Exceptions and Processes

Much of the material for this lecture is drawn from Computer Systems: A Programmer’s Perspective (Bryant & O’Hallaron) Chapter 8.

Time sharing
Just one CPU, but each program appears to have its own CPU

Application program 1

Application program 2

10 milliseconds

Memory sharing
Just one memory, but each program appears to have its own memory

Application program 1

Application program 2

Device sharing
Just one keyboard, but each program appears to have its own keyboard

Application program 1

Application program 2

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

Starting Now
- Application Program
- Operating System
- Hardware

Application programs, OS, and hardware interact via exceptions.
Agenda

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

Example Program

```c
int f(char *p, int n) {
    int i;
    for (i=0; i<n; i++)
        fputc(p[i], stdout);
}
```

Multiple processes but only 1 register bank!

Normal execution

Exception! (timer interrupt)
Copy registers to OS memory

Copy registers to OS memory (for process 2)

System call!

System call!

Copy registers to OS memory

Now executing in the O.S. "process"

Exceptions

Synchronous Exceptions

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

Some exceptions are synchronous
• Occur as result of actions of executing program
• Examples:
  • System call: Application requests I/O
  • System call: Application requests more heap memory
  • Application pgm attempts integer division by 0
  • Application pgm attempts to access privileged memory
  • Application pgm 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:
  - 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

Application program → Exception handler in operating system

Exception handler

Exception return (sometimes)

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
- 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 prog’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 same 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 instruction

Traps occur when:
- Application program requests OS service
- Application program requests more heap memory

Traps provide a function-call-like interface between application program and OS.

(3) Faults

Application program

Exception handler

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

Faults occur when:
- Application program causes a (possibly recoverable) error
- Application program accesses privileged memory (seg fault)
- Application program accesses data that is not in physical memory (page fault)

(4) Aborts

Application program

Exception handler

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

Aborts occur 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</th>
<th>Return Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>External device requests attention</td>
<td></td>
<td>Return to next instruction</td>
</tr>
<tr>
<td>Trap</td>
<td>Application program requests OS service</td>
<td>Sync</td>
<td>Return to next instruction</td>
</tr>
<tr>
<td>Fault</td>
<td>Application program causes (maybe recoverable) error</td>
<td>Sync</td>
<td>Return to current instruction (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: 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:

```
movq $12, trax
movq $newAddr, tdi
syscall
```

Aside: System-Level Functions

Traps are wrapped in system-level functions

Example:

- To change size of heap section of memory...

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

/* unistd.s */
brk:
    movq $12, trax
    movq $newAddr, tdi
    syscall
    ret

/* 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

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

All processes use the same physical memory
Hardware and OS provide application pgms 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 the CPU and OS implement virtual memory?

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

Private Address Space Examples

Private Address Space Example 1
- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU checks if required virtual page is in physical memory: yes
- CPU does load/store from/to physical memory

Private Address Space Example 2
- Process executes instruction that references virtual memory
- CPU determines virtual page
- CPU checks if required virtual page is in physical memory: no
- CPU generates a page fault

Page Fault (Exception)

Exceptions (specifically, page faults) enable the illusion of private address spaces

Agenda

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

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’s state, that is...
  - Register contents
    - RIP, EFLAGS, RDI, RSI, etc. registers
  - Memory contents
    - TEXT, RODATA, DATA, BSS, HEAP, and STACK

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 Blocks

Question:
- Where does OS save a process’s 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

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>
<tr>
<td>Hierarchy</td>
<td>ID of parent process; ID of child processes (if any)</td>
</tr>
<tr>
<td>Priority</td>
<td>High, medium, low</td>
</tr>
<tr>
<td>Time consumed</td>
<td>Time consumed within current time slice</td>
</tr>
<tr>
<td>Context</td>
<td>When process is not running... Contents of all registers (In principle)</td>
</tr>
<tr>
<td>Etc.</td>
<td>Content of all of memory</td>
</tr>
</tbody>
</table>

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 save/load register contents
  - But there are few registers
  - 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 they implement context 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

Context switches occur while OS is handling 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
- OS
- Process Y

(1) Process X is running
- OS gains control of CPU
- OS examines "time consumed" field in X's PCB
- OS decides to context switch

(2) OS executes trap, sets its status field to "running"
- Sets X's PCB to "ready"
- Adds X's PCB to ready set

(3) OS removes Y's PCB from ready set, sets its status field to "running"
- Loads Y's context from its PCB

Context Switch

(4) Process Y is running

Private Control Flow Example 2

Process X
- OS
- Process Y

(1) Process X is running
- OS decides to context switch

(2) OS removes Y's PCB from ready set, sets its status field to "ready"
- Sets X's PCB to "running"
- Loads X's context from its PCB

Context Switch

(3) Process X keeps running

Private Control Flow Example 3

Process X
- OS
- Process Y

(1) Process X is running
- OS gains control of CPU
- OS examines "time consumed" field in X's PCB
- OS decides to context switch

(2) OS removes Y's PCB from ready set, sets its status field to "ready"
- Loads Y's context from its PCB

Context Switch

(3) Process X keeps running

Private Control Flow Example 4

Process X
- OS
- Process Y

(1) Process X is running
- Process X accesses memory, generates page fault
- OS gains control of CPU
- OS removes X's PCB from ready set, sets its status field to "ready"
- Sets X's PCB to "running"
- Loads X's context from its PCB

Context Switch

(2) Process X keeps 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 interprocess 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

#### 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