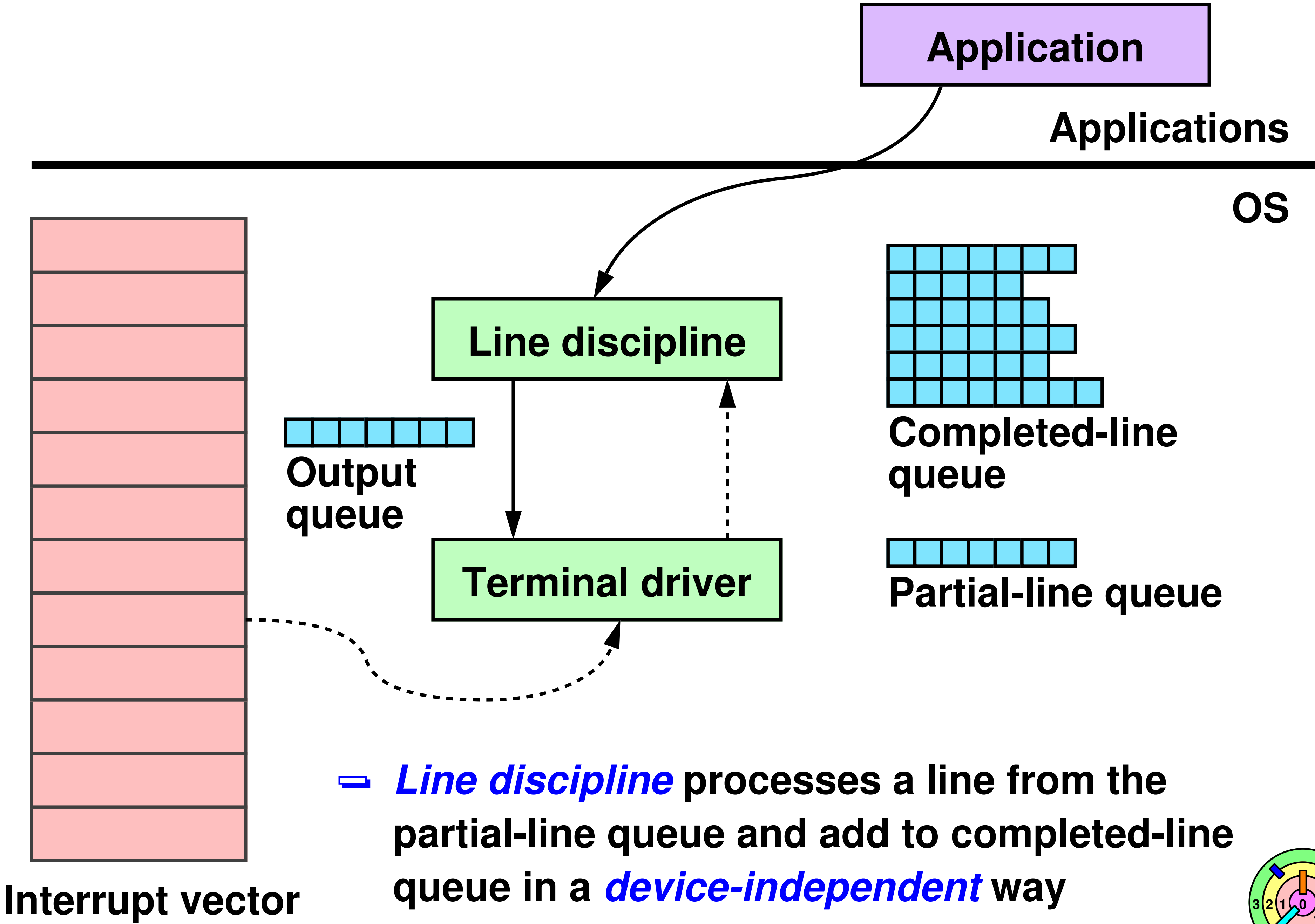
Modularization

- ***Device independence*** **consideration (figure out the *common* part)**
  - **for many different serial-line devices, character processing is common**
    - **actually, character processing is performed in situations where the source and sink of characters aren't even a serial line**
      - **e.g., bit-mapped display, network connection**
  - **therefore, it makes sense to separate the device dependent part from the *common*, *device independent* part**
    - **promotes *reusability***

