I/O Management

Chapter 5

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

- **User-level I/O software**
- **Device-independent operating system software**
- **Device drivers**
- **Interrupt handlers**

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

Interrupt Handler Example

```c
int lh_iocb(struct lh_softc *lh, iostruct *iop) {
    ...
    /* 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_busy);
        ...
        /* Tell it what sector we want... */
        lh_wreg(lh, LH_SECT, sector+i);
        /* and start the operation. */
        lh_wreg(lh, LH_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;
    }
    lh_iodone(lh, -errno);
}
```
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
- **Ack/Mask interrupt controller**, re-enable other interrupts
  - What does this imply?

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

- Interrupt generally has no **context** (runs on current stack)
  - Unfair to sleep 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.

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
- 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 as they can be called by another process while a process is already blocked in the driver.
  - Re-entrant: Code that can be executed by more than one thread (or CPU) at the same time
    - Manages concurrency using synch primitives
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

No Buffering

- Process must read/write a device a byte/word at a time
  - Each individual system call adds significant overhead
  - Process must what 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 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 main memory for an I/O request
- Stream-oriented
  - 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

- Block-oriented
  - Input transfers made to buffer
  - Block moved to user space when needed
  - Another block is moved into the buffer
  - Read ahead

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

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