CSC 252: Computer Organization
Spring 2022: Lecture 20

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

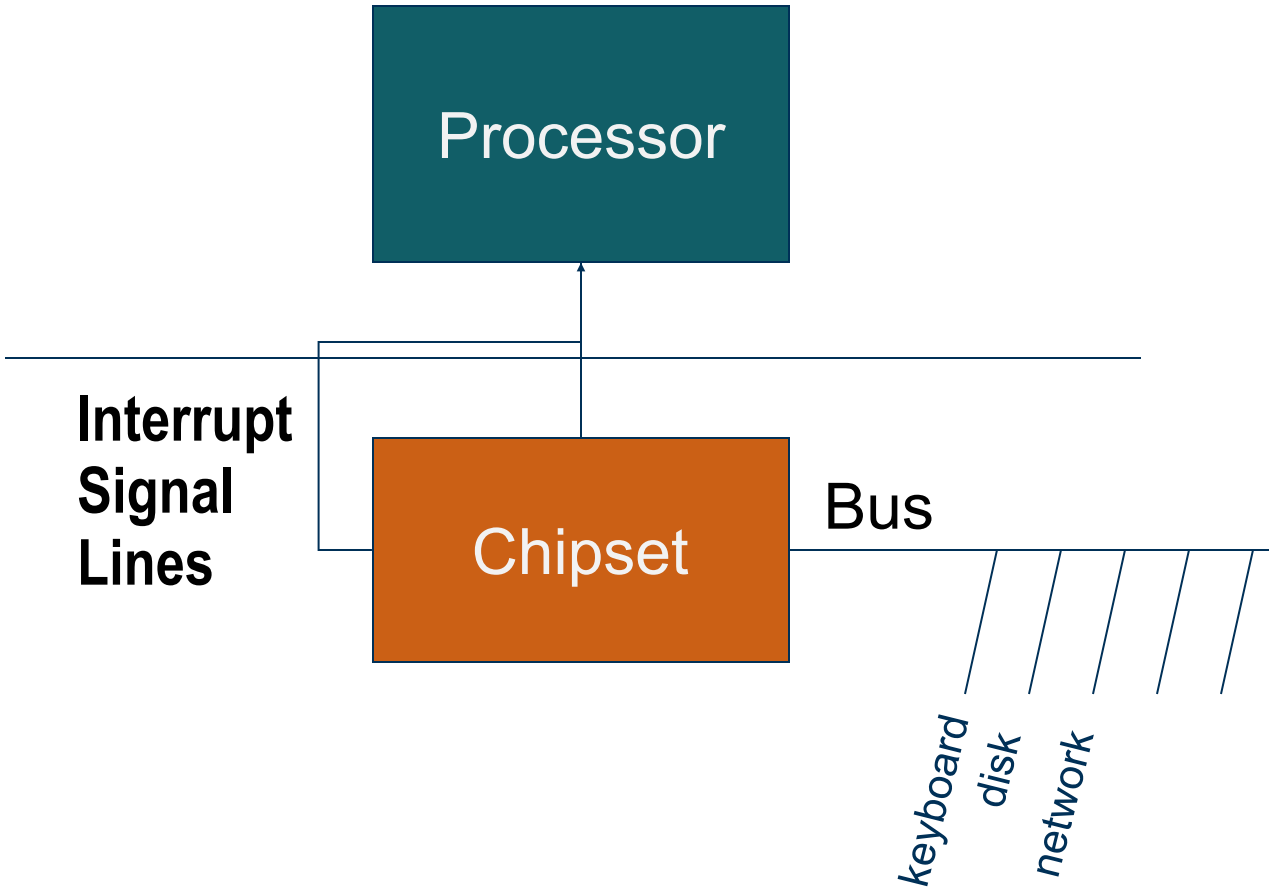
- Cache problem set: https://www.cs.rochester.edu/courses/252/spring2022/handouts.html
  - Not to be turned in. Won't be graded.
- Assignment 4 due April 8.

SUN	MON	TUE	WED	THU	FRI	SAT
27	28	29	30	31	Apr 1	2
				Today		
3	4	5	6	7	8	9
					Due	

Today

- Signals: The Way to Communicate with Processes
- Interrupts and exceptions: how signals are triggered

Interrupts in a Processor



Interrupts, a.k.a., Asynchronous Exceptions

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

Interrupts, a.k.a., Asynchronous Exceptions

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt

Interrupts, a.k.a., Asynchronous Exceptions

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
  - *Faults*

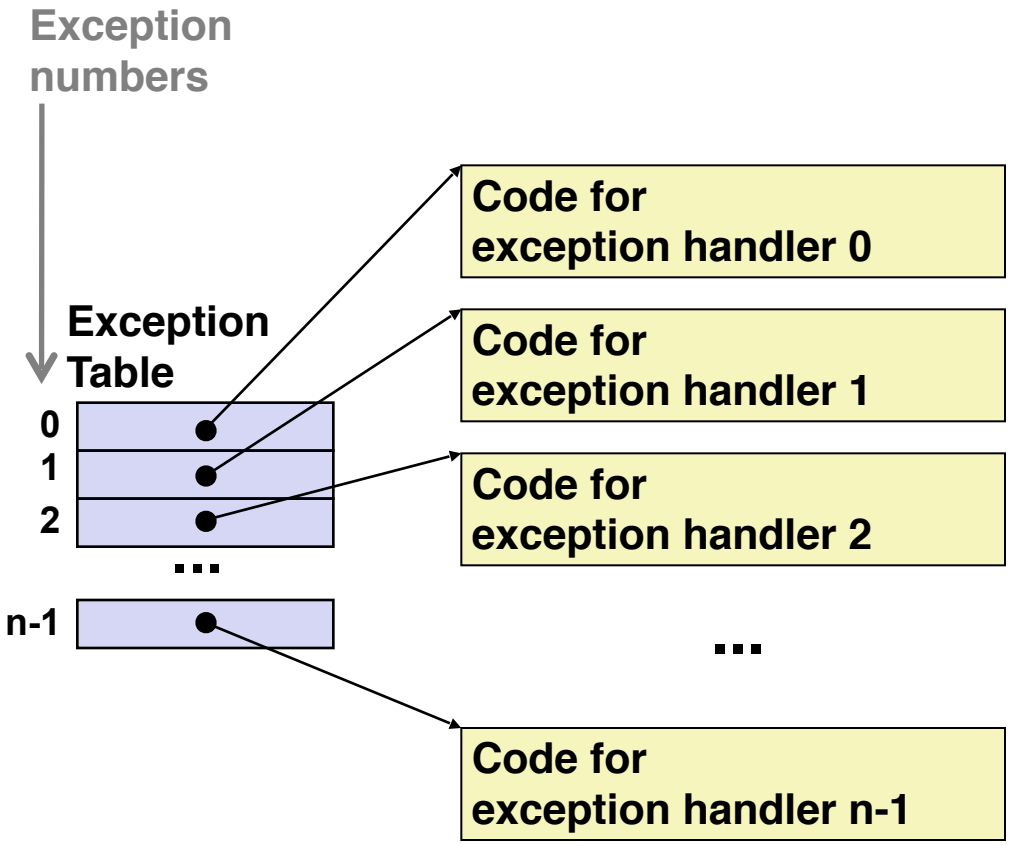
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), ***protection faults (the infamous Segmentation Fault!)*** (unrecoverable in Linux), floating point exceptions (unrecoverable in Linux)
    - These exceptions will generate signals to processes
  - *Aborts*

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), ***protection faults (the infamous Segmentation Fault!)*** (unrecoverable in Linux), floating point exceptions (unrecoverable in Linux)
    - These exceptions will generate signals to processes
  - *Aborts*
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program through a SIGABRT signal

Each Exception Has a Handler



- Each type of event has a unique exception number k
- k = index into exception table
- Exception table lives in memory. Its start address is stored in a special register
- Handler k is called each time exception k occurs

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?
  - Ctrl + C sends a keyboard interrupt to the CPU, which triggers an interrupt handler

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?
  - Ctrl + C sends a keyboard interrupt to the CPU, which triggers an interrupt handler
  - The interrupt handler, executed by the kernel, triggers certain piece of the kernel, which generates the SIGINT signal, which is then delivered to the target process

When to Execute the Handler?

- Interrupts: when convenient. Typically wait until the current instructions in the pipeline are finished
- Exceptions: typically immediately as programs can't continue without resolving the exception (e.g., page fault)
- Maskable verses Unmaskable
  - Interrupts can be individually masked (i.e., ignored by CPU)
  - Synchronous exceptions are usually unmaskable
- Some interrupts are intentionally unmaskable
  - Called non-maskable interrupts (NMI)
  - Indicating a critical error has occurred, and that the system is probably about to crash

Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction

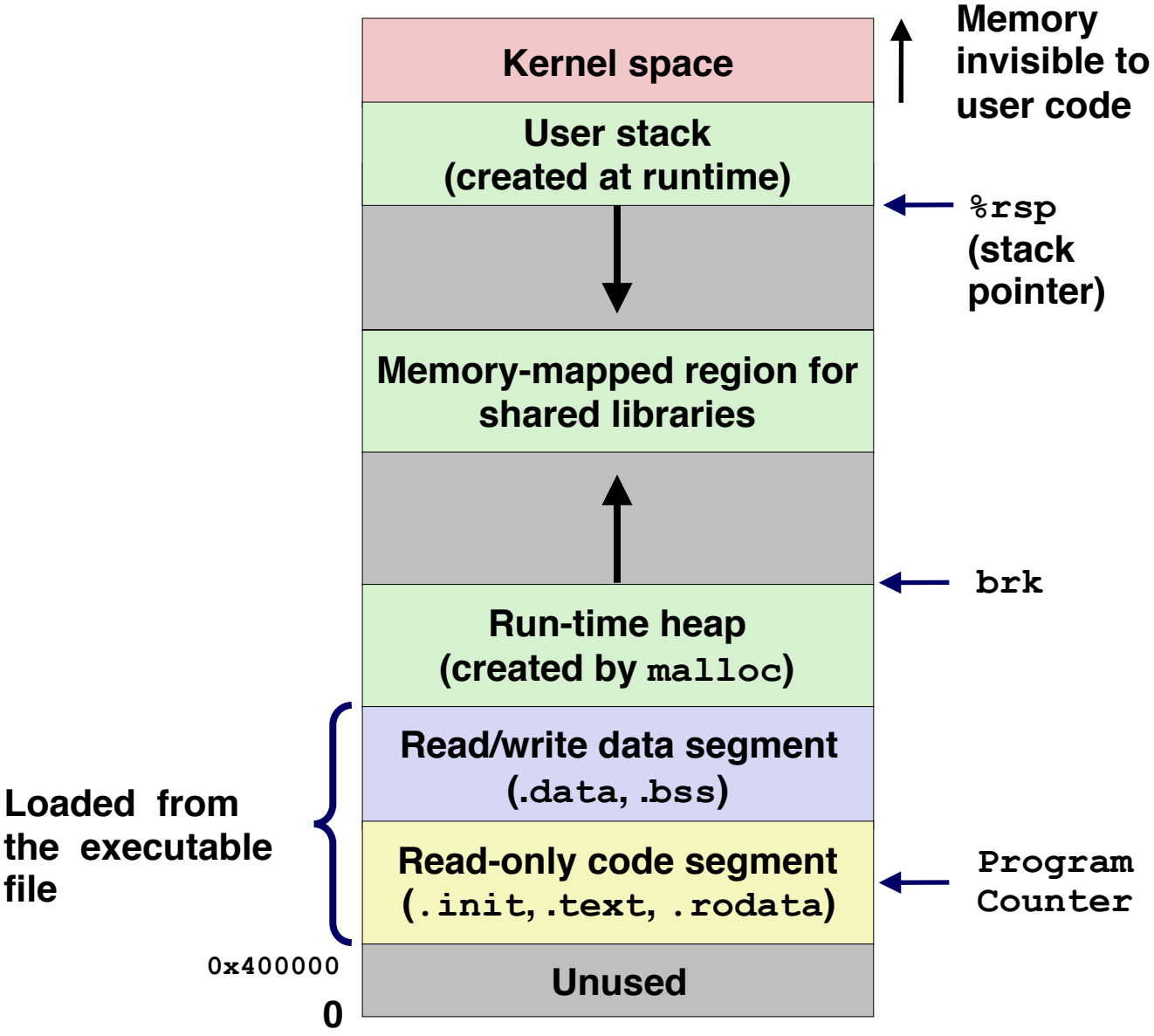
Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction
- Faults
  - Exception handler returns to the instruction that caused the exception, i.e., ***re-execute*** it!

