Module 24 Week 13 Thread Concurrency and Deadlocks

Concurrency and Deadlocks 1.2

In this model we're going to continue our discussion of threads and we're going to talk about some of the features of having threads work together with one another. So we're going to talk just simply a quick reminder about what threads are we're going to talk about some features of having multiple threads accessing multiple resources. We're going to have a discussion on ways that we can end up with a synchrony and what a synchrony is. We're going to talk about the idea of critical sections. We're going to talk about concurrency issues. We're going to explain mutual exclusion and talk about some software and hardware options or solutions for mutual exclusion. We're going to describe the operating system semaphores and how they work. We're going to describe deadlocks and how we can resolve deadlocks and then we're going to talk about the dining philosophers problem as a wrap up problem. So there's a lot to go through in this module and we'll get started immediately.

Reminder about Threads 1.3

So a quick reminder threads all share the resources of the process, which means that they have access to all the same memory access to all the files access to everything that the process has access to and each of the individual threads work as a separate program but they're all working towards the same goal. So, if the process that we're talking about is Microsoft Word the ultimate goal is to create a document and print it, save it, do something with it but the problem that runs is that threads can run asynchronously. The problem that we have is that these threads are going to do work and that the work is not done in a synchronous manner in other words we don't have one thread running to completion and then another thread starting up we have all the threads running at the same time and if we have multiple threads running at the same time accessing the same resources we have the potential for conflict on those resources and we have to figure out ways to deal with the conflict and address the potential for catastrophic loss of information.

Features of having Multiple Threads 1.4

So one of the nice features of having multiple threads of course is that they can communicate with one another. Very easily we can have a lot of threads created and all of the threads are accessing the same memory in the same resources. So, if we just for example created a buffer in main memory and had all the threads have access to the buffer then we wouldn't need the operating system to intervene as we would if we had multiple processes. So, multiple processes can only communicate with each other via IPC via the interprocess communication system, but multiple threads can communicate with one another directly because they're both accessing or they're all accessing the same amount of memory.
So this is a really effective solution. For example when we're reading in data from an IO device we could have two threads one that processes data that's already been read in and another thread that's actually doing the read operation. So when the thread that's doing the read operation performs a blocking system call to actually bring in the data from the secondary storage device the operating system blocks the thread, but the other thread that's processing the data is allowed to continue operating.
So, we have the ability to do a read in operation while we're doing some processing on data that's already been read in. The other nice feature about having a thread is that it's relatively easy to create. It takes a very very little memory and we can create it very quickly and easily as compared to a process but the big downside of having multiple threads is that we run the risk of asynchrony we run the risk of these threads doing something out of a normal order and we're going to see that in this module.

Asynchrony 1.5

So what is asynchrony? Well asynchrony occurs when two threads are running seemingly at the same time. So for example we might have a running thread that is interrupted due to some hardware considerations maybe the network card needs processing it doesn't really matter and a different thread is chosen to run. So, we have one thread that's running on the CPU and it stops running. It does some work and it stops running and then when we come back we're choosing a different thread to run. We could also have a thread that runs out of time. Remember we're in a preemptive system. So we're not allowing threads to run until they're finished we're not allowing processes to run until they're finished. We're allowing them to run only for a certain period of time and when that period of time runs out whether or not the thread is finished we have to pause the thread we have to bring the thread out of the running state bring it back into the ready state and then we can go on to running either this thread or we can choose a different thread to run. So, there's a possibility of asynchrony there and the other possibilities if we have actually multiple CPU's multiple different CPU's we can have two of these threads running on the CPU all at the same time. So, what's happening here is that these threads are doing some work and the work that they're doing it is not a complete atomic instruction in other words we don't start it and know when we finish it. We don't do one function call and finish the function call and then pause to run another thread. We could be in the very middle of an instruction or in the very middle of a line of code and we could stop and run a different thread altogether so we always have to keep that in the back of our minds when working with threads that we never really know when the thread is going to be stopped and when a different thread and possibly a conflict thread would be run.

Critical Sections 1.6

So sometimes code is going to be written with the expectation that once we start running a particular piece of code it's going to run through with to completion without being interrupted and we know from our discussion just a minute ago that that's a fallacy we know that. It's not going to be run as an atomic element. It's not going to be run from start to finish without interruption that interrupts are going to happen in the system and that we're going to stop this thread and started up again at almost random intervals and in fact I would say that we would consider the worst case scenario and the worst case scenario is where we have a real bad luck situation where we run some piece of code stop at a really bad place and choose a different piece of code that conflicts with exactly the first piece of code. So the code is going to be interrupted and we know that asynchrony can occur during interruption. So we have to take that into account. What we do is we identify those pieces of code that have to be run atomically In other words we identify those pieces of code that once we start doing this code we can't be interrupted. Now unfortunately we can never say that we can't be interrupted.

