Operating Systems

Elvis C. Foster

Lecture 09: Input/Output Management

Despite all the considerations that have discussed so far, the work of an operating system can be summarized in two main activities — input/output (I/O) and processing. In some respects, the main job is really I/O, and processing is merely incidental. People (end users) do not use computers to define computing problems, but to access or specify information. Processing is often required in order to facilitate the end user, but in many cases, their primary focus is not processing.

In this lecture, we will examine how the operating system bridges the gap between end users focus on I/O and the necessary internal processing. The lecture proceeds under the following captions:

- Overview
- Organization of the I/O Function
- Nature of I/O Calls
- I/O Strategies
- Buffering, Caching, and Spooling
- Other Issues
- Summary and Concluding Remarks

9.1 Overview

How does the operating system recognize and communicate with the various types of I/O devices that exist? To complicate matters, these devices all operate at differing speeds, and new ones continue to be introduced.

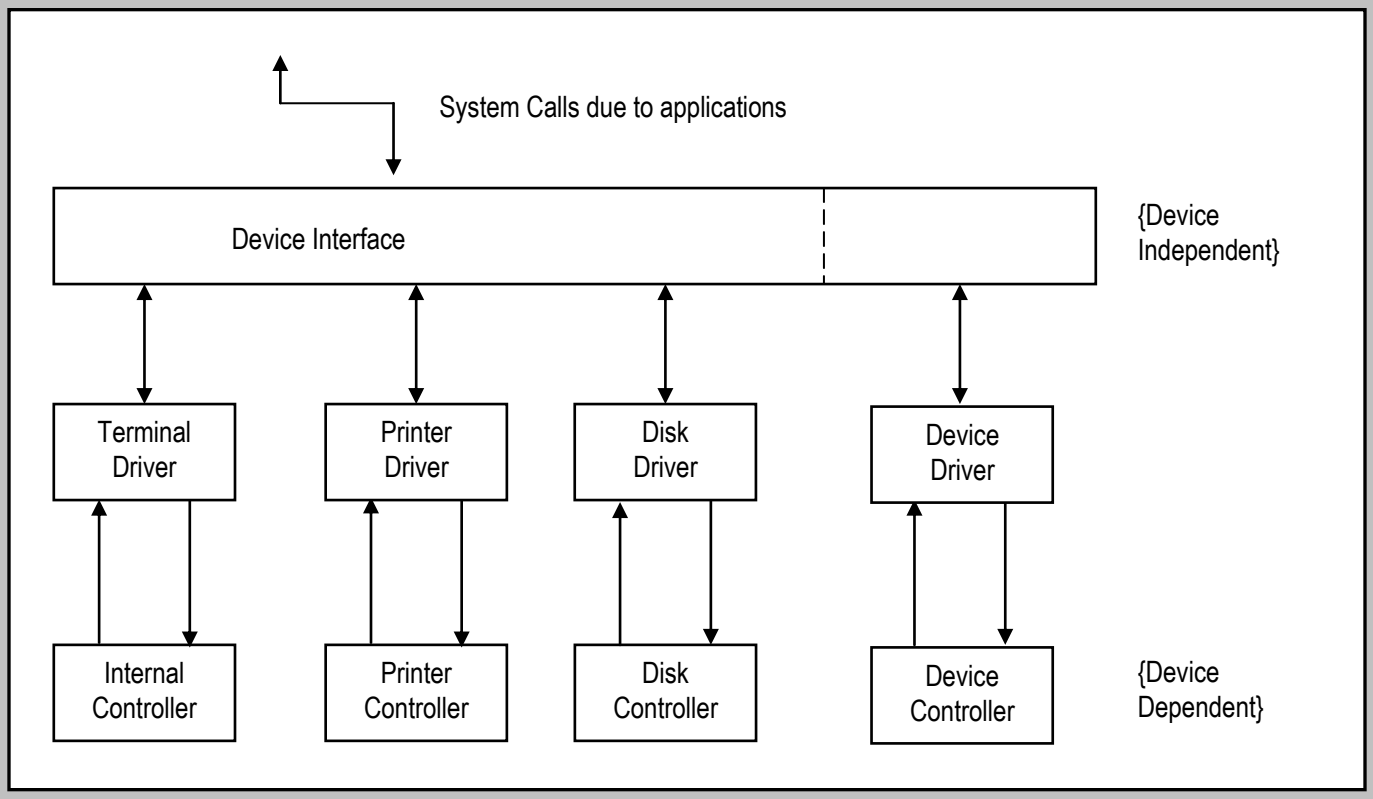
To address this problem, the kernel of an operating system is structured to use *device-driver* modules. Device drivers provide a uniform device-access interface to the I/O subsystem in the same way that system calls provide a standard interface between the operating system and executing applications.

