ProgrammingParadigms-Lecture17

Instructor (Jerry Cain):I have one handout for you today. I expected to have two but we had photocopier drama this morning so I haven't photocopied your assignment yet.

I'll post it as a handout after class today. You have plenty of time to do it. I'm not gonna make this due until two Monday's from now. So it's not like you're in any immediate rush to get started on it, so that's why I wasn't stressing about it. But nonetheless the assignment will formally go out today and you'll have two weekends to do it.

It's actually not too bad. It certainly involves some of the new threading stuff, you're revisiting Assignment 4. You're implanting some thread directives to make a lot of stuff that was running sequentially before and you felt the pain of that sequential execution before. Make it so that lots of things run in parallel and things run really, really quickly, really, really beautifully. It's very impressive, it's a good end result once you actually get that thing up and coded.

When I left you on Wednesday, you were all stressed because you had a midterm in seven hours. But I was just getting through the dining philosopher's example. Let me review the picture. The idea is that there is food, a big plate of spaghetti, at the center of a table and there are – I'll be less caricature about it this time – there are four philosophers – and I'll put P0 there – sitting in a circle, and there are actually what look like pitchforks but just regular forks in between each of them.

And each of those philosophers is designed to run in his own thread and they follow the recipe, that is, they think, they eat, they think, they eat, they think, they eat, and then they think again, and then they exit as a thread. That is their day. In fact, that's their entire life because they only live for one day, okay.

So I set this up as a global resource, because that's the way the handout deals with it. I will emphasize again that the way I'm writing this – this is shorthand, all I'm really saying here is this is a gesture to the fact that I have five global semaphores and an array of link five that are all initialized to one. There really are some semaphore new calls that are involved to build that thing up but I don't want to focus on that other than just mentioning it.

Each one of those ones basically represents the availability of a resource. If I call this Fork 0 and this is Fork 1 and Fork 2 and Fork 3 and Fork 4, when this particular semaphore is locked up in this synchronized manner at a 1, I know that this particular fork is available. And that is of interest to Philosopher 1 and Philosopher 2 because both of those have to contend for that fork in order to pass into the eating phase of the thread, okay.

So without concerning ourselves with the deadlock scenarios that was outlined last time, this is basically the thread I want them to follow. Each philosopher knows their subscript or their ID number and that influences which forks they grab, or they try to grab.

And so if – where did if come from? For N to I is equal to 0, I less than 3, I++, I want each philosopher to grab the fork to the right, fork to the left, that grabbing happens via two semaphore wait calls against forks of ID. I messed that up last time, I wrote I, forks of ID and then forks of ID + 1 but we want to modify 5 just so it wraps around. And let's – there we go, there, okay. And as long as they pass through those and return from those two semaphore wait calls in sequence they're allowed to eat. This is after they think for a while.

Okay, and after that happens, they eat for a while, they semaphore signal these things. I don't really care about the order in which they free them. It doesn't really impact execution in an interesting way. Forks of ID plus 1, mod 5 and that is more or less what they do. There's an isolated think after the last meal and then they return.

And I did my little choreography last time where in principal because each of these five threads may be running in round robin fashion, they probably will be, each one might get to the point where they grab the right fork. They're enough through this call that they've effectively acquired the fork because the 1 has gone done to the 0, but they get swapped out sometime between the actual demotion there and the time they actually demote the 1 to a 0 inside. Does that make sense?

So it's possible, I don't want to say it's unlikely, it's actually unlikely unless you force it happen but actually I don't want to say that. Because it's possible it's a problem. If all five threads could be swapped out right here and all be experiencing mutual deadlock because they're all depending on the philosopher to his or her left to be releasing the fork that is their right but your left fork, okay. Does that make sense?

Normally what you do, put all your semicolons in, but normally what you do is you implant the minimum amount of work to remove the possibility of deadlock. We made some observations in the final ten minutes of lecture on Monday – or Wednesday rather, where we know because of 5 div 2 is 2 that at most four forks can be held while philosophers are eating, okay.

