU see binary 0111 0100 0110 1000
  - Virtual page is **0x7**, offset is **0x468** (**0111 0100 0110 1000**)
- Page table entry 0x7 contains 0x42
  - Page frame number is 0x42
  - Virtual page 0x7 is stored in physical frame 0x42
- **Physical address** =  $0x42 \ll 12 + 0x468 = 0x42468$
- MMU hardware generates address of page table entry, does lookup without OS
- OS has to load PTBR for new process on context switch
  - Remember that Page Tables are per process!



# The Page Table

- Simplest version
  - A linear array of *page table entries*, 1 entry per virtual page
  - Stored in OS memory, attached to process structure
  - Virtual page number (VPN) is array index
  - Allocate enough physical memory (ppages) for entire page table

```
struct addrspace {  
    paddr_t pgtbl;  
    ...  
}
```

nentries calc: max vaddr is all 1's (i.e.  $\sim 0$ ). To get number of pages, divide by page size ( $\gg 12$ ). But only half of address space is user and mapped by page table so divide by 2 ( $\gg 13$ ).

```
struct addrspace *as_create(void) {  
    struct addrspace *as =  
        kmalloc(sizeof(struct addrspace));  
    int nentries = (~0 >> 13) + 1 ;  
    int npages = DIVROUNDUP(nentries*  
        sizeof(pte_t), PAGE_SIZE);  
    as->pgtbl = getppages(npages);  
    ...  
}
```



# Page Table Entries

1	1	1	3	26
M	R	V	Prot	Page Frame Number

- Page table entries (PTEs) control mapping
  - **Modify bit** (M) says whether or not page has been written
    - Set when a write to a page occurs
  - **Reference bit** (R) says whether page has been accessed
    - Set when a read or write to the page occurs
  - **Valid bit** (V) says whether PTE can be used
    - Checked on each use of virtual address
  - **Protection bits** specify what operations are allowed on page
    - Read/write/execute
  - **Page frame number** (PFN) determines physical page
  - Not all bits are provided by all architectures



# MIPS R2000 Page Table Entry

20	1	1	1	1	8
Page Frame Number	N	D	V	G	unused

- N == not cached
- D == dirty (meaning “writable”, not set by hardware)
- V == valid
- G == global (can be used by all processes)
- Maximum  $2^{20}$  physical pages, each 4 kB → maximum 4GB of physical RAM



# Where are page tables stored?

- Too big too fit into MMU => store in memory
- More specifically, in protected memory
- Recall that a user process cannot access its own page table!



# Paging Limitations - Space

- Memory required for page table can be large (space overhead)
  - Need one PTE per page
  - 32 bit virtual address space w/ 4K pages
    - =>  $2^{20}$  PTEs
  - 4 bytes/PTE => 4MB/page table
  - 100 processes => 400MB just for page tables! OUCH!
    - It gets worse: modern processors have 64-bit address spaces => 16 petabytes per page table!
- Solutions?





# Solution: Bigger Pages

- Use 16K pages instead of 4K pages
  - 32 bit virtual address space w/ 16K pages
    - =>  $2^{18}$  PTEs
  - 4 bytes/PTE => 1MB/page table
  - 100 processes => 100MB for page tables!
    - A factor of 4 less space... Yay??
- Say a process only needs 1KB of memory
  - Page size = 4K => 3KB wasted
  - Page size = 16K => 15KB wasted! OUCH!
- So, what does this mean?
  - The bigger the pages, the more severe internal fragmentation may occur
  - Ideally, we want something else ...







# Better Solutions?

- Solution 1: Hierarchical page tables
- Solution 2: Hashed page tables
- Solution 3: Inverted page tables
- To be continued ...



# Announcements

- Midterm next Wednesday during lecture timeslot
  - Make sure to attend your own section's midterm!
  - Covers up to Virtual Memory (excluding)
  - Write in pen, cannot regrade tests written in pencil!
- Location and logistics:
  - Check website!
  - Check Piazza announcements!