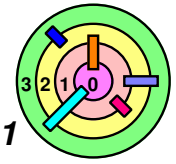
4.1 A Simple System (Monolithic Kernel)

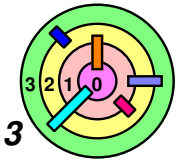
- *A Framework for Devices*
- Low-level Kernel (will come back to talk about this after Ch 7)
- Processes & Threads
- Storage Management (will come back to talk about this after Ch 5)



Computer Terminal

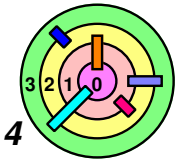
VT100

A "tty"



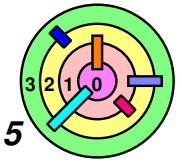
Devices

- 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

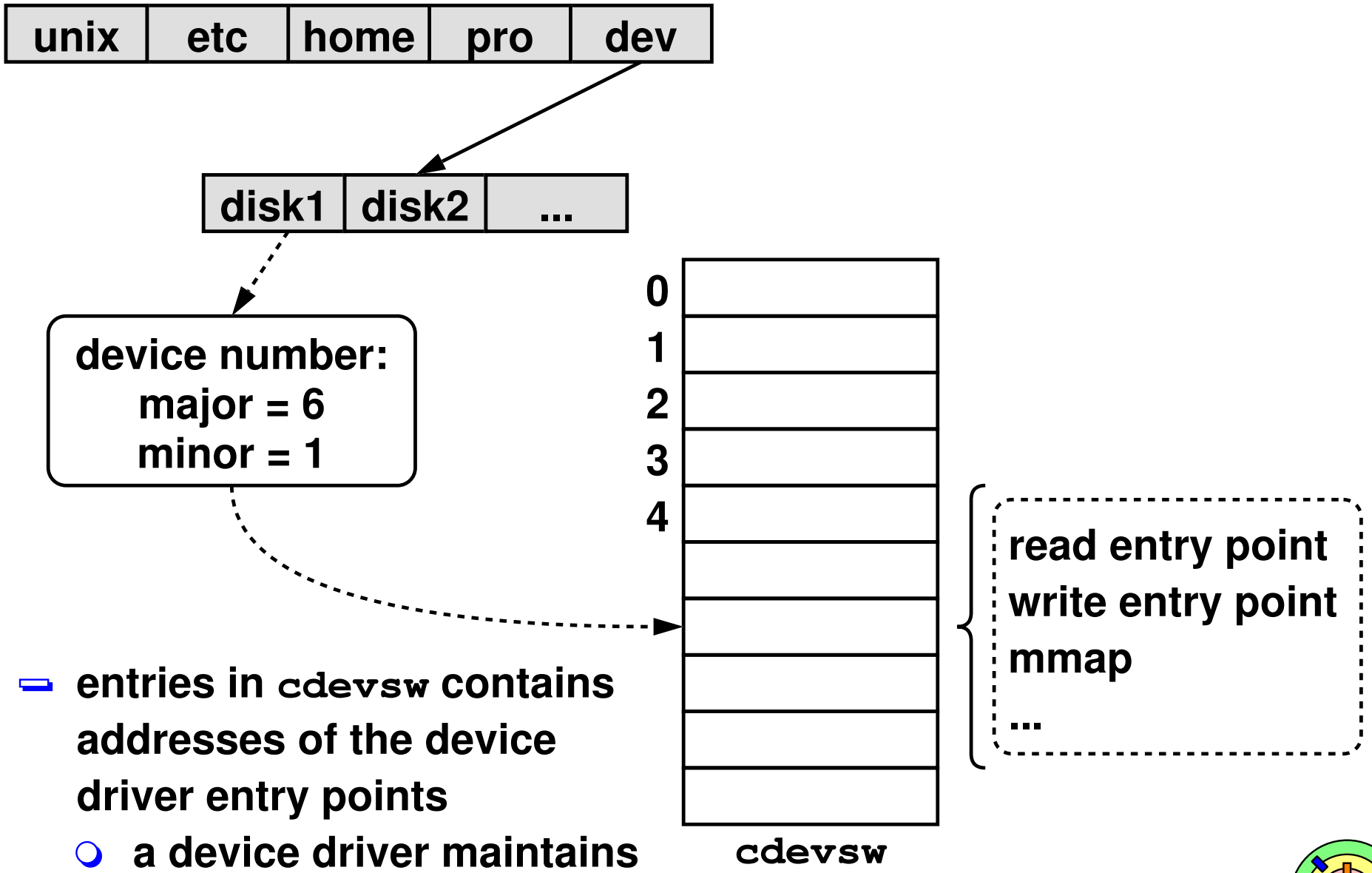


A Framework for Devices

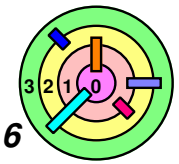
- Device driver:
  - every device is identified by a device "number", which is actually a pair of numbers
    - a *major device number* - identifies the device driver
    - a *minor device number* - device index for all devices managed by the same device driver
- Special entries were created in the file 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

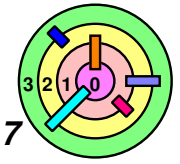


- entries in `cdevsw` contains addresses of the device driver entry points
  - a device driver maintains its own data structure



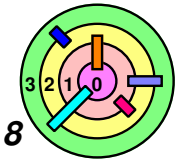
Device Drivers in Early Unix Systems

- 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



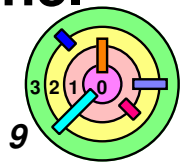
Device Probing

- 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



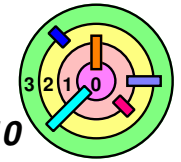
Device Probing

- 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?



