CSCE 313 Introduction to Computer Systems

Instructor: Guofei Gu

http://courses.cse.tamu.edu/guofei/csce313/

Introduction to OSs

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

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

Four Components of a Computer System

[Diagram showing four components: user, compiler, assembler, text editor, database, system and application programs, operating system, computer hardware]
What is an operating system?

- What an operating system is **not**:
  - An o.s. is **not** a language or a compiler
  - An o.s. is **not** a command interpreter / window system
  - An o.s. is **not** a library of commands
  - An o.s. is **not** a set of utilities

A Short Historical Tour

- **First Generation** Computer Systems (1949-1956):
  - Single user: writes program, operates computer through console or card reader / printer
  - Absolute machine language
  - I/O devices
  - Development of libraries; device drivers
  - Compilers, linkers, loaders
  - Relocatable code
Programming Early Machines

Wiring the ENIAC with a new program
(U.S. Army photo, from archives of the ARL Technical Library)

Second-Generation Computers (1956-1963)

- Problems: scheduling, setup time
- Automation of Load/Translate/Load/Execute
  - Batch systems
  - Monitor programs
  - Job Control Language
  - Advent of operators: computers as input/output box
- Problem: Resource management and I/O still under control of programmer
  - Memory protection
  - Timers
  - Privileged instructions
Example: IBM Punch Card System

Card Punch

Card Verifier

Card Sorter

(Computer Museum of America)

Batching Program Execution

$FTN

$JOB

$END

... Data ...

$RUN

... Program ...

$LOAD

...
Overlapping CPU and I/O Operations

Traditional Batch Operation:
- Card reader → CPU → Line printer

Off-Line Processing:
- Card readers → CPU → Line printers

Spooling: I/O Channels:
- Disk/tape

Off-Line vs. Pure Batch

- **batch**
  - Card reader → CPU → Printer

- **off-line** *(single set of card reader/printer)*
  - Card reader → CPU → Printer
  - Rewind setup
  - Tape reader
  - CPU
  - Tape reader
  - Printer
  - Setup rewind
Off-Line vs. Pure Batch (II)

- **batch**
  - Card reader
  - CPU
  - Printer

- **off-line (multiple of card readers/printers)**
  - Card reader
  - Tape reader
  - CPU
  - Tape reader
  - Printer


- Problem with batching: one-job-at-a-time
  - Sequential: CPU
  - Better: CPU

- Solution: **Multiprogramming**
  - Job pools: have several programs ready to execute
  - Keep several programs in memory

- New issues:
  - Job scheduling
  - Memory management
  - Protection
Time Sharing (mid 1960s on)

- OS interleaves execution of multiple user programs with time quantum
  - CTSS (1961): time quantum 0.2 sec

- User returns to own the machine

- New aspects and issues:
  - On-line file systems
  - Resource protection
  - Virtual memory
  - Sophisticated process scheduling

- Advent of systematic techniques for designing and analyzing OSs.

The Recent Past

- Personal computers and Computing as Utility
  - History repeats itself
- Parallel systems
  - Resource management
  - Fault tolerance
- Real-Time Systems
- Distributed Systems
  - Communication
  - Resource sharing
  - Network operating systems
  - Distributed operating systems
- Secure Systems
More

- Auto system
- Mobile system
- Home system

The Future?

- The “Invisible Computer”
- Computing-in-the-ultra-small
- Speed vs. Power vs. Heat
- Breaking up the layered design
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?”

---

**Resources**

- OS
- CPU
- Memory
- Disks and other Devices
- I/O Controllers
- Power / Heat
- Locks
- Timers / Clocks
The OS as Servant to Two Masters

Introduction to OSs

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

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles

Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt
How a Modern Computer Works

Architectural Support for OS’s

- Dealing with Asynchronous Events: Exceptions, Interrupts
  - Modern OS’s are interrupt-driven (some still 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

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.