So we could recognize that there are at most two philosophers gonna be able to eat at any one time so we could actually put something out of the generalized counter right there and right there and actually force each philosopher to become one of the blessed two that's allowed to go ahead and grab forks. Does that make sense to people, okay.

So what I did is I did this semaphore, a single semaphore, none allowed to eat, and I could initialize it to two. Let me get rid of these arrows. And I could sneak in a call right here where I say, "Please wait for me to be one of the two that is allowed to eat."

Okay, it's something of a critical region. It's not the same type of critical region we saw before. Critical region normally means at most one because they are concurrency – there are race condition possibilities. This is a different type of critical region. In fact, most people wouldn't call it a critical region but I'll call it that. But we only want two threads to be somewhere between here and this part right here, okay, because if we have any

more than that then we're nearing the scenario where we might have deadlock, okay. Does that make sense to people?

So let them go ahead grab forks, eat, release the two forks and when they're doing saying basically, "I have left the table." So that means I should signal all the other threads or all the other philosophers that might be waiting to – for permission to grab forks and do a final semaphore signal call right there, okay. Does that make sense?

Now, if you push two there I would completely understand why you did that. I personally, even though I recognize that there are at most two threads allowed to be in there, okay, or the way we've actually programmed it up that at most two philosophers will really be calling the eat function. My preference, just because I'm a purist about it, is that this be a 4 instead. Okay? Now a really terrible value would be 5. If I have 5 philosophers then basically the semaphore wait is just this gratuitous semaphore that everyone has to consume but there will always be one there.

The reason the 4 works is because as long as somebody's prevented from grabbing either fork then there's at least one philosopher thread that's capable of grabbing two forks. Maybe the other three or blocked, okay? But it's always the case that exactly – at least one philosopher will be able to grab two forks. Does that make sense to people?

That's the minimum amount of a fix that I need to implant into the code to make sure I don't have deadlock. So concurrency and multithreading purists usually like to do the minimum amount of work to prevent deadlock. There is a reason for that, because the minimum amount that you implant there – you remove the deadlock but you still grant the thread manager as much flexibility in how they schedule threads as possible, okay.

When I put a 2 there I'm taking more control over how threads are scheduled. That means up to three threads can block on this line right here as opposed to just one. Does that make sense? Okay, if you make this a 4 that means that up to four threads can make as much progress as is really possible before they're blocked and pulled off the processor, whereas if I make it a 2 we're blunting some threads prematurely, okay. Does that make sense?

Okay, so that's what I liked about that and this is why I prefer the 4. If you put 2, 3 or 4 there it will be programmatically correct. If you put 2 or 4 there I think I'd be correct from a design standpoint. I prefer the 4. Yeah?

Student: Are there – I mean, maybe this is a dumb question but why would we use a semaphore as opposed to, like, an if statement on the global variable? Is there any really good reason to?

Instructor (Jerry Cain):We could. In fact, I'm gonna talk about that specifically. You could – rather than doing this where you actually have the integer. Basically this tracks a resource. I like to think of this as the number of shopping carts that are available in front of Safeway, okay. And if there are five people that approach the store and it's a

requirement that all of them have a shopping cart, they all flash mob the four shopping carts but only four walk away with one. So one is blocked until at least one shopping cart comes out the exit door. Does that make sense?

I could – and I think this is actually a really good question so pay attention to my beautiful answer here. That I could have just put a global integer here and said it equaled to 4. I could have had this check to see whether or not it was greater than 0 and if so, acted on it, but then I have the separation between test and action that was problematic in the ticket agents example. Does that make sense?

Well, you could solve that by having an exposed global integer and a binary lock like we did for the ticket agent's example but then what happens – and you have to think about this. I don't really want to write any code because I don't think I have to.

Rather than just blocking on this semaphore and letting the thread manager decide when you can get the processor back, what you'd have to do is you'd have to do some little while loop, okay, around and repeatedly check to see whether or not the global variable went positive from 0 if you were blocked on it. Does that make sense?

You'd have to keep acquiring a lock and releasing it, acquiring the lock and releasing it because you can't check that variable unless you actually are the only one looking at it. And – I'm losing my train of thought here. Where'd it go? The problem with that from a programmatic standpoint, and this is I think is a pretty good point, is that if you're basically while looping you really aren't allowed to do anything meaningful until you get beyond the while loop.

