Input/Output

Chapter 5

I/O Systems

- I/O Hardware Principles
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations

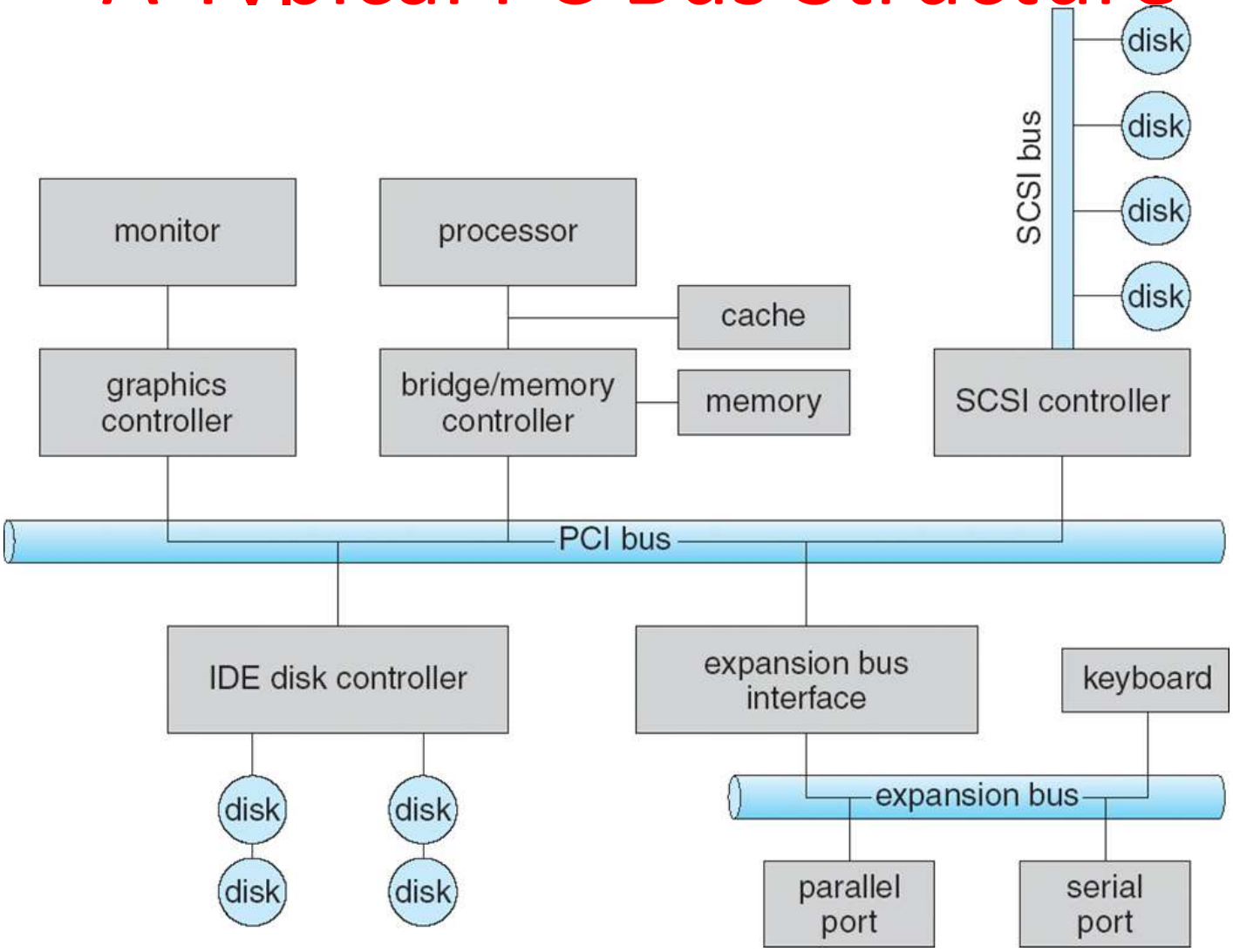
Objectives

- Discuss the principles of I/O hardware and its complexity
- Explore the structure of an operating system's I/O subsystem

I/O Hardware

- Incredible variety of I/O devices
- Common concepts
  - **Port**
  - **Bus** (**daisy chain** or shared direct access)
  - **Controller** (**host adapter**)

A Typical PC Bus Structure



Device Controllers

- Devices often consist a mechanical component and an electronic component
    - Controller or adapter is the electronic component
- Controller Card usually has a connector to plug in the device
- Controllers job is to convert the serial bit stream into a block of bytes which the operating system then formats into a user buffer and back to the device
    - Reads bytes from memory generating the signals to modify the polarization of the back light for the screen.

Device Controllers

- The interface between the controller and device is often at a very low level. Example: for a disk with 512 byte sectors, what comes off the device is a bit stream
  - Preamble
    - Written when the disk is formatted
    - Contains the cylinder-sector number, sector size, other meta/synch data
  - 4096 bits of the sector data
  - Error Correcting Code (ECC)

I/O Hardware

- I/O Controller typically has 4 registers to communicate with the CPU
  - Data-in register
  - Data-out register
  - Status register
  - Command register

- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped** I/O

Direct I/O Port Locations on PCs (partial)

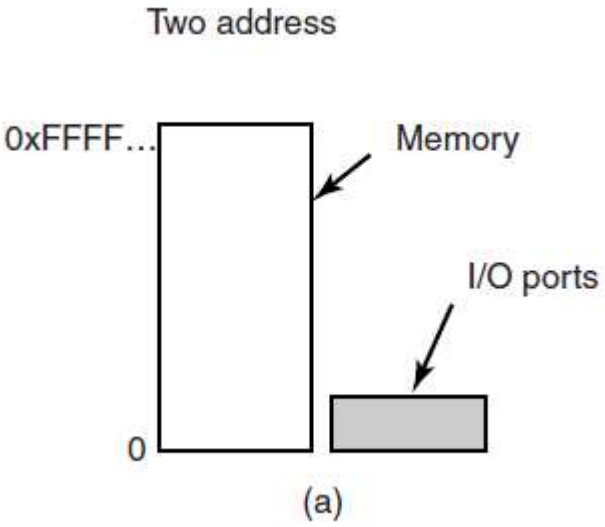
Each control register is assigned an I/O port number which is protected.

Using a special I/O instruction the OS can read and write in control register:

IN REG,PORT

OUT PORT, REG

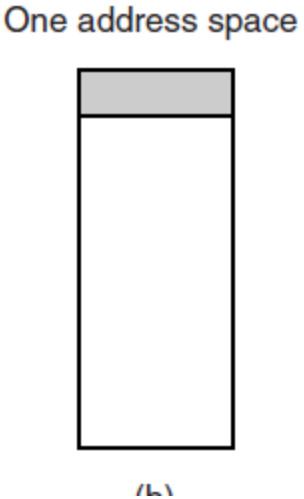
I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)



Memory-Mapped I/O

Map all control registers into the member space

- When the CPU wants to read a word, it pus the address it needs on the bus
- If it is memory or I/O the device responds to the request.



- No need for protected space
- Can share access to control registers

Memory-Mapped I/O

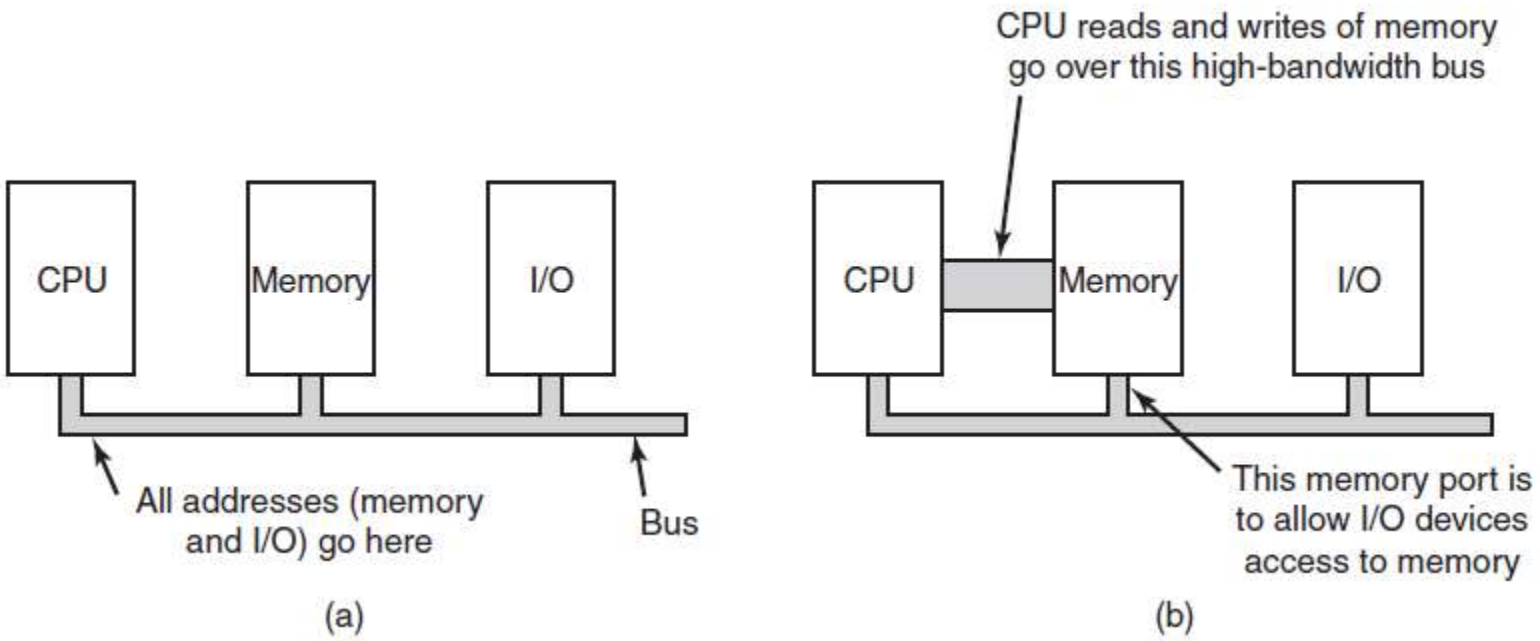


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

Direct Memory Access

- Used to avoid **programmed I/O** for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory

Direct Memory Access

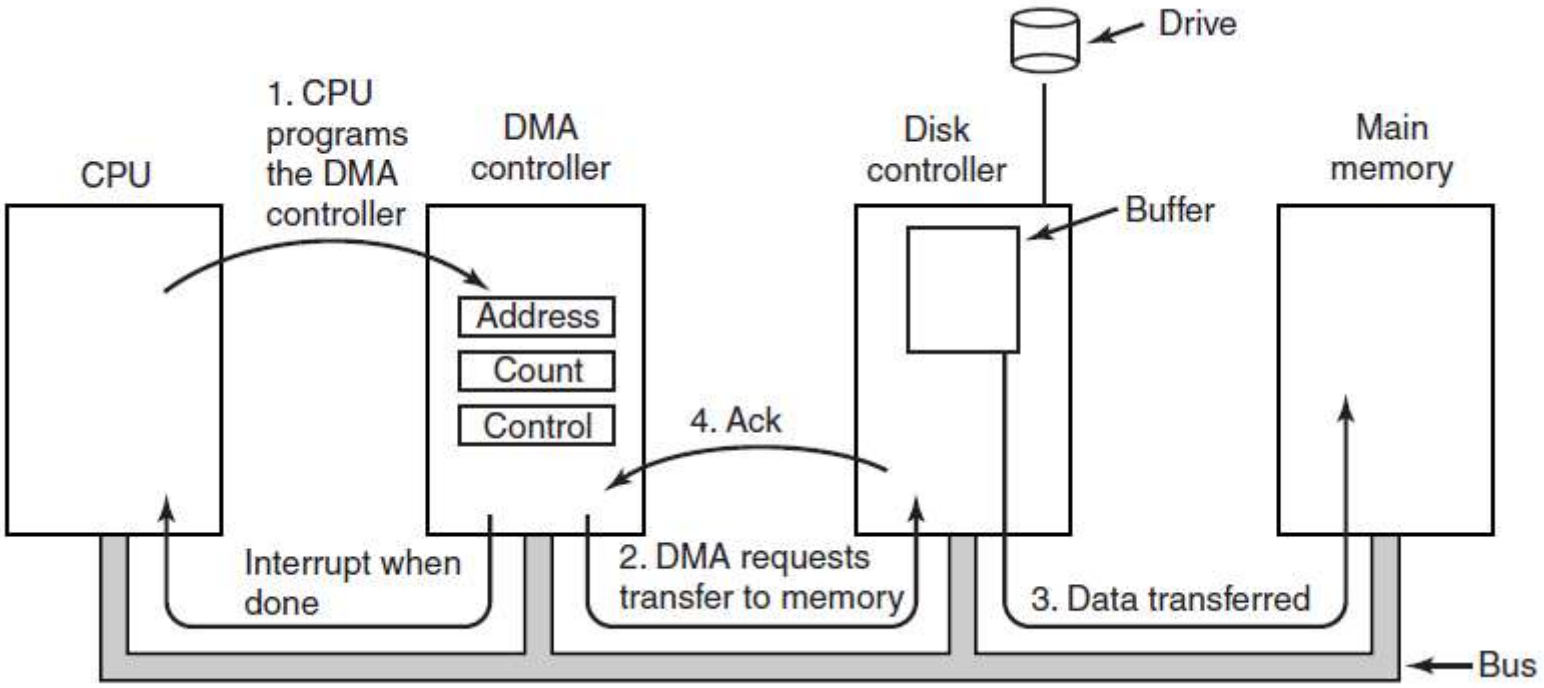
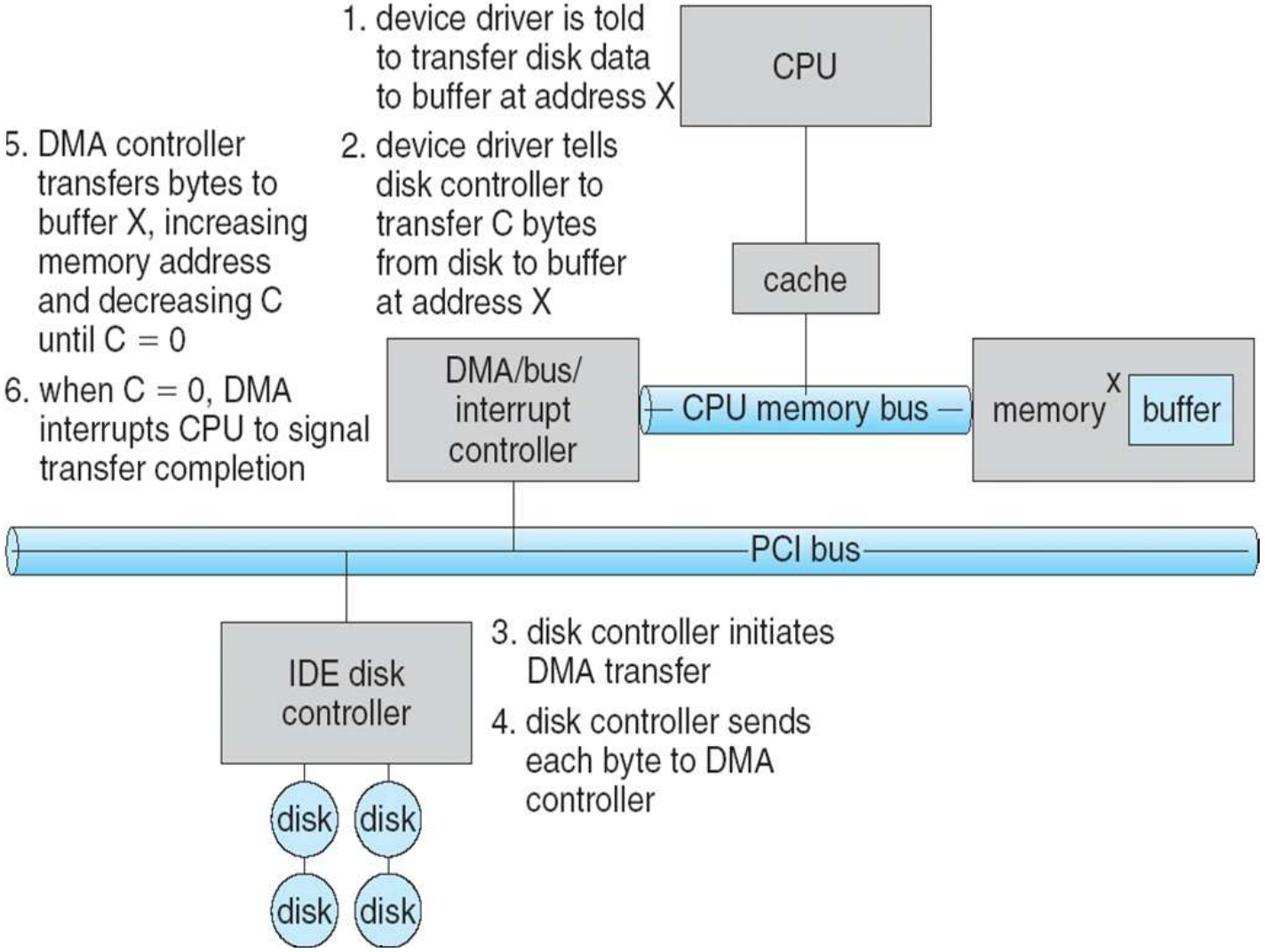


