Input/Output

Chapter 5

I/O Devices (1)

- 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 (2)

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

Figure 5-1. Some typical device, network, and bus data rates.

Memory-Mapped I/O (1)

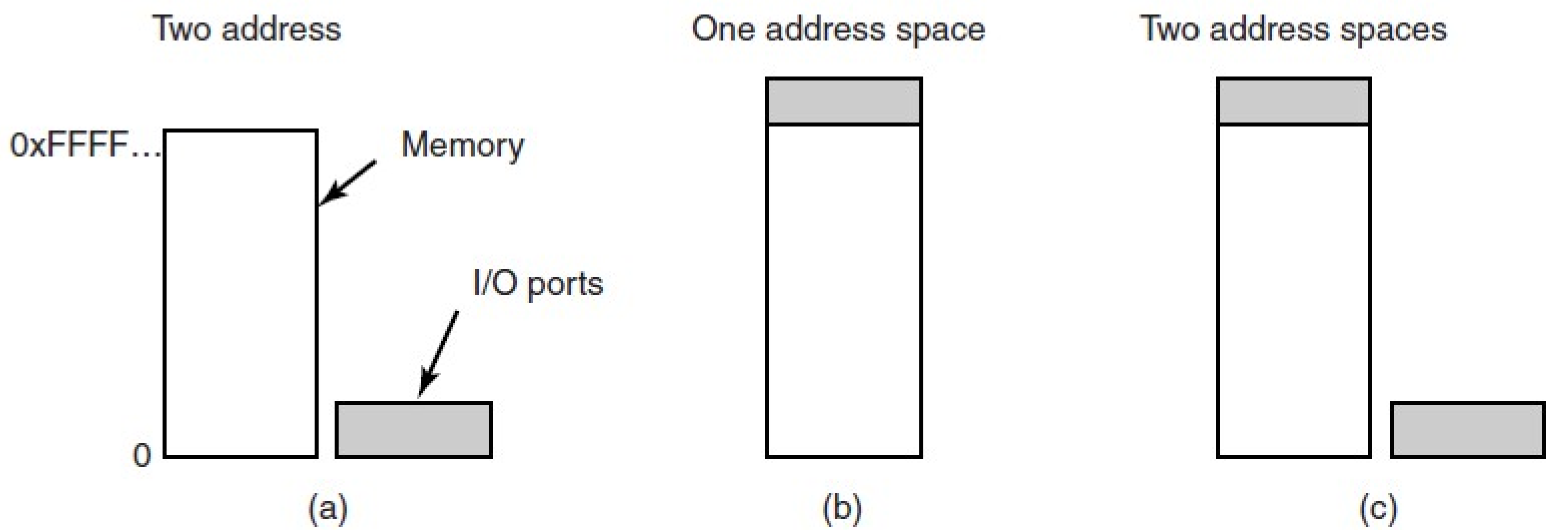


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

Memory-Mapped I/O (2)

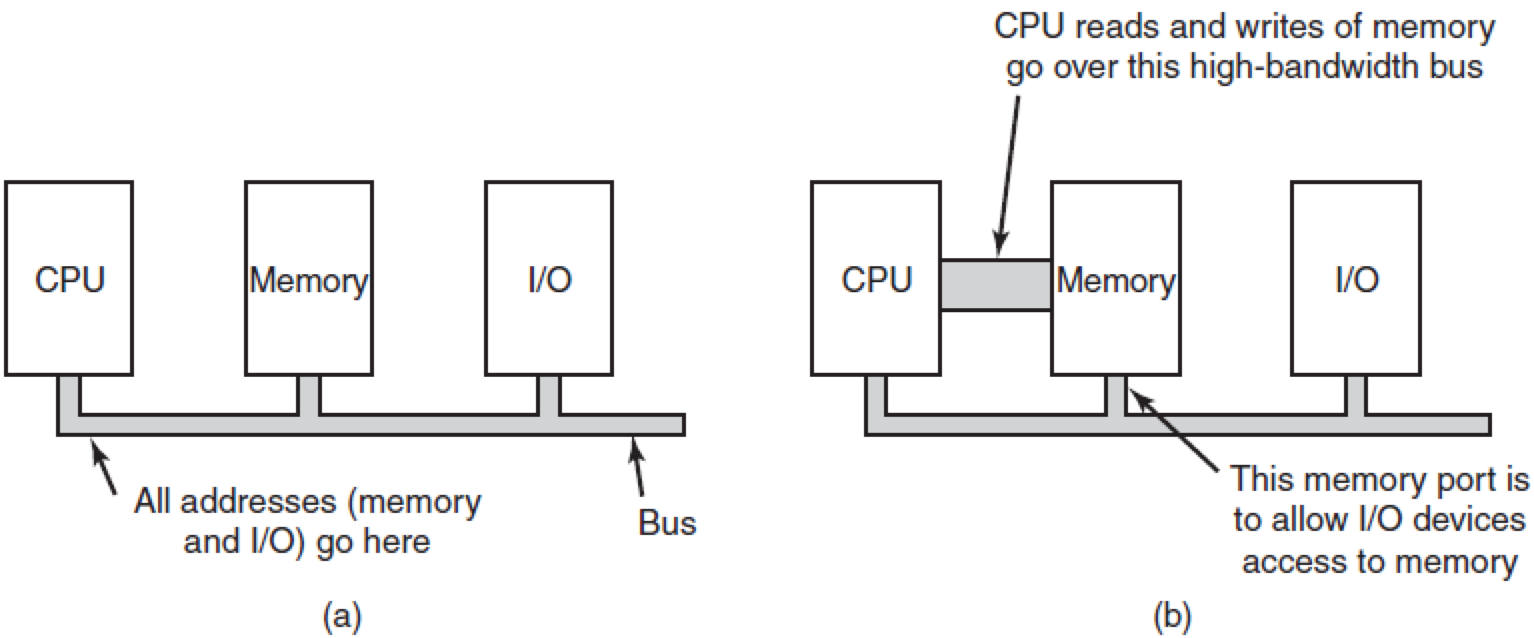


Figure 5-3. (a) A single-bus architecture.
(b) A dual-bus memory architecture.

Direct Memory Access

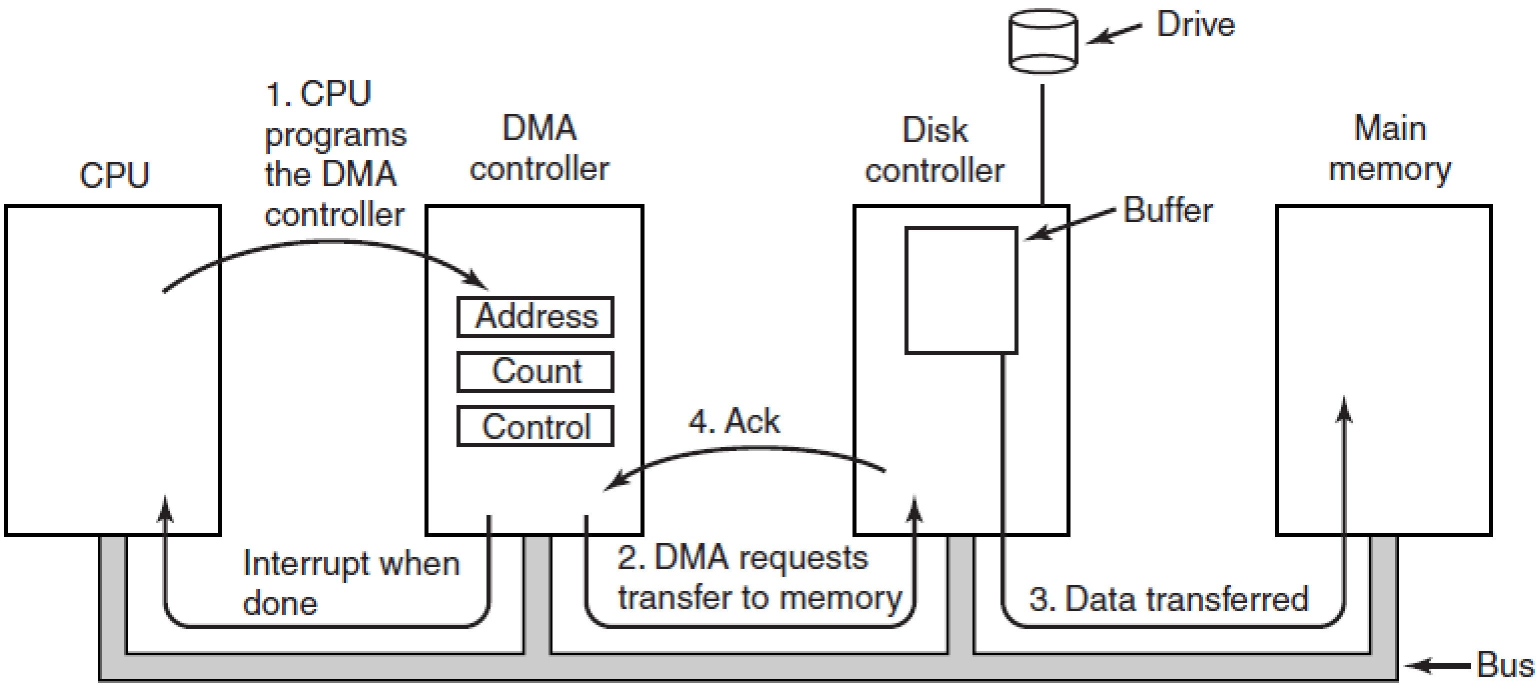


Figure 5-4. Operation of a DMA transfer.

