Operating systems overview
  ○ Intermediary between hardware and users.
  ○ Ensures correct hardware function.
  ○ Provides high-level programming abstractions to users.
  ○ Enables resource sharing.
  ○ A simplified model:
OS concepts

- **Program**: a written representation of what to do.
- **Process**: a structure, based upon a program, that does the task described by a program.
- **Driver**: a fragment of code that controls a device.
- **System call**: a subroutine called by a program that communicates with the operating system.
- The **Operating System**: that part of the execution image that manages processes and other resources.
A very simplified model of process execution

Wednesday, September 04, 2013  6:28 PM
Hardware and OS-centric views

<table>
<thead>
<tr>
<th>Hardware</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupts</td>
<td>Signals</td>
</tr>
<tr>
<td>Contexts</td>
<td>Processes</td>
</tr>
</tbody>
</table>

**Context:** that combination of memory map, and execution data, that defines a phase of execution.

**Interrupt:** physical signal that attention is required, normally by changing the voltage on a CPU pin.

**Process:** autonomous unit of computation.

**Signal:** process-level abstraction that can interrupt a process and inform it of a state change.

A process runs in a context.
An interrupt can generate a signal.
Jobs of an operating system

- **Manage state** of processes, hardware, and devices.
- **Protect self, hardware, and devices** from errant (or malicious) processes.
- Enable and manage **resource sharing**.
Managing state

Hardware, devices (and processes!) have state.
One job of the OS: provide a coherent view of device state; ensure that multiple entities using the same device do not get confused.
States and state botches

- Each I/O device has a "state machine" that describes its states and functions.
- Purpose of a driver: maintain state, insure that one can change from existing state to any desired state.
- State (protocol) botch: process thinks device is in one state, while it is in another.
- Result: device does not work properly, or even fails.
States of a (primitive) network card
The polite illusion that this card does multiple things is maintained by the operating system.
Caveat: state botches

- If send request to device occurs when listening, it works.
- If send request to device occurs when it is receiving, it fails.
- If two parts of a program maintain different ideas of state, we have a state coherence problem. This is the reason for drivers.
- The reason that drivers exist is to avoid state coherence problems.
Another job of the OS: protect self, devices, other processes from errant or malicious processes

Buggy processes should not crash the OS. Intentionally malicious processes should not be able to take over.
Main mechanisms: memory protection, contexts.
Basics of memory protection

Each process runs in a context.
A context contains segments.
Each segment contains pages.
Each segment can be:
  Read-only.
  Read-write.
  Copy-on-write.
Process has no way to read the OS code itself.
Segments in a typical process:
- Text: contains program, read-only.
- BSS: contains global variables, read-write.
- Stack: contains block variables, read-write.
- Heap: contains dynamically allocated storage, read-write.
How the OS protects itself

Processes run in their own private address spaces.
The only way to get to the operating system is to call a system call.

Calling a system call "safely" exposes the OS:
Switches from "user mode" to "kernel mode".

(gets access to all of memory, not just one process)
Disables memory protection.
Calls the OS to do something.
Enables memory protection.
Switches from "kernel mode" back to "user mode". (process can only see its own memory)
Returns control to the process.

In other words, a process can't even see OS code, much less run it.
The OS's view of processes

Tuesday, September 04, 2012  3:37 PM

![Diagram of memory mapping and process relationships in an operating system.]

"Kernel mode"

"User mode"
Enabling resource sharing
   Multiple processes can run at once, consuming part of memory, computation time, and devices.
The OS gives out memory, and takes it away:
   When a process runs or dies.
   When a process asks for memory.
Example: new/malloc

Application

Library

Operating System

new calls malloc calls malloc calls

used span

the OS

$\text{shark}$

(# pages)
Example: printf

user span

printf chars

known format type

library

write

known length

OS
Multiprogramming (or multitasking)

Devices are slow, CPU instructions are fast. Waiting for devices could be spent doing useful work.
Idea: multitasking/latency hiding
- Hide latency waiting for slow devices by running a different program.
- Result: same time, twice as much work accomplished:
Process concepts

- Context: that information that the process needs to run
  - Values of all CPU registers
  - Image of all memory
- Context switch: changing from one context to another.
  - Switch to privileged mode.
  - Save all registers for process #1 (in process record)
  - If needed, save process memory for process #1.
  - Restore all registers for process #2.
  - If needed, restore memory for process #2.
  - Back to unprivileged mode.
  - Jump to location in other process and run.
The operating system balances overhead against interactivity.
Contexts in motion

Operating_Systems Page 24
The catch

There is no "perfect" OS
Instead, there are a series of **design**
**tradeoffs** that decide between
functionality and speed, and which kind of
functionality is more important.
Exercise rules

Wednesday, September 3, 2014           6:59 PM

Turn in ONE SHEET per group.
All sign the ONE SHEET.
Grades are recorded via LINUX LOGIN (EECS).
Exercises are returned electronically to all group members via the grades web page (progress.cgi)