- Q1: What is a process?

- Strict POSIX definition: "An address space with one or more threads executing within."
- A process is an instance of a program in execution, with its own isolated access to system resources mediated by the OS and its own execution state.
    - A **program** is a recipe—e.g., instructions on how to bake a cake.
    - A **process** is all the mess you make in your kitchen while following the instructions on how to bake a cake, including all the resources you have to use.
- Simple programming model
    - Want to run a program? Need a process.
    - Want to run multiple programs? Need multiple processes.

- Q2: Why use a process?

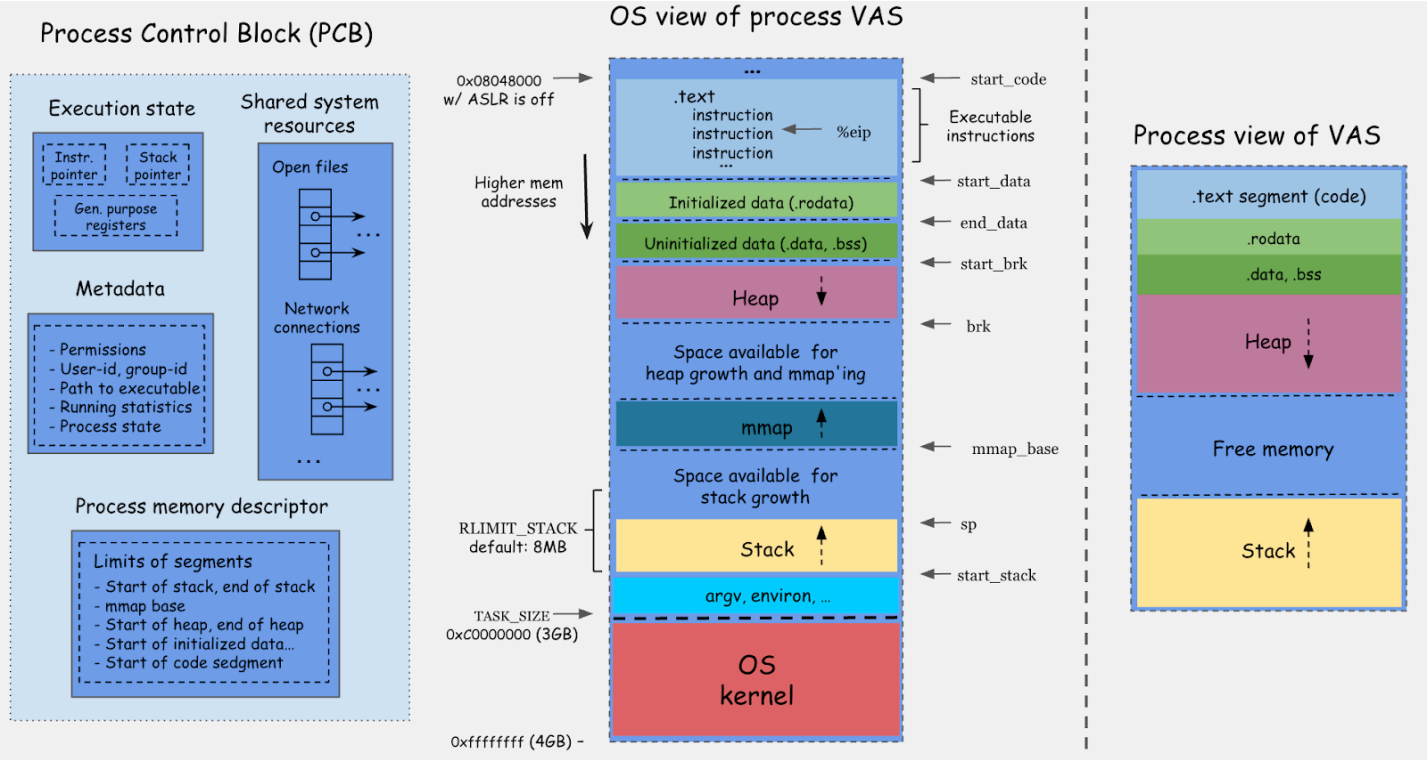
- Little responsibility - Life is good!
    - All the memory is available to user programs in a confined and isolated manner.
    - All system services on your disposal, as long as you request them properly.
    - All shared resources are seamlessly managed by the OS so that user programs have a "pleasant" experience.
- Helps **increase throughput:** Complete as many jobs as possible in a unit of time.
    - **How**? Increase utilization of shared resources.
    - **Multiprogramming:** Multiple processes reside in memory at the same time, and the processor switches between them when a process is waiting for I/O.