Interrupts Revisited

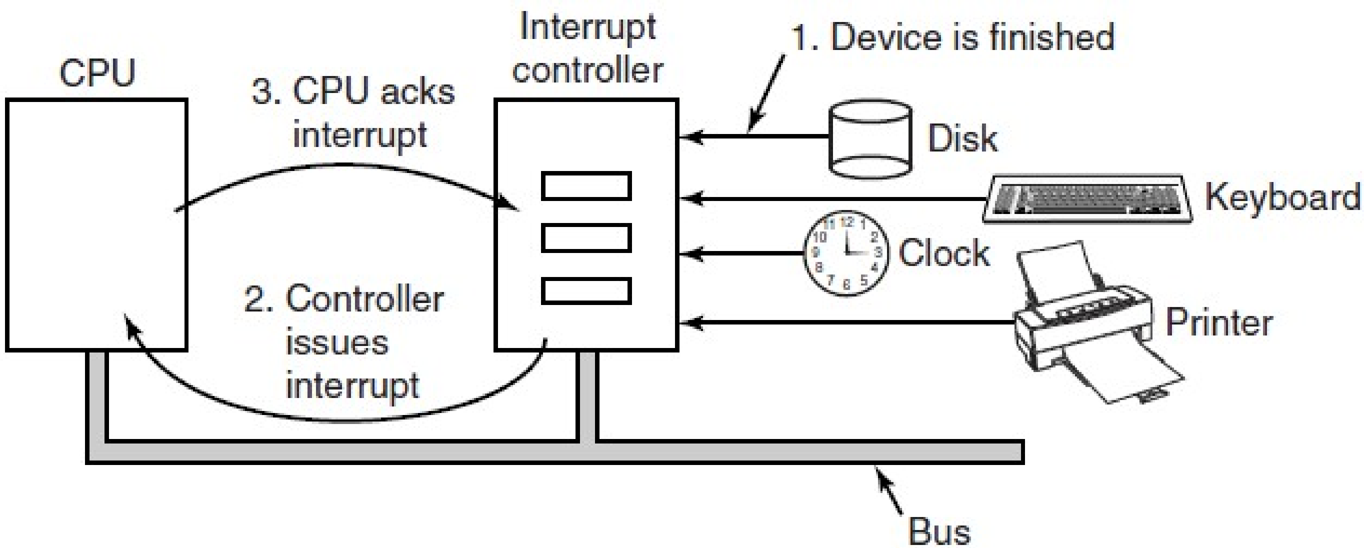


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

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

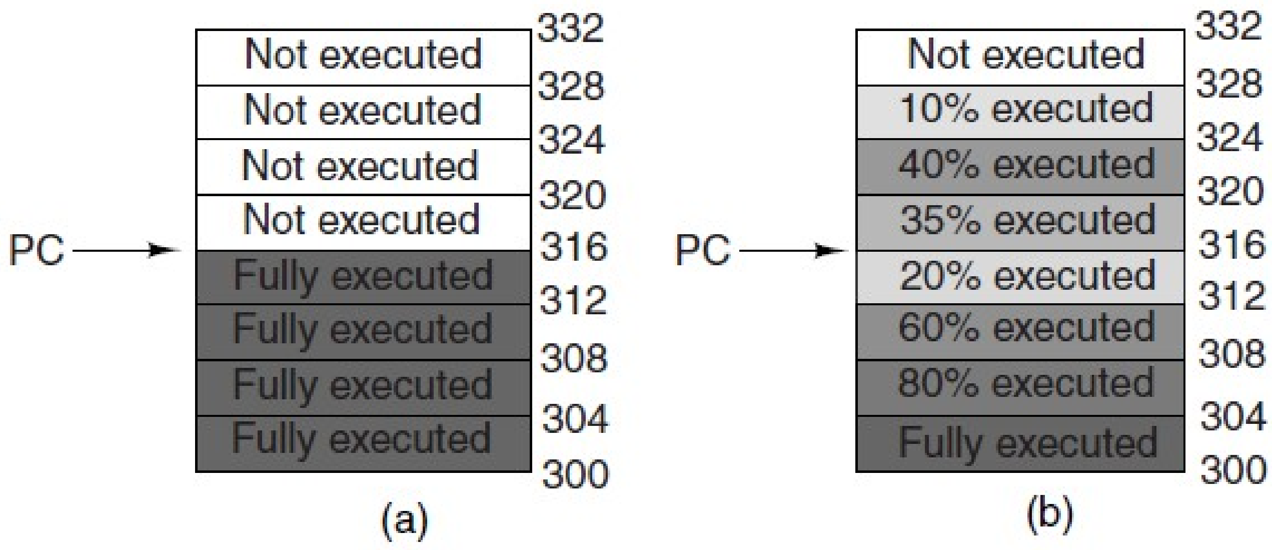


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

Goals of the I/O Software

Issues:

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

Programmed I/O (1)

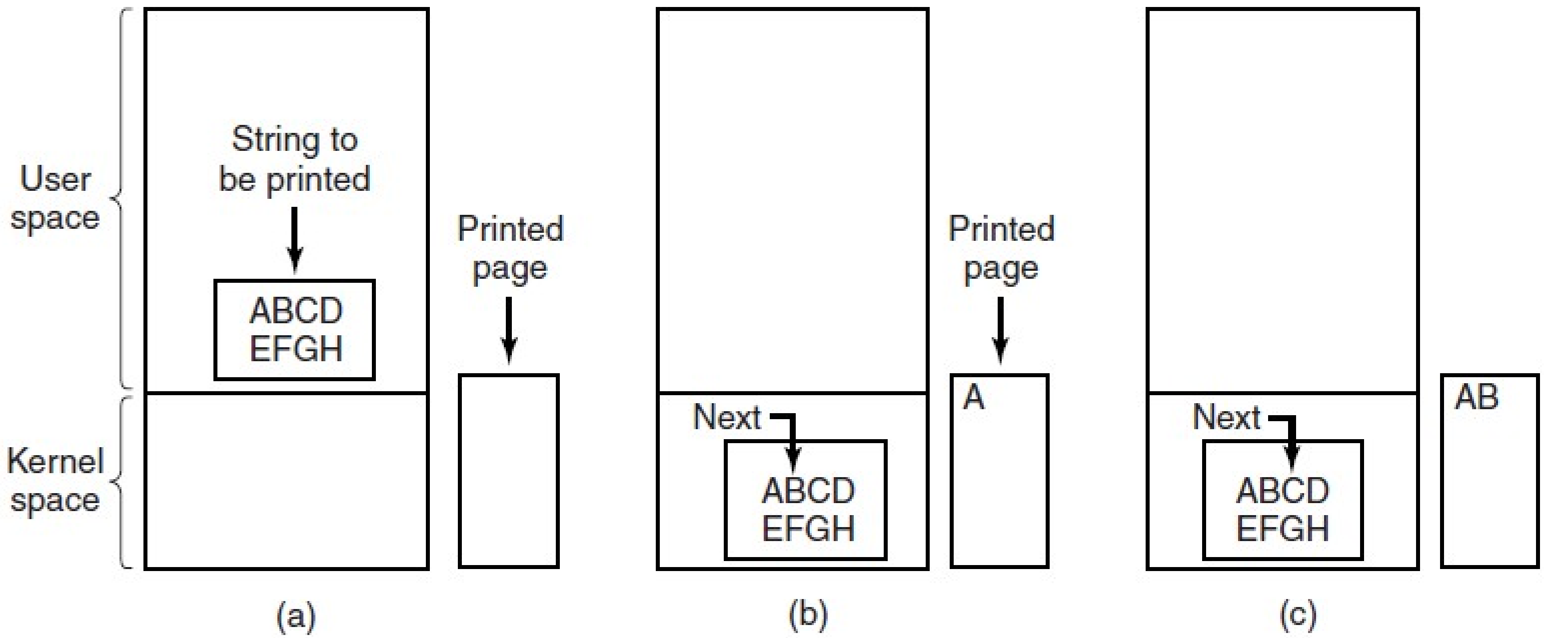


Figure 5-7. Steps in printing a string.

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

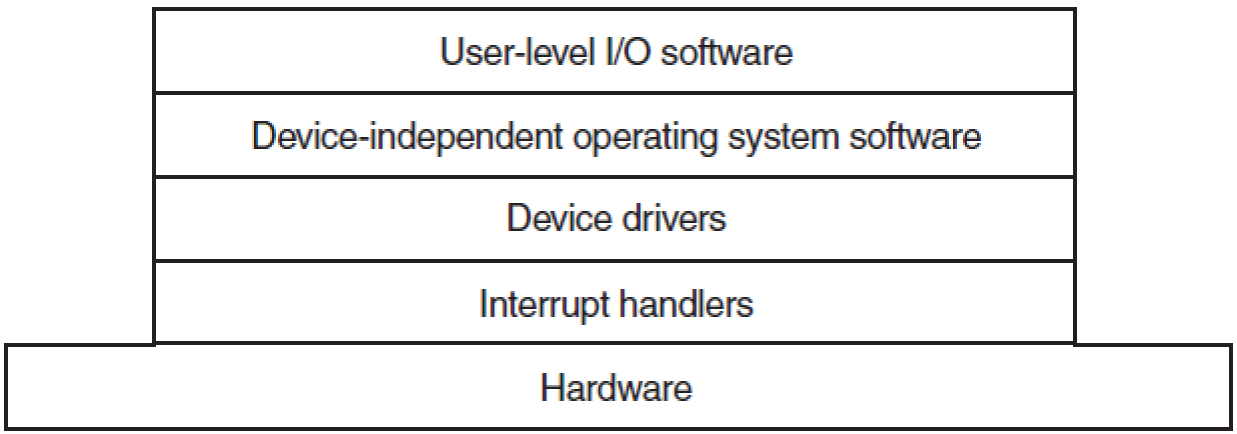


Figure 5-11. Layers of the I/O software system.

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.

