Modern Operating Systems

Chapter 5 – Input/Output Software

Zhang Yang

Spring 2022

Content of the Lecture

- 5.1 Principles of I/O Hardware
- 5.2 Principles of I/O Software
- 5.3 I/O Software Layers

Principles of I/O Hardware

- I/O Devices
- Device Controllers
- Interfacing with Controllers
- DMA
- Interrupt

I/O Devices (1)

- I/O Devices Types
  - Block Device
    - Information is stored and accessed in fixed-size blocks.
    - Block addressable, not byte addressable.
    - Common block size: 512 bytes-32768 bytes.
    - Hard Disk, blu-ray discs, USB Sticks, etc.
  - Character Device
    - Send or receive streams of characters.
    - NOT byte or block addressable.
    - Printer, Network Interface, Mouse, etc.
  - Classification not perfect.
    - What about
      - Clocks?
      - Memory-mapped screens?

I/O Devices (2)

- Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

I/O Devices (3)

- Typical Device Data Rates

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 device, network, and bus data rates

Device Controllers (1)

- I/O devices have components
  - Mechanical component
  - Electronic component
- Electronic component controls the device
  - Device Controller/Adapter
  - May be able to handle multiple devices. (e.g., daisy chained)
  - May be more than one controller per mechanical component (e.g., hard drive).
  - May implement a standard interface (SCSI/EIDE/USB)
- Controller's Tasks
  - Convert serial bit stream to block of bytes.
  - Perform error correction as necessary.
  - Make available to main memory.

Device Controllers (2)

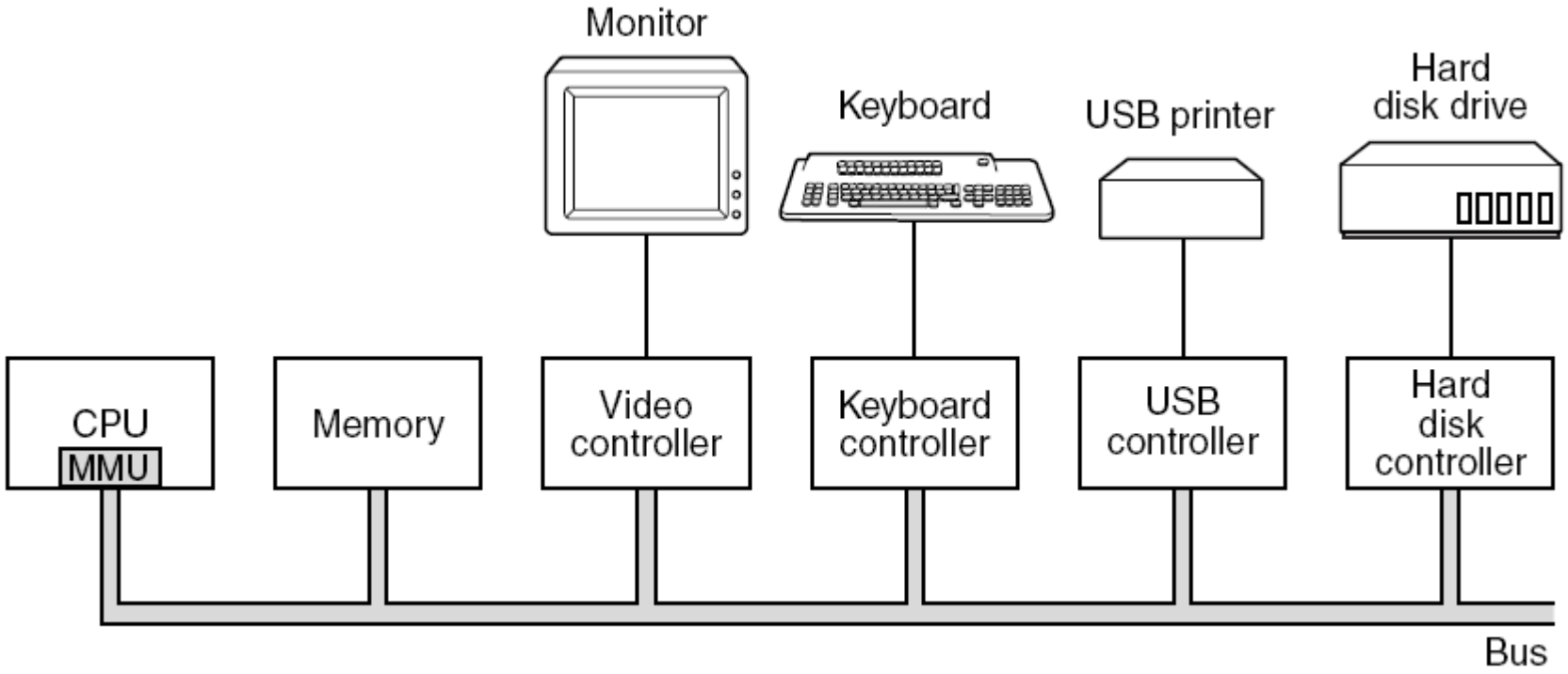


Figure 1-6. Some of the components of a simple personal computer.

Interfacing with Controllers (1)

- The OS interacts with a controller
  - By writing/reading registers (command/status)
  - By writing/reading memory buffers (actual data)
- Port-Mapped I/O (Isolated I/O)
  - Each controller is assigned a special address called a port.
  - The set of all the I/O ports form the I/O port space.
  - Using a special I/O instruction.
    - E.g. IN REG, PORT
  - The address space for memory and I/O are different.

Interfacing with Controllers (2)

- Port-Mapped I/O (ctd.)
  - Advantage of Port-Mapped I/O
    - Useful for CPU with small addressing capability (e.g. 16-bit).
  - Disadvantages of Port-Mapped I/O
    - Need separate I/O instructions.
    - Limited instruction set.

Interfacing with Controllers (3)

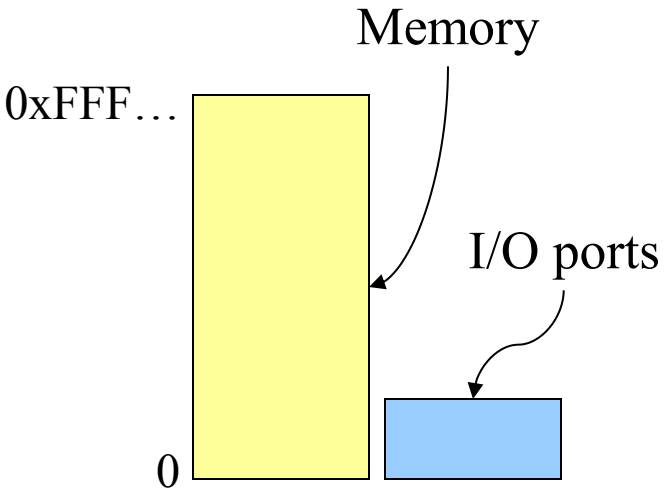
- Port-Mapped I/O (ctd.)
  - Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

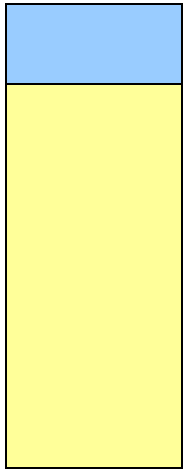
Interfacing With Controllers (4)

- Memory-Mapped I/O
  - Introduced in PDP-11.
  - Controller Registers are part of regular address space.
  - Usually flags in special registers indicate that the memory access is for memory-mapped I/O.
  - Used by Motorola 680x0 processors.
- Hybrid
  - Memory-mapped I/O data buffers and separate I/O ports.
  - E.g. Intel x86
    - 640K – 1M-1 reserved for device data buffer
    - 0 – 64K-1 for I/O ports

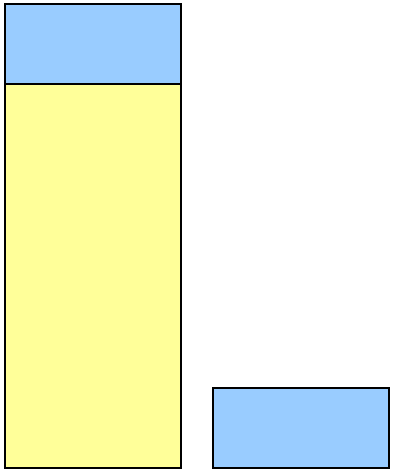
Interfacing With Controllers (5)



Separate I/O & memory space



Memory-mapped I/O



Hybrid: both memory-mapped & separate spaces

Interfacing With Controllers (6)

- When a controller's registers has to be accessed
  - CPU puts address on the bus.
  - CPU sets a line that tells if this address is a memory address or an I/O port.
  - In case the register/buffer is memory-mapped, the corresponding controller is responsible for checking the address and service the request if the address is in its range.

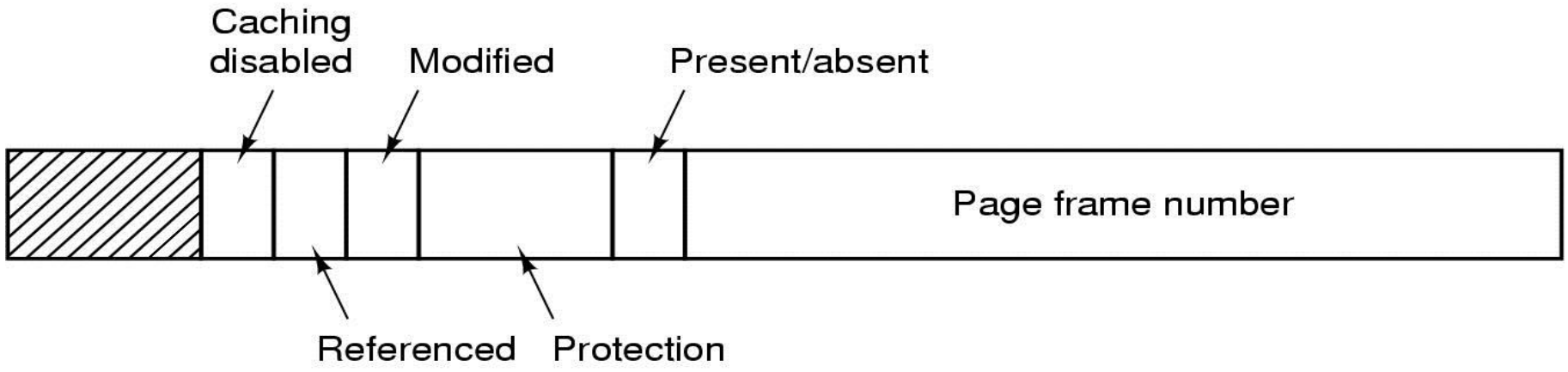
Interfacing With Controllers (7)

- Advantages of Memory-Mapped I/O
  - With memory-mapped I/O, a I/O device driver can be written entirely in C. Without it, some assembly code is needed.
  - With memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O.
  - With memory-mapped I/O, every instruction can reference memory can also reference control registers.
    - LOOP: TEST PORT_4 //check if port 4 is 0
      BEQ READY //if it is 0, go to ready
      BRANCH LOOP //otherwise, continue testing

      READY:

