

Instructor (Jerry Cain): Hello, welcome. I don't have any handouts for you today. You have plenty of handouts from Monday that we still have to spend the next few lectures on. You're not getting an assignment today, so you have this grace period from tonight at 10:00 p.m. until Friday, where you have no responsibility for 107 whatsoever. Remember, the exam is this evening at 7:00. It's in Hewlett 200, which is this huge auditorium in the building across the – beyond the fountain from gates. I'm going to send an email out after lecture, but just in case SUPD students are watching this before the TV exam tonight, I'm planning on posting the exam as a handout at 7:01 p.m. tonight, and then remote students just download it, self-administer, call in if they have questions and then fax it in when they're done. They don't need a proctor. I don't need any of that business. I just assume people are well suited to just sit by themselves and take an exam without somebody of authority hanging over their shoulder. And then, SUPD students actually have the option to take it tomorrow morning as well, and I actually prefer that SUPD students take it, because if there's a disaster during the exam tonight, people in the room can be dealt with immediately, where as it's very difficult to probably get that information outward. So, I actually prefer SUPD students to take it tomorrow and then fax it in sometime before 5:00 tomorrow so we can grade them.

Okay, I'm going to try to get the graded exams back to you and available by Sunday evening. I can't promise that. Okay, I actually have not dealt with a class this large in a long time, so we're dealing with – I know it looks like it's this cozy little family here, but it's not. It's actually 230 some people and it's been a while since I have had to manage a grading effort that involved that many people. It's also complicated by the fact that I am out of town this weekend, so my CAs are grading it and it might be difficult for them to get you the exams back by Sunday evening, but we'll do our best to make sure that that happens, okay? When I left you last time, I had focused specifically on the first multi-threaded example where we had to introduce this notion of a semaphore in order to control access to what we called a critical region. So, if you remember last time, the threaded function, the one it's the recipe that ten different dogs follow while they're trying to get their work done. It looked like this. Sell tickets, token INT, agent, it took an INT star called non-tickets and non-tickets P and then it also took this semaphore, I call lock. Just to review, since this was kind of a fleeting comment in the last ten minutes of Monday's lecture, the semaphore – it is more less like a – basically let's say a synchronized counter variable that is always greater than or equal to zero, okay? And so if I construct a semaphore around the number one, and I levy a semaphore weight call against the semaphore, this as a function figures out how to atomically reduce this one to a zero. Okay, and so this is basically equivalent to the minus minus but it does the minus minus in such a way that it actually fully commits to the demotion of the number to one that's one lower than it, okay? Semaphore signal on the same exact semaphore would actually bring this back up to a one, okay? If this were followed by a semaphore weight call followed by another semaphore weight call, then something more interesting happens, where this one right here decrements the one down to a zero. This one right here would have a very hard time.

Because semaphores at least in our library – this isn't the case in all systems, but our semaphores are not allowed to go negative. So, when you do a semaphore weight against a zero variable, then this thread actually says oh, I can't decrement that, at least not now. I need somebody else in another thread to actually plus plus this so that I can actually pass through a minus minus without making the number negative. Does that make sense to people? Okay, so programmatically, the implementation of semaphore weight is in touch with the thread library and so it actually when it's attached to a zero behind the scenes, it immediately says ok, I can't make any progress right now. It pulls itself off the processor. It records itself as something that's called blocked and it puts it in this cue of threads are not allowed to make progress until some other thread signals a semaphore they're waiting on, okay? Does that sit well with everybody? Okay, this is not constrained to go between one and zero. It can – this can be set to either be zero or one or five or ten. The only example we've seen so far is where the semaphore that's coming in is initialized to surround the one because we really want it to function not so much as one as we want it to function as a true, and it's basically a light switch that goes on and off, on and off, and on and off, and it's used. And, we use semaphore weight and semaphore signal against that semaphore to protect access to the non-tickets variable that we have been addressed to.

