ProgrammingParadigms-Lecture14

Instructor (Jerry Cain):We're on. Hey, everyone. Welcome. I don't have any handouts for you today. I actually want to finish up the implementation section of the course. I think we'll get through it today. I'll make it a point to finish it today. Come Monday, we're gonna start talking about multithreading and I'll even preface that a little bit today, provided I don't run out of time. Remember that your midterm is Wednesday evening over in Hewlett 200, the largest room on this end of campus. It's 7:00 to 10:00 p.m. It's open note, open book. You can bring print outs of your programs, whatever you need. Just plan on taking the exam in the three-hour period. It's designed to be more than enough time for the midterm. Okay. There's also the technology talk right after this class right here over in Hewlett 201. So you can first visit Hewlett 200 to see what room it's like for the exam. But in 201, there'll be a technology talk from 12:00 to 1:00. Okay. I left you last time with this example, but I got several questions about how it worked, which probably means that I rushed through it in the last five minutes of class, which is probably true. We had something like this, where I declared an int array of length four, and int i to serve as four loop index, and then I'm just gonna go and do this. I don't care that the array hasn't been initialized. I want to go ahead and do this right here. And then just return. What you probably do remember from Wednesday is that given R memory model, that this would prompt the program to run forever. Why is that the case? Based on this local variable set, we're dealing with this as an activation record. One, two, three, this is the array. As far as that four loop is concerned, it's just one too small. This is the i variable. It goes through and it demotes all of these variables by four. What values were they before? We have no idea. But it will certainly take whatever values happen to reside there and demote them by four. Unfortunately, because the test is wrong, it goes up and it demotes this numerically by four, as well. Now that four isn't so much a four as it is – it isn't so much a negative four delta as it is a negative one instruction delta, because you know that this is the Safe PC in our model. This was supposed to be pointing somewhere in the code segment to the line that called the Fu function. This is planted down there in response to that assembly code statement. Does that sit well with everybody? Okay.

When you do this, you inadvertently tell the Safe PC that is wasn't pointed to this instruction, but that it should back up four bytes, which happens to be this nice round number, as far as assembly code instructions are concerned. And to stop pointing there, and to point there, instead. So when this as a function returns, it jumps back to this right here, and it executes the call, having no memory whatsoever that it called Fu like 14 or 15 assembly code instructions ago. Okay. Does that make sense to people? So this is this very well disguised form of recursion. You won't be able to emulate this on the Solaris boxes or on the pods or the mist, because their memory model is a little bit more sophisticated than ours. They don't put the Safe PC right next to this right here. But nonetheless, that is probably the fifth example of infinite recursion we've seen in the last two days. Let me show you another program. No infinite recursion in this one. I want to write a simple main program, int name. I don't care about the parameters. I want to do this. I want to do – let's just say, "declare and init array." Now, I'm not declaring any local variables whatsoever. So you should already be a little bit suspicious as to what that type of function should do. But imagine the CS 106a program in week four, learning C

instead of Java. And they just don't have arrays and parameters passing down. So they have this right here. And then afterwards, they have this things called "print array." And that is it. Now unless there's global variables involved, there's no legitimate, even though we don't like globals, it still would be a legitimate way to communicate information between function calls. But suppose there's no globals, either. The program doesn't think they need globals because they do this: void declare and init array.

And they just declare an array of length 100. They declare the forward variable i. They're not going to overrun the bounds this time. They're gonna get it right. And they're gonna set array of i equal to i. So that's a beautiful little code, but it does very little outside of the scope of the declare and init array function. But for whatever reason, they've decided that they want to declare this array, locally update it to be a counting array, and then leave. Okay. Then they come back and they want to print out that array and this is not that uncommon when we taught in C. They think as long as they name the array the exact same thing, that all of a sudden there's a relationship set up between that array and this one, as if the word array has now been reserved throughout the entire program. And they do this and then they do this for i is equal to zero. They get the forward bounds right again; that's not the issue. Print out percent D backslash N array of i. And they come in during office hours because there's some part of the program beyond "print array" that's not working properly. But then a TA looks at this or I'll look at this and say, "Oh, this is wrong right here." And they're like, "No, that's working fine. It's actually down here." Well, it may be something that's wrong down here, as well, but clearly there's something wrong going on right here. However they make the argument, "Well, it's working." And the answer is, it is working as far as they're concerned, because that manages to print out zero through 99, inclusive. You all probably have some sense as to why that's happening. Try explaining that to someone who's never programmed before. This right here, built-in activation record of 104 bytes, goes down and it lays down the counting array in the top 100 of the 101 slots. Returns.