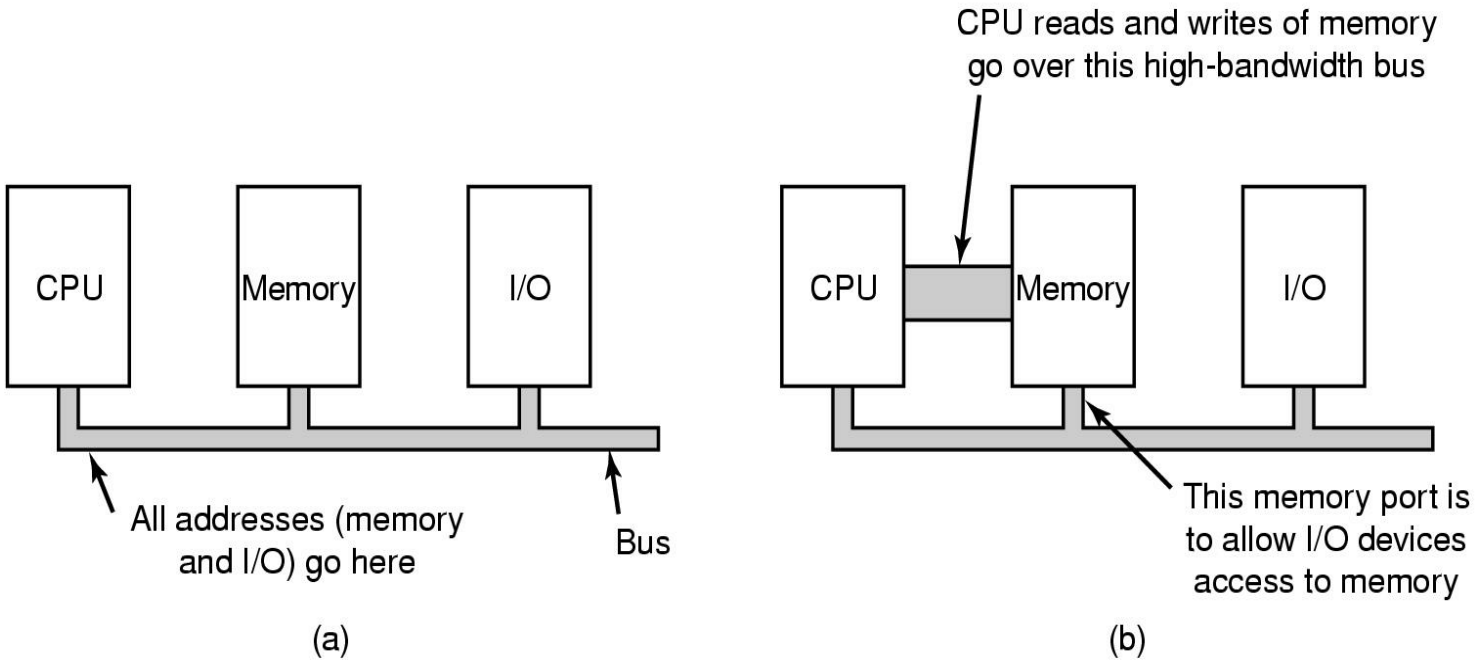
Interfacing With Controllers (8)

- Disadvantages of Memory-Mapped I/O
  - Most computers have capability of caching memory words. Caching a device control register would be disastrous.
    - Solution: The hardware has to be equipped with the ability to selectively disable caching.



Interfacing With Controllers (9)

- Disadvantages of Memory-Mapped I/O (ctd.)
  - In some computer systems, the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding to them.



Interfacing With Controllers (10)

- Disadvantages of Memory-Mapped I/O (ctd.)
  - In some computer systems, the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding to them.

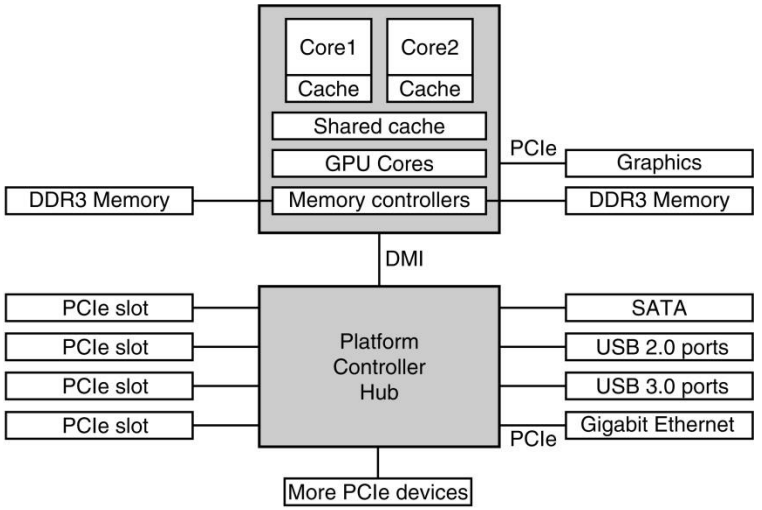


Figure 1-12 A x-86 system structure

Direct Memory Access (1)

- Direct Memory Access (DMA) is a capability provided by some computer bus architectures that allows data to be sent directly from an attached device (such as a disk drive) to the memory on the computer's motherboard.
- Used mainly for block devices.
- DMA Operations:
  - CPU program the DMA controller.
  - DMA requests transfer to memory.
  - Data transferred.
  - The disk controller sends an acknowledgement.

Direct Memory Access (2)

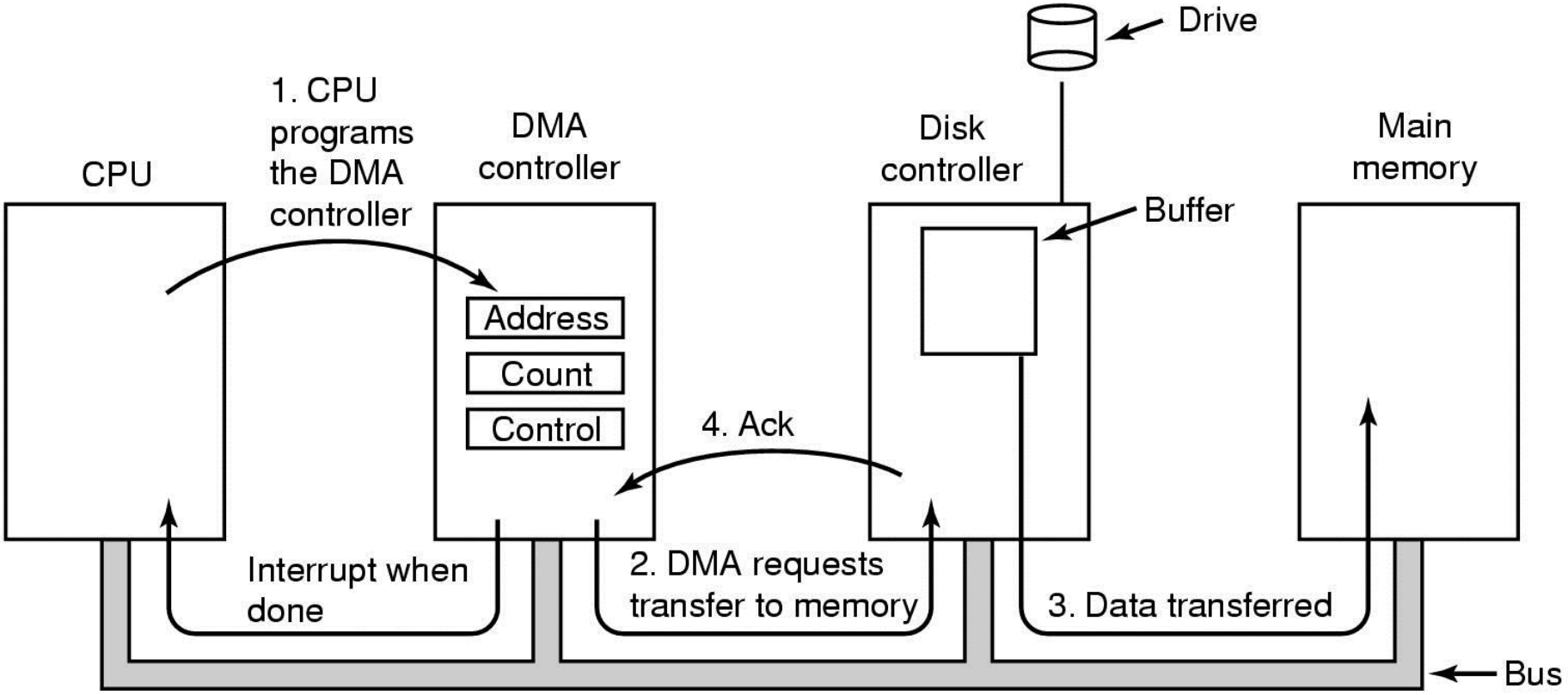
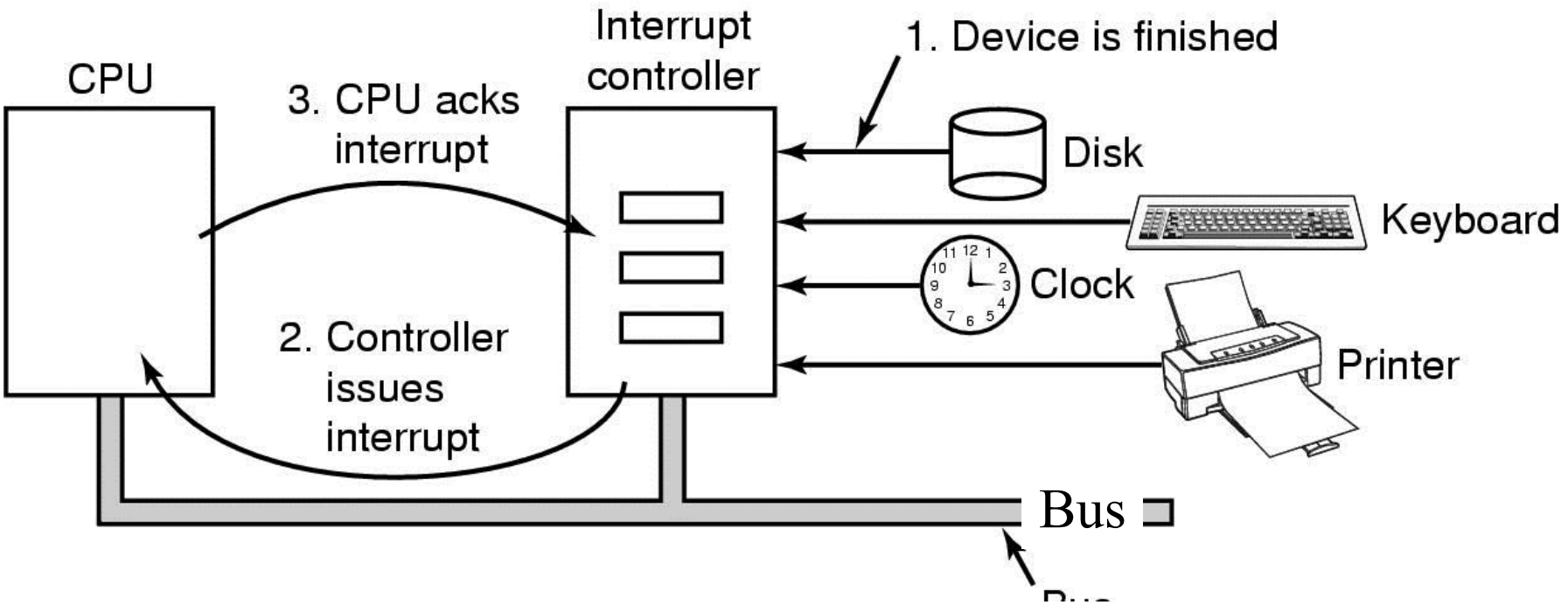


Figure 5-4 Operations of a DMA transfer

Interrupts (1)

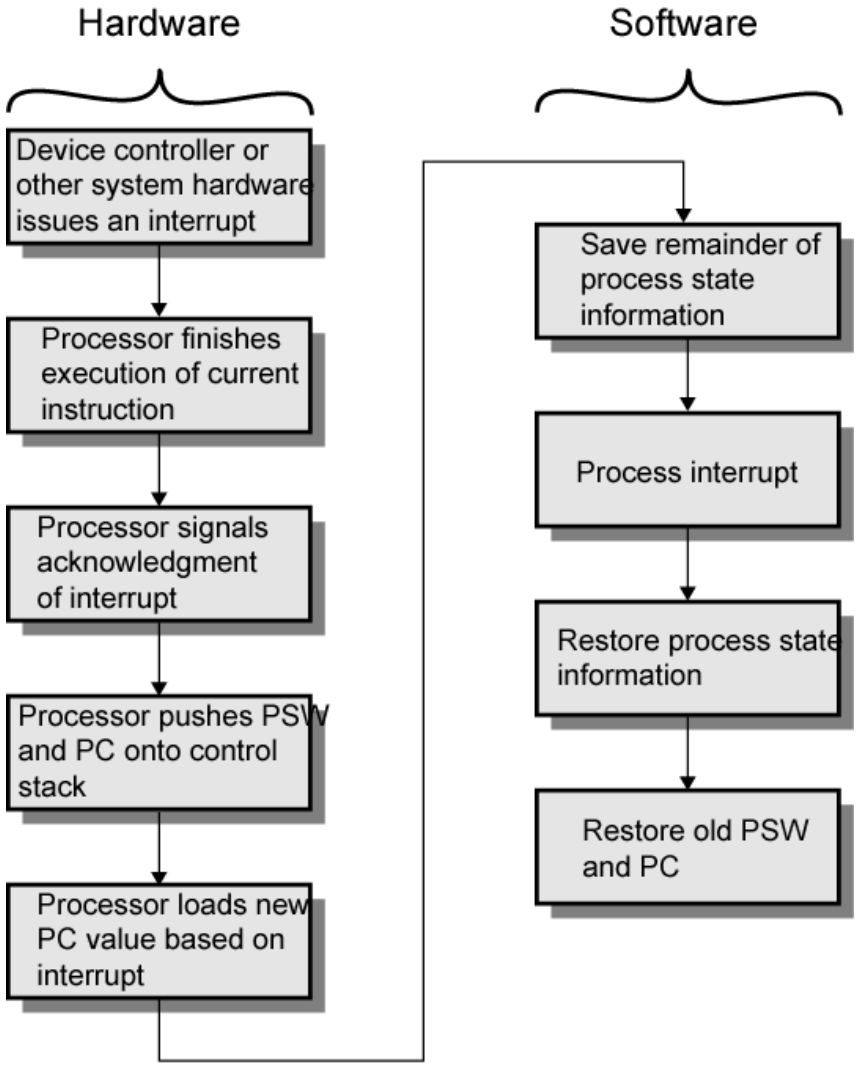
- Hardware's View of Interrupts



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

Interrupts (2)

- Interrupt Processing



Interrupts (3)

- In interrupt processing, restoring CPU state is easier said then done when instructions may end up... half-baked (in case of pipelining).
- Pipeline and superscalar technologies complicate interrupt handling.

