Chapter 5

Input/Output

The I/O Subsystem

- **The largest, most complex subsystem in OS**
- **Most lines of code**
- **Highest rate of code changes**
- **Where OS engineers most likely to work**
- **Difficult to test thoroughly**

- **Make-or-break issue for any system**
  - » Big impact on performance and perception
  - » Bigger impact on acceptability in market

I/O Devices

- **Types of I/O devices**
  - **Block devices**
    - » Stores information in fixed-size blocks, each one with its own address
    - » Read or write each block independently
    - » Common block size: 512 bytes to 32,768 bytes
    - » E.g. disks
  - **Character devices**
    - » Delivers or accepts a stream of characters, without regard to any block structure
    - » Not addressable, no seek operation
    - » E.g. printers, network interfaces, mice
  - **Other devices**
    - » Clocks, memory-mapped screens

Principles of I/O Hardware

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Some typical device, network, and data base rates

Wide range of types and data rates

Device Controllers

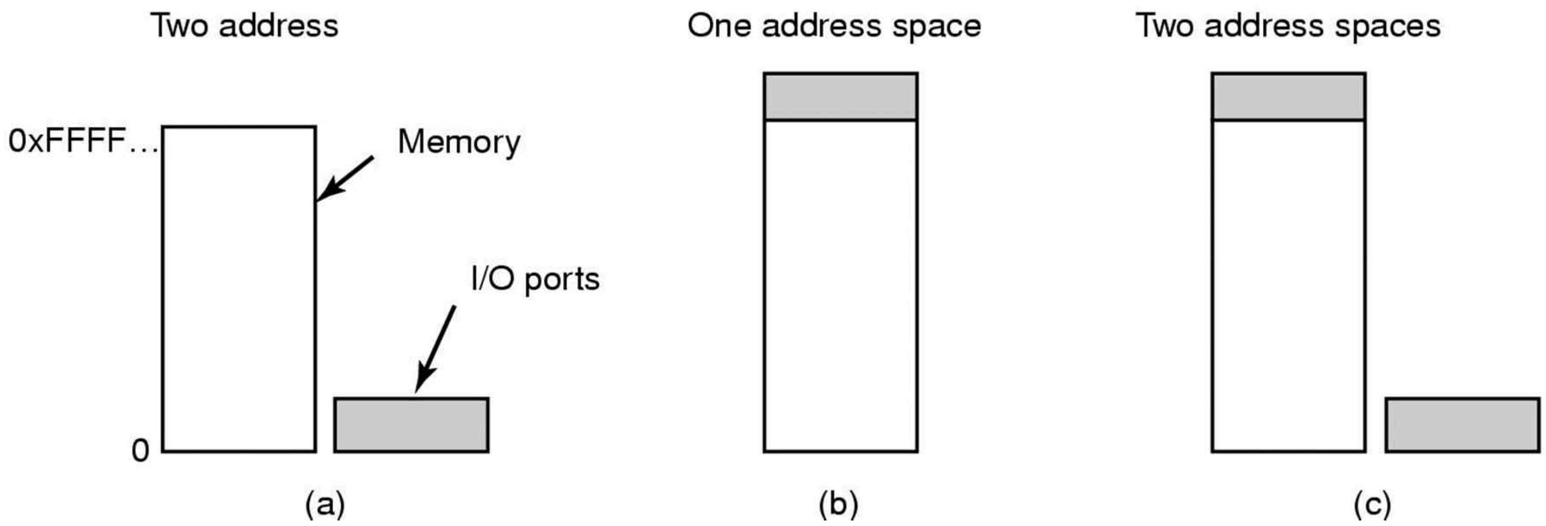
- I/O devices have components:
  - mechanical component
  - electronic component
- The electronic component is the device controller
  - may be able to handle multiple devices
- Controller's tasks
  - convert serial bit stream to block of bytes
  - perform error correction as necessary
  - Copy data to main memory if necessary

Memory-Mapped I/O (0)

- Separate I/O and Memory space
  - Address spaces for I/O and Memory are different
  - I/O port number : assigned to each control register
  - Use special I/O instructions
  - E.g. “IN REG, PORT” or “OUT PORT, REG”

- Memory-mapped I/O
  - Each control register is assigned a unique memory address
  - Use regular memory reference instructions to access I/O control registers
  - E.g. “store REG, Ether_Control_REG1”
- Hybrid
  - Both of the above schemes

Memory-Mapped I/O (1)



- Separate I/O and memory space
- Memory-mapped I/O
- Hybrid

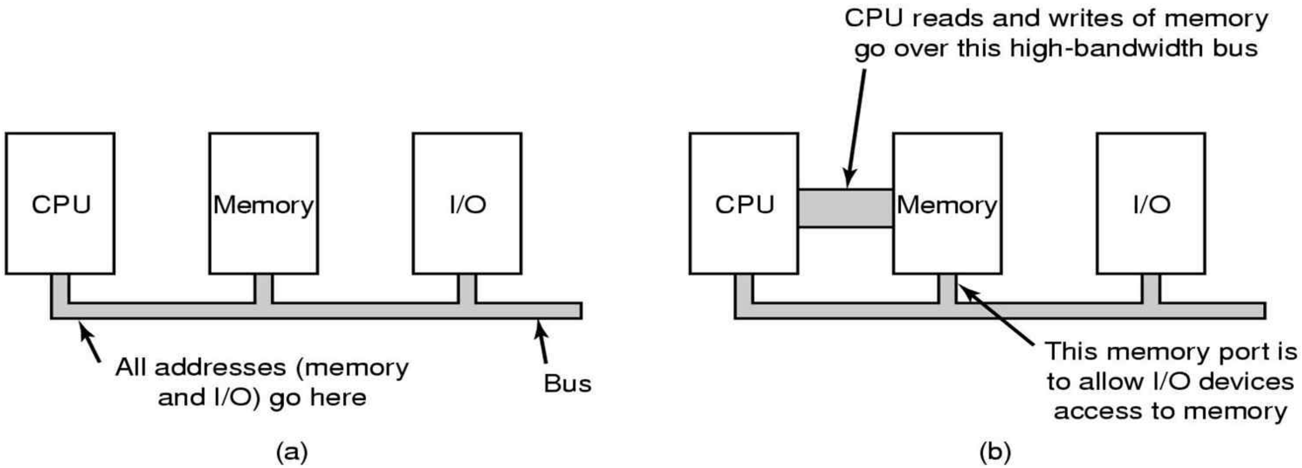
Memory-Mapped I/O (0)

- **Advantages of Memory-mapped I/O**
  - **No special instructions needed**
    - » Device registers can be addresses in C/C++ without using assembly code
  - **No special protection mechanism is needed to keep user processes from performing I/O**
    - » Assign address spaces for I/O registers to protected pages
  - **Every instruction that can reference memory can also reference control registers**
    - » E.g.

```
LOOP :  TEST PORT_4
        BEQ READY
        BRANCH LOOP
READY:
```

Memory-Mapped I/O (0)

- **Disadvantages of Memory-mapped I/O**
  - **Need hardware able to selectively disable caching**
    - » A device register should not be cached
  - **All memory modules and I/O devices must examine all memory references to see which ones to respond to**
    - » Memory bus and I/O bus are separated, we have three choices
      - CPU try memory first and if it fails, try other I/O buses
      - A snooping device on the memory bus
      - Filter addresses in the PCI bridge chip (Pentium Configuration) Fig 1-11

