METU - EE 442 - Operating Systems

Input/Output

Chapter 5

Cüneyt F. Bazlamaççı (METU-EEE)

The slides include material from Tanenbaum & Bos, Modern Operating Systems, 4th ed., Pearson

Overview

- What is I/O?
- Principles of I/O hardware
- Principles of I/O software
- Methods of implementing input-output activities
- Organization of device drivers
- Specific kinds of devices

Overview

- OS controls I/O devices =>
  - Issue commands,
  - handles interrupts,
  - handles errors
- Provide easy to use interface to devices
  - Hopefully device independent
- First look at hardware, then software
  - Emphasize software
  - Software structured in layers
  - Look at disks (also clocks, keyboards, and displays)
  - Finally, consider power management

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

Hardware

- Electrical engineer’s view
  - chips, wires, power supplies, motors, and all the other physical components that comprise the hardware
- Programmer’s view
  - interface presented to the S/W
    - commands the hardware accepts,
    - the functions it carries out, and
    - errors that can be reported back

I/O Devices (1)

- Block devices
  - fixed-size blocks, each one with its own address.
  - independent transfers but in units of blocks
    - Hard disks, Blu-ray discs, USB sticks, etc.
  - Block sizes typically in the range of 512-32,768 bytes

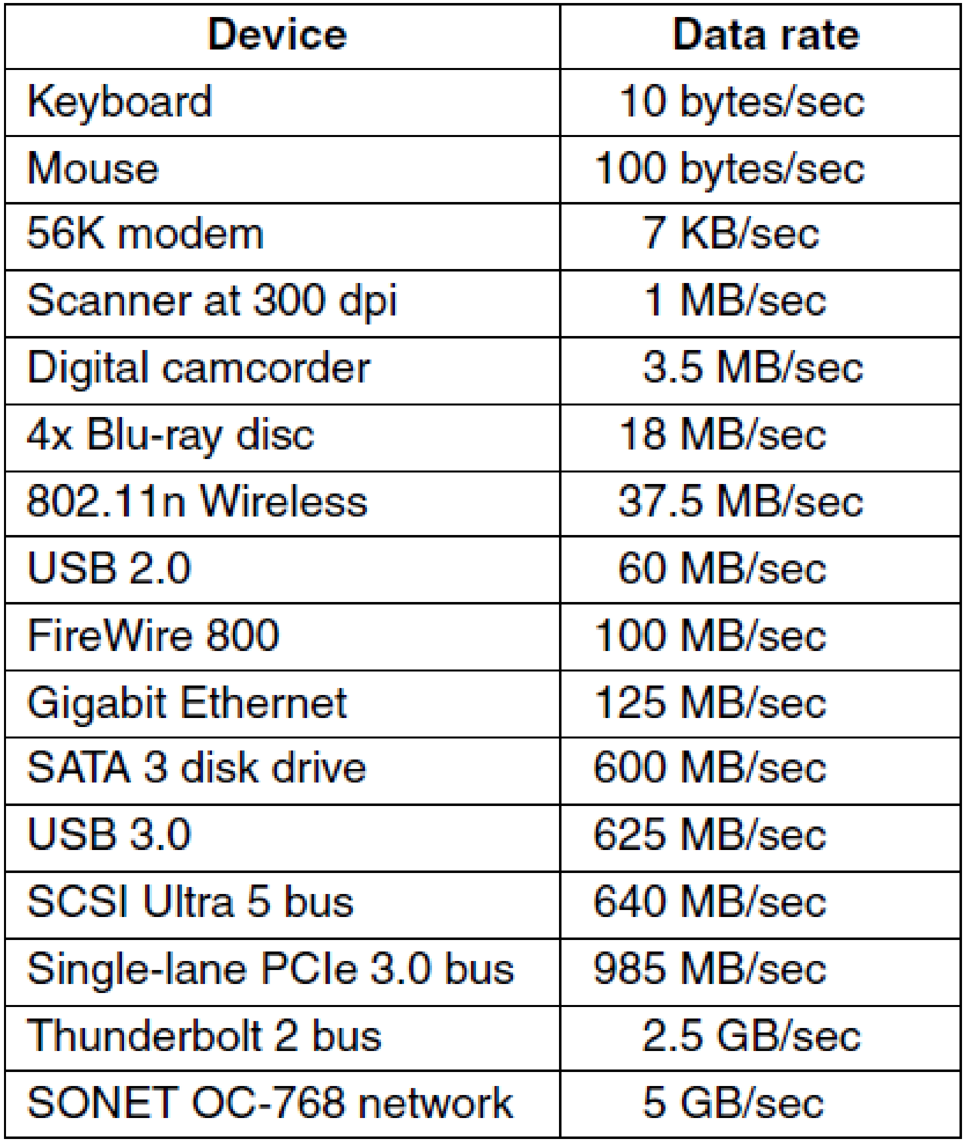
- Character devices
  - delivers or accepts stream of characters,
  - no block structure
  - not addressable, does not have any *seek* operation
    - printers, network interfaces, mice (for pointing), etc.

I/O Devices (2)

- Some devices do not fit in. For example
  - Clocks
    - are not block addressable
    - do not generate or accept character streams.
    - they cause interrupts at well-defined intervals.
  - Memory-mapped screens
  - Touch screens
- Block and character device models are general enough and can be used as a basis for making some of the OS software dealing with I/O devices independently.
  - The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software.

I/O Devices (3)

- I/O devices cover a huge range in speeds
- Puts considerable pressure on software to perform well over many orders of magnitude in data rates.

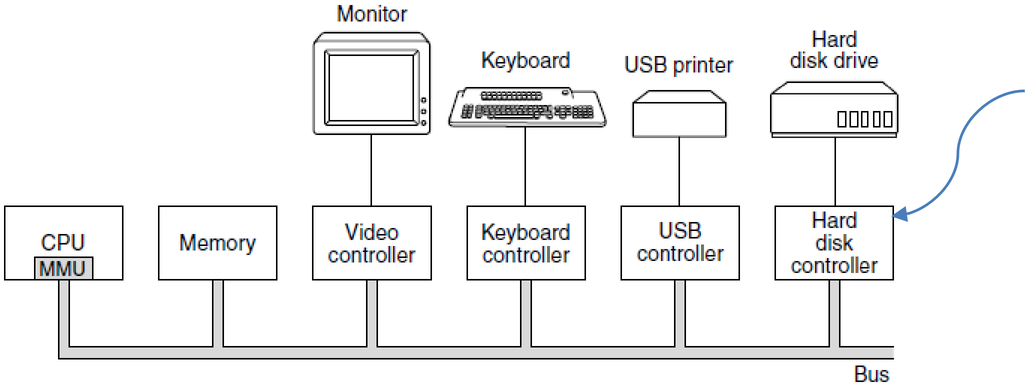


Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

Some typical device, network, and bus data rates.

Device Controllers

- Typical I/O unit
  - mechanical component
  - electronics component (device controller or adapter)
    - a chip on the motherboard,
    - a PCB on an expansion slot such as PCIe, etc.



Exampe:

- Disk might have 10,000 sectors of 512 bytes per track
- a serial bit stream comes out starting with
  - a **preamble** (with sector number, cylinder number, sector size), then
  - 4096 bits/sector, and finally
  - a checksum, or ECC (Error-Correcting Code).

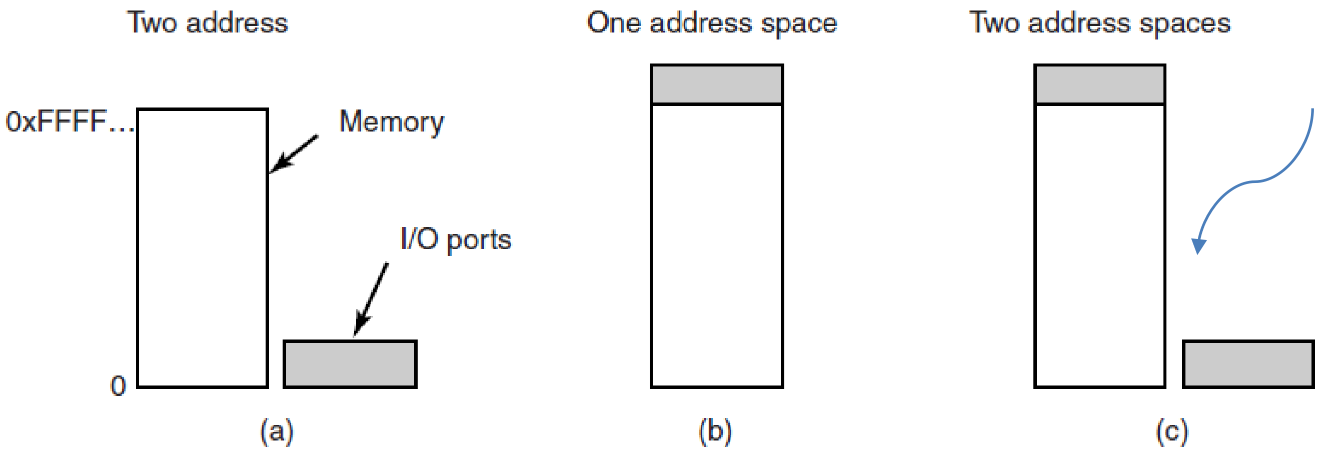
- The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary.
-
- Blocks are what are sent from a disk controller.

Device Controllers

- Each controller has few registers that are used to communicate with the CPU.
  - By writing into these registers, OS can command the device to
    - deliver data,
    - accept data,
    - switch itself on or off, or
    - perform some other action.
  - By reading from these registers, OS can learn
    - what the device’s state is,
    - whether it is prepared to accept a new command, etc.
- In addition, many devices have a data buffer that the OS can read and write.
  - For example, computers display pixels on the screen using a video RAM, just a data buffer, available for programs or the OS to write into.

Memory-Mapped I/O (1)

- How does the CPU communicates with the control/status registers and also with the device data buffers?



X86 arcitecture

(a) Separate I/O and memory space (b) Memory-mapped I/O (c) Hybrid

(a)

IN REG,PORT, CPU can read in control register PORT and store the result in CPU register REG.

OUT PORT,REG CPU can write into control register PORT the contents of CPU register REG.

IN R0,4 and MOV R0,4 are completely different.

How does CPU address registers and buffers?

- Separate I/O and memory space approach
  - puts read signal on control bus
  - puts address info. on address bus
  - puts I/O space or memory space signal on control bus to differentiate between memory or I/O
  - read from memory or I/O space
- Memory mapped approach
  - puts address info. on address bus and let memory and I/O devices compare the address with the ranges they serve

Advantages of memory mapped I/O

- no need for special instructions to read/write control/status registers (you can write a device driver purely in C)
- no need for special protection to keep users from doing I/O directly
  - Just don't put I/O memory in any user space

- An instruction can reference control registers and memory

```
loop  test port_4      //check if port 4 is zero
      beq ready       //it is is zero, go to ready
      branch loop   //otherwise, continue testing
ready …
```

  - If instruction just references registers, then we may need more instructions to do the read first and then test it, etc.

Disadvantages of memory mapped I/O

```
loop  test port_4      //check if port 4 is zero
      beq ready        //it is is zero, go to ready
      branch loop   //otherwise, continue testing
ready …
```

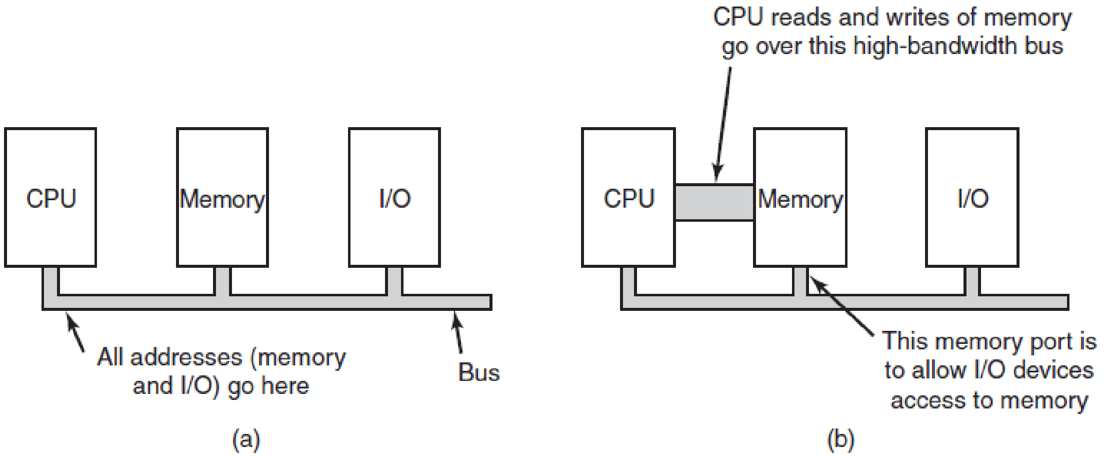
Disadvantages of Memory mapped I/O :

1. Caching a device control register can be disastrous.
    - Consider the above code in the presence of caching.
    - The first reference to port_4 would cause it to be cached.
    - Subsequent references would take the value from the cache and not even ask the device.
    - Then when the device finally becomes ready, the software has no way of finding out.
    - It loops forever.