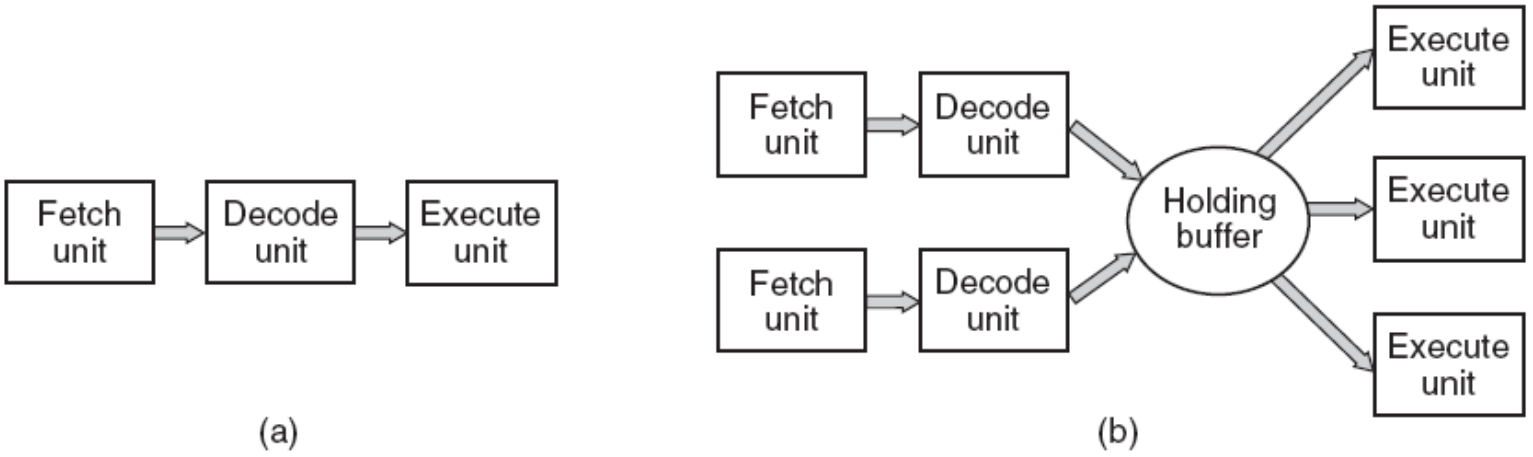
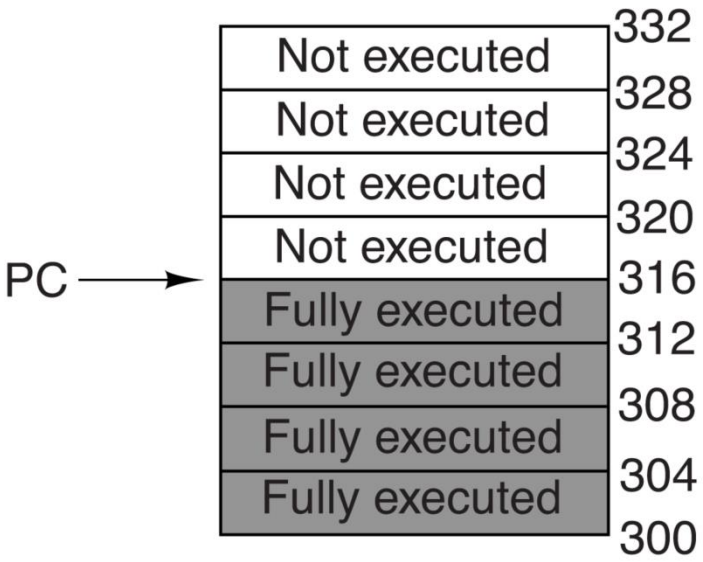


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

Interrupts (4)

- Precise Interrupt
  - An interrupt that leaves the machine in a well-defined state is called a precise interrupt.
  - Precise interrupts preserve the model that instructions execute in program-generated order, one at a time.
    - If an interrupt occurs, the processor can recover from it.

Figure 5-6 (a) A precise interrupt



precise state at interrupt

- all instructions older than interrupt are complete.
- all instructions younger than interrupt haven't started.

Interrupts (5)

- Precise Interrupt (ctd.)
  - Four Properties
    - The PC (Program Counter) is saved in a known place.
    - All instructions before the one pointed to by the PC have fully executed.
    - No instruction beyond the one pointed to by the PC has been executed.
    - The execution state of the instruction pointed to by the PC is known.

Interrupts (6)

- Imprecise Interrupt
  - An interrupt that does not meet these four requirements is called an imprecise interrupt.

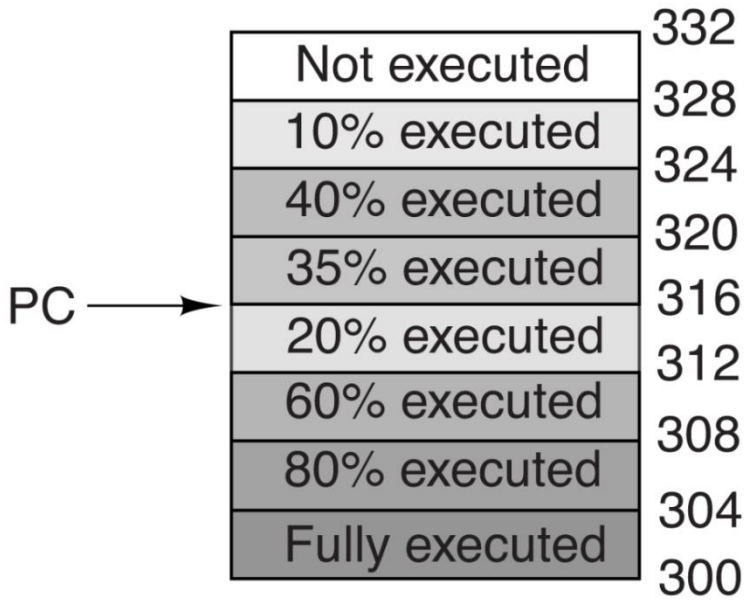


Figure 5-6. (b) An imprecise interrupt.

Interrupts (7)

- Precise Interrupt vs. Imprecise Interrupt
  - Some computers are designed so that some kinds of interrupts and traps are precise and others are not.
  - Some machines have a bit that can be set to force all interrupts to be precise.
  - Some superscalar machines, such as the x86 family, have precise interrupts to allow old software to work correctly.

Principles of I/O Software

- Goals of I/O Software
  - Device Independence
  - Uniform Naming
  - Error Handling
  - Synchronous vs. Asynchronous Transfer
  - Buffering
  - Sharable vs. Dedicated Devices
- I/O Operation
  - Programmed I/O
  - Interrupt-driven I/O
  - I/O Using DMA

Goals of I/O Software (1)

- Device Independence
  - Read and Write from Floppy, Disk, CD-ROM without modifying. (Programs can access any I/O device without specifying device in advance.)
  - I/O software should abstract the device details.
  - E.g. sort <input> output

Goals of I/O Software (2)

- Uniform Naming
  - The way we specify a file or device should be done in the same way.
  - Name of a file or device is a string or an integer not depending on which machine.
    - Different devices of same type have similar name.
  - In UNIX, all I/O is integrated with the file system.

Goals of I/O Software (3)

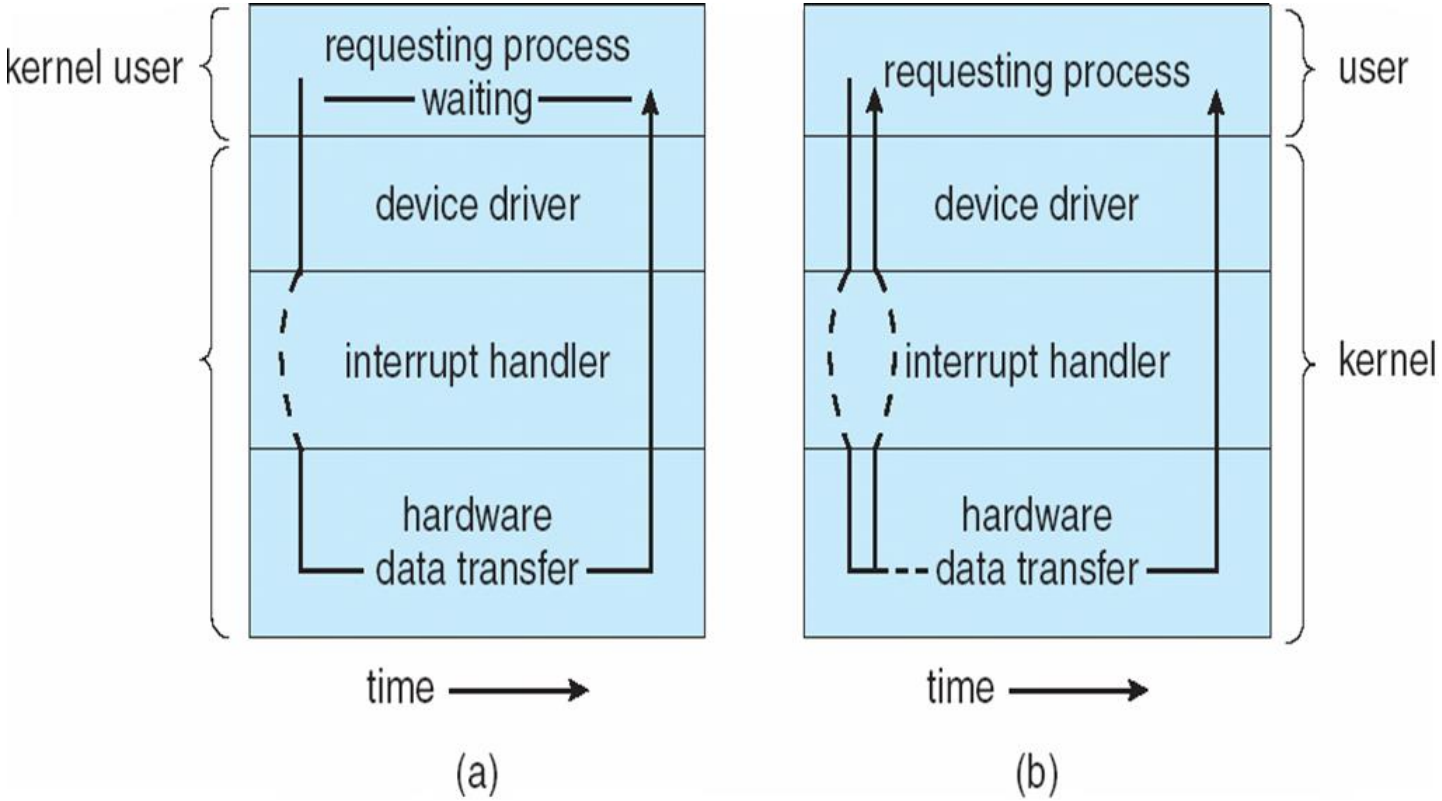
- Error Handling
  - Handle the errors close to the hardware as possible.
  - Propagate errors up only when lower layer cannot handle it.
  - Hide errors as much as possible—many HW errors are transient.

