Exceptions and Processes

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 application programs interact with operating systems and hardware

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

Second half of the course

<table>
<thead>
<tr>
<th>Starting Now</th>
<th>Later</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Program</td>
<td>C Language</td>
</tr>
<tr>
<td>Operating System</td>
<td>Assembly Language</td>
</tr>
<tr>
<td>Hardware</td>
<td>Machine Language</td>
</tr>
</tbody>
</table>

Application programs, OS, and hardware interact via exceptions

Context of this Lecture (cont.)

• More precisely…
• First (now):
  • Service levels tour
    • Exceptions and processes
    • Memory management
• Then:
  • Language levels tour
    • Assembly language
    • Machine language
    • Assemblers and linkers
• And then:
  • Service levels tour (continued)
    • Process management, I/O management, signals
Motivation

Question:
- Executing program thinks it has exclusive control of CPU
- But multiple executing programs must share CPU(s)
- How is that illusion implemented?

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

Answer: Exceptions and Processes

Exceptions

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

- **Synchronous exceptions:**
  - Caused by something the application program does:
    - 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)

- **Asynchronous exceptions**
  - Caused by something not initiated by the application program
    - User: presses key on keyboard
    - Hardware: 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 processing
  - exception return (optional)
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
  - Processor pushes data onto OS’s, not application program’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
    - Sometimes, control returns to current instruction
    - Sometimes, control does not return at all

Classes of Exceptions

- There are 4 classes of exceptions…
(1) Interrupts

Application program

Exception handler

(1) CPU interrupt pin goes high
(2) After current instr finishes, control passes to handler
(3) Handler runs
(4) Handler returns control to next instr

Cause: Signal from I/O device
Examples:
User presses key
Disk controller finishes reading/writing data

(2) Traps

Application program

Exception handler

(1) Application pgm traps
(2) Control passes to handler
(3) Handler runs
(4) Handler returns control to next instr

Cause: 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 (seg 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>Interrupt</td>
<td>Signal from I/O device</td>
<td>Asynch</td>
<td>Return to next instr</td>
</tr>
<tr>
<td>Trap</td>
<td>Intentional</td>
<td>Sync</td>
<td>Return to next instr</td>
</tr>
<tr>
<td>Fault</td>
<td>(Maybe) recoverable error</td>
<td>Sync</td>
<td>(Maybe) return to current instr</td>
</tr>
<tr>
<td>Abort</td>
<td>Non-recoverable error</td>
<td>Sync</td>
<td>Do not return</td>
</tr>
</tbody>
</table>

## Exceptions in Intel Processors

Every 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...

```
/* unistd.h */
void *sbrk(intptr_t increment);
... 
/* unistd.s */
Defines sbrk() in assembly lang
Executes int instruction
... 
/* client.c */
... 
sbrk(1024);
... 
```

sbrk() is a system-level function

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

• When a program “runs”, it 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

• The process abstraction provides application programs with two key illusions:
  • Private control flow
  • Private address space
Private Control Flow: Illusion

Hardware and OS give each application process the illusion that it is the only process running on the CPU(s)

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

![Diagram showing context switch process](image-url)
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
Context Details

- What does the OS need to save/restore during a context switch?
  - 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

Illusion: Private Address Space

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

All processes use the same real memory
Hardware and OS provide application programs 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, hw timer
  - **Traps**: synchronous; e.g. app requests heap memory, I/O
  - **Faults**: synchronous; e.g. segmentation 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 switching)
    - Private address space (reality: virtual memory)

Appendix: System-Level Functions

Linux system-level functions for I/O management

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

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

Described in *Process Management* lecture

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

Described in *Signals* lecture