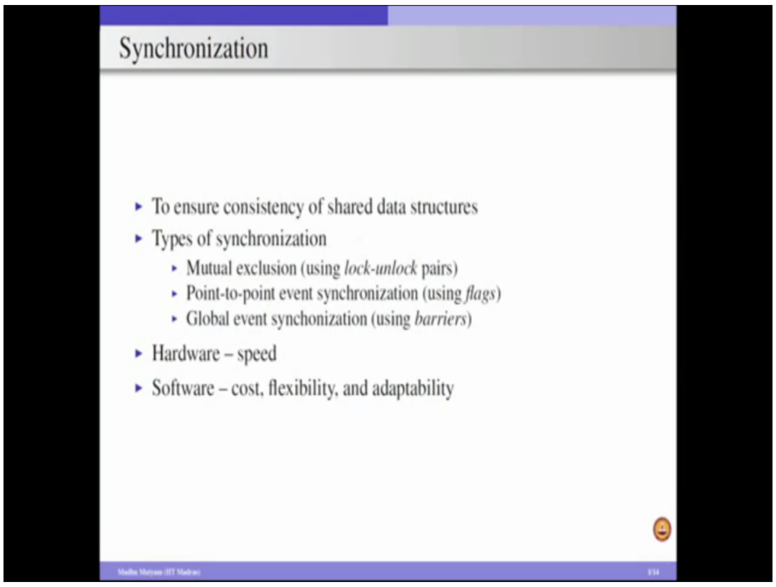
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 09
Lecture – 31
Synchronization

So, when we are working with multi core systems, we have to deal with three main issues. One is cache coherence, the second one is synchronization and the third one is memory consistency. So, as part of the last week lectures, we covered cache coherence problems and we also discussed three state cache coherence protocol to deal with cache coherence problems. And in this week we are going to discuss synchronization as well as memory consistency. And the material whatever I am going to cover for their synchronization and memory consistency is taken from "Parallel Computer Architecture" book by Culler Jaswinder Pal and Anup Gupta. So, why do we have to worry about synchronization?

(Refer Slide Time: 00:58)



Synchronization

- To ensure consistency of shared data structures
- Types of synchronization
  - Mutual exclusion (using *lock-unlock* pairs)
  - Point-to-point event synchronization (using *flags*)
  - Global event synchonization (using *barriers*)
- Hardware – speed
- Software – cost, flexibility, and adaptability

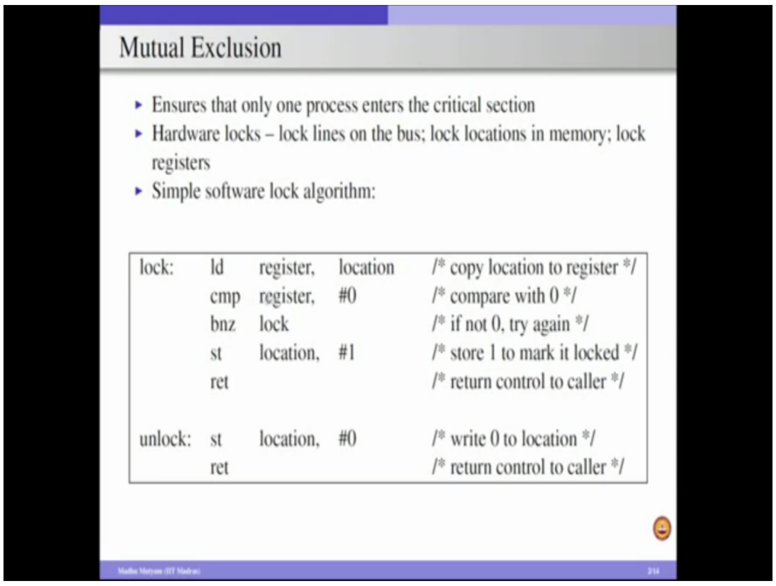
So, the synchronization ensures that the consistency among shared data structures. When we are running multithreaded application on a multi core system and these multi threads may share some set of data structures and so on. So, we have ensure that at any point of time only one thread will access the critical section, and so no two processes should enter the critical section. And for that we have to come up with set of synchronization mechanisms.

Also in our multithreaded program execution, so may be some threads are waiting for some event to be occurred, and some other thread is going to produce a result for that particular event and again based on that event. So, the other threads which are waiting for this event to occur can proceed further. And similarly, there may be a scenario where all the threads have to wait for some event to occur, and then proceed further and also again we need synchronization mechanism.

So effectively, so there are different types of synchronizations we have to consider one is a mutual exclusion, second one is point to point event synchronizations and the third one is the global event synchronization. And for the mutual exclusion we generally use lock and unlock pairs, and for point to point event synchronization, we use flags. And finally, for the global event synchronization we use barriers. So, as part of this course we concentrate on what are the hardware primitives, we can support for ensuring the synchronization mechanisms in multi core systems.

So, we can actually provide hardware mechanisms as well as software mechanisms for providing the synchronization, but there are advantages and disadvantages with each of these things. So, obliviously with the hardware we can get better performance, but with the lack of flexibility hardware mechanisms may not be useful in all scenarios, but whereas, in the case software. So, in the case of software, the performance may not be that much significant, but will have greater flexibility and adaptability to the scenarios and so on. And also the cost wise it will be cheaper to implement synchronization mechanisms in software.

(Refer Slide Time: 03:28)



Mutual Exclusion

- Ensures that only one process enters the critical section
- Hardware locks – lock lines on the bus; lock locations in memory; lock registers
- Simple software lock algorithm:

```
lock:    ld    register,  location    /* copy location to register */
         cmp   register,  #0          /* compare with 0 */
         bnz   lock                   /* if not 0, try again */
         st    location,  #1          /* store 1 to mark it locked */
         ret                          /* return control to caller */

unlock:  st    location,  #0          /* write 0 to location */
         ret                          /* return control to caller */
```

So, we first start with mutual exclusion. So here the mutual exclusion says that when two processes are competing to enter into a critical section, only one process will be allowed to enter and the other process will be stopped from entering into the critical section. So, for that we have to come up with lock and unlock mechanisms, either by using the hardware or by using the software.

So, in the older systems typically the hardware locks were used in terms of lock lines and so these lock lines are separate from the address and the data lines in our processor. And whenever any process wants to enter into a critical section, it sets the lock line. Once the lock line is set if any other process wants to enter into the same critical section, then it will just wait until this lock line will be reset. So, as a result like we can prevent any other process entering into the critical section, and the process which is in the critical section, whenever it wants to come out of the critical section then it resets the lock line. So that one of the processes, which are waiting to enter into the critical section will set the lock line and then it will go to the critical section.

So, this is a simple method, but it is not scalable because at any point of time we can have a limited number of lock lines in our system. So, as a result not more than that many lock lines can be used and as a result, we cannot implement this method for larger multicore system, where large number of threads are there and they want to enter into the critical section, then we cannot apply using this particular mechanism. And also we can consider lock locations in memory or lock registers in place of lock lines.

