Exceptional Control Flow:
System Calls, Page Faults etc.

Slides adapted from: Gregory Kesden and Markus Püschel of Carnegie Mellon University
Control Flow

Processors do only one thing:
- From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
- This sequence is the CPU’s control flow (or flow of control)

**Physical control flow**

<startup>
inst<sub>1</sub>
inst<sub>2</sub>
inst<sub>3</sub>
...
inst<sub>n</sub>
<shutdown>
Altering the Control Flow

- **Up to now:** two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in *program state*

- **Insufficient** for a useful system:
  **Difficult to react to changes in** *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

- **System needs mechanisms for “exceptional control flow”**
Exceptional Control Flow

- Exists at all levels of a computer system

- Low level mechanisms
  - 1. Exceptions
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software

- Higher level mechanisms
  - 2. Process context switch
    - Implemented by OS software and hardware timer
  - 3. Signals
    - Implemented by OS software
  - 4. Nonlocal jumps: `setjmp()` and `longjmp()`
    - Implemented by C runtime library
Exceptions

- **Exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

![Diagram of exception handling]

- **Event** triggers a transition from user code to kernel code.
- **Exception** processing is handled by the exception handler.
- Possible actions include:
  - Return to *I_current*
  - Return to *I_next*
  - Abort
Exception Tables
(also known as Interrupt Vector)

- Each type of event has a unique exception number \( k \)
- \( k \) = index into exception table (a.k.a. interrupt vector)
- Handler \( k \) is called each time exception \( k \) occurs
(partial) Taxonomy
Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor’s *interrupt pin*
  - Handler returns to “next” instruction

- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - aborts current program
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```c
int a[1000];
main ()
{
    a[500] = 13;
}
```

80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10

---

**User code**

```
movl   $0xd,0x8049d10
```

**Kernel code**

- Exception: page fault
- Copy page from disk to memory
- Return and reexecute movl
Fault Example: Invalid Memory Reference

```c
int a[1000];
main ()
{
    a[5000] = 13;
}
```

User code

Kernel code

- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”
Traps: System Calls

- Each x86-64 system call has a unique ID number
- Examples:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
System Call Example: Opening File

- **User calls:** `open(filename, options)`
- **Calls** `__open` function, which invokes system call instruction `syscall`

```
000000000000e5d70 <__open>:
  ...
  e5d79:  b8 02 00 00 00       mov  $0x2,%eax  # open is syscall #2
  e5d7e:  0f 05               syscall    # Return value in %rax
  e5d80:  48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
  ...
  e5dfa: c3                  retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
System call

- Applications should be prevented to directly access hardware such as
  - Physical memory,
  - disk,
  - network,
  - halt

- But nevertheless, they need to access these resources in a controlled way:
  - Read/write their own memory
  - Access the files that they have permission
  - Access the network for its own communications
  - Halt

- Processors run at different security levels:
  - User level:
  - Kernel-level:
Privileged instructions

- At kernel level, CPU can execute certain instructions (such as halt) that directly access hardware.
- At user-level the use of privileged instructions are not allowed by hardware.
- User applications do not include privileged instructions.
- Only System Call code includes privileged instructions.
System calls

- **Programming interface to the services provided by the OS**
  - A set of functions (“API” (Application Programming Interface)) provided by the OS to the user applications
  - Allow the user applications to access hardware in a controlled way

- **System calls are functions that can directly access hardware**
Library example
System Calls

- **Process Control**
  - Load, execute and, abort
  - create and terminate process

- **File management**
  - create file, delete file
  - open, close, read, write, seek

- **Device Management**
  - request device, release device
  - read, write, reposition

- **Information Maintenance**
  - get/set time or date, get/set system data

- **Communication**
  - create, delete communication connection
  - send, receive messages
Most common System API

- **Most common system API**
  - **POSIX** API (most versions of UNIX, Linux, and Mac OS X)
  - **Win32** API for Windows

- On Unix, Unix-like and other POSIX-compliant operating systems, popular system calls are `open`, `read`, `write`, `close`, `wait`, `exec`, `fork`, `exit`, and `kill`
Most common System API

- **Most common system API**
  - POSIX API (most versions of UNIX, Linux, and Mac OS X)
  - Win32 API for Windows

- **POSIX (IEEE 1003.1, ISO/IEC 9945)**
  - Very widely used standard based on (and including) C-language
  - Defines both
    - *system calls* and
    - compulsory *system programs* together with their functionality and command-line format
      - E.g. `ls -w dir` prints the list of files in a directory in a ‘wide’ format
  - Complete specification is at [http://www.opengroup.org/onlinepubs/9699919799/nframe.html](http://www.opengroup.org/onlinepubs/9699919799/nframe.html)

- **Win32 (Microsoft Windows based systems)**
  - Specifies system calls together with many Windows GUI routines
    - VERY complex, no really complete specification
System programs

- System programs are “utilities” that are commonly bundled with the Operating System, to facilitate its use by the user.
  - File Management
    - rm
  - Status information
    - ps
  - File modification
    - vi
  - Programming Language support
    - gcc
  - Program loading and execution
    - ld
  - Communication
    - ssh

- There is nothing special about a system program. They are merely user applications, and you can replicate them.
  - E.g. you can write your own “ls”

- Don’t ever confuse them with system calls!