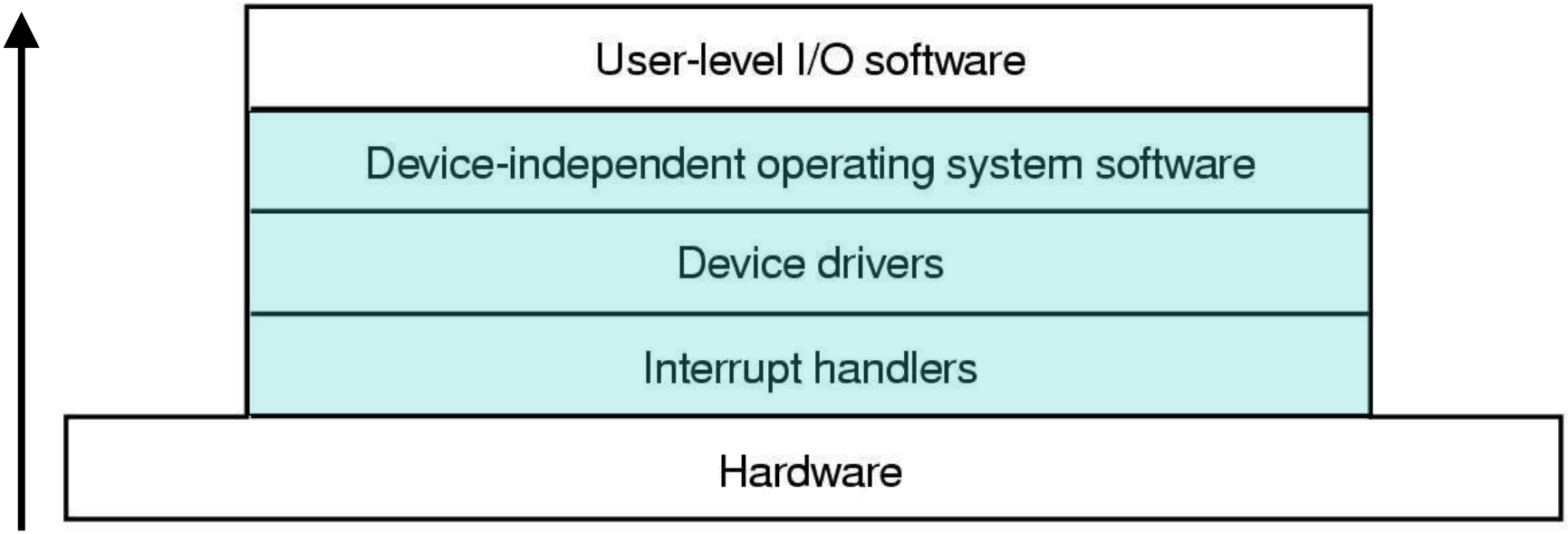
I/O

Goals for I/O Handling

- Enable use of peripheral devices
- Present a uniform interface for
  - Users (files etc.)
  - Devices (drivers)
- Hide the details of devices from users (and non-I/O OS subsystems)

How to provide interfaces



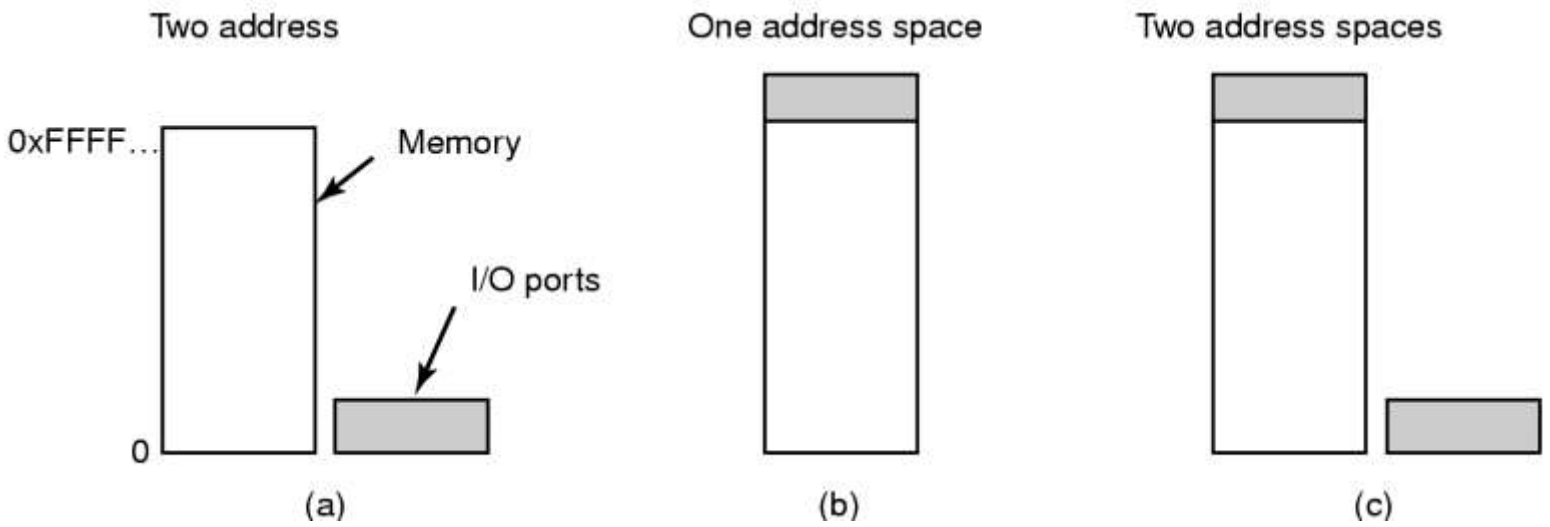
Hardware: Device Controllers

- I/O devices have:
  - mechanical components
  - electronic components
- The *device controller* component
  - may be able to handle multiple devices
  - is the device interface accessible to the OS
- Controller's tasks
  - convert bit stream to block of bytes
  - perform error correction as necessary

Accessing Devices (1)

- Most device controllers provide
  - buffers (in / out)
  - control registers
  - status registers
- These are accessed from the OS/Apps
  - I/O ports
  - memory-mapped
  - hybrid