So, in a nutshell, while it's the case that true is true, I want this right here to be marked as a critical region. What that means is that I want to be able to do surgery on that non-tickets variable, without anyone else bothering it, okay? And the way you do that is to do a semaphore. I'll spell it out here. Semaphore weight on the lock, you do the check to see whether or not non-tickets of P is equal to zero and if so, you don't do the surgery. Okay, on the end, somebody else has done it a hundred times already and there is no reason to do it again. Otherwise, you want to go through and do this, that's true surgery on what functions as a global variable, at least from the perspective of all the threads running this and then you want to release the lock. You might do some printing here, okay? You might sleep for a little bit. Down here there was an extra semaphore signal call against the lock to accommodate the scenario where the person who breaks out of the wild loop does so after securing the lock, so they actually have to release the lock kind of as they go outside the bathroom window, as the analogy I used on Monday, okay?

This is considered to be – I'm sorry, this right here is considered to be a critical region. It's supposed to be something that when they're inside there, they cannot have any other threads during their time slices mucking around with this type of stuff, okay? As an arbitrary thread actually gets here, given that this thing was initialized to one, and because every single semaphore weight call is balanced by a semaphore signal, it's going to toggle up and down between zero and one. When a thread gets here, there is one of two scenarios. It's staring at a one and so it actually successfully does the minus minus and is allowed to pass in here and do the work or the thread blocks on this. You may say, well, how would that happen? A thread could potentially block on this if there's a zero, if another thread saw a one here decremented to a zero, made partial progress through this but the time slice ended before it got to the signal call. Does that make sense to people? Yes? No? Okay, just because a thread owns a lock doesn't mean that it can't be pulled off the processor. It might acquire the lock; get two-thirds of the way through this final

instruction here, okay? And then be pulled off the processor so that other threads can actually say, oh maybe I can make some progress, but if they get this far, they're still seeing a zero because of the thread that owns the lock hasn't released it yet.

Okay, so as other threads hit this semaphore weight call and it surrounds a zero, they are pulled off the processor. That is kind of what you want. If they can't do any meaningful work, you want the thread manager to say you can't do any meaningful work. I'm not going to let you even use your full time slice and eventually it'll get back to the only thread that can do work, which in particular would be the one that owns the lock right here, okay? Does that make sense? Okay, so there's that right there. Typically, try to keep the critical regions as small as possible. If you're going to lock down access to code, you don't want to make it arbitrarily long. That's basically like saying I want to do all the work in the world, so I'm just going to acquire a lock and I'm going to run this triple four loop. Okay, you only do that if you have to because it's a critical region. This print up in particular, if it's just logging information, it might not be imperative that you actually print to the console while you hold the lock, so you could release the lock and let other people make some progress, and then without holding the lock, just go ahead and print to this screen, okay? Does that make sense? Okay, there are a couple of other things. Somebody asked a very good question at the end of a lecture on Monday and I think I want to go over it. Some people were concerned with the case where non-tickets paid minus minus takes a one down to a zero, I don't mean the lock, I mean the actual number of tickets and they thought that was the one problem we were worrying about.

The answer is that's not the case and I can actually tell you a little bit more about what happens as threads get swapped off the processor and where all of their data gets stored, and show you that if the number of tickets is originally 100, there is as much of a race condition without the semaphore weight and semaphore signal calls in the minus minus bringing the 100 down to a 99, as there is in bringing a one down to a zero. So, this is what would happen and this is going to be the most important line to concern ourselves with. Forget about the fact that there are ten or 15 ticket agents. Just think about the scenario where there's two. Okay, it is subdivided and this is the main stack frame. Okay, it's the thing that sets up the two threads and caldronal threads. Let's say that this is ticket agent one and this is ticket agent two. Okay, these little like tornadoes are actually stack frames, okay, that each of the threads has. So, each of these things right here have their own activation records for their own call to sell tickets, okay? Declare someone a name is the number of tickets variable that will set equal to a hundred. That make sense? Each of these stack frames stores a pointer to that one hundred, okay? Imagine the scenario where the semaphore weight and semaphore signal are not there. This is how the two ticket agents could each sell one ticket even though it's only globally reported as a single sale, as opposed to the two tickets that were really sold. Coming down from here, la la la la, you see that it's not equal to zero, so you go in and try to sell a ticket. You know that this type of instruction actually expands to quite a number of assembly code instructions. Okay, does that make sense to people? So, in a local register set, this R1 may be set to .2, that right there, or two may be set to 100, because you do R2 is equal to that of R1.

