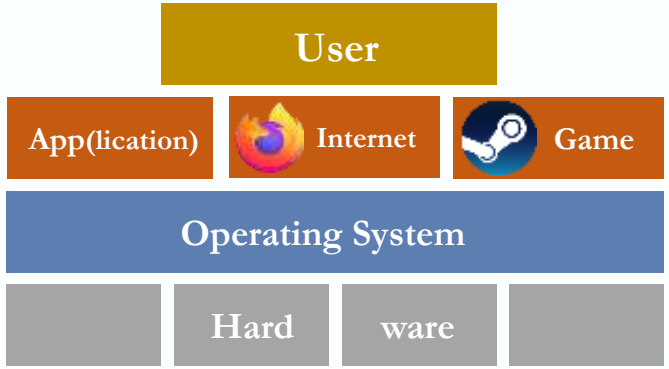
CS-446/646

Architectures & OS

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Hardware

Why start with Hardware?



Functionality fundamentally depends upon the H/W *Architectural* features

- Key goals of an OS are to enforce *Protection* and *Resource Sharing*
- If done well, applications can remain oblivious to H/W details

H/W *Architectural* support can greatly simplify or complicate OS tasks

- Early DOS, MacOS lacked *Virtual Memory* in part because the H/W did not support it
- Early Sun 1 computers used two M68000 CPUs to implement *Virtual Memory*

Architectural Features for OS

Some *Architectural* features that directly support the OS:

- *Bootstrapping*
  - At each stage of Booting, a smaller/simpler program loads and executes the more complicated one of the next stage
- Protection (*Kernel / User Mode*)
- Protected *Instructions*
- *Memory* Protection
- *System Calls*
- *Interrupts* and *Exceptions*
- *Timer*(s)
- I/O Control and Operation
- Synchronization

Architectural Features for OS

Types of *Architectural* Support:

I. **Manipulating *Privileged* machine *State***

- Protected *Instructions*
- Manipulate Device *Registers*, TLB entries, etc.

II. **Generating and Handling *Events***

- *Interrupts*, *Exceptions*, *System Calls*, etc.
- Respond to *Asynchronous (External) Events*
- CPU requires Software intervention to handle a *Fault* or a *Trap*

III. **Mechanisms to support *Synchronization***

- *Interrupt* disabling/enabling, *Atomic Instructions*

Architectural Features for OS

Types of *Architectural* Support:

I. **Manipulating *Privileged* machine *State***

- Protected *Instructions*
- Manipulate Device *Registers*, TLB entries, etc.

Why?

Imagine if any Program in the system could

- Directly access I/O Devices
- Write anywhere in *Memory*
- Read content from any *Memory* Address
- Execute a machine halt (e.g. **`hlt`**) *Instruction*

Malicious 3rd Party S/W could access your bank password data if found somewhere in *Memory*…

Note: Check out *CVE-2017-18344* security vulnerability which allowed leaking of *Kernel Space* memory addresses to *User Space* !

Protection – H/W Support

Implement execution with *Privilege*-based limitations

A first idea:

- Execute every Program *Instruction* in a "sandbox"
  - e.g. OS *Virtualization / Hosted Virtualization*
- If the Program *Instruction* is permitted, it is executed (as a CPU *Instruction*); otherwise stop
- Basic model in Javascript and other interpreted languages

… But we need a faster implementation!

Observation: most CPU *Instructions* are perfectly safe

- Run the Unprivileged code directly on the CPU
- Perform the checking on H/W

Protection – H/W Support

H/W Support: **Dual-Mode Operation** in CPU

User Mode:

- Limited Privileges
- Only those granted by the OS Kernel

Kernel Mode:

- Execution with the full Privileges of the Hardware
- Read/write to any *Memory*, access I/O Device, read/write Disk *Sector*, send/read Network *Packets*

Note:

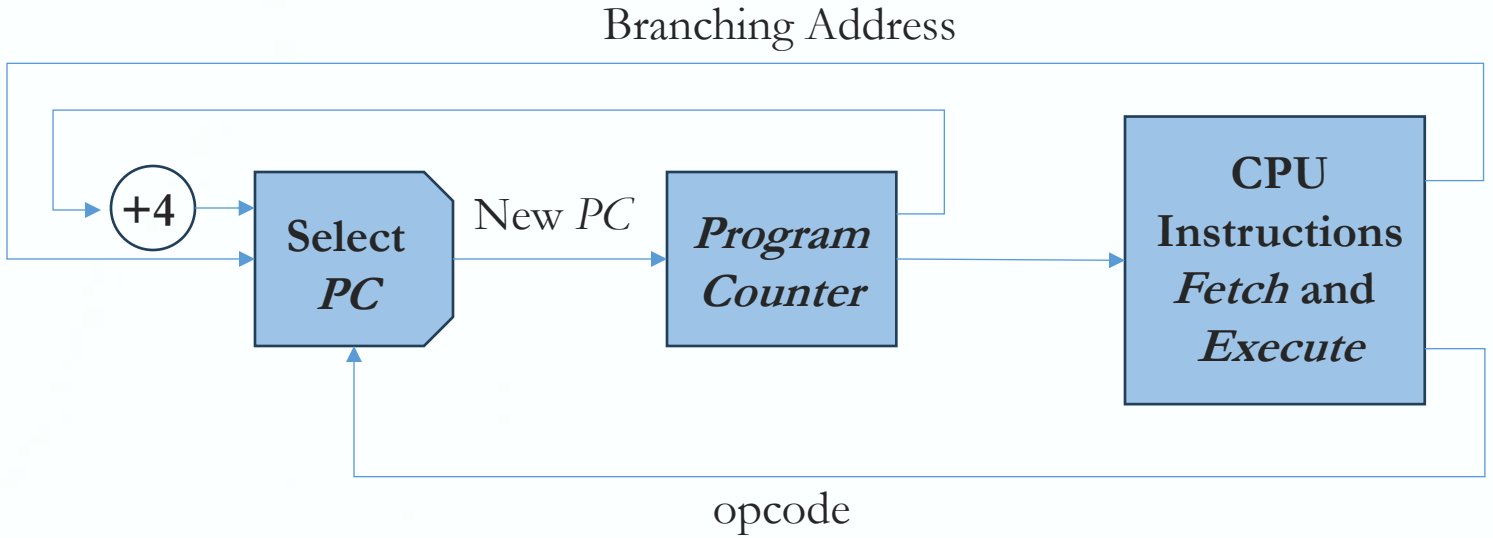
- On the x86 arch, the *Current Privilege Level* (*CPL*) is a 2-bit field in the **`CS`** *Register*
- On the MIPS arch, the *Status Register*

Protection – H/W Support

Simple CPU Model

Basic Operation:

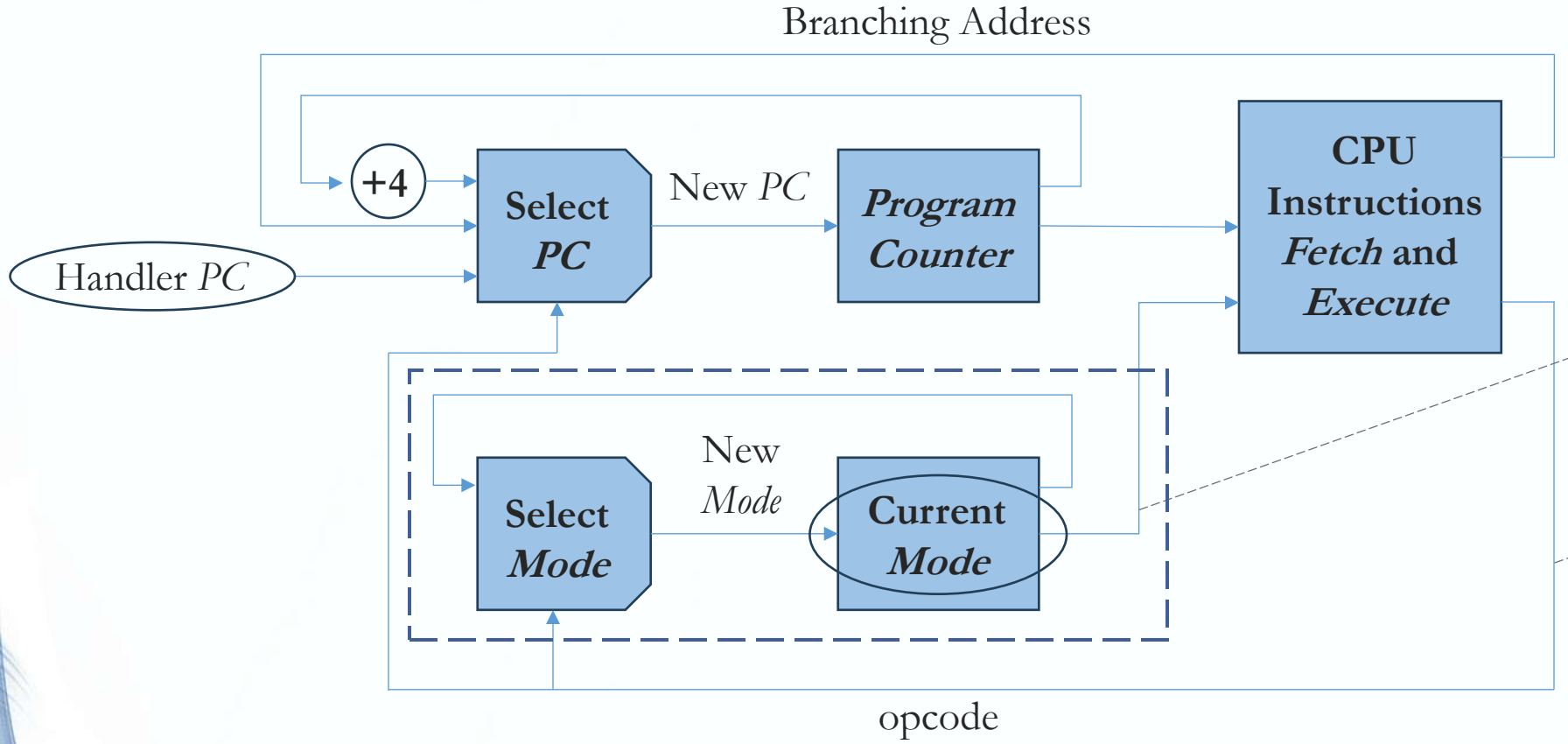
- *Fetch* (next instruction) → *Decode* (instruction) → *Execute*



Program Counter	*Instruction* Stream
4	`lea`
8	`add`
12	`sub`
16	`cmp`
20	`jne 32`
24	`mov`
28	`jmp 44`
32	`call`
…	…

Protection – H/W Support

Dual-Mode Operation CPU Model



Protection – H/W Support

H/W Support & OS: **Protected *Instructions***

A subset of *Instructions* restricted to use only by the OS:

- Known as *Protected* (Privileged) *Instructions*

Only the Operating System is allowed to:

- Directly access I/O Devices (Disks, printers, etc.)
  - *Security*, Fairness
- Manipulate *Memory Management* state
  - *Page Table* pointers, *Page* protection, *Translation Lookaside Buffer* (TLB) management, etc.
- Manipulate Protected *Control Registers*
  - *Kernel Mode*, *Interrupt Level*
- Halt *Instruction*

Protection – H/W Support

H/W Support & OS: **Protected Instructions**

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

Protection – H/W Support

How to use INVLPG on x86-64 architecture?

Asked 7 years, 2 months ago Modified 7 years, 2 months ago Viewed 3k times

3

I'm trying to measure memory access timings and need to reduce the noise produced by TLB hits and misses

In order to clear a specific page out of the TLB I tried to use the INVLPG instruction, following those two examples: http://wiki.osdev.org/Paging and http://wiki.osdev.org/Inline_Assembly/Examples

I wrote the following code:

```
static inline void __native_flush_tlb_single(unsigned long addr)
{
   asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

But the resulting binary throws an SIGSEGV on execution. As I prefere Intel syntax, I had a look at the particular disassembly:

```
invlpg BYTE PTR [rdi]
```

If I understand this correctly, invlpg will be called with the byte value at RDI, but would rather require a QWORD address.

However the second link says "The m pointer points to a logical address, not a physical or virtual one: an offset for your ds segment"

So INVLPG needs an offset from the ds segment? But the ds segment isn't used in AMD64 anymore, is it?

Can someone explain me how to use the INVLPG instruction with AMD64 or how to evict an TLB entry on this architecture?

7

The SIGSEGV happens because INVLPG is a privileged instruction and can only be called out of kernel code (Thanks Cody Gray).

In order to demonstrate the use of INVLPG I wrote a little LKM *invlpg_mod.c*:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/moduleparam.h>
#include <linux/unistd.h>
#include <linux/types.h>

// LICENSE
MODULE_LICENSE("GPL");

static inline void invlpg(unsigned long addr) {
    asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}


// init module
static int __init module_load(void) {
    int mem;
    invlpg((unsigned long) &mem);
    printk("Evicted %p from TLB", &mem);
}


//unload modules
static void __exit module_unload(void) {
    printk("Goodbye.");
}

module_init(module_load);
module_exit(module_unload);
```

Make sure you have the linux-headers installed and build the LKM with this Makefile:

```
KDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)

obj-m = invlpg_mod.o

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

Load the LKM with:

```
sudo insmod invlpg_mod.ko
```

And unload it:

```
sudo rmmod invlpg_mod.ko
```

See the output:

```
dmesg | tail
```

H/W Support & OS: **Protected Instructions**

C. Papachristos

Protection – H/W Support

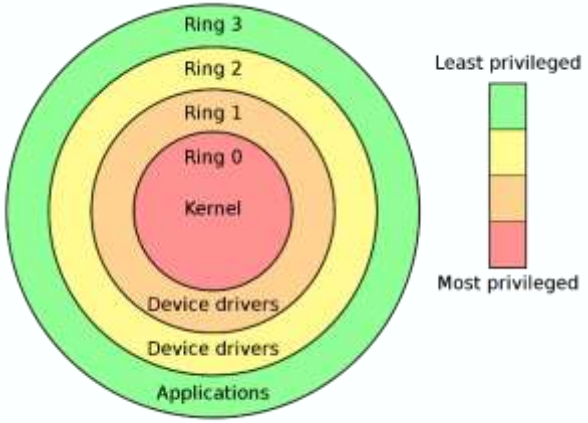
H/W Support: ***Protection Rings*** (beyond *Dual-Mode* Operation)

Modern CPUs may provide more than 2 *Privilege Levels*, called *Hierarchical Protection Domains* or *Protection Rings:*

- x86 supports four (4) levels of *Privilege*:
  - bottom 2 bits (CPL) of the **`CS`** *Register* indicate execution *Privilege*
  - Ring 0 (CPL=00) is *Kernel Mode*, Ring 3 (CPL=11) is *User Mode*
- Multics provides eight (8) levels of *Privilege*:
- ARMv7 CPUs (e.g. in smartphones) have eight (8) levels of *Privilege*

Why?

- Protect the OS from itself
  - Software-induced Vulnerabilities
- Reserved for specific OS vendors
  - e.g. *Virtualization*, where Ring 0 calls are made through a *Hypervisor* that performs them on behalf of the *VM Guest OS*



Privilege Rings for the x86 available in Protected Mode

Protection – H/W Support

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
    - e.g. Intel Software Guard eXtensions (SGX) – Running code at higher *Privilege Levels* than even the Operating System and even any underlying *Hypervisors*

Memory Management Unit (*MMU*)

- It is *Memory* management H/W that examines all *Memory* references on the *Memory Bus*, translating these requests, known as *Virtual Memory Addresses*, into *Physical Addresses* in main (*Physical*) *Memory*
  - *Base* and *Limit Registers*
  - *Page Table* pointers, *Page* protection, *Segmentation*, *Translation Lookaside Buffer* (TLB)
  - Manipulating the H/W uses *Protected* (*Privileged*) operations

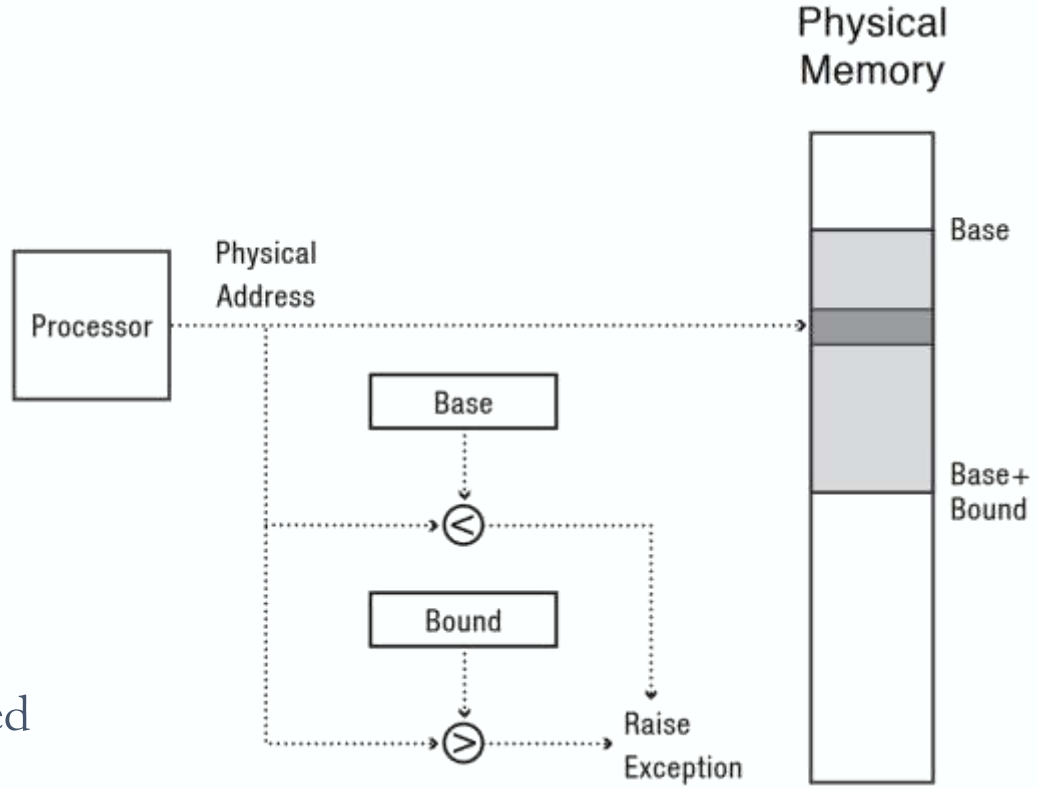
Protection – H/W Support

Memory Protection

Simple *Memory* Access w/ *Bounds* Checking

This 1st idea has issues:

- Flexibility
  - Fixed allocation,
    difficulty to expand *Heap* and *Stack*
- Memory *Fragmentation*
- *Physical Memory* dependence
  - Require changes to **MEM** (*Memory* Access)
    Instruction targets each time the Program is loaded



Protection – H/W Support

Memory Protection

Virtual Addresses

The better idea:

- Programs should refer to memory by *Virtual Addresses*
  - Start from 0
  - Illusion of "owning" the entire *Memory Address Space*
  - Every *Process* given the impression that it is working with large, contiguous sections of *Memory*
- *Virtual Address* is translated to *Physical Address*
  - Upon each *Memory* Access
  - Done in H/W (in the *MMU*) using a *Page Table* which is setup by the OS Kernel

Architectural Features for OS

Types of *Architectural* Support:

I. **Manipulating *Privileged* machine State**

- *Protected Instructions*
- Manipulate Device *Registers*, TLB entries, etc.

II. **Generating and Handling *Events***

- *Interrupts*, *Exceptions*, *System Calls*, etc.
- Respond to *Asynchronous* (External) *Events*
- CPU requires Software intervention to handle a *Fault* or a *Trap*

Events

Event

An *Event* is a "forced" change in Control Flow

- *Events* immediately stop current execution
- Changes *Mode*, *Context* (machine *State*), or both

The OS Kernel defines a *Handler* for each *Event* type

- The specific types of *Events* are defined by the *Architecture*
  - e.g. *Timer* Event, I/O *Interrupt*, *System Call*, *Trap/Software Interrupt*
- In effect, the OS is one big *Event* handler

Events

OS Control Flow

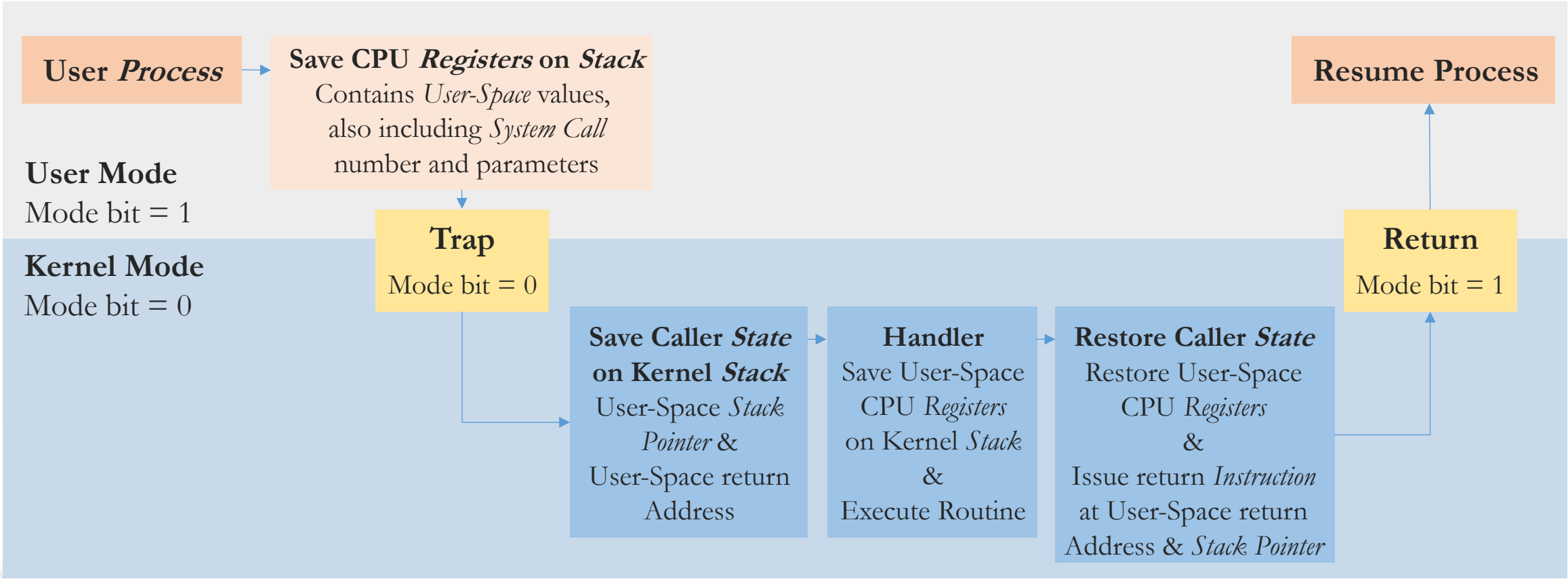
After OS Booting, all entry to Kernel is a result of some *Event*

- *Event* immediately stops current execution
- Changes mode to *Kernel Mode*
- Invoke a piece of code to handle event (*Event Handler*)

When the CPU receives an *Event* of a given type, it:

- Transfers control to *Handler* (within the OS Kernel)
- *Handler* saves Program *State* (*Program Counter* (CPU *Register* w/ Address of next *Instruction*), *Regs*, etc.)
- *Handler* executes core OS functionality, e.g., writing data to Disk
- *Handler* restores Program *State*, resumes Program execution

Events


User *Process*
Save CPU *Registers* on *Stack*
Contains *User-Space* values, also including *System Call* number and parameters
Resume Process
User Mode
Mode bit = 1
Kernel Mode
Mode bit = 0
Trap
Mode bit = 0
Save Caller *State* on Kernel *Stack*
User-Space *Stack Pointer* & User-Space return Address
Handler
Save User-Space CPU *Registers* on Kernel *Stack* & Execute Routine
Restore Caller *State*
Restore User-Space CPU *Registers* & Issue return *Instruction* at User-Space return Address & *Stack Pointer*
Return
Mode bit = 1

Events

Interrupt vs *Exception*

Two *Event* types:

Interrupts (or *H/W Interrupts*) are caused by an *External Event* (*Asynchronous*)

- Some device finishes I/O, *Timer* expires, HID user input, etc.

Exceptions (or *S/W Interrupts*) are caused by the CPU executing *Instructions* (*Synchronous*)

- Classified as *Traps*, *Faults* or *Aborts*

  Note: in Intel arch:

  - *Traps* are restarted at the Address following the Address causing the *Trap*
  - *Faults* are restarted at the Address of the *Fault*-ing *Instruction*
  - *Aborts* give no reliable restart Address
- x86 arch **`int`** instruction, Page Fault, Divide-by-Zero, etc.

Both vectored though the *Interrupt Descriptor (/Vector) Table* (*IDT*)

Events

Interrupts

Interrupts signal *Asynchronous Events*

- Indicates some Device needs services
- I/O Hardware *Interrupts*
- Hardware (and Software?) *Timers*

Why?

- A computer is more than a CPU
  - Keyboard, Disk, Printer, Camera, etc.
- These Devices occasionally need attention, but cannot predict when
  - … or it is not useful trying to anticipate it

Events

Interrupts

Simple 1st Approach to dealing with *Asynchronous Events*:

Polling

Have CPU periodically check if each Device needs service

Issues:

- Consumes CPU time even when there are no *Events* pending
- Trying to reduce checking frequency leads to longer response times

Note: Polling is not a –generally– bad practice

- Can actually be efficient when *Events* arrive very rapidly

Events

Interrupts

Better Approach:

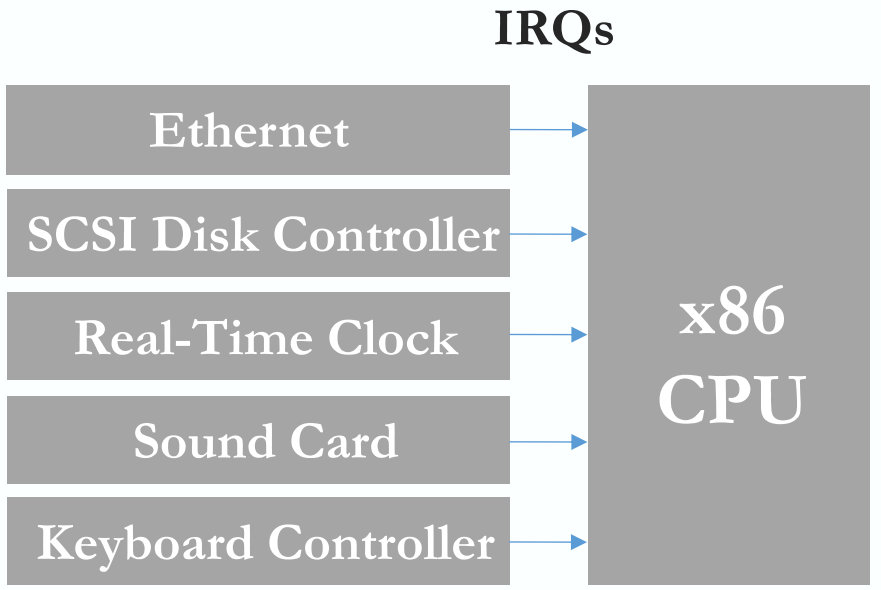
Device *Interrupt ReQuest (IRQ)* lines

Give each Device access to *Interrupt* the CPU

- Each Device can be connected to an *Interrupt ReQuest (IRQ)* line

Issues:

- Non-flexible handling of *Interrupts* ("hardcoded")
- CPU might get *Interrupted* non-stop
- A Device may overwhelm CPU
- Critical *Interrupts* can be delayed



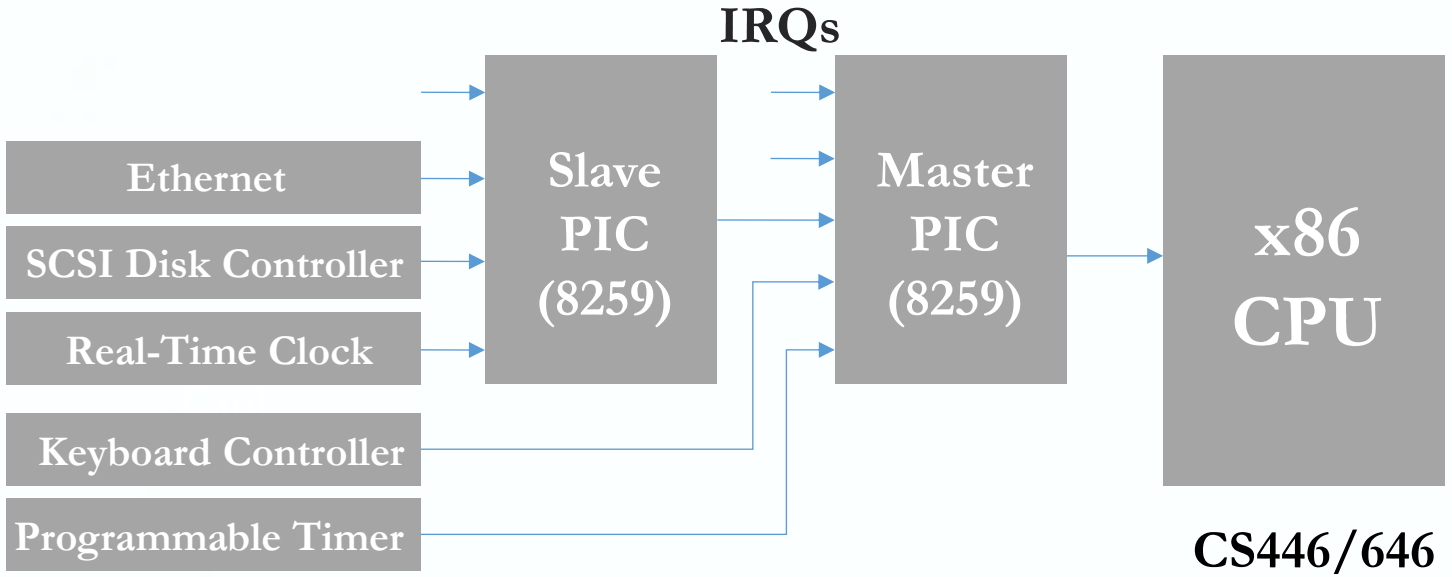
Events

Interrupts

Even better approach:

Programmable Interrupt Controller (PIC)

- I/O Devices have (unique or shared) *IRQs*
- *IRQs* are mapped by the *PIC* hardware to *Interrupt Vectors* and passed to the CPU
  - *Interrupt Vector* : Each entry of the *Interrupt Descriptor* (/*Vector*) *Table*



Events

Interrupts

Programmable Interrupt Controller (PIC)

Responsible for telling the CPU when and which device wishes to "*Interrupt*"

Example: Programmable Interrupt Controller 8259A

- Has 16 wires to devices (IRQ0 – IRQ15)

PIC translates *IRQs* to CPU *Interrupt Vector Number*

- Vector number is signaled over INTR line
  - e.g. in Pintos: IRQ0...15 delivered to vector 32...47 (src/threads/interrupt.c)

Interrupts can have varying priorities

- *PIC* needs to be able to prioritize multiple requests

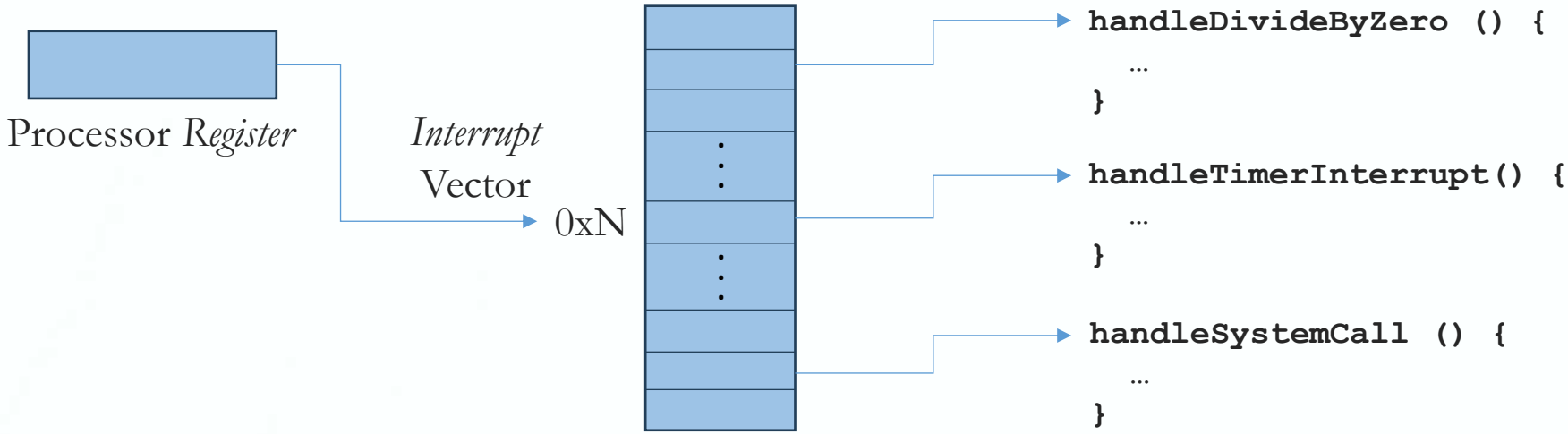
Note: It is also possible to "mask" (disable) *Interrupts* at the level of the *PIC* or the CPU

Interrupts

Interrupt Descriptor (/Vector) Table

A data structure to associate *Interrupt ReQuests* with *Handlers*

- Each entry called an *Interrupt Vector*
  - specifies *Memory* Address of *Handler Routine*
- *Architecture*-specific implementation



Interrupt Descriptor (/Vector) Table

Events

Interrupts

Interrupt Descriptor (/Vector) Table

A data structure to associate *Interrupt ReQuests* with *Handlers*

- Each entry called an *Interrupt Vector*
  - specifies *Memory* Address of *Handler Routine*
- *Architecture*-specific implementation

Note: In x86 called *Interrupt Descriptor Table* (*IDT*)

- *Architecture* supports 256 *Interrupts*, so the *IDT* contains 256 entries
- Each entry specifies the Address of the *Handler* plus some flags
- Can be programmed by the OS
  - e.g. in Pintos: **`make_intr_gate`** (src/threads/interrupt.c)

Events

Interrupt Usage Scenario: ***Timer***

Timer(s) are critical for an OS

Fallback mechanism for OS to reclaim control over (/*Preempt*) the machine's operation

- *Timer* is set to generate an *Interrupt* periodically
- Setting the *Timer* is a Privileged *Instruction*
- When *Timer* expires, generates an *Interrupt*
- Handled by Kernel, which controls resumption context
  - Basis for OS *Scheduler*

Prevents infinite loops

- OS should always be able to regain control from erroneous or malicious Programs hogging CPU

Used for time-based functions

- e.g. `sleep()`

Interrupt Usage Scenario: ***Timer***

Timer in Pintos

```
/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}

/* Timer interrupt handler. */
static void timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
```

```
/* Called by the timer interrupt
   handler at each timer tick. */
void thread_tick (void)
{
  struct thread *t = thread_current ();
  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
  else
    kernel_ticks++;

  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();
}
```


Events

Interrupt Usage Scenario: ***I/O Control***

I/O issues:

- Initiating an I/O
- Completing an I/O

Interrupts are the basis for *Asynchronous* I/O

- OS initiates I/O
- Device then proceeds to operate as an independent system
- Upon completion, device sends an *Interrupt* signal to CPU
- OS maintains an *Interrupt Descriptor (/Vector) Table* (*IDT*), based on which the CPU looks up the received *Interrupt Vector Number*, and *Context-Switches* to *Handler Routine*

Events

Interrupt vs *Exception*

Two *Event* types:

Interrupts (or *H/W Interrupts*) are caused by an *External Event* (*Asynchronous*)

- Some device finishes I/O, *Timer* expires, HID user input, etc.

Exceptions (or *S/W Interrupts*) are caused by the CPU executing *Instructions* (*Synchronous*)

- Classified as *Traps*, *Faults* or *Aborts*

  Note: in Intel arch:

  - *Traps* are restarted at the Address following the Address causing the *Trap*
  - *Faults* are restarted at the Address of the *Fault*-ing *Instruction*
  - *Aborts* give no reliable restart Address
- x86 arch **`int`** instruction, Page Fault, Divide-by-Zero, etc.

- i.e. broadly speaking, a deliberate *Exception* is a *Trap*; an unexpected one is a *Fault*
  - CPU requires software (kernel) intervention to handle a *Fault* or *Trap*

Events

Deliberate *Exception:* ***Trap***

A *Trap* is an intentional software-generated *Exception*

- The main mechanism for Programs to interact with the OS
  - On x86, Programs use the **`int`** (*Interrupt*) / **`syscall`** (*System Call*) *Instruction* to cause a *Trap*
  - On ARM, **`SVC`** (*SuperVisor Call*) *Instruction*

The *Handler* for a *Trap* is defined in the *Interrupt Descriptor (/Vector) Table*

- Kernel chooses the vector for representing the *System Call* (*syscall*) *Trap*
  - Pintos uses **`int $0x30`** to make a *syscall Trap*
  - Linux used **`int $0x80`** to make a *syscall Trap*
    - *Note:* The **`syscall`** instruction is the primary trap instruction in 64-bit x86 systems. Earlier x86 programs performed *System Calls* by triggering an interrupt with the **`int $0x80`** instruction; the Kernel would use **`iret`** to return from the *Interrupt*. For performance reasons, this approach was replaced with the **`sysenter`** and **`sysexit`** instructions on 32-bit systems. **`syscall`** and **`sysret`** are the 64-bit equivalent of these faster *System Call Instructions*.

Events

System Call *Trap*

For a *User Space* Program to invoke an OS service

- Also referenced as "*crossing the Protection Boundary*", or "*Protected Control Transfer*"

The *System Call Instruction*

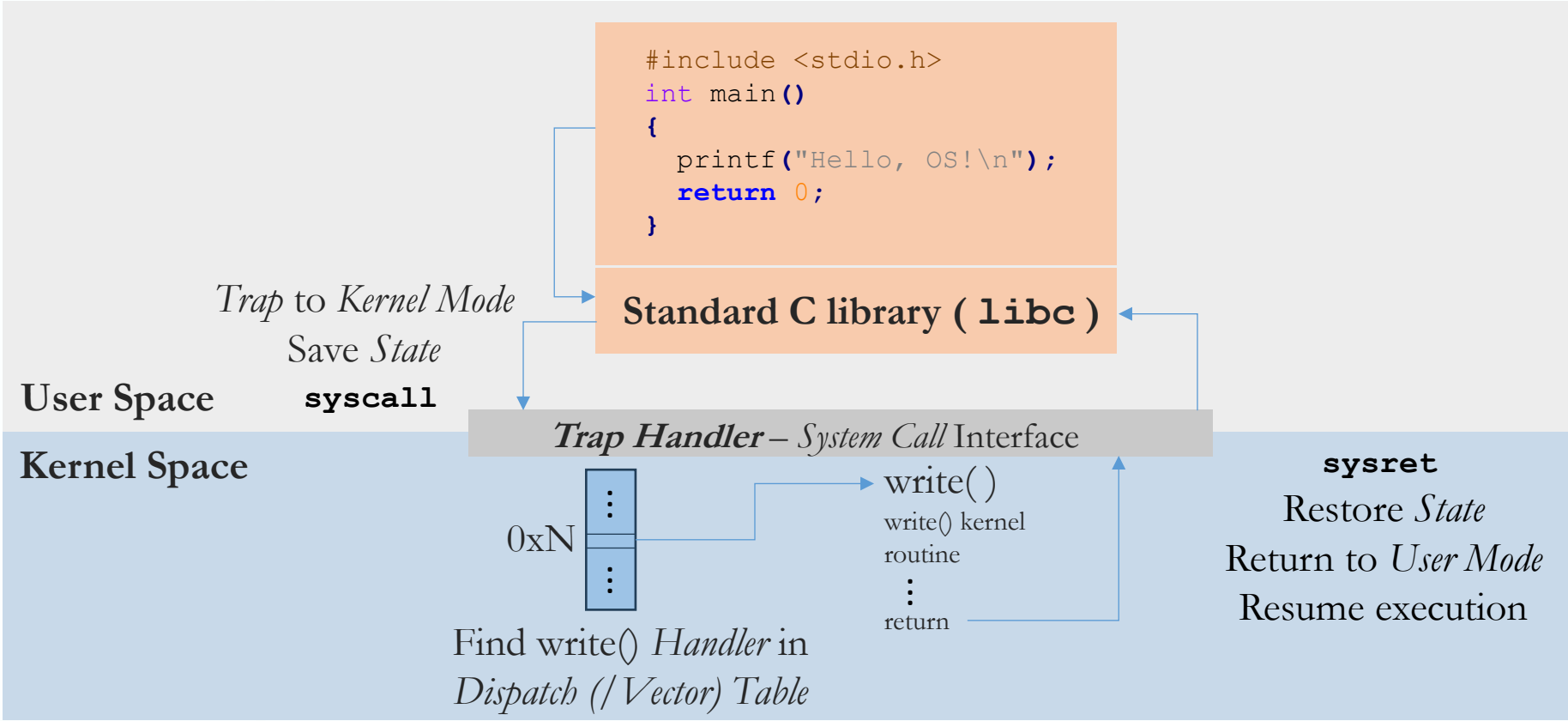
- Causes an *Exception*, which is vectored to a kernel *Handler*
- Passes a parameter determining the desired system routine to call

```
movl $20, %eax  # Get PID of current process and store to eax CPU register
int $0x80       # Invoke system call!
# Now %eax holds the PID of the current process
```

- Saves the caller *State* (*Program Counter (PC)*, *Regs*, *Mode*) so it can be restored
  - Returning from a *System Call* restores this *State*

- *Architectural* support is required to:
  - Restore saved *State*
  - Change *Mode*
  - Resume execution

System Call *Trap*



Events

System Call *Trap*

- *Architectural* support is required to Save / Restore *State*, Change *Mode*:
  - In x86 *Architecture*, when an *Interrupt* happens (e.g. on **`int`**), the CPU automatically pushes the value of the following CPU *Registers* onto the *Stack*:
    - **`SS`**: Stack Segment Selector
    - **`ESP`**: Stack pointer
    - **`EFLAGS`**: CPU flags
    - **`CS`**: Code Segment Selector
    - **`EIP`**: Instruction Pointer (i.e. the *Program Counter*)
  - On returning from *Interrupt* (e.g. on **`iret`**), the CPU automatically pops and restores the above
    - *Note:* **`iret`** only allows returning to a *Privilege Level* lesser or equal-to the current one

Events

System Call *Trap*

- *Architectural* support is required to Save / Restore *State*, Change *Mode*:
- Then, appropriate OS handling is required (*Example:* Pintos)
  - *Kernel* to *User Space*: On Program loading (e.g. initial Bootloading & **`init`** *Process* creation), to jump to a *Process* entry point and change the *Privilege Level*, we simulate a return-from-*Interrupt* by creating an artificial *Interrupt Stack Frame* with the entry point of the program in **`EIP`**, and *Code* and *Data Segments* (more on these later) **`CS`**, **`SS`** that correspond to *User Space Privilege Level*, and calling **`iret`**
    - *Remember:* CPL is lowest 2 bits of **`CS`** *Register* in x86
  - *User to Kernel Space*: Can't directly change to higher *Privilege Level* through an **`iret`** (*Security*). Instead, call **`int`** while an appropriately setup *Interrupt Descriptor (/ Vector) Table* specifies which *Code Segment* to use; then the *Interrupt Handler* entry point takes over and sets up the *Data Segment* selector(s) to ones that have *Kernel Space Privilege Level*

Events

System Call *Trap*

What would happen if the Kernel did not save *State*?

What happens if the Kernel itself executes a *System Call* ?

- "… a Device driver shouldn't read a configuration file using *Kernel-Space System Calls*; reading a file involves error management and parsing of file contents -- not something suited for Kernel code."
- (*Scheduling* ? – more on that later)

How to reference Kernel objects as arguments or results to/from *System Calls*?

- Use integer *Object Handles* or *Descriptors* (also called *Capabilities*)
  - e.g., Unix has *File Descriptors*, Windows has *HANDLEs*
  - Only meaningful as parameters to other *System Calls*
- Why not just use Kernel Addresses to define objects to be used by the Kernel?

Events

System Call Trap

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*

Events

Unintended *Exception:* ***Fault***

Hardware constantly monitors, detects, and reports "exceptional" conditions

- *Page Fault*, Unaligned Access, Divide-by-Zero

Upon exception, the hardware "*Faults*"

- Must save *State* (*PC*, *Regs*, *Mode*, etc.) so that the *Fault*-ing process can be restarted

Faults are not necessarily bad

- Modern OSes use *Virtual Memory Faults* for many functions
  - e.g. Debugging, *End-of-Stack* Detection, Garbage Collection, *Copy-on-Write*
- *Fault Exceptions* are essentially a performance optimization
  - Could detect *Faults* by inserting extra *Instructions* into code (at a significant performance penalty)

Handling ***Fault***s

Some faults are handled by "fixing"

- Fix the exceptional condition and return back to the *Fault*-ing context
  - *Page Faults* handled on the OS side by placing the missing *Page* into *Memory*
  - *Fault Handler* then resets *PC* to re-execute *Instruction* that caused the *Page Fault*

Some *Faults* are handled by just notifying the *Process*

- *Fault Handler* may change the saved context to transfer control to a *User Mode Handler*
  - *User Mode Handler* must be registered with OS
    - e.g. Unix *SIGNALs* (**`SIGALRM`**, **`SIGHUP`**, **`SIGTERM`**, **`SIGSEGV`**, etc.)
      or Win *User-Mode Async Procedure Calls* (*APC*s)

Handling *Fault*s

Kernel may handle unrecoverable *Faults* by killing the *Process*

- e.g. 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* taking place while in *Kernel Mode*?

- e.g. Dereference **`null`**, Divide-by-Zero, Undefined *Instruction*
  - These *Faults* considered fatal, OS crashes
    - Unix *panic* , Windows *Blue Screen Of Death*
  - Kernel is halted, state dumped to a core file, machine locked up

Note (from *xv6 Reference*, *C trap handler*): … If the *Trap* is not a *System Call* and not a hardware device looking for attention, *Trap* assumes it was caused by incorrect behavior (e.g., divide by zero) as part of the code that was executing before the *Trap*. If the code that caused the *Trap* was a User Program, xv6 prints details and then sets **`cp->killed`** to remember to clean up the *User Process*. … If it was the Kernel running, there must be a Kernel bug: *Trap* prints details about the surprise and then calls **`panic`**.

Architectural Features for OS

Types of *Architectural* Support:

I. **Manipulating *Privileged* machine *State***

- Protected *Instructions*
- Manipulate Device *Registers*, TLB entries, etc.

II. **Generating and Handling *Events***

- *Interrupts*, *Exceptions*, *System Calls*, etc.
- Respond to *Asynchronous (External) Events*
- CPU requires Software intervention to handle a *Fault* or a *Trap*

III. **Mechanisms to support *Synchronization***

- *Interrupt* disabling/enabling, *Atomic Instructions*

Synchronization

Interrupts cause difficult problems

- An *Interrupt* can occur at any time
- A *Handler* can execute that interferes with the code that was *Interrupted*

OS must be able to synchronize *Concurrent* execution

Need to guarantee that short *Instruction* sequences execute *Atomically*

- *Disable Interrupts*
  - Temporarily turn off *Interrupts* before sequence, execute sequence, re-enable *Interrupts*
- Special *Atomic* instructions exist to ensure atomicity (why?) when performing:
  - *Read* / *Modify* / *Write* of a *Memory* Address, *Exchange*, *Compare-and-Swap*, etc.
    - e.g. **`xchg`** *Instruction* on x86

CS-446/646

Time for Questions !