Figure 5-4. Operation of a DMA transfer.

Six Step Process to Perform DMA Transfer



Polling

- Determines state of device
  - command-ready
  - busy
  - Error
- **Busy-wait** cycle to wait for I/O from device

- Handshaking between device and OS via status register
  - Busy bit
  - Command ready bit

Interrupts

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority

Interrupts Revisited

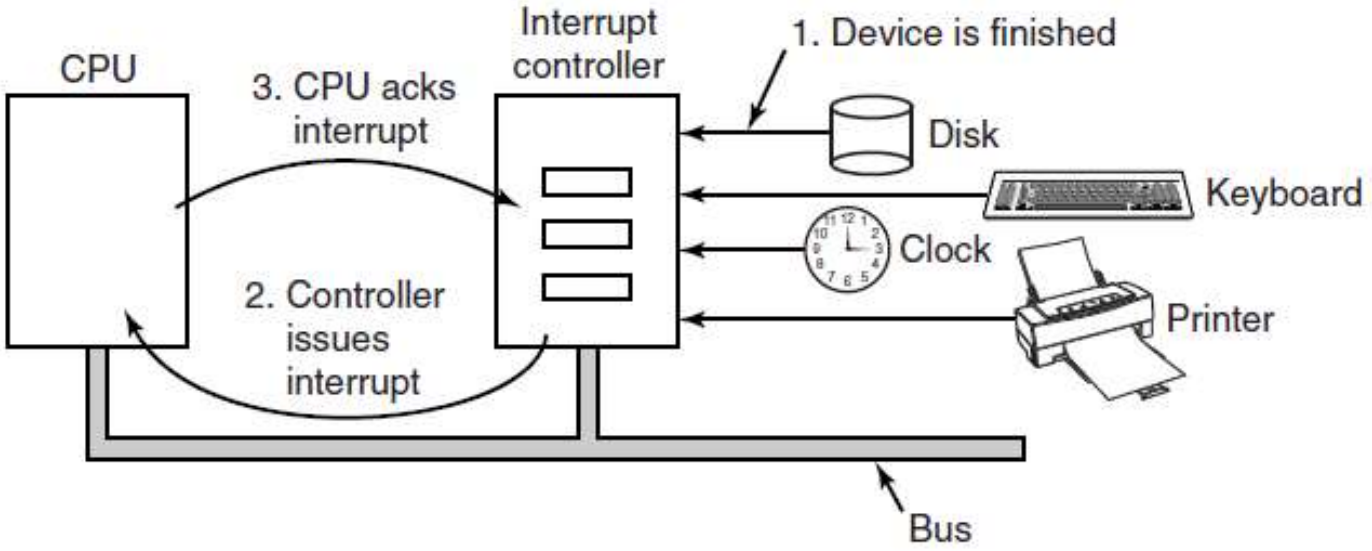
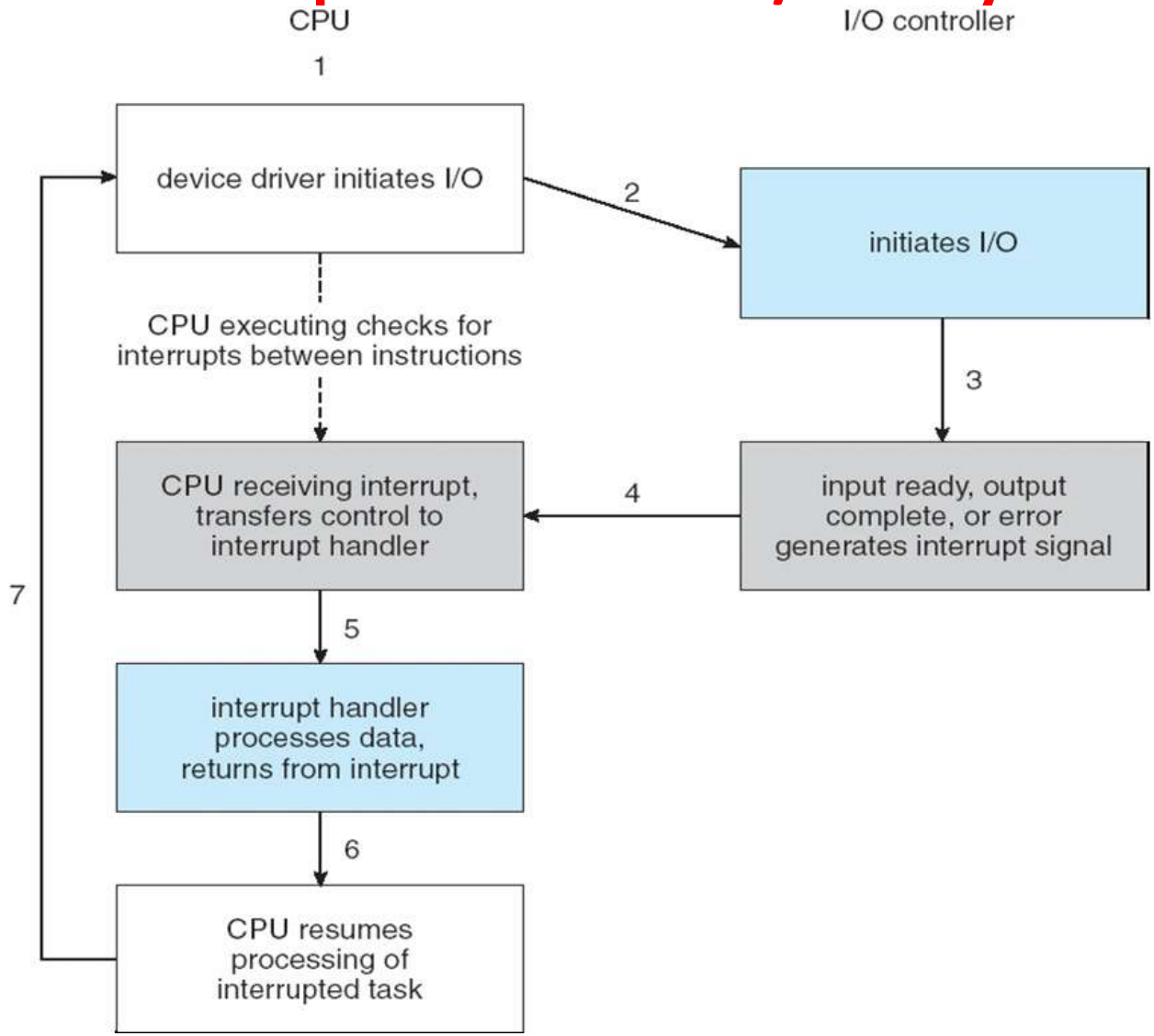


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Interrupt-Driven I/O Cycle



Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Precise Interrupt

An interrupt that leaves the machine in a well-defined state

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

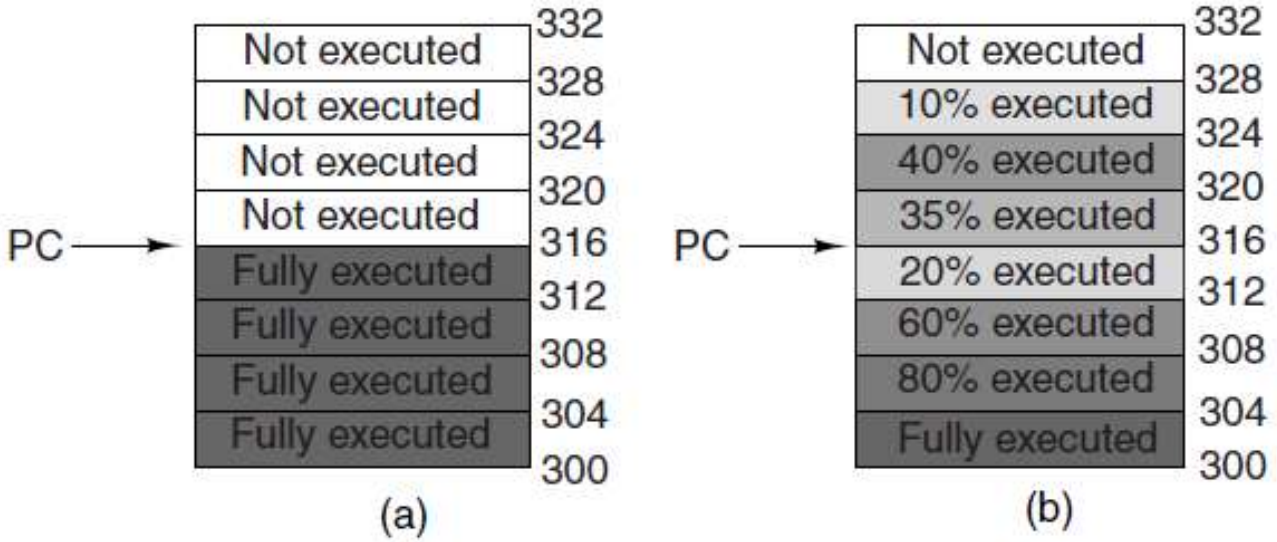


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

I/O Devices

Some typical device, network, and bus data rates.

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

I/O Interface

- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential or random-access**
  - **Sharable or dedicated**
  - **Speed of operation**
  - **read-write, read only,** or **write only**

I/O Devices

I/O Devices can be divided into two categories

- Block devices
  - Stores information in fixed-size blocks
  - Transfers are in units of entire blocks
- Character devices
  - Delivers or accepts stream of characters, without regard to block structure
  - Not addressable, does not have any *seek* operation

Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
  - Commands include `get, put`
  - Libraries layered on top allow line editing

Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9*x*/2000 include socket interface
  - Separates network protocol from network operation
  - Includes `select` functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Goals of the I/O Software

Issues:

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

Programmed I/O (1)

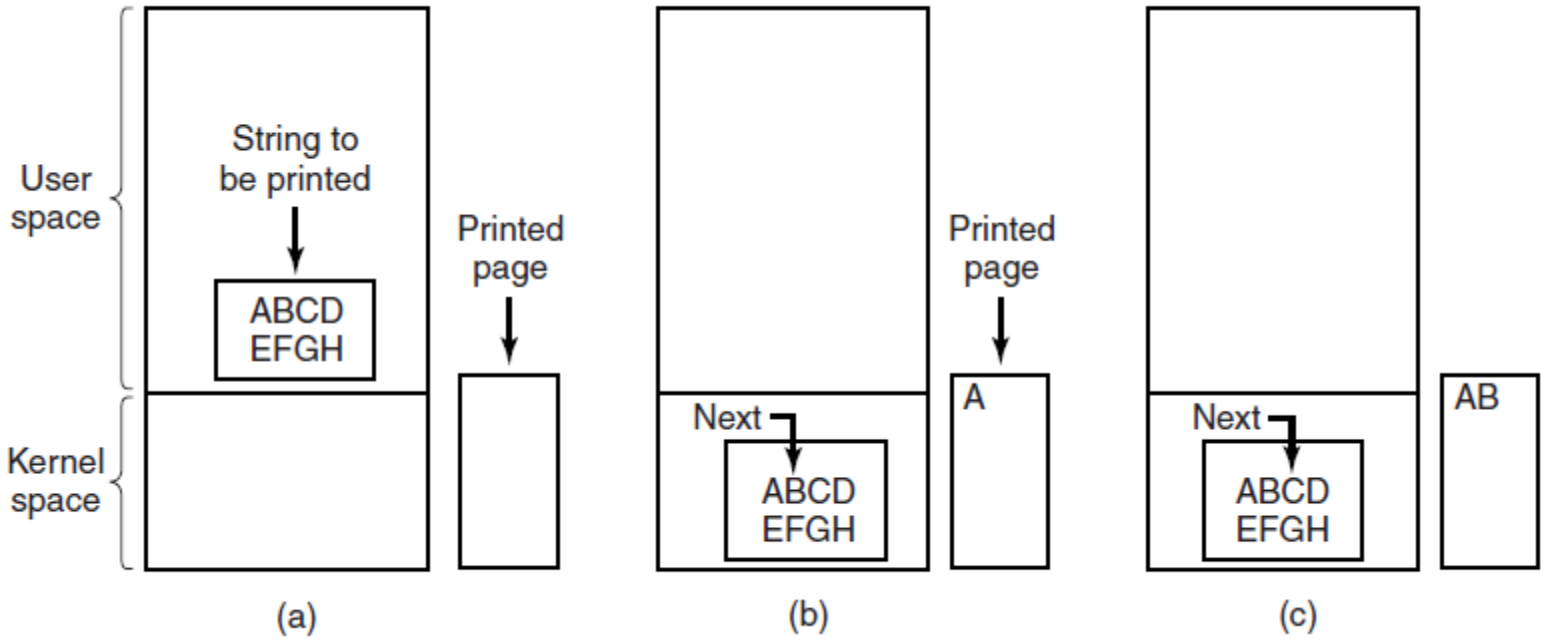


Figure 5-7. Steps in printing a string.