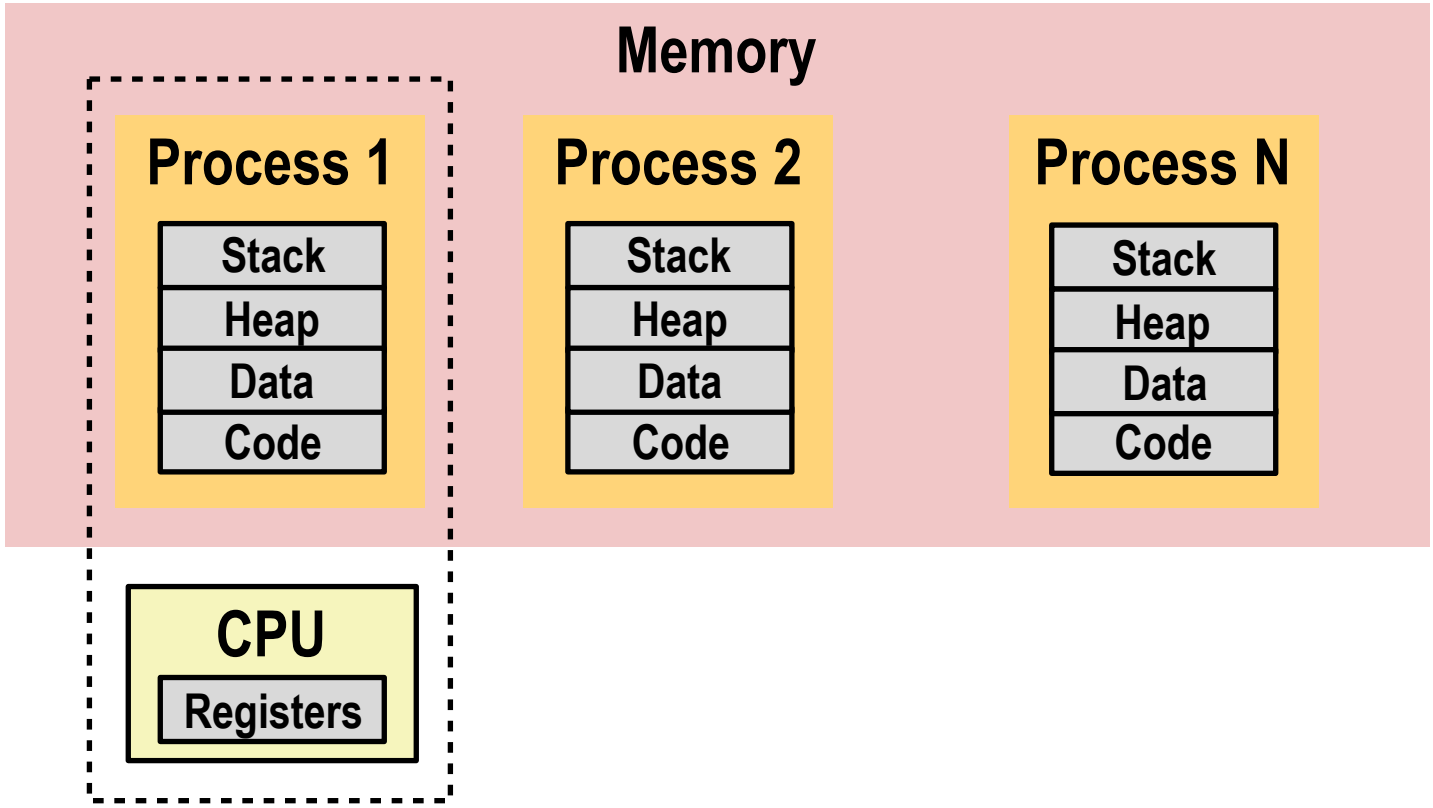
Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction
- Faults
  - Exception handler returns to the instruction that caused the exception, i.e., ***re-execute*** it!
- Aborts
  - Never returns to the program

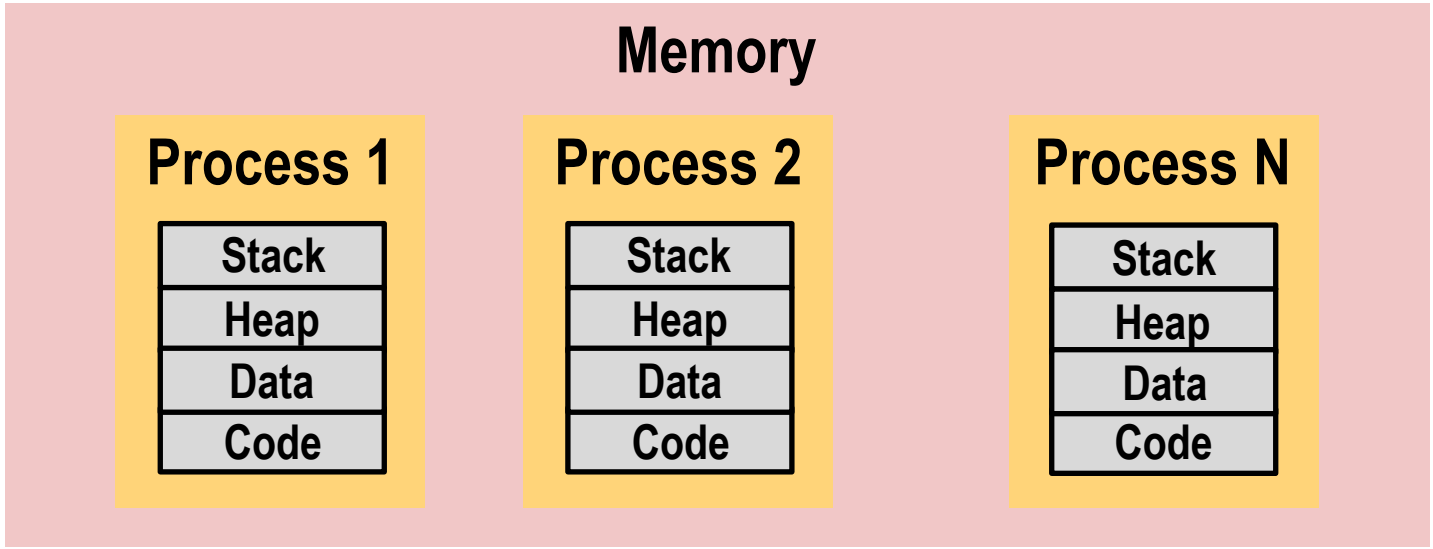
Process Address Space



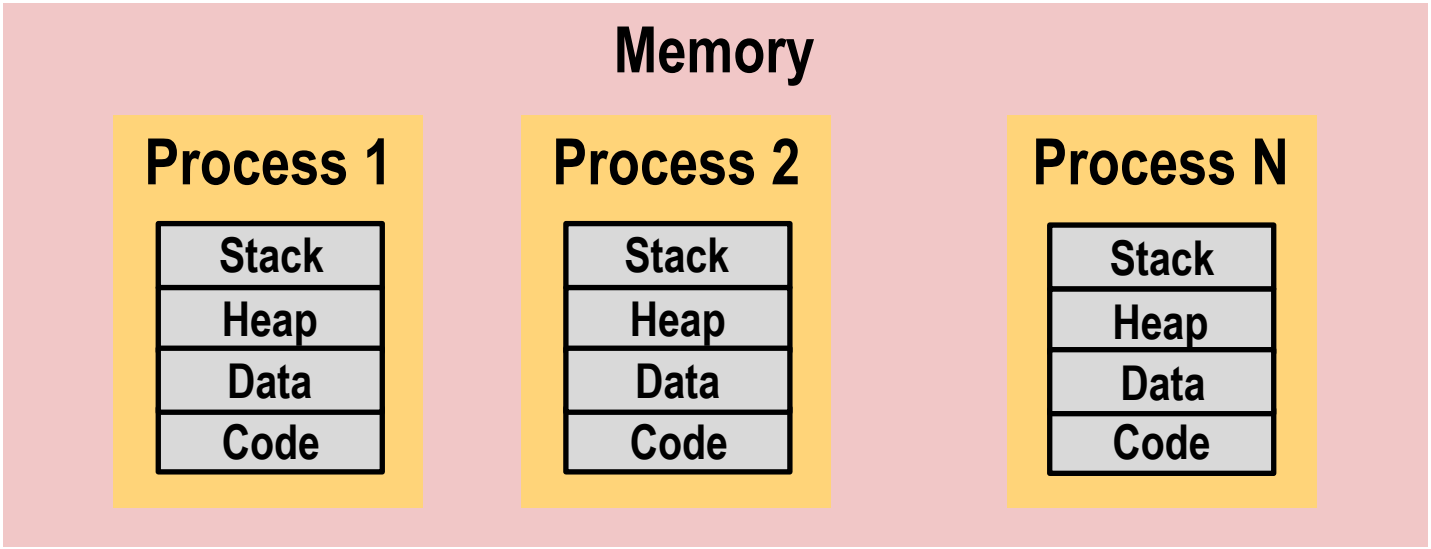
Multiprocessing Illustration



Problem 1: Space



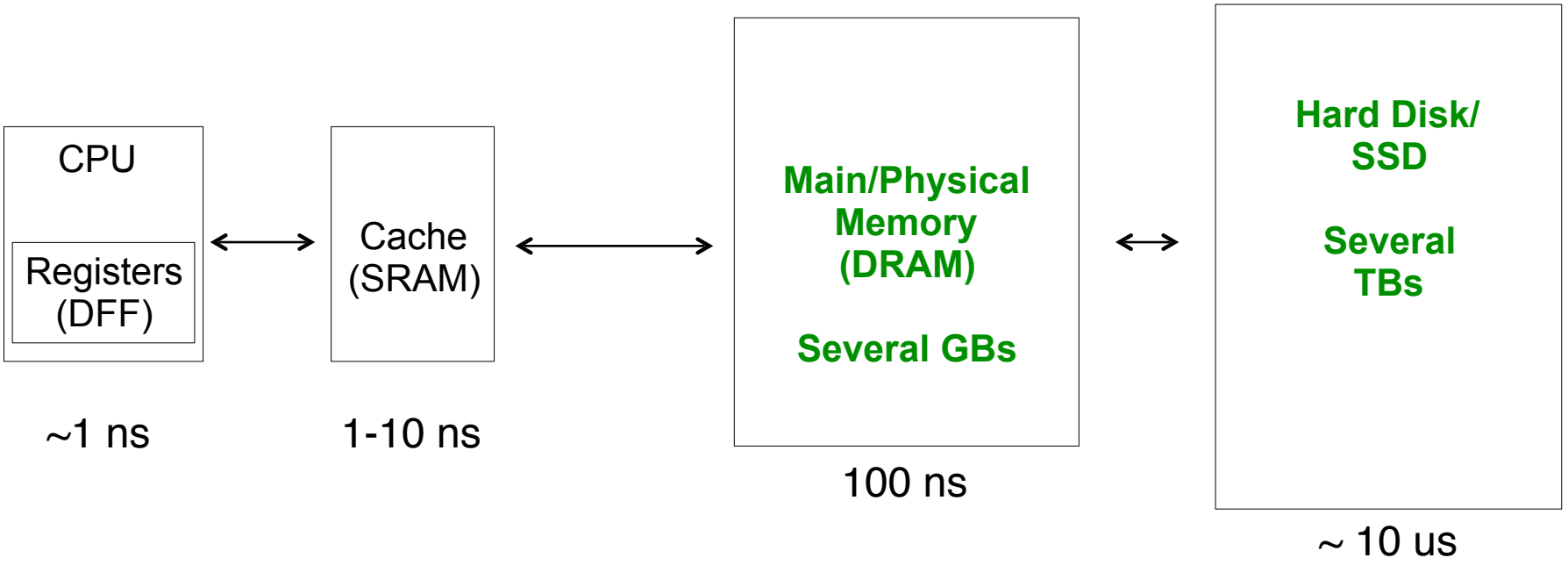
Problem 1: Space



- Space:
  - Each process's address space is huge (64-bit): can memory hold it (16GB is just 34-bit)?
  - 2^48 bytes is 256 TB
  - There are multiple processes, increasing the storage requirement further

