

1 Disk Scheduling

The processor is much faster than the disk, so it's highly likely that there will be multiple outstanding disk requests before the disk is ready to handle them. Because disks have non-uniform access times, re-ordering disk requests can greatly reduce the latency of disk requests. For instance, two requests to the same track are much faster to process in sequence than requests to different tracks. Similarly requests to nearby tracks are faster than requests to distant tracks. A crafty OS will then try to reduce latency and increase disk throughput by reordering disk requests to best utilize the disk. Following are descriptions of several disk scheduling policies. Performance of these algorithms is usually measured against so-called **random scheduling** where requests are selected randomly from the queue of outstanding requests. To explain these algorithms we're going to use the example of a disk with 200 tracks, and the read/write head starts at track 100. The request queue, in order, contains requests for tracks: 55, 58, 18, 90, 160, 38

1.1 FIFO

With a FIFO scheduler, jobs are processed in queue order. If a process is kind and accesses tracks nearby, FIFO may not perform too badly. Even if a process is being nice to the disk scheduler, if there are many running processes, their requests are going to be interleaved, which will mess up the nice process-local ordering.

Track Number	Tracks Traversed
100 → 55	45
55 → 58	3
58 → 18	40
18 → 90	72
90 → 160	70
160 → 38	122
Average	59.67

1.2 SSTF

SSTF stands for shortest seek time first. This algorithm is based on the observation that seek times are lower for nearby tracks. So SSTF picks the request from the queue closest to the current read/write head location. The only tricky part is if there are two jobs with the same distance (one would be towards the spindle, the other towards the edge). In this case, some kind of tie-breaking needs to be employed to pick one. For instance, you could just use a random number to effectively flip a coin and pick one.

Track Number	Tracks Traversed
100 → 90	10
90 → 58	32
58 → 55	3
55 → 38	17
38 → 18	20
18 → 160	142
Average	37.33

As you can see, this is a marked improvement over simple FIFO. However, if a process requests many nearby tracks it can dominate disk activity and greatly increase the latency for other processes (whose tracks are more distantly located). This condition is known as starvation, because one process is preventing the other processes from accessing the disk (starving them from disk access). This may be optimal, however it is not fair to the other processes. Fairness dictates that the latency be as evenly spread among running processes as possible. So other algorithms have been developed to prevent a processes with high track locality from starving the other other processes.

1.3 SCAN/LOOK

Starvation is a bad thing, so OS developers devised a scheduling algorithm based on the elevator algorithm. SCAN services tracks in only one direction (either increasing or decreasing track number). When SCAN reaches the edge of the disk (or track 0), it reverses direction. LOOK is the obvious optimization of having the read/write head reversed when the last track in that direction is serviced.

Increasing		Decreasing	
Track Number	Tracks Traversed	Track Number	Tracks Traversed
100 → 160	60	100 → 90	10
switch			
160 → 90	70	90 → 58	32
90 → 58	32	58 → 55	3
58 → 55	3	55 → 38	17
55 → 38	17	38 → 18	20
switch			
30 → 18	20	18 → 160	142
Average	35.0		37.33

LOOK behaves almost identically to SSTF, but avoids the starvation problem of SSTF. This is because LOOK is biased against the area recently traversed, and heavily favors tracks clustered at the outermost and innermost edges of the platter. LOOK is also biased towards more recently arriving jobs (on average).

1.4 C-LOOK

C-LOOK (circular LOOK) is an effort to remove the bias in LOOK for track clusters at the edges of the platter. C-LOOK basically only scans in one direction. Either you sweep from the inside out, or the outside in. When you reach the end, you just swing the head all the way back to the beginning. This actually takes advantage of the fact that many drives can move the read/write head at high speeds if you're moving across a large number of tracks (e.g. the seek time from the last track to track 0 is smaller than you'd expect and usually considerably less than the time it would take to seek there one track at a time).

