THE PROCESS ABSTRACTION

CS124 – Operating Systems

Spring 2024, Lecture 6

The Process Abstraction

- Most modern OSes include the notion of a **process**
  - Term is short for a "**sequential process**"
  - Frequently described as "an instance of a program in execution"
  - The primary means by which multiprogramming / multitasking is offered in modern operating systems
- A process generally consists of:
  - The program's instructions (aka. the "program text")
  - CPU state for the process (program counter, registers, flags, …)
  - Memory state for the process
  - Other resources being used by the process
- A primary task of the OS is to manage processes
  - Maintain the illusion of multiple processes executing concurrently by giving each process a portion of time on the CPU
  - Handle other stages of the process lifecycle

The Process Lifecycle

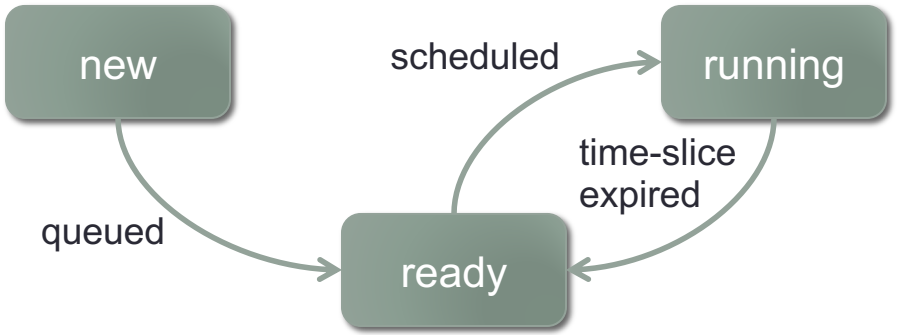
- Process lifecycle can be represented as a state diagram
- Processes must be created before they can run (duh)
- Example origins:
  - Created by the operating system at startup (e.g. Linux: **`systemd`**)
  - Created when a user invokes a program via command line or GUI
  - Created when a process spawns another process

The Process Lifecycle (2)

- Generally, there is a set of processes that can make progress (i.e. not waiting for I/O to complete, etc.)
- Only one process may be running on each CPU at a time
- When a process is in “running” state, it holds the CPU!
- Other processes that could run, but don’t currently have the CPU, are in the “ready” state

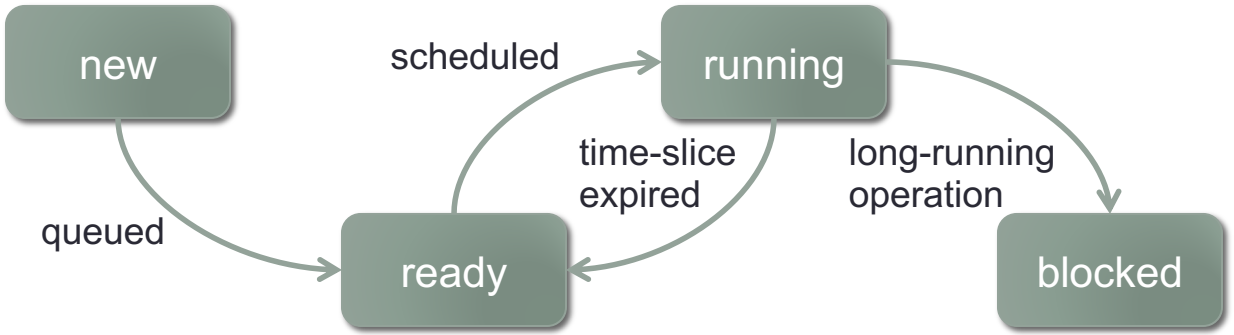
The Process Lifecycle (3)

- New processes don't necessarily get the CPU right away
  - They initially go into the collection of "ready" processes
- The OS only allows the currently running process to hold the CPU for a specific amount of time…
- When time-slice expires, running process is preempted, and the OS chooses another process to get the CPU



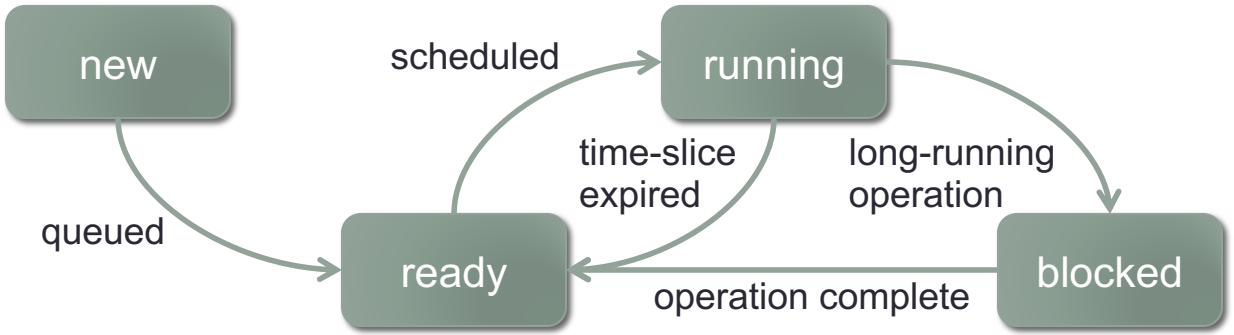
The Process Lifecycle (4)

- Processes often perform long-running tasks
  - e.g. read from hard disk, network, or some other external device
  - e.g. process waits for another process (e.g. a signal or termination)
  - The process becomes **blocked** until the resource is available
- Instead of holding everyone up, kernel removes process from CPU, and chooses another ready process to run



