

L04a: Shared Memory Machines

1. Shared Memory Machine Model

In this day and age, parallelism has become fundamental to computer systems. Any general-purpose CPU chip has multiple cores in it. Every general-purpose operating system is designed to take advantage of such hardware parallelism.

In this unit, we will study the basic algorithms for synchronization, communication, and scheduling in a shared-memory multiprocessor, leading up to the structure of the operating system for parallel machines.

Lesson Outline

Machine Model

Synchronisation

Communication

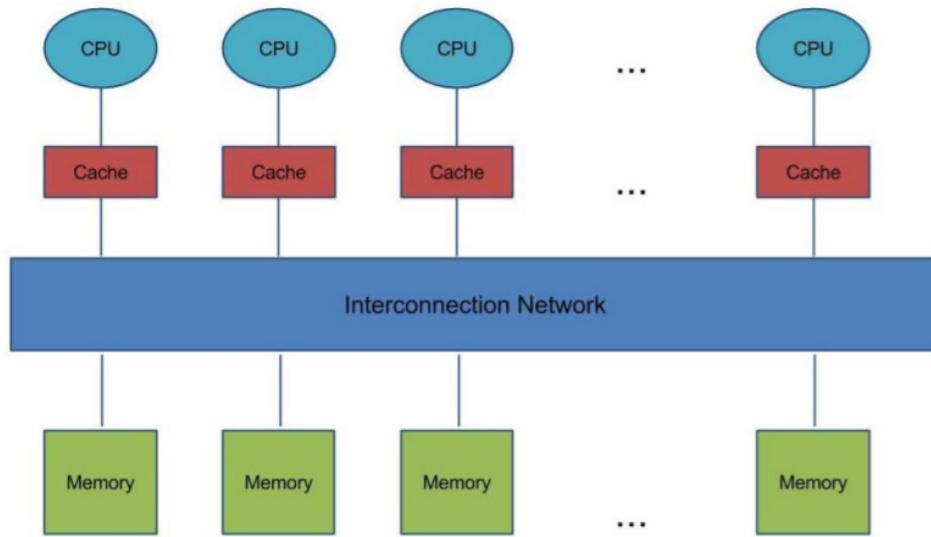
Scheduling

Parallel OS case studies

We'll start today's lecture with a discussion of the model of a parallel machine. A shared memory multi-processor, or a shared memory machine. we can think of three different structures for this shared memory machine.

Shared Memory Machine Model

Dance Hall Architecture



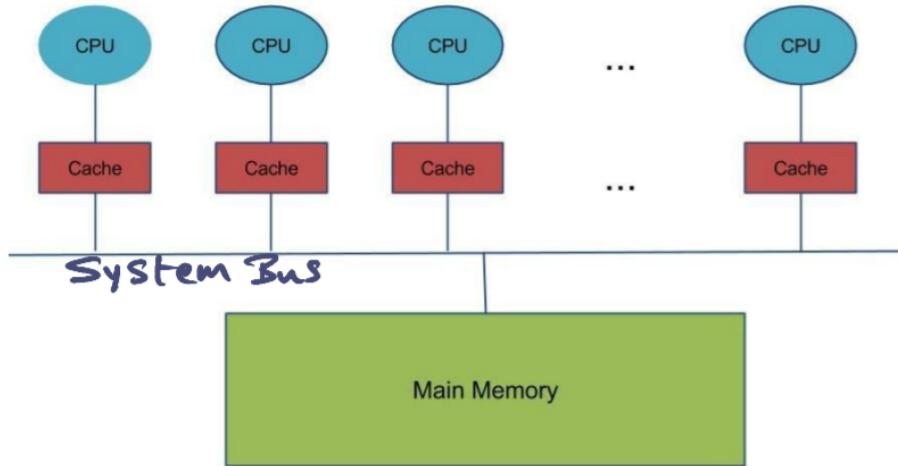
The first structure is what we call a dance hall architecture and a dance hall architecture is one in which you have CPUs on one side and the memory on the other side of an interconnection network.

let me say something that is **common to all three structures** that I'm going to describe to you. The common things are that in every one of these structures, **there's going to be CPUs, memory, and an interconnection network**. And **the key thing is it's a shared memory machine, that the entire address space defined by the memories is accessible from any of the CPUs**. So that's one common thing that you see in all the three styles that I'm going to talk to you about.

And in addition to that, you'll see that there is a cache that is associated with each of these CPUs. So there's a Dance Hall Architecture.

Shared Memory Machine Model

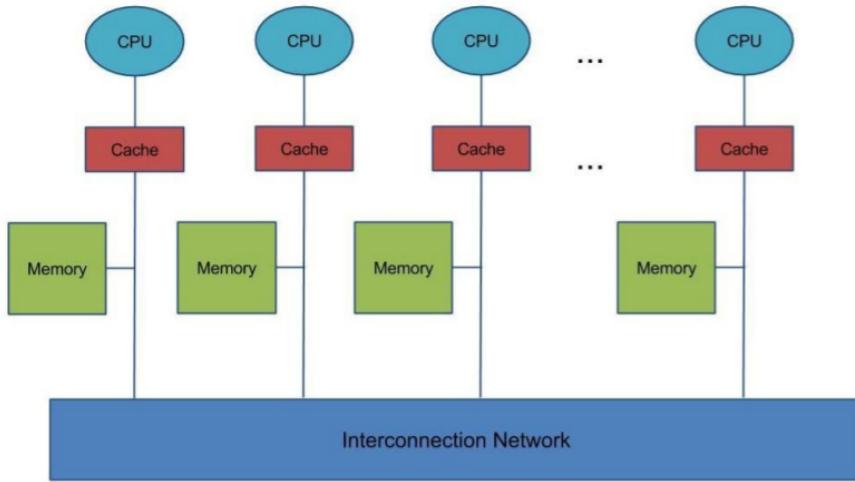
SMP (Symmetric MultiProcessor)



And the next style is what is called an SMP architecture, or a Symmetric multiprocessor. Here what you see is the interconnection network that I showed you from the dance hall architecture. I simplified it considerably, showing a simple bus. A system bus that connects all the CPU's to talk to the main memory. And it is symmetric because the access time from any of the CPUs to memory is the same. And that's the idea of the system bus that allows all of these CPUs to talk to the main memory. The other thing that you'll notice that I already mentioned is that every CPU comes equipped with a cache and we'll talk more about the shared memory machine, the caches in a minute.

Shared Memory Machine Model

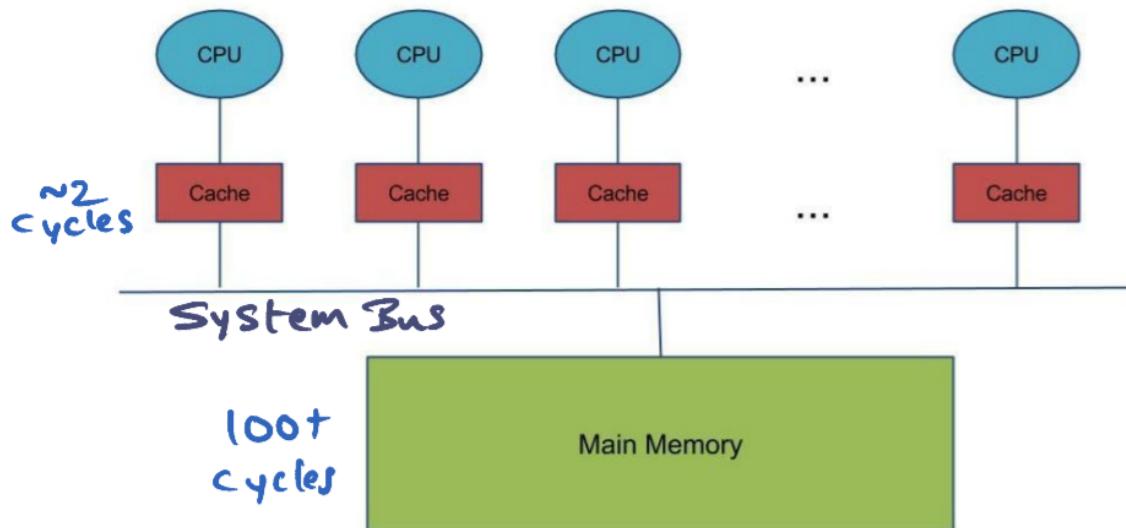
DSM (Distributed Shared Memory)



So the third style of architecture is what is called distributor shared memory architecture. So in this distributor shared memory architecture what you have, or DSM for short is that. You have memory and a piece of memory that is associated with each CPU. At the same time, each CPU is able to access all of the memories through the interconnection network. It is just that the access to memory that is close to this guy is going to be obviously faster than trying to access memory that is farther from here that has to be accessed from the interconnection network.

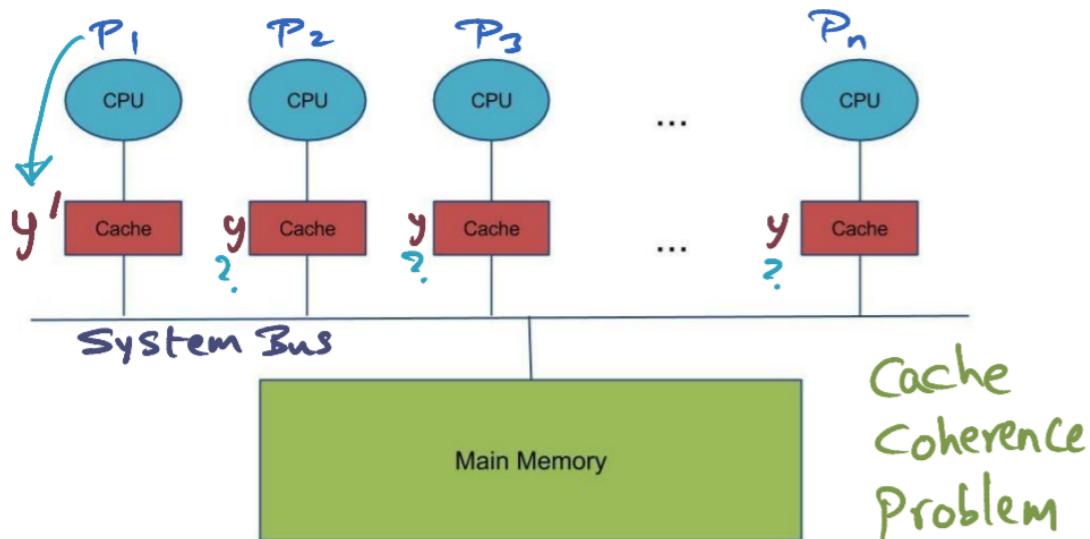
2. Shared Memory and Caches

Shared Memory and Caches



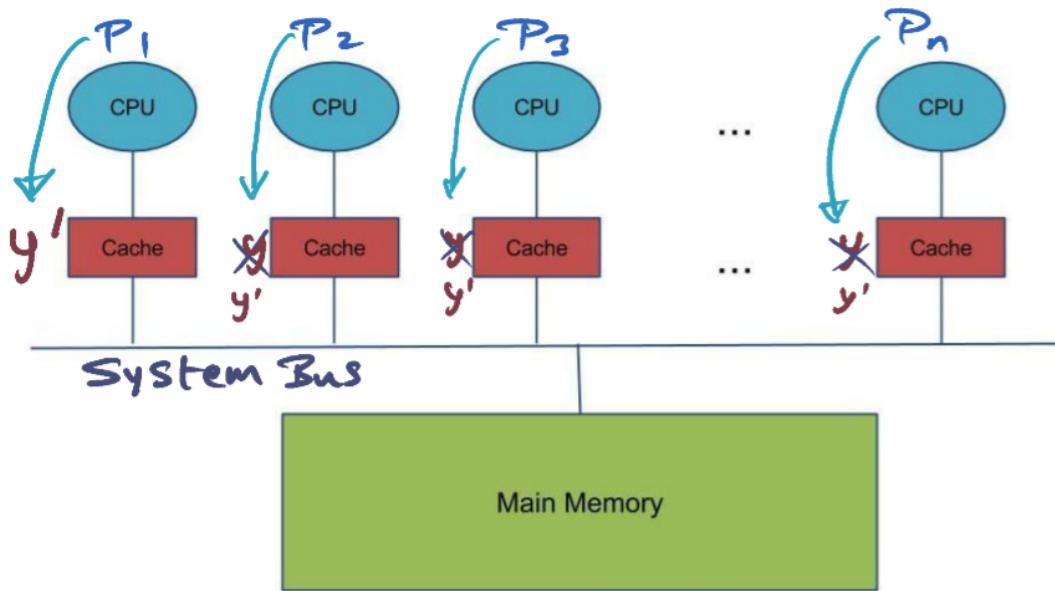
Now let's start discussing shared memory and private caches. And in order to simplify the discussion what I've done is, I'm using the simplest form of the shared memory machine that I told you about. That is an SMP where there's a single system bus that connects all these processes to talk to the main memory. Now cache that is associated with the CPU serves exactly the same purpose in a multiprocessor like this, as it does in a uniprocessor. And that is, the CPU, when it wants to access some memory, memory location, of course, it is going to go to the cache and see if it is there. If it is there, life is good, it can get it from the cache. If it is not in the cache, then it has to go to the main memory. And fetch it from the main memory, and put it into the cache so it can reuse it later, and that's the purpose that cache performs in a uniprocessor. **A multiprocessor performs exactly the same function as a uniprocessor. By caching data that is pulled in or instructions that are pulled in from memory into the cache so that the CPU can re-use it later.** When cache in a multiprocessor, associated with each of these CPUs performs exactly the same role. As it does in the uniprocessor. And that is, if the CPU goes to the main memory, and pulls in some data, it's going to come and sit in the cache. So obviously when the CPU is looking for something, first it is going to come and look at the cache. If it is not there, it's going to go to the main memory. And fetch the data and put it into the cache so that in the future the CPU doesn't have to go to the main memory, but get it from the cache itself. That's the purpose of the cache in a uni-processor. Exactly the same purpose a cache performs in a multiprocessor as well. However, there's a **unique problem** with a multiprocessor. **The fact that there are private caches associated with each one of these CPUs, and the memory itself is shared across all of these processors.** Let me explain that.

Shared Memory and Caches



Let's say that there's a memory location y that is currently in the private caches of all the processors. Well, maybe y is a hot memory location so all the processes happen to fetch it and therefore it is sitting in the private caches of all the processes. Let's say that process P_1 decides to write to this memory location y now y is changed to y' . Now, what should happen to the value of y that is sitting in all the P caches? And clearly, you know, in a multiprocessor, a multi-threaded program, there could be a shared data structure that is being shared with all the processors. And therefore if this guy writes to a particular memory location it is possible that that same memory location is in the private caches of the peers. So this is referred to as the **cache coherence problem**.