Programmed I/O (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

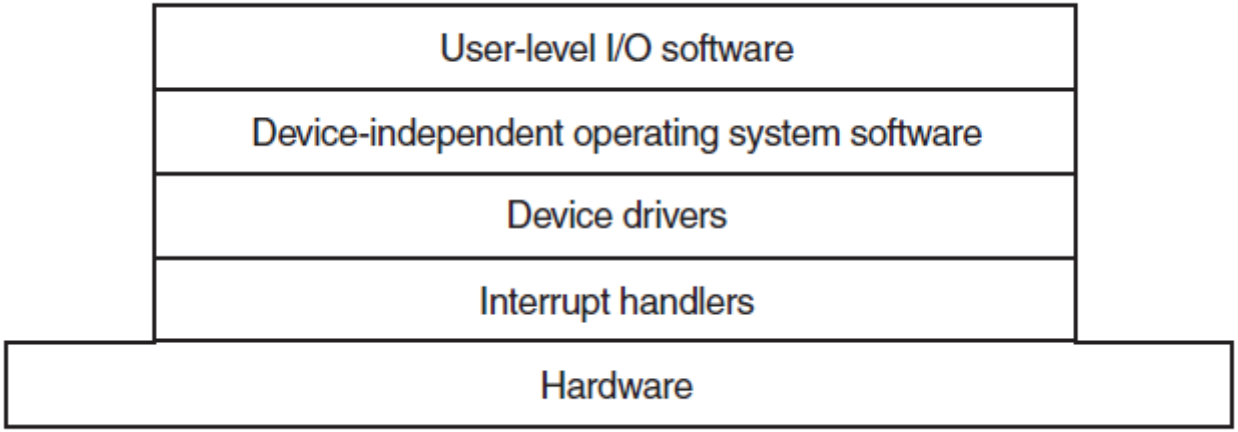


Figure 5-11. Layers of the I/O software system.

Interrupt Handlers (1)

Typical steps after hardware interrupt completes:

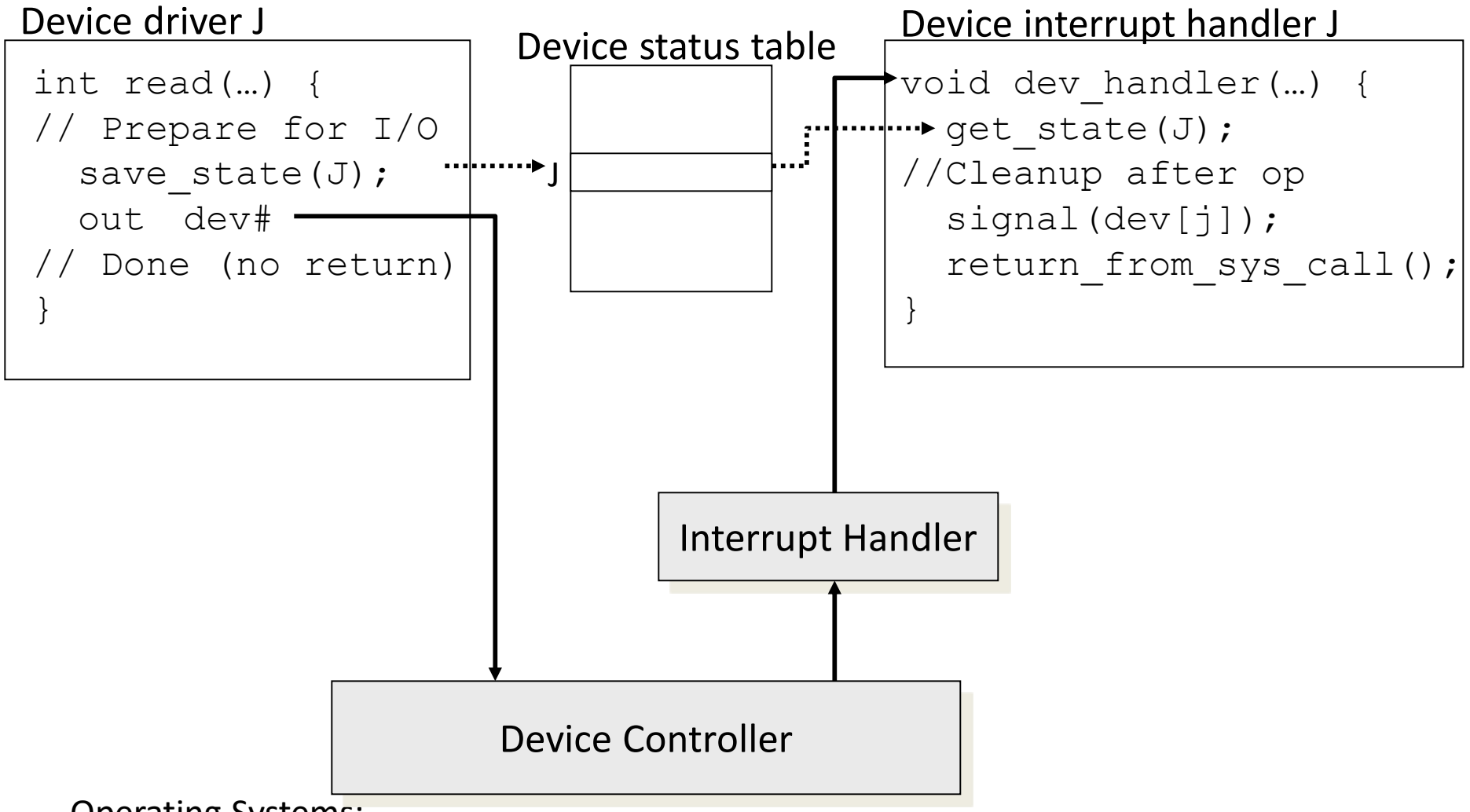
1. Save registers (including the PSW) not already saved by interrupt hardware.
2. Set up context for interrupt service procedure.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge interrupt controller. If no centralized interrupt controller, reenable interrupts.
5. Copy registers from where saved to process table.

Interrupt Handlers (2)

Typical steps after hardware interrupt completes:

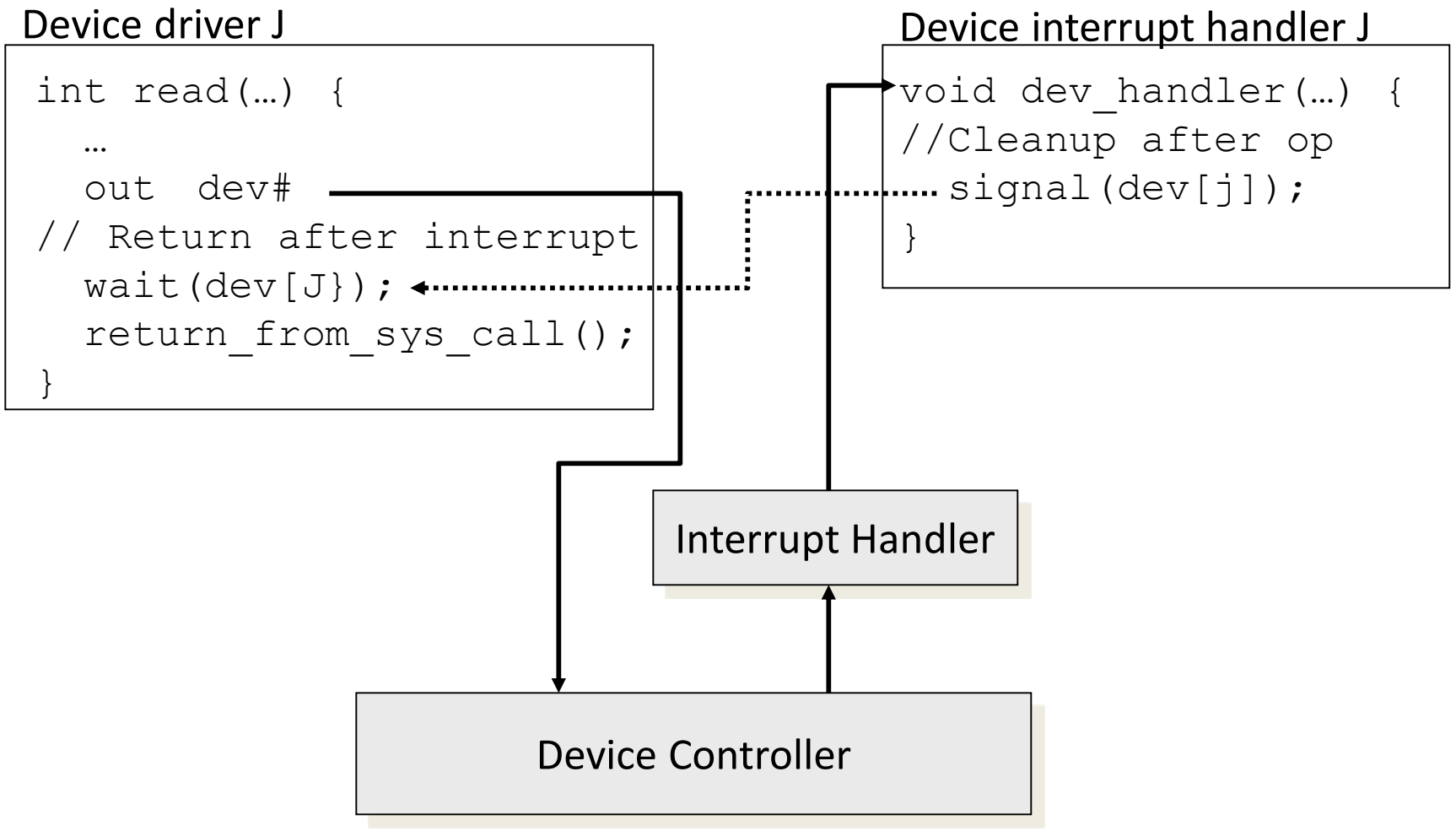
6. Run interrupt service procedure. Extract information from interrupting device controller's registers.
7. Choose which process to run next.
8. Set up the MMU context for process to run next.
9. Load new process' registers, including its PSW.
10. Start running the new process.

Handling Interrupts



Operating Systems:
A Modern
Perspective, Chapter

Handling Interrupts(2)



Operating Systems:
A Modern
Perspective, Chapter

Device-Independent I/O Software

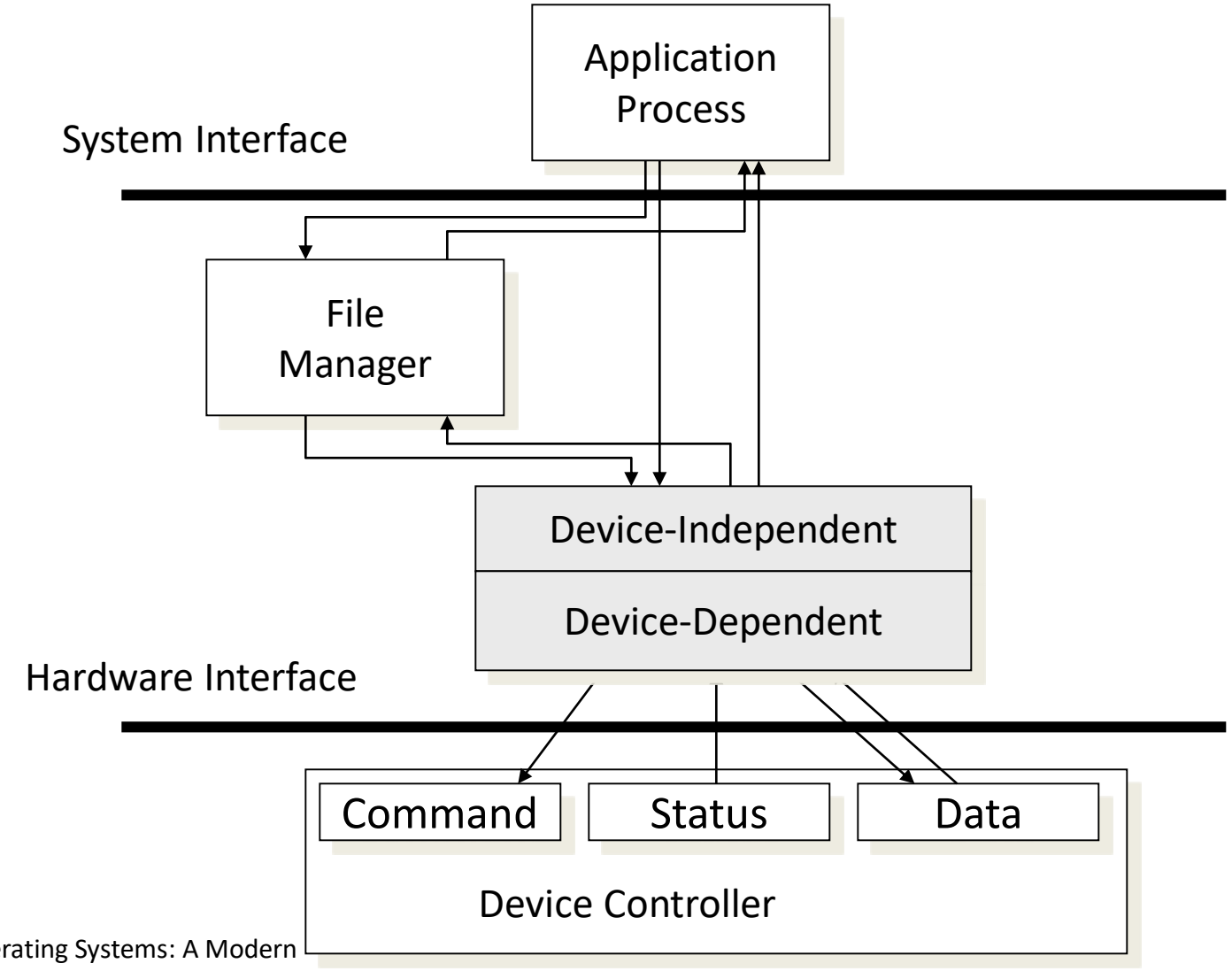
A key concept of OS design: DEVICE INDEPENDENCE

The ability for users to write programs that can access any I/O device without having to specify the device in advance.

