Memory Management

Kevin Webb

Swarthmore College

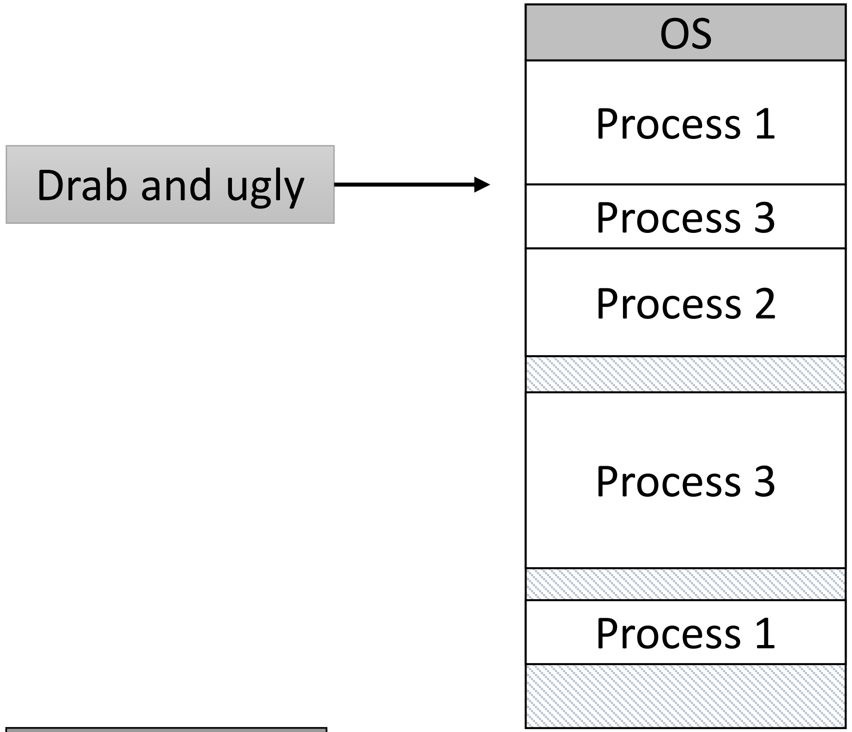
February 22, 2024

Today's Goals

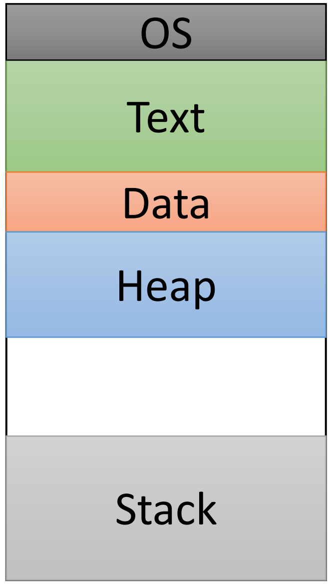
- Shifting topics: different process resource – memory
- Motivate virtual memory, including what it might look like without it
- How different views of memory affect stakeholders (user, programmer, OS, compiler, hardware)
- Big picture: the components and how they fit together.
  Later: implementation details.

Memory

- Reality: there's only so much memory to go around, and no two processes should use the same (physical) memory addresses.



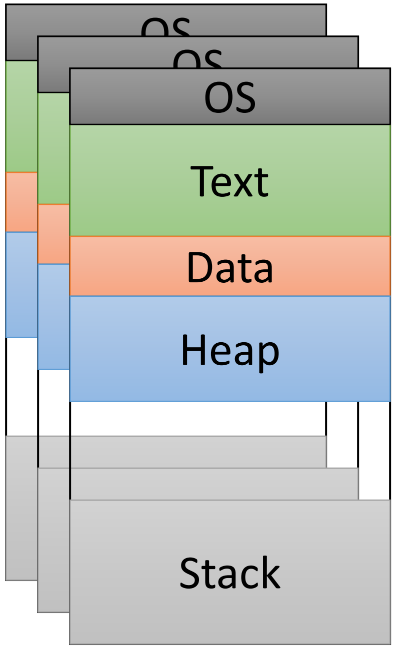
- Abstraction goal: make every process think it has the same memory layout.



Memory Terminology

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

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



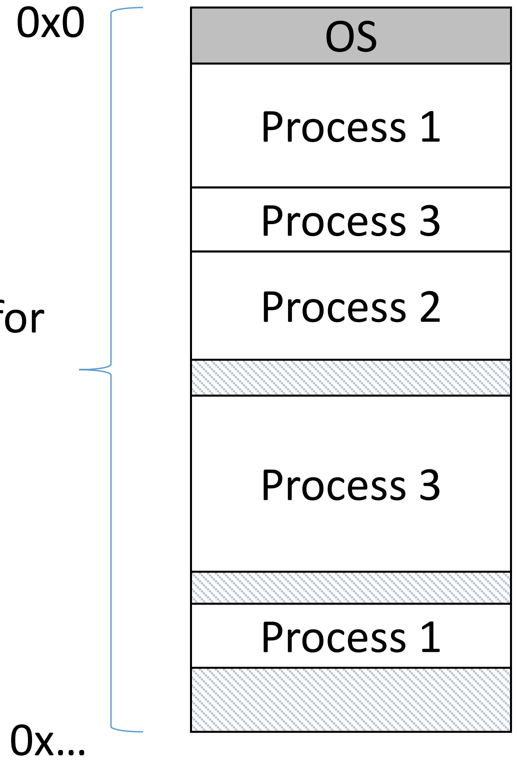
0x0

<u>Address Space</u>: Range of addresses for a region of memory.

The set of available storage locations.

0xFFFFFFFF

Virtual address space (VAS): fixed size (CPU).



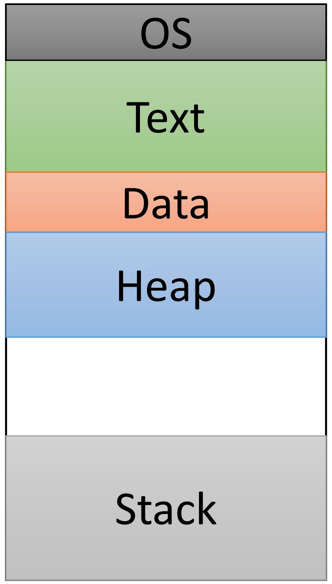
(Determined by HW and amount of installed RAM.)

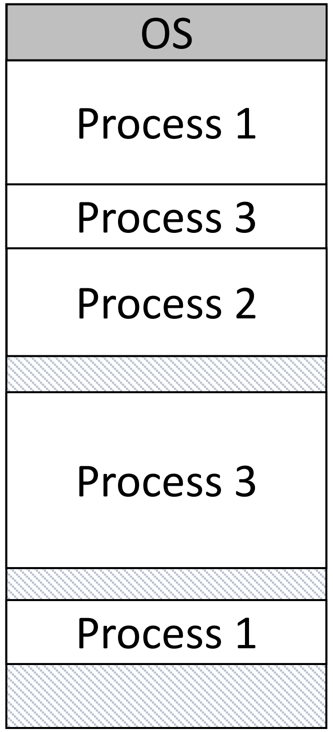
VAS vs. PAS Sizes

- Example 1: 32-bit x86: VAS < PAS

32-bit virtual addresses.

=> 4GB VAS





36-bit physical addresses (with PAE turned on).

=> 64 GB PAS

VAS vs. PAS Sizes

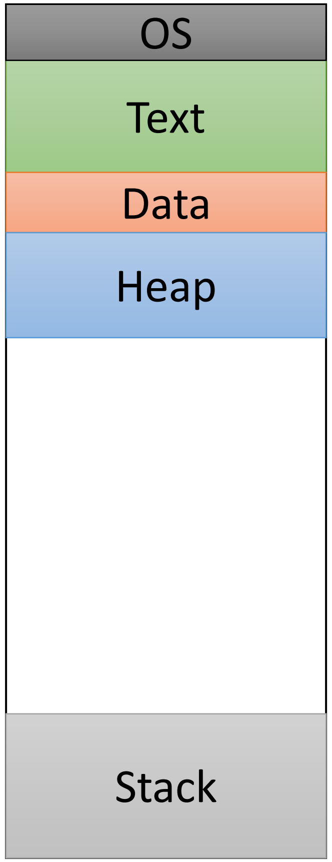
Implication: the user can ask for more memory (and assume it's available) than the system can physically support.

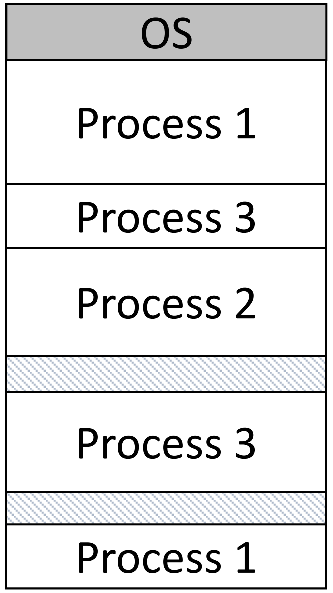
Uh-oh?

- Example 2: 64-bit x86: VAS >> PAS

48-bit virtual addresses.