Shared Memory and Caches



Now someone has to ensure that, if at a little point of time if the process of p two or p three or any of these processes that happen to have this memory location y, in the private caches decide to access it. They should get y prime and not y. Now, who should ensure this consistency? Here again, there's a partnership between hardware and software. In other words, **the hardware and software have to agree as to what is called the memory consistency model**. That is, this memory consistency model, is a contract between hardware and software as to what behavior a programmer can expect, writing a multi-threaded application running on this multiprocessor. An analogy that you may be familiar with is a contract between hardware and software. If you just think about a uniprocessor, if you think about a uniprocessor. There's a compiler writer that knows about the instruction set provided by the CPU. And the architect goes and builds a CPU, and he has no idea how this instruction set is going to be used, but there is an expectation that the instruction set, the semantics of that instruction set. Is going to be adhered to in the implementation of the processor. So that, the compiler writer can use that instruction set in order to compile high-level language programs. Similarly, when you're writing a multithreaded application, you need a contract between the software and the hardware, as to the behavior of the hardware when. Processors are accessing the same memory location. And that is what is called the memory consistency model. And what we're going to do now, is in order to get your creative juices flowing, I'm going to ask you a question.

3. Processes

Question

Assume $a = b = 0$ initially.

Process P1

$a = a + 1;$

$b = b + 1;$

Process P2

$d = b;$

$c = a;$

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1$

Now, let's talk through what possible values d and c can have. You may have picked several of these choices, but it is okay, you know, whatever you picked, it's okay. Let's talk through these different choices, to see what is possible given this set of instructions and the fact that processing speed one and speed two are executing, independently on two different processors and, we have no way of knowing, what is going on with the shared memory.

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$

$b = b + 1;$

Process P2

$\left. \begin{array}{l} d = b; \\ c = a; \end{array} \right\}$

What possible values for d and c ?

$c = d = 0$

$c = d = 1$

$c = 1, d = 0$

$c = 0, d = 1$

Now the first possibility, is that these two instructions, assignment of, a B to D and C to A, they happen. In time order, before any of these instructions in P1 are executed. That's possible. because if these shared memory accesses happen before these guys, they're responsible that both of these instructions are executed before any of these instructions executed. In that case, what you would get into D and C are the old values of a and b, namely zero.

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$

$b = b + 1;$

Process P2

$\{ d = b;$

$c = a;$

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1$

The second possibility is that both these instructions that are executed on P2 are executed after both the instructions on P1 have completed execution. So in this case, both a has gotten a new value, b has gotten a new value, and so when we go here and make the assignments. Then both d and c are going to have new values that are in b and a respectively. And so, this is a possibility, right? There is a possibility that both c and d have a value of one in them.

Question

Assume $a = b = 0$ initially.

Process P1

$a = a + 1;$
 $b = b + 1;$

Process P2

$\{ d = b;$
 $c = a;$

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1$

Let's see if the third possibility can happen. The third possibility for it to happen, it is conceivable that we insert these two instructions in the middle of this. Or in other words Process P1 executed this instruction and in time order it so happens that these two instructions got executed, and then this, this instruction got executed. And therefore, once you get into d is the old value of b , that is zero. And once you get into c is the new value of a . Because this instruction is executed. And therefore, you get one into c . And that's why this possibility is also, is also perfectly valid.

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$
 $b = b + 1;$

Process P2

$d = b;$
 $c = a;$

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1$? Can this happen?

?

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$
 $b = b + 1;$

Process P2

~~$d = b;$~~
 ~~$c = a;$~~

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1 \Rightarrow$ msgs go out of order

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$
 $b = b + 1;$

Process P2

$d = b;$
 $c = a;$

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1$? Do you want it? to happen

?

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$
 $b = b + 1;$

Process P2

$d = b;$
 $c = a;$

What possible values for d and c ?

- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1$? Not intuitive

Question

Assume $a=b=0$ initially.

Process P1

$a = a + 1;$
 $b = b + 1;$

Process P2

$d = b;$
 $c = a;$

What possible values for d and c ?

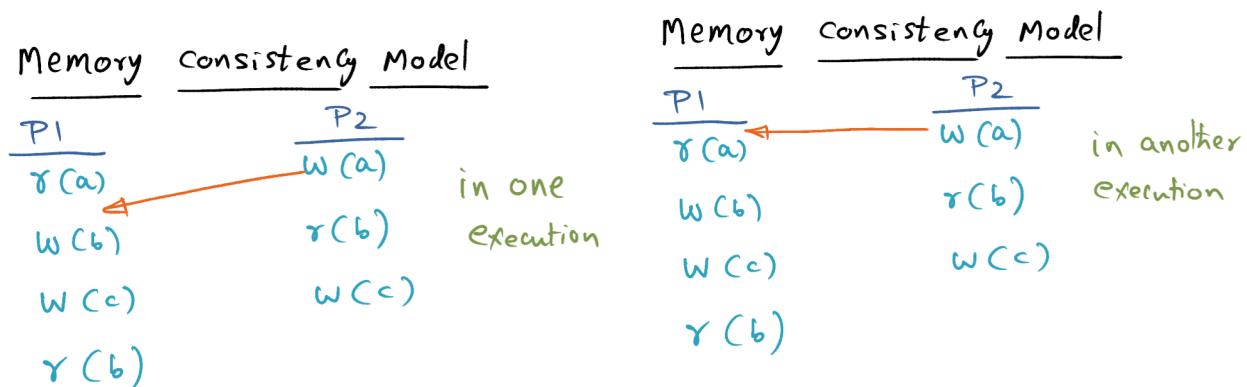
- $c = d = 0$
- $c = d = 1$
- $c = 1, d = 0$
- $c = 0, d = 1 \Rightarrow$ model should disallow

Now, let's look at this last choice that I have. C gets zero and D gets one. Can this happen? And the reason I ask you this question is that if you look at this piece of code and this piece of code here. If D were to get one, what that means is that this assignment of B gets B plus one has already happened on P1. That's how the new value of B has gotten into D. But yet, we're saying when this process completes, C has a value of zero. What does that mean? It means that the new value of A hasn't come into the processor P2. How can this happen? It can happen if messages go out of order. You have to remember that, if you recall the picture of the shared memory machine, you've got an interconnection network that is connecting all these processors. And a write that happens on this processor has to go through the interconnection network and get to this other processor. Now it is conceivable that if a message goes out of order. It is possible that when this process executes this statement. This new value of B has arrived, the message that contains a new value you B has arrived and therefore this assignment gets a new

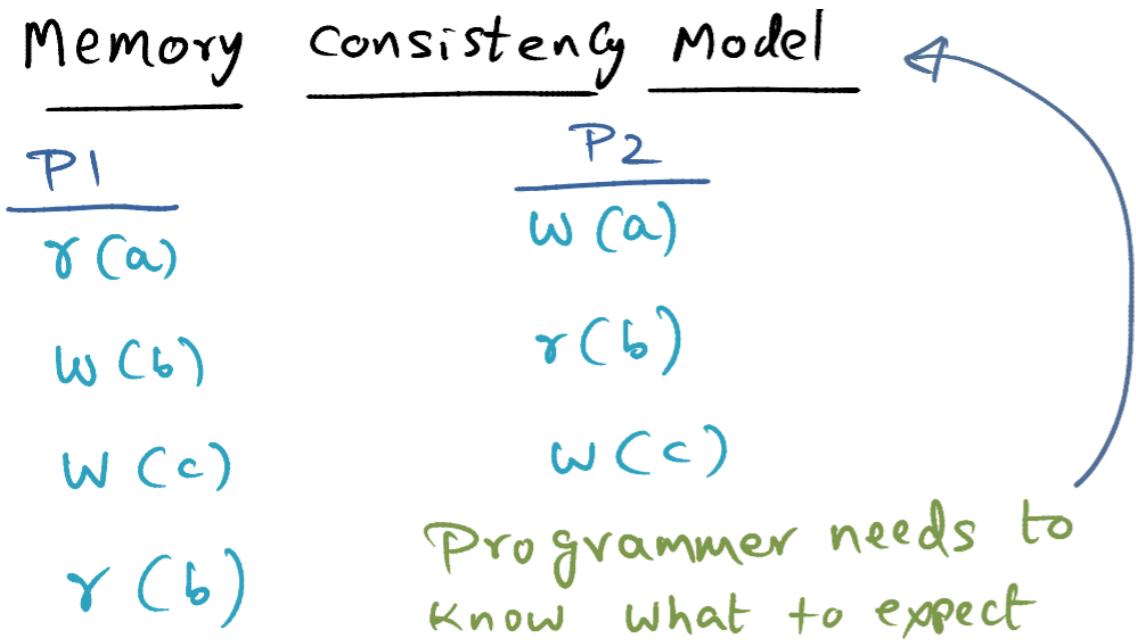
Now, let's look at this last choice that I have. C gets zero and D gets one. Can this happen? And the reason I ask you this question is that if you look at this piece of code and this piece of code here. If D were to get one, what that means is that this assignment of B gets B plus one has already happened on P1. That's how the new value of B has gotten into D. But yet, we're saying when this process completes, C has a value of zero. What does that mean? It means that

value of b, but when the process executes this statement. The new value of a hasn't arrived yet and it can happen if the messages go out of order in that case, you can end up with this particular choice of c having a value of zero and d having a value of one when this process completes execution. Do you want it to happen? Now intuitively, you would see that this is not something you expect to happen. As a programmer, you don't want surprises, right? And if you don't want surprises, perhaps if it is a non-intuitive result, that's something that should not be allowed by the model. So, when we talk about the memory consistency model, we're saying what is the contract between the programmer and the system? And what we are seeing through this example is that **this particular outcome is counter-intuitive and therefore the model should not allow this particular outcome to be possible in the execution**. And this is the reason why you have a memory consistency model.

4. Memory Consistency Model

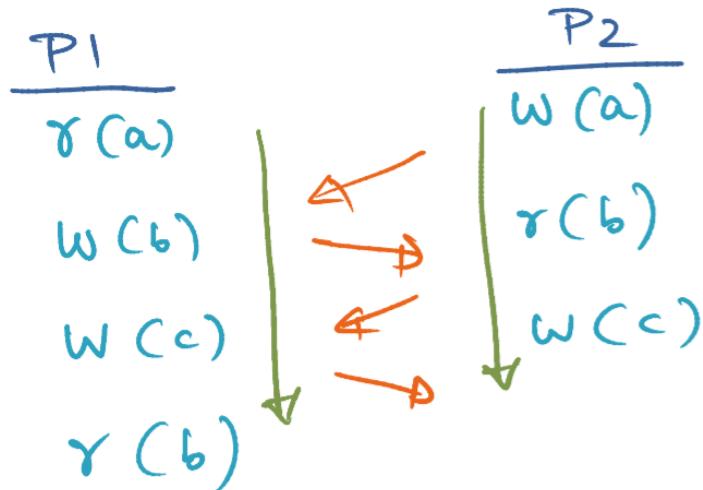


So here I'm showing you a set of accesses, memory accesses down on processor P1 read access, and write access and so on, and these are the memory location being touched by these accesses on P1. And on P2 I'm showing some real set of accesses to shared memory locations, and we know that **Processor P1 accessing memory and processor P2 accessing memory are completely independent of one another**, therefore it is possible that in one execution of P1 and P2 this particular access of writing it to memory location A Happens after reading a memory location, a happens on P1 in one execution. And if you run the same program again, P2 and P1 constituting the program run it again. It's possible that another execution of the same program the write of a happens before the read of a. It's perfectly feasible for this to happen because there is **no guarantee of the ordering** of these accesses going to the main memory.



And if you think about it, both of these executions, whether it is earlier execution where write happened after this read, or this execution in which the write is happening before the read. Both these executions are reasonable and correct and something that the programmer can live with. It's acceptable to the programmer. Now in other words, **what the programmer needs to know is what to expect from the system in terms of the behavior of shared memory reads and writes that can be emanating from several different processors.** And this is what is called **the memory consistency model.** So the expectation of the programmer is what is engrained in this memory consistency model. As a programmer, you don't want any surprises. And **there's a purpose of the memory consistency model to sort of satisfy the expectation of the programmer.**

Sequential consistency (SC)



Program
order
+
arbitrary
interleaving

So I'm going to talk to you about one particular memory consistency model, which is called a **sequential consistency memory model**. And you consider the access from P1 and P2. Well. **One expectation that you have of the programmer is that the accesses that you have on a particular processor, is going to be exactly in the order in which your written** or in other words, if you look at these sequences of accesses, you have the right of b here and the need of b here. You know that your one expects to see when you do this V, **whatever you wrote here is what you expect to see. That's what's called a program order**. What you expect is the program order to be maintained, namely the order in which you've generated memory access should be maintained by the execution on that processor. That's the program order. And **in addition to that, there is this interleaving of memory accesses between P1 and P2**. And this is where we said, we have no way of controlling, the order in which these accesses are going to be satisfied by the memory. Because it depends on the execution of P1 on processor P1. And the execution on P2 and how that each memory and so on. And so this interleaving can be arbitrary. That is, interleaving between accesses that you see here and the accesses that you see here can be arbitrary. So, that's the sequential consistency memory model, which has two parts to it. One is the program order. That is the order that you see, textually, in every individual process. I'm showing you two here, but you can have any of these processes. But in each one of these processes, the textual order in which memory accesses are generated, they're going to be satisfied. That's the program order. On the other hand, the interleaving of this memory access has occurred all of the processes are going to be an obituary. So those are the two properties of the sequential memory consistency model. In order for an analogy that will drive home the point about the sequential consistency and what you might see in a casino and if you watch a casino card shark shuffle cards. He might take a card deck and split it into two

halves, and then he'll do a merge shuffle of two splits, and, and, and create a complete deck. Exactly what's going on with sequential consistency. You have splits of memory access on several different processors, and they're getting interlinked in some fashion. Just like card shuffler is interweaving the cards from two decks and creating one card deck. All of it. By the way, this particular memory consistency model's sequential consistency was proposed by Leslie Lamport and, this is a popular guy. You're going to see him again later on when we talk about distributor systems. But he came up with this idea of a sequential consisting memory model back in 1977. And since then there have been a lot of different consistency models that have been proposed. And in future lessons on distributed systems, we will see other forms of memory consistency models such as release consistency, lazy release consistency, and eventual consistency. But hold on. We will come back to that later on.

5. Memory Consistency and Cache Coherence

SC + our earlier question

Assume $a=b=0$ initially.