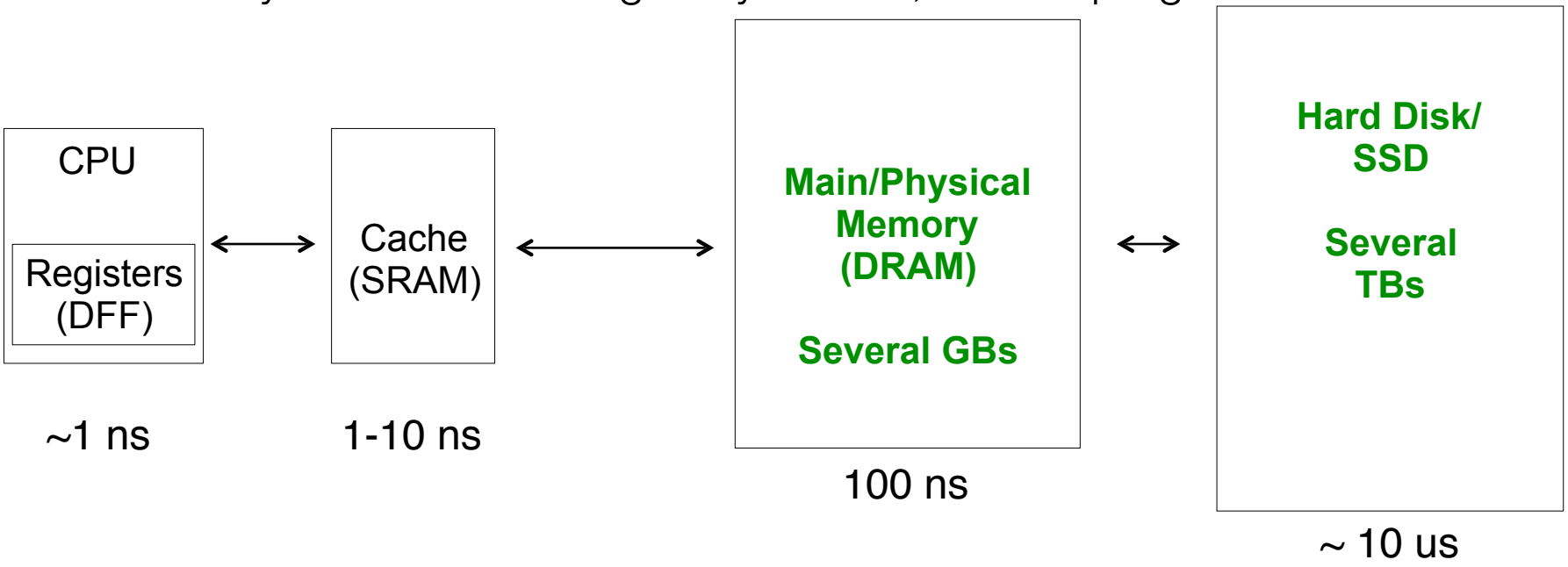
Recall: Memory Hierarchy

- Solution: store all the data in disk (several TBs typically), and use memory only for most recently used data
  - Of course if a process uses all its address space that won't be enough, but usually a process won't use all 64 bits. So it's OK.



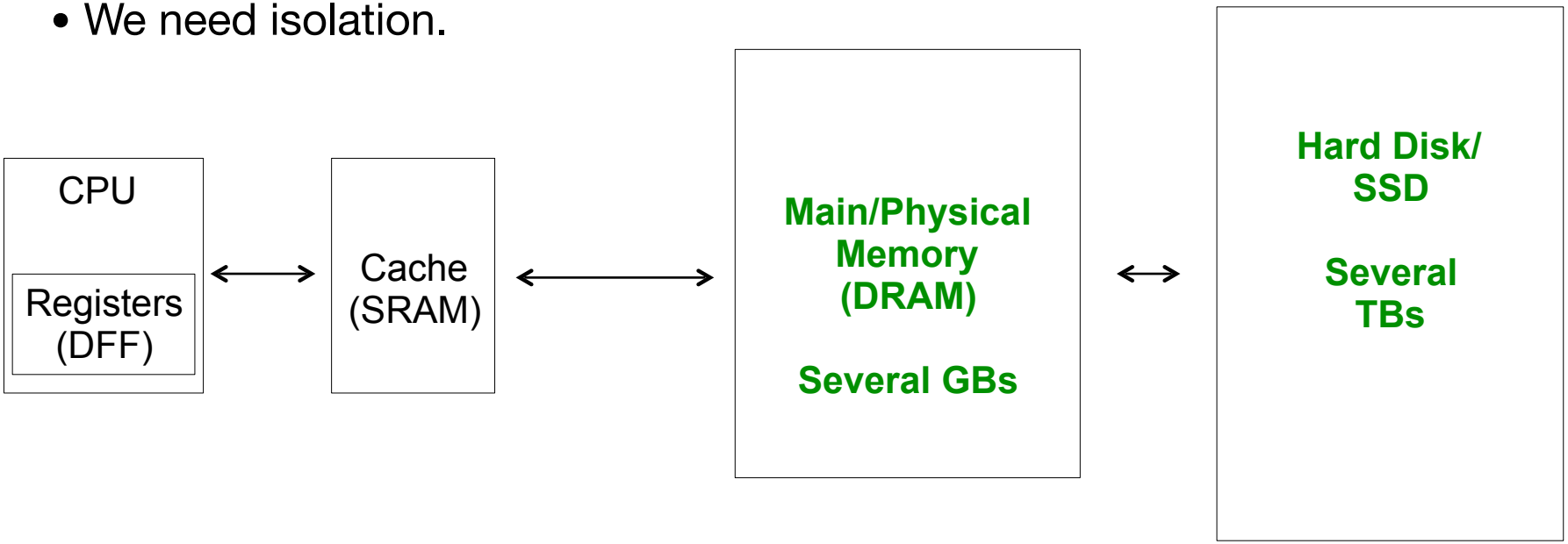
Recall: Memory Hierarchy

- Solution: store all the data in disk (several TBs typically), and use memory only for most recently used data
  - Of course if a process uses all its address space that won't be enough, but usually a process won't use all 64 bits. So it's OK.
- Challange: who is moving data back and forth between the DRAM/main memory/physical memory and the disk?
  - Ideally should be managed by the OS, not the programmer.



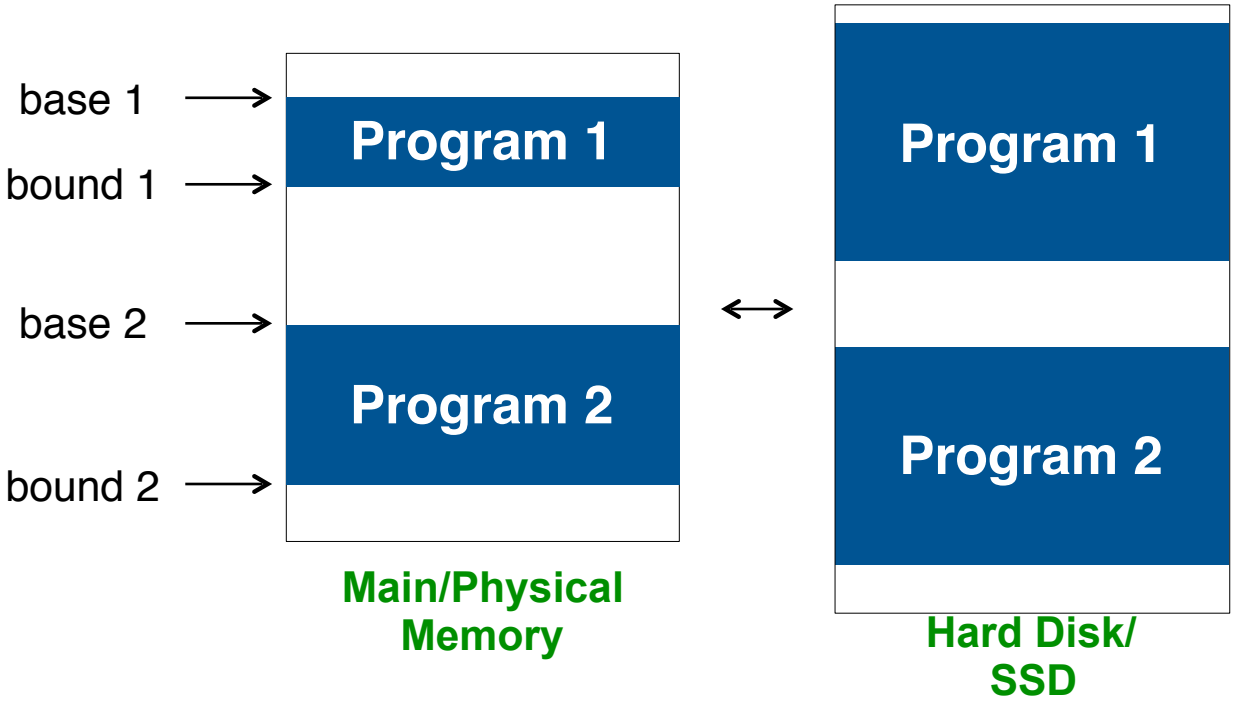
Problem 2: Security

- Different programs/processes will share the same physical memory
  - Or even different uses. A CSUG machine is accessed by all students, but there is one single physical memory!
- What if a malicious program steals/modifies data from your program?
  - If the malicious program get the address of the memory that stores your password, should it be able to access it? If not, how to prevent it?
- We need isolation.



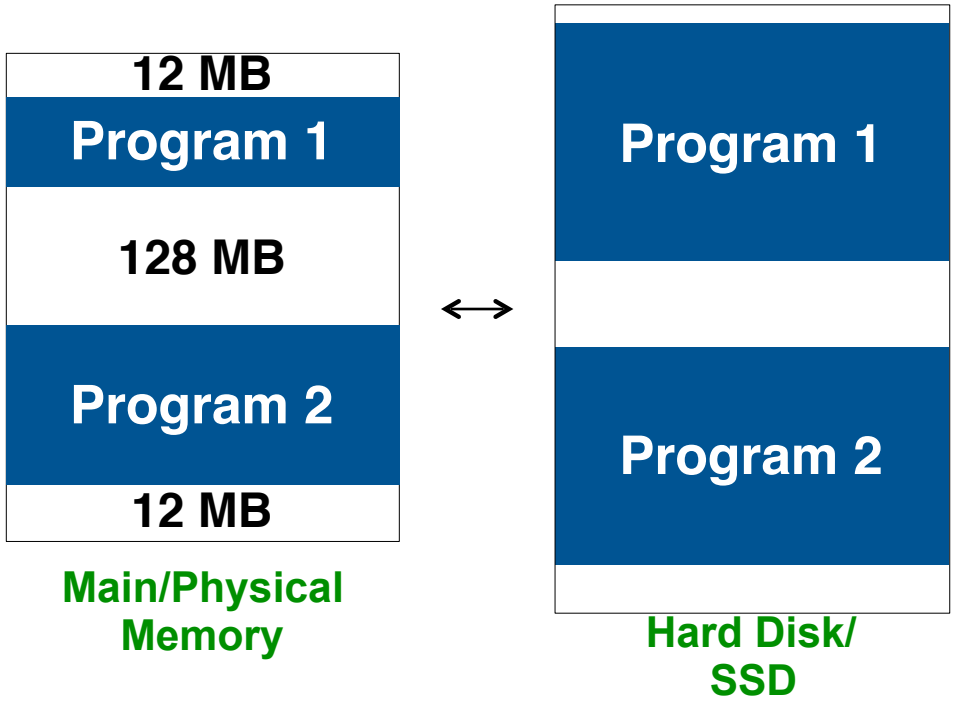
One Way to Isolate: Segments

- Different processes will have exclusive access to just one part of the physical memory.
- This is called ***Segments***.
  - Need a base register and a bound register for each process. Not widely used today. x86 still supports it (backward compatibility!)
  - Fast but inflexible. Makes benign sharing hard.



