**2025-03-27-memory2-lang1-en**

0:00  
Mistaken.

0:00  
I've ended on this slide, so we didn't do anything.

0:04  
I haven't told you anything about caches, so let's pick up here.

0:08  
Caches are perhaps the most important piece of hardware and you will hear about them all the time, especially if you browse into operating systems or into high performance applications.

0:19  
Because every time you need to do something that has something to do with with memory and you either need to do it in a more secure or more sensitive way or you need to do it in a high performance way.

0:31  
You're really concerned about the performance of your application?

0:35  
Then memory caches are the first culprit, are the 1st place to start to look for optimizations, etcetera.

0:42  
So what are caches?

0:44  
First of all, cache.

0:45  
The term cache is a general term.

0:47  
So there are various types of caches.

0:49  
For instance, your browsers have caches because if you open the same web page and I wouldn't venture a guess what that page might be in in your browser, that it's a good idea that the browser keeps the code of the page cached inside inside the browser or on on your local hard drive.

1:06  
So it doesn't have to be loaded every time you open that particular web page.

1:11  
Same goes with the memory.

1:12  
So some parts of memory can be cached much closer to the CPU.

1:16  
So every time you need to access this particular memory, the CPU doesn't have to make the long request to the main memory, but it can access the data much faster.

1:27  
So basically cache is in some structure, or in this case a hardware that holds some data.

1:32  
And the important part is that, sorry, yeah, the cache is used only when it speeds the things up.

1:44  
So cache is somewhat a transparent mechanism which is only there to speed the things up.

1:49  
So assume that the loading of the data or sometimes it might be you need to compute the data, you need to compute the result and the cache is there just to memorise your results for your common answers.

2:02  
So every time you need to do something which is slow or expensive, then cache is there to help you with that to give you the answer more quickly.

2:09  
However, the cache can be removed anytime and the system should work the same.

2:15  
The performance may be hit severely, but the the semantics of the system or the application or the hardware in this case would remain the same.

2:23  
So caches are somewhat transparent to the application.

2:28  
The generic cache operation is that you make a request for some data, so there is some identification of the data.

2:34  
In case of memory, it's the address, in case of the browser example it would be the URL, right?

2:39  
So it's just sometimes unique identification of the data you need to request and you just request the data and the cache.

2:48  
If the data are placed in the cache then they are returned.

2:51  
Otherwise you you get nothing.

2:53  
So you need to make the long trip to the memory.

2:56  
You need to load the the website from the Internet.

3:00  
You need to compute the data.

3:01  
Whatever you need to do to get the data, the expensive the long way in the CPU.

3:08  
The caches are mainly there to hide memory latency.

3:12  
The latency is the delay between the memory requests and the answer when the the operand or the instruction is filled with the data from the memory.

3:24  
This whole concept is based on something which is called locality of reference.

3:29  
That's just a term that describes the assumption why we are placing.

3:33  
Well, not we are.

3:34  
The vendors are placing the caches on the CPU.

3:38  
The notion is that when your application runs, it very often access the same small area in the memory over and over again.

3:47  
Yeah, this assumption doesn't hold for every application.

3:52  
Obviously it can't.

3:53  
However, for many situation it does.

3:56  
Just assume how large is your program in the memory.

3:59  
The program isn't usually that large, at least not the part that's currently running.

4:03  
So the the currently running, but the hot part of your code usually fits several kilobytes or 10s of kilobytes.

4:11  
If you're doing some intensive computations, then your hot data, the data you are currently processing also usually fits a few kilobytes or 10s of kilobyte sometime, sometimes it's megabytes.

4:22  
Yeah.

4:22  
And of course you get stupid algorithms that just wander around the memory in a random fashion.

4:29  
And yeah, you cannot do anything about those.

4:31  
However, most of the algorithms are tend to access the data which are close together in the memory.

4:38  
And if that is so, then caches will help us sincerely severely, because the the caches will localize will give you the localized memory you are accessing.

4:49  
The small portion of the memory you are accessing will bring it closer to the CPU so you can access it in a much faster fashion.

4:57  
So the CPU cache operation is basically embedded into the load operation of the OR the load instruction which is in the CPU.

5:06  
So when the every time when there is a request to the memory.

5:09  
And remember, if there is a load store architecture, the only access is made by load operation and possibly stores.

5:15  
Of course if you have a X86 architecture for instance, some of these memory operations are embedded in other instructions.

5:24  
Usually you are using move instruction to move data from the memory to the register or vice versa.

5:28  
But there are some other instructions that can accept reference to the memory as an operand.

5:34  
So all these instructions that access memory trigger this request of the data and then we then the cache just search itself whether the data are in the cache.

5:47  
By the way, in CPUs there are multiple levels.

5:49  
You already noticed that when I was talking about the CPU design.

5:53  
Usually we have like 3 levels in contemporary CPUs and the each level is like the L1.

5:59  
The first level is the closest one to the CPU which is smallest and fastest.

6:03  
L 3 is usually shared among all the cores, it is largest and slowest.

6:08  
So it's like there are three levels to make this more convenient.

6:12  
Because it's very difficult to create a cache which is both fast and large.

6:18  
That's why we have very small caches, very close and very fast, and larger caches slightly further, slightly slower, but much larger.

6:26  
So yeah, this request is repeated on all levels.

6:31  
So first you are going to the first level to L1 cache.

6:34  
If the data are there, you get the data, if not, you are going to the second level.

6:38  
You asking the second level of of the cache for the data.

6:40  
If they are there, you get them, if they're not, you are going to the last level, to the third level.

6:45  
If if the data are there, you get them and if not, you need to make the long trip to the main memory.

6:52  
OK, so this is basically how caches work.

6:54  
One way how to remember this is if you don't remember what caches are.

6:59  
Well, caches are something you already used during your high school, right?

7:03  
So why waste the long trip from your bed into the into your wardrobe if you need to fetch a new fetch?

7:11  
I wouldn't say a fresh set of clothes.

7:13  
I said less dirty set of clothes or something like that.

7:16  
So in this case, you don't want to waste the long trip to the wardrobe, right?

7:21  
So you just access the pile of the laundry just next to your bed, which every bachelor can access in constant time, right?

7:28  
Because we have some special abilities.

7:31  
Well, I'm married, so I'm not no longer bachelor.

7:32  
So sorry.

7:33  
But when I was a bachelor, I, I could use this, this unique ability to just search my pile of, of clothes and access any clothes which it was there in constant time.

7:44  
So this is this is a cache, this is the memory, and this is the CPU in our sense.

7:53  
Few things specifically related to the caches in the CPU.

7:57  
So first of all, what?

7:59  
What are we actually caching?

8:00  
Because it would be very inconvenient to cache every single bit or every single byte.

8:05  
That wouldn't be easy because yeah, we can technically address every single byte of memory we can, but it would be very impractical to cache them individually because the overhead would be enormous.

8:17  
And you probably already remember that we are accessing the data not by bytes, but more usually by words like integers.

8:26  
So normal CPUs are incapable of accessing anything smaller than 32 bits.

8:32  
Normal CPUs like in your laptops not.

8:34  
Not like Arduino, right?

8:38  
So basically it will be very impractical to make this too fine grained.

8:43  
So what we define is that there are cache lines.

8:45  
Cache line is a small block of data and we are always caching the whole cache line.

8:50  
So we divide the memory into smaller box.

8:53  
For instance, on most architectures like X86 architecture, I believe all your laptops you have here has cache line of of the size 64 bytes.

9:03  
It's very likely.

9:04  
Yeah, some architectures have 32, some architectures might have 128.

9:08  
But roughly if you say something around 64, you're basically never wrong, right?

9:13  
And it's not not that important whether it's 32 or 64, it's the block about this size.

9:18  
So again like in the memory allocation, this is the block which is not too large.

9:23  
So it's somewhat fine grained, but not too small not to produce too large overhead.

9:28  
And also the size is selected so that the sketch line can be easily fetched from the physical memory.

9:34  
So usually the physical memory is operating somewhere close to this range.

9:39  
So fetching 64 bytes from the main memory would be like on the similar speed as fetching anything slower.

9:48  
So basically thanks to the parallel buses when you're serving data from main memory, you you probably remember that from last week.

9:57  
What was the size of the bus of one dim dim chip I I demonstrated on the slide.

10:03  
Does anybody remember?

10:06  
Can I give you a hint?

10:07  
There were three bits dedicated for the bus address for the byte address.

10:11  
So if we have 3 bits for the byte address, that's that's a byte.

10:18  
So from each chip you can get 8 bytes.

10:20  
If you have double channel, you can get 16 bytes.

10:23  
If you have more channels, you can get more bytes.

