Virtual Memory

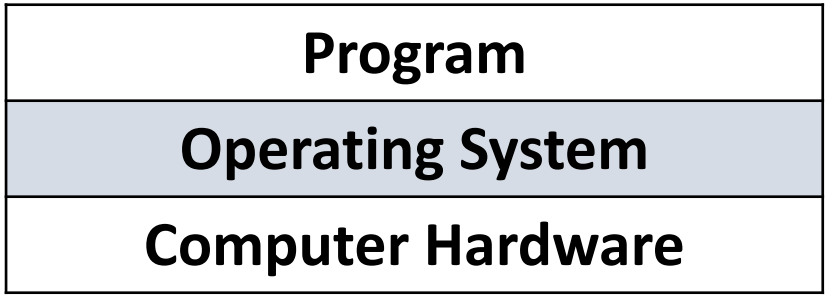
11/8/16

(election day)

Vote!

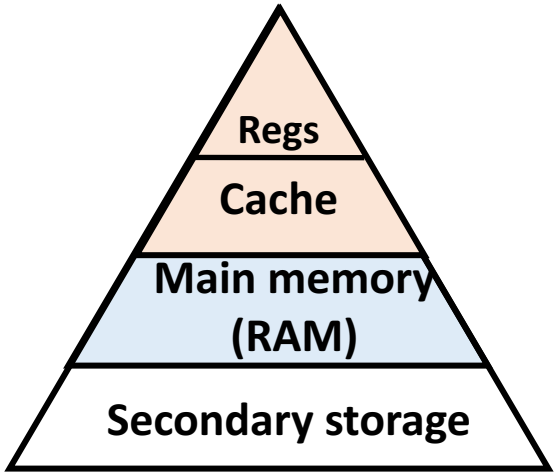
Recall: the job of the OS

The OS is an interface layer between a user's programs and hardware.



It provides an abstract view of the hardware that makes it easier to use.

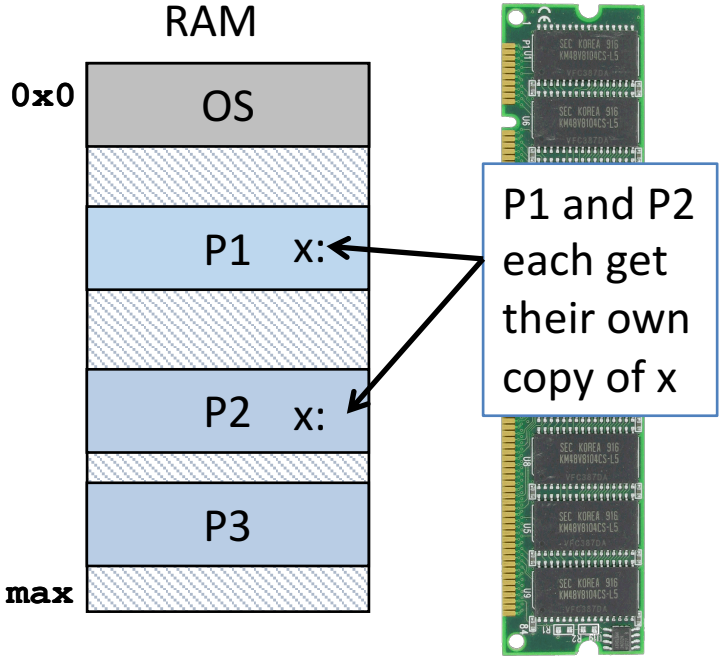
What we Know about Physical Memory (RAM)



RAM is array of addressable bytes, from 0x0 to max

(ex) address 0 to 2^{30}-1 for 1 GB of RAM space

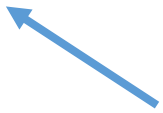
- The OS needs to be in RAM
  - Usually loaded at address 0x0
- Physical Storage for running processes: Process Address Spaces are stored in RAM



How much memory is allocated?

```
int main(){
  int i;
  printf("a stack address: %p\n", &i);
  printf("a text address:  %p\n", main);
  printf("the difference:  %d\n", (unsigned int)(&i) -
                                  (unsigned int)(main));
}
```

```
a stack address: 0x7fff7f1e9774
a text address:  0x400596
the difference:  2128515550
```



> 2 gigabytes

The virtual memory abstraction

Solves two problems:

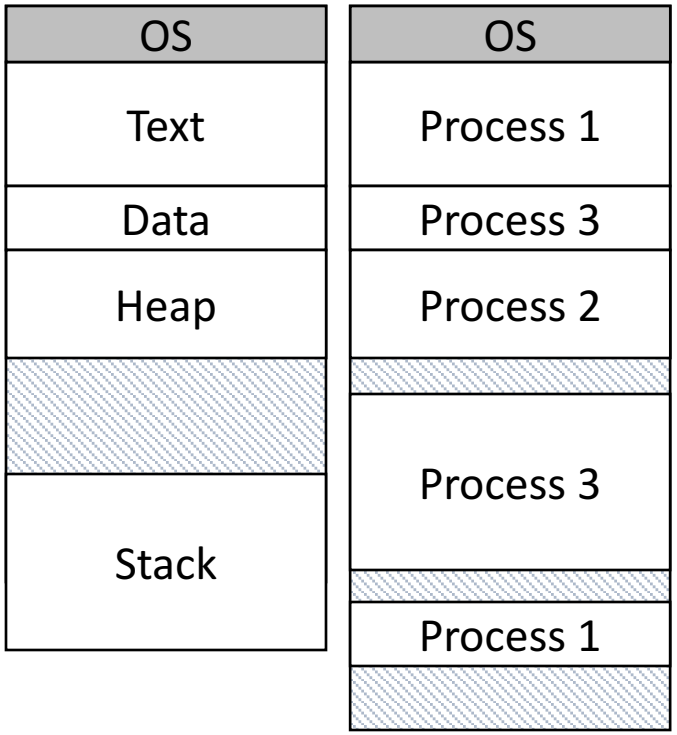
- Not enough physical memory to go around.
- Don’t want multiple programs to accidentally modify the same physical memory location.

Key ideas:

- OS allocates memory to processes as needed.
- Program’s addresses get translated to physical addresses.

Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.
- Reality: there's only so much memory to go around, and no two processes should use the same (physical) memory addresses.

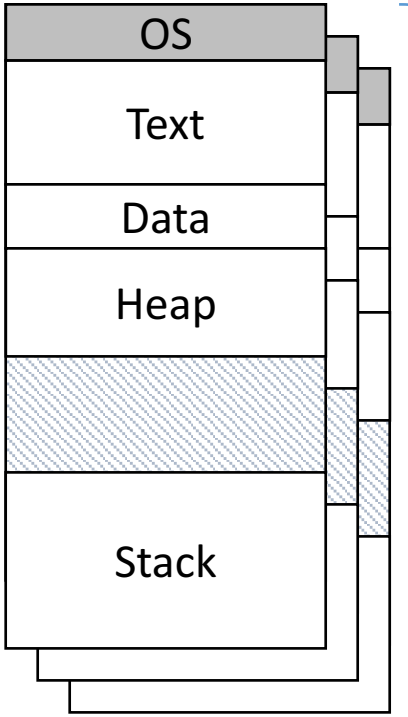


OS (with help from hardware) will keep track of who's using each memory region.

Memory Terminology

Virtual (logical) Memory: The abstract view of memory given to processes. Each process gets an independent view of the memory.

Physical Memory: The contents of the hardware (RAM) memory. Managed by OS. Only ONE of these for the entire machine!



0x0

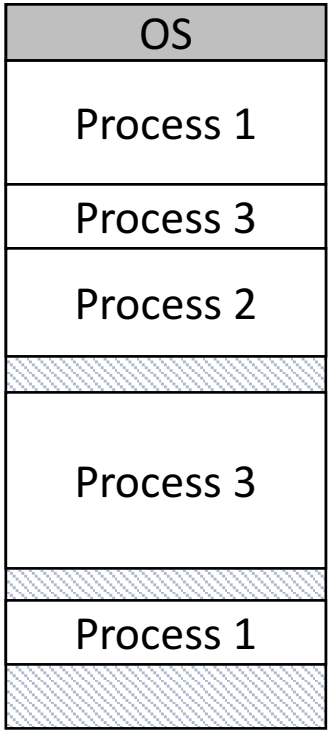
0xFFFFFFFF

Virtual address space (VAS): fixed size.

Address Space: Range of addresses for a region of memory.

The set of available storage locations.