Goals of I/O Software (4)

- Synchronous vs. Asynchronous Transfers
  - Blocking transfers vs. Interrupt-driven
  - Most I/O hardware operates asynchronously.
  - Synchronous read/write at application level.
  - Synchronous (blocking) read/write easier to program.
  - It is up to OS to make I/O operations that are actually interrupt-driven look blocking to the user program.

Goals of I/O Software (5)

- Synchronous vs. Asynchronous Transfers (ctd.)



Blocking/Synchronous I/O **Interrupt-driven/Asynchronous I/O**

Goals of I/O Software (6)

- Buffering
  - Data coming off a device cannot be stored in final destination.
- Sharable vs. Dedicated Devices
  - E.g. Disks are sharable.
  - E.g. Tape drives are dedicated, may introduce a variety of problems, such as deadlock.
  - OS must be able to handle both shared and dedicated devices in a way that avoids problem.
    - Make dedicated devices appear sharable. E.g printers

I/O Software

- System call in user-space.
- Data is copied from user space to kernel space.
- I/O software can operate in several modes.
  - Programmed I/O
    - Polling/Busy waiting for the device.
  - Interrupt-Driven I/O
    - Operation is completed by interrupt routine.
  - DMA-based I/O
    - Set up controller and let it deal with the transfer.

Programmed I/O (1)

- CPU is always busy with I/O until I/O gets completed.
  - Fine for single process systems.
    - MS-DOS
    - Embedded systems.
  - But not a good approached for multi-programming and time-sharing systems.

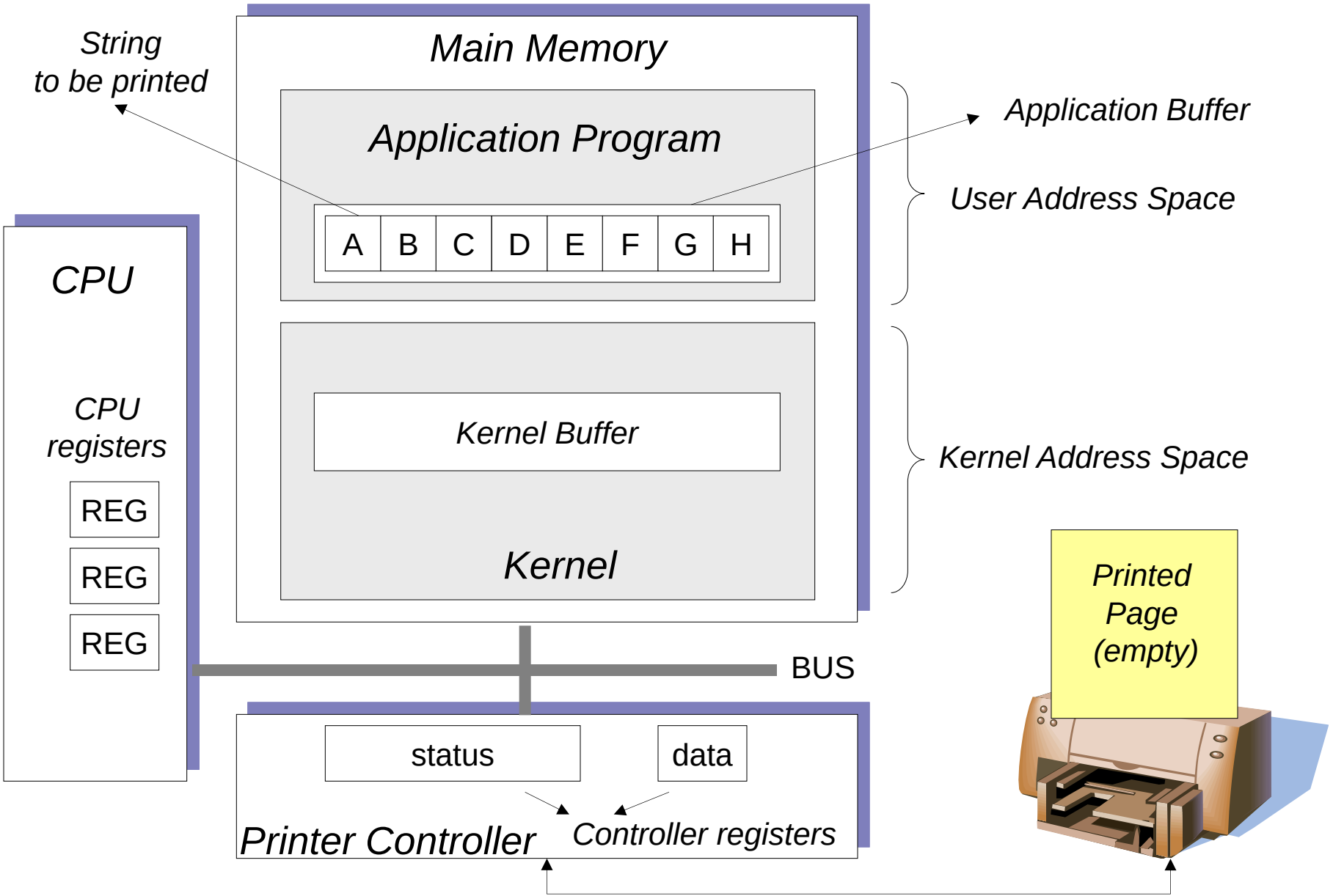
Programmed I/O (2)

- Programmed I/O Operations (output)
  - Data copied to kernel space from user space.
  - OS then enters a tight loop (Polling) the device (usually checking a status register) to see if the device is ready for more data.
  - Once the device is ready to accept more data, the OS will copy the data to the device register.
  - Once all data has been copied control is returned to the user program.

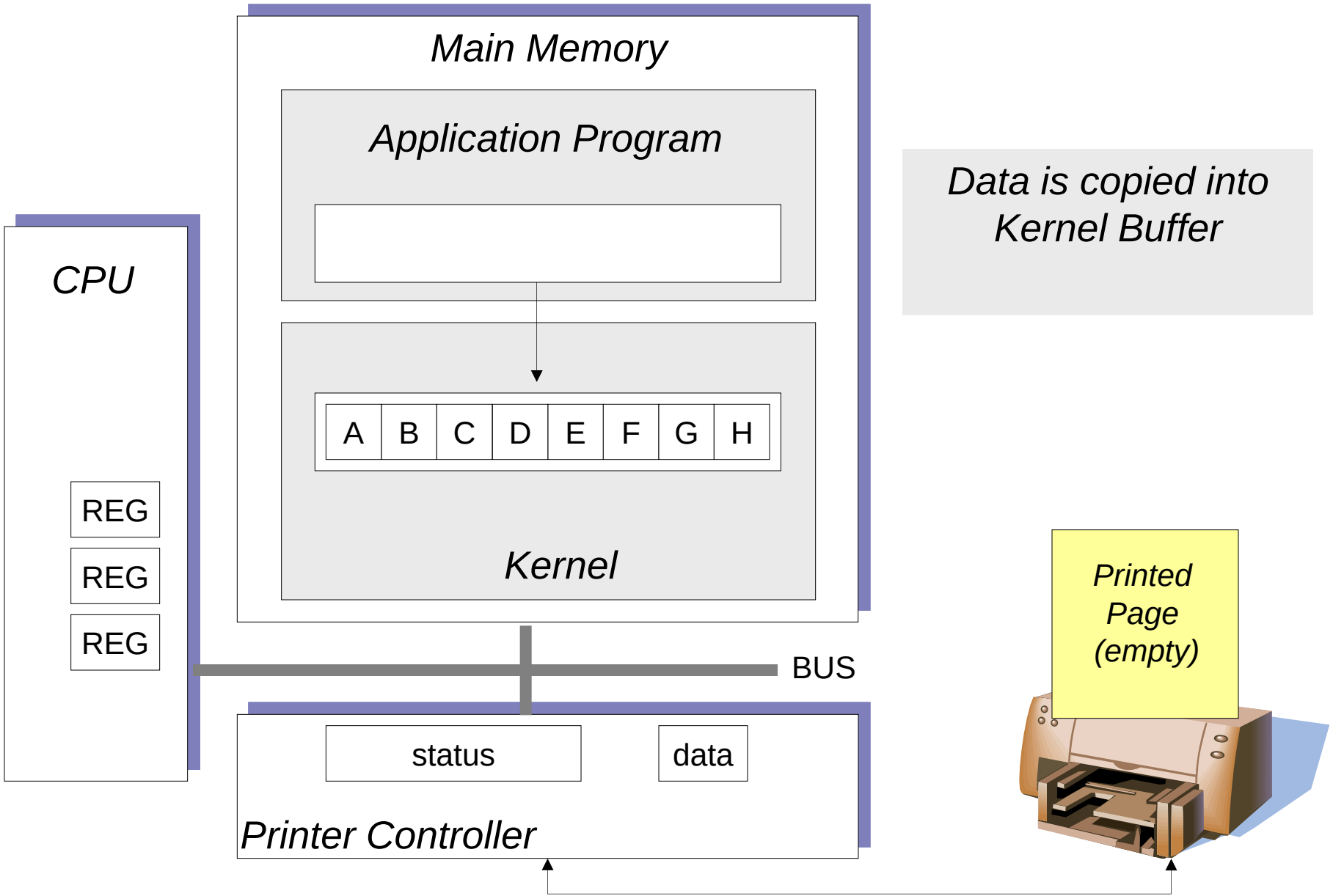
Programmed I/O (3)

- Printing String Example
  - Assume a program that wants to print a string to a printer.
  - Assume the string is “ABCDEFGH”.
    - 8 character long string.
  - Assume printer has 1 byte of data buffer to put the character that needs to be printed.

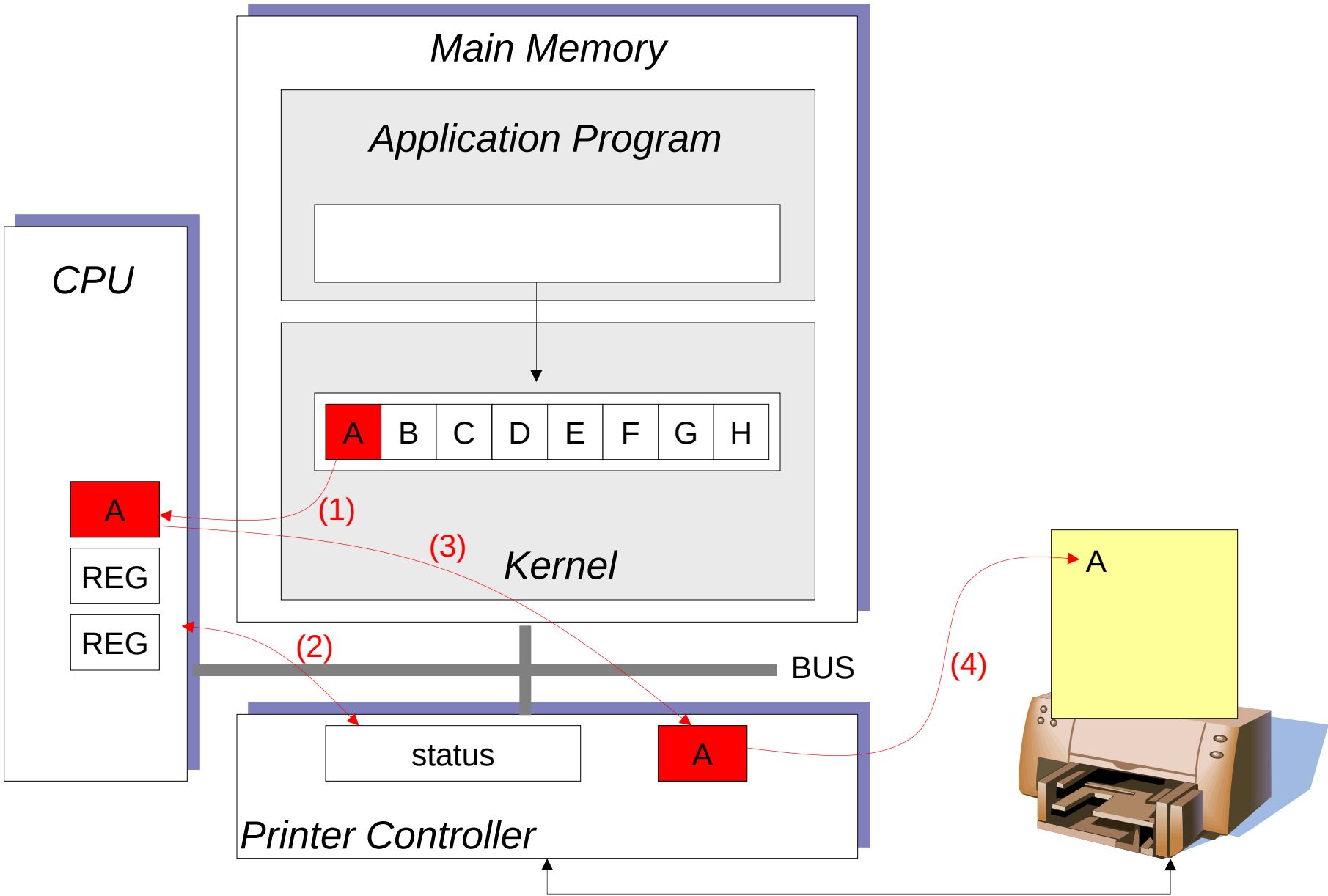
Programmed I/O (4)



