Input/Output

Chapter 5

Part 1 outlines

- Requirements of I/O
- I/O devices
- Bus, Controller
- Communication between Processor and I/O
- Interrupt revisited

Requirements of I/O

So far, we have studied:

» Abstractions: Process, Thread, Address Space, File System

» Synchronization/Scheduling: How to manage the CPU

» What about I/O?

➡Without I/O, computers are useless (disembodied brains?)

Requirements of I/O

- But… thousands of devices, each slightly different
  - How can we standardize the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
  - How can we make them reliable???
- Devices unpredictable and/or slow
  - How can we manage them if we don't know what they will do or how they will perform?

Range of Timescales

"Numbers Everyone Should Know"
— Jeff Dean,
Google Senior Fellow, known for "MapReduce", "TensorFlow"

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Some typical device, network, and bus data rates

- Device rates vary over multiple orders of magnitude!!!
- System must be able to handle this wide range
  - Better not have high overhead for fast devices
  - Better not waste time waiting for slow devices

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 Devices

- 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

- 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

I/O Devices

- Net devices
  - The handled information is the **Package**
    - ➡ the OS maintains the protocol to wrap, analyze, locate…, of these packages,
  - The interface to access these data is **Socket**
  - Users communicate with socket, e.g., receive(), send()

I/O Devices

- Clocks and Timers:
  - Provide current time, elapsed time, timer
  - Normal resolution about 1/60 second
  - Some systems provide higher-resolution timers
  - Programmable interval timer used for timings, periodic interrupts

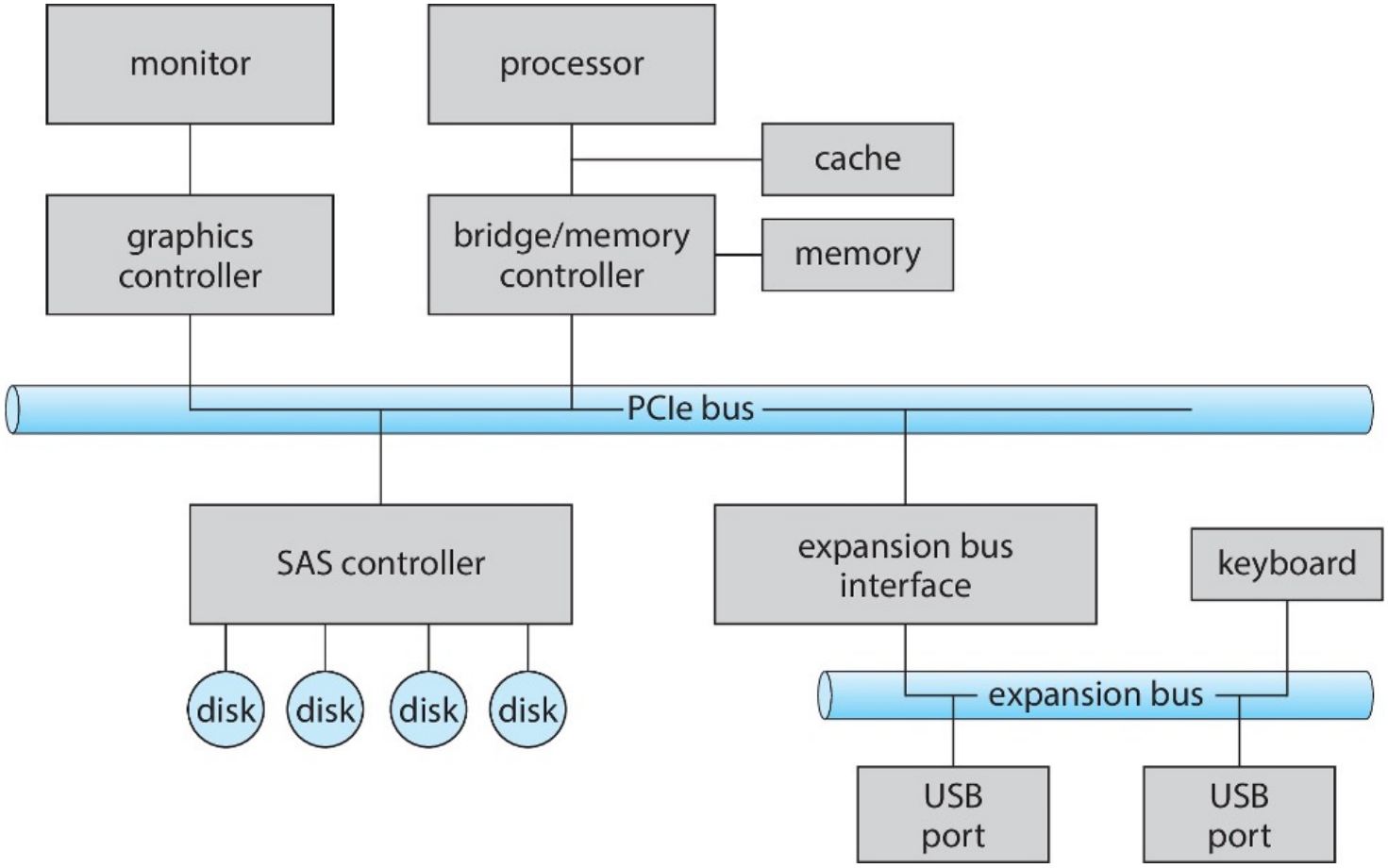
I/O Devices

- UNIX and Linux use tuple of “major” and “minor” device numbers to identify type and instance of devices

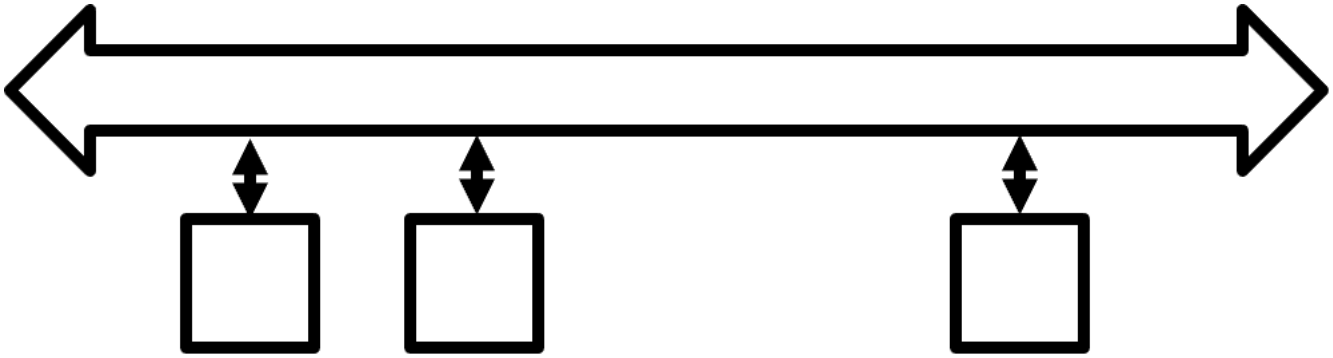
```
% ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

  - Here, major 8 and minors 0-3.

Modern I/O Systems



What's a Bus?



- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
  - Operations: e.g., Read, Write
  - Control lines, Address lines, Data lines
- Protocol: initiator requests access, arbitration to grant, identification of recipient, length, data

Why a Bus?

- Buses let us connect *n* devices over a single set of wires, connections, and protocols
  - O(n^2) relationships with 1 set of wires (!)

- Downside: Only one transaction at a time
  - The rest must wait
  - "Arbitration" aspect of bus protocol ensures the rest wait

Refer to PCIe (Peripheral Component Interconnect Express) for more details

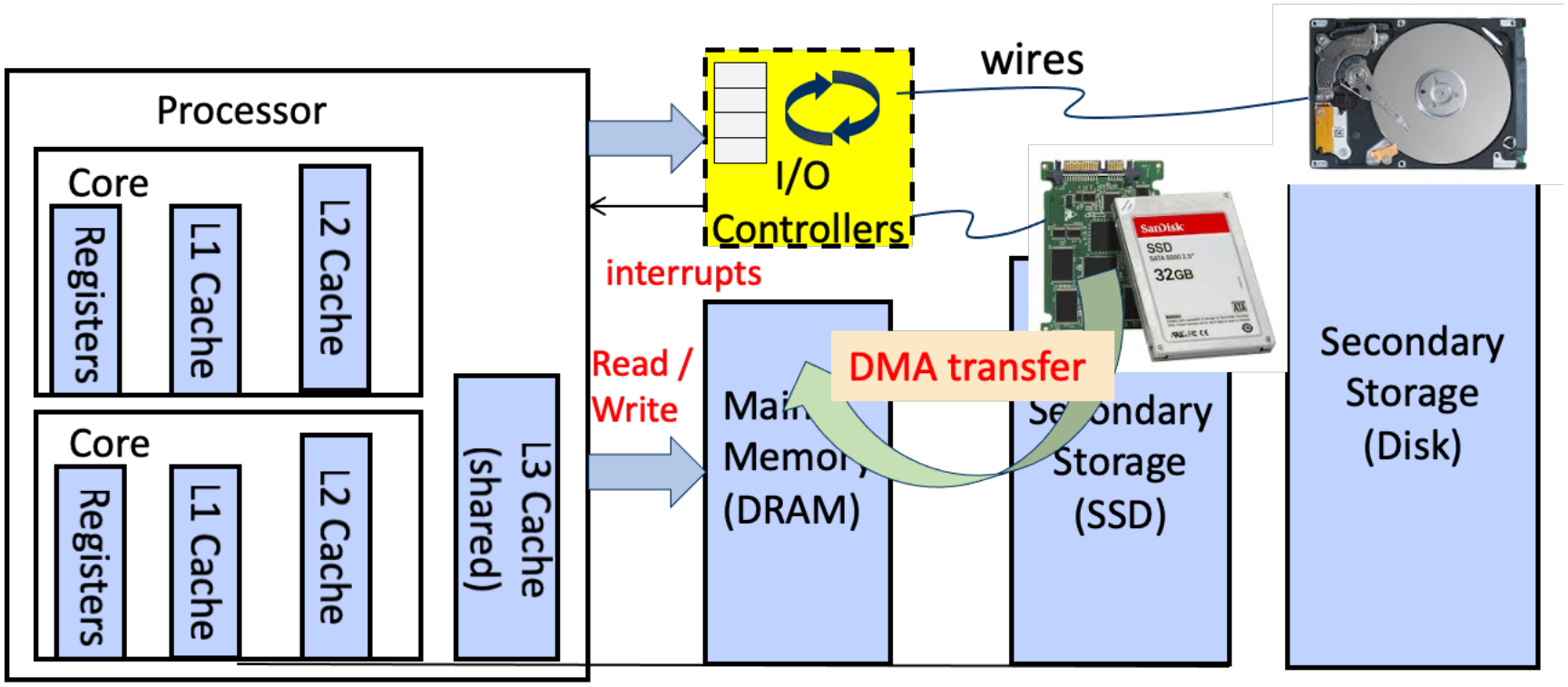
Device Controllers

- A controller is a collection of electronics that can operate a port, a bus, or a device.
  - Sometimes integrated
  - Sometimes separate circuit board
  - Some devices have their own built-in controllers.

Device Controllers

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

How does the Processor Talk to the Device?



»I/O devices you recognize are supported by I/O controllers

»CPU accesses them by reading and writing I/O registers like memory

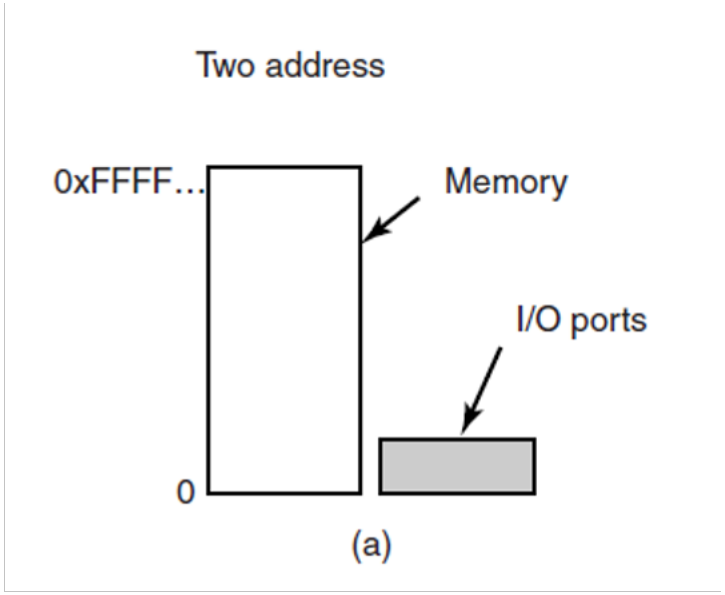
- Write commands and arguments, read status and results

How does the Processor Talk to the Device?

- CPU interacts with a Controller
  - Contains a set of registers that can be read and written
  - May contain memory for request queues, etc.
- Processor accesses registers in two ways:
  - Port-Mapped I/O: in/out instructions
    - Example from the Intel architecture: out 0x21,AL
  - Memory-mapped I/O: load/store instructions
    - Registers/memory appear in physical address space
    - I/O accomplished with load and store instructions

Port-Mapped I/O

- 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

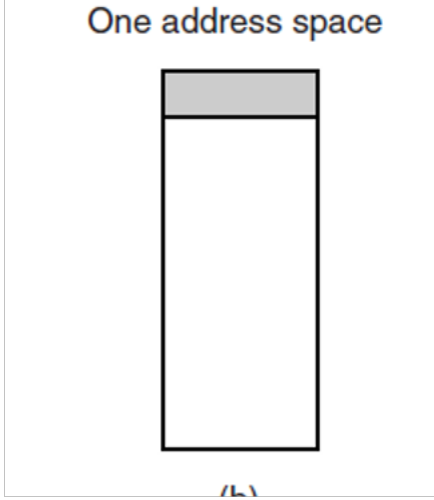


Direct I/O Port Locations on PCs (partial)

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 address space
- When the CPU wants to read a word, it puts the address it needs on the bus
  - If it falls in the I/O address range, the device responds to the request.



✓ Without memory-mapped I/O, some assembly code is needed.

✓ No additional protected space (as long as it falls in the user's virtual address space)

✓ Every instruction that can reference memory can also reference control registers.

Memory-Mapped I/O

Flexible when using memory Mapped I/O, but…

