**2025-05-22-para-en**

0:10  
So hello everyone, welcome to the last talk.

0:15  
Today we need to finish everything so it will be last long run.

0:20  
We have a few more things to wrap up in the in the talk about operating systems.

0:27  
Afterwards we have one full lecture about parallelism and parallel programming.

0:31  
And in the end, I will talk about how the exam will look like and what should you prepare for the exam.

0:38  
So again, like last week, we are we are going to make a short recess, short break in the middle.

0:45  
By middle I mean roughly after one hour and a half.

0:48  
So we will take a break because it won't be possible to for you to stay awake all the time during the whole session.

0:59  
So please try to bear with me.

1:01  
I know it will be a long session, but I will try to wrap it up in the painless possible way.

1:07  
So last time just to just to smoothly continue.

1:11  
Last time we talked about paging and in general virtual memory virtualization.

1:16  
So the paging is now the technology which is being currently used for translation from virtual memory to physical memory.

1:24  
This is just the slide that wraps it up very nicely.

1:27  
So we need to translate an address from the virtual address space into the address into the physical address space.

1:34  
The address is chopped in two parts based on the size of the pages and frames.

1:39  
Pages and frames are the same size, so we need to chop off last n -, 1 bits, last small N bits, sorry of the offset.

1:50  
That's like the address within one page or the address within one frame.

1:54  
And the front part of the address is the index of the page.

1:59  
So the index of the page goes to the page translation table.

2:03  
From the page translation table, we get the address of the physical frame where the page is allocated.

2:09  
And we combined with the offset from the which was chopped from the previous address and this together forms the physical address.

2:17  
That's it.

2:18  
The tricky part is how to translate the page index into the physical address.

2:23  
So basically we have several options.

2:26  
My pointer isn't working, sorry.

2:29  
We have several options, but the most straightforward option would be to have a page table, which means a huge array where for each page there is one slot.

2:39  
So each page has one record one element in this table and they are mapped 1 to 1.

2:45  
So page with the index 14 has the 14th, actually sixty 15th because it's indexed from zero.

2:53  
So sixteen 15th, sorry element in this array and the element holds the address, the physical address of the frame where this page is actually stored in the physical memory.

3:06  
That's it.

3:07  
The greatest advantage and possibly disadvantage of this translation mechanism, well of actually whatever translation mechanism we would use is that some of the translations might not exist, might not be available.

3:21  
So here in this entry over here for the page 111 is actually a special record that indicates that there is no valid address present here, so the translation doesn't exist.

3:34  
That means that when the computer, when the CPU tries to access this memory in this virtual address space, tries to translate this pointer into the physical address, it fails.

3:47  
It throws off an exception, it throws off a page fault which can be handled by the operating system.

3:52  
This may happened because of an error, because your program is doing something it isn't supposed to be doing, there is a bug in your code or something, or it might be intentional.

4:03  
Operating system uses these page faults, for instance, to allocate your memory lazily.

4:08  
So it gives you the memory in virtual address space immediately.

4:11  
But the physical memory is given to page π page frame by frame as you start using these pages, these frames.

4:20  
So this is one option, one thing that can happen.

4:22  
OK, this is like, this was like a few things to wrap up and this is a real mechanism.

4:30  
This is just the theory because we agreed that this page table would be huge even for 32 bit address space, this page table would take megabytes and that's too much for a normal space.

4:40  
That's waste of memory, especially that for normal applications, for small applications like the applications you're trying to write during your programming courses, using 4 gigabytes of virtual memory is unthinkable.

4:54  
You don't really need 4 gigabytes.

4:56  
So for small applications you need only few megabytes of memory and it will be very very wasteful if you need to allocate this whole page table.

5:03  
So one option you can do is to compress it and compress it using a hierarchy.

5:09  
By hierarchy I mean you chop basically the address, the page address in two.

5:14  
So instead of having 120 bit index into a page table, we split it in half and we have a 10 bit index into the first level of page tables and 2nd 10 bit index into the second level.

5:29  
The benefit we got is that this 10 bit index will require only a page table that holds 2 to the power of 10, meaning 1024 records.

5:44  
And assuming that this is a 32 bit architecture, then one record will have 4 kilobytes, 4 bytes.

5:50  
So one KB of records have 4 kilobytes.

5:54  
Accidentally, incidentally, or by design as you wish.

5:59  
This fits exactly one page.

6:01  
So it's like very well, very well designed so that this page table also fits one page.

6:08  
That's like a benefit.

6:10  
And this first level needs to be always there because that's like the main translation mechanism.

6:16  
And by the way, where this page is located, where this page table is located in memory is stored in a special system register.

6:24  
It's like a hidden register not available for regular applications, but it can be written by the operating system.

6:30  
So the operating system may allocate A-frame and say, hey, this is this is the first level page table for this process, OK.

6:38  
And then whenever this process runs, it makes sure that the corresponding CPU core has this appropriate address stored in this hidden register.

6:48  
OK, so this is like a first level and the second level and there comes the benefit is that the second level doesn't have to be there for all the pages.

6:58  
So basically the translation goes that we find the first translation for the first level in the first page table, then we go to the second level.

7:06  
This is a physical address or a frame where the second level page table is.

7:12  
Then we perform the same translation using the second-half of the page index, finding the appropriate value in here.

7:20  
And here finally resides the address, the physical address.

7:25  
The dark blue is physical address of the frame where the actual data are, where we are trying to locate our data during our translation.

7:34  
So it's like a 2 step process instead of one step process.

7:37  
And as I said, the benefit is that not all the second level page tables need to be there.

7:43  
Yeah, if we allocate all the second level page tables, there will be again 1024 of these four KB entries.

7:51  
So in in, in fact, we will lose lose some memory because we need this extra page, this extra table.

7:59  
So if we need to allocate the entire space, we would waste 4 kilobytes of memory.

8:04  
Yeah, that that's a small price to pay.

8:06  
But if we don't use the entire space, we can omit some of the second level page tables and save space.

8:13  
For instance, if you need only few kilobytes, you allocate only few of these page tables or even even A1, even only one page table at the second level.

8:22  
By the way, if we have only single page table at the second level, how many, how large memory, how large memory block can we allocate using that?

8:33  
Can we administrate if we have only?

8:37  
There is only always only one page table at the first level.

8:41  
That's how it's designed.

8:43  
But let's say we have only the first entry here filled and only the first entry points to the valid A second level page, a table.

8:52  
So there are only two one first level and one second level page tables.

8:56  
How much memory can we administrate using only these two tables?

9:11  
4 kilobytes means one single frame.

9:17  
So if we have one page table at the second level, we can have no OK, let's go through that step by step.

9:28  
So we have one entry at the second level.

9:31  
How many how?

9:32  
One table at the second level?

9:33  
How many entries?

9:34  
This table has 1024.

9:40  
Yeah, it's it's there is a hint over there.

9:43  
So we have 1024 entries.

9:47  
That means we can address 1024 pages or frames.

9:53  
How much memory can cover 1?

9:57  
And it's it needs to be continuous memory in the virtual address space because these int entries cover continuous indices at the second level.

10:06  
So it's a continuous memory in the virtual address space.

10:10  
Yeah, it can be sparse in the physical address space, but we are we don't care about that.

10:14  
So how large portion of the virtual address space one table actually covers when it covers 1024 entries of pages and each page takes 4 kilobytes?

10:33  
4 megabytes?

10:34  
4 megabytes?

10:35  
1024 \* 4 kilobytes is 4 megabytes.

10:39  
So basically if we have only one, this second level table, it covers its spans over 4 megabytes of continuous virtual memory address space.

10:52  
And actually it's aligned because it's like these values are all zeros.

10:57  
So the other way around, I could ask how much memory could you address if we utilize this part of the address?

11:04  
So if we use 20 bits of the address, 22 bits, sorry of the address.

11:08  
If we use 22 bits, we cover 4 megabytes.

11:12  
That's the that's the same question just from the the opposite side Ask.

11:16  
OK, so if everybody understood that, then you probably hopefully understand how the translation through the page tables goes.

11:26  
OK, I'm sure.

11:28  
Did we cover this exercise last week?

11:31  
I'm sorry, did did we go through this?

11:34  
No.

11:34  
OK, so let's go through this quickly or in a decent pace.

11:39  
I'm trying to rush things up a little.

11:41  
So we will finish in a due time.

11:44  
So let's just have it.

11:45  
Let's just have this these assumptions.

11:47  
So we have a 32 bit address space.

11:49  
We use this two level paging which you just saw in the previous slide.

11:54  
We use 4 KB pages and let's assume that the integers size of the integers 4 just so we have all the all the bases covered and we execute this piece of code.

12:05  
OK, so basically there is some pointer to some data somewhere.

12:10  
Yeah.

12:11  
And we have some variable not very important.

12:13  
And what we are doing is that we traversing this data, we are actually this is like an array, array of integers and we are accessing 2000 of these integers in a continuous memory because arrays are always continuous in C or C So this is like a continuous block of 2000 integers and we are reading them all and just adding them to some sum.

12:36  
This actual operation isn't important.

12:38  
We are focusing on memory operations right now.

12:41  
So let's assume that we are executing this piece of code.

12:44  
What we don't know actually is where this pointer points.

12:47  
It can be anywhere in the virtual memory address space, right?

12:50  
We don't know what this memory is.

12:52  
It can be on the heap, it can be on the stack, it can be in the global variables.

12:56  
I don't know actually, but it doesn't matter that much.

12:59  
I have no assumptions where this pointer point, I just know it's somewhere in the virtual address space.

13:04  
And yeah, because it's an array of integers, it needs to be aligned to the size of an integer, but that's it.

13:09  
I don't know anything else.

13:11  
Yeah.

13:12  
So yeah, this address is divisible by 4 because integer has four bytes, but that's it.

13:17  
I don't know anything else about this address.

13:19  
So the questions how many data pages are actually read are actually accessed by this operation.

13:31  
We are reading 2000 4 byte indigers.

13:35  
I know this is going to be wrong, but it might be about 2000 well pages.

13:46  
OK page is 4 kilobytes.

13:48  
So 2000 pages would cover roughly 8 megabytes, a little less than 8 megabytes.

13:55  
So that's a huge 2 pages will cover 8 kilobytes and we are reading 8000 bytes.

14:04  
So you might be right.

14:06  
It's not not we are at least in the in the right order.

14:10  
So, so one possible question is 2 pages, would everybody agree or is is it wrong somehow?

14:21  
Everybody agrees with with your colleague in the first, in the second row, volume up, please.

14:29  
I don't hear right now.

14:34  
First, we are focusing on the data pages only.

14:36  
So at the moment, the first question is we are ignoring the translation process.

14:41  
We are just asking how many data pages we are accessing.

14:47  
You agree with two, OK, Everybody agrees with two, OK, I don't, sorry.

14:54  
It might be two.

14:55  
Sometimes it's 2, but sometimes it's not.

15:02  
Why?

15:04  
Well, if you have the data array perfectly aligned with the pages, this is the memory.

15:11  
OK, where this is going, If it's misaligned, then we might be looking at the.

15:22  
Yeah, that that's, that's the.

15:23  
That's my point.

15:23  
So let's just assume that these blocks are pages, right?

15:28  
So one option is that the data structure, the array starts aligned with a page, so it starts where a page starts.

15:39  
In that case, it will take 2 pages, slightly less than two pages.

15:43  
48 bytes will be here to spare, right?

15:47  
But it also might happen that this data array starts over here right before the end of this page.

15:57  
In that case it will take it will access this page because there will be small amount of data here, then the entire page afterwards, and then some portion of another page here.

16:08  
Probably not.

16:09  
Not entire page, there will be slightly larger trailing space over here, but it will access three different pages, right?

16:17  
So when you when I'm asking these questions, you need to ask yourself how this data pointer can be or cannot be aligned.

16:25  
And if we don't know anything, you need to assume the worse.

16:29  
So if we are asking how many data pages, the answer would be two or three depending on where the data pointer actually points to.

16:37  
OK, so you weren't completely wrong, you have half of the question right, but you need to also consider bad situations or other situations where the pointer might be slightly differently aligned.

16:50  
OK, another question, what is the minimal amount of page faults?

16:55  
So if you have an optimistic scenario, how many page faults will this operation perform?

17:05  
Zero.

17:05  
Yeah.

17:06  
The most optimistic scenario is that everything is there, all the data are allocated, all the translations in the page table are filled properly, all the frames are there, everything is fine.

17:17  
So normally, and this is actually what happens in in.

17:20  
In normal situations, if your code runs for some time and all the data are already cached and everything, then you usually experience 0 optimistically.

17:31  
If everything runs smoothly, you usually have zero page faults.

17:35  
So that's not a very interesting question, right?

17:38  
So the interesting question is what's the worst case scenario?

17:42  
So what is the maximal amount of page faults?

17:45  
The pessimistic scenario?

17:55  
What's the worst case that might happen?

17:58  
Hypothetically slightly different 2000 times and it can completely be run by all.

18:10  
The word is 8000 accessing each individual byte of the integer.

18:16  
Well, yes and no.

18:19  
Well, let's, so let's assume that the architecture reads the integer and as an M1 operation.

18:26  
So yeah, there are architectures that actually reads data byte by byte like Spark.

18:30  
But usually normal architectures like MIPS or X86, which you were presented in these classes read the integer as a one operation.

18:39  
So the integer is read as let's say one load.

18:42  
I've got, I've got a 3rd guest then OK, the maximum number of page faults is 1 because it holds the code.

18:53  
No, I I mean no, no, no, no.

18:55  
Actually assuming that each page fault is duly, let's say, handled by the operating system.

19:02  
Otherwise the the the question has no meaning.

19:04  
So yeah, the page fault will like trigger the operating system that needs to somehow handle the situation.

19:11  
And I'm assuming that the data are valid, like the data are in the properly allocated part of memory.

19:17  
It's not like that you are you have a new pointer or or some invalid pointer that points somewhere to the unallocated part of the memory.

19:27  
In that case you were you would be correct because in that case the data, the first access to the data pointer will cause the the operating system to kill the application.

19:36  
So yeah, but that's not very interesting scenario.

19:40  
So let's say just this code is valid, it it runs properly, it will perform what we are expecting it to do.

19:46  
We just assume that in the background actually transparent to the user.

19:51  
This happens on the background user doesn't actually know it.

19:53  
You might read some statistics from the operating system, like how many page faults happened during some time or so.

19:59  
But technically you are ignorant to this effect.

20:03  
You don't care about how many how many page faults are there during your execution.

20:08  
If you run something in Python, you don't care how many page faults will happen.

20:13  
But now we are studying operating systems, so we are interested in such stuff.

20:17  
So we would like to know how many page faults might might happen.

20:21  
So 1 isn't the answer I was looking for, and 8000 is somewhat or 2000 is somewhat.

20:29  
Yeah, technically that might be possible, but again, it's not completely accurate.

20:37  
So if I want to be accurate, the right number would be Infinity because I don't know if there are enough frames in the system it can accommodate both the data and the code.

20:51  
So what might happen is that the data are there, then when I try to load the, the code is there.

20:57  
When I try to load the data, the code got swapped away.

21:01  
So technically I cannot do both.

21:04  
I cannot fit this code in the memory and it will end up in a.

21:08  
Another thing that might happen is that every time I try to read the data, it will actually swap something from the memory and then during the swap this thread this this process will be paused, another process will kick in, it will then swap again the the array swap memory.

21:26  
So it will be like trashing it will only swap the memory.

21:29  
So the worst, yeah, so but this is like just a hypothetical scenario.

21:35  
Infinity is again not very, not very likely answer, not very interesting answer.

21:42  
So let's let's just rephrase this.

21:45  
What is the I can I can rephrase it in this way?

21:49  
How many distinct pages may cause page 4 in the worst case?

21:53  
So let's assume that each page can fold at most once we have enough frames to accommodate the pages if we need to.

22:00  
Because if we don't have enough frames, then we are doomed anyway.

22:04  
So let's let's not dwell on these awkward scenarios.

22:08  
Let's assume that we have enough physical memory, just it happens that the process was sleeping for a long time or something, so the the memory isn't swapped in or isn't allocated here or something.

