Input/Output

Today
- Principles of I/O hardware & software
- I/O software layers
- Secondary storage

Next
- File systems
Operating systems and I/O

- Two key operating system goals
  - Control I/O devices
  - Provide a simple, easy-to-use, interface to devices

- Problem – large variety
  - Data rates – from 10B/sec (keyboard) 12,800MB/sec (100 Gigabit Ethernet)
  - Applications – what the device is used for
  - Complexity of control – a printer (simple) or a disk
  - Units of transfer – streams of bytes or larger blocks
  - Data representation – character codes, parity
  - Error condition – nature of errors, how they are reported, their consequences, …

- Makes a uniform & consistent approach difficult to get
I/O hardware - I/O devices

- I/O devices components
  - Device itself – mechanical component
  - Device controller or adapter – electronic component

- Device controller
  - Maybe more than one device per controller
  - Some standard interface between controller and devices: IDE, ATA, SATA, SCSI, FireWire, …
  - Converts serial bit stream to block of bytes (think of a disk)
  - Performs error correction as necessary
  - Makes data available in main memory

<table>
<thead>
<tr>
<th>Registers</th>
<th>Status</th>
<th>Command</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-controller (CPU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory (DRAM or SRAM or both)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other hardware specific chips</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interfaces</th>
</tr>
</thead>
</table>

| Internals |
Ways I/O can be done (OS take)

- **Programmed I/O**
  - Simplest – CPU does all the work
  - CPU basically polls the device
  - ... and it is tied up until I/O completes

```c
void print_page()
{
    copy_from_user((buffer_p, count);
    for (i = 0; i < count; i++) {
        while(*printer_status_reg != READY);
        *printer_data_reg = p[i];
    }
    return_to_user();
}
```
Ways I/O can be done

- **Interrupt-driven I/O**
  - Instead of waiting for I/O, context switch to another process & use interrupts

```c
void interrupt_service_procedure()
{
    if(count == 0) {
        unblock_user();
    } else {
        *printer_data_register[i] =
        --count;
        ++i;
    }
    acknowledge_interrupt();
    return_from_interrupt();
}
```

Diagram:
- Issue read command to I/O module
- Read status of I/O module
  - Check status
    - Ready
      - Read word from I/O module
        - Write word into memory
          - CPU → Mem
    - Error
      - Do something else
        - Interrupt
          - CPU → I/O
          - I/O → CPU
          - Ready
          - Check status
```
Ways I/O can be done

- Direct Memory Access
  - Obvious disadvantage of interrupt-driven I/O?
    An interrupt for every character
  - Solution: DMA - Basically programmed I/O done by somebody else

```c
copy_from_user((buffer_p, count);
set_up_DMA_controller();
scheduler();
```

Interrupt service procedure
```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```
Interrupts revisited

- When I/O is done – interrupt by asserting a signal on a bus line
- Interrupt controller detects it and puts a # on address lines – index into interrupt vector
- Signal makes CPU stop what is doing and change PC to interrupt service procedure (found in vector)
- Interrupt service procedure ACK the controller
- Before serving interrupt, *save context* …
Interrupts revisited

- Not that simple …
  - Where do you save the state?
  - Internal registers? Hold your ACK (avoid overwriting internal regs.)
  - In stack? You can get a page fault … pinned page?
  - In kernel stack? Change to kernel mode $$$

- Besides: pipelining, superscalar architectures, …

Many instructions in various stages of exec

Upon interrupt the PC may not reflect the correct boundary between executed/non-executed instructions
Interrupts revisited

- Ideally - a precise interrupt, leaves machine in a well-defined state
  1. PC is saved in a known place
  2. All previous instructions have been fully executed
  3. All following ones have not
  4. The exec state of the instruction pointed by PC is known

- No prohibition on what instructions to start, but all changes to register or mem must be undone before interrupt happens

- Tradeoff – complex OS or really complex interrupt logic within the CPU (design complexity & chip area)
Direct Memory Access

- Clearly OS can use it only if HW has DMA controller
  - Either on the devices (controller) or on the parentboard
- DMA has access to system bus, independent of CPU
- DMA operation

<table>
<thead>
<tr>
<th>Address</th>
<th>Count</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Main</td>
<td>memory</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td>controller</td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU issues command to disk – read, to internal buffer and check</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU program the DMA controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt when done</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMA requests xfer to mem.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data xfer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some details on DMA

- One or more transfers at a time
  - Need multiple set of registers for the multiple channels
  - DMA has to schedule itself over devices served

- Buses and DMA can operate on one of two modes
  - Cycle stealing – device controller occasionally steals the bus
  - Burst mode (block) – DMA tells the device to take the bus for a while

- Two approaches to data transfer
  - Fly-by mode – just discussed, direct transfer to memory
  - Two steps – transfer via DMA; it requires extra bus cycle, but now you can do device-to-device transfers