Device drivers are programs carefully written to communicate at two levels:
- The device controllers (hardware level)
- The device interface of the I/O subsystem (software level).

Figure 9.1 Illustrates this. At the hardware level are the device controllers; these are device dependent. At the software level are the device drivers (one per device). These are written to communicate with device controllers at the hardware level, and the device interface (which is device independent).

This infrastructure allows the OS to treat common device interface calls as systems calls. These calls are then routed on the generic interface, to specific device driver programs. The *device manager* is a special program, which facilitates communication between the application program(s) and the I/O interfaces.

Figure 9.1: Device Manager Infrastructure



9.2 Organization of the I/O Function

There are three approaches (techniques) for performing I/O operations:
- Programmed I/O
- Interrupt-driven I/O
- Direct Memory Access

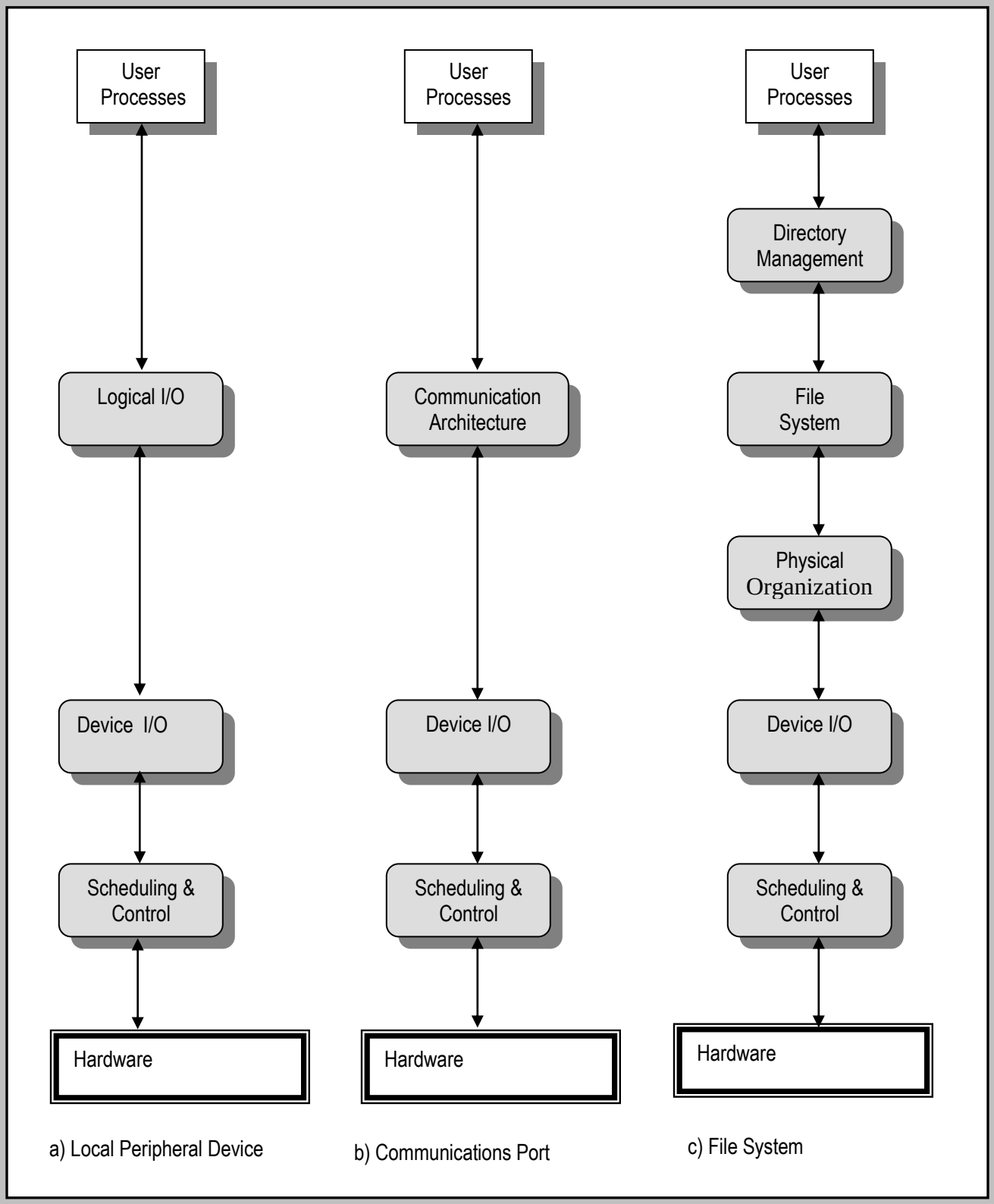
Programmed I/O: The processor issues an I/O command on the behalf of an executing process. That process then goes in a state of *busy-waiting* — it awaits completion of the I/O operation before proceeding.