Okay, then maybe you go in and you do a minus minus on R2, and bring it down to a 99, but now you're swapped off the processor. So, you've actually committed to the sale of a ticket, right? But, you're swapped off the processor. What happens is that the entire register state, all 32 registers, including the PC and the SP and the RV registers, if this is the binary state of all those registers, it's actually copied to the bottom of the thread that's being swapped out, little stack frame. Okay, and is that image that I just drew, that is used to restore the register set for that thread when it gets the processor back, okay? Embedded inside this image is the 99 that's going to be used to flush back to this space right here. Does that make sense? Okay, but this has just left the processor and so this thing does exactly the same thing with the register set as if it owns it. It sets R1 equal to that right there. The 100 still resides there because we didn't successfully flush back. We didn't get to the point where we actually update it to be the decremented value. So, R2 gets set equal to 100, gets set equal to a 99. This one is prepared to flush with a 99 back to the global space when this thing gets the processor back; it's going to flush a 99 back to the same space. So, this 99 is designed to override a 100. So, is this one, but unless you have semaphore weight and semaphore signal in place the way we do right here, one of the 99s is going to override a 100 and one of the 99s is going to overwrite the other thread's 99. Does that make sense to people the way I'm saying that?

Yes? No? You gotta nod your head, okay. If you put that there, and you put that there to balance it, and basically unlock the door, then only one thread is allowed to go through and actually pull the global value into the local register, decrement it locally, and then flush it back out to the global integer, the thing that functions as a global integer, because everybody has a pointer to the same integer. Before any of the thread is allowed to do any part of that, okay, so that is what I mean when I say that this is more or less committed to atomically, okay? Now, that is the overarching principle that is in place when you have threads, and in particular, you have threads accessing shared information, okay? This is the programmatic equivalent of the two Wells Fargo ATM machines where me and my best friend try to take out the last remaining \$100.00 in my account at the same time, thinking we're going to get \$200.00. Okay, make sense? Okay, if I were to initialize this semaphore to zero, then I would actually block all threads from entering this critical region right here. Okay, so I'd get deadlock. If I initialize the semaphore not to one but to two, that's as bad in principle as initializing it to ten, because you don't want any more than one thread in this region at any one time, okay. Does that sit well with everybody? Okay, good.

Okay, so let me move on and give you another example using threads and a different way to use semaphores. The handout actually uses global variables more than I like to, but this next example, I am going to use globals just so the code matches up a little bit more cleanly with the handout version. I actually like it better if you declare all of the shared variables in main and pass addresses to them, to the threads, because at least everything has a scope to it, whereas globals, it's a free for all and for two and a half quarters, we have been saying globals are awful. Oh, except for when it's convenient, okay, and I don't like that. But, this one next one, I want to frame it in terms of globals. I'm trying to model right now the Internet, where in all the world there's only server that serves up all the web pages and you have the only other computer with the only browser in the world.

Okay, I know you know enough about the HTML server process. You may not know all the mechanics at the low level, but fundamentally, you know that you request a web page from another server. It serves up text in the form of HTML normally. It could be XML, but normally it's HTML, and as the HTML comes over, it does a partial and eventually a full rendering of the HTML in your browser page. That make sense?

You know and you felt this before where a page is loaded like 70 percent but it's not quite done yet, you see the progress bar at the bottom, the bottom right, where it's like three-fourths of the way through and you know there's more to come. That's usually because the server has only delivered 75 percent of the content and so this thing has to block in much the same way that the threads up there block, this has to block and stop its rendering process until it gets more data from the server. Does that make sense? So, just use that as a guiding principle for this example. I'm going to insult all the Internet and I'm going to reduce it to a character buffer of size eight, okay? And what I want to do is I want to write a program that simulates the writing and the reading process and I'm just going to reduce the server to something that has to populate that buffer as a ring buffer. In other words, it's going to write 40 bytes of information, but it's going to cycle through the same array five to five times and I'm going to write another thread that consumes the information by cycling through the same array five times and digesting all the characters that are written there. Does that make sense to people? Okay, so the main function – I don't care about those. I have to emit all threads, I'm sorry, that's not right. It's the hybrid of two functions. I want to emit thread package of [inaudible] meaning I don't care about the debug information, and then I want to do this. I want to call thread new twice. I'm going to give them both names. This one is going to be called Writer. This one's going to be called Reader. Okay, I'm going to call the function writer and I'm going to call the function reader. I only have one instance of each one and neither one of them takes any arguments. Okay, it doesn't need to take arguments if you use global variables. Then, I do this. Run all threads. This is somewhat pathetically, but on the well intentioned – it is trying to emulate the fact that the server and the client as computers are both running at the same time.

