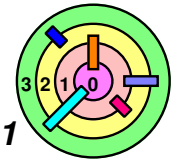
Ch 4: Operating-System Design

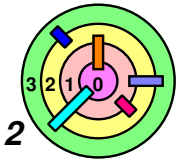
Bill Cheng

http://merlot.usc.edu/william/usc/



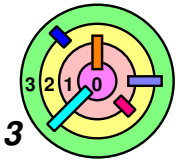
OS Design

- We will now look at how OSes are constructed
  - what goes into an OS
  - how they interact with each other
  - how is the software structured
  - how performance concerns are factoered in
- We will introduce new components in this chapter
  - scheduling (Ch 5)
  - file systems (Ch 6)
  - virtual memory (Ch 7)
- We will start with a simple hardware configuration
  - what OS is needed to support this
- Applications views the OS as the "computer"
  - the OS needs to provide a *consistent* and *usable interface*
    - while being *secure* and *efficient*
  - that's a pretty tall order!



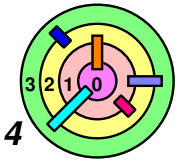
OS Design

- Our goal is to *build a general-purpose OS*
  - can run a variety of applications
    - some are interactive
    - many use network communication
    - all read/write to a file system
  - it's like most general-purpose OSes
    - Linux
    - Solaris
    - FreeBSD
    - Mac OS X
    - Chromium OS (has a Linux kernel)
    - Windows (the only one that's not directly based on Unix)
  - all these OSes are quite similar, functionally!
    they all provide:
    - processes
    - threads
    - file systems
    - network protocols with similar APIs
    - user interface with display, mouse, keyboard
    - access control based on file ownership and that file owers can control

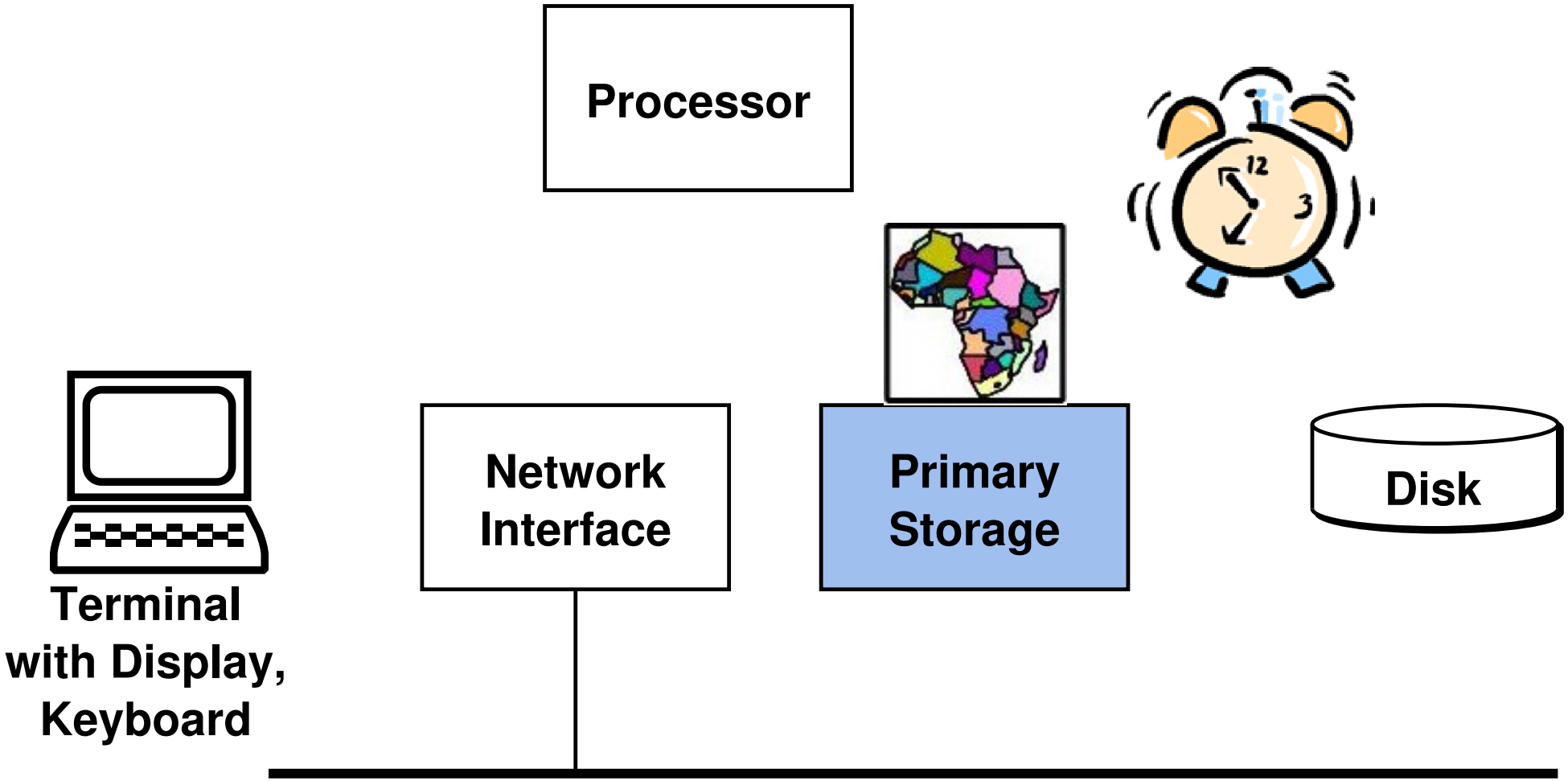


OS Design Issues

- **Performance**
  - efficiency of application
- **Modularity**
  - tradeoffs between modularity and performance
- **Device independence**
  - for new devices, don't need to write a new OS
- **Security/Isolation**
  - isolate OS from application

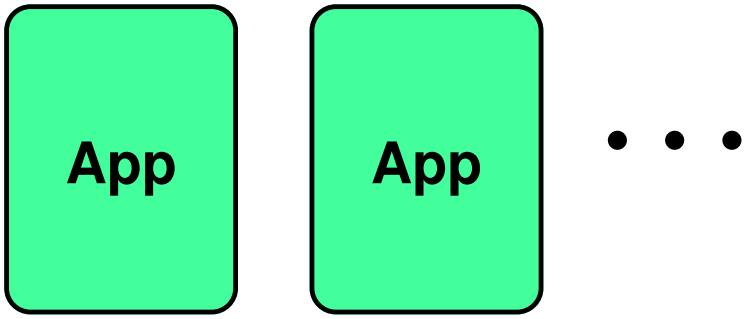


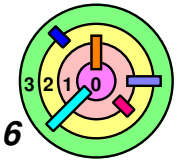
Simple Configuration



- Early 1980s OS, so we can focus on the basic OS issues
  - no support for bit-mapped displays and mice
  - generally *less efficient* design

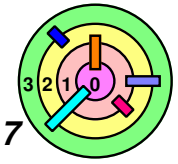
OS Components



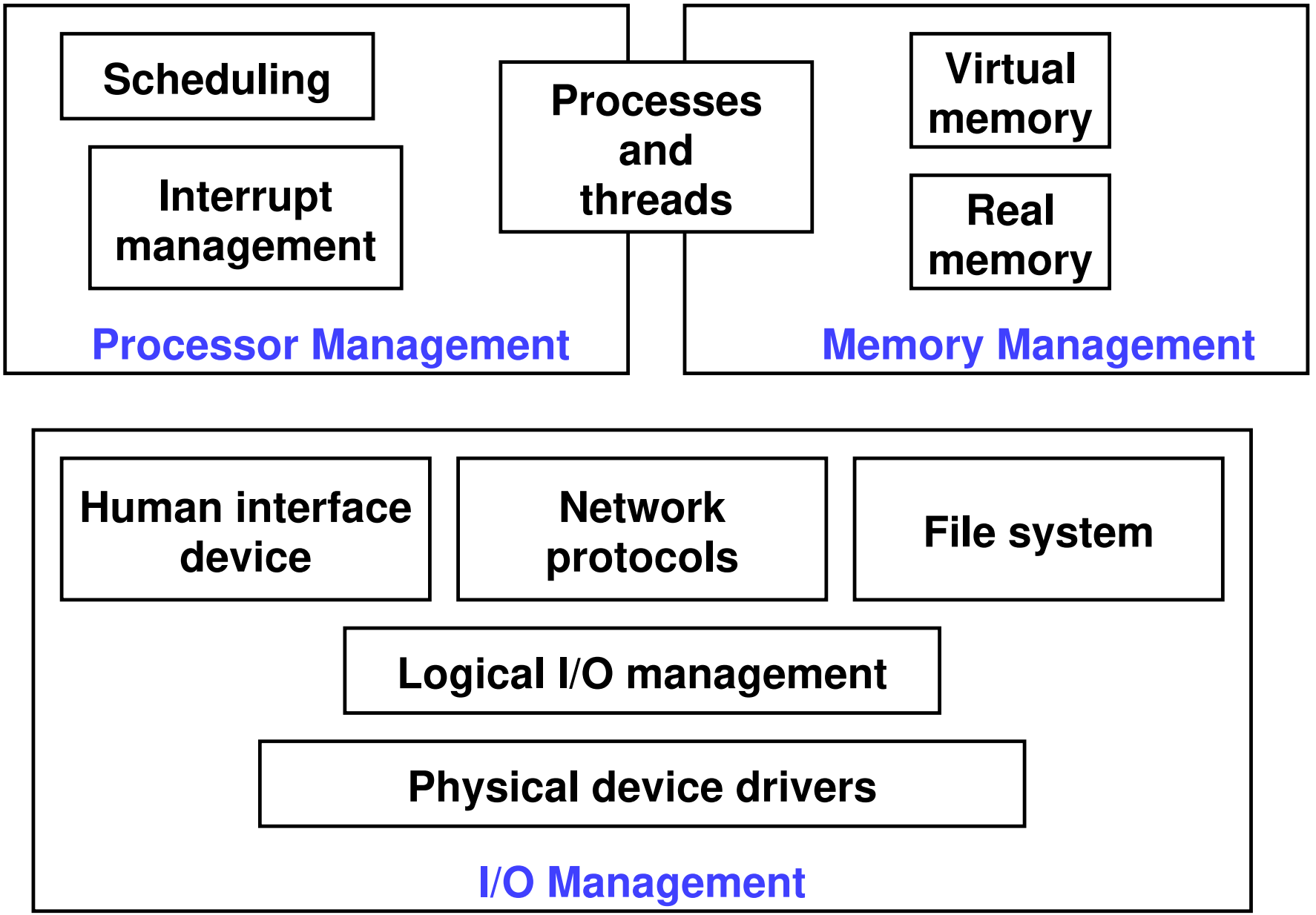


A Simple System: To Be Discussed

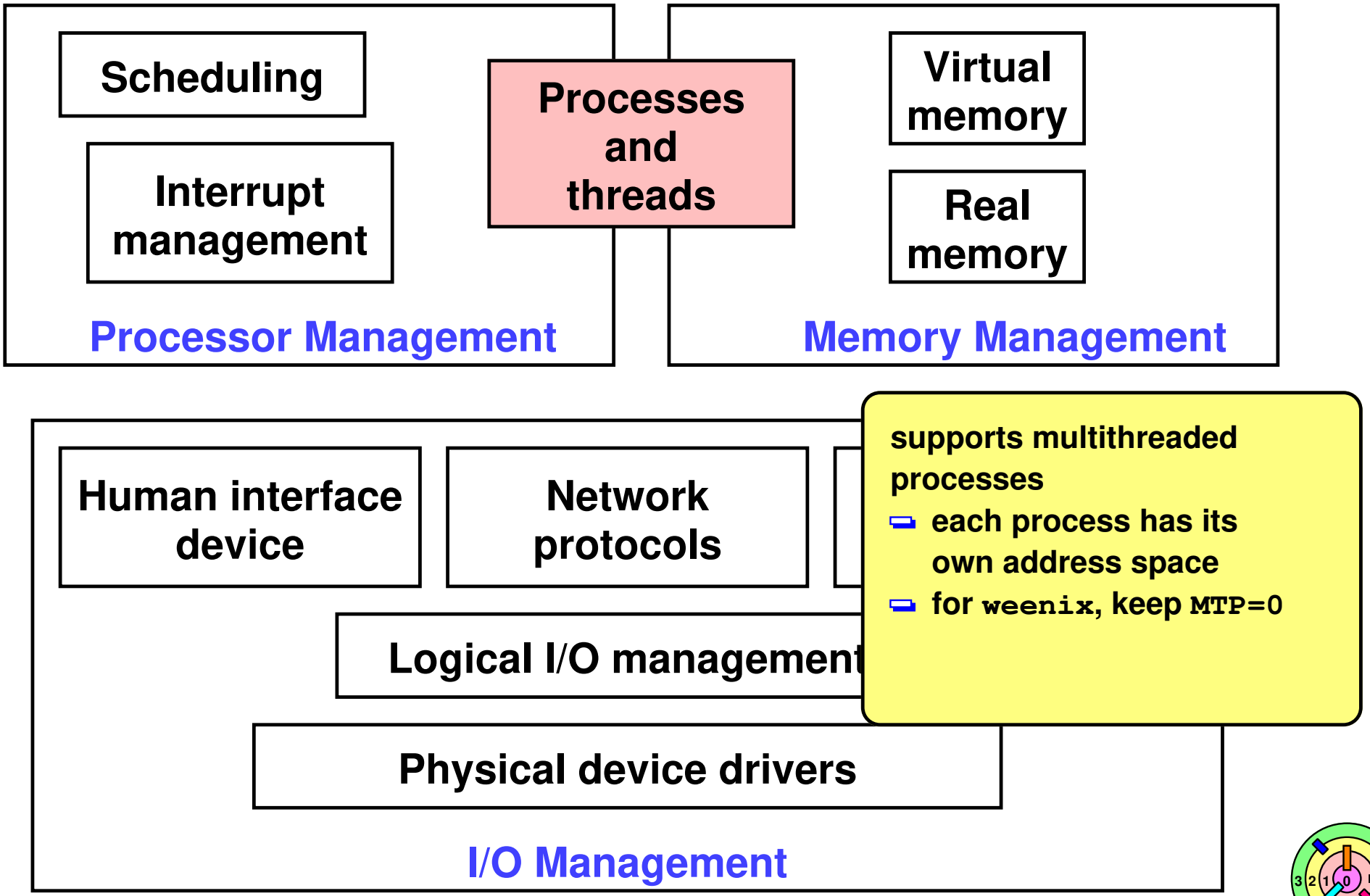
- What is the functionality of the components?
- What are the key data structures?
- What mechanisms are there to support the applications?
- How is the system broken up into modules?
- To what extent is the system extensible?
- What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- In which execution contexts do the various activities take place?
  - e.g., thread context vs. interrupt context

