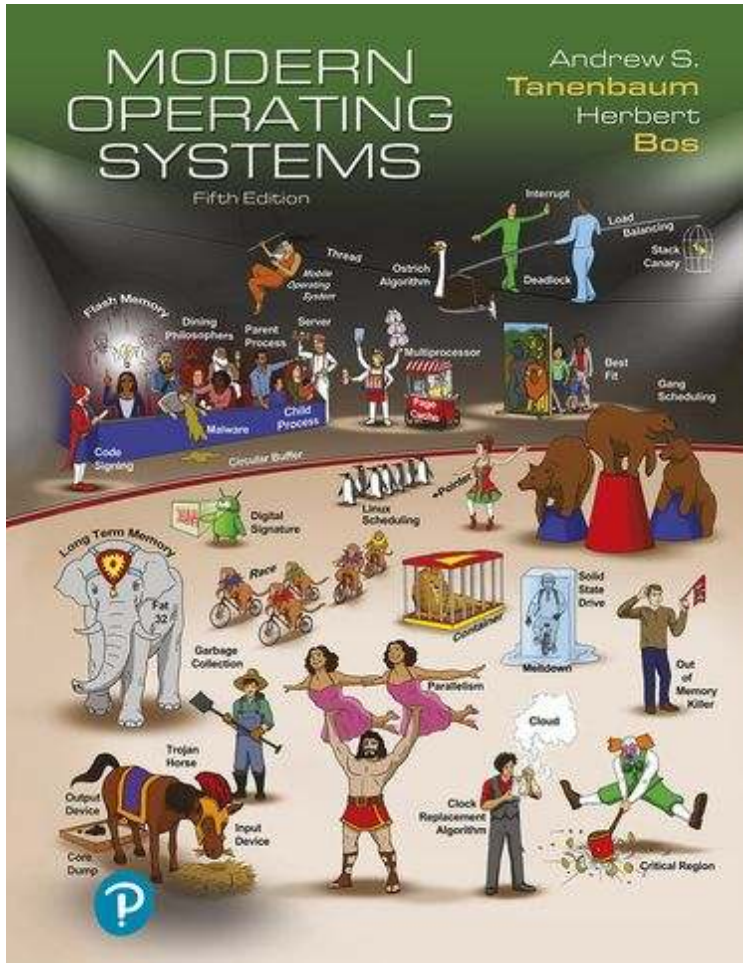
Modern Operating Systems

Fifth Edition



Chapter 5

Input /Output

Overview

- Principles of I/O Hardware
- Principles of I/O Software
- I/O Devices

Overview

- **Principles of I/O Hardware**
- Principles of I/O Software
- I/O Devices

I/O Devices (1 of 2)

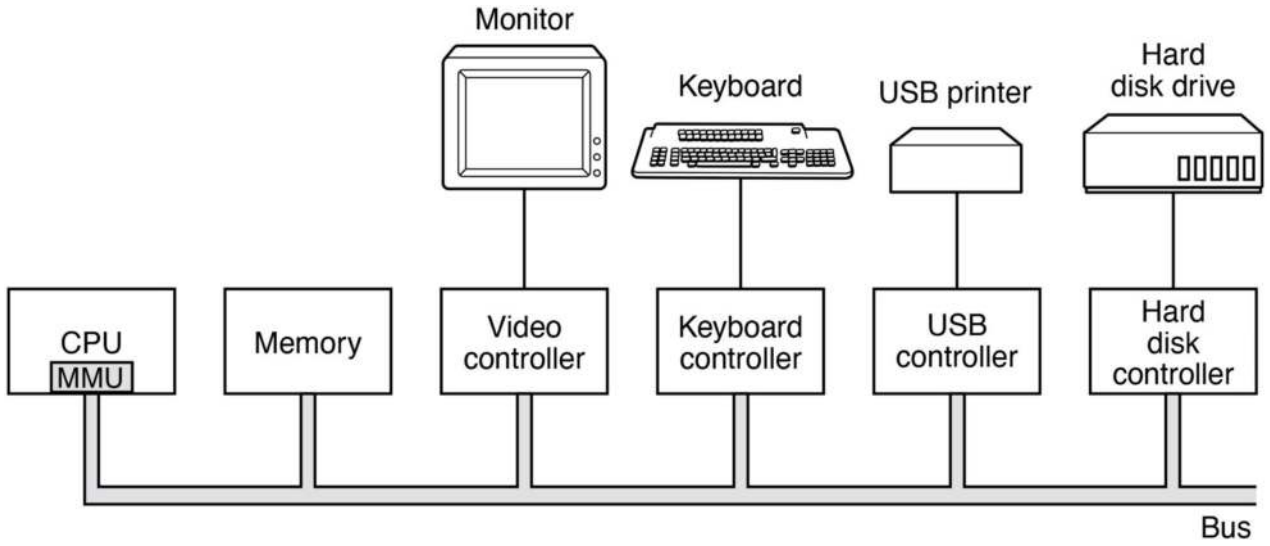
- Block devices (e.g., hard disks and SSDs)
  - Stores information in fixed-size blocks
  - Transfers are in units of entire blocks
  - I/O occurs with **random** access (by block ID)
- Character devices (e.g., printers, network interfaces)
  - No blocks, only stream of characters (or bytes)
  - Not addressable, does not have any seek operation
  - I/O occurs with **sequential** access
- Distinction is a bit blurred
  - Is a tape drive a block or a character device?
  - Some devices do not fit in this model, e.g. clocks

I/O Devices (2 of 2)

Some typical device, network, and bus data rates.

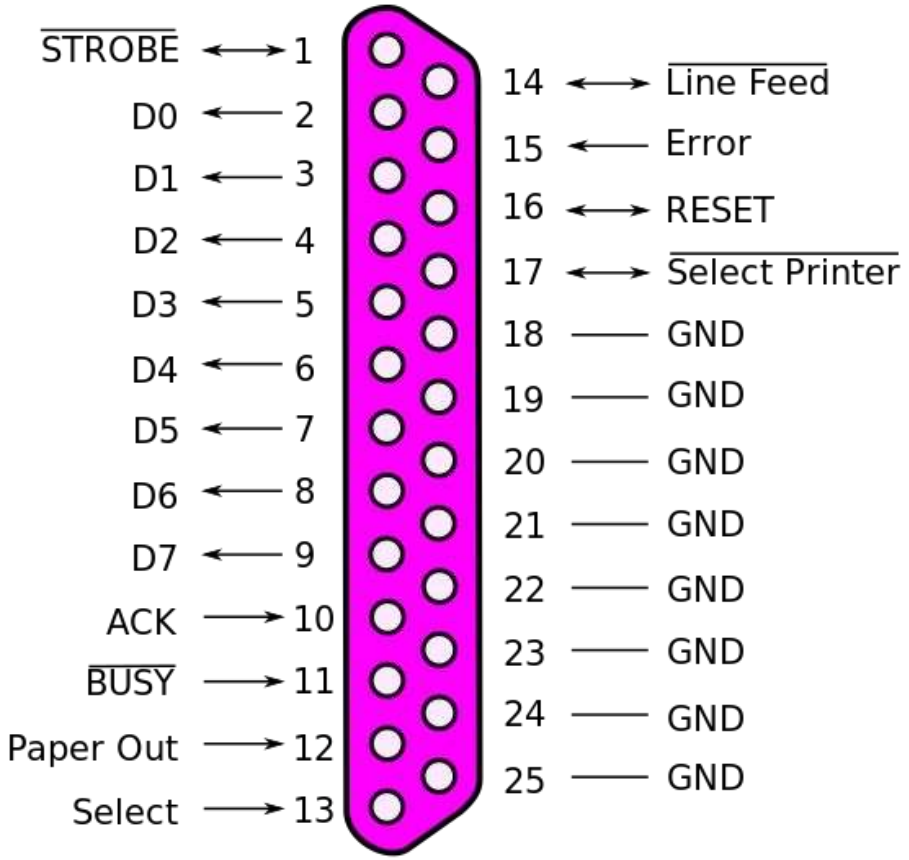
Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Bluetooth 5 BLE	256 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital video recorder	3.5 MB/sec
802.11n Wireless	37.5MB/sec
USB 2.0	60 MB/sec
16x Blu-ray disc	72 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
802.11ax Wireless	1.25 GB/sec
PCIe Gen 3.0 NVMe M.2 SSD (reading)	3.5 GB/sec
USB 4.0	5 GB/sec
PCI Express 6.0	126 GB/sec

Device Controller



- Located between actual device and computer
- Offers electronic interface in form of **I/O registers**
- R/W those registers to ask the controller to perform actions

Example: Parallel Port

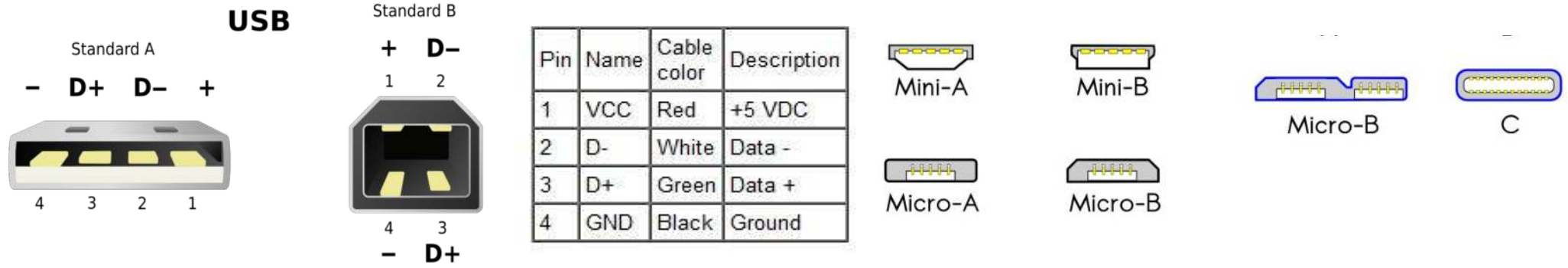


Example: Parallel Port

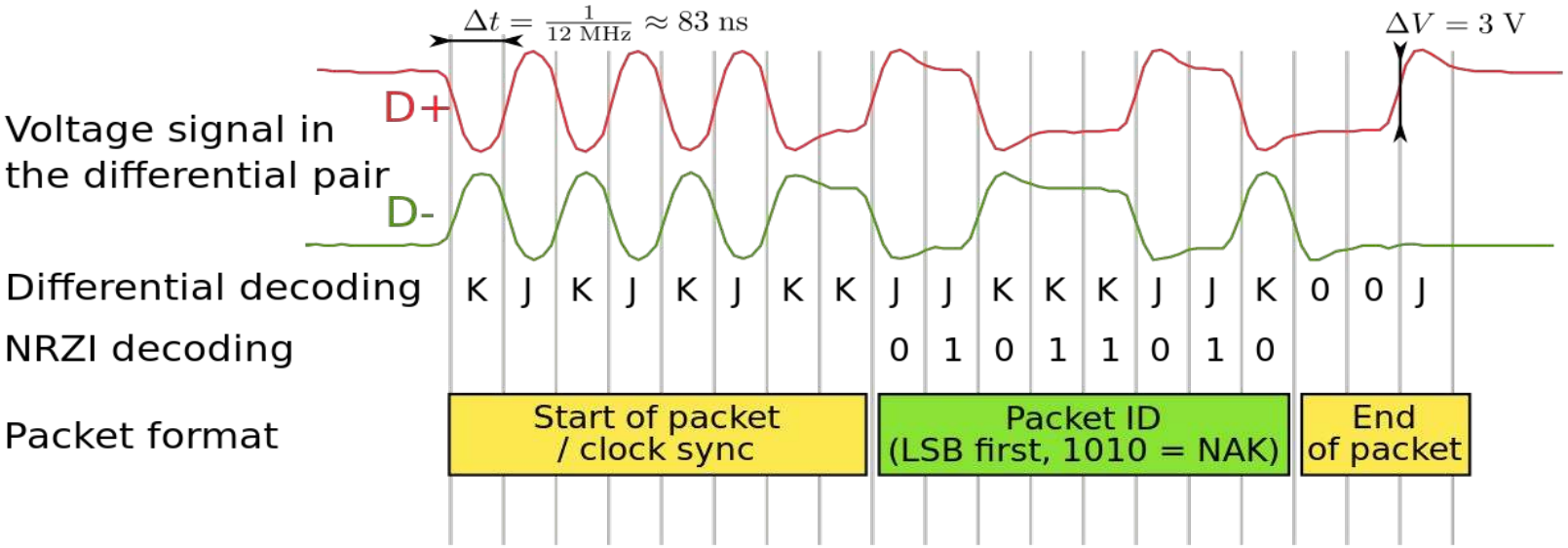
Address:		MSB							LSB
	Bit:	7	6	5	4	3	2	1	0
Base (Data port)	Pin:	D7	D6	D5	D4	D3	D2	D1	D0
Base+1 (Status port)	Pin:	BUSY	ACK	P/E	SEL	ERR			
Base+2 (Control port)	Pin:					SEL	INIT	AF	STR