Interrupt-Driver I/O: The processor issues on I/O instruction on behalf of the executing process, and move on to execute other instructions (within that executing process or in another process). When the I/O request has been serviced, the processor is interrupted by the I/O module.

Direct Memory Access (DMA): The I/O module controls the exchange of data between itself and main memory. This is facilitated by a DMA component, which may be a component of the I/O module, or an independent component to which the I/O module is connected.

Like the operating system itself, the I/O functions are structured hierarchically as illustrated in figure 9.2.

Figure 9.2: Organization of I/O Function



9.3 Nature of I/O Calls

There are two types of I/O calls that an operating system has to address:
- Blocking Calls
- Non-Blocking Calls

Blocking Calls: When a process issues a blocking I/O call, it makes an I/O request that must be serviced before the process can continue. An example of this is reading a file into an array, for subsequent processing. When a process issues a blocking I/O call, it is suspended and goes into a state of waiting (review lecture 4). When the I/O request has been serviced, the process is moved back into a state of readiness for CPU attention.

Non-blocking Calls: When a process issues a non-blocking I/O call, it makes a I/O request which can be serviced while it is carrying out some other processing. An example of this is a user interface that receives keyboard and mouse input, while processing and displaying other information on the screen. Another example is an application which has (was written with) multiple threads.

9.4 I/O Strategies

Four strategies for I/O implementation have been proposed:
- Direct I/O with polling
- Direct I/O with interrupts
- Direct Memory Access I/O with polling
- Direct Memory Access I/O with interrupts

Irrespective of the strategies employed, the device manager is an important component of the operating system, which facilitates communication with the I/O devices. Among the important functions of the device manager are the following:
- Facilitation of device independence through the device interface (review figure 1)
- Facilitation of communication with application programs
- Servicing of interrupts caused by I/O requests from executing programs

9.4.1 Direct I/O with Polling

In this approach, all I/O transfers between the I/O device and memory occur via the CPU (see figure 9.3). The CPU determines when the I/O operation has completed, and then transfers the data between the I/O device and the primary storage. Figure 9.4 outlines the steps involved in input as well as output.

