Operating Systems!

Process Implementation (part 2)

From *"how to use processes"* to *"how an OS implements them!"*

Prof. Travis Peters

Montana State University

CS 460 - Operating Systems

Fall 2020

https://www.cs.montana.edu/cs460

Today

- Announcements
  - *Heads up.... PA1 due* ***Sunday [09/20/2020] @ 11:59 PM (MST)***

- Learning Objectives
  - Understand the big ideas behind the "OS API"
    - ~~**user vs. kernel, modes, syscalls, libraries**~~
  - Understand the big ideas behind process control
    - *[theory]* **control info, creation, termination, states, etc.**
    - *[reality]* ~~**fork, exec, getpid, waitpid, etc.**~~

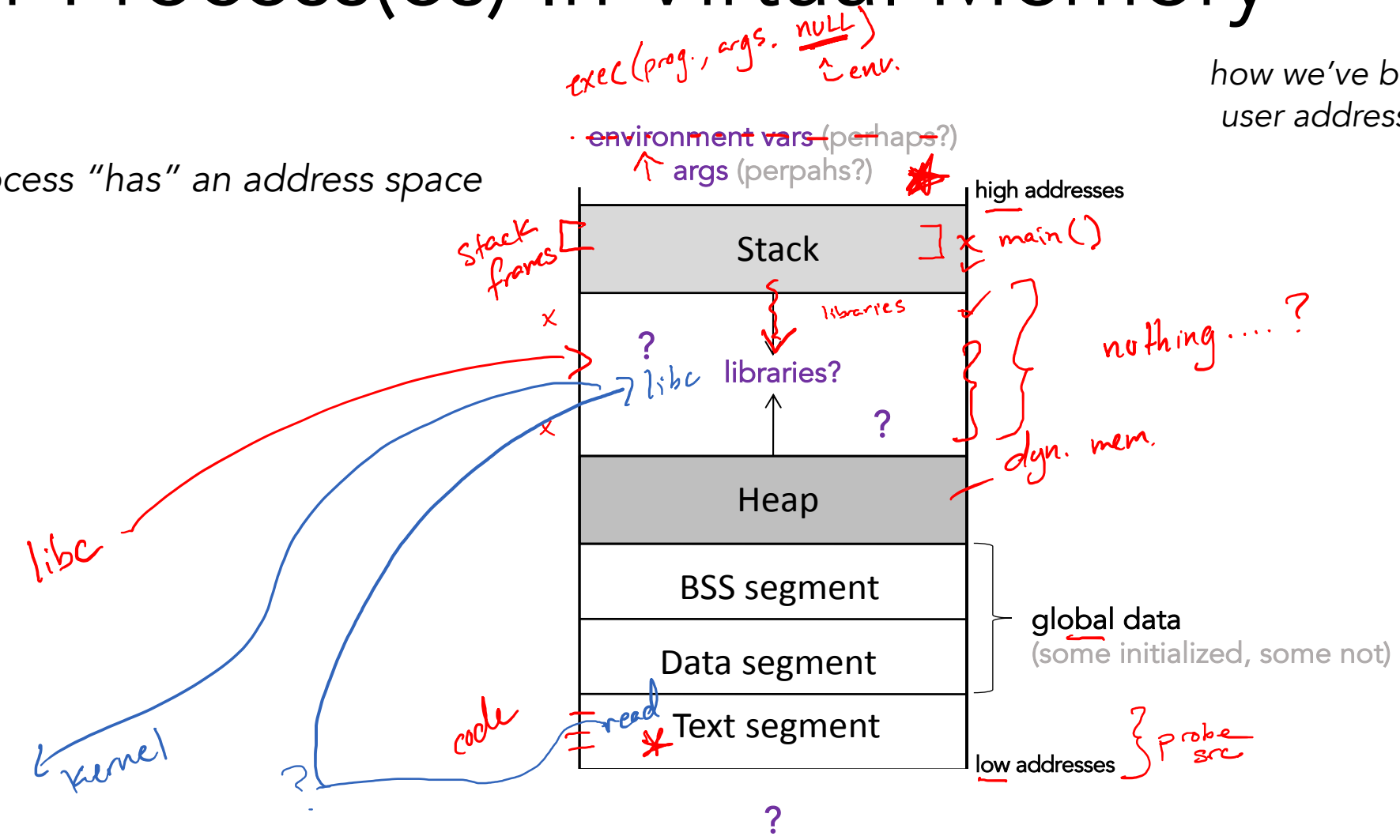
Address Spaces

*Enhancing our view of the address space: **userspace** and **kernelspace***

User Process(es) In Virtual Memory

*how we've been thinking about the user address space (**"userspace"**)*

- *each process "has" an address space*



see demos:

