Processes (Continued)

Professor Natacha Crooks
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi
Recall: The Process

A executing program with restricted rights

Enforcing mechanism must not hinder functionality or hurt performance
Recall: Dual Mode Operation

Use a bit to enable two modes of execution

**In User Mode**
- Processor checks each instruction before executing it
- Executes a limited (safe) set of instructions

**In Kernel Mode**
- OS executes with protection checks off
- Can execute any instructions
Recall: Operating System Kernel

Lowest level of OS running on system.
Kernel is **trusted** with **full access** to all hardware capabilities

All other software (OS or applications) is considered **untrusted**

<table>
<thead>
<tr>
<th>Untrusted</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rest of OS</td>
</tr>
<tr>
<td>Trusted</td>
<td>Operating System Kernel</td>
</tr>
<tr>
<td>Untrusted</td>
<td>Hardware</td>
</tr>
</tbody>
</table>
Recall: Hardware must support

1) Privileged Instructions
Unsafe instructions cannot be executed in user mode

2) Memory Isolation
Memory accesses outside a process’s address space prohibited

3) Interrupts
Ensure kernel can regain control from running process

4) Safe Transfers
Correctly transfer control from user-mode to kernel-mode and back
Goals for today

• (Continued) Hardware support for dual mode

• 61C Review: The Stack

• How to switch from user mode to kernel mode and back?
  – For interrupts, exceptions, syscalls?
Hardware must support

1) Privileged Instructions
   Unsafe instructions cannot be executed in user mode

2) Memory Isolation
   Memory accesses outside a process’s address space prohibited

3) Interrupts
   Ensure kernel can regain control from running process

4) Safe Transfers
   Correctly transfer control from user-mode to kernel-mode and back
Kernel must be able to **regain control** of the processor

**Hardware to the rescue! (Again x 2)**

**Hardware Interrupts**

Set to interrupt processor after a specified delay or specified event and transfer control to (specific locations) in Kernel.

**Resetting timer is a privileged operation**
Hardware must support

1) Privileged Instructions
Unsafe instructions cannot be executed in user mode

2) Memory Isolation
Memory accesses outside a process’s address space prohibited

3) Interrupts
Ensure kernel can regain control from running process

4) Safe Transfers
Correctly transfer control from user-mode to kernel-mode and back
How do safely/correctly transition from executing user process to executing the kernel?

1) System Calls
2) Exceptions
3) Interrupts

- Asynchronous
- Can be maskable or non-maskable

Synchronous Events (trapping)
Safe Control Transfer: System Calls

User program requests OS service
Transfers to kernel at well-defined location

Synchronous/non-maskable

Read input/write to screen, to files, create new processes, send network packets, get time, etc.

How many system calls in Linux 3.0?
- a) 15
- b) 336
- c) 1021
- d) 21121

https://man7.org/linux/man-pages/man2/syscalls.2.html
System Calls are the “Narrow Waste”

Simple and powerful interface allows separation of concern
Eases innovation in user space and HW

Compliers
Word Processing
Web Browsers
Email
Web Servers
Databases
Application / Service

Portable OS Library
System Call Interface
Portable OS Kernel
Platform support, Device Drivers

User
System

Software

Hardware

x86
PowerPC
ARM
PCI

Ethernet (1Gbs/10Gbs)
802.11 a/g/n/ac
SCSI
Graphics
Thunderbolt

Crooks CS162 © UCB Fall 2022
System Calls in the Wild (In Linux)
Safe Control Transfer: Exceptions

Any unexpected condition caused by user program behaviour

Stop executing process and enter kernel at specific exception handler

Synchronous and non-maskeable

Process missteps (division by zero, writing read-only memory)
Attempts to execute a privileged instruction in user mode
Debugger breakpoints!
Exceptions in the Wild (In Linux)

```c
/* SPDX-License-Identifier: GPL-2.0 */
#define X86_TRAP_DE 0 /* Divide-by-zero */
define X86_TRAP_DB 1 /* Debug */
define X86_TRAP_NMI 2 /* Non-maskable Interrupt */
define X86_TRAP_BP 3 /* breakpoint */
define X86_TRAP_OF 4 /* Overflow */
define X86_TRAP_FLT 5 /* Round Range Exceeded */
define X86_TRAP_DF 6 /* Invalid Opcode */
define X86_TRAP_NM 7 /* Device Not Available */
define X86_TRAP_PF 8 /* Double Fault */
define X86_TRAP_TR 9 /* Coprocessor Segment Overrun */
define X86_TRAP_TO 10 /* Invalid TSS */
define X86_TRAP_NT 11 /* Segment Not Present */
define X86_TRAP_SS 12 /* Stack Segment Fault */
define X86_TRAP_GP 13 /* General Protection Fault */
define X86_TRAP跚 14 /* Page Fault */
```
Safe Control Transfer: Interrupts

Asynchronous signal to the processor that some external event has occurred and may require attention

When process interrupt, stop current process and enter kernel at designated interrupt handler

Timer Interrupts, IO Interrupts, Interprocessor Interrupts
Safe Control Transfer: Kernel->User

New Process Creation
Kernel instantiates datastructures, sets registers, switches to user mode

Resume after an exception/interrupt/sySCALL
Resume execution by restoring PC, registers, and unsetting mode

Switching to a different process
Save old process state. Load new process state (restore PC, registers). Unset mode.
Goal 2: The Stack is Back (Review)

**Stack** Contains temporary data such as method/function parameters, return address and local variables.

**Heap** Dynamically allocated memory to a process during its run time.

```c
define malloc(sizeof(foo))
```
Stack Terminology (Review)

Stack Frame
All the information on the stack pertaining to a function call

Frame Pointer (%ebp)
Contain base address of function's frame.

Stack Pointer (%esp)
Points to the next item on the stack.

Instruction Pointer (%eip)
Indicates the current address of the program being executed
The Call Stack (Review)

```c
int add(int a, int b) {
    int result = a + b;
    return result;
}

void foo() {
    int x = add(5, 10);
}
```
The Call Stack (Review)

```c
int add(int a, int b) {
    int result = a+b;
    return result;
}

void foo() {
    int x = add(5,10);
}
```

crooks@laptop> gcc -S -m32 add.c

```
add:
pushl  %ebp
movl  %esp, %ebp
subl  $16, %esp
movl  8(%ebp), %edx
movl  12(%ebp), %eax
addl  %edx, %eax
movl  %eax, -4(%ebp)
movl  -4(%ebp), %eax
leave/ret

foo:
pushl  %ebp
movl  %esp, %ebp
pushl  $10
pushl  $5
call  add
addl  $8, %esp
movl  %eax, -4(%ebp)
leave/ret
```
The Call Stack (Review)

```c
void foo() {
    int x = add(5,10);
}
```

```assembly
foo:
    pushl %ebp
    movl %esp, %ebp
```

Save old frame pointer.

Set current frame pointer to stack pointer

Frame pointer is base of stack frame
void foo() {
    int x = add(5, 10);
}

foo:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    pushl $10
    pushl $5

Load Function Parameters On Stack (reverse order)

Create space for x

Stack Pointer (esp)

Stack Pointer (esp)

Stack Pointer (esp)
The Call Stack (Review)

```c
void foo() {
    int x = add(5, 10);
}
```

foo:
- pushl %ebp
- movl %esp, %ebp
- subl $4, %esp
- pushl $10
- pushl $5
- call bar

Call instruction pushes EIP to stack and jumps to bar location.
The Call Stack (Review)

int add(int a, int b) {
    int result = a+b;
    return result;
}

add:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp

1) Save frame pointer and set to stack pointer

Stack Pointer (esp)

Stack Pointer (esp)
int add(int a, int b) {
    int result = a + b;
    return result;
}

add:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
addl %edx, %eax

Stack Pointer (esp)

ebp
X=?
12 (%ebp)
10
5
8(%ebp)
EIP
ebp
### The Call Stack (Review)

#### Function `add`

```c
int add(int a, int b) {
    int result = a + b;
    return result;
}
```

### Stack Frame

- **ebp**
- **X=?**
- **10**
- **5**
- **EIP**
- **ebp**
- **result**

Local Variables are stored in the stack frame.
The Call Stack (Review)

```c
int add(int a, int b) {
    int result = a + b;
    return result;
}
```

Move return value to eax register

```
add:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
addl %edx, %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
```

**X=?**

- ebp
- 10
- 5
- EIP
- ebp
- result
The Call Stack (Review)

```c
int add(int a, int b) {
    int result = a + b;
    return result;
}
```

```
add:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
addl %edx, %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
leave
ret
```

Leave instruction restores caller's frame (pops local variables and edb)

Return instruction pops EIP and restores control to EIP
The Call Stack (Review)

```c
void foo() {
    int x = add(5,10);
}
```

```assembly
foo:
    pushl %ebp
    movl %esp, %ebp
    pushl $10
    pushl $5
    call add
    addl $8, %esp
```

Stack Pointer (esp)

Pop function parameters
void foo() {
    int x = add(5, 10);
}

foo:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    pushl $10
    pushl $5
    call add
    addl $8, %esp
    movl %eax, -4(%ebp)
    x = 15
The state of a program’s execution is succinctly and completely represented by CPU register state

EIP, ESP, EBP, Eflags/PSW
Goal 2: User -> Kernel Mode
Goal 3: User -> Kernel Mode

Key Requirement:
Malicious user program (or IO device) cannot corrupt the kernel.

Interrupts, exceptions or system calls handled similarly
=> fewer code paths, fewer bugs.

1) Limited Entry
   Cannot jump to arbitrary code in kernel

2) Atomic Switch
   Switch from process stack to kernel stack

3) Transparent Execution
   Restore prior state to continue program
Interrupt Handling Roadmap

1) Processor detects interrupt!

2) Suspend user program and switch to kernel stack

3) Identify interrupt type and invoke appropriate interrupt handler

4) Restore user program
What happens when I type “OS is cool” on my keyboard while the Add program is running?
1) Interrupt Detection (Hardware)

Device sends electric signal over interrupt request line (IRQ) to interrupt controller
1) Interrupt Detection (Hardware)

APIC converts IRQ to a vector number and sends signal to processor

Processor detects interrupt

OS is cool
## IRQs

<table>
<thead>
<tr>
<th>IRQ</th>
<th>Bus type</th>
<th>Typically used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>none</td>
<td>Non-maskable Interrupt (NMI); system timer</td>
</tr>
<tr>
<td>01</td>
<td>none</td>
<td>Keyboard port</td>
</tr>
<tr>
<td>02</td>
<td>none</td>
<td>Programmable Interrupt Controller (PIC); cascade to IRQ 09</td>
</tr>
<tr>
<td>03</td>
<td>8/16-bit</td>
<td>Communications Port 2 (COM2:)</td>
</tr>
<tr>
<td>04</td>
<td>8/16-bit</td>
<td>Communications Port 1 (COM1:)</td>
</tr>
<tr>
<td>05</td>
<td>8/16-bit</td>
<td>Sound card; printer port (LPT2:)</td>
</tr>
<tr>
<td>06</td>
<td>8/16-bit</td>
<td>Floppy disk controller</td>
</tr>
</tbody>
</table>
2) Save Recovery State (Hardware)

Save register values (recovery state) for process recovery

```c
int add(int a, int b) {
    int result = a+b;
    return result;
}
```

Which registers need to be saved by hardware to restore program?

- Stack Pointer (esp)
- Program Counter (eip)
- Execution Flags / Program Status Word (Eflags)
3) Switching (atomically) to Kernel Stack

```c
int add(int a, int b) {
    int result = a + b;
    return result;
}
```

Switches stack pointer to base of kernel stack

Pushes recovery state onto the new stack
(+ optional error code)

Question 1:
Why did hardware need to save registers before switching to kernel stack?

Must overwrite EIP/SP when switching!

Question 2:
Why do we need a separate kernel stack?

Integrity and privacy concerns
A Tale of Two Stacks

```c
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
    uint sz;                     // Size of process memory (bytes)
    pde_t* pgdir;                // Page table
    char *kstack;                // Bottom of kernel stack for this process
    enum procstate state;       // Process state
    int pid;                     // Process ID
    struct proc *parent;        // Parent process
    struct trapframe *tf;       // Trap frame for current syscall
    struct context *context;    // swtch() here to run process
    void *chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;           // Current directory
    char name[16];               // Process name (debugging)
};
```

Xv6 Kernel (proc.h)
4) Invoke Interrupt Handle (Hardware)

Interrupt vector is an index into Interrupt Vector Table (or interrupt descriptor table).

Index contains appropriate Interrupt Handler Routine

Control Unit sets EIP to handler

Handler saves all remaining user registers into stack and implements necessary logic (Transition software)
5) Return to Program

Pop all user registers from kernel stack (restore register state)

Invoke iret instruction to pop saved EIP, EFLAGS, and SP registers from kernel’s exception stack to relevant registers

Return to user mode
Concurrent Interrupts

What happens if an interrupt happens while processing an interrupt?

Hardware provides instruction to temporarily defer delivery of interrupt (disable interrupt), and re-enable them when safe (enable interrupt)

Interrupts are disabled when an interrupt handler is running

Periods during which interrupts are disabled should be very short!
Interrupt Summary

1) Device sends signal to APIC

2) Processor detects interrupt

3) Save Recovery State and switch to Kernel Stack

4) Jump to interrupt handler table at appropriate vector. Invoke interrupt handler

5) Restore user program
What about syscalls?

System calls are user functions that request services from the OS. Described as function call, with a name, parameters and return value.

Good news!
Syscalls are handled (almost) identically to interrupts.
What about syscalls?

Syscalls issue a “trap” instruction (\texttt{int \ 0x80})
Generated interrupt will trigger exception vector 128!

<table>
<thead>
<tr>
<th></th>
<th>Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>rtc_handler</td>
</tr>
<tr>
<td>33</td>
<td>keyboard_handler</td>
</tr>
<tr>
<td>...</td>
<td>floppy_handler</td>
</tr>
<tr>
<td>127</td>
<td>disk_handler</td>
</tr>
<tr>
<td>128</td>
<td>syscall_handler</td>
</tr>
</tbody>
</table>

How does handler know which syscall to execute?
System Call number fed in to %eax register.
System call number entry into system call dispatch table,

What about parameters and return values?
Propagated through registers.

Warning: Parameters must be carefully checked.
What about syscalls?

Four differences:

1) Extra-layer of indirection (system call table)

2) Leverage registers for parameters/values

3) When executing iret, increment EIP by one to go to next instruction

4) Usually, interrupts not disabled
What about exceptions?

It’s the same!
The magic of the IVT

Single, well-defined entry point in the kernel helps with security
Tension between performance and simplicity

Accessing IDT can be slow if not in cache.
Syscalls very common, can we make them cheaper?

Allocate a special register (machine specific register) to directly store address of system call dispatch table

Store register call in the rax register

But backwards compatibility ...
Goals for today

- (Continued) Hardware support for dual mode
  Privileged Instructions, Memory Isolation, Timer Interrupts, Safe Context Switching.

- 61C Review: The Stack
  Stack Pointer, Frame Pointer, Program Counter

- How to switch from user mode to kernel mode and back?
  Switch to specified location in kernel & atomic.
  Interrupts, Syscalls, Exceptions handled identically. Use of the interrupt vector table