Ch 4: Operating-System Design

Bill Cheng

http://merlot.usc.edu/cs402-s16

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!

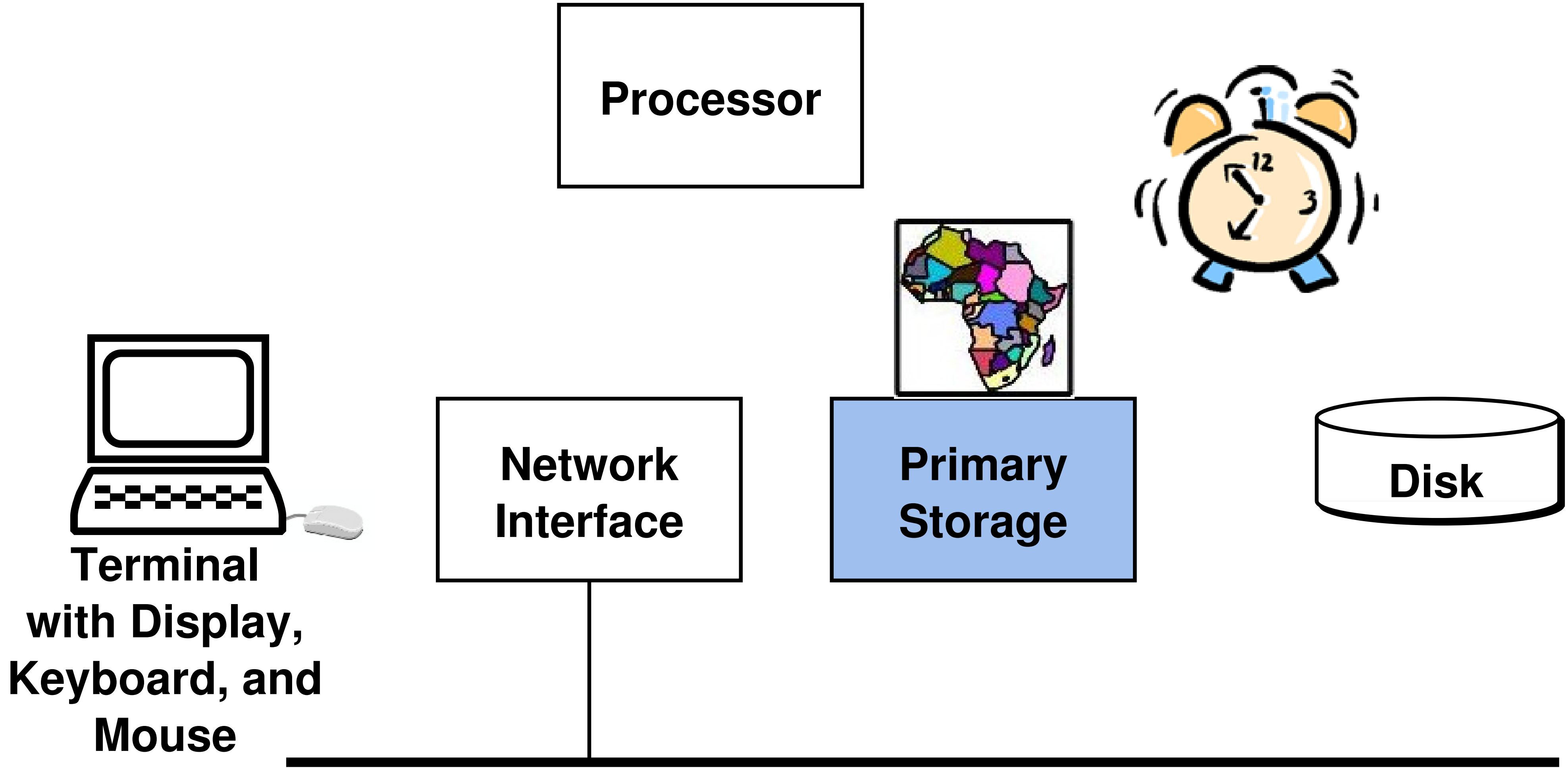
OS Design

- 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

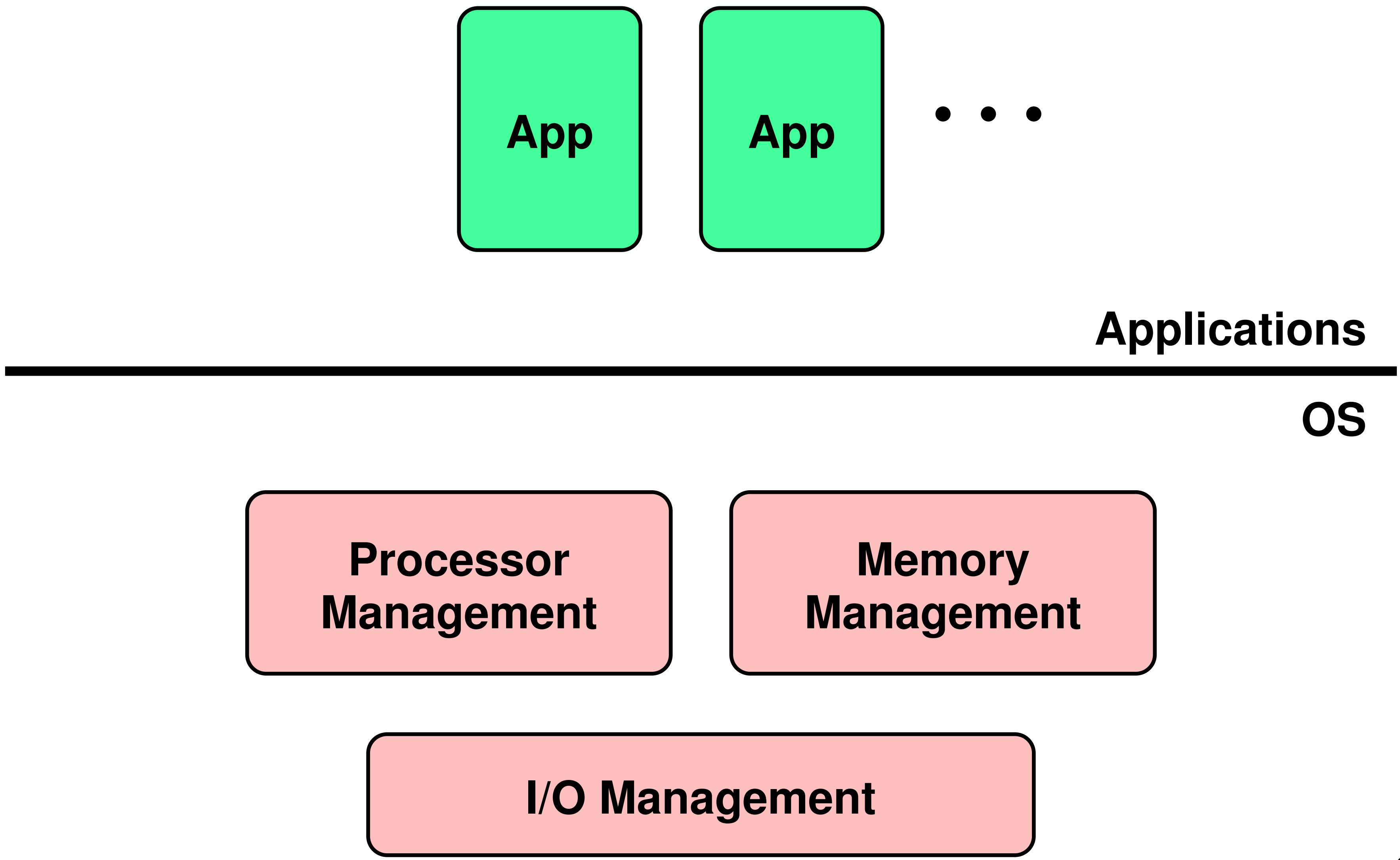
- **Performance**
  - efficiency of application
- **Modularity**
  - tradeoffs between modularity and performance
- **Device independence**
  - for new devices, don’t need to write a new OS
- **Security/Isolation**
  - isolate OS from application

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

Processor Management

- Scheduling
- Interrupt management
- Processes and threads

Memory Management

- Virtual memory
- Real memory

I/O Management

- Human interface device
- Network protocols
- File system
- Logical I/O management
- Physical device drivers

OS Components

Scheduling

Interrupt management

Processes and threads

Virtual memory

Real memory

Processor Management

Memory Management

Human interface device

Network protocols

Logical I/O management

Physical device drivers

I/O Management

supports multithreaded processes

- each process has its own address space

OS Components

Scheduling

Interrupt management

Processes and threads

Virtual memory

Real memory

Processor Management

Memory Management

supports virtual memory

Human interface device

Network protocols

Logical I/O management

Physical device drivers

I/O Management

OS Components

Processor Management
- Scheduling
- Interrupt management
- Processes and threads

Memory Management
- Virtual memory
- Real memory

I/O Management
- Human interface device
- Network protocols
- Logical I/O management
- Physical device drivers

theads executing is multiplexed on a single processor
- by a simple time-sliced scheduler

OS Components

has a file system
- layered on disks

Processes and threads

nent

Virtual memory

Real memory

Memory Management

Human interface device

Network protocols

File system

Logical I/O management

Physical device drivers

I/O Management

OS Components

Scheduling

Interrupt management

Processes and threads

Processor Management

Me

user interacts over a terminal
- text interface (typically 24 80-character rows)
- every character typed on the keyboard is sent to the processor