In the case of software, we can come up with algorithm which is a routine, which is going to acquire a lock, and then there is a routine which will unlock this particular thing. So, we can write a sequence of instructions in our program, and so this set of instructions are going to acquire a lock and release the lock. And a simple example we can consider here is we can see here, there is a load instruction we are loading some data from a memory location to a register, and we compare the register with the value 0. That means like weather the value stored in the location is 0 or not, we are going to check.

If the value is not equal to 0 that indicates that someone has already acquired the lock and then we have to again go back to this, and then execute this sequence of instructions. And whenever if the location is storing the value 0, so as a result this condition will be false and as a result, we can just go to this store instruction and we store value 1 into that memory location. And once we complete this then automatically we say that the lock is acquired, and the process which acquired the lock and enter into the critical section and it executes the sequence of instructions that are there in the critical section.

Whenever it completes its execution then it can reset the lock by using this storing value 0 on to this memory location so by using a store instruction. So that any other process which is trying to enter the critical section will see that now the lock, the location has the value 0, so that it can acquire the lock and then proceed. So, this is a simple piece of code written in software to achieve lock and unlock pair, for mutual exclusion.

So, here the overall idea is, so there will be a lock variable stored in some memory location and we have to load that value on to a register, and we have to check whether the variable has value 0 or not. If it is 0 that indicates that no one has acquired the lock, so that the process which loaded this value on to a register can acquire this lock. And to say that this process has acquired the lock it has to store a value 1, on to the memory location. So that if any other process is trying to enter into the critical section or trying to acquire the lock will see that the location has value 1, and it cannot proceed further because of this loop here, because of branch not equal to 0 lock is there, so the second process which tries to enter into the critical section will fail acquiring the locks, so it will be in this loop.

Once the process which has the lock and it wants to release the lock then it stores the value 0 on to the location. And though this is a simple code actually there is a problem in this particular piece of code. The problem is so let us say process p1 is trying to acquire the lock so process p1, first loads the value from the memory location to the register, then it compares. And when it compares let us assume that the location has a value 0, so as a result register has the value 0 and this compare statement is correct. So, as a result this condition will be false, so it tries to go to this second instruction that is a store location one.

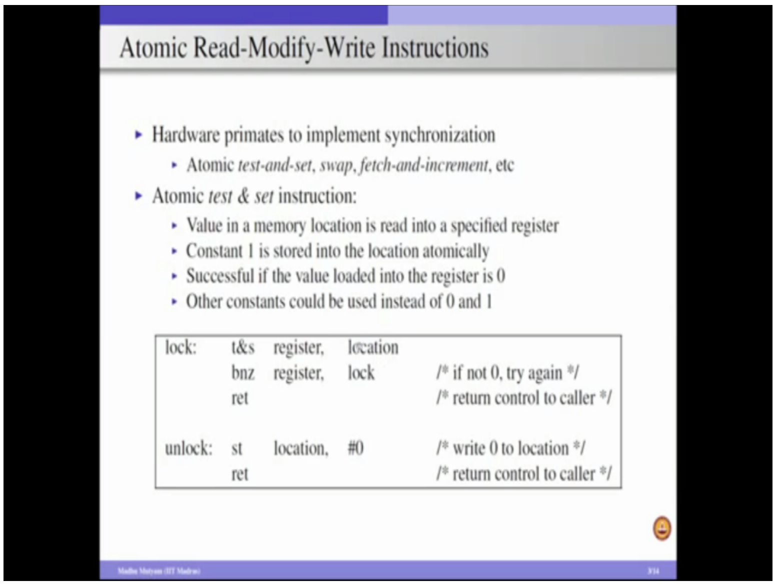
But assume that simultaneously another process p2 also trying to enter into the critical section and trying to acquire this lock. It also reads the value from this location to a corresponding register, and it also checks the register whether the value in the register is 0 or not. And it also finds that the value in the memory location is 0, so as a result the register also has the value 0, and this condition is false. So, as a result it also goes for this store instruction.

Now process p1 and process p2 both are executing the store instruction, and both are trying to go for store location 1. So, they are trying to write value 1 into the memory location though this memory location is unique for both the things, but because process p1 may write first and process p2 may write second and so on, but does not matter. By the time you come here then automatically the process is going to acquire the lock because this load, compare and store these three operations are independent operations. So, as a result multiple processes can execute this instruction simultaneously, and as a result multiple processes can acquire the lock.

So, as a result at the end both processes p1 and p2 are going to enter into the critical section, and that actually defeats the whole purpose of having this particular routine because we want to ensure that only one process enters the critical section by using this code. But actually this code is not actually doing that. That is mainly because so we are loading at once, comparing second and then writing after that. So because these three operations are independent operations, and these are not formed atomically and as a result we have this problem.

So that means we have to come up with an atomic operation to load the value from the memory location, to update the value and to compare the value and so on. So, we need a single instruction that forms all these three operations, so that any process which is actually executing that atomic operation, atomic instruction will ensure that it will get the lock and no other process will enter the critical section.

(Refer Slide Time: 11:30)



Atomic Read-Modify-Write Instructions

- Hardware primates to implement synchronization
    - Atomic *test-and-set, swap, fetch-and-increment*, etc
- Atomic *test & set* instruction:
    - Value in a memory location is read into a specified register
    - Constant 1 is stored into the location atomically
    - Successful if the value loaded into the register is 0
    - Other constants could be used instead of 0 and 1

```
lock:    t&s   register,   location
         bnz   register,   lock        /* if not 0, try again */
         ret                           /* return control to caller */

unlock:  st    location,   #0          /* write 0 to location */
         ret                           /* return control to caller */
```

So, in order to do that we have to actually have atomic read, modify, write instructions in our ISA. Again as I mentioned earlier we have going to deal with the hardware support to ensure this synchronization as the result, we have to look at what is the support we can get from the instructions and architecture to achieve this mutual exclusion or the synchronization mechanisms. So, as a result we need to have read-modify-write instructions in our ISA. So that whatever software routine we write for acquiring the lock and releasing the lock, they can use this hardware primitives are these atomic instructions supported by ISA and as a result we can achieve mutual exclusion.

So, we need hardware primitives such as test and set, swap fetch and increment etc type of instructions in our ISA. Effectively first step is we load the value from memory location to register, we modify the register content and finally we write this register content back to the memory location because the memory location is actually holding this synchronization variable. So, as a result once we performed this three operations one after another in atomic way, so there is no other process which is going to interfere with this and then update. And as a result we can prevent multiple processors entering into the critical section by preventing multiple processors acquiring same lock.

So, we just consider a test and set instruction and what are the sequences of steps that happen as a part of this atomic instruction. So, in this test and set we are going to load the value with test and set it, so first we will, the value in a memory location is read into specified register in our processor. And the constant one is stored into the memory location automatically. While we are reading the value from the memory location to a register we are also writing a value 1 to the memory locations. So, these two will happen simultaneously, we said that this test and set operations is set to be successful only when we load the value 0 into our register.