22:21  
So we need to properly fill in all the frames.

22:24  
So how many distinct pages may cause the?

22:26  
That's the the most interesting question, worst case scenario.

22:39  
It's not three or one.

22:40  
We are assuming the worst case scenario, so the the largest amount.

22:45  
So if you have like multiple numbers, it will be the maximum of them.

22:47  
So you're saying 3.

22:49  
So yeah, three are for the data pages.

22:52  
But now I'm asking about all the pages.

22:58  
So yeah, we have three data pages.

23:00  
So let's add 3 to this number because we will definitely need three data pages that we need to read.

23:07  
That's it, yeah.

23:08  
But we might need more because there is this translation process.

23:14  
We have two level paging which might also require some extra pages to be loaded.

23:27  
Yeah, if we have two level pages, that gives us two additional like pages that eventually have phones, exactly because the first level table is always there.

23:38  
It has to be loaded.

23:39  
That that's like a rule because if you don't have the first level there, then you're doomed.

23:45  
It won't work.

23:46  
So we don't count the first level, but at the second level.

23:50  
The second level of pages might be swapped away or might not exist yet.

23:54  
So we need to create them, allocate them.

23:57  
The operating system needs to fill them in if they're missing.

24:00  
So how many these second level page tables we need to cover these three data pages?

24:08  
We know that these three data pages are in a continuous region.

24:12  
The area is continuous.

24:13  
So these 3 pages are one after another.

24:16  
They have like incremental addresses, incremental if you chop off the offset.

24:24  
The maximum is what we are interesting in.

24:26  
Yeah, hopefully one because obviously 3 entries would fit 1000 entry page entry table.

24:32  
But in the worst case you are right.

24:35  
If this this page requires the last entry for instance of this table, then these two will be in another table.

24:44  
So yeah, 2 is the is the right maximum.

24:48  
So we need three data pages and two second level pages.

24:52  
Anything else?

24:58  
These situations are always more complicated.

25:00  
Yeah, I know, I know.

25:00  
I'm I'm asking questions like which are somewhat embedded with hints.

25:06  
Because when I'm asking these stupid questions, I obviously the code itself.

25:12  
Exactly what about the code?

25:14  
We need to ask the questions about the code as well.

25:16  
So is the code present?

25:19  
We made no assumptions about that.

25:21  
So theoretically this code also may not be present when we start executing it.

25:26  
It the whole application was swapped away because it was sleeping for a long time, for instance.

25:31  
So if this code is also swapped away, how many pages will be required for this?

25:40  
Again, maximum is the number that that's interesting.

25:42  
So yeah, this is this seems that this would be definitely more than one or two instructions, right?

25:48  
So if this is more than one or two instructions, it may span over again, because if the one instruction is at the end of one page, it can span over 2.

25:57  
And then again, if you take a good look at it, it won't probably take more than one page of code, right?

26:03  
This, this won't be like thousands of instructions.

26:05  
It's just a simple for loop and, and some addition.

26:08  
So yeah, this might span over two data code pages, and two code pages will require how many again?

26:22  
2, because in the worst case scenario, the first page may be may have entry here and the second page will require another table.

26:30  
So four additional pages for the code.

26:34  
Yeah, you need to ask this question kind of separately because there might be an assumption that the code is already in the memory.

26:41  
Yeah, if the code is already in the memory, then you're counting only the data or vice versa.

26:46  
And again, I would ask these questions only in situations when it's obvious how large this code could be because yeah, we have no assumptions regarding the compiler, etcetera, but that this code would be larger than one instruction and and it it won't probably take more than 1015 instructions.

27:05  
That's kind of thing that should be clear to everyone.

27:09  
OK, so that was like a few questions.

27:11  
We get to few of these questions at the end when I was when I will talk about how the how the test will be conducted.

27:19  
OK, what if we have a larger virtual address space or even larger physical address space.

27:26  
So if we have more bits to spare, more bits to allocate, then yeah, we need more levels of the page tables.

27:33  
Actually this is what they do in the AMD when they were extending the XA6 architecture into 64 bits and the best solution they come up with was to add two additional levels.

27:48  
So the principle is all the same, we are just adding levels.

27:51  
Yeah, we need to chop the address into more, more segments, more fragments.

28:00  
And yes, we cannot possibly accommodate it all.

28:03  
So technically the 64 bit architecture, at least this architecture is not entirely 64 bit architecture.

28:12  
It can only take first 48 bits of the address and translate it.

28:19  
So yeah, you you're being cheated.

28:22  
We are telling you that we have the 64 bit processors and 64 bit address spaces.

28:27  
But no, in X86 ext to 64 bits there are only 48 bits actually usable for the process.

28:37  
So what are the nothing you can't use?

28:45  
Well, you can use them for additional stuff, but they're like ignored or something.

28:50  
So you can use them for the address.

28:54  
Sorry.

28:55  
And another thing that changes is that we are using only 9 bits for each entry.

29:00  
Oops.

29:01  
Why?

29:01  
Well, The thing is that we are still keeping 4 KB pages, so we want each table to fit one page.

29:09  
Why?

29:10  
Because it's much simpler to handle.

29:12  
Technically this is not like a hard prerequisite, but it's like it's a good one because if you have, if you manage to fit each of these tables exactly into one page, it's much more easier to manage.

29:25  
So we need each of these tables to fit 4 kilobytes.

29:29  
But then again we are using 64 bits for addresses.

29:32  
So when we are using 64 bits for each address, we need to extend these entries.

29:37  
This entry cannot can no longer stay small enough to fit 4 bytes.

29:42  
We need 64 bit entries.

29:44  
Well, not entirely 64 bits we can.

29:46  
We have some bits to spare, but we won't fit it into 32 bits.

29:50  
So we need to double the size of the entry.

29:54  
And we if we double the size of the entry and we keep the whole structure of the same size, we need to half the number of items in this in this page.

30:04  
So technically we are not using 1024, only 512 of these entries and each entry has double size.

30:12  
Each entry takes 64 bits, 8 bytes OK, but everything else is the same.

30:19  
And by the way, how large is each entry can be easily inferred from the bits the architecture used.

30:27  
If we have 32 bits, we can use 4 bytes.

30:30  
If we have 64 bits, we need to use 8 bytes.

30:33  
It's pretty straightforward in this way.

30:35  
But again, this is like a thing you can derive or sometimes you must derive from the knowledge of the architecture.

30:42  
So this is what happens.

30:43  
And that's also why modern architectures don't want to rely on this translation process.

30:48  
And it's completely essential.

30:50  
There are some caches like TLB the translation look aside buffer which actually caches the translations.

30:58  
Because in 64 bit architecture 1 memory translation will go through 4 distinct tables.

31:06  
So additional 4 reads to the memory needs to be performed to translate 1 virtual address into physical address.

31:14  
And that's a lot.

31:16  
So we need, we need these caches, we need the look up side look aside buffers to keep all this data cached.

31:23  
So this this translation isn't performed every time with every access, but only when you're entering a new page, basically.

31:30  
OK, so this is just an example of of a real actual translation mechanism.

31:35  
This is taken from a manual of the AMD 64.

31:39  
And let's move on a few more things to finish.

31:43  
Yeah, this is a formalization of the process and a few more additional information.

31:50  
So yeah, the steps I've already depicted to explain in detail on the graph.

31:57  
So let me skip some points.

32:02  
As I mentioned, the caching is very essential.

32:05  
So basically the first step would be let's check the cache if the translation is directly there.

32:10  
Note that the in the TLB, of course we are caching only the virtual, only the the page part of the virtual address.

32:18  
We are chopping the offset as a first step even before we try accessing the caches because it would be wasteful to add the offset into the cache that that's, we don't need that only the page index part is being translated.

32:34  
Yeah.

32:34  
Then we go through the page tables.

32:36  
If we miss, if we miss the TLB, we go through the page tables again.

32:40  
The page tables themselves can be in the regular caches.

32:43  
So again, caches might help.

32:46  
And note that in these tables there are some additional bits.

32:51  
I haven't spoke, I haven't spoken about it in in a detail.

32:55  
But if we check, yeah, if we check this, then you might note that the translation itself takes the translation entry itself takes 32 bits in this in this architecture or 64 bits on 64 bit architecture.

33:11  
But only first part, only the 1st 20 bits without the offset is actually required because the offset is provided from here.

33:22  
So we need only the 20 bits of the physical address to be translated.

33:27  
So we have 12 bits to spare here, right?

33:32  
So for what?

33:33  
We are using these bits.

33:34  
So first we need a valid bit.

33:36  
That's very important.

33:37  
We need some bit that will indicate whether this entry is valid or not, whether this translation exists or not.

33:45  
Yeah, whether this is a valid value.

33:47  
So there is one extra bit, one bit which indicates this is a valid value.

33:51  
This is an invalid value.

33:53  
And also we need some additional bits, not.

33:56  
Most notably, we take the excess bit and dirty bit.

34:00  
Yeah, and the excess bit says it's updated every time the page is accessed.

34:05  
So it's set to 1 every time this translation occurs, and dirty bit is set every time a write operation occurs, so every time a modification to this page occurs.

34:16  
And this is done automatically by the hardware.

34:19  
Yeah.

34:20  
And finally we assemble, we concatenate the offset and the and the physical address which is translated.

34:29  
Yeah, this is just the formalization of the translation.

34:31  
I can skip that for for time reasons.

34:35  
I've already explained that on the on the on the diagram.

34:39  
So I hope everybody got that.

34:40  
If you don't, then you have it one more time spelled out on the on the slide and what happens during the translation if something goes wrong.

34:51  
So if something goes wrong and what, what might go wrong?

34:55  
The most often, most typical thing that goes wrong is that the invalid bit, the valid bit is set to 0.

35:02  
It's the the entry isn't valid.

35:04  
So the translation fails because somewhere during the the the look up process in the page tables it comes across an entry which is not valid so it cannot be translated.

35:15  
So what happens is that this the fault is risen so the then exception within a CPU is thrown and the handler of the operating system is executed.

35:27  
So the CPU switches into the kernel mode and starts executing the routine that was provided by the operating system.

35:35  
And this routine, the 1st order of business, it is that it needs to determine why this happened.

35:43  
Because yeah, one option, as I mentioned before, is that something bad happened in your code.

35:49  
You are trying to do something you're not supposed to do by an accident, by design, it doesn't matter.

35:54  
So the operating system is there to protect you or in this case, maybe to protect the other processes from your process.

36:02  
So technically, if an unauthorized access is detected, you are trying to access a memory where you shouldn't be accessing or.

36:11  
This might be also happening when you're trying to write into the read only memory.

36:16  
That's another flag that might be there, mark some memory read only so it cannot be written.

36:21  
This is usually used for the constants or for the code because you don't want to overwrite your code once the application is executing.

36:31  
Also we have the execution bit, so the pages that holds the code are marked with execute bit, so the instructions may be loaded from them, but the data are have this bit set off so the data cannot execute code.

36:45  
So these are types of errors that might happen and if they do the operating system has no option but to kill your application.

36:54  
Usually you get some report, some you might get some dump or something.

36:58  
But technically if your application, if your command line application ends up with segmentation fault or Linux, it's signal 11.

37:06  
If you end up there then this happened.

37:10  
You are doing something wrong with your virtual memory.

37:13  
You are trying to dereference a wrong pointer.

37:18  
Another option is that the translation is actually valid.

37:22  
The address should be valid, but the translation does not exist yet.

37:27  
So the operating system was lazy or have had some reasons on its own why this translation isn't filled in yet.

37:35  
But it's a valid address, it should be filled in.

37:38  
So in that case, the typical 2 examples would be one that I've already given you.

37:44  
You've allocated a huge block of memory, but this memory is given to you only at the virtual level.

37:50  
You don't really get the memory physically.

37:53  
And the operating system uses this mechanism to slowly, lazily give you the memory as as as it goes, as as you access the pages.

38:04  
Because technically it might happen that you allocate one terabyte of memory.

38:09  
Why not?

38:10  
It's not forbidden, right?

38:11  
Right.

38:12  
But you don't want to access it all.

38:15  
You just want to have a good feeling that you like own one terabyte of memory and in the end you just access a few few pages.

38:23  
So only these few pages will be actually assigned a physical frame where they will reside and the rest will be like virtually there.

38:31  
You can theoretically access them if you want, but if you do, you will be provided the memory up to the point when you run out of the memory.

38:42  
But if you don't access it entirely, it might work.

38:45  
Your application will run.

38:47  
Another possibility is that the pages were swapped away so they were offloaded to a persistent storage, like to the hard drive, so that we have more room in the physical memory.

38:58  
For instance, your application wasn't running for some time and the operating system decided it needs more memory, more physical memory for other processes.

39:06  
So it takes your pages and saves them to the hard drive and mark their places as not mapped because they were swept away.

39:17  
So when this happens, the operating systems say, Oh yes, I've just moved these pages, move these frames out of the main memory into the hard drive.

39:25  
So yeah, now, now it's time to load them back.

39:28  
So what happens here is that the memory is provided either as a newly allocated memory, as a new free frame, or it's loaded from the hard drive, from a persistent storage, for instance.

39:38  
So the operating system performs this operation for you and when it's when it does it re executes the instruction that causes the fault.

39:47  
So your applications kind of transparently gets the memory where it was, where the operation where the data operation was trying to read or write the data.

40:00  
OK.

40:01  
Yeah, you may theoretically detect these operations of the of the operating system.

40:09  
By measuring time, for instance, if you carefully measure time, you might detect some discrepancies, like when this happens often, frequently, then your code is kind of running slower than it should be.

40:22  
So yeah, there are ways how to detect that some something is happening.

40:26  
But otherwise, besides timing, this goes completely smoothly, seamlessly, transparently for your application.

40:34  
So you don't want to detect that it happens automatically by the operating system.

40:42  
OK.

40:43  
So yeah, when this happens, the operating system needs to find a new frame, create the mapping and either load the contents of the frame from the hard drive or fill the frame with zeros or something if it's allocating new memory.

40:57  
So it will give you the data if you give you the memory and construct the corresponding page table entries, etcetera.

41:06  
The tricky part is that we need to find a free frame.

41:10  
Yeah, I told you before that we have enough memory.

41:12  
So we have enough frames.

41:13  
Yeah, usually we do, but sometimes we don't, and sometimes we run out of frames.

41:18  
And what happens then?

41:19  
Well, this is the easy part.

41:21  
There is one frame waiting for us, so we just take it.

41:24  
But it might happen that all the frames are being currently used.

41:28  
By the way, there is a saying, if you're not using 100% of your physical memory, then you're under utilizing your PC, right?

41:36  
So the operating system actually is trying to utilize as much frames as possible for various internal things.

41:44  
Like the operating system might cache various files from the hard drive in the main memory.

41:50  
They may not be that important.

41:52  
So every time this happens, it just dumps some of this cached data from the hard drive, throw them away and reclaim the frame for your process.

42:02  
But technically, by the way, sometimes try to run top process on your Linux and you usually find out that there is a very little amount of free memory.

42:13  
Most of the memory is being used.

42:15  
Sometimes you can see that most of the memory is being used for IO buffers.

42:19  
That's what the system is doing.

42:21  
It's allocating the memory for itself so it can cache data from the hard drive if the memory isn't used.

42:26  
And that's a good thing because you are actually utilizing your hardware to its full potential.

42:35  
But when this happens, we need to find the victim for swapping.

42:38  
And if this happens, we need to use some algorithm for replacement, save the victim to the persistent storage, or possibly throw it away if it's if it's no longer used, and then remove its mapping from the TLB.

42:54  
That's again something that might happen if you if you're trying to swap in things.

42:59  
You also need to take care of the caches.

43:01  
Some some somehow dump the TLB entries so they no longer contain invalid values.

43:08  
OK, few page replacement algorithms.

43:11  
By the way, these replacement algorithms work for other things as well, for instance for cash cache replacement.

43:17  
So it's like every time you have a limited resource that can accommodate only copies of few things that these replacement algorithms can be, can be used.

43:27  
Actually there is something called optimal page algorithm.

43:32  
It replaces the page that will not be used for the longest period of time.