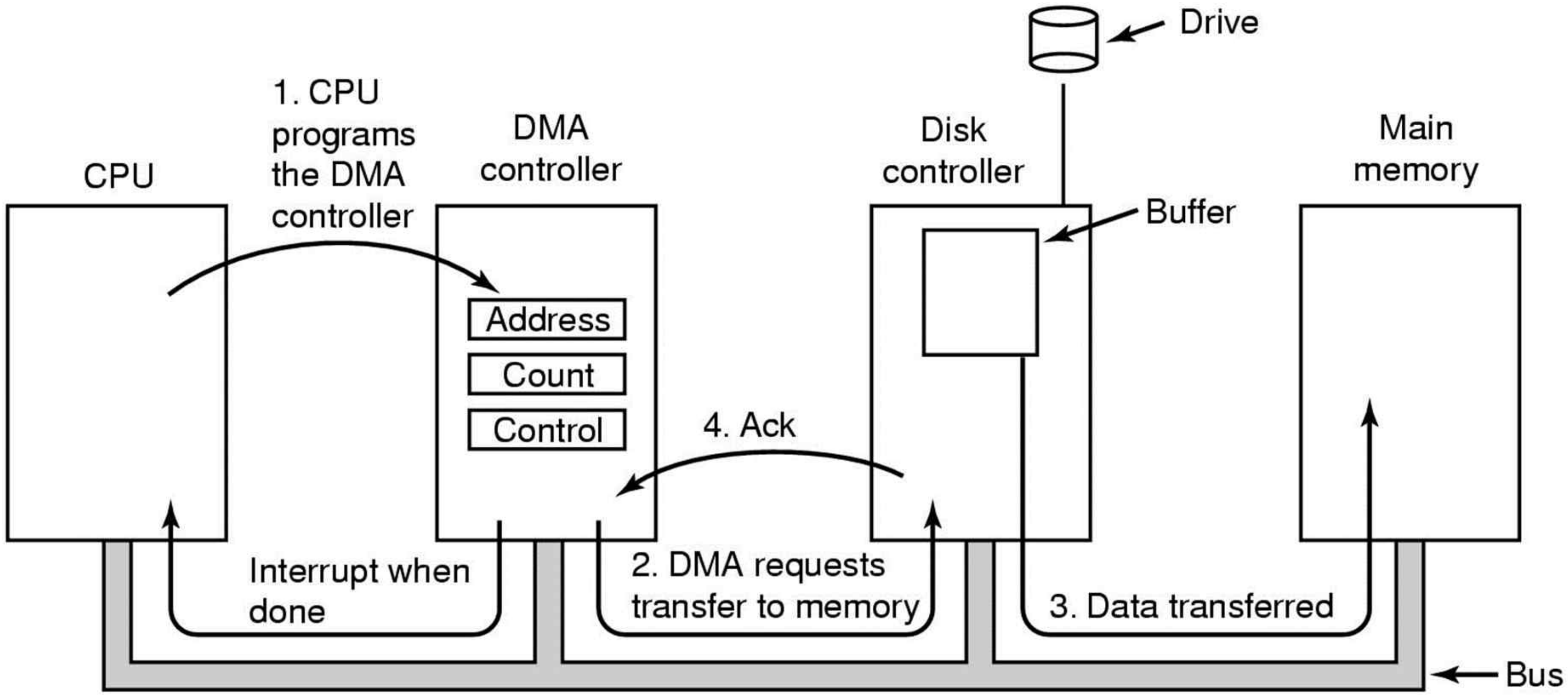


(a) A single-bus architecture
(b) A dual-bus memory architecture

Direct Memory Access (DMA)

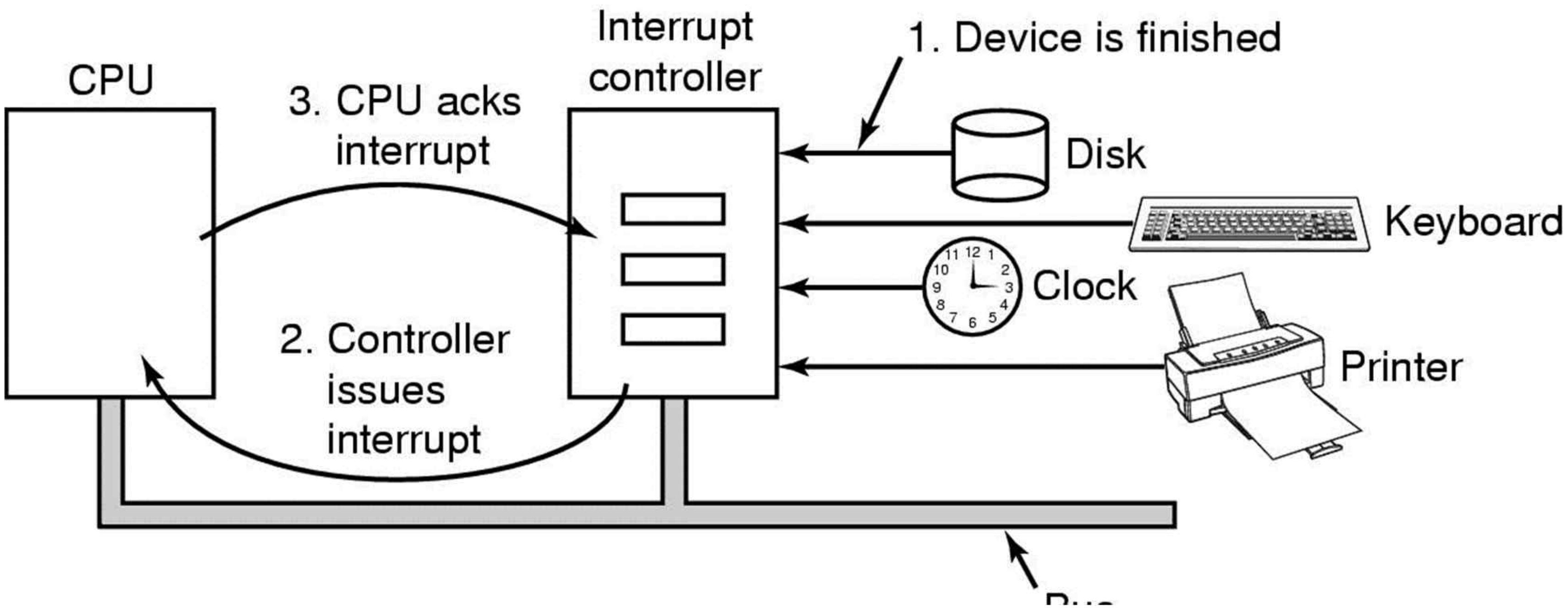
- **DMA (Direct Memory Access)**
  - **allows certain hardware devices to access memory for reading and writing independently of CPU**
    - » Without DMA, CPU typically has to be occupied for the entire time it's performing a data transfer
    - » With DMA, CPU would initiate the transfer, do other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done
- **DMA transfer mode**
  - **Cycle stealing**
    - » The controller sneaks in and steals an occasional bus cycle from the CPU to transfer a word at a time
  - **Burst mode**
    - » The controller acquire the bus and transfer multiple words
    - » It blocks the CPU or other devices during the transfer
- **Most DMA controllers use physical addresses**

Direct Memory Access (DMA)