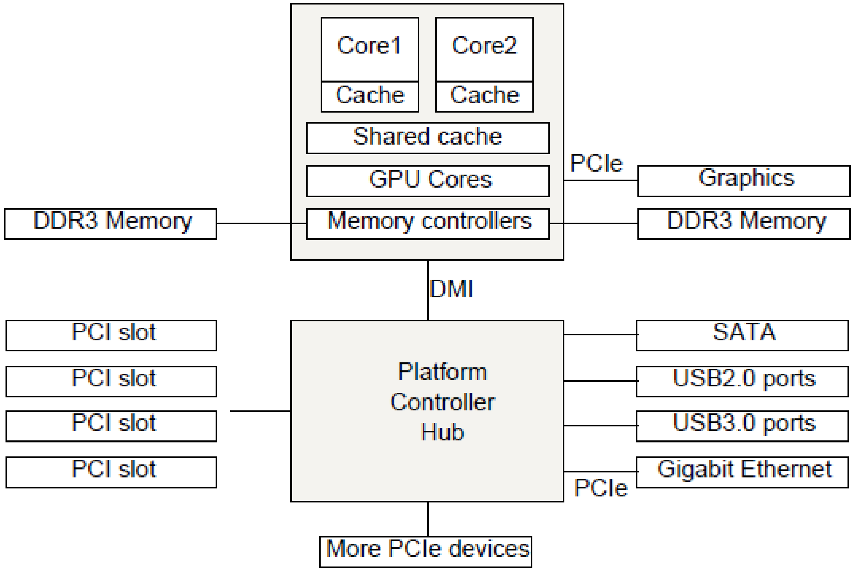
    - Solution: Perform selective caching on per page basis (extra complexity on H/W)

2. If there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to.

Memory-Mapped I/O (2)


CPU reads and writes of memory go over this high-bandwidth bus
CPU
Memory
I/O
CPU
Memory
I/O
All addresses (memory and I/O) go here
Bus
This memory port is to allow I/O devices access to memory
(a)
(b)

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



The structure of a large x86 system

Solutions for memory mapped I/O with dual bus structures

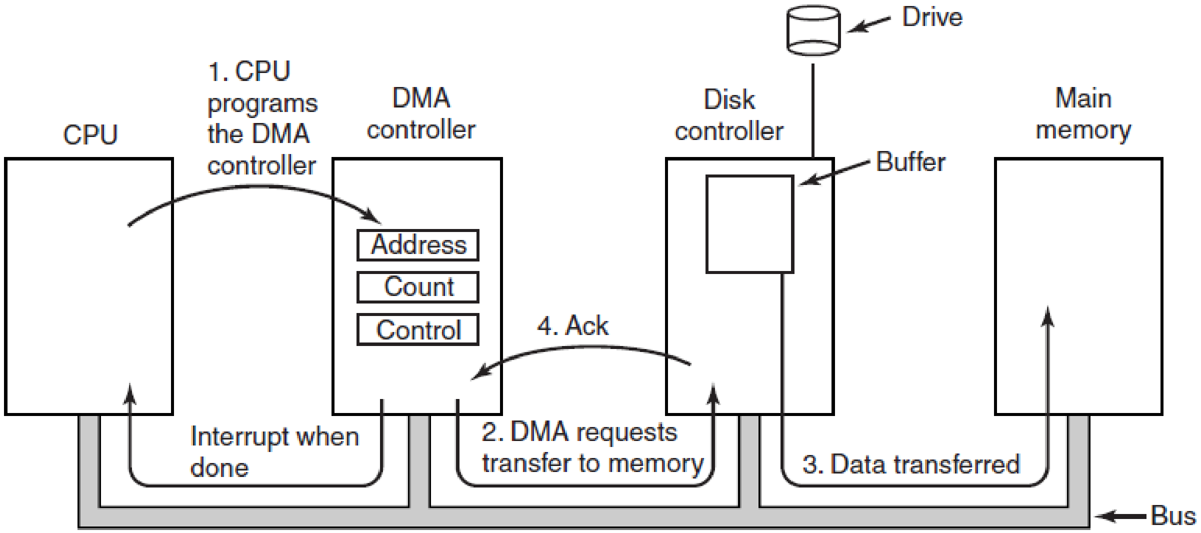
- CPU tries memory first. If it does not get a response then it tries I/O device.
- Snooping the devices, filtering addresses

- Main point: all complicates hardware

Direct Memory Access

- CPU could request data one byte at a time from I/O controller
- Big waste of time, hence use DMA
- DMA controller on mother-board; normally one controller for multiple devices
- CPU reads/writes to registers in DMA controller
  - memory address register
  - byte count register
  - control registers - I/O port, direction of transfer, transfer units (byte/word), number of bytes to transfer in a burst, etc.

Direct Memory Access



Operation of a DMA transfer.

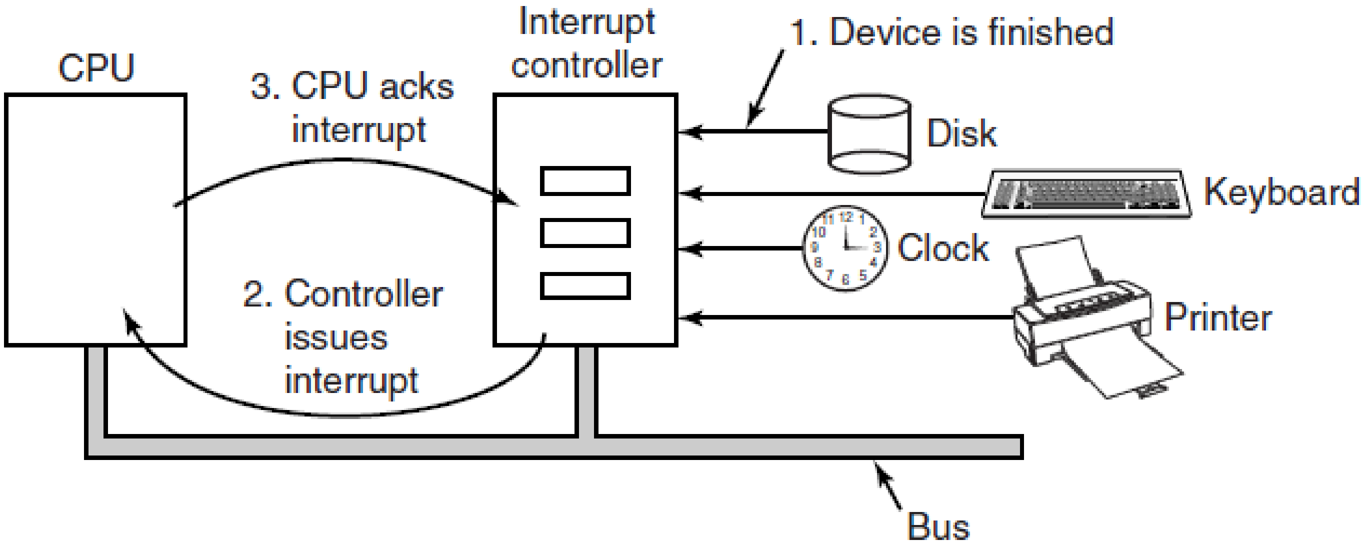
DMA controller modes

- Cycle stealing mode
  - transfer goes on word at a time, competing with CPU for bus cycles.
  - Idea is that CPU loses the occasional cycle to the DMA controller
- Burst mode
  - DMA controller grabs bus and sends a block
- Fly by mode
  - DMA controller tells device controller to send word to it instead of memory. Can be used to transfer data between devices.

Questions

- Why buffer data in controllers?
  - Can do check-sum
  - Bus may be busy
    - need to store data someplace
- Is DMA really worth it? Not if
  - CPU is much faster then DMA controller and can do the job faster
  - don’t have too much data to transfer

Interrupts Revisited



How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Interrupt processing

- Controller puts number on address line telling CPU which device wants attention and interrupts CPU
- Table (interrupt vector) points to interrupt service routine
- Number on address line acts as index into interrupt vector
- Interrupt vector contains PC which points to start of service routine

Interrupt processing

- Interrupt service routine acks interrupt
- Saves information about interrupted program
- Where to save information
  - User process stack, kernel stack are both possibilities
  - Both have problems

Can we save the PC and PSW?

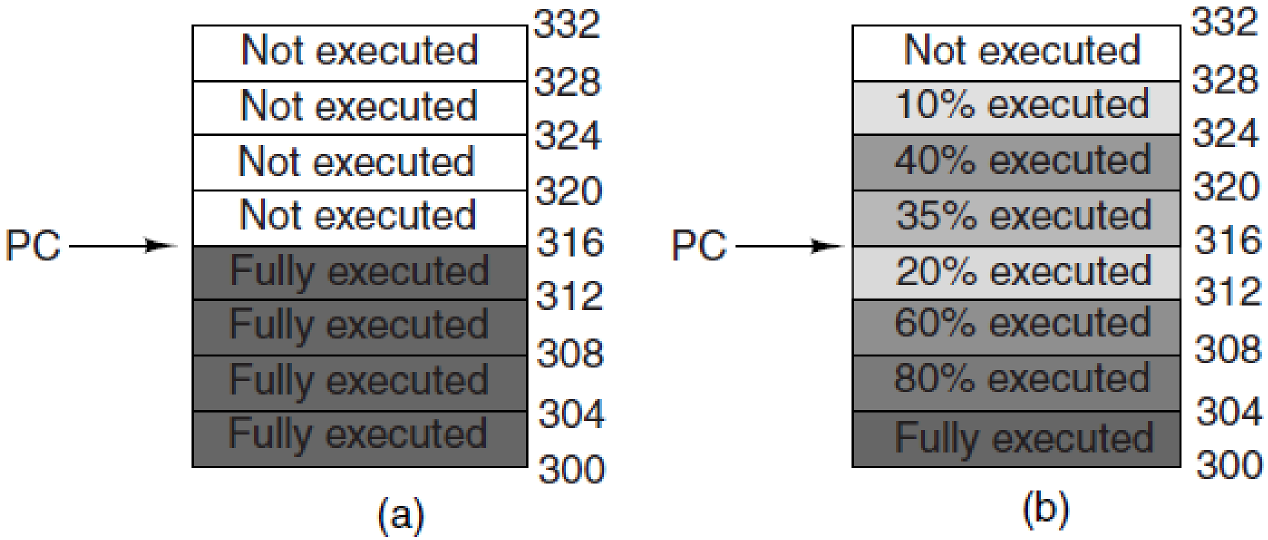
- Sure, if we don’t use pipelined or superscaler CPU’s. But we do use them.
- Can’t assume that all instructions up to and including given instruction have been executed
  - Pipeline-bunch of instructions are partially completed
  - Superscaler-instructions are decomposed and can execute out of order

Precise Interrupt

Four properties of a *precise interrupt*:

1. The PC saved in a known place.
2. All instructions before the address pointed by PC have fully executed.
3. No instruction beyond the address pointed by PC has been executed.
4. Execution state of instruction at the address pointed by PC is known.

Precise vs. Imprecise



(a) A precise interrupt. (b) An imprecise interrupt.

How to process an imprecise interrupt?

- With great difficulty
- Either need complicated hardware logic to re-start after interrupt (Pentium)
- Or have complicated processing in the OS

Goals of the I/O Software

Issues:

- Device independence
  - don’t have to specify the device in advance when accessing the device
  - e.g. a program that reads a file as input should be able to read a file on a hard disk, a DVD, or on a USB stick without having to be modified for each different device
- Uniform naming
  - name should not depend on device type (should simply be a string or an integer)
- Error handling
  - do it as close to the device as possible (e.g. controller should be the first to fix error, followed by the driver)
- Synchronous versus asynchronous
  - blocking or unblocking
- Buffering
  - when a packet arrives it may need to be buffered

I/O Types

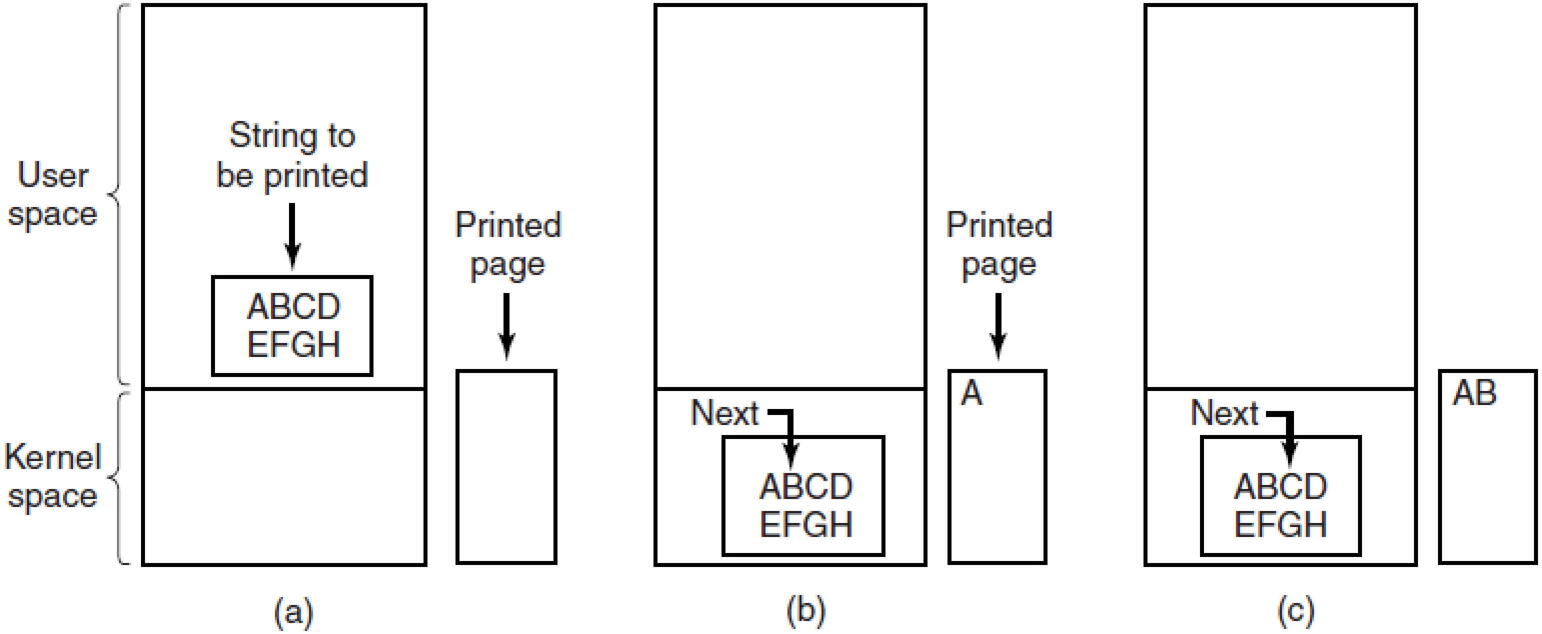
I/O can be performed in three fundamentally different ways:

- programmed I/O
- interrupt-driven I/O
- I/O using DMA

Programmed I/O (simplest one)

- have the CPU do all the work

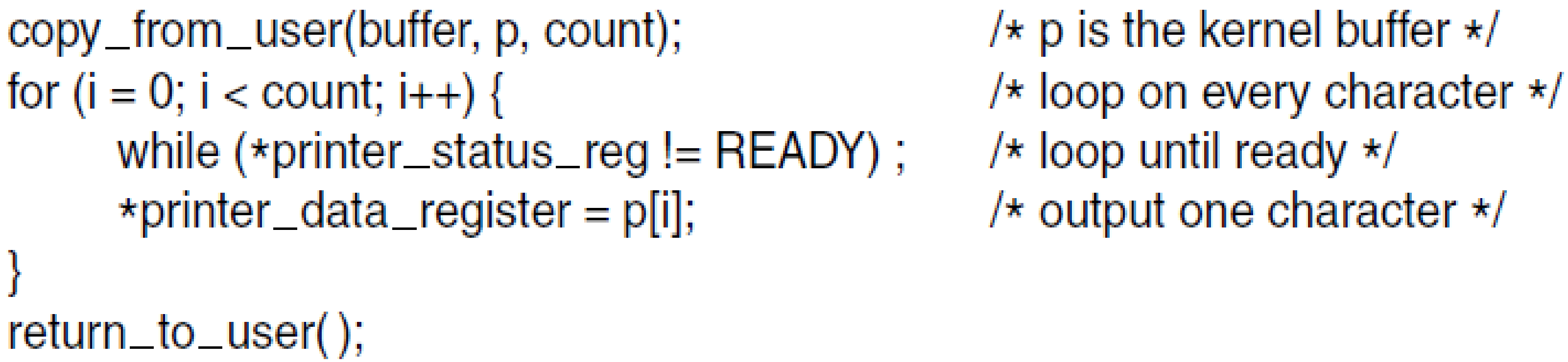
Programmed I/O (1)



Steps in printing a string via a serial interface

Programmed I/O (2)

- Send data one character at a time using polling (busy waiting)



```
copy_from_user(buffer, p, count);                /* p is the kernel buffer */
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( );
```

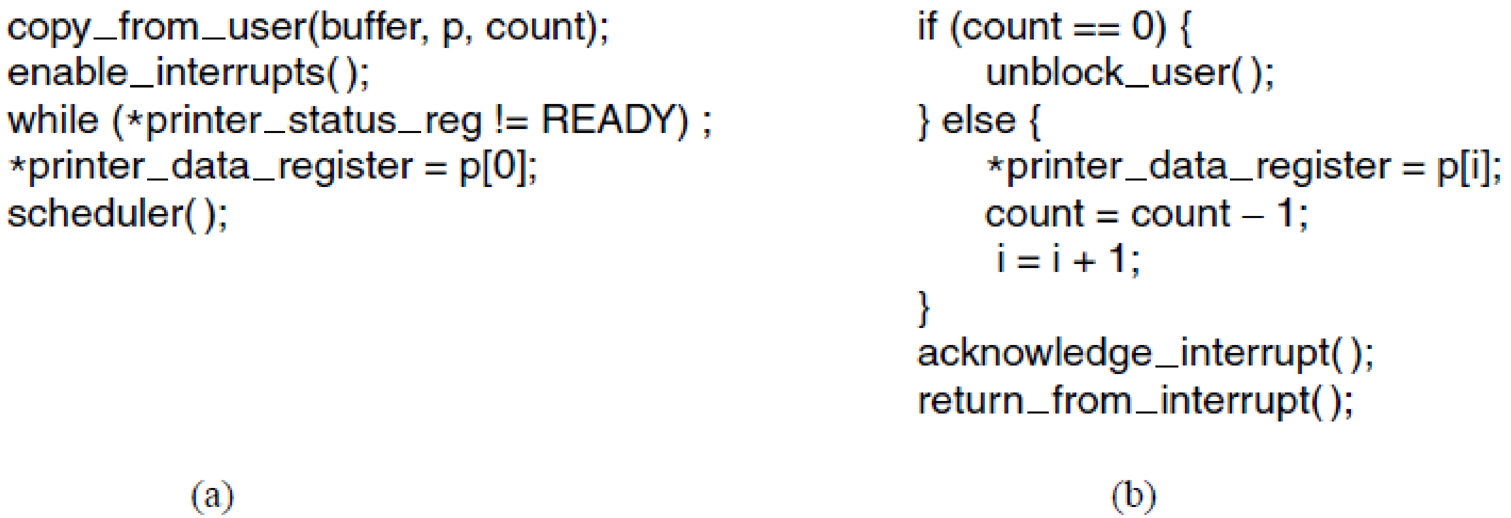
- Programmed I/O is simple (GOOD) but tying up the CPU full time until all the I/O is done (BAD).
- Acceptable for example if time to ‘‘print’’ a character is very short or CPU has nothing else to do.

Interrupt Driven I/O

Idea

- block process which requests I/O, schedule another process
- return to calling process when I/O is done
- printer generates interrupt when a character is printed
- keeps printing until the end of the string
- re-instantiate calling process

Interrupt-Driven I/O



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

(a)

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

(b)

Writing a string to the printer using interrupt-driven I/O.

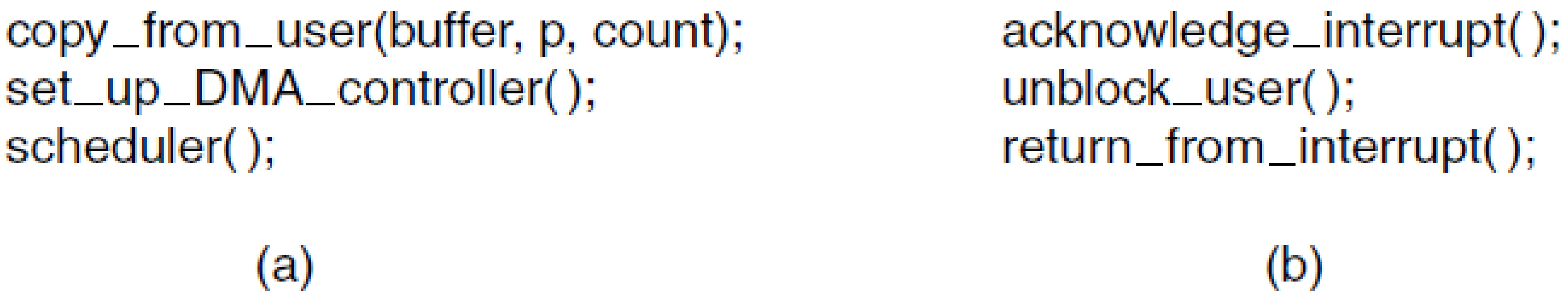
(a) Code executed at the time the print system call is made.

(b) Interrupt service procedure for the printer.

DMA

- Use DMA controller to send characters to printer instead of using the CPU
- CPU is only interrupted when the buffer is printed instead of when each character is printed
- DMA is worth it if
  - DMA controller can drive the device as fast as the CPU could drive it
  - there is enough data to make it worthwhile

I/O Using DMA



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

(a)

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

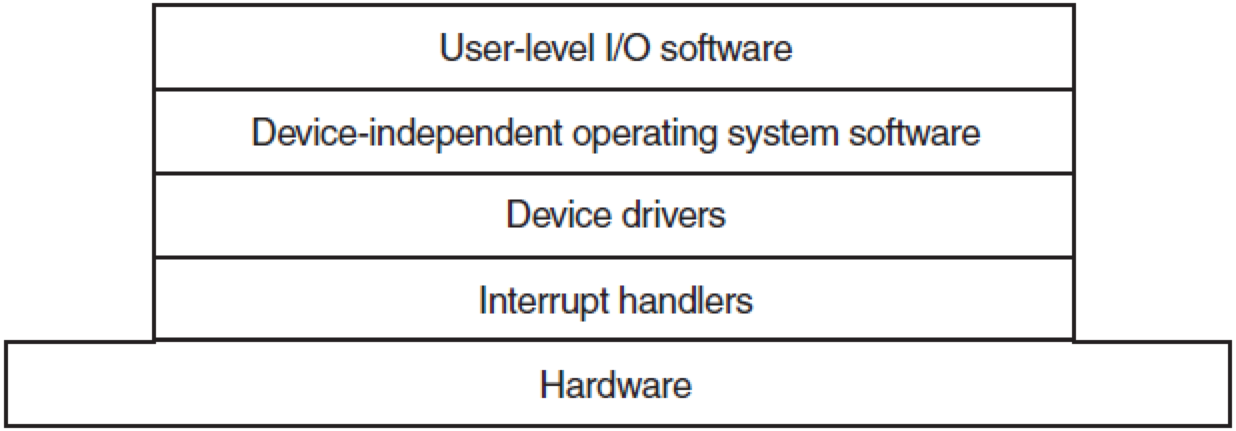
(b)

Printing a string using DMA.

(a) Code executed when the print system call is made.

(b) Interrupt service procedure.

I/O Software Layers



User-level I/O software

Device-independent operating system software

Device drivers

Interrupt handlers

Hardware

Layers of the I/O software system.

Interrupt Handlers

Idea:

- driver starting the I/O blocks until I/O finishes and interrupt happens
- handler processes interrupt
- wakes up the driver when processing is finished
- drivers are kernel processes with their very own
  - stacks
  - PCs
  - states

Interrupt Handlers (1)

Typical steps of hardware interrupt processing:

1. Save registers (including the PSW) not already saved by interrupt hardware.
2. Set up context for interrupt service procedure.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge interrupt controller.

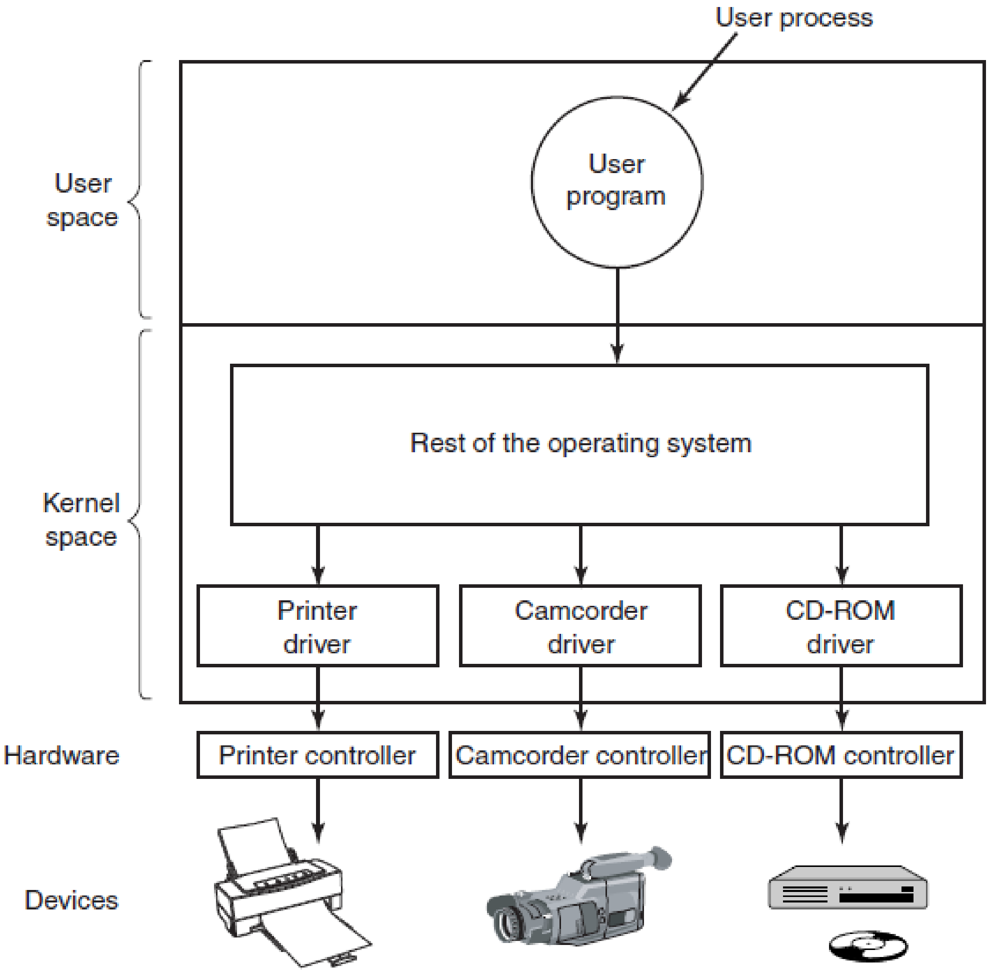
   - If no centralized interrupt controller, re-enable interrupts.