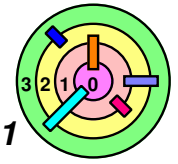
Discovering Devices

- 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



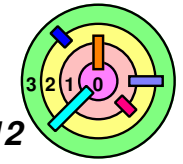
Discovering Devices

- 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)



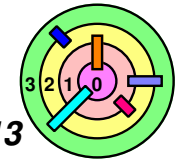
4.1 A Simple System (Monolithic Kernel)

- 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)

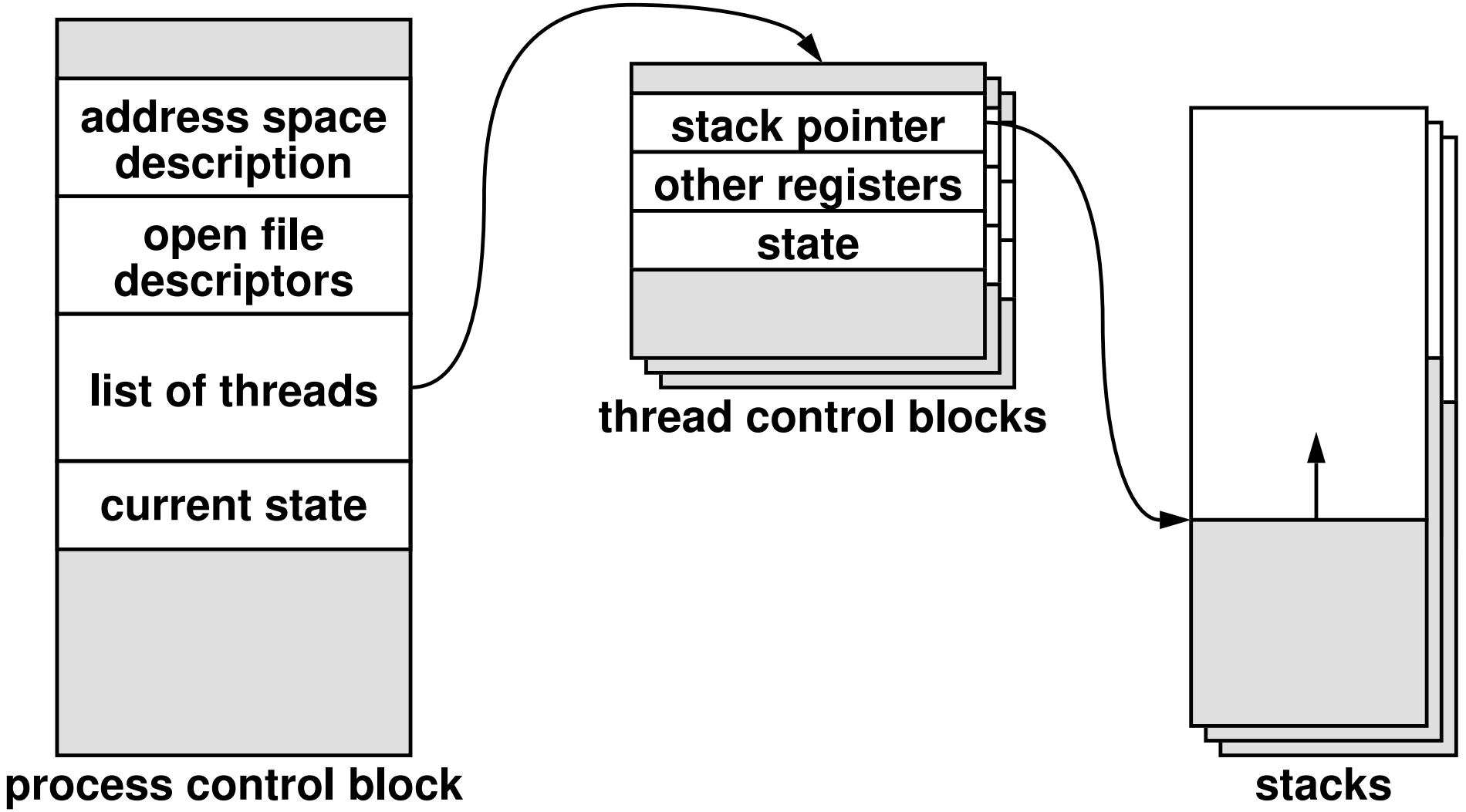


Processes and Threads

- 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()`



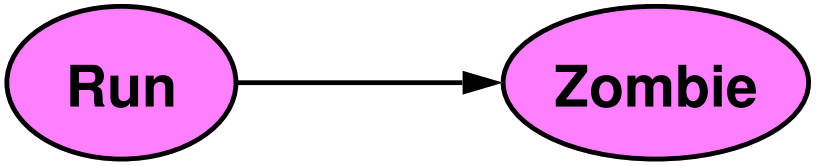
Processes and Threads

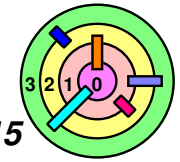


- Note: all these are relevant to your Kernel Assignment 1
  - although we are only doing *one thread per process*