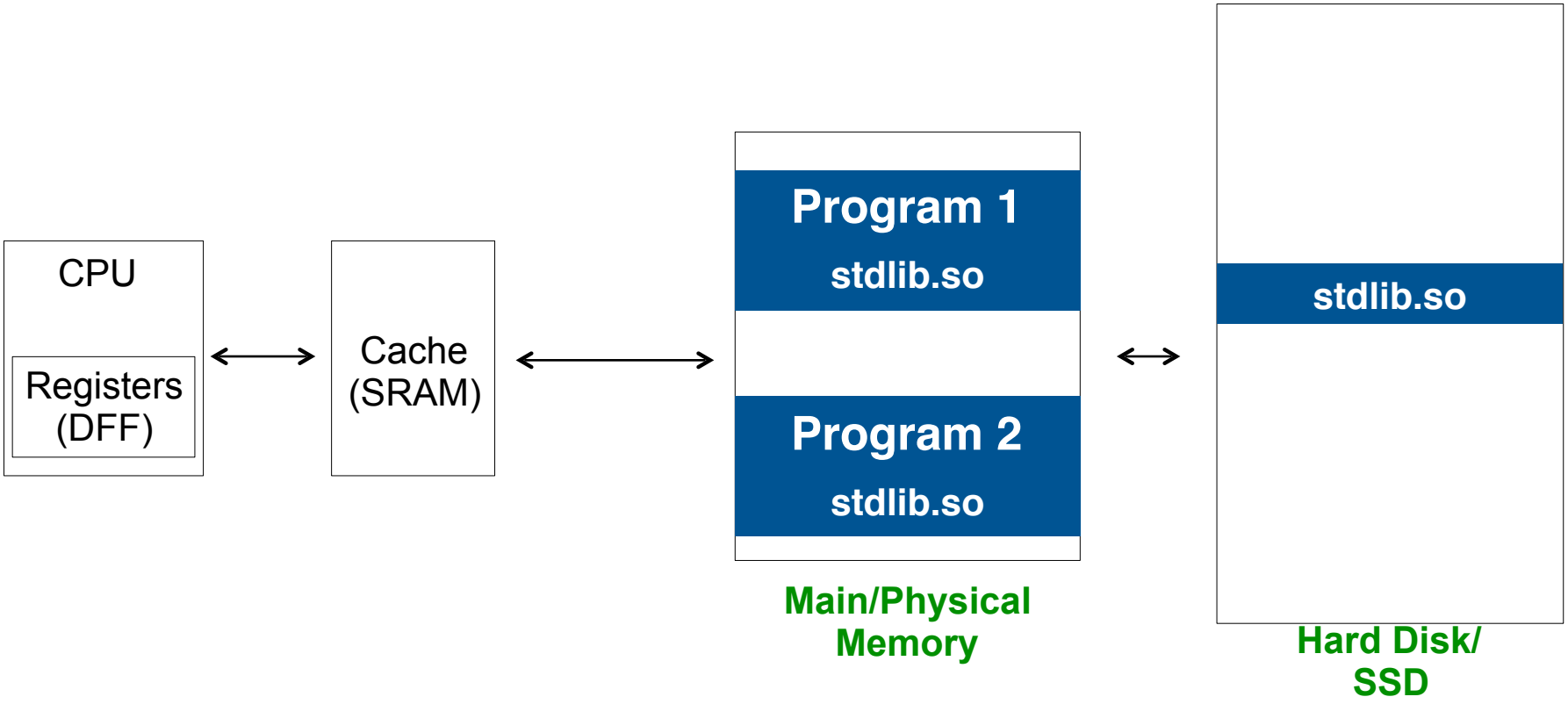
Problem 3: Fragmentation (with Segments)

- Each process gets a continuous chunk of memory. Inflexible.
- What if a process requests more space than any continuous chunk in memory but smaller than the total free memory?
  - This is called “fragmentation”; will talk about this more later.
- Need to allow assigning discontinuous chunks of memory to processes.

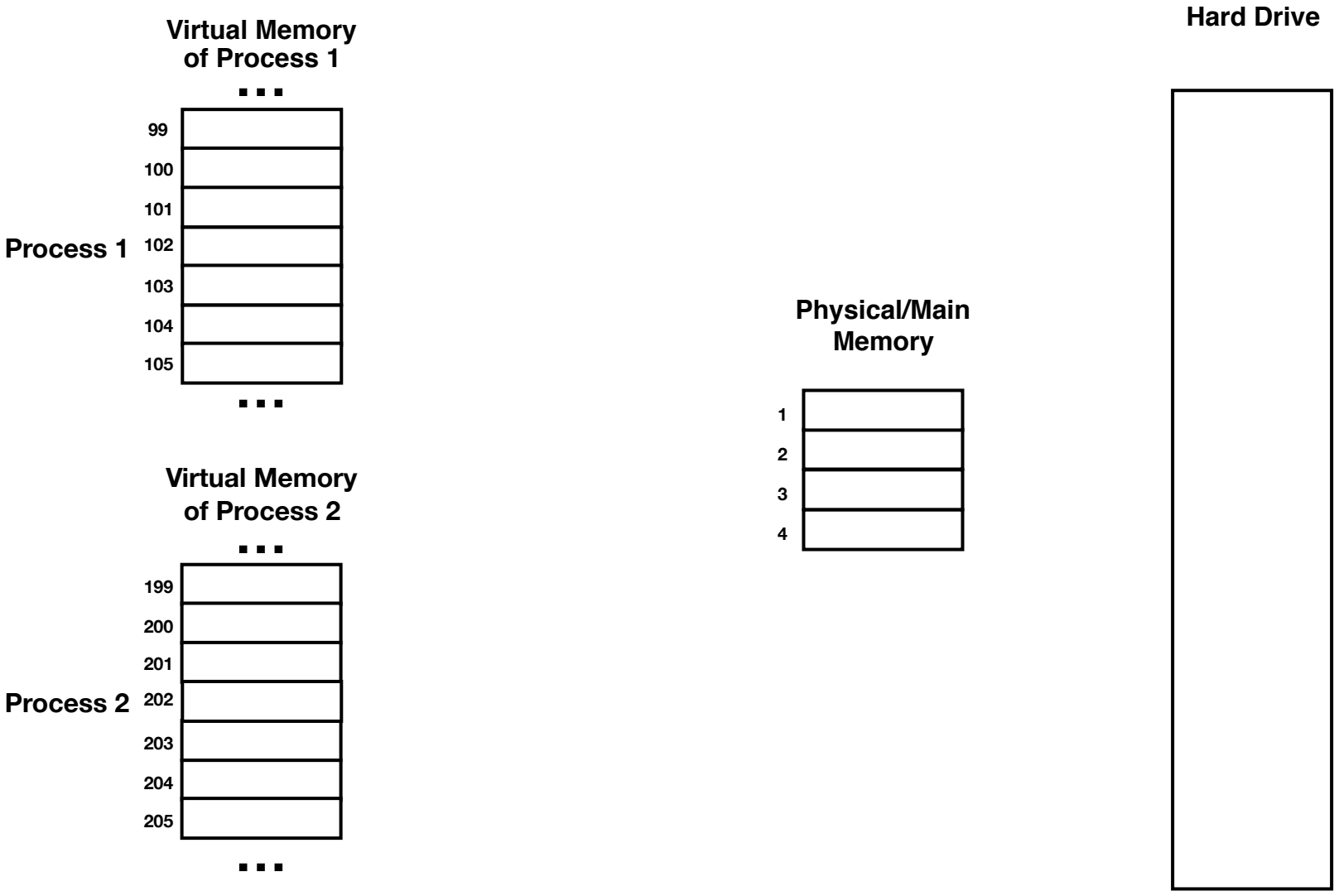


Problem 4: Benign Sharing (with Segments)

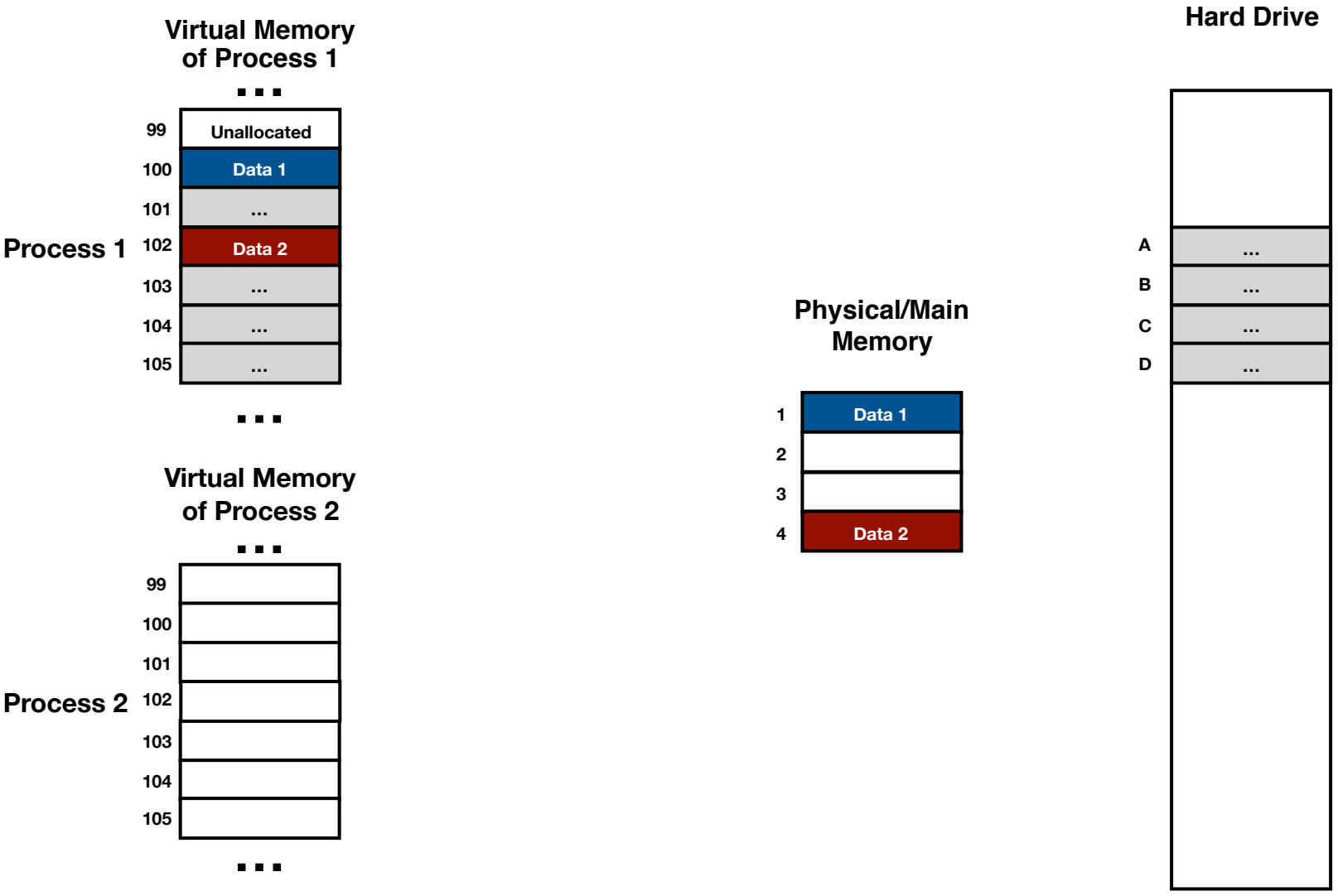
- Different programs/processes will share same data: files, libraries, etc.
- No need to have separate copies in the physical memory.
- Would be good to let other processes access part of the current's process' memory based on the "permission".



