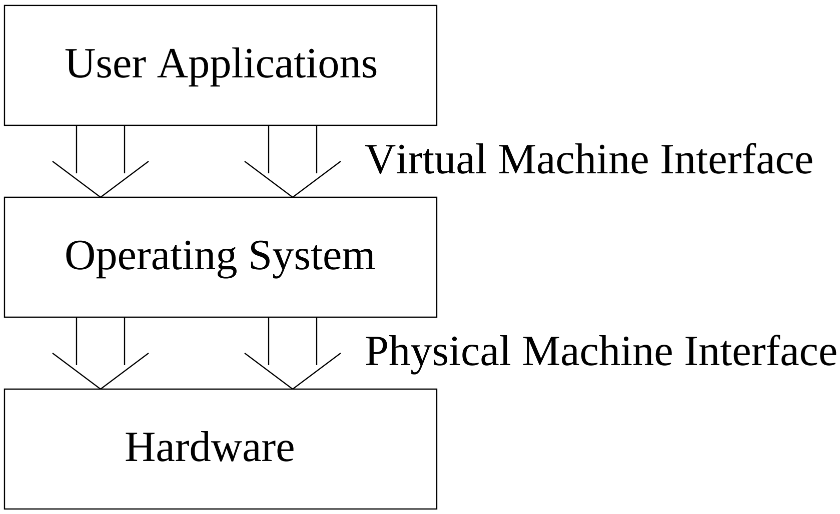
Introduction to Operating Systems *Last Time*

An operating system is the interface between the user and the architecture.



- OS development reacts to changes in hardware
- ... and changes in user expectations

OS and Computer Architecture *Today*

- Basic OS Functionality
- Architecture reminder
- What the OS can do is dictated in part by the architecture.
- Architectural support can greatly simplify the OS.
  Example: Early PC operating systems were primitive in large part because PC hardware was primitive. For instance, they did not support Virtual Memory

Modern Operating System Functionality

1. Concurrency - doing many things at the same time (I/O, processing, multiple programs, etc.)
   - Several users work at the same time as if each had a private machine
   - Threads (unit of OS control) - one thread on the CPU at a time, but many threads active concurrently
2. I/O devices - let the CPU work while an I/O device is working (especially a slow device, such as a terminal)
3. Memory management - OS coordinates allocation of memory and moving data between disk and main memory
4. Files - OS coordinates how space is used for files, in order to find files efficiently
5. Distributed systems & networks - allow a group of workstations to work together on distributed hardware

Operating System Roles

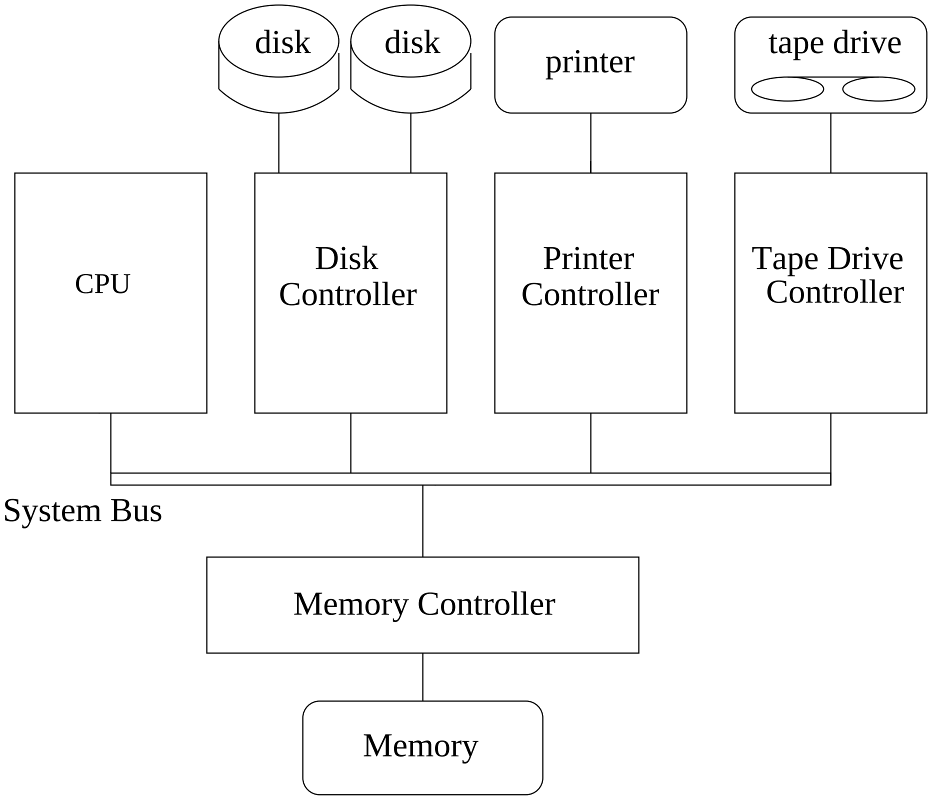
OS as juggler: providing the illusion of a dedicated machine with unbounded memory and CPU

