CIS520 Operating Systems Course Intro

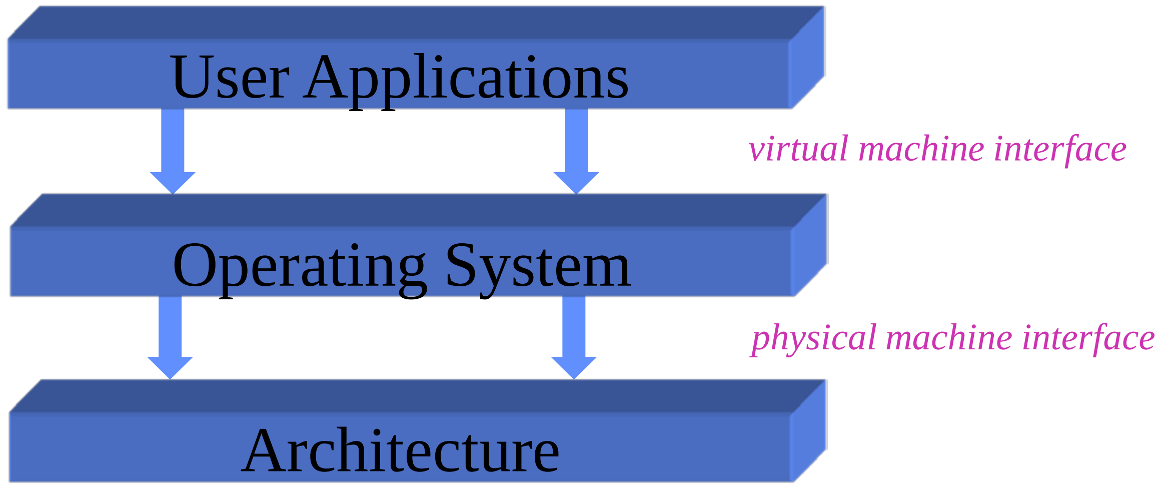
Dr. Daniel Andresen

Computing & Information Sciences

Kansas State University

Operating Systems: The Big Picture

The operating system (OS) is the interface between user applications and the hardware.



An OS implements a sort of *virtual machine* that is easier to program than the raw hardware.

[McKinley]

The OS and the Hardware

The OS is the “permanent” software with the power to:

- control/abstract/mediate access to the hardware

  CPUs and memory

  I/O devices

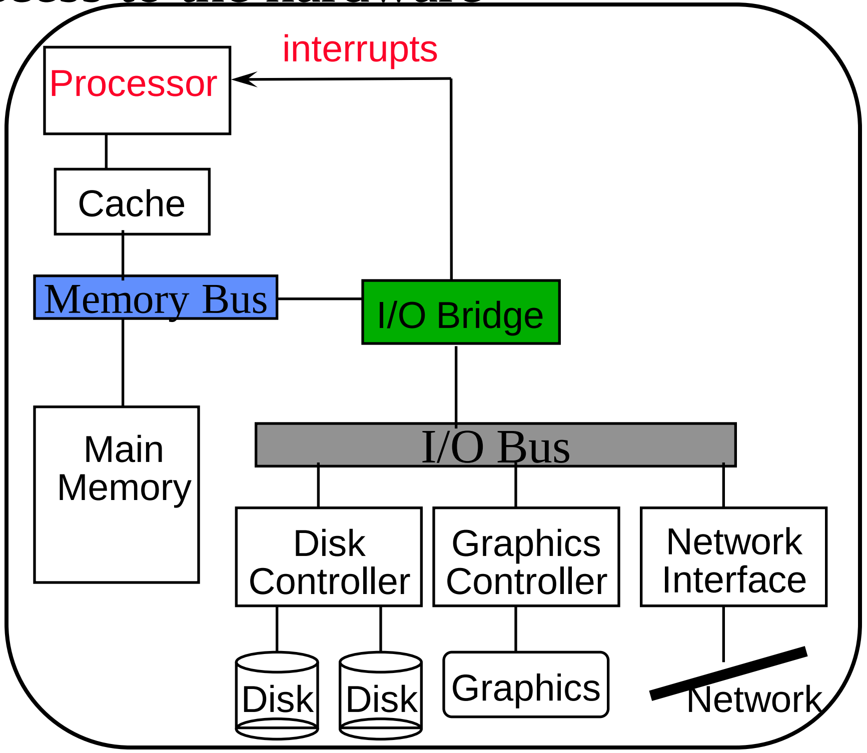
- so user code can be:

  simpler

  device-independent

  portable

  even “transportable”



The OS and User Applications

The OS defines a framework for users and their programs to coexist, cooperate, and work together safely, supporting:

- concurrent execution/interaction of multiple user programs
- shared implementations of commonly needed facilities

  "The *system* is all the code you didn't write."