Accessing Devices (2)

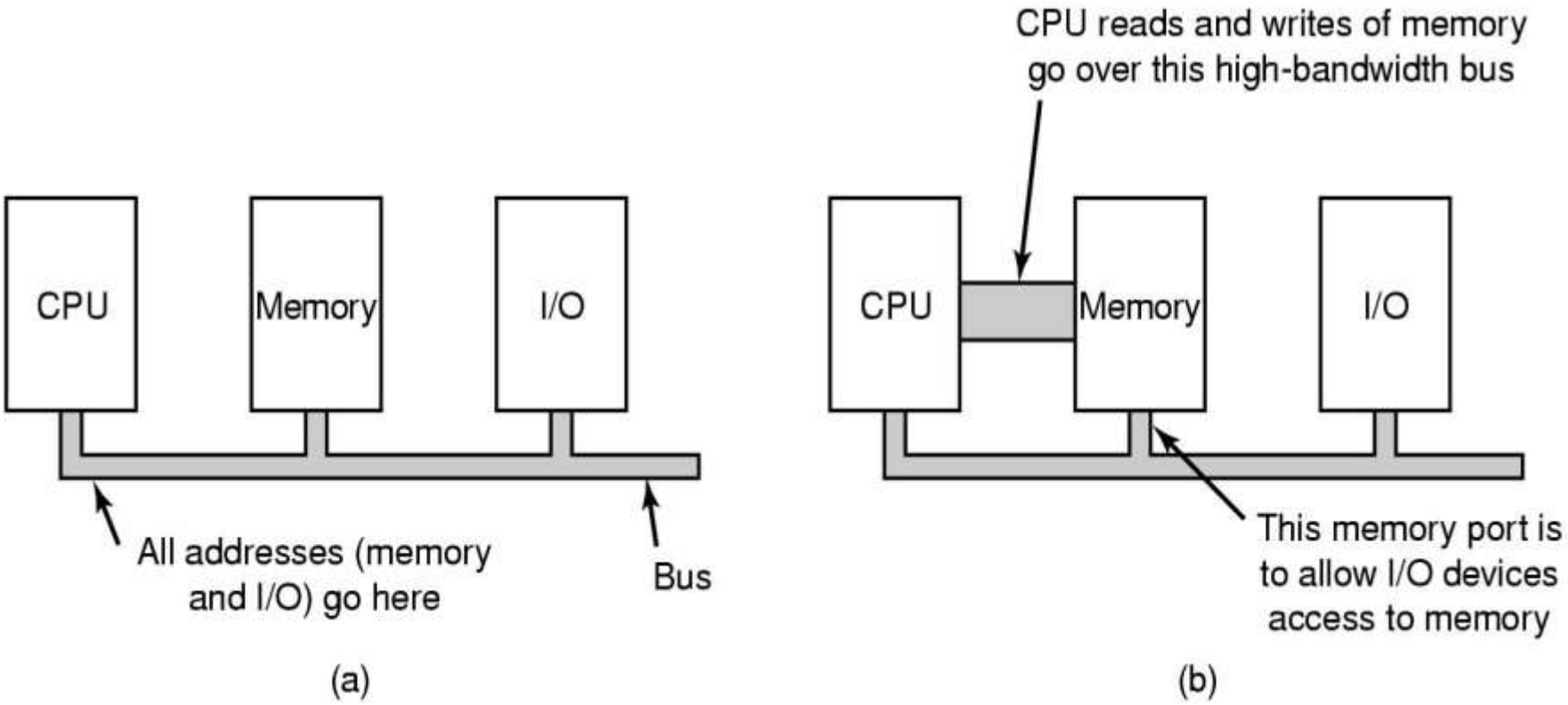


a) Separate I/O and memory space
b) Memory-mapped I/O
c) Hybrid

Memory-Mapped I/O

+ No special instruction in assembler needed
+ No special protection needed
  (inherit from virtual memory)
+ Testing status registers directly
  Not load and test!
- Caching of status registers?
- One bus?