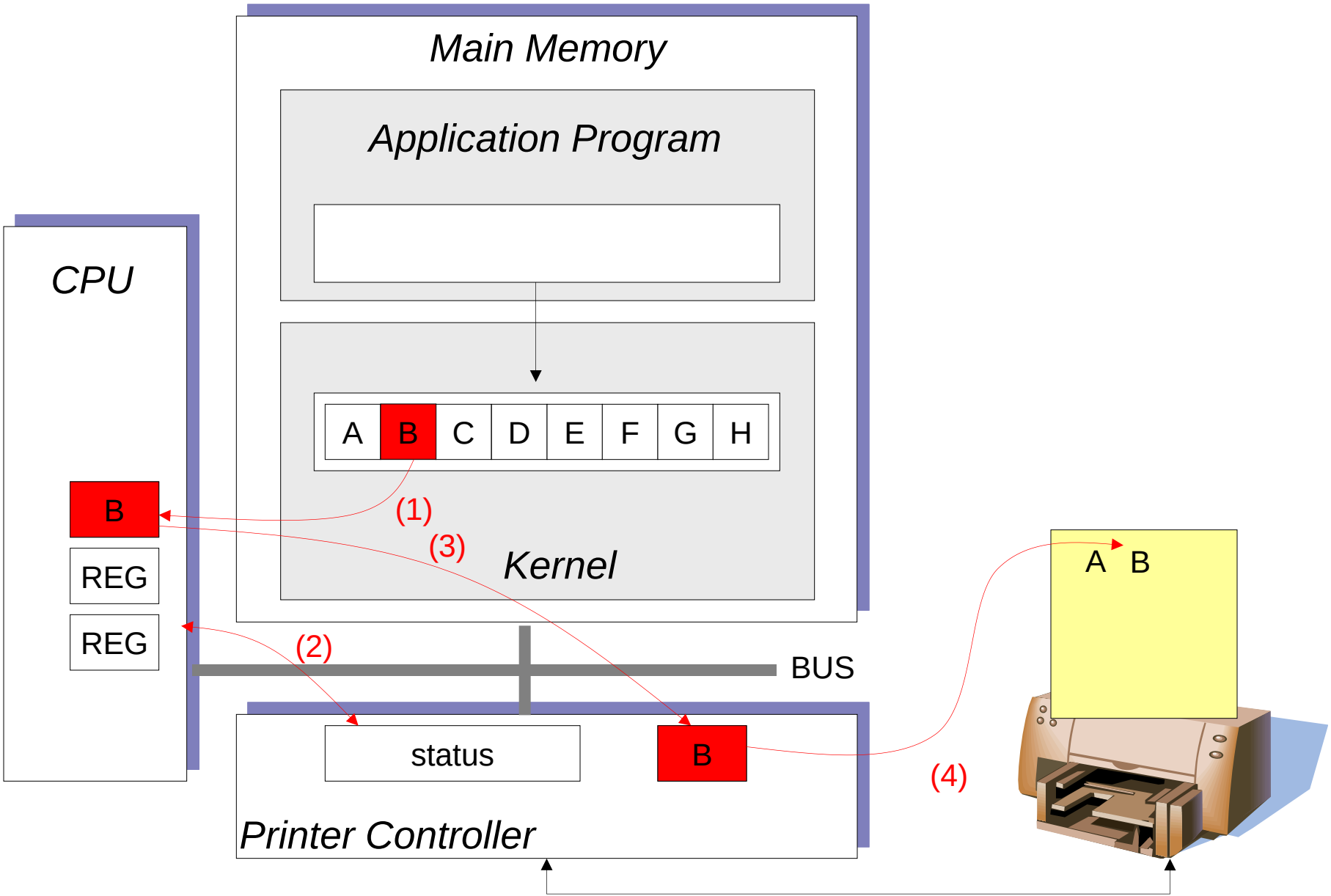
Programmed I/O (5)



Programmed I/O (6)



Programmed I/O (7)



Programmed I/O (8)

- Printing String Example (ctd.)
  - How would we implement copy in kernel?

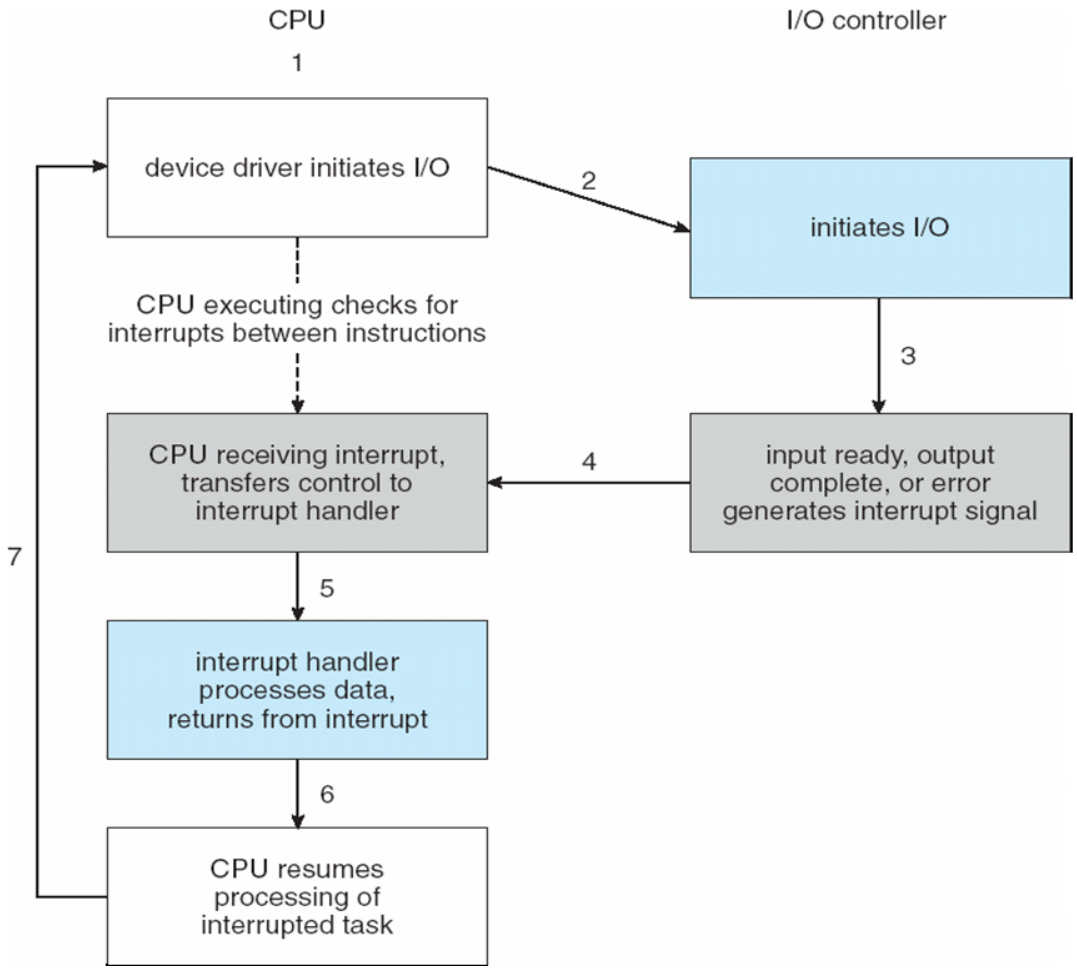
```
/*
    buffer is user buffer
    p is kernel buffer
    count is number of bytes to be copied
*/
copy_from_user(buffer, p, count);
for (i=0; i < count; ++i) /* loop in every character */
        {
            while (*printer_status_register != READY)
            {};   /*busy waiting – loop until ready*/
            *printer_data_register = p[i];  /* output one
                                                character */
        }
return_to_user();
```

Programmed I/O (9)

- What is good about programmed I/O?
  - Usually simple to implement.
- What is bad about programmed I/O?
  - Since we are tying up the CPU checking to see if we can access a device, we are in a state of "busy waiting". No "real" work is being done.
  - Assume printer prints at a rate 100 chars/second
    - Each character takes 10ms to print.
    - During this 10 ms, CPU will be busy with checking the status register of printer (controller).
    - This is waste of CPU.
    - During 10ms period something else can be made (Another process can be run).

Interrupt Driven I/O (1)

- Interrupt Driven I/O Cycle



Interrupt Driven I/O (2)

- Printing String Example
  - The user program makes a system call telling the OS to print the string.
  - After copying application buffer content into kernel buffer, and sending 1 character to printer:
    - CPU does not wait until printer is READY again.
    - Instead the scheduler() is called and some other process is run.
    - The current process is put into *blocked* state until the entire string is printed.
    - Printer controller generates an hardware interrupt when the printer has printed the character and is ready again.

Interrupt Driven I/O (3)

- Printing String Example (ctd.)
  - Code executed in kernel when the print() system call is made by the application process.

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY)
{
};  /* loop until printer controller becomes READY */
*printer_data_register = p[0]; // send to the first character
                                                    to the printer
controller */
scheduler(); /* do some other task */
```

Interrupt Driven I/O (4)

- Printing String Example (ctd.)
  - Interrupt Service Routine that is executed when the printer controller becomes READY for one more byte again.

