Agenda

Overview: I/O Devices

Software Layers

- Device Controllers
- Interrupts Revisited
- Drivers
- Device-independent interfaces
- User I/O

Principles of I/O implementations

I/O Devices

A primary job of the operating system is controlling all the computer's I/O (Input/Output) devices

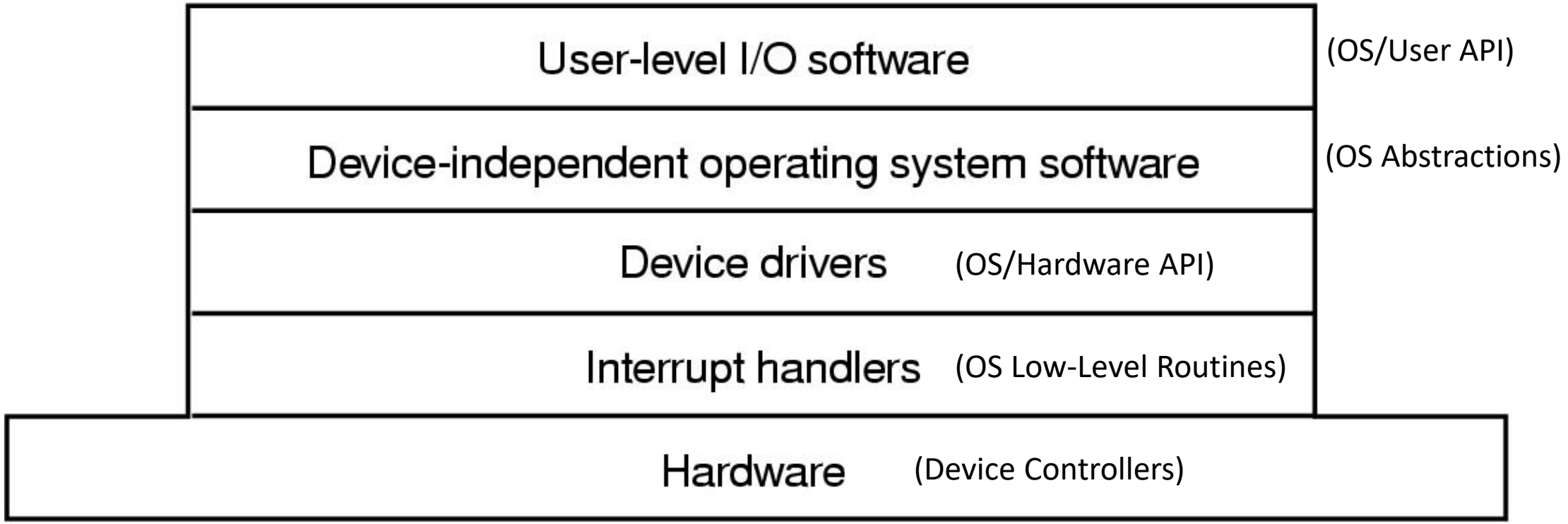
Block devices – stores information in fixed-sized blocks, each with its own address

- E.g. Disks, USBs

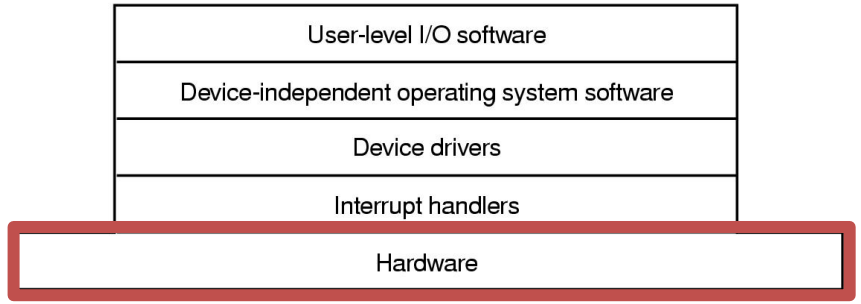
Character devices – operates on a stream of characters, is not addressable and does not have seek operations

- E.g. Printers, Terminals, Network interfaces, Mice (pointing devices), Keyboard

Overview: I/O Software Layers



Device Controllers



I/O devices have components:

- mechanical component
- electronic component

The electronic component is the device controller

Controller's task: Execute requests on the hardware

- Example: Convert a serial bit stream to block of bytes
- Example: Display a buffer of colors to the screen

Communicating with devices

Issue: How should the software communicate with devices?

Devices typically come with a

- control registers
- data buffer

Examples

- Printer: A command indicates that the device should read data from the buffer
- Screen: A command indicates that we should refresh the screen using the data in video RAM

Device communication: registers

How should the OS communicate with control registers?

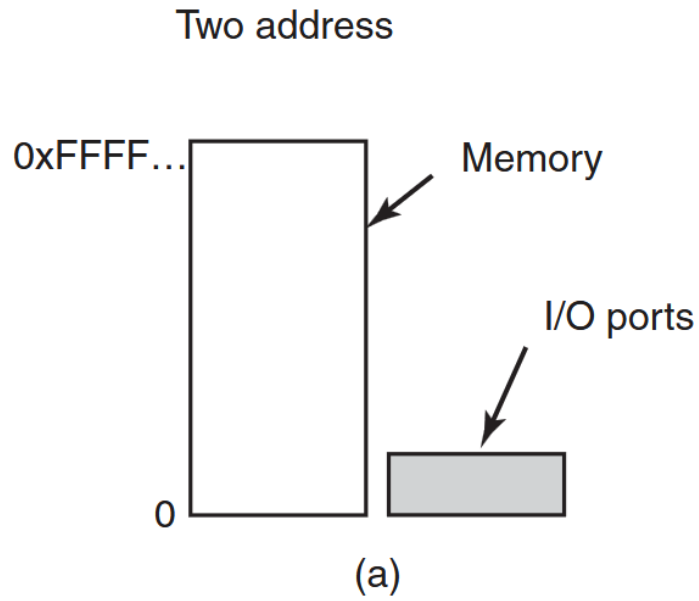
Two Approaches:

- Separate I/O space and memory spaces
- Memory-mapped I/O

Approach 1: Separate I/O and memory

An **I/O port** is a control register for a device, labeled with an integer. The set of I/O ports is called the **I/O space**