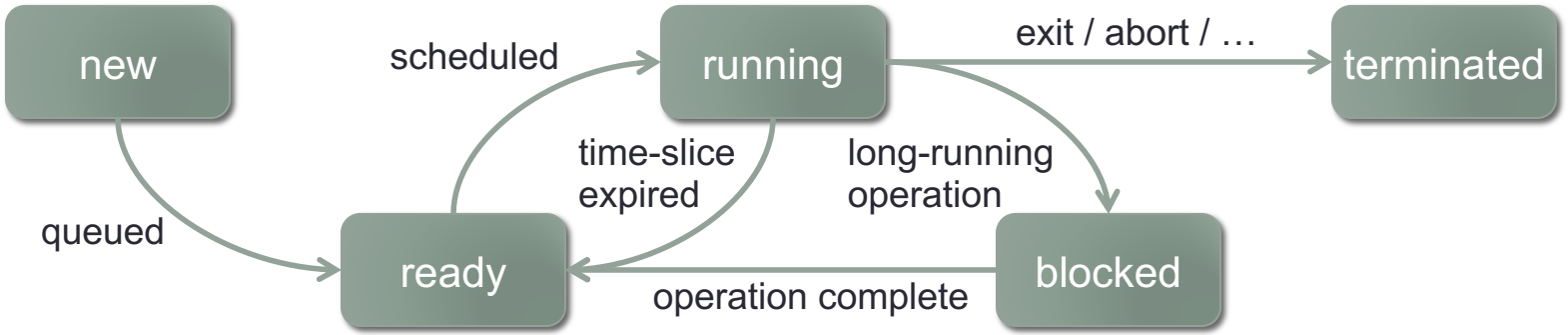
The Process Lifecycle (5)

- When the long-running task is completed, the blocked process can resume execution
  - Process is moved back into the ready state
  - Will eventually be chosen by the OS to run on the CPU again



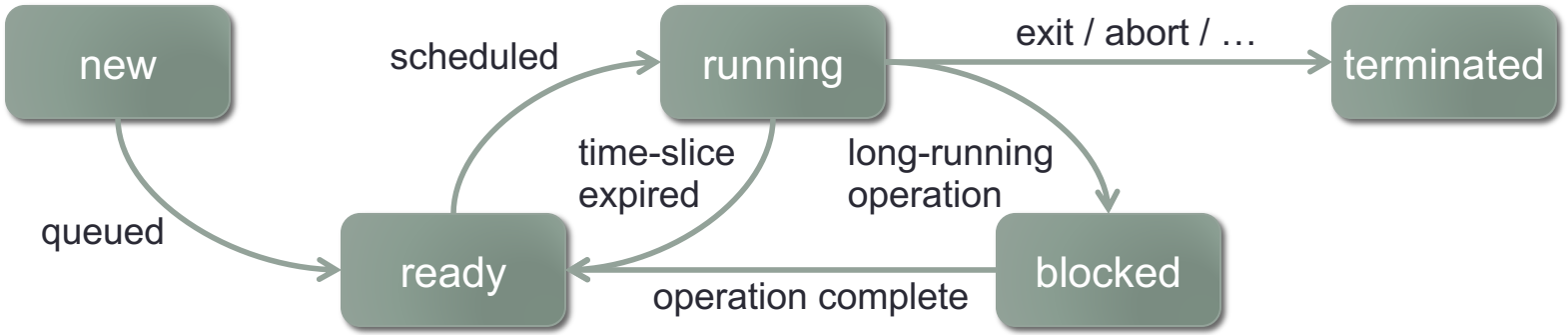
The Process Lifecycle (6)

- Processes eventually terminate
  - May live for seconds, hours, months, …
- Several tasks must be completed at process termination
  - Any “at-exit” operations must be performed
  - Reclaim resources the process is still holding
  - Other processes may need to observe terminating process’ status



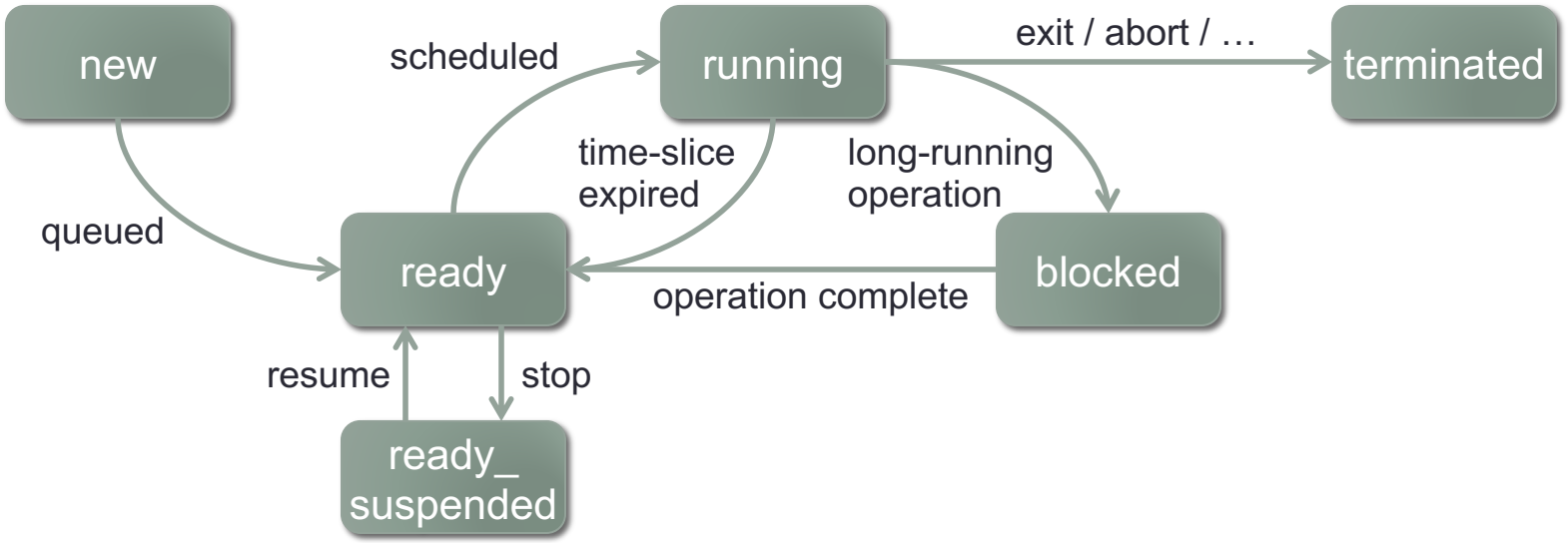
The Process Lifecycle (7)

- Processes can terminate for several different reasons
  - Voluntary termination by the process itself (e.g. it calls **`exit()`** or returns from **`main()`**, either with success or error status
  - Involuntary termination due to an unrecoverable fault in the process (e.g. segmentation fault due to dereferencing a **`NULL`** pointer)
  - Involuntary termination due to a signal from another process (e.g. another process issues a **`SIGINT`** (^C), **`SIGTERM`** or **`SIGKILL`**)