```
LOOP:   TEST PORT_4          // check if port 4 is 0
        BEQ READY            // if it is 0, go to ready
        BRANCH LOOP          // otherwise, continue testing
READY:
```

- Caching a device control register would be disastrous.
- Consider the above code
- The first reference to PORT 4 would cause it to be cached. Subsequent references would just take the value from the cache and not even ask the device.

When the device finally became ready, the software would have no way of finding out. Instead, the loop would go on forever.

Direct Memory Access

- Requires DMA controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller
  - Bus mastering of DMA controller – grabs bus from CPU
    - Cycle stealing from CPU but still much more efficient
  - When done, interrupts to signal completion

Direct Memory Access

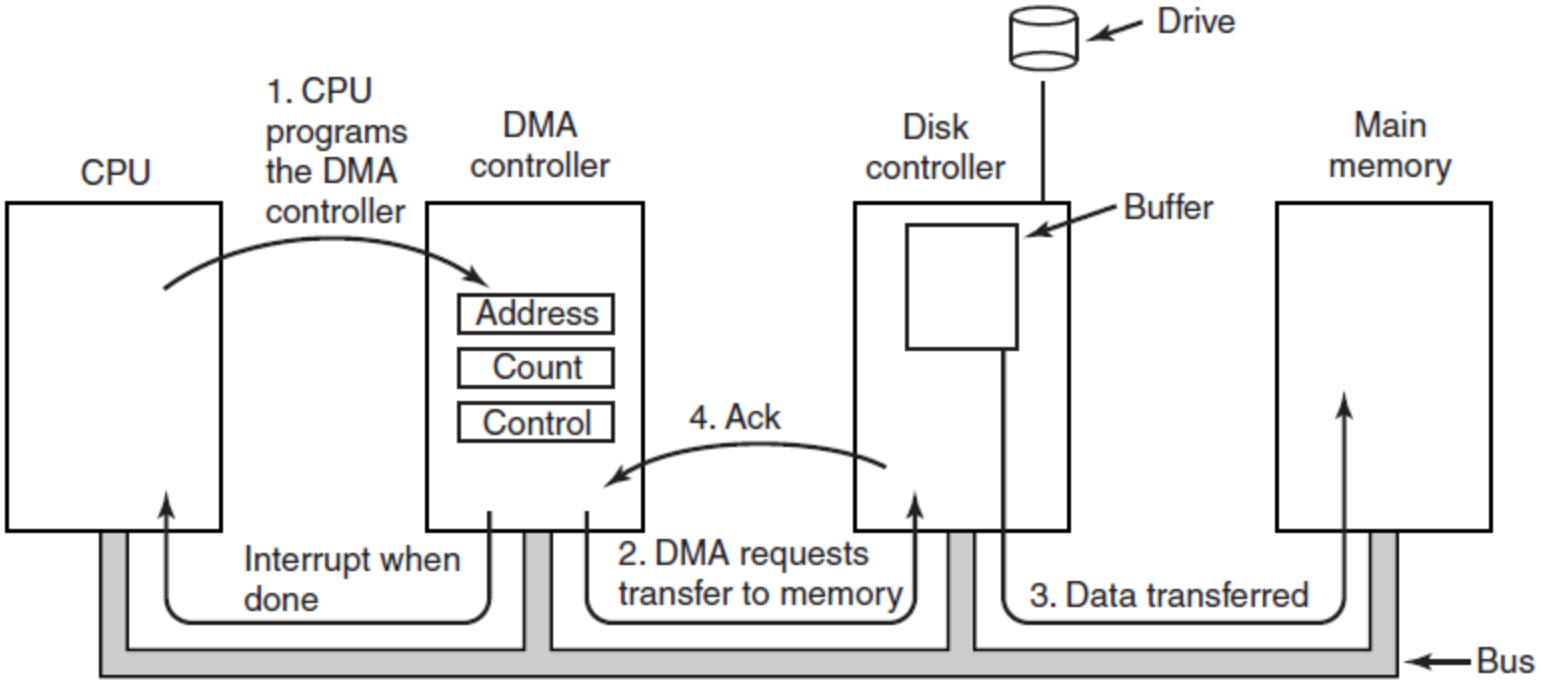


Figure 5-4. Operation of a DMA transfer.

I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- Polling: OS periodically checks device-specific status register
  - Low overhead, but may waste cycles for infrequent or unpredictable I/O
- I/O Interrupt: Device generates interrupt when it needs service
  - Handles unpredictable events well, but high overhead
- Actual devices normally combine both polling and interrupts

Interrupts Revisited

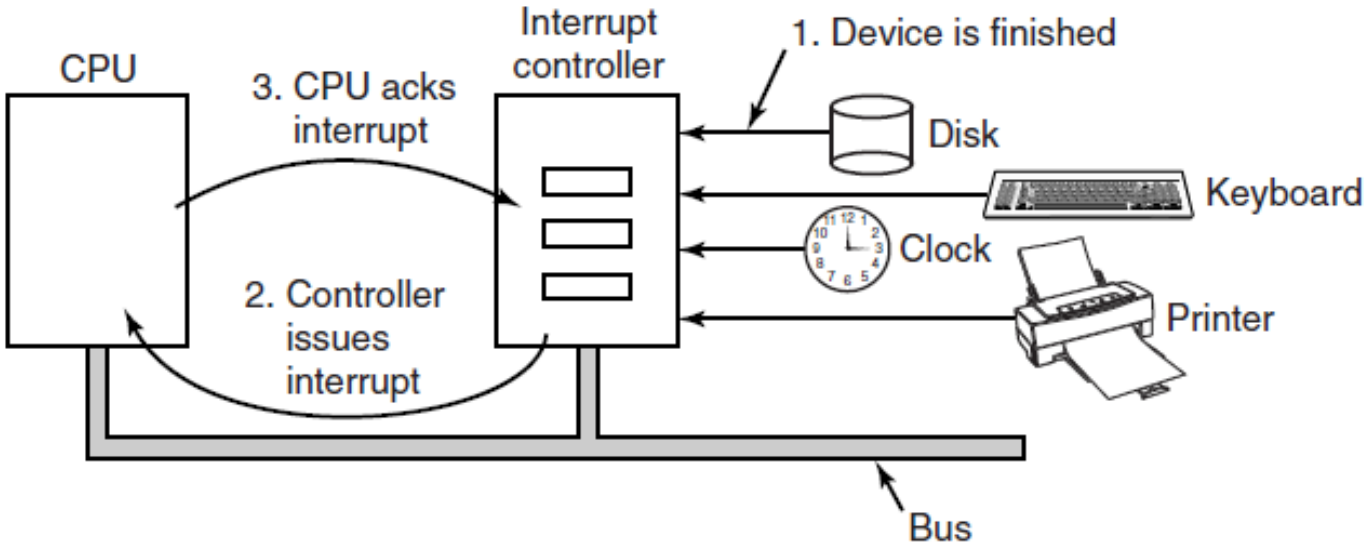


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Interrupts Revisited

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

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

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

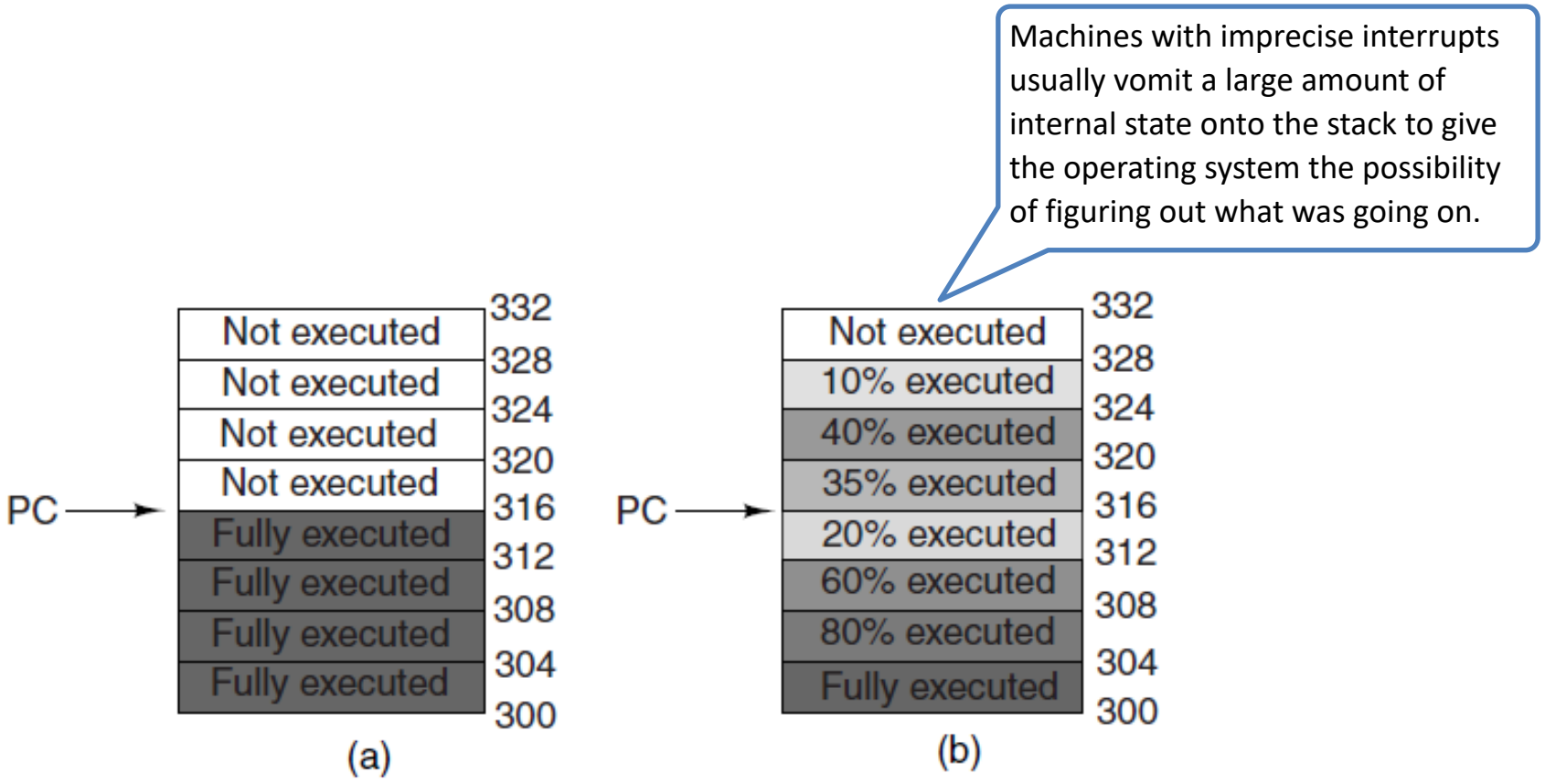


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

Part 2 Outlines

- Programmed I/O
- Interrupt-Driven I/O
- I/O Using DMA
- Device Drivers
- Buffering

Programmed I/O

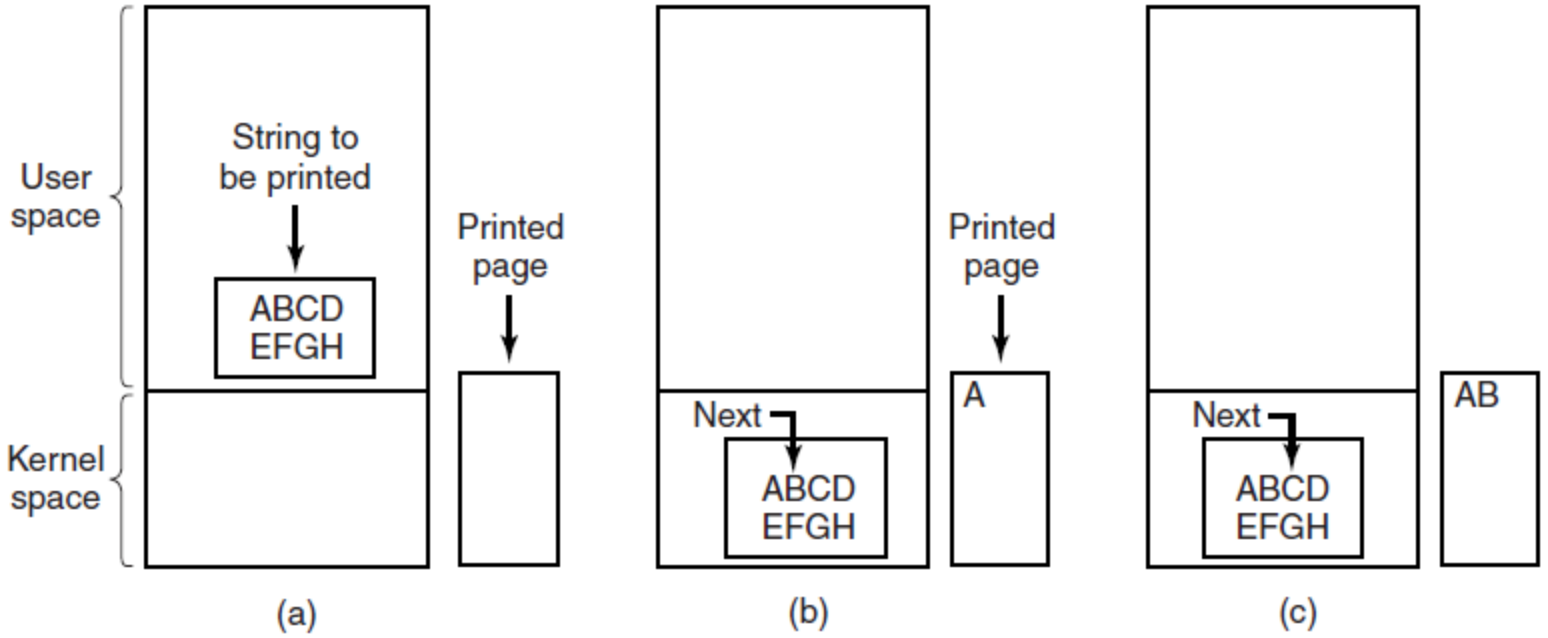


Figure 5-7. Steps in printing a string.

Programmed I/O

