

# Input/Output

Week 12 - Lecture

Interrupts and Algorithms

# Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - Nikita Lozhnikov (also Tutorial Instructor)
  - Manuel Rodriguez
  - Shokhista Ergasheva

# Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013  
Prentice-Hall, Inc.and customised for the needs of this course.
- Additional input for the slides are detailed later

# Interrupts Revisited (1)

- In a typical personal computer system, the interrupt structure is as shown in Fig. 5-5
- At the hardware level, interrupts work as follows:
- When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned
- This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do

# Interrupts Revisited (2)

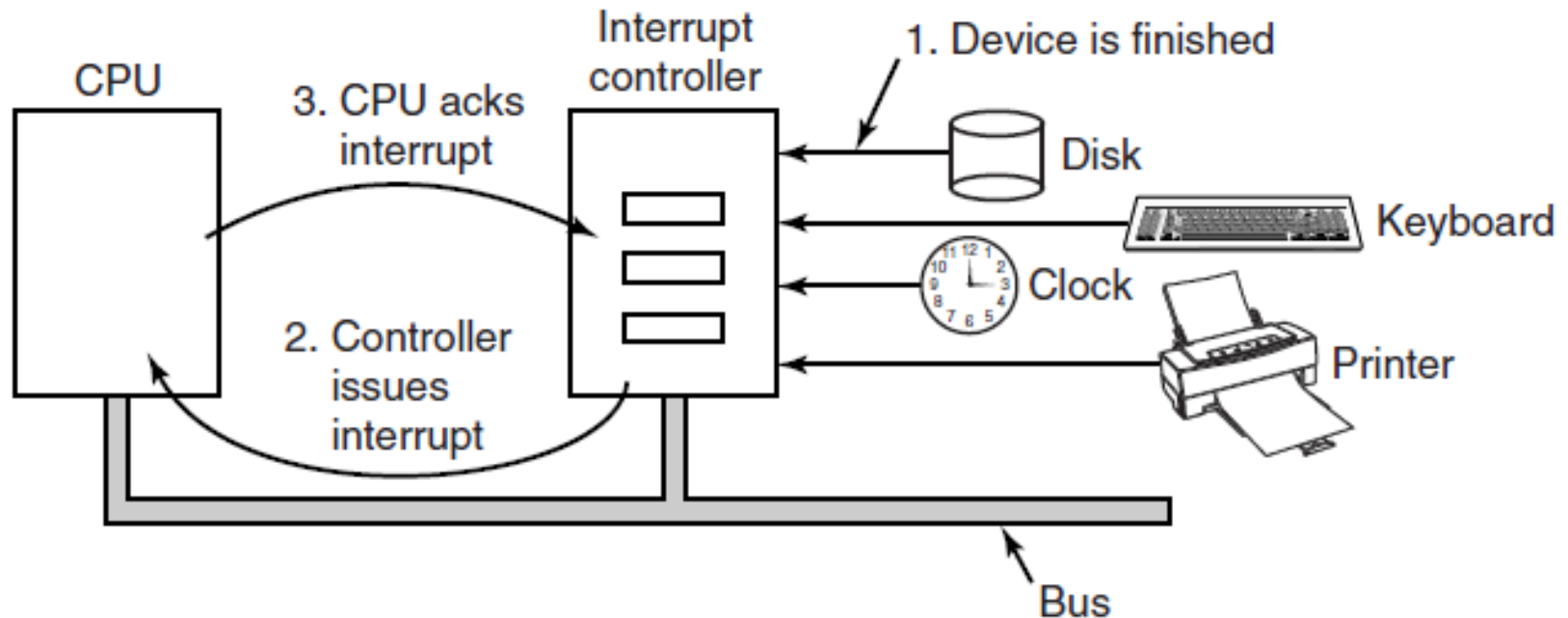


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

# Interrupts Revisited (3)

- To handle the interrupt, the controller puts a number on the address lines specifying which device wants attention and asserts a signal to interrupt the CPU
- The interrupt signal causes the CPU to stop what it is doing and start doing something else

# Interrupts Revisited (4)

- The number on the address lines is used as an index into a table called the interrupt vector to fetch a new program counter
- This program counter points to the start of the corresponding interrupt-service procedure

# Precise Interrupt (1)

- An interrupt that leaves the machine in a well-defined state is called a **precise interrupt**



# Precise Interrupt (2)

- Four properties of a precise interrupt:
  - The PC is saved in a known place
  - All instructions before the one pointed to by the PC have completed
  - No instruction beyond the one pointed to by PC has finished
  - The execution state of the instruction pointed to by the PC is known

# Imprecise Interrupt

- An interrupt that does not meet the requirements such as “all instructions up to the program counter have completed and none of those beyond it have started”, is called an **imprecise interrupt**

