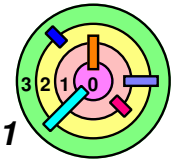
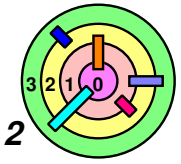
4.1 A Simple System (Monolithic Kernel)

- A Framework for Devices
- *Low-level Kernel*
- Processes & Threads
- Storage Management



Low-Level Kernel

- Let's talk about how devices are handled, starting at the lowest levels of the kernel
  - (although ***bottom-up*** is not a good way to ***design*** an OS)
    - but it may be a reasonable way to implement OS components
- We will start by looking at two such devices
  - terminals
  - network communication

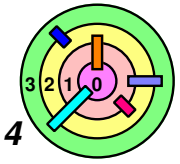


Terminals

- VT100

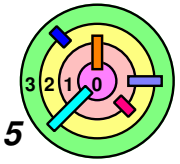
Terminals

- Long obsolete, but still relevant
  - on Linux, you would probably use a "terminal" program (such as `xterm` or `gnome-terminal`) to interact with the system
  - on Windows, `putty` or `xwin-32` brings up a "terminal" for you to interact with a remote system
    - `ssh` client program interact with `sshd` on a server
      - once authenticated, `sshd` forks to exec `tcsh`/`bash`
    - you login session is *on the target machine*
      - i.e., if you login as root and type "`halt`", you would halt the machine!
- How to interact with a *terminal device*?
  - characters to be displayed are simply sent to the output routine of the *serial-line driver*
  - to fetch characters that have been typed at the keyboard, a call can be made to its input routine
  - as it turns out, not to straight-forward

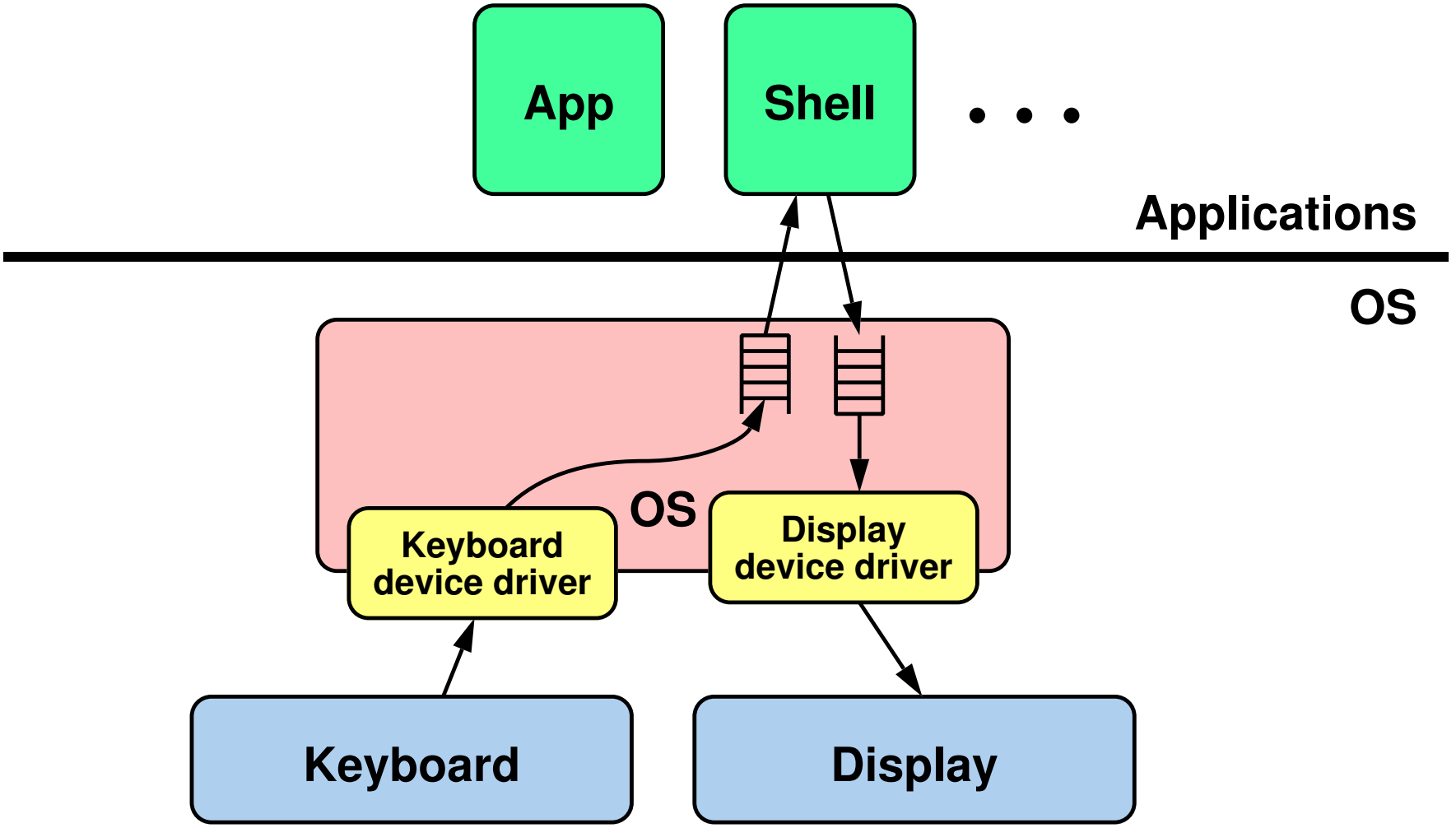


Terminals

- In implementing a device driver, need to take *device-specific characteristics* into account
  - but how *device-specific* does it have to be?
- Issues
  1) terminals are slow and characters generation are too fast
     - need to tell the application to slow down and wait for the terminal to catch up
     - so, we need an *output buffer* to buffer the output and send characters to the terminal from the buffer
     - we have an instance of the producer-consumer problem!
  2) characters arrive from the keyboard even though there isn't a waiting read request from an application
     - so, we need an *input buffer* to buffer incoming characters and wait for an application to issue a read request
     - we have another instance of the producer-consumer problem!