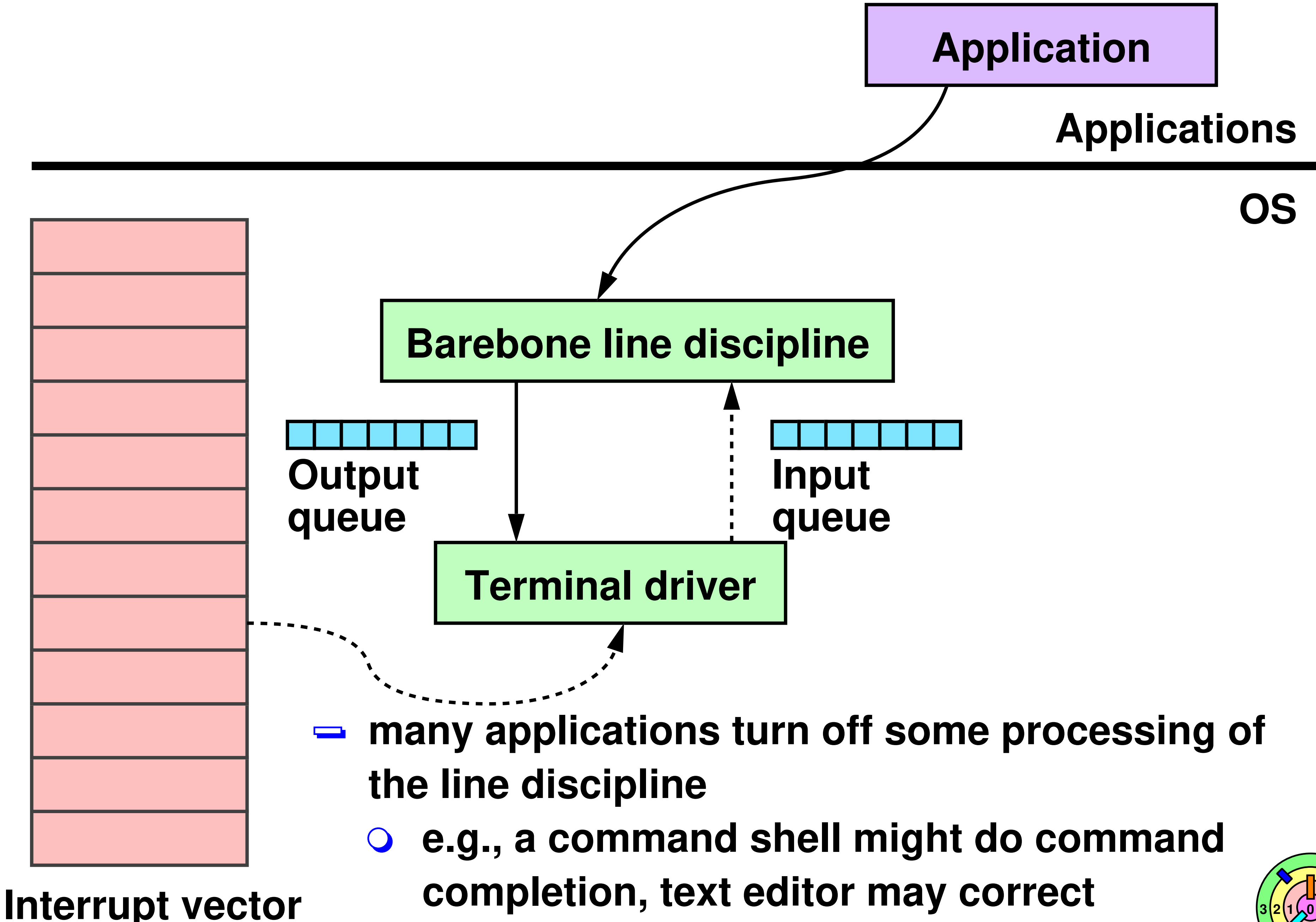
- **A separate module, known as the *line-discipline* module in some systems, provides the *common character-handling code***
  - **it can interact with any device driver capable of handling terminals**
  - **can even use a different line-discipline module to deal with an alternative character set**

Terminals



- *Line discipline* processes a line from the partial-line queue and add to completed-line queue in a *device-independent* way

Terminals



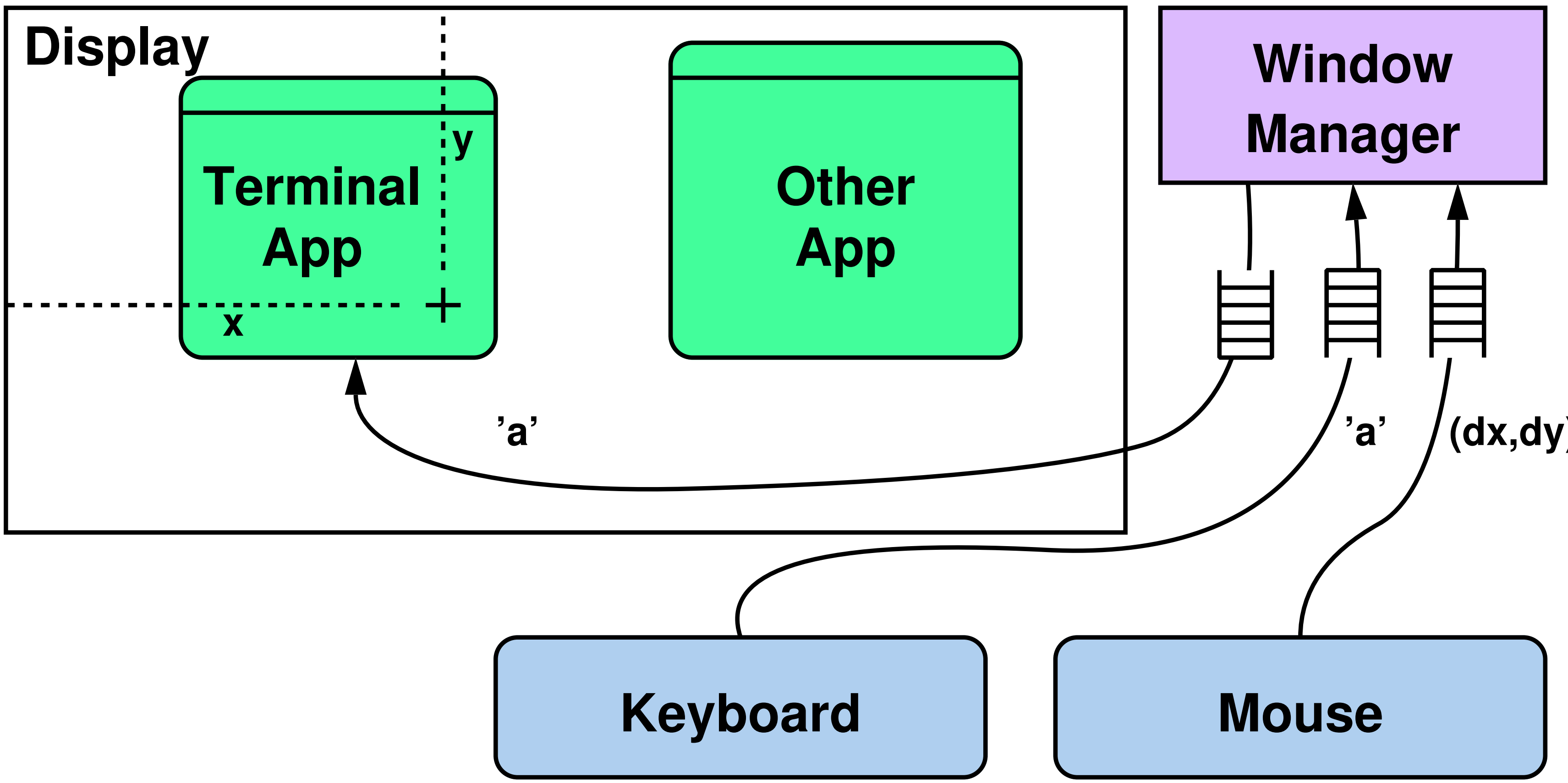
- many applications turn off some processing of the line discipline
  - e.g., a command shell might do command completion, text editor may correct spelling

Where to Put the Modules?