Process P1

$$\begin{aligned} a &= a+1; \\ b &= b+1; \end{aligned}$$

Process P2

$$\begin{aligned} d &= b; \\ c &= a; \end{aligned}$$

What possible values for d and c ?

$c = d = 0$

$c = d = 1$

$c = 1, d = 0$

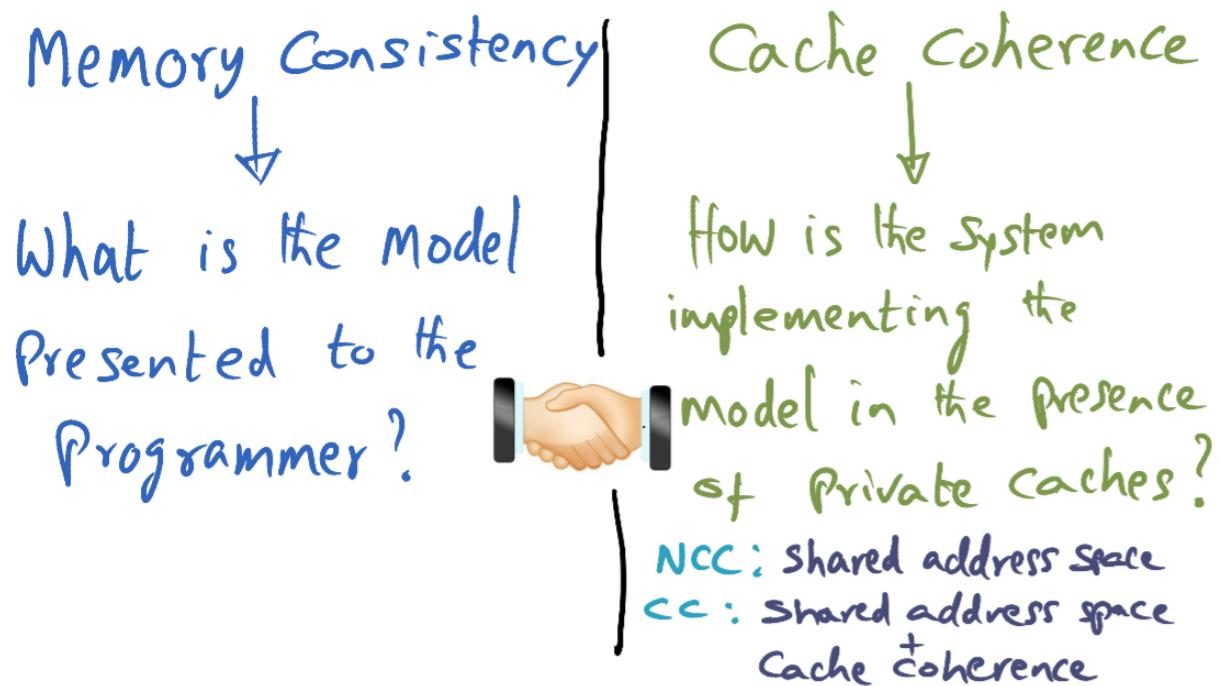
~~$c = 0, d = 1$~~

Not possible with SC

So now having seen the sequential memory consistency model, what we can do is go back to our original example, and ask the question, what are all the possible outcomes for this particular set of memory accesses performed on p1 and p2? Now what possible values can d and c get? Well obviously, you can get the first choice, no problem with that. Can get the second choice, it can get the third choice, and as we illustrated earlier, all of these are just interleaving of these memory accesses on P1 and P2. But the fourth one is not possible with sequential consistency, because there's no interleaving of these memory access and these memory access that'll result in this particular outcome. That's comforting, that's exactly what we thought would be a useful

thing to have in a memory-consistency model that gives only intuitive results and, and makes sure that non-intuitive results don't happen.

Memory consistency model is what the application programmer needs to be aware of to develop his code and know that it will execute correctly on the shared memory machine. As operating system designers, however, we need to help make sure that this code runs quickly. To do that, we need to understand how to implement the model efficiently. And also the relationship between hardware and software that makes it possible to achieve this goal.



So now, we understand the memory consistency model. What is the model that is presented to the programmer? That's what the memory consistency model is. On the other hand, **cache coherence is how is the system implements the model in the presence of private caches**. So this is a handshake, a partnership between hardware and software, between the application programmer and the system, in order to make sure that the consistency model is actually implemented correctly by the cache coherence mechanism that is ingrained in the system. And **the system implementation of cache coherence is a hardware-software trade-off**.

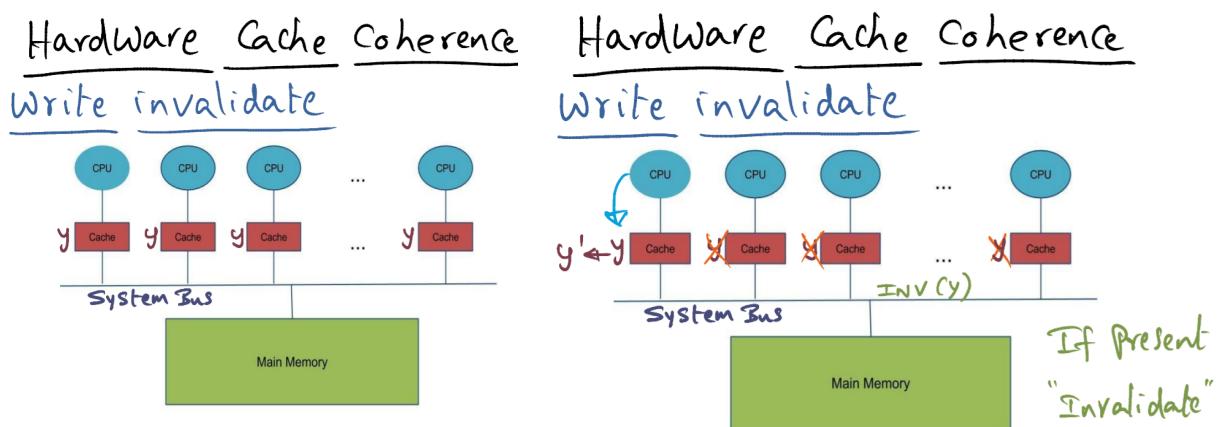
NCC: Now for instance one possibility, is that the hardware is only giving shared address space. It's not giving you any way of making sure that the caches are coherent, but it is giving you the **shared address space**. And it is letting the software, the system software ensure that this contract is somewhat satisfied. **And the working of the cache coherence is maintained in software by the system**. That's one possibility, and **that is what is called a non-cache coherent shared address multiprocessor**, meaning that there is shared address space, that's available for all the processors, there are private caches for holding data that you bring from

memory. If you modify data, it is a problem for the system software to make sure that the caches remain coherent. So it's non-cache coherent. That is called **NCC shared memory multi-processor**.

CC: The other possibility of course is that the **hardware does everything**. It provides the shared address space, but it also maintains cache coherence in hardware. And that's what is called a **cache-coherent multi-processor**, or a **CC multi-processor**.

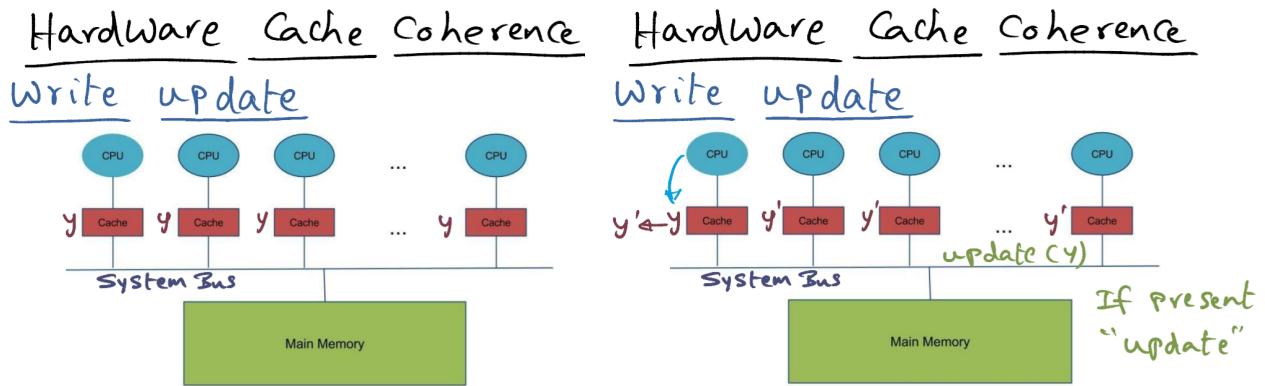
6. Hardware Cache Coherence

Now let's focus on the hardware implementing cache coherence entirely in addition to giving the shared address space. There are **two possibilities** if the hardware is going to maintain the cache coherence.



One possibility is what is called the **write invalidate scheme**. And here the idea is, if a particular memory location is contained in all the caches, all these processes have fetched this particular memory location Y, and it's been sitting in the private caches of all these processes. And if now, the process of P1, decides to write to this particular memory location it changes from y to y prime. When that happens, what we're going to do is, **the hardware is going to ensure that all of these caches are invalidated**. So, the way it's done is that **the hardware, as soon as this change happens, is going to broadcast a signal on the bus** called invalidate memory location Y. So that's something that propagates on the system bus, and all these processes of caches, are observing the caches, and this is sometimes referred to as **snoopy caches**, in a lighter vein, these caches are snooping on the bus to see if there's any change to memory locations that are cached locally. And in this case, if an invalidation signal goes out for a particular memory location y, then each of these caches is going to say "do I have that particular memory location? If I do, let me invalidate it". So, that particular memory location gets

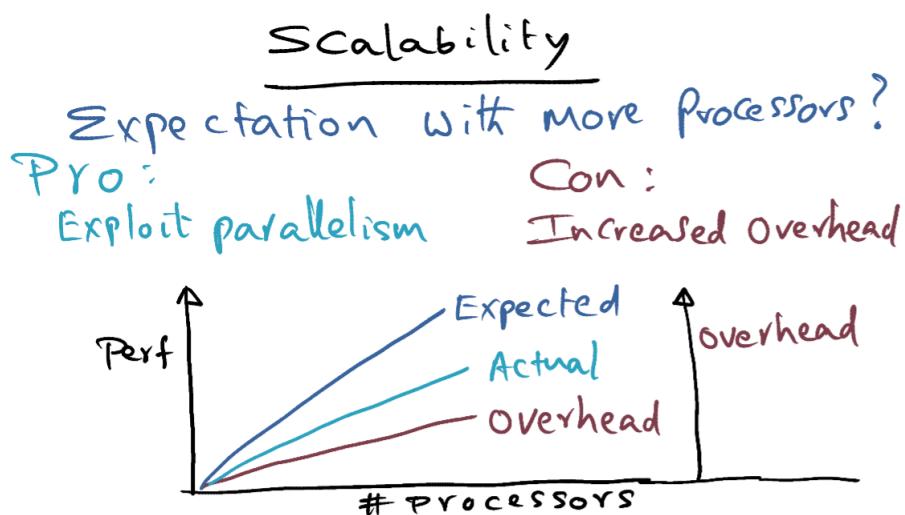
invalidated. So the idea is if you have that particular memory location, invalidate it. If you don't have that memory location, ignore it. Right? So if you don't have it, you don't have to bother, but if you particularly happen to have this memory location cached in your private cache, and if you observe an invalidation for that particular memory location, you go ahead and invalidate it. That's what is called the **write invalidate cache coherence scheme**. You may already be one step ahead of me, and you may be thinking what would be an alternative to doing this invalidation? And, and you may be right. You thought of perhaps updating the caches.



That's what is called the **write update scheme**. The idea here is if this guy is going to write to this particular memory location, modify to y' , what we do is, instead of invalidating it on the bus, if there is a capability in hardware to send an update for this particular memory location on the bus. You send it out saying that I modified this particular memory location, this is a new value, and if these caches happen to have the same memory location, they all modify it from y to y' . And now, all of these caches have the new value of y' and the old values disappear from the system. So in this case, what we are doing is, if you have it, update it. Once again, you're snooping on the bus. Each of these process of caches is snooping on the bus and if they see an update for a particular memory location, they're saying, "well, let me modify it, so that future accesses by my CPU will get the most recent value that had been written into this particular cache line". That's the idea behind the write update scheme. Now whether we're talking about the write update scheme or the earlier write invalidate scheme, one thing should become very clear in your mind and that is there is work to be done whenever you change some memory location that could conceivably be cached in the other private caches of the CPUs. And the invalidate scheme has sent out an invalidate message. If it's an update scheme, it sends out an update message. **And that kind of transaction that's going on is, is an overhead.** And as, as a system designer, one of the things that we've been emphasizing all along is that we want to keep the overhead to a minimum. **But you can also see immediately that the overhead is going to be something that grows as you increase the number of processors.** As you change this inter-connection network from a simple bus to a more exotic network.

And also depending on the amount of sharing that is happening for a particular shared memory location.

7. Scalability



Now as a programmer, you have a certain expectation as you add more processors to the system. Your expectation is natural if you think that if you add more processors your performance should go up. So this is expected. This is what is called **scalability that the performance of a parallel machine is going to scale up as you increase the number of processes**. Reasonable to expect that. However, I mentioned just now that the overhead is associated with increasing the number of processes in terms of maintaining cache coherence when you have sharing that is happening for shared data. And so, therefore, the pro in adding more processors is the fact that you can exploit parallelism. That's the reason why you're able to get this expectation of increased performance with processors. **But unfortunately, as you increase the number of processors, there is increased overhead.** The increased overhead also grows. As you increase the number of processors more, overhead is going to be incurred by the system. If we have an eight-processor SMP the overhead for cash coherence is less than if we have a 16 processor SMP or a 32 processor or a 64 processor, so the overhead is going to grow. As a result, you can see that you have the proof exploiting parallelism but you have the con of increased overhead and **you end up with an actual performance that's somewhere in the middle between your expectation and the overhead.** So, in some sense, this is a difference between what your expectation is and what the overhead you're paying. And that becomes the actual delivered performance of a parallel machine. **And this is very important to remember, that your delivered performance may not necessarily be linear in the number of processors that you add to the system.**

So what should we do to get good performance? Don't share memory across threads as much as possible. If you want good performance from the parallel machine. **A quote that is attributed to a famous computer scientist Chuck Thacker comes to mind, shared memory machines scale well when you don't share memory.** Of course, as operating system designers, we have no control over what the application programmer does. **All we can do is to ensure that the users' shared data structures are kept to a minimum and the implementation of the oppressing system caught itself.** You will see how relevant Chuck Thackers quote is as we visit operating system synchronization, communication, and scheduling algorithms and more generally. The structure of the operating system itself in this lesson. See if you can remind yourself of this quote, and how often it permeates our discussion as we go through this lesson.

