Operating Systems

**Problem:** unwieldy hardware resources
  - complex and varied
    - storage, networks, displays, user interfaces...
    - many different implementations
  - limited
    - one/few processors, fixed-size memory

**Solution:** operating system
- Manage, abstract, and virtualize hardware resources
- Simpler, common interface to varied hardware
- Share limited resources among multiple processes, users
- Protect co-resident processes and users from each other

A (brief) 240 tour of Operating Systems

**Focus:** key abstractions provided by kernel
  - barely scraping surface of kernel – take a full OS course
  - "OS" often used to refer to much more than the kernel

**Abstractions:**
- process
- virtual memory
- virtual devices, I/O

**Virtualization mechanisms and hardware support:**
- context-switching
- exceptional control flow
- address translation, paging, TLBs

Processes

**Program** = code (static)
**Process** = a running program instance (dynamic)
- code + state (all registers, memory, other resources)

**Key illusions:**
- Logical control flow
  - Each process seems to have exclusive use of the CPU
- Private address space
  - Each process seems to have exclusive use of full memory

**Why are these abstractions important?**
**How are these abstractions implemented?**
Implementing logical control flow

**Abstraction:** every process has full control over the CPU

**Implementation:** time-sharing

Context Switching

*Kernel* (shared OS code) switches between processes
- Kernel code runs as part of every process (NOT its own separate process)
  - Controls scheduling: which process to run next, and when.

Control flow passes between processes via context switch.
- (how?) context = state = all registers (including PC) + memory

**Exceptions**

**Synchronous: caused by instruction**
- **Traps:** like procedure call to OS
  - Intentional: transfer control to OS to perform some function
  - Returns control to “next” instruction
- **Faults:** unintentional, maybe recoverable
  - page faults, segment protection faults, divide by zero
  - Fix and re-execute faulting instruction or abort process.
- **Aborts:** unintentional, unrecoverable
  - hardware failure detected

**Asynchronous (Interrupts): caused by external events**
- incoming I/O activity, reset button, timers
Exceptions: hardware support for OS

**transfer control to OS in response to event**

What code should the OS run to handle the event?

**User Process** | **OS**
---|---
**event** | **exception**
| **exception processing by exception handler**
| **return or abort**

Interrupt Vector

stored in memory
special register holds base address

Unique ID per type of event
ID = index into exception table (a.k.a. interrupt vector)

Handler i is called each time exception i occurs

a jump table for exceptions...

Open a file (trap/system call)

User process calls: open(filename, options)
open executes system call instruction int

Segmentation Fault

Write to invalid memory location.

User Process | **OS**
---|---
movl | **exception: page fault**
| detect invalid address
| signal process

aborts process with SIGSEGV signal
Page Fault

Write to valid memory location
... but contents currently on disk instead
(more later: virtual memory)

User Process    OS

movl exception: page fault
reexecute same instruction

Load page into memory

int a[1000];
main () {
  a[500] = 13;
}