=> 256 TB VAS





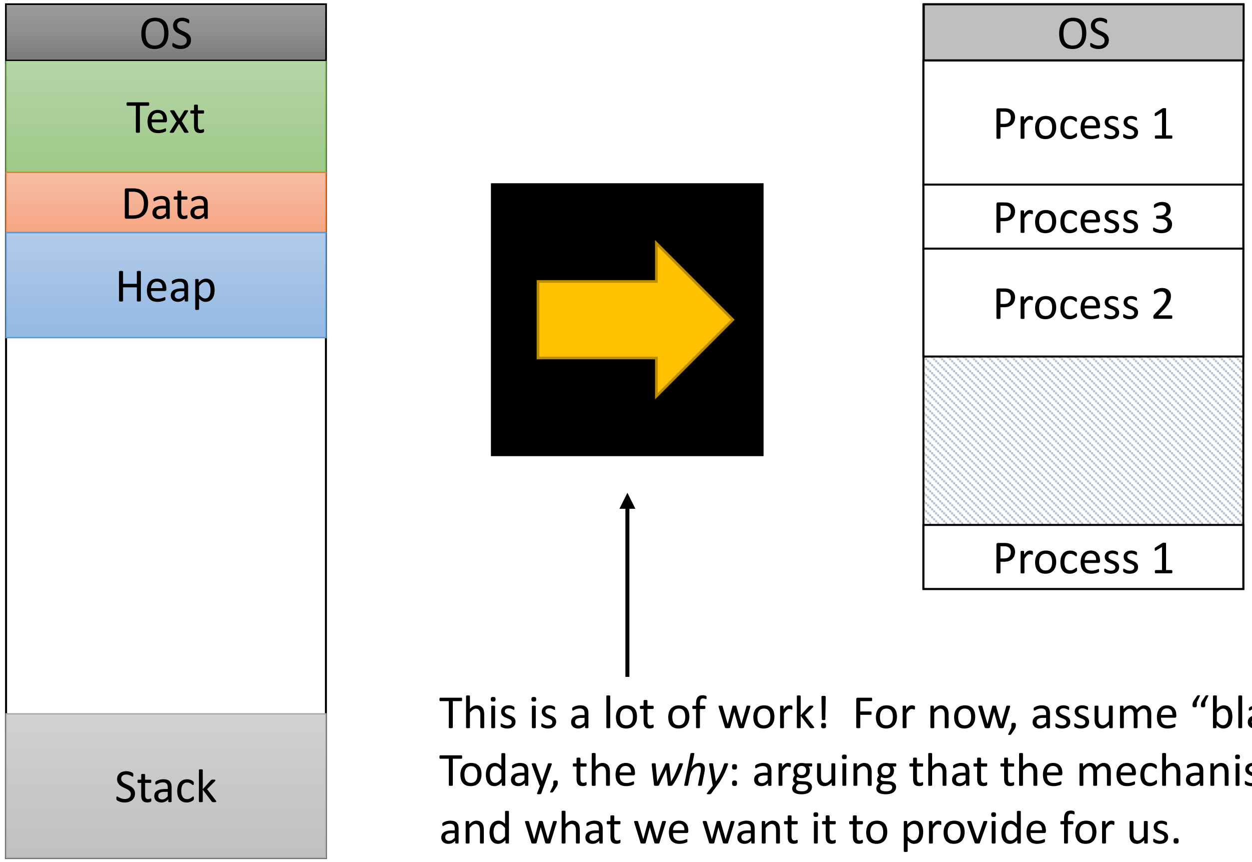
36-bit physical addresses.

=> 64 GB PAS

(These values come from our lab machines. The architecture itself allows for 64-bits, but most hardware doesn't go nearly that far => 16,777,216 TB)

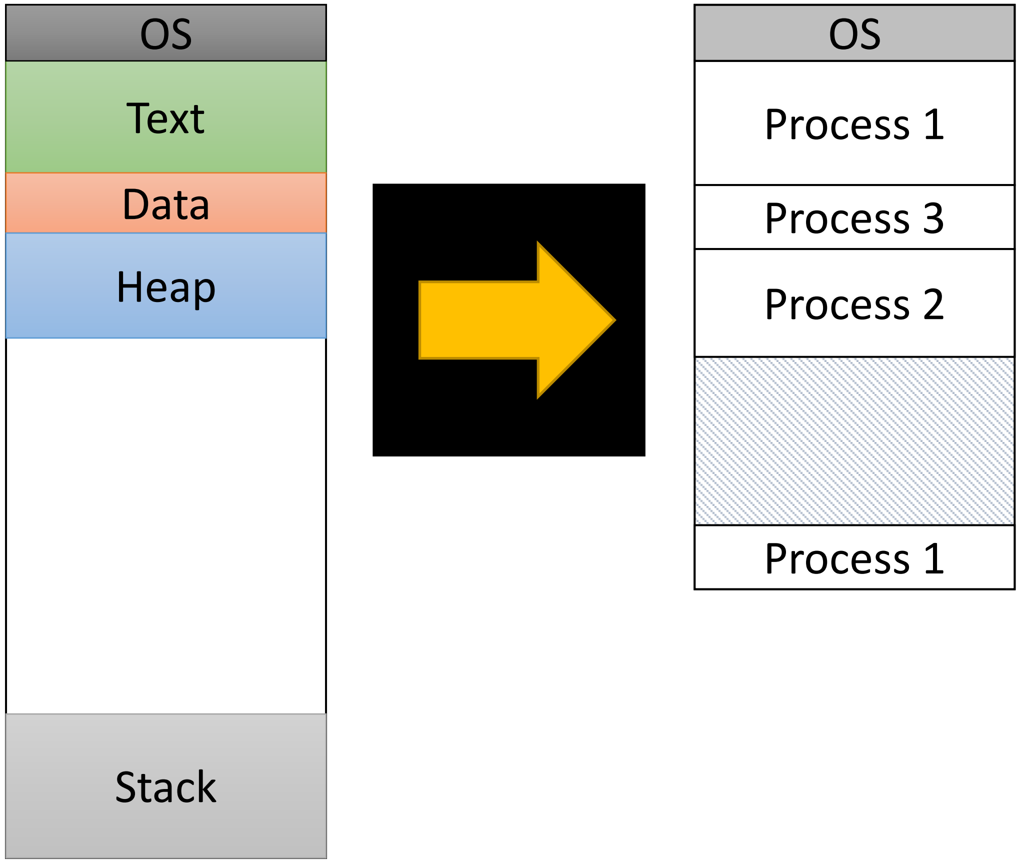
Address Translation

- Virtual addresses must be *translated* to physical addresses.



This is a lot of work! For now, assume "black box" mechanism does it.
Today, the *why*: arguing that the mechanism is worth it / necessary and what we want it to provide for us.

Address Translation: Wish List



- Map virtual addresses to physical addresses.

Who benefits most from having a logical memory abstraction? Why?

A. The user

B. The programmer

C. The compiler

D. The OS / OS designer

E. The hardware / hardware designer

User Perspective

- Average user doesn't care about "address spaces" or memory sizes

- User might say:
  - I want all my programs to be able to run at the same time.
  - I don't want to worry about running out of memory.

- If OS does nothing / has no virtual memory:
  - Best we can do is give them all of the physical memory.
  - Is that enough? Recall that VAS size can be larger than PAS…

Let's explore what the OS might be able to do to help.

Multiprogramming, Revisited

- Recall multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.
    - For CPU resource: context switch quickly between processes