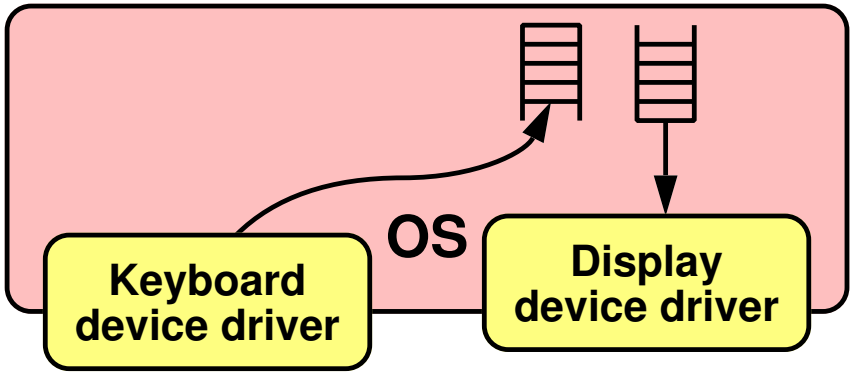


Terminals



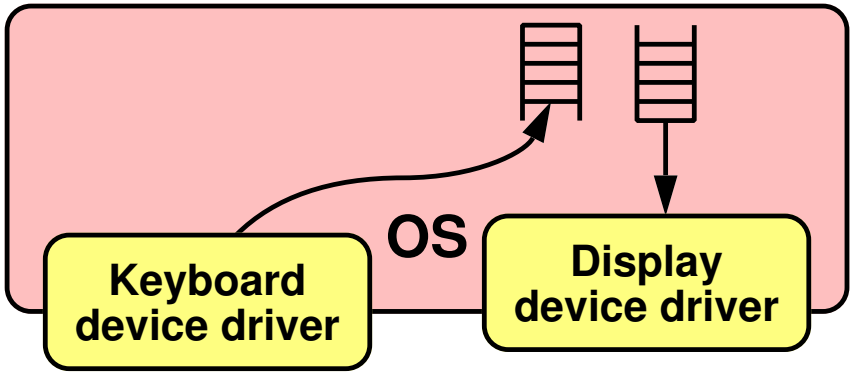
Terminals

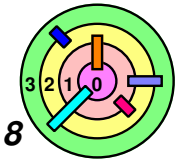
- To deal with concerns (1) and (2)
  - use two queues, one for input and one for output
  - characters are placed on the output queue and taken from the input queue in the context of the *application thread*
    - i.e., application write to the output queue and read from the input queue
    - a thread producing output would block if output queue is full
    - a thread consuming input would block if input queue is empty
  - what about the other ends of these queues? who are handling them?



Terminals

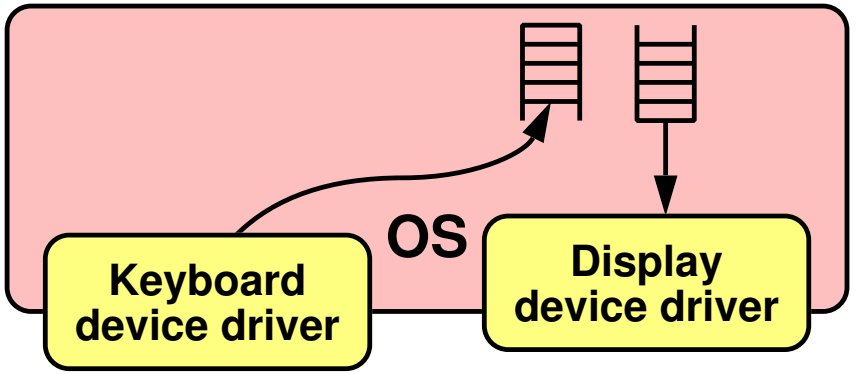
- To deal with concerns (1) and (2)
  - how about using a keyboard reading thread (that would do the following)?
    1) issue a read to the device
    2) block itself and wait for interrupt from the device
    3) when interrupt occurs, the thread is woken up
    4) the thread reads from the device and move one character from the device to the input queue
    5) goto step 1
  - this approach of using *thread context* seems to be an overkill and may be inefficient





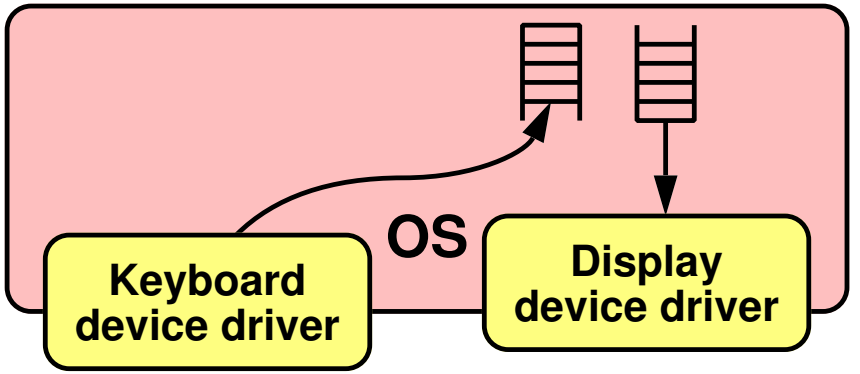
Terminals

- To deal with concerns (1) and (2)
  - how about just using an *interrupt handler*?
    - in the *read-completion interrupt*, the handler moves one character from the device to the input queue and issue another read request to the device and blocks
      - if the queue is full, the character is thrown away
      - the application thread must *mask interrupts* when it's taking a character from the queue
  - can do the same for the output queue ...