# Precise vs. Imprecise (1)

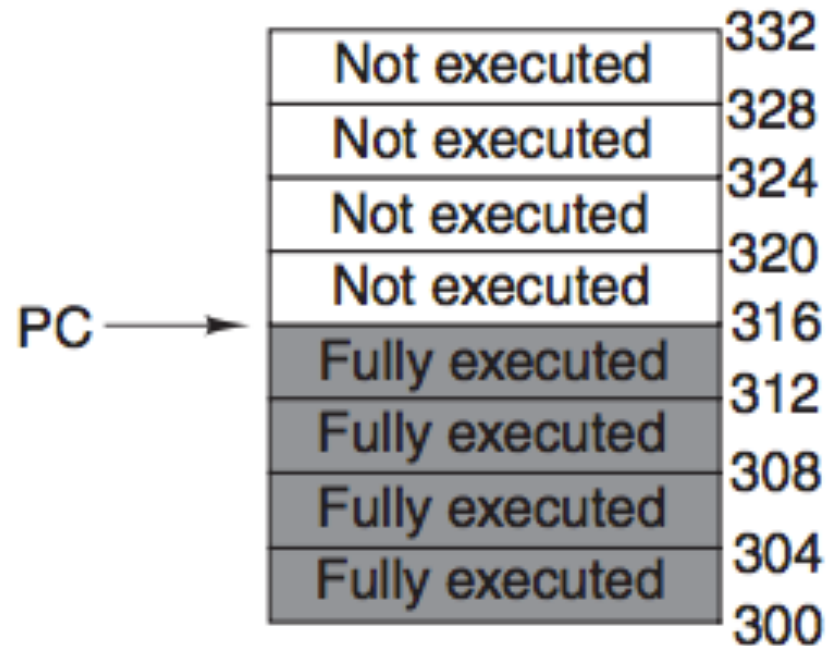
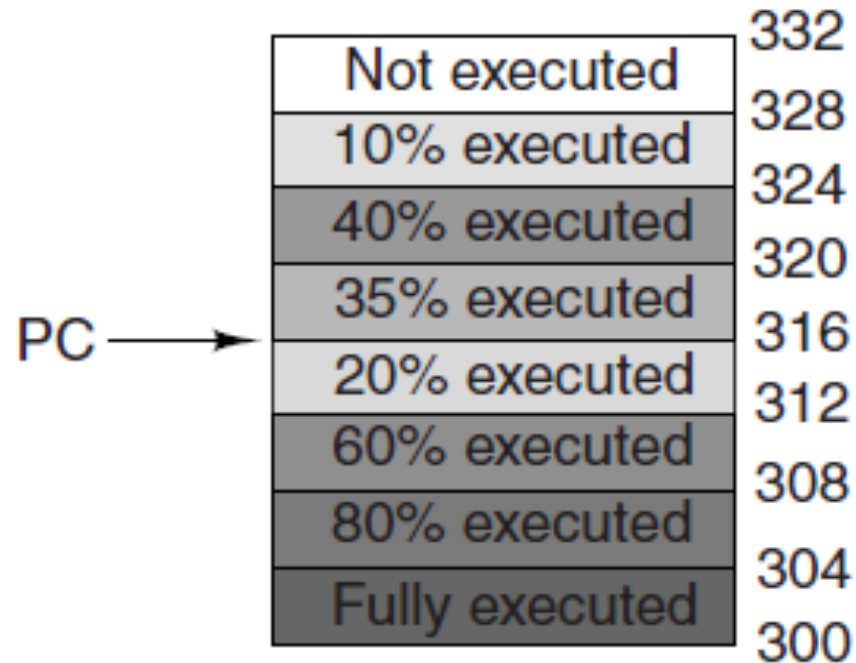


Figure 5-6. (a) A precise interrupt

# Precise vs. Imprecise (2)



The diagram illustrates an imprecise interrupt. It shows a vertical stack of nine states, each with a percentage of execution completed and a corresponding memory address. The states are: 'Not executed' (332), '10% executed' (328), '40% executed' (324), '35% executed' (320), '20% executed' (316), '60% executed' (312), '80% executed' (308), and 'Fully executed' (304). The addresses decrease from top to bottom. A Program Counter (PC) is shown on the left with an arrow pointing to the '35% executed' state at address 320. The '40% executed' state is highlighted with a darker background, and the 'Fully executed' state is also highlighted.

Not executed	332
10% executed	328
40% executed	324
35% executed	320
20% executed	316
60% executed	312
80% executed	308
Fully executed	304
	300

Figure 5-6. (b) An imprecise interrupt

# Goals of the I/O Software (1)

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

# Device Independence

- We should be able to write programs that can access any I/O device without having to specify the device in advance. Such as:
  - a program that reads a file as input should be able to read a file on a hard disk, a DVD, or on a USB stick without having to be modified for each different device
  - similarly, one should be able to type a command such as **sort <input> output** and have it work with input coming from any kind of disk or keyboard and the output going to any kind of disk or the screen

# Uniform Naming

- The name of a file or a device should simply be a string or an integer and not depend on the device in any way
- Example: A USB stick can be mounted on top of the directory **/usr/ast/backup**
- So that copying a file to **/usr/ast/backup/monday** copies the file to the USB stick
- In this way, all files and devices are addressed the same way such as by a path name

# Error Handling

- Errors should be handled as close to the hardware as possible
- If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again
- In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error



# Asynchronous vs. Synchronous Transfers

- Asynchronous (Interrupt Driven):
  - Most physical I/O is asynchronous
  - The CPU starts the transfer and goes off to do something else until the interrupt arrives
- Synchronous (Blocking):
  - User programs are much easier to write if the I/O operations are blocking
  - After a *read()* system call the program is automatically suspended until the data are available in the buffer

# Buffering

- Often data that come off a device cannot be stored directly in their final destination. For example:
  - when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it
- Buffering involves considerable copying and often has a major impact on I/O performance

# Programmed I/O (1)

- The simplest form of I/O is to have the CPU do all the work. This method is called programmed I/O

# Programmed I/O (2)

- Example:
  - Consider a user process that wants to print the eight-character string “ABCDEFGH” on the printer via a serial interface
  - The software first assembles the string in a buffer in user space, as shown in Fig. 5-7(a)

# Programmed I/O (3)

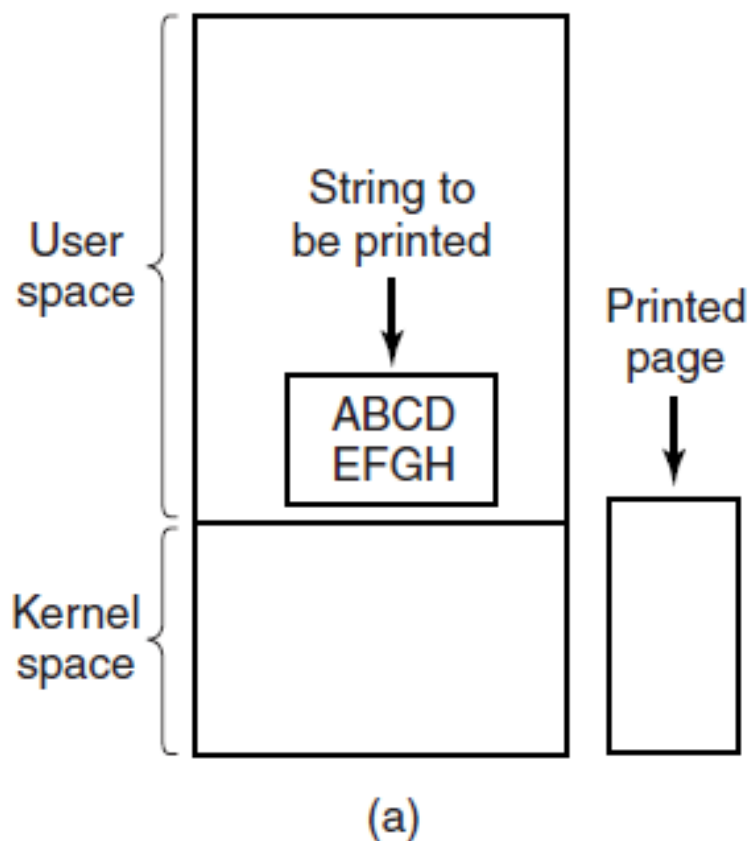


Figure 5-7. Steps in printing a string. Step 1

# Programmed I/O (4)

- The user process then acquires the printer for writing by making a system call to open it
- If the printer is currently in use by another process, this call will fail and return an error code or will block until the printer is available
- Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer

# Programmed I/O (5)

- The operating system then (usually) copies the buffer with the string to an array, say,  $p$ , in kernel space, where it is more easily accessed
- As soon as the printer is available, the operating system copies the first character to the printer's data register, in this example using memory-mapped I/O
- In Fig. 5-7(b), however, we see that the first character has been printed and that the system has marked the "B" as the next character to be printed

# Programmed I/O (6)

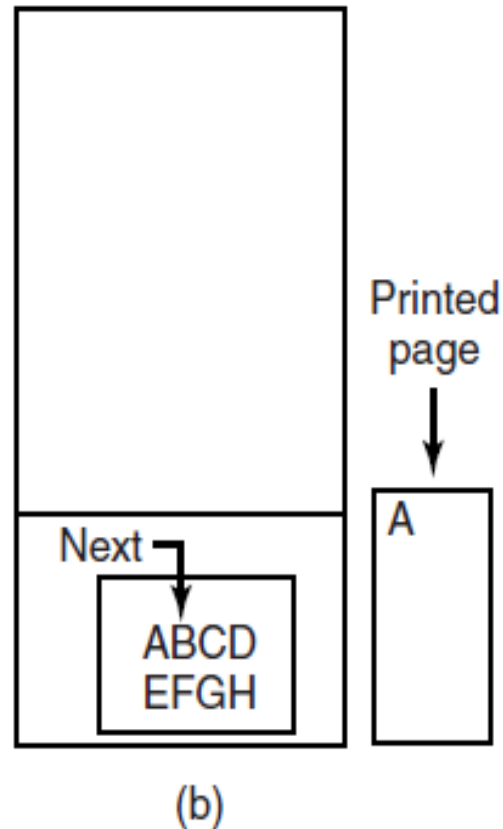


Figure 5-7. Steps in printing a string. Step 2



# Programmed I/O (7)

- When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it
- At this point the operating system waits for the printer to become ready again
- When that happens, it prints the next character, as shown in Fig. 5-7(c). This loop continues until the entire string has been printed. Then control returns to the user process

# Programmed I/O (8)

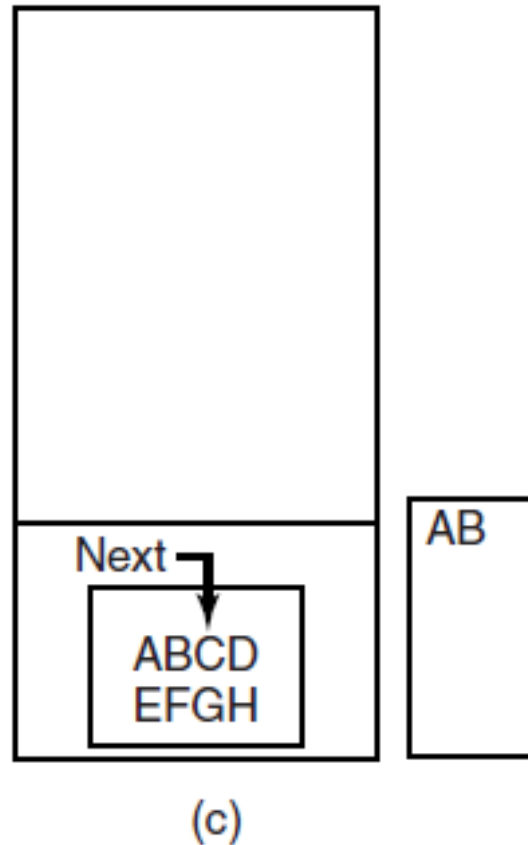


Figure 5-7. Steps in printing a string. Step 3

# Programmed I/O (9)

- The actions followed by the operating system are briefly summarized in Fig. 5-8
- First the data are copied to the kernel. Then the operating system enters a tight loop, outputting the characters one at a time
- The essential aspect of programmed I/O, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called polling or busy waiting

# Programmed I/O (10)

```
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 (1)

- The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts
- For Example:
  - When the system call to print the string is made, the buffer is copied to kernel space, and the first character is copied to the printer as soon as it is willing to accept a character
  - At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed

# Interrupt-Driven I/O (2)

- The work done on the system call is shown in Fig. 5-9(a)
- When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt-service procedure is run. A crude version of this code is shown in Fig. 5-9(b)

# Interrupt-Driven I/O (3)

```
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 (1)

- The DMA controller feed the characters to the printer one at time, without the CPU being bothered
- DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU
- This strategy requires special hardware (the DMA controller) but frees up the CPU during the I/O to do other work



