# Input/Output

## Week 10 - Lecture 2

# Team

**Instructors**

Giancarlo Succi

Joseph Brown

**Teaching Assistants**

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

# Sources

- These slides have been adapted from the original slides of the adopted book:
- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
  Prentice-Hall, Inc.
  and customized for the needs of this course.
- Additional input for the slides are detailed later

# Programmed I/O (1)

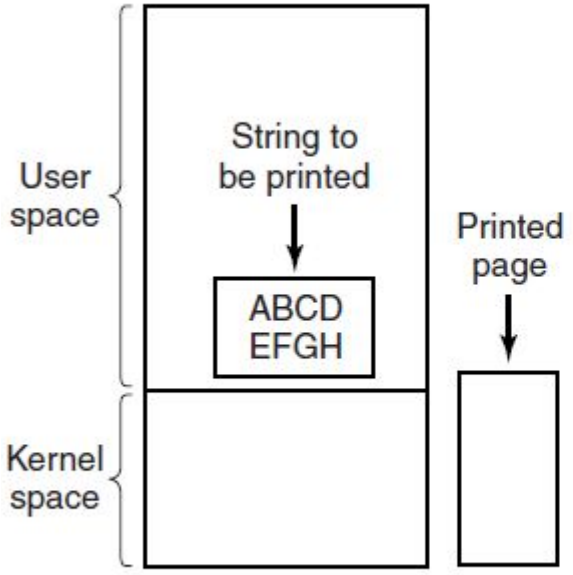
The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O.**

# Programmed I/O (2)

**Example:**

Consider a user process that wants to print the eight-character string ‘‘ABCDEFGH’’ on the printer via a serial interface.

The software first assembles the string in a buffer in user space, as shown in **Fig. 5-7(a).**




(a)

# Programmed I/O (3)

- The user process then acquires the printer for writing by making a system call to open it.
- If the printer is currently in use by another process, this call will fail and return an error code or will block until the printer is available.
- Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer.

# Programmed I/O (4)

- The operating system then (usually) copies the buffer with the string to an array, say, *p*, in kernel space, where it is more easily accessed.

- As soon as the printer is available, the operating system copies the first character to the printer's data register, in this example using memory-mapped I/O.

- **In Fig. 5-7(b),** however, we see that the first character has been printed and that the system has marked the ''B'' as the next character to be printed.

# Programmed I/O (5)

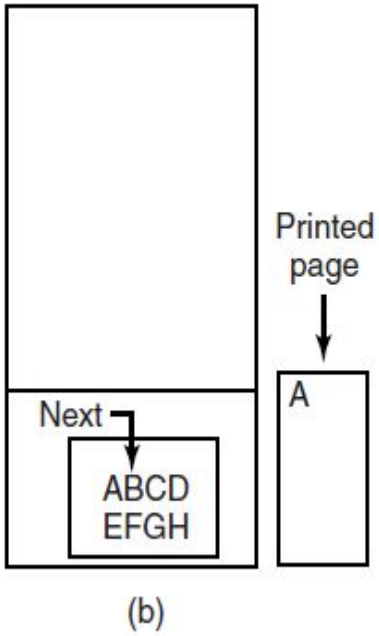



Figure 5-7. Steps in printing a string.

# Programmed I/O (6)

- When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it.

- At this point the operating system waits for the printer to become ready again.

- When that happens, it prints the next character, as shown in **Fig. 5-7(c).** This loop continues until the entire string has been printed. Then control returns to the user process.

# Programmed I/O (7)

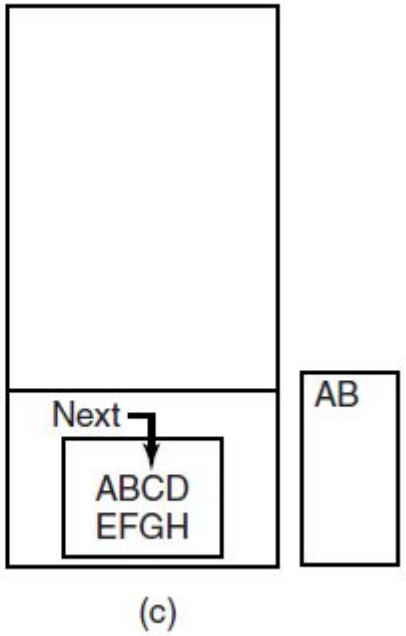



Figure 5-7. Steps in printing a string.

# Programmed I/O (8)

The actions followed by the operating system are briefly summarized in **Fig. 5-8.**

- First the data are copied to the kernel. Then the operating system enters a tight loop, outputting the characters one at a time.
- The essential aspect of programmed I/O, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called **polling** or **busy waiting**.

# Programmed I/O (9)

```
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( );
```

Figure 5-8. Writing a string to the printer using programmed I/O.

# Interrupt-Driven I/O (1)

The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts. **For Example:**

When the system call to print the string is made, the buffer is copied to kernel space, and the first character is copied to the printer as soon as it is willing to accept a character.

At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed.

# Interrupt-Driven I/O (2)

The work done on the system call is shown in **Fig. 5-9(a).**

When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt-service procedure is run. A crude version of this code is shown in **Fig. 5-9(b).**

