**2025-04-17-os2-en**

0:00  
Let's get started.

0:02  
Last week we ended about here.

0:05  
So I give you a brief introduction into the operating systems and then we end up in this example when I showed you that actually many functions of a very simple application are performed are realized via operating system calls, via sys calls.

0:25  
Today we move on to another sub topic, which would be the architecture of the kernels of operating systems.

0:33  
So basically an operating system is a software like any other.

0:38  
Well, maybe not like any other because it needs needs to be written more delicately and and has many functions.

0:45  
But in general, you need to, let's say, follow some regular software engineering design, software engineering engineering patterns.

0:54  
On the other hand, it's much more difficult because you have some limitations.

0:58  
For instance, the first one you should remember from last time, your entire code is running in special mode of the CPU in the kernel mode.

1:07  
So every mistake you make is much more profound, much more fatal mistake, since any error you put you do in your code.

1:18  
Any, any potential buck in your code might lead into the crash of the entire system, not just a single application.

1:28  
Bugs in applications are usually handled.

1:32  
Let's say fatal bugs in your in your applications are handled by the operating systems by killing the applications.

1:39  
So the operating system is just killing the malfunctioning application, and the system the rest of the applications may run on may continue.

1:48  
However, bugs in operating systems tend to crush the entire system, and that's if you're lucky.

1:54  
If you're unlucky, it might introduce subtle additional bugs into your software, it might compromise security of your applications, etcetera.

2:04  
So this code is a bit tricky.

2:07  
There are many approaches how to design the code well in the early days, and that goes for Linux as well as Microsoft Windows as well As for some other operating systems.

2:18  
The main approach was to create a monolithic architecture, which means everything is part of a one huge ball of software.

2:27  
Of course, even here you need to do some, let's say, leveling of your functionality.

2:33  
For instance, that the most fundamental thing you can do is that you divide your procedures, divide your functions into at least 2 levels or multiple levels, where on the lowest level are internal procedures, internal functions, and on the highest level are service procedures.

2:49  
So basically these functions, these procedures directly correspond to the system course.

2:55  
This is what the system is offering to the other applications via some libraries, via system calls.

3:01  
And then there is a bunch of internal internal procedures, internal functions that are called only from these upper functions.

3:08  
Yeah.

3:08  
So even here you can do some layering.

3:11  
Even here you can do some basic code decomposition.

3:14  
You may follow some good practices, by the way, like the practices you are embracing right now during the labs.

3:22  
OK, well, maybe not entirely.

3:24  
We had some minor disagreements with Mr.

3:28  
Sobota, but but basically what you're doing there is is the similar thing.

3:32  
You try to decompose your code into multiple functions.

3:35  
The trouble, the main trouble with the monolithic architecture is that everything is in one place.

3:42  
So first, it's a good thing, because if you need to call something, it's there.

3:46  
If a new module, if a new part, if a new functionality needs to raise something, take something from the file system or need to allocate memory internally or something, everything is there, everything is available.

3:59  
The trouble is that there is no or very limited extensibility of the design, and extensibility is very important for operating system.

4:10  
Let me give you a very trivial example.

4:13  
When I plugged this into the operating system, the operating system needs to load a new driver for this USB dongle, which connects me to with this with this pointer, which connects the PC with this with this pointer.

4:27  
So basically a new piece of software, new lines of code.

4:32  
Well, not lines of code, but a new piece of binary code needs to be introduced, added into the kernel at runtime.

4:40  
Well, actually it doesn't.

4:41  
It may be.

4:42  
It may be that the that the code to handle this piece of hardware is already in the kernel.

4:48  
That may be.

4:49  
But if if that is so, it means that the the kernel of the operating system would need to encompass all the all the drivers for all conceivable devices that I can connect to this USB.

5:02  
And if if that was so, the kernel would be really huge.

5:07  
So usually we cannot take this approach anymore because there are so many different devices that can be attached dynamically at runtime to the, to the PC, to the, to the hardware.

5:19  
So we need some extensibility.

5:22  
So only the pieces of code, only the drivers, only the the modules that are really required during the runtime, during this particle when this particular PC is running should be in there.

5:35  
So in the case of monolithic architecture, introducing new modules is technically possible, but at first it's quite tedious because when you have a monolithic architecture, then adding something is always breaking the monolithic architecture.

5:51  
And 2nd, it's very error prone because most of this driver, most of these drivers, drivers for various USB devices are not written by the vendor of the operating system.

6:04  
Microsoft isn't designing all these, all these pieces of code because it would be impossible.

6:11  
There are so many vendors of hardware.

6:13  
So usually the vendor of the hardware is creating this piece of software.

6:18  
So we are basically allowing someone else, a third party, to introduce a code into our kernel.

6:29  
And this kernel, as I said before, is a very delicate piece of software.

6:33  
It's the most delicate piece of software you are running on your PC.

6:37  
So we need to keep this software safe.

6:40  
We need to keep it free of bugs.

6:42  
And then again, we need to keep it extensible.

6:44  
So we need to allow others to introduce pieces of software into our kernel.

6:49  
So that's these two, these two assumptions, these two requirements are somehow going one against the other.

6:56  
Yeah, so there were, there were ways and design attempts how to avoid this monolithic architecture, at least how to separate some some of these concerns because it's very, it's very there is no information hiding a a piece of software driver can potentially access any part of the operating systems, even the parts it doesn't require for this.

7:23  
For instance, this is emulating a keyboard.

7:26  
This tries to create an appearance.

7:29  
Is a keyboard actually keyboard with only two keys?

7:32  
Granted, but keyboard nonetheless.

7:35  
And keyboard definitely does not have to access the file system.

7:39  
For instance, it doesn't have to allocate memory, it only needs to report when a key is pressed and that's it.

7:46  
But technically, if I have a monolithic architecture, the software that drives that controls this stuff might technically access my hard drive or it might send something over the network.

7:58  
And that would be, let's say, unfortunate.

8:02  
If it's intentional, it would be a very, very serious security breach.

8:06  
So some other attempts were made.

8:12  
This is perhaps the oldest.

8:14  
The idea is OK, we know that we have some devices, some some pieces of code that doesn't need to run the very core of the kernel.

8:25  
The most important, actually this is the most important part.

8:28  
The part that controls the CPU, the part that schedules the which code is currently running, the part that handles low level hardware stuff like interrupt.

8:37  
We will get to them.

8:38  
So this is the most important part.

8:41  
And theoretically, this part can be and should be written only by the the vendors of the operating system preferably.

8:50  
So let's keep this part separated from the others and build everything else on top of this part because this is like an essential part which everybody requires.

9:00  
OK.

9:00  
The next step would be to introduce memory management because when we can manage code, when we can manage the CPU, then we can manage, then we need to manage the memory, the allocation of memory, etcetera.

9:12  
Actually, there is more than meets the eye here because you don't, some of you perhaps don't know yet about the memory protection.

9:20  
So there is another mechanism called paging and memory protection and we will get to that next week or the other week, I'm not sure.

9:27  
So we will get to that.

9:28  
But this is much more difficult than it appears.

9:31  
This doesn't only involve the allocation algorithms, it involves many, many more things.

9:38  
If we have the memory management, then the next step would be to create another layer which would handle the input and output.

9:43  
So handle things like hard drives, handle things like keyboards, mice, etcetera.

9:50  
And on top of that, we need some more user friend interfaces for things, namely for the files for the hard drive.

9:59  
You don't want to access your hard drive in a way that you would say, hey, I want to read that block or I want to write that block.

10:05  
That would be very tedious, that would be impossible to manage.

10:08  
So we need another abstraction, namely for the file system and namely for the hard drive.

10:14  
So the hard drive isn't operated in the terms of blocks, but it's operated in the terms of files.

10:20  
So the file system introduced the the abstraction of files and directories, etcetera and connects this to the IO and to the memory management.

10:28  
So the files can be properly loaded and stored from and two hard drives, they can be cached in, in memory, etcetera.

10:37  
And on top of that there is another layer, the final layer, which forms the user interface.

10:42  
And by user interface I don't mean only the interface like buttons and windows etc.

10:49  
Mainly I mean interface for the programmers.

10:51  
So this is like an application interface interface for other pieces that doesn't run in the kernel, that runs outside of the kernel external applications.

11:02  
By the way, if you run Windows or Linux, most of the applications you consider part of the operating systems aren't part of the operating systems for the management of Windows.

11:13  
In Windows, yeah, I know that is the name of the operating system and term from the UI.

11:19  
So these two things are the graphical things in Windows aren't embedded entirely in kernel, they are separated libraries in user space that you can use to make your life more convenient.