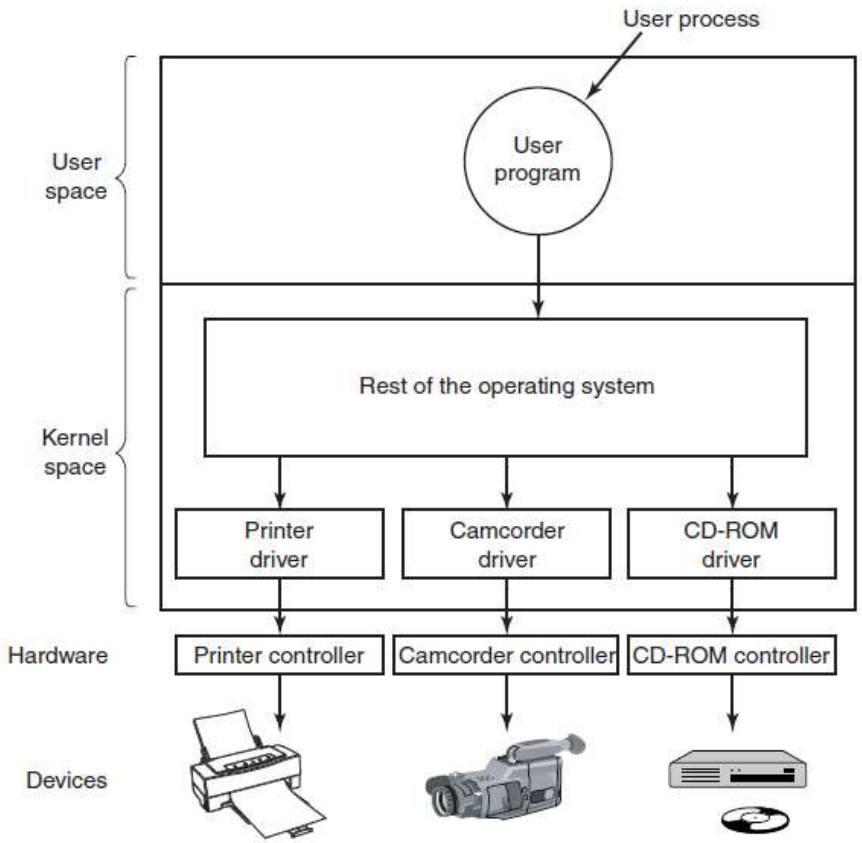
Functions of the
device-independent I/O software.

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

Device Management Organization



Device Drivers



Must be re-entrant

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

Uniform Interfacing for Device Drivers

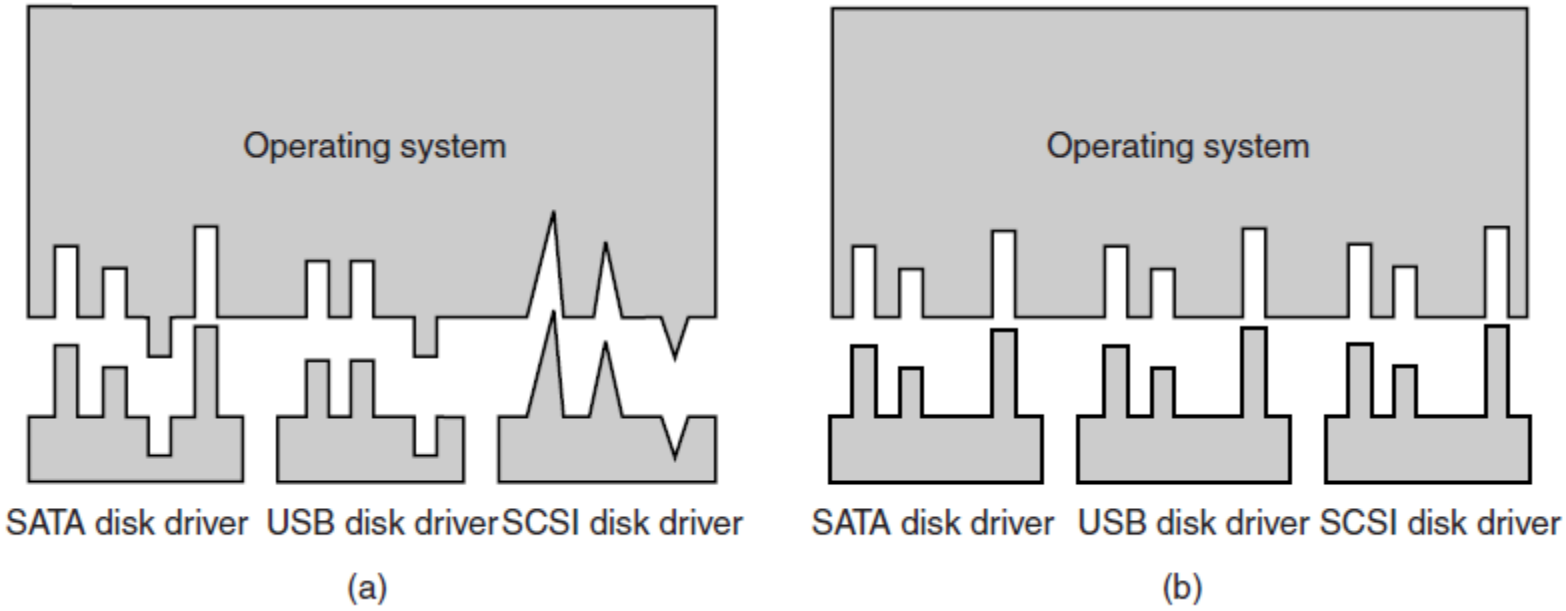
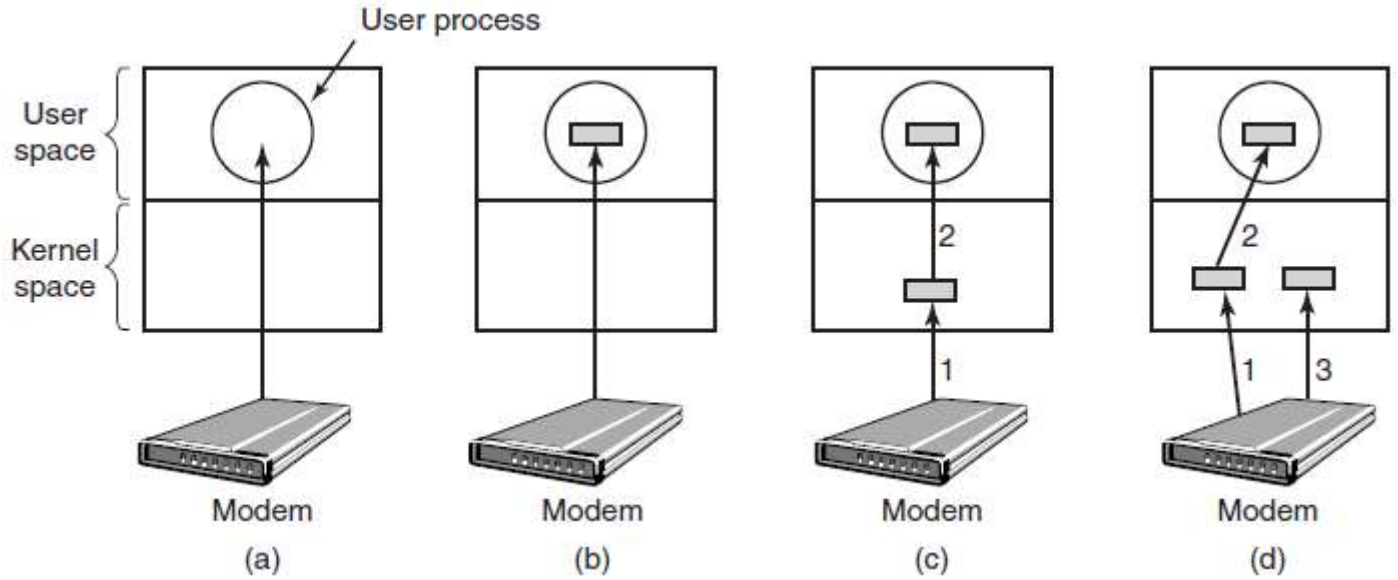


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

Buffering



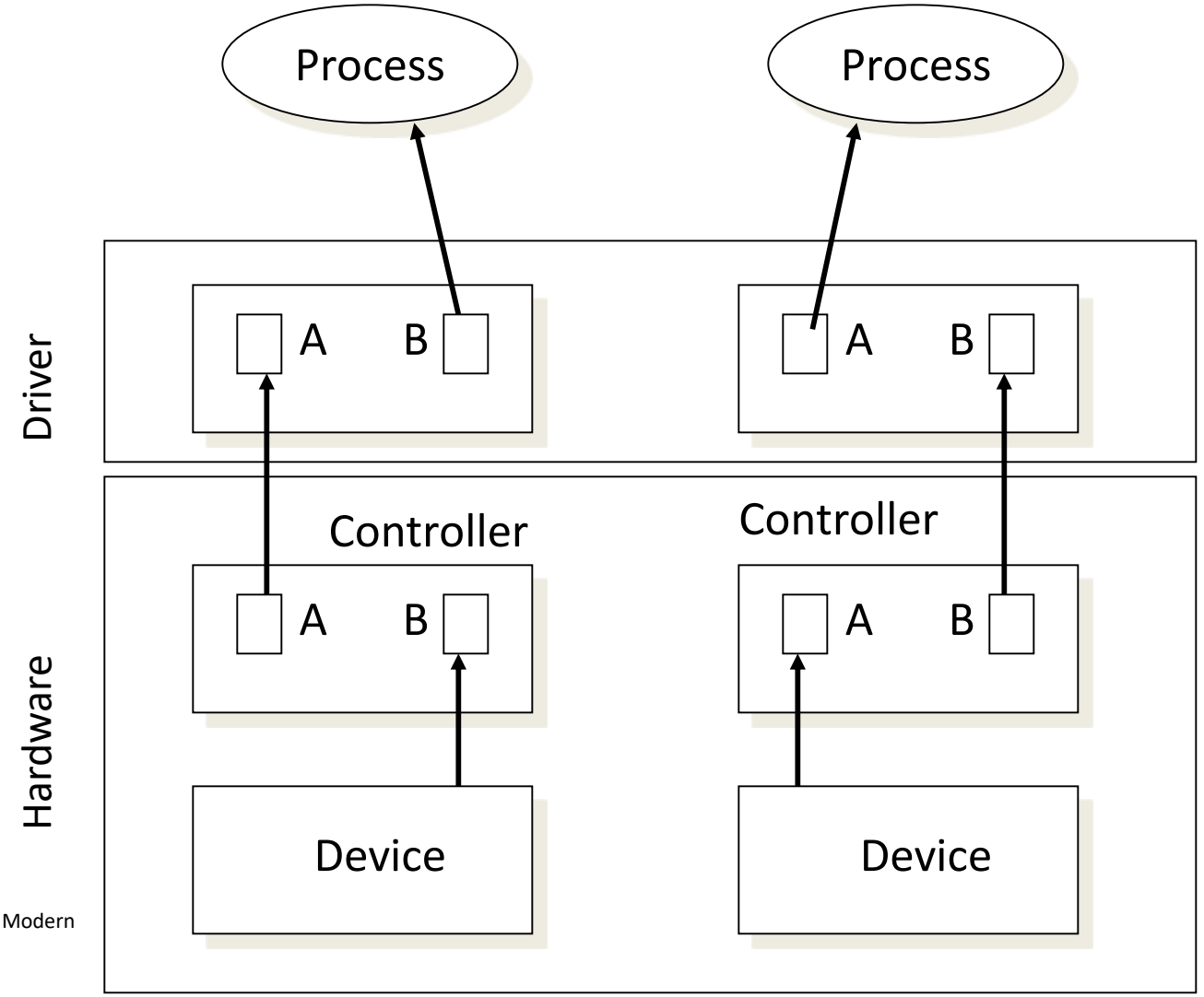
(a) Unbuffered input.

(b) Buffering in user space.

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

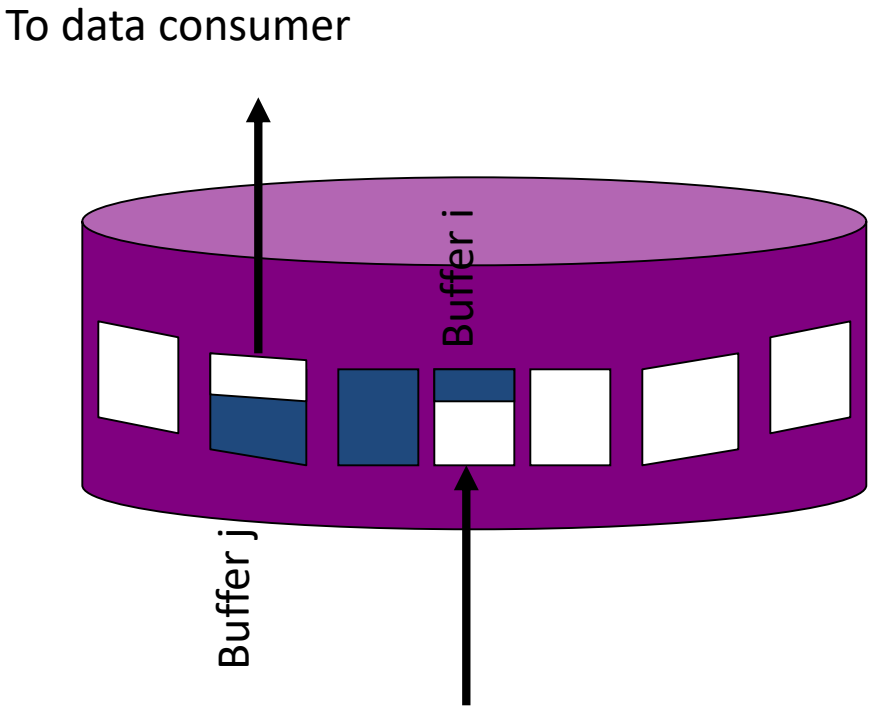
(d) Double buffering in the kernel.

Double Buffering in the Driver



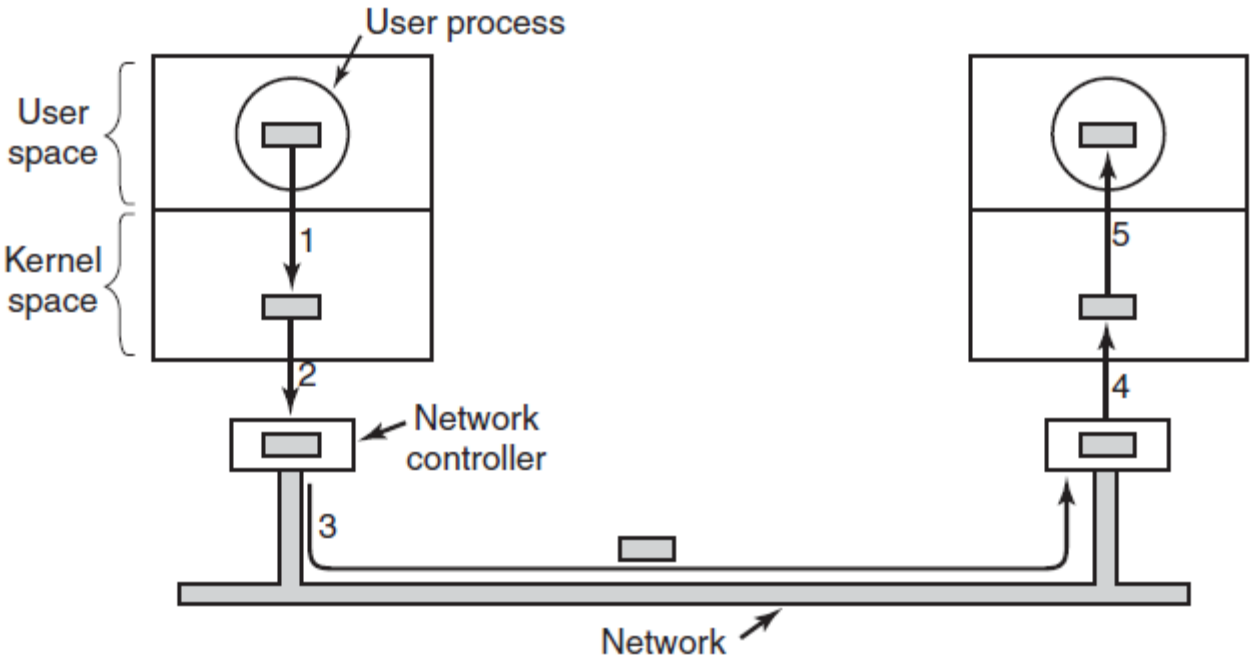
Nutt: Operating Systems: A Modern Perspective, Chapter 5

Circular Buffering



Operating Systems: A Modern Perspective, Chapter 5

Buffering



Performance issue: Networking may involve many copies of a packet.