For example, if the value in the memory location is 1 that indicates that there is some other process, which has already acquired this lock. So, as a result when we get value one from the memory location, or test and set operation is not successful for this particular process. So, whenever any other process acquires the lock then the memory location is going to have a value one. So that any other process subsequently trying to acquire the same lock, then it reads value one into its register. And as a result this test and set operation for the subsequent process is actually unsuccessful.

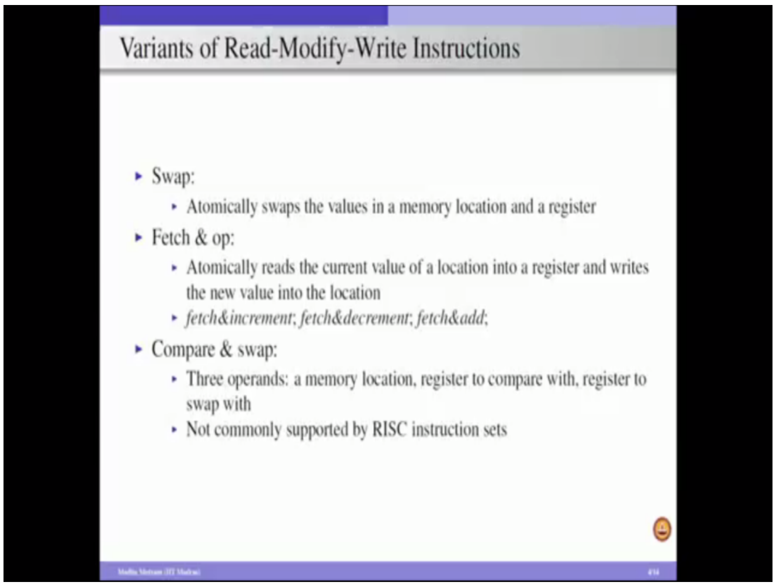
And this can be applied for by considering any other values rather than just 0 and 1. But overall, so this test and set instruction is going to load a value from memory location to a register simultaneously it writes a value to memory location. And it checks whether the register content is 0 or not, if it is not 0 then the test and set instruction is set to be failed. If it is 0 then test and set instruction is set to be a true, as a result the corresponding process which executed this test and set instruction acquires the lock. So, we are testing and then setting, both are happing simultaneously, as a result this is called as an atomic instruction.

Now, we will see how the previous code is modified by using this test and set, and so that we actually acquire the lock, and we prevent multiple processes entering the critical section. So, previously we have a load instruction, compare instruction and store instruction, three independent instructions, but now we have only one instruction that is test and set instruction. And remember this test and set instruction is supported by our underlying ISA.

So, we execute test and set, here we are reading a value from location in a memory to register, at the same time we are setting the value in this location. And now we are checking the register content whether the register content 0 or not, if it is not equal to 0, then we are actually going back to this and then we are trying to again acquire the lock. And if it is 0 then lock is acquired. So that this process is going to enter critical section, and it is execute the instruction section.

Once it comes out of the critical section, then it execute this unlock instructions so that will be like it is going to store value 0 into this memory location. So that any other process which is actually waiting for this lock to be acquired, now can succeed with this condition and then it can acquire the lock. So, these are simple instructions using which we can ensure the mutual exclusion.

(Refer Slide Time: 17:02)



Variants of Read-Modify-Write Instructions

- Swap:
  - Atomically swaps the values in a memory location and a register
- Fetch & op:
  - Atomically reads the current value of a location into a register and writes the new value into the location
  - *fetch&increment*; *fetch&decrement*; *fetch&add*;
- Compare & swap:
  - Three operands: a memory location, register to compare with, register to swap with
  - Not commonly supported by RISC instruction sets

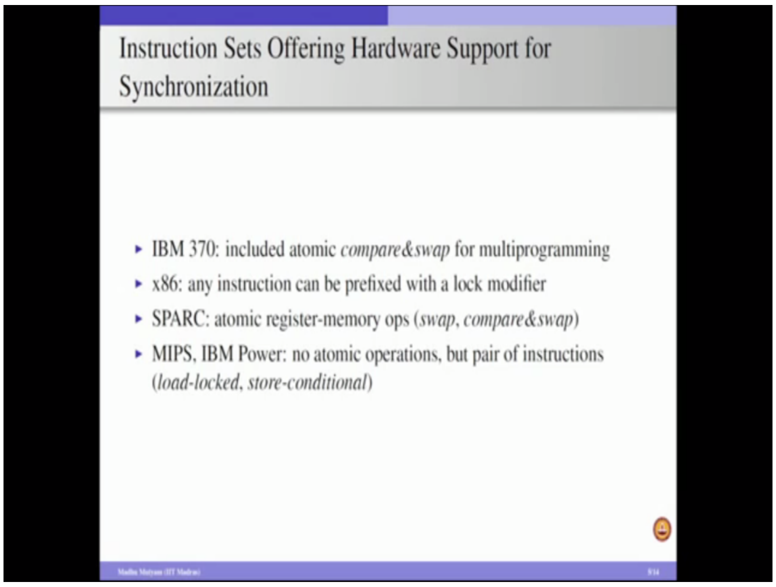
So similarly, various ISAs have the support for different type of these atomic instructions. One is swap instruction and the other one is fetch and operation instruction, the third one is compare and swap. So, in the case of swap we swap the values between the memory location and register, and this swapping is going to happen simultaneously. Similarly, the fetch and the operation instruction will have a fetch and increment, fetch and decrement, fetch and add, different type instructions, where we going to fetch the value from memory locations and simultaneously we are going to perform an increment, or decrement, or addition. Effectively this increment or decrement or addition will be like modifying the content. And finally, we store this value back to the memory location. All these things are going to happen simultaneously or in an atomic sense.

And in the case of compare and swap also again here we require a memory location, a register to compare with and a register to swap with. We read a value from memory location, we compare this value with some register content and if the condition is true then we are going to swap the contents of some other register with this memory location so that we acquire the lock. So again so here typically this instruction requires three operands and RISC type of instructions are not actually supporting these three operand instructions.

And as a result we cannot use compare and swap in RISC type ISAs. And also again if you see test and set, and swap fetch operation or compare and swap all these instructions are actually performed atomic sense. And also there are so many sequence of steps we have to do as a part of this atomic instructions and that is actually going to take significant amount of time, which is also going to increase our CPI clock cycle per instruction.

And whenever, we have to deal with RISC type of ISA's. The major advantage with RISC type of architecture is typically all the instructions are simple instructions, and so they are going to take less amount of time to execute and our pipeline stage time will be simple if you are dealing with RISC type of architectures, but now if you want to consider any of these atomic instructions in our RISC type of architecture then that is going to put so much burden. As a result we have to look for some other alternative type of instructions in RISC type of ISAs to deal with mutual exclusion.