0x0



0x…

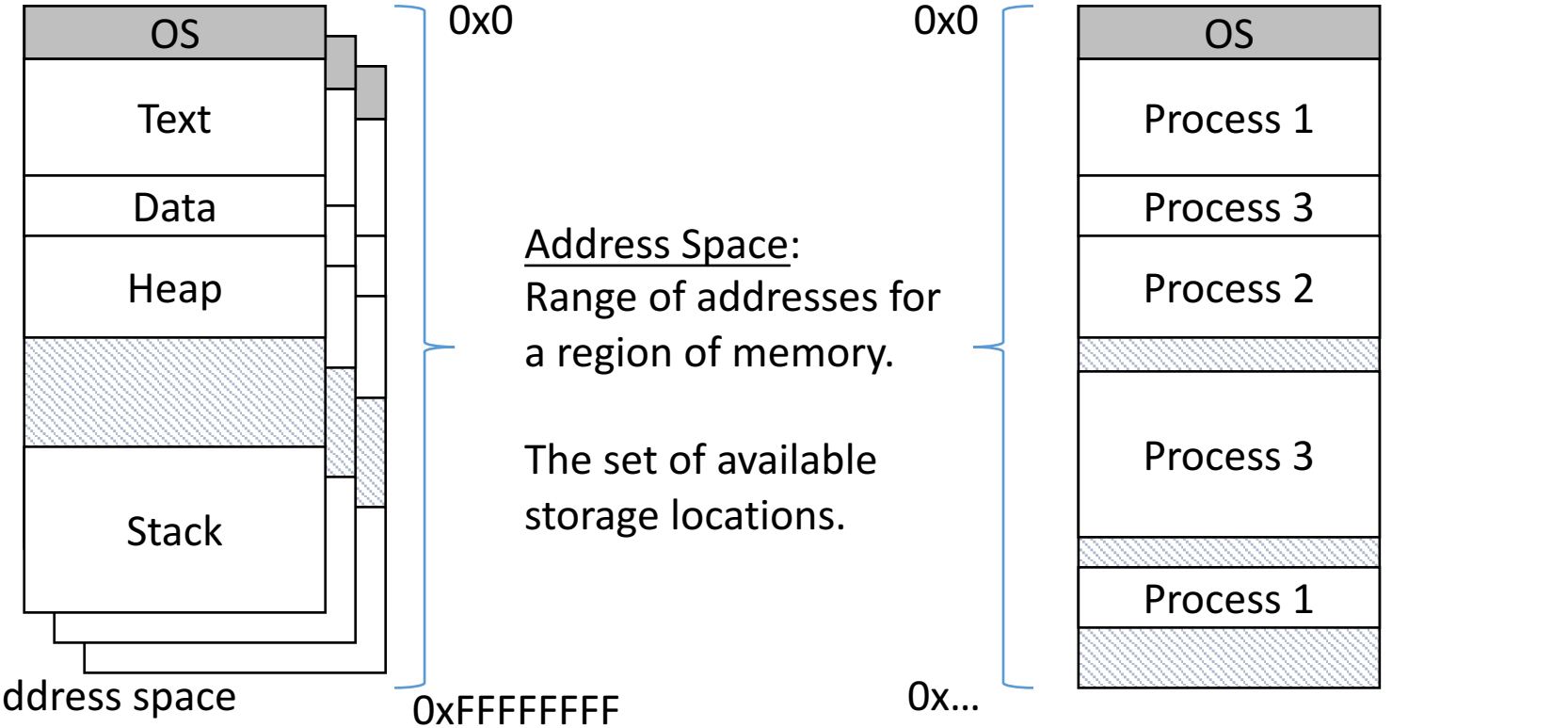
(Determined by amount of installed RAM.)

Memory Terminology

Note: It is common for VAS to appear larger than physical memory.

32-bit (IA32): Can address up to 4 GB, might have less installed

64-bit (X86-64): Our lab machines have 48-bit VAS (256 TB), 36-bit PAS (64 GB)



Cohabitating Physical Memory

- If process is given CPU, must also be in memory.
- Problem
  - Context-switching time (CST): 10 µsec
  - Loading from disk: 10 MB/s
  - To load 1 MB process: 100 msec = 10,000 x CST
  - Too much overhead! Breaks illusion of simultaneity
- Solution: keep multiple processes in memory
  - Context switch only between processes in memory

Memory Issues and Topics

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

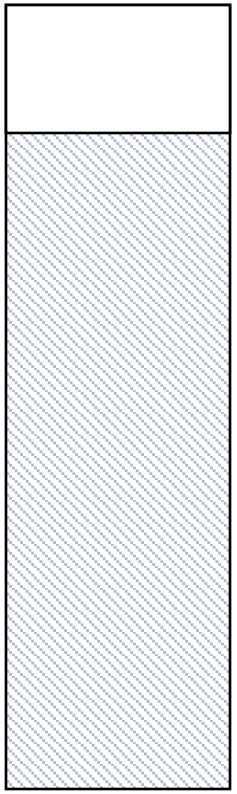
Plan: Start with the basics (out of date) to motivate why we need the complex machinery of virtual memory and paging.

Problem: Placement

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

Memory Management

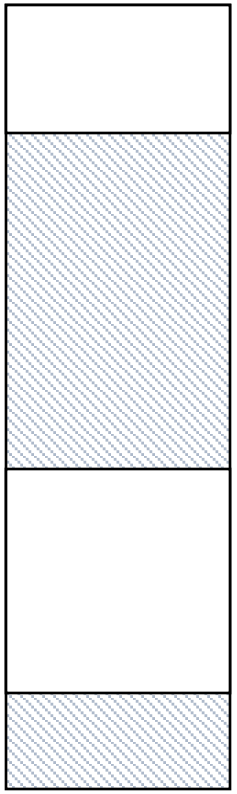
- Physical memory starts as one big empty space.



Memory Management

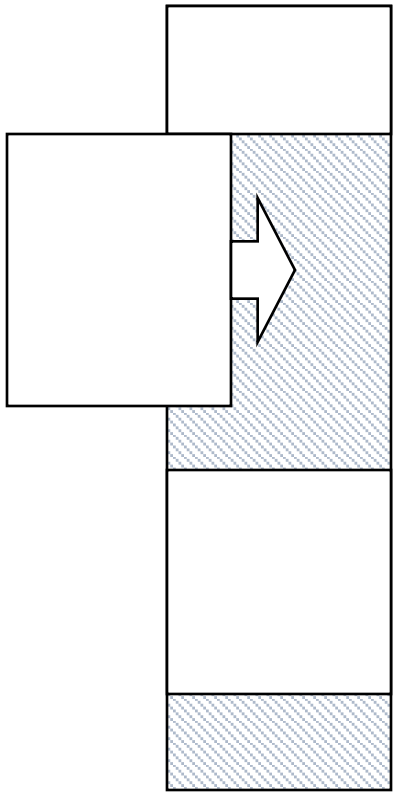
- Physical memory starts as one big empty space.

- Processes need to be in memory to execute.



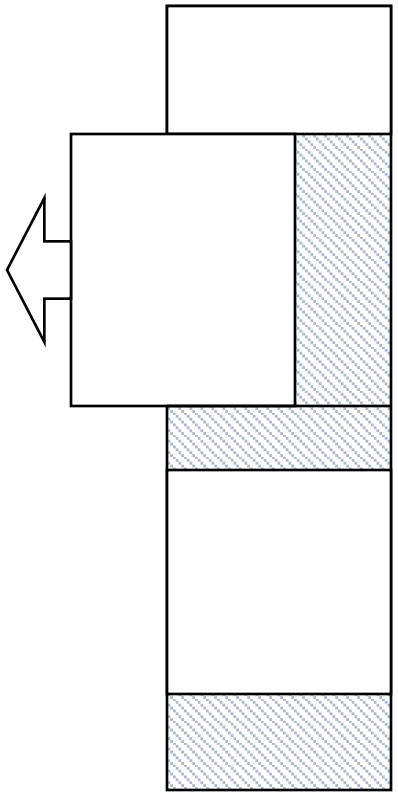
Memory Management

- Physical memory starts as one big empty space.
- When creating process, allocate memory
  - Find space that can contain process
  - Allocate region within that gap
  - Typically, leaves a (smaller) free space



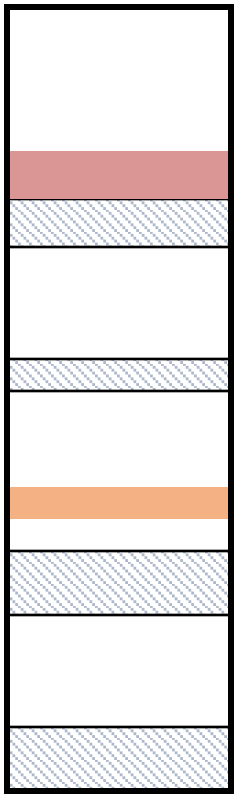
Memory Management

- Physical memory starts as one big empty space.
- When creating process, allocate memory
  - Find space that can contain process
  - Allocate region within that gap
  - Typically, leaves a (smaller) free space
- When process exits, free its memory
  - Creates a gap in the physical address space.
  - If next to another gap, coalesce.



Fragmentation

- Eventually, memory becomes fragmented
  - After repeated allocations/de-allocations
- Internal fragmentation
  - Unused space within process
  - Cannot be allocated to others
  - Can come in handy for growth
- External fragmentation
  - Unused space outside any process (gaps)
  - Can be allocated (too small/not useful?)



Which form of fragmentation is easiest for the OS to reduce/eliminate?