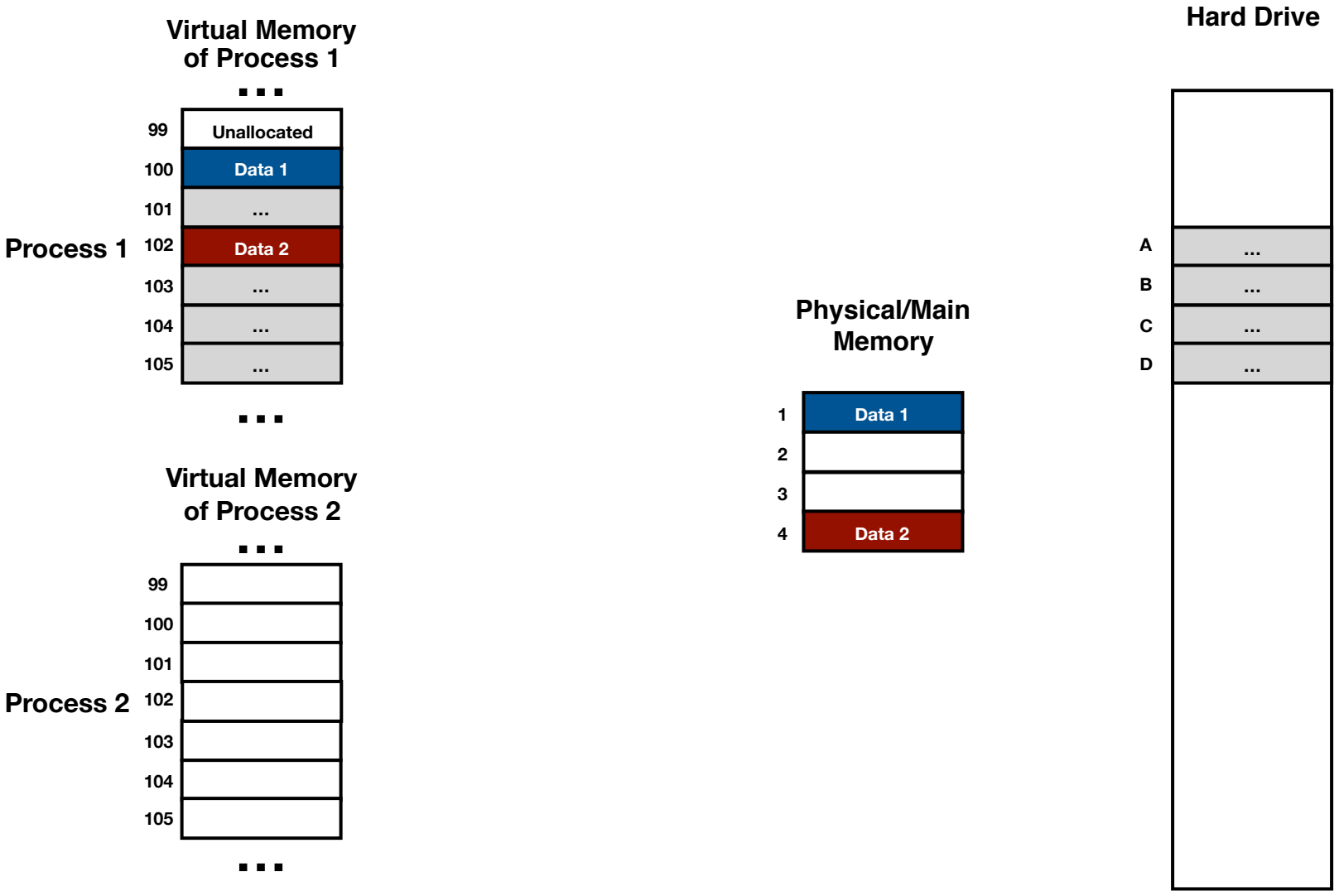
The Big Idea: Virtual Memory



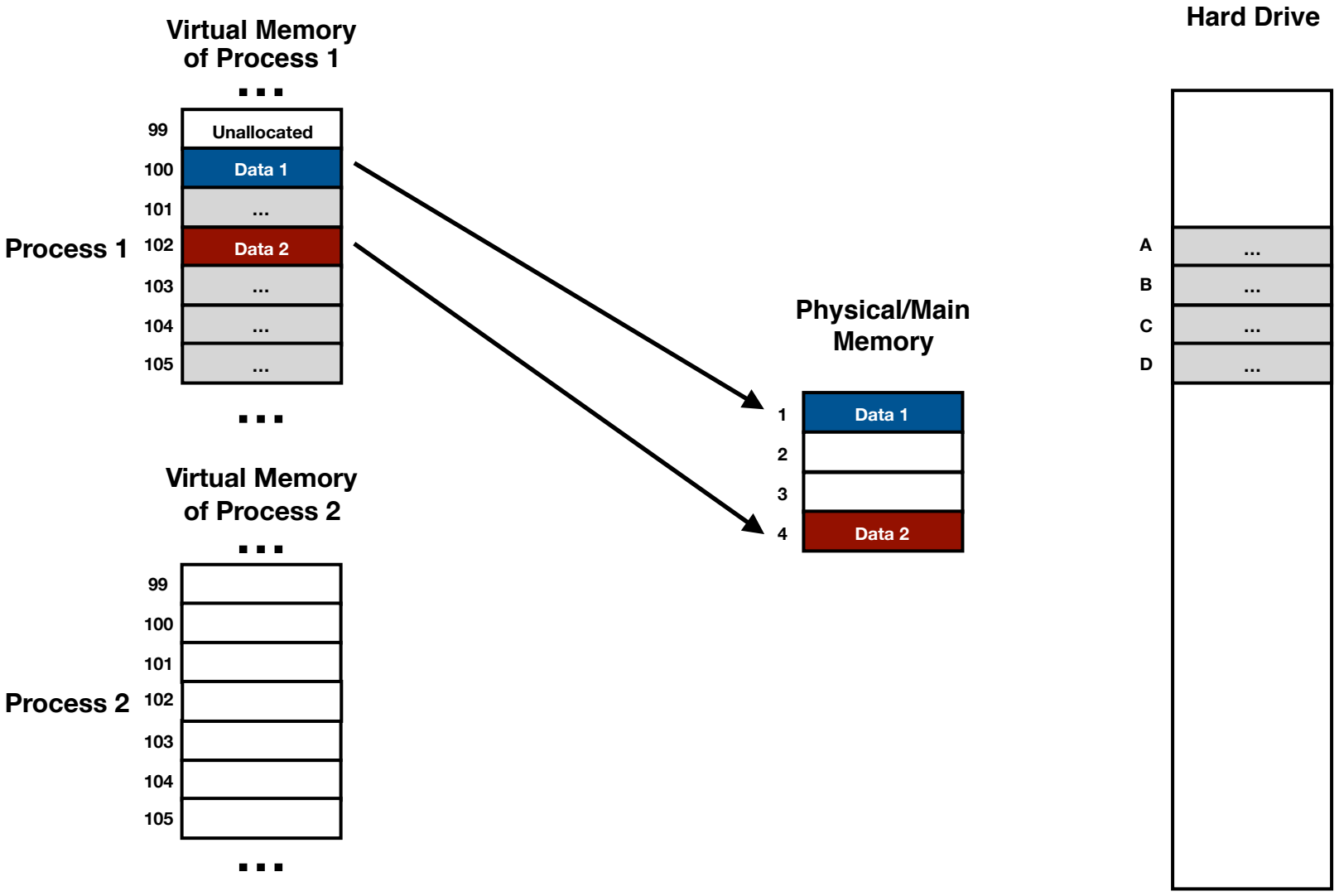
“Cache” Data in Physical Memory



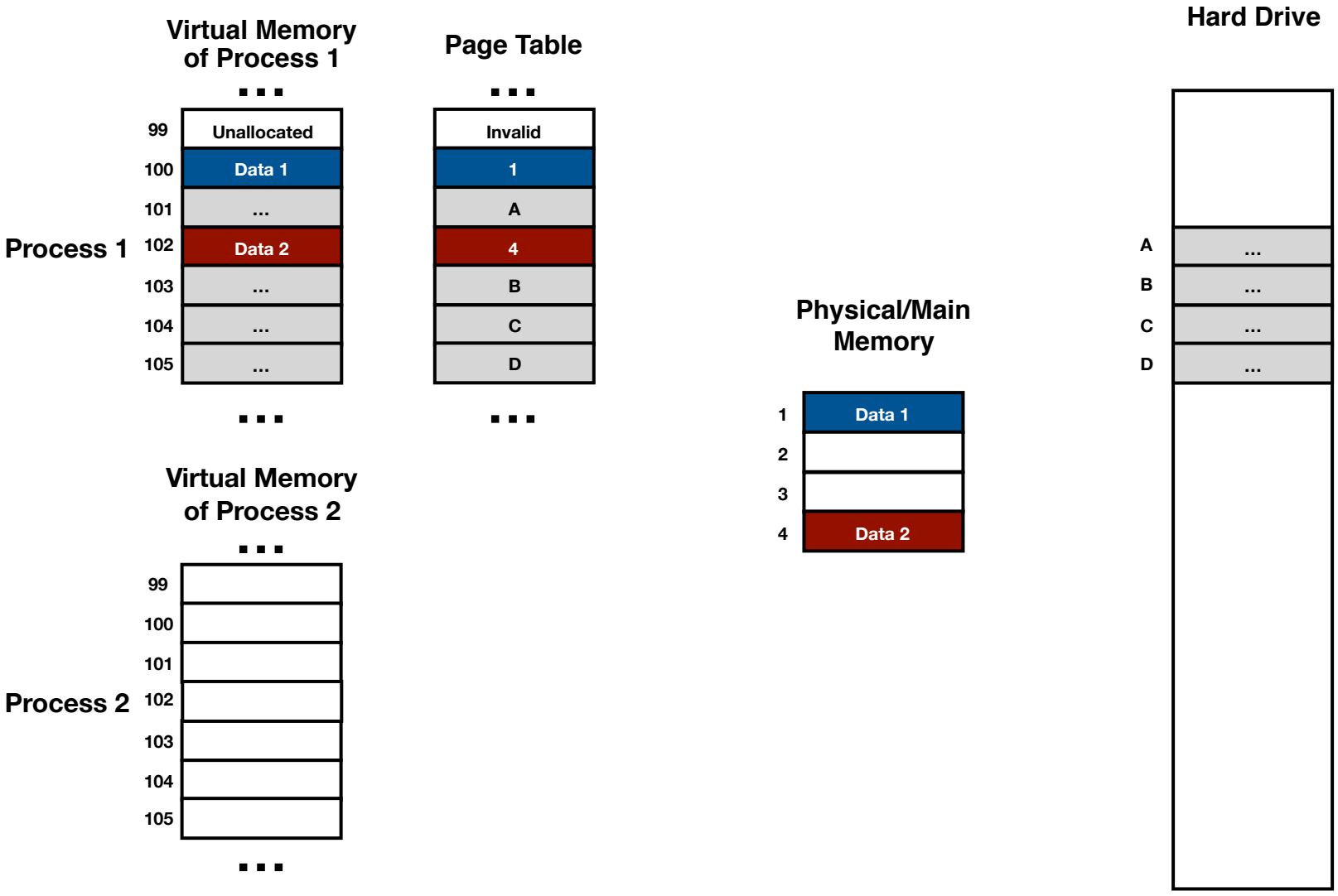
Allow Using Discontinuous Allocation



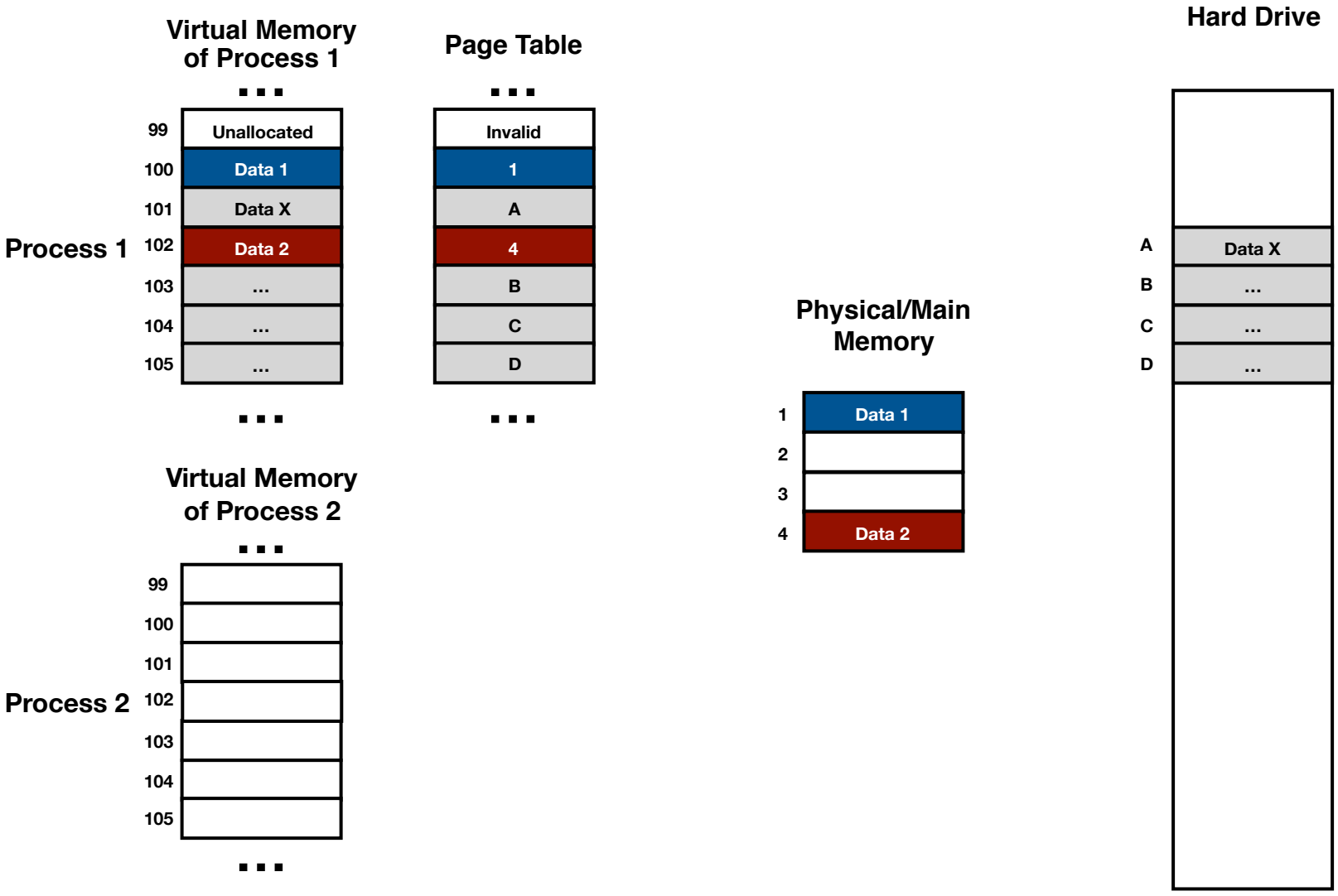
Allow Using Discontinuous Allocation