- *probe.c*
- *sf.c*

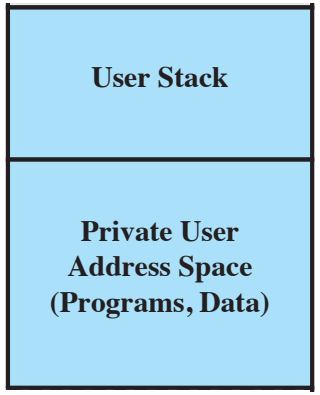
+ procfs

Q: When a running user program makes a syscall down into the OS, do we change processes?

User Process(es) In Virtual Memory

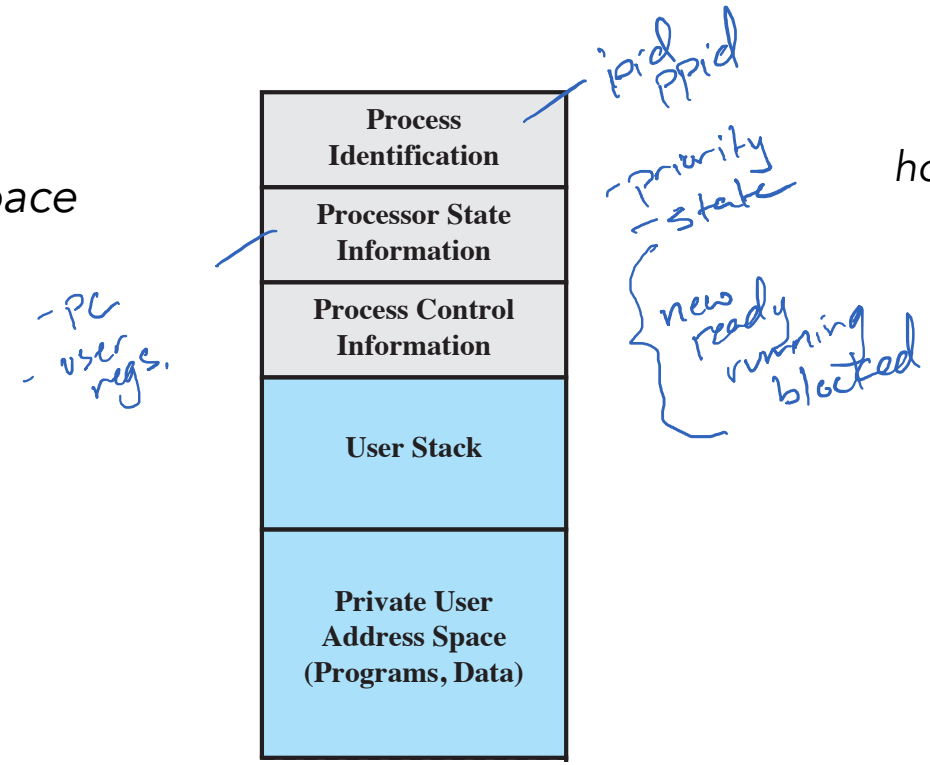
- *each process "has" an address space*