The Process Lifecycle (8)

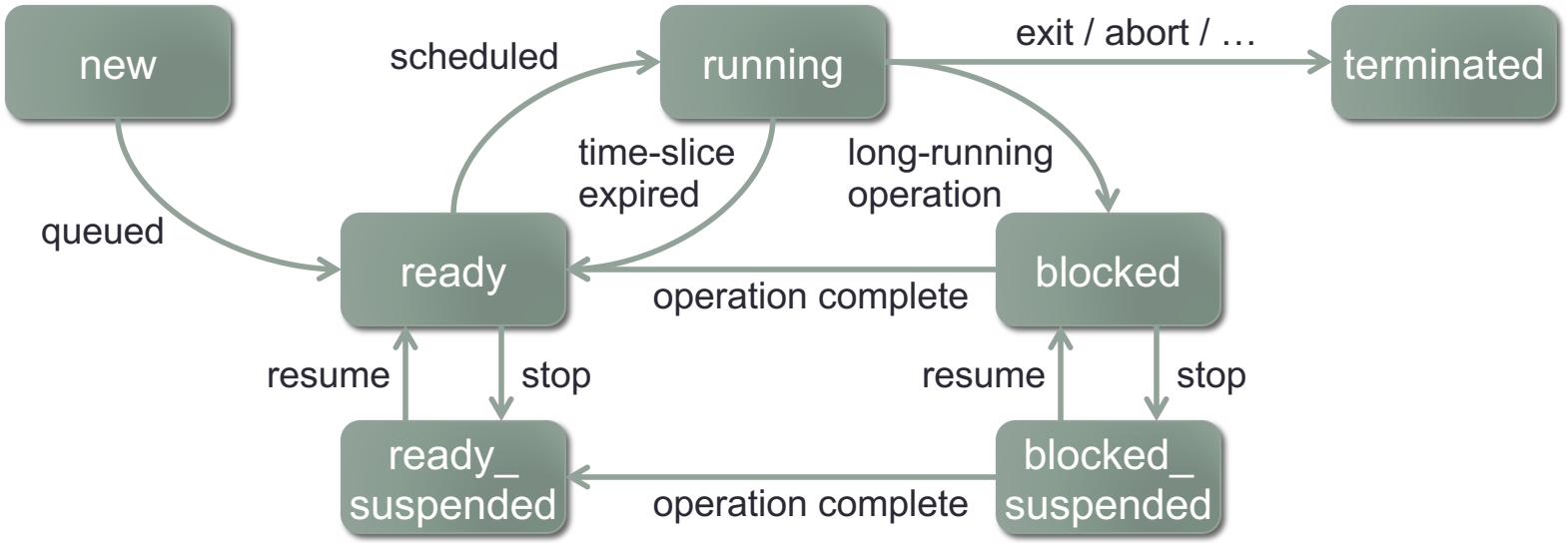
- This state diagram is sufficient for most OS needs…
- Frequently, other states are also introduced to provide additional capabilities
- Common feature: ability to suspend / resume processes
  - A suspended process will not be scheduled until it is resumed
  - A user can suspend a process with e.g. Ctrl-Z at command shell
  - A process can send **`SIGSTOP`** to another process to suspend it



The Process Lifecycle (9)

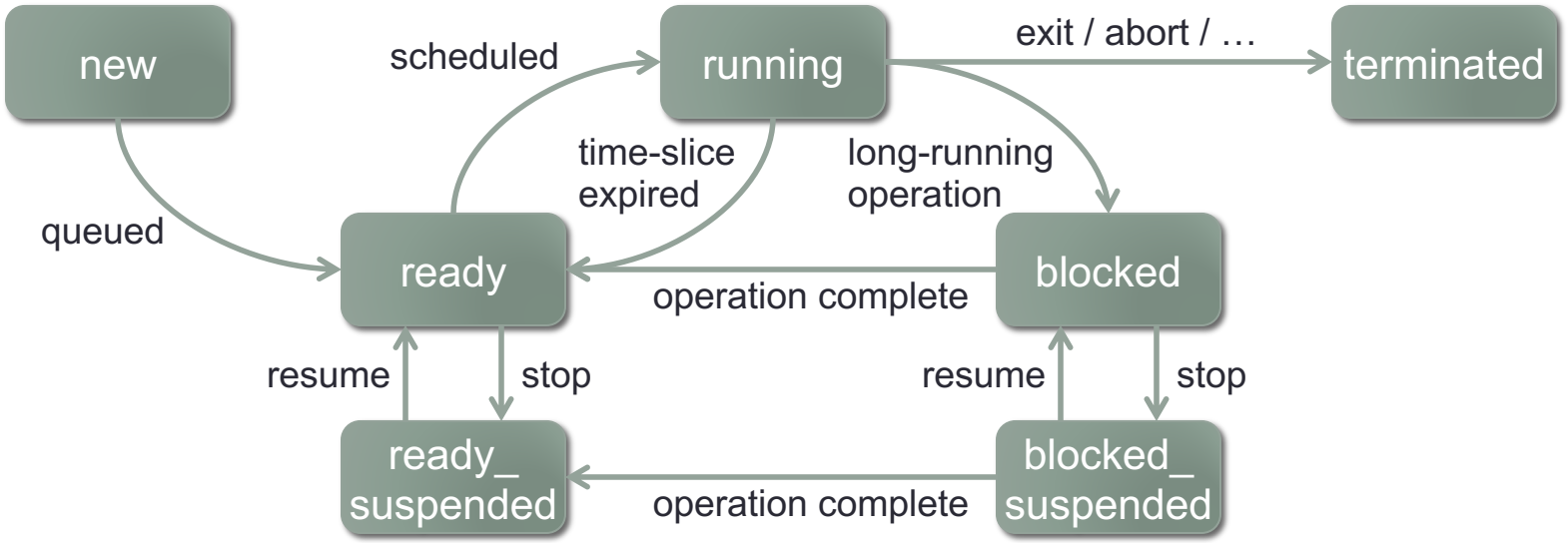
- The process being suspended might also have been blocked on a long-running operation…
  - Introduce another state to manage such processes

- Final process lifecycle:



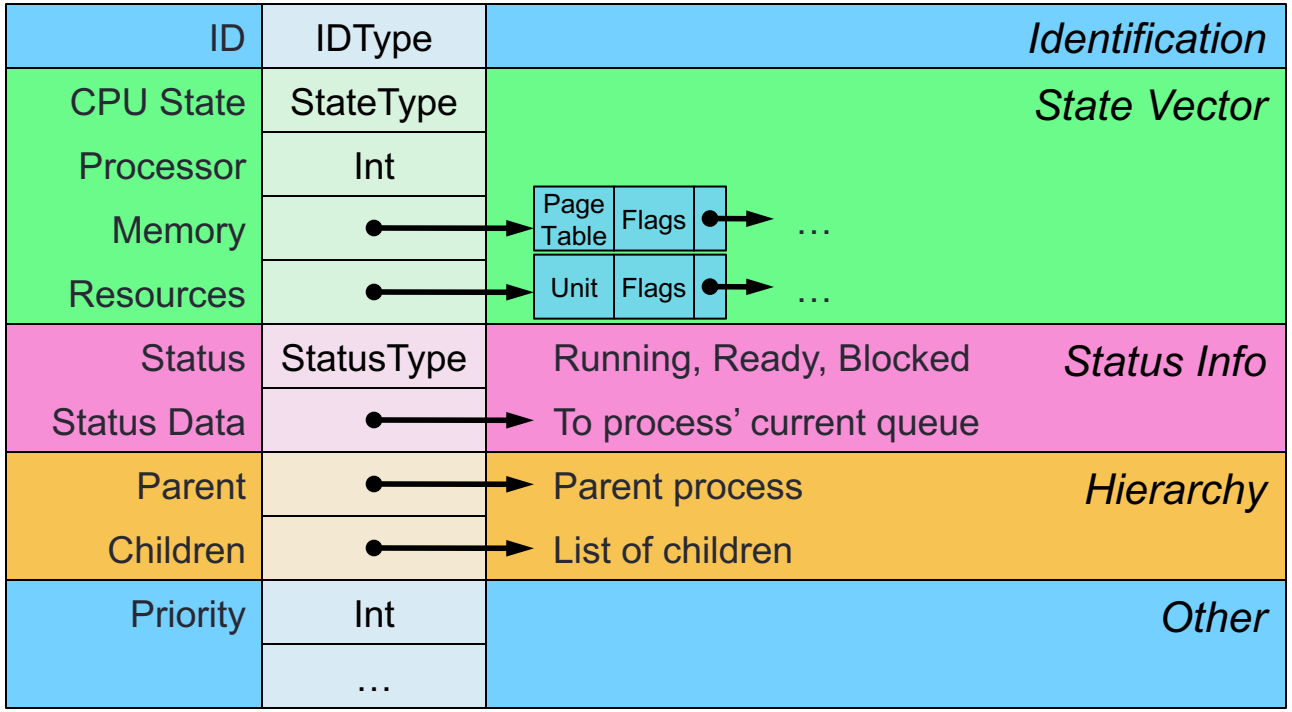
Implementation Questions

- How are a process' resources managed and reclaimed?
- How are blocked processes managed by the OS?
- How do we switch what process is currently running?
  - i.e. how do we perform a **context switch**?
- How does the OS choose what process should run next?
  - i.e. how does **process scheduling** work?



Process Control Block

- Process-specific details are managed in a data structure often called the **process control block** (PCB)
  - Also called a task control block, a task struct (Linux), etc.
- A representative example:



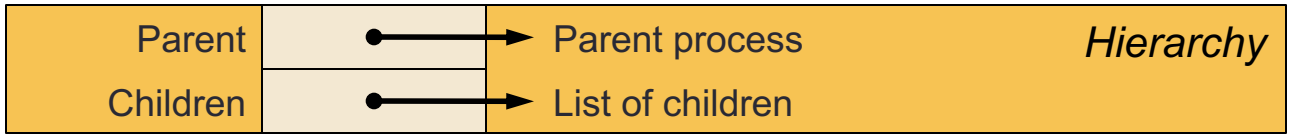
Process Identification

- The kernel manages a mapping of Process IDs to Process Control Blocks
  - Identification information uniquely identifies the process

- Several options for mapping PIDs to PCBs
  - Linux uses a hashtable, with bins containing linked-lists of PCBs
  - Rationale:
    - More space-efficient than a table where PIDs are indexes
    - Expect that actual process-count will typically be *much* smaller than the system limits

Process Hierarchy

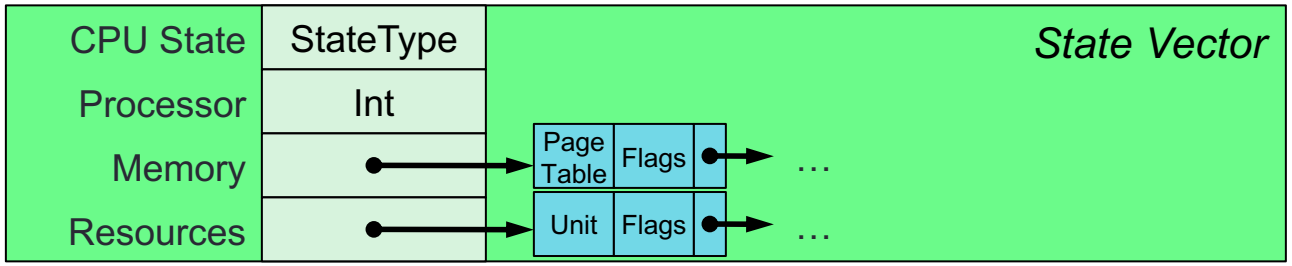
- The UNIX process model also includes process hierarchy



- Every process has a parent
  - Not possible for a child process to disown its parent
  - (init becomes the parent process of orphaned children)
- Every process has zero or more children it has started, as well as a process-group it is a member of
- Not all operating systems provide a process hierarchy
- Windows effectively treats all processes as peers
  - Processes can be grouped, and a process can spawn children
  - No real relationship between parent and children is imposed by Windows process model

