**2025-04-10-lang2-os1-en**

0:00  
Of course.

0:00  
So we have a very simple application.

0:02  
The application have one correct, one message, one string which is to be printed out and it's linked with the library which is using some show function that displays this message somehow, somewhere, probably on the console.

0:16  
And oh sorry, last time I talked about several ways how to link stuff together and how linker works and we ended up about here.

0:32  
So I would like to show you a few examples about how the object files, the products of the compiler look like inside, so that you have a better idea which data needs to be linked together by the linker.

0:45  
The compiler produces some sort of an object file.

0:49  
In Windows, in window ecosystem this is.

0:53  
This uses the cough file format, which is common object file format or something like that.

0:59  
And basically it's like a concatenated bunch of segments of of some sections.

1:05  
This is a list of sections that you can find in the product of the compilation.

1:10  
This is actually actual object file.

1:12  
This is the product of compilation of the example you saw.

1:15  
So this is not an arbitrary file, and this is a real list of all the sections that will be present there.

1:22  
The most important section probably is this one, albeit the name doesn't do this justice because it's just a text.

1:29  
What the hell is text?

1:30  
Actually?

1:31  
Text means code in here.

1:33  
So here is the compiled assembly assembly code of the module.

1:38  
So the all the instructions that should print this message are there and the remind the remaining sections data, our data, etcetera.

1:47  
They're related to data stuff.

1:48  
For instance, I will show you data in our data in short order.

1:52  
The initialized data that needs to be loaded as well into the memory will be here, here X data is related to some some data, some stuff that's that's connected to throwing exceptions.

2:06  
That's not important.

2:07  
AP data I forget right now.

2:09  
And debug is for debugging symbols because if you compile your code into an assembly, you lose a lot of information.

2:16  
So if you want to debug the code later, and by debug I mean you really run it step by step or inspect the memory, what's where, which variables have what values, etcetera, then you need some additional information so that you can properly visualize your code.

2:32  
You can properly inspect all the variables and all this additional information which is used only 4 debugging is here in this section.

2:41  
You can try it yourself if you just compile whatever C code you have and then you can use this dump in tool to inspect it.

2:49  
If you're Linux based you can do the same thing with.

2:53  
Sorry I forget the name.

2:56  
The name of the tool is written in the notes, but there is a very similar tool GNU tool you can use on Linux to inspect LF files.

3:06  
LF file is the object file format which is used for Linux ecosystem.

3:13  
Sorry, no cat just dumps whatever is in the file.

3:19  
This is more sophisticated too.

3:21  
That really needs to understand the format and and read it in a more, let's say user friendly fashion.

3:29  
But you, you cannot just dump it as a, as a hex dump or something.

3:33  
So let's start with the data because there's this like a simplest section.

3:36  
The R data means raw data.

3:39  
There are some additional stuff by the way, D means it's it's in hexa.

3:43  
So so D is not.

3:45  
This is a number and you you can count the bytes that they are really D bytes.

3:49  
This is the actual raw data which which you can inspect in hex format or as a text.

3:56  
And this is a hint for you.

3:58  
Yeah, this is the string because the string needs to be stored somewhere.

4:02  
It's a literal which is which needs to be initialized at the beginning of your application.

4:06  
So it needs to be stored somewhere.

4:07  
So it's stored in this section.

4:09  
By the way, it's it says it's read only section.

4:13  
So it's for constants only.

4:16  
So this is like a initialize data, initialize global variables.

4:22  
Another section holds the relocations just as to refresh your memory.

4:27  
Last time I talked about pieces of code places in your code which which needs to be adjusted when the code is either linked together or when it's loaded to the memory.

4:39  
Because these pieces of code these these places in code are dependent on where exactly in the memory the code is being loaded.

4:48  
Like for instance addresses of of functions where the the function actually ends in the memory.

4:55  
This address we need to remember and put it on every piece of code where this function is actually called.

5:02  
So this pieces of code which needs to be adjusted are called relocations and you can list in this section, you can list all the pieces, all the places in your code that needs to be fixed.

5:13  
Fortunately, it's a very simple code.

5:15  
So there are only two relocations and these are all both are 32 bit relocations.

5:22  
And these are indices in the code where we can, where we can find them.

5:26  
This is just some notion.

5:28  
One of them is the address of the message itself and the second one is the address of the show function which is being called.

5:36  
So they are both addresses obviously and we can put this in in in connect in connection with the code.

5:44  
So let me just dump the disassembled code which is stored in the text in the main text section.

5:53  
I'm not going into the details what's what's each instruction doing, but obviously this is the important stuff we need to consider.

6:00  
This is the call to the show function which we were which we were waiting for.

6:05  
And as you can see this holds like the the call is E8 prefix.

6:09  
This byte says it's a call instruction and the following 4 bytes hold the address of the function.

6:16  
We are in a 32 bit environment, so we need only four bytes for the address of the function being called, and incidentally this address is 0.

6:25  
That's obviously wrong, probably due to bad contrast you don't see right now.

6:31  
These 4 zeros are emphasized in red so you can spot them more easily.

6:37  
However, the Sun is working against us today, so sorry about that.

6:42  
But these 4 zeros are obviously not correct.

6:45  
However, they are listed on the previous slide.

6:53  
Index F points to these four.

7:00  
Actually we are indexing the end, I'm not sure why.

7:03  
So this is the.

7:04  
This is the number that needs to be fixed by the loader.

7:09  
Once the program is loaded into the memory, then this relocation needs to be fixed and the loader, the program that's responsible for initializing and loading and starting this application, is responsible to putting the correct 4 bytes in here where the actual address of the function will be.

7:28  
OK, so this is how how this will work.

7:31  
The same goes for the message.

7:33  
There are another 4 bytes over here because these bytes hold the address of the message where in the memory it ends.

7:41  
So when the program is loaded and the address of the R data segment is is clear, then the loader will fill here the right address where the where the message will be stored.

7:54  
This can be done in a more complex fashion.

7:56  
I'm not going into the details, just to remember there are pieces of code that need to be fixed so the code will run properly after the exact address where the program was loaded in the memory is clear.

8:09  
By the way, a trick question.

8:11  
It is possible some architectures has it.

8:13  
It is possible that some of the core instructions or some of the smooth instructions might be shorter.

8:20  
We have reserved 4 bytes over here and four bytes over here.

8:23  
But it's possible.

8:24  
Sometimes not, maybe not not in here, but in some cases it's possible to replace a call or a jump function, a jump instruction with a long address with a shorter address, because the offset in the memory is not that far.

8:40  
So we can use for instance A relative address which is much shorter.

8:45  
What shall we do with the extra bytes over here?

8:47  
Because we we need to fill them somehow.

8:50  
We cannot just shrink the program, because if we shrink the program we will create additional possible additional problems along the way because we we might disrupt whatever addresses are already pre computed there.

9:03  
So what can we do in case we need to somehow use only part of this and and replace these bits, these bytes with something harmless?

9:22  
Exactly.

9:23  
Fortunately for us, there is an instruction that does exactly nothing.

9:28  
It's called NOP and it has only one byte prefix, A1 byte op code, so we can place it anywhere.

9:37  
Every time we need to pad some space in our code, we can always place specific byte which says NOP.

9:45  
It's a shorthand for no operation.

9:47  
And this instruction is just ignored basically by the by the CPU.

9:52  
It just skipped by the CPU.

9:55  
OK, so good call.

9:59  
These are relocations.

10:01  
And enough about compiling and linking and let's move to the runtime.

10:06  
Support the application.

10:08  
Any application in any language needs some additional help from the environment because it would be very tedious.

10:15  
If you need to write your application in the way it will directly attached itself to the hardware or to the operating system.

10:22  
That would be very, very complicated.

10:25  
So for many things in our applications we take for granted there is some piece of code somewhere that takes care of it.

10:33  
For instance in CC Plus Plus we have a plenty of this support.

10:37  
Well it begins with the compiler.

10:39  
Some some part is already done in the compiler.

10:42  
And also we have the standard library like all other languages C or in C++ have standard library.

10:48  
So it has some standard set of functions and other stuff that is at your disposal.