It calls a function over there that has exactly the same image for its activation record. So it goes down and embraces the same 404 bytes. This is me embracing 404 bytes. Okay. And it happens to print over the footprint of this function right here. It's not like when this thing returned it cleared out all of those bits and said we got to scramble this so it doesn't look like a counting array. It doesn't take the time to do that. So basically, what happens is if this is the top of the activation record and this is the bottom of the activation record, SP is descendant once, this is filled up with happy information, comes back up after the first call returns, comes back to the same exact point, the happy face is still there. We print over it and it happens to be revisiting the same exact memory with the same exact activation record, so it prints everything out exactly the same. Makes sense? Now, there's some advanced uses of this, where it really does help to do this. A lot of times, not in sequential code like we're used to, but about 11 years ago, I had to rely on this little feature when I was writing a driver for a sound card. And you had to actually execute little snippets of code with each hardware interrupt. And so a lot of times, you had relatively little to do in one little heartbeat of an interrupt, and you had a lot to do the next one. In fact, you had so much to do that you weren't sure you were going to have time for it consistently. So a lot of times, you would prepare stuff ahead of time and put it

in exactly the right space so you knew where it was the next time without having to actually generate it. Do you understand what I mean when I say that? Okay. So you almost think of this as this abusive, perverse way of dealing with a global variable. And you're setting up parameters for code that needs to run later on. This example wouldn't exactly work that way. This isn't a great example of that, but at least it highlights the feature.

This thing called "channeling," that's exactly what this thing is when really advanced C++ programmers take advantage of this type of knowledge as to how memory is laid out. As far as the problem with hand is concerned, you feel helpless, you try to explain, for instance, if you've just put a "call to printf" right there, that its activation record has nothing to do with these activation records. So it goes in and garbles all of the sacred data from zero to 99. And their response is, well, just don't do that. And comma is out and they think they restored program, but they really have not. Does this make sense to people? Now, I want to revisit this printf thing, actually. I don't know how many of you know the prototype for printf. You really are just learning it, basically, on the fly in context when you're dealing with a simon four and you're just seeing lots of printf's in the start code and you just kind of understand that there's a one-to-one mapping between placeholders and these percent D things or percent G things and the number of additional parameters. Well, you know sample calls are things like this: printf hello. And that's in contrast to C out less than less of hello is string constant with an "endl" at the end. When you have something like this: percent D plus percent D equals percent D backslash N, and you want to fill it in with four and four and eight, if you wanted to do it that way, you certainly could. But the interesting thing from a programming language standpoint is that printf seems to be taking either one argument or four arguments, or really, it takes anything between one and, in theory, an infinite number of arguments. It's supposed to be the case that those things line up, in terms of placement and data type, with the additional parameters. What kind of prototype exists in the language to accommodate that kind of thing? Well, we always need the control string.

That's either the template or the verbatim string that has to be posted to the console. So the first argument to this thing is either a char star or a const char star that can polish supports const. So I call it "control." But then, there's no data type that really has to be set in stone. There doesn't even have to be a second argument, much less a data type attached to it. I could put a string there. If this is a percent S, a float there if this is a percent G, things like that. The prototype for that is dot, dot, dot. And so forth. Whatever they want to type in. And the complier is actually quite promiscuous, and what it will accept is arguments two, three, and four, provided that dot is there. If you don't want to insert anything, it's fine. If you want to insert 55 things, it's great. The complier is not obligated to do any kind of type checking between this and what this evaluates to. There's nothing implicit in the prototype right there. It's just a char star, free form char star, and then whatever you want to pass in. You want to pass in structs? Great. Pointers, structs, you can do that. GCC, for quite some time, has an extension to the C spec that is implementing. Where it does try to do type checking between this and that right there, if you mismatch, it's doing a little bit more work at compile time than it's really obligated to do. It wants to make sure that this printf call works out. So if you were to put percent