# Interrupt-Driven I/O (3)

```
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)

Figure 5-9. 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.

# I/O Using DMA (1)

The DMA controller feed the characters to the printer one at time, without the CPU being bothered.

DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU.

This strategy requires special hardware (the DMA controller) but frees up the CPU during the I/O to do other work.

# 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)

Figure 5-10. Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

# I/O Software Layers (1)

I/O software is typically organized in four layers, as shown in **Fig. 5-11.**

Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers.

# I/O Software Layers (2)

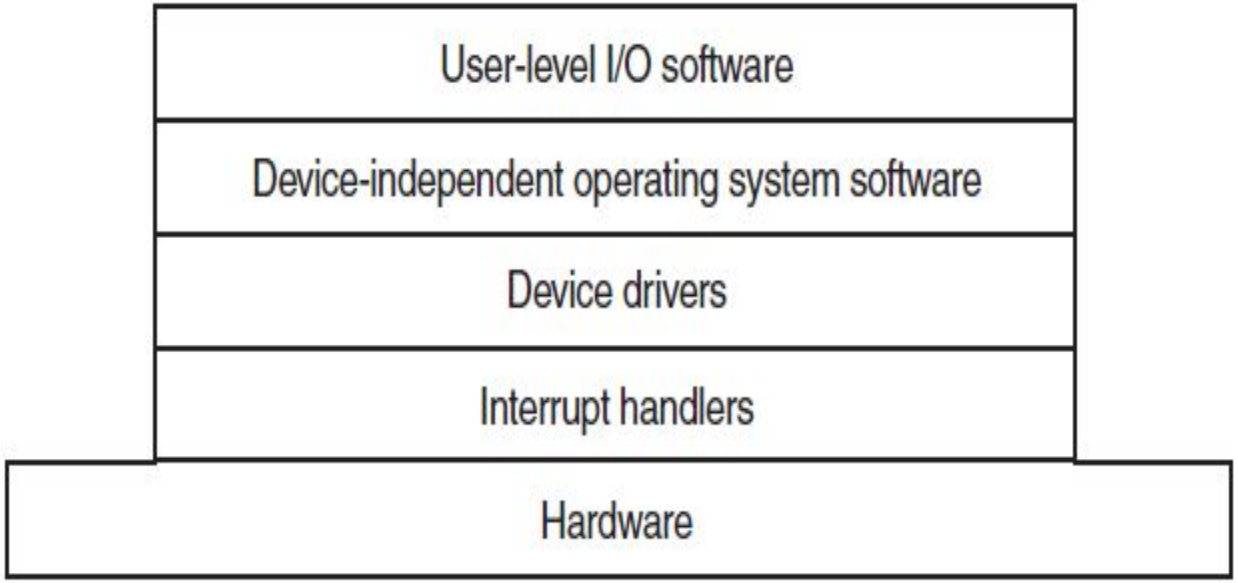



Figure 5-11. Layers of the I/O software system.

# Interrupt Handlers (1)

Typical steps after hardware interrupt completes:

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, reenable interrupts.
5. Copy registers from where saved to process table.

# Interrupt Handlers (2)

Typical steps after hardware interrupt completes:

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 (1)

Each I/O device attached to a computer needs some device specific code for controlling it. This code, called the **Device Driver**, is generally written by the device's manufacturer and delivered along with the device.

Each device driver normally handles one device type, or at most, one class of closely related devices.

# Device Drivers (2)

**For example:**

- A SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI Blu-ray disk as well.
- On the other hand, a mouse and joystick are so different that different drivers are usually required.

Device drivers are normally positioned below the rest of the operating system, as is illustrated in **Fig. 5-12.**

# Device Drivers (3)

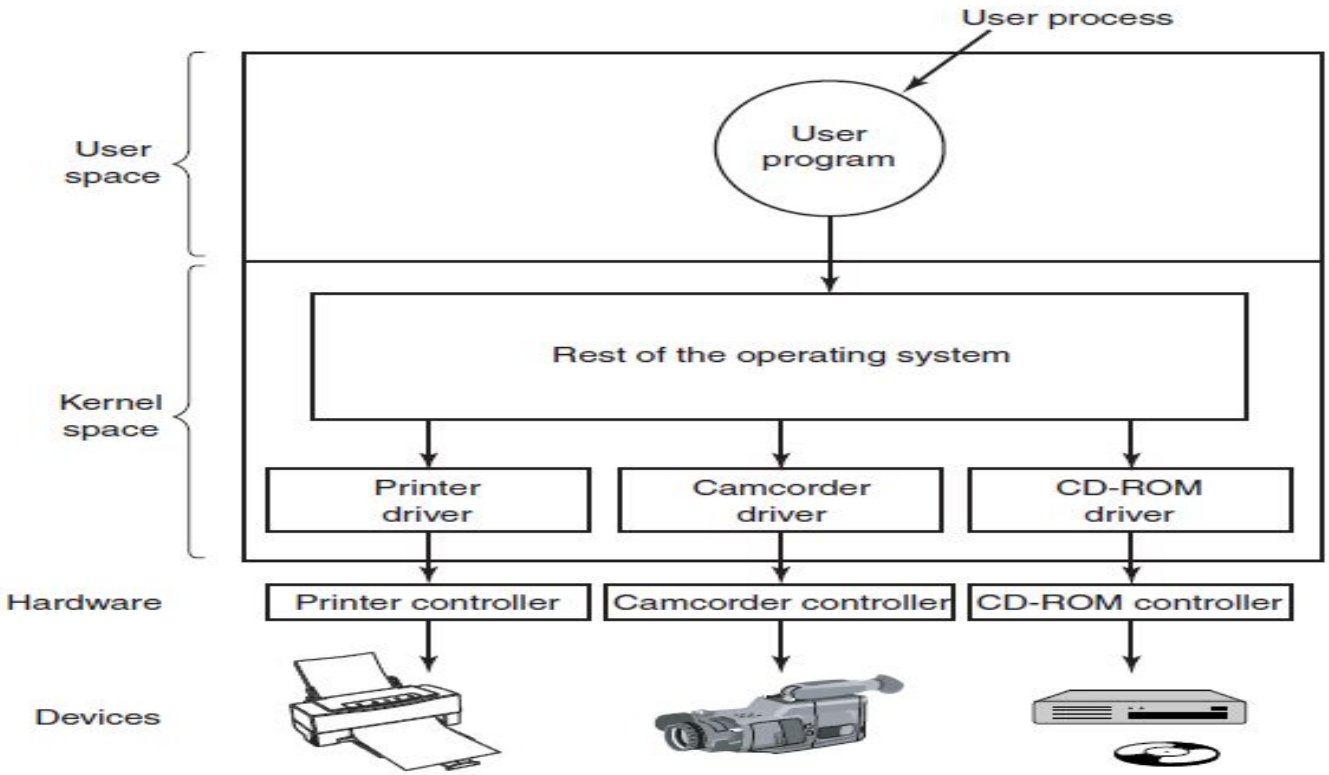



**Figure 5-12. Logical positioning of device drivers.**
**In reality all communication between drivers and device controllers goes over the bus.**

# Device-Independent I/O Software (1)

- Although some of the I/O software is device specific, other parts of it are device independent.

- The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the drivers.

- The functions shown in **Fig. 5-13** are typically done in the device-independent software.

# Device-Independent I/O Software
## (2)

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

Figure 5-13. Functions of the device-independent I/O software.

# Uniform Interfacing for Device Drivers (1)

A major issue in an operating system is how to make all I/O devices and drivers look more or less the same.

If disks, printers, keyboards, and so on, are all interfaced in different ways, every time a new device comes along, the operating system must be modified for the new device.

**Fig. 5-14(a)** illustrates a situation in which each device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver.

# Uniform Interfacing for Device Drivers (2)

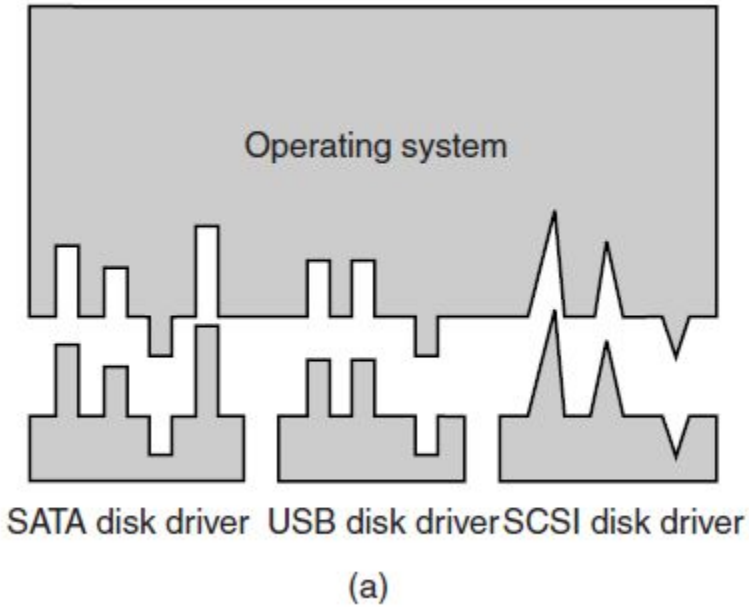



Figure 5-14. (a) Without a standard driver interface.

# Uniform Interfacing for Device Drivers (3)

**Fig. 5-14(b),** shows a different design in which all drivers have the same interface. Now it becomes much easier to plug in a new driver, providing it conforms to the driver interface.

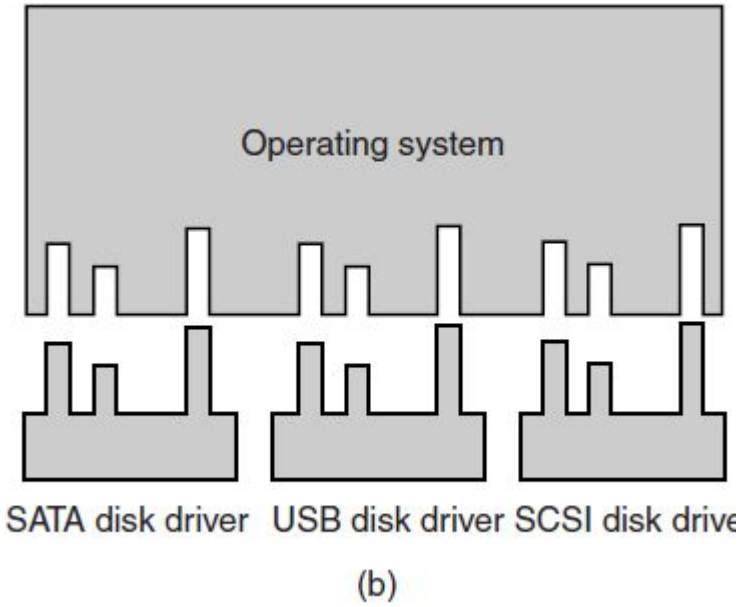



Figure 5-14. (b) With a standard driver interface.

# Buffering (1)

## Unbuffered Input

Consider a process that wants to read data from an (ADSL—Asymmetric Digital Subscriber Line) modem:

- One possible strategy for dealing with the incoming characters is to have the user process do a read system call and block waiting for one character.

- Each arriving character causes an interrupt.

- The interrupt-service procedure hands the character to the user process and unblocks it.

# Buffering (2)

## Unbuffered Input

- After putting the character somewhere, the process reads another character and blocks again. This model is indicated in **Fig. 5-15(a).**

- The trouble with this way of doing business is that the user process has to be started up for every incoming character.

- Allowing a process to run many times for short runs is inefficient, so this design is not a good one.

# Buffering (3)

## Unbuffered Input

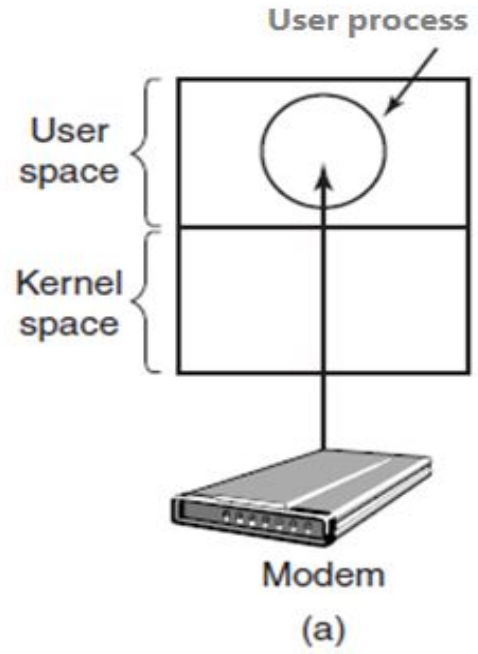



Figure 5-15. (a) Unbuffered input.

# Buffering (4)

## Buffering in User Space

An improvement is shown in **Fig. 5-15(b)**.

Here the user process provides an *n*-character buffer in user space and does a read of *n* characters.

The interrupt-service procedure puts incoming characters in this buffer until it is completely full. Only then does it wakes up the user process.

This scheme is far more efficient than the previous one, but it has a drawback: what happens if the buffer is paged out when a character arrives?

# Buffering (5)

## Buffering in User Space

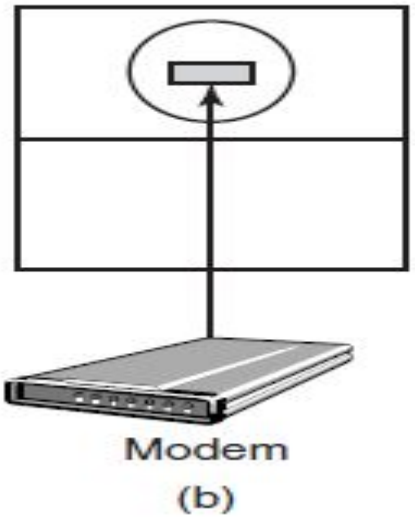



Figure 5-15. (b) Buffering in user space.

# Buffering (6)

## Buffering in the kernel followed by copying to User Space

Yet another approach is to create a buffer inside the kernel and have the interrupt handler put the characters there, as shown in **Fig. 5-15(c).**

When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation. This scheme is far more efficient.

# Buffering (7)

## Buffering in the kernel followed by copying to User Space

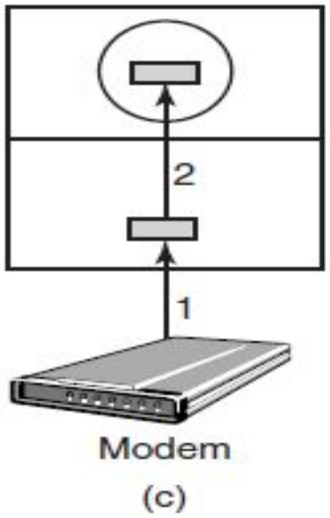



Figure 5-15. (c) Buffering in the kernel followed by copying to user space.

# Buffering (8)

## Double buffering in the kernel

What happens to characters that arrive while the page with the user buffer is being brought in from the disk?

Since the buffer is full, there is no place to put them. A way out is to have a second kernel buffer.

After the first buffer fills up, but before it has been emptied, the second one is used, as shown in **Fig. 5-15(d).**

# Buffering (9)

## Double buffering in the kernel

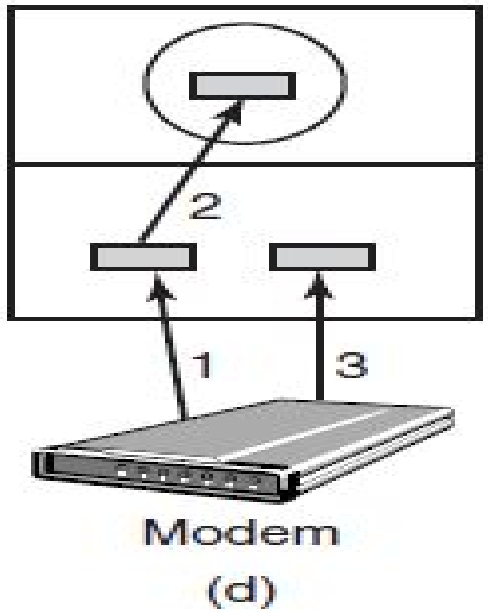



Figure 5-15. (d) Double buffering in the kernel.

# Buffering (10)

## Circular Buffer

Another common form of buffering is the **circular buffer**. It consists of a region of memory and two pointers.

One pointer points to the next free word, where new data can be placed.

The other pointer points to the first word of data in the buffer that has not been removed yet.

# Buffering (11)

## Disadvantage

If data get buffered too many times, performance suffers.

Consider, for example, the network of **Fig. 5-16.** Here a user does a system call to write to the network.

The kernel copies the packet to a kernel buffer to allow the user to proceed immediately (step1). At this point the user program can reuse the buffer.

When the driver is called, it copies the packet to the controller for output (step 2).

# Buffering (12)

## Disadvantage

- After the packet has been copied to the controller's internal buffer, it is copied out onto the network (step 3).
- Next the packet is copied to the receiver's kernel buffer (step 4).
- Finally, it is copied to the receiving process' buffer (step 5).

It should be clear that all this copying is going to slow down the transmission rate considerably because all the steps must happen sequentially.

# Buffering (13)

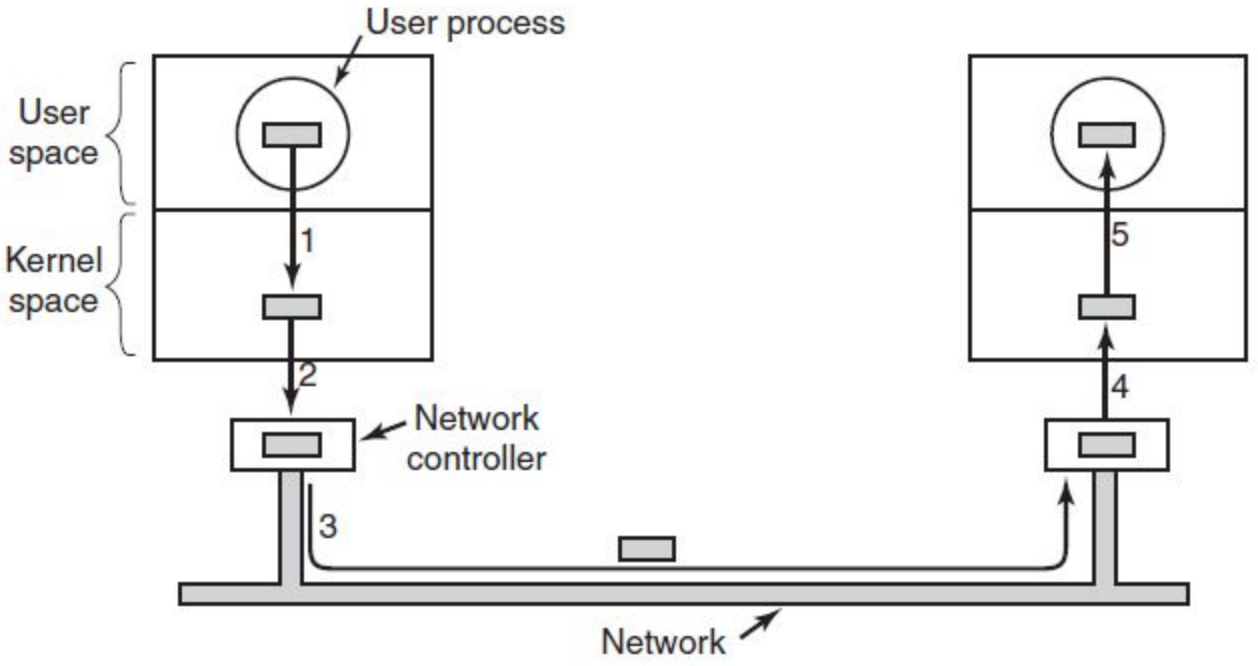



Figure 5-16. Networking may involve many copies of a packet.

# Error Reporting (1)

One class of I/O errors is programming errors.

These occur when a process asks for something impossible, such as writing to an input device (keyboard, scanner, mouse, etc.) or reading from an output device (printer, plotter, etc.).

The action to take on these errors is straightforward: just report back an error code to the caller.

# Error Reporting (2)

Another class of errors is the class of actual I/O errors, for example,

Trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off.

In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software.

# Allocating and Releasing Dedicated Devices

There are special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing.

Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

# Device-Independent Block Size

Different disks may have different sector sizes.

It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, **For example**:

By treating several sectors as a single logical block.

Similarly, some character devices deliver their data one byte at a time (e.g., mice), while others deliver theirs in larger units (e.g., Ethernet interfaces). These differences may also be hidden.

# User-Space I/O Software (1)

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. For Example,

When a C program contains the call