Error Reporting

- Device specific- device driver can handle
- Framework for error handling is device independent
- Types of errors
- Programming Error-Occurs when aprocess ask for something Impossible(writing to an input device)
- I/O error-trying to write a disk block that has been damaged

Driver pass the information to the device oindepentant software

- Critical errors –such as root directory, free block list may have been destroyed

Allocating and releasing dedicated devices

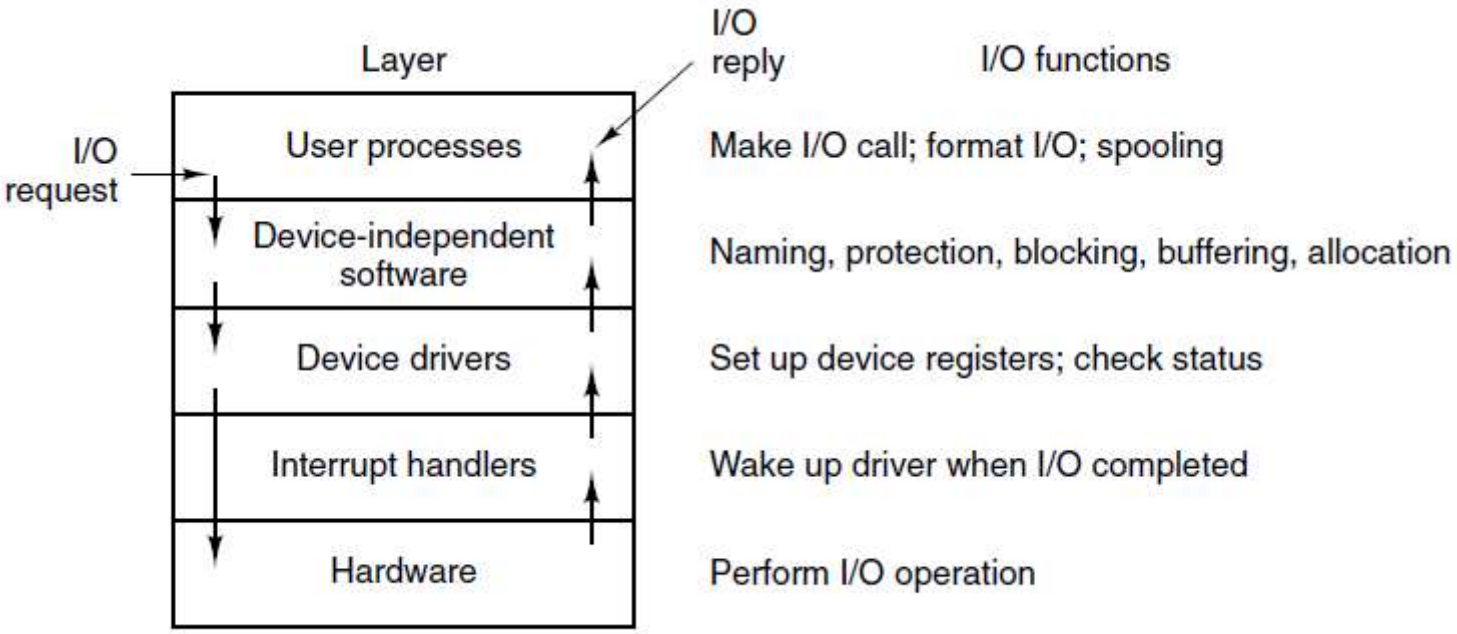
- Basic approach- Open the device-if the device is unavailable open fails
- If open fails ,block the caller and put on a queue

Device Independent Block size

- Uniform Block size to higher levels-by treating several sectors as a single logical block

User-Space I/O Software

Spooling



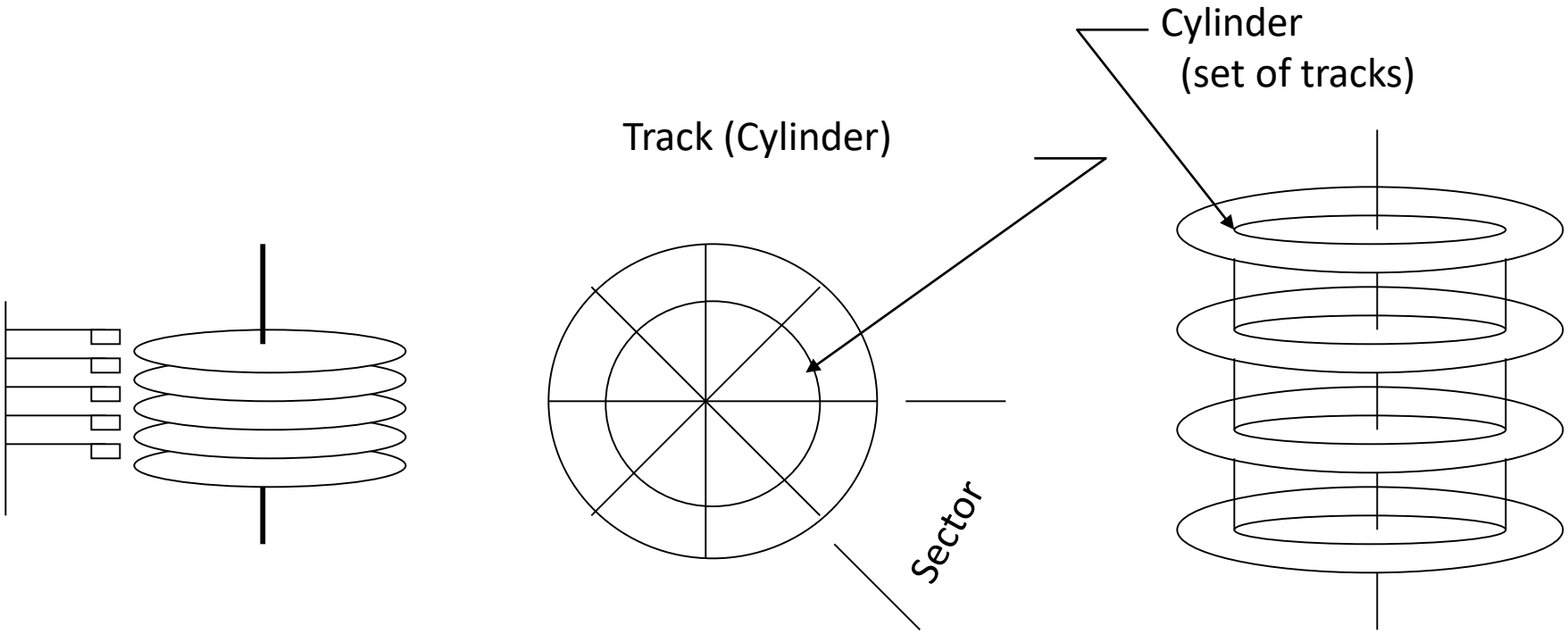
Layers of the I/O system and
the main functions of each layer.

Magnetic Disks (1)

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

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

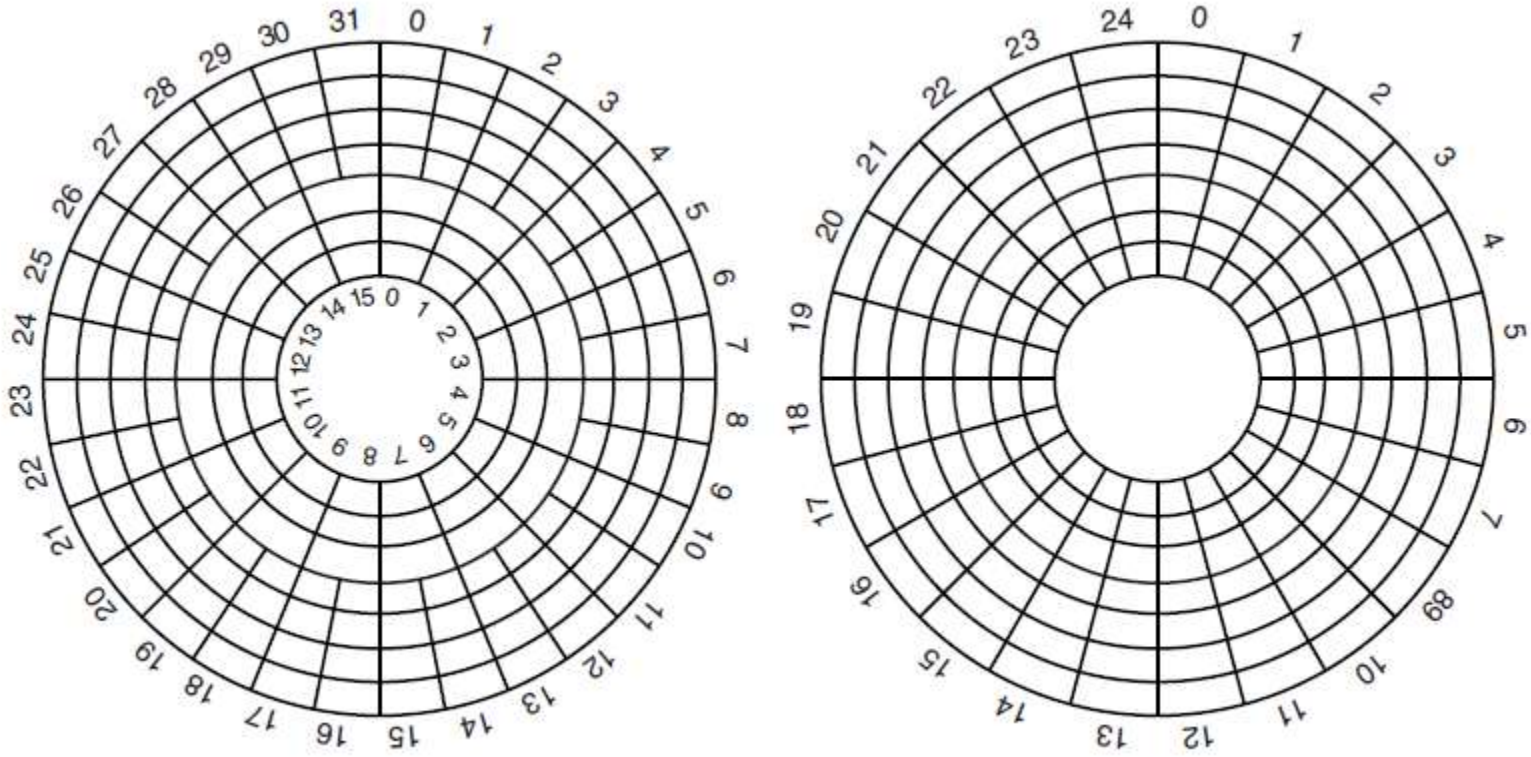
Rotating Media



(a) Multi-surface Disk (b) Disk Surface (b) Cylinders

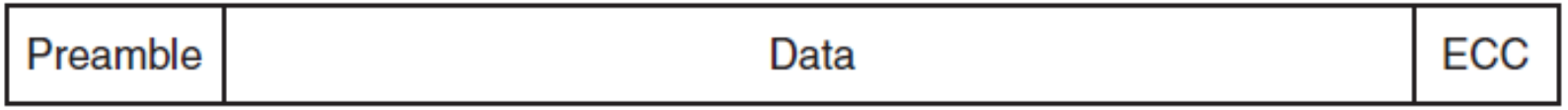
Operating Systems:
A Modern
Perspective, Chapter

Magnetic Disks



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

Disk Formatting



A disk sector.

Preamble:

- Bit pattern that identifies the start of sector
- Cylinder and sector numbers
- Other meta data

Error Correction Code (ECC)