S, percent S, percent S, and put four, four, eight there, most compliers would just let you do it and run and it would just lead to really bad things. But GCC will, in fact, flag it and say, you didn't mean to do that. Okay. Does that make sense? This return type – I mentioned this on Wednesday – this return type is the number of placeholders that are successfully bound to. So as long as everything goes well, it would be zero for this call and three for that call. It's very unusual for printf to fail. Scanf, the read in equivalent of printf, can fail more often.

But if, for whatever reason something goes badly, this would return negative one. Do you know how IF streams set the fail bit to true so that when you call the dot fail method inside C++, it'll basically evaluate to true and that's the way you break out of a file reading program? Well, you're relying on the equivalent of printf's return value, which is called "scanf" or "f scanf" to return to negative one when it's at the end of the file. The reason I'm bringing this up is because, based on what we know and the way we've adopted a memory model, I now can defend why we push parameters on the stack from right to left, why the zero f parameter is always at the bottom and the one f parameter is always above that. Let's just consider that call right there. The prototype during compilation just says that either of these calls is legitimate, but when it actually complies that second printf there, it really does go and count parameters and figures out exactly how many bytes to decrement the stack pointer by for that particular call. So the way that the stack frame would be set up is that this would be the Safe PC that's set up by the call to printf. Above it would be a pointer to the string, percent D plus percent D equals percent D backslash N. And this would have a four, this would have a four, and this would have an eight. The activation record for the first call would just have this many bytes, and would have a pointer to the hello string. So the activation records, the portion above the Safe PC actually is completely influenced, not surprisingly, by the number of parameters that are pushed onto it. Now, when we actually jump to printf, and this is where the SP is left, it doesn't have any clear information about what resides above the one char star that's guaranteed to be there. Does that make sense to people?

So really all that happens – I'm gonna draw this arc again, like I did before – it knows about that much, if there were special directives in the implementation of printf that allow it to manually crawl up the stack. But a number of arguments and the interpretation of the four-byte figures that reside there, it could only figure that stuff out by really analyzing and crawling over this control string character by character. This is almost like the roadmap to what resides above it in the stack frame. Does that make sense? So the printf function really does need to get to the control string, and it reads it character by character, and every time it read a percent D – let's say if reads a percent D right at the front, it says, oh, the four bytes above the control string must be an integer. And then it sees another percent D along the way. It says, above that there must be some other integer. And this is how it discovers the stuff that should fill in the control string. So you understand that, otherwise, if it didn't have this right here, this would truly be a big series of question marks. It's still kind of is a series of question marks. If this is the wrong roadmap, if I do percent D plus percent D equals percent D and I pass in three strings, these things will be laid down as char stars. That's what the caller would do. And then it would interpret them as four-byte integers. So whatever addresses happen to be stored

there would be taken as unassigned integers, and it would just fill those three things in that way. Makes sense? This is consistent with the way we push parameters onto the stack, from right left, last argument first, then the second to last argument below that, etc., so that the zeroth argument is at the bottom.