Process State Vector

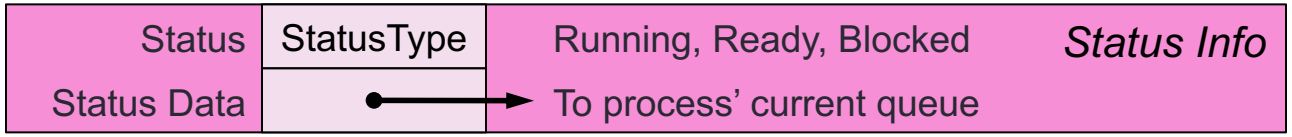
- State vector specifies all process context information



- CPU state:
  - Process capabilities and protection info
  - When suspended, includes program counter + register contents
  - Depends on processor architecture
- Processor:
  - Set to CPU number when running; otherwise undefined
- Memory:
  - Contents of process' code, data, stack, etc.
  - (Heavily leverages virtual memory system)
- Resources:
  - All allocated resources (files, network sockets, etc.)
  - Resource class + unit descriptions

Process Status Information

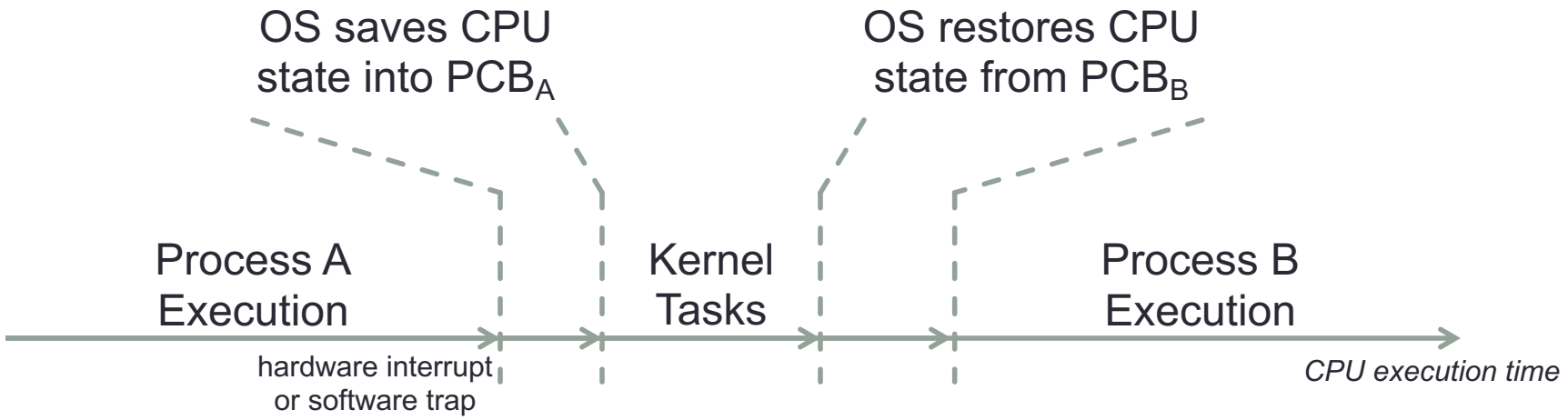
- Process control block also includes scheduling details



- Running:
  - Process is currently running on a CPU
- Ready:
  - Process is ready to run, but waiting for a CPU
- Blocked:
  - Process cannot proceed until it receives a resource or a message
- Also includes other states, e.g. Suspended, etc.
- Status data can be used for:
  - Specifying pending resource-requests for this process
  - Specifying other processes in the same state and priority

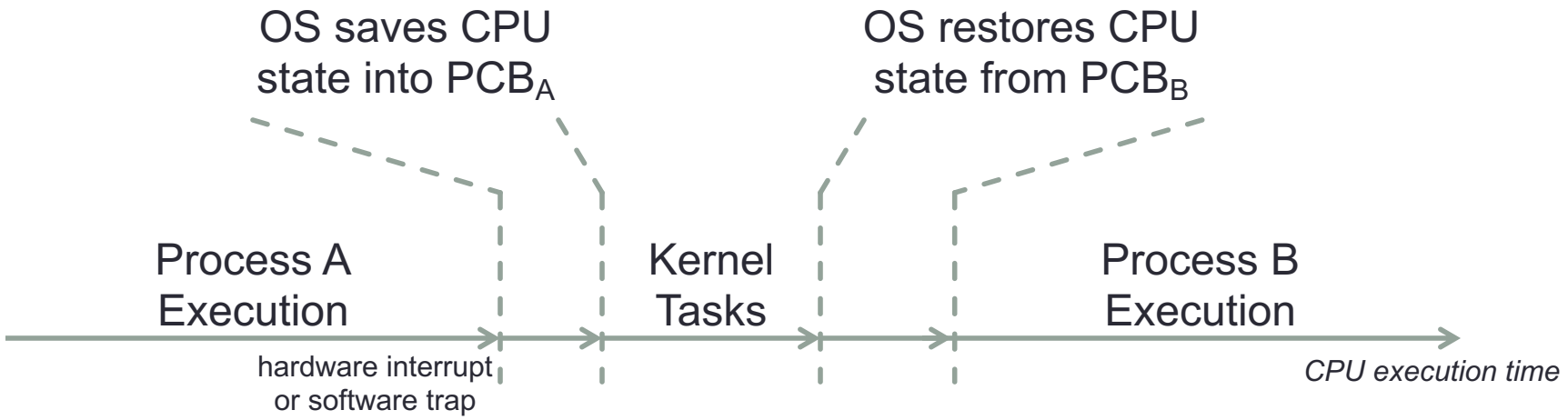
Process Context-Switch

- When the OS switches from running a given process, the process' context must be saved into the process' PCB
  - CPU state: registers, program counter, stack pointer, status flags
  - (Other process state is usually already recorded in the PCB)
- Similarly, when the OS switches to another process, the new process' context must be restored from the PCB
- (Will cover more details when we get to scheduling…)



