Input/Output

EECE6029

Yizong Cheng

2/15/2016

Overview

- OS controls I/O devices =>
  - Issue commands,
  - handles interrupts,
  - handles errors
- Provide easy to use interface to devices
  - Hopefully device independent
- First look at hardware, then software
  - Emphasize software
  - Software structured in layers

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 typical device, network, and bus data rates.

Block and Character I/O

- Two types of I/O devices- block, character
- Block- can read blocks independently of one another
  - Hard disks, CD-ROMs, USB sticks
  - 512-32,768 bytes
- Character-accepts characters without regard to block structure
  - Printers, mice, network interfaces
- Not everything fits, e.g. clocks don’t fit
- Division allows for OS to deal with devices in device independent manner
  - File system deals with blocks

Mechanical and Electronic Components

- I/O unit has 2 components-mechanical, electronic (controller)
  - Controller is a chip with a connector which plugs into cables to device
- Disk
  - Disk might have 10,000 sectors of 512 bytes per track
  - Serial bit stream comes off drive
  - Has preamble, 4096 bits/sector, error correcting code
  - Preamble has sector number, cylinder number, sector size….
- Controller assembles block from bit stream, does error correction, puts into buffer in controller
  - Blocks are what are sent from a disk

Data Buffers

- Controller has registers which OS can write and read
- Write-gives command to device
- Read-learn device status……
- Devices have data buffer which OS can read/write (e.g. video RAM, used to display pixels on a screen)
- How does CPU communicate with registers and buffers?

Memory-Mapped I/O

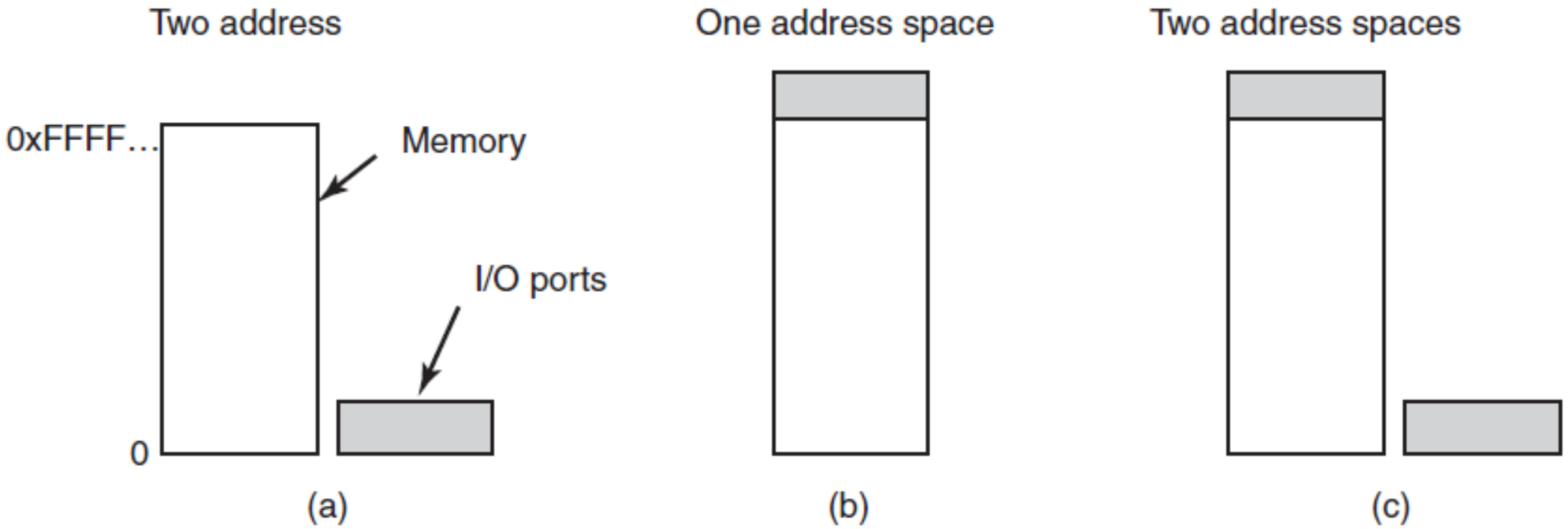


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

I/O Space and Memory Space

- First scheme
  - Puts read on control line
  - Put address on address line
  - Puts I/O space or memory space on signal line to differentiate
  - Read from memory or I/O space
- Memory mapped approach
  - Put address on address line and let memory and I/O devices compare address with the ranges they serve

Memory-Mapped I/O Advantages

- Don’t need special instructions to read/write control registers=> can write a device driver in C
- Don’t need special protection to keep users from doing I/O directly. Just don’t put I/O memory in any user space

Memory-Mapped I/O Disadvantages

- Can cache memory words, which means that old memory value (e.g. for port 4) could remain in cache
- => have to be able to disable caching when it is worthwhile
- I/O devices and memory have to respond to memory references
- Works with single bus because both memory and I/O look at address on bus and decide who it is for
- Harder with multiple buses because I/O devices can’t see their addresses go by any more