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

```
CPU >= base
  |   <= base + limit
  |    <
  |      no
  |    no
  |  trap to operating system!
  |
```

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
Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

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

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

(Note that the system-call names used throughout this text are generic)
**Standard C Library Example**

- C program invoking printf() library call, which calls write() system call

```c
#include <stdio.h>
int main ()
{
    ...
    printf ("Greetings");
    ...
    return 0;
}
```

**API – System Call – OS Relationship**

[Diagram showing the relationship between user application, system call interface, and kernel mode]
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table
System Call Implementation: Linux on x86

- Example (setuid system call is coded as):
  \_syscalll(int, setuid, uid_t, uid)
- expands to:

```
_syscall:
  subl $4,%esp
  pushl %ebx
  movl %edx,%eax
  movl %eax,4(%esp)
  movl "$setuid",%eax
  movl "$setuid",%eax
  int $0x80
  movl %eax,%eax
  testl %eax,%eax
  jge L2
  negl %eax
  movl %eax,_errno
  movl $-1,%eax
  popl %ebx
  addl $4,%esp
L2:
  movl %eax,%eax
  popl %ebx
  addl $4,%esp
  ret
```

Examples of Windows and Unix System Calls

<table>
<thead>
<tr>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Creation</td>
<td>fork()</td>
</tr>
<tr>
<td>Control</td>
<td>exit()</td>
</tr>
<tr>
<td>File Manipulation</td>
<td>open()</td>
</tr>
<tr>
<td>ReadFile()</td>
<td>read()</td>
</tr>
<tr>
<td>WriteFile()</td>
<td>write()</td>
</tr>
<tr>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
<tr>
<td>Device Manipulation</td>
<td>ioctl()</td>
</tr>
<tr>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
<tr>
<td>Information</td>
<td>getpid()</td>
</tr>
<tr>
<td>SetTimer()</td>
<td>slew()</td>
</tr>
<tr>
<td>Sleep()</td>
<td>slew()</td>
</tr>
<tr>
<td>Communication</td>
<td>pipe()</td>
</tr>
<tr>
<td>CreateFileMapping()</td>
<td>shmat()</td>
</tr>
<tr>
<td>MapViewOfFile()</td>
<td>map()</td>
</tr>
<tr>
<td>Protection</td>
<td>chmod()</td>
</tr>
<tr>
<td>SetFileSecurity()</td>
<td>chown()</td>
</tr>
<tr>
<td>InitializeSecurityDescriptor()</td>
<td>seawak()</td>
</tr>
<tr>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
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.

---

What does a process look like? (Unix)