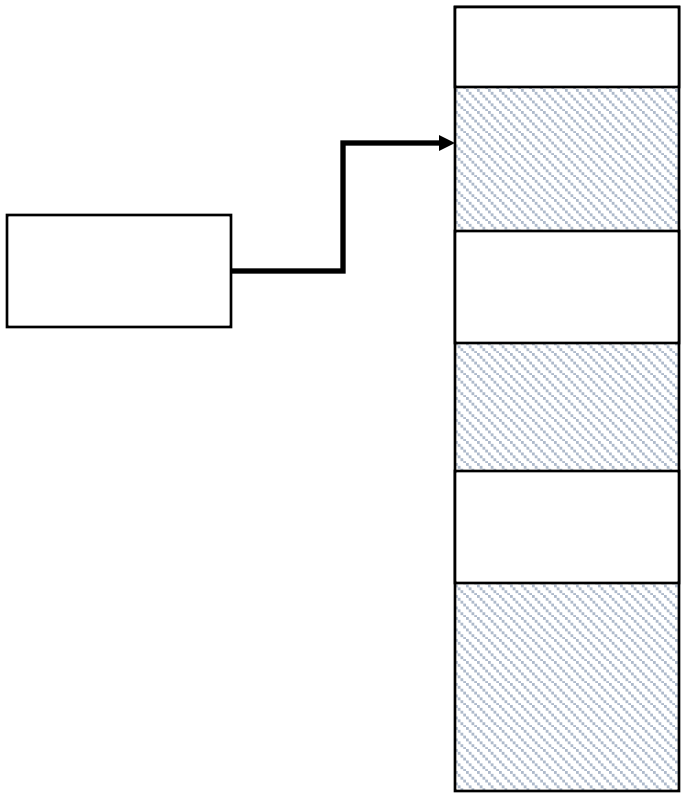
A. Internal fragmentation

B. External fragmentation

C. Neither

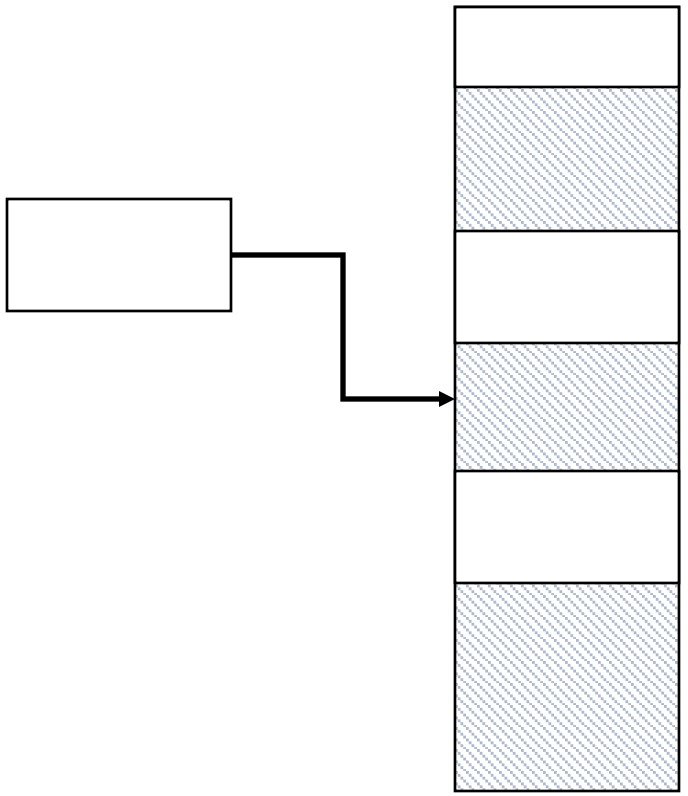
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - *First (or next) fit*
  - Best fit
  - Worst fit
- Complication
  - Is region fixed or variable size?



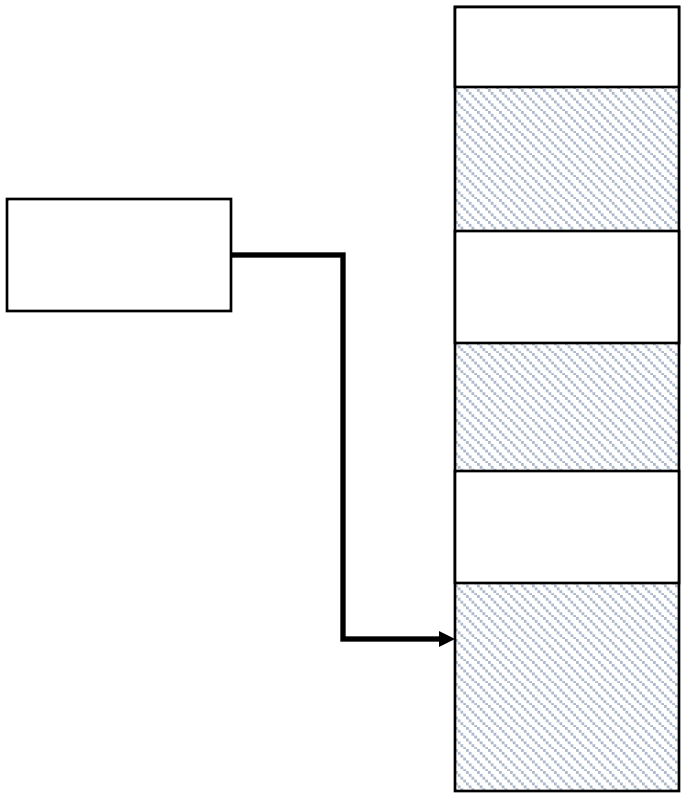
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - *Best fit*
  - Worst fit
- Complication
  - Is region fixed or variable size?



Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - Best fit
  - *Worst fit*
- Complication
  - Is region fixed or variable size?



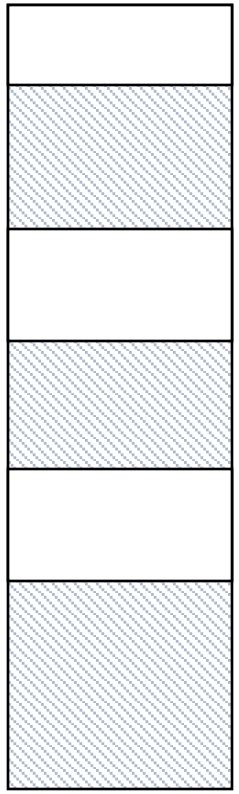
Which memory allocation algorithm leaves the smallest fragments (external)?

A. first-fit

B. worst-fit

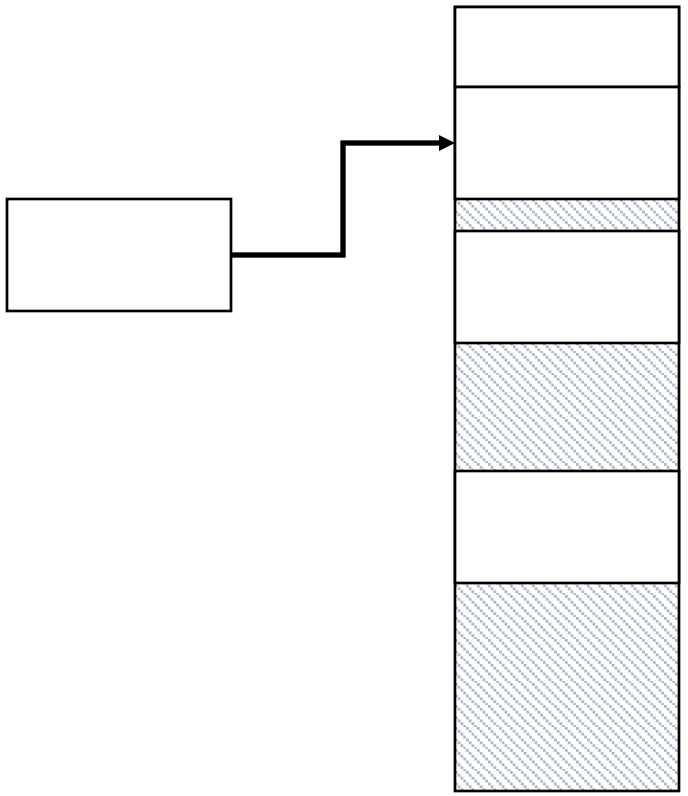
C. best-fit

Is leaving small fragments a good thing or a bad thing?



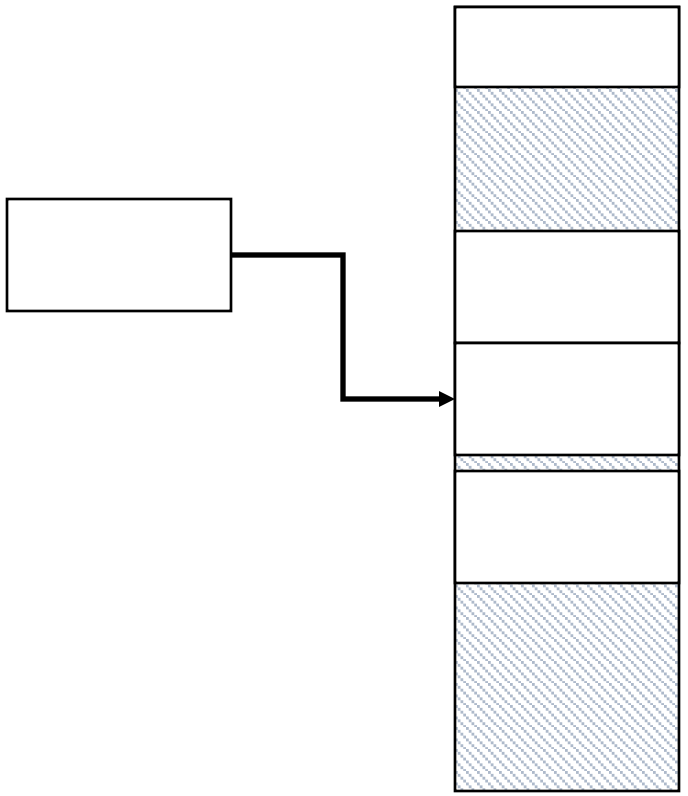
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - *First (or next) fit: fast*
  - Best fit
  - Worst fit
- Complication
  - Is region fixed or variable size?