Memory-Mapped I/O

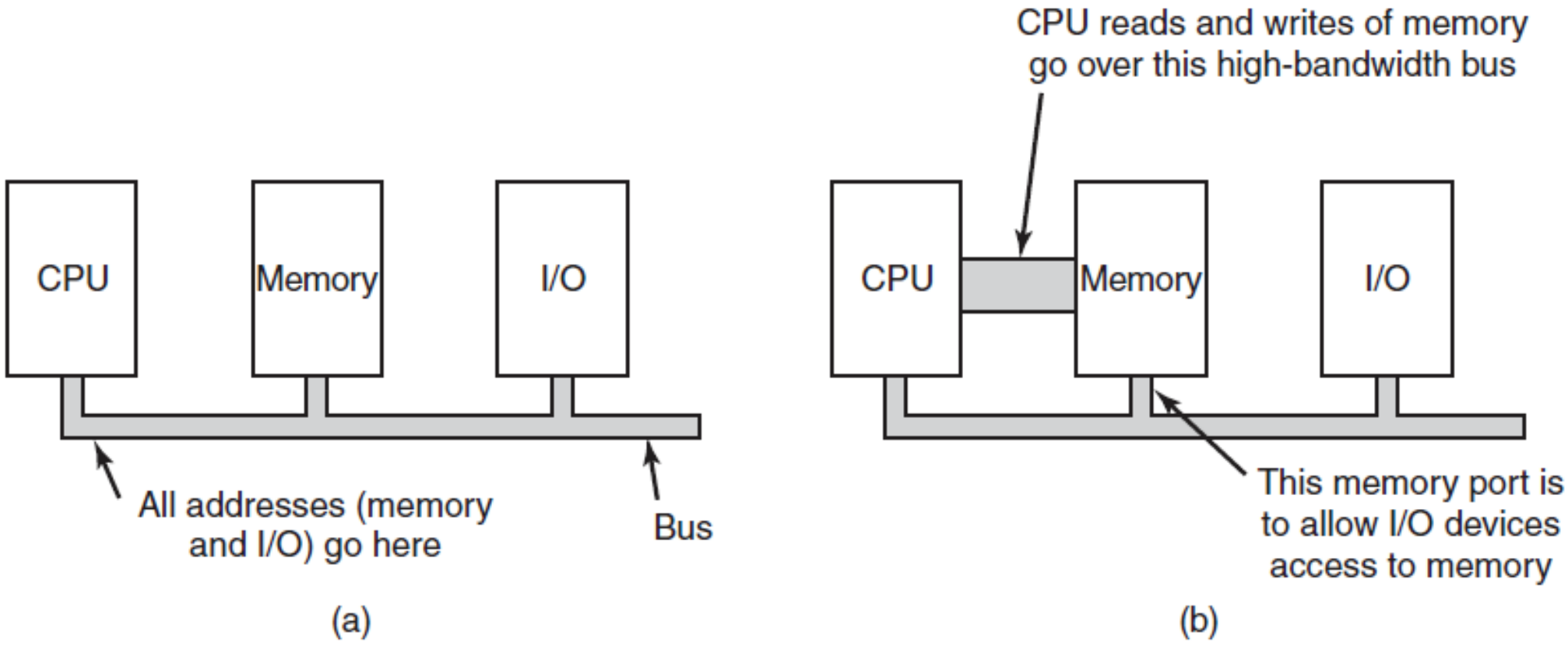


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

Direct Memory Access

- CPU COULD request data one byte at a time from I/O controller
- Big waste of time, use DMA
- DMA controller on mother-board; normally one controller for multiple devices
- CPU reads/writes to registers in controller
  - Memory address register
  - Byte count register
  - Control registers-I/O port, direction of transfer, transfer units (byte/word), number of bytes to transfer in a burst

When DMA Is not Used

- Controller reads a block into its memory
- Computes checksum
- Interrupts OS
- Sends byte at a time to memory

DMA Transfer

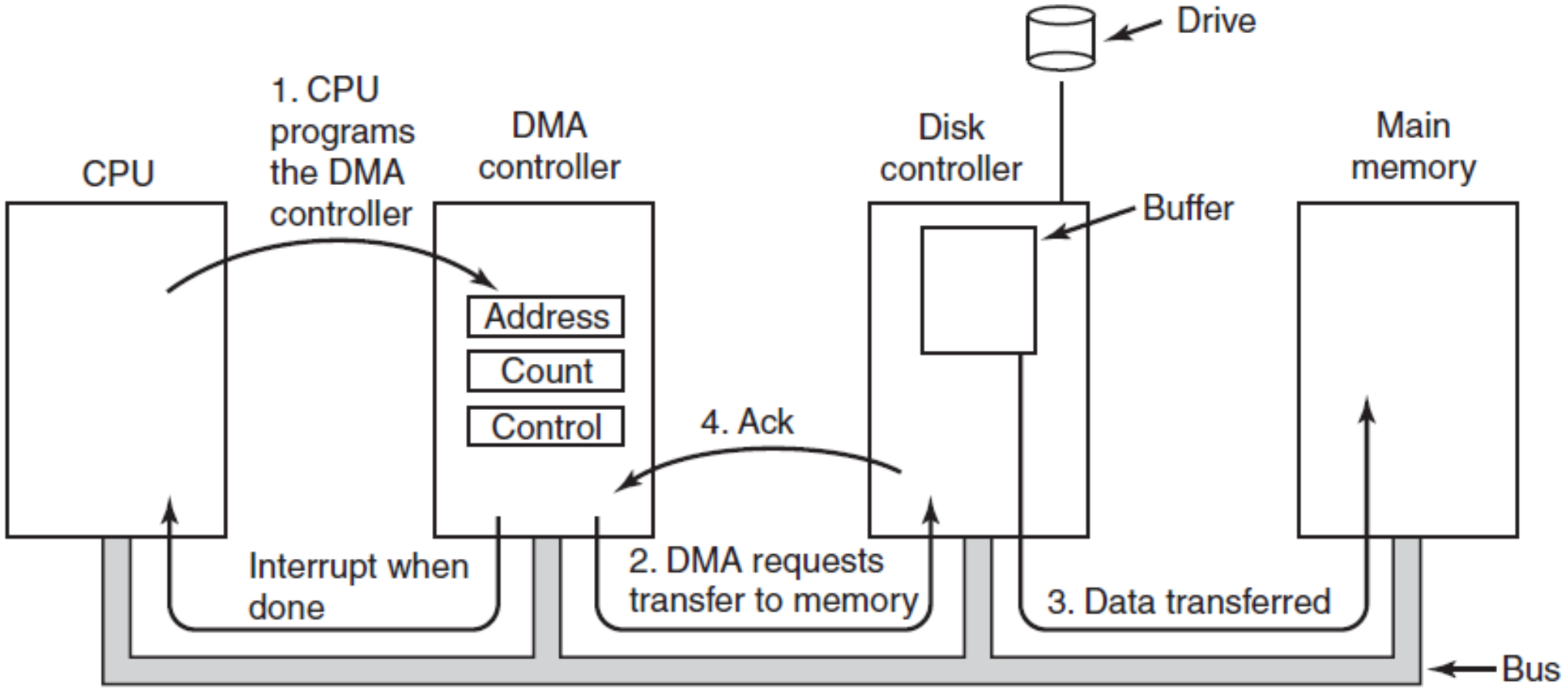


Figure 5-4. Operation of a DMA transfer.

