Chapter 5 - Input / Output

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Motivation

Recall that an OS provides abstractions for:

1. Processes;
2. Addresses spaces;
3. Files;
4. Etc...

OS must also also:

- control all the computer's I/O devices;

In your opinion what does this mean: ``control all the computer's I/O devices''? Any ideas?

In your opinion what does this mean: ``control all the computer's I/O devices''? Any ideas?

- Issue commands to devices;
- Catch interrupts
- Handle errors;
- Provide API:
  - Interface should be the same for all devices:
    - Not always possible...

Principle of I/O Hardware

This chapter focuses on:

Programming I/O devices

This chapter does not focus on:

- Designing I/O devices;
- Building I/O devices;
- Maintaining I/O devices;

I/O Devices

Most I/O devices can be divided into two categories:

- **Block devices:** store information in fixed-size blocks:
  - Each block has its own address;
  - All transfers are in units of one or more entire blocks.
  - Possible to read / write each block independently of all others;
  - *E.g.:* hard disks, blu-ray disk, USB sticks, etc...
- **Character devices:** reads / writes a stream of characters:
  - Without regard to any block structure;
  - Not addressable
  - *E.g.:* printers, network interfaces, mice, etc...

I/O devices cover a huge range in speeds:

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: Some typical device, network and bus data rates (Source: (Tanenbaum and Bos, 2015))

Device Controllers

I/O units often consist of:

- **Mechanical component:** device itself (disk head, laser, etc...)
- **Electronic component:**
  - A.k.a **device controller** or **adapter**
  - Many controllers can handle several identical devices
  - **Interface** examples between controller and device:
    - SATA;
    - SCSI;
    - USB;

Interface between controller and device is often low-level.

Disk Example

Consider a disk with 2,000,000 sectors of 512 bytes per track:

- Information that comes off the drive is a serial bit stream:
  - Starting with a preamble:
    - Cylinder number;
    - Sector number;
    - Sector size;
    - Synchronization information
  - Then the 4096 bits in a sector;
  - And finally a checksum;

Disk Example

Controller's job is to:

- Convert the serial bit stream into a block of bytes:
  - Block of bytes is assembled in a buffer inside the controller;
- and perform any error correction necessary:
  - If block is error free: copy block to memory;

LCD Display Example

LCD display monitor controller also works at a low level:

- Reads bytes containing the characters to be displayed from memory;
- Generates the signals to modify pixels in order to write them on screen;
- If it were not for the display controller:
  - OS programmer would have to specify electric fields of all pixels;
  - With the controller the OS:
    - Initializes the controller with a few parameters;
    - Controller takes care of specifying electric fields.

Memory-Mapped I/O

Each controller has **registers:**

- Used for communicating with the CPU;
- OS can **write** into these registers in order to:
  - Deliver data, accept data, switch device on or off, etc...
- OS can **read** from these registers in order to:
  - Learn device state and so on;

Besides registers many devices also have a **data buffer:**

- OS can read or write into;
- *E.g.:* computers display pixels on the screen through video ram:
  - Data buffer available for programs or OS to write into.

How does the CPU communicate with the control registers and also with the device data buffers? Any ideas?

How does the CPU communicate with the control registers and also with the device data buffers? Any ideas?

Two alternatives exist:

- Each control register is assigned an **I/O port** number;
  - Set of all I/O ports forms the **I/O port space**;
- Map all control registers into memory space;
  - A.k.a. **memory-mapped I/O**;
  - Approach used in Computer Architecture labs =)

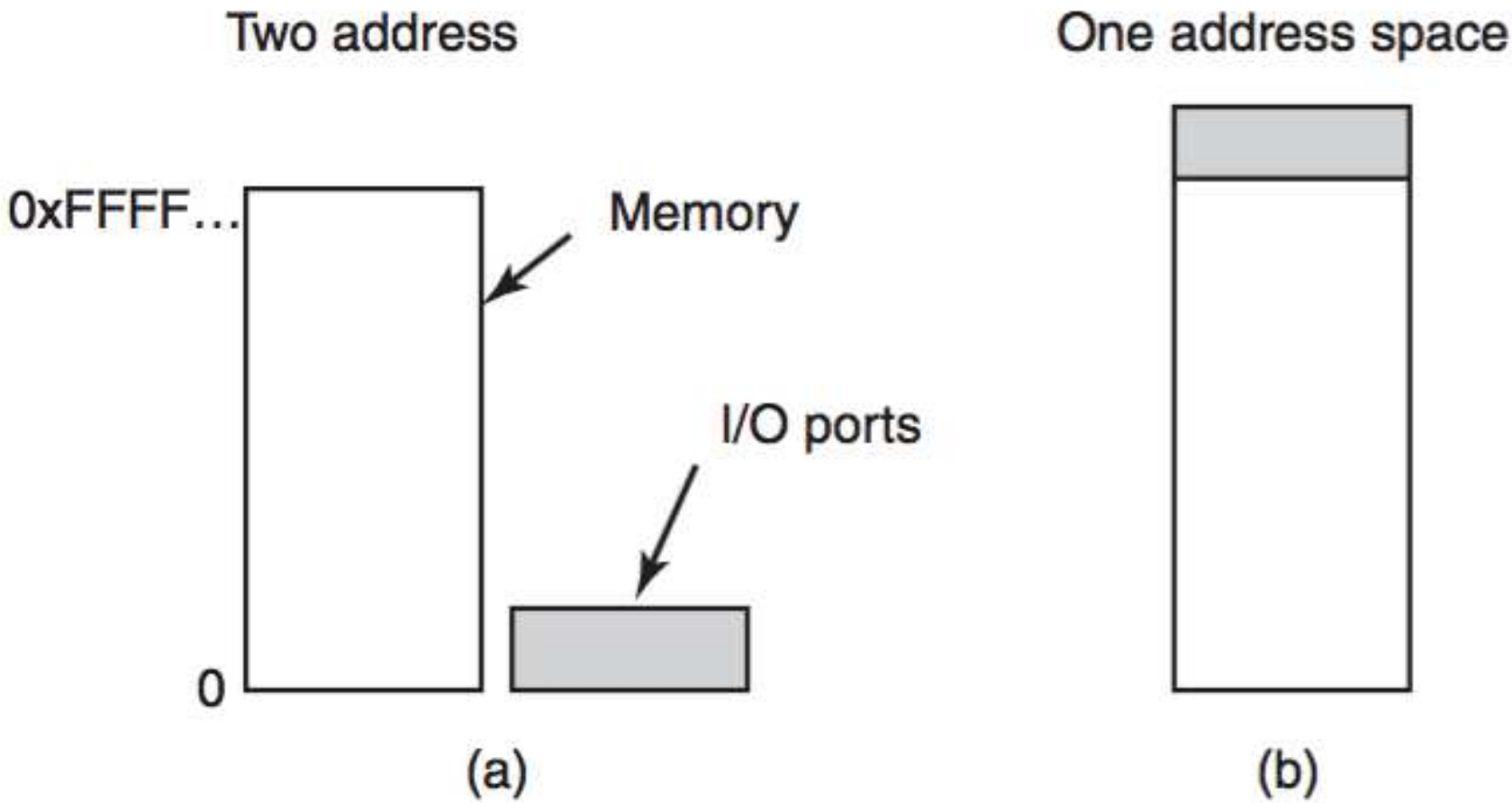

Figure: (a) Separate I/O and memory space. (b) Memory-mapped I/O, approach used in computer architecture laboratories. (Source: (Tanenbaum and Bos, 2015))

How do these schemes actually work in practice?