43:39  
Yeah, this is a future tense.

43:42  
So if we have an Oraculum, if we have some divine power that can tell us which of the pages, which of the frames which are currently allocated will not be used in the future for the longest possible time, then that's the best victim to choose because we will run for the longest time without actually needing it.

44:02  
Yeah.

44:02  
And that's something we cannot do.

44:04  
So it's like a hypothetical, it's a theoretical algorithm.

44:08  
It's like an optimum towards.

44:10  
We are trying to strive, but we are the only thing we have is the past data.

44:15  
So we are trying to assume from the past data how the pages will be used in the future.

44:21  
So it's like kind of like reading the cards or tea leaves or something.

44:28  
OK, few examples how this works.

44:33  
Very, very simple example.

44:34  
Let's let's utilize the excess bit and let's have a like an arrow, like a clock.

44:40  
So we organized all possible frames in a circular fashion.

44:45  
And this arrow points to always points to 1 frame and it moves as a clock.

44:52  
So it moves as it goes.

44:55  
And every time it encounters an sorry, every time it encounters a page sorry A-frame with access flag set, it resets this flag to zero and moves on.

45:07  
So it resets this flag to zero and moves.

45:09  
And once it finds a page A-frame with 0 flag.

45:13  
So it this frame wasn't accessed, this page wasn't accessed recently because it already has its access flag set to 0, so it takes this one.

45:23  
So it tries to take the 1st frame which has a page with a set to 0 because this page wasn't used recently.

45:33  
Actually there is an extension of this algorithm called not recently used replacement algorithm by recently.

45:41  
We just have some flex that indicate whether a page was recently in some small period of time in the past, whether it was used or not.

45:52  
In this extension we are using not only the access flag, but The Dirty flag as well.

45:56  
Because it's a good idea to distinguish whether the page is only accessed for reading or whether it was actually modified.

46:03  
Because when it was modified, it probably needs some write back.

46:06  
It needs to be copied back to the hard drive or something.

46:09  
So technically we just divide the pages in four classes based on these two bits.

46:16  
Yeah, quite easy actually.

46:17  
We just take these two bits as values.

46:20  
And yeah, these are the classes.

46:23  
So if the page wasn't accessed for some time, there are two lower classes for the access pages, there are two upper classes.

46:30  
And The Dirty flag is like the helping deciding factor which class we will select.

46:35  
And the algorithm is very simple.

46:37  
We are trying to take a random page from the lowest occupied class.

46:41  
So first we are trying to look at are there any pages which with both flags set to 0?

46:47  
If they are, we just pick random page, random frame from this set.

46:50  
If not, we are we move here and we're trying to pick a page from this set.

46:55  
If not we are moving upwards.

46:57  
And finally, if there are no pages with these flags set to 0, we will pick one page which was recently accessed and recently written.

47:08  
It's not much, but it's honest work.

47:11  
OK, least recently used is a slightly more difficult.

47:19  
Again like the previous algorithms, it tries to derive something from the past.

47:23  
And what is this?

47:25  
It actually, it tries to prioritize the page that was just actually right now used.

47:34  
So technically what we can do is every time a page is accessed, it's moved to the front of some queue.

47:41  
So you're maintaining a ordered list or a heap or something of of pages.

47:48  
And every time one page is used, you move it on the top.

47:52  
So you know that on the top there is the most, there is the least, the most recently used page.

48:00  
And on the bottom, on the end of the list there is the least recently used.

48:05  
So if you maintain this list at the end you will find the least recently used page.

48:11  
The thing is that this is very complicated.

48:15  
I'm not sure if you ever implement this algorithm.

48:18  
It's very useful algorithm.

48:19  
It's used in compression, for instance, but it's difficult to do so because you need to find the page in the list or in in any other data structure.

48:28  
Then you need to take it from, extract it from the list and then move it to the front.

48:33  
Yeah, it's it's somewhat difficult.

48:35  
So there are some approximations that can be used instead of that.

48:39  
But still this is not very frequently used.

48:42  
Sorry, sorry for the word.

48:45  
This is not one of the first choices because it's quite complicated.

48:49  
There are some some versions how to implement it using some caches or some bit matrices in in hardware.

48:54  
But again, this is very complicated.

48:57  
Another quite complicated but slightly less complicated algorithm is not frequently used and by frequently notice the change in the word.

49:06  
The frequently indicates that we are trying to compute some sort of frequency how the pages are being used.

49:12  
So some sort of a number that will indicate whether the page is used frequently or less frequently.

49:18  
So it won't be a simple bit, it will be a number.

49:21  
And this number is usually derived or somehow computed from the excess bit.

49:26  
So every once in a while, periodically, this process scans all the pages available and all the pages which has this bit set to 1 have this counter, this frequency counter incremented.

49:41  
So basically this bit is kind of added to the to the frequency counter.

49:47  
And when it's added, it's always cleared.

49:50  
So they take the bit, they add it to the counter and then they clear it.

49:53  
So the excess bit, by the way, the excess bit may be set to 1 multiple times.

49:59  
The page might have been accessed multiple times before this, before this two epochs of this.

50:05  
Actually happened.

50:06  
But this is like accumulated like a aggregation of the excesses.

50:11  
We are only concerned whether this was the page was accessed.

50:14  
We don't know how many times during this period of time, but there's like an aggregation of information.

50:19  
No problem.

50:20  
And when we try to find the victim, we are trying to find the victim with the lowest counter, lowest frequency counter.

50:28  
So that's the page that was in the past least frequently used.

50:34  
Again, this is kind of complicated because you need to scan all the pages and take all the counters and find minimum.

50:41  
But still finding minimum is somewhat easier than moving a page to the front.

50:45  
So it's, it might be easier in some, some cases, but then again, it's again, it's not ideal.

50:54  
And also there's one problem that needs to be solved because if you have a page that was very frequently used at the beginning and then it was, it will be used no more, then it will create a problem because this page would remain in the memory forever because it has a huge frequency counter.

51:12  
So we need to introduce aging.

51:14  
So each frequency counter needs to be somehow decreased over time.

51:20  
And what happens is again, periodically, usually with a greater.

51:23  
Than this one, we somehow decrease.

51:28  
For instance, we can divide them by two because that's a simple shift.

51:31  
That's a like a very easy operation to perform.

51:33  
So there's another algorithm that can be used for a page replacement.

51:37  
A few more things that can be derived from paging.

51:41  
By the way, if we have the memory virtualisation, sharing memory between two processes is like a thing that is already implemented there.

51:52  
You just need to properly set the page tables.

51:54  
So if you need the 2 processes, 2 applications share some data, easy thing to do.

52:02  
You just make sure that the page tables, the translation process points to the same data in physical memory.

52:10  
So if you want that application one in the virtual address space one share some data with application 2, the only thing you need to do is that you create appropriate mapping for the application one and for the application 2 so they both access the same frames.

52:28  
That's it.

52:29  
Nothing else needs to be done, OK?

52:32  
So it's like a very easy, if you want 2 applications to share a data structure, this usually happens automatically, transparently for the code.

52:42  
Because usually if you run a piece of code in Linux, let's say, you definitely need some libraries like libc at least.

52:51  
At the very least you need libc.

52:52  
So this library is required by every application.

53:00  
OK, no problem.

53:02  
It needs to be in the physical memory only once.

53:05  
And every application creates this mapping to this part of physical memory and that's it.

53:11  
So it can be shared among all the applications.

53:13  
Yeah, we need to take some precautions.

53:15  
For instance, it should be marked as read only for all the applications.

53:20  
Otherwise one application can rewrite the code for another application.

53:23  
That's really bad, but that's the only thing.

53:25  
We need to make sure it's there, but otherwise it's pretty straightforward.

53:29  
Otherwise, the translation mechanism, which is embedded in the CPU will take care of all the technical details of this process.

53:36  
So it's just a matter of proper setting.

53:39  
Another thing that we can do is memory file mapping.

53:46  
What's memory mapped files?

53:48  
What happens here if we take part of the virtual address space and we say, hey, I'm saying that this part of virtual address space actually mirrors a file on the hard drive.

54:03  
Again, this is just something I need to tell the operating system.

54:07  
And the operating system will ensure that every time I'm accessing this memory, it will load the appropriate data from the hard drive to my memory.

54:16  
So it will automatically allocate some data frames, load the data from the hard drive to these data frames, and provide appropriate mapping so I can access the data as if I'm just reading them from the file.

54:29  
And vice versa, I can modify this memory and the operating system will ensure that when time comes this data will be transparently written back to the hard drive.

54:40  
So it works.

54:41  
It might work both ways actually.

54:44  
This mechanism is like one of the most frequently used mechanisms in operating system, because even if you're trying to read the file using the old fashioned F open and F read and other operations, somewhere underneath the operating system will perform this, at least on the operating system level or somewhere in some library you are using on top of underneath your code.

55:07  
So this will always happen.

55:10  
There are no other mechanisms used nowadays in modern operating systems, so it's always there.

55:18  
By the way, if you're allocating new memory, it's also handled using this mechanism because there is some swap file or some helping files for memory allocation.

55:29  
So this mechanism is always there even if you don't know about it.

55:35  
Few more things to wrap up the operating system.

55:37  
So yeah, before I have these few final slides about virtualization, are there any questions regarding memory translation, memory virtualization, memory management?

55:47  
I just missed a lot of stuff like we'll have to rewatch the.

55:56  
Well, yeah, hopefully you will do it in a due speed so I you won't waste too much time on it.

56:01  
But then again, everything is pretty straightforward.

56:05  
So if you just run through the slides, I hope that everything will kick back to you and you will just refresh your memory because it's not that complex.

56:13  
To be honest.

56:13  
It's not that complex.

56:14  
It's pretty straightforward.

56:15  
You just, if all the pieces fall into place, then you'll be fine.

56:21  
There are no special complexities like in mathematical analysis.

56:26  
You know where you really need to know the proofs and everything.

56:31  
No, this is quite straightforward if you just think it the right way.

56:34  
About the computers.

56:36  
OK, a few final slides about operating systems.

56:41  
This is a very huge topic, so I'm opening it only for a few moments.

56:46  
And the modern operating systems, modern systems needs to employ some sort of separation which goes beyond normal memory control.

56:59  
So we need to separate individual applications, individual processes into more secure containers, more secure sandboxes than what the regular memory virtualization will do for us.

57:14  
So what we are, what we can do is that we introduce virtual machines or virtual containers into the operating systems.

57:24  
Just to clarify, virtual machine is a machine or like an illusion of a machine which is emulated inside your regular hardware machine.

57:37  
How this can go well?

57:38  
Imagine the easiest thing you can do is that you take your Linux, Windows, whatever you're using and you install application like VirtualBox for instance.

57:47  
You can do that for free and it's like a regular application.

57:51  
But this regular application will create something like a virtual PC inside your regular PC and you can run it.

57:59  
It looks like you're running an application with one window, but this window is actually emulation of the screen of the PC which is inside.

58:07  
And you can install a different operating system for instance inside.

58:11  
And the whole disk of this virtual machine is stored as one file of your regular operating system.

58:18  
So technically, you created an emulated PC inside your regular PC and you can do whatever you want in, in this virtual environment.

58:28  
I mean, you, you can destroy it without worrying about too much because the only thing that happens is that you, you just erase one file from your, from your regular host system and you erase the whole virtual machine.

58:42  
It's very easy.

58:43  
So if you need to do some experimentation, for instance, a few weeks ago, I need to set up some Chinese hardware, some reader from China, which was very cheap.

58:55  
And it has the software also made in China, which, which can set up this, this device which can configure it.

59:03  
And this, this software was distributed only as a very small exit file, which I needed to download for a very suspicious website.

59:12  
So at this point, yeah, no problem.

59:14  
I just run a virtual virtual machine.

59:16  
And in this virtual machine, I have no, let's say, anxieties about running this piece of software, because if anything goes wrong, I just delete the virtual machine and my original system will be intact, it will be fine.

59:29  
So this is 1, let's say, use case for which we can use the virtualization quickly, serious and intent, but maybe not so much the subject.

59:42  
So it's obvious to me that you could put a virtual machine inside of another virtual machine.

59:46  
Yep.

59:47  
About how many layers deep could you go before?

59:50  
I think theoretically infinite, but you're limited with memory.

59:53  
So this virtualization takes some overhead.

59:58  
So in the end we will run out of memory because yeah, you need some extra memory to manage this virtual box.

1:00:06  
So if you run a virtual box in a virtual box in a virtual box, then in the end you will run out of memory and in the end it will be unimaginably slow.

1:00:13  
So if you do just one layer, you're fine because current hardware has a huge support for virtualization.

1:00:21  
So if you're trying to run this VirtualBox as I just described, what really happens is the D applications running in this virtualized operating system run in the same way your applications in your regular operating systems run, meaning they run the same instructions they are.

1:00:40  
Only the kernel course are redirected to the virtual operating system.

1:00:45  
So only the kernel mode of the CPU needs to employ some additional security layers and current CPUs allow that.

1:00:54  
So they allow you to run in a virtual kernel mode.

1:00:58  
I'm not being very precise, it's slightly more complicated, but technically you are like, have a regular operating system and then you can run a virtual kernel mode of the CPU.

1:01:09  
So it runs almost as fast as the regular instructions, but certain operations are intercepted and the the driver or the the execution control is turned over to the original operating system so it can emulate the virtual operating system or it's it's operation.

1:01:30  
So it's quite fast that that's my point.

1:01:32  
There is a huge support in that.

1:01:34  
In hardware, if you go into the nested levels, you are losing some of the support and in the end if you go deep enough you might end up with complete emulation.

1:01:46  
You can, you can always do complete emulation.

1:01:48  
There are tools for that like QMO, which actually allows you to emulate a CPU completely.

1:01:54  
Like the instructions are executed in the virtual machine, right?

1:01:59  
But that's very slow because for executing each virtual instruction you need to perform 10s or even hundreds of regular instructions.

1:02:05  
So there is at least one order of magnitude slow down for that.

1:02:09  
You don't want that.

1:02:10  
You want at least your applications to run almost on par with real hardware, so that each regular instruction in your code in application code is really one instruction on the CPU.

1:02:24  
Mostly.

1:02:25  
Not entirely, but almost.

1:02:26  
I guess a rough number on let's say I have a laptop, normal laptop, about how many layers I, I don't know, I, I think you can do safely 2 levels, 2 levels isn't too much to ask for.

1:02:40  
So I think on a regular laptop, even on a regular laptop, you probably will be fine with two levels, but I don't know.

1:02:47  
And actually who cares?

1:02:48  
I, I really don't, don't need more than one layer for me.

1:02:51  
So, and, and most people don't.

1:02:53  
So yeah, 2 levels are maybe sometimes, sometimes you might need 2 levels maybe, but that's it.

1:02:59  
Don't worry about something that might be intentionally trying to get out.

1:03:02  
No, no, yeah, we can go to security, but maybe another time because that's a huge topic, but it's it's I wouldn't say it's impossible, but it's unlikely.

1:03:16  
OK, so this is one way to go.

1:03:18  
We can have a full virtualization, complete virtualization of a of a complete CPU.

1:03:24  
By the way, we do this all the time with the servers.

1:03:29  
We have this virtual infrastructure in our school of computer science.

1:03:34  
And most of the servers you are accessing, for instance, if you're accessing our web page, the web page runs on a some web server which runs in a virtual machine on this virtual infrastructure.

1:03:48  
Because they're only like, I, I think 5 servers, but these 5 servers are hosting at least 100 or maybe more than 100 Now, real virtual servers, because most of these servers are web page servers or similar similarly trivial things.

1:04:05  
So they don't need that much memory, They don't need too many CPUs.

1:04:10  
So we can manage to squeeze them on five bulky, but still only 5 blade servers.

1:04:17  
Another option is a process virtualization.

1:04:20  
We don't have to virtualize the entire operating system, but still we can benefit from virtualizing single applications.

1:04:28  
This is what happens if you have a special platform, Java or NET does this.

1:04:33  
So if you compile your Java code or if you compile your C code, you won't get real instructions.

1:04:39  
You will get byte code or you will get Silk code, which are virtual instructions designed for this virtual Java virtual machine or for this NET CR.