Process Life Cycle

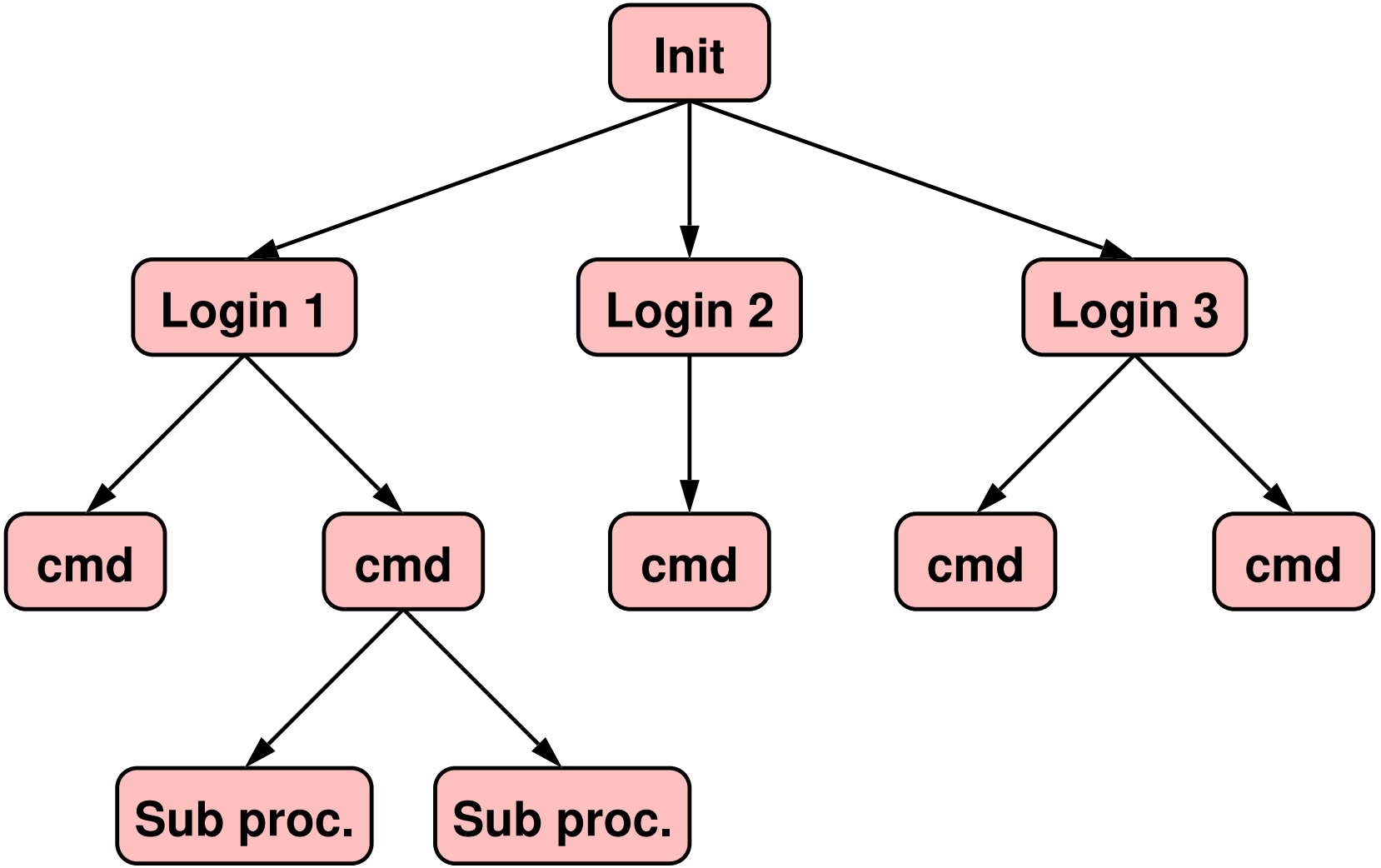
- Pretty simple
  - a process starts in the *run* state

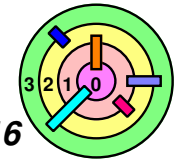




Process Relationships (1)

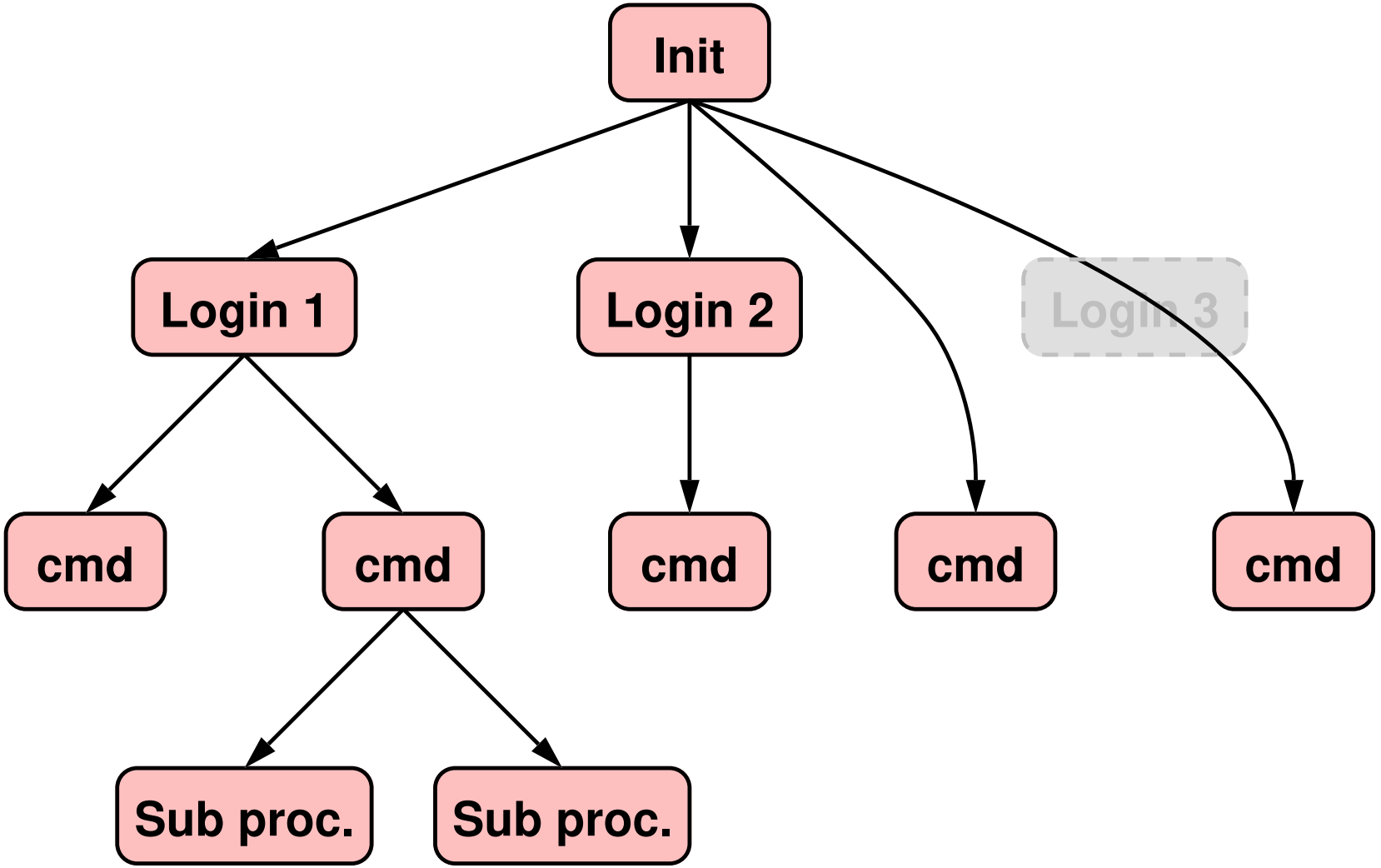
- Process hierarchy
  - run "`pstree`" on Linux

