Learning Outcomes

• An understanding of the structure of I/O related software, including interrupt handers.
• 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
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;
}

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

lhd_iodone(struct lhd_softc *lh, int err)
{
  lh->lh_result = err;
  V(lh->lh_done);
}
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. Upper software
2. Interrupt processing (interrupts disabled)
3. Deferred processing (interrupt re-enabled)
4. Interrupt while in bottom half

Kernel Stack

1. 
2. 
3. 
4.
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**

![Diagram of normal process/thread stack and in-kernel thread stack with DW and H]
Buffering
(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 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 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

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

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

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

- 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

- 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