10:53  
And this also starts with the compilation because you need the you need the inter interface for these particular piece of code for these for these libraries.

11:03  
However, there is much more going on.

11:06  
We also need some runtime support.

11:08  
So your application is isn't running alone, it always runs within some runtime within the ecosystem of the libraries which are there to support it.

11:19  
The most important stuff that's being cared of in the runtime is the memory control.

11:25  
So the organization of the storage, organization of your memory, heap control, heap allocation and this stuff.

11:33  
Of course, at the beginning this needs to be properly initialized.

11:36  
So before the execution, before your main code is, your main function literally is being executed.

11:44  
All the memory which needs to be initialized before that, like all the global variables are somehow loaded or filled with zeros or something.

11:53  
And all the constructors of the global objects needs to be called before your main function.

11:59  
So there's plenty of stuff that needs to be taken care of.

12:02  
And once the runtime is done with that, it's, it executes your main application and it's still there.

12:08  
It still supports you because you can still call some functions from the standard library and stuff like heap control, heap management, It's still there because you need to.

12:19  
Sometime you need to allocate or de allocate some objects dynamically throughout your application.

12:29  
And there is another thing besides the libraries, which are pretty obvious.

12:33  
By libraries, I mean stuff like opening files or writing to consoles or reading the system time, stuff like that, which is pretty obvious.

12:42  
But actually it, we take it for granted, but you need to remember somebody had to roll that.

12:47  
And it's kind of a complicated, it's a, it's a huge stuff, a huge pile of code.

12:53  
And finally, there is one important topic we haven't covered that and we need to dig in right now and that's calling convention.

13:00  
So how functions are being called?

13:03  
Actually this is much more important than you might realize at the first, because calling convention is needs to be observed by all the parties.

13:12  
It needs to be observed by your code, by the compiler, by all the libraries involved, by the operating system, at least the libraries of the operating system, etcetera.

13:21  
Because all the parties needs to use the same protocol, same convention, so that each party understand the other party.

13:29  
You need to lay down properly all the rules, how functions will be named, how the attributes will be encoded or how the parameters, how the arguments will be encoded on the stack, etcetera, etcetera.

13:41  
So the callee or the caller will do their jobs properly.

13:48  
Let's start with what's actually being on the stack when you're performing function call.

13:54  
So when you're calling something or something is being called, depending whether you're the caller or the colleague, this stuff needs to go on the stack.

14:02  
So first of all, there is a some room for return value.

14:06  
Actually, this might be a, this might be a empty space that there might be no place allocate allocated for return value in some not that rare cases, in some I I dare, I don't dare say special cases in maybe more regular cases when the return value is so small, it can be passed down via registers.

14:28  
So.

14:28  
But in the cases the return value is large or whether the core convention does not specify how to return stuff through registers, then we need to allocate appropriate amount of bytes on the on the stack for the return value.

14:44  
Actually, yeah, return value should be something like the last thing we're, we are concerned about because we step, we're just starting the call, but return value, allocating place for the return value is the first thing we need to do.

14:58  
And right away with that, we need to somehow stack all the calling parameters or the function call parameters on the stack as well.

15:09  
By the way, again, some of these parameters might be sometimes placed into registers.

15:16  
If you remember, if you recall the specific the ABI specification of MIPS, then first four arguments are always in registers if they fit the registers, if they are small enough to fit the registers.

15:30  
So if you have more than 4, then they need to be stored in here.

15:37  
This is by the way, prepared by the caller before the call takes place.

15:43  
So this is like a preparation for the call.

15:46  
When the call is done, this stuff needs to be stored at the return address.

15:52  
That's obvious.

15:53  
We need to somehow store where we return from the function.

15:58  
This is very important information and it's it needs to be keep safe somewhere in MIPS.

16:05  
You probably remember that we get the return address in some specific register.

16:10  
There is a designated register where the return address will appear and so that we don't lost it.

16:16  
So, so that we don't lose it, so that it don't don't get lost, we need to store it somewhere.

16:21  
So we save it here.

16:22  
The runtime is responsible for that.

16:26  
And another thing that needs to be saved and is the address of the previous record.

16:32  
This whole record, by the way, I forget to mention, but you probably read that this is activation record.

16:38  
For some reason it's it's called activation record or it's called a stack frame.

16:42  
So this frame needs to point to the previous frame because there is always a previous function on the stack.

16:51  
In the very least there is the main function.

16:54  
Your main function has been called at the beginning, so there should be activation record of your main function.

17:01  
So the callers activation record needs to be saved.

17:06  
So we need this control ring to point to the previous activation record of the core so it's not lost.

17:12  
We know how to return, we know how to de allocate it from the stack and any machine status.

17:22  
And by machine status I usually mean registers that needs to be saved so it's not accidentally changed or something.

17:30  
It's also saved here.

17:32  
Some architectures do do this automatically or semi automatically, like they have instructions for that which just dump something onto the stack.

17:41  
Some architectures require you to do this manually actually, not you the compiler with the runtime.

17:47  
So this is not your job as a programmer.

17:49  
This is taking place transparently in in the background, but nevertheless it has to be done.

17:57  
And finally, once all this stuff is saved, the rest of the stack is available to us for locally and temporary variables.

18:10  
So basically local variables and parameters and intermediate values of your expressions if they don't fit registers again.

18:18  
So everything that needs to be somehow stored in the memory because it doesn't fit registers is then stored after this record on the stack.

18:25  
So these things need to be stored there.

18:28  
And this whole thing is called the activation record.

18:32  
By the way.

18:33  
That's the reason we don't need only the stack pointer, because the stack pointer points over here and sometimes it it can fluctuate, it can move on the stack as we visit separate parts of the function code.

18:49  
For instance, if we visit a for loop which has declared internal variable, then the for loop and for some reason the compiler will will decide not to place this variable into the register.

19:07  
That might be awkward, but it might happen.

19:09  
So if you have something like so a new variable comes to life over here.

19:24  
Or maybe more variables are in the body of the loop and these variables just need to be allocated somewhere.

19:32  
Yes, sometimes they can fit the registers, that's the easy part.

19:35  
But if they don't, they need to be allocated on the stack.

19:38  
So the stack pointer need to move further and allocate room for them that that might happen.

19:45  
However, we still need somehow to keep the pointer to this record.

19:51  
So usually we use a separate pointer which is called frame pointer or sometimes also base pointer which usually points over here.

20:00  
So it points at the bottom of this part which is which is fairly thick, which is firmly fixed because you can calculate the size of this of this data structure from the description of the function.

20:15  
You know what the return value is, what you know what, what the actual parameters is, etcetera, etcetera.

20:20  
So you can, you can calculate what's the size, the size of this.

20:24  
At least the caller can do that.

20:26  
So we just put the pointer over here so we remember where the actual record ends and where the local data starts.

20:36  
By the way, this pointer is then used by the compiler as a base plate, as a base pointer for all the references to all the local variables.

20:47  
So if we need to address a local variable, load or store a local variable, this is the base address and relative offset which might point to the actual parameters or to the local data is used.

21:00  
And this relative offset is compiled, is computed by the compiler.

21:08  
Another, another thing that needs to be taken care of is calling convention.

21:15  
By the way, the convention as I told you this is like like a protocol that specifies all the details, namely the naming or something which is called name mangling.

21:25  
And it's a bit tricky.

21:26  
I get to it in the next slide, the sequence of the call and return housekeeping and everything else.

21:34  
So who is responsible for placing or cleaning up which part of the stack?

21:40  
Who is responsible for cleaning up the core parameters for instance, which parameters are passed via registers, which parameters are passed via stack, etcetera.

21:53  
And also the role of independent registers.

21:58  
So some of the registers needs to be preserved, some of the registers may be, may be disposed of.

22:05  
So this all this is specified in the calling convention.

22:11  
I mentioned the name mangling, it's a very specific thing.

22:16  
I need to explain why we are doing that.

22:19  
The linker which we are using to put everything together needs to somehow identify the functions being interconnected.

22:27  
You know in your code there was a function called show and in some library there is a function called show and the linker needs to identify I that that's the same function.

22:38  
The guy in the code is calling this particular function and the way how the linker does it is via the name.

22:46  
So the name is the only thing we can use for identification.