Memory-Mapped I/O

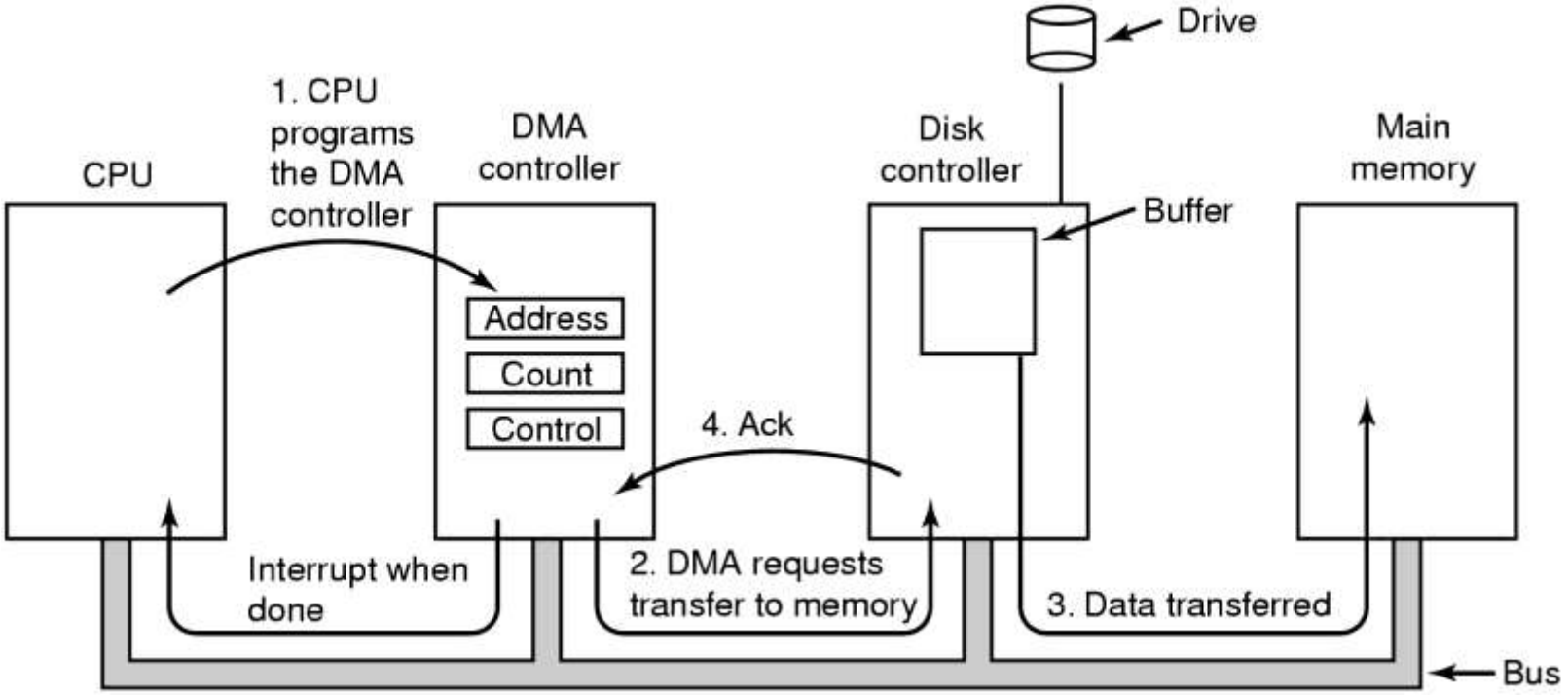


(a) A single-bus architecture
(b) A dual-bus memory architecture

Shuffling Data

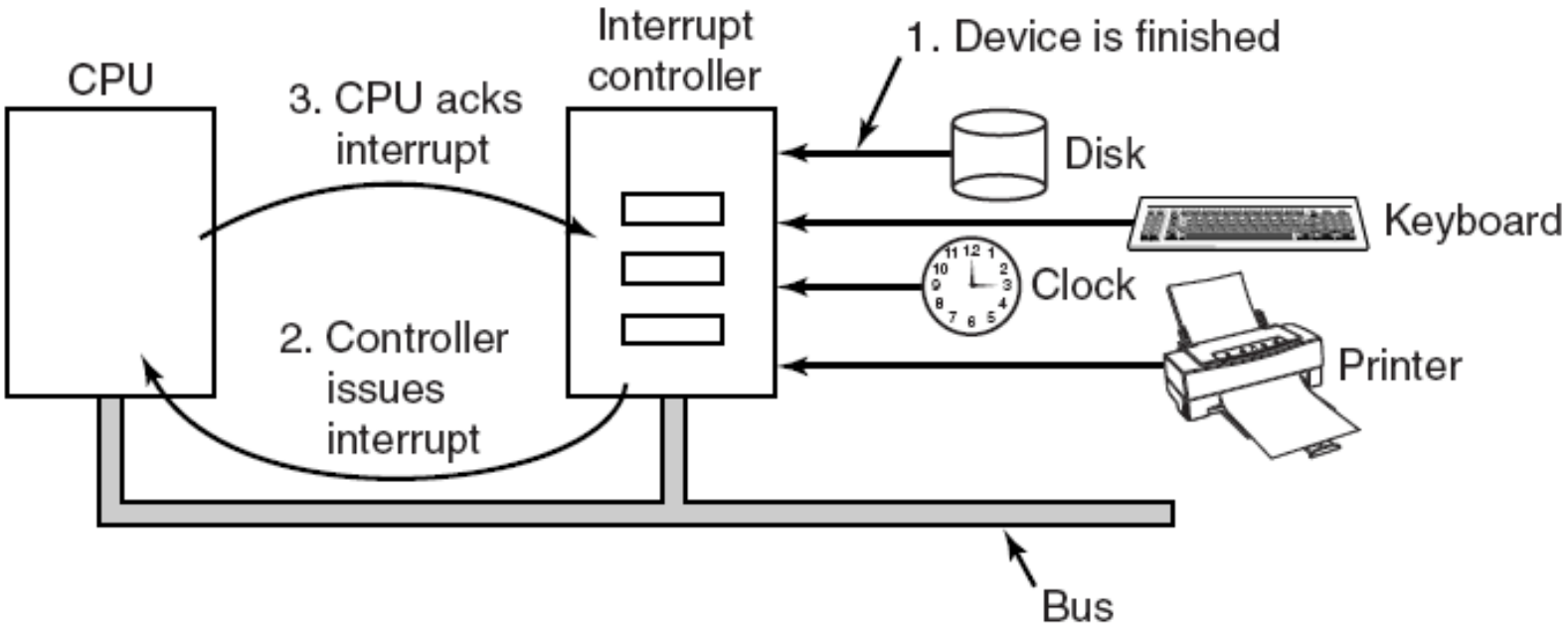
- Data needs to be transferred between device controllers & memory
  - one byte at a time
  - several bytes at a time
- Using the CPU to shuffle small amounts of data may be inefficient → Direct Memory Access (DMA)

Direct Memory Access (DMA)