5. Copy registers to process table from where they were saved .

Interrupt Handlers (2)

6. Run interrupt service procedure. Extract information from interrupting device controller’s registers.
7. Choose which process to run next.
8. Set up the MMU context for process to run next.
9. Load new process’ registers, including its PSW.
10. Start running the new process.

Device Drivers



Logical positioning of device drivers.
(In reality all communication between drivers and device controllers go over the bus)

Device Drivers-Act 1

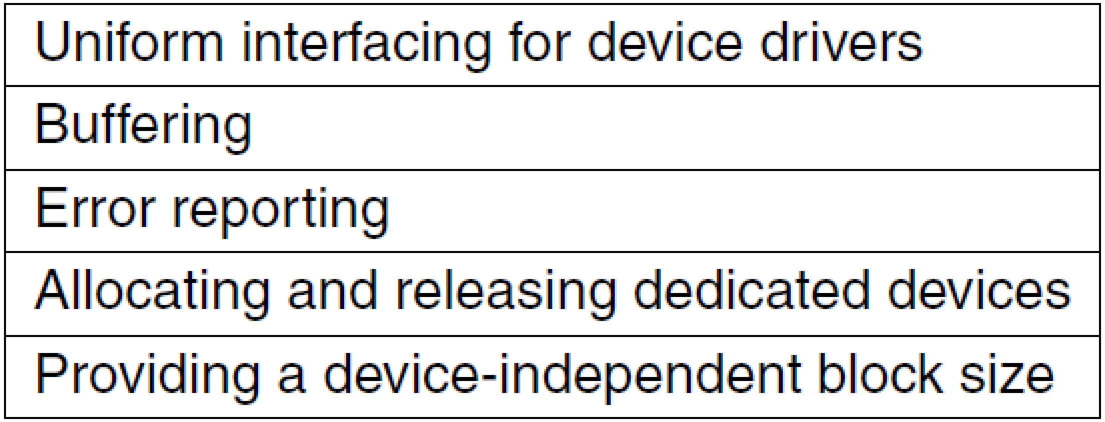
- Driver contains code specific to the device
- Supplied by manufacturer
- Installed in the kernel
- User space might be better place
- Why? Bad driver can mess up kernel
- Need interface to OS
  - block and character interfaces
  - procedures which OS can call to invoke driver (e.g. read a block)

Device drivers Act 2

- Checks input parameters for validity
- Abstract to concrete translation (block number to cylinder, head, track, sector)
- Check device status. Might have to start it.
- Puts commands in device controller’s registers
- Driver blocks itself until interrupt arrives
- Might return data to caller
- Does return status information

- Drivers should be re-entrant
- OS adds devices when system (and therefore driver) is running

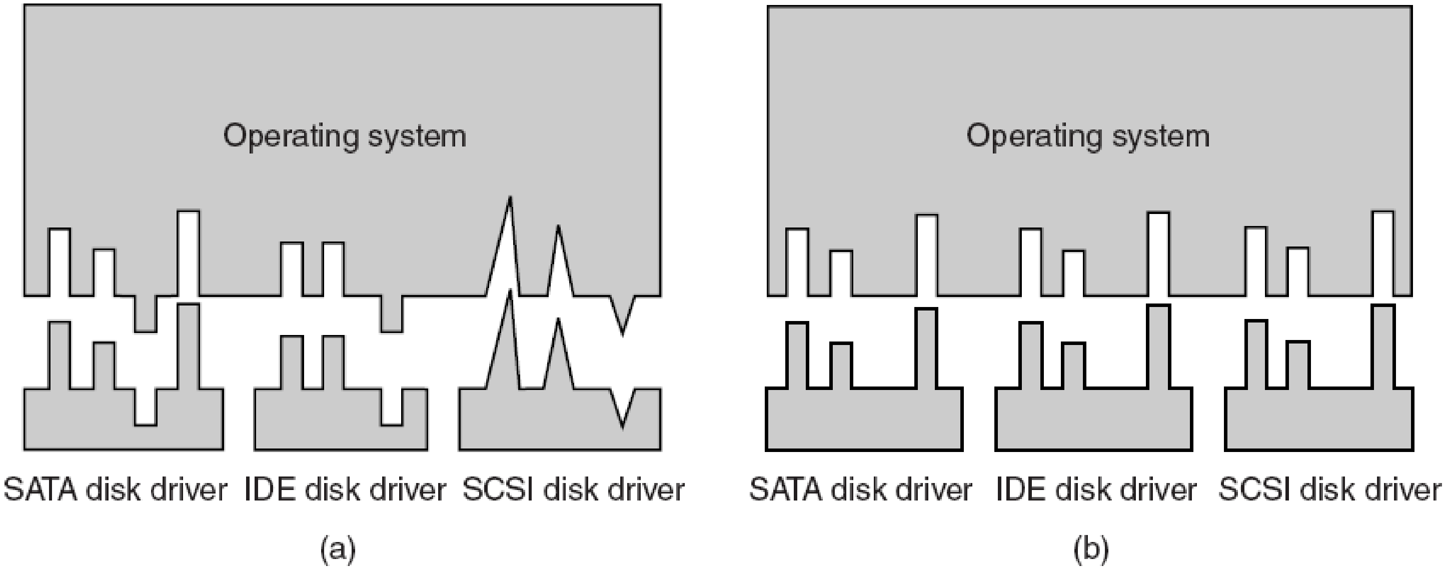
Device-Independent I/O Software



Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Functions of the device-independent I/O software.

Why the OS needs a standard interface



- Driver functions differ for different drivers
- Kernel functions which each driver needs are different for different drivers
- Too much work to have new interface for each new device type

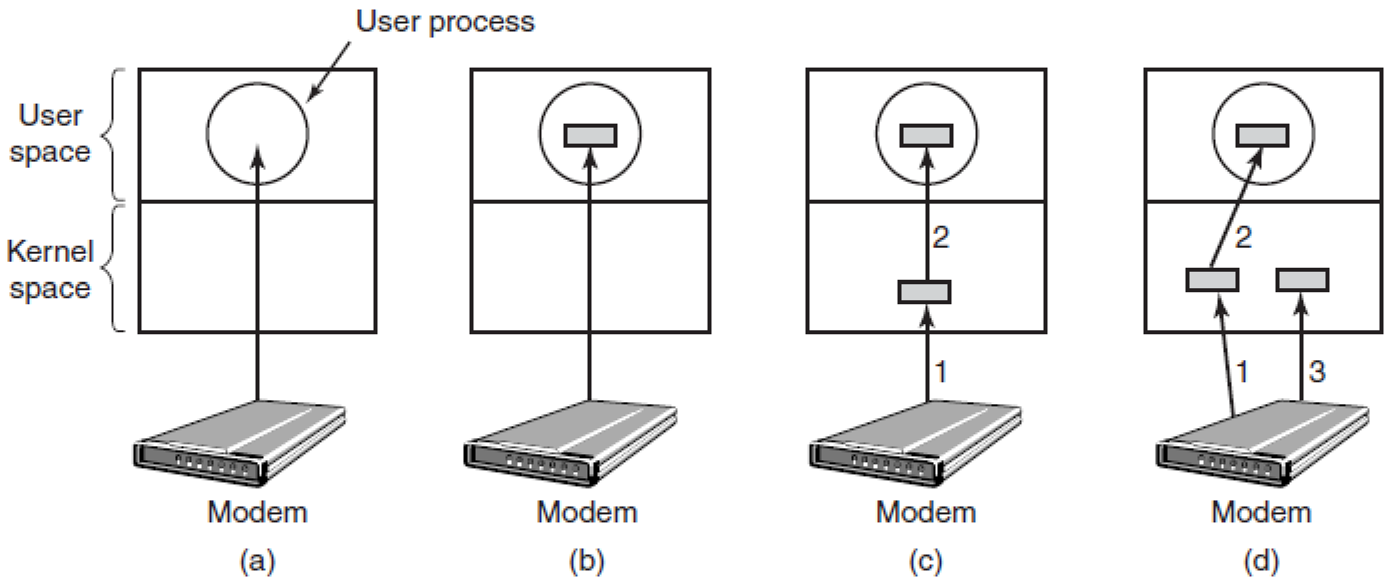
Interface: Driver functions

- OS defines functions for each class of devices which it MUST supply, e.g. read, write, turn on, turn off……..
- Driver has a table of pointers to functions
- OS just needs table address to call the functions

Interface: Names and protection

- OS maps symbolic device names onto the right driver
- Unix: /dev/disk0 maps to an i-node which contains the major and minor device numbers for disk0
  - Major device number locates driver, minor device number passes parameters (e.g. disk)
- Protection: In Unix and Windows devices appear as named objects => can use file protection system

Buffering (1)



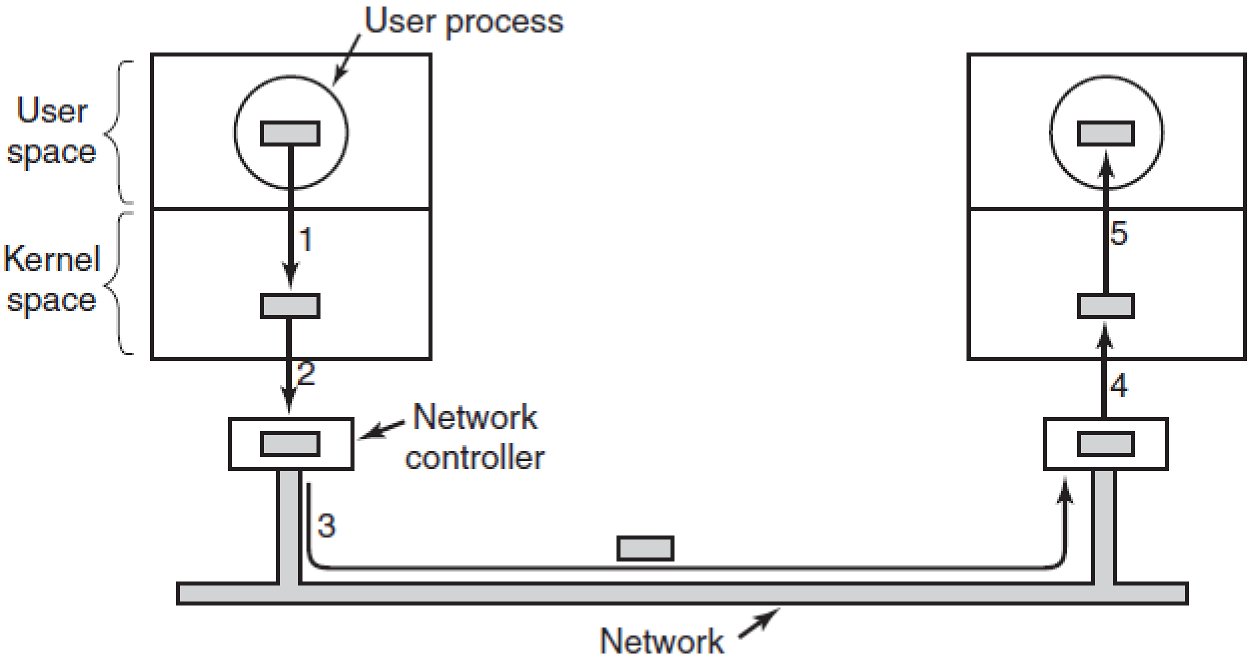
(a) Unbuffered input.

(b) Buffering in user space.

(c) Buffering in the kernel followed by copying to user space.

(d) Double buffering in the kernel.

Buffering (2)



Networking may involve many copies of a packet.