- Redundant information used to recover read errors

Disk Formatting

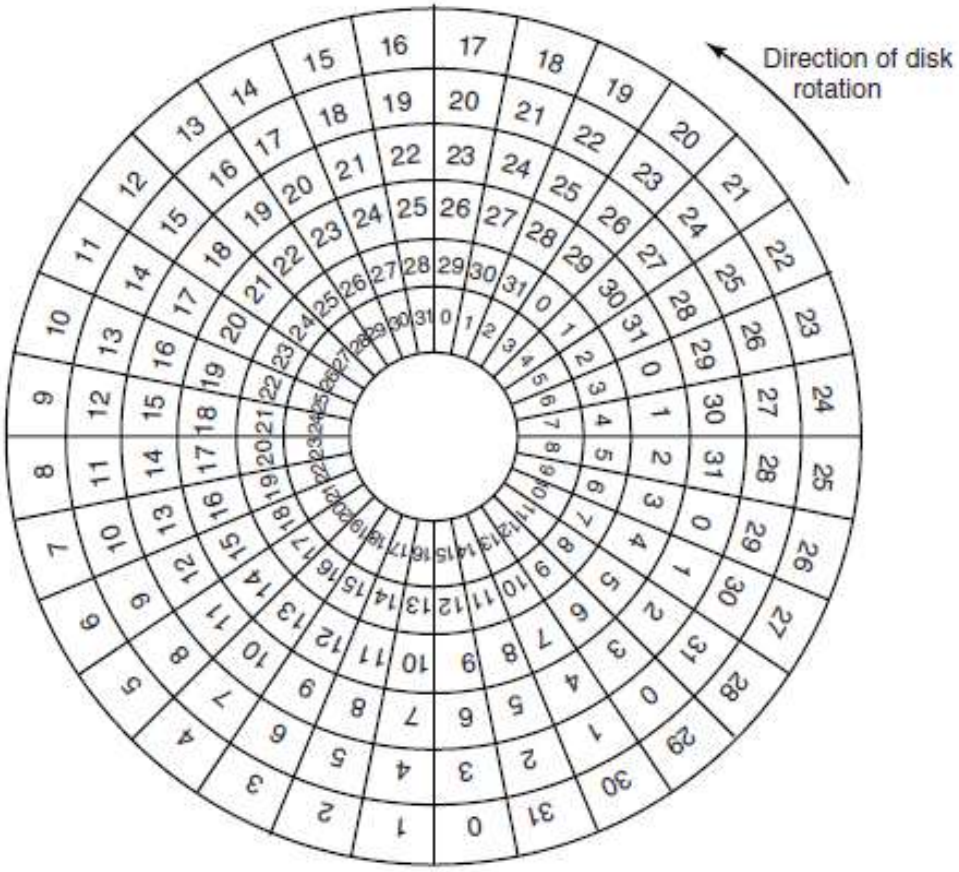
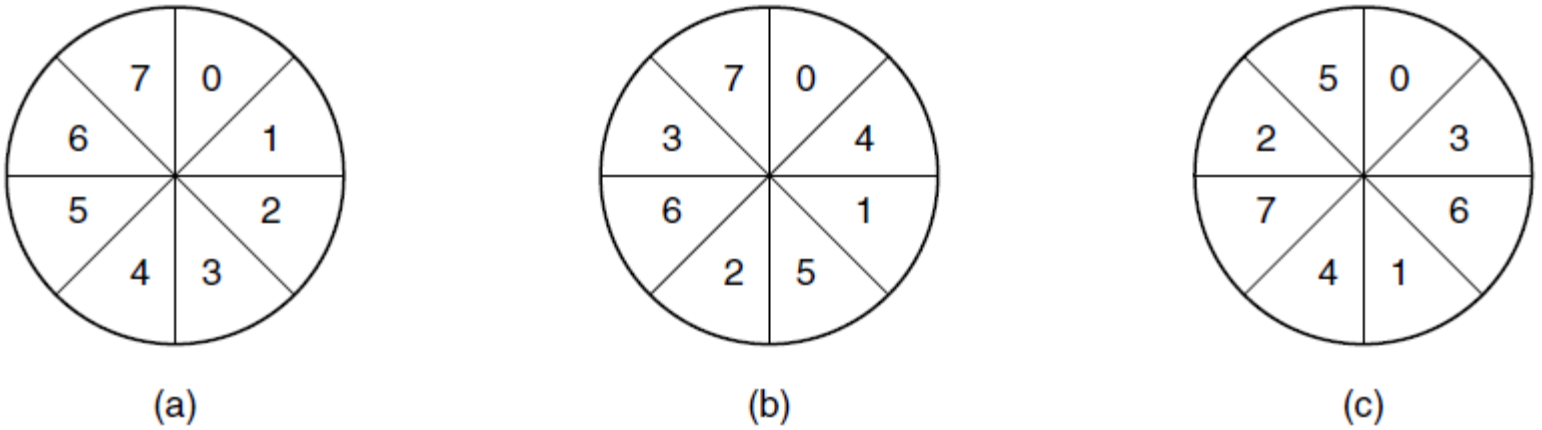


Figure 5-22. An illustration of cylinder skew.

Disk Formatting



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

Disk Optimizations

- Transfer Time: Time to copy bits from disk surface to memory
- Disk latency time: Rotational delay waiting for proper sector to rotate under R/W head
- Disk seek time: Delay while R/W head moves to the destination track/cylinder
- Access Time = seek + latency + transfer

Disk Arm Scheduling Algorithms

Factors of a disk block read/write:

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

Disk Arm Scheduling Algorithms (2)

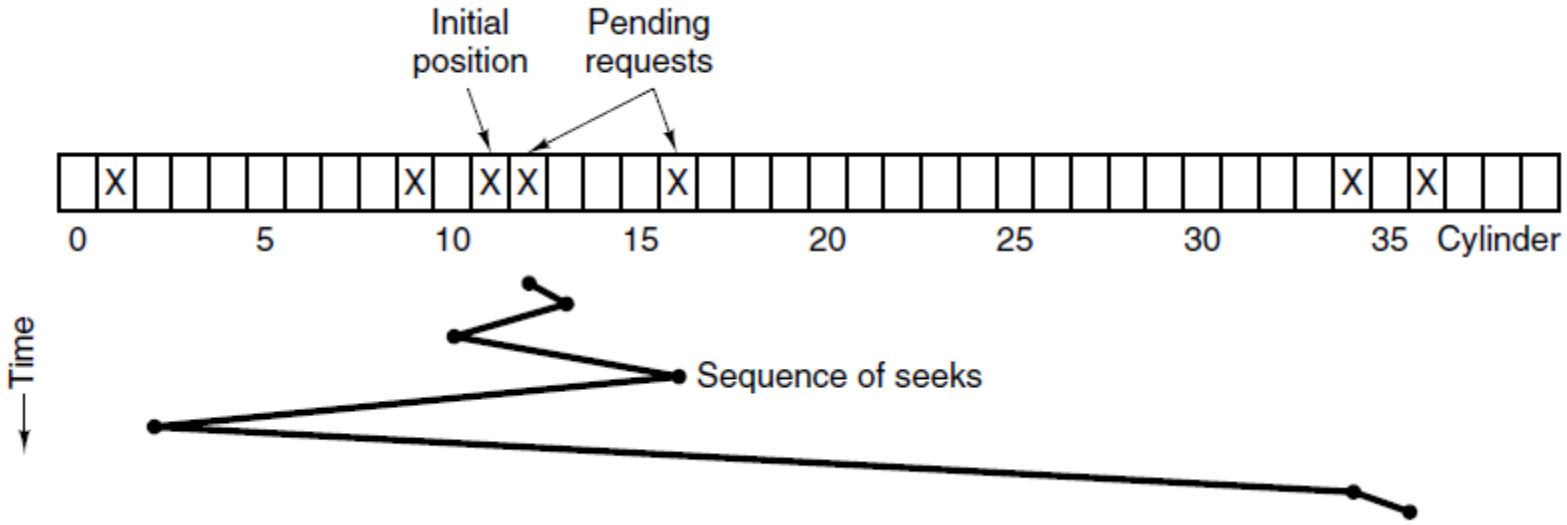


Figure 5-24. Shortest Seek First (SSF) disk scheduling algorithm.

Disk Arm Scheduling Algorithms (3)

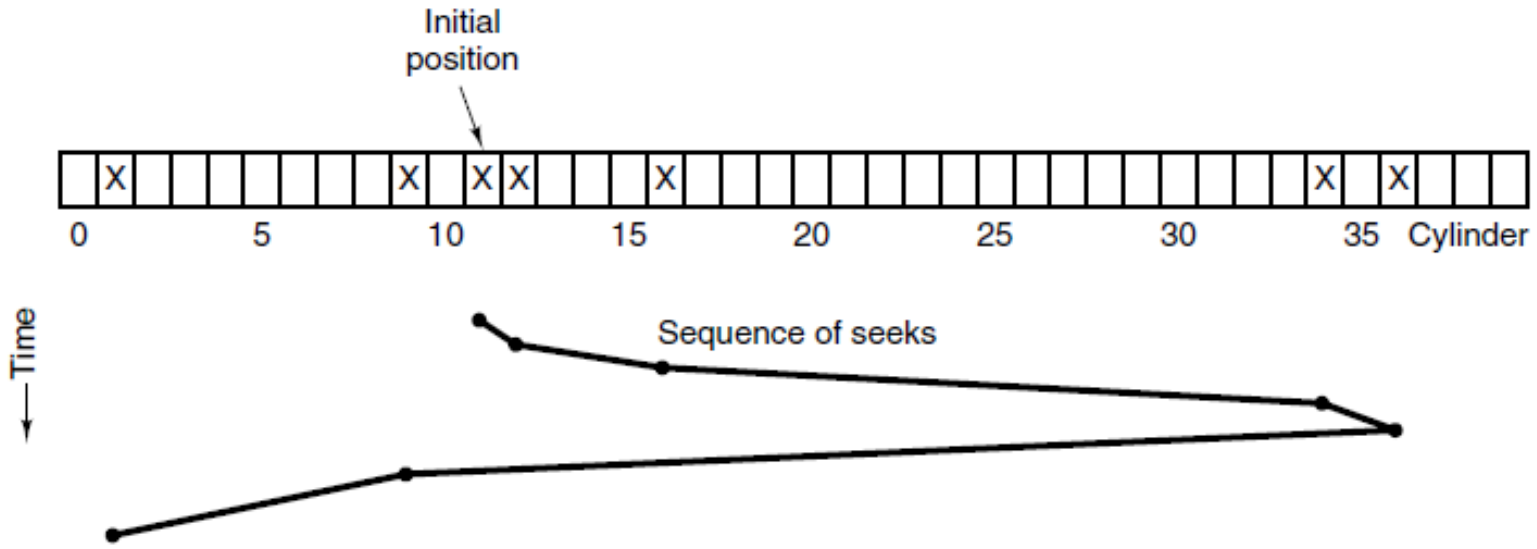


Figure 5-25. The elevator algorithm for scheduling disk requests.

Optimizing Seek Time

- Multiprogramming on I/O-bound programs => set of processes waiting for disk
- Seek time dominates access time => minimize seek time across the set
- Tracks 0:99; Head at track 75, requests for 23, 87, 36, 93, 66
- FCFS: 52+ 64 + 51 + 57 + 27 = 251 steps

Optimizing Seek Time (cont)

- Requests = 23, 87, 36, 93, 66
- SSTF: (75), 66, 87, 93, 36, 23

  11 + 21 + 6 + 57 + 13 = 107 steps
- Scan: (75), 87, 93, 99, 66, 36, 23

  12 + 6 + 6 + 33 + 30 + 13 = 100 steps
- Look: (75), 87, 93, 66, 36, 23

  12 + 6 + 27 + 30 + 13 = 87 steps

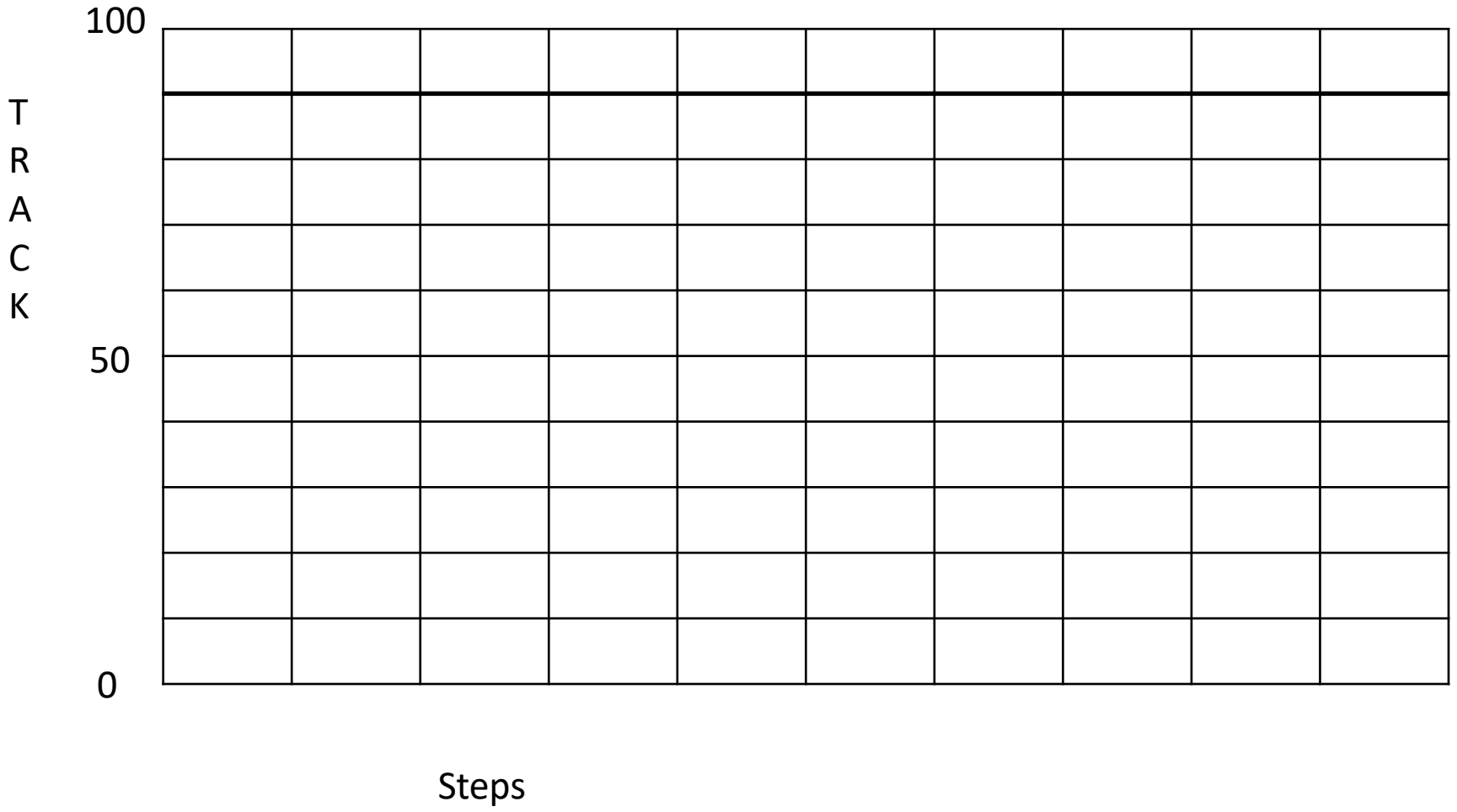
Optimizing Seek Time (cont)

- Requests = 23, 87, 36, 93, 66
- Circular Scan: (75), 87, 93, 99, 23, 36, 66

  12 + 6 + 6 + home + 23 + 13 + 30 = 90 + home
- Circular Look: (75), 87, 93, 23, 36, 66