```
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( );
```

- Reasonable if device is fast
- But inefficient if device slow

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.

Goals of the I/O Software

Issues:

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

I/O Software Layers

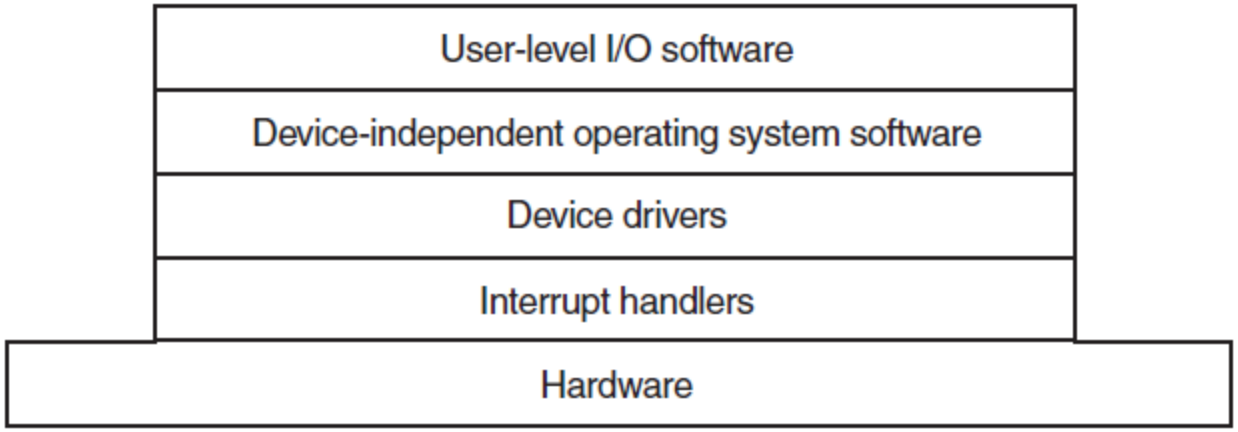
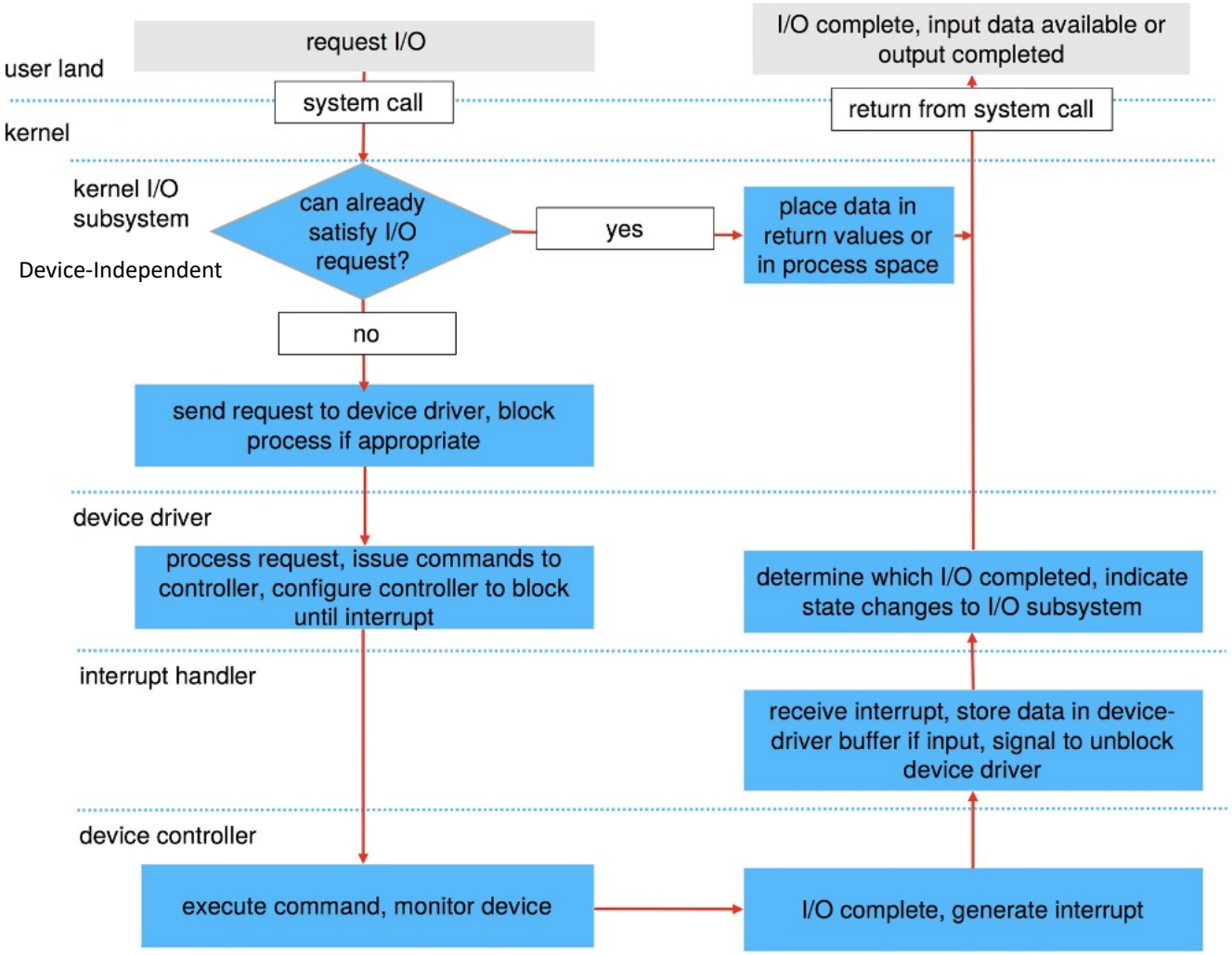


Figure 5-11. Layers of the I/O software system.

I/O Software Layers



Device-Independent I/O Software

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

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.

Device Drivers

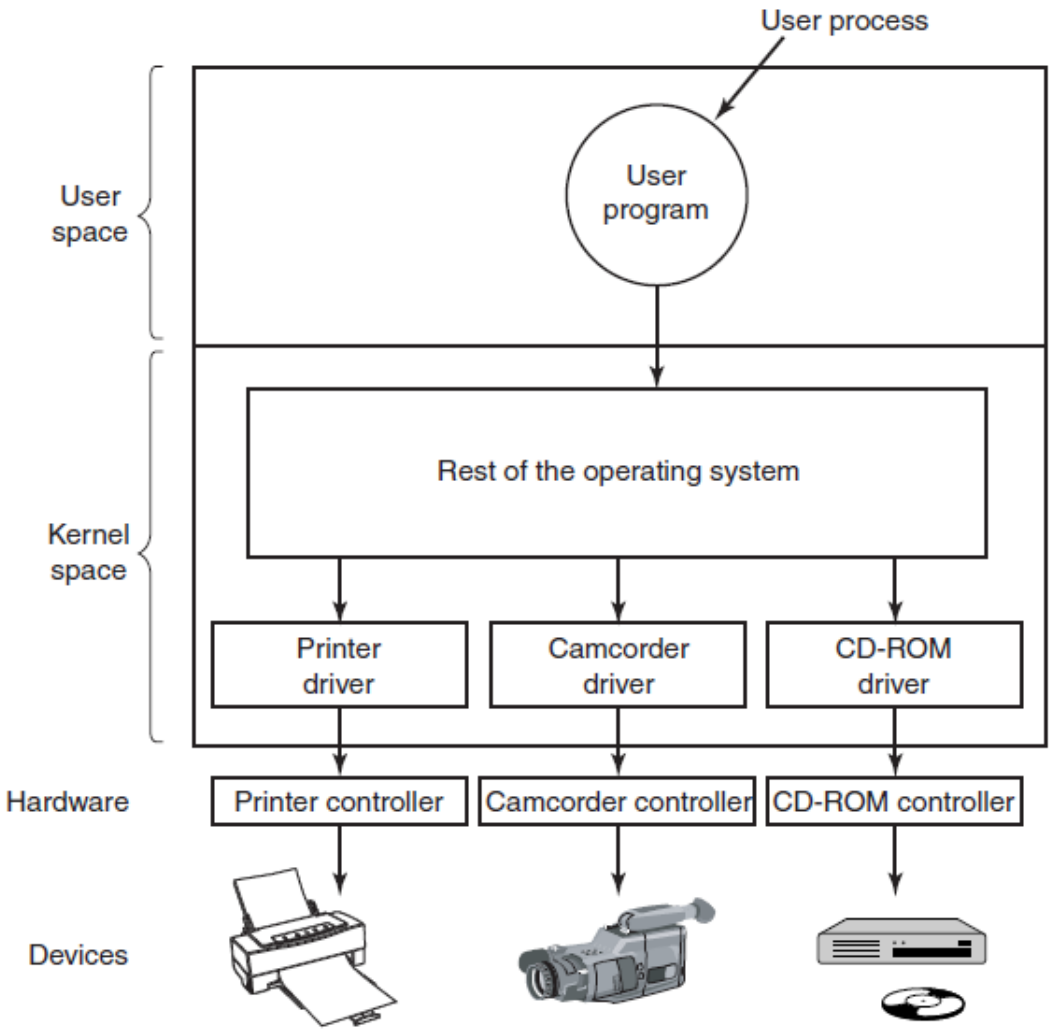


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Uniform Interfacing for Device Drivers

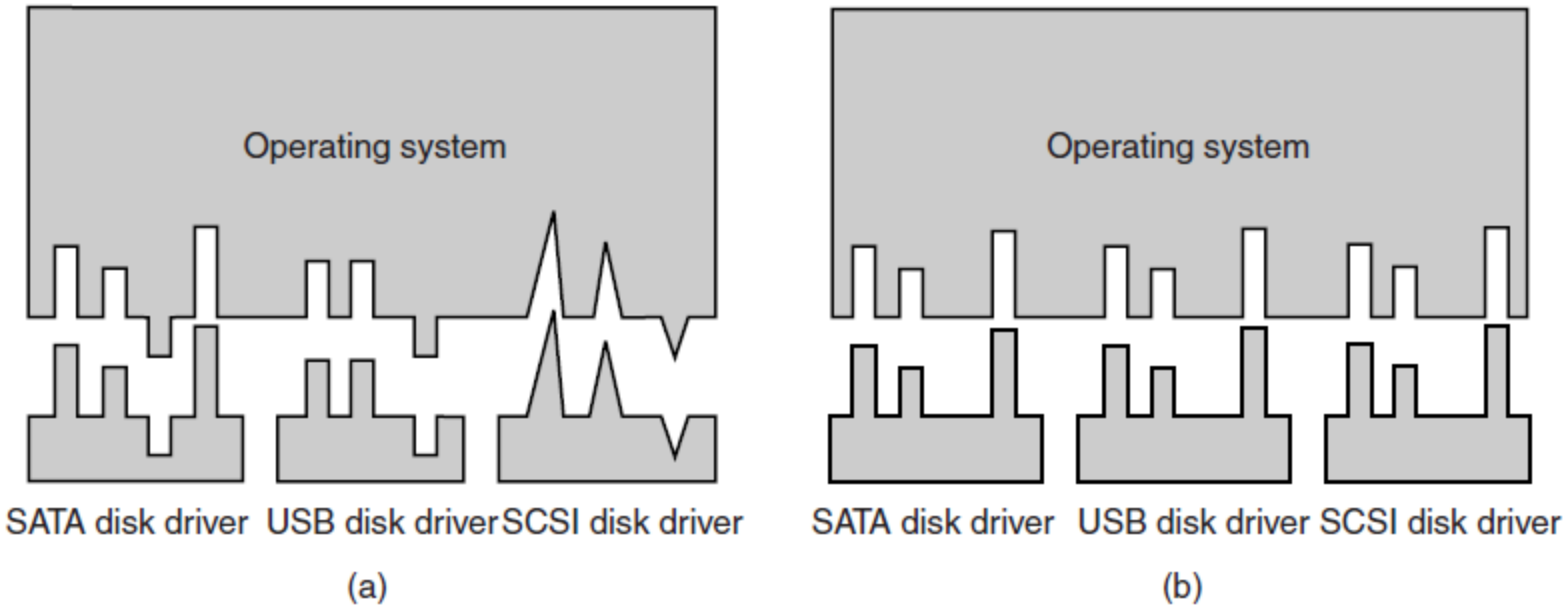


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

What Does A Device Driver Do?

- Provide “the rest of the OS” with APIs
  - Init, Open, Close, Read, Write, ...
- Interface with controllers
  - Commands and data transfers with hardware cont
- Driver operations
  - Initialize devices
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Uniform Interfacing for Device Drivers — Character device interface