Okay, programmatically I want the writer thread to cycle through and write 40 characters to the Internet. Okay, I want this reader thread to consume the 40 characters that are written in the Internet. Okay, this is what the writer function looks like at the moment. Four INT I is equal to zero, I less than 40, I plus plus, da da da, what I want to do is with these iteration, I want to call some function. I'll assume it's thread safe. Prepare random car and then I want to write to buffer of IMOG eight, whatever, let's give that variable a name, C. Whatever C happened to become bound to and so as an isolation function, I think you can look at this and understand that it's going to write down random characters in this loop over the buffer five times. Okay, I write this with hopes that it writes data down before the reader consumes it but it doesn't go so far that it clobbers data that has yet to be read. Does that make sense to people? Okay, let me write the reader, which has the same exact structure: I less than 40, I plus plus, there you have that. What I want to do is I want to basically do this and then I want to basically like, you know, process car, which I don't care about the details of what process does. This is the consumption line. This is the thing that takes a meaningful piece of data in shared space and brings it into

local space, so it kind of owns it. It can do whatever it wants to with it. Okay, let me draw the Internet. That was easy. There you have it. Now, you know, without concurrency, you know exactly how you want writer and reader to behave, so that everything is preserved and the data integrity is respected, and that the reader processes all the character data in the order that the writer writes it.

So, think about the scenario where the writer gets to run first and its first several time slices, it writes those three characters down. Okay, and internally it has a variable of *I* that is associated with that index, so that's where it'll carry on next time. Okay, but it writes those three variables down. And, then the reader gets a time slice and for whatever reason, process character is a little bit more time consuming. It actually has to like open a file or a network connection or whatever it has to do, just pretend that it actually is slower – its [inaudible] is slower, so it only really consumes that *A*. It doesn't really remove the *A*, but it just consumes it so it doesn't matter what's there anymore. Okay, so this is where the writer will pick up and this is where the reader gets swapped off, okay? I think it's pretty clear that if the writer is able to make more progress per time slice than the reader, then there's the danger that this might happen. And that on the very next iteration, it gets far enough in its time slice that it overwrites data that the reader has yet to deal with. Does that make sense? Okay, you can't have that obviously. Now, clearly, I'm simplifying things here, but the idea that someone is providing content and someone else is digesting it, that's not an unfamiliar one with large systems. It's also in theory possible. Just because I spawn off and set up writer to be the first thing that runs and this the second thing that runs, it might be the case that the reader gets the processor first. In which case, it will be digesting information that has never even been written or created. Does that make sense?

So, what I want to do. I want to create Internet so I can put some more global variables here. I have to make sure that the writer never gets so far ahead that it's clobbering data that has yet to be consumed. I have to also make sure that the reader never gets – never catches up or passes the writer and consumer information programmatically that isn't really there. Does that make sense? Okay, so what I could do is I could introduce two global integers and have semaphores that walk them down, but I'm actually going to use semaphores a little bit differently. I'm going to declare two semaphores here. I'm going to call one empty buffers and I'm going to call one full buffers. And, I'm going to let them actually manage integers that are always, almost always, but we're going to pretend always, are always in sync with the number of slots that can be written to and the number of slots that can be read from. Okay, I also want to enforce that the writer is also just a little bit ahead of the reader in terms of thread progress and that the reader can get and catch up to the writer, but it can't pass them, and that the writer can't get so far ahead of the reader that he actually is more than a cycle ahead of him. That make sense?