More Functions of Independent Software

- Error reporting - programming errors (the user asks for the wrong thing), hardware problems (bad disk) are reported if they can’t be solved by the driver
- Allocates and releases devices which can only be used by one user at a time (CD-ROM players)
  - Queues requests or
  - Lets open simply fail
- Device independent block size - OS does not have to know the details of the devices
  - e.g. combine sectors into blocks

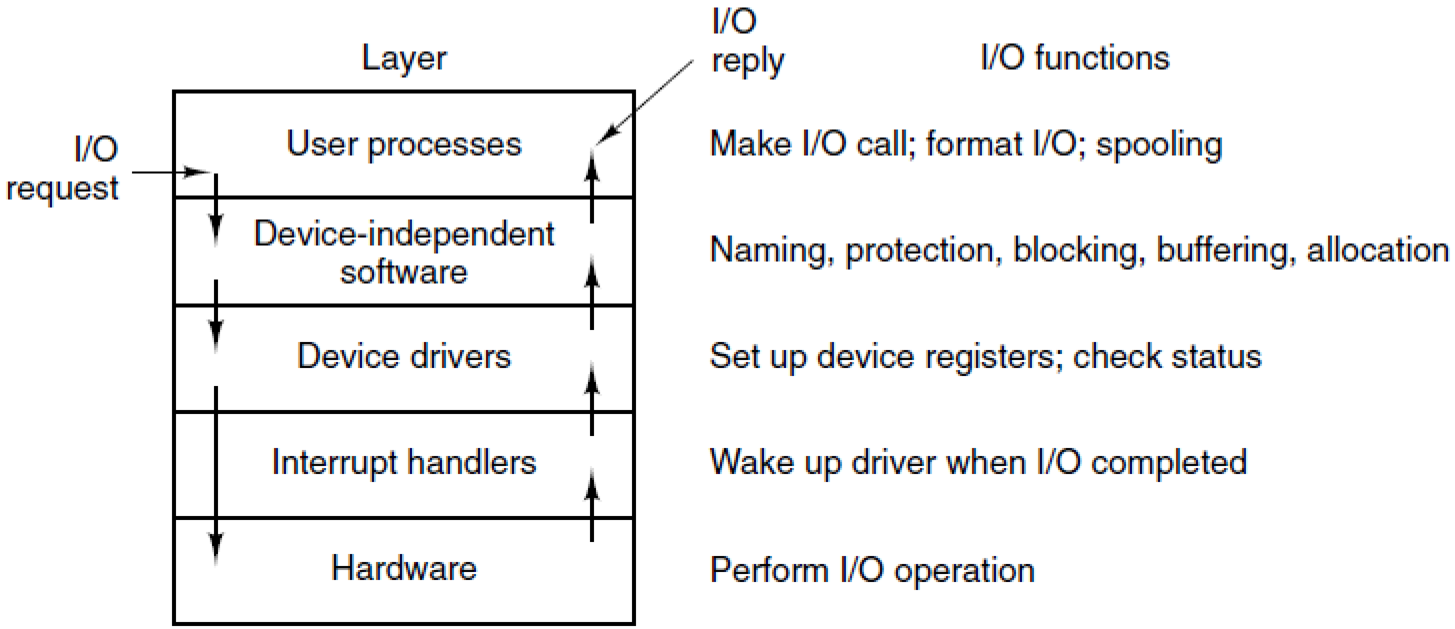
User Space I/O Software

- Library routines are involved with I/O - `printf`, `scanf, write` for example. These routines makes system calls.
- Spooling systems - keep track of device requests made by users.
- Think about printing.
  - User generates file, puts it in a spooling directory.
  - Daemon process monitors the directory, printing the user file
- File transfers also use a spooling directory

Example flow through layers

- User wants to read a block, asks OS
- Device independent software looks for block in cache
- If not there, invokes device driver to request block from disk
- Transfer finishes, interrupt is generated
- User process is awakened and goes back to work

User-Space I/O Software



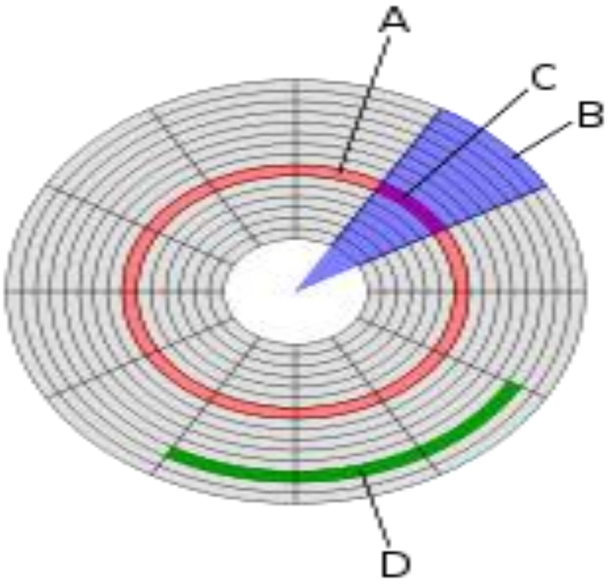
Layer	I/O functions
User processes	Make I/O call; format I/O; spooling
Device-independent software	Naming, protection, blocking, buffering, allocation
Device drivers	Set up device registers; check status
Interrupt handlers	Wake up driver when I/O completed
Hardware	Perform I/O operation

Layers of the I/O system and the main functions of each layer.

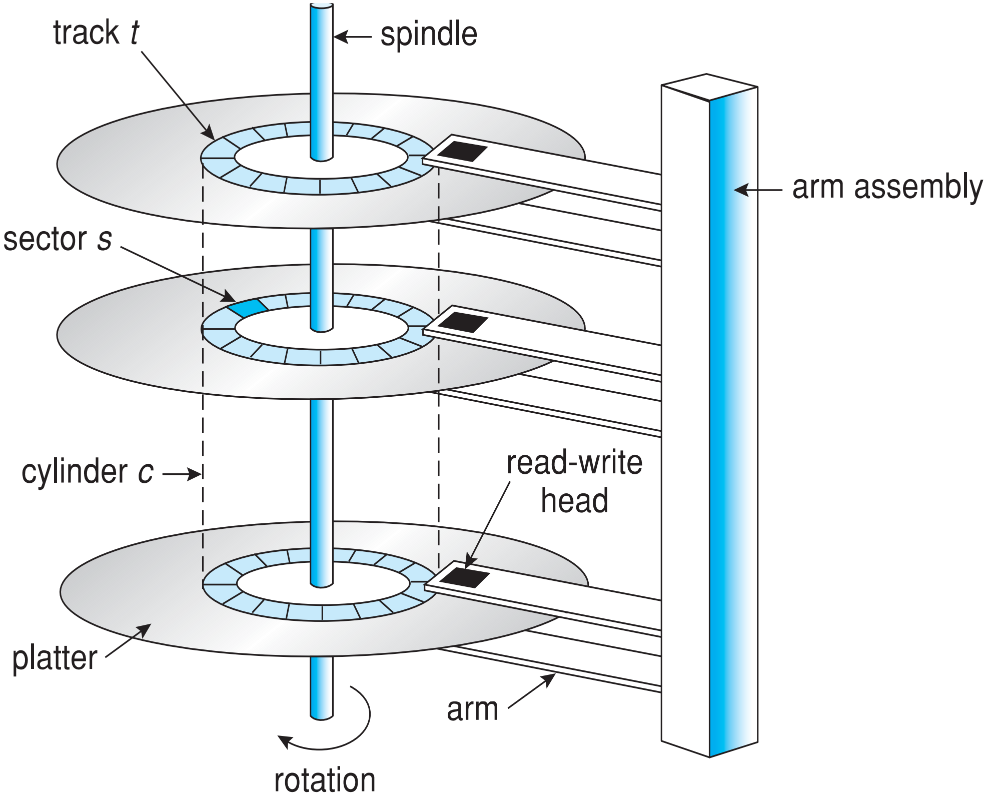
Disks

- Magnetic (hard)
  - Reads and writes are equally fast => good for storing file systems
  - Disk arrays are used for reliable storage (RAID)
- Optical disks (CD-ROM, CD-Recordables, DVD) used for program distribution

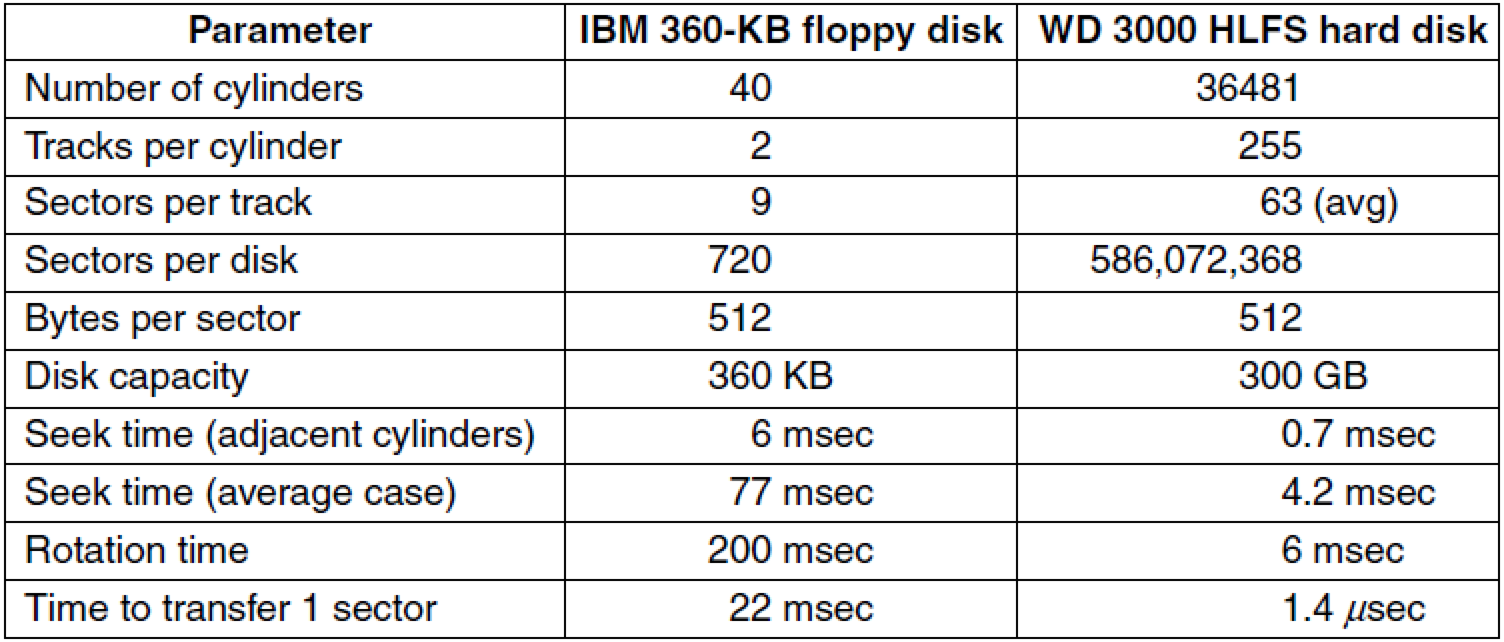
Disk geometry



A is track, B is sector, C is geometrical sector, D is cluster



Magnetic Disks (1)



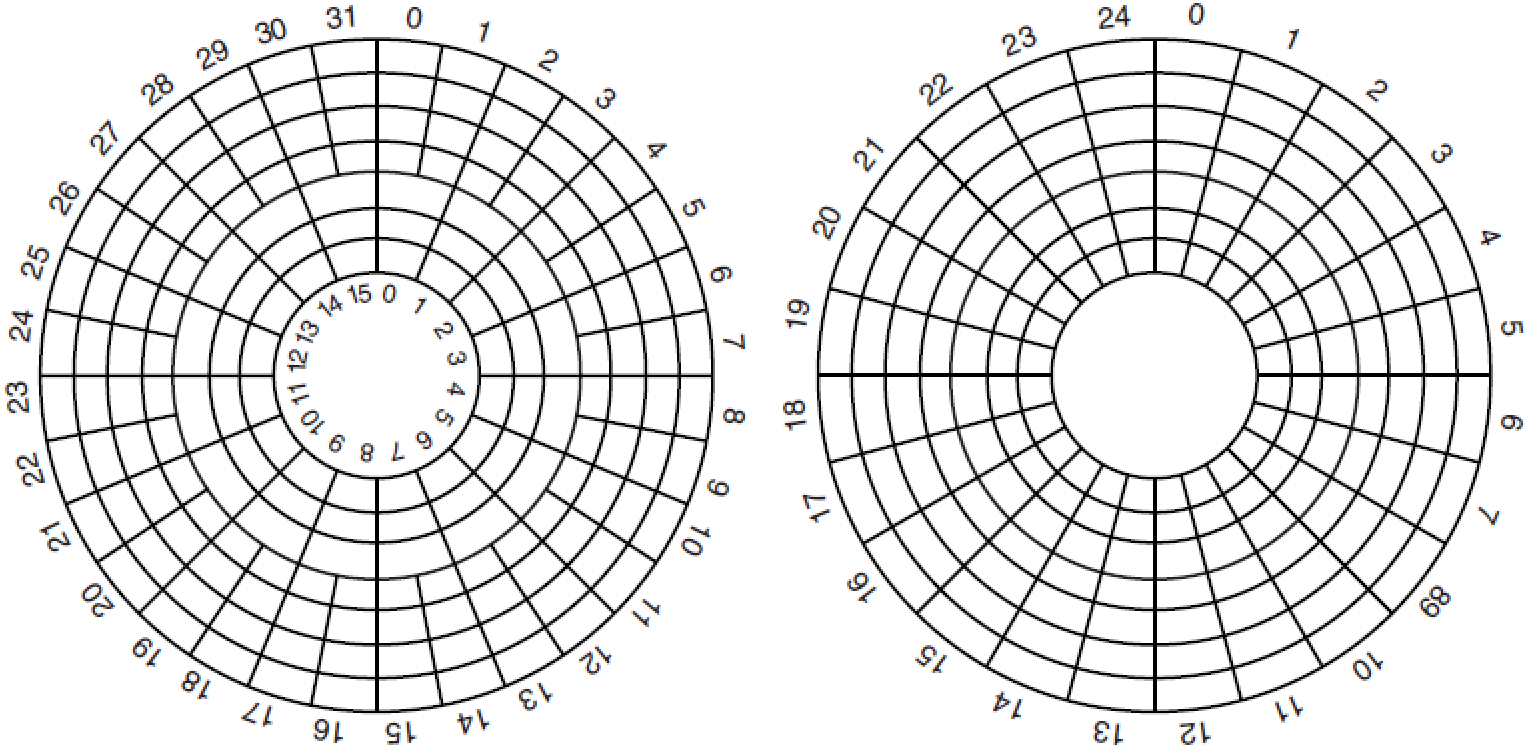
Parameter	IBM 360-KB floppy disk	WD 3000 HLFS hard disk
Number of cylinders	40	36481
Tracks per cylinder	2	255
Sectors per track	9	63 (avg)
Sectors per disk	720	586,072,368
Bytes per sector	512	512
Disk capacity	360 KB	300 GB
Seek time (adjacent cylinders)	6 msec	0.7 msec
Seek time (average case)	77 msec	4.2 msec
Rotation time	200 msec	6 msec
Time to transfer 1 sector	22 msec	1.4 μsec

Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD 3000 HLFS (‘‘Velociraptor’’) hard disk.

