Chapter 5
Input/Output

- Principles of I/O software
- I/O software layers
- Disks

OS and I/O

- One of the main functions of OS is to control all the I/O devices
  - Issue commands to the devices
  - Catch interrupts
  - Handle errors
  - Provide an interface between the devices and the rest of OS
    - Make device independent

Principles of I/O Software

- Goals of I/O Software
- I/O Operations
  - Programmed I/O
  - Interrupt-driven I/O
  - I/O using DMA

Goals of I/O Software (1)

- Device independence
  - programs can access any I/O device
  - without specifying device in advance
    - (USB device, hard drive, or CD-ROM)
- Uniform naming
  - name of a file or device a string or an integer
  - not depending on which machine
- Error handling
  - handle as close to the hardware as possible

Goals of I/O Software (2)

- Synchronous vs. asynchronous transfers
  - blocked transfers vs. interrupt-driven
- Buffering
  - data coming off a device cannot be stored in final destination
- Sharable vs. dedicated devices
  - disks are sharable
  - tape drives would not be

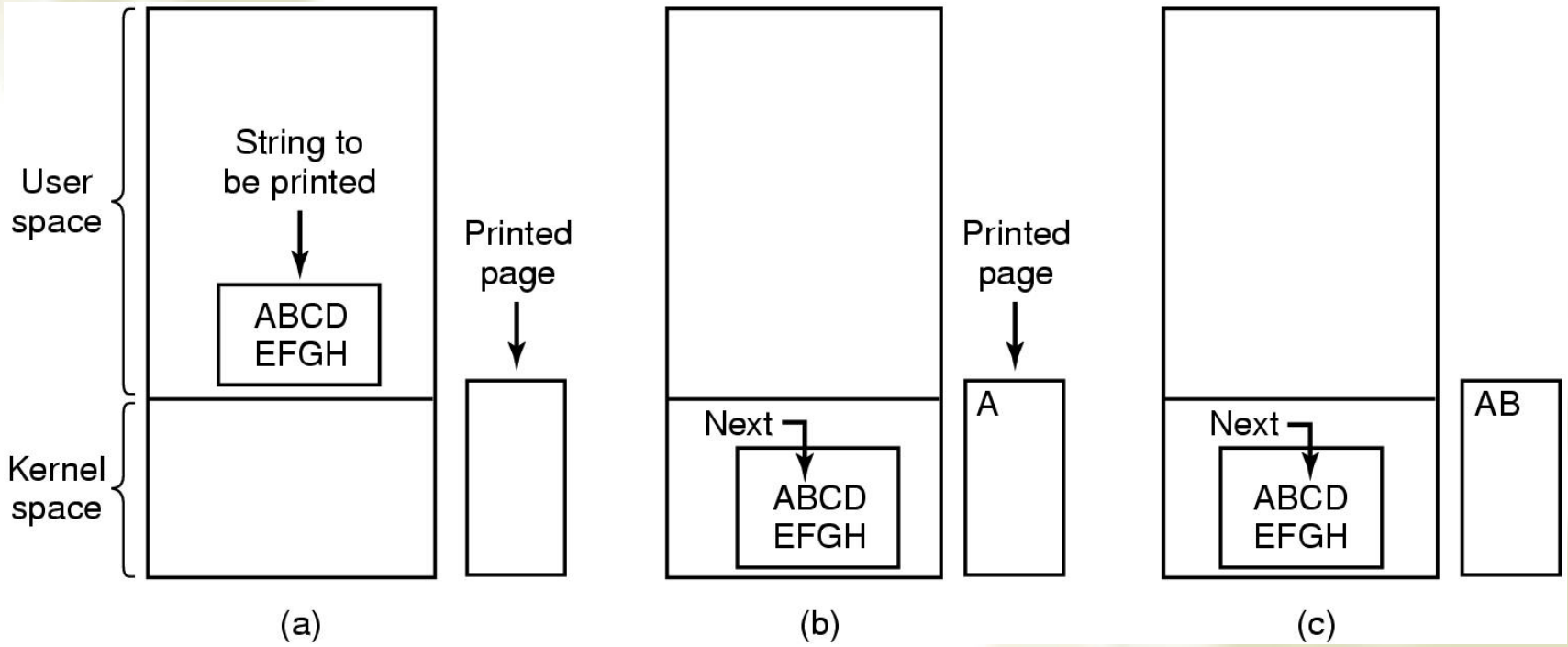
Programmed I/O (1)

- CPU is always busy with I/O until I/O gets completed.
  - Fine for single process systems
    - MS-DOS
    - Embedded systems
  - But not a good approach for multi-programming and time-sharing systems

Programmed I/O (2)

- Assume a program that wants to print a string to a printer.
- Assume string is “ABCDEFGH”, 8 character long string
- Assume printer has 1 byte of data buffer to put the character that needs to be printed.

Programmed I/O (3)



Steps in printing a string

Programmed I/O (4)

```
/*
buffer is user buffer
p is kernel buffer
count is number of bytes to be copied
*/
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

Programmed I/O (5)

- Polling: CPU busy with checking the status register of printer to determine its state
  - Ready
  - Busy
  - Error
- Busy-wait cycle to wait for I/O from device, waste of CPU

Interrupt-Driven I/O (1)

- After copying application buffer content into kernel buffer, and sending 1 character to printer,
  - CPU does not wait until printer is READY again.
  - Instead, the scheduler() is called and some other process is run.
    - The current process is put into blocked state.
  - Printer controller generates an hardware interrupt:
    - When the character is written to the printer and the printer becomes READY again.

Interrupt-Driven I/O (2)

```
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 when print system call is made
  (b) Interrupt service procedure

I/O using DMA (1)

- Disadvantage of Interrupt-Driven I/O is that interrupt occurs on every character.
- DMA controller will feed the characters from kernel buffer to the printer controller.
  - CPU is not bothered during this transfer
- After transfer of whole buffer (string) is completed, then the CPU is interrupted.

I/O using DMA (2)

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

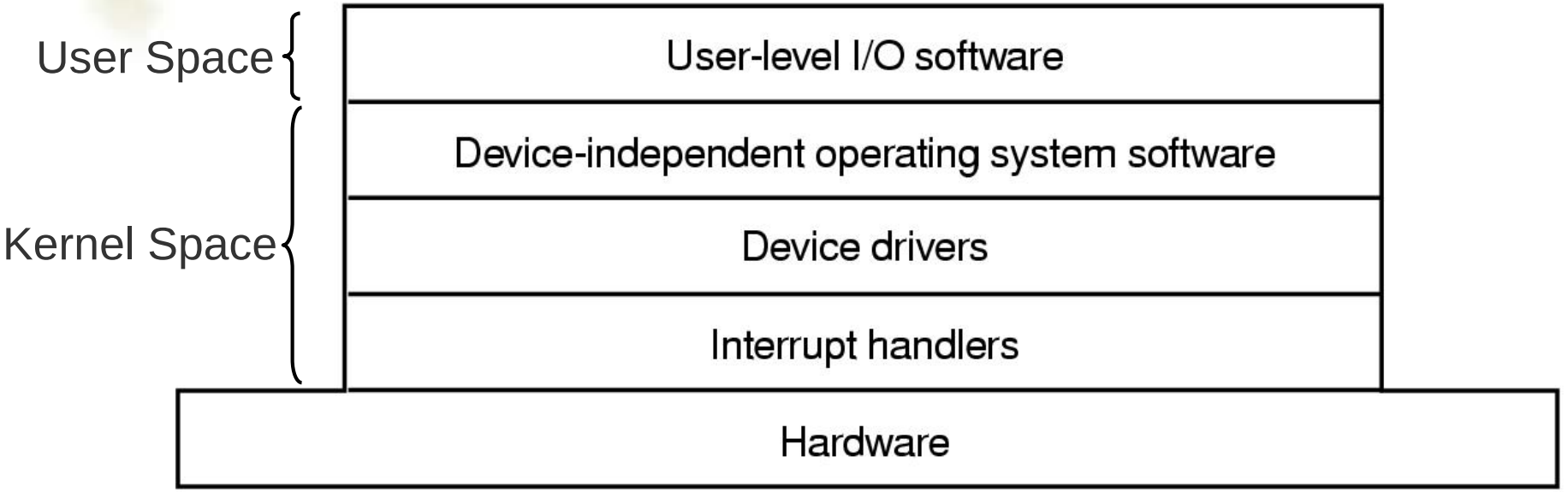
(a)