L04b: Synchronization

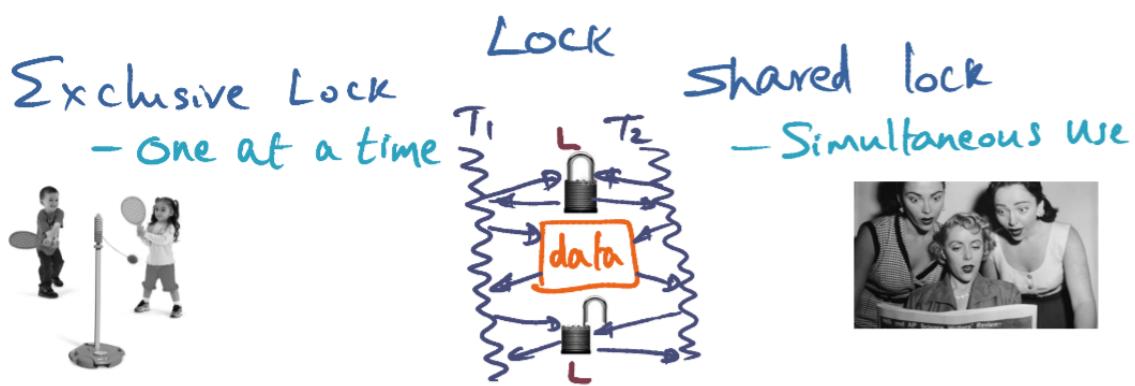
1. Lesson Summary

Lesson Outline

- ✓ Machine model
- Synchronization
- Communication
- Scheduling
- Parallel OS case studies

In the previous lecture, we got done with discussing the model of a parallel machine. And in this lesson, what we're going to start doing is talking about synchronization algorithms that goes into the guts of any parallel operating systems that is supporting multi-threaded applications. And as we discuss the synchronization algorithms, watch out for Thacker's quote that I mentioned in the previous lesson on sharing, in shared-memory multiprocessors that are going to be key in terms of understanding the scalability of the synchronization algorithms.

Synchronization primitives for shared memory programming



Synchronization primitives are a key for parallel programming. In your first project, you implemented a threads library, which provides the mutual exclusion lock. Let's talk about locks. What exactly is a lock? Well, you know, in the metaphor that you know about in real life. The lock is something that protects something that is precious. And in the context of parallel programming, if you have multiple threads executing and they share some data structure, it is important that the threads don't mess up each other's work. And a **lock is something that allows a thread to make sure that when it is accessing some particular piece of shared data It is not being interfered with by some other thread**. That's the purpose of a lock. So the idea would be that, a thread would acquire a lock, and once it acquires a lock, it knows that it can access this data that it shares with potentially other threads. I'm showing only two threads here, but potentially in a multi-threaded program, you can have a lot more threads that are sharing a data structure. And once T1 knows that it has access to this data structure, then it can do whatever it wants with it. And then once it is done with whatever it wants to do with this data it can release the lock. So that's sort of the idea behind a lock.

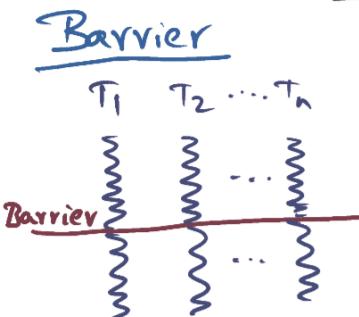
And locks come in two flavors, one is what we'll call an **exclusive lock or a mutual exclusion lock**. And this is exactly the one that you implemented in your first project. And the idea is, as the name suggests, a mutually exclusive lock means that it can be used by a thread, one thread at a time. That's the idea. And here's a silly example of two children playing, and you know, they have to take turns in order to hit this ball, and obviously, you don't want both of them hitting the ball at the same time. Not good for the game and not good for the safety of the children either. That same, same thing applies to the mutual exclusion lock that we use in parallel

programming. The idea is that a thread that wants to modify data has to make sure that when it is modifying the data, nobody else is going to be accessing that particular data structure. And therefore it is going to get a mutual exclusion lock, it knows that nobody else is going to be messing with it. Then it can modify the data and then release the lock. And similarly, if another thread wants to read that data and wants the assurance that nobody is going to be modifying this data while it is reading it, it can get an exclusive lock, access the data, read it and then release it. That's the idea behind the mutually exclusive lock.

You can also have a **shared lock**. Now, what that means is that this **lock is something that allows multiple threads to access the data at the same time**. Well, under what conditions would that be meaningful? Well, here is an analogy again. If there is a newspaper, and multiple people want to read the newspaper at the same time, perfectly fine to do that, right? That's the same sort of thing that happens often in parallel programming. That you have a database, and there are records in the database that multiple threads want to inspect. But they want to make sure that while they're inspecting it the data itself is not going to be changed so a shared lock is something that allows multiple readers to access some data with the assurance that nobody else is going to be modifying the data. So these are two different types of locks that you might have that might be useful in developing multi-threaded shared-memory programs.

2. Synchronization Primitives

Synchronization primitives for shared memory programming



Barriers - like a reverse from a semaphore, will block all threads until n threads arrive at this point.

Another kind of synchronization primitive that is very popular in multithread apparel programs, and extremely useful in developing applications, especially in the scientific domain, is what is called **barrier synchronization**. The idea here is that there are multiple threads and they are doing some computation. And they want to get to a point where they want to know where everybody else is at that, at that point of time. And they want that insurance that everybody has

reached a particular point in the respective computations so that then they can all go forward from, from this barrier to the next phase of the computation. Now I'm sure that you've gone to dinner with your friends and one of the experiences that you may have had is that, and you may have a party of four or five people that are going for dinner. Two or three of you are showing up at the dinner restaurant. And the usher says "wait, you know, do you have the entire members of your party here? If they're not here wait til the other members of the party show up, so that I can seat you all at the same time". And that's sort of the same thing that's happening with barrier conditions. It is possible that you know thread t1 and t2 arrive at the barrier, meaning they completed their portion of the work. They've gotten to this barrier but the other threads that are lagging behind and those shirkers are going to eventually show up but they're not here yet, and until everybody shows up nobody can advance to the next phase of the computation. So that's the idea, behind barrier synchronization, exactly similar to the analogy that I mentioned here. So we looked at two types of synchronization primitives. One is the lock, and the other is the barrier synchronization. Now, these are concepts I am expecting that you know already. If you find that these two concepts are either new to you, or you would like some refresher for that, I strongly advise you to go and, and take a look at the review lesson that we have on multithreaded programming. Now that we understand the basic synchronization primitives that are needed for developing multithreaded applications on a shared memory machine. It's time now to look at how to implement them. But before we do that, let's do a quiz to get you in the right frame of mind.

3. Programmer's Intent

Question

P1
Modify struct(A)

P2
Wait for mod;
use struct(A);

Assuming only read/write atomic
instructions is it possible to achieve
programmer's intent? Yes No

Code:

To get you primed up to answer this question, let's first discuss a little bit about the instructions as architecture of a processor. In the instruction set architecture of a processor, instructions are atomic by definition, or in other words if you think about Reads and writes to memory which are usually implemented at loads and stores, and the instructions have architecture for processor. Those are atomic instructions, and what that means is, during the execution of either a load instruction or a store instruction or also, as you might think about them, read or write instruction, the memory. The processor cannot be interrupted. That's important that's the **definition of an atomic instruction that the processor is not going to be interrupted during the execution of an instruction**. Now the question that I'm going to ask you to think about is, if I have a multi-threaded program And in that program, there is a process of P1, which is modifying a data structure A, and there is a process of P2. That is waiting for the modification by P1 to be done, and after the modification is done, it wants to use that structure. Very natural, to think about situations in which you have this kind of a producer-consumer relationship. This guy is the producer of data, this guy is the consumer of data. And the consumer wants to make sure that the producer is done producing it before he starts using it. Quite natural. Now, given that the instructions of architecture is only read and write atomic instructions, The question that I'm going to pose to you is, is it possible to achieve the programmer's intent that I have embodied in this code snippet here? And, you know, the the answer is a binary answer, yes or no. And and if you, if you answer yes, I would like to see a code snippet that you think would make this particular code snippet work correctly on a multiprocessor.

Question

P1
modify struct(A)

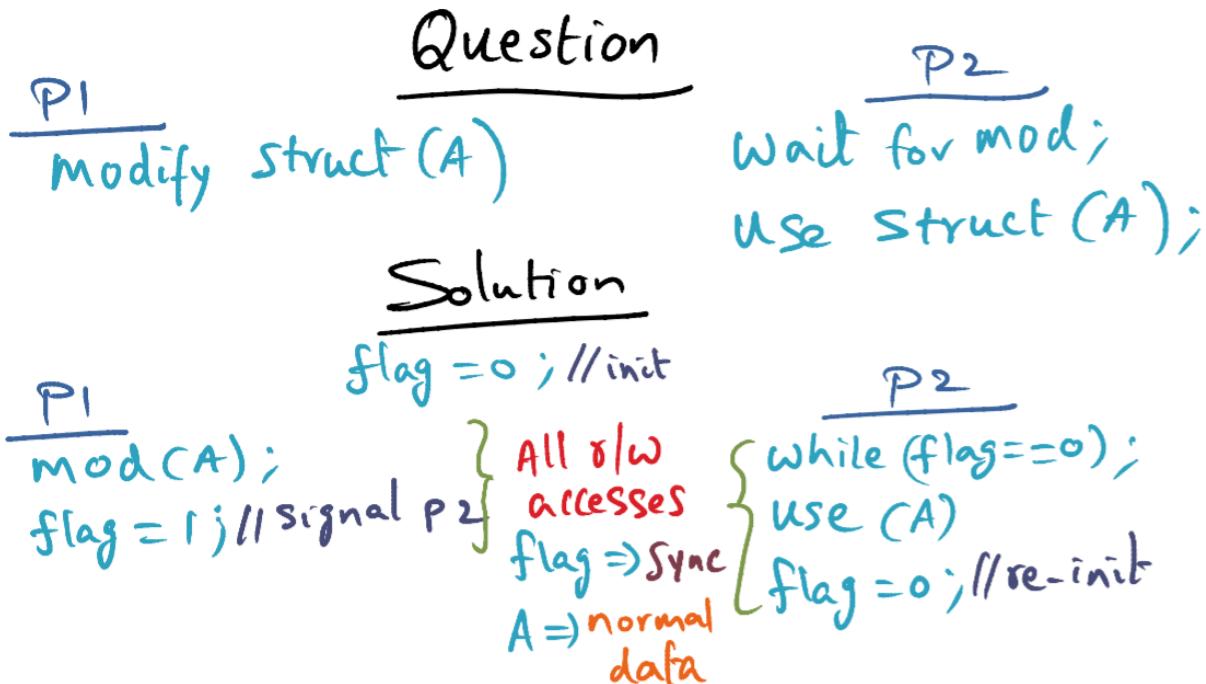
P2
wait for mod;
use struct(A);

Solution

<u>P1</u> mod(A); flag = 1; // signal P2	flag = 0; // init } All r/w accesses flag => Sync A => normal data	<u>P2</u> while (flag == 0); use (A) flag = 0; // re-init
--	---	--

If you answered yes, then you and I are on the same wavelength. And in the next few panels, I'm going to show you how this particular programming construct that a multithreaded program may execute in terms of producer and a consumer, can actually be accomplished with simple read/write atomic operations available in the instruction set of a processor. The solution, it turns out is surprisingly very simple. The idea is that between p1 and p2, I'm going to introduce a new variable, a shared variable, and that variable, I'll call it a flag. And I'll initialize this flag to be zero to start with. And the agreement between these two. Producers in consumer is that the producer will modify the data structure that he wants to modify and once he's done with the modification he will set this flag to be a one. And that's the signal to p2 that this guy is done with the modification. Now, what is p2 doing? Well, p2 is basically waiting. For this flag which initial, initially the flag was initially zero. And basically, the processor P2 is waiting for the flag to change from a zero to a one. Now once p1 is done with its modification, it's going to set this flag to a, to a one. And that's the signal that this guy's waiting for. And as soon as this flag changes to a one. Then he'll break out of this spin loop here, and he is now ready to use the real structure. And once he is done using the real structure, he can flip it back to zero, to indicate that he is, that he is done using it. So that the next time the producer wants to modify it again he can do that. So that's sort of the solution. Now, let's analyze the solution and see why it works. It will just atomic reads and writes.

4. Programmer's Intent Explanation



Now the first thing to notice is that all of these are read and write accesses. There's nothing special about them. This is, this is going to be modifying data using loads and stores, and this is storing a value into it, and this is reading a value and using a value. So all of these are normal read write accesses, but there is a difference between the way the program uses this flag variable versus this data structure. **The flag variable is being used as a synchronization variable.** And that's a, a secret that only this P1 and P2 know about. That this, even though innocuously it looks like a simple Integer variable that is used in a program where there is special semantic for this particular variable so far as this, this program is concerned. P1 and P2 know that this is the way by which their signalling each other, that something that this guy waiting on is available from P1. Right? And so its a synchronization variable. On the other hand, the data structure A is a normal data. But, both accessing the synchronization variable and normal data is being accomplished by simple read write accesses that's available in the processor. And that's how we're able to get the solution for this particular question.

It's comforting to know that **atomic read and write operations are good for doing simple co-ordination among processes** as we illustrate it through this question. And in fact, when we look at certain implementation of barrier algorithms later on, you'll find that this is all that is needed from the architecture in order to implement some of them. **But now, how about implementing a synchronization primitive like a mutual-exclusion lock? Are atomic reads and writes sufficient to implement a synchronization primitive like a mutual-exclusion lock?** Let's investigate that.