  12 + 6 + home + 23 + 13 + 30 = 84 + home

Optimizing Seek Time



Error Handling

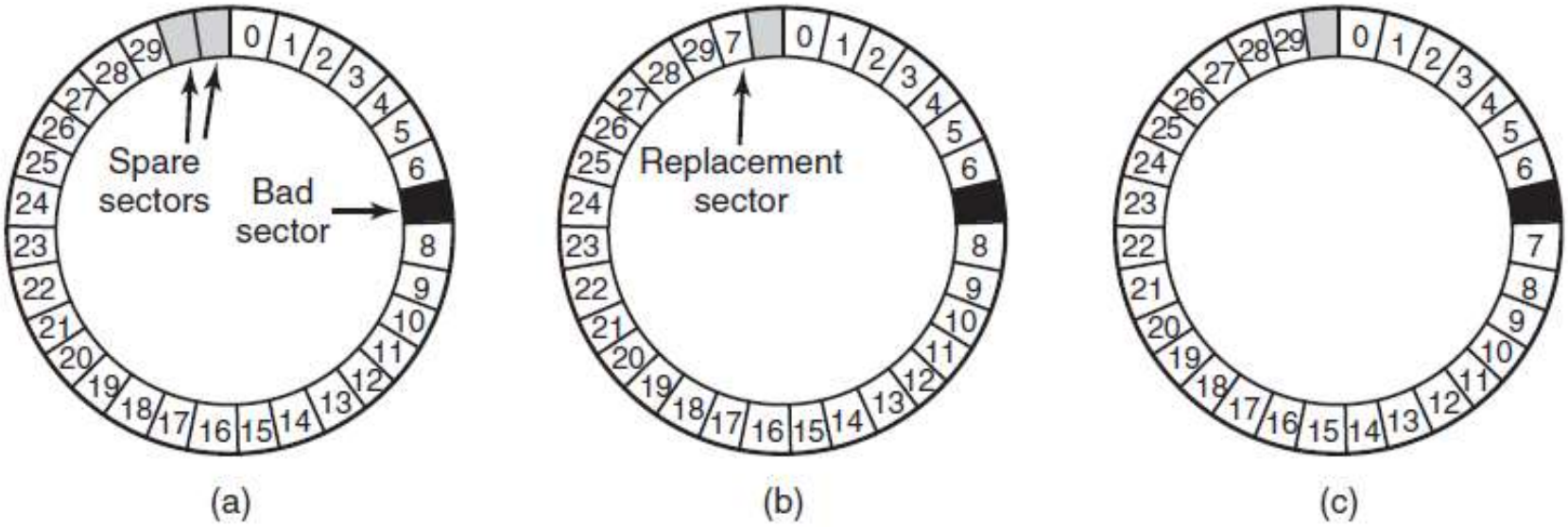
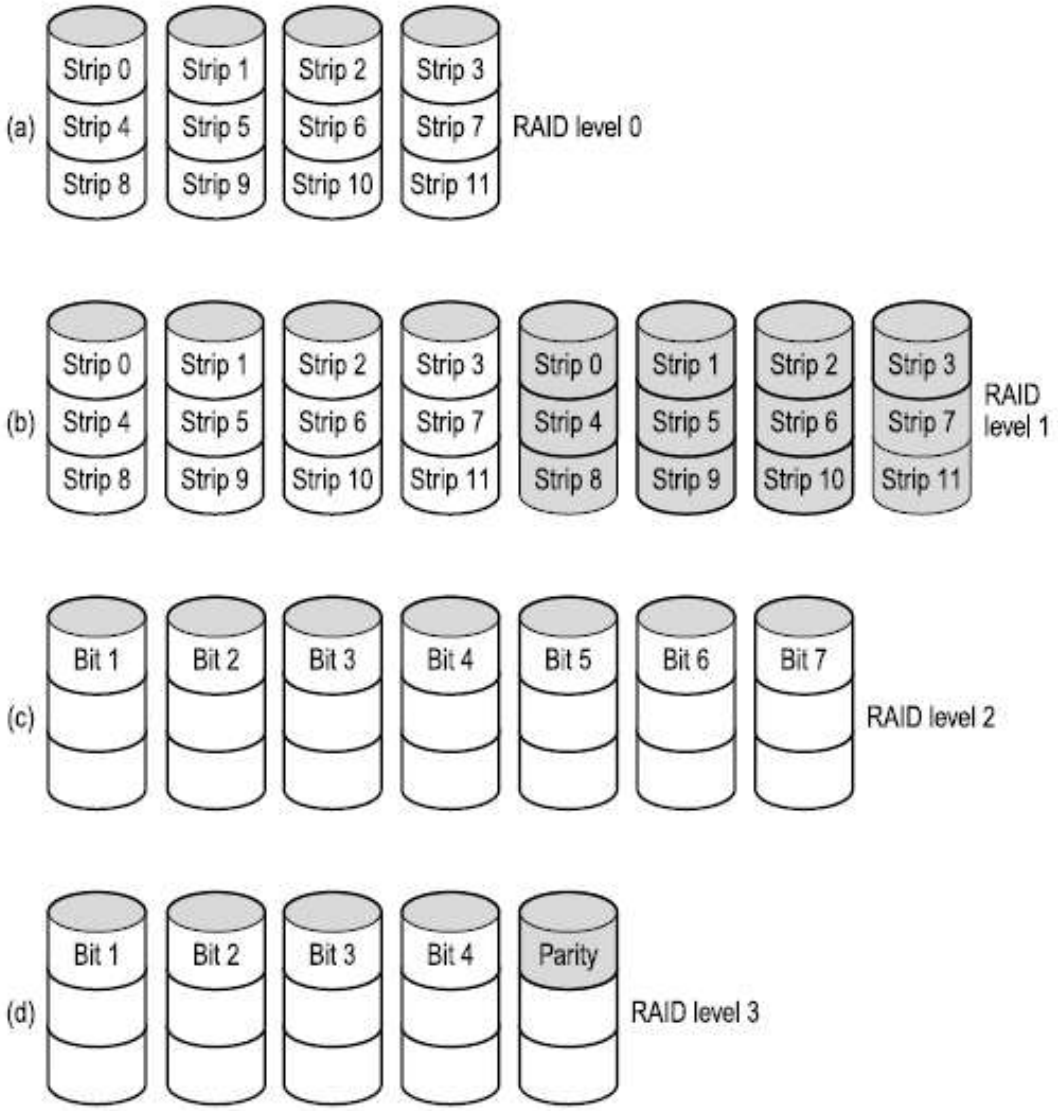


Figure 5-26. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

RAID

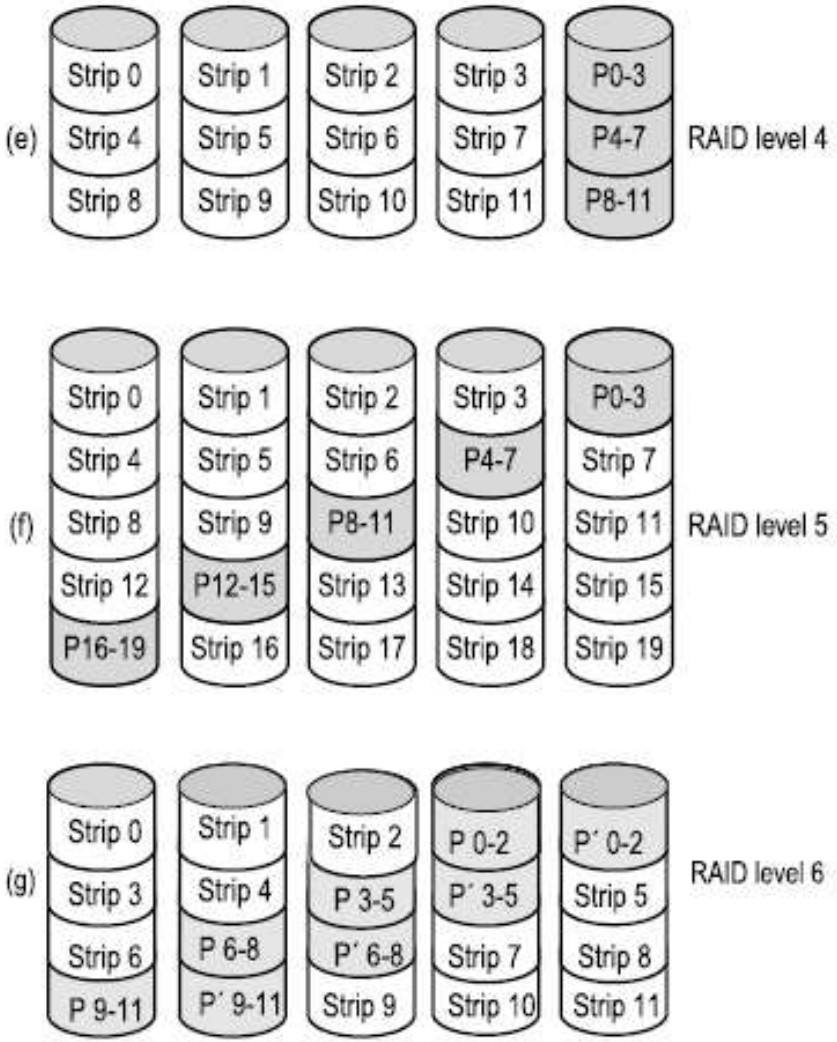
Redundant Array of Inexpensive Disks



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

RAID

Redundant Array of Inexpensive Disks



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

Clock Hardware

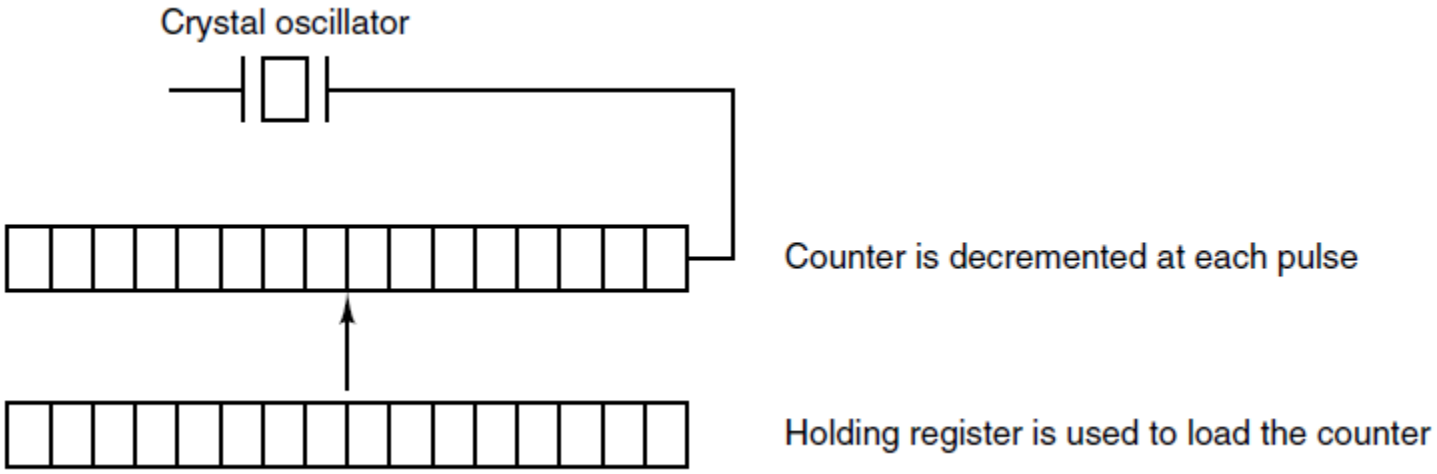


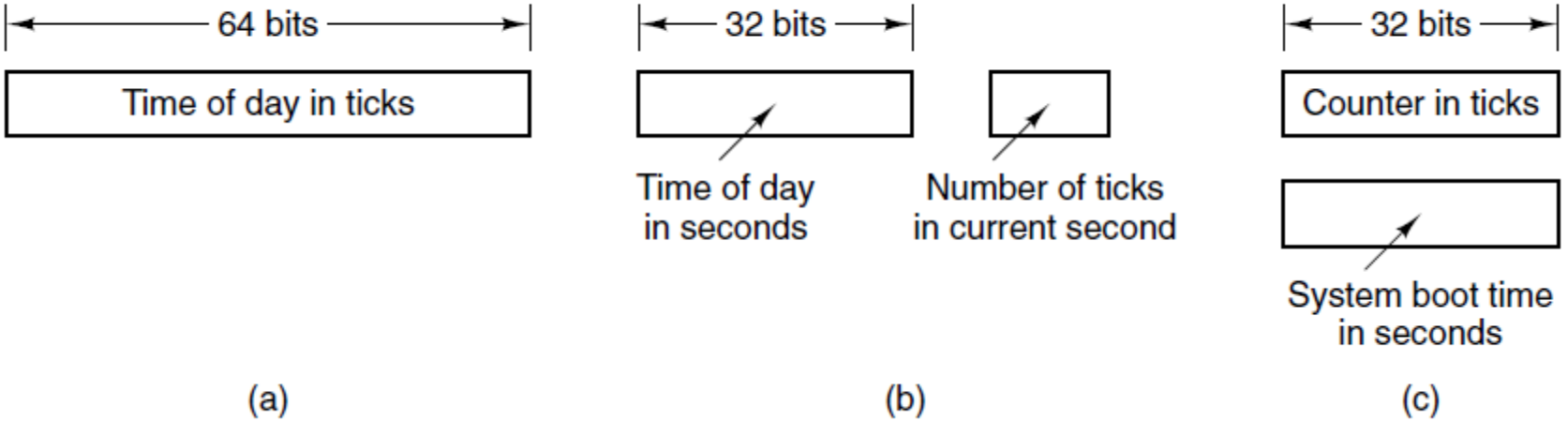
Figure 5-28. A programmable clock.

Clock Software

Typical duties of a clock driver:

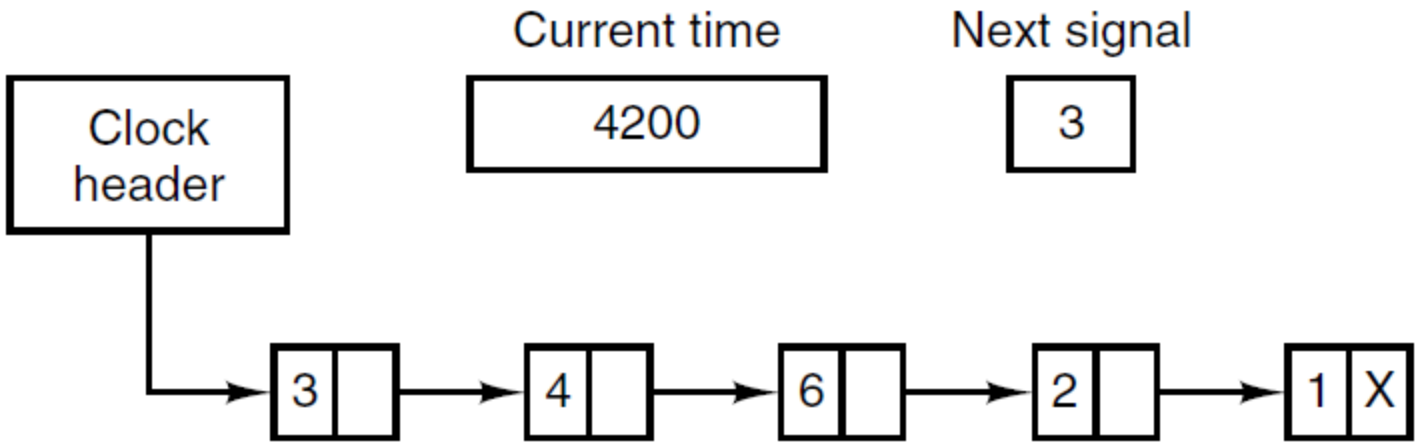
1. Maintaining the time of day.
2. Preventing processes from running longer than allowed.
3. Accounting for CPU usage.
4. Handling alarm system call from user processes.
5. Providing watchdog timers for parts of system itself.
6. Profiling, monitoring, statistics gathering.

Clock Software



Three ways to maintain the time of day.