(Refer Slide Time: 19:42)



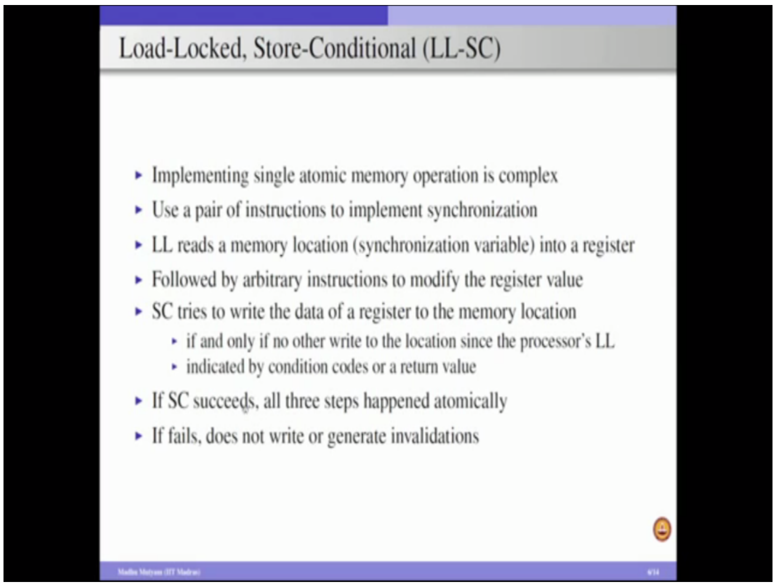
Instruction Sets Offering Hardware Support for Synchronization

- IBM 370: included atomic *compare&swap* for multiprogramming
- x86: any instruction can be prefixed with a lock modifier
- SPARC: atomic register-memory ops (*swap*, *compare&swap*)
- MIPS, IBM Power: no atomic operations, but pair of instructions (*load-locked*, *store-conditional*)

So, these are different type systems they are actually supporting different atomic instructions in their system. For example, IBM370 is considering compare and swap, x86 ISAs typically consider any memory instructions as an atomic instruction with a prefix before that instruction, a lock prefix will be considered. So that any instruction with this lock prefix is said to be considered as an atomic instruction.

Similarly, in SPARC type of machines we can consider swap or compare and swap and but in the case of MIPS and IBM power type of systems we actually consider non atomic instructions, but we achieve this mutual exclusion. So that will be done by using a pair of instructions, they are called as load lock and store conditional. Load lock are also called as load linked, so we execute these instructions load lock and store conditional independently. And if second instruction successful then we can say that the corresponding process has acquired the lock. So, in other words we actually execute these two instructions separately, but we achieve the mutual exclusion, that we are going to discuss in the coming foil in detail.

(Refer Slide Time: 21:06)



Load-Locked, Store-Conditional (LL-SC)

- Implementing single atomic memory operation is complex
- Use a pair of instructions to implement synchronization
- LL reads a memory location (synchronization variable) into a register
- Followed by arbitrary instructions to modify the register value
- SC tries to write the data of a register to the memory location
  - if and only if no other write to the location since the processor's LL
  - indicated by condition codes or a return value
- If SC succeeds, all three steps happened atomically
- If fails, does not write or generate invalidations

So, this is actually considered in a mix processors and this is called as load locked or load linked, and store conditional is a two instructions put together constitute this atomic execution, and it achieves the locking. So, this load locked is not same as a conventional load instruction. So, we have going read a value from memory location into a register and at that same time, we are going to set some flag associated with this memory location that is called as lock flag. So, once we perform this then we say that the load lock is completed.

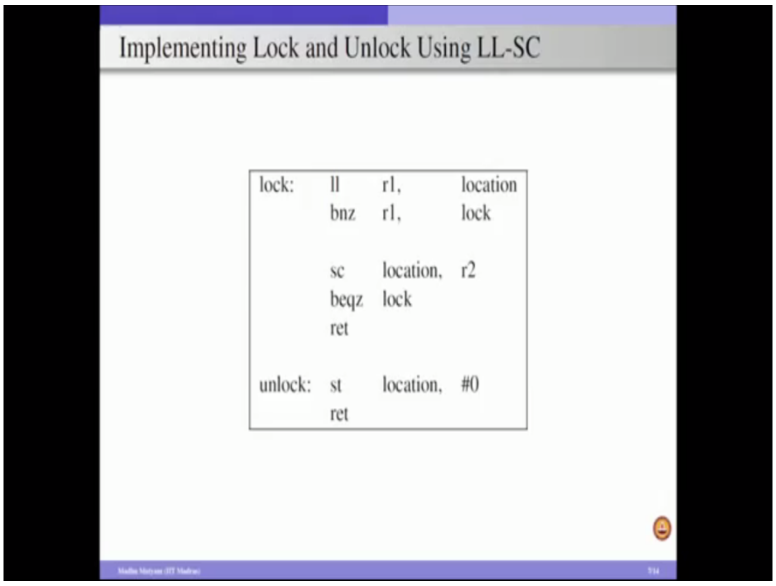
And after this we can perform arbitrary number instructions operations to modify thus register value. And again we are not immediately performing this store conditional, so store conditional can be performed after some time after performing load lock. So, here once we perform this arbitrary instruction execution to modify this register value, then will go for this store conditional. But this store conditional also again is not same as the conventional store instruction, the conventional store instruction is going to write some value to a memory location. But the store conditional is going to be performed only when some condition is set to be true. So that is what here it is going to do.

So, store condition is going to write some data of register to the memory location, if and only if there is no other write to the particular location happened. Since, this particular process has performed this load lock. So, in this discussion, so we can interchangeably consider process or processor. Now, here consider a scenario where we have two core system where processor p1 perform load lock operation. So, it reads a value from memory location to its register and also it sets the corresponding lock flag register in its local clash. And after that the processor p1 tries to go for store conditional operation. Mean while let us say if processor p2 also perform load lock operation on the same memory location, and it reads value from the location to its register. And also it sets lock flag register in its local cache.

Now if processor p2 actually performed store conditional before processor p1 is going to perform the store conditional, then automatically using this log flag mechanism processor p1 knows that there is some other write happened to the same memory location. So, as a result it has to again go for load lock instruction execution. So that is what it says here. So, store conditional set to be performed only when, there is no other processor which performs write operation to the same memory location, after this particular processor which performed its load lock, because we are performing this load lock and store conditional not an atomic sense. So, there may be some other processor which performed operations between this load lock and store conditional. So that we have to check and if no other processor performed any write operation to this memory location then we can go ahead with this store conditional and corresponding processor is going to acquire the lock.

If SC is succeeded that indicates that there is no other intermediate write from any other processor to this memory location. So, as a result semantically we achieve this atomic execution of read modify write. And if it fails then we have to again go back to the load lock, and then we have to execute. And also again the way it is implemented is that we actually set the location flag in our local cache, and we always have to do is we have to check this flag. And if the flag bit is reset that indicates that someone else has already written to the memory location, and as a result we have to again go ahead with LL operation and we do not have to generate any invalidation signals. So, as a result this is the simple method and we can perform this atomic operation using non atomic instructions and we acquire the lock, if our store condition is succeeded.