5. Atomic Operations

Atomic operations

```
LOCK(L):  
    if (L == 0)  
        L = 1;  
    else  
        while (L == 1);  
        //wait  
        go back;
```

Atomic operations

```
LOCK(L):  
    if (L == 0)  
        L = 1;  
    else  
        while (L == 1);  
        //wait  
        go back;
```

Let's look at a very simple implementation of a mutual exclusion lock. In terms of the instructions that the processor will execute in order to get this lock, will be to come in and check if the lock is currently available and that is done by this check. And if it is available then we're going to set it to one to indicate that, "I've got the lock, nobody can get it." That's the idea behind this check and then setting this to one. On the other hand, if somebody already has the lock L is going to be one and therefore I'm going to wait here until the lock is released. And once the lock is released, then I can go back and check again, to make sure that the lock is available and set it to one. So this is the basic idea. Very simple implementations of this lock. And, and how will I know that the lock has been released? Unlocking this is a very simple operation again. All that you have to do is reset this L to zero, and that'll indicate that the lock has been released. So, if I am waiting here, and somebody else has got the lock, they going to come and unlock it by setting it to zero. And that way, I will know that it has been released. I can go back. I double-check to make sure it is still zero because somebody else could have gotten in the middle. If nobody else has gotten it, then I can set it to one. **So this is the idea of a simple minor implementation of a lock algorithm.**

Atomic operations

```
LOCK(L):  
    if (L == 0)  
        L = 1;  
    else  
        while (L == 1);  
        //wait  
        go back;
```

UNLOCK(L):

```
L = 0;
```

possible to implement with atomic read/write?

Is it possible to implement the simple implementation of the lock using atomic reads and writes alone? Let's talk through this implementation here.

Atomic operations

LOCK(L):

```
if (L == 0) {  
    L = 1;  
}  
else  
    while (L == 1);  
    // wait  
    go back;
```

UNLOCK(L):

L = 0;

} group of
3 instr

need to be atomic

need new RMW
semantic atomic
instruction

Now, if you look at this set of instructions that the processor has to execute in order to acquire the lock. It has to first read L from memory, and then check if it is 0. And store that new value which is 1 into this memory location. That's a group of three instructions that the processor has to execute and the key thing is these three instructions have to be executed atomically in order to make sure that I got the lock and nobody else is going to interfere with my getting the lock. And as we know, read and write instructions by themselves are atomic, but a group of reads and writes are not atomic, and therefore what we have here is a group of three instructions and we need them to be atomic. What that means is we cannot have just reads and writes as the only atomic operations if we want to implement this lock algorithm. And we need a new semantic for an atomic instruction, and the semantic is what I call the **read modify write operation (RMW)**. Meaning that I'm going to read from memory, modify the value and write it back to memory. So that's the kind of instruction that is needed in order to ensure that we can implement a lock algorithm.

Atomic RMW instructions

Test-and-set (<mem-loc>)

return current value in <mem-loc>

Set <mem-loc> to 1

Fetch-and-inc (<mem-loc>)

return current value in <mem-loc>

increment [<mem-loc>]

Generically

Fetch-and- ϕ

Now several flavors of read-modify-write instructions have been proposed and/or have been implemented in processor architectures. And we will look at a couple of them.

The first one is what is called a **test and set** instruction. The idea here is, the test and set instruction take on a memory location as an argument. And, what it does is that it returns the current value that is in this particular memory location and also sets the memory location to a one. So, these two things are being done. That is, getting the current value from memory and setting it to one, is being done atomically. That's the key thing. **That it is testing the old value and setting it to this new value, atomically.**

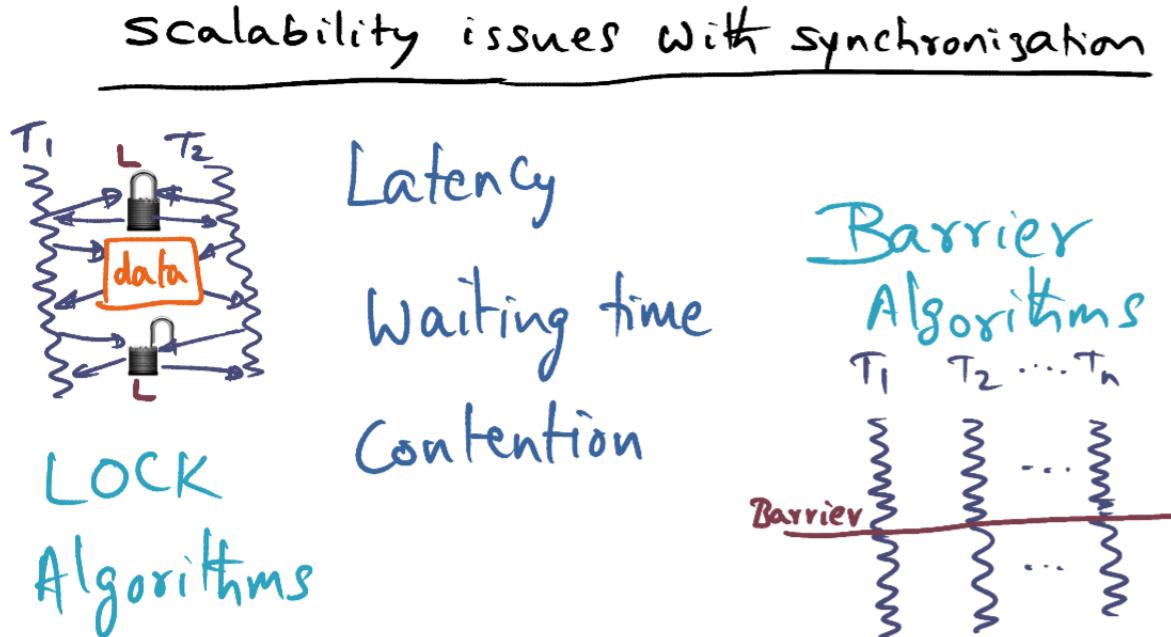
Another atomic Read Modify Write instruction that has been imposed and/or implemented is what is called a **fetch and increment** instruction. And this takes on again, a memory location of an argument, and what it is going to do is, **it is going to fetch the old value of what was in the memory. And then increment the current value that is in the memory by one or whatever value.** So it could be that this may take on an extra argument to indicate how much it is going to change it by. But in the simple version, it might simply imply increment in the simple version it might simply increment the current value that is in the memory location by one.

As I said before, there have been several flavors of read modify write instructions that have been proposed in the literature. And often generically these read modify instructions are called **fetch and phi instructions. Meaning that it is going to fetch an old value from memory. And do some operation on that fetched value and write it back to memory.** So, for instance, fetch an increment is one flavor of that. There are other flavors like fetch and store,

fetch and decrement compare and swap, and so on. And you can read about that in the papers that we've identified for you.

Okay, now that we have an atomic read modify write instruction available from the architecture, we can start looking at how to implement the mutual exclusion lock algorithms. Now, I gave you, of course, a very simple version of it, we'll talk more about that in a minute. And, and I'm sure that in the first project when you implemented the mutual exclusion lock, you did not care too much about the scalability of your locking implementation. Now if you are implementing your mutual exclusion algorithm on a large-scale shared-memory multi-processor, let's say with 1000's processes. You'll be very worried about making sure that, that your synchronization algorithm scale and scalability issues fundamental to the implementation of synchronization algorithms.

6. Scalability Issues With Synchronization



Now let's discuss some of the issues with scalability of the synchronization primitives in a shared memory multiprocessor. Now we already saw that locks, both mutual exclusion as well as shared lock is one type of a synchronization operation. And we also saw that barrier algorithms is another type of synchronization operations. And when you look at both of these types of synchronization perimeters that a parallel operating system is going to provide for application programmer developing multi-threaded applications.

The sources of inefficiencies that come aboard is first of all **latency**. What do we mean by that? Well, **If the thread wants to acquire this lock, it has to do some operation**. Has to go to memory, get this lock, and make sure that nobody else is competing with it. And, so that's the **the latency that is inherently what is the time that is spent by a thread in acquiring the lock**. That's what we mean by latency. Well to be more precise what we mean is that latency is saying, lock is currently not being used. How long does it take for me to go and get it? That's really the key question that latency is trying to look at.

The second source of scalability with synchronization is the **waiting time**, and that is **if I want to go and get the lock, how long do I wait in order to get that lock?** Well, clearly this is not something, that you and I as the OS designer have complete control over, because it really depends on what these threads are doing with this lock. So for instance, if this thread acquires this lock, and then it is modifying the data for a long time before releasing it, and if another thread comes along and wants the team lock, it's going to wait for a long time. **So the waiting time is in the purview of the application**. And there's not much you can do, as an OS designer, in reducing the waiting time.

The third source of the unscalability of locks is **contention**. What we mean by that is. If currently some guy is using the lock, and he releases it, when the lock is released, it's now up for grabs. Maybe there's no, I've shown you only one thread here, but maybe there's a bunch of threads waiting here to access this particular lock. **If they're all waiting to access this lock, they're all contending for this lock. And how long does it take in the presence of contention for one of them to become the winner of the Lock and the others to go away.** So that's the contention part of it, implementing a synchronization primitive.

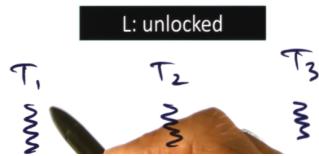
And all of these things, latency, waiting time, and contention even though I mentioned it in the context of a mutual exclusion lock appear when you're talking about barrier synchronization, algorithms, or shared locks. So latency and contention are two things as all designers, we have to be always worried about, and implement scaleable versions of synchronization primitives.

7. Native Spinlock

Naive Spinlock (Spin on T+s)

LOCK(L):

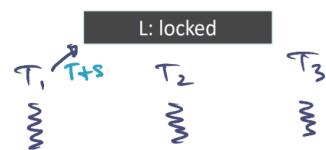
while ($T+s(L) == \text{locked}$);



Naive Spinlock (Spin on T+s)

LOCK(L):

while ($T+s(L) == \text{locked}$);



Let's start our discussion with the world's most simplest and naive implementation of the lock, and we're calling it **spinlock** because, as you're going to see a processor that is waiting for lock has to spin, in order to spin, meaning, doing no useful work, but it is waiting for the lock to be released.

And the first one that we're going to look at, is what is called **spin on test and set**. The idea is very simple and straightforward. There's a shared memory location, L and it can have one of two possible values. Either unlocked or locked. And let's say that at the beginning, we've initialized unlocked, so nobody has the lock. And the way to implement this naive spinlock algorithm is the following. What you do is, you go in and check, using test and set primitive, the memory location, L. So when you call this lock primitive the lock primitive executes this instruction test and set of L, and what that is going to do is, it's going to return the old value from L, and set it to the new value which is locked, that's going to be done automatically, we that from the architecture, that it is going to provide you that, that is a primitive, and so now, if we find that this test and set instruction execution returns the value locked, it means that somebody else has bought the lock. And therefore, I cannot use it and I'm going to basically spin here. That's why it's called **spin on test and set, so you're basically spinning waiting for this test and set instruction to return to me**. A value that says, the old value is unlocked. If I, if it gives me, the old value is unlocked, then I know I won. But if I don't, then I, basically, I'm going to wait here. That's why it's called spinning on test and set. So let's put up some threads here that are trying to get this lock. And, so let's say that T1 is the first one to make a test and set call on this lock, and it finds it unlocked, and therefore, it locks it. And once it locks it, T1 knows that it's got the lock.

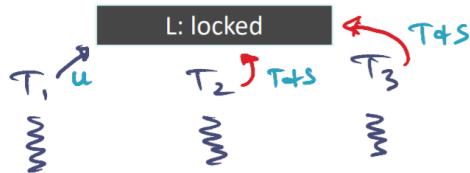
Naive Spinlock (Spin on T+S)

`LOCK(L):`

`while (T+S(L) == locked);`

`UNLOCK(L):`

`L = unlocked;`



So, it's got the lock, it can go off to mess with the data structure that it wants to mess with, and that is good. So far as T1 is concerned. In the meanwhile, T2 and T3 may come along and say well, we also want the lock, and they also execute the same lock algorithm. And when they execute the lock algorithm, they're going to do the Test-and-Set and you know that the Test-and-Set when they do that, the old value that is going to be returned for T2 or T3 is going to be that the value L is locked and therefore these two guys, both T2 and T3 are stuck here. How long are they going to be stuck? Until this guy releases the lock, the way to do that is very simple. So he comes along and calls an unlock function, and what the unlock function does, is it basically goes in and clears this lock. Meaning it resets this lock to the unlocked state. And so once it does that, then this lock becomes available.

Naive Spinlock (Spin on T+S)

`LOCK(L):`

`while (T+S(L) == locked);`

`UNLOCK(L):`

`L = unlocked;`



So, it becomes unlocked and at this point, T2 and T3 were stuck here. When they tried to do this testing set again, they're going to find, at least one of them hopefully exactly one of them, is going to find that, that the lock is unlocked and therefore they're going to get it. And one of them will get it, and will go on to executing whatever code they want to do, and the protection of the lock, and so only exactly one of T2 or T3 will win, because that's semantic of test and set. So that's the world's simplest lock algorithm, spinning on test and set.

8. Problems With Native Spinlock

Question

What are the problems with Naive Spinlock?

- Too much contention
- Does not exploit caches
- Disrupts useful work

Question

What are the problems with Naive Spinlock?

- Too much contention
- Does not exploit caches
- Disrupts useful work

If you checked all three of them, you're exactly on the right track. Let's talk about it. First of all, you know that with this, with the naive implementation there is going to be too much contention for the lock when the lock is released. Because everybody, both t2, and t3 in the previous example, jumped in and started looking at the test and set instructions, trying to acquire the lock. And there are thousands of processes, everybody is going to be executing this test and set

instruction, so there is going to be plenty of contention on the network, in order to get to that shared variable, that's the first problem.

Now, let's talk about why the second answer is also the right answer. You know from the previous lesson that a shared memory multiprocessor has private caches associated with every one of the processors. And it is often the case that the caches may be kept coherent by the hardware. Now if the private caches are associated with every processor and if a value from memory can be cached in that, there is an issue with test and set instruction. And that is, **test and set instruction cannot use the cached value, because it has to make sure that the memory value is modified atomically when it is inspecting the memory**. And therefore, by definition, a test and set of instructions are not going to exploit caches, it is going to bypass the cache and go to memory, in order to do the test and set operation. And therefore yes, this is also true, that the spin algorithm that I gave you, spin on test and set, is not going to be able to exploit the caches.

The third problem is, is the fact that it might disrupt useful work. And it's also a good answer and the reason is that when a processor releases the lock. After releasing the lock, that processor wants to go on and do some useful work. And similarly. If, let's say there are four processors trying to acquire the lock. Only one of them is going to get it, and the others are going to have to back off because they're not going to have the lock. Now one guy that did get the lock has useful work to do. But, **because there's a lot of contention, the guy that can actually do useful work is being impeded from doing useful work**. In all the other processors trying to go and get the lock when it is not available.