Operation of a DMA transfer

Interrupts



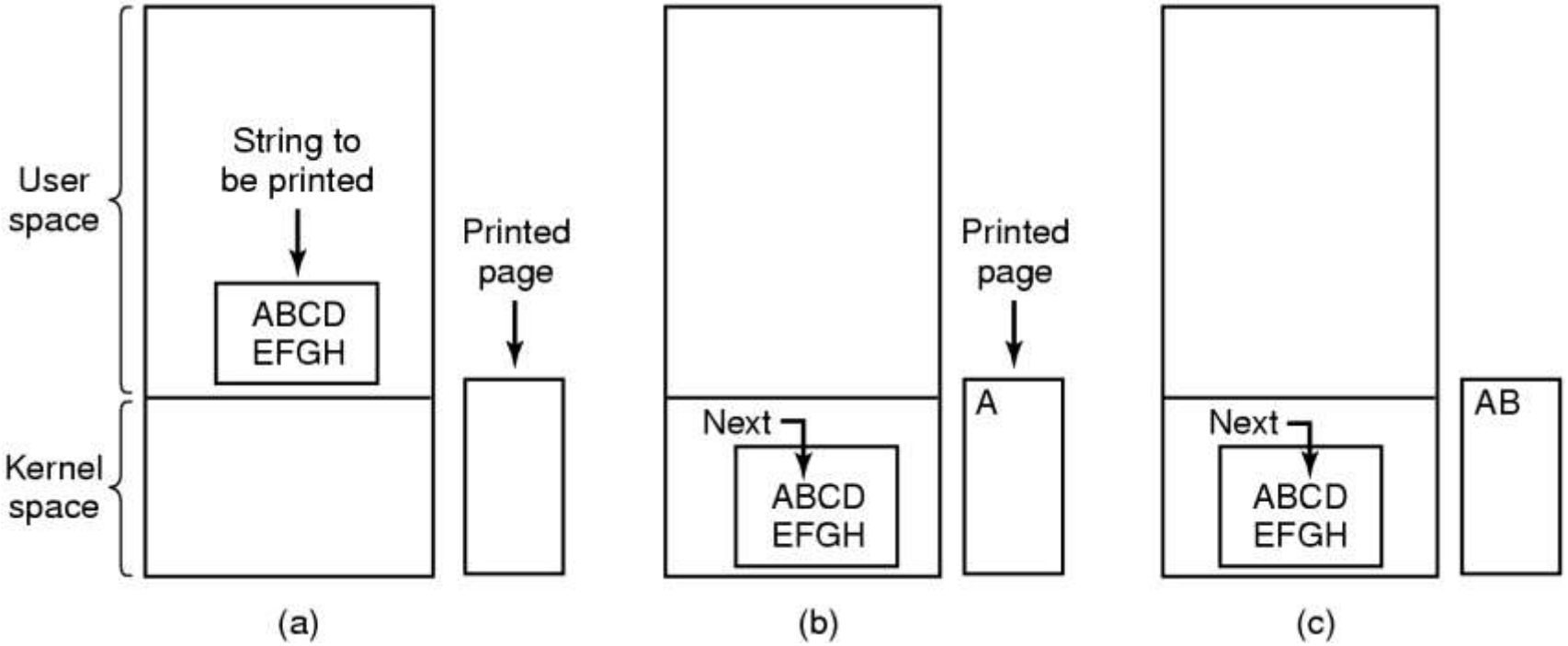
Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires

I/O

There are three kinds of I/O handling

- Programmed I/O
  - "Do it all yourself"
- Interrupt-driven I/O
  - "Here you are, now tell me when its done"
- DMA-based I/O
  - "Let someone else do it"

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( );
```

Interrupt-Driven I/O

Code for system call