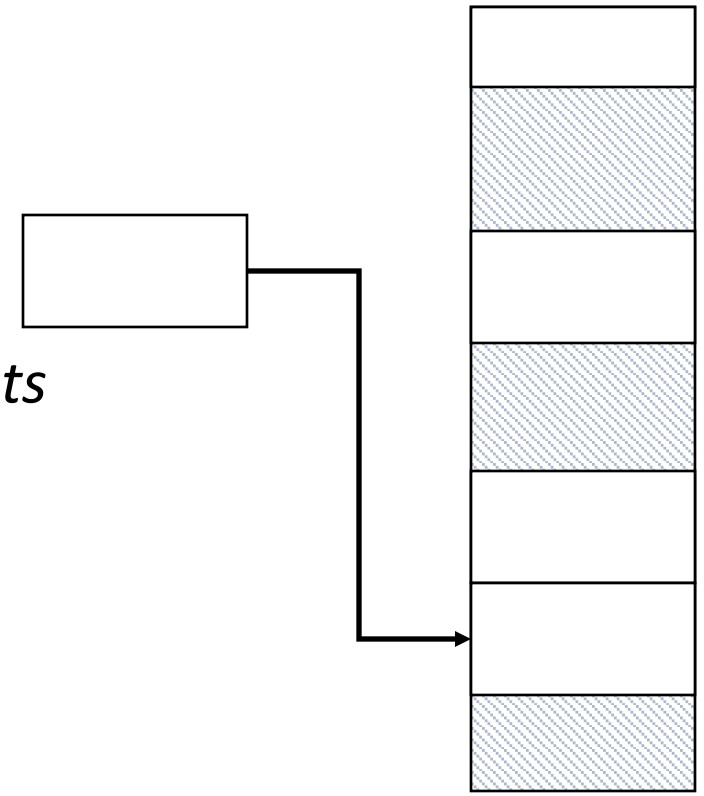
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - *Best fit: leaves small fragments*
  - Worst fit
- Complication
  - Is region fixed or variable size?



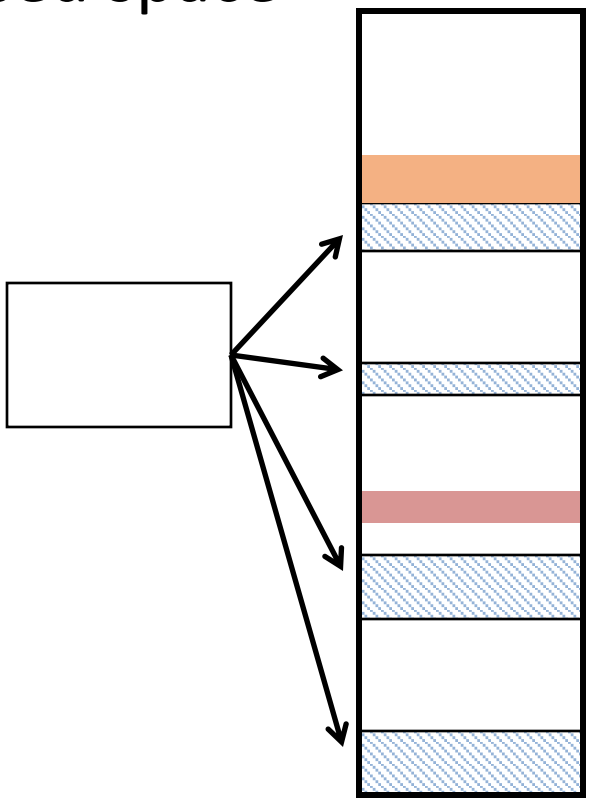
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - Best fit
  - *Worst fit: leaves large fragments*
- Complication
  - Is region fixed or variable size?



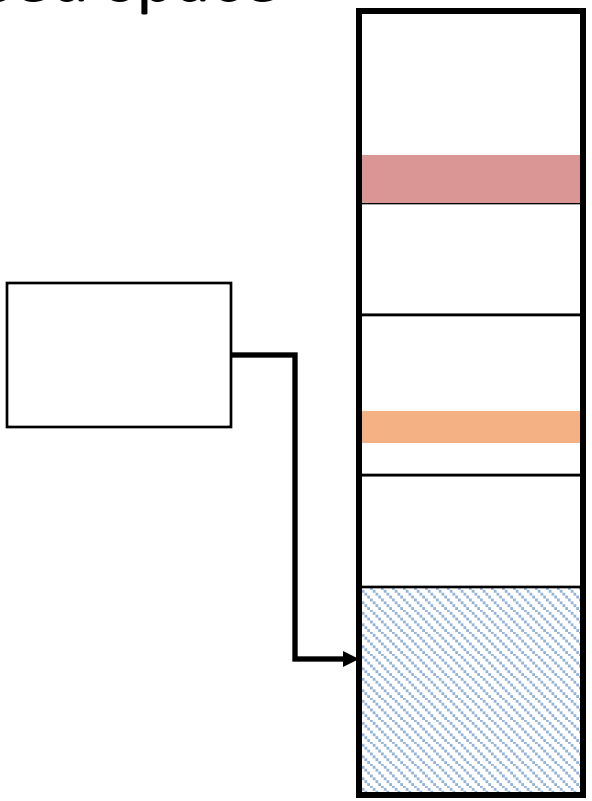
What if it doesn't fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches



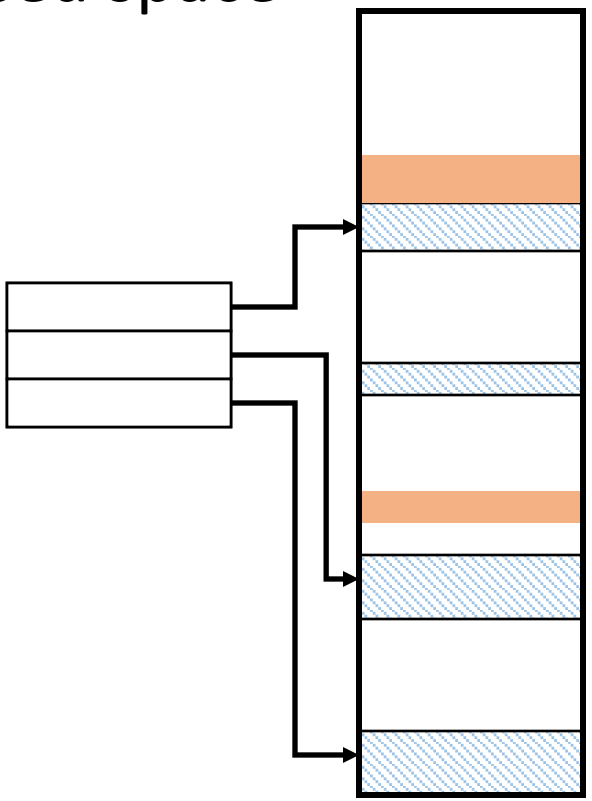
What if it doesn't fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - *Compaction*



What if it doesn't fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - Compaction
  - *Break process memory into pieces*
    - Easier to fit.
    - More state to keep track of.