Okay, so what I want to do, I'm not going to do the semaphore new column. I'm just going to say that this is going to be initialized to eight as a semaphore. That is not the syntax floor but that's conceptually what I want to happen. Okay, I want to mention that up front that there are absolutely no full buffers whatsoever. Okay, make sense? I'm going to change this function right here to do this. Now, this is a slightly different pattern

with the semaphores, but I think it's really fun. Before I go ahead and write to this buffer, I better make sure that I'm allowed to do that, okay? What I'm going to do is I'm going to semaphore wait on empty buffers. Now, initially, empty buffers is equal to eight, which is consistent with the fact that we don't care if the writer makes a lot of initial progress. Okay, but if for whatever reason and the writer makes so much more progress than the reader that he gets really far ahead, this eight will have been demoted to a seven, to a six, to a four, to a two, to a one, and it really is just about the clobbered data that has yet to be consumed. It will be waiting on something that will have been demoted so many times that it's actually zero, okay? So, it will be a victim of its own aggression and it will be blocked out and be pulled off the processor, so that the reader can actually do some work. Okay, the balance here is a semaphore signal call but it's not against the same semaphore. After you write something down, you want to communicate to the reader that there is even one more piece of information that it's allowed to consume. So, I'm going to wait for something to be empty. I'm going to change from empty to full and I'm going to signal the full buffer semaphore, okay? The pattern over here is somewhat symmetric.

Let me rewrite it, is that I want to do the same thing, semaphore wait, but I want to wait for there to be a full buffer. When I know that there's at least one and I pass it that semaphore wait call, I can consume the character that is in global space and pull it down and then after I bring it to local space, I can immediately tell the writer that it's okay to write there if they're waiting, and then process can pass the C. Okay, there's that. Whoops, so it's like each thread has a little buzzer. Each of them are twittering each other as far as when they're allowed to proceed to read or write information. Does that make sense? This right here is sending a little buzzer that allows that to execute and return with much more likelihood. This right here is really communicating to the thread at that point and promoting full buffers so that the writer can actually write down more data, if it was previously blocked. Does that make sense? Okay, so think about what happens now. Empty buffers is eight, full buffers is zero. That means the writer has all of this free space to write to. It's going to have a very easy time passing through the semaphore wait call initially. Whole buffers is zero. The reader thread is bumming because the very first thing it has to do is semaphore wait on something that is set to zero. So, imagine the scenario where the reader actually gets the processor first. It's going to execute this much. It's going to declare I, it's going to consider to zero. It's going to pass the test. It's going to come down here and it's going to be immediately blocked from this line right here because it's going to be waiting on something that is in fact zero.

Okay, so the reader thread is actually being blocked right up front just like we want it to be. Okay, the other scenario is that the writer thread really fast and very efficient, it actually cycles through this thing eight times and then it hits a wall. Okay, so pair the character out before it actually went to bother on waiting on the lock, but – and it blocks here, it's because it's been a processor hog and it's actually done a lot of work whereas the reader hasn't really been able to do much at all. Okay, or at least comparatively. That make sense, people? Okay, so the ticket agents example where it uses a semaphore wait and a balance semaphore signal on exactly the same semaphore, and it brackets this thing called a critical region, that semaphore pattern or that semaphore is being used as a binary lock. Okay, binary meaning it's toggling between zero and one, true and false;

however we want to think about it. That's not the pattern that's being used here. We certainly have thread communication. We use the semaphore for rudimentary thread communication, okay, but right here what's happening is we're actually using these as basically two telephone calls. Okay, between the two threads, okay, this one calls this one whenever it can make more progress. This one calls this one whenever the writer can make more progress.

That is a pattern; it's what called a rendezvous pattern. Like I'm syncing up with you, that kind of thing, okay? There are more complicated examples of this. This is what's called binary rendezvous which really just – one thread to one thread communication. This basically says as this type of semaphore weight call means I cannot make any progress until some other thread makes some required amount of progress in order for me to move forward. This thing does the same thing on behalf of this semaphore. This says that I have to wait for some other thread that makes enough progress in order for me to pass, okay, or else the work I will be doing will be meaningless, okay? Make sense? Okay, so what I want to do is I want to just experiment. What happens if I make that a four? It doesn't change the correctness of the code or it depends on how you define correctness, but you will not get deadlock, okay? And you will not have any integrity data issues, all you're constraining is that the writer and reader stay within more of a delta of one another than they would have been able to otherwise. When it was an eight, it allowed some more degrees of freedom. It allowed the writer to go much further ahead if that's just the way thread scheduling worked out. When I made it a four, it just means that the writer can be no more than half of an Internet ahead of the reader, okay? Does that make sense? If I do that right there, I'm really pushing the limits of what's useful from a threading standpoint. If I'm going to do that, I also will just actually write the reading and writing in the same function and have it alternate between read and write, but if I really let these two threads run with those two initial values, all that's going to need – this is my W finger, this is my read finger. It just means it's going to run like this. Okay, does that make sense to people? And really if it tries to like run forward two slots, it'll be blocked by a semaphore weight call.

