CS 444 Operating Systems

Chapter 12 Operating System Design

J. Holly DeBlois

October 9, 2025

"No consensus exists among OS designers about the best way to design an OS..." so Tanenbaum and Bos start by defining goals (pp1041-1042, section 12.1)

- For a general-purpose OS:

1. Define abstractions.
2. Provide primitive operations.
3. Ensure isolation.
4. Manage the hardware.

- Consider designing programming languages, also hard.
- Dennis Ritchie designed C for system programming over 50 years ago.
- PL/I designed by IBM in 1960s combining languages – it's gone.

In 1991, Fernando Corbato, one of the designers of CTSS and MULTICS, emphasized 'minimum of mechanism and maximum of clarity' (see p1046, section 12.2)

- Where do you being?
- Interface design
- What principles do you choose?

> *First, it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties and as we have seen, creating mistakes. My definition of elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.*

- Or St. Exupery: "Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away."

12.2.2 Paradigms: Have a good (execution or user interface) paradigm for how to look at each interface:

- Be consistent throughout, to provide "architectural coherence":

```
main()
{
        int ... ;

        init();
        do_something();
        read(...);
        do_something_else();
        write(...);
        keep_going();
        exit(0);
}
```

(a)

```
main()
{
        mess_t msg;

        init();
        while (get_message(&msg)) {
                switch (msg.type) {
                        case 1: ... ;
                        case 2: ... ;
                        case 3: ... ;
                }
        }
}
```

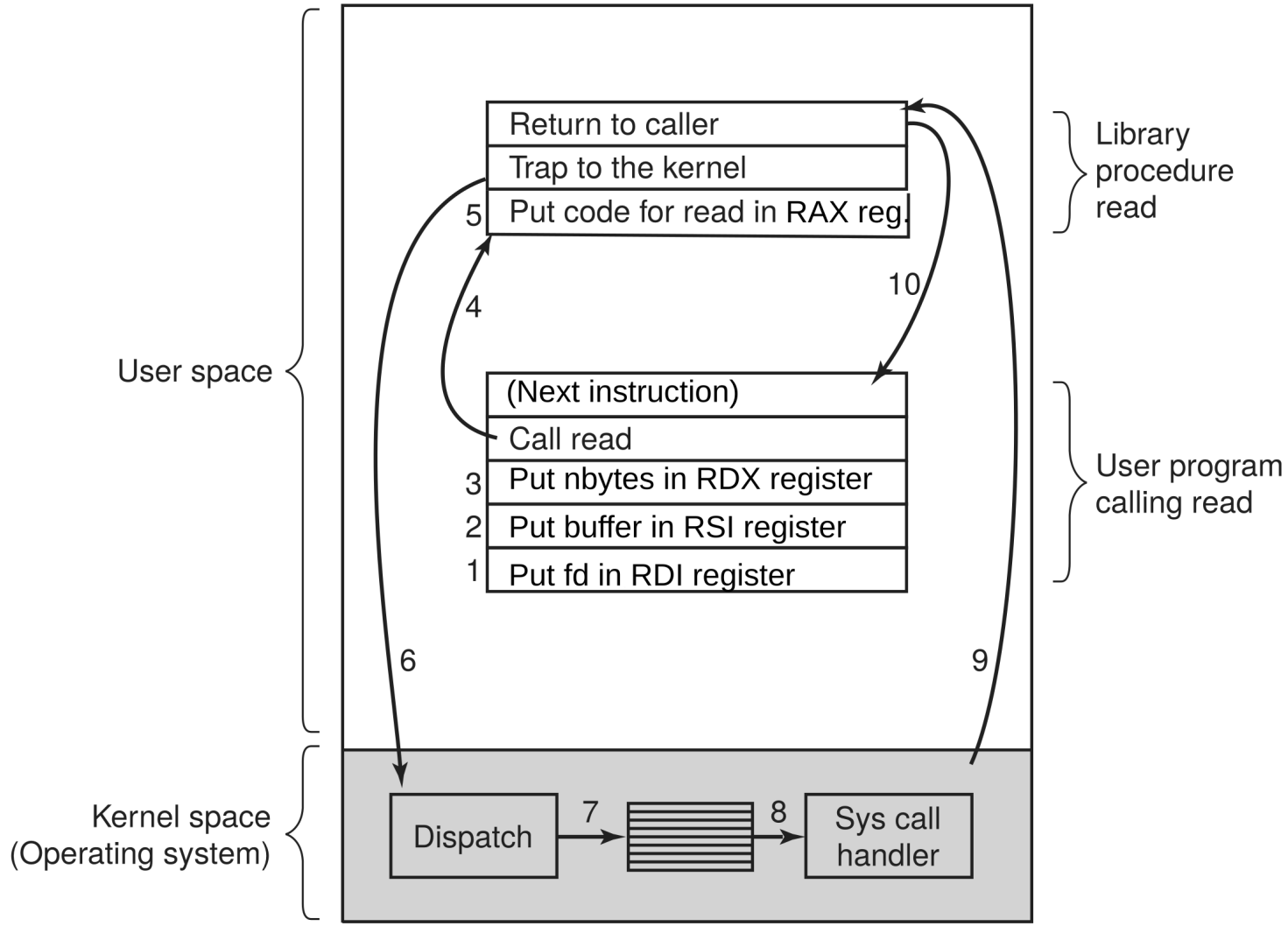
(b)