Problem Summary: Placement

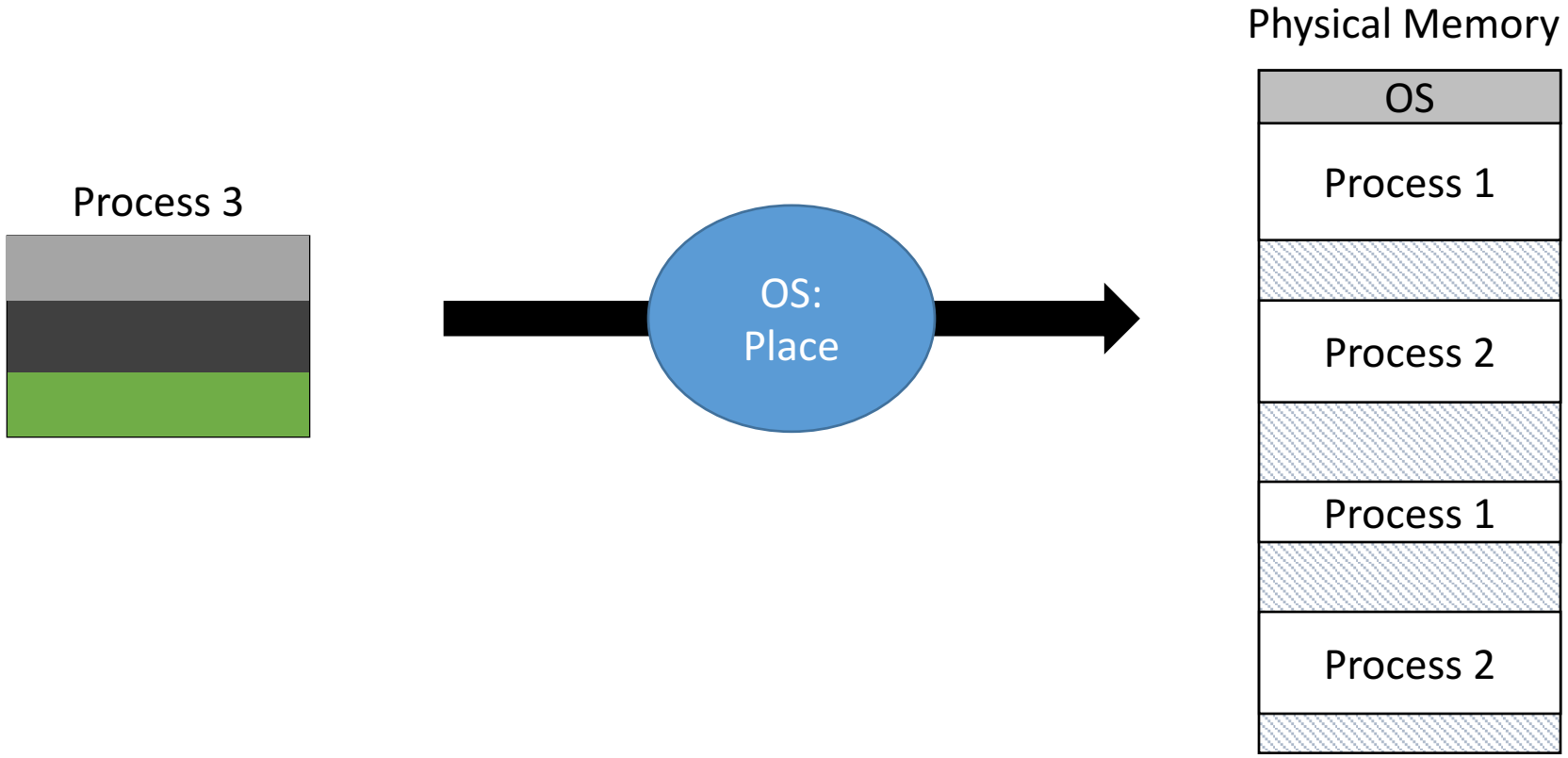
- When placing process, it may be hard to find a large enough free region in physical memory.
- Fragmentation makes this harder over time (free pieces get smaller, spread out).
- General solution: don’t require all of a process’s memory to be in one piece!

Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!

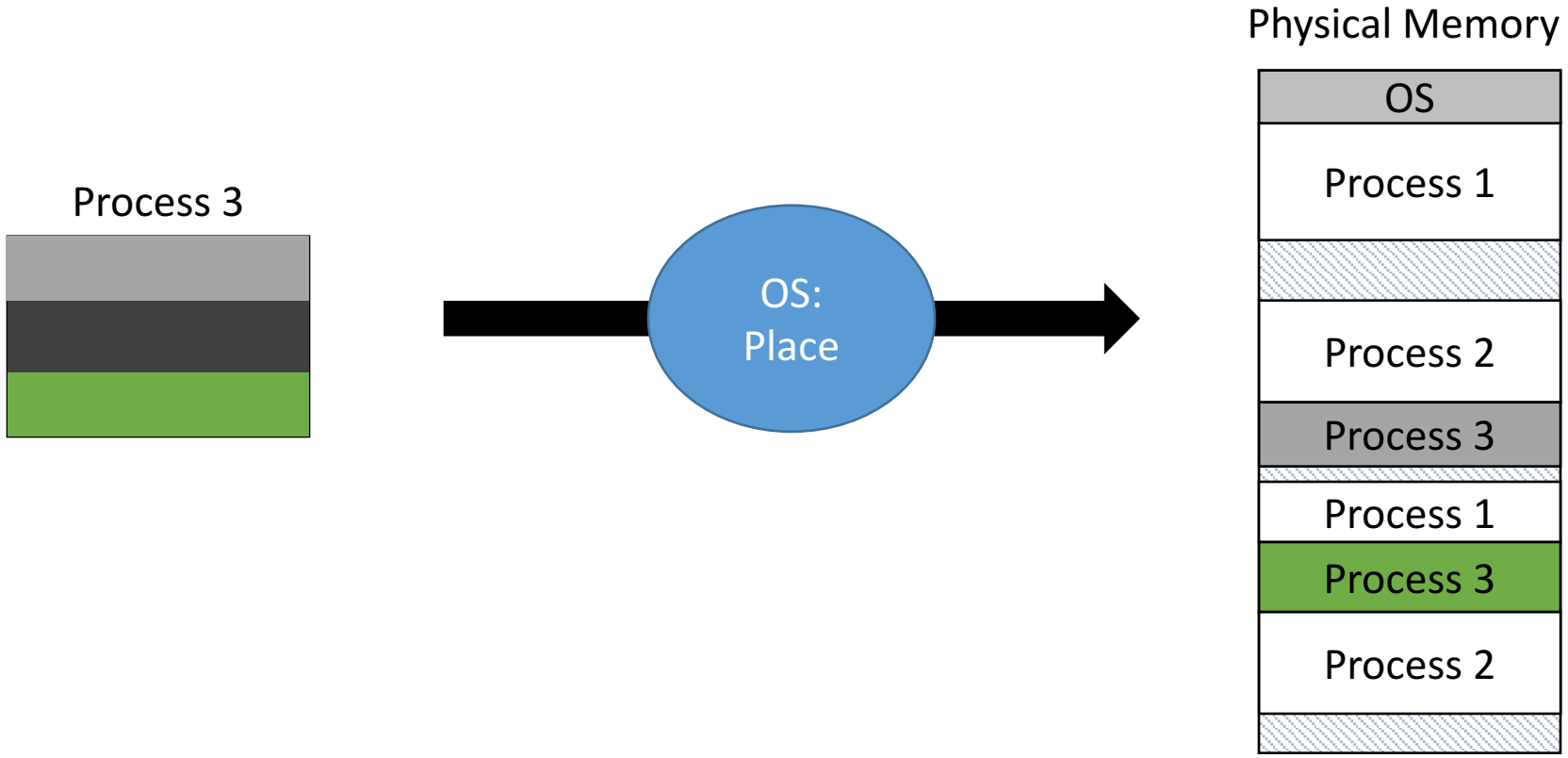
Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!



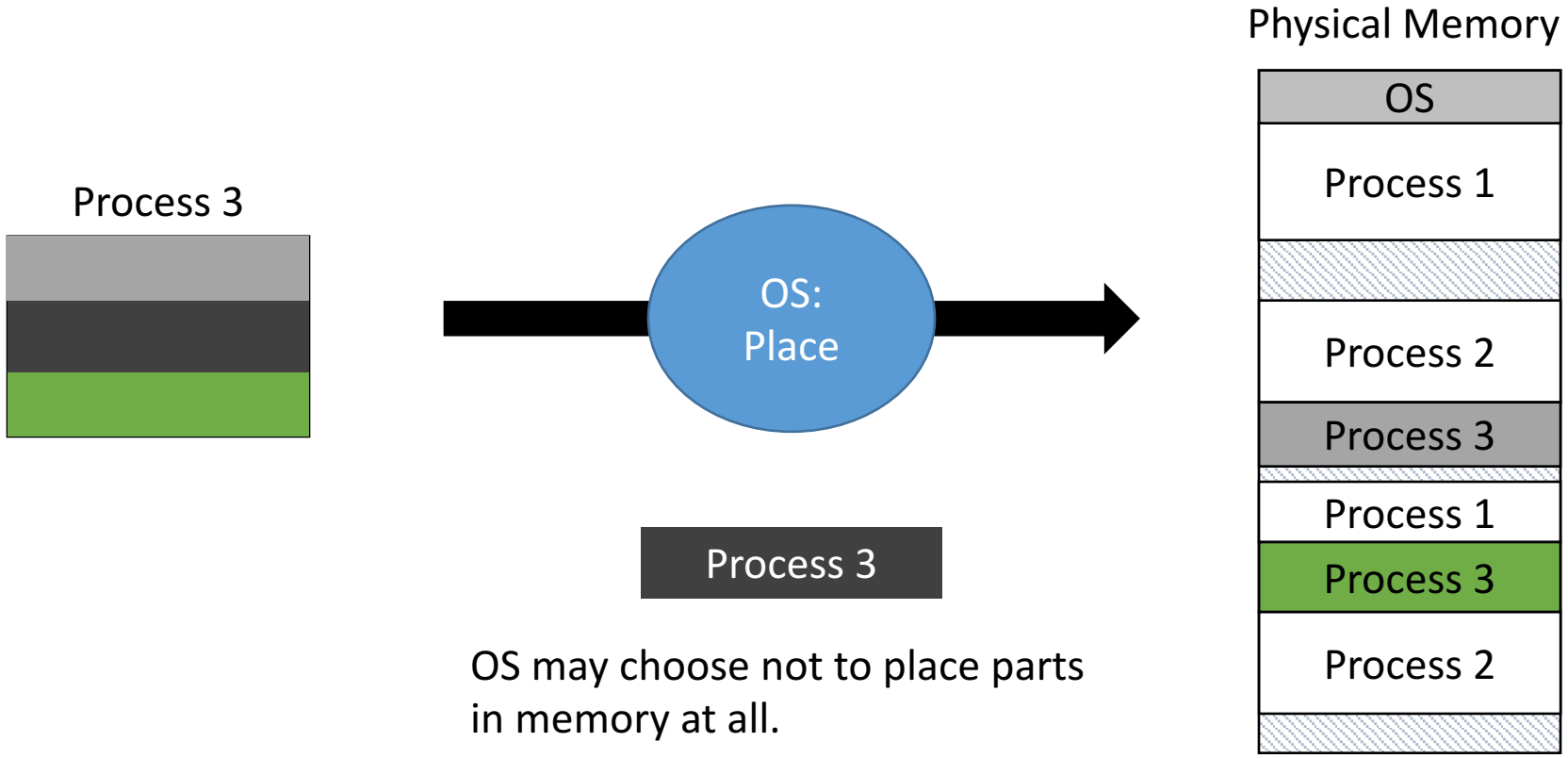
Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!



Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!

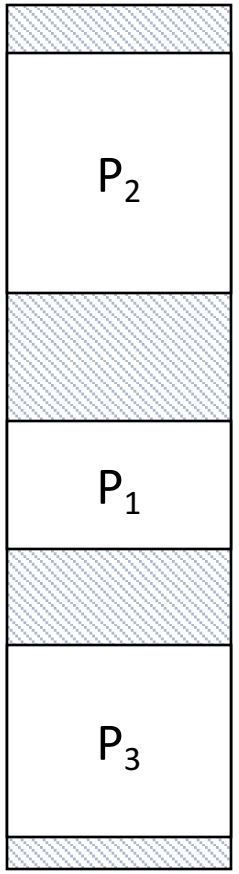


Problem: Addressing

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

(More) Problems with Memory Cohabitation

- Addressing:
  - Compiler generates memory references
  - Unknown where process will be located
- Protection:
  - Modifying another process's memory
- Space:
  - The more processes there are, the less memory each individually can have



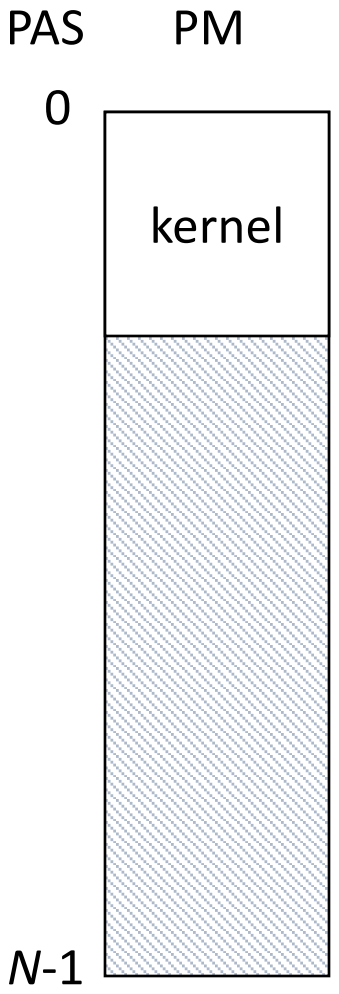
Compiler's View of Memory

- Compiler generates memory addresses
  - Needs empty region for text, data, heap, stack
  - Ideally, very large to allow heap and stack to grow
  - Alternative: four independent empty regions

- What compiler needs to know, but doesn't
  - Physical memory size
  - Where to place data (e.g., stack at high end)
    - Must avoid allocated regions in memory

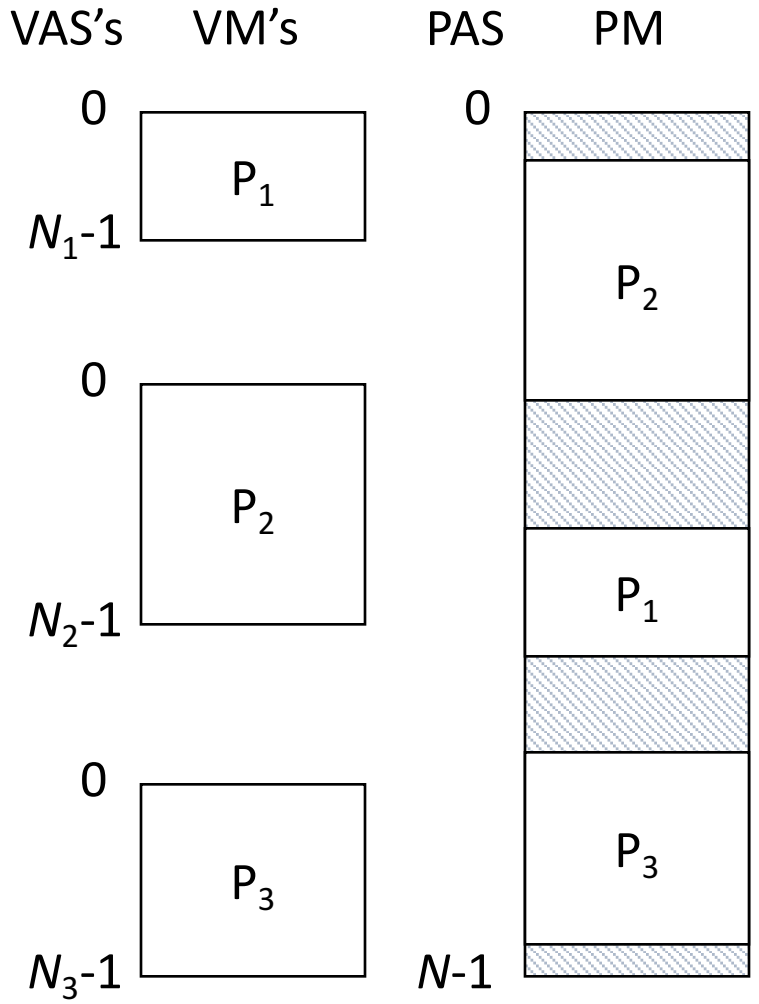
Address Spaces

- Address space
  - Set of addresses for memory
- Usually linear: 0 to *N*-1 (size *N*)
- Physical Address Space
  - 0 to *N*-1, *N* = size
  - Kernel occupies lowest addresses



Virtual vs. Physical Addressing

- Virtual addresses
  - Assumes separate memory starting at 0
  - Compiler generated
  - Independent of location in physical memory
- OS: Map virtual to physical



When should we perform the mapping from virtual to physical address? Why?

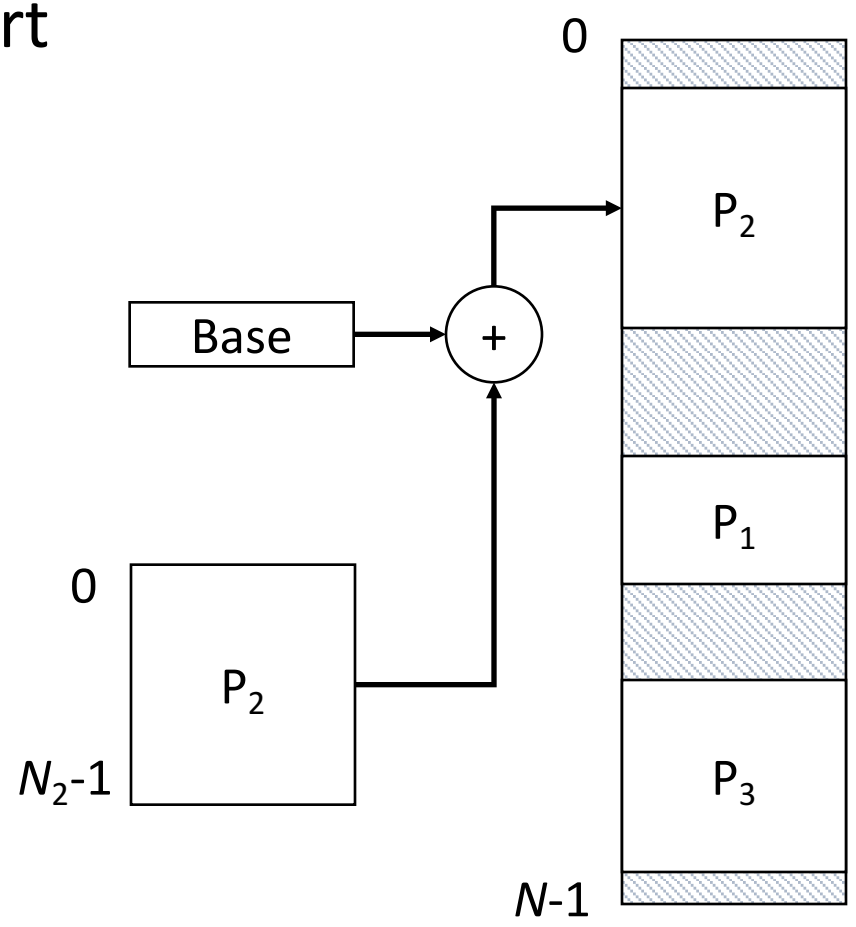
A. When the process is initially loaded: convert all the addresses to physical addresses