So the better solution is to say that we if we are interrupted that no other thread is allowed to go into that piece of code that could conflict with us and what we're trying to do is identify something known as a critical section and a critical section is a piece of code that once entered you have to prohibit all other threads from entering the critical section on the same resource. If we have two different resources and two different critical sections. We can run both those critical sections at the same time because the two different resources don't conflict with one another but if we have a critical section that operates on a particular resource then we can't run one critical section in conjunction with another critical section or else we have asynchrony. So, if we reach a critical section in thread one for example then if thread one is interrupted thread two can't enter its critical section until thread one is allowed to complete. So, if thread two wants to enter it's critical section it has to be blocked it has to be paused. The critical section

should be as small as is possible because we don't want to spend a lot of time blocking all the other processes from accessing sections of code, their critical sections if we don't have to. So, we want this these pieces of code to be absolutely as small as possible but we do recognize that they exist and that they'll have to be there. So, there will be situations in which we block other threads from running because we don't want to have the possibility that those threads are going to interfere with what we're doing and once we're complete with that critical section we'll allow all the other threads to continue running.

Supplier/Demand Explanation 1.7

So the code that I'm going to show you right now is what we call a supplier demander example and the supplier demander example really just has multiple threads running of either a supplier thread or demander thread. In fact we're going to have multiple instances of both suppliers and demanders. So, think about a factory for example where things are coming off the production line there's a lot of people producing things, there's a lot of people taking those things and moving them on to the next stage in the production. So, we have both supplier and demander one of the important considerations here and why this has a critical section is that there is a shared variable there what we're calling the buffer and the buffer count is actually two shared variables there. The buffer count tells us how many items there really are in the buffer. Now you recognize from the code that the buffer can store five hundred items. The buffer count let's say it's initialized at zero there's nothing in the buffer when the morning starts the suppliers come in and they start producing things that go into the buffer. Now the supplier just has a simple piece of code that says as long as I'm not finished then I'll put something in the buffer. Now the reason that we have that while not done is because it's possible that the buffer is full and if the buffer is full we want this thread to just kind of pause, pause for maybe half a second and let one of the demanders take something out of the buffer so it makes space and then we can put this thing in the buffer. We can update the buffer count and then we can be finished. So the suppliers a very simple piece of code the demander is almost the same thing except it checks to see that there's actually something in the buffer.

So the while not done here is for checking to make sure that there actually is something in the buffer rather than making sure that the buffer is full as long as there is something in the buffer we'll decrement the buffer count and return the item that was in the buffer at the position.
So that's the supplier demander idea. Now we do have multiple supplier threads and multiple demander threads they're all active they're all running in the system. However we've got a uni processor system so we can only run one thread at a time we choose either supplier or we choose demander and we have to remember that there are still going to be interrupts they're still going to be preemption there still going to be situations where the thread is interrupted and we don't know when it's going to be interrupted. Now the core idea of this is that the buffer count should never exceed five hundred. We should never exceed five hundred because there are only five hundred places to store in the buffer. So we can recognize that the buffer count can get as high as five hundred but it can't ever exceed five hundred. Unfortunately as we'll see because of asynchrony because we haven't taken care of the fact that the code is protected by a critical section that buffer count can easily go higher than five hundred. So we'll see that.

Steps to Producing a Problem 1.8

So here we can see the steps that we can run through to produce this problem. What we see here is that we're running the supplier thread the buffer count is already at four hundred ninety nine which means there's only one slot left in the buffer that we can fill up. So at this point in the code the supplier thread let's call it thread number one supplier thread number one is going to be running. It checks to make sure

it's not done, it checks to make sure that the buffer count is less than five hundred, and the buffer count is less than five hundred. It's four hundred ninety nine so that thread is allowed to enter that if statement. It goes past that if statement and then out in thread one right at that point for whatever reason either because of preemption or because of a hardware interrupt we get an interrupt and after entering the if statement we haven't made any changes but interrupts. So, now we have an interrupt and when we run from the interrupt, when we return from the interrupt the operating system has to choose a thread to run and it's not guaranteed that it's going to choose thread one again.

So, what does it choose? Unfortunately in this example it chooses to run thread two. So now Thread two is allowed to run and what does thread two do? The first thing it does of course is say while not done is equal to while not done so it says Yeah I'm not done checks the buffer count buffer count is four hundred ninety nine. So that's allowed, we enter into the if statement. It adds the element to the buffer it increments the buffer count now to five hundred so effectively at this point the buffer is completely full so thread two is finished thread two ends and we're going to go back now and finish out thread one. But what's already happened is thread one checked to make sure the buffer count was less than five hundred and buffer count was less than five hundred it was four ninety nine the last time thread one ran but we didn't take into account the critical section. we didn't take into account the possibility for asynchrony, so now we don't go back and recheck the buffer count so the buffer count now is five hundred but thread one the last time it ran checked it was for ninety nine so we left off at that opening curly brace and that's what we're going to pick up from and the first thing it does is increment buffer count to five hundred and one. How do we get to five hundred and one when we only have five hundred positions in the buffer. So this is really the problem that we're going to run into we don't have that extra one space and it can get much much worse than this in a real scenario but I'm just demonstrating simply that we can run out of buffer space and continue on if we don't take into account the asynchrony that can happen between these two threads.