- Q3: What are the components of a process?

Based on what a process is (from Q1 above) as well as on what a program looks like in memory (from the previous chapter), the basic components of a process are as follows.

- **Virtual Address Space**
    - A linear array of bytes: [0, 2^32/64] with all static and dynamic segments in virtual memory. (static segments: .text segment / .rodata, .data, .bss data segments; dynamic segments: heap, stack, and mmap'ed segments.)
    - See also "How does a program look in memory?" (previous chapter).
- **Execution state**
    - An instruction pointer indicating the next instruction to be fetched from memory, and a stack pointer indicating the latest active procedure call.
    - A set of general-purpose registers with their values.
- **Shared systems resources**
    - Open files, network connections, and so on.
- **Process-specific metadata**
    - Credentials (user/group ID), permissions, running statistics (e.g., processor time consumed), process state (e.g., "ready" or "waiting").
- The OS maintains a struct—called Process Control Block (PCB)—to keep track of all the above, and much more. (e.g., see task_struct and mm_struct, in Linux.)

Overview of process components



- **Q4: How do we create a process?**
  - Approach-1: Cloning (e.g., POSIX **fork()**).
    - Duplicating and diverging address spaces.
    - Pause the current process and save its state.
    - Duplicate its PCB (can select what to copy and what not).
    - Add a new PCB to the ready queue.
    - Use the return value of fork() to distinguish parent and child processes.
    - Requires distinguished **init process**: The first user process is initiated by the kernel. All other user processes are its descendants.
  - Approach-2: Instantiating a process tabula rasa (Win32 **CreateProcess()**).
    - Load code and data into memory.
    - Create and initialize a new PCB.
    - Add a new PCB to the ready queue.
    - Does not require a distinguished process.
  - Approach-1 vs. -2
    - Elegant simplicity of the POSX interface: fork() takes no arguments vs. CreateProcess() requires a ton of different arguments.
    - The user program uses the return value of fork() to distinguish the flow of execution on the parent versus the child process. [Q: How to execute existing executables?]

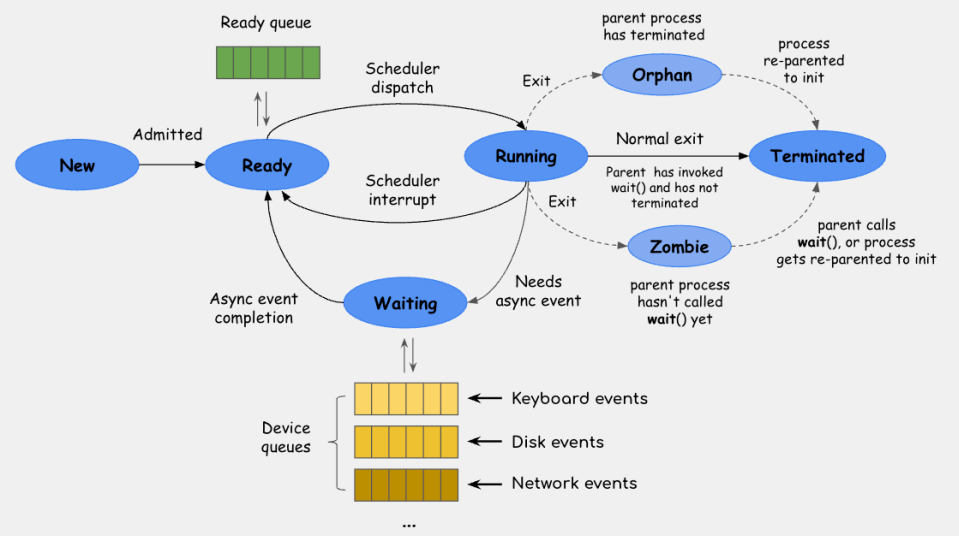
- Q5: How does a process terminate?