11:32  
OK, so the idea was to create one layer up on the next level and theoretically this could work.

11:40  
And this approach is somehow now incorporated in current operating systems.

11:45  
Not entirely.

11:46  
The original idea would be that each of these layers would run with a separate privilege mode of the CPU.

11:54  
So the CPU would have multiple privilege modes, not not only the kernel mode and the user mode, but multiple levels and and each of these levels would need a special special switch.

12:05  
So for instance, the kernel could kill any of the parts of the any of the modules that would handle memory or input or cetera, because the kernel would would run in a more privileged mode than the others.

12:18  
OK.

12:19  
And this this idea still has some footprints in the real hardware.

12:26  
Do you remember last week I told you about the rings in the Intel CPU design that there are ring zero and ring 3?

12:36  
Ring 0 is the kernel mode and ring 3 is the user mode, right?

12:40  
So these rings were inspired by this idea.

12:43  
The original idea would be that the ring 0 would be for the kernel, ring one would be for the memory management or IO management, Ring 2 would be for external drivers for 3rd party drivers, and ring 3 would be for for user applications.

12:58  
So each level would have its own hardware level of protection.

13:03  
In the end it didn't work out because when the when Intel designed this, the monolithic architecture was dominating architecture.

13:13  
And basically it still is.

13:14  
If you take a look at Linux kernel, yeah, they're trying to do their best to redesign some parts and to to fix this problem, but it's still basically monolithic architecture.

13:24  
So this is one way to go.

13:26  
Another way, another idea was completely different.

13:29  
Actually it came from somewhat from networking, because in networking you usually have the smart network, the Internet for instance, and then you have servers and you have applications, you have services and clients, you have web applications and browsers, etcetera.

13:45  
You have a mail server and mail client, right?

13:48  
So this idea was adopted into the operating systems.

13:52  
The basically it basically says let's keep the kernel as small as possible.

13:58  
Only the most essential parts will be really in the kernel, meaning they will really run in the kernel mode by the CPU and everything that doesn't have to be there will be exported as a service.

14:12  
So basically we will have services which are part of the operating system, but they will run in user mode as the regular applications.

14:21  
So in the end, there will be a very little difference between system services and applications.

14:27  
By the way, if you if you list all your processes in your Linux or Windows or whatever you're using, you probably notice that there are much more.

14:36  
There is much more.

14:38  
There are much more applications, much more procedures and much more programs.

14:43  
Sorry, OK, I cannot do that here.

14:47  
I just want to show you the hey, go away, I want to show you the process list on this PC, but I don't have the administrative rights right now.

14:59  
So you can you can imagine that there are many services running your windows, for instance, as well.

15:06  
So the idea is, is viable and it's, it's still being adopted.

15:10  
But technically the, the services that are running on your windows are not the the system services.

15:15  
They are like additional management services that you also require for smoothly running all your software to give you more comfort, but they are not required.

15:25  
Most of them are not required for the kernel itself, but these services would really perform the kernel job like memory allocation.

15:33  
Memory allocation wouldn't be part of the kernel.

15:36  
There will be a service and you can ask the service for another block of memory for instance, and it will give it to you if some memory is available.

15:46  
And the kernel is mainly responsible for handling the communication.

15:52  
So there is a mechanism where application can communicate with services.

15:57  
All the services can communicate one with the other using some very quick and very efficient protocol.

16:05  
The trouble is that if you defer too much into these services, this communication overhead would be very tedious, very bulky, because technically for every operation between two services or between application and a service needs to be handled via communication.

16:25  
Normally in regular application when application needs something, it calls the kernel.

16:29  
So this core is inevitable when application requires something.

16:33  
But normally in in monolithic org, not only monolithic, but in current kernel.

16:39  
So you usually stay in the kernel mode until the the request from the application is serviced until it's done.

16:46  
But here you just emerge back from the kernel mode passing the message to a service.

16:51  
When this first service realized it cannot handle this request alone.

16:55  
Imagine that this is a request for reading a block from a file, reading some data from a file.

17:01  
So yeah, first service say OK, I'm handling the files, yeah that's right.

17:05  
But I cannot handle it right now because I have no no more memory to to for buffer to buffer the data.

17:11  
So first I need to ask the memory service to give me some memory so I can properly buffer some data.

17:17  
OK.

17:17  
And each of these system calls needs to re emerge back into the user space, trigger another application.

17:25  
So it might cause many system calls because each communication is 1 system call which might produce a undesired overhead not mentioning the communication.

17:38  
Sometimes communicating via small messages may be very efficient if you need to just trigger something or do something very simple.

17:45  
Sometimes it might be very tedious especially if you want to load huge amount of data.

17:50  
And these communication channels only allow to efficiently pass small, small messages.

17:57  
But then again, sorry, this architecture was very popular 20-30 years ago, but it was never really used in wide in widely adopted operating system.

18:11  
It was used in experimental system that there was a like a ton of experimental operating systems written 20 or 30 years ago on this architecture.

18:20  
By the way, if you heard about, I'm not sure if we have it.

18:23  
No, I don't have the name here.

18:25  
An OS which was designed at our faculty by a colleague of mine.

18:30  
That's also an experimental operating system which runs which is built on this idea of micro kernel architecture.

18:39  
Then again, some ideas were adopted to to real world applications, real world operating systems for instance.

18:50  
The internal kernel design follows this paradigm.

18:55  
It may be shocking to some of you that Windows is here is in this terms a bit way ahead when compared to Linux for instance, but that is so.

19:08  
Then again, Linux is trying to adopt this idea by offloading some parts of the kernel.

19:14  
The best example would be the file system in user space.

19:18  
So basically the file system is one of the most actually the least important things from the terms, from the perspective of protection.

19:27  
It really doesn't have to run in kernel mode.

19:30  
So what they do is they they defer all the functions that are necessary to handle file system operations like decoding the internal file system structures, allocating the proper blocks, formatting the data format, formatting the files, meaning properly adjusting the data from the files into the into some blocks, etcetera.

19:50  
So these operation operations are very simple and most of them does not require any hardware support, does not require any interactions with other services.

20:00  
So basically they they deferred this file system to the special application which runs in in user space like a regular application.

20:10  
And every time some other application requires to access the file system, it actually asks the service for.

20:20  
Loading the data or a file or something.

20:23  
So in some cases this is feasible, like in file systems.

20:27  
For some other things like allocating the memory or scheduling processes, it might not be completely feasible.

20:37  
Or it might be feasible but not very practical.

20:40  
The main problem of these systems were that they were quite slow when compared to regular monolithic architectures or so.

20:50  
To give you glimpses of real software, real design, this is actually a map of Linux kernel.

20:57  
Yeah, it's a mess, but then again, all the operating systems are that complex.

21:02  
But you can see that they did some pretty good job in trying to separate some concerns.

21:08  
Even though this is a monolithic architecture, they at least try to divide it into some modules into some blocks that are not entirely independent unfortunately.

21:17  
But at least they are somewhat independent.

21:20  
The layout is 2 dimensional, so the vertical parts refer to different areas of concern.

21:30  
The low level system, the processes, memory management, storage management, network management, human interface, that's pretty obvious.

21:37  
So this is somewhat similar to the well actually it's not.

21:42  
I would be lying to the the vertical is slightly more similar to the to the to the nested approach.

21:51  
And on the vertical scale, the bottom is formed with the hardware interfaces.

21:57  
So this is the most low level stuff you can write.

21:59  
This is the really the the handling of IO, handling of the CPU, handling of the memory allocation, etcetera.

22:06  
And as you go up, upwards, it creates a better and better layers of abstraction.

22:12  
This is like the most important level because it bridges the low level abstractions and the high level virtual abstractions.

22:21  
Technically speaking, communicating between these modules, each of these squares can be perceived as a module almost.

22:31  
So communicating vertically is fine.

22:34  
So if you need a memory that there is some top level memory abstraction at the user space interface, so it communicates downwards to allocate the memory and then report the results backwards upwards, that's fine.

22:47  
If you communicate this way, that's perfect.

22:49  
Communicating horizontally isn't that perfect because each of these domains should be as separate as possible.

22:57  
So every time you see a cross line or a vertical line, sorry horizontal line, that's a not very good line.

23:05  
It's like not entirely a flaw in the design.

23:08  
Sometimes these interconnections are inevitable.

23:11  
However, every time there is a cross section crossing line on the horizontal axis, it indicates a potential problem.

23:30  
Are listed in physical memory operations are die.

23:38  
I'm sorry I don't follow.