Base: 0x278 → LPT1
0x2f8 → LPT2

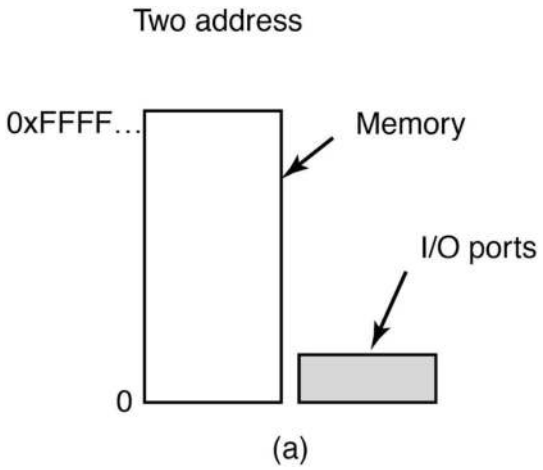
Example: USB



Pin	Name	Cable color	Description
1	VCC	Red	+5 VDC
2	D-	White	Data -
3	D+	Green	Data +
4	GND	Black	Ground



Accessing I/O Registers

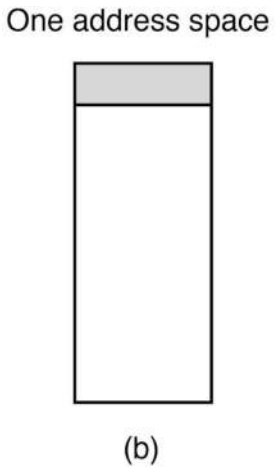


Port-mapped I/O (PMIO)

- I/O Registers are accessed via dedicated port numbers and special instructions

```
inb ax, 0x278   # load value of I/O register
                # 0x278 to register ax
```

Accessing I/O Registers



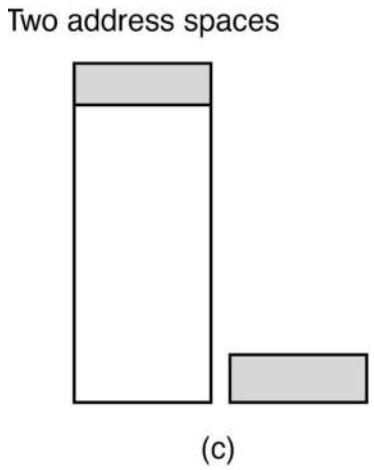
(b)

Memory-mapped I/O (MMIO)

- I/O Registers are mapped into addresses of main memory and accessed like memory

```
mov 0x278, ax    # store value of ax
                 # to address 0x278
```

Accessing I/O Registers



Hybrid (PMMIO + MMIO)

- Both implementation can coexist on a given architecture
- Example: x86

Memory-Mapped I/O

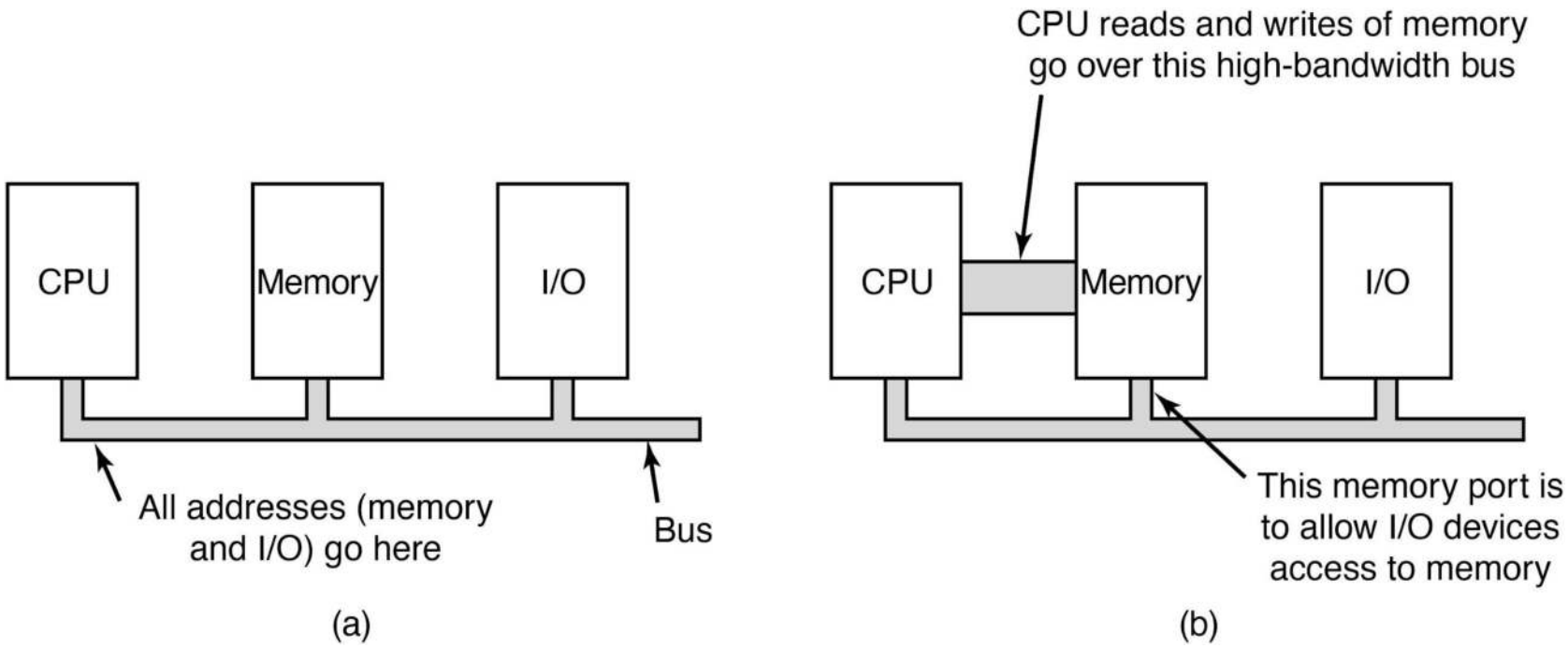


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

PMIO Example (IBM PC)

Port range	Summary
0x0000-0x001F	The first legacy DMA controller, often used for transfers to floppies.
0x0020-0x0021	The first Programmable Interrupt Controller
0x0040-0x0047	The PIT (Programmable Interval Timer)
0x0060-0x0064	The "8042" PS/2 Controller or its predecessors, dealing with keyboards and mice.
0x0070-0x0071	The CMOS and RTC registers
0x0080-0x008F	The DMA (Page registers)
0x0092	The location of the fast A20 gate register
0x00A0-0x00A1	The second PIC
0x00C0-0x00DF	The second DMA controller, often used for sound blasters
0x0170-0x0177	The secondary ATA hard disk controller.
0x01F0-0x01F7	The primary ATA hard disk controller.
0x0278-0x027A	**Parallel port**
0x02F8-0x02FF	**Second parallel port**
0x03B0-0x03DF	The range used for the IBM VGA, its direct predecessors, as well as any modern video card in legacy mode.
0x03F0-0x03F7	Floppy disk controller
0x03F8-0x03FF	First serial port

MMIO Example (OMAP 3)

Display subsystem • DSI • Display subsystem top • Display controller • RFBI • Video encoder	0x4804 FBFF	0x4804 FFFF	1KB	DSI
	0x4805 0000	0x4805 03FF	1KB	Display subsystem top
	0x4805 0400	0x4805 07FF	1KB	Display controller
	0x4805 0800	0x4805 0BFF	1KB	RFBI
	0x4805 0C00	0x4805 0FFF	1KB	Video encoder
	0x4805 1000	0x4805 1FFF	4KB	L4 interconnect
Reserved	0x4805 2000	0x4805 5FFF	16KB	Reserved
sDMA	0x4805 6000	0x4805 6FFF	4KB	Module
	0x4805 7000	0x4805 7FFF	4KB	L4 interconnect
Reserved	0x4805 8000	0x4805 FFFF	32KB	Reserved
I2C3	0x4806 0000	0x4806 0FFF	4KB	Module
	0x4806 1000	0x4806 1FFF	4KB	L4 interconnect
USBTLL	0x4806 2000	0x4806 2FFF	4KB	Module
	0x4806 3000	0x4806 3FFF	4KB	L4 interconnect
HS USB HOST	0x4806 4000	0x4806 4FFF	4KB	Module
	0x4806 5000	0x4806 5FFF	4KB	L4 interconnect
Reserved	0x4806 6000	0x4806 9FFF	16KB	Reserved
UART1	0x4806 A000	0x4806 AFFF	4KB	Module
	0x4806 B000	0x4806 BFFF	4KB	L4 interconnect
UART2	0x4806 C000	0x4806 CFFF	4KB	Module
	0x4806 D000	0x4806 DFFF	4KB	L4 interconnect
Reserved	0x4806 E000	0x4806 FFFF	8KB	Reserved

Quiz

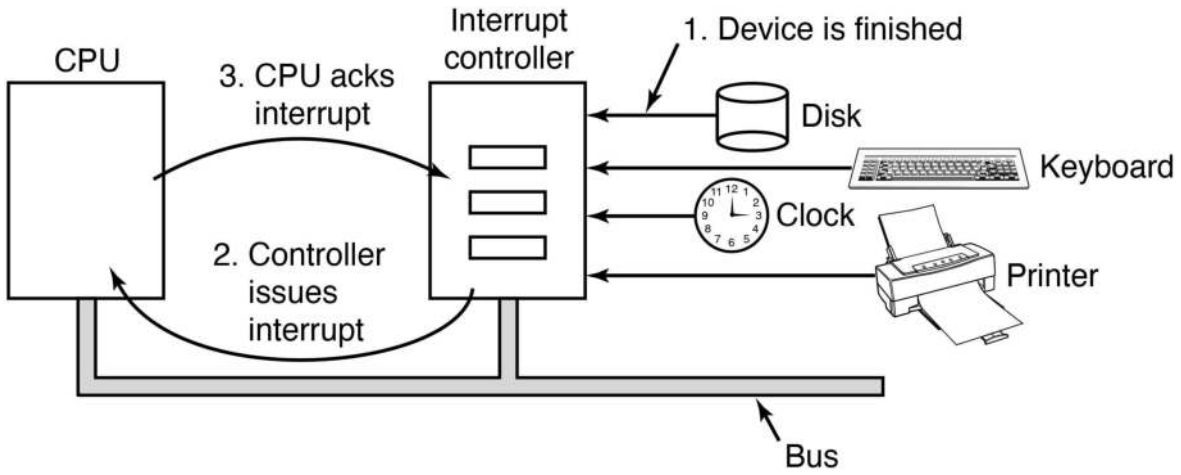
What mechanism (PMIO vs. MMIO) is more flexible and efficient to implement device access control for unprivileged user processes?

Waiting for I/O Requests…

We can now send commands to devices, but what if the requested operation takes time?

- Most devices offer status bits in their registers to signal that a request has been finished (and the error code)
- OS can poll this status bit (**polling**)
- Is this a good solution?