Okay? If I do this and I have a different form of deadlock, but deadlock is deadlock. I have a reader saying I can't do anything because I have no place to read from. The writer says well, I can't do anything because I have no place to write to. Okay, so you would have deadlock. You look at that and you say I would never do that, yes, you would. You just have, like when you're writing down all of the semaphore values, maybe you have like 20 semaphores in a real program, it's very easy for you to cut and paste a zero in place where you really wanted a one or a four or an eight, okay? So, if you have deadlock and you've never had that before, maybe you have because you've been in some wild true loop, but that's not the same thing. You really are making progress, you just don't see it. With threads, if you have deadlock, everything seemingly stops. You get nothing published to the console at all. It doesn't return. You don't get your command line prompt back, so things just expand and then you go okay, that probably means that two threads are waiting on each other.

Okay, or that nobody released a lock or something like that. Okay, if I do this, just think about whether that's damaging or not. You may think that initially [inaudible] should be more than full buffers. Let me do this. You could say well, I just want to kind of constrain full buffers plus empty buffers to always be eight. Okay? But if you do that, that actually allows the reader thread to get one hop ahead of the writer. Okay, so that's a kind of contrived example, but nonetheless that's exactly what it will be permitted to do. It doesn't mean it would actually happen, but it means programmatically it's possible. Another scenario is when I get this one right, but I do something like that. Okay, you may think that you're limiting things because you have semaphores in both directions but that thing has to be between one and eight for it to be programmatically correct. To put a 16 means that the writer is allowed to make two loops and take two tracks or two loops on the reader thread and that's not allowed. It's supposed to be at most eight slots ahead of, not 16 slots ahead of the reader, okay? Now, I had one and four and eight there before, my argument is that it should be the eight. Okay, if you have multiple options as to what you can initialize your semaphores to be, you always error on the side – although error is not the right word – you always kind of move toward the decision that grants the thread manager the most flexibility as to how he schedules threads, okay? And it also improves the likelihood that every single thread will be able to use all of its time slice. Okay, to the extent that you artificially constrain the threads, if you were to make that eight a one again and you get this again, it probably means that each thread is being hiccupped and pulled off a thread prematurely. I'm pulled off the processor prematurely. Does that make sense? Okay, and so you usually try to maximize throughput and you choose your semaphore values accordingly. Okay, does that sit well with everybody? Yeah?

Student:[Inaudible] semaphore weight?

Instructor (Jerry Cain):Which one is this? This is on the semaphore you mean?

Student:So, you called your semaphores, so now [inaudible] semaphore weight, so –

Instructor (Jerry Cain):That's actually not. You mean you call semaphore here before you wrap around and wait on empty buffers?

Student:[Inaudible]

Instructor (Jerry Cain):Well, this one never waits on full buffers. That one does.

Student:[Inaudible] signal before you call semaphore weight?

Instructor (Jerry Cain):Which semaphore weight are you talking about? Oh, I see what you're saying. In other words, if the reader doesn't agree with the processor, what happens here – is it the writer just happens to go first? It brings an eight down to a seven and it promotes a zero up to a one. But that's okay because it really is one slot ahead of where the reader is. The reader hasn't even started yet.

Okay, and so if this makes, let's say four full iterations and it brings empty buffers down up to four and full buffers down to four, that's fine, because if it gets swapped with the processor here, this thing just discovers a world that it's born into – it says wow, there are four characters I can read right away. Okay, and so it doesn't matter that it hasn't blocked – it hasn't called weight yet.

