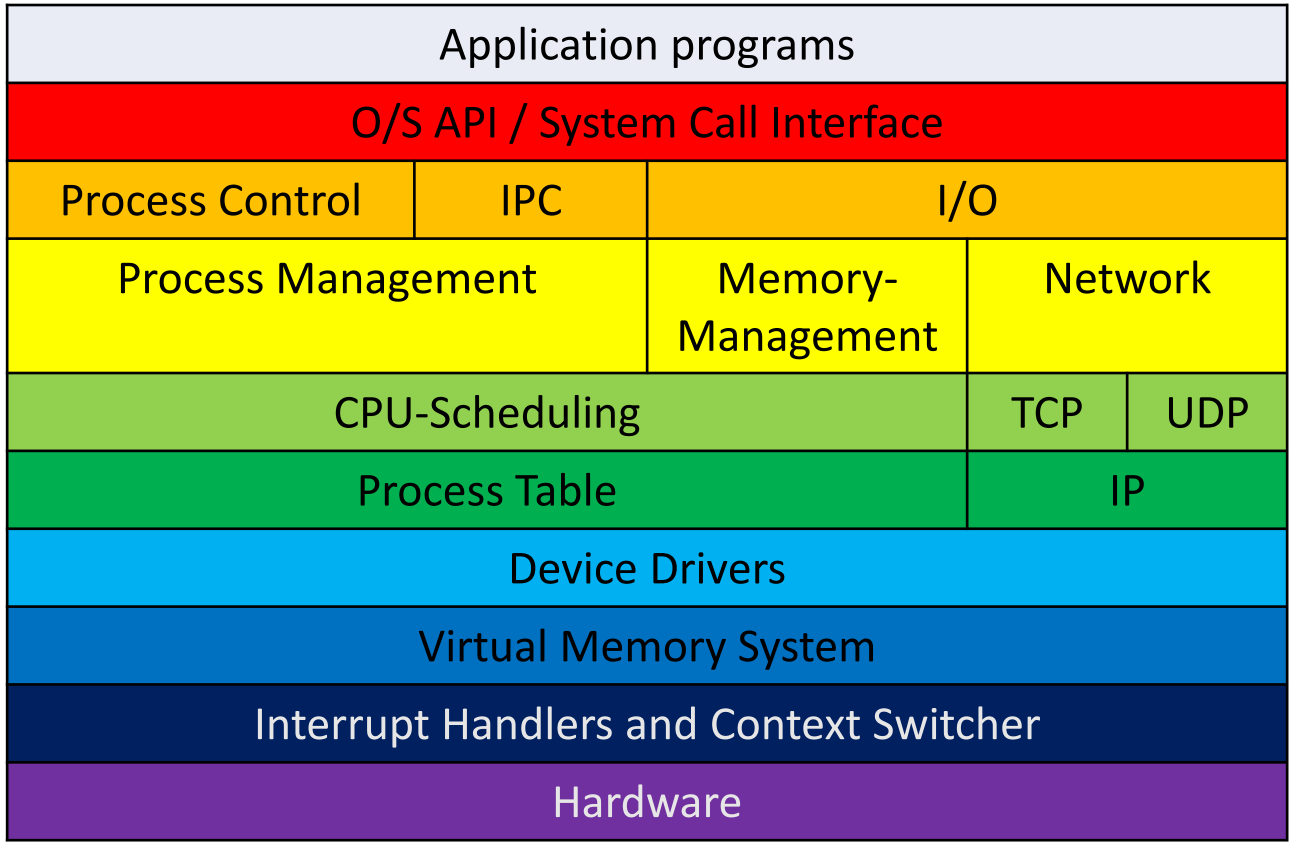
Agenda

- Assignment 1 is out
- Today’s lecture
  - OS overview
    - OS Structure
    - Virtual Machines
    - Operation
  - Process
    - Composition
    - State
    - Creation and termination
    - Data structures used to manage processes
- Reading: Sections 2.7-2.9, 3.2, 3.3, 5.1

The Layered Approach

- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Allows separation between
  - **Policy**
    - What needs to be done
  - **Mechanism**
    - How it is done