23:41  
It's possible because the, the indicators here, you can, if you can read them, actually are function names.

23:49  
So yeah, some, some of the functions needs to be buried very low because they need to communicate with the CPU directly or something.

23:57  
They they basically trigger a specific instruction or an exception or something.

24:00  
So yeah, don't be scared by by awkward naming and awkward positions of these functions.

24:08  
It's a very complicated architecture.

24:15  
No, no, no.

24:16  
It's not like a stuff it does.

24:19  
It's like a functions that it offers.

24:22  
So if something goes wrong, you need to function to die or something.

24:27  
Yeah, but it doesn't mean that you're there.

24:29  
That doesn't mean that this function needs to be always executed or something.

24:33  
OK By the way, some of these square blocks, some of these modules cannot be even fit properly.

24:39  
For instance, there is a page cache and swap which is responsible for page management and swapping, so offloading the pages from memory into the hard drive and vice versa.

24:53  
And as you can see, this rectangle is stretched over memory management and storage management because it needs to perform both.

25:03  
It's very tight, it's very tightly coupled module which needs to step into both domains and it cannot be avoided easily.

25:12  
Actually, yeah, it can be split into two modules like a swap memory management and swap storage management.

25:19  
But then again, the communication between these two modules would be so intensive.

25:23  
It would it could be impracticals or it they just didn't know how to split it properly because of the spaghetti design of this code.

25:32  
Both both are possible.

25:35  
For comparison, this is a structure or diagram of Windows kernel.

25:44  
You may, yeah.

25:50  
It is based somewhat on the micro architecture.

25:53  
So yeah, at the very bottom there is something we just call hardware abstraction layer.

25:58  
So Windows internally performs some level of abstraction over the hardware, which is very important.

26:04  
This is actually embedded in the Linux as well.

26:06  
But in the Linux it's divided into all the sub areas and there is a micro kernel.

26:13  
So there is some communication level which is the most important part, one of the most important parts of this of this kernel because it handles communications of the all other executive units of the of the of the kernel like I/O, IPC, scheduling, memory management, etcetera.

26:33  
And by the way, everything you see in Windows, like the like the Windows that appear, the buttons, the file system itself.

26:44  
Some people complain that that Windows still uses a case insensitive file system naming in some cases and uses things like dot exa to recognise whether a file is executable file or not.

26:58  
Yeah, that's so.

26:59  
But actually that's not the the part of the kernel.

27:02  
The kernel itself is quite cool.

27:04  
This is the.

27:05  
The trouble lies in the Win 32 library, which handles everything else.

27:09  
So the system itself doesn't know anything about file naming.

27:13  
The the operating system can run also POSIX, it can run any Linux without a problem.

27:19  
It can run OS2 without a problem.

27:21  
The kernel itself.

27:23  
The trouble is with the Win 32 library, which takes many, many burdens along the way from the older systems like Windows 1233.13.11, ETC.

27:36  
By the way, this is the kernel which was used in Windows NT and then later in Windows 2000 and Windows XP and ever since.

27:48  
It is an old kernel, but it was well designed 25 years ago, or maybe more so even though it's old.

27:55  
Yeah, of course it has made some progress over the years, but technically it was quite well designed from the beginning.

28:02  
I'm not counting Windows 95 or Windows 98.

28:07  
They they were built on a different kernel.

28:08  
They were basically an extensions of MS-DOS.

28:11  
They they weren't full blown operating system.

28:14  
The kernel were completely different.

28:16  
So this was just to give you some ideas about the architecture inside.

28:22  
Yeah, the architecture itself is very complicated.

28:25  
So in the first year, it would be, I would say, irresponsible of us if we just taught you everything we know about kernels.

28:33  
That's, that's impossible.

28:35  
But then again, there will be, there is another course called operating systems.

28:40  
So if you're interested in that, you can visit that course.

28:42  
And actually you will try to write your very small kernel for a very specific hardware.

28:47  
So it's, it's fun.

28:48  
Actually, I've done it and it's not that bad if you learn CN and start start coding at that level.

28:55  
So another topic now we we would like to cover, I would like to cover this, this lecture would be devices.

29:03  
Devices is one of the essential parts.

29:05  
Device handling is one of the essential parts of operating system and operating system responsibilities.

29:10  
So let's start with a, with a terminology.

29:11  
By device, I mean any peripheral, anything that has a specific purpose like a keyboard or a mouse or a or a monitor or a hard drive.

29:23  
So all these are devices attached to a single system.

29:27  
Yeah, technically there are many devices embedded in the system.

29:31  
For instance, if you just take the motherboard, it looks like 1 device.

29:34  
But actually motherboard is like besides it's a hub which interconnects the the CPU with the memories and with the with extension cards.

29:42  
It also has some integrated devices on it.

29:45  
Usually it has integrated network adapter, which is also a device.

29:49  
It has a sound card integrated which is also a device.

29:53  
So yeah, sometimes appearance appearances can be deceiving, but I will try to keep the example simple.

29:58  
So from now on a device would be usually something you can plug into a computer via things like USB or via SATA cable or something.

30:08  
Each device needs something which is called device controller.

30:12  
So it needs a counterpart which connects the device on the hardware level.

30:16  
For instance, this device requires a USB hub or USB controller that interconnects it on the wire level, on the electrical level with the computer itself.

30:28  
If you connect a hard drive, you need a SATA controller or M2 controller that will communicate with the hard drive on the hardware level, on the electrical level and send some signals on the wires, etcetera.

30:40  
And on the other hand, on the other side, the device controller provides us with a uniform, uniform interface for all the devices of the same kind.

30:50  
So for instance, if I have a SATA controller, SATA device controller, it can connect any SATA device.

30:57  
I don't mind whether this is an SSD device, whether it's a it's a magnetic hard drive.

31:02  
I don't mind whether it was made by Seagate or or Western Digital or whoever now builds hard drives that that that beyond the point I know that whatever hard drive is plugged there I can access via this device controller using some form of some uniform protocol, some some well defined abstraction.

31:23  
And this abstraction itself isn't enough because even among device controllers, I may have multiple device controls.

31:31  
I may have a SATA driver from Intel, I may have a SATA driver from other vendors that cooperate with AAMDI, may have driver made by MSI or whoever now builds the motherboards.

31:47  
So yeah, there may be multiple vendors of the devices and each of them might have slightly different abstraction layer.

31:54  
Yeah, they usually have the same the same common parts.

31:57  
Like when I'm communicating with hard drives, I would usually have things like I need to list all the drive drives attached to this controller.

32:07  
I need to send a block or read a block from this particular drive.

32:11  
So yeah, they will be very similar, but then again, they might be there might be minor differences.

32:18  
Sometimes I may have a SATA drive, a SATA controller that supports rate grouping the disks together virtually, and some SATA controllers doesn't support that.

32:28  
So I need to know how to communicate properly with this particular SATA controller, and that's where the device driver steps in.

32:37  
Device driver is a piece of software, a module, a piece of code that allows us to interconnect this particular device controller on the software level so that the operating system knows how to communicate with this device controller.

32:54  
And again, it's another bridge on the layers of abstraction.

32:57  
It's a software bridge, so it creates the necessary adaptation from a universally well known layer for all the hard drives.

33:07  
So each operating system specifies whoever writes, whoever creates this device.

33:15  
Controls for hard drives need to implement this set of functions.

33:20  
OK, if you want to sell, if you want to attach your hard drives to our windows, you need to implement this set of functions.

33:29  
That's it.

33:29  
And the device driver is responsible for implementing all these functions and then send appropriate signals, translate these functions calls into the appropriate signals into the appropriate communication protocol with this particular device controller.

33:45  
So usually the vendor of the device controller is the one who is responsible for writing the device driver.

33:52  
Yeah, on some levels, some devices are so easy, keyboards, mice, that you don't need an explicit driver from your vendor.

34:04  
Yeah, that that may not be true.

34:05  
If you, if you have a regular keyboard with regular, I don't know, 102 keys, whatever, then you probably don't because it's very generic, generic piece of hardware.

34:17  
It's connected via generic USB.

34:20  
So you don't need a specific driver for this particular keyboard.

34:25  
If you just use a generic driver which expects some generic things from the keyboard, it will work fine for this particular keyboard.

34:34  
However, if you buy a very specific gaming mouse that have not two or three, but 10 buttons or whatever, and these buttons are configurable or, and you can, you can set which color the the mouse glows or whether it vibrates when you get hit or something.

34:52  
I'm just making this stuff up because I'm not that, that much of A gamer.