Human interface device

Network protocols

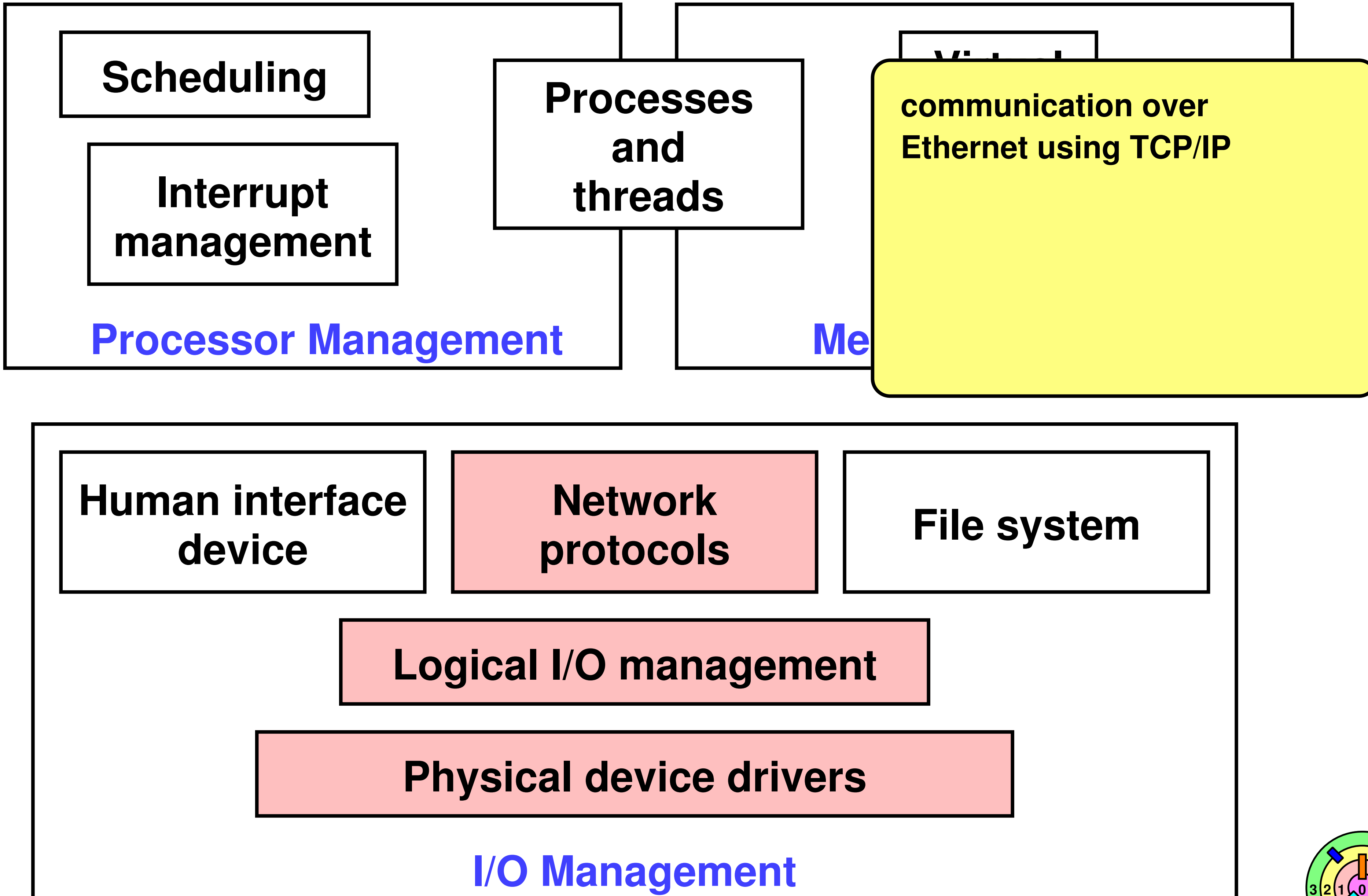
File system

Logical I/O management

Physical device drivers

I/O Management

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 regions and what's in them

Processes And File Systems

- Design issue:
  - how should the OS *initialize* these address space regions?
- Simple approach: copy their 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

Remember This?

Virtual Memory

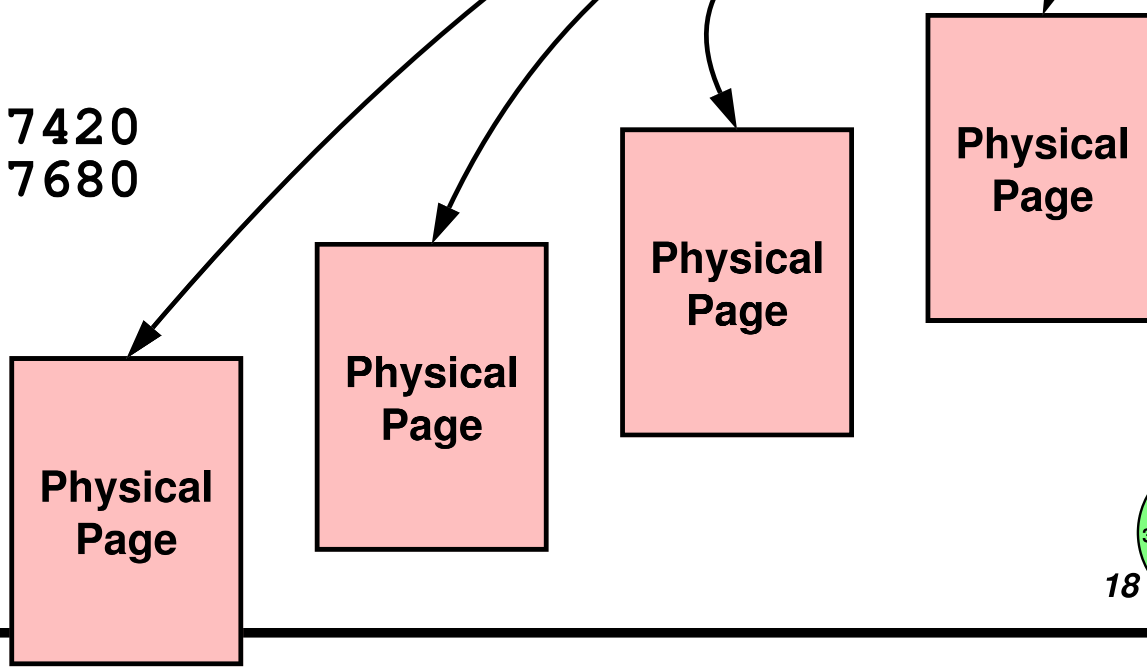
```
Text
   main                4096
   subr                4132
   printf              4156
   write              16156
   startup            16172

Data
   aX                 16384
   printfargs         16388
   StandardFiles      16396

BSS
   X                  17420
   errno              17680
```

Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	•
8192	R	•
12288	R	•
16384	R/W	•



ask buddy system to allocate these pages

Processes Can Share Memory Pages

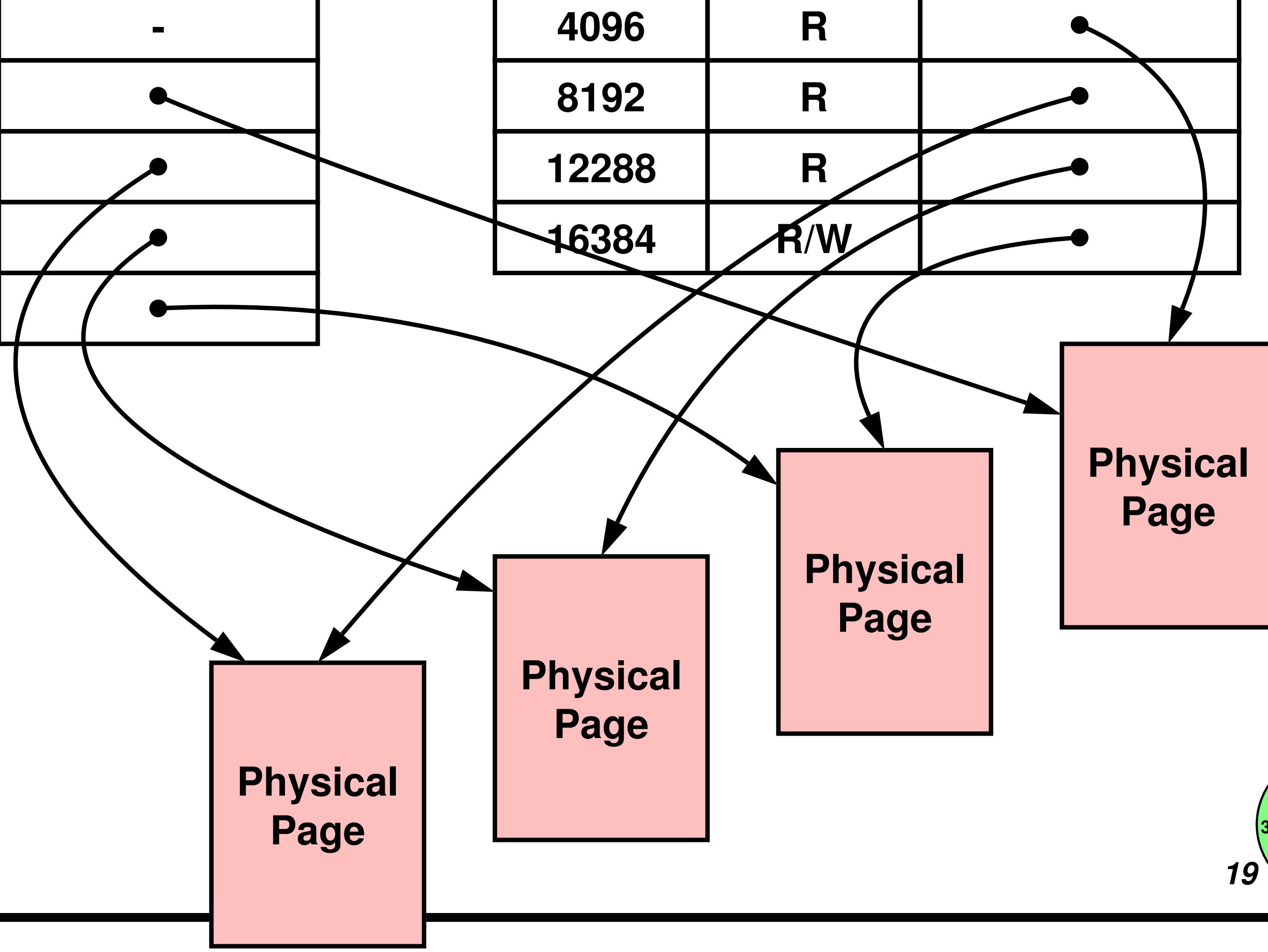
- Inside `fork()`, can simply copy parent's page table to child