Interrupts



- Device controller can trigger **interrupts** to signal that a I/O request is complete
  - For the device, this means simply changing the voltage on an electrical **interrupt line**
- Each controller has **interrupt vector** assigned
  - CPU runs vector-specific handler when int occurs

Interrupts

- Different terminology used to describe interrupts
  - **Trap**: deliberate action from a program
  - **Fault** (**exception**): usually not deliberate
  - **Hardware interrupt**: device such as printer or network sends a signal to the CPU
- Interrupts can be **precise** or **imprecise**

Precise Interrupt

Four properties of a **precise interrupt**:

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

Precise vs. Imprecise

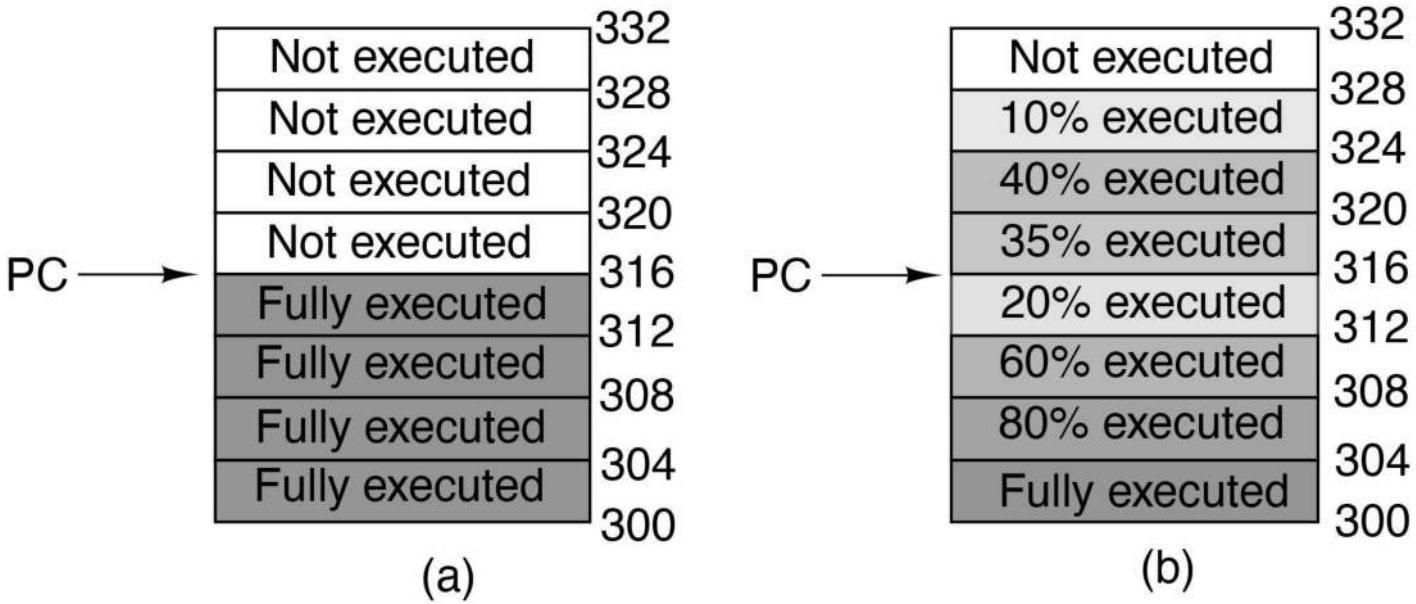


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

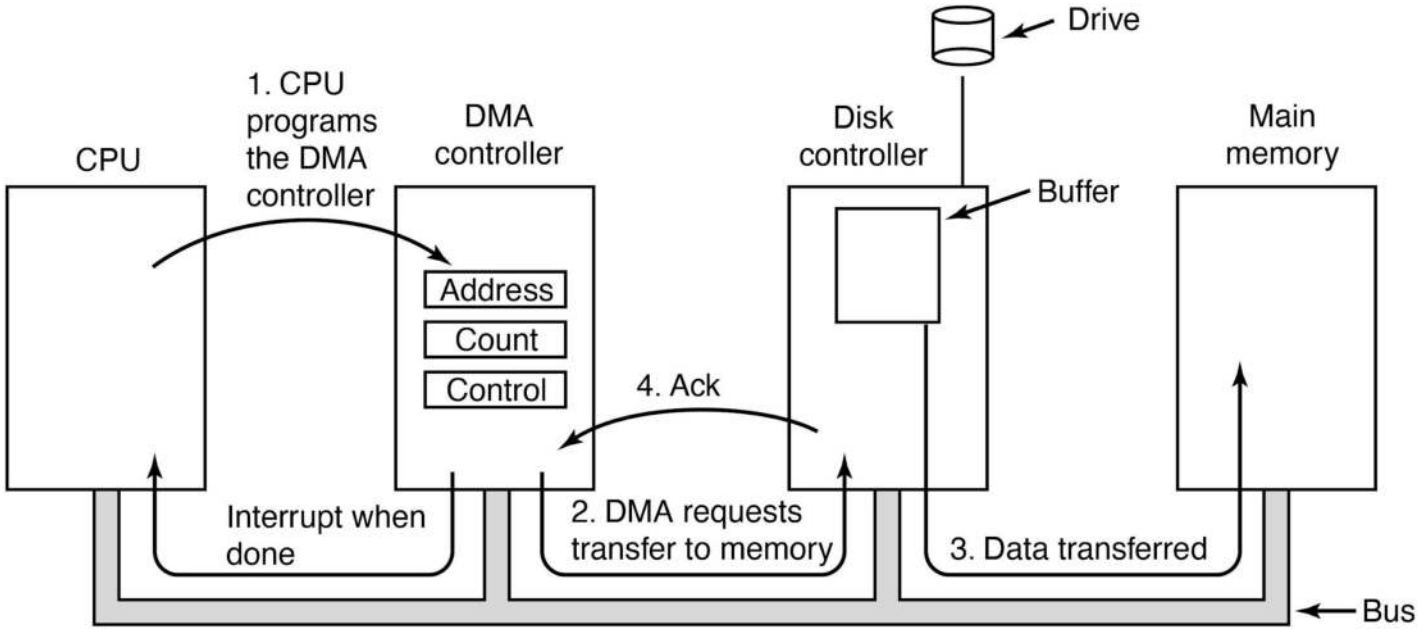
Data Exchange Between Device and CPU

- How do we transfer data from e.g., hard disk to memory?
  - Program disk controller to read a sector
  - Wait for interrupt
  - Read sizeof(sector) bytes from I/O registers
  - Repeat for next sector
- Problem?

Data Exchange Between Device and CPU

- How do we transfer data from e.g., hard disk to memory?
  - Program disk controller to read a sector
  - Wait for interrupt
  - **Read sizeof(sector) bytes from I/O registers**
  - Repeat for next sector
- Problem?
  - CPU cycles can be spent in a better way!

Direct Memory Access (DMA)



- Solution:
  - Let the hardware do the transfer via DMA!

DMA Controller

- On ISA systems there was a dedicated DMA controller (**third-party DMA**)
- On PCI (and PCIe) systems each PCI device may become “Bus Master” and perform DMA (**first-party DMA**)
  - Device and DMA controller are combined
  - Do you see a problem here?

DMA Controller

- On ISA systems there was a dedicated DMA controller (**third-party DMA**)
- On PCI (and PCIe) systems each PCI device may become “Bus Master” and perform DMA (**first-party DMA**)
  - Device and DMA controller are combined
  - Do you see a problem here?
  - **You have to trust your Hw (or use a I/O MMU)**
- Note:
  - Embedded systems still have dedicated DMA controller
  - Disk controller still uses own buffers

Quiz

You're comparing the performance of two systems, both using no interrupts for device data transfer, but one using DMA + polling while the other system uses polling alone.

Do you expect comparable or different performance for the two systems?

Overview

- Principles of I/O Hardware
- **Principles of I/O Software**
- I/O Devices

Principles of I/O Software

Issues:

- Device independence
- Uniform naming
- Error handling
- Buffering
- Synchronous versus asynchronous

Principles of I/O Software

- **Device independence**
  - I/O software provides abstraction over actual hardware
  - Programs should not have to care about device particularities
- **Uniform naming**
  - We don’t want to type ST6NM04 to address the first hard disk
  - /dev/sda is better
  - /mnt/movies even better
- **Error handling**
  - Errors should be handled closest to their source

Principles of I/O Software

- **Buffering**
  - Networking: incoming packets have to be buffered
  - Audio: buffering to avoid clicks
- **Synchronous vs. asynchronous I/O**
  - Programs don’t want to deal with interrupts → OS turns async operations into blocking operations
  - Lower levels have to deal with interrupts, DMA, etc.

Programmed I/O (1 of 2)

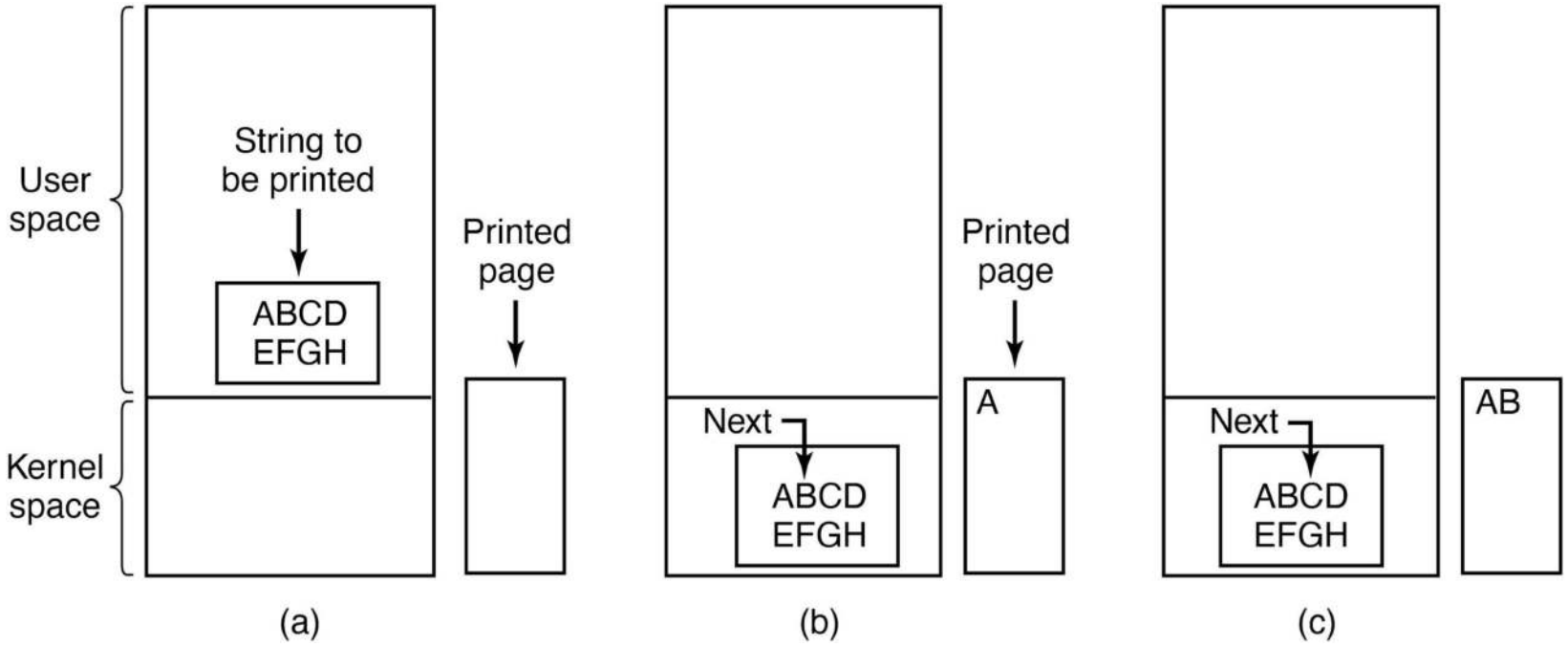


Figure 5-7. Steps in printing a string.