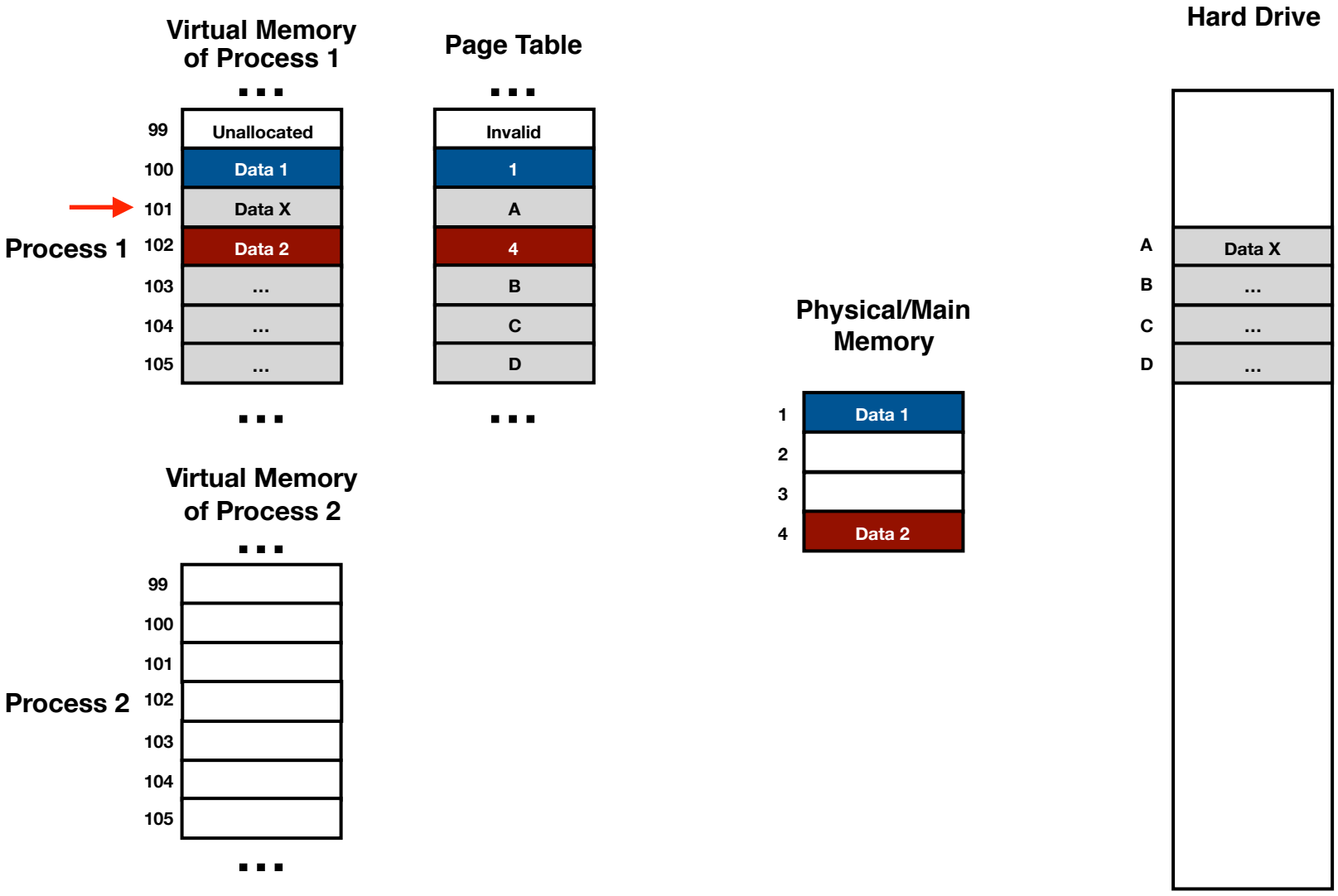
Page Table


Virtual Memory of Process 1
Page Table
Process 1
99
100
101
102
103
104
105
Unallocated
Data 1
...
Data 2
...
...
...
Invalid
1
A
4
B
C
D
Physical/Main Memory
1
2
3
4
Data 1
Data 2
Hard Drive
A
B
C
D
...
...
...
...
Virtual Memory of Process 2
Process 2
99
100
101
102
103
104
105