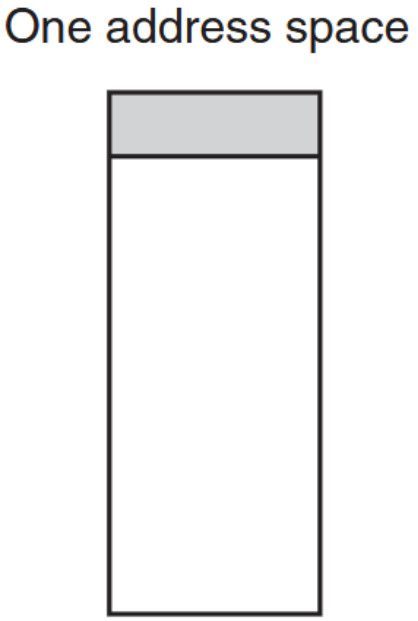
How it works: the OS can either read/write to I/O space or memory, using special instructions for each case



Example: The OS uses special commands to interact with devices

IN REG, PORT
OUT PORT, REG

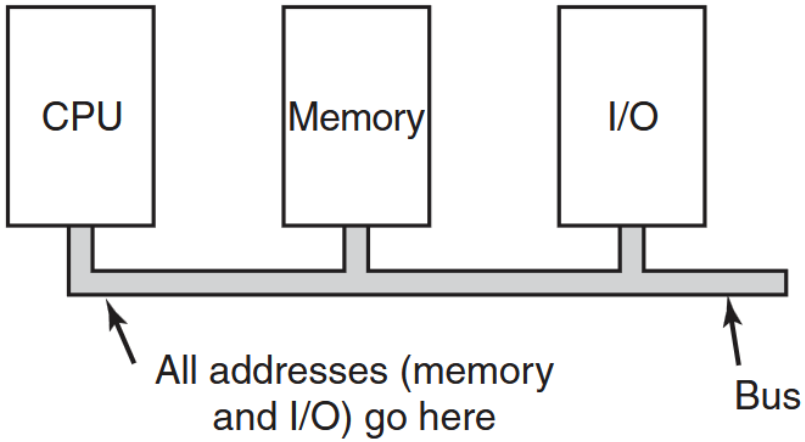
Approach 2: Memory-mapped I/O



Idea: Map all control registers to memory space

E.g. Memory address 0x10 corresponds to the printer

How it works: The OS reads/writes to devices and memory using the same instructions. If the address corresponds to a device, the device handles it. If it corresponds to memory, memory is accessed in the usual way.



Approach 2: Memory-mapped I/O

Advantages:

- Can be written without assembly (easier)
- No special protection mechanisms are needed
- No special instructions need to be supported

Disadvantages:

- Makes caching more complicated (device values should never be cached. Why?)
- Bus optimizations are more complicated

Device communication: data

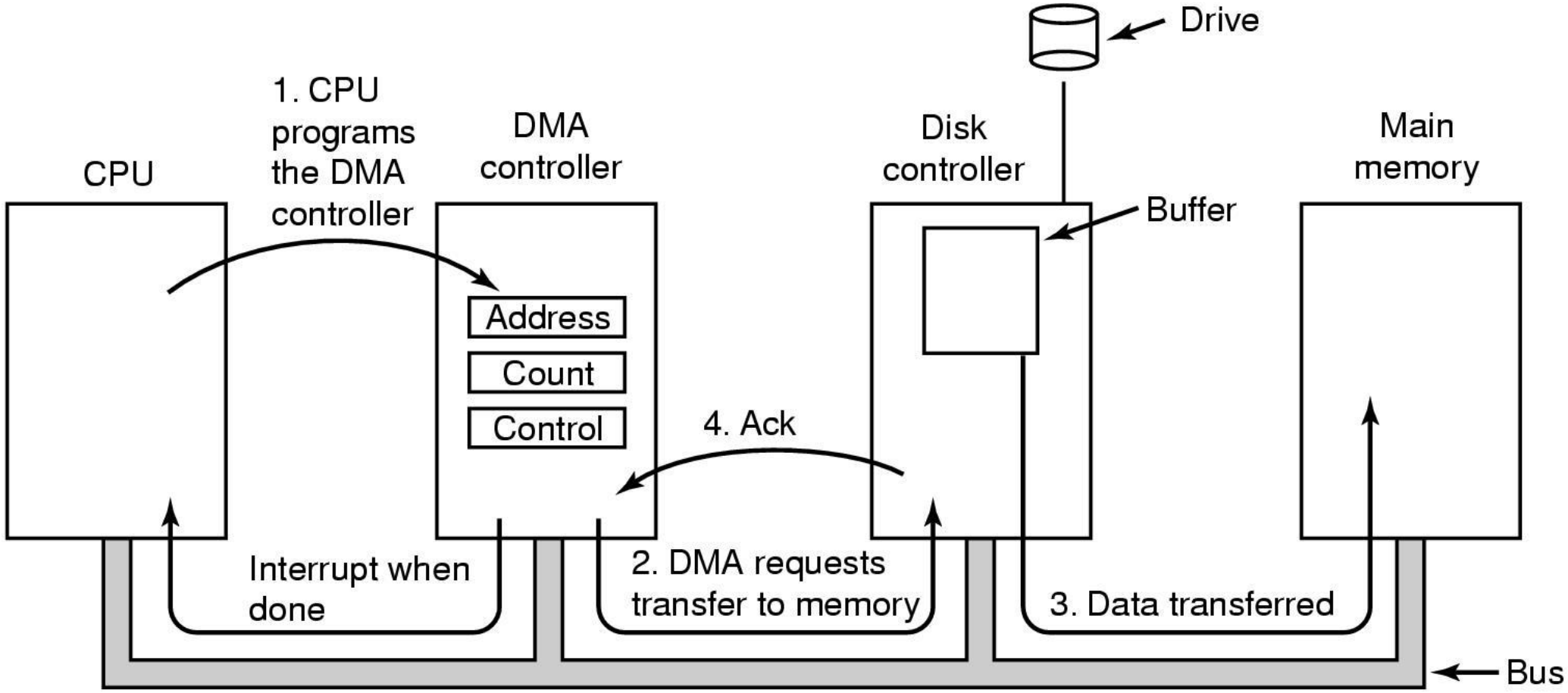
How should the OS read and write data to devices?