Increasing		Decreasing	
Track Number	Tracks Traversed	Track Number	Tracks Traversed
100 → 160	60	100 → 90	10
160 → 18	142	90 → 58	32
18 → 38	20	58 → 55	3
38 → 55	17	55 → 38	17
55 → 58	3	38 → 18	20
58 → 90	32	18 → 160	142
Average	45.67		37.33

1.5 N-LOOK, F-LOOK

N and F LOOK were designed to offset LOOK's bias towards recent jobs. Both algorithms partition the request queue into smaller subqueues and process the subqueues in order (oldest first). N-LOOK is so-called because the request queue is divided into N subqueues. F-LOOK is a simplification where there are only 2 queues, but they are used in a double-buffered fashion. While F-LOOK is processing one queue, all new requests go into the other one. For this example, we assume that the request queue is split into two, with the oldest one containing the requests for tracks: 55, 58, 18, 90. In this instance, N-LOOK and F-LOOK behave the same. Also notice, that in this configuration, it doesn't matter which direction the head was moving in, all requested tracks are less than 100 so it will only move in the direction of decreasing tracks.

Track Number	Tracks Traversed
100 → 90	10
90 → 58	32
58 → 55	3
55 → 18	37
Queue Switch	
Direction Switch	
18 → 38	20
38 → 160	122
Average	37.33

Even though the average number of tracks traversed is the same as LOOK in the worst case,

N and F LOOK are in some sense, more fair than plain old LOOK. The subqueue system caps the maximum latency a process can expect between a request and it being serviced (unlike SSTF that can starve processes for arbitrary lengths of time).

2 RAID

RAID stands for Redundant Array of Inexpensive Disks. An individual hard drive is limited in speed and size, and it's vulnerable to crashing (if your one HD dies, you lose all the data). RAID attempts to overcome these issues by making a bunch of normal small drives act like one giant drive. The RAID controller is responsible for maintaining this illusion (one massive logical drive being simulated by data spread among many smaller drives). RAID has "levels" which are semi-arbitrary numerical designations of different RAID organizations. Some are meant to increase reliability, others performance, and some are a compromise between the two.

2.1 Speed improvements

Striping is a technique for increasing drive performance. Striping spreads data out among multiple disks. In the abstract, striping is like spreading the tracks of one giant drive out among multiple drives. As an example, say that a file is stored in sectors on two tracks 16 and 92. If those tracks were striped across two drives, then both track 16 and track 92 could be read out at the same time (or close to it). Striping increases the transfer rate by attempting to hide the latency of multiple seek requests by parallelizing the tracks across multiple drives (which can effectively seek in parallel). Of course, raw striping degrades reliability. With one drive, you had probability P of failure, with N striped drives, you have probability NP . So, most RAID installations also incorporate some reliability mechanisms to ensure that one bad drive doesn't ruin your whole RAID cluster.

2.2 Reliability Improvements

Data Mirroring is keeping copies of data on multiple drives. Essentially, several drives all act as clones of one another. This is not particularly space-efficient, but it does speed up parallel reads. Parity drives is another technique, where drive groups have a parity drive, on which is stored only the parity information from the other drives. This is much more space efficient than mirroring, but it doesn't speed up reads. Both mirroring and parity drives require redundant writes. Each mirrored drive needs to write the same data, and each write to a parity group potentially requires a write of different parity values to the parity disk.

3 IO Software

Disks are wacky. Now you know how wacky. So, there's sectors and tracks on the physical disk, and the OS writes blocks into these sectors to organize your data. But when you actually

use a computer you see nicely named files, in nicely named directories. You don't see the block structure, let alone sectors and tracks. What's going on here?

3.1 The IO Software Stack

The OS has a whole pile of software sitting between you and the actual hardware.

