Module I

Course Overview
And
Introduction To Operating Systems

COURSE MOTIVATION AND SCOPE

Scope

This is a course about the design and structure of computer operating systems. It covers the concepts, principles, functionality, tradeoffs, and implementation of systems that support concurrent processing.

What We Will Cover

- Operating system fundamentals
- Functionality an operating system offers
- Major system components
- Interdependencies and system structure
- The key relationships between operating system abstractions and the underlying hardware (especially processes and interrupts)
- A few implementation details and examples

What You Will Learn

- Fundamental
  - Principles
  - Design options
  - Tradeoffs
- How to modify and test operating system code
- How to design and build an operating system

What We Will NOT Cover

- A comparison of large commercial and open source operating systems
- A description of features or instructions on how to use a particular commercial system
- A survey of research systems and alternative approaches that have been studied
- A set of techniques for building operating systems that run on unusual hardware

How Operating Systems Changed Programming

- Before operating systems
  - Only one application could run at any time
  - The application contained code to control specific I/O devices
  - The application had to overlap I/O and processing
- With an operating system in place
  - Multiple applications can run at the same time
  - An application is not built for specific I/O devices
  - A programmer does not need to overlap I/O and processing
  - An application is written without regard to other applications

Why Operating Systems Are Difficult To Build

- The gap between hardware and high-level services is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
  - An operating system must bridge the gap between low-level hardware and high-level abstractions
- Everything is now connected by computer networks
  - An operating system must offer communication facilities
  - Distributed mechanisms (e.g., access to remote files) are more difficult to create than local mechanisms

An Observation About Efficiency

- Our job in Computer Science is to build beautiful new abstractions that programmers can use
- It is easy to imagine magical new abstractions
- The hard part is that we must find abstractions that map onto the underlying hardware efficiently
- We hope that hardware engineers eventually build hardware for our abstractions (or at least build hardware that makes our abstractions more efficient)

The Once And Future Hot Topic

- In the 1970s and 1980s, operating systems was one of the hottest topics in CS
- By the mid-1990s, OS research had stagnated
- Now things have heated up again, and new operating systems are being designed for
  - Smart phones
  - Multicore systems
  - Data centers
  - Large and small embedded devices (the Internet of Things)

The Xinu Operating System

Motivation For Studying A Real Operating System

- Provides examples of the principles
- Makes everything clear and concrete
- Shows how abstractions map to current hardware
- Gives students a chance to experiment and gain first-hand experience

Can We Study Commercial Or Open-Source Systems?

- Windows
  - Millions of lines of code
  - Proprietary
- Linux
  - Millions of lines of code
  - Lack of consistency across modules
  - Duplication of functionality with slight variants

An Alternative Operating System: Xinu

- Small — can be read and understood in a semester
- Complete — includes all the major components
- Elegant — provides an excellent example of clean design
- Powerful — has dynamic process creation, dynamic memory management, flexible I/O, and basic Internet protocols
- Practical — has been used in real products

REQUIRED BACKGROUND AND PREREQUISITES

Background Needed

- A few concepts from earlier courses
  - Integer arithmetic and bit-wise operators *and*, *or*, and *not*
  - I/O: you should know the difference between standard library functions (e.g., *fopen*, *putc*, *getc*, *fread*, *fwrite*) and system calls (e.g., *open*, *close*, *read*, *write*)
  - File systems and hierarchical directories
  - Symbolic and hard links
  - File modes and protection
  - Key concepts from computer architecture, such as the purpose of a bus

Background Needed
(continued)

  - Basic data structures (e.g., linked lists)
  - Binary and hex representation
  - The run-time stack concept
  - Local and global variable allocation
  - Function calls, arguments, and calling conventions
- Concurrent programming experience: you should have written a program that uses *fork* or *threads*

Background Needed
(continued)

- An understanding of runtime storage
  - Segments (text, data, bss, and stack)
  - Basic heap storage management (e.g., malloc and free)
- C programming
  - At least one nontrivial program
  - Comfortable with low-level constructs (e.g., bit manipulation, pointers, and pointer arithmetic)

Background Needed
(continued)

