I/O Management
Intro

Chapter 5 - 5.3
Learning Outcomes

• A high-level understanding of the properties of a variety of I/O devices.
• An understanding of methods of interacting with I/O devices.
I/O Devices

• There exists a large variety of I/O devices:
  – Many of them with different properties
  – They seem to require different interfaces to manipulate and manage them
    • We don’t want a new interface for every device
    • Diverse, but similar interfaces leads to code duplication

• Challenge:
  – Uniform and efficient approach to I/O
• Logical position of device drivers is shown here
• Drivers (originally) compiled into the kernel
  – Including OS/161
  – Device installers were technicians
  – Number and types of devices rarely changed
• Nowadays they are dynamically loaded when needed
  – Linux modules
  – Typical users (device installers) can’t build kernels
  – Number and types vary greatly
    • Even while OS is running (e.g. hot-plug USB devices)
Device Drivers

• Drivers classified into similar categories
  – Block devices and character (stream of data) device

• **OS defines a standard (internal) interface to the different classes of devices**
  – Example: USB *Human Input Device* (HID) class specifications
    • human input devices follow a set of rules making it easier to design a standard interface.
## USB Device Classes

<table>
<thead>
<tr>
<th>Base Class</th>
<th>Descriptor Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Device</td>
<td>Use class information in the Interface Descriptors</td>
</tr>
<tr>
<td>01h</td>
<td>Interface</td>
<td>Audio</td>
</tr>
<tr>
<td>02h</td>
<td>Both</td>
<td>Communications and CDC Control</td>
</tr>
<tr>
<td>03h</td>
<td>Interface</td>
<td>HID (Human Interface Device)</td>
</tr>
<tr>
<td>05h</td>
<td>Interface</td>
<td>Physical</td>
</tr>
<tr>
<td>06h</td>
<td>Interface</td>
<td>Image</td>
</tr>
<tr>
<td>07h</td>
<td>Interface</td>
<td>Printer</td>
</tr>
<tr>
<td>08h</td>
<td>Interface</td>
<td>Mass Storage</td>
</tr>
<tr>
<td>09h</td>
<td>Device</td>
<td>Hub</td>
</tr>
<tr>
<td>0Ah</td>
<td>Interface</td>
<td>CDC-Data</td>
</tr>
<tr>
<td>0Bh</td>
<td>Interface</td>
<td>Smart Card</td>
</tr>
<tr>
<td>0Dh</td>
<td>Interface</td>
<td>Content Security</td>
</tr>
<tr>
<td>0Eh</td>
<td>Interface</td>
<td>Video</td>
</tr>
<tr>
<td>0Fh</td>
<td>Interface</td>
<td>Personal Healthcare</td>
</tr>
<tr>
<td>10h</td>
<td>Interface</td>
<td>Audio/Video Devices</td>
</tr>
<tr>
<td>DCh</td>
<td>Both</td>
<td>Diagnostic Device</td>
</tr>
<tr>
<td>E0h</td>
<td>Interface</td>
<td>Wireless Controller</td>
</tr>
<tr>
<td>EFh</td>
<td>Both</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>FEh</td>
<td>Interface</td>
<td>Application Specific</td>
</tr>
<tr>
<td>FFh</td>
<td>Both</td>
<td>Vendor Specific</td>
</tr>
</tbody>
</table>
I/O Device Handling

• Data rate
  – May be differences of several orders of magnitude between the data transfer rates

  – Example: Assume 1000 cycles/byte I/O
    • Keyboard needs 10 KHz processor to keep up
    • Gigabit Ethernet needs 100 GHz processor…..
## Sample Data Rates