22:51  
Unfortunately this works in old C because in old C we have only regular functions and function cannot be overloaded.

23:01  
So once you have something like a function Foo, there may be no other Foo anywhere in your code, anywhere in your libraries.

23:16  
So the Foo is kind of a unique name in C This may not be true because in C, first we have namespaces.

23:28  
So you may have multiple fools in multiple namespaces.

23:31  
Second, you have you can have multiple fools.

23:34  
Let's say one fool processes X and another fool takes X&Y.

23:50  
And that's fine because the compiler can make the distinction.

23:53  
The compiler knows whether you're calling Foo with one or two arguments.

23:57  
So the compiler can say we are calling this Foo.

24:00  
Now that's fine.

24:01  
However, the linker can't.

24:04  
The linker can see only the name Foo and that's it.

24:07  
So the linker would be confused.

24:10  
Third, we have methods.

24:12  
We have functions within classes, and that's completeness.

24:16  
So to make the linker compatible with a more advanced computing languages, with more advanced programming languages like C++, we need name mangling, which somehow transfers not just the name, but it also somehow put in information about the namespace and the class and the and the calling arguments, etcetera, in the functional name for the proper identification within the linker.

24:44  
And these are a few examples, by the way.

24:47  
Let's start with the C examples, OK?

24:49  
These are various architectures, various compilers, and these, by the way, these are different protocols, different calling standards.

24:58  
So these are just shorthands, some identifications for various ways how you can call functions.

25:07  
And by the way, if you're writing your software, you should which should be compatible on the function level with other libraries or with other software.

25:15  
You need to specify which course are you using with your functions.

25:20  
So let's start with C this is C this and this is C And we have function F1 with some parameters.

25:30  
So in C it's easy, it's F1F1 with something F1F1.

25:35  
It's easy sometimes, yeah, sometimes the convention says, OK, let's prefix it with\_I don't know why actually, but that that's still fine.

25:43  
You can, you can understand it easily, but you can see it's always F1.

25:47  
Sometimes it's F1 with some stuff, but but basically it's always the same.

25:53  
It's easy in C it's not, because in C we need to distinguish F1 not only by name, but also by the arguments after it.

26:02  
So we need somehow encode.

26:04  
It takes 1 integer, one constant character pointer and one structure S Somehow we need to place that into the name so we can properly identify it.

26:15  
And yeah, I cannot decode this, sorry, from the top of my head.

26:18  
Don't ask me how this is done.

26:20  
But this is almost readable if you, if you just carefully enough, examine it very carefully.

26:27  
The F1 is over here.

26:28  
So that's the name.

26:29  
Yeah, so far so good.

26:31  
The Z2, I don't know why.

26:33  
Why?

26:33  
Why Z2?

26:34  
Then don't ask.

26:34  
But this is the return value.

26:37  
I stands for integer, PK means pointer constant character and P1S is somehow the structure.

26:47  
Maybe I didn't read it completely accurately, so don't just split hairs with me.

26:54  
It may not be entirely accurate, but somehow it's encoded over there.

26:59  
If I misread it slightly, it's possible, but generally it's there.

27:06  
So this mangling is also a very important part of the naming convention, because if you have two compilers or two pieces of software which would perform different naming, then you're screwed because the linker wouldn't identify the names of the functions that needs to be called.

27:27  
OK, and finally one thing I mentioned but I haven't fully expressed is who's responsible for what.

27:36  
So let's say this is maybe better visualization because now we have 2 activation records on the display and this is the this is the activation record of the caller of the function which is just calling another function, OK?

27:52  
This might might be your main function for instance and this is the show function from the example.

27:58  
So the main function already has its has its activation record and it's calling the callee function show.

28:08  
The show function needs to remember the previous address.

28:11  
So basically the frame pointer which points over here also stores the link to the previous activation record.

28:19  
So basically the link over here is then used to adjust FP when we are returning from the call.

28:27  
That's necessary because otherwise we cannot fix.

28:30  
We cannot fix the frame pointer.

28:34  
By the way, this is the responsibility.

28:38  
The responsibility refers to who's responsible of filling this data and then cleaning them up from the stack.

28:46  
So note that the parameters and the return value are the responsibility of the caller, not the callee.

28:56  
So the caller needs to properly allocate both the calling parameters and the return value on the stack, and once the call is done, it needs to remove them.

29:07  
For the return value, that's obvious, because the caller needs to read the return value before it's discarded from the stack, so that that should be obvious.

29:17  
For the parameters that might not be that obvious because it might make make sense that the callee should clean up the parameters.

29:27  
In the end the parameters, the arguments are basically pre initialized local variables and local variables are indeed the responsibility of the core E.

29:39  
Each function is responsible for its own local variables.

29:43  
So basically why the parameters are not responsibility of the core E but the caller?

29:48  
Well, the answer lies in very adic functions.

29:53  
Anyone heard about variadic function?

30:02  
Those are functions which can be called with arbitrary number of arguments usually.

30:10  
Well, in C it's very very tricky.

30:13  
Or in C In Python you probably already used variadic functions.

30:18  
You may not even realize that.

30:21  
Good example would be function like print, not some specific print.

30:28  
I'm just making this up.

30:32  
But hey, I can actually use the real function print F from C This function takes as a first argument a string with a template with some string to be printed out.

30:53  
But this string may hold some specific placeholders which are replaced with parameters that come after the string.

31:03  
So for instance I can say print F and this says I'm going to print 1 decimal number, so it will expect one additional value and this needs to be int or something integer like.

31:21  
So this can be converted printed out as a number.

31:24  
But I can have multiple such placeholders in this string, so I need multiple values afterwards.

31:32  
So this printf function can be called with a different number of parameters and only the caller knows how many actual parameters were put there.

31:42  
The callee doesn't know, the callee has to find out somehow.

31:46  
OK, so This is why the caller is responsible for cleaning up the parameters.

31:54  
OK, so this is how this works.

31:58  
Just the refresher since all the data stored on the stack are basically copied.

32:05  
So this also says that all arguments are made by value.

32:10  
So if you if you if you're calling a function, all the arguments or the parameters you are giving to the function are copied are given as a value.

32:20  
If you need to somehow pass down something which is mutable, like you need an input output parameter or output only parameter, then you need to use a pointer in C or reference in C, But it's the same thing, the reference is hidden pointer.

32:37  
It's just a more convenient way how to deal with pointers.

32:40  
So in any case it will be copied to the to the stack.

32:44  
But if you use reference it's it's copying the address, not the value.

32:49  
But in any case it will end up on the stack like this.

32:53  
So basically if you have a function with one parameter as a regular integer and another parameter as a reference to an integer, the value parameter both are values, but the the first one will be copied as the value.

33:07  
So if the value was 5, sorry, yeah, the a is 5 S the value was 5, so it's copied.

33:15  
The number 5 is copied here and here.

33:18  
Since this is reference, then the pointer of B is taken.

33:22  
B lies on the address 3456, so the 3456 is copied in here instead of the value of B.

33:29  
But it's still the value.

33:31  
It's still some data stored on the on the stack.

33:37  
This is just a refresher.

33:38  
We already covered this in the C startup lectures.

33:43  
So basically this is just to connect, to get your information connected with something you learn at the beginning of the semester.

33:52  
OK, a few more words about variables.

33:54  
So now on.

33:55  
You should get us some notion where each variable lives.

34:00  
The static data and the global data are in static segment which is initialized at the beginning before your code starts and it's placed on in one location in your memory space and it's fixed.

34:14  
It doesn't change size during time and it's always there till the end of your application.

34:20  
Steak serves as a temporary lodging for local variables and for function arguments, which are basically the same from our perspective.

34:30  
And 3rd heap.

34:32  
Also your recovered heap is for dynamic memory allocation.

34:35  
So if you need to allocate some data and you don't know a priori the size of the data because you need to 1st list it or get the input from the user or something, then you can allocate it as you go throughout your your runtime.

34:51  
But the price we pay for that is the heap management.

34:54  
You already seen the heap allocation algorithms.

34:57  
There are some problems like fragmentation, etcetera.

34:59  
Yeah, those cannot be avoided.

