Chapter 11
Input/Output Management

This could be the messiest aspect of an operating system. There are just too much stuff involved, it is difficult to develop a uniform and consistent theory to cover them all. For example, those devices could be either human readable, machine readable, or for communication purposes.

A human readable device is used for the machine to communicate with users. Examples include printers, terminals, and wii remote controls, etc. A machine readable device is used for communication between electronic equipment. Examples are disk and tape drives, sensors, etc.. A communication device is used to communicate with remote devices. Examples include various wired or wireless modems.
What is the difference?

Depending on its category, these devices behave very differently, in terms of data rate (bps), unit of transfer (stream of bytes or blocks of characters), data representation (ASCII, Unicode, or encoded bits), where it can be used, complexity of control, etc..

For example, if a disk is used to keep data files, we have to provide file management software; on the other hand, if it is mainly used for backing up pages associated with a virtual memory system, we have to back it up with virtual memory related hardware and software.
How to do an I/O?

There can be a few ways to do an I/O, including: Programmed I/O, Interrupt-driven I/O and Direct memory access.

With a programmed I/O approach, the processor sends an I/O command for a process to an I/O module. The process is then put into a blocked state, waiting for the operation to be completed, when it is put back into the ready state, and resume its operation later.
In an Interrupt-driven I/O, the processor sends an I/O command for a process, and then continues subsequent instructions, until interrupted by an I/O module when the data transmission is done. Depending on whether the requesting process has to wait for the completion of that I/O request, it may be suspended and the processor will then switch to other processes.

Finally, with a direct memory access approach, a DMA module controls the exchange of data between the memory and an I/O module. The processor sends a request to a DMA module with specific information and will be interrupted when the whole block is transferred.
How do we get here?

When computer systems have progressed, so is the I/O function, in the following pattern:

1. Initially, the processor directly controls a peripheral device, just like in today’s TV set.

2. Then, a controller or I/O module is added. The processor uses programmed I/O without interrupts. This way, the processor is partly cut from the details of the external devices, just like a general purpose processor should.

3. The interrupt mechanism is added. Now, the processor needs not wait for the completion of the I/O operation, thus its efficiency goes up.
4. The I/O module is given direct control over the memory via the DMA mechanism. The processor is only involved at the beginning and the end of the data block movement.

5. The I/O module is enhanced to become a separate processor, with a specialized, I/O tailored, instruction set. It is directed by the CPU to carry out various I/O related instructions, and will only involve the CPU when it has to be initiated.

6. The I/O develops into a separate computer with its own local memory to manage a large set of I/O operations with minimum CPU involvement. It is not a simple end of a loop, but rather the beginning of another layer up.

I would imagine IRS uses a machine or machines to do the preprocessing and at the end, uses a couple to do the refunding (direct depositing).
More about DMA

Below shows the internal logic of a typical DMA module. It has to take the control from the processor when transferring data between memory and the CPU by forcing the processor to suspend its operation.
How does DMA work?

When the processor needs to read or write a block of data, it sends out a command to the DMA module with the following information: The nature of this operation, i.e., a *read* or a *write*, using the R/W control lines; the address of the I/O device involved, using the data line; the *starting location* of the data block it tried to read or write, using the address register; together with the number of words to be read or written, stored in the data count register.

The processor then continues with other work; while the DMA module transfers the entire block of data, one word at a time, directly to or from the memory. When the transfer is completed, the DMA module sends an interrupt to the processor.
Impact on the processor

Since the DMA has to use system bus to transfer data, it has to suspend the processor at some point, when the processor has to use the bus also. The DMA will then transfer a word and then give the control back to the processor.

During the process, the suspension is not really an interrupt, but rather a pause for a bus cycle, since it does not need save anything.

The net effect is, however, delaying the processor to some degree. But, for a multi-word I/O transmission, DMA is far more efficient, compared with either programming I/O or interrupt driven I/O.
Make it more efficient

This delay can be further cut by grouping the DMA module with the I/O modules so that there is a direct connection between them, instead of having to use the system bus.
Efficiency is important since I/O often forms a bottleneck in a computer system, as most of the I/O devices are extremely slow compared with processor and main memory.