- Physical (common) or virtual address for DMA transfer

- Why you may not want a DMA?
  *If the CPU is fast and there’s not much else to do anyway*
I/O controller & CPU communication

- Device controllers have
  - A few registers for communication with CPU
    - Typically: data-in, data-out, status, control, command, …
  - A data buffer that OS can read/write (e.g. video RAM)

- How does the CPU use that?
  - Separate I/O and memory space, each control register assigned an I/O port (a) – IBM 360 (IN REG, PORT)
  - Memory-mapped I/O – first in PDP-11 (b)
  - Hybrid – Pentium (c) (graphic controller is a good example)
Memory-mapped I/O – pros and cons

✔ No special instructions or protection mechanism needed
  - Instruction that can reference memory can reference control registers

✔ Driver can be entirely written in C (how to do IN/OUT in C?)

✗ What to do with caching? Disable it on a per-page basis

✗ One AS, so all mem modules must check all references
  - Easy with single bus, but harder with dual-bus arch
  - Possible solutions
    • Send all references to memory first, if fails try bus
    • Snoop in the memory bus
    • Filter addresses in the PCI bridge (Pentium config.) (preloaded with range registers at boot time)
I/O software – goals & issues

- **Device independence**
  - Programs can access any I/O dev w/o specifying it in advance

- **Uniform naming, closely related**
  - Name independent of device

- **Error handling**
  - As close to HW as possible (1st controller should try, then device driver, …)
    - Many errors are transient or can be dealt with, transparently at a low lever

- **Buffering for better performance**
  - Check what to do with packets, for example
  - Decouple production/consumption

- **Deal with dedicated (tape) & shared devices (disks)**
  Dedicated devices bring their own problems – deadlock?
I/O software layers

- I/O normally implemented in layers

  ![I/O Subsystem Diagram]

  - User-level I/O software
  - Device-independent OS software
  - Device driver
  - Interrupt handlers
  - Hardware

- Interrupt handlers
  - Interrupts – an unpleasant fact of life – hide them!
  - Best way
    - Driver blocks (semaphores?) until I/O completes
    - Upon an interrupt, interrupt procedure handles it before unblocking driver
Layers - Device drivers

- Different device controllers – different registers, commands, etc → each I/O device needs a device driver

- Device driver – device specific code
  - Written by device manufacturer
  - Better if we have specs
  - Clearly, it needs to be reentrant (I/O device may complete while the driver is running, interrupting the driver and maybe making it run …)
  - Must be included in the kernel (as it needs to access the device’s hardware) - How do you include it?
    - Is there another option?
  - Problem with plug & play
Layers - Device-independent SW

Some part of the I/O SW can be device independent

- Uniform interfacing with drivers
  - Fewer modifications to the OS with each new device
  - Easier naming (/dev/disk0) – major & minor device #s in UNIX, driver + unit, (kept by the i-node of the device’s file)
  - Device driver writers know what’s expected of them

- Buffering
  - Unbuffered, user space, kernel, …

- Error reporting
  - Some errors are transient – keep them low
  - Actual I/O errors – reporting up when in doubt

- Allocating & releasing dedicated devices

- Providing a device-independent block size
User-space I/O software

- Small portion of I/O software runs in user-space
- Libraries that linked together with user programs
  - E.g., stdio in C
  - Mostly parameter checking and some formatting (printf)
- Beyond libraries, e.g. spooling
  - Handling dedicated devices (printers) in a multiprogramming system
  - Daemon plus spooling directory
Disk – a concrete I/O device

- **Interface**
  - A large number of sectors that can be read/written; each numbered from 0 to $n - 1$
    - Multi-sector operations are possible but only a single block write is atomic (so a portion of a larger write can fail)

- **Magnetic disk hardware - organization**
  - Platter with two surfaces
  - Tracks – divided into sectors
  - Cylinders – made of vertical tracks
  - Sectors – minimum transfer unit

- **Simplified model - careful with specs**
  - Sectors per track are not always the same
  - Zoning – zone, a set of tracks with equal sec/track
  - Hide this with a logical disk w/ constant sec/track
Disk operation (Single-platter view)

The disk surface spins at a fixed rotational rate (5400-15000 RPM)

The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

read/write heads move in unison from cylinder to cylinder
Disk access time

- Avg time to access target sector approximated by
  \[ T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} \]

- Seek time \((T_{\text{avg seek}})\)
  - Time to position heads over cylinder containing target sector.
  - Typical \(T_{\text{avg seek}} = 9\) ms

- Rotational latency \((T_{\text{avg rotation}})\)
  - Time waiting for first bit of sector to pass under r/w head.
  - \(T_{\text{avg rotation}} = T_{\text{max rotation}}/2 = 1/\text{RPMs} \times 60\ \text{sec}/1\ \text{min} \times 1/2\)

- Transfer time \((T_{\text{avg transfer}})\)
  - Time to read the bits in the target sector.
  - \(T_{\text{avg transfer}} = 1/\text{RPM} \times 1/(\text{avg # sectors/track}) \times 60\ \text{secs}/1\ \text{min}\).
Disk access time example

- **Given (~Seagate Barracuda)**
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.