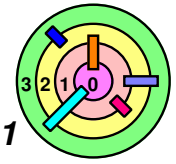
Terminals

- To deal with concerns (1) and (2)
  - how about just using an *interrupt handler*?
  - can do the same for the output queue
    - in the *write-completion interrupt*, the handler moves one character from the output queue to the device and issue another write request to the device and blocks
      - if the application writes to an *empty queue*, it would *setup the write-completion interrupt handler* and issue a write request to the device



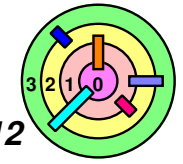
Terminal

- Additional issue
  3) input characters may need to be processed in some way before they reach the application
     - e.g., characters typed at the keyboard are *echoed* back to the display
     - characters may be grouped into lines of text and subject to simple *editing* (such as backspace)
     - some applications prefer to process all characters themselves, including their editing



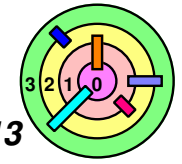
Terminals

- To deal with concern (3)
  - remember, once you allow an application to take a character, you cannot ask for it back
    - can only give it to the application when there is no chance that you will want it back
      - this happens when a line is completed
  - therefore, we need *two input queues*
    - one for the *partial-line*
      - subject to editing
    - the other contain characters from *completed lines*
    - in the read-completion interrupt, the handler moves one character from the device to the *partial-line queue*
      - if the input character is a carriage-return, the entire content of the partial-line queue is moved to the completed-line queue



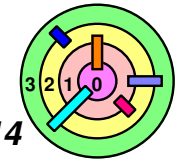
Terminals

- To deal with concern (3)
  - when a character is typed, it should go to the display immediately (due to the *echoing* requirement)
    - it may be competing with the output thread, but it's okay (and that's how it's done in Unix)
    - Windows handle this differently
      - typed characters are only echoed when an application consumes them
      - therefore, echoing is not done in the interrupt context
      - echoing is done in the context of the thread consuming the characters (i.e., the one that calls `read()`)