Two Approaches:

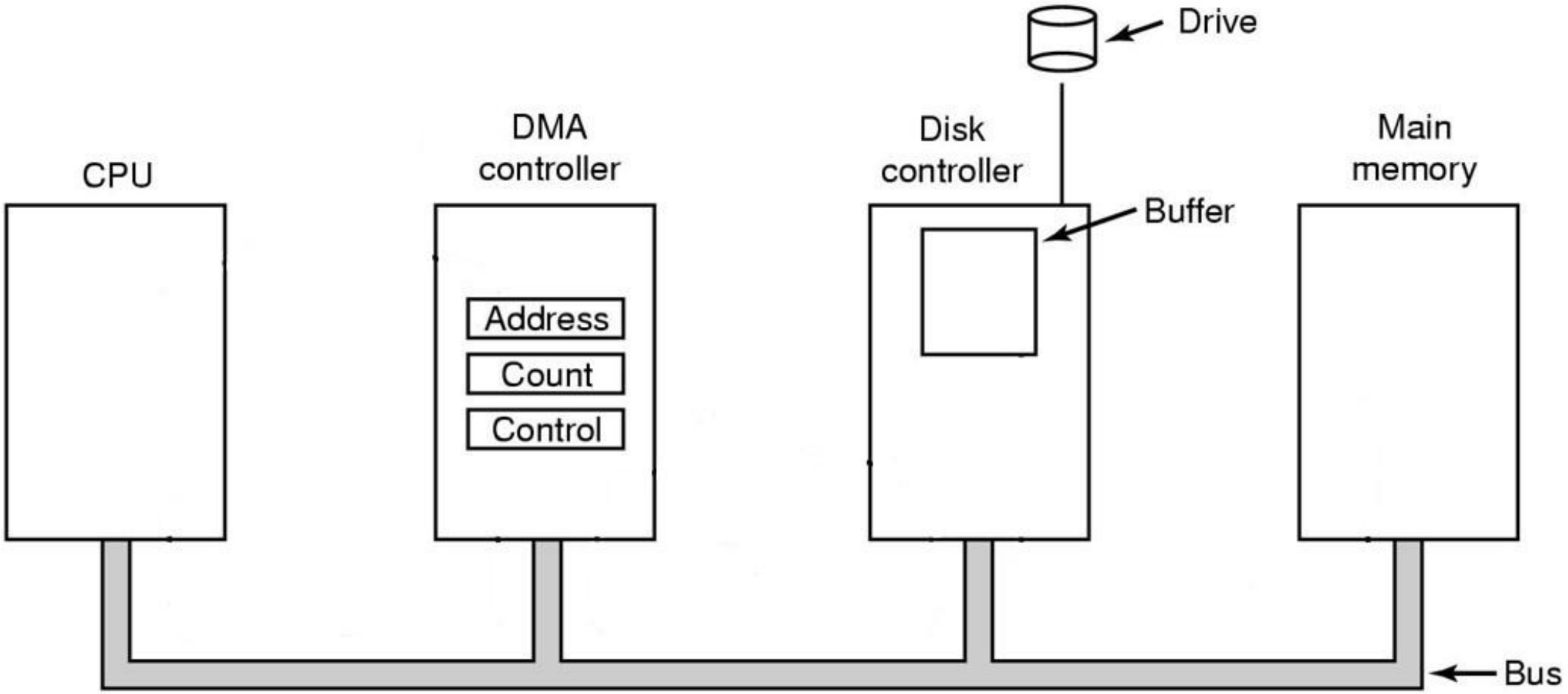
- Direct interaction between OS and the device
  - Too slow, needs to be one byte at a time
- With **Direct Memory Access (DMA),** the OS offloads reading and writing data to a controller so it can work on other tasks in the meantime

The DMA controller will tell you when your bytes are ready.

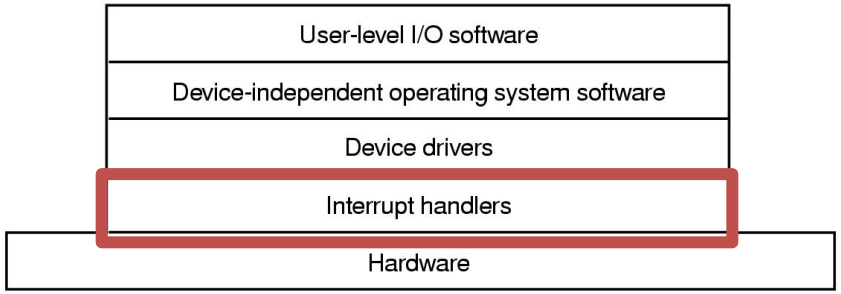
Direct Memory Access (DMA)



Exercise: Direct Memory Access (DMA)



Interrupt handlers



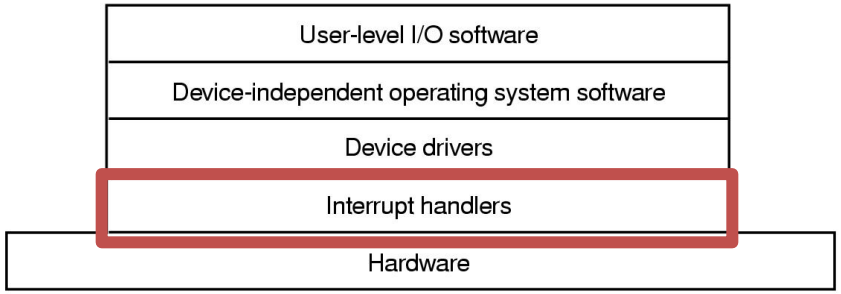
Recall: Devices can send interrupts to indicate when a request is complete

Without interrupts, the OS must sit and wait for requests to complete

How it works:

- An interrupt controller manages interrupts from devices
- When an interrupt occurs, the controller will check if a current interrupt is in progress
  - If yes, it ignores the interrupt (the device will repeat it until it is handled)
  - If no, the controller will tell the OS which device needs attention
    - The controller sends a number that corresponds to the device
    - The number is used to lookup a handler in a table, called the **interrupt vector**
    - Currently executing code is stopped and saved.
    - We run the handler. When complete, we acknowledge the interrupt. Why?
    - Previous code is resumed when the process is restarted later.

