The Process Abstraction

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

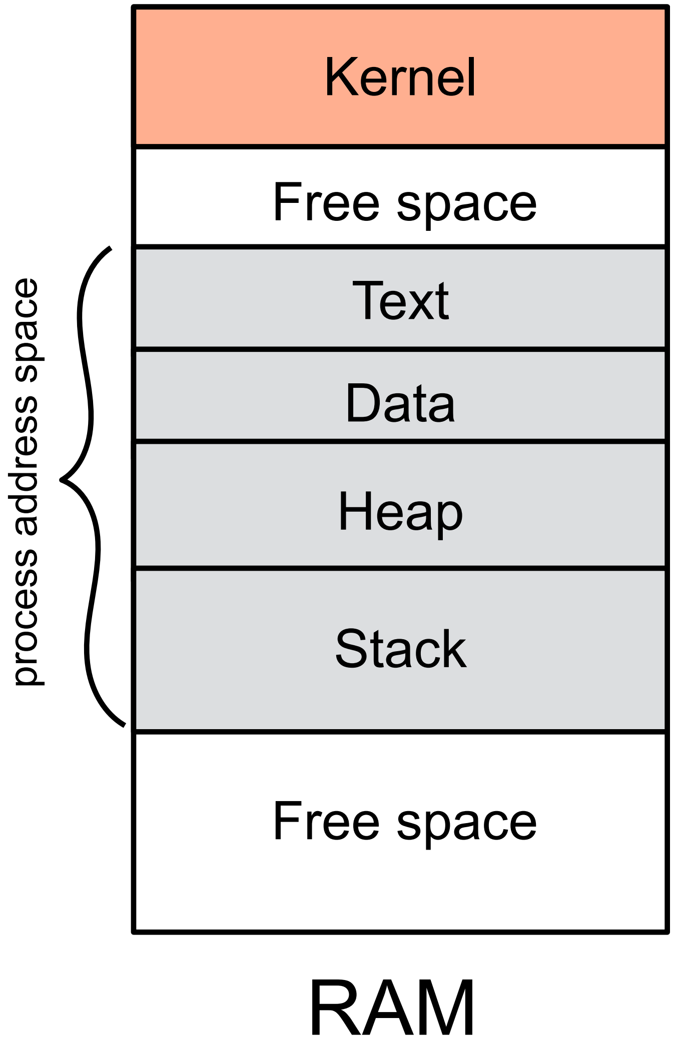
Definition

- A process is a program in execution
  - Program: passive entity (bytes stored on disk as an executable file)
  - Becomes a process when it is loaded into memory, at which point the fetch-decode-execute cycle can begin
  - The process abstraction is defined by the OS to virtualize the CPU
- Multiple processes can be associated to the same program
  - A user can start multiple instances of the same program (e.g., bash)
- Typically many processes run on a system
  - System processes (started by the OS to do “system things”)
  - User processes (started by users)
  - The terms “process” and “jobs” are used interchangeably in OS textbooks
- The set of locations that store bytes that a process can use/reference is called the process’ address space…

Process Address Space

- The code (also called text)
  - Binary instructions, loaded into RAM by the OS from an executable file
- The static data
  - The global variables and static local variables, which can be initialized (.data segment in x86 assembly) or not (.bss segment in x86 assembly)
- The content of all registers
  - They represent the state of the CPU in the current fetch-decode-execute cycle
  - This includes the program counter (PC)
- The heap
  - The zone of RAM in which new data can be be dynamically allocated (using malloc, new, etc.)
- The runtime stack
  - The zone of RAM for all bookkeeping related to method/procedure/function calls (more in the next slides)
- The page table
  - Let's not talk about it now and leave it for later…

Process Address Space



- The OS can be configured to limit parts of a process' address space
  - On UNIX-like systems you can find out what some limits are (all in KB):
    - **`ulimit -d`** (data + heap)
    - **`ulimit -s`** (stack size)
    - **`ulimit -m`** (maximum Resident Set Size)
  - These limits can be changed system-wide using the **`ulimit`** command
  - They can also be changed by the process itself using the **`setrlimit()`** system call
  - Let's see what limits are on my server…
- When running a Java program you can specify some limits
  - **`java -Xmx512m -Xss1m`** …
  - 512 MiB maximum heap size, 1MiB maximum stack size

The Heap

- New (i.e., dynamically allocated) bytes (objects, arrays, etc.) are allocated on the Heap (**`malloc()`** in C, **`new`** in Java/C++/C#, implicit in Python, etc.)
- Can be handled by a memory manager (e.g., the JVM, a library, the Python interpreter) but ultimately it is the OS that provides dynamic memory allocation
  - There is a system call that says “please OS, give me XX more bytes”
- At some point you will get an Out Of Memory error if you keep dynamically allocating memory
- On my Linux box (not Docker), let’s write a simple C program that calls **`malloc()`** 10,000 times for 1 byte and look at the addresses returned…

The Heap: what we observed

- When calling **`malloc()`** for 1 byte, the space used is actually more than 1 byte!
  - In our case addresses were 32 bytes apart, so we "wasted" 31 bytes for each **`malloc()`**!!
- Calling **`malloc()`**, say, 10,000 times for 1 byte "wastes" memory when compared to calling **`malloc()`** 1 time for 10,000 bytes
- This is due to the implementation of the OS's "memory allocator"
  - It needs to store meta-data about the chunk of memory allocated so that later it knows what to do when **`free()`** is called
  - It will often allocate memory at addresses that are multiple of some small power of 2
- Let's now **`strace`** this program we just wrote and see what the "give me more memory!" system call is…

The `brk` syscall

- The “give me more memory!” system call is **`brk()`**
- The man page for **`brk()`** shows that it is used to extent the heap up to some address that is beyond the current “end of the heap” address
  - **`brk(NULL)`** “asks” where the data+heap ends
- Weirdly, although our programs calls **`malloc()`** a lot, it calls **`brk()`** only a few times
- This is an optimization:
  - A call to **`malloc()`** can get more memory than asked
  - Subsequent calls to **`malloc()`** just grab some of that extra memory without any syscall at all! (less overhead, more speed)
- So when calling **`malloc(1)`**, memory footprint can grow by a lot more than 1 byte!
- Let’s figure out how much memory, in KiB, is allocated by a “first” call to **`malloc()`**
  - Anybody has some idea how we can do this?

The Heap: what we found out

- What we likely just did:
  - Count the number of **`malloc()`** calls between two calls to **`break()`**
  - Multiply that number of calls by 32
  - Divide by 1024
  - And that's the number of KiB we get the "first time" we call **`malloc(1)`**

- Of course are a little bit more complicated than that… isn't everything in the OS?

The Runtime Stack

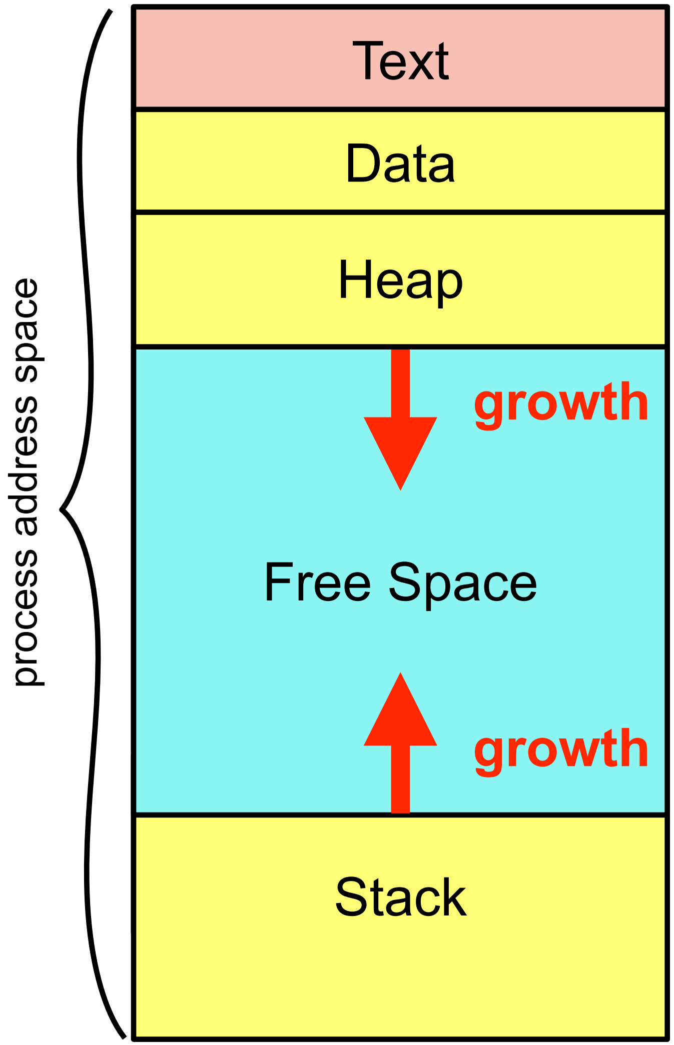
- Each process has in RAM a stack (a last-in-first-out data structure) where items can be pushed or popped
- It is used to manage method/procedure/function calls and returns
- On each call, an activation record is pushed onto the stack to do all the bookkeeping necessary for placing/returning from the call
  - It contains parameters, return address, local variables, saved register values
- The code to manage the stack is generated by compilers/ interpreters
  - In ICS 312 we learn all the details
- The stack size is limited
  - But configurable upon process creation
- Going over that limit is called a Stack Overflow
  - Happens, for instance, with a deep (or infinite) recursion

The Kernel Stack

- The code in the kernel uses functions, and therefore it must have a stack to call these functions
- But, to save space, the kernel's stack is very small (16KiB!!)
- Therefore, when writing functions in the kernel, these functions cannot allocate a lot on the stack
  - Not many parameters, not many local variables, no deep call sequences, and definitely no recursion