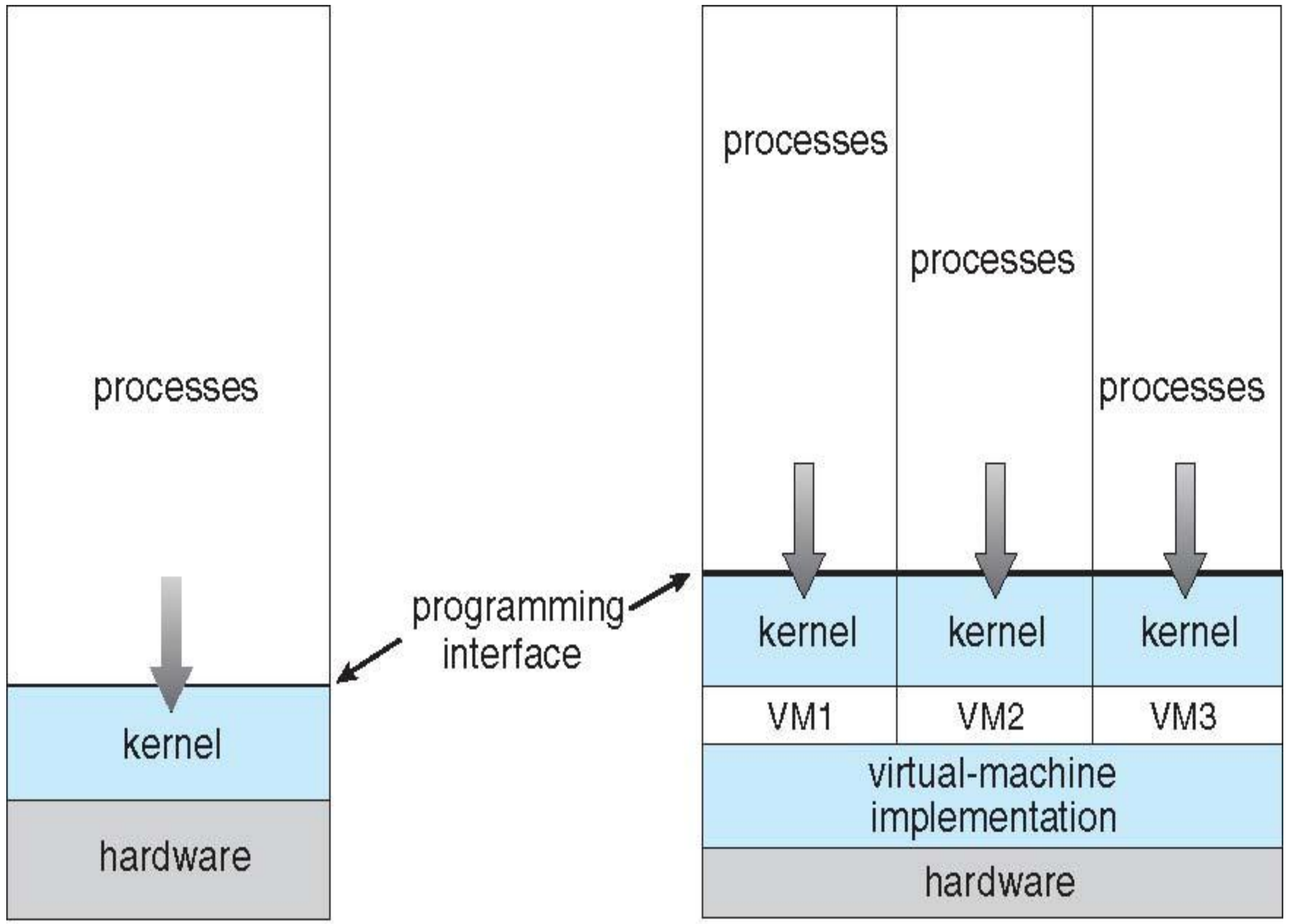


Virtual Machines

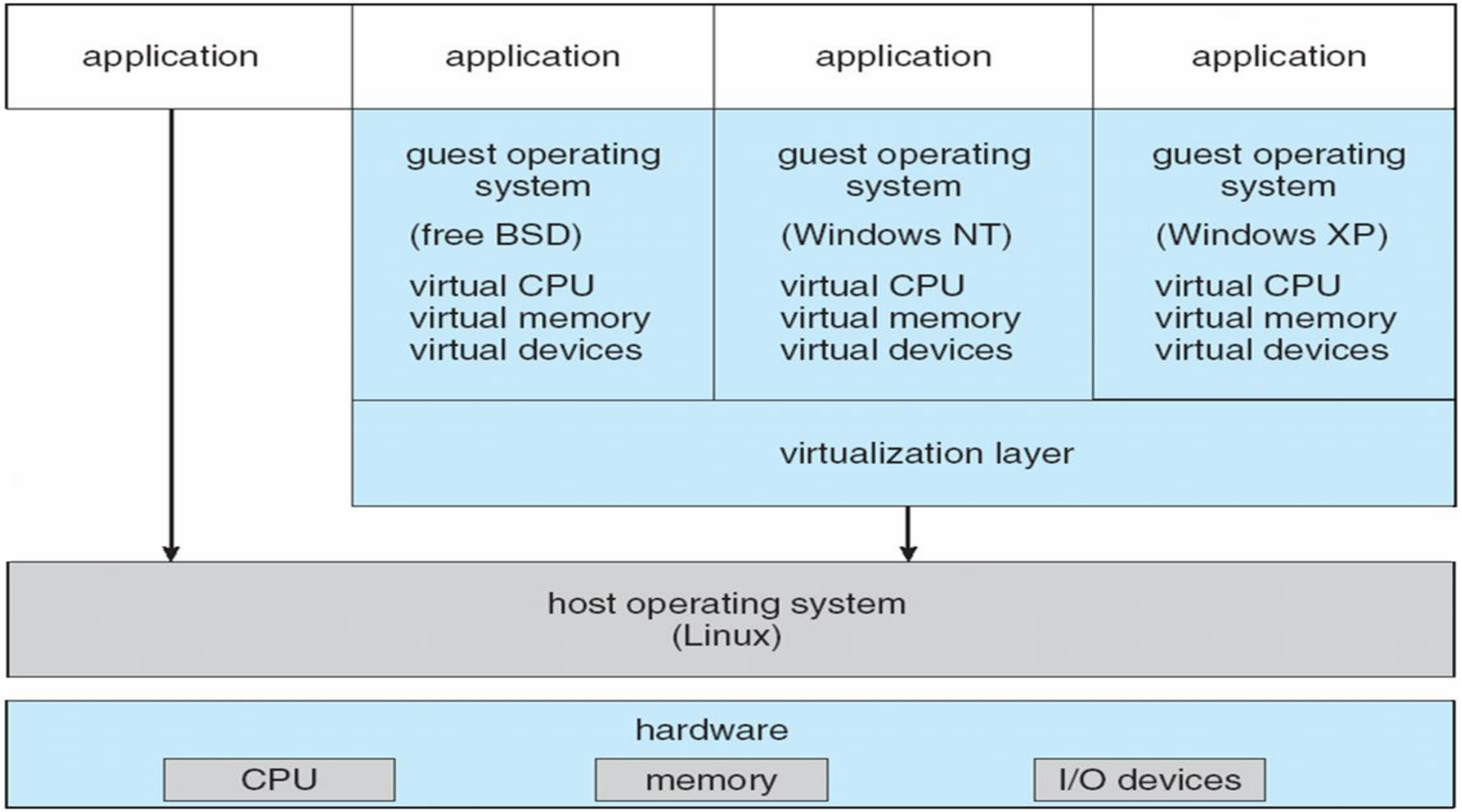
- A **virtual machine** takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system **host** creates the illusion that a process has its own processor and (virtual memory).
- Each **guest** is provided with a (virtual) copy of underlying computer.