- Where to put the terminal driver and the line-discipline module?
  1) kernel
  2) separate user process
  3) library routines that are linked into application processes
  - *driver* should be in the kernel since device registers access needs to be protected from arbitrary manipulation by application programs
  - *line-discipline* may be shared by multiple applications
    - putting it in *library routines* will make it *difficult to share* one terminal with many user applications
    - can it go into a separate *user process*?
      - but can have serious *performance problems*
      - would need to transfer characters into the line-discipline process, then transfer to another process
    - putting it in the *kernel* seems to be the best choice
      - although kernel code is *hard to modify*, replace, and debug

Terminals and Pseudo Terminals

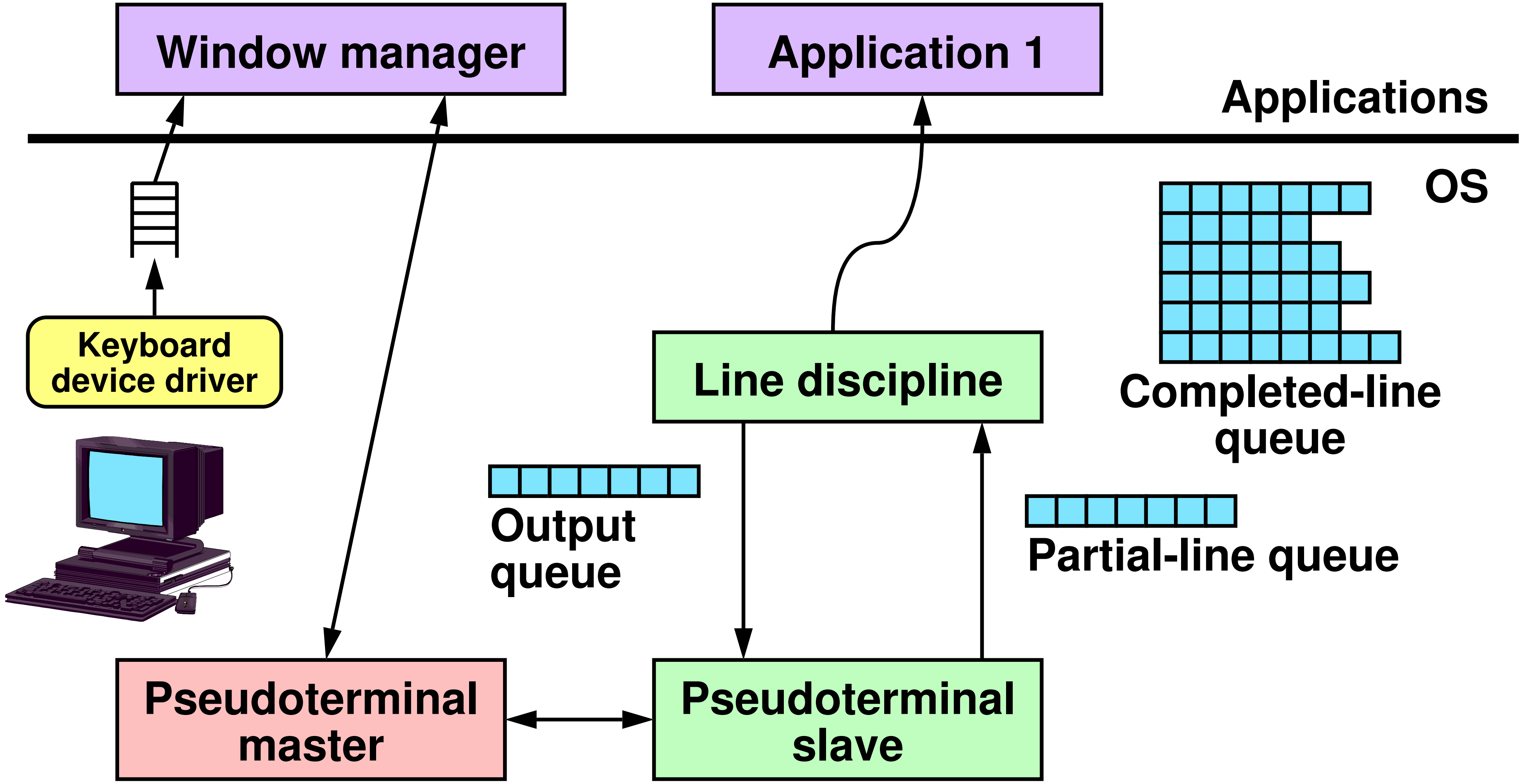
- **Modern systems do not have terminals**
  - they often have bit-mapped displays, keyboards and mice connected via USB
  - a ***window manager*** implements windows on the display and determines which applications receive typed input (***input focus***)



