ProgrammingParadigms-Lecture15

Instructor (Jerry Cain):Here we go. He's obviously ready. Welcome. I have three handouts for you today. I put them up on the board because they're not in sequence. I went back and used the No. 4 Assignment 5 that I gave out last week, a week ago today. So people who are watching on TV make sure you go back and get the solution to handout 19. I think that's what it is. Let me just check, yes, it is indeed. So there's all of these handouts in the last week that have corresponding solution sets, so make sure that you actually download the solutions as well so you can compare your answers to mine and make sure they're in sync with one another. You know you have a midterm on Wednesday evening, it's 7:00 to 10:00 in Hewitt 200. It's a huge room over there; plenty of space as well as the other room, so it'll be a great place to hang out for three hours. I want to be clear that I am certainly covering co-generation; you've seen that on the sample exams that I gave out last week. I am not gonna include co-generation for C++ features, no true references and no object orientation, no methods. So pointers, that's fine, anything related to asterisk, that's real C and I've emphasized that in the early part of the co-generation, but references and methods, the co-generation for that it's not testable in the midterm. You will certainly see it on the final. I always know exactly what type of questions I put on the final for that stuff, but you will not see it this Wednesday. Okay? I also promise to not have any preprocessor or linker or compiler stuff. I went through that kind of as a transition from C figuring out how to build executables from the C language, but I'm not gonna test that material because I have tested it in the past and it just never goes well because it's so esoteric and we don't have any kind of real problems to exercise the materials. So people just didn't do well, so I just stopped testing it. When I left you last time, I had written this partially simple program that was supposed to model the selling of a 150 airline tickets on a single flight. So let me repeat that and point out why it's problematic and how we're gonna move away from it.

I went ahead and did something like this. I wrote it a little bit differently last time, but I'll write it like this, num tickets is equal to 150. All I want to do, in this brute force four loop, I'll write agent is equal to one agent less than or equal to num agents, agent ++ – I went ahead and I called this function, called sell tickets, and I'm gonna frame in terms of the agent ID num tickets dib, not agents, so that it's planarized in terms of these two values right here, and [inaudible] each ticket agent knows that he or she has to sell that many tickets as part of his or her function call and that's it. That's return zero to satisfy the compiler. Okay. We're not gonna allow anything to go wrong in this simple program. The implementation of sell tickets – it's not gonna be rocket science. I'm gonna write it a little bit differently than I would traditionally write because I'm paying forward to the way we're gonna change the example to in a second. Void sell tickets, int agent ID, int num ticks to sell, and even though it's a little weird, let me not use a four loop, let me use a wild loop. Wild loop is the case that num tickets to sell is greater than zero – go ahead and do print F agent percent V sells a ticket. Agent number and then does num tickets to sell minus minus, finally, my arms tired, but I'm gonna keep on writing, print F agent percent D all done. Agent num – this arms gonna be bigger than the other one by the end of lecture. Okay. There we go. From a code standpoint, it's moronically simple. I'm not trying to revisit four loops and wild loops. What I'm more interested in doing is figuring

out why this is as simulation is really not all that good in the sense that it's not really modeling what would truly happen.

The way this is set up, and I'm speaking as this is new material, but it's not, it's clearly gonna be sequential and ticket agent one is gonna sell all of his or her 15 tickets before anything happens with ticket agent two. So you know that the print out of this would have a 160 lines, 16 lines per ticket agent – okay, I'm sorry, yeah, 16 lines per ticket agent, but they would be sorted all ticket agent number one followed by an all done comment, that's the 16th. Okay. Does that make sense to people? Okay. I'm sorry, 165 because there's 15 agents going on here – no, I'm sorry, there's 10 agents so that's 160. I'm just confused, but everything's gonna be all about agent number one before it's agent number two, before it's agent number three, etcetera. I really don't like that. Okay? This is a fairly compelling example where if we're really trying to model the simulation of an actual airline ticketing room that you want to see all of these ticket agents running simultaneously and working, not competing, but working collaboratively to sell all 150 tickets at the same time. Now, what I'm gonna do is I'm gonna repeat the four loop, int agent, agent, less than or equal to the agents, agent ++. Here is how you set up the actual dogs at the racetrack. What I want to do is I want to create a name that's unique to a particular ticket agent. I'm gonna do that by declaring this in place buffer – you haven't seen this function, it's not the emphasis, but I might as well show it to you cause it's in the handout. I'm gonna print F, not to the console, but to a character buffer, there's a function called S print off to do that rather than actually echoing the characters to the screen, I echo the characters in place to a character array and make sure it turns out to be a C string.