```
copy_from_user(buffer,  p, count);
enable_interrupts( );
while (*printer_status_reg != READY)   ;
*printer_data_register  = p[0];
scheduler( );
```

Code for interrupt handler

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register  = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

Writing a string to the printer using interrupt-driven I/O

I/O Using DMA

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

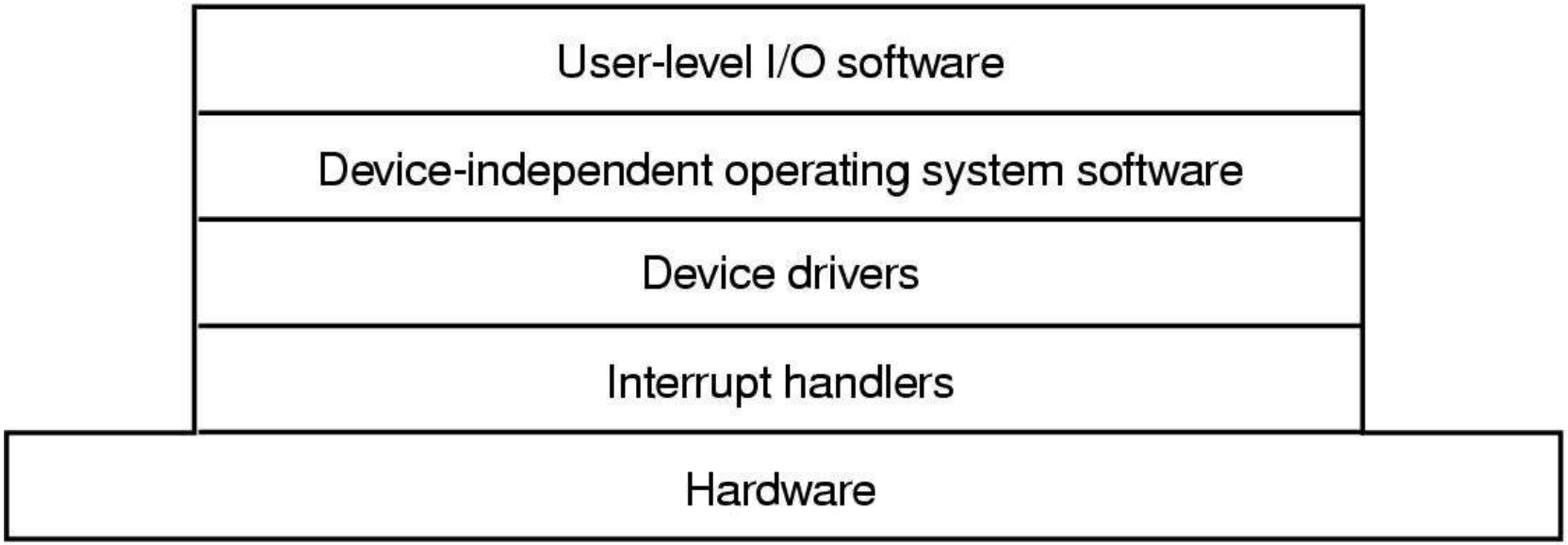
(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

- Printing a string using DMA
  - a) code executed when the print system call is made
  - b) interrupt service procedure

I/O Software Layers



Layers of the I/O Software System

Interrupt Handlers

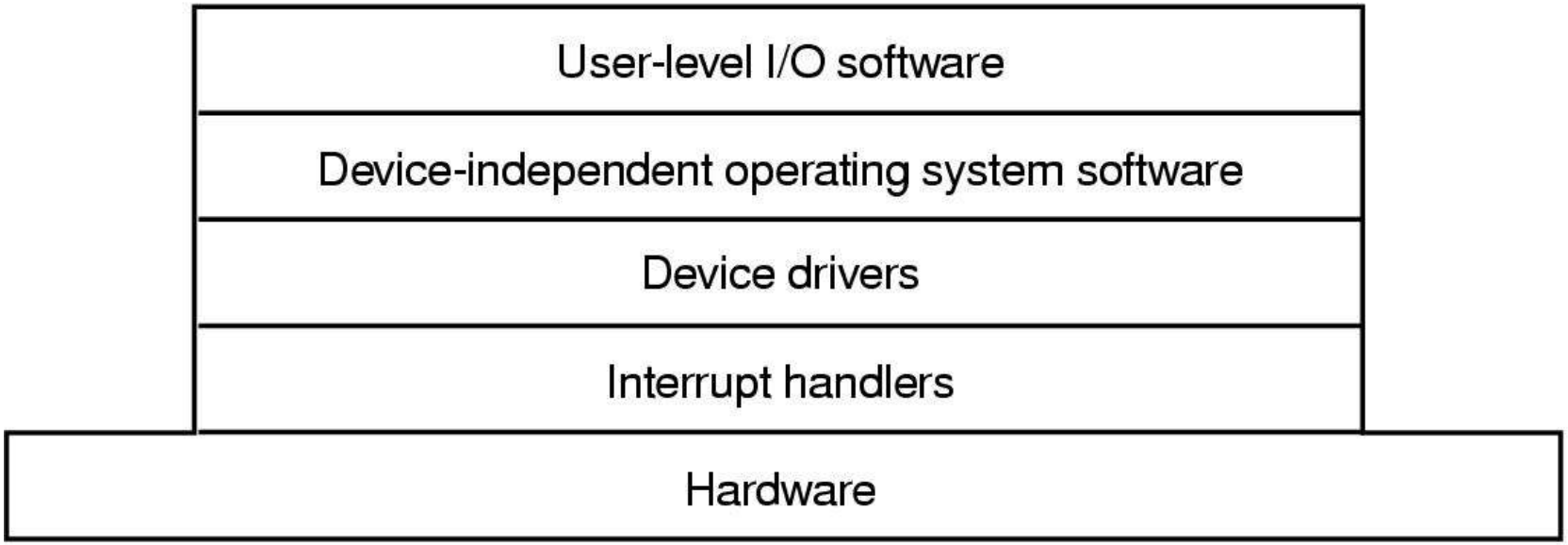
- Interrupt handlers should be fast (why?)
- Sometimes there is a lot of things to do
  - Copying buffers, waking up processes, starting new I/O ops, etc.
- Solution: split in two parts
  - Top half: disabled interrupts, only essential work
  - Bottom half: enabled interrupts, does most of the work

Interrupt Service Routine