```
If (count == 0) {      /* we are finishes printing the string */
        unblock_user(); /* unblock the user process that
                                wanted to print the string */
} else {
        /* copy one more character to the controller buffer */
        *print_data_register = p[i];
        count = count – 1;
        i = i + 1;
}
acknowledge interrupt();
return_from_interrupt(); /* finished serving the interrupt */
```

Interrupt Driven I/O (5)

- Advantages of Interrupt-Driven I/O
  - Its faster than Programmed I/O.
  - Efficient too.
- Disadvantages of Interrupt-Driven I/O
  - It can be tricky to write if using a low level language.
  - Interrupt occurs on every character.

I/O Using DMA (1)

- Disadvantage of Interrupt Driven I/O is that interrupt occurs on every character.
- In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU.
- Printing String Example
  - DMA controller will feed the characters from kernel buffer to the printer controller.
    - CPU is not bothered during this transfer.
  - After transfer of whole buffer (string) is completed, then the CPU is interrupted.

I/O Using DMA (2)

- Printing String Example (ctd.)
  - *Code executed in Kernel when the print() system call is made by the application process.*

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

  - Interrupt Service Routine that is executed when a ‘transfer completed interrupt’ is received from DMA.

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

I/O Using DMA (3)

- What is good about I/O using DMA?
  - Reducing the number of interrupts from one per character to one per buffer printed.
- What is bad about I/O using DMA?
  - The DMA controller is usually much slower than the main CPU.

I/O Software Layers

- Overview of the I/O Software
- Interrupt Handlers
- Device Drivers
- Device Independent I/O Software
- User Space I/O Software

Overview of the I/O Software (1)

- The I/O component of the OS is organized in layers.
  - Interrupt Handlers
  - Device Drivers
  - Device-independent I/O
  - User-level I/O System Calls
- Abstraction, encapsulation and layering
  - Any complex software engineering problem.
  - Layers can be modified independently without affecting layers above and below.

Overview of the I/O Software (2)

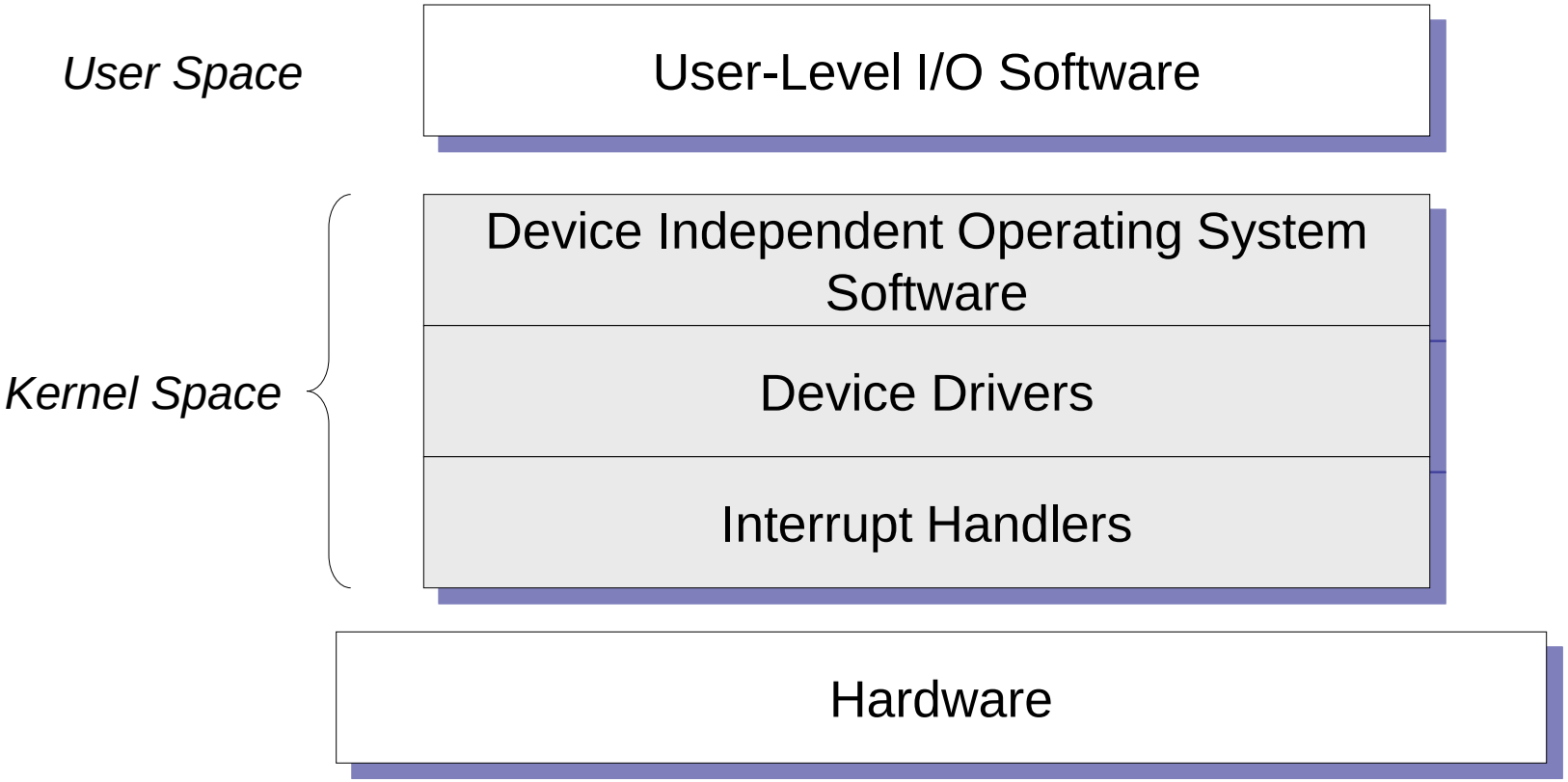


Figure 5-11 Layers of the I/O Software System

Interrupt Handlers (1)

- Interrupt handlers are best hidden.
  - So as little of OS know about it as possible.
- Best way to hide is to have the driver starting an I/O operation block until interrupt notifies of completion.
  - A *down* on a semaphore
  - A *wait* on a condition variable
  - A *receive* on a message
  - ...
- Interrupt handler does the actual work and then unblocks driver that started it.
- Mechanism works best if device drivers are threads in the kernel.

Interrupt Handlers (2)

- Interrupt Handler Routines
  - Interrupts (asynchronous, external to process) basically use the same mechanism as exceptions and traps (synchronous, internal to process).
  - When an interrupts happen, the CPU saves a small amount of state and jumps to an interrupt-handler routine at a fixed address in memory.
  - The interrupt routine’s location is determined by an interrupt vector.

Interrupt Handlers (3)

- Interrupt Handler Routines (ctd.)
  - Intel Pentium processor event-vector table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19Ð31	(Intel reserved, do not use)
32Ð255	maskable interrupts

Non-maskable, used for various error conditions

Maskable, used for device-generated interrupts

Interrupt Handlers (4)

- Typical steps followed by an interrupt routine
  1. Save regs not already saved by interrupt hardware.
  2. Set up context for interrupt service procedure: TLB, MMU, and page table.
  3. Set up stack for interrupt service procedure.
  4. Ack interrupt controller, reenable interrupts.
  5. Copy registers from where saved to the process table.
  6. Run service procedure.
  7. Choose which process to run next.
  8. Set up MMU context for process to run next.
  9. Load new process' registers.
  10. Start running the new process.
  - Interrupt processing is a complex operation that takes a great number of CPU cycles, especially with virtual memory.

Device Drivers (1)

- A device driver is a specific module that manages the interaction between the device controller and the OS.
  - device independent request -> device driver -> device dependent request
- Generally written by device manufacturers.
  - Not necessarily by OS writers.
- A driver can be written for more than one OS.
- Each driver can handle one type of device or one class (e.g. SCSI)

Device Drivers (2)

- Driver usually part of OS kernel
  - Unix – kernel compiled with device drivers.
  - Linux – uses loadable modules which are chunks of code loaded into kernel while system is running.
  - Windows – dynamically loaded into system at startup and plug-n-play.

Device Drivers (3)

- A driver has several functions:
  - Accept abstract read/write requests from the device-independent software above and translate them into concrete I/O-module-specific commands.