- For example, keyboard, mouse, ports
- read(deviceNumber, bufferAddr, size)
  - Reads “size” bytes from a byte stream device to “bufferAddr”
- write(deviceNumber, bufferAddr, size)
  - Write “size” bytes from “bufferAddr” to a byte stream device

Uniform Interfacing for Device Drivers
— Block device interface

- For example, disk drives
- read(deviceNumber, deviceAddr, bufferAddr)
  - Transfer a block of data from “deviceAddr” to “bufferAddr”
- write(deviceNumber, deviceAddr, bufferAddr)
  - Transfer a block of data from “bufferAddr” to “deviceAddr”
- seek(deviceNumber, deviceAddress)
  - Move the head to the correct position

Uniform Interfacing for Device Drivers —Network Devices

- Different enough from the block & character devices to have own interface
- Unix and Windows/NT include socket interface
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Uniform Interfacing for Device Drivers —Devices Clocks and Timers

`ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers

Uniform Interfacing for Device Drivers

Some common Entry Points

- `init()`
  - Initialize hardware
- `start()`
  - Boot time initialization
- `open(dev, flag, id)` and `close(dev, flag, id)`
  - Initialization resources for read or write and release resources
- `halt()`
  - Call before the system is shutdown
- `intr(vector)`
  - Called by the kernel on a hardware interrupt
- `read(...)` and `write()` calls
  - Data transfer
- `poll(pri)`
  - Called by the kernel 25 to 100 times a second
- `ioctl(dev, cmd, arg, mode)`
  - special request processing

Buffering

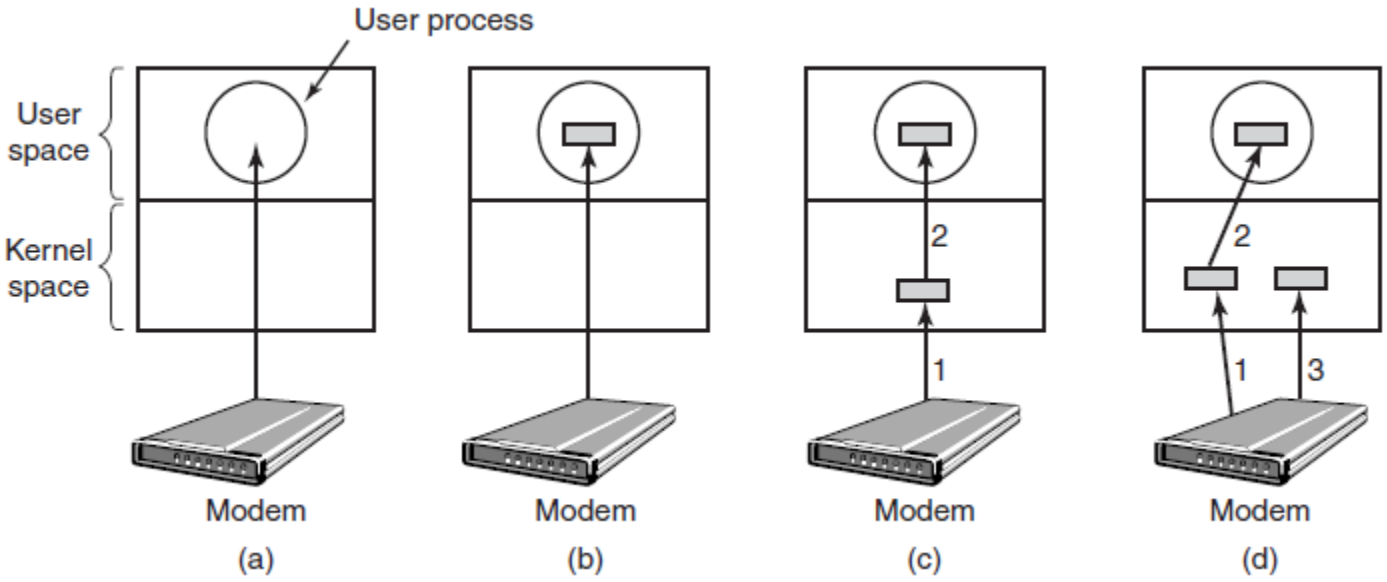
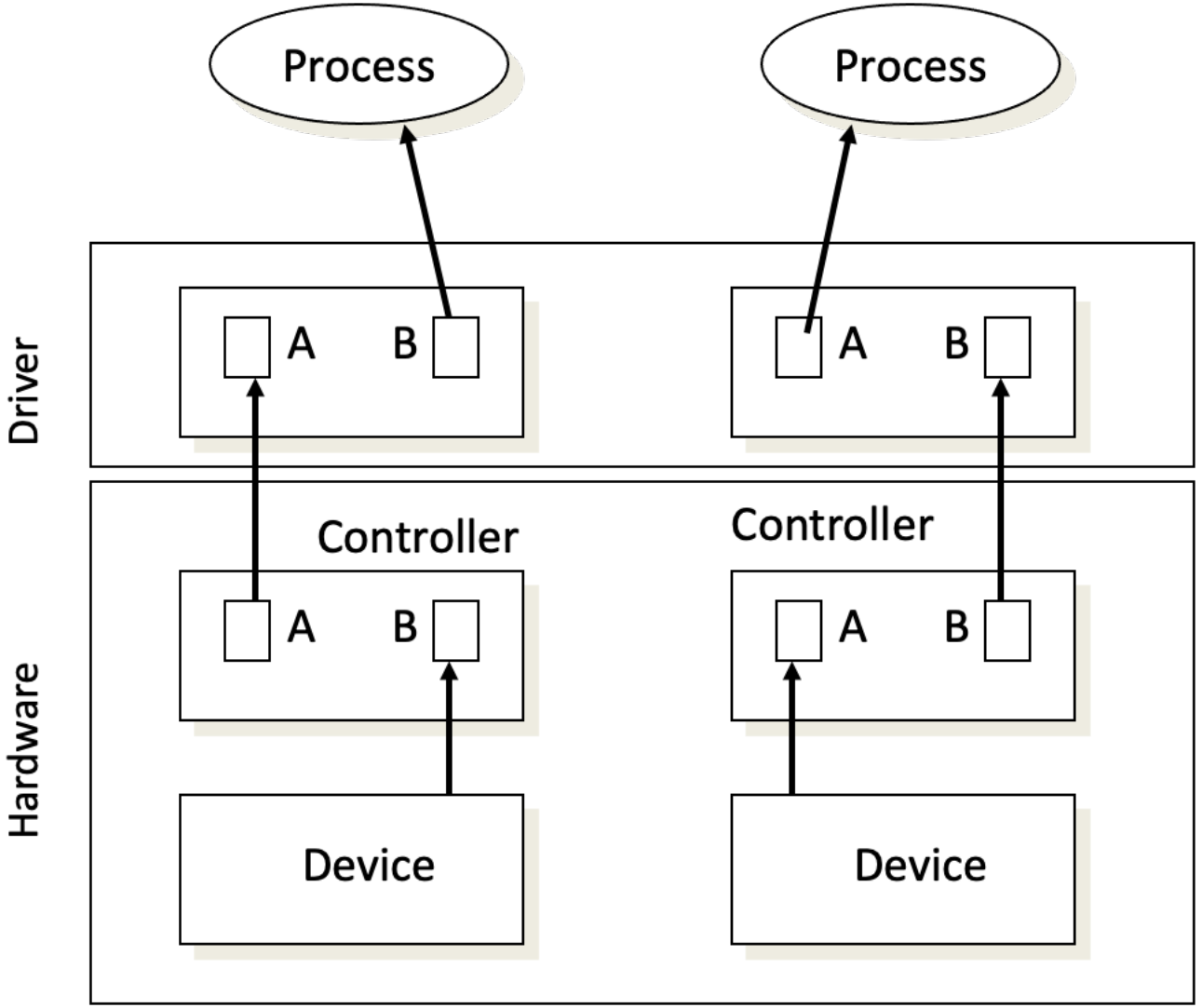
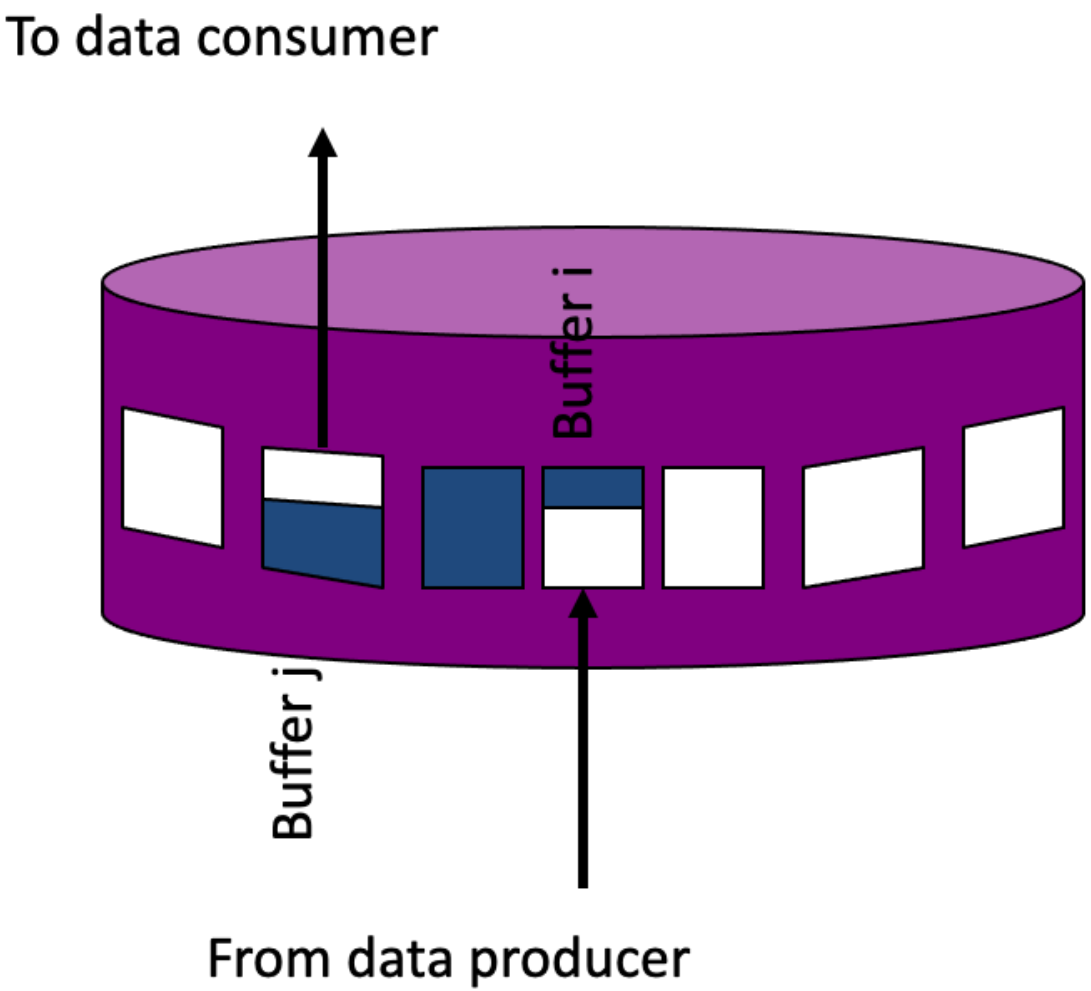


Figure 5-15. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Double Buffering in the Driver



Circular Buffering



Error Handling

- OS can recover from disk read, device unavailable, transient write failures
  - Retry a read or write, for example
  - Some systems more advanced – Solaris FMA, AIX
  - Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

Synchronous and Asynchronous I/O

- Synchronous I/O
  - Read() or write() will block a user process until its completion
  - Easy to use and understand
  - Blocking versus non-blocking variants
- Asynchronous I/O
  - Process runs while I/O executes
  - Let user process do other things before I/O completion
  - I/O completion will notify the user process

User-Space I/O Software

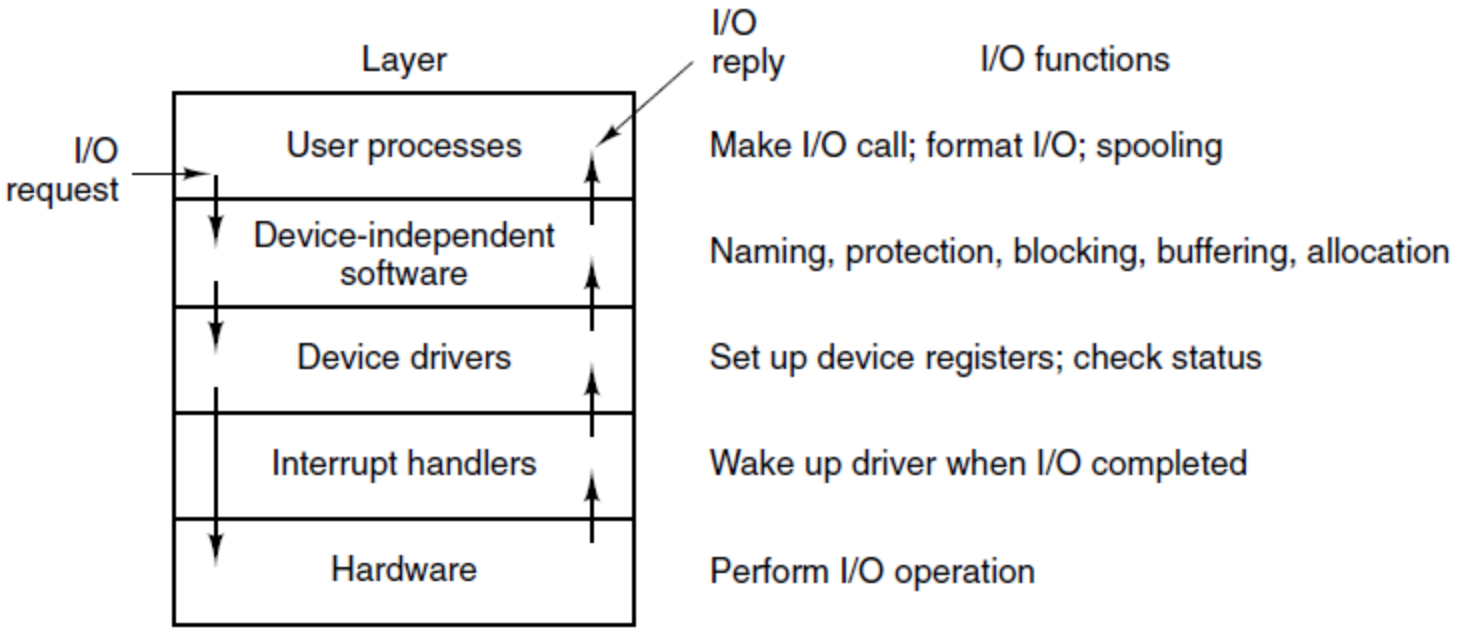