Interrupt handlers



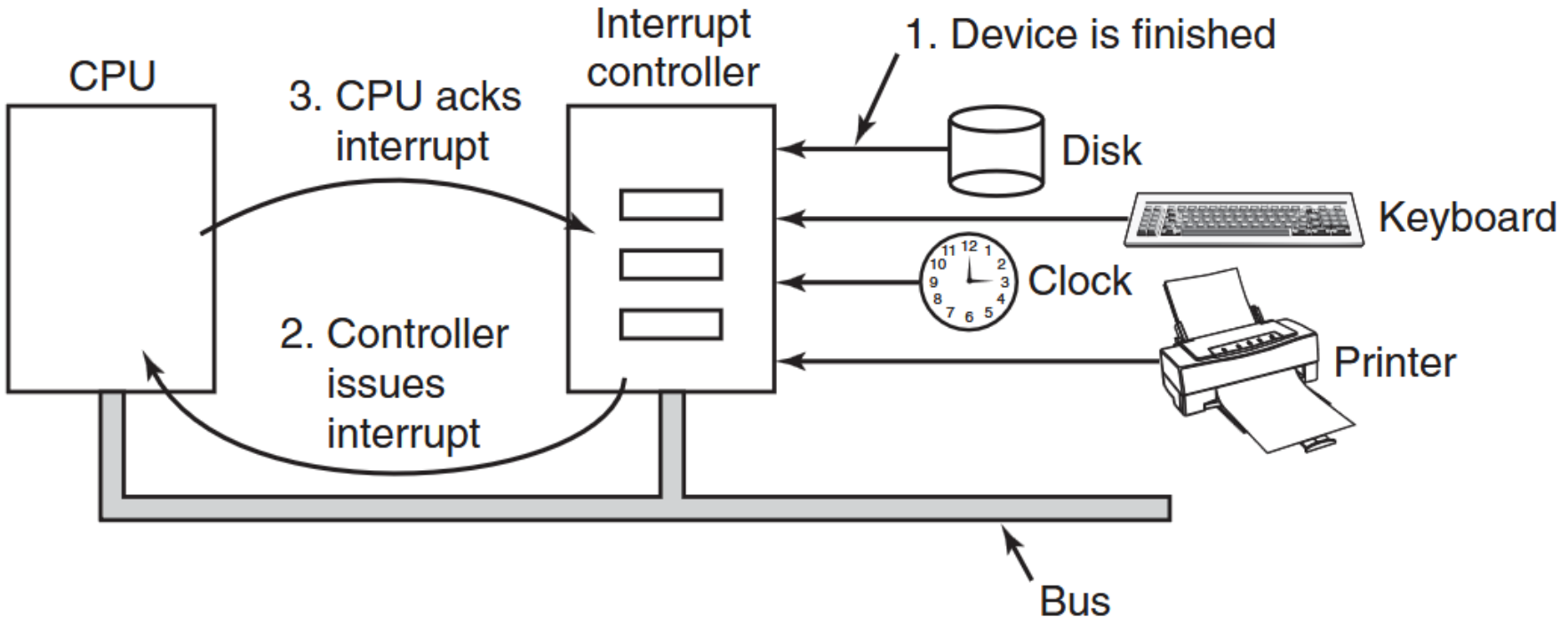


Figure 5-5. How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

Three approaches to I/O handling

Programmed I/O: The CPU does all the work

Interrupt-driven I/O: The CPU does other work while a controller handles the request. An interrupt indicates when the work is done.

I/O with DMA: The CPU does other work while a DMA controller handles the request. An interrupt indicates when the work is done.

Programmed I/O: Busy-waiting, polling

- Suppose you want to send *n* bytes to printer
- Printer has two memory-mapped registers
  - **`status`** – OS reads for printer status
  - **`data`** – OS writes to send data
  - We busy-wait on **`status`**

```
for (i=0; i<N; i++) {
  /* wait for device to be ready */
  while (*status != READY) ;
  *data = buffer[i];
}
```

Interrupt-Driven I/O

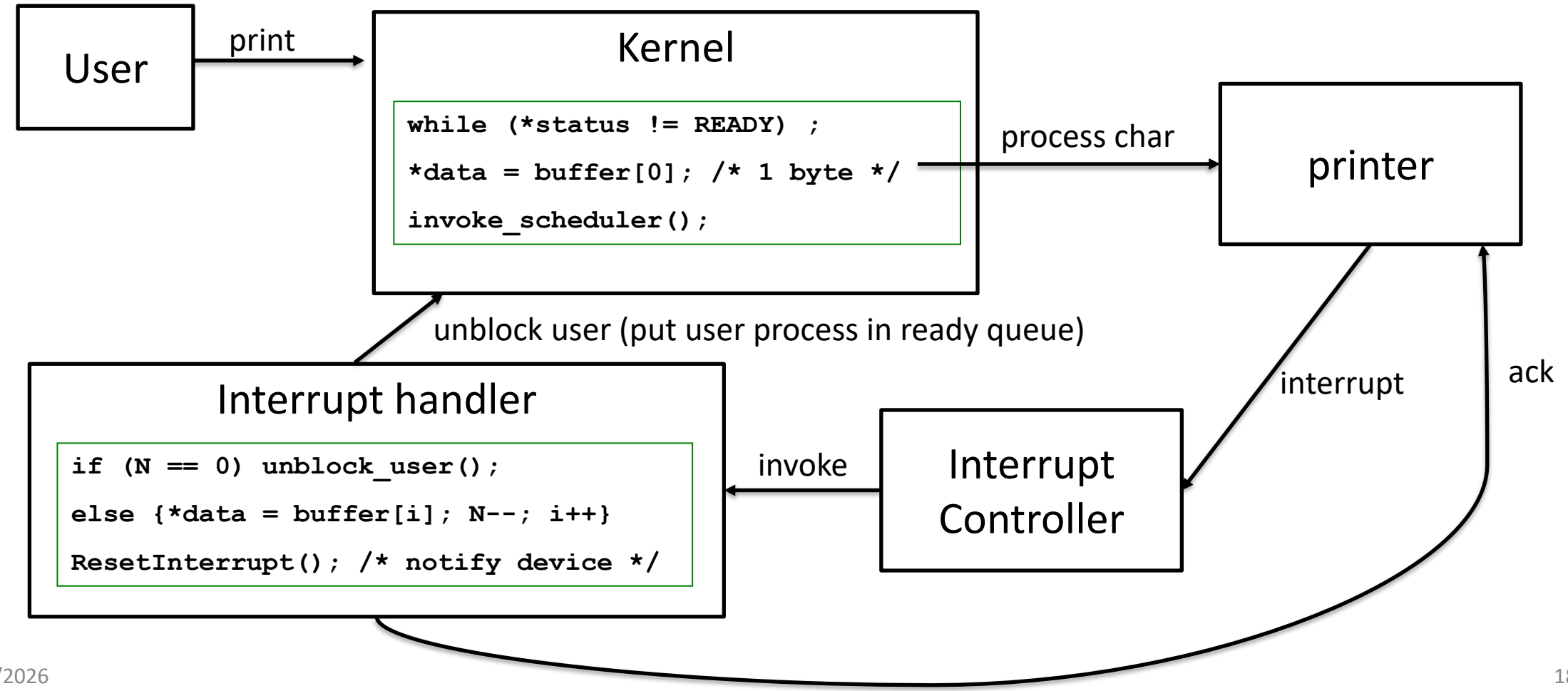
- Need both driver and interrupt handler
- Driver copies a single byte into buffer and calls scheduler:

```
while (*status != READY) ;
*data = buffer[0]; /* 1 byte */
invoke_scheduler();
```