1:04:49  
So they are designed to run in a virtual environment.

1:04:53  
There is an application, actually the Java VM.

1:04:55  
Java Runtime is a virtual application that is taking this bytecode and executing it 1 instruction at a time.

1:05:03  
And then again, there would be a huge problem with performance because if really one bytecode instruction would be emulated by this virtual machine, it will take at least several, maybe 10 instructions.

1:05:15  
So there will be like an order of magnitude slowdown in the execution.

1:05:19  
We don't want that.

1:05:20  
Actually, this is what happened when you when you, when you try to execute Java One or Java 2.

1:05:24  
If anybody happened to know Java 1, I don't think so.

1:05:28  
It's like 30 years actually.

1:05:30  
Yeah, it's 30 years ago, almost exactly.

1:05:33  
So yeah, that's very, very slow.

1:05:35  
So what happens is that they try to compile this byte code into regular instructions, but they usually do it on demand.

1:05:44  
That's what this JIT goes.

1:05:46  
This JIT is just in time compilation.

1:05:49  
So it means that as you execute your code, the pieces of code for you are calling a new method in Java or in C#, it doesn't matter.

1:05:58  
So what happens is the the internal runtime says, hey, I, I haven't seen this method before, so let's compile it and it creates the the compilation on demand.

1:06:09  
It provides the real instructions on the target platform for that particular method.

1:06:14  
And then it injects it somehow into your runtime.

1:06:18  
Yeah, it's it's crazy, but it works.

1:06:22  
Actually JIT was originally used in Java compilation and still being used in Java and C# prefers AOT which is ahead of time compilation, which means when you're trying to run your C# program, the compilation happens all at the beginning.

1:06:39  
So, yeah, the startup of AC Sharp application takes a long time because at the beginning it takes the Silk code and quickly transpose it, translates it into the machine code.

1:06:52  
This translation is much faster than full compilation because it's just like one assembly interpretation into another assembly interpretation interpretation.

1:07:00  
So there's almost one to one mapping between the instructions.

1:07:03  
Yeah, please don't, don't tell this anyone to to don't tell this to anyone who actually do this type of translations because it's much more complex than I would just said.

1:07:13  
But but technically you can imagine that they are in the bytecode or in the Silk code.

1:07:16  
They are operations like add to numbers.

1:07:19  
So yeah, it can be translated into one or very few instructions of of the real CPU.

1:07:25  
And another thing you can do is containerization, which means that you instruct your operating system to create a sandbox, to create a small, let's say fence around some process.

1:07:37  
And this process is then executed in a safer manner.

1:07:40  
For instance, you can say, hey, this process is running only on the CPU, not accessing any other cores, or is given only this amount of memory, it cannot eat more memory than this.

1:07:50  
Or there is a virtualization of the file system, so the container actually is using only few folders from your file system or some virtual folders in your file system, so it doesn't access everything your computer has.

1:08:04  
And this is actually what is currently the most popular thing you're doing if you ever use things like Docker.

1:08:12  
You people have at least heard of Docker, right?

1:08:15  
Yeah, a few people are nodding.

1:08:17  
So yeah, Docker is a tool that performs this containerization.

1:08:21  
It's an application that helps you instruct the operating system how to wrap something into a separate sandbox so you can safely run it.

1:08:31  
And Docker has some additional features like the file system virtualization.

1:08:34  
So it allows you to run completely different file, file system or even completely different like a Linux distribution.

1:08:42  
If you have a Ubuntu, you can run completely different Linux distribution in this in this container.

1:08:47  
And it's also widely deployed in in industry because if you have a huge farm of servers, web servers or my SQL servers or whatever servers, you run that again, you want to deploy them easily with the same configuration, with the same libraries and everything.

1:09:04  
So again, you, you wrap them in a container and just deploy them on an infrastructure.

1:09:08  
And the infrastructure is only set up to run containers.

1:09:12  
You probably heard of Kubernetes, for instance.

1:09:14  
That's another buzzword.

1:09:15  
That's by the way, if you don't know Kubernetes, then there's a small reminder.

1:09:20  
H in Kubernetes stands for happiness.

1:09:26  
OK, so how this works, just a small visualization.

1:09:29  
Quick one, I'm, I'm really wrapping up.

1:09:31  
So this is how things run on a normal hardware.

1:09:34  
So you have a regular hardware, let's say say 6 architecture.

1:09:38  
On the top of the hardware there is the operating system.

1:09:40  
The operating system manages the hardware and provides the abstraction.

1:09:44  
Yeah, I'm closing the circle.

1:09:45  
I talked about this at the beginning of the operating system part.

1:09:49  
And the IT, it creates a layer between the hardware and regular applications.

1:09:55  
There might be many of these applications.

1:09:57  
Virtualization takes this and just moves it one layer upwards.

1:10:02  
So there is this virtualization layer which provides virtualization of the hardware for the operating systems.

1:10:10  
So technically the operating system isn't accessing the hardware directly, but through some virtualization level which allows some level of control.

1:10:18  
And it also allows a separation of multiple operating systems, multiple virtual machines running on the same physical hardware.

1:10:27  
So this is what what's it all about?

1:10:29  
Yeah, there are many things that needs to be resolved, like how we can virtualize CPUs, how we can virtualize RAM, etcetera.

1:10:37  
Some things are easier, like RAM, because we can just separate the RAM into multiple boxes and use the virtual address translation to properly manage the memory, memory separation.

1:10:48  
That's easy.

1:10:49  
With the CPUs, it's slightly more complicated, but as I mentioned before, there are mechanisms embedded in current CPUs that helps us with that.

1:10:57  
And with the disks or IO that's completely different, you usually need to virtualize them completely.

1:11:03  
So technically, if there is a hard drive over here, then hard drive over here is actually a file on some file system on this hard drive.

1:11:11  
So it's like you create a completely different abstractions for files or for iOS.

1:11:17  
OK, that's it.

1:11:19  
Any more questions regarding operating systems before we swap to the sorry, you mean this, this picture?

1:11:37  
So what?

1:11:38  
What was the question?

1:11:48  
You can imagine you can, you can abstract that the hard drive is basically a huge, huge array of blocks, right?

1:11:59  
A file is basically a huge chunk of bytes, right?

1:12:05  
So if I have a file on a regular file system on my regular Windows laptop, I can say, OK, I just managed to store a file system of another operating system in this file.

1:12:18  
I just say, OK, so I just managed to allocate the file by blocks.

1:12:24  
It doesn't matter because bytes are smaller than blocks, so no problem.

1:12:28  
So I just allocate large enough file so it can encompass.

1:12:33  
It would act as a virtual hard drive for a different operating system.

1:12:37  
And then the virtual layer will just transcribe all the iOS that goes to the hard drive directly into this file and vice versa.

1:12:47  
That's it.

1:12:48  
It's not that complex in the at least if we look at the basics, it's not that complex.

1:12:54  
The hard part is how to do this efficiently and and how to manage this.

1:12:59  
The storage and usually the virtualisation doesn't require allocating the entire file beforehand.

1:13:05  
Because if you have like one terabyte hard drive, virtual terabyte hard drive usually not using the entire hard drive, right.

1:13:15  
So you don't want to have a huge one terabyte file over here which will be mostly filled with zeros.

1:13:21  
So there are some techniques how to compress this, etc.

1:13:25  
And they are difficult, but if you look at the basics, this mapping is trivial.

1:13:31  
You just store the data over file system, which is like a huge set of blocks into the file.

1:13:35  
That's it.

1:13:47  
Yeah, that's a completely different story.

1:13:50  
If you use something like a virtual box and you actually set up shared folder, it goes completely the other way around because it's like you need to run a specialized application in here and there is a special, let's say, API in this virtualization layer.

1:14:09  
And this application is somehow aware of this special tunnel.

1:14:15  
And underneath the virtualization layer will allow you to access some files from the real underlying operating system.

1:14:24  
And this application will create an appearance.

1:14:26  
They kind of are created in the virtual.

1:14:30  
So it's much more difficult, it's much more technical, but you need the help of the both virtualization layer and you need some virtual application or usually it's implemented as a module or as a driver in your operating system, which you need to install additionally in your normal operating system.

1:14:49  
Usually there are some virtual tools, virtualization tools you need to install in the hosted system.

1:14:55  
So you get this kind of support and then this virtual part of the operating system, this special part is communicating via some specialized API which is not normally present in the regular hardware.

1:15:09  
So this is much more complicated.

1:15:14  
OK, let me do this.

1:15:18  
Let me do it this way.

1:15:18  
Let's take a break now because I have only 15 minutes till the regular end of the class.

1:15:23  
So let's take a quick break now and then we will continue with another topic, OK?

1:15:29  
So if anybody needs to go to the restroom or something, please do now and we will resume in five to 10 minutes based on how fast it will be.

1:15:50  
So let's resume our session with parallel programming.

1:15:56  
That's the final topic.

1:15:58  
Yeah, unfortunately we need to rush it slightly.

1:16:01  
So I won't dwell on details, but rest assured that there is another, another course called Programming Parallel Environment where we can go through this stuff in more thorough manner.

1:16:13  
So this is just to give you a few glimpse about the most important thing that's happening in the hardware for for past 25 years.

1:16:21  
So you may have noticed that your hardware is equipped with multiple CPU cores.

1:16:26  
If you buy a laptop, it has multiple CPU.

1:16:28  
If you buy a cell phone, it has multiple CPU cores.

1:16:31  
So this is like the basis for this for this course, for this lecture.

1:16:36  
And you need to be aware of this because if you want to utilize the CPUs, this hardware properly and efficiently, then you need to utilize all the cores.

1:16:46  
And then you need to resort into the parallel programming, a few terms at the beginning.

1:16:51  
Parallel.

1:16:52  
By parallel computing or programming, we refer to calculations or execution of code that really happens in parallel.

1:17:01  
So there are multiple execution units or processes or something that are carried out.

1:17:06  
They are computed, they are running simultaneously.

1:17:09  
At the same time multiple things are happening.

1:17:12  
This happens at multiple levels.

1:17:14  
You can do this at the bit level.

1:17:16  
Actually, the simplest example would be bitwise AND or bitwise OR.

1:17:21  
It's an operation that performs some bit manipulation on all the bits of one large word, right?

1:17:28  
It happens on a larger levels on instruction level.

1:17:32  
1st is that there is a pipelining, superscalarity and pipelining within the CPU.

1:17:38  
So the CPU core is executing multiple instructions simultaneously.

1:17:44  
You've seen the examples of the CPU architecture so you know that there are like 5 front end decoders.

1:17:50  
5 instructions are actually being decoded simultaneously.

1:17:54  
There are like 8 pipes, 8 ports with with different pipes.

1:17:57  
So there are multiple execution units that can crunch numbers at the same time.

1:18:03  
Also there are vector like instructions.

1:18:05  
We haven't talked about that much, but there are also specialized instructions that allow you to perform the same operation like addition or subtraction or even bit wise operation, etcetera on multiple registers or multiple data blocks or multiple values at the same time.

1:18:22  
And finally, most importantly, there is a task parallelism which allows you to execute multiple tasks, multiple pieces of code, multiple threads, multiple processes at the same time.

1:18:33  
And that happens thanks to the fact that we have multiple CPU cores, multiple execution units that are running independently within your PC, within your host system.

1:18:44  
There is also another term, concurrency or concurrent computing.

1:18:48  
This usually refers to the situation where there are multiple computations.

1:18:54  
Imagine processes that are available.

1:18:58  
They might run simultaneously, but we don't actually know that they are being executed, that the code is actually being computed by multiple execution units at once.

1:19:10  
It also is possible that you have concurrent computing without parallelism.

1:19:15  
So we have like a multitasking, but you have only one CPU.

1:19:19  
So it's like an appearance that they are running, that they are running in parallel, but it's only an appearance.

1:19:25  
Usually the concurrency is defined as a potential for parallelism and there are many things for consideration.

1:19:34  
Most important one is how this is happening.

1:19:38  
So in normal situations you've encountered in this class is that we consider all these processes, all these threats, all these units of scheduling that they are sharing the same address space.

1:19:52  
This is like one type of parallelism, but this is the only type we are considering in this class mostly.

1:19:59  
But sometimes you also may have a parallelism that utilize different memory spaces, like distributed computing, for instance.

1:20:07  
So now here we will focus on shared address space.

1:20:11  
That's more interesting because in shared address space, it might happen that 2 processes, 2 units of scheduling access the same data at the same time.

1:20:20  
And that creates potential problems that you need to address.

1:20:23  
That's why it's more important, more interesting.

1:20:26  
Usually we'll talk about threats because.

1:20:29  
Normally in regular operating system, in regular environment, this happens when you have one address space, one process, and within this process you execute multiple threads.

1:20:39  
So each thread can be scheduled on a different CPU core, but they are all accessing your one memory space, they are all accessing your data structures, all accessing the same heap.

1:20:52  
And the difficult part here is also scheduling, because scheduling is always difficult.

1:20:58  
We already covered this in separate class and operating system, so I'm not going into the details again.

1:21:04  
But remember, scheduling is always tricky because you don't know what the scheduler will do next.

1:21:09  
And currently the scheduling is preemptive.

1:21:12  
So at any given time it might happen that your process, your threat is being suspended in the middle of of of your code and another thread kicks in, another thread is scheduled on on the CPU.

1:21:23  
So the your threads may interleave their instructions unpredictably.

1:21:30  
The most pronounced problem that rises from the set up that you have multiple threads, multiple scheduling units running on the same data, running in the same memory space is called race condition by the word.

1:21:44  
By the way, the word race isn't from some racial issues.

1:21:48  
It's from the race as as a competition as if you're running or driving cars or something.

1:21:54  
So it's it's race like the the threads are competing who will first cross the finish line.

1:22:01  
OK, so just to clarify, this is not some ugly word.

1:22:05  
This is a regular word.

1:22:08  
And what happens here is that you have a piece of code, a data structure accessed by multiple threads, but these threads perform some operation which is divided into multiple steps.

1:22:21  
And it happens that if these multiple steps, if they are interleaved in the threat improperly, they may cause some damage, they may broke the data structure.

1:22:33  
By the way, just to clarify, we do have cache coherence.

1:22:37  
So in the hardware there are some mechanisms that prevent some very ugly situations from happening because the data are always accessed through caches.

1:22:47  
So they are always accessed with cache line granularity.

1:22:51  
Yeah, it helps, but it doesn't prevent some malicious scenario.

1:22:55  
So it doesn't prevent race condition.

1:22:57  
It's slightly helps, but it doesn't completely solve anything.

1:23:02  
And if we have a architectures like MIPS or like RISC 5 where there are load instructions and store instructions explicitly in the code, it makes this problem more pronounced because even a very simplest operation, let me give you example, let's just have an integer A and I perform a plus plus trivial operation, one line code of code.

1:23:26  
So in load store architecture, this actually translates into load, right?

1:23:34  
Let's say to register T1 to T0, it loads the variable a I'm writing this symbolically.

1:23:42  
Then there is some add operation like Eddy which performs something like this, right?

1:23:54  
Writing in MIPS.

1:23:56  
And then you need to store the value back.

1:23:58  
So there's something like 3 instructions.

1:24:09  
So it's not one atomic operation, it's divided into 3 separate steps, even something this simple.

1:24:17  
And the trouble is that after each instruction, the thread performing these instructions might get suspended, right?

1:24:26  
So imagine that piece of this code or piece of this code is executed by multiple by two threads.

1:24:32  
For instance, let me give you an example from the previous slide.

1:24:37  
So let's just say that two threads are performing this operation.

1:24:42  
We have a.

1:24:43  
Sorry, I haven't explained this.

1:24:44  
We have a linked list and the structure remembers pointer to the roots to the first element of the linked list.

1:24:50  
And this push front actually replaces the front node with another node.

1:24:55  
So it takes the root pointer, append it after the element being pushed, and then replace the pointer to the first element, right?

1:25:06  
Assuming that that the caller duly allocated this new element.

1:25:10  
So I'm not handling allocation here.

1:25:13  
So what happens if 2 threads simultaneously push two different new structures at the front?