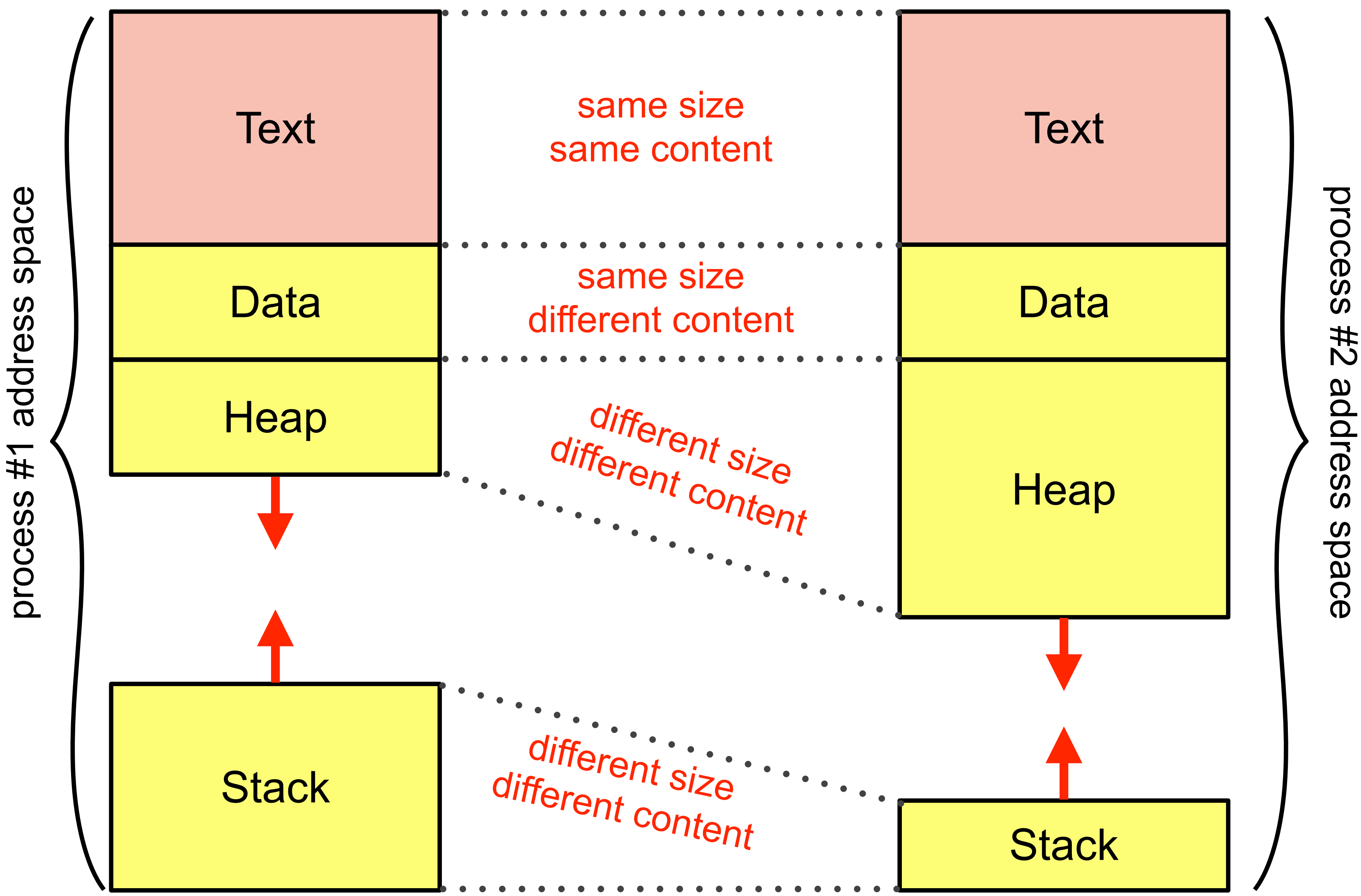
- This is one of the differences between user-level development and kernel-level development
  - Recall others, like the fact that kernel code cannot use standard libraries, because standard libraries use system calls, which are implemented in the kernel (chicken and egg problem)
  - e.g., you can't use `printf` when writing kernel code

Logical Address Space



- Typical depiction of a process' address space
  - The heap grows toward high addresses
  - The stack grows toward low addresses
  - If they collide you've run out of memory
- This is the logical view of a process' address space (i.e., virtualization of memory)
- Let us easily experience this logical view by writing a C program that prints text, data, heap and stack addresses on Linux…
- But this is **not at all** what things look like in physical memory
  - Because of "paging", which we'll talk about much later in the semester
  - And because that "free space" (in blue) would be a total waste if the program doesn't need additional stack/heap space!

Two Processes / One Program Example



Process Life Cycle

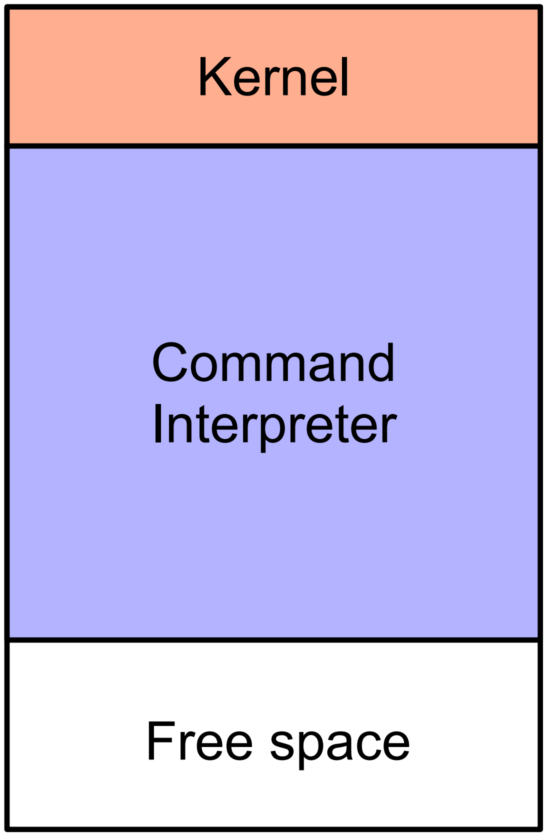
- Each process goes through a lifecycle
- This term (in computer science) means that:
  - There is a finite number of possible states
  - There are allowed transitions between some states
  - These transitions happen when some event occurs

- Before we look at the current process file cycle, let's go back in time to so-called "single-tasking OSes"…

Single-Tasking Ones

- OSes used to be single-tasking: only one process could be in memory at a time
- MS-DOS is the (last commercial?) most well-known example
  - A command interpreter is loaded upon boot
  - When a program needs to execute, no new process is created
  - Instead the program's code is loaded in memory by the command interpreter, **which overwrites part of itself with it!**
    - Done to cope with a very small RAM back in the days
  - The instruction pointer is set to the 1st instruction of the program
  - The small left-over portion of the interpreter resumes after the program terminates
  - This small portion reloads the full code of the interpreter from disk back into memory
  - The full interpreter is resumed