Imagine the scenario where the Safe PC is addressed by the stack pointer, but you have the mystery number of bytes below the control string. And that question mark region would be of height zero for the printf hello call, and of height 12 for the printf four plus four is equal to eight call. It would have no consistent, reliable way of actually going and finding the roadmap as to how to interpret the rest of the activation record. Does that sit well with everybody? So as long as you understand that, then at least you have a defense for why that dot, dot, dot – because of the dot, dot, dot, the C spec more of less has – I guess it doesn't have to, but it just made sense to for compliers to implement this left to right parameter pushing strategy. Because they want to support that dot, dot, dot in the language. C++ has to do the same thing, because it's backwards compatible with C. Java just recently introduced the ellipses – I think it was either Java 1.5 or Java 1.6, I'm not sure – but very recently they introduce the ellipses. And so I just know, without actually reading anything about it, that they have to push their parameters on the stack in exactly the same way. Pascal, old school, wasn't old school for me when I was in college, but it's old school for everybody here. I didn't learn Pascal. I learned C first, but when I read about Pascal, it doesn't have this ellipses option. You have to specify the number of arguments. It happens to press the argument on in the opposite order. And it doesn't cause any problems. They had the flexibility to do it an either order because they never had to deal with this question mark region in a struct. Does that make sense? But as far as structs are concerned, you may ask – this is a hard point to make; I'm gonna try and do it. Struct Fu, let's say I have an int code and I have, let's say this. Int code, and I'm just gonna do that right there.

It's unusual for you to have a struct around one data type, but I'm gonna do it anyway. And then I have struct type one, which has an int code and, let's say, several other parameters. Maybe it's the case that the code inside a type one struct is always supposed to be one. So just take this as a series of equipments of the example. If I have struct type two, with an int code at the front, I might require that all instances of type two actually have a two at the front. These are really esoteric examples. I'm just making this up. I haven't done this is past quarters. But this right here is a data structure that I guarantee, whenever I give you a pointer to a struct base, the idea is that there is one of two values that sits right there. It's almost like it's a little opt code in the assembly instruction, in a sense, it to figure out how to interpret what resides, or figure out what resides above the one or two in memory. You could cast that pointer to a struct base knowing that there's gonna be at – or maybe it is typed to be a struct base, which means that you know that there's some kind of opt code or type code sitting there. And then based on the result here, you can either cast this arrow to be a type one star or a type two star to figure out how the rest of the information is fleshed out. A complicated example, but there are various structs – I don't know whether you've looked at – I don't think I've exposed the code for all the networking in the URL connection stuff. If I did, you would have hated Assignment 4 even more, because you would have thought you were responsible for it.

But there are lots of structs that are afforded by GCC and G++ to help manage networking, and I tried to insulate you from that. Old school networking deals with four byte representations of IP addresses. They realized about 15, 20 years ago that they were going to run out of IP addresses pretty soon, so there's actually a six byte universal version of IP codes. It's standard, but it's not really widely adopted yet. There are two different structs associated for the two different protocols. The four byte version IP, version four, there's IP version six. The IP version four struct has been around for some 25, 30 years. Those things aren't going to go away. So when they designed the IP version six struct, they had to make sure that the first half of it had exactly the same structure as the IP four version, and then they extended it with all this extra information. Does that make sense? Do you always know that you're getting a pointer in networking code?

You always know you're getting a pointer to one of those two structs, and you analyze the first few bytes to figure out whether or not you have an IP four struct or an IP six struct. Does that make sense? The reason I'm mentioning this is because now it kind of gives you some sense as to why the first parameter, or the first field in a struct or a class actually has to be at the lowest address. It doesn't have to be, but that's just the way they did it, because they had these types of things in mind. If this and this and this were always the last thing, or the thing at the top of the struct in the activation record, it would always be at a mystery distance from the base address of the entire thing. Does that make sense? Now you could just argue that you could make the code the last thing in a struct so you could do it the other way. It just makes sense to people who designed the spec or designed the compliers. I don't think it's part of the specification, although maybe it is, that the first field is always in offset zero, and you can exploit that knowledge to do clever things with C and C++ structs. Does that make sense? So I'm cranking on time. So what I will do now is I will give you a little bit of a head's up on how we're gonna transition things. Everything so far in C and C++ has been sequential. And I'm talking in Java it's – actually, not in Java necessarily, but all of the C++ and C you've done, in all 106b and 106x, and for example, 107, has either been strictly or seemingly sequential. And you know what sequential means now, because you're waiting for the sequence of RSS news feeds to all load over a five-minute period while you test. So you know what sequence means more than you did a week ago. I want to introduce the idea of how two functions can seemingly run at the same time. That's going to be a huge win in the context of Assignment 4. You'll all be delighted that you're going to revisit Assignment 4 before Assignment 6. And you're going to make it run oh so much faster by using this thing called threading. What I want to do is I want to talk about how two applications on a – and just give you very high level stuff – two applications can seemingly run on the same processor simultaneously, and then use ideas from that to defend how two functions within a process can seemingly run simultaneously. I'll frame this this way. Let's just think about one of the pods. This is the virtual address base, virtual meaning the allusion of a full address base that make has wireless running. You don't think about Make as an application, but it certainly is.