Steps that must be performed in software when interrupt occurs (no complete list)

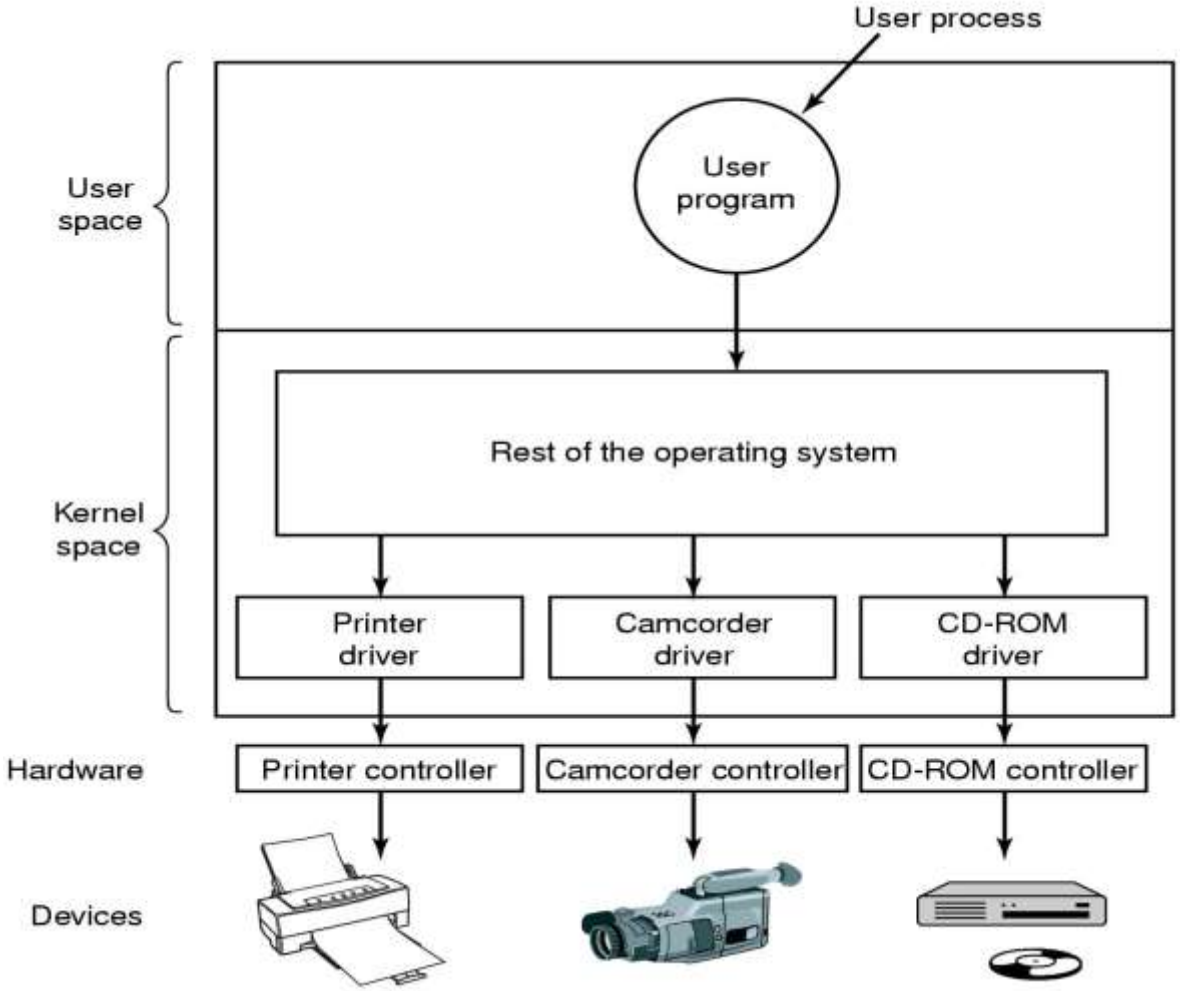
1. Save regs not already saved by interrupt hardware
2. Set up context for interrupt service procedure
3. Ack interrupt controller (& reenable interrupts)
4. Run service procedure
5. Schedule and run a new process (+ new context)

I/O Software Layers



Layers of the I/O Software System

Device Drivers (1)



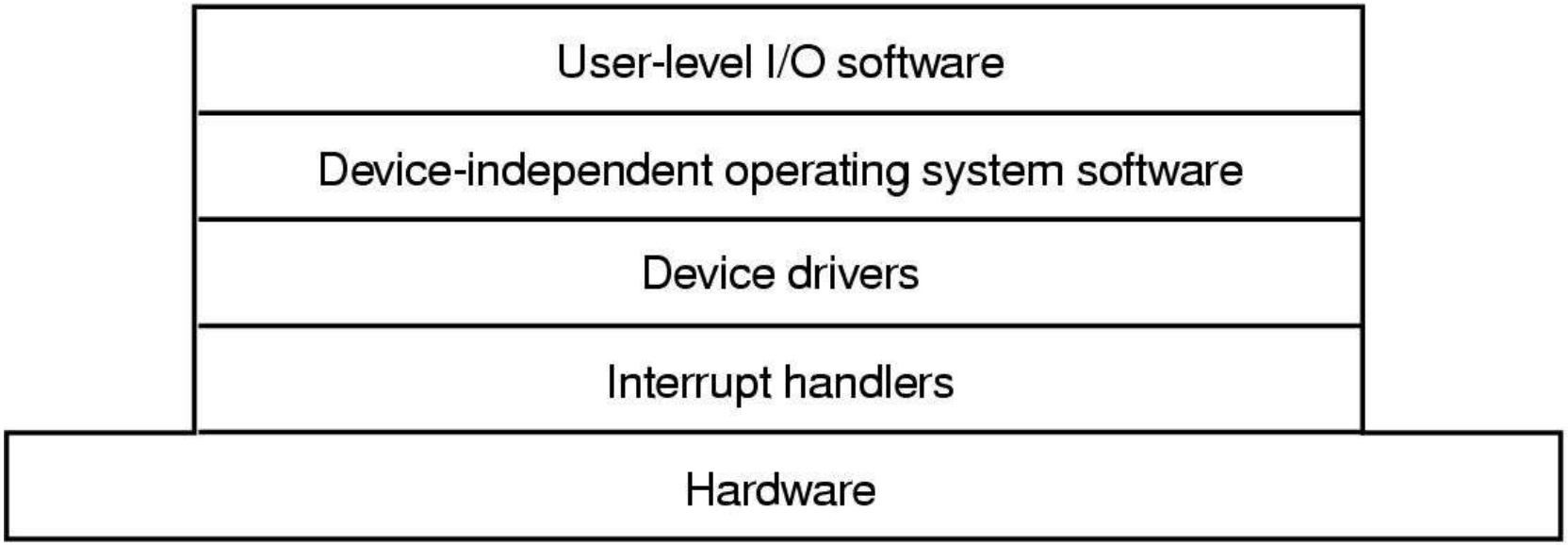
Logical position of device drivers

Device Drivers (2)

- Classically we distinguish two types
  - Block devices (disks…)
  - Character devices (keyboards, printers…)

- Handles (typically)
  - Abstract requests "reads" and "writes"
  - Communication with device controller
  - Initialization (registering interrupt handlers, etc.)
  - Power management

I/O Software Layers



Layers of the I/O Software System

Goals for I/O Handling

- Enable use of peripheral devices
- Present a uniform interface for
  - Users (files etc.)
  - Devices (drivers)
- Hide the details of devices from users (and non-I/O OS subsystems)