Double Update/Missing Update 1.9

Another example that we've got of a concurrency control problem is one of a double update/missing update situation. And in this scenario what we have is two pieces of code: just a deposit function and it withdraw function. So you can imagine a situation where you have two ATM cards for the exact same bank account and you go to the bank, not at not at nearly the same time but literally at exactly the same microsecond in time, and one person is doing a deposit and the other person is doing withdrawal. And when we do that, we have two transactions that are happening at the same time on different processors. So, now we have the possibility of asynchrony because we're dealing with this with separate processors and symmetric multiprocessing. We're going to assume that the bank has a good enough system that we have more than one CPU, so we can expect that this these pieces of code are running on two separate processors at exactly the same time. The biggest issue is that they share a balance, and the balance starts out at one hundred dollars and the first transaction is the deposit which is fifty dollars and the second transaction is the withdrawal of a hundred dollars. So, we can imagine a situation where two people go to the bank at same time and one is depositing fifty dollars, and the other is withdrawing one hundred dollars. Ultimately, what we should see is that the bank balance is fifty dollars because we had one hundred plus fifty minus one hundred and I'm not great at math but that's pretty easy to do and we'll say that that's fifty dollars. So let's go out and see if that's really the case.

Double Update/Missing Update 1.10

So, we can take a look at the way this code is going to run; we can see the program counter here. I've broken it down into multiple steps, so that you can see. What we start off with is the deposit function

which is going to put in fifty dollars into the account. The balance, the new balance, becomes a hundred fifty dollars so this new balance variable is local to the deposit function and it makes a copy of the balance which was one hundred dollars and it adds the fifty dollars, so we have one hundred fifty dollars. Unfortunately, we get an interrupt or we're running on a separate CPU, so we go to the other CPU and we check out what's happening there. Now we look at the withdrawal function now and it's running on the other CPU or it's now running on the one CPU, and we say that the new balance is now the balance minus the amount, which would now be zero, so the new balance is now zero. And then we go back and check the previous thread which then goes back and copies the stored New Balance information. We go back and finish out the deposit function put the balance in as a hundred fifty dollars, which makes sense because if we had a hundred dollars and we deposited fifty dollars then we go ahead and put that a hundred fifty value back into the common shared variable, which is balance, and then the deposit thread is finished; the deposit thread exits. But we have to go back and finish out the withdraw thread, and when we do that we get into a lot of trouble because we're going to copy New Balance back up into balance and New Balance is now zero. So, our fifty dollars disappeared, because it was hidden; what happened is we had asynchrony. We had the deposit function running at the same time as the withdrawal function, and the deposit function performed its action, asynchronously, with the withdraw function, and the withdraw function never got the update according to The New Balance. So, in other words when deposit changed new balance, withdraw knew nothing about the new balance and it used the previous balance. So, that over wrote the original balance. So, ultimately what this means is that our bank gets a free fifty dollars, and we're going to have to go to the bank and fight for our fifty dollars and that's kind of annoying. Thankfully we're going to take care of this with critical sections, so that we know that we're not going to change the balance, so we're not going to lose our money.

Critical Sections Identified 1.11

So, we saw in the supplier/demander thread, an example of checking and changing the buffer and the buffer count and how that has to be done atomically; we have to look at that as one atomic instruction. We have to not be able to stop running that or run anything else until we finished running them, that set of instructions. So, we have to consider the complete supplier threat or at least the subset of the supplier thread in that portion where does the checking the if statement and the changing of the buffer count and the buffer; we have to do that in one atomic instruction, although we can't do it in one atomic instruction. So, the only way that we can consider this is that the that portion of the function needs to be protected by a critical section and we'll see how to do that. In the double update/missing update problem, really those entire functions which are very simple functions well they are as two lines of code, both of those functions need to run atomically so they cannot be interrupted from the time we start the function to the time we finish the function. The need for running those sections of code atomically, actually indicates that those sections of code are critical sections, and that's what we're trying to identify. We're trying to make sure that we know where the critical sections are and that we protect them by protection mechanisms, which we'll talk about in just a few minutes, from any untoward asynchrony that might happen inside the system.

Mutual Exclusion Rules 1.12

So the mutual exclusion rules are real simple. No two threads may be in a critical section at the same time; it just means that we can have two threads that are in the critical section at the same time. I know we said before that that is possible if we're talking about two threads that are working in critical sections on completely different resources, and that's fine we're not talking about that. What we're talking about is when we have a shared resource, we have to protect that shared resource by not allowing any threads

that have that are in their critical section to access that resource. So, when no thread is in a critical section, any thread that requests entry has to be allowed in without delay. And that means that we can't as the operating system to act as a traffic cop; we can't say to the operating system every time we want to access a critical section: are we allowed to access the critical section? Because that would be a call to the operating system and then the operating system is going to have to stop and check and then run the thread again, which requires context switching in we talked about how expensive that is.

So, we'd rather have is the way for the thread to check itself to see if it can go into its critical section, to see if it's allowed to go into its critical section, and if it is to just go straight in and not stop. If the thread is not allowed into its critical section, of course, we're going to have to have the thread pause, we're gonna have to have block and of course that's going to involve the operating system because the thread is going to move from the running state into the block state and that means the operating system has to be invoked. But really we want to avoid any unnecessary delay from the thread if it has to enter its critical section; let it in as long as nobody else is. The other part about this is that the thread can really only remain in its critical section for a very small amount of time; we don't want to have large sets of functions very large functions with a lot of work in a critical section, or in considered to be a critical section, when all we need is a couple of lines of code to be in that critical section. So, we want to have that is minimal as possible; we want to allow the thread to enter into its critical section, do its work and then get out as quickly as possible.