How do these schemes actually work in practice?

When the CPU wants to read a word (memory or I/O port) (1/2):

1. Address placed on the bus' address lines;
2. READ signal on a bus' control line;

How do these schemes actually work in practice?

When the CPU wants to read a word (memory or I/O port) (2/2):

3. Additional signal on the bus' control line:
   - To tell whether I/O space or memory space is needed;
   - If it is memory space:
     - Memory responds to the request;
   - If it is I/O space:
     - I/O device responds to the request.

If there is only memory space:

- Every memory module and every I/O device:
  - Compares address lines to the range of addresses that it services;
  - If the address falls in its range, it responds to the request;
  - Since no address is ever assigned to both memory and an I/O device:
    - There is no ambiguity and no conflict.

Direct Memory Access

Eventually with I/O: CPU needs to exchange data with devices

- CPU can request data from an I/O controller one byte at a time...

Can you see any problem with this approach? Any ideas?

Direct Memory Access

Eventually with I/O: CPU needs to exchange data with devices

- CPU can request data from an I/O controller one byte at a time...

Can you see any problem with this approach? Any ideas?

- Wasteful of CPU's time. Why?

Direct Memory Access

Eventually with I/O: CPU needs to exchange data with devices

- CPU can request data from an I/O controller one byte at a time...

Can you see any problem with this approach? Any ideas?

- Wasteful of CPU's time. Why?
  - CPU is a powerful tool that is being used to copy bytes...

Do you know any other mechanism for copying data between I/O devices and memory?

Do you know any other mechanism for copying data between I/O devices and memory?

Direct Memory Address (DMA): (1/2)

- OS can use only DMA if the hardware has a DMA controller;
- Usually a single DMA controller is available:
  - Typically on the motherboard;
  - Regulates transfers to multiple devices, often concurrently.

Direct Memory Address (DMA): (2/2)

- DMA has access to the system bus independent of the CPU;
- DMA contains several registers that can be written and read by the CPU:
  - **Memory Address Register**;
  - **Byte count register**;
  - One or more control registers:
    - Specify the I/O port to use;
    - Read / Write from / into I/O device;
    - Transfer unit: byte? word?
    - Number of units to transfer in one burst;

Programmed Interruptions

Consider how disk reads occur when DMA is **not used**, the disk controller:

1. Reads block bit by bit:
   - Until entire block is in the controller's internal buffer.
2. Computes checksum to verify that no read errors have occurred.
3. Causes an interrupt:
   - When the OS executes:
     - Disk block is read from the controller's buffer and stored in main memory;

DMA Transfers

Consider how disk reads occur when DMA is **used** (1/5):

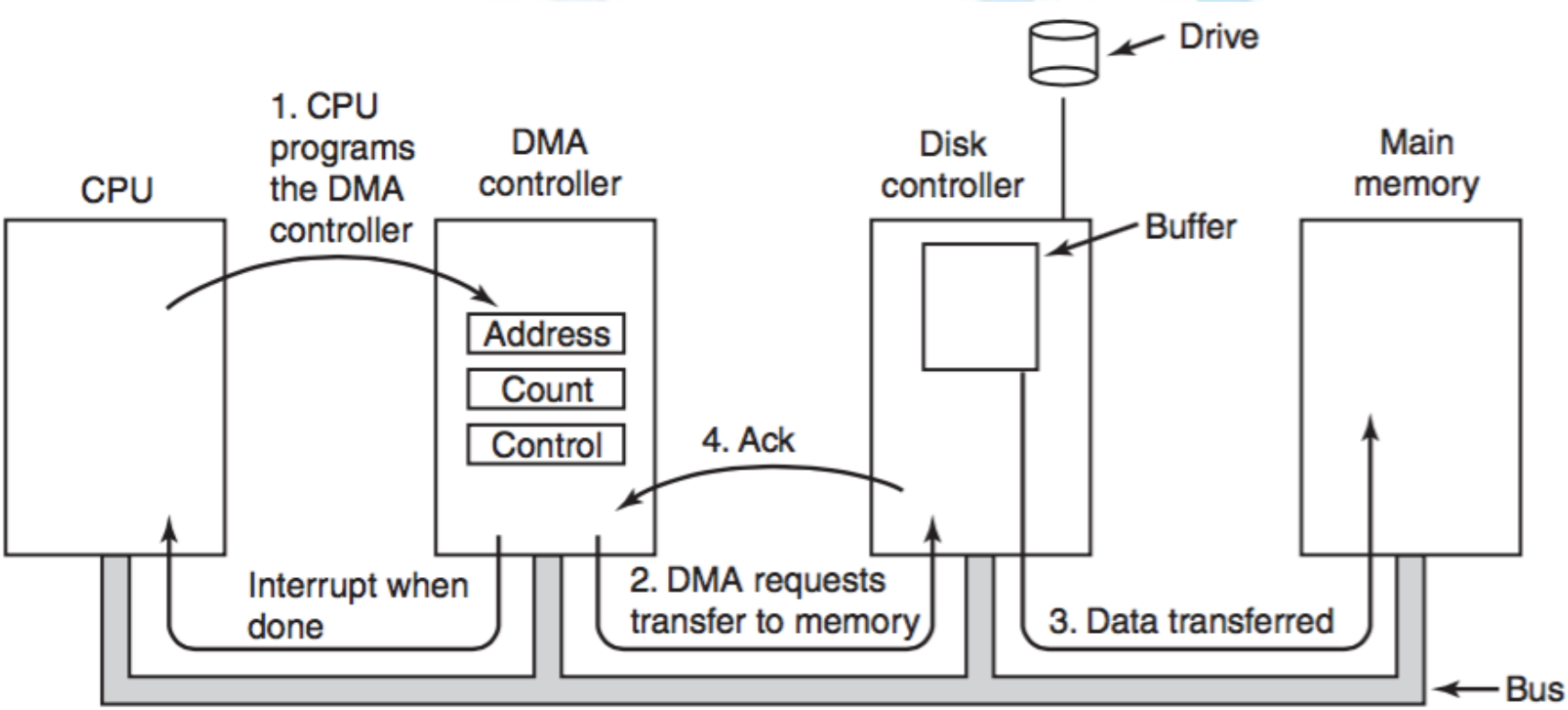

Figure: Operation of a DMA transfer. (Source: (Tanenbaum and Bos, 2015))