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

(b)

- Printing a string using DMA
  - code executed when the print system call is made
  - interrupt service procedure

I/O Software Layers

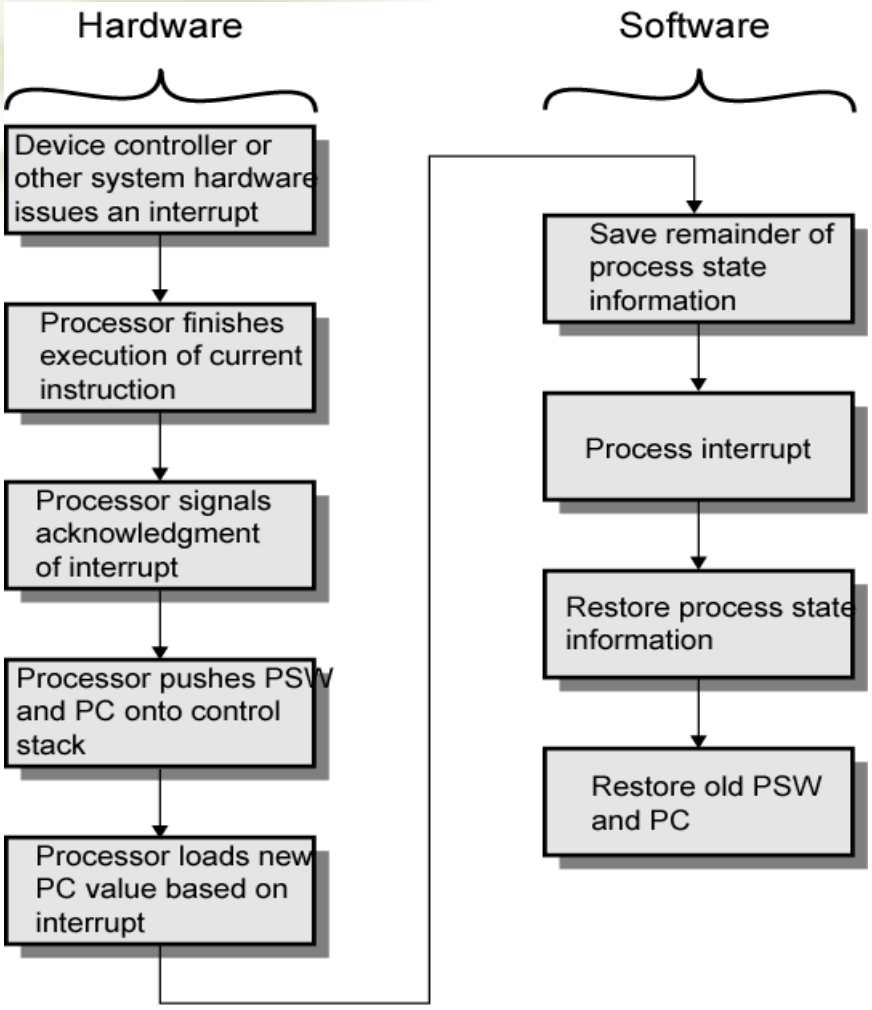


Layers of the I/O Software System

1. Interrupt Handlers (1)

- Interrupt handlers are best hidden
  - So as little of OS know about it as possible
- Best way to hide is to have driver starting an I/O operation block until interrupt notifies of completion
  - A *down* on a semaphore
  - A *wait* on a condition variable
  - A *receive* on a message
  - …
- When the interrupt happens, interrupt handler does its task and then unblocks driver that started it

Interrupt Processing Flowchart



- When an interrupts happen, the CPU saves a small amount of state and jumps to an interrupt-handler routine at a fixed address in memory.
- The interrupt handler routine's location is determined by an interrupt vector.

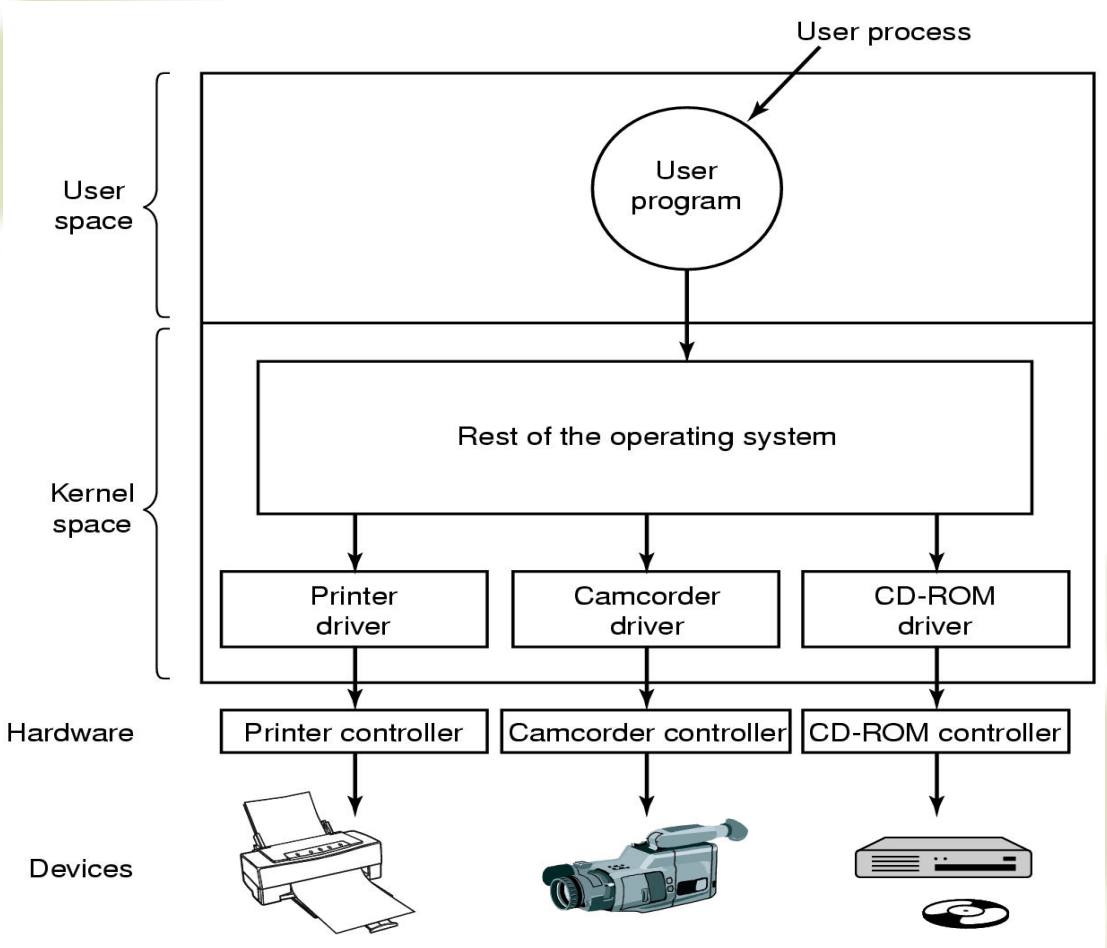
1. Interrupt Handlers (2)

- Steps must be performed in software after the hardware interrupt completed
  1. Save regs not already saved by interrupt hardware
  2. Set up context for interrupt service procedure: TLB, MMU and page table, etc.
  3. Set up stack for interrupt service procedure
  4. Ack interrupt controller, reenable interrupts
  5. Copy registers from where saved to process table
  6. Run service procedure
  7. Choose which process to run next.
  8. Set up MMU context for process to run next
  9. Load new process' registers
  10. Start running the new process

