Housekeeping (Lecture 11 - 10/2/2013)

- Warmup #2 due at 11:45pm this Friday, 10/4/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - it's a good idea to run your code against the grading guidelines
- After submission, make sure you *Verify Your Submission*
- Have you installed *Ubuntu 11.10* on your laptop/desktop?
- Do you have partners for kernel assignments?
  - work with your potential partners for warmup 2
    - again, work at high level and must *not* share code

Housekeeping (Lecture 11 - 10/2/2013)

- Please do not address your post to the class Google Group to me (or the TA or the course producer)
  - if you post to the class Google Group, I will assume that it's not addressed to us (even if you copy us in the e-mail)
- We will not respond to posts to the class Google Group about the kernle assignments immediately
  - we will wait for at least 2 hours
    - this gives your classmates an opportunity to earn extra credits

Kernel Programming Assignments

Bill Cheng

http://merlot.usc.edu/cs402-f13

Kernel Programming Assignments

- Tom Doeppner's *weenix* source and binary code
  - provided as `weenix-assignment-1.0.7.tar.gz`
  - incomplete
  - contains code like:

```
NOT_YET_IMPLEMENTED("PROCS: bootstrap");
```

assignment name → PROCS

function name → bootstrap

    - your job is to implement these functions by replacing these lines with your code
      - please replace them *in-place*
  - to look for such code:

```
grep PROCS: kernel/*.c
grep PROCS: kernel/*/*.c
grep PROCS: kernel/*/*/*.c
grep PROCS: kernel/*/*/*/*.c
```

Download and Setup

```
% gunzip -c weenix-assignment-1.0.7.tar.gz | \
        tar xvf -
% cd weenix-assignment-1.0.7/weenix
% make clean
% make
% ./weenix -n
```

- if all goes well, you should see tons of stuff fly by and the following at the bottom of the console:

```
Not yet implemented: PROCS: bootstrap, file
    main/kmain.c, line 127
panic in main/kmain.c:129 bootstrap():
    weenix returned to bootstrap()!!! BAD!!!

Kernel Halting.
```

- Make sure you have tried the above this weekend
  - any problem, let us know ***NOW***

Documentation

- The `weenix` documentation is in `doc/latex/documentation.pdf`
  - introduces `weenix` to you
  - detailed instructions on assignments
  - you must read it thoroughly
- We are doing three of the assignments
  - *Processes and Threads (PROCS)*
  - *Virtual File System (VFS)*
  - *Virtual Memory (VM)*
- We are *not* doing two of the assignments
  - Drivers (DRIVERS)
  - System V File System (S5FS)
  - these are done for you and they are compiled and provided as libraries
    - `kernel/libdrivers.a` and `kernel/libs5fs.a`

Compilation and Configuration

- `Config.mk` controls what gets compiles and configured into the kernel
  - for PROCS, use the original `Config.mk`
    - set `DRIVERS` to 1 to complete this assignment
  - for VFS, set `DRIVERS` and `VFS` to 1
  - for VM, set `DRIVERS`, `VFS`, `S5FS`, and `VM` to 1
    - set `VM` to 0 at the beginning to get `kernel/mm/pframe.c` working
    - then set `VM` to 1 to work on the rest of the assignment
    - set `DYNAMIC` to 1 in the end if everything is working

- Modify `Config.mk` first, then do:

```
% make clean
% make
% ./weenix -n
```

- Use the debugger *right away*!

```
% ./weenix -n -d gdb
```

Debugging

- Use the debugger *right away*!

```
% ./weenix -n -d gdb
```

  - unfortunately, when you do the above, you won't see kernel debugging messages

```
% ./weenix -n -d gdb
```

- To see kernel debugging messages AND debug the kernel, do:
  - set `GDB_WAIT` to 1 in `Config.mk` then recompile kernel
  - run `weenix` under gdb with:

```
% ./weenix -n -d gdb -w
```

  - unfortunately, if you want to run *without gdb*, `weenix` will *hang*
    - you have to set `GDB_WAIT` back to 0 and recompile if you want to run `weenix` without gdb
    - you should set `GDB_WAIT` back to 0 when you submit your assignment for grading

Submissions

- Electronic submissions only with `bsubmit`
- Processes and Threads (PROCS)
  - need to fill out `procs-README.txt`, it's your assignment documentation
    - this is where you should include
      1) how to split the points (in terms of percentages and must sum to 100%)
      2) brief justification about the split

```
% make procs-submit
tar cvzf procs-submit.tar.gz \
        Config.mk \
        procs-README.txt \
        ...
```

    - your must add the required information about what you have implement in `procs-README.txt` as well
  - `scp` or `sftp` `procs-submit.tar.gz` to `nunki/aludra` and run `bsubmit` there

Submissions

- Virtual File System (VFS)
  - need to fill out `vfs-README.txt`

```
% make vfs-submit
...
```

- Virtual Memory (VM)
  - need to fill out `vm-README.txt`

```
% make vm-submit
...
```

- Submit source code only
  - we will deduct 2 points if you submit binary files
  - we will deduct 2 points if you submit unmodified files
  - we will deduct 2 points if you do not keep the same directory structure

Verify Your Submission

- Assume that in your home directory, you have
  - a *pristine* `weenix-assignment-1.0.7.tar.gz`
  - your submission file, e.g., `procs-submit.tar.gz`
- Do the following to verify your submission

```
% rm -rf /tmp/xyzzy
% mkdir /tmp/xyzzy
% cd /tmp/xyzzy
% gunzip -c ~/weenix-assignment-1.0.7.tar.gz | \
          tar xvf -
% cd weenix-assignment-1.0.7/weenix
% gunzip -c ~/procs-submit.tar.gz | tar xvf -
% make clean
% make
% ./weenix -n
```

Grading Guidelines

- For *every function* you need to implement, you need to think about its pre-conditions and post-conditions
  - you need to add `dbg()` and `KASSERT()` statements to show:
    1) you understand what a function is suppose to do
    2) what the important pre-conditions and post-conditions are
  - see grading guidelines for required `KASSERT()` statements
    - right after `KASSERT()`, prefix string in `dbg()` with `"(GRADING# X.Y)"`, where X and Y are from grading guidelines and # is the kernel assignment number
- You need to show us that you have thoroughly verified that your code works
  - turn in additional code or procedure and document how to enable/use these tests
  - it's okay if this is similar to our test code
- Run your code with additional test code from us
  - we will make suggestions in the grading guidelines

Backing Up Your Work & Collaboration

- You *have to* have a plan to *backup your code* and backup routinely
  - if you lose your work, there's nothing we can do
- Ubuntu 11.10 comes with "Ubuntu One"
  - free (up to 5GB) personal cloud
  - set one up for your group
    - all group members can sync with Ubuntu One
    - create a directory called `cs402backup` under `~/Ubuntu One`
- One simple way to backup your work
  - at the end of each day, do (for example, if today is 10/01/13):

```
% make procs-submit
% mv procs-submit.tar.gz procs-100213.tar.gz
% cp procs-100213.tar.gz ~/Ubuntu\ One/cs402backup
```

  - make sure only one person does this, or you will wipe out your partner's backup
  - can do backups more often if you'd like
    - `procs-100213-[time].tar.gz`

Backing Up Your Work & Collaboration

- You should use `git` to collaborate among project partners
  - read "Pro Git" (a free online book, one of our textbooks)
- But you need a shared repository in the cloud
  - there are free `git` repositories on the web
    - unfortunately, most of the free ones are required to be visible by the world - you must ***not*** use these
    - if you can find one that can be setup such that only you and your partners can see it, then you can use it
  - may be you can use ***Ubuntu One*** as your repository
    - the course producers will share their experience
- If you have two people working on the same file and then update the repository one after another
  - `git` will attempt to merge the changes, but it may not be what you want
  - may be it's best to ***coordinate*** and not have two pepole modifying the same file

Early and Late Policies

- Similar to warmup projects

Extra Credit

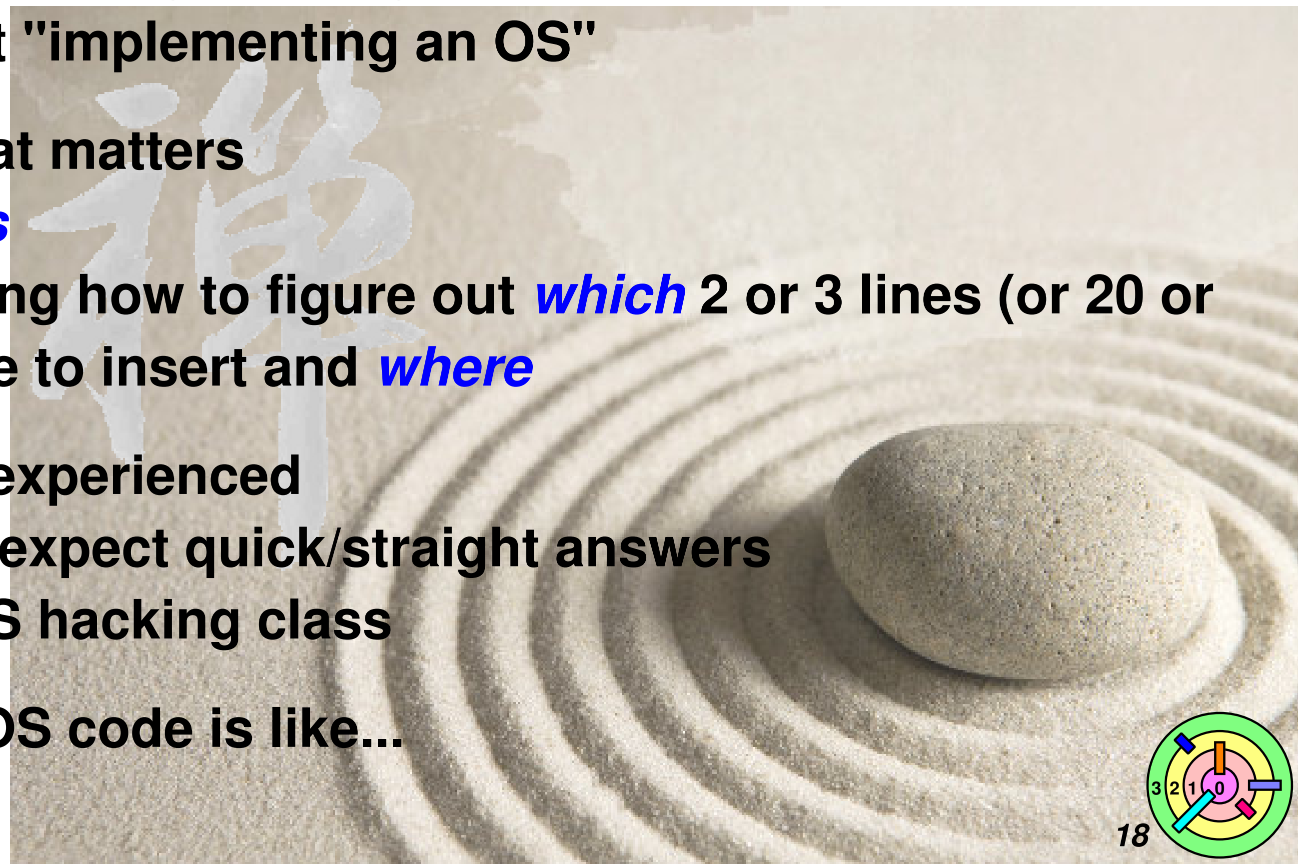
- You can get extra credit for posting good, useful, and insightful answers to the class Google Group in response to questions posted by other students regarding ***kernel*** programming assignments
  - the maximum number of extra credit points you can get is ***10*** points for each of the kernel assigments (on a 100-point scale)

How Do You Start?

- Definitely start with the documentation
- Read code, read lots and lots of code
  - try things out and see what happens (debugging statements)
  - you need to *absorb* other people's code, make sense of it
  - that's what OS hacking (in the good sense) is all about
    - it's *not* about "implementing an OS"
- It's the *process* that matters
  - *not* the *answers*
  - it's about learning how to figure out *which* 2 or 3 lines (or 20 or 30 lines) of code to insert and *where*
- So, it needs to be experienced
  - you should not expect quick/straight answers
  - this is not an OS hacking class
- Learning to write OS code is like...

How Do You Start?

- Definitely start with the documentation
- Read code, read lots and lots of code
  - try things out and see what happens (debugging statements)
  - you need to *absorb* other people's code, make sense of it
  - that's what OS hacking (in the good sense) is all about
    - it's *not* about "implementing an OS"
- It's the *process* that matters
  - *not* the *answers*
  - it's about learning how to figure out *which* 2 or 3 lines (or 20 or 30 lines) of code to insert and *where*
- So, it needs to be experienced
  - you should not expect quick/straight answers
  - this is not an OS hacking class
- Learning to write OS code is like...
  - *Zen*



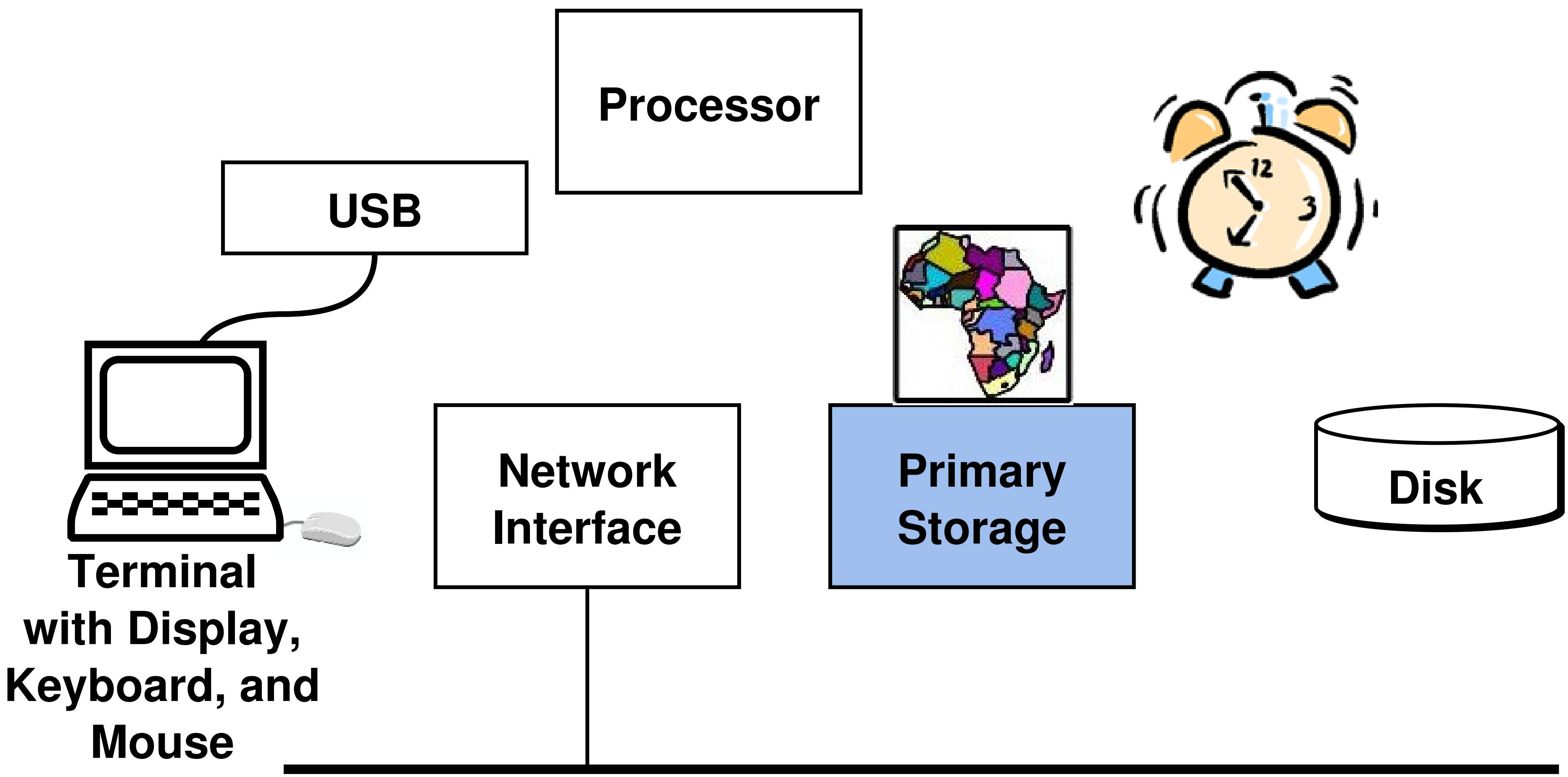
Getting Help

- If you have questions about the kernel assignments
  - read documentation, textbook, lecture slides, read more code
  - post your questions to the class Google Group
    - we will not immediately answer these questions to give your classmates an opportunity to earn extra credit points
      - we will wait 2 hours
    - if you send us questions about the assignments directly, we may simply forward your post to the class Google Group

OS Design Issues

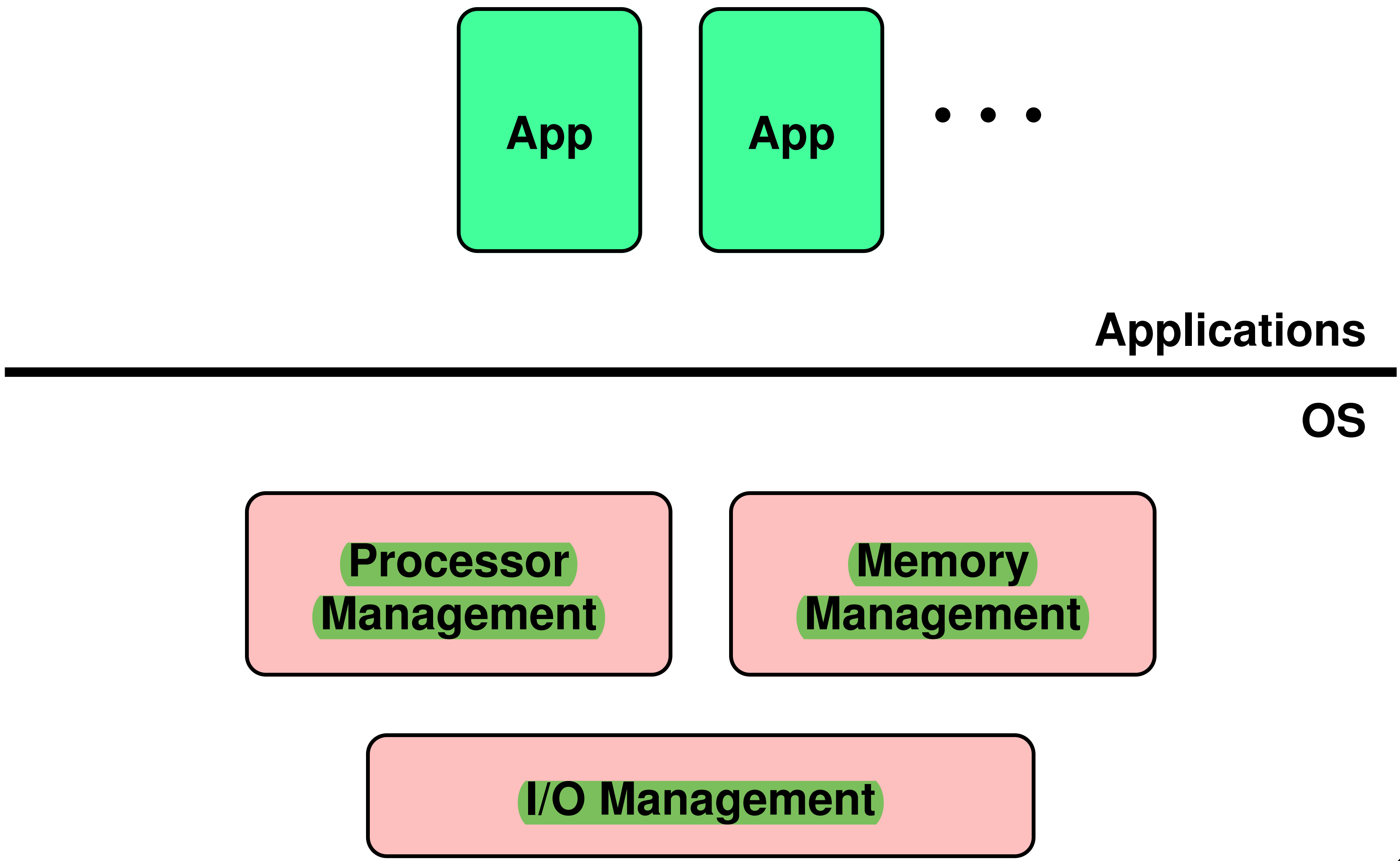
- **Performance**
  - efficiency of application
- **Modularity**
  - tradeoffs between modularity and performance
- **Device independence**
  - for new devices, don’t need to write a new OS
- **Security/Isolation**
  - isolate OS from application

Simple Configuration

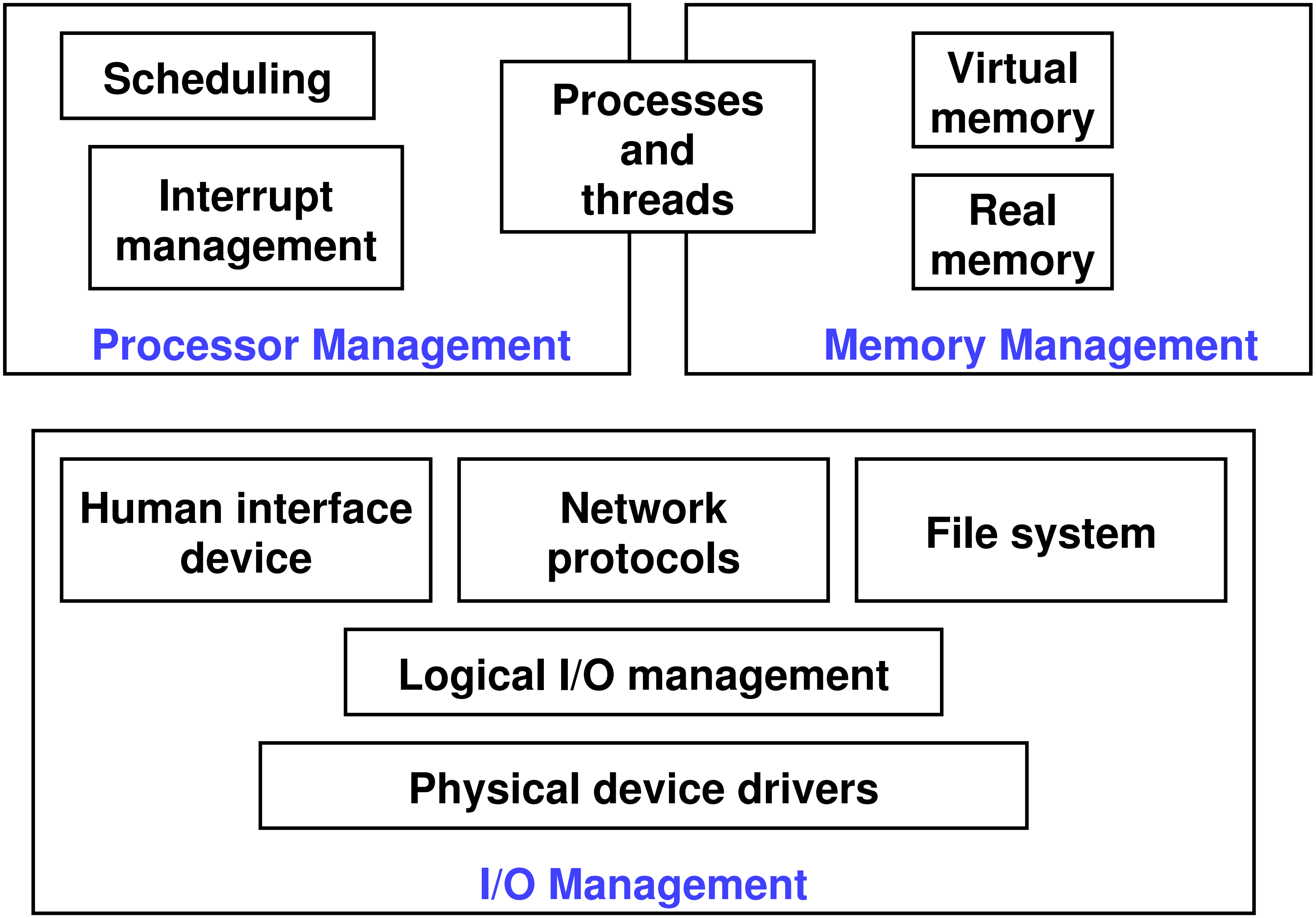


- Early 1980s OS, so we can focus on the basic OS issues
  - no support for bit-mapped displays and mice
  - generally less efficient design

OS Components



OS Components



OS Components

Scheduling

Interrupt management

Processes and threads

Processor Management

Virtual memory

Real memory

Memory Management

Human interface device

Network protocols

Logical I/O management

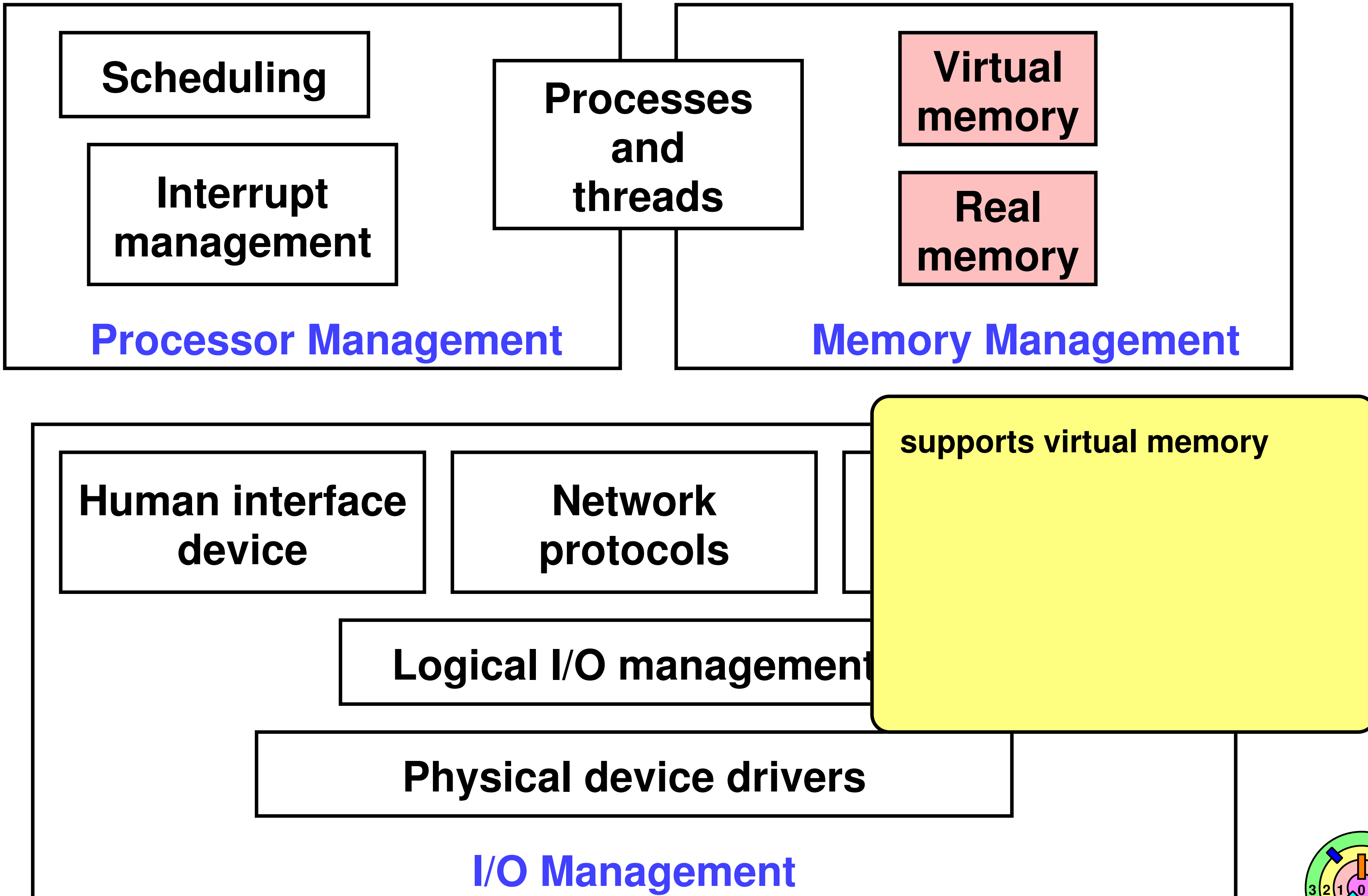
Physical device drivers

I/O Management

supports multithreaded processes

- each process has its own address space

OS Components



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

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

...ent

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

Scheduling

Interrupt management

Processes and threads

Processor Management

communication over Ethernet using TCP/IP

Me

Human interface device

Network protocols

File system

Logical I/O management

Physical device drivers

I/O Management

Applications

- How does an *application* view these OS components?
- 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 components... Need to look at:
  - 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*** **(using *memory maps*)**

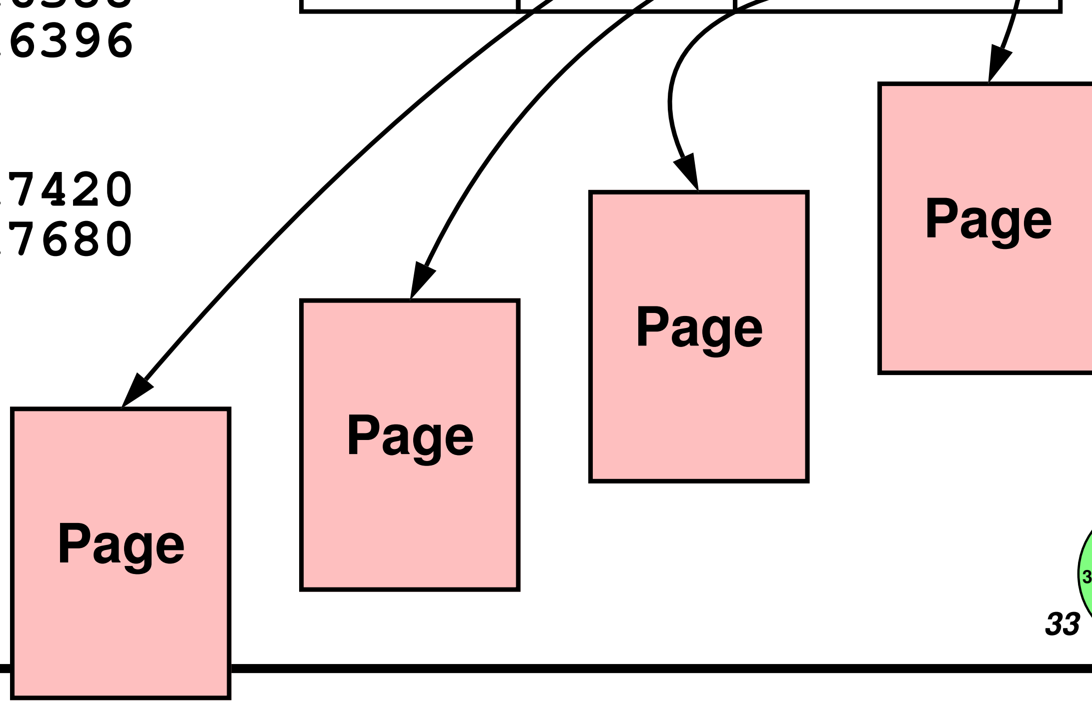
```
Text
   main           4096
   subr           4132
   printf         4156
   write         16156
   startup       16172

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	•



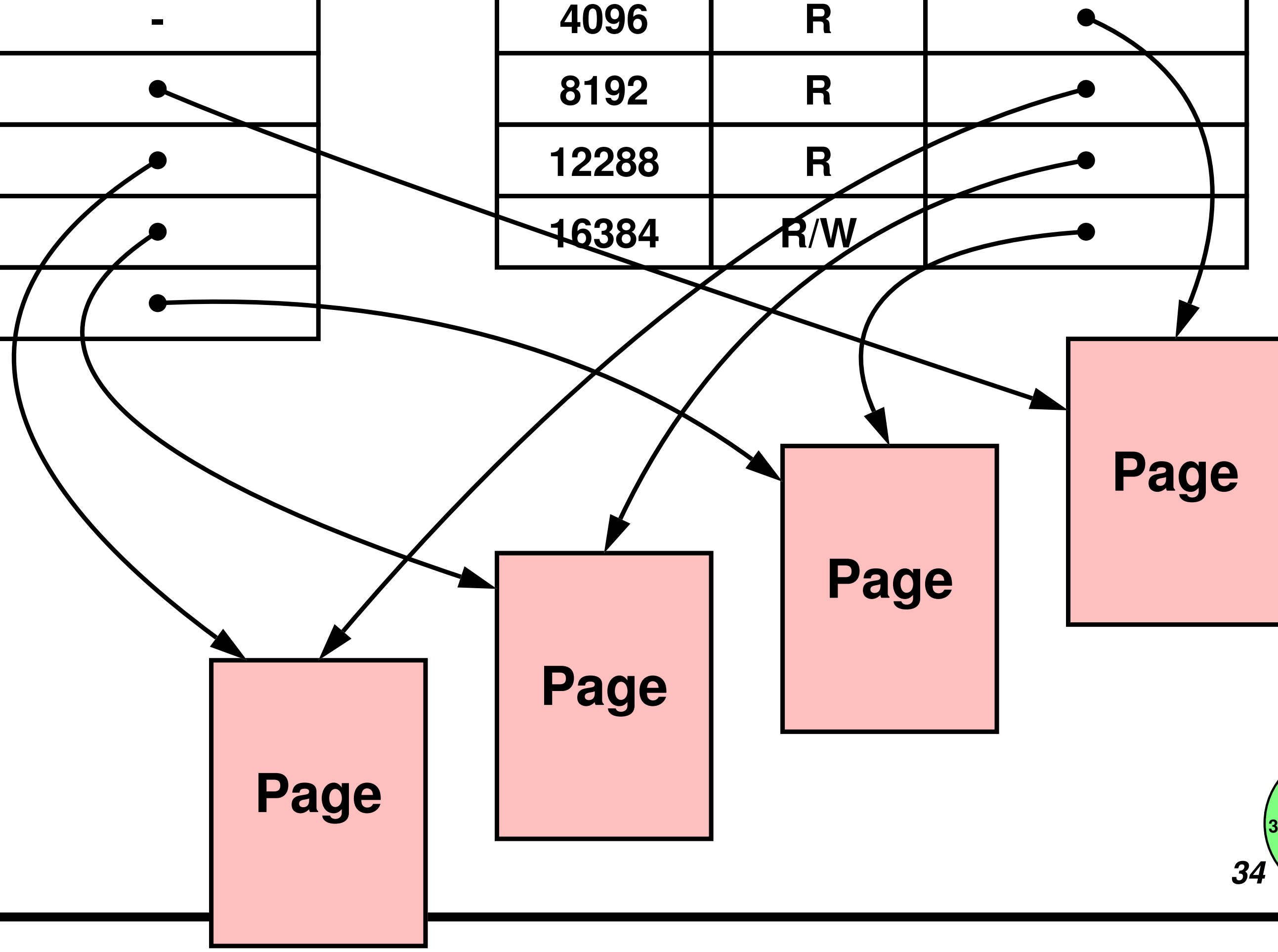
Processes Can Share Memory Pages

Process 2 Page Table

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

Process 1 Page Table

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



Processes And File Systems

- **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* 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

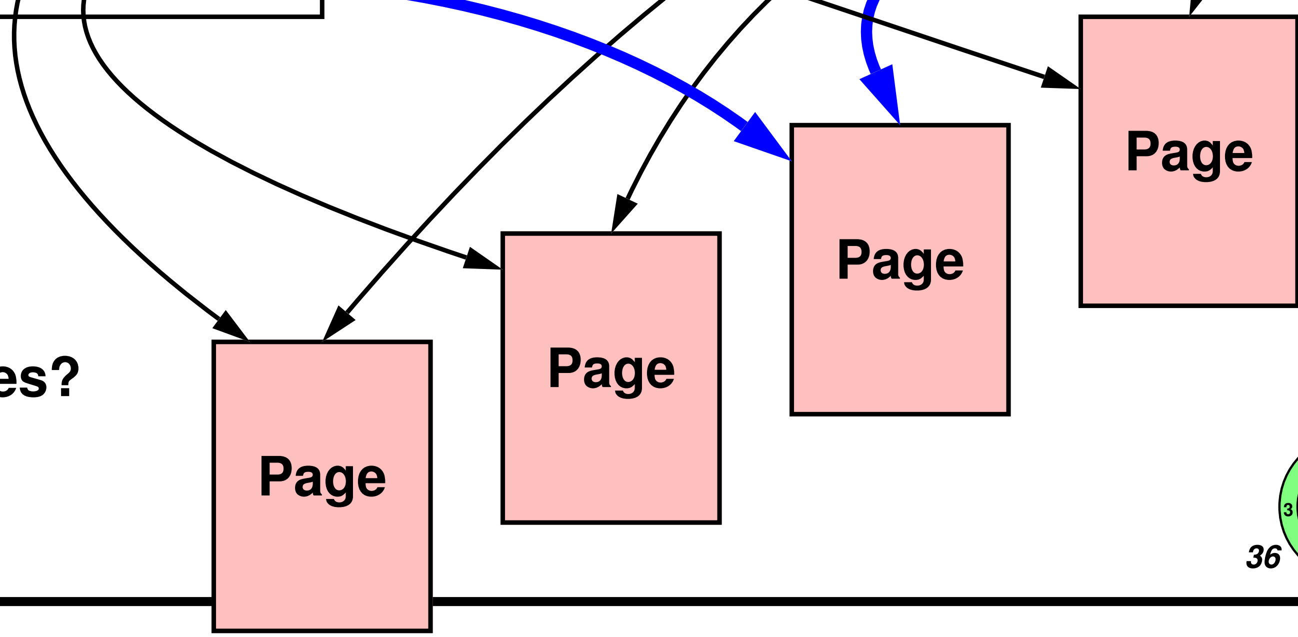
Processes Can Share Memory Pages

Process 1 Page Table

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

Process 2 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 *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

Processes And File Systems

- *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 BSS and stack regions use a special form of private mapping
  - their pages are initialized, with zeros; *copy-on-write*
    - known as *anonymous objects* in `weenix`
- If a bunch of processes share a file
  - we can also map the file into the address space
  - 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

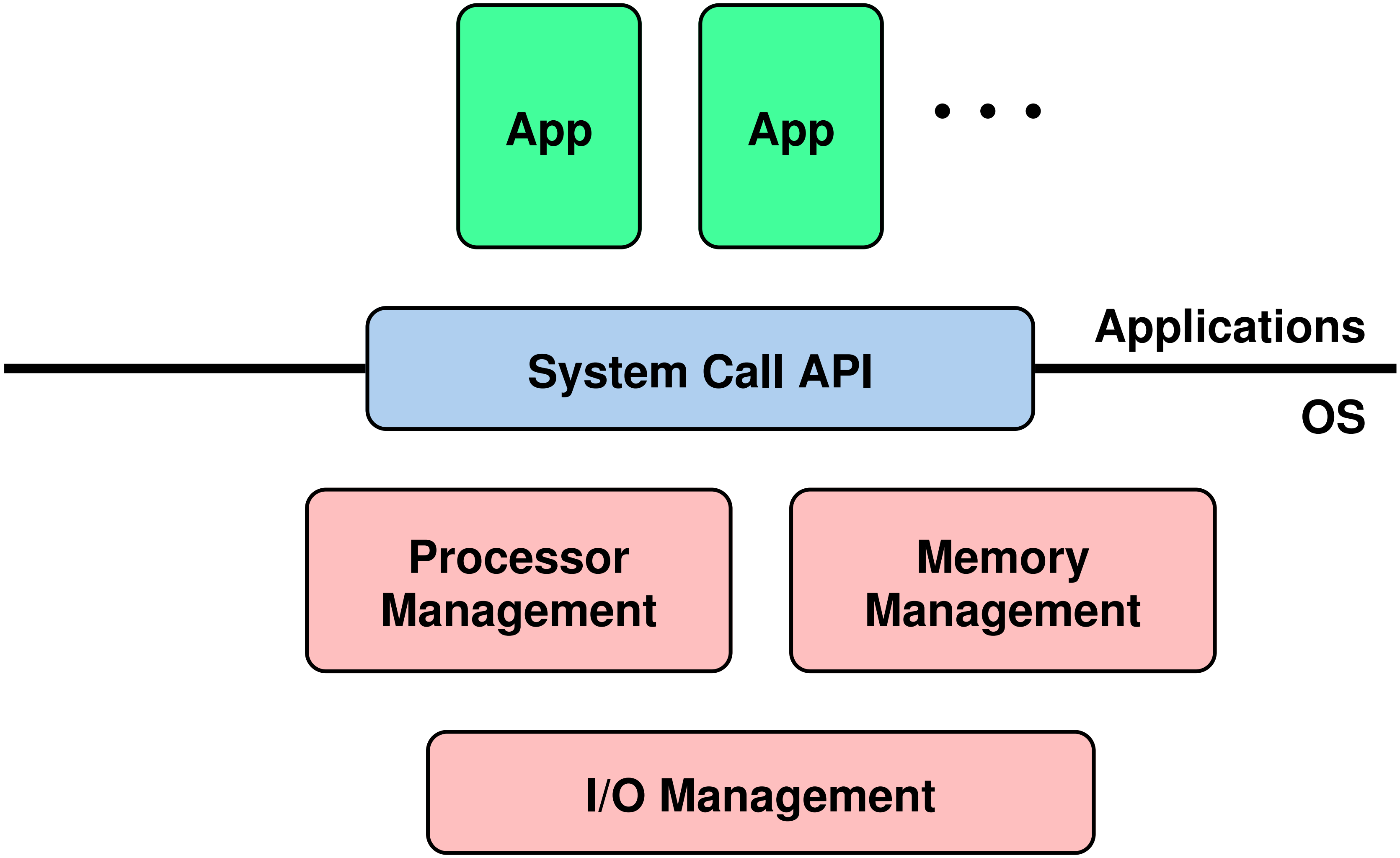
- Mapping files into address space is one way to perform I/O on files
  - block/page is the basic unit
  - this is referred to as *block I/O*
- Some devices cannot be mapped into the address space
  - e.g., receiving characters typed into the keyboard, sending a message via a network connection
  - need 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)

A Simple System: To Be Discussed

- What is the functionality of the components?
- What are the key data structures?
- 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

System Call API

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



4.1 A Simple System (Monolithic Kernel)

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

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

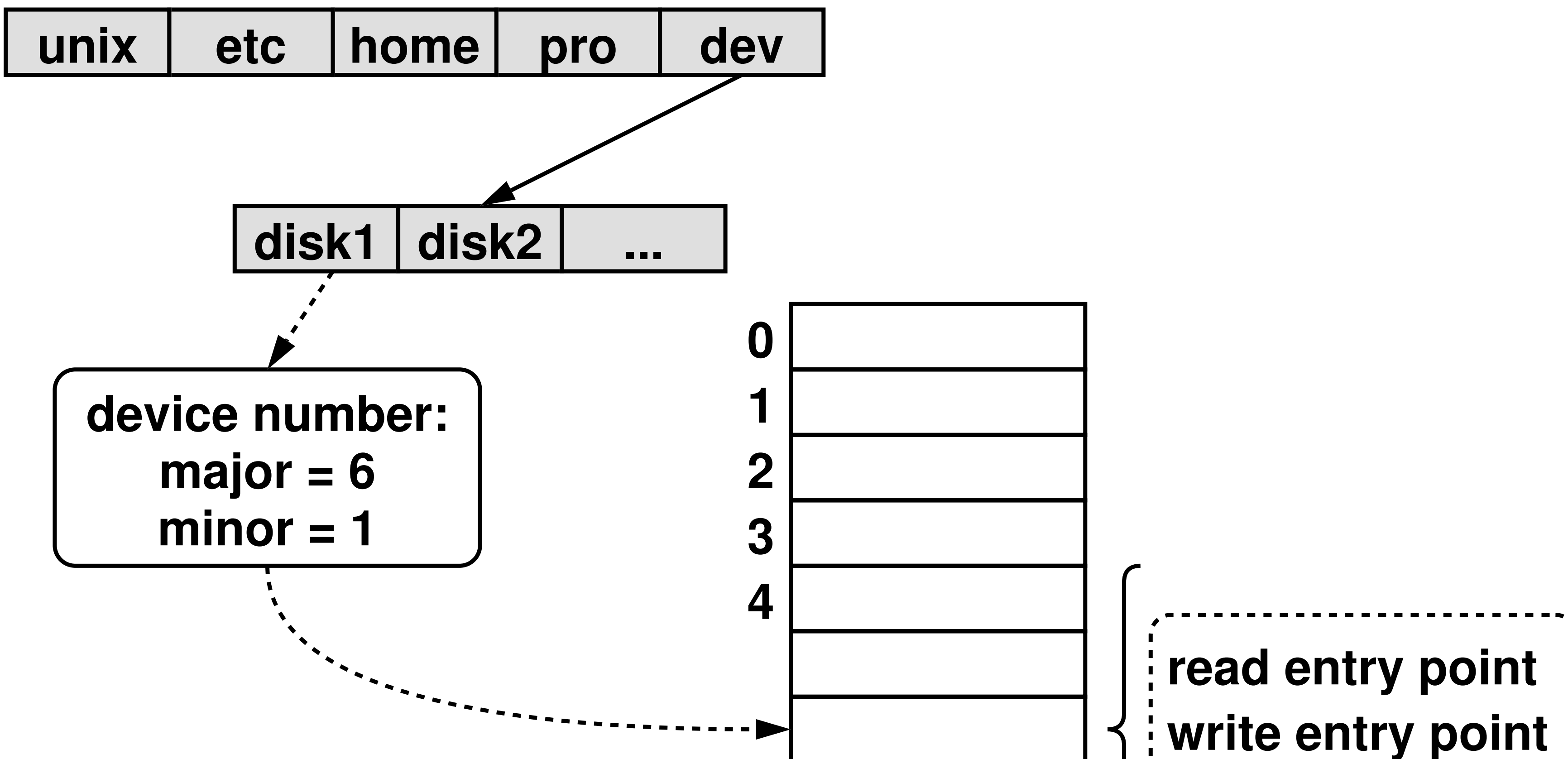
A Framework for Devices

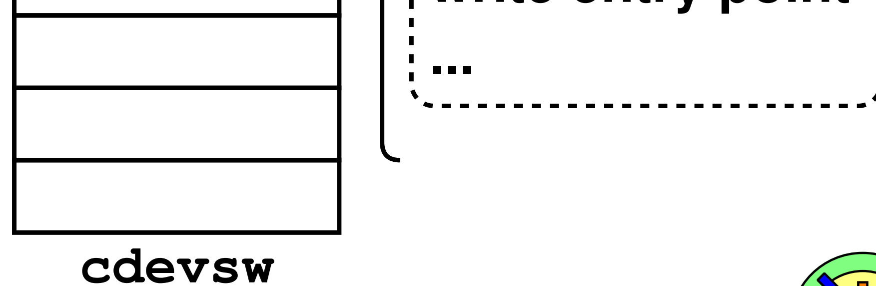
- Device driver:
  - every device is identified by a device "number", which is actually a pair of numbers
    - a *major device number* - identifies the device driver
    - a *minor device number* - device index for all devices managed by the same device driver

- 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)

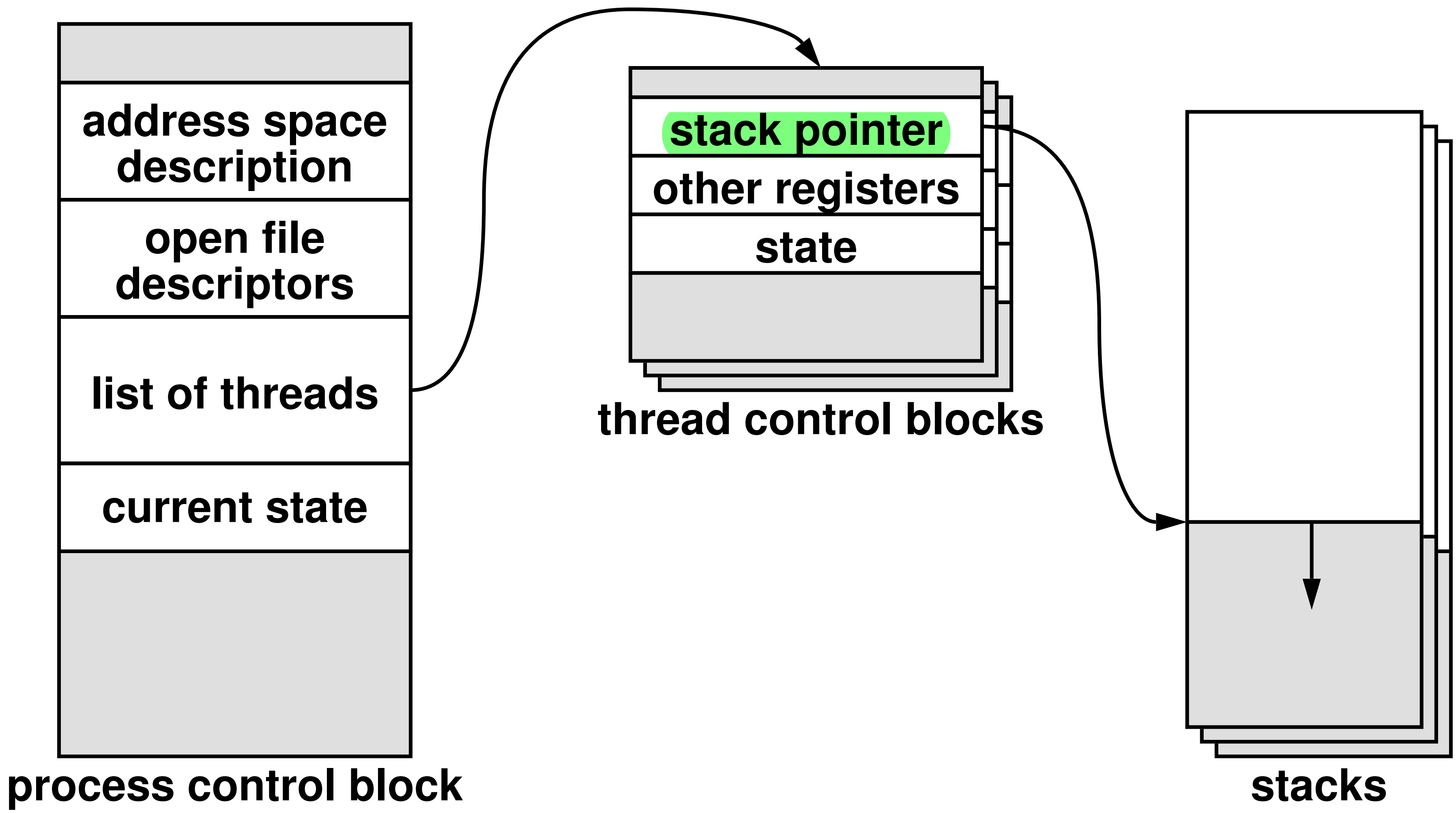
4.1 A Simple System (Monolithic Kernel)

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

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