```
count = write(fd, buffer, nbytes);
```

The library procedure **write** might be linked with the program and contained in the binary program present in memory at run time. In other systems, libraries can be loaded during program execution. Either way, the collection of all these library procedures is clearly part of the I/O system.

# User-Space I/O Software (2)

However, Not all user-level I/O software consists of library procedures. Another important category is the spooling system.

**Spooling** is a way of dealing with dedicated I/O devices in a multiprogramming system. **For Example:**

Consider a typical spooled device: a printer. Suppose a process opened it and then did nothing for hours. No other process could print anything.

# User-Space I/O Software (3)

In order to solve this problem, a special process, called a **daemon**, and a special directory, called a **spooling directory** is created.

To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the directory.

# User-Space I/O Software (4)

**Figure 5-17** summarizes the I/O system, showing all the layers and the principal functions of each layer.

Starting at the bottom, the layers are the:

- Hardware
- Interrupt handlers
- Device drivers
- Device-independent software
- User processes.

# User-Space I/O Software (5)

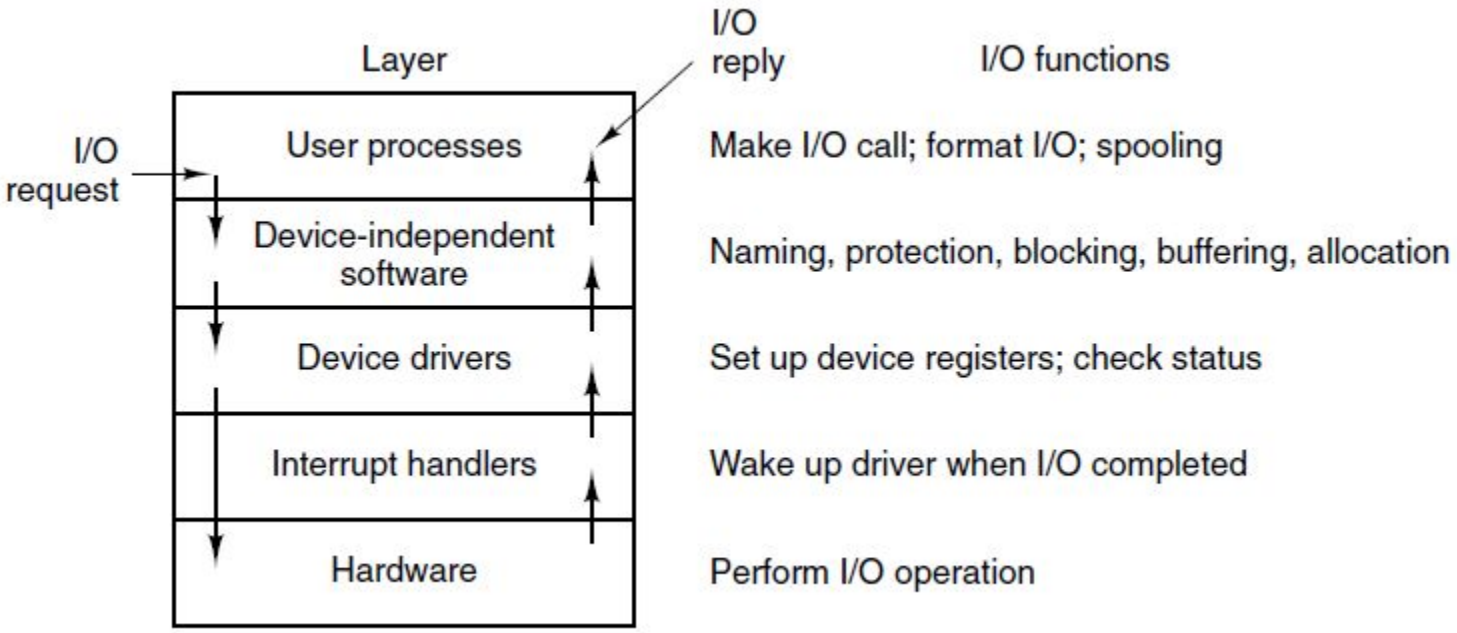



Figure 5-17. Layers of the I/O system and the main functions of each layer.

# Magnetic Disks (1)

- Magnetic disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically.

- The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on hard disks.

- The number of heads varies from 1 to about 16.

# Magnetic Disks (2)

| 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 $\mu$sec |

Figure 5-18. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD 3000 HLFS (‘‘Velociraptor’’) hard disk.

# Magnetic Disks (3)

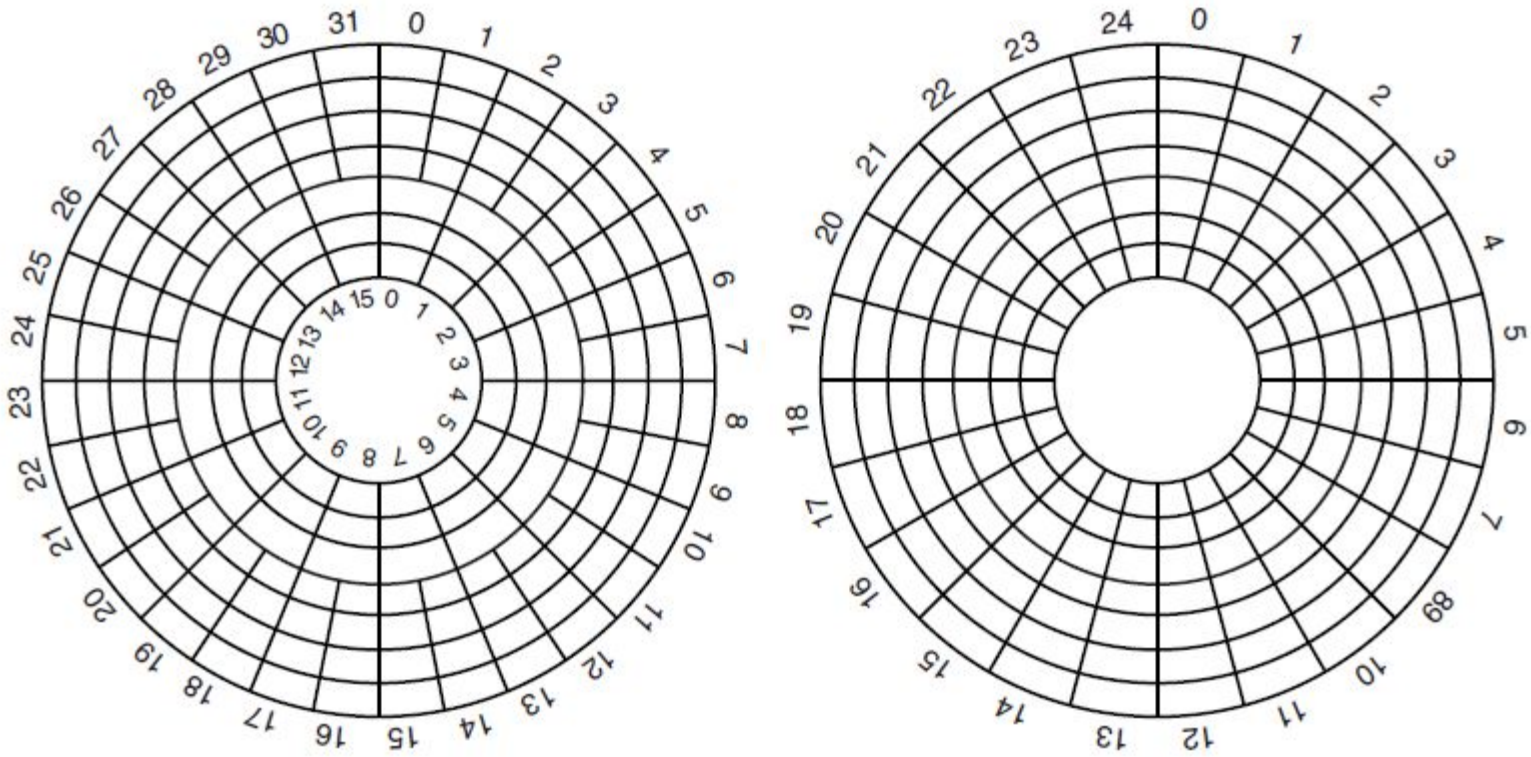



Figure 5-19. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

# RAID (1)

The fundamental idea behind a RAID is to install a box full of disks next to the computer, typically a large server, replace the disk controller card with a RAID controller, copy the data over to the RAID, and then continue normal operation.

# RAID (2)

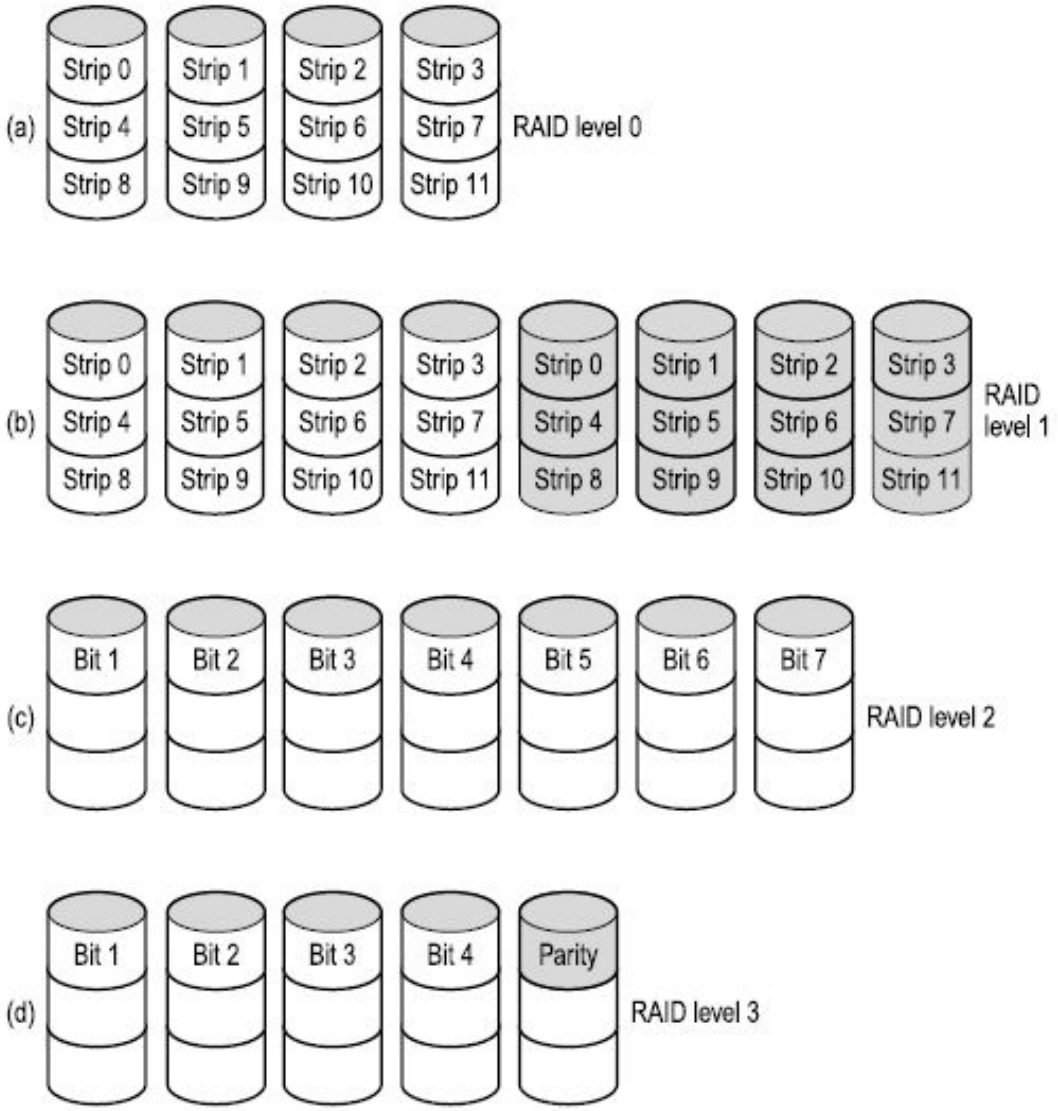



Figure 5-20. RAID levels 0 through 3. Backup and parity drives are shown shaded.

# RAID (3)

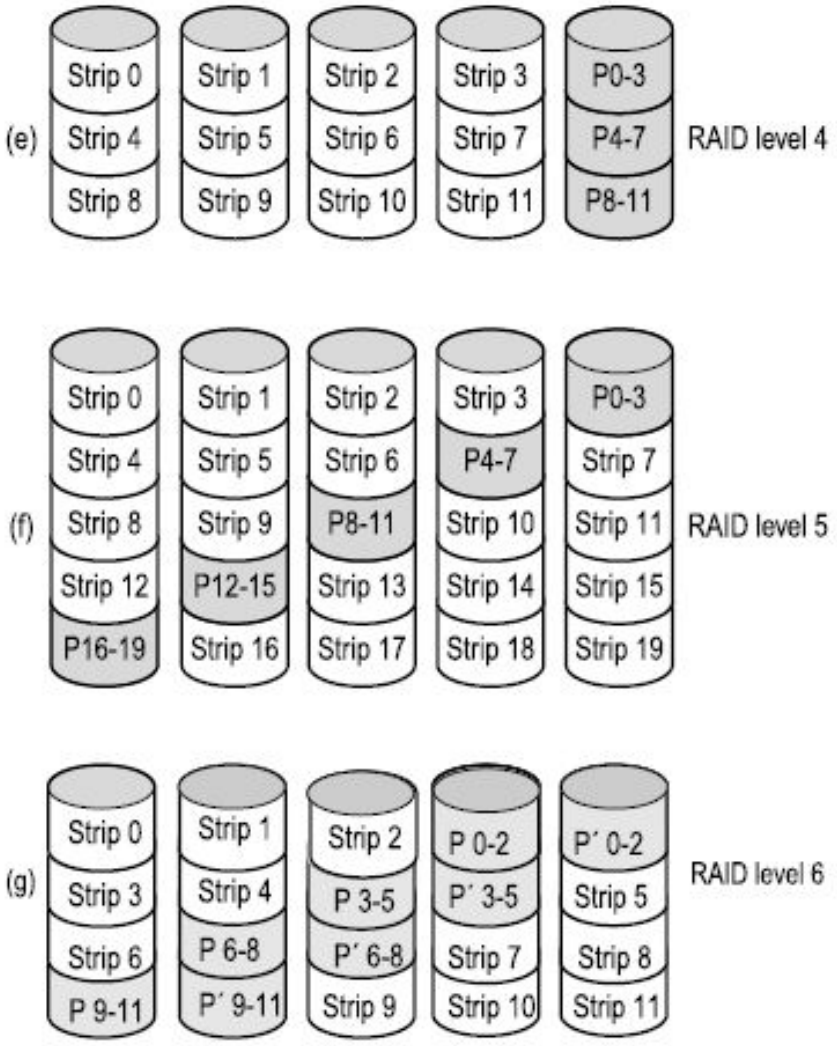



Figure 5-20. RAID levels 4 through 6. Backup and parity drives are shown shaded.