Fundamental Mutual Exclusion 1.13

The hardware provides us with the mutual exclusion protection mechanism, built into it; we have the system bus, and we know that the system bus can only be used by one processor at any given time. So, even in this scenario where we have symmetric multiprocessing, with multi processors in the system, we recognize that it's impossible for two processors to change the same memory location at the same time. It is absolutely impossible because one of those processors will gain access to the system boss will be allowed to change that memory location and then the other will have to wait. So, we have a protection mechanism built directly into hardware just by the nature of the way we do things, so that we can't change a memory location to two different values at the same time. So, what we'd like to do is develop something that we can use at a much higher level, to protect our code in C++ or in a high level language, from any problems that might happen because of mutual exclusion cases. So, having that fundamental hardware mechanism is very useful, is helpful, but ultimately we need something a lot easier to deal with on a high level and we'll see that.

Peterson's Algorithm 1.14

We have better solutions today; we definitely do, we'll deal with some hardware solutions in just a minute. But originally we didn't have hardware that could help us deal with this mutual exclusion problem and so computer scientists started working to try and solve the mutual exclusion problem using just software, so not having anything in the hardware to directly deal with this except for that system bus limitation. Peterson came up with a fundamental way to protect two threads from accessing the same resource at the same time and what he did was he provided each thread with the Boolean flag variable. So, there's an array of Boolean flag variables for each of the threads and each thread would put up its flag; would set it its Boolean value equal to true, if it wanted to access its critical section.

Now, unfortunately that's not enough because we're going to have to go and check everybody else's, or the other thread let's say if there's only two threads, we're going to go and check the other threads flag to make sure it's not in its critical section. And we run the risk if we set our flag and then go check the

other thread, that the other thread is doing exactly the same thing at the same time, and now we have both threads have their flags up and both threads think that the other thread is in it's critical section and in fact, nobody is in a critical section. So, what Peterson did to solve that problem specifically was introduce another variable he called the turn variable. And in here it's being generous is very helpful he thought, so what the threads do is they offer the turn to the other thread so thread zero would offer the turn to thread one and thread one would offer the turn to thread zero.

So, in that very rare situation where both threads want to access the critical section, both would raise up their flags and one would offer the turn to the other and the second one would win; the second turn right, if you will, would overwrite the first change of the turn variable. So, either thread zero or thread one would be allowed access into its critical section, and then of course eventually it would unset its flag and allow the other thread to access its critical section because it no longer has its flag up. So, the nice feature here is that it does provide protection for mutual exclusion, if we have two threads that need access to mutual exclusion. If we have more than that we have a bit of a problem, but Peterson's algorithm allows us to deal with the situation where we have two threads and we can provide mutual exclusion for those two threads using only software solutions.

Hardware Solutions 1.15

Now that we have the software solutions figured out, we'd much rather take a look at hardware solution. So, this is what we're really going to do today, after a very short while the CPU designers realize that the computer scientists were having this problem and they said: hey guys we can deal with this much easier and give you some hardware solutions to take care of your concurrency controllers, take care of dealing with your locks. And so what they did was they created some hardware built into the CPU to take care of this; most of these are instructions that are built directly into the CPU.

Now the first one is the disabling of interrupts. I don't think this is a great idea because interrupts are really necessity; we want to have interrupts in the system. If an interrupt does occurs it means a piece of hardware needs immediate servicing and we want to go ahead and do that. But if we take away the possibility that an interrupt might occur then we effectively take away the possibility of asynchrony; we guarantee that this thread will not be interrupted until it decides to re-enable interrupts. Now this would probably be limited to the operating systems control, so this would require some intervention by the operating system to use this because imagine a situation where a thread disables interrupts and then crashes; the whole system is stuck you can't get out except with a big power switch. So, disabling interrupts is something that we use very sparingly if it is in existence and even then we try to avoid it.

The other options that we have our instructions that are built into the instruction set on the processor. So, the first one is called a test and set, and in the test and set function, test and set instruction, what we do is check a memory location and if its value is of a particular value. So, for example zero if we find that there is a zero stored in that memory location we will change it to a one. So, this is perfect for a Boolean flag variable, for example, which says that maybe somebody is in the critical section. By having this test and set instruction, we can have that single variable that single Boolean variable that says there's somebody in the critical section and we can set it equal to one at the same time it’s testing it. Now this is an atomic machine level instruction. So, the CPU takes control of the system bus, goes and checks the location in memory, and then if it is the zero, goes resets that value to one in one atomic instruction; nobody else will be allowed to access that memory location. So, even if we have another processor that processor will not be allowed access to the memory location during a test and set because it has to happen atomically. So, a result is returned back to the back to the operating system or back to

the calling program that asked for the test and set, and that indicates whether it was successful. Meaning, we both tested this to make sure nobody's in their critical section and at the same time we set it to go so indicate that you are in the critical section and that means that the process is allowed to go into its critical section. Or we have a failure in which case the thread or the process has to pause and delay and try it again. So, the test and set actually works fairly well.

