CSE 120
Principles of Operating Systems

Fall 2022

Lecture 2: Architectural Support for Operating Systems

Geoffrey M. Voelker

Administrivia

- Project 0
  - Due 9/30 11:59pm, done individually
- Homework #1
  - Due 10/6
  - Submit via gradescope (entry code on piazza)
- Project groups
  - We will use a Google form to collect group members
  - Just need one submission per group
  - Fill out even if you are working alone
- Lab hours
  - Will post as a Google calendar

Why Start With Architecture?

- Operating system functionality fundamentally depends upon the architectural features of the computer
  - Key goals 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)

Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers
  - Manage memory protection (e.g., TLB entries)
- Generating and handling "events"
  - System calls, interrupts, exceptions, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Mechanisms to handle concurrency, synchronization
  - 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
    - Protection (why?)

HLT—Halt

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F4	HLT	ZO	Valid	Valid	Halt

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Enter Halt state;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

OS Protection

- How do we know if a protected instruction can execute?
  - Architecture must support (at least) two modes of operation: kernel mode and user mode
    - » VAX, x86 support four modes; earlier architectures 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

(Live Demo)

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
  - 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 OS 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
  - OS only executes in reaction to events

Categorizing Events

- Two kinds of events, interrupts and exceptions
- Interrupts are caused by an external event
  - Device finishes I/O, timer expires, etc.
  - Analogy: Receiving a phone call, text message
- Exceptions are caused by executing instructions
  - CPU requires software intervention to handle a fault or trap
- 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 vary by OSes, CPU architectures
  - Software interrupt
    - Asynchronous system trap (AST), asynchronous or deferred procedure call (APC or DPC)
    - Used by the OS to defer work until all hardware interrupts have been handled

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 (think array bounds checking), but at a significant performance penalty

Handling Faults (Recovery)

- Some faults are handled by "fixing" the exceptional condition and returning to the faulting context
  - Page faults cause the OS to bring missing pages 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 (Termination)

- 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

(Live Demo)

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
  - Returning from system call restores this state
- Requires architectural support to:
  - Restore saved state, reset mode, resume execution

INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT3	ZO	Valid	Valid	Generate breakpoint trap.
CD *ib*	INT *imm8*	I	Valid	Valid	Generate software interrupt with vector specified by immediate byte.
CE	INTO	ZO	Invalid	Valid	Generate overflow trap if overflow flag is 1.
F1	INT1	ZO	Valid	Valid	Generate debug trap.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	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 INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6.[1] Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recom-

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							

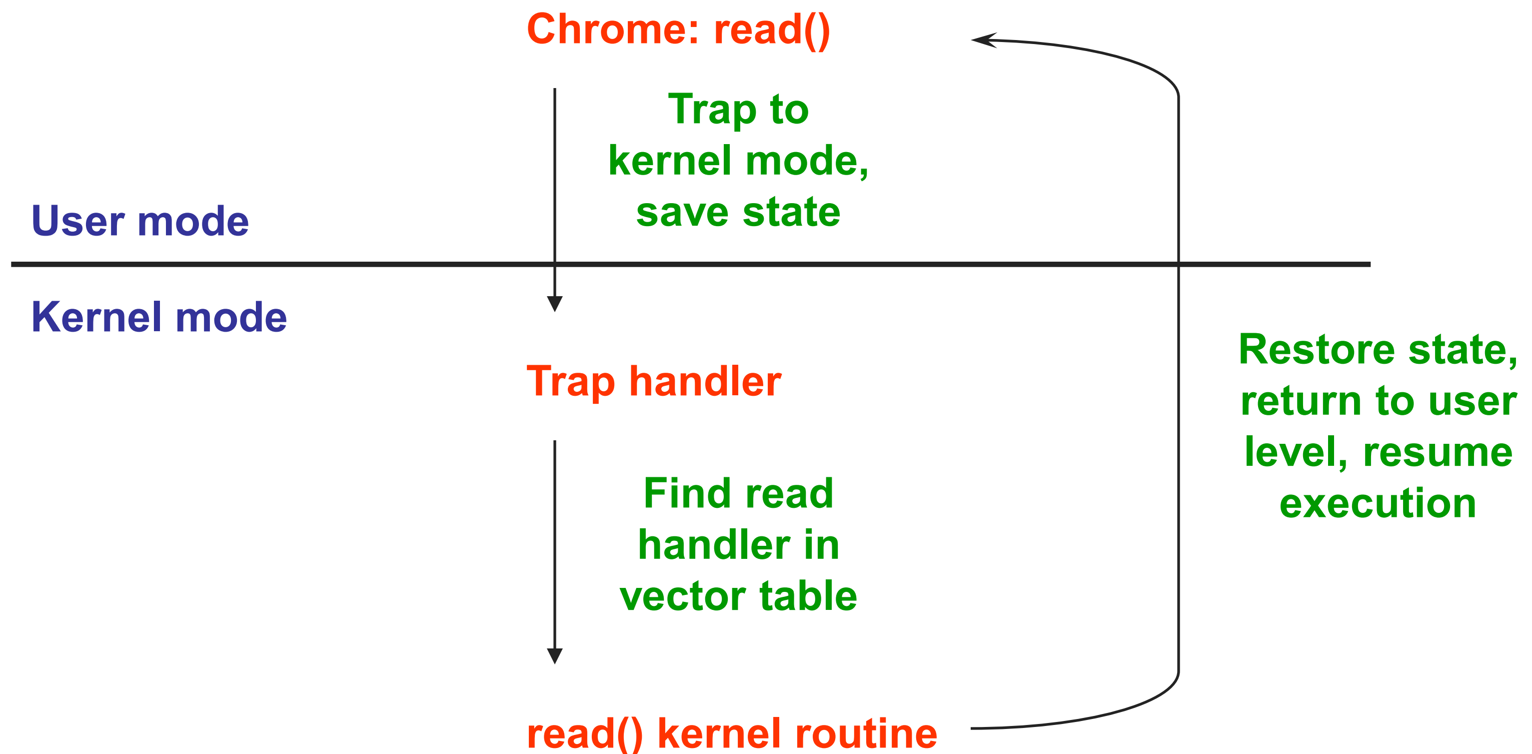
```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

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



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*

Referencing OS Data

- Processes and the OS are in different address spaces
  - How can the OS return references to kernel data structures?
- A naming issue
  - Use integer object handles or descriptors
    - » Unix file descriptors, Windows HANDLEs
    - » Only meaningful as parameters to other system calls
  - Also called capabilities (more later when we do protection)

Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers
- Interrupts on modern CPUs are precise
  - CPU transfers control only on instruction boundaries

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 what runs next
    - Basis for OS scheduler *(more later in quarter…)*
- 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 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

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 atomically
    - 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 3-5 (Processes)
- Start on Project 0