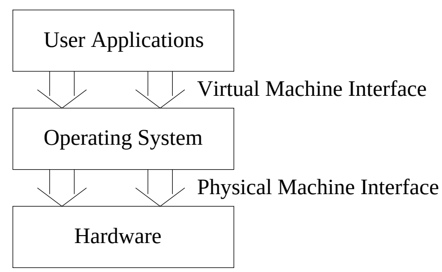
Last Class: Introduction to Operating Systems



- An operating system is the interface between the user and the architecture.
  - History lesson in change.
  - OS reacts to changes in hardware, and can motivate changes.

Today: OS and Computer Architecture

- Basic OS Functionality
- Basic Architecture reminder
- What the OS can do is dictated in part by the architecture.
- Architectural support can greatly simplify or complicate the OS.

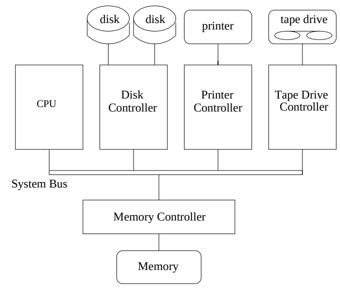
Modern Operating System Functionality

1. **Concurrency:** Doing many things simultaneously (I/0, processing, multiple programs, etc.)
   - Several users work at the same time as if each has 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 a slow I/O device is working
3. **Memory management:** OS coordinates allocation of memory and moving data between disk and main memory.
4. **Files:** OS coordinates how disk space is used to store multiple files
5. **Distributed systems & networks:** allow a group of workstations to work together on distributed hardware

Summary of Operating System Principles

- **OS as juggler:** providing the illusion of a dedicated machine with infinite 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 past to predict the future, i.e., 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: communication medium between 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 trap vectors
I/O	Interrupts or 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 processors, some instructions are restricted to use only by the OS. Users may not

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

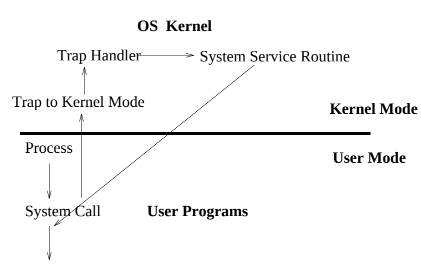
but in kernel mode, the OS can do all these things.

The hardware must support at least kernel and user mode.

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

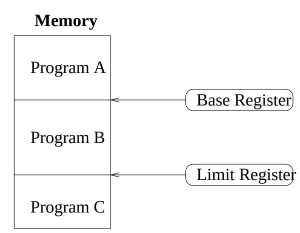
Crossing Protection Boundaries

- **System call:** OS procedure that executes privileged instructions (e.g., I/O)
  - 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.



Memory Protection

- Architecture must provide support so that the OS can
  - protect user programs from each other, and
  - protect the OS from user programs.
- 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.



Traps

- **Traps:** special conditions detected by the architecture
  - Examples: page fault, write to a read-only page, overflow, systems call
- On detecting a trap, the hardware
  - Saves the state of the process (PC, stack, etc.)
  - Transfers control to appropriate trap handler (OS routine)
    * The CPU indexes the memory-mapped trap vector with the trap number,
    * then jumps to the address given in the vector, and
    * starts to execute at that address.
    * On completion, the OS resumes execution of the process

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.
- Traps are a performance optimization. A less efficient solution is to insert extra instructions into the code everywhere a special condition could arise.

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/0 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

- Enables 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.

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*
  5. Restore software state
  6. Enable interrupts
  7. Restore hardware state, and continue execution of interrupted process

Timer & Atomic Instructions

Timer

- 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.

⇒ 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

Keep your architecture book on hand.

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

⇒ The OS and hardware combine to provide many useful and important features.