1:25:20  
So this is the current state, this is the the initial state of this data structure and this is what what's happening.

1:25:25  
So we are having 2 threads, 1 is pushing A and another is pushing B.

1:25:30  
So one possible outcome would be that the first thread comes first.

1:25:35  
It takes the pointer to the current root, so the A next points to the current root and then replaces the root so it points to A.

1:25:45  
Afterwards comes thread 2 and does the same with B.

1:25:48  
So B takes pointer to A which is current root and then updates the root.

1:25:53  
OK, no problem.

1:25:55  
It might happen that thread 2 was slightly faster in the race, so it goes the other way around.

1:26:00  
So first the thread B, thread 2 will push the B and then thread one will push the A.

1:26:07  
Yeah.

1:26:08  
However, both outcomes are valid.

1:26:12  
We don't know which which threat comes first.

1:26:14  
There is no guaranteed order.

1:26:16  
So both outcomes are fine.

1:26:19  
We are counting on this.

1:26:20  
This is like both outcomes are desirable ones.

1:26:24  
Yeah.

1:26:25  
If we need some form of ordering, then we need to completely redesign our operations.

1:26:30  
We just push the items as they come, as they, they're spawned by the threats.

1:26:35  
So we, we don't really care which of these two outcomes will, will, will happen.

1:26:42  
We just happy that either of them didn't damage the data structure.

1:26:47  
Everything's fine.

1:26:48  
Both are still valid linked lists.

1:26:50  
You can traverse them and everything is OK.

1:26:53  
But it also might happen that these two threads are competing are running this code almost simultaneously, so they are interleaving these instructions 1 by 1.

1:27:03  
So what happens is that they both take the pointer to the root at the same time or virtual at the same time and then one of them will replace the root and then the second one will replace the root.

1:27:16  
So in the end, we will end up with a data structure which is not no longer valid.

1:27:20  
This is like a, this is damaged data structure.

1:27:22  
We don't want this.

1:27:23  
We will lose one of these two nodes because it wouldn't be pushed properly into the linked list.

1:27:29  
So this is something that shouldn't happen.

1:27:32  
OK.

1:27:33  
And yeah, it it might happen in any other way that there may be a different outcomes of this.

1:27:37  
So there are several other options.

1:27:39  
What what might happen?

1:27:41  
So if we have this piece of code that updates some data structure do something which is kind of a critical, we cannot allow that multiple threats perform this operation simultaneously in this manner because some damage might come out the the data structure may be broken or something.

1:27:58  
So this piece of code is called a critical section.

1:28:02  
We will, we can designate this code a critical section.

1:28:05  
And what happens is that we have tools, tools that usually ensure mutual exclusion, which means that within the critical section, there could be only one thread at a time.

1:28:18  
So there are some mechanisms.

1:28:20  
I will introduce them that will ensure mutual exclusion of threats within this designated part of call code call critical section.

1:28:29  
By the way, it might be 11 routine, the code might be scattered among multiple routines.

1:28:36  
It it doesn't matter.

1:28:37  
It doesn't have to be 1 continuous piece of code.

1:28:39  
The code can be scattered among multiple pieces of of your functional design or something, so it might be more complex.

1:28:47  
But for this let's just assume it's something like one function or one block of code.

1:28:54  
OK, what we need to do is perform synchronization.

1:28:58  
The synchronization is an unfortunate word because it's very overloaded.

1:29:03  
What I mean by synchronization, we usually mean the process synchronization or the data synchronization.

1:29:09  
But sometimes we need we talk about the abstraction, about the concept, sometimes we talk about the actual implementation.

1:29:16  
So yeah, the word synchronization is somewhat overloaded.

1:29:22  
The basic terms are process synchronization.

1:29:24  
And by process synchronization I mean that multiple units of scheduling, multiple threats, for instance, need to perform some operations, some handshakes, some protocol.

1:29:35  
So they agree on some order of things.

1:29:38  
So for instance they will perform a handshake and they will agree that I will execute this method 1st and you will execute it second and you will wait before I terminate the execution or something like that.

1:29:49  
So they will agree on how they will proceed with some instructions.

1:29:54  
Data synchronization is usually a higher level of abstraction which is designed to protect data structures, and they usually tend to either keep multiple copies of data coherent or somehow otherwise maintain data integrity.

1:30:09  
So like in this case, it will maintain that the data integrity will the the pointers of the linked list will remain in the right order.

1:30:17  
So they will allow only this version or this version.

1:30:20  
They won't allow any of these versions of of the data structure to happen.

1:30:28  
However, synchronization comes at cost.

1:30:31  
The obvious cost is it will have some overhead.

1:30:34  
Everything we do have some overhead.

1:30:36  
So yeah, synchronization isn't for free.

1:30:38  
We need to perform some additional operations.

1:30:40  
We need to invoke the hardware instruction in certain way.

1:30:45  
And everything has overhead.

1:30:47  
And also synchronization if used improperly can lead either to deadlocks, I'll explain them later, or to starvation, which means that the processes needs to wait for a long time or perform idle operations or something.

1:31:04  
So let me start with synchronization primitives, because they are the tools we need to use to perform synchronization.

1:31:11  
So we need these tools to do something in parallel code, so we make it safe for parallel execution.

1:31:19  
These primitives are usually implemented by operating system and they are provided to some libraries.

1:31:26  
So usually, for instance, if you use C#, you already know C#.

1:31:29  
Right now that there are multiple classes, multiple tools in C# which are technically translated into some system operating system calls, but you can use them for synchronization.

1:31:41  
They divide into two basic groups, active and passive.

1:31:44  
The active ones are actively waiting which means they are consuming CPU whilst waiting.

1:31:50  
They are eagerly trying to perform the operation just like knocking on the door waiting for the door to open.

1:31:58  
OK and the most important thing is they are actually executing instructions while waiting.

1:32:04  
They are usually using some testing conditions in in a loop.

1:32:09  
So they are performing a loop which tries to do some operation and and repeatedly fails until some door is open, some lock is released or something.

1:32:19  
This is usually used for short short pieces of code.

1:32:22  
If you have a short pieces of code where only few instructions happen, this is a good choice because yeah, if we perform a few additional instructions to knock on the door, it usually doesn't matter.

1:32:34  
Passive or blocking primitives usually suspend the unit of scheduling, suspend the threats that are running until the resource, the critical section or something becomes available.

1:32:48  
So this works also in hand to hand with the operating system scheduler.

1:32:53  
So this allows a threat to be blocked, to be put out of scheduling until the resource, the desired resource becomes available.

1:33:03  
This is useful for situations when the waiting might be long, when the resources are locked or allocated for longer periods of time.

1:33:11  
So this way the threats is put out of its misery temporarily.

1:33:17  
So it won't consume the resources, it won't consume the CPU instructions.

1:33:23  
For that we need some hardware support.

1:33:25  
And the basic hardware support is via atomic instructions.

1:33:28  
The basic atomic instruction, the one actually the only one we need in the end is called test and set or compare and swap.

1:33:36  
These are two names for the same thing.

1:33:40  
Task is usually used in the parallel programming.

1:33:44  
Test and set is usually used in operating systems, but it's the same instruction.

1:33:48  
It performs the same operation, OK.

1:33:50  
And what it does is it's given a variable.

1:33:54  
This is like the T stands for some type.

1:33:56  
It's like an integer, for instance, OK, it's a small enough variable.

1:34:01  
The T isn't something large, it's not a data structure.

1:34:04  
It's a integer, float, something very small.

1:34:07  
And you are giving it two additional parameters, the old value or the expected value.

1:34:14  
So we already assume that this variable has some value and we have some replacement for it.

1:34:22  
So we know what's currently, we should know what's currently in the variable, and we have some new value, we want to replace it.

1:34:32  
So what we are doing is that we test that this assumption still holds.

1:34:38  
It's still, it's true that the variable holds this old value, and if it does, we proceed.

1:34:46  
If it doesn't, we terminate with failure.

1:34:49  
So this atomic instruction can only proceed if the value within this variable still holds with our assumption here.

1:34:59  
Because if it doesn't, what, what, What does it mean if?

1:35:01  
If this is false, it means that someone else been there first, some other threat came there and perform an update before us, right?

1:35:12  
So if this happens, we need to stop our operation and try it again, rethink our position or something.

1:35:19  
So this is like a safety guard that prevents other threats to perform modifications before us.

1:35:28  
And if it happens, we will fail because someone else got into the line before us.

1:35:33  
And if it doesn't, if this holds, if it hasn't been modified, we can proceed.

1:35:39  
We write the new value into this variable, we just assign it, and then we report success.

1:35:46  
And the trick is that this code, which looks like a huge piece of code, there is a, if there is a return, it looks like a function call.

1:35:54  
This is actually just a seep SODA code for an instruction.

1:36:00  
This is performed by one single hardware instruction which makes sure that all the data operations, the whole thing happens atomically.

1:36:12  
So no one else, not even another core can access this variable when this instruction is being executed.

1:36:20  
And that's the whole point that this huge piece of code, well, huge from the perspective of individual instructions, these, these all operations are performed at once by the hardware.

1:36:31  
That's what, what's what to give us the edge, that's what to give us the advantage.

1:36:36  
And using this instruction we can easily implement active waiting waiting primitives because we can just use it in a loop to test some condition.

1:36:51  
For instance, there might be a variable that guards the critical section and when a thread wants to enter the critical section, the thread needs to write its ID into this variable.

1:37:04  
So initially the variable is 0 and I'm trying to write my ID in it, so I assume it's zero and my value is my ID.

1:37:12  
And when I succeed, I know I wrote my own ID which is unique into this variable, so no one else got there before me, right?

1:37:22  
And when I leave the critical section, I reset the value to 0.

1:37:25  
That's it.

1:37:26  
That's how I implement one type of active primitive.

1:37:30  
And using active primitives I can easily implement within the kernel of the operating system.

1:37:35  
I can easily implement more complex passive blocking primitives.

1:37:40  
OK, the easiest thing I can do I just described.

1:37:44  
It's called spinlock.

1:37:45  
It tries to write some some variable, usually using this SAS or TSL.

1:37:51  
So it can it can enter some small piece of code, perform some modifications and then retreat.

1:37:58  
The most fundamental passive primitive is called semaphore.

1:38:04  
Yeah, it's not widely used actually, but it was the first primitive ever designed.

1:38:10  
So we always starting with the semaphore.

1:38:11  
But I will simplify the semaphore on the next slide into a mutex.

1:38:15  
And the mutex is the one which is currently being used for the passive passive blocking.

1:38:22  
So semaphore works in a similar manner like a semaphore on railroad.

1:38:27  
You know, if you have a railroad, there is a train going next to the semaphore and when it does, the semaphore drops.

1:38:34  
So no train can enter the trail before the train reaches another marker or something like that.

1:38:44  
So here it's slightly more complex.

1:38:45  
The semaphore has a counter, and the counter is initially set into how many threads can simultaneously enter some critical section.

1:38:55  
So it's more versatile because it can it can allow multiple threads, but the number of threads is pretty fine, and we can limit to one if we want to.

1:39:04  
But this counter is usually initially set to number greater than 0 and there are two operations, down and up.

1:39:12  
Down is used when you're entering the critical section when you're trying to allocate the resource, and up is used to release the semaphore.

1:39:20  
This is the same way with the semaphore.

1:39:21  
When the train passes, the semaphore goes down and when it leaves the the the part of the rail which is protected, then the semaphore goes up and the down operation it always test the counter.

1:39:34  
The counter is critical and if the counter is greater than 0, it's decremented.

1:39:41  
So we are counting how many threads has passed this this down marker, this this semaphore and if there are no more slots available, if the if the counter reaches 0, then the the US denotes unit of scheduling which you can imagine as a threat.

1:39:58  
So.

1:39:59  
So you can push the current thread into a queue which is associated with the with the semaphore, and then the thread is blocked, it's suspended by the operating system.

1:40:11  
And vice versa, when we performing up operation, we need to check whether the counter is still 0.

1:40:19  
If the counter is 0 and they are waiting some other threats in the queue, then we just stop one of the threads from the queue and start and allow it in immediately.

1:40:31  
So we unblock it and the thread is allowed into the critical section.

1:40:36  
By the way, if you call down, if your thread calls down, it will either immediately enter the section or it's blocked until it can enter.

1:40:45  
So once you emerge from this call, you know you can enter the section, you know you can perform whatever you need to do.

1:40:53  
So this happens if your thread gets blocked, then other thread will unblock it in the app operation and otherwise if there is, if the counter is either above 0, so we we have nothing, nothing in the queue or the queue is empty.

1:41:08  
So it's the edge case where the counter is still 0 but no more threads are waiting in the queue.

1:41:13  
Then we simply increment the counter.

1:41:15  
There is no one else to wake up, no one else to unblock.

1:41:20  
Is this clear or was I too fast?

1:41:23  
OK, and obviously to make this work, these operations need to be need to be made atomic.

1:41:31  
So these but not by hardware, but by operating system, by additional locking, by additional atomic instructions.

1:41:38  
But both operations are considered to be atomic from the perspective of threat execution.

1:41:45  
That's why this this whole thing works.

1:41:48  
We are now not usually using semaphores because they are kind of obsolete, but we are frequently using mutexes.

1:41:55  
Mutex is a semaphore which has its counter sets to 1.

1:41:59  
Easy, because usually you want to.

1:42:01  
You want to allow only a single thread into the critical section.

1:42:05  
You want allow only a single thread to do something.

1:42:08  
So if we set the counter to one, we will get something which is more often referred to as a mutex.

1:42:14  
And again, we usually refer to the up and down operations as a lock and unlock.

1:42:19  
It's slightly more readable than down and up because yeah, not everybody knows nowadays what semaphores are for, because we don't have the regular semaphores on the railroads anymore.

1:42:31  
We have lights.

1:42:32  
And so lock and unlock is probably more to the point, a few other examples.

1:42:39  
A barrier is a primitive that allows a rendezvous of multiple threads so that the barrier is executed by all the scheduling units, all the threads that wants to meet at a certain point.

1:42:52  
For instance, let's say that the threads are cooperating on some initial part of your work there.

1:42:58  
They're initializing some data structures at the beginning, and you need to make sure that all the threads have finished before you proceed with the computation.

1:43:07  
So you place a barrier between the initialization part and the computation part of your code so that every thread that was initializing something will meet on the barrier before all the threads are allowed to proceed after the barrier.

1:43:22  
Yeah, in various languages you also have some specific constructs.

1:43:26  
For instance, in C and Java you have synchronized and locked.

1:43:29  
Which are the most simplest ways how to introduce critical sections or locking to your code.

1:43:36  
Basically in the locked section of code there is some mutex in behind, hidden mutex, hidden lock somewhere in behind.

1:43:44  
So this is just the way how you can denote critical sections and the the C# environment.

1:43:50  
The C# runtime provides the locking mechanisms provides the synchronization primitives for you for free.

1:43:57  
So it's it's much easier in higher level languages to perform this type of synchronization.

1:44:01  
On the other hand, it's not a good idea to write a lock or synchronized everywhere, because if you do, you might end up with a code which will be very slow.

1:44:11  
Or even worse, you can you can reach into a deadlock.

1:44:15  
Yeah, Speaking of deadlocks, deadlock is a very specific situation you can create if you lock, use synchronization primitive inappropriately.

1:44:27  
These are like real world examples of a of a deadlocks.

1:44:30  
Yeah, I, I always wanted to try this one somewhere, but I've never got this this question anywhere, unfortunately.

1:44:37  
But maybe you do so.

1:44:38  
So if you do some hiring interviews, you can try that.

1:44:42  
This is a very, very illustrative way how to depict a deadlock.

1:44:48  
If you, if you watch this, if you decode this image properly, none of these vehicles can move.

1:44:55  
Yeah, at least none of them can move in the direction they they would like to.

1:45:00  
It's possible for some cars over here to turn right, right, and just avoid this, avoid this mess.

1:45:07  
And actually until some of these cars do so, until some of these cars will actually turn somewhere and and use a different Rd., they won't, nobody will move, right.

1:45:21  
So how to define deadlock more formally?

1:45:23  
There is actually formal specification for that.