*how the textbook likes to visualize the user address space (**"userspace"**)*



User Process(es) In Virtual Memory

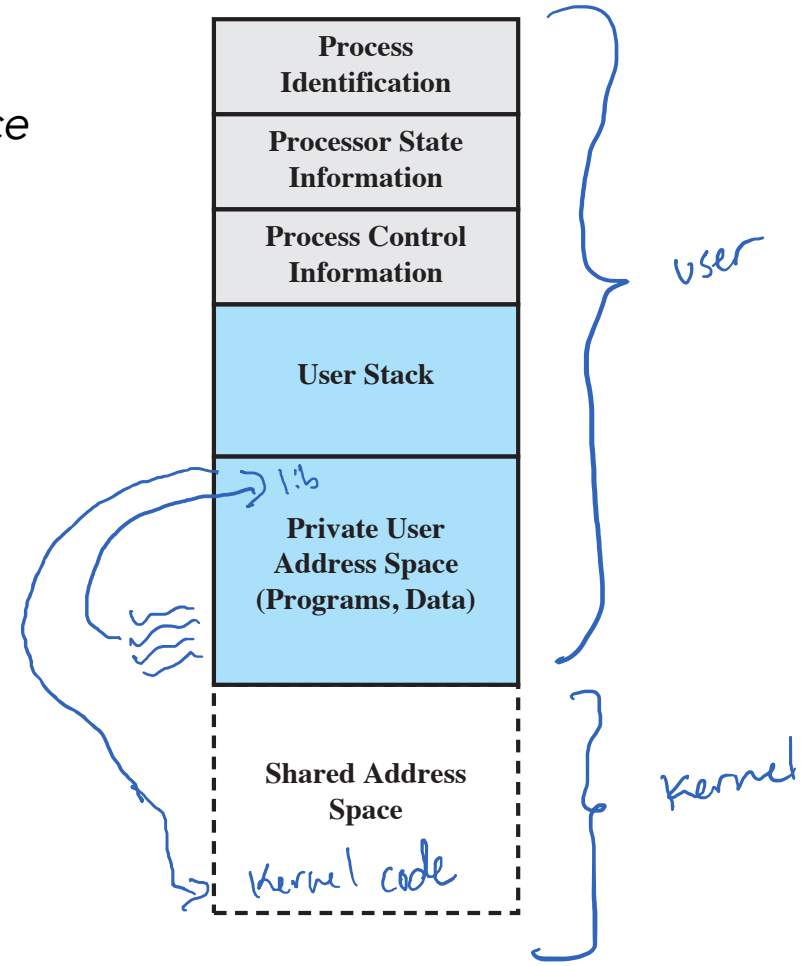
- *each process "has" an address space*
- *+ some other stuff (a "PCB")*



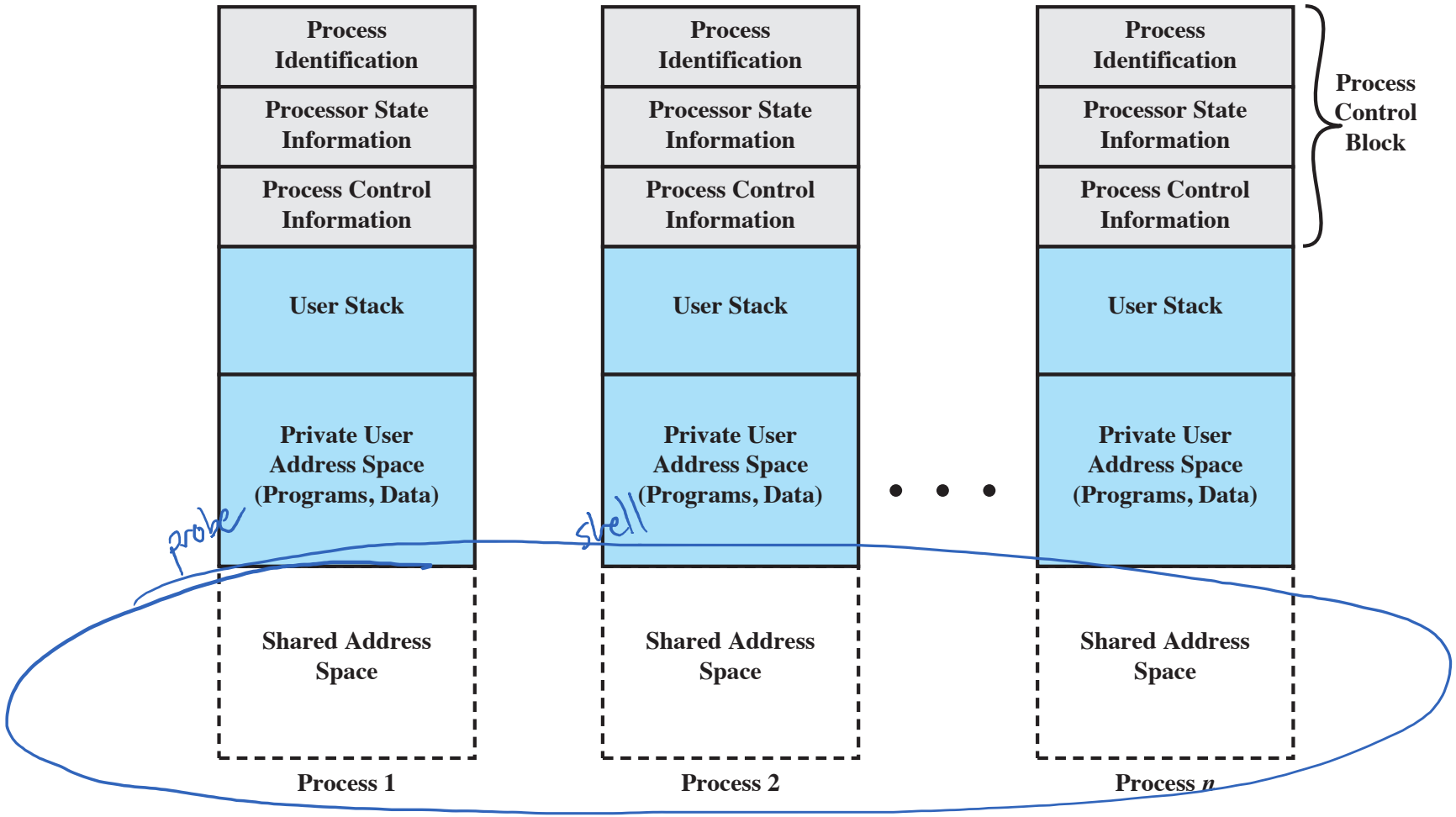
*how the <u>textbook</u> likes to visualize the user address space (**"userspace"**)*