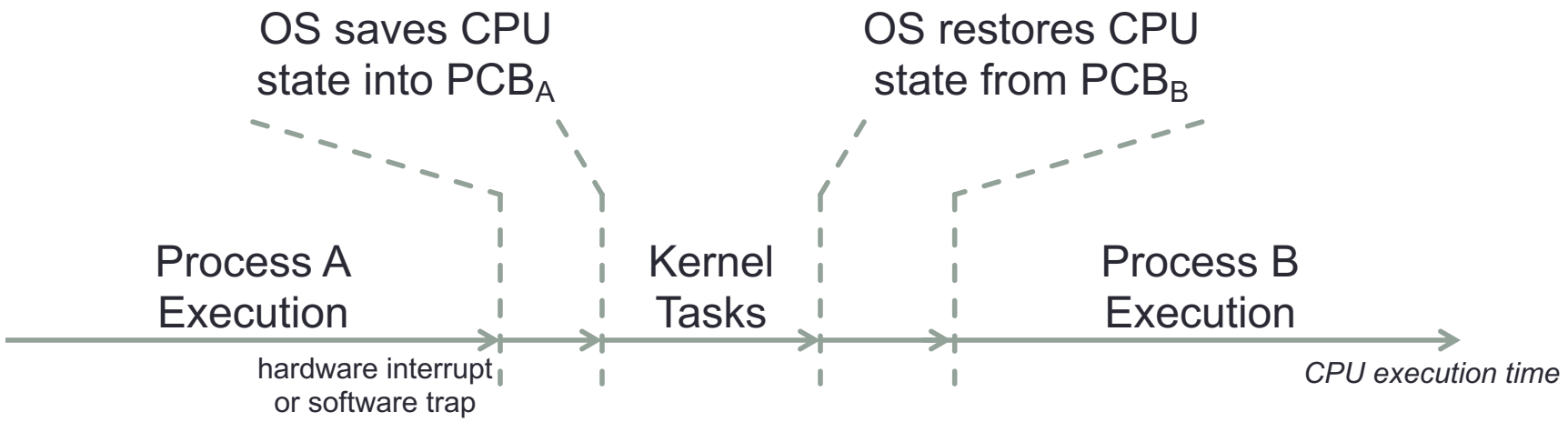
Process Context-Switch (2)

- Context-switches clearly require a certain amount of time
- Entering into the kernel:
  - CPU handles the interrupt (save program counter/stack, stack-switch)
  - Handler saves CPU state of current process into the process' PCB
- Kernel often has to invoke the scheduler in order to choose what process to execute next
  - Some syscalls don't cause a context-switch, but most tend to…



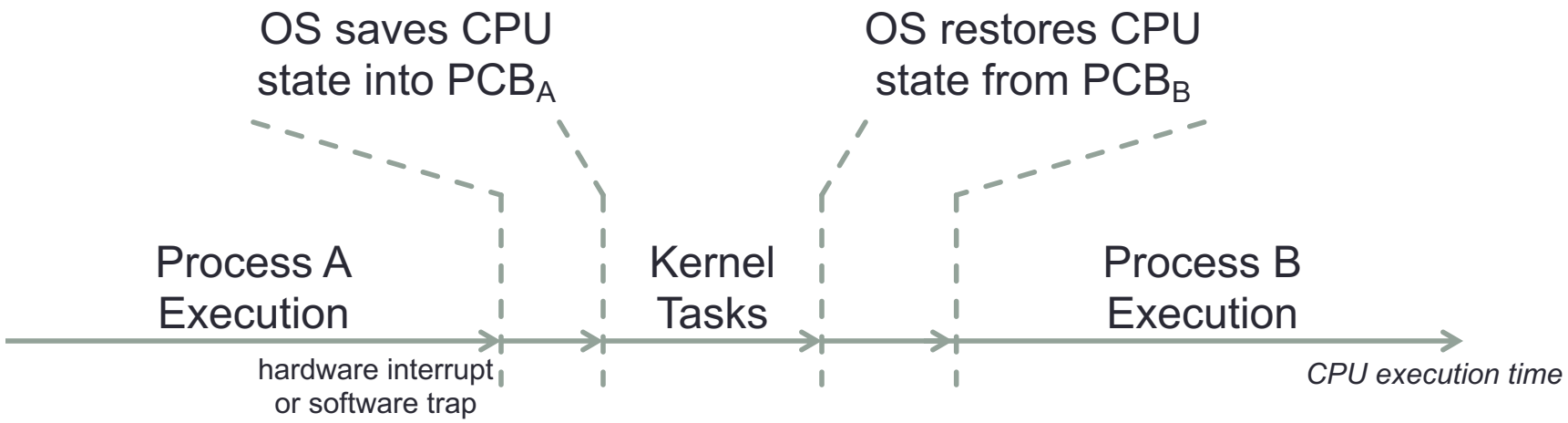
Process Context-Switch (3)

- Leaving the kernel:
  - Kernel must restore CPU state from new process' PCB
- Kernel must also switch to new process' memory state
  - Each process has its own page-table hierarchy in its own PCB
  - Must switch the virtual memory system to using the new process' memory mapping
  - (e.g. on IA32, load `%cr3` register with new process' page table)



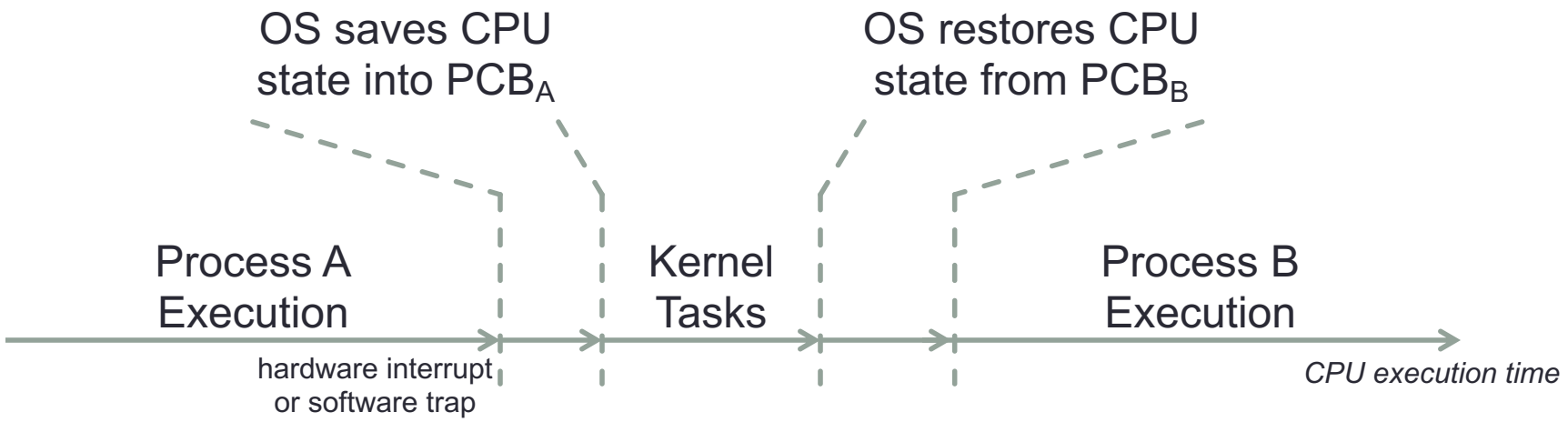
Process Context-Switch (4)

- When kernel changes the memory mapping, it must also clear the MMU's Translation Lookaside Buffers
  - They cache page-table entries for the MMU…
  - …but we just changed the mapping, so those are now invalid.