10:26  
Plus, thanks to some specialized circuitry as as you should remember, if you just if you just accessing the data which are near to near together in the column order not in on the different rows, then you can get them much faster.

10:42  
So reading consecutive data from the memory is much faster than just wandering around the memory.

10:48  
So if you want to write read consecutive 64 bytes from the memory, it's it's very fast.

10:53  
OK, so This is why they choose a size which is somewhat close to what's good for the main memory buses.

11:01  
And of course it's aligned obviously because you want to load the data, you want to address the data in the fashion that you can just drop the lowest bits of the address and the upper bits keep the address of the whole cache line.

11:13  
By the way, how many bits can we drop from the address if we are, if we want to address 64 byte blocks, how many bits on the in the address would be 0 in the end of the address if we have aligned address to the 64 bytes, that's right, 64 two to the power of 6 is 64 exactly.

11:34  
So by the way, so if we are addressing cache lines, we can forget 6 bits because we know they're they're zero.

11:41  
That's a trick that can be used in in in caching.

11:46  
Cache hit is a terminology when you searching the cache and the data are there.

11:51  
So you've got to hit the data are there.

11:53  
You can't get the data fast.

11:56  
Cache miss is the opposite.

11:58  
If you if the data are not there, you you miss the opportunity for speed up.

12:02  
So that's cache miss.

12:03  
And in that in that case you need to search the higher level of cache hierarchy or you need to access the main, main memory.

12:13  
By the way, the cache miss is somewhat more complicated because every time cache miss occurs, we would like to make sure it it doesn't hit miss again.

12:23  
Because usually when you access some data, it's very likely you are going to access the same data or the next byte.

12:29  
So it's a good idea to load the whole cache line anyway very soon.

12:34  
So basically every time a cache miss occurs, what you want to do is to populate the cache to save the data in the cache.

12:42  
So next request for the same address or at least next request for the same cache line wouldn't miss, right?

12:49  
However, for that there is one one cache.

12:52  
We need to select a victim.

12:55  
Either there is a free cache line so we can just put it somewhere, but usually sorry sorry sorry, it's probably on the next line.

13:06  
But usually we have a problem because the cache is full, so we need to select a victim.

13:12  
We need to select someone who will be removed from the cache and the data will be loaded instead of the of the removed record.

13:21  
By the way, this is even more tricky because you can access memory from multiple CPU cores.

13:28  
That's a completely normal thing to do nowadays because you have multicore CPUs.

13:32  
All of you here sitting here probably have multicore CPUs.

13:36  
It it would be very unlikely that someone here has a single core CPU.

13:40  
So there is a another protocol that needs to be put in place to make sure that all the CPUs are accessing the caches in some coherent manner so that the CPUs cannot break anything.

13:53  
For instance, you cannot make 2 CPUs right into the same cache line at the very same moment.

13:59  
That's something you need to prevent because that would be disastrous.

14:03  
On the other hand, when multiple CPU cores are reading the same cache line, there is no problem there because reading is fine.

14:09  
Yeah, you can, you can share the cache line and and you can read.

14:12  
For that there is a specialized message protocol.

14:14  
I'm not going to into the details because it's it's quite technical, but just be assured that there is some specialized hardware protocol that makes sure that the caches distributed among multiple CPU cores will stay coherent.

14:30  
Basically the CPUs can communicate over the over the memory buses and or they are snooping on the buses and they can alter the state of their internal cache lines if another cache, if another CPU core cache is gets populated by a particular cache line, for instance.

14:47  
So there there is a protocol for that.

14:50  
OK, by the way, yeah, I forget to read this one line when when we are selecting the victim for the for the new cache line, don't remember that we cannot just sacrifice the old cache line and just drop it.

15:07  
Sometimes the stored cache line can also contain modified data.

15:10  
So the caches are not only for reads, they are also for writes.

15:14  
If you're writing data, if you're writing some variable, some, some data in the memory which are already in the cache, the modifications take place place in the cache and sometime later they will be propagated to the main memory.

15:26  
So there there is this small detail over here which cannot be overlooked.

15:31  
Just remember that the rights needs to be also handled properly properly in the caches.

15:38  
By the way for normal application.

15:40  
This is some some general statistics that was made by some scientists.

15:45  
If you just take overview of contemporary pieces of software that are usually running your CPUs, they usually have a very high success rate, very high hit rate for the caches.

15:57  
So this tells us that the caches are doing their job properly.

16:01  
It's a good idea to have these caches.

16:04  
This code, all this code that that succeeds on the cache hits would run much, much slower if the caches wouldn't be, wouldn't be there.

16:13  
OK, this is just the the to remind you about the levels of caches.

16:18  
So first of all, the first level of cache, which is smallest and closest.

16:22  
By the way, this level is divided into two parts.

16:25  
I didn't emphasize that when I was talking about CPU.

16:28  
So level 1 cache is actually divided into instruction cache and data cache.

16:33  
The instruction cache is used only for loading instructions, for loading the executable code and the data cache is for all other memory operations.

16:42  
L2 cache is shared for both memory for all the memory, so for both for the code and for your data.

16:48  
And L3 cache is also shared not only for instructions and data, but but also it's usually shared among the cores.

16:57  
Today.

16:58  
Usually the L3 is chopped into some directories into some fragments and each core somehow manages its own fragment.

17:06  
But other cores can access a fragment of an neighboring or or any other core.

17:11  
So it's much more difficult than that.

17:13  
But you don't need to know these details, it's just important thing you know that usually there are like 3 levels of caches.

17:19  
By the way, L1 has roughly today something like 32 kilobytes, so it's not that large.

17:25  
On the other hand, 32 kilobytes of instruction case should be enough to keep all your hot code.

17:30  
You usually don't use more than 32 kilobytes of your contemporary running code for the data.

17:36  
It's much smaller because 32 kilobytes is not that much.

17:40  
But again, then again, it's very fast.

17:42  
It's like it can be accessed with within units of cycles, so it's not as fast as registers, but only a few times slower than registers.

17:52  
So it's very fast.

17:54  
L2 cache is somewhat slower, like again several times to L1 cache, so like 10 times slower perhaps than the registers roughly.

18:03  
I'm not giving you the exact numbers, but again, then again, L2 is much, much larger.

18:07  
On current servers we have like 256 kilobytes of L2, but contemporary CPUs can easily have megabytes of L2 cache and L3 cache again is much, much larger.

18:19  
It has today I think in the the maximum is hundreds of MB.

18:22  
I'm not sure if maybe even more.

18:29  
Sorry, what?

18:30  
What?

18:31  
This is just a general, this is general depictation of execution unit.

18:38  
So this is just a very small block which which will contain the arithmetic logical units, the the float floating point units, everything that computes something.

18:48  
So all the executive stuff is here.

18:51  
Yeah, it's not, this is more important from the perspective of cashiers.

18:54  
This is just actually, this is much larger that this much more detailed.

18:58  
You, you remember the the diagram with the pipes.

19:01  
So there are many types of units and they are all in here like just concatenated though.

19:07  
No, no, no, it's not there.

19:10  
Well, it should be in blue if it's UU plus it has to have some stars around.

19:16  
So no, no, this is not a European unit.

19:19  
OK, so how the caches are implemented?

19:21  
This is quite important because you need to understand the reason why the caches are so small when they need to be fast.

19:29  
Well the reason is they using something special called associative memory.

19:35  
You know the regular memory is address by address, right?

19:38  
So by an index by an offset.

19:40  
So you have a number that tells you which memory block which which byte or which sequence of bytes depends on how wide is the word you're accessing the memory.

19:50  
So the address will tell you where to look, which row to activate, which column to activate, where you read the data.

19:57  
So it's very efficient because you don't have to activate the whole memory, you activate only very small portion of the memory and the portion is deduced from the address.

20:06  
So This is why the memory is organized this way.

20:09  
The associated memory isn't address at all.

20:12  
You have a key value store.

20:14  
So basically it's something like a dictionary in Python, just not.

20:19  
Implemented the same way just to give you an idea notion.

20:23  
So basically you have something which is called key, an identifier and the value the data, in this case a cache line or actually this will hold for the browser example as well.

20:34  
The key would be the URL and the value would be a HTML document for instance.

20:40  
OK, so this is the same, same idea.

20:43  
However, in memory, the key is actually the address.

20:48  
So yeah, it's it's tricky because it's like a copy of the address.

20:50  
Actually, it's not the whole address.

20:52  
We don't have to copy the whole address because I already told you that exactly the last six bits are always 0 because it's address of a cache line.

21:04  
So we don't have to put all 32 or or 64 bits.

21:07  
We can, we can put some bits away and also we are usually cropping some bits from the top because we are not making the cache fully associated, but that's another story.

