0.1 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**
- 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, lock, unlock

0.2 I/O Software Layers (see Fig. 1)

0.2.1 Interrupt Handlers

- Interrupt handlers are best hidden
  - Can execute at almost any time
  - Raise (complex) concurrency issues in the kernel
  - Have similar problems within applications if interrupts are propagated to user-level code (via signals, upcalls).
  - Generally, have driver starting an I/O operation block until interrupt notifies of completion; Example dev_read() waits on semaphore that the interrupt handler signals
- Interrupt procedure does its task then unblocks driver that started it

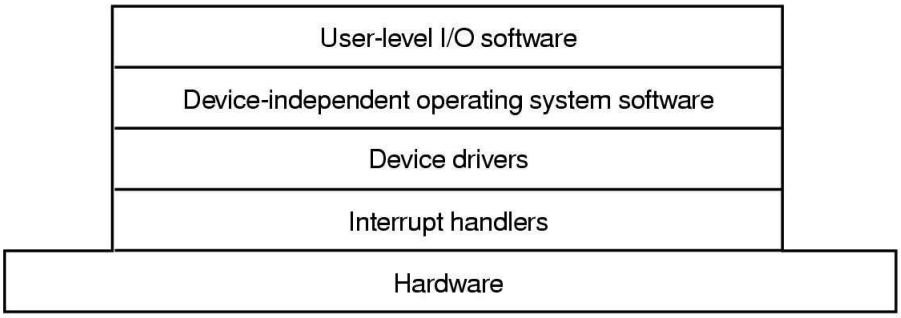


Figure 1: Layers of the I/O Software System.

- Steps must be performed in software upon occurance of an interrupt
  - Save registers not already saved by hardware interrupt mechanism
  - Set up context (address space) 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
    - Implies handler cannot block as the unlucky current process will also be blocked $\Rightarrow$ might cause deadlock
  - Ack/Mask interrupt controller, reenable other interrupts
  - 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
  - Load new/original process' registers
  - Start running the new process

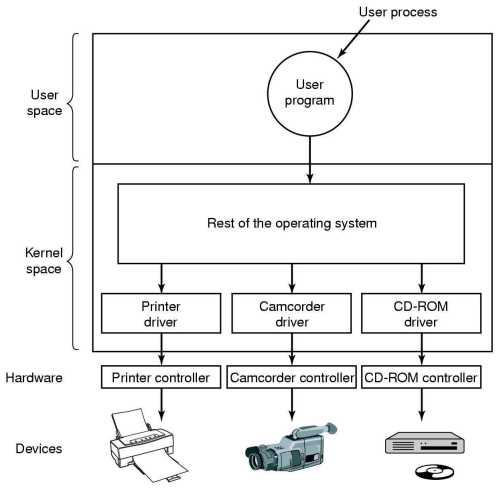


Figure 2: Logical positioning of device drivers. In reality all communications between drivers and device controllers goes over the bus.

0.2.2 Device Drivers (see Fig. 2)

- Drivers (originally) compiled into the kernel
  - 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)
- Drivers classified into similar catagories; Block devices and character (stream of data) device
- OS defines a standard (internal) interface to the different classes of devices
- 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

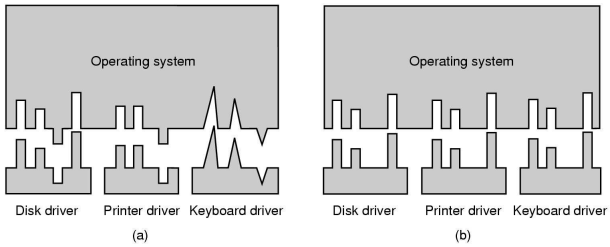


Figure 3: (a) Without a standard driver interface (b) With a standard driver interface.