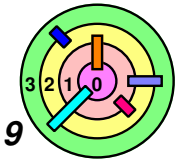


OS Components

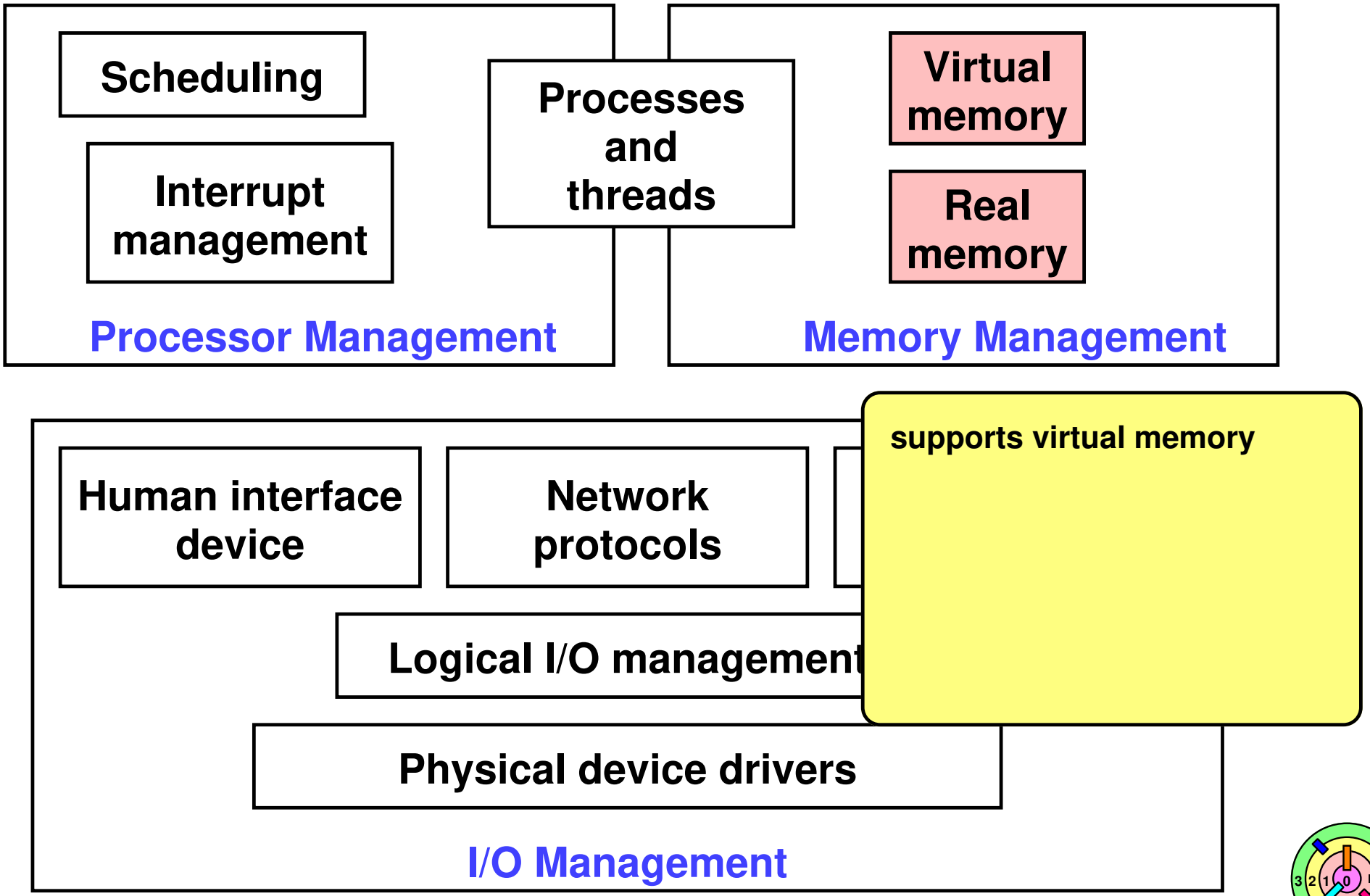


OS Components

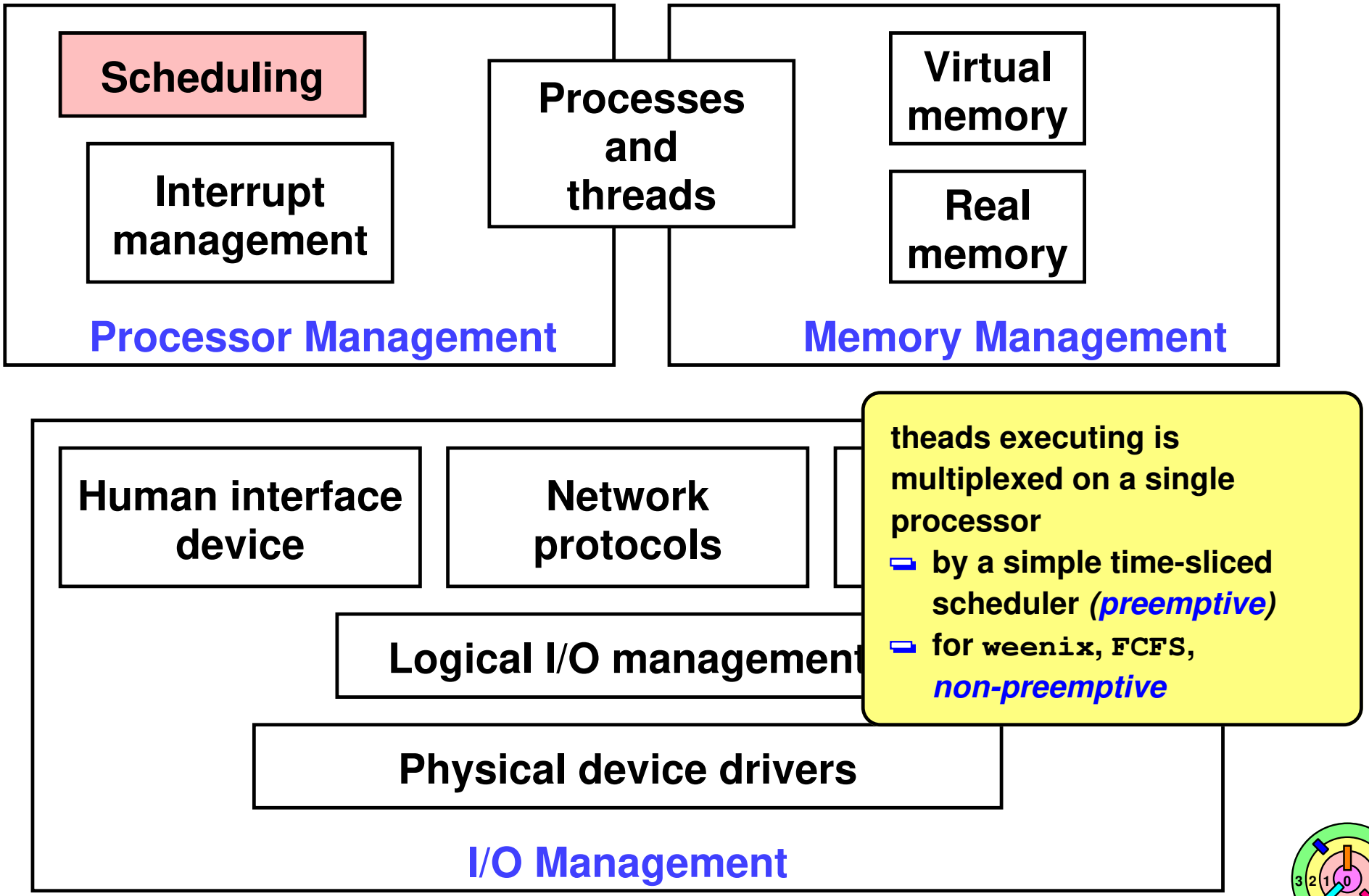




OS Components

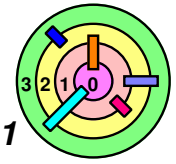


OS Components



theads executing is multiplexed on a single processor

- by a simple time-sliced scheduler *(preemptive)*
- for `weenix`, `FCFS`, *non-preemptive*