2. Device Drivers (1)

- A device driver is a specific module that manages the interaction between the device controller and the OS.
  - device independent request -> device driver -> device dependent request
- Generally written by device manufacturers
  - Not necessarily by OS writers
- A driver can be written for more than one OS
- Each driver can handle one type of device

2. Device Drivers (2)



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

2. Device Drivers (3)

- Device Driver Functions
  - Accept abstract read and write requests from rest of the OS (device-independent part).
    - Translate from abstract terms to concrete terms.
      - Example: A disk driver translates from a linear block address to a physical head, track, sector, and cylinder number.
  - Initialize the devices if necessary
  - Check if the device is currently in use for some other request.
    - If so, enqueue the request.
  - Issue sequence of commands to control the device
  - Check errors

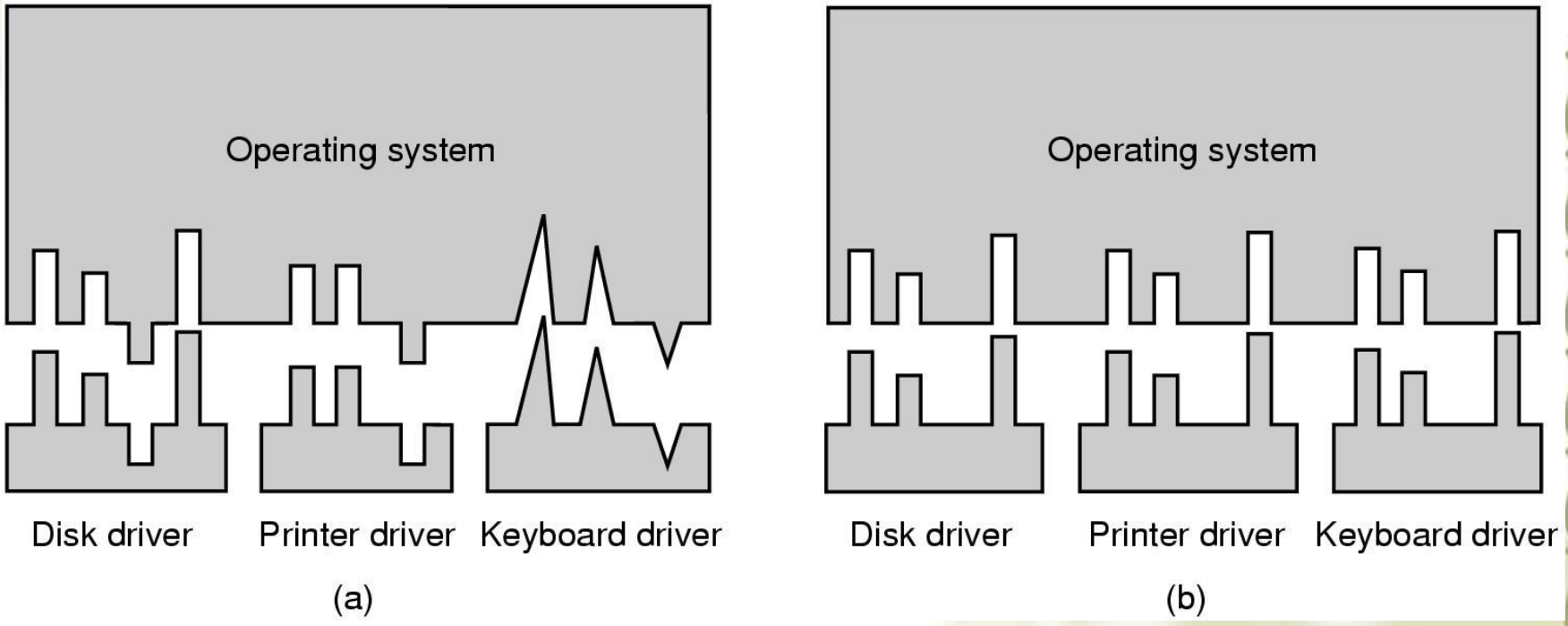
2. Device Drivers (4)

- We need to have a well-defined model of what a driver does and how it integrates with the rest of the OS.
  - This is because, OS implementers and drivers implementers may be different.
  - Most OS define a standard interfaces for device driver. The interface consists of a number of procedures that the rest of OS can call.
- OSs usually classify drivers into two classes
  - Drivers for block devices
  - Drivers for character devices

3. Device-Independent I/O Software

- Boundary between device driver and device-independent software varies between systems and devices.
- Functions of the device-independent I/O software
  - Uniform interfacing for device drivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicate devices
  - Providing a device-independent block size

Uniform Interfacing (1)



(a) Without a standard driver interface

(b) With a standard driver interface

Uniform Interfacing (2)

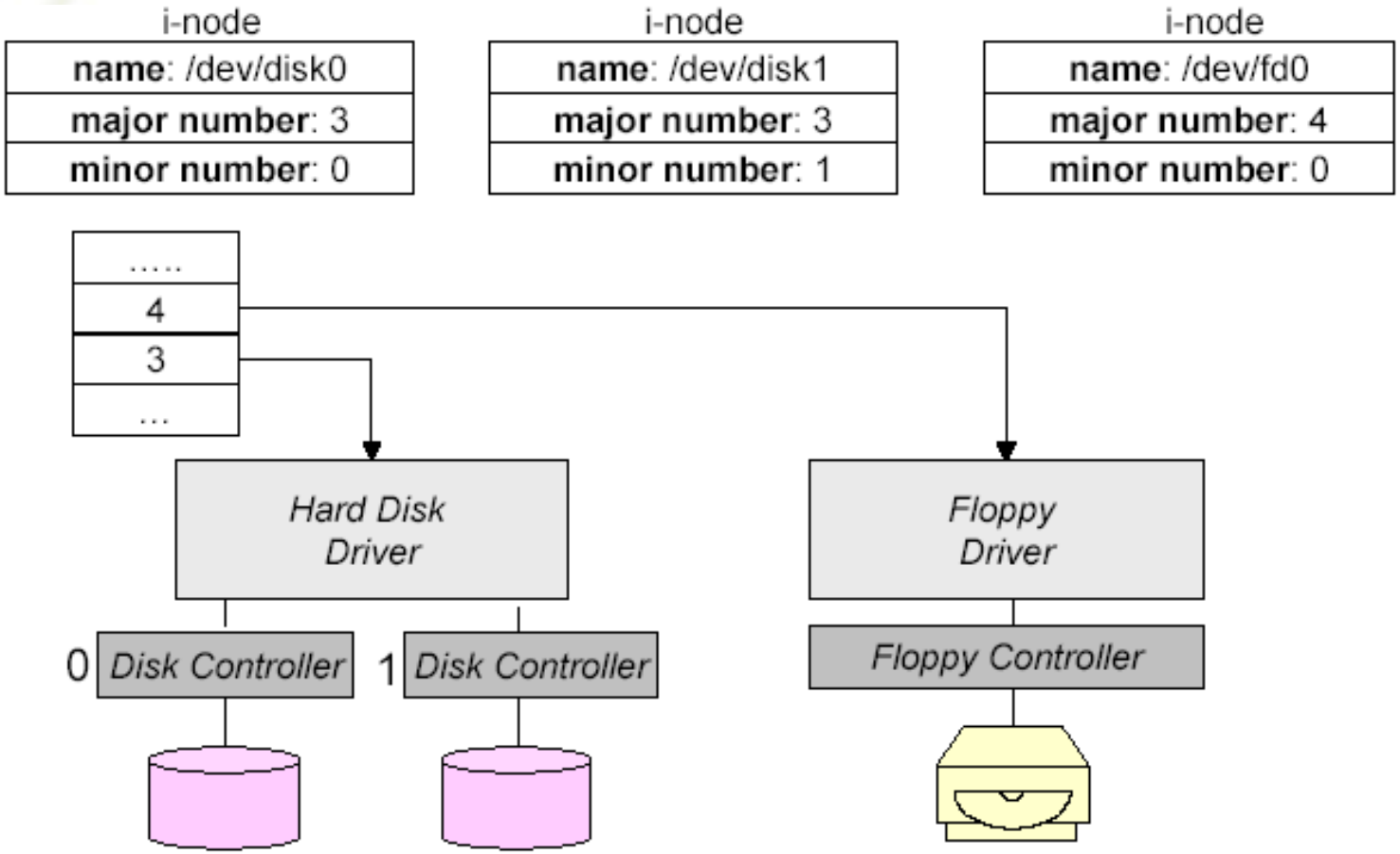
- Drivers need to have uniform interface. The benefits are:
  - The driver implementers know what is expected from them
  - The OS implementers can developed device-independent I/O function on top of a well-defined lower-layer driver interface.
    - They know which functions each driver implements and the pro-types of these function.