DMA Controller Modes

- Cycle stealing mode-transfer goes on word at a time, competing with CPU for bus cycles. Idea is that CPU loses the occasional cycle to the DMA controller
- Burst mode-DMA controller grabs bus and sends a block
- Fly by mode-DMA controller tells device controller to send word to it instead of memory. Can be used to transfer data between devices.
- Why buffer data in controllers?
  - Can do check-sum
  - Bus may be busy-need to store data someplace
- Is DMA really worth it? Not if
  - CPU is much faster then DMA controller and can do the job faster
  - don’t have too much data to transfer

How and Interrupt Happens

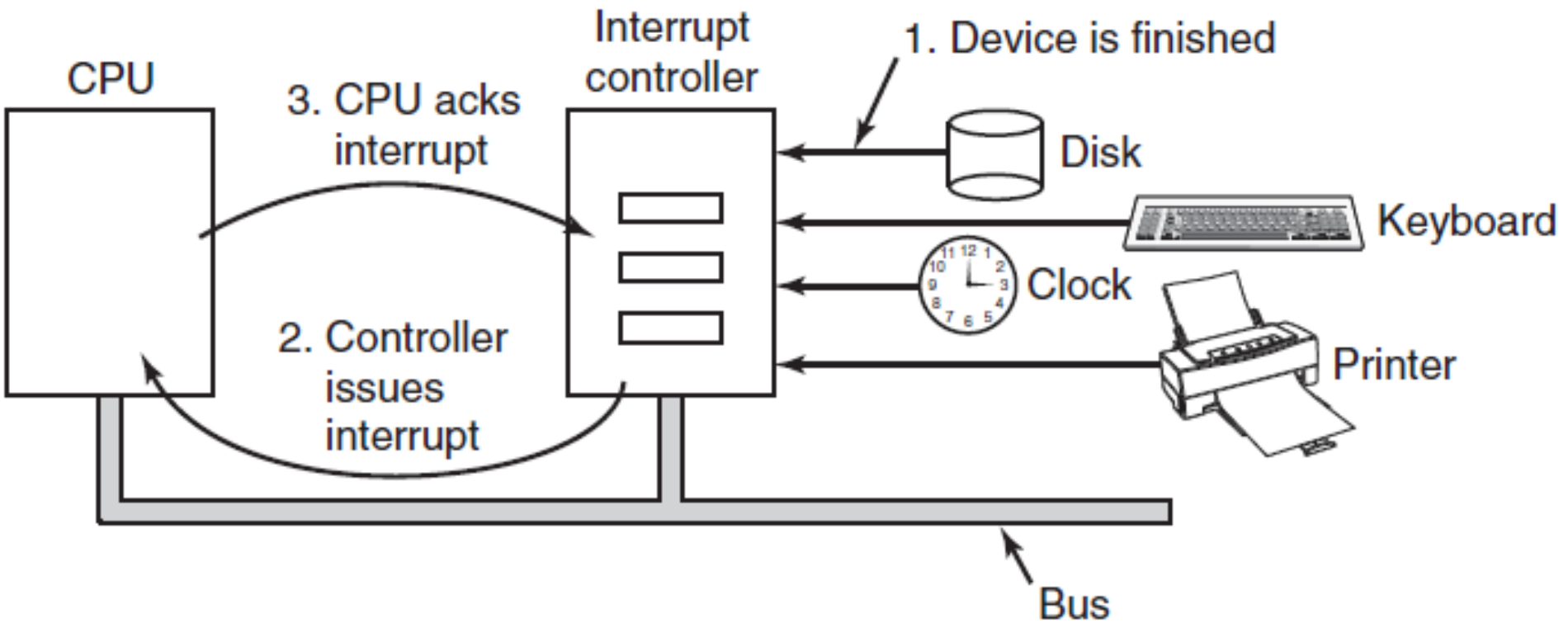


Figure 5-5. How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

Interrupt Processing

- Controller puts number on address line telling CPU which device wants attention and interrupts CPU
- Table (interrupt vector) points to interrupt service routine
- Number on address line acts as index into interrupt vector
- Interrupt vector contains PC which points to start of service routine
- Interrupt service routine acks interrupt
- Saves information about interrupted program
- Where to save information
  - User process stack, kernel stack are both possibilities
  - Both have problems

Precise Interrupts

- 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.
- Execution state of the instruction pointed to by the PC is known.