  - Initialise the hardware at boot time, and shut it down cleanly at shutdown

- After issue the command to the device, the device either
  - Completes immediately and the driver simply return to the caller
  - Or, device must process the request and the driver usually blocks waiting for an I/O complete interrupt.
- Drivers are reentrant as they can be called by another process while a process is already blocked in the driver
  - Reentrant: Code that can be executed by more than one thread (or CPU) at the same time
  - Manages concurrency using synch primitives

0.2.3 Device Independent I/O Software(see Fig. 3)

- 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 $\Leftrightarrow$ Kernel Interface; Major Issue is uniform interfaces to devices and kernel

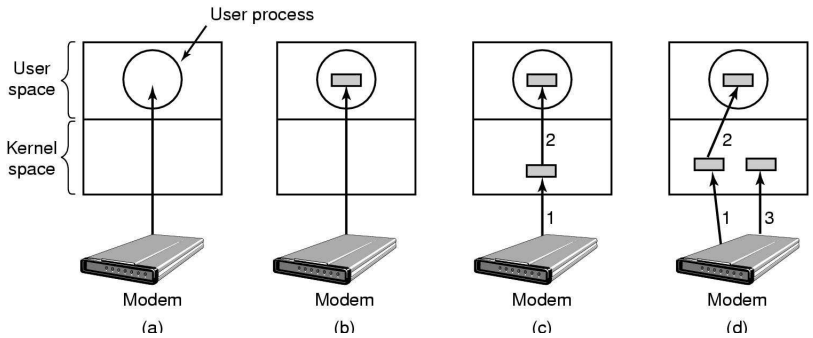


Figure 4: (a) Unbuffered input (b) Buffering in user space (c) *Single buffering* in the kernel followed by copying to user space (d) Double buffering in the kernel.

  - Uniform device interface for kernel code
    - Allows differents devices to be used the same way
      - No need to rewrite filesystem 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 (see Fig. 4)
  - 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 (see Fig. 4)

  - 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
  - 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 (see Fig. 4)
  - Operating system assigns a buffer in main memory for an I/O request
  - Stream-oriented
    * 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
  - Block-oriented
    * Input transfers made to buffer
    * Block moved to user space when needed
    * Another block is moved into the buffer; Read ahead
    * 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
  - What happens if kernel buffer is full, the user buffer is swapped out, and more data is received??? We start to lose characters or drop network packets

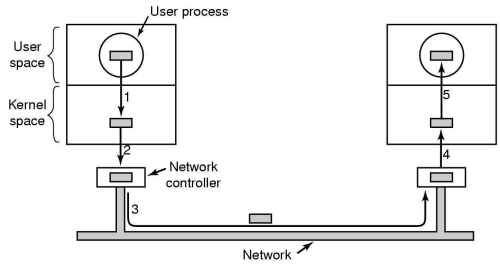


Figure 5: Networking may involve many copies.

- Double Buffer (see Fig. 4)
  - 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
  - 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
- Notice that buffering, double buffering are all Bounded-Buffer Producer-Consumer Problems
- Buffering in Fast Networks (see Fig. 5)
  - 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

0.2.4 User Level Software

- library calls
  - users generally make library cals that then make the system calls
  - example:
    - int count=**write**(fd,buffer,n);
    - **write** function is run at the user level

    - simply takes parameters and makes a system call
  - another example:
    - `printf("My age: %d \n",age);`
    - takes a string, reformats it, and then calls the write system call

- spooling
  - user program places data in a special directory
  - a *daemon* (background program) takes data from directoryy and outputs it to a device
    - the user doesn't have permission to directly access the device
    - daemon runs as a privileged user
  - prevents users from tying up resources for extented periods of time; printer example
  - OS never has to get involved in working with the I/O device

0.3 Disks (see Fig. 6)

- Management and ordering of disk access requests is important:
  - Huge speed gap between memory and disk
  - Disk throughput is extremely sensitive to
    - Request order $\Longrightarrow$ Disk Scheduling
    - Placement of data on the disk $\Longrightarrow$ file system design
  - Disk scheduler must be aware of *disk geometry*