It's designed to read in the data file that you call a Make file to figure out how to invoke GCC and G++ and the linker and purifying and all of those things, to build an instrument and executable. This has a stack segment associated with it. All the local variables of the

thing that implements make go there. There is the heap. There is the code segment. While make is running, it's probably the case that GCC, as an executable, is running several times, but we'll just talk about the snapshot or time slice where just one GCC is running. GCC is an executable. You're first C complier was probably written in C – not written in C, was written in Assembly, but then it kept bootstrapping on the original compiler to build up more and more sophisticated compliers. So the C complier was written in C, I'm sure of it. It also thinks its stack is there and its heap is there and its co-segment where all the assembly code stuff resides right there. I can tell you right now that they're not both all in the same place. They do not share the stack and they do not share the heap and they do not share the code. This virtual picture was in place so that make can just operate thinking it owns all of memory. And it lets the smoke and mirrors that the OS managers, to map these to real addresses and map this to real addresses and map that to real addresses. It just wants to be insulated from that. Maybe it's the case that you have, I don't know, Firefox up and running on one of the Linux boxes, has the same picture. And then you have some other application, like Clock or whatever you have, up there and it has the same exact picture. Those are four virtual address bases that all seem to be active at the same time. I'm gonna call this process one, I'm gonna call this process two, call this process three, and I'll call this process four.

When I draw these little bands of segments right here, these segments is what they're called, they're all assuming that they have as much room to stretch out to make the stack as big as necessary, the heap as big as necessary, to meet the demands of the program. But on a single processor machine, there is only one address base. This is the real deal right there. That's physical memory. And this has to somehow host all the memory that's meaning to GCC and Make and Clock and Firefox, or I should say the processes that are running according to the code that's stored in those executables. Does that sit well with everybody? Well, it turns out that this and that right there, they may have the same virtual address, but they can't really be the same physical address. The space has to be truly owned, or the values there in virtual space have to be truly owned by this process I've called number one. So what the operating system will do is it will invoke what's called the memory management unit to basically build a table of – let's say that this is address 4,000. It'll actually build a table of process and something related to an address. And it'll actually map it to a real address somewhere in memory. Does that make sense? The idea, I think, probably makes sense to people. So that this right here, it thinks it's storing something in address 4,000. Let's say address 600,000 is right there. Any request to manipulate or deal with address 4,000 is somehow translated to a request to deal with the real address at address 600,000. Any type of load or store or access of this right here has to somehow be proxied or managed by this daemon process behind the scenes, this thing that just runs in the background all the time, to actually map virtual addresses to physical addresses. And it knows that this address 4,000 is the one that process one owns, so it just has this little map of information – I've drawn it has a map of pairs to physical addresses – so it knows exactly where to look on behalf of this process.

It stores them in address 4,000; it updates the four bytes that resides at address 600,000. Now, it doesn't really clip these things down at the four-byte level. Normally, what will happen is it'll allocate things – I don't mean allocate in a malik sense – it'll just associate