- Process address space divided into “segments”
  - text (code), data, heap (dynamic data), and stack

```
stack  address 2^n-1

heap
initialized data

code
address >= 0
```
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;
    }
}
```

Stack Separation sufficient?

- Buffer overruns in kernel code?
- Device drivers?
Reason 4: Mutual Exclusion in Kernel

1. User process 1 sends a system call.
2. The system call causes a trap.
3. 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.

A View of Operating System Services

<table>
<thead>
<tr>
<th>user and other system programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
</tr>
<tr>
<td>user interfaces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>program execution</td>
</tr>
<tr>
<td>error detection</td>
</tr>
</tbody>
</table>

operating system

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

Traditional UNIX System Structure

<table>
<thead>
<tr>
<th>Kernel interface to the hardware</th>
<th>CPU scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>page replacement</td>
</tr>
<tr>
<td></td>
<td>demand paging</td>
</tr>
<tr>
<td></td>
<td>virtual memory</td>
</tr>
<tr>
<td>terminal controllers</td>
<td>file system</td>
</tr>
<tr>
<td>terminals</td>
<td>swapping block I/O</td>
</tr>
<tr>
<td></td>
<td>system</td>
</tr>
<tr>
<td>character I/O system</td>
<td>disk and tape drivers</td>
</tr>
<tr>
<td>terminal drivers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Internal Structure: $\mu$-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 $\mu$-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, ...
Benefits of $\mu$-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.

$\mu$-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"
Modules

- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Solaris Modular Approach
Windows 2000 Architecture

Why are OSs so Slow?

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Abbreviation</th>
<th>RISC/CISC</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS M2000</td>
<td>M2000</td>
<td>RISC</td>
<td>20</td>
</tr>
<tr>
<td>DECstation 3100</td>
<td>DS3100</td>
<td>RISC</td>
<td>13</td>
</tr>
<tr>
<td>Sun-4/280</td>
<td>Sun4</td>
<td>RISC</td>
<td>9</td>
</tr>
<tr>
<td>VAX 8800</td>
<td>8800</td>
<td>CISC</td>
<td>6</td>
</tr>
<tr>
<td>Sun-3/75</td>
<td>Sun3</td>
<td>CISC</td>
<td>3.8</td>
</tr>
<tr>
<td>Microvax II</td>
<td>MVAX2</td>
<td>CISC</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 1: Hardware Platforms

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (microseconds)</th>
<th>MIPS-Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2000 RISC/OS 4.0</td>
<td>18</td>
<td>0.54</td>
</tr>
<tr>
<td>DS3100 Sprite</td>
<td>26</td>
<td>0.49</td>
</tr>
<tr>
<td>DS3100 Ulitrix 3.1</td>
<td>25</td>
<td>0.60</td>
</tr>
<tr>
<td>8800 Ulitrix 2.0</td>
<td>28</td>
<td>1.15</td>
</tr>
<tr>
<td>Sun4 SunOS 4.0</td>
<td>32</td>
<td>0.68</td>
</tr>
<tr>
<td>Sun4 Sprite</td>
<td>32</td>
<td>0.58</td>
</tr>
<tr>
<td>Sun3 Sprite</td>
<td>92</td>
<td>1.0</td>
</tr>
<tr>
<td>Sun3 SunOS 3.5</td>
<td>108</td>
<td>1.0</td>
</tr>
<tr>
<td>MVAX2 Ulitrix 3.0</td>
<td>207</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 2: Getpid kernel call time
Why are OSs so Slow? (2)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Abbreviations</th>
<th>RISC/CISC</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS M2000</td>
<td>M2000</td>
<td>RISC</td>
<td>20</td>
</tr>
<tr>
<td>DECstation 3100</td>
<td>DS3100</td>
<td>RISC</td>
<td>13</td>
</tr>
<tr>
<td>Sun-4/280</td>
<td>Sun4</td>
<td>RISC</td>
<td>9</td>
</tr>
<tr>
<td>VAX 8800</td>
<td>8800</td>
<td>CISC</td>
<td>6</td>
</tr>
<tr>
<td>Sun-3/75</td>
<td>Sun3</td>
<td>CISC</td>
<td>1.8</td>
</tr>
<tr>
<td>Microvax II</td>
<td>MVAX2</td>
<td>CISC</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 1: Hardware Platforms

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (ms)</th>
<th>MIPS-Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2000 RISC/m-4.0</td>
<td>0.30</td>
<td>0.70</td>
</tr>
<tr>
<td>DS3100 Ultras 3.1</td>
<td>0.24</td>
<td>0.96</td>
</tr>
<tr>
<td>DS3100 Sprite</td>
<td>0.51</td>
<td>0.65</td>
</tr>
<tr>
<td>8800 Ultras 3.0</td>
<td>0.70</td>
<td>1.00</td>
</tr>
<tr>
<td>Sun4 SunOS 4.0</td>
<td>1.02</td>
<td>0.47</td>
</tr>
<tr>
<td>Sun4 Sprite</td>
<td>1.17</td>
<td>0.41</td>
</tr>
<tr>
<td>Sun3 SunOS 3.5</td>
<td>2.36</td>
<td>1.00</td>
</tr>
<tr>
<td>Sun3 Sprite</td>
<td>2.41</td>
<td>1.00</td>
</tr>
<tr>
<td>MVAX2 Ultras 3.0</td>
<td>3.66</td>
<td>1.30</td>
</tr>
</tbody>
</table>

Table 3: Cswthr: echo one byte between processes using pipes.

Why are OSs so Slow? (3)

- The benchmarks suggest at least two possible factors that contribute to non-scalability of operating systems
  - The first is memory bandwidth, which has not scaled to match processor speed in faster machines
  - The second factor is file systems, some of which require synchronous disk I/Os in common situations. The synchronous I/O requirements limit the performance of operating systems when processors get faster but disks don’t.