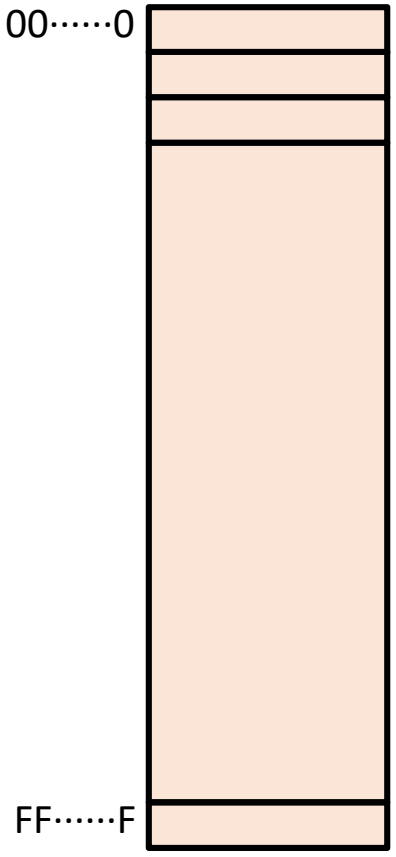
Lecture 15: Virtual Memory

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

- **Introduction to virtual memory**
- Mapping virtual to physical memory
- Dynamic relocation using base/bound registers
- Short discussion on mid semester feedback

Virtual Memory

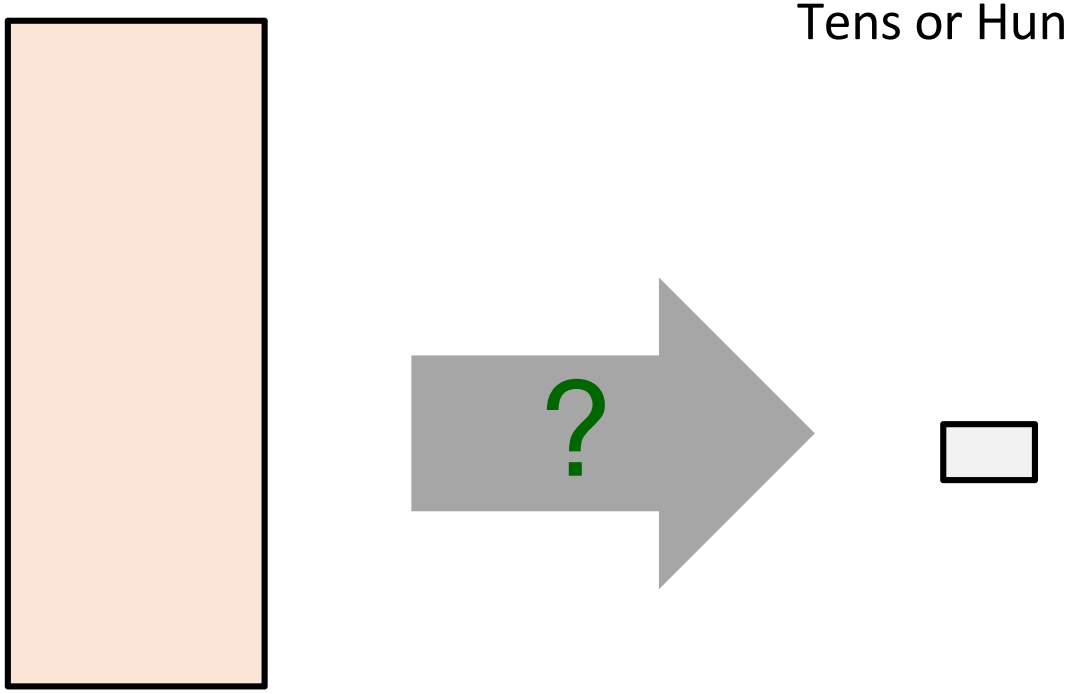


- Programs refer to memory using virtual memory addresses
  - `int* ptr = malloc(4);`
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Operating system provides address space private to particular “process”
- Compiler and run-time system allocate VM
  - Where different program objects should be stored
  - All allocation within single virtual address space

Problem 1: How Does Everything Fit?

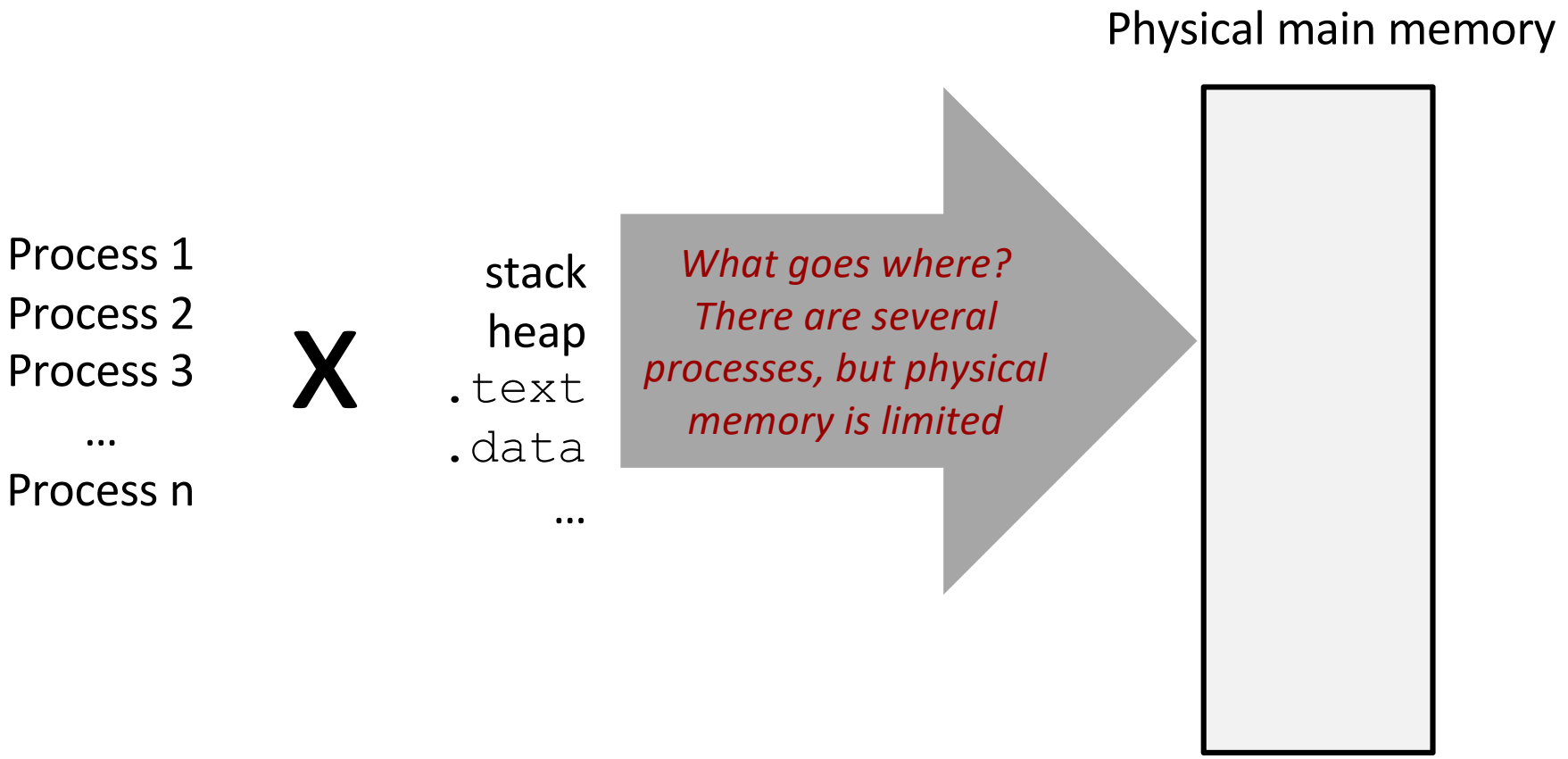
64-bit addresses:
16 Exabyte (16 billion GB!)