Child Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	•
8192	R	•
12288	R	•
16384	R/W	•

Parent Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	•
8192	R	•
12288	R	•
16384	R/W	•

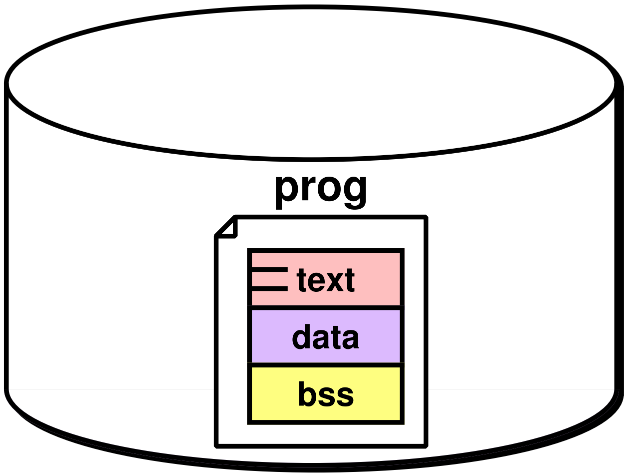


exec()

- 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	-	-
4096	-	-
8192	-	-
12288	-	-
16384	-	-



Memory Map

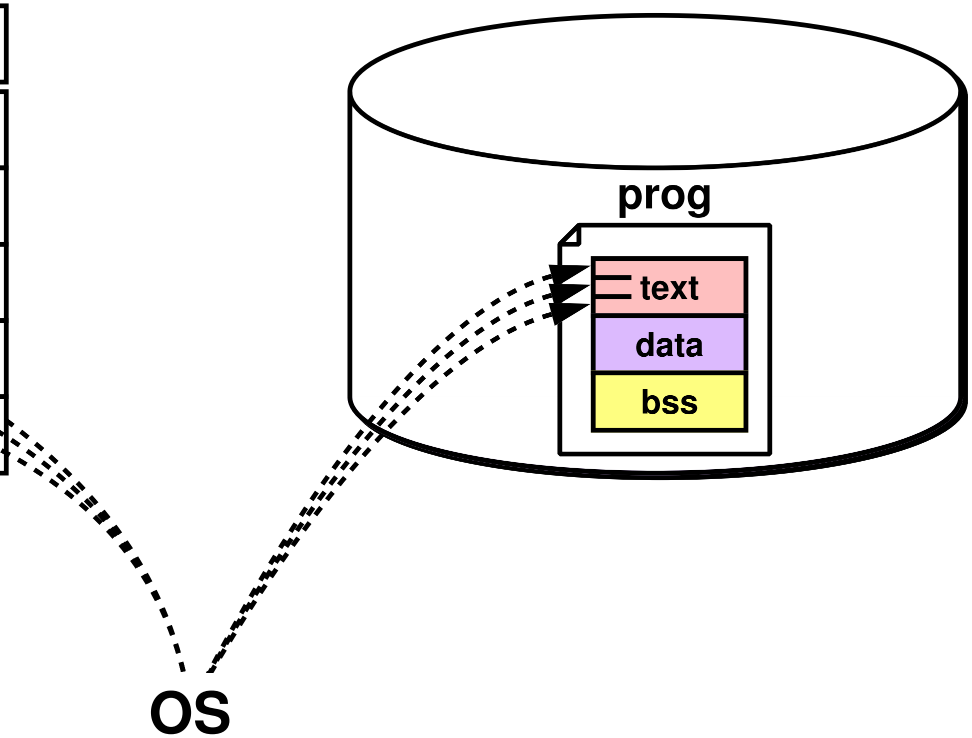
- **For the text region, why bother copying the executable file into the address space in the first place?**
  - **can just *map* the *file* into the *address space* (Ch 7)**
    - ***mapping* is an important concept in the OS**
    - ***mapping* let the OS *tie* the regions of the address space to the 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	-	-
4096	-	-
8192	-	-
12288	-	-
16384	-	-



  - the kernel also uses *memory map* to keep track of the mapping from *virtual pages* to *physical pages*

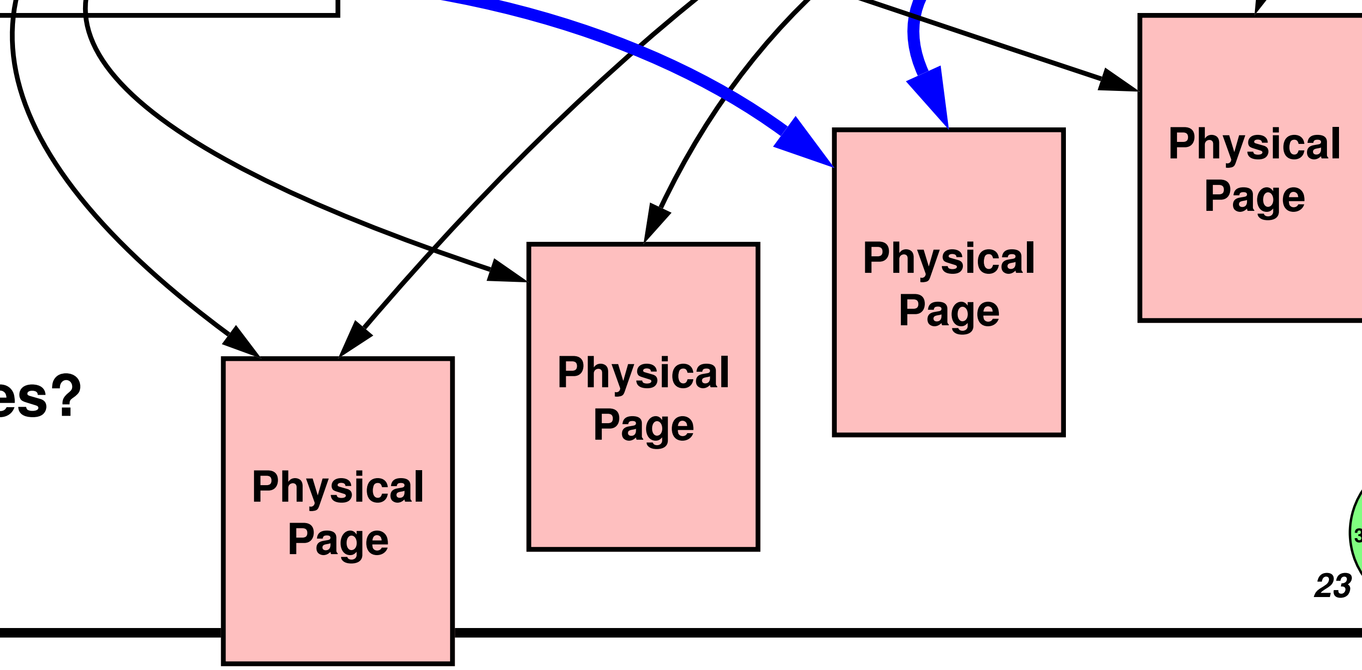
Processes Can Share Memory Pages

Parent Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	•
8192	R	•
12288	R	•
16384	R/W	•

Child Page Table

Start	*Access*	*Physical Addr*
0	-	-
4096	R	•
8192	R	•
12288	R	•
16384	R/W	•



- can we really share *data segment* pages?

Processes And File Systems

- ***Data regions*** **of all processes running this program** ***initially*** **refer to pages of memory containing a copy of 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* for the *first time***
    - **the basic idea is that only those pages of memory that are modified are copied**

- **The dynamic/heap and stack regions use a special form of private mapping**
  - **their pages are initialized, with zeros; *copy-on-write***
    - **these are known as *anonymous pages***
    - **managed by *anonymous objects* in `weenix`**