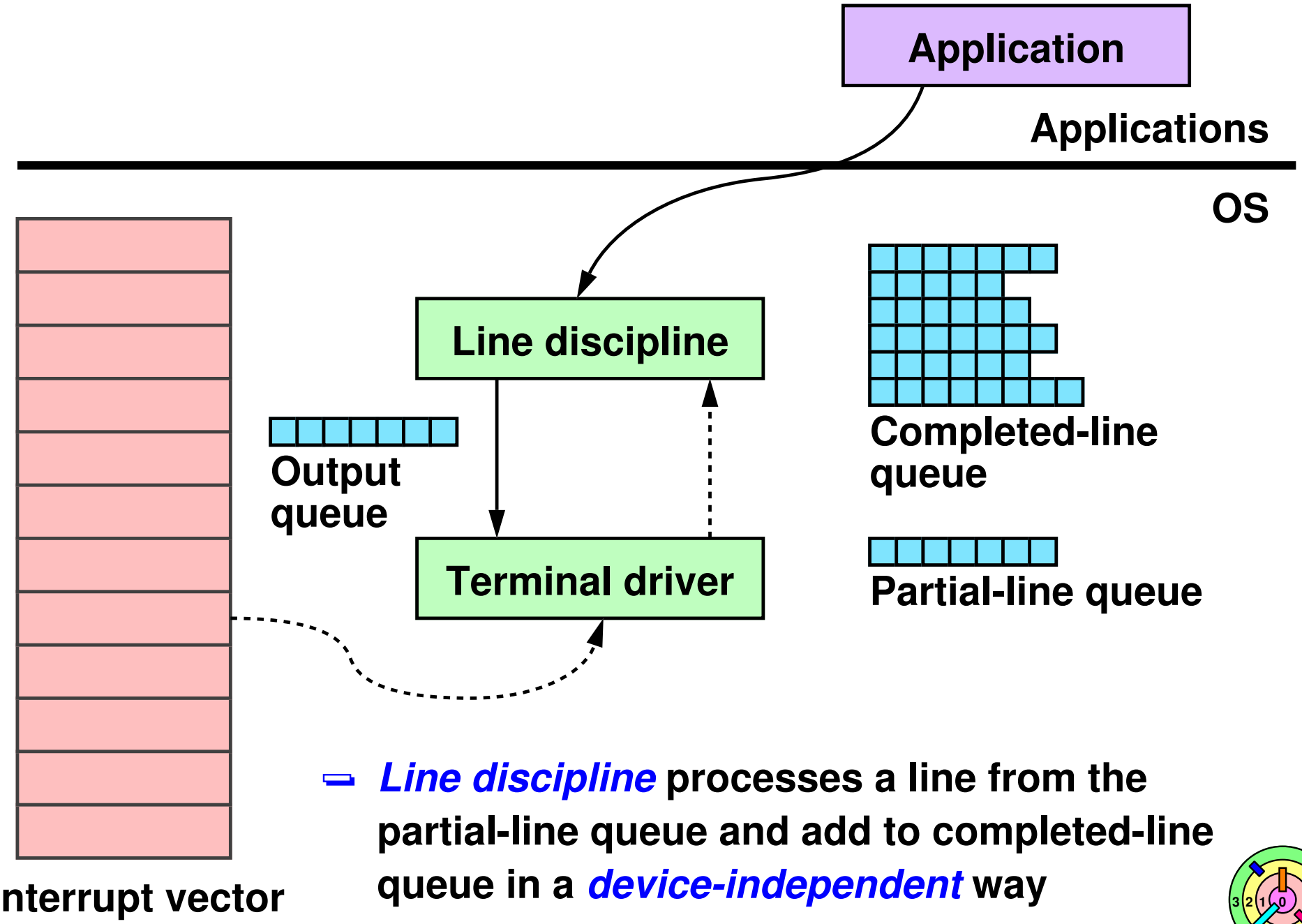


Modularization

- ***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

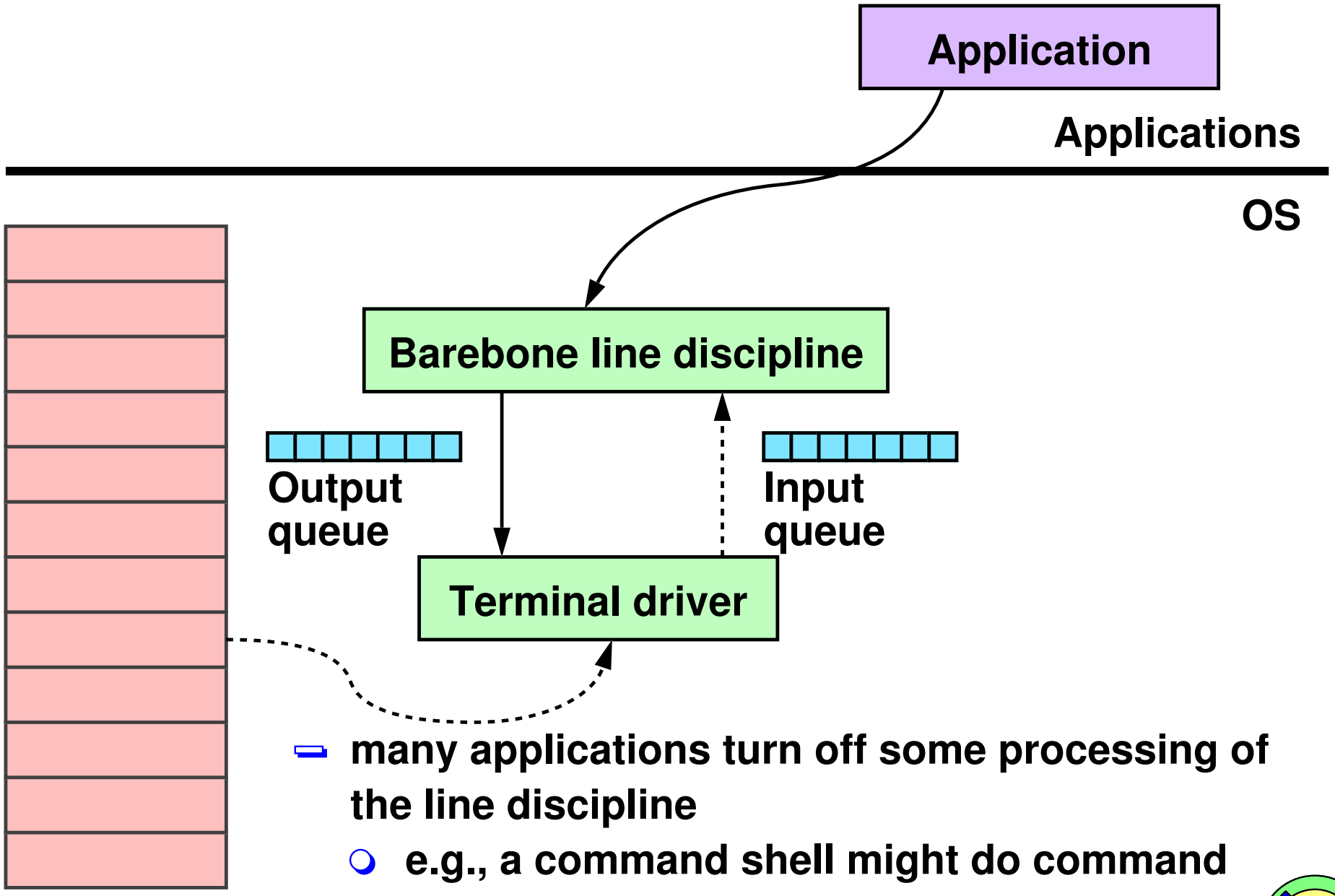


Terminals



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

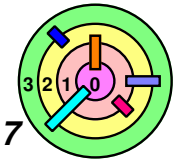
Terminals



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

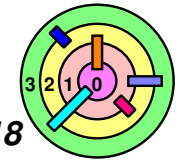
Where to Put the Modules?