Figure 9.3: Illustrating Direct I/O

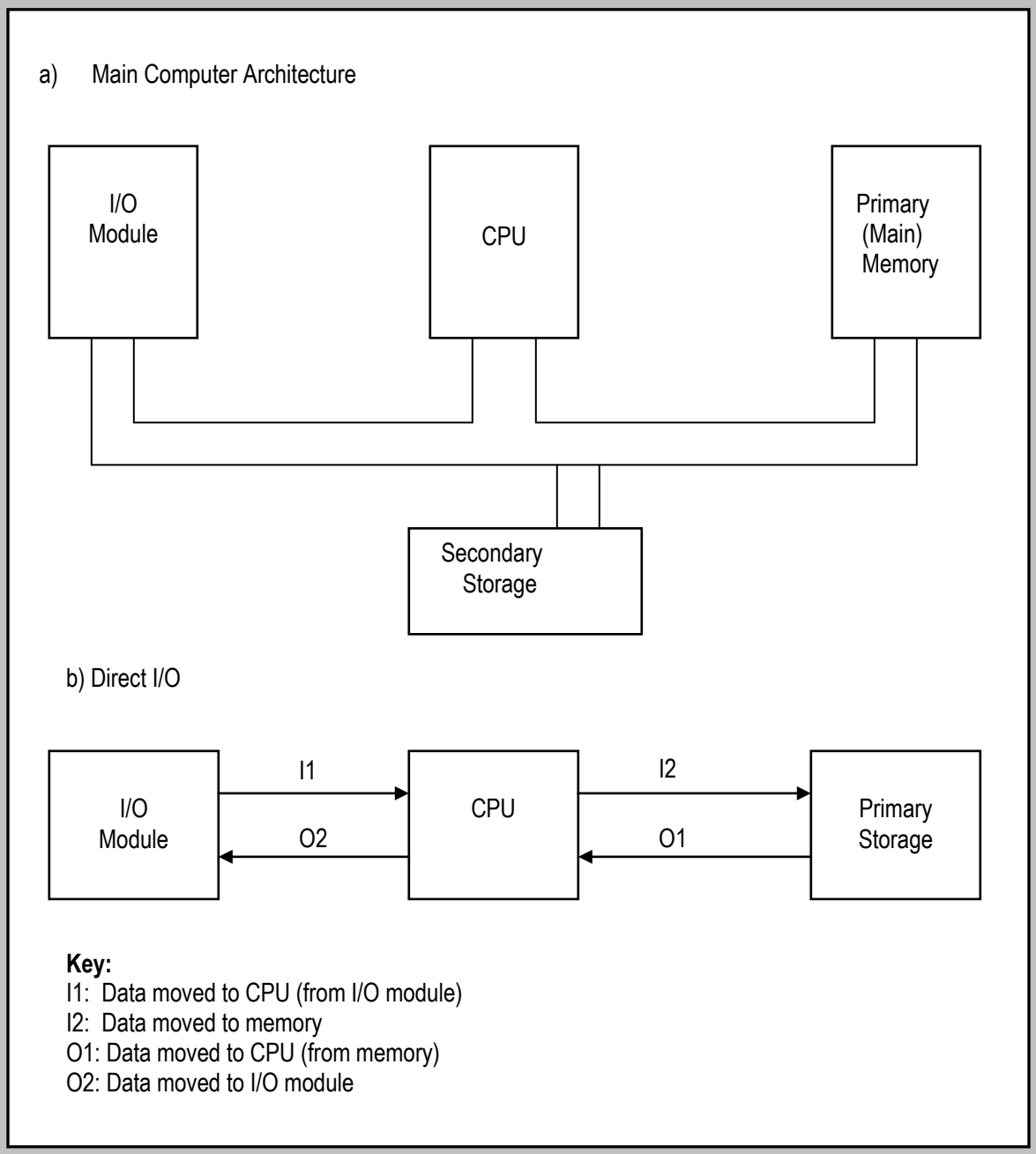


Figure 9.4: Steps in Direct I/O with Polling

Input:
1. Process requests a read operation.
2. **Device driver** checks the status register to ensure that the device is idle (if it is busy, the driver waits until it becomes idle).
3. **Device driver** puts input command into the controller's **command register**, thereby starting the device.
4. **Device driver** repeatedly checks (polls) the device **status register** to get the idle signal (which would mean read operation is complete)
5. When the read operation is complete (indicated by idle signal), the **device driver** copies the controller's data register into the **user process area**.

Output:
1. Process requests a write operation
2. **Device driver** checks the status register to ensure that the device is idle (if it is busy, the driver waits until it becomes idle).
3. **Device driver** copies data from the user process area to the data register(s).
4. Drive driver puts an output command into the controller's **command register**, thereby starting the device.
5. When the write operation is complete (indicated by idle signal), the **device driver** copies the **user process area** to the controller's data register.

Note: I/O components (hardware as well as software) are highlighted.

9.4.2 Direct I/O with Interrupts

The motivation for introducing interrupts in I/O devices is to eliminate the need for the device driver software to constantly poll the controller status register. Instead, the device controller automatically notifies the device manager when the I/O operation has completed. The illustration of Figure 9.3 would still be relevant, but the steps involved would be slightly different, as outlined in figure 9.5

Figure 9.5 Steps: Direct I/O with Interrupt

Input:
1. Process requests a read operation.