Operation of a DMA transfer

Interrupts Revisited

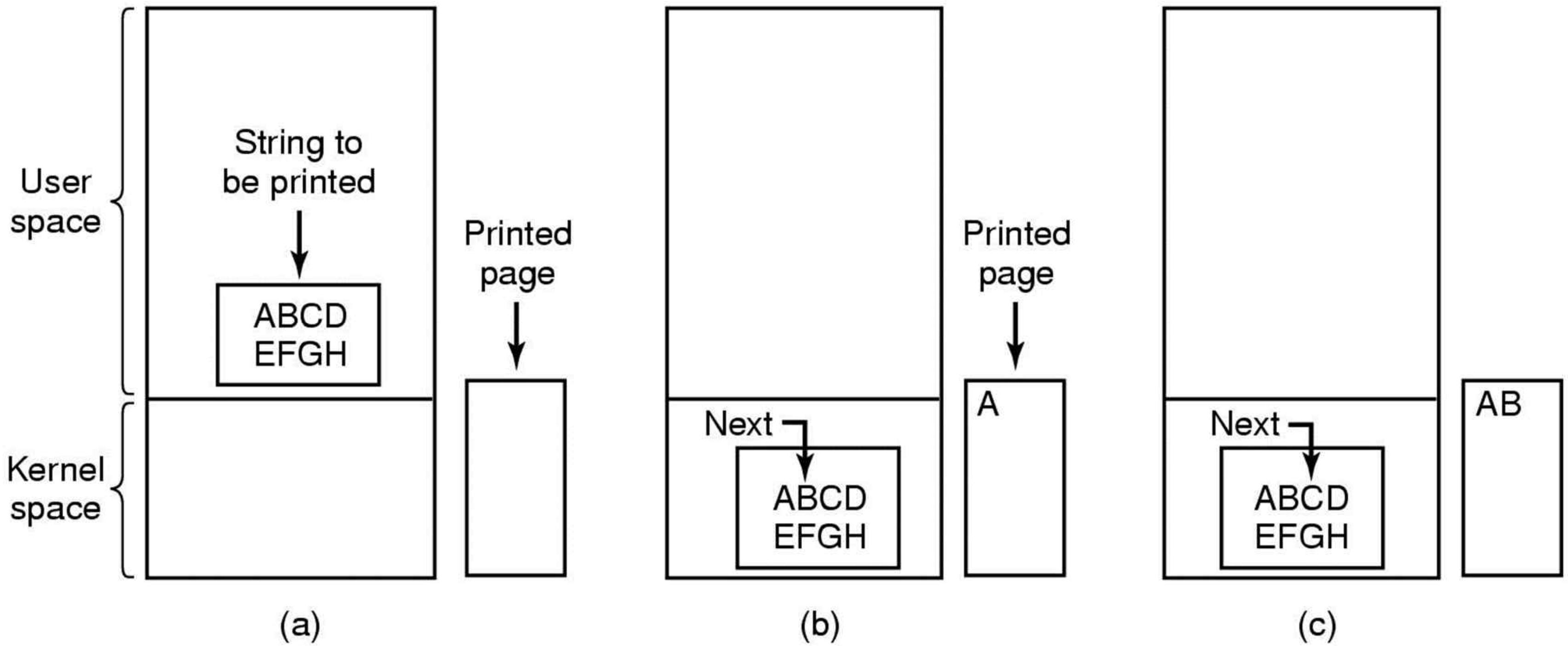


How interrupts happens. Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires

Interrupts Revisited

- **Interrupt vector**
  - **To fetch a new program counter for a corresponding interrupt service routine**
- **Where to save hardware state information for interrupt service returns**
  - **The current user stack**
    - Stack pointer may not even legal
    - May be the end of a page, causing a page fault
  - **Kernel stack**
    - Better chance of the stack pointer being legal and in a pinned page
    - Require change MMU contexts to switch to kernel
    - Probably invalidate most or all of the cache and TLB
- **Precise vs. imprecise interrupts**
  - **A precise interrupt that leaves the machine in a well-defined state and an imprecise interrupt does not**
  - **OS has to handle a large amount of internal state information generated by an imprecise interrupt**

Programmed I/O (1)



Steps in printing a string

Programmed I/O (2)

```
copy_from_user(buffer, p, count);             /* p is the kernel bufer */
for (i = 0; i < count; i++) {                 /* loop on every character */
    while (*printer_status_reg != READY) ;    /* loop until ready */
    *printer_data_register = p[i];            /* output one character */
}
return_to_user( );
```

Writing a string to the printer using programmed I/O

- **CPU do all the work - tying up the CPU full time until I/O is done**
- **Polling or busy waiting**