(Refer Slide Time: 26:10)



Implementing Lock and Unlock Using LL-SC

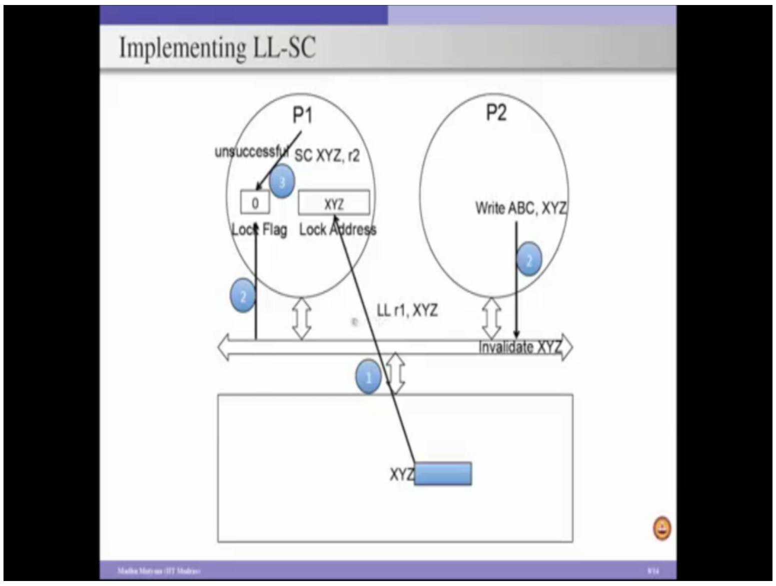
```
lock:    ll     r1,         location
         bnz    r1,         lock

         sc     location,   r2
         beqz   lock
         ret

unlock:  st     location,   #0
         ret
```

So, this is a piece of code that achieves locking and unlocking using this load lock and store conditional. So, we can see here we have ll instruction this is a load lock or load link instruction and it is going to read value from a memory location to a register r1 and at the same time it is also going to send the lock flag associated with this particular location in its local cache. And then it checks whether the location is already having a value 0 or not. If the value is 0 then it can proceed with modifying the contents of the register if it is not 0, then that indicates that some other processor already acquired the lock then it has to go back to this. And again it has to read the value from the memory location.

If it is successful in completing this ll operation and successful in completing this condition then it will go for this store conditional. Whenever it goes for store conditional again it checks whether the flag bit is reset or not. So, if there is a change in the flag bit then automatically store conditional is not set to be successful, then again we have to go for this lock. So, that is what we are doing here, so store conditional location r2. Here what we are doing is we are writing contents of some other register to this location and at the same time we are checking the lock flag register or lock flag bit. If the bit is equal to 0, that means someone else reset this the lock flag bit, then we have again go back to this lock and then we have to start with ll again. If it is not then automatically no one has written to this memory location, so as a result we can acquire the lock and we enter the critical section.

And whenever the process completes executing in the critical section then it is going to release the lock by writing value 0 to the location. This unlock is same as our previous the mechanism whatever we discussed earlier but this locking mechanism is different. Rather than considering a test and set or swap or compare swap type of atomic instructions, now we are actually performing these operations by using two independent instructions one is load lock the other one is store conditional. So, we will see with an example how this lock flag mechanism is implemented.

(Refer Slide Time: 28:48)



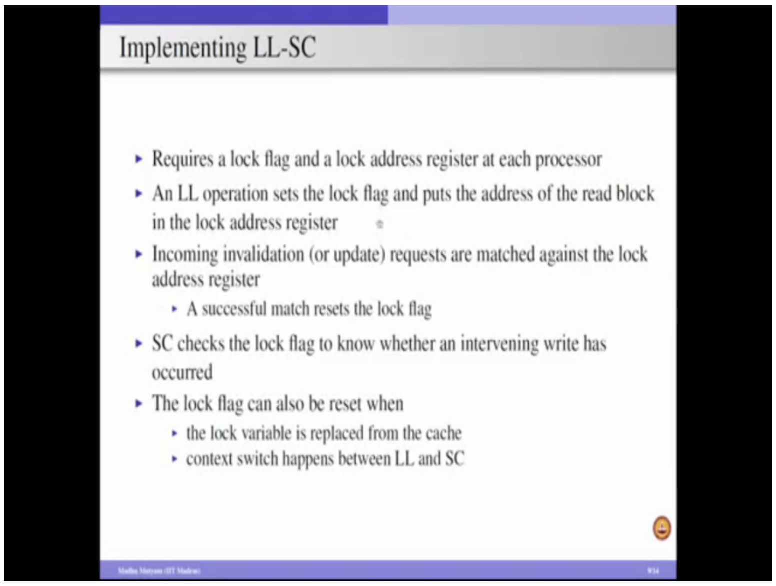
So, consider two processor system in our multicore and processor p1 is actually trying to acquire lock on this memory location. Let us say this is the variable which is stored at a address xyz. Now, we want to acquire the lock on that particular variable, so we read this value to our register in processor p1. And we have support of lock flag bit, as well as the lock address register. So, we store this address into this lock address register and we set this lock flag bit. Whenever we are performing this ll operation, if ll operation is successful then we are going to store this address in the lock address register. Also we set this lock flag bit.

Now, we will see if there is any other processor which is going to update this memory location, if it is so then what is going to happen? Because anyway our multicore system has the support of cache coherency, now if any processor is trying to write to a location it has send an invalidation signal on the bus which is connected to this processor. Again we are here assuming the invalidation based protocol. So because invalidation based cache coherency protocol is implemented in this particular system, whenever we are going to write then we are going to send an invalidation signal for this particular address.

So, all the cache controllers connected to this bus will snoop on the bus and they will check in there caches, to see if weather they have the corresponding addressed location in them or not. If there is max then they have invalidate because here the xyz is there and processor p1 actually storing this particular value in this lock address. So, it is going to invalidate and this will be indicated by this resetting this lock flag bit.

Now, processor p1 again wants to go for this store conditional because previously executed the ll operation. Now, it is going to go for SC to acquire the lock, when it is goes for SC, SC operation said to be unsuccessful because the lock flag bit is reset now. So, as a result again it has to go back to this ll operation, and again it has to perform this ll operation. And then again it has to go for SC operation if it wants to acquire the lock. So, this is the whole set of sequence of operations that happen whenever we are performing locking mechanism using this ll and SC instructions.