Figure 5-17. Layers of the I/O system and the main functions of each layer.

Part 3 Outlines

- Magnetic Disks
- RAID
- Disk formatting
- Disk Arm Scheduling Algorithms
- Error Handling
- Stable Storage

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.

Magnetic Disks (2)

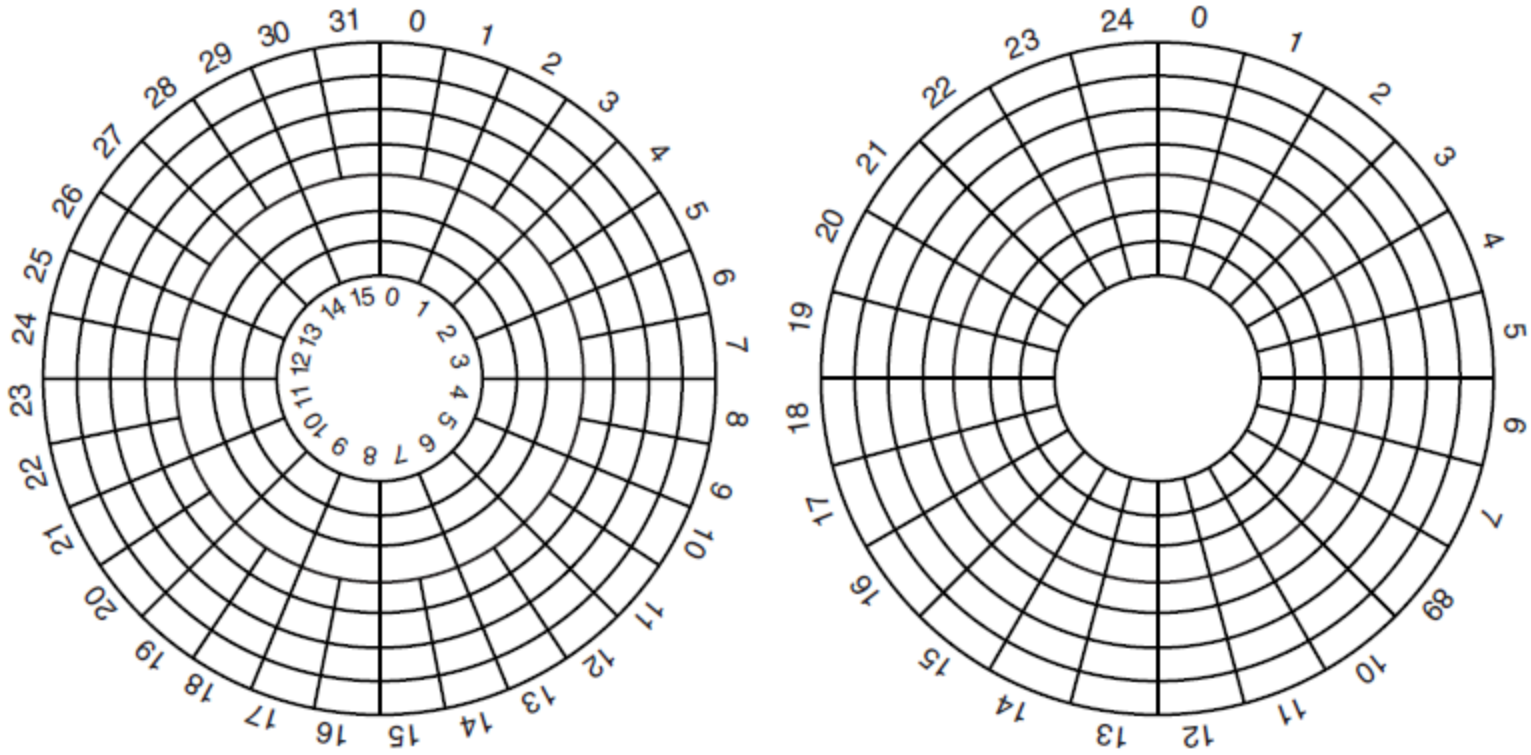


Figure 5-19. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

RAID (1)

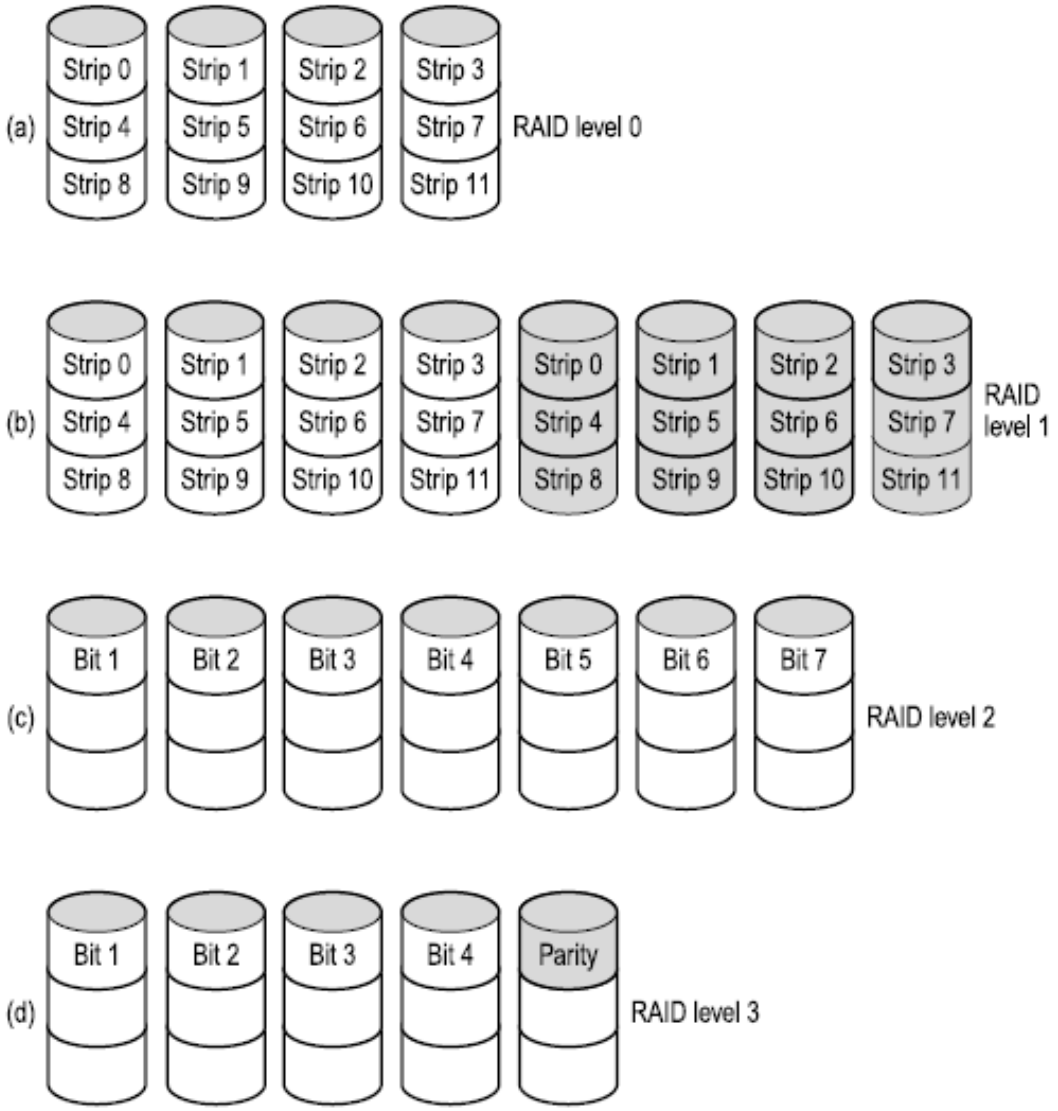


Figure 5-20. RAID levels 0 through 3. Backup and parity drives are shown shaded.

RAID (2)

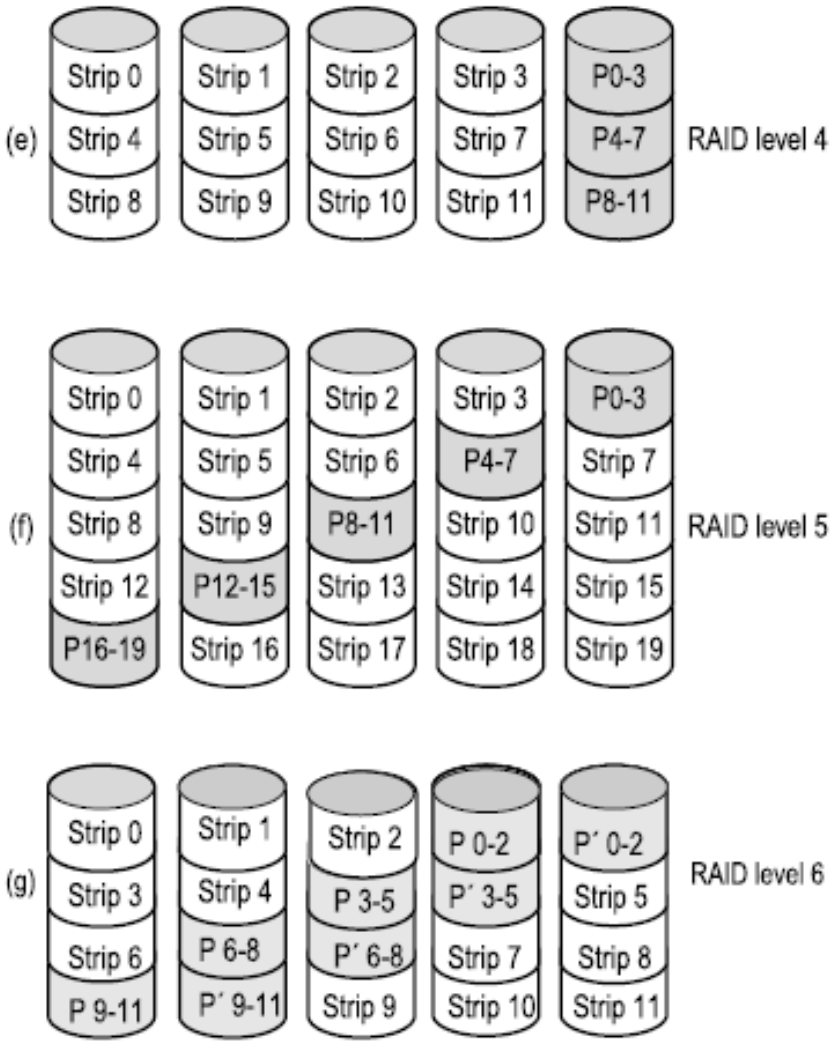


Figure 5-20. RAID levels 4 through 6. Backup and parity drives are shown shaded.

Disk Formatting (1)

Figure 5-21. A disk sector.

Disk Formatting (2)

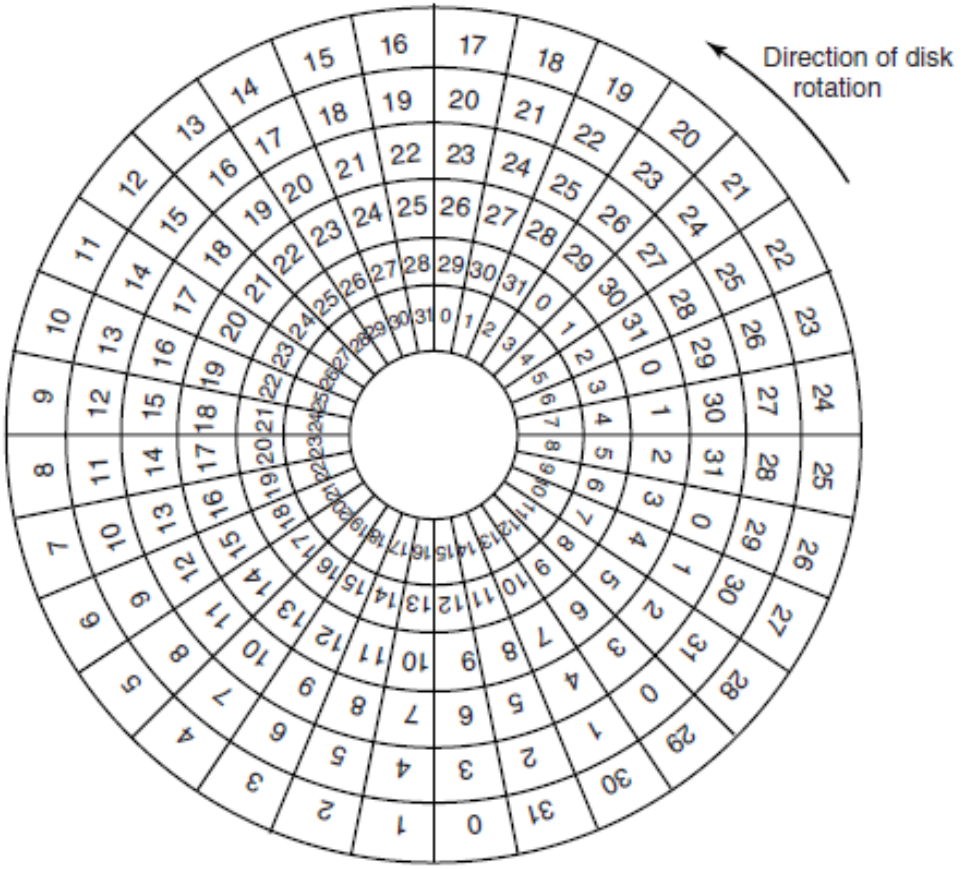


Figure 5-22. An illustration of cylinder skew.

Disk Formatting (3)

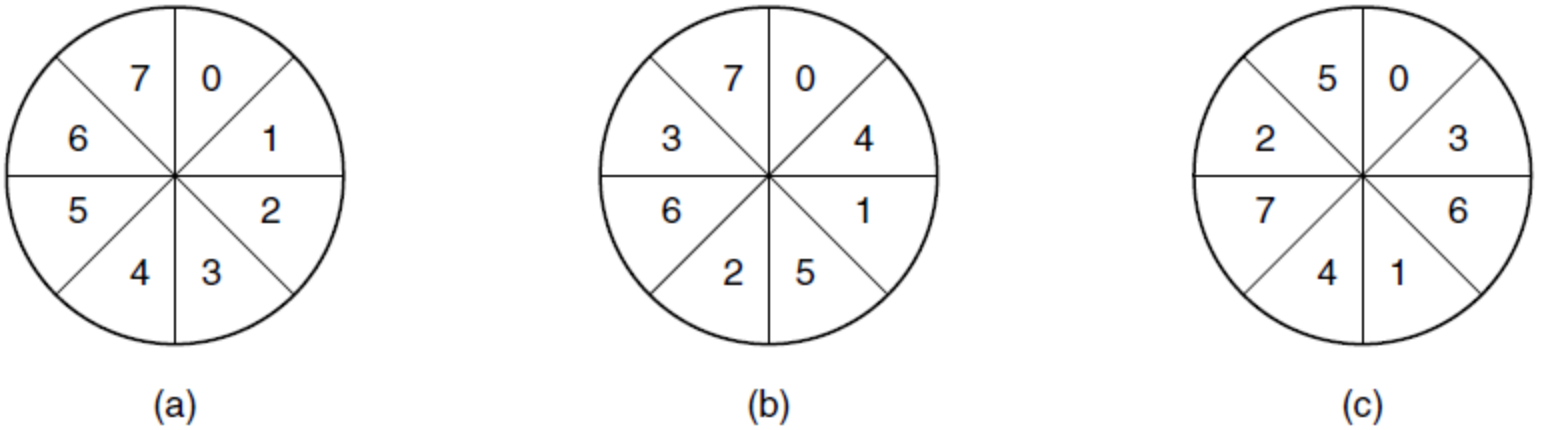


Figure 5-23. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

Disk Arm Scheduling Algorithms (1)

Factors of a disk block read/write:

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

Disk Arm Scheduling Algorithms (2)

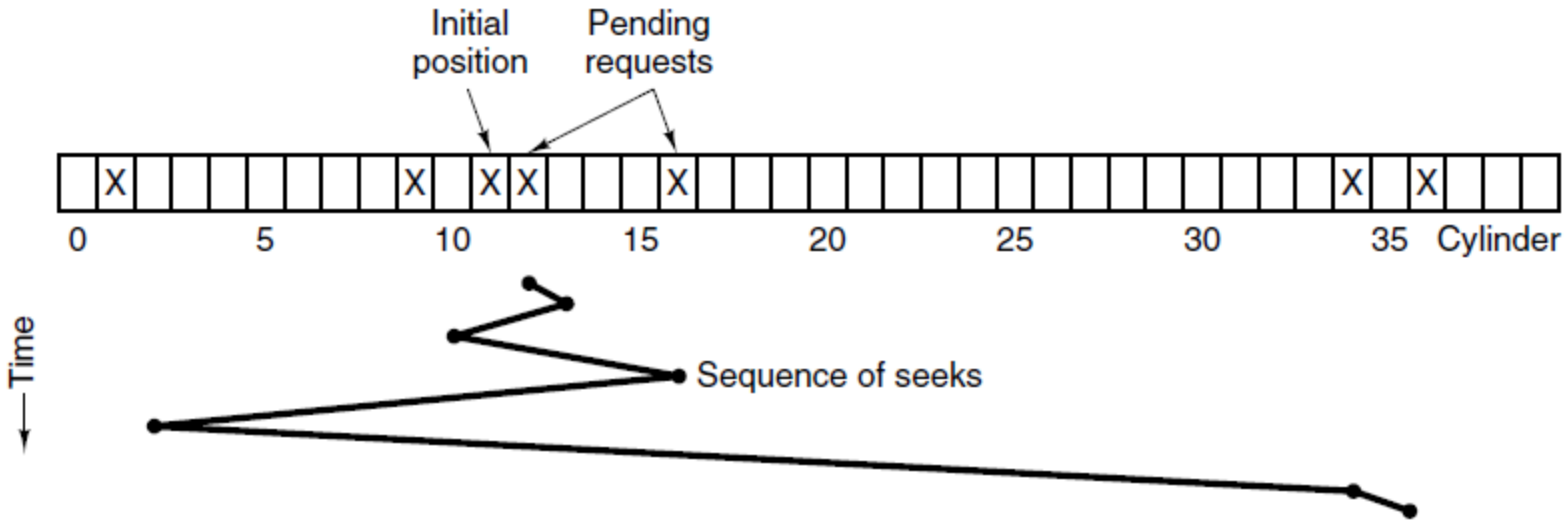


Figure 5-24. Shortest Seek First (SSF) disk scheduling algorithm.