Precise and Imprecise Interrupts

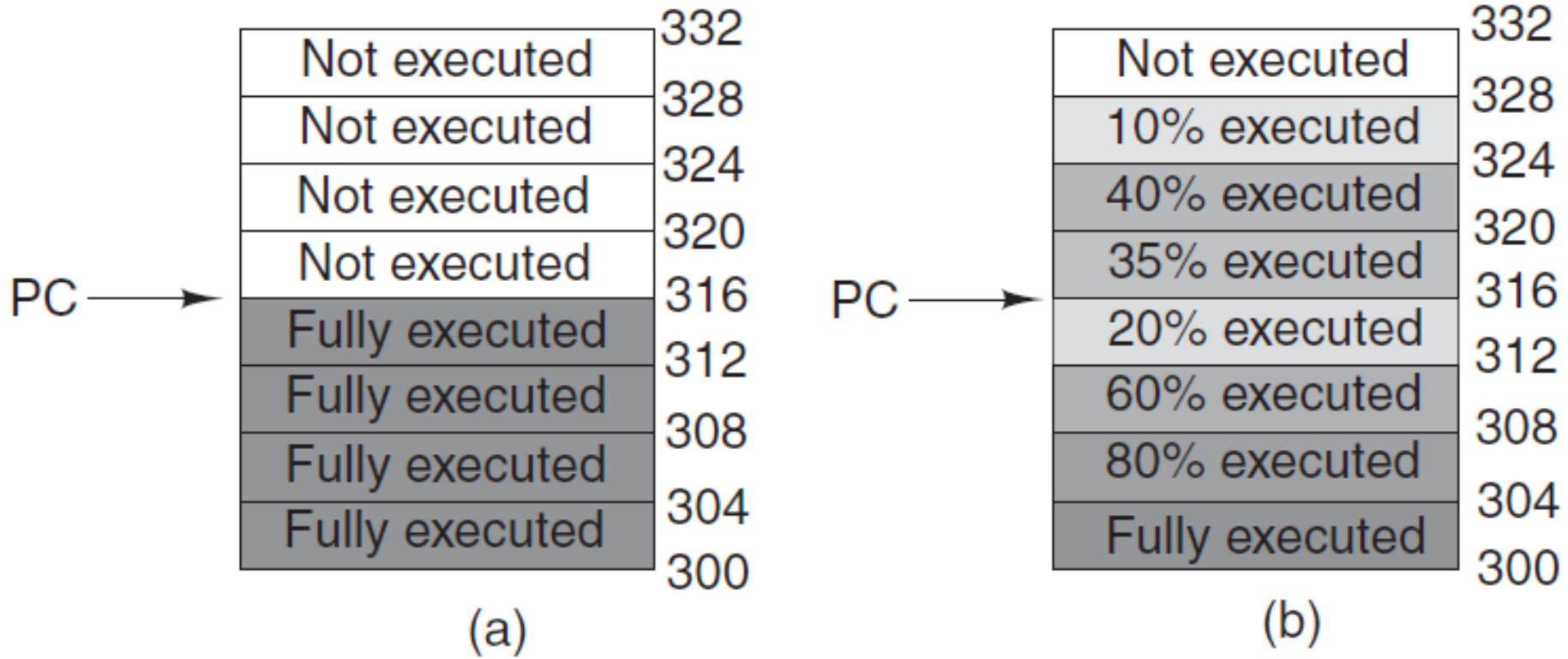


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

I/O Software Goals

- Device independence-don’t have to specify the device when accessing the device
- Uniform naming-name should not depend on device type
- Error handling-do it as close to the device as possible (e.g. controller should be the first to fix error, followed by the driver)
- OS needs to make I/O operations blocking (e.g. program blocks until data arrives on a read) because it is easy to write blocking ops
- Buffering- e.g. when a packet arrives
- Shared devices (disks) and un-shared devices (tapes) must be handled

Programmed I/O

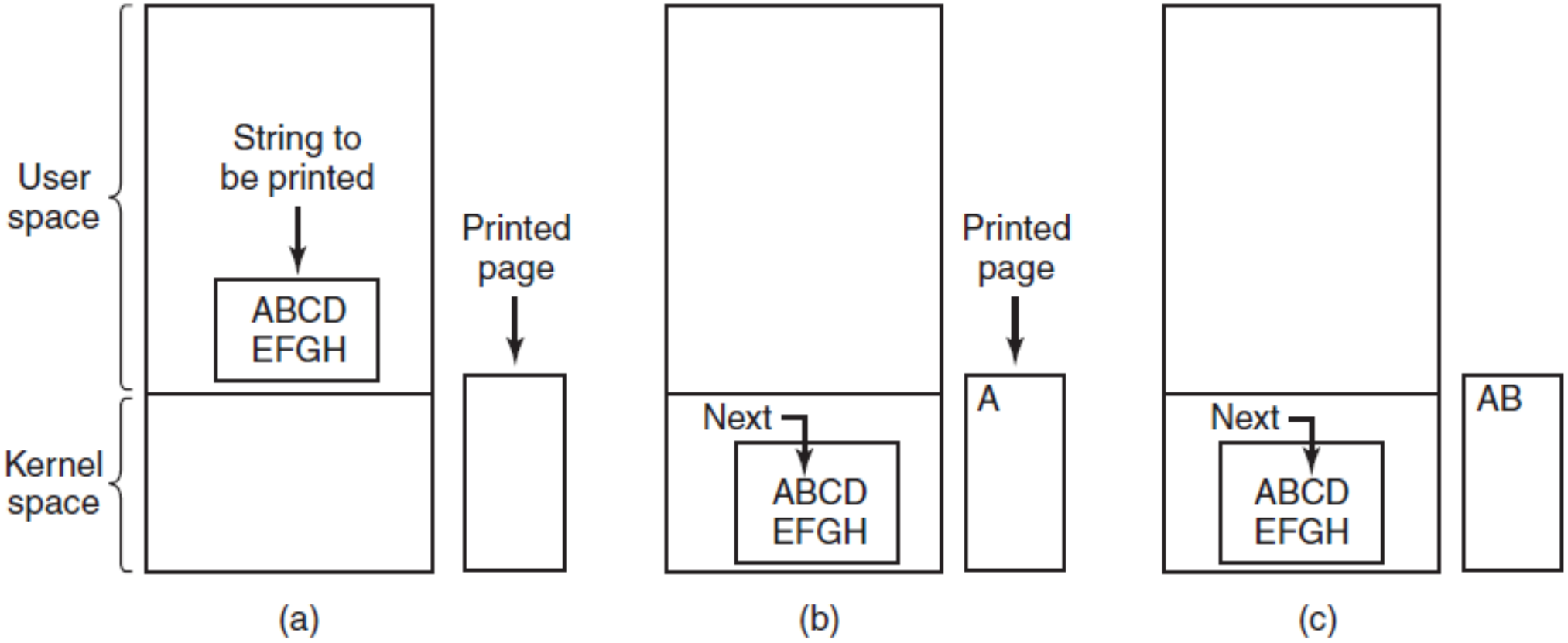


Figure 5-7. Steps in printing a string.

