CS444/544
Operating Systems II

Lecture 8
User/Kernel Context Switch
4/29/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Today’s Topic

• User/Kernel Space Switch
  • How does the OS kernel run a program in Ring 3 (user level)?
  • How does the OS kernel take back the execution to Ring 0 (kernel)?

• System call
  • How could a user level program let OS serve for them?
Today’s Topic

- Process Context Switch
  - How could our CPU run multiple applications at the same time?

- 3 design candidates
  - Not switching
  - Co-operative Multitasking
  - Preemptive Multitasking
Today’s Topic

- User/Kernel Space Switch
  - Interrupt
  - System calls
  - Fault / Exceptions
Kernel (Ring 0)

- Runs with the highest privilege level (Ring 0)
- Configures system (devices, memory, etc.)
- Manages hardware resources
  - Disk, memory, network, video, keyboard, etc.
- Manages other jobs
  - Processes and threads
- Serves as trusted computing base (TCB)
  - Set privilege
  - Restrict other jobs from doing something bad.
**User (Ring 3)**

- Runs with a restricted privilege (Ring 3)
  - The privilege level for running an application...
- Most of regular applications runs in this level

- Cannot access kernel memory
  - Can only access pages set with PTE_U
- Cannot talk directly to hardware devices
  - Kernel must mediate the access
A High-level Overview of User/Kernel Execution

```c
int main() {
  printf("CS444");
}
```
A High-level Overview of User/Kernel Execution

printf("CS444")
A library call in ring 3

sys_write(1, "CS444", 5);
A system call, From ring 3
Interrupt!, switch from ring 3 to ring 0

A kernel function
do_sys_write(1, "CS444", 5)
A High-level Overview of User/Kernel Execution

- **printf(“CS444”)**
  - A library call in ring 3

- **sys_write(1, “CS444”, 5);**
  - A system call, From ring 3
  - Interrupt!, switch from ring3 to ring0

- **do_sys_write(1, “CS444”, 5)**
  - A kernel function

```c
int main() {
    printf("CS444");
}
```
A Library Call

• A function call within the application’s memory space

• All regular C/C++ API calls are library calls
  • fwrite(), printf(), time(), srand(), etc.
  • Calls that you did not implement but prepared by others (in ring 3)

• From Ring 3 to Ring 3
A System Call

• A function call from applications that request OS to do something special for them

• System APIs
  • I/O access (read(), write(), send(), recv(), etc.)
  • Process creation, destruction (exec(), fork(), kill(), etc.)
  • Other hardware access..

• From Ring 3 to Ring 0

A system call, From ring 3 to ring 0

printf()
Returning from a Call

- Returning from a Library Call
  - `ret`
  - No ring switch (ring 3 -> ring 3)

- Returning from a System Call
  - `iret` (interrupt return)
  - Ring switch happens (ring 0 -> ring 3)
A High-level Overview of User/Kernel Execution

```c
int main() {
    send(4, "I have a question...", 30, 0);
}
```

- **send()**: A library call in ring 3
- **sys_send()**: A system call, From ring 3 to ring 0
- **do_sys_send()**: Interrupt!
A High-level Overview of User/Kernel Execution

```
int main() {
    send(4, "I have a question...", 30, 0);
}
```

A library call in ring 3

```
sys_send()
```

A system call, From ring 3 to ring 0

```
do_sys_send()
```

iret (ring 0 to ring 3)

ret (ring 3)
How does Kernel Execute an Application?

Lab1: Booting
Lab2: Set VM
Lab3: Set kernel/user env

How does an OS run an application?
How does Kernel Execute an Application?
How does Kernel Execute an Application?

1. Prepare a process, an environment for running an application

Assign a separated Virtual Memory Space
New page directory
New page table
Etc..

OS Kernel (Ring 0)
How does Kernel Execute an Application?

1. Prepare a process, an environment for running an application

2. Put an application! load code!
How does Kernel Execute an Application?

1. Prepare a process, an environment for running an application

2. Put an application! Load code!

3. Execute!
How does Kernel Execute an Application?

1. Prepare a process, an environment for running an application

2. Put an application! load code!

3. Execute!
How does Kernel Get the Execution Back?

Ring 3

OS Kernel (Ring 0)
How does Kernel Get the Execution Back?

```c
int main() {
    printf("CS444");
}
```
How does Kernel Get the Execution Back?

System call!

sys_write()
How does Kernel Get the Execution Back?

System call!

\texttt{sys\_write()}

\texttt{do\_sys\_write()}

\begin{verbatim}
int main() {
    printf("CS444");
}
\end{verbatim}
Is System Call the Only Way to Execute in Kernel?

• No
  • In such a case, we have lots of problems..
  • E.g., kernel waits until an application runs a system call
  • What if an application never calls a system call????

• We have the following ways to switch
  • System call (ring 3 -> ring 0)
  • Interrupt (usually runs in ring 0, sometimes runs in ring 3)
  • Fault/Exception (runs in ring 0)
User Execution Strawman 1

• Just run user application

• Seems OK, but...

iret (ring 0 to ring 3)

OS Kernel (Ring 0)
User Execution Strawman 1’

• Just run user application

• What happens if we run 2 applications at the same time?

• How can we switch execution?
User Execution Strawman 2

• Co-operative Multitasking

• Yield()
  • Surrender the execution right when a process finishes / pauses its execution
User Execution Strawman 2

• Co-operative Multitasking

• Yield()
  • Surrender the execution right when a process finishes / pauses its execution

• Schedule()
  • Execute a different process...

OS Kernel (Ring 0)

iret (ring 0 to ring 3)

Schedule()
User Execution Strawman 2’

• What if a process runs

```c
int main() {
    while(1);
}
```
User Execution Strawman 2’

• What if a process runs

```c
int main() {
    while(1);
}
```
User Execution Strawman 3

• Preemptive Multitasking (Lab 4)

• CPU generates an interrupt to force execution at kernel after some time quantum
  • E.g., 1000Hz, on each 1ms..
User Execution Strawman 3

- Preemptive Multitasking (Lab 4)

- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..

- Guaranteed execution in kernel
  - Let kernel mediate resource contention
User Execution Strawman 3

• Preemptive Multitasking (Lab 4)

• CPU generates an interrupt to force execution at kernel after some time quantum
  • E.g., 1000Hz, on each 1ms..

• Guaranteed execution in kernel
  • Let kernel mediate resource contention
How are Popular OSes doing?

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Preemption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amiga OS</td>
<td>Yes</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Yes</td>
</tr>
<tr>
<td>Linux kernel before 2.6.0</td>
<td>Yes</td>
</tr>
<tr>
<td>Linux kernel 2.6.0–2.6.23</td>
<td>Yes</td>
</tr>
<tr>
<td>Linux kernel after 2.6.23</td>
<td>Yes</td>
</tr>
<tr>
<td>classic Mac OS pre-9</td>
<td>None</td>
</tr>
<tr>
<td>Mac OS 9</td>
<td>Some</td>
</tr>
<tr>
<td>macOS</td>
<td>Yes</td>
</tr>
<tr>
<td>NetBSD</td>
<td>Yes</td>
</tr>
<tr>
<td>Solaris</td>
<td>Yes</td>
</tr>
<tr>
<td>Windows 3.1x</td>
<td>None</td>
</tr>
<tr>
<td>Windows 95, 98, Me</td>
<td>Half</td>
</tr>
<tr>
<td>Windows NT (including 2000, XP, Vista, 7, and Server)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Trap: Interrupt/Faults/Exception

• Trap
  • An event that forces CPU to execute (some) code in kernel
  • Will run trap handler

• Interrupts
  • Hardware interrupt
  • System call (software interrupt)

• Faults
  • An error that OS may recover and continue execution (e.g., page fault)

• Exception
  • An error that OS cannot recover and must stop the current execution (e.g., divide by zero)

• Many others, please refer to the Intel Manual
  • Chapter 6 of volume 3A
Trap Summary

TRAP

Hardware Interrupt (Asynchronous)

Software Interrupt (Synchronous)

Exceptions (synchronous)

Faults (synchronous, Recoverable)
Hardware Interrupt

• A way of hardware interacting with CPU

• Example: a network device
  • NIC: “Hey, CPU, I have received a packet for the OS, so please wake up the OS to handle the data”
  • CPU: call the interrupt handler for network device in ring 0 (set by the OS)

• Asynchronous (can happen at any time of execution)
  • It’s a request from a hardware, so it comes at any time of CPU’s execution

• Read
Software Interrupt

- A software method to run code in ring 0 (e.g., \texttt{int $0x30} )
  - Telling CPU that "Please run the interrupt handler at 0x30"

- Synchronous (caused by running an instruction, e.g., \texttt{int $0x30})

- System call
  - \texttt{int $0x30} \leftarrow \text{system call in JOS}
Exceptions/Faults

• Exceptions
  • An error caused by the current execution (may or may not be recovered)
  • Examples of non-recoverable exception (cannot continue the execution)
    • Triple fault
    • Divide by zero
    • Breakpoint

• Fault
  • An error caused by the current execution that may be recovered and continue the execution
  • Examples
    • Page fault
    • Double fault

• Synchronous (an execution of an instruction can generate this)
  • E.g., divide by 0
## Handling Interrupt/Exceptions

- Set an Interrupt Descriptor Table (IDT)

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>0xf0130304</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>0xf0153333</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>0xf0183273</td>
</tr>
<tr>
<td>3 (Breakpoint)</td>
<td>0xf0223933</td>
</tr>
<tr>
<td>4 (Overflow)</td>
<td>0xf0333333</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 (Double Fault)</td>
<td>0xf0222293</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>14 (Page Fault)</td>
<td>0xf0133390</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x30 (syscall in JOS)</td>
<td>0xf0222222</td>
</tr>
</tbody>
</table>

![Figure 6-1. Relationship of the IDTR and IDT](image)
Opening a file

App calls open()

Set arguments (fn, flag) int $0x30 (syscall in JOS)

Interrupt!

Consult IDT

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>0xf0130304</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>0xf0153333</td>
</tr>
<tr>
<td>2 (NMI)</td>
<td>0xf0183273</td>
</tr>
<tr>
<td>3 (Breakpoint)</td>
<td>0xf0223933</td>
</tr>
<tr>
<td>4 (Overflow)</td>
<td>0xf0333333</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8 (Double Fault)</td>
<td>0xf0222293</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>14 (Page Fault)</td>
<td>0xf0133390</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x30 (syscall in JOS)</td>
<td>0xf0222222</td>
</tr>
</tbody>
</table>
At the kernel (in running `open()`):

- Access arguments from Ring 3
  - Need to check its security...

- Access disk to open a file
  - Need to check permissions...

- Return a file descriptor
  - `iret`
Summary

• A user program can invoke a system call to ‘request’ the OS to run code in a higher privileged level, ring 0
  • System call, and it is a synchronous interrupt

• A hardware would like to talk to the CPU to tell that blocks of data is ready for the OS
  • Hardware interrupt, an asynchronous interrupt

• A program generated an error that is not recoverable, a triple fault
  • A non-recoverable exception, synchronous

• A program generated a page fault
  • Fault, because OS regards page fault as recoverable error, synchronous
  • (we will learn more about this in coming lectures)