Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 11
Lecture - 59

Lec 59: Uninterruptible Instructions

Hello everyone. We are starting module 8 on synchronization. This is lecture number 1 in which we are going to discuss about uninterruptible instructions. Okay, so let us first understand why we need to synchronize. In the lectures on memory consistency, when we discuss sequential consistency, we observed that sequential consistency was very strict.

It did not allow any compiler optimizations, it did not allow us to reorder and that would affect the performance of a single core as well as a multi core system. So we want the memory consistency model to be weaker. When we go to a weaker model and we still want to satisfy sequential consistency, we decided that we will enclose such set of instructions into a safety net using some fence instructions or barrier instructions or read-modify-write type of atomic instructions. Okay.

So in this module, we are going to understand further on this concept of atomically executing an instruction. Apart from sequential consistency or memory consistency in general, even in the context of coherence, coherence only says that at a time one process can change a particular variable. But in a generic scenario, when multiple processes are accessing one shared variable and it is required that exactly one process is able to use that variable for an extended duration, we need to give exclusive access to this process to that particular shared resource. Popular example is a printer. So if you want to print a task, you cannot allow multiple processes to use the printer.

You want one process to first complete its printing task only then the second process can begin. So this is a shared resource and how do you guarantee exclusive access to a process? Okay. So this is popularly done using mutual exclusion where you access the resource in an exclusive manner then the process enters a critical section, completes the use of the resource and then relinquishes that resource. Okay. So this is the popular concept of synchronization so that exclusive access to one shared resource is established.

Right. So we need to synchronize so that shared data as well as resources can be used safely by different processes. Synchronization of course is more generic and we can't

say that if a system is coherent or a system is sequentially consistent, I would not need synchronization. Well, we definitely need synchronization and that it transcends the coherence as well as consistency models. So in spite of the presence of coherence and consistency, we do need synchronization.

Okay. So even if my system is sequentially consistent, I will still need synchronization. Okay. So for example, we have a printer and a shared variable in the memory and that has to be accessed by multiple processes. So exactly one of them should get access and for this we need to synchronize. How do you enforce this? We normally do this using flags or semaphores. So this shared variable which is there in the memory, we need to access exclusively.

So if I am using a flag to access to that shared variable, so this flag or something called synchronization variable is used inside the memory. So we have this variable, I can call it a lock variable and I will gain access over this lock variable so that I get permission to use that resource. So this shared variable or the lock variable or the flag has to be free for us to use the shared resource. If it is not free, then it is derived that there is some other process using that resource. Hence, we need to wait until the resource becomes available.

Okay. So this lock variable which is there in the memory, before we want to acquire access, we have to go to this lock variable and then lock it. I mean, is the lock variable free or is it locked? So we need to establish this and to acquire a lock, we need to go to that location and then set a particular value inside this. Now writing a value into this has to be done atomically. Okay. So we need to do atomic update of this shared variable.

So how do you implement this in hardware? So synchronization mechanisms are definitely there in software and most of these software programs they rely on hardware supplied instructions. Right. So they rely on the instructions given by the hardware to implement synchronization. Okay. So how much support does a software need? So there has been always a debate that the software should do more and the hardware less and vice versa.

Okay. So how much support should the hardware give to the software is exactly not known and there has been a debate about who should do more task. Okay. But leaving that apart, we need a generic understanding and that was established by the classic works of Dijkstra and Knuth. So they observed that it is possible to provide mutual exclusion with only an atomic read-modify-write type of an instruction. So if I have a read modify write type of an operation in a memory which is sequentially consistent then I will be able to provide synchronization.

Okay. So this has to be implemented. So our job in the hardware is to implement such atomic read-modify-write type of instructions and all practical implementations rely on such a support from the hardware. Okay. So no matter whether hardware provides bigger routines or complete routines to establish synchronization or whether software does more of the task but the bottom line is hardware has to provide some sort of an atomic read-modify-write operation. And this atomic operation essentially means that you have to first read that variable, modify it and write.

So while you are doing these three actions, they have to be done atomically without intervening accesses from other processes. Right. So there should be no other process which is able to go and modify this variable when one particular process is doing that. So if we can do this only then synchronization can be implemented in hardware.