(Refer Slide Time: 31:38)



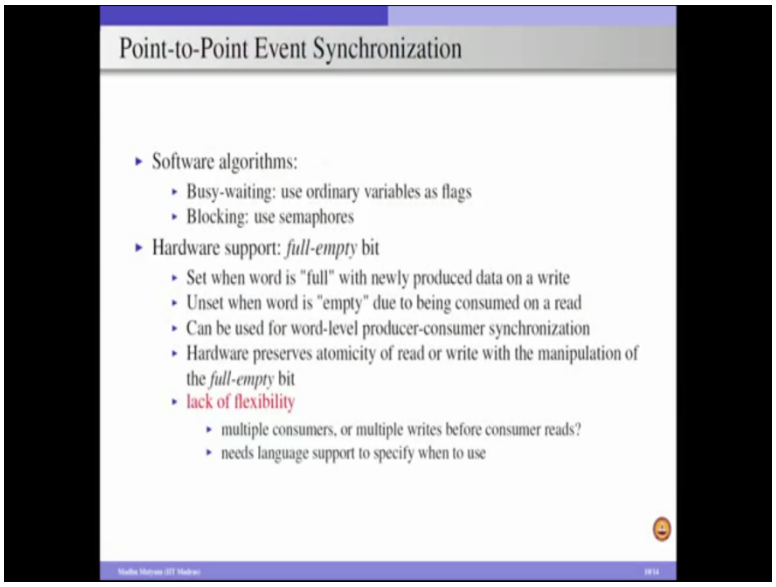
Implementing LL-SC

- Requires a lock flag and a lock address register at each processor
- An LL operation sets the lock flag and puts the address of the read block in the lock address register
- Incoming invalidation (or update) requests are matched against the lock address register
  - A successful match resets the lock flag
- SC checks the lock flag to know whether an intervening write has occurred
- The lock flag can also be reset when
  - the lock variable is replaced from the cache
  - context switch happens between LL and SC

So, whatever I have mentioned previously is represented in this foil here. So, we require lock flag, as well as a lock address register at each processor for each memory location on which we are going to acquire a lock. And ll operation sets the lock flag and puts the address of this memory location into this lock address register, and any incoming invalidations for the same address will invalidate this lock flag or by reset this lock flag.

If there is no incoming invalidation signal for this particular address, then we can go ahead with SC operation, and so that this processor is going to get the lock on that particular memory location and it can enter the critical section. So, here is not that just a invalidation signals, which will reset the lock flag, but the lock flag bit can also be reset under other scenarios, where so if the cache block which actually holds this the lock address value is evicted from the cache, then also we are going to the reset the lock flag bit. So, that again processor has to go for the ll operation. Similarly, whenever there is a context which happen then we have to reset this lock flag. So, this is about the mutual exclusion, now we are going to see the other one which is event synchronization mechanism. Here event synchronization can happen on point to point basis or on a global sense. So, point to point event synchronization, we are going to use the flag bits, and for the global synchronization we have to use the barriers. First we will discuss this point to point event synchronization.

(Refer Slide Time: 33:45)



Point-to-Point Event Synchronization

- Software algorithms:
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores
- Hardware support: *full-empty* bit
  - Set when word is "full" with newly produced data on a write
  - Unset when word is "empty" due to being consumed on a read
  - Can be used for word-level producer-consumer synchronization
  - Hardware preserves atomicity of read or write with the manipulation of the *full-empty* bit
  - lack of flexibility
    - multiple consumers, or multiple writes before consumer reads?
    - needs language support to specify when to use

So, here this point to point event synchronization can be implemented by using software algorithms, which are mainly classified into two groups. One is busy waiting the other one is blocking. So in the busy waiting typically, so all processors which are actually waiting for some event to happen will just execute in a loop, infinite loop. And whenever the event happens then automatically, the processors which are busy waiting on this event will come out of the busy wait and then they can proceed further.

Now, here in this busy wait typically what happens is the processor cycles will we wasted because the corresponding process is actually busy waiting. So, in order to eliminate this wastage we can actually go for the other type of mechanism that is blocking. So, here a process, which checks with are event is occurred or not. If the event has not yet occurred then it will block itself from execution, so that this processor can be given for some other process. And whenever this event completes then automatically it sends a signal so that all the processes which are blocking they can now come for ready state and they can execute further.

So, here the busy waiting is going to waste processor cycles, but if the busy waiting is not too much then automatically it is not a good idea to go for blocking because blocking is effectively a context which happen. So, there are trade-offs positive and negatives for both the things. And in the case of busy waiting typically, we can consider some variables or flag variables or flag bits on which the process is going to busy wait. And similarly, for the blocking we use the semaphore mechanisms and typically this semaphores and this busy waiting mechanisms so on, will be discussed in operating systems course. So, from the hardware point of view, we can actually consider full and empty bit. So, using this full and empty bit we can achieve this point to point event synchronization. For example, consider a producer and consumer scenario where producer is going to write a value to register and a consumer is going to read the value from that register, or the buffer.

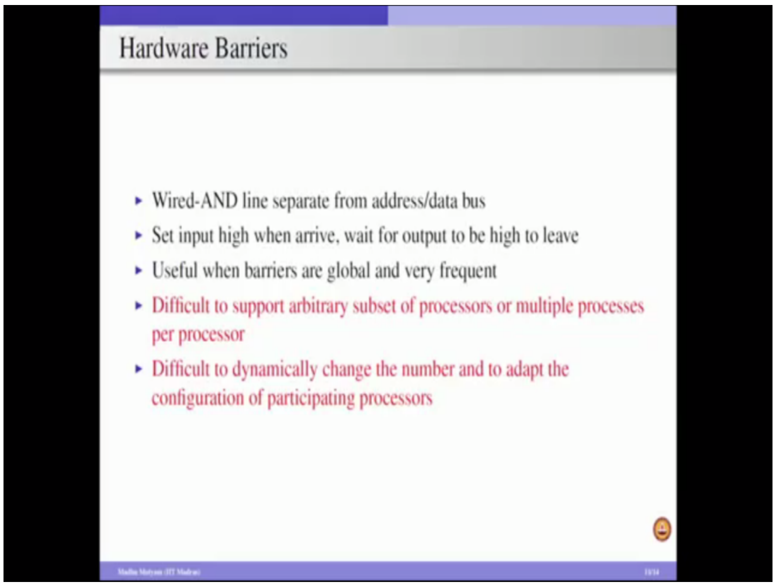
So, consumer cannot read the value from the buffer unless producer is producing value to that particular buffer. So that means whenever the producer is produces the value to this buffer then it is going to set the corresponding bit, that is a full empty bit. And the consumer always looks at this full empty bit, if the bit is set that indicates that the value that is there in the buffer is the valid value so that the consumer is going to consume it. If the bit is the reset that indicates that producer has not yet produced the value to this buffer, so that the consumer has to just wait. So that is what the whole idea.

So, we set when the word is full with the newly produced data on a write, and unset when the word is empty due to consumer process, which is actually consume this particular data. So, this can be implemented at ((Refer Time: 37:11)) word level granularity or it can be implemented at bigger buffer level and so on. So, this can be used for word level producer consumer synchronisation. Producer writes consumer consumes and this is synchronised by using this full and empty bit.

So, effectively because of this full and empty bit we can ensure atomicity of a read or a write with the manipulation of this bit because read cannot happen unless producer is writing the value to that location and setting the full and empty bit. Similarly, if the consumer is not consuming the value then it is not going to reset this full and empty bit. So, as a result producer is not going to produce a new value in to this buffer location. So, as a result we can ensure the proper synchronisation between the producer and consumer just by using this full and empty bit, but again so it is not providing enough flexibility unlike these software mechanisms.