# Disk Formatting (1)

A hard disk consists of a stack of aluminum, alloy, or glass platters typically 3.5 inch in diameter.

Before the disk can be used, each platter must receive a **low-level format** done by software.

# Disk Formatting (2)


Figure 5-21. A disk sector.

# Disk Formatting (3)

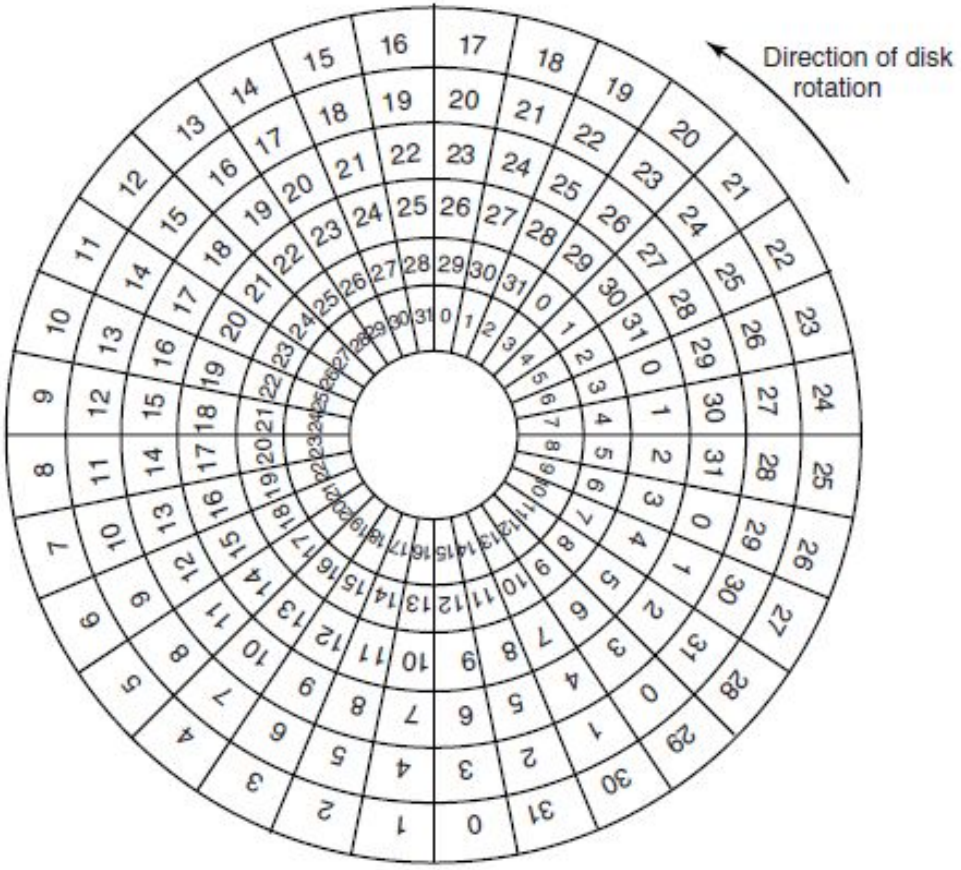



Figure 5-22. An illustration of cylinder skew.

# Disk Formatting (4)

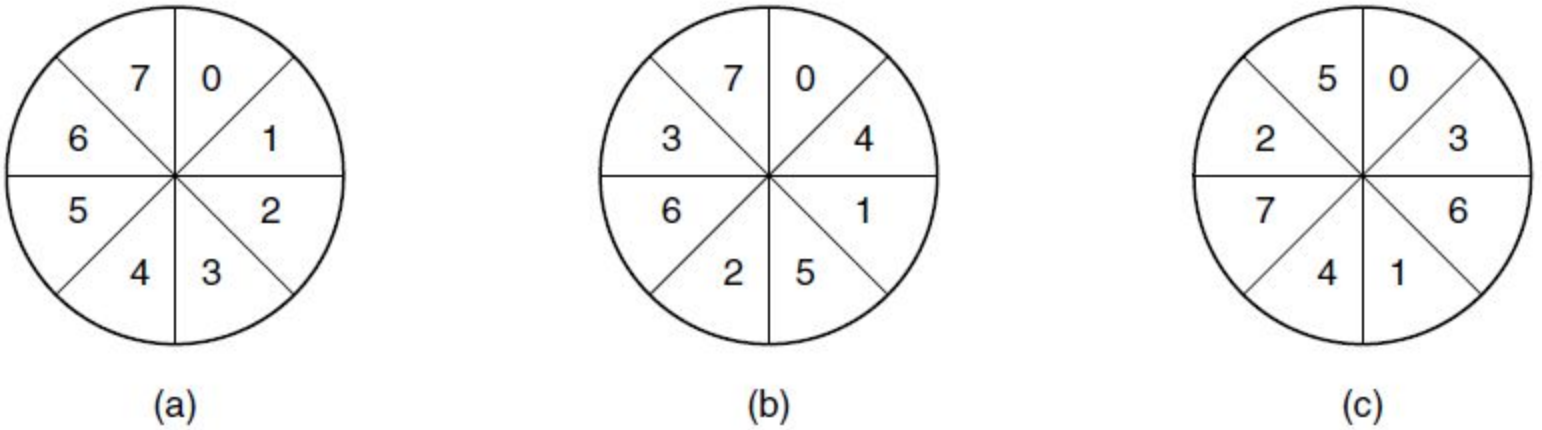



Figure 5-23. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

# Disk Arm Scheduling Algorithms (1)

To consider how long it takes to read or write a disk block. The time required is determined by three factors:

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 (2)

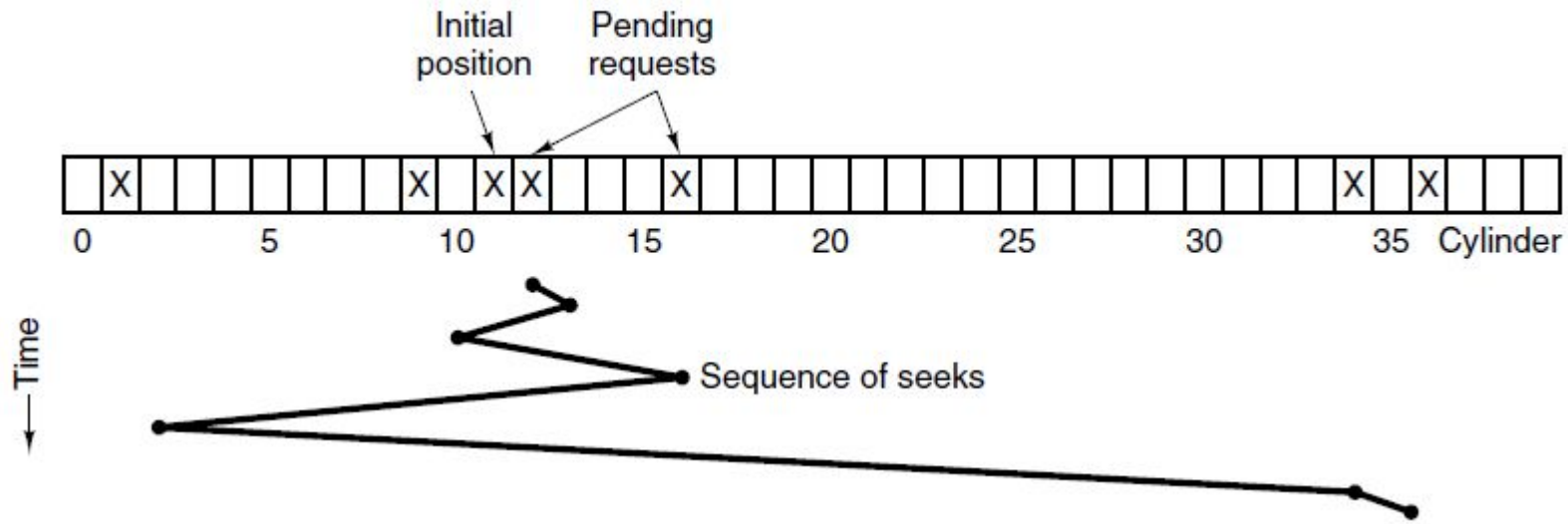



Figure 5-24. Shortest Seek First (SSF) disk scheduling algorithm.

# Disk Arm Scheduling Algorithms (3)

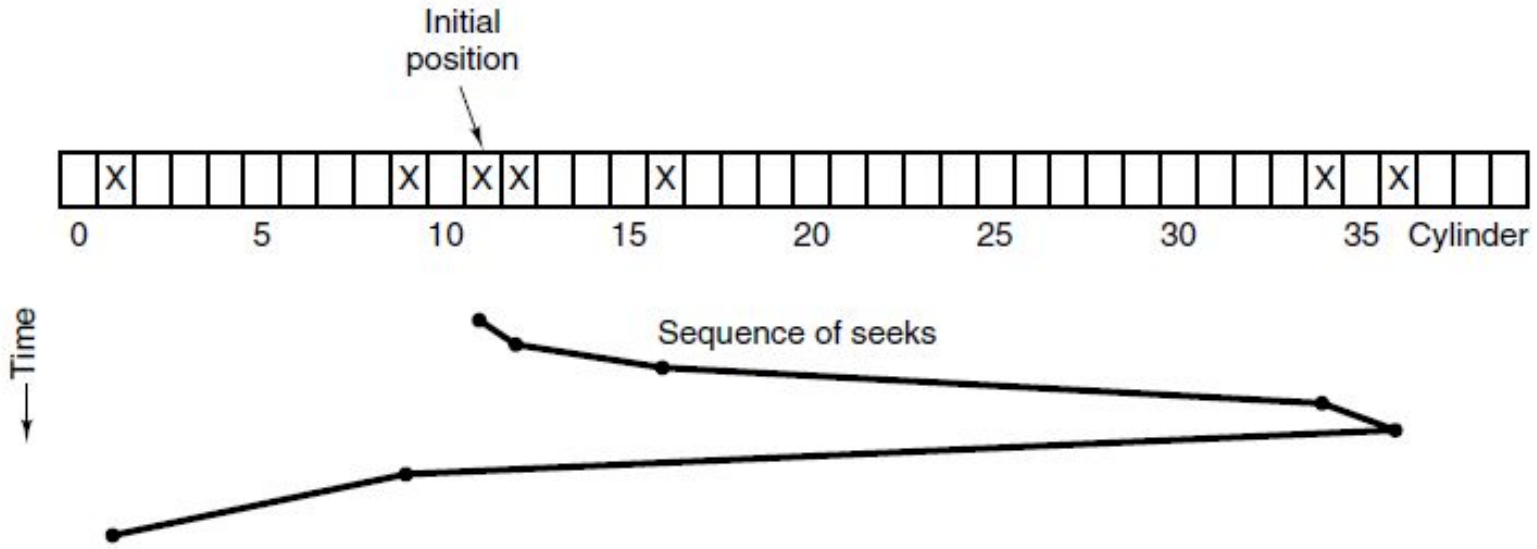



Figure 5-25. The elevator algorithm for scheduling disk requests.

# Error Handling

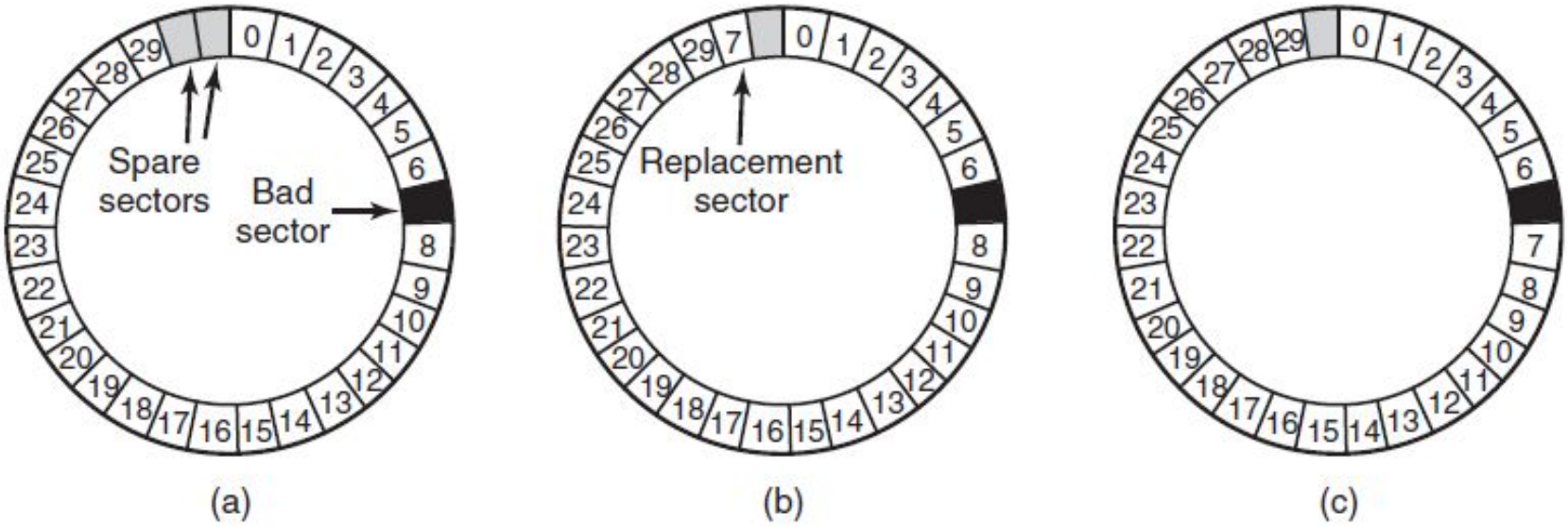



Figure 5-26. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

# Stable Storage (1)

- Uses pair of identical disks.
- Either can be read to get same results.
- Operations defined to accomplish this:
  1. Stable Writes
  2. Stable Reads
  3. Crash recovery

# Stable Storage (2)

**Stable writes:**

- A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not, the write and reread are done again up to *n* times until they work.
- After *n* consecutive failures, the block is remapped onto a spare and the operation repeated until it succeeds, no matter how many spares have to be tried.
- when a stable write completes, the block has correctly been written onto both drives and verified on both of them.

# Stable Storage (3)

**Stable reads**:

A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to *n* times.

If all of these give bad ECCs, the corresponding block is read from drive 2.Given the fact that a successful stable write leaves two good copies of the block behind, a stable read always succeeds.

# Stable Storage (4)

**Crash recovery**:

•After a crash, a recovery program scans both disks comparing corresponding blocks.

•If a pair of blocks are both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block.

•If a pair of blocks are both good but different, the block from drive 1 is written onto drive 2.

# Stable Storage (5)

In **Fig. 5-27(a),** the CPU crash happens before either copy of the block is written. During recovery, neither will be changed and the old value will continue to exist, which is allowed.

