Introduction to OSs

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

- Problems: scheduling, setup time
- Automation of Load/Translate/Load/Execute
  - Batch systems
  - Monitor programs

```
+----------------+
|       Monitor   |
| user program area|
+----------------+  
  device drivers → job sequencer / loader
```

- 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


- Problem with batching: one-job-at-a-time
  - sequential:
    - CPU
    - I/O
  - better:
    - CPU
    - I/O

```
  sequential:   
              CPU  I/O
  better:      CPU  I/O
``` 

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

```
+----------------+         
|       Monitor  |
|       Job1     |
|       Job2     |
|       JobN     |
+----------------+
```

- 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
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?”

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
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 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

- 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).

Interrupts / Exceptions
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 \texttt{CAUSE} register.

MIPS Exception Handler (low-level)

- \texttt{xcptlow_handler}
  - set up exception frame on stack
  - save enough registers to get by
  - save rest of registers
  - call C exception handler
  - restore registers
  - return from exception
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

---

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

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:

  ```
  _setuid:
    subl $4, %esp
    pushl %ebx
    movl 12(%esp), %eax
    movl %eax, 4(%esp)
    movl $23, %eax  \<----- System Call number (setuid = 23)
    movl 4(%esp), %ebx
    int $0x80
    movl %eax, %edx
    testl %edx, %edx
    jge L2
    negl %edx
    movl %edx, _errno
    movl $-1, %eax
    popl %ebx
    addl $4, %esp
  L2:
    movl %edx, %eax
    popl %ebx
    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 4: Mutual Exclusion in Kernel

1. User process 1
   - User space
   - System call
   - Kernel
   - Trap
   - Process 1 executing in kernel
     - Interrupts are masked
   
2. User process 2
   - Trap
   - Process 2 cannot enter kernel because of masked interrupts
   - 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
- 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**

  ![Layered Kernels vs. Microkernels Diagram]

  - 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, ...
Figure 2 – System architecture