The place where they function as that console of sorts begins at the name address, okay, and as long as I don't print more than 32 characters I won't overrun the boundaries of this thing. This is the structure of what gets printed and then I will fill in agent. So for all intensive purposes, on the zero generation or the first generation of this thing, after that S print off call is made, name contains – it goes from garbage to the C string, agent one thread. The reason I do that is because I want to call this function called thread new and the name actually serves as the name of the thread and it's also helpful for debugging purposes should you be passing a true to that thread package. And then you go ahead and you pass the address of the function that you'd like to execute in a single thread of execution. Okay. This right here is just an arbitrary function pointer. All of the arguments have to be four bi arguments with just the constrain of the system so you can pass ints and floats and pointers, but you can't throw in struts or characters or shorts, it'll confuse the system. You have to tell it how many arguments are expected of this particular function. We're gonna have scenarios where the thread functions don't need to take any arguments. We're gonna have a scenario like we have right here where sell tickets takes two of them. Beyond the two, you pass in the numbers that are of interest, so agent and num tickets divided buy num agents. Now, this does not actually prompt sell tickets to start executing. All it does is it lines it up at a gate.

Student:[Inaudible]

Instructor (Jerry Cain):The prototype of thread new just requires a thread name right here. We have to do it because it's [inaudible] but that's a lame answer. You really do want something available to you to identify a particular thread, that for instance, is failing you during the debugging process, and if you have 15 of these things, does that make sense? I'm sorry, you have 10 of these things and then one of them is falling or one of them is never exiting, which is actually a common thing we'll see in multi-threading. You want to be able to know which of the threads is not exiting so you can go and look at the particular implementation of that function. Okay.

Okay. So, procedurally, what happens up front, as we say, we're using threads and then you lay down gates one through 10, all of these things that are gonna follow, the sell tickets recipe, that's what they need to follow in order to run from the starting gate to the finish line and then this basically sounds the bell, fires the gun. Okay. This is a function block until all of these threads actually finish and run a completion and then when all 10 threads are finished, this returns and it passes on to what will be the end of the entire function.

Student: If I wanted to do the same process somewhere else, is this the int package and run all threads within the scope of the function?

Instructor (Jerry Cain):It actually does not have to be in main, it's just most conveniently put there.

Student:[Inaudible] as well when [inaudible] everything out there?

Instructor (Jerry Cain):What has to happen is that before you call run all threads you have to call the package just exactly once and you have to set up all the threads, whether it's directly in main or through sub functions to set up all of the dogs. Okay. Does that make sense? This actually fires the gun and tells all the threads to start running. It turns out that as threads execute, as part of their implementation, they can themselves call thread new. Does that make sense? The threads themselves can spawn their own child threads, okay, grandchildren threads, whatever you want to you. The ridiculous metaphor I have is that somehow while a dog is in the race it gives birth to three new babies and throws them back to the beginning of the gate, okay, and say's, "Please run," and sometimes there's interesting concurrency issues that can come up with that type of thing, but I'll actually get to that with a more advanced example probably Wednesday or Friday.

Student:[Inaudible]

Instructor (Jerry Cain):This just basically says now we're in thread mode. Okay. Once this has been called all threads that are ever created in the process, even if they're children threads, just aren't executing immediately. Okay. So as far as this is concerned, I want to just invent a function right here. If random chance, 0.1, I want to call a thread sleep function and I'll just pass in and say 1,000. Okay. That thread sleep basically says that as part of execution if a thread is running and it executes the thread sleep function

that it pulls itself off the processor for at least, in this case, a second, that number that passes as an argument is expressed in milliseconds. So this means that every time it flips a biased coin and it comes up heads with probability of 10 percent or .1, rather, it'll force it to halt. Now, that's not the only way a thread will halt, but this is a way for you to problematically tell a thread to stop running. Let's forget about thread sleep. I shouldn't have talked about that yet. What actually happens is that when you spawn off two or more threads, even technically one child thread, but two or more is when it's interesting, run all threads establishes something of a heartbeat.