- Normal Termination
  - A process exits using the syscall exit(...).
  - The OS passes the exit status to the parent via the syscall wait(...).
  - The OS then frees the resources of the exited process.
- Abnormal Termination
  - A process is forcibly terminated by the OS or by another process. (Will talk about signals and process communication later.)
  - **Zombie Process**: This occurs when a child exits before its parent invokes the wait(...) to collect the child's exit status.
    - The OS retains the exit status until the parent calls wait().
    - The process is not fully removed until reaped.
  - **Orphan Process**: This occurs when a parent exits before their child.
    - The child gets re-parented to the init process, which manages its termination.

- Q6: What are all the possible states a process could be in during its lifetime?

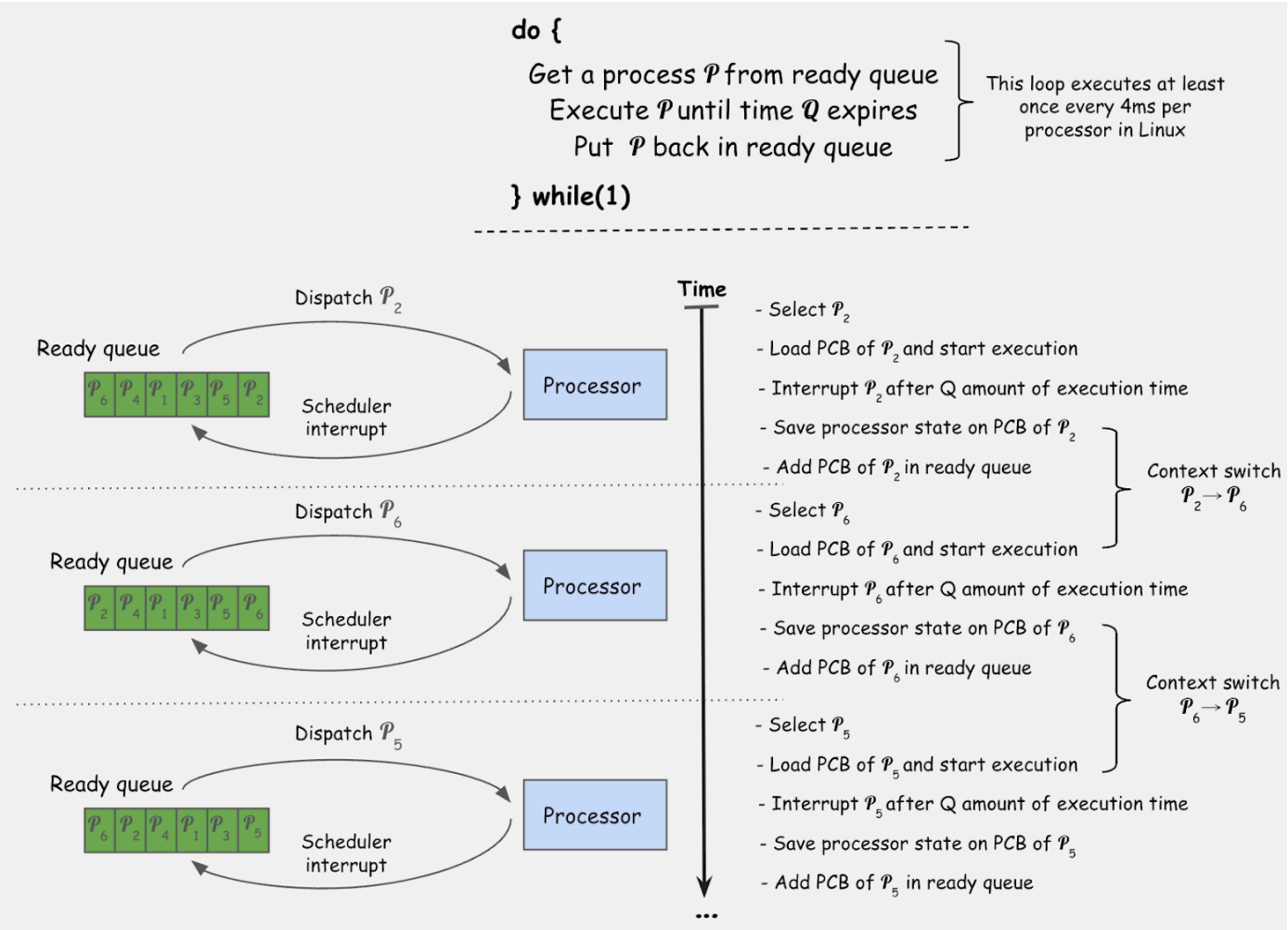
- A POSIX process has an execution state that indicates what it is currently doing.
- Each process' PCB is queued on the respective queue, accordingly.
- As a process executes, it moves from state to state.
  - **Ready**: A process which is ready to be executed—i.e., ready to be scheduled in ("dispatched"), but it's waiting because another process is using the processor.
  - **Waiting** (blocked): A process is blocked waiting for an async event to complete—e.g., a disk I/O—and cannot make progress until the event completes.
  - **Running**: A process which is executing on the processor until either (i) an async event is required and the process transitions to the "waiting" queue; or (ii) it exceeds its maximum quantum of processor execution time, a scheduler interrupt occurs, and it transitions to the "ready" queue.
  - **Terminated** (normally)**:** A process which finished execution normally (either via calling exit or by returning from its main) and its parent has called wait to collect the exit status. Otherwise
    - The child process becomes a **zombie,** if the parent still exists but hasn't called wait() yet.
    - Or, the child process becomes an **orphan**, if the parent has exited already.