In **Fig. 5-27(b),** the CPU crashes during the write to drive 1, destroying the contents of the block. However the recovery program detects this error and restores the block on drive 1 from drive 2. Thus the effect of the crash is wiped out and the old state is fully restored.

# Stable Storage (6)

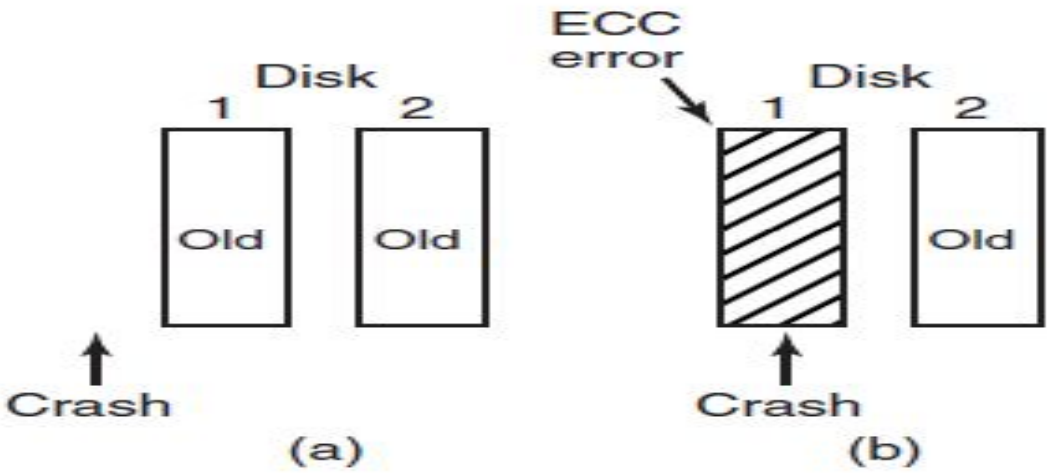



Figure 5-27. Analysis of the influence of crashes on stable writes.

# Stable Storage (7)

In **Fig. 5-27(c),** the CPU crash happens after drive 1 is written but before drive 2 is written. The point of no return has been passed here: the recovery program copies the block from drive 1 to drive 2. The write succeeds.

**Fig. 5-27(d)** is like Fig. 5-27(b): during recovery, the good block overwrites the bad block. Again, the final value of both blocks is the new one.

Finally, in **Fig. 5-27(e)** the recovery program sees that both blocks are the same, so neither is changed and the write succeeds here, too.

# Stable Storage (8)

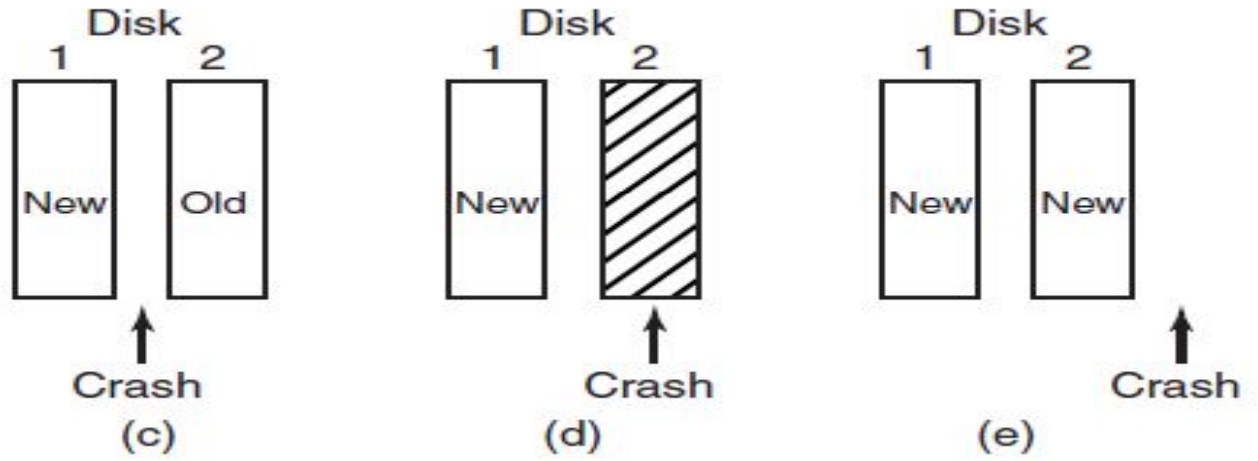



Figure 5-27. Analysis of the influence of crashes on stable writes.

# Clock Hardware (1)

Two types of clocks are commonly used in computers:

1) The simpler clocks are tied to the 110- or 220-volt power line and cause an interrupt on every voltage cycle, at 50 or 60 Hz. These clocks used to dominate, but are rare nowadays.

2) The other kind of clock is built out of three components: a crystal oscillator, a counter, and a holding register

# Clock Hardware (2)

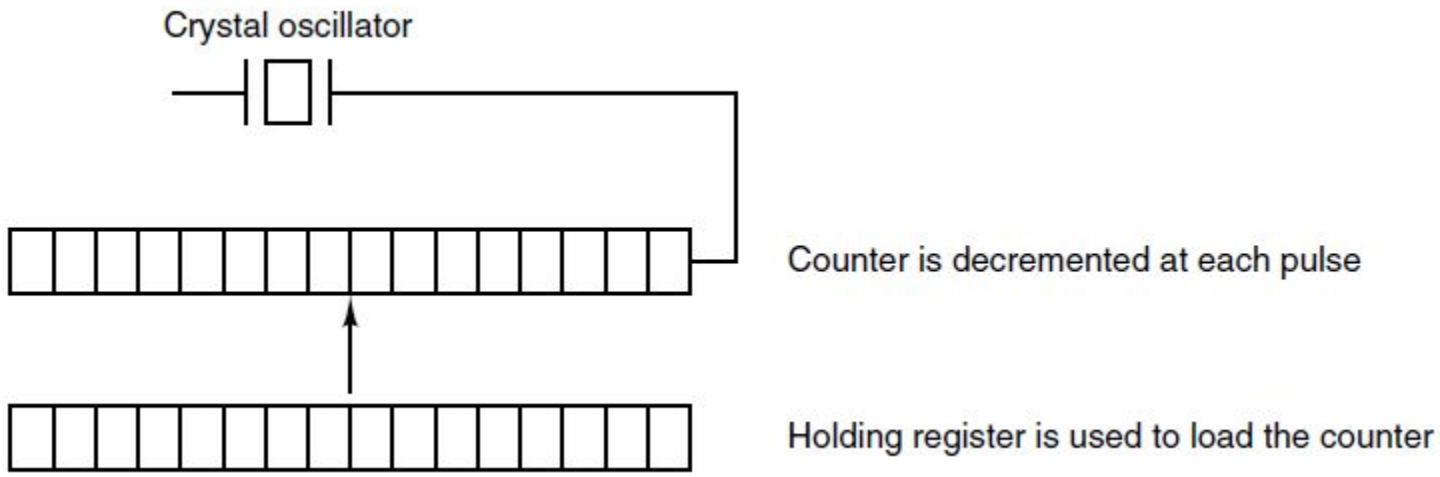



Figure 5-28. A programmable clock.

# 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 (2)

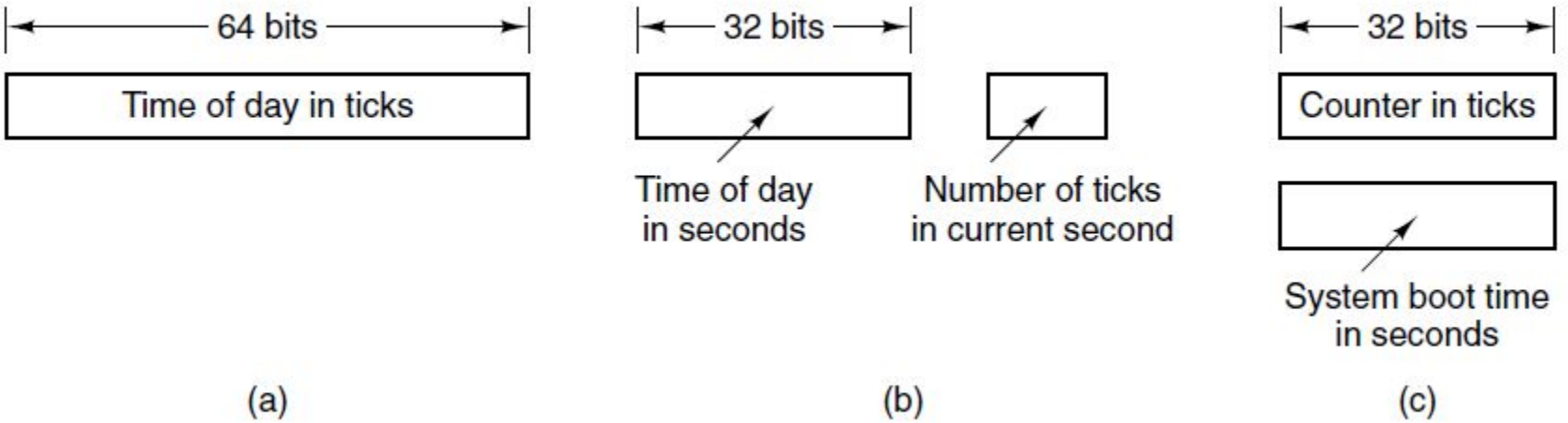



Figure 5-29. Three ways to maintain the time of day.

# Clock Software (3)

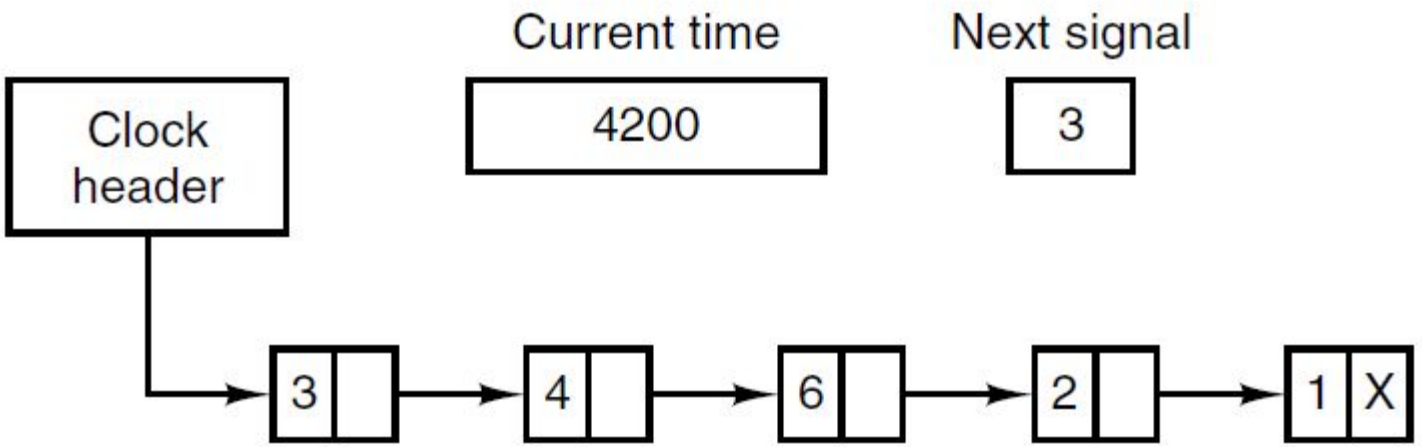



Figure 5-30. Simulating multiple timers with a single clock.

# Soft Timers (1)

Most computers have a second programmable clock that can be set to cause timer interrupts at whatever rate a program needs.

# Soft Timers (2)

Soft timers stand or fall with the rate at which kernel entries are made for other reasons. These reasons include:

1. System calls.
2. TLB misses.
3. Page faults.
4. I/O interrupts.
5. The CPU going idle.

# Keyboard Software

| Character | POSIX name | Comment |
|---|---|---|
| CTRL-H | ERASE | Backspace one character |
| CTRL-U | KILL | Erase entire line being typed |
| CTRL-V | LNEXT | Interpret next character literally |
| CTRL-S | STOP | Stop output |
| CTRL-Q | START | Start output |
| DEL | INTR | Interrupt process (SIGINT) |
| CTRL-\ | QUIT | Force core dump (SIGQUIT) |