Alright. So what are the different options or what are the different options used by the processors? So this slide lists the different types of instructions used by variety of real life implementations. The IBM 370 uses compare and swap type of an instruction. So this instruction is an atomic instruction which goes to the memory location which is the shared lock variable. It goes there, compares that value and swaps that value with its local variable. So for example, to lock the variable, I need to write a 1 into that lock variable and the lock variable if it has a value of 0 means the lock variable is free.

So compare and swap what is it going to do? It is going to go there, check the lock variable. If it is 1, we can't do anything because some other processes already locked it. But if the lock variable is 0, then we need to write a 1 into it. So we are going to compare whether the value is 0 and if it is 0, we are going to swap that value with a 1. So I am going to remove the 0 and write a 1 into that location.

So this is the compare and swap instruction. Intel uses a prefix for those instructions which it wants to be executed in an atomic manner. Sparc architectures also use swap and compare and swap. Then the MIPS has come up with something new. They do not use a single instruction but they use a combination of two instructions and which would eventually give us the same effect. Okay.

So we are going to see more details on this. So it is going to use a special combination of a load and a store to establish the read-modify-write type of an operation. And this approach of using two instructions has become more popular and was incorporated by newer systems. Okay. So synchronization where we want multiple processors to access a shared resource but before that it has to obtain a lock on the lock variable would become a bottleneck if I have multiple processors and several processes trying to access. So our task should be or objective should be how do I reduce the contention for this variable,

how do I optimize on the latency of obtaining a lock on the system.

So as a designer you have to keep these things in mind. Okay. So what are the different components of a synchronization event? So I can divide it into three parts. One is the acquire, second is the waiting algorithm and the third is the release method. So acquire is the method where the process tries to get right on the synchronizing variable. So I want access or control of this lock.

So the lock variable I want to go and lock it for myself. So this is the acquire method. When I want to do this, if I do not get the lock I need some mechanism to keep waiting and checking for this lock. So that is the waiting algorithm. And third is after I finish using the resource I need to release the lock, that is go to the lock variable and maybe reset it or decrement it depending on the scenario. Okay.

So synchronization is established using three major components. You need an acquire method, you need a waiting algorithm and you need a release method. Okay. So let us see some examples to understand this further. So suppose I want to implement a distributed linked list something like this and this linked list is being updated by multiple processes, okay, running on different processes. So if P1, P2, P3 all of them want to add a node to this list at the head position they will create that node and this new node has to become the head node.

So this is a head pointer. So this is the data and this is just a pointer. So this pointer variable needs to be changed by these three processes because once they add a new node, the head pointer should point to that particular new node. So how should they do this? So for changing the head pointer, they first need access to a lock variable which is our synchronization variable. So we will declare a flag or a lock variable in the memory then all three of them will go and try to write a 1 inside this because 0 means the lock variable is free and 1 means the lock variable is engaged. So when P1 wants to add a new node, so suppose P1 creates a new data node for itself and now this new node should become the new head pointer. So it wants to update the head pointer.

So it goes to the lock variable and then checks whether it is free or locked. If it is free it locks it, updates the head pointer and then releases the lock variable. Same thing will be done by P2. If P1 has already locked that variable and if P3 comes to check it will see that the variable is locked, so it will go back and keep waiting until P1 has finished its task. When P1 releases the lock P3 will see that the lock is now free and it creates its own new node, locks the variable and then updates the head pointer.

So on this right side you can see a pseudo code for doing this. I have multiple processes

and when one particular process say Pi wants to update the head pointer, it first has to acquire the lock. Okay. So for acquiring the lock, it has to go to this lock variable and then check whether it is free or engaged and accordingly first obtain a lock onto the system. So this is going to take some time. Okay. This is going to take some time. Once this is finished then we can enter this area which is called the critical section.

So here we are going to update the head pointer with the new node. Once we have finished updating the head pointer, you have to unlock the lock variable, so that others can start using it. Okay. So this was an example of when do we need synchronization. To update the head pointer, exclusive access by every process and for that you obtain the lock and then use the code to update the head and then release the lock. How do you actually get this lock implemented? How do I acquire the lock? What are the different methods to change the lock variable from 0 to 1? Okay. So that should be our objective when we understand synchronization from a hardware perspective.