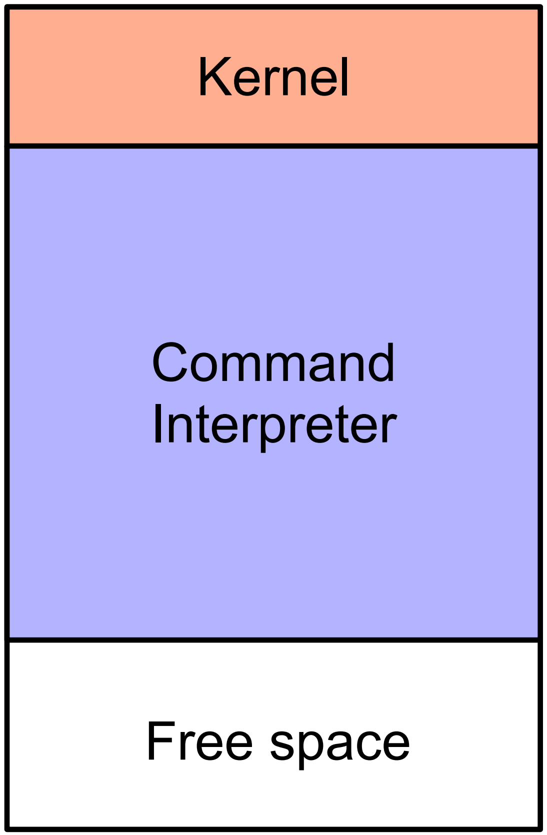
Single-Tasking with MS-DOS



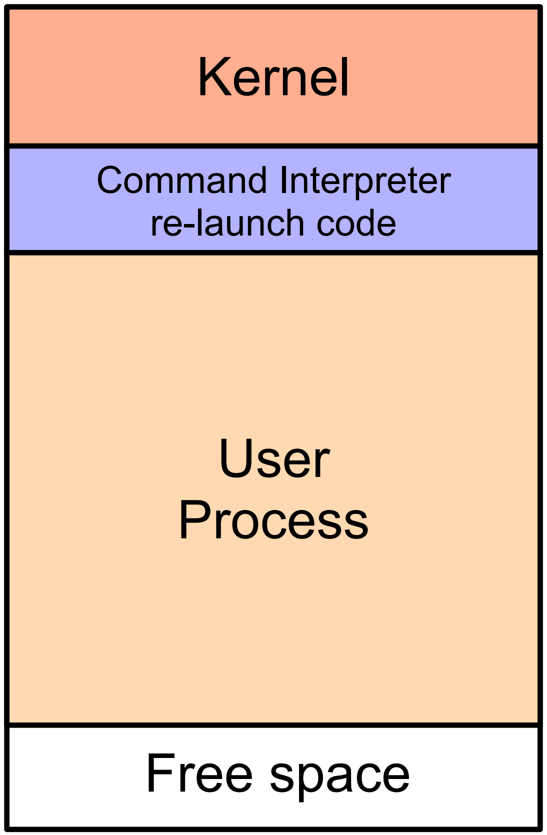
Idle

Full command interpreter

Single-Tasking with MS-DOS



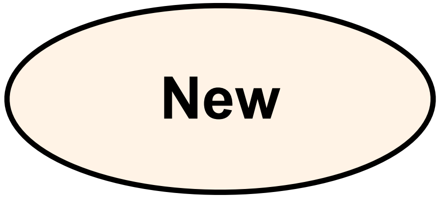
Idle
Full command interpreter



Running a program
Reduced command interpreter

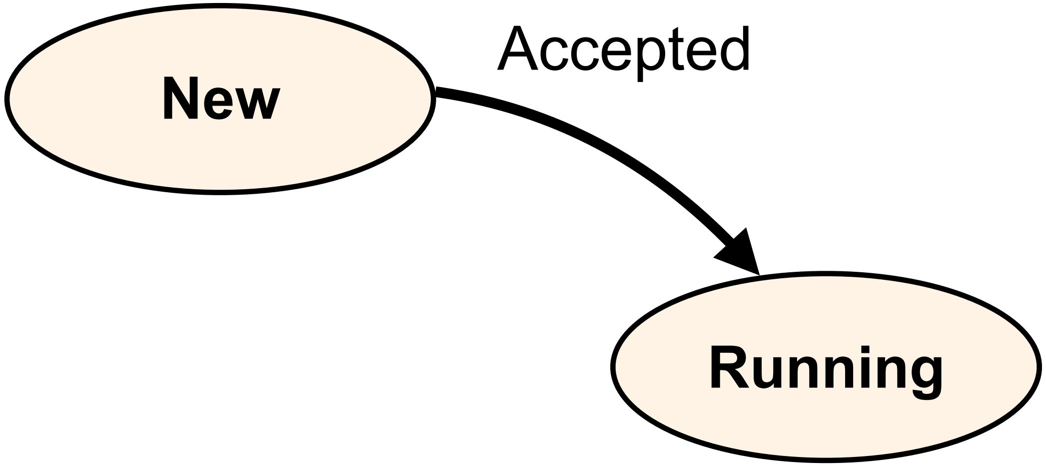
Single-Tasking Process Lifecycle

- The process lifecycle was very simple:



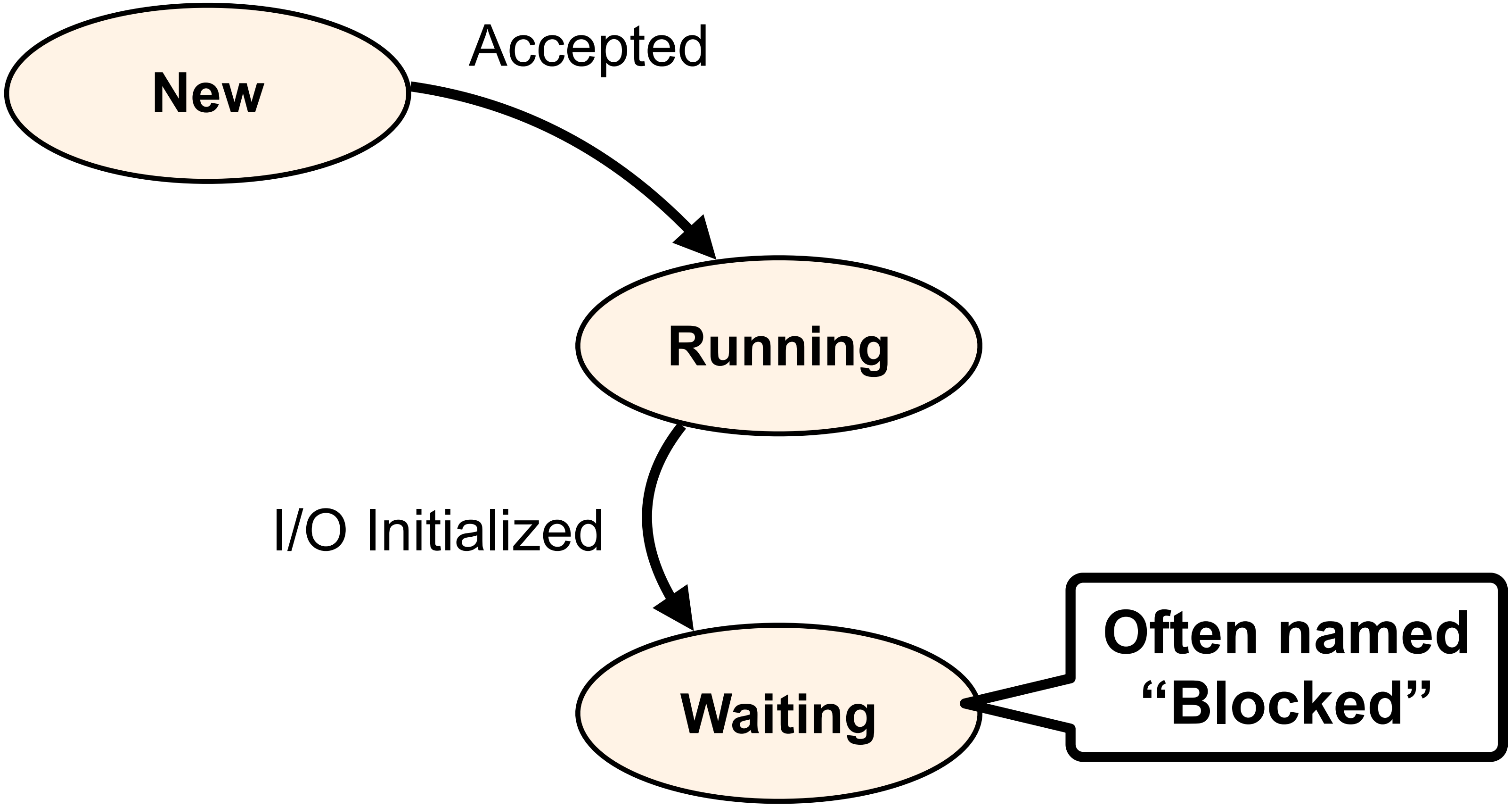
Single-Tasking Process Lifecycle

- The process lifecycle was very simple:



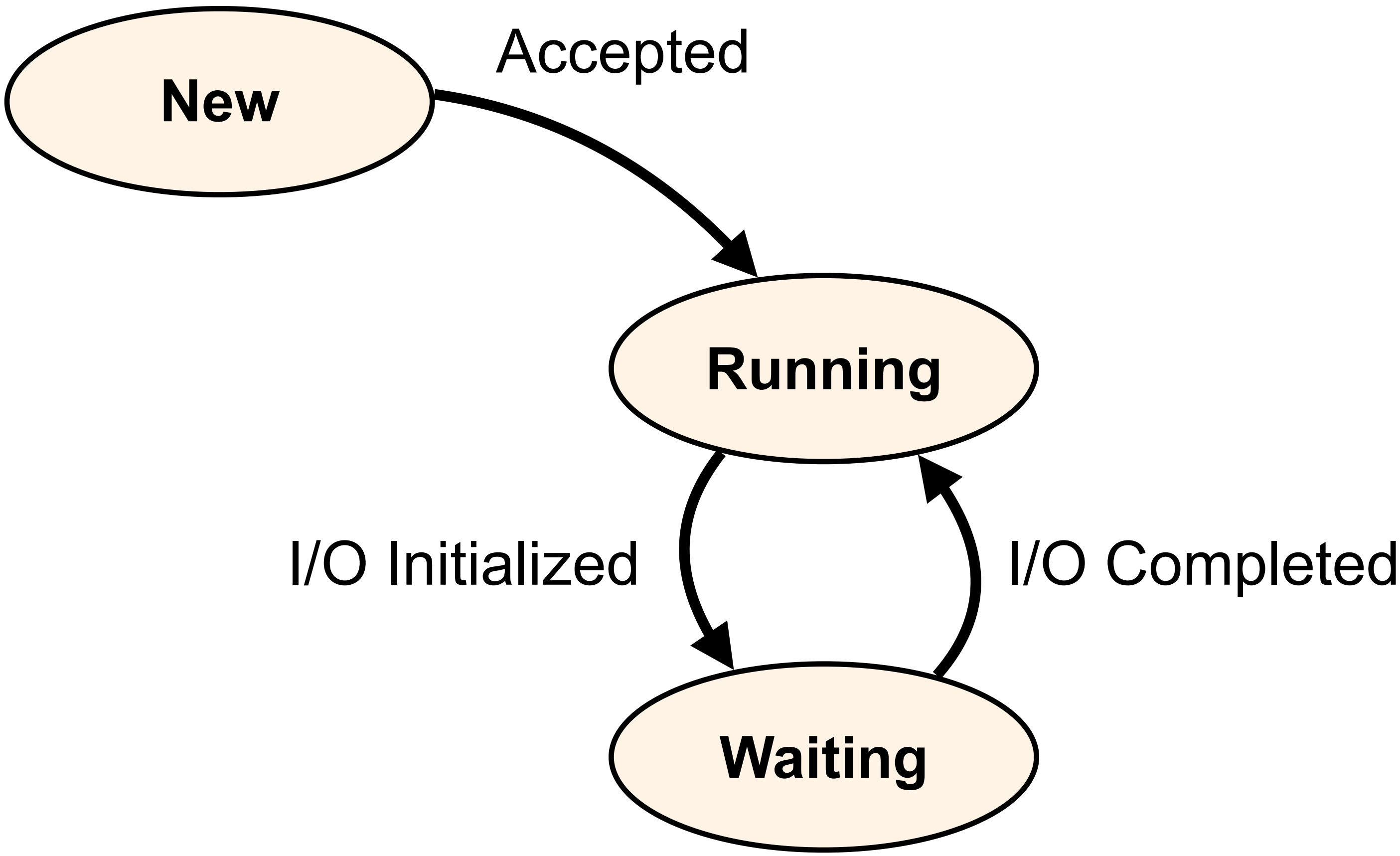
Single-Tasking Process Lifecycle

- The process lifecycle was very simple:



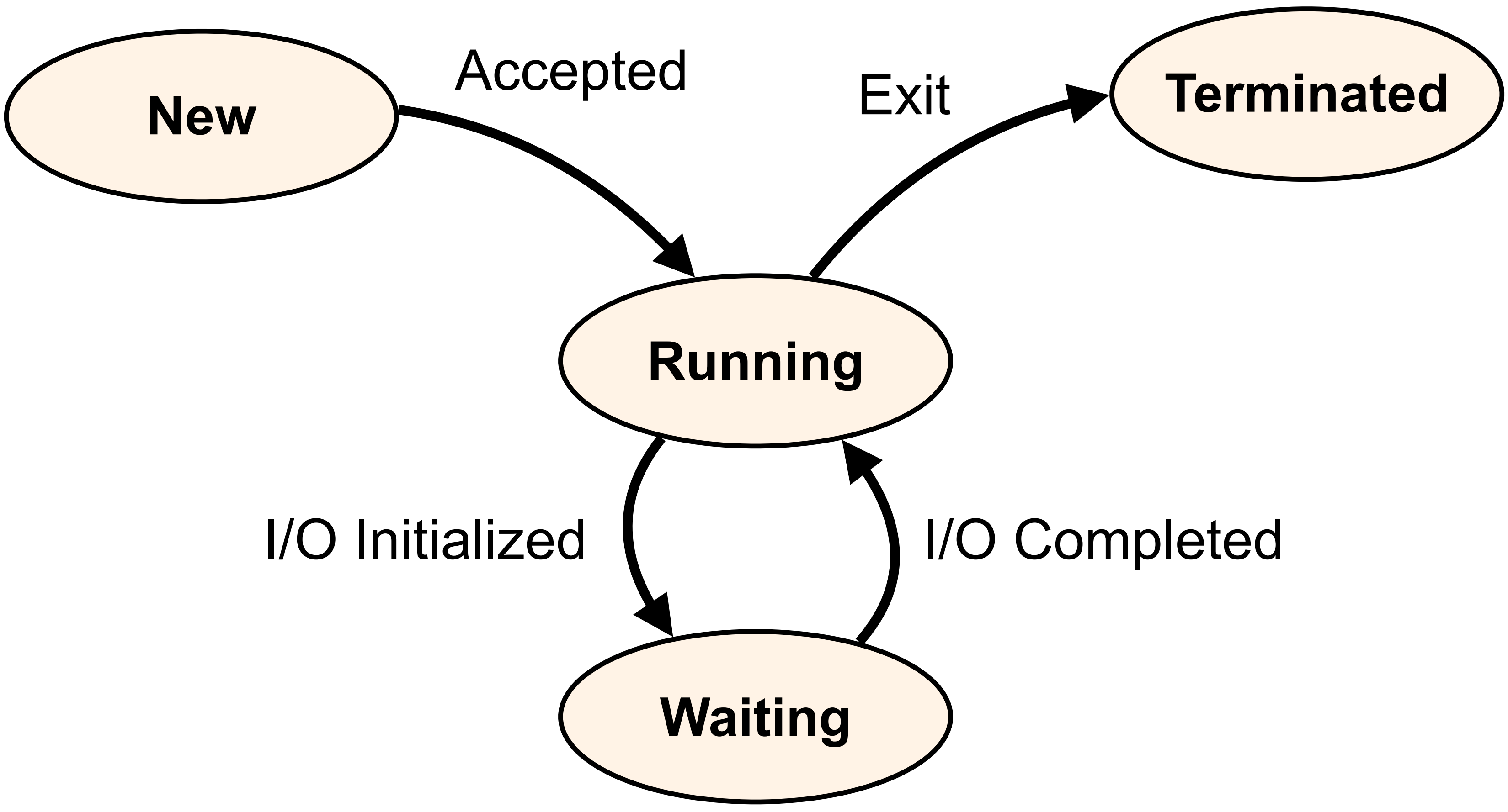
Single-Tasking Process Lifecycle

- The process lifecycle was very simple:



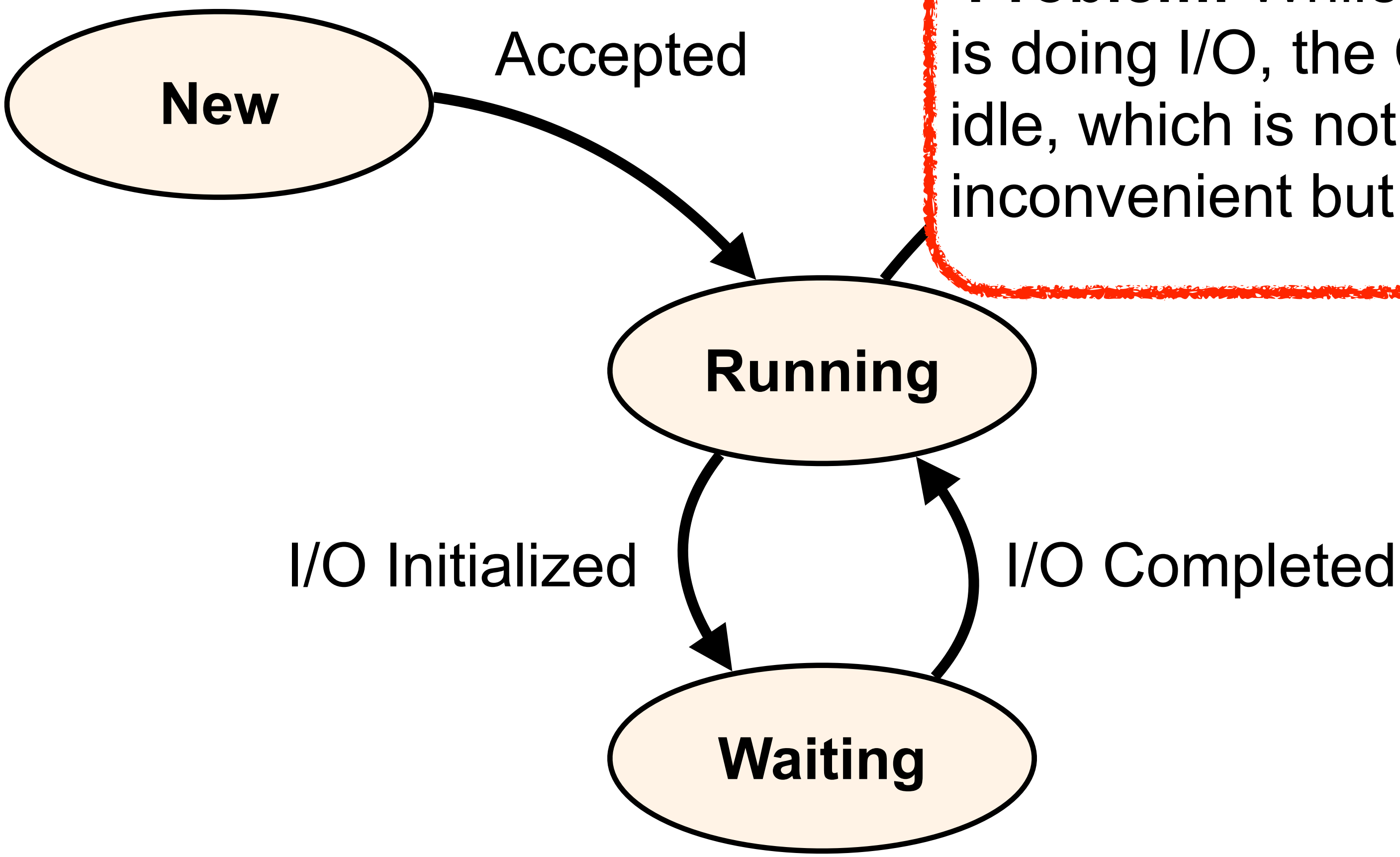
Single-Tasking Process Lifecycle

- The process lifecycle was very simple:



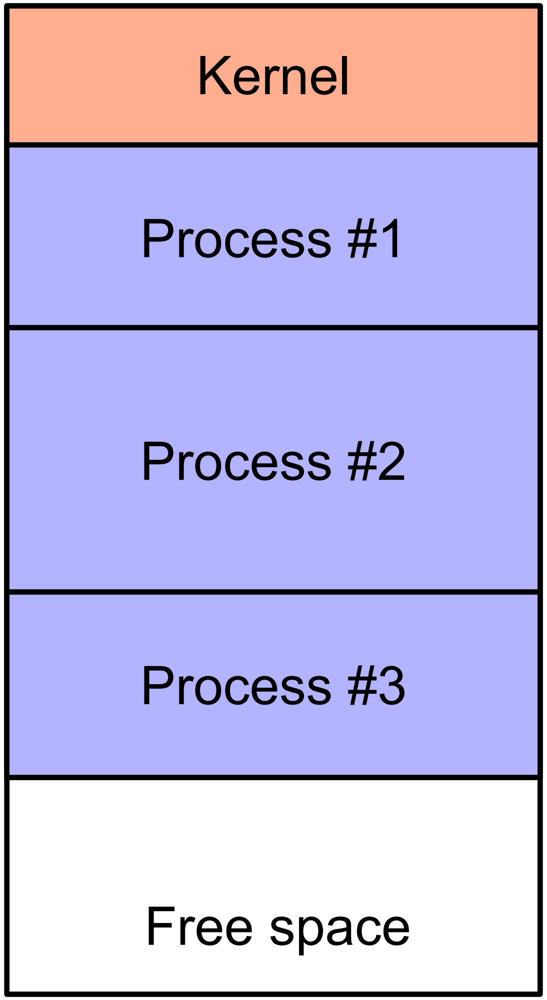
Single-Tasking Process Lifecycle

- The process lifecycle was very simple:



Problem: While a process is doing I/O, the CPU is idle, which is not only inconvenient but a waste

Multi-Tasking (aka Multiprogramming)

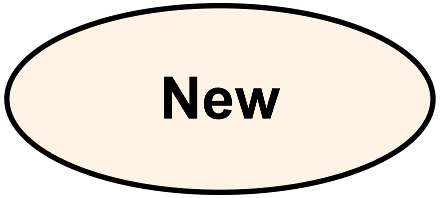


- In modern OSes, multiple processes can be in RAM at the same time
  - Each with its own address space
  - While it's running, a process thinks it's alone on the machine (it doesn't see anything outside of its address space)
- There is a system call to create a new process that any process can place (to create a "child" process)
- When a process terminates, its RAM space is reclaimed by the OS
- Therefore, a process can be ready to run but not running because another process is currently running on the CPU
- **The lifecycle needs a new state!**