35:01  
That that's the price we pay for dynamic allocation.

35:06  
This view, this notion of variables is, let's say, specific to compiled static languages like CC.

35:17  
But this will also almost except for the heap.

35:20  
The heap is, but not entirely.

35:24  
It will also hold for languages like C or Java.

35:27  
However, there are also dynamic languages like Python.

35:32  
You have already seen Python.

35:33  
Some of you may also see PHP or JavaScript.

35:36  
Those are the other two most popular languages I can come up with.

35:40  
These languages actually don't use this type of distinction.

35:45  
They also have global variables.

35:46  
They also have local variables as a model, so the programmer will feel at home.

35:54  
It's a good idea to have global variables and local variables separate, but the variables doesn't mark specific place in memory.

36:03  
If you have a variable, like if you see X as a variable somewhere was int X in your C code, I can tell you exactly where this X is in the memory.

36:20  
Well, maybe not exactly.

36:21  
I need to choose.

36:22  
I need some information like where the application was loaded, what was the offset I need, I need to examine the object files or the executable file so I but I can find out where in the memory the X lies.

36:35  
I can point my finger to it.

36:37  
I know it will be always here in this piece of of memory, either on the stack or in the static data segment.

36:46  
However, in languages like Python, X doesn't mean that X is just a name in some huge dictionary.

36:55  
So imagine in Python you don't have this type of memory allocation.

36:59  
All the variables are in some huge dictionary where the names are translated to some arbitrary values.

37:07  
So basically you have like a huge cloud of names and every time you can ask this cloud, hey, I would like the value of X and it just gives you some value.

37:18  
So it's a different approach because it's much more costlier.

37:22  
You need to look up all the variable accesses first and 2nd.

37:28  
In these languages, X doesn't have specific type.

37:33  
Here I always need to define that X is integer.

37:38  
And we know what integer means on this particular architecture.

37:41  
So we need.

37:42  
We know.

37:43  
For instance, it takes 32 bits in PHP or Python.

37:49  
You ask for X and you get something, and it might be integer.

37:53  
By the way, integer in Python doesn't take 32 bits.

37:56  
It might, might not.

37:57  
It usually doesn't because it has arbitrary precision.

38:01  
So you can, you can have an integer that grows indefinitely.

38:04  
So that's the price we pay.

38:06  
But the benefit is that you don't have to get too much concerned about the variables.

38:12  
I'm telling you this just that you get the idea that there are some types of languages that takes variables and memory management very seriously and other types of languages, usually script languages or dynamic languages, which kind of a don't care about that too much.

38:28  
On the other hand, if you're writing something in Python And you're really concerned about the memory consumption or speed, then you're doing something wrong.

38:37  
You should have chosen a different language.

38:39  
On the other hand, if you're writing something very quickly and you need to rapidly prototype your 20 line code, then Python might be a good idea.

38:49  
A few words about heap.

38:51  
The allocation was pretty much covered in the part where we talk about memory.

38:58  
So I'm not going into more details about just to remember.

39:02  
Yeah, we can use heap to dynamically allocate memory, that's fine.

39:08  
De allocation is the topic we haven't covered thoroughly, at least not from the perspective of programmer.

39:16  
If you want to release your variable, if you want to release the memory you previously allocated in, I wouldn't say Nah, I'm looking for the right word.

39:31  
In serious languages, which take memory management very seriously, you need to do this explicitly.

39:37  
So you need to say, hey, I'm no longer using this memory, get rid of it.

39:42  
This is like CC.

39:45  
You need to say to the runtime this memory should be freed.

39:50  
We don't.

39:51  
We no longer need it.

39:53  
And I'm declaring that if I ever use this memory again, it should be considered a bad idea, it should be considered a fault and should blow up in my face.

40:03  
On the other hand, this is tricky because usually it leads into many errors.

40:09  
The simplest error you can do is that you do nothing regarding your memory D allocation.

40:15  
And if you don't care about your memory D allocation then sooner or later you end up with too much memory and you cannot allocate more.

40:26  
This may not be a problem if you have like a one time script that just runs and then it ends.

40:31  
But if you have a mission critical system or if you have at least a server that runs for days or weeks or months, it, it might be a problem if it even if it slowly grows in memory.

40:44  
So even if you forget to release tiny small bits of memory every now and then, the accumulative error over several months or over years may be disastrous.

40:56  
So it it's very error prone.

41:00  
That's why that's why some languages said, hey, programmers make mistakes all the time.

41:07  
So screw it.

41:09  
We, we don't let the programmers de allocate the memory on their own.

41:13  
We will take care of it for you, for them, because we know better.

41:17  
We won't, we won't fall through, we won't forget.

41:19  
We will do this all the time.

41:21  
So they implement mechanisms that de allocate the memory automatically once it's, it's no longer used.

41:29  
So every time some piece of memory is kind of stopped being used, then the system will automatically get rid of this memory.

41:38  
Hooray.

41:40  
Well, obviously, as you gathered, there is a catch how to find out that some piece of memory is no longer used.

41:48  
It will no longer be used in the future if the programmer hasn't told us.

41:54  
Well, in C, the programmer will tell us, hey, I'm no longer using this piece of this object I've allocated previously.

42:01  
So since then afterwards, it's obvious no one will ever use this object afterwards.

42:08  
Fine, in C you just allocate an object and then sometime later it gets disposed off.

42:16  
But when?

42:18  
How do the compiler, how did the runtime, how did the garbage collector that takes off, that takes care of the things?

42:24  
No, it can be removed.

42:29  
Well, there are several approaches to that.

42:37  
One approach is tracing.

42:40  
This is actually what C# or Java does.

42:43  
So they have some set of let's say active or hot variables.

42:47  
This include all the global static variables which can hold references and all currently active local variables and transitively it scans all the objects.

43:01  
So if you have like a let's say global variable that's a reference to an object then it transitively scans all the member variables of the object and look for other references.

43:17  
So this this needs to be a very depth wide search so all the references are traversed and and mapped properly mapped and that takes time.

43:28  
So C# doesn't do this all the time.

43:31  
Every now and then, usually when it starts running out of memory, it needs to stop all your execution.

43:39  
It freezes.

43:40  
Your program scans all this references to the depth mark all the used objects and get rid of everything else.

43:51  
So everything that wasn't marked during the sweep is removed from the memory.

43:57  
Yeah, it works.

43:58  
The price that we pay is execution stall every now and then.

44:06  
Your execution is frozen.

44:09  
And while it's frozen, the garbage collector can do it's stuff can can remove remove unused objects.

44:17  
The trouble is that for the most part, you don't have a direct control over this behavior.

44:23  
You cannot say, hey now I really need to run.

44:26  
Sometimes it just interrupts your program in the very inconvenient moment.

44:31  
So it it's get embarrassed, embarrassing because the user just clicked on a button and the the the click just triggered an algorithm that requires more memory and there is no more memory.

44:42  
So the garbage collector kicks in and the button just freezes because the the program is performing garbage collection instead of the the interactive feature that was implemented.

44:53  
So the user thinks that the program is broken.

44:56  
Yeah, that might happen.

44:57  
Worse, you use C# or Java for something like self driving car and you're approaching very speedily towards a crossroad and a pedestrian walks in and the system starts for the detection of pedestrians.

45:12  
Needs more memory.

45:14  
So your car is still riding, but the self driving system is waiting for the garbage collector.

45:20  
That also might happen.

45:23  
Obviously not.

45:24  
These systems need some fail safe so that won't get stored by garbage collectors.

45:28  
But just to get the idea, it might get troublesome in some cases.

45:32  
Another possibility would be reference counting.

45:36  
So let's say that every object that can be allocated, every piece of memory that can be allocated is associated with a counter.

45:49  
And every time you copy a reference, this counter gets incremented.

45:55  
And every time a reference is removed, the variable is removed from stack or or transitive reference from another object is removed or something, this counter is decremented.

46:07  
Once the counter reaches 0, the object can be discarded, right?

46:13  
This is very efficient and very easy way.

46:15  
It's much easier to implement than the previous one than tracing.

46:20  
The tracing is quite complex and and you need to see all the operations of your application.

46:25  
This this can be implemented with a with atomic variables.

