I/O Management Software
Chapter 5

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 be 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

<table>
<thead>
<tr>
<th>User-level I/O software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device-independent operating system software</td>
</tr>
<tr>
<td>Device drivers</td>
</tr>
<tr>
<td>Interrupt handlers</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>

Layers of the I/O Software System

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. */
        lh->lh_clear=0;
        P(lh->lh_done);
        /* Get the result value saved by the interrupt handler. */
        result = lh->lh_result;
    }
}
```

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

```
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_iodone(lh,
                       lhd_code_to_errno(lh, val));
            break;
    }
}
```

Interrupt Handler Steps

- **Run interrupt service procedure**
  - Acknowledges interrupt at device level
  - Figures out what caused the interrupt
  - Reenabled interrupt; re-enable other interrupts

- **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

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.

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
    - Genricly, 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. Upper software
2. Interrupt processing (interrupts disabled)
3. Deferred processing (interrupt re-enabled)
4. Interrupt while in bottom half
```

```
Kernel Stack
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>B</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>
```
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

---

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
  - Device specs often help, e.g. USB

---

USB Device Classes

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

---

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 re-entrant (or thread-safe) as they can be called by another process while a process is already blocked in the driver.
  - Re-entrant: Mainly no static (global) non-constant data.
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
  - 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

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

Issue

- 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 a 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 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

No Buffering Cost

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

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 Producer-Consumer Problems
Is Buffering Always Good?

\[
\frac{T + C}{\max(T, C) + M} \quad \text{Single} \quad \quad \quad \quad \quad \quad \frac{T + C}{\max(T, C + M)} \quad \text{Double}
\]

• 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