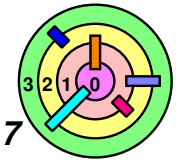




Process Relationships (2)

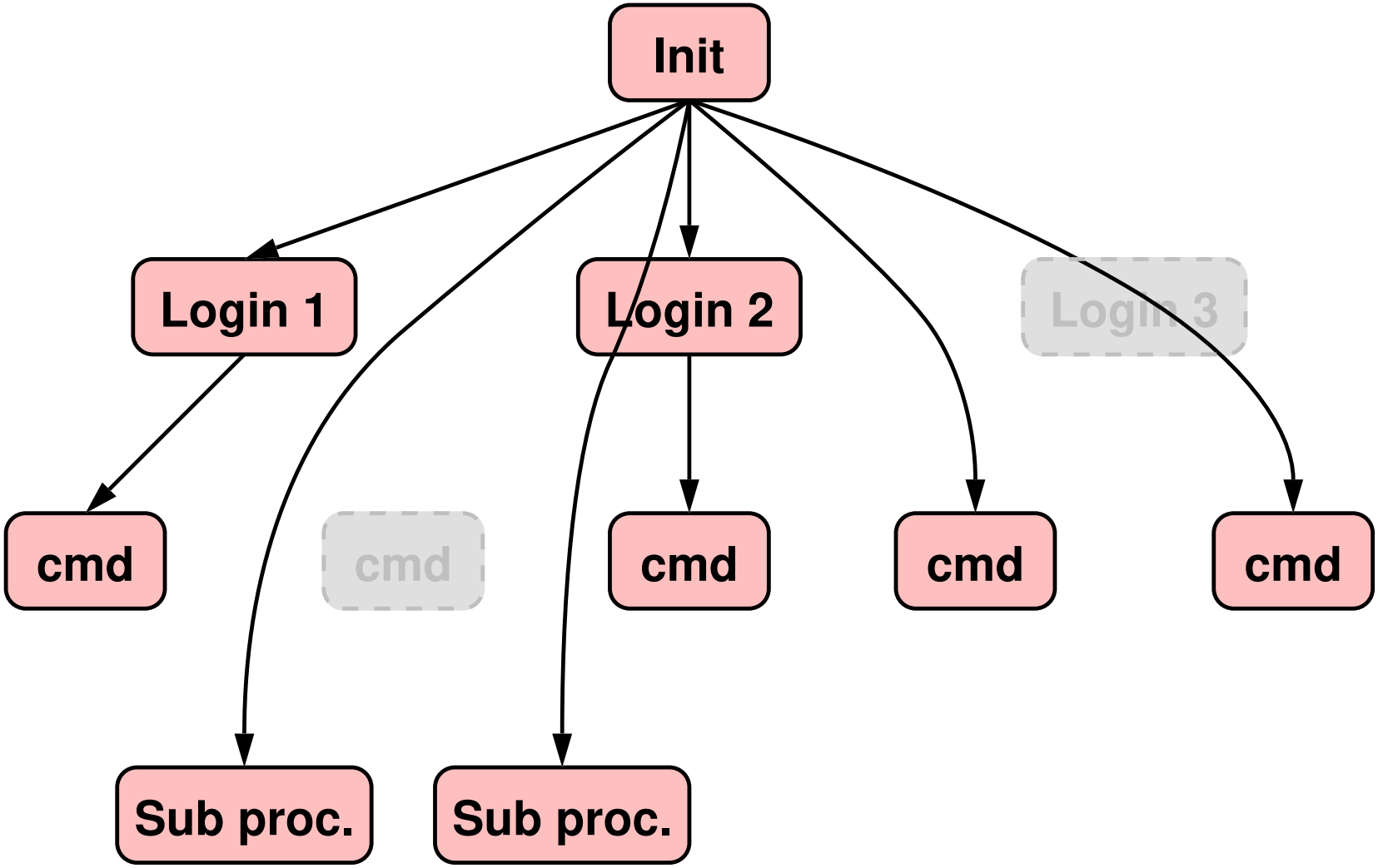
- If a process dies, you must *reparent* all its child processes

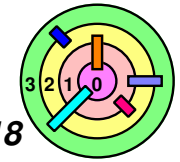




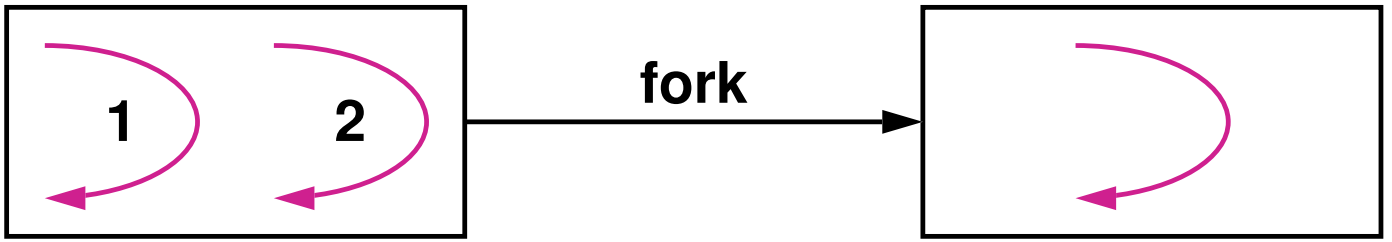
Process Relationships (3)