Programmed I/O (2 of 2)

```
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

```
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

```
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

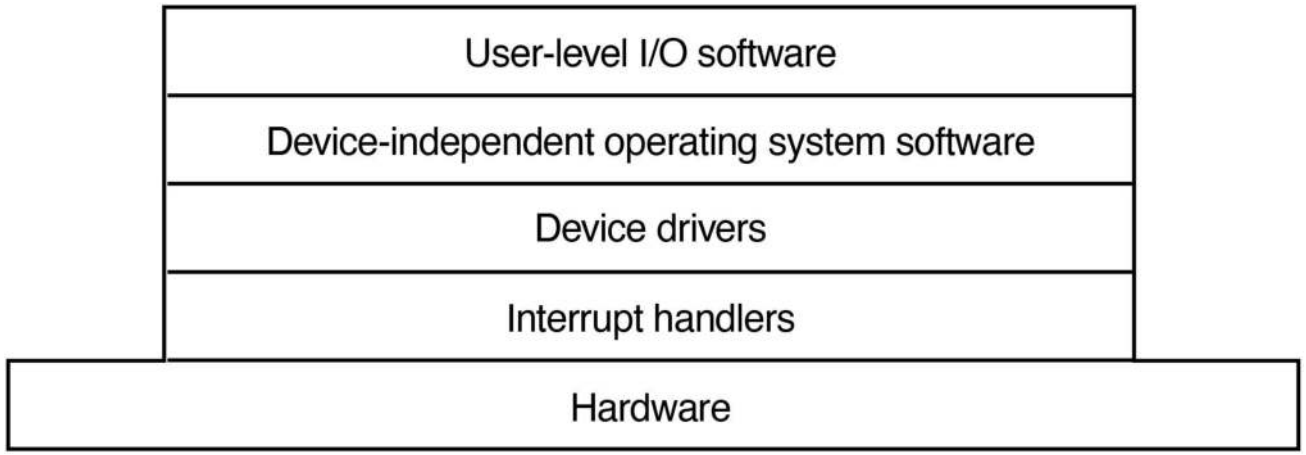


Figure 5-11. Layers of the I/O software system.

Interrupt Handlers (1 of 2)

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, re-enable interrupts.
5. Copy registers from where saved to process table.

Interrupt Handlers (2 of 2)

Typical steps after hardware interrupt completes:

6. Run interrupt service procedure. Typically, extract information from interrupting device controller's registers. Often split into low-level interrupt service procedure (ran immediately) and high-level interrupt service procedure (scheduled later).
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 Driver

- Accepts abstract requests from device-independent layer and transforms them into commands for the device
- Can queue new requests as necessary
- Usually needs to wait for the completion of the request (IRQ)
- Checks for errors as necessary
- Answers requests from device-independent layer

Device Drivers

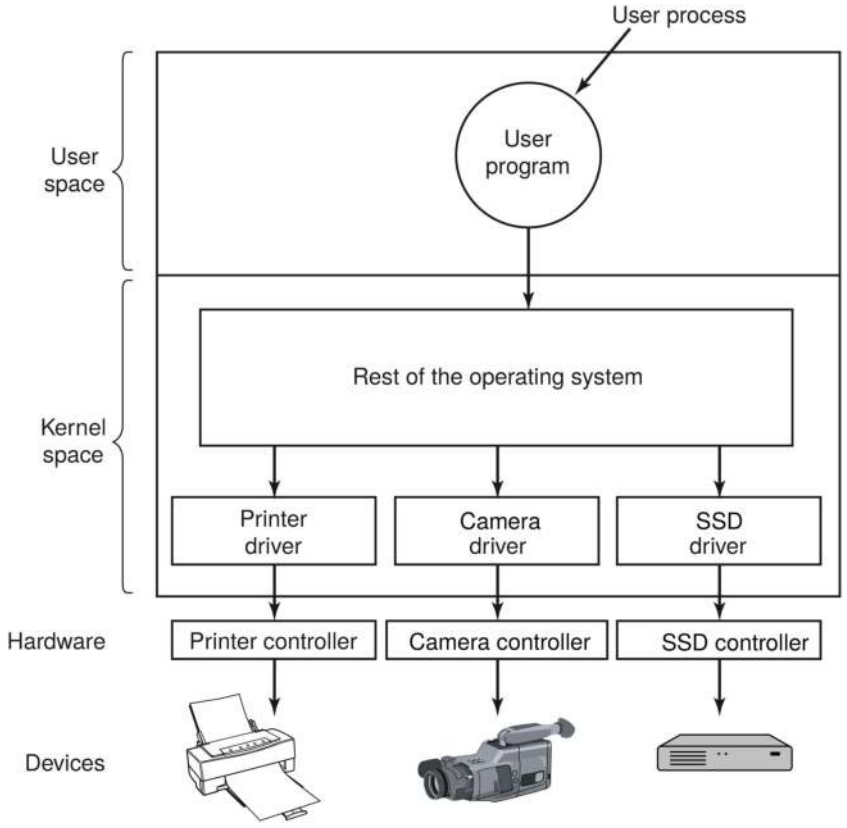


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Device-Independent I/O Software

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

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

Device-Independent I/O Software

- **Uniform interfacing for device drivers**
  - Driver interface, naming (e.g., major/minor numbers), protection
- **Buffering**
  - Necessary for both character and block devices
- **Error reporting**
  - Hardware-level, driver-level, etc.
- **Allocating and releasing dedicated devices**
  - e.g., Printers
- **Providing a device-independent block size**
  - Unify blocks/characters across devices

Uniform Interfacing for Device Drivers

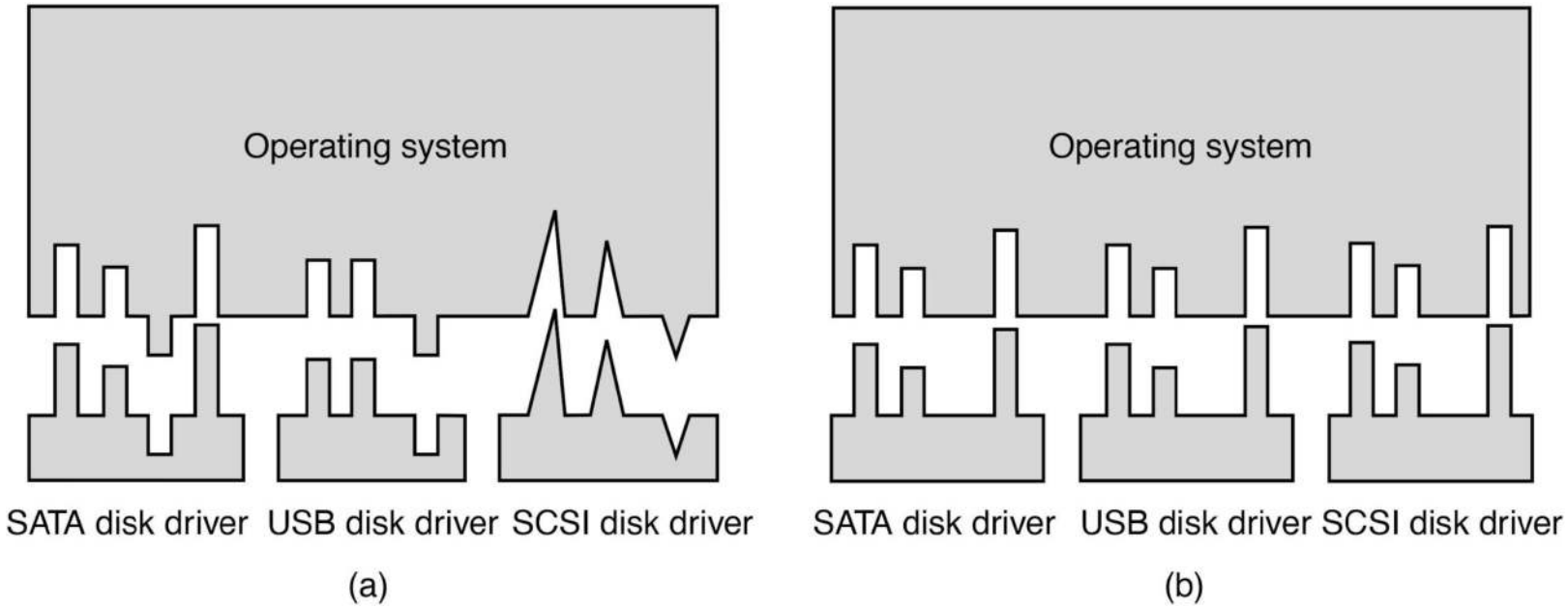


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

Buffering (1 of 2)

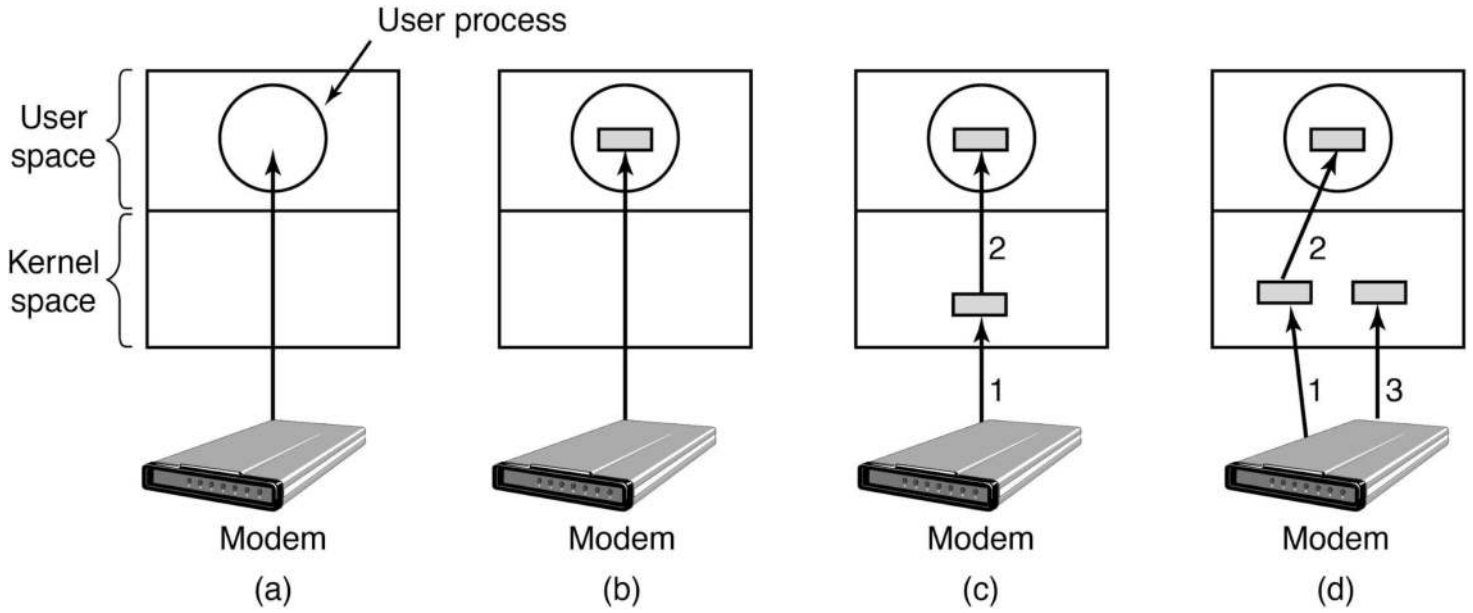


Figure 5-15. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Buffering (2 of 2)

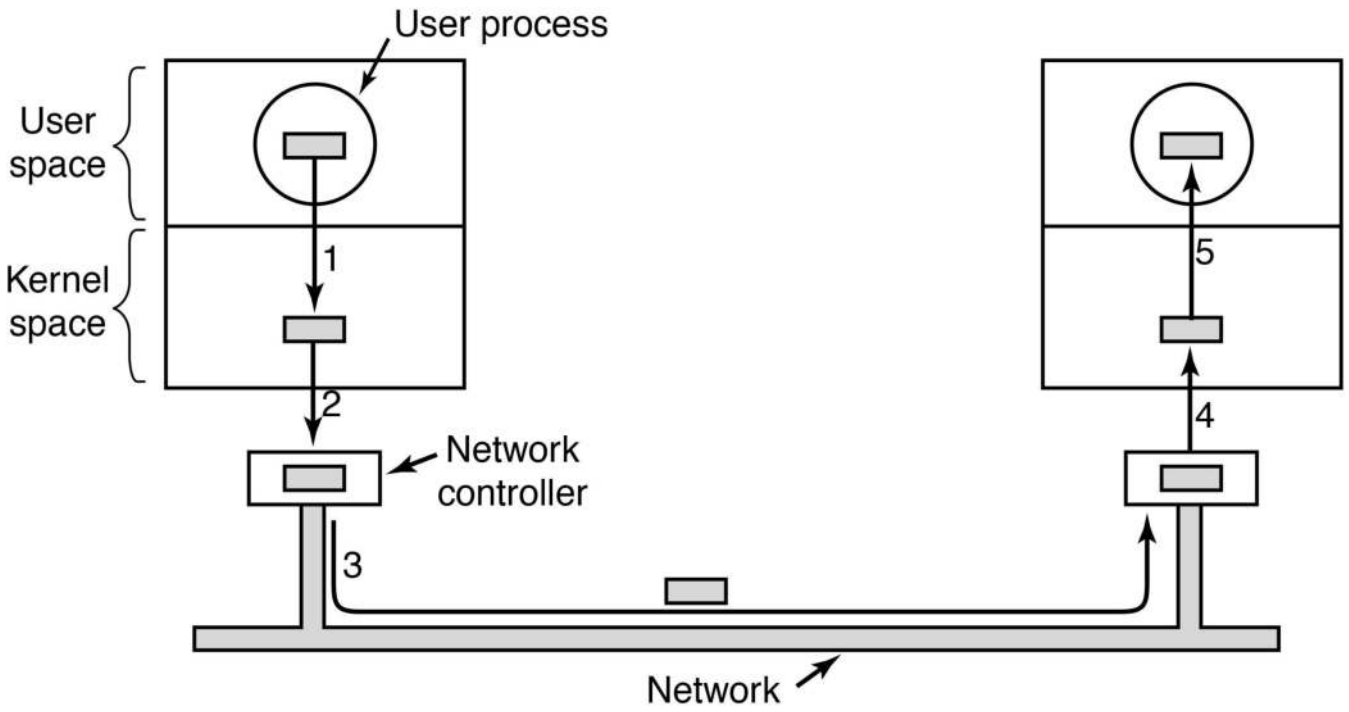


Figure 5-16. Networking may involve many copies of a packet.

User-Level I/O Software

- Offers a uniform interface to user to access devices
- C standard I/O library
  - `fopen`, `fclose`, `fflush`, `fprintf`, etc
- Safely multiplexes access to exclusive devices
  - E.g. Printing spooler

Layers of I/O Software

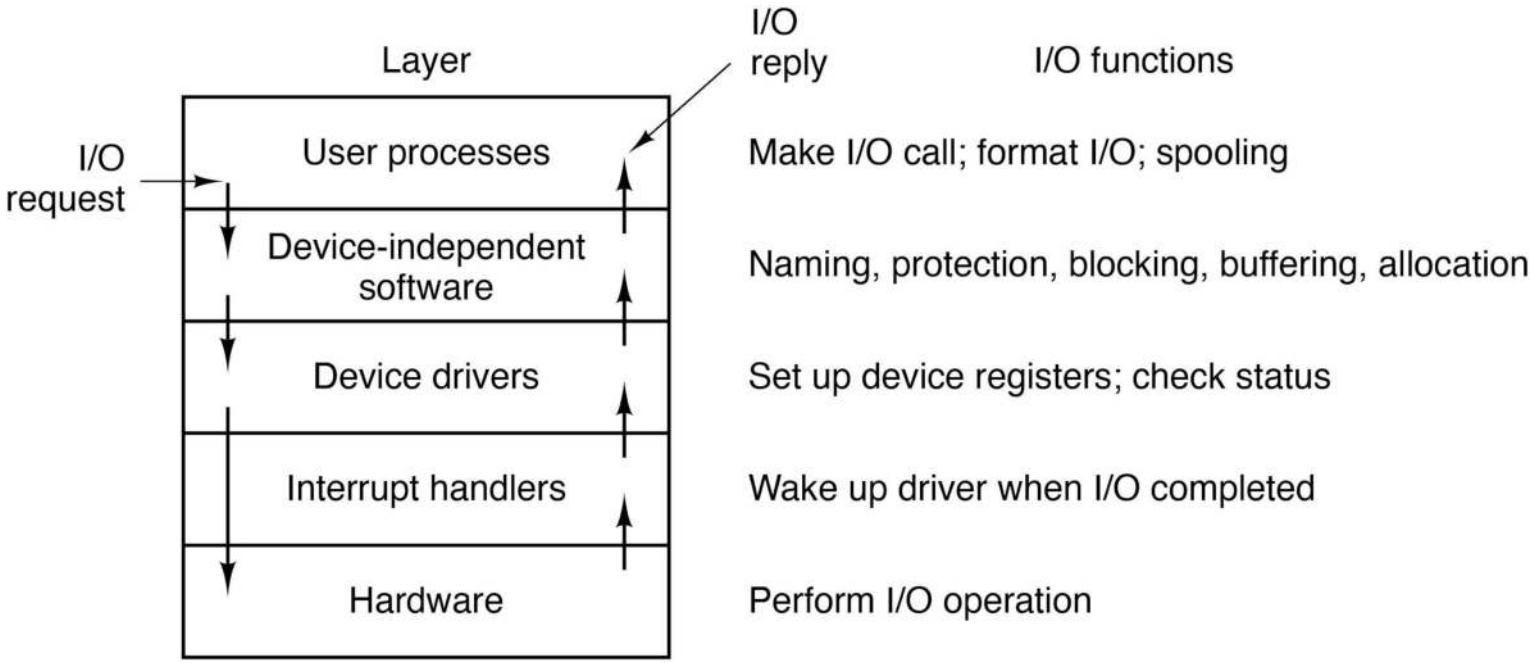


Figure 5-17. Layers of the I/O system and the main functions of each layer.

Quiz