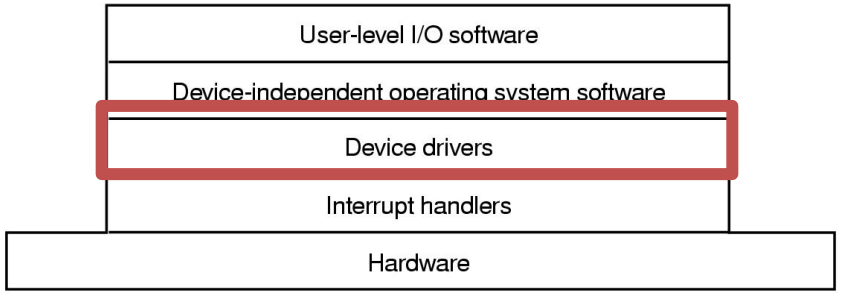
- Printer generates interrupt after printing one char. Interrupt handler does the rest:

```
if (N == 0) unblock_user();
else {*data = buffer[i]; N--; i++}
ResetInterrupt(); /* notify device */
```

Interrupt-driven I/O



Device drivers



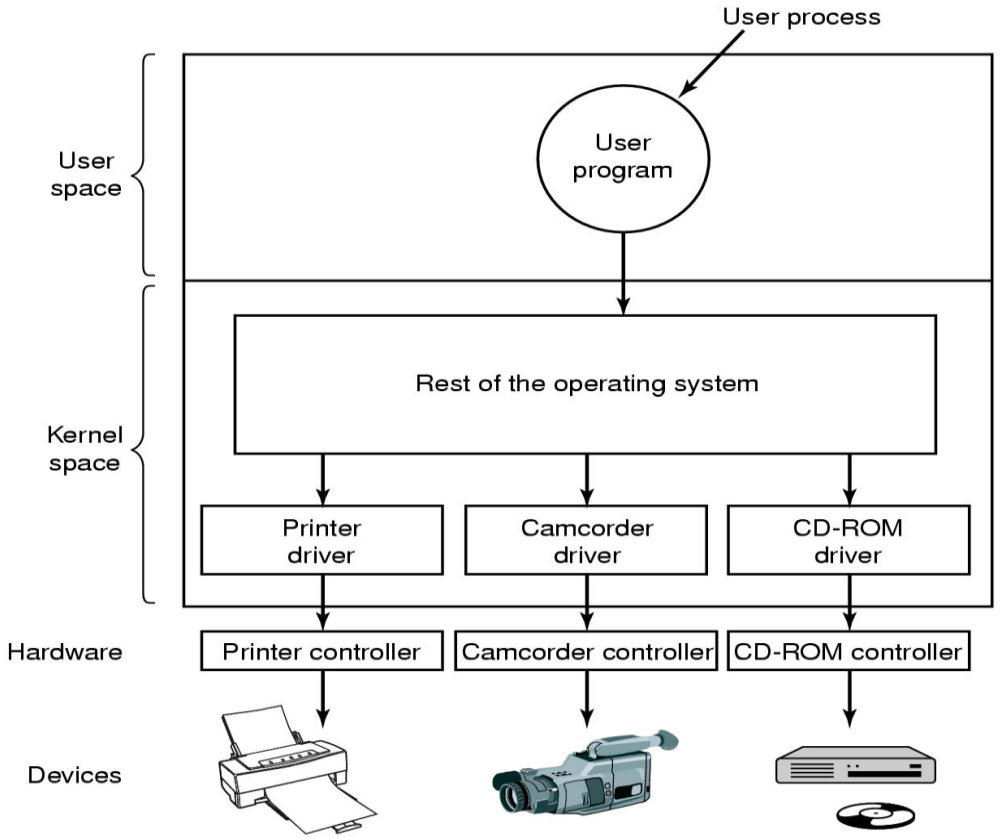
A **device driver** is software that implements device-specific code for an operating system

Operating systems define standard interfaces for block-based devices and character-based devices.

