Introduction to OSs

- What is an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System
The OS Manages System Resources

- Disks and other Devices
- Timers / Clocks
- Locks
- Memory
- Power / Heat
- I/O Controllers
- ... (other resources)

The OS as Servant to Two Masters

- Devices
- Clocks & Timers
- Locks
- Memory
- Heat & Power
- I/O Controllers
- CPUs

- Performance
- Plug & Play
- Security
- Predictability
- Convenience
- Fault-Tolerance
- Power-Effectiveness
- ... (other attributes)
What, then, is an Operating System?

- Controls and coordinates the use of system resources.

- **Primary goal**: Provide a convenient environment for a user to access the available resources (CPU, memory, I/O)
  - Provide appropriate abstractions (files, processes, ...)
  - “virtual machine”

- **Secondary goal**: Efficient operation of the computer system.

- **Resource Management**
  - **Transforming**: Create virtual substitutes that are easier to use.
  - **Multiplexing**: Create the illusion of multiple resources from a single resource
  - **Scheduling**: “Who gets the resource when?”

Introduction to OSs

- What is an Operating System?

- **Architectural Support for Operating Systems**

- System Calls

- Basic Organization of an Operating System
Architectural Support for OS’s

- Dealing with Asynchronous Events: Exceptions, Interrupts
  - Modern OS’s are interrupt-driven (some are not!).
  - Simple interrupt handling vs. exception handling MIPS-style.

- Hardware Protection
  - Privilege Levels (e.g. user/kernel/supervisor, etc.)
  - Privileged instructions: typically CPU control instructions
  - I/O Protection
  - Memory Protection

- Support for Address Spaces

- Timers

Modern OS’s are Interrupt-Driven

CPU
servicing
interrupt
process
executing
IO Device
busy
idle
keyboard
pressed
idle
Interrupts / Exceptions

- When an interrupt occurs, CPU stops, saves state, typically changes into supervisor mode, and immediately jumps to predefined location.
- Appropriate interrupt service routine is found through the interrupt vector.
- Return-from-interrupt automatically restores state.

Interrupts/Exceptions can be invoked by asynchronous events (I/O devices, timers, various errors) or can be software-generated (system calls).

Exceptions, MIPS-Style

- MIPS CPU deals with exceptions.
  - Interrupts are just a special case of exceptions.

- The MIPS Architecture has no interrupt-vector table!
  - All exceptions trigger a jump to the same location, and de-multiplexing happens in the exception handler, after looking up the reason for the exception in the CAUSE register.
MIPS Exception Handler (low-level)

<table>
<thead>
<tr>
<th>xcptlow_handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>set up exception frame on stack</td>
</tr>
<tr>
<td>save enough registers to get by</td>
</tr>
<tr>
<td>save rest of registers</td>
</tr>
<tr>
<td>call C exception handler</td>
</tr>
<tr>
<td>restore registers</td>
</tr>
<tr>
<td>return from exception</td>
</tr>
</tbody>
</table>

Hardware Protection

- Originally: User owned the machine, no monitor. No protection necessary.
- Resident monitor, resource sharing: One program can adversely affect the execution of others.
- Examples
  - `halt` and other instructions
  - modify data or code in other programs or monitor itself
  - access/modify data on storage devices
  - refuse to relinquish processor
- Benign (bug) vs. malicious (virus)
Hardware Protection (2)

- Dual-mode operation
  - user mode vs. supervisor mode
  - e.g. halt instruction is privileged.
- I/O Protection
  - define all I/O operations to be privileged
- Memory Protection
  - protect interrupt vector, interrupt service routines
  - determine legal address ranges

Timers

- Timers can be set, and a trap occurs when the timer expires. (And OS acquires control over the CPU.)

- Other uses of timers:
  - time sharing
  - time-of-day
Introduction to OSs

- What is an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

External Structure of an OS

The outsider’s view of the OS.
System Calls

Provide the interface between a process and the OS.