- 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

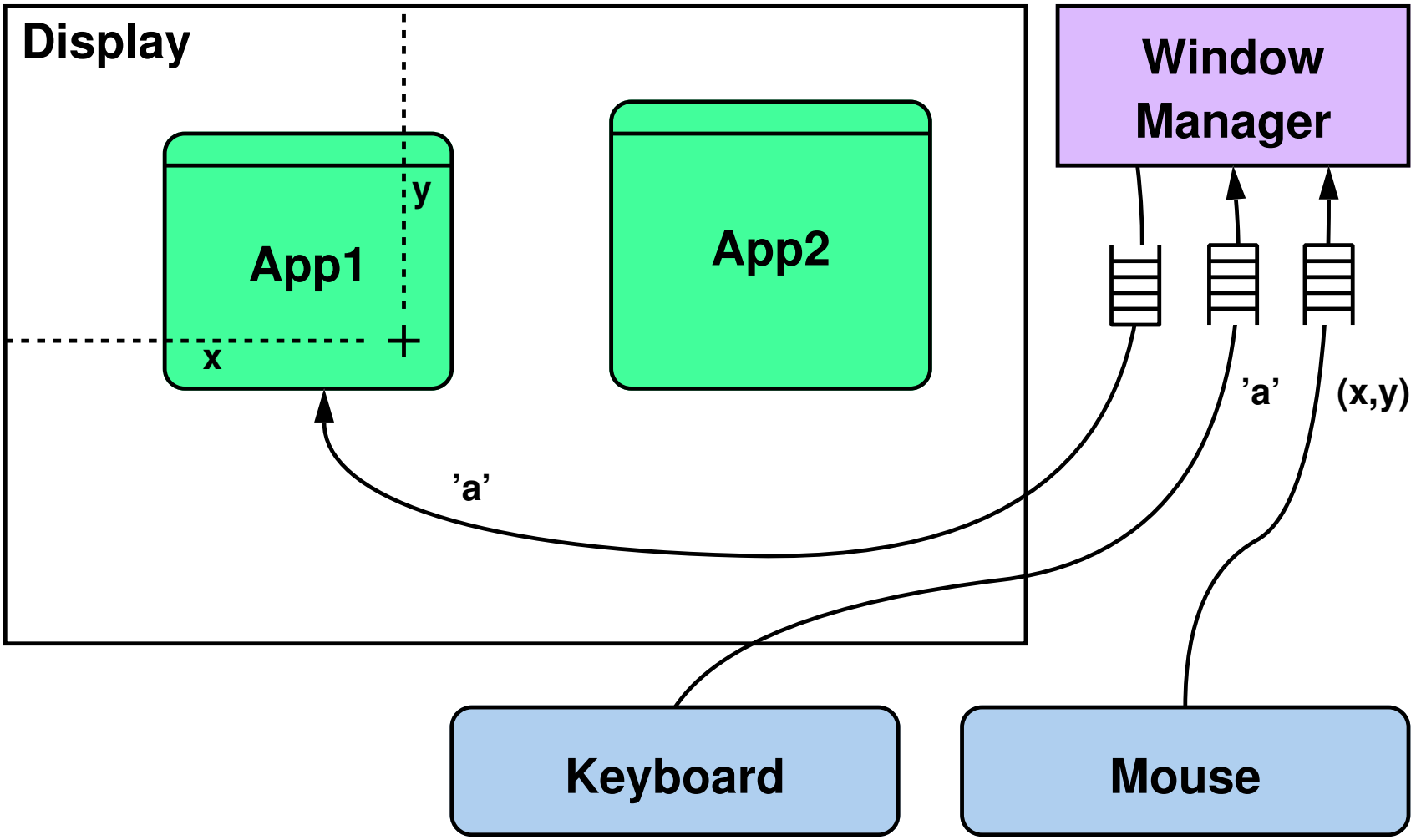


Terminals and Pseudo Terminals

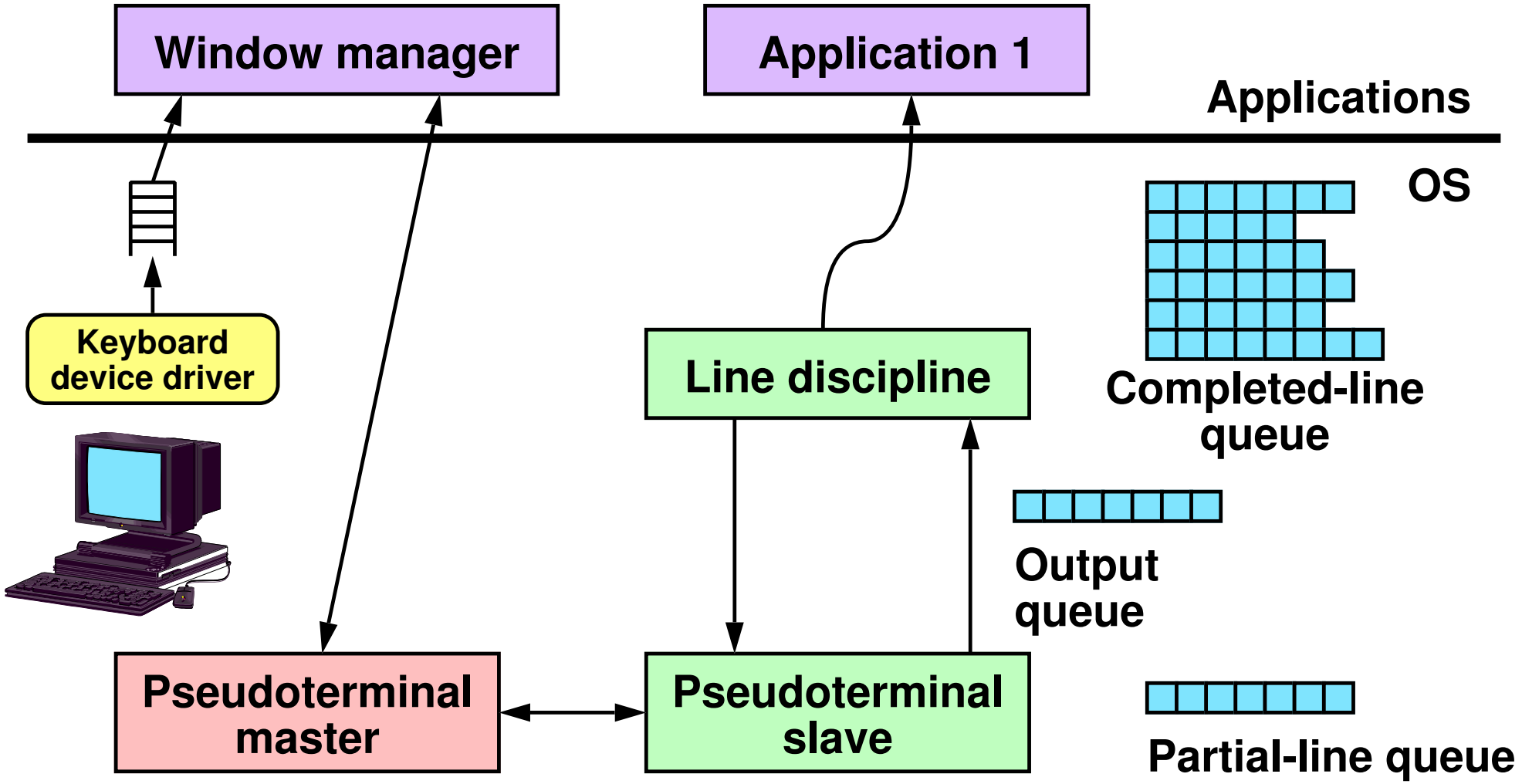
- Modern systems do not have terminals
  - they often have bit-mapped displays, keyboards and mice connected via USB
  - a *window manager* implements windows on the display and determines which applications receive typed input *(input focus)*
  - a server might support remote sessions where applications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