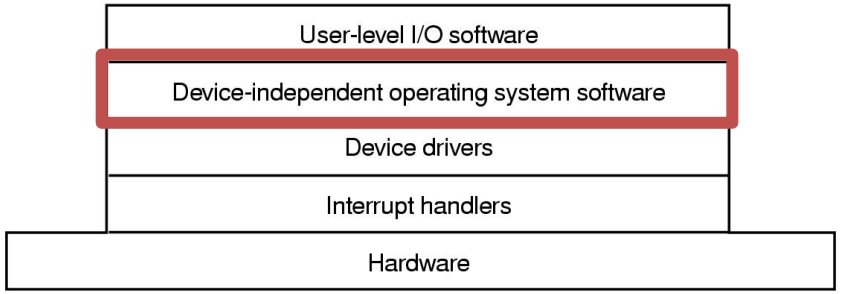
Driver-code and kernel-code are separated in modern OS's

- Drivers can be updated/added without modifying the kernel's code

Typically implements requests and interrupt handling between the OS and the device



Device-independent software



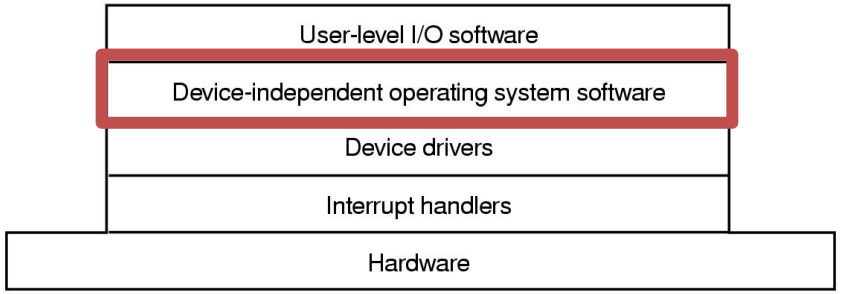
Goal: Provide a uniform interface for working with devices

Example: We want the same interface for all disks, regardless of whether it is a CD, USB, or SATA drive

Major Issues:

- Uniform interfacing
- Buffering
- Error reporting

Uniform interfacing



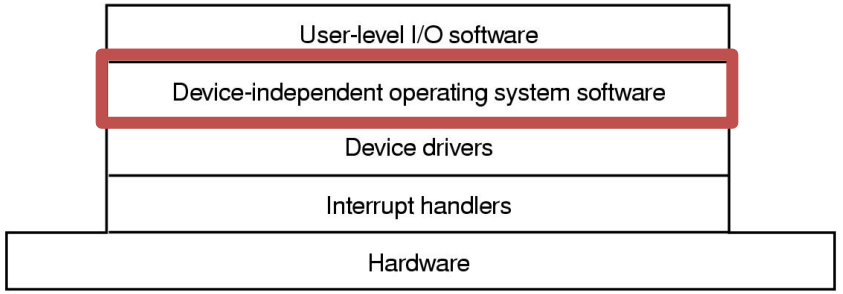
Common technique 1: API Specification

- Device drivers implement an interface using function pointers
- The OS registers these pointers as system calls

```
typedef struct {
  fd_t (*f_open)(vnode_t *vn, const char *filename, int flags);
  size_t (*f_read)(vnode_t *vn, void *data, size_t size, int num, fd_t fd);
  ... /* and the rest of the file system library functions */
} fs_driver_t;
```

```
typedef struct vnode {
  int vnode number;
  char name[255];
  struct vnode *parent;
  int permissions;
  int type;
  fs_driver_t *driver;
  ... /* file system dependent info */
} vnode_t;
```

Uniform interfacing

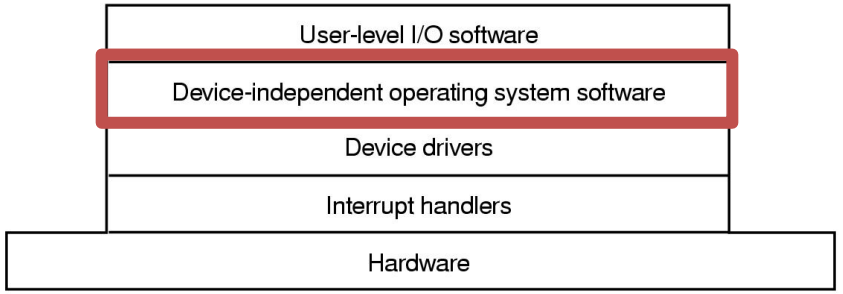


Common technique 2: Naming conventions

- On UNIX, symbolic names correspond to devices
  - Examples: /dev/disk0, /dev/tty
  - Software communicates with the drivers using special files having these names

  Example: The inode for /dev/disk0 will store the driver ID for the device, called the **major device number**. This inode will also store the unit to be read or written, called the **minor device number**.

Buffering



Buffering refers to the practice of accumulating data in an array for reading or writing.

More efficient than reading and writing individual bytes

A **circular buffer** stores data with two pointers, one for the next free space (for writing) and one for the start of the data (for reading

A **double buffer** collects data in a secondary buffer and then swaps with the primary buffer when full

- This technique is used a lot for rendering, to avoid flickering

Error Handling

Errors are common in I/O

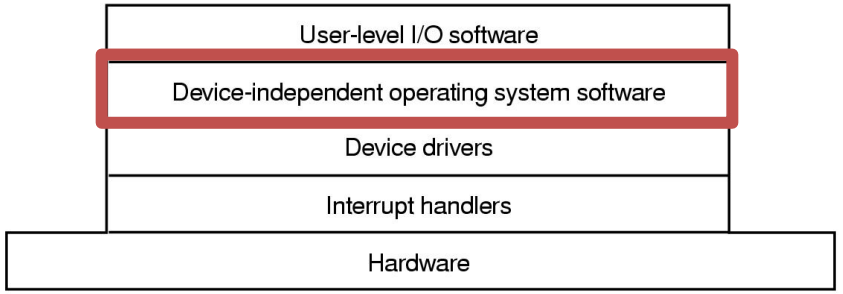
Recoverable Errors:

- Programming errors, e.g. accessing a device that doesn't exist; reading from an output device
  - Solution: Report an error back to the user program with an error code
- Device errors, e.g. writing to a damaged disk block, or accessing a device that is turned off
  - Solution: The driver should try to fix these by finding an undamaged block

Non-recoverable Errors:

- Example: a damaged superblock
  - Solution: Report error

Uniform interfacing

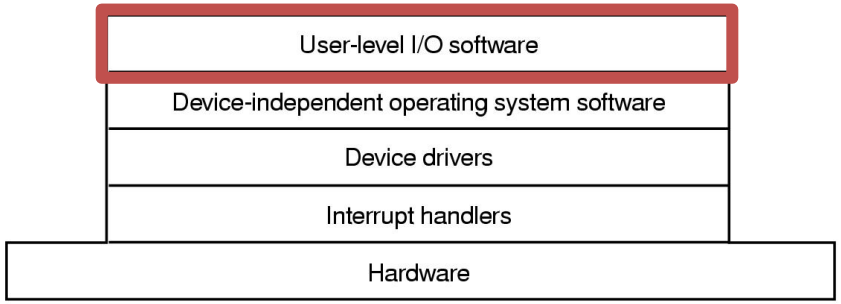


Common technique 2: Naming conventions

- On UNIX, symbolic names correspond to devices
  - Examples: /dev/disk0, /dev/tty
  - Software communicates with the drivers using special files having these names

  Example: The inode for /dev/disk0 will store the driver ID for the device, called the **major device number**. This inode will also store the unit to be read or written, called the **minor device number**.