2. **Device driver** checks the status register to determine that the device is idle (if it is busy, the driver waits until it becomes idle).
3. **Device driver** puts input command into the controller's **command register**, thereby starting the device.
4. **Device driver** puts an entry in the **device status table**. [This table contains an entry for each device in the system.]
5. Device eventually completes and interrupts the CPU.
6. **Interrupt handler** determines which device interrupted the CPU. It then branches to the device handler for that device.
7. **Device handler** retrieves the pending I/O status from the **device status table**.
8. The **device handler** copies the content of the controller's data register to the user process area.
9. **Device handler** returns control to the application process.

Note: Output would be similar. I/O components (hardware as well as software) are highlighted.

9.4.3 Direct Memory Access will Polling or Interrupt

The motivation behind direct memory access (DMA) is to direct traffic away from the CPU so that it can concentrate on other essential functions that it must carry out. The overall effect is increased efficiency of the system, but not without a cost: the I/O modules now need an intelligent component — a DMA module — to access memory directly. Figure 9.6 provides an illustration.

The I/O device may still employ either polling or interrupt. With polling, the device manager polls the device controller to determine whether transfer between memory and the I/O device is required. With interrupt, the device controller interrupts the device manager whenever I/O transfer between memory and the device is required. Obviously, the optimum (ideal) situation would be DMA I/O with interrupt.

As mentioned earlier, there may be different configurations for the DMA and I/O units. Figure 9.7 provides three possible configurations.