Device Drivers

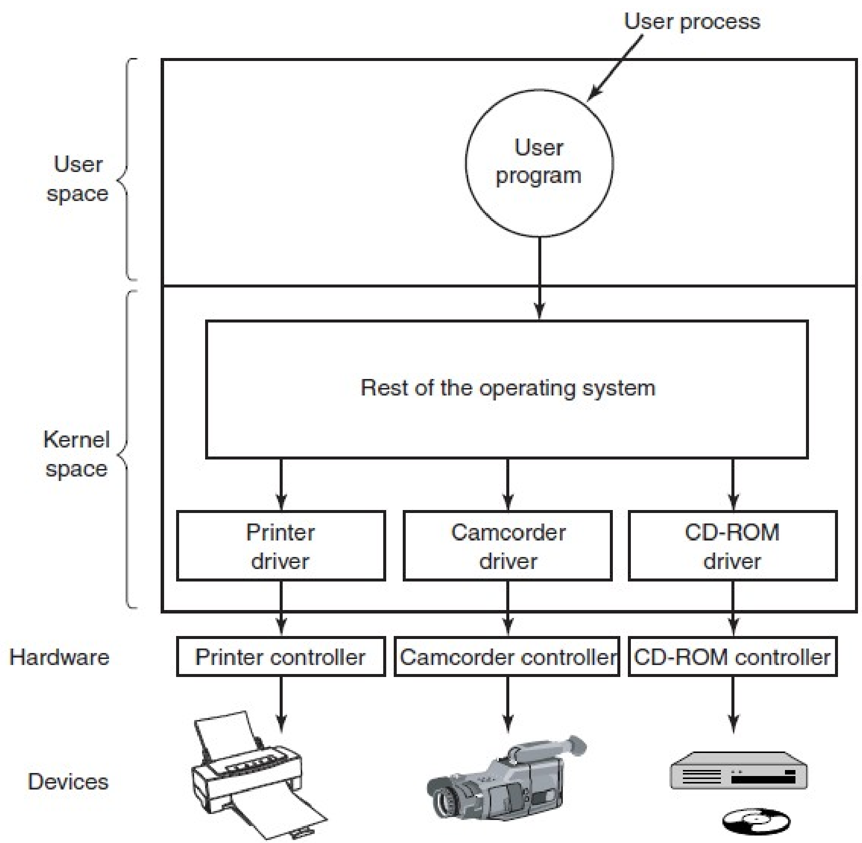


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Device-Independent I/O Software

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

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

Uniform Interfacing for Device Drivers

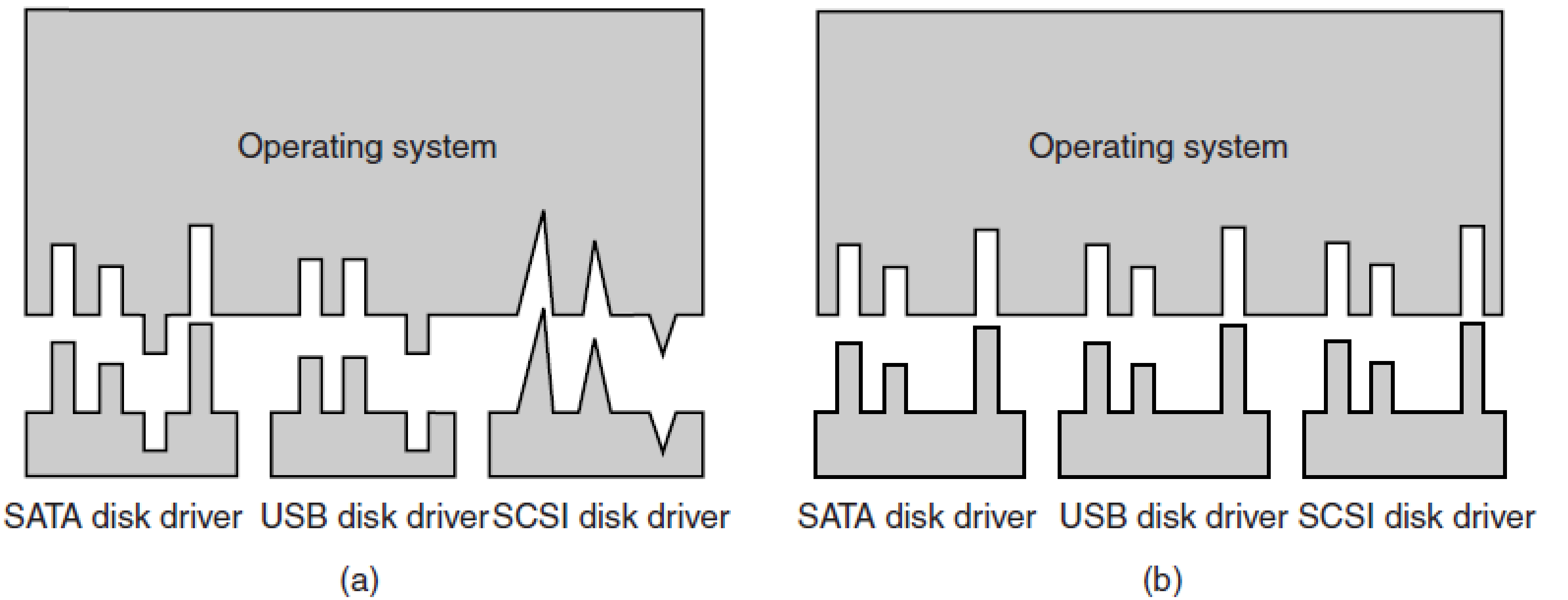


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

Buffering (1)

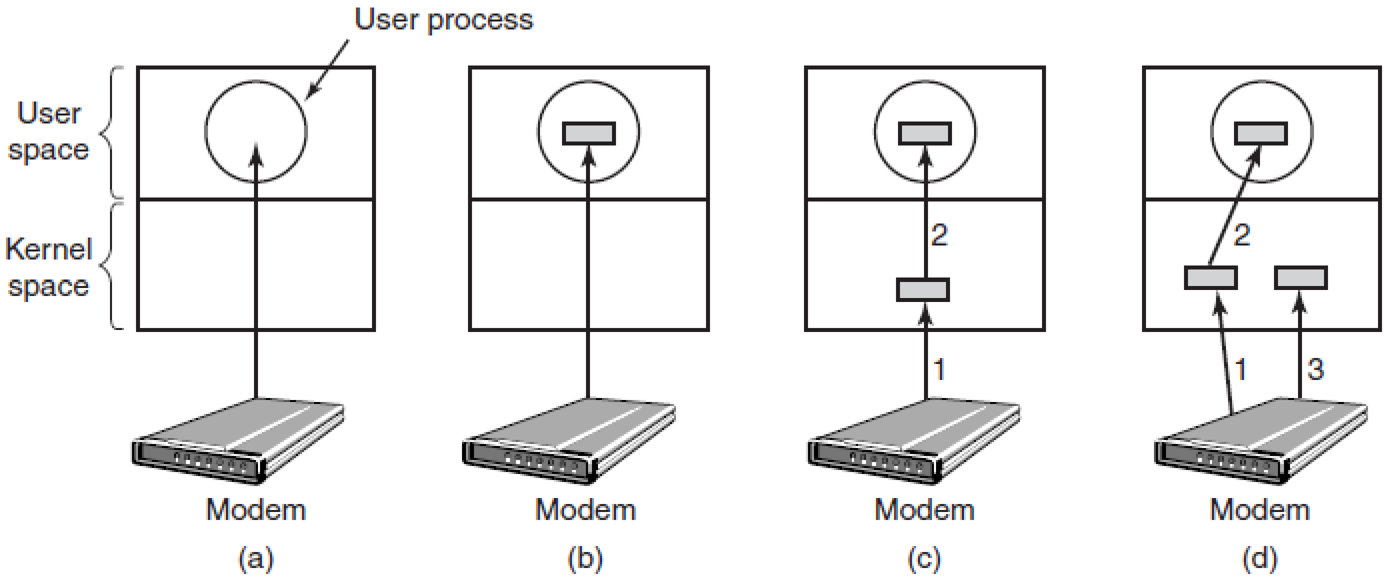


Figure 5-15. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Buffering (2)

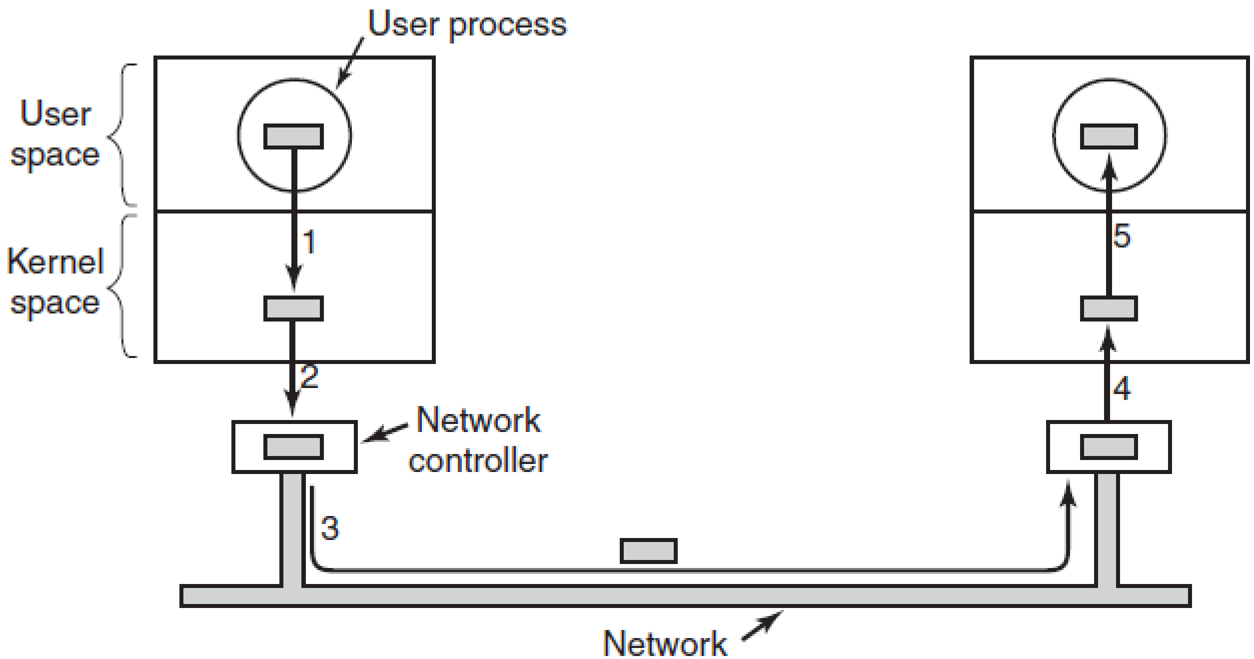


Figure 5-16. Networking may involve many copies of a packet.