Well, if you think can I do this in software? Software meaning not necessarily high level programs but can I do it in pure assembly using some set of assembly instructions. So simple software lock algorithm. So here we are going to discuss that if I want to obtain lock on this lock variable what should we do? We have to first go to that lock variable, read its value. If its value is 0 then we have to go again to that lock variable and write a 1 into it. Now imagine a scenario that there were three processes, all three go to the lock variable, they read it, they get the value as 0. Okay.

So if they all are lucky they will see the value as 0 and when they see the value 0 they want to go and write the value as 1. Okay. So when they write the value of 1 what has happened? All three were able to establish a lock and they all three go into the critical section so which is not desired. Okay. So just having this set of instructions that you go first, read the value and then go again later to write the value is not going to be effective. Okay. So what has gone wrong here, between the read and the write, there has been no communication and it was not atomic and therefore we could not establish the mutual exclusion. Hence our reading of the lock and setting it to 1 should be done as an atomic operation and that can be done only in the hardware. Okay.

So this is the program where you will load the value of the lock variable. If it is not equal to 0, that is lock variable has got the value equal to 1 you go and keep looping. Right. So this is the waiting algorithm. So you keep waiting for obtaining the lock. Once you see that the lock variable is 0, you go and write a 1 inside this, so you obtain the lock and when you want to release, you simply write a 0 inside the lock variable.

So we have a busy wait which waits until we get a value of 0 and then we want to set

that value to 1. Okay. So as discussed two processes can read the value 0 into their local registers at the same time. So this can be done by say, P1 also does this, P2 also does this in parallel. Both of them see the value as 0. Now both of them have seen the value of 0 they will go and write 1 into it. So P1 also tries to write a 1 and P2 also tries to write a 1.

So P2 also tries to write a 1 into the lock variable and nobody can stop this from happening because it will be done in a coherent manner. But the 0 has become a 1 and it does not check what that value was, it simply goes and writes a 1. So both of them will eventually write 1 and then enter the critical section. So this is not satisfying our requirement and why has this happened? Because we could not do the load and store atomically. Even if they are two consecutive instructions, even if I put them one after the other, still we cannot guarantee atomicity and therefore we need support from the hardware for atomic instructions like the read-modify-write type of an instruction. So what are we going to see overall in this topic? We are going to look at uninterruptible instructions, spin locks, point to point event synchronization and global barrier synchronizations.

Presently we will look at the first one of uninterruptible instructions. So we will begin with an example of test and set. So test and set is a atomic instruction like read-modify-write which goes to the location, tests that location and then sets it to 1. This is how it will acquire a lock. So I am going to use pseudo assembly code for writing this.

So test and set, so first I am going to do write, this is the instruction t and s. So test and set, lock variable. So I have a lock variable. I am going to read this and set it to 1. So R1 I am assuming that this register R1 has got what value of 1. So I am going to write 1 into the lock variable and this is the lockvar in the memory.

So we go here, we are going to test the value and we are going to write a 1. So when I go, I test the value. So the testing value will return the old contents. So return the old value into R1 and write a 1 inside this. Right.

So that is also similar to an exchange. So the present contents of lock variable will come in the R1. So in case the returned value is 1, what will you conclude, that the lock is not available. So branch if not 0, so if R1 is not 0, that is the lock is already locked by somebody. What should you do? You should keep on trying to acquire the lock. So I will just say go branch to try. So try is a label in my program which is simply the first instruction.

So you keep looping in this to acquire the lock. Once you acquire the lock, you can enter the critical section and then you have to unlock. So what should you do for unlocking? You have to simply write a 0 inside that. So you will simply say store lockvar value of 0. Okay. So this is the unlock.

So this is the loop where we are going to wait for the lock. This is the critical section and this is the unlock. So once you finish the work, you have to unlock. So we have seen an example of how to use test and set as an atomic instruction. This is instruction implemented in hardware which goes, checks the value and writes the one.

So it does both read and write at the same time. Okay. So a similar code I have written here. The only thing is the label is called lock in this. So the lock function call will obtain the lock and we will have an unlock set of instructions which will simply write a 0 inside the lock variable. Okay. So I hope you got a flavor of how these instructions look like the test and set.