34:55  
So if you have a very specific device, then yeah, you need to have a specific driver for that that particular device.

35:03  
Having devices creates modular devices and device drivers actually creates the problem of chicken and egg, right?

35:11  
Because let me give you an example.

35:14  
You have a very specific device controller to handle hard drives.

35:19  
So we have a device controller that allows you to communicate with your hard drive.

35:24  
So you need to load the device driver for this controller.

35:29  
Guess where this device driver should be stored?

35:33  
When your PC is off, right?

35:35  
That's right, on the hard drive.

35:37  
So how the hell I will load from the hard drive the code that runs the hard drive?

35:45  
Yeah, that's typical, right?

35:48  
So for that purpose there are some generic interfaces like BIOS or modern UFI and these are like well known interfaces that allows you to some basic communication with the most fundamental hardware like the hard drives.

36:06  
So this is used usually during the boot because that's where this problem arises the most.

36:12  
So you during the boot you can communicate or the operating system can communicate with this UFI to identify all the devices to to get some basic inter operations with these devices.

36:24  
So to get like a vendor string or some identification from the devices with the hard drive, it can perform some low level, albeit slow communication.

36:35  
So it can at least load some data from the hard drive, it can load the boot loader from the hard drive, it can load the most fundamental part of the kernel and it can load the driver for the hard drive.

36:49  
When it initialize all this, then it can drop the UFI and then it can communicate with the driver directly using the device driver that was just loaded.

36:58  
OK, so there is a mechanism how to avoid this chicken and egg problem.

37:03  
So we can slowly and very clumsily load data from hard drives using BIOS or UFI so that we get to the code that will finally run the the device efficiently and and in a much, much elaborate manner.

37:21  
By the way, the interaction between the device controllers and devices might be quite difficult or elaborate.

37:30  
Normally, or what would you probably expect is that the device controller has a direct link to all the devices attached to it.

37:38  
Well, in many cases there is one device controller and one device attached to it.

37:43  
But then again, if you have a more elaborate PC rig, then you probably have multiple hard drives, right?

37:51  
I have two hard drives in my laptop, but that's because I take my old hard drive from my previous laptop, so now I get 2.

38:00  
But then again, if you have like a big case, you can have easily 3 or 4 hard drives in your case.

38:06  
So there is one SATA controller which is connected to using multiple wires, small wires to all the hard drives you have in your rig.

38:19  
That's the easy part.

38:21  
In all days when we have we didn't use SATA or M2.

38:25  
Have you heard of Pata?

38:27  
Pata it it stands for parallel ATA.

38:33  
The ATA is for communicating with the hard drive.

38:37  
So this was parallel ATA.

38:39  
This is serial ATA by the way, just a small the diversion SATA was devised because communicating via serial cable is faster than communicating via parallel cable, because if you have a serial cable, you need only to synchronize 2 wires.

39:01  
If you have parallel cable where there are 40 or 80 wires running in parallel, it's much more complex for synchronization and it much easier.

39:11  
Creates some noise on the lines.

39:17  
I need to flue that up exactly.

39:21  
I don't know, don't remember from from the top of my head, but you can Google it up easy, easy enough.

39:27  
So the parallel part was used to create a very bulky piece of cable.

39:36  
The cable was thin and wide because you need to place 40 or 80 lines in parallel.

39:43  
And it was also easily damaged because you just wrap it a few times and then you can throw it away because one of these 80 lines was broken or or interrupted or something.

39:54  
And that was very tedious.

39:55  
So This is why SATA is better than than Pata.

39:58  
But then again, once you got this cable, it was very difficult to have multiple cables in your case because you cannot possibly have like 8 Pata cables in your case.

40:10  
There was no room for that because these cables are very bulky.

40:13  
So what they did and also the design of device controllers was very expensive back then because each device controller was basically a separate chip.

40:22  
So what they did is they allowed you to connect 2.

40:28  
Two disks on one cable.

40:30  
So the cable basically had three connectors which were all connected to all 40 or 80 pins.

40:37  
And these connectors, one was plugged into the device controller and the other two were plugged to two disks or to a disk and a CD-ROM.

40:47  
Yeah, you probably don't remember what CD-ROM is, right?

40:49  
Because nobody is using cd-roms any longer.

40:53  
So this way, 2 devices were connected over the same bus.

41:00  
By the way, this is also the topology that we used in a very old setup of in networking.

41:09  
So if you had, if you ever heard of coaxial cable, T10 network, Ethernet T10, it uses one huge cable which was running throughout the whole classroom and all PCs were connected to this one particular cable.

41:26  
So they were all communicated over the single, single bus.

41:31  
Another possibility is a ring.

41:34  
This is quite of a awkward topology because in this case the controller has one output and one input, and each device has one output and one input.

41:46  
So basically when any communication needs to happen, the DC sends a frame for communication, which is transferred to device one, then to device 2, then to device 3, and then it goes back to the DC.

42:00  
And each device is interacting with this frame in a manner whether it's designed for this device or not, whether it's allocated, whether it's addressed for this device or not.

42:11  
So basically the DC specifies in the frame to which devices this frame belongs to.

42:19  
And the devices that don't shouldn't interact with this frame, just pass it along.

42:26  
Yeah, it's very awkward, but then again it increases throughput.

42:30  
In bus architecture you have the trouble that if the first device is communicating with the DC, then no other device can communicate with the DC or even these two devices cannot communicate one with another.

42:42  
Well, well, normally this is not what happens.

42:45  
But in networking that might happen in normal, in in disk design it wouldn't happen.

42:51  
So when you communicating with the first disk, the second disk remains idle.

42:56  
You cannot communicate with the second disk at the same time and vice versa.

43:00  
Here you can technically communicate with both, but you need to somehow properly order the right commands and properly read the responses, because when you send a command it takes a few ticks before the response arrives.

43:15  
So it's much more complicated, but it increases the throughput.

43:20  
This is the setup we are using now, which is much more efficient because each device could communicate independently.

43:26  
That's why they switch to this topology.

43:28  
And finally, there are also hierarchical topologies like 3.

43:32  
This is typical for USB.

43:34  
In USB you have one controller with multiple ports and you can plug a hub into a port.

43:41  
Then you plug multiple devices into this hub.

43:43  
So there's there can be some topology.

43:45  
By the way, I remember that there, there used to be this huge fair computer fair called Invex in Brno.

43:53  
I'm not sure if you heard about Brno.

43:55  
That's some place where the usually it's for witness protection program.

44:01  
If you need to disappear, you go to Brno.

44:03  
So that the there was this this affair and first time USB came started to be a normal part of CPUs.

44:12  
They created an experiment and they connected like 120 devices to the same PC.

44:18  
They had like 120 mice there and they, I don't know how many hubs they used, but they all connected these mice into the same PC.

44:27  
Yeah, it was completely useless because you couldn't use any of these mice because just by blowing your nose, you probably moved some of these mice so the cursor was jiggling over the screen all the time.

44:39  
But they just prove it's possible.

44:41  
Technically, OK, so this is about how devices are connected and now how we can communicate with the devices.

44:52  
That's probably the most part because well, we are not teaching about the wiring and about the buses on the hardware level.

44:59  
That's not a part of this course.

45:01  
But we need to to explain how this stuff is operated from the software perspective.

45:07  
That's our business in all days.

45:11  
For communicating with any peripheral devices, the CPUs had specific instructions.

45:17  
For instance, ASICS architecture used this in and out instructions that were used to send or receive data to and from devices.

45:27  
Each device used something which is called a port IO port, usually a device that multiple these ports because one port was for instance used to sending and receiving commands and another port was to come to send and receive data or additional ports were used to indicate errors etcetera.

45:48  
And that was indicated by DX register.

45:51  
So you place the number of the port into DX and you use AL register to send or receive.

45:58  
By the way, how much data we can send or receive by each instruction?

46:02  
What's L?

46:03  
What's the size of AL register?

46:09  
8 bits.

46:10  
It's one byte.

46:11  
Yeah, that's that instruction.

46:13  
Receive or sends one byte of information.

46:18  
OK, you do the math.

46:19  
How many bytes can I process using this instruction when I need to read or write the hard drive?

46:28  
Assume that you have A1 gigahertz CPU and this instruction takes 11 clock cycle.

46:38  
Yeah, not that much, right.

46:40  
So this is very tedious and very problematic.

46:43  
By the way, DX is has 16 bits, 2 bytes.

46:46  
So you can use at most 65,000 ports, not too much as well.

46:51  
But then again, you don't don't need that much port because usually normal devices use only a few ports to operate and you don't have like thousands of devices usually connected to your PC.