21:16  
But basically you store the address as the key and the value here is the cache line plus some additional information.

21:25  
For instance, you need to keep the state of the cache line for the massive protocol, for the coherence protocol.

21:30  
So some additional bits needs to be placed there for for management reasons.

21:35  
OK, the trick is that to search, sorry to search this memory, it's very expensive because here I know the address.

21:46  
So I'm if I say OK, my data are on the address, let's say 4096.

21:52  
So I know I need to activate only this part of the memory so I can access I can power only this memory cell.

22:00  
Technically I'm I'm lying to a bit because DRM doesn't work that way.

22:03  
But basically I'm I don't have to power the whole memory the full I need to access only one memory cell here.

22:11  
I need to activate them all at once and broadcast to them the address and ask them all at once.

22:18  
Hey guys, does anyone here has the number 4096?

22:23  
Hey, does does anybody have it's the same here?

22:25  
Because if I give you everyone one piece of my memory, I can just broadcast the request.

22:31  
Hey, does anybody has this piece of memory?

22:33  
And only the person who has that will answer me.

22:37  
Yeah, I got it.

22:38  
And if nobody answers, I know it's not there, but I need to activate you all at once here.

22:44  
I need to activate only oneself.

22:45  
That's why this memory is so expensive, because it's more much more power expensive to perform such such a request.

22:52  
That's why we need to keep the caches much smaller.

22:54  
And there are other tricks, for instance this.

22:58  
This is something which is called fully associated cache, which means any any address can be placed on any of these positions.

23:10  
That's that's the best cache we can have and also the most expensive one in the terms of implementation.

23:17  
So they usually using some which something we just call N way associative cache, which means the address cannot be on any arbitrary place in the cache.

23:26  
But we already know that the cache is like divided into smaller groups and we know that some addresses using some hash functions, some some straightforward hash function, some addresses can be placed only to some small portion of the cache.

23:39  
So we don't have to search the, the whole cache, we are searching only a small part of it.

23:43  
So there are some tricks, hardware tricks, but they are not important from this perspective.

23:48  
What's important is we have some clever circuitry that does that for us.

23:53  
And it's, it's addressed by keys, which are basically cropped addresses and the values, remember this is the cache line, OK, this is how caches basically work.

24:04  
And finally, final part, final piece of the puzzle to the memory is the NUMA systems.

24:10  
I already told you that the multiprocessors are somewhat tricky because of the caches.

24:15  
So we need to have a protocol that that keeps the caches coherent among amongst multiple cores.

24:21  
But that that's not it.

24:22  
We have more complex systems on on our hands.

24:26  
If you're in your, if you have a single CPU core in your laptop, that's pretty straightforward because this is usually it has some single internal bus.

24:36  
Remember the, the circular bus inside the, the Coffee Lake example I, I showed you last week or the week before.

24:44  
So there is someone system internal bus and it's attached to the all CPU cores and it's also attached to the system agent which holds the memory controllers.

24:53  
So basically all the CPU cores are accessing the RAM via single bus.

24:59  
Yeah, there, there might be some redundancy, but that's not important right now.

25:03  
What's problematic is that if you have too many of these cores, then this bus gets overloaded.

25:09  
That's why they come up, come up came up with tricks like a circular buses or in even more more modern CPUs which have much more CPU cores, they are using a grid bus.

25:20  
So basically the cores are organized in a grid and there are row buses and column buses is much more complicated, but technically the bus can get overwhelmed with the requests to the memory.

25:31  
So it doesn't scale well, scale well.

25:34  
What can we do?

25:35  
One one option what we can do is to divide our system into multiple CPU, not cores, but the dies, the sockets, which means we have multiple CPU units, multiple CPU chips technically, which are communicating over some external buses, right.

25:53  
And the benefit is that each of these CPUs, these might these were denoting in my previous speech CPU cores.

26:00  
These are really CPU chips, so technically inside might be multiple CPU cores, but that's not important right now.

26:07  
What's important is that they communicating via external buses and each CPU, each CPU die, each CPU socket has its own memory.

26:17  
The benefit is that when the CPU is accessing its own memory, it's much faster because it it doesn't get clocked too much with the other CPUs.

26:26  
However, it would be very impractical for the programmers if every CPU, and I mean CPU die would be would could access only its own RAM.

26:36  
It would be very difficult to design A software that would run on such a system.

26:39  
So these NUMA systems actually make the whole RAM visible to all the CPU cores.

26:46  
So from the perspective of your operating system, from the perspective of your application, you have 4 pieces of RAM which are usually concatenated one after another, or they may be interleaf or that that's not that important that they are somehow combined together.

27:00  
So if you have like 4 times the the size of RAM or the combined size of the RAM of all the CPUs and each CPU core can access each other's RAM, which means there has to be additional protocol that allows the CPUs to communicate one with another requesting memory.

27:21  
Technically, it's just an extension of the massive protocol.

27:24  
Basically, they're using the same principle.

27:26  
So if this CPU needs to access this RAM, it sends a request for the RAM and this CPU and the other CPUs, remember that this cache line, this piece of RAM is stored, is temporarily cached on this CPU.

27:40  
So technically it's just an extension of the same protocol, but basically it's much more slower because in this case, in this previous example I was giving you, the system buses were internal.

27:51  
They are much shorter, much smaller, but even when they are smaller, they can communicate with a short latency.

27:58  
By the way, as a as a voluntary home example, a home assignment, you can just compute how fast the the light travels.

28:06  
If you have two circuitries, 22 chips 30 centimeters apart, for instance.

28:12  
So just imagine that you have two CPUs or CPU and A and a associated device or something 30 centimeters apart.

28:21  
And imagine that they are communicating using some optics or something which is slightly faster than than electric wires.

28:28  
So they can communicate at the speed of light.

28:31  
So compute the latency.

28:34  
Yeah.

28:34  
So I, I believe everybody can compute it within a minute.

28:39  
So I, I'm not giving you bother to, I won't bother you too much with that anymore.

28:43  
OK.

28:44  
It's just to give you notion, sorry.

28:47  
If the bus, if the bus is internal, it might be slightly faster than if these buses are external.

28:55  
Nevertheless, these systems are normally used widely deployed and usually we are using in our servers NUMA servers which has two or four CPUs.

29:04  
That's pretty normal.

29:07  
OK, this is pretty much it.

29:10  
This will conclude the talk about the memory.

29:12  
Are there any questions regarding memory, memory organization, memory access, caches, whatever.

29:20  
OK, let's move on to let's move on with another topic.

29:24  
And the follow up topic is programming languages one second.

29:37  
So programming languages, yeah, obviously this is a broad topic, so maybe this should be called something like an introduction to compilers and runtimes.

29:46  
So we are not going to talk about various programming languages.

29:50  
We are going to talk about the support various programming languages has from the compilation part, from the runtimes and from other stuff.

29:58  
So let's start with the compiler or the compilation process.

30:02  
Basically, you already know, or should know at least this interpretation of a compiler.

30:08  
So basically compiler is responsible for turning your software, your written pieces of code into something that's that can be absorbed by the CPU.

30:19  
So into the machine code, into the instructions encoded in binary so they can be intermediately interpreted by the CPU.

30:30  
Yeah, here the compiler is depicted as a black box because many people perceive it that way.

30:35  
And technically, if you're just a user, it's a correct, correct view because you basically take your source code, put it into GCCMVC, whatever compiler you prefer.

30:46  
And then you either get the executable, which you can immediately deploy on your operating system.

30:51  
So it's given to the CPU to execute.

30:53  
Or sometimes, of course, if you make a mistake in your source code, if ever, if you typed it incorrectly, if cannot, it cannot be translated into the instructions.

31:03  
You will get a set of error messages.

31:05  
So you can, based on them, you can refine or fix your code and then recompile it.

31:09  
That's it.

31:10  
By the way, in the first semester, you weren't doing this, right?

31:14  
Because you just have the Python script and you just run Python, your Python script, and then that was it.

31:20  
So there was no compiler, right?

31:23  
Wrong, actually.

31:24  
The compiler was there, you just haven't seen it first hand.

31:28  
The compiler was embedded in the Python runtime.

31:32  
Every time you run your script, there was compiler present who actually chew up on your code, spill out the instructions, and the instructions were immediately executed so the runtime did the compilation for you without actually telling you.

31:47  
So it's like a one lie to the baby switch starting with a with a simple languages.

31:53  
No offense man, it's just like a the C guys are making fun of Python guys, that's all.

31:58  
That's just this funny stuff.

32:01  
So basically it's just something that's hidden from the regular user so the regular user doesn't have to be bothered with with intermediate steps like compilations.