Terminals and Pseudo Terminals

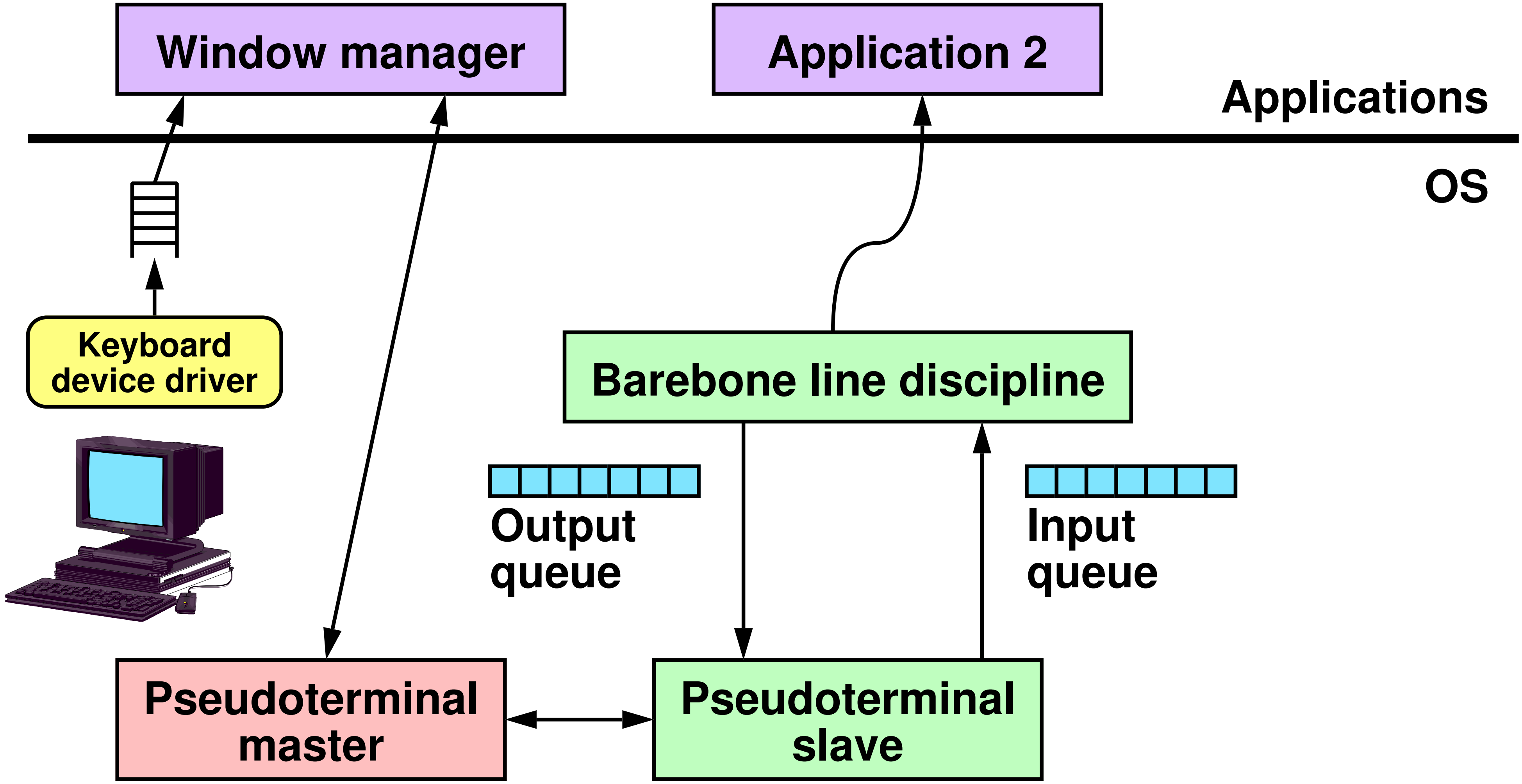
- **Modern systems do not have terminals**
  - they often have bit-mapped displays, keyboards and mice connected via USB
  - a *window manager* implements windows on the display and determines which applications receive typed input (*input focus*)
    - the window manager is a *user space program*
    - a server might support remote sessions where applications receive input and send output over a network
  - they use *pseudoterminals*
    - which implements a line discipline whose input comes from and output goes to a *controlling application* (and not a physical device)
    - two types of pseudoterminal (pseudo-)device drivers
      - *window manager* interacts with *pseudoterminal master*
      - *line discipline module* interacts with *pseudoterminal slave*

Pseudo Terminals



- the OS provides a pair of entities (*pseudoterminal master* and *pseudoterminal slave*) that appear to applications as *devices*

Pseudo Terminals



- pseudoterminal slave acts like a ***terminal device driver*** to the rest of the OS
- choose barebone line discipline module if desired