# I/O Using DMA (2)

```
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 (1)

- I/O software is typically organized in four layers, as shown in Fig. 5-11
- Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers

# I/O Software Layers (2)

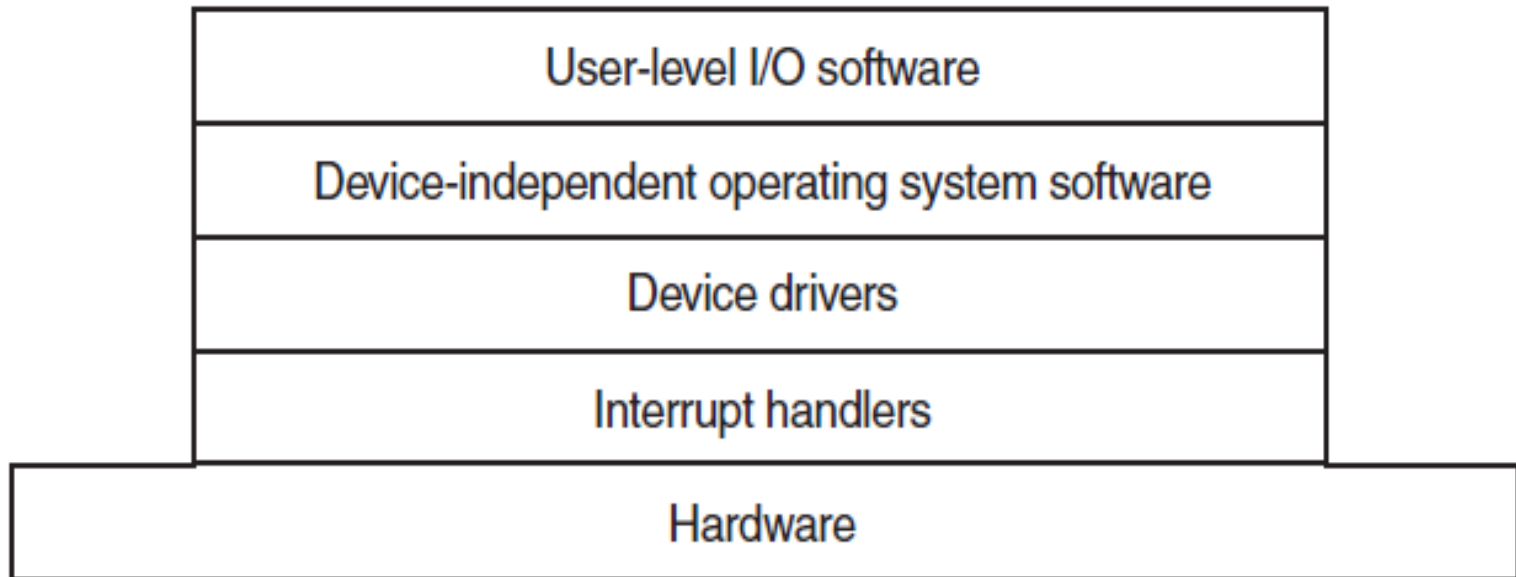


Figure 5-11. Layers of the I/O software system

# Interrupt Handlers (1)

- Typical steps after hardware interrupt completes:
  - Save registers (including the PSW) not already saved by interrupt hardware
  - Set up context for interrupt service procedure
  - Set up a stack for the interrupt service procedure
  - Acknowledge interrupt controller. If no centralized interrupt controller, reenale interrupts
  - Copy registers from where saved to process table

# Interrupt Handlers (2)

- Typical steps after hardware interrupt completes:
  - Run interrupt service procedure. Extract information from interrupting device controller's registers
  - Choose which process to run next
  - Set up the MMU context for process to run next
  - Load new process' registers, including its PSW
  - Start running the new process

# Device Drivers (1)

- Each I/O device attached to a computer needs some device specific code for controlling it. This code, called the **device driver**, is generally written by the device's manufacturer and delivered along with the device
- Each device driver normally handles one device type, or at most, one class of closely related devices

# Device Drivers (2)

- For example:
  - A SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI Blu-ray disk as well.
  - On the other hand, a mouse and joystick are so different that different drivers are usually required.
- Device drivers are normally positioned below the rest of the operating system, as is illustrated in Fig. 5-12