In between the top of every single finger some different function, usually in a round robin fashion, but not necessarily, gets the time slice that exists in between the two finger taps. Does that make sense? So it's, like, agent one, agent two runs, agent three runs, agent four runs, in that round robin manner, okay, and however much progress they happen to make in that time slice is the progress they make. Now, if I don't introduce any randomization it's probably the case that on a real system it would execute the same exact number of assembly code instructions. Okay. Or very close to it so everyone would make exactly the same amount of partial progress with each time slice. Okay. Does that make sense? To make it a little bit more real world, we introduce some [inaudible] process here where things all of a sudden get a little bit random and maybe it's the case that ticket agent one sells two tickets and it's in his or her time slice and then gets pulled off the processor instead of actually being allowed to sell two more tickets. Maybe ticket agent two comes next and sells four tickets; maybe ticket agent three comes next and sells four tickets because this coin flip never comes up heads. Does that make sense to people? Yep.

Student:[Inaudible], which threads [inaudible]?

Instructor (Jerry Cain):Well, the one that's executed. The one that actually calls it. Okay. I mean, it's only called once, but the 10 dogs up there are actually each following this recipe, they each have their own little pointer in to be an assembly code that this compiled to. Okay. And if they happen to jump into this function right here then the thread that is actually is stuck inside that function is pulled off the processor and it's even pulled off what is called the Ready Q and put on this thing called the Blocked Q until this number of milliseconds, in terms of time, elapses. Does that make sense? Yep.

Student:[Inaudible] or run all threads, is there any control over how many [inaudible] cycles each one will get?

Instructor (Jerry Cain):Not in our system. Some very sophisticated thread libraries, I don't want to say very sophisticated, a lot of thread libraries, they don't always give you control over the amount of time that's in a time slice. They do want that to be somewhat regular because they don't want you to have unpredictable results, certainly not during the development process. Even though ours does not, some thread libraries, particularly the one in Java, everybody will be the most familiar with by the end of next year when you take 108, you learn about the thread library there. You can attach priorities. There's only three degrees of priorities in Java. I'm sorry, that's not true – there's 10 levels of

priorities in Java where you can assign a priority of one to 10 and then, usually, it's the case that they're all sorted and all the threads with priority 10 execute and run to completion before anything with priority 9 is given time. Okay. But we don't have any of that here. We really just want to think of all threads as equally likely to get the processor for a particular time slice in this round robin manner unless there are other things in place that actually block it from being able to make process.

Okay. So think about this line as basically really not blocking it all or blocking for an arbitrary amount of time. Okay. What I want you to imagine here is what type of print out you might actually get in response to this thread implementation. You might get three print Fs, agent one sells a ticket, it may happen like that. Maybe it sells three, maybe agent two comes next, maybe agent three sells five because that's how much time slice allows, maybe five is actually the most you'd actually see as the number of tickets that sold. But you understand what I'm getting at here. Does that make sense? It would just keep on cycling through all of them. Maybe it is the case after 130 or so lines that, for whatever reason, agent seven all done gets printed and then maybe it's the case that agent eight sells a ticket, agent eight all done and then eventually maybe it's the case, for whatever reason, agent four is the last one to sell a ticket and this is just representative of the type of output you might see from this. Now, there's nothing interesting about this from a simulation standpoint because there's really no compelling reason; from a performance stand point to use threading here except that you're trying to emulate the real world a little bit more. There are situations that you want to go with threading for performance reasons, this isn't one of them. I'm just trying to illustrate the thread package. Yep.

Student: If you didn't do the thread sleeve, would you see agent one sells a ticket, agent one sells a ticket, agent one sells a ticket or is it agent one sells a ticket, agent two sells a ticket?