Figure 12-1. (a) Algorithmic code. (b) Event-driven code.

- See WIMP, p430, user interface paradigm, and above

12.2.3 The System-Call Interface - use as few system calls in your design as possible (p1051-1053)

- Reread Ch1
- Know 10 or fewer steps
- Know params in read



UNIX has a system call for overlaying a process' virtual address space: "exec (p1052), which illustrates Corbato's "minimal of mechanism" strategy

```
exec(name, argp, envp);
```

which loads the executable file *name* and gives it arguments pointed to by *argp* and environment variables pointed to by *envp*. Sometimes it is convenient to list the arguments explicitly, so the library contains procedures that are called as follows:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

- One call loads the executable file name and gives it arguments
- The call can have additional versions, that also call the basic exec
- Creating a process uses two calls: fork followed by exec

12.3.1 System Structure possibilities: layered (see below) or end-to-end (not layered) (p1054)

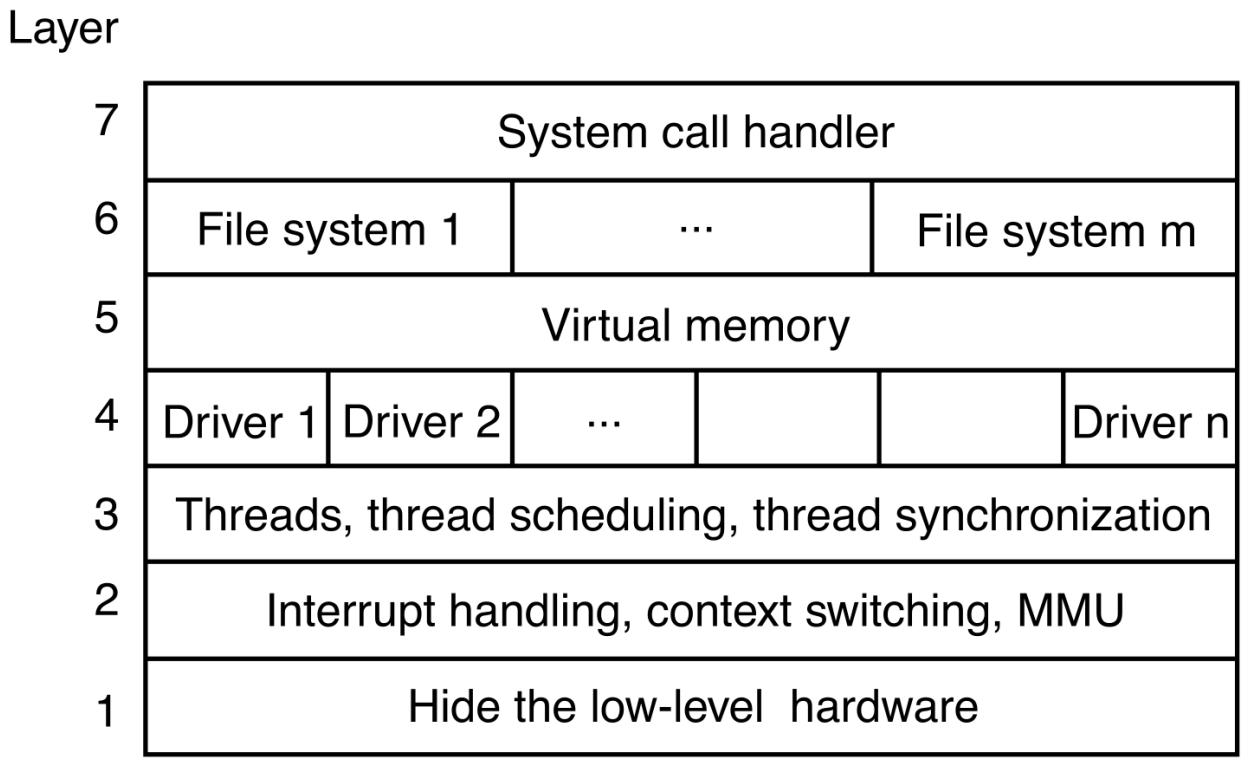


Figure 12-2. One possible design for a modern layered operating system.

Compare Windows kernel layering, section 11.3, p895)