Virtual Machines

(a) Non-virtual machine (b) virtual machine

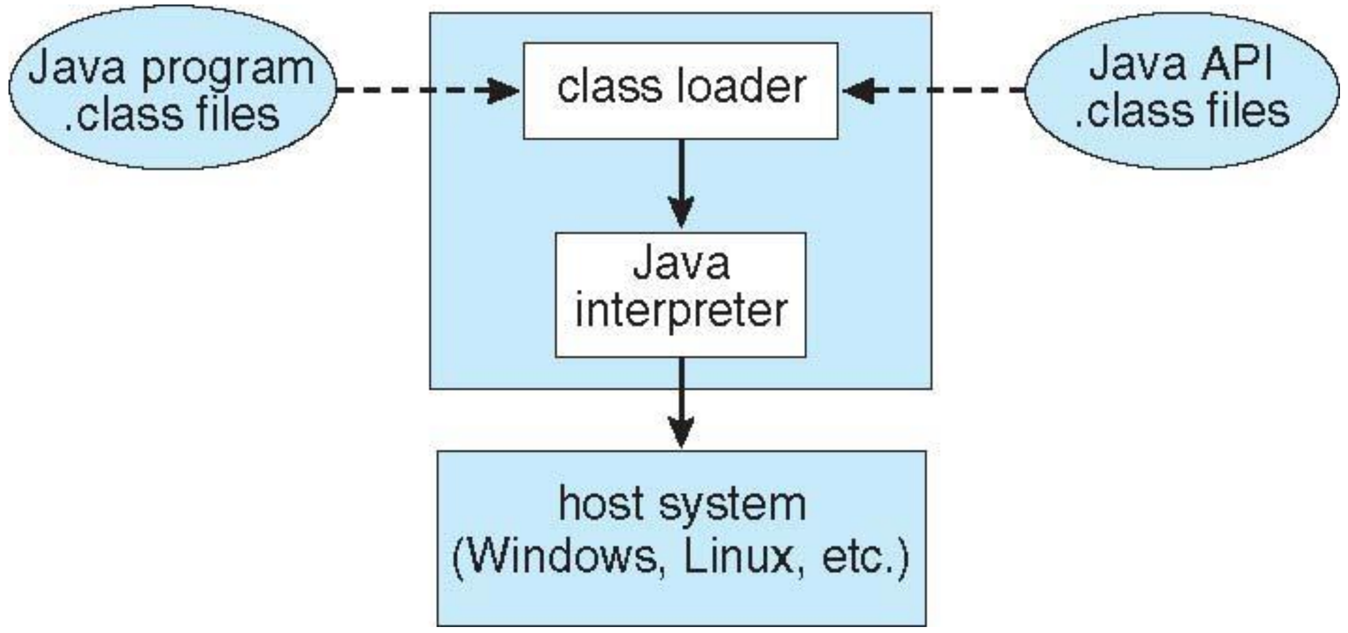


VMware Architecture

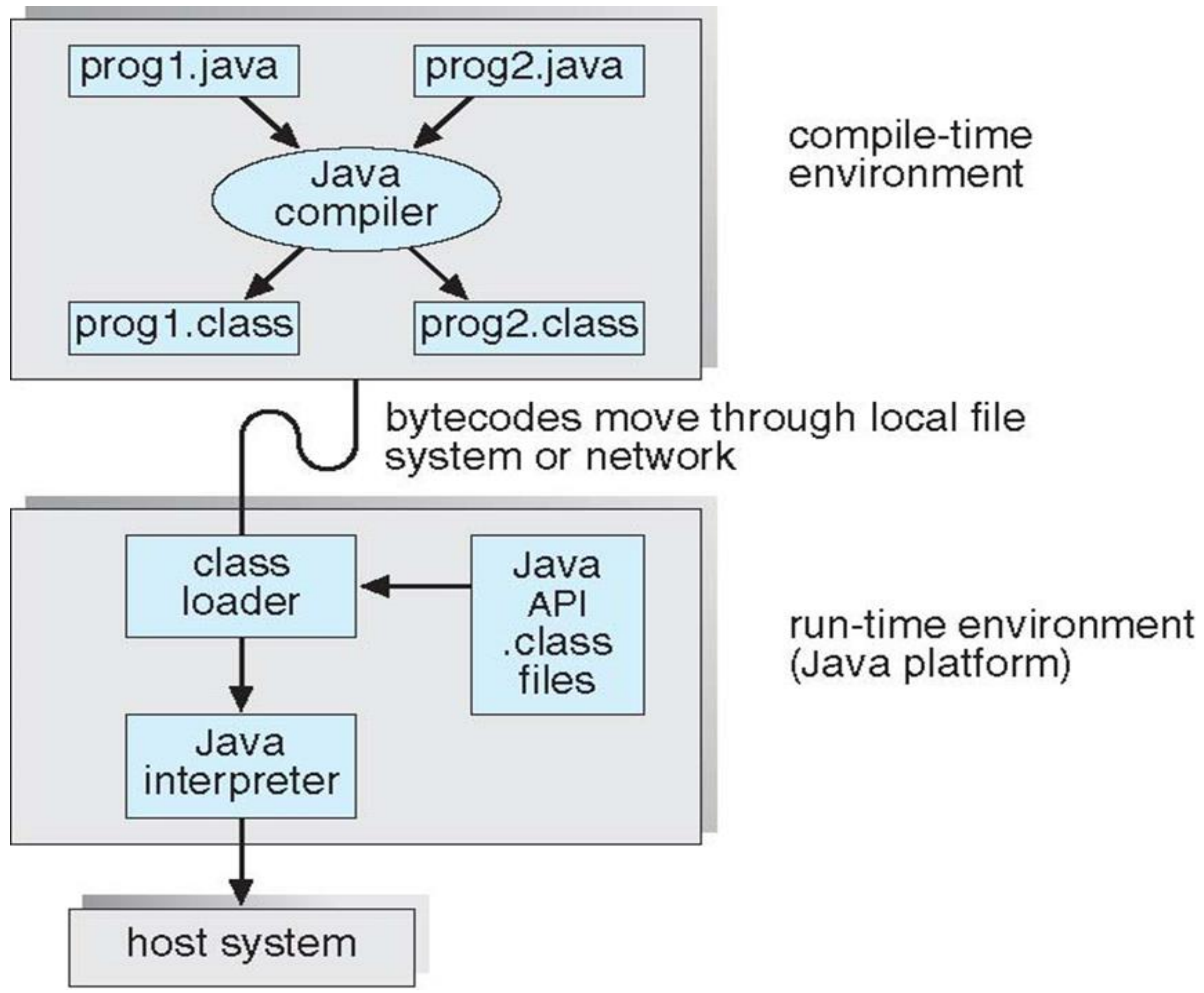


Java

- Java consists of:
  1. Programming language specification
  2. Application programming interface (API)
  3. Virtual machine specification



The Java Development Kit



Running Processes

Dual Mode of Operation

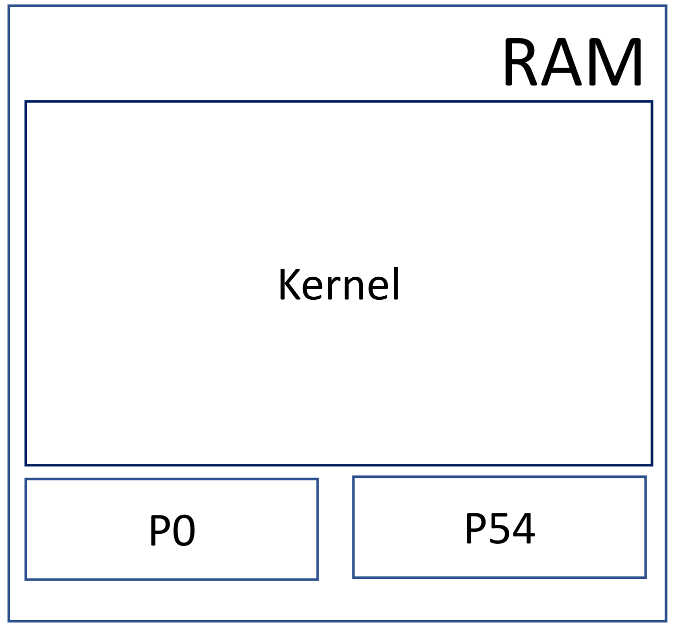
- **Kernel** mode:
  - Program running on CPU has access to all hardware both I/O devices and memory
  - Can execute all CPU instructions
  - a.k.a **superviso**r mode, **system** mode, or **control** mode
- **User** mode:
  - Program running on CPU has restricted access to devices and memory
  - Can execute a restricted set of CPU instructions

Starting the OS

- An operating system must be made available to hardware so hardware can start it
- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM (fixed memory location), generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system **kernel** and starts execution

The Kernel

- Is the portion of the OS that is loaded into memory upon start-up
- Performs most OS roles
  - Protect processes and hardware
  - Manage resources and hardware
  - Create and run processes
  - Service process requests
- Resides in memory at all times
- Runs in supervisor mode (kernel mode)
- Cannot be executed, read, or modified by other processes



Processes

- Created and run by the **Kernel**
  - Has complete control of the CPU
  - Run in user mode
- A process
  - **Can** make requests to the kernel to perform I/O and other functions
  - **Can** access its own memory
  - **Cannot** directly access any of the I/O devices
  - **Cannot** directly access, execute, or modify the kernel

Question: how does the kernel regain control from a process?

Interrupts

- Mechanism by which other modules may interrupt the normal sequencing of the processor
- The OS preserves the state of the CPU by storing registers and the program counter (process context)
- Determines which type of interrupt has occurred:
    - **polling**
    - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Handling an Interrupt

- In the event of an interrupt
  - CPU switches to kernel mode
  - Index interrupt table using int ID
  - Invokes interrupt handler
- When the interrupt terminates
  - CPU switches to user mode
  - Returns to where the process was before the interrupt