User-level I/O



Includes features such as the following:

- User-level function calls, such as read or write
- Spooling for dedicated devices, such as printers

NOTE: **Spooling** refers to saving requests in files (in a **spooling directory**). A **daemon** is a process that waits and responds to requests. A printer daemon waits for print requests and then chooses which request is satisfied next.

User-level I/O: Graphical User Interfaces

A **graphical user interface (GUI)** is a visual interface for working with a computer

Keyboard, mouse, screen have device controllers

Example: keyboards have a register with the scan code (e.g. pressed key)

Keyboard-drivers can be either character-based (**raw mode**) or line-based (**cooked mode**, or **canonical mode**).

Mode determines when events are reported (e.g. when interrupts are sent)

Most terminals will automatically **echo** keys to the screen as the user is typing them

Mice report changes in x or y position as well as any buttons (left/middle/right) that are pressed.

User-level I/O: Graphical User Interfaces

The **X Window System** is the basis for nearly all UNIX user interfaces

- developed at M.I.T. as part of project Athena in the 1980s
- portable (e.g. compiles on different machine easily)
- runs entirely in user space (unlike Windows, which integrates its GUI features into the OS)

An **X Server**

- Collects input from the keyboard and mouse
- Writes output to the screen
- Keeps track of the active windows (location of mouse pointer)

Running GUI programs are X clients. The **window manager** of your desktop is an example.

An **intrinsics** layer manages buttons, scroll bars, and other GUI elements, called widgets

GUIs also have high-level APIs that apply a consistent look and feel to widgets. Examples:

- Examples: Motif, Gnome, KDE
- APIs: GTK+, Qt

User-level I/O: Graphical User Interfaces

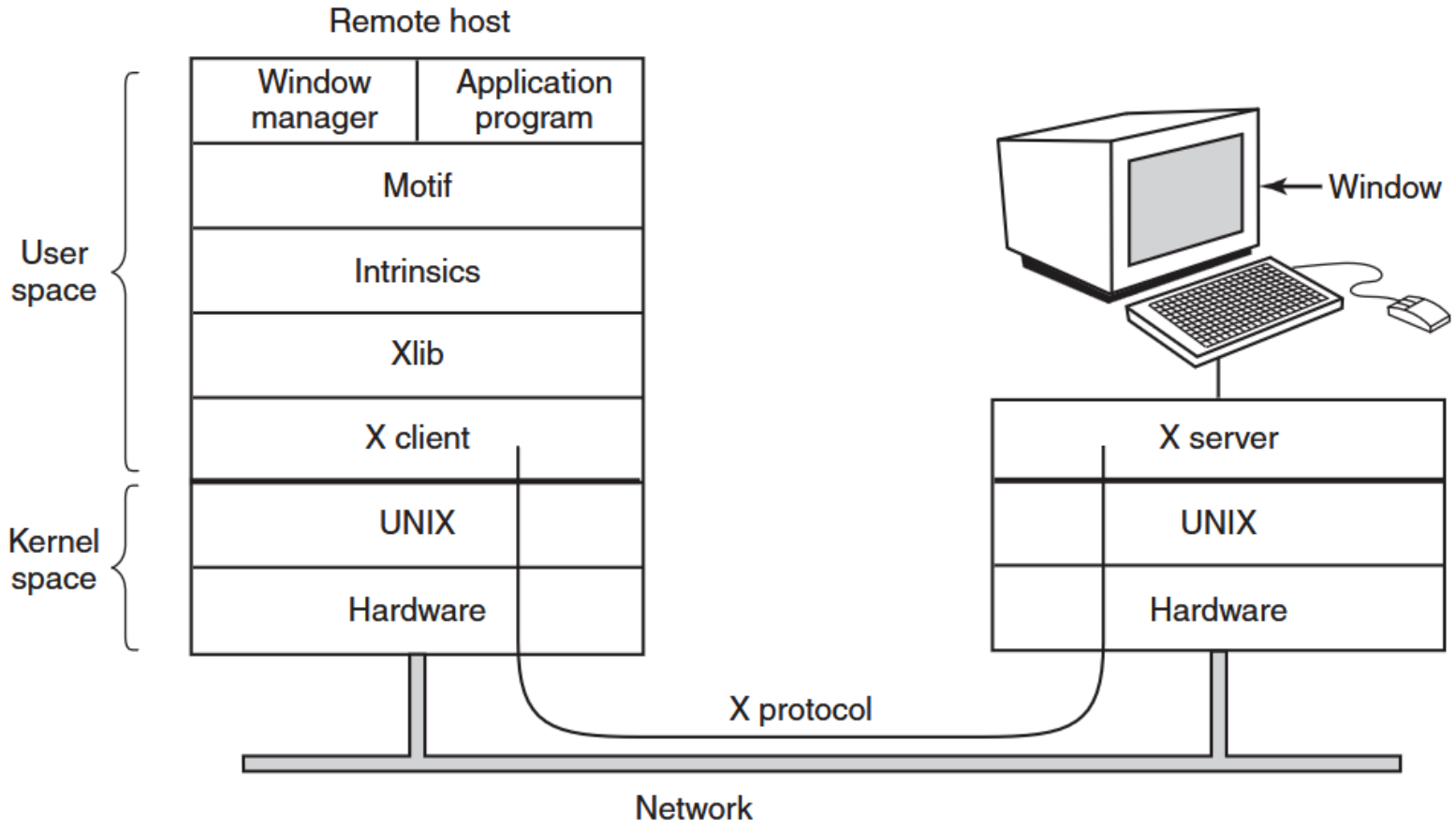


Figure 5-33. Clients and servers in the M.I.T. X Window System.