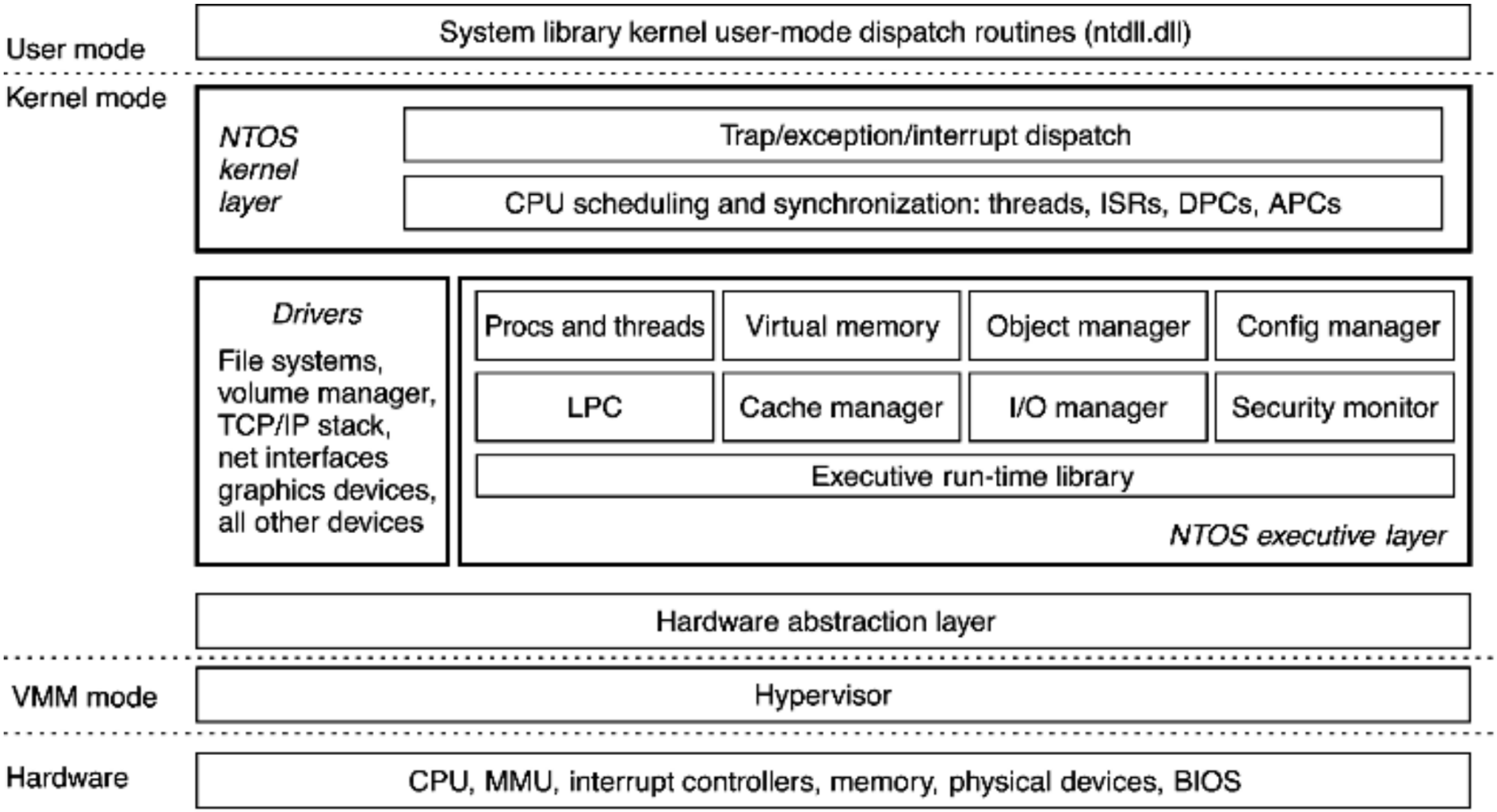


Figure 11-11. Windows kernel-mode organization.

Microkernel-Based Client-Server System is a compromise between fully layered OS and idea that OS should let user level processes do all but "securely