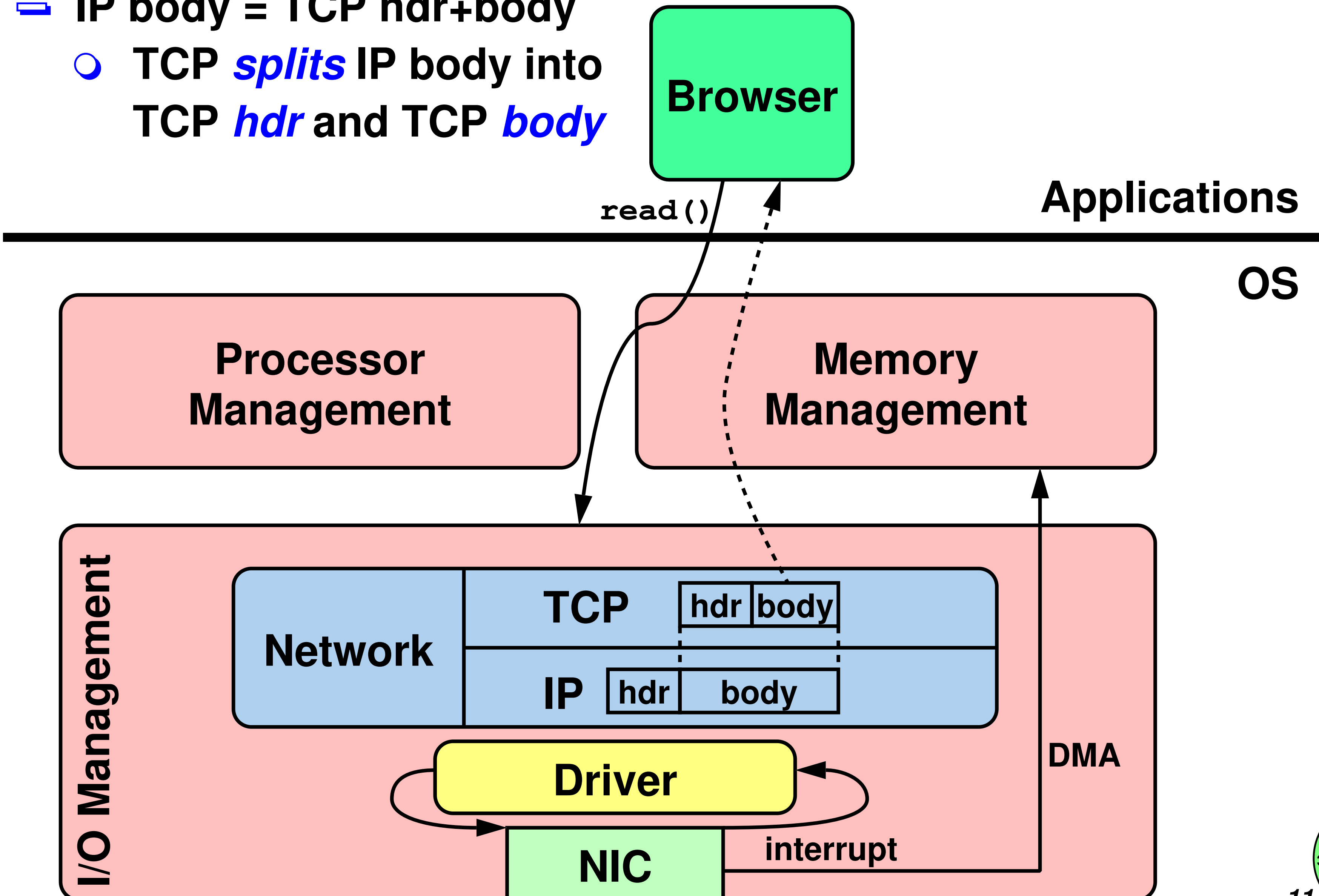
Network Communication

- Network communication and terminal handling are very similar, in architecture and implementation
  - device is called a *Network Interface Card (NIC)*
  - data arrives in a packet (instead of a character)
  - incoming data must be processed via *network-protocol modules* similar to line-discipline module

- Main difference
  - performance is crucial in network communication
    - data from keyboard can go at tens of characters per second
    - data going to display can go at a few thousand characters per second (for character-based display)
  - protocols are *layered* on top of one another
    - *data* in lower layer is views as *header + body* in higher layer
  - cannot afford to copy network data from queue to queue!
    - no copying allowed inside the kernel!
    - must pass by memory addresses!

Network Communication

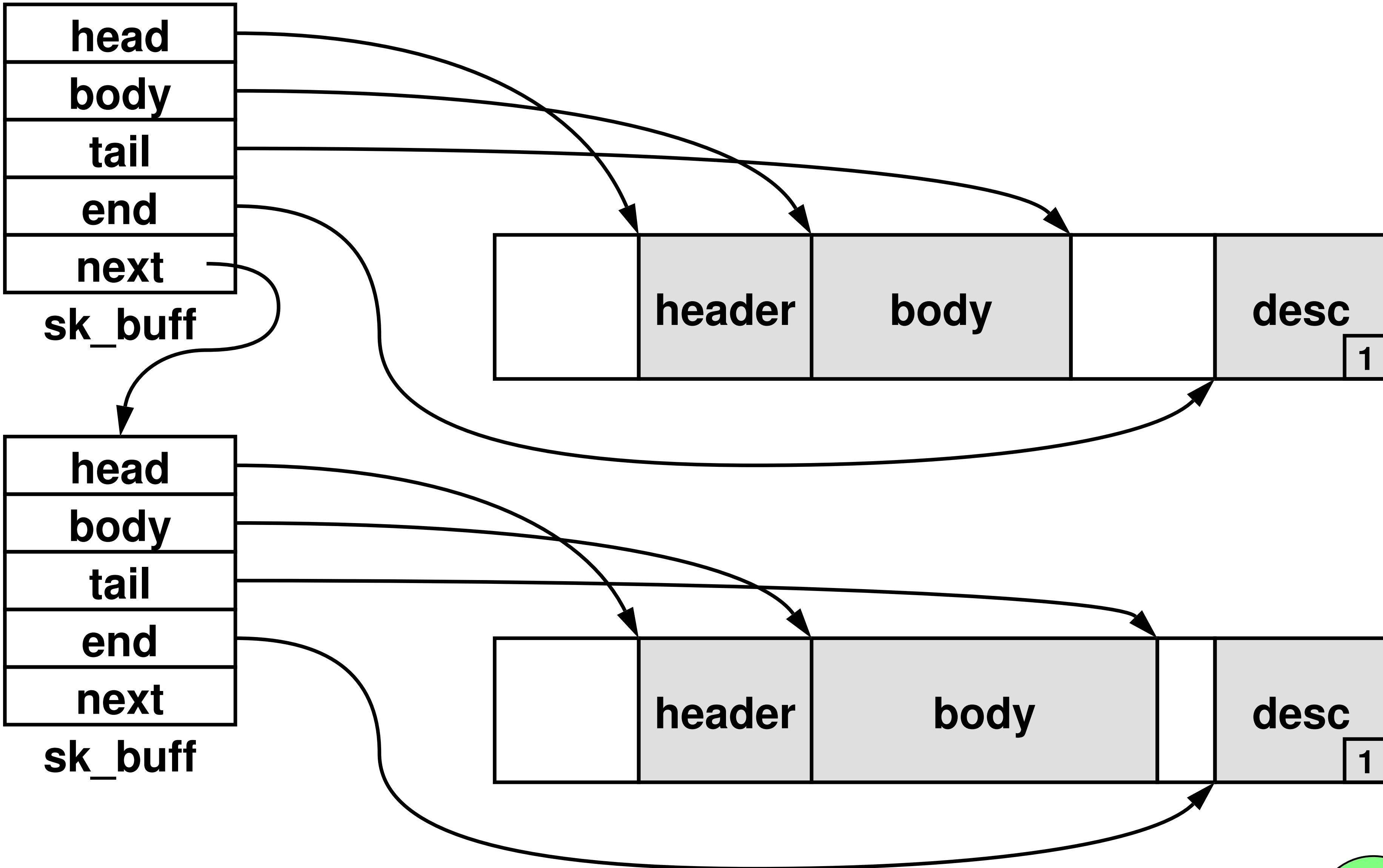
- Ex: TCP (details in Ch 9 which we will not cover)
  - IP body = TCP hdr+body
    - TCP *splits* IP body into TCP *hdr* and TCP *body*



