Exceptions and Processes

Jennifer Rexford

The material for this lecture is drawn from

Computer Systems: A Programmer’s Perspective (Bryant & O’Hallaron) Chapter 8
Goals of this Lecture

• Help you learn about:
  • Exceptions
  • The process concept
  ... and thereby...
  • How operating systems work
  • How applications interact with OS and hardware

The process concept is one of the most important concepts in systems programming
Context of this Lecture

Second half of the course

Previously:
- C Language
- Assembly Language
- Machine Language

Starting Now:
- Application Program
- Operating System
- Hardware

Application programs, OS, and hardware interact via exceptions
Motivation

Question:
- How does a program get input from the keyboard?
- How does a program get data from a (slow) disk?

Question:
- Executing program thinks it has exclusive control of CPU
- But multiple programs share one CPU (or a few CPUs)
- How is that illusion implemented?

Question:
- Executing program thinks it has exclusive use of memory
- But multiple programs must share one memory
- How is that illusion implemented?

Answers: Exceptions…
Exceptions

• **Exception**
  • An abrupt change in control flow in response to a change in processor state

• **Examples:**
  • Application program:
    • Requests I/O
    • Requests more heap memory
    • Attempts integer division by 0
    • Attempts to access privileged memory
    • Accesses variable that is not in real memory (see upcoming “Virtual Memory” lecture)
  • User presses key on keyboard
  • Disk controller finishes reading data
Exceptions Note

• Note:

Exceptions in OS ≠ exceptions in Java

Implemented using try/catch and throw statements
Exceptional Control Flow

- Application program
- Exception handler in operating system

- Exception
- Exception return (optional)
- Exception processing
Exceptions vs. Function Calls

- Exceptions are **similar to** function calls
  - Control transfers from original code to other code
  - Other code executes
  - Control returns to original code

- Exceptions are **different from** function calls
  - Processor pushes **additional state** onto stack
    - E.g. values of *all* registers (including EFLAGS)
  - Processor pushes data onto **OS’s stack**, not application’s stack
  - Handler runs in **privileged mode**, not in **user mode**
    - Handler can execute all instructions and access all memory
  - Control **might return** to next instruction
    - Control sometimes returns to **current** instruction
    - Control sometimes does not return at all!
There are four classes of exceptions...

- Interrupts
- Traps
- Faults
- Aborts
(1) Interrupts

**Cause:** Signal from I/O device

**Examples:**
- User presses key
- Disk controller finishes reading/writing data
- Timer to trigger another application to run

An alternative to wasteful polling!
(2) Traps

**Cause:** Intentional (application program requests OS service)

**Examples:**
- Application program requests more heap memory
- Application program requests I/O

Traps provide a function-call-like interface between application and OS.
(3) Faults

**Cause:** Application program causes (possibly) recoverable error

**Examples:**
- Application program accesses privileged memory (segmentation fault)
- Application program accesses data that is not in real memory (page fault)
(4) Aborts

**Cause:** Non-recoverable error

**Example:**
Parity check indicates corruption of memory bit (overheating, cosmic ray!, etc.)
## Summary of Exception Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Cause</th>
<th>Asynch/Synch</th>
<th>Return Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt</strong></td>
<td>Signal from I/O device</td>
<td>Asynch</td>
<td>Return to next instr</td>
</tr>
<tr>
<td><strong>Trap</strong></td>
<td>Intentional</td>
<td>Sync</td>
<td>Return to next instr</td>
</tr>
<tr>
<td><strong>Fault</strong></td>
<td>(Maybe) recoverable error</td>
<td>Sync</td>
<td>(Maybe) return to current instr</td>
</tr>
<tr>
<td><strong>Abort</strong></td>
<td>Non-recoverable error</td>
<td>Sync</td>
<td>Do not return</td>
</tr>
</tbody>
</table>
Exceptions in Intel Processors

Each exception has a number
Some exceptions in Intel processors:

<table>
<thead>
<tr>
<th>Exception #</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Fault: Divide error</td>
</tr>
<tr>
<td>13</td>
<td>Fault: Segmentation fault</td>
</tr>
<tr>
<td>14</td>
<td>Fault: Page fault (see “Virtual Memory” lecture)</td>
</tr>
<tr>
<td>18</td>
<td>Abort: Machine check</td>
</tr>
<tr>
<td>32-127</td>
<td>Interrupt or trap (OS-defined)</td>
</tr>
<tr>
<td>128</td>
<td>Trap</td>
</tr>
<tr>
<td>129-255</td>
<td>Interrupt or trap (OS-defined)</td>
</tr>
</tbody>
</table>
Traps in Intel Processors

• To execute a trap, application program should:
  • Place number in EAX register indicating desired functionality
  • Place parameters in EBX, ECX, EDX registers
  • Execute assembly language instruction “int 128”

• Example: To request more heap memory…

movl $45, %eax
movl $1024, %ebx
int $128

In Linux, 45 indicates request for more heap memory

Request is for 1024 bytes
Causes trap
System-Level Functions

• For convenience, traps are wrapped in **system-level functions**

• Example: To request more heap memory...

```c
/* unistd.h */
void *sbrk(intptr_t increment);
```

```c
/* unistd.s */
Defines sbrk() in assembly lang
Executes int instruction
```

```c
/* client.c */
... sbrk(1024);
```

A call of a system-level function, that is, a **system call**

See Appendix for list of some Linux system-level functions
Processes

- **Program**
  - Executable code

- **Process**
  - An instance of a program in execution

- Each program runs in the **context** of some process

- **Context** consists of:
  - Process ID
  - Address space
    - TEXT, RODATA, DATA, BSS, HEAP, and STACK
  - Processor state
    - EIP, EFLAGS, EAX, EBX, etc. registers
  - Etc.
Significance of Processes

• **Process** is a profound abstraction in computer science

• The process abstraction provides application pgms with two key illusions:
  
  • Private control flow
  • Private address space
Hardware and OS give each application process the illusion that it is the only process running on the CPU.
Private Control Flow: Reality

All application processes -- and the OS process -- share the same CPU(s)
Context Switches

- **Context switch**
  - The activity whereby the OS assigns the CPU to a different process
  - Occurs during exception handling, at discretion of OS

- Exceptions can be caused:
  - Synchronously, by application pgm (trap, fault, abort)
  - Asynchronously, by external event (interrupt)
  - Asynchronously, by hardware timer
    - So no process can dominate the CPUs

- Exceptions are the mechanism that enables the illusion of private control flow
Context Switch Details

- **Context**
  - State the OS needs to restart a preempted process

- **Context switch**
  - Save the context of current process
  - Restore the saved context of some previously preempted process
  - Pass control to this newly restored process
When Should OS Do Context Switch?

- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk access

1: CPU I/O CPU I/O CPU I/O

2: CPU I/O CPU I/O CPU I/O

- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine
  - Trade-off efficiency for a finer granularity of fairness
Life Cycle of a Process

- **Running**: instructions are being executed
- **Waiting**: waiting for some event (e.g., I/O finish)
- **Ready**: ready to be assigned to a processor

![Diagram showing the life cycle of a process: Create → Ready → Running → Termination. The diagram also includes a cycle from Running back to Ready via Waiting.](image-url)
Context Switch: What Context to Save?

- **Process state**
  - New, ready, waiting, terminated

- **CPU registers**
  - EIP, EFLAGS, EAX, EBX, ...

- **I/O status information**
  - Open files, I/O requests, ...

- **Memory management information**
  - Page tables (see “Virtual Memory” lecture)

- **Accounting information**
  - Time limits, group ID, ...

- **CPU scheduling information**
  - Priority, queues
Private Address Space: Illusion

Hardware and OS give each application process the illusion that it is the only process using memory.
Private Address Space: Reality

Memory is divided into pages

All processes use the same real memory
Hardware and OS provide application pgms with a virtual view of memory, i.e. virtual memory (VM)
Private Address Space Details

• Exceptions (specifically, page faults) are the mechanism that enables the illusion of private address spaces

• See the Virtual Memory lecture for details
Summary

- **Exception**: an abrupt change in control flow
  - **Interrupts**: asynchronous; e.g. I/O completion, hardware timer
  - **Traps**: synchronous; e.g. app pgm requests more heap memory, I/O
  - **Faults**: synchronous; e.g. seg fault
  - **Aborts**: synchronous; e.g. parity error

- **Process**: An instance of a program in execution
  - Hardware and OS use exceptions to give each process the illusion of:
    - Private control flow (reality: context switches)
    - Private address space (reality: virtual memory)
### Linux system-level functions for I/O management

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| 3      | `read()` | Read data from file descriptor  
|        |          | Called by `getchar()`, `scanf()`, etc. |
| 4      | `write()` | Write data to file descriptor  
|        |          | Called by `putchar()`, `printf()`, etc. |
| 5      | `open()` | Open file or device  
|        |          | Called by `fopen()` |
| 6      | `close()` | Close file descriptor  
|        |          | Called by `fclose()` |
| 8      | `creat()` | Open file or device for writing  
|        |          | Called by `fopen(..., "w")` |

Described in I/O Management lecture
### Appendix: System-Level Functions

Linux system-level functions for **process management**

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>exit()</td>
<td>Terminate the process</td>
</tr>
<tr>
<td>2</td>
<td>fork()</td>
<td>Create a child process</td>
</tr>
<tr>
<td>7</td>
<td>waitpid()</td>
<td>Wait for process termination</td>
</tr>
<tr>
<td>7</td>
<td>wait()</td>
<td>(Variant of previous)</td>
</tr>
<tr>
<td>11</td>
<td>exec()</td>
<td>Execute a program in current process</td>
</tr>
<tr>
<td>20</td>
<td>getpid()</td>
<td>Get process id</td>
</tr>
</tbody>
</table>

Described in **Process Management** lecture
## Appendix: System-Level Functions

Linux system-level functions for I/O redirection and inter-process communication

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td><code>dup()</code></td>
<td>Duplicate an open file descriptor</td>
</tr>
<tr>
<td>42</td>
<td><code>pipe()</code></td>
<td>Create a channel of communication between processes</td>
</tr>
<tr>
<td>63</td>
<td><code>dup2()</code></td>
<td>Close an open file descriptor, and duplicate an open file descriptor</td>
</tr>
</tbody>
</table>

Described in *Process Management* lecture
Appendix: System-Level Functions

Linux system-level functions for *dynamic memory management*

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>brk()</td>
<td>Move the program break, thus changing the amount of memory allocated to the HEAP</td>
</tr>
<tr>
<td>45</td>
<td>sbrk()</td>
<td>(Variant of previous)</td>
</tr>
<tr>
<td>90</td>
<td>mmap()</td>
<td>Map a virtual memory page</td>
</tr>
<tr>
<td>91</td>
<td>munmap()</td>
<td>Unmap a virtual memory page</td>
</tr>
</tbody>
</table>

Described in *Dynamic Memory Management* lectures
## Appendix: System-Level Functions

### Linux system-level functions for **signal handling**

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td><code>alarm()</code></td>
<td>Deliver a signal to a process after a specified amount of wall-clock time</td>
</tr>
<tr>
<td>37</td>
<td><code>kill()</code></td>
<td>Send signal to a process</td>
</tr>
<tr>
<td>67</td>
<td><code>sigaction()</code></td>
<td>Install a signal handler</td>
</tr>
<tr>
<td>104</td>
<td><code>setitimer()</code></td>
<td>Deliver a signal to a process after a specified amount of CPU time</td>
</tr>
<tr>
<td>126</td>
<td><code>sigprocmask()</code></td>
<td>Block/unblock signals</td>
</tr>
</tbody>
</table>

Described in **Signals** lecture