  - Schedule requests: Optimize queued request order for best device utilization (eg: disk arm).
  - Initialize the device, if needed.
  - Manage data transfer.
  - Maintain the integrity of driver and kernel data structures.
  - Manage power requirements and log device events.

Device Drivers (4)

- Typical code organization of a device driver
  - Check validity of input parameters coming from above.
    - If valid, translate to concrete commands, e.g., convert block number to head, track & sector in a disk's geometry.
  - Check if device currently in use; if yes, queue request; if not, possibly switch device on, warm up, initialize, etc.
  - Issue appropriate sequence of commands to controller.
  - If needs to wait, block.
  - Upon interrupted from blocking, check for errors and pass data back.
  - Process next queued request.

Device Drivers (5)

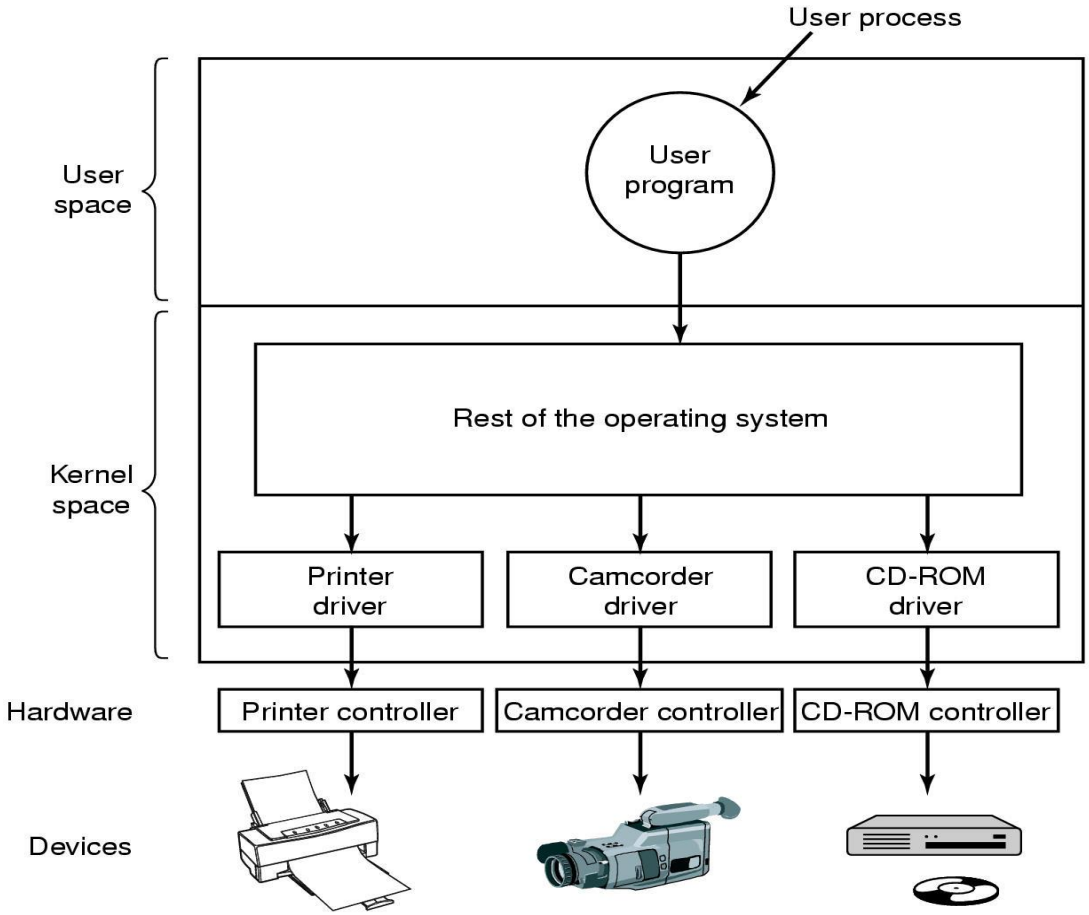


Figure 5-12 Logical position of device drivers is shown here

Communications between drivers and device controllers goes over the bus

Device Drivers (6)

- Operating system and driver communication
  - Commands and data between OS and device drivers.
- Driver and hardware communication
  - Commands and data between driver and hardware.
- Device Driver Types
  - Block: Fixed sized block data transfer.
  - Character: Variable sized data transfer.
  - Terminal: Character driver with terminal control.
  - Network: Streams for networking.
- Most OS define a standard interfaces for device driver. The interface consists of a number of procedures that the rest of OS can call.

Device Drivers (7)

- Character Device Interface
  - read(deviceNumber, bufferAddr, size)
    - Reads “size” bytes from a byte stream device to “bufferAddr”.
  - write(deviceNumber, bufferAddr, size)
    - Write “size” bytes from “bufferAddr” to a byte stream device.
- Block Device Interface
  - read(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “deviceAddr” to “bufferAddr”.
  - write(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “bufferAddr” to “deviceAddr”.
  - seek(deviceNumber, deviceAddress)
    - Move the head to the correct position.
    - Usually not necessary.

Device Drivers (8)

- Some Other Issues
  - A driver code must be reentrant as they can be called by another process while a process is already blocked in the driver.
    - Re-entrant: Code that can be executed by more than one thread (or CPU) at the same time.
  - A driver code needs to handle pluggable devices.
  - Drivers are not allowed to make system calls, but they may call kernel procedures.
    - E.g. Manage MMU, timers, DMA controller, interrupt controller, etc.

Device-Independent I/O Software

- Boundary between device driver and device-independent software varies between systems and devices.
- Device-Independent I/O Software Functions
  - Uniform interfacing to device derivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - Providing a device-independent block size

Uniform Interfacing (1)

- The interface between the device driver and the rest of the OS.

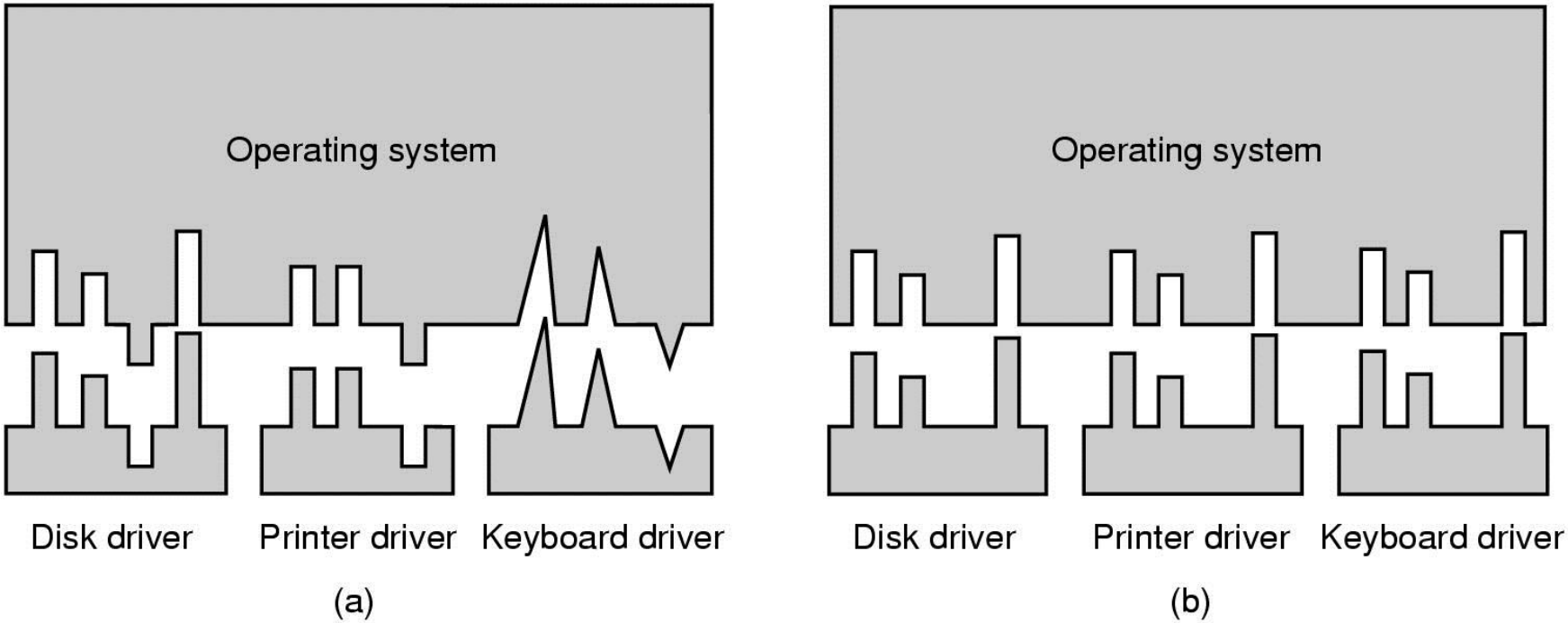


Figure 5-14 (a) Without a standard driver interface (b) With a standard driver interface

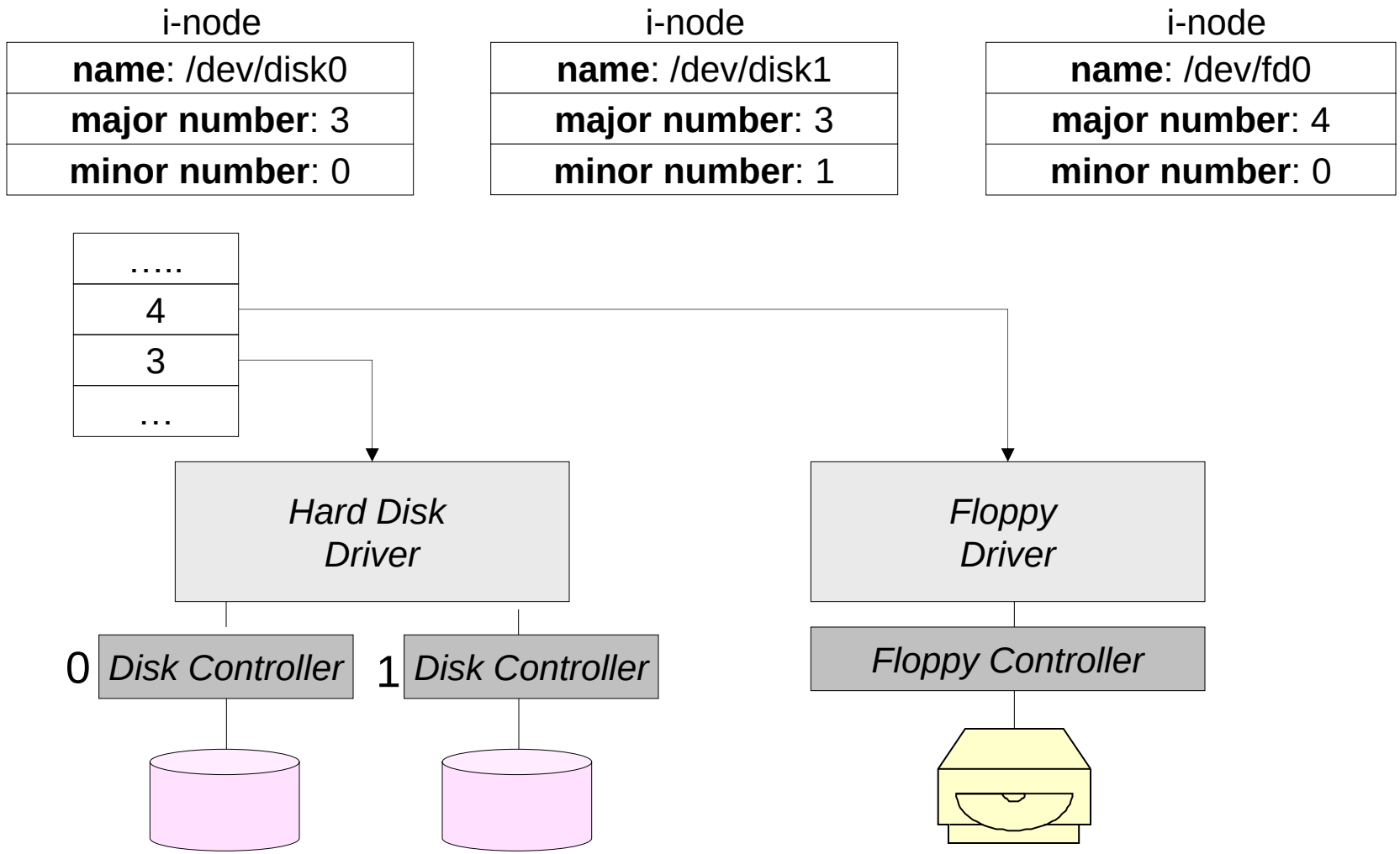
Uniform Interfacing (2)

- Uniform device interface for kernel code
  - Allows different devices to be used the same way.
    - No need to rewrite file-system to switch between SCSI, IDE or RAM disk.
  - Allows internal changes to device driver with fear of breaking kernel code.
- Uniform kernel interface for device code
  - Drivers use a defined interface to kernel services (e.g. kmalloc, install IRQ handler, etc.)
  - Allows kernel to evolve without breaking existing drivers.
- Together both uniform interfaces avoid a lot of programming implementing new interfaces.

Uniform Interfacing (3)

- How I/O devices are named?
  - Device-independent software maps symbolic device names onto the proper driver.
  - In Unix, devices are modeled as special files.
    - They are accessed through the use of system calls such as open(), read(), write(), close(), ioctl(), etc.
    - A file name is associated with each device.
    - Major device number locates the appropriate driver.
    - Minor device number (stored in i-node) is passed as a parameter to the driver in order to specify the unit to be read or written.
- The usual protection rules for files also apply to I/O devices.

Uniform Interfacing (4)



Mapping symbolic I/O device names to their appropriate drivers

Buffering (1)

- Memory area that stores data in kernel space while transferred between device and application.
- Cope with a speed mismatch between producer and consumer.
  - E.g. Modem thousand times slower than disk.
- Adapt between services with different data-transfer sizes.
  - E.g. Fragmentation and reassembly of network packets.

Buffering (2)

- Example: consider a process that wants to read data from a modem.

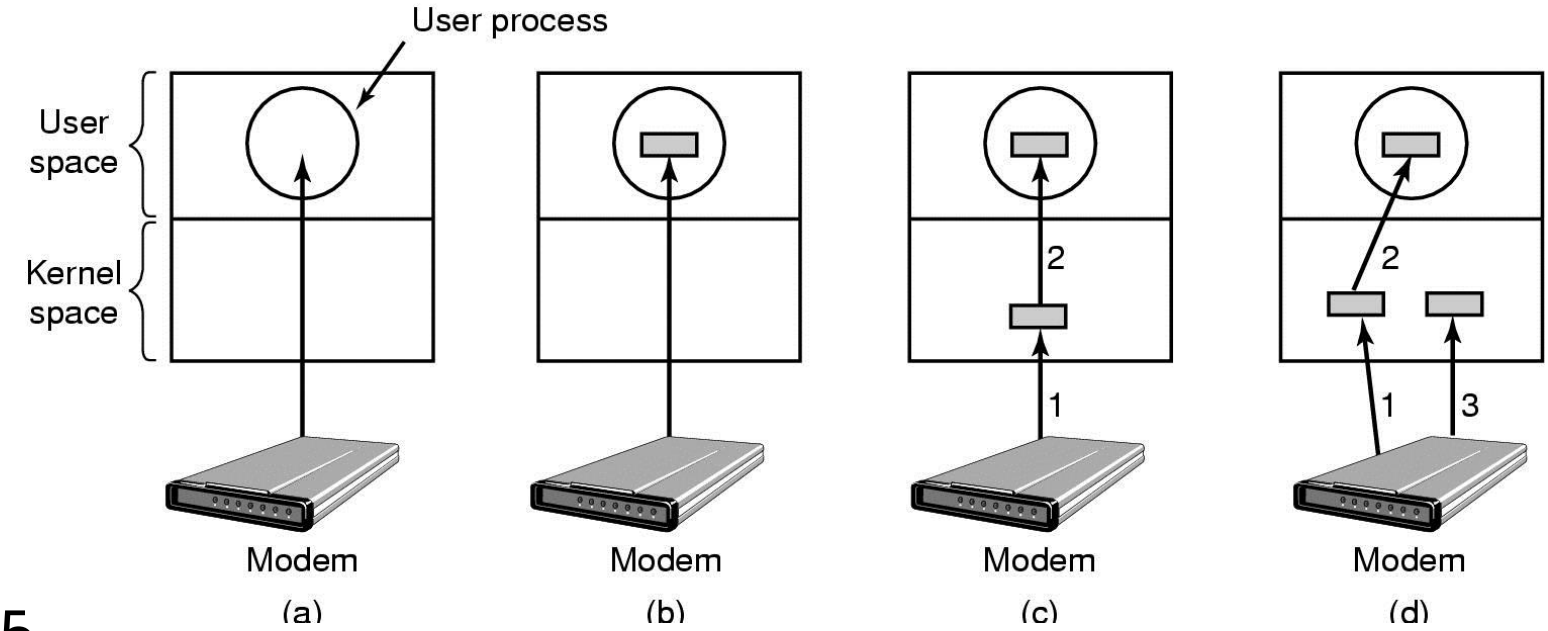


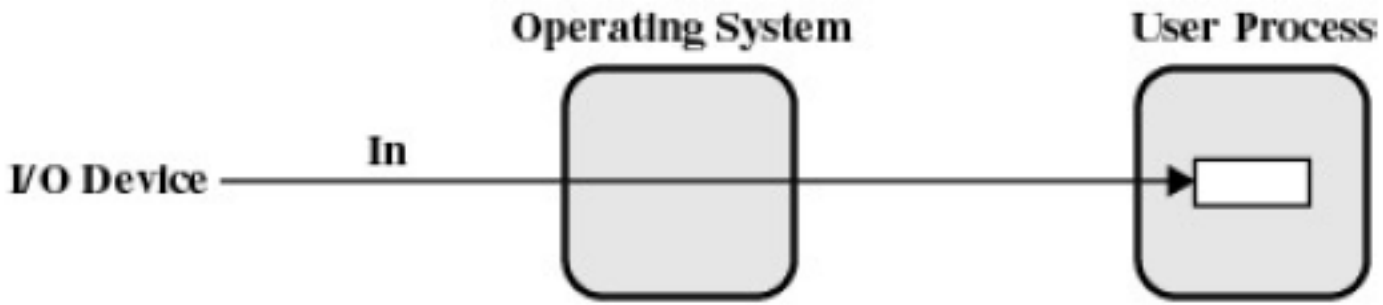
Figure 5-15
(a) Unbuffered input
(b) Buffering in user space
(c) Buffering in the kernel followed by copying to user space