One solution is to use multiprogramming, which allows some processes waiting for I/O results, while others running.

As we discussed in the previous talks, multiprogramming alone will not solve the problem. Hence, a major design issue is to improve I/O efficiency, particularly, that of the disk operations.
Uniformity

It is also interesting to provide a uniform and general way to handle all I/O devices. Thus, the *generality* objective.

This covers both the way a process views the I/O devices, and the way an OS manages them.

Always to achieve this objective is to adopt an hierarchical, and modular, approach which hides the messy details of I/O device at lower level, while providing uniform high-level operations for the users. (Plug’n play)
A logical structure

The *logical I/O module* deals with an I/O device as a logical resource only. It serves as an interface to the user, providing some uniform operations such as open, close, read, and write, organized via a GUI jacket.

The *Device I/O module* converts the requested data and operations into appropriate sequence of I/O instructions, channel commands, and controller orders.

Buffering techniques, to be discussed later, may be used here to raise efficiency.
The scheduling and control layer does the actual queuing, scheduling, as well as the control of the operations. Providing such mechanisms as interrupt, status collection, this is the layer that actually deals with specific I/O modules, thus the hardware.

Another case involves with managing I/O on a secondary device that supports a file system.

Besides what have been discussed, there are three additional layers, including: A directory management layer converts symbolic file names to identifiers that either references the files directly or indirectly via a file description or index table. It also provides such operations as add, delete, or reorganize for the management of the directory structure.
The *file system* layer deals with the logical structure of files, providing operations such as open, close, read, and write, as well as access rights.

The *physical organization* layer converts logical references to files and records to physical secondary storage addresses, in terms of physical tracks and sector structure. This layer also takes care of storage space allocation, and buffering.
A picture worths....
A scenario

As an example, assume that a process wishes to read 512 bytes from a tape one at a time, and put it into its virtual location 1000 to 1511. The easiest way is just to execute an I/O command, such as \texttt{Read\_Block(1000, tape)}, and then wait for the data.

There are at least two problems with this approach: The processor waits for the relatively slow I/O to finish.

Moreover, during the whole process, the virtual space 1000 to 1511 for that process must remain in the main memory. Thus, it interferes with the swapping decision by the OS. This process can’t be completely swapped out.
It could get even worse:

If the process issuing this I/O command is suspended, i.e., swapped out of the memory, before the operation starts, due to, e.g., its time share is used up.

The process is blocked and will not be brought back because this I/O operation is not completed. On the other hand, the I/O operation, another process, is also blocked waiting for the process to be swapped back in so that the data it gets can be put somewhere for this process.

This will lead to a deadlock. The same thing can happen to an output operation.
I/O buffering

To avoid this sorts of problems, sometimes, input transfers should be done prior to an actual request, and similarly, output transfers should be done after an actual write operation request.

This technique is referred to as buffering. We will discuss a few buffering schemes here to have a better understanding.
More than one...

There are two kinds of I/O devices related to the buffering operations: *Block-oriented* devices stores information in terms of discrete blocks of fixed size, and transfers are done one block at a time. Disks and tapes are examples of this type of device.

*Stream-oriented* devices transfer data as a stream of bytes, with no block structure. Examples include terminals, printers, mouse, etc..
Single buffering

When a user process issues an I/O request, the OS assigns a buffer in the system portion of the main memory to the operation.

More specifically, for a block-oriented device, input transfers are made to the system buffer. When the transfer is done, the process moves the block to the user’s space and immediately asks for the next block.

This technique is referred to as reading ahead, based on the assumption that the next block is eventually needed. When data are accessed sequentially, only the last block will be read unnecessarily.
What does it do?

This approach will usually provide a speedup, since when the next block is being read in, the process can work with the block just read in in.

The process can be completely swapped since the data is kept in the system’s space.

The downside is that now the OS has to keep track of the buffers it has assigned.
For stream-oriented I/O device, the single buffering scheme can be used in a line-at-a-time fashion, or even a byte-at-a-time fashion. For something like a dumb terminal where a user inputs one line at one time, with a return indicating the end of that line, this style is certainly appropriate. The situation for the write operation is similar.