Instructor (Jerry Cain):You would actually see – the thread library has no notion of what a wild loop is so it's not like it detects it, you jump back and uses that as a signal to pull it off the processor. Let's say that the typical time slice is 100 milliseconds, however many tickets can get sold in a 100 milliseconds is how many would be published. Sometimes it's gonna be partial, maybe you're gonna be halfway through the implementation of a call to print F, right, when it gets pulled off a processor and then when it gets the processor back it continues through the partial execution of print out to complete it, return and decrement the num tickets to sell count. Okay. Does that make sense?

Student: There's no way to [inaudible] two processes that you want to run parallel, but you want them to switch back and forth faster than 100 milliseconds?

Instructor (Jerry Cain):You can. Well, [inaudible] you certainly do not have that. I have to think that there's some thread packages out there that do allow you to control the time slice. It's usually not that high priority. Usually you don't introduce threading into a program to have control over the time slicing, you really just do it to have concurrency in

the first place, let the thread manager figure out which thread is gonna make the most progress. In an ideal world, we actually don't want to pull agent one off the processor at all if agents two through 10 are on extremely long phone conversations, and that doesn't happen here, but in a real simulation, you might want agent one to keep on selling tickets if all the others are blocked from something else. Okay. This is just in place to illustrate the thread new and the run all the threads and the initial [inaudible] concept. Okay. Yep.

Student:[Inaudible] doesn't [inaudible], it just sort of pulls that one off the Q and it keeps running?

Instructor (Jerry Cain):That's right. And you have to think that this is actually being called by 10 different threads in this set up. It's only written once. It's, basically, like 10 copies of the same book, but it's not even that. It' actually 10 copies of the same webpage and the webpage itself is hosted on one machine. Okay. Does that make sense? There's one copy of the code, 10 independent threads are following the same recipe. Okay.

Student: Well, and [inaudible] run threads directly after any initial thread package?

Instructor (Jerry Cain):Well, it has to be called – oh, I see what you're saying. In other words, to set this up and maybe call it right there –

Student: Yeah.

Instructor (Jerry Cain):– with the idea that these actually run immediately, I know what you're doing. In our system, it wouldn't work because this is a function blocks until all threads have been completed. So what would happen is you could it an int thread package, you would call one whole thread, but there wouldn't be any, so it would return and then it would go on and spawn these threads that aren't allowed to run because run all threads is actually a return.

Student: Okay.

Instructor (Jerry Cain):Okay. This is just idiomatic. Do this, set up at least one thread to run to make sure that all the work that needs to get done gets done, but in this concurrent manner as opposed to this sequential manner, and then call that just to fire the gun. Okay. Yep.

Student: What happens to all sell tickets as that whole thread contacts [inaudible] thread [inaudible] command?

Instructor (Jerry Cain):It's not inside a thread?

Student: Right.

Instructor (Jerry Cain):It could work. The way that the thread library works, the main thread is also a thread, so as part of as sequential execution, it's really not sequential. It

happens to still be in a thread, it just happens to be the main thread as opposed to one of these child threads that's spawned off by what was reachable for main. Now, I have to say I've never tested that because I've never given an assignment or done an example where I exercised the edges of the thread library, I've just kind of gone with the way it was designed to be just so I can make progress. But you could try it when the assignment goes out and see what happens with it. You'll never see a meaningful example from me that actually will realize that. Yep.

Student: You said the thread doesn't know if it's in a wild loop, does it know if it's in an instruction, like, does it know that [inaudible] and then the time went off, is it –