32:09  
Now, since you've grown up, you're given C or C and in C or C, and actually the same goes for for Java or C#, because normally most modern applications, modern languages use this compilation step which takes place a priori before the application is executed.

32:28  
By the way, the benefit of having the compilation, the compiler producing and by the way, in in Java or in, in C#, it doesn't have to produce immediately the output of the of the binary of the of the instructions for the CPU.

32:45  
It can produce some intermediate result which is then later turned to the actual binary code, but that's not that important.

32:51  
OK, So what what I'm saying is, if the compiler needs to be used at the design time, at the coding time, at the development time, it's always better because you can get some feedback.

33:02  
Some mistakes you made can be intercepted by the compiler and given back to you so you can fix your code.

33:09  
If your code is distributed to the end user and then it fails because you have a typo in your code, that's obviously not a good approach.

33:20  
This, this is somewhat embarrassing for the program.

33:24  
So how this works, don't get scared.

33:27  
Yeah, this don't panic.

33:29  
Maybe should be May be on the top of this slide, so don't get too excited about this.

33:37  
Just to give you some glimpses of what compiler is formally.

33:41  
Basically just assume that we have something which is called a language, an input language.

33:48  
Technically, this is like a language, your programming language you are currently using, and each language has something which is called a grammar.

33:56  
Grammar is like a formal description of that language.

33:59  
For instance, in Python it tells you what are the keywords that for or if is a keyword.

34:05  
So it has to be used in some sense, in some manner in some context, right?

34:10  
In C++, the grammar tells you that again the keywords and and that the block is wrapped into curly braces and stuff like that.

34:20  
And then you have output language, which is basically the instruction set of your CPU, which also can have a formal grammar which which is basically derived from ISA.

34:30  
If you have AISA which specifies what instruction set your CPU can use, then you have a grammar for your for your CPU.

34:38  
Basically, you just adjust minor details, but let's not dwell on the details at the moment.

34:46  
And we can say that this language is accepted by some automaton.

34:54  
The automaton in this context is just a formal specification of the CPU.

34:59  
Yeah, I'm not going to explain in detail what automaton can be.

35:02  
If you heard about Turing machines or stuff like that, there is a it's one of type of the automatons, which is one way how to abstract the CPUs.

35:10  
If you want to be a theoretician, again, this is not that important right now.

35:14  
What's important, what's the more important part is that when we formalize it like that, then compiler is basically just a mapping.

35:22  
Compiler discovers and creates a mapping between your input language into the output language.

35:30  
By the way, the language, the programming language isn't the language.

35:35  
The programming language is the grammar.

35:37  
The language in this context is your code written in that language.

35:40  
So the language is the string or the strings that represent your code.

35:46  
That's what's in this context called a language.

35:49  
Don't ask me, I'm not, I'm not this.

35:51  
I'm not expert in this area.

35:52  
I didn't know why they use this slightly confusing terminology.

35:56  
So they create a mapping between your code in your selected programming language into the output grammar.

36:05  
So it generates a binary code which should be accepted by the CPU.

36:14  
By the way, sorry yo.

36:22  
If there is for all words in the in the language in the input language exist an output word in the output language, then the the mapping exists.

36:30  
If there it doesn't, the mapping doesn't exist and the compiler is responsible for telling you which words in the input language were and there.

36:39  
So that's just a formalization of this process.

36:42  
Sorry about that.

36:43  
I'm always getting confused.

36:44  
I'm not the expert on this.

36:46  
So again, don't worry about it.

36:48  
If you want to get formal, there are plenty of opportunities.

36:51  
First of all, you will get the mandatory automata and grammars course and it's actually essential to get the notion about all these terminologies and all these details.

37:00  
So again, this this will be one of the courses you will pick up in the second, second year.

37:07  
So you will get the hang of it once you once you get the basics.

37:11  
And then there is a course called Compiler principles, if anybody is, it's elective actually.

37:16  
So you don't have, you can skip it if you'd like, but it's a good, good course because it will give you a good idea about how the compiler really works.

37:24  
And you will, you will create your own compiler for simplified C language.

37:28  
So you really write your own compiler.

37:30  
It's a cool thing.

37:32  
So basically this is it.

37:34  
Just to give you another glimpse what the grammar is.

37:38  
The grammar is specified using some rules and one of these rules can look something like this, an iteration statement.

37:46  
This is a like a variable, like it's called non terminal.

37:50  
It's like a placeholder or variable or something.

37:53  
So it's not actually actual letters or something.

37:57  
It's just a placeholder and these bold things are actual letters or actual tokens from the language.

38:02  
So basically iteration statement in C is either while do or for and this describes how these statements will look like.

38:10  
And the the thing in italics which is not bold is again non terminal.

38:15  
So it's again a variable.

38:16  
So basically while it's always when you have a parenthesis and in parenthesis there is something which is expression.

38:23  
And again, expression has its own definition and it's followed by a statement.

38:28  
A statement may be a expression ended by; Or it can be a block of statements in curly braces.

38:37  
Yeah, and and some some other things.

38:38  
So this is like a prescription of what can be written in the language.

38:45  
And this grammar can be used to derive, to parse your language and verify whether it's correctly written.

38:53  
And then this grammar is used to translate it into the, into the second language, into the output language, into the binary.

39:01  
So this is how it works on the theoretical level.

39:04  
Let's go on to the practical level, which is much more fun.

39:07  
On the practical level, compiler is actually composed of multiple components or parts.

39:14  
These are the most important ones actually.

39:16  
Again, this is just a simplified depictation.

39:19  
It's just a simplified schema.

39:22  
There are many internal.

39:24  
For instance, a compiler or assembler are quite complex parts and there are many internal steps inside.

39:29  
So I'm not going into all the details here, but basically these parts are important because if you'd like, you can force your GCC or whatever compiler you are using to give you the intermediate products after each of these parts.

39:46  
So if you like you can you can take a look how your code would look like if you just run the preprocessor and nothing else.

39:53  
You can you can just run the preprocessor and compiler and you will get get your code assembly language so you can examine the instructions there.

40:01  
Or you can run it through through here and here you will get something which is called an object file.

40:07  
And this is almost the binary, almost the binary which can be executed if it holds some parts of the binary, but it also holds other information like what memory should be allocated where, how the functions are named.

40:21  
So usually you have multiple object files because some are produced from your source code, some are produced by libraries and you need to put them all together to produce the ultimate final executable code.

40:34  
OK, let's start with this part.

40:37  
The preprocessor basically performs a text level tasks only.

40:42  
So things like includes I will use C or C++ as a reference.

40:47  
So if we are using C++, includes or macros are resolved at this particular level.

40:56  
So basically after that you get a huge C++ file because all the includes and everything are put into one file.

41:02  
So you get a huge C++ file which is still AC plus plus file.

41:06  
It's somewhat awkward if you want to read it, but you would recognize most most of the code because most of the code isn't malformed yet.

41:14  
After the compiler.

41:15  
This is the most difficult step from the user perspective because it really transforms your code into something else.

41:22  
But then again, it's it's pretty straightforward.

41:25  
If you if you take the compilers course, you will learn this part how to transform NAC code into into the some intermediate assembly language or intermediate instructions.

41:37  
The assembly language is very close to the machine code, so as it's represents your application, it represents your program almost at the instruction level.

41:47  
But some pieces are still missing.

41:49  
First of all, some optimizations are not performed.

41:52  
Some optimizations are performed afterwards.

41:54  
And 2nd, this also holds some additional stuff.

41:59  
For instance, you still don't have exact addresses.

42:02  
If you have some jump, conditional jump for instance, you don't have the address filled in yet, you have still the label.

42:08  
So you have some mnemonics inside the code.

42:12  
Still, this is resolved after the assembler, because over here all this stuff is finally resolved and here you really get the binary or something very close to binary, some intermediate binary.

42:24  
For instance, in C++, most of the object tile will be really the binary.

42:30  
OK.

42:30  
By the way, these three things usually are bundled together in one executable.

42:36  
So if you write like something like GCC and you give it your source code, all these steps will be automatically invoked and automatically performed.

42:47  
You don't have to do it manually.

42:49  
So it's like usually all in one bundle.

42:53  
The linker is sometimes also invoked immediately by the GCC.

42:57  
But if you perform one CPP file, if you perform pair CPP file compilation, so each CPP is compiled individually, then each CPP will produce the one object file by the compiler.

43:11  
And at the end you need to call the GCC one more time using all the object files together and then to link them.

43:19  
Why is it so?

43:20  
The answer is to simplify recompilation, because if you're writing your code and you need to recompile something, then if you if you cache your object files, you don't have to recompile all of them.

43:34  
Sometimes you do, because if you if you change something which is inadvertently included in audio files, then audio files will get recompiled.