(d) Double buffering in the kernel

Buffering (3)

- No Buffering
  - Process must read/write a device a byte/word at a time.
    - Each individual system call adds significant overhead.
    - Process must wait until each I/O is complete.
      - Blocking/interrupt/waking adds to overhead.
      - Many short runs of a process is inefficient (poor CPU cache temporal locality).

Buffering (4)

- User-level Buffering
  - Process specifies a memory buffer that incoming data is placed in until it fills.
    - Filling can be done by interrupt service routine.
    - Only a single system call, and block/wakeup per data buffer.
      - Much more efficient

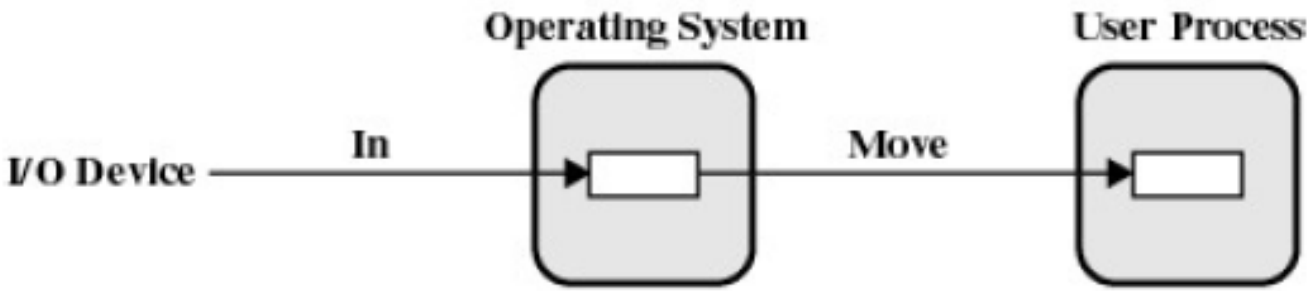


Buffering (5)

- User-level Buffering (ctd.)
  - Issue
    - What happens if buffer is paged out to disk?
      - Could lose data while buffer is paged in.
      - Could lock buffer in memory (needed for DMA), however many processes doing I/O reduce RAM available for paging. Can cause deadlock as RAM is limited resource.

Buffering (6)

- Single Buffer
  - Operating system assigns a buffer in main memory for an I/O request.
  - Stream-oriented
    - Used a line at time.
    - User input from a terminal is one line at a time with carriage return signaling the end of the line.
    - Output to the terminal is one line at a time.



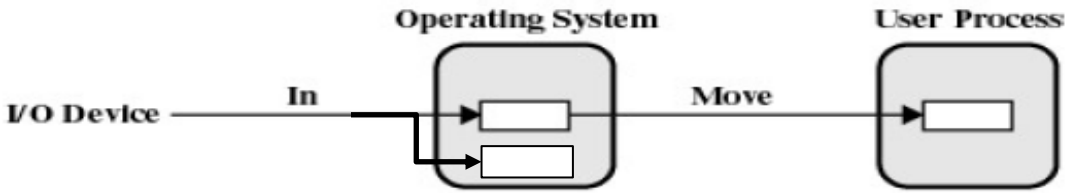
Buffering (7)

- Single Buffer
  - Block-oriented
    - Input transfers made to buffer.
    - Block moved to user space when needed.
    - Another block is moved into the buffer.
      - Read ahead
    - User process can process one block of data while next block is read in.
    - Swapping can occur since input is taking place in system memory, not user memory.
    - Operating system keeps track of assignment of system buffers to user processes.

Buffering (8)

- Single Buffer (ctd.)
  - What happens if kernel buffer is full, the user buffer is swapped out, and more data is received???
    - We start to lose characters or drop network packets.

Buffering (9)



- Double Buffer
  - Use two system buffers instead of one.
  - A process can transfer data to or from one buffer while the operating system empties or fills the other buffer.
  - May be insufficient for really bursty traffic.
    - Lots of application writes between long periods of computation.
    - Long periods of application computation while receiving data.
    - Might want to read-ahead more than a single block for disk.

Buffering (10)

- Circular Buffer
  - More than two buffers are used.
  - Each individual buffer is one unit in a circular buffer.
  - Used when I/O operation must keep up with process.

Buffering (11)

- Notice that buffering, double buffering, and circular buffering are all Bounded Buffer Problem (Producer-Consumer Problem).
- Buffering in Fast Networks
  - Networking may involve many copies.
  - Copying reduces performance.
    - Especially if copy costs are similar to or greater than computation or transfer costs.
  - Super-fast networks put significant effort into achieving zero-copy.

Buffering (12)

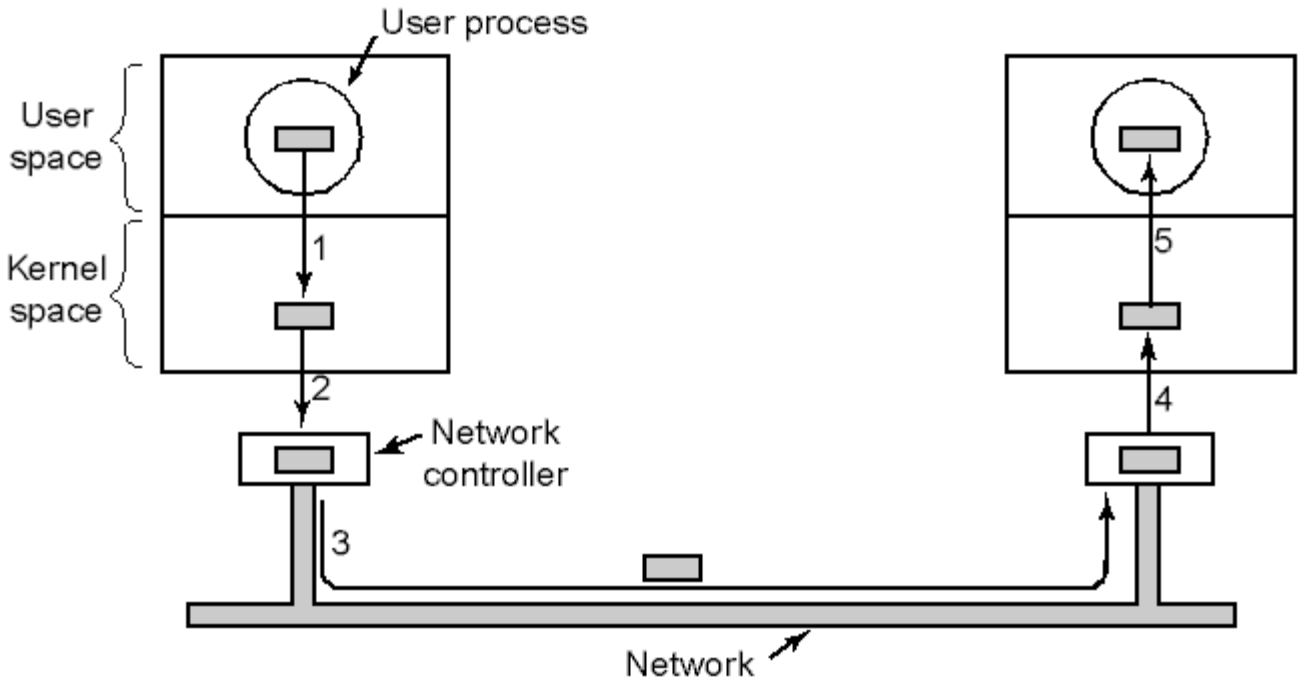


Figure 5-16 Networking may involve many copies

Error Reporting

- Some errors are handled by device controllers.
  - E.g., Checksum incorrect, re-correct the block by using redundant bits.
- Some by device derivers.
  - E.g., disk block could not be read, re-issue the read operation for block from disk.
- Some by the device-independent software layer of OS.
  - Retry I/O or report to user with error code.
  - Identifies programming error vs. I/O error
  - Programming errors
    - E.g., Attempt to write to a read-only device.
  - Actual I/O errors
    - E.g., The camcorder is shut-off, therefore we could not read. Return an indication of this error to the calling application.

Allocating& Releasing Dedicated Devices

- Do not allow concurrent accesses to dedicated devices such as CD-RWs.
- Require processes to perform open()'s on the special files for devices directly.
  - The process retries if open() fails.
- Have special mechanisms for requesting and releasing dedicated devices.
  - An attempt to acquire a device that is not available blocks the caller.

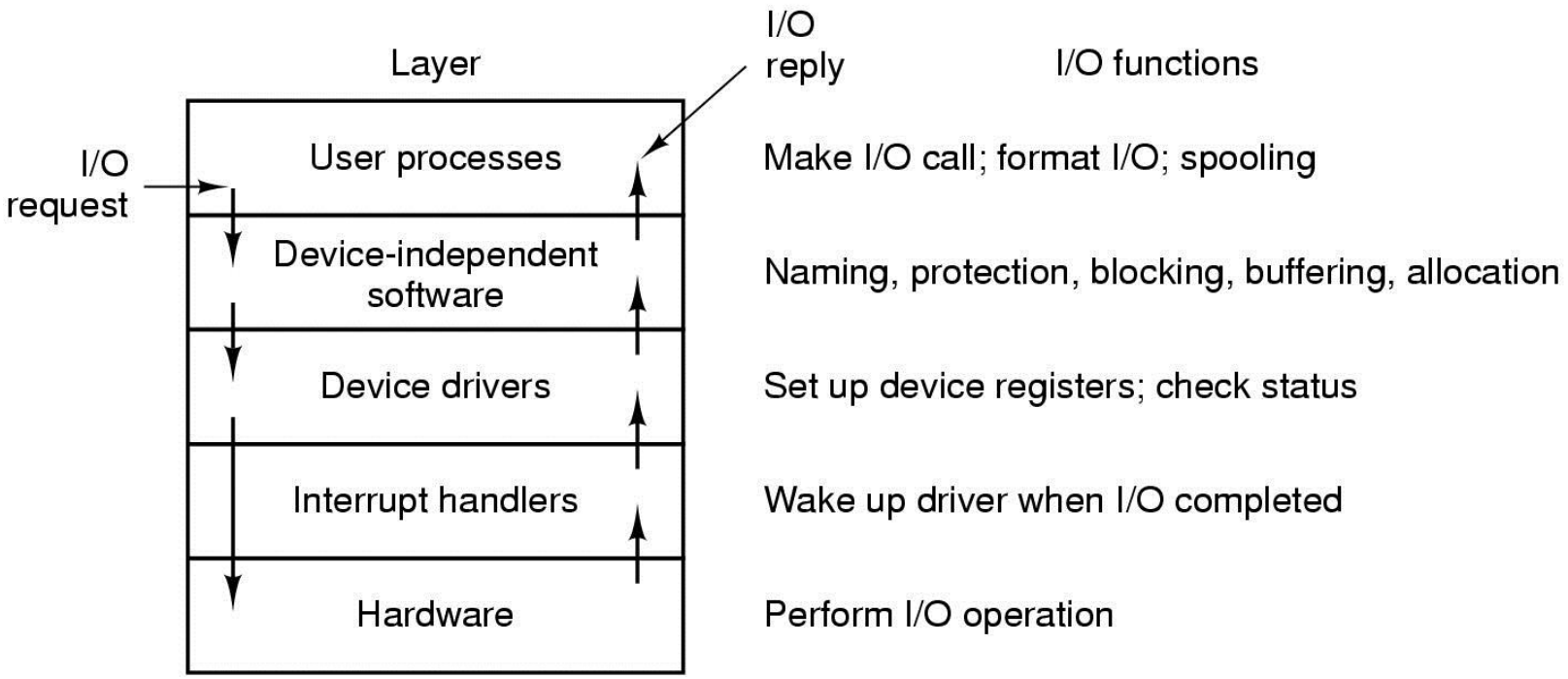
Device-independent Block Size

- There are different kind of disk drives. Each may have a different physical sector size.
- A file system uses a block size while mapping files into logical disk blocks.
- This layer can hide the physical sector size of different disks and can provide a fixed and uniform disk block size for higher layers, like the file system
  - Several sectors can be treated as a single logical block.

User-Space I/O Software

- This includes
  - The I/O libraries that provides the implementation of I/O functions which in turn call the respective I/O system calls.
    - These libraries also implemented formatted I/O functions such as *printf*() and *scanf*()
  - Some programs that does not directly write to the I/O device, but writes to a spooler.

Summary of I/O Software



Layers of the I/O system and the main functions of each layer

Homework

- P428 3, 9, 11