The other option is very similar it's called exchange, and this is a very common one that we use today, in which we have a swapping of a location in main memory with that of a register. So, we can again use, this is a Boolean flag variable, except now we can put a one in a particular register, call exchange for the memory location and we literally get a swap of the value in the register with the value in the memory location. So, if the memory location had a zero, it will now be set to one and we can go back afterwards and check to make sure that it was a zero, if it was a one then we'll know that it was a one because we'll get it back. But the nice feature is that when we do the exchange, even if we're interrupted immediately after the exchange instruction, the indication that we're in the critical section has already been set and now we just have to check to make sure whether it was successful or not. So, the exchange instruction is a very common one that's in use today and it's relatively easy to understand.

Semaphores 1.16

The common solution that a programmer would use today as a high-level solution would be the semaphores. Now semaphores of course are going to have some issues internally to them, so they're going to use some of the hardware level instructions that we just talked about. But the semaphore will provide for mutual exclusion. Now the word semaphore actually comes from a nautical term; I think it's needs a flag that used to be flown on top of the ship and it's very much similar. So, what we have here is two functions that we generally use inside the semaphore, we construct the semaphore and then we send it a signal. The idea of a semaphore is that it's a message passing structure, so one thread could send a signal and the other thread could receive that signal. Now the way we receive the signal is by calling wait. But here's where this is very useful for mutual exclusion, the wait function is blocking. So, if there is no signal that has been sent, then the wait function causes a block to wait for the signal to be sent, and that's where we can really use it for mutual exclusion.

Semaphores – How to Use Them 1.17

So, initially when we create the semaphore we send a signal in the semaphore, so that's an initial cueing of the signal. Now assuming that we just created it I'm going to assume that nobody's waiting on the semaphore, so the signal is just going to sit there and wait to be received. When we enter a critical section the first thing that we do is call wait. Now as long as there's a signal already there, then wait is non-blocking and the thread is just going to be allowed to enter its critical section. But the key feature is that the wait will consume the signal that's queued, so now there's no signal that's queued. If another thread calls wait to enter its critical section then it will not have a signal and it will be blocked and the operating system will keep a queue of the blocked processes so that it can start those processes again as the signals come in. When we're done, when the thread is finished with its critical section, it’s going to call signal and signal is going to either queue up the signal if there's no threads waiting or if there is a thread waiting then the signal is going to release the next thread from the queue. So, this allows us to create a mutual exclusion protection mechanism using just a simple signal passing routine

Semaphore Internals 1.18

So, we need to look at the internals of the semaphore and we need to have an understanding of how the semaphore works internally. So, what we're going to do is kind of piece together what happens inside a semaphore, without actually going through the nitty-gritty of how the operating system creates a semaphore and maintains a semaphore. But what we do is we create a counter inside the semaphore object, and the semaphore is going to indicate how many signals, or the counter is going to indicate how many signals have been sent. Each weight causes the counter to decrement and if the counter ever becomes negative then the thread that causes it to go negative, and of course all subsequent threads, are going to be blocked and then they're placed on a queue. Now the operating system maintains the queue, and it's a queue of blocked threads waiting for a signal on that semaphore. So, that's going to be our indication that we can release a thread, but they all go into the blocked queue; all those threads that come in and call wait, that cause the counter to go negative, when the counter is negative those threads are all blocked and the only thread that's allowed to run is the one thread that consumed the one and only signal. Now when that thread is finished, it has to send a signal, and if there are any threads on the queue then the signal will cause the counter to increment; first signal always causes the counter to increment. But it also releases the next thread in the queue, if the counter actually goes positive that means that there's nobody in the queue waiting for a signal so there's nothing to be done there, but we recognize that the signal in the wait function allow us to create this mutual exclusion environment. Unfortunately, internally to the semaphore, there exists the critical section; the act of looking at the counter and then changing the counter is itself a critical section. So, here we have a mechanism that we're trying to use to protect against critical sections, and we created a critical section. So, that's a real problem so we have to use a hardware solution to prevent asynchrony when the semaphore has to both check the counter and set the counter. So, this is where the semaphore is going to use someone of the hardware options, either disable interrupts or test and set, probably exchange, and it's going to take care of it internally. So, we have to recognize that semaphore itself has a critical section and it has that in order to solve our critical section.

Deadlocks 1.19

Now that we've talked about semaphores enough, we have to talk about one downside to having all these locks and that’s the idea of a deadlock. And the deadlock occurs in a system, or can occur in a system, if all the threads are waiting for each other; if we have a set of threads that are all waiting for each other, then there's really no way that any one of them are going to complete. Usually it results from one thread waiting for another thread to release a resource, and it's not really clear what that resource could be; it could be a lock, in terms of a mutual exclusion lock like semaphore, or we could be talking about a file, or we could be talking about a mouth of memory. But there's some resource which is shared among the threads and one thread is waiting for another thread to release the resource, while that other thread is waiting about eventually back for the first thread to release the resource. So, we have one A waiting for B, B waiting for A neither is going to get done; it's a permanent block, it is not going to resolve itself over time. This requires the intervention of an operating system if the deadlock does occur, there’s going to have to be some resolution that the operating system or that's a monitor program takes on top of the the deadlock threads to resolve the issue. And unfortunately today there's really no efficient solution, even today we don't have a very efficient solution to deadlocks. It either amounts to a considerable waste of system resources, processor time or memory, or it amounts to a deadlock occurring and then resolving the deadlock. So, we really don't have a good solution today to deadlocks, other than to say we have to keep them in mind and make sure that they don't happen.