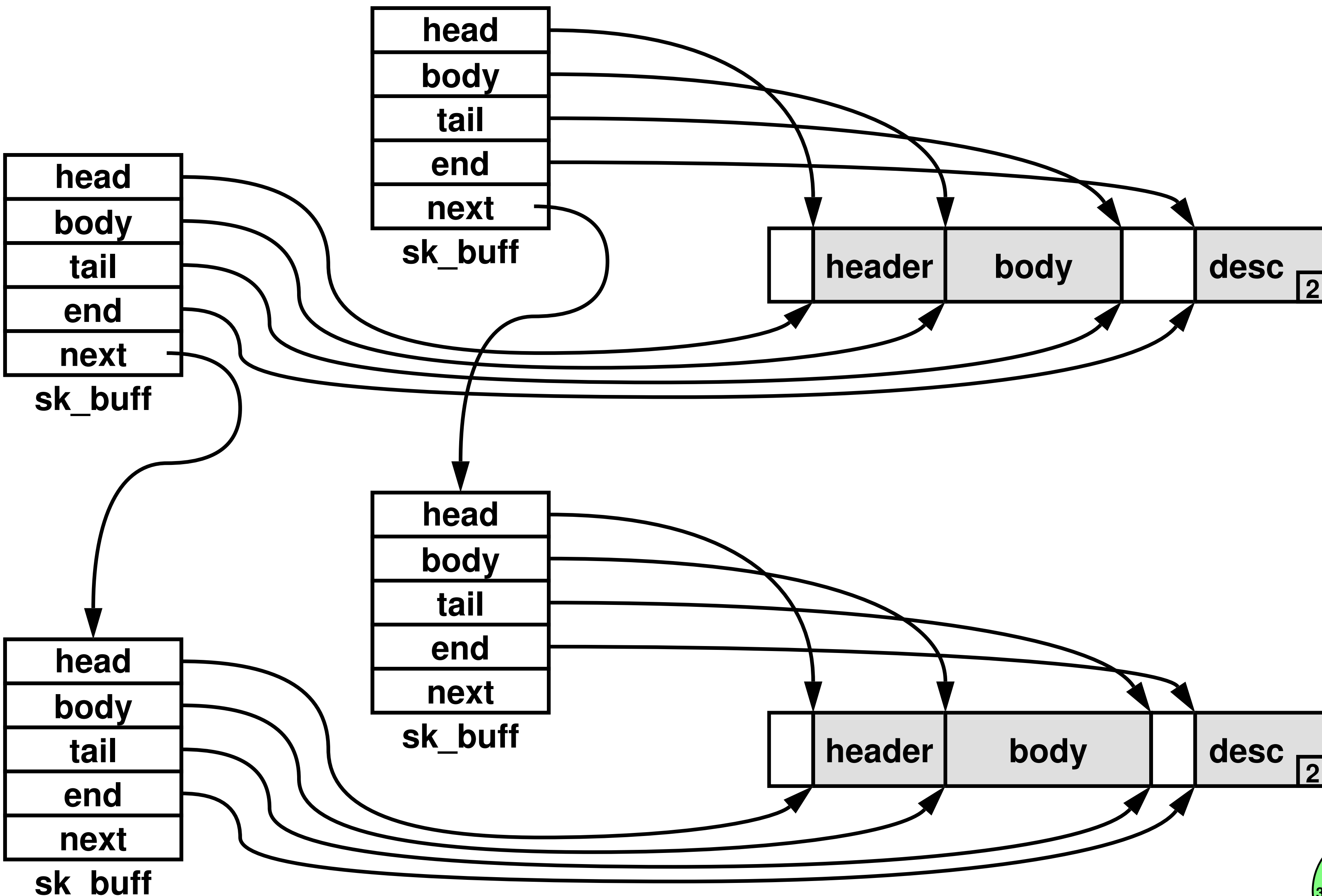
Network Communication

- Ex: TCP (details in Ch 9 which we will not cover)
- Performance challenge:
  - need to be able to pass blocks from one *module* to the next *without copying* data
    - copying came from
      - splitting data into "header" and "body"
      - copying data into application-provided buffer
  - append headers to the beginning of outgoing packets; remove headers from incoming packets *(known as layering)*
  - *hold on to packets* for possible retransmission
  - request and *respond to time-out* notifications
- To accomplish all this in our simple OS, we use a data structure adapted from Linux
  - `sk_buff` (socket buffer)
    - we will described the high-level ideas and not go into details