1:45:26  
And informally deadlock is a state when the group of units of scheduling, group of threats and group of some resources by resources.

1:45:35  
You can, you can imagine critical section protecting some data or mutexes or something are in a state where everybody is waiting for an action which needs to be performed by someone else.

1:45:48  
It's like an informal definition.

1:45:49  
Formally there are 4 rules, they are called Kauffman conditions.

1:45:53  
And if all these four conditions are met, then you are in deadlock.

1:45:58  
So first of all, there is a mutual exclusion of the resources.

1:46:02  
So we really have some locking mechanisms, some resources that allow that only one thread, only one unit of scheduling is accessing that particular resource at a time.

1:46:13  
2nd is all the threats, all the units of scheduling are performing hold and wait.

1:46:18  
So they are accumulating resources.

1:46:20  
When they get a resource, they are holding that resource and they want more.

1:46:24  
So they are trying to accumulate several of these resources, at least two.

1:46:29  
And there is no preemption, which means you cannot remove a resource from a threat from a unit of scheduling which was previously allocated to it.

1:46:39  
So once a threat locks a mutex for instance, you cannot break this lock.

1:46:46  
You cannot unlock it without causing some external damage, without violating semantics of the threat, for instance.

1:46:54  
And finally and most obviously, there is a circle in the weighting graph.

1:47:00  
So if you depict a graph where the edges are between units of scheduling and resources and thread between thread and resource means that this thread holds this resource, sorry, this thread wants this resource.

1:47:15  
And between resource and thread is that this resource belongs to this thread, is held by this thread.

1:47:22  
So if you create this graph and you detect a cycle, a loop in this graph, then you're in deadlock.

1:47:29  
OK, there are several ways how to prevent deadlock.

1:47:32  
But before that, before I get to them, let me give you a very similar example, practical example of what can happen in a real code.

1:47:40  
So let's say I have two mutexes, yeah, in C# for instance, and I think mutex, I'm not entirely certain.

1:47:49  
These methods might might be named slightly differently, but basically the mutex have two operations, lock and unlock.

1:47:56  
So thread one performs lock and lock on both mutexes and thread 2 performs lock and lock on these two mutexes.

1:48:04  
But unfortunately it takes mutex 2 first and mutex one second.

1:48:09  
So what might happen is that thread one performs this lock and then thread one is scheduled away, it's paused.

1:48:18  
Thread 2 comes in and performs lock on the second mutex and then it tries to look on the 1st mutex but it can't because it's already held by thread one.

1:48:30  
And then thread 1 resumes and tries to look at the second mutex but it can't because it's held by thread 2.

1:48:37  
So they both waiting until the 2nd until the resource they want is released.

1:48:41  
Is it possible to create a deadlock where each thread is only controlling 1 mutex at the time?

1:48:53  
Well yeah, if you have a regular normal mutexes, you can try to lock the same mutex twice.

1:49:01  
So yeah, you can have one mutex and one thread is trying to relock the mutex.

1:49:07  
So technically, yeah, you can create it with one thread and one mutex if you don't have recursive locks.

1:49:15  
But that's just a technicality.

1:49:18  
So as you see, this is what can happen and actually the solution to this problem is rather simple.

1:49:25  
Actually in this case it's simple.

1:49:29  
It may not be always possible in your code to do so, but if you have code like this, it's simple.

1:49:35  
We just need to flip these two lines.

1:49:38  
If you can impose an ordering on all the locks you have in your application.

1:49:43  
So you have like all the mutexes have the unique numbers and you say hey, if you need to lock multiple mutexes, you always need to do that in that order.

1:49:54  
So you need to always do that incrementally.

1:49:57  
So if you want to lock M1 and M2, you always need to lock M1 first and then lock M2.

1:50:05  
If you say that, you're fine because if there is imposed ordering, there will be no deadlock because 1 lock will be covering all the locks after it in it's in this ordering.

1:50:20  
If you didn't catch that, no don't no worries.

1:50:23  
It's like a technical thing that you can came up to that later when we talk about more detailed locking in programming parallel programming courses.

1:50:37  
To conclude this part of lecture, there are few synchronisation problems.

1:50:42  
It's like a allegories and it's a good for you to know them because they usually depict some situations in the real.

1:50:51  
Like actually the allegories are like from a real world and they depict some situations in the programming world.

1:50:57  
But if you know them, you can think about programming and locking, parallel programming and locking in a certain way, which is helpful.

1:51:03  
So that's why we are introducing these these allegories.

1:51:10  
Let's start with the simplest problem and probably the most profound problem in parallel computing, and that's producer, consumer.

1:51:19  
So imagine a situation that you have two entities.

1:51:23  
One of these entities is producing something and another entity is consuming it.

1:51:28  
So yeah, regularly in the real world, this would be like a factory and this would be like a customer, right?

1:51:33  
And in between them there is a store, there is a warehouse or something.

1:51:37  
Because we are usually not going into the factories to get our food, our cars, our clothes, whatever.

1:51:47  
We usually go to stores or to warehouses or somewhere which is closer to us.

1:51:52  
So imagine that if the factory is producing something to buffer the outcome, because usually the factory has a steady output and the customers act more randomly.

1:52:05  
So you need to put everything in some storage place so the factory can produce its load per day and also the customers can can access this this goods on demand.

1:52:19  
The trouble is what if the factory is producing too much, then the warehouse gets full and in that case the factory needs to stop.

1:52:28  
And vice versa, if the customer comes to the warehouse and it's currently empty because it's still waiting for the shipment from the factory, the customer will wait.

1:52:38  
And this allegory just explains a buffer in between two processes.

1:52:44  
So what happens if you have a buffer and two processes, 1 is outputting something and another is consuming something?

1:52:51  
This usually happens in a pipeline.

1:52:53  
If you have two stages, 2 follow up stages in a pipeline, then sorry I haven't explained pipeline in other contexts done in ACPU, but you can imagine.

1:53:04  
So this is like a buffer which tries to mitigate the problem that both processes are running in different speeds.

1:53:13  
Yeah, and sometimes it doesn't work correctly because if one of these two processes is much faster than the other, then it will always wait for the other.

1:53:22  
OK.

1:53:23  
Another problem is dining philosophers.

1:53:25  
Imagine a situation that you have multiple philosophers around the circular table and each philosopher has a bowl of a Chinese food in front of them and there are two chopsticks across each bowl.

1:53:38  
So basically you have a bowl, chopstick, bowl, chopstick, etcetera.

1:53:41  
And as you probably know, a philosopher can do only two two things, think and eat.

1:53:47  
So they are sitting there and thinking and once they get hungry, they want to start eating.

1:53:53  
And what they need to do is they need to pick two chopsticks because I'm not sure if anybody here trying to eat Chinese food, but it's almost impossible to eat Chinese food with one chopstick, right?

1:54:05  
So you need 2.

1:54:06  
So you can properly pick all the good meat, good stuff from the meal and imagine situation that all the philosophers reached the same conclusion and meaning that they all get hungry at the same time.

1:54:23  
So they all perform the following algorithm.

1:54:27  
They will grab the chopstick at their right hand and then they try to grab the chopstick at their left hand, right.

1:54:36  
So if they manage to do this all at once, they will create a deadlock because everybody is holding the right chopstick and the left chip left chipstick isn't there anymore.

1:54:50  
So they're waiting for the left chipstick to be placed there.

1:54:53  
But because they're all holding the right one, it's a deadlock.

1:54:57  
So this is just another visualization of the loop, which is inherent in the Kauffman conditions.

1:55:04  
So in this case, the loop might be much wider.

1:55:08  
And yeah, you might say, hey, this will never happen, right?

1:55:11  
Because it's so improbable that multiple philosophers will reach to the same at the same point to the conclusion that they all want to start eating at the same time.

1:55:23  
Well, it will happen sooner or later, because yeah, it might not happen in real life because the people are too unpredictable.

1:55:31  
But machines are much more organized.

1:55:33  
So if these are not actual philosophers but they are processes, it will happen sooner or later.

1:55:38  
Yeah, I can give you many examples from real world.

1:55:41  
But programmer said, hey, this super unprobabilistic conditions couldn't meet all together.

1:55:48  
And if somebody says that, it usually happens twice a day, so you can rely on that.

1:55:54  
So OK, let's augment this algorithm.

1:55:56  
Let's say, OK, it's a bad idea to hold your chopstick forever if the second chopstick isn't there.

1:56:02  
So let's augment this algorithm.

1:56:04  
Let's say they take the right chopstick, and if the left chopstick isn't there, they release the right chopstick and try again.

1:56:14  
Seems reasonable, right?

1:56:16  
Well, it's, it is reasonable because again, in normal world it's it's unlikely that they will be all performing this task at the same pace.

1:56:25  
So one of them will finally grab the right chopstick and the left chopstick at the same time, which will break the circle and allow the others to eat sequentially or partially sequentially, but at least it will break the deadlock sequence.

1:56:40  
But if they are very organized and they are all proceeding at the same pace, they are all of the same age, for instance, what will what will happen?

1:56:48  
They will take the right chopstick and the left isn't there.

1:56:52  
So they will release the right chopstick and then, then try again.

1:56:55  
They will take the right chopstick and the left isn't there.

1:56:57  
So they will repeat this routine all over again and again and again, which is even worse than deadlock because they not only cannot, they can't eat, but they also consuming a resources that they're, they're spending their energy moving there right up and down.

1:57:14  
So it's even worse than deadlock.

1:57:17  
So these are things that we need to consider when, when dealing with operations like this.

1:57:22  
And by the way, one solution could be put a salt glass in in the middle and say hey, everybody who wants to eat first needs to put salt in the into their meal and then then they take the chopsticks and then they release the salt.

1:57:39  
OK, so if there is only 1 salt in the middle, only one will win the salt and the the person with the salt will start eating and then it will release the salt.

1:57:51  
So someone else can try that.

1:57:54  
So place a small lock aside before the grand locks can be taken.

1:58:03  
Another problem is a reader and writer.

1:58:06  
When considering locking, there are usually more complex scenarios in the real world.

1:58:11  
Let's assume that we have a data structure.

1:58:14  
Yeah, this is less of an allegory and more of a programming problem.

1:58:20  
So let's say we have a data structure.

1:58:22  
And usually what happens is data structures are more often read than written to.

1:58:27  
If you have a database, you much more often perform reading operations on the database than updates of the database.

1:58:35  
So it's a good idea to separate these two types of operations because readers can work simultaneously.

1:58:42  
And that's a benefit.

1:58:44  
You don't have to lock the readers out each other out.

1:58:47  
You can allow all the read simultaneously in.

1:58:51  
That's a good idea because you have more parallelism, you have more performance.

1:58:55  
On the other hand, if you have a writer, then the writer must be mutually exclusive to all other writers and also to all other readers.

1:59:03  
So a writer must wait for all other readers to finish before a writer can go in and perform modifications of the data structure.

1:59:10  
And since this is a very profound, a very common problem, we have a specialized primitive called Reader writer Lock, which is designed specifically for this operation.

1:59:23  
It works similarly to mutex, but when you're locking this structure, you are specifying whether you would like to perform a reader's lock or a writer's lock.

1:59:34  
So whether we are trying to read or write the data structure, the tricky part is how to make this fair and efficient.

1:59:41  
Because imagine that if you have multiple readers and then the writer came in and writer asks for a lock.

1:59:50  
So the writer needs to wait because it's a lock, so it cannot acquire the lock.

1:59:55  
So it waits in a queue until all the readers are done.

1:59:59  
That's how it must be.

2:00:01  
But imagine that while the writer is waiting, another reader came in.

2:00:07  
Should this reader be allowed to read or not?

2:00:12  
And the answer is obviously difficult because if you allow it to read the data structure, you may increase the performance because it it might happen that this writer will will read it, read the data very quickly along with all the other finishing readers, and you just enabled another reader to just jump in and and do whatever it needs to do.

2:00:35  
Then again, if you allow every incoming reader in, then the writer will starve because it's possible that it will never be allowed to perform the update.

2:00:45  
So it's a tricky question.

2:00:50  
And finally, the last problem is sleeping Barber.

2:00:53  
Yeah, you may notice that I need a haircut, but there's just a coincidence actually.

2:00:58  
This is about Barber shop.

2:00:59  
And at least in in this terminology, I think the Barber shop is about shaving the beards, right.

2:01:05  
So in the, in this scenario, there is a Barber shop and the Barber has one chair for a customer which is currently being served and multiple chairs, chairs for customers that are waiting in queue.

2:01:18  
And what happens is when the the new customer comes in, then the Barber is sleeping usually because he has nothing to do.

2:01:26  
So the customer wakes the Barber and the Barber immediately starts shaving the customer.

2:01:32  
And if customer comes and Barber is already at work, then the customer takes one of these chairs.

2:01:38  
And if customer comes in and all the chairs are full, then the customer needs to go away.

2:01:44  
Yeah, again, this is a very simple scenario, but it completely it accurately describes what happens at a web server.

2:01:50  
For instance, if you have a web server and a new request comes for a for a web page, then it wakes the waiting thread that's locked on the on the socket, and the thread starts to serving the answer to the to the to the incoming network request.

2:02:10  
If you have a multiple request, multiple requests, they need to be queued somewhere, so they wait until the 1st request is served before they take their turn.

2:02:21  
If you have too many requests, you need to have a fixed number of chairs, otherwise your server will run out of memory because it will be holding too many requests in a queue.

2:02:33  
So at some point you need to start rejecting the requests right away.

2:02:38  
And that's what usually happens if you have overloaded web server, you got rejected, your request got rejected right away.

2:02:44  
They aren't waiting in a queue.

2:02:47  
So these problems were to illustrate some of the common scenarios, some of the common situations that happens in a parallel or concurrent environment.

2:02:57  
Sorry about rushing through this lecture.

2:02:59  
I was trying to not not prolong this, this session too much.

2:03:05  
So are there any questions regarding parallel programming?

2:03:11  
No, not at this point.

2:03:13  
OK, let's move on to the final part.

2:03:17  
Few final words and few examples about testing questions.

2:03:24  
So first of all, just a quick reminder what was covered in this course.

2:03:31  
First of all, C++ will not be explicitly examined during the test.

2:03:37  
But still you need to know C++, C++ at some level at least to understand some questions or to express some pieces of code.

2:03:44  
So you won't be coding during the test, but you need to at least read the C++ during the test.

2:03:52  
Then we got several categories.

2:03:53  
Yeah, in CPUs we cover things like instruction encoding and execution.

2:03:58  
So you need to know about that.

2:04:00  
Again, you don't need to know from the top of your head the whole instruction set of a MIPS.

2:04:05  
That would be very difficult.

2:04:08  
But if I show you some MIPS instructions and say, hey, this is the instruction that performs branch if equal, which means it it jumps.

2:04:15  
If these two registers are equal, then you need to be able to use this instruction or you need to be able to assemble a few instructions or read few instructions when you're explained what these acronyms are.

2:04:28  
And you need to also know to how these instructions are mapped to the higher language.

2:04:33  
So you need to be able to translate from C++ to assembly and vice versa at some fundamental level.

2:04:40  
And of course, things that are related to that, you need to know how registers work, how registers are used with the instructions obviously, and some basics of the hardware architecture so that there is some pipelining and stuff like that in the memory.

2:04:56  
We covered some fundamentals regarding addressing in DNS.

2:04:59  
Yeah, this is like a rehearsal for you, because that was already covered in the previous course.

2:05:03  
How the data structures are represented in memory, how the memory allocation works, and some technical details like how caches works or how NUMA works.

2:05:14  
In the programming languages, we mainly focus on compilation and runtime.

2:05:17  
So you need to know how compilation works, linking, dynamic linking, how the program is loaded into memory when it's executed, etc.

2:05:24  
How the memory is organized with an executed program so that there's some code segment, some some data segment, heap stack, stuff like this.

2:05:32  
How things are organized on the stack, which goes hand to hand with calling conventions.

2:05:37  
Because when you're calling a function, how the data, how the arguments are transferred to the call callee from the caller and how the result is from the callee transferred back to the caller, what's on the stack, what's in the registers, things like that.