OS Components

has a file system
- layered on disks

Processes and threads

Virtual memory

Real memory

ment

Memory Management

Human interface device

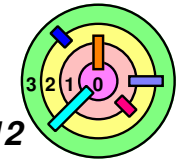
Network protocols

File system

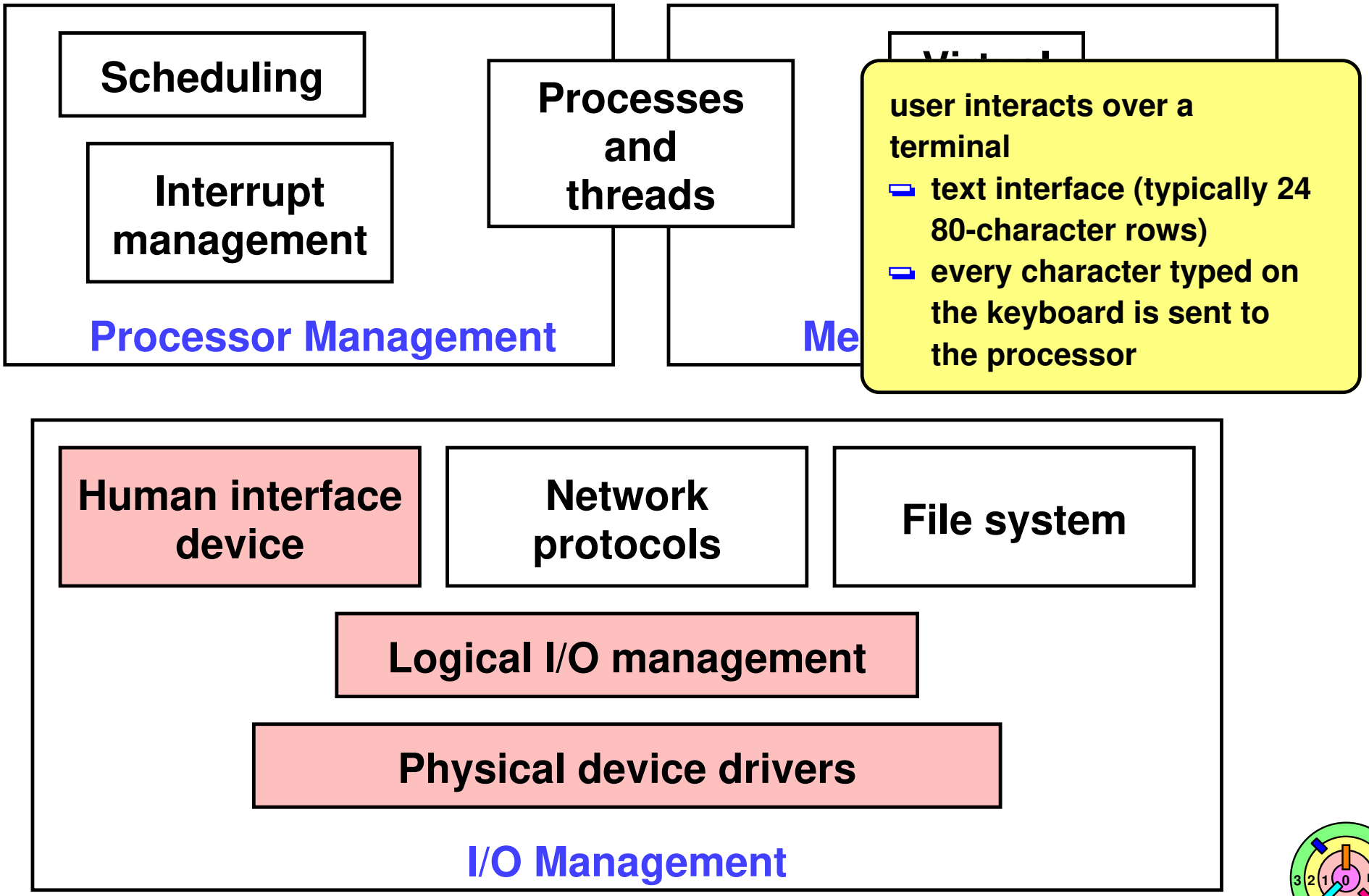
Logical I/O management

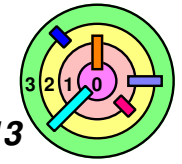
Physical device drivers

I/O Management

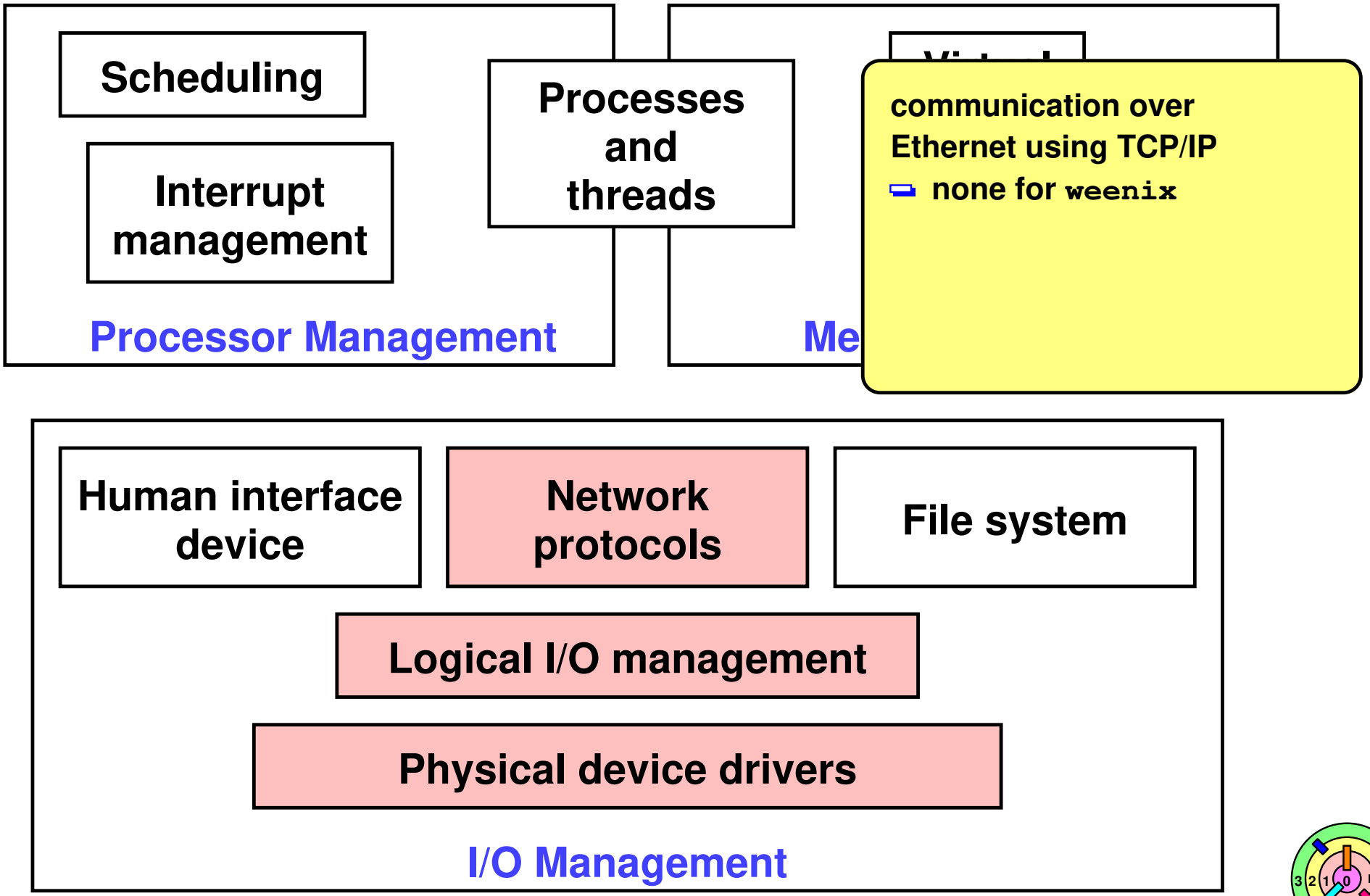


OS Components



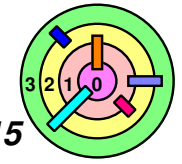


OS Components



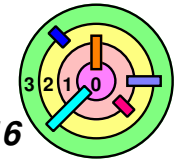
Some Important OS Concepts

- From an application program's point of view, our system has:
  - processes with threads
  - a file system
  - terminals (with keyboards)
  - a network connection
- Need more details on these... Need to look at:
  - how can they be provided
  - how applications use them
  - how this affects the design of the OS



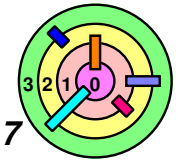
Processes And File Systems

- The purpose of a *process*
  - holds an *address space*
  - holds a group of *threads* that execute within that address space
  - holds a collection of references to *open files* and other *"execution context"*
- *Address space:*
  - set of addresses that threads of the process can usefully reference
  - more precisely, it's the content of these addressable locations
    - text, data, bss, dynamic, stack segments/regions and what's in them
      - a *memory segment/region* contains usable *contiguous* memory addresses



Address Space Initialization

- Design issue:
  - how should the OS *initialize* these address space regions?
- Unix does it in two steps
  - make a copy of the address space using `fork()`
  - then copy contents from the file system to the process address space (as part of the *exec* operation)
  - quite wasteful (both in space and time) for the text region since it's read-only data
    - should *share* the text region
  - what about data regions? they can potentially be written into
    - can also *share* a portion of a data region if that portion is never modified
    - copy data structures are much faster than copy data



Remember This?

Virtual Memory

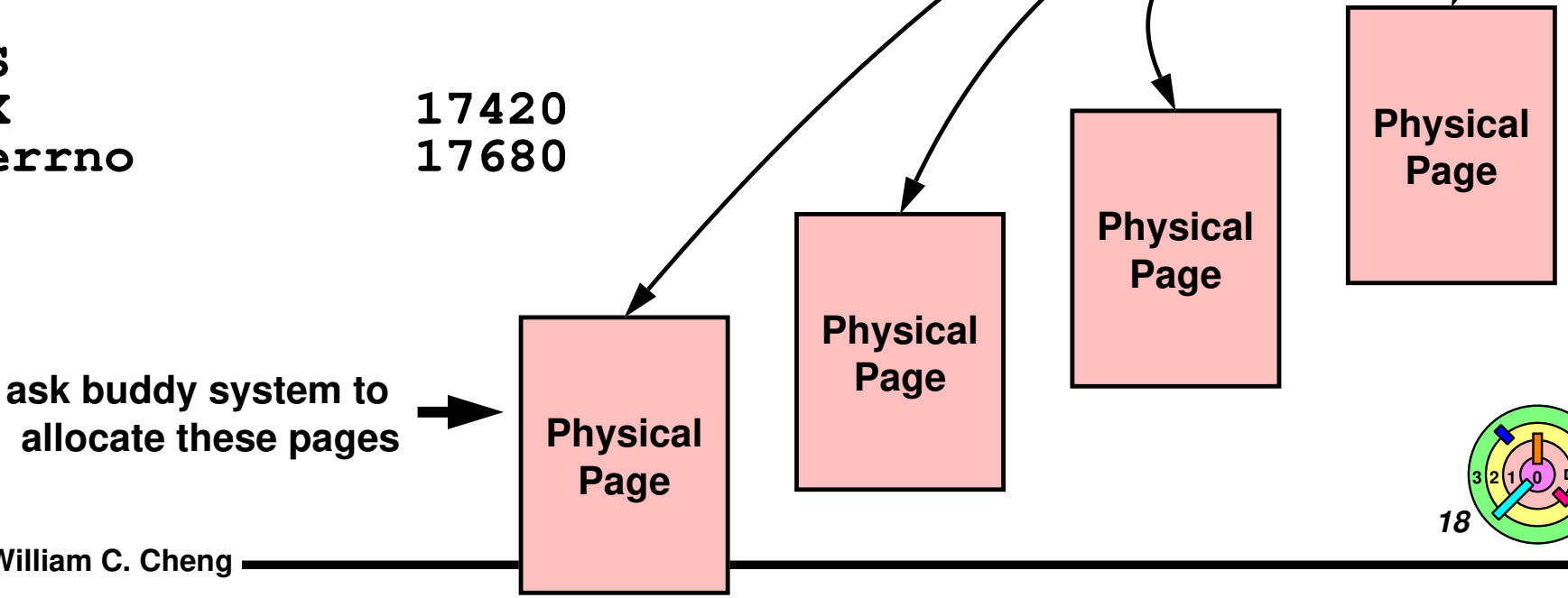
```
Text
   main              4096
   subr              4132
   printf            4156
   write            16156
   startup          16172

Data
   aX               16384
   printfargs       16388
   StandardFiles    16396

BSS
   X                17420
   errno            17680
```

Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	R	
2	8192	R	
3	12288	R	
4	16384	R/W	



Processes Can Share Memory Pages

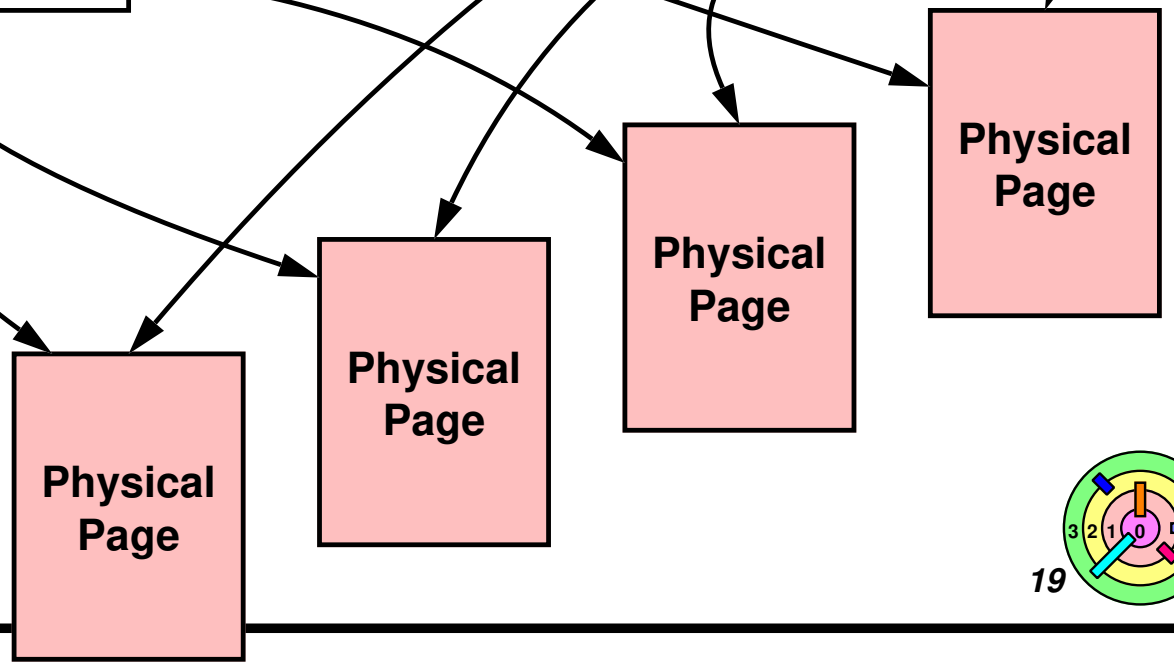
- Inside `fork()`, can simply copy parent's page table to child

Child Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	R	
2	8192	R	
3	12288	R	
4	16384	R/W	

Parent Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	R	
2	8192	R	
3	12288	R	
4	16384	R/W	



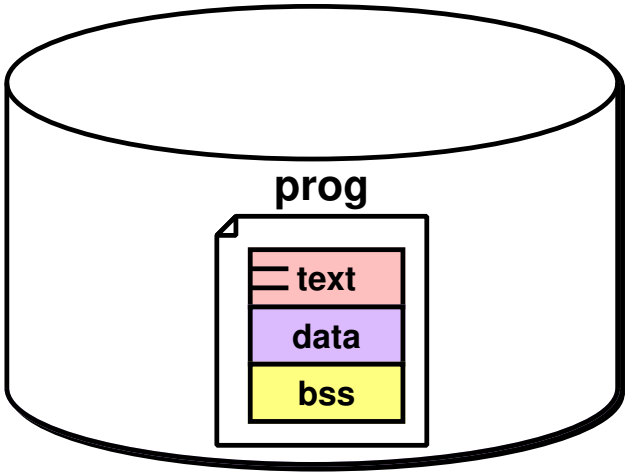
- power of *indirection*

exec()

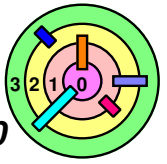
- Inside `exec()`, need to wipe out the address space (and page table) and create a new address space (and page table)

Child Page Table

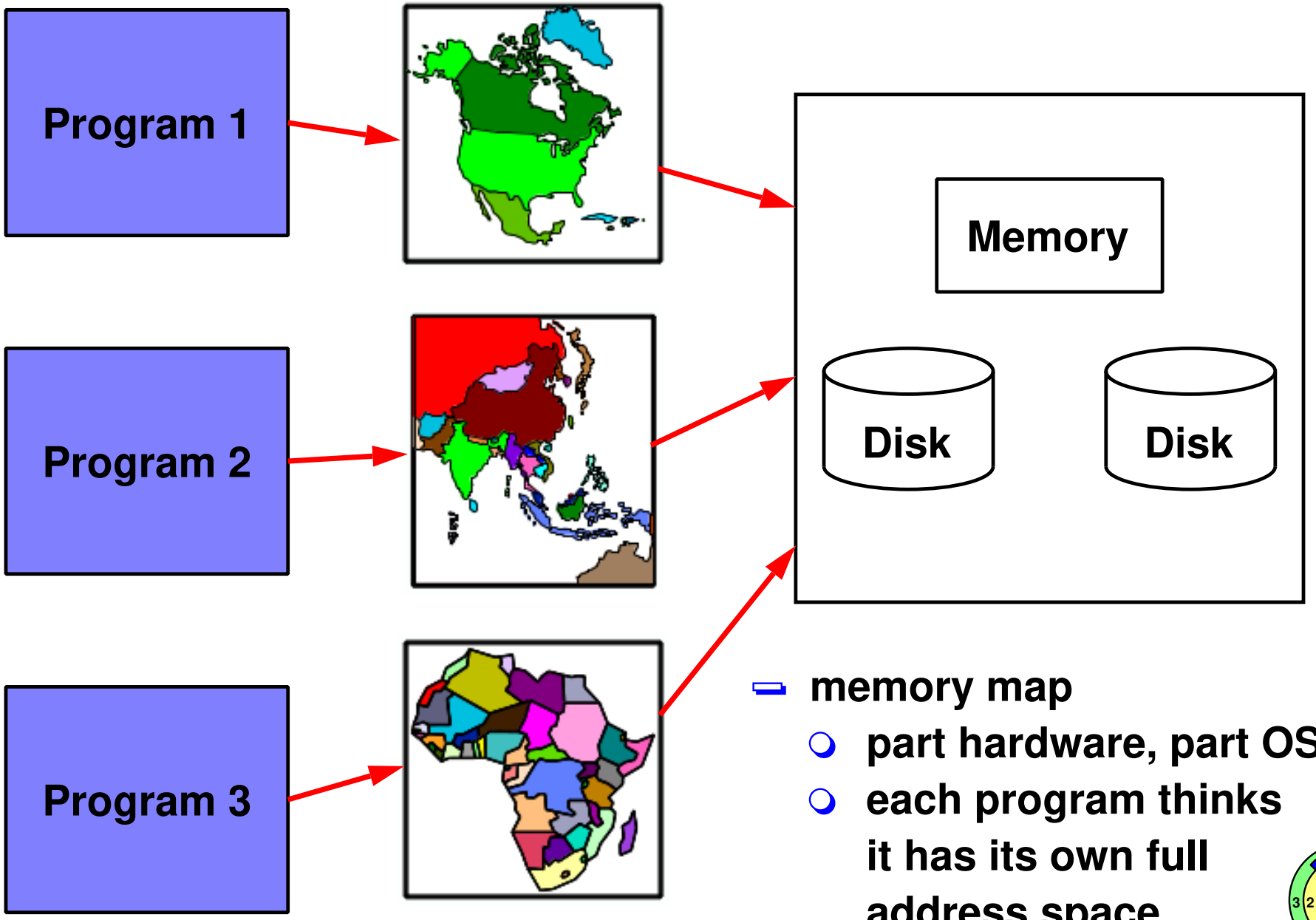
#	Start	Access	Physical Addr
0	0	-	-
1	4096	-	-
2	8192	-	-
3	12288	-	-
4	16384	-	-