User-Space I/O Software

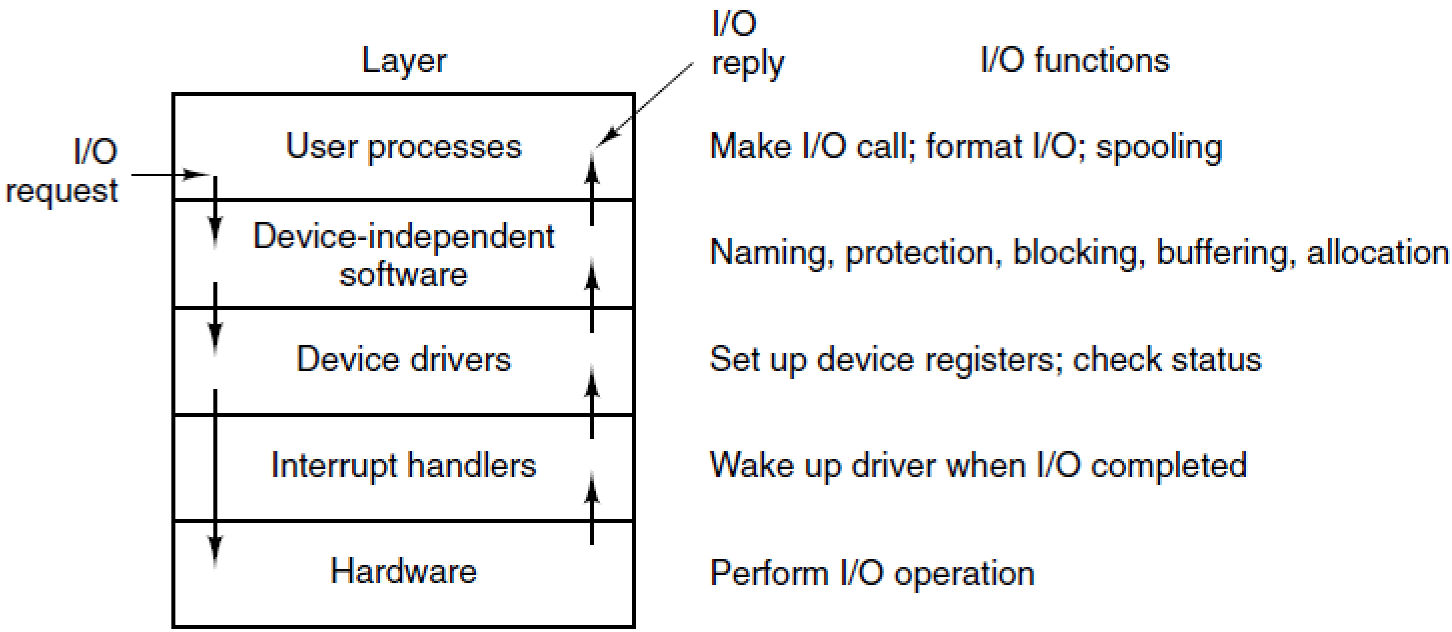


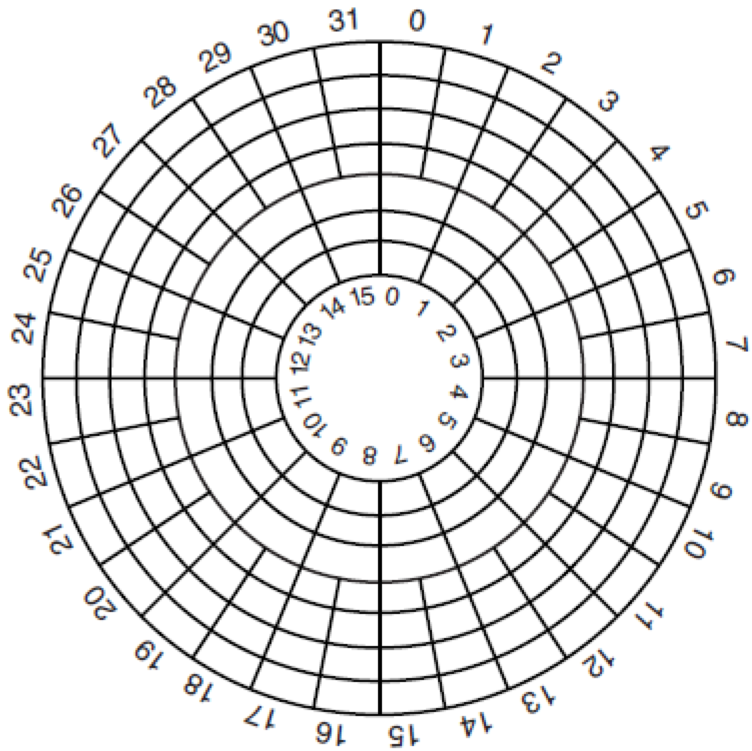
Figure 5-17. Layers of the I/O system and the main functions of each layer.

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.

Magnetic Disks (2)



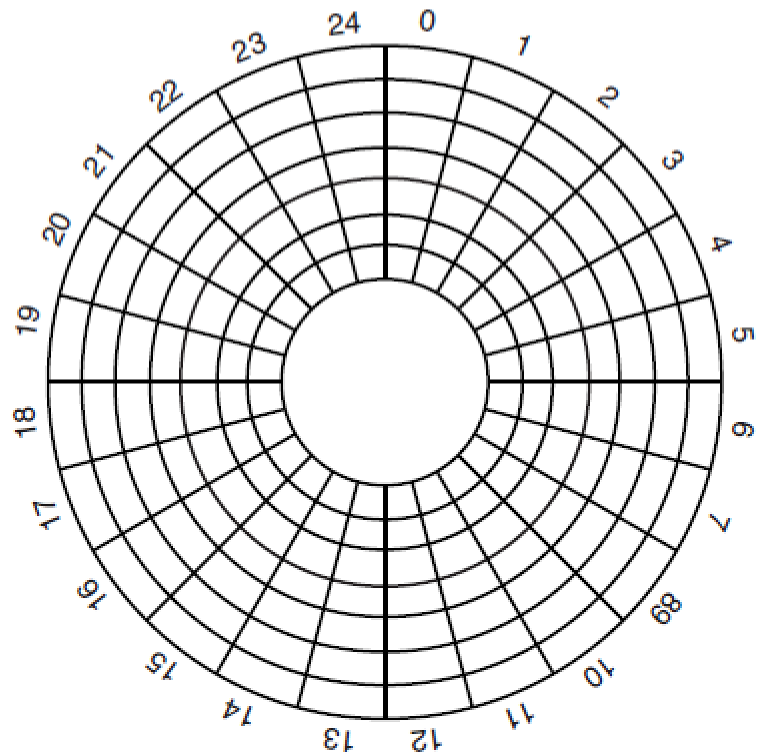


Figure 5-19. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

RAID (1)

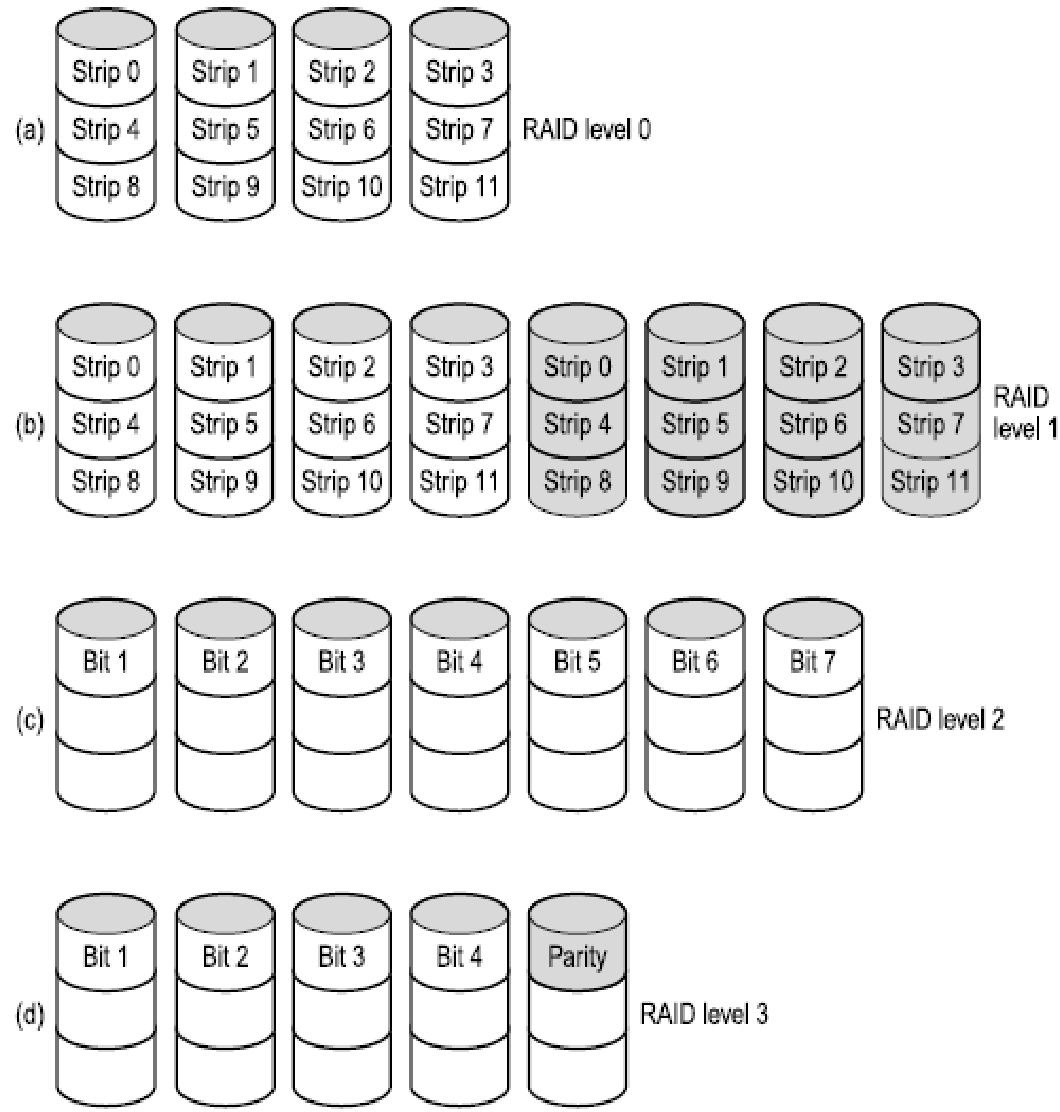


Figure 5-20. RAID levels 0 through 3. Backup and parity drives are shown shaded.