43:44  
Yeah, what can you do?

43:45  
But if you just modify some, some small piece in some CPP file, then only that CPP file, so only that object file needs to be recompiled.

43:52  
So this steps doesn't have to be called for all the other object files.

43:56  
So just to speed things up, the linker can be called separately, especially if you're producing multiple object files from your source code.

44:05  
OK, by the way, libraries, and we will get to that in more detail, have two parts.

44:12  
Usually in C++, 1 part is the interface specification, so things like function names or classes or or stuff like that.

44:21  
So the compiler will immediately know what functions it should expect in the long run.

44:29  
For instance, if there is a some library which promises you, you can have a function that prints something out to the console, then the interface here of that library should told should tell the compiler that there is such a function.

44:42  
Because otherwise compiler would say, hey, I'm trying to find this this print function somewhere print F For instance, you already know print F from your home assignments, right?

44:52  
So I, I I'm reading this print F, but I really don't know whether this print F is correct because I don't know how it how this should look like.

45:01  
Whether it takes one argument or two arguments, should it be integer or a string or something else.

45:06  
So the compiler has to know so it can match the function calls with the appropriate arguments to the what what it thinks about the function declaration.

45:17  
And in the end the linker needs to be provided library objects where the actual implementation is.

45:23  
So the implementation of the print function doesn't have to be here.

45:27  
Here is only the interface specification, how the function will look like what what arguments should it accept.

45:33  
But here the the function should be implemented.

45:36  
So in the end, the linker needs to find all the functions that were previously declared in the interfaces or used from the interfaces of these libraries.

45:46  
To give you more concrete example, they'll just say that this is a compilation of this this CPP code which includes this display H header over here.

45:58  
So basically, yeah, basically this display H will be included here where include is like a copy paste it over here.

46:12  
And then these macros needs to be resolved.

46:15  
So let me give you a glimpse what will happen if we go step by step through the compilation process.

46:21  
So first part, if you want to try it on your own, this is an example from from MSVC.

46:28  
So from the Microsoft compiler, if you're using AGCC or something else, you need to find out the appropriate switches for that.

46:35  
But they are there.

46:36  
I just don't know.

46:37  
I just don't know them from top of my head, but you can find them quite easily.

46:41  
So if we just run the preprocessor here, now let's see what the result would be.

46:48  
OK, just zoom in so everybody has a good view.

46:58  
I think this should should be enough.

47:00  
So yeah, this is the same thing you you've seen on the slide.

47:03  
So there is the main code which includes the display and the disk, sorry.

47:09  
And the display which basically defines one function.

47:13  
You can see it only defines the function, it doesn't implement it.

47:17  
It's just the sorry, declares the function.

47:19  
It doesn't define the body of the function, sorry.

47:23  
So it just tells you that the function will be named show it takes one string, one constant character pointer, and it returns void, so it doesn't return anything.

47:34  
That's it.

47:35  
And what does the preprocessor do is it combines these things together.

47:39  
So basically it copy pasted this show over here.

47:45  
The end if and the if death these parts have disappeared.

47:52  
Why?

47:52  
So do you know what these these things do?

48:01  
And maybe we didn't explain this to into detail.

48:04  
So does anybody know what these lines do?

48:10  
OK so my fault my fault.

48:12  
Let me explain.

48:16  
So the first line says if not defined and this is the name of the macro.

48:23  
So this if this back row is not defined then we are going in.

48:29  
So this is what we want.

48:30  
Then we defined the macro and this and if goes to this if not def.

48:38  
So if this macro is defined, then everything in between will be skipped, will be discarded by the preprocessor.

48:47  
Why would we do that in in header?

48:49  
Well, actually we have to do that to prevent accidental multiple include.

48:56  
Let me give you an example.

48:58  
Let's just say you have a header AH and BH and BH holds include AH, right?

49:14  
So basically B includes A and then we have a main CPP which includes both.

49:28  
So technically it will include A.

49:31  
So the preprocessor will copy everything in A into the into the main, then it will include B and in B it will include A again.

49:40  
So it will copy the A two times in in the CPP.

49:44  
Normally if you do, if you do nothing, this will actually happen if you don't place any additional code there.

49:51  
So what we need to do, we need to prevent that.

49:53  
And this actual code prevents that because if the A is included the first time, this macro is not defined.

50:04  
So it's defined when it.

50:06  
When it goes into this code for the first time, it defines this macro and all this code is placed inside.

50:12  
It's not discarded, it's used.

50:14  
So when A is included in main, it will be properly included.

50:19  
And second time when B is being included and transitively A is is trying the, the preprocessor tries to include A.

50:27  
What happens is that A over here already has this macro defined, so it skips this whole block.

50:35  
And presumingly this whole block covers the whole file, so the A isn't in the a isn't included twice.

50:44  
If it were included twice and it holds something like this, it would be bad because the compiler would say, hey, there are two declarations of the same function.

50:52  
Show that's that's an error.

50:55  
You cannot have two declarations of the same function that that's a bad so that that's a mistake.

51:00  
So it would actually told you told you we cannot compile this.

51:04  
It's wrong.

51:05  
But even though it's correct because it would be the same function, by the way, there is a shorthand for that.

51:10  
It's called pragma ones.

51:17  
Just if you, if you want to use it, just look it up.

51:19  
It does the same thing.

51:21  
It's just shorthand for this basically.

51:24  
OK.

51:25  
And another thing I would like to point out is what's over here.

51:29  
Yeah.

51:31  
Why do you think there are some such awkward things that appear over here?

51:36  
What?

51:36  
What's this stuff?

51:38  
Would anyone venture a guess?

51:45  
Why would compiler, sorry preprocessor put this stuff over here?

51:56  
What?

51:57  
What purpose does it serve?

52:05  
No ideas.

52:11  
Really.

52:11  
No ideas.

52:12  
OK, error reporting every time you need In the subsequent steps when an error occurs, you need to properly report this error to the user.

52:24  
But the proper reporting is to tell the user where the error actually is in the source code.

52:31  
And if you just put all the source code on one pile, you need somehow distinguish which part of the of the code.

52:38  
So this tells you, hey this is where this is where line 1 of display H actually starts in the whole bundled CPP code.

52:48  
OK, so this part was copied since line one from display H.

52:53  
And here we ended on line 8 of display H and we starting since line 2 of hello CPP.

52:59  
So every time there is an error somewhere in here, thanks to these markers, the compiler will know how to properly point out where the error was in the original code.

53:12  
Is this clear?

53:14  
So this is this is like a hex you need to do if you need to keep track of where your code comes from.

53:22  
So this is preprocessor.

53:23  
This is the simplest part.

53:25  
Second part is slightly more tricky because we are now transferring from CPP, which is pretty understandable by humans, into assembly language, which is far, far worse.

53:35  
So let's try it.

53:37  
OK, the same example, the same piece of code, and this will be the the result.

53:42  
Yeah, this is much more incomprehensible, but let me just point out few things that we should recognize.

53:49  
First of all, there is a some string.

53:52  
Yeah, this string will be here.

53:54  
The DB is like a sequence of bytes.

53:58  
This is a sequence of bytes ended by zero.

54:01  
AH, with 0AH I I will reveal this syntax to you, the H at the end.

54:08  
This is a alt assembly syntax in in CPP.

54:12  
In C and CPP, we are using 0X for hex numbers.

54:19  
In assembly, they're using the hex number followed by large HI.

54:25  
Don't know why, but this is just a convention.

54:28  
So this is a hex number.

54:30  
What?

54:30  
What is 0?

54:31  
A hex In hex 10, which is the #10 Yeah, but but the hello world should give you the hint.

54:55  
Right now it's obvious, so this
Has to go somewhere.

54:59  
So this zero a is a character hexa representation of the character
What else is important or interesting here?

55:11  
Let me scroll a slightly more to the bottom.

55:14  
Is this part.

55:15  
Yeah, this is the most important part, the most intriguing part.

55:18  
And you probably recognize this one because you should at least slightly recognize some of these some of these lines.

55:26  
This this is obvious.

55:27  
What's this set EAX?

55:34  
Yeah, that's XOR, which sets the EAX to to zero.

55:37  
Yeah, that's the shorthand for writing zero into EAX.

55:41  
But you are right, that's XOR operation.

55:44  
And similarly, this is call to the show function.

55:47  
This is move of RCX which save the pointer of course to the message.

55:54  
And the message is the segment which was previously marked over here.

55:59  
Sorry.

55:59  
Yeah, this is the message segment which holds this data.

56:04  
So basically this is how the pointer is acquired and then how the how the function is called.

56:13  
But by the way, why do you think the EAX is zeroed?