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

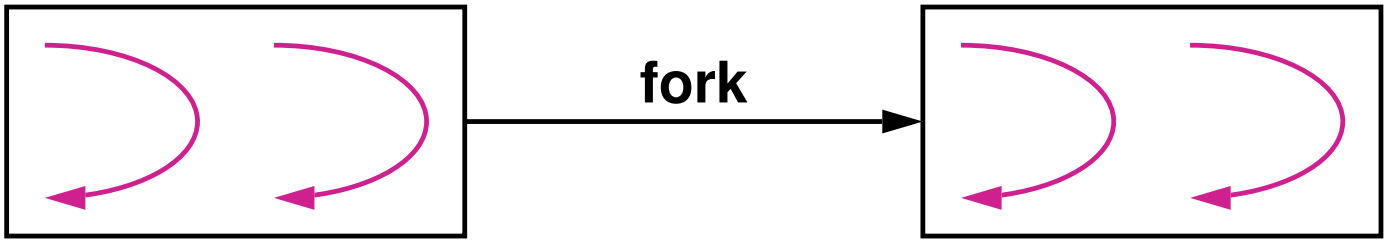




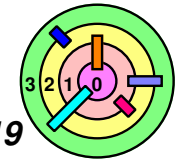
Fork and Threads



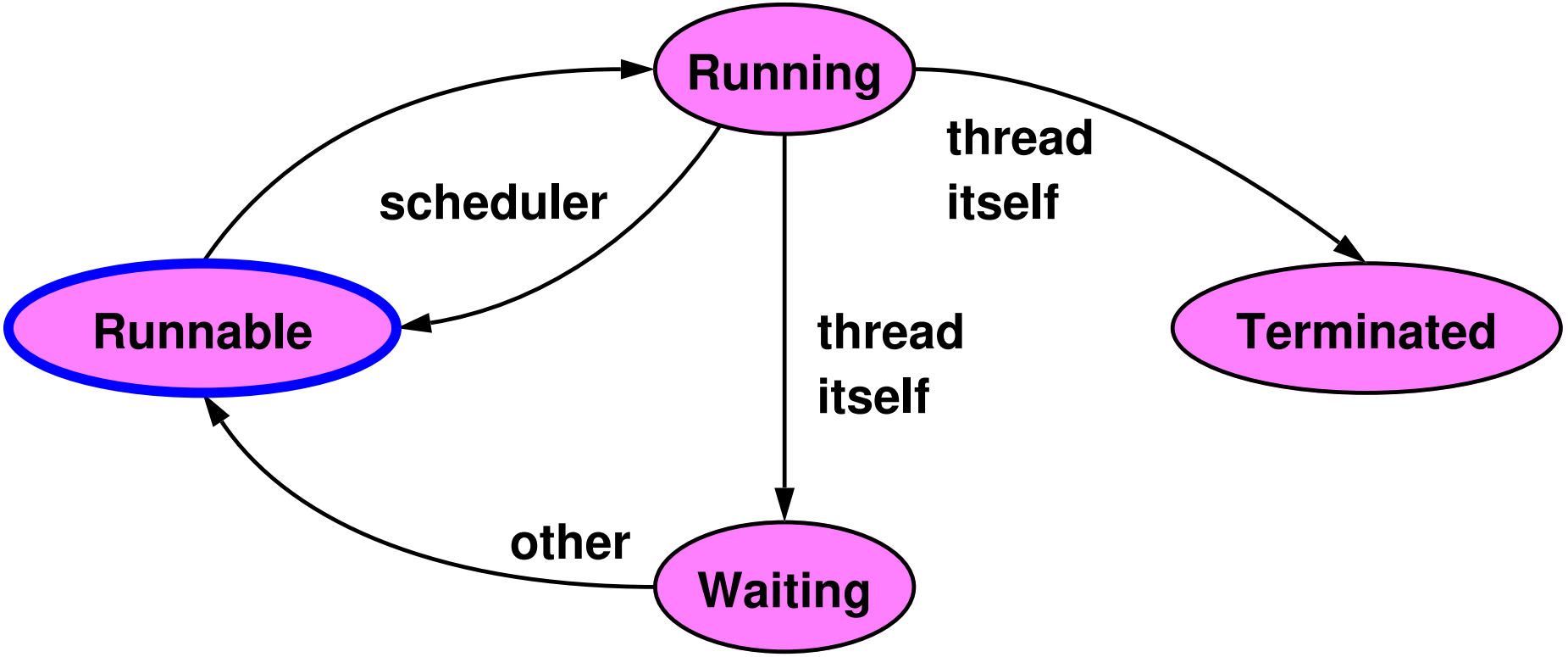
Or



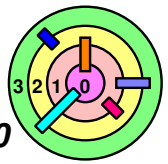
- Solaris uses the 2nd approach
  - expensive to fork a process
- Problem with 1st approach
  - thread 1 called `fork()` and thread 2 has a mutex locked
    - who will unlock the mutex?
  - POSIX solution is to provide a way to unlock all mutex before `fork()`