Uniform Interfacing (3)

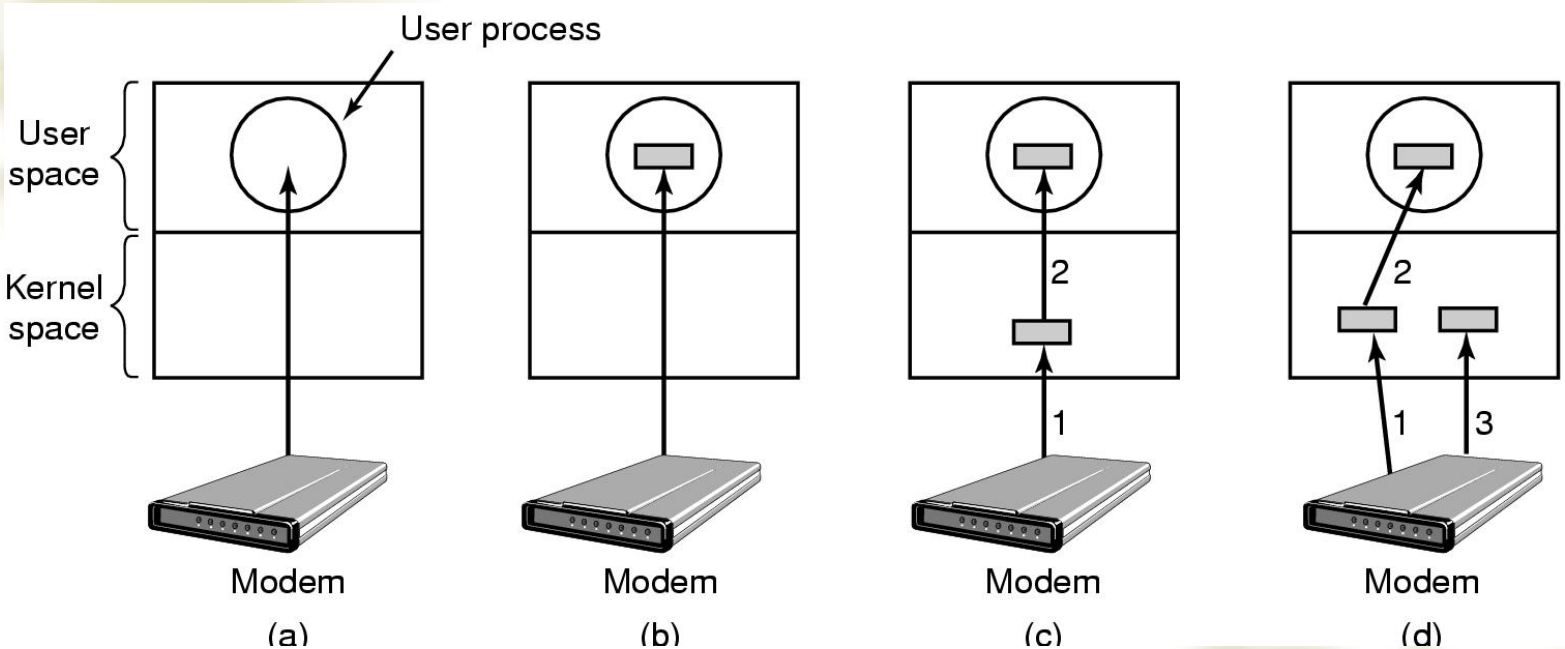
- In Unix, devices are modeled as special files.
  - They are accessed through the use of system calls such as open(), read(), write(), close(), ioctl(), etc.
  - A file name is associated with each device.
- Major device number (stored in i-node) locates the appropriate driver.
  - Minor device number (stored in i-node) is passed as a parameter to the driver in order to specify the unit to be read or written.
- The usual protection rules for files also apply to I/O devices

Uniform Interfacing (4)

- Uniform Interfacing: Mapping symbolic I/O device names to their appropriate drivers

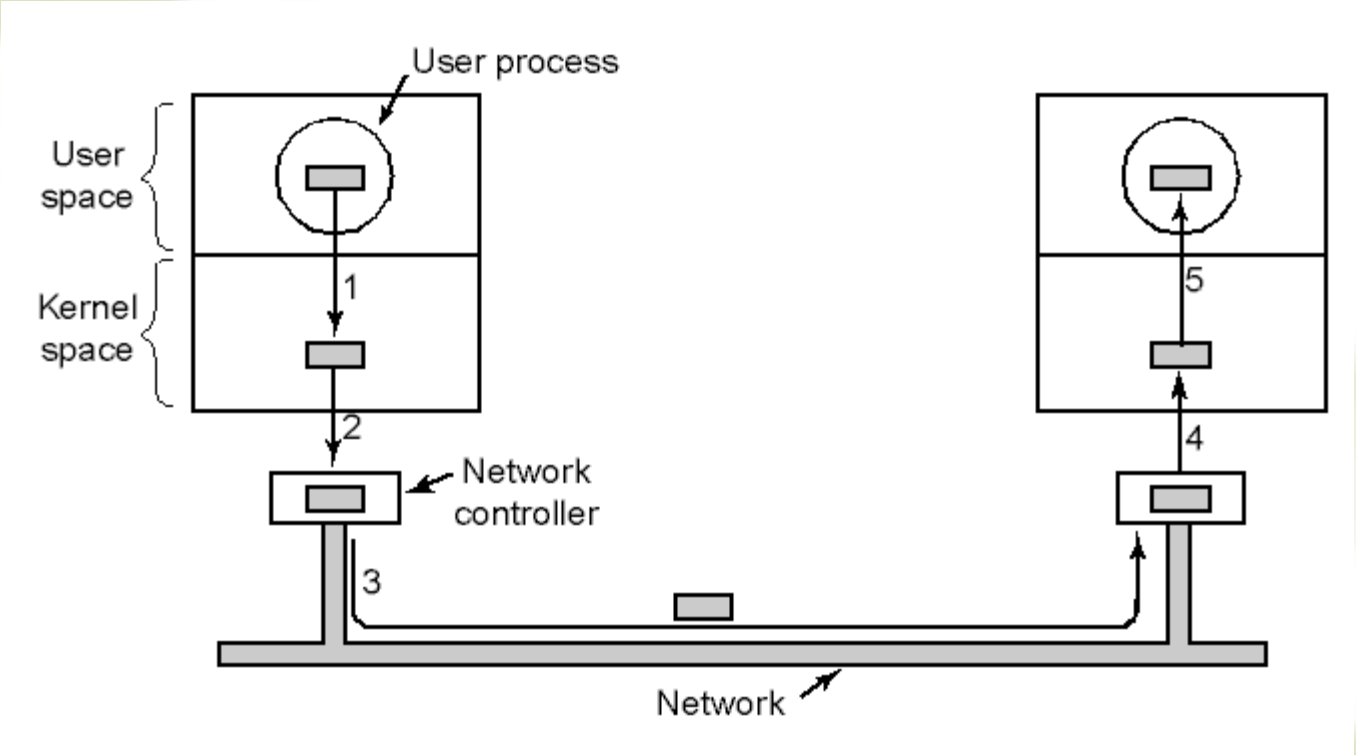


Buffering (1)



(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

Error Reporting

- Some errors are handled by device-controllers
  - Example: Checksum incorrect, re-correct the block by using redundant bits
- Some by device derivers
  - Example: disk block could not be read, re-issue the read operation for block from disk
- Some by the device-independent software layer of OS.
  - Programming errors
    - Example: Attempt to write to a read-only device
  - Actual I/O errors
    - Example: The camcorder is shut-off, therefore we could not read. Return an indication of this error to the calling application.