Disks-more stuff

- Some disks have microcontrollers which do bad block re-mapping, track caching, etc.
- Some are capable of doing more then one seek at a time, i.e. they can read on one disk while writing on another
- Real disk geometry is different from geometry used by driver => controller has to re-map request for (cylinder, head, sector) onto actual disk
- Disks are divided into zones, with fewer tracks on the inside, gradually progressing to more on the outside

Magnetic Disks (2)



(a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

Redundant Array of Inexpensive Disks (RAID)

- Parallel I/O to improve performance and reliability
- RAID vs SLED (Single Large Expensive Disk)
- Bunch of disks appear like a single disk to the OS
- SCSI disks often used - cheap, 7 disks per controller
- SCSI is a set of standards to connect CPU to peripherals
- Different RAID architectures - level 0 through level 7

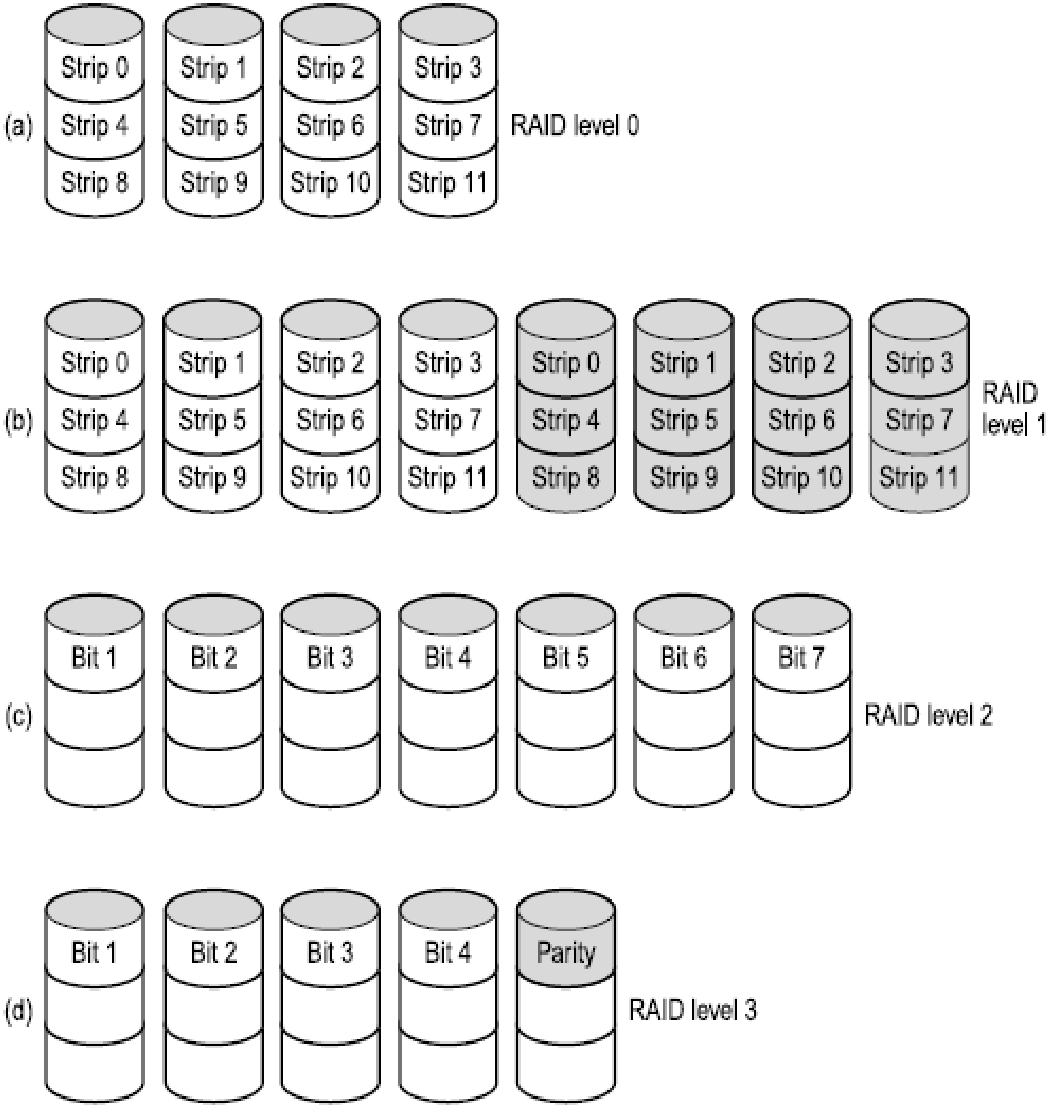
Raid Levels (0,1,2)

- Raid level 0 uses strips of *k* sectors per strip.
  - Consecutive strips are on different disks
  - Write/read on consecutive strips in parallel
  - Good for big enough requests
- Raid level 1 duplicates the disks
  - Writes are done twice, reads can use either disk
  - Improves reliability
- Level 2 works with individual words, spreading word + ECC over disks.
  - Need to synchronize arms to get parallelism

Raid Levels (4,5)

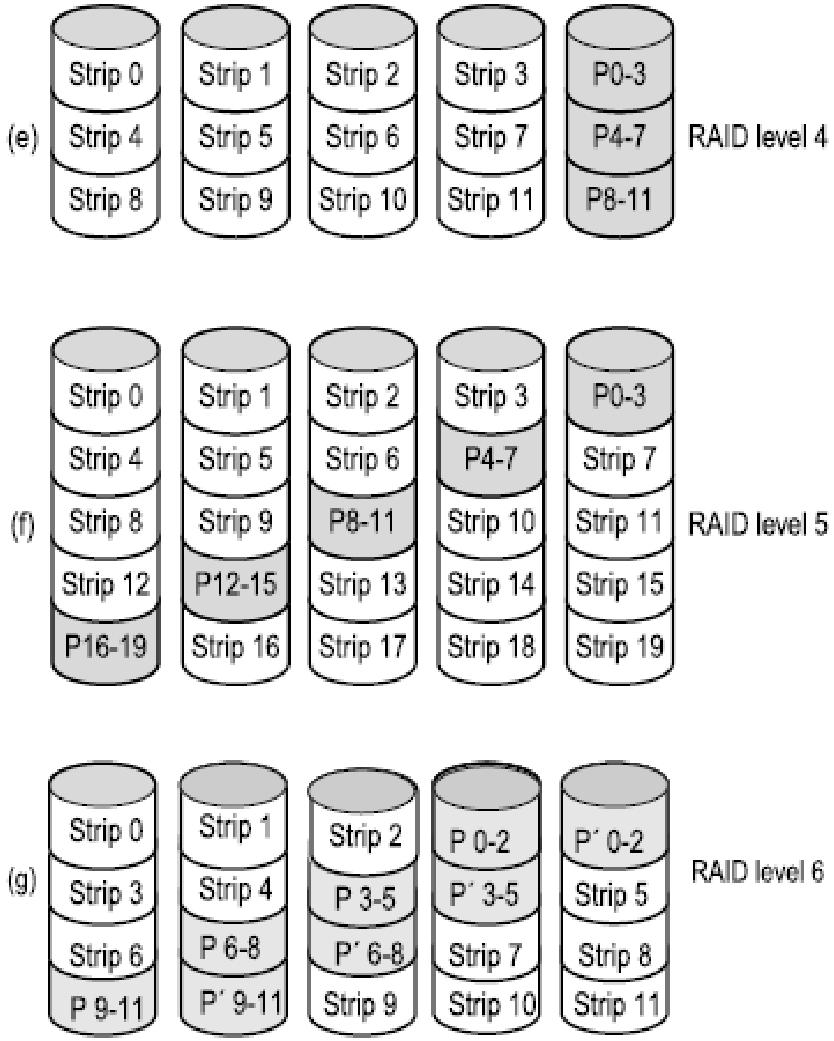
- Raid level 3 works like level 2, except all parity bits go on a single drive
- Raid 4,5 work with strips. Parity bits for strips go on separate drive (level 4) or several drives (level 5)

RAID (1)



RAID levels 0 through 3. Backup and parity drives are shown shaded.

RAID (2)

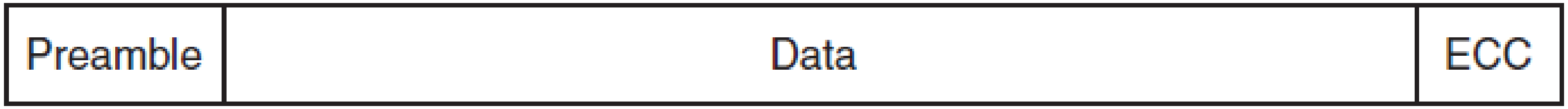


RAID levels 4 through 6. Backup and parity drives are shown shaded.

Hard Disk Formatting

- Low level format
  - software lays down tracks and sectors on empty disk
- High level format
  - partitions

Disk Formatting (1)

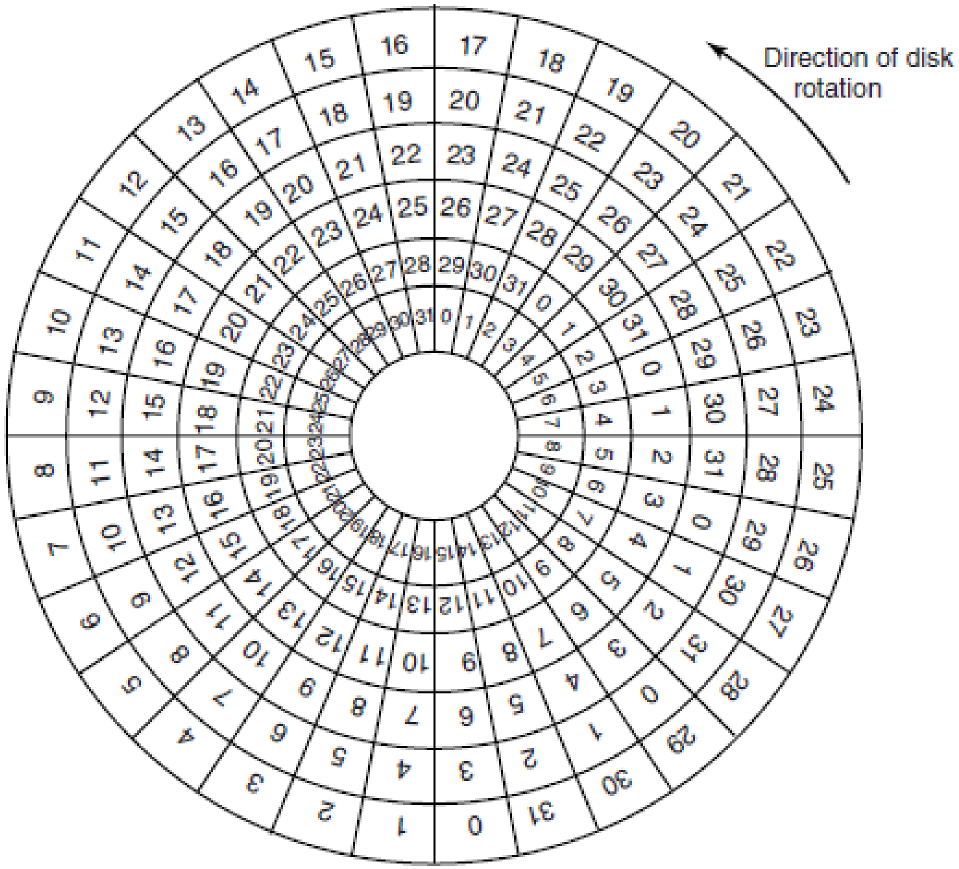


Preamble | Data | ECC

A disk sector.