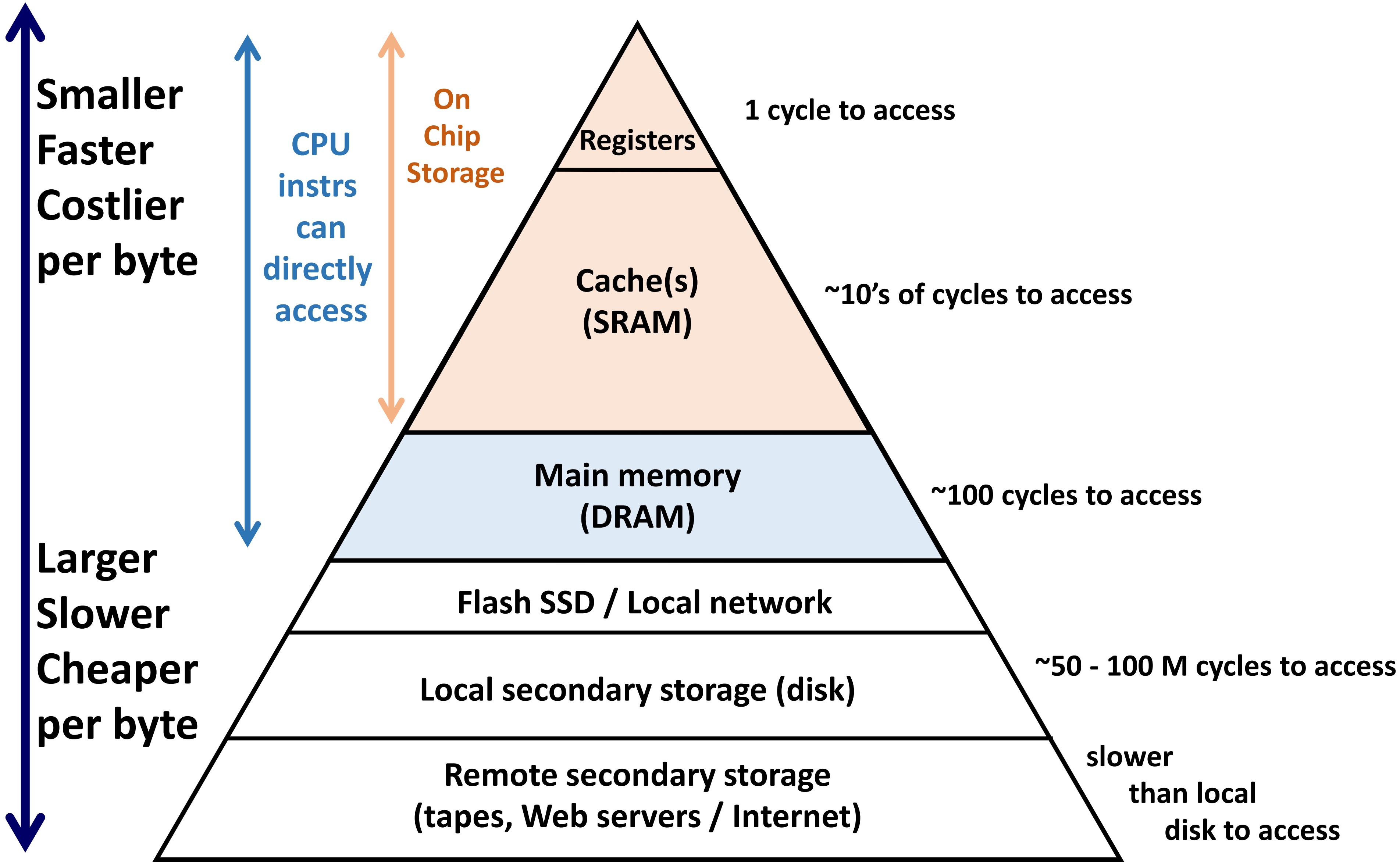
Multiprogramming, Revisited

- Recall multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.
  - For CPU resource: context switch quickly between processes

- Can we perform something analogous to a context switch for process memory?

A. Yes (how? Where will process memory be stored?)
B. No (why not?)
C. It depends (on what?)

Recall: The Memory Hierarchy



Multiprogramming, Revisited

- Recall multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.
  - For CPU resource: context switch quickly between processes

- Can we perform something analogous to a context switch for process memory?
  - Suppose disk transfer rate is 100 MB/s
  - "switching" a 1 MB process would take 10 ms (+ disk seek time)
  - CPU context switch: approx. 10 – 50 µs
  - Moving that 1 MB would make context switch take 200 – 1000 times longer!

Conclusion: We can't swap entirety of process memory on a context switch. It needs to already be in memory.

Using Disk

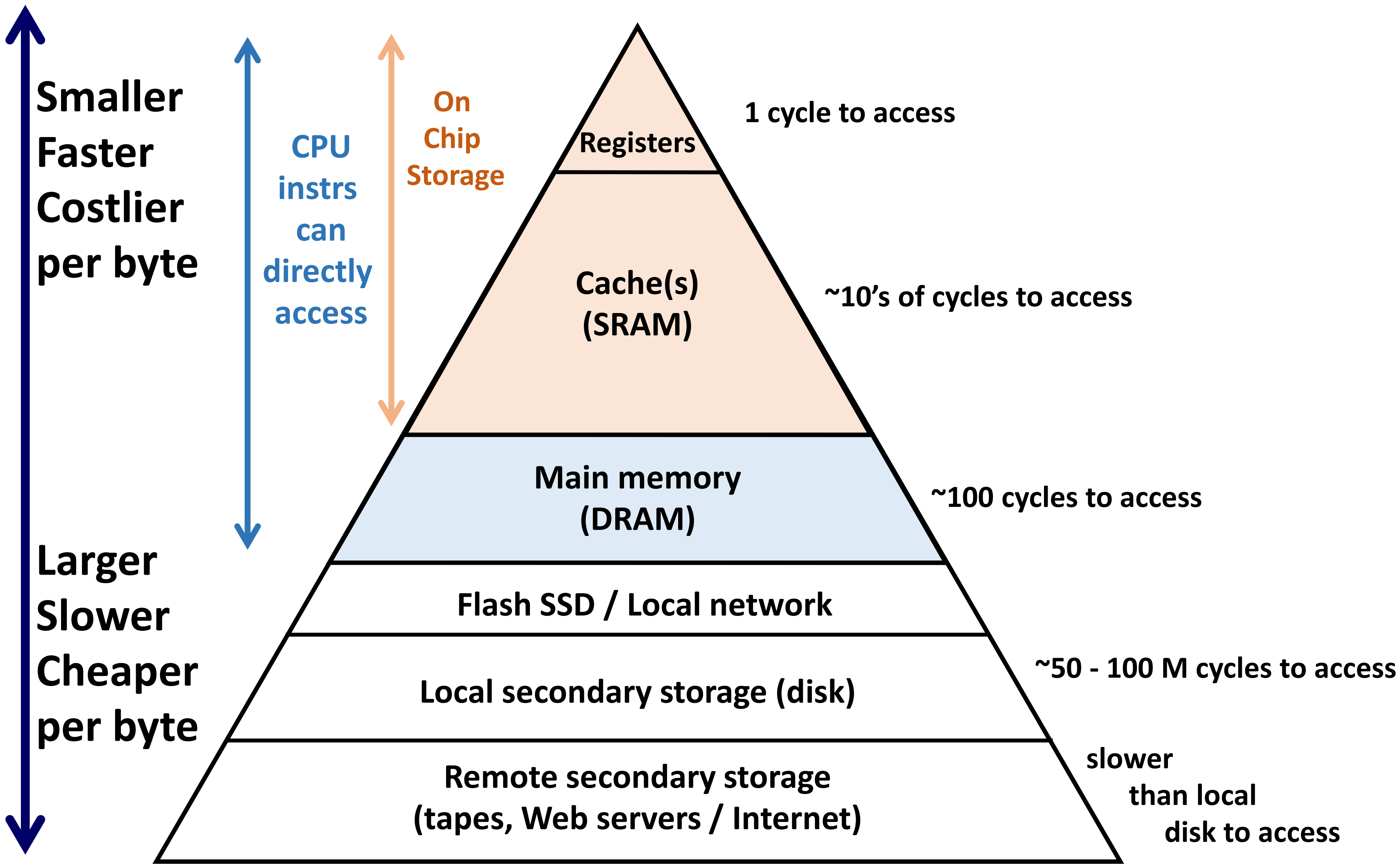
- We still have a large amount of disk space though!

- If the total size of desired memory is larger than PAS, overflow to disk.
  - Disk: can store a lot, but relatively painful to access
  - Memory: much faster than disk, but can only store a subset

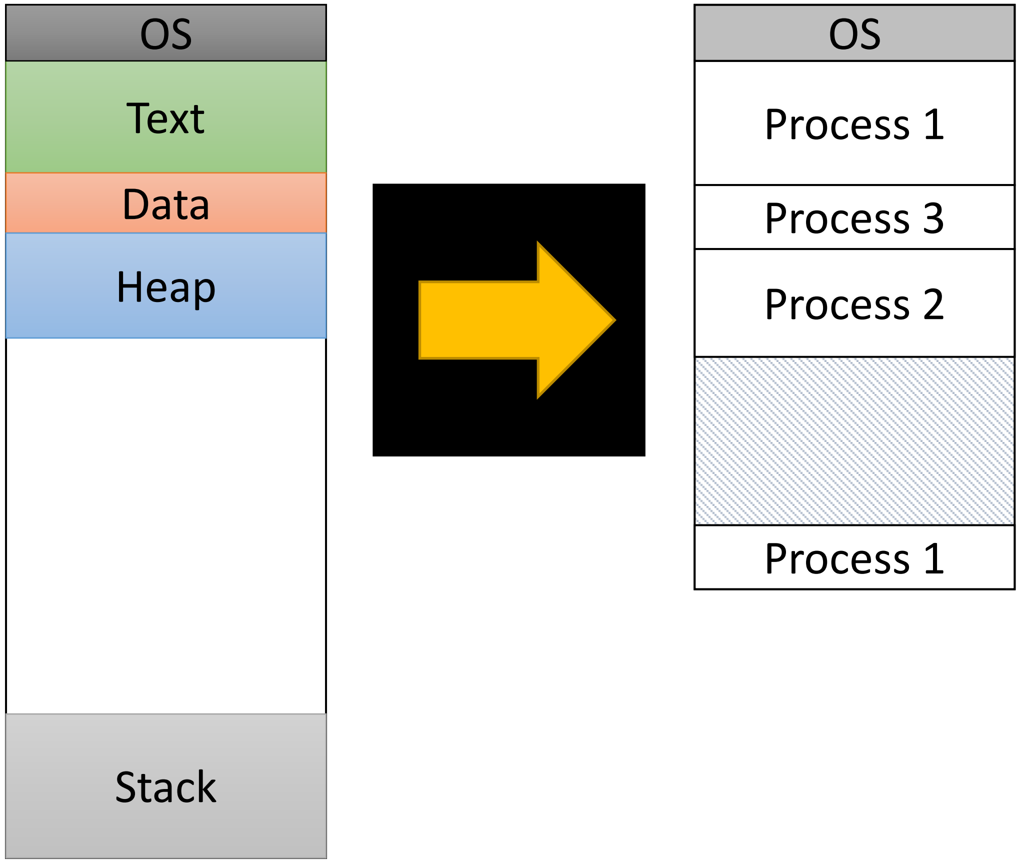
- This should sound familiar to a big CS 31 topic… Caching

- Recall locality: we tend to repeatedly access recently accessed items, or those that are nearby.