So there are more options for this. So we have test and set which goes, checks the value and writes one into it. The second instruction available is fetch and increment. So fetch means go read the value and whatever is the value increment it and come back. So fetch and increment as an atomic operation.

So if it is 0 make it 1, if it is 1 make it 2 and so on. Then we have swap instruction which swaps the contents of one register with the lock variable. So you can load the register with the value you want to lock. For example, I wrote the, load the register with 1 and then I swap the register contents with the lock variable to see what was there in the lock variable and if it was 0, I have already written a 1 into it. Okay. So swap is going to do the same thing like test and set. Okay. And then we have a fourth option which is LL-SC, that is load linked store conditional.

So in case implementing single atomic RMW type of an instruction is difficult, we can go for these two instructions which load and store separately but they guarantee atomicity. Okay. So these are the different options available for us. Test and set, fetch and increment, different types of swap instructions and load linked and store conditional. So test and set, we have seen one example.

Fetch and increment would have similar type of a code. So let us try the swap instruction which is popularly called the exchange instruction. So here again I want to exchange the value of the lock variable with my register. So let me say I have this lockvar that is in the memory. I have a register say R2 and to obtain the lock I should be writing a 1 into it. So I will initialize this to 1 and I will say you write this here and you

bring this back here.

Okay. So that is the exchange we are going to do. So to establish this either you can simply say that initialize R2 to 1 or you can write more fancier instructions, I will say add unsigned immediate. So this is one instruction which adds a value to R0 and the immediate value is equal to 1. So R0 is a register which has the value of 0. So if I am taking R0, # 1, it says add 1 to R0 and write it into R2.

So this is the destination. So in assembly most of the time the first operand here is the destination. So R2 is the destination, inside this what am I writing R0 plus 1. So R2 will get the value of 1. Once we do this, I will write the atomic instruction exchange.

So this is the instruction name. What am I exchanging? R2 and I am exchanging R2 with the lock var. Okay. That is shown in the picture. So this is what we are doing. Once we do this, what should we do? The exchange has already written a 1 into R2 but what was there inside the lock variable has come back into R2.

If it was 0, we can move on. If it was 1, we should not be allowed to obtain the lock. Okay. So we have to keep trying. So I will say try again or try again. When should we keep trying? If the R2's current contents after the exchange is equal to 1.

So branch if not 0, that is the lock was not 0. So check R2. If R2 was not 0 you can go to the try label. Okay. So this is how we can use exchange as an instruction to acquire the lock.

So branch if not equal to 0. So BNZ is also same as BNEZ. Okay. They are same instructions. Alright. So we have seen exchange. Now I will discuss the concept of that waiting algorithm.

So waiting algorithm was going to check the value of variable. So you do an exchange and then after you do the exchange you want to check what value was there in the lock variable. So if I have a system where there is only the memory. Okay. There is no cache.

So this is your exchange instruction which I wrote in this previous slide. So this exchange instruction. It goes to the memory's lock variable and simply does the exchange. Exchange actually means it does a write to that system. Okay. So when I am doing a write to that variable, what is happening in a multi core system and in the presence of caches, you are going to generate lot of invalidations. So all the other copies will get invalidated and unnecessary bus traffic and invalidations will happen because of

our decision of exchanging.

So what can I do? Most of the time I want to first read the value of the lock variable. If it is 0, only then I want to make it 1. So can I divide this task into two pieces? Let me first check the lock variable. If it is fre,e only then I will try to exchange. Okay. So this solves the problem of unnecessary invalidations and it also helps me to spin on my local variable.

So this lock variable I have brought already into my cache because I am going to only read the variable. I am not going to exchange. I will go read the lock variable.

If it is free then I go and try to exchange the value. Okay. So I have these three processes and in the cache of this process, I will do a read on the lock variable. The idea is that we use locality concept, that is if a process tries to acquire a lock, it will again try to acquire the lock in near future. So it is good to cache the lock variable. Most of the time you are going to get cache hits on this variable.