Process Context-Switch (5)

- During a context-switch, the OS isn't doing useful work
  - By "useful work," we mean "running the user's applications"
- Want to minimize amount of time a context-switch takes
  - e.g. make the scheduler fast, save/load CPU state fast, etc.
  - (Still, context-switches will take some amount of time…)
- Also want to minimize the frequency of context-switches
  - If system performs many context-switches, it will be spending less time doing useful work



Ready and Blocked Processes

- The OS must manage multiple collections of processes
  - Frequently implemented as **queues**: [doubly] linked lists, where elements hold pointers to the relevant Process Control Blocks
- Example: the **run-queue** or **ready queue** holds all processes that are able to execute, but don't have a CPU
- The implementation is often *much* more complex than a simple linked list…
  - Don't want the kernel to have to consider every single ready process when deciding who should run next
  - Frequently, schedulers manage sophisticated data structures to drive scheduling decisions
- Even with complex process-management data structures, collection of ready processes is still called the "run-queue"

Ready and Blocked Processes (2)

- Processes frequently block on long-running operations
  - e.g. read data from a file on disk/CD-ROM/flash drive/etc.
  - e.g. read data from a network socket
  - e.g. wait for another process to terminate
  - Need to remove such processes from the ready queue and put them into a collection of blocked processes
- Blocked processes usually become unblocked in interrupt handlers
  - e.g. the disk controller signals that a read is complete
  - Interrupt handler needs to be short, simple and fast to keep from holding up other parts of the system
- Again, OSes don't maintain just one collection of all the blocked processes

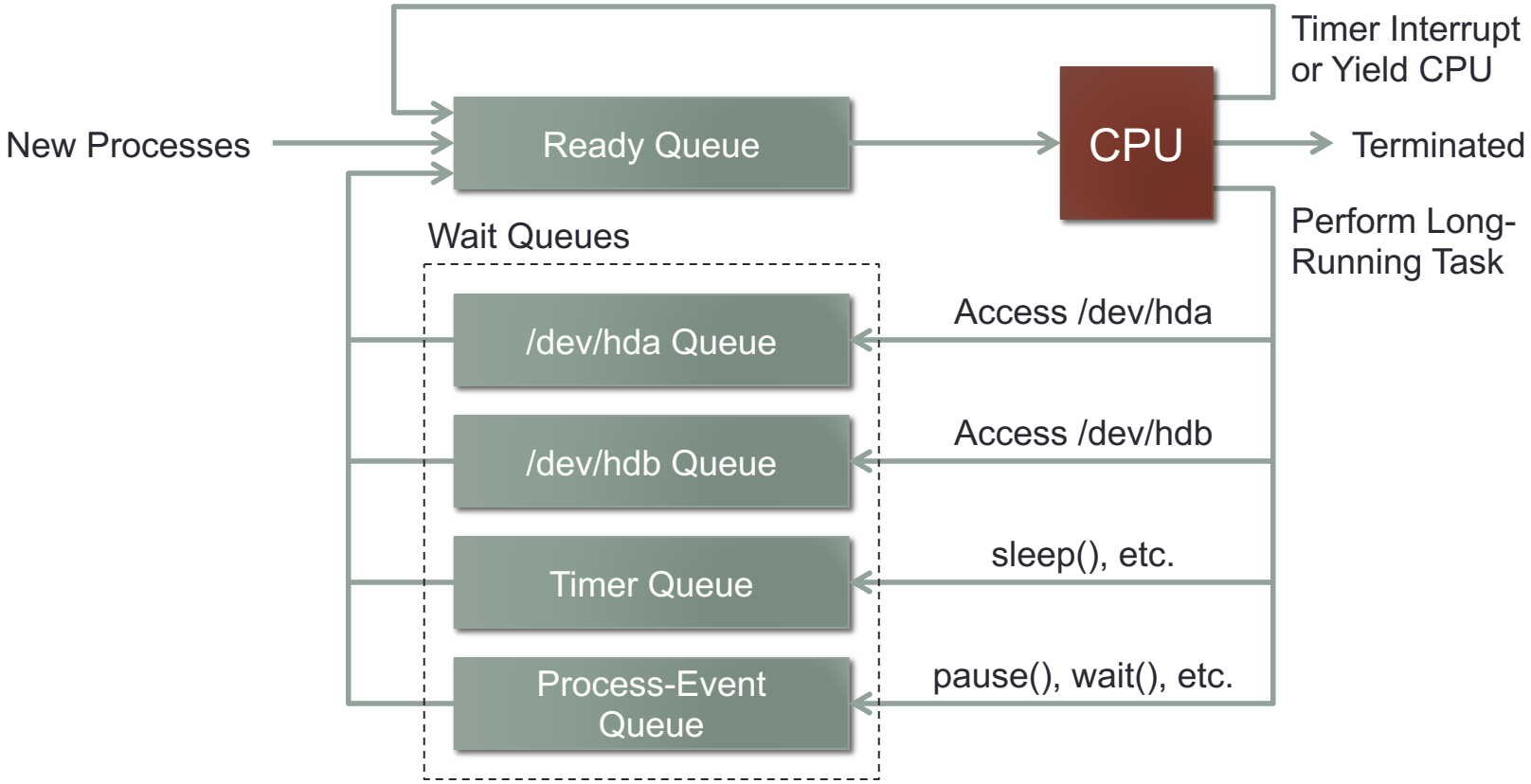
Ready and Blocked Processes (3)

- Operating systems maintain a **wait queue** for each device that processes can become blocked on
  - When a process requests data from a device, kernel initiates the task, then moves the process into the wait-queue for that device
  - Later, when the device fires an interrupt, the handler can easily access the processes actually waiting on that device
- Can also implement wait queues for processes waiting for signals, waiting for child-process termination, etc.