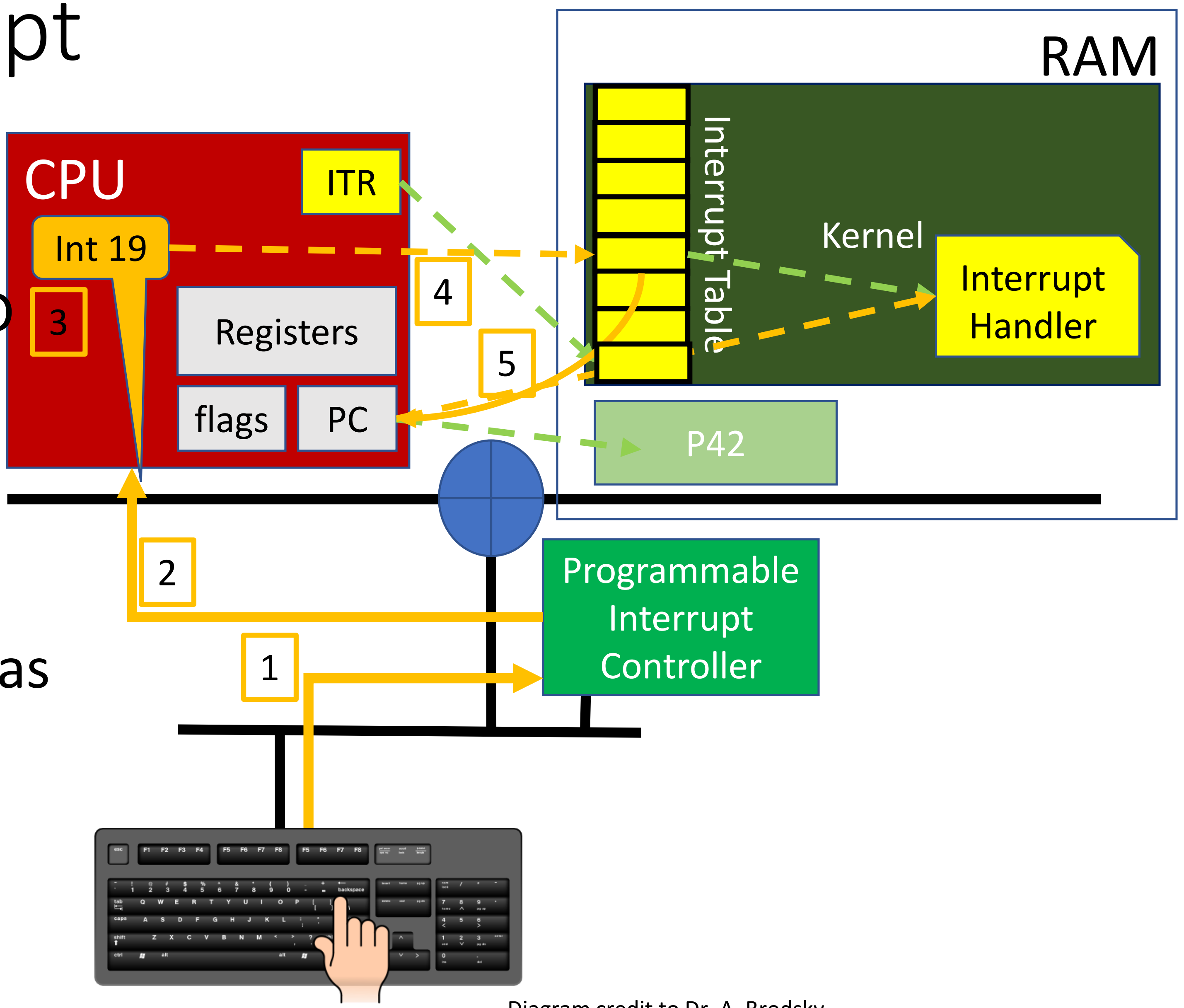


Diagram credit to Dr. A. Brodsky

Purpose of Interrupts

The purpose of interrupts is to

- ✓Help the kernel gain control of the CPU
- ✓Notify the kernel when an I/O event has occurred
- ✓Help processes make requests to the kernel in a control way
- ✓Notify kernel when an error has occurred
- ✓Improve processor utilization since most I/O devices are slower than the processor (more on this later)

Question: what if multiple interrupts occur?

Multiple Interrupts

An interrupt occurs while another interrupt is being processed

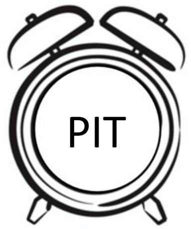
- e.g. receiving data from a communications line and printing results at the same time

Two approaches:

- Disable interrupts while an interrupt is being processed
- Use a priority scheme

Question: what if no interrupts occur?

Timers



- Want to generate an interrupt on a regular basis to ensure **kernel** gains control
- Use a hardware timer to generate the interrupt, every 10ms or so
- This allows kernel to gain control on a regular basis
- No process can monopolize the CPU because kernel can switch to another process
- **But, how do processes make requests to the kernel?**

Traps

- Processes need to access the kernel in a controlled manner
- A process can generate an exception called a trap (software interrupt)
  - Division by zero, request for OS service, infinite loop
- On the Intel architecture, this is the int # instruction
  - For example, int 53 invokes interrupt 53
- The mechanism works exactly as if the PIC sent a notice to the CPU with ID 53.

Question: **How does the actual switch occur?**

Context Switching

- When an interrupt handler runs
  - Current process has to be context switched out
  - The Kernel context has to be context switched in
- To switch out a process, the handler
  - Copies all registers from CPU to memory
  - Initializes all registers to what the kernel expects
  - Switches to the kernel stack
  - Runs the handler
- To switch in the process
  - Switches to the processes stack
  - Copies registers from memory to the CPU
  - Returns from where the process left-off
- **Key Idea: The process has no sense that an interrupt has occurred!**

OS Debugging

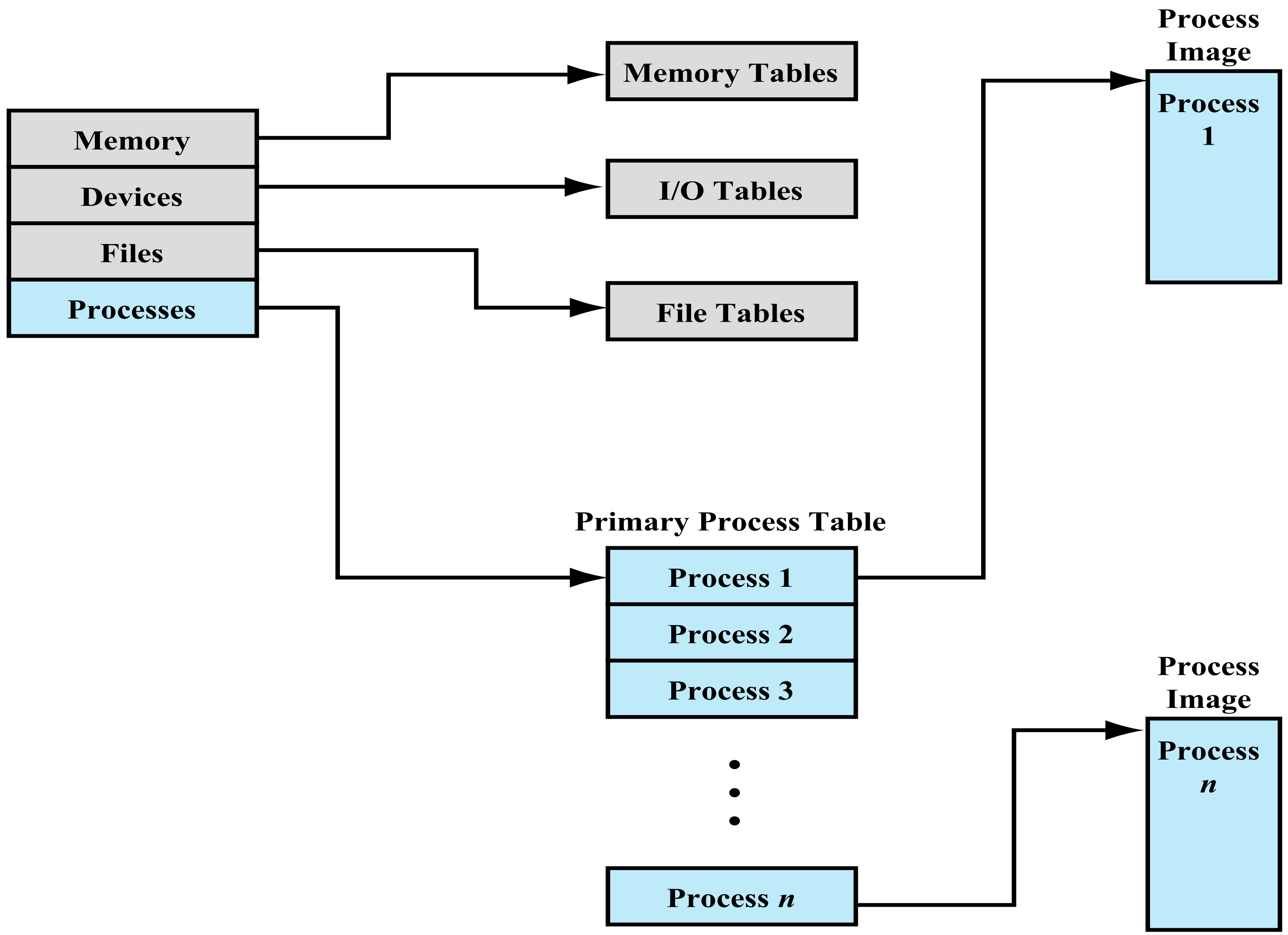
- **Debugging** is finding and fixing errors, or **bugs**
- Kernighan’s Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”
- Oses generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Oses failure can generate **crash dump** file containing kernel memory
- Dtrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems

Let’s talk more about Processes!