- should you copytext and data segments of the new program from disk into memory now?
  - can be quite wasteful if you quit your new program quickly (and only use a small amount of the data you just copied form disk)



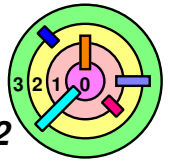
Memory Map



- memory map
  - part hardware, part OS
  - each program thinks it has its own full address space

Memory Map

- For the text region, why bother copying the executable file into the address space in the first place?
  - can just *map* the *file* into the *address space* (Ch 7)
    - *mapping* is an important concept in the OS
      - file mapping is *not* the same thing as address translation
      - some virtual memory pages map to files, and some map to physical memory
    - *mapping* let the OS *tie* the regions of the address space to the file system
    - address space and files are divided into pieces, called *pages*
    - if several processes are executing the same program, then at most one copy of that program's text page is in memory at once
  - *text regions* of all processes running this program are setup, using hardware address translation facilities, to share these pages
    - this type of mapping is known as *shared mapping*

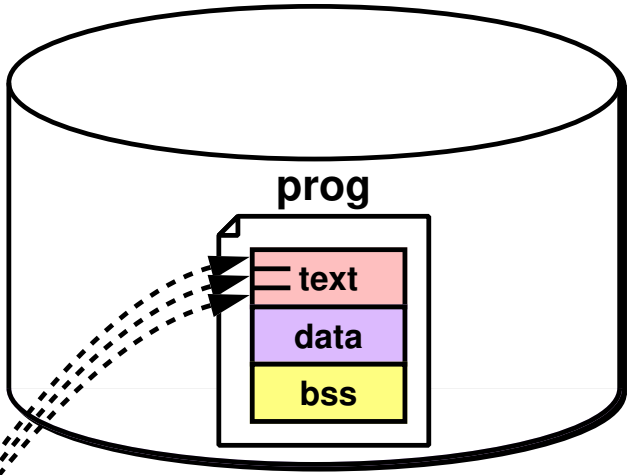


Memory Map

- The kernel uses a *memory map* to keep track of the mapping from *virtual pages* to *file pages*

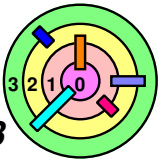
Child Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	-	-
2	8192	-	-
3	12288	-	-
4	16384	-	-

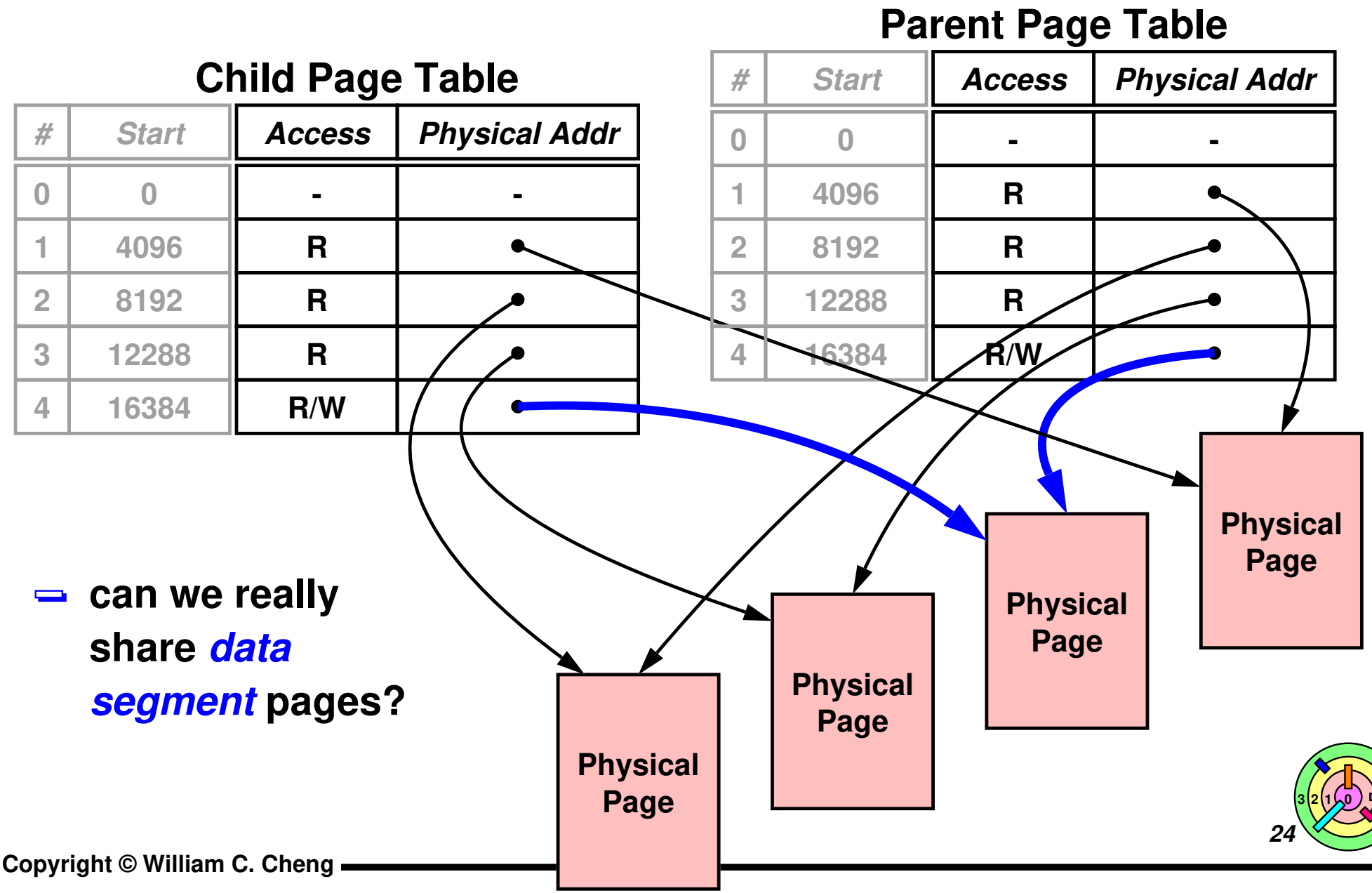


OS

- the kernel also uses *memory map* to keep track of the mapping from *virtual pages* to *physical pages*
  - also use it to maintain the page table data structure



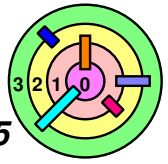
Processes Can Share Memory Pages



- can we really share *data segment* pages?

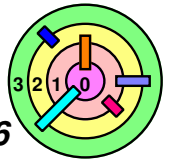
Address Space Initialization

- *Text regions* uses *shared mapping*
- *Data regions* of all processes running this program *initially* refer to pages of memory containing the *initial* data region
  - this type of mapping is known as *private mapping*
    - when does each process really need a private copy of such a page?
      - when data is *modified* by a process, it gets a *new* and *private* copy of the initial page



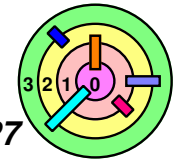
Copy-On-Write

- ***Copy-on-write (COW):***
  - a process gets a ***private*** copy of the page after a thread in the process performs a ***write*** to that page for the ***first time***
    - the basic idea is that only those pages of memory that are modified are copied
- Use private mapping and copy-on-write for data and bss regions
- The dynamic/heap and stack regions use a special form of private mapping
  - their pages are initialized, with zeros (in Linux); ***copy-on-write***
    - these are known as ***anonymous pages***
- If we can implement copy-on-write at the right time, then it's perfectly okay for processes to share address spaces
  - details in Ch 7



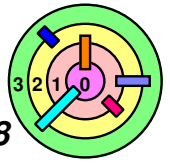
Shared Files

- If a bunch of processes *share* a *file*
  - we can also *map* the file into the *address space* of each process
  - in this case, the mapping is *shared*
  - when one process modifies a page, no private copy is made
    - instead, the original page itself is modified
    - everyone gets the changes
    - and changes are written back to the file
      - more on issues in Ch 6
- Can also share a file read-only
  - writing through such a map will cause segmentation fault



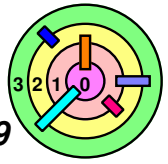
Memory Maps Summary

- File mapping
  - shared mapping
    - R/W: may change shared data *on disk*
    - R/O: read-only
  - private mapping
    - R/W: copy-on-write (will *not* change data on disk)
    - R/O: read-only
- Anonymous mapping
  - shared mapping (may be just shared with child processes)
    - R/W: may change shared data *in memory*
    - R/O: read-only
  - private mapping
    - R/W: copy-on-write
    - R/O: read-only
- Can also use all of the above in an application
  - `mmap()` system call



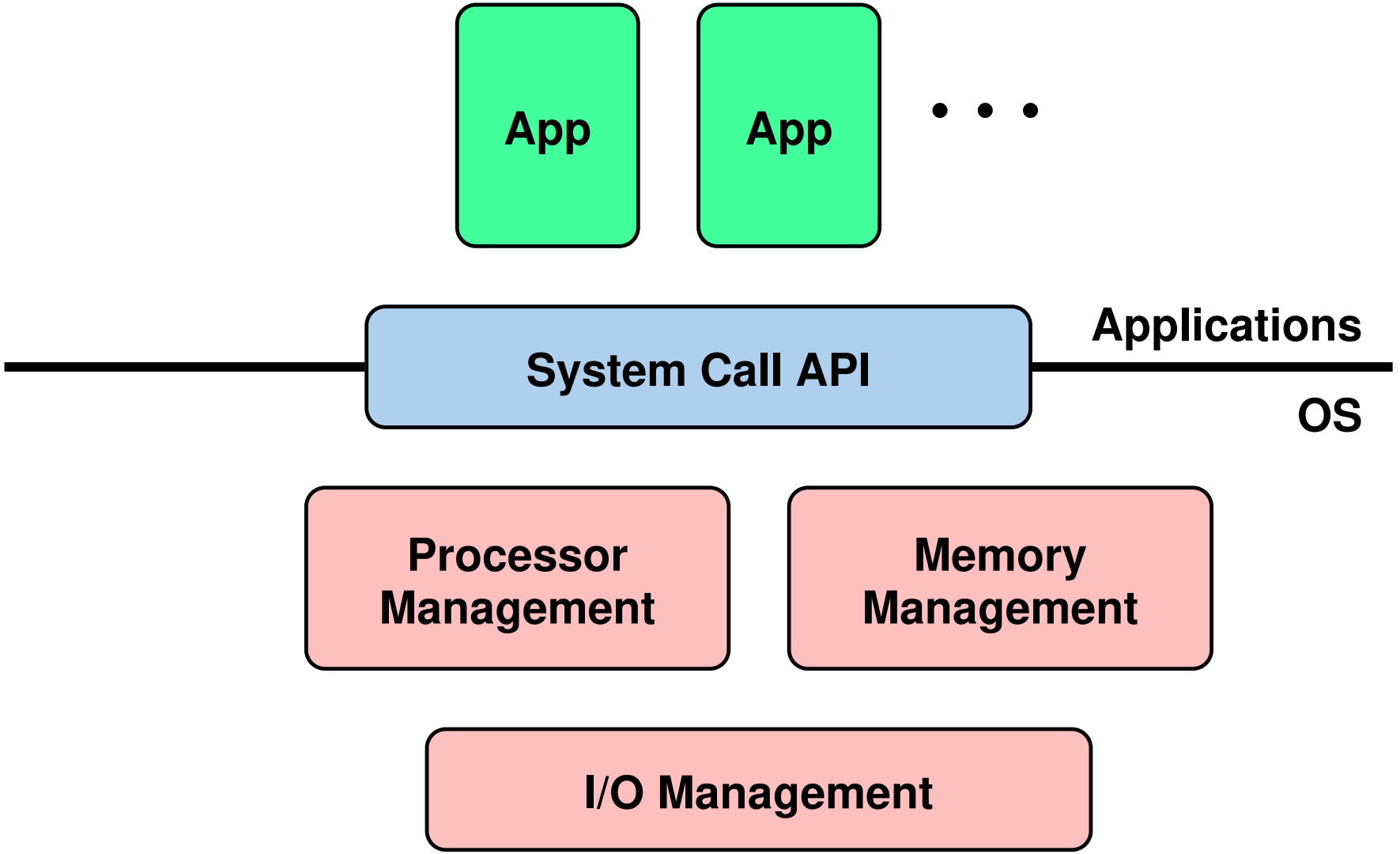
Block I/O vs. Sequential I/O

- Mapping files into address space is one way to perform I/O on files
  - block/page is the basic unit
  - some would refer to this as *block I/O*
- Some devices cannot be mapped into the address space
  - e.g., receiving characters typed into the keyboard, sending a message via a network connection
  - need a more traditional approach using explicit system calls such as `read()` and `write()`
  - this is referred to as *sequential I/O*
- It also makes sense to be able to read a file like reading from the keyboard
  - similarly, a program that produces lines of text as output should be able to use the same code to write output to a file or write it out to a network connection
  - makes life easier! (and make code more robust)