Bitmapped Display

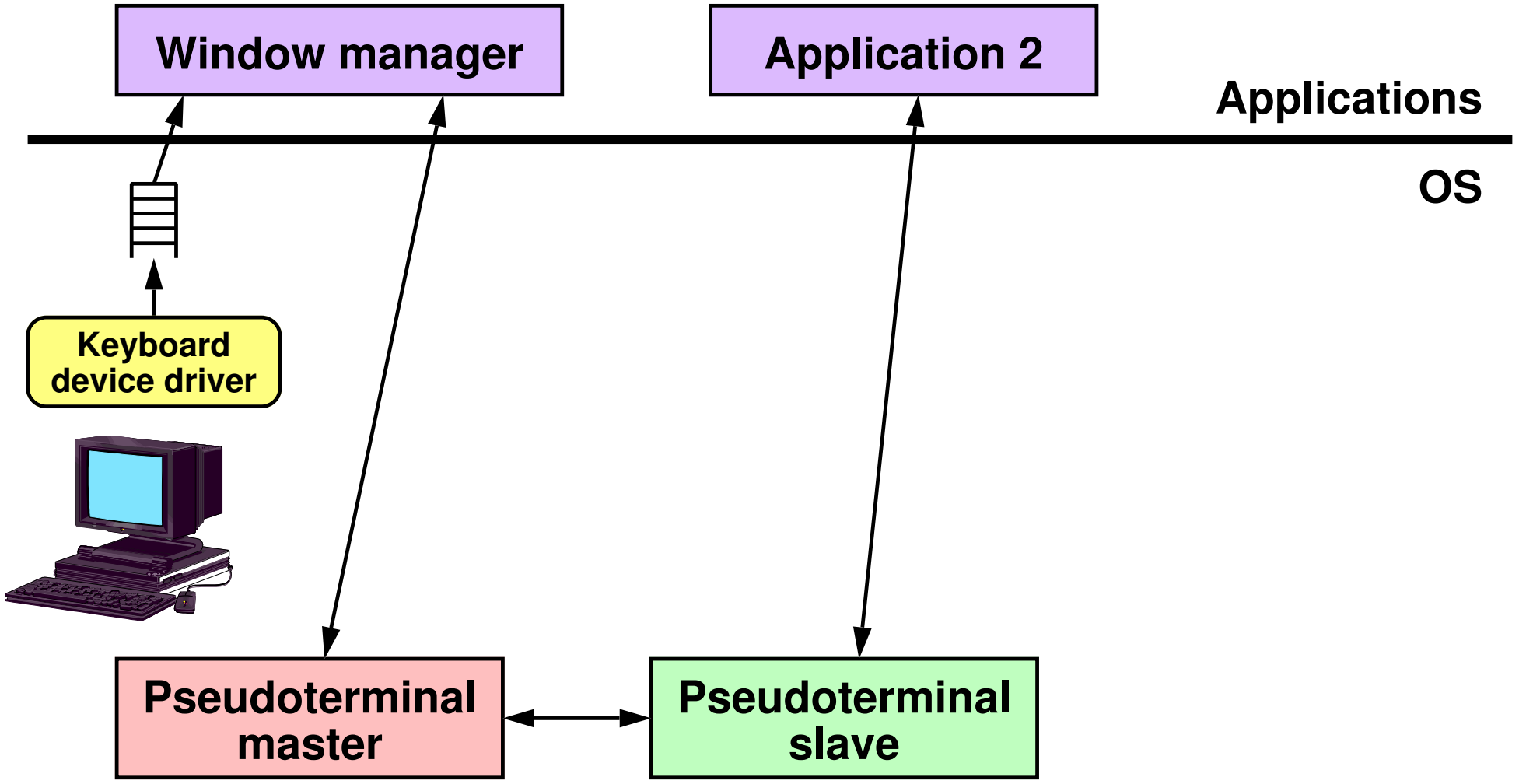


Pseudo Terminals



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

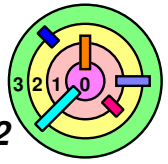
Pseudo Terminals



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

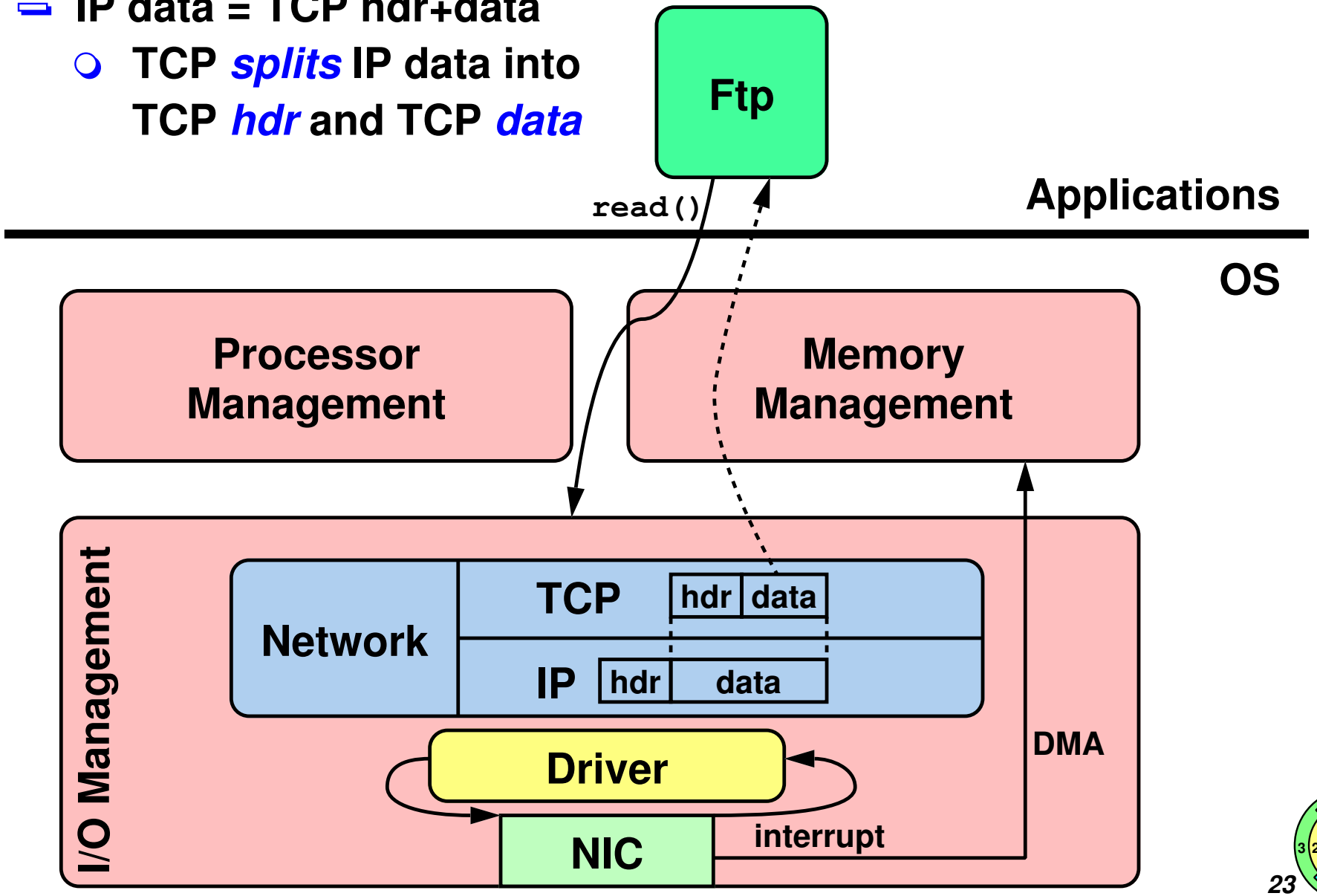
Network Communication

- 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 + data* in higher layer
  - cannot afford to copy network data from queue to queue!
    - no copying allowed!
    - must pass by memory address!