Processes

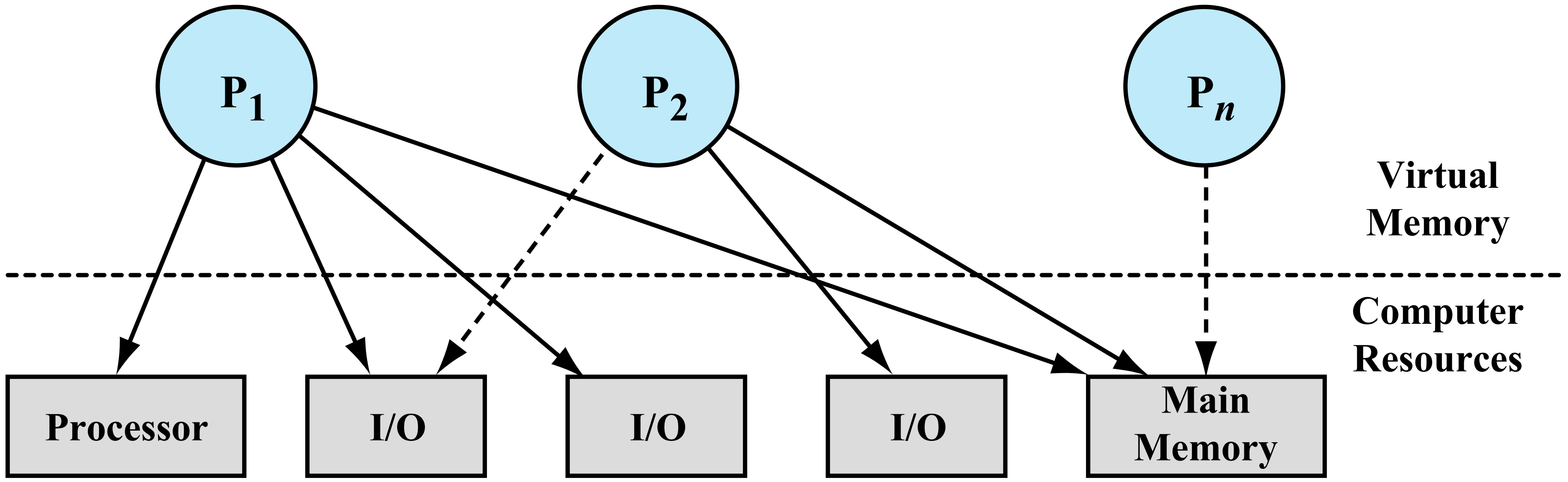
- What we know so far
  - A process is a running instance of a program
  - Processes run in user mode or kernel mode
  - Interrupts are used to switch between modes
  - Multiple processes can exist in the system
  - The OS is responsible for managing and controlling processes
- What we don’t know
  - What are the elements of a process?
  - How does the OS keep track of and manage processes?
  - What type and role data structures used have?

Control Tables



Process Elements

- Two essential elements so far:
  - **Program code:** might be shared with other processes
  - **Data**: associated with the program code
- What else do we need?

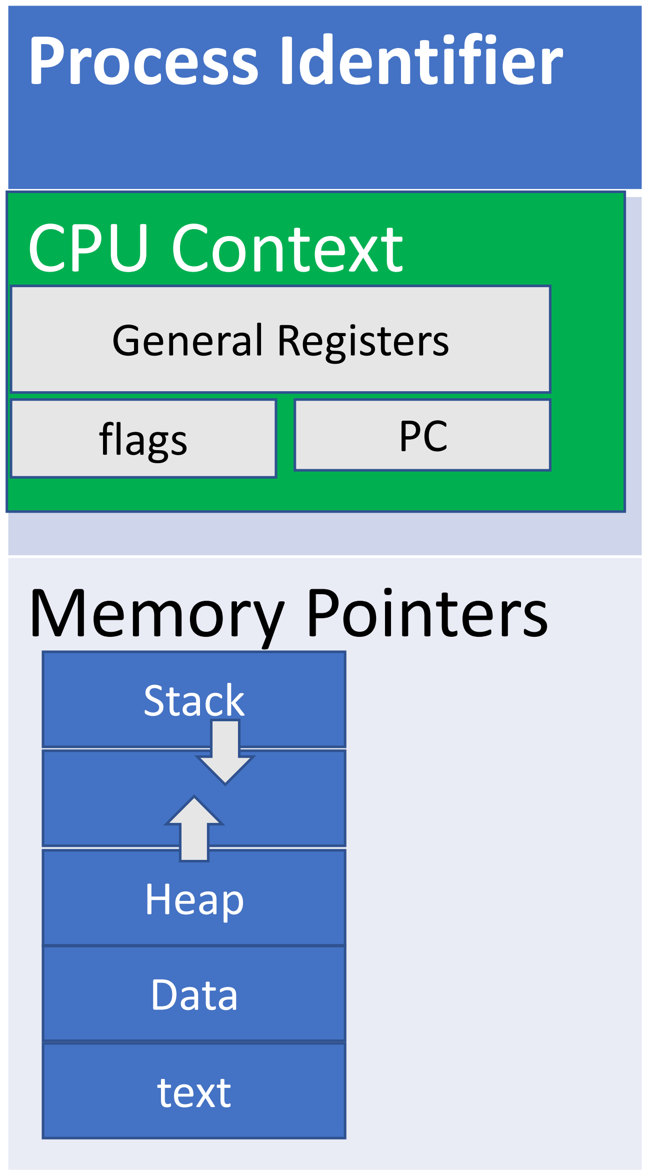


Process Control Structures

- To manage and control processes the OS needs to know
  - Where the process is located
    - Program or set of programs to be executed
    - Sufficient memory to hold the program and the data associated with it
    - Stack to track of procedure call and parameter passing between procedures
  - The attributes of the process that are necessary for its management
    - Collection of program, data, stack and attributes refereed to as process image
    - Stored using a **Process Control Block**