2:05:50  
And also we covered some basic things about portability and, and, and virtualization and, and garbage collector.

2:05:57  
Yeah, the the largest part was covered by operating system.

2:06:03  
So yeah, we got several points, devices, processes, file system and virtual memory.

2:06:08  
It's like 4 topics, 4 main topics.

2:06:10  
And basically you need to understand the basic principles in all.

2:06:13  
So for instance, you need to know how interrupts work, to report that something happens on a device to the CPU.

2:06:20  
That's a very important thing.

2:06:22  
You need to know how scheduling works.

2:06:23  
So you need to know about processes, threats, you need to know some algorithms, you need to know how file systems work.

2:06:30  
So we got 2 examples.

2:06:32  
For instance, if I show you FAT file system, you should know how to use it, how to read it.

2:06:38  
And the same goes for virtual memory.

2:06:39  
There are some algorithms within virtual memory, how the address is translated from virtual address space to physical address space, how new page is allocated, how new frame is allocated, when new page is required to be accommodated, et cetera.

2:06:53  
And in the final parallel processing part, which we covered today, yeah, the fundamentals like what's race condition and what's deadlock, that's obvious.

2:07:02  
You need to know what that is, and you need to know some synchronization primitives, how they work.

2:07:06  
If I ask you how the task instruction works inside, you need to be able to write it.

2:07:13  
If I ask you how the semaphore works inside, you need to be able to write the down and up functions, et cetera.

2:07:22  
So I asked you about revising the previous lectures from the throughout the semester.

2:07:31  
If you have any specific questions regarding anything we covered this semester, fire away.

2:07:37  
I think I'm screwed.

2:07:43  
No, no you aren't.

2:07:44  
Well, if you, if you're screwed, that better be screwed here in the first year then then wait to the other years because things are getting tougher.

2:07:51  
So I'm sorry to bring that to you.

2:07:54  
But we are still at the very basic level.

2:07:57  
The follow up courses will go much deeper.

2:07:59  
So don't look at it this way.

2:08:02  
You just need to pull through.

2:08:06  
You need to muzzle it.

2:08:10  
No questions.

2:08:11  
OK, I will ask again at the end.

2:08:14  
Let me give you a few piece of information about the exam.

2:08:18  
So first of all, the exam, the exam dates are already in CIS.

2:08:22  
So you can you can enroll for the exams.

2:08:26  
We expect there will be no other dates unless we detect some major problems like many of the dates were colliding with other other exams or something like that.

2:08:36  
So unless we detect some major problems, which is like a systematic not minor errors, then we are not planning to plan any other dates.

2:08:47  
We are not sure whether we will be able to write any.

2:08:52  
Sorry I I was talking about for I've been talking for too long time so I'm not pronouncing correctly anymore.

2:09:02  
Sorry, my speech processor got damaged.

2:09:05  
Sorry, the we are not sure whether we will be able to provide any terms in September.

2:09:13  
So please do not rely on September terms.

2:09:16  
We will try to muster maybe one term in September for real emergencies for those who really weren't able to attend during the the summer part.

2:09:26  
But try to squeeze in, there are like I think 19 dates so everybody can pick their own favorite date.

2:09:33  
And I hope that you will find some dates that are not colliding with other exams.

2:09:38  
Then again, each date has only 20 places, 20, The capacity is limited.

2:09:43  
So yeah, make sure you enroll sooner than later.

2:09:48  
Again, I told this at the beginning of the course.

2:09:52  
The credit is not required for the test.

2:09:54  
The credit is completed separate from the exam.

2:09:56  
So you can visit the exam independently on the credit.

2:09:59  
If you fail the credit, no problem.

2:10:01  
We will.

2:10:01  
We will acknowledge.

2:10:02  
We will, right?

2:10:05  
We will copy your exam results from this year to the next year.

2:10:09  
We don't want you to attend these things twice.

2:10:12  
If you just pass exam this year and you don't finish credit this year, then you can retake only the credit next year, no problem.

2:10:21  
This exam is focusing on understanding, so if you just memorize the slides or memorize the lectures, it probably won't help you.

2:10:32  
So it's focusing on whether you understand how the algorithms, the data structures, anything we describe to you works.

2:10:40  
So questions are usually designed in a way that you need to apply something.

2:10:46  
Yeah, this is the how the memory looks like, trying to apply this algorithm of memory allocation of address, translation, whatever, and tell us the result, compute the result or something.

2:10:59  
So yeah, I know this looks like additional extra skill, but it isn't an extra skill.

2:11:05  
It's just an direct application of what you should know.

2:11:09  
So maybe together with some high school math, Yeah, you need to count to maybe not to tend to more, but yeah, you need to be able to add to 3 digit numbers together, stuff like that.

2:11:23  
Yeah, that's like a normal skill to do.

2:11:26  
You need to be able to read English.

2:11:27  
Yeah, that's another skill that that should be obvious.

2:11:31  
But beyond that, you the applications of the knowledge is direct.

2:11:36  
So you just need to know the stuff from the from the lecture and be able to apply to a very simple example, simple problem.

2:11:48  
The structure is as follows.

2:11:50  
You can, you have 10 questions, 20 points.

2:11:53  
So obviously each question is 2 points, right?

2:11:56  
No, actually there are one point and three-point questions.

2:11:59  
But for that later it's a test.

2:12:02  
The most important thing, it's a test, which means we are counting only the answers, not any intermediate thoughts, processes, whatever.

2:12:13  
We just collecting the answers and most of the answers are written as a you just check some options.

2:12:20  
You just select an option, you just put some options in order or you write a number or a few numbers, that's it.

2:12:26  
And we taking only the answers.

2:12:29  
I know it's tough.

2:12:30  
It's it's more difficult than if we are just if we would interview if you if you perform oral exams.

2:12:38  
But then again, to be completely honest, if you write a code and you made a small error like you just add +1 incidentally at the end, it made happen that the two trains collide or the plane falls or something.

2:12:55  
And nobody will care that that you just make a tiny mistake.

2:12:58  
But the whole procedure of your deduction, reasoning or the rest of the code worker was correct.

2:13:03  
No, the one mistake matters.

2:13:05  
So that's why we decided to take this test this way.

2:13:12  
And you will be given enough time to double check, triple check how many checks you need, your answers.

2:13:19  
So take your time.

2:13:20  
Normally the test is designed for 65 minutes.

2:13:25  
That's the number we just calculated using some common metrics for the amount of text and amount of questions.

2:13:32  
But it's a soft deadline.

2:13:33  
So it's like, yeah, after after hour and 5 minutes or so, we will start start gently reminding you that it's time to wrap things up and finish.

2:13:44  
But if you still need 10 more minutes to double check your answers, you will get 10 more minutes no problem.

2:13:50  
Yeah, you will get how much time you need within a reason.

2:13:54  
Yeah, we can stay in the labs till the next day.

2:13:58  
The the labs are allocated for some amount of time.

2:14:01  
But it really does.

2:14:02  
We don't really care if you take few more minutes or no, it's up to you.

2:14:06  
So take your time to do the stuff properly, take your time to double check, triple check your answers because that's what you should do in a real life.

2:14:14  
So we want to you to do this on the test as well.

2:14:18  
This is the grading scale.

2:14:20  
So it's a slightly, it might be strict or it might look strict, but then again, it's a test.

2:14:25  
So many of the questions are kind of easy.

2:14:29  
If you see the answers and you know the stuff from the lectures, you should be able to pick them easily enough.

2:14:36  
So yeah, this looks harsh, but our past experience indicate that we set this scale properly.

2:14:44  
Actually, many people that did well during or had regular questions during the semester ended up with mark one or was mark 2.

2:14:54  
So yeah.

2:14:55  
And people I have never seen throughout the semester usually end up with mark 4, which is a good indicator for for me that the test scale is set properly.

2:15:05  
OK question times since it's a test that some part of the test is covered by simple single or multi choice answers.

2:15:16  
So you select one or multiple options from the answer list and again, you need to make this correctly.

2:15:25  
So yeah, in the single best answer it's obvious you need to pick the right answer.

2:15:29  
In the multi choice test, you need to make all the check boxes that should be checked, checked and that shouldn't be checked are unchecked.

2:15:38  
So you need to create the proper combination of all checked and unchecked answers options to make the answer correct.

2:15:46  
And by the way, this the single option tests are the single option questions are called best answer, which means they might be multiple options which are seemingly similar.

2:16:00  
And in these cases, you need to pick the best one, the one that answers the question the most accurately, right?

2:16:08  
So sometimes it's slightly more difficult than a regular single answer test because if you have like 3 options and two are completely idiotic, then you can kind of deduce that the third option is the best one.

2:16:23  
But sometimes you can have like two or three options which are seemingly correct, so you need to read them carefully so you pick the right one.

2:16:31  
Sometime if a part of the answer, part of the option isn't completely correct, then obviously it's not the right answer.

2:16:38  
It's not the best answer.

2:16:41  
Then there is a multi choice with order.

2:16:44  
So you need to pick the right options and place them in the right order.

2:16:48  
This usually is used for the coding questions if you have like 10 lines of code.

2:16:53  
So you need to pick the right lines and put them in the correct order so the program makes sense.

2:16:59  
That's it.

2:16:59  
That's how we test coding.

2:17:02  
But by the way, by coding I mean you don't need to design A specific algorithm or something.

2:17:08  
You just usually need to assemble something that was previously on slides or you need, you should know from the slides.

2:17:16  
And finally, there are some tests questions where you need to answer with a number or multiple numbers.

2:17:22  
So there are text boxes and you fill one or multiple numbers in it.

2:17:26  
Like compute how many page faults will happen.

2:17:29  
In this scenario.

2:17:33  
The questions are divided into two parts.

2:17:35  
One are quiz questions.

2:17:37  
Quiz questions aim more directly to the point regarding the knowledge.

2:17:43  
So basically there is a question, there are some answers and you need to answer the question by remembering some stuff from the slide.

2:17:52  
So this is less application, more about memorization and they are only one point each, so they are less valuable.

2:18:02  
And basically this is an example of such a question.

2:18:05  
The question will read the instruction set ISA defines, right?

2:18:10  
And it's a single best answer.

2:18:13  
So if you really know what is ISA about.

2:18:16  
And yeah, I assume that some people don't know that at this point because you didn't bother to read through the previous slides and to watch the previous lectures, right?

2:18:26  
Yeah, I understand that.

2:18:28  
But you should do that before the test.

2:18:30  
And before the test, when you will be properly refreshed and your memory will be properly joked, you will arise to what conclusion?

2:18:49  
Specific parsing of individual phrases were kind of often not no, there there is no trickery like don't look for tricks like missing dashes or missing commas or something like that.

2:19:01  
No, the the best answer should stand up to you if you know the stuff.

2:19:07  
So it's it's not like we are missing an apostrophe somewhere and that's that's what causes the the problem.

2:19:13  
Yeah, if, if, if there is a source code, you need to read it carefully, because sometimes a a; In the source code can make a difference, right?

2:19:23  
But in the text, in the regular text, there will be no trick rates.

2:19:27  
It's not about the syntax, it's about the semantics.

2:19:30  
If you find out a typo or something in the text, it's just a typo.

2:19:34  
It's not like a incidental error.

2:19:40  
So are there any?

2:19:41  
Are there any guesses what should what the correct answer should be?

2:19:56  
No one.

2:19:57  
The problem is, I don't remember what exactly ALU stands for arithmetical logical units.

2:20:06  
So this is the electrical stuff that actually does something with the numbers.

2:20:11  
Compute addition of two numbers or something like that does that.

2:20:16  
This is hardware, yeah.

2:20:22  
And.

2:20:25  
The instruction set architecture, I don't think that defines hard.

2:20:31  
That's right.

2:20:32  
So it's not any of the one.

2:20:34  
That's right.

2:20:35  
So, so you can, you can rule out anything with Alus.

2:20:39  
You can rule out caches.

2:20:44  
Yeah, that's it.

2:20:46  
I think.

2:20:47  
I'm not sure if I'm reading everything right now quickly.

2:20:51  
So yeah, if you if you detect any hardware strictly hardware related to a word, then you can rule that out because hardware architecture isn't part of ISA.

2:21:02  
That's one thing.

2:21:03  
You can you can do another thing.

2:21:06  
So now I have to figure out if the instruction semantics is a specific phrase or if it's like a general phrase that needs to proxy.

2:21:21  
Well, it technically it can be both because sometimes we need to use exact terms in in the questions.

2:21:27  
So yeah.

2:21:28  
But here it's it's a normal term.

2:21:30  
So it's like a semantics of the instructions what the instructions do.

2:21:50  
It definitely includes construction semantics and registrants.

2:21:55  
Yeah, it must.

2:21:56  
That's right.

2:21:56  
That still needs us.

2:21:58  
Well, you mean this one?

2:22:06  
I think we fall underneath that are possessed.

2:22:08  
But I know I feel confident about memory management.

2:22:19  
I don't know how people like the IO, I understand, but if you don't have defined how the CPU handles the memory, how it accesses the memory and how it's performed, the virtual memory translation, and if you don't define how the CPU should handle the IO, at least on the abstraction level, then the CPU wouldn't be able to do anything.

2:22:46  
I feel like I feel like you're describing.

2:22:50  
Maybe I'm thinking of some other higher level content, but I feel like there's something that like we've said, there's something that technically the instructions that architecture doesn't define and believes, like technically there's some additional term and all would be like, yeah, don't, don't try to look at it too harshly.

2:23:09  
It's like we are not trying to trick you.

2:23:11  
So your initial, your soaked up information, I understand.

2:23:17  
So your initial assessment is was correct.

2:23:20  
You try.

2:23:21  
You need to rule out everything that's related to hardware specific design.

2:23:26  
OK, so you rule out the Allus, you rule out the caches, you rule out the micro micro instructions and you end up with this, this and this.

2:23:41  
So we have three options left.

2:23:45  
And if you check them carefully, you will find out that none of the things that are left in these three options are related to hardware.

2:23:52  
So they all need to be in the ISA.

2:23:57  
So the most precise 1 is in this case the longest one because it defines everything.

2:24:03  
This is, this is too short.

2:24:04  
This is, this is missing memory management, for instance.

2:24:08  
This is missing something else.

2:24:10  
This is missing instruction semantics, for instance.

2:24:13  
Yeah, right.

2:24:14  
So in this case, the most complete answer is the the right answer.

2:24:18  
However, as you can see, the longest answer isn't always the right answer because this one contains a few things that shouldn't be there.

2:24:26  
So you need to rule out the wrong answers.

2:24:30  
And then probably the most precise answer, in this case the longest answer, because it enumerates everything important, is the right answer.

2:24:40  
OK, you will be allowed to ask questions.

2:24:46  
Yeah, you will be allowed to ask questions, but you might not receive the answers.

2:24:50  
So not receiving that.

2:24:52  
If you if you ask something that's that you should know, then you won't get the answer.

2:24:56  
But yeah, you can of course you can ask questions.

2:24:58  
I just, I I know that I have the lean really heavy on that on the quick phrase, this kind of bluntly, the kind of exam that has the potential to give me a panic attack, which I recognize this is being sorry about that.

2:25:13  
I'm the subject matter.

2:25:17  
This is not Yeah, I I understand that I had a panic attacks from from mathematical analysis because that was something I cannot learn.

2:25:29  
But yeah, sorry.

2:25:32  
So let's move on The another 5 questions would be for three points.

2:25:40  
So they together they will form the 20 point barrier.

2:25:43  
And another example would be a numeric question.

2:25:47  
So another typical example could be how many page faults at most will occur when we copy 2 kilobytes of memory in user space.

2:25:57  
So it's a similar question to what we actually answered today.

2:26:03  
You need some set of assumptions.

2:26:04  
So we need to properly said that there is a 32 bit address space.

2:26:09  
That's very important.

2:26:10  
What is the size of pages?

2:26:12  
That's again important.

2:26:14  
There is A2 level paging again that's important.

2:26:17  
So we need everything set up.

2:26:19  
This is like the easiest thing we can set up because this is actually the the example from the slides.

2:26:24  
But these numbers may vary.

2:26:26  
We can use different size of pages, we can use three level paging for instance.