- Disk management issues
  - Formatting
    - Physical: divide the blank slate into sectors identified by headers containing such information as sector number; sector interleaving
    - Logical: marking bad blocks; partitioning (optional) and writing a blank directory on disk; installing file allocation tables, and other relevant information (file system initialization)
  - Reliability
    - disk interleaving or striping

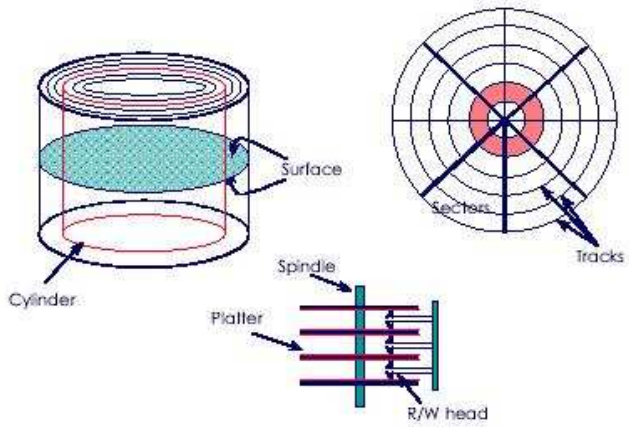
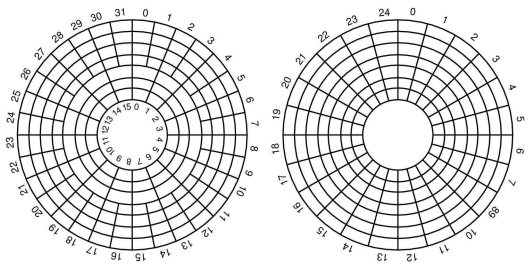


Figure 6: Disk Structure.

  - ∗ RAIDs (Redundant Array of Inexpensive Disks): various levels, e.g., level 0 is disk striping)
- – Controller caches newer disks have on-disk caches (128KB 512KB)

0.3.1 Disk Hardware

- Disk drives addressed as large 1-dimensional arrays of *logical blocks* (smallest transfer unit)
- 1-dimensional array of logical blocks mapped onto sectors of disk sequentially
  - sector 0: 1st sector of 1st track on outermost cylinder
  - mapping in order through that track, then rest of tracks in that cylinder, then through rest of cylinders from outermost to innermost
- Outer tracks can store more sectors than inner without exceed max information density (see Fig. 7 Left)
- Evolution of Disk Hardware (see Fig. 7 Right)
  - Average seek time is approx 12 times better
  - Rotation time is 24 times faster
  - Transfer time is 1300 times faster



Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 µsec

Figure 7: Left: (a) Physical geometry of a disk with two zones (b) A possible virtual geometry for this disk, Right: Disk parameters for the original IBM PC floppy disk and a Western Digital WD 18300 hard disk.

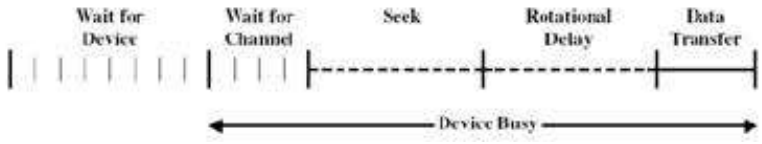


Figure 8: Disk Performance.

  - Most of this gain is due to increase in density
  - Represents a gradual engineering improvement

- Disk Performance (see Fig. 8)
  - Disk is a moving device; must be positioned correctly for I/O
  - Execution of a disk operation involves
    - Wait time: the process waits to be granted device access
      - Wait for device: time the request spend in wait queue
      - Wait for channel: time until a shared I/O channel is available
    - Access time: time hardware need to position the head
      - Seek time: position the head at the desire track
      - Rotational delay (latency): spin disk to the desired sector
    - Transfer time: sectors to be read/written rotate below head

- Estimating Access Time;
  - Seek Time T_s: Moving the head to the required tgrack not linear in the number of tracks to traverse: startup time, settling time. Typical avearge seek time: a few milliseconds