47:00  
So that that's this part is fine.

47:04  
The second possibility is to use memory mapped devices.

47:07  
So all the communication is taking place using regular IO like load store operations or move operations in architecture.

47:17  
So you're basically reading or writing in memory how this works.

47:22  
The hardware when it when it boots, the hardware maps its control structures to a specific address in the memory.

47:35  
So the hardware when it boots through UFI or through BIOS, you find out that particular device, like the controller of the hard drive is mapped at this fixed address.

47:47  
So you somehow determine this address and then you have a specification for this device controller.

47:53  
So you know how this structure looks like in memory.

47:56  
You know that the 1st 32 bits are the control word, the command word or something.

48:01  
Then there is an error word or something.

48:03  
So you have a structure, you have the layout how this structure looks in the memory, and every operation, every read and every write triggers some hardware action.

48:12  
So instead of storing or loading data to and from memory, it stores and loads or sends or receives commands from this particular device.

48:22  
Basically, when you write something to the specific location in memory, it's the same as if you sent a explicit command to the device, and vice versa.

48:30  
When you read something from the memory, then you perform the loading operation, the fetch the input operation.

48:37  
With this particular device, the benefit is obvious.

48:40  
You can choose or the device can choose its own width for communication.

48:44  
So basically you don't have to rely on communicating by sending one or receiving one byte.

48:50  
You can write the whole structure.

48:52  
Furthermore, this structure may have some additional space, for instance for data buffers.

48:58  
Let me give you an example.

48:59  
You're communicating with a network device.

49:02  
You probably remember that when you use network, there are some IP packets, IP data grams, right?

49:07  
They have maximum size, right?

49:11  
So it's easy to allocate this maximum size as a buffer somewhere in the memory.

49:17  
And automatically the device not only uses the memory to communicate, to operate the device, but it also gives you a glimpse of its own buffer into your memory.

49:27  
So if you need to send or receive a packet, an IP datagram, you just read it or write it from this particular address in the memory.

49:36  
So This is why today all the devices basically use the memory mapping approach.

49:47  
All right, let me give you an example how the handling with the with the device.

49:55  
Some communication with the device is performed in a few, let's say well defined steps.

50:02  
In the beginning, some application issues an IO request.

50:06  
To be more specific, to be more concrete, let's just say that you have something like.

50:21  
You are reading some data from the disk.

50:23  
This is the file point handler you just opened.

50:26  
And then you have a buffer here and and how much bytes you want to read.

50:29  
Something like that.

50:31  
This is a pure C function which is implemented in in libc.

50:37  
So this is some IO library you are using fine.

50:42  
This function calls.

50:52  
A POSX function read which performs the same task but on the system level.

50:58  
This translates directly into the system call into the syscall.

51:03  
So here we are entering to the kernel mode using this operation.

51:10  
Afterwards here the kernel performs some checking.

51:14  
For example, it checks whether this file handler which needs to be passed over here is valid file handler, whether the reader, whether the application has the privileges to read this file, whether the file exists, etcetera, etcetera.

51:27  
But in the end, at some point, assuming all the checks were OK, it transfers the request to read a specific data buffer from the device.

51:38  
So it asks the device driver to transfer some data buffer from explicit address on the hard drive.

51:46  
And yeah, addressing on the hard drive is a bit tricky.

51:48  
I haven't explained that yet, but imagine that the hard drive also have some addressing mechanism to identifying particular blocks.

51:55  
So it asks for a particular block or even multiple blocks from this device driver.

52:01  
The device driver then actually now we are crossing the threshold from the software to hardware.

52:06  
So the device driver is initiated from the software using some memory, memory mapping communication and then the device driver itself initiates the communication on the wires on the hardware level with the device controller.

52:21  
So basically we are like sending signals to the device controller.

52:25  
The device controller is communicating with the device.

52:28  
That's completely hardware business.

52:30  
And actually as I said, it's none of our business in this class.

52:33  
So there is some communication over the wires and then the device controller needs to somehow report back to the CPU, reports back to the software that something is happen, something has happened.

52:45  
This is the most tricky part this this is the most difficult part because we need somehow to trigger an impulse to the CPU.

52:53  
We need somehow to notify the CPU that the loading of the block is is completed.

52:58  
Technically speaking, the device driver can wait for this, can loop infinitely, waiting when something happens, reading some specific data in memory, some some some memory location, waiting when when it's changes its state to signal that that the data block has been loaded into the memory, and then it reports this back to the regular kernel code.

53:23  
It's possible that reading one block isn't enough, because technically I just cheated.

53:29  
I assume that from this you can immediately infer which data block we are reading.

53:34  
That may not be true.

53:35  
We need somehow to translate first where the actual data for this particular file lies on the disk.

53:42  
And that's the business of the file system module, right?

53:45  
So actually there will be probably much more communication going on.

53:50  
1st we need to load the the directory structure, then we need to load some index structure to find out where the actual blocks of this particle file are and then finally we can load the block of the file itself.

54:02  
So presumingly this repeats multiple times to handle all the internal sub requests of the kernel.

54:11  
And finally when all the data are in some memory buffer then the kernel can report back.

54:16  
Sorry, sorry.

54:18  
Can report back to the IO library which gives you the data which returns from this call.

54:23  
And I have my data filled in my buffer.

54:26  
Yeah, I do realize that this was a very top level overview of what's going on.

54:31  
But it's important that you realize that any communication, especially with IO is hierarchical and it takes, it takes many steps to get there.

54:41  
And also it may internally perform multiple independent steps and you don't even know about it.

54:47  
So each F read perhaps performs several accesses to the disk, disk, sometimes even disks.

54:56  
It's possible if you have rate for instance, so that your request is satisfied.

55:04  
Again.

55:05  
As an example I will show you how the most low level handling of IO with a hard drive works.

55:14  
This is a very old example which is using a PAIO.

55:18  
Paio is the oldest interface for hard drive I know and it still uses the old input output instruction.

55:26  
So this one doesn't use memory mapping.

55:29  
It really explicitly sends commands in and out, sends instructions in and out to send bytes to and read bytes from the device.

55:40  
And how this works?

55:42  
I said when I talk about the, when I introduce these two instructions, I told you that they have a DX which is the number of the port.

55:52  
So we need the numbers of the port to fill into the DX when we are communicating.

55:58  
And these numbers actually are different from this table.

56:02  
These are offsets.

56:03  
So I may have multiple Pi O devices and for each of these devices I will need a base port address.

56:10  
So it's, it's something like during the boot, I just query the BIOS and the BIOS tells me, hey, there are 2π O devices.

56:18  
For instance, you have two hard drives, 2 device controllers and first of them is at the address 80 and the second is on the address 90.

56:27  
OK, no problem, why not?

56:29  
So I know that the address 80 + 0 is the data register data port for the first device, 81 would be the error register and 87 would be the status and the command register for instance.

56:45  
Easy enough.

56:45  
OK, so this is the algorithm.

56:47  
How can I read a sector simplified algorithm?

56:51  
I skipped some other parts which are not that important to explain this example.

56:58  
So first of all we always need to check the status of the device.

57:01  
So first I read this register, I read byte from this port and if this byte signals that the device is idle and fine then I can proceed.

57:12  
Otherwise I need to read the error register and find out what caused the last error of the device and how can I reset it or something if necessary.

57:21  
So let's assume the device is fine and I can commence my reading.

57:25  
First I need to set the count register, how many sectors I will read and the address registers or the address ports.

57:34  
Actually the address splits over multiple multiple numbers.

57:38  
There is a sector number, cylinder number which is actually 16 bit wide, so it's divided into two ports and drive at registers.

57:46  
So these 4 registers, these 4 numbers, these 4 bytes actually together from the index of the sector I'm trying to read or the the first sector of the block I'm trying to read.

57:58  
And then once this, once this is ready, I need to read the status register because sorry, I need to send first, I need to send the command, the command 20 initiates the read.

58:14  
If you don't know that, that's fine.

58:16  
You can find this on Wikipedia.

58:18  
This example is taken from Wikipedia.

58:20  
So there is a table list of all commands I can send to this device.

58:25  
So I write 20 to the status command register.

58:29  
And then I wait and wait.

58:32  
How?

58:32  
I'm waiting.

58:33  
I'm actively waiting.

58:34  
I'm actively reading the status register.

58:37  
I'm reading this register, this port until it signals it changes state.

58:43  
It gives me a specific byte that tells me that the data are ready, that the reading has finished.

58:48  
OK, usually it's a good idea not to read it eagerly, like in every loop, because it takes some time to read the data from the hard drive.