Instructor (Jerry Cain):Right. Right. That's certainly the most interesting part of today's lecture is that right now, you know enough about co-generation, I'm hoping, because you're gonna be tested on it in two days, what we say is that it's not an atomic operation. It looks like it's atomic because it's written in one statement right here, but what happens is that this really corresponds to probably what we would as a local variable, it would be a three-assembly code statement. Does that make sense to people? So it's gonna basically load num tickets to sell and do a registered decrement it by one and flush it back out. This is going to be a complexity that we start to solve in the last 20 minutes of lecture right here. So when it gets swapped off a processor, it could be right before the first instruction of the three that this compiles to. It could actually finish right after the third of the three, or it could be pulled off the processor 33 percent or 66.7 percent of the way through the code block that this compiles to. Does that make sense? You feel every single stall time in sequence. If you use threading, and this is the best example of all of the one that I'm gonna have, I think threading is really important. If you use threading and you spawn off 12 download from BBC server threads, all of them make enough progress, all of them try to open the connect and because that's considered a block at the kernel level, it's pulled off the processor. It's a much more harsh version of thread sleep. But it sleeps for a meaningful reason because it really can make a good progress and the thread realizes that and the thread manager realizes that so it pulls it off a processor while it's waiting for the connection to be established. Does that make sense? Well, imagine that all happening with 12 threads, all of those dead times that are associated with the network connection, they all align and overlap and pipeline in this way that really saves us a lot of time. Does that make sense? Okay. Yep. Go ahead.

Student: The last lecture you mentioned about downloading strings in class; I was thinking if you only have download capacity, you only have so much speed you can download, so what size files can you download in a certain amount of time, how can you download more than that?

Instructor (Jerry Cain):You're actually not. As far as the downloading is concerned, if you're dealing with a unit processor and you're dealing with one processor with one ram and one core, then you're dealing, primarily, only with the ability to index one article at a time as the text comes through. So you're right, you don't save time for the actual pulling of the text and parsing of it, and updating your hash sets, but you really save the time

with your network connections, and that's what the huge win is. Okay. Does that make sense? Okay.

So what I want to do here is I want to complicate this problem a little bit, but complicate it in a meaningful way. In a real world simulation, it might be the case that you have two ticket agents, and you have to sell 10 more tickets and if somebody's stuck on the phone because they want to buy the ticket or not, so the other ticket agent should be able to sell all nine or 10 tickets while the other is blocked with some time consuming customer. So what I'd rather do is rather than actual instructing each particular thread to sell a pre-determined number of tickets, I'd rather grant them all access to the same shared integer, the master variable, that stores the number of remaining tickets and do something like this; int agent and int star numb tickets and I'll put a P there. I'll close it off for the moment. I'll change this up here in a second. What I want to do is I want each agent to know what their badge number is, but I also want them to be able to go back to the main function and find the master copy of the number of variables that are remaining. This is basically the equivalent of the one master copy of your checking account balance. Every single ATM machine in the world is supposed to have atomic transactional access to. Does that make sense to people? Okay. Here is the main thread and its stack frame. All 10 other stack frames for the 10 other executing threads all have pointers to that one 150 inside, and that's how they kind of keep dibs on how many tickets there are remaining to sell. Okay. Does that make sense? Now, the problem and this is actually not even the full problem, but I'll simplify the problem to make it seem like it's easily solved, is that I, as ticket agent one, might come through and I might commit to that test and say, "Oh, wow, there is, in fact, one ticket left, that's greater than zero, so I'm gonna commit to selling it."

Make sense? And then boo-hoo, it gets swapped off the processor right after the curly [inaudible], but before anything associated with the num tickets minus minus. Okay. Make sense? So it gets swapped off the processor and thread number two comes in and executes the same test. "Oh, look, there's one ticket left, I'm gonna sell it," and it comes in and it commits to trying to sell it, but it gets swapped off the processor. Same thing for thread three, thread four, it could be this diabolical situation where everybody is really excited to sell the one remaining ticket. They don't go back and recheck the test after they get the processor back, that's not what is probability encoded, so they're all gonna try and decrement this shared global and so one, could, potentially, go down to negative nine. I don't think this is why airlines overbook flights, okay? But you can understand the type of concurrency problem that exists here. They're all depending on the same shared piece of data, and if they're not careful in the way they manage the shared data and if it's partway through the execution and it makes decisions based on information that will become immediately stale if its pulled off the processor then the integrity of the global data can actually be mucked with and be compromised. So, at the very least, we want this all the way through that to, more or less, be executed in full. Okay. So basically what that top bracket and what the bottom bracket does it kind of marks that thing right there, is what's called a critical region. It's, like, once I enter this region, no one else is supposed to be in here while I'm doing surgery on that global variable. Does that make sense? Now, there's nothing in the code that actually says, "Please other threads, don't come in