Allocating Dedicated Devices

- Do not allow concurrent accesses to dedicated devices such as CD-RWs.

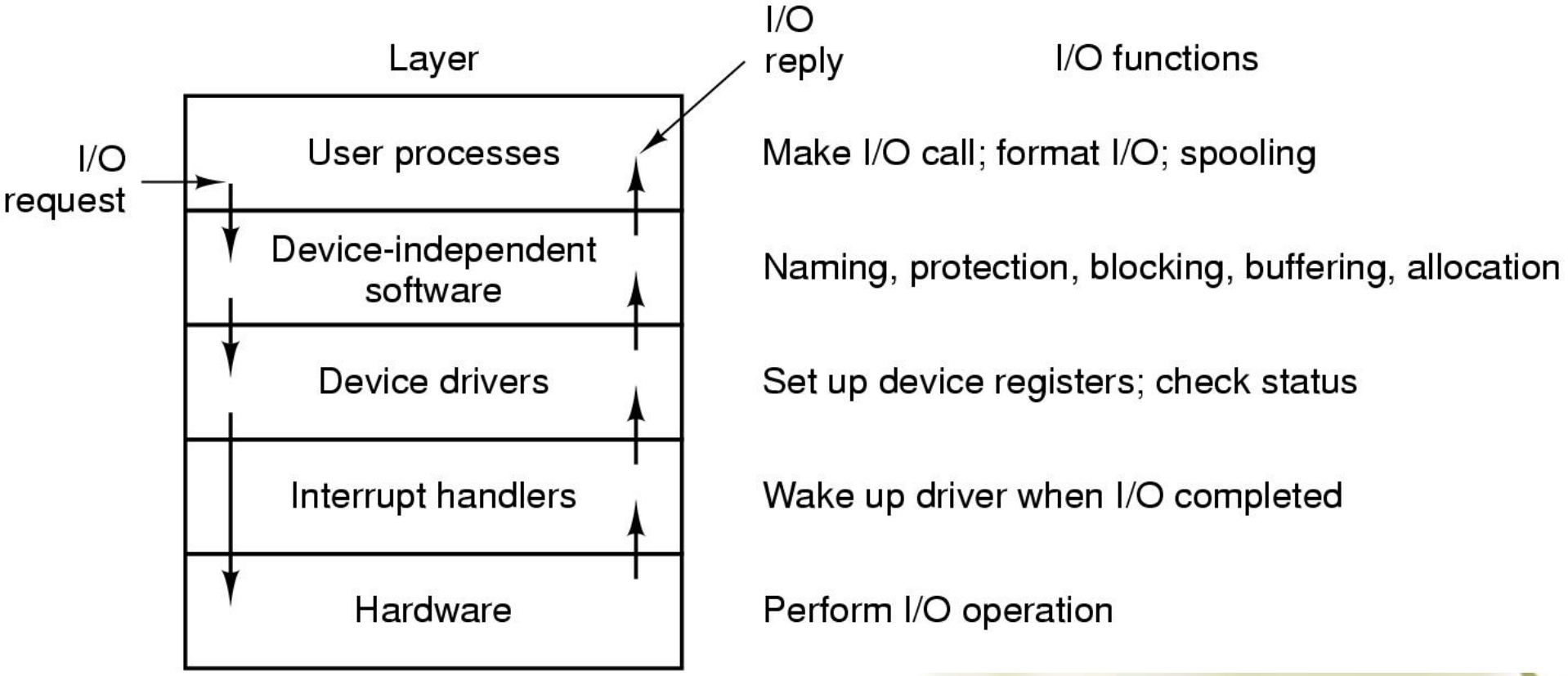
Device-independent Block Size

- There are different kind of disk drives. Each may have a different physical sector size.
- A file system uses a block size while mapping files into logical disk blocks.
- This layer can hide the physical sector size of different disks and can provide a fixed and uniform disk block size for higher layers, like the file system
  - Several sectors can be treated as a single logical block.

4. User-Space I/O Software

- This includes
  - The I/O libraries that provides the implementation of I/O functions which in turn call the respective I/O system calls.
    - These libraries also implemented formatted I/O functions such as printf() and scanf()
  - Some programs that does not directly write to the I/O device, but writes to a spooler.

Summary of I/O Software



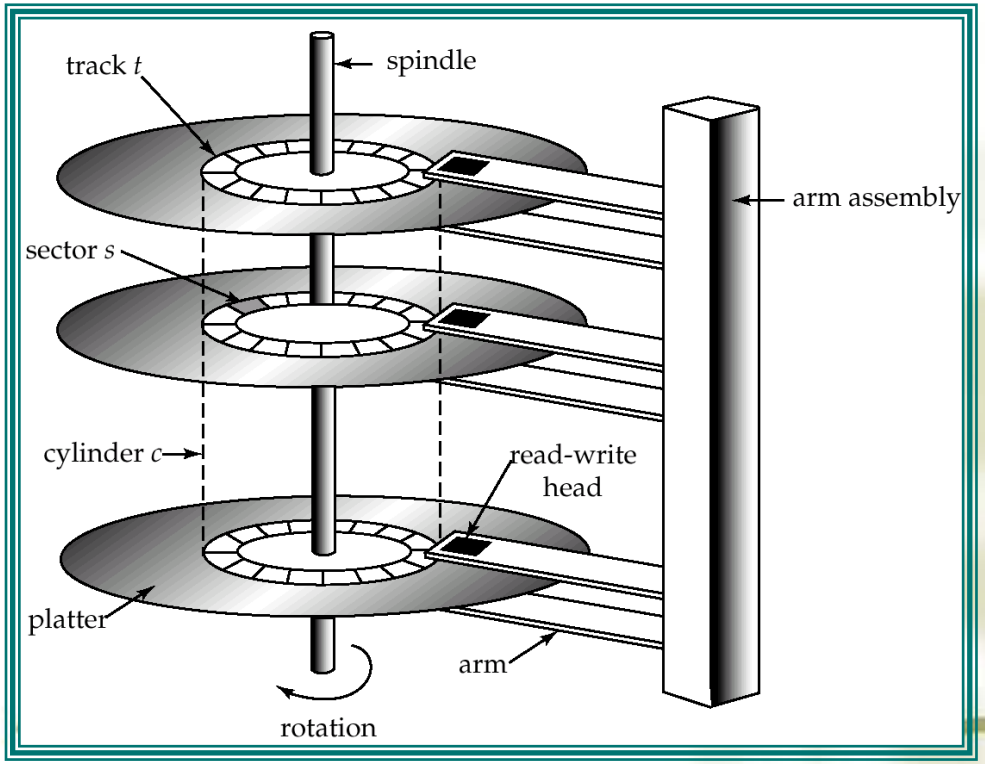
Layers of the I/O system and the main functions of each layer

Disks

- Disks have many types
  - Magnetic disks
    - Hard disk
    - Floppy disk
  - Optical disks
    - CD-ROM
    - CD-Writable
    - CD-Recordable
    - DVDs

Magnetic Disks (1)

- Organized into cylinders
  - Each cylinders contains tracks
    - Each track is divided into a number of sectors
      - The sector size is usually 512 bytes.