| CTRL-D | EOF | End of file |
| CTRL-M | CR | Carriage return (unchangeable) |
| CTRL-J | NL | Linefeed (unchangeable) |

Figure 5-31. Characters that are handled specially in canonical mode.

# Output Software – Text Windows

| Escape sequence | Meaning |
|---|---|
| ESC [ *n* A | Move up *n* lines |
| ESC [ *n* B | Move down *n* lines |
| ESC [ *n* C | Move right *n* spaces |
| ESC [ *n* D | Move left *n* spaces |
| ESC [ *m* ; *n* H | Move cursor to (*m*,*n*) |
| ESC [ *s* J | Clear screen from cursor (0 to end, 1 1from start, 2 all) |
| ESC [ *s* K | Clear line from cursor (0 to end, 1 from start, 2 all) |
| ESC [ *n* L | Insert *n* lines at cursor |
| ESC [ *n* M | Delete *n* lines at cursor |
| ESC [ *n* P | Delete *n* chars at cursor |
| ESC [ *n* @ | Insert *n* chars at cursor |
| ESC [ *n* m | Enable rendition *n* (0=normal, 4=bold, 5=blinking, 7=reverse) |
| ESC M | Scroll the screen backward if the cursor is on the top line |

Figure 5-32. The ANSI escape sequences accepted by the terminal driver on output. ESC denotes the ASCII escape character (0x1B), and *n, m,* and *s* are optional numeric parameters.

# The X Window System (1)

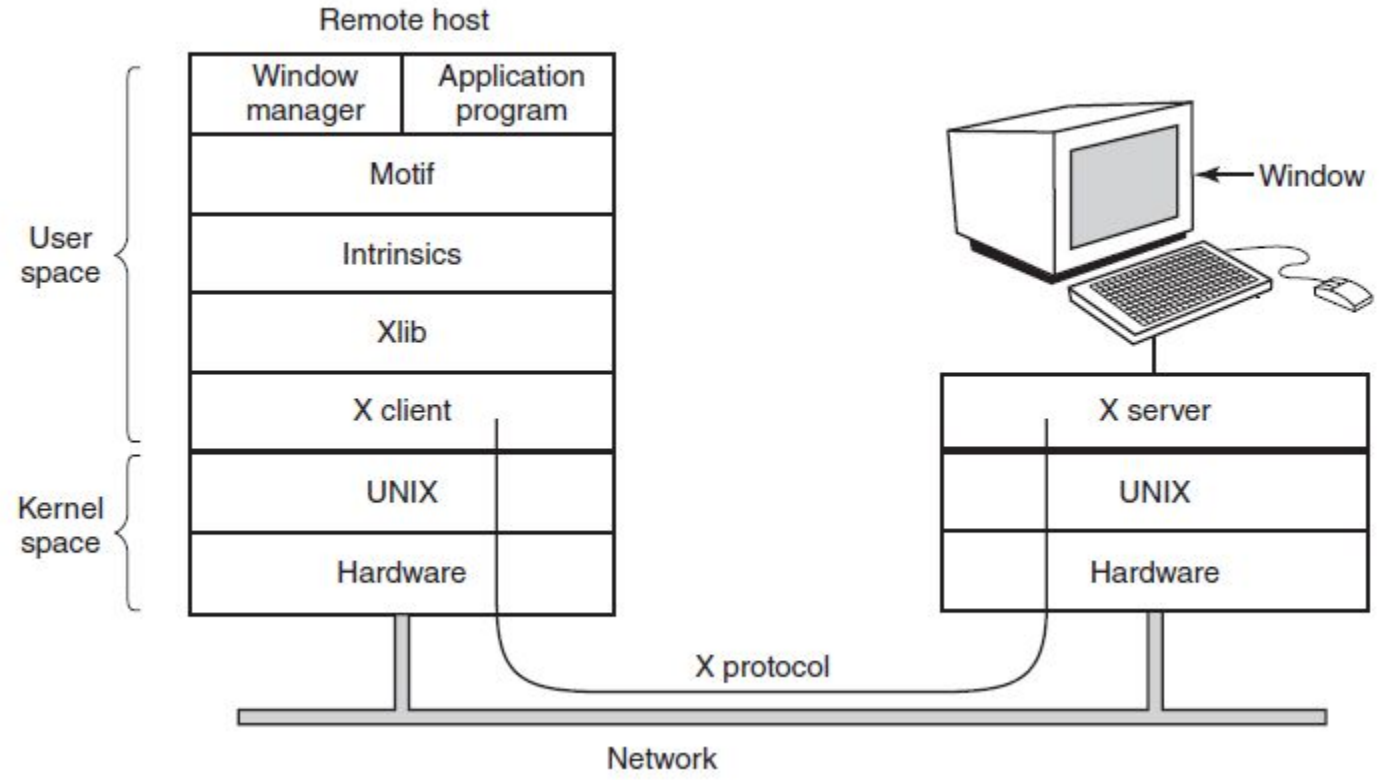



Figure 5-33. Clients and servers in the M.I.T. X Window System.

# The X Window System (2)

Types of messages between client and server:

1. Drawing commands from program to workstation.
2. Replies by workstation to program queries.
3. Keyboard, mouse, and other event announcements.
4. Error messages.

# The X Window System (3)

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
        Display disp;                                   /* server identifier */
        Window win;                                     /* window identifier */
        GC gc;                                          /* graphic context identifier */
        XEvent event;                                   /* storage for one event */
        int running = 1;

        disp = XOpenDisplay("display_name");            /* connect to the X server */
        win = XCreateSimpleWindow(disp, ... ); /* allocate memory for new window */
        XSetStandardProperties(disp, ...);      /* announces window to window mgr */
        gc = XCreateGC(disp, win, 0, 0);        /* create graphic context */
        XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
        XMapRaised(disp, win);                  /* display window; send Expose event */

        while (running) {
                XNextEvent(disp, &event);       /* get next event */
                switch (event.type) {
```