- Rotational delay: rotational speed, r, of 5000 to 10000 rpm. At 10000 rpm, one revolution per 6ms⇒ average delay 3ms
- Transfer time: to transfer b bytes, with N bytes per track;

$$T = \frac{b}{rN}$$

Total average access time:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

- A Timing Comparison
  - $T_s = 2$ ms, $r = 10000$rpm, 512B sect, 320 sect/track
  - read a file with 2560 sectors (=1.3MB)
  - file stored compactly (8 adjacent tracks): Read first track

Average seek	2ms
Rot. Delay	3ms
Read 320 sectors	6ms
Total	11ms
All sectors	11+7*9=74ms

  - Sectors distributed randomly over the disk: Read any sector

Average seek	2ms
Rot. Delay	3ms
Read 1 sectors	0.01875ms
Total	5.01875ms
All	2560*5.01875=20,328ms

- Disk Performance is Entirely Dominated by Seek and Rotational Delays
  - Will only get worse as capacity increases much faster than increase in seek time and rotation speed (it has been easier to spin the disk faster than improve seek time)
  - Operating System should minimise mechanical delays as much as possible

0.3.2 Disk Formatting

- A hard disk consist of a stack of aliminum, aaloy, or glass platters 5.25 inch or 3.5 inch in diameter. On each platter is deposited a thin magnetizable metal oxide

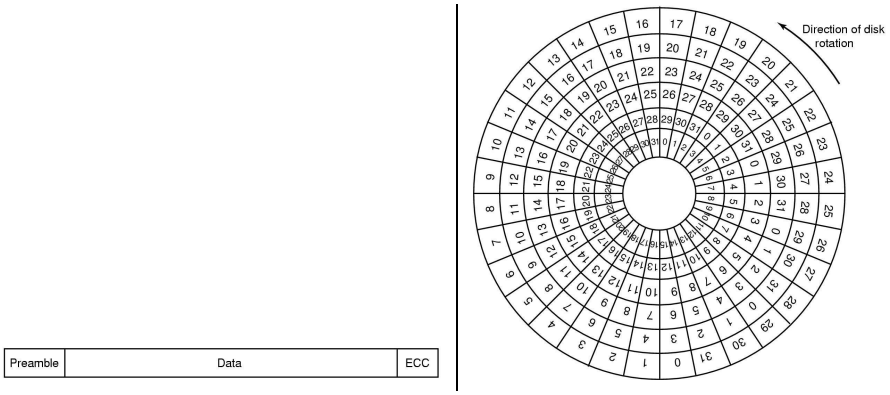


Figure 9: Left: Low-level Disk Formatting; A disk sector, Right: An illustration of cylinder skew.

- Before the disk can be used, each platter must recieve a **low-level format , or physical formatting** ; divide disk into sectors that disk controller can read and write (see Fig. 9 left)
- To use disk to hold files, OS needs to record own data structures on disk
  - *partition* disk into ≥ 1 groups of cylinders *logical formatting* or "making a file system"
- Boot block to start up system
  - bootstrap code in ROM
  - *bootstrap loader* program minimum in ROM
- When reading sequential blocks, the seek time can result in missing block 0 in the next track
- Disk can be formatted using a cylinder *skew* to avoid this (see Fig. 9 right)
- Issue: After reading one sector, the time it takes to transfer the data to the OS and receive the next request results in missing reading the next sector
- To overcome this, we can use interleaving (see Fig. 10)
- Modern drives overcome interleaving type issues by simply reading the entire track (or part thereof) into the on-disk controller and caching it.

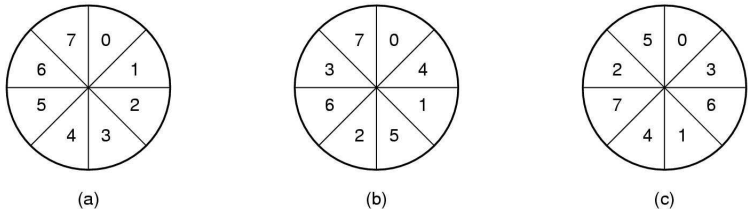


Figure 10: a) No interleaving b) Single interleaving c) Double interleaving.