You're given a system where 5 out of 10 devices have soft real-time requirements.

How would you design your interrupt handling logic to cater to these requirements?

Overview

- Principles of I/O Hardware
- Principles of I/O Software
- **I/O Devices**

Storage Devices: SSD vs. HDD

Vs

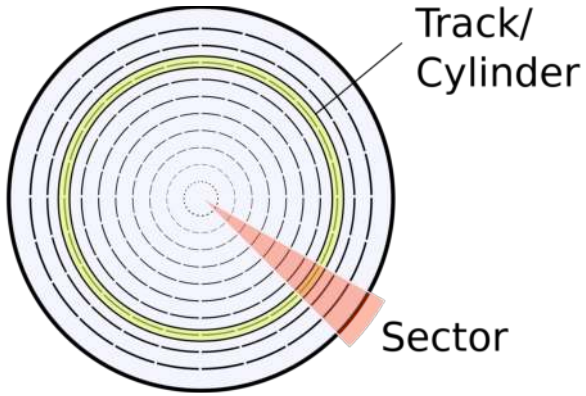
Storage Devices: SSD vs. HDD

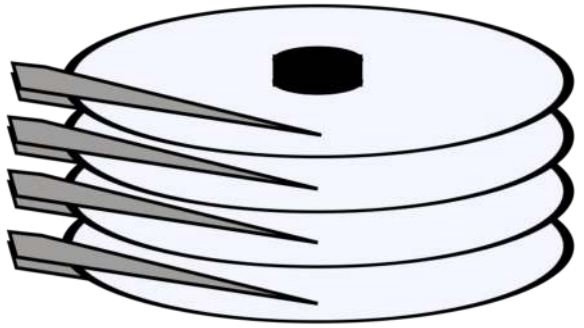
	SSD	HDD
Power	✓	✗
Heat	✓	✗
Noise	✓	✗
Vibration	✓	✗
Weight	✓	✗
Durability	✓	✗
Speed	✓	✗
Cost	✗	✓
Capacity	✗	✓

Solid-State Drive (SSD)

- DRAM or NOR/NAND flash memory based
  - No mechanical parts (only controller+mem)
  - Can be used as part of SSHDs
- NAND flash memory
  - Organized in cells (SLCs or MLCs), pages, blocks
  - Controller can read/program at page granularity
  - Controller can erase at block granularity
  - RND/SEQ access latency comparable
  - Reads more efficient than writes
  - Wear leveling to improve write endurance
  - Garbage collection to implement write operations

Hard Disk Drive (HDD)





Heads

8 Heads,
4 Platters

- Multiple **platters** and **cylinders** in a single disk with as many **tracks** as their **heads**. Each track → N **sectors**
- Addressing modes:
  - **CHS** (Cylinder-head-sector), virtual/physical
  - **LBA** (Logical Block Addressing), virtual

Hard Disk Geometry

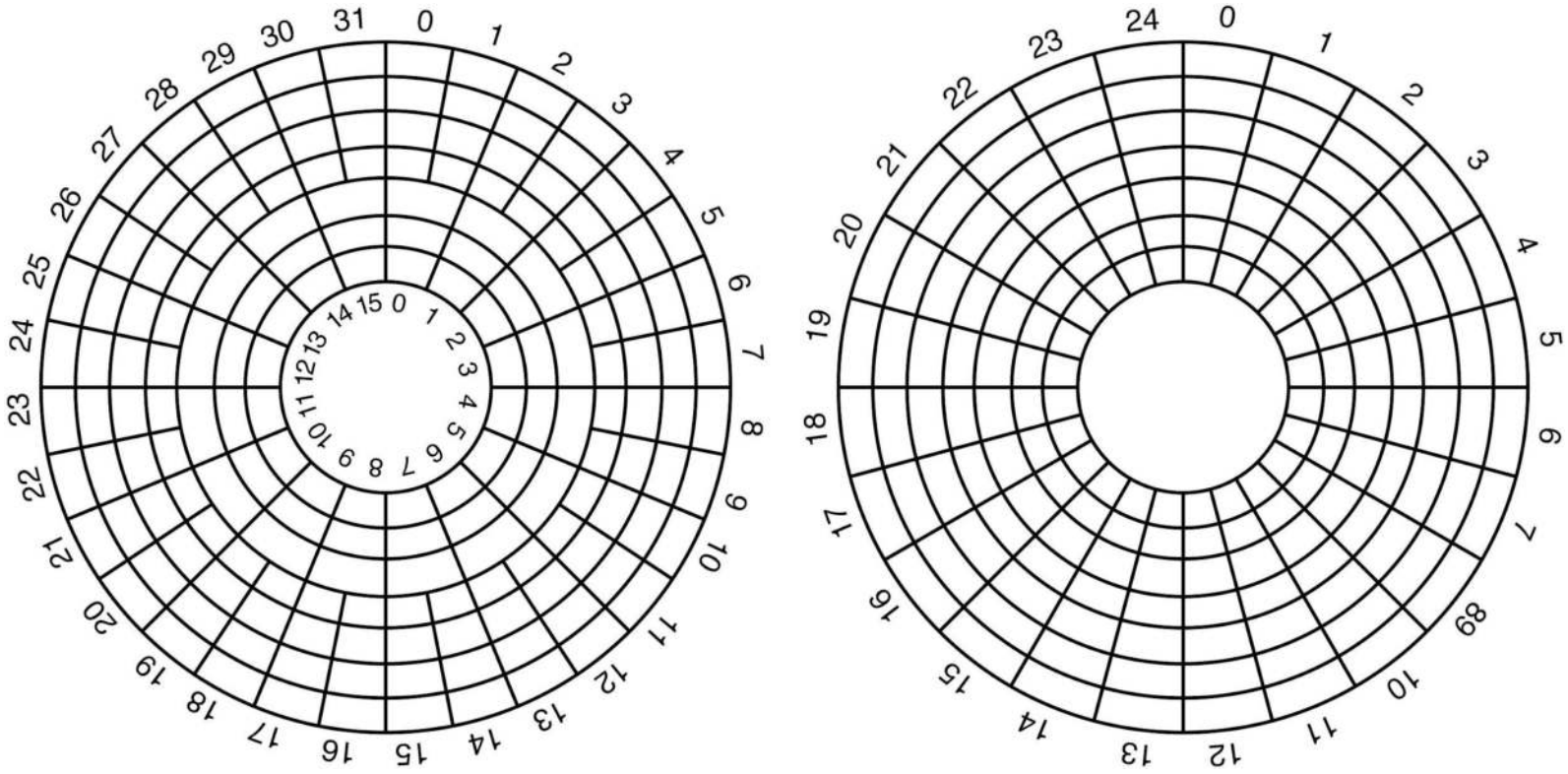
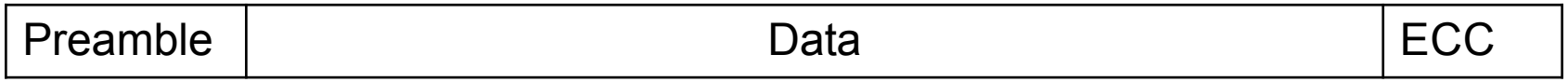


Figure 5-18. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

Disk Formatting (1 of 3)

Figure 5-19. A disk sector.



Disk Formatting (2 of 3)

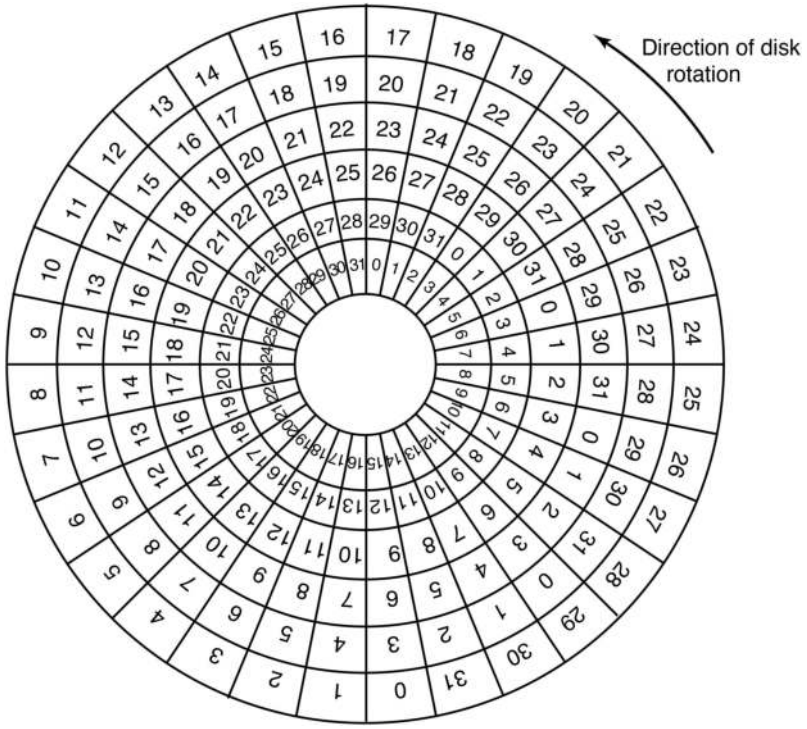


Figure 5-20. An illustration of cylinder skew.

Disk Formatting (3 of 3)

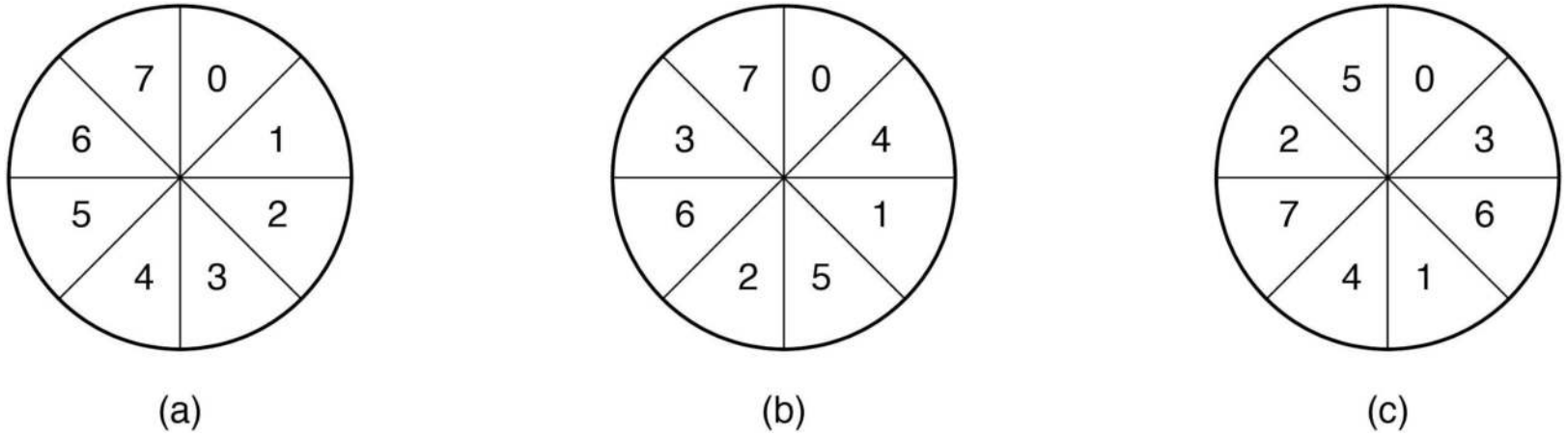


Figure 5-21. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

Hard Disk: Block Read/Write Time

Factors determining disk block read/write time:

1. Seek time
   - The time to move the arm to the proper cylinder
2. Rotational delay
   - The time for the sector to come under the head
3. Data transfer time
   - The time to transfer a single block

- Which one dominates the read/write time?
- How do we minimize it and where?

Disk Arm Scheduling Algorithms (1 of 2)

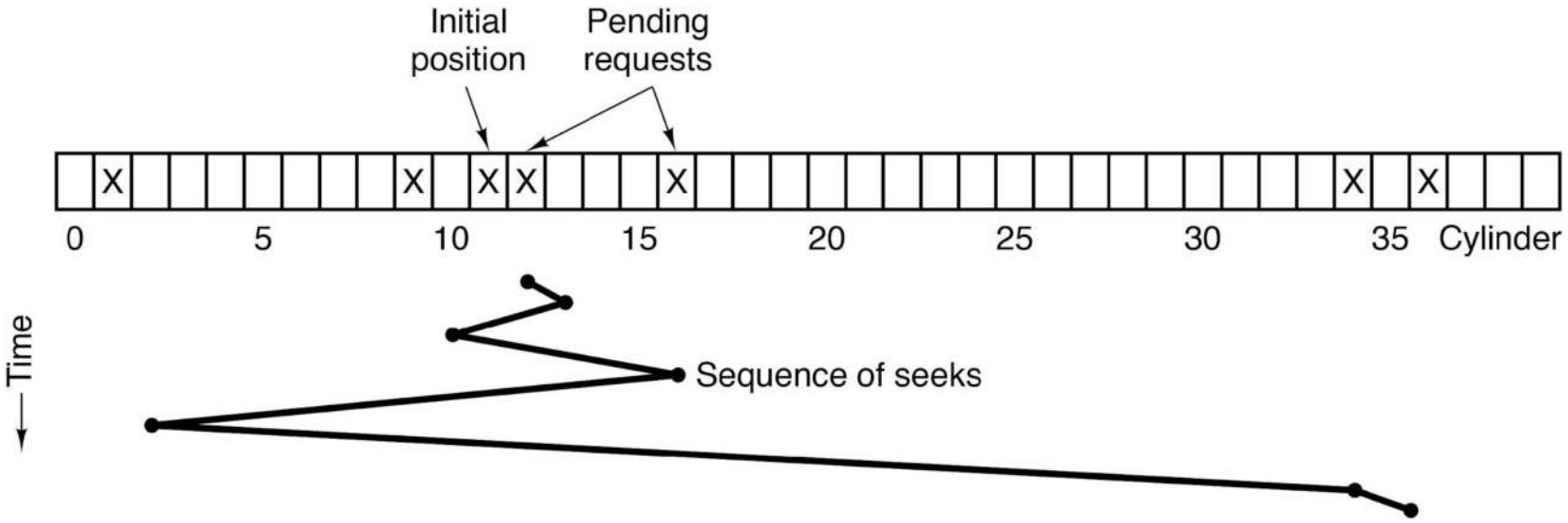


Figure 5-22. Shortest Seek First (SSF) disk scheduling algorithm.

Disk Arm Scheduling Algorithms (2 of 2)

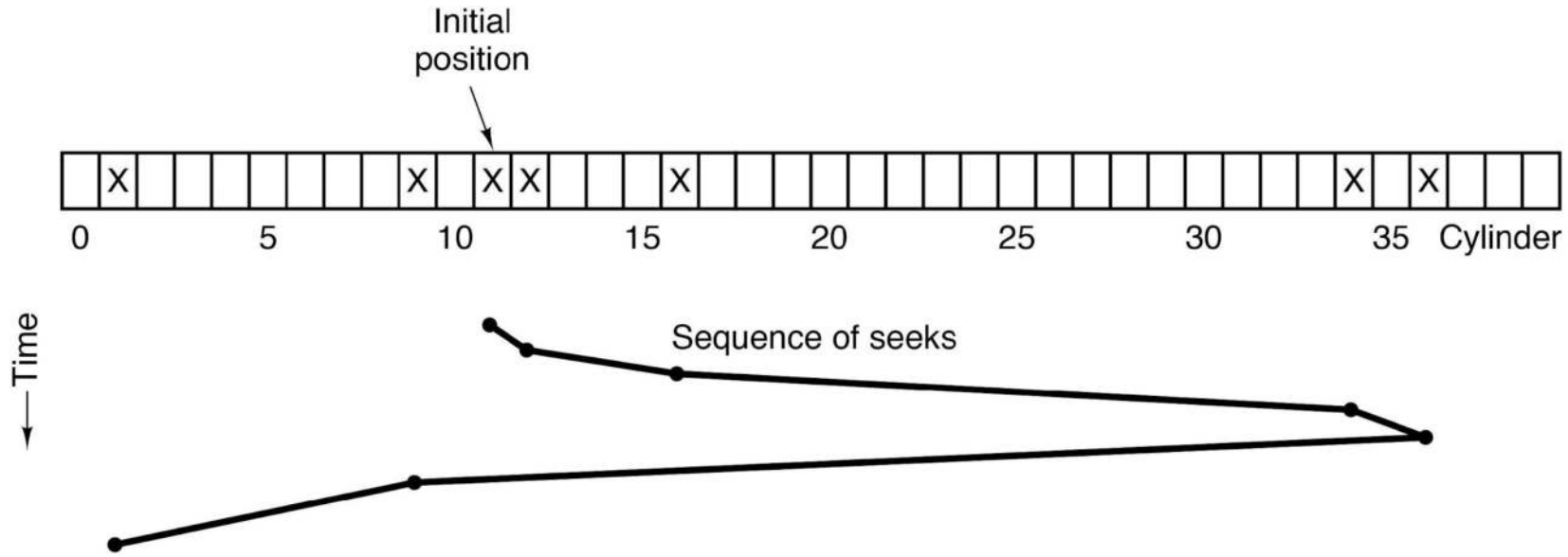


Figure 5-23. The elevator algorithm for scheduling disk requests.

Hard Disk: Error Handling

- **Programming Errors**
  - e.g., request for nonexistent sector
- **Transient Errors**
  - e.g., caused by dust on the head
- **Permanent Errors**
  - e.g., disk block physically damaged