So what's gonna happen is you're gonna be hogging the processor, maybe in the same time slice you're gonna just keep reacquiring and releasing the lock just to confirm that it's still 0. Does that make sense?

That's what's called busy waiting. There are some scenarios where busy waiting is actually fine. It usually is in the case where you have multiple processors and you expect some other thread running on another processor to release a resource pretty quickly. But in a single processor environment, which is what we're pretending we have, there's no reason for a thread to spinlock and keep checking the value of a global variable because it's not gonna change until somebody else gets it. Does that make sense?

So this right here is this very clean way to say, "You know what? I'm not making progress. Let the thread manager pull me off; put me on the blocked queue. Only when someone signals the semaphore that I'm blocked on will the thread that's blocked on this thing ever be considered for the processor."

The alterative is to use the global – exposed global with binary lock, programmatically correct but it encourages busy waiting and busy waiting is like the – probably the – I don't want to say the worst thing, the worst thing is having, like, a race condition exposed, but it's as far as correct coding is concerned it's the least good in terms of

design because it wastes processor time that could otherwise be shared with people who – for the threads that really can get the work done, okay. Does that make sense?

Two second question, four-minute answer. Okay, you guys are good with this right here, okay. There's that. I want to start talking about a couple other things. I have two related examples. One I'll actually write code for and then the other one I'll just talk about the design.

I want to start thinking about more practical problems. I'm gonna frame the next example in terms of just FTP downloads, okay. I know FTP is kind of this 1990's technology, but we all still use it, okay. We actually go and fetch files. We actually use programs that use FTP, but let me just assume the existence of this function.

I have this function and I'll call it download single file. And what I'll do is I'll give it two things. I'll give it a const Rstar. I call it a server and I'll give it a second name called path. I know you know what this means. Server is basically about the computer that's hosting the file that you're trying to get and path is basically relative to where the web directory or the FTP server is behind the scenes, how to get to the particular file of interest, okay.

So this is basically the computer it lives on and this is effectively the file name with directory structures, nested directory structures funneling down and drilling down to where the file lives on the server, okay. The return value here is the number of bytes that we're downloaded to pull the entire file over. Does that make sense?

Okay, what I want to do is I want to write this function called download all files. I want it to return the total number of bytes that were downloaded as basically the sum of all the sizes of the files that are downloaded. I'll call it download all files. I'm gonna assume all the files are hosted on the same server.

Okay, but I am going to give you – that's Rstar, the files an array of files on that server, and the number of files in that array. Now, two weeks ago – or a week ago for that matter, if I asked you to write this certainly on an exam you'd be dancing a jig because it's just a four loop, okay, with a plus equals going up and building up a return value.

But in spite of the fact that it's the same computer there's some problems with that. But just pretend that the server is capable of hosting as many simultaneously requests as possible, okay. Does that make sense? What you'd rather do is you're rather spawn off N different threads, okay. Does that make sense? N threads where each one is dedicated to pulling over one of these files, okay. Does that sit well with everybody?

Okay, so I'm gonna assume that run all threads has already been called and this is running as part of some thread that was spawned in the main function, okay. So I'm already dealing with a multithreaded environment. What I could do is I could declare something called total bytes and said it equaled to 0 and I can be prepared to return total bytes.

You're not necessarily sure how that's updated yet but you can be sure that each of the N threads that I'm gonna spawn off to download a single file is gonna somehow do a plus equals against this thing right here, okay. This is functioning somewhat as a num tickets in the very first example except we're adding to it instead of minus minusing from it, okay.

Now thread new doesn't return anything, okay. So what has to happen is that I have to spawn off N threads in addition to passing the server and one of the file names to the thread, okay, so that the thread can call this function.

I also want to pass the address of this integer right here, okay. Does that make sense? And a lock that's designed to protect access to this because you're gonna have several threads trying to plus equals potentially at the same time, okay.