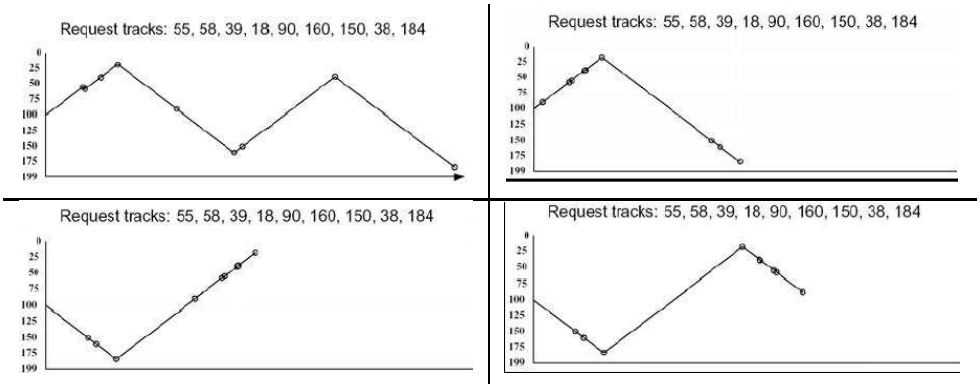


Figure 11: From left to right: First-in, First-out (FIFO); Shortest Seek Time First; Elevator Algorithm (SCAN); Modified Elevator (Circular SCAN, C-SCAN)

0.3.3 Disk Arm Scheduling Algorithms (see Fig. 11)

- Time required to read or write a disk block determined by 3 factors; Seek time, Rotational delay, Actual transfer time
- Seek time dominates
- For a single disk, there will be a number of I/O requests
- Processing them in random order leads to worst possible performance
- **First-in, First-out (FIFO)**
  - Process requests as they come
  - Fair (no starvation)
  - Good for a few processes with clustered requests
  - Deteriorates to random if there are many processes
- **Shortest Seek Time First**

  - Select request that minimises the seek time
  - Generally performs much better than FIFO
  - May lead to starvation

- **Elevator Algorithm (SCAN)**
  - Move head in one direction; Services requests in track order until it reaches the last track, then reverses direction
  - Better than FIFO, usually worse than SSTF
  - Avoids starvation
  - Makes poor use of sequential reads (on down-scan)

- **Modified Elevator (Circular SCAN, C-SCAN)**
  - Like elevator, but reads sectors in only one direction; When reaching last track, go back to first track non-stop
  - Better locality on sequential reads
  - Better use of read ahead cache on controller
  - Reduces max delay to read a particular sector

- Selecting a Disk-Scheduling Algorithm
  - SSTF common, natural appeal
  - SCAN and C-SCAN perform better if heavy load on disk
  - Performance depends on number and types of requests
  - Requests for disk service influenced by file-allocation method
  - Disk-scheduling should be separate module of OS, allowing replacement with different algorithm if necessary

0.3.4 Error–Handling

- Bad blocks are usually handled transparently by the on-disk controller (see Fig. 12)

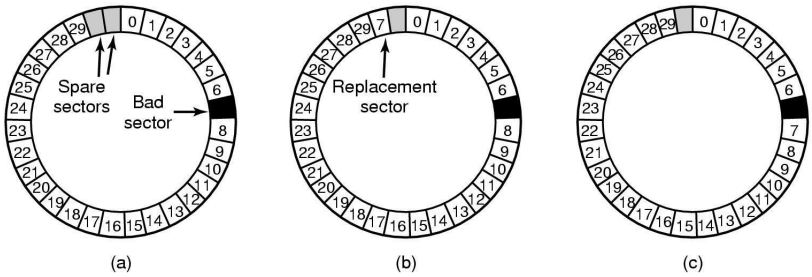


Figure 12: a) A disk track with a bad sector b) Substituting a spare for the bad sector c) Shifting all the sectors to bypass the bad one.