- **Seek Errors**
  - e.g., the arm was sent to cylinder 6 but it went to 7
- **Controller Errors**
  - e.g., controller refuses to accept commands

Hard Disk:
Bad Sector Remapping

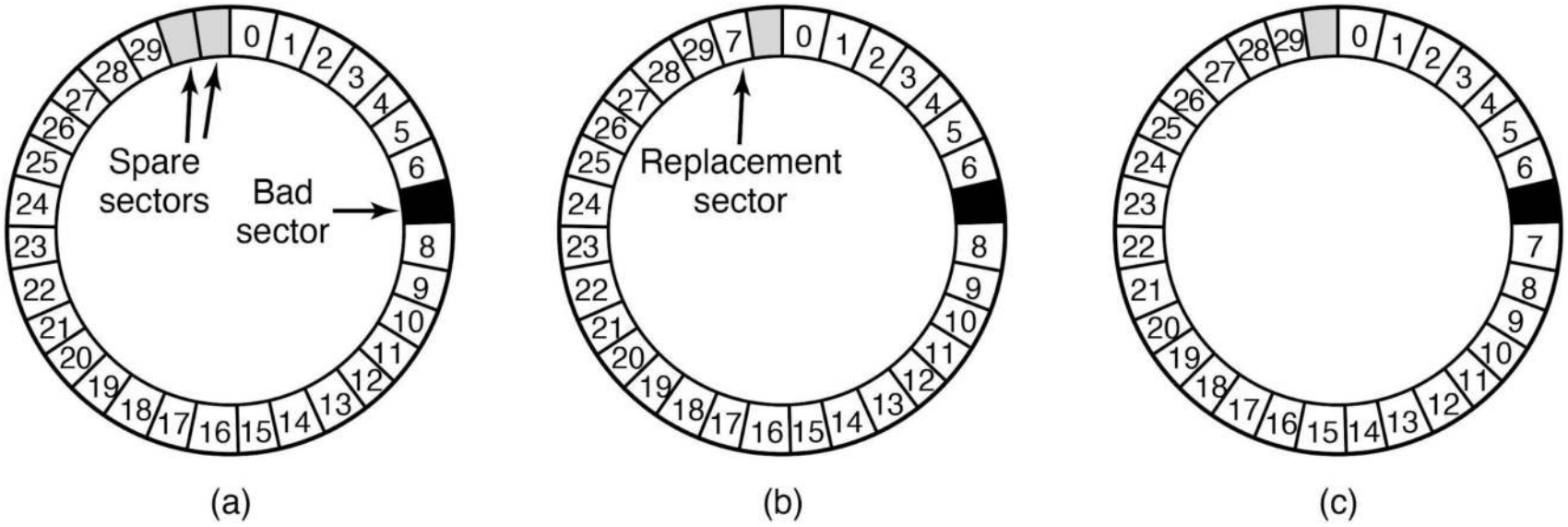


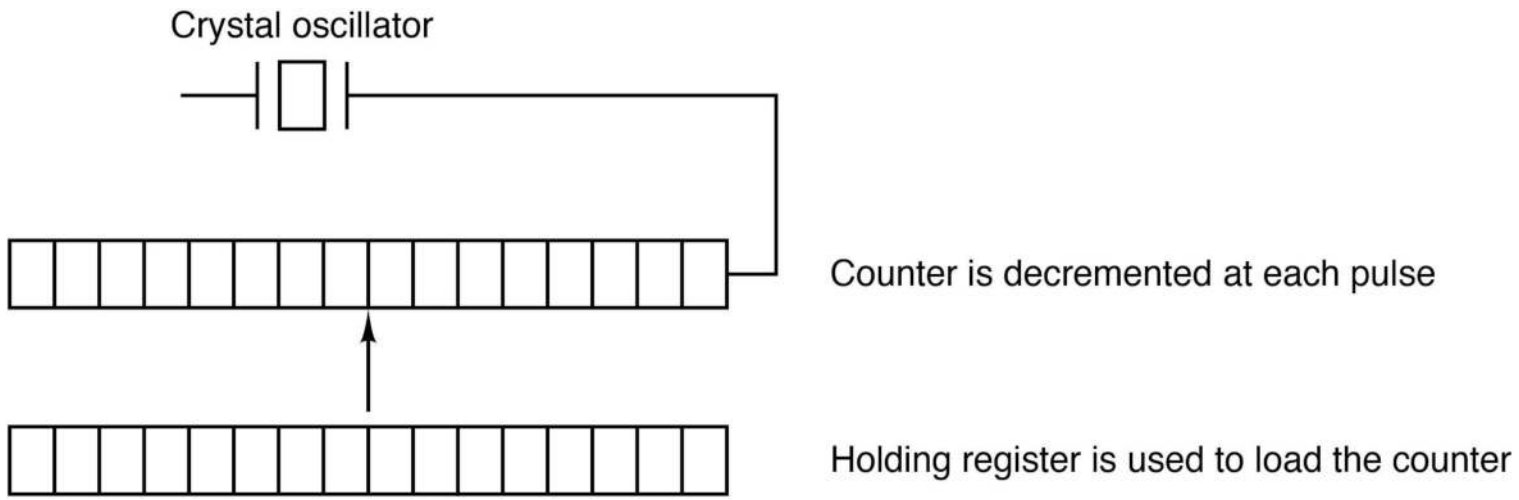
Figure 5-24. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

Clock Hardware

- Two types:
  - Simple clocks deliver hardware interrupts for each voltage cycle of the power supply (i.e., every 20 or 16.7 ms)
  - Advanced programmable clocks that have own crystal oscillator that decrements a counter in a register. If counter hits zero → HW interrupt

- Hw also provides “counters” sw can read as needed

Advanced Programmable Clock Example



- Oscillator has a frequency of 1 MHz
- The register is 16 bit
- Hence, we can set the timer between 1 and 65536 μs

Clock Software (1 of 4)

Typical duties of a clock driver:

1. Maintaining the time of day
   - System boot time (backup clock) + uptime (ticks)
2. Preventing processes from running longer than allowed
   - Decrement current process' quantum at each tick
3. Accounting for CPU usage
   - Increment current process’ cpu time at each tick

Clock Software (2 of 4)

Typical duties of a clock driver:

4. Handling alarm system call from user processes
   - Decrement alarm counter at each tick
5. Providing watchdog timers for parts of system itself
   - Generate sys notifications for synchronous alarms
6. Profiling, monitoring, statistics gathering
   - Increment specific counters at each tick

Clock Software (3 of 4)

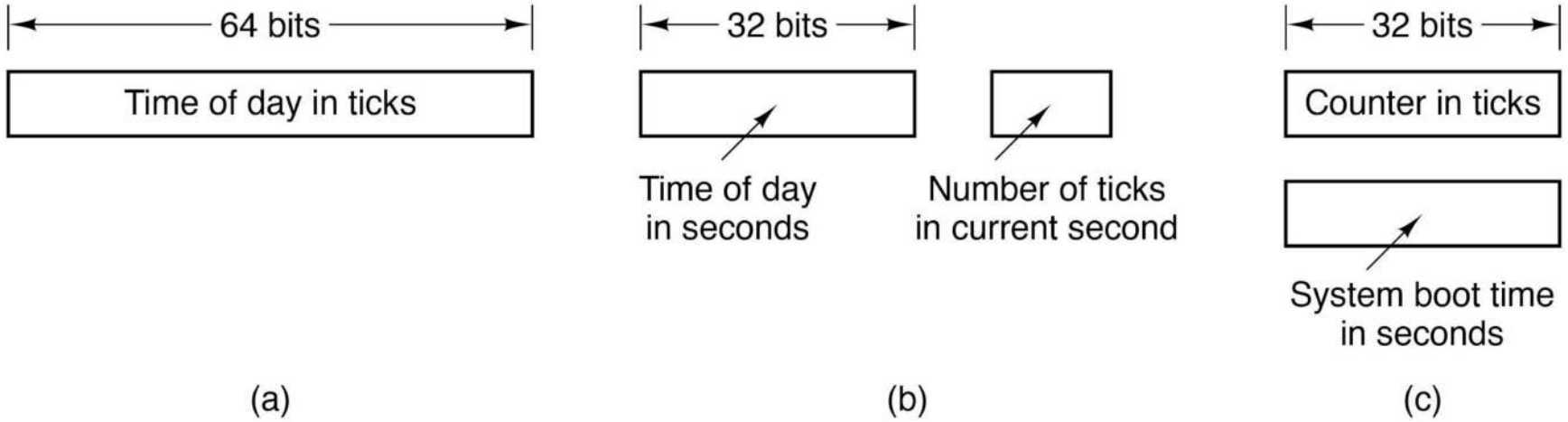


Figure 5-28. Three ways to maintain the time of day.

Clock Software (4 of 4)

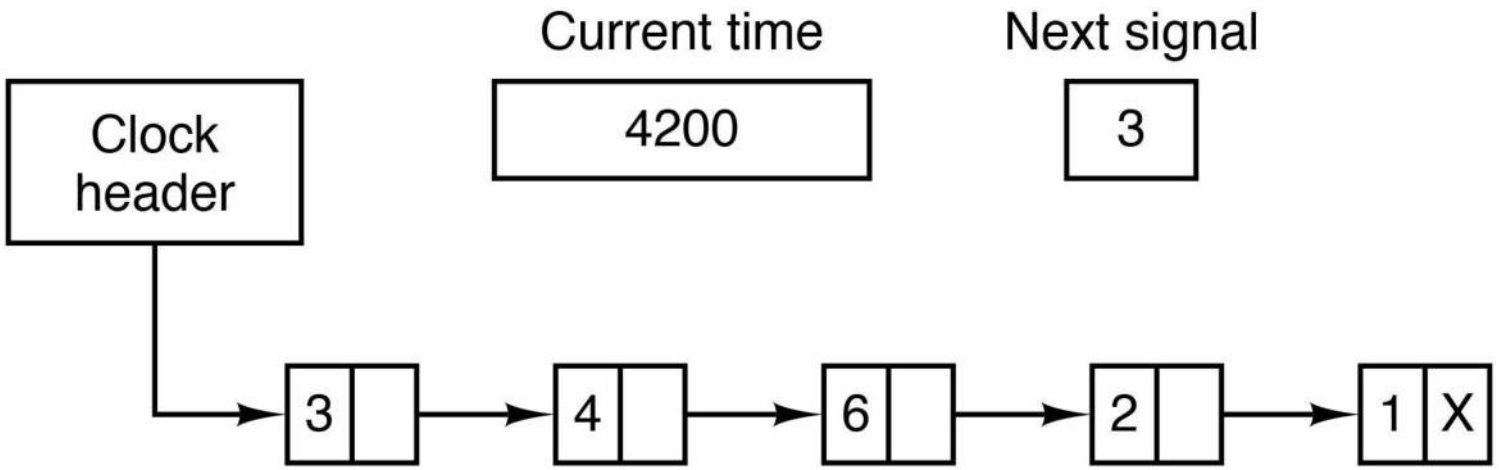


Figure 5-29. Simulating multiple timers with a single clock.

Soft Timers

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

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

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
CTRL-\	QUIT	Force core dump
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Output Software – Text Windows (1 of 2)

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.

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 1from start, 2 all)

Output Software – Text Windows (2 of 2)

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.

Escape Sequence	**Meaning**
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

The X Window System (1 of 4)

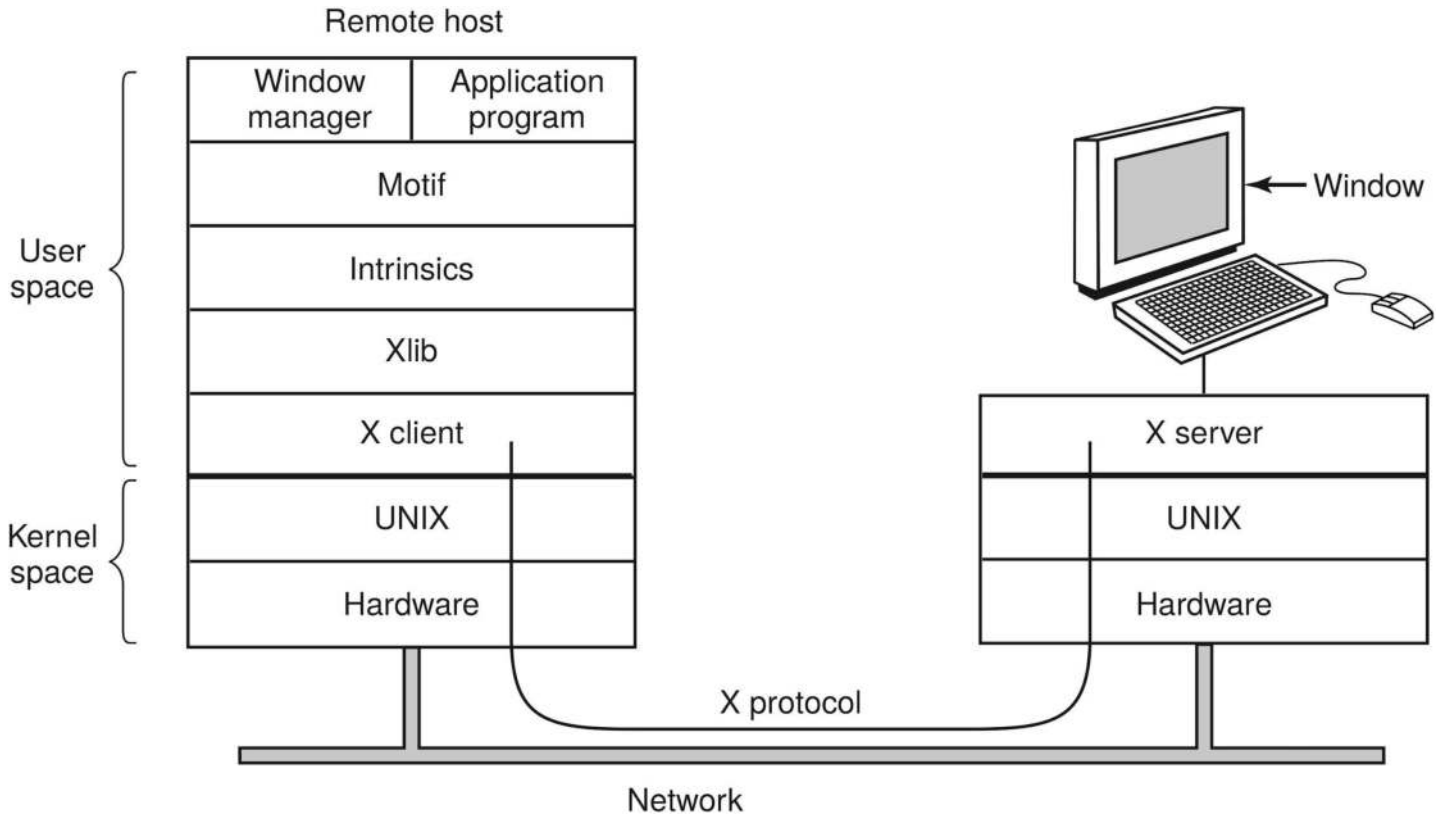


Figure 5-33. Clients and servers in the M.I.T. X Window System.

The X Window System (2 of 4)

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 of 4)

