CSE 120 Principles of Operating Systems

Fall 2016

Lecture 2: Architectural Support for Operating Systems

Geoffrey M. Voelker

Administrivia

- Project 0
  - Due 10/4, done individually
- Homework #1
  - Due 10/6
- Project groups
  - Fill out Google form
  - Just need one per group
  - Fill out even if you are working alone

Why Start With Architecture?

- Operating system functionality fundamentally depends upon the architectural features of the computer
  - Key goals of an OS are to enforce protection and resource sharing
  - If done well, applications can be oblivious to HW details
  - Unfortunately for us, the OS is left holding the bag
- Architectural support can greatly simplify – or complicate – OS tasks
  - Early PC operating systems (DOS, MacOS) lacked virtual memory in part because the architecture did not support it
  - Early Sun 1 computers used two M68000 CPUs to implement virtual memory (M68000 did not have VM hardware support)

Architectural Features for OS

- Features that directly support the OS include
  - Protection (kernel/user mode)
  - Protected instructions
  - Memory protection
  - System calls
  - Interrupts and exceptions
  - Timer (clock)
  - I/O control and operation
  - Synchronization

Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Mechanisms to handle concurrency
  - Interrupts, atomic instructions

Protected Instructions

- A subset of instructions of every CPU is restricted to use only by the OS
  - Known as protected (privileged) instructions
- Only the operating system can …
  - Directly access I/O devices (disks, printers, etc.)
    - » Security, fairness (why?)
  - Manipulate memory management state
    - » Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - » Kernel mode, interrupt level
  - Halt instruction (why?)

INVLPG—Invalidate TLB Entries

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01/7	INVLPG *m*	M	Valid	Valid	Invalidate TLB entries for page containing *m*.

NOTES:

* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.[1]

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidates only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" and Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different

OS Protection

- How do we know if we can execute a protected instruction?
    - Architecture must support (at least) two modes of operation: kernel mode and user mode
        - » VAX, x86 support four modes; earlier archs (Multics) even more
        - » Why? Protect the OS from itself (software engineering)
    - Mode is indicated by a status bit in a protected control register
    - User programs execute in user mode
    - OS executes in kernel, privileged mode (OS == "kernel")
- Protected instructions only execute in kernel mode
    - CPU checks mode bit when protected instruction executes
    - Setting mode bit must be a protected instruction
    - Attempts to execute in user mode are detected and prevented

Memory Protection

- OS must be able to protect programs from each other
- OS must protect itself from user programs
- May or may not protect user programs from OS
  - Raises question of whether programs should trust the OS
  - Untrusted operating systems? (Intel SGX)
- Memory management hardware provides memory protection mechanisms
  - Base and limit registers
  - Page table pointers, page protection, segmentation, TLB
- Manipulating memory management hardware uses protected (privileged) operations

Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both
- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine
- Once the system is booted, all entry to the kernel occurs as the result of an event
  - In effect, the operating system is one big event handler

Categorizing Events

- Two kinds of events, interrupts and exceptions
- Exceptions are caused by executing instructions
  - CPU requires software intervention to handle a fault or trap
- Interrupts are caused by an external event
  - Device finishes I/O, timer expires, etc.

- Two reasons for events, unexpected and deliberate
- Unexpected events are, well, unexpected
  - What is an example?
- Deliberate events are scheduled by OS or application
  - Why would this be useful?

Categorizing Events (2)

- This gives us a convenient table:

	Unexpected	Deliberate
Exceptions (sync)	fault	syscall trap
Interrupts (async)	interrupt	software interrupt

  - Terms may be used slightly differently by various OSes, CPU architectures…
  - Software interrupt – a.k.a. async system trap (AST), async or deferred procedure call (APC or DPC)
- Will cover faults, system calls, and interrupts next

Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, unaligned access, divide by zero
- Upon exception, hardware “faults” (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
- Modern OSes use VM faults for many functions
  - Debugging, end-of-stack, garbage collection, copy-on-write
- Fault exceptions are a performance optimization
  - Could detect faults by inserting extra instructions into code (at a significant performance penalty)

Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
    - Page faults cause the OS to place the missing page into memory
    - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault
- Some faults are handled by notifying the process
    - Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
    - Handler must be registered with OS
    - Unix signals or Win user-mode Async Procedure Calls (APCs)
        - » SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.

Handling Faults (2)

- The kernel may handle unrecoverable faults by killing the user process
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)
- What about faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows “Blue screen of death”
    - » Kernel is halted, state dumped to a core file, machine locked up

System Calls

- For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or protected procedure call, or protected control transfer
- CPU ISA provides a system call instruction that:
  - Causes an exception, which vectors to a kernel handler
  - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - » Why save mode?
  - Returning from system call restores this state
- Requires architectural support to:
  - Verify input parameters (e.g., valid addresses for buffers)
  - Restore saved state, reset mode, resume execution

INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT 3	NP	Valid	Valid	Interrupt 3—trap to debugger.
CD *ib*	INT *imm8*	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	NP	Invalid	Valid	Interrupt 4—if overflow flag is 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	imm8	NA	NA	NA

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

A6.7.136 SVC (formerly SWI)

Generates a supervisor call. See *Exceptions* in the *ARM Architecture Reference Manual*.

Use it as a call to an operating system to provide a service.

Encoding T1 All versions of the Thumb ISA.

```
SVC<c> #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

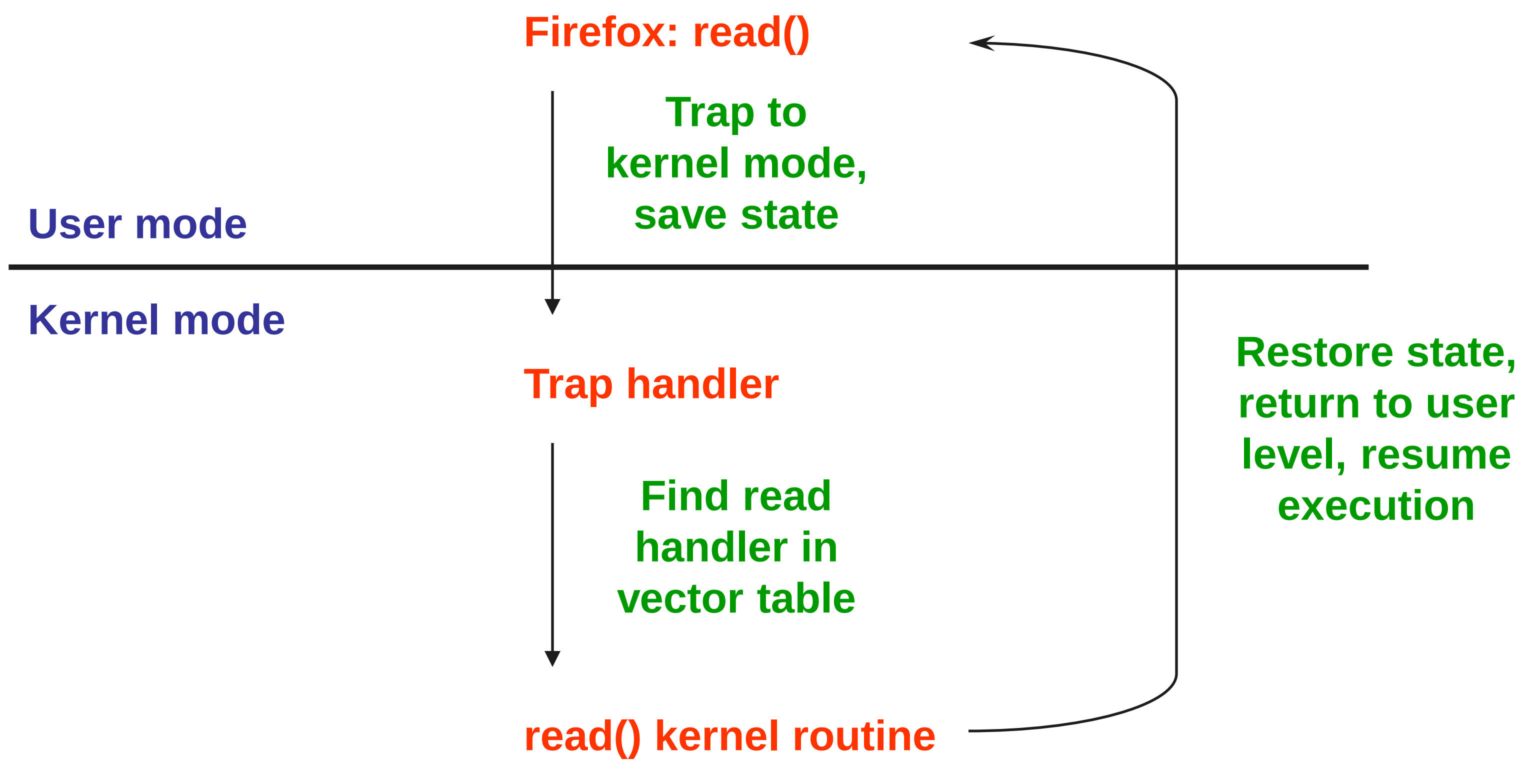
```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

Nachos (test/start.s)

```
/* --------------------------------------------------------------
 * System call stubs:
 *      Assembly language assist to make system calls to the Nachos kernel.
 *      There is one stub per system call, that places the code for the
 *      system call into register r2, and leaves the arguments to the
 *      system call alone (in other words, arg1 is in r4, arg2 is
 *      in r5, arg3 is in r6, arg4 is in r7)
 *
 *      The return value is in r2. This follows the standard C calling
 *      convention on the MIPS.
 * --------------------------------------------------------------
 */

#define SYSCALLSTUB(name, number) \
        .globl  name            ; \
        .ent    name            ; \
name:                           ; \
        addiu   $2,$0,number    ; \
        syscall                 ; \
        j       $31             ; \
        .end    name
```

System Call


Firefox: read()
Trap to kernel mode, save state
User mode
Kernel mode
Trap handler
Find read handler in vector table
read() kernel routine
Restore state, return to user level, resume execution

LINUX System Call Quick Reference

Jialong He
Jialong_he@bigfoot.com
http://www.bigfoot.com/~jialong_he

Introduction

System call is the services provided by Linux kernel. In C programming, it often uses functions defined in **libc** which provides a wrapper for many system calls. Manual page section 2 provides more information about system calls. To get an overview, use "man 2 intro" in a command shell.

It is also possible to invoke **syscall()** function directly. Each system call has a function number defined in **<syscall.h>** or **<unistd.h>**. Internally, system call is invokded by software interrupt 0x80 to transfer control to the kernel. System call table is defined in Linux kernel source file "**arch/i386/kernel/entry.S**".

System Call Example

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {

        long ID1, ID2;
        /*-----------------------------*/
        /* direct system call          */
        /* SYS_getpid (func no. is 20) */
        /*-----------------------------*/
        ID1 = syscall(SYS_getpid);
        printf ("syscall(SYS_getpid)=%ld\n", ID1);

        /*-----------------------------*/
        /* "libc" wrapped system call  */
        /* SYS_getpid (Func No. is 20) */
        /*-----------------------------*/
        ID2 = getpid();
        printf ("getpid()=%ld\n", ID2);

        return(0);
}
```

System Call Quick Reference

No	Func Name	Description	Source
1	**exit**	terminate the current process	*kernel/exit.c*
2	**fork**	create a child process	*arch/i386/kernel/process.c*
3	**read**	read from a file descriptor	*fs/read_write.c*
4	**write**	write to a file descriptor	*fs/read_write.c*
5	**open**	open a file or device	*fs/open.c*
6	**close**	close a file descriptor	*fs/open.c*
7	**waitpid**	wait for process termination	*kernel/exit.c*
8	**creat**	create a file or device ("man 2 open" for information)	*fs/open.c*
9	**link**	make a new name for a file	*fs/namei.c*
10	**unlink**	delete a name and possibly the file it refers to	*fs/namei.c*
11	**execve**	execute program	*arch/i386/kernel/process.c*
12	**chdir**	change working directory	*fs/open.c*
13	**time**	get time in seconds	*kernel/time.c*
14	**mknod**	create a special or ordinary file	*fs/namei.c*
15	**chmod**	change permissions of a file	*fs/open.c*
16	**lchown**	change ownership of a file	*fs/open.c*
18	**stat**	get file status	*fs/stat.c*
19	**lseek**	reposition read/write file offset	*fs/read_write.c*
20	**getpid**	get process identification	*kernel/sched.c*
21	**mount**	mount filesystems	*fs/super.c*
22	**umount**	unmount filesystems	*fs/super.c*
23	**setuid**	set real user ID	*kernel/sys.c*
24	**getuid**	get real user ID	*kernel/sched.c*
25	**stime**	set system time and date	*kernel/time.c*
26	**ptrace**	allows a parent process to control the execution of a child process	*arch/i386/kernel/ptrace.c*
27	**alarm**	set an alarm clock for delivery of a signal	*kernel/sched.c*
28	**fstat**	get file status	*fs/stat.c*
29	**pause**	suspend process until signal	*arch/i386/kernel/sys_i386.c*
30	**utime**	set file access and modification times	*fs/open.c*
33	**access**	check user's permissions for a file	*fs/open.c*
34	**nice**	change process priority	*kernel/sched.c*
36	**sync**	update the super block	*fs/buffer.c*
37	**kill**	send signal to a process	*kernel/signal.c*
38	**rename**	change the name or location of a file	*fs/namei.c*
39	**mkdir**	create a directory	*fs/namei.c*
40	**rmdir**	remove a directory	*fs/namei.c*
41	**dup**	duplicate an open file descriptor	*fs/fcntl.c*
42	**pipe**	create an interprocess channel	*arch/i386/kernel/sys_i386.c*
43	**times**	get process times	*kernel/sys.c*
45	**brk**	change the amount of space allocated for the calling process's data segment	*mm/mmap.c*
46	**setgid**	set real group ID	*kernel/sys.c*
47	**getgid**	get real group ID	*kernel/sched.c*
48	**sys_signal**	ANSI C signal handling	*kernel/signal.c*
49	**geteuid**	get effective user ID	*kernel/sched.c*
50	**getegid**	get effective group ID	*kernel/sched.c*

System Call Questions

- What would happen if the kernel did not save state?
- What if the kernel executes a system call?
- What if a user program returns from a system call?
- How to reference kernel objects as arguments or results to/from system calls?
  - A naming issue
  - Use integer object handles or descriptors
    - E.g., Unix file descriptors, Windows HANDLEs
    - Only meaningful as parameters to other system calls
  - Also called capabilities (more later when we do protection)
  - Why not use kernel addresses to name kernel objects?

Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers
- Two flavors of interrupts
  - Precise: CPU transfers control only on instruction boundaries
  - Imprecise: CPU transfers control in the middle of instruction execution
    - » What the heck does that mean?
  - OS designers like precise interrupts, CPU designers like imprecise interrupts
    - » Why?

Timer

- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
  - When timer expires, generates an interrupt
  - Handled by kernel, which controls resumption context
    - Basis for OS scheduler *(more later…)*
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., *sleep()*)

I/O Control

- I/O issues
  - Initiating an I/O
  - Completing an I/O
- Initiating an I/O
  - Special instructions
  - Memory-mapped I/O
    - » Device registers mapped into address space
    - » Writing to address sends data to I/O device

I/O Completion

- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine
- If you have ever installed early versions of Windows, you now know what IRQs are for

I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads descriptors to find packet in memory)
6. Upon completion, restores saved state from stack

Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the OS be interrupted? (Consider why there might be different IRQ levels)
- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?

Synchronization

- Interrupts cause difficult problems
  - An interrupt can occur at any time
  - A handler can execute that interferes with code that was interrupted
- OS must be able to synchronize concurrent execution
- Need to guarantee that short instruction sequences execute atomically
  - Disable interrupts – turn off interrupts before sequence, execute sequence, turn interrupts back on
  - Special atomic instructions – read/modify/write a memory address, test and conditionally set a bit based upon previous value
    - » XCHG instruction on x86

Summary

- Protection
  - User/kernel modes
  - Protected instructions
- System calls
  - Used by user-level processes to access OS functions
  - Access what is “in” the OS
- Exceptions
  - Unexpected event during execution (e.g., divide by zero)
- Interrupts
  - Timer, I/O

Next Time...

- Read Chapters 4-6 (Processes)
- Homework #1
- Project 0