- **Derived:**
  - \( T_{\text{avg rotation}} = \frac{1}{2} \times \left( \frac{60 \text{ secs}}{7200 \text{ RPM}} \right) \times 1000 \text{ ms/sec} = 4 \text{ ms.} \)
  - \( T_{\text{avg transfer}} = \frac{60}{7200} \text{ RPM} \times \frac{1}{400} \text{ sec/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms} \)
  - \( T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms} \)

- **Important points:**
  - *Access time dominated by seek time and rotational latency*
  - First bit in a sector is the most expensive, the rest are free
  - SRAM access time ~4 ns/doubleword, DRAM ~60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower than DRAM
Disks over time

- 20 years trends

<table>
<thead>
<tr>
<th>Parameter</th>
<th>IBM 360KB floppy</th>
<th>WD 18300 HD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>360KB</td>
<td>18.3GB</td>
</tr>
<tr>
<td>Seek time (avg)</td>
<td>77msec</td>
<td>6.9msec</td>
</tr>
<tr>
<td>Rotation time</td>
<td>200msec</td>
<td>8.33msec</td>
</tr>
<tr>
<td>Motor stop/start</td>
<td>250msec</td>
<td>20msec</td>
</tr>
<tr>
<td>Time to transfer 1 sector</td>
<td>22msec</td>
<td>17μsec</td>
</tr>
</tbody>
</table>

- Note different rates of improvements on seek time, transfer rate and capacity
Disk arm scheduling

- Time to read/write a disk block determined by
  - Seek time – dominates!
  - Rotational delay
  - Actual transfer time

- If request come one at a time, little you can do - FCFS

Starting at 53
Requests: 98,183,37,122, 14,124,65,67
SSTF

- Given a queue of request for blocks → scheduling to reduce head movement

Starting at 53
Requests: 98, 183, 37, 122, 14, 124, 65, 67

- As SJF, possible starvation
Assuming a uniform distribution of requests, where’s the highest density when head is on the left?
RAIDs

- Redundant Array of Inexpensive Disks
- Disk transfer rates are improving, but slower than CPU performance
- Use multiple disks to improve performance
  - Strip content across multiple disks
  - Use parallel I/O to improve performance
- To the file system, a RAID looks like any other disk – a linear array of blocks
  - Only bigger, potentially faster and potentially more reliable
- Evaluating RAID designs
  - Capacity – with \( N \) disks, what is the useful capacity available?
  - Reliability – how many disk faults can we tolerate?
  - Performance
RAIDs

- **RAID 0** – non-redundant disk array
  - Files are striped across disks, non redundant info
  - High read throughput
  - Best write throughput (nothing extra to write)
  - Worst reliability than with a single disk

- But striping reduces reliability (n*MTBF)
  - Add redundancy for reliability

- **RAID 1** – mirrored disk
  - Files are striped across half the disks
  - Data is written in two places
  - Read from either copy
  - On failure, just use the surviving one
  - Of course you need 2x space
RAIDs

• Another form of redundancy
  – Parity – add a bit to get even number of 1’s
  – Any single missing bit can be reconstructed
    • More complex schemes can detect/correct multiple bit errors

• RAID 2, 3 work on word (or byte) basis
  – Extra EEC bits added to parts of a word and distributed over all disks – RAID 2
  – A single parity bit computed per word and written to the parity disk – RAID 3
  – A read can access all data disks
  – A write updates 1+ data disks and parity disk
RAIDs

- **RAID 4 and 5**
  - Work with strips (not individual words as 2 and 3) and do not require synchronized drivers
  - RAID 4 is like RAID 0 with a strip-for-strip parity written onto an extra drive

- **RAID 5 – block interleaved distributed parity**
  - Distribute parity info over all disks
  - Much better performance (no hot spot)
RAIDs tradeoffs

- **Granularity**
  - Fine-grained – stripe each file over all disks
    - High throughput for the file
    - Limits transfer to one file at a time
  - Course-grained – stripe each file over only a few disks
    - Limit throughput for one file
    - Allows concurrent access to multiple files

- **Redundancy**
  - Uniformly distribute redundancy information on disks
    - Avoid load-balancing problems
  - Concentrate redundancy information on a small # of disks
    - Partition the disk into data disks and redundancy disks
    - Simpler
Next time

- File systems
  - Interface
  - Implementation
  - And a number of good examples