allocate resources among competing users" (p1055)

- Note how the OS features are all on the same level as the user processes:

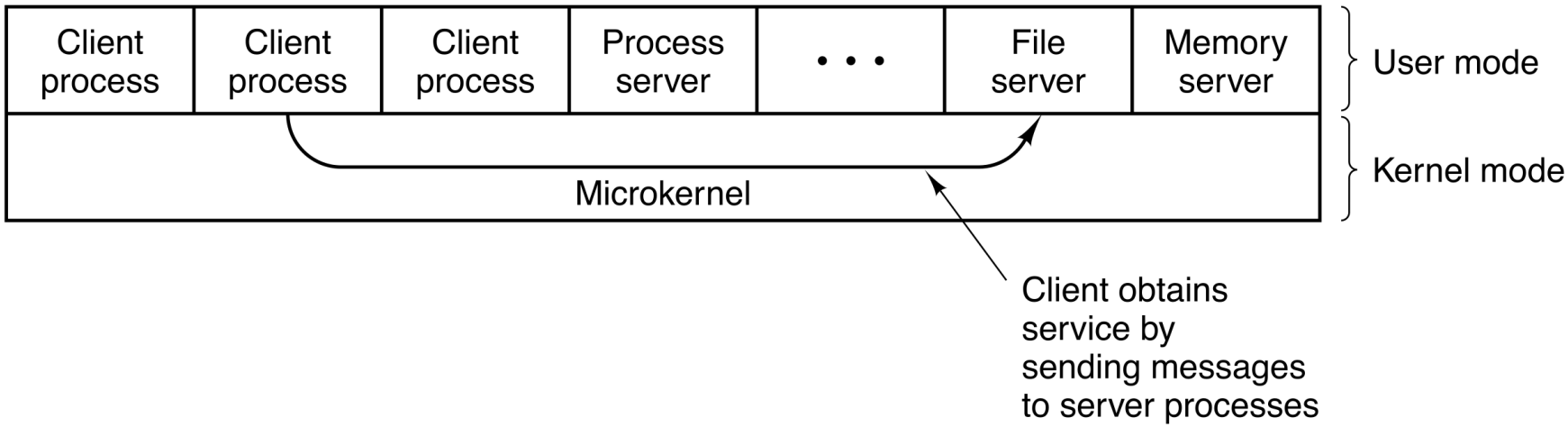


Figure 12-3. Client-server computing based on a microkernel.