A Real Deadlock 1.20

I like this picture because it really shows traffic inside a normal city environment. And what we have here is that each of the cars that would enter the intersection sort of went a little bit over and are now blocking a portion of the intersection. And what you can see is that each car is waiting for another car to move and the last car is waiting for the first car to move, and ultimately as you can tell, nothing is going to get done here. There is no way that any of these cars are going to move until someone decides to take action, and just completely change the scenario; maybe one of the cars could make a turn and get out of there and then free up the intersection. But ultimately what's happening is we are controlling one quadrant of the intersection, each car controls one quadrant of the intersection, and needs one more quadrant in order to complete its task of moving through the intersection and we recognize that that can never occur.

Deadlock Resource Types 1.21

I've been talking about using locks, or using a semaphore, but in reality we have two different resource types that we can talk about when we get into deadlocks. One is reusable resource types, and the other is consumable resource types. Now a reusable resource type happens when we have a resource that we will temporarily consume, for example, we will temporarily take action on the resource, we will temporarily take over control of that, resource. But then once we're done with it we'll return the resource back to the operating system or back to the the whole so it can be reused again. So, examples of this include things like memory, devices, data structures, even semaphores; once we're done with these things we return them back to their original state and they're allowed to be used again. But there's consumable resources, which once we use it it's gone. For example, if we have an interrupt and we deal with the interrupts, or we if you will consume the interrupt, the interrupts can't be replaced. If we get a signal then the signal can't be replaced; if we got a message then the message cannot be replaced. If we're for example, processing data in an IO buffer as you did in previous examples where you read in from the keyboard, once you read in that data there's an easy way to put it back into the buffer. So, consumable resource types, once they're used they're gone, and the reason that we're recognizing this is because how we deal with a deadlock on these different resources is going to be different from one resource type to another.

Items Required for a Deadlock 1.22

There are four things that have to come into play for a deadlock to actually occur. And some of them are requirements that the operating system needs to have, and some of them are just bad luck that have to happen, but we need all four of these to happen in order for us to have a deadlock. The first one is mutual exclusion and we already know what mutual exclusion is, we've been talking about it, we understand that there's going to be a resource and this resource can be held by only one thread at a time; so we have a lock on that resource if you will. If we have that then, we have the first item that's required for a deadlock.

The second one is called a hold-and-wait and that means that while a thread is waiting on the allocation of a new resource it retains all of its existing locks on the old resources. So, if a thread obtains a lock on A and then requests a lock on B, and the lock on B is not available immediately, the thread is going to be blocked. So, it means that the thread could possibly have a lock on something and be in a block state at

the same time. So this happens, we understand that this happens, certainly when we're creating more than one lock at any given time.

The third thing that we need in order for deadlock to happen is that there is no preemption. The operating system cannot forcibly remove a resource from a thread. So, for example on a semaphore, once we've decremented the semaphore counter and it's now zero, the operating system has no way of intervening and saying, 'oh whoops I'm sorry you no longer have that lock on that semaphore and you have to go back to doing something entirely different.' We usually don't have that as a possibility; it's the idea of no preemption.

The last one: and the last one is just case of bad luck it's called a circular wait. And that's a closed chain of threads in which the last thread is waiting on the first thread. This could be as simple as A waiting for B, and B waiting for A, but it's often not as simple as that. We can have four different threads: where we have A, B, C, and D. And we have A waiting for B, B waiting for C, C you waiting for D, but D waiting back for A, so it's not easy to recognize that a circular wait has occurred; it's not just take a snapshot and see which process, which threads are waiting for which threads it doesn't work that that easily. There's lots of levels of iteration and recursion that might have to happen in order for us to detect that the circular wait is actually happening. If we have all four of these things in play: we have mutual exclusion because the operating system provides us with locks, we have hold and wait because we can have more than one lock and we request one lock separate from the other locks, and we have no preemption meaning the operating system can't take things away from us, and we have a circular wait which just happened to occur, we have a deadlock.

Solutions for Deadlocks 1.23

So there's three ways of dealing with deadlocks: and the first way is called deadlock prevention, the second is deadlock avoidance, and the third is deadlock detection. Deadlock prevention requires us to take away one of those four requirements; either we're going to take away no preemption, or we're going to take away holding weight, or we're going to avoid the possibility of a circular weight. The idea of taking away preemption, kind of difficult, we'll talk about that in just a minute. We might have dead lock avoidance; deadlock avoidance creates an algorithm in which we are going to check the system to see if a deadlock could occur whenever we make an allocation for a resource. So, now the operating system now is going to intervene, and say, if you're going to make a request if you're going to lock something, the operating system will check the state of the system to see if a deadlock will occur as a result of you're making a lock on that. And if a deadlock will occur as the result of you're making a lock on that device, the lock is not approved and the thread is either told to go do something else or it's blocked all together until the lock can be approved. So, deadlock avoidance says we recognize that all four requirements for a deadlock are in play, however, we're going to avoid the deadlock by preventing such a situation just from occurring. Deadlock detection is the last way and it's the easiest way, it just says deadlocks are going to happen we'll let them happen and periodically we'll just check to make sure that if there are any dead locks that we just deal with them in some method. So, deadlock detection is the easiest, so to speak but it's also the most difficult to deal with after the fact. So, now we have to deal with the fact that a deadlock has occurred, and we have to recover from it. Those are our solutions that we have for dead locks and we're going to look at them individually.

