Exceptions and Processes

Jennifer Rexford

Much of 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 system programming

Context of this Lecture

Second half of the course

Previously

C Language

Assembly Language

Machine Language

language levels tour

Starting Now

Application Program

Operating System

service levels tour

Hardware

Application programs, OS, and hardware interact via exceptions

Motivation

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

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

Answers: Exceptions…

Agenda

Exceptions

Processes

Illusion: Private address space

Illusion: Private control flow

Exceptions

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

Some exceptions are **synchronous**
- Occur as result of actions of executing program
- Examples: Synchronous exception occurs when:
  - Application program requests I/O
  - Application program requests more heap memory
  - Application program attempts integer division by 0
  - Application program attempts to access privileged memory
  - Application program accesses variable that is not in physical memory
  - See later in this lecture
  - See upcoming Virtual Memory lecture

Asynchronous Exceptions

Some exceptions are **asynchronous**
- Do not occur (directly) as result of actions of executing program
- Examples: Asynchronous exception occurs when:
  - User presses key on keyboard
  - Disk controller finishes reading data
  - Hardware timer expires

Exceptions Note

Note:
Exceptions in OS ≠ exceptions in Java
- Implemented using try/catch and throw statements

Exceptional Control Flow

![Diagram showing the flow of control between an application program and an exception handler in the operating system.]

Exceptions vs. Function Calls

Handling an exception is **similar to** calling a function
- CPU pushes arguments onto stack
- Control transfers from original code to other code
- Other code executes
- Control returns to some instruction in original code

Handling an exception is **different from** calling a function
- CPU pushes additional data onto stack
- E.g., values of all registers
- CPU pushes data onto OS’s stack, not application program’s stack
- Handler runs in kernel/privileged mode, not in user mode
- Handler can execute all instructions and access all memory
- Control might return to some instruction in original code
- Sometimes control returns 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 exception handler
3. Exception handler runs
4. Exception handler returns control to next instr

**Occurs when**: External (off-CPU) device requests attention
**Examples**: User presses key, Disk controller finishes reading/writing data, Hardware timer expires

(2) Traps

- **Application program**
- **Exception handler**

1. Application pgm traps
2. Control passes to exception handler
3. Exception handler runs
4. Exception handler returns control to next instr

**Occurs when**: Application pgm requests OS service
**Examples**: Application pgm requests I/O, Application pgm requests heap memory
**Traps provide function-call-like interface between application pgm and OS**

(3) Faults

- **Application program**
- **Exception handler**

1. Current instr causes a fault
2. Control passes to exception handler
3. Exception handler runs
4. Exception handler returns control to current instr, or aborts

**Occurs when**: Application pgm causes a (possibly recoverable) error
**Examples**: Application pgm divides by 0, Application pgm accesses privileged memory (seg fault), Application pgm accesses data that is not in physical memory (page fault)

(4) Aborts

- **Application program**
- **Exception handler**

1. Fatal hardware error occurs
2. Control passes to exception handler
3. Exception handler runs
4. Exception handler aborts execution

**Occurs when**: HW detects a 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>Occurs when</th>
<th>Asynch/Synch</th>
<th>Return Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>External device requests attention</td>
<td>Asynch</td>
<td>Return to next instr</td>
</tr>
<tr>
<td>Trap</td>
<td>Application pgm requests OS service</td>
<td>Sync</td>
<td>Return to next instr</td>
</tr>
<tr>
<td>Fault</td>
<td>Application pgm causes (maybe recoverable) error</td>
<td>Sync</td>
<td>Return to current instr (maybe)</td>
</tr>
<tr>
<td>Abort</td>
<td>HW detects non-recoverable error</td>
<td>Sync</td>
<td>Do not return</td>
</tr>
</tbody>
</table>

### Aside: Exceptions in x86-64 Processors

Each exception has a number
Some exceptions in x86-64 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>32-255</td>
<td>OS-defined exceptions</td>
</tr>
</tbody>
</table>
Aside: Traps in x86-64 Processors

To execute a trap, application program should:
• Place number in RAX register indicating desired OS service
• Place arguments in RDI, RSI, RDX, RCX, R8, R9 registers
• Execute assembly language instruction syscall

Example: To request change in size of heap section of memory (see Dynamic Memory Management lecture)...

```assembly
movq $12, %rax
movq $newAddr, %rdi
syscall
```

Place 12 (change size of heap section) in RAX
Place new address of end of heap in RDI
Execute trap

Aside: System-Level Functions

Traps are wrapped in system-level functions
Example: To change size of heap section of memory...

```c
/* unistd.h */
int brk(void *addr);
```

```assembly
define brk() in assembly lang
Executes syscall instruction
```

```c
/* client.c */
brk(newAddr);
```

A call of a system-level function, that is, a system call
See Appendix for some Linux system-level functions

Agenda

Exceptions
Processes
Illusion: Private address space
Illusion: Private control flow

Processes

Program
• Executable code
• A static entity
Process
• An instance of a program in execution
• A dynamic entity: has a time dimension
• Each process runs one program
  • E.g. process 12345 might be running emacs
• One program can run in multiple processes
  • E.g. Process 12345 might be running emacs, and process 54321 might also be running emacs – for the same user or for different users

Processes Significance

Process abstraction provides application pgms with two key illusions:
• Private address space
• Private control flow

Process is a profound abstraction in computer science

Agenda

Exceptions
Processes
Illusion: Private address space
Illusion: Private control flow
Private Address Space: Illusion

Hardware and OS give each application process the illusion that it is the only process using memory.

Process X
----------
| Memory for Process X |
----------

Process Y
----------
| Memory for Process Y |
----------

Private Address Space: Reality

All processes use the same physical memory. Hardware and OS provide application programs with a virtual view of memory, i.e., virtual memory (VM).

Private Address Space: Implementation

Question:
- How do the CPU and OS implement the illusion of private address space?
- That is, how do CPU and OS implement virtual memory?

Answer:
- Exceptions!
- Specifically, page faults
- Overview now, details next lecture...

Private Address Space Example 1

Private Address Space Example 2

Agenda

Exceptions
Processes
- Illusion: Private address space
- Illusion: Private control flow

Exceptions (specifically, page faults) enable the illusion of private address spaces.
Private Control Flow: Illusion

Simplifying assumption: only one CPU

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

Private Control Flow: Reality

Multiple processes share the CPU

Multiple processes run concurrently

OS occasionally preempts running process

Process Status

More specifically…

At any time a process has status:
- **Running**: CPU is executing process’s instructions
- **Ready**: Process is ready for OS to assign it to the CPU
- **Blocked**: Process is waiting for some requested service (typically I/O) to finish

Process Status Transitions

- **Service requested**: OS moves running process to blocked set because it requested a (time consuming) system service (often I/O)
- **Service finished**: OS moves blocked process to ready set because the requested service finished
- **Time slice expired**: OS moves running process to ready set because process consumed its fair share of CPU time
- **Scheduled for execution**: OS selects some process from ready set and assigns CPU to it

Process Status Transitions Over Time

Throughout its lifetime a process’s status switches between running, ready, and blocked

Private Control Flow: Implementation (1)

Question:
- How do CPU and OS implement the illusion of private control flow?
- That is, how to CPU and OS implement process status transitions?

Answer (Part 1):
- Contexts and context switches…
Process Contexts

Each process has a context
- The process’ state, that is...
- Register contents
  - RIP, EFLAGS, RDI, RSI, etc. registers
- Memory contents
  - TEXT, RODATA, DATA, BSS, HEAP, and STACK

Aside: Process Control Blocks

Question:
- Where does OS save a process’ context?

Answer:
- In its process control block (PCB)

Process control block (PCB)
- A data structure
- Contains all data that OS needs to manage the process

Context Switch

Context switch:
- OS saves context of running process
- OS loads context of some ready process
- OS passes control to newly restored process

Aside: Process Control Block Details

Process control block (PCB):

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Unique integer assigned by OS when process is created</td>
</tr>
<tr>
<td>Status</td>
<td>Running, ready, or waiting</td>
</tr>
</tbody>
</table>
| Hierarchy| ID of parent process  
|          | ID of child processes (if any)  
|          | (See Process Management Lecture)                     |
| Priority | High, medium, low                                    |
| Time consumed | Time consumed within current time slice           |
| Context  | When process is not running...  
| Etc.     | Contents of all registers  
| Etc.     | (In principle) contents of all of memory            |

Context Switch Efficiency

Observation:
- During context switch, OS must:
  - Save context (register and memory contents) of running process to its PCB
  - Restore context (register and memory contents) of some ready process from its PCB

Question:
- Isn’t that very expensive (in terms of time and space)?

Answer:
- Not really!
- During context switch, OS does **not** save/load memory contents
  - But there are few registers
- During context switch, OS does not save/load memory contents
  - Each process has a page table that maps virtual memory pages to physical memory pages
  - During context switch, need only deactivate process X page table and activate process Y page table
  - See Virtual Memory lecture
Private Control Flow: Implementation (2)

**Question:**
- How do CPU and OS implement the illusion of private control flow?
- That is, how do CPU and OS implement process status transitions?
- That is, how do CPU and OS implement context switches?

**Answer (Part 2):**
- Exceptions!
- Context switches occur while the OS handles exceptions...

Exceptions and Context Switches

Exceptions occur frequently
- Process explicitly requests OS service (trap)
- Service request fulfilled (interrupt)
- Process accesses VM page that is not in physical memory (fault)
- Etc.
- ... And if none of them occur for a while ...
- Expiration of hardware timer (interrupt)

Whenever OS gains control of CPU via exception...
It has the option of performing context switch

Private Control Flow Example 1

- Process X is running
- Hardware clock generates interrupt
- OS gains control of CPU
- OS examines “time consumed” field in process X’s PCB
- OS decides to do context switch
  - OS saves process X’s context in its PCB
  - OS sets “status” field in process X’s PCB to ready
  - OS loads process Y’s context from its PCB
  - Process Y is running

Private Control Flow Example 2

- Process Y is running
  - Process Y executes trap to request read from disk
  - OS gains control of CPU
  - OS decides to do context switch
  - OS saves process Y’s context in its PCB
  - OS sets “status” field in process Y’s PCB to blocked
  - OS adds process Y’s PCB to the blocked set
  - OS removes process X’s PCB from the ready set
  - OS sets “status” field in process X’s PCB to running
  - OS loads process X’s context from its PCB
  - Process X is running

Private Control Flow Example 3

- Process X is running
  - Read operation requested by process Y completes => disk controller generates interrupt
  - OS gains control of CPU
  - OS sets “status” field in process Y’s PCB to ready
  - OS moves process Y’s PCB from the blocked list to the ready list
  - OS examines “time consumed within slice” field of process X’s PCB
  - OS decides not to do context switch
  - Process X is running
Private Control Flow Example 4

Exceptions enable the illusion of private control flow

- Process X is running
- Process X accesses memory, generates page fault
- OS gains control of CPU
- OS swicths page from memory to disk, loads referenced page from disk to memory
- OS examines "time consumed" field of process X's PCB
- OS decides not to do context switch
- Process X is running

Summary

Exception: an abrupt change in control flow
- Interrupt: asynchronous; e.g. I/O completion, hardware timer
- Trap: synchronous; e.g. app pgm requests more heap memory, I/O
- Fault: synchronous; e.g. seg fault, page fault
- Abort: synchronous; e.g. failed parity check

Process: An instance of a program in execution
- CPU and OS give each process the illusion of:
  - Private address space
  - Reality: virtual memory
  - Private control flow
  - Reality: Concurrency, preemption, and context switches
- Both illusions are implemented using exceptions

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>0</td>
<td>read()</td>
<td>Read data from file descriptor; called by getchar(), scanf(), etc.</td>
</tr>
<tr>
<td>1</td>
<td>write()</td>
<td>Write data to file descriptor; called by putchar(), printf(), etc.</td>
</tr>
<tr>
<td>2</td>
<td>open()</td>
<td>Open file or device; called by fopen()</td>
</tr>
<tr>
<td>3</td>
<td>close()</td>
<td>Close file descriptor; called by fclose()</td>
</tr>
<tr>
<td>85</td>
<td>creat()</td>
<td>Open file or device for writing; called by fopen(&quot;w&quot;, ...)</td>
</tr>
<tr>
<td>8</td>
<td>lseek()</td>
<td>Position file offset; called by fseek()</td>
</tr>
</tbody>
</table>

Described in I/O Management lecture

Linux system-level functions for process management

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>exit()</td>
<td>Terminate the current process</td>
</tr>
<tr>
<td>57</td>
<td>fork()</td>
<td>Create a child process</td>
</tr>
<tr>
<td>7</td>
<td>wait()</td>
<td>Wait for child process termination</td>
</tr>
<tr>
<td>11</td>
<td>execvp()</td>
<td>Execute a program in the current process</td>
</tr>
<tr>
<td>20</td>
<td>getpid()</td>
<td>Return the process id of the current process</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>32</td>
<td>dup()</td>
<td>Duplicate an open file descriptor</td>
</tr>
<tr>
<td>22</td>
<td>pipe()</td>
<td>Create a channel of communication between processes</td>
</tr>
</tbody>
</table>

Described in Process Management lecture

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>12</td>
<td>brk()</td>
<td>Move the program break, thus changing the amount of memory allocated to the HEAP</td>
</tr>
<tr>
<td>12</td>
<td>sbrk()</td>
<td>(Variant of previous)</td>
</tr>
<tr>
<td>9</td>
<td>mmap()</td>
<td>Map a virtual memory page</td>
</tr>
<tr>
<td>11</td>
<td>munmap()</td>
<td>Unmap a virtual memory page</td>
</tr>
</tbody>
</table>

Described in Dynamic Memory Management lecture
### 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>37</td>
<td>alarm()</td>
<td>Deliver a signal to a process after a specified amount of wall-clock time</td>
</tr>
<tr>
<td>62</td>
<td>kill()</td>
<td>Send signal to a process</td>
</tr>
<tr>
<td>13</td>
<td>sigaction()</td>
<td>Install a signal handler</td>
</tr>
<tr>
<td>38</td>
<td>setitimer()</td>
<td>Deliver a signal to a process after a specified amount of CPU time</td>
</tr>
<tr>
<td>14</td>
<td>sigprocmask()</td>
<td>Block/unblock signals</td>
</tr>
</tbody>
</table>

Described in *Signals* lecture