Process states, and state queues diagram



- **Q7: How does the OS run multiple processes simultaneously?**
  - The goal is to run multiple processes simultaneously, giving each process the illusion that it has full and exclusive access to all the available hardware resources.
  - We'll talk about scheduling in detail later, but for now, assume multiprogramming (i.e., many processes in memory, one allocated on the processor) and a simple timesharing dispatching loop implemented with hardware support for preemption (i.e., periodic timer interrupts handled by the scheduler).
    - **while (1) {**
      run each process for a bit
      }
  - Switching between processes—known as context switching (a context is a mapping of virtual to physical addresses)—is tricky and architecture-dependent
    - Need to save process execution state (registers) to the process' PCB.
    - Run code to save general-purpose registers actually changes registers.
    - Use hardware support (e.g., pushad/ popad on x86_32) or use stack (x86_64).
    - Must balance context switch frequency with scheduling requirement
      - Saving and restoring many things repeatedly is expensive
      - A bigger problem exists. [Context switch requires flushing the TLB.]

Process dispatching loop (timesharing)



- **Q8: What are the steps involved in process creation and process switching in Linux?**
  - Linux uses a neutral term for processes (and threads) called tasks.
  - The implementation semantics respect POSIX expectations for processes.
  - Each process in Linux has two stacks, a user and a kernel stack (default: 8KB).
    - The kernel stack can only be accessed in kernel mode.
    - Interrupt and exception handlers run on the kernel stack because the user stack cannot be trusted. [Recall all the drama with stack switching.]

    [Q: Since switching address spaces is costly, how are we avoiding this overhead when entering kernel mode from user mode?]

  - **Fork on Linux x86**

    SYSCALL_DEFINE0(fork)
    kernel_clone(...)
    copy_process(...)
    dup_task_struct(...)
    arch_dup_task_struct(...)
    copy_*(...)
    wake_up_new_task()

  - **Context switch on Linux x86**

    schedule(...) [We will talk about scheduling later.]
    __schedule(...)
    pick_next_task(...)
    context_switch()
    switch_to(...)
    finish_task_switch(...)