- 512 bit sectors standard
- Preamble contains address of sector, cylinder number
- ECC for recovery from errors

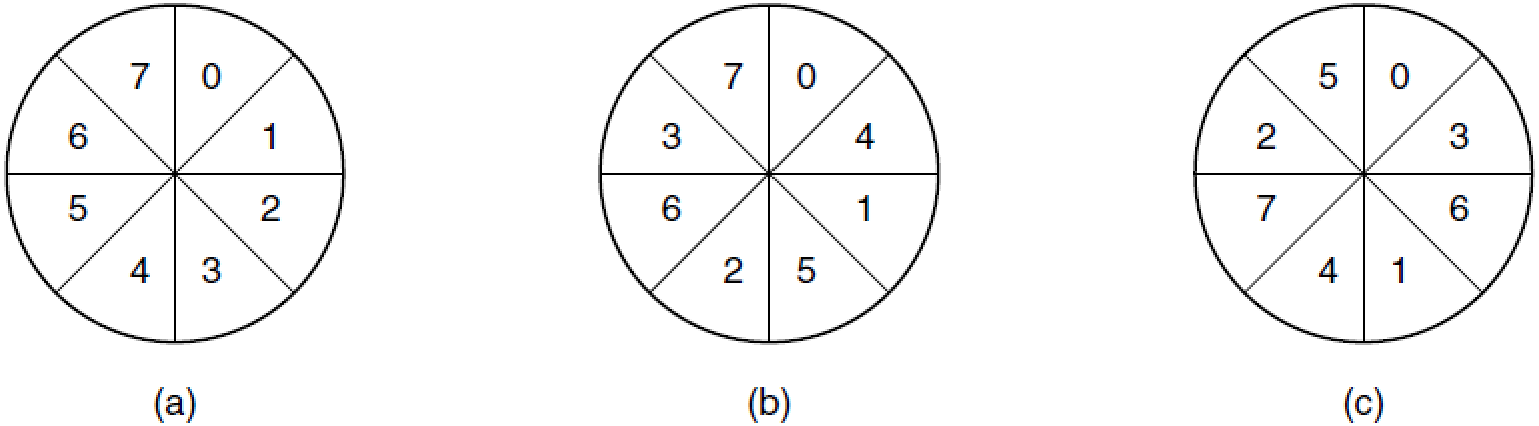
Disk Formatting (2)



- Offset sector from one track to next one in order to get consecutive sectors

An illustration of cylinder skew.

Disk Formatting (3)



(a) No interleaving. (b) Single interleaving.
(c) Double interleaving.

- Copying to a buffer takes time; could wait a disk rotation before head reads next sector. So interleave sectors to avoid this (b,c)

High level format

- Partitions
  - for more then one OS on same disk
- Pentium
  - sector 0 has master boot record with partition table and code for boot block
- Pentium has 4 partitions
  - can have both Windows and Unix
- In order to be able to boot, one sector has to be marked as active

High level format for each partition

- master boot record in sector 0
- boot block program
- free storage admin (bitmap or free list)
- root directory
- empty file system
- indicates which file system is in the partition (in the partition table)

Power-up sequence

- BIOS reads in master boot record
- Boot program checks which partition is active
- Reads in boot sector from active partition
- Boot sector loads bigger boot program which looks for the OS kernel in the file system
- OS kernel is loaded and executed

Disk Arm Scheduling Algorithms (1)

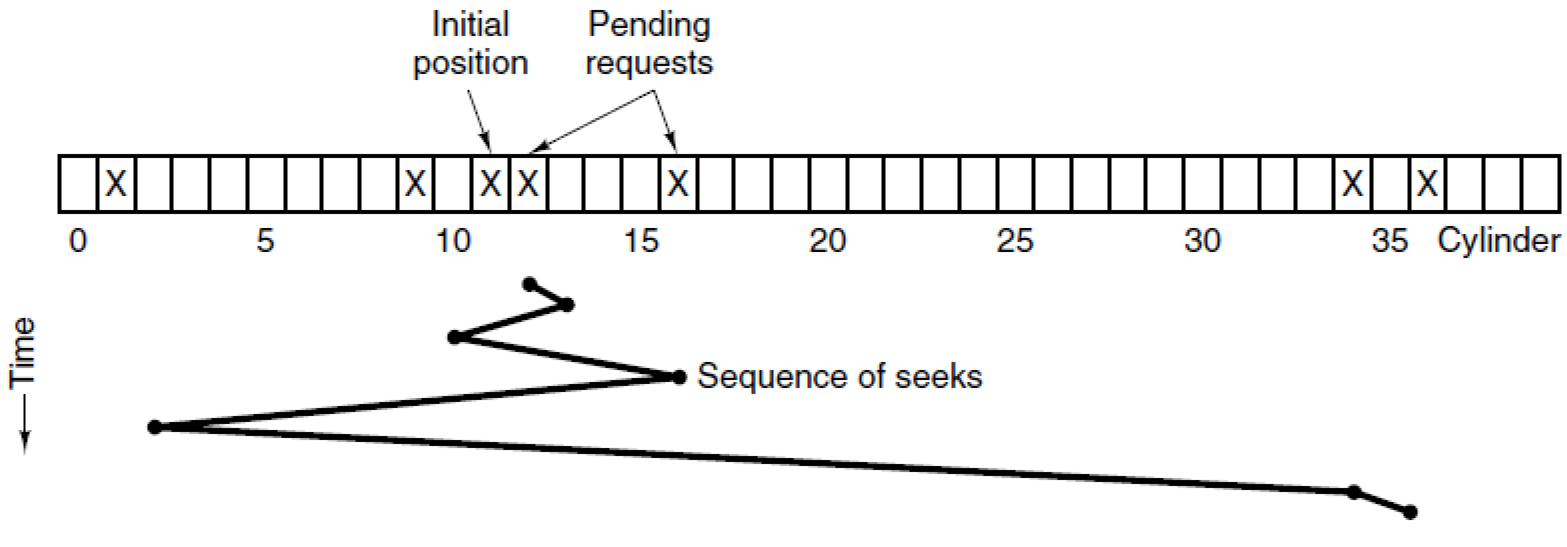
Factors of a disk block read/write:

1. Seek time (the time to move the arm to the proper cylinder).
2. Rotational delay (how long for the proper sector to come under the head).
3. Actual data transfer time.

Disk Arm Scheduling Algorithms

- Driver keeps a list of requests
  - cylinder number, time of request, etc.
- Try to optimize the seek time
- FCFS is easy to implement, but optimizes nothing

Disk Arm Scheduling Algorithms (2)



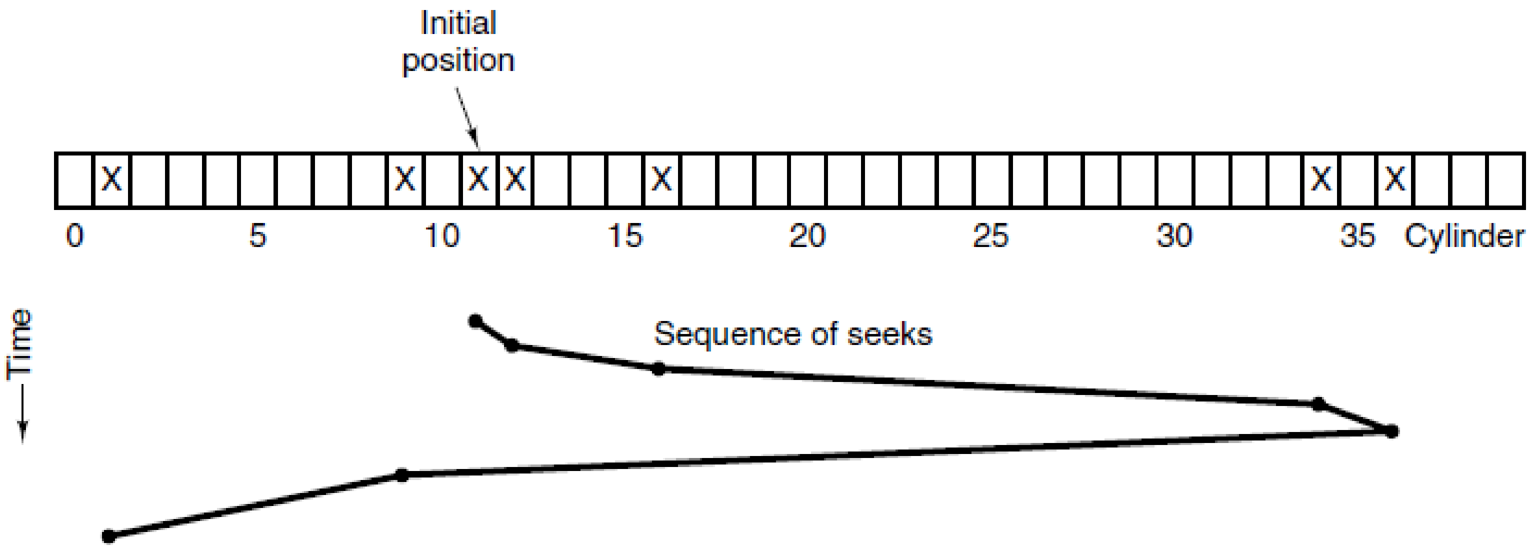
Shortest Seek First (SSF) disk scheduling algorithm.

- While head is on cylinder 11, requests for 1,36,16,34,9,12 come in
- FCFS would result in 111 cylinders
- SSF would require 1,3,7,15,33,2 movements for a total of 61 cylinders

Elevator algorithm

- It is a greedy algorithm
  - -the head could get stuck in one part of the disk if the usage was heavy
- Elevator
  - keep going in one direction until there are no requests in that direction, then reverse direction
- Real elevators sometimes use this algorithm
- Variation on a theme
  - first go one way, then go the other

Disk Arm Scheduling Algorithms (3)



The elevator algorithm for scheduling disk requests.

- Uses 60 cylinders, a bit better

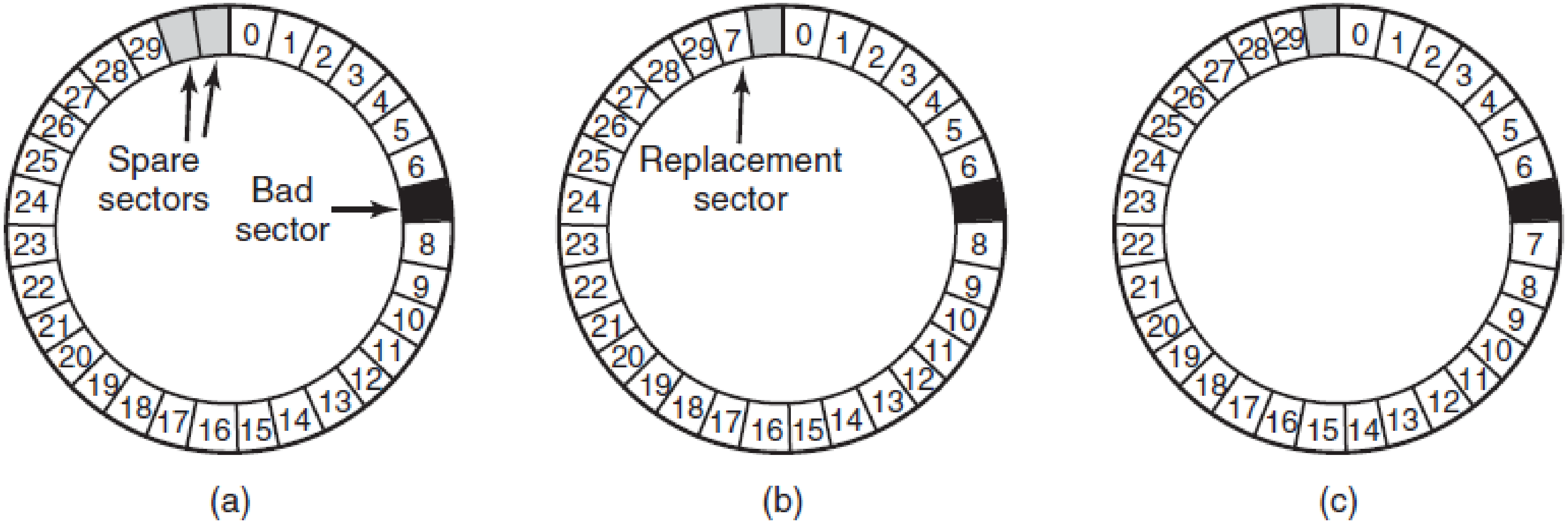
Disk Controller Cache

- Disk controllers have their own cache
- Cache is separate from the OS cache
- OS caches blocks independently of where they are located on the disk
- Controller caches blocks which were easy to read but which were not necessarily requested

Bad Sectors-the controller approach

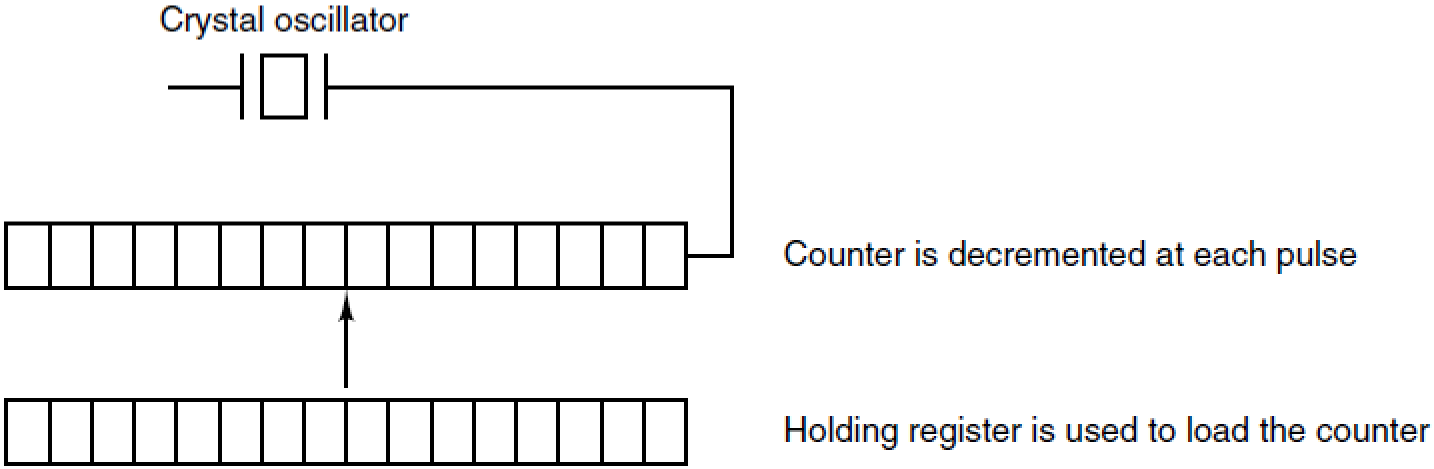
- Manufacturing defect - that which was written does not correspond to that which is read (back)
- Controller or OS deals with bad sectors
- If controller deals with them the factory provides a list of bad blocks and controller remaps good spares in place of bad blocks
- Substitution can be done when the disk is in use-controller “notices” that block is bad and substitutes

Error Handling



(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.