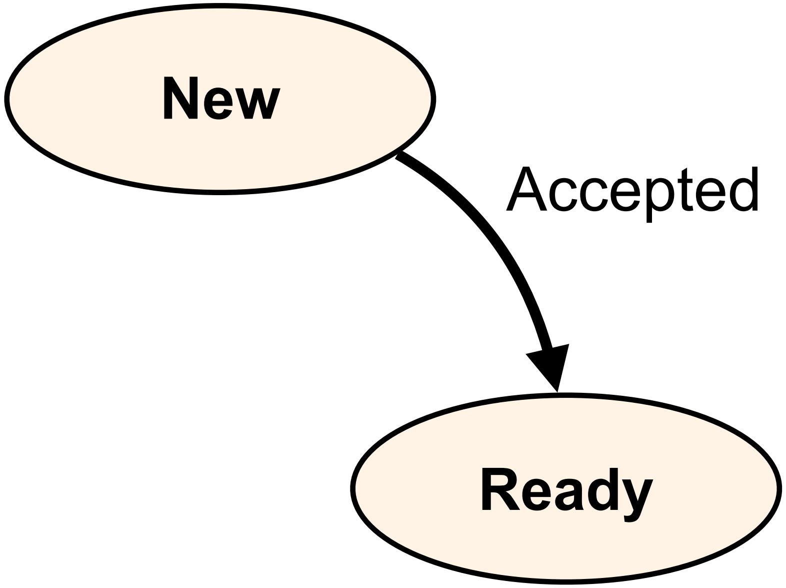
The Ready State

- A process can be ready to run but not currently running: it's in the ready state
- It is the job of the OS to select one of the ready processes whenever the CPU becomes idle
  - This is part of what's called "scheduling"
- This is how the OS virtualizes the CPU, so that each process has the illusion it is the only one using the CPU
- We have a more complicated lifecycle…