If it calls weight it only means weight if full buffers is zero. Otherwise, it just means decrement. Does that make sense? The words weight and signal are really, I think, were adopted with the binary lock metaphor in mind. I also hear when the thing is really a lock, I'll hear acquire and release as the verbs. Some versions of thread libraries actually define a lock type that things that lock acquire and lock release, which are really just the wrappers for semaphore weight and semaphore signal with the understanding that they're protecting ones and zeroes, okay? But it's not like this thing has to wait on that before this thing is allowed to signal it. Okay, sometimes you arrive at the bathroom and the door is open already. Okay, it just happens. Make sense to people?

As far as I have a single writer and I have a single reader, if you have – I won't write code for this but I'll just let you think about it. A lot of times when you are loading a web page, the HTML is often being sourced from multiple servers. Okay, maybe the primary HTML is being served from I don't know, facebook.com, but all the images actually reside on some other servers that are residing elsewhere, okay? And so all the information is pulled simultaneously.

Okay, imagine the scenario where you have not one of these things right here but you have three writers and they're all kind of competing to write and prepare the information that's actually sent over the wire to the single client. Well, then all of a sudden, you would have to – you would still want to use this empty buffers thing but then you would have to have an extra global, okay, or either that or something that is declared in main and shared with all the writers, so that they can all agree on what the next insertion point into the Internet is. Do you understand what I mean when I say that? And you have to have some kind of binary lock in place to make sure that no one, no two threads try to write to in this race condition matter to the same slot in the Internet or some piece of data will be lost, okay? Does that sit well with everybody? In the scenario where you have multiple readers consuming the information, maybe it's not really a web page but maybe it's just content that's being sent up from an FTP server and it's being handled by threads and that actually takes several different files simultaneously. You could use the same exact type of thing for the readers. Okay, does that make sense? Okay, so there's that. Let me work on one more example. This is a canonical problem that I think appears in all courses that teach concurrency. This next example is what's called the dining philosophy problem, where it's kind of a ludicrous setup, but nonetheless it is the setup. So, at the center of a table, I've drawn this picture 18 times now, there is at the center of the table you're looking down from the heavens at this table which has a plate of spaghetti at the center. Okay, and surrounding the table are five philosophers with their big brains, okay? And, in between each philosopher is a fork and every single philosopher follows the same formula every single day. They wake up, they think for a while and they eat and they think for a while and they eat and they think for a while and they eat, and of course they

think for a while before they go to bed. Okay? So, there is four think sessions [inaudible] by three eat sessions, but the interesting part and philosophers are actually confused by this idea that they actually need to grab both forks on either side of them in order to eat the spaghetti. So, they are these geniuses who have to eat like this, okay? If I model each of these philosophers as C threads, so each one follows the same think, eat, think, eat, think, eat, think pattern, it might be the case that they both decide to stop thinking.

Two neighboring philosophers decide to stop thinking at the same time and will eat but I can tell you right now that this one and this one will never be eating at the same time, because they both demand the fork in between them to be in his hand, in their hands before they actually eat spaghetti, okay? But let me not worry about that, let me just write the code and pretend that everything just works out and that will illustrate the deadlock problem. Okay, I want to model the forks as an array of semaphores and I'm going to write them down this way. This is shorthand. You actually would have to prepare them with five calls to semaphore new. But that just means that every single fork initially before they all wake up, there are five forks sitting on the table waiting to be grabbed by philosophers. Okay, and that's all I need. I'm actually going to write a thread, one philosopher, and each philosopher knows where he sits around the table, okay? And so the formula I want them to all follow is the following: four, oh, I can't use I there, I'm going to use ID. My first stab at this is this right here where they think for a while, just assume that that's some thread safe function that doesn't require any kind of interaction with other think calls, and then they want to eat but that isn't going to happen. There is actually a think call down there but that's not that important. What has to happen is that the four of the philosopher calls in his thread space, he needs to acquire fork I and fork I plus one, being sensitive of the fact that fork I plus one may be fork of zero is I is really high. Okay, does that make sense?