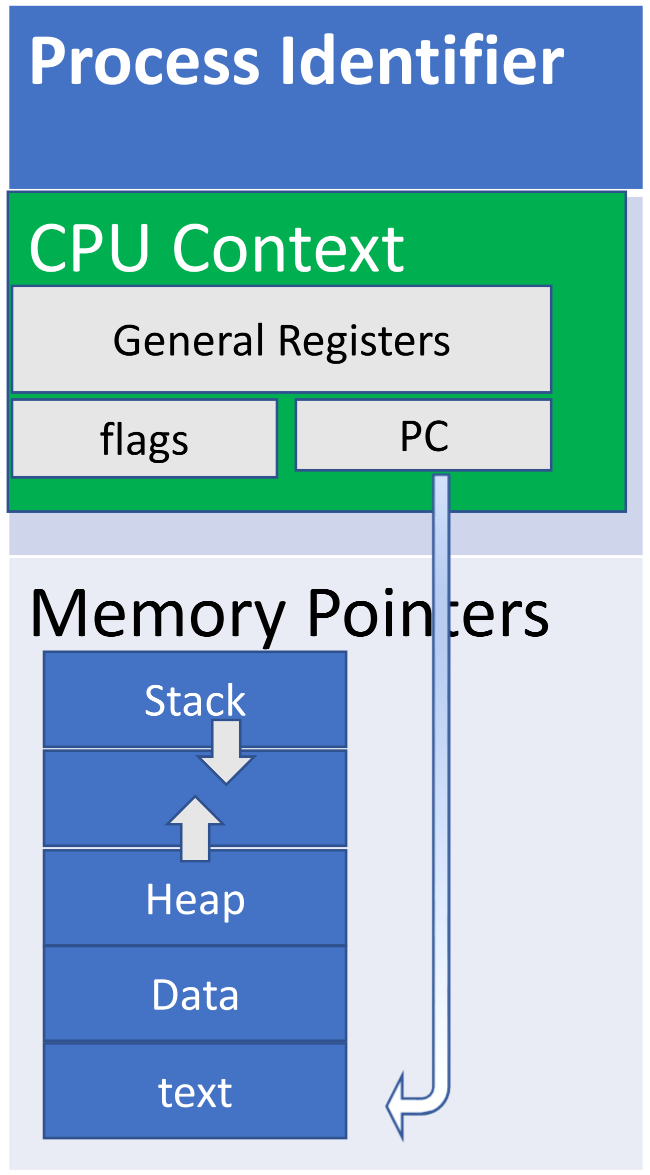
The Process Control Block

- Created and managed by the OS
- Contains process elements
- Key tool to support multiple processes
- A **thread** is an active execution of program code
  - Uniquely identified
  - Located in memory
- A thread of execution depends only on:
  - CPU context
  - Memory Pointers



The Process Control Block

- A thread of execution depends only on:
  - CPU context
    - General registers
    - Program counter: points to the current instruction
    - Special purpose registers
  - Memory Pointers
    - **Stack** : local variables and return addresses
    - **Heap** : dynamically allocated memory
    - **Data** : global and static variables
    - **Text**: program code

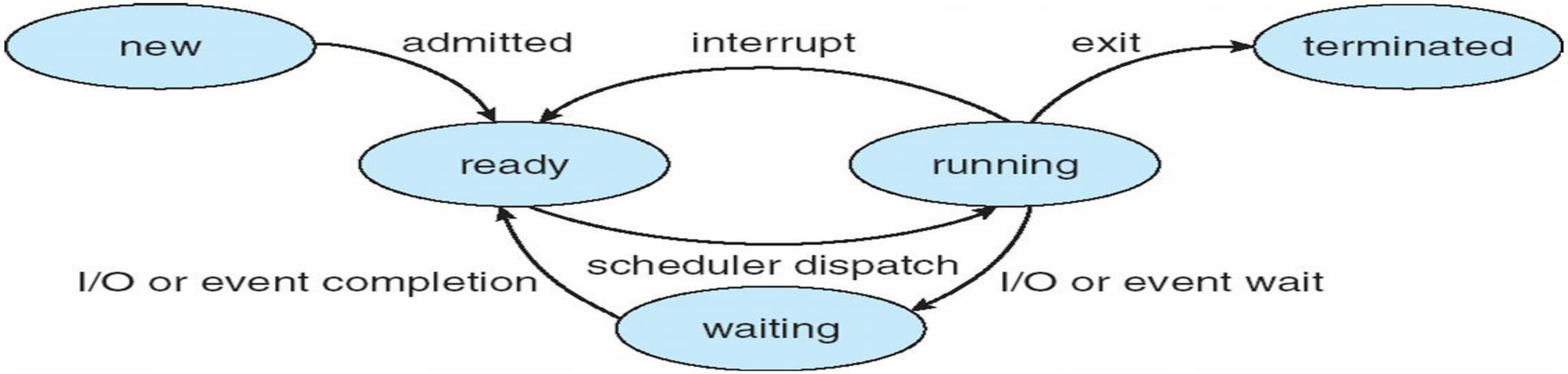


Process State

- A thread of execution changes state
    - **New**: the process is being created
    - **Ready**: the process is waiting to be assigned to a processor
    - **Running**: instructions are being executed
    - **Waiting (blocked)**: the process is waiting for some event to occur
    - **Terminated**: the process has finished execution

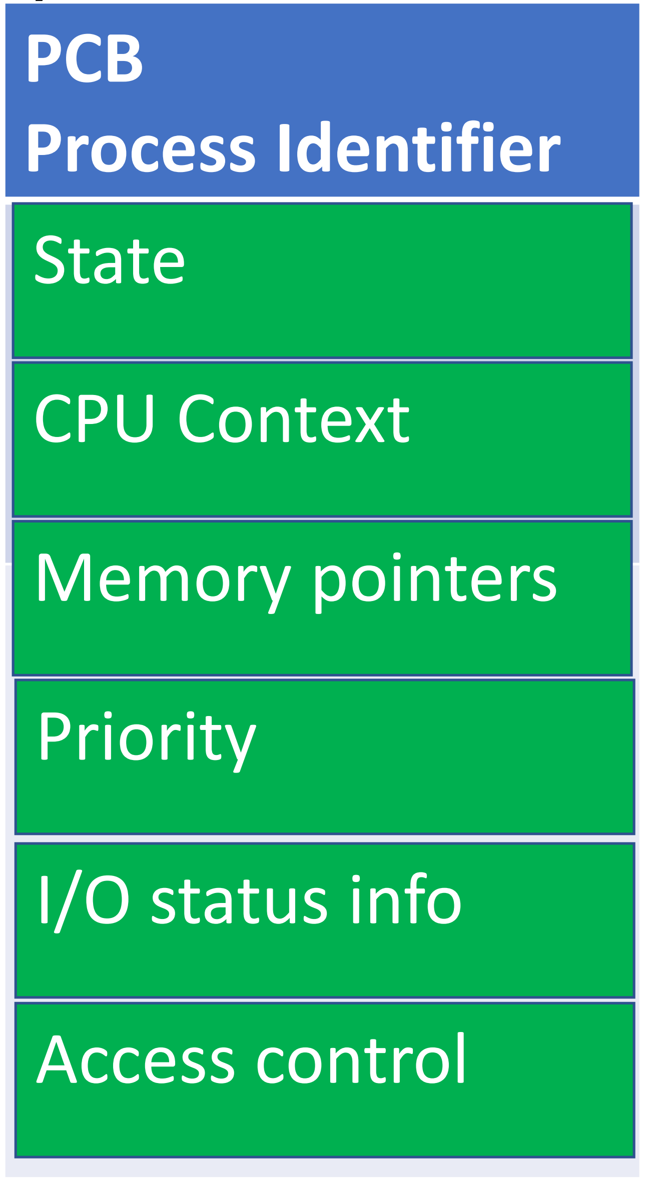
Process State Transitions

- A thread of execution changes state
  - **New**: the process is being created
  - **Ready**: the process is waiting to be assigned to a processor
  - **Running**: instructions are being executed
  - **Waiting (blocked)**: the process is waiting for some event to occur
  - **Terminated**: the process has finished execution



The Process Control Block (revisited)

- When a new process is created, a PCB is allocated
- PCB contains process information:
  - **pid**: unique integer identifier
  - **State**: process' current state (new, ready,….)
  - **CPU context:** including program counter and registers
  - **Memory pointers:** including location of address space
  - **Priority:** used for scheduling
  - **I/O**: any open files (file descriptors)
  - **Access control :** and any accounting information
- A ***process table*** is used to store PCBs