12.3.4 Naming (p1059-1060)

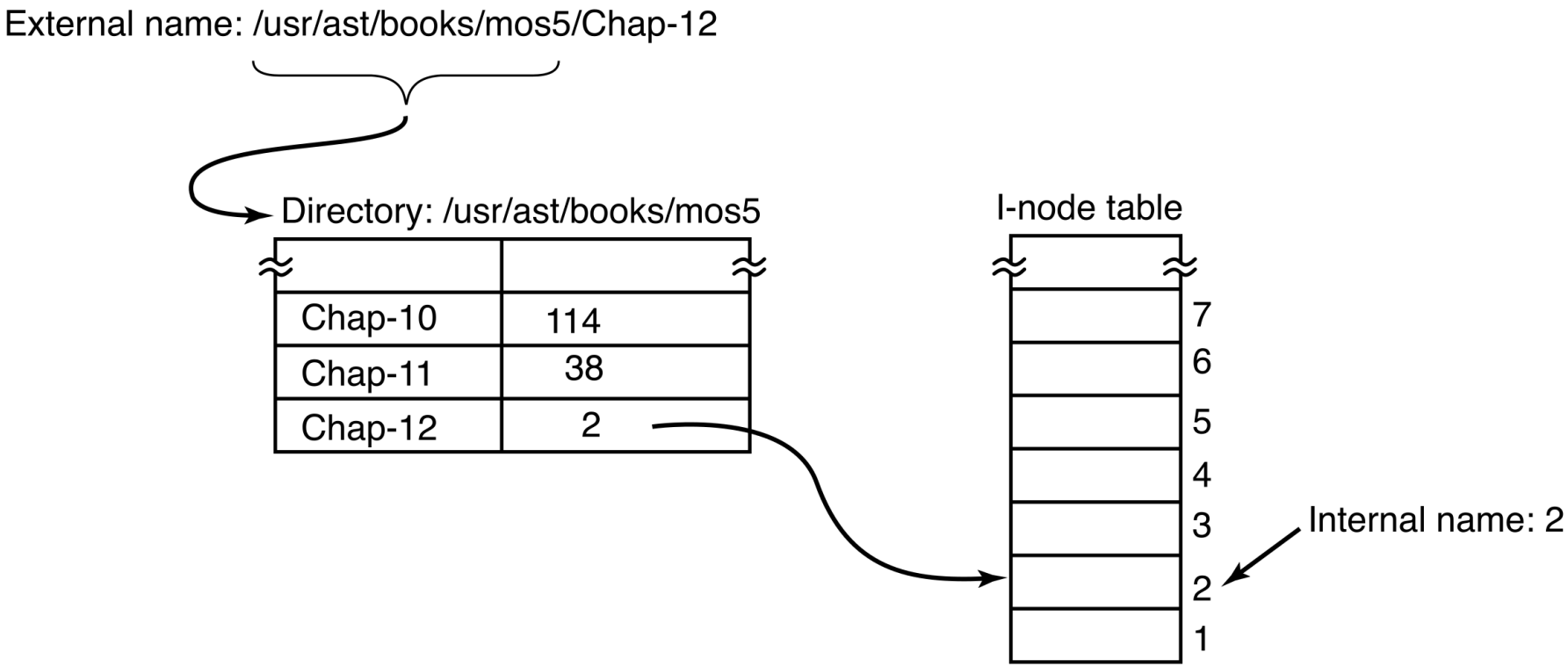


Figure 12-4. Directories are used to map external names onto internal names.

- Naming often includes long, friendly names for human use and brief internal names
- In your C code, you must consider naming carefully, to combine clarity and brevity!