OS as government: protecting users from each other, allocating resources efficiently and fairly, and providing secure and safe communication

OS as complex system: keeping OS design and implementation as simple as possible is the key to getting the OS to work

OS as history teacher: learning from the past to predict the future — OS design tradeoffs change with technology

Generic Computer Architecture



- CPU - the processor that performs the actual computation
- I/O devices - terminal, disks, video board, printer, etc.
- Memory - RAM containing data and programs used by the CPU
- System bus - the communication medium between the CPU, memory, and peripherals

Architectural Features Motivated by OS Services

OS Service	**Hardware Support**
Protection	Kernel/User mode
	Protected Instructions
	Base and Limit Registers
Interrupts	Interrupt Vectors
System calls	Trap instructions and vectors
I/O	Interrupts, Memory-Mapping
Scheduling, error recovery, billing	Timer
Synchronization	Atomic instructions
Virtual memory	Translation look-aside buffers

Protection

Kernel mode vs. User mode: To protect the system from aberrant users and processes (programs in execution), some instructions are restricted to use only by the OS. Users may not

- address I/O directly
- use instructions that manipulate the state of OS memory (page table pointers, TLB load, etc.)
- set the mode bits that determine user or kernel mode
- disable and enable interrupts
- halt the machine

but in kernel mode, the OS can do all these things.

The architecture must support at least kernel and user mode. (VAX/VMS had 4 protection modes.)

- A status bit in a protected processor register indicates the mode.
- Protected instructions can only be executed in kernel mode.

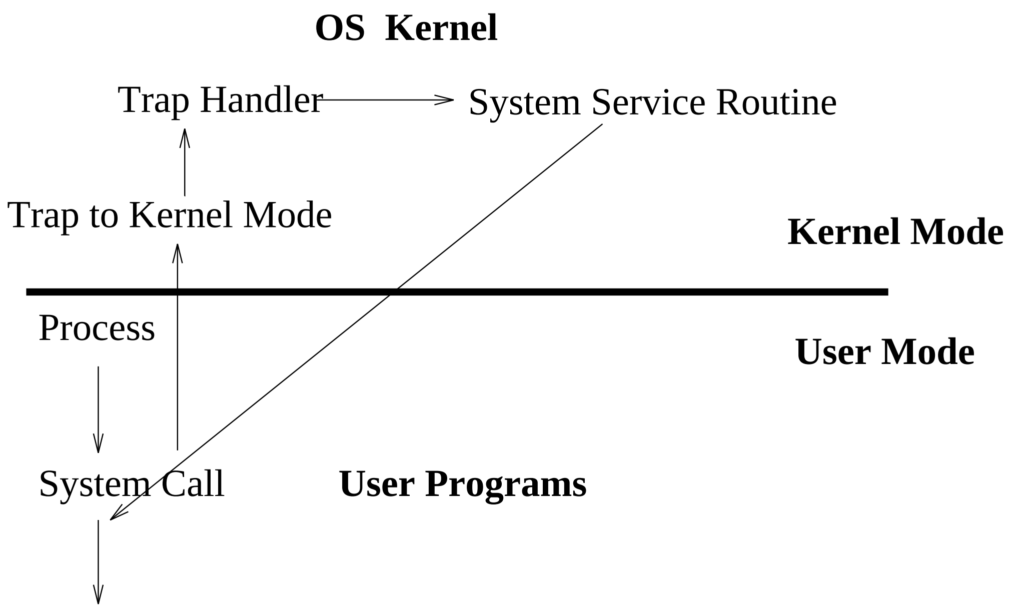
Crossing Protection Boundaries

Users can't execute privileged instructions, yet they need services that can only be accomplished using privileged instructions (e.g., I/O).

Users make a *system call* to an OS procedure to execute privileged instructions.

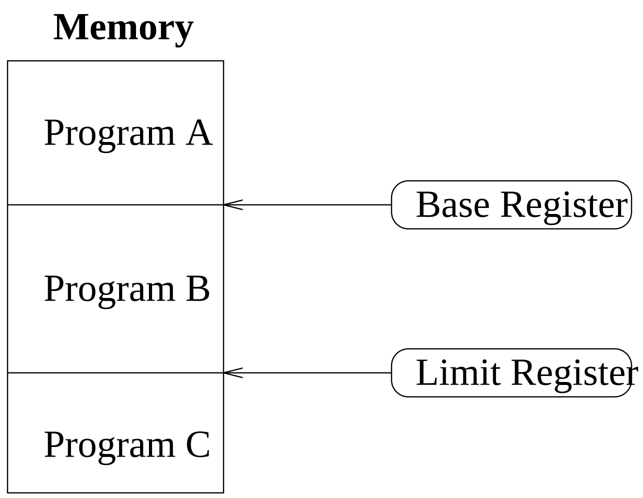
A System Call:

- A special instruction causes a trap, which vectors (jumps) to the trap handler in the OS kernel.
- The trap handler uses the parameter to the system call to jump to the appropriate handler (I/O, Terminal, etc.).
- The handler saves caller's state (PC, mode bit) so it can restore control to the user process.
- The architecture must permit the OS to verify the caller's parameters.
- The architecture must also provide a way to return to user mode when finished.