```
#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) {
                case Expose:    /* repaint window */
                        break;
                case ButtonPress:       /* possibly start resizing */
                        if (event.xbutton.button > 1) running = 0;
                        break;
                case KeyPress:
                        running = 0;
                }
        }
        XUnloadFont(disp, font_info->fid);
        XFreeGC(disp, gc);
        XCloseDisplay(disp);
}
```

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

The X Window System (4 of 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 of 4)

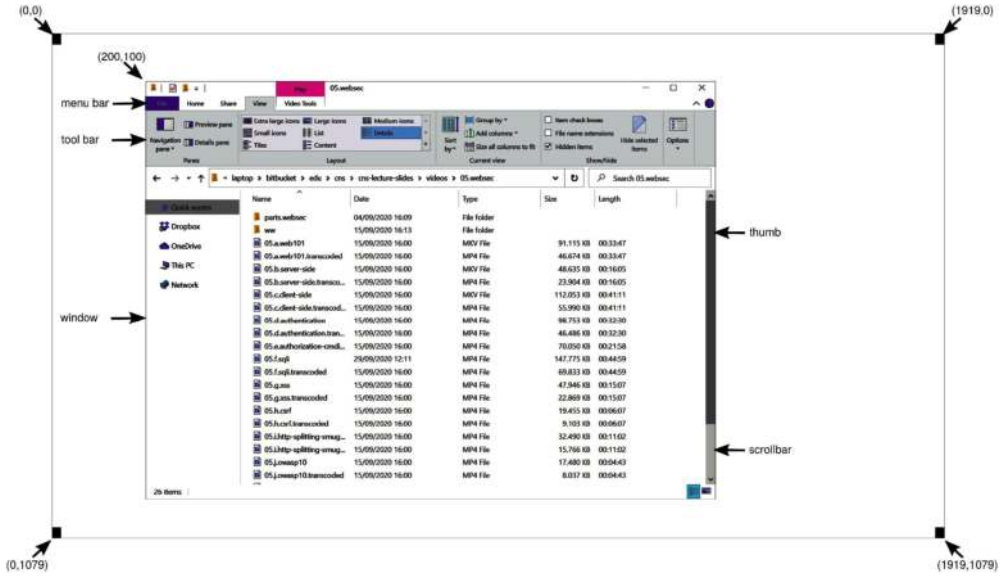


Figure 5-35. A sample window located at (200, 100) on an XGA display.

Graphical User Interfaces (2 of 4)