# Device Drivers (3)

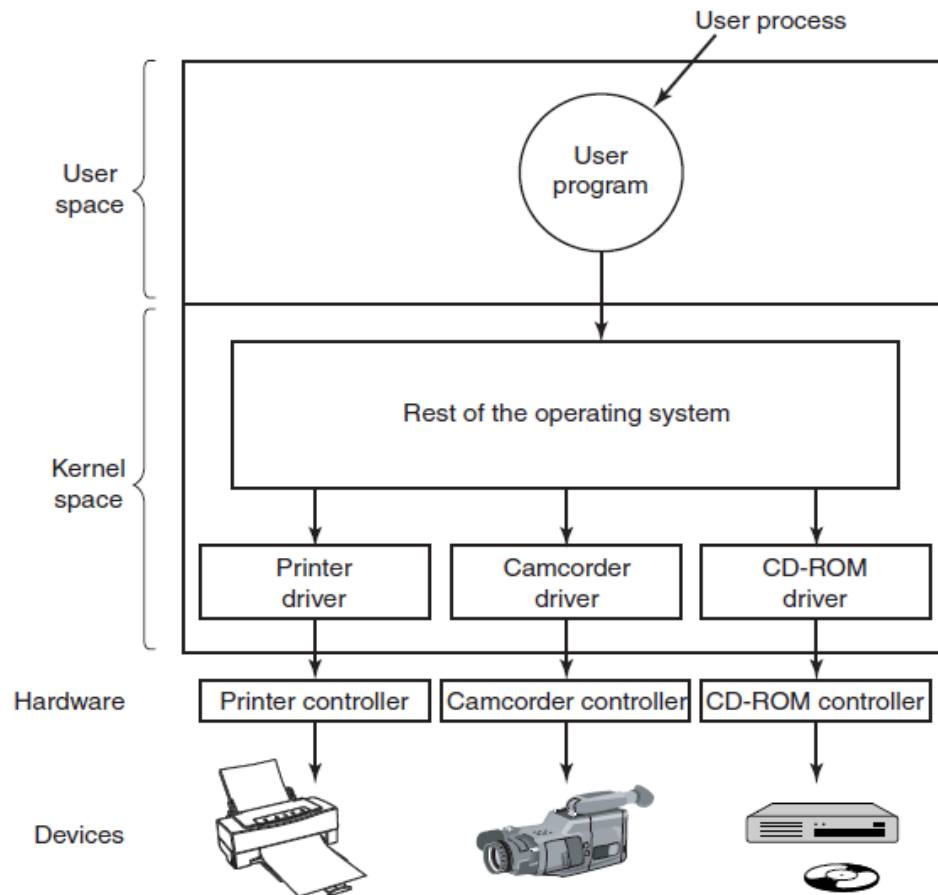


Figure 5-12. Logical positioning of device drivers.

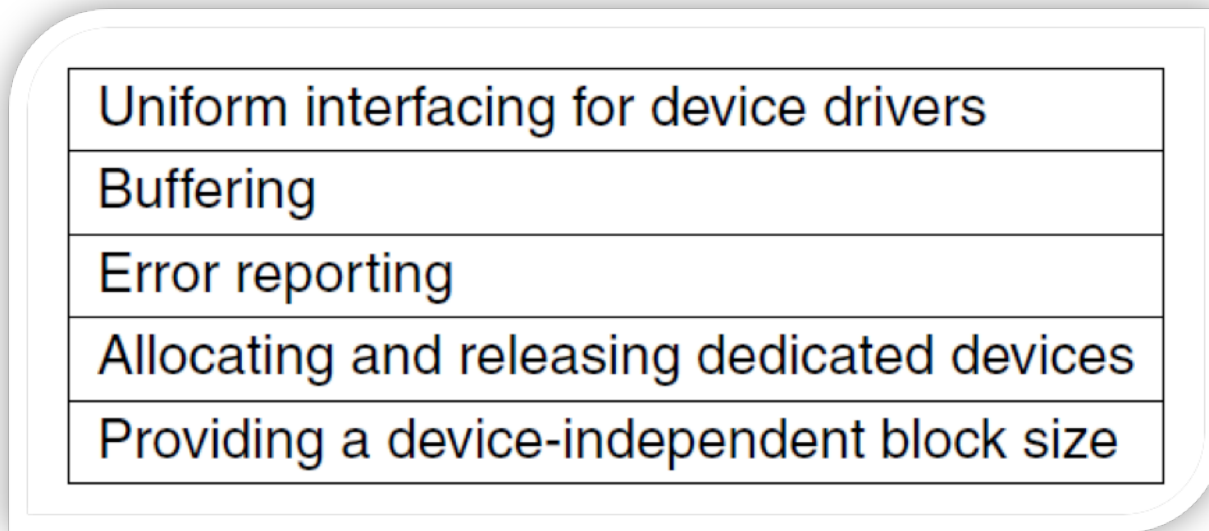
In reality all communication between drivers and device controllers goes over the bus



# Device-Independent I/O Software (1)

- Although some of the I/O software is device specific, other parts of it are device independent
- The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the drivers
- The functions shown in Fig. 5-13 are typically done in the device-independent software

# Device-Independent I/O Software (2)



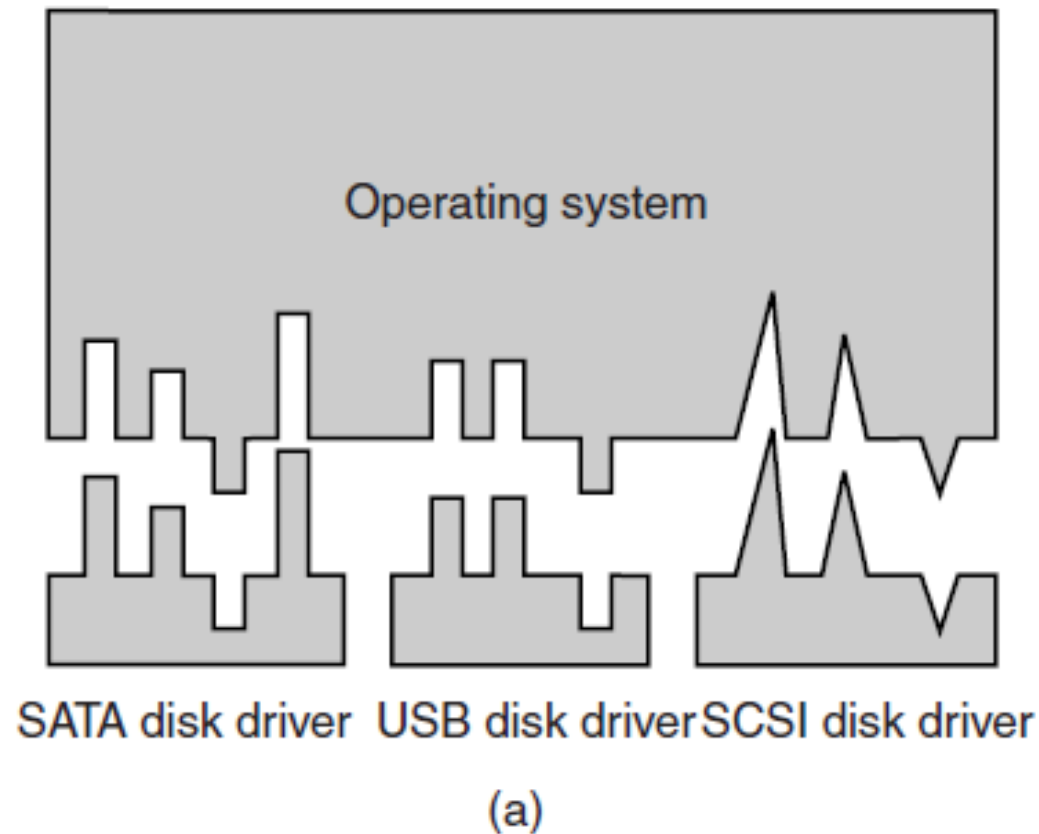
Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

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

# Uniform Interfacing for Device Drivers (1)

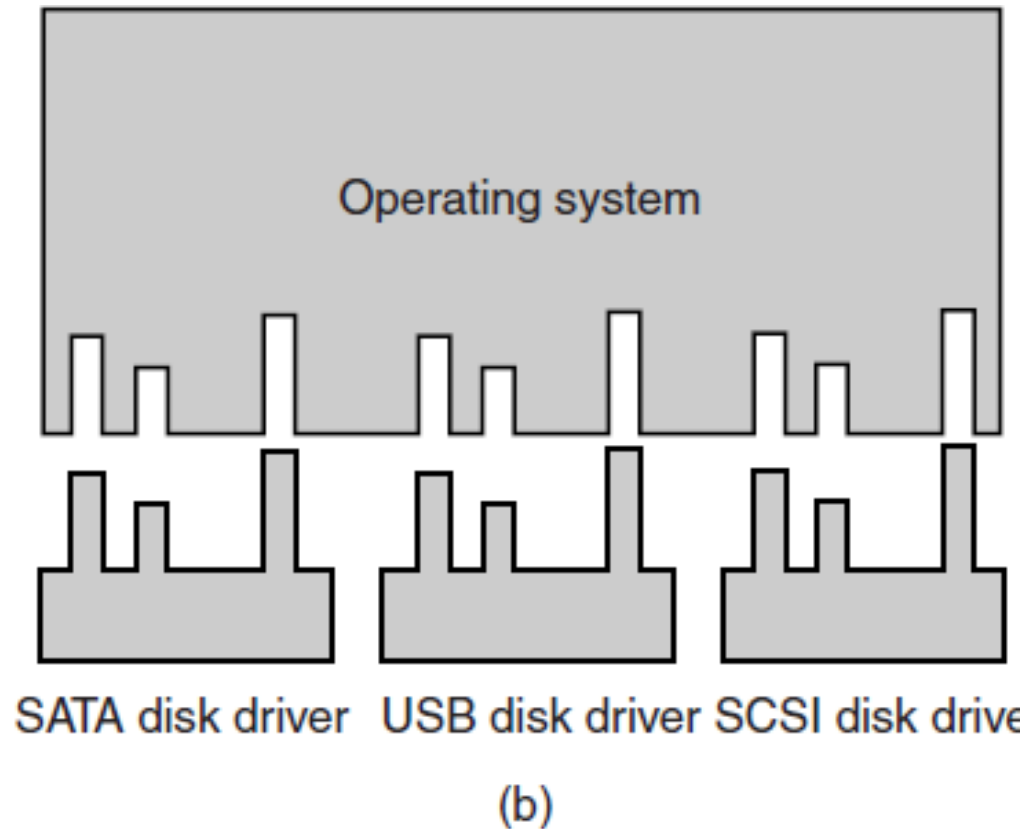
- A major issue in an operating system is how to make all I/O devices and drivers look more or less the same
- If disks, printers, keyboards, and so on, are all interfaced in different ways, every time a new device comes along, the operating system must be modified for the new device
- Fig. 5-14(a) illustrates a situation in which each device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver

# Uniform Interfacing for Device Drivers (2)



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

# Uniform Interfacing for Device Drivers (3)



- Figure 5-14. (b) With a standard driver interface

# Unbuffered Input (1)

- Consider a process that wants to read data from an ADSL (Asymmetric Digital Subscriber Line) modem:
  - One possible strategy for dealing with the incoming characters is to have the user process do a read system call and block waiting for one character
  - Each arriving character causes an interrupt