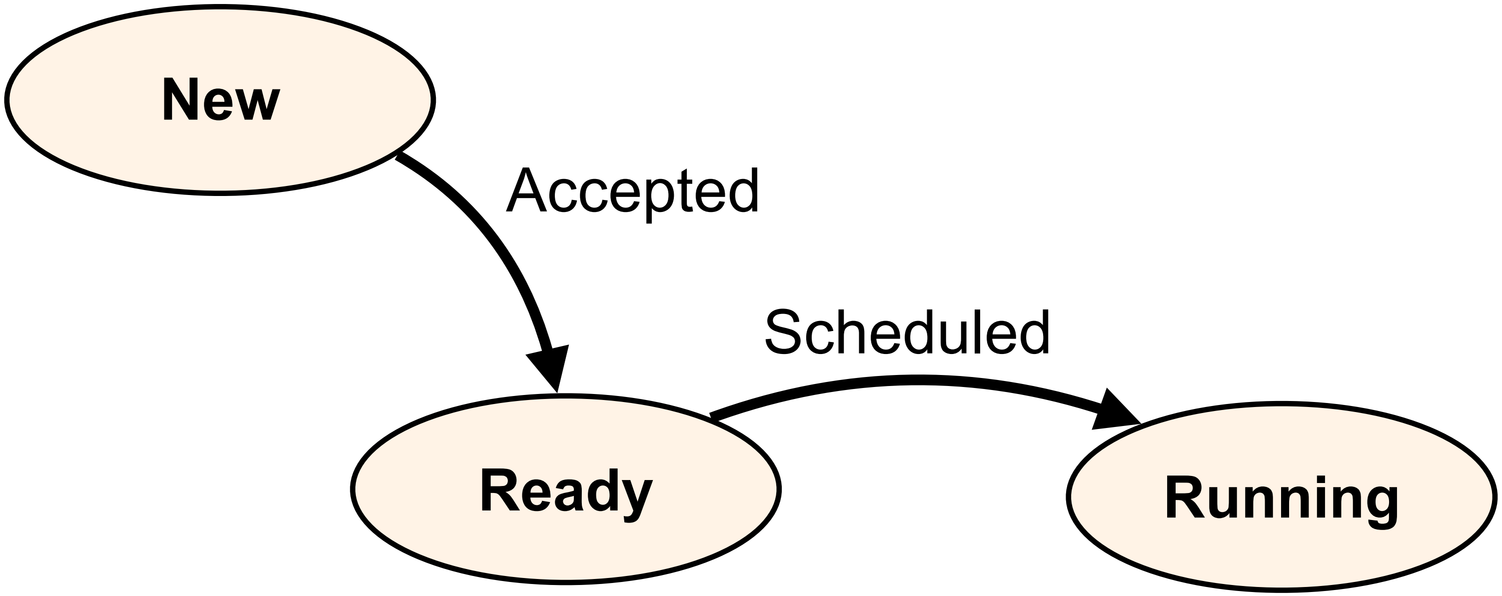
Process Lifecycle



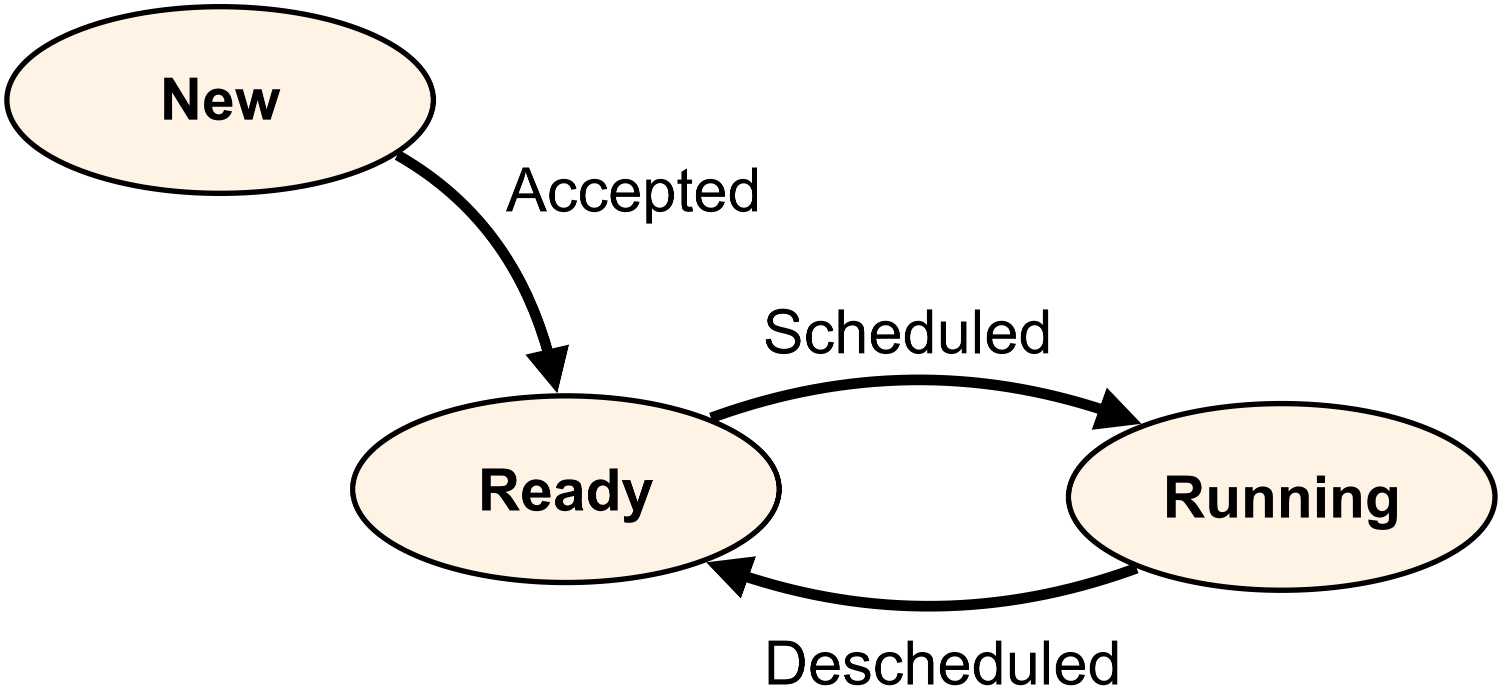
Process Lifecycle



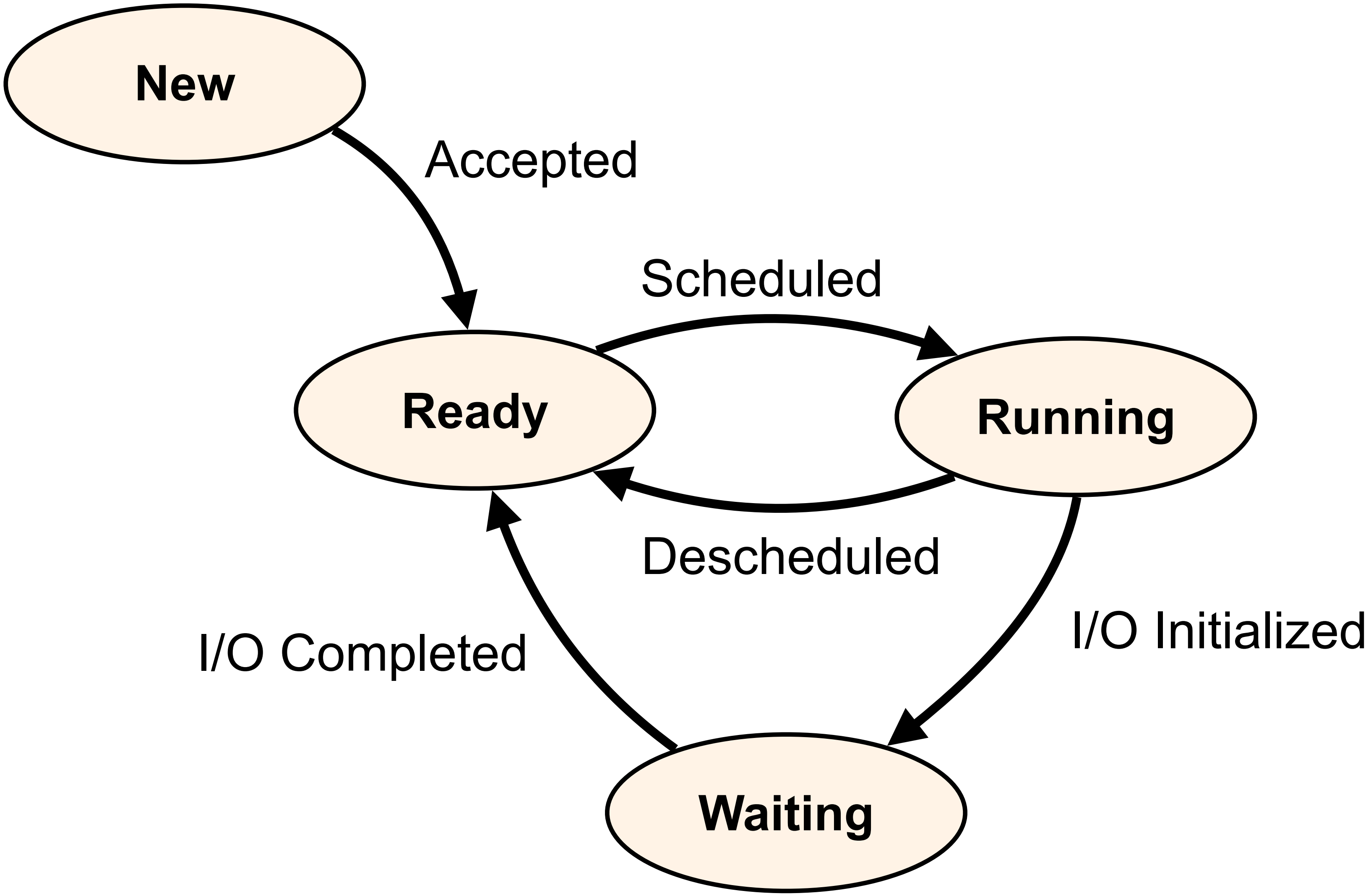
Process Lifecycle



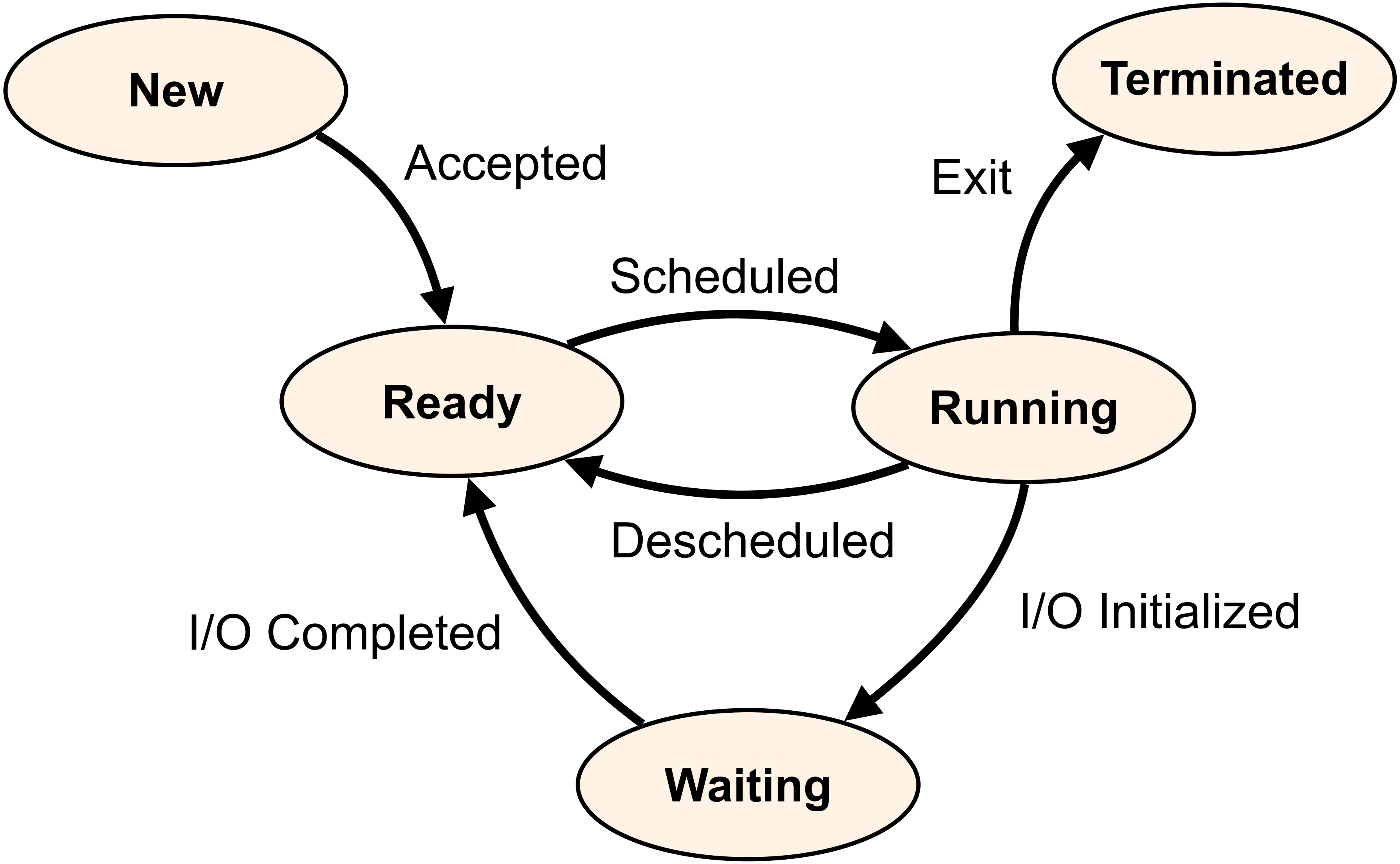
Process Lifecycle



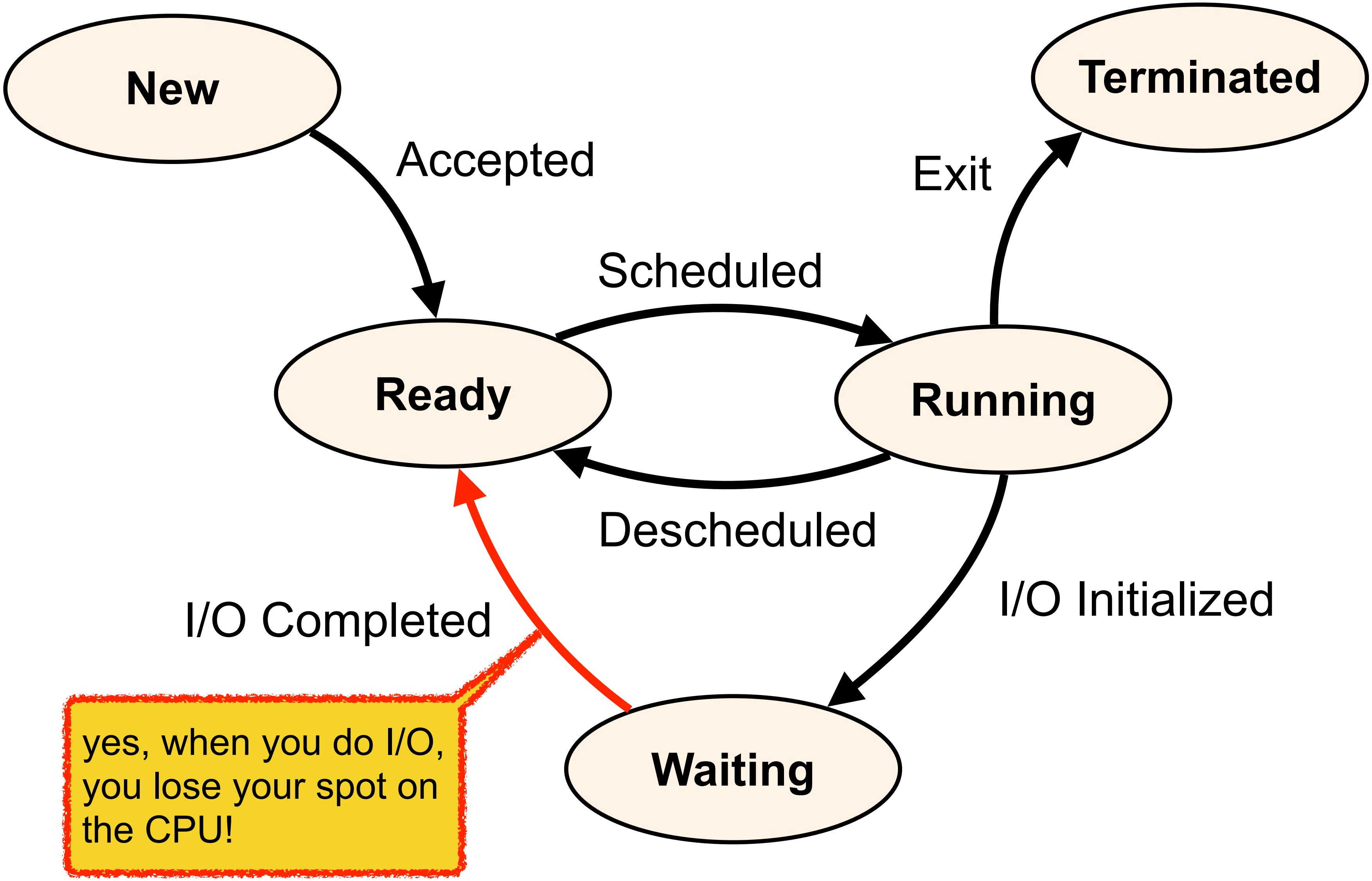
Process Lifecycle