Physical main memory:
Tens or Hundreds of Gigabytes

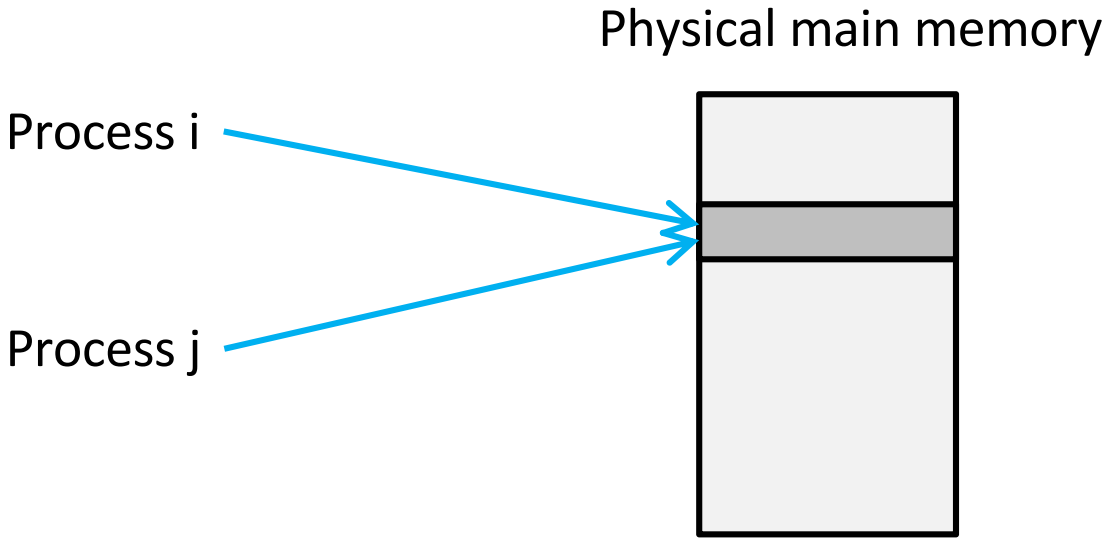


And there are many processes ….

Problem 2: Memory Management

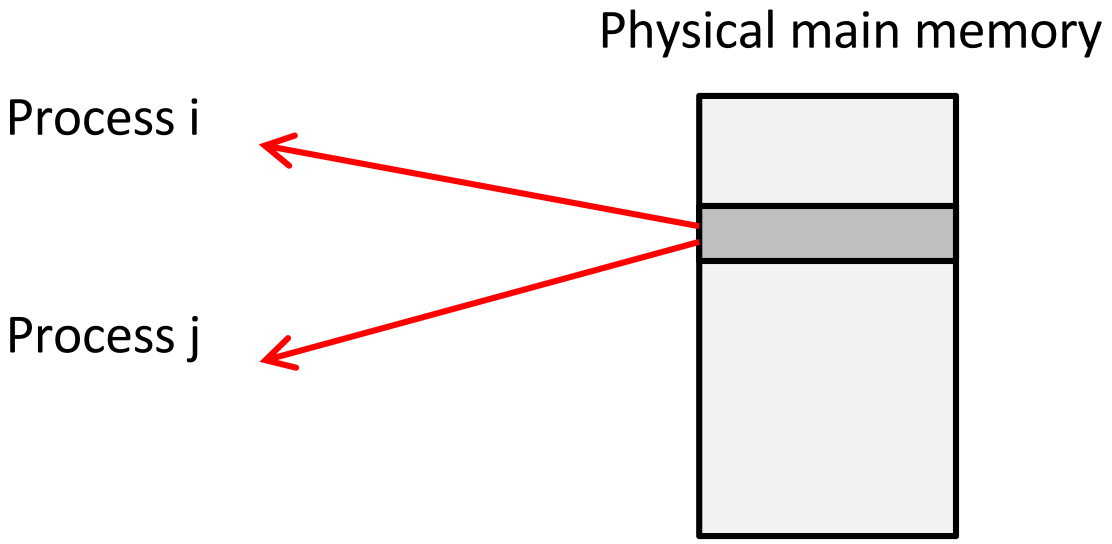


Problem 3: How To Protect?



- How to prevent Process-1 accessing the memory owned by Process-2 (unless shared)?