Shared Files

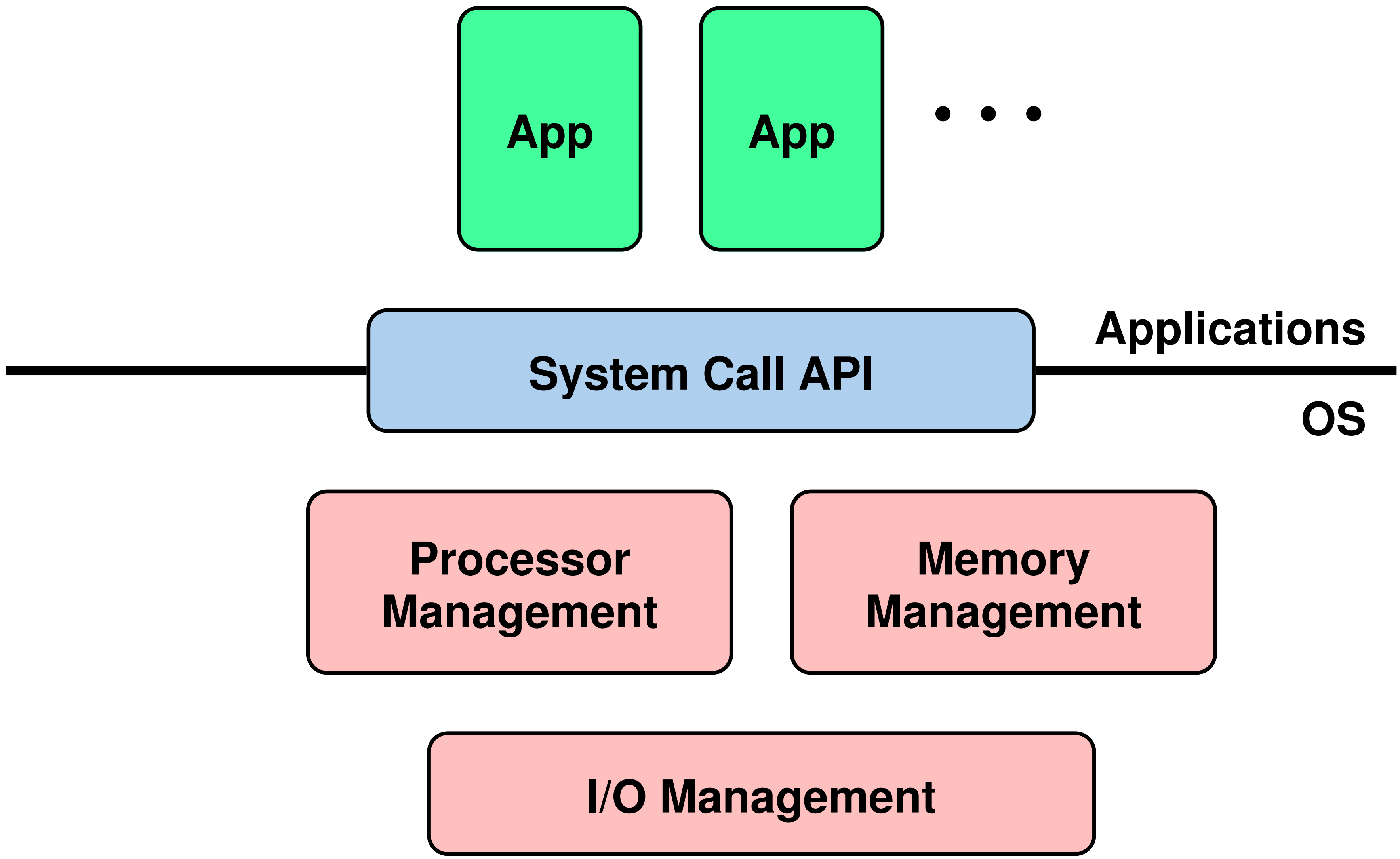
- If a bunch of processes *share* a *file*
  - we can also *map* the file into the *address space* of each process
  - in this case, the mapping is *shared*
  - when one process modifies a page, no private copy is made
    - instead, the original page itself is modified
    - everyone gets the changes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

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
  - this is referred to 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

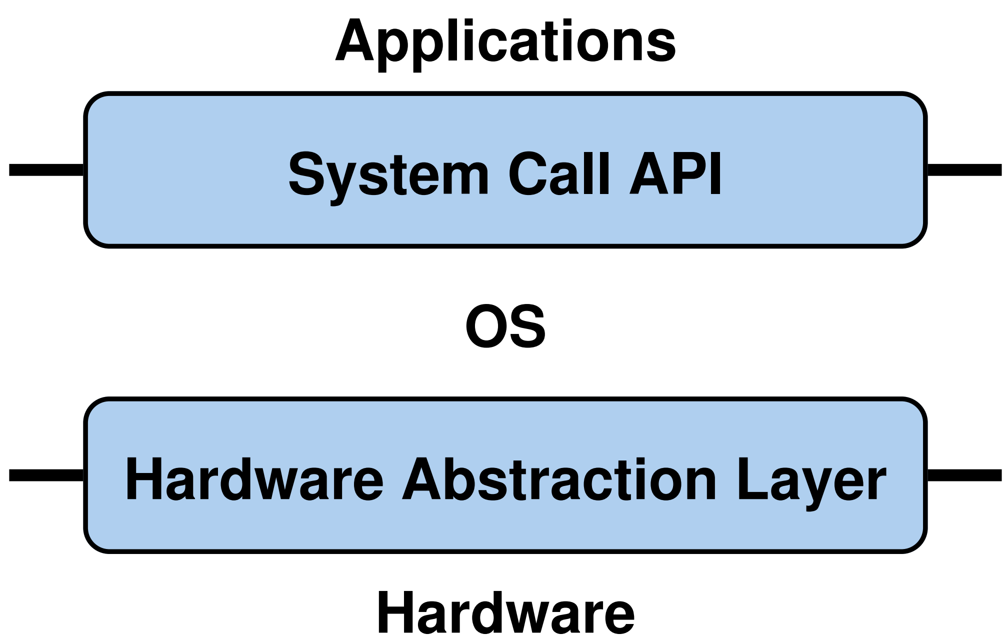
- Backwards compatibility is an important issue
  - try not to change it much (to make the developers happy)



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

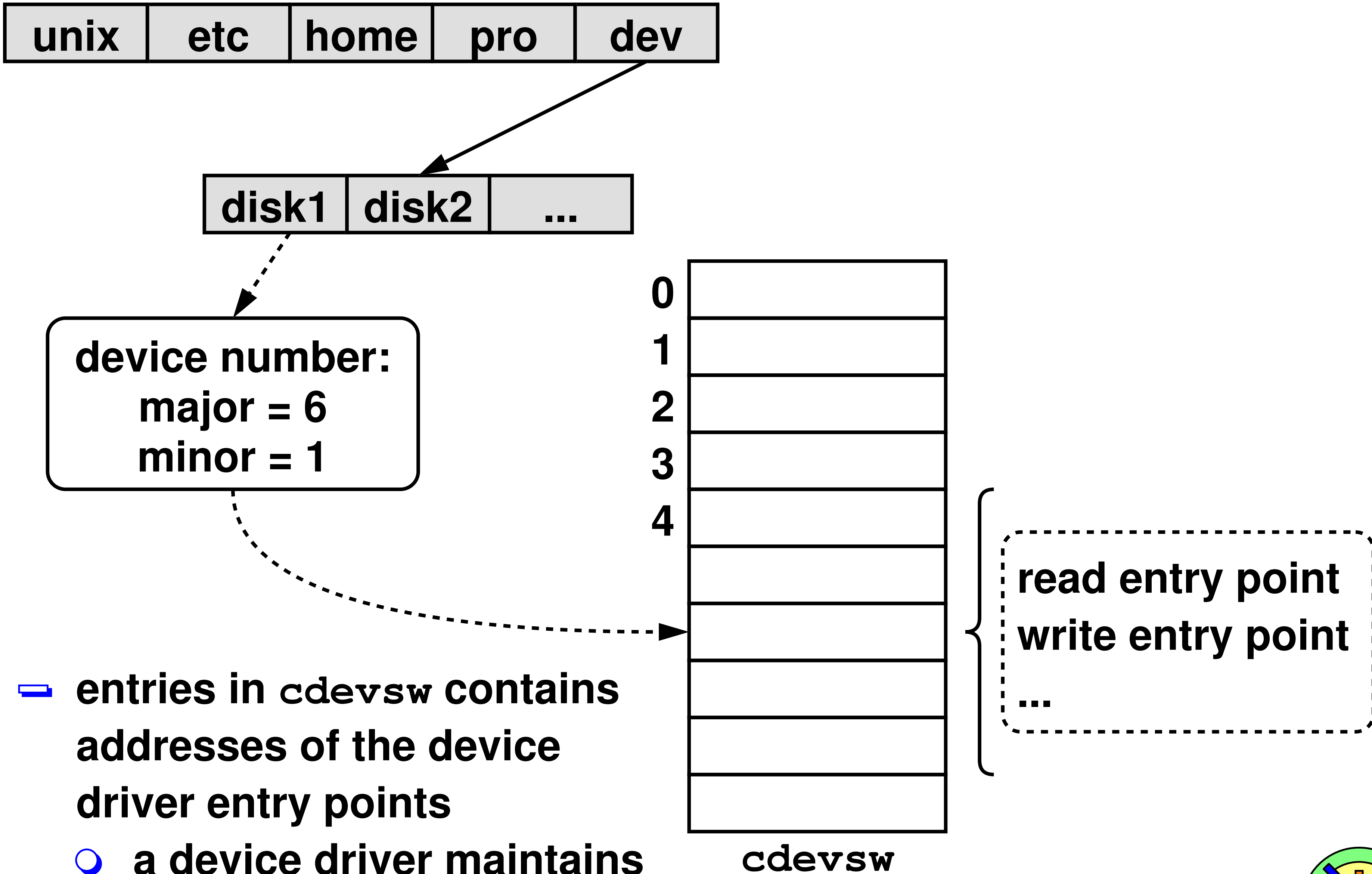
A Framework for Devices

- 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)