So, what I want to do is I want to semaphore weight on two different forks. Forks of I and forks of I plus one mod five, if a philosopher is able to pass through those two things then he just realizes brilliantly that he has two forks in his hands, so he is permitted to eat and call this function. He is a polite philosopher, so that when he is done eating, he is aggressive about putting the forks down by calling semaphore signal of I and I plus one mod five, and it is perfectly possible that if I spawn off five of these functions in main. I set up the forks full array of semaphores and it is four loop from I is equal to zero up to but not including five, I call thread new and I pass in zero, one, two, three and four, so they all have an ID number between zero and five and everyone has a unique number. It is possible for this to work. Okay, some philosophers think more enthusiastically about a problem before they eat. Some just really – they have like I don't know philosophy block and they actually – and they want to eat right away but this is technically programmatically encoded into this as a possibility. I am philosopher number zero and I thought for a while, but I would like to eat now. And so I do grab fork number zero and I'm holding fork number zero and then I get swapped off the processor. Okay, make sense? This as a resource is not available.

That first zero, first one is now decremented to a zero at the time that the thread is swapped off. Make sense? Maybe the processor gets, I'm sorry, philosopher, the second

philosopher gets the processor next, thinks for a while, gets this fork, okay, gets swapped off the processor. And then it happens again, and then it happens again, and then it happens again and then maybe this one wants to grab the right fork and it's like oh, can't do it because it's blocked. Okay, it's semaphore weighting on something that isn't available as a resource. So, it gets swapped off the processor, right? Maybe thread two, philosopher two says oh, I have the processor again, I'm going to grab the fork. Oh, I'm blocked. Okay, everybody is holding a right fork that is somebody else's left fork. So, when this slightly more obscure way, I think it's still pretty clear though, you have this mutual deadlock among all five threads because every single one of them is waiting on some resource held by the philosopher's to his left. Okay, does that make sense to people? So, you have deadlock. We can certainly overcome it. It's not a disastrous problem. There are multiple ways to solve it. You could actually alternate and have philosophers alternate between whether they grab the odd indexed fork first or the even indexed fork first, that could help solve things. But, I'm not going to take that approach. I'm going to take something else. Somebody had a question? Yeah?

Student:[Inaudible]

Instructor (Jerry Cain):Oh yeah, I'm sorry, it should be. This is just three iterations. Every single one of these should be IDs, that is correct. Sorry about that. So, I have to let you go in a minute, but I can really just communicate the problem I think pretty clearly. Remember earlier I said I wanted to do the minimum amount of work to prevent deadlock?

Okay, I want to implant the least amount of functionality to make something possible while still maximizing the amount of thread, amount of work that any particular thread can do during its time slice. Well, I could actually make this a critical region and that would be really rude because that would require that at most one philosopher can eat at any one moment, and that in itself is rude because you're not supposed to eat while others cannot.

Okay, but that isn't really the problem. What you could do is say you know what, I actually know that given that there are five forks and ten forks need to be held in order for everybody to eat, I can tell that at most two philosophers are going to be allowed to eat at any one time. Because if three philosophers are eating, that requires six forks to be held and we just don't have that many. Okay, we could also say that as long as we just prevent one philosopher from grabbing any forks whatsoever, it's technically possible to let four philosophers grab forks. Three of them may be blocked, but since I'm only allowing four philosophers to grab forks, exactly one of them will be able to grab two forks as opposed to one, okay? Does that make sense?

Now, two being allowed to eat, allowing four to try to eat, there are two different heuristics for solving the deadlock problem. I think both of them are very clear, are wise heuristics to go with. I will in the parting comments – I will tell you that I will at the handout take this approach. None allowed to eat, and that it's initialized to four. Okay, and the idea is that you have to be one of the graced four who is allowed to grab forks in

the first place and try and semaphore weight on them allowed to eat right there. Most of them will be able to pass through it immediately. The only philosopher that would be blocked on a call to semaphore weight on none allowed to eat right there will be the one who is the only one who hasn't tried to start eating yet. Does that make sense?

After you release the forks, you could call semaphore signal right there on none allowed to eat. You may think it's really weird to allow four to try but all I'm trying to do is remove the deadlock. Okay, and I technically will remove the deadlock if I limit the number of philosophers trying to simultaneously eat, to not be five, but to be four. Okay, does that sit well with everybody? Okay, I will go over the actual code for that on Friday, okay, and then I will give you some more intense examples than that. Okay, have a good night. I will –

[End of Audio]

Duration: 51 minutes