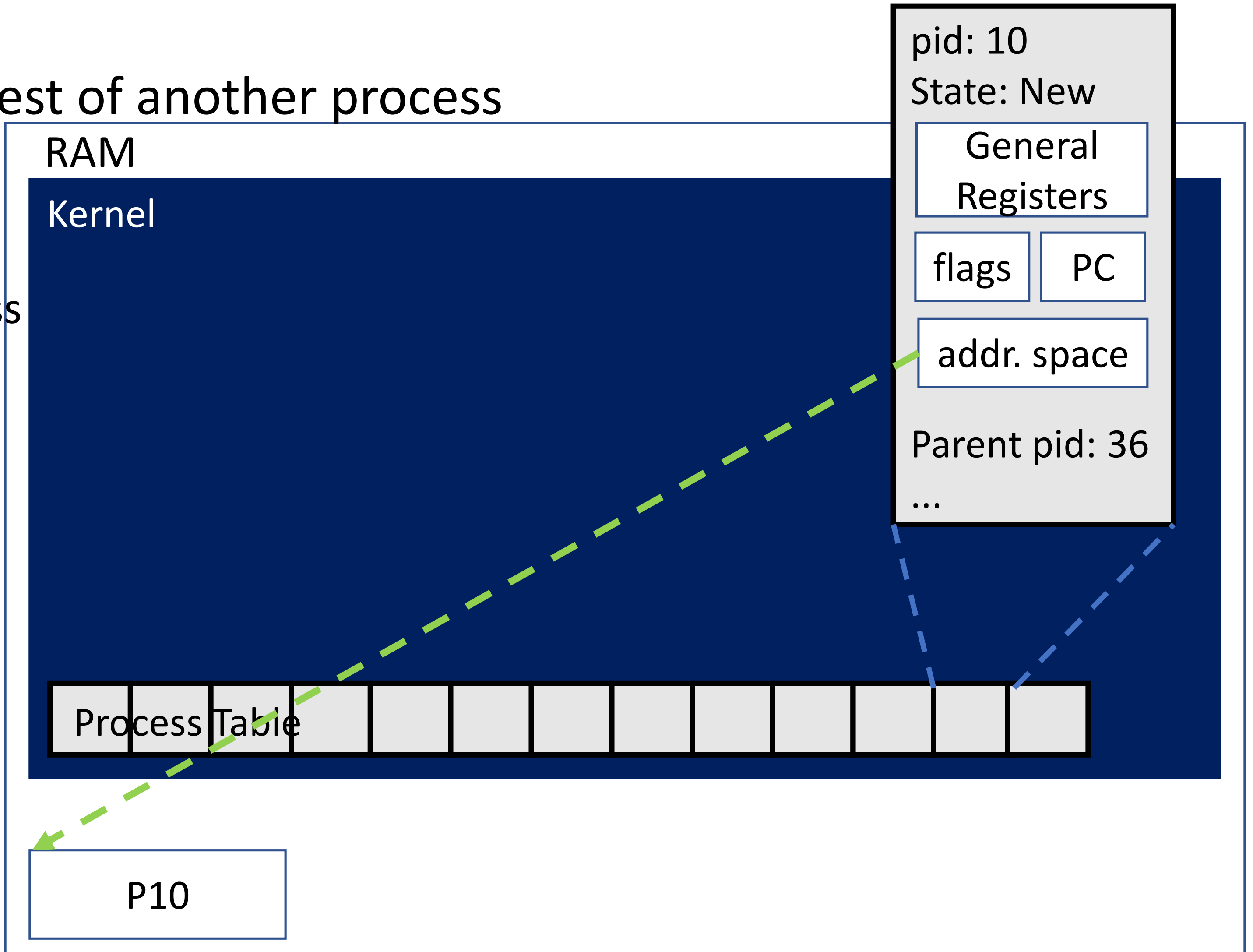


Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation

- **Process creation**
  - OS creates a process at the request of another process
  - Assigns a new pid
  - Allocates space for the process
    - Makes a copy of the parent process
  - Allocates new PCB
  - Copies PCB from parent to child
  - Sets state to New

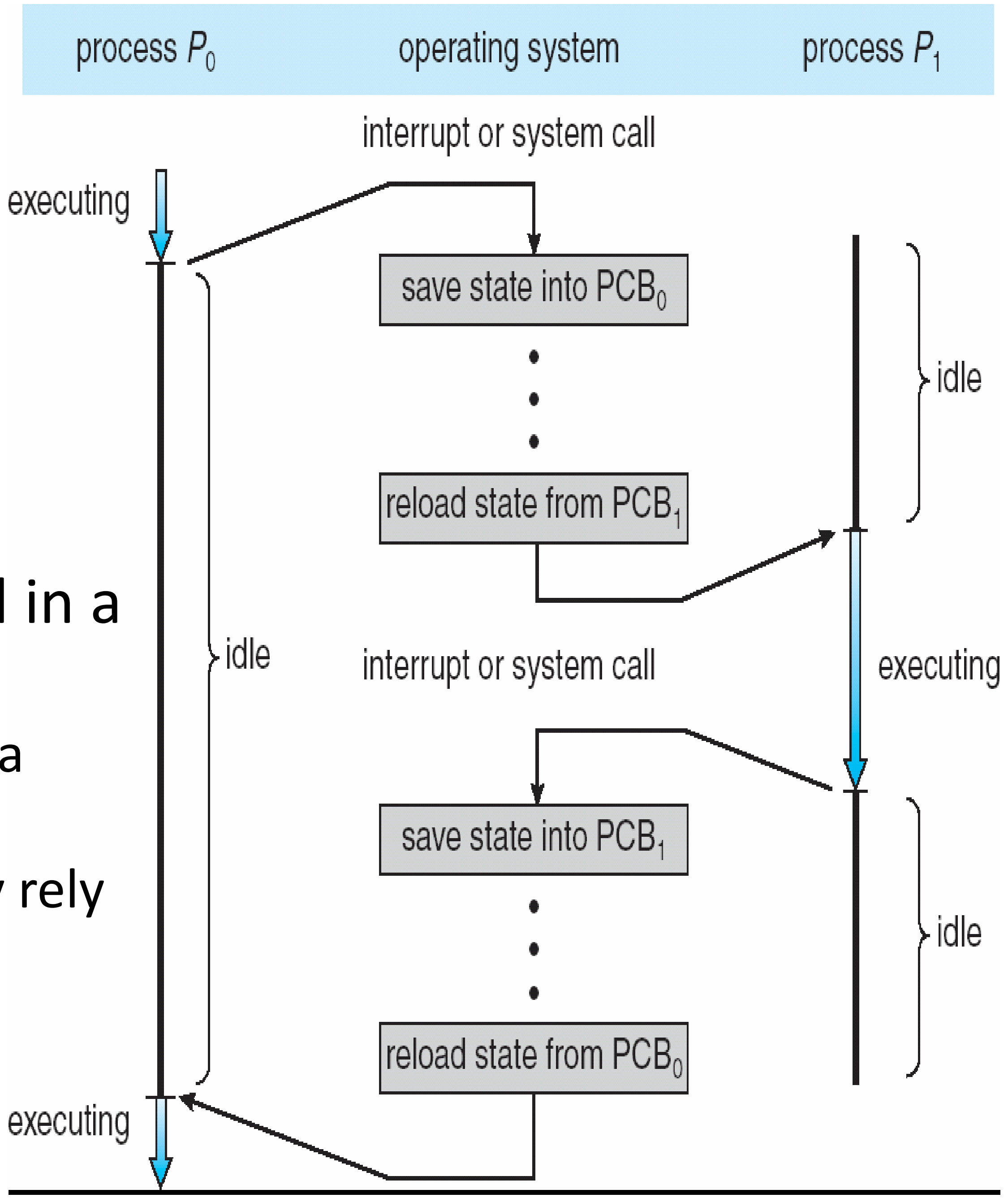


Process Termination

- Reasons for termination
    - Normal completion
    - Time limit exceeded
    - Protection error
    - Arithmetic error
    - …
- Process termination, involves
    - Marking process’ state as terminated
    - Closing process’ files
    - Freeing process’ memory
    - Deallocating process’ PCB and pid