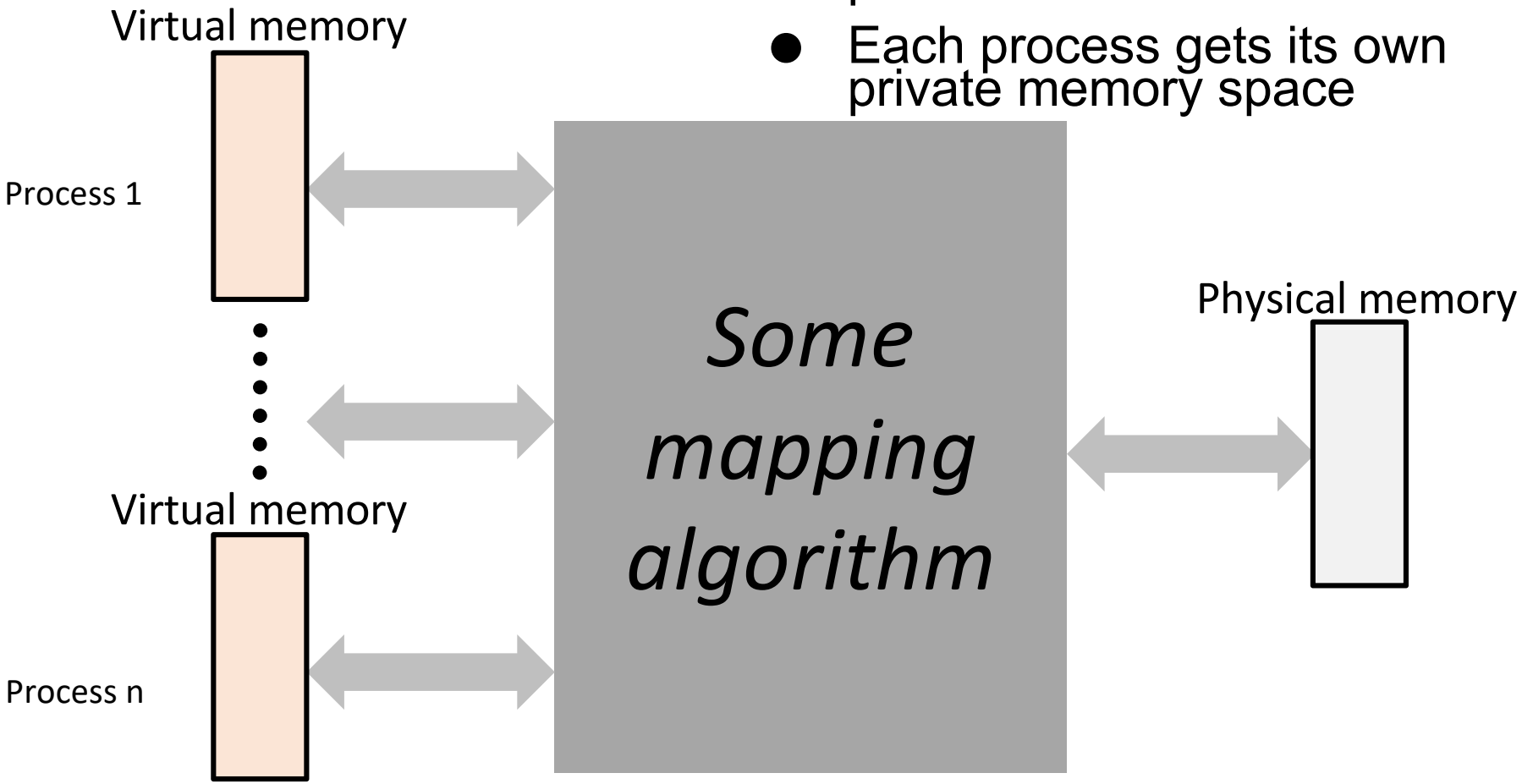
Problem 4: How To Share?



- How to share memory between two processes?

Solution: Indirection

- Using some mapping technique can solves all the previous problems
- Each process gets its own private memory space



Address Spaces

- Using some mapping technique to translate the address seen by the user to the actual address on the RAM gives two different views of addresses
  - Virtual address space (as seen by the process)
  - Physical address space (as seen by the OS depending on DRAM size)

- Virtual address space: Set of $N = 2^n$ virtual addresses
  {0, 1, 2, 3, …, N-1}

- Physical address space: Set of $M = 2^m$ physical addresses
  {0, 1, 2, 3, …, M-1}

Virtual Memory (VM)

- Efficient use of limited main memory (RAM)
  - Use RAM as a cache for parts of a virtual address space
    - Some non-cached parts stored on disk
  - Keep only active areas of virtual address space in RAM
    - Transfer data back and forth to disk as needed
- Simplifies memory management for programmers
  - Each process gets a full private address space
- Isolates address spaces
  - One process can't interfere with another's memory
    - Because they operate in different address spaces