RAID (2)

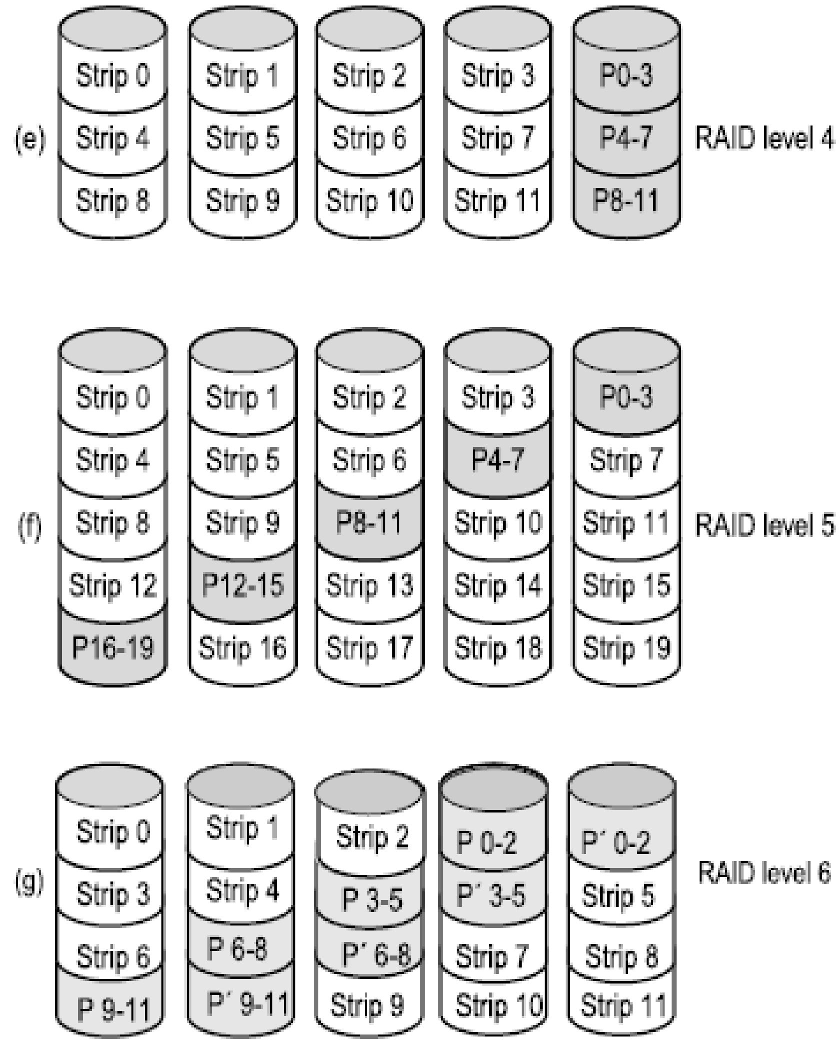


Figure 5-20. RAID levels 4 through 6. Backup and parity drives are shown shaded.

Disk Formatting (1)

Figure 5-21. A disk sector.

Disk Formatting (2)

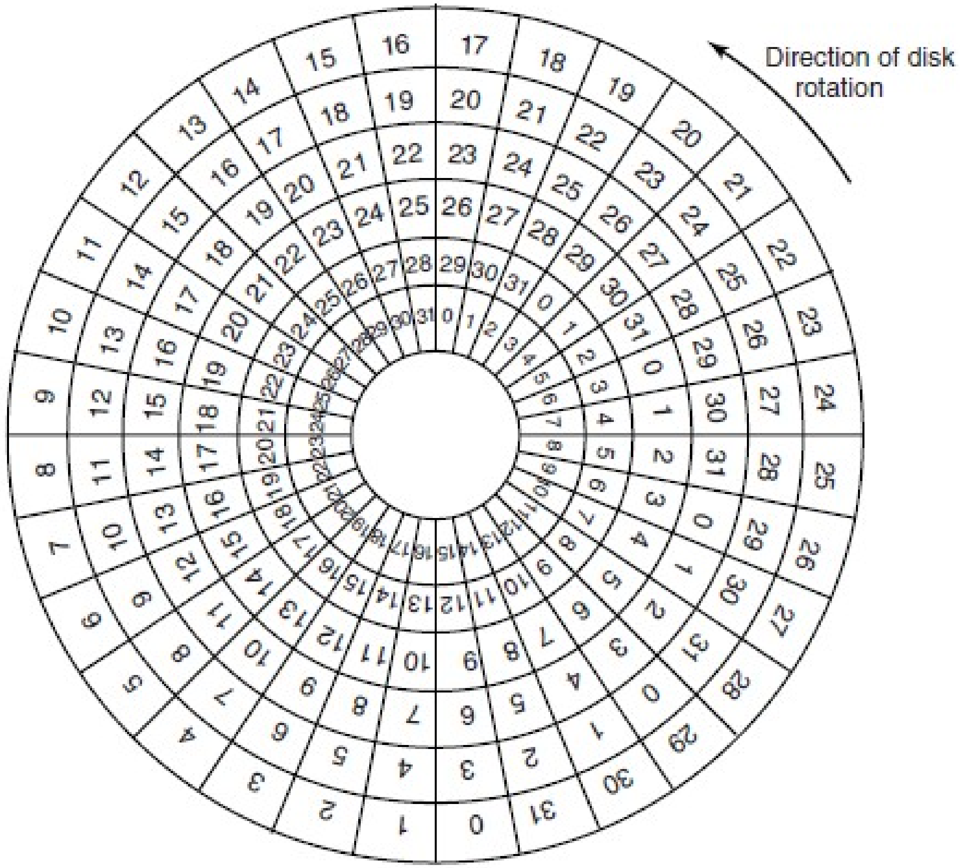


Figure 5-22. An illustration of cylinder skew.

Disk Formatting (3)

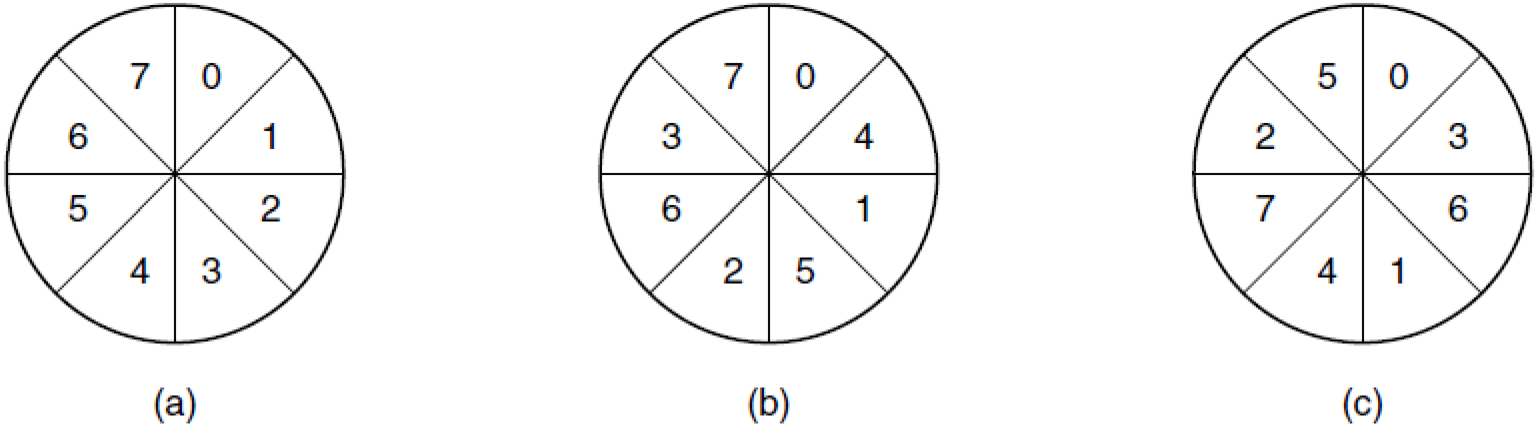


Figure 5-23. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

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)

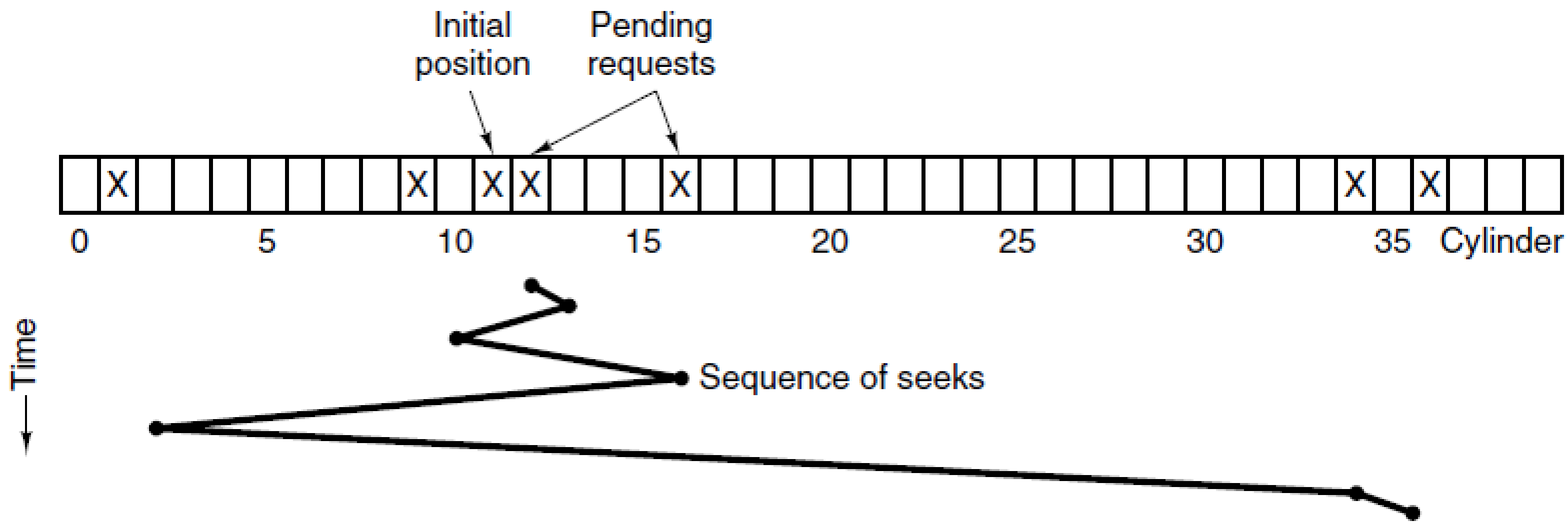


Figure 5-24. Shortest Seek First (SSF) disk scheduling algorithm.