Figure 9.6 Illustrating DMA

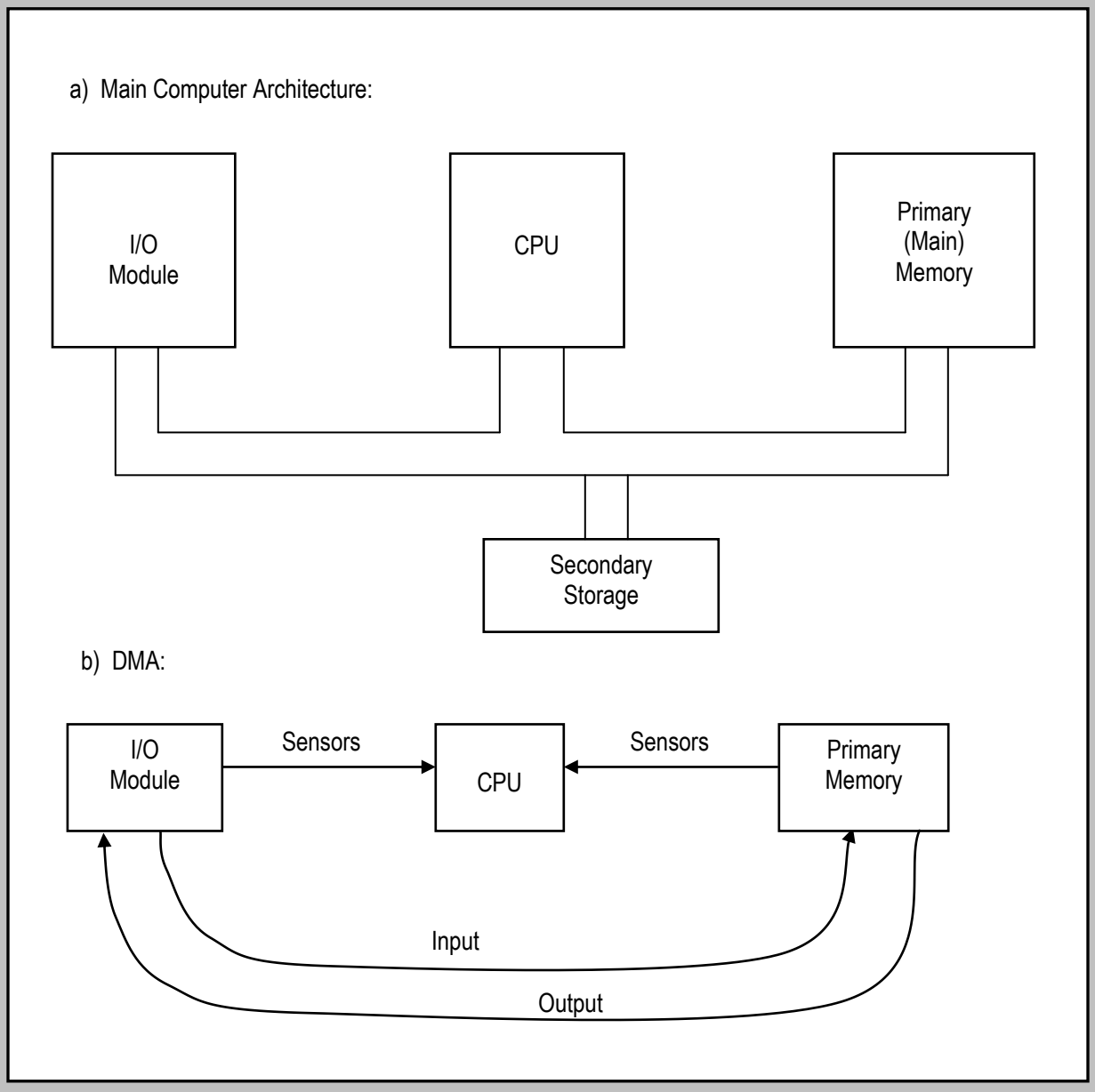