Process Lifecycle



Process Lifecycle

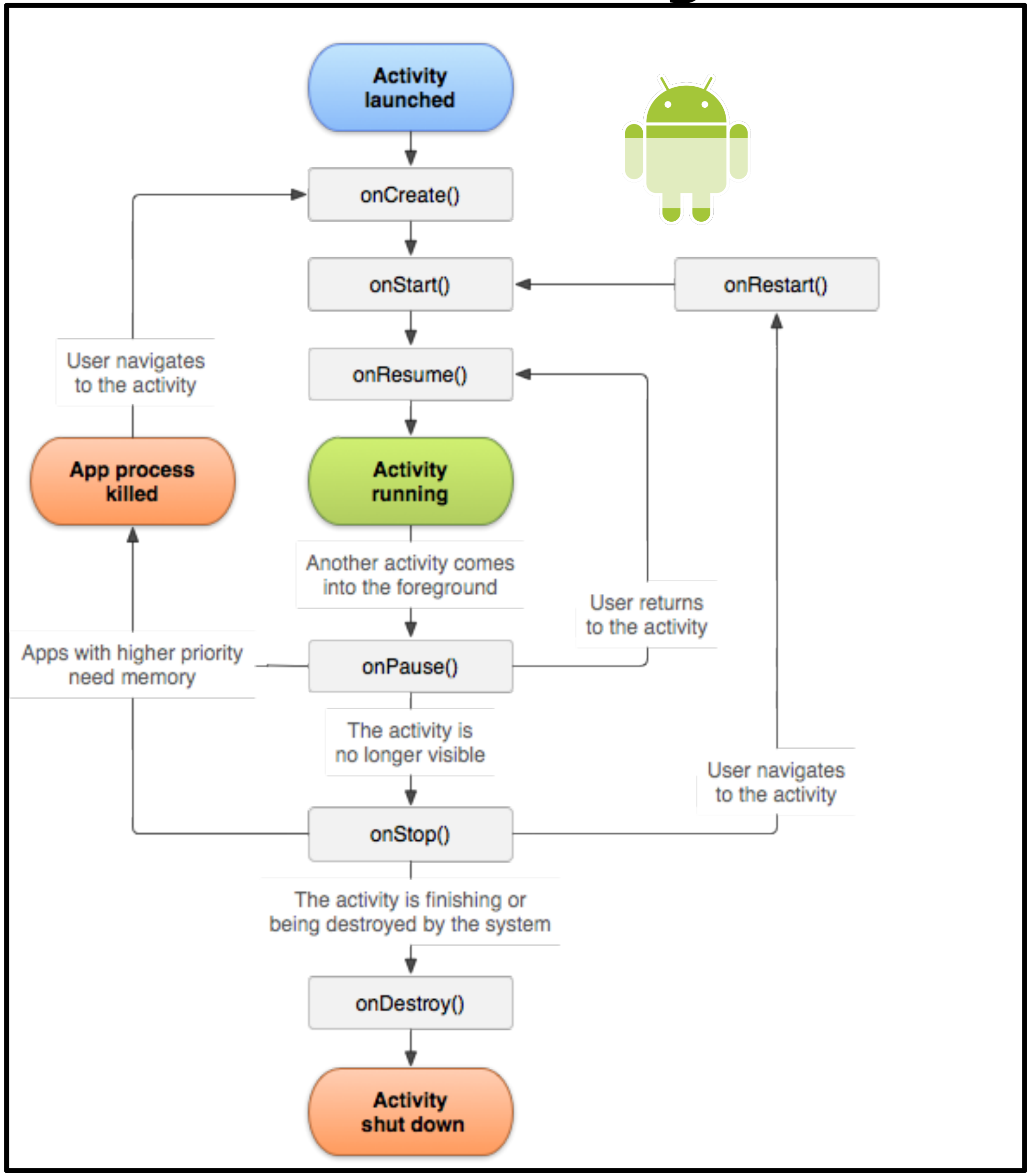


Process Lifecycle

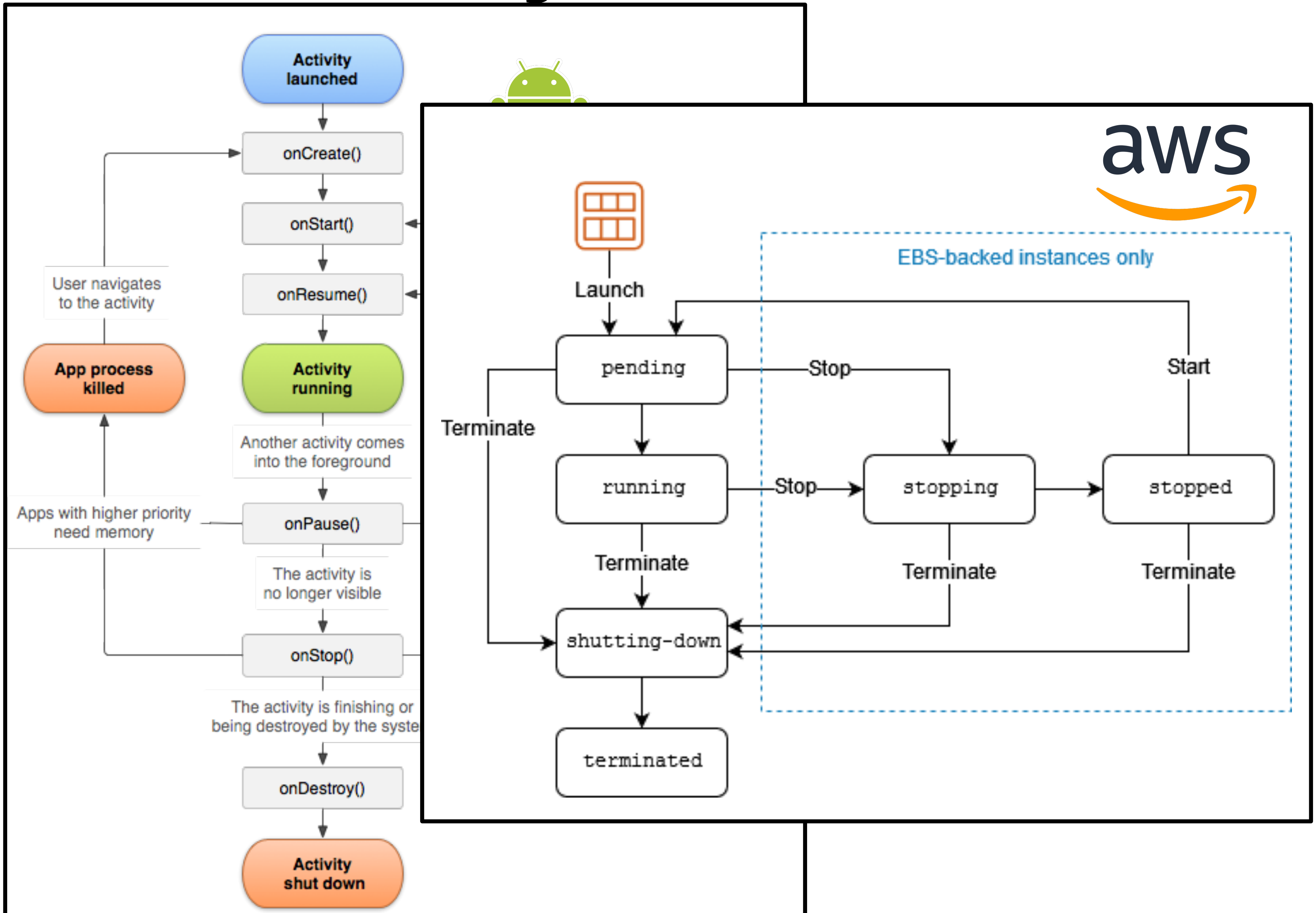
It's important that you have this diagram in mind

The narrative is straightforward: just practice drawing this diagram by telling yourself the story, not by memorizing it!