Disk Arm Scheduling Algorithms (3)

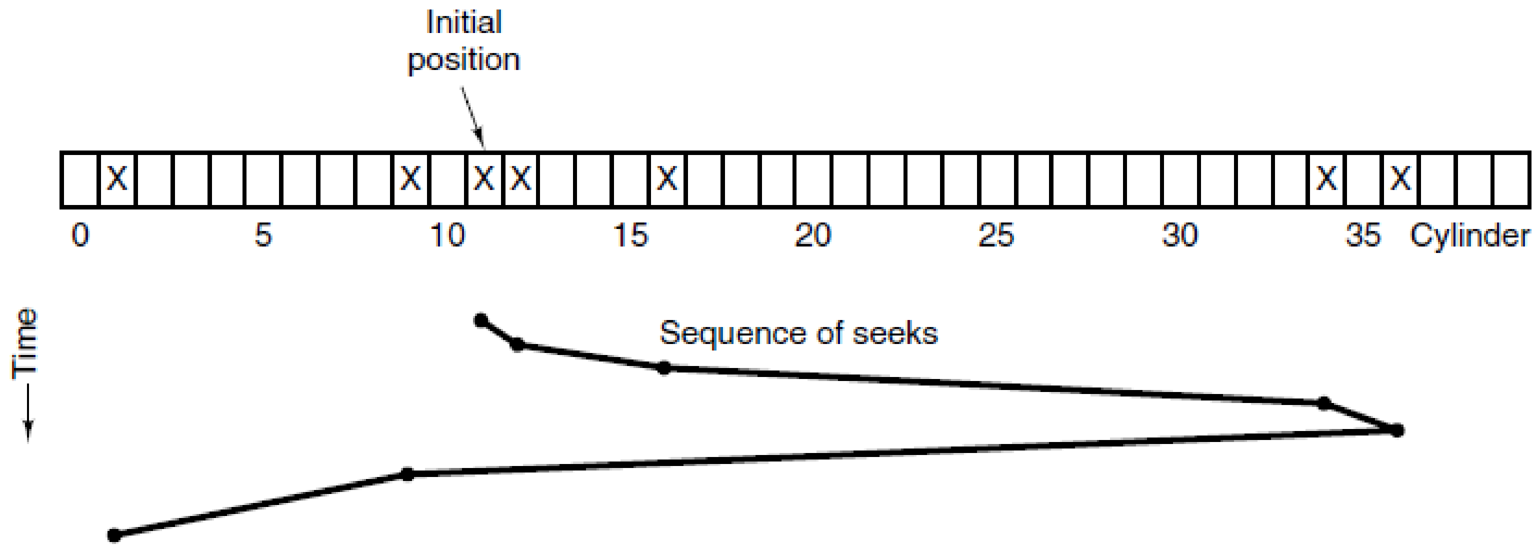


Figure 5-25. The elevator algorithm for scheduling disk requests.

Error Handling

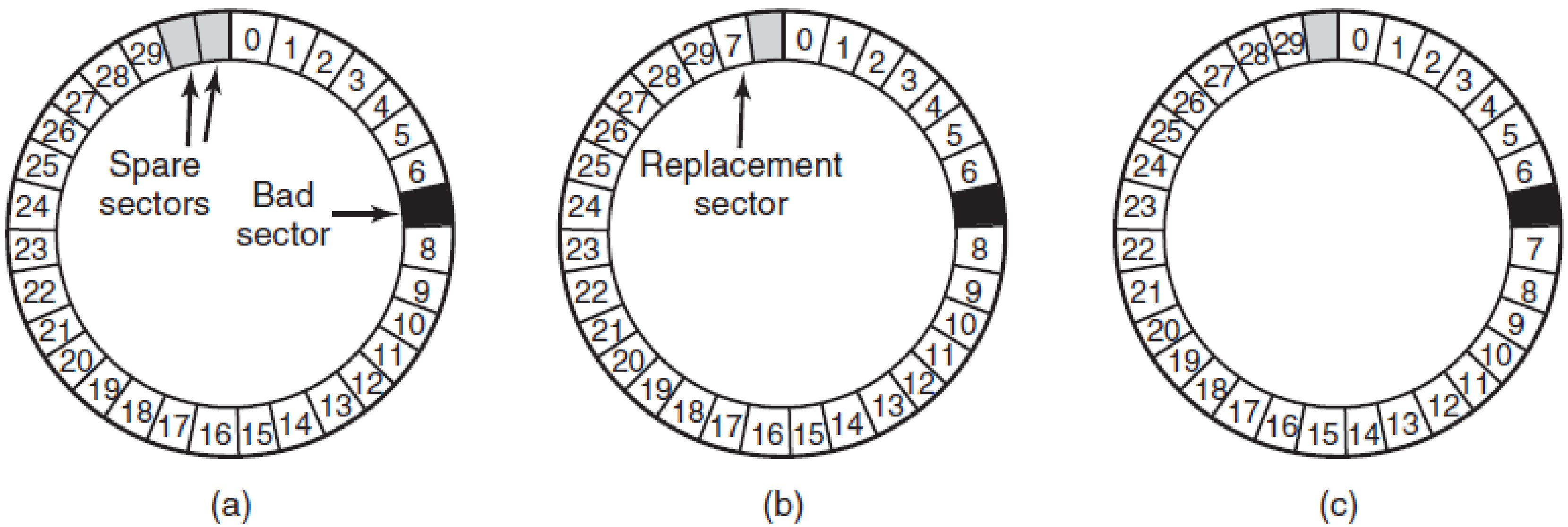


Figure 5-26. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

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)

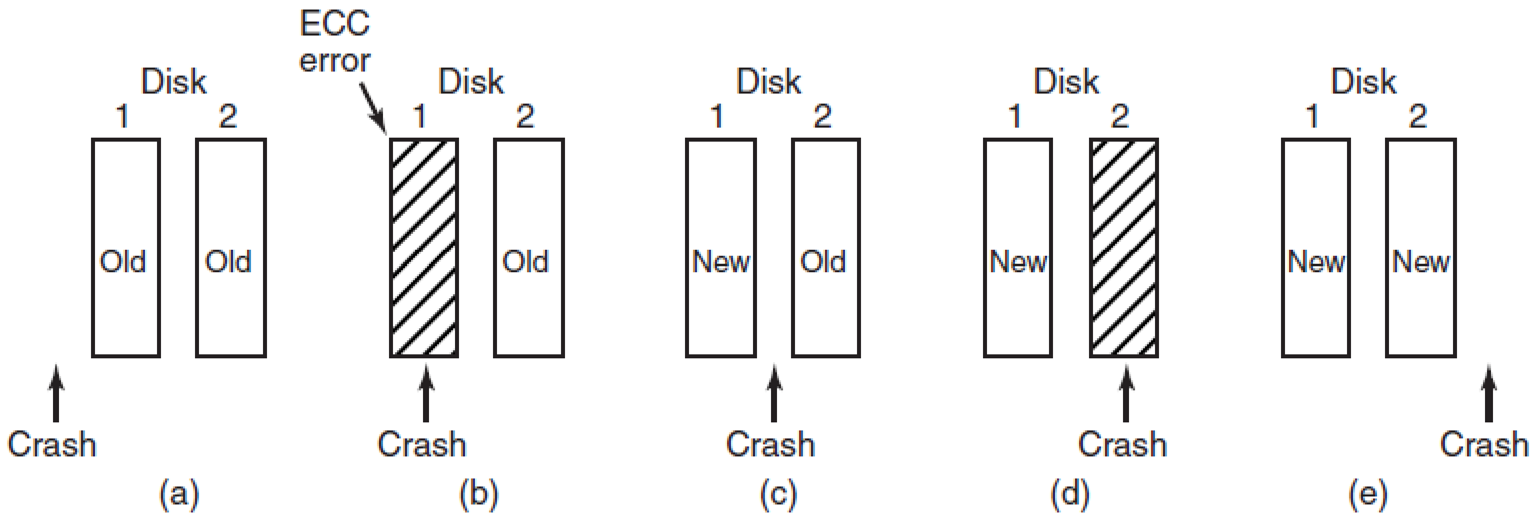


Figure 5-27. Analysis of the influence of crashes on stable writes.

Clock Hardware

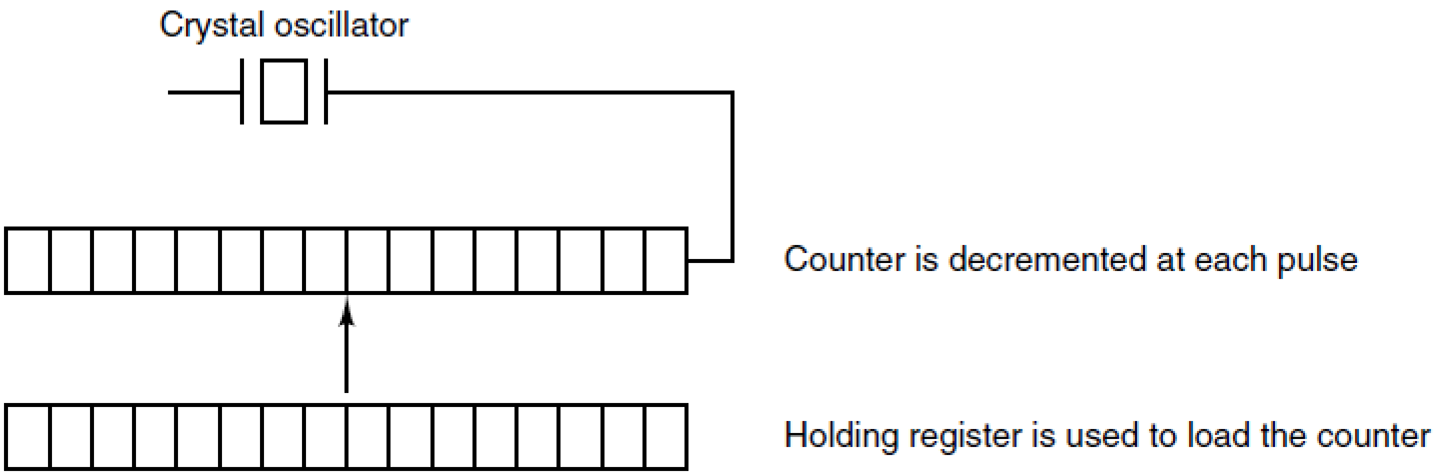


Figure 5-28. A programmable clock.