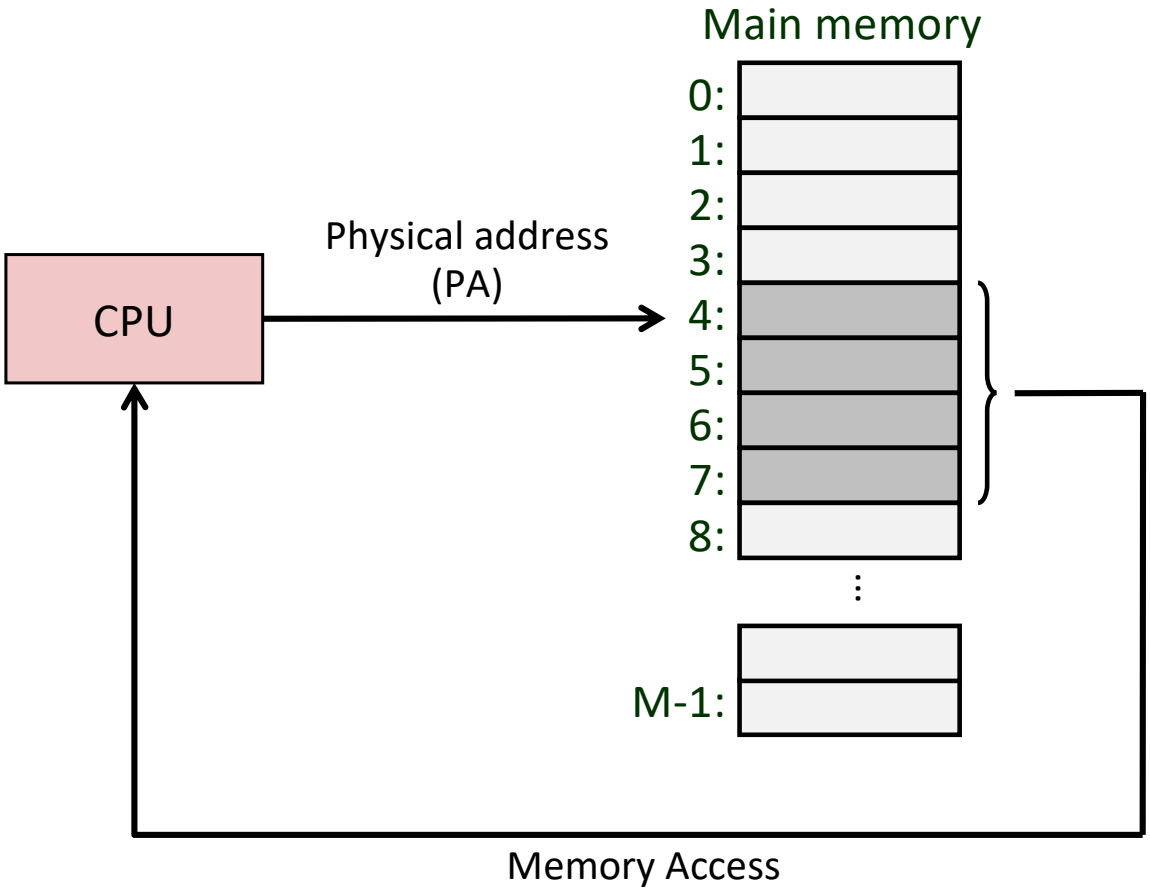
Today's Class

- Introduction to virtual memory
- **Mapping virtual to physical memory**
- Dynamic relocation using base/bound registers
- Short discussion on mid semester feedback

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address

A System Using Physical Addressing

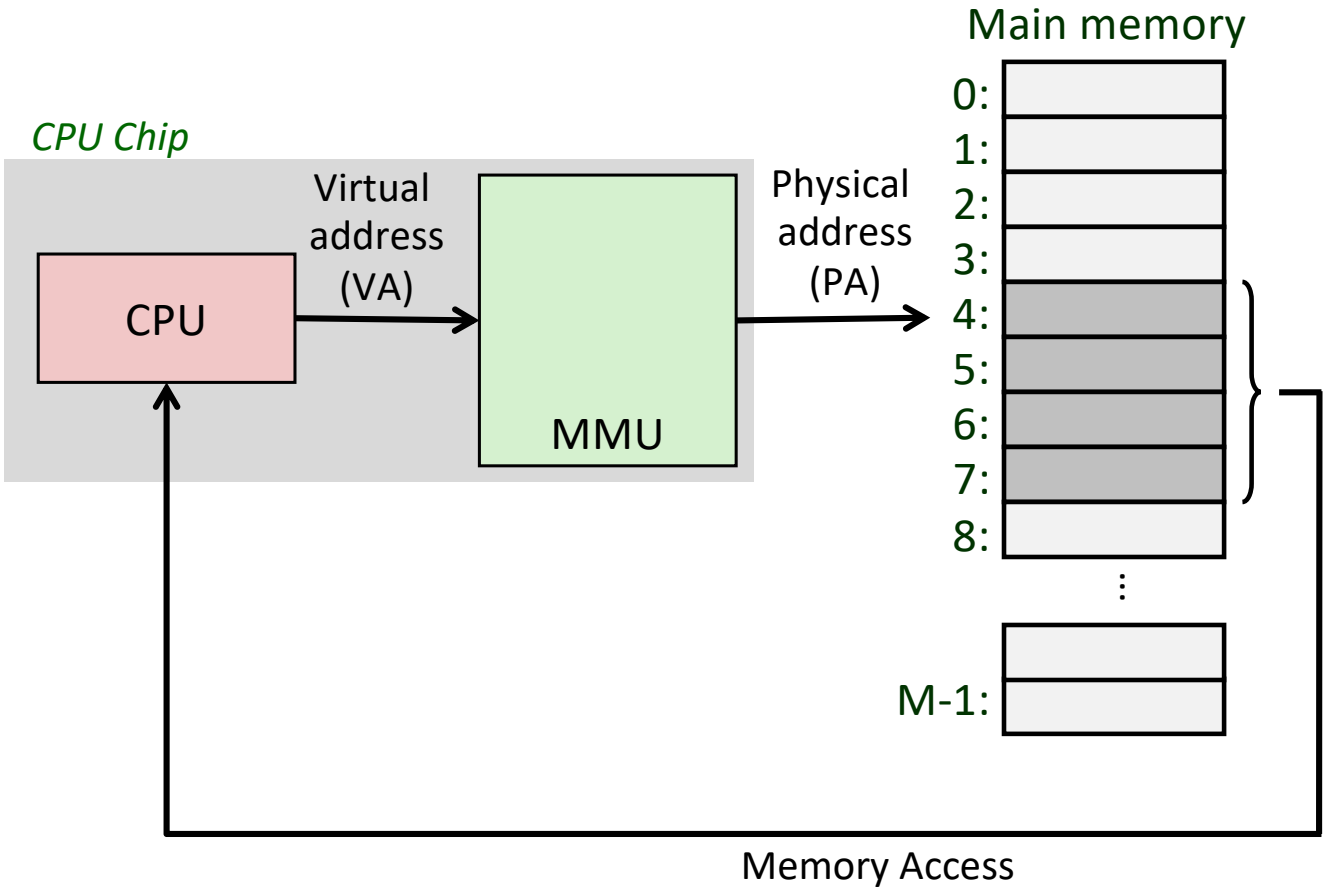


- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
  - Lowers production cost of the hardware (limited support of MMU)

Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address

A System Using Virtual Addressing



- Used in all modern servers, desktops, laptops, tablets, and cell phones
  - Why we cannot have VA same as PA in this case?