Interrupt-Driven I/O

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a) Code executed when print system call is made

```
if (count == 0) {
      unblock_user( );
} else {
      *printer_data_register = p[i];
      count = count – 1;
      i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b) Interrupt service procedure

- **Writing a string to the printer using interrupt-driven I/O**
  - **CPU does something else while waiting for I/O to become ready**
  - **I/O interrupts when it completes the operation and is ready**
  - **Interrupts take time, so frequent interrupts waste CPU time**

I/O Using DMA

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a) Code executed when print system call is made

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b) Interrupt service procedure

- **Printing a string using DMA**
  - **DMA controller feed the characters to the printer one at a time**
  - **Without the CPU being bothered**
  - **Reduces the number of interrupts from one per char to one per buffer printed**

Interrupt Handlers (1)

- **Interrupt handlers are best hidden**
  - **have driver starting an I/O operation block until interrupt notifies of completion**

- **Interrupt procedure does its task**
  - **then unblocks driver that started it**

- **Steps must be performed in software after interrupt completed**

1. **Save regs not already saved by interrupt hardware**
2. **Set up context for interrupt service procedure**
   - Set up TLB, MMU, and a page table

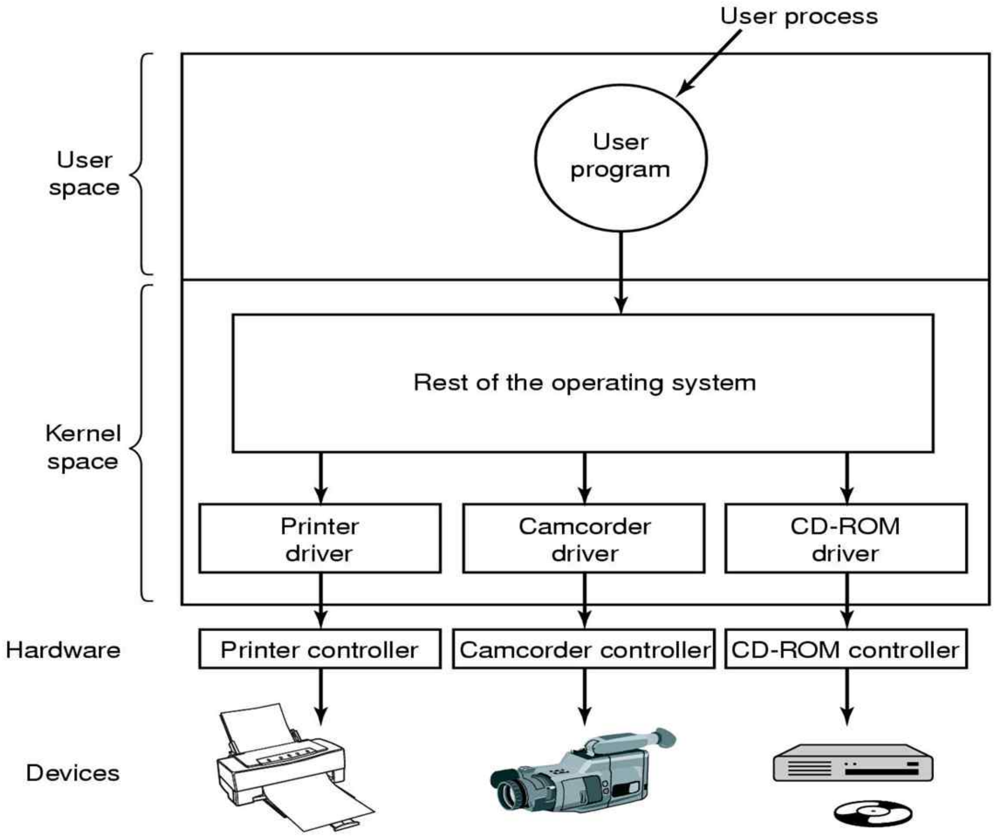
Interrupt Handlers (2)

3. Set up stack for interrupt service procedure
4. Ack interrupt controller, reenable interrupts
5. Copy registers from where saved
6. Run service procedure
7. Set up MMU context for process to run next
8. Load new process' registers
9. Start running the new process

Device Drivers

- Each I/O device attached to a computer needs some device-specific code for controlling the device
  - The code is called *device driver*
  - Device manufacture commonly supply drivers for OS
- Device driver normally has to be part of the OS
  - Need to access device's registers
- Device drivers are dynamically loaded to the system
  - Old approach : every driver need to be compiled into the OS
- Device drivers
  - Provide read/write funcs to upper level sw