- Working knowledge of basic Unix tools (needed for programming assignments)
  - Text editor (e.g., emacs)
  - Compiler / linker / loader (i.e., gcc)
  - Tar archives and how to use them
  - Make and Makefiles
- Desire to learn

Course Syllabus

See the handout

or

download a copy

How We Will Proceed

- We will examine the major components of an operating system
- For a given component we will
  - Outline the functionality it provides
  - Understand principles involved
  - Study one particular design choice in depth
  - Consider implementation details and the relationship to hardware
  - Quickly review other possibilities and tradeoffs
- Note: we will cover components in a linear order that allows us to understand one component at a time without relying on later components

A FEW THINGS
TO THINK ABOUT

Real concurrency — in which one program actually continues to function while you call up and use another — is more amazing but of small use to the average person. How many programs do you have that take more than a few seconds to perform any task?

(From an article about new operating systems for the IBM PC in the New York Times, 25 April 1989)

Perfection [in design] is achieved not when there is nothing to add, but rather when there is nothing more to take away.

– Antoine de Saint-Exupery

Introduction To Operating Systems (Definitions And Functionality)

What Is An Operating System?

- Answer: a large piece of sophisticated software that provides an abstract computing environment
- An OS manages resources and supplies computational services
- An OS hides low-level hardware details from programmers
- Note: operating system software is among the most complex ever devised

Example Services An OS Supplies

- Support for concurrent execution (multiple applications running at the same time)
- Process synchronization
- Process-to-process communication mechanisms
- Process-to-process message passing and asynchronous events
- Management of address spaces and virtual memory support
- Protection among users and running applications
- High-level interface for I/O devices
- File systems and file access facilities
- Internet communication

What An Operating System Is NOT

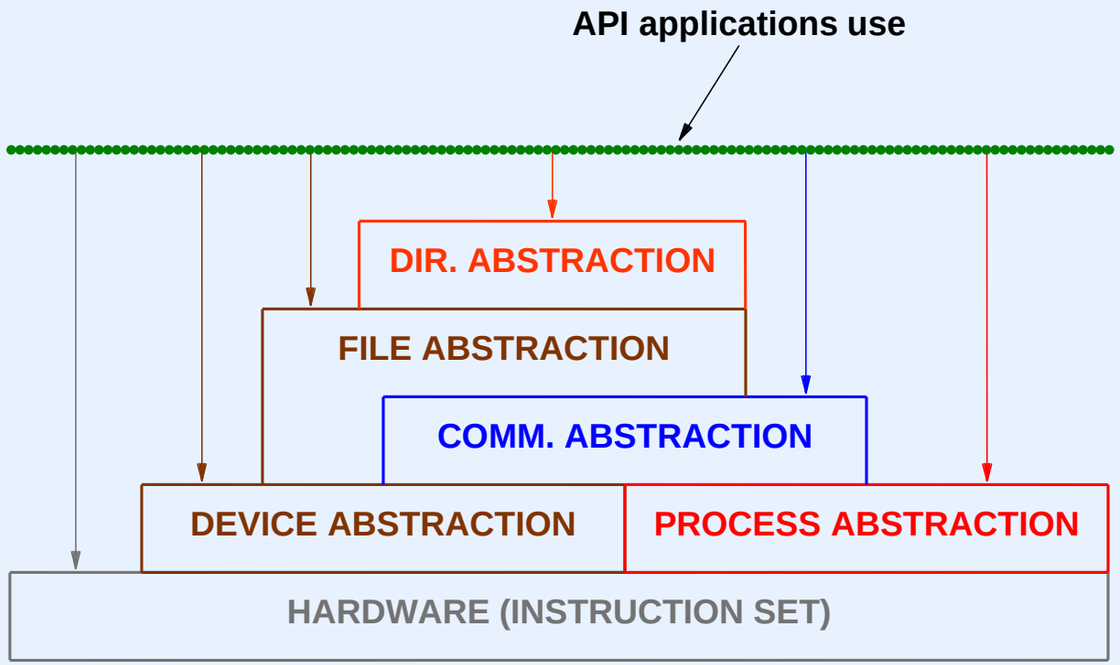
- A hardware mechanism
- A programming language
- A compiler
- A windowing system or a browser
- A command interpreter
- A library of utility functions
- A graphical desktop