So you first do a read on the lock variable. Bring the variable in your cache and keep checking it. Because if it was 1, it will remain 1 until somebody changes it. If you read this value as 1 there will be somebody else who is using this lock. When that process releases the lock, it is going to write a 0 to this lock variable. So when it writes a 0, it is going to send an invalidation to all the caches which will then remove this variable from their cache. Okay. So suppose this is the one which is currently owner of the lock and when I do a read on the lock we get the value as 1.

Why? Because the yellow process has already acquired the lock. When the yellow process finishes and it executes a release what will it do? It writes a 0. So it has to write a 0 inside this lock variable and that 0 will be sent to the memory and invalidations will be sent onto the bus to all the processes. So this invalidation will remove this variable from these caches. Okay. Next time when the green process again tries to loop, that is check the lock variable it will again issue a read on the lock variable.

It will incur a cache miss because the variable was deleted. So it goes onto the bus to the memory and then fetches the latest value. Very luckily it would get the value as 0 and it will then execute the exchange to lock that variable. Okay. So there will be definitely a race among the processors to acquire the lock. So for example green and pink both want to acquire the lock.

So both of them will want to go into the bus to the main memory to write a 1 into the lock variable. But one of them will succeed because either it is a directory coherence

protocol or a bus based coherence protocol, there is serialization and exactly one process will succeed in writing a 1 to that particular copy and definitely other copies will get invalidated. Subsequent reads by the other processes will show that the value is equal to 1 and they will have to keep waiting. Okay. So this is how we can spin on a local cached copy. Spinning on a cached copy means keep reading the value and when you see the value as 1, just keep reading your local 1 until your local copy shows that the value is equal to 0. Okay.

So you keep reading and once you see the value was equal to 0 then you try to exchange. Exchange is a write operation and exactly one of them will succeed. So most of the time when we are spinning, so this spinning time is saving the performance because if we had only done exchange then every exchange would have invalidated all the copies continuously. Right. So we are saving on these invalidations by spinning on my locally cached variable. Okay. So how do I implement this using the exchange instruction? So let us see that.

So we want to spin on our local copy. Load the value. So I will say lockvar. I want to read this lock variable into my register. So load. So I will say load lockvar in R2. So lockvar's value will come and sit in the register R2 and you keep checking the register R2.

So is R2, if it R2 is 1, you keep looping. If R2 is 0, you move ahead. So if R2 is 1, branch if not equal to 0, R2 to try. So keep trying and this is my try label. Okay. So you are going to loop in this in your locally cached lock variable. So this has saved invalidations which could have happened, otherwise if we had already done exchange.

So I am not going to execute exchange right now. Once I see that R2 is equal to 0, we try to exchange. So I will try to set a 1 into the lock variable. So R2 is anyway 0, that's why we've come to this line of the code. So I want to make it 1.

So I'll simply say add 1 to the register 0 and write it in R2. So I'm doing R0 plus 1 going to R2. R0 has the value of 0 plus 1. So 0 plus 1, 1 goes into R2.

So R2 has got the value of 1. Now I will do an exchange. So this value and the lockvar. So these two, this goes here, this comes here. So that's the exchange I'm trying. This is the fourth line where I am doing the write. So I'll do the exchange R2 with the lockvar. Okay. So once I exchange, our job is still not done because at the time of exchange it could so happen that here, here I saw that the lock was free, right, and I started executing the add.

So when I was at this instruction it is possible that there is some other process in the system which has already acquired the lock. Hence it is necessary that I check again even after I finish the exchange, I need to check whether the lock was still free by this time. Okay. So I again need to check for the 0. So if it is not 0, if we are unlucky maybe somebody else acquired the lock.

So even here you check if branch is not equal to 0, keep looping again. So here we are going to spin on local copy, spin on your local copy of the lock variable. Once you see that it is 0, you try to exchange it. So you want to try to write a 1. You want to write a 1 into it. But while doing this you have to again check whether the value was still 0, because you came to the fourth line because it was 0 but in the meantime it may not be 0.

Hence you need to check it again, otherwise keep looping. So neater code is written here. So this is for locking and once you finish using, you have to release the lock by storing a 0 into the lock variable. So this is very straightforward. You simply have to say st lockvar with 0. So you have to write the value of 0 into that particular location.