Thus, the buffer can hold one single line. When the line is being transferred, the user process simply suspends.

For output, the process can place a line into the buffer and then continue its other processing tasks. It does not need to suspend, until and unless it has another line to output, but the buffer is yet to be emptied.
Double buffering

This is an improvement by assigning two buffers to an I/O operation A process can now transfer data using one buffer, while the OS works with the other.

Assume that $T$ is the time to input one block of data, and $C$ is the computing time in between bringing two blocks of data.

For a no-buffering case, the execution time is essentially $C + T$: Get the stuff then take care of it in a sequential environment.
Buffering is faster...

For block-oriented device, the execution time is roughly $\max\{C, T\}$. Moreover, when $C \leq T$, we can keep the device at full speed with just a single buffer, since the process can always finish its processing before another block comes.

For the case of $C > T$, a single buffer would have to make the buffer wait in $C - T$, while double-buffer strategy will always keep the device at full speed, at the cost of increased complexity.
Disk scheduling

Disks are currently at least four order or magnitudes slower than main memory. Thus, the performance of the disk system is of vital importance. Below shows a general timing frame of disk I/O transfer.
How long does it take disk...?

When the disk drive is operating, the disk is rotating at constant speed.

To either read or write, the disk head must be positioned at the desired track and at the beginning of the desired sector of that track. The time it takes to position the head at the desired track is called the *seek time*, and the time it takes for the head to reach the beginning of that track is called the *rotation time*, and their combination is referred to as the *access time*.

Once the head is in place, the actual operation will start. This time for this last part is referred to as the *data transfer time*.
A bit more...

There are also a few queuing delays normally associated with a disk I/O operation.

For example, when a process issues an I/O request, it has to wait in a queue for that device.

When a device is assigned to a process, if it shares a single I/O channel with other devices, it might have to join another queue, before the seek time begins to kick off.
More specifics

The seek time consists of two parts, the start-up time, and the time it takes to go down the tracks until it gets to the desired one. This later part is not linear to the number of tracks it has to go through, since it is also involved with a settling time, when the identity of the desired one is confirmed.

Magnetic disks have rotational speeds in the range 3,600 to 15,000 rpm, while floppy has a rotational speed in the range of 300 and 600 rpm.
An example

Assume a disk runs at 15,000 rpm, i.e., 250 rps, thus, each rotations takes about 4 ms, with its average being 2 ms.

The total average access time can be represented as

\[ T_a = T_s + \frac{1}{2r} + \frac{b}{rN}, \]

where \( T_s \) is the average seek time, \( b \) the number of bytes to be transferred, \( N \) the number of bytes on a track, and \( r \) the rotation speed.

Thus, \( \frac{1}{2r} \) is the average time for the disk to get “there”, and \( \frac{b}{N r} \) is how long it takes the disk to rotate this many times to get the \( b \) bytes.
An example

Consider a typical disk with an average seek time of 4 ms, rotation speed of 15,000 rpm, and 512-byte sectors with 500 sectors per track. Assume that we wish to read in a file consisting of 2500 sectors for a total 1.28 MB.

If the file is stored sequentially, i.e., it occupies all the sectors on 5 adjacent tracks. Then, the time to read the first track will be 4 ms for the average seek time, 2 ms for the rotation delay, and 4 ms for reading 500 sectors (fitting one track), with a total of 10 ms. For the other tracks, ignoring the seek time, and throwing the average delay, we have that the total time will be 10 + 4 × 6 = 34 ms.

But, under random access, it will take roughly 0.008 ms = (4 ms/500) to read in a sector, then the total time will be 2500 × (4 + 2 + .008) = 2500 × 6.008 = 15.02 s.

**Homework:** Problems 11.7–11.10.
The worst scheduling approach

From the above example, we see that if we select tracks randomly, the performance will be poor. Hence, to improve the efficiency, we need to reduce the average seek time.

In a typical multiprogramming environment, where the OS uses a queue to manage I/O request, thus, for any device, there will be a list of I/O requests coming from various processes. If we select them in a random order, we can expect to visit tracks also randomly, which leads to the worst performance.