here because I am," there have to be some directives that are put in place to block out other threads. This is the situation where you're really glad that the bathroom door locks because if you're in there, you don't want them to have the privilege of just walking in because they're running in their own little thread. You actually have to have a directive in place, this thing called a lock, I'm gonna frame it as a binary lock, I think for obvious reasons, because you only want one person in the bathroom or in the critical region, right here, at any one moment. Okay.

So what I want to do is I want to talk about the most common concurrency tool that's in place to actually help delineate what is considered to be a critical region. It involves me introducing another variable type. I want to introduce something called a semaphore, and I'm gonna call it lock and I'm gonna set it equal to semaphore new. It takes two arguments; the first one I don't care about, the first one is gonna be some integer. Now, I'm just introducing semaphore like it's a word that you're all familiar with. I know you probably know what semaphore means in a general sense, but in a programming sense what a semaphore really functions as is non-negative integer, at least in our library it's considered to be a non-negative integer, that as a data type has functionality that supports atomic plus plus and atomic minus minus. This, basically, sets this glorified integer equal to one, okay, the minus minus and the plus plus against this lock, comes in the form of two different functions. There's a function called semaphore weight which, in this case, would take the lock variable. There's also another function called semaphore signal, which also takes the semaphore. Now, those are functions that, behind the scenes, emulate minus minus and plus plus, but they just figure out using special hardware or special instructions of the assembly code language to actually take the integer that's wrapped around by the semaphore, in this case, what's initially a one, and provide atomic minus minus. Okay. So, in other words, this right here would be decremented to zero if this were called against it. This would promote it back up to one. The reason that weight is the verb here is because we're gonna generalize a little bit. Think about the semaphore as tracking a resource. In this case, there's exactly one person allowed in the bathroom or there's one person allowed into the critical region, okay, which is why that's a one in the first place and you acquire that resource or you wait for that resource to be available and when you don't need it anymore, you signal it or you somehow release the lock. There's one key that I forgot to make – is that because the semaphore integers in our world are never allowed to go from non-negative to negative, there's a one special scenario that's handled by semaphore weight. If a semaphore weight is passed to semaphore, that at the moment, it analyzes it and is surrounding a zero, it doesn't decrement it to negative one; it's not allowed to do that. That's just the definition of what a semaphore is. If it detects that it's a zero, it actually does what is called block and it blocks on that semaphore.

It actually pulls itself off the processor because it knows that it's obviously waiting, presumably for some other thread to signal that thing before it could ever pass through that semaphore weight [inaudible]. Does that make sense to people? Okay. Basically, if I'm jiggling the door for the bathroom, like we always do at restaurants to wonder whether somebody is really in there or not, okay, you need, before you can really pass in there, you need someone else to release the lock, some other thread or some other agent in the form of a semaphore signal call before you really can go and open that door and

then you can look it yourself. What I want to do is I want to do is pass in three arguments to sell tickets. The reason I want to do that is because I want to tell the ticket agent what his or her idea is. I want to pass in the address of the shared resource, but I also want to pass in this thing I called lock. Now, the semaphore type is actually a pointer to an incomplete type. It's not copies, it's actually share some kind of strut behind the scenes that tracks the integer inside of it. And then the prototype of this would be the change to take a semaphore, I'll call lock, and this is the implementation I want to go with. I'm gonna simplify it a little bit. I'm gonna say while true, I'm gonna semaphore weight for the lock. As a thread, I have no business following that pointer and looking at its value and comparing it, using it in any sense, even comparing it to zero, because as I advance through the execution, I can't trust that that comparison is actually meaningful, if at any point during progression, it actually gets swapped off the processor and other threads can go and muck with that shared variable. Does that make sense to people? Okay. So what I want to do is I want to wait on the locked bathroom door and if I happen to be the one that first detects that it's unlocked and I can go in and, in this atomic manner, actually do a decrement. So as I detect that it's been promoted from zero to one, I actually take it from one down to zero and actually pass through this semaphore weight call, then I can do this. Num tickets P is double equal to zero then I want to break, otherwise, I want to do this – I want to print up that I have right here to say that I sold a ticket and then I want to semaphore the signal lock. The one thing I want to do here is that if, as a thread, I acquire the lock and I notice that there are no more tickets to be sold, when I break out I don't want to forever hold the lock on the bathroom. Okay. If you can programmatically unlock the door from afar you're no longer in the critical region, but you still somehow manage to unlock the bathroom door. Now, there's a couple of points I can make about this just to let it rest for you because this is probably where I'm gonna leave things until Wednesday. I initialize the semaphore to one up there.

That basically functions as a true. It basically says that the resource is available to exactly one thread and the first thread to get here actually does manage to, in an atomic way, take the one and do a minus minus on a down to zero because it actually committed to the minus minus, it returns – it executes this. It takes the zero back to a one one. It may come back around and take the one back down to a zero, but it's always, like, lock, unlock, lock, unlock, lock, and maybe it actually gets swapped off the processor right here. That would normally be dangerous except that it's leaving the semaphore in a state that it surrounds a zero. Okay. So there's some other threads that the processor and it certainly will then they come here and they, basically, are blocked by a zero semaphore. Does that make sense? Okay. Imagine a scenario where I accidentally – and this is actually the type of thing you have to be careful about because it's so easy to type a zero versus a one when you're typing a lot of them. If I do that right there, this creates a situation that you really have to be worried about when you're dealing with concurrency and threads, is that if I accidentally lock the bathroom door before anyone comes to the party, everybody's' gonna be blocked and no one is in a position to actually unlock it. At least not the way I've coded things up right here. Does that make sense? If I make the mistake of putting a zero up there, then every single thread will get this far and they're all gonna be thinking that someone else is gonna be [inaudible] that semaphore, so all 10 of them are pulled off the processor and everybody's just waiting. That isn't the case because of that one little

bug that I put up there. Okay. Make sense? If I have the opposite error and I do that right there, from a programmatic standpoint, if it's gonna be two, it might as well be 10. If you're gonna let two people in the bathroom why not let all 10? If you're gonna actually let two people go into the critical region and muck with global data at the same time, then you have the potential for having two threads deal with a shared global variable in a way that they really can't trust each other. Does that make sense to people?

Student: Yeah.

Instructor (Jerry Cain):Okay. So there's that. So the real answer here is that this, in this particular case, should be a one. Now, we will see situations where a zero is the right value. Okay. We will see situations where two or five or eight or 20 or 64 are the right values, but for this one scenario where I'm using a semaphore to basically limit access to what's clearly identified as a critical region, okay, that is the common pattern for using a semaphore. Okay. Question right there?

Student: Do you have two signal locks?

Instructor (Jerry Cain):Two signal locks, oh, this one right here? This is the one that actually is there whenever I actually do do a decrement. Because I can break out of the loop right here – if I break out of the loop, I circumvent this final call right here, but other threads may be blocks on the semaphore right here. All they need to do is to verify, as well, that there are no tickets left, but you still have to allow them to program as they get there so they as threads can also exit. Okay. Does that make sense? Okay. Yep.

Student: Is there something stronger than a semaphore that actually won't let the thread get pulled if you have something time sensitive?

Instructor (Jerry Cain):Actually, just priorities is really it. Even then, it's probably up to the thread manager as to whether or not – what would probably happen is a really sophisticated thread manager might actually know behind the scenes before it even grants the thread or processor, but there's only one thread with that priority. So it might actually have – and I don't know that this is the case and I'm just speaking in terms of implementation details – it might say, "Okay, that's the only one of that high priority, so unless we see a spawn of thread of equal priority or higher priority, we're just gonna let it run until it actually blocks itself," in which case, we don't have any choice. I don't know that many systems do that. It's technically possible to do it. Okay. So we'll have more examples come Wednesday, but I just wanted to make sure that you all got this. Have a good night.

[End of Audio]

Duration: 53 Minutes