56:22  
Yeah, I know this is tricky question.

56:25  
OK, So what we quick refresher, Quick refresher we are just performing show message.

56:32  
So we are just calling a function show giving it the pointer to our message and then we are doing return 0.

56:44  
That's right.

56:44  
So we are returning the value into the EAX register.

56:48  
That's what you can deduce from this from this piece of code.

56:51  
The EAX should hold the return value of a function in this case.

56:56  
Yeah, this is just some stuff that's required to do to be done.

57:01  
This is just altering the the stack pointer so we get to that in a short order.

57:07  
How stack pointer works and return 0 means a return to the address 0 and the 0 here doesn't actually hold the actual address, it's just some intermediate value that will be filled in later.

57:19  
OK, so this is how the assembler works.

57:24  
Yeah, the the full compilation you already know that will produce some executable file like MZ exec on on Windows or L file on Windows.

57:33  
So I'm not going to show you that.

57:35  
And the object file we we get to that in a short order.

57:38  
Before we do, let me explain a few more terms.

57:43  
Now we need to think about how all the stuff is linked together.

57:47  
So in the end the compilation process will produce object files.

57:51  
OK, so we get the object files and now we need to put the object files together and we need to put the libraries into the object files.

58:03  
Library is basically a piece of code which was pre compiled by someone else in this context which was pre compiled by someone else and given to you either as a binary or as a part of the operating systems.

58:16  
Most of the libraries are in the operating systems or some libraries are given to you in the runtime of your of your application.

58:24  
So there are many ways how to get get the library and there are two ways how to link the library, static linking and dynamic linking.

58:33  
I will explain in detail on a next door few slides from now.

58:38  
So static linking use this type of libraries.

58:41  
This is notation for Windows and this is notation for Linux.

58:44  
OK, I will use Windows notations for various reasons.

58:48  
So on Windows you will get a Lib file and you can get the Lib file to statically link it into your application.

58:55  
For dynamic linking you use DLLS in Windows or SOS shareable objects on.

59:01  
On Linux the linking process is the bundling process where the object files are put together with the libraries and everything is probably rearranged and re associated so it can be placed into one executable file.

59:15  
There are several steps that that take that takes place there.

59:19  
And then the important part is it needs to handle something which is called relocation because you need to place your code somewhere in in your address space.

59:32  
Sometimes your code can be generated in a position independent way.

59:37  
So it doesn't matter on which address the code runs.

59:40  
Let me give you an example.

59:42  
You have something like a for loop for instance.

59:48  
I just write it schematically.

59:51  
So basically somewhere at the beginning or at the end or maybe both, there should be some jump.

59:57  
For instance, there is some conditional jump.

1:00:04  
I just symbolically write it as jump conditional, which basic based on this result will take a jump at the end of the loop.

1:00:16  
Usually it's the other way around.

1:00:18  
At the end of the loop we take a jump back, but it's not that important.

1:00:22  
So there is some jump and the jump needs to know the address where to jump, right?

1:00:30  
So if you have only absolute jumps, so the address has to be absolute, then it's tricky because this address needs to be adjusted when the program is loaded into the memory, because before that you don't know where the program will be loaded if the jump is relative.

1:00:48  
So you just write something like jump C + 16, which tells the compiler, which tells the CPU jump 16 bytes ahead.

1:00:58  
It's just like a add function for the program counter for the instruction pointer.

1:01:03  
OK, so if it's written in this way, then we're fine because you can place this for loop anywhere in your memory and it will work the same.

1:01:12  
Because if we get to this jump and we need to jump, it will tell the compiler, hey, just jump 16 bytes ahead and everything will be fine.

1:01:19  
And yeah, of course you need to have the proper management that these instructions in the loop plus and everything else over here will take exactly 16 bytes.

1:01:30  
So the jump will jump after the the last instruction here.

1:01:34  
So you need to the compiler actually needs to do appropriate calculation here.

1:01:39  
But in the end, this code is ideal for us because it can be placed anywhere in the memory and it will work the same.

1:01:47  
So that's what we call the position independent code.

1:01:50  
Other code needs to be relocated, so places where an absolute address will be.

1:01:56  
So if there is an instruction that jumps to an absolute address or an instruction that loads or stores data from an absolute address, it needs to be recalculated during the linking and then again during the program deployment 2 times.

1:02:15  
That's why there is the third part, the third player in this game, and that's called Loader.

1:02:20  
You cannot just put your application in the memory and start executing it from the top to the bottom.

1:02:27  
It's much more to it than this.

1:02:29  
So Loader is a part of the operating system which understands the format of the executables in your operating system.

1:02:36  
By the way, an egg note, last week I had an incident.

1:02:40  
I wouldn't call it an incident.

1:02:41  
I have a story with one of the students from the second or third year and the student just compiled an executable exa file on on his Windows machine on his laptop.

1:02:52  
And then he uploaded this exa file to the our Linux server and he was very surprised that it wasn't running there.

1:03:02  
Why is that so?

1:03:03  
Well, it's obvious because Windows is an operating system and Linux as an operating system are using a different formats of executable files, so they have different loaders.

1:03:14  
This part of the operating system is different, slightly differently implemented in both.

1:03:20  
They're doing basically the same thing, but they recognize different executable formats and they perform slightly different functions.

1:03:28  
The purpose is the same, but they're implemented in slightly different ways.

1:03:31  
So you cannot compile Mac application and just deploy it on Linux without any additional support it it wouldn't work because you need to deploy it on the operating system for which it was designed because the loader is expecting some some format of the executable.

1:03:50  
By the way, as I just told you few moments ago, Windows are using MZ executables while there is another word for it, I just forget the proper name.

1:04:00  
But Linux is using L files for that.

1:04:03  
So there's a different file format for binaries.

1:04:07  
And the job of the loader, regardless of the operating system is to yeah, load the executable to the memory.

1:04:12  
That's the, that's the easy part.

1:04:14  
And then perform first dynamic linking.

1:04:17  
So it needs to dynamically attached the libraries which are provided by the operating system and which are this is, This is why it's called dynamic linking.

1:04:27  
So which are linked at when the application is executed at the runtime and then it needs to perform relocations again.

1:04:35  
So this is like a first stage when the relocations are just like a pre computed when the stuff is integrated together and then the relocations need to be recalculated again based on where in the memory the application was actually placed.

1:04:51  
OK, so this is a tricky part.

1:04:54  
I will I will show you this in more detail on an example.

1:04:58  
But first, let's start with the with the linking process.

1:05:02  
So what I mean by linking is that let me let me explain this example.

1:05:09  
We have a library called.

1:05:11  
My library doesn't matter and it provides this this stuff, one variable, one global variable which is accessible from everywhere, and two functions.

1:05:21  
OK, so this stuff is in my library H and you can see these are just declarations.

1:05:29  
So we are just telling you we are telling ahead the compiler what type of functions the compiler can expect what what is available on the palette.

1:05:38  
And this is placed in the H file so it can be included in the any application we want to we want to use which ones to use this library.

1:05:47  
And then this application can actually call this functions open or it can call this function read.

1:05:54  
The compiler is fine with that because it was told a priori ahead that these functions will exist, should exist.

1:06:03  
And what are the the arguments?

1:06:06  
This one takes a pointer to the constant character.

1:06:08  
This one takes an integer and integer and a buffer.

1:06:11  
OK, so we the compiler knows that when I'm calling these functions like this, it's fine because yeah, I'm providing a string which is constant pointer to a constant character.

1:06:22  
This function accepts constant character pointer.

1:06:24  
Yeah, that's fine.

1:06:25  
It's matching.

1:06:26  
This function accepts integer and integer and a buffer.

1:06:30  
And I'm I'm giving that an integer and an integer and a buffer.

1:06:32  
Yeah, that's fine, no problem there.

1:06:35  
Actually there is slight problem because this is a pointer to character.

1:06:39  
Actually it's a, it's a character array.

1:06:41  
But it's it's the same as a pointer to character.

1:06:43  
This is a void pointer, but anything, any pointer can be casted to void pointer without a warning.

1:06:50  
It's fine to cast anything to void pointer.

1:06:52  
So this is fine for the compiler.

1:06:54  
Compiler just performs the cast and and is very happy with with this result.

1:06:58  
OK, but in the end we need to make sure that the function actually exists.

1:07:05  
The the the appropriate function will be actually called and that's the job for the linker and the loader which are responsible for finding out the right implementation which should be in my library C and put these things together.

1:07:20  
So making sure that this and this function will be matched somehow via addresses in the memory.

1:07:27  
By the way, the linker and the loader work on the strings, so the name of the function is the important part for matching.

1:07:37  
So in old C this string open and this string open is the thing that will be matched by the by the compiler.