Clock Software



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

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)

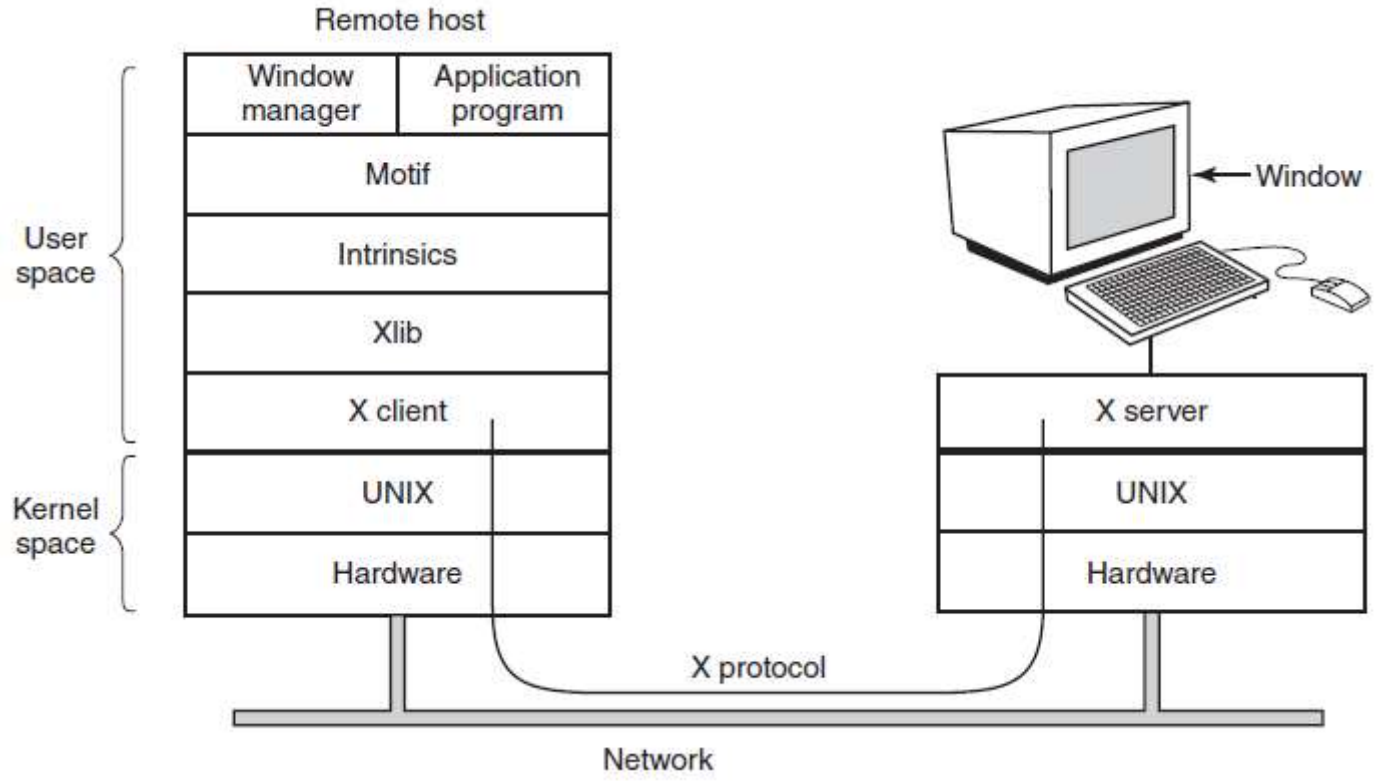


Figure 5-33. Clients and servers in the M.I.T. X Window System.

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)

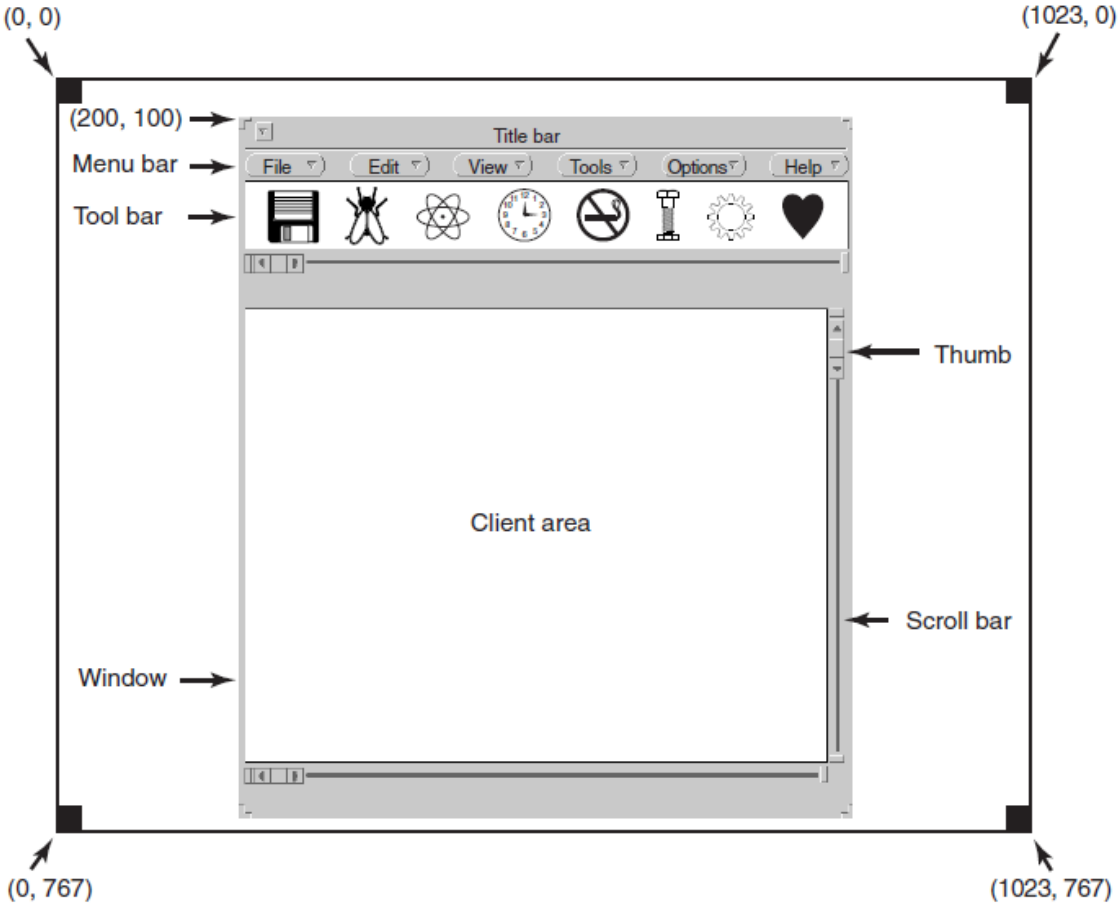


Figure 5-35. A sample window located at (200, 100) on an XGA display.

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)

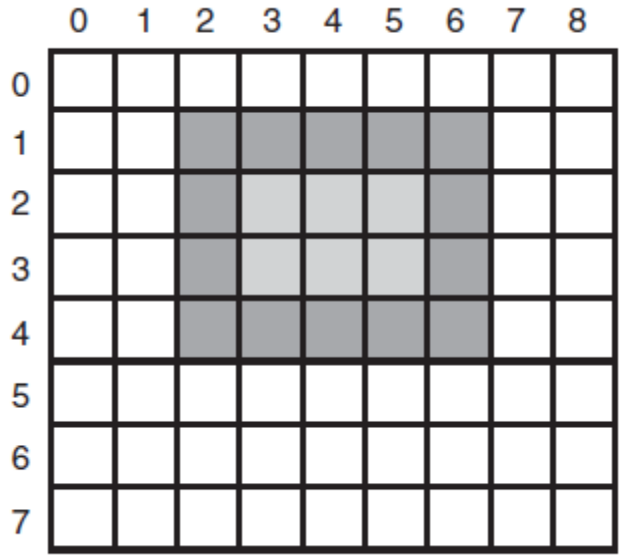


Figure 5-37. An example rectangle drawn using *Rectangle*. Each box represents one pixel.

Bitmaps

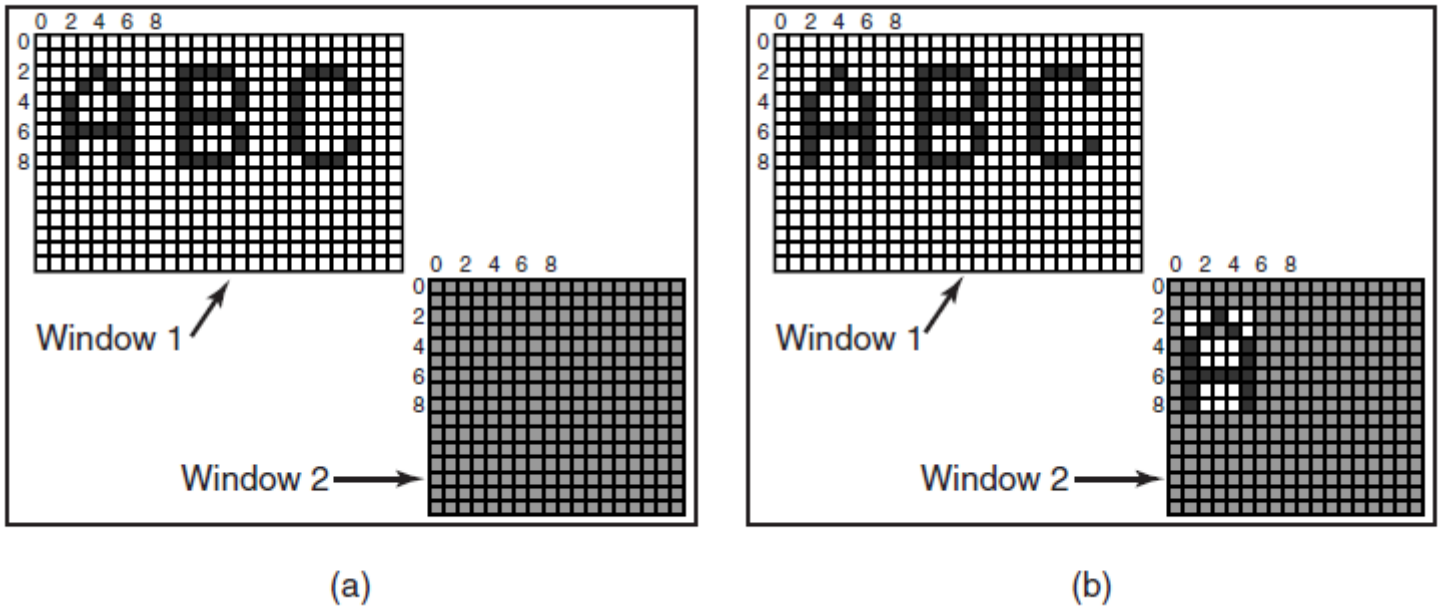


Figure 5-38. Copying bitmaps using BitBlt.
(a) Before. (b) After.

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

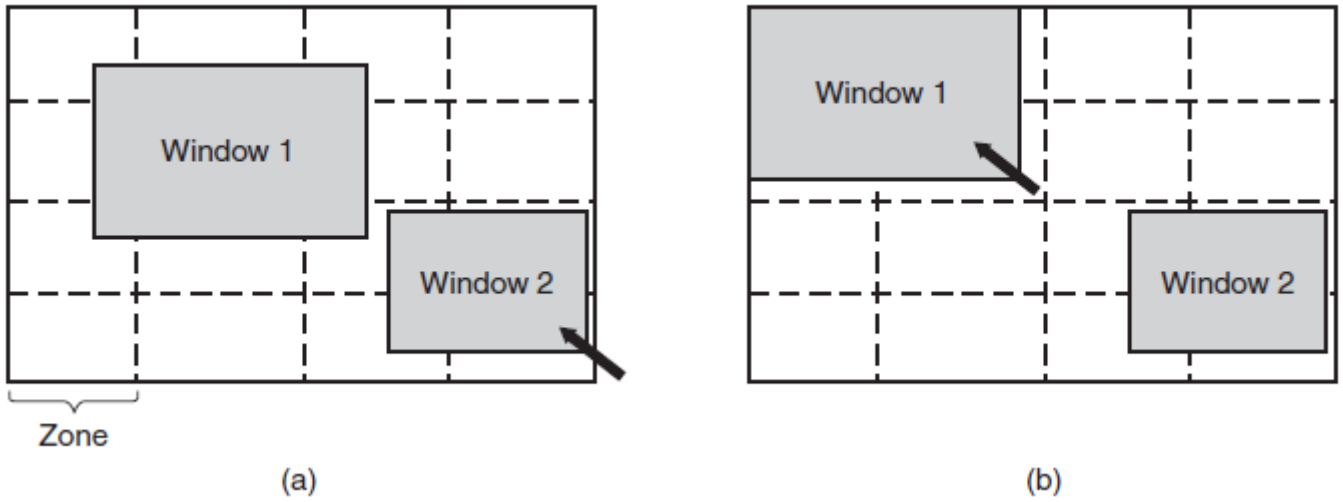


Figure 5-41. The use of zones for backlighting the display. (a) When window 2 is selected it is not moved. (b) When window 1 is selected, it moves to reduce the number of zones illuminated.

Operating System Issues
The CPU

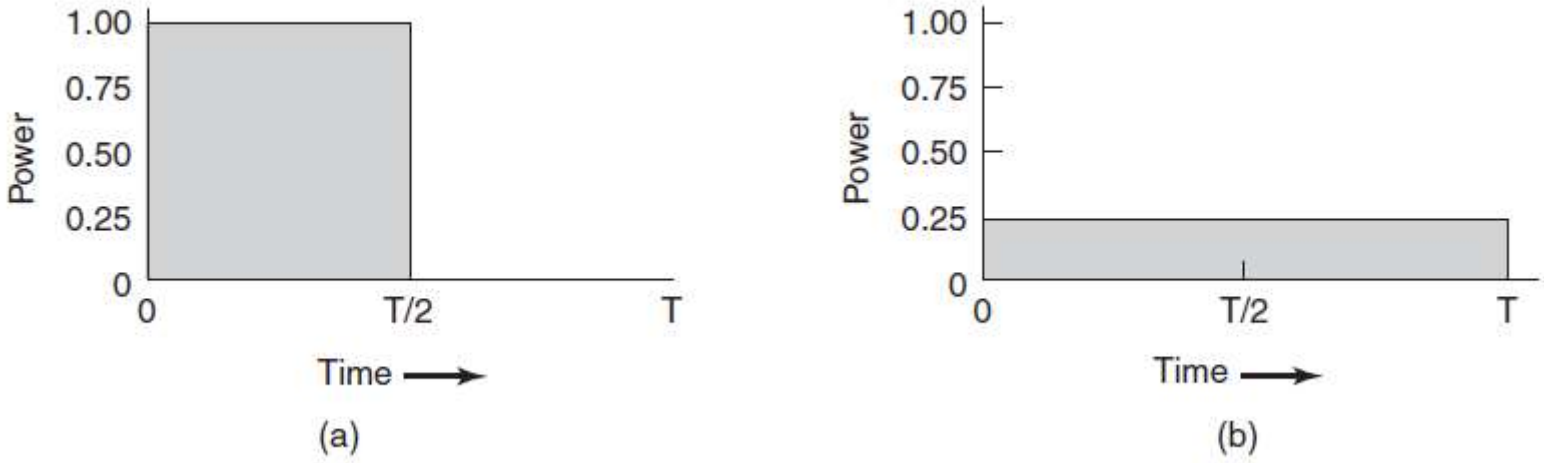


Figure 5-42. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four

End

Chapter 5