46:29  
For instance can be implemented without disrupting the the runtime of the rest of your application.

46:36  
However, there is this problem.

46:40  
Let's say that we have object P which has internal variable that points to Q, and the object Q has an internal variable that points to P These two bodies will support themselves indefinitely because thanks to this reference, this counter for this for this object will never decrease below 1.

47:04  
And thanks to this reference, this counter for this object will never decrease below 1.

47:11  
So these two objects will never be discarded, even though there there will be no external references to either of them.

47:18  
They will live there forever.

47:20  
So yeah, this is a imperfect solution for imperfect situations.

47:25  
However, these are still being used.

47:27  
For instance, PHP uses such a such a the garbage collecting system.

47:33  
Why?

47:33  
Because PHP was originally designed to handle short term scripts and scripts that run only as a response to a particular HTTP request.

47:42  
So it's expected that such a script is running for a very short time.

47:47  
So for a very short time, this should be a good idea.

47:51  
This should be a good garbage collector because even for a short time, you might need some garbage collection doing.

47:57  
But then again, if such thing happens like this cyclic reference, then what the heck, we will just clean it up, clean it up at the end when the script terminates.

48:09  
OK, so this is about garbage collection by the way C++ isn't today entirely without any support.

48:20  
But the approach languages like C++ or probably Rust are taking now is we are doing the garbage collection or let's say the memory disposal semi automatically.

48:32  
So it's not OK to rely 100% on the on the programmer.

48:37  
But what we are doing in in CC plus plus for instance what we are usually doing is that we are creating references like unique pointers.

48:51  
So instead of raw pointers which can be faulty, we are using these specific unique pointers which are in fact classes which are like smart pointers.

49:01  
And once a unique pointer cease to exist, it automatically dispose of the memory it's pointing to.

49:09  
So basically the programmer can't forget about disposing the memory.

49:13  
This smart pointer will take care of it.

49:15  
But the price you need to pay for that is that you are always using smart pointers instead of regular pointers.

49:22  
If you are always using smart pointers then you're probably fine and it's better alternative to the garbage collection from the perspective of performance.

49:31  
From the perspective of programmer convenience, a garbage collection is always better.

49:37  
OK, a few more additional words regarding the code compilation and program design.

49:43  
About portability, you've probably already written a few lines of code in Python, right?

49:51  
I hope so, because otherwise I don't know how you pass the first semester.

49:55  
So if you wrote some some pieces of code in Python, just out of curiosity, how many of you have coded in Python in more than one platform?

50:04  
Have you tried multiple platforms?

50:05  
Have you tried to run your code on at least Windows or Linux or Mac and Linux or something?

50:11  
A few people?

50:13  
Have you run into any troubles regarding running your Python on 2 platforms?

50:20  
I would guess none whatsoever, right?

50:23  
That that would be a fair guess, and I hope so, because Python you only.

50:27  
The only thing you need for Python is that you have a compatible runtime with a decent version.

50:32  
So if you're writing something which is compatible with Python 3.11, then your runtime is compatible with 3.11 or newer or something and then you're fine, no problem.

50:43  
If you're writing in C or C++, it's much more difficult.

50:48  
First level, you already witnessed in your Arduino classes.

50:54  
Different CPU architectures may involve your code on the lowest way.

50:59  
For instance, you have different type sizes, different in size.

51:04  
You all witnessed that you have int, which takes 16 bits on Arduino, but it takes 32 bits in a regular PC.

51:13  
Yeah, that's what happens.

51:16  
By the way, this can be easily fixed because some languages with a similar problem just said, hey, we are embedding the size of the type into the specification of the language.

51:27  
What the heck?

51:28  
Integer is always 32 bits.

51:30  
That's what C# or Java does.

51:32  
If you write in C#, your integer is always 32 bits wide, regardless of the architecture.

51:37  
So that that might not be a bad idea.

51:40  
On the other hand, in C++ there are ways how to do that as well.

51:45  
You can for instance use a specific type specific type aliases like U, int 32 sorry 32 T it's in the STD namespace.

52:01  
This is in the standard library.

52:03  
You can use it so there are specific type aliases you can use.

52:08  
And this type alias is fixed.

52:10  
It's somehow defined so that it's always 32 bit wide regardless of the architecture.

52:26  
You mean this one?

52:28  
This is the namespace separator.

52:31  
So this is the namespace and this is I'm accessing the namespace STD and this is the identifier within this namespace.

52:39  
So these two, these four dots are used to separate namespaces and identifiers or class names and identifiers.

52:48  
It's just a token.

52:52  
It's not that it has no special meaning.

52:54  
It's just a concatenation.

52:57  
So yeah, we have ways how to fix this in C++ as well, but you need to take extra care.

53:04  
Another level is the compiler.

53:07  
Unlike Python, which has, maybe that's not entirely true.

53:12  
Well, for PHP that's true.

53:13  
So unlike PHP, which has one interpreter, C++ has many compilers.

53:19  
Just from the top of my head, you have the GCC which is probably the most popular GNU Linux compiler.

53:27  
Microsoft has its own compiler.

53:30  
There are another compilers like Clank which is very popular nowadays.

53:34  
Intel has its own compiler.

53:36  
Many other players created their own C or C compilers, so you can use them all.

53:41  
And theoretically, since C has specification, has a very well defined norm for how to how to write things, how to write compilers, etcetera.

53:52  
They should be compatible, but they are not.

53:56  
So if you want to compile something for multiple of these compilers and usually you need to.

54:01  
For instance, I I used to write a lot of code that that needs to be compiled with the MCSV so we we can run it on Windows easily and also with GCC so we can run it easily on Linux nowadays.

54:14  
Nowadays, fortunately it's not that necessary because you can easily compile with GCC for Windows etcetera.

54:21  
But sometimes it's a good idea to support multiple compilers and it's pain in the \*\*\* because each compiler has its own flavor.

54:29  
Each compiler use slightly different approach.

54:31  
For instance, if you write in Visual Studio and you make yourself compatible with Visual Studio C and it works and everything's fine, and then you just move it to GCC.

54:44  
I tried to compile it for Linux and it just blows up into your face because Microsoft guys were very, let's say, more user friendly in the terms that they want to support less fortunate programmers.

54:58  
And they try to make the compiler very friendly in the way it forgives you if you just don't follow the standard rigidly and make some tiny mistakes.

55:10  
It says, hey, it's obvious what you mean by this code, so I'm not going to badger you about you.

55:16  
Forget this specific keyword that needs to be there because it's obvious you meant what you meant, so it's fine.

55:22  
But then you compile it with GCC and it blows up in your face because you didn't follow the specification of C++ rigidly.

55:29  
And he just said, hey, I'm expecting this keyword no matter what, so you need to place this superfluous keyword that need that sometimes needs to be there, sometimes not, but I want it there all the time.

55:41  
So sometimes it's difficult to make something compilable by multiple compilers.

55:46  
By the way, if you start with GCC, you probably will be fine, because I think GCC is one of the most pedantic compilers in the world.

55:55  
Then again, beware that some of the compilers like GCC have their own flavors, so it has some additional features and you should avoid using these additional features.

56:08  
And third level is obviously the operating system.

56:12  
I mentioned here that I wrote code that was expected to run on Windows and Linux.

56:17  
Sometimes it's easy because if you're doing something only using the standard library, like opening files, accessing the the console, etcetera, it's easy because it's the standard library takes care of all the nuances or all the differences.

56:33  
If you're not.

56:33  
If you need to do something more low level or something specific, then you need to write it in two ways.

56:39  
You need to write one piece of code for Linux and one piece of code for Windows, and then probably write some abstraction that covers both versions so you can use at least the rest of your application.

56:50  
You write the rest of your application only once for these.

56:56  
Sorry, I don't have libraries over there.

56:59  
So for these purposes it's a good idea to look for existing libraries like Boost for instance, which already take care of multi of portability of multi OS support.

57:11  
So this tedious work when you need to write your code multiple times for different operating systems was already done by someone else and you only need to use the library which already abstracts something away for you.

57:25  
Again, this is just a message.

57:27  
A portability is much more difficult in lower level languages than in Python.