Polling or Busy Waiting for One Character

```
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

- Idea: block process which requests I/O, schedule another process
- Return to calling process when I/O is done
- Printer generates interrupt when a character is printed
- Keeps printing until the end of the string
- Re-instantiate calling process

Interrupt Driven I/O

```
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

```
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

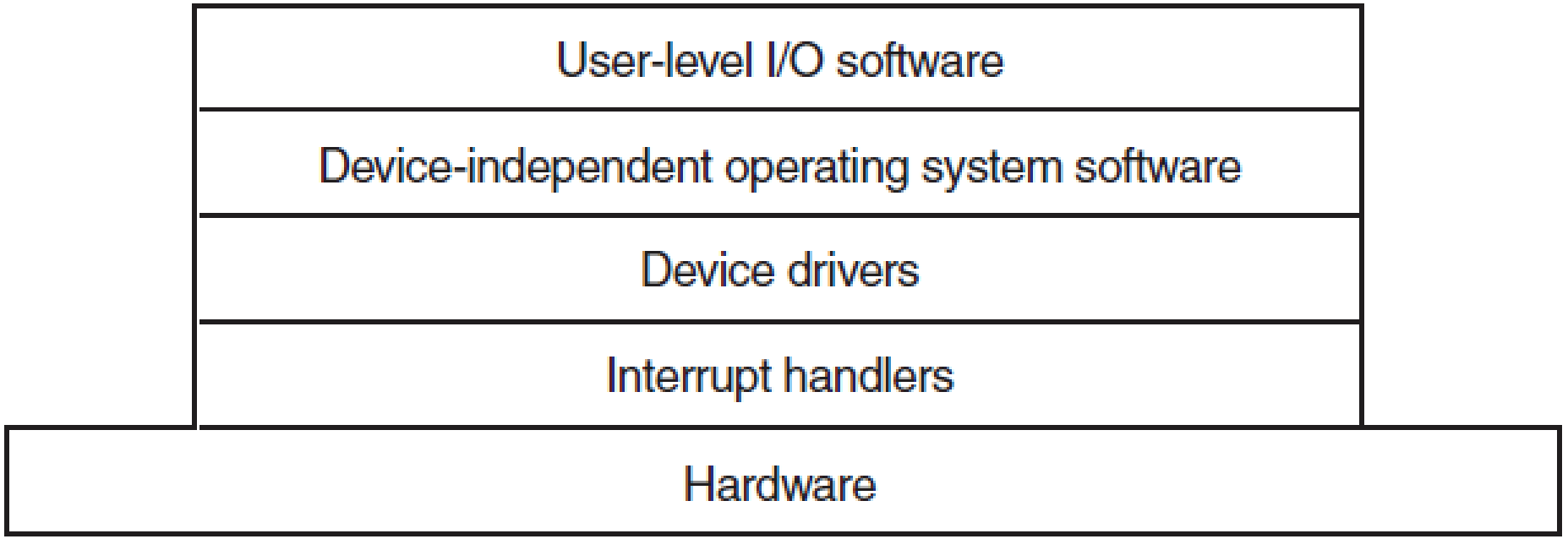


Figure 5-11. Layers of the I/O software system.

Interrupt Handlers

- The idea: driver starting the I/O blocks until interrupt happens when I/O finishes
- Handler processes interrupt
- Wakes up driver when processing is finished
- Drivers are kernel processes with their very own
  - Stacks
  - PCs
  - states

Interrupt Processing Details

- Save registers not already saved by interrupt hardware.
- Set up a context for the interrupt service procedure.