Recall: The Memory Hierarchy



Address Translation: Wish List

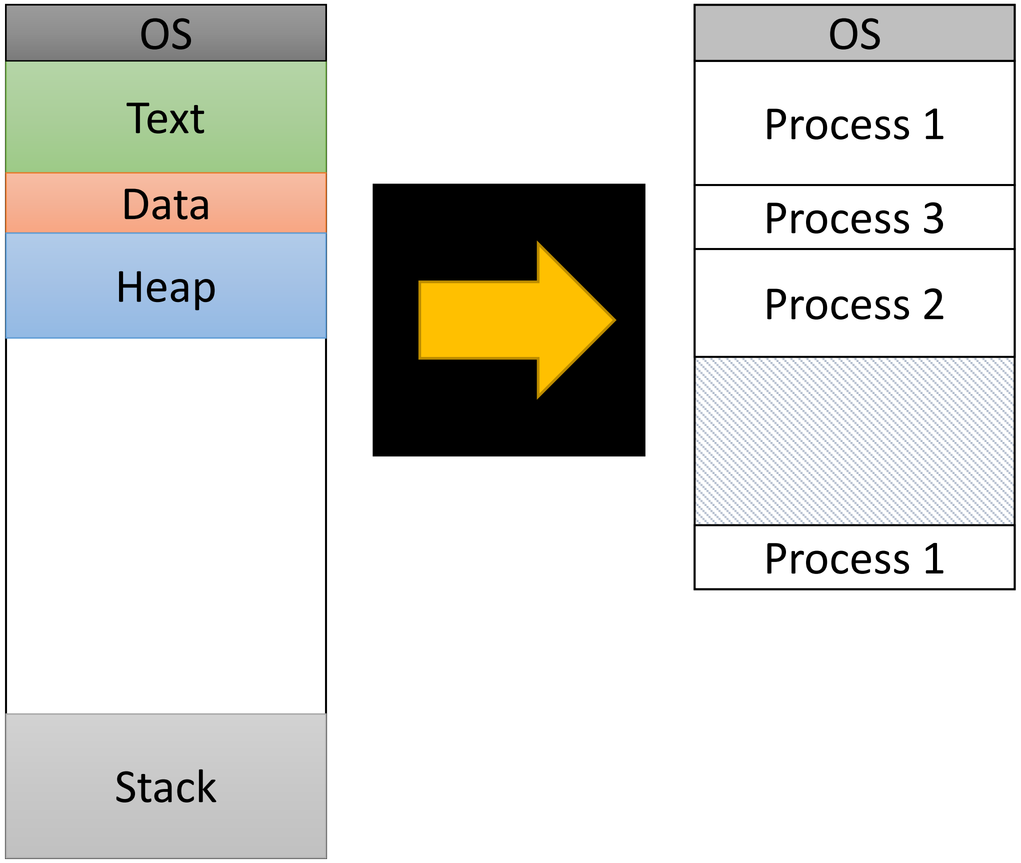


- Map virtual addresses to physical addresses.
- Determine which subset of data to keep in memory / move to disk.

Protection

- Another thing users want/expect, even if they don't realize it…
- Reality: Multiple processes *will* be in memory at the same time.
- Processes should *not* be able to read/write each other’s memory (unless we approve them to, with shared memory)

Address Translation: Wish List



- Map virtual addresses to physical addresses.
- Determine which subset of data to keep in memory / move to disk.
- Allow multiple processes to be in memory at once, but isolate them from each other.

Programmer Perspective

- Mix of user and complier needs.
  - High-level language: probably care more about memory availability
  - Low-level language: probably care a lot about memory addresses

- One major concern: library code
  - I want to #include lots of functionality for free!

If multiple processes want to use the same library, how should we support that? Why?

A. Add a copy of the library code to the executable file at compile time.

B. Load a copy of the library code into memory when the process begins executing.

C. Map a shared copy of the library code in each process's virtual address space.

Linking

- Static Linking: bundle up one giant executable, with copies of all library code.
  - Advantage: fully self-contained, not dependent on system libraries (portable)
  - Disadvantage: makes executable take up lots of space (on disk and in memory)

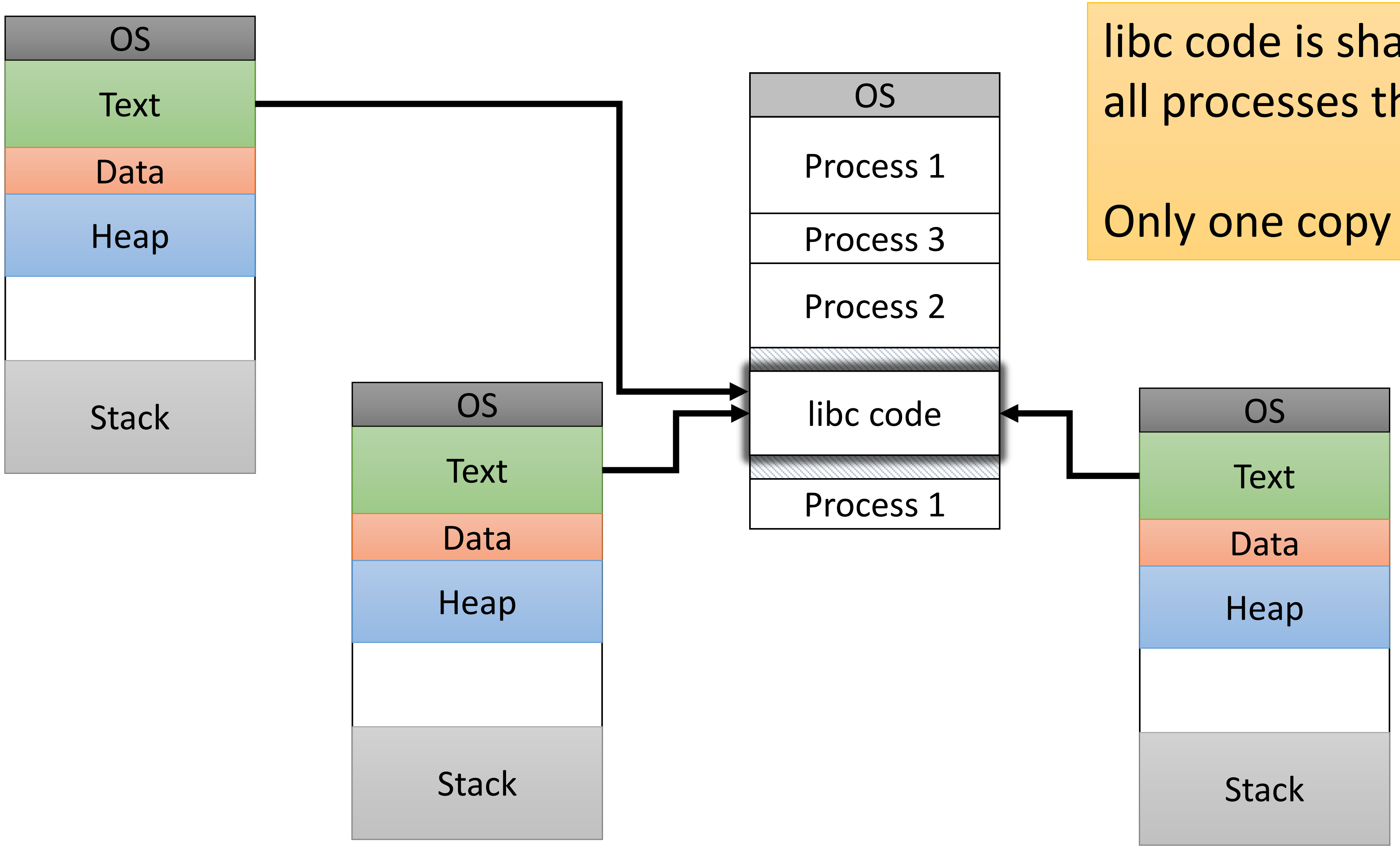
- Dynamic Linking: executable refers to external library code, which must be installed on system (or runtime error)
  - Advantage: memory efficiency, only one copy of library code needed
  - Disadvantage: must have library installed on system to use it

Dynamic Libraries

- On Linux: .so (shared object) file
- On Window: .dll (dynamically linked library) file

- Example: C standard library (libc)
  - Every process can use the same libc code (printf, malloc, strlen, etc.)