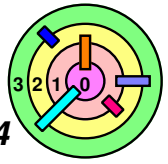
Network Communication

- Ex: TCP (details in Ch 9)
  - IP data = TCP hdr+data
    - TCP *splits* IP data into TCP *hdr* and TCP *data*

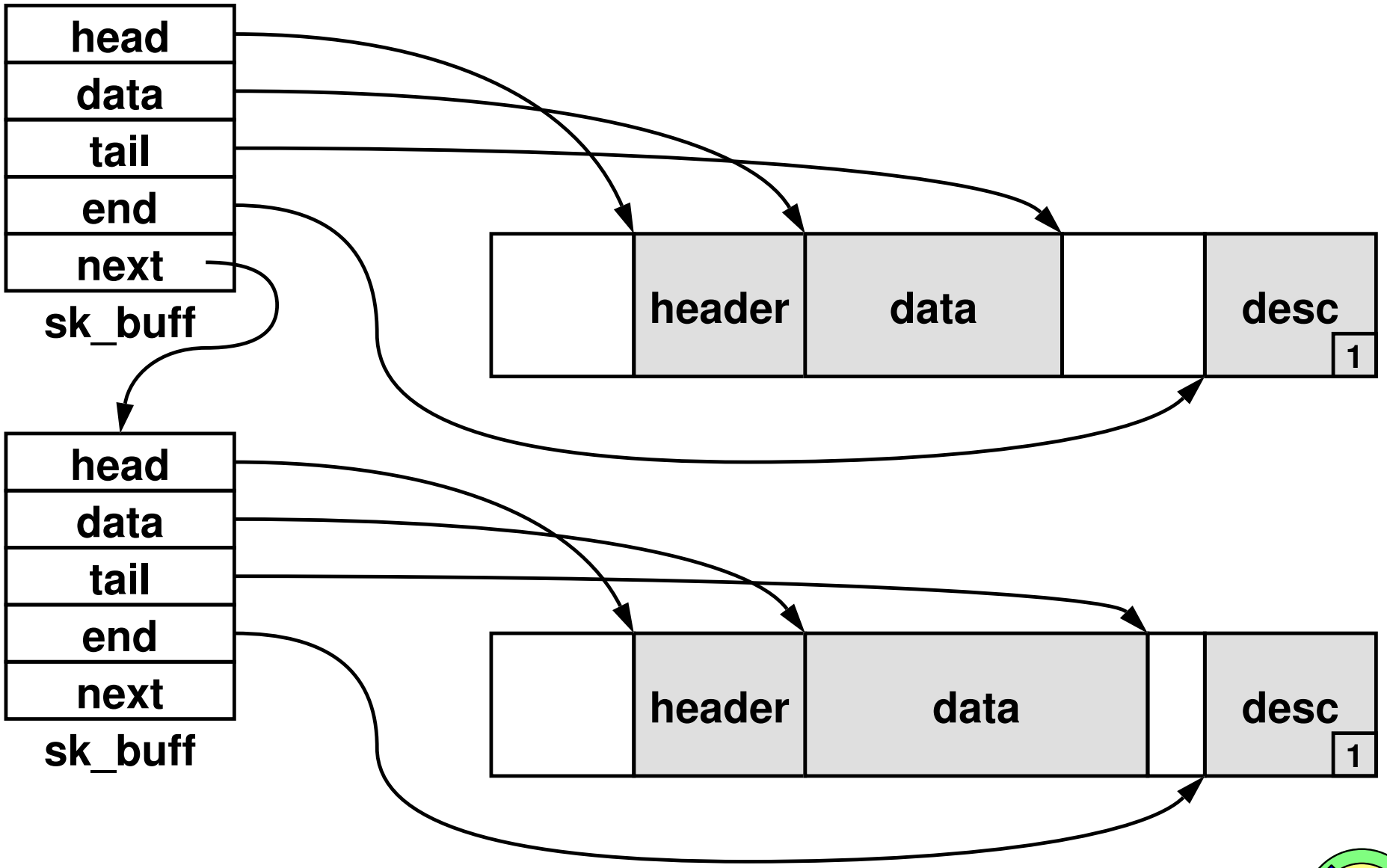


Network Communication

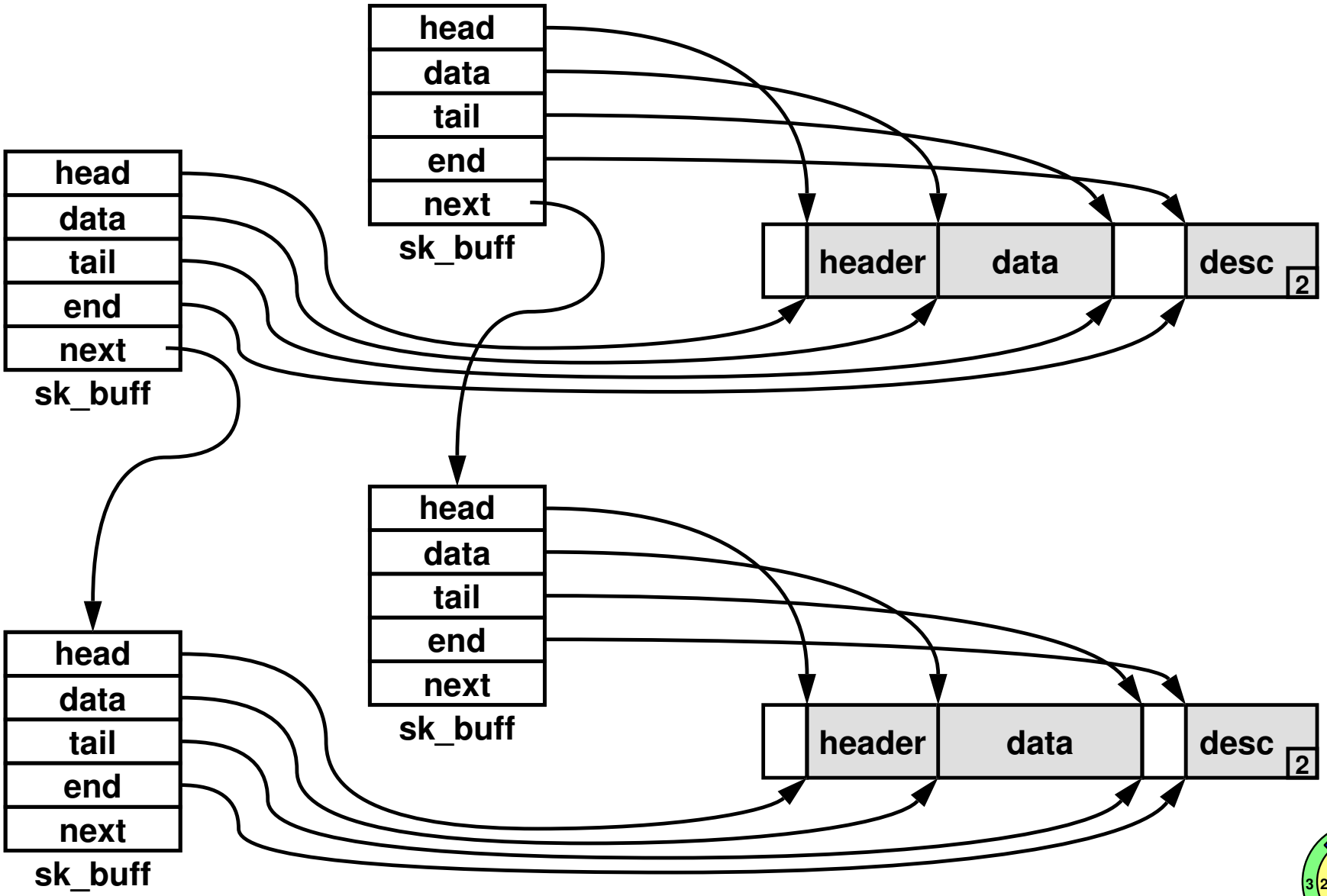
- Ex: TCP (details in Ch 9)
- Performance challenge:
  - need to be able to pass blocks from one *module* to the next *without copying* data
    - copying came from
      - splitting "header" and "data"
      - copying data into application-provided buffer
  - append headers to the beginning of outgoing 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)

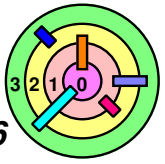


Two Queued Segments

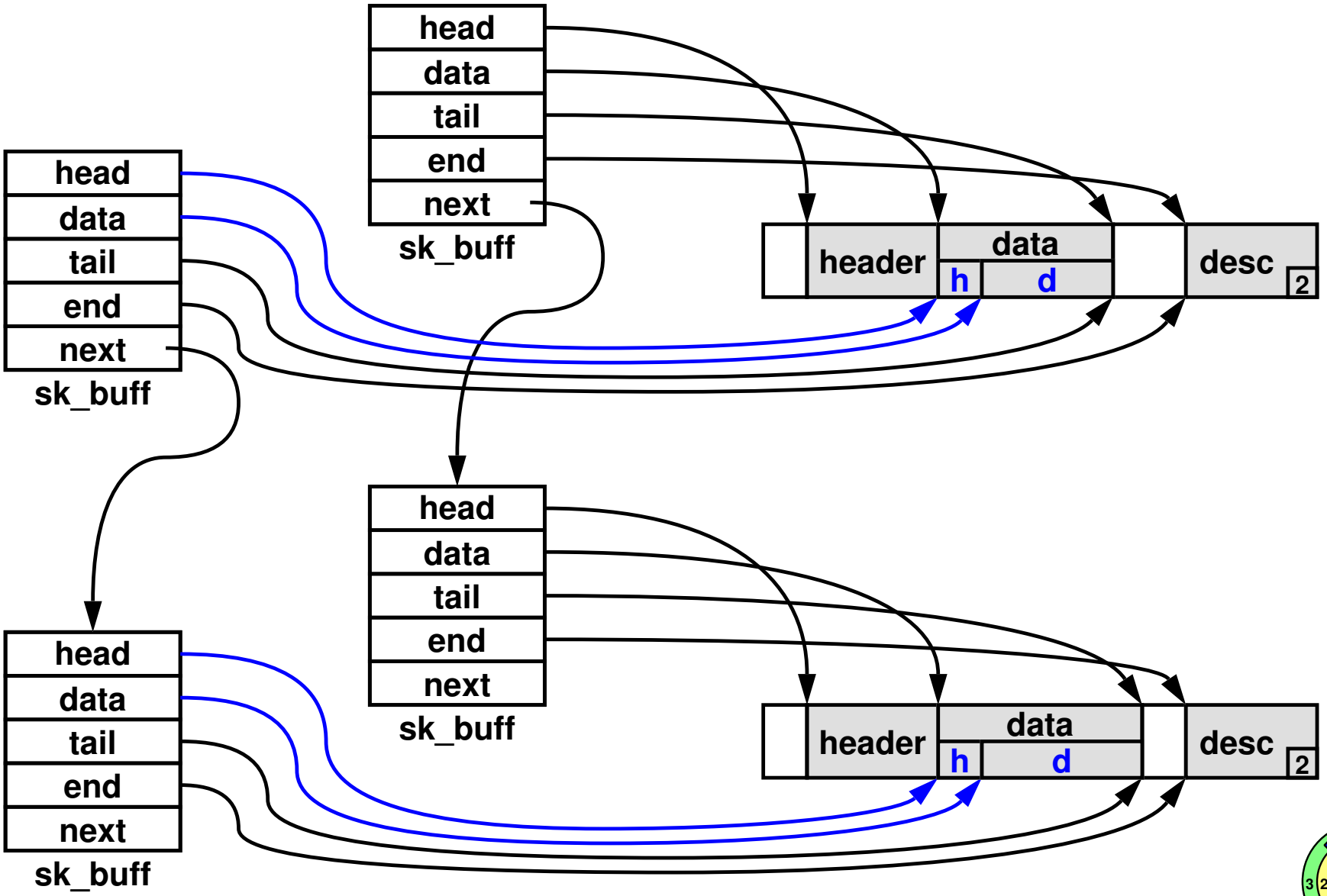


Passed to the Another Module...





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