User Process(es) In Virtual Memory

- *each process "has" an address space*
- *+ some other stuff (a "PCB")*
- *+ (some of) the kernel?*

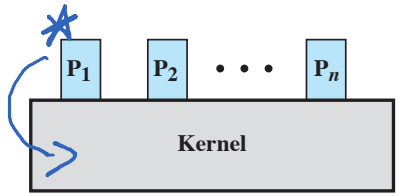


User Process(es) In Virtual Memory

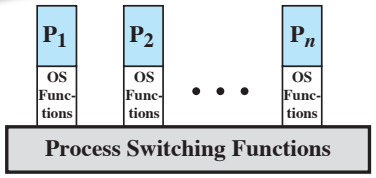


User Process(es) In Virtual Memory

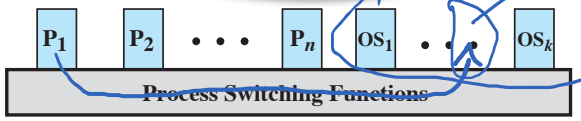
Textbook...



(a) Separate kernel

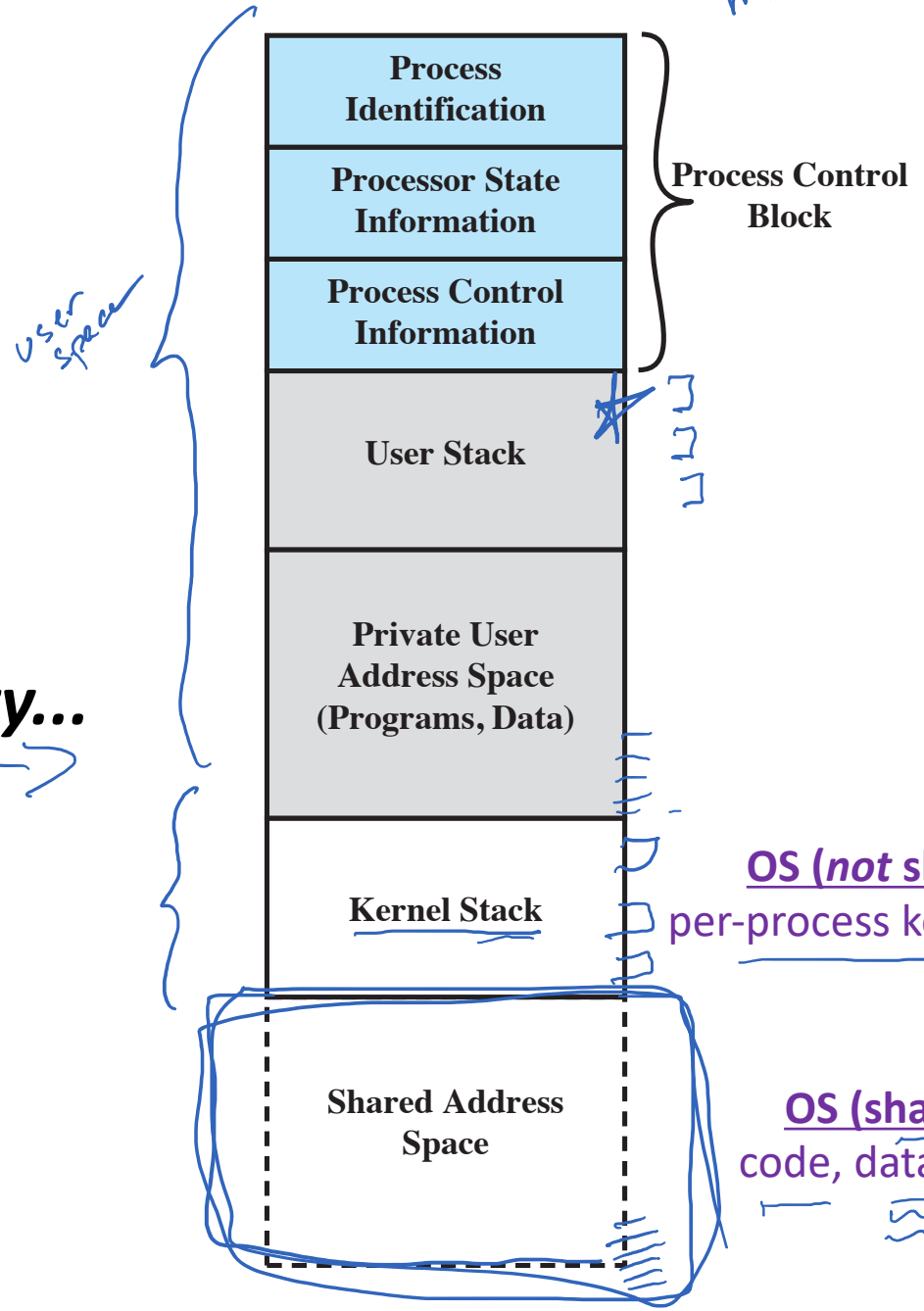


(b) OS functions execute within user processes

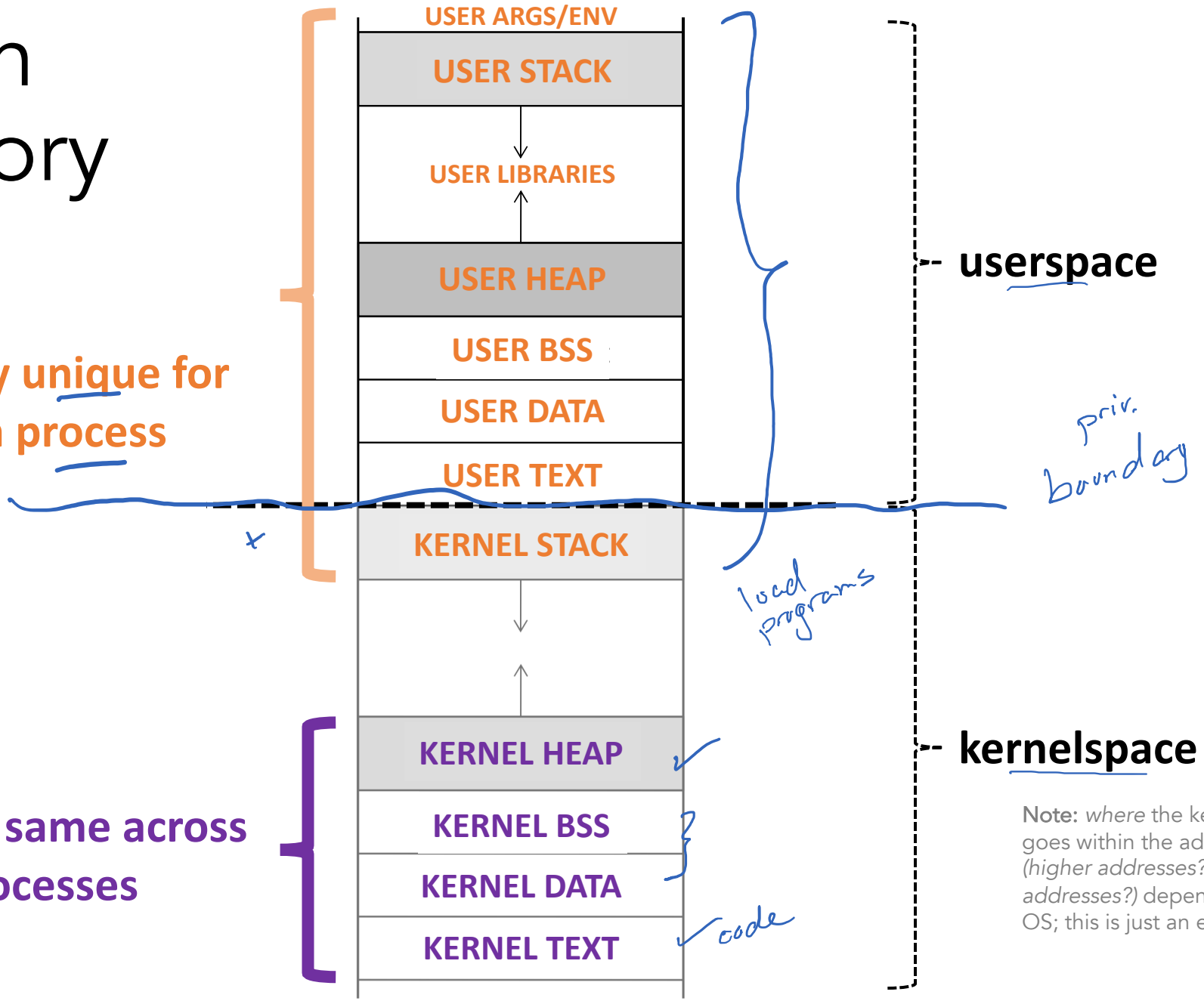


(c) OS functions execute as separate processes

Closer to our reality...



Process(es) In Virtual Memory



Note: *where* the kernel region goes within the address space *(higher addresses? lower addresses?)* depends on the OS; this is just an example.