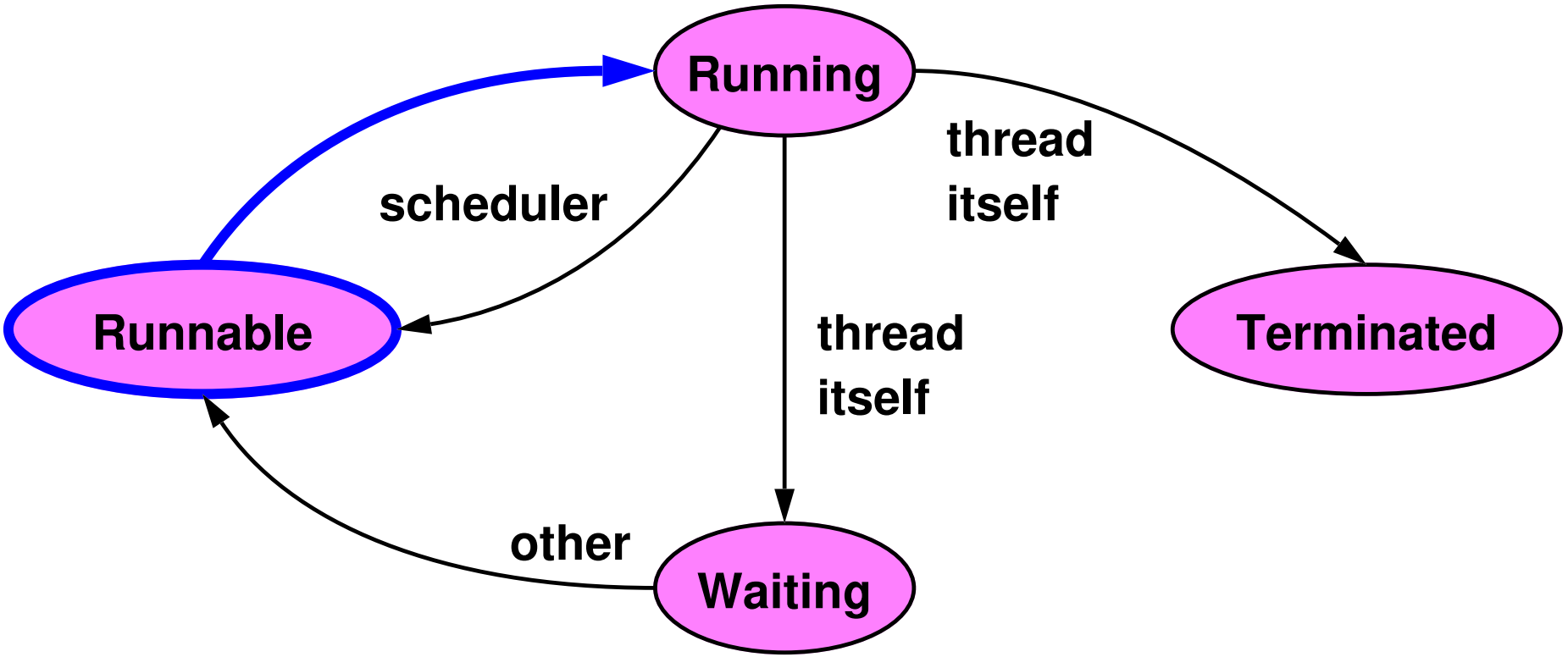
Thread Life Cycle



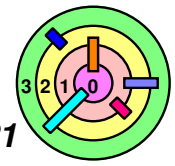
- a thread starts in the *runnable* state
  - *sleeps* in the *run queue* (or "ready queue")
    - threads sleep in the run queue to wait to use the CPU



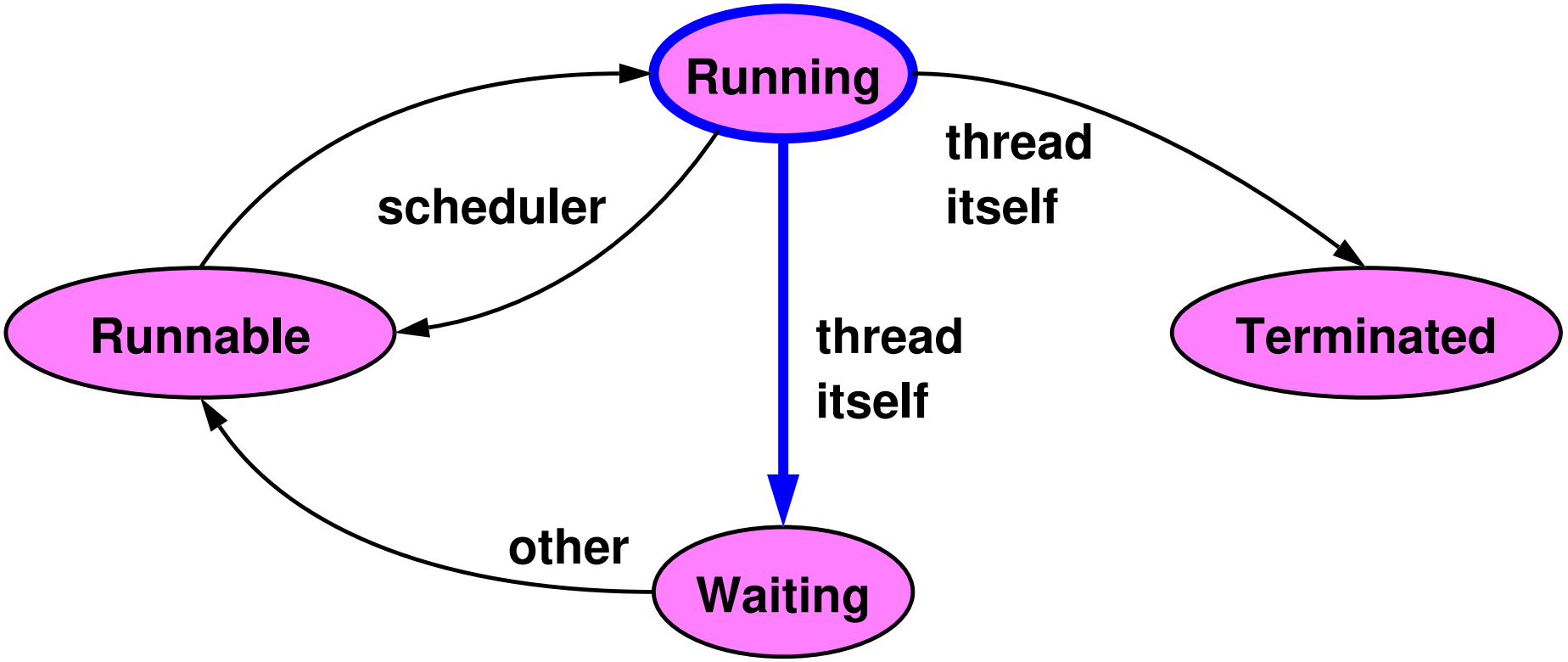
Thread Life Cycle



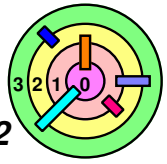
- the *scheduler* switches a thread's state from runnable to running
  - the scheduler decides who to run next inside the CPU



