

# Announcements

- Program #0
  - Due Friday
- Reading
  - Threads - Chapter 4 (ch 5, 6<sup>th</sup> Ed)

## Process Control Block

- Stores all of the information about a process
- PCB contains
  - process state: new, ready, etc.
  - processor registers
  - Memory Management Information
    - page tables, and limit registers for segments
  - CPU scheduling information
    - process priority
    - pointers to process queues
  - Accounting information
    - time used (and limits)
    - files used
    - program owner
  - I/O status information
    - list of open files
    - pending I/O operations

# Storing PCBs

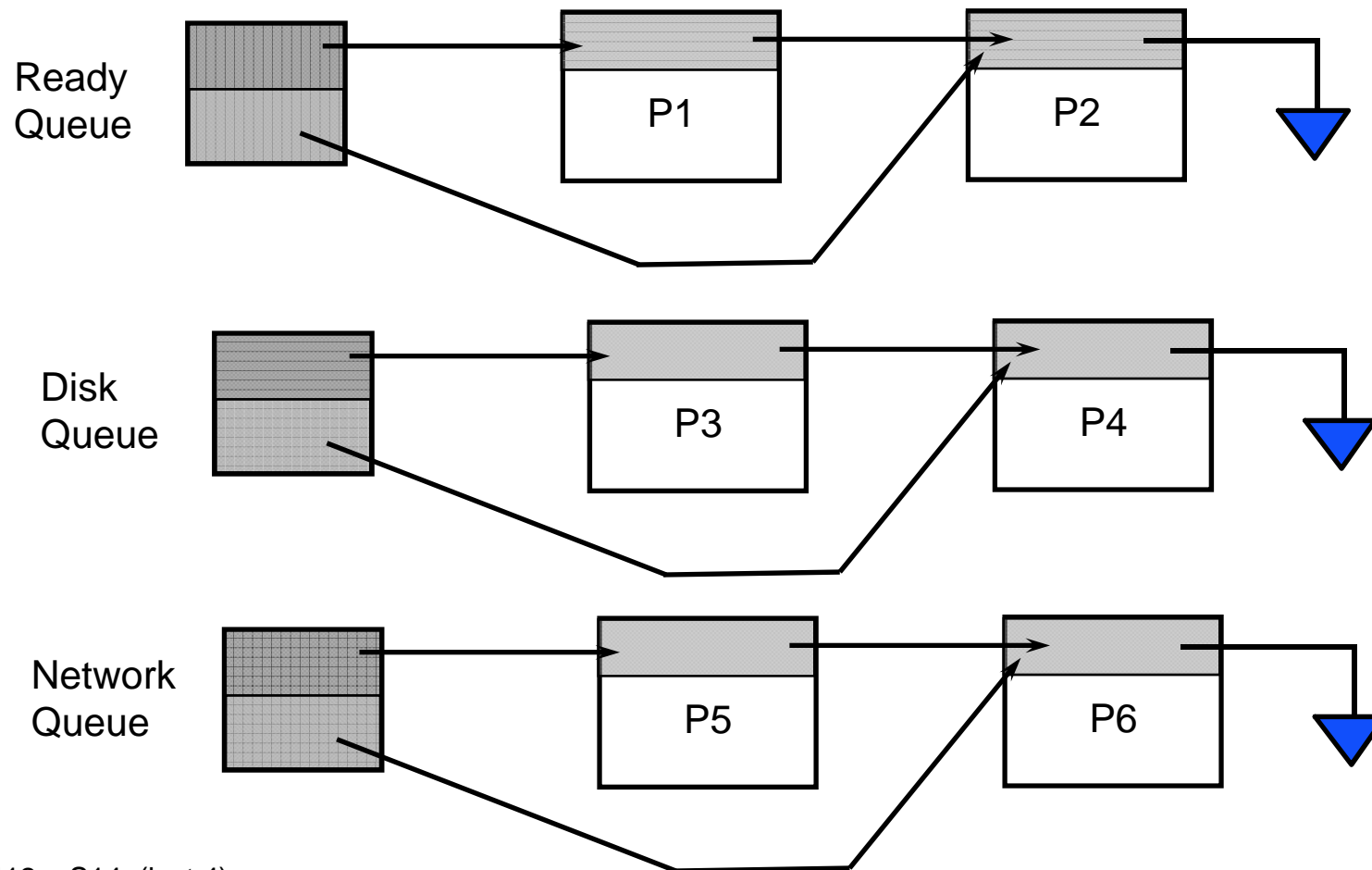
- Need to keep track of the different processes in the system
- Collection of PCBs is called a process table
- How to store the process table?
- First Option:

P1	P2	P2	P3	P4	P5
Ready	Waiting	New	Term	Waiting	Ready

- Problems with Option 1:
  - hard to find processes
  - how to fairly select a process

# Queues of Processes

- Store processes in queues based on state



# forking a new process

- create a PCB for the new process
  - copy most entries from the parent
  - clear accounting fields
  - buffered pending I/O
  - allocate a pid (process id for the new process)
- allocate memory for it
  - could require copying all of the parents segments
  - however, text segment usually doesn't change so that could be shared
  - might be able to use memory mapping hardware to help
    - will talk more about this in the memory management part of the class
- add it to the ready queue

# Variations on Creating a Process

- **Fork()** [ often used with exec too]
  - Create a new process with new address space
  - Parent address space copied into child
  - Child resumes at return of fork
- **Spawn(program)**
  - Create a new process with a new address space
  - Child starting running the passed program
  - Parent returns from spawn and continues executionn
- **Clone(func, stack)**
  - Creates a new process that **shares** parents address space
  - Child starts running func using the passed stack for locals
  - Parent returns from clone and continues execution

# Process Termination

- **Process can terminate self**
  - via the exit system call
- **One process can terminate another process**
  - use the kill system call
  - can any process kill any other process?
    - No, that would be bad.
    - Normally an ancestor can terminate a descendant
- **OS kernel can terminate a process**
  - exceeds resource limits
  - tries to perform an illegal operation
- **What if a parent terminates before the child**
  - called an orphan process
  - in UNIX becomes child of the root process
  - in VMS - causes all descendants to be killed

# Termination (cont.) - UNIX example

- Kernel

- frees memory used by the process
- moved process control block to the terminated queue

- Terminated process

- signals parent of its death (SIGCHLD)
- is called a zombie in UNIX
- remains around waiting to be reclaimed

- parent process

- wait system call retrieves info about the dead process
  - exit status
  - accounting information
- signal handler is generally called the reaper
  - since its job is to collect the dead processes