  - Initialize the device and perform power management for the device etc.
- Device drivers have to be reentrant
  - Interrupt requiring the driver to run may happen during the driver execution

Device Drivers



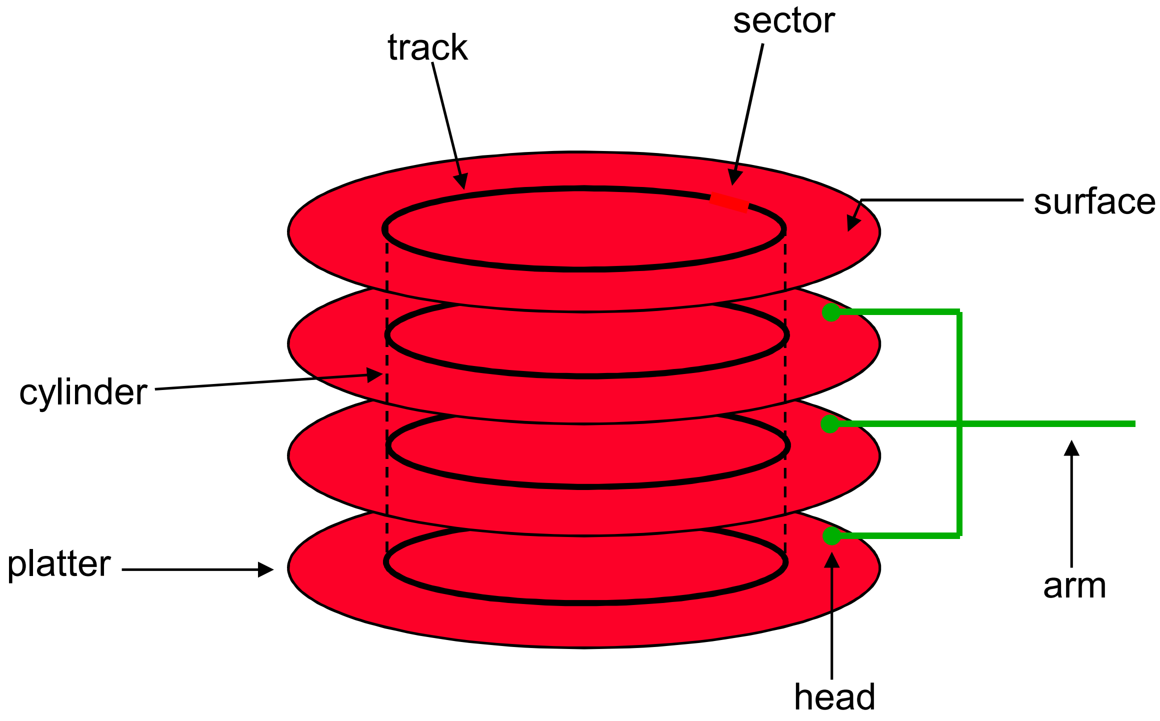
- Logical position of device drivers is shown here
- Communications between drivers and device controllers goes over the bus

Disks and the OS

- Disks are messy, messy devices
  - errors, bad blocks, missed seeks, etc.
- Job of OS is to hide this mess from higher-level software
  - low-level device drivers (initiate a disk read, etc.)
  - higher-level abstractions (files, databases, etc.)
- OS may provide different levels of disk access to different clients
  - physical disk block (surface, cylinder, sector)
  - disk logical block (disk block #)
  - file logical (filename, block or record or byte #)

Physical Disk Structure

- **Disk components**
  - platters
  - surfaces
  - tracks
  - sectors
  - cylinders
  - arm
  - heads



Interacting with Disks

- **In the old days…**
  - **OS would have to specify cylinder #, sector #, surface #, transfer size**
    - » I.e., OS needs to know all of the disk parameters
- **Modern disks are even more complicated**
  - **not all sectors are the same size, sectors are remapped, …**
  - **disk provides a higher-level interface, e.g. SCSI**
    - » exports data as a logical array of blocks [0 … N]
    - » maps logical blocks to cylinder/surface/sector
    - » OS only needs to name logical block #, disk maps this to cylinder/surface/sector
    - » as a result, physical parameters are hidden from OS
      - both good and bad

Example disk characteristics

- IBM Ultrastar 36XP drive
  - form factor: 3.5”
  - capacity: 36.4 GB
  - rotation rate: 7,200 RPM (120 RPS, musical note C3)
  - platters: 10
  - surfaces: 20
  - sector size: 512-732 bytes
  - cylinders: 11,494
  - cache: 4MB
  - transfer rate: 17.9 MB/s (inner) – 28.9 MB/s (outer)
  - full seek: 14.5 ms
  - head switch: 0.3 ms

Disk Performance

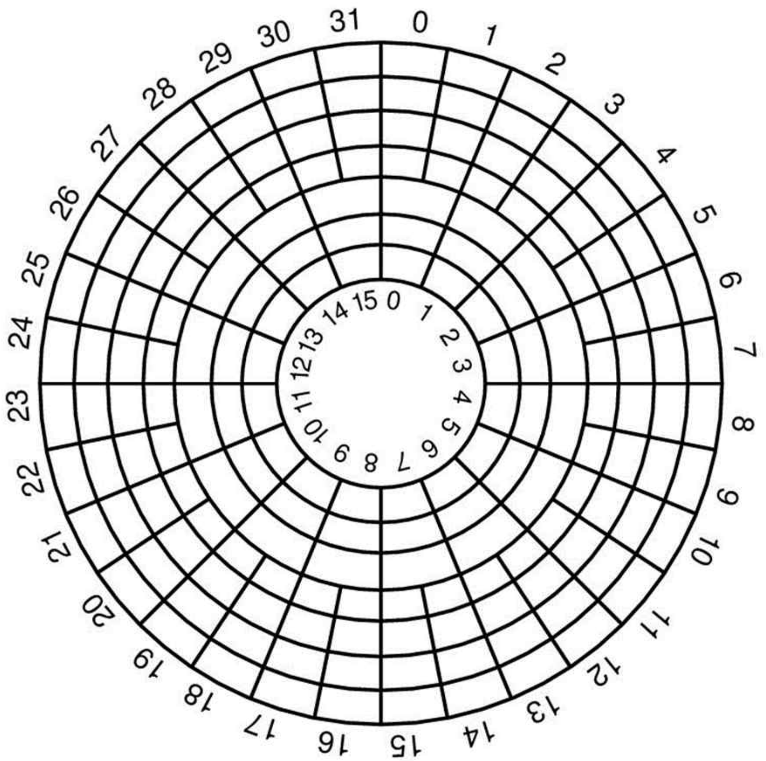
- Performance depends on a number of steps
  - seek: moving the disk arm to the correct cylinder
    - » depends on how fast disk arm can move
      - seek times aren't diminishing very quickly
  - rotation: waiting for the sector to rotate under head