12.3.6 Static vs. Dynamic Structures (p1062) - example process table as static table with fast access or process table as linked list with expandable size

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
        if (p->proc_pid == pid) {
                found = 1;
                break;
        }
}
```

Figure 12-5. Code for searching the process table for a given PID.

- If process table is a static data structure of fixed size, as shown below, it has fast access
- but could run out of space
- If process table were implemented as a linked-list, it could easily be expanded, but would be slower to access

12.4.3 Performance Space-Time Trade-offs (p1073-1075):

```
#define BYTE_SIZE 8                          /* A byte contains 8 bits */

int bit_count(int byte)
{                                            /* Count the bits in a byte. */
        int i, count = 0;

        for (i = 0; i < BYTE_SIZE; i++)      /* loop over the bits in a byte */
                if ((byte >> i) & 1) count++; /* if this bit is a 1, add to count */
        return(count);                       /* return sum */
}
```

(a)

```
/*Macro to add up the bits in a byte and return the sum. */
#define bit_count(b) ((b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                      ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1))
```

(b)

```
/*Macro to look up the bit count in a table. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
```

(c)

Figure 12-7. (a) A procedure for counting bits in a byte. (b) A macro to count the bits. (c) A macro that counts bits by table lookup.

Similarly, image compression technique GIF uses table lookup to encode 24-bit RGB pixels:

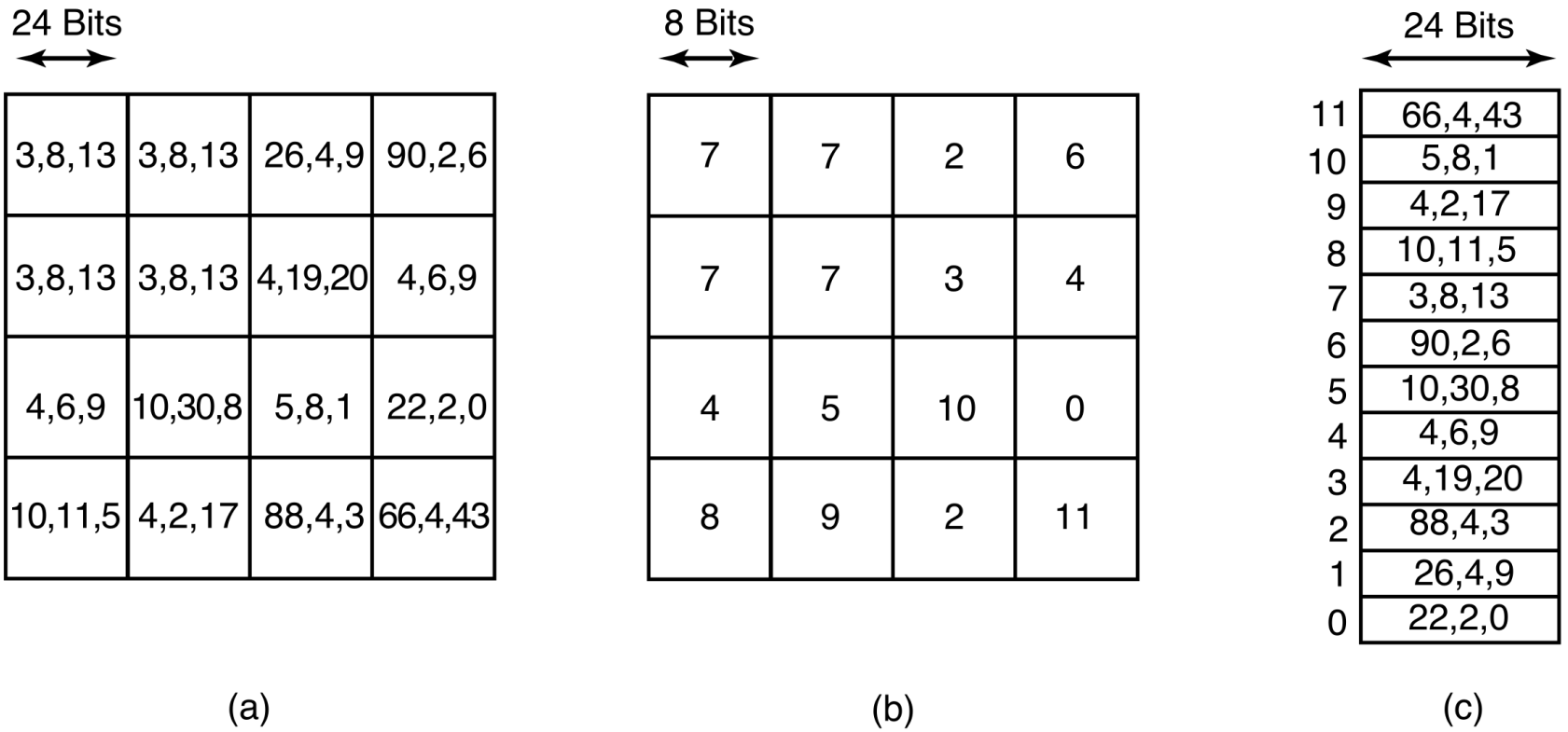


Figure 12-8. (a) Part of an uncompressed image with 24 bits per pixel. (b) The same part compressed with GIF, with 8 bits per pixel. (c) The color palette.

- each 8-bit value is an index
- color palette is store with image file

12.4.4 Caching (p1075-1076): certainly can improve performance but requires careful design

