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.

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.
Course Staff Office Hours

• Instructor: Prashant Shenoy  
  – TuTh 9:45 to 10:45, CS building, room 336 or by appt

• TAs:
  – Anna Deng
    • Wed: 2-3pm, CS building room 207
    • Fri: 4-5pm, CS building room 207
    • Email: adeng [at] cs.umass.edu
  – Wenzhao Liu
    • Mon 2-4pm, CS Building room 207
    • Email: wenzhao [at] cs.umass.edu

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.
Computer Architecture Basics

- Picture of a motherboard/logicboard

Generic Computer Architecture

- **CPU**: the processor that performs the actual computation
  - Multiple “cores” common in today’s processors
- **I/O devices**: terminal, disks, video board, printer, etc.
  - Network card is a key component, but also an I/O device
- **Memory**: RAM containing data and programs used by the CPU
- **System bus**: communication medium between CPU, memory, and peripherals
Modern Operating System Functionality

- **Process and Thread Management**
- **Concurrency**: Doing many things simultaneously (I/O, 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
- **I/O devices**: let the CPU work while a slow I/O device is working
- **Memory management**: OS coordinates allocation of memory and moving data between disk and main memory.
- **Files**: OS coordinates how disk space is used for files, in order to find files and to store multiple files
- **Distributed systems & networks**: allow a group of machines to work together on distributed hardware

Architectural Features Motivated by OS Services

<table>
<thead>
<tr>
<th>OS Service</th>
<th>Hardware Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Kernel/user mode, protected instructions, base/limit registers</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Interrupt vectors</td>
</tr>
<tr>
<td>System calls</td>
<td>Trap instructions and trap vectors</td>
</tr>
<tr>
<td>I/O</td>
<td>Interrupts and memory mapping</td>
</tr>
<tr>
<td>Scheduling, error recovery, accounting</td>
<td>Timer</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Atomic instructions</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>Translation look-aside buffers</td>
</tr>
</tbody>
</table>
Protection

- CPU supports a set of assembly instructions
  - MOV [address], ax
  - ADD ax, bx
  - MOV CRn (move control register)
  - IN, INS (input string)
  - HLT (halt)
  - LTR (load task register)
  - INT n (software interrupt)

- Some instructions are sensitive or privileged

**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); also API exported by the kernel
  - 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.

![Diagram: User process calling system call, then returning to user mode]

**Example System calls**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid = fork()</code></td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td><code>pid = waitpid(pid, &amp;statloc, options)</code></td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td><code>s = execve(name, argv, environp)</code></td>
<td>Replace a process' core image</td>
</tr>
<tr>
<td><code>exit(status)</code></td>
<td>Terminate process execution and return status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = open(file, how, ...)</code></td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get a file’s status information</td>
</tr>
</tbody>
</table>
Windows System Calls

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td>waitpid</td>
<td>WaitForSingleObject</td>
<td>Can wait for a process to exit</td>
</tr>
<tr>
<td>execve</td>
<td>(none)</td>
<td>CreateProcess = fork + execve</td>
</tr>
<tr>
<td>exit</td>
<td>ExitProcess</td>
<td>Terminate execution</td>
</tr>
<tr>
<td>open</td>
<td>CreateFile</td>
<td>Create a file or open an existing file</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close a file</td>
</tr>
<tr>
<td>read</td>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>write</td>
<td>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>lseek</td>
<td>SetFilePointer</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>stat</td>
<td>GetFileAttributesEx</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>link</td>
<td>(none)</td>
<td>Win32 does not support links</td>
</tr>
<tr>
<td>unlink</td>
<td>DeleteFile</td>
<td>Destroy an existing file</td>
</tr>
<tr>
<td>mount</td>
<td>(none)</td>
<td>Win32 does not support mount</td>
</tr>
<tr>
<td>umount</td>
<td>(none)</td>
<td>Win32 does not support mount</td>
</tr>
<tr>
<td>chdir</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>chmod</td>
<td>(none)</td>
<td>Win32 does not support security (although NT does)</td>
</tr>
<tr>
<td>kill</td>
<td>(none)</td>
<td>Win32 does not support signals</td>
</tr>
<tr>
<td>time</td>
<td>GetLocalTime</td>
<td>Get the current time</td>
</tr>
</tbody>
</table>

Some Win32 API calls

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
Process Layout in Memory

- Processes have three segments: text, data, stack

Registers

- Register = dedicated name for one word of memory managed by CPU
  - General-purpose: “AX”, “BX”, “CX” on x86
  - Special-purpose:
    - “SP” = stack pointer
    - “FP” = frame pointer
    - “PC” = program counter
- Change processes:
  save current registers & load saved registers = context switch
Memory Hierarchy

- Higher = small, fast, more $, lower latency
- Lower = large, slow, less $, higher latency

Caches

- Access to main memory: “expensive”
  - ~ 100 cycles (slow, but relatively cheap ($))
- Caches: small, fast, expensive memory
  - Hold recently-accessed data (D$) or instructions (I$)
  - Different sizes & locations
    - Level 1 (L1) – on-chip, smallish
    - Level 2 (L2) – on or next to chip, larger
    - Level 3 (L3) – pretty large, on bus
  - Manages lines of memory (32-128 bytes)

- Caches are managed by hardware (no explicit OS management)
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

<table>
<thead>
<tr>
<th>Trap Vector:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 0x00080000 Illegal address</td>
</tr>
<tr>
<td>1: 0x00100000 Memory violation</td>
</tr>
<tr>
<td>2: 0x00100480 Illegal instruction</td>
</tr>
<tr>
<td>3: 0x00123010 System call</td>
</tr>
</tbody>
</table>

- 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.
- Recap of System Calls from page 8
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

Three I/O Methods

- Synchronous, asynchronous, memory-mapped

![Synchronous and Asynchronous Diagram](image-url)
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:

<table>
<thead>
<tr>
<th>Vector</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x2ff080000</td>
<td>keyboard</td>
</tr>
<tr>
<td>1</td>
<td>0x2ff100000</td>
<td>mouse</td>
</tr>
<tr>
<td>2</td>
<td>0x2ff100480</td>
<td>timer</td>
</tr>
<tr>
<td>3</td>
<td>0x2ff123010</td>
<td>Disk 1</td>
</tr>
</tbody>
</table>

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.