Process Switching

Reflection/Revisit

Process Switching

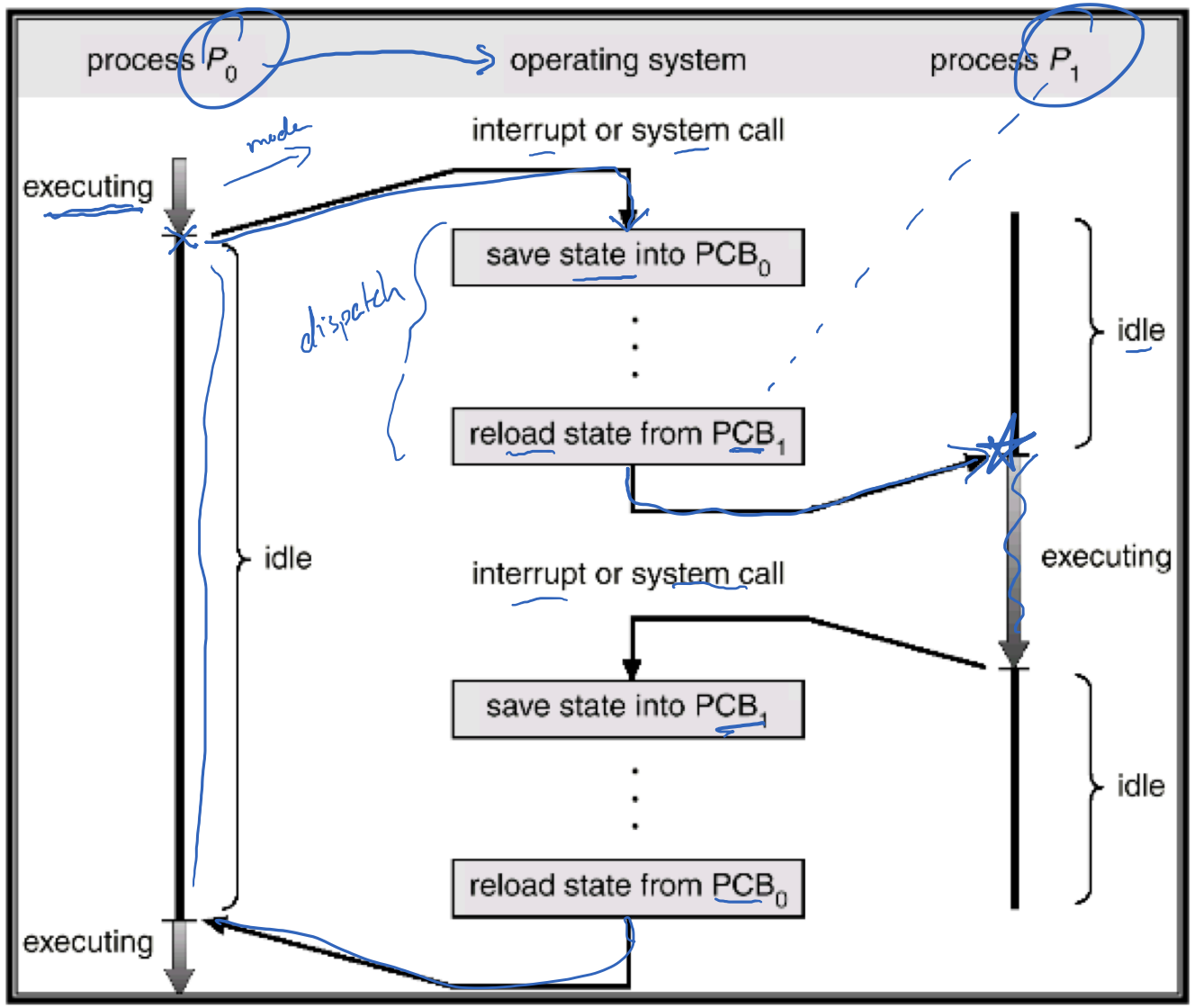
- What events **trigger a process switch**?

Mechanism	Cause	Use
Interrupt	**External to** the execution of the **current instruction**	Reaction to an asynchronous external event
Trap	**Associated with** the execution of the **current instruction**	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

- What is the difference between **mode switching** and **process switching**?
  - mode switch = change of privilege
    → process isn't necessarily changed
    → *recall "OS Interactions (Part 1)"*
  - process switch = change of process
    → swap out all process-related info (e.g., registers, page tables)
    → *see next slide...*

- What is **the role of the OS**? (i.e., ***HOW*** is a process switched?)
  - *save context, update PCB(s), update resource tables, restore context of next process, etc.*

Process Switching



save the context of the processor

update the PCB of the process currently in the running state

move the PCB of this process to the appropriate queue

select another process for execution

update the PCB of the selected process

update memory management data structures

restore the context of the processor to that which existed at the time the selected process was last switched out

Syscalls

Reflection/Revisit – So how might some of our syscalls work?

Fork

- What sorts of things happen when process A calls **fork**?
  - go down to fork handler, in kernel mode, in process A
  - build a new PCB and new address space for a new process B
  - how should we initialize process B's user context?
  - ...what about process B's kernel context?
  - What to do with process B? (where does it "go"?)

Exec

- What sorts of things happen when process A calls **exec**?
  - rebuild the user space to be the intended program *(in the appropriate, initial state)*
  - *... what else?! ...*

Waitpid

- What sorts of things happen when process A calls **waitpid(B)**?
  - has process B terminated already? *(we better know...)*
  - if B has **not** terminated, what should we do?
    - *At a minimum, block A and move it to the waiting queue*
  - If B has terminated, what should we do?

- Other things to consider:
  - what if child has died, but its exit code hasn't been collected? **"zombie" or "defunct"**
  - what if child is alive, but its parent dies? **"orphan" → "adopted"? (perhaps by Init)**
  - what if child has died, and its parent is dead too...? (so it can't/won't collect the exit code)

Probably need to store PCBs of processes that have been terminated...

see demo:
- ***defunct.c***