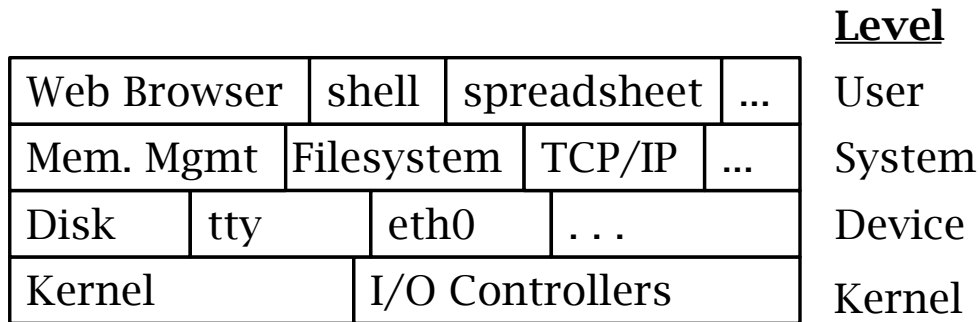


Figure 1: A Beautiful Diagram of the I/O Stack

At the lowest level is the Kernel components, essentially interrupt handlers. Above that are the device drivers. A driver is a device-specific chunk of code that communicates directly with the hardware. Segregating the device specific code into a separate component is a good move from a software engineering and code portability point-of-view. Above the device drivers are the system components, where the device independent code sits. This is often called the 'logical I/O' layer. This is where devices are abstracted into logical devices. For instance, this is where functions to seek, read, and write on a device are exported to the user level, additionally, this is often where filesystem security (permissions, etc.) is addressed. At the top level sits the user code, which makes requests (usually through several layers of library code). Through all this code, the operating system abstracts files that are split up among multiple blocks (perhaps on multiple tracks) into sequential streams of bytes.

3.2 Communication through the stack

Requests to the stack are passed downward, becoming more hardware specific (and consequently less abstract) along the way. For instance, a user process requests a file, the system layer turns this request into a series of block requests (the system layer translates filenames into block numbers), the disk drivers turn the block requests into head seek requests and the kernel suspends the process (assuming this is a blocking I/O request).

Responses from the stack bubble upward. The kernel handles an interrupt from the disk, the driver reads out the track bytes and turns them into blocks in memory, the system level may have to reorder the blocks (in case the driver or the disk rescheduled the requests), these blocks are

then wrapped with the file information which is exported to the user level, where it is perceived as a nicely named file! Errors bubble up as well (you may even consider an error as a kind of exceptional response). At each level, correctable errors are corrected (corrections are often just retrying the request) uncorrectable ones are passed up.

There are two main ways of communicating up and down the stack: Synchronous and Asynchronous. Usually each layer has a choice as to how to communicate with the layer below it (as long as it maintains the semantics of the request it received). Synchronous communication is often called blocking communication. With blocking I/O the requestor waits (blocks) until the response to continue computations. Programmatically, blocking I/O takes the form of a function that causes the I/O request and the program can't continue until the blocking function returns (i.e. the response arrives). Asynchronous I/O is often called non-blocking I/O. In this case, the requestor doesn't wait for the response before continuing on. Programmatically, this amounts to registering a callback function or thread that is run when the response arrives. Efficiency-wise, asynchrony can be more efficient than synchronous communication (particularly in a network setting), however asynchronous communication requires a kind of concurrency, and concurrency is hard.

3.3 Communication Techniques

It seems reasonable for a user process to communicate in the following way (at least for reads). The process allocates a chunk of memory, and then requests that the OS read in a block of data from the disk into the allocated memory. However, now the OS can't fully swap out the waiting process because the allocated memory must remain available to receive the read-in data. Therefore, I/O requests can mess with the OS's process scheduling and memory management. Rather than this direct communication, OSes perform communication through buffers. When a process needs to write out some data it calls into the OS, the OS then copies the data over into a buffer within the OS, thus freeing up the process' memory to be swapped in and out as the OS sees fit. Similarly for reads the OS reads data into an OS buffer first, and then copies it into the process' memory whenever it is convenient. Buffering decouples process scheduling from I/O scheduling and latency. Buffering helps smooth out IO requests, but it complicates OS design, but it's so useful (and makes the memory management and process scheduling guys happy) that modern OSes all use buffered IO.

3.3.1 Single Buffering

Single buffering assigns a single buffer to an I/O task. The user process requests an I/O operation for a certain amount of data and the OS allocates a buffer of the appropriate size. This works, however it basically halts the progress of the user process until the buffer is processed. Consider a process that is generating and writing out data (processing an image, for example). With single buffering, when each block is done the user process will have to wait for the OS to finish writing the buffer out before the process can issue another write.

3.3.2 Double Buffering

Double buffering (or buffer swapping) is an improvement to single buffering where the OS has two buffers available. One is available for processing by the OS, while the other is available for the process to read or write. When the OS is done with its buffer, the OS switches the buffers over. Consider a program that's reading blocks sequentially from a file, when the first block is read into the buffer the OS swaps buffers and starts reading into the other one. The first buffer is then immediately available to be copied into user space. In this way the OS can interleave buffer copying with reading/writing data from devices.