    - » depends on rotation rate of disk
      - rates are increasing, but slowly
  - transfer: transferring data from surface into disk controller, and from there sending it back to host
    - » depends on density of bytes on disk
      - increasing, and very quickly
- When the OS uses the disk, it tries to minimize the cost of all of these steps
  - particularly seeks and rotation

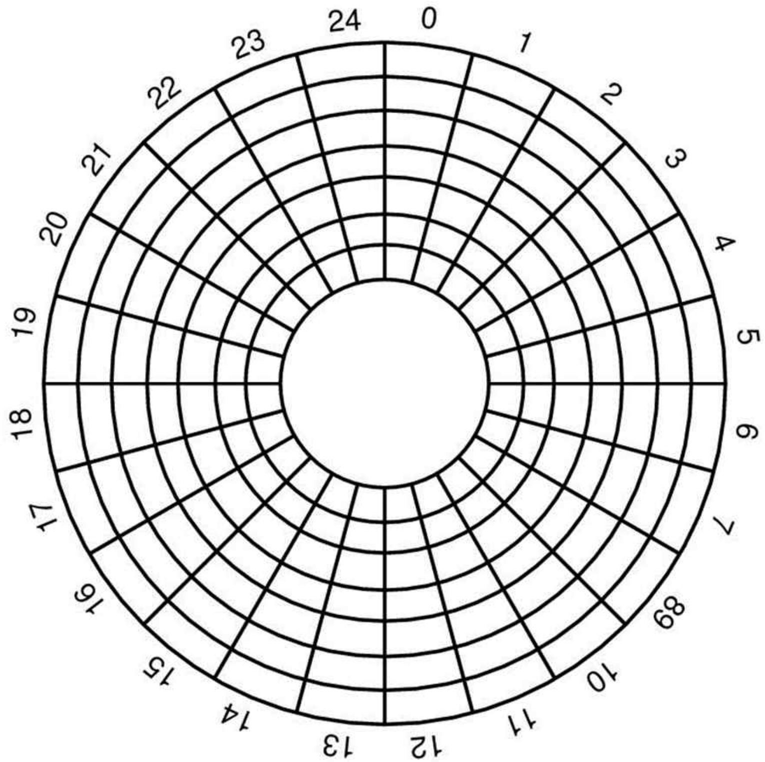
Disks
Disk Hardware (1)

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

Disk parameters for the original IBM PC floppy disk and a Western Digital WD 18300 hard disk

Disk Hardware (2)

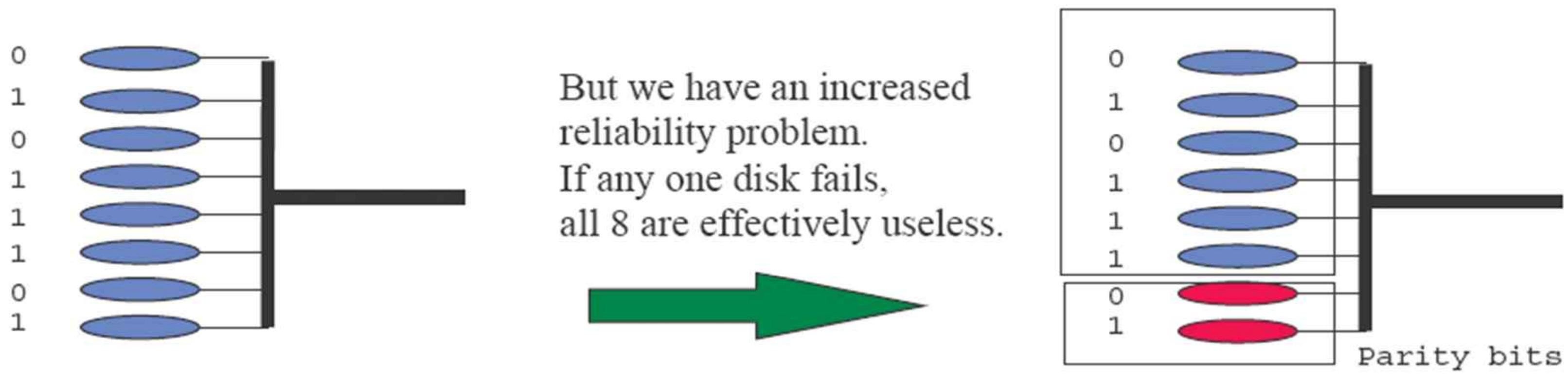




- **Physical geometry of a disk with two zones**
  - More sectors on the outer zone than the inner one
- **A possible virtual geometry for this disk**
  - View presented to the OS
  - Used by the driver software, different than the physical format

RAID
(Redundant Array of Independent Disks)

- **Caching, RAM disks deal with the latency issue.**
- **DISKS can also be used in PARALLEL**
- **This is the idea behind RAIDs**
  - **Redundant Array of Inexpensive Disks**
  - **Many RAID levels (Raid0-5)**



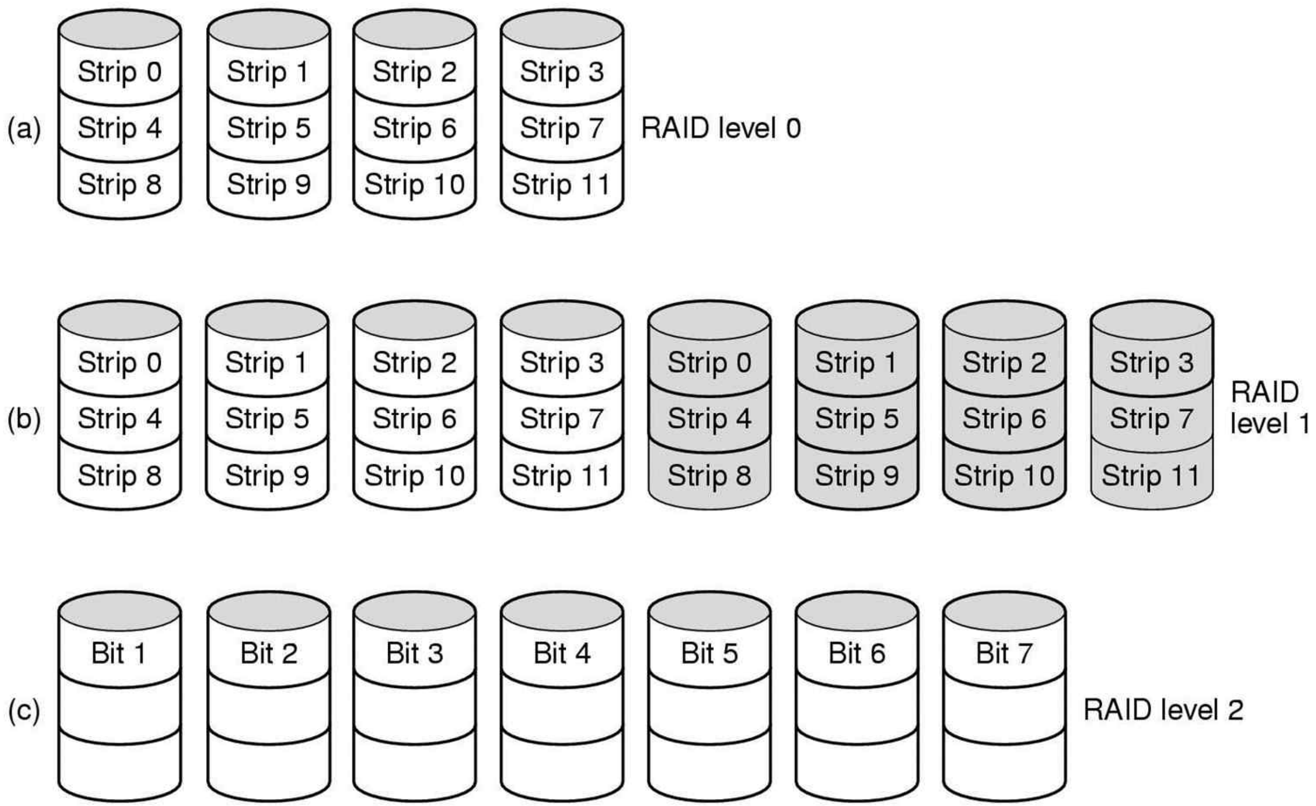
An array of inexpensive disks
(Can read 8 tracks at once)

A **redundant** array
of inexpensive
disks.

RAID

- **Level 0**
  - **Strips (*of k sectors*) are distributed in round-robin fashion**
  - **best with large requests; the implementation is straightforward**
  - **worst with requests that habitually ask for data one sector at a time : no parallelism, performance gain**
  - **No redundancy: reliability is worse than a SLED(Single Large Expensive Disk)**
- **Level 1**
  - **Duplicates all the disks**
  - **Read performance can be up to twice as good.**
- **Level 2**
  - **Works on a word or byte basis**
  - **E.g.. Each byte spitted into a pair of 4-bit nibbles, Hamming code added to form a 7-bit world**
  - **Drives should be rotionally synchronized**

RAID

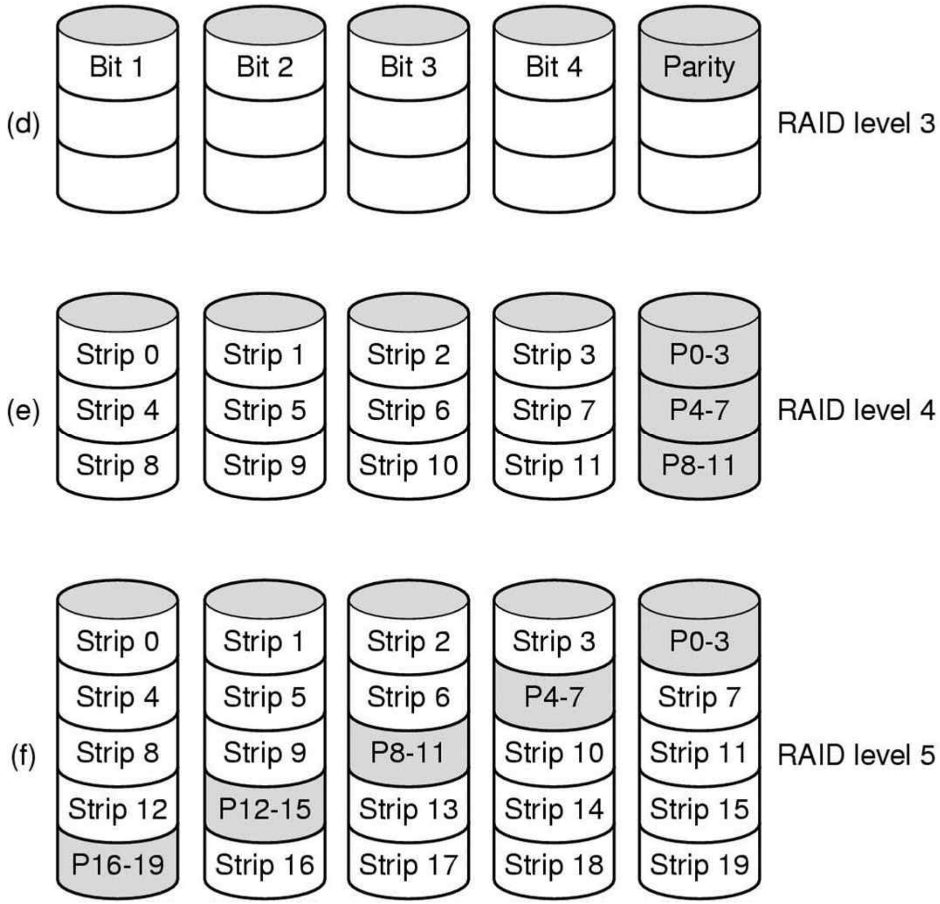


- **Raid levels 0 through 2**
- **Backup and parity drives are shaded**

RAID

- **Level 3**
  - **Single parity bit is computed for each data word and written to a parity drive**
  - **Drivers must be synchronized**