A System Using Virtual Addressing

```
vivek@possum:~/os23$ readelf -a ./fib
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independer
  Machine:                           Advanced Micro Devices
  Version:                           0x1
  Entry point address:               0x1080
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14056 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

- Can two processes have same virtual addresses?

  $./fib &

  $./fib &

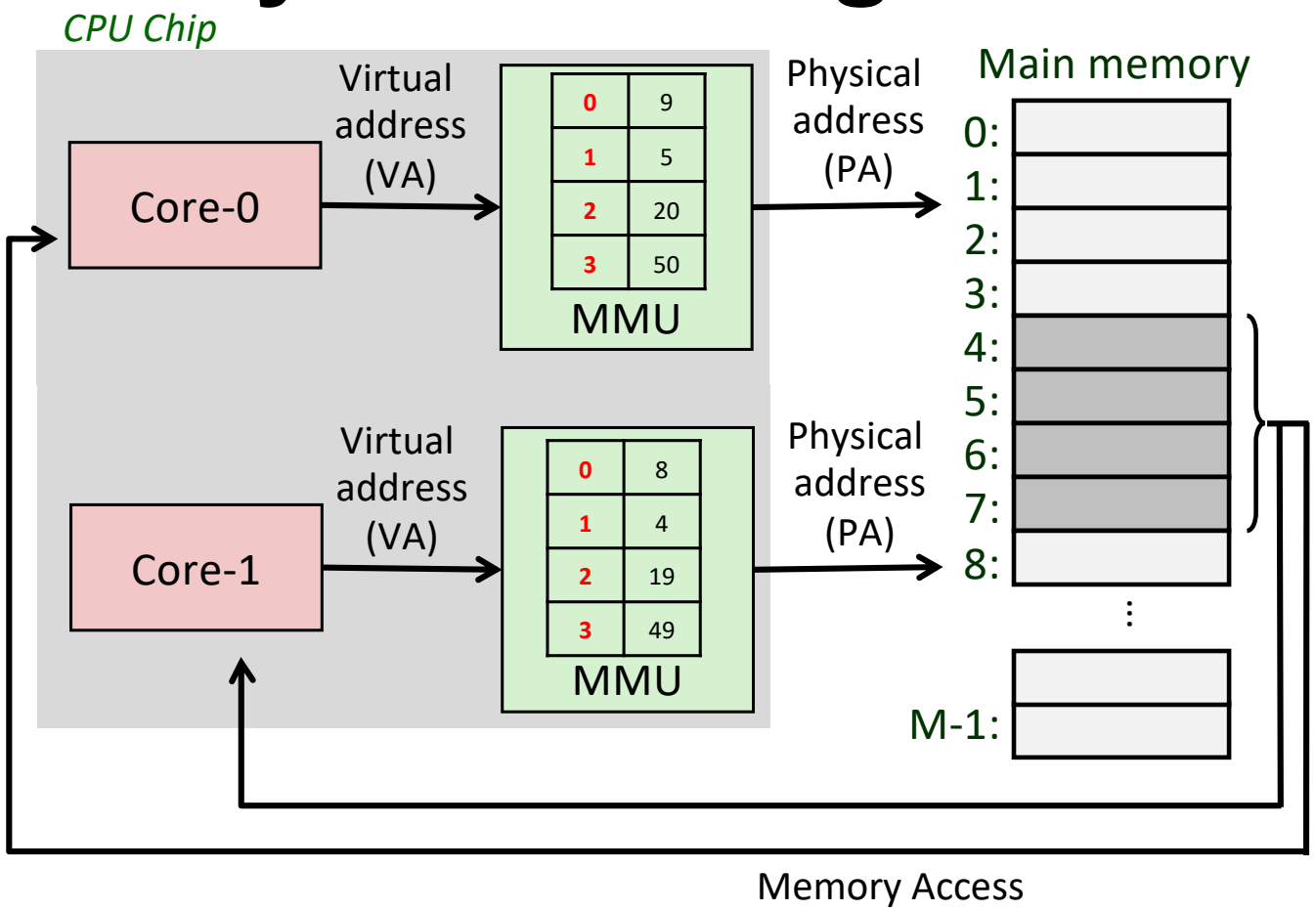
  $./fib &

  $./fib &

  $./fib &

  $./fib &

A System Using MMU Table



- Can two processes have same virtual addresses?
  - Yes, as long as the VA-PA mapping is different for both processes, and they are reflected in the **MMU table** during scheduling

A System Using MMU Table

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_mmu_table(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

- Can two processes have same virtual addresses?
  - Yes, as long as the VA-PA mapping is different for both processes, and they are reflected in the MMU table during scheduling
- How to support?
  - Save/restore the MMU table during context switching of the processes at each core

Issues with MMU Table Based Approach

- A process is not just going to have a few virtual addresses
  - It would be accessing several virtual addresses throughout its execution
- Not possible to store the entire VA-PA mapping inside per-process MMU table