  - The interrupt-service procedure hands the character to the user process and unblocks it

# Unbuffered Input (2)

- After putting the character somewhere, the process reads another character and blocks again. This model is indicated in Fig. 5-15(a)
- The trouble with this way of doing business is that the user process has to be started up for every incoming character
- Allowing a process to run many times for short runs is inefficient, so this design is not a good one

# Unbuffered Input (3)

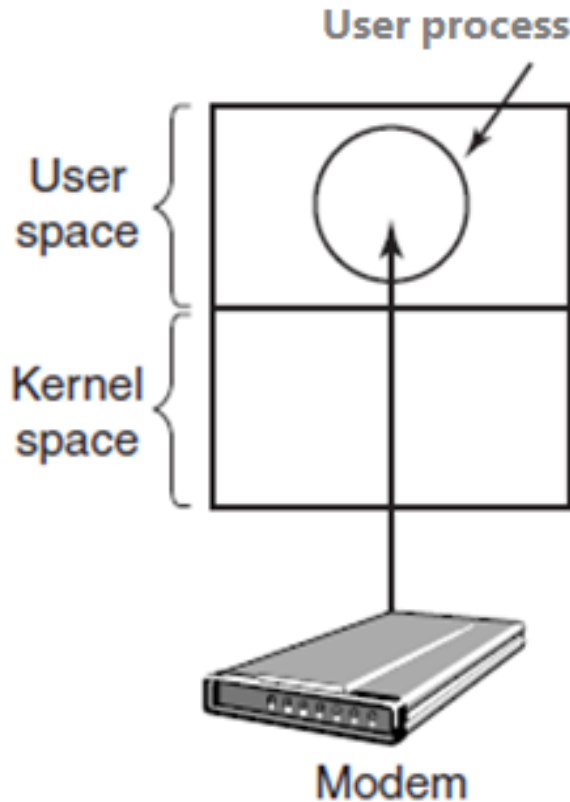


Figure 5-15. (a) Unbuffered input.



# Buffering in User Space (1)

- An improvement is shown in Fig. 5-15(b).
  - Here the user process provides an  $n$ -character buffer in user space and does a read of  $n$  characters
  - The interrupt-service procedure puts incoming characters in this buffer until it is completely full. Only then does it wakes up the user process
  - Drawback: what happens if the buffer is paged out when a character arrives?

# Buffering in User Space (2)

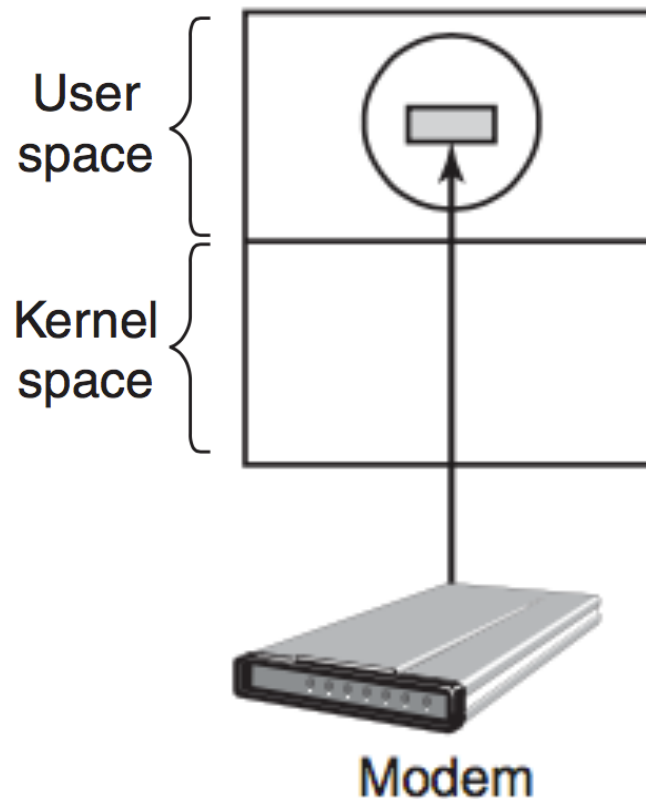


Figure 5-15. (b) Buffering in user space

# Buffering in the Kernel (1)

- Yet another approach is to create a buffer inside the kernel and have the interrupt handler put the characters there, as shown in Fig. 5-15(c)
- When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation. This scheme is far more efficient

# Buffering in the Kernel (2)

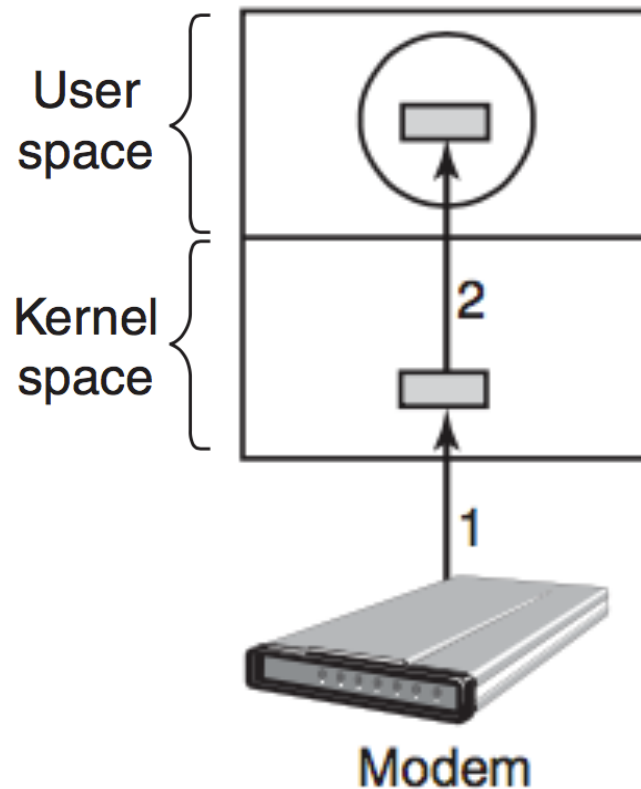


Figure 5-15. (c) Buffering in the kernel followed by copying to user space

# Double Buffering in the Kernel (1)

- What happens to characters that arrive while the page with the user buffer is being brought in from the disk?
  - Since the buffer is full, there is no place to put them. A way out is to have a second kernel buffer
  - After the first buffer fills up, but before it has been emptied, the second one is used, as shown in Fig. 5-15(d)

# Double Buffering in the Kernel (2)

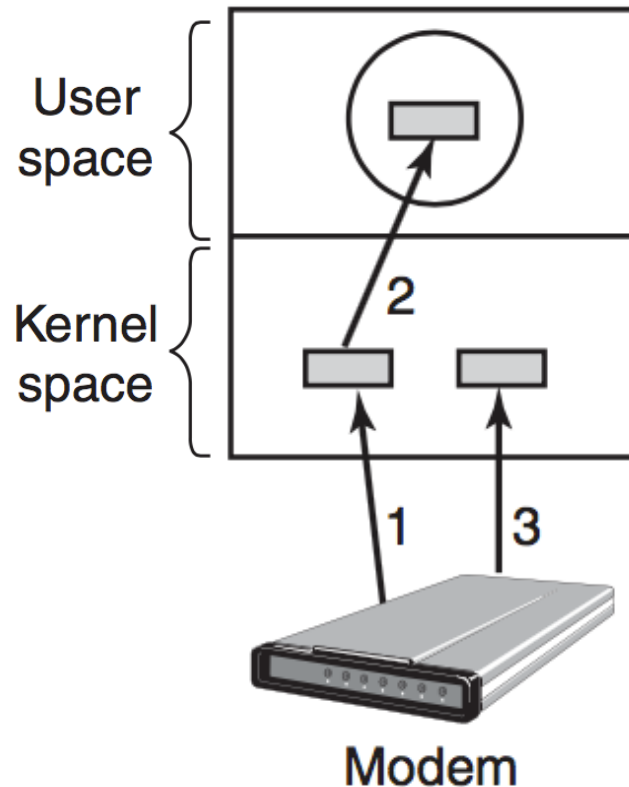


Figure 5-15. (d) Double buffering in the kernel

# Circular Buffer (2)

- Another common form of buffering is the circular buffer. It consists of a region of memory and two pointers
- One pointer points to the next free word, where new data can be placed
- The other pointer points to the first word of data in the buffer that has not been removed yet

# Buffering - Disadvantage (1)

- If data get buffered too many times, performance suffers
- Consider, for example, the network of Fig. 5-16. Here a user does a system call to write to the network:
  - Step 1: the kernel copies the packet to a kernel buffer to allow the user to proceed immediately. At this point the user program can reuse the buffer
  - Step 2: when the driver is called, it copies the packet to the controller for output



# Buffering - Disadvantage (2)

- Step 3: after the packet has been copied to the controller's internal buffer, it is copied out onto the network
- Step 4: next the packet is copied to the receiver's kernel buffer
- Step 5: finally, it is copied to the receiving process' buffer
- It should be clear that all this copying is going to slow down the transmission rate considerably because all the steps must happen sequentially

# Buffering

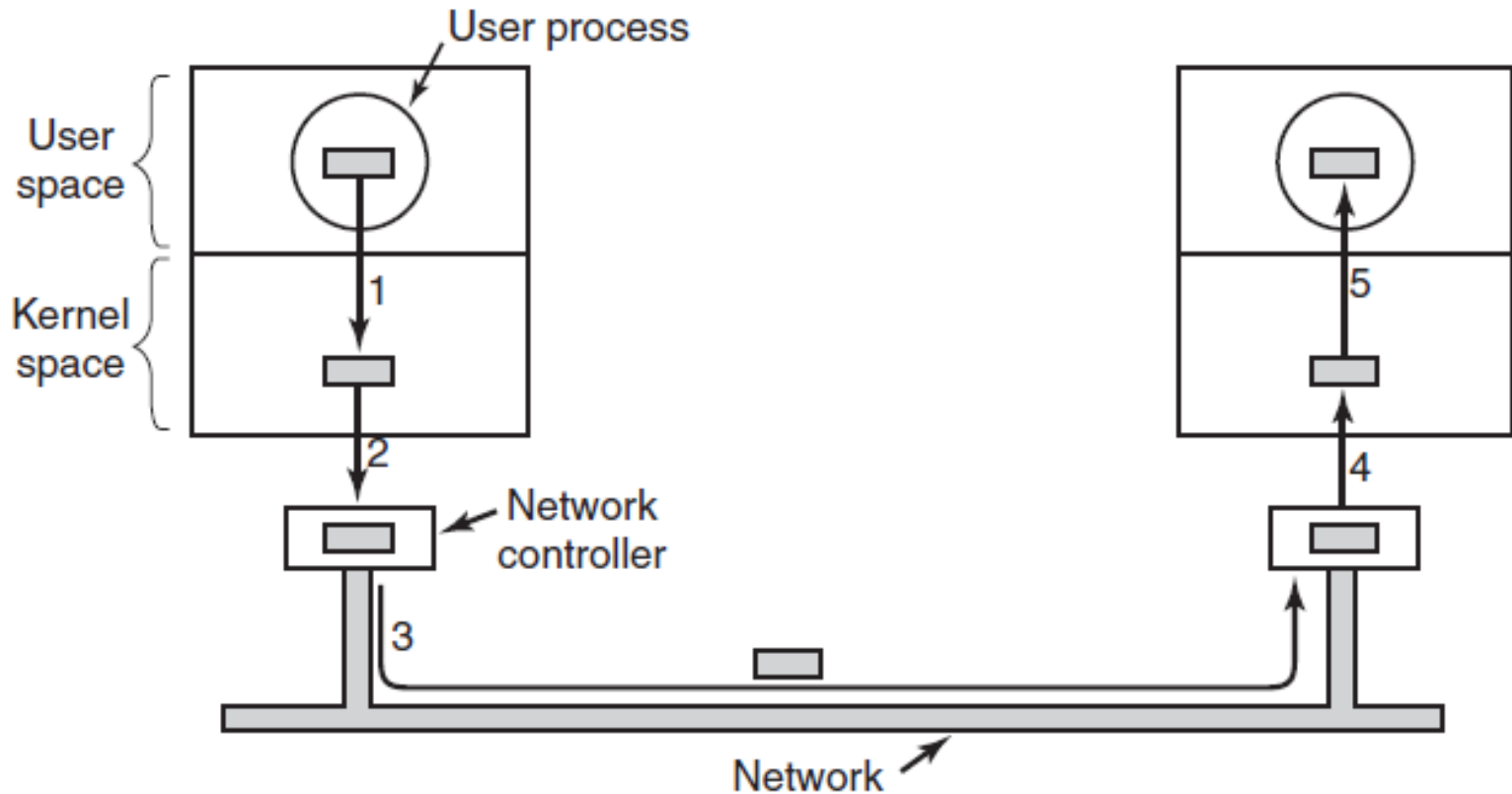


Figure 5-16. Networking may involve many copies of a packet.

# Error Reporting (1)

- One class of I/O errors is programming errors
- These occur when a process asks for something impossible, such as writing to an input device (keyboard, scanner, mouse, etc.) or reading from an output device (printer, plotter, etc.)
- The action to take on these errors is straightforward: just report back an error code to the caller.

# Error Reporting (2)

- Another class of errors is the class of actual I/O errors, for example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off.
- In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software

# Allocating and Releasing Dedicated Devices

- There are special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing
- Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution

# Device-Independent Block Size

- Different disks may have different sector sizes
- It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block
- Similarly, some character devices deliver their data one byte at a time (e.g., mice), while others deliver theirs in larger units (e.g., Ethernet interfaces). These differences may also be hidden

# User-Space I/O Software (1)

- A small portion of I/O software consists of libraries linked together with user programs
- For example, when a C program contains the call **count = write(fd, buffer, nbytes);** the library procedure *write()* might be linked with the program and contained in the binary program present in memory at run time

# User-Space I/O Software (2)

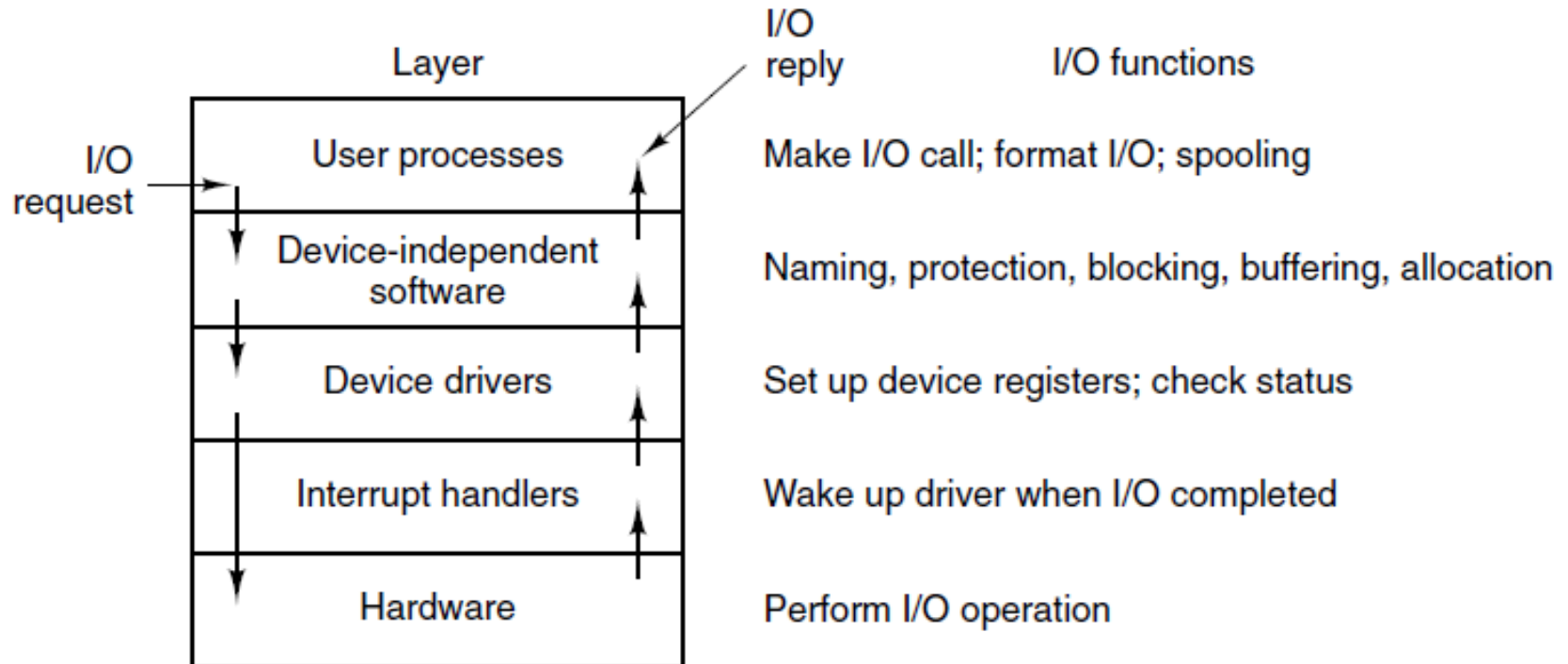


Figure 5-17. Layers of the I/O system and the main functions of each layer



END

Week 12 - Lecture

# References

- Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013, Prentice-Hall, Inc.
- <http://rodrev.com/graphical/programming/Deadlock.swf>
- <http://www.utdallas.edu/~ilyen/animation/cpu/program/prog.html>
- <https://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html>
- <http://inventwithpython.com/blog/2013/04/22/multithreaded-python-tutorial-withthreadworms/>