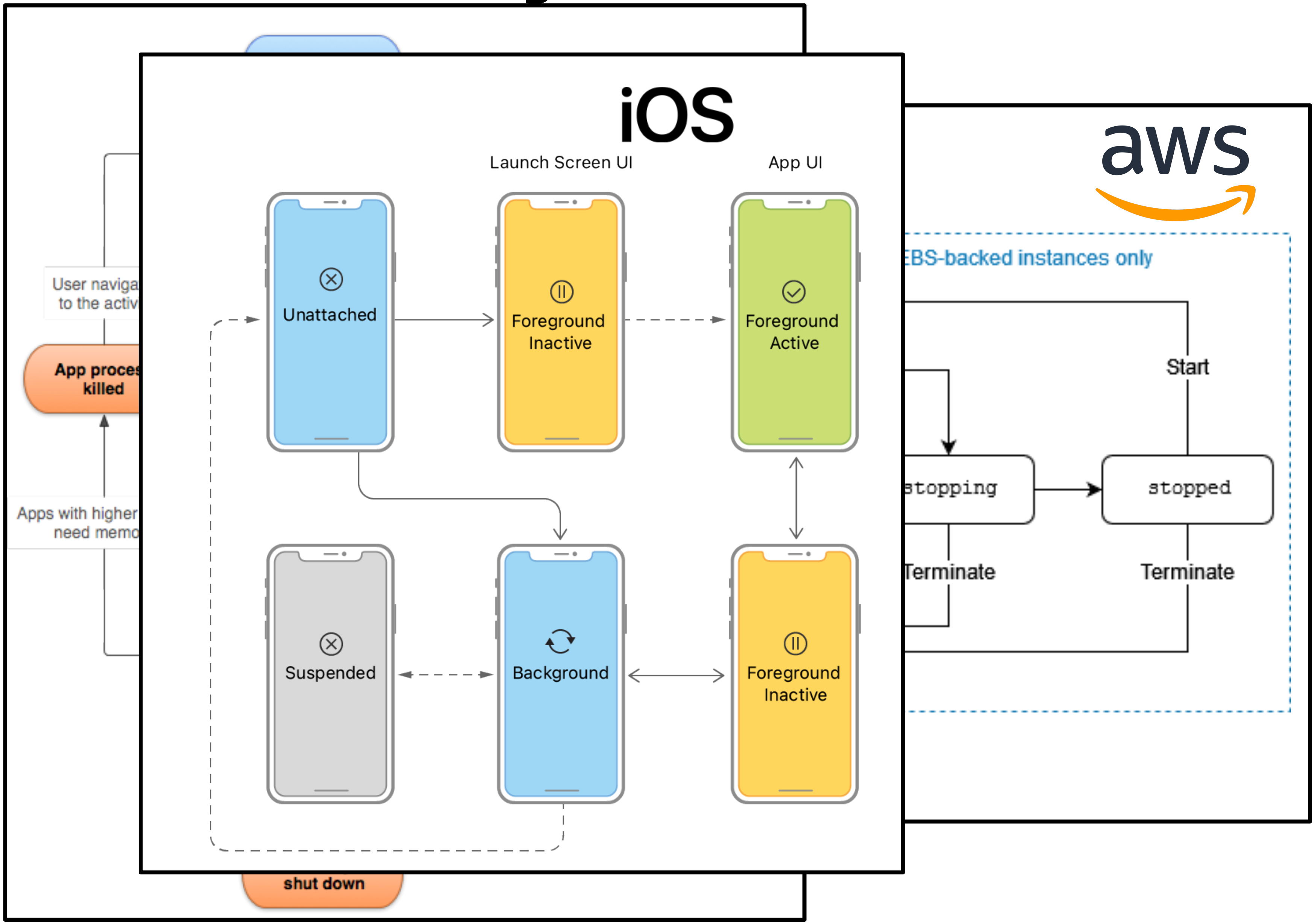
Other Lifecycles



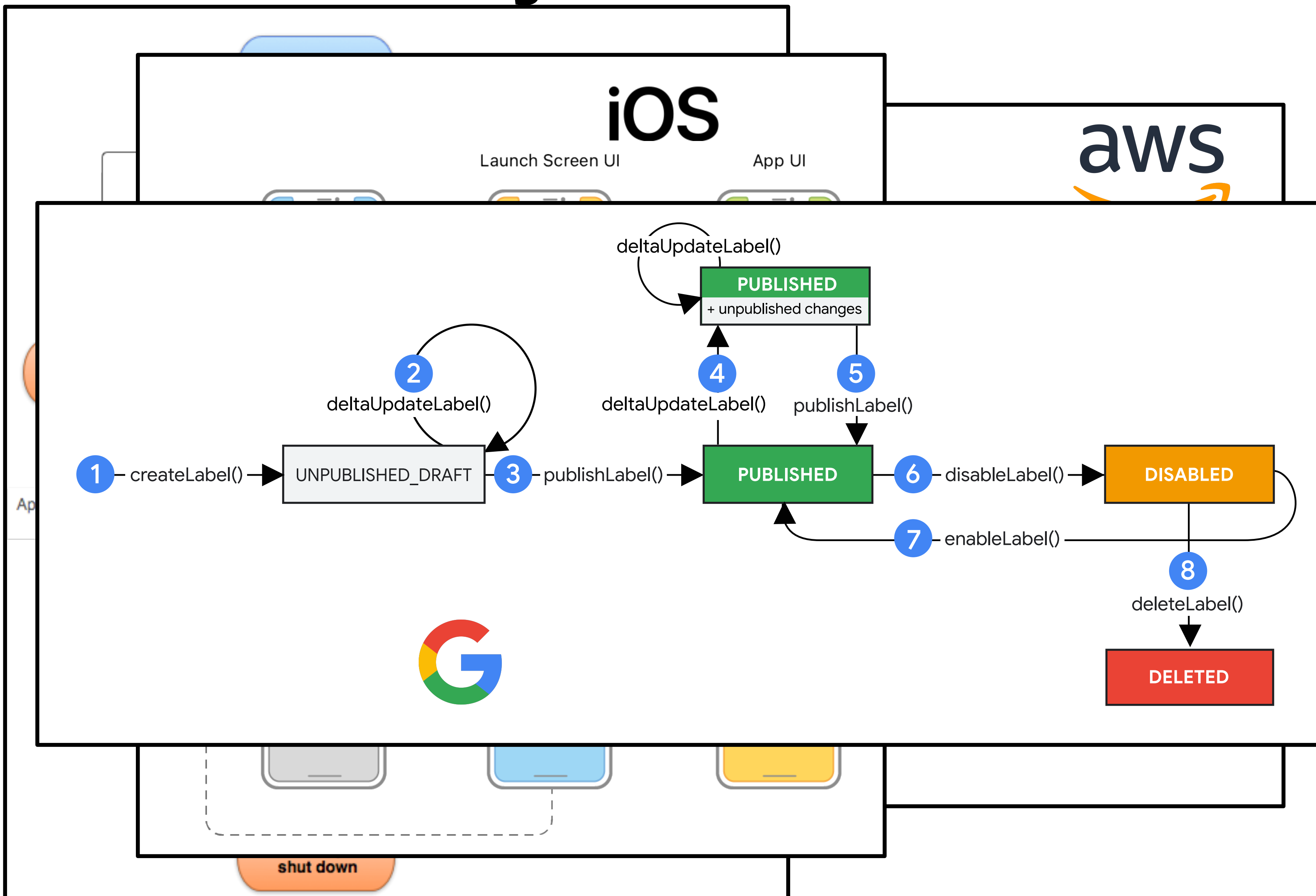
Other Lifecycles



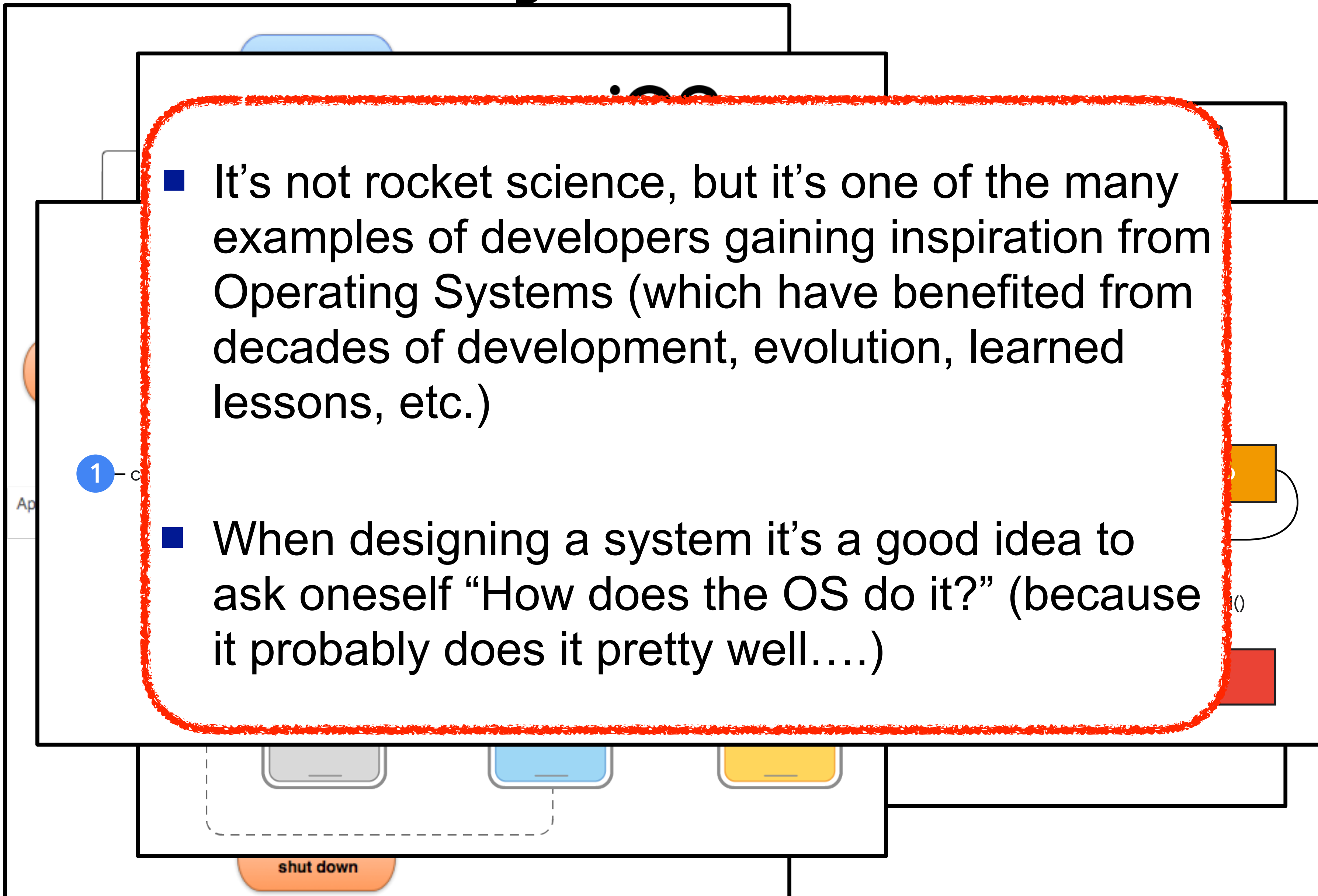
Other Lifecycles



Other Lifecycles



Other Lifecycles



- It’s not rocket science, but it’s one of the many examples of developers gaining inspiration from Operating Systems (which have benefited from decades of development, evolution, learned lessons, etc.)
- When designing a system it’s a good idea to ask oneself “How does the OS do it?” (because it probably does it pretty well….)

Process Control Block

- The OS uses a data structure to keep track of each process
- This structure is called the Process Control Block (PCB) and contains:
  - Process state
  - Process ID (aka PID)
  - User ID
  - Saved Register Values (include PC)
  - CPU-scheduling information (see “Scheduling” Module)
  - Memory-management information (see “Main Memory” and “Virtual Memory” modules)
  - Accounting information (amount of hardware resources used so far)
  - I/O Status Info (e.g., for open files)
  - … and a lot of other useful things
- Let’s look at Figure 4.5 in OSTEP (for the Educational xv6 kernel)
- Let’s look at the **`task_struct`** data structure in **`/usr/src/linux-headers-6.8.0-79/include/linux/sched.h`** (on our Docker image)

The Process Table

- The OS has in memory (in the Kernel space) one PCB per process
  - A new PCB is created each time a new process is created
  - A PCB is destroyed each time a process terminates
- The OS keeps a “list” of PCBs: the Process Table
- Because Kernel size (i.e., its memory footprint) is bounded, so is the Process Table
- Therefore, the Process Table can fill up!
- If you (or your program) keeps creating new processes, at some point, the process creation will fail
  - One of the many ways in which a system can become unusable
  - Because at that point you can’t even start a new Shell, since the Shell is a process!
- Anybody has heard of the “fork bomb” term?
- Let’s find out the max number of possible processes on our Docker container…
  - **`cat /proc/sys/kernel/threads-max`**

Main Takeways

- Processes are running programs
- Multiple processes co-exist in RAM
  - The old single-tasking MSDOS case
- The concept of a process address space
  - Code/text, heap, data, stack
- The Process Control Block
- The Process Table
- The Process lifecycle

- Onward to the Process API....

Conclusion

- We have all necessary concepts for processes here
- But how do programs actually create/manage processes?
- Onward to the Process API....