57:31  
In Python you just, you're fine, you just need the runtime.

57:35  
In C it might be much more much more complicated by the way, for the binary compatibility.

57:42  
So this is not another level.

57:46  
On the previous slide I was talking about source code compatibilities or source code portability.

57:53  
So how to compile your source code for multiple operating systems, devices, etc.

58:00  
Sometimes you also need the portability of already compiled code.

58:04  
You're not always distributing your code as a source code by the way.

58:08  
Most people doesn't, unless you're writing in JavaScript or something.

58:11  
But usually you want to distribute your code as a binary code, as something that's not that transparent cannot be the the ideas cannot be stolen away just by copying in the code.

58:23  
So for that we need binary portability.

58:28  
For C, binary portability is very difficult to achieve.

58:32  
Yeah, we can.

58:33  
We can do that with some virtualization or with containers, but but it's difficult.

58:38  
But other languages like Java or C# already thought ahead and they embedded binary portability in their design.

58:46  
And here the binary portability is achieved by using by the usage of intermediate languages.

58:52  
Instead of compiling the source code of Java or C# directly into the instructions of the target CPU, they are compiled into the intermediate language which is called byte code in Java and cell in C You may already have heard about this.

59:10  
So this is almost like instructions.

59:12  
This is like a very close to assembly language.

59:16  
But this code actually it's, it's like an assembly language but for a virtual machine, not for actual particular CPU, but for a virtual non existing CPU which is called Java runtime or C runtime.

59:31  
And these environments, so Java or C# actually only implement a very tiny piece of software which is called the runtime which interprets.

59:43  
So it like executes virtually these instructions, it reads the byte code or it reads the cell and it tries to interpret each instruction in this byte code as it goes.

59:56  
So it's like a virtual CPU.

59:58  
And the only thing they, the authors need to port for different hardwares or different architectures is this runtime.

1:00:06  
The rest of the software, the rest of your software needs not to be ported because it's interpreted on all the machines in the same way.

1:00:14  
However, this is quite inefficient, right?

1:00:17  
Because if you really need to interpret every instruction, you need at least several instructions to perform the task.

1:00:26  
So it would be several times slower and nobody would want that.

1:00:30  
So there are techniques how to speed things up and these are just in time and ahead of time compilation.

1:00:37  
So they realized, hey, that that won't work.

1:00:40  
Just pure pure simulation, pure virtualization wouldn't work.

1:00:45  
So what they actually do is that they try to, they start running, sometimes they start running, especially in JIT your code in this virtualized manner.

1:00:55  
And then then they realize, hey, this code is run many times.

1:00:59  
So I just will compile, I will take the byte code and recompile the byte code into the code for the specific architecture I'm running on right now.

1:01:09  
So the compilation part of the compilation is differed from the compilation process of which programmer does during the development to the runtime process when the actual program is started.

1:01:21  
So nowadays, usually when you're starting an application application like in C# or Java, part of or sometimes entire code is compiled once more just when it starts.

1:01:35  
The head of time means it's compiled altogether.

1:01:38  
When the application is loaded.

1:01:40  
Just in time means that the small portions of code are being compiled on demand.

1:01:45  
But basically it's the same technique.

1:01:47  
It differs only in the timing when the actual compilation takes place.

1:01:55  
OK, that's about it.

1:01:57  
I think we covered all the topics we have prepared for the language and compilation session.

1:02:04  
Are there any questions regarding compilers, linkers, runtimes?

1:02:09  
OK, no questions so far.

1:02:12  
Fine, so let's move to another topic.

1:02:19  
Just a second.

1:02:19  
And another topic is the operating systems.

1:02:26  
Operating systems will be a slightly larger topic.

1:02:29  
We have scheduled 4 lectures.

1:02:31  
I hope we can squeeze it a little because we are running out of time as I mentioned at the beginning.

1:02:36  
So operating systems actually before I start operating systems is a very huge topic.

1:02:45  
So this is like an introduction.

1:02:47  
There is another another subject which which covered the this topic in more detail and where we can actually write piece of operating system kernel.

1:02:59  
Oh yeah, this was just a landmine I prepared for my colleague Jacob.

1:03:03  
So let's move on.

1:03:06  
Operating system is basically a piece of software that has two major roles.

1:03:10  
The first role is abstraction.

1:03:12  
Abstraction is very important because you don't want to design your software to be compatible with whatever possible hardware is out there.

1:03:21  
I will give you a let's say slightly nostalgic story that that's related to this.

1:03:30  
When I was young gamer I used to play games which first introduced more modern sounds.

1:03:37  
Nowadays you are very used to high defined sounds in your applications and and games and everything.

1:03:47  
And if the sound is in 3D and played on on 7 different boxes, that plays the sound or whatever.

1:03:56  
It's considered low tech, so yeah.

1:03:59  
But back in my days, the speaker of the of the PC was much like the speaker in your Arduino Shields.

1:04:09  
So it was just beeping and the very new piece of hardware that was emerging back then was a sound card.

1:04:18  
I'm not sure whether you know what sound card is, because nowadays sound cards are integrated parts of your motherboard or your laptop.

1:04:25  
So you don't even know they are there.

1:04:27  
It's it's like they are automatically always there.

1:04:30  
But, but back in these days, you need to, to buy a very expensive piece of hardware that needs to be added to your PC.

1:04:39  
And this specific piece of hardware was capable of playing sound in let's say normal way.

1:04:44  
So it was capable of playing music or doing realistic side effects, not just beeping.

1:04:51  
And the first games that start using these sound cards were like, you just open the game, you need to open the setup in the game.

1:05:01  
And then you you have like 20 possible sound cards or sound card systems and you need to select the right one.

1:05:09  
Usually they didn't have the one you had but you need to select the compatible one.

1:05:13  
And every game just re implemented its own driver for 20 different possible sound cards.

1:05:22  
And if they didn't, you were out of luck.

1:05:26  
You couldn't play the sound in your game.

1:05:29  
So it was very tedious for the game developers.

1:05:31  
It was also very tedious for game users because sometimes they didn't do it right or sometimes we have slightly different different cards and suddenly the sound wasn't working.

1:05:41  
Or worse, it was working but in a bad way.

1:05:44  
So it was making noise instead of sound.

1:05:46  
So that's like a world in which I was growing up.

1:05:51  
Fortunately for today, we use much more, better abstractions.

1:05:57  
And one of the roles of the operating systems is that no application, no game needs to implement its own drivers for sound cards, for video cards, for keyboards, etc.

1:06:10  
They are just using the abstraction provided by the operating system, which is well defined.

1:06:17  
The same for all your keyboards.

1:06:19  
You can imagine there is the same abstraction for all the audio cards.

1:06:23  
For all the sound cards there are, there is the same abstraction.

1:06:27  
The application just needs to ask the operating system to play some sound and the operating system do the difficult stuff like loading the right driver, communicating with the hardware in an appropriate way, etc.

1:06:42  
So this is the first important role.

1:06:45  
The second role is the resource management, because there are many resources that needs to be shared among applications in your system, in your PC.

1:06:57  
Again, back in my days, it was like you were always running one application.

1:07:01  
In MS-DOS, you just run one game and that's it.

1:07:05  
Everything else was suspended, everything else wasn't running.

1:07:09  
Only this one game was running and and that's it.

1:07:12  
In your regular Windows or micos or whatever system you're using, you're probably running at least 10 application simultaneously.

1:07:20  
Sometimes you don't even know about it.

1:07:23  
So even if you open multiple windows with browser or multiple tabs within one browser, that's already actually using multiple applications.

1:07:35  
And all these applications need to somehow share the resources they all have but one CPU.

1:07:42  
They all have but one main operating memory.

1:07:46  
They all have but one hard drive usually.

1:07:50  
So these resources needs to be somehow shared by these applications.

1:07:56  
And basically there are two main ways how to do that allocation.

1:08:01  
This works well with memory.

1:08:03  
Yeah, I have one application that needs 1 GB of memory over here, Another application needs another GB.

1:08:08  
So we just place that GB over there and we can manage to handle these two gigabytes separately, right?

1:08:16  
Other things cannot be shared this easily, like CPU.