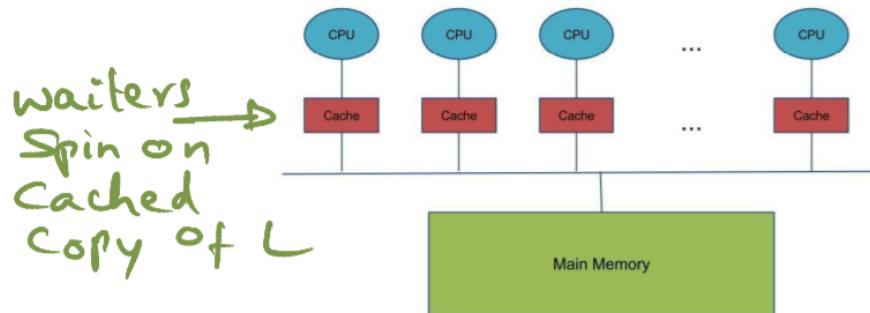
So, this is really the problem, that the test and set instructions, because it is bypassing the caches, it's, first of all, causing a lot of contention on the network and it is also impeding some of the useful processors from carrying on with its work. Which may advance the cause of the parallel program. So all of these are good answers in terms of the problems with this naive spinlock.

9. Caching Spinlock (Spin on read)

Caching spinlock (Spin on read)

LOCK(L):

```
While (L == locked);
    if (T+S(L) == locked) go back;
```



```
// Change the slide code as below
// The indent confused me every time looking at it
while(true){
    // If L cache is locked, keep looping on it
    while(L == locked);

    // once L != locked, check t+s
    if(T+S(L)==locked){
        continue;
    }

    break;
}

// do stuff
```

Now let's look at how we can exploit the caches available. Now, it is a fact that a test and set instruction has to necessarily go to memory when we want to acquire the lock, we have to execute a test and set instructions so that we can atomically make sure that exactly one processor gets the lock. **But on the other hand, the guys that don't have the lock could exploit the caches in order to wait for the lock.** And that's why this particular algorithm that I'm going to describe to you is what is called **spin on read**, and the assumption here is that you have a shared memory machine in which the architecture is providing cache coherence, or in other words, through the system bus or interconnection network, the hardware is ensuring that the caches are kept coherent. Well that gives us an idea as to how we can exploit the caches. **The waiters, instead of executive a test and set instruction that has to go to memory, they can spin locally on the cached value of the lock. because when you are spinning on the local cached value of the lock. If that value changes in memory, these guys are going to notice that. That's the principle behind the cache coherence that is implemented in hardware.** And so we can exploit that fact in implementing a more efficient way of spinning. Which is called spin on read.

The idea is that the lock algorithm, the first thing it's going to do is go and do a check on the memory location to see if it is locked. So this is a normal atomic read operation that is being done, not a test and spin operation, so if it is not in the cache, you're going to go to memory and bring it in, and once you bring it in, so long as this value doesn't change, we're going to basically looking at the value that is in that cache in order to do the checking. And I'm not going to go to the bus and therefore I'm not producing any contention on the network. And there could be any number of processes waiting on the lock simultaneously. No problem with that because all of them are going to be spinning on the local value of L in their respective caches. And so if there is one processor that's actually doing useful work and it has to go to memory, it's not going to find that to be a problem. No contention on the network from the waiting processors because of this.

Now, if the one processor that was having the lock eventually releases it, everybody's going to notice that. And so if I'm waiting for the lock, and I've been spinning here locally in my cache when the lock is released, I'll notice that through the cache coherence mechanism as I'll break out of this spin loop. But immediately, I want to check if the lock is available by doing this test and set and get it uniquely for myself. So multiple processors are trying to execute this testing set simultaneously. It's possible somebody else is going to beat me to the punch and if that happens, I simply go back and, and, and do the grouping on my private copy of L and wait for the guy who beat me to the punch to release a lock eventually. So that I can get it. So that's the idea. **The idea is that you spin locally. When you notice that the lock has been released you try and do a test and set. If you get lucky you win, if you lose you go back and spin again locally.** So that's the idea behind spinning on reading. The unlock operation of course is pretty straightforward. The guy that wants to unlock is simply going to change the memory location to indicate that L is no longer locked. So that's all it has to do. And then the other processes can observe it through the cache coherence mechanism, and be able to acquire the lock. **But note what happens when the lock is released. When the lock is released, all the processes that are stuck here in the spin loop, are going to go and try to do this test and**

operation at the same time, and we know that test and set have to bypass the cache, everyone is hitting on the bus. Everybody is hitting on the bus, trying to go to memory, in order to do this test and operation. And so that essentially means that in a right invalidate this cache coherence mechanism is going to result in $O(n^2)$ bus transactions. For all of these guys to stop chattering on the bus, because every one of these test and set instructions is going to result in invalidating the caches, and as a result, you have an order of n squared operation that is going to result when a lock is released, where n is the number of processors that are simultaneously trying to get the lock. And, obviously, this is impeding that one guy that got the lock and can actually get some useful work done. And this is clearly disruptive. And earlier one of the things that we said is that we want to avoid or limit the amount of disruption to useful work that can be done by the process that acquired the lock.

10. Spinlocks With Delay

Spinlocks with Delay

Delay after lock release

```
while((L == locked) ||  
     (T + s(L) == locked))  
{  
    while (L == locked);  
    delay (d [Pi]);  
}
```

Delay with exp. backoff

```
while  
    (T + s(L) == locked)  
{  
    delay (d);  
    d = d * 2;  
}
```

Now, in order to limit the amount of contention on the network when a lock is released, we're going to do something that we often do in real life. procrastination. So basically, the idea is the following: **Each processor is going to delay asking for the lock, even though they observe that the lock is released.** Then I will immediately go and try to get the lock. They're going to wait for a little bit of time. It's sort of like what happens at rush hour. If you find that the traffic is too much, you might decide that I don't want to get on the highway right now. I'm going to delay a little bit so that I don't have to spend as much time on the highway. So that's sort of the same thing that is being proposed here, and this is what is called **spinlocks with delay**. Let's discuss **two different delay alternatives**.

In the first one, you're here. You found that you did not get the lock and therefore, you're here locally spinning in your cache, waiting for the lock to be released. Normally what you would have done, when the lock is released, is go back out, break out of this loop and go back and check if you can get the lock again. But what we're going to do is, instead of doing that, when we break out of this loop, meaning that the lock has been released, I'm not going to immediately go and check to see if I can get the lock. I'm doing to delay myself by a certain amount of time. And you notice that **the delay is conditioned by what processor id I have**. So **every processor is waiting for a different amount of delay in order to contend for the lock**. So since the delay is being chosen differently for each processor, even though all of them notice that the lock has been released simultaneously, only one of them will go and check it. And so we are sort of sequentializing the order in which the processors that are waiting for the lock are going to check whether the lock is available. So that is one possible scheme for delaying. Now the problem with this is it's a static delay, right? So every processor has been preassigned a certain amount of delay, which means that even if the lock is available, I may not immediately go and check because my delay may be very high compared to some other processor. And that's always an issue when you have static decision-making.

What we can do is instead **make the decision dynamically**, and what we're going to do is, when we notice that we don't have the lock, we're going to delay ourselves by a certain amount of time before we try for the lock again. You notice that if you're going to delay checking for whether I have the lock or not. It's not super critical that, that you spin locally or go to memory. But in this example, I'm making it very simple by saying that if you don't get the lock, just delay a little bit before you try for this lock again. And the idea here is this delay is, is some small number to start with. But suppose I go and check and I find it again to be locked. Now, what I'm going to do is the next time around, I'm going to increase the delay. That's why it's called **exponential backoff**. So I'm increasing the delay, doubling the amount of delay that I'm going to do. So that the next time, if I don't find the lock to be available, I delay by twice the amount from the previous time. And this is essentially saying that **when the lock is not highly contended for, I'm not going to delay myself too much. I'm going to immediately go and get it. But on the other hand, if I go back again and again, and every time I go and check, I find it is locked, I'm going to increase the amount of delay**. Because that's saying that a lot of people are contending for the lock at the same time. And therefore, in order to make sure that we are being sensitive to the contention that is there for the lock, we increase the amount of delay that we're experiencing. Now one nice thing about this simple algorithm that I've shown you is that I'm not using the caches at all. **And if the processor happens to be a non-cache coherent multi-processor, this algorithm will still work**. Because we're always using test and set, and not using just loading from the memory. Because if it is not a cache-coherent multiprocessor, your private cache is now going to be coherent with respect to memory. And so you have to execute the test and set. But you don't want to do it all the time. And this delay makes sure that you can reduce the amount of contention on the network.

Generally speaking, if there's a lot of contention, then the static assignment of delay may be better than the dynamic exponential backoff. But in general, any kind of delay, any kind of procrastination, will help a lock algorithm better than the naive spin lock that we talked about.

11. Ticket Lock

Ticket Lock

```
struct lock{  
    int next-ticket;  
    int now-serving;  
};  
  
acquire-lock(L):  
    int my-ticket = fetch-and-inc(L->next-ticket);  
  
    loop  
        pause (my-ticket - L->now-serving);  
        if (L->now-serving == my-ticket) return;  
  
release-lock(L):  
    L->now-serving ++;
```

16 25

my-ticket

↑
got it !

Up to now, what we've talked about is how to reduce the latency for requiring the lock and the contention when the lock is released. So far we've not talked about **fairness**. What do we mean by fairness? Well, if multiple people are waiting for the lock, should we not be giving the lock to the guy that made the lock request or tried to acquire the lock first. **Unfortunately, in Spinlock, there is no way to distinguish who came first. Because, as soon as the lock is released, they are going to try and grab the lock.** And, it's entirely up for grabs, as to, who may be the winner. So next, we're going to do is, we're going to look at a way by which we can, we can ensure fairness in the lockout position. Now many shops and restaurants, busy ones, that is, often use a ticketing system to ensure fairness for those who are waiting to get served. So for instance, in this example here let's say, I walk in the deli shop. And my ticket is 25, and I notice that currently they're serving 16. So I know that I have to wait for a little bit of time. And you know, once my number comes up, I can get served. So this is actually, and if I know that there at least nine

people ahead of me who need to be served before my turn comes up. And by similar argument. If people come after me, I know that they're not going to be served before me.

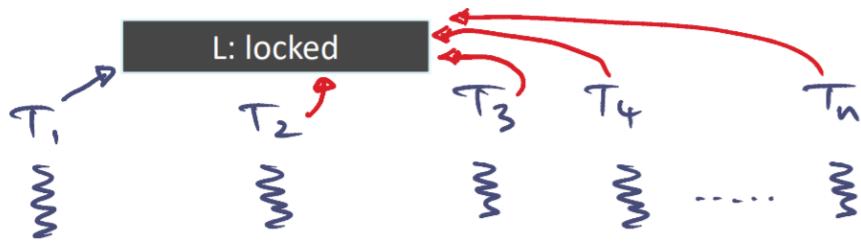
That's the basic idea that we're going to use in this ticket lock algorithm. The ticket lock algorithm is basically implementing what I described to you as to what happens in a deli shop. **The lock data structure has two fields to it, a next-ticket field, and a now-serving field. And the lock algorithm, in order to acquire a lock, what I'm going to do is I'm going to mark my position.** And the way I do that is I'm going to get a ticket just like when I walk in a deli shop. I get a unique ticket, I get a unique ticket by doing a fetch and increment on the next ticket field of the log data structure, and when I do the structure increment, I get a unique number and this number is also advanced, exactly like how it would happen in a deli shop. And once I have my position marked, as to when I can get my lock, I can then wait by procrastination. And what I'm doing here is pausing to see if I've won my lock by an amount that is proportionate to the difference between my ticket value and who is being served currently. And after there's an amount of dealing, I'm going to go and check if the now serving value equals my ticket value. And if, if it is, then I'm done, I can return. Otherwise, I go back to looping. So basically I'm looping, waiting for my number to be up so that I can assume that I've got the lock. And how am I going to get, get this information that, that my ticket is up for serving? That is going to be done with the current holder of the lock. He's going to come and release the lock, and when he releases the lock, he's going to increment the now_serving value in the lock data structure, and that's all, eventually, the now_serving will advance to be equal to my_ticket, and I'll get the ticket, and then I can return from the acquire lock. Now, this algorithm is good, in that it preserves fairness, but you notice that **every time the lock is released, there is now serving value that is in my local cache is going to be updated with a cache coherence mechanism, and that's going to cause contention on the network.**

So on the one hand fairness is achieved and on the other hand, we have not really completely gotten rid of the contention that can happen on the network when the lock is released.

12. Spinlock Summary

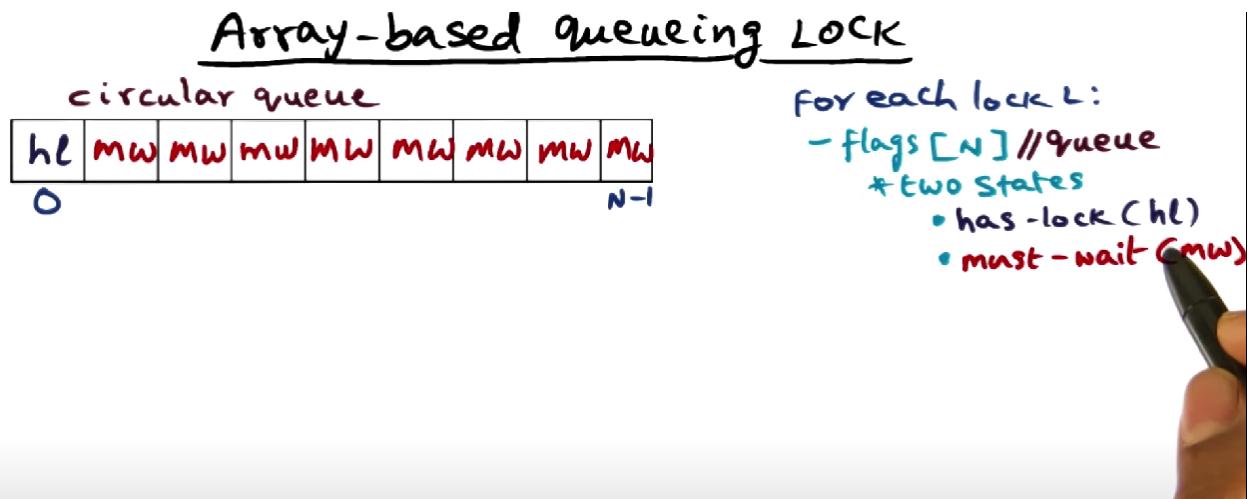
Spinlock Summary

- 1) Read + T+S } no fairness
 - 2) T+S with delay }
 - 3) Ticket lock - fair but noisy
- ideally, T_1
Signals ONLY
next thread!