- *mechanisms* to share and combine software components

  Extensibility: add new components on-the-fly as they are developed.

- *policies* for safe and fair sharing of resources

  physical resources (e.g., CPU time and storage space)

  logical resources (e.g., data files, programs, mailboxes)

Overview of OS Services

Storage: primitives for files, *virtual memory*, etc.

- control devices and provide for the “care and feeding” of the memory system hardware and peripherals

Protection and security

- set boundaries that limit damage from faults and errors
- establish user identities, priorities, and accountability
- access control for logical and physical resources

Execution: primitives to create/execute programs

- support an environment for developing and running applications

Communication: “glue” for programs to interact

The Four Faces of Your Operating System

- **service provider**

  The OS exports commonly needed facilities with standard interfaces, so that programs can be simple and portable.

- **executive/bureaucrat/juggler**

  The OS controls access to hardware, and allocates physical resources (memory, disk, CPU time) for the greatest good.

- **caretaker**

  The OS monitors the hardware and intervenes to resolve exceptional conditions that interrupt smooth operation.

- **cop/security guard**

  The OS mediates access to resources and grants/denies requests.

- Or, at the core, a **virtual machine** and **resource manager**

Other Useful Metaphors

1. ***Phone systems***

   Defines an infrastructure for users to call each other and talk.

   - doesn’t dictate who you call; doesn’t dictate what you say
   - supports services not imagined by creators, e.g., 900 numbers

2. ***Government***

   Sets rules and balances demands from a diverse community.

   Users/subjects want high levels of service with low taxes.

3. ***Transportation***

   Wide range of choices for a wide range of user goals and needs.

   - **race car**: simple interface to powerful technology…goes fast…crashes hard.
   - **cadillac**: makes choices automatically…comfortable…no fun to drive.
   - **SUV**: runs on any terrain…rolls over bumps…burns gas…but gas is cheap.

Studying Operating Systems

This course deals with "classical" operating systems issues:

- the services and facilities that operating systems provide;
- OS implementation on modern hardware;

  (and architectural support for modern operating systems)
- how hardware and software evolve together;
- the techniques used to implement software systems that are:

