Introduction to Operating Systems

Logical Control Flow
Exceptional Control Flow
Process Model
Operating Systems

Problem: unwieldy hardware resources
- complex and varied
- limited

Solution: operating system
Manage, abstract, and virtualize hardware resources
- Simpler, common interface to varied hardware
- Share limited resources among
- Protect
Key abstractions provided by kernel
- process
- virtual memory

Virtualization mechanisms and hardware support:
- context-switching
- exceptional control flow
- address translation, paging, TLBs
Processes

Program = code (static)

Process = a running program instance (dynamic)
  - code + state (contents of registers, memory, other resources)

Key illusions:
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private address space
    - Each process seems to have exclusive use of full memory

- Why? How?
Implementing logical control flow

**Abstraction:** every process has full control over the CPU

**Implementation:** time-sharing
Context Switching

*Kernel (shared OS code)* switches between processes

Control flow passes between processes via *context switch*.

- Context =

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>user code</td>
<td>user code</td>
</tr>
<tr>
<td>kernel code</td>
<td>kernel code</td>
</tr>
<tr>
<td>user code</td>
<td>user code</td>
</tr>
</tbody>
</table>

\[ \text{context switch} \]

\[ \text{context switch} \]
Exceptional Control Flow

Hardware support for reacting to the rest of the world
Control Flow

- Processor: read instruction, execute it, go to next instruction, repeat

**Physical control flow**

<startup>

\[ \text{inst}_1 \]
\[ \text{inst}_2 \]
\[ \text{inst}_3 \]
...
\[ \text{inst}_n \]

<shutdown>

Explicit changes:

Exceptional changes:
Exceptions

Synchronous: caused by instruction
  - *Traps*: system calls

  - *Faults*: unintentional, maybe recoverable

  - *Aborts*: unintentional, unrecoverable

Asynchronous (Interrupts): caused by external events
  - incoming I/O activity, reset button, timers, signals
Exceptions: hardware support for OS

transfer control to OS in response to event

What code should the OS run?

User Code | OS Kernel
---|---

event

exception

return or abort

exception processing by exception handler
Interrupt Vector

in memory
special register holds base address

Exception Table

0
1
2
... n-1

code for exception handler 0
code for exception handler 1
code for exception handler 2
code for exception handler n-1

... a jump table for exceptions...
Open a file (trap/system call)

- **User process calls:** `open(filename, options)`
- **open executes system call instruction** `int`

```
0804d070 <__libc_open>:
   ...
804d082:    cd 80     int  $0x80
804d084:    5b       pop  %ebx
  ...
```

![Diagram of user code and OS kernel interaction]
Segmentation Fault

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

Write to invalid memory location.

User Code

```
80483b7: c7 05 60 e3 04 08 0d movl $0xd,0x804e360
```

OS Kernel

- exception: page fault
- detect invalid address
- signal process
- aborts process with SIGSEGV signal
Page Fault

Write to valid memory location ...
... but contents currently on disk instead

(more later: virtual memory)