1:08:19  
Yeah, technically you might have multiple CPU cores.

1:08:23  
So technically you can allocate 1 core for one application, another core for another application.

1:08:29  
That would work in some situations, some scenarios.

1:08:32  
But if you just open your regular Windows task manager or your or just list all your processes on your Linux or whatever, then you you can find out that there are 10s or hundreds of applications or processes or services running simultaneously.

1:08:48  
And it's not possible.

1:08:49  
It's not feasible that you would allocate one CPU core for each.

1:08:54  
That's just not possible.

1:08:55  
So we need time sharing.

1:08:57  
You just borrowing the CPU or CPU cores to individual processes, to individual threads actually for short periods of time.

1:09:08  
So they can compute something, they can do their stuff, but afterwards the CPU is taken away from them and it's given to another process.

1:09:16  
And this, if you do this fast enough, you will create an illusion that they are like sharing the CPU in a in a very fair in a in an every very efficient way.

1:09:29  
And also some parts needs to be abstracted.

1:09:31  
For instance hard drive or also network needs to be abstracted in a way that you are never using hard drive directly.

1:09:40  
No application.

1:09:41  
Well, there may be exceptions like if you have a system application for defragmenting the hard drive or something or fixing the errors on your file system, then yeah, these applications might use the hard drive almost directly.

1:09:54  
But usually normal applications doesn't access the hard drive itself.

1:09:59  
They are using the file system abstraction.

1:10:02  
So they are operating at the file level.

1:10:04  
They are reading files and writing files, reading directories, writing directories, but they are never accessing the hard drive.

1:10:11  
Actually, your application doesn't even know whether the file it's actually reading is on the hard drive or is being transferred via network to you, or it's being currently stored in the main operating memory and it's just making an appearance.

1:10:27  
It's a hard drive.

1:10:28  
All things are possible nowadays.

1:10:30  
So there's a specific abstraction and the application doesn't even take, it doesn't even bother itself with the details like where the actual file is being stored.

1:10:41  
So this is about operating system introduction.

1:10:48  
And before we get started, let's start from the CPU.

1:10:51  
And for that, we need one piece of information I've neglected to mention when I talked about CPU architectures.

1:10:59  
Well, actually, I didn't neglect to mention it.

1:11:01  
I didn't tell you this information deliberately, so you wouldn't be bothered with that.

1:11:07  
Before we actually need it, but now we really need it, so let's start with that.

1:11:16  
CPUs have at least two modes of operation, and by modes I mean privileges privileged levels.

1:11:25  
One level is for everyday regular applications, so it's usually called the user mode.

1:11:31  
So normal applications are running in like a sandboxed or a restricted regime of the CPU.

1:11:40  
They cannot do everything.

1:11:42  
They can do only the regular instructions.

1:11:43  
They can compute, they can access memory, they can do the regular stuff, but they cannot, for instance, access the system registers that tweak how the CPU behaves.

1:11:53  
For instance, they cannot handle things like interrupts.

1:11:58  
We get to interrupts later.

1:11:59  
So they cannot communicate directly with the hardware, from the CPU, with the external hardware.

1:12:06  
And besides that, there is this other mode which is called kernel mode or system mode or privileged mode.

1:12:13  
And in this mode, usually the CPU can do everything, it can communicate with hardware, it can execute all the instructions, no instructions are restricted, it can access even the system registers etc.

1:12:27  
And the tricky part is how to ensure a secure and efficient transition between these two modes.

1:12:34  
Actually the transition from the kernel mode to the user mode is quite easy because if you are in the kernel mode, your omnipotent, you can do everything you can you can.

1:12:46  
So if you can do everything then you can just say hey I'm restricting myself to the regular mode no problem.

1:12:52  
But the tricky part is how to securely enable transition from user mode to the kernel mode.

1:12:57  
Because if the application just can say, hey, I want to switch to kernel mode, then there will be no security at all.

1:13:06  
If every application can just switch to the kernel mode every time it wants and just continue executing its own instructions in privilege mode, then there would be there wouldn't be any security, there wouldn't be any reason for this.

1:13:18  
So what what's there is there is a trap.

1:13:21  
There is a specific instruction which is called syscall.

1:13:26  
It's.

1:13:26  
It behaves almost like a regular call instructions or regular jump instructions.

1:13:31  
Actually not not jump call.

1:13:33  
We need to return from this call.

1:13:34  
So it's it's like a call instruction.

1:13:37  
But embedded in this instruction is the increase of the privilege level.

1:13:43  
So within this call is also the transition of the CPU mode.

1:13:48  
So when you perform this call, it also jumps.

1:13:52  
It also switches to the kernel mode and the the catch is the tricky part is that you cannot call arbitrary function.

1:14:00  
You cannot call arbitrary piece of code.

1:14:03  
Actually you are not even specifying what you are calling.

1:14:07  
You are just performing this call.

1:14:09  
And the kernel was responsible for filling the right address of the function of the system function which is being called into some system register which cannot be altered from the user mode.

1:14:22  
Let me explain an example.

1:14:25  
Let's just perform something very simple.

1:14:28  
We are calling printf hello.

1:14:30  
So this is very straightforward piece of code, The most easy code I could probably write, the easiest code I could write in C this will translate into something like this.

1:14:40  
The call itself actually, well, this is not entirely true because this is actually call to a Lib C function which is somewhere in the runtime of the C, But somewhere in the print F embedded deep inside will be the system call.

1:14:58  
Because yeah, we need to perform at least one sys call so that we can take this data and put it out to the operating system.

1:15:05  
So the operating system will use it and write it to the stream, to the standard output stream.

1:15:12  
And here is how this sys call will look like.

1:15:16  
Before that, there will be some initialization.

1:15:19  
We place all the important stuff into registers and our operating system provides us with the guidelines, with the manual which operations of the system has which op code.

1:15:30  
So this Rix and RBX, actually RBX 1 means we are writing to the standard output, but the RAX is the right operation.

1:15:40  
So somewhere somehow in some manual, somebody wrote that if I want to print something out into a stream, I need to put 4 into RAX.

1:15:50  
So yeah, that's it.

1:15:52  
Don't ask.

1:15:53  
It's it's copied from the documentation.

1:15:56  
And these these additional parameters are the number of the stream to which we are writing, the address of the message we are writing, the pointer to our memory which we want to dump, and the length of the data that will be dumped into the written into the stream.

1:16:12  
OK, so we are putting the core arguments into the registers.

1:16:17  
And by the way, all the call arguments must be in the registers.

1:16:21  
And then we perform syscall without any explicit argument because the syscall will actually take this register.

1:16:29  
You haven't seen this one.

1:16:30  
This is a system register.

1:16:31  
It will take this register and take the address in this register and jump to this address whilst switching to ring 0.

1:16:42  
This is Intel terminology for the kernel mode.

1:16:46  
So they call.

1:16:47  
Ring 0 is the kernel mode, ring 3 is the user mode.

1:16:51  
They have a specific register where this number 0123 is stored, and in the past they have a.

1:17:01  
They thought it might be useful to have multiple levels, not just the kernel mode and the user mode.

1:17:07  
Sorry about some levels in between.

1:17:10  
So they they said OK the ring zero that the main level this is like a root level would be would be the kernel mode and the user level will be ring 3 and there will be ring one and two in between.

1:17:22  
But later they find found out that no operating system wanted this distinction.

1:17:28  
They either want to run application software or they want to run the kernel.

1:17:32  
So in the end we are no longer using the intermediate rings.

1:17:37  
I'm not sure even if they are supported in the hardware right now, but the terminology remained.

1:17:45  
So in here we switch to the kernel and call the kernel function.

1:17:51  
The kernel is responsible for handling whatever we put into the registers and performing the task we ask it to do.

1:17:58  
And then someplace in the kernel code it calls sysreturn which will actually jump back into our application while switching back to the system system mode user mode.

1:18:12  
Sorry, is this clear?

1:18:18  
So it's not that difficult from the perspective of the programmer, but internally in the CPU there is plenty stuff that needs to be done.

1:18:26  
And also this would be very tedious for us to write.

1:18:29  
So we are usually not writing this.

1:18:31  
This is written.