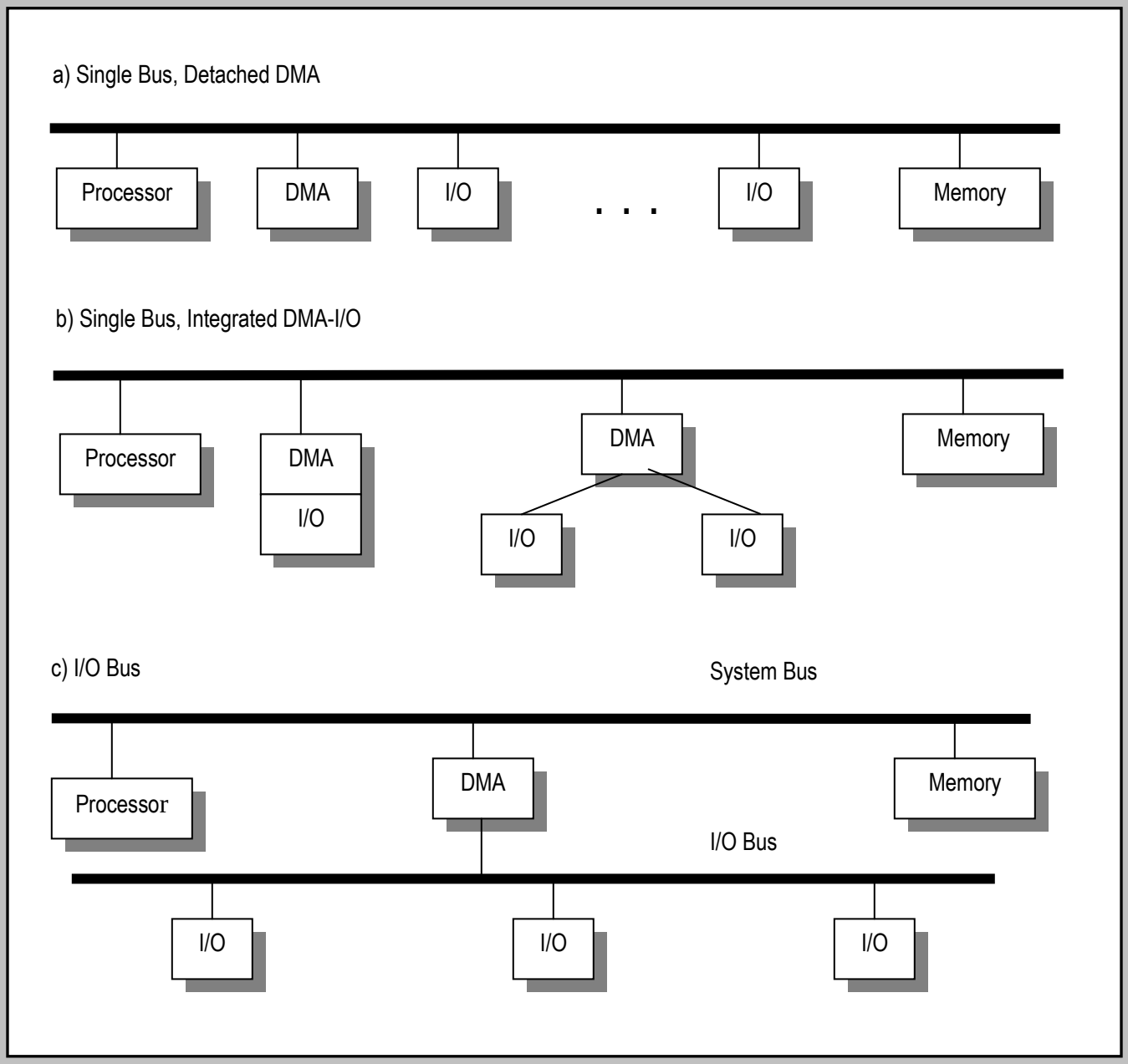