So I'm gonna do this semaphore, I'll call it lock. I'm gonna set it equal to 1 and that's – this is just shorthand for what would really have to be there. And then I'm gonna do this, 4 int I is equal to 0, I less than N, I ++. What I'm gonna do is I'm gonna call thread new. I don't care about the debug name but I'm gonna call this function called download helper.

You may ask why I don't call this function directly. The function that's passed right here to thread new has had the void return type. But even if it didn't have a void return type there'd be no way to actually catch the return value from the thread new environment. Okay?

So what really has to happen is I have to call this proxy function that sits in between this one and that one that knows to call this one and also to accept its return value and do a plus equals against this thing right here, okay. I have to pass in a few arguments. I have to pass in server. I have to pass in files of I. I should pass in address of total bytes and I should pass in lock. That means I should pass in four parameters, okay. And that's all I want.

Now there is a problem with this, the setup already, but I'm just gonna implement it like this is the okay thing to do. But conceptually you know what's happening here. Basically this thread is being the typical manager at a company where he doesn't want to do anything except delegate, okay.

And this thread has the responsibility of pulling into these files. It happens to have N employees or N people it can hire on an instant like it just does right here. And it gives each one of them a task of downloading each of these things in succession. Does that make sense?

This download helper has to be a function that returns void. I'll call it DH for download helper. And it takes these arguments, const car star server const car star path int star, let's say, num bytes p, for pointer, and then it has this semaphore I'll call lock.

Now the semaphores, remember, are pointers themselves so they don't have to – you don't have to pass the ampersands there, you can just pass lock itself, okay. So you have a hook on the master copy of the integer that needs to be plus equaled against. What has to happen is that you want to do this, let's say, bytes downloaded and you wanted to clear that ahead of time because you want to do – actually, I don't have to do that. I'm sorry.

And you want to set it equal to the result of that function. Download single file where you pass in server and you pass in pass. It looks like it's being done sequentially but it's not. There are several there – there are N minus 1 of the threads trying to call this same exact function to download the files in parallel, okay. Does that make sense?

It's in the same spirit as the type of thing you have to do for Assignment 6, okay. The reason you catch the return value is because after you let this thing run and do its work, as a side effect of this function you're just supposed to assume it's on file and full appeared on your host machine.

But afterwards what you want to do is you want to semaphore wait on the lock and all the other threads are quite possibly waiting on because once you acquire that lock you are the blessed thread that's allowed to plus equals against the single integer that you have a pointer to.

So num bytes p plus equals bytes download. Then you go ahead and you release the lock and then you return. There's no explicit return value here. The thing that feels like a return value is actually a return value via reference via this point that you have a side effect of downloading the one file you're supposed to and you're also supposed to update this variable to be that much higher so that when the thing is returned it returns presumably because all threads have returned and contributed to this thing, okay.

So when this happens it really is the accumulation of all the bytes that have been downloaded, okay. Does that make sense? Okay. If I'm silly and I accidentally acquire the lock before I call download single file it'll still return but it'll be even slower than it would have been had you just not used threading at all and just done the download single files sequentially.

Because this just means if I put this up there and this is serving as a binary lock then you're putting the very important time consuming function inside the critical region so that at most one thread can be involved in the downloading of a file. Does that make sense?

So it's imperative that this be outside the critical region and we only have one critical region that it has to come after because it has to appear somewhere it has to come after you download the file, okay. Does that make sense? Okay, now there's one problem with this right here. Do you have a question?

Student: If you had, like, 10,000 files to download would it make all 10,000 threads at once?

Instructor (Jerry Cain):It will in principal it would. That's a scalability issue that I'm not concerning myself with. I'm assuming that we're dealing with something like 40, okay? Or even 100. Most thread systems, including our own, actually has a limit on the number of threads that can be reused. I'm sorry, that can be spawned off.

Our system is like 500. Some systems it's like 1000. And in principal really sophisticated systems can actually reclaim threads that have completed and reuse that thread space for something that's created afresh, okay. Does that make sense?

There is a little bit of a problem with this. I've framed this in a way that might not make sense to you but I just want to make sure you understand it is that I'm assuming that run all threads has already been called and this is a function, it's actually running in some of the thread.