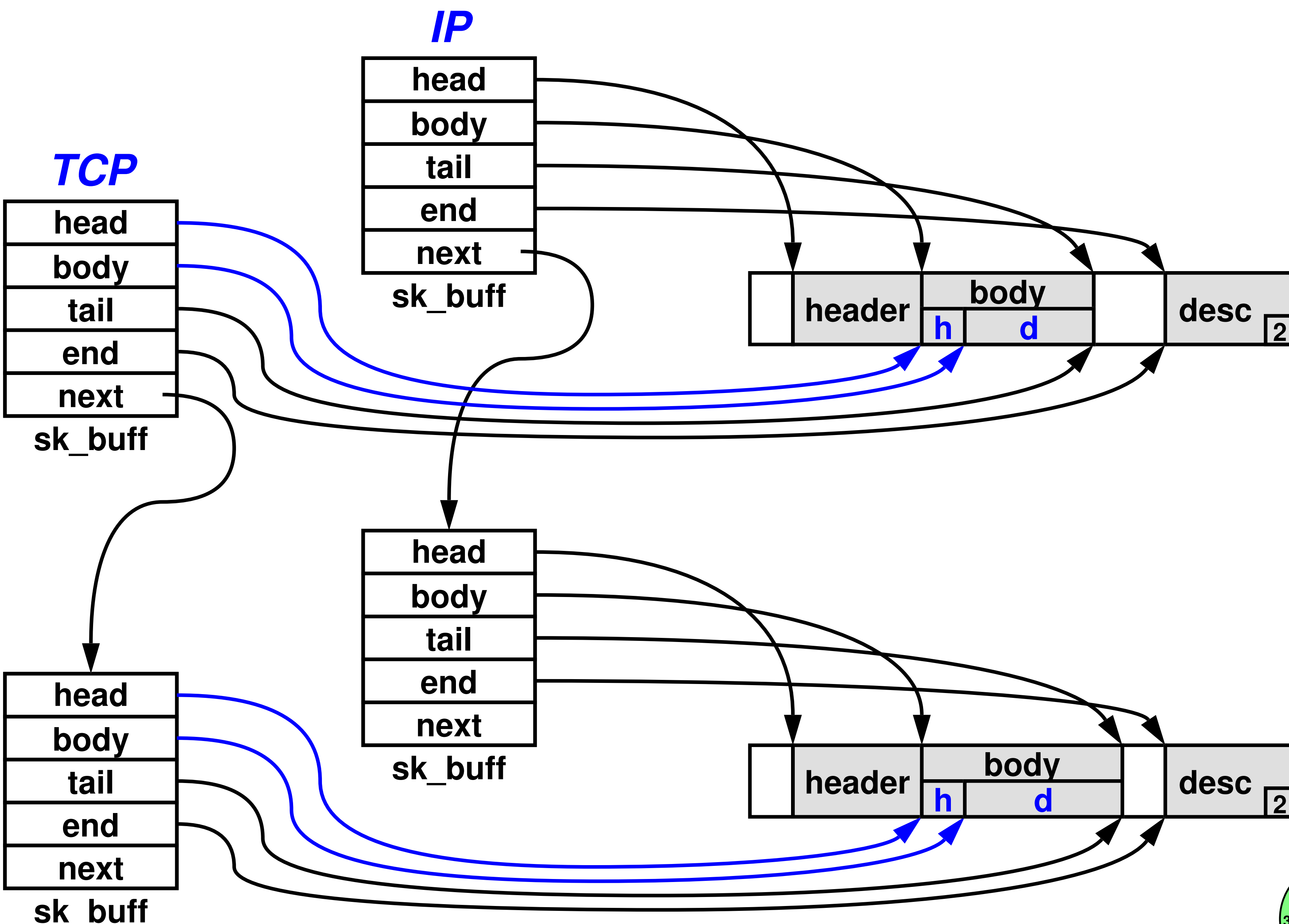
Two Queued Segments



Passed to the Another Module...



Passed to the Next Module at Higher Level...



Support Timeout

- Lots of timers in network programming!
  - if you send a message/packet that needs a response or an acknowledgement (such as in TCP internal), and
  - if it's possible for the message/packet to be lost
    - you need to set a "reasonable" timeout
- To implement *timeout*, can use a *callback* mechanism
  - use a function pointer and pass it to the interval timer
    - when timeout occurs, call the callback function
  - if the acknowledgement was received before timeout occurred
    - need to cancel the timer
    - can also specify a cancel routine

4.2 Rethinking Operating-System Structure

- Virtual Machines
- Microkernel

Monolithic Kernel

- Major advantage of monolithic kernel
  - performance
- Major down side of monolithic kernel
  - reliability (i.e., buggy kernel)
- Proposal to fix the reliability problem
  - shrink the code in "privileged mode"
- Two major approaches
  - *virtual machines*
  - *microkernel*

Virtual Machines

- A nicely designed and implemented monolithic OS is great
  - but that's not the reality
- Major problem with a monolithic OS *implementation*
  - bugs in one component can adversely affect another component
    - worse if large number of programmers contribute code
      - some coders are not as good as others
      - good coders have bad days
- Modern OSs *isolate* applications from one another
  - code executing in the privileged mode can do things the user mode code cannot
    - e.g., invoking privileged instructions
  - if you invoke a privileged instruction in user mode, you will cause a violation and trap into the kernel
- Can the same kind of *isolation* be provided for *OS components*?
  - if yes, at what cost? (there is no free lunch)

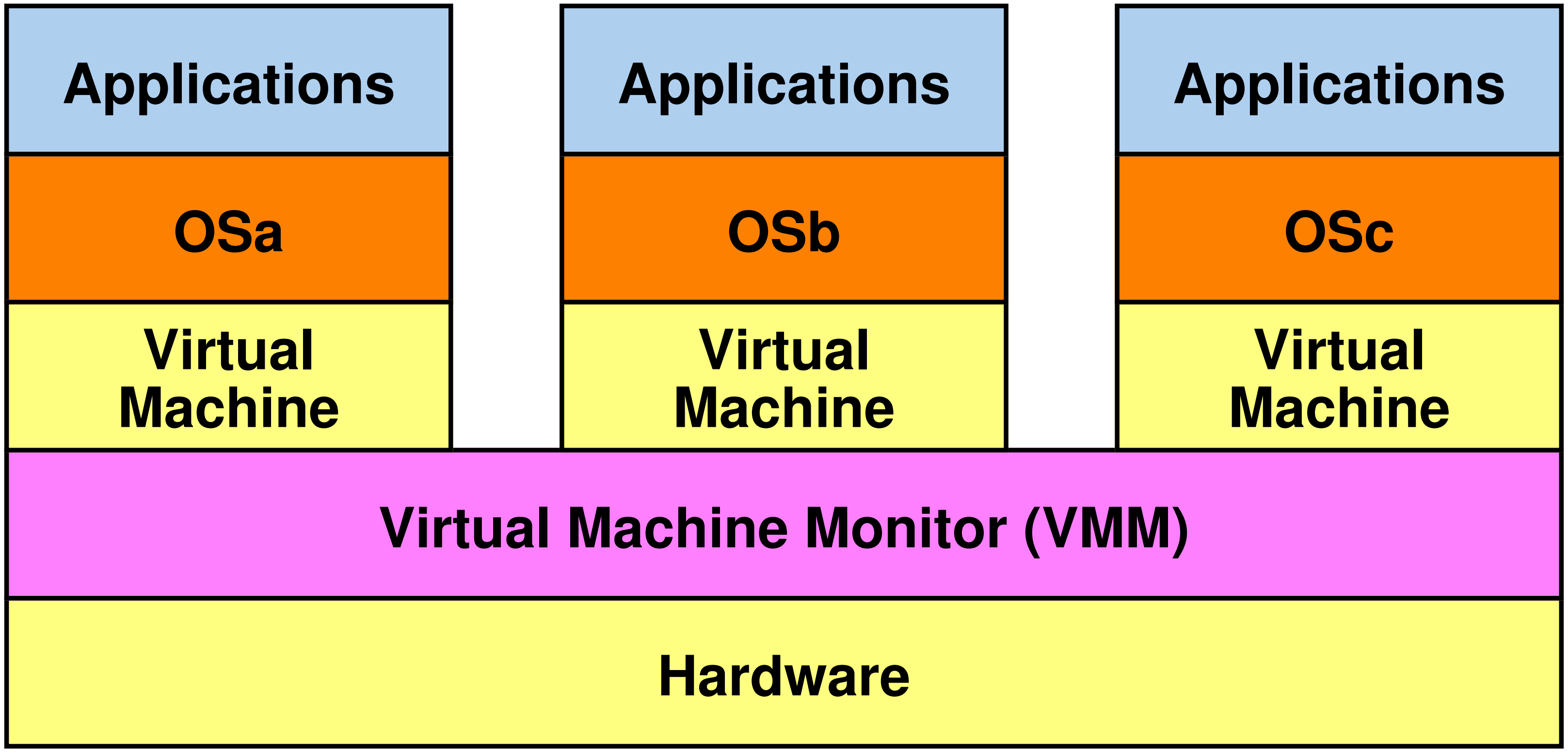
Virtual Machines
Part 1: > 50 Years Ago

- Had a different motivation

It's 1964 ...