1:18:31  
This is embedded somewhere in the libraries of the operating system where.

1:18:46  
Well, what do you mean by security reliability?

1:18:50  
Could you could you like elaborate on that?

1:18:53  
Could you present some vector of attack that that could be exploited via this?

1:18:56  
Like surely if any random like what is something in state for example, like what it would be interpreted as characters that we just printed out the The thing is that this register holds the address in memory where we are jumping.

1:19:25  
This address in memory isn't accessible by the by the memory of the operating of the application which is running.

1:19:32  
So the application cannot override this address nor the code which is at the address.

1:19:39  
That's the whole point.

1:19:41  
Yeah, I see where you're taking this.

1:19:44  
So there is another mechanism which protects the memory, and we haven't talked about it yet.

1:19:49  
We started with the CPU, so obviously we need memory protection as well.

1:19:53  
The application cannot just write everywhere on the memory.

1:19:56  
The application has its own memory and it cannot write anywhere else.

1:20:00  
That's the whole point.

1:20:01  
So the kernel memory itself, it's protected.

1:20:04  
It cannot be accessed.

1:20:07  
So yeah, if it can be accessed, then yeah, we can inject something into the kernel memory and then perform the jump.

1:20:13  
That would be that would be disastrous.

1:20:15  
But thanks to the memory protection, this is not possible.

1:20:23  
Yeah, I think this would be a good example to to cover for the end.

1:20:28  
So just to give you some notions, what actually are and aren't system calls, let's just take this piece of code and examine this from the perspective of what syscalls will be performed when we are doing this code.

1:20:44  
By the way, just to explain for those who are not familiar with streams, we are reading this file, this text file, we are reading it into a string and then we just print it out to the standard output.

1:20:57  
This is just an elaborate way how to read a file to a text file into a string.

1:21:03  
Yeah, it's slightly more difficult in C than it is in Python to read the whole text file in a stream in a string, but yeah, this code will take care of it.

1:21:17  
So I think I will need the example, hopefully.

1:21:32  
Here we go.

1:21:34  
So this is again the code.

1:21:35  
I think I can scroll it out.

1:21:37  
This is the contents of the text file.

1:21:42  
This is actually if you want to replicate, replicate this.

1:21:45  
This is what we did.

1:21:47  
I just compiled the code using standard GCC G because it's C code and I run S trace.

1:21:54  
S trace is a Linux tool that runs an application.

1:21:58  
I wrote it as a prefix of the application itself.

1:22:00  
I run S trace app and it runs the application and it monitors all the system calls that they are performed.

1:22:08  
And I just save this S trace, the output of this S trace into the file and this is the dump of this file.

1:22:17  
So let's take a quick look on what's going on here.

1:22:21  
So first operation is some exec and yeah, sorry, this is a confusion.

1:22:30  
I somehow I renamed it somewhere.

1:22:34  
So I I wrote yeah, it's it's main, but I wrote up here, but it was main.

1:22:40  
My mistake.

1:22:41  
Just imagine that there was main over here.

1:22:44  
Sorry, I I just mixed up when I was fixing the the stuff for the slide, but it's not that important.

1:22:50  
So this is basically we are loading entire main executable into our memory space and we are starting executing it.

1:22:59  
So this is how every application actually begins with loading its own code.

1:23:05  
Yeah, and we need to access you.

1:23:08  
You may notice that we are accessing several things like, ETC.

1:23:13  
Refers to files that are that holds the configuration.

1:23:16  
So we need to find out where are the loading library path is etcetera.

1:23:21  
And we need to load several libraries.

1:23:24  
This is the the runtime for standard C++ DLL actually.

1:23:29  
SO dynamic, dynamic library.

1:23:31  
And there are several more.

1:23:36  
All this stuff we are just loading.

1:23:38  
We are just loading libraries.

1:23:40  
By the way, do you know what MMM map is?

1:23:44  
Anybody knows?

1:23:45  
This is memory map and it's used to take part of a file and map it into memory, like load a huge block of file in a memory.

1:23:56  
It's faster to map this stuff into memory than to use standard read and write operations which are much slower.

1:24:04  
And we are finally approaching the last part.

1:24:08  
So once all the initialization and take a note that there was plenty sys calls just to initialize, just to start up the application and we are finally get to the stuff which is of our particular interests.

1:24:22  
So let's start over here.

1:24:24  
This is open at.

1:24:26  
Yeah, this is just some note, some shorthand for open operation with some specific attributes.

1:24:31  
So we are opening the atxt file in the read only mode and we are getting #3 as a file handle.

1:24:41  
Why the heck we are getting 3?

1:24:42  
This is the first file we are opening.

1:24:44  
Actually, that's not true.

1:24:45  
We have opened many files as libraries before.

1:24:49  
But you can see that the first file we are opening over here also yields 3.

1:24:56  
Why the heck the first file, the first handler we are getting is 3?

1:25:01  
Why not 0 Or OK, 0 might be reserved for for failures or something.

1:25:06  
So why not one or two for some for for?

1:25:13  
Well actually they are not reserved, they they just already exist.

1:25:17  
How?

1:25:17  
How that's possible that we just started an application and there are also already file handlers that exist?

1:25:29  
Exactly.

1:25:30  
There is standard input and output.

1:25:32  
Actually there are three input, standard output and error output 01 and two.

1:25:40  
So the three is the first available file handler there is.

1:25:46  
So This is why every open gives me 3.

1:25:49  
And by the way, if you follow up, every open has its corresponding close.

1:25:53  
So every time we open something, it's closed so this file handle becomes available again and the next open will also get 3.

1:26:02  
That's why we are getting 3 here.

1:26:05  
When we are finally opening our own text file, right then we perform read 33 is the file handler.

1:26:15  
This is the result of the.

1:26:17  
This is the buffer which will be filled.

1:26:19  
This is the maximum amount of the buffer.

1:26:20  
The read will read up to 8000 bytes, 8 kilobytes and it yields 21 which says hey we only find 21 bytes in this in this particular file.

1:26:31  
So hey, I would like to satisfy your L8 KB, but I don't have more stuff to fill you.

1:26:37  
So this is the read which we get.

1:26:40  
Then there are some stuff which is not that important and right, we are writing back the contents of this buffer with the appropriate size now, because we already know how many bytes we read and that's it.

1:26:53  
Why the hell we are writing to 1?

1:26:58  
Yeah.

1:26:59  
1 is standard output, 0 is standard input, 1 is standard output, 2 is standard error.

1:27:03  
You probably know that from Linux classes already, right?

1:27:06  
Have you have you already redirected your outputs to standard output or, or error output?

1:27:11  
Yeah.

1:27:11  
So so you already know that one is standard and then 2 is error output.

1:27:15  
So that's it.

1:27:16  
This is the yeah, this is the example of a strace.

1:27:21  
And I've just run out of time today and we agreed that I won't be, at least I will be trying not to prolong the, the lectures too much.

1:27:31  
So as I announced at the beginning, some, some people weren't there weren't here at the beginning.

1:27:37  
So let me let me repeat that due to my absence last week, we created, I created an unfortunate situation because we were already very tight in our schedule.

1:27:50  
We were already behind our schedule.

1:27:53  
Unfortunately this year we have only 12 lectures and the stuff the slides are to 12 1/2 lecture.

1:28:01  
So we are one lecture and a half about behind.

1:28:05  
So we agreed that most people would like to have an extra lecture instead of prolonging all the lectures every week.

1:28:14  
So I will write to you and with some poll where would be the best time having this extra lecture.

1:28:22  
I would guess there are two options.

1:28:24  
One option is that we somehow fit it in the the vacant space in the either 1st or 8th May because the those weeks are the the weeks where the holidays are.

1:28:36  
Or we will do it some sometime at the end, because in the end, we will know how much time we, we actually need to, to finish everything.

1:28:44  
And it will be probably either the one week after the semester.

1:28:48  
So one week, the first week of the, of the exam.

1:28:51  
Or sometime at the end of the semester.

1:28:54  
I will investigate possible options and I will send you a poll which option is the best for you so that most people could come.

1:29:04  
So thank you for your attention and hopefully nothing will disrupt our flow next week.

1:29:09  
So thank you and and see you next week.