And so what I really want to happen is this is a child thread of main to really be spawning grandchildren threads, okay? If run all threads is already – has already been called and this is running, as soon as this is called it actually sets this up for all – or all N of these up on the ready queue immediately so that they start running immediately, okay.

I used the analogy Monday, I think, of a dog that's already in the race giving to N dogs and throwing them back to the beginning of the race and letting them run, okay. The problem here is that the way this is coded up the job here is actually very easy. This isn't the equivalent of the manager of a company who's delegating all of his work basically going home before the work gets done. You understand what I mean?

It's one thing for you to delegate work and then to go home for lunch and then never come back. It's another thing for you to delegate the work but then to wait for all of it to be done even if you're just in your office surfing the Internet. You can actually just – you should hang out until all the work is done so that you can properly return the total bytes value, okay.

The way this is technically coded up at the moment it says, "Okay, I'm gonna declare a local variable. I'm gonna – yeah, yeah, yeah, I'm gonna share a lock with all of my underlings over here so that they all have atomic synchronized access to the shared global right here."

And I spawn them off and then I return immediately. It's quite possible that I would return a 0 here because I may advance from the very last iteration of the for loop to this return statement. Okay? Does that make sense? Before any one of these threads makes any partial progress whatsoever.

Presumably this is a time consuming function, like, on the order of, like, milliseconds or even seconds. It would take milliseconds for it to advance from the very last thread to this one right here. Okay? It's quite possible that maybe even one or two time slices all of these threads could be spawned off and it could return before that makes any work whatsoever.

So what really has to happen is right there I really need the manager thread, the download all files thread, to really block and not go anywhere and certainly not advance to the return statement until it has verification that all of these guys have completed. Because if they've completed then the manager knows that this thing has really been built up to have the true return value. Does that make sense to people? Yes? No?

Okay, so what do you do? Well, not surprisingly you use concurrency and you use semaphores. So basically implant thread communication, thread-to-thread communication. The reader-writer example we had the reader and the writer communicating in this binary rendezvous way. When I do this I'm talking about the crisscross of semaphore signal and semaphore wait calls so that each one can tell the other one that there was an empty buffer or a full buffer, okay.

There was this one to one relationship between reader and writer there. I really have a 1 to N relationship with this setup. I have a single master thread right here that's supposed to do all the work. It elected to spawn off N threads to get the jobs done because it can take advantage of parallel computing right here, okay, and download many of the files simultaneously.

What really has to happen is I need about six inches more of board space. What I want to do is I want to declare a semaphore up here and I'll say a children done and I'm gonna set it equal to 0. I'll do the same thing there just to make – there really is a semaphore new call.

What I want to do is I want to use this children done semaphore basically as this connection to each of the threads that it's spawning off. I wanted to do this for hence I is equal to 0, I less than N, I ++. I wanted to semaphore wait on children done.

Now I haven't passed children done to that thing yet, but I will in a second. What I want to do instead is to change this to a 5. I want to pass down children done as an extra parameter. I have to abbreviate because I'm out of room there, okay?

So what I'm basically doing is I'm giving, like, it's almost like a little baby monitor to each of the threads, okay, that I'm spawning off. And when each one of them is done they go, "Wah," into the baby monitor in the form of a semaphore signal and when I hear N of those, okay, I know that all of the threads have completed. Does that make sense to people?

So the signature for this has to move over a little bit, semaphore, I'll give it a different name here. I'll call it parent signal and then before I return over here I will semaphore signal the parent to signal. This is the "wah" into the baby monitor and this thing is actually aggressively for looping, okay, and a semaphore waits on this thing not once, but N times, once for each of the threads it's spawned off.

Programmatically each thread signals this thing exactly one time. So I expect this thing to be signaled exactly N times. I need all N of those signals in order for this thread to know that it's done. Does that sit well with everybody?

And then once I have that I can advance to the return statement knowing that it's safe to return total bytes there because all of the N threads I gave birth to have actually done their work and died. But as a side effect their legacy was to plus equals my total bytes integer, okay.

Yeah.