58:58  
So usually I can take an app, the CPU can take an app, or it can perform this poll and then for 1000 or several 1000 instructions perform something else switch for another thread for instance.

59:12  
So here I'm reading until it completes or it fails.

59:17  
If it fails, then I need to read the error register to find out why it fails.

59:20  
But if it succeeds, then I know that the data are ready in the internal buffer of the device controller with whom I'm communicating right now.

59:32  
So then I need to read the data register, in this case 256 times because this specific IO operates in blocks of this size.

59:43  
And yeah, if I read multiple sectors, I need to read it multiple multiple times.

59:48  
But usually I just read 256 bytes and then check the check the, check the status register again and then I can read another 256 bytes.

1:00:00  
So I repeat these two operations until I read everything I want it to read.

1:00:05  
Yeah, just imagine, just count how many instructions this would require.

1:00:10  
This would require quite a lot of instructions.

1:00:12  
And then again, I'm reading the data byte per byte.

1:00:15  
It's very slow and very tedious.

1:00:17  
But then again, this design was this interface was designed 30-40 years ago or something.

1:00:24  
So back then, this was the only way how to read data from hard drives.

1:00:30  
OK, obviously this is not the idea way how to do this.

1:00:34  
One of the besides the bad communications, besides that I need to access the device byte by byte.

1:00:40  
There is another problem which is called.

1:00:42  
I need to perform pulling.

1:00:44  
I need to repeatedly ask the device whether it has completed.

1:00:50  
Have you seen the the film Shrek 2?

1:00:52  
I think it was, Yeah.

1:00:54  
So if you're traveling to a Kingdom far, far away, pulling it means that you're repeatedly asking, are we there yet?

1:01:00  
Are we there yet?

1:01:01  
No.

1:01:01  
Are we there yet?

1:01:02  
No.

1:01:03  
So you need to repeat this procedure until someone tells you either yes or either we won't reach the destination ever.

1:01:11  
So until then, you're trying to ask the question again and again and again.

1:01:16  
Obviously it wastes resources because if you need to repeatedly read something, if you need to repeat it, pull something.

1:01:23  
It takes some resources, it takes time.

1:01:26  
You are actively consuming the CPU cycles.

1:01:31  
So better ways for device how to notify the CPU that the job is done and that way is the interrupt.

1:01:39  
So basically interrupt is a mechanism that allows to notify other devices, external devices that something has happened.

1:01:47  
So the device doesn't have to check all the devices all the time using pulling, but it just waits.

1:01:53  
And if a device wants something or the device wants to notify the CPU that an operation has finished, like the loading of a block is completed, then it triggers an interrupt.

1:02:05  
And yeah, the interrupt is really performed on the hardware level.

1:02:08  
So the CPU had less at least one pin, one electrical pin wired to something that can be used to trigger a electrical pulse that triggers internally the interrupt in the CPU.

1:02:22  
OK, technically speaking if we have multiple devices, we would require multiple pins because these devices may trigger independently the interrupt.

1:02:35  
But then again it would be very tedious.

1:02:36  
So what we are doing is there is a IRQ controller interrupt request controller specific chip which is like a connecting all the devices that want to perform interrupts and aggregate all the interrupts together before it notifies the CPU.

1:02:54  
So in in the end the CPU requires only one pin for interrupting and then it requires some communication protocol with the IRQ controller.

1:03:05  
So when an interrupt happens, it just asks the IRQ hey what was the interrupt and the IRQ will tell the interrupt comes from that guy.

1:03:14  
So there is some aggregation performed, usually in front of the CPU, because it would be quite impractical if the CPU would have to have so many pins just for handling many devices.

1:03:26  
For interrupts by interrupt I will explain in short order in more detail.

1:03:32  
By by interrupt, I mean the current flow of instructions of the CPU core is interrupted, so it's suspended temporarily and another code is placed is executed there.

1:03:45  
That should handle the interrupt OK.

1:03:48  
So it's like an immediate termination or suspension of the actual code which is running and temporarily replacing this code with the handler.

1:03:59  
That should do something to correctly answer the device or correctly handle whatever the device needs.

1:04:07  
By the way, there is another thing that you need to know about device communication right away, and that's DMA.

1:04:14  
Since many devices need to communicate with memory, need to load or store data from or to the memory, then this special case is handled separately using the DMA controllers.

1:04:27  
Let me give you an example.

1:04:29  
Even if I have the interrupt, it will still be very tedious if I need to read every data block from the hard drive independently.

1:04:38  
Even if I replace this step, yeah, this one reading the status register which is pulling, even if I replace that with interrupt.

1:04:50  
So I just instead of Step 5, I just ignore everything, ignore this device until this device interrupts and then I will continue with the step 6.

1:05:02  
And yeah, putting aside that this step 6 is quite tedious, but even then I would always get only 25256 sorry bytes or another block size.

1:05:15  
So I will always get a very small portion of the data and then I need to restart another load.

1:05:21  
So it wouldn't be very much useful.

1:05:24  
Then again, I'm usually transferring megabytes or sometimes even larger amounts of memory from or to hard drive.

1:05:31  
So the DMA is a more elaborate controller that allows me to transfer huge blocks, huge chunks of data from a device to the memory or vice versa, without any attention of the CPU.

1:05:44  
So this is going completely transparently, and the CPU is not notified by the device itself, it's notified by the DMA controller.

1:05:52  
That's some bulky huge transfer that was taking the place on the background has terminated and has concluded.

1:06:00  
By the way, modern DMA controllers are much more sophisticated because they not only transfer a continuous chunks of blocks, they can perform operations like scatter or gather.

1:06:10  
Scatter and gather operations are operations where the data are somehow first together.

1:06:16  
Is that the data needs to be first assembled from multiple places in the memory before they are written in one compact block.

1:06:23  
And scatter is the other way around, that some compact block is put into many separated buffers or depending which direction you are looking at.

1:06:34  
The scatter gather is particularly useful in cases like network transfers.

1:06:41  
You probably remember from networking then if you when you're transferring a stream of data, do you remember anything about TCP protocol?

1:06:48  
I believe, yeah, I believe you got discovered in the basic networking course.

1:06:55  
So in you're sending over a stream of data, right?

1:07:00  
You have these buffers, you are sending over huge bulky buffers of data, but the TCP needs to chop this data into smaller packets because one TCP packet needs to be only like well in a regular Ethernet it should be like 1 1/2 kilobytes or something like that.

1:07:19  
I don't remember the exact numbers, but something like a KB.

1:07:24  
So it's split into these small chunks and each of these chunks needs to be put into a IP data gram TCP packet, sorry, which has a header.

1:07:35  
And then again this TCP packet needs to be put into IP edit datagram which has a header.

1:07:45  
So you need to put a small amount of data in front of every chunk here, which is very impractical because normally you would need to copy this into a separate buffer and then copy this into a separate buffer.

1:07:58  
So many copying, but with scatter, sorry, with getter operation, you can specify a separate buffer here for TCP header, a separate header over here for IP header.

1:08:12  
And then when you're performing the DMA, you just say, hey, take this and this and this, put it together and send it over there, right?

1:08:21  
And the hardware does this automatically.

1:08:22  
You don't have to bother with the details.

1:08:25  
So this is like why it's being useful.

1:08:28  
Similar, it might operate when you're loading data from hard drive and data on hard drive.

1:08:34  
One file on a hard drive may be located on multiple locations.

1:08:37  
You have a chunk over here, then there is a block which doesn't belong to this particular file.

1:08:43  
Then there is another chunk over here and you want to load all these chunks.

1:08:47  
So you just put the addresses of all the chunks and one buffer in the memory and it will be loaded together.

1:08:53  
Together operation will take care of that.

1:08:55  
That good point.

1:09:10  
Yeah, technically it might.

1:09:13  
If this device isn't working properly, then it might be a very, very big security issue.

1:09:20  
But then again, you are trusting the hardware.

1:09:22  
So the DMA controller isn't something which is arbitrarily connectable via USB.

1:09:28  
It's not like that I'm connecting another DMA controller over USB.

1:09:32  
This is part of the main motherboard.

1:09:34  
This is part of the system agent basically.

1:09:36  
So yeah, we are trusting that this piece of hardware is doing its job properly.

1:09:41  
And of course, if the operating system screws up and gives wrong addresses to the DMA controller, then yeah, the DMA controller will overwrite memory, which it shouldn't over overridden.

1:09:53  
But then again, it's operating systems fault.

1:09:56  
So yeah, if you have box in your code, your software won't run properly.

1:09:59  
That's that's the risk we're taking.