So spinning on a cached copy is performance wise beneficial. Okay. Now we look at the other instruction LL and SC which is load linked and stored conditional. So spinning was okay. That is we have, we were doing the read and once the read said that the lock was free we went to the third line and did an exchange. So exchange was again a write and after the exchange I again checked whether the lock variable was still free by the fourth or fifth line of my program. Okay. Because in the meanwhile somebody would have changed it. Okay. So this exchange which we did it is again a write operation and in a multi core system it is likely that if one process succeeds to do the exchange there will be several who did not succeed.

So these failed attempts of exchanging, that is they tried to exchange and saw that the lock was not available. So we want to avoid these exchanges of those processes which were failing in my previous scenario, okay. So you do a loop on that, you check that the lock is free. So that is the read loop. After you see that the lock is free, you go to the exchange, but what I am saying is I do not want to do the exchange every time.

I want to do the exchange only if there is a chance of success, okay. So the failed attempts of exchange are going to do unnecessary invalidations. So this problem was solved by this pair of instructions called load link and stored conditional. So load link is equivalent to the reading loop, right. So I am going to read in a loop using the LL instruction and the exchange instruction in my previous slide will now be replaced by the store conditional instruction. How is this different? This is different because store conditional though it is a store instruction, it only succeeds if the lock was available.

That is, in the meanwhile if some other process has already acquired the lock then the store instruction will not execute. When the store instruction does not execute, it does not lead to invalidations, okay. So that is the beauty of this pair of instructions. Okay. So we are going to spin on the local copy and similar to what we did in the exchange and when we try to exchange, it is going to generate several invalidations. The failed attempts of exchange are unnecessarily creating these many invalidations.

So the modern processors are using a pair of instruction load linked and store conditional. The first instruction is a read, the second one is a conditional store instruction which writes the modified value into the lock variable and when does it write this? When does it write this? Only if other processors has not written to that location. If there was a processor which was succeeded in writing to that location then this store instruction will not execute. For example, here, here you derive that the lock was free, okay, third line. When you came to the fourth line you executed your fourth line and while doing this you found that the lock was locked, okay.

So your exchange was wasteful. I want to stop this wasteful exchange and therefore this exchange will be replaced by a store conditional. This instruction will only execute given that nobody in the meantime has changed the lock variable. We will see implementation of this in some other lecture but right now you can assume that there is some mechanism which implements this. Now when do I say my atomic exchange has completed? When both of them complete together, okay.

So they complete as a pair and not individually. The store conditional instruction whether it finishes or not can be easily identified using the return values or condition codes. So when you execute store conditional, see store instruction is different. Store instruction simply goes and modifies the value whereas store conditional will only store given that nobody in the meanwhile has changed.

So we have LL and we have store conditional. In the meanwhile, if any process changes the variable, okay. So I had the LL and then I do something in the middle and then we go to the store conditional. Between these two instructions if any other process has updated the lock variable then the store conditional instruction will not succeed. So its return value will indicate accordingly. So let us see how I can do this exchange using LL and SC. So I am again going to spin on my local cached copy and but I am not going to use the exchange instruction, I am going to use the LL and SC instruction, okay.

So we were doing a read first. So read will be replaced by LL. So I am going to read the lock variable into the register R2, okay. So read the lock variable into R2. What are

you going to check whether R2 is 0 or not.

If it is 0, you can move on if it is 1, keep looping. So branch if not equal to 0. So if R2 is not 0 you keep on trying. So this is my try label, okay. So keep going and trying to read the value. Once you see that R2 is 0, you have to write a 1 into it and then exchange that R2 and lockvar. So I am going to again do add destination is R2 and R0 has value of 0 and this is a immediate value of 1.

So I am going to do 0 plus 1 and write it in the register R2. Once we do this try the store, that is earlier it was an exchange instruction now I am writing store conditional. What am I storing R2 in the lockvar. So R2 where I wrote the value of 1 and lockvar is the variable.

I want to write this exchange or exchanging the values. Here I am going to do a store conditional. It is not an exchange. So store conditional will go and write a 1 into it provided the implementation says that nobody has changed the lock variable in the meantime. If somebody has changed, the store conditional will fail, it will not do the lock and it will return the success or failure code into the R2 variable. So R2 variable is going to bring the success or failure. So here in SC, if the return value is 0 means it has failed and if the return value is 1 means it has succeeded. So if R2 value is 0 means it has failed, so I will check for 0 branch if equal to 0. It was 0 I have failed the, failed the store conditional, what should we do? keep trying.