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)

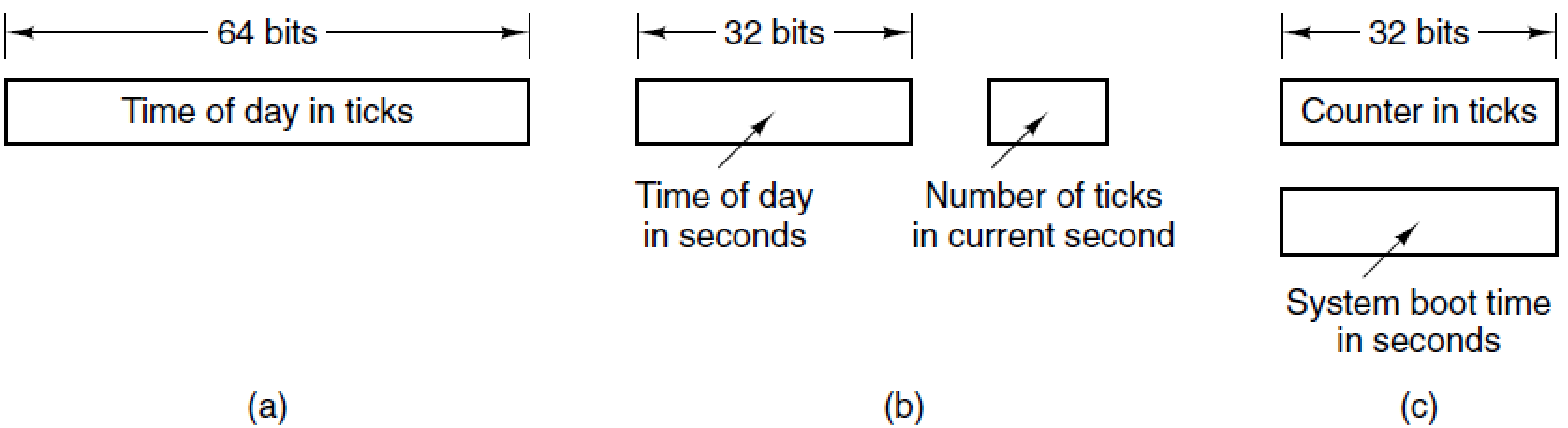


Figure 5-29. Three ways to maintain the time of day.

Clock Software (3)

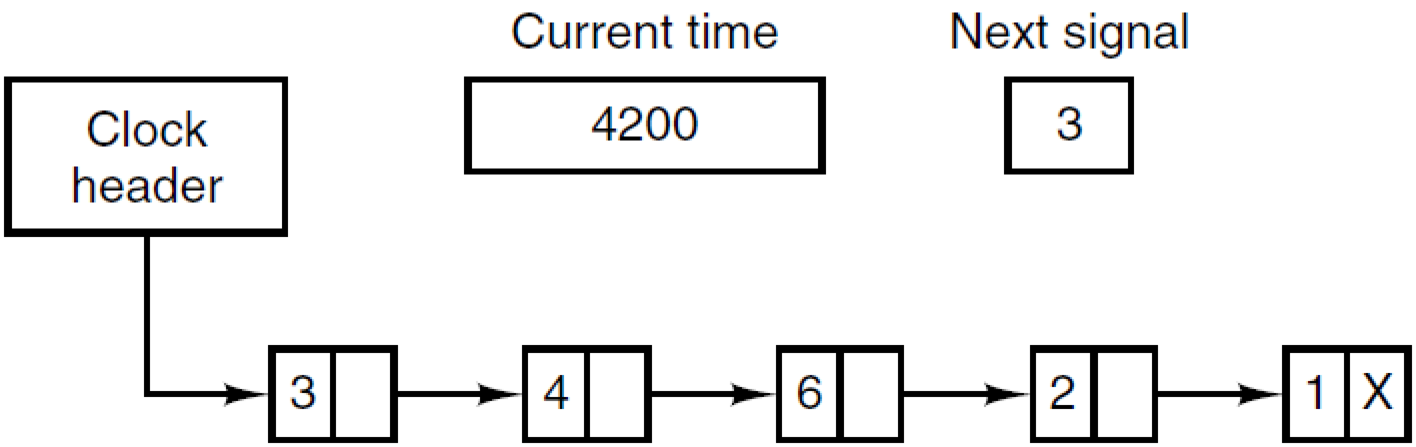


Figure 5-30. Simulating multiple timers with a single clock.

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)

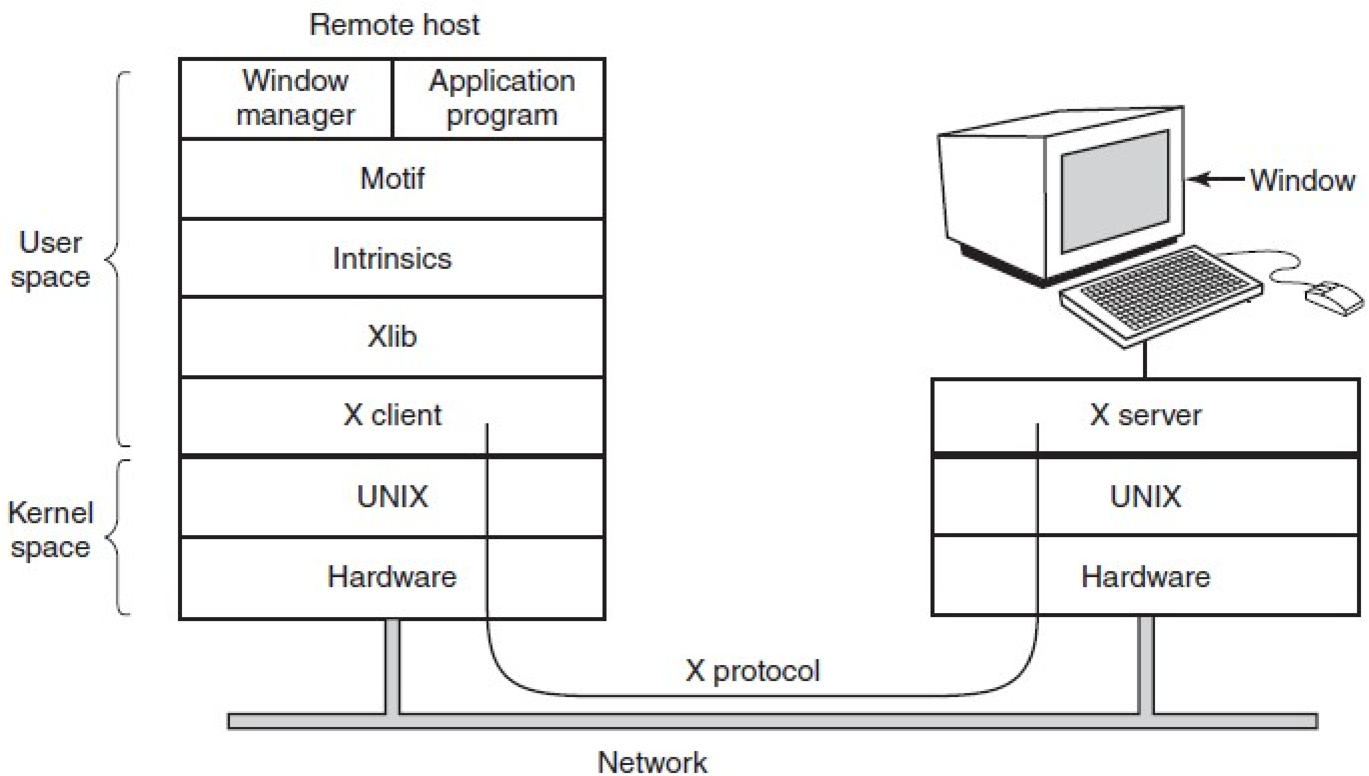


Figure 5-33. Clients and servers in the M.I.T. X Window System.

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)

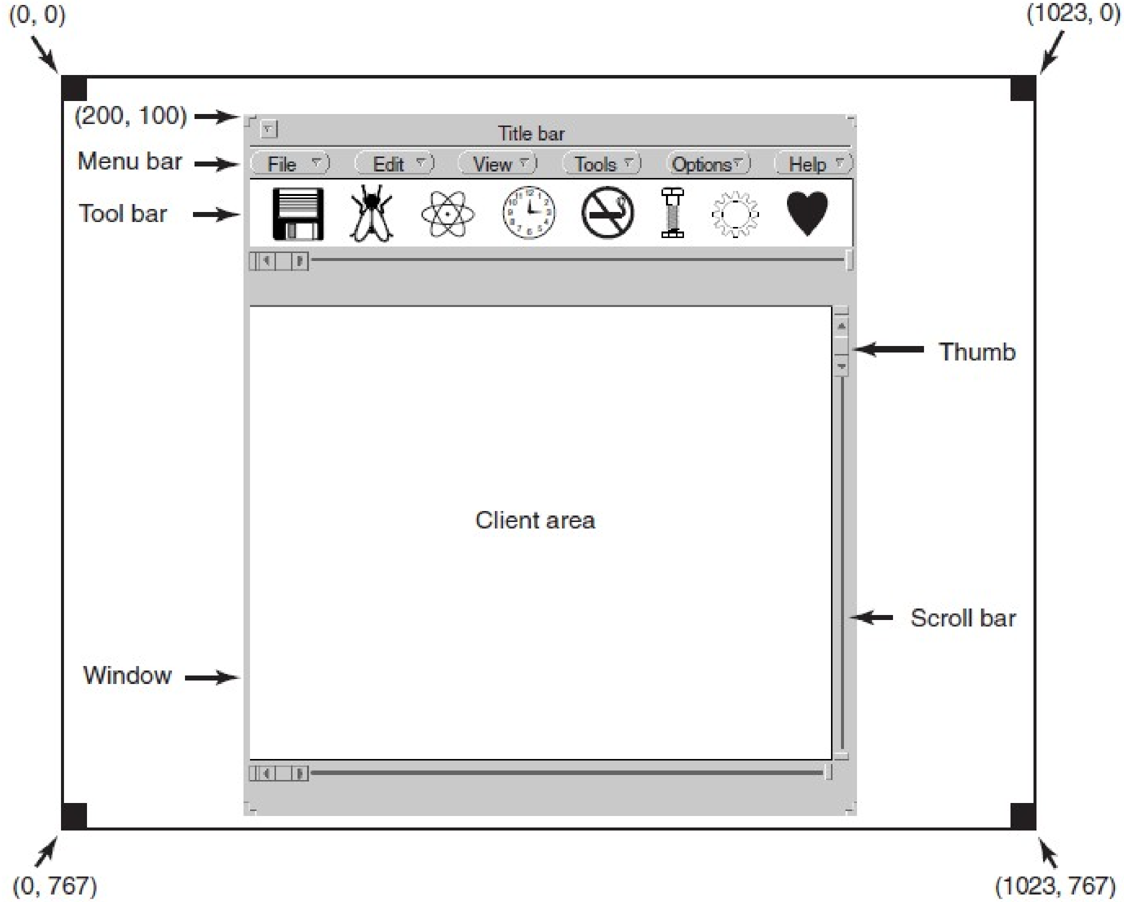


Figure 5-35. A sample window located at (200, 100) on an XGA display.