Disk Arm Scheduling Algorithms (3)

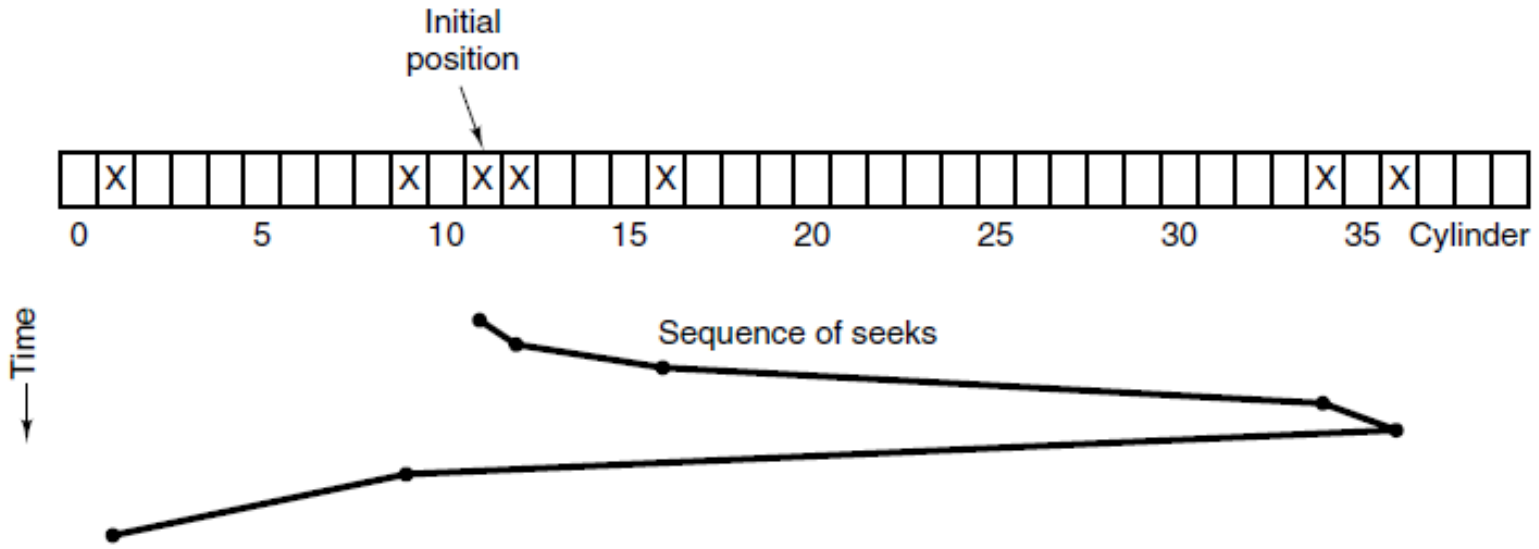


Figure 5-25. The elevator algorithm for scheduling disk requests.

Error Handling

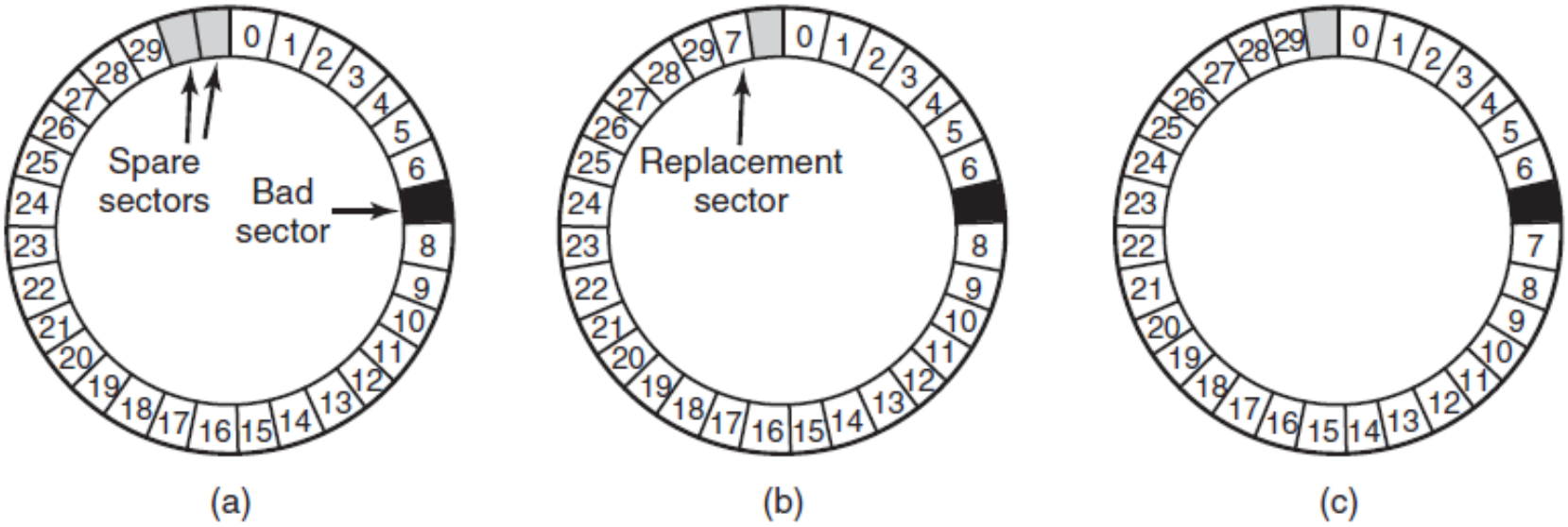


Figure 5-26. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

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)

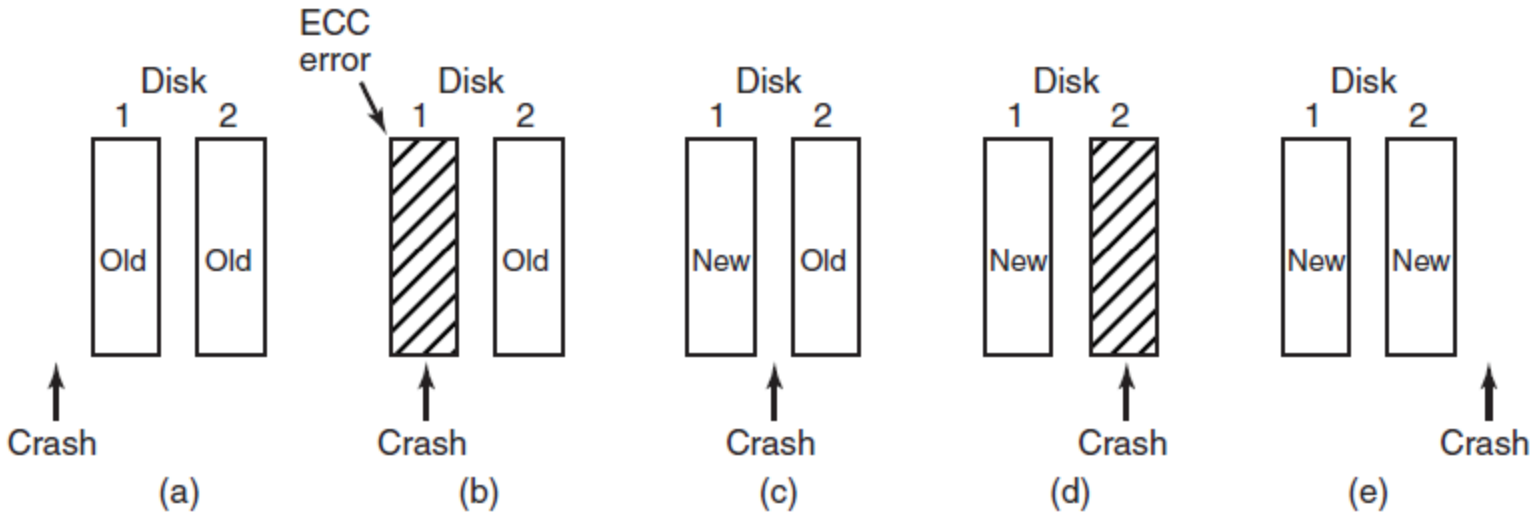


Figure 5-27. Analysis of the influence of crashes on stable writes.

Part 4 Outlines

- Clock
- Keyboard
- Text Windows
- Graphical User Interfaces

Clock Hardware

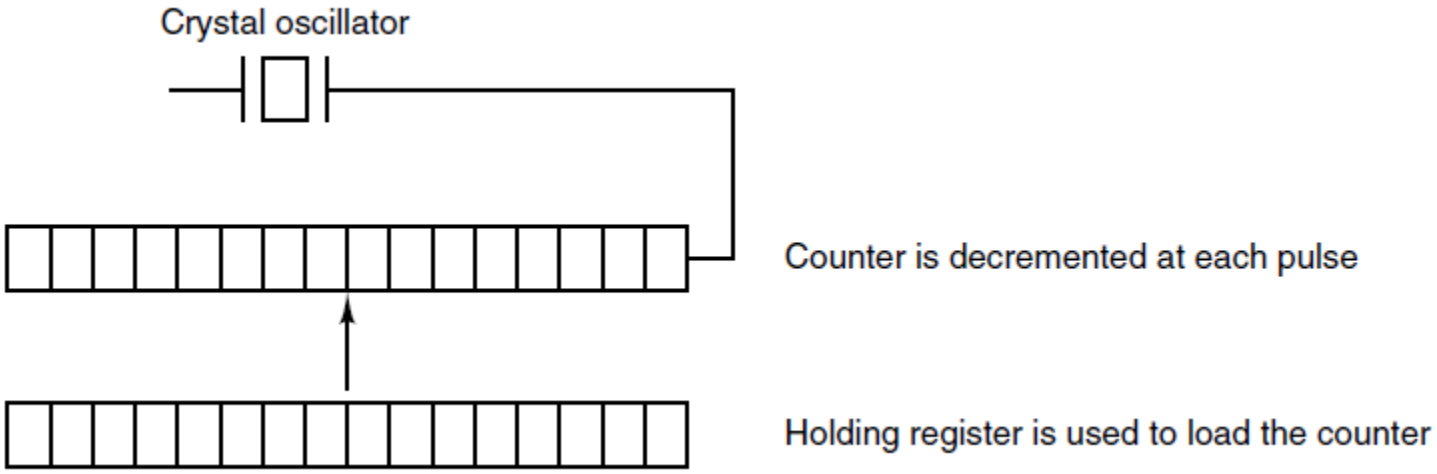


Figure 5-28. A programmable clock.

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)

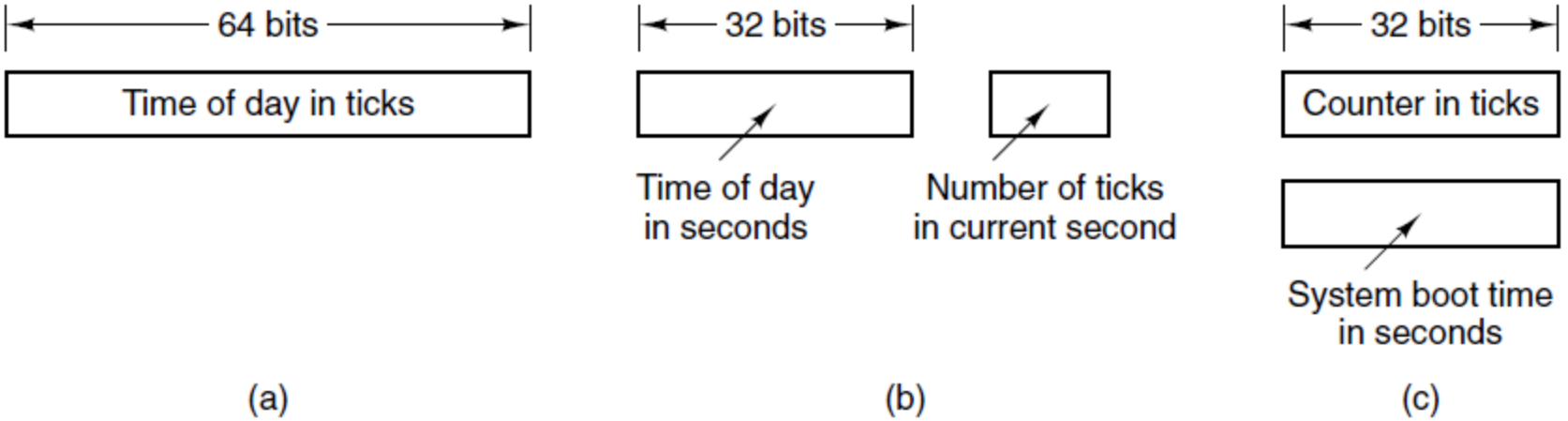


Figure 5-29. Three ways to maintain the time of day.

Clock Software (3)

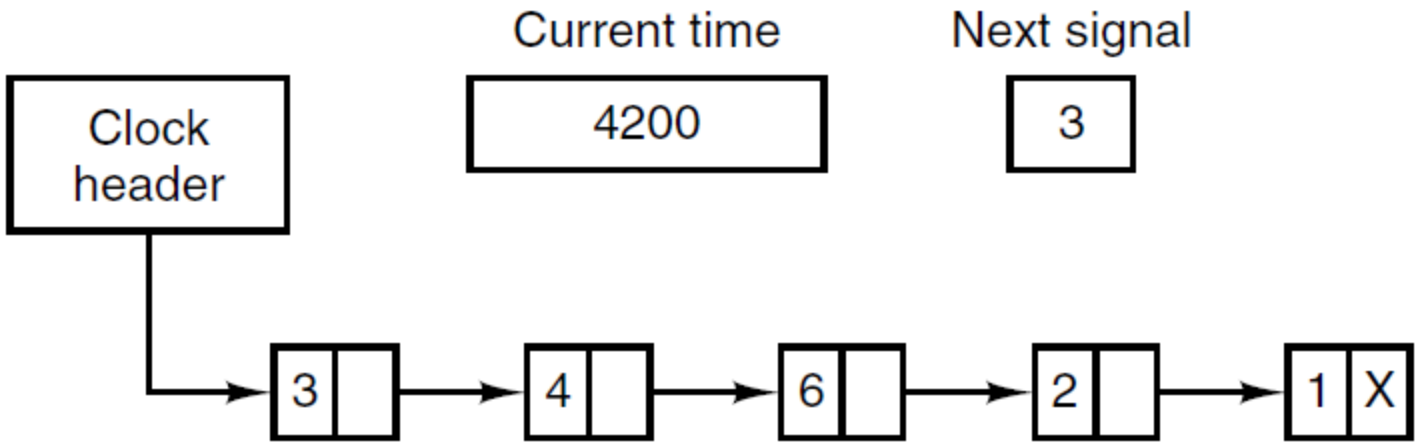


Figure 5-30. Simulating multiple timers with a single clock.

Soft Timers

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)

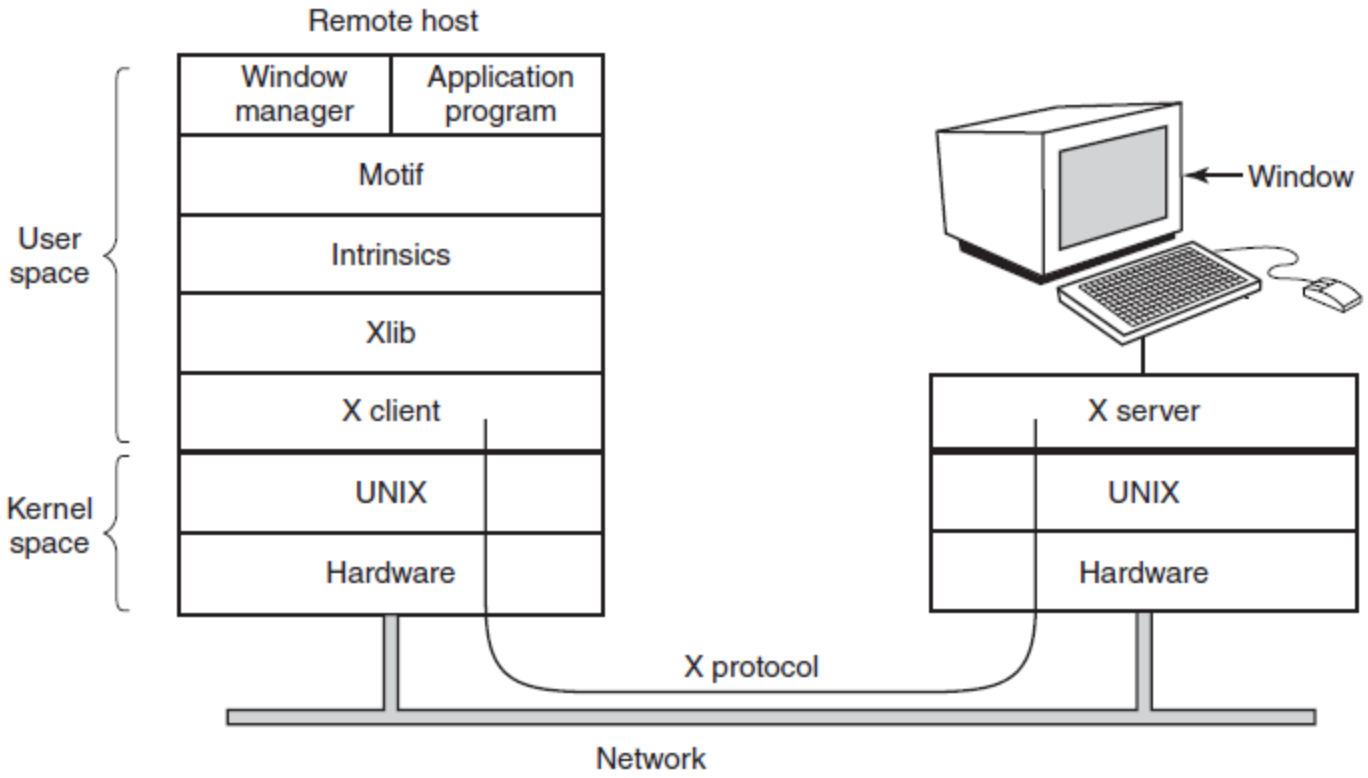


Figure 5-33. Clients and servers in the M.I.T. X Window System.

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) {
                case Expose:                    break;  /* repaint window */
```