- IBM has a *single-user time-sharing system* called CMS
  - IBM wants to build a *multiuser time-sharing system*
- TSS (Time-Sharing System) project
  - it's a very difficult system to build
  - large, monolithic system
  - lots of people working on it
  - for years
  - total, complete flop
- CP67
  - *virtual machine monitor (VMM)*
  - supports multiple virtual IBM 360s
- Put the two together ...
  - a (working) *multiuser time-sharing system*

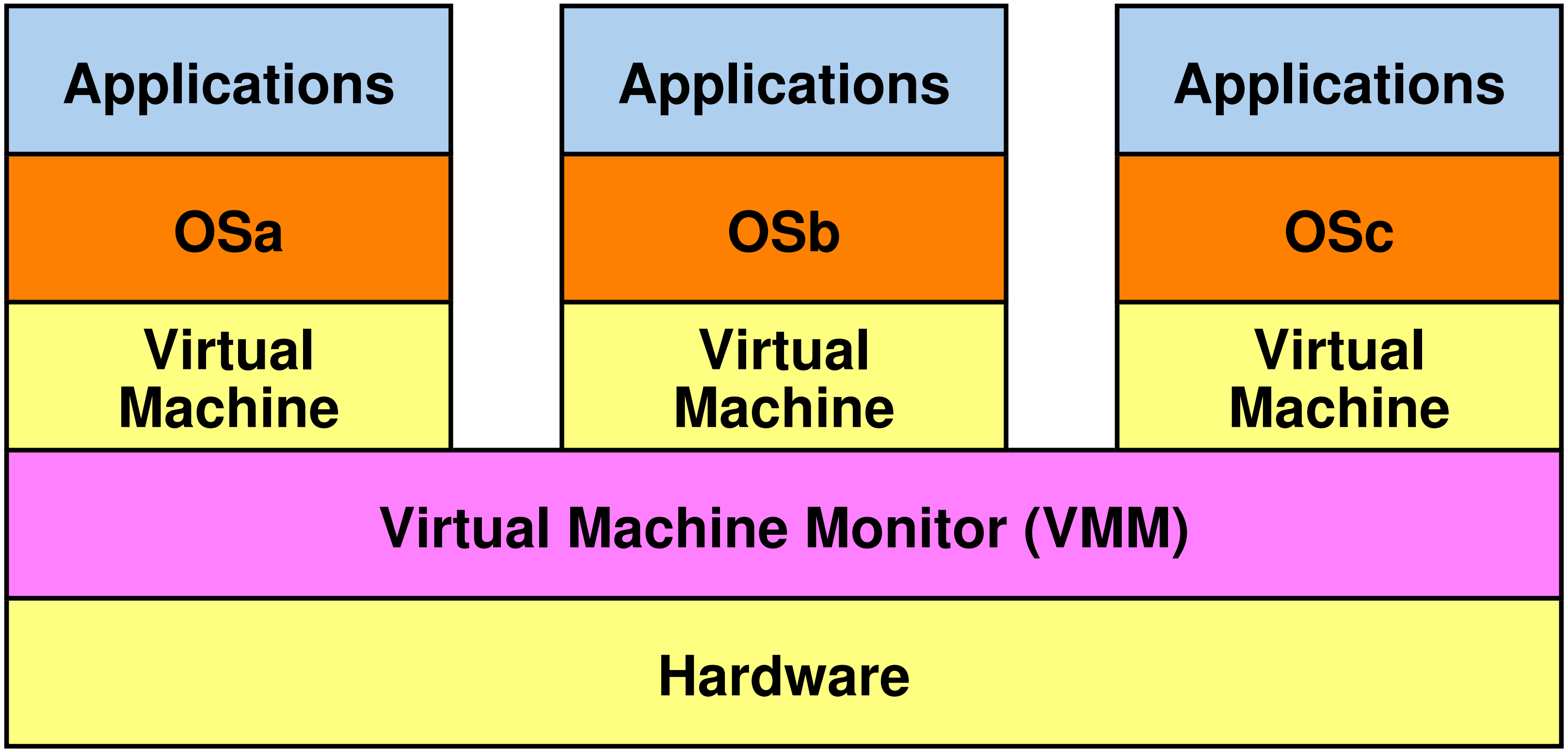
Virtual Machines



- A *"monitor"* is a *synchronization construct* that allows executing entities to have both *mutual exclusion* and the ability to *wait* (block) for a certain condition to become true

- What abstraction does a *virtual machine* provide?
  - *hardware*

Virtual Machines



- A single user time-sharing system could be developed independently of the VMM
  - and it can be tested on a real machine (which behaves identical to the VM)
  - no ambiguity about the interface VMM must provide to its applications - *identical* to the *real machine!*

Virtual Machines

- What is considered a virtual machine (for this class)?
  - *run* (not emulate/simulate) OSx "inside" / "on-top-of" OSy
    - we will refer to OSx as the "guest OS" and OSy as the "host OS" (these terms came from VMware)
    - a virtual machine is *not* an OS *emulator*
      - must *execute* "guest OS" on the real CPU *directly*
  - make "guest OS" think that it's running on hardware, but in reality, it is running inside a *virtual machine*
    - therefore, the *code* and *data structures* you put into "host OS" so that you can run "guest OS" in it is called "virtual machine"
  - "host OS" may be a specialized OS

- Different types of virtualization technologies
  - *pure virtualization:* "guest OS" is unmodified
    - "guest OS" thinks it's running directly on hardware
  - *para-virtualization:* "guest OS" is modified
    - modified "guest OS" can only run inside virtual machine
  - something else: we *shouldn't* call it a virtual machine