- Set up a stack for the interrupt service procedure.
- Acknowledge the interrupt controller. If there is no centralized interrupt controller, re-enable interrupts.
- Copy the registers from where they were saved to the process table.
- Run the interrupt service procedure.
- Choose which process to run next.
- Set up the MMU context for the process to run next.
- Load the new process' registers, including its PSW.
- Start running the new process.

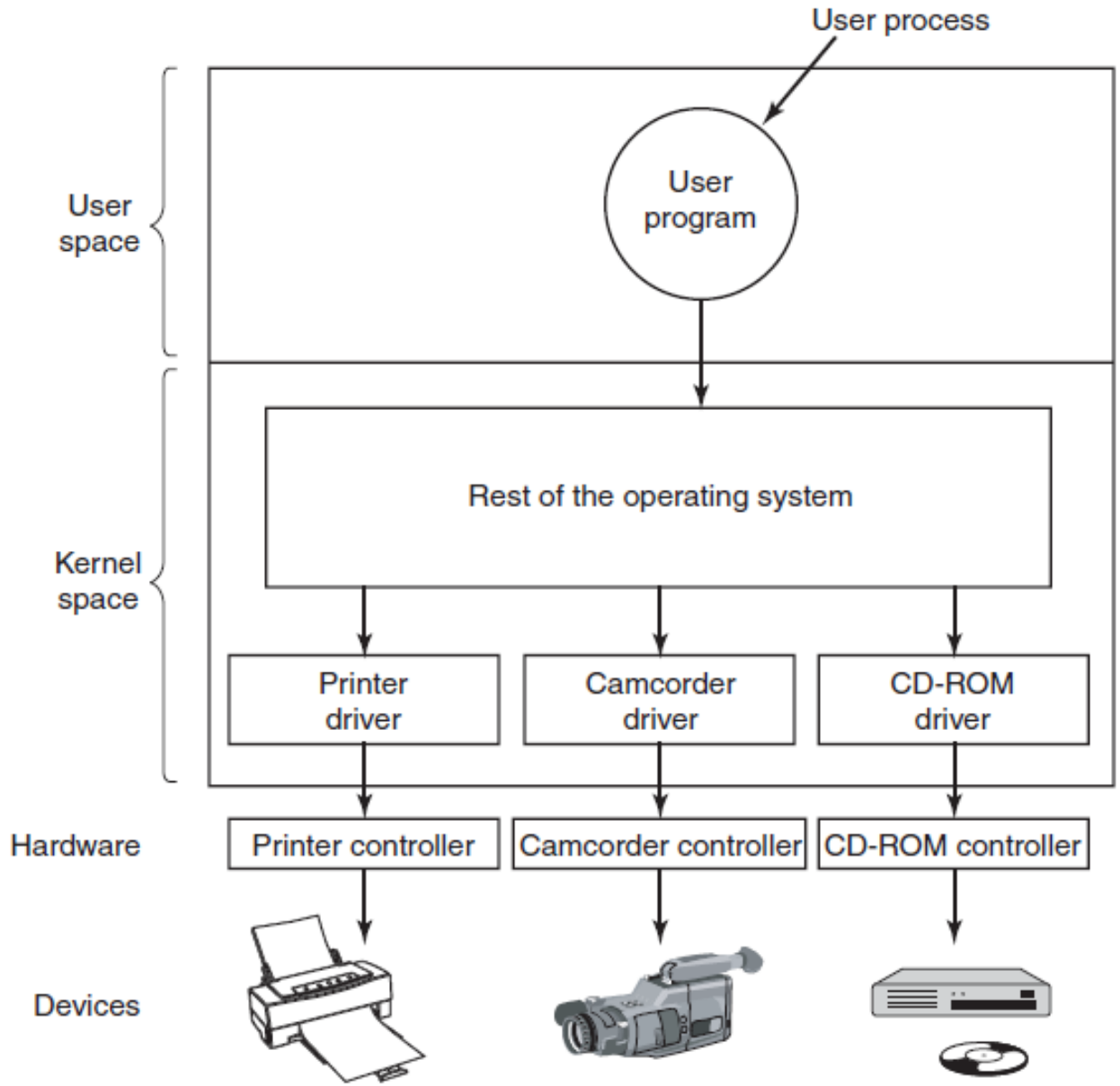


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Device Drivers

- Driver contains code specific to the device
- Supplied by manufacturer
- Installed in the kernel
  - User space might be better place
  - Why? Bad driver can mess up kernel
- Need interface to OS
  - block and character interfaces
  - procedures which OS can call to invoke driver (e.g. read a block)

Device Driver Acts

- Checks input parameters for validity
- Abstract to concrete translation (block number to cylinder, head, track, sector)
- Check device status. Might have to start it.
- Puts commands in device controller’s registers
- Driver blocks itself until interrupt arrives
- Might return data to caller
- Does return status information
- The end
- Drivers should be re-entrant
- OS adds devices when system (and therefore driver) is running

Device-Independent I/O Software

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.

Buffering

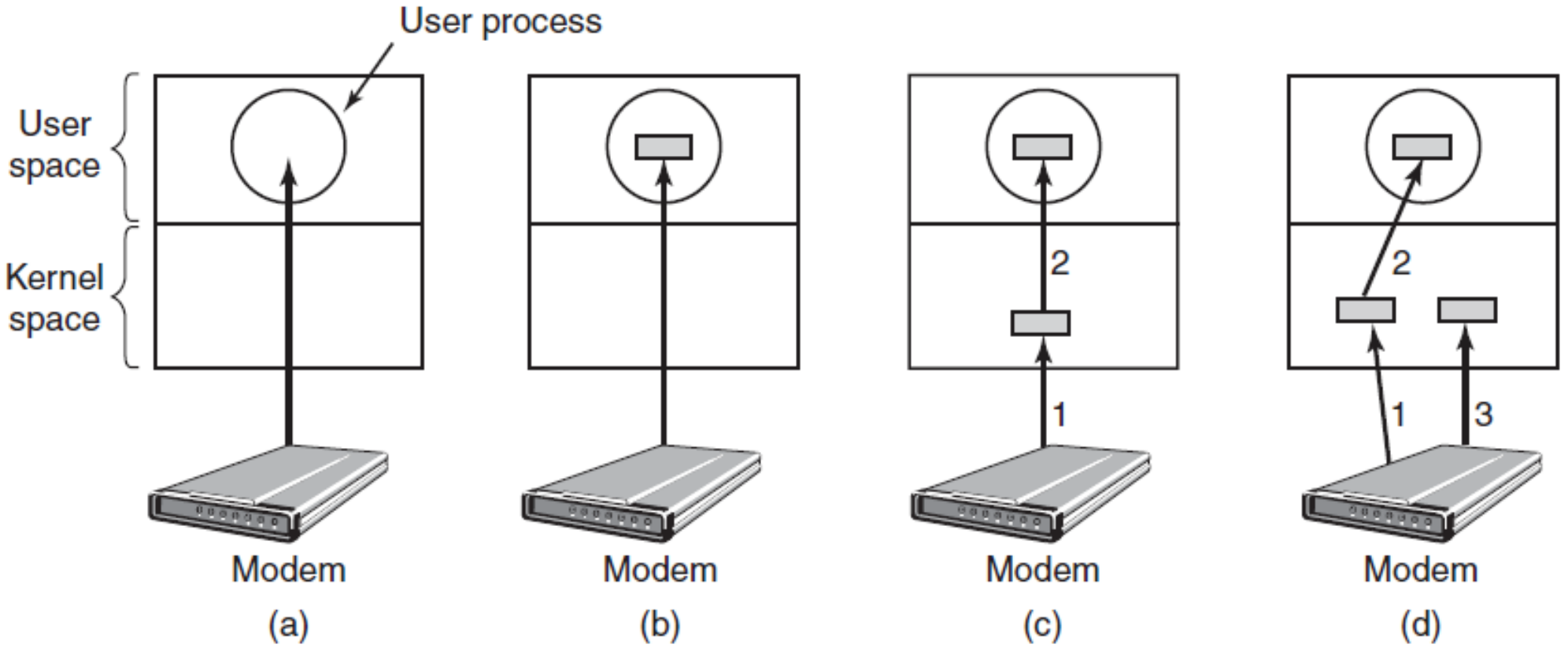


Figure 5-15. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Layers of the I/O System

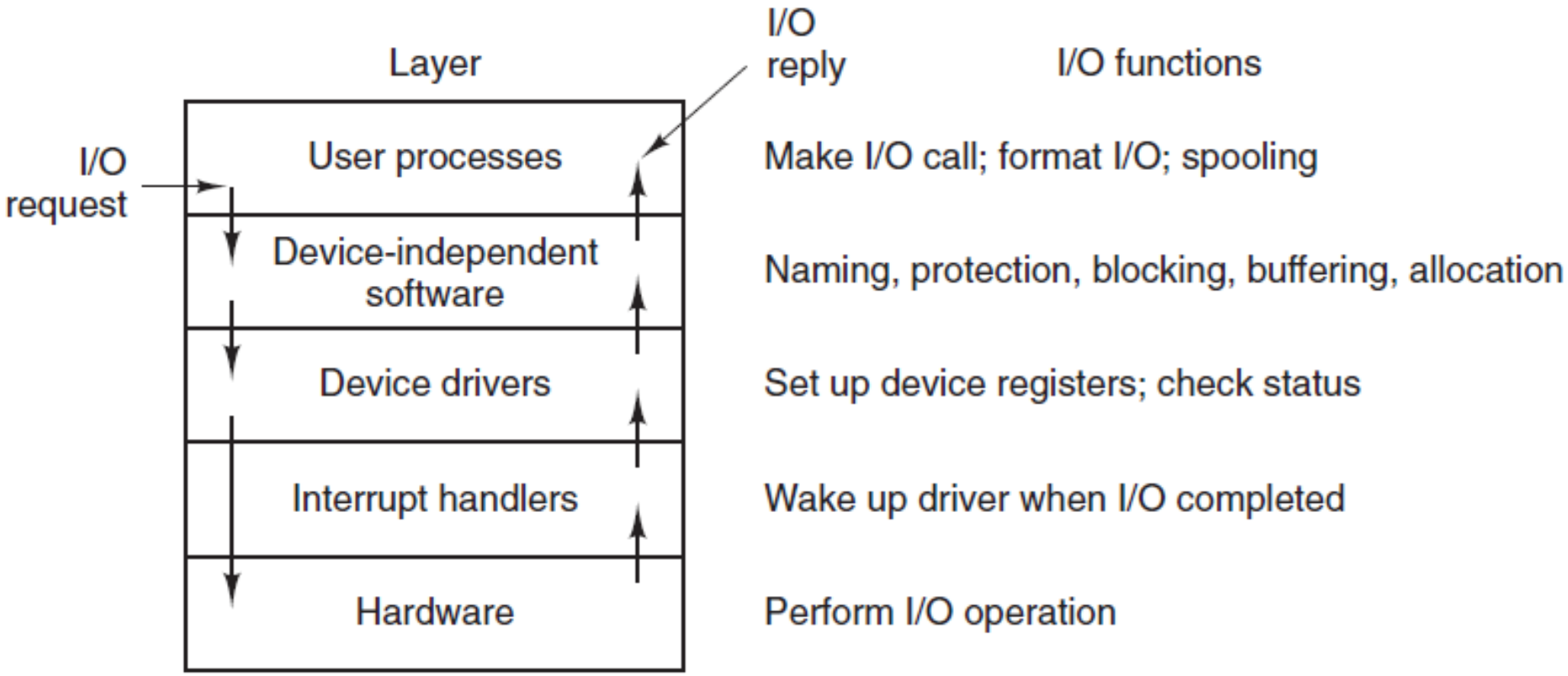
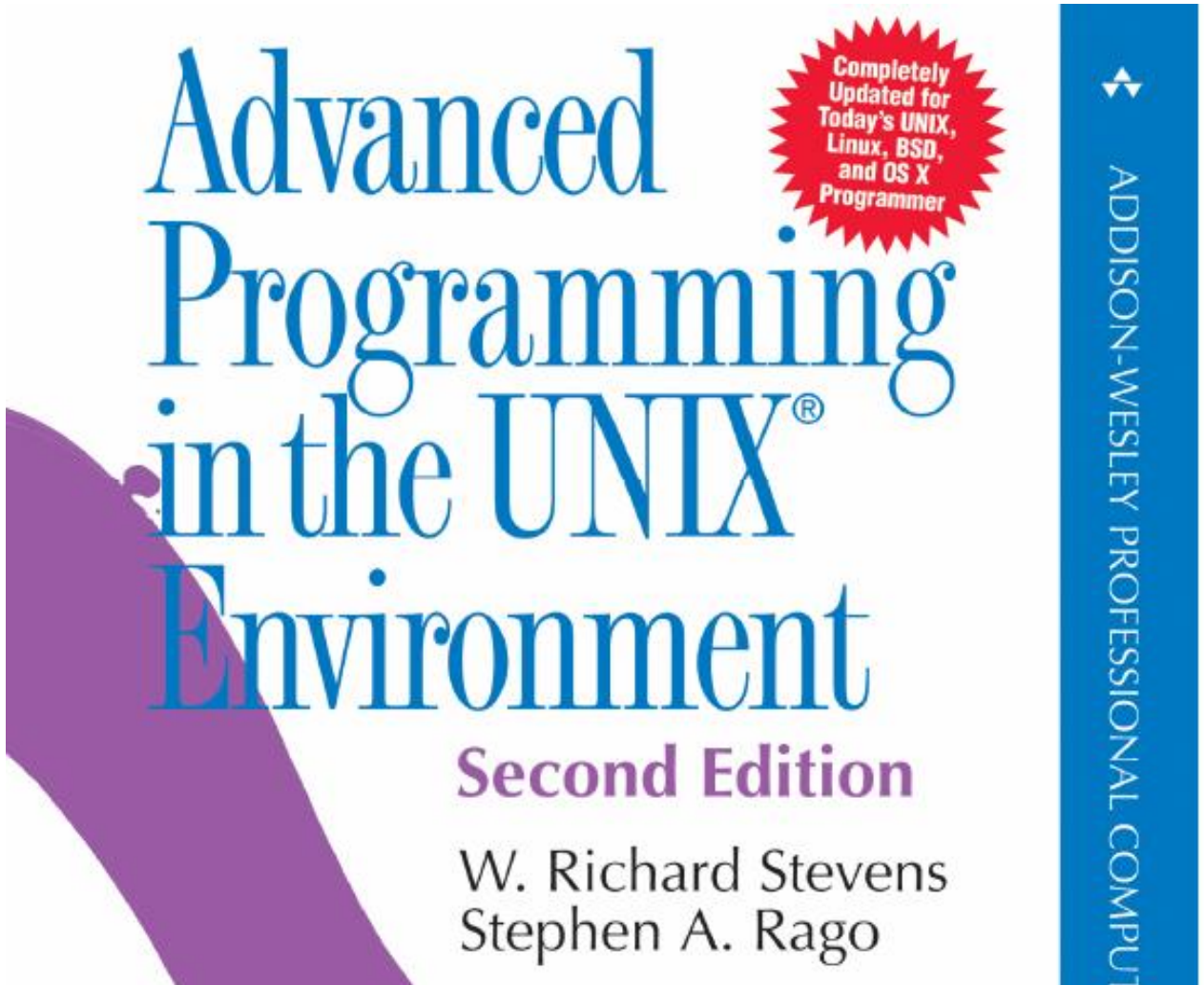


Figure 5-17. Layers of the I/O system and the main functions of each layer.

The APUE Book



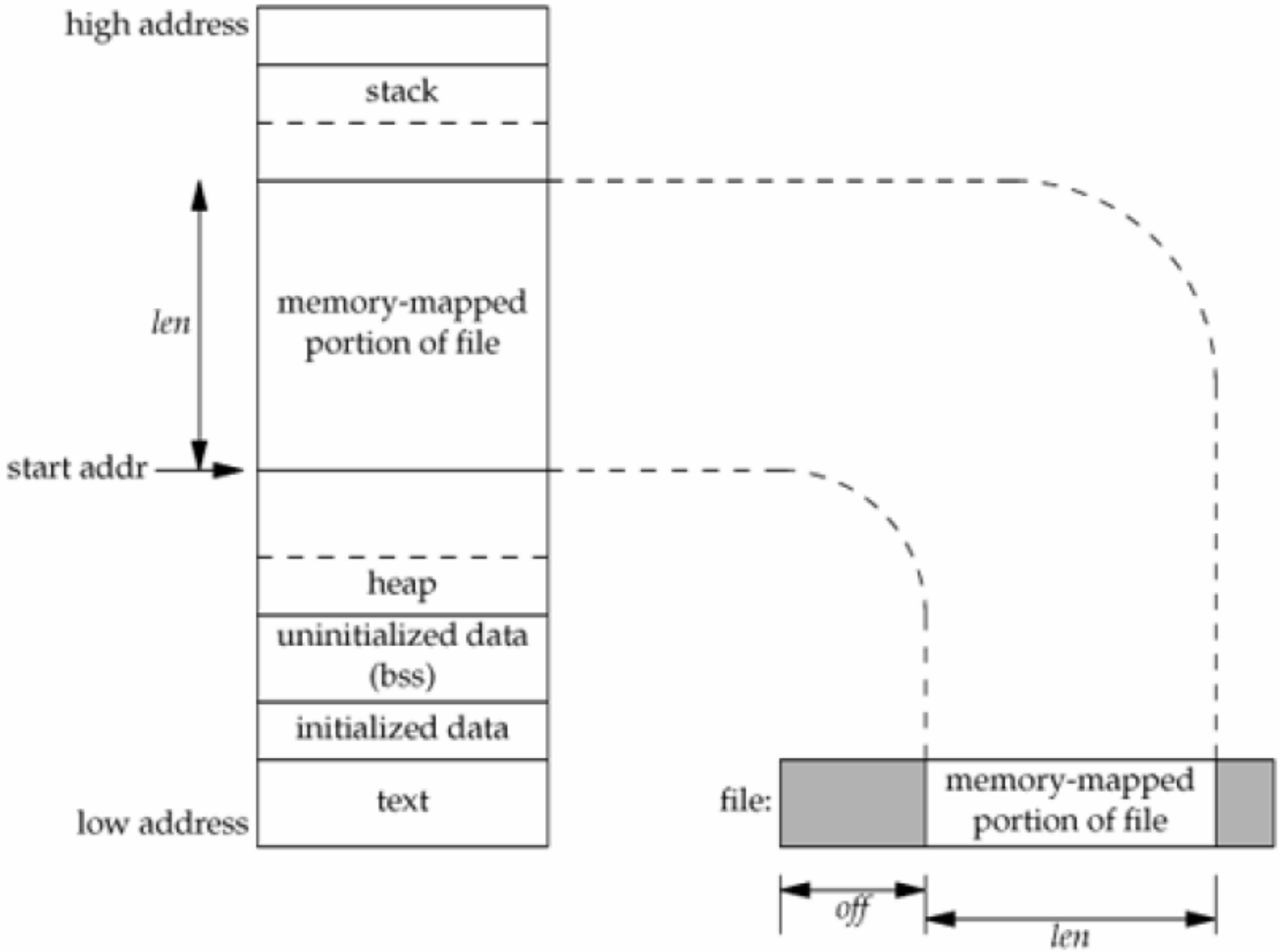
The mmap System Call of UNIX

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int filedes,
           off_t off );
```

Returns: starting address of mapped region if OK, `MAP_FAILED` on error

```
#include <sys/mman.h>

int munmap(caddr_t addr, size_t len);
```

Returns: 0 if OK, –1 on error


high address
stack
len
memory-mapped
portion of file
start addr
heap
uninitialized data
(bss)
initialized data
text
low address
file:
memory-mapped
portion of file
off
len

```
#define FILE_MODE 0700

int main(int argc, char *argv[])  // copy file argv[1] to argv[2] with mmap
{
   int fdin, fdout;
   void *src, *dst;
   struct stat statbuf;
   off_t file_position;

   fdin = open(argv[1], O_RDONLY);
   fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
   fstat(fdin, &statbuf);
   file_position = lseek(fdout, statbuf.st_size - 1, SEEK_SET);
   write(fdout, "", 1);
   src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED, fdin, 0);
   dst = mmap(0, statbuf.st_size, PROT_WRITE, MAP_SHARED, fdout, 0);
   memcpy(dst, src, statbuf.st_size);
   close(fdin); close(fdout);
   munmap(src, statbuf.st_size);
   munmap(dst, statbuf.st_size);
   return 0;
}
```

Anonymous Memory Mapping

```
int update(int *ptr){ return (*ptr)++; }

int main(int argc, char *argv[])
{
   int i, counter;
   pid_t pid;
   void *area;

   area = mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_ANON | MAP_SHARED, -1, 0);
   if (area == MAP_FAILED) exit(1);
   pid = fork();
   for (i = 0; i < 10; i += 2){
        counter = update((int *)area);
        printf("%d %d %d\n", pid, i, counter);
   }
}
```