So, to summarize the Spinlock algorithm that we've seen so far, we saw that **1) spin on read, and 2) spin on test and set, and 3) spin on test and set with a delay**. All of these spin algorithms, there's no fairness associated with them. And if you think about the **ticket lock algorithm**, it is fair but it is noisy. So, all of them are not quite there yet in terms of our twin objectives of reducing latency and reducing contention and if you think about it, let's say that you know, that currently this T_1 has got this lock. And all of these guys are waiting for this lock to get released. You know when T_1 releases the lock, exactly one of them is going to get it. Why should all of them be attempting to see if they've got the lock? **Ideally, what we would want is that when T_1 releases a lock, exactly one guy, one of these white reading guys is, is a signal to indicate that you've got the lock.** Because exactly one guys can, can get the lock to start with. And therefore, ideally T_1 should signal exactly on the next thread and not all of them. Now, this is the idea behind queueing locks that you're going to see next.

13. Array-Based Queueing Lock



We will discuss two different variants of the queueing lock.

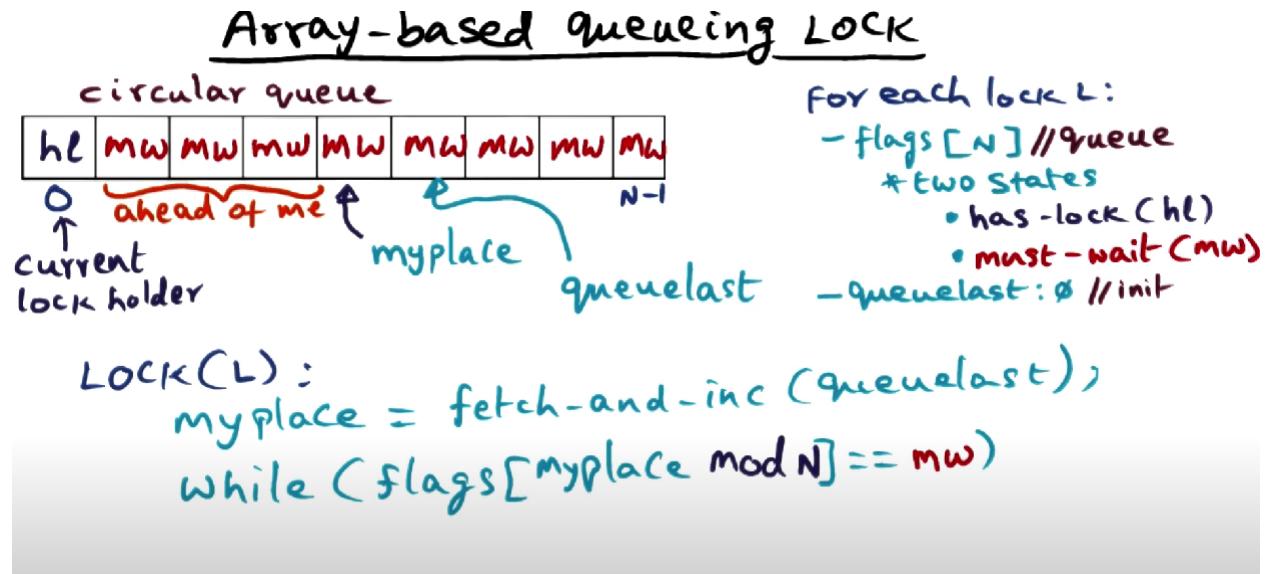
Note: Processors are physical entities and threads of execution are logical ones. At any instance, a processor (core) is executing code of one thread. The professor uses processors here because these spin locks only make sense if there are at least two processors.

The first one we'll talk about is the **array-based queueing lock**, and this is due to Anderson. And I'll refer to it as **Anderson's lock** later on as well. Associated with each lock L, is an array of flags. And the size of this array is equal to the number of processes in the SMP. So if you have an N-way multiprocessor, then you have N elements in the circular flags array. And this flags array serves as a circular queue. For N-queuing the requesters that are requesting this particular lock L. So every lock has associated with this flags array and it's really intuitive that since we have at most we have N processors in this multiprocessor. We can have at most N requests simultaneously waiting for this particular lock so the size of the data structure, **the flags data structure is equal to N where N is the number of processors in the multiprocessor**.

Now each element in this flags array can be one of two states. One state is the **has-locks state**. And the other state is a **must-wait state**. Has-lock says that whoever is waiting on a particular slot has the lock. So this particular entity let's say, is "hl". **And that means that whichever processor happens to be waiting on this particular slot is a current winner of the lock and is using the lock**. On the other hand, **must-wait is indicating that if a processor has must-wait as the entry in this particular element of the array, and is waiting on this particular slot, means that the processor has to wait**. You guessed it. There can be exactly one processor that can be in the "hl" happy state because it's a mutually exclusive lock. And therefore, at most one processor can have a lock at a time, and all the others should be waiting. And, so what we do is, in order to, when we get this lock. To initialize the lock, what we do is that we initialize the lock data structure, this array data structure. The flags of the array data structure represent a circular queue by marking one slot as "hl". And all

the others as must-wait. An important point I want you all to notice is that **the slots are not statically associated with any particular processor**. As requesters come in, they're going to line up in this flags array at the spot that they get in the next available slot. **The key point is that there is a unique spot that is available for every waiting processor. But it is not statically assigned** and we'll see how do requests get formed using this circular queue in a minute.

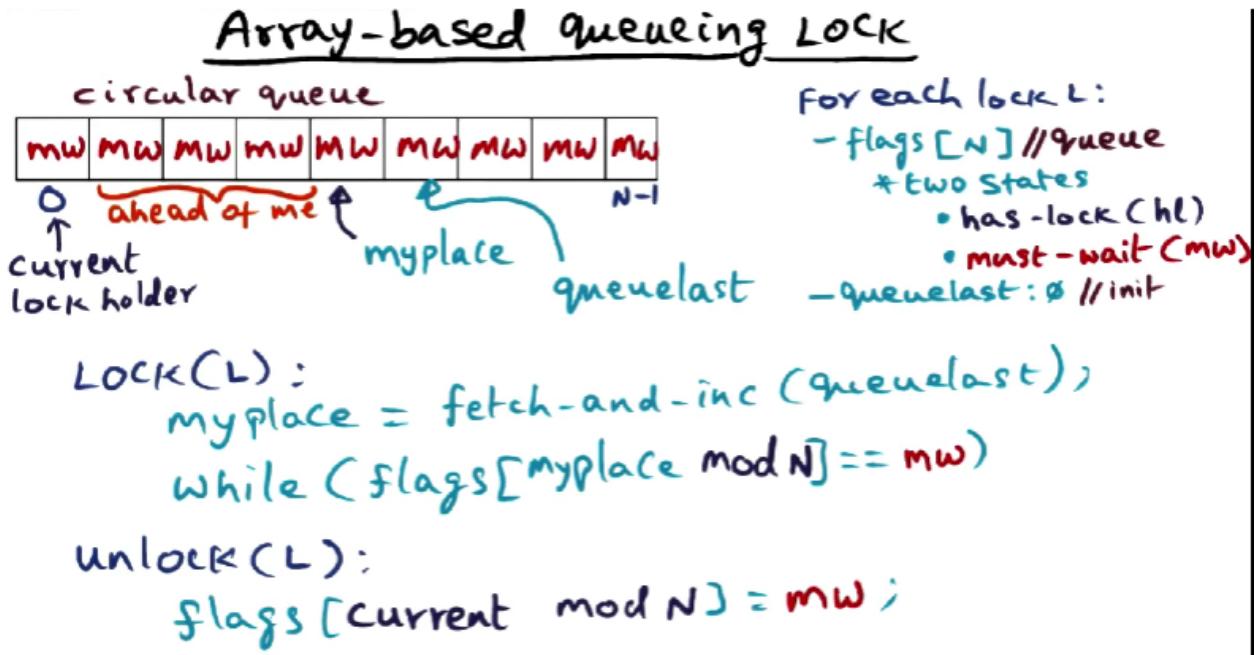
14. Array-Based Queueing Lock (cont)



Since we've initialized this array with "hl" in the first spot and "mw" in all of the other spots of this array, to enable the queuing what we will do is associated with each lock another variable, which is called a **queuelast** variable. And this queuelast variable is initialized to zero. And so these two are the two data structures associated with every lock. So every lock that you have in your program, the operating system is going to assign two data structures for you. One, which is the circular queue, represented by the flags array. And the other is the queuelast variable, which is saying, what is this part that is available for you to queue yourself in this, in this particular array? So as you can see, since there is no lock request yet, we just initialized the queue, the first guy that comes around to ask for the lock will get it, and, and he will queue himself here and he will get the lock as well. So let's say some processor came along, and, and made a lock request. It's going to get it immediately because there's no locks request currently pending. And so it's got this position and it's got the lock and **what will happen is that the queuelast variable will advance to the next spot to indicate that future requesters have to start queuing up from here**. And now this current lock holder has got the lock and he can go off in the critical section and do whatever he wants in terms of managing or messing up with the data structure that is governed by this particular lock. Let's say that at some point in time, I come along and request the same lock. Now depending on who else got ahead of me at the point that I made that lock request, there may be some number of people that are lined up ahead of me, and where ever queuelast is pointing is my place. And, and so this is where I'm going to queue myself, waiting

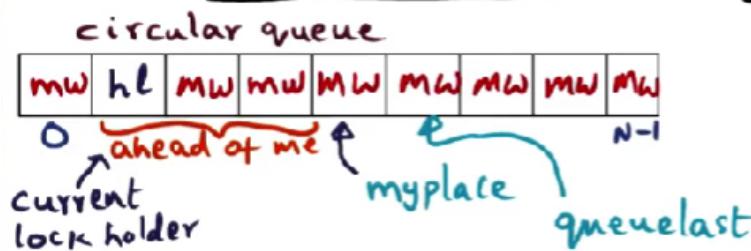
for that lock, and of course queuelast will advance to the next open spot for future requesters that come after me. **Now the important point that I want you to notice is that since the array size is N and the number of processes is N, nobody will be denied. Everybody can come and queue up waiting for this lock.** Because since there are N processes at most N simultaneous requests can be there for the lock and everybody will get their unique spot to wait for if in fact, the lock is currently in use. Given the timing of my lock request and the position of the current lock holder, you can see that I have some waiting to do, because there are quite a few requests that are ahead of me, and so I have some waiting to do before I get my turn in acquiring this particular lock. So now I can tell you what the lock algorithm is going to look like, pretty simple. When I make a lock request what I'm going to do is mark my place in this flags array and the way I do that is by calling fetch and increment on the queuelast variable. And that ensures that I get my unique spot due to the fetch operation and I increment the queuelast to point to the next spot which is available to the next spot for future requesters. And since fetch-and-increment is an atomic operation, remember that we have read modify write operations, fetch-and-increment is one of those. And it's an atomic operation and therefore, even though it's a multiprocessor there could be multiple guys trying to get the same block at the same time. They're all good to be sequenced through this fetch-and-increment atomic operation, and so there is no issue of any risk condition in that sense. So, I will get my spot and I'll increment queuelast. And, of course, if the architecture does not support this fancy fetch and increment read modify write operation, then, you know, you have to simulate that operation using, using test and increment instructions. So once I've marked my position in this flags array, then I'm going to basically wait for my turn. So what I do in order to wait is I'm basically waiting for this spot that I've marked myself, it is right now must wait, it has to change to hl. Once it changes to hl, I know I have the lock, and therefore I'm going to do a spin on this particular location. and I'm going to wait for this location to change its value from mw to hl, so that's the spin loop that you see here. So basically once I have marked my position, I'm going to wait on my position becoming hl to know that I have acquired the lock. And, I will get it eventually, because that's the way this algorithm is supposed to work.

15. Array-Based Queueing Lock (cont)



So let's see what happens when the current lock-holder comes around to unlocking the lock. What he's going to do is, he's going to execute the unlock algorithm. And the unlock algorithm, the first thing that it does, is it sets this position that the lockholder had from HL to MW. And the reason for that is, is that this is a circular queue and since it's a circular queue even though queuelast is here future requesters can come around and then eventually somebody may come here and may want to occupy this particular slot and they have to know that they have to wait. And that's the reason, the first thing that the current lock holder does, is to mark this spot that he used to be hl, as mw.

Array-based queueing LOCK



For each lock L:

- flags[N] // queue
- * two states
 - has-lock (hl)
 - must-wait (mw)
- queueLast := 0 // init

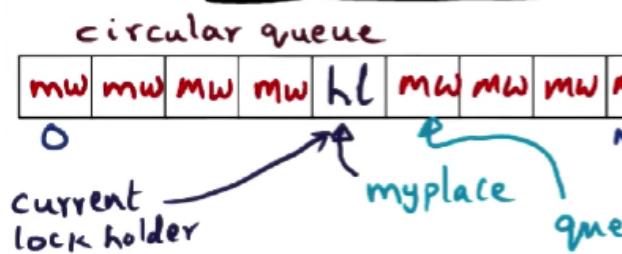
LOCK(L):
myplace = fetch-and-inc(queueLast);
while (flags[myplace mod N] == mw)

unlock(L):
flags[current mod N] = mw;
flags[current + 1 mod N] = hl;



The next thing that the current lock holder is, is going to do is signal the next guy in the circular queue. So, the current lock holder was here, so you'd mark it as mw for future requesters that may come and wait on his spot. And the next request in the circular queue is the guy next to him. And therefore what he is doing is, he is saying you know, current plus one mode N, is going to be set to hl. And so, that guy would have been waiting in this position and so he'll get the signal. And therefore he will be getting ready to go. And he can get into the critical section and do whatever he wants to do with the data structure that is protected by this particular lock.

Array-based queuing LOCK



for each lock L:

- flags[N] // queue
- + two states
 - has-lock(hl)
 - must-wait(mw)
- queuelast := 0 // init

LOCK(L):
 $\text{myplace} = \text{fetch-and-inc}(\text{queuelast})$,
 $\text{while } (\text{flags}[\text{myplace} \bmod N] == \text{mw})$

unlock(L):

$\text{flags}[\text{current} \bmod N] = \text{mw}$;
 $\text{flags}[\text{current} + 1 \bmod N] = \text{hl}$

} got it!

Now this will go on, and eventually, my predecessor will become the current lock holder. And when my predecessor is done using the lock, he'll come around to do an unlock and when the current lock holder who's my predecessor does the unlock operation, that's going to be resulting in a signal for me, because basically. He's going to set the flags array, the next spot in the flags array, as hl. And that's the spot I'm waiting on. So good news for me. I've got my position marked as hl, and what that means is that now I've got the lock. And now I can go off into the critical section do what I need to do in order to do the code that is associated with the critical section protected by, this particular lockout.