- Review UNIX example in Fig 4-36, p329, which has these disk accesses to look up a path:

    1. Read the i-node for the root directory (i-node 1).
    2. Read the root directory (block 1).
    3. Read the i-node for */usr* (i-node 6).
    4. Read the */usr* directory (block 132).
    5. Read the i-node for */usr/ast* (i-node 26).
    6. Read the */usr/ast* directory (block 406).

- some systems optimize path-name parsing by caching (path,i-node) combinations
- then for a lookup of a long path, the cache may already have part of it

Example_continued, consider accessing /usr/ast/grants/erc:

Path	I-node number
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

Figure 12-9. Part of the i-node cache for Fig. 4-36.

- cache can return /usr/ast inode 26 saving 4 disk accesses
- but of course if files are deleted, inodes get reused, so cache entries must be removed when no longer valid
- c

12.5 Project Management (pp1078-1080) discusses the "chief programmer team paradigm (Baker, 1972) which organizes like a surgery team around a surgeon

Title	Duties
Chief programmer	Performs the architectural design and writes the code
Copilot	Helps the chief programmer and serves as a sounding board
Administrator	Manages the people, budget, space, equipment, reporting, etc.
Editor	Edits the documentation, which must be written by the chief programmer
Secretaries	The administrator and editor each need a secretary
Program clerk	Maintains the code and documentation archives
Toolsmith	Provides any tools the chief programmer needs
Tester	Tests the chief programmer's code
Language lawyer	Part timer who can advise the chief programmer on the language

Figure 12-10. Mills' proposal for populating a 10-person chief programmer team.

- we know top programmers are 10x as productive as others, but we don't always have top people
- Harlan Mills gave the above list as a candidate way to build software