Deadlock Prevention 1.24

The idea of deadlock prevention is that we eliminate one of the four requirements for a deadlock. The first one is mutual exclusion, and it's really hard to get rid of mutual exclusion because it's usually a requirement. Whenever we're working with threads, we recognize that these threads are going to have the possibility for asynchrony and if they have the possibility for asynchrony, we have to provide a possibility for mutual exclusion. We have to provide the protection mechanisms for mutual exclusion; so getting rid of mutual exclusion is usually not a possibility.

Hold-and-wait: we can say that all locks are required to be obtained at the same time and either they're all approved or the thread is blocked. Now if we do that then there's no additional lock that might occur. If a thread holds a lock, it cant obtain another lock, which means that we can't get into a situation where we're holding a lock and we're asking for a new lock and therefore, we're blocked because we're asking for the new lock. So, if we make the threads ask for all of its locks at the same time, then we've eliminated the possibility of hold-and-wait.

No preemption: we can say that if a thread requests a new lock and the new lock is denied then the thread has to release all of its existing locks. So, that's a possibility, it's got to require a callback from the operating system to the thread to say if the request for the lock is denied unlock everything else. Put it back in a safe state; undo what you were doing and release those locks. The operating system might also have the possibility the authority to remove existing locks individually but that's a bit of a bad idea.

The last one is actually an easy one to deal with especially if we're talking about locks on individual items that the programmer can control; one of the easiest solutions for dealing with the circular wait is to order all the resources. We can number them and order them so that there's never a possibility of holding a higher level resource and requesting a lower level resource. So, one of the problems of the circular weight is that the last thread is waiting on the first thread. What that means is that the first thread holds a lower lock and the last thread holds a higher level lock and if we prevent the last thread from accessing its lower level lock, because it holds the higher lock number then we've prevented the situation of the circular wait. Either that higher level, that last thread will either release the higher level lock and then request the lower level lock allowing other threads to continue or it won't be allowed to do it. So, the circular wait is actually an easy one and it's the one that's recommended by a lot of computer scientists because the programmer can protect his own program internally, just simply by ordering the locks, ordering resources. So, that we never ask for a lower lock while we're holding a higher lock

Deadlock Avoidance 1.25

In the deadlock avoidance scenario, the operating system has to know everything that the thread is going to do before it actually does it. So, the operating system has to know what locks the thread is going to want, in order to look into the future, in order to have a premonition about what could happen in this deadlock scenario. So, what the operating system does is it takes all the information for all the threads that are running, and says that whenever a thread makes a new request, the operating system is going to run what's called a banker's algorithm, or what's often called a banker's algorithm. And the idea is that we know all the resources that the operating system has, we know all the resources that the threads have, and we have some premonition about what the resources the thread is going to want in its entire lifetime. So, we can tell which threads are going to run to completion just by knowing what we currently have available and what the threads are going to want before they're finished. And what we can do is constantly check, whenever a resource is requested, the operating system is going to run the bankers algorithm to try and figure out if the allocation of that request would cause the system to be in

what we call an unsafe state. And an unsafe state is one in which all the threads cannot complete, rather that we can say that any thread cannot complete, so we're protecting the system from ever getting into that unsafe state. So, we sort of simulate by making the allocation in a simulation, and saying can we now finish all of the threads and if we can, we can say that that allocation keeps us in a safe state and we make the allocation and if we can't then we deny the allocation. So, the system is by definition always in a safe state, but we're checking every single time to make sure that the allocations will not put the system in an unsafe state. It's a lot of work and we usually don't have information about what the threads are going to ask for ahead of time, until they actually ask for it. So, that makes dead lock avoidance a very rare strategy.

Deadlock Detection 1.26

In deadlock detection, it's really easy going. All it does is say we have all the requirements, there's a possibility for a deadlock, so we'll check periodically to see that the deadlock is not occurring. It's not at all restrictive; we're not saying that we were going to stop any situation from occurring. We're saying in fact the exact opposite: that deadlocks will occur and when they do occur we're going to have to deal with them, and we'll talk about that in just a minute. But we recognize that the deadlocks will occur and that periodically we're going to have to check the system to see if there are any dead locks that are in existence. Once we detect the deadlock, well we have to figure out what that's going to do and the only ways that we can really deal with a deadlock are either rolling back the deadlock process, or the deadlock thread, to a previous unlock state that's useful for things like SQL servers and database servers, but it requires transaction logs. So, we have to know what this thread, or what this process that's deadlock did in the past X time interval, so that we can roll it back to that X time interval and either restart it or let it continue from a previous state, when which it wasn't deadlocked. We could also just say let's abort all the deadlock threads; we'll kill them all. We could abort a single thread and see if the deadlock is being resolved, so we can arbitrarily just pick one of the deadlock threads and say it's now killed, its resources are released, and now see if the system is back to a stable state, let it run again and see if there's a deadlock or not. Or we can start pre-empting resources.