B. When the process is running: map the addresses as they're used.

C. Perform the mapping at some other time. When?

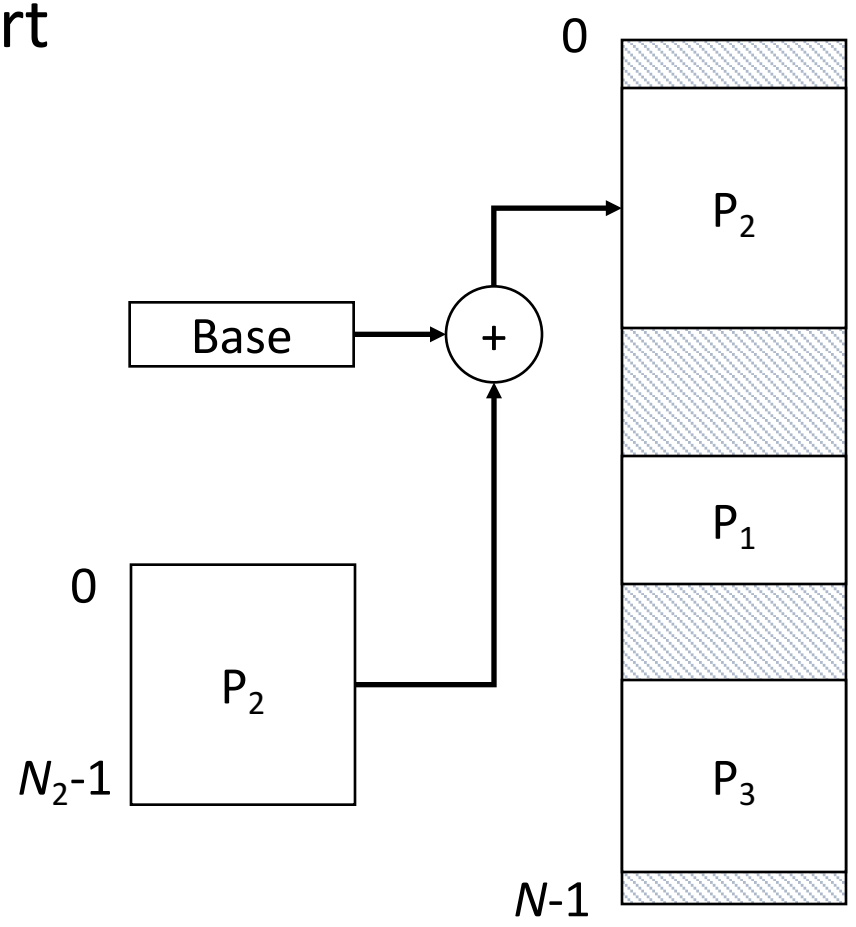
Hardware for Virtual Addressing

- Base register filled with start address
- To translate address, add base
- Achieves relocation
- To move process: change base



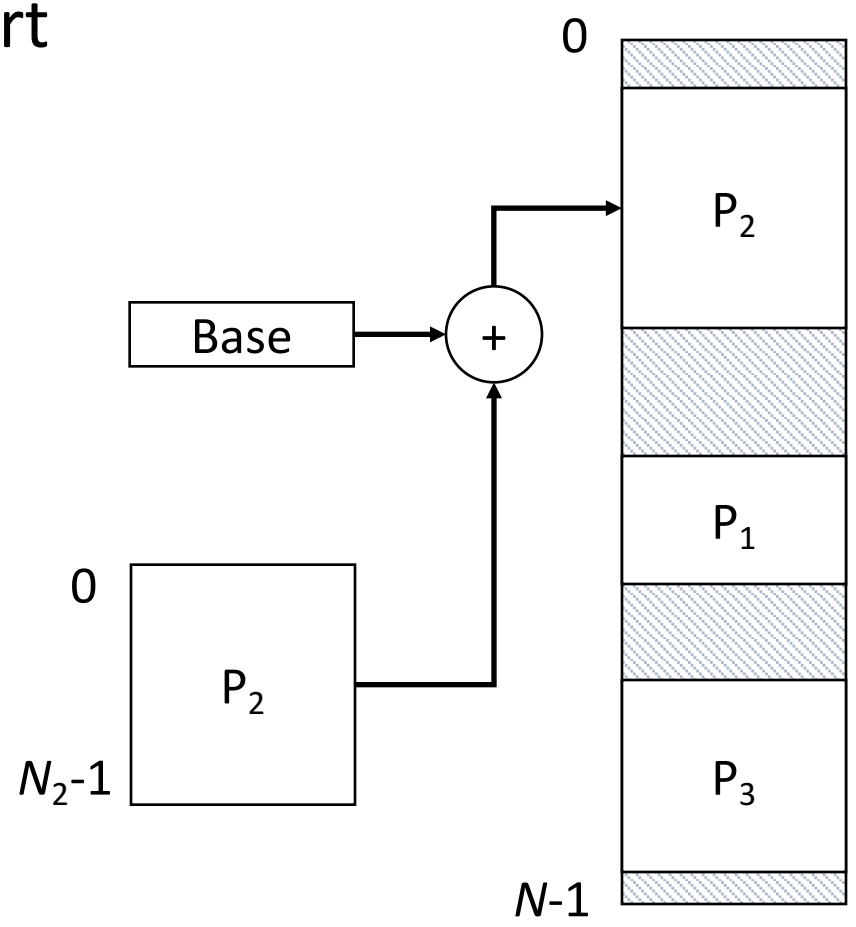
Hardware for Virtual Addressing

- Base register filled with start address
- To translate address, add base
- Achieves relocation
- To move process: change base



Hardware for Virtual Addressing

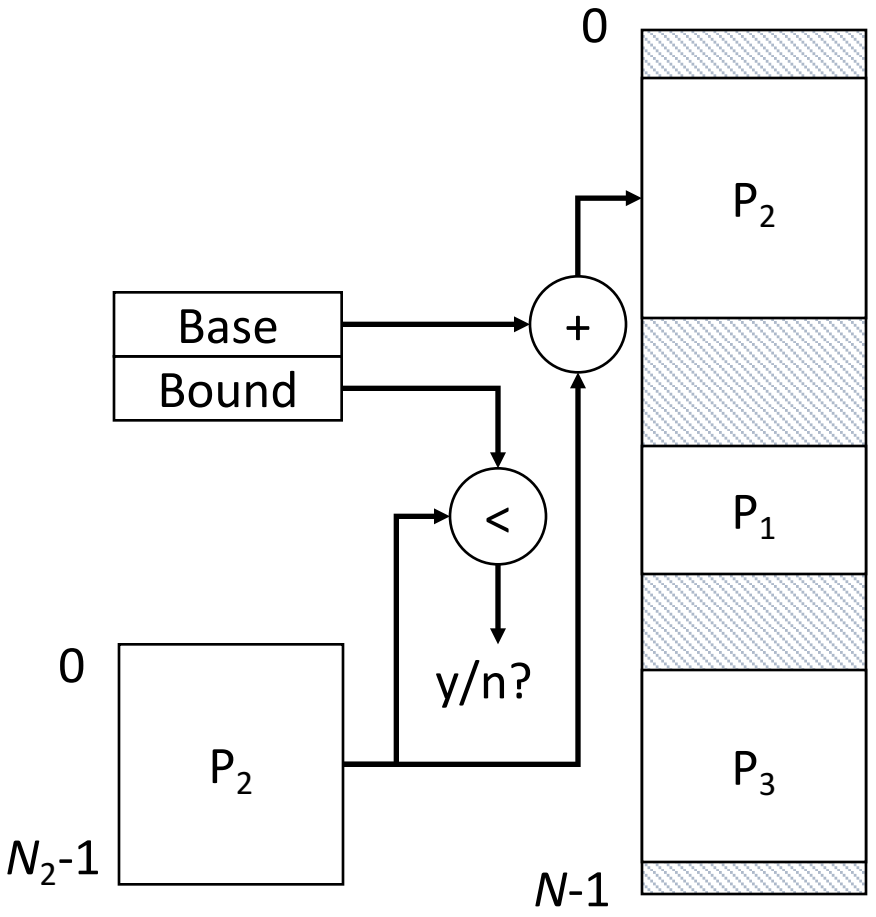
- Base register filled with start address
- To translate address, add base
- Achieves relocation
- To move process: change base
- Protection?



Protection

- Bound register works with base register
- Is address < bound
  - Yes: add to base
  - No: invalid address, TRAP
- Achieves protection

When would we need to update these base & bound registers?



Given what we currently know about memory, what must we do during a context switch?

- A. Allocate memory to the switching process
- B. Load the base and bound registers
- C. Convert logical to physical memory addresses

Memory Registers Part of Context

- On Every Context Switch
  - Load base/bound registers for selected process
  - Only kernel does loading of these registers
  - Kernel must be protected from all processes
- Benefit
  - Allows each process to be separately located
  - Protects each process from all others

Problem Summary: Addressing

- Compiler has no idea where, in physical memory, the process's data will be.
- Compiler generates instructions to access VAS.
- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.

Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.

Process 3

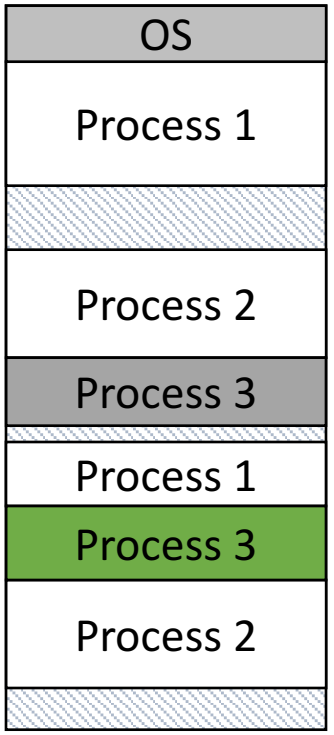
movl (address 0x74), %eax

When the process tries to access a virtual address, the OS translates it to the corresponding physical address.

OS: Translate

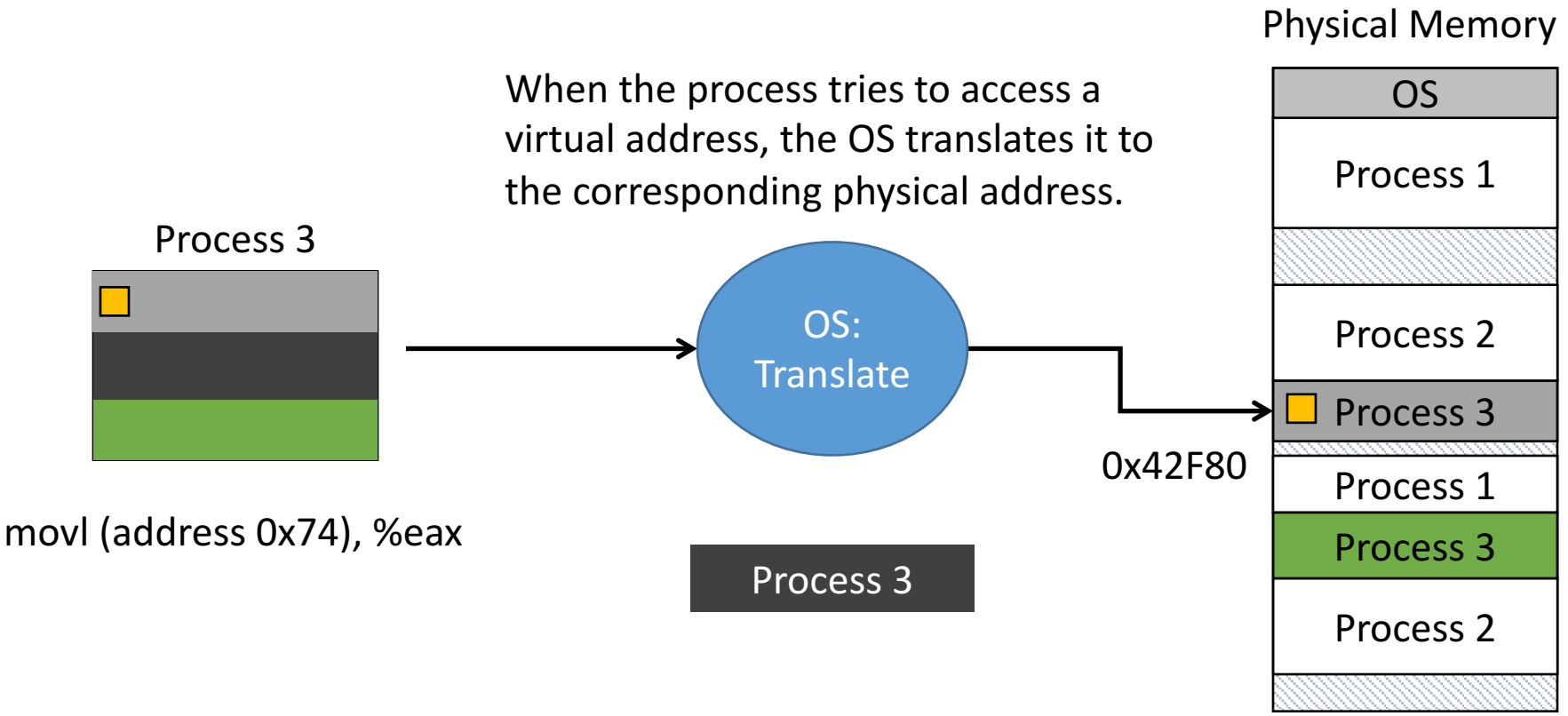
Process 3

Physical Memory



Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.



Let’s combine these ideas:

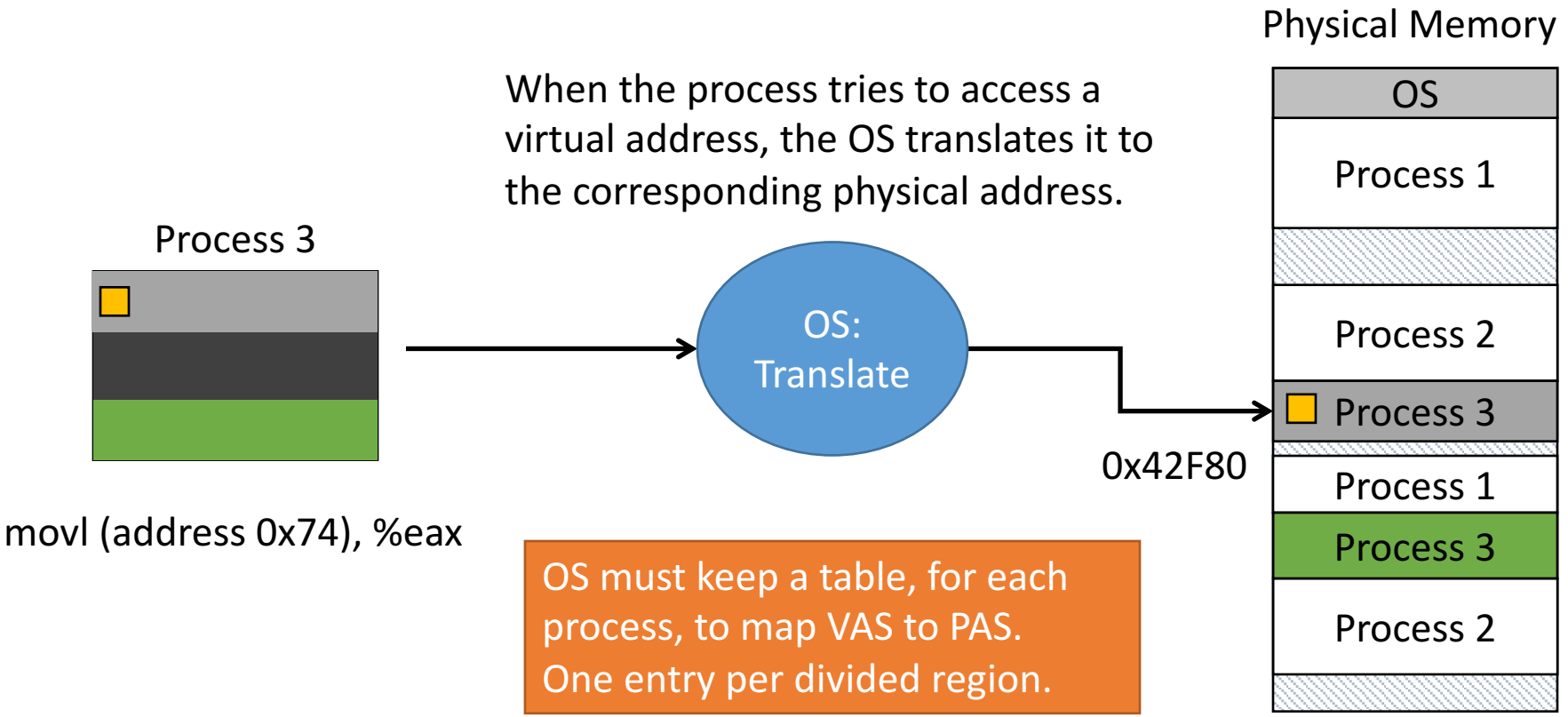
1. Allow process memory to be divided up into multiple pieces.

2. Keep state in OS (+ hardware/registers) to map from virtual addresses to physical addresses.

Result: Keep a table to store the mapping of each region.

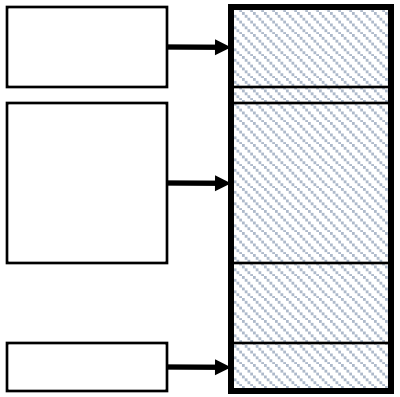
Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.



Two (Real) Approaches

- Segmented address space/memory
- Partition address space and memory into segments
- Segments are generally different sizes



- Paged address space/memory
- Partition address space and memory into pages
- All pages are the same size