Figure 5-34. A skeleton of an X Window application program.

# The X Window System (4)

```
    XSetStandardProperties(disp, ...);        /* announces window to window mgr */
    gc = XCreateGC(disp, win, 0, 0);          /* create graphic context */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                    /* display window; send Expose event */

    while (running) {
        XNextEvent(disp, &event);             /* get next event */
        switch (event.type) {
            case Expose:      ...;  break;    /* repaint window */
            case ButtonPress: ...;  break;    /* process mouse click */
            case Keypress:    ...;  break;    /* process keyboard input */
        }
    }

    XFreeGC(disp, gc);                        /* release graphic context */
    XDestroyWindow(disp, win);                /* deallocate window's memory space */
    XCloseDisplay(disp);                      /* tear down network connection */
}
```

Figure 5-34. A skeleton of an X Window application program.

# Graphical User Interfaces (1)

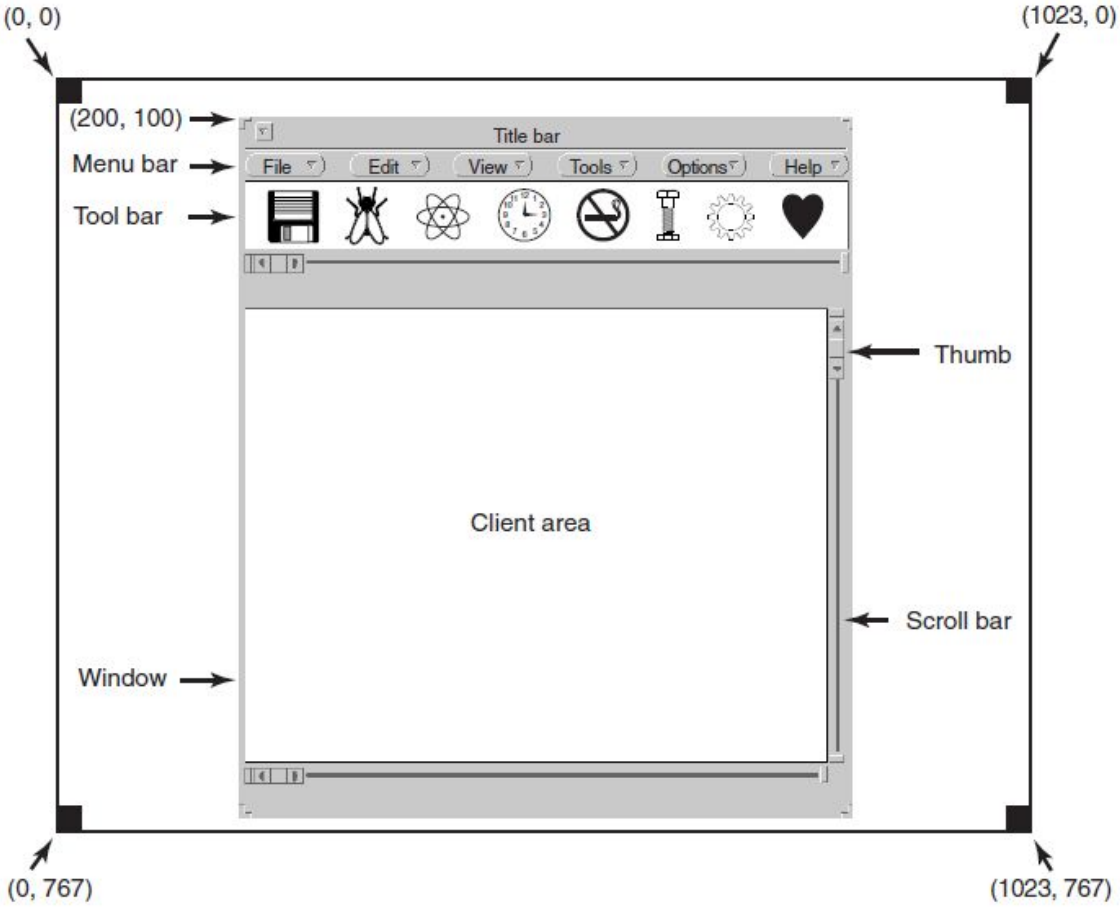



Figure 5-35. A sample window located at (200, 100) on an XGA display.