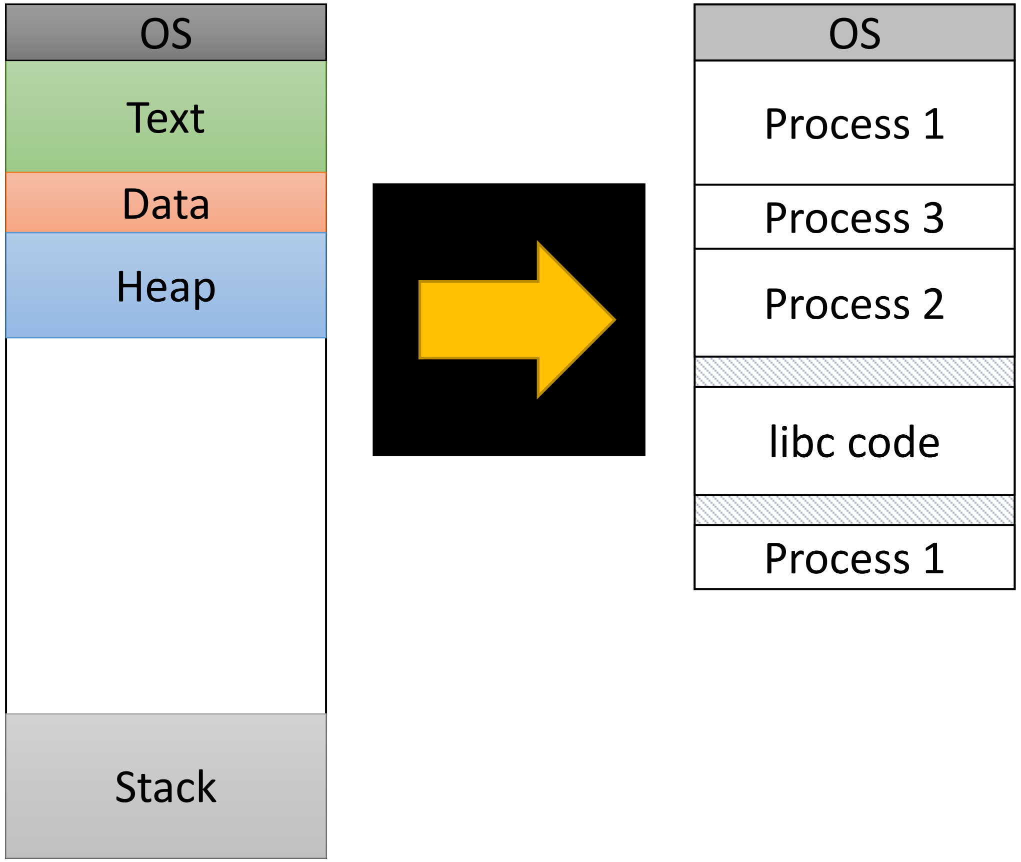
Dynamic Library in Memory



libc code is shared (read only) by all processes that need it.

Only one copy need be in memory!

Address Translation: Wish List



- Map virtual addresses to physical addresses.
- Determine which subset of data to keep in memory / move to disk.
- Allow multiple processes to be in memory at once, but isolate them from each other.
- Allow the same physical memory to be mapped in multiple process VASes.

Compiler Perspective

- Compiler's goal: generate assembly code that will run… *later*.

- It generates the instructions for code and puts them somewhere in the resulting executable.

Changing the Program Counter

- Recall: PC register contains address of next instruction

- The compiler must change the PC when program control flow needs it
  - if / else: skip over some section of code (jump over instructions)
  - loops: keep repeating the same code (jump back to same instructions)
  - function call: execute code at some other location, come back later

- All of these cases: compiler must be setting the PC to *some* value

Placing and Finding Code *This is simplified a lot.

Suppose we're generating code for two functions: f1() and f2(), and f1 calls f2.

Option A: Choose addresses

f1:	0x1000	add %eax, %ecx
		…
	0x100C	call f2 (jump to 0x104C)
		…
f2:	0x104C	movl (%edx), %eax
		…
		ret

Option B: Use relative addresses

f1:	BASE	add %eax, %ecx
		…
	BASE + 0x0C	call f2 (jump forward 0x40)
		…
f2:	BASE + 0x4C	movl (%edx), %eax
		…
		ret

Placing and Finding Code

*This is simplified a lot.

Now suppose we're generating a function that makes a library call.

Option A: Choose addresses

```
f1:      0x1000   add %eax, %ecx
                  …
         0x100C   call lib_f (jump to 0x0xF460)
                  …
    Elsewhere in memory…
lib_f:   0xF460   movl (%edx), %eax
                  …
                  ret
```

Option B: Use relative addresses

```
f1:      BASE            add %eax, %ecx
                         …
         BASE + 0x0C     movl (load LIB_BASE)
         BASE + 0x10     call f2 (jump to loaded LIB_BASE)
                         …
    Elsewhere in memory…
lib_f:   LIB_BASE        movl (%edx), %eax
                         …
                         ret
```

Which would you use?
Why? How does it relate to OS / virtual memory?

Option A: Choose addresses

f1:	0x1000	add %eax, %ecx
		…
	0x100C	call lib_f (jump to 0x0xF460)
		…

Elsewhere in memory…

lib_f:	0xF460	movl (%edx), %eax
		…
		ret

Option B: Use relative addresses

f1:	BASE	add %eax, %ecx
		…
	BASE + 0x0C	movl (load LIB_BASE)
	BASE + 0x10	call f2 (jump to loaded LIB_BASE)
		…

Elsewhere in memory…

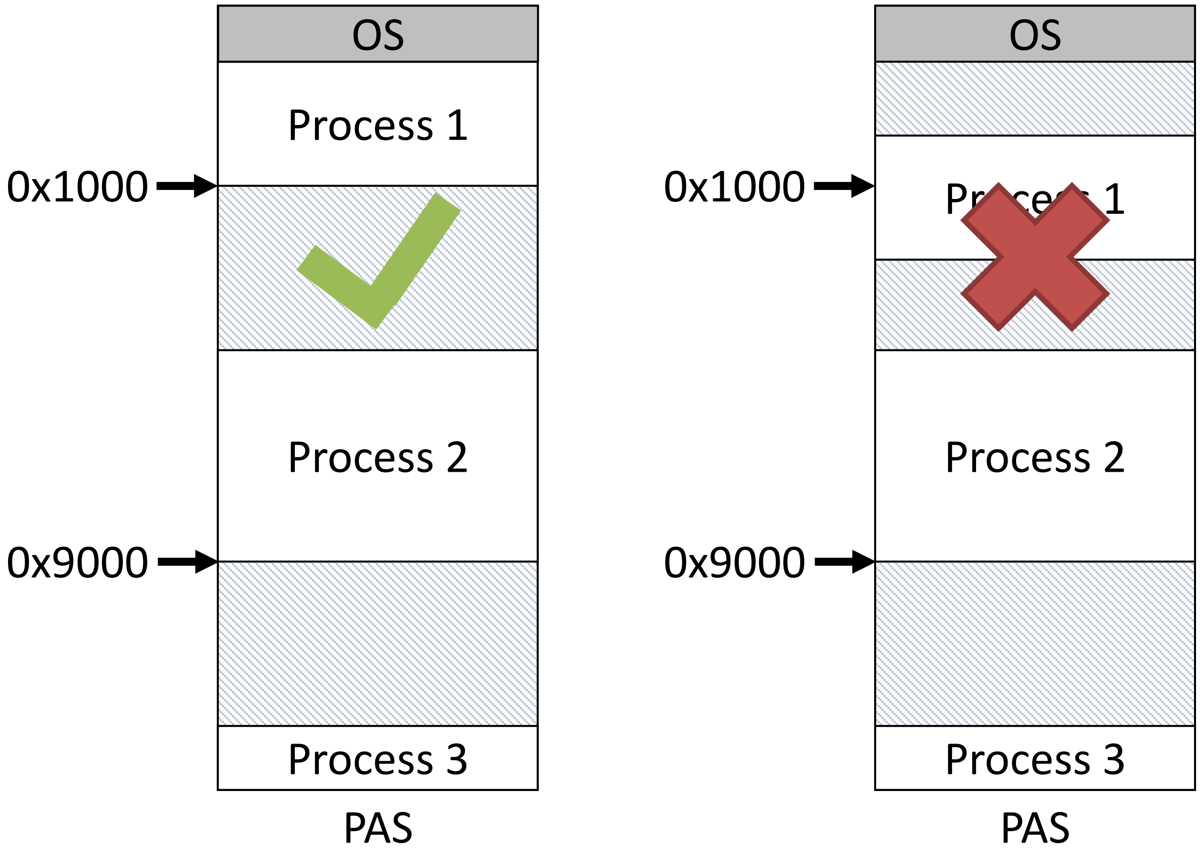
lib_f:	LIB_BASE	movl (%edx), %eax
		…
		ret

Without Help (Virtual Memory or Hardware)

- Without help from the OS/hardware, can't do B.

- Option A works…*sometimes.*

```
f1:   0x1000   add %eax, %ecx
               …
      0x100C   call f2 (jump to 0x1050)
               …
f2:   0x104C   movl (%edx), %eax
               …
               ret
```



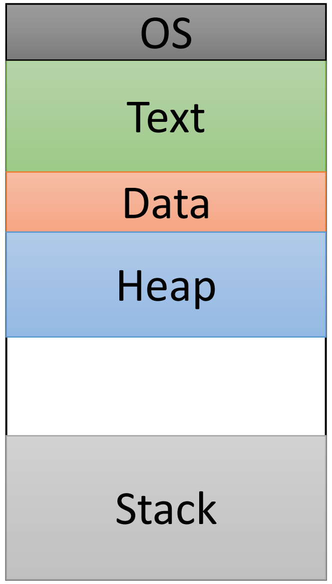
Challenge: Dynamic Environment

- Compiler can’t realistically know:
  - When will the code run?
  - Which machine(s) will the code run on?
  - How much memory will be available at the time?
  - Where in the address space will that memory be available?