1:07:45  
The name must match.

1:07:46  
So the sorry the linker will perform this this matching on the names of the functions.

1:07:53  
OK, knowing this, there are several ways how to perform the the linking.

1:08:00  
The easiest part, most straightforward part is if you have the source code, because if you have the source code of this library, then you can just compile everything together.

1:08:09  
That's the easiest part.

1:08:10  
You just take the library and take your application and the compiler will first compile the library into object file.

1:08:18  
Then your application will be or Application C will be compiled into object file and then the linker will put it all together.

1:08:24  
So here is where the preprocessor needs to know needs to put in the header files.

1:08:30  
So the compiler will know what functions to expect, how to match the parameters and here the linker will find out call to the open function here and that there is an open function exported from here and it will perform the matching.

1:08:46  
So it will adjust the address where this core is performed to this function, how it lies and it places the the the right function into the memory somewhere.

1:08:57  
This is the easiest part because if you have the source code, you you can do whatever you want.

1:09:02  
This is this is also what happens if you just separate your application into multiple C files or CPB files.

1:09:08  
And that's by the way, a good idea to do if you're writing a slightly larger application, if it has several 1000 lines of code, then yeah, it's a really good idea to separate it into multiple files and the same will go.

1:09:20  
You can, you can name your files part partial library or something.

1:09:24  
So you will know this is like a slight library file.

1:09:27  
This is the main, main, main file that where the main function is etcetera.

1:09:32  
Another option is that the the provider of the library doesn't want to give you the source code.

1:09:37  
Yeah, nowadays everything is given as a source code, right?

1:09:40  
Because we have GitHub.

1:09:41  
Well, this might be true for JavaScript or Python, but it's still not very true for C or C++ and usual not not true for for C# or Java.

1:09:52  
By the way, there are still many software companies which believe that keeping some intellectual property as the source code would be a good idea to not to give it to everybody, because then everybody can just use it.

1:10:06  
Yeah, today maybe we are slightly shifting from the source codes to memory model to large models and AI models and etcetera.

1:10:14  
So today we aren't protecting the code anymore, we are protecting the data, but it's basically the same thing.

1:10:21  
So if somebody wants to protect their source code because there is some hidden intellectual property and it would be bad if someone can just read our ingenious algorithms, then we can hide the CPP file and provide only.

1:10:36  
Yeah, we still need to provide H file.

1:10:38  
Everything which is in the header still needs to be provided because otherwise no one could compile against our library.

1:10:46  
The H files needs to be distributed always, but the CC file can be converted into the object file and only the object file.

1:10:53  
Actually not the object file but the derivate of the object file which is the library file for static linking can be provided.

1:10:59  
So instead of instead of giving you the source code, the the author of this library will is giving you the library derivate that the compiled library if you will.

1:11:09  
And in this case as a Lib file.

1:11:11  
So the linker will embed the Lib file.

1:11:13  
This is just just more fancy elaborate way how to combine multiple object files into one file.

1:11:21  
So basically this look at this as a collection of object files.

1:11:26  
That's one way how to look at it.

1:11:27  
So basically you are linking together object files and collections of object files together via linker.

1:11:33  
So basically it's the same thing, we just skip the compilation of the object file.

1:11:37  
This is performed in hidden in private by the author of the library and this is performed by you when you're compiling your code.

1:11:47  
That second option, trouble with that is some of the some of the libraries are used in every other application.

1:11:54  
For instance, if you're on Windows and you are using some libraries for operating the GUI of the windows, so creating forms and creating buttons on these forms and stuff like this, then you need some basic library which is always present in the Windows system.

1:12:12  
And it would be very impractical if this library has to be embedded, copied over and over and over again into every executable that it wants to use it.

1:12:22  
That would be very impractical, right?

1:12:24  
So what this operating system does is that it just creates a shared library, a DLL or multiple DLLS for that matter, which are actually distributed alongside with the operating system.

1:12:37  
Or sometimes they are not there implicitly, you have to install them, but they are shared among all the applications that use this particular environment.

1:12:46  
So basically the linker.

1:12:49  
By the way, the linker needs to know the DLL a priori, so the DLL needs to be present during the compilation time.

1:12:55  
But the difference is that the DLL is not embedded into the executable code.

1:13:00  
The DLL is used only to verify that all the functions that should be there are there.

1:13:06  
So the linker is satisfied that the DLL will provide all the functions that are promised via via the interface.

1:13:15  
And then it somehow puts in some stubs which are later fixed by the loader.

1:13:24  
So it just puts there some some markers that say, hey, loader, when you're loading this application, we really need this DLL and this DLL needs to supervise these functions.

1:13:34  
So make sure they are properly relocated once the executable code is loaded into the memory.

1:13:41  
So there is external stuff that seems to be performed after the application is loaded to the memory by the runner, by the loader.

1:13:49  
But the the price, the the benefit of this approach is that this DLR is not embedded into the executable.

1:13:55  
It's in the memory only once, it's on the system only once and it's loaded only when needed, on demand, if you will, by this, by this executable code.

1:14:06  
By the way, second benefit is, and that's just a tricky stuff.

1:14:09  
We get to that when we talk about operating system.

1:14:12  
If we have a multiple applications that are using this DLL code simultaneously, then actually some memory can be saved because the DLL can be present in the memory only once.

1:14:24  
But that's another story, just another way how to save some additional stuff like like the memory.

1:14:32  
So these are three ways how to compile applications through the source code, which is the simplest and then static and dynamic linking.

1:14:40  
OK, we have 15 minutes left so we can still continue.

1:14:48  
The most difficult part when combining object files and when actually placing the the executable into the memory is how this executable or actually its memory space should be organized.

1:15:00  
So let's start from beginning.

1:15:02  
This is a schema.

1:15:04  
You probably, you probably already seen this how data are organized within application memory space.

1:15:11  
Right now.

1:15:12  
Let's not dwell on details about that.

1:15:16  
Multiple applications needs their own multiple virtual spaces.

1:15:20  
So let's not let's not go into the details of memory translation.

1:15:25  
So let's just assume that every application has its own memory space, either an own memory block or somehow it's, it's assured that this this particular memory is only it, it belongs only to this, to this program.

1:15:37  
So at the top, usually there is a code.

1:15:39  
So there's something called called code segment.

1:15:42  
And this holds all the instructions.

1:15:45  
Everything that needs to be at some point could be at some point executed.

1:15:50  
After that there is the static data.

1:15:53  
So this is the part we know at the beginning that will be there, like global variables, for instance.

1:15:58  
And then we need two additional memories, stack and heap.

1:16:02  
Stack is where all the local variables and function arguments will go during function calls and heap.

1:16:10  
We covered heap last week.

1:16:11  
Heap is the dynamic memory from which we allocate dynamic variables like with malloc or new or other mechanisms that are provided via your your application, your own time.

1:16:24  
It's slightly more difficult than that because for instance, the static debt data can be divided into three parts.

1:16:32  
First of all, there are constants, some literals, and they are somewhat.

1:16:37  
They lie somewhere in between code and and the data.

1:16:39  
Because they are not code, they cannot be executed obviously, but also there cannot be modified.

1:16:45  
So this is like a static data but read only static data.

1:16:49  
And then there are two types of static data, initialized and initialized.

1:16:54  
Initialized means that you provide you provided in the code the initial values for this data like an initializer for your global variable or if you have a modifiable string, if you have a not constant character pointer but character pointer so it can be modified some some time later.

1:17:13  
And initialize static data is like a huge array for instance, which is not filled by anything or which is, which is initialized not with the data as a data embedded in the code which were embedded in your source code, but initialized with something implicit like zeros.

1:17:31  
If you create a vector, huge vector of of data.

1:17:35  
But that's a bad idea because that's a bad term because vector actually is STL container.

1:17:40  
So if you create a huge array, this array is zeroed at the beginning, but there is no implicit part of the data which is present in the executable.

1:17:50  
This data are initialized via data stored in the executable file on the hard drive.

1:17:57  
So this is basically like copying the data from the file into here this is just we need to allocate the memory for them and perhaps 0 them or something implicit.

1:18:08  
So this is these data are not present, they have not their own copies in the executable file.

1:18:16  
Second thing is that the stack needs to be allocated for each thread separately.

1:18:24  
So if we have multiple threads running our application, then each thread has its own stack.

1:18:32  
We will talk about thread scheduling and stuff like that in the operating systems part.

1:18:36  
So right now, just let let me assure you that it's not that important for the rest of the part.

1:18:43  
And then again, stack is allocated dynamically.

1:18:48  
So it's not present in the executable file.

1:18:51  
It's it's allocated by the runner when the file when the application is being started.