DMA Transfers

Consider how disk reads occur when DMA is **used** (2/5):

1. CPU programs the DMA controller's registers so that:
   - Source device and destination memory addresses are configured;
2. DMA controller initiates transfer by:
   - Issuing a read request over the bus to the disk controller;
   - Placing destination address on the bus address lines;

DMA Transfers

Consider how disk reads occur when DMA is **used** (3/5):

3. Once word has been placed on the disk's controller internal buffer:
   - Disk controller issues write request to memory module;
4. When memory write is complete:
   - Disk controllers sends acknowledgement signal to the DMA controller;

DMA Transfers

Consider how disk reads occur when DMA is **used** (4/5):

5. DMA controller then:
   - Increments memory address to use;
   - Decrements the byte count;
   - If byte count is greater than zero:
     - Steps 2 through 4 are repeated;
   - If byte count is zero:
     - DMA controller interrupts the CPU: transfer is now complete.

DMA Transfers

Consider how disk reads occur when DMA is **used** (5/5):

6. When OS starts up:
   - Disk block is already in memory;

Important: Whenever the DMA is using the bus:

- If the CPU also wants the bus, it has to wait;
- This delays CPU operation:
  - Slightly if a small amount of information is transferred;
  - Substantially if a big amount of information is transferred;

Interrupts Revisited

What do you think is the typical interrupt structure? Any ideas?

Interrupts Revisited

Typical interrupt structure:

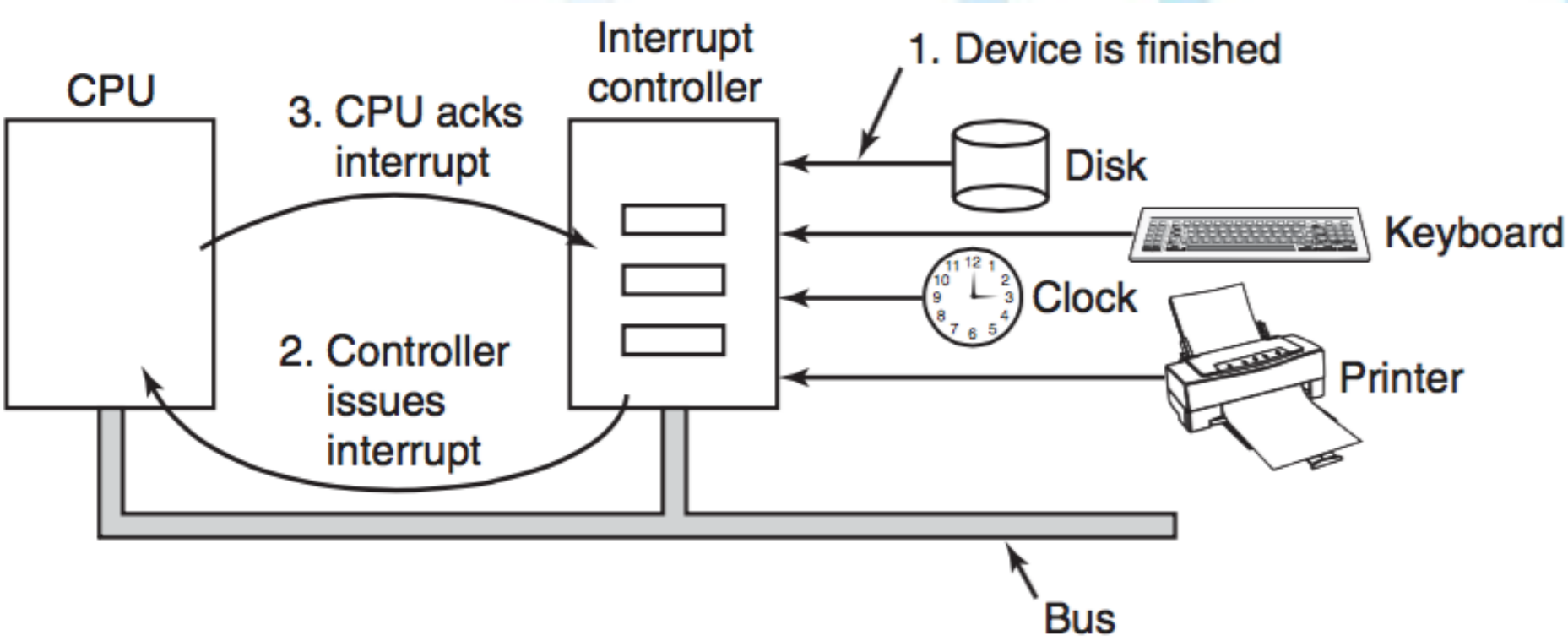

Figure: How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires. (Source: (Tanenbaum and Bos, 2015))

Interrupts work as follows (1/5):

1. Once I/O device finishes work it causes an interrupt;
2. Interrupt is a signal on a bus line;

Interrupts work as follows (2/5):

3. Signal is detected by **interrupt controller chip** on the motherboard:
   - If no other interrupts are pending:
     - interrupt controller handles the interrupt immediately.
   - If another interrupt or higher interruption exists:
     - Device is ignored for the moment;
     - Device continues to assert interrupt signal on the bus;

Interrupts work as follows (3/5):

4. Device controller responsible for generating interruption:
   - Configures address lines with a number specifying the device;

Interrupts work as follows (4/5):

5. Interrupt signal causes the CPU to switch context:
   - Number in address lines indexes **interrupt vector**;
   - Giving the PC of the interrupt-service procedure;
   - Context needs to be:
     - **Saved** before processing interruption;
     - **Restored** after processing interruption;

Interrupts work as follows (5/5):

6. Interrupt-service procedure **acknowledges** interruption:
   - By writing a certain value to one of the **interrupt controller's** I/O ports;
   - Acknowledgement tells:
     - Device controller that it is free to issue another interrupt.

But what happens if we have a pipelined processor? Any ideas?

But what happens if we have a pipelined processor? Any ideas?

First: what is a pipelined processor? Any ideas?

First: what is a pipelined processor? Any ideas?