```
#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 of 4)

```
        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 of 4)

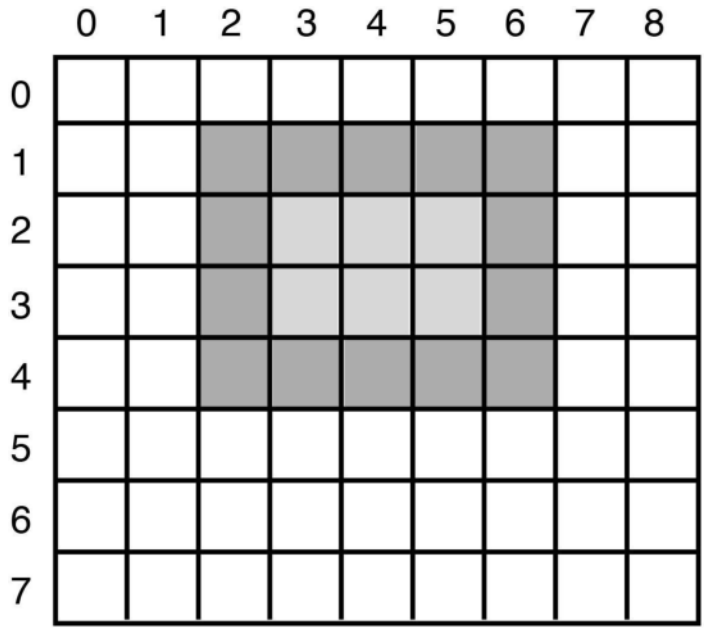


Figure 5-37. An example rectangle drawn using **Rectangle**. Each box represents one pixel.

Bitmaps

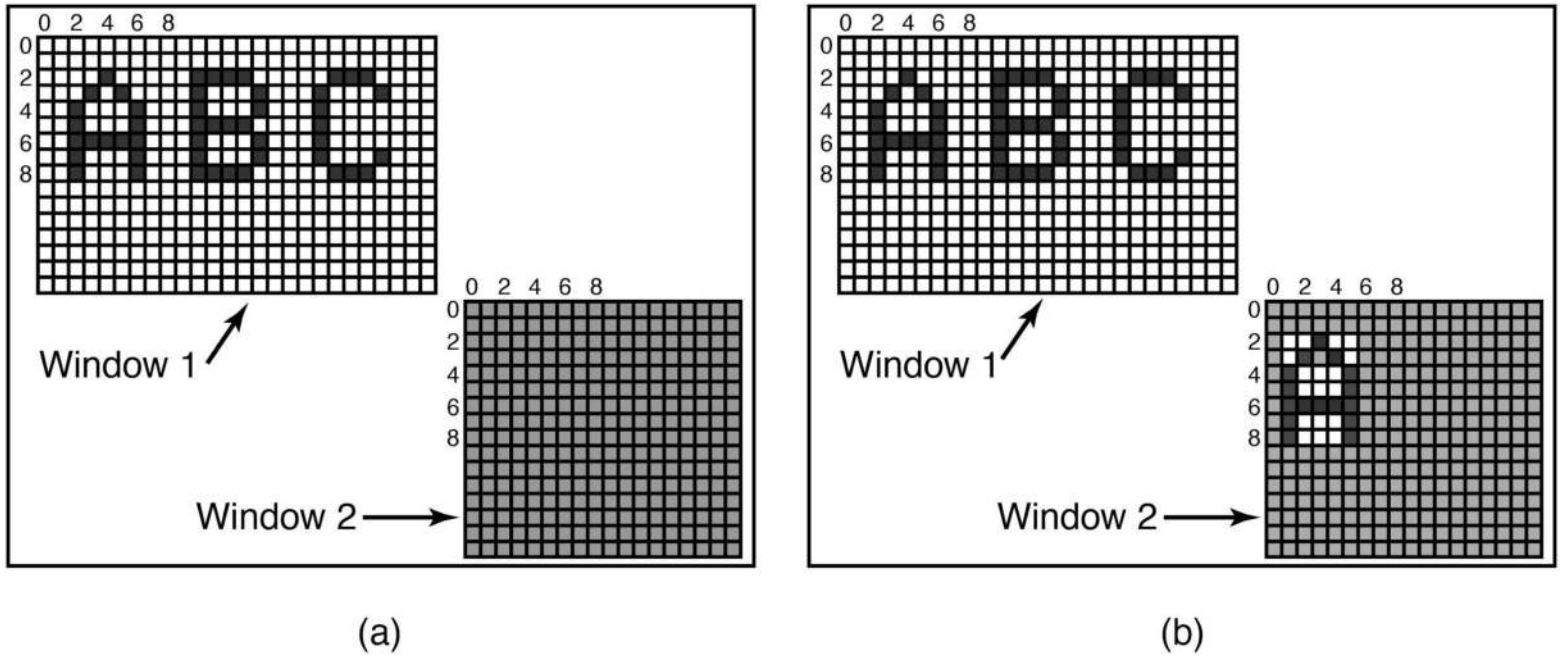


Figure 5-38. Copying bitmaps using BitBlt. (a) Before. (b) After.

Fonts

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

Hardware Issues

Power consumption of various parts of a notebook computer.

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%

Operating System Issues: The Display

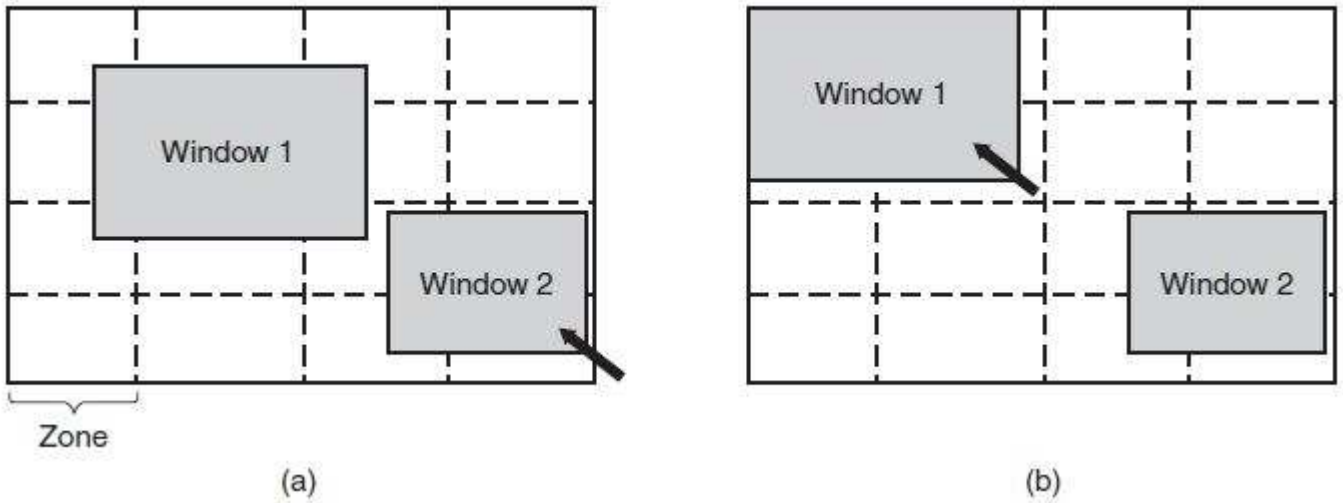


Figure 5-41. The use of zones for backlighting the display. (a) When window 2 is selected it is not moved. (b) When window 1 is selected, it moves to reduce the number of zones illuminated.

Operating System Issues: The CPU

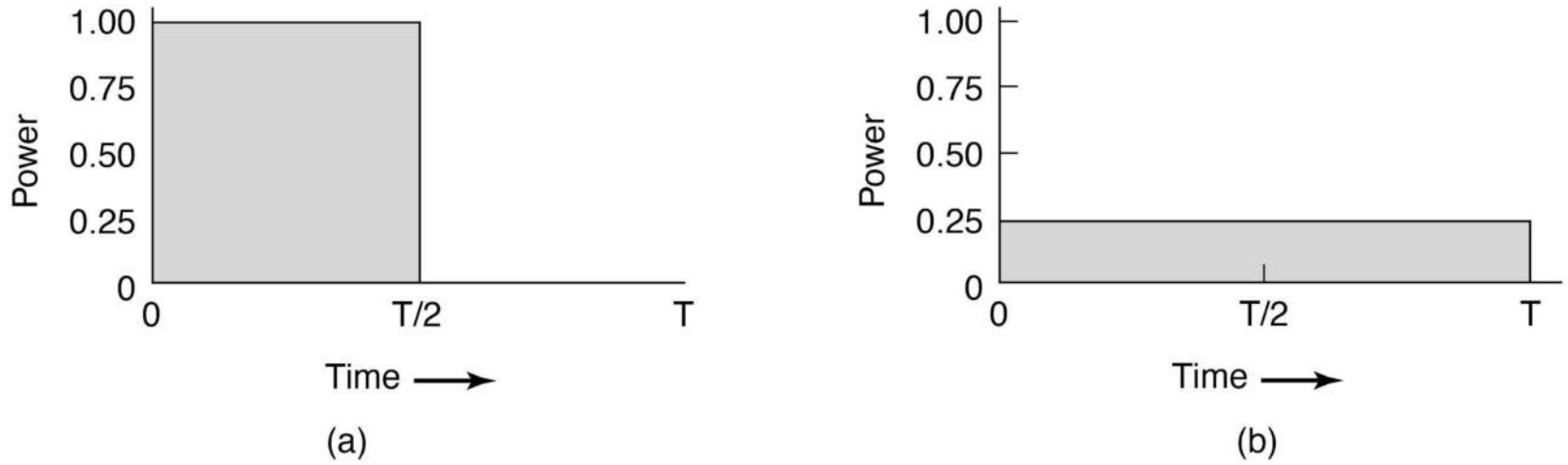


Figure 5-40. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four

Copyright