Architectural Support for Operating Systems

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
Anne Bracy

See: Chapter 2 in OSPP textbook
What is an operating system?

Software to manage hardware resources

Operating System

Applications (Maps, Siri, Safari, ...)

Hardware (CPU, RAM, Modem, ...)

Virtual Machine Interface

Physical Machine Interface

What hw is needed to help the OS do its job?
What is a Process?

…an instance of a program

Final Run Step: OS sets PC to program’s first insn
Protect who from what?

What could possibly go wrong?

- Stack
- Heap
- Data
- Insns

OS
Kernel

someone's first C program in 3410

US military defense software

4
Privilege Levels

Some processor functionality cannot be made accessible to untrusted user applications

Operating System = the mediator between untrusted/untrusting apps

Need to differentiate untrusted apps and OS code

Solution: “privilege mode” bit in the processor

0 = Untrusted = user  1 = Trusted = OS
Privileged Instructions

• changing the privilege mode
• writing to certain registers (page table base register)
• enabling a co-processor
• changing memory access permissions
• manipulate device settings
• signal other users’ processes
• print character to screen
• send a packet on the network
• allocate a new page in memory

CPU knows which instructions are privileged.

opcode == privileged && mode == 0 \rightarrow Exception!
Context Switch/Mode Transfer

Hardware transfer to kernel:
1. save privilege mode, set mode to 1
2. mask interrupts  (see slide 14)
3. save: SP, PC, eflags register (x86)
4. switches SP to the kernel stack
5. save values from #2 onto kernel stack
6. save error code
7. set PC to the interrupt vector table

Interrupt handler
1. saves all registers
2. examines the cause
3. performs operation required
4. restores all registers

Performs “Return from Interrupt” insn (maybe)
• restores the privilege mode, SP and PC
Process Control Block (PCB)

For each process, the OS has a PCB containing:

- location in memory
- location of executable on disk
- which user is executing this process
- process privilege level
- process arguments
- register values
- PC, SP, eflags

... and more!

Usually lives on the kernel stack
**Privileged Instructions**

- changing the privilege mode
- writing to certain registers (page table base register)
- enabling a co-processor
- changing memory access permissions
- manipulate device settings
- signal other users’ processes
- print character to screen
- send a packet on the network
- allocate a new page in memory

CPU knows which instructions are privileged.

opcode == privileged && mode == 0 ➔ *Exception!*

_is this really so bad?_
**System Call** *(slight variation on standard context switch)*

Hardware transfer to kernel:
1. save privilege mode, set mode to 1
2. mask interrupts *(see slide 14)*
3. save: SP, PC, eflags register (x86)
4. switches SP to the kernel stack
5. save values from #2 onto kernel stack
6. save error code ← sys call not an error
7. set PC to the interrupt vector table

Interrupt handler
1. saves all registers ← callee only
2. examines the cause ← which syscall was called,
3. performs operation required might have arguments
4. restores all registers ← callee only

Performs “Return from Interrupt” insn (maybe)
• restores the privilege mode, SP and PC

Which insn?
Let’s Start at the Very Beginning…

In the beginning… (when the system starts up)
• privilege mode set to 1
• PC contains address of boot code
• boot code loads kernel into memory
• kernel does some setup (devices, initializes MMU, creates interrupt vector table, etc.)
• picks an application, loads it
• resets privilege bit
• changes PC to starting instruction of the chosen application

Now what?
How does the OS re-take control?
Interrupts

Timer Interrupts:
Process interrupted after certain period
(number of instructions executed or time passed)

More Generally: **Hardware Interrupts**
External Event has happened.
OS needs to check it out.
Process stops what it’s doing, invokes OS,
which handles the interrupt.
Interrupt Management

Interrupt controllers manage interrupts

Interrupts have descriptor of the interrupting device
Priority selector circuit examines all interrupting devices, reports highest level to the CPU

Interrupt controller implements interrupt priorities
Can optionally remap priority levels
Masking Interrupts

**Maskable interrupts:** can be turned off by the CPU for critical processing (misnomer: *delayed*)

**Nonmaskable interrupts:** signifies serious errors (e.g., unrecoverable memory error, power out warning, etc)

*Why would we want to mask interrupts?*  
(“discuss later” on slide 7)
Three ways for the OS to be invoked

1. Hardware interrupt
   - *some other entity trying to get CPU’s attention*
   - Asynchronous = caused by an external event
   - Examples: keystroke, arrival of a packet from network

2. System Call
   - *process needs help from the OS*
   - Intentional, Synchronous = caused by the syscall insn
   - Examples: open, write, fork, exit

3. Exception
   - *something went wrong*
   - Unintentional, Synchronous = caused by executing insn
   - Examples: privileged insn in user mode, page fault

Terminology Chaos.

are we done yet?
Uniprogramming

No Translation or Protection

Application:
• Only one application at a time
• Always runs at same place in physical memory
• Can access any physical address
• Illusion of dedicated machine achieved by reality of a dedicated machine
Multiprogramming, V1

No Translation
Adjust addresses (ld, st, jmps) when program loaded into memory

- Everything adjusted to memory location of program
- “Translation” by Linker/Loader
- Common in early days

No protection
Any process can crash another (or the OS!)
/*
 * Corresponds to Figure 2.7 in the textbook
*/

#include <stdio.h>
#include <unistd.h>

int globalVar = 0;    // a static variable

int main() {
    int localVar = 7;

    globalVar += 1;

    // sleep causes the program to wait for x seconds
    sleep(5);
    printf ("Loc Var: Addr: %p; Val: %d\n", &localVar, localVar);
    printf ("Gl Var: Addr: %p; Val: %d\n", &globalVar, globalVar);
    printf ("Location of Main: Address: \t%p\n", &main);
}

“When multiple copies of this program simultaneously, the output does not change.”
Multiprogramming, V1++

Add Protection

• Two registers (base and limit) keep user inside designated area
• Access illegal address → error
• During context switch, kernel saves/loads base/limit from PCB
• User not allowed to change base/limit registers
Dissatisfied?

Why is this a shame?

Don’t worry, that’s not the final version of how processors provide memory protection.
Minimum Hardware Requirements

• Privileged Instructions
• Timer Interrupts
• Memory Protection