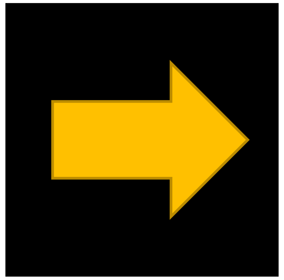
Conclusion: the compiler’s job is much easier if it can rely on the OS/Hardware to help with placement.

With Virtual Memory (OS and Hardware)

- Both options A and B work easily:
  - Compiler gets an abstract view of memory to use however it wants



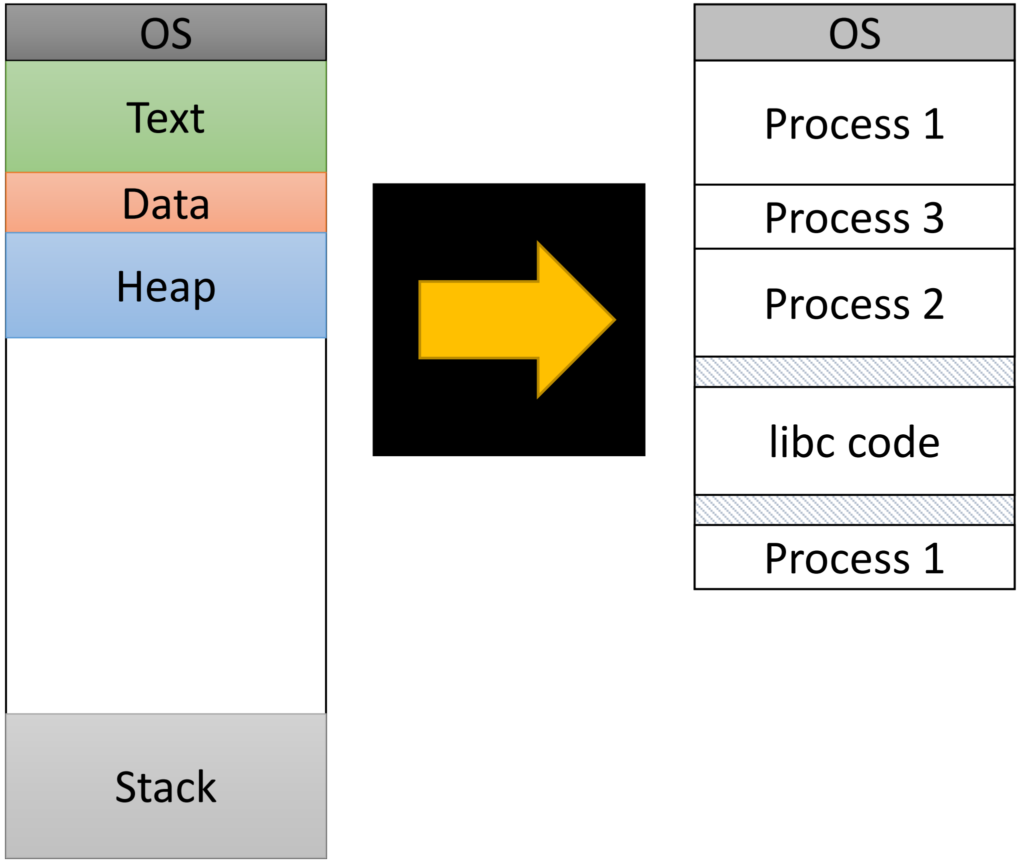
VAS



¯_(ツ)_/¯

*Don’t worry, the compiler still has a lot to worry about. Code generation is not easy…

Address Translation: Wish List



- **Map virtual addresses to physical addresses.**
- Determine which subset of data to keep in memory / move to disk.
- Allow multiple processes to be in memory at once, but isolate them from each other.
- Allow the same physical memory to be mapped in multiple process VASes.

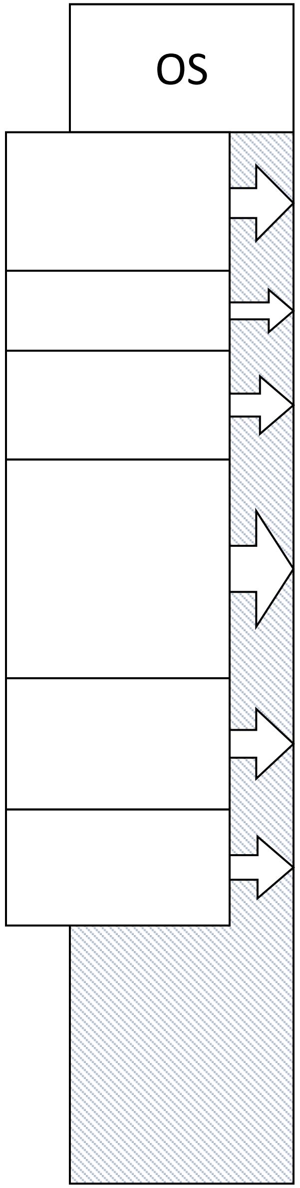
OS Perspective

- Primary challenge: Which physical memory do we give to processes?

- Other important considerations:
  - Protection: OS is resource gatekeeper, must isolate itself (and processes)
  - Performance: OS should map memory for best performance, as long as it doesn’t violate protection

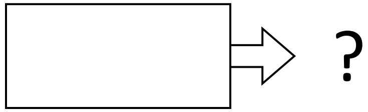
Without Virtual Memory Abstraction…

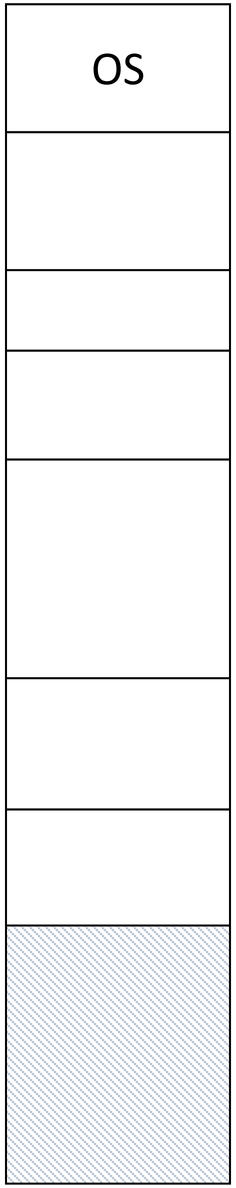
- Physical memory starts as one big empty space.
- When starting new processes, allocate memory.
  - At first, placement is easy: lots of large chunks free



Without Virtual Memory Abstraction…

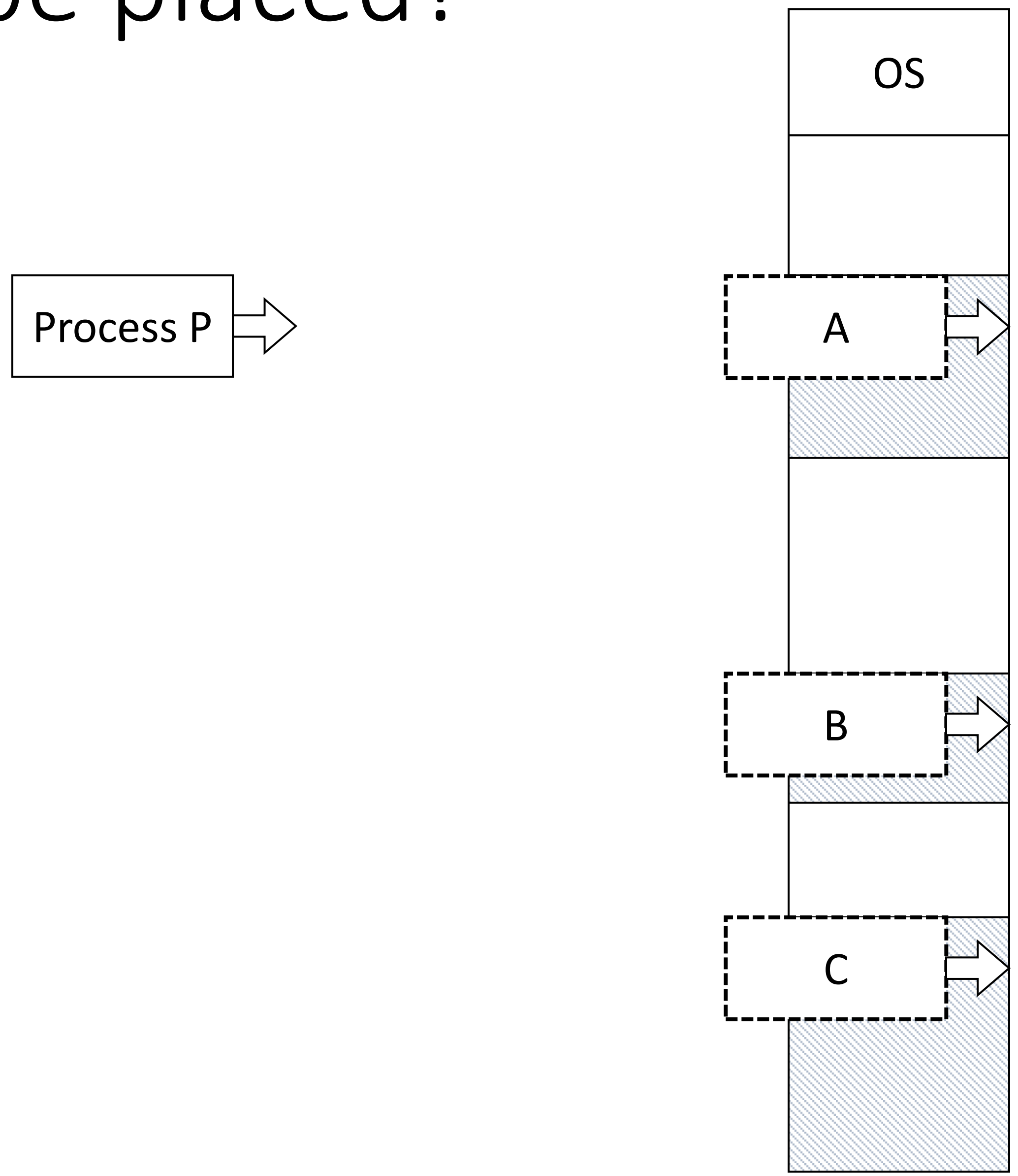
- Physical memory starts as one big empty space.
- When starting new processes, allocate memory.
  - At first, placement is easy: lots of large chunks free
- Over time, processes will terminate, leaving gaps.
- Now we have to decide, for new processes, where should they go?