- A simple mechanism for conditional-wait operations
  - i.e. when a process cannot progress until a condition becomes true

Process Scheduling: Overview

- Example "**queuing diagram**" of ready queue, wait queues



Process Scheduling: Overview (2)

- Process scheduling is a large and complex topic
  - Will spend more time on it in the future; for now, general approach
- Scheduling can be applied at several different levels
- **Long-term scheduling** (a.k.a. **job scheduling**):
  - Often used in batch-processing systems that receive far more jobs than they can actually execute at one time
  - Jobs are **spooled** onto external storage for eventual execution by the batch system
    - "SPOOL" = Simultaneous Peripheral Operations On-Line
  - Long-term scheduler must choose which jobs to bring into memory for execution, and when
- Long-term scheduler tries to maximize the utilization of the batch-processing system's hardware

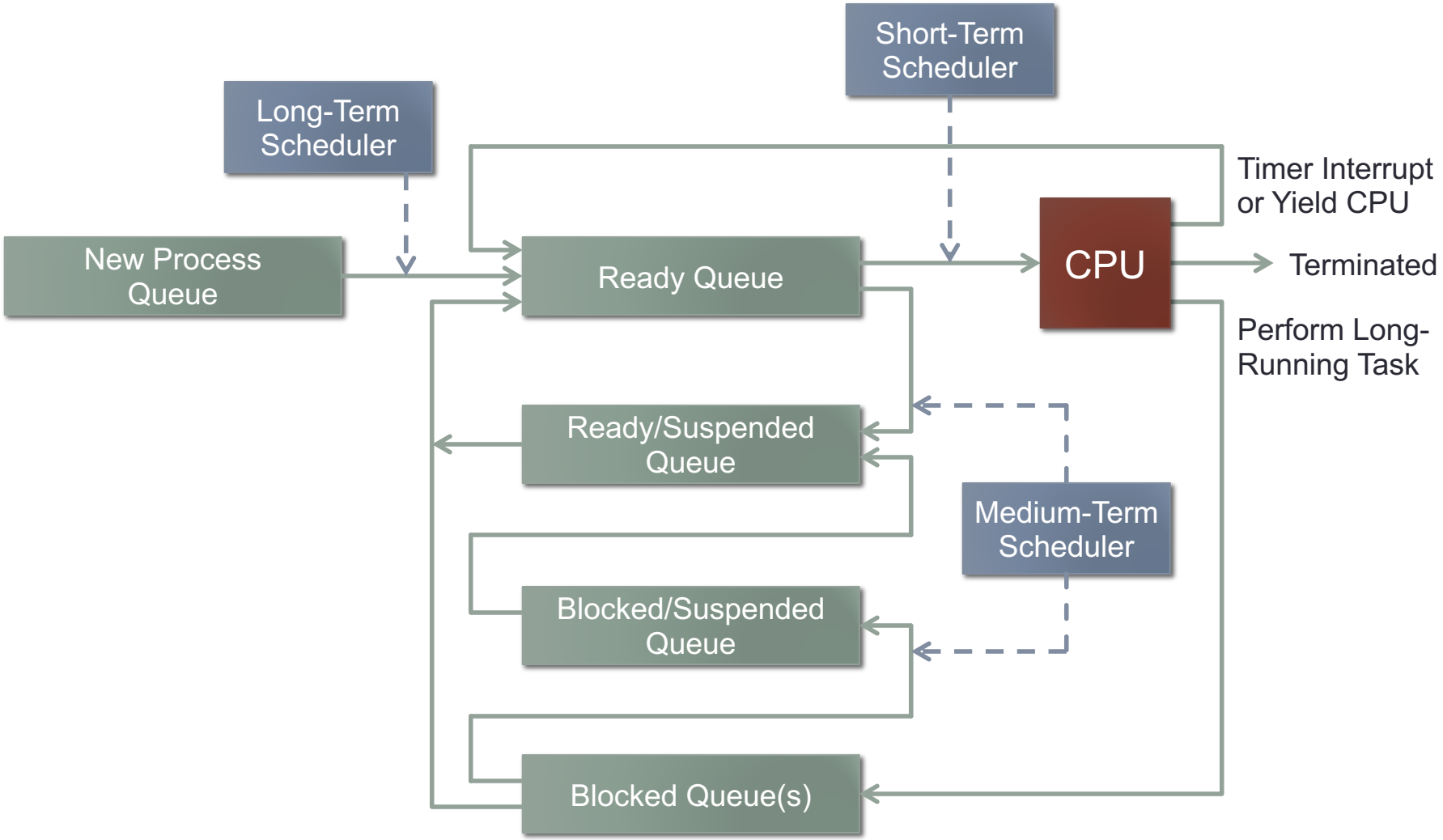
Process Scheduling: Overview (2)

- If all processes in batch-processing system are I/O bound:
  - Processes will usually be waiting for devices to respond…
  - Scheduler's run-queue will be empty. CPU will be underutilized.
- If all processes are CPU bound:
  - The peripheral devices will be underutilized.
- Long-term scheduler tries to achieve a good **process mix** of I/O-bound and CPU-bound processes
  - CPU-bound processes can effectively use the CPU while I/O-bound processes are waiting for devices to respond
- Long-term schedulers run infrequently
  - (e.g. on the order of minutes between invocations)
  - Primarily needs to run when processes terminate, so that the OS can determine what job(s) to begin executing next

Process Scheduling: Overview (3)

- Most general-purpose operating systems don't include a long-term scheduler
  - The user controls how many processes are running, not the OS
  - If OS performance is substandard, the user runs fewer processes
- All multitasking operating systems include a **short-term scheduler** to control which process gets the CPU next
- Some systems include **medium-term schedulers** to tune the process mix on an ongoing basis
- Medium-term schedulers can temporarily suspend and swap out processes when needed
  - e.g. if a process' resource requirements are negatively impacting the overall system performance

Process Scheduling: Overview (4)



Next Time

- The thread abstraction