  - Huge overhead in the operating system
- How about assigning a contiguous chunk of physical memory to each process?
  - **Any issues you can foresee?**
  - Nonetheless, let us see how to support it

Today's Class

- Introduction to virtual memory
- Mapping virtual to physical memory
- **Dynamic relocation using base/bound registers**
- Short discussion on mid semester feedback

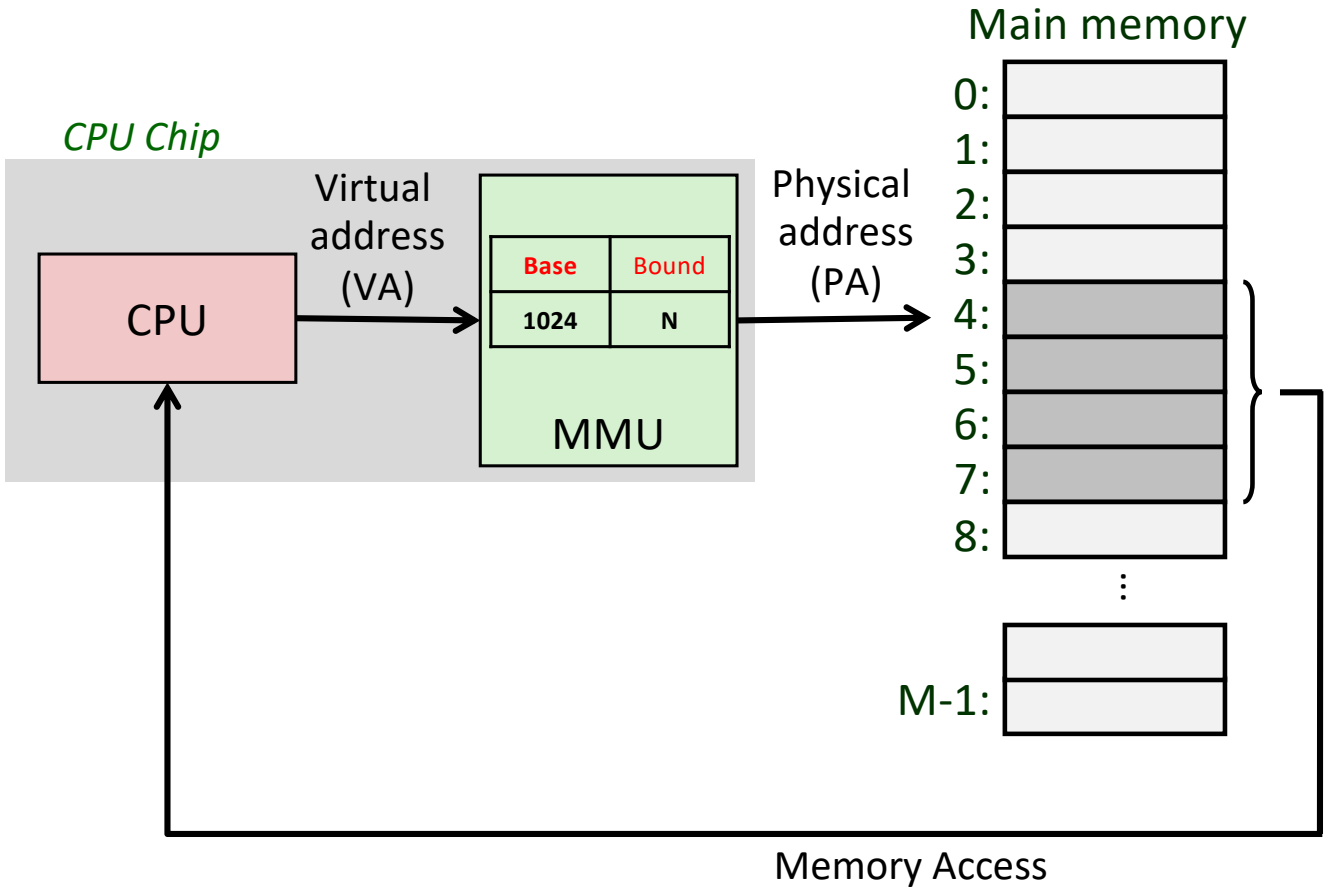
Dynamic Relocation

- “Relocation” means fixing an “indirection”
- Recall the ”relocation” from the linker/loader lecture
  - In case of linker/loader, the linker created an “indirection” by just providing the pointers to the machine code inside the shared libraries
  - Loader used dynamic relocation during execution time by patching the pointers to the exact location of the machine code (e.g., printf’s actual implementation in glibc)
- Similar approach of dynamic relocation can be used by the OS to decide the physical memory mapping for each processes

Mapping Virtual to Physical Address

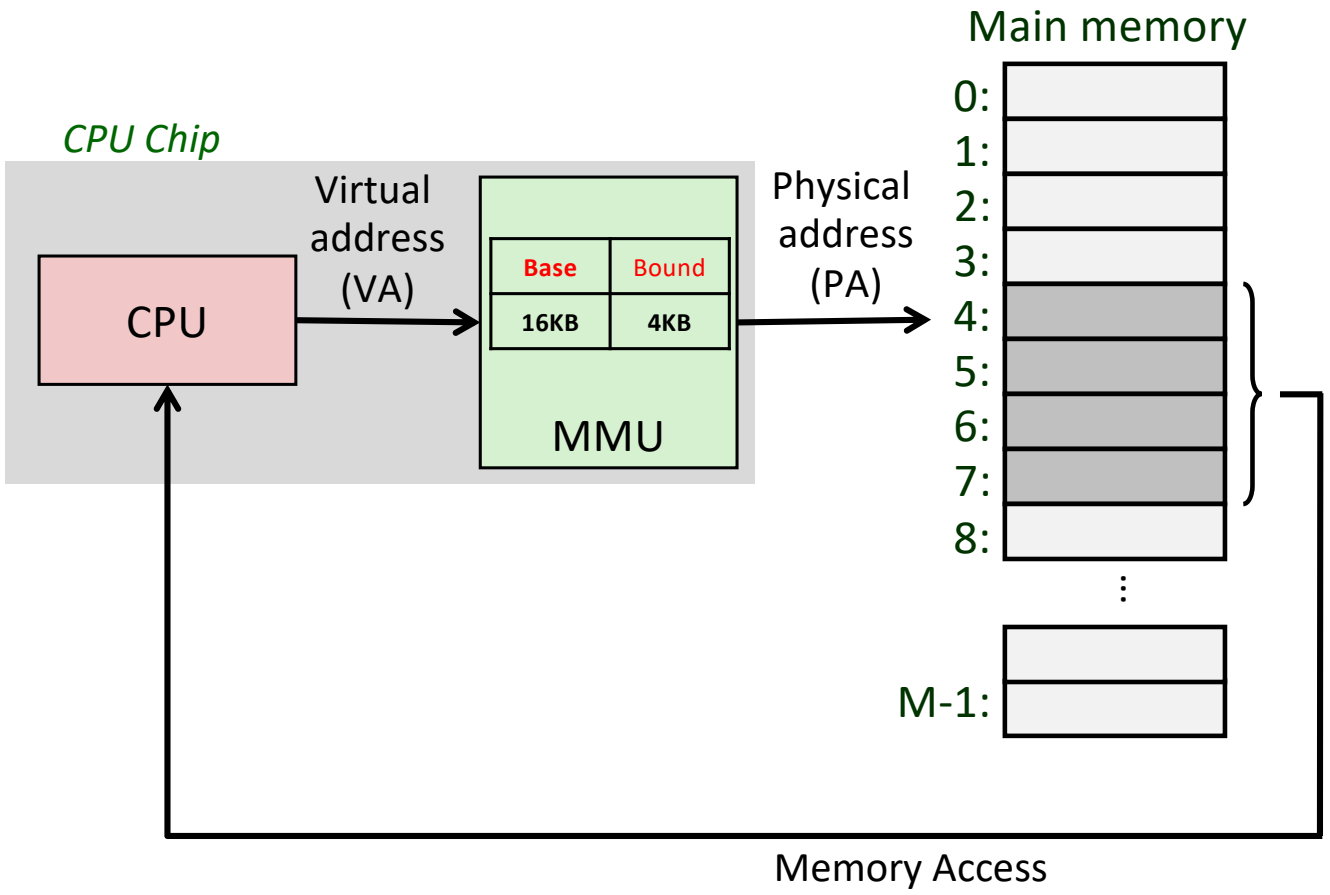
- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound registers for VA to PA mapping

Approach-1: Using Base/Bound Registers



- Assume, during "exec", the loader somehow knows the exact memory requirement of the process (e.g., **N** bytes)
- OS will allocate a **contiguous** chunk of physical memory (a.k.a. **segment**) of **N** bytes for this process starting at some address. E.g., **1024**