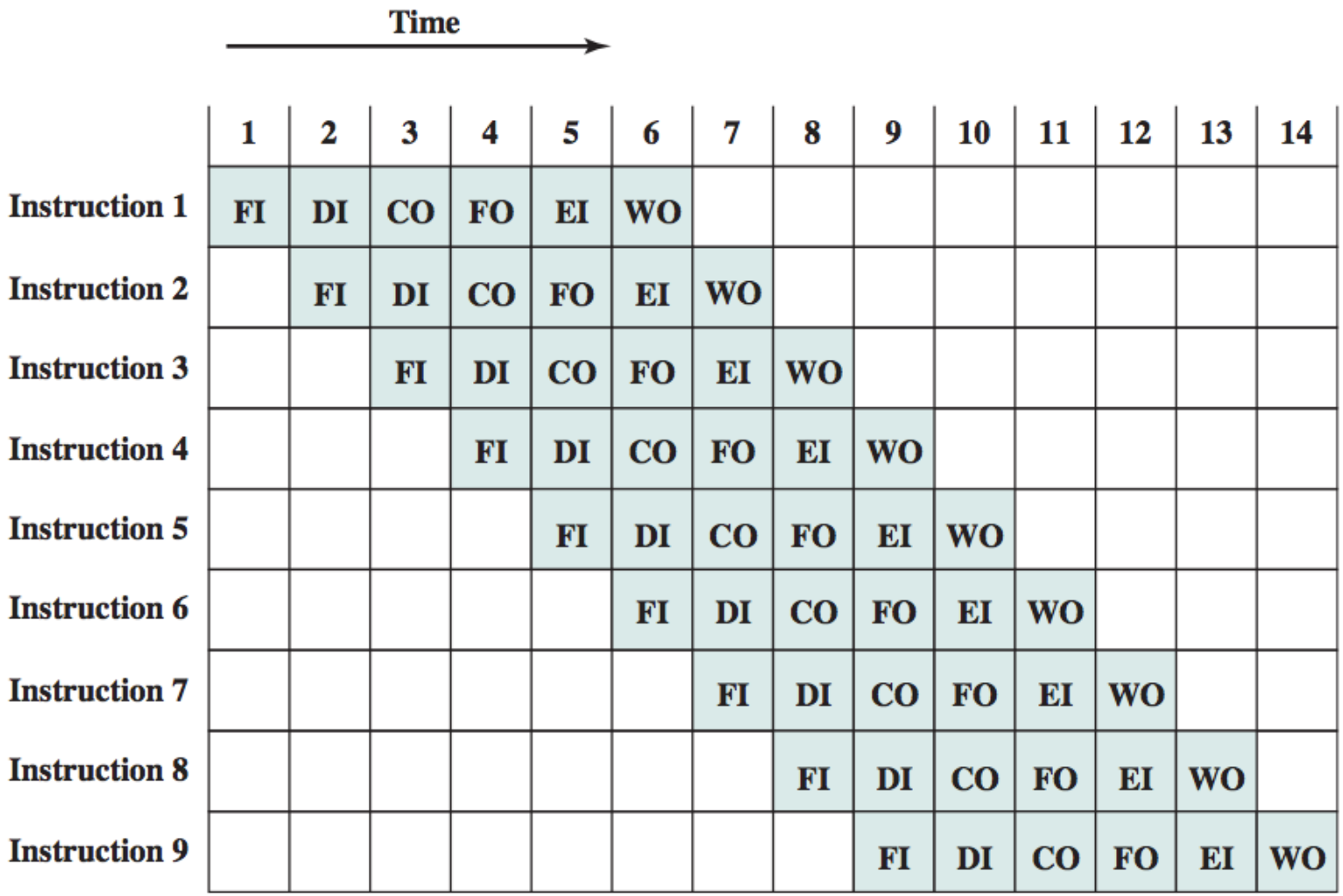

Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

But what happens if we have a pipelined processor? Any ideas?

What happens if an interrupt occurs while the pipeline is full (the usual case)? Any ideas?

What happens if an interrupt occurs while the pipeline is full (the usual case)? Any ideas?

- Many instructions are in various stages of execution;
- When the interrupt occurs:
  - PC may not reflect the correct boundary between:
    - Executed instructions and non-executed instructions

Why do you think this happens? Any ideas?

Why do you think this happens? Any ideas?

- Many instructions may have been partially executed;
- PC most likely reflects address of the next instruction to be:
  - Fetched and pushed into the pipeline;
  - Rather than the address of the instruction that just was processed;

What happens if we have a superscalar processor?

What happens if we have a superscalar processor?

- Things are even worse;
- Instructions may be decomposed into **micro-operations:**
  - μ-operations may execute out of order;
  - Depending on availability of functional units and registers;
- At the time of an interrupt:
  - Some instructions started long ago may not have finished;
  - Others started more recently may be almost done;

This leads to the concept of **precise interrupt:**

- All instructions before the one pointed to by the PC have completed.