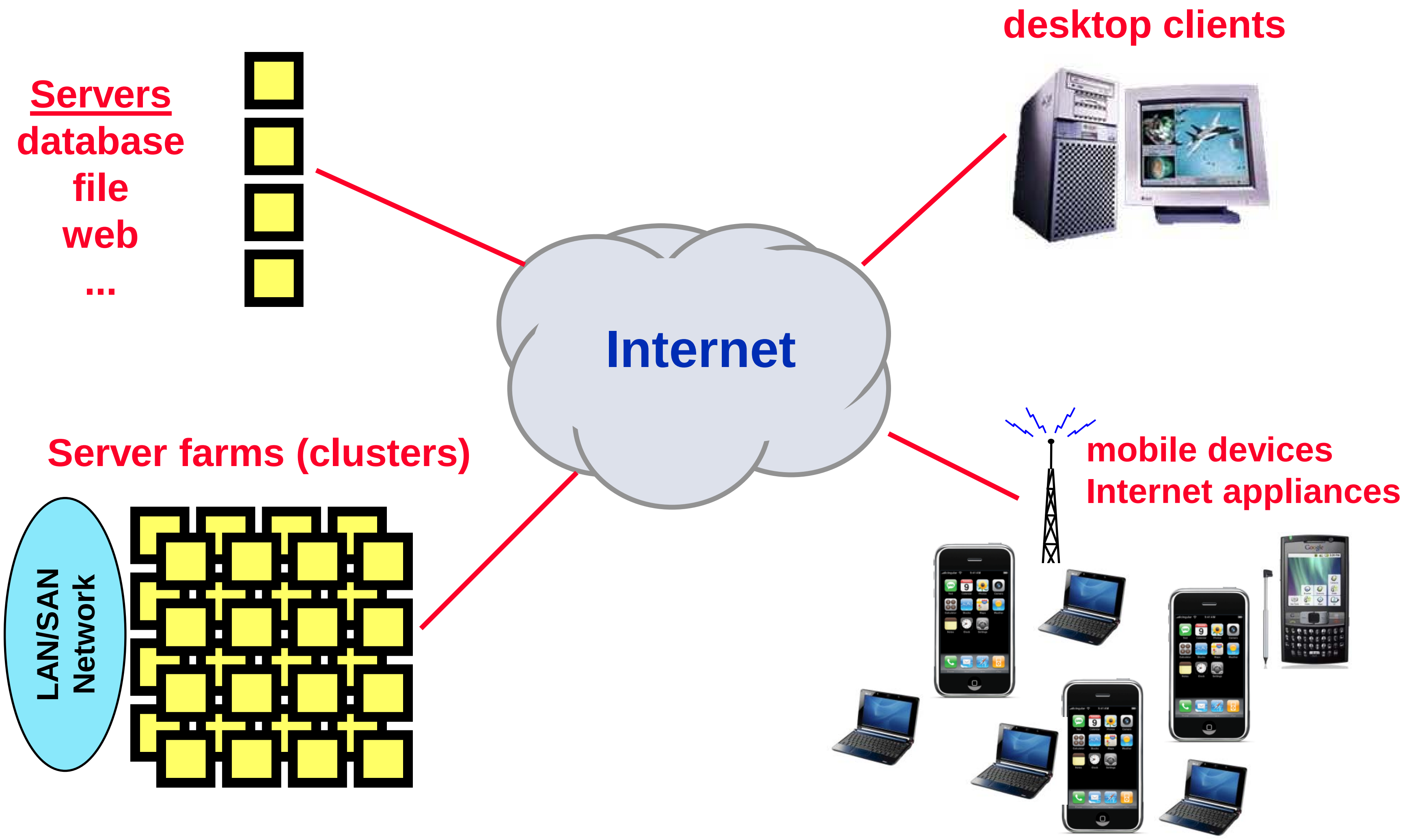
  large and complex,

  long-lived and evolving,

  concurrent,

  performance-critical.

The World Today



The Big Questions

1. How to divide function/state/trust across components?

    reason about flow of data and computation through the system

2. What abstractions/interfaces are sufficiently:

    powerful to meet a wide range of needs?

    efficient to implement and simple to use?

    versatile to enable construction of large/complex systems?

3. How can we build:

    reliable systems from unreliable components?

    trusted systems from untrusted components?

    unified systems from diverse components?

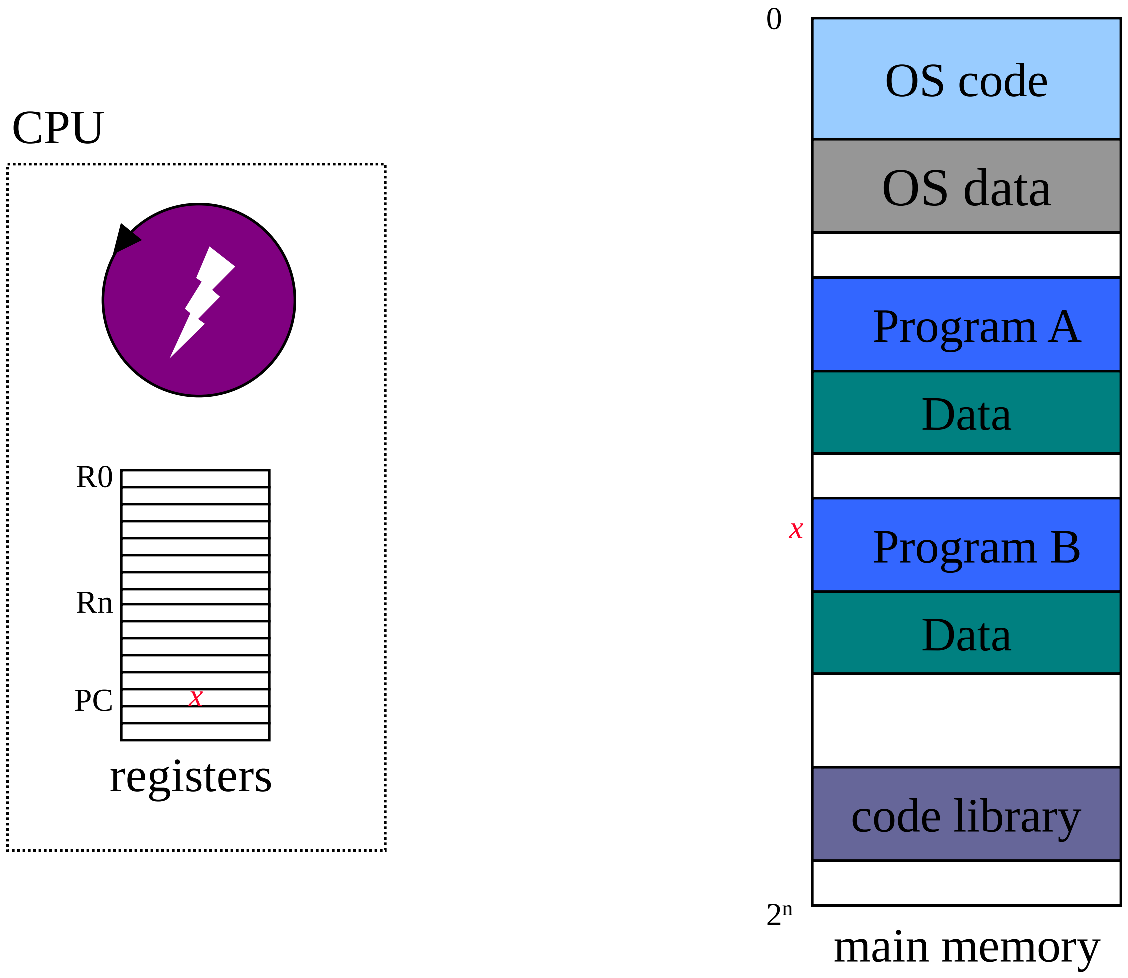
    coherent systems from distributed components?

Classical View: The Questions

The basic issues/questions in this course are *how to*:

- allocate memory and storage to multiple programs?
- share the CPU among concurrently executing programs?
- *suspend* and *resume* programs?
- share data safely among concurrent activities?
- protect one executing program's storage from another?
- protect the code that implements the protection, and mediates access to resources?
- prevent rogue programs from taking over the machine?
- allow programs to interact safely?

Memory and the CPU



A First Look at Some Key Concepts

kernel

The software component that controls the hardware directly, and implements the core privileged OS functions.

Modern hardware has features that allow the OS kernel to protect itself from untrusted user code.

thread

An executing stream of instructions and its CPU register context.

virtual address space

An execution context for thread(s) that provides an independent name space for addressing some or all of physical memory.

process

An execution of a program, consisting of a virtual address space, one or more threads, and some OS kernel state.

The Kernel

- The *kernel* program resides in a well-known executable file.

  The “machine” automatically loads the kernel into memory (*boots*) on power-on or reset.

- The kernel is (mostly) a library of service procedures shared by all user programs, **but** *the kernel is protected*:

  User code cannot access internal kernel data structures directly.

  User code can invoke the the kernel only at well-defined entry points (*system calls*).

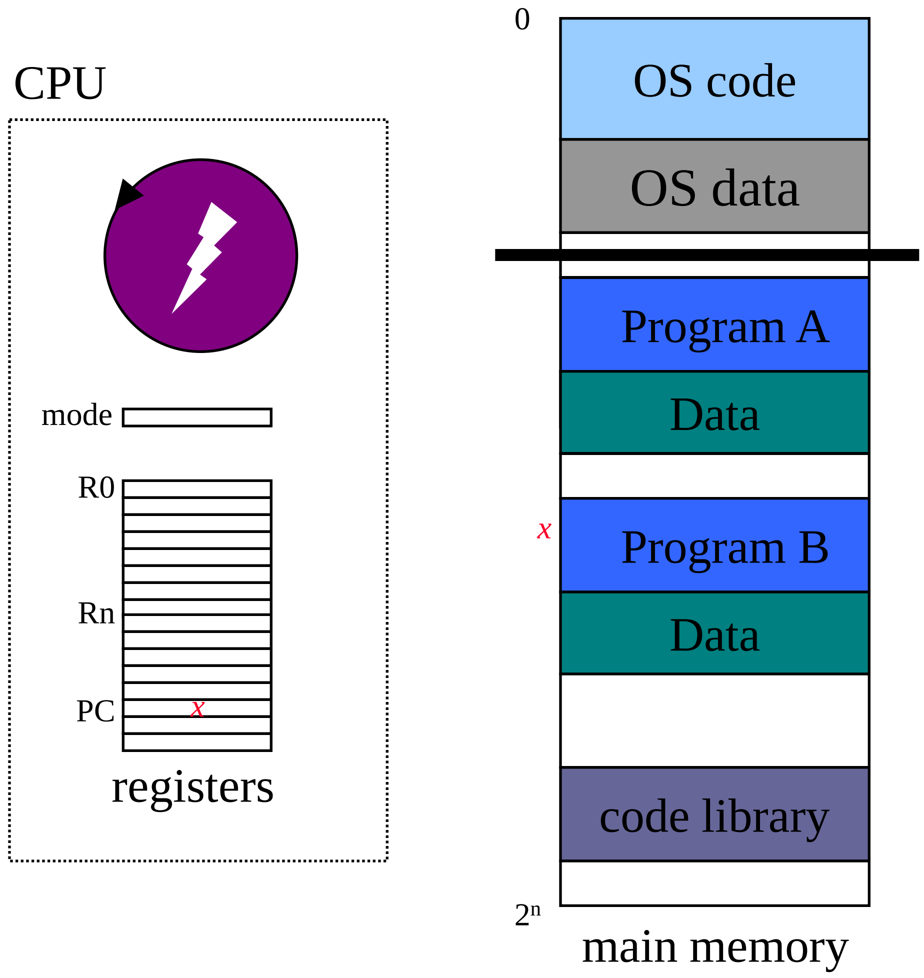
- *Kernel code is like user code, but the kernel is privileged*:

  Kernel has direct access to all hardware functions, and defines the machine entry points for interrupts and exceptions.

A Protected Kernel

Mode register bit indicates whether the CPU is running in a user program or in the protected kernel.

Some instructions or data accesses are only legal when the CPU is executing in kernel mode.



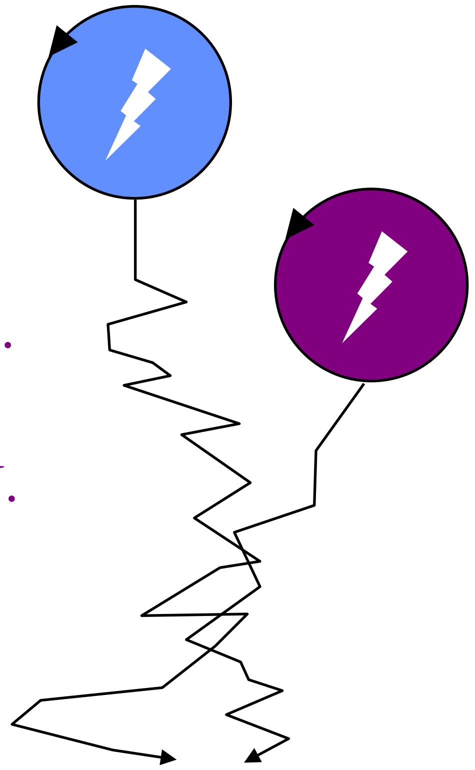
Threads

A *thread* is a schedulable stream of control.

- defined by CPU register values (PC, SP)
- *suspend*: save register values in memory
- *resume*: restore registers from memory

Multiple threads can execute independently:

- They can run in parallel on multiple CPUs...
  - ***physical concurrency***
- …or arbitrarily interleaved on a single CPU.
  - ***logical concurrency***
- Each thread must have its own stack.



Overview of Nachos Project 0-1

In the thread assignments, you build and test the kernel's *internal* primitives for processes and synchronization.

- Think of your program as a "real" kernel doing "real basic" things.

  boot and initialize

  run ***main()****, parse arguments, initialize machine state*

  run a few tests

  create multiple threads to execute some kernel code

  shut down

- *...runs native, directly on the host machine.*
- *Kernel only; no user programs (until Project 2)*