very large blocks in virtual address space with very equally large blocks in physical address space. So this might be some 1K or 8K block of memory. And if it's ever used, then it's sure to map the same size block or adopt the same size block in physical memory so that address 4,000 through, let's say, address 8,000, would map to whatever this is through whatever this is plus 4,000. So there's some contiguous nature going on between all the things in this virtual address base and what it maps to in physical space. Does that make sense? A lot of work. It's a very difficult thing to implement, certainly, the first time. I've never implemented one of these things. Maybe it's even more difficult than I'm giving the impression it is. But it's doing all this stuff in the background, it's threads, it's all kinds of stuff that makes OS and systems code interesting, but difficult. So we've solved the memory problem. In theory, you can run 40 applications. Usually it's the case that the stack segment is never really that big. It's initially slotted off to be fairly small for most applications. Because unless you're going to call Fibonacci of a billion, it's probably not going to have a call depth greater than 50 or 60. In fact, when you have a stack called f of 50 or 60, it usually means a lot of things are happening. The distance down from main to the subhelper to the 59th power function. I don't want to say that's unusual – maybe 100 is normal, maybe 200 is normal; 2 billion is not. So you know that most activation records are on the order of, let's say that they're 1K. It might be the case that you set aside 64K for the virtual stack or the stack in virtual space. You have two the thirty-second different addresses. 64K is ridiculously small amount of that.

It's like that. So it definitely has space for it. You could be more aggressive about the way you use the heap. You could allocate megs and megs of memory there. It's still going to be a relatively small portion of memory when you're talking about two to the thirty-second different addresses. Makes sense? So the smoke and mirrors that's in place so that every single application can run at the same time and not have its address space, or what it thinks is its address space, being clobbered by other processes. That's managed pretty well by the OS – not pretty well – ostensibly perfectly by this memory management. You can share address spaces across applications, but you have to use advanced unit directives to do that. The part that is not clear, and this is going to become more clear, hopefully, next week and the Monday after it, is how the applications seemingly run at the same time, when there's really only one register set, one processor digesting instructions at a time. I did this the first day of class, but it totally makes sense to do it again here. Forget about Firefox and Clock, let's just deal with Make and GCC, which is what you've really been doing. And think about Make and GCC actually running seemingly sequentially.

You look at them both running – and my hands are sifting over the assembly code instructions. And they're both seemingly running at the same time. That's not what's happening. What really happens if that Make makes a little bit of progress, and then GCC makes a little bit of progress, Make makes a little bit, and this just all happens in this interlay fashion, so fast that you don't see any one lagging over the other one. It's like watching two movies at the same time, where not much is happening. So you can actually follow both movies fairly well, as long as it's clear that both of them are actually running. The argument for two hands scales perfectly well to three hands and five hands and ten hands and 50 hands, as long as the processor has the bandwidth to actually switch

between all of the processes fairly quickly. That make sense? Now in a dual processor machine or a four processor machine or a multiple core machines, it can actually really run two processes and four processes at the same time, but you can always run more processes than there are processors on any sophisticated system. If something's running a dishwasher, then it probably can't deal with threading, but if it's actually running some real program, it probably is dealing with a real processor, and the OS can actually dispatch and switch between processes fairly quickly. Makes sense? The reason I bring this up is because that, as a concept, is going to translate, I think, somewhat nicely to the notion of threading. This is multiprocessing. Several processes are seemingly running at the same time, and each process has its own heap and its own stack and its own code segment, and its virtual space.

Slightly different, but certainly related, is the idea that two functions in the same process, one code segment, one heap segment, technically one stack segment. We're curious as to whether or not it's possible for two functions to seemingly run at the same time inside a single process. You know; you've seen this before. Microsoft Office, like you're typing and then while you're typing, all of a sudden in the background, some little paperclip comes up and says, I think you're trying to write a letter. And that happens in the background, and that's because something in the event handlers that actually catch your keystrokes have done enough synthesis of the string to look that it looks like a header of a letter. And so it spawns up this other function that doesn't – it's not really supposed to interfere with your typing, and from a computational standpoint, it doesn't. From an actual mood standpoint, it does, because you actually go down and look at it. But that is an example of a thread that is spawned off in reaction to an event, or something like that. That makes sense?

iTunes, you buy an album of 13 songs – I'm hip; I buy music online – and you check the actual download screen and three songs at a time are actually downloading. Not really. It's really doing – and iTunes is another hand over here. And it's pulling things in in little time slices, but it happens so fast and the time slices are so small compared to what we can detect, that it looks like all three songs are being downloaded simultaneously. There's one process going on there. It's not like it spawns off a different executable called the "paperclip executable." It's just some function to bring up that little widget. When you're downloading music, there's one process that's doing it, it's iTunes. And internally, it has some function related to the downloading of a single song that at any one moment it's allowing to run, seemingly, three times – sorry, three at the same time. Does that make sense to people? So just imagine the scenario where there are two songs downloading at the same time. In that case, they both would be following the same assembly code block. They have the same recipe for downloading a song, for the most part. In that case, the stack segment itself could be subdivided into smaller substacks, where the stack frame for song one could be right there, and the stack frame for the downloading of song two is in this completely unrelated space – not unrelated. It's in the same segment, but far enough away that you're not going to have one stack run over the top of another one. Do you understand what I mean? So when the process has the processor, it also subdivided its time to switch back and forth between the two threads – that's what you call these things. And so basically, it goes you have time, you have time, you have time, hope you're

downloading songs. And it keeps doing that until one or both of them ends. Does that make sense? It shares the heap. So they all share the same call to malik and they all draw their memory from the same memory pool.

There's only one copy of the code. You only need one copy of the code. It's read only. It's fine for this stack and this stack to both be guided by the same set of assembly code instructions, as long as each one has some kind of thread identifier. If the word thread is bothering you, just think take the phrase "thread of discussion" or "discussion thread" and just translate it to "function thread." Does that make sense? You may ask what types of scenarios actually require threading and which ones really don't require threading. Let me just go over this very simple example where you really would expect threading to be in place to model the real world situation. I will revisit this example on Monday. I have this really simple program, int main – and there's no threading whatsoever. I have this four loop, int num agents. When I call this num agents, I'm actually thinking about ticket agents who answer the telephone at United or some other airline that still hasn't declared bankruptcy yet. Num agents is equal to ten, and let's assume that the job of this program is to simulate the sale of 100 tickets for the only flight that happens to be flying anymore. You might do it this way. You might intrinsically hard code the 100 in there by calling some function "sell tickets," where you pass in i and you pass in ten.

And I don't need to tell you what sell tickets – I'll write code for sell tickets in a second. But the idea here is that we have to sell 100 tickets. This is the number of agents right here. That number is the same as there. In fact, let's say that there are 150 seats on the flight, so we don't have to mix ten's. The idea of "sell tickets" is that it's supposed to be in place to simulate the requirement that some ticket agent actually sells 15 tickets before his or her job is done. As long as these run in sequence, eventually you'll get to the point where you actually sell 150 tickets in this really rude but simple simulation. The problem here is that in the real world, it's just fine for all ten of these agents to be answering telephones simultaneously. It's not – I don't want to start introducing thread functions yet, but I just want to leave you with the idea that we're going to be able to get that function right there to run in ten different threads. In other words, I'm going to spawn off ten different threads. All of them are going to be following the same exact recipe, where each one has to sell 15 tickets, and when a particular thread exists, we just know because of the way we code it up that 15 tickets have been sold. I'm not going to write code for this yet, but this is, I think, a fairly good analogy. Imagine a horserace or a dog race. Sequentially, if you wanted all ten dogs to get to the finish line, you could just let one go, and when it gets to the finish line, let the next one go. And just do it that way, and take 15 times as long or ten times as long as you really need to. Or you can line them all up in ten gates and lift the gates at once. And some will be faster than the other ones, but eventually, they're all going to get across the finish line. Like you're basically pipelining and taking advantage of the fact that things can happen more or less at the same time. Now the difference is that they don't run like this every time, and they're not respecting time slices.

They actually are really independent agents. But we're going to try and simulate that idea as much as possible with this example when we introduce the thread library next time. So

this is a great place to stop because I’m at the end of a segment in the course, and to try and introduce threading in five minutes would be pretty difficult. I will see you on Monday and we’ll talk more about threading then.

[End of Audio]

Duration: 44 minutes