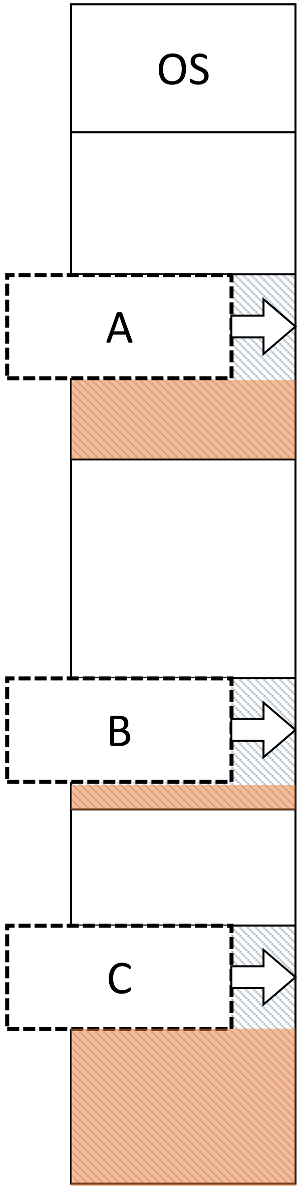
Where should process P be placed?

- Why place it there?



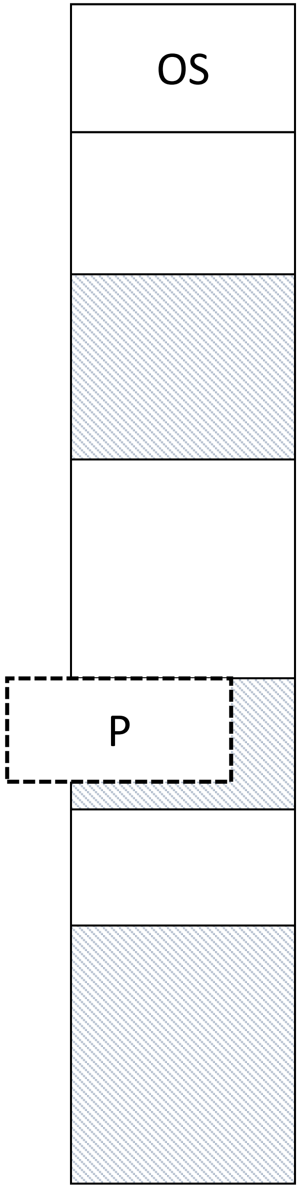
(External) Fragmentation

- No matter where it ends up, the remaining gaps get smaller.
- Large gaps are probably still usable, small ones likely aren't.
- Fragmentation: over time, we end up with these small gaps that become more difficult to use (eventually, wasted).
- "External" because the gaps are between allocated pieces



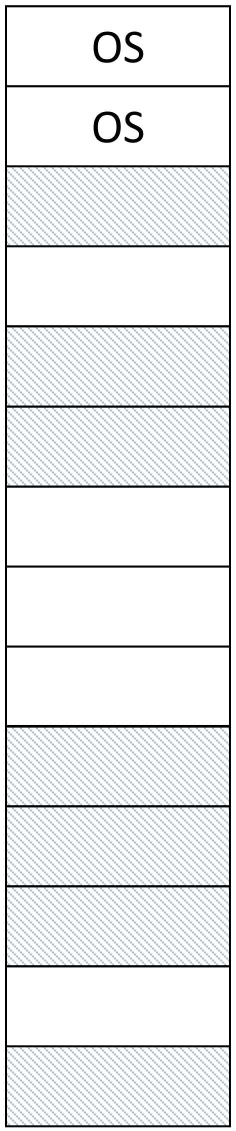
(External) Fragmentation

- Suppose we put it here, and later, P asks for more memory?
- What if there isn't enough space…
  - Move P?
  - Move everybody to compact the address space?
- This seems bad. Lots of tough problems (placement, fragmentation) with no clear solutions.

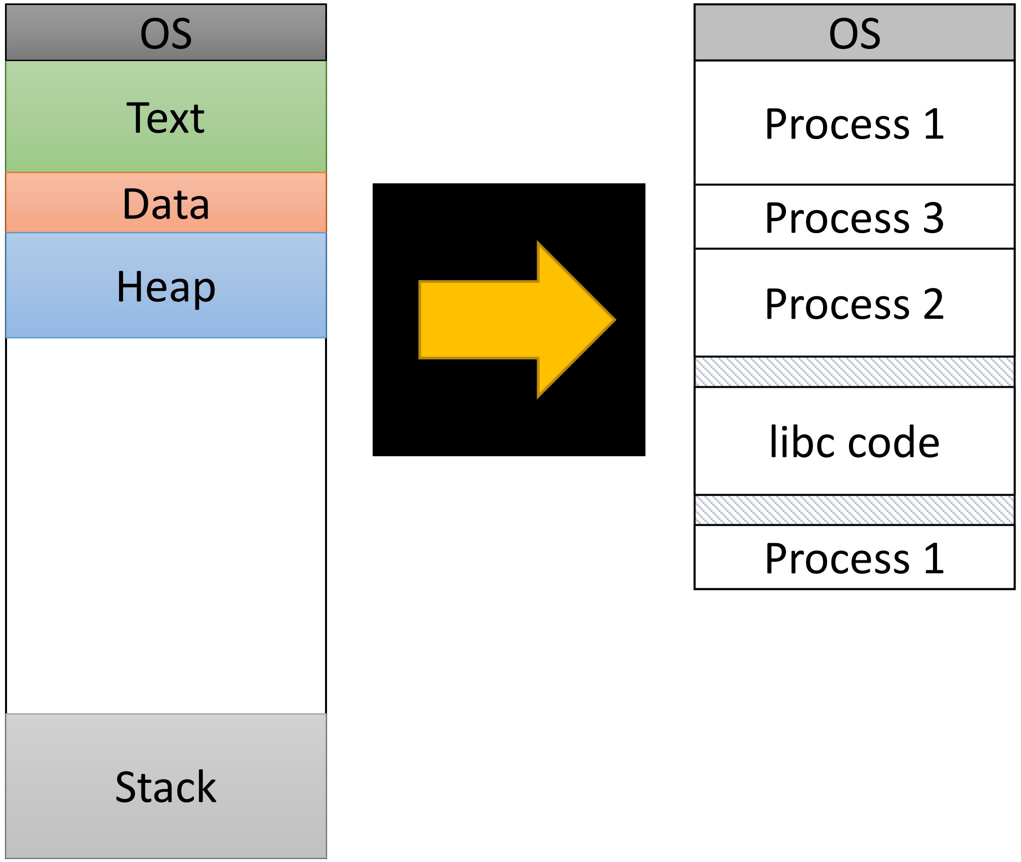


With Virtual Memory

- Divide PAS into fixed size pieces
- Use memory translation to assign virtual addresses to physical locations
- Every physical location is an equally good choice!