So you have to go back to try and here the first two is called the spinning. So you keep spinning to check the value then you set R2 to 1 and then try to write. Okay. So you try to lock. When you try to lock, in the meanwhile if somebody has already locked then you will get a failure code inside R2 and you will have to keep looping again. So this is how LL and SC as a pair can be used for doing this atomic exchange. Okay. So well, we read that and keep spinning until the value is 0 and once you get the value of 0, you want to try to write but in case of multiple processors if everybody tries to write a 1, one of them will succeed because in the meanwhile if the lock variable is changed by others my SC instruction will not succeed. And why will this happen because when multiple processes execute this SC. See SC is an instruction which will be executed by multiple, if there are multiple processes, trying to acquire a lock, only one of them will go on to the bus or reach the directory, get serialized and get access to that lock variable.

So the process which succeeds will complete the write and the other processes will continue to retry. So they will keep retrying, that is continue in this loop until they get the lock. So LL and SC, they succeed as a pair, they do not succeed alone, they succeed as a pair because only when both of them execute correctly, we are successful in

acquiring the lock. Right. So I hope it was clear it is a very interesting combination of these two instructions.

We will see how does the SC return a failure or a success code in some other lecture. Okay. This pair helps me to do the exchange. Now what should be there between these two pairs because we say that these two instructions together establish the task of atomic thing. So when I have two instructions, how many instructions should I write between them. Okay. So I had a LL instruction here and this LL then there were few more instructions and then we wrote the SC instruction. Can I write 10 instructions here? Can I write 100 instructions between them? Because you're saying well, they both will succeed as a pair and should I really bother how much work I do between those two instructions. Okay.

So we need to understand that eventually I want to acquire the lock. So I should not waste the time between these two instructions doing something which is not right now required. Again if SC fails, right, your store conditional instruction if it fails, we keep on trying again. So between LL and SC whatever work you did will get wiped off. Right. It is not going to be useful. So therefore it is advised that you should not change your program variables between these two, between LL and SC don't change important variables, you only do the work which is most necessary.

So between these two instructions what are you going to do you only change the variable which you are going to use to establish the lock. You don't change your data or program variables inside this loop. Because if it fails and I keep looping, changes will be there but may affect the correctness of the program. Okay. So overall what all instructions should you write between them. Definitely there has to be very few instructions because eventually you want to execute the sequential, sorry because eventually you want to execute the store conditional. The intermediate instructions are not guaranteed to be done atomically because they will be easily interleaved.

So I have LL-SC 1, 2, 3 instructions then I have another process which is also doing LL-SC, few more instructions. So these intermediate instructions ,they will again get interleaved. So it is not guaranteed that these middle instructions will happen all together. No, they will get interleaved and therefore I can't say that this middle part is a critical section.

So this is not a critical section. Your critical section will begin only after you acquire a lock. This part of the code is to acquire the lock. Okay. So these middle instructions, they get interleaved across different processes and they do not constitute a critical section. In case your store conditional fails then all these intermediate instructions if they

get executed at all, they should not affect the correctness of the performance because correctness of the program. Because if you are say writing an instruction which increments, suppose I wrote a plus plus which was one of my program variables, every attempt of this loop is going to keep on incrementing your a variable which is going to affect the correctness of your program.

So do not use or change program variables inside this loop. Okay. Don't make major changes in this segment. Do a small set of work. Only do changes to the register which you are going to use during the sequence, which you are going to use during the store conditional. Let them be simple register operations. Try to avoid memory operations because if you have a memory operation, it is going to add further delays and your gap between the LL and the SC is going to increase. Okay.

So keep the number of instructions small, so that eventually you should get a chance to execute your store conditional. So so that is the concept of LL and SC. We will now see two examples of how to do atomic swap and another type of an instruction using LL and SC. Okay. So what we are going to do. I have a register R4 and R1 has got an address of the memory and we want to do this, atomic swap,okay, similar to an exchange. So these are some additional examples so that you understand the concept of doing the same thing using LL SC. Okay.

So we want to do this exchange, R1 is actually the lock variable. R1 is storing the address of the lock variable. So what I will do? I will first try to read it using LL. So LL says you read this R2, so if I say 0 R1, it means R1's contents which is the address, any offset I want to add to it.

So this is the memory address and this is the destination register. Okay. So that is the destination register. So load from that lock variable into R2. Now you want to write a 1 or R4 should go there. So I will do a store conditional. I will store inside this, what will I store? I will store the value of R4. Okay. So I will store R4 inside this store conditional, back to back both of them is okay.

Once I do store conditional, SC is going to send a return value of failure or success. Okay. So failure will return a 0, success will return a 1 and where will this value come? It will come in my register which I have right now given as R4. So this success of failure value will come in R4 and it is going to overwrite the value I intended to use. So this is creating a problem. Hence before I use R4 with this I need to make a copy or keep a copy of that.

So I am going to make some change to this and I am going to keep a copy of R4 in the

register R3 as a temporary variable. So make a copy of R4 in R3, then execute this. Now I can safely use R3 here. Because let me use R3 as a temporary scratch pad variable. R4 retains the original value I am interested in. After SC, you are going to get the success code of R3 and if R3 is 0, that is it has failed, you have to keep on looping. So where should a loop start? Loop should start here at the first line and we will say branch if equal to 0.

If R3 is equal to 0, branch to the label try and keep on looping. Once you succeed this, you want to again regain your value which you have done the exchange. So the lock variables value came in R2, right, you did the loading here. So this R2's value should get copied to R4 because store conditional is going to move R4 to lock variable. So this is how we can implement the atomic swap using LL-SC instruction pair, okay. The first instruction of moving this value, this instruction is actually used to keep the R4 intact because SC is going to change this destination register, that R3 will get changed with the success or failure code and I would lose my value of R4.

Hence I first try to store R4 and in R3 and R3 is used as a temporary variable. So every time I loop, I make a fresh copy of R4 inside R3 and use R3 for store conditional instruction, okay. Let us do the next one using fetch and increment. Now fetch and increment says, go, read the value and add 1 to it. Suppose I have a lockvar. This lock var's address is given in R1. So I can use, I am just using different types of instructions to give more flavors to the idea.

So R1 is storing the address and our idea is fetch and increment. So you want to bring this value, do a plus 1 and then write the value back, okay. So this is what we want to do. Okay. So let us first read the value. So this is my lock variable. I am going to do a load linked 0 of R1, that is contents of the memory location pointed to by register R1 and read the value inside register R2.

Whatever is this value I want to do a plus 1. In the previous case, we wrote a 1 inside that. In the fetch and increment I am going to add a 1 to that value. So I need to add a value to this.

So I will use an assembly instruction to add 1 to R2. So I am going to do R2 plus 1 and let that answer come in R3. So R3 is the destination. So R3 is equal to R2 plus 1. Now this R3 I am going to write inside the lock variable using store conditional. So store conditional write R3 in the lock variable. Lock variable is 0 of R1, that is address is given in the register R1.

Once you do this, what happens success of failure code will come in R3. If R3 is 0,

failed, if R3 is 1, success. So if we have failed, we have to keep trying. So I need the try label here and I will say branch if equal to 0, that is if we have failed. If R3 is equal to 0, go and try again. Okay. So branch if equal to 0, keep trying. See here R3 will be 0 for a failure and R3 will be 1 for a success.

This is saying R3 is equal to value of R2 plus 1 because we are doing fetch and increment. Okay. So the same code is written here. So first you do the load linked, increment the value, attempt store conditional, success move on, failure then keep on looping. Okay. So this is how I can use LL and SC for fetch and increment. So we tried variety of options, atomic exchange, exchange instruction and fetch and increment using the same combination. So your hardware could be providing you an atomic compare and swap type of instruction or if it provides LL and SC, you will be able to use them to do the same task. Okay.

So LL and SC is a more generic type of a pair of instructions to acquire the lock. Okay. So with this we have got a feel of what are these uninterruptible instructions. So we stop this lecture here. Thank you so much.