<table>
<thead>
<tr>
<th>Device</th>
<th>Data rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>10 bytes/sec</td>
</tr>
<tr>
<td>Mouse</td>
<td>100 bytes/sec</td>
</tr>
<tr>
<td>56K modem</td>
<td>7 KB/sec</td>
</tr>
<tr>
<td>Telephone channel</td>
<td>8 KB/sec</td>
</tr>
<tr>
<td>Dual ISDN lines</td>
<td>16 KB/sec</td>
</tr>
<tr>
<td>Laser printer</td>
<td>100 KB/sec</td>
</tr>
<tr>
<td>Scanner</td>
<td>400 KB/sec</td>
</tr>
<tr>
<td>Classic Ethernet</td>
<td>1.25 MB/sec</td>
</tr>
<tr>
<td>USB (Universal Serial Bus)</td>
<td>1.5 MB/sec</td>
</tr>
<tr>
<td>Digital camcorder</td>
<td>4 MB/sec</td>
</tr>
<tr>
<td>IDE disk</td>
<td>5 MB/sec</td>
</tr>
<tr>
<td>40x CD-ROM</td>
<td>6 MB/sec</td>
</tr>
<tr>
<td>Fast Ethernet</td>
<td>12.5 MB/sec</td>
</tr>
<tr>
<td>ISA bus</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>EIDE (ATA-2) disk</td>
<td>16.7 MB/sec</td>
</tr>
<tr>
<td>FireWire (IEEE 1394)</td>
<td>50 MB/sec</td>
</tr>
<tr>
<td>XGA Monitor</td>
<td>60 MB/sec</td>
</tr>
<tr>
<td>SONET OC-12 network</td>
<td>78 MB/sec</td>
</tr>
<tr>
<td>SCSI Ultra 2 disk</td>
<td>80 MB/sec</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>125 MB/sec</td>
</tr>
<tr>
<td>Ultrium tape</td>
<td>320 MB/sec</td>
</tr>
<tr>
<td>PCI bus</td>
<td>528 MB/sec</td>
</tr>
<tr>
<td>Sun Gigaplane XB backplane</td>
<td>20 GB/sec</td>
</tr>
</tbody>
</table>

USB 3.0 625 MB/s (5 Gb/s)
Thunderbolt 2.5GB/sec (20 Gb/s)
PCIe v3.0 x16 16GB/s
Device Drivers

• **Device drivers job**
  – translate request through the device-independent standard interface (open, close, read, write) into appropriate sequence of commands (register manipulations) for the particular hardware
  – Initialise the hardware at boot time, and shut it down cleanly at shutdown
Device Driver

• After issuing the command to the device, the device either
  – Completes immediately and the driver simply returns to the caller
  – Or, device must process the request and the driver usually blocks waiting for an I/O complete interrupt.

• Drivers are thread-safe as they can be called by another process while a process is already blocked in the driver.
  – Thread-safe: Synchronised...
Device-Independent I/O Software

- There is commonality between drivers of similar classes
- Divide I/O software into device-dependent and device-independent I/O software
- Device independent software includes
  - Buffer or Buffer-cache management
  - TCP/IP stack
  - Managing access to dedicated devices
  - Error reporting
Driver ↔ Kernel Interface

• Major Issue is uniform interfaces to devices and kernel
  – Uniform device interface for kernel code
    • Allows different devices to be used the same way
      – No need to rewrite file-system to switch between SCSI, IDE or RAM disk
    • Allows internal changes to device driver with fear of breaking kernel code
  – Uniform kernel interface for device code
    • Drivers use a defined interface to kernel services (e.g. kmalloc, install IRQ handler, etc.)
    • Allows kernel to evolve without breaking existing drivers
  – Together both uniform interfaces avoid a lot of programming implementing new interfaces
    • Retains compatibility as drivers and kernels change over time.
Accessing I/O Controllers

a) Separate I/O and memory space
   - I/O controller registers appear as I/O ports
   - Accessed with special I/O instructions

b) Memory-mapped I/O
   - Controller registers appear as memory
   - Use normal load/store instructions to access

c) Hybrid
   - x86 has both ports and memory mapped I/O
Bus Architectures

(a) A single-bus architecture
(b) A dual-bus memory architecture
Intel IXP420

- Ethernet NPE A Ethernet MAC
- Ethernet NPE B Ethernet MAC
- North AHB Arbiter
- Queue Manager 8 KB SRAM
- SDRAM Controller 8 - 256 MB
- UART 921Kbaud
- Interrupt Controller
- Timers
- AHB/APB Bridge
- North/South AHB Bridge
- South AHB Arbiter
- Intel XScale® Core
  - 266/400/533 MHz
  - 32 KB Data Cache
  - 32 KB Instruction Cache
  - 2 KB Mini-Data Cache
- Test Logic Unit
- PMU (AHB)
- USB Device V1.1
- GPIO Controller
- UART 921Kbaud
- PCI Controller
- Expansion Bus Controller
- 16 GPIO
- 32 GPIO
- 32-BI

133.32 MHz x 32 bits North Advance High-Performance Bus
Queue Status Bus
133.32 MHz x 32 bits South Advance High-Performance Bus
66.66 MHz Advanced Peripheral Bus
Interrupts

- Devices connected to an *Interrupt Controller* via lines on an I/O bus (e.g. PCI)
- Interrupt Controller signals interrupt to CPU and is eventually acknowledged.
- Exact details are architecture specific.
I/O Interaction
Programmed I/O

- Also called *polling*, or *busy waiting*
- I/O module (controller) performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete
  - *Wastes CPU cycles*
Interrupt-Driven I/O

- Processor is interrupted when I/O module (controller) ready to exchange data
- Processor is free to do other work
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor
Direct Memory Access

- Transfers data directly between Memory and Device
- CPU not needed for copying

**Diagram**

- **DMA Controller in Device**
  - CPU
  - Memory
  - Device

- **Separate DMA Controller**
  - CPU
  - Memory
  - Device
  - DMA Controller
Direct Memory Access

• Transfers a block of data directly to or from memory
• An interrupt is sent when the task is complete
• The processor is only involved at the beginning and end of the transfer
DMA Considerations

- Reduces number of interrupts
  - Less (expensive) context switches or kernel entry-exits
- Requires contiguous regions (buffers)
  - Copying
  - Some hardware supports “Scatter-gather”
- Synchronous/Asynchronous
- Shared bus must be arbitrated (hardware)
  - CPU cache reduces (but not eliminates) CPU need for bus
The Process to Perform DMA Transfer

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

CPU

cache

DMA/bus/interrupt controller

CPU memory bus

memory

buffer

PCI bus

IDE disk controller

disk
disk
disk
disk
I/O Management Software

Chapter 5 – 5.3
Learning Outcomes

• An understanding of the structure of I/O related software, including interrupt handlers.
• An appreciation of the issues surrounding long running interrupt handlers, blocking, and deferred interrupt handling.
• An understanding of I/O buffering and buffering's relationship to a producer-consumer problem.
Operating System Design Issues

• **Efficiency**
  – Most I/O devices slow compared to main memory (and the CPU)
    • Use of multiprogramming allows for some processes to be waiting on I/O while another process executes
    • Often I/O still cannot keep up with processor speed
    • Swapping may used to bring in additional Ready processes
      – More I/O operations

• **Optimise I/O efficiency – especially Disk & Network I/O**
Operating System Design Issues

- **The quest for generality/uniformity:**
  - Ideally, handle all I/O devices in the same way
    - Both in the OS and in user applications
  - Problem:
    - Diversity of I/O devices
    - Especially, different access methods (random access versus stream based) as well as vastly different data rates.
    - Generality often compromises efficiency!
  - Hide most of the details of device I/O in lower-level routines so that processes and upper levels see devices in general terms such as read, write, open, close.
I/O Software Layers

Layers of the I/O Software System

1. Hardware
2. Interrupt handlers
3. Device drivers
4. Device-independent operating system software
5. User-level I/O software
Interrupt Handlers

- Interrupt handlers
  - Can execute at (almost) any time
    - Raise (complex) concurrency issues in the kernel
    - Can propagate to userspace (signals, upcalls), causing similar issues
    - Generally structured so I/O operations block until interrupts notify them of completion
      - kern/dev/lamebus/lhd.c
### Interrupt Handler Example

```
static int
lhd_io(struct device *d,
       struct uio *uio)
{
    ...
    /* Loop over all the sectors */
    /* we were asked to do. */
    for (i=0; i<len; i++) {
      /* Wait until nobody else * is using the device. */
      P(lh->lh_clear);
      ...
      /* Tell it what sector we want... */
      lhd_wreg(lh, LHD_REG_SECT, sector+i);
      /* and start the operation. */
      lhd_wreg(lh, LHD_REG_STAT, statval);
      /* Now wait until the interrupt */
      /* handler tells us we're done. */
      P(lh->lh_done);
      /* Get the result value */
      /* saved by the interrupt handler. */
      result = lh->lh_result;
    }
}
```

```
lhd_iqdone(struct lhd_softc *lh, int err)
{
    lh->lh_result = err;
    V(lh->lh_done);
}
```

```
void
lhd_irq(void *vlh)
{
    ...
    val = lhd_rdreg(lh, LHD_REG_STAT);
    switch (val & LHD_STATEMASK) {
      case LHD_IDLE:
      case LHD_WORKING:
        break;
      case LHD_OK:
      case LHD_INVSECT:
      case LHD_MEDIA:
        lhd_wreg(lh, LHD_REG_STAT, 0);
        lhd_iqdone(lh,
                   lhd_code_to_errno(lh, val));
        break;
    }
}
```
Interrupt Handler Steps

- **Save Registers** not already saved by hardware interrupt mechanism

- (Optionally) **set up context** for interrupt service procedure
  - Typically, handler runs in the context of the currently running process
    - No expensive context switch

- **Set up stack** for interrupt service procedure
  - Handler usually runs on the kernel stack of current process
  - Or “nests” if already in kernel mode running on kernel stack

- **Ack/Mask interrupt controller**, re-enable other interrupts
  - Implies potential for interrupt nesting.
Interrupt Handler Steps

- **Run interrupt service procedure**
  - Acknowledges interrupt at device level
  - Figures out what caused the interrupt
    - Received a network packet, disk read finished, UART transmit queue empty
  - If needed, it signals blocked device driver
- **In some cases, will have woken up a higher priority blocked thread**
  - Choose newly woken thread to schedule next.
  - Set up MMU context for process to run next
  - What if we are nested?
- **Load new/original process' registers**
- **Re-enable interrupt**: Start running the new process
Sleeping in Interrupts

• An interrupt generally has no context (runs on current kernel stack)
  – Unfair to sleep on interrupted process (deadlock possible)
  – Where to get context for long running operation?
  – What goes into the ready queue?

• What to do?
  – Top and Bottom Half
  – Linux implements with tasklets and workqueues
  – Generically, in-kernel thread(s) handle long running kernel operations.
Top/Half Bottom Half

- **Top Half**
  - Interrupt handler
  - remains short

- **Bottom half**
  - Is preemptable by top half (interrupts)
  - performs deferred work (e.g. IP stack processing)
  - Is checked prior to every kernel exit
  - signals blocked processes/threads to continue

- Enables low interrupt latency
- Bottom half can’t block
Stack Usage

1. Higher-level software
2. Interrupt processing (interrupts disabled)
3. Deferred processing (interrupt re-enabled)
4. Interrupt while in bottom half
Deferring Work on In-kernel Threads

- **Interrupt**
  - handler defers work onto in-kernel thread

- **In-kernel thread** handles deferred work (DW)
  - Scheduled normally
  - Can block

- **Both low interrupt latency and blocking operations**
Buffering
Device-Independent I/O Software

(a) Unbuffered input
(b) Buffering in user space
(c) *Single buffering* in the kernel followed by copying to user space
(d) Double buffering in the kernel
No Buffering

- Process must read/write a device a byte/word at a time
  - Each individual system call adds significant overhead
  - Process must wait until each I/O is complete
    - Blocking/interrupt/waking adds to overhead.
    - Many short runs of a process is inefficient (poor CPU cache temporal locality)
User-level Buffering

- Process specifies a memory buffer that incoming data is placed in until it fills
  - Filling can be done by interrupt service routine
  - Only a single system call, and block/wakeup per data buffer
    - Much more efficient
User-level Buffering

• Issues
  – What happens if buffer is paged out to disk
    • Could lose data while unavailable buffer is paged in
    • Could lock buffer in memory (needed for DMA), however many processes doing I/O reduce RAM available for paging. Can cause deadlock as RAM is limited resource
  – Consider write case
    • When is buffer available for re-use?
      – Either process must block until potential slow device drains buffer
      – or deal with asynchronous signals indicating buffer drained
Single Buffer

• Operating system assigns a buffer in kernel’s memory for an I/O request

• In a stream-oriented scenario
  – Used a line at time
  – User input from a terminal is one line at a time with carriage return signaling the end of the line
  – Output to the terminal is one line at a time
Single Buffer

- Block-oriented
  - Input transfers made to buffer
  - Block copied to user space when needed
  - Another block is written into the buffer
- Read ahead
Single Buffer

- User process can process one block of data while next block is read in
- Swapping can occur since input is taking place in system memory, not user memory
- Operating system keeps track of assignment of system buffers to user processes
Single Buffer Speed Up

• Assume
  – $T$ is transfer time for a block from device
  – $C$ is computation time to process incoming block
  – $M$ is time to copy kernel buffer to user buffer

• Computation and transfer can be done in parallel
• Speed up with buffering

\[
\frac{T + C}{\max(T, C) + M}
\]

No Buffering Cost

Single Buffering Cost
Single Buffer

• What happens if kernel buffer is full
  – the user buffer is swapped out, or
  – The application is slow to process previous buffer
    and more data is received???

=> We start to lose characters or drop network packets
Double Buffer

- Use two system buffers instead of one
- A process can transfer data to or from one buffer while the operating system empties or fills the other buffer

![Diagram of double buffering](c) Double buffering
Double Buffer Speed Up

• Computation and Memory copy can be done in parallel with transfer
• Speed up with double buffering

\[
\frac{T + C}{\max(T, C + M)}
\]

• Usually \( M \) is much less than \( T \) giving a favourable result
Double Buffer

- May be insufficient for really bursty traffic
  - Lots of application writes between long periods of computation
  - Long periods of application computation while receiving data
  - Might want to read-ahead more than a single block for disk
Circular Buffer

- More than two buffers are used
- Each individual buffer is one unit in a circular buffer
- Used when I/O operation must keep up with process
Important Note

• Notice that buffering, double buffering, and circular buffering are all

Bounded-Buffer Problems
Is Buffering Always Good?

\[
\begin{align*}
\text{Single} & : \quad \frac{T + C}{\max(T, C) + M} \\
\text{Double} & : \quad \frac{T + C}{\max(T, C + M)}
\end{align*}
\]

- Can \( M \) be similar or greater than \( C \) or \( T \)?
Buffering in Fast Networks

- Networking may involve many copies
- Copying reduces performance
  - Especially if copy costs are similar to or greater than computation or transfer costs
- Super-fast networks put significant effort into achieving zero-copy
- Buffering also increases latency
I/O Software Summary

Layers of the I/O system and the main functions of each layer

- **User processes**: Make I/O call; format I/O; spooling
- **Device-independent software**: Naming, protection, blocking, buffering, allocation
- **Device drivers**: Set up device registers; check status
- **Interrupt handlers**: Wake up driver when I/O completed
- **Hardware**: Perform I/O operation

I/O request → User processes → Device-independent software → Device drivers → Interrupt handlers → Hardware → Hardware