Example: Button (X11)

```
while (1) {
      XNextEvent(display, &event);
      if (event.type == Expose) {
         // Draw button rectangle
         XSetForeground(display, gc, BlackPixel(display, screen));
         XDrawRectangle(display, window, gc, BUTTON_X, BUTTON_Y, BUTTON_WIDTH, BUTTON_HEIGHT);
         XDrawString(display, window, gc, BUTTON_X + 15, BUTTON_Y + 25, "Close", 5);
      }

      if (event.type == ButtonPress) {
         int x = event.xbutton.x;
         int y = event.xbutton.y;
         if (x >= BUTTON_X && x <= BUTTON_X + BUTTON_WIDTH &&
            y >= BUTTON_Y && y <= BUTTON_Y + BUTTON_HEIGHT) {
            break;  // Exit loop on button click
         }
      }
   }
```

Example: Button (Qt)

```
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QWidget window;
    window.setFixedSize(200, 150);
    window.setWindowTitle("Qt Button Example");

    QPushButton *button = new QPushButton("Close", &window);
    button->setGeometry(50, 50, 100, 40);

    // Connect the button's clicked signal to the window's close slot
    QObject::connect(button, &QPushButton::clicked, &window, &QWidget::close);

    window.show();
    return app.exec();
}
```

Example: Button (GTK+)

```
static void on_button_clicked(GtkWidget *widget, gpointer window) {
   gtk_window_close(GTK_WINDOW(window));
}
...
   // Create a new button
   button = gtk_button_new_with_label("Close");

   // Connect the button click signal to the callback function
   g_signal_connect(button, "clicked", G_CALLBACK(on_button_clicked), window);

   // Add the button to the window
   gtk_container_add(GTK_CONTAINER(window), button);

   // Connect the destroy signal so the app quits when the window is closed
   g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

   gtk_widget_show_all(window);
   gtk_main();
```

Summary: The I/O System

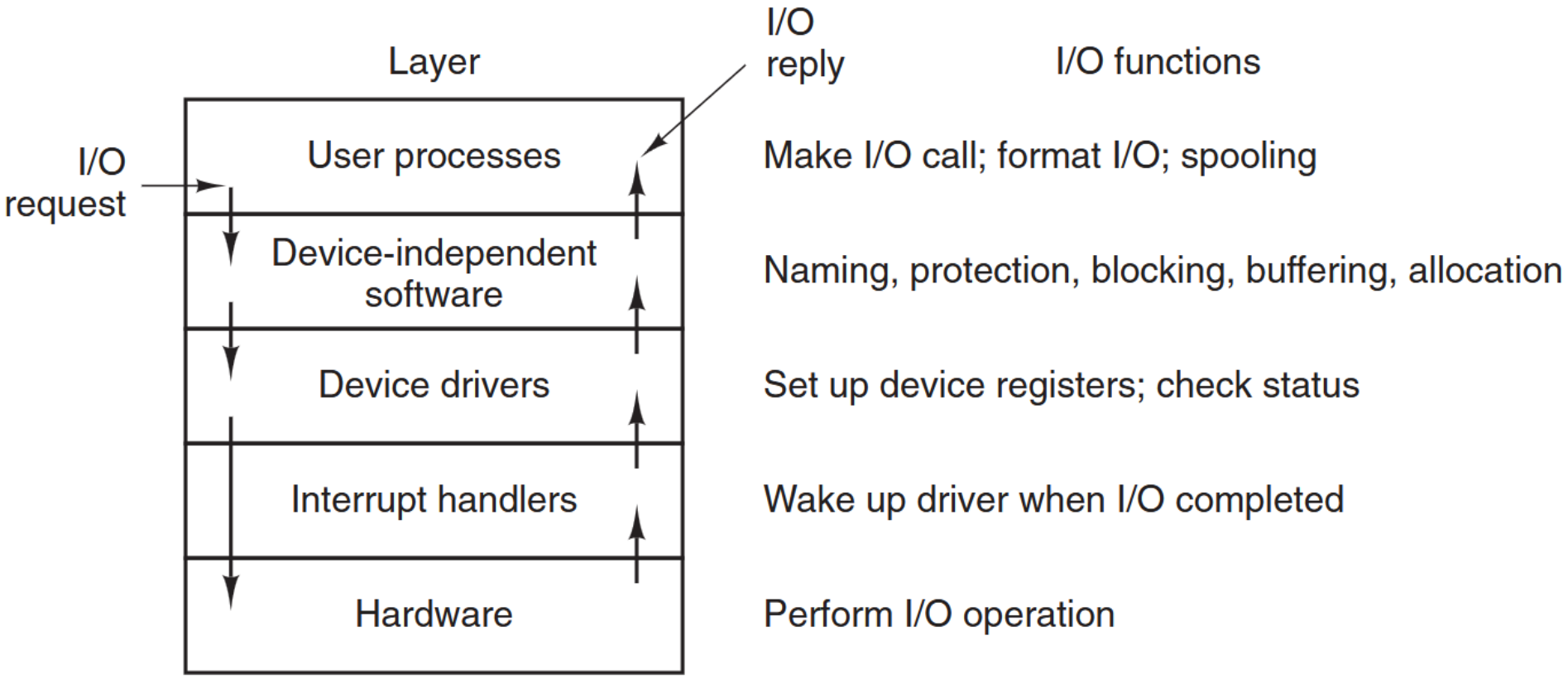


Figure 5-17. Layers of the I/O system and the main functions of each layer.

Summary: Principles of I/O Programming

- Device independence
  - Programs can access any I/O device without specifying device in advance
- Uniform naming
  - Name of a file or device should be a string or an integer
  - Not dependent on which machine
  - On Unix, directories such as /dev/lp, /dev/tty
- Error handling
  - Handle as close to the hardware as possible

Summary: Design choices for I/O Software

- Synchronous vs. asynchronous transfers
  - Polling or waiting transfers vs. interrupt-driven
- Buffering
  - Data coming off a device often cannot be stored in final destination directly
- Sharable vs. dedicated devices
  - Disks are sharable
  - Printers are dedicated (only one can use it at a time)