1:10:01  
But then again, we are in operating systems, we we really should take good care that we don't make these kind of mistakes.

1:10:07  
But technically, yeah, you're right.

1:10:10  
So this is how basic communication works.

1:10:13  
And now let's let me give you a few more details on interrupts.

1:10:17  
Actually, interrupt is a specific state or specific operation of the CPU.

1:10:22  
Interrupt is an embedded function of a CPU.

1:10:25  
By the way, your Rd.

1:10:27  
Nos are capable of interrupts as well.

1:10:30  
We are just not using them because we are we don't have the time nor the capacity in this course to cover them as well.

1:10:38  
We are focusing on something else, but technically they are there.

1:10:43  
So what is an interrupt?

1:10:46  
First you need to know that there are two basic types of interrupts.

1:10:49  
External.

1:10:50  
That's what I was talking about so far.

1:10:52  
So the external interrupt is triggered by a specific hardware source which just pulls the one of the pins, one of the legs of the CPU to signal that something is going on outside the CPU and the CPU should really pay attention to it.

1:11:10  
By the way, the CPU can perform something which is called masking.

1:11:15  
It's it's allowed that for some parts, for some pieces of code, the CPU will will say OK for, for this small piece of code.

1:11:23  
For this part of code, I'm not using interrupts, I'm ignoring all the interrupts.

1:11:29  
So yeah, there are pieces of code that where the the CPU may run without interruptions, but it's a tricky, tricky preposition because if you ignore the interrupts for too long, it might happen that something really important is going outside the CPU and the CPU isn't paying the attention to it.

1:11:46  
So it might break things down if you if you're not using it correctly.

1:11:51  
But then again, it may secure some some very essential pieces of code which really needs to be performed one after another one, one instruction, instruction after another.

1:12:00  
For instance, if you're handling some delicate communication with the device or if you're handling some very delicate piece of kernel code where you like rescheduling some some threats or something, then you really need to make sure that this, these few lines of code, these few instructions will run as they should without any interruptions.

1:12:19  
Then you can internally the CPU can trigger something which is called a software interrupt, which software interrupt is usually referred to as an exception.

1:12:30  
Exception is software interrupt.

1:12:33  
And these exceptions may be, might be triggered by either hardware.

1:12:38  
So you're you're running an instruction and this instruction somehow fails or something.

1:12:44  
Then it triggers internally this exception.

1:12:48  
Or you can do this deliberately.

1:12:51  
There are specific instructions like the int instruction.

1:12:53  
This is a name of an instruction.

1:12:55  
It's a shorthand for interrupt, not an integer.

1:12:59  
It's a tricky part.

1:13:01  
So this is an interrupt instruction.

1:13:04  
So you can trigger some exceptions manually explicitly to give you some ideas what the exceptions are.

1:13:16  
This is a list of X86 exceptions.

1:13:18  
Actually X86 is capable of handling 256 different interrupts.

1:13:24  
It uses 1 byte identifier for interrupt code.

1:13:30  
The 1st 32 interrupt codes.

1:13:32  
The 1st 32 interrupts are the exceptions are the software are the internal interrupts.

1:13:39  
This is lists of some of them actually.

1:13:42  
I'm not expecting you to remember them, nor I'm going through all of them.

1:13:46  
Just to give you some ideas what could be this kind of exception.

1:13:51  
Well, the first one is kind of obvious, division by zero.

1:13:54  
It speaks for itself.

1:13:56  
So if you try to divide perform the division instruction and the second operand is 0, the divisor is 0, then, well, normally the the universe could end or you can create a black hole or something.

1:14:10  
But in in a computer world, we just handle this as an error.

1:14:13  
So the CPU say, hey, I cannot, I really can't perform this, this operation because the physicist or the mathematicians or I'm not I'm not sure who would be really mad.

1:14:22  
So I'm just giving you the error.

1:14:23  
Don't do this.

1:14:24  
This is a bad instruction.

1:14:27  
Another instruction, similar one would be like overflow or invalid op code.

1:14:32  
Yeah, this is maybe this may be a better example.

1:14:35  
If you're binary of your application holds some gibberish, some some data that doesn't make sense, some instruction doesn't make sense, the CPU doesn't recognize the instruction itself.

1:14:48  
That's possible because it might be that that you are not executing code, you're executing data accidentally.

1:14:53  
Or it may be your compiler is faulty.

1:14:56  
Or it may be your compiler is fine, but it assumes 2 modern CPU and you're running your code on an older version of the CPU which doesn't have this particular instruction.

1:15:06  
Yeah, that's all possible.

1:15:07  
Then this exception is triggered.

1:15:09  
And by the way, sometimes this exception can be even useful because you can emulate newer versions of your hardware on the older hardware.

1:15:18  
Just imagine that this invalid OP code is triggered and the operating system, the handler of this exception, finds out that actually the instruction is valid, just the CPU doesn't know it because it's too old.

1:15:32  
Then it can emulate the instruction.

1:15:35  
It can somehow try to create an appearance as if the instruction was executed correctly, although it was actually handled by an exception.

1:15:47  
Now the exception, which is actually 2 exceptions which are very important, but I'm deferring the explanation for later.

1:15:55  
Are these page fault or general protection fault?

1:15:58  
These are related to memory mapping and memory protection and we will get to that later.

1:16:04  
So there, there are some internal mechanisms of, of CPU that allow you to translate virtual memory to physical virtual addresses to physical addresses in the memory and they are very important.

1:16:15  
So we will get to that in more detail.

1:16:17  
And the problems with this memory mappings with this memory handling is also handled via exceptions.

1:16:24  
I will, I will get back to that.

1:16:26  
These exceptions when we, when we're talking about memory management.

1:16:31  
Yeah, a few more things.

1:16:33  
I think.

1:16:33  
One more thing I wanted to point out.

1:16:35  
Ah, yeah, the the tricky part is what should happened if handler of an exception triggers an exception, right?

1:16:47  
Because that sounds like a recursion without the endpoint.

1:16:53  
So normally it's not allowed for an exception handler to trigger an exception.

1:17:00  
But if it still happens, there is this special exception called double fault.

1:17:07  
And double fault is automatically triggered if you throw.

1:17:12  
If you cause any exception within an exception handler.

1:17:17  
So the CPU somehow knows.

1:17:19  
CPU remembers that you are in an exception handler, it's CPU's jobs to remember that.

1:17:25  
And if you are an exception handler and you still trigger an exception, it causes double fault, which is also an exception, but a very specific exception.

1:17:33  
Usually operating system can only kill the running application at this point, or kill the driver that caused the exception or something.

1:17:42  
So at this point the CP, the operating system is forced to take very strong measures against the code that actually caused this kind of failure.

1:17:55  
What happens if we trigger an exception in a double fault exception?

1:18:02  
Yeah, we can change this further.

1:18:04  
Well, actually only to the third level, because there is triple fault.

1:18:08  
There is a specific exception called Triple fault, but its handler is default.

1:18:12  
You cannot change that handler easily.

1:18:14  
I'm not sure even if you can change this handler at all.

1:18:18  
And if you cause triple fault, it resets the CPU right away.

1:18:22  
It says, hey, this code is so faulty that even the operating system cannot run properly.

1:18:28  
So the only course of action we can take is to shut everything down and reset the CPU altogether.

1:18:35  
OK, so these are the examples of software interrupts of exceptions.

1:18:43  
Yeah.

1:18:43  
And that were the exceptions caused explicitly by accidents.

1:18:50  
There are also exceptions which are triggered implicitly.

1:18:54  
Sorry, the other way around.

1:18:57  
So far I was talking about the implicit exceptions that that are caused by troubles in software.

1:19:03  
But there are also exceptions you want to trigger manually.

1:19:06  
You want to actually 'cause them accidentally, like a single step interrupt this exception actually or a break point.

1:19:17  
Yet another great idea is break point.

1:19:20  
These exceptions are triggered when you're debugging, so that after instruction or after a break point, the CPU goes into a different mode and stops the execution of your application at certain point.

1:19:33  
Then it can return back to a different piece of your code where the debugger runs and tells you, hey, we just stop your code there, do you want to inspect it, do you want to do something else with it, etcetera.

1:19:46  
So these are handled by software interrupts, by explicit interrupts.

1:19:50  
By the way, there is one thing I haven't explained so far.

1:19:55  
Yeah, here there are two types of software interrupts.

1:19:59  
One it's called a trap and one it's called a fault.

1:20:04  
Software interrupt takes place either, well, it happens during the execution of the instructions.

1:20:10  
So the question is, what should we do with this instruction?