Student: Is there any way that there's a way – it seems like there's a way to increase the number of N semaphore and is there a way to decrease it [inaudible]?

Instructor (Jerry Cain):Well, semaphore wait certainly does decrease it. If this is surrounding a 7, semaphore wait brings it to a 6, okay. So semaphore wait is like a minus minus that's guaranteed to be atomic and it's also a block if the number inside happens to be 0. Semaphore signal is an atomic plus plus and not much more than that. Okay?

Student: So you could initiate it to N semaphore children have not done, I guess? And then [inaudible] until the lock is –

Instructor (Jerry Cain):Right, but I have the – our version of the semaphore, some more recent libraries have decided to do this even though this is an argument against it, you'll notice – and this is mentioned in the handout. We don't provide a get value within method on the semaphore. We could ask for the value of a semaphore, okay. And we could say, "Oh, what is the value now?" And it tells you 7. And you go, "Oh, I'm gonna act on that 7." Right?

But it may have changed by the time you get a chance to act on it because in theory if you have a lot of threads the time between the return of get value within and the code that acts on that value could be separated by seconds, okay. Exaggerate it. Think about it in terms of years, which is what it really effectively feels like at the processor level, okay.

A lot can happen in years, okay. So you don't necessary – you can't trust a value that was acquired from within the semaphore and act on it until you've actually released a lock on the check, okay. Does that make sense? Now actually getting a 0, there are some situations where it would be okay. You could keep on looping and only break out once you get the value out that's a 0, but that's a busy waiting thing that I was arguing against when I answered her question earlier. Does that make sense?

Technically this is a little bit of busy waiting but it makes as progress as is possible until it blocks because not enough children threads have completed. There are some versions of semaphores. The one I'm thinking about are the ones that come with the 1.5 version and later of Java, which you'll learn all about in the autumn when you take 108.

Do you notice the semaphore wait right here, it's an implicit request to just do a minus minus; a minus equals 1, right? There are some flavors, not in our library but some more modern libraries where you can actually for a total of, like, N or 12 or 20 or 3 dozen or whatever, decrements against the semaphore and just call this once as opposed to exposing the for loop as an internal for loop instead. Does that make sense?

Okay, ours it's exposed. I think that's fine at this stage of the game because I want you to understand the mechanics of what's involved in order for this thread to block, okay. And it's not technically busy waiting so it's not really a bad design, okay. Make sense? Okay, question.

Student: Yeah, there is nothing to prevent all the threads from downloading into the same portion of –

Instructor (Jerry Cain):I didn't – not the same portion of the file. My assumption is that each of these files is actually different. And I'm just assuming – I didn't mention it explicitly but I'm assuming the download single file is the thread save function that doesn't actually interact with other calls of the same function at the same time, okay, which is technically not true because two download single files may have to create at the same directory on the host machine, and that would be a little bit problematic except that the make directory command is implicitly atomic on the operating system anyway.

I'm sure it is. I've never read that but I'm sure it is, okay. You got the setup here? Yep.

Student: Is there any reason not to, like, in [inaudible] when you secrement it there and just semaphore wait once at the end?

Instructor (Jerry Cain):But the thing is – if you semaphore wait you might wait on – you want it to semaphore wait after it's become 0. There's no way to do that. Like, semaphore wait will succeed if it's positive and you're making it initially positive. All you're gonna do is make it N + 1, okay, or take it from 0 to 1 at the end. But it's, like, you actually want it to – the only way you can block if it's semaphore waiting on a value of 0 and you really do want this thing to halt, okay. Does that make sense? Okay.

Some version of the library, and actually we can't initialize – even though a semaphore in our world can never go from 0 to a negative number, some versions – the Java library does this as well, allows you to initialize a semaphore with a negative number, okay. That seems a little weird but if I were to initialize this semaphore to be negative N + 1, and then I had a single semaphore wait call here, it would have to be signaled N times before it became positive. Does that make sense?

So that's what I think the two of you are trying to get at. It's just not supported by our version of the library or our particular thread library. Yep?

Student: So you can [inaudible] the function together you make it one of the semaphores like with positive have a [inaudible] of counters and some of the counters at the end?