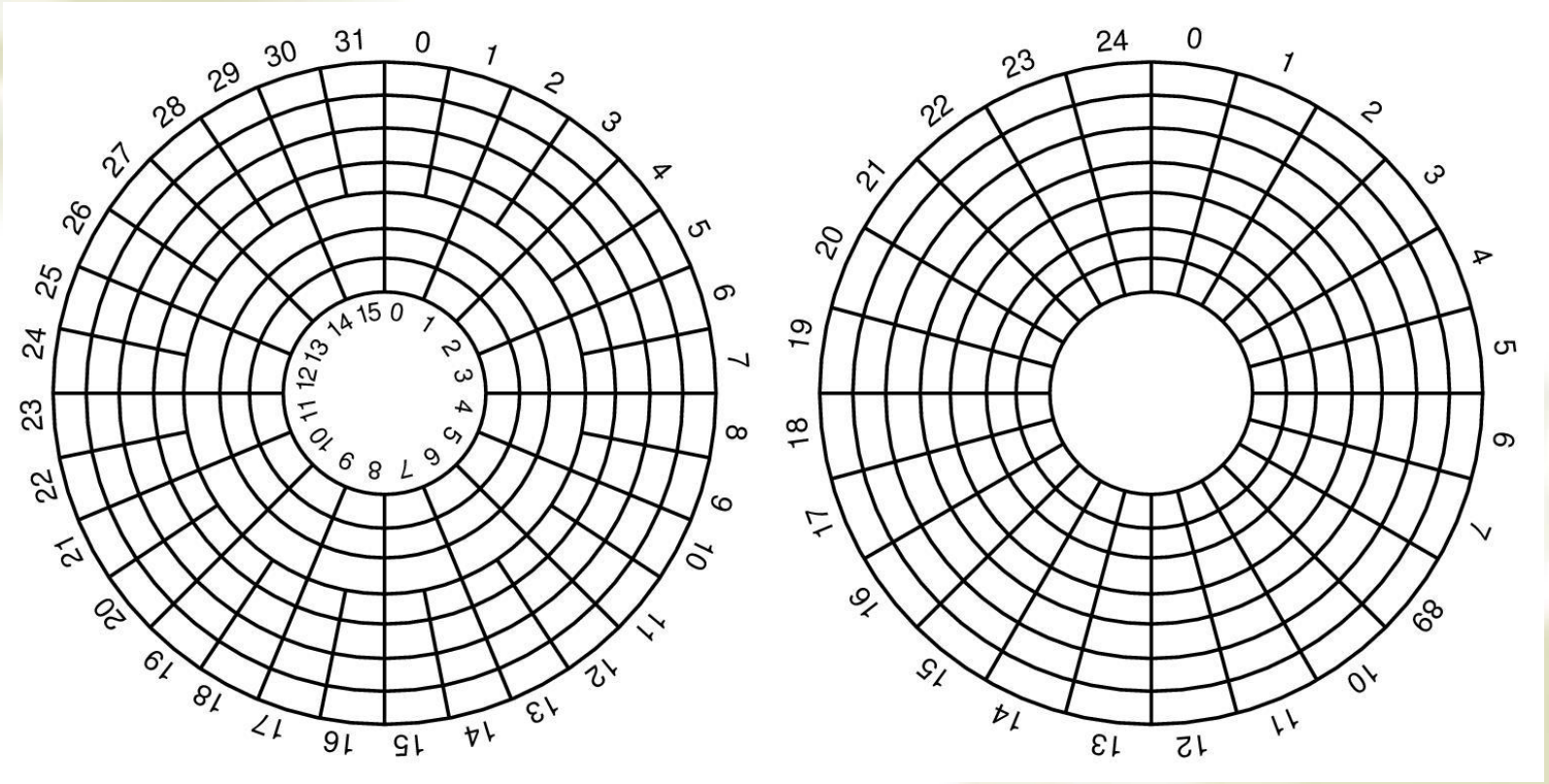


Magnetic Disks (2)

Parameter	IBM 360-KB floppy disk	WD 3000 HLFS hard disk
Number of cylinders	40	36,481
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 floppy disk and a Western Digital WD 30000 HLFS hard disk

Magnetic Disks (3)



- Physical geometry of a disk with two zones
- A possible virtual geometry for this disk

Disk Formatting (1)

- After manufacturing, there is no information on the disk
  - Just empty bits
- Each platter need to have
  - A low-level format
  - A high-level format

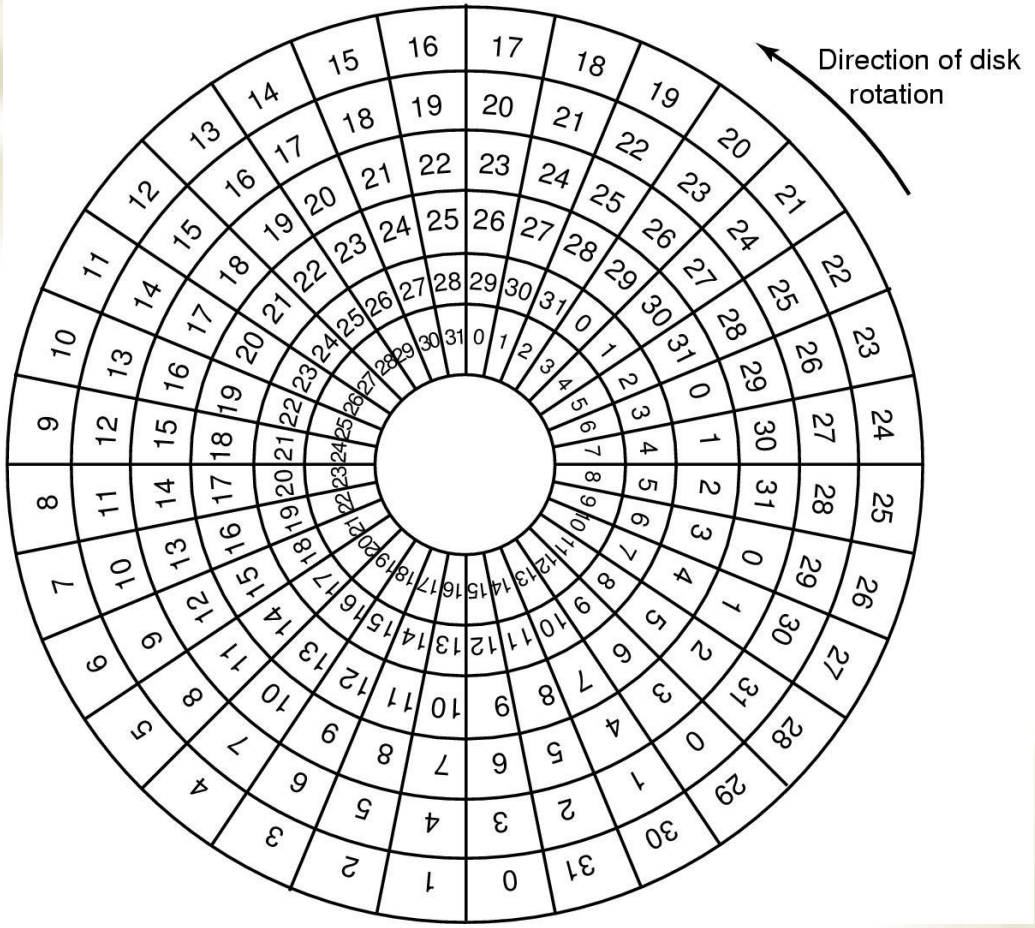
  before disk can be used
- Low-level formatting
  - Dividing a disk into sectors that the disk controller can read and write
  - Low-level formatting is usually done by vendors

Disk Formatting (2)

- Each track should have formatted in the following way:
  - Sectors and inter-sector-gap between them
  - A sector will be have format like the following:
    - Preamble – data (512 bytes) – Checksum

-certain bit pattern
- cylinder number
-sector number

Cylinder Skew



Example

- 10,000rpm disk drive (6ms/round)
- 300 sectors per track (20 μs/sector)
- Seek time: 800 μs
- Cylinder skew: 40secotrs

An illustration of cylinder skew(柱面斜进)

Quiz

- How much cylinder skew is needed for a 7200-RPM disk with a track-to-track seek time of 1 msec? The disk has 200 sectors of 512 bytes each on each track.