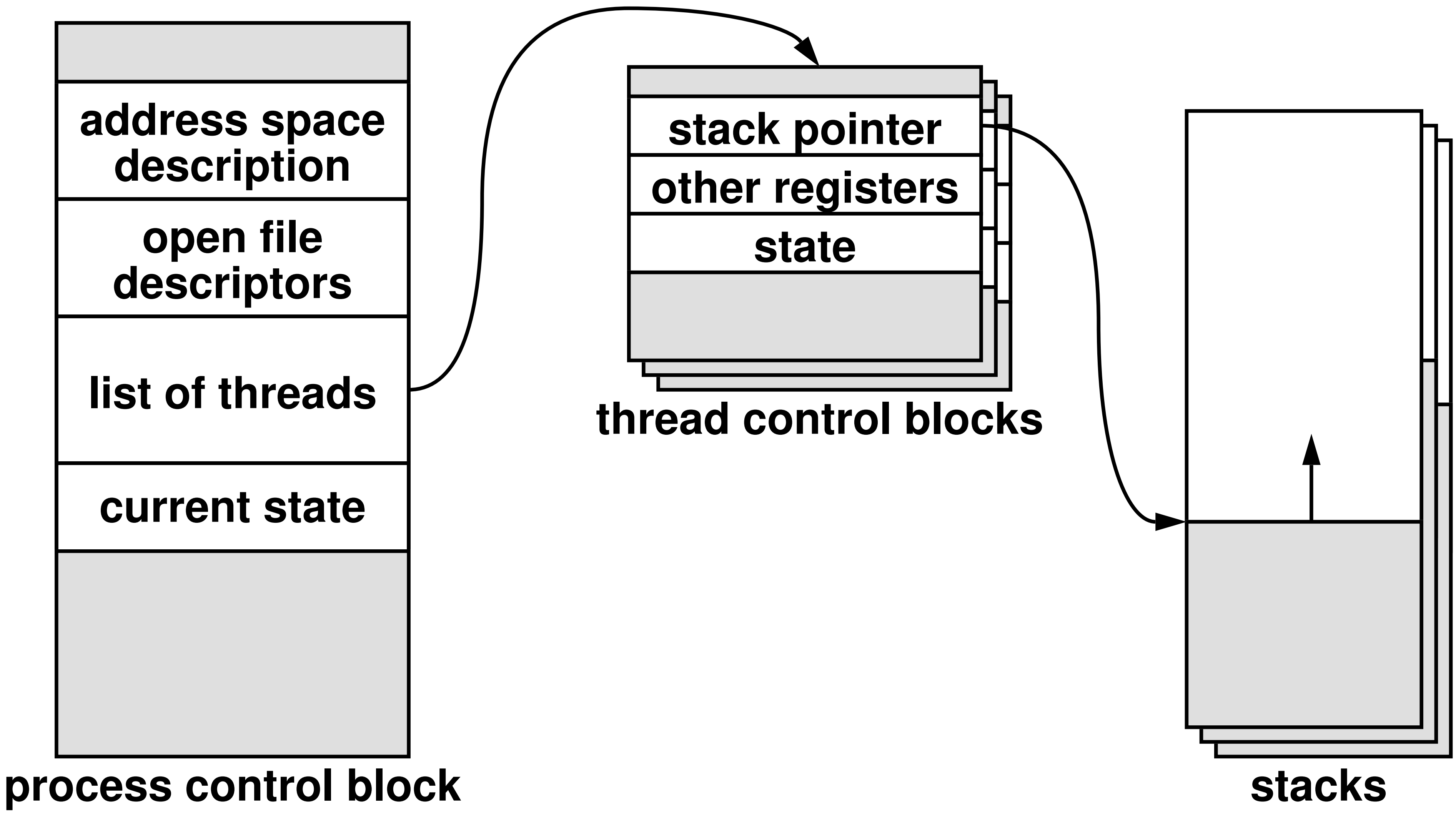
4.1 A Simple System (Monolithic Kernel)