Instructor (Jerry Cain):I'm sorry, what? Oh, I see what you're saying. So in other words you're thinking, like, some how do a –

Student:[Inaudible] or is it there's an each [inaudible] that's in reference to one [inaudible] so you don't really need to have a global counter.

Instructor (Jerry Cain):I'm still – I'm not quite understanding. I think I know what you're saying. Are you just asking this thing to pull, like, an array of Booleans or something like that?

Student: That's [inaudible].

Instructor (Jerry Cain):Into an array, yeah.

Student: Of elements?

Instructor (Jerry Cain):Right.

Student: Each?

Instructor (Jerry Cain):I see what you're saying, yes. I have seen that before. What he is suggesting, I think it's pretty clever although I don't – I think it's clever but I think it's just a little bit more work than it needs to be. He's setting aside not just one integer but an array of length N where each of these things can write without fear of race conditions to a slot in the array that's dedicated to that thread.

And then once you get past this point you'd still need the children's done semaphore, you can go through and safely do all the addition yourself there. Right? That's actually a fine idea. I actually – I can't even say that it's a bad thing. In many ways it's actually pretty good. The only thing about it that it might be problematic is that, I mean, the number N here is huge. But even that's not that big of a deal. So that's a great idea. Yep?

Student: Reference you back to the question two questions ago, if you made children done equals N and you replace the for loop with while children done does not equal 0, would it work then and just like a closed while loop?

Instructor (Jerry Cain):Well, you can't do equals 0 on child's done. I'm writing this as shorthand but it's not really an exposed integer. This one doesn't actually – if the semaphore did provide a get value of method or function there are a lot of scenarios where that method – that type of functionality breaks down. But this would be one where it would work because it's only gonna hit 0 once.

So that is true but it's interesting to spoil that. If you want, like, after you've done this Assignment 6, you should go – you already know enough Java to digest the syntax, just go and read, like, the two or three pages of the concurrency model in Java 1.5. You're

gonna read it anyway if you take 108 in the autumn, just so you understand all the things that are available and more sophisticated for our libraries, okay.

I just wanted to minimilize I wanted to. Basically we've used this package for so long because it really is lean and clean and very easy to digest in full. So you have to do all the interesting things in terms of the atomics, okay. You guys doing okay?

Okay, I do want to make it clear that when I initialized this to 0 I may get down here and I may spawn all these things off and in theory semaphore wait against a 0 and block immediately. Does that make sense? I may actually get swapped off right after this closed curly brace but not get inside the for loop immediately. I could get swapped with the processor as this download all files thread.

It might be the case that 50 percent of these things actually complete and promote this from say 0 to 6 because there were 12 child threads, okay. And this thing would come down right here and it would happen to notice – or not notice but it would benefit from the fact that this thing had been promoted 6 times. So it would make 6 immediate for loop iterations – or carry through all 6 of the iterations immediately before it's blocked on the seventh.

It's not a problem, okay. It's technically possible for the download all files thread to exit before the download helper thread does. I'm sorry, before any of the 12 download helper threads exit. Now you would think that'd be a problem but this is the scenario. This one blocks the 12 different threads that I spawned off all get this far and they get swapped off just after they've returned from semaphore signal here but before they actually call the return instruction to emulate the closing curly brace. Do you understand what I mean when I say that?

So all swapped threads could, like, right before here be swapped off the processor. And this thing could get the time slice after that's happened; get the processor back and say, "Oh, wow. Look, I actually went through this thing 12 times and I can return an I do." Okay?

You're not – I mean, if you want to be as completely realistic about the emulation as possible you can do it. But you have to effectively understand that this function is really over when SS – semaphore signal returns. And that to the extent that you needed the information to flow and be synchronized the way it is it still works, okay. Does that make sense?

In Java you actually have the option of releasing the lock after the closed curly brace. There's a key word in that language that doesn't exist in C and C++ called synchronized. You'll become very familiar with it next autumn, that releases the lock after you've formally finished the method. And I say formally finished after you've gone past the closed curly brace, okay.

But I don't care in this situation because all I really use the semaphore for was to get a meaningful full total bytes value. Make sense? Okay, so that's – oh, go ahead.