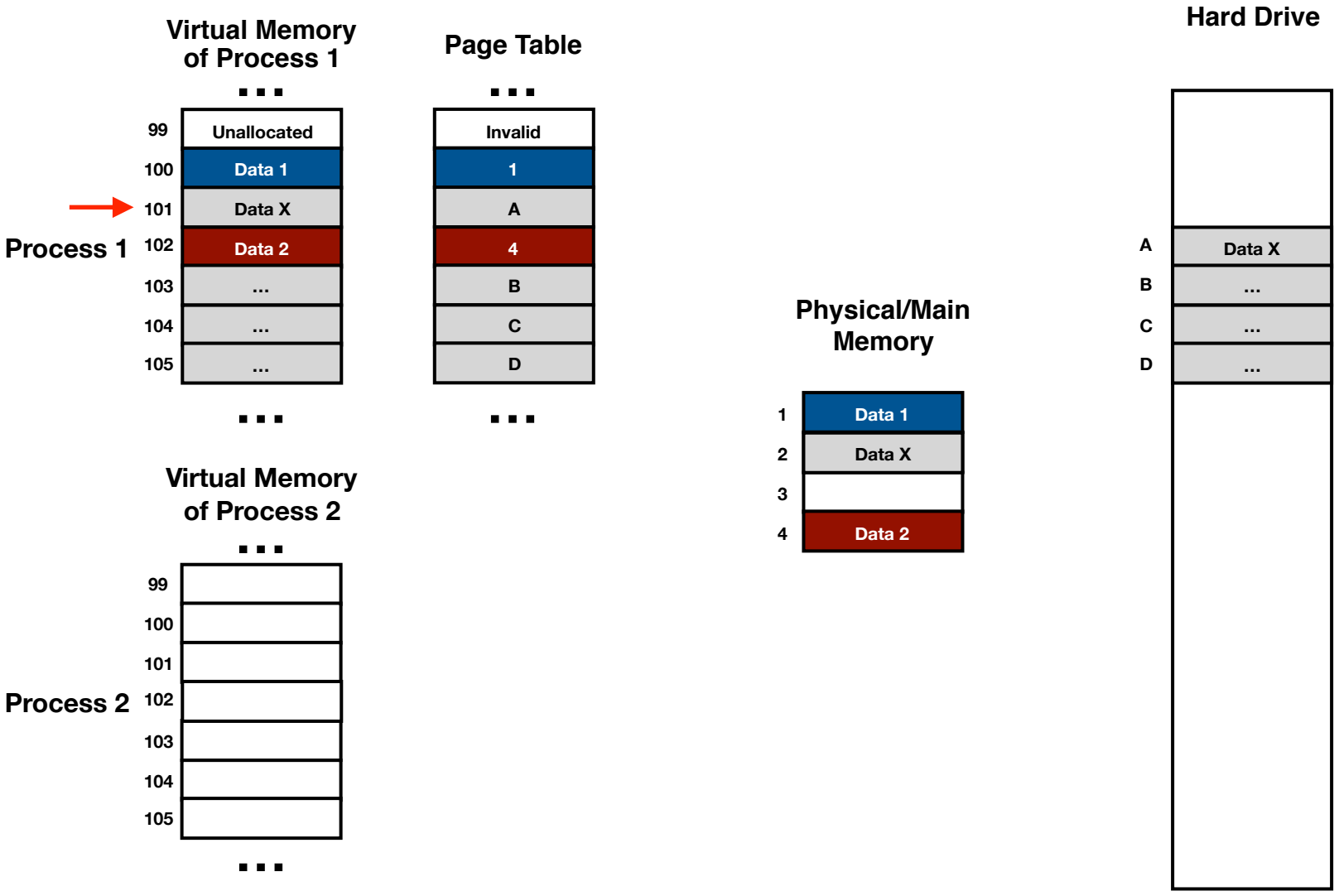
Demand Paging



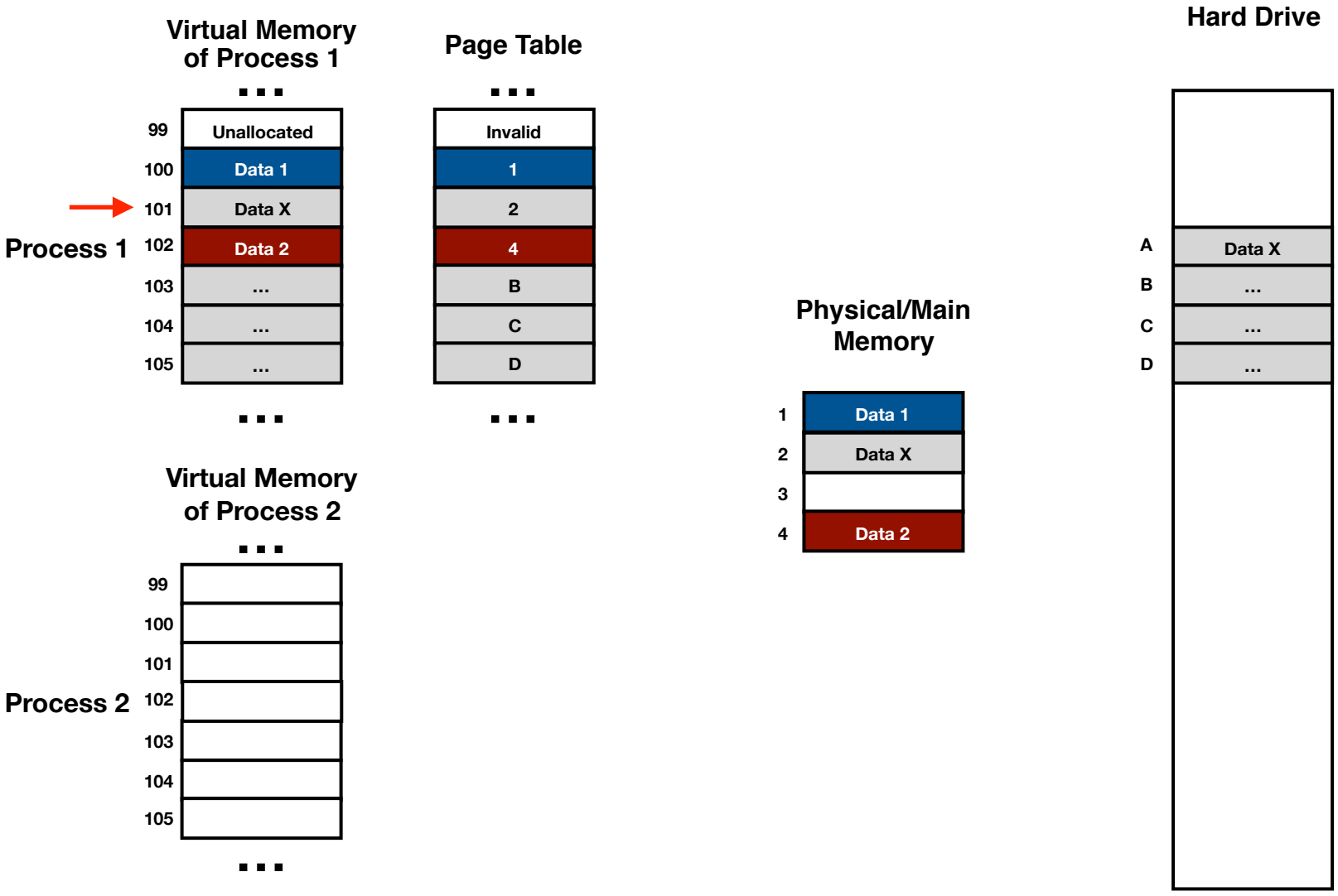
Demand Paging



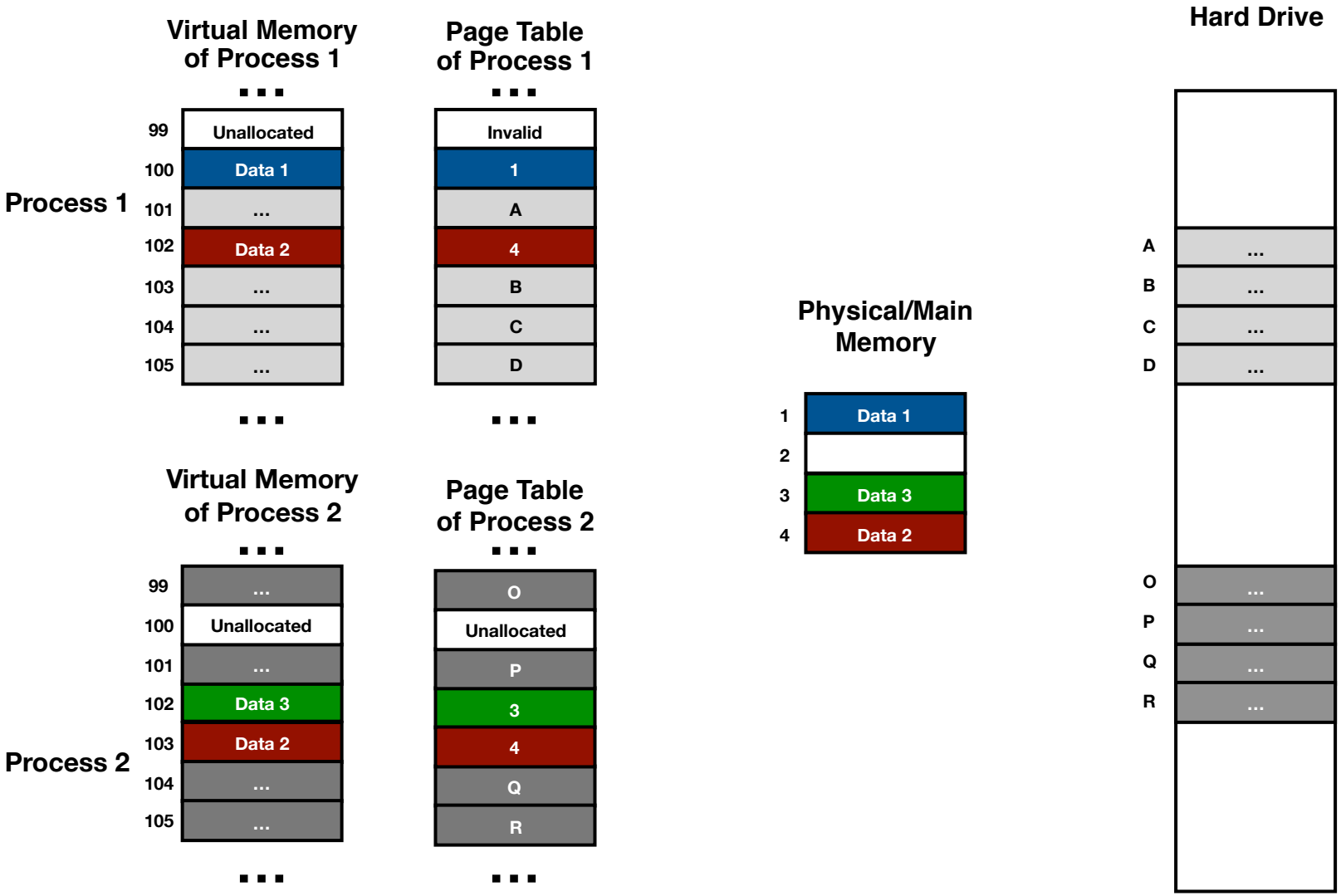
Demand Paging



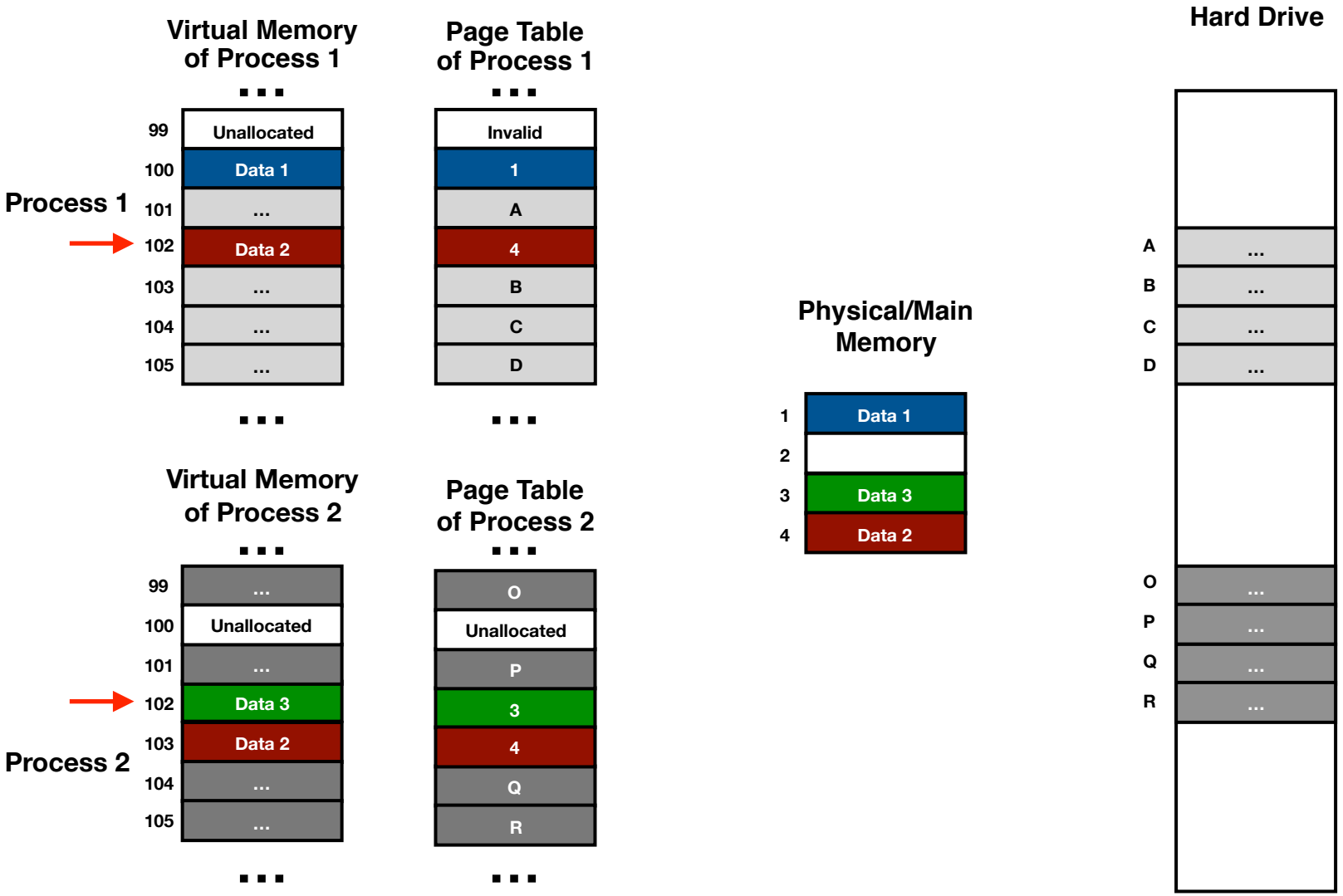
Demand Paging



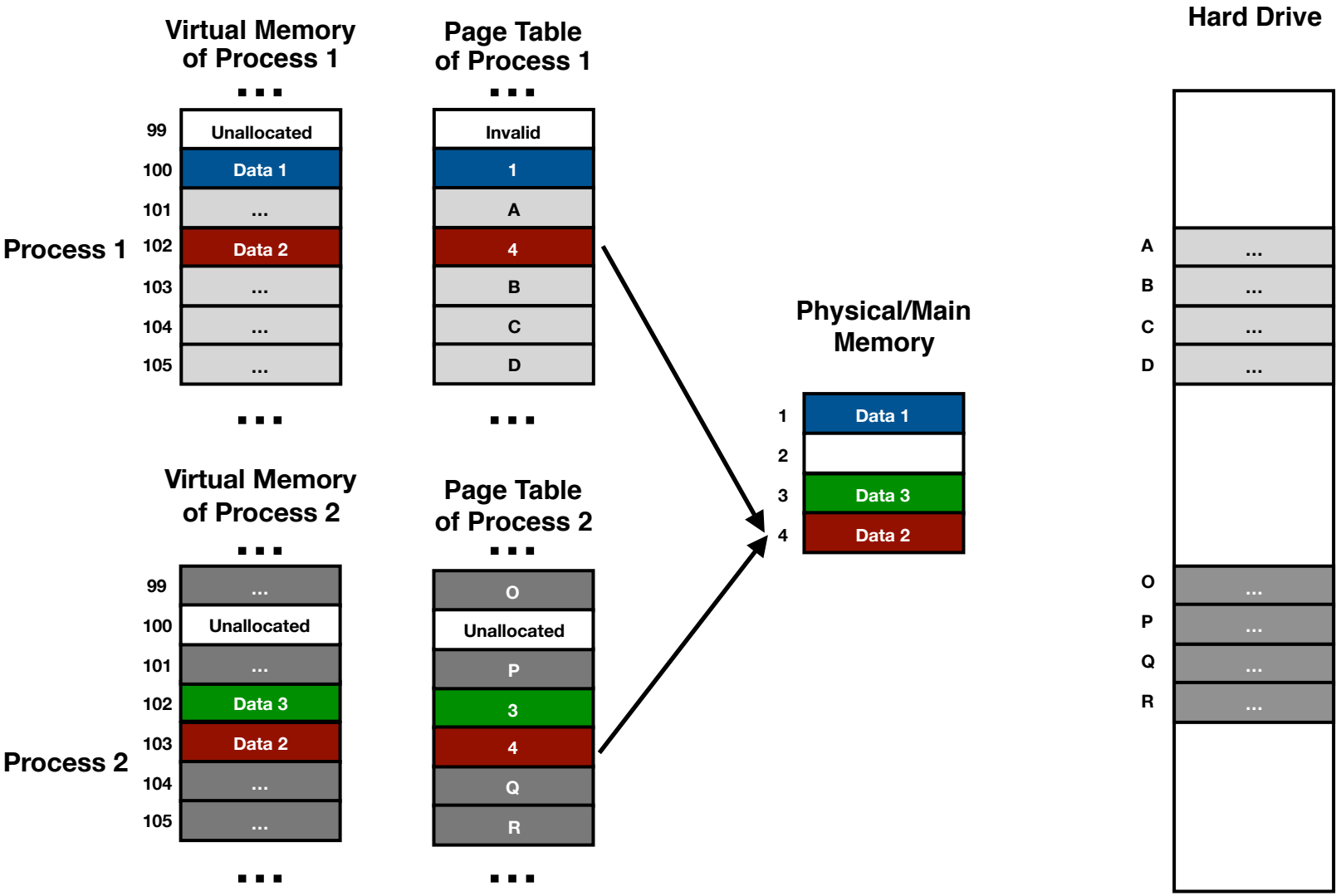
Prevent Unwanted Sharing



Prevent Unwanted Sharing



Enable Benign Sharing



Analogy for Virtual Memory: A Secure Hotel

Analogy for Virtual Memory: A Secure Hotel

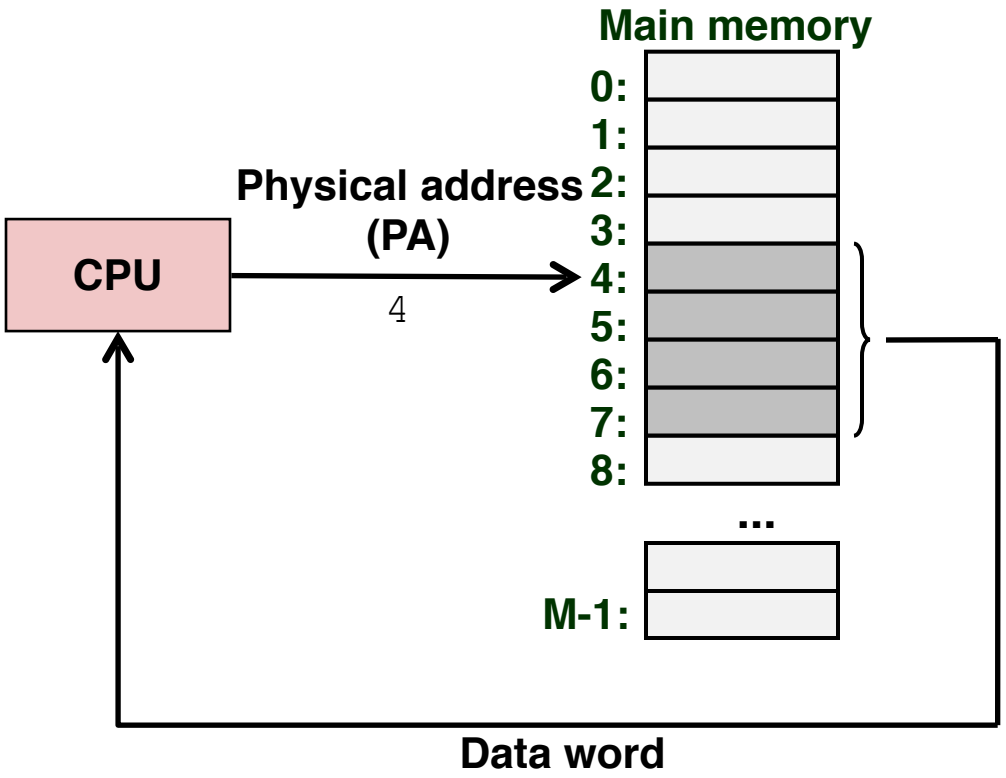
- Call a hotel looking for a guest; what happens?
  - Front desk routes call to room, does not give out room number
  - Guest's name is a virtual address
  - Room number is physical address
  - Front desk is doing address translation!

Analogy for Virtual Memory: A Secure Hotel

- Call a hotel looking for a guest; what happens?
  - Front desk routes call to room, does not give out room number
  - Guest's name is a virtual address
  - Room number is physical address
  - Front desk is doing address translation!

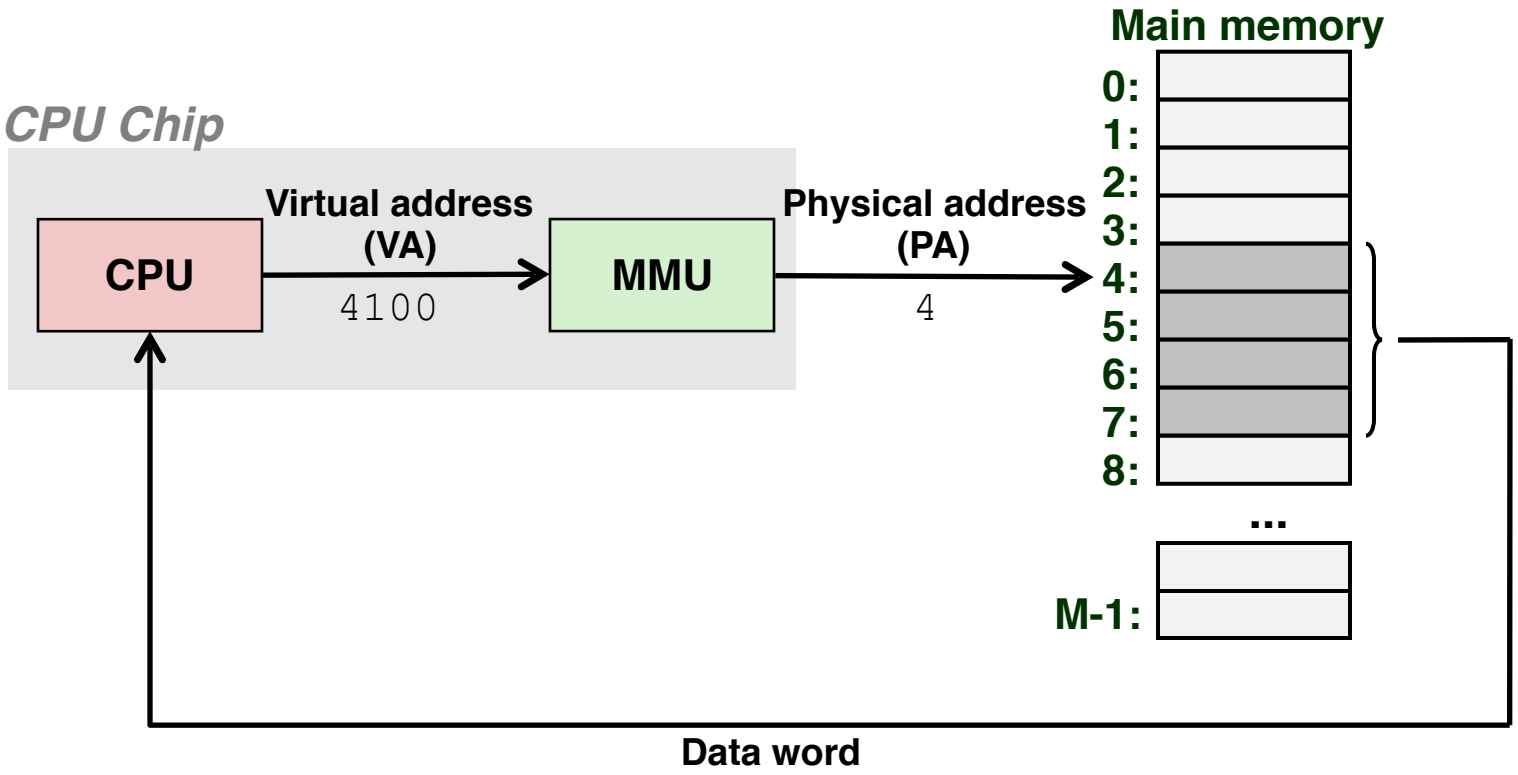
- Benefits
  - **Ease of management**: Guest could change rooms (physical address). You can still find her without knowing it
  - **Protection**: Guest could have block on calls, block on calls from specific callers (permissions)
  - **Sharing**: Multiple guests (virtual addresses) can share the same room (physical address)

A System Using Physical Memory Only



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Memory



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science (back in the 60s)
- MMU: Memory Management Unit; part of the OS