1:20:13  
Should we discard it, rollback and and erase everything this instruction did so far?

1:20:19  
Or should we let this instruction finish and then we perform the interrupt so the trap does?

1:20:25  
The letter trap triggers the exception after the instruction.

1:20:29  
So once the instruction concluded, the results are written, whatever the instruction performed.

1:20:34  
Then it triggers.

1:20:35  
The exception fault causes the instruction to rollback, so everything is undone as if the instruction hasn't happened yet and it's triggered before the instruction.

1:20:49  
Obviously we cannot trigger an interrupt in in between instructions.

1:20:54  
Well, technically we could.

1:20:55  
Technically it's possible to implement handling of interrupts which would interrupt in one instruction in in the middle.

1:21:05  
But it's very difficult to implement, it's almost impossible to handle correctly.

1:21:10  
So they decided, no, we are not doing that.

1:21:13  
Every interrupt is either before or after the instruction.

1:21:16  
Even the hardware interrupts are either before or after an instruction.

1:21:20  
Usually when you're doing a hardware interrupts, external interrupts, you just let the actual instruction finish and perform the interrupt right after the instruction.

1:21:29  
That's the easiest way how to do that because the rollback, the undoing or of everything the instruction did is kind of tedious.

1:21:37  
It's very complicated.

1:21:42  
So I will explain that on a breakpoint.

1:21:46  
That's a very, very good example, a very favorite instruction of mine.

1:21:51  
By the way, the software interrupt is triggered by the int instruction, so you can write something like int.

1:22:03  
In this case, it's the three which triggers the break point.

1:22:11  
Normally this translates to two bytes.

1:22:16  
This is CD and this is.

1:22:19  
This is 1 byte because 1 byte identifies the code of the interrupt.

1:22:24  
So this goes.

1:22:26  
The number of the interrupt goes here.

1:22:28  
However, this particular instruction has a shorthand which is 0 XCC.

1:22:39  
It's a single byte instruction, which is quite important because since it's one byte, we can use it to place it everywhere in the code and it can override every instruction you know, In normal code we have instructions of different lengths X ASICS architecture MIPS is fine.

1:23:03  
In MIPS all the instructions take 32 bits no problem.

1:23:07  
But in Intel instructions can be 1 byte wide or if they can take I think 13 bytes are the most are the most lengthy instructions.

1:23:17  
So if you want to replace an instruction in your existing compiled code for break points, that's what debugger does.

1:23:26  
It needs to put a some marker in the code so the break point will break on a specific instruction.

1:23:32  
So normally it would be difficult because if the breakpoint takes more than one byte, we cannot for instance override this one because we would override 2 instructions together.

1:23:45  
That's no good.

1:23:46  
So with this particular setup where the breakpoint is single byte instruction, I can easily put it here or here or here to place a breakpoint on a very specific instruction of my code.

1:24:06  
Of course the debugger needs to remember what was underneath.

1:24:10  
It needs to remember what I was replacing.

1:24:12  
So when the break point takes place, when I enter this instruction, the break point is triggered.

1:24:17  
And by the way, the break point is a is fault.

1:24:21  
So break point it's no, it's not there.

1:24:25  
But break point stops and roll backs, so it ends.

1:24:29  
It triggers the exception in front of the instruction that was break pointed.

1:24:33  
So then the debugger can flip it back, can correct it and then rerun the code.

1:24:41  
OK, so this is another way, one way how to use software interrupts.

1:24:51  
OK.

1:24:52  
And finally, and I think this is the last thing we can manage today, is how the actual interrupt is handled.

1:24:58  
And this goes both for hardware and software interrupts, but I will explain it on a hardware interrupt because that is where it's most interesting.

1:25:05  
So imagine something happens, I stroke a keyboard letter so the keyboard sends an interrupt to the CPU.

1:25:13  
Hey, hey, I'm interrupting because I have a very important message to send.

1:25:18  
The user pressed Y.

1:25:20  
So what happens?

1:25:22  
First of all the IRQ controller, the interrupt controller which has many many pins will receive a pin from USB controller which which will signal him.

1:25:32  
Hey, one USB device I'm I have connected to me is signaling that there is an interrupt coming.

1:25:41  
So the IRQ will signal via this specific pin to the CPU.

1:25:45  
Hey we got an interrupt and via some other means via some bus it sends it to some specific register that's not that important.

1:25:55  
It sends it the code of the interrupt.

1:25:57  
So the CPU will know which interrupt is coming and every device has its own interrupt number assigned.

1:26:05  
So basically from the interrupt from the IRQ controller we are getting one byte which will tell us which device or which IRQ number is being triggered is being signaled.

1:26:18  
And from this number we will we will find out which what is the address of the handler that will handle handle this particular interrupt that will handle this particular exception.

1:26:32  
Normally these addresses can be fixed, defined by ISA.

1:26:38  
Or there may be an interrupt table, some specific data structure which translates the number of the IRQ interrupt to the specific pointer to the address where the code that should handle this piece of this type of interrupt resides.

1:26:55  
For instance, X86 has this table located at the very beginning of your memory, so at address 0 it starts this table.

1:27:04  
So at address 0 there is the pointer to the IRQ 0 to the exception 0.

1:27:10  
Actually, because the 1st 32 addresses are dedicated to to software interrupts.

1:27:16  
So this the table tells us the address of the interrupt handler.

1:27:21  
That's what we need to know to start performing, to start executing the interrupt itself.

1:27:27  
So the first step is that the current stream of instructions, whatever it is, whether it is in a kernel mode, whether it is usually it's in application mode, but nevertheless the current instruction stream is interrupted.

1:27:40  
Usually the currently running instruction is terminate is is let the hardware let it complete and the interrupt takes place right after the instruction that was currently running.

1:27:53  
Then a privileged switch needs to happen.

1:27:56  
Because the handler is always performed in kernel mode, it's basically usually part of the of the operating system.

1:28:04  
So we need to switch to the kernel mode and also the CPU needs to save the most essential part of its state in.

1:28:13  
At very least it needs to save the instruction pointer of the application that was currently running.

1:28:20  
It's like a function call.

1:28:22  
We need to know where to resume the stream of instructions that was running and how.

1:28:27  
So the essential part of the state.

1:28:29  
At the very least the instruction point of the program counter needs to be saved somewhere and that somewhere needs to have already pre allocated space.

1:28:38  
So this is safe to usually to some special register or registers.

1:28:42  
These registers are dedicated only to interrupt handling.

1:28:46  
So there is a specific register where the instruction pointer will be copied right during the during the interrupt.

1:28:55  
Then the code is switched to the, to the interrupt handler, to the to the routine we found in this table.

1:29:03  
And this handler needs to save.

1:29:06  
Usually it needs to save the rest of the CPU states.

1:29:10  
Sometimes it needs to save some registers so they can be later properly restored.

1:29:14  
Remember that the whole handler needs to keep the appearance as if the code wasn't interrupted.

1:29:21  
So the the interrupted code that shouldn't know this needs to be completely transparent to the interrupted code.

1:29:27  
So everything that that could be broken by this interrupt handler needs to be properly safe, preserved.

1:29:35  
So it is, it's later restored properly.

1:29:39  
And in between, of course, the interrupt handler performs some useful job.

1:29:43  
It communicates with the device, it updates some internal kernel structures that some device is ready to read or receive the data or whatever.

1:29:52  
So it performs its job.

1:29:54  
Usually this should be a very small routine.

1:29:58  
This should be very fast because another interrupt might happen.

1:30:02  
So we need to keep it short so that the interrupt handler isn't interrupted, right?

1:30:08  
Because that would be a bad thing.

1:30:11  
So normally we just schedule or play some information to the kernel structures that later it needs to pick up some data from the device or something and then complete restoration is required.

1:30:24  
So the interrupt handler restores whatever it saves here.

1:30:27  
So usually the registers and then the CPU restores whatever the CPU has stored.

1:30:34  
So the CPU actually restores what was saved.

1:30:36  
Here at least the interrupt pointer is restored and the stream of instructions continues from where it was interrupted.

1:30:47  
That was the interrupt handling, by the way.

1:30:49  
This is important stuff.

1:30:50  
This is something you should really remember how the interrupt is handled in the CPU.

1:30:56  
Yeah, and that's pretty much a good place to stop for today because the next week we start with processing.

1:31:03  
Are there any questions regarding interrupts, device controls, device handling, anything related to this topic or anything else related to this course?

1:31:15  
Nope, no questions.

1:31:17  
OK.

1:31:17  
So thank you for your attendance and see you next week.