"Uniform interface" / "Hiding details"

- Device independence
  - programs can access any I/O device
  - without specifying device in advance
    - (file on USB flash drive, hard drive, or CD-ROM)
- Uniform naming
  - name a file or device as a string or an integer
  - not depending on the device
- Error handling
  - handle as close to the HW as possible

Challenges for I/O Software

- Synchronous vs. asynchronous transfers
  - blocking transfers vs. interrupt-driven
  - OS may make interrupt-driven operations look like blocking to the user
- Buffering
  - data coming off a device cannot be stored in final destination
  - OS should buffer for pre-processing/RT…
- Sharable vs. dedicated devices
  - disks are sharable
  - tape drives would not be
  - OS should be able to support both

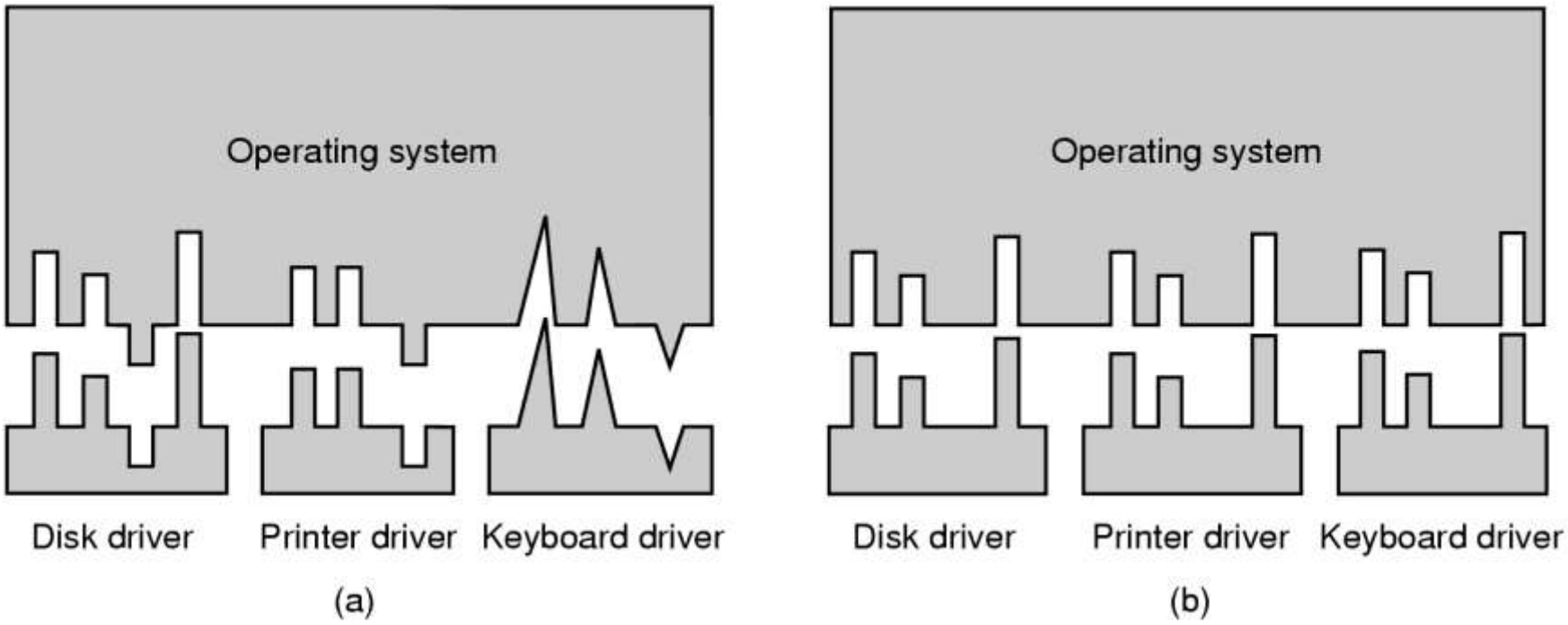
Device-Independent I/O Software

Generic I/O management in the kernel

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

Functions of the device-independent I/O software

Uniform Interfacing for Device Drivers



(a) Without a standard driver interface