Clock Hardware


Crystal oscillator
Counter is decremented at each pulse
Holding register is used to load the counter

A programmable clock.

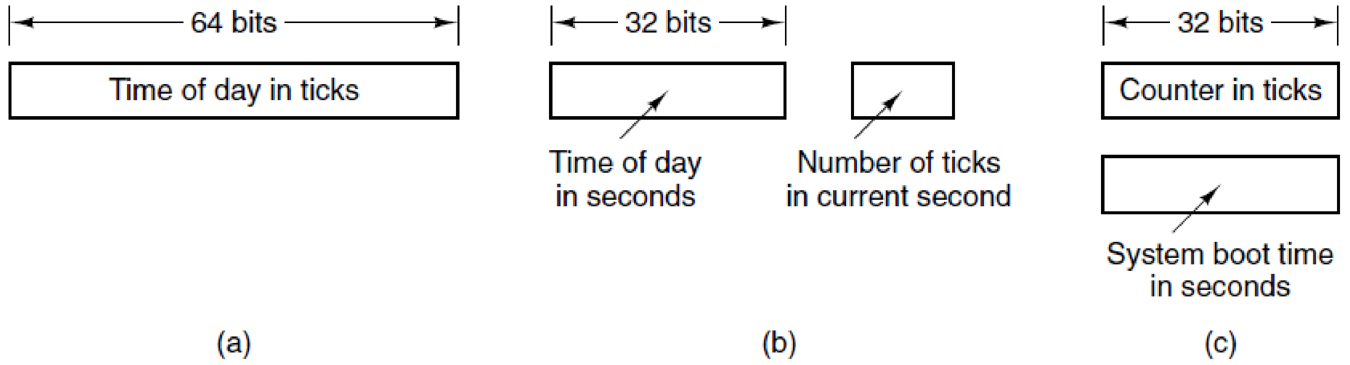
Clock Software (1)

Typical duties of a clock driver:

1. Maintaining the time of day.
2. Preventing processes from running longer than allowed.
3. Accounting for CPU usage.
4. Handling alarm system call from user processes.
5. Providing watchdog timers for parts of system itself.
6. Profiling, monitoring, statistics gathering.

Clock Software - Maintaining the time of day

- Watch out for is the number of bits in the time-of day counter.
- With a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years.



Three ways to maintain the time of day.

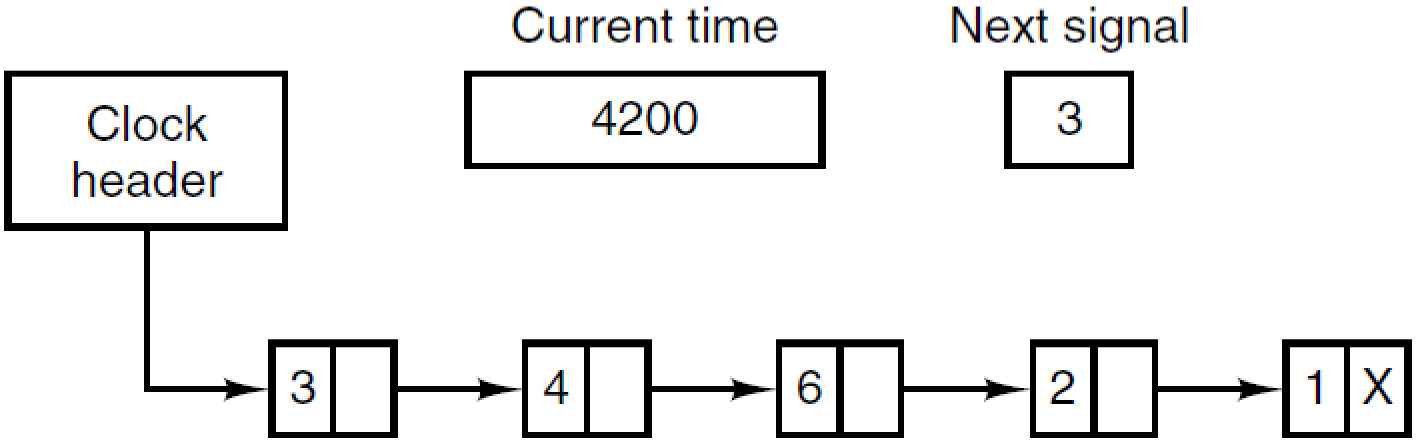
Clock Software - Preventing processes from running longer

- Whenever a process is started, the scheduler initializes a counter to the value of that process’ quantum in clock ticks.
- At every clock interrupt, the clock driver decrements the quantum counter by 1.
- When it gets to zero, the clock driver calls the scheduler to set up another process.

Clock Software – CPU Accounting

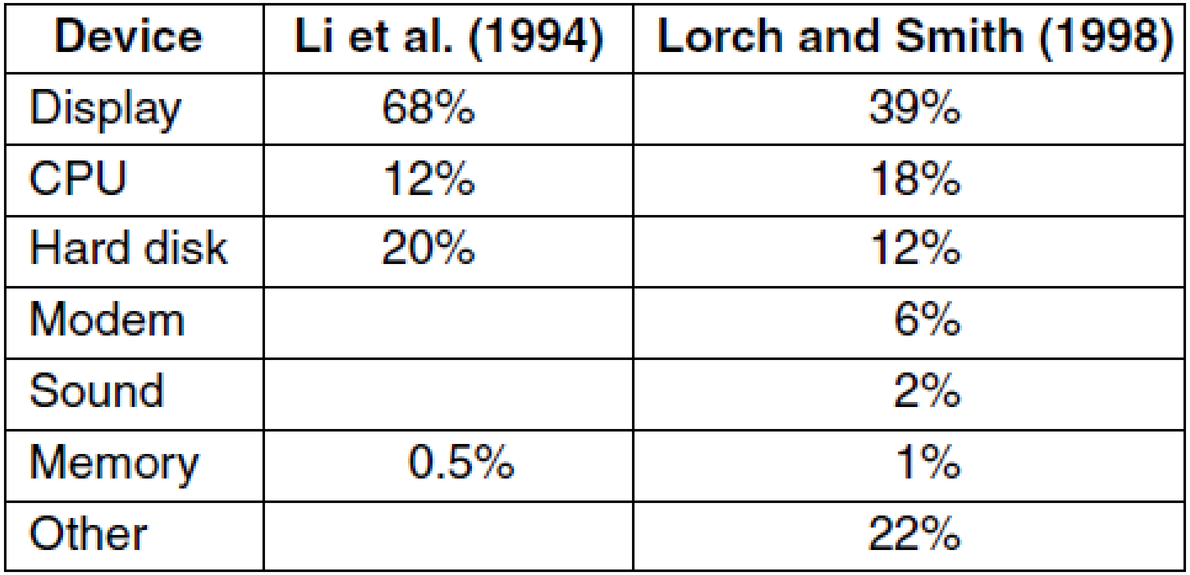
- The most accurate way
  - start a second timer, distinct from the main system timer, whenever a process is started up.
  - when that process is stopped, the timer can be read out to tell how long the process has run
  - to do things right, the second timer should be saved when an interrupt occurs and restored afterwards
- Simpler but less accurate way
  - maintain a pointer to the process table entry for the currently running process in a global variable.
  - At every clock tick, a field in the current process' entry is incremented.
  - In this way, every clock tick is ''charged'' to the process running at the time of the tick.

Clock Software - Handling alarm system call from user processes



Simulating multiple timers with a single clock.

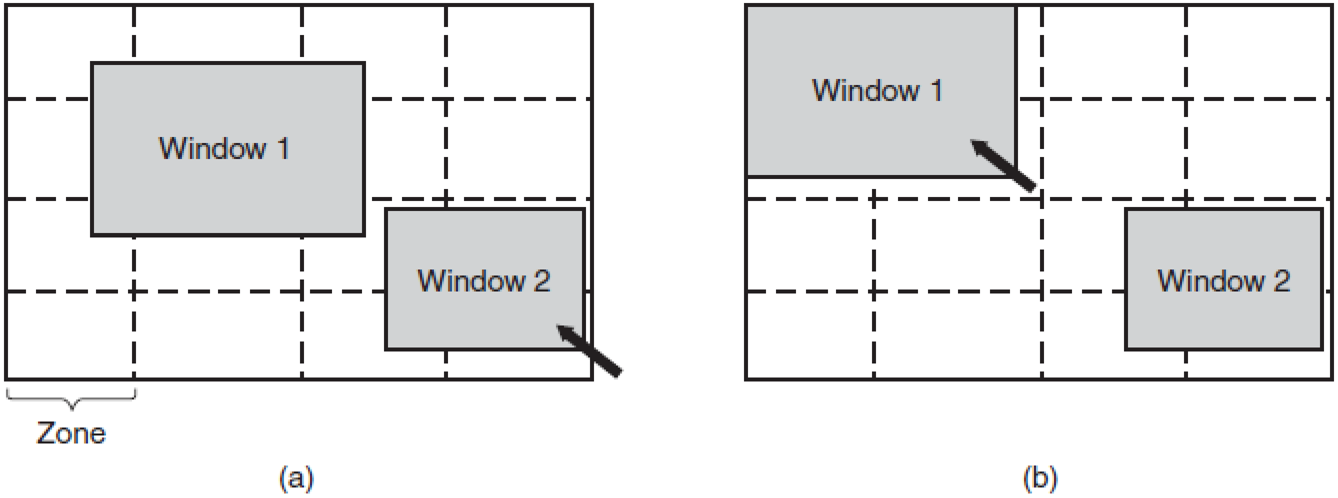
Hardware Issues



Device	Li et al. (1994)	Lorch and Smith (1998)
Display	68%	39%
CPU	12%	18%
Hard disk	20%	12%
Modem		6%
Sound		2%
Memory	0.5%	1%
Other		22%

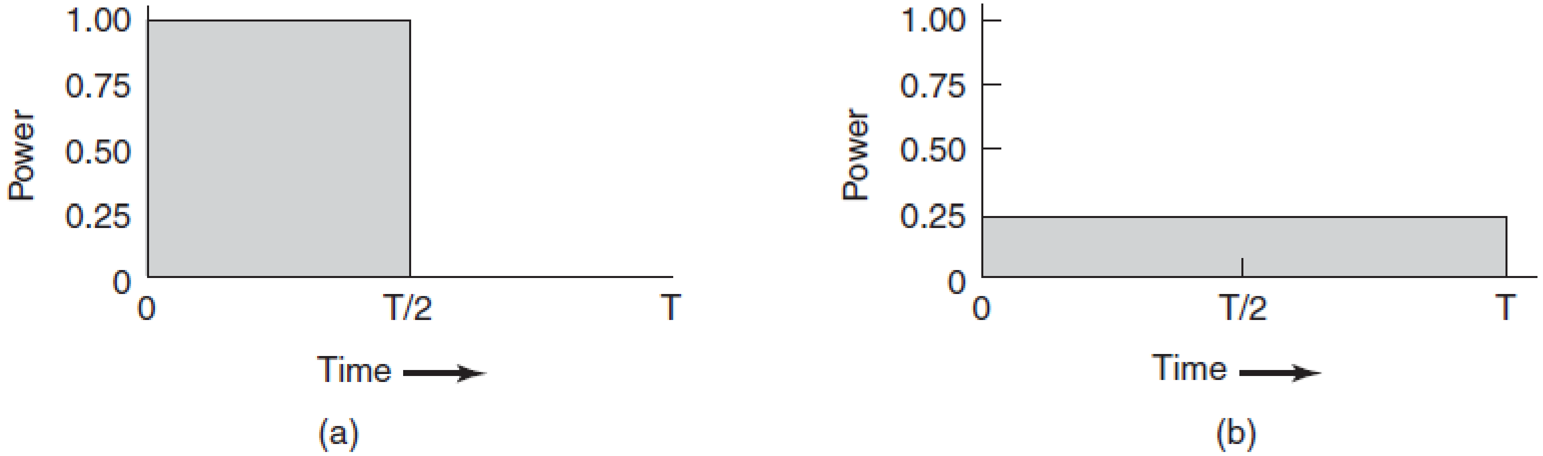
Power consumption of various parts of a notebook computer.

Operating System Issues
The Display



The use of zones for backlighting the display.
(a) When window 2 is selected it is not moved.
(b) When window 1 is selected, it moves to reduce the number of zones illuminated.

Operating System Issues
The CPU



(a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four

End

Chapter 5