Example: vanilla copy:

```c
int copy(char * fname1, *fname2) {
    FILE *f, *g;
    char c;
    f = fopen(fname1, "r");
    g = fopen(fname2, "w");
    while (read(f, &c, 1) > 0)
        write(g, c, 1);
    fclose(f);
    fclose(g);
}
```

System Call Implementation: Linux on x86

- Example: _syscall(int, setuid, uid_t, uid)
- expands to:

  ```asm
  _setuid:
  subl $4,esp
  pushl %ebx
  movdel 12(%esp),%eax
  movl %eax,4(%esp)
  movl $23,%eax
  mov 4(%esp),%edx
  int $0x80
  <------ call transfer to kernel entry point _system_call()
  movl %eax,%edx
  testl %edx,%edx
  jge l2
  negl %edx
  movl %edx,%eax
  movl 5-1,tlax
  popl %ebx
  addl $4,%esp
  retl2:
  movl %edx,tlax
  popl %edx
  addl $4,%esp
  ret
  ```
Why Interrupts?

**Reason 1:** Can load user program into memory without knowing exact address of system procedures.

**Reason 2:** Separation of address space, including stacks: *user stack* and *kernel stack*.

**Reason 3:** Automatic change to *supervisor mode*.

**Reason 4:** Can control *access* to kernel by masking interrupts.

---

**Reason 2: Buffer Overrun Attacks** *(Silberschatz et al)*

```c
#include <stdio.h>
define BUFFER_SIZE 256
int main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE];
    if (argc < 2)
        return -1;
    else {
        strcpy(buffer,argv[1]);
        return 0;
    }
}
```

---

[Example and illustrations from Silberschatz et al. "Operating Systems Concepts" Ch. 15]
Stack Separation sufficient?

- Buffer overruns in kernel code?
- Device drivers?

Reason 4: Mutual Exclusion in Kernel

1. User process 1 in user space
2. System call
3. Trap
   - Process 1 executing in kernel
   - Interrupts are masked
4. Process 2 cannot enter kernel because of masked interrupts
5. Unmask interrupts and return
Introduction to OSs

- What is an Operating System?
- Architectural Support for Operating Systems
- System Calls
- Basic Organization of an Operating System

External Structure of an OS

The outsider’s view of the OS.

- Applications programs/processes
- System call interface
- Kernel
- Device drivers
- Hardware
Internal Structure: Layered Services

The insider's view of the OS.
Example: XINU [Comer 1984]

- user programs
- file system
- intermachine network communication
- device manager and device drivers
- real-time clock manager
- interprocess communication
- process coordinator
- process manager
- memory manager
- hardware

Internal Structure: μ-Kernels

- Layered Kernels vs. Microkernels

Hierarchical decomposition. Interaction only between adjacent layers.

Kernel has only core operating system functions (memory management, IPC, I/O, interrupts)
Other functions run in server processes in user space.
Operations in a µ-Kernel

- Non-kernel components of the OS are implemented as server processes.
- Communication between user and servers using messages through kernel.
- “client-server architecture within a single computer”
- Examples: Mach, Windows NT, Chorus, L4, ...

Windows 2000 System Structure

Figure 2: System architecture
Benefits of µ-Kernels

- Extensibility:
  - New services can be added by adding server processes.
- Flexibility:
  - Services can be customized.
- Portability:
  - Kernel small, with well-defined interface.
- Distributed System Support:
  - Interface between users and services is message-based.

µ-Kernels: Performance is Problem

- Request traverses user/kernel boundary twice, same for reply.
- Solutions:
  - Move critical services back into the kernel ("make kernel bigger")
  - Make kernel "smaller"
Exokernel: Library Operating Systems

- Abstractions should be implemented at application level.
- Library OSs use exokernel interface, and implement higher-level abstractions to best meet performance and functionality goals of applications.