- 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()`

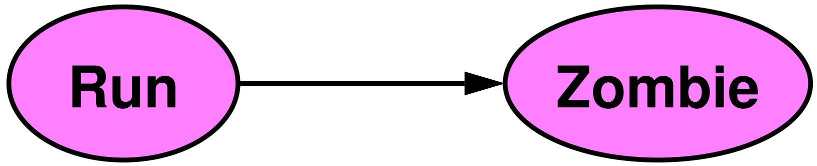
Processes and Threads



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

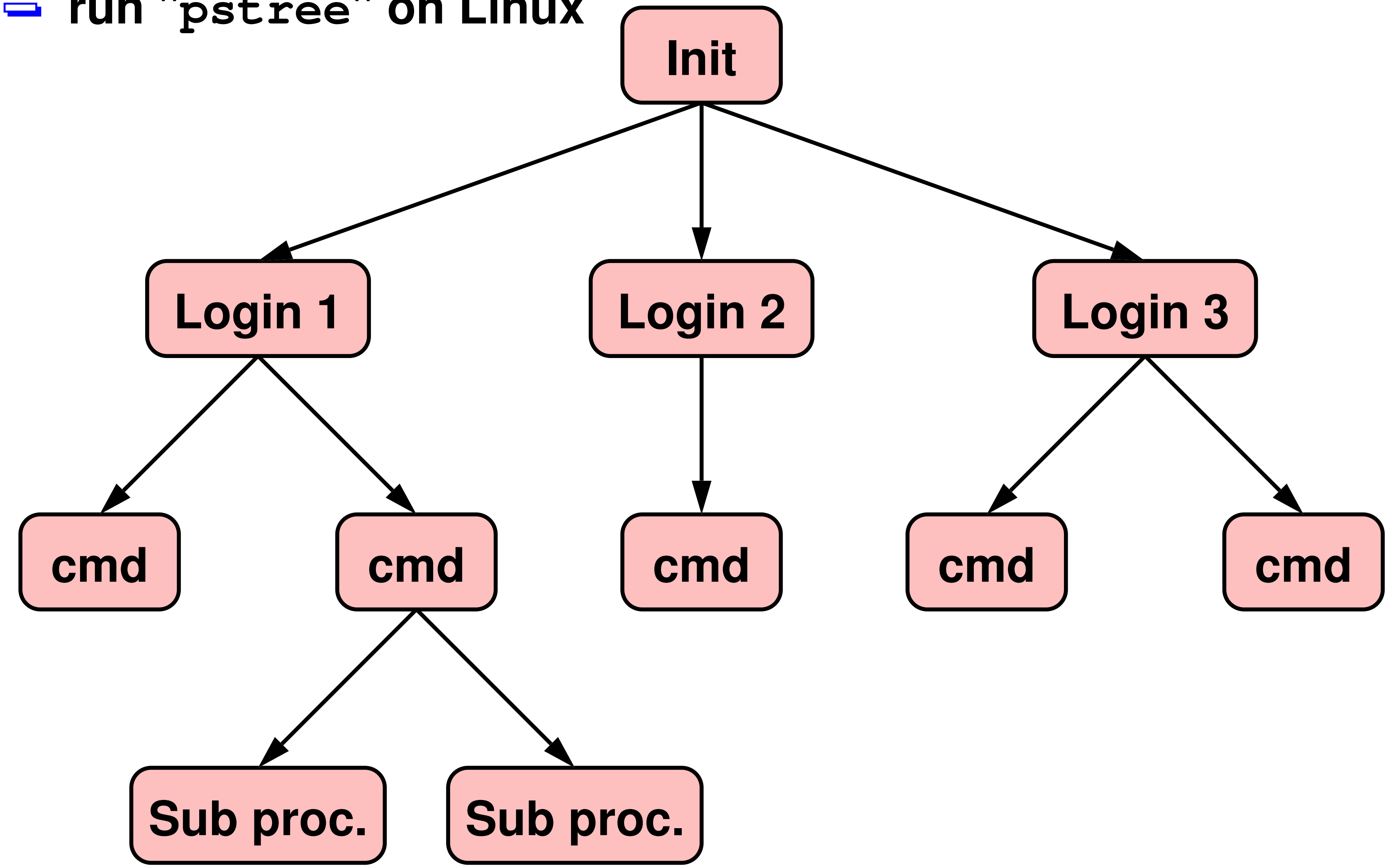
Process Life Cycle

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

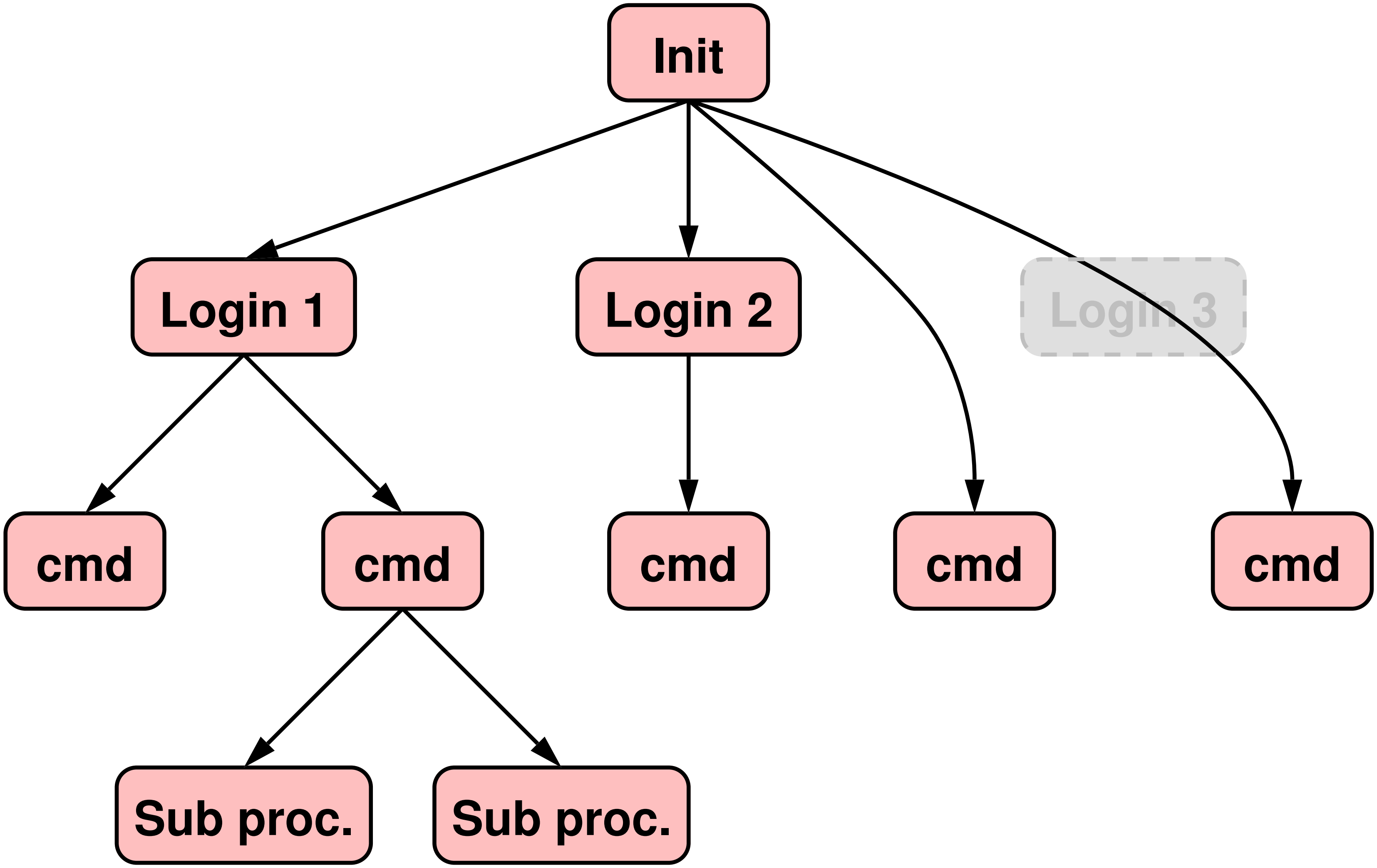


Process Relationships (1)

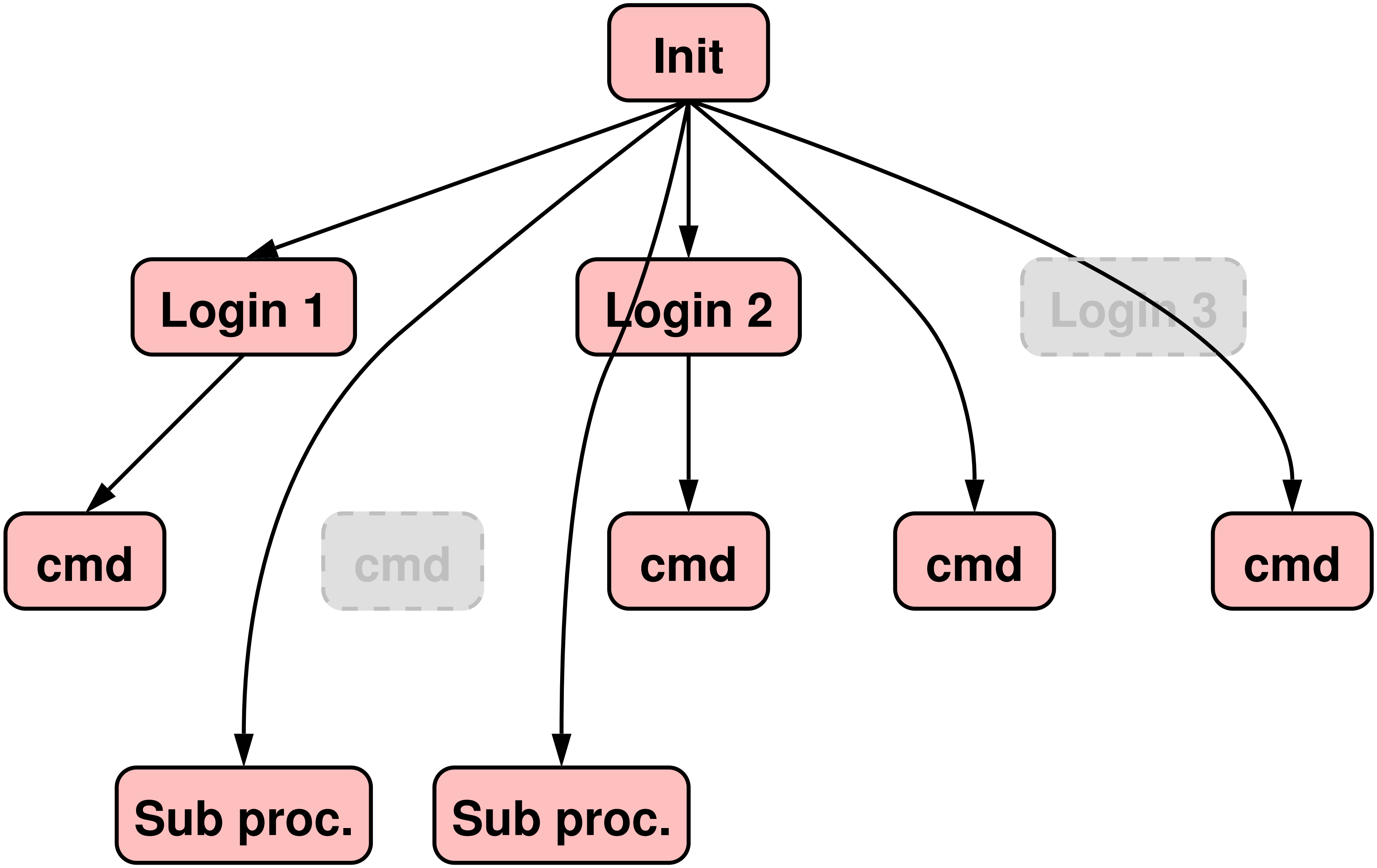
- Process hierarchy
  - run "`pstree`" on Linux



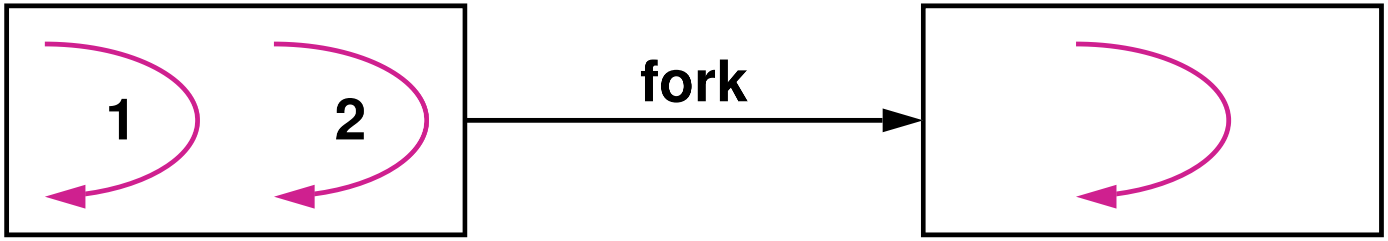
Process Relationships (2)



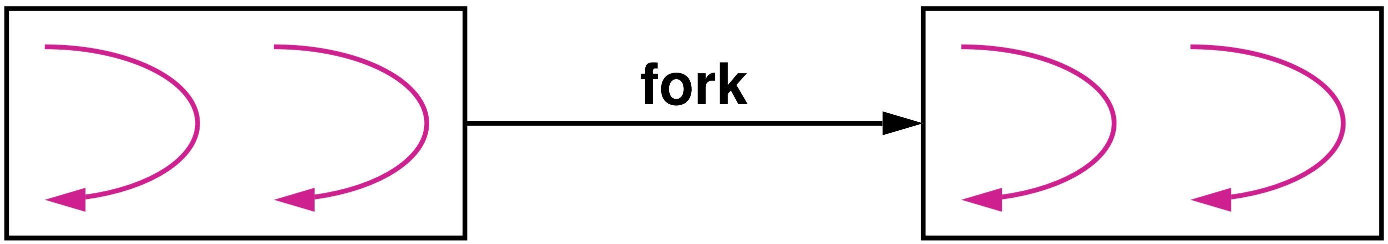
Process Relationships (3)



