ECE 350: Real-Time Operating Systems Spring 2023

Lecture 20 — Input/Output Buffering and Scheduling

Jeff Zarnett *2023-03-15*

Buffering

Regardless of whether a device is block- or character-oriented, the operating system can improve its performance through the use of buffering. As you may have experienced with Java, the use of a buffer speeds things up quite a lot. A buffer is nothing more than an area of memory that stores data being transferred, whether it is from memory to a device, device to memory, or device to device.

A buffer is a good way to deal with a speed mismatch between devices. Users type very slowly, from the perspective of the computer, and it would be awfully inefficient to ask the disk, a block oriented device, to update itself on every single character. It is much better if we wait until we have some certain amount of data (e.g., a whole line or whole block or full buffer, whichever it is) and then write this out to disk all at once.

The write is, however, not instantaneous and in the meantime, the user can still keep typing. Thus, to solve this, the typical solution is *double buffering*, that is, two buffers. While buffer one is being emptied, buffer two receives any incoming keystrokes. Double buffering decouples the producer and consumer of data, helping to overcome speed differences between the two [SGG13].

I/O Scheduling

As a final note, we sometimes want to schedule I/O requests in some order other than First-Come, First-Served. A simple analogy: Imagine you need to go to the grocery store, the dry cleaners, and the bank. The bank is located 1 km to the west of your current location, and the grocery store is 3 km west. The dry cleaners is in the same plaza as the grocery store. It is obvious that it would be fine to go to the bank, then the dry cleaners, then the grocery store, but not to go to the dry cleaners, then the bank, then the grocery store. The unnecessary back-and-forth wastes time and energy (whether walking or fuel depends on your mode of transportation).

Clearly, the operating system will want to do something similar with I/O requests. It will maintain a structure of requests and can then re-arrange them to be accomplished most efficiently. The literature sometimes refers to this as a queue but... is it really a queue when it does not exhibit the first-in, first-out behaviour[1]? This will, naturally, have some limits: requests should presumably get scheduled before too much time has elapsed even if it would be "inconvenient". It might also take priority into account, dealing with the I/O requests of a high priority process even if they are not particularly nearby to other requests.

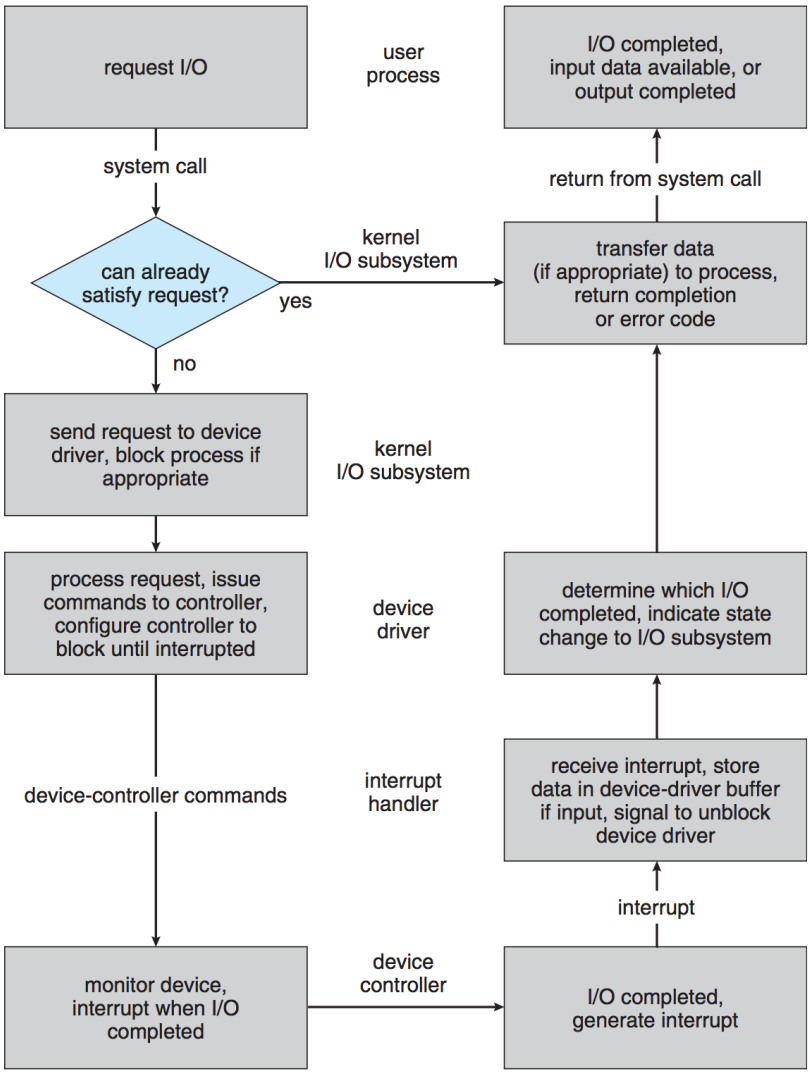
In particular, this is important when examining hard disk drive operation. And that will therefore be the next subject we will consider.

Transforming I/O Requests to Hardware Operations

In the previous section we discussed the idea of taking a command like `read` and said it is the job of the device driver to translate this command into a hardware operation. Reading from the file system on disk, for example, requires a few steps. If I want to open a file like `example.txt`, the file system (not yet discussed) will associate this file with some information about where on the disk this is (a set of disk blocks representing the file). Then, to read the file, `read` commands can be issued to get those blocks into memory so I can edit it with `vi`.

[1] But I'm kind of strict about my use of grammar and language. Things like "10 items or less" signs bother me, because I know it should be "10 items or **fewer**". For those who are fans of *Game of Thrones*, it will probably not surprise you that I like Stannis Baratheon.

The diagram below shows the life cycle of an I/O request:



The life cycle of an I/O request [SGG13].

In general, the life cycle follows some ten steps from start to finish [SGG13]:

1. A process issues a `read` command (assume the file is already open).
2. The system call routine checks the parameters for correctness. If the data is in a cache or buffer, return it straight away.
3. Otherwise, the process is blocked waiting for the device and the I/O request is scheduled. When the operation is to take place, the I/O subsystem tells the device driver.
4. The device driver allocates a buffer to receive the data. The device is signalled to perform the I/O (usually by writing into device-control registers or sending a signal on a bus).
5. The device controller operates the hardware to do the work.
6. The driver may poll for status, await the interrupt when finished, or for the DMA controller to signal it is finished.

7. The interrupt handler receives the interrupt and stores the data; it then signals the device driver to indicate the operation is done.
8. The device driver identifies what operation has just finished, determines the status, and tells the I/O subsystem it is done.
9. The kernel transfers the data (or error code or whatever) to the address space of the requesting process, and unblocks that process.
10. When the scheduler chooses that process, it resumes execution.

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.