  - **Provide full 1-bit correction by knowing the position of the bad bit (crashed driver)**
- **Level 4**
  - **Like level 0 RAID with a strip-for-strip parity written onto an extra drive**
  - **Performs poorly for small updates**
    - » If one sector is changed, it is necessary to read all the drives in order to recalculate the parity, which must then be rewritten.
  - **Heavy load on the parity drive**
- **Level 5**
  - **Eliminates bottleneck in the parity drive by distributing parity bits uniformly over all the drives, round robin fashion**
  - **Reconstructing the contents of the failed drive is a complex process.**

Disk Hardware (4)



- **Raid levels 3 through 5**
- **Backup and parity drives are shaded**

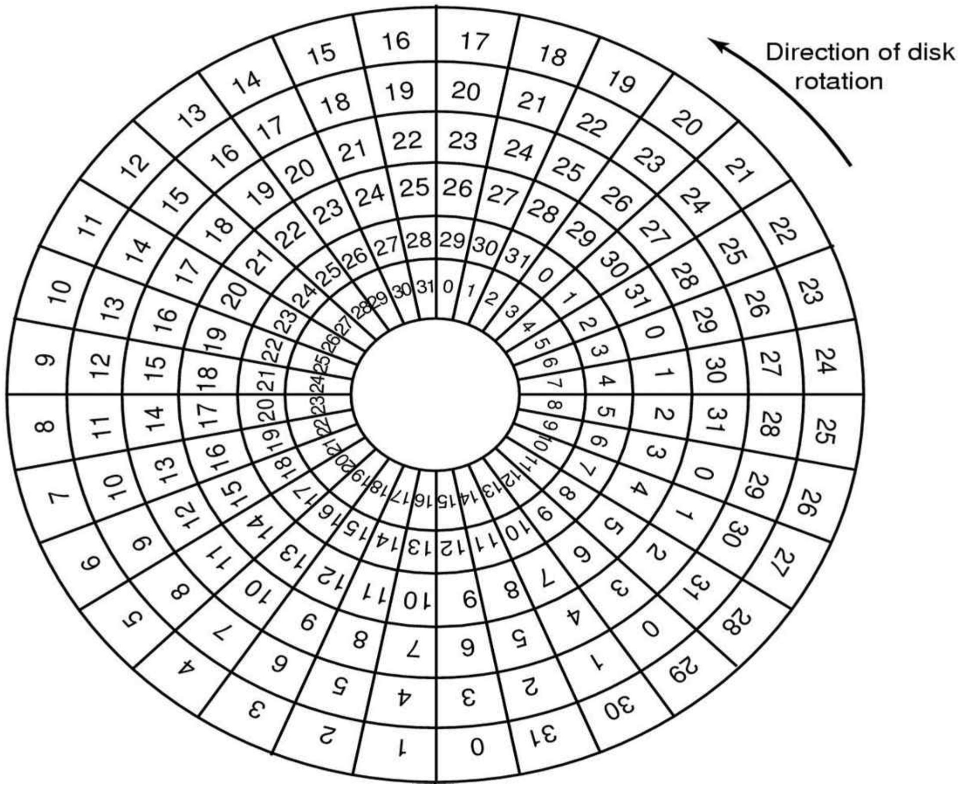
Disk Formatting
Low-level Formatting

Preamble	Data	ECC

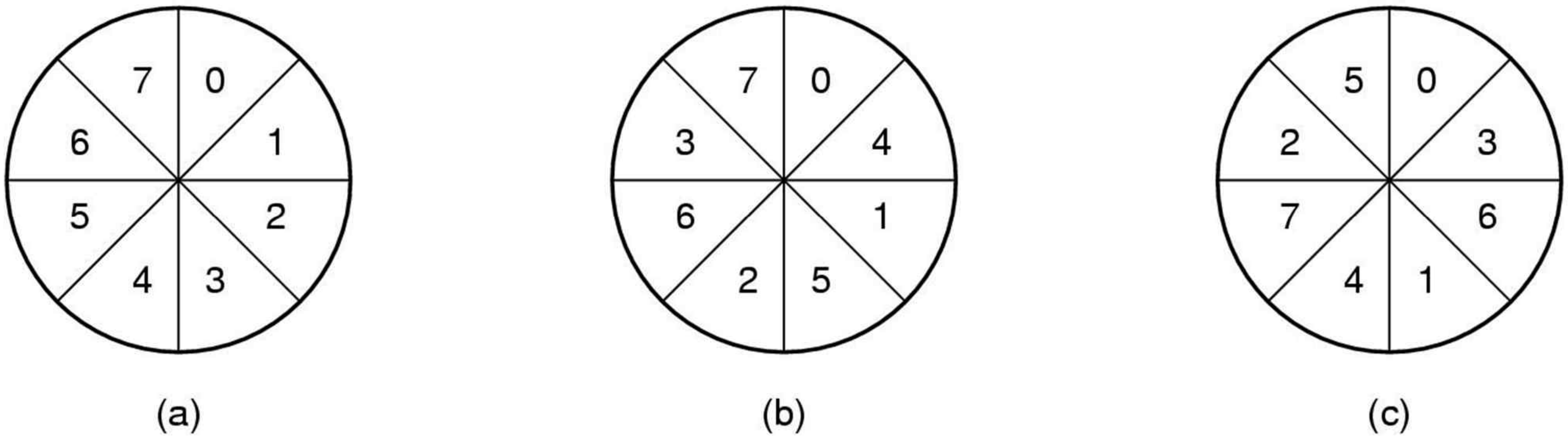
- **A disk sector**
  - **Preamble**
    - » Starts with a bit pattern that allows the hardware to recognize the start of the sector
    - » Contains the cylinder and sector numbers, etc.
  - **Data**
  - **ECC(Error-Correcting Code)**
    - » Redundant information that can be read to recover from read errors
- **Spare sectors**

Cylinder Skew

- **Cylinder skew**
  - **The position of sector 0 on each track is offset from the previous track when the low-level format is laid down.**
  - **To improve performance**
    - » Allows the disk to read multiple tracks in one continuous operation without losing data.



Disk Interleaving



(a) No interleaving

(b) Single interleaving

(c) Double interleaving

- To give the controller time to process the sector when consecutive sectors are read
- To avoid the need for interleaving, the controller should be able to buffer an entire track.

Sequence of Disk Formatting

- **Low-level formatting**
- **Partitioning**
  - **MBR (Master Boot Record)**
    - Boot code
    - Partition table
      - Starting sector and size of each partition
- **High-level formatting**
  - **Done for each partition separately**
  - **Creates a file system**
    - Boot block, super block, free storage adm., root dir, etc.

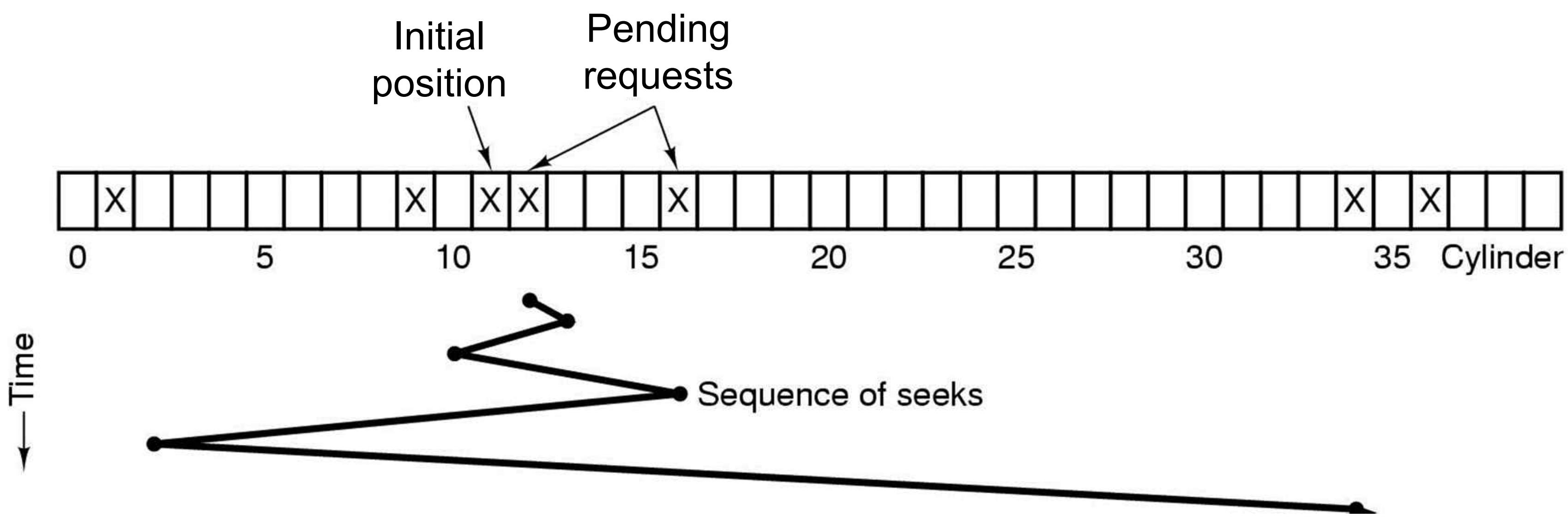
Disk Arm Scheduling Algorithms (1)

- Time required to read or write a disk block determined by 3 factors
    1. Seek time
    2. Rotational delay
    3. Actual transfer time
