History of Operating Systems and Dual Mode Execution

CS439: Principles of Computer Systems
September 6, 2017
Last Time

Operating Systems are:

- Referees:
  - Manage shared resources
  - Provide protection and communication for processes

- Illusionists:
  - Provide the illusion of infinite resources

- Glue:
  - Provide standard services which the hardware implements
Today’s Agenda

• History of Operating Systems
  – Batch systems, Asynchronous I/O, Time Slicing

• Dual Mode Execution
  – User vs. Kernel modes
History of Operating Systems
Overview

• Phase 1: Hardware expensive, humans cheap
• Phase 2: Hardware cheap, humans expensive
• Phase 3: Hardware very cheap, humans very expensive
Phase 1: Expensive Hardware, Cheap Humans

1. One user on the console, one process at a time (1945-1955)
   - Single user system
   - OS is a subroutine library (and a loader)
     • A stack of cards you pull off the shelf to do, for example, a matrix multiply
   - Problem: Low utilization of expensive components
Phase 1: Expensive Hardware, Cheap Humans

2. Batch processing: load program, run, output to tape, print results, repeat (1955-1965)
   - Users give their program (on punch cards) to a human who schedules the jobs in batches
   - Each batch is input into a card reader and output on tape
   - Tape is carried to CPU machine
   - OS on CPU machine loads each job in the bath from tape, runs, and writes any output to another tape
   - Advantage: next job can be loaded immediately as previous one finishes
     • Better use of hardware but debugging is much more difficult
   - Disadvantages:
     • No protection---a buggy program can crash the batch monitor
     • Computer is idle during I/O
Batch Processing

An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.
Batch Processing (1955-1965)

**Before.** Each action was done by the main computer.

- **Card Reader:** Read Job 1
  - **CPU:** Execute Job 1
  - **Printer:** Print Job 1
  - **3 jobs complete!**

**After.** Card reader and printer are separate machines from the expensive CPU.

- **Card Reader:** Read Batch 1
  - **CPU:** Execute Batch 1
  - **Printer:** Print Batch 1
  - **6 jobs complete!**

Each batch represents two jobs.
Phase 1:
Expensive Hardware, Cheap Humans

3. Overlap of I/O and computation, interrupts
   - OS requests I/O, goes back to computation, gets interrupt when I/O device finishes
   - No sharing, only protect OS from applications
   - Add concurrency *within same process*
   - Buffering and interrupt handling in OS
   - Spool jobs on the drum
   - Performance improves because I/O and processing happen concurrently
Overlapping I/O and Computation

Program | OS | I/O Device
----- | ---- | -------
main{
    
    \[ k: \text{read}() \]
    \[
    \text{read}\{
        \text{startIO()}
        \text{waitIO()}
    \}
    \]
    \[
    \text{endo}()\]
    \[
}

User Program

“System Software”

Operating System

interrupt
Overlapping I/O and Computation

User Program
“System Software”
Operating System

Program

```
main{
  k: read()
  read{
    startIO()
  }
  endio()
  k+1:
}
```

OS

I/O Device

interrupt
Phase 1: Expensive Hardware, Cheap Humans

4. Multiprogramming: several programs run at the same time sharing the machine
   – One job runs until it performs I/O, then another job gets the CPU
   – OS manages interactions between concurrent programs (which ones start and execute, provides protection)
   – Requires: Memory protection and relocation
Multiprogramming (1965-1980)
Keep several jobs in memory and multiplex CPU between jobs

```
Program          OS           I/O Device

main{
    ...
    k: read() \rightarrow read{
        ...
        startIO()
        ...
    }
    endio()
    ...
    k+1: ...
    }
```
Multiprogramming (1965-1980)

Keep several jobs in memory and multiplex CPU between jobs

```
User Program n
.
.
User Program 2
User Program 1
"System Software"
Operating System

Program 1  OS  Program 2

main{

k: read()  read{

startIO()

schedule()  main{

endio{

}

schedule()  }

}

}

k+1:

interrupt
```
iClicker Question

In batch systems, each job must completely finish before the next job may begin.

A. True
B. False
5. Interactive timesharing (1970-)

- Use cheap terminals to let multiple users interact with the system at the same time
  - Debugging is a lot easier
  - Process switching occurs much more frequently
- Requires: more sharing, more protection, more concurrency
- New OS services: shell, file system, rapid process switching (users can interact!), virtual memory (processes running simultaneously!)
- New problems: response time, thrashing
Timesharing (1970- )

A timer interrupt is used to multiplex CPU among jobs

User Program n
...
User Program 2
User Program 1
“System Software”
Operating System

Program 1

main{

k:

}
Phase 3: Very Cheap Hardware, Very Expensive Humans

6. Personal computing
   - Computers are cheap, so give everyone a computer
   - Simplify OS by eliminating multiprogramming, concurrency, and protection
     • A subroutine library again! (MSDOS, MacOS)
     • Failed: humans are expensive, so don’t waste their time letting programs crash each other!
Phase 3: Very Cheap Hardware, Very Expensive Humans

7. Parallel and Distributed Computing
   – Computers are SO cheap, give people a bunch of them!
   – In parallel systems, multiple processors are in the same machine, sharing memory, I/O devices, ...
   – In distributed systems, multiple processors communicate via a network
   – Advantages: increased performance, increased reliability, sharing of specialized resources
Genealogy of Modern Operating Systems

- MSDOS (70’s)
  - Windows (80’s)
    - Windows Mobile
    - Windows 7 (2010)
  - Windows NT (90’s)
    - Windows NT (90’s)
- VMS (70’s)
- MVS (60’s)
- UNIX (70’s)
  - BSD UNIX (80’s)
  - Free BSD
  - LINUX (90’s–today)
    - Android
- Mach (80’s)
- Multics (60’s)
- VMware
- NEXT
  - Mac OS
  - Mac OSX
- iOS
Operating System Complexity

From MIT’s 6.033 course
(I took it from John Kubiatowicz’s CS162 course at Berkeley)
Dual Mode Execution
(and some Processes)
OS Interfaces

Last time, we saw that the OS has three interfaces

– Abstract Machine Interface (AMI)
  • between OS and apps: API + memory access model + legally executable instructions

– Application Programming Interface (API)
  • function calls provided to apps

– Hardware Abstraction Layer (HAL)
  • abstracts hardware *internally to the OS*

*Why?*
Logical OS Structure

AMI/API

Applications

OS

HAL

Applications

Quake

Sql Server

System Utils

Shells

Windowing & graphics

Networking

CPU Scheduling

Virtual Memory

Access Control

File System

Process Management

Device Drivers

Hardware-specific software

Disks, Cache, Physical Memory, TLB, Hardware Devices
If Applications Had Free Rein...

Buggy or malicious applications could:

– crash other applications
– violate privacy of other applications
– hog all the resources
– change the OS
– crash the OS

We would be trusting every software developer!
The Process: Boxes in the Application

- An abstraction for protection
  - Represents an application program executing with restricted rights

- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware
The Process: Boxes in the Application

- An abstraction for protection
  - Represents an application program executing with restricted rights
- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware
  - Must still allow safe communication
What is a Process?

• A process is a program during execution.
  – Program = static file (image)
  – Process = executing program = program + execution state

• A process is the basic unit of execution in an OS

• Different processes may run different instances of the same program
  – e.g., my gcc and your gcc process both run the GNU C compiler

• At a minimum, process execution requires following resources:
  – Memory to contain the program code and data
  – A set of CPU registers to support execution
How can the OS enforce restricted rights?

• Easy: OS interprets each instruction!
  – Good solution?
    • No! Slow
  – Most instructions are safe: can we just run them in hardware?

• *Dual Mode Execution*
  – User mode: access is restricted
  – Kernel mode: access is unrestricted
  – Supported by the hardware
    • Mode is indicated by a bit in the process status register
Kernel vs. User Mode: Privileged Instructions

User processes may not:
• address I/O directly
• use instructions that manipulate OS memory (e.g., page tables)
• set the mode bits that determine user or kernel mode
• disable and enable interrupts
• halt the machine

But in kernel mode, the OS does all these things.

Executing a privileged instruction while in user mode causes a processor exception...

...which passes control to the kernel
Transitioning from User Mode to Kernel Mode...

- Often called *entering the kernel*
- Three methods:
  - Exceptions
    - user program acts silly (e.g. division by zero)
    - or attempts to perform a privileged instruction
      - sometimes on purpose! (breakpoints)
    - synchronous (related to instruction that just executed)
  - Interrupts (asynchronous exceptions)
    - something interrupts the currently executing process
      - timer, HW device requires OS service, ...
    - asynchronous (not related to instruction that just executed)
  - System calls/Traps
    - user program requests OS service
    - looks like a function call
    - synchronous
User Mode to Kernel Mode: Details

- OS saves state of user program
- Hardware identifies why boundary is crossed
  - system call?
  - interrupt? then which hardware device?
  - which exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked
Saving the State of the Interrupted Process

• Privileged hw register points to Exception Stack
  – on switch, hw pushes some of interrupted process registers (SP, PC, etc) on exception stack before handler runs. Why?
  – then handler pushes the rest (pushad on x86)
  – On return, do the reverse (popad on x86)

• Why not use user-level stack?
  – reliability: even if user’s stack points to invalid address, handlers continue to work
  – security: kernel state should not be stored in user space (or could be read/written)

• One interrupt stack per processor/process/thread
System Calls

- A request by a user-level process to call a function in the kernel is a \textit{system call}
  - Examples: read(), write(), exit()

- The interface between the application and the operating system (API)
  - Mostly accessed through \textit{system-level libraries}

- Parameters passed according to calling convention
  - registers, stack, etc
System Calls: A Closer Look

(user process)

- user process executing
- calls system call
- return from system call

(mode bit = 1)

(trap)

(kernel)

- execute system call

(mode bit = 0)

(mode bit := 0)

(return)

(mode bit := 1)
System Calls: A Closer Look

• User process executes a trap instruction
• Hardware calls the OS at the system-call handler, a pre-specified location
• OS then:
  – identifies the required service and parameters (e.g. open(filename, O_RDONLY))
  – executes the required service
  – sets a register to contain the result of call
  – Executes an RTI instruction to return to the user program
• User program receives the result and continues
iClicker Question

The interrupt vector is used to determine the action taken by the OS when:

A. An exception occurs
B. An interrupt occurs
C. A system call is executed
D. All of the above
E. None of the above
Switching Back!

• From an interrupt, just reverse all steps!
  – asynchronous, so not related to executing instruction

• From exception and system call, increment PC on return
  – synchronous, so you want to execute the next instruction, not the same one again!
  – on exception, handler changes PC at the base of the stack
  – on system call, increment is done by the hardware
Dual Mode Execution: One Piece of the Protection Pie

For efficient protection, the hardware must support at least three features:

• Privileged instructions
  – Instructions only available in kernel mode
  – In user mode, no way to execute potentially unsafe instructions
  – Prevents user processes from, for instance, halting the machine
  – Implementation: mode status bit in the process status register

• Timer interrupts
  – Kernel must be able to periodically regain control from running process
  – Prevents process from gaining control of the CPU and never releasing it
  – Implementation: hardware timer can be set to expire after a delay and pass control back to the kernel

• Memory protection
  – In user mode, memory accesses outside a process’ memory region are prohibited
  – Prevents unauthorized access of data
  – Implementation: We’ll return to this later in the course
Summary

• Application of ideas has changed over time
  – Every design decision has driven change in other designs

• Operating System provides protection through dual-mode execution
  – Mode changes through interrupts (e.g., time slice), exceptions, or system calls.
  – A status bit in a protected processor register indicates the mode
  – Privileged instructions can only be executed in kernel mode
Announcements

• TAs, contact information, etc. are in the online syllabus
• Office hours are posted (and there are pictures!)
  – Mine will begin late today
• Discussion sections begin this week! (Remember, you must attend your own!)
• Problem Set 1 is up---complete and take answers with you to section on Friday
• Next time: Processes!