Beyond 224: Operating Systems

CMPU 224 – Computer Organization
Jason Waterman
What is Operating Systems

• Operating Systems
  • The system software that works together with the hardware to run application programs
Operating System (OS)

• Responsible for
  • Making it easy to run programs
  • Allowing programs to share memory and other resources
  • Enabling programs to interact with devices

The OS is in charge of making sure the system operates **correctly and efficiently**
The OS as a resource manager

• The OS **manage resources** such as *CPU, memory and disk*

• The OS allows
  
  • Many programs to run → Sharing the **CPU**
  • Many programs to *concurrently* access their own instructions and data → Sharing **memory**
  • Many programs to access devices → Sharing **disks**
Virtualization

• The OS takes a physical resource and transforms it into a virtual form of itself
  • Physical resource: Processor, Memory, Disk ...
  • The virtual form is more general, powerful and easy-to-use
  • Sometimes, we refer to the OS as a virtual machine for programs
Virtualizing the CPU

• The system has a very large number of virtual CPUs
  • Turning a single CPU into a seemingly infinite number of CPUs
  • Allowing many programs to seemingly run at once → Virtualizing the CPU
How to Efficiently Virtualize the CPU with Control?

• The OS needs to share the physical CPU by **time sharing**
  • **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  • **Control**: How can we run processes efficiently while retaining control over the CPU?

• Idea: let the process run directly on the CPU for general computation
  • If a process wishes to perform some kind of restricted operation, ask the OS

• Solution: Using protected control transfer
  • **User mode**: Applications do not have full access to hardware resources
  • **Kernel mode**: The OS has access to the full resources of the machine
System Call

• Allow the kernel to **carefully expose** certain **key pieces of functionality** to user program, such as ...
  • Accessing the file system
  • Creating and destroying processes
  • Communicating with other processes
  • Allocating more memory
• **Trap** instruction
  • Jump into the kernel
  • Raise the privilege level to kernel mode
• **Return-from-trap** instruction
  • Return into the calling user program
  • Reduce the privilege level back to user mode
How does the OS take control

• How can the OS **regain control** of the CPU so that it can switch between **processes**?

• A **timer interrupt**
  • During the boot sequence, the OS start the **timer**
  • The timer **raise an interrupt** every so many milliseconds
  • When the interrupt is raised:
    • The currently running process is halted
    • Save enough of the state of the program
    • A pre-configured interrupt handler in the OS runs

A **timer interrupt** gives OS the ability to run again on a CPU
Saving and Restoring Context

• **Scheduler** makes a decision:
  • Whether to continue running the **current process**, or switch to a **different one**
  • If the decision is made to switch, the OS executes a **context switch**
Context Switch

• A low-level piece of assembly code
  • **Save a few register values** for the current process onto its kernel stack
    • General purpose registers
    • PC
    • Kernel stack pointer
  • **Restore a few register values** for the soon-to-be-executing process from its kernel stack
  • **Switch to the kernel stack** for the soon-to-be-executing process
Worried About Concurrency?

• What happens if, during interrupt or trap handling, another interrupt occurs?

• OS handles these situations:
  • Disable interrupts during interrupt processing
  • Use a number of sophisticated locking schemes to protect concurrent access to internal data structures
Virtualizing Memory

• The physical memory is *an array of bytes*
• A program keeps all of its data structures in memory
  • **Read memory** (load):
    • Specify an **address** to be able to access the data
  • **Write memory** (store):
    • Specify the data to be written to the given address
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "common.h"

// To compile: gcc -Og -Wall -no-pie -o mem mem.c -lpthread

int value;

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: mem <value>\n");
        exit(1);
    }

    printf("\n(pid:%d) addr of value: %p
", (int) getpid(), &value);

    value = atoi(argv[1]); // assign value to addr stored in p
    while (1) {
        Spin(1);
        value = value + 1;
        printf("(pid:%d) address of p: %p, value of p: %d\n", getpid(), &value, value);
    }

    return 0;
}
Virtualizing Memory (Cont.)

- Each process accesses its own private **virtual address space**
- The OS maps a process’s **address space** onto the **physical memory**
- A memory reference within one running program **does not affect** the address space of other processes
- Physical memory is a **shared resource**, managed by the OS

```
prompt> ./mem 100 & ./mem 200 &
[1] 204268
[2] 204269

(pid:204268) address of p: 0x40408c, value of p: 101
(pid:204269) address of p: 0x40408c, value of p: 201
(pid:204269) address of p: 0x40408c, value of p: 202
(pid:204268) address of p: 0x40408c, value of p: 102
(pid:204269) address of p: 0x40408c, value of p: 103
(pid:204269) address of p: 0x40408c, value of p: 203
(pid:204268) address of p: 0x40408c, value of p: 104
(pid:204269) address of p: 0x40408c, value of p: 204
...
```
Memory Virtualization Overview

• Just like the CPU, memory is virtualized
• Every address generated by a user program is a virtual address
• Virtual memory provides the illusion of a large private address space to programs
  • For instructions and data
• The OS, with help from the hardware, translates each virtual memory address to an actual physical address
  • Protects and isolates processes from each other
  • And from the kernel
Memory Virtualizing with Efficiency and Control

- Memory virtualizing takes a similar strategy to virtualizing the CPU with limited direct execution (LDE)
  - Must have efficiency and control
- In memory virtualizing, efficiency and control are attained by **hardware support**
  - E.g., registers, TLB (Translation Look-aside Buffer), page-table
  - Control means no process is allowed to access any memory but its own
- Desire flexibility
  - Applications should be allowed to use their address space however they would like
- **How do we efficiently and flexibly virtualize memory?**
Address Translation

• Mechanism: **hardware-based address translation**
  • Hardware transforms a **virtual address** to a **physical address**
  • The desired code/data is actually stored in a physical address

• The OS must get involved at key points to set up the hardware
  • The OS manages memory
  • Keeps track of which locations are free and which are in use

• We’ll go over the details in 334
Brief History of Operating Systems

Phase 1: Hardware expensive, humans Cheap

• One user at a time
• First “OS”: I/O subroutines shared by users
• Simple batch monitor
  • Load and run user jobs
• Timesharing – several users share the system
Brief History of Operating Systems
Phase 2: Hardware cheap, humans expensive

• PC era: everyone one gets their own computers
  • Are all the fancy features developed for timesharing still needed?

• Future of OSes:
  • Very small (embedded devices)
  • Very large (datacenters, cloud computing)

• Current OSes:
  • Huge: millions of lines of code
  • Complex: asynchronous, quirky hardware, performance is key
  • Poorly understood

• Most OS functions fall in the category of coordination
  • Concurrency, I/O devices, Memory, Files, Networks, Security
Bonus: Field Effect Transistor

Diagram of a Field Effect Transistor (FET): Source, Gate, Drain, Gate Oxide, n+, p, Body.