Now that we understand that the lock and the unlock algorithm works with this array-based queuing, let's talk about some of the virtues of this algorithm.

The first thing that you notice is that there is **exactly one atomic operation that you have to carry out**, put critical sections so, every time you want to acquire a lock you come in and do a fetch and increment and that is all that you do in order to get the lock. And so there's one atomic operation that you do per critical section, that's good news.

And the other thing that you also notice is that the processes are all sequenced in other words **there is fairness**, so whoever comes first. Gets into the queue ahead of me and when I come in if people are going to come after me they're going to get queued up after me. So that's good news also.

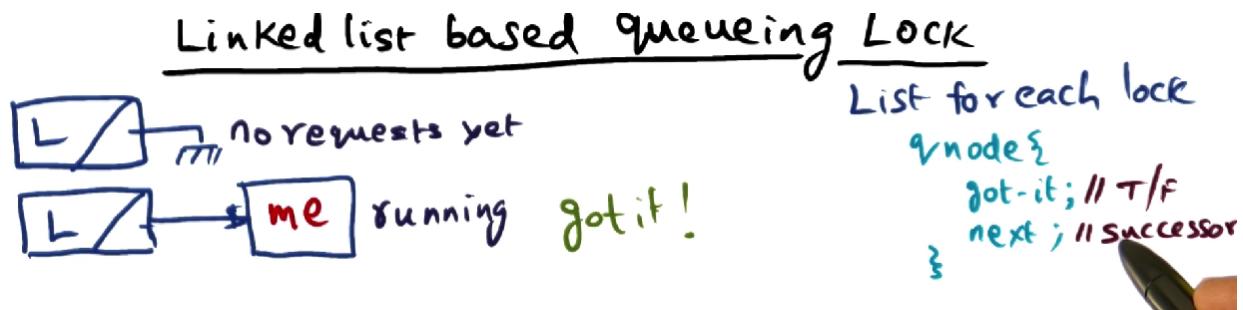
And the spin variable we're going to mark my position in this array my spin variable is distinct from the spin variable of all the other guys that may be waiting for the same lock. That's another

good thing. In other words, I'm completely unaffected by all the signaling that it will happen when the guys that are ahead of me were getting the lock and, and signaling the next guy and so on. I'm completely impervious to that because I'm spinning on my own private variable. Waiting for the lock.

And of course, correlating to what I just said is that whenever a lock is erased, exactly one guy is signaled to indicate that they've got the lock. And, and that's another important virtue of this particular algorithm. So, it is fair. And it is also not nice, so these are two things that very good things about this algorithm. And those we saw were you know the deficiency of the ticket lock algorithm was exactly that where it is fair, but it is noisy when the lock is released. So that problem has gotten away with this queuing lock.

Now you might be wondering, **is there any downside to this array-based queuing lock?** I assure you there is. The first thing I'm sure that you've noticed already is the size of the data structure is as big as the number of processors in the multiprocessor. **So the space complexity for this algorithm is order of N for every lock that you have in the multiprogram.** So if you have a large-scale multiprocessor with dozens of processors, that can start eating into the memory space. So that's something that you have to watch out for. **So the space can be a big overhead.** And the reason I'm emphasizing that is that in any well-structured multi-threaded program even though we may have lots of threads executing in all the processors. At any point in time for a particular lock, they might not be in contention but all the processors, only a subset of them may be requesting the lock. But still, **this particular algorithm has to worry about the worst case contention for a lock**, and therefore it creates a data structure that is as big as a number of processes that you have in the multiprocessors. And that's the only downside to this, but all the other things are good stuff about this algorithm. And of course, the reason why you have that downside with this particular Anderson's queueing lock is the fact that the queue is being simulated by a static data structure, an array. And since it is a static data structure and you have to worry about the worst-case contention among requesters for a lock we have to make this static array as big as the number of processors. So that's really the catch in this particular algorithm. Next, we will look at another algorithm, a lock algorithm that is also based on queuing, but it doesn't have the space complexity of Anderson's queuing lock.

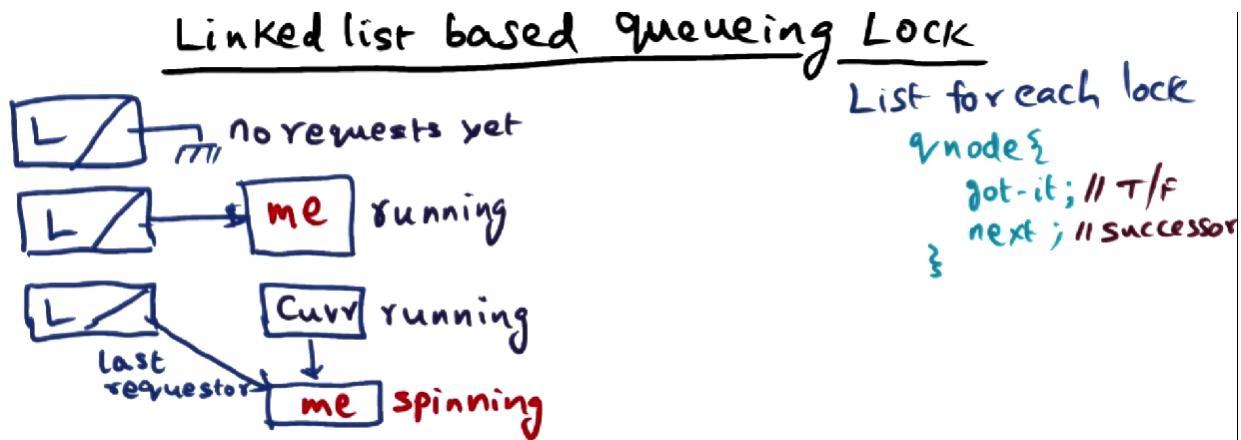
16. Link Based Queueing Lock



So to avoid the space complexity in the Anderson's array based queueing lock, we're going to use a linked list representation for the queue. **So the size of the queue is going to be exactly**

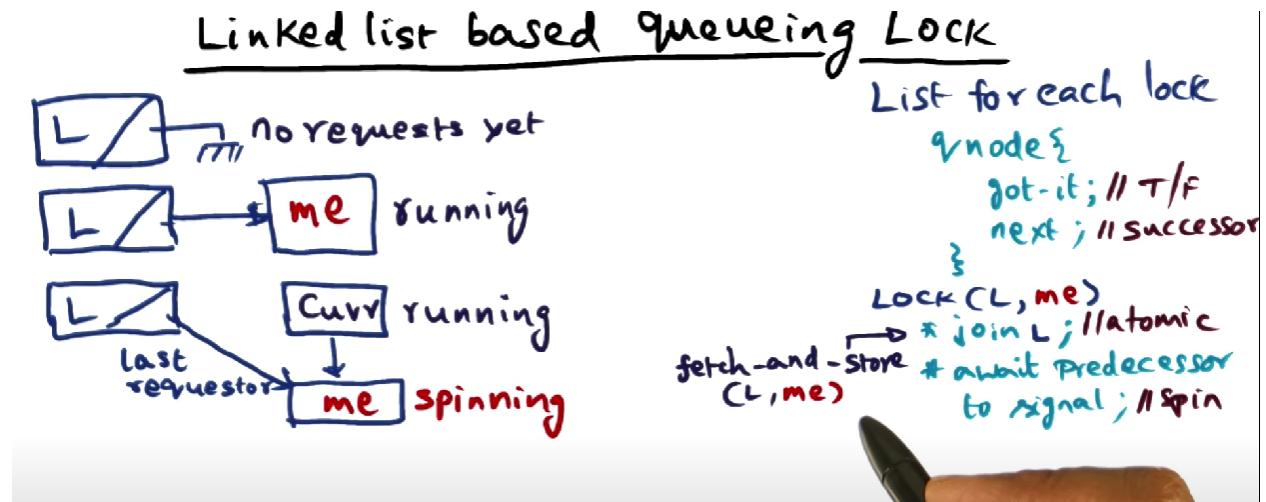
equal to the dynamic sharing of the lock. And this particular linked list based queueing lock algorithm is due to the authors of the paper that I've prescribed for you in the reading list. Namely Mellor-Crummey and Scott. And so sometimes this particular queueing lock is also referred to as the **MCS lock**. So the lock data structure. The head of the queue is a dummy node. It is associated with every lock so every lock is going to have this dummy node associated with it and will initialize this dummy node to indicate there is no lock requesters presently for this particular lock. So, this pointer is pointing to nil. Nobody's got the lock. And there are two fields for every q node for a requester. So every new requester is going to get this q node. And in this q node there are two fields. One field is the got-it field. And got-it is basically a boolean that says whether I have the lock or not. If it is true, I've got it. If I don't have, if it is false I don't have it yet. And the next field in the queue note is pointing to my successor in the queue. So if I came in and I requested the log, I get into the queue. And if a, if a successor comes along and requests a log, he gets queued up behind me. So that's this basic data structure, every queue note is associated with a requester. The dummy node that we start with is representing the lock itself. And since we are implementing a queue, fairness is automatically assured. The requesters get queued up in the order in which they make the request, and so we have fairness built into this algorithm, just like the Anderson's array-based queue lock. The lock to, to nil indicating there are no requests yet. And let's say that I come along and request a lock. I don't have to wait because currently, there's nobody in the queue and therefore I get the lock right away. And, and I can go off into the critical section and start executing the critical section code, that is associated with this particular lock. So what I would have done, when I came in to make this lock request, is to get this q node. And make the lock data structure point to me. And I'd also set the next pointer to null, to indicate there's nobody after me. And once I've done that, I know that I've got the lock. And I can go off in the critical section, and do whatever I need to do.

17. Link Based Queueing Lock (cont)



I was lucky this time that there was nobody in the queue when I first came and requested the lock. But another time, I may not be that lucky. There may be somebody else using the lock already, and if that is the case, then what I would have to do is to queue myself in this data structure. And the way to do that is to indicate by setting the last pointer, in this list to point to me. This pointer is always pointing to the last requestor. In this case, the original case that I showed you, I was the only requestor that was also the last requestor. But now, the queue has somebody using that particular lock, and so when I come in, what I'm going to do is, I'm going to set this field of the lock data structure, the dummy load, the head node, of the lock data structure to point to me and the last requester. And I'm also going to fix up the link list so that the current guy is going to point to me. Why am I doing this? Well, the reason I do this is that when he is done using the lock, he needs to reach out and signal me. What am I going to be doing? I'm going to be spinning. And what am I spinning on? **I'm spinning on the got-it flag**. So this is a data structure that is associated with me, and one of the fields, you know, is the got-it field in the data structure. So I'm going to spin on this got-it field in the data structure, waiting for this guy to set it to two. So, I initialized it to false when I came in, and form this request. When I form this request, what I did was to set myself as the last requester, I'll clear out this field to indicate that I don't have the lock, and I'll set up the link list so that the current lock holder points to me through his next field. And my next field, of course, is null because there is no requester after me. So once I fixed up this, link list and in this fashion, then I basically can spend on my got it a boolean variable.

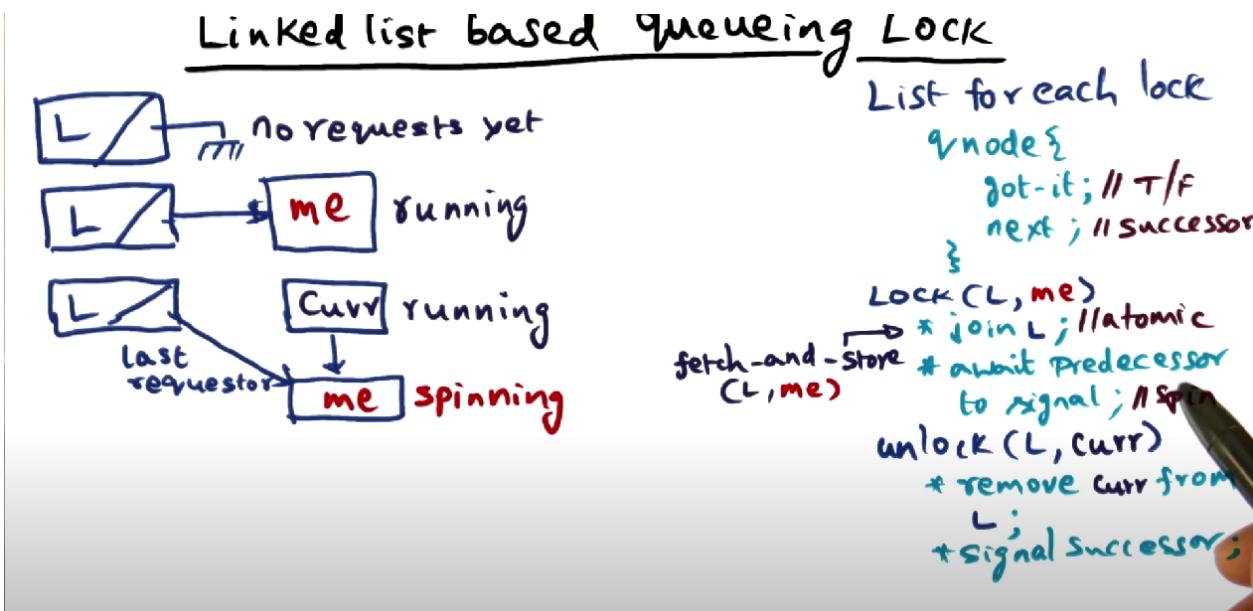
18. Link Based Queueing Lock (cont)



So now we can describe to you the lock algorithm. **Basically, the lock algorithm takes two arguments. One is this name dummy node that is associated with this particular lock. And it's also taking my queue node**, the one that I am providing, to say that this is my queue node, please queue me into this lock request queue. And when I make this call it could be that I'm in this happy state, in which case, I don't have any lock requesters ahead of me. But if it turns out that, when I come in there is somebody is using this lock, then I'm going to join this queue. And has to be done atomically. There are two things going on here in joining this queue

atomically. What I do is, I set the last pointer. This list is always pointing to the last requester. So, it used to point to this guy, he was the only requester. I came along, so we had to fix up this list so that this, and pointer is going to point to me, the last requester. And I also had to fix up, the current requestor point to me. And once I have done that, then I can await the predecessor, namely this guy, to signal me, by spinning on the got-it variable that is associated with my data structure. And the other thing that I would do as part of joining this queue is to set my next point at null, because there is nobody after me, I just made the lock call. Notice that when I'm joining this queue, I'm doing two things simultaneously. One is, I'm taking the pointer that was pointing to him and making it point to me. And I also need the coordinates of the previous guy so that I can set his next pointer to point to me. So I have to do this double act. So this has to be done atomically as well. So joining the queue, essