Systems Design and Implementation

1.8 - Device Drivers

System Architecture Group, SS 2009
University of Karlsruhe
June 16, 2008

Jan Stoess
University of Karlsruhe
Overview

- Device I/O related concepts
- L4 support for interfacing to hardware
- Software issues and structure
- PC Screen and Keyboard
Simplified Hardware Model

CPU

Bus

Device Controller

Memory

Interface

Device

© 2009 University of Karlsruhe, System Architecture Group
Example Hardware
Example Hardware

CPU → Bus → IDE Controller → IDE → Disk

Memory
Example Hardware
Example Hardware

CPU

Bus

Memory

UART Controller

RS-232

Terminal
Example Hardware

- CPU
- Bus
- USB Controller
- USB
- Memory
- Disk
Device Controller Registers

- Writing values
  - Configures the controller
  - Commands the controller to perform I/O actions
  - Provides data needed by the controller
    - E.g. a network packet

- Reading values
  - Identifies controller configuration
  - Queries status information
  - Receives data provided by controller
    - E.g. a network packet
Access to Device Registers

- Memory mapped I/O
  - Example: video ram
  - Device register access similar to physical memory
    - Normal load/store instructions used
      - Devices integrate cleanly with programming languages and normal protection mechanisms
    - Widely varied device speeds complicate the approach
      - Avoid slow devices on fast memory buses
    - Caches also complicate the approach
      - Don’t cache device registers
Access to Device Registers

- I/O address space
  - Example: UART registers on standard PC
  - Device register access via special instructions
    - Awkward language integration
      - Must include special instructions in languages
    - Requires additional protection mechanisms
      - Kernel-protected I/O insns., I/O fpages
    - Hardware can adjust behavior for I/O operations
      - Avoid caches, dedicated I/O bus, special bus cycles

- IA-32
  - Combination of both methods
Memory-to-Device Transfers

- CPU reads from memory and writes to device controller
  - Two bus transactions per word transferred
  - CPU is busy
    - Typically fast CPU, slow device
Device-to-Memory Transfers

- CPU reads from device controller and writes to memory
  - Two bus transactions per word transferred
  - CPU is busy
    - Typically fast CPU, slow device
DMA – Direct Memory Access

- DMA engine reads/writes directly to memory
  - CPU specifies a DMA transfer (start address, size, etc.)
  - DMA engine transfers data independently of CPU
    - Cycle stealing, burst-mode
  - At most one bus transaction per word transferred
  - CPU free to process in parallel
    - Assuming no memory contention
Interrupts

- Used to signal CPU that a device-related event has happened
  - Example: network packet received
Simple Interrupt Model

- Normal computation is *interrupted*
- Control is transferred to the interrupt service routine
- The interrupt event is processed
- The interrupt is acknowledged
- Normal computation is resumed
A little closer to reality – PIC

- Programmable interrupt controller
  - Many interrupt sources and few CPU interrupt input lines
  - Prioritizes interrupts from many devices
PIC in more detail
Cascading Interrupt Controllers
More recent: APICs and IO-APICs

Table:
input PIN ⇒ CPU, int#

Pentium4+:
APIC messages use the memory bus.
Much faster!

IO-APIC (2)

Serial APIC bus
Message: <dst cpu, int type, int#>

CPU0
APIC (0)

CPU1
APIC (1)

IO-APIC (3)
More recent: APICs and IO-APICs
L4 Support for Device Drivers

- Memory mapped devices
  - Accessed using normal instructions
  - Control access via virtual memory management functionality

- IA-32 I/O space instructions
  - Available to user-level
  - Can be restricted – I/O flexpages
Abstraction: Interrupts as IPC

- In-kernel threads represent IRQ line

  \[
  \text{global interrupt ID} \begin{array}{|c|c|}
  \hline
  \text{intr no (18/32)} & \text{1 (14/32)} \\
  \hline
  \end{array}
  \]

- System interrupt ID
- Interrupt handlers threads can register for interrupts
  \[
  \text{AssociateInterrupt (ThreadId InterruptTID, HandlerTID)}
  \]
- Interrupts are transformed into IPC messages
  - In-kernel thread performs IPC call to handler thread

  \[
  \begin{array}{|c|c|c|c|c|}
  \hline
  -1_{(12/14)} & 0_{(4)} & 0_{(4)} & t = 0_{(6)} & u = 0_{(6)} \\
  \hline
  \end{array}
  \text{MR}_0
  \]
  - Kernel disables IRQ on PIC after IRQ fires
  - Interrupt handler replies to acknowledge interrupt

  \[
  \begin{array}{|c|c|c|c|c|}
  \hline
  0_{(16/48)} & 0_{(4)} & t = 0_{(6)} & u = 0_{(6)} \\
  \hline
  \end{array}
  \text{MR}_0
  \]
  - IRQ gets re-enabled on PIC
Software
I/O Methods

- Programmed I/O
- Interrupt-driven I/O
- DMA-based I/O
Programmed I/O

Output system call code

\[
\text{copy\_from\_user(buffer,p,count);} \\
\text{for (i = 0; i < count; i++) { } } \\
\quad \text{while (*UART\_status\_reg != ready);} \\
\quad \text{*UART\_data reg = p[i];}
\]

- Problem
  - If device is slower than CPU, CPU spends all its time *polling* or *busy waiting*
Interrupt-driven I/O

Output system call code

```c
copy_from_user(buffer, p, count);
enable_interrupts();
while (*UART_status_reg != ready);
*UART_data_reg = p[i];
scheduler();
```

- CPU can perform other work while “waiting”
- However, interrupts are not for free!!!
DMA-based I/O

Output system call code
   copy_from_user(buffer, p, count);
   setup_DMA_controller();
   trigger_device_operation();
   scheduler();

Interrupt service routine
   acknowledge_interrupt();
   unblock_user();
   return_from_interrupt();

- Avoids interrupt per transfer
- One interrupt per buffer
Device Independence

- Avoid writing different applications for each device
- Categorize device into similar classes or types
  - Block versus character devices (?)
  - Disk, CDROM, DVD, Tape
  - Network
- Device drivers implement a standard interface for their device class
  - Abstracts from the details of particular hardware
  - Hides implementation details of managing the device
Synchronous/Asynchronous Access

- Classical applications expect synchronous access to a device
  - Application waits for read() result
  - Easy error recovery
- Devices are usually of asynchronous nature
- Operating System must reconcile these differences
  - Application blocking
  - Asynchronous *signals*
  - WaitForObject, WaitForMultipleObjects
(Double) Buffering

- **Write**
  - Preserve data until I/O completes while app can continue
  - Batch enough data to supply device data rate

- **Read**
  - Store data until destination identified
    - E.g., network
  - Combine received data to a single *packet*.
    - E.g., avoid unblock per character
  - However, copying is inefficient
Shareable vs. Exclusive Access

- Some devices are obviously shareable between users
  - Disk, network, …
- Other device only make sense when used by one user at a time
  - Tape drive, CD-writer, …
- OS must manage access to devices
  - Avoid deadlocks etc., …
Typical Software Model – Layers

- **User-Level I/O**
  - Software
  - Read/write on device

- **Device-independent**
  - OS software
  - Block device management

- **Device Driver**
  - IDE driver

- **Interrupt Service Routine**
  - IDE Interrupt Service Routine

- **Hardware**
Problem

- The ISR unblocks the blocked thread
  - What happens if significant processing is required to unblock the thread?
  - We may miss the next interrupt!!!
- Particularly problematic when *burst-rate* can be much higher than processing rate (throughput).
Typical Software Model

- ISR is as short as possible
  - Longer processing deferred until after burst
  - Linux: bottom half or tasklets
  - Windows NT: deferred procedure calls

<table>
<thead>
<tr>
<th>User-Level I/O</th>
<th>Read/write on socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td></td>
</tr>
<tr>
<td>Device-independent</td>
<td></td>
</tr>
<tr>
<td>OS software</td>
<td></td>
</tr>
<tr>
<td>Device Driver</td>
<td></td>
</tr>
<tr>
<td>Deferred ISR</td>
<td></td>
</tr>
<tr>
<td>Interrupt Service Routine</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
</tr>
<tr>
<td>IP stack</td>
<td></td>
</tr>
<tr>
<td>Deferred packet processing</td>
<td></td>
</tr>
<tr>
<td>Net ISR</td>
<td></td>
</tr>
</tbody>
</table>
Single-Threaded Device Driver

- Accepts requests
- Manages device
- Handles IRQs

Client

L4

Hardware
Multi-Threaded Device Driver

Client

Device Driver

Server Thread

Deferred ISR Thread

Interrupt Service Thread

L4

Hardware

IPC/synchronization
User Interaction

Screen and Keyboard in a PC
How to display something?

- Display controller
  - Human Interface Device
    - Converts contents of “virtual screen” into electrical signals for monitor
    - Video memory – addressable by CPU
    - Monitor image – readable by human
  - Supports different modes – Text vs. GFX
    - Mode defines interpretation of video memory contents – characters vs. pixels
VGA Display

- VGA BIOS initializes text mode 80x25
- Video memory starts at physical address 0xb8000
- Memory format – 2 bytes per character
  - Address x+0: ASCII character code
  - Address x+1: Color code

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>blink</td>
<td>Background color</td>
<td>bright</td>
<td>Foreground color</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- 80 characters per line (160 bytes)
- Cursor control via controller registers
How to type something?

- Keyboard controller
  - HID – converts depress/release of labeled key to scan-code
- Keyboard is now a PS/2 bus device
  - Shares bus with PS/2 mouse
- Chipset simulates an i8042 controller
  - Status register: read IO port 0x64
  - Control register: write IO port 0x64
  - Data register: read/write IO port 0x60
i8042 Keyboard Controller

- Status register
  - Bit0: 1 = Output buffer full
  - Bit1: 1 = Input buffer full

- Data register
  - Bit0-6 = scan code
  - Bit7: 0 = key pressed
    1 = key released

- Need to translate scan code to ASCII
- Shift keys - maintain state
i8042 Keyboard Controller

- Simple read loop:

  kbdRead:
  WaitLoop:      in al, 64h ; Read Status byte
                 and al, 10b ; Test IBF flag (Status<1>)
                 jz WaitLoop ; Wait for IBF = 1
                 in al, 60h ; Read input buffer

- Simple write loop:

  kbdWrite:
  WaitLoop:      in al, 64h ; Read Status byte
                 and al, 01b ; Test OBF flag (Status<0>)
                 jnz WaitLoop ; Wait for OBF = 0
                 out 60h, cl ; Write to output buffer
Console Server

- Allow clients to
  - Read from keyboard
  - Write to screen
- Focus client
  - Which client has the keyboard?
  - How to switch clients?
What’s missing in SDIOS?

- Shell
  - Goal: Launch new programs from a minimal shell
    - Exercise file system, task server, pager, console
- Furthermore …
  - Disk driver
  - File systems
  - Network driver
  - USB driver
  - IEEE1394 driver
  - … Feel free to go wild!