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, 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 */
        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)

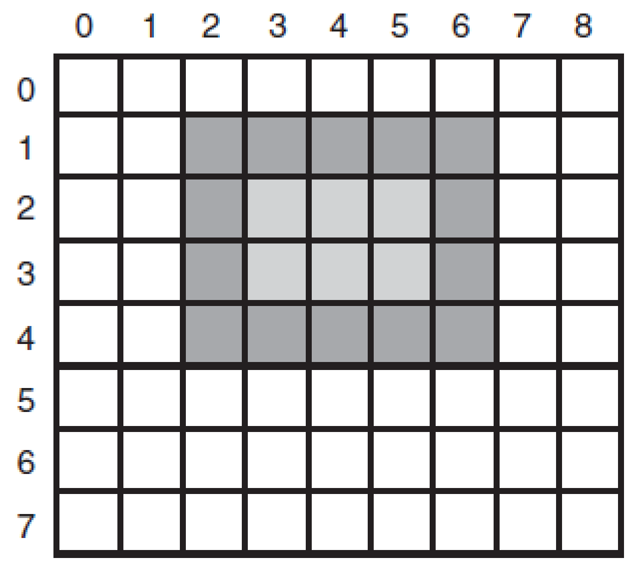


Figure 5-37. An example rectangle drawn using *Rectangle*. Each box represents one pixel.

Bitmaps

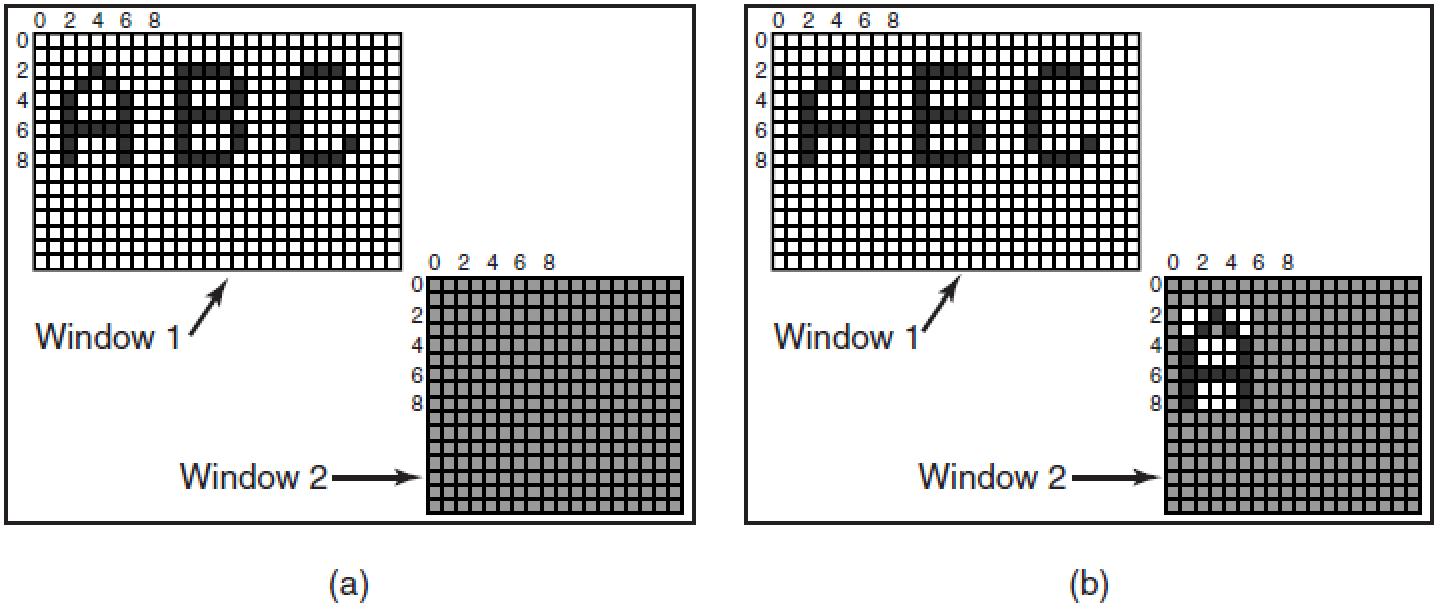


Figure 5-38. Copying bitmaps using BitBlt.
(a) Before. (b) After.

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

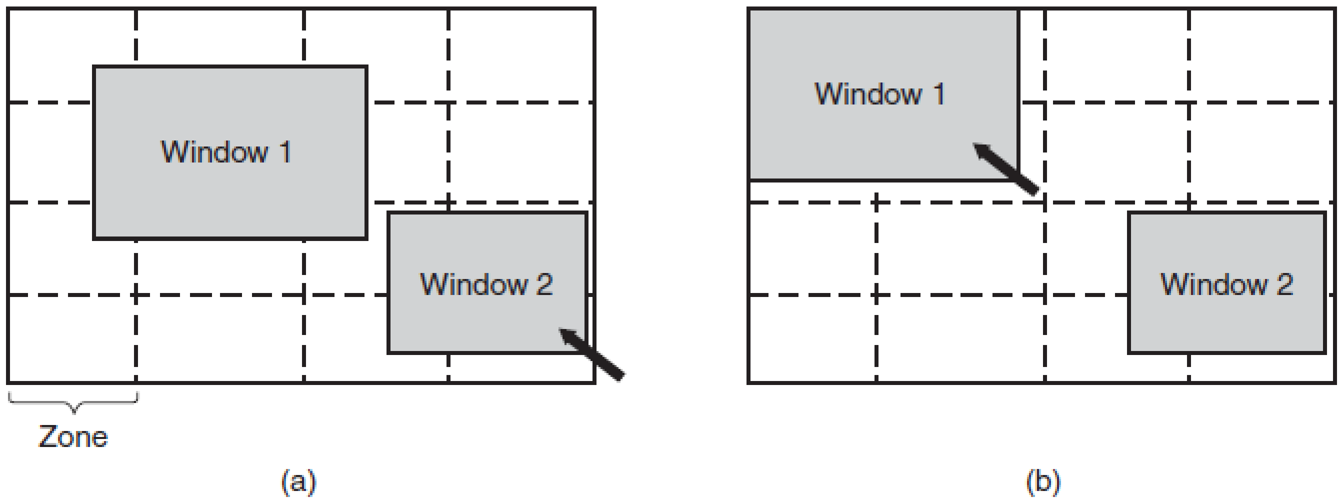


Figure 5-41. The use of zones for backlighting the display.
(a) When window 2 is selected it is not moved.
(b) When window 1 is selected, it moves to reduce the number of zones illuminated.

Operating System Issues
The CPU

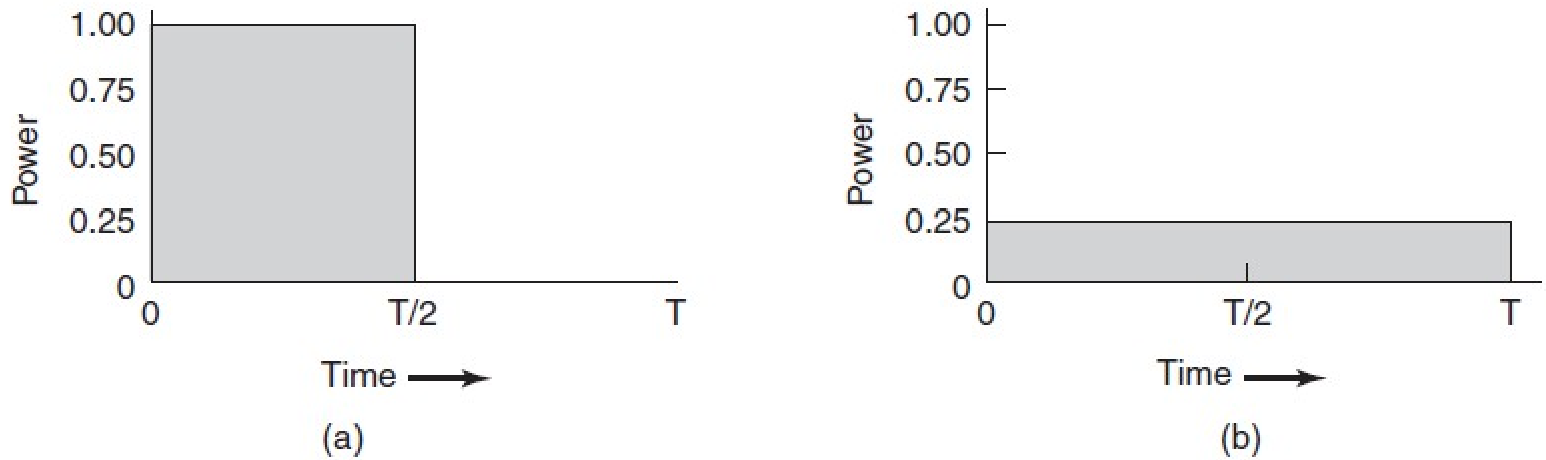


Figure 5-42. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four

End

Chapter 5