3.3.3 Circular Buffering

Circular buffering is a generalization of double buffering. Instead of two, multiple buffers exist and they are organized into a circular queue.

4 Software Components

Device drivers are software, so they run on the CPU. Drivers communicate directly with the hardware. Drivers know what all the control registers on a particular device (disk controller, network card, etc) mean, and they usually communicate with the device through memory-mapped I/O. In the case of disks, the driver has to know all the ins and outs of the hard drive protocol (e.g. SATA vs. SCSI) as well as specific hard drive geometry (number of tracks, sectors, etc.). Because drivers are at a very low-level, they only have to contend with hardware errors (e.g. hard drive failure). Drivers often do some minimal sanity checking (for instance, asking for a sector that isn't there). Software may have to check because a lot of hardware assumes that it will only be receiving valid requests (it simplifies design and makes it faster). Note that sanity checking isn't always going to happen, however. Drivers may have to buffer requests if they can arrive faster than the device can process them, however device drivers have to be small and fast so any buffering is going to be with a fixed sized queue (a bounded buffer).

System software (or service-level software) abstracts away hardware specifics and presents them to the user in a convenient fashion. On UNIX systems, this means giving the user a convenient string name for a device that corresponds to a specific chunk of hardware at a specific bus location. The service level also aggregates related drivers into a single service. For instance, the filesystem incorporates hard drive drivers as well as drivers for removable media (floppies, jump drives, CD-ROM, etc), and presents them all as part of the same directory tree (except for windows which persists in maintaining the absurdity of explicit volumes). The service level handles higher-level errors, like block checksum failures or filesystem disturbances caused by power failures in the middle of updates. The service level also abstracts errors into a more digestible form so that user software stands a chance of making sense of I/O failures. The service level also handles security issues. For the filesystem this means checking ownership and permissions (at least on UNIX).

At the user level, most programs access I/O devices through library routines. This software abstracts block-oriented I/O into more convenient stream-like I/O. Operating Systems can expose some more low level details at the user level. Linux has device files (those entries in the `/dev`

directory) which give user processes access to raw data blocks on the device. This data is revealed through file semantics. Standard open, read, write calls are used to access the devices.

5 Linux Devices

Linux uses the device filesystem (devfs or udev) to create entries in the `/dev` directory that correspond to real devices in the machine. Both devfs and udev support hotpluggable devices (meaning that it can detect devices that weren't attached to the machine at boot). `procfs` exports kernel state to the `/proc` directory. Device information is at `/sys` and is handled by the sysfs system. `/sys` exists so that tuning programs can tweak device parameters to increase performance.

6 Sample Driver Uses

These examples are derived from the MINIX OS. MINIX was the inspiration for the Linux system and as such, represents a sort of simplified Linux.

6.1 Initialization

During boot, the OS calls init routines for installed drivers. The drivers then read the physical device data from the hardware (or BIOS) and store it in a data structure in kernel memory. Then a jump table is initialized to provide linkage between the generic kernel APIs and the specific routines in the driver. This indirection allows for drivers to be loaded piecemeal without having to recompile the entire kernel.

6.2 Mounting, open

The device is accessed only at the first physical read to the device. Then the driver prepares for subsequent accesses (data structures needed only for reading and writing are created only when necessary). The minor device number is used to identify exactly which device (or partition) is being used. The driver then installs an interrupt handler to respond to the drive controller. The driver then queries the controller for all the specifics (drive geometry, volume label, etc.). The driver then sets the comb to a starting position.

6.3 Data Transfer

The driver first sets up data structures for the transfer, including source/destination and size of the transfer. The driver then waits for previously scheduled requests to finish. Then the driver gains control of the bus and sends the request to the actual device. At this point the driver waits to be activated by the interrupt that signals when the operation is done. When the interrupt is received, the driver updates status flags and signals the user process.

6.4 Unmount

Usually there's not much to do at the hardware level (apart from ejecting the CD or tape). Mostly unmounting is just cleaning up the data structures associated with the now unmounted drive.