Unfortunately, in all of these situations we have the potential, except for rolling back, in all of these other situations we have the potential for catastrophic loss of data or corruption of data. Basically we're saying that we don't think that the system is in a consistent state, but we really have no other resolution other than to just say we have to take some emergency action and that means we recognize that if the data is corrupt we have to deal with that. So, deadlock detection says, just let it happen and then we'll deal with it afterwards, we hope it doesn't happen. And believe it or not this is the one that's most common today, that we just say let it happen and if it does happen we'll deal with it. You might have had this in your past you have an application that just completely freezes up on a Mac, you get the beach ball on a Windows machine you just get the screen graying out; it's doing something we have no idea what it's doing, so what happens we kill it. We are the deadlock detection algorithm; we have recognized that a deadlock has occurred and we abort all the deadlock threads by killing the entire process. So, congratulations everybody has been promoted to a deadlock detection algorithm.

Dining Philosopher's Problem 1.27

I'd like to take a look at one classic computer science problem. It was first presented by Dr. Edsger Dijkstra who was a famous computer scientist that you're going to hear a lot about Dr. Dijkstra with his various assorted algorithms throughout your careers in computer science. But it's a computer, sorry, it's a

concurrency control problem and what he said was… and I really have no idea how he came up with this idea, this scenario but somehow he created it. He said that there's a house where we have five philosophers and the Philosopher's are numbered 0 through 4. And they alternate, they only do two things, philosophers only do two things: they eat and they think. And when it's time to eat, somebody, we were not figuring out who, sets up a circular table and it has a place setting for each philosopher. And the place setting consists of one fork on the left side, and one plate in front of the philosopher. The meal unfortunately that they're going to eat is a particularly difficult to eat kind of spaghetti. I'm quoting dr. Dijkstra here, this is not me, this is Dijkstra's dining philosophers problem. I don't know why spaghetti, or why you eat spaghetti with two forks, but he said that you need to have two forks to eat this kind of spaghetti. And unfortunately, each philosopher is only given one fork. But we recognize that there's a fork on the left of the philosopher and a fork on the right of the philosopher.

The way that we can think about this is that when the philosopher comes he can pick up the fork on the left, and he can try and pick up the fork on the right, the unfortunate problem is that if all five philosophers come to the table at exactly the same time, all of them will pick up their left forks and there will be no right forks. In other words, all of them are holding a lock where there's mutual exclusion because the philosopher doesn't let go of his fork, all of them are holding the lock on the left fork and waiting for the lock on the right fork. All of them have no preemption so nobody's going to reach across the table and start grabbing a fork these are nice philosophers, I guess they're not from Brooklyn, I'm kidding. And we have a circular wait we have the last philosopher in a sequence waiting for the first philosopher on the sequence. So, ultimately what we have is all four requirements, we've met the deadlock requirements, and in fact you can see that we have a deadlock. In this scenario all the philosophers will starve to death, because none of them will put down the fork and allow somebody else to eat.

The Dining Deadlock 1.28

So like we see, if all the philosophers are allowed to take a fork, all will have one fork and none will ever eat. So, what's the solution to this? Well Dijkstra proposed the solution of resource ordering, of course. That's the easiest one. And he said, let's order the forks zero through four. Now, if you pick up fork zero, you're not allowed to pick up fork four, and that leaves fork four for the philosopher on his right to pick up, and that means that the fourth philosopher will be allowed to eat because he has his own fork plus the last fork. We can't select a larger, we can't have a larger fork and a smaller number fork at the same time unless you ask for the smaller fork first then the larger fork second. So ultimately what this means is that one of the philosophers will get both forks, will eat, and put both forks down, allowing more philosophers to eat and more philosophers to eat and ultimately, everybody eats. So that's the resource ordering solution. But there's another solution in which we have a semaphore. And obviously, we're going to have semaphores for each of the forks, you can see that already cause we need mutual exclusion on each of the individual forks. What we're talking about is adding yet another semaphore, and saying that there's a semaphore that has to be met that has to hold mutual exclusion for the entire room. So with the semaphore solution, we have four signals that go into the queue under semaphore; the first four philosophers are allowed to enter the room, and they're allowed to eat. The last philosopher, of course the fourth philosopher, will pick up the fork from the fifth philosopher because he's not allowed in the room at all. And that allows the fourth philosopher to eat, and then he'll put his fork down, allowing the third philosopher to eat, and then he'll put his fork down, and so on and so forth. Eventually, after the fourth philosopher leaves the room, the fifth philosopher will be allowed in and he'll eventually be allowed both forks; everybody is allowed to eat and the problem is resolved. So, we've got two scenarios here and there's actually even more scenarios that get even more complex; there are more solutions that get more complex. There are solutions using a monitor, and there's even more complex ones.

But the fundamental, the easiest to understand, is that we either order the resources, and order the forks, or we have a semaphore to prevent one of the philosophers from even entering the room.

Conclusion 1.29

We talked about a lot during this module. We talked about what threads are; we gave a little reminder about the threads. We talked about features of having multiple threads accessing multiple resources and we said that there was the possibility, of course, that we could have asynchrony. And if we do have the possibility of a synchrony, we said that we need to identify critical sections so that we can protect those critical sections using concurrency control mechanisms. And those concurrency control mechanisms that we talked about are software and hardware based. We talked about the idea of mutual exclusion using semaphores. We talked about the design of the semaphores, and how they work internally we also went into how deadlocks occur and how we can prevent deadlocks. and then we talked about the dining philosophers problem. So, we really covered a lot during this module.