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)

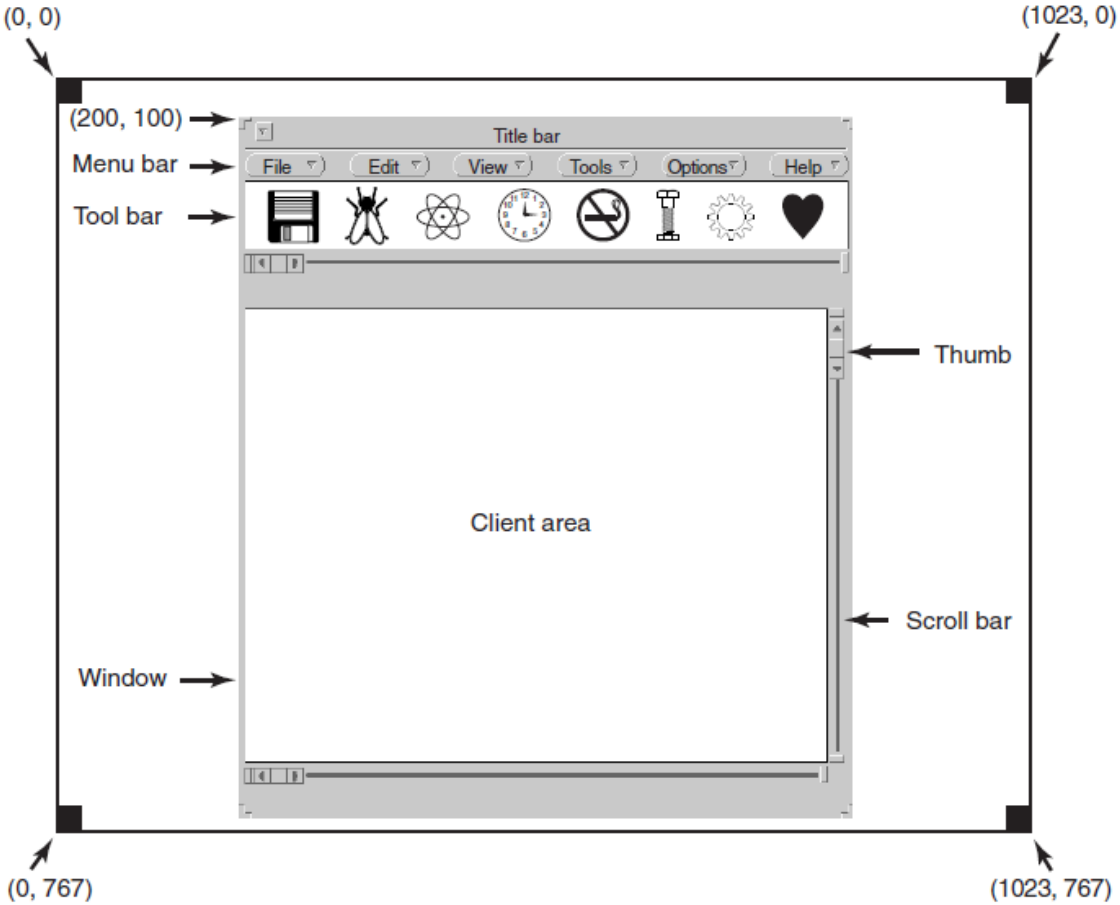


Figure 5-35. A sample window located at (200, 100) on an XGA display.

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)

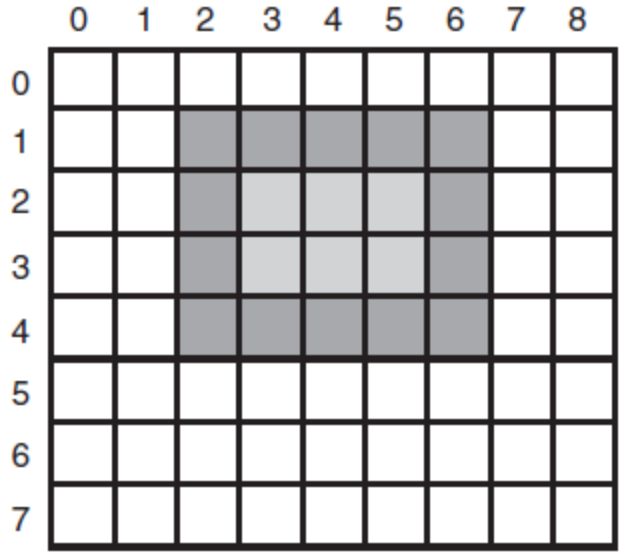


Figure 5-37. An example rectangle drawn using *Rectangle*. Each box represents one pixel.

Bitmaps

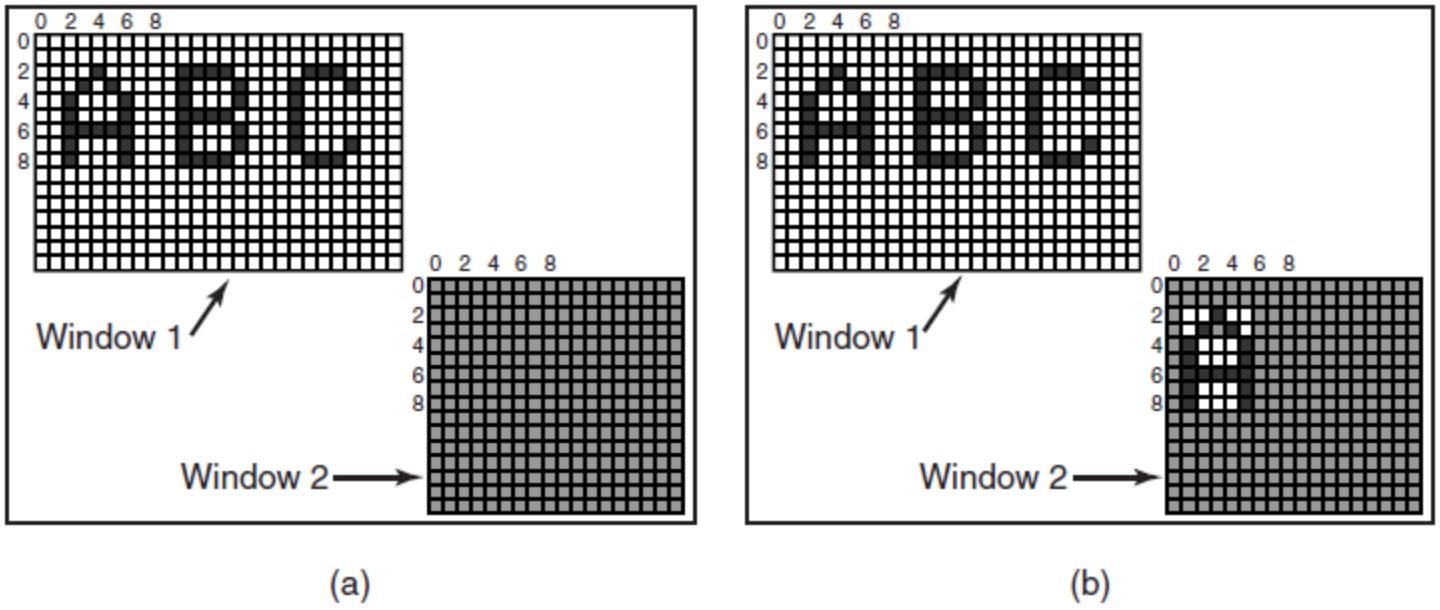


Figure 5-38. Copying bitmaps using BitBlt.
(a) Before. (b) After.

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

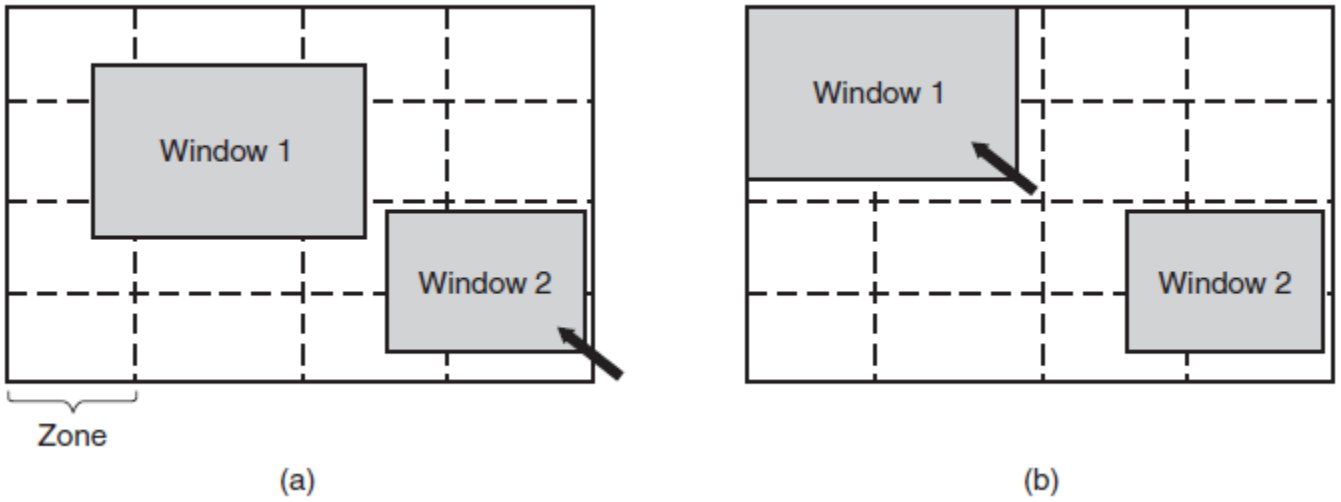


Figure 5-41. The use of zones for backlighting the display.
(a) When window 2 is selected it is not moved.
(b) When window 1 is selected, it moves to reduce the number of zones illuminated.

Operating System Issues
The CPU

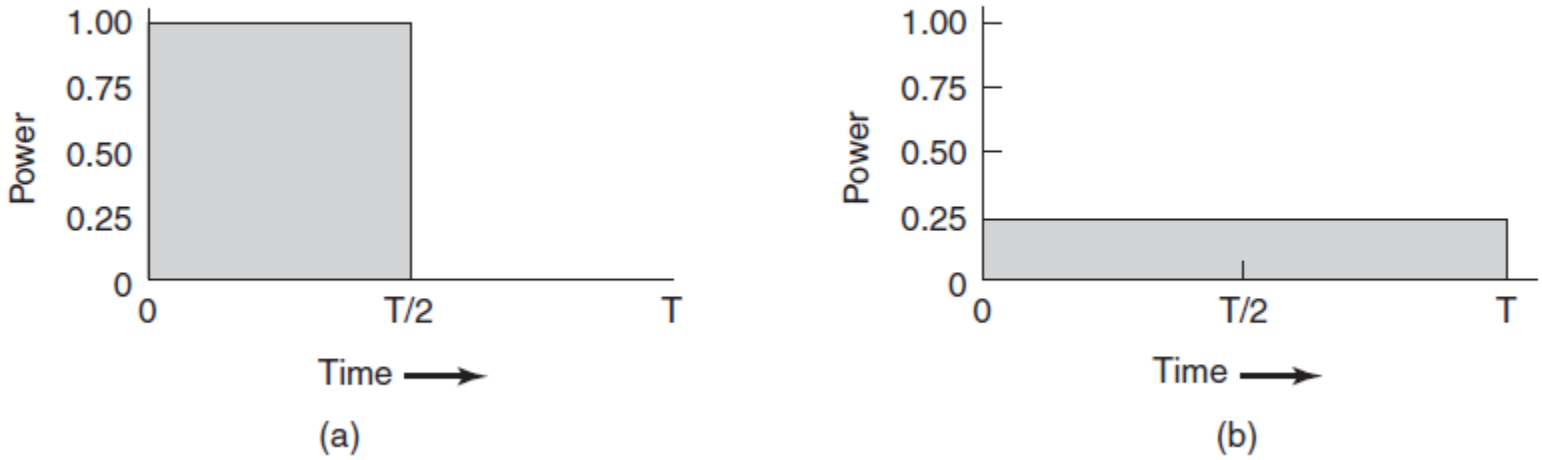


Figure 5-42. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four

End

Chapter 5