2:26:31  
So yeah, we can use different different setups, but this is the most simplest 1 and we assume there is enough physical memory so each page fault occurs only once.

2:26:42  
Once we satisfy a page with appropriate frame, once we allocate the mapping, then it won't happen again with this particular page.

2:26:51  
OK, the operating system is simple, which means that every time a page fault happens, it allocates.

2:26:58  
It provides mapping only for that one page because in theory, when a page fault is detected, the operating system might be smart and provide mapping for some other pages like to provide some future mappings that are expected to happen.

2:27:14  
But no, it's not happening over here.

2:27:16  
The operating system is simple and per one fault allocates only one frame for one page and we are ignoring the instruction fetching.

2:27:26  
So let's assume the instructions are already in memory.

2:27:32  
OK, No, fully recognizing that is not standard.

2:27:38  
So when a page fault happens, it doesn't just correct like the single byte that was like it corrects the single page.

2:27:45  
The entire page gets correct.

2:27:48  
Yep, OK.

2:27:51  
In this setup, as I describe it here, we assume if a page fault happens, so a page isn't, the mapping isn't properly set for one of these pages.

2:28:01  
The page will be filled in properly fixed by the operating system and the instruction will be restarted.

2:28:12  
So once a page, this is like a question.

2:28:14  
Another way how to ask this question will be how many pages will be accessed, how many unique pages will be accessed throughout this operation?

2:28:25  
Data pages, we are ignoring instruction fetching.

2:28:28  
That's another way how to pose this question.

2:28:34  
It's 7.

2:28:34  
Let's go through that.

2:28:35  
So first of all, there are, sorry, 2 kilobytes of memory, so 2 kilobytes of memory, Yeah, No, no, no, we have 4 KB pages.

2:28:52  
So first of all 2 kilobytes.

2:28:57  
2 kilobytes can span over how many pages?

2:29:04  
Just two pages?

2:29:05  
That's right, we cannot possibly span 2 kilobytes over more than two pages.

2:29:10  
The best way how to imagine this is imagine that the last byte is at the at the boundary of page.

2:29:17  
So last byte is here and then count how many pages it will cover.

2:29:23  
So if the last byte here then the rest of the block will fit another page without problem.

2:29:30  
Because we have 4 gigabyte page, we are copying the data, which means that this is the source and there is another buffer somewhere else where the same happens for the data destination.

2:29:48  
So the destination where we are copying the data will also be covered by two completely different pages.

2:29:54  
We don't know whether the source and the destination are next to each other or at the opposite sides of the virtual other space.

2:30:01  
We don't know that.

2:30:02  
So we need to assume the worst, that they are widely apart, so we need to compute them independently.

2:30:10  
There might be a different assumption.

2:30:11  
It might be we are moving the 2 kilobytes in the memory right after the 2 kilobytes.

2:30:18  
No problem.

2:30:18  
In that case, we are counting like 4 KB block, right?

2:30:22  
So you need to assess this.

2:30:24  
And finally, when you need to access these and these, you do the same stuff with the page level table, page table.

2:30:30  
So we need to ask ourselves how many page tables at the second level we will need to address these and these, right?

2:30:38  
And the maximum, of course.

2:30:40  
So it's eight in this case, because we have like 4 data pages and each entry for each data page can be at the separate second level table page.

2:30:51  
So in this case it's 8.

2:30:56  
And if you understand how the paging works, you will be easily able to arrive to this conclusion phrase user space.

2:31:07  
Now I suspect that I don't know because I mixed the lectures.

2:31:08  
Yeah, maybe we should write a virtual address space, but that's the same thing.

2:31:15  
Sorry.

2:31:17  
User space is the virtual address space.

2:31:19  
It's the space where the application runs.

2:31:21  
So yeah, maybe I should rewrite it on the slide, it might be more clear.

2:31:26  
But yeah, this means virtual address space, of course.

2:31:28  
See.

2:31:28  
See, that's why subjects like this kind of a lot of the times terms are overloaded.

2:31:36  
I see, I see.

2:31:39  
Practical question #3 or work again.

2:31:44  
This is like something we've done several times.

2:31:46  
So assume that you have this structure in C first item is corrector, then there is double and there is int.

2:31:54  
And we have array of these structures so that the array of 20 for instance.

2:31:59  
And the question is what is the offset of this item?

2:32:05  
So if I write SA 3 dot count so I'm accessing this field of the 4th 3 means 4th right element in this array.

2:32:20  
So what is the address of the beginning of this integer from the beginning of the array?

2:32:26  
The CU?

2:32:31  
Is it best fit or is it just first?

2:32:33  
This question is exact.

2:32:35  
There is no best fit or sorry.

2:32:42  
Yeah, is always one byte.

2:32:43  
That's actually you should know that.

2:32:45  
But yeah, cars are always cars are like the smallest types there are.

2:32:56  
And yeah, we need to know the double is 64 bits and integer 32 bits, just to clarify, because this is not specified by C So this is additional clarification that is required.

2:33:10  
One bit for the car, 8 for the double, yeah, four and four for the in.

2:33:17  
That's right, that is so, except I'm not sure that's how it's supposed to be counted.

2:33:24  
Well, you can count it like that, but it won't help you much in this case, not fairly for.

2:33:34  
So let's figure out what's going on here.

2:33:36  
First of all, there is some padding, right?

2:33:38  
Because we need all these data blocks.

2:33:40  
All these variables needs to be aligned.

2:33:44  
Character is always aligned because it takes only one byte.

2:33:48  
Double is needs to be aligned to its size.

2:33:52  
It means to 64 bits, it means to 8 bytes and count the integer here needs to be aligned to 4 bytes, remember.

2:34:01  
So that's what I'm asking about.

2:34:03  
So that's what we need to do here.

2:34:05  
So imagine how this will be laid out in memory.

2:34:09  
There will be this character type.

2:34:14  
It will take 1 byte and then we need to add padding so the next position is divisible.

2:34:23  
We assume this one is divisible by whatever.

2:34:25  
OK, so this one will be divisible by 8.

2:34:29  
So we need to place additional 7 bytes here, virtual non used bytes so that the following double is properly aligned right.

2:34:43  
This will take 8 bytes.

2:34:46  
This address is divisible by 8 at least, so it's also divisible by 4:00.

2:34:51  
So we need no extra padding here.

2:34:53  
So the int will be immediately here.

2:34:57  
So what's the total size of this data structure?

2:34:59  
Isn't there a specific padding on the bar right?

2:35:06  
Should it I do not?

2:35:09  
OK, so since we are placing this data structure into an array, we need to make sure that not only this double is aligned, but also the double of the following structure.

2:35:23  
So the structure itself needs to be aligned to its largest type inside, so in this case to 8 bytes.

2:35:30  
So we need to wrap it up so the end, actually not the end but the beginning of the next one.

2:35:37  
But that's the same thing here is aligned to the size of eight.

2:35:41  
So we have like a 8 + 8 plus and we need to place additional 4 bytes here so it's properly aligned.

2:35:50  
So the two yeah, so well, it's on the slides just like 10 lectures before this one.

2:36:00  
And total size is 24 bytes 8 + 8 + 8 pair each structure S So if we know that, what is the answer to this question?

2:36:20  
24 \* 3 + 8 + 8, so it's 24 \* 3 because we need to count all the previous structures before the the 4th one.

2:36:32  
That's right.

2:36:33  
And plus we need to this offset, so plus 8 + 8.

2:36:37  
That's right.

2:36:38  
You do the math.

2:36:43  
Yeah, you are allowed calculators during the test.

2:36:46  
So sorry You can use the one on on the laptop on on on the PC no problem.

2:36:55  
So everybody's clear on this one.

2:37:05  
Which one?

2:37:08  
This one So so 3 \* 3 \* 24 is fine.

2:37:29  
OK, so we skip 3 \* 24 and finally we are accessing this this particular field.

2:37:36  
So we don't we have to skip everything that's before that.

2:37:41  
It's three times this structure plus this part of the structure which is before the count.

2:37:48  
So we take how much, how many bytes take these two fields.

2:37:54  
So this is the first one takes 8 and the second one takes 8.

2:38:02  
So yeah, two more questions.

2:38:09  
FAD data structure, This is more like a direct application of what you should know, but again, we need to apply some common sense to it so that you're properly aligned with everything.

2:38:21  
So the setup is we have a FAD 16.

2:38:25  
The 16 here, as you should remember, means that the data structure uses 16 bits per index, but it's not that important in here.

2:38:35  
It's important if I give you hex dump, but if I give you the table with numbers, it doesn't really matter because it just tells you additionally that all these numbers are 16 bit numbers and each cluster, each data block has 2 kilobytes, right?

2:38:53  
So each element in the Fed table represents 2 KB block on the data area.

2:39:01  
And what we are doing here, we are opening a file, a text which has some entry properties in fact.

2:39:08  
And the most important part extracted from this is that it starts at the cluster 3 and the size of the file is 7654 byte of course.

2:39:20  
And we are performing these two operations.

2:39:23  
So the user opened the file, then it seeks to offset 3210 and then the user reads 1000 bytes.

2:39:34  
And my question is write all the indices of all the clusters from the from the data store which will be loaded to satisfy this operation, assuming the fat table itself is already cached in the memory.

2:39:49  
That's what usually happens in operating system because the fat table is accessed all the time, so it's already in the memory.

2:39:56  
We are reading only the data box.

2:39:57  
We are not reading the fat table.

2:40:03  
First cluster has index 2 because before the 1st cluster there is directory entry, but that's not important.

2:40:16  
The first cluster in the Fed table starts with index two.

2:40:19  
You you can read it from here.

2:40:22  
00 actually means in the in the Fed table, 0 means empty again.

2:40:29  
That's something which was described previously on slide and -1 is the indicator of end of the chain, just to remind you.

2:40:38  
So how shall we proceed?

2:40:58  
Sorry, speak up, please.

2:41:00  
I don't hear you first Gust is here.

2:41:09  
That's right.

2:41:09  
So what we?

2:41:16  
Actually, we are not reading the first cluster because we are seeking, you are talking about travel through the fat table if I'm.

2:41:34  
Yeah.

2:41:34  
So let me put that aside for just a second because first we need to know which part of this fat table we're actually going to need to read.

2:41:44  
So first let's just read this.

2:41:47  
Imagine that the file is a continuous block of bytes and it's stored chopped by the size to the cluster by the size.

2:41:57  
So we have like 2 kilobytes, 2 kilobytes, et cetera.

2:42:03  
And 1st we need to know which part of the file will be actually read, because we are skipping this.

2:42:12  
So we are skipping 3210, which means we are skipping entirely the first block.

2:42:19  
This is just two kilobytes and we ended up somewhere in the middle of the second block, right?

2:42:25  
Then we read 1000 bytes.

2:42:28  
1000 plus 23210 is still here.

2:42:33  
Or does it cross the border to the next class?

2:42:38  
It crosses here.

2:42:39  
So we are reading this part.

2:42:41  
So we will need to load.

2:42:44  
Since we are loading always the whole cluster, we will need to load the.

2:42:49  
If I number it from the zero.

2:42:51  
So the zero is here we skipping the zero.

2:42:53  
We need the 1st and the second cluster of the file loaded and now we need to translate which actually data blocks correspond to these clusters.

2:43:04  
So now we are resuming what you are saying.

2:43:16  
Yeah.

2:43:16  
So just to explain to everybody, we first cluster is 3 and we are skipping that.

2:43:22  
So we are ignoring that.

2:43:24  
But we need to read the next cluster from the position three.

2:43:28  
Position 3 is here, so we read 10.

2:43:32  
So the second cluster, cluster number one is actually placed at the data block 10.

2:43:39  
So we are reading data block 10 and we also need to move on as you suggested position 10 exactly.

2:43:53  
So we need to find out the number of the next cluster which is 15.

2:43:56  
So the second or third data block block number 2 is stored at the data block 15.

2:44:05  
If we need to continue, then we just move on.

2:44:08  
15 is over here, so it moves to 14, and 14 is here and holds -1 which means we are terminating the link chain over here.

2:44:16  
So this is a pointer chase.

2:44:17  
We are just traversing the data structure, the linked list, to find out which data blocks are there.

2:44:23  
But remember, we need to find out which of these data blocks are actually being read.

2:44:28  
That's perhaps best to start with.

2:44:30  
This is the first step.

2:44:32  
And by the way, if we need only these two blocks, we don't have to chase the pointer all the way down.

2:44:38  
Once we find out that the second, sorry, the third data block is at the data block 15, the third cluster is at the data block 15, we're finished.

2:44:47  
That's it.

2:44:48  
So the answer will be 10 and 15.

2:44:53  
Yep.

2:44:58  
The size is also the size also matters because we need to perform a mental check whether these two numbers together are still lower than this one.

2:45:10  
Yeah, but that's just to verify.

2:45:13  
In this case, the size is somewhat slightly redundant information because we don't care.

2:45:19  
But if you're if you're writing, for instance, we might be interested in that because the size might indicate when the file is growing, for instance.

2:45:28  
But in this case, this is just to mentally verify that we are still within the the the file for because if we perform seek operation beyond the file, it might have some repercussions if we if we exceed the size.

2:45:49  
Last week finally, yeah, finally few words about instructions.

2:46:01  
This is actually a simple 1, so and a question like this might rise.

2:46:06  
We have a piece of code in assembly language in MIPS instruction set.

2:46:13  
And this is like this is a simple 1.

2:46:15  
So we can we can make it quick and it's comprised 2 additions and one subtraction, right?

2:46:21  
And both editions are operating on different inputs.

2:46:25  
Remember this is the output, this is the input, this is the output, this is the input.

2:46:30  
So we are taking two different inputs for each edition, creating two different distinct temporary values.

2:46:38  
And finally, we're taking these two temporary values and subtracting each other.

2:46:44  
Which line of code corresponds to this assembly?

2:46:50  
Yeah, this is easy.

2:46:51  
The 4th 1, you can go the other way around.

2:46:53  
Since we have two ads and one sub, only these two lines actually qualify.

2:46:58  
So even to perform a quick pre filtering, only these two lines seems to be legitimate for this kind of operations.

2:47:07  
Yeah, Obviously we tend to play slightly more difficult pieces of assembly code, usually involving some jump because that's where things get interesting.

2:47:18  
But basically this is how you proceed.

2:47:20  
You just identify the instructions, try to map them on the basic operations.

2:47:24  
And then we need to identify this is completely different from this, right, Because we need to perform the additions first and then the subtraction not perform to this could be mapped to this set of operations.

2:47:36  
OK, that's pretty much it.

2:47:39  
Are there any more questions regarding anything?

2:47:57  
Yeah, nothing will be read because the question is which clusters will be loaded.

2:48:04  
So the if the user tries to read beyond the end of the file, no clusters will be loaded.

2:48:09  
The the the operating system will handle it some way.

2:48:12  
Actually, it doesn't matter how it will be handled.

2:48:14  
No clusters will be loaded because there are no clusters after the end of the file.

2:48:28  
Then you need to follow this chain to the end and enumerate all the clusters.

2:48:33  
So sorry well if if the read operation needs to be satisfied, you need to find out which clusters need to be loaded to satisfy this read.

2:48:46  
Only the reads and writes matter.

2:48:48  
Seeks are just jumping.

2:48:49  
Seeks are ignored basically.

2:48:52  
So he could just move you through this pointer chase, but they don't load any clusters.

2:48:59  
OK, any more questions?

2:49:04  
No, everything is clear.

2:49:05  
Hopefully a lot of stuff of money, but I couldn't even articulate the question necessary to get cleared up.

2:49:12  
Well.

2:49:12  
If you need consultation later, don't afraid to ask me.

2:49:16  
We can set something up.

2:49:17  
But in the end, you need to just go through the materials yourself.

2:49:22  
Try to try to, you know, put them in in the right order so that you can understand them.

2:49:28  
But as I said, if you have any questions later, just write me.

2:49:33  
OK, great.

2:49:36  
So if there are no more questions, let the Mayhem mayhem begin.

2:49:42  
Good luck to you all.

2:49:44  
I hope I will see each and everyone of you exactly once during the exam.