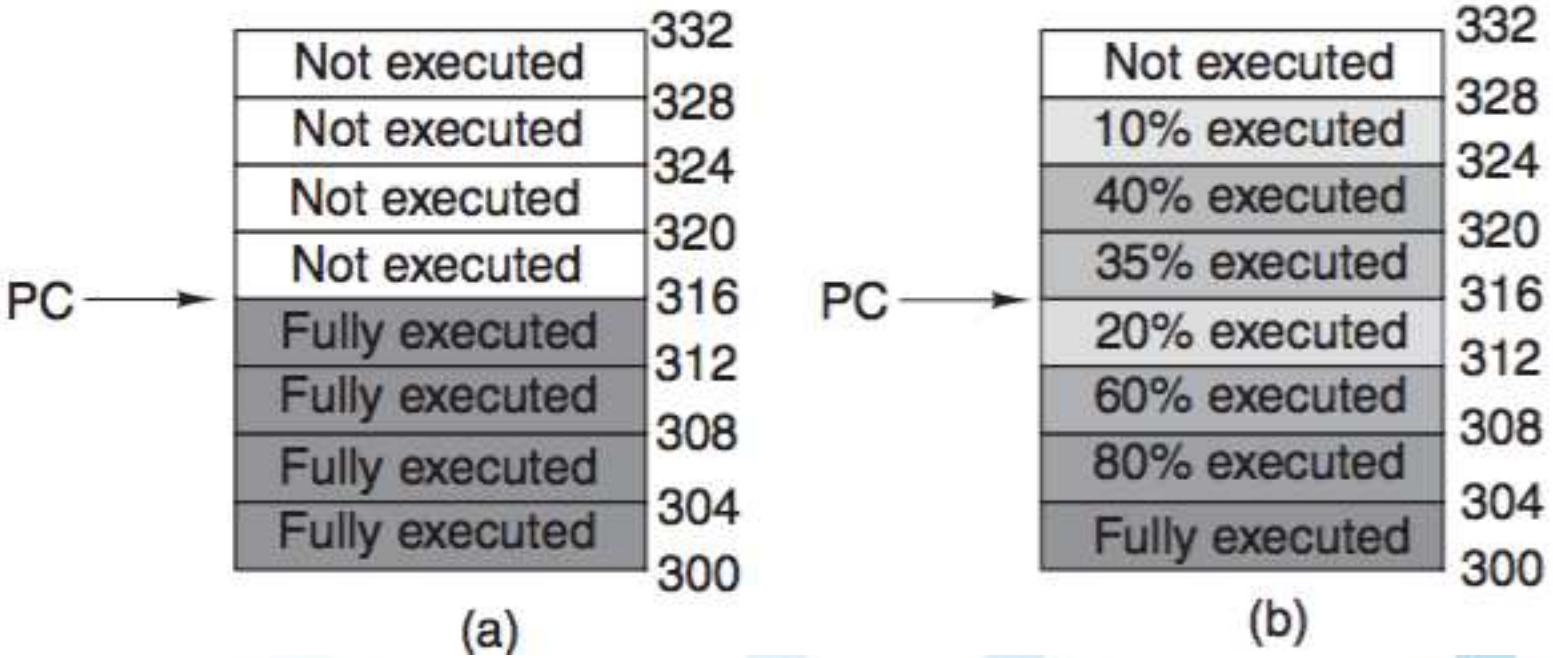

Figure: (a) A precise interrupt. (b) An imprecise interrupt. (Source: (Tanenbaum and Bos, 2015))

imprecise interrupts makes life most unpleasant for the OS writer:

- $\neq$ instructions near the program counter are in $\neq$ stages of completion;
- Machines with imprecise interrupts usually:
  - Vomit a large amount of internal state onto the stack;
  - OS must analyze this to figure out what was going on;
  - Code necessary to restart the machine is typically exceedingly complicated.
- Each interruption saves a lot of information into memory:
  - Memory access implies slower performance;
  - Bad performance for superscalar CPUs;

So what can be done to solve these issues? Any ideas?

So what can be done to solve these issues? Any ideas?

A common answer in engineering: it **depends**

- x86 processor family have precise interrupts:
  - All instructions up to some point are allowed to finish;
  - Before the interruption is processed;
  - Requires extra chip complexity;
- Some processors allow for imprecise interrupts:
  - Making OS far more complicated and slower;

Conclusion: hard to tell which approach is really better.

I/O software layers

I/O software is typically organized in four layers:

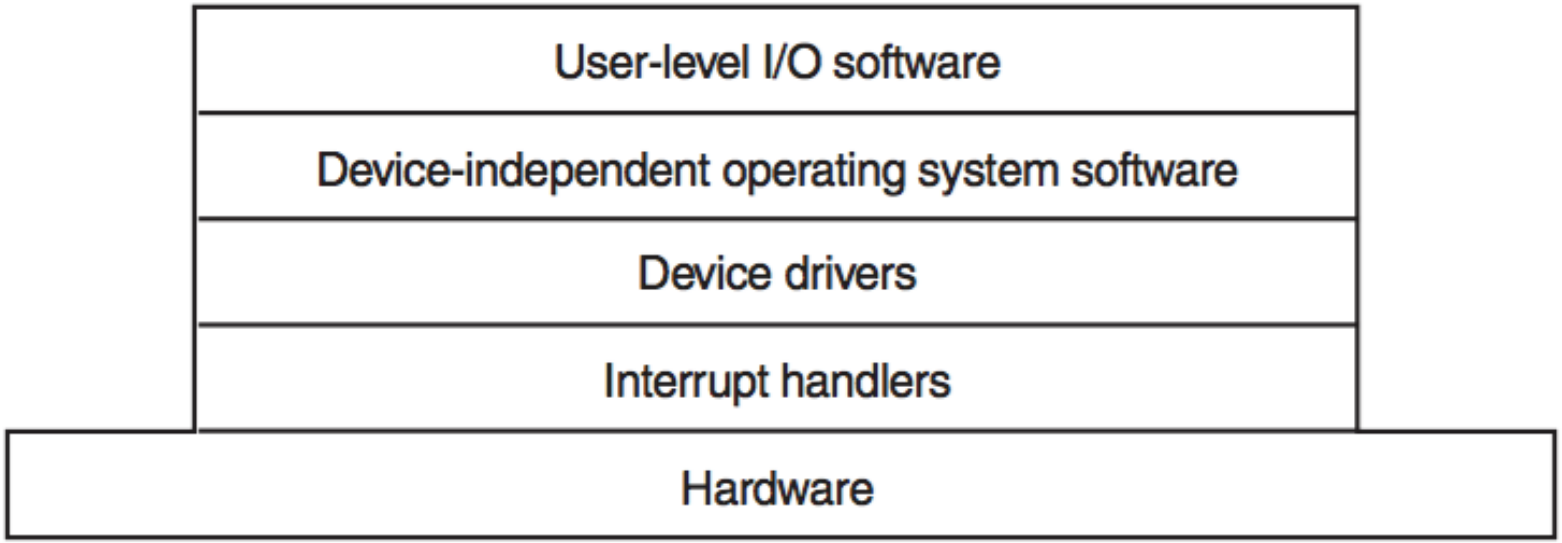

Figure: Layers of the I/O software system. (Source: (Tanenbaum and Bos, 2015))

Lets have a look at each of these =)

Interrupt Handlers

Interrupt process requires the following steps to be performed:

1. Save registers that will be used by handler;
2. Set up a context for the interrupt-service procedure.
3. Set up a stack for the interrupt service-procedure;
4. Acknowledge interrupt controller;
5. Copy saved registers to process table;
6. Run the interrupt-service procedure;
7. Choose which process to run next;
8. Load the new process' registers;
9. Start running the new process.

Device Drivers

Earlier we saw **device controllers:** (1/2)

- Each controller has some **device registers** used to :
  - Give the device commands;
  - Read status;

Device Drivers

Earlier we saw **device controllers:** (2/2)

- Number of registers and commands vary from device to device, *e.g.:*
  - Mouse driver has to accept information from the mouse, *e.g.:*:
    - How far it has moved
    - Which buttons are pressed;
  - Disk driver has to know all about:
    - Sectors, tracks, cylinders, heads, arm motion, etc...
- Obviously, these drivers will be very different.

Each I/O device needs some device-specific code, *i.e.*, **device driver**:

- Usually written by the device's manufacturer;
- Different for each OS;
- Each device driver normally handles one device type:
  - Mouse driver;
  - Joystick driver;
- Each device driver can also handle closely related devices:
  - SCSI disk driver can usually handle multiple SCSI disks

Different devices can also be based on the same underlying technology:

- Example: **Universal serial bus:**
  - Disks, memory sticks, cameras, mice, keyboards, etc...
  - There is a reason why it called **universal**
- In order to access the device's hardware (*i.e.* data / control registers):
  - Device driver normally has to be part of **kernel**;
  - Device drivers are normally positioned below the rest of OS;

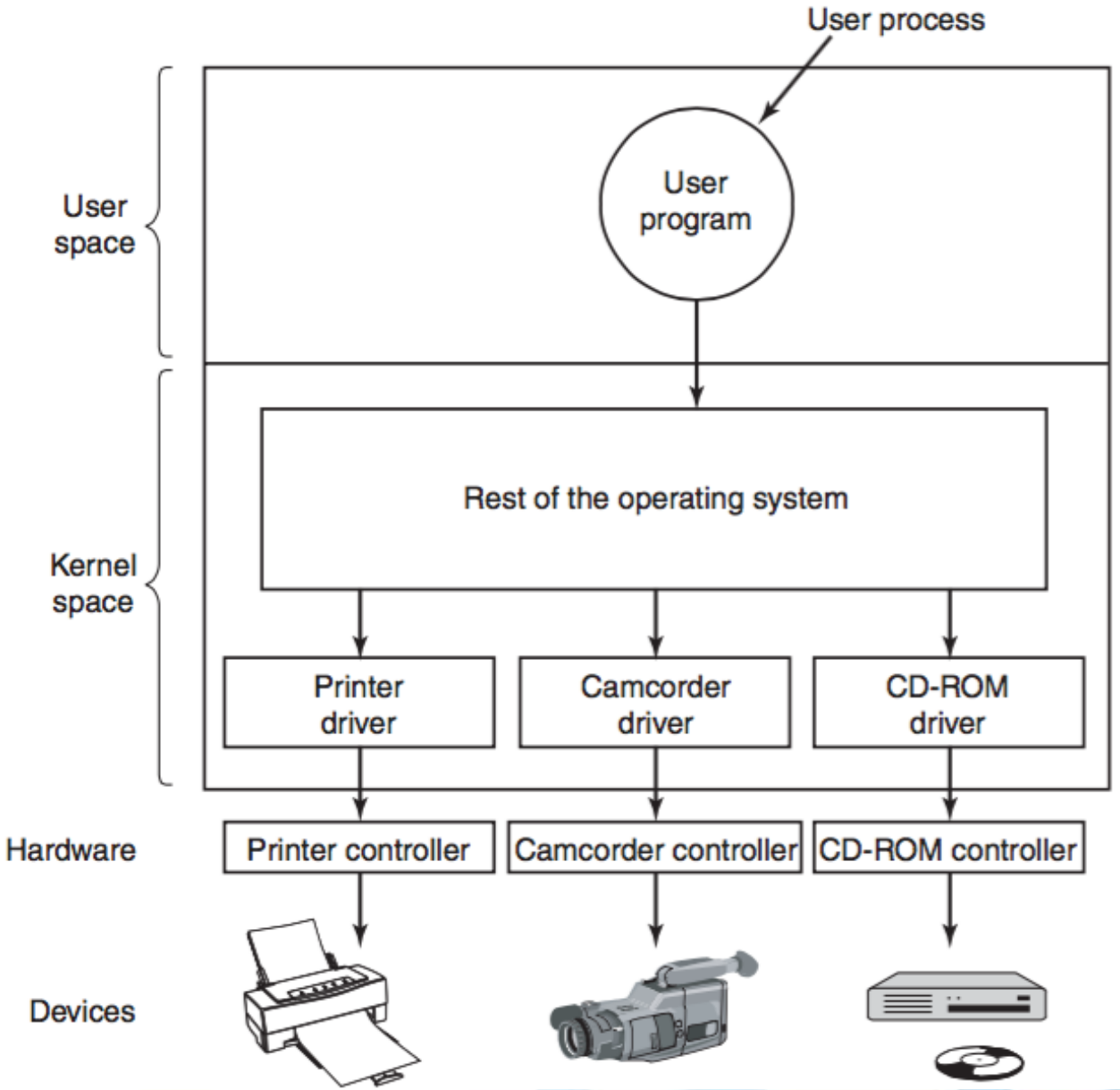

Figure: Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus. (Source: (Tanenbaum and Bos, 2015))

Remember that OS usually classify drivers into:

- **Block devices:** containing multiple data blocks:
  - Each block can be addressed independently;
  - OS define a standard interface that:
    - Block drivers must support, *e.g.:* read a block;
- **Character devices:** accepting a stream of characters:
  - Such as keyboards and printers;
  - OS define a standard interface that:
    - Character drivers must support, *e.g.:* write string;

In your opinion:

What are the set of responsibilities of a device driver? Any ideas?

What are the set of responsibilities of a device driver? Any ideas?

Device driver **functions:**

- Minimum set of functions:
  - Accept read / write requests;
  - See that read / write requests are performed;
- Additional set of possible functions:
  - Initialize device;
  - Manage power requirements;
  - Log events;

In your opinion:

What is the general structure of a device driver? Any ideas?

Device driver general structure (1/2):

1. Confirm input parameters are valid:
   - If not return an error;
2. Check if the device is currently in use:
   - If device is **busy:**
     - Queue request for later processing;
   - If device is **idle:**
     - Check if device can handle request...
     - ...may be necessary to: switch device, start motor, etc..
     - ...Proceed to process request

Device driver general structure (2/2):

3. Commands are written into the controller's device registers;
4. After each command is written to the controller:
   - May be necessary to check to see if:
     - Controller accepted the command and...
     - ...is prepared to accept the next one.
   - Sequence continues until all the commands have been issued;

5. After the commands have been issued:
   - **Situation 1:** driver waits until controller finishes work;
     - Driver **blocks**;
     - Can be awakened by an interruption;
   - **Situation 2:** operation finishes without delay:
     - No need for the driver to block;

6. After operation completes: driver checks for **errors**
   - If no error: data is returned to original requesting application;
7. If any other requests are queued:
   - They can now be selected and started.s
8. If nothing is queued:
   - Driver blocks waiting for the next request.

Device-Independent I/O Software

Although some I/O software is device specific:

- Other parts of it are device independent.
- The following functions are typically device-independent:
  - Uniform interfacing for device drivers;
  - Buffering;
  - Error reporting;
  - Allocating and releasing dedicated devices;
  - Providing a device-independent block size;

Lets have a look at each one of these...

Uniform Interfacing for Device Drivers

How can we make all I/O devices and drivers look more or less the same?
Any ideas?

Uniform Interfacing for Device Drivers

How can we make all I/O devices and drivers look more or less the same? Any ideas?

All drivers have the same interface:

- Easy to plug in a new driver:
  - Driver just needs to implements methods specified in interface;
  - Driver writers know what is expected of them;
  - In practice: not all devices are absolutely identical:
    - But usually there are only a small number of device types...
    - ...and even these are generally almost the same.

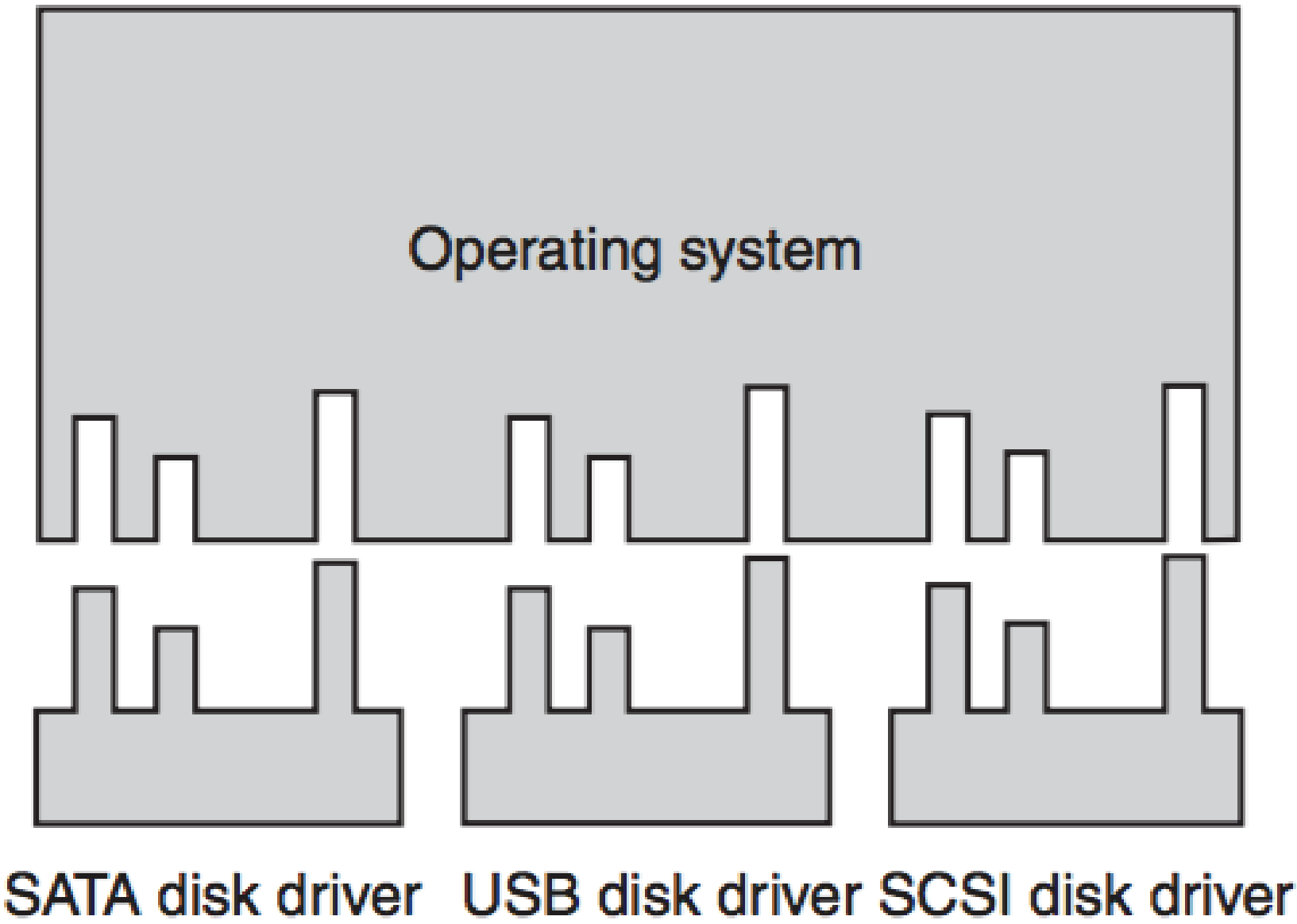

Figure: Standard driver interface.(Source: (Tanenbaum and Bos, 2015))

Usual methods specified by the interface:

- Read;
- Write;
- Turn power on / off;
- Formatting;

Buffering

How to buffer data from / into the device? Any ideas?

Buffering

How to buffer data from the device? Any ideas?

- Read/Write one item of information?
  - Device driver process is executed once for each item;
  - Not very efficient;
- Read / Write multiple items of information?
  - Device driver process is executed once for multiple items;
  - More efficient;
- Read/Write data into a kernel register?
  - Copy data to device driver when full;

Error reporting

Many errors are device specific:

- Handled by the appropriate driver;
- What software does depends on environment and error nature;
- Option include:
  - Asking user what to do;
  - Retry a certain number of times;
  - Ignore the error;
  - Killing the calling process;
  - Have system call return with an error code;

However, some errors cannot be handled this way:

- If error involves critical data:
  - System may have to display an error message and terminate:
    - Not much else it can do.

Allocating and Releasing Dedicated Devices

OS examines requests for device usage and accepts or rejects them:

- Depending on whether the requested device is available or not;
- If request is authorized OS must:
  - Allocate device during request;
  - Free allocated resource (including device) after request is processed;

Device-Independent Block Size

E.g.: Different disks may have different sector sizes:

- Up to the device-independent software to hide this fact:
  - Providing a uniform block size to higher layers,
- Several sectors as a single logical block;
- This way higher layers all use the same block size:

User-Space I/O Software

Summary of the I/O system:

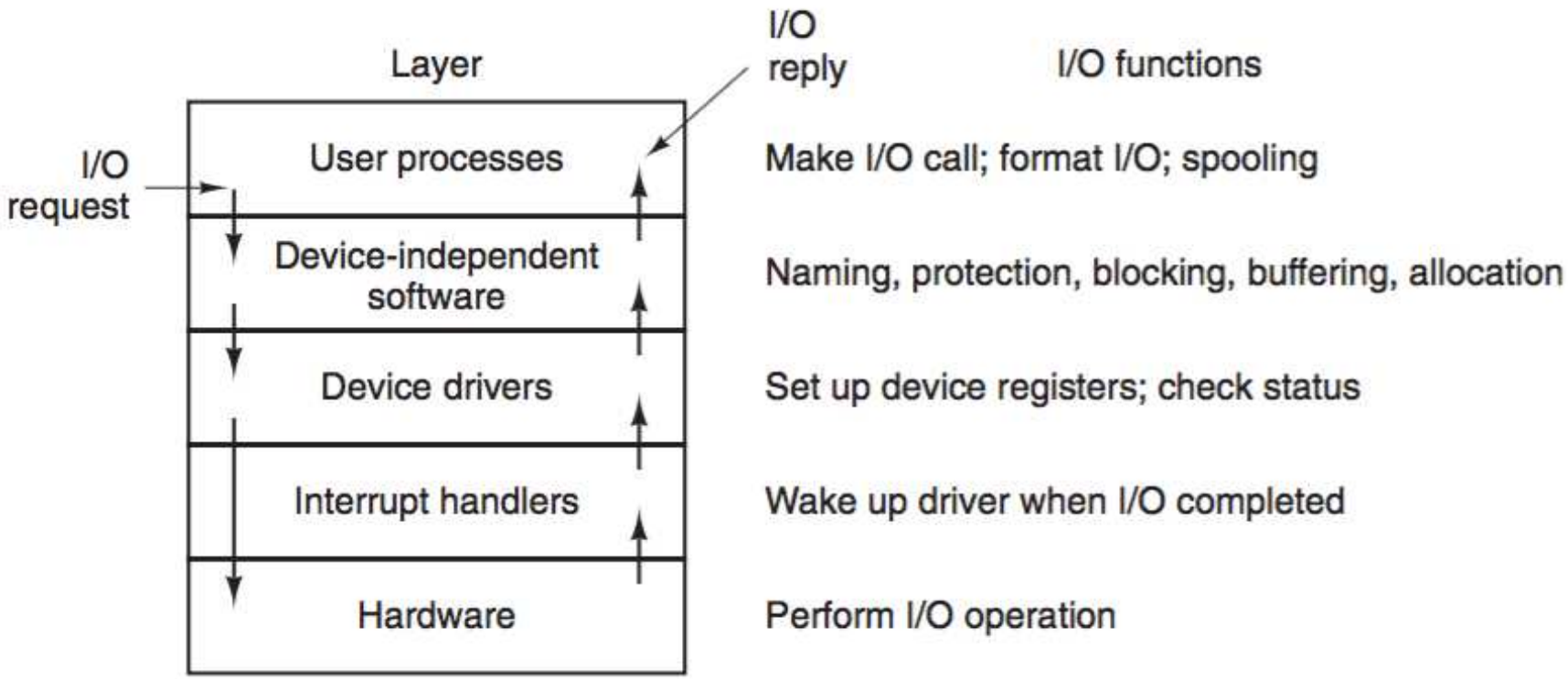

Figure: Layers of the I/O system and the main functions of each layer. (Source: (Tanenbaum and Bos, 2015))

Example:

1. **User program** tries to read a block from a file;
2. **Device-independent software** looks for it in buffer cache:
3. If the needed block is not there: **device driver** issues request to **hardware**;
4. **Hardware** fetches block from disk;
5. Process is then **blocked** until disk operation finished;
6. When disk finishes: **interruption** is generated;
7. **Interruption handler** runs and notifies sleeping process;

Clocks

Clocks (also called timers) are essential to OS:

- Maintain the time of day;
- Prevent one process from monopolizing the CPU;
- Among other things;

Clock software can take the form of a device driver:

- Even though a clock is neither a block device nor a character device;
- Lets look first at the clock hardware and then the software;

Clock Hardware

A clock is built out of three components:

- **Crystal oscillator:**
  - When piece of quartz crystal is properly cut and mounted under tension:
    - Generate a periodic signal of very great accuracy;
    - Synchronizing signal to the computer's various circuits.
- **Counter:**
  - Clock signal is fed into the counter to make it count down to zero;
  - When counter gets to zero, it causes a CPU interrupt.
- **Holding register:** used to load the counter;

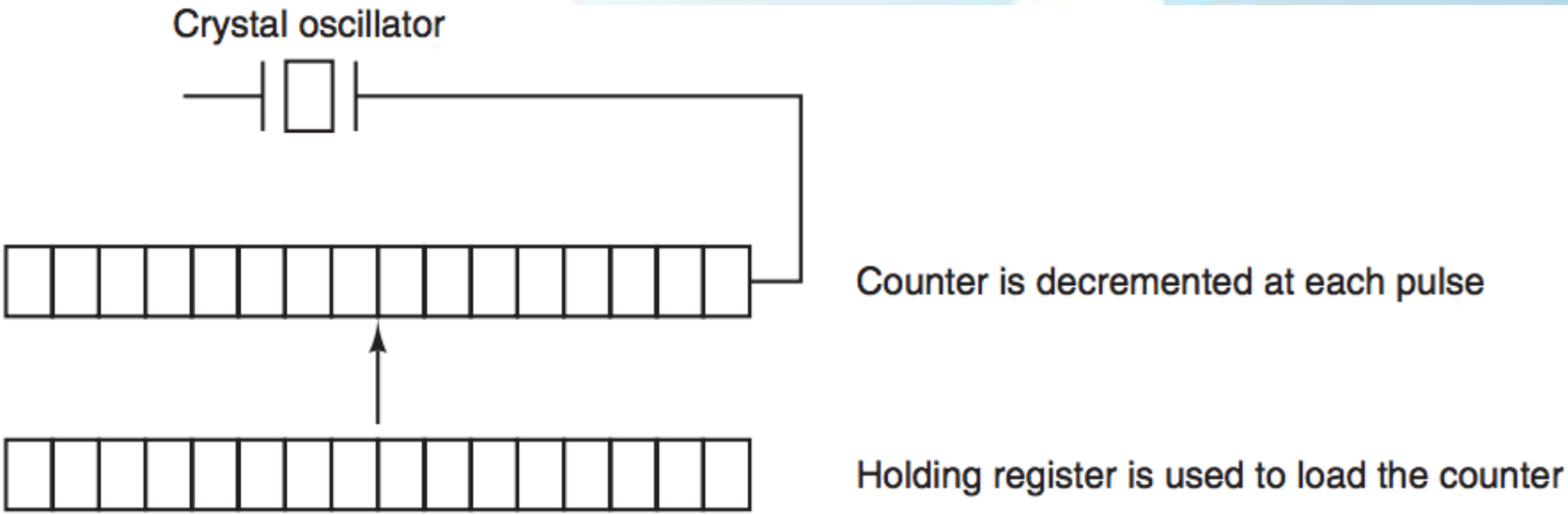

Figure: A programmable clock. (Source: (Tanenbaum and Bos, 2015))

Two working modes:

- **One-shot mode:**
  - When clock is started it copies holding register value to counter;
  - Counter is decremented at each pulse;
  - When counter gets to zero:
    - Interruption is activated;
    - Stops until explicitly started again;
- **Square-wave (periodic) mode:**
  - After getting to zero and causing interruption:
    - holding register is automatically copied into the counter;
    - process is repeated again indefinitely.

Programmable clock chips usually contain:

- Two or three independently programmable clocks;
- Many other options:
  - counting up;
  - counting down;
  - interrupts disabled;
  - and more;

Clock Software

Clock hardware is responsible for generating interrupts at known intervals:

- Everything else must be done by the **clock driver**;
- Exact duties usually include:
  1. Maintaining the time of day;
  2. Preventing processes from running longer than they are allowed to;
  3. Accounting for CPU usage;
  4. Handling the alarm system call made by user processes;
  5. Providing watchdog timers for parts of the system itself;
  6. Doing profiling, monitoring, and statistics gathering.

References I

Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.

Tanenbaum, A. and Bos, H. (2015).

Modern Operating Systems.

Pearson Education Limited.