Address Translation: Wish List



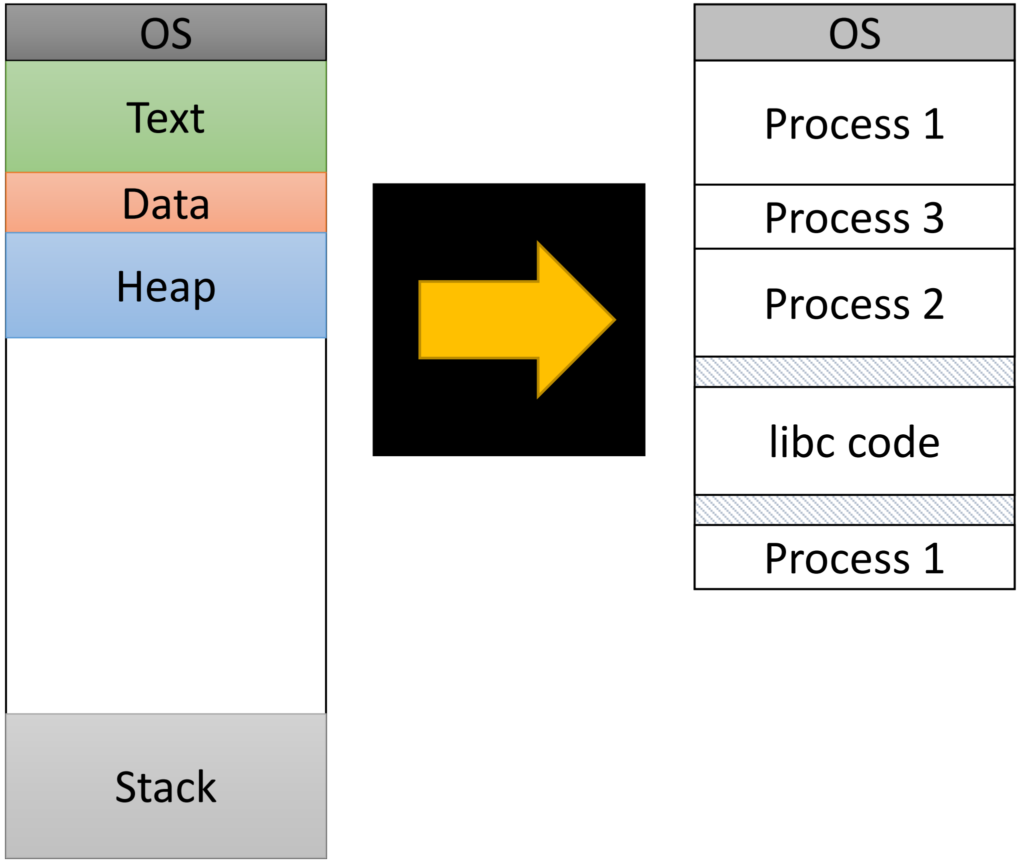
- Map virtual addresses to physical addresses.
- Determine which subset of data to keep in memory / move to disk.
- Allow multiple processes to be in memory at once, but isolate them from each other.
- Allow the same physical memory to be mapped in multiple process VASes.
- Make it easier to perform placement in a way that reduces fragmentation.

OS Perspective

- Primary challenge: Which physical memory do we give to processes?

- Other important considerations:
  - **Protection**: OS is resource gatekeeper, must isolate itself (and processes)
  - **Performance**: OS should map memory for best performance, as long as it doesn't violate protection

Address Translation: Wish List



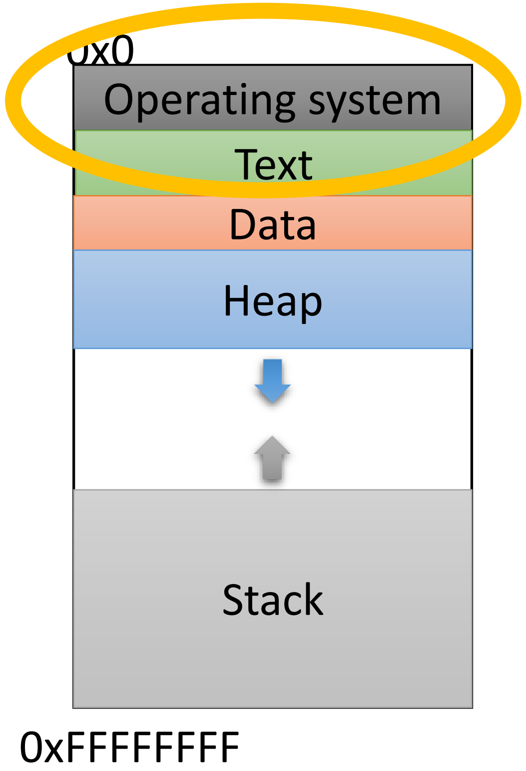
- Map virtual addresses to physical addresses.
- Determine which subset of data to keep in memory / move to disk.
- **Allow multiple processes to be in memory at once, but isolate them from each other.**
- **Allow the same physical memory to be mapped in multiple process VASes.**
- Make it easier to perform placement in a way that reduces fragmentation.

Protection: OS is resource gatekeeper, must isolate itself (and processes)

Performance: OS should map memory for best performance, as long as it doesn’t violate protection

Recall: Context Switching Performance

- Even though it's fast, context switching is expensive:
  1. time spent is 100% overhead
  2. must invalidate other processes' resources (caches, memory mappings)
  3. kernel must execute – it must be accessible in memory



- Solution to #3:
  - keep kernel mapped in every process VAS
  - protect it to be inaccessible

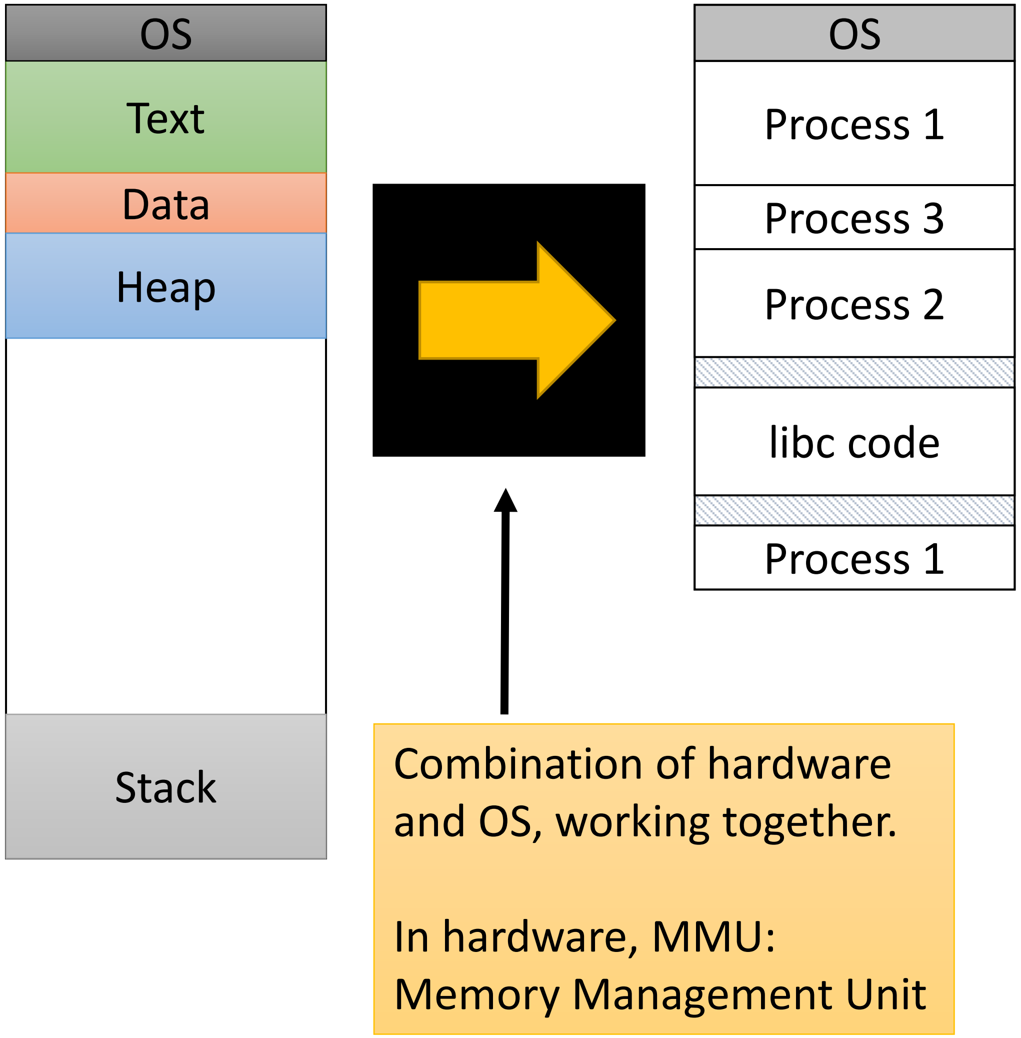
Hardware

- Hardware and OS are symbiotic, often influence each other.
  - We’ve seen one example already: atomic instructions

- Memory management is another important area of collaboration

- Hardware goals:
  - Make translation fast
  - Give OS storage for and control over mappings

Address Translation: Wish List



- Map virtual addresses to physical addresses.
- Determine which subset of data to keep in memory / move to disk.
- Allow multiple processes to be in memory at once, but isolate them from each other.
- Allow the same physical memory to be mapped in multiple process VASes.
- Make it easier to perform placement in a way that reduces fragmentation.
- Map addresses quickly with a little HW help.

Summary

- Users, programmers, compiler, OS all face difficult memory challenges.

- Virtual memory abstraction, despite being complex, is worth it to help solve these challenges.

- We've decided what virtual memory needs to do. (wish list)

- Up next… making it happen.