For example, consider a scenario where a single producer is there and multiple consumers. Now when producer is producing the value, it is going to set this full and empty bit, but now we have multiple consumers which want to consume the value from this buffer. Now how do we reset? If we are going to reset after one consumer is consuming this value, then immediately producer can produce a new value. So that the other consumers they cannot consume the previous value and so and so. As a result it is not providing flexibility if you are going for the hardware mechanism.

(Refer Slide Time: 39:11)



Hardware Barriers

- Wired-AND line separate from address/data bus
- Set input high when arrive, wait for output to be high to leave
- Useful when barriers are global and very frequent
- Difficult to support arbitrary subset of processors or multiple processes per processor
- Difficult to dynamically change the number and to adapt the configuration of participating processors

Now finally, we consider the barrier mechanism and this barriers also again we can implement either at the hardware level or the software level. In the case of hardware level, we can consider a wide and line, which is separate from the address line and the data lines. Let us say we have 4 core system and we are going to apply barrier, for this 4 core system where four threads running on this four cores and we have to ensure that all these threads have to come to a particular point and synchronise then they can proceed further to execute the remaining set of instructions.

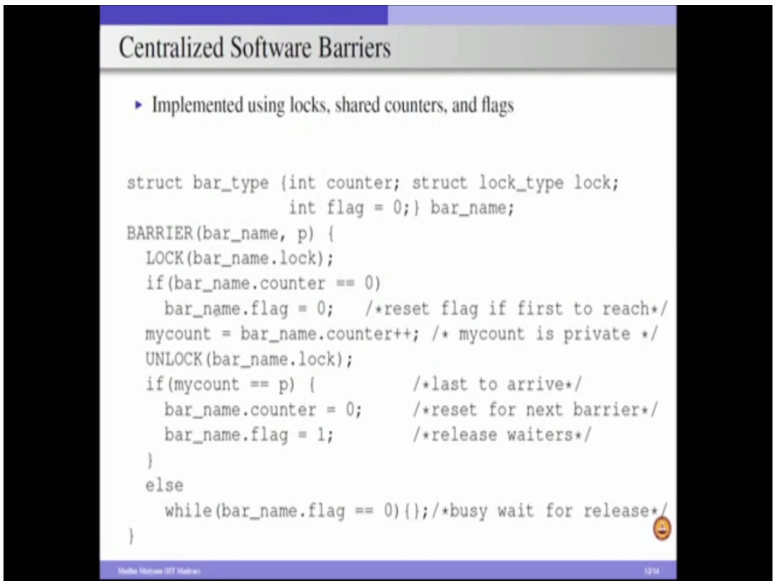
So, it means in order to achieve this we are going to put a barrier, and so even if three processors come to the barrier, then they cannot proceed beyond this barrier unless the fourth one comes. So that means, when we have a barrier instruction in our piece of code so none of the processors will proceed the barrier unless all the processors come to the barrier. So, this can be achieved by using this hardware mechanism by considering wired-AND line and here wired-AND line is initially having a value 0. And the value of this line will become one only when all the inputs to the wired-AND line will be one.

Now, consider a scenario where we have four processes and only one comes to this the barrier signal. So, it is going to set the value one, but because the other three processes have not come. So as a result their value will be 0. So the resultant value on the wired-AND line will be 0. So, process which comes to this barrier signal will not proceed further, and it waits for all the other three has to come. Now, let us say if all four have come then everyone is going to send a value one on to the wired-AND line because this four have come, so the outcome on wired-AND line will be one. So, as a result they can proceed further. So this is simple method, but again it provides no flexibility.

In other words for example, if you have more number of cores then we cannot actually scale up this design for larger core count. And also another thing is let us say if you are going for barrier signal for only fewer processor in our system, then also we cannot use this particular mechanism. So, it is difficult to support arbitrary subset of processors or multiple processes per processor in our system. And also it is difficult to change dynamically the number of processors, which are actually involved in the barrier.

For one piece of code, let us say four processors are involved in the barrier, and for the second piece of code may be like eight processors are going to involve in the barrier. So, we cannot adopt to these dynamic change in conditions if you are going for this hardware barrier mechanisms. So, we have to go for the software barrier mechanisms and that we are going to discuss now.

(Refer Slide Time: 42:38)



Centralized Software Barriers

- Implemented using locks, shared counters, and flags

```
struct bar_type {int counter; struct lock_type lock;
                 int flag = 0;} bar_name;
BARRIER(bar_name, p) {
  LOCK(bar_name.lock);
  if(bar_name.counter == 0)
    bar_name.flag = 0;   /*reset flag if first to reach*/
  mycount = bar_name.counter++; /* mycount is private */
  UNLOCK(bar_name.lock);
  if(mycount == p) {          /*last to arrive*/
    bar_name.counter = 0;     /*reset for next barrier*/
    bar_name.flag = 1;        /*release waiters*/
  }
  else
    while(bar_name.flag == 0){};/*busy wait for release*/
}
```

So, in the case of software barriers, we can go for centralized or decentralized barriers but we are going to discuss on the centralized barrier mechanism here. Though we are actually going for software barriers, but internally we can take the help of the hardware support in terms of hardware primitives to acquire the locks. So, this barrier can be implemented using locks, shared counters and the flags.

So, in order to implement this we will consider a simple piece of code, where we have structure variable for a barrier, which consists of an integer counter, and also it consists of struct variable lock, and also it has a flag bit. So, having defined this particular barrier struct variable, now we will see how we can go ahead with the software barrier mechanisms. In this particular barrier code, what we are doing is first we acquire a lock on that particular barrier variable. So we acquire a lock and this locking mechanism, we already discussed previously in the mutual exclusion mechanism. So, we acquire the lock so that there is no other processor, which is going to enter into this critical section.

Let us say, this is a critical section here we are going to check whether the counter of this barrier variable is 0 or not. And if it is so then we are going to reset the flag associated with this particular barrier, because we have to perform this in an atomic sense or a mutual exclusion sense. So, we are actually acquiring lock on this particular barrier variable, so that we prevent all other processors to enter into the critical section to check and update these values by using this lock.

So, we acquired the lock and we check the condition and if the condition is true. That means this particular processor is the first processor to come to the barrier, then immediately it is going to reset the flag bit. And also it reads, it increments this counter, and it reads the value into its local variable mycount. Now, mycount will be one and after this it is going to release the lock, so that second processor again tries to enter the the critical section by performing these operations.

So, second processor will come and it acquires the lock, and it checks whether the counter is equal to 0 or not. Now because already processor p1 incremented the counter so this condition will be false, so the second processor is not going to rest the flag bit. But the second processor is going to increment this counter, and now the counter value is 2 and it is going to get the counter value 2 into its location. And it also releases the lock, so that the third processor can go to the critical section and increment the counter and so on.