Student:[Inaudible] blocks before the [inaudible], right?

Instructor (Jerry Cain):This one right here? Yeah. It's a one line body of a for loop, so just this is right here and like that. Is that what you're asking?

Student: Yeah.

Instructor (Jerry Cain):Okay, very good. Okay, so what I want to do is I want to set up a big example for Monday, okay. All of the examples have been focused primarily on either one or maybe two synchronization directives. The handout that I have you out today – gave out today – has this great first 2 or 3 pages that was written ten years ago but it hasn't changed. It hasn't changed in 50 years, really, much less the last ten years.

It describes all of the different patterns that you have to be sensitive to or situations you have to be sensitive to and the patterns you use to be sensitive to actually make sure that you don't have race conditions and you avoid deadlock. It's all about protecting global variables in a binary lock – or a critical regions situation and implanting directives for two threads to communicate with one other – two or more threads to communicate with one other, okay.

And so it's good stuff to read. I've hit on all of it in the past three days – or past three lectures rather. The large example and it's kind of – it's a little intimidating. It's actually larger than any exam problem. It's probably the merging of two past final exam questions from, like, ten years ago. It's a great problem, okay?

It is the ice cream store simulation and all I'll do is I'll set it up so that you know what all the players will be in this big grandiose ice cream store, okay. We're going to have ten customers, okay. There's going to be a customer function. We're gonna spawn off ten of those things and we're gonna actually know how many ice cream cones each of the customers order. It's gonna be between 1 and 4, okay?

There's gonna be a cashier that each of the ten customers approaches as they're ready to pay. So there's a little bit of contention and I think there's already clearly a flavor of currency and race condition going on because these ten customers have to approach this cashier and the best metaphor I can think of is when your mom, when you were going with her a little kid to the market she'd have to go to the deli and she'd have to potentially edge out and get atomic access to that little number thing.

She pulled 33 or the 94 or whatever it was, and you always looked at the number and it was like 22 away and you go, "Oh, my God, we're gonna be here forever." Okay? But there certainly is gonna be thread-to-thread communication between all the customers, okay, and this one cashier. Make sense?

There's gonna be a single manager thread – I'm being very cynical about managers but this manager doesn't do very much either. What he does is he approves or disapproves of ice cream cones, okay. He doesn't make them himself, he just approves and disapproves of them with a – like a coin flip is what really happens.

They're gonna be between 10 and 40 clerks. The reason it's 10 and 40 is because every single customer is going to give birth to a clerk thread – one clerk for every single ice cream cone that needs to be made. So the people who are ordering, they're really impatient, this is real life. They want – they have four ice cream cones to be made, they want all four to be made in parallel, okay.

So the customer never sees the manager but these ten customers interact with 10 to 40 clerks, order the ice cream cones, they accept the money, make the ice cream cones, but all the clerks interact with the manager. They have to acquire a lock on the manager's time, okay, because the manager is overwhelmed if more than two people are in his office at any one moment.

So he can only accept one person in the office with one ice cream cone so there's no confusion as to which ice cream cone he's approving or disapproving of, okay. Does that make sense? So we're gonna run a simulation where there are up to 52 threads running simultaneously and there's certainly gonna be 4 different types of threads, okay.

I want the manager to basically arrive at the store and not to leave the store until he's approved the number of ice cream cones that he needs to approve. I want each thread – the clerk, to only – to live as long as is necessary to make a good ice cream cone and hand it over to the customer. I want the customers to be able to order their ice cream cone, get it, go and pay for it, and leave the store. I want the cashier to know that he's rung up ten different customers. Does that make sense?

Okay, so to the extent that you can read the problem statement between now and Monday, that's great, although don't worry if you can't because I'll certainly review this at the beginning of Monday.

But even though parts of the simulation are certainly a little contrived, they're contrived specifically to bring out some scenario where you have to deal with concurrency properly, okay, and use semaphores to either foster thread communication or to manage binary lock or access to a resource, okay. You have a good weekend. I will see you next week.

[End of Audio]

Duration: 49 minutes