Lecture 20 — More About Input/Output Devices

Jeff Zarnett
`jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

April 29, 2023

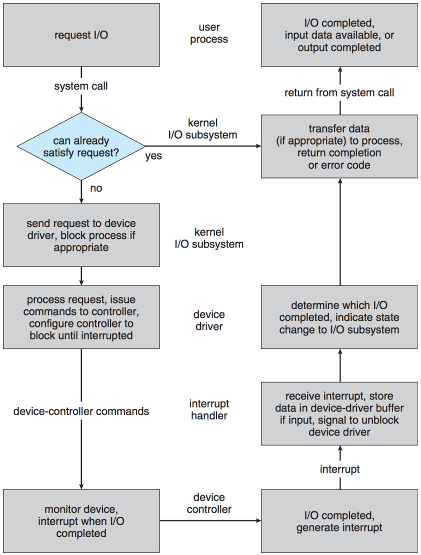
Transforming I/O Requests to Hardware Operations

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 will associate this file with some information about where on the disk this is.

Then, to read the file, `read` commands can be issued to get those blocks into memory so I can edit it with `vi`.

I/O Request Life Cycle


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
no
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt

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, the use of a buffer speeds things up.

A buffer is nothing more than an area of memory that stores data being transferred, from memory to a device, device to memory, or device to device.

Buffering

A buffer is a good way to deal with a speed mismatch between devices.

Users type very slowly, from the perspective of the computer.

It would be awfully inefficient to ask the disk, a block oriented device, to update itself on every single character.

Buffering, Please Wait

Probably you’re familiar with this from YouTube or similar:

Buffer Calculation

We'll say T is the time required to input one block and C is the computation time between input requests.

With no buffer, the execution time to complete a block is $T + C$.

With a buffer it is $\max(C, T) + M$, where M is the time to move the data from a system buffer to process memory.

Capacity is Limited

A buffer is normally created with a maximum capacity to it.

Sometimes we talk about unbounded buffers, but there’s always a limit.

Implementing a Buffer

Implementing a buffer is relatively straightforward operation from the point of view of the operating system because it really only requires two operations.

Add something to and remove something from the buffer.

Does that sound familiar?

Buffer Implementation

Producer

```
1. [produce item]
2. wait( spaces )
3. wait( mutex )
4. [add item to buffer]
5. post( mutex )
6. post( items )
```

Consumer

```
1. wait( items )
2. wait( mutex )
3. [remove item from buffer]
4. post( mutex )
5. post( spaces )
6. [consume item]
```

Okay, how about now?

Double Buffering

Just a minute ago, we said typing should go into a buffer…

When the buffer is full, write it out to disk, right?

The write is not instantaneous and in the meantime, a user can still keep typing.

Double Buffering

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.

Double Buffering

The obvious extension is to have MORE buffers – circular buffering.

Not Unlimited

Buffering does not solve all problems...

Although it can smooth out peaks and valleys in producing and consuming, if one is consistently higher than the other, buffering doesn't help.

I/O Scheduling and Performance

The OS must keep track of in-progress I/O operations.

Like a semaphore, track the device status and have a queue for it.

How to manage the queue?

History Rhymes

When a thread wants to use the device, check the device status!

If it’s available then we can mark the device as busy and submit the request to the device.

Until the operation is complete, mark the requesting thread as blocked.

History Rhymes

If a thread shows up and wants to use a device that’s already in use, just block it and add that thread to the queue.

When the operation is complete, unblock the thread waiting for it and let it continue.

If there are no further requests waiting, mark the device as idle (not in use).

Otherwise, keep the status as busy and begin the next request.

This is FCFS

That’s FCFS – and maybe that’s adequate.

It’s fair, but perhaps not optimal.

Thought: what if devices have non-uniform access times?

I/O Scheduling

Sometimes we want to schedule I/O requests in some order other than First-Come, First-Served.

Analogy: you need to go to the grocery store, the dry cleaners, and the bank.

The bank is 1 km to the west of your current location; the grocery store is 3 km west; and dry cleaners is in the same plaza as the grocery store.

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.

I/O Scheduling

The operating system will want to do something similar with I/O requests.

Maintain requests structure; re-arrange them to be accomplished efficiently.

This will, naturally, have some limits: requests should presumably get scheduled before too much time has elapsed even if it would be “inconvenient”.

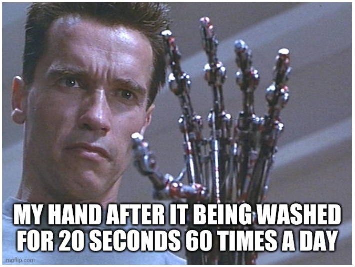
It might also take priority into account.

There Should be a Limit

On the other hand, requests should get scheduled eventually.

There’s tradeoffs between efficiency and fairness!

Too Much of a Good Thing



We've discussed the idea already that interrupts are more efficient than polling, you might question if there's such a thing as too responsive.

Downloading a File Example

For downloading a file, say, the number of interrupts is probably manageable.

As each chunk arrives, the interrupt is triggered, and the chunk is handled.

Eventually, all the file is downloaded and there we are.

Keyboard Interrupt

Let’s imagine I have `eceubuntu` open and I’m typing in an editor.

I press Z locally and then…

- Keyboard interrupt locally
- Network request locally
- Network arrival remotely
- Editor processes keypress, updates screen
- Network request remotely
- Network arrival locally

How to Reduce This?

The book suggests some special hardware.

But we *have* looked at some specialized hardware: e.g., TLB.

Other ideas?

Reducing Interrupt Overload

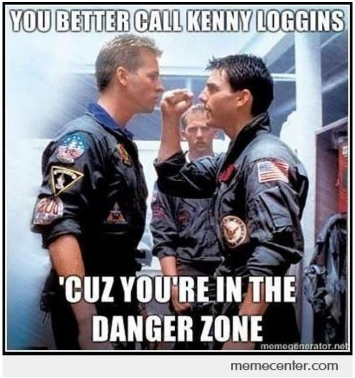
Reduce the amount of time to handle an interrupt?

Wait for a bigger chunk of data?

Use DMA hardware?

Security in I/O Devices

Device drivers run at a high level of trust in the operating system, they offer an easy route for attackers!



Attack vector: privilege escalation in a driver.

But How?

The vulnerable driver has to be on the target system in the first place.

If an administrator wishes, they can intentionally install a driver with a known vulnerability to be able to do something bad.

But admins can already do bad things?

Bad Drivers

Bad drivers should not get installed automatically because of driver signing that we talked about earlier on.

Trying to install a driver that’s not signed tends to result in the operating system throwing a ton of warnings.

Do people ignore these?

Found the Vulnerability Later?

This doesn’t solve the problem of what happens if there’s a vulnerability in an otherwise-legitimate driver.

Such a driver would have been tested, approved, and signed and the problem is only discovered after the driver is released.

That doesn’t prevent installing the old version with the vulnerability.

Revocation

What is needed is some idea of revocation: un-approve a driver.

Microsoft has not been properly applying updates to the driver block-list which resulted in exploitable drivers being installed.

That’s a Bit Late

Microsoft did address this once it was reported, but a little late.

There were some known cases of actual exploits taking place using drivers in the block-list.

And as you may imagine, the installation rate for patches to any operating system is never 100%, so vulnerable systems will continue to exist…