CS 439
Principles of Computer Systems

2: Anatomy of an Operating System

Operating System

Hardware

Operating System

Web Browser + JavaScript Engine

Hardware
Operating System?

Today

- What is the operating system.
- Common characteristics of an OS
- How did OS develop to their current form.
- Little history of OS
- How are OS typically structured
  - Common services
  - Alternative designs

Operating Systems

- Abstraction
  - Execution Contexts
  - Some Portability for Applications
  - Devices
- Protection
  - Fault Isolation
  - Permissions / Access Control
- Resource Management
  - Performance Isolation
  - Multiplexing
  - Physical (limited) memory vs. virtual memory

History of Computing and Operating Systems

- Hand programmed machine

Hardware is very expensive, Human labor is very cheap.
Little abstraction
Little protection
Little resource management
**Batch Processing (‘55-‘65)**

Operating system = *loader + sequencer + output processor*

- **User Data**
- **User Program**
- "System Software"
- **Operating System**

**Needs protection**

- **Card Reader**
- **Tape**
- **Compute**
- **Output**

**Input**

---

**Multiprogramming (‘65-‘80)**

Keep several jobs in memory and multiplex CPU between jobs

**Program P**

```
begin
  Read(var)
end P
```

Simple, "synchronous" input: What to do while we wait for the I/O device?

---

**Human labor still very cheap.**

---

**Multiprogramming (‘65-‘80)**

Keep several jobs in memory and multiplex CPU between jobs

**Program 1**

```
main{
  read()
}
```

**OS**

```
startIO()
waitIO()
endIO()
```

**I/O Device**

```
interrupt
```

---

**User Program n**

```
begin
  Read()
end P
```

**System Software**

```
begin
  StartIO(input device)
  WaitIO(interrupt)
  EndIO(input device)
end
```

**Operating System**
Multiprogramming (‘65–‘80)

Keep several jobs in memory and multiplex CPU between jobs

Program 1  OS  Program 2  I/O Device

main{
  k: read()
  read{
    startIO()
    schedule()
    main(
      )
  }
}

interrupt

More resource management

User Program n

... User Program 2

User Program 1

"System Software"

Operating System

Timesharing (‘70–)

A timer interrupt is used to multiplex CPU among jobs

Program 1  OS  Program 2

main{
  k: timer interrupt
  schedule()
  main(
    )
}

k+1:

schedule(
)

timer interrupt

User Program n

... User Program 2

User Program 1

"System Software"

Operating System

History of OS

○ Terminals

Hardware is getting cheaper, Human labor is more expensive.

More abstraction

Protection

Resource management

○ Personal Computers

Hardware is getting even more cheaper, Human labor is more expensive.

Abstraction

Protection

Resource management
History of OS

- Internet, Cloud Computing

Protection

- What could go wrong with an application?
  - Execute illegal instruction
    - E.g., division by zero
  - Write on memory region belonging to another application
    - Could compromise the program code of other application
    - Could compromise the confidentiality of the data in memory
  - Hog resources
    - DOS attack

- This can be unintentionally (bug) or intentionally (malicious program)

Protection

- We want to give applications limited control over the hardware
- Certain operations should not be possible from user code
  - Access devices directly
  - Manipulate OS data structures
  - Disable or enable interrupts
  - Halt the machine

- How can this be achieved?
  - Interpretation?

Dual-Mode Execution

- Separate execution into privileged (supervisor) and unprivileged (user) mode(s)
- Applications run in unprivileged mode
  - Some instructions cannot be executed in this mode

- Hardware support
  - Switch from privileged mode into unprivileged mode through a MSR
    - ARM: CSPR_c
    - x86: IRET instruction
  - How can we switch back?
    - We will see...
x86: “Dual” mode

- Ring 0
  - Operating System Kernel
- Ring 1
  - Operating System Services
- Ring 2
  - Extensions
- Ring 3
  - User Applications

Kernel

- Runs in supervisor mode (kernel mode)
- Implicitly trusted
- Central authority in the system
  - Does resource management, protection, access control, etc.
  - Maintains and enforces policies
- Responsible for initiating “user-level applications”
  - Processes

Processes

- Unit of management for a user program instance
  - Kernel can assign permissions per process
  - Kernel can manage resources per process
- Process gets a private region of memory
- Process is assigned to a hardware execution context
  - Register state, PC, ...
- Protection through boxing
  - Processes are isolated from each other
  - Processes execute in user mode

Dual-Mode Execution

- In the simplest form:
System Calls

- User mode processes can call functionality of the kernel through syscalls
  - Idea: kernel does what user processes are not allowed to do
- Kernel can decide whether or not the operation is permitted
  - Access control list
  - Resource accounting
  - Play referee

System Call Example

```c
#include <stdio.h>
void foo(void) {
    printf("Hello world");
}
```

#include: libc
Kernel

Kernel Mode

- `__syscall(SYS_write, ...)`
- `write()`

User Mode

To the user program, it appears as a function call executed under program control

System Calls on ARM

```
00004710 <syscall>
  14710: e1abc00d  mov   r10, sp
  14714: e92d00f0  push  r4, r5, r6, r7
  14718: e1a00001  mov   r0, r1
  14720: e1a01002  mov   r2, r3
  14724: e1a02003  mov   r3, r4
  14728: e89e0078  ldm   ip, {r3, r4, r5, r6}
  14730: e8bd00f0  pop   r4, r5, r6, r7
  14734: 312fff1e  bxcc  lr
  1473c: ea0005ef  b    #15F00
```

- SWI on older ARM

User program executes a trap instruction (system call)

Hardware

- Sets the PC to the OS system call handler.
- saves a small amount of state in Special Purpose Registers (SPRs)
- sets UM “off” in the MSR
- loads/executes the instruction at PC

Operating system

- identifies the requested operation and parameters (e.g. open(filename, O_RDONLY))
- executes the requested operation
- restores user’s state (including registers)
- replaces a register value with indicator of success/failure
- executes an RFI instruction to return to the user program

User program receives the result and continues
Operating System

Interrupts

- Devices can interrupt the execution
- Kernel provides interrupt handlers which perform a task and then switch back to the preempted process.

- What happens if there is an interrupt during handling an interrupt?
  - Don’t want this to happen
  - “Disable” interrupts during handling
    - Actually, mask them = defer. Interrupts need to be handled.

Interrupts

- Hardware
  - (sets the PC to) the operating system at a pre-specified location
  - saves a small amount of state in Special Purpose Regs (SPRs)
  - sets UM “off” in the MSR
  - loads the next instruction

- Operating system
  - saves state of the user process from Regs/SPRs
  - identifies the device and/or cause of interrupt
  - responds to (processes) the interrupt
  - restores state of the user program (if applicable) or some other user program
  - executes an RFI instruction to return to the user program
  - (RFI changes PC, sets UM “on” in the MSR)

- User program continues exactly at the same point it was interrupted.

Exceptions

- Program code causes an unexpected condition

- Hardware stops current execution and start running an exception handler registered by the kernel

- What could cause an exception?
  - Trying to execute a privileged instruction from user mode
  - Process writing outside of its memory region
  - Division by zero
  - Some CPUs: Reading from a non-aligned memory address
  - Breakpoints for debugging
Exceptions

- **Hardware**
  - (sets the PC to) the operating system at a pre-specified location ("Exception" handler)
  - saves a small amount of state in SPRs
  - sets UM "off" in the MSR
  - loads the next instruction

- **Operating system**
  - identifies the cause of the exception (e.g. divide by 0)
  - If user process has exception handling (signals) specified for the specific exception
    - then OS adjusts the user program state so that it calls its handler
    - Execute an RFI instruction (with PC to user's exception handler)
    - else OS kills it and runs some other user program, as available

Effects of exceptions can be visible to user programs and cause abnormal execution flow

Returning from Kernel to User Mode

- **Starting a new process**
  - After bootup, Unix systems start an initial process called *init*
  - All further processes become (direct or indirect) children of *init*

- **Resume from exception, interrupt, or system call**

- **Switch to different process**
  - Scheduler

- **User-level upcall**
  - Signals
  - Callback

Device Driver

- **Glue between OS and some piece of hardware**
  - Mediates between generic APIs and specific commands to the HW

- **What are drivers good for?**
  - Abstraction
  - Encapsulation, Modularity
  - Some form of isolation?
  - Causing you pain by crashing your kernel 😊

Kernel Architecture

- **Monolithic**
  - Drivers are statically linked into the kernel
  - One monolithic binary
Kernel Architecture

- Modular
  - Linux
    - `lsmod`, `modprobe`

Kernel Driver

- Microkernel
  - Minix, L4, Mach, ...
  - (Hybrid approach: FUSE in Linux, Windows can run some drivers in user space)