Since "bad news does not travel well up the organization tree", Brooks's solution was to go to more agile approach (p1082):

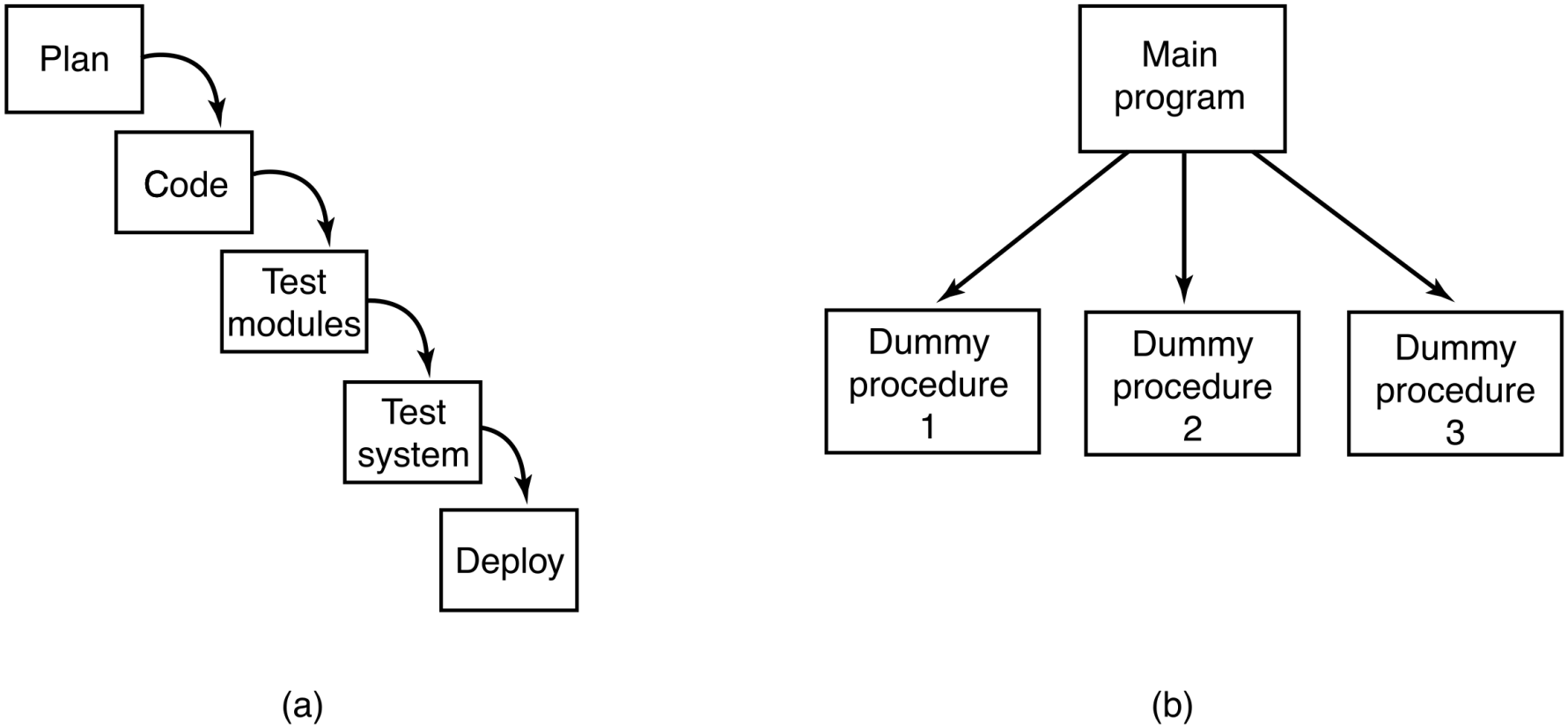


Figure 12-11. (a) Traditional software design progresses in stages. (b) Alternative design produces a working system (that does nothing) starting on day 1.

As a user interface followup, consider the next two slides from section 5.6.2 Output Software, specifically X window system (pp404-408) and Graphical User Interface (GUI) example (pp408-413):

- The X Window System, illustrated in Fig 5-33, p405, and included in Ch5 slides shows networked access developed at M.I.T. in the 1980s to connect remoste user terminals with a central server
- It is a windowing system, not a complete Graphical User Interface.
- Gnome and KDE are examples of commercial Common Desktop Environments and they run on top of X
- A rough example of the code is in the next slide

```
#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 */
                    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.

Second example from section 5.6.2 Output Software, specifically Graphical User Interface (GUI) example (pp408-413):

- The GUI was invented by Douglas Engelbart and his research group at Stanford Research Institute.
- It is a "massive topic" (p409) starting with basic item, the window.
- The program shown is explained on pages 410-414.
- A rough example of the code is in the next slide, called a skeleton of what the Windows code for the main program might be.
- We will discuss it further when we study Ch11 Windows. Note the header file, the main declaration and the main loop.
- Tanenbaum mentions that the design "of the whole mechanism" of the main loop likely could have been made simpler, but it was done this way and "we are stuck with it" (p412).
- Windows OS roughly 50 million lines of code, not open source. Linux OS roughly 15 million lines of code, is open source.

```
#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 */
        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.