OS Kernel
Trap Handler
System Service Routine
Trap to Kernel Mode
Kernel Mode
Process
User Mode
System Call
User Programs

Memory Protection

- Architecture must provide support so the OS can
  - protect user programs from each other, and
  - protect the OS from user programs.
- Architecture may or may not protect user programs from the OS.
- The simplest technique is to use base and limit registers.



- Base and limit registers are loaded by the OS before starting a program.
- The CPU checks each user reference (instruction and data addresses), ensuring it falls between the base and limit register values.
- Virtual memory and segmented memory have additional requirements and more complex (but elegant) solutions

Traps

Architecture must detect special conditions: page fault, write to a read-only page, overflow, trace-for-debugging, bad address, privileged instruction, system call, etc. We call these **traps**. When the architecture detects these conditions, it must

- save state on trap (PC, stack, etc.), so that the process may be restarted after the trap is serviced, and then
- The CPU transfers control to the appropriate trap handler (OS routine) via a memory-mapped trap vector.
  - The CPU indexes the trap vector with the trap number,
  - then jumps to the address given in the vector, and
  - starts to execute at that address.
  - When the handler completes, the OS resumes the execution of the process that caused the trap.

Traps

Trap Vector:

0:	0x00080000	Illegal Address
1:	0x00100000	Memory Violation
2:	0x00100480	Illegal Instruction
3:	0x00123010	System Call
...	...	

Modern OS use Virtual Memory traps for many functions: debugging, distributed VM, garbage collection, copy-on-write, etc.

I/O Control

- Each I/O device has a little processor inside it that enables it to run autonomously.
- CPU issues commands to I/O devices, and continues
- When the I/O device completes the command, it issues an interrupt
- CPU stops whatever it was doing and the OS processes the I/O device's interrupt

Memory-Mapped I/O

- makes direct access to I/O controller (vs. being required to move the I/O code and data into memory)
- PCs (no virtual memory), reserve a part of the memory and put the device manager in that memory (e.g., all the bits for a video frame for a video controller)
- Access to the device then becomes almost as fast and convenient as writing the data directly into memory
- Instruction set does not need special I/O instructions

I/O Control

Interrupt-based asynchronous I/O

- Device controller has its own small processor which executes asynchronously with the main CPU.
- Device puts an interrupt signal on the bus when it is finished.
- CPU takes an interrupt.

1. save critical CPU state (hardware state),
2. disable interrupts,
3. save state that interrupt handler will modify (software state)
4. invoke interrupt handler using the *in memory Interrupt Vector* which contains a list of addresses of kernel routines to handle various events. (The Interrupt Vector works like the Trap Vector, except that for the duration of the interrupt handler, interrupts are disabled).
5. restore software state
6. enable interrupts
7. restore hardware state, and continue execution of interrupted process

Timer

- Timer device runs off "line-clock"
- Time of day
- Accounting and billing
- CPU protected from being hogged using timer interrupts that occur at say every 100 microsecond.
- At each timer interrupt, the CPU chooses a new process to execute.

Interrupt Vector:

0:	0x2ff00000	Keyboard
1:	0xfc0000b0	Mouse
2:	0x2df00000	Timer
3:	0x2ffc6810	Disk 1
...	...	

Synchronization

- Interrupts interfere with executing processes.
- OS must be able to synchronize cooperating, concurrent processes.

$\Longrightarrow$ Architecture must provide a guarantee that short sequences of instructions (e.g., read-modify-write) execute atomically. Two solutions:

1. Architecture mechanism to disable interrupts before sequence, execute sequence, enable interrupts again.
2. A special instruction that executes atomically (e.g., test&set).

Virtual Memory

- Virtual memory allows users to run programs without loading the entire program in memory at once.
- Instead, pieces of the program are loaded as they are needed.
- The OS must keep track of which pieces are in which parts of physical memory and which pieces are on disk.
- In order for pieces of the program to be located and loaded without causing a major disruption to the program, the hardware provides a translation lookaside buffer to speed the lookup.

Summary – OS and Computer Architecture

Keep your architecture book on hand.

OS provides an interface to the architecture, but also requires some additional functionality from the architecture.

$\Longrightarrow$ The OS and hardware combine to provide many useful capabilities

Homework 1 notes

Problem 1: Read up on process description in Stallings, Chapter 3.2, pages 120-127, especially Table 3.6

Problem 2: Read up on process control in Stallings, Chapter 3.3, pages 127-135

Problem 3: Practice for the programming projects. We must learn to use process-control operating system services available to user applications in C. Consult the `man` page for the `fork` function:

```
FORK(2)             Linux Programmer's Manual             FORK(2)

NAME
       fork, vfork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);
       pid_t vfork(void);

DESCRIPTION
       fork  creates a child process that differs from the parent
       process only in its PID and PPID, and  in  the  fact  that
       resource  utilizations are set to 0.  File locks and pend
       ing signals are not inherited.
```