Now after releasing the lock now each of these processors is going to execute the remaining piece of code. So, they just check whether the mycount is equal to p or not where p is the total number of processors involved in this particular barrier. So, if the count is equal to p that indicates that this particular processor is the last processor to reach the barrier. And so as a result it has to reset the counter and also it has to set the flag bit.

If any of these processors is not the last processors to enter the critical, not the last processor to reach the barrier, then this condition will be false and then they will go to else, and in the else part there is an infinite loop. They are just waiting for this flag bit to be 1, if the flag bit is 0, then there will be in the infinite loop. Whenever the flag bit is 1 then they will come out of this busy wait loop, and so that they can proceed with all the instructions which are subsequent to this barrier instruction, in the code.

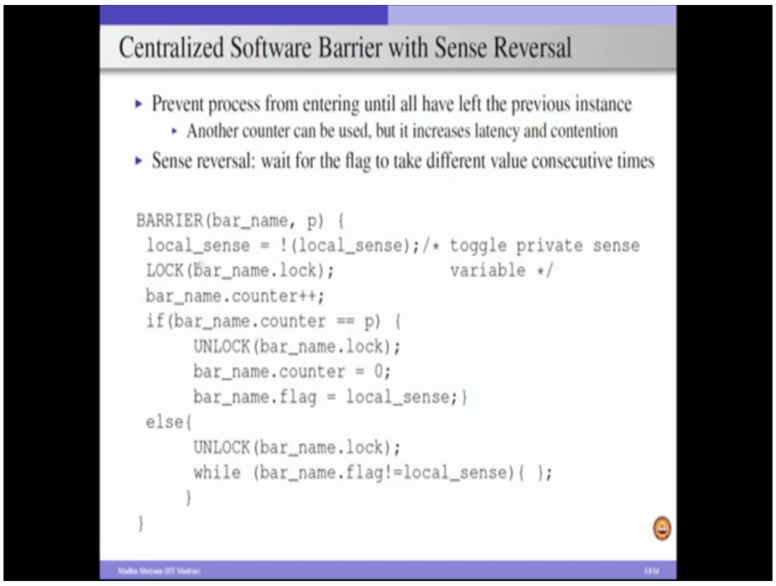
So, by looking at this code we feel like this is going to achieve the global synchronization, but actually there is problem with this particular piece of code. So, the problem is, let us say, we have four processors in our system and all this four processors are involved with this barrier. Let us say processor p1 comes to the barrier first, so the counter value is 0, so it is going to reset the flag bit and its count is equal to 1 and it releases the lock. And this condition is false, so it will be in this while loop because the flag is equal to 0. So, it is in the while loop.

Next processor two comes and it also acquires lock and counter is not equal to 0 now so it is not going to execute this, but it increments the counter value and now its count equal to 2. And now it releases the lock and this condition will be false, and it will be again in the while loop, waiting for this flag bit to be 1. Third processor also comes here, acquires the lock and then increments the counter and its count value is 3 now releases the lock and this condition again is false. So it will be in the while loop waiting for flag bit to be 1.

Now, fourth processor came and it also acquires the lock and increments the counter and now the count is equal to 4, it releases the lock and now the count is equal to 4. So, as a result this condition is true and now it is going to reset the counter value, and it sets the flag bit. Now, the flag bit is set so as a result all the processors, which are actually busy waiting for this flag bit to be 1 will see that this while condition is false, so as a result they can now go ahead with the instruction subsequent to this barrier instruction.

The main problem is here we are actually using the same flag bit, where it is all the processors are actually looking at this flag bit to be set to 1, and if any of the processors is not seen the moment when the flag bit is set to 1, then we are going to have a problem. In order to eliminate this problem we have to go for another mechanism that is called as the sense reversal mechanism.

(Refer Slide Time: 49:47)



Centralized Software Barrier with Sense Reversal

- Prevent process from entering until all have left the previous instance
  - Another counter can be used, but it increases latency and contention
- Sense reversal: wait for the flag to take different value consecutive times

```
BARRIER(bar_name, p) {
 local_sense = !(local_sense);/* toggle private sense
 LOCK(bar_name.lock);           variable */
 bar_name.counter++;
 if(bar_name.counter == p) {
      UNLOCK(bar_name.lock);
      bar_name.counter = 0;
      bar_name.flag = local_sense;}
 else{
      UNLOCK(bar_name.lock);
      while (bar_name.flag!=local_sense){ };
     }
}
```

So, because with the previous 1, now one processor is waiting at the previous barrier and all the other three processors are waiting at the second instance of the same barrier. So that none of the four processors will proceed further, and as a result we will have a deadlock scenario. So, in order to eliminate this problem we have to go for a modified version of this mechanism by using the sense reversal concept.

So, here what we actually do is we wait for the flag to take a different value on the consecutive times because the previous problem happened mainly because the same set of processors are actually going to take the same barrier second time. When these processors are going to go for the same barrier next time, if we change the flag value or in other words if you change the condition on which they are actually waiting, in the second instance then we can eliminate the problem. So that we have done here see here for example, we are actually changing this while condition. Previously, we considered while flag is equal to 0 there is an infinite loop. But here we are considering while barrier name flag is not equal to the local sense then there is an infinite loop.

Now, the code is the first processor comes to the barrier and it executes this piece of code first it changes its local sense. Previously whatever the locals sense value, let us say if it is a 0. Now it is going to consider local sense is equal to 1 by using this negation operation. So it is now going to get value which is exactly opposite to whatever the value we considered in the earlier invocation of this barrier.

For example, if the same processor previously entered the barrier with the local sense value equal to 0. Now it is going to consider local sense value equal to 1, now it acquires the lock, on the barrier variable, and now it increments the counter value it is similar to the previous code whatever we have discussed in the previous foil. And whether this processor is the last processor to reach the barrier or not, and if it so then it is going to release the lock.

After releasing the lock it is going to execute the remaining piece of code, that is like it is going to reset the counter and also it is going to set the flag as the value whatever it has in its local sense variable. If this processor is not the last processor to reach the barrier, then it will go for the else part and here it releases the lock and again because this not the last processor to reach the barrier, so it will just wait infinitely here for this condition to be false. So, here the condition is the barrier name flag is not equal to the local sense.

Since, this particular processor updated the barrier name flag is equal to the local sense. So, as a result now if this value is different, then it will be in the infinite loop. If it is same then it will exit from that and then it can proceed further to the instructions, which are following the barrier. Now here consider again the previous problem scenario, where four processors are there, three processors reach the barrier and when the fourth processor comes to the barrier. Now, it gets this if condition true, so it releases the lock and then this last processor is going to set the barrier flag, as its local sense value.

If any of the three processors which are actually waiting by executing this while loop is actually context switch then there will not be any problem, because when it comes back because it is actually waiting for this particular condition and each has a different local sense value and they are actually waiting on that corresponding local sense. So that is not going to create a problem, whatever we discussed earlier. So, this is about synchronization mechanism that is implemented in multicore systems or multiprocessor systems. So, with that I am concluding this module and in the next module, I am going to discuss the memory consistency issues.

Thank you.