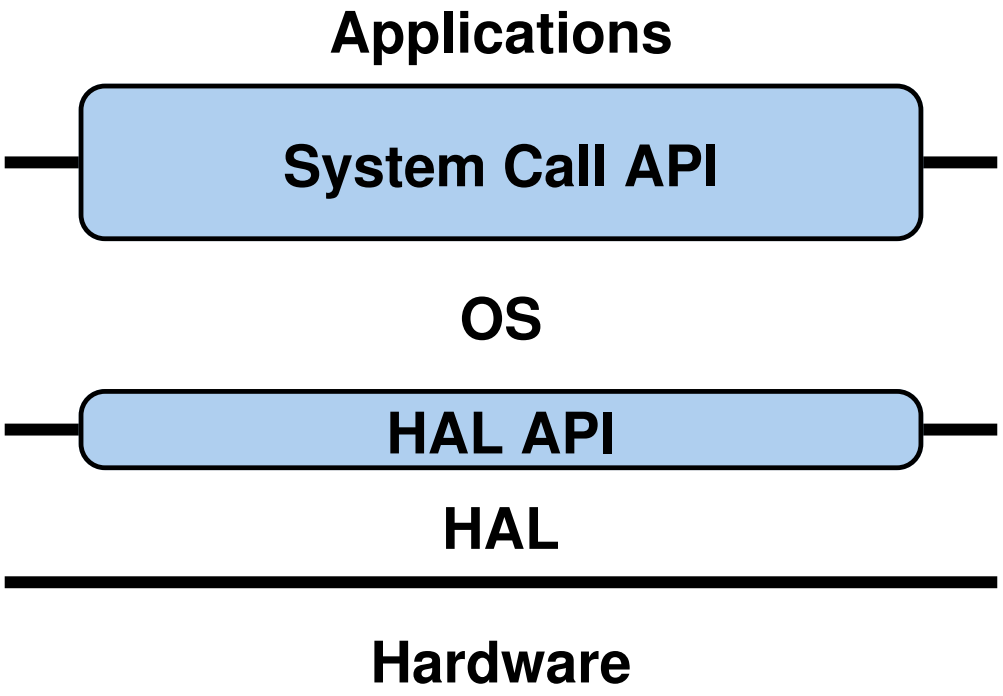
System Call API

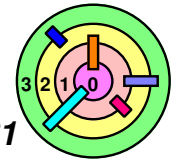
- Backwards compatibility is an important issue
  - try not to change it much (to make developers happy)



Portability

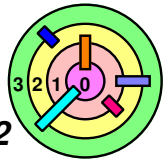
- It is desirable to have a portable operating system
  - portable across various hardware platforms
- For a monolithic OS, it is achieved through the use of a *Hardware Abstraction Layer (HAL)*
  - a *portable interface* to *machine configuration* and *processor-specific operations* within the kernel





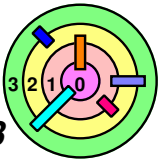
Hardware Abstraction Layer (HAL)

- Portability across machine configuration
  - e.g., different manufacturers for x86 machines will require different code to configure interrupts, hardware timers, etc.
- Portability across processor families
  - e.g., may need additional code for context switching, system calls, interrupting handler, virtual memmory management, etc.
- With a well-defined Hardware Abstraction Layer, most of the OS is *machine* and *processor independent*
  - porting an OS to a new computer is done by
    - writing new HAL routines
    - relink with the kernel



4.1 A Simple System (Monolithic Kernel)

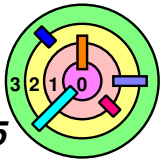
- *A Framework for Devices*
- Low-level Kernel (will come back to talk about this after Ch 7)
- Processes & Threads
- Storage Management (will come back to talk about this after Ch 5)



Computer Terminal

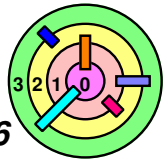
VT100

A "tty"



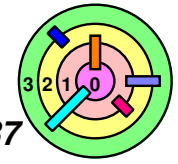
Devices

- Challenges in supporting devices
  - device independence
  - device discovery
- Device naming
  - two choices
    - independent name space (i.e., named independently from other things in the system)
    - devices are named as files

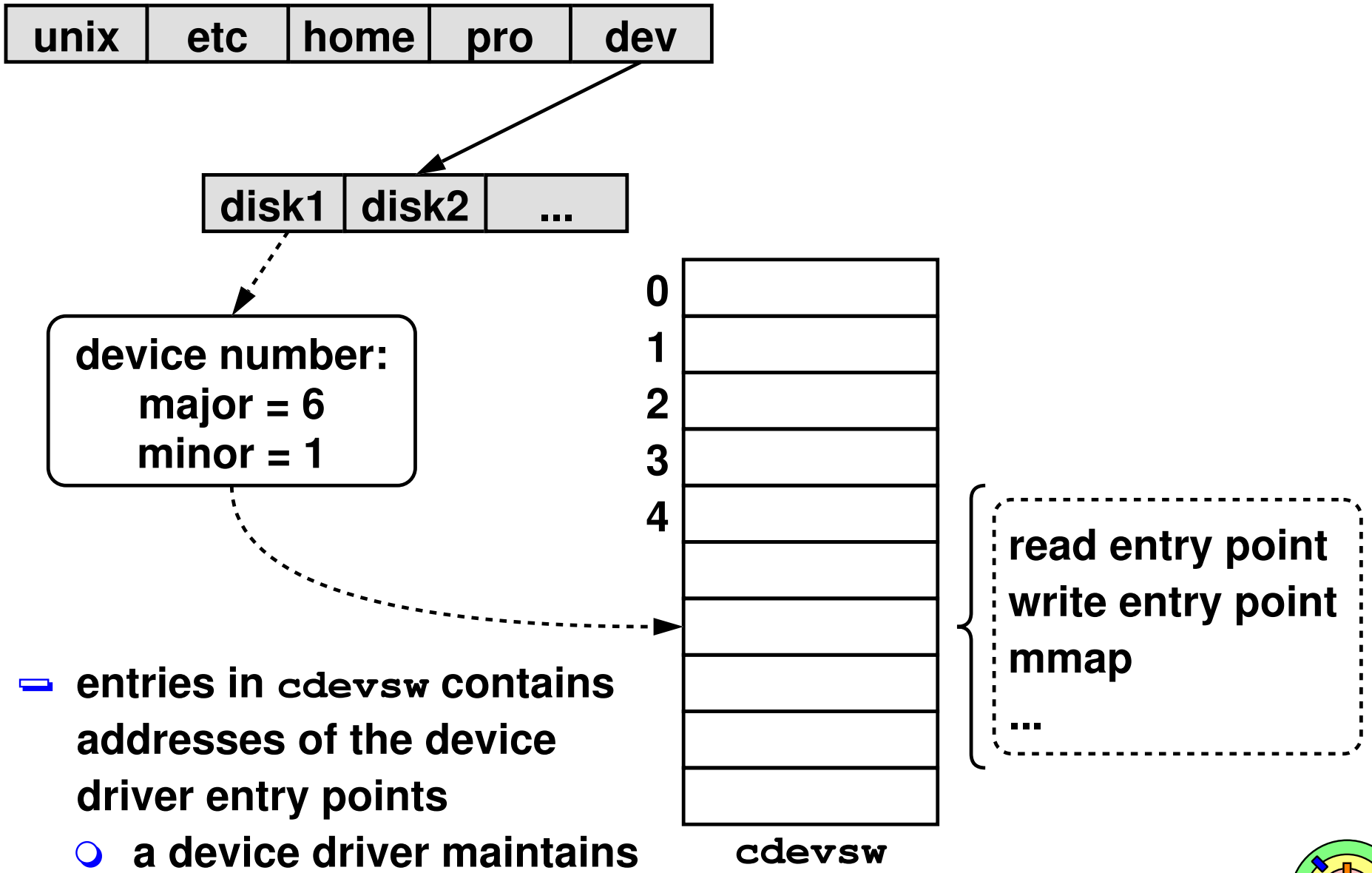


A Framework for Devices

- Device driver:
  - every device is identified by a device "number", which is actually a pair of numbers
    - a *major device number* - identifies the device driver
    - a *minor device number* - device index for all devices managed by the same device driver
- Special entries were created in the file system to refer to devices
  - usually in the `/dev` directory
    - e.g., `/dev/disk1`, `/dev/disk2` each marked as a *special file*
      - a *special file* does not contain data
      - it refers to devices by their major and minor device numbers
      - if you do "`ls -l`", you can see the device numbers
- Data structure in the early Unix systems
  - statically allocated array in the kernel called `cdevsw` (character device switch)



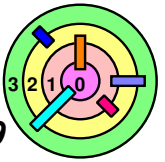
Finding Devices



- entries in `cdevsw` contains addresses of the device driver entry points
  - a device driver maintains its own data structure

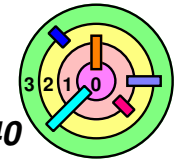
Device Drivers in Early Unix Systems

- The kernel was statically configured to contain device-specific information such as:
  - interrupt-vector locations
  - locations of device-control registeres on whatever bus the device was attached to
- Static approach was simple, but cannot be easily extended
  - a kernel must be custom configured for each installation



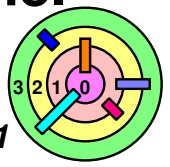
Device Probing

- First step to improve the old way
  - allow the devices to to be found and automatically configured when the system booted
  - (still require that a kernel contain all necessary device drivers)
- Each device driver includes a *probe routine*
  - invoked at boot time
  - probe the relevant buses for devices and configure them
    - including identifying and recording interrupt-vector and device-control-register locations
- This allowed one kernel image to be built that could be useful for a number of similar but not identical installations
  - boot time is kind of long
  - impractical as the number of supported devices gets big



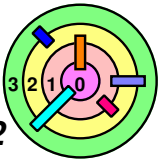
Device Probing

- What's the right thing to do?
  - Step 1: discover the device without the benefit of having the relevant device driver in the kernel
  - Step 2: find the needed device drivers and dynamically link them into the kernel
  - but how do you achieve this?
- Solution: use meta-drivers
  - a meta-drive handles a particular kind of bus
  - e.g., USB (Universal Serial Bus)
    - a USB meta-driver is installed into the kernel
    - any device that goes onto a USB (Universal Serial Bus) must know how to interact with the USB meta-driver via the *USB protocol*
    - once a connected device is identified, system software would select the appropriate device driver and load into the kernel
    - what about applications? how can they reference dynamically discovered devices?



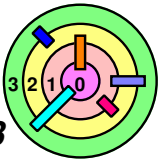
Discovering Devices

- So, you plug in a new device to your computer on a particular bus
  - OS would notice
  - find a device driver
    - what kind of device is it?
    - where is the driver?
  - assign a name, but how is it chosen?
  - multiple similar devices, but how does application choose?
- In some Linux systems, entries are added into `/dev` as the kernel discovers them
  - lookup the names from a database of names known as `devfs`
    - downside of this approach is that device naming conventions not universally accepted
    - what's an application to do?
  - some current Linux systems use `udev`
    - user-level application assigns names based on rules provided by an administrator



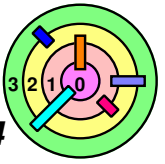
Discovering Devices

- What about the case where different devices acted similarly?
  - e.g., touchpad on a laptop and USB mouse
  - how should the choice be presented to applications?
- Windows has the notion of *interface classes*
  - a device can register itself as members of one or more such classes
  - an application can *enumerate* all currently connected members of such a class and choose among them (or use them all)



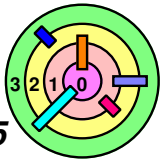
4.1 A Simple System (Monolithic Kernel)

- A Framework for Devices
- Low-level Kernel (will come back to talk about this after Ch 7)
- *Processes & Threads*
- Storage Management (will come back to talk about this after Ch 5)

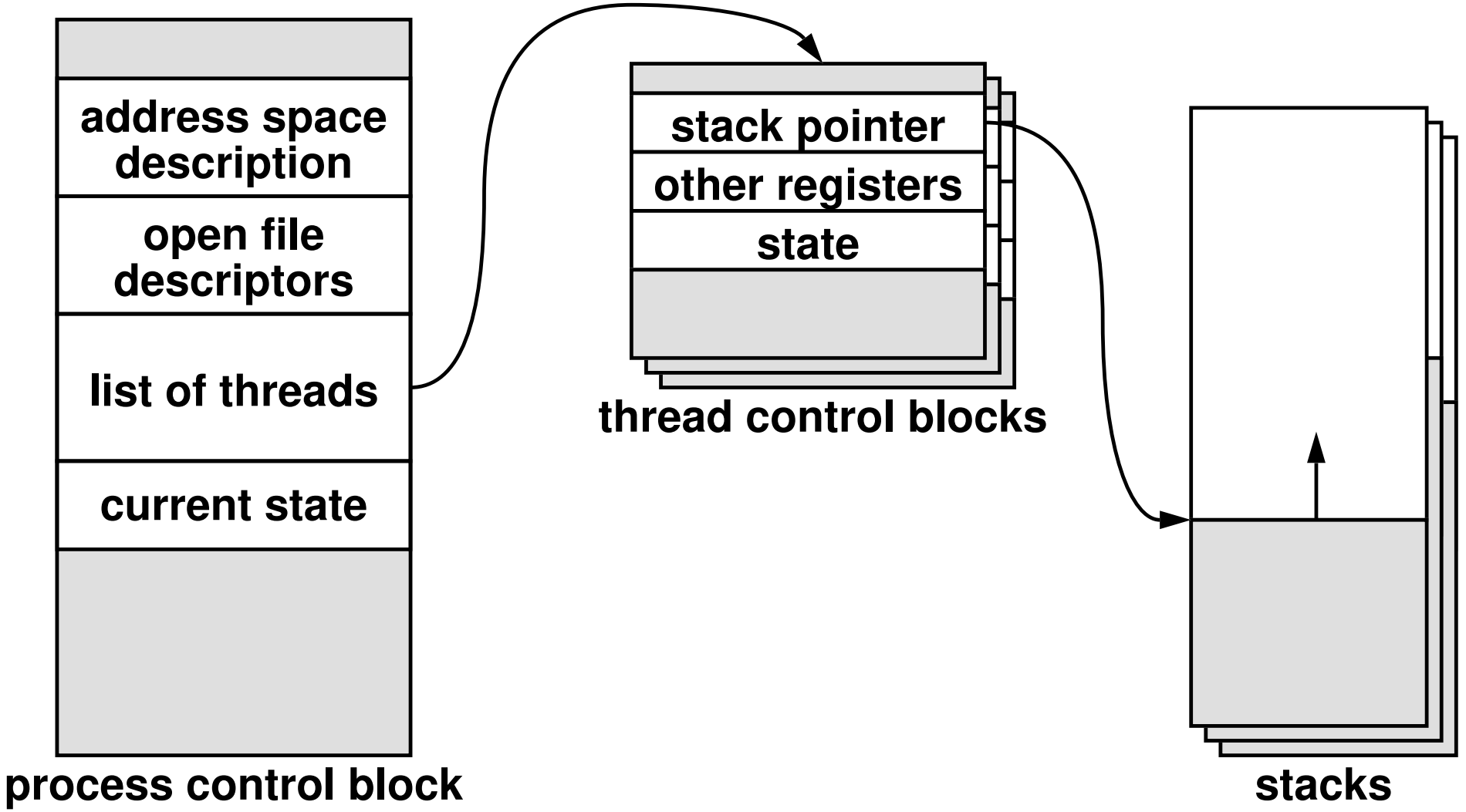


Processes and Threads

- A process is:
  - a holder for an *address space*
  - a collection of other information shared by a set of *threads*
  - a collection of references to *open files* and other "execution context"
- As discussed in Ch 1, processes related APIs include
  - `fork()`, `exec()`, `wait()`, `exit()`



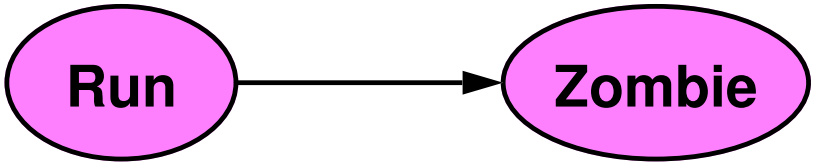
Processes and Threads

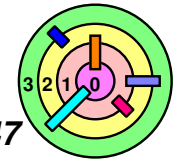


- Note: all these are relevant to your Kernel Assignment 1
  - although we are only doing *one thread per process*

Process Life Cycle

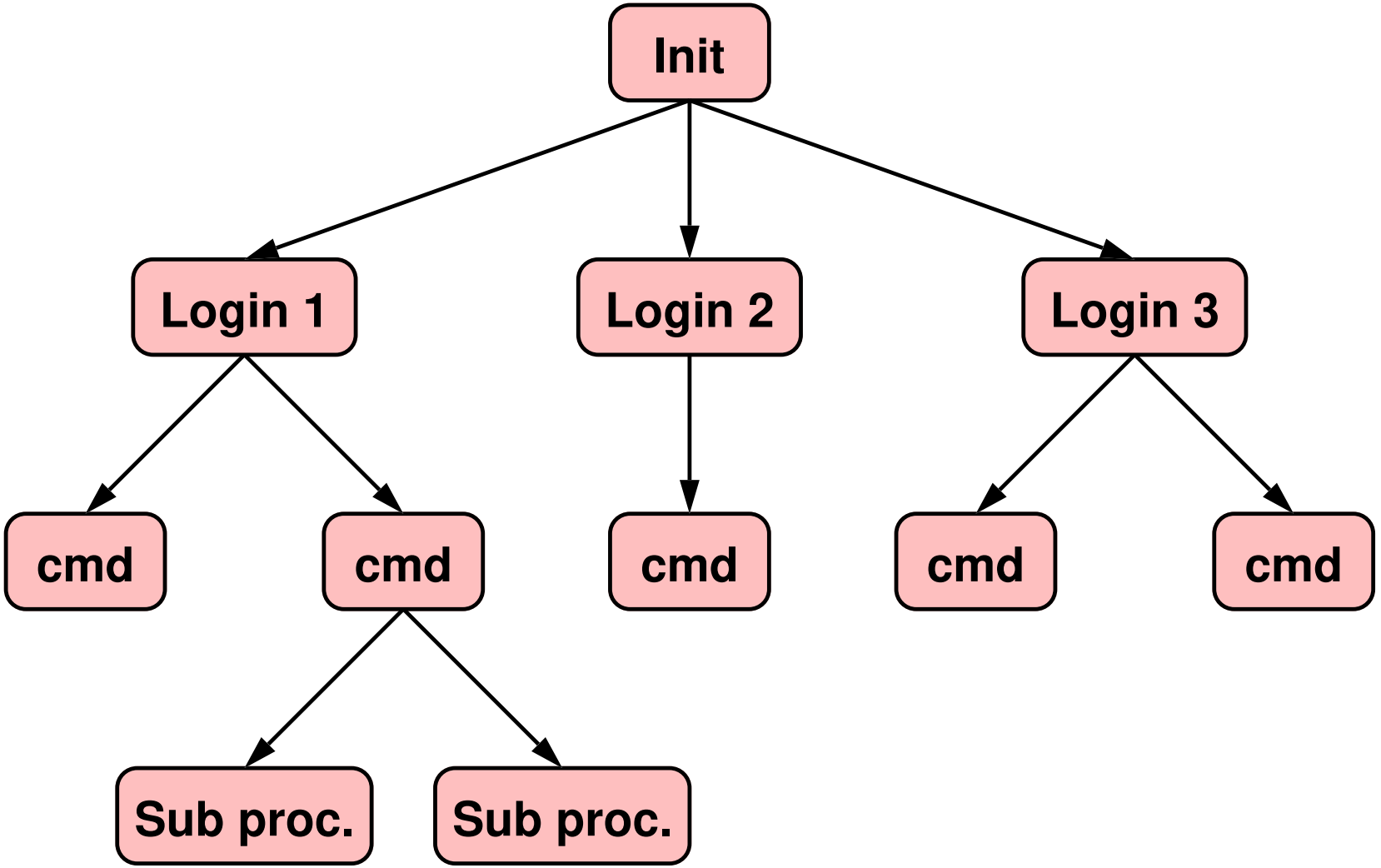
- Pretty simple
  - a process starts in the *run* state

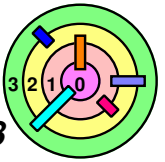




Process Relationships (1)

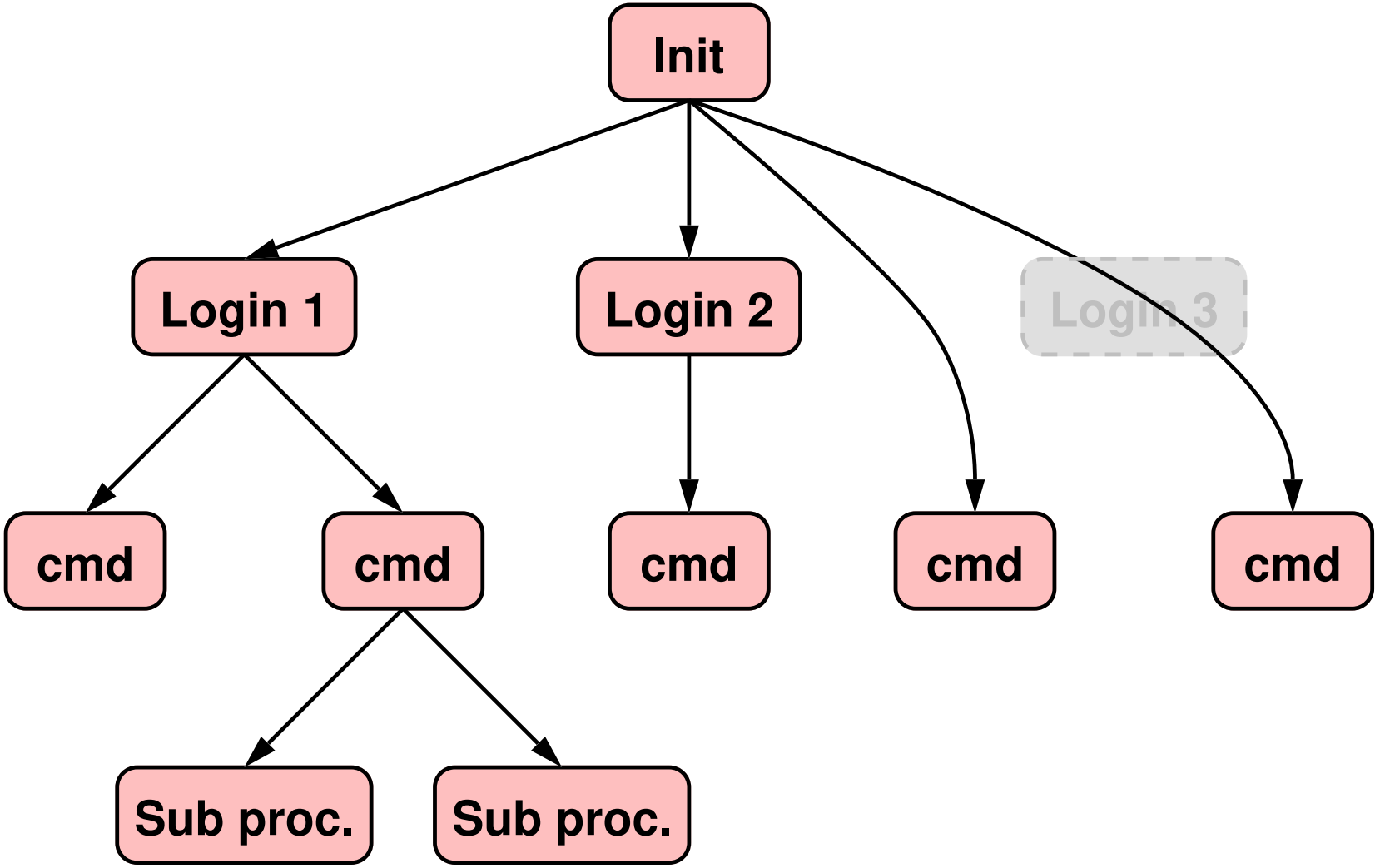
- Process hierarchy
  - run "`pstree`" on Linux

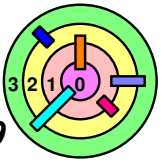




Process Relationships (2)

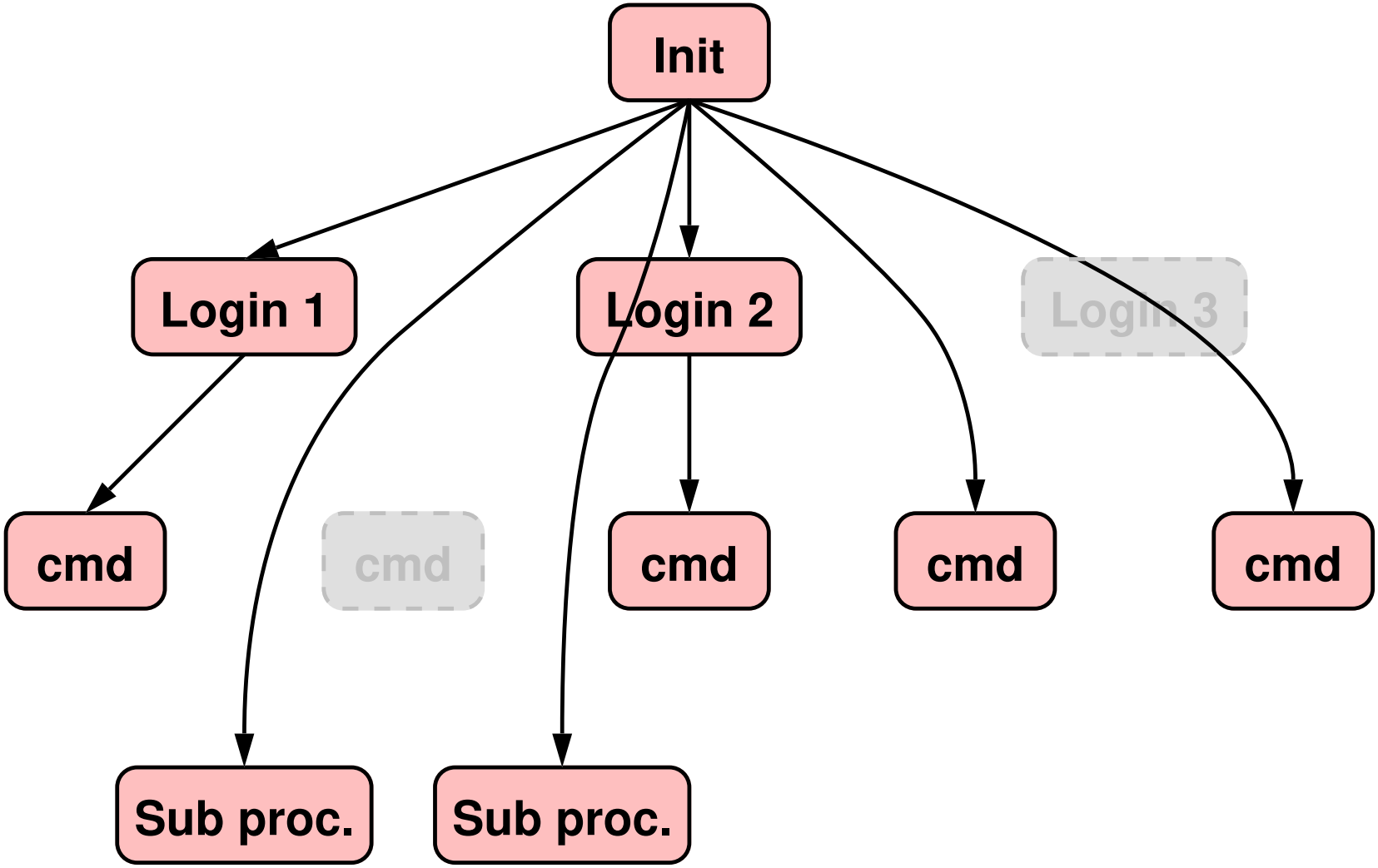
- If a process dies, you must *reparent* all its child processes

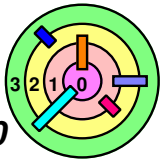




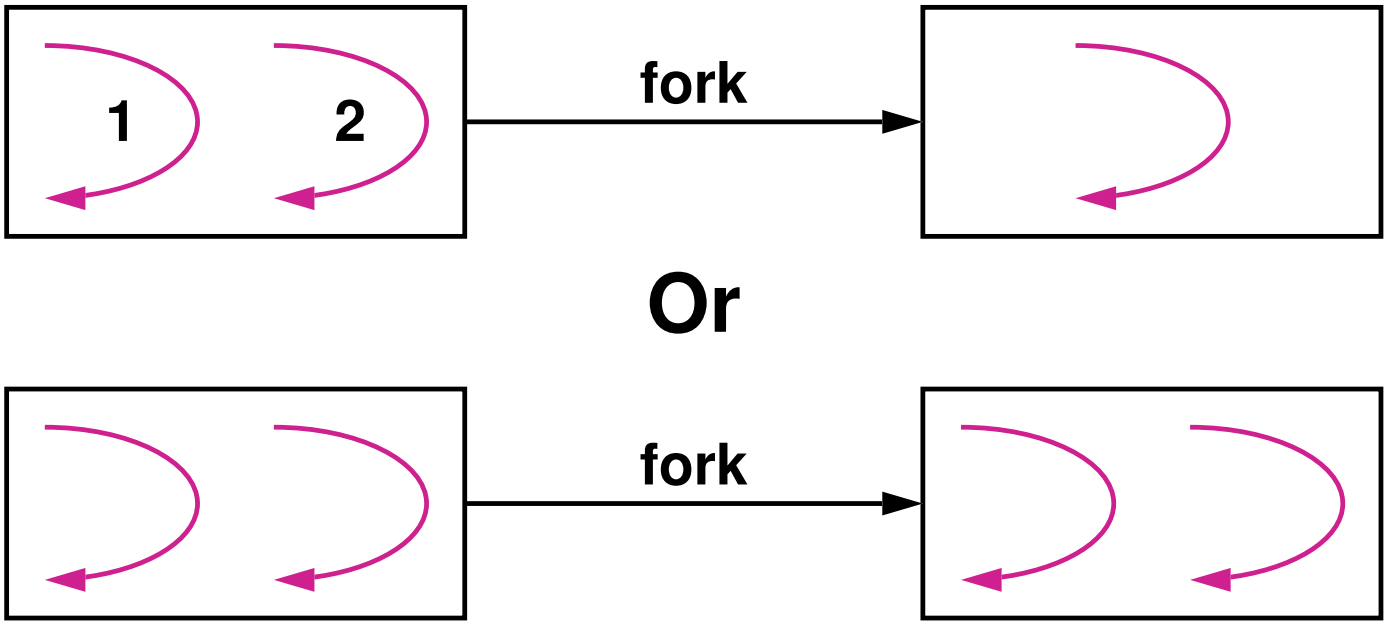
Process Relationships (3)

- If a process dies, you must *reparent* all its child processes
  - new parent is the INIT process

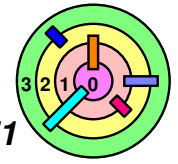




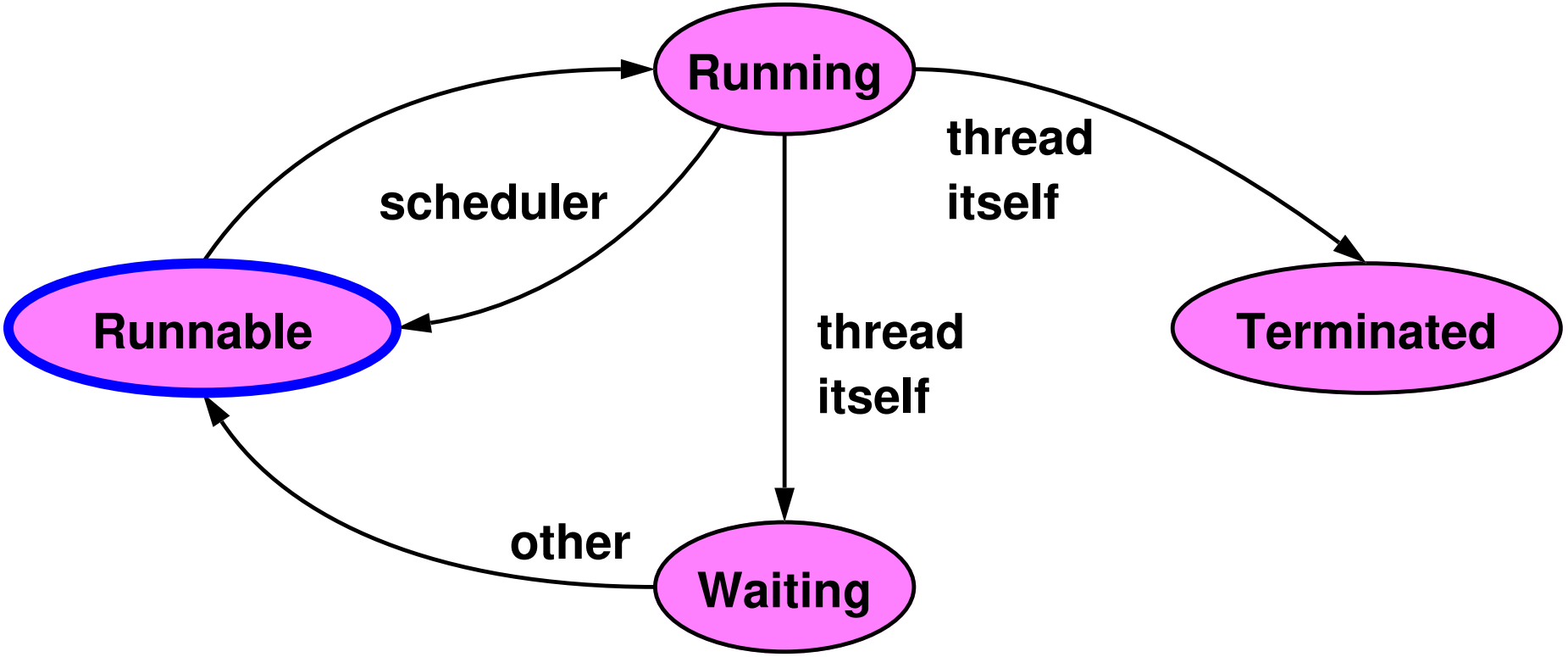
Fork and Threads



- Solaris uses the 2nd approach
  - expensive to fork a process
- Problem with 1st approach
  - thread 1 called `fork()` and thread 2 has a mutex locked
    - who will unlock the mutex?
  - POSIX solution is to provide a way to unlock all mutex before `fork()`

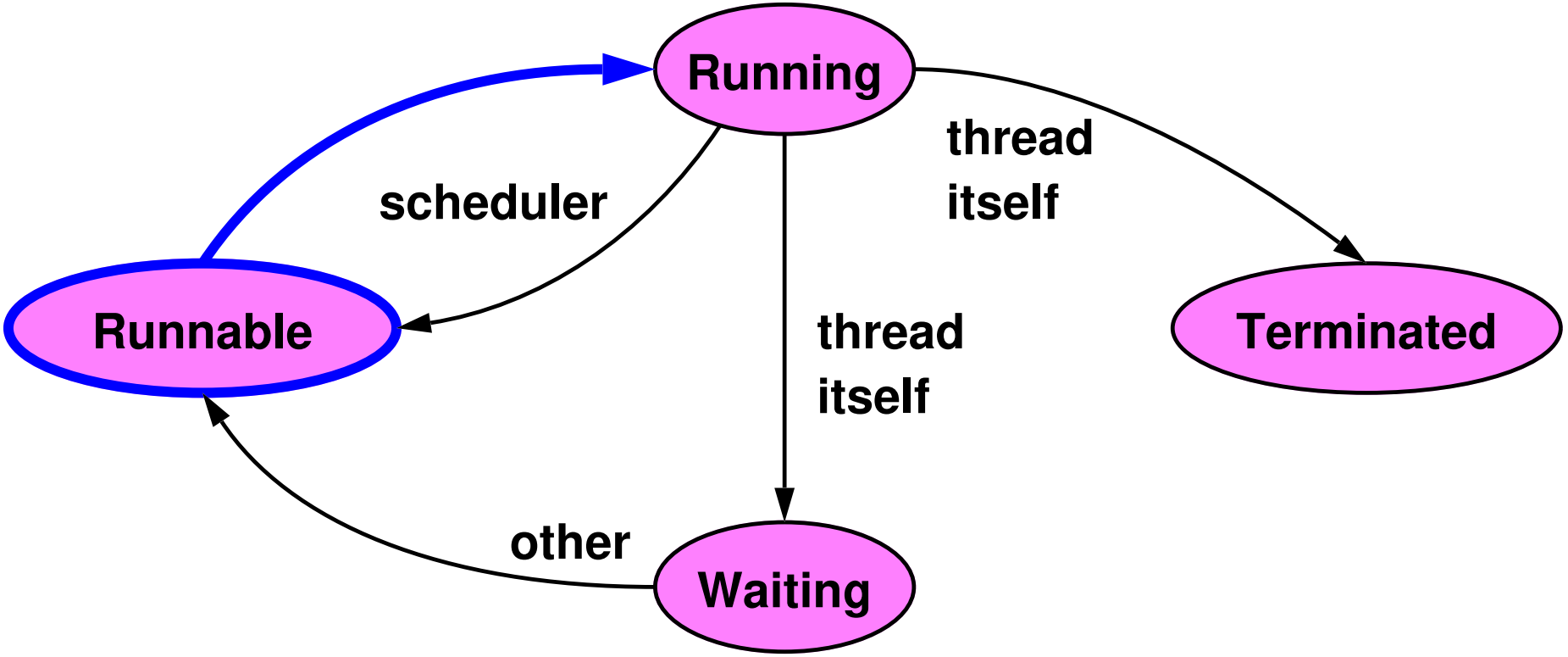


Thread Life Cycle

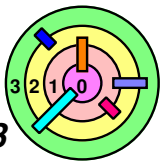


- a thread starts in the *runnable* state
  - *sleeps* in the *run queue* (or "ready queue")
    - threads sleep in the run queue to wait to use the CPU

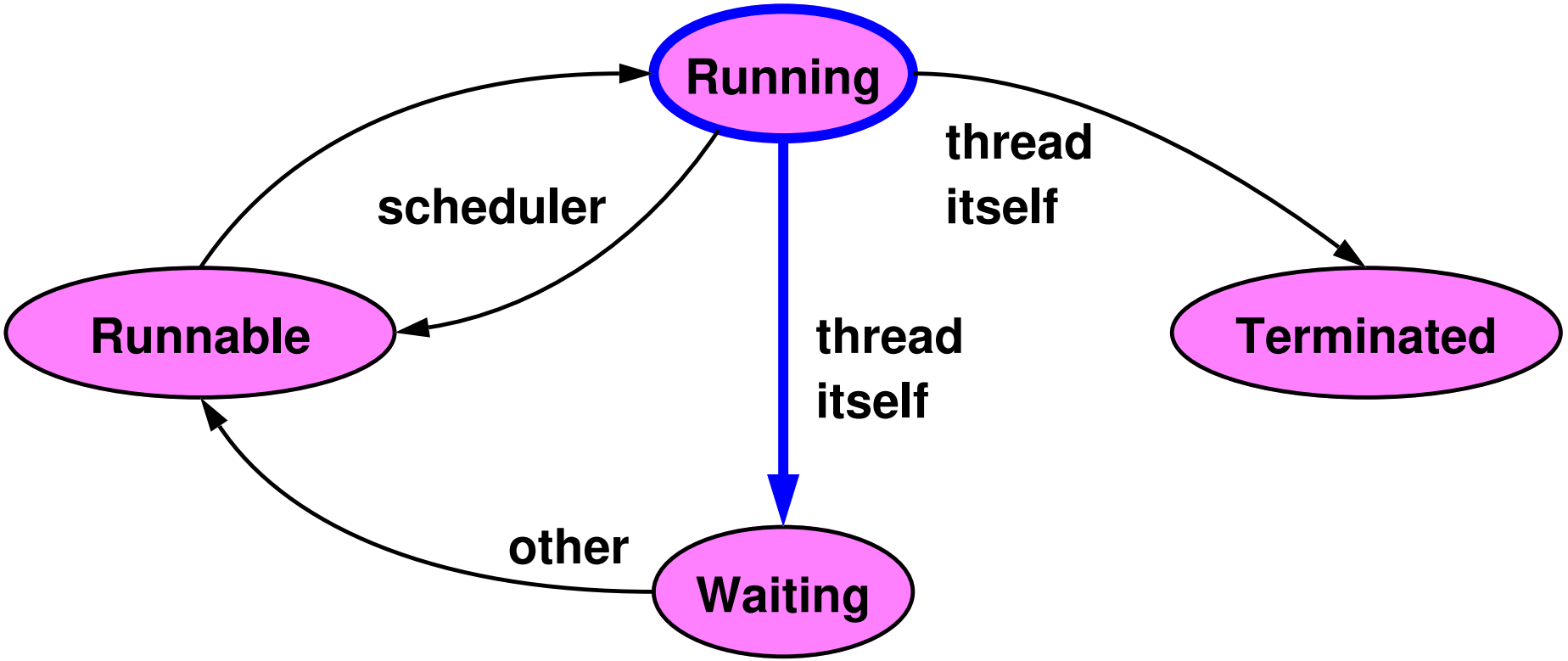
Thread Life Cycle



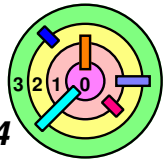
- the *scheduler* switches a thread's state from runnable to running
  - the scheduler decides who to run next inside the CPU