- MMU will now have a pair of registers called as base (=**1024**) and bound (=**N** bytes)

Example: Base and Bound



- Each process will have a unique **base** value, but bound (**limit**) may be same for some processes (e.g., launching several fib executable)
- During execution, the value of base and bound can be loaded into respective registers to help in address translation from VA to PA
- Example for an address translation: A process with 4KB address space placed at index 16KB in physical memory

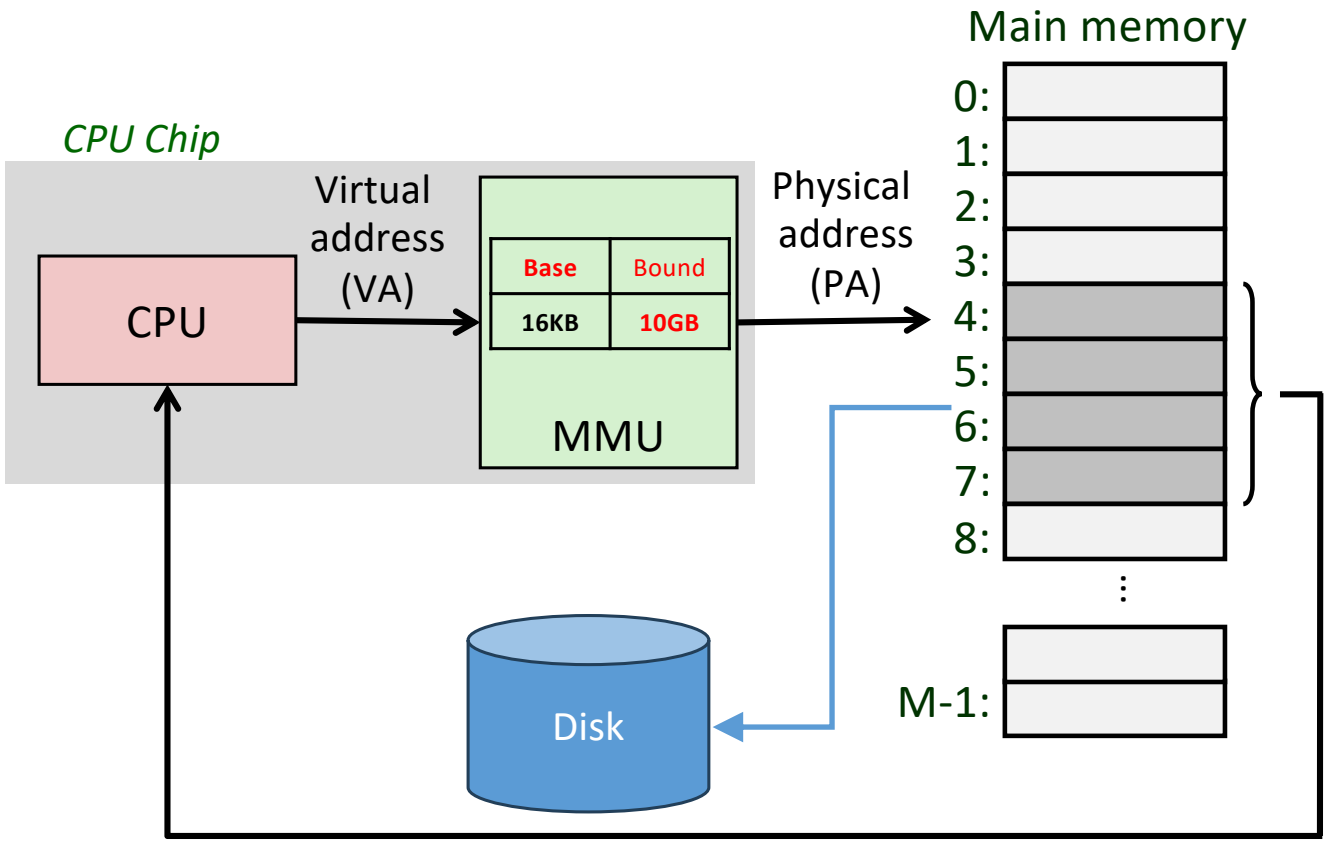
Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

A System Using Base/Bound Registers

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table))
{
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch_mmu_bbRegisters(scheduler_process, p);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

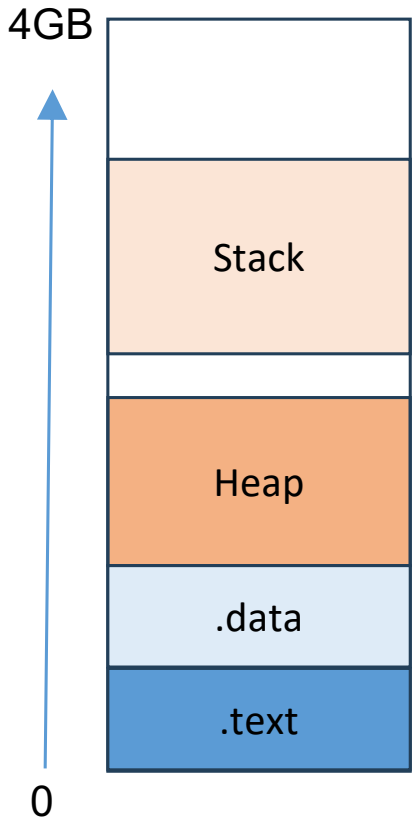
- swtch_mmu_bbRegisters (p1, p2)
  - Simply save the content of base/bound registers of currently executing P1 process into its PCB, and load the base/bound registers from P2's PCB

Reusing Same Physical Addresses



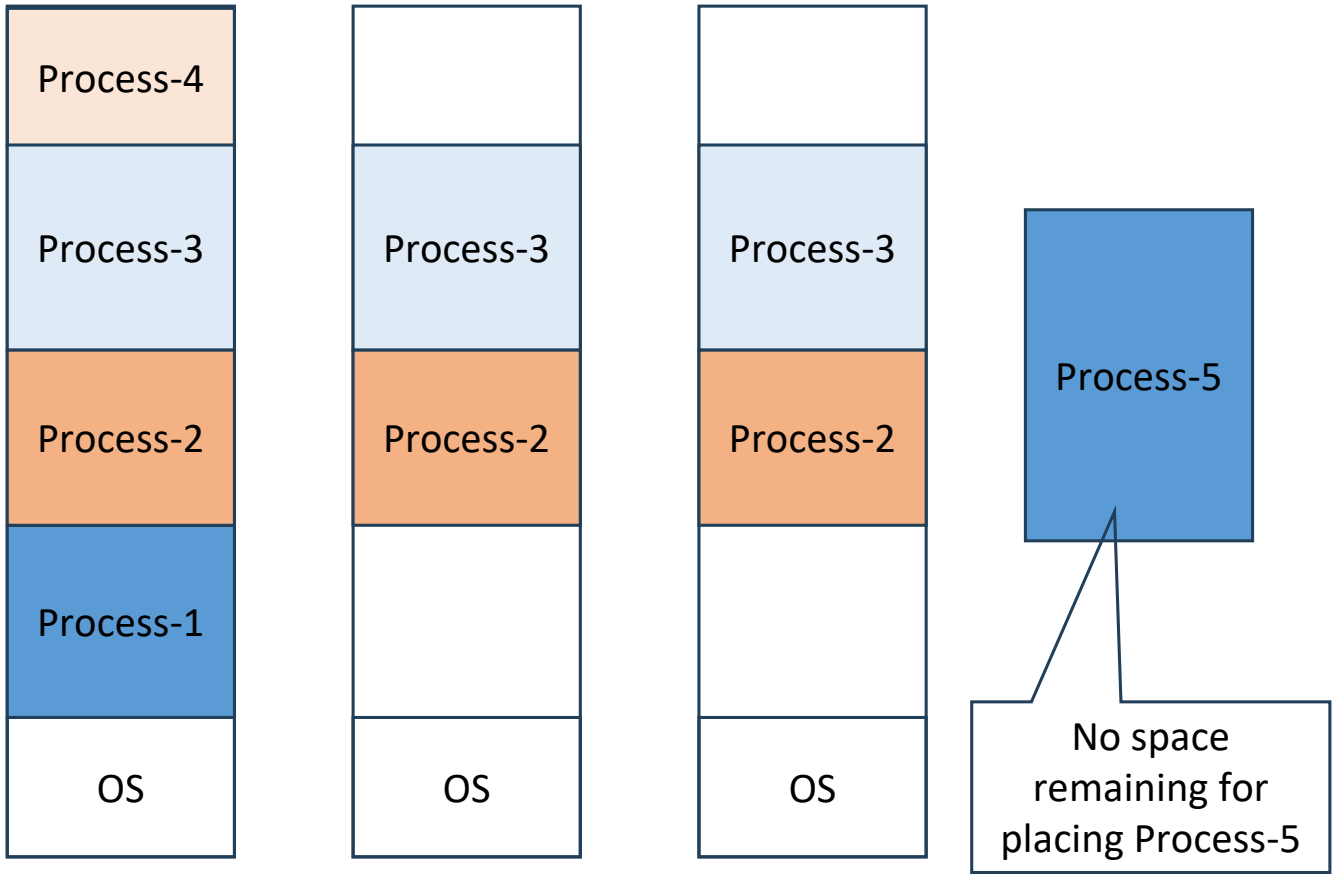
- Let us consider a situation where only one process P1 is running on a single-core processor and requires a huge address space that took away a major chunk of DRAM
- How to run another process P2 on this processor as it requires huge amount of address space?
  - Solution: Both could have overlapping physical address space
  - Swap out the physical address space of the running process to the disk during context switch
  - Huge overhead, but doable!

Fragmentation with Base/Bound (1/2)



- Large gap in address space between stack and heap
- Range of memory is unused, but still allocated to process
- Our approach-1 will lead to fragmentation
  - Internal or external fragmentation in this case?

Fragmentation with Base/Bound (2/2)



- There would also be situation that no contiguous free space is left for the address space of a new process
  - What type of fragmentation is this?
- Any other issues?
  - Can we create shared memory between processes?

Base/Bound: Summary

- Simple approach
  - Finding space for a new process's address space in memory
  - OS must *save and restore* the base-and-bounds pair at context switches
  - OS must provide exception handlers to be invoked if a process tries to access memory outside its bounds
  - OS will be able to move an address space of a stopped process from one location in memory to another
  - Reclaiming all of its memory at process termination
- Impractical to implement
  - Mandates OS to allocate contiguous chunk of physical address space to each process

Reference Material

- You can read chapter 15 of OSTEP book (Address Translation)

Next Lecture

- Segmentation
- Quiz-3 on 24th October
  - Syllabus: lectures 13, 15-17