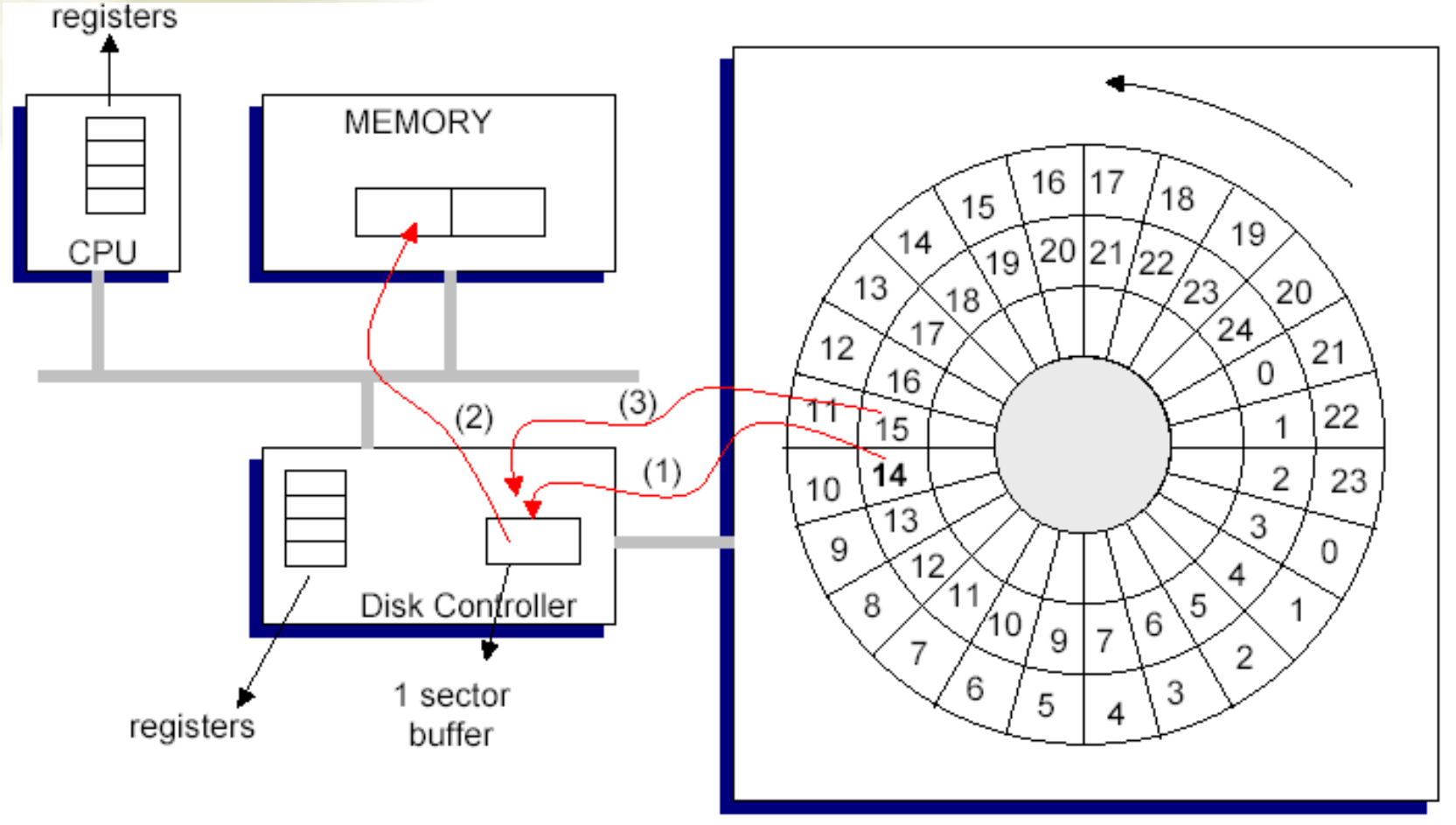
Solution

- The disk rotates at 120 RPS, so 1 rotation takes 1000/120 msec.
- With 200 sectors per rotation, the sector time is 1/200 of this number or 5/120 = 1/24 msec.
- During the 1-msec seek, 24 sectors pass under the head. Thus the cylinder skew should be 24.

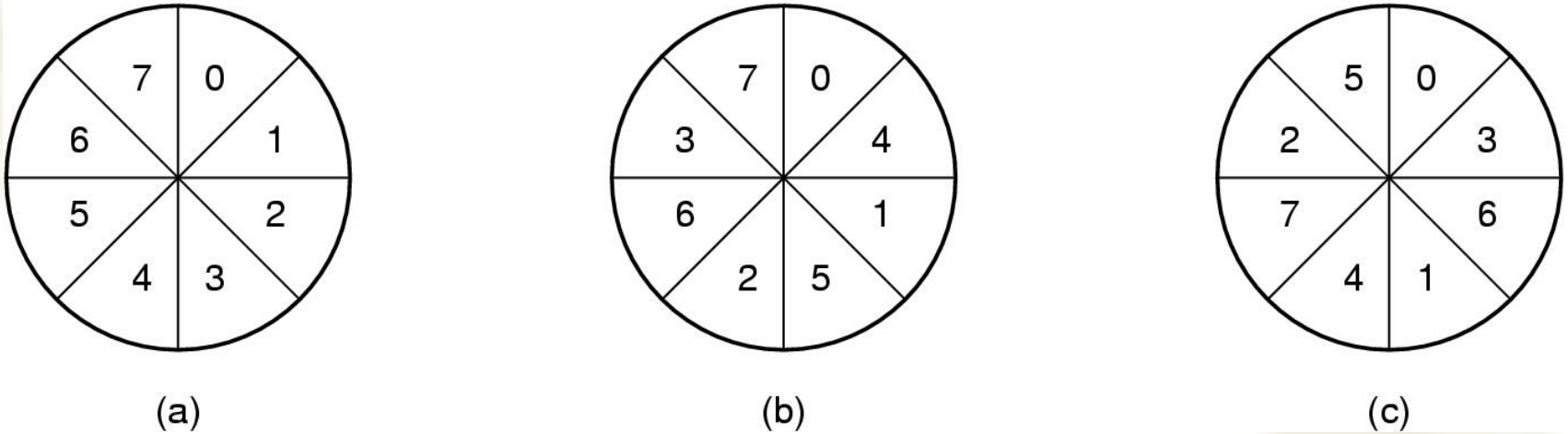
Interleaving (1)

- If we have ONE sector buffer in the disk controller:
  - After we have transferred one sector of data from hard-disk to controller buffer:
    - We will copy the controller buffer to the memory
    - During this time, the disk-head will pass the start of the next sector.
    - Therefore, the next logical sector, should not be the next physical sector in hard-disk.
      - There should be some interleaving.

Interleaving (2)



Interleaving (3)



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

- Modern drives overcome interleaving type issues by simply reading the entire track (or part thereof) into the on-disk controller and caching it.

Discussion

- If a disk has double interleaving, does it also need cylinder skew in order to avoid missing data when making a track-to-track seek?

- Maybe yes and maybe no. Double interleaving is effectively a cylinder skew of two sectors. If the head can make a track-to-track seek in fewer than two sector times, then no additional cylinder skew is needed. If it cannot, then additional cylinder skew is needed to avoid missing a sector after a seek.

Disk Arm Scheduling Algorithms (1)

- Time required to read or write a disk block determined by 3 factors
  - Seek time
    - Move arm to the correct cylinder
  - Rotational delay
    - Wait until correct sector comes under head.
  - Actual data transfer time
- For most systems, the seek time dominates the other two times.