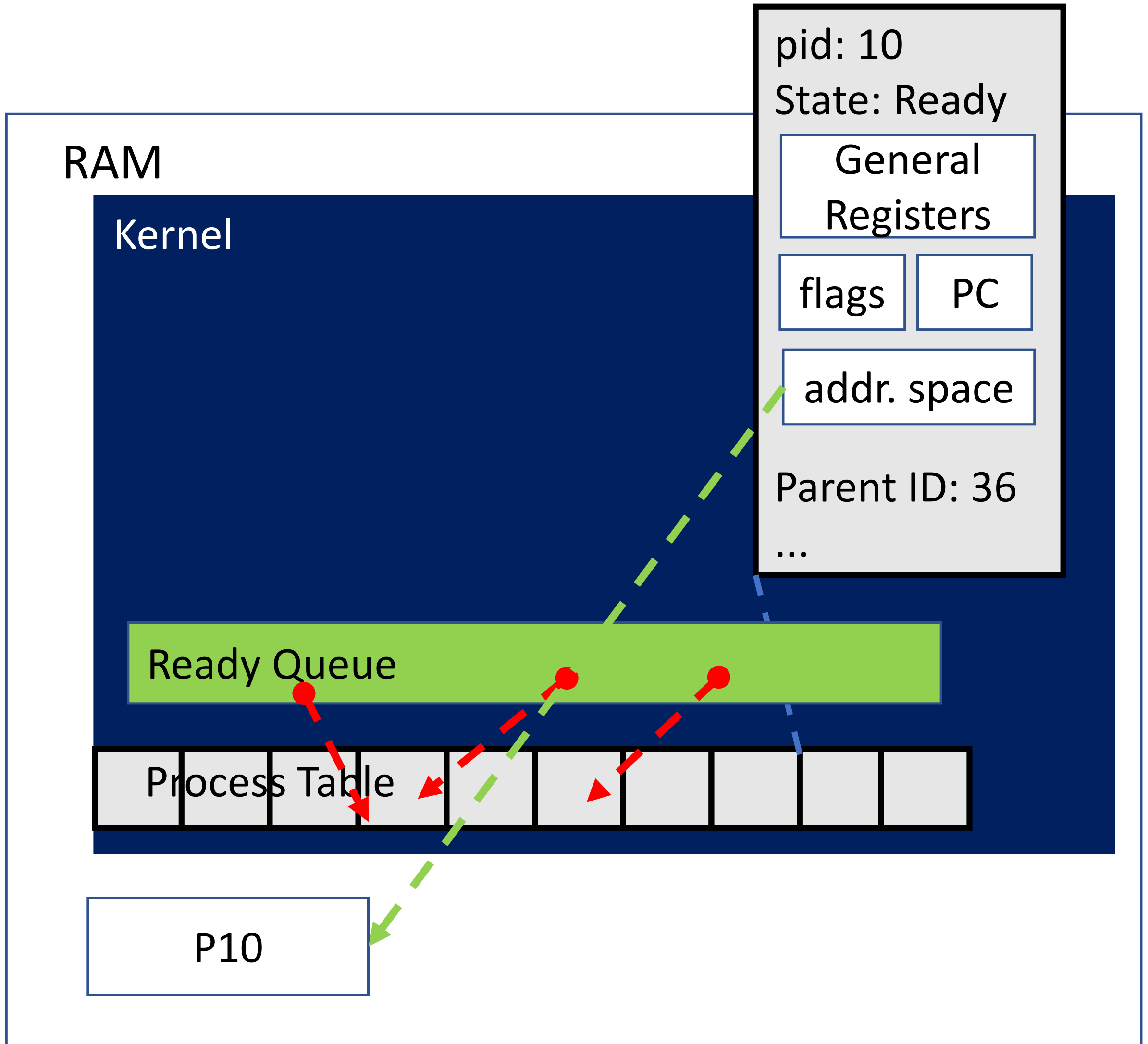
CPU Switch

- Not all processes can run at the same time. Why?
- Processes run based on a schedule
- Those that are ready to run are stored in a ***Ready queue***
  - Abstract data structure implemented as a Queue, priority queue, etc…
  - Scheduler and scheduling algorithm may rely on more than one queue



Queues

- Used to schedule processes
- While a process is waiting in the queue, it is in the waiting state
  - **Job** queue – set of all processes
  - **Ready** queue – processes in main memory, ready to execute
  - **Device** queues – processes waiting for an I/O device
- Processes move among various queue creating a cycle



An I/O Request

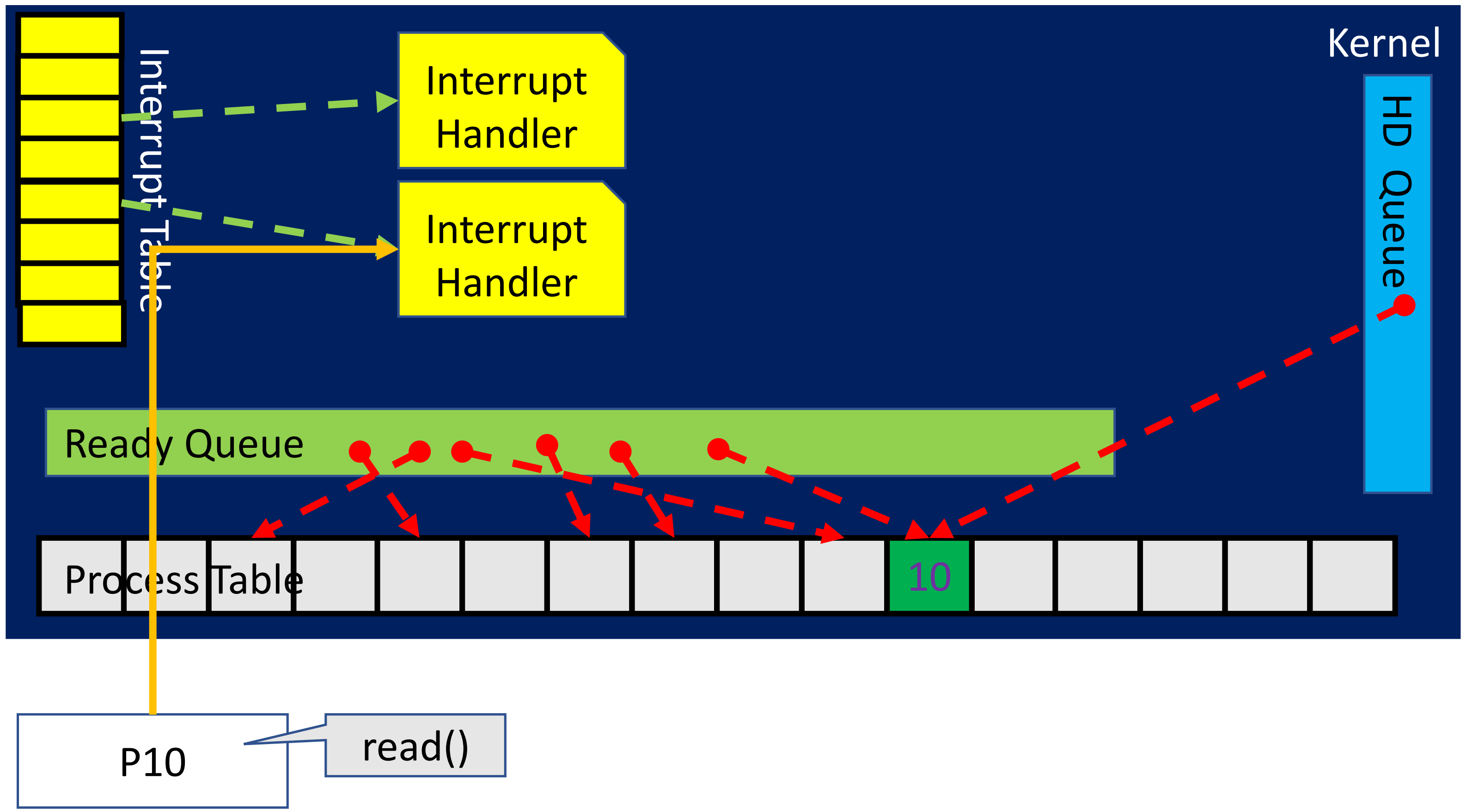


Diagram credit to Dr. A. Brodsky