1:18:56  
But just this, this hiccup is there because if we have only one stack, it's easy because steak grows one way and he grows the other way.

1:19:05  
So if you just place one on the on the bottom of the statically allocated piece of memory, and the size of this is obvious from as long as you load, as soon as you load the file, you know how large this part is, How much memory for this.

1:19:25  
So you can place the stack over here and heap on the other side of your memory space and they can just grow one against the other.

1:19:33  
Yeah, obviously they can overlap.

1:19:36  
And once they overlap, you're in big trouble because at that point your memory run out and your application will be terminated because there is nothing else we can do.

1:19:45  
But yeah, or we can we can deny the heap growth or stack growth.

1:19:51  
Well, if we deny stack growth, your application re terminate it anyway because that means you cannot call a function.

1:19:56  
And once you cannot call a function, there is nothing else you can do.

1:20:00  
You just need to terminate your code.

1:20:01  
But if you cannot grow heap, there are some mechanisms that can be put in place like the allocation can fail for instance.

1:20:09  
And there there are maybe ways how to deal with failed allocation so your application doesn't have to end.

1:20:14  
But if you place multiple stacks one after another, you are somewhat deny this opportunity because one stack can run into the other stack.

1:20:25  
Yeah, that that's a tricky part.

1:20:26  
So basically that we are putting some traps at the top of each stack.

1:20:31  
So if one stack tries to overrun another stack, it's detected by operating system.

1:20:37  
It's just that there is.

1:20:38  
This is much more tricky when you placing the stacks one after another.

1:20:42  
But then again, stack should be relatively small.

1:20:45  
You shouldn't place two large structures on your stack.

1:20:48  
You shouldn't create two large local variables if you are creating a local variable.

1:20:55  
And this local variable should take megabytes or more, then you are not doing your job right.

1:21:00  
So local variables can be small buffers, integers, pointers, etcetera.

1:21:05  
That's fine.

1:21:06  
So it usually takes a few kilobytes.

1:21:09  
If your stack grows unexpectedly, you are doing something bad.

1:21:13  
Either you have two large data structures and you should use heap for those.

1:21:18  
For larger data structures you should use heap or sometimes you can use global memory.

1:21:23  
And secondly, second option, second possibility is that the stack is growing out of control because you have unchecked recursion, right?

1:21:32  
If you, if you call a recursive function and you do it wrong, like you don't have the right final case, you probably run into that.

1:21:41  
But did anybody write something recursive?

1:21:43  
I I believe you do you have to do that in the first year, right?

1:21:48  
Yeah.

1:21:48  
So so you know what a recursion is.

1:21:50  
So if you do it wrong, then your recursion routes runs out of control and your function is being called indefinitely.

1:21:57  
So at some point it will run out of stack.

1:21:59  
In Python you wouldn't know because Python has a very sophisticated mechanism how you detect that there is a too deep, that the stack gets too deep, you have too many calls performed.

1:22:10  
So it it kills you before it runs out of memory.

1:22:13  
But technically in C++, you can run out of stack memory if you do something wrong.

1:22:19  
Yeah, second, second thing.

1:22:20  
By the way, in X86 architecture, this is this is the other way around.

1:22:25  
Stack is on the bottom and it's growing up, and heap is on the top and it's growing down.

1:22:30  
But that's just a technical detail.

1:22:32  
The hardware has specified that when the stack is growing, the address is decreasing.

1:22:37  
So we need to place stack on the bottom.

1:22:40  
Just a technical technicality.

1:22:42  
It's not that important.

1:22:43  
OK, Yeah, I think we can cover the linking part and linking part not linking park and then we can we can call it a day for today.

1:22:54  
So what's the linking process?

1:22:56  
We already know that that any C file will produce an object file and every object file will actually hold the first part of this memory space.

1:23:04  
So remember this, the code constants and initialize static data needs to be put into the executable.

1:23:11  
The rest will be allocated dynamically or created dynamically by the runner.

1:23:16  
So these three blocks, these three segments needs to be placed into the object files and then subsequently into the executable file.

1:23:25  
So this object file will hold code A, constants A and static data A.

1:23:30  
The same goes for the B.

1:23:32  
That's right.

1:23:33  
So each each object file will have these three segments somewhere.

1:23:37  
And there are there is a library.

1:23:39  
And as I told you, library is basically like a package of multiple object files.

1:23:44  
So it contains object file P&Q.

1:23:47  
So it has all three parts of P and all three parts of Q.

1:23:52  
And the job of the linker is put all these segments together, but in the correct order, because we need to make these segments compact.

1:24:02  
So sorry.

1:24:05  
Yeah, so we put it all into linker and the linker will concatenate first the code, then the constants and then the static data is enough, right?

1:24:18  
Well, it's not that easy.

1:24:21  
Let's just assume that we have piece of code over here.

1:24:25  
Actually not.

1:24:26  
There is a function over here which is being called somewhere from the code B.

1:24:32  
So the address of this function here is calculated assuming this is the address 0.

1:24:38  
So the offset of this function is calculated from this offset.

1:24:42  
But if we move the code B segment over here, this address is no longer valid.

1:24:48  
We need to move it by the size of code A, right?

1:24:52  
This is what is called a relocation.

1:24:54  
So every piece of code in B or somewhere else which refers to this line of code, either a jump or a function call or anything else that absolutely refers to this address needs to be adjusted, needs to be relocated, recomputed.

1:25:09  
So the correct value is over here.

1:25:12  
So the correct, correct value for this address is all over the code.

1:25:16  
And that's the tricky part because you you can't, you're not just adjusting this line, you are adjusting every other line of code which refers to this line.

1:25:25  
So you need to map your code, you need to save this relocation information in your object files so the linker can perform these adjustments when the code is linked.

1:25:38  
The same goes for constants, the same goes for the data.

1:25:42  
OK, so this is the main objective of the linker to combine the things properly and then to perform the relocations to the adjustment of addresses because some of the pieces, some of the segments can move.

1:25:56  
By the way, where did P?

1:25:57  
Where did, where did P go?

1:25:59  
The code that P is not not there.

1:26:01  
Why?

1:26:07  
Actually, yeah, linker is smart.

1:26:09  
So linker finds out that that any, no, none of the functions in code P was called from either of these other functions or other segments.

1:26:18  
So it says, OK, hey, I can drop it no problem.

1:26:21  
So it just compacts your code.

1:26:24  
Only the necessary code, only the reference code is put there.

1:26:27  
The same goes for constants and everything else.

1:26:30  
So we have like a few more minutes.

1:26:36  
So I will just start this.

1:26:38  
I will just tell you a few words about it, but we get into the examples and into the details next week.

1:26:43  
So the object file actually has a very specific format for Windows we are using cough yeah, I forget what what that means.

1:26:53  
Compact FF is file format, Compact object file format, something like that.

1:27:01  
And this is the format of the file where all the segments are placed.

1:27:06  
So it needs to cover the format of code, constant static data and few other things like the relocation.

1:27:12  
So basically it's a collection of segments.

1:27:16  
This is a list of segments.

1:27:17  
If you just take a object file, this is the object file I compiled from the hello world example you saw before.

1:27:24  
And this is if I just print out the summary.

1:27:27  
By the way you can try it your yourself.

1:27:29  
There is dump bin executable on windows objective dump on on Linux or dump object.

1:27:36  
I'm not sure there is an equivalent tool on Linux.

1:27:40  
So you can you can try this yourself.

1:27:42  
And if you just ask for summary, then this is what what will goes out.

1:27:45  
So it will list all the all the segments which are present in the object file.

1:27:51  
By the way, text, yeah, over here text is the code.

1:27:55  
Yeah, I know.

1:27:56  
But it's it's it's not called code for some reason.

1:27:58  
It's called text.

1:27:59  
Our data, our data, our data over here.

1:28:03  
This is the read only data or raw data.

1:28:06  
This is where where the constant will be.

1:28:09  
Yeah, X data, P data, it's very, very difficult debugging symbols.

1:28:13  
It's very important.

1:28:14  
You can, you can either generate a version with debugging symbols and then all additional stuff is placed there so it's easier to debug your code or it's dropped if you're, if you're compiling for production, this is the initialized data, P data and X data.

1:28:28  
Also use it's awkward stuff for for exceptions and directives.

1:28:32  
Yeah, there are some some formal declarations.

1:28:35  
So this is like a list of sections what can be found in the object file.

1:28:39  
And since I'm running out of time, I will I will just cut it over here and we will take a look next week about how some of these important segments look like.

1:28:47  
OK, are there any questions, immediate questions before we conclude for the day?

1:28:53  
Nope.

1:28:54  
OK.

1:28:54  
So thank you for your attention and hopefully see you next week.