Fork and Threads

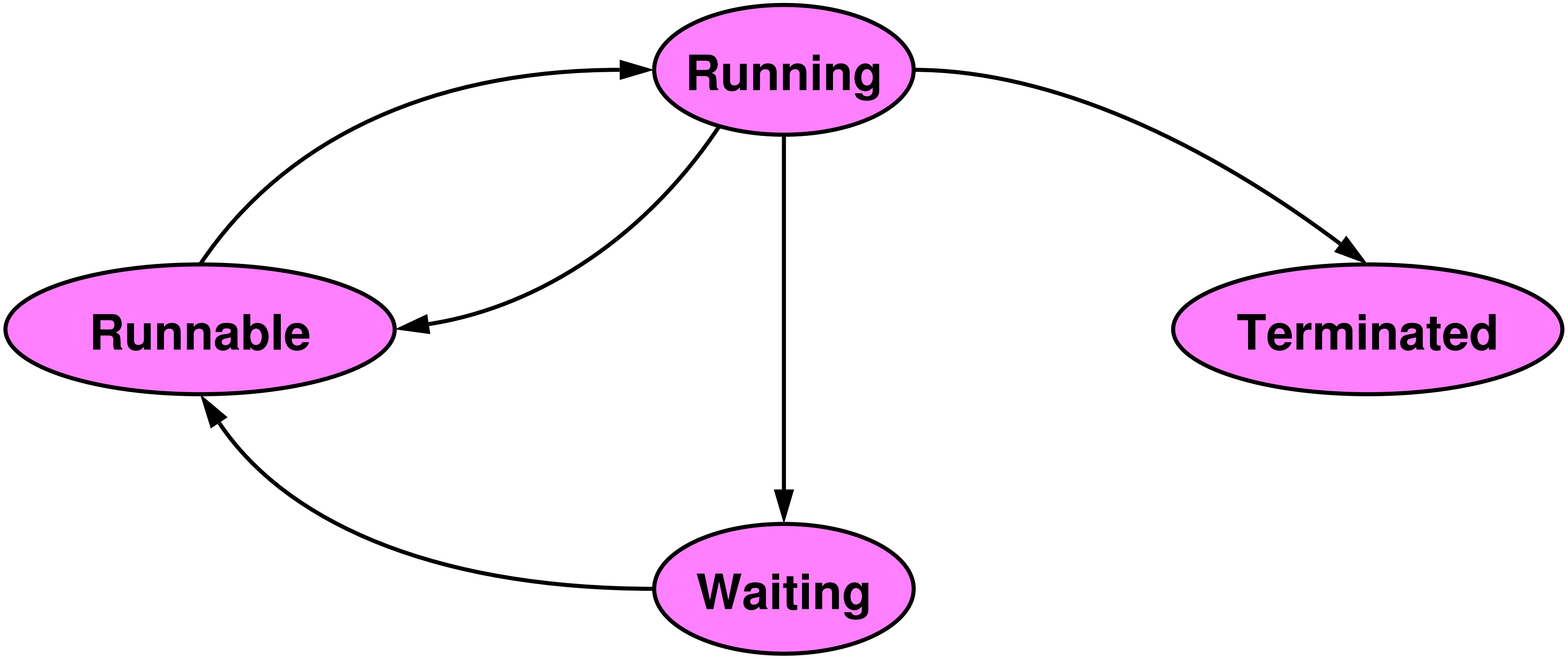


Or



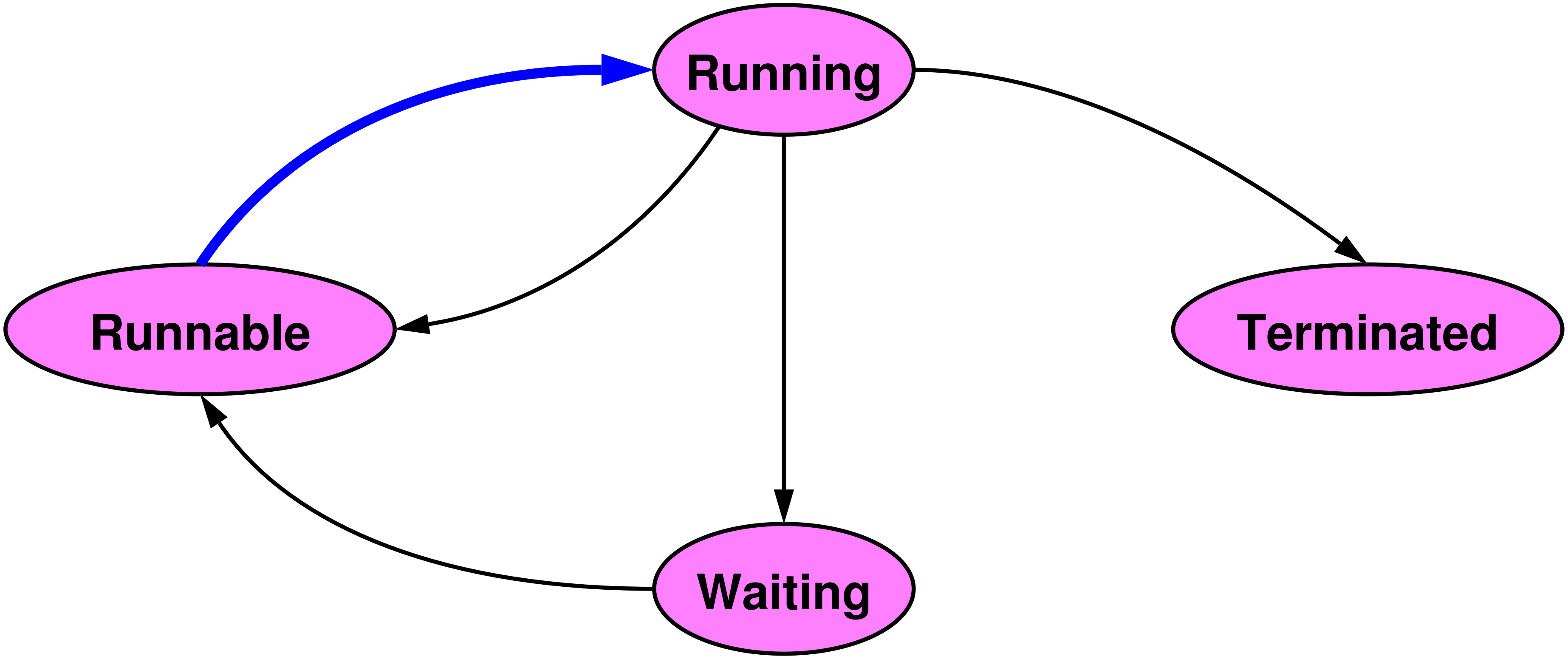
- Solaris uses the 2nd approach
  - expensive to fork a process
- Problem with 1st approach
  - thread 1 called `fork()` and thread 2 has a mutex locked
    - who will unlock the mutex?
  - POSIX solution is to provide a way to unlock all mutex before `fork()`

Thread Life Cycle



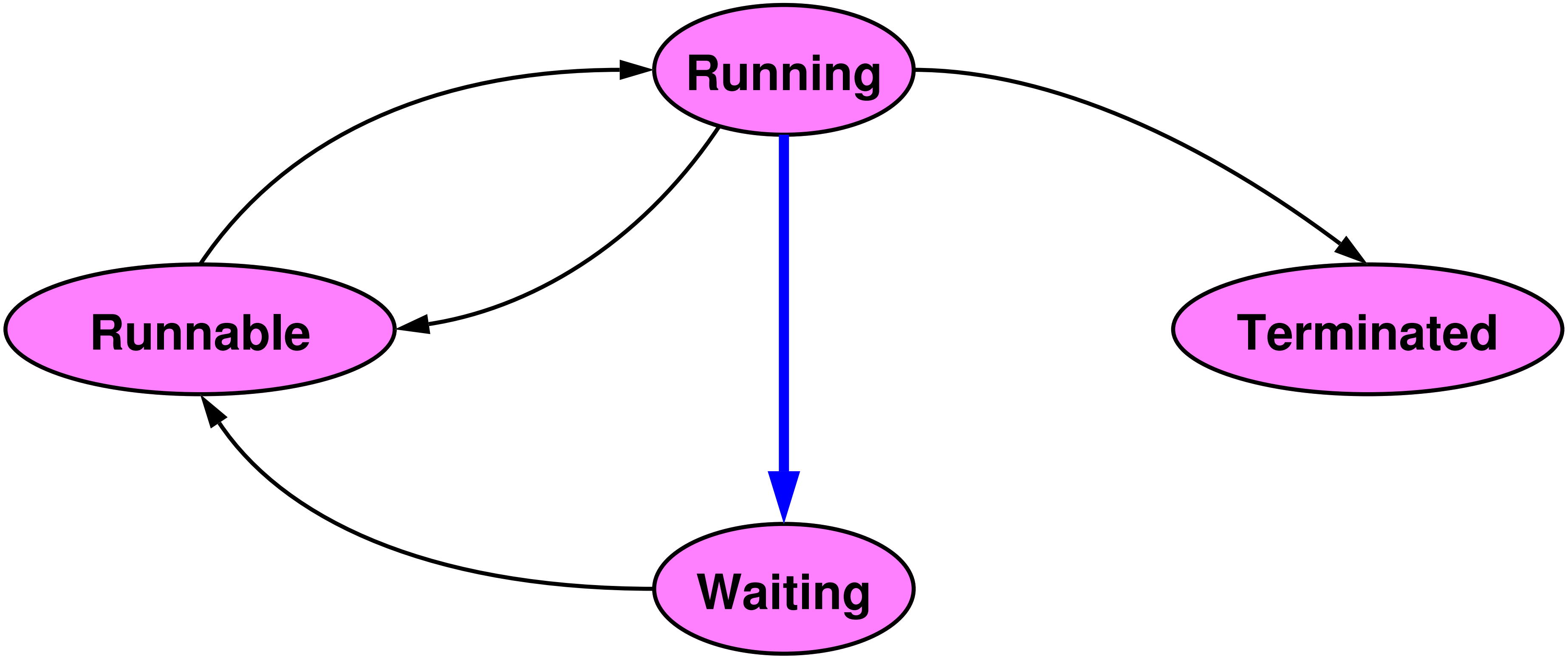
- a thread starts in the *runnable* state

Thread Life Cycle



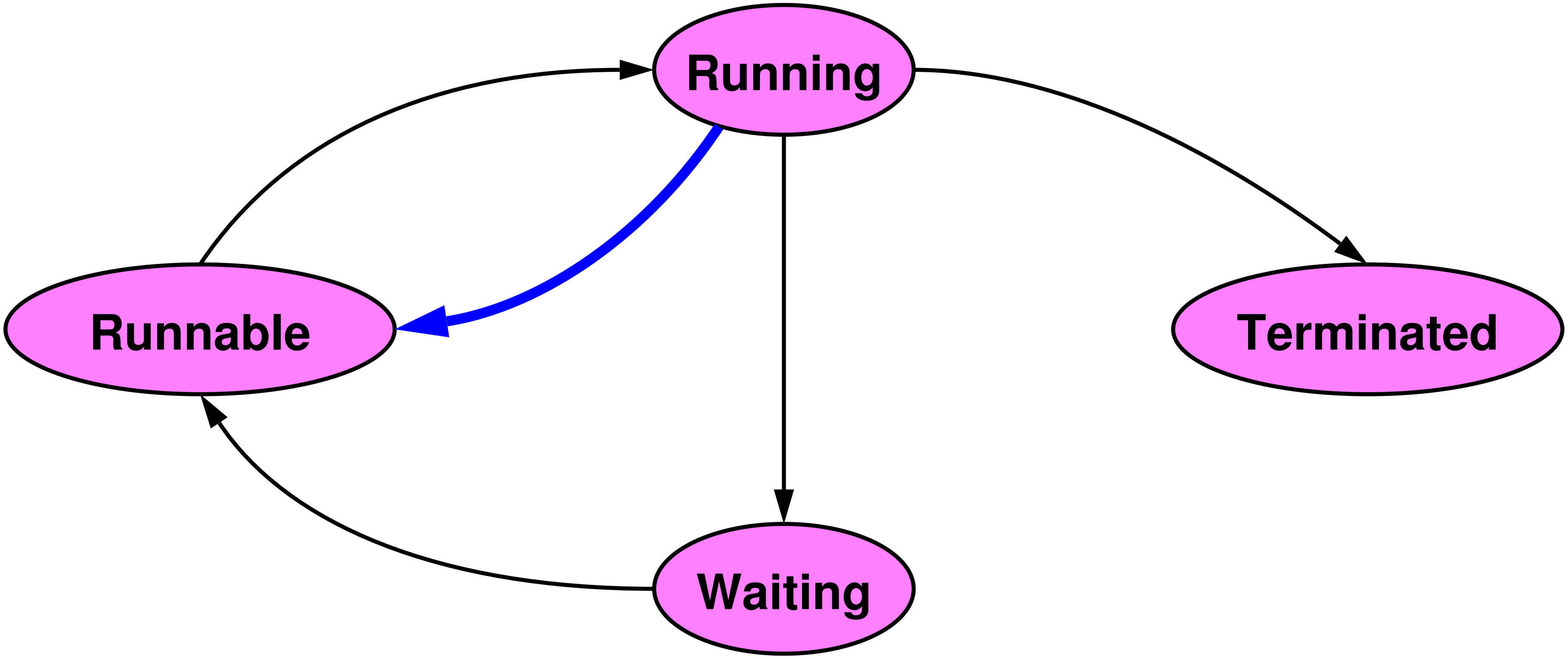
- a thread starts in the *runnable* state
- the *scheduler* switches a thread's state from runnable to running

Thread Life Cycle



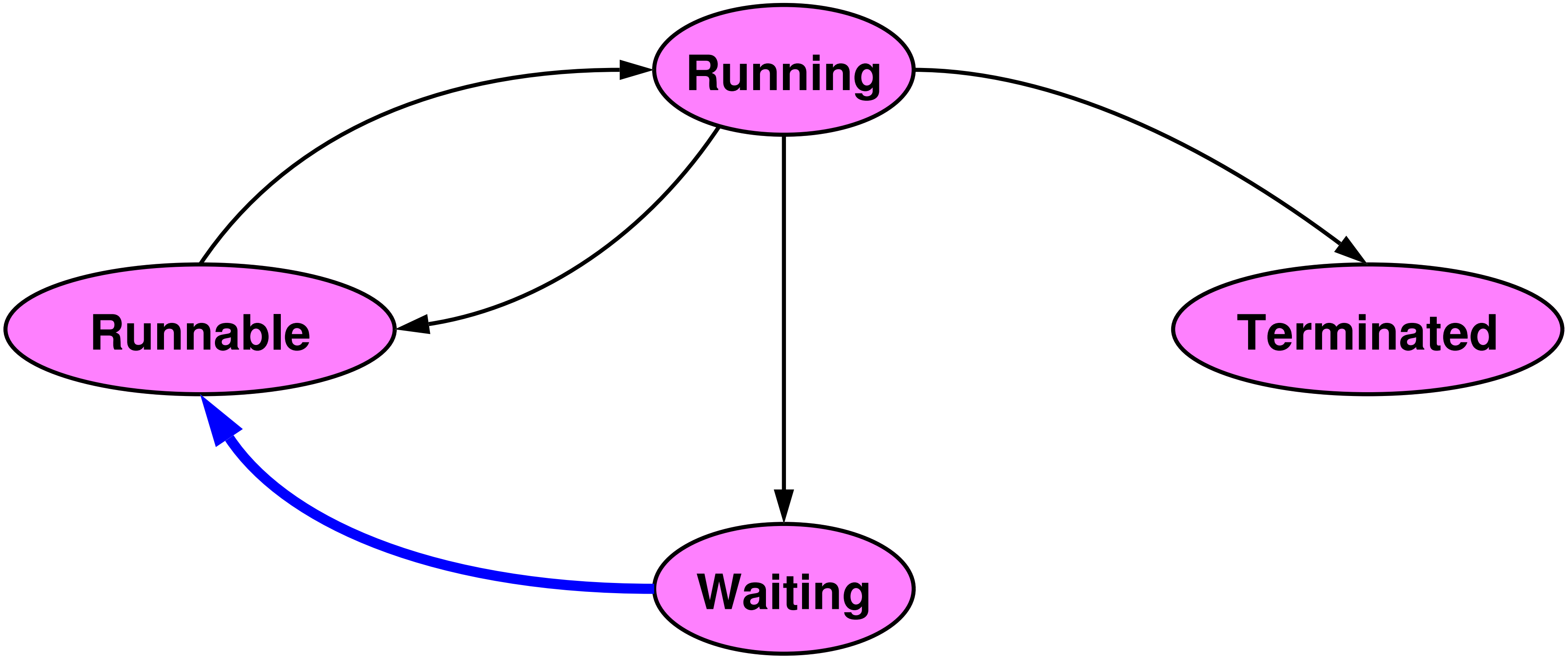
- a thread starts in the *runnable* state
- the *scheduler* switches a thread's state from runnable to running
- a thread goes from running to waiting when a *blocking call* is made

Thread Life Cycle



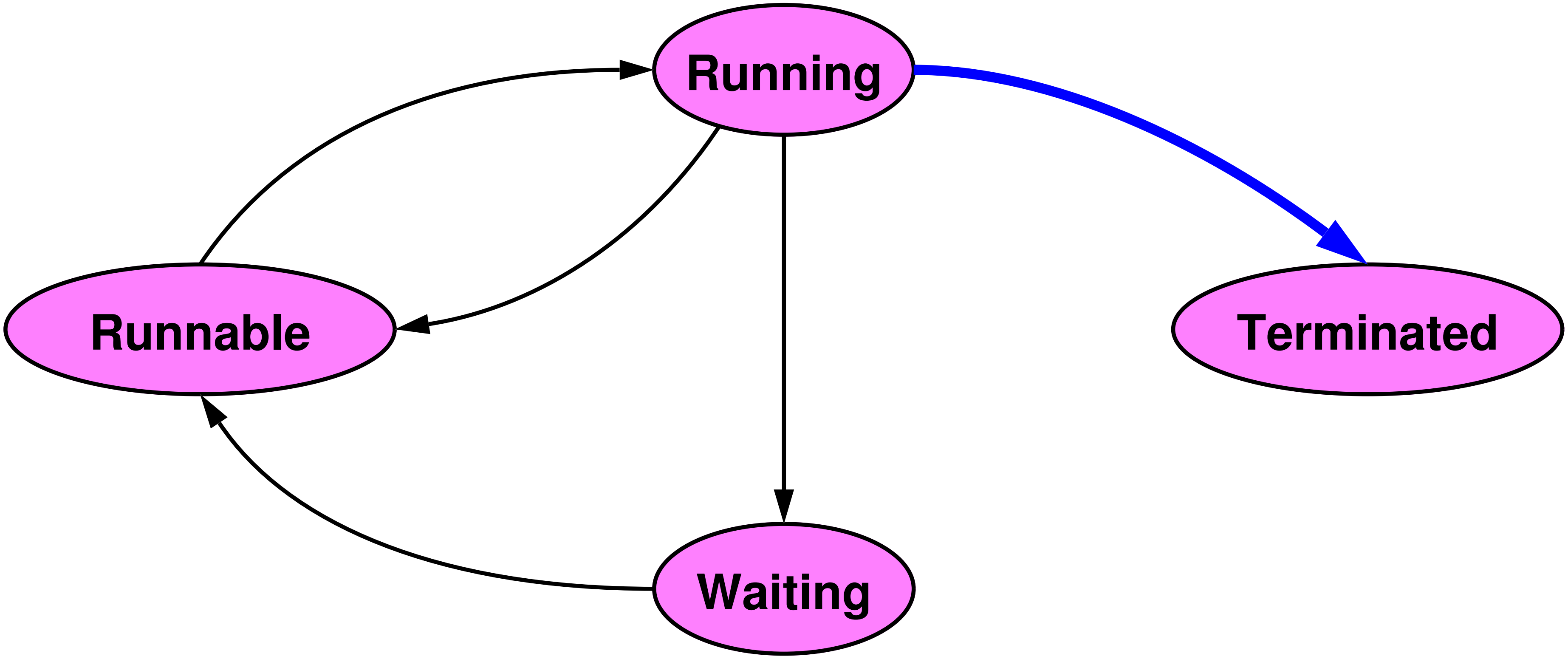
- a thread starts in the *runnable* state
- the *scheduler* switches a thread's state from runnable to running
- a thread goes from running to waiting when a *blocking call* is made
- the *scheduler* switches a thread's state from running to runnable when the thread used up its execution quantum

Thread Life Cycle



- a thread get *unblocked* by the action of another thread or by an interrupt handler

Thread Life Cycle



- a thread get ***unblocked*** by the action of another thread or by an interrupt handler
- in order for a thread to enter the terminated state, it has to be in the running state just before that
  - what if `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
- 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!

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)