(b) With a standard driver interface

Uniform Interface: Naming

- How are devices addressed?

- UNIX/Linux
  - Major number
    - Identifying the device driver
  - Minor number
    - Identifying the (virtual) device

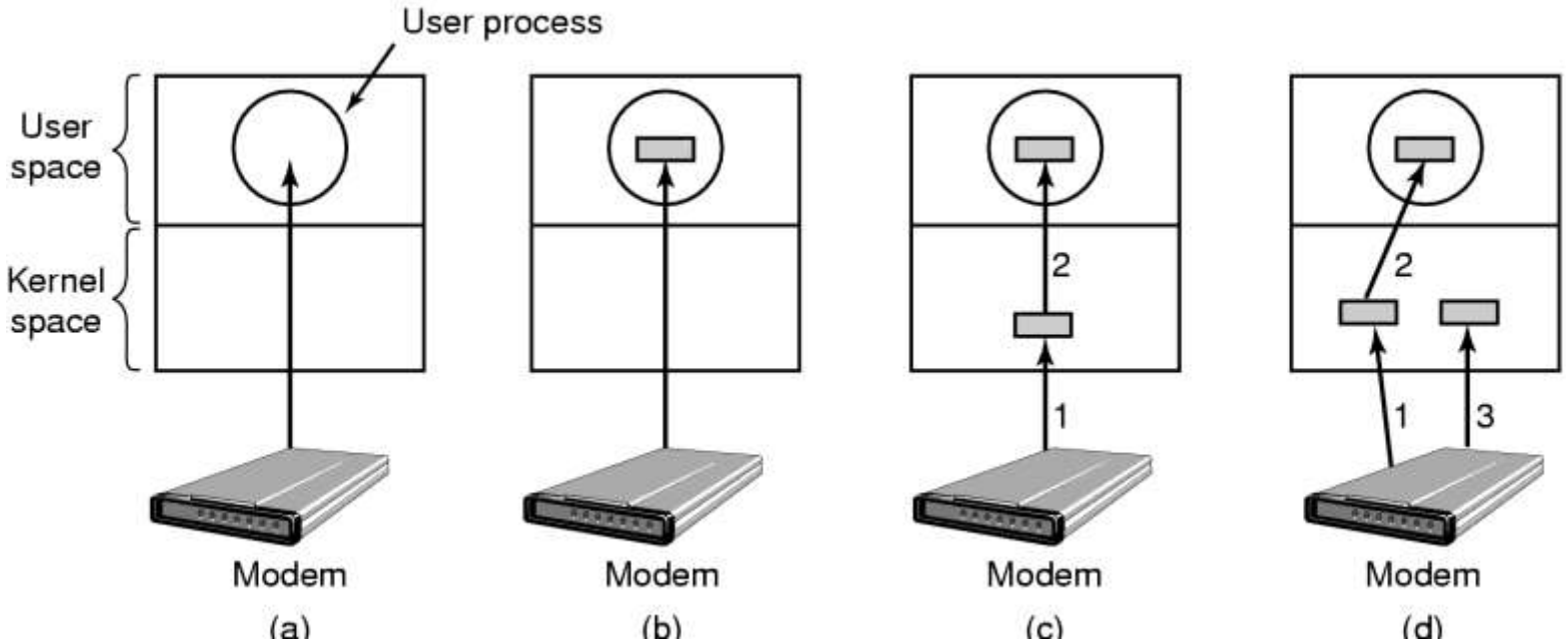
Linux Driver Registration

- Drivers (modules) need to register with the kernel
- Kernel keeps table of drivers
- Special kernel routines for adding/deleting entries

Buffering

- Where to put the buffers
  - User
  - Kernel
- What if the buffer is full?
- How many buffers?
- When to notify the user?

Buffer Strategies

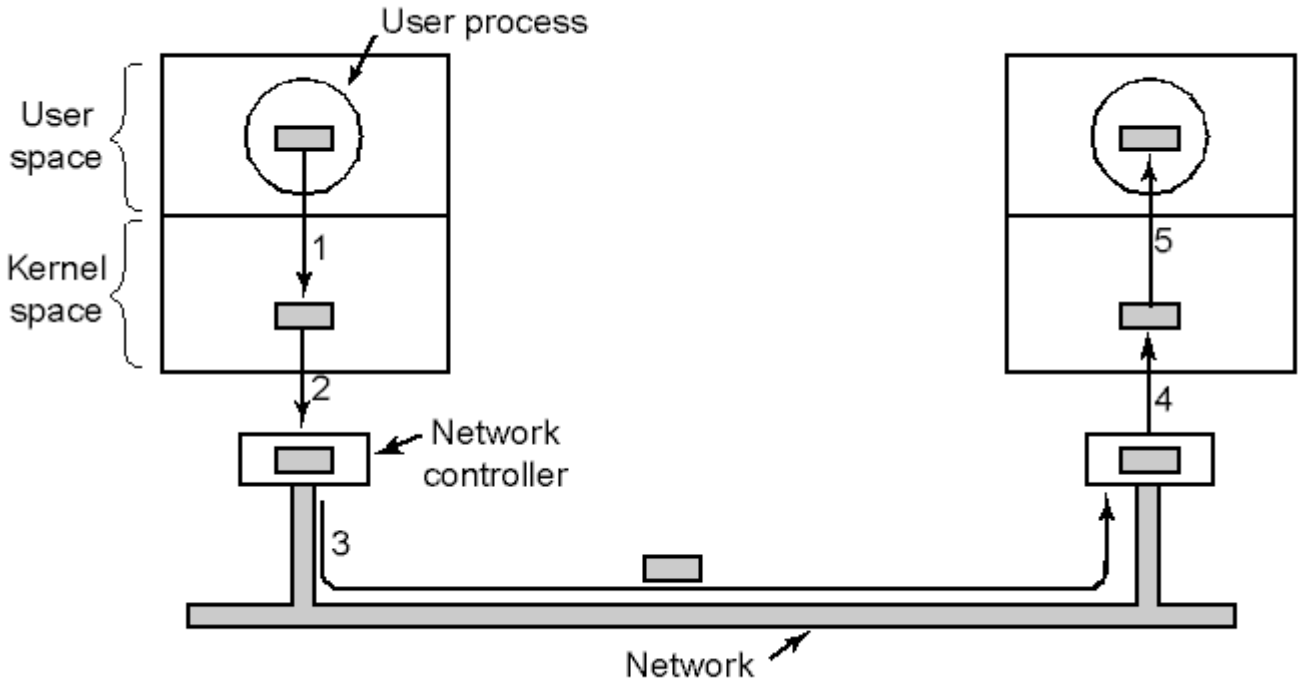


(a) Unbuffered input
(b) Buffering in user space
(c) Buffering in the kernel followed by copying to user space
(d) Double buffering in the kernel

Buffer Strategies Summarized

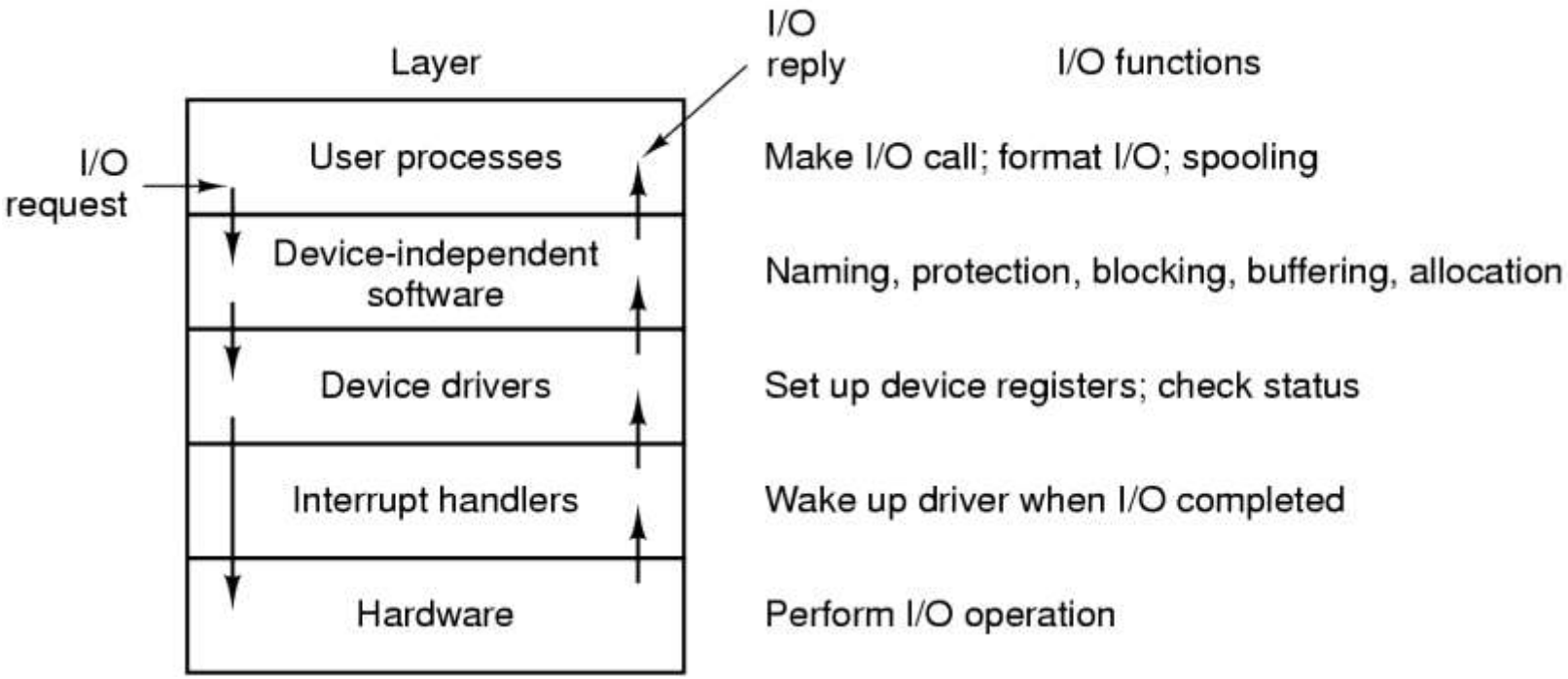
- Where to store data to be sent?
  - User buffer
    - Locking?
  - Kernel buffer
    - No locking
  - Device controller

Latency Impact of Buffers



Networking may involve many copies

User-Space I/O Software



Layers of the I/O system and their main functions