# Graphical User Interfaces (2)

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
        WNDCLASS wndclass;                  /* class object for this window */
        MSG msg;                            /* incoming messages are stored here */
        HWND hwnd;                          /* handle (pointer) to the window object */

        /* Initialize wndclass */
        wndclass.lpfnWndProc = WndProc;     /* tells which procedure to call */
        wndclass.lpszClassName = "Program name";     /* Text for title bar */
        wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);     /* load program icon */
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);       /* load mouse cursor */

        RegisterClass(&wndclass);           /* tell Windows about wndclass */
        hwnd = CreateWindow ( ... )         /* allocate storage for the window */
        ShowWindow(hwnd, iCmdShow);         /* display the window on the screen */
        UpdateWindow(hwnd);                 /* tell the window to paint itself */

        while (GetMessage(&msg, NULL, 0, 0)) {          /* get message from queue */
                TranslateMessage(&msg);         /* translate the message */
```

Figure 5-36. A skeleton of a Windows main program.

# Graphical User Interfaces (3)

```
        ShowWindow(hwnd, nCmdShow);     /* display the window on the screen */
        UpdateWindow(hwnd);             /* tell the window to paint itself */

        while (GetMessage(&msg, NULL, 0, 0)) {      /* get message from queue */
                TranslateMessage(&msg);     /* translate the message */
                DispatchMessage(&msg);      /* send msg to the appropriate procedure */
        }
        return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
        /* Declarations go here. */

        switch (message) {
                case WM_CREATE:     ... ;   return ... ;    /* create window */
                case WM_PAINT:      ... ;   return ... ;    /* repaint contents of window */
                case WM_DESTROY:    ... ;   return ... ;    /* destroy window */
        }
        return(DefWindowProc(hwnd, message, wParam, lParam));/* default */
}
```

Figure 5-36. A skeleton of a Windows main program.

# Graphical User Interfaces (4)

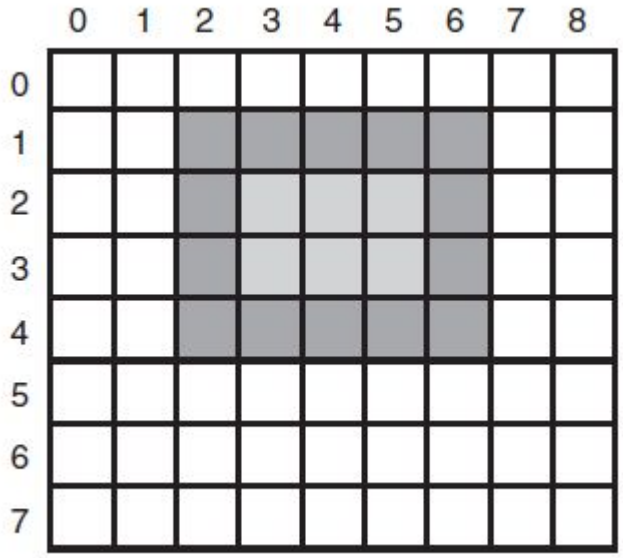



Figure 5-37. An example rectangle drawn using *Rectangle*. Each box represents one pixel.

# Bitmaps

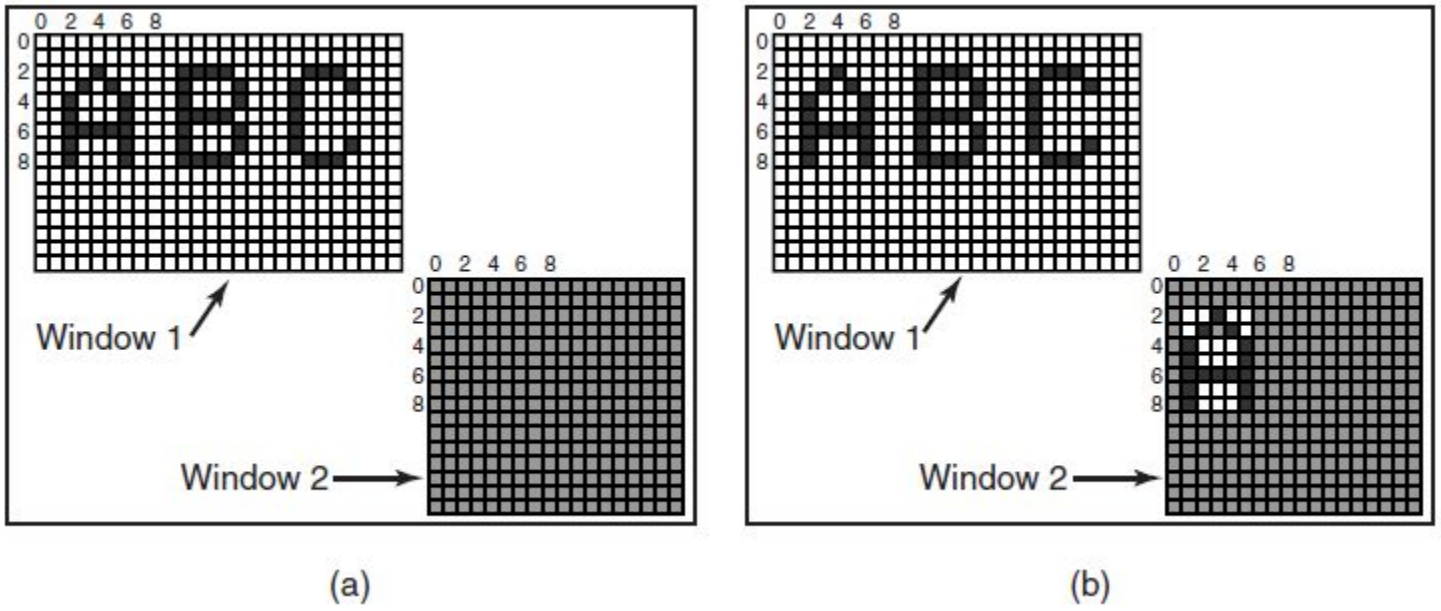



Figure 5-38. Copying bitmaps using BitBlt.
(a) Before. (b) After.

# Fonts


Figure 5-39. Some examples of character outlines at different point sizes.

# 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% |

Figure 5-40. Power consumption of various parts of a notebook computer.

# Operating System Issues
# The Display

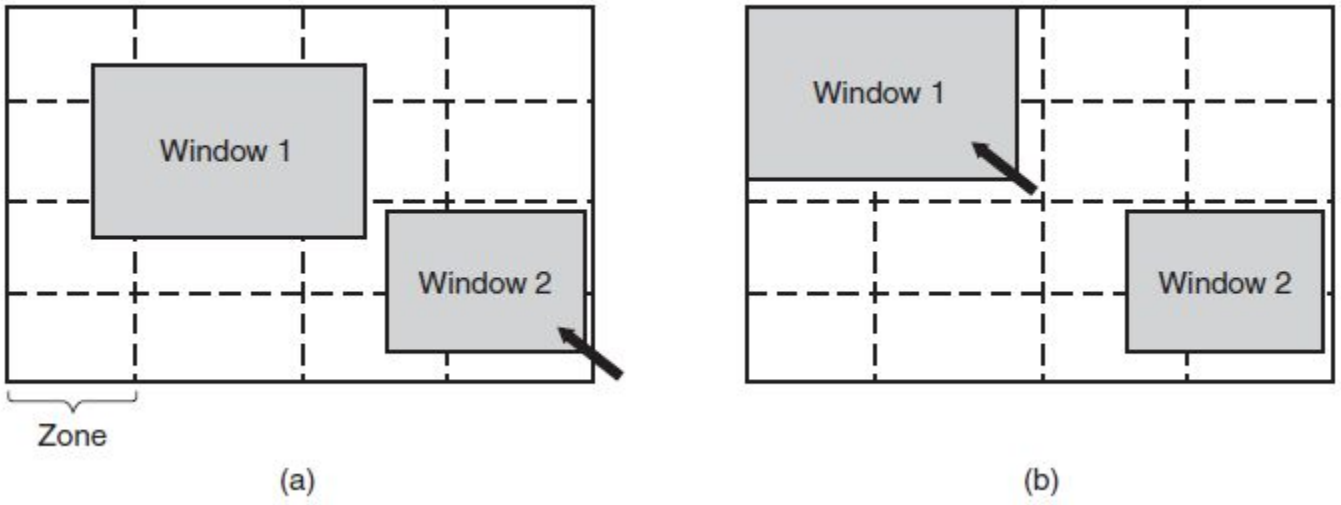



Figure 5-41. The use of zones for backlighting the display. (a) When window 2 is selected it is not moved. (b) When window 1 is selected, it moves to reduce the number of zones illuminated.

# Operating System Issues
# The CPU

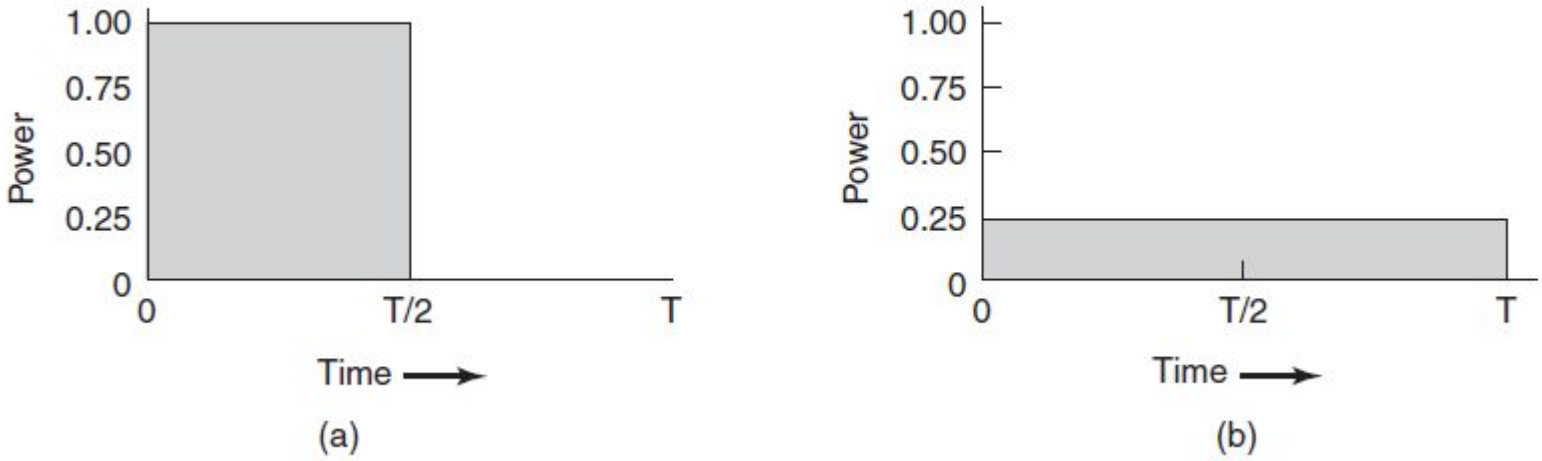



Figure 5-42. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four

**END**

Week 10 – Lecture 1

# References


- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013

  Prentice-Hall, Inc.
- http://rodrev.com/graphical/programming/Deadlock.swf
- http://www.utdallas.edu/~ilyen/animation/cpu/program/prog.html
- https://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html
- http://inventwithpython.com/blog/2013/04/22/multithreaded-python-tutorial-with threadworms/