The Big Idea: Virtual Memory

- What Does a Programmer Want?

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers
  - Data in virtual memory by default stays in disk

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers
  - Data in virtual memory by default stays in disk
  - Data moves to physical memory (DRAM) “**on demand**”

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers
  - Data in virtual memory by default stays in disk
  - Data moves to physical memory (DRAM) "**on demand**"
  - Disks (~TBs) are much larger than DRAM (~GBs), but 10,000x slower.

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers
  - Data in virtual memory by default stays in disk
  - Data moves to physical memory (DRAM) "**on demand**"
  - Disks (~TBs) are much larger than DRAM (~GBs), but 10,000x slower.
  - Effectively, virtual memory system transparently share the physical memory across different processes

The Big Idea: Virtual Memory

- What Does a Programmer Want?
  - Infinitely large, infinitely fast memory
  - Strong isolation between processes to prevent unwanted sharing
  - Enable wanted sharing
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers
  - Data in virtual memory by default stays in disk
  - Data moves to physical memory (DRAM) "**on demand**"
  - Disks (~TBs) are much larger than DRAM (~GBs), but 10,000x slower.
  - Effectively, virtual memory system transparently share the physical memory across different processes
  - Manage the sharing automatically: hardware-software collaborative strategy (too complex for hardware alone)