We hate it, thus will not use it, but can use this approach as a benchmark to evaluate other strategies.
Other scheduling approaches

The simplest scheduling is still the FCFS approach: we simply process requests kept in a queue sequentially. This is at least fair, since everybody will be treated in the order they come. IF there are only a few processes requiring access and if many of them are to close sectors, then we can expect a good performance with FCFS. But, if there are many processes competing for the disk, FCFS may lead to a random scheduling.

With a system based on priority, we are not aiming at a high disk utilization, but rather trying to meet other OS objectives. For example, we often give higher priority to short batch jobs, and interactive jobs. Thus, although these jobs will go through the pipe quickly; the longer jobs will have to wait much longer.
The *Last-in-first-out* policy actually has some advantage. In a transaction processing system, giving the device to the most recent user will lead to little or even no arm movement when moving through a sequential file. But, this might lead to starvation of some process if it falls back.

The *shortest-service-time-first* policy tries to pick up those requiring the least movement of the disk arm, with the purpose of shortening the average seek time. A random-tie braking algorithm can be used in case of equal distance in the two opposite directions.

**Homework:** Complete problem 11.3.
Comparison

Assume that we use a disk with 200 tracks, starting at 100, and the requested tracks in a process is the follows: 55, 58, 39, 18, 90, 160, 150, 38, and 184, below shows the total number of tracks some of the policies have to go through.
What is going on?

The case for FCFS is straightforward. It jumps around according to the given order, which leads to an average of visited tracks of 55.3.

The SSTF follows a greedy approach, trying to minimize the number of tracks visited between two visits. It leads to an average of 27.5.

SCAN always goes upwards from the current point on, then comes downwards. It gets a average of 27.8 for this sample.

C-SCAN goes in one direction. It goes upwards from the current position on, and, when stuck, goes to the other end, and then continue upwards. For this sample, it gets a 35.8.
RAID

The rate of secondary storage performance improvement is far behind that of the processor and/or memory. This mismatch has really made the secondary storage piece the focus for further improvement.

The same idea of parallel processing has been applied to the disk management in terms of array of disks that operate independently and in parallel. With multiple disks, multiple I/O requests can be handled in parallel, thus raising the storage speed. Since there could be lots of ways to implement such a mechanism, people have agreed to a standard schema, called RAID (Redundant Array of Independent Disks).
What is in the RAID?

The RAID standard consists of the following few pieces:

1. It is a set of physical drives viewed by the OS as a simple logic unit.

2. Data are distributed across the physical drives of this array.

3. Redundant disk capacity is used to store parity information, that guarantees the data recoverability in case of a disk failure.

The use of multiple disks increase the efficiency and performance, but also increases the probability of failure, which is compensated with the redundancy.

There are seven RAID levels to support various layer of requirements.
Disk cache

The same cache idea as we discussed for the main memory can be used to disk memory. In this case, a cache is a buffer in main memory for some disk sectors, containing a copy of some of the sectors on the disk. When an I/O request is made for a sector, we will check the disk cache first to see if it is already there. If that is the case, the request will be served with the cache support. Otherwise, that sector will be read in to the cache from the disk.

Again because of the locality principle, when a block of data is read into the cache to satisfy some I/O request, it is likely that in the future some data in that block will also be needed.
UNIX I/O management

In UNIX, each individual I/O device is associated with a special file, that is read and written in the same manner as any other user data file.

There are two kinds of I/O in UNIX, buffered and unbuffered. The former passes through system buffers, while the latter goes through the DMA facility.

The system also provides two kinds of caches to support the buffered I/O: buffer cache and character queue. Buffered cache is mainly used to carry out block-oriented devices such as disks and tapes; while character queue is more appropriately used to support character-oriented device such as terminals and printers.
Other stuff

Linux I/O management system is very similar to that of UNIX.

Windows manages the I/O via a I/O manager, consisting of four modules: Cache manager handles caching for the entire I/O system; File system drivers transmits messages between the hardware device and its software driver, network drivers to support the built-in networking ability and hardware drivers (tons of them) address the hardware registers of the devices through the entry point libraries located in the Windows Executive.

**Homework:** Go through the rest of §11.10. Discuss the asynchronous and synchronous I/O operations existing in Windows I/O management.