Figure 9.7: Alternate DMA Configurations



9.5 Buffering, Caching and Spooling

Buffering

A buffer is a memory holding area that holds temporary data being transferred between two devices, or between a device and an application. This is necessary because devices and/or components operate at different speeds; some are much faster than others. Without buffering, the faster components/devices are made to waste time, while waiting on the slower components/devices.

Buffering can be done on input as well as output.
- **On Input**: the process obtains input data from an input buffer, which is fed by the I/O device.
- **On Output:** The process writes its output to a buffer, which is accessed by the I/O device.

Caching

A cache is a region of fast memory that holds copies of data. Access to the cache copy is much more efficient than access to the original (review lecture 5).

In a computer, the cache size must be carefully chosen. If it is too large, this renders the hardware too expensive; if it is too small, it becomes counter-productive.

Spooling

A spool is a special buffer that holds output for a device (such as a printer) that cannot accept interleaved data streams. Although a printer can handle only one job at a time, several applications may wish to print information at the same time. If processes had to line up and wait directly, on a printer, one of two problems would be certain: system bottlenecks and system deadlocks (review lecture 7).

To avoid both problems, printers are assigned to spool files. On systems such as Windows, there is one spool file per device. On other systems such as System i (formerly OS-400) and Unix, there may be several output queries, which may or may not be directly connected to output devices. This strategy provides more flexibility: A user could direct output to an output queue that is not connected to a printer. The output could remain there until the user is read to have it printed, at which point, it is simple moved to another output queue that is connected to a printer.

9.6 Other Issues

Other issues relating to I/O are typically treated by the filing system of the operation system. These include:

- Input and output streams
- Disk scheduling
- Data compression
- Data encryption
- Exemption handling
- System security
- Backup and recovery

Some of these issues have already been discussed in earlier lectures. Issues such as file streams, disk scheduling and data compression will not be addressed in this course, as they vary significantly, depending on the software environment. Suffice it to say that I/O management will continue to be a very important aspect of operating system design.

9.7 Summary and Concluding Remarks

Here is a summary of what has been covered in this lecture:

The OS manages I/O at two levels:

- At the hardware level, it manages the communication between device controllers and their related devices.
- At the software level, it manages the synchronism and translation between user applications and the device drivers. Device drivers communicate with device controllers as well as the device interface, which is device independent.

There are three approaches for performing I/O in the operating system:

- Programmed I/O — caused by executing programs
- Interrupt-driven I/O — initiated by the Os without user intervention
- Direct Memory Access — caused by I/O components that can access memory directly without CPU intervention.

There are two types of I/O calls that an operating system has to address:

- Blocking Calls — these tie up a resource until the process is finished with it.
- Non-Blocking Calls — these do not tie up resources and are processed at the convenience of the OS, but in a timely manner.

Communication with the I/O devices and servicing of I/O requests are managed by the OS component called the device manager.

9.7 Summary and Concluding Remarks (continued)

Four strategies for I/O implementation have been used in OD design are:

- Direct I/O with polling — the device driver constantly polls the device controller to determine when I/O is required.
- Direct I/O with interrupts — the device controller interrupts the device manager as to when I/O is needed or completed.
- Direct Memory Access I/O with polling — the device manager polls the device controller to see if I/O transfer between the device and memory is required.
- Direct Memory Access I/O with interrupts — the device controller interrupts the device manager whenever I/O transfer between memory and the device is required.

Buffering, spooling, and caching are strategies used to increase the efficiency of the system as the valuable resource of the CPU is shared among competing jobs.

Like object management, directory system, and security, I/O management is typically addressed in the filing system for the operating system. However, in the interest of clarity, and due to its importance, it was treated separately in the course. Our next lecture deals with concurrent processing.

9.8 References and/or Recommended Reading

[Bacon & Harris 2003] Davis, William S. & T. M. Rajkumar. 2005. *Operating Systems: A Systematic View*, 6th Ed. Boston: Addison-Wesley. See Chapters 4 & 5.

[Nutt 2004] Nutt, Gary. 2004. *Operating Systems: A Modern Perspective* 3rd ed. Boston: Addison-Wesley. See Chapter 5.

[Silberschatz 2012] Silberschatz, Abraham, Peter B. Galvin, & Greg Gagne. 2012. *Operating Systems Concepts*, 9th Ed. Update. New York: John Wiley & Sons. See Chapter 13.

[Stallings 2005] Stallings, William. 2005. *Operating Systems* 5th ed. Upper Saddle River, New Jersey: Prentice Hall. See Chapter 11.