- Seek time dominates
- Error checking is done by controllers

Disk Scheduling

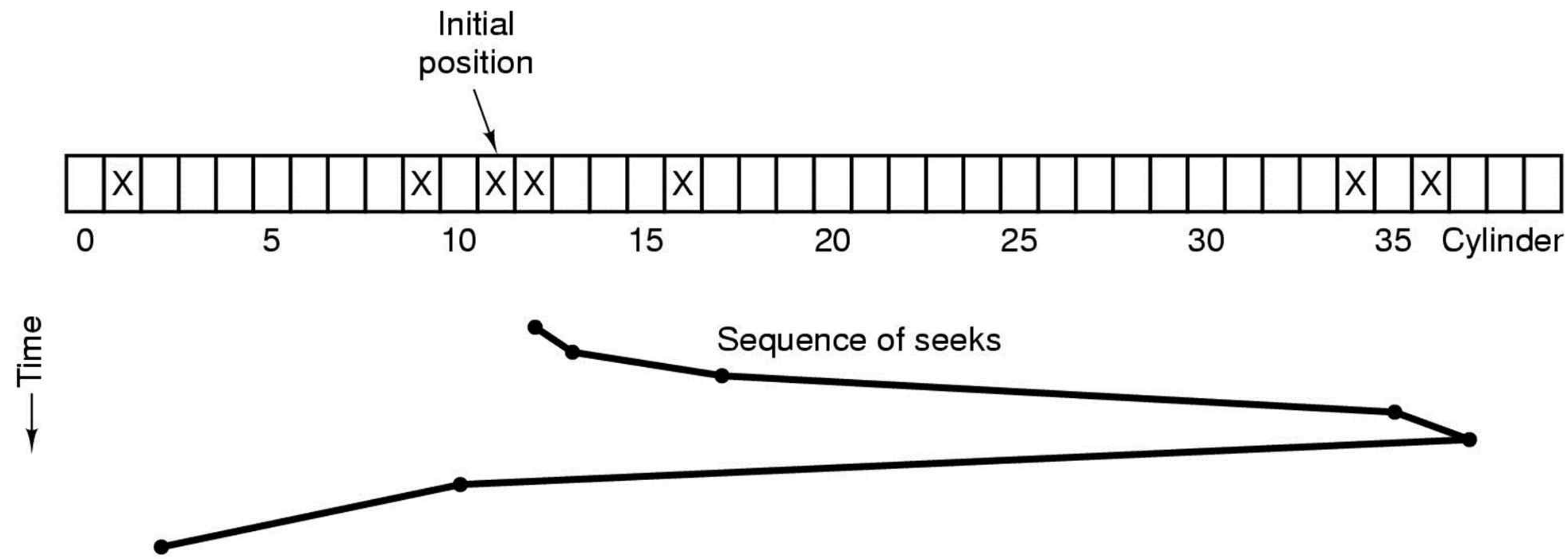
- **Seeks are very expensive, so the OS attempts to schedule disk requests that are queued waiting for the disk**
  - **FCFS (do nothing)**
    - » reasonable when load is low
    - » long waiting time for long request queues
  - **SSTF (shortest seek time first)**
    - » minimize arm movement (seek time), maximize request rate
    - » unfairly favors middle blocks
  - **SCAN (elevator algorithm)**
    - » service requests in one direction until done, then reverse
    - » skews wait times non-uniformly (why?)
  - **C-SCAN**
    - » like scan, but only go in one direction (typewriter)
    - » uniform wait times
- **If real disk geometry != virtual geometry, what happens! - disk controller may use the algs.**

Disk Arm Scheduling Algorithms (2)



- **Shortest Seek First (SSF) disk scheduling algorithm**
  - **Handles the closest request next, to minimize seek time**
  - **Request far from the middle may get poor service.**

Disk Arm Scheduling Algorithms (3)



- **The elevator algorithm for scheduling disk requests**
  - **Keeps moving in the same direction until there are no more requests in that direction, then switches direction**
  - **Requires one bit to keep track of the current direction: Up or Down**
  - **The upper bound on the total motion is fixed:**
    - » 2 * (number of cylinders)

C-Scan (Circular-Scan) Algorithm

- Similar to Scan
- Moves in one direction only (upward only)
- Smaller variance in response times than Scan
- The lowest-numbered cylinder is thought of as being just above the highest-numbered cylinder

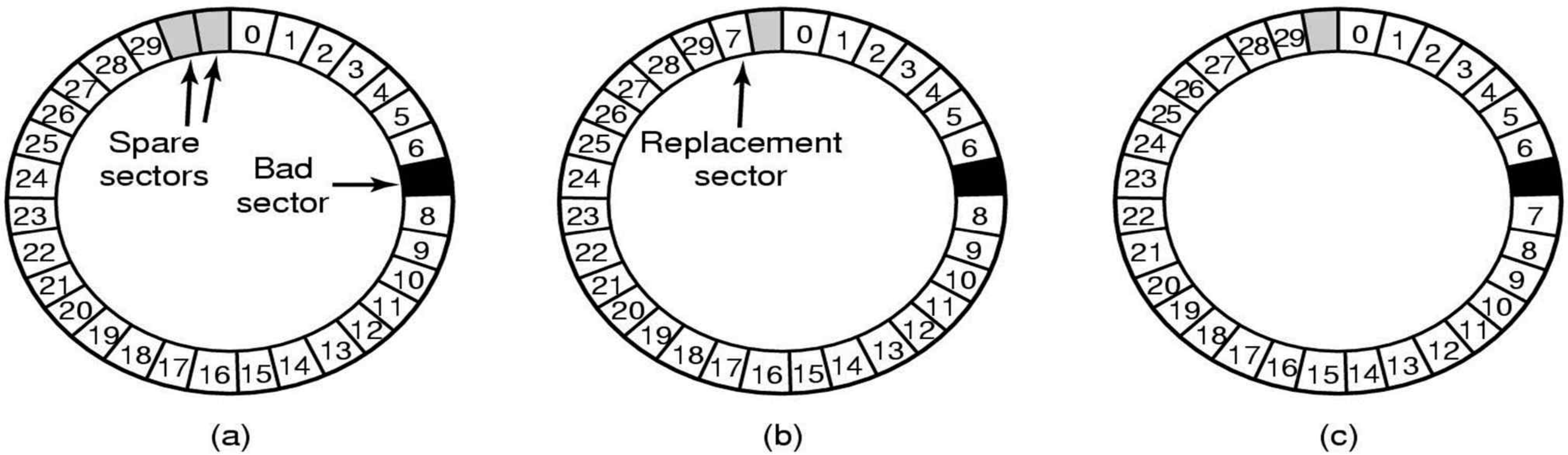
Other Optimization

- Any request to read a sector will cause that sector and much or all the rest of the current track to be read, depending upon how much space is available in the controller's cache memory

- Disk controller's cache holds blocks that have not actually been requested, while any cache maintained by the operating system will consist of blocks that were explicitly read.

Error Handling

- **Controller : bad sector substitution**



(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

- **OS :**
  - **Remapping**
  - **A secret file with all the bad sectors**