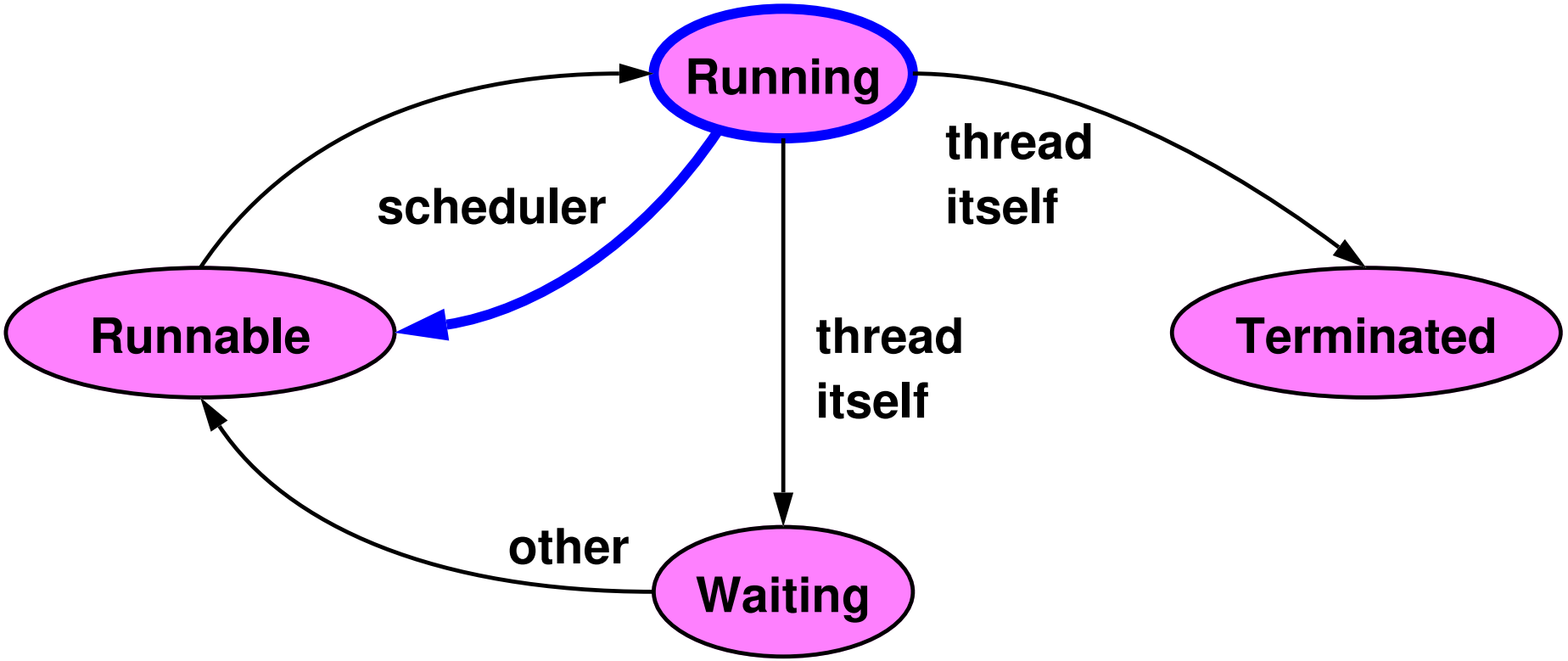
Thread Life Cycle



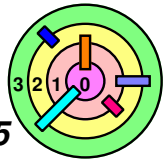
- a thread goes from running to waiting when a *blocking call* is made by the thread itself
  - the scheduler is *not involved* here



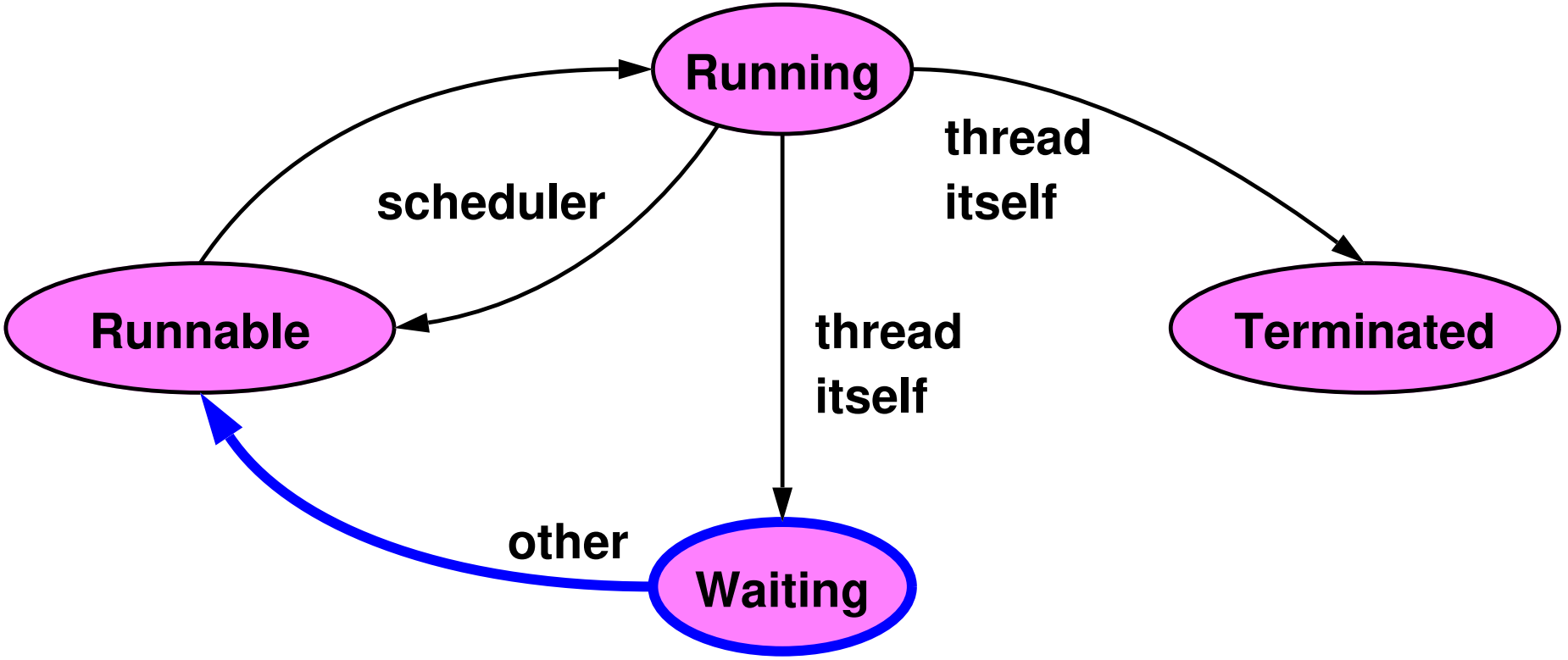
Thread Life Cycle



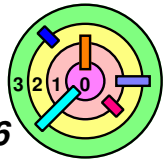
- the *scheduler* switches a thread's state from running to runnable when the thread used up its execution quantum
  - a thread can also *"yield"* the CPU (see examples in `faber_thread_test()` in kernel 1)



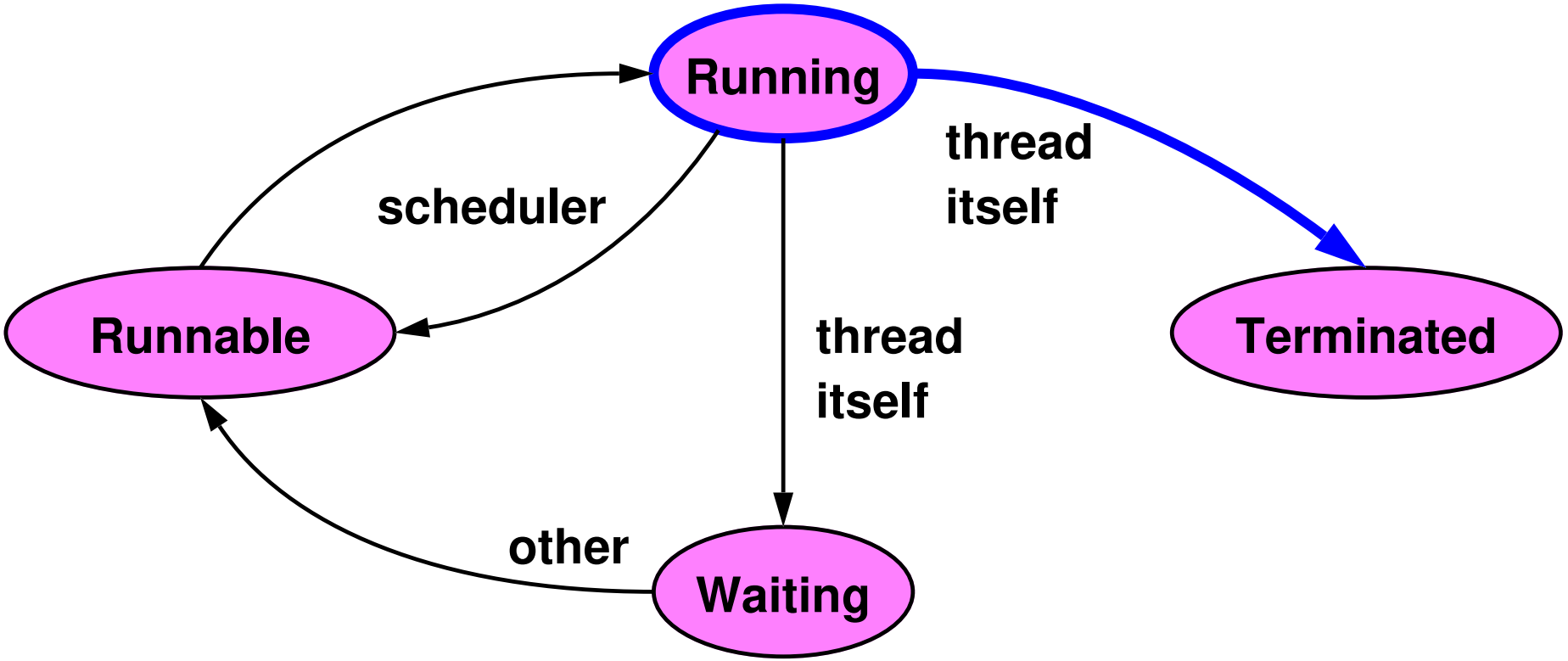
Thread Life Cycle



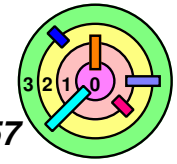
- a thread gets *unblocked* by the action of another thread or by an interrupt handler
  - the scheduler is *not involved* here



Thread Life Cycle

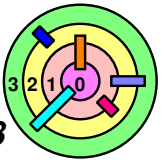


- in order for a thread to enter the terminated state, it has to be in the running state just before that
  - what if something like `pthread_cancel()` is invoked when the thread is not in the running state?



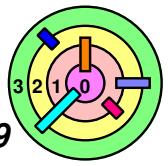
Thread Life Cycle

- Does `pthread_exit()` delete the thread (completely) that calls it?
  - no, the thread goes into a zombie state (i.e., "terminated")
- What's left in the thread after it calls `pthread_exit()`?
  - its thread control block
    - needs to keep thread ID and return code around
  - its stack
    - how can a thread *delete its own stack*? no way!
      - which stack are we talking about anyway?



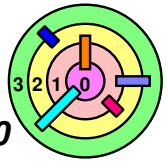
Thread Life Cycle

- Who is deleting the *thread control block* and freeing up the thread's *stack* space?
- If a thread is not detached
  - it can be taken care of in the `pthread_join()` code
    - the thread that calls `pthread_join()` does the clean up
- If a thread is detached (our simple OS does not support this)
  - can do this is one of two ways
    1) use a special *reaper thread*
       - basically doing `pthread_join()`
    2) queue these threads on a list and have other threads free them when it's convenient (e.g., when the scheduler schedule a thread to run)



Kernel 1 Process & Thread Life Cycles

- Part of the kernel 1 assignment is to implement the *life cycles* of processes and threads
  - process/thread creation/termination
    - since we are only doing one thread per process (`MTP=0`), when a thread dies, the process must die as well
  - process/thread cancellation
  - process waiting (and no thread joining since `MTP=0`)
  - etc.
- Unlike warmup2, in kernel assignments, first procedures of almost all kernel threads have been written for you already!
  - the thread code there make function calls and some of these functions are *not-yet-implemented*
    - your job is to implement those functions so that these kernel threads can run *perfectly*



Kernel 1 Process & Thread Life Cycles

- Hint on how to do this is by reading kernel code
  - *read the code* in "`kernel/proc/faber_test.c`"
    - if it calls a function that you are suppose to implement, it's telling you what it's expecting from that function!
      - feel free to discuss things like that in the class Google Group
    - you need to understand what every line of code is doing there
    - you need to pass every test there (see grading guidelines)
      - you must not change anything there
      - make sure the printout is correct (you may want to discuss it in the class Google Group)
    - if you need to do something similar in another module, just *copy* the code from it
      - you can copy code that's given to you as course material and you don't have to cite your source