Disk Arm Scheduling Algorithms (2)

- Assume the disk is heavily loaded.
  - There are a lot requests for disk blocks that are arriving to the hard disk driver.
  - The drivers queues the requests.
    - The driver knows for each block request, where in hard disk the block should be stored (the cylinder number).
  - Since moving between cylinders (seek time) is costly, we should try to minimize these seek times.

Disk Arm Scheduling Algorithms (3)

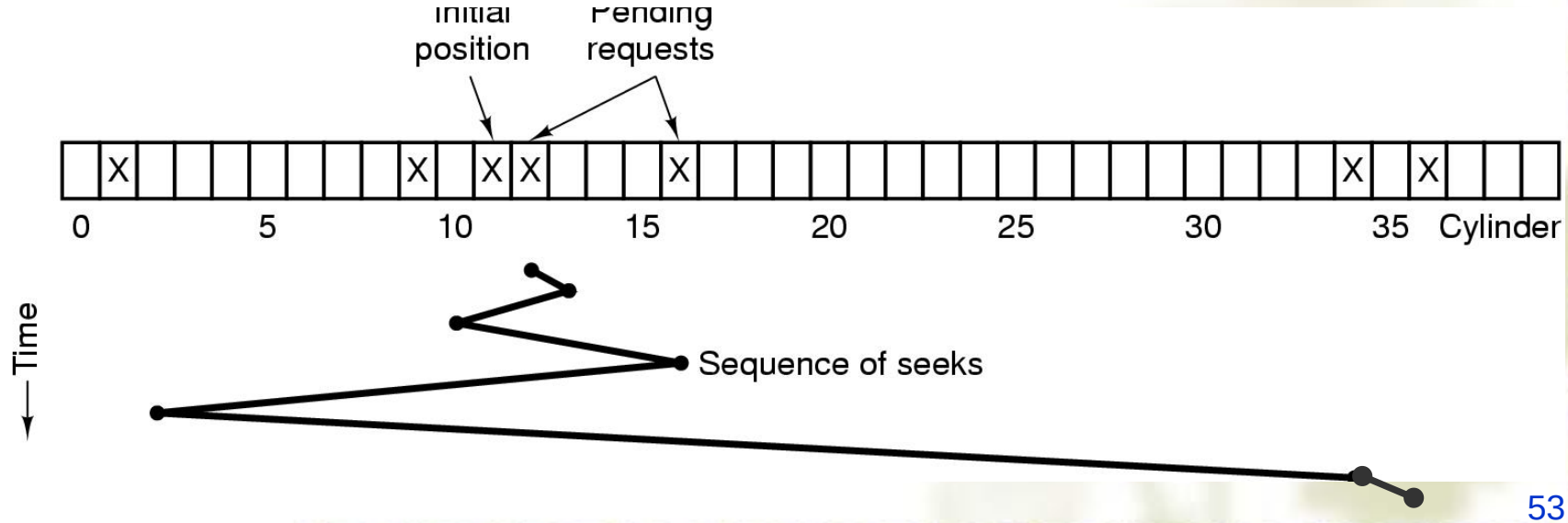
- Algorithms order pending accesses to tracks so that disk arm movement is minimized
  - First Come First Served (FCFS)
  - Shortest Seek First (SSF)
  - Elevator Algorithm

FCFS

- Serve the request in the order they arrive (don't look to the cylinder numbers)
- Example:
  - Assume the current position is cylinder 11.
  - The following requests arrive in the given time order: 1, 36, 16, 34, 9, 12
  - The request will be served in the same order.
  - The head will go to:
    - Cylinder 1 first (10 cylinders motion)
    - Cylinder 36 next (35 cylinders motion)
    - Cylinder 16 next (20 cylinders motion)
    - Cylinder 34 next (18 cylinders motion)
    - Cylinder 9 next (25 cylinders motion)
    - Cylinder 12 next (3 cylinders motion)
    - Total of 111 cylinders are skipped. Cost = 111

Shortest Seek First (SSF)

- Satisfy the request that need shortest seek from the current position.
  - Cylinder request sequence: 11、1、36、16、34、9、12
  - Sequence of seeks: 11、12、9、16、1、34、36
  - Moving the head to these cylinders require: 1, 3, 7, 15, 33, and 2 arm motions, total cost=61

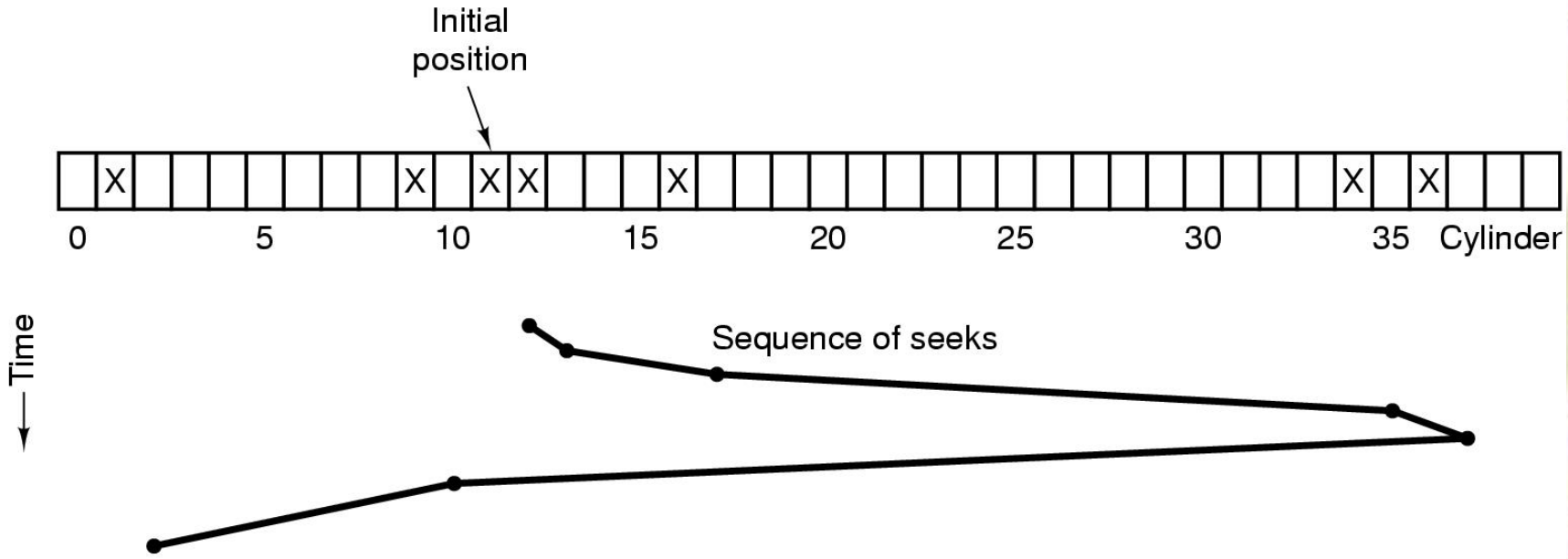


Elevator algorithm (1)

- Elevator algorithm (Also called SCAN algorithm)
  - Move disk arm in one direction (from outer to inner tracks or vice versa), processing next request in that direction, till no more requests in that direction, then reverse direction and repeat
- In the previous example, assume initial position was 11 and initial direction was up (going to higher numbered cylinders).

Elevator algorithm (2)

- Cylinder request sequence: 11、1、36、16、34、9、12
- Sequence of seeks: 11、12、16、34、36、9、1
- Moving the head to these cylinders require: 1, 4, 18, 2, 27, and 8 arm motions, total cost=60



Summary

- Principles of I/O Software
  - Goals of I/O software
  - Programmed I/O
  - Interrupt driven I/O
  - I/O using DMA
- I/O software layers
- Disks
  - Disk formatting
  - Disk arm scheduling algorithm

Homework

- 14、28、31