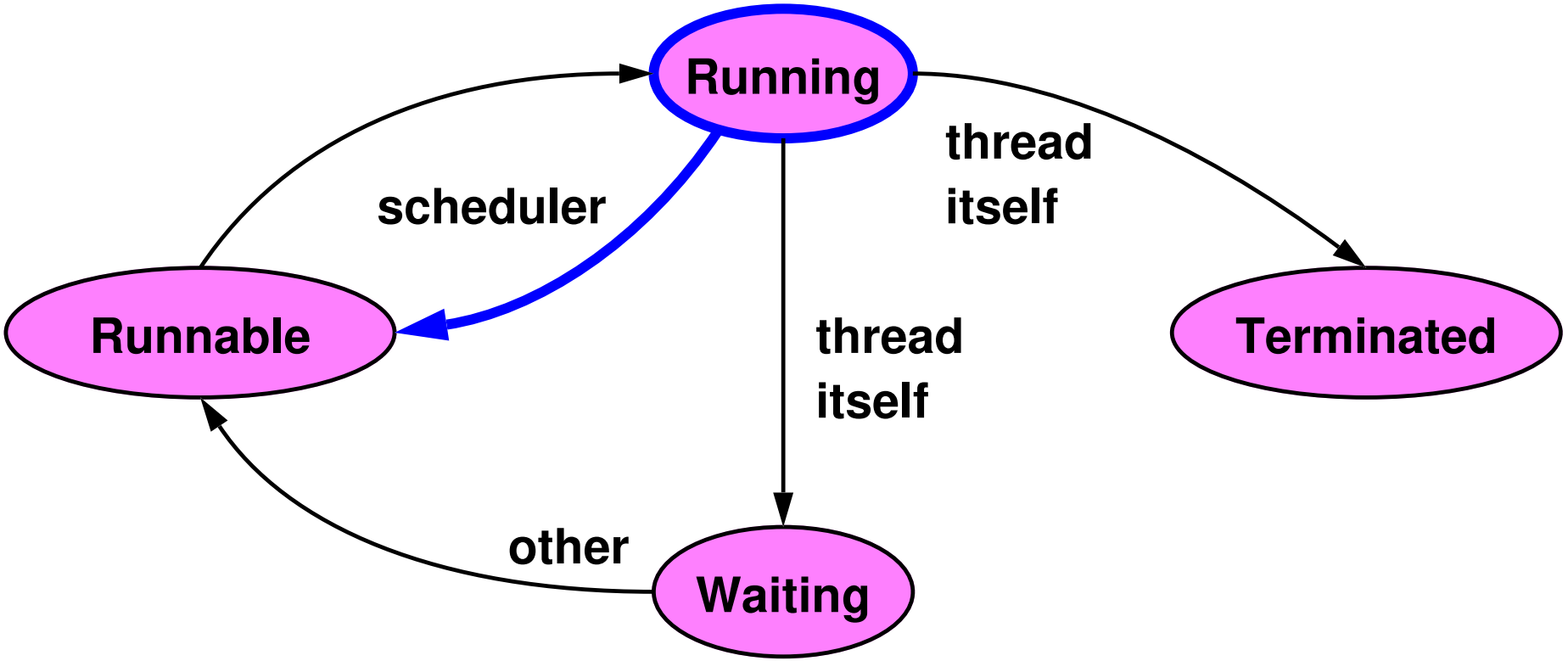
Thread Life Cycle



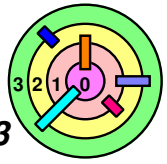
- a thread goes from running to waiting when a *blocking call* is made by the thread itself
  - the scheduler is *not involved* here



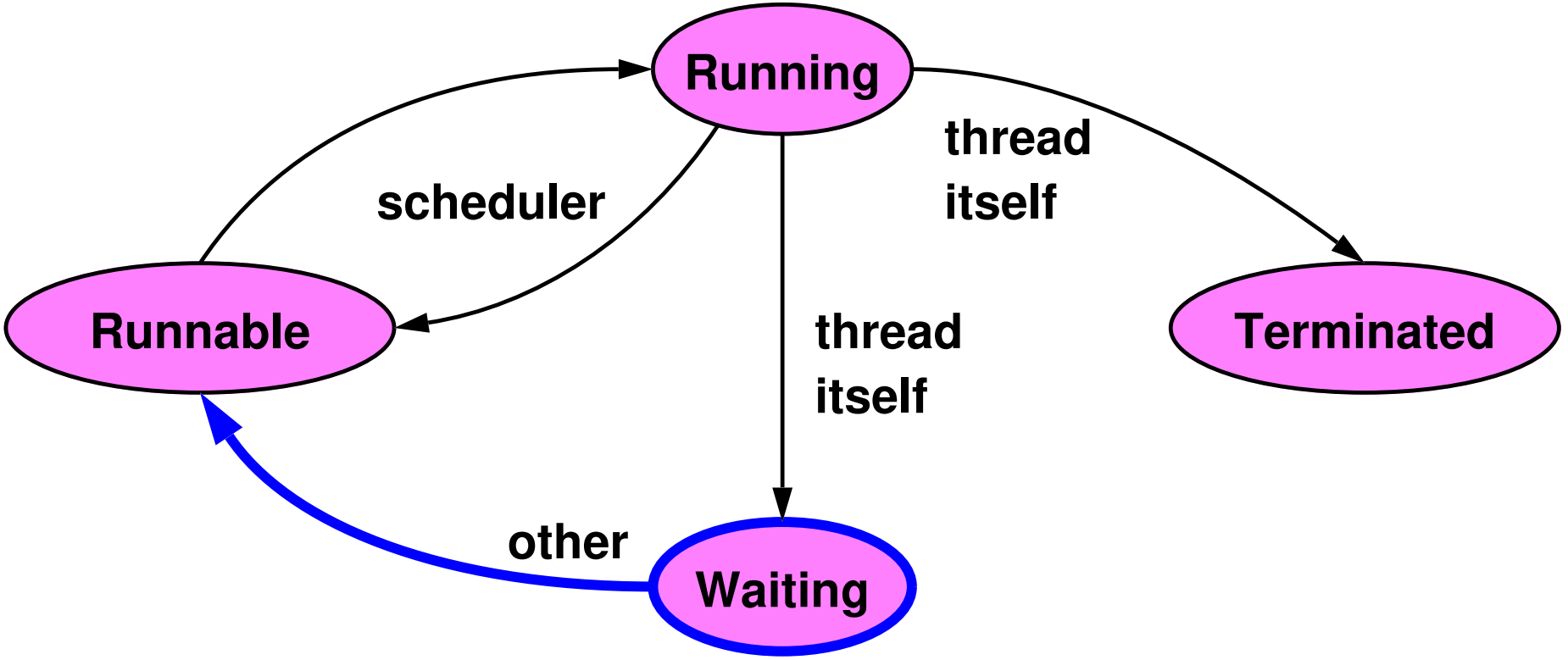
Thread Life Cycle



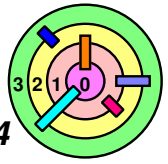
- the *scheduler* switches a thread's state from running to runnable when the thread used up its execution quantum
  - a thread can also *"yield"* the CPU (see examples in `faber_thread_test()` in kernel 1)



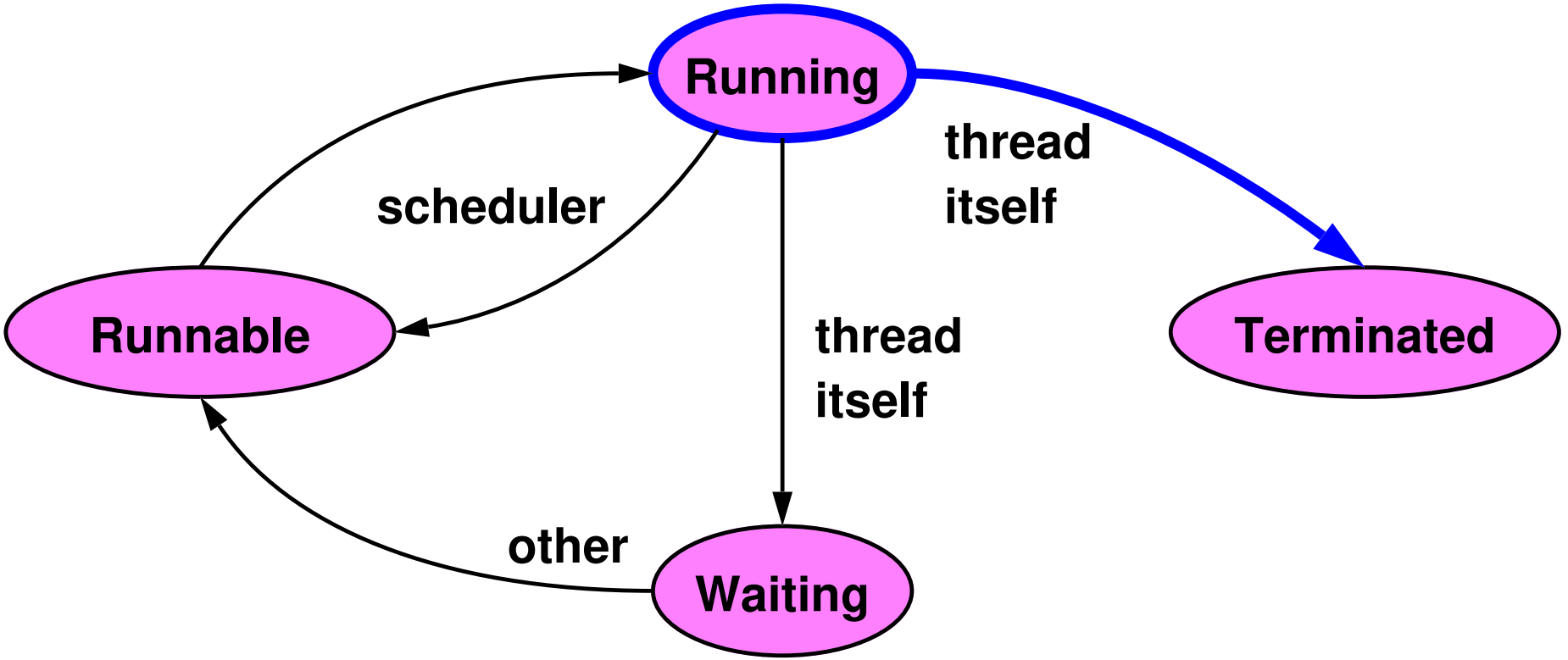
Thread Life Cycle



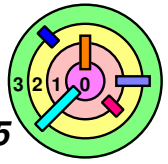
- a thread gets *unblocked* by the action of another thread or by an interrupt handler
  - the scheduler is *not involved* here



Thread Life Cycle

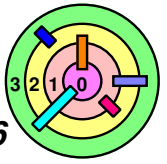


- in order for a thread to enter the terminated state, it has to be in the running state just before that
  - what if something like `pthread_cancel()` is invoked when the thread is not in the running state?



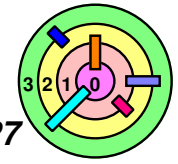
Thread Life Cycle

- 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?



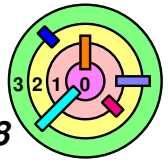
Thread Life Cycle

- 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