AN OPERATING SYSTEM FROM THE OUTSIDE

The System Interface

- A single copy of the OS runs at any time
  - Hidden from users
  - Accessible only to application programs
- The *Application Program Interface (API)*
  - Defines services OS makes available
  - Defines arguments for the services
  - Provides access to OS abstractions and services
  - Hides hardware details

OS Abstractions And The Application Interface



- Modules in the OS offer services to applications
- Internally, some services build on others

Interface To System Services

- Appears to operate like a function call mechanism
  - OS makes a set of “functions” available to applications
  - Application supplies arguments using standard mechanism
  - Application “calls” an OS function to access a service
- Control transfers to OS code that implements the function
- Control returns to caller when function completes

Interface To System Services
(continued)

- Requires a special hardware instruction to invoke an OS function
  - Moves from the application's *address space* to OS's address space
  - Changes from application *mode* or *privilege level* to OS mode
- Terminology used by various hardware vendors
  - *System call*
  - *Trap*
  - *Supervisor call*
- We will use the generic term *system call*

An Example Of A System Call In Xinu: Output A Character To The Console

```
/* ex1.c - main */

#include <xinu.h>

/*------------------------------------------------------------------------
 * main  -  Write "hi" on the console
 *------------------------------------------------------------------------
 */
void    main(void)
{
        putc(CONSOLE, 'h');
        putc(CONSOLE, 'i');
        putc(CONSOLE, '\n');
}
```

- Note: we will discuss the implementation of *putc* later

OS Services And System Calls

- Each OS service accessed through system call interface
- Most services employ a set of several system calls
- Examples
  - Process management service includes functions to *suspend* and then *resume* a process
  - Communication service includes functions that allow an application to communicate over the Internet

System Calls Used With I/O

- Open-close-read-write paradigm
- Application
  - Uses *open* to connect to a file or device
  - Calls functions to *write* data or *read* data
  - Calls *close* to terminate use
- Internally, the set of I/O functions coordinate
  - *Open* returns a descriptor, *d*
  - *Read* and *write* operate on descriptor *d*

AN OPERATING SYSTEM FROM THE INSIDE

Operating System Properties

- An OS contains well-understood subsystems
- An OS must handle dynamic situations (processes come and go)
- Unlike most application programs, an OS uses a heuristic approach
  - A heuristic can have corner cases
  - Policies from one subsystem can conflict with policies from others
- Complexity arises from interactions among subsystems, and the side-effects can be
  - Unintended
  - Unanticipated, even by the OS designer
- We will see examples

Building An Operating System

- The intellectual challenge comes from the design of a “system” rather than from the design of any individual piece
- Structured design is needed
- It can be difficult to understand the consequences of individual choices
- We will study a hierarchical microkernel design that helps control complexity and provides a unifying architecture

Major OS Components

- Process manager
- Memory manager
- Device manager
- Clock (time) manager
- File manager
- Interprocess communication system
- Intermachine communication system
- Assessment and accounting

Our Multilevel Structure

- Organizes all components
- Controls interactions among subsystems
- Allows an OS to be understood and built incrementally
- Differs from a traditional layered approach
- Will be employed as the design paradigm throughout the text and course

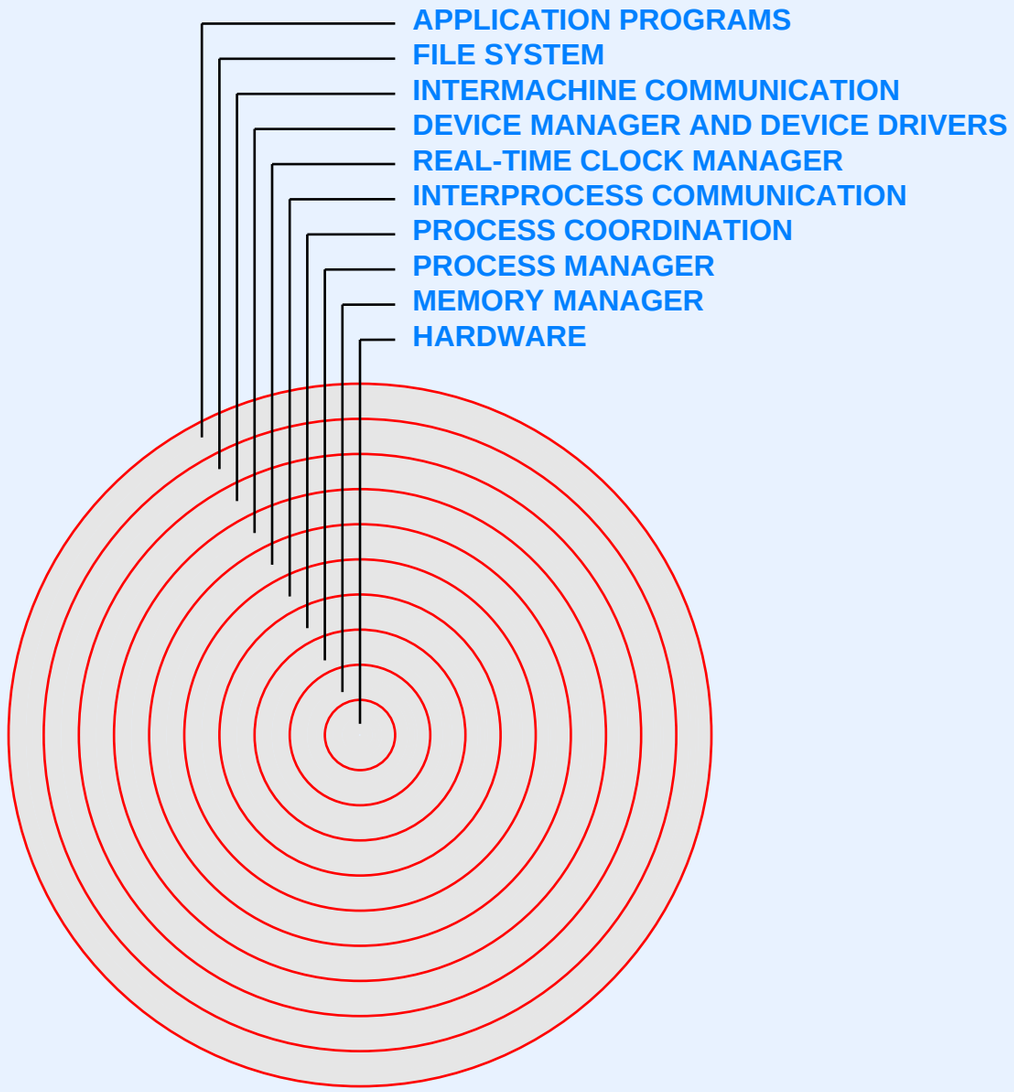
Multilevel Vs. Multilayered Organization

- Multilayer structure
  - Visible to the user as well as designer
  - Software at a given layer only uses software at the layer directly beneath
  - Examples
    * Internet protocol layering
    * MULTICS layered security structure
  - Can be extremely inefficient

Multilevel Vs. Multilayered Organization (continued)

- Multilevel structure
  - Separates all software into multiple levels
  - Allows software at a given level to use software at *all* lower levels
  - Especially helpful during system construction
  - Focuses a designer's attention on one aspect of the OS at a time
  - Helps keeps policy decisions independent and manageable
  - Is efficient

Multilevel Structure Of Xinu



How To Understand An OS

- Use the same approach as when designing a system
- Work one level at a time
- Understand the service to be provided at the level
- Consider the overall *goal* for the service
- Examine the *policies* that are used to achieve the goal
- Study the *mechanisms* that enforce the policies
- Look at an *implementation* that runs on specific hardware

A Design Example

- Example: access to I/O
- Goal: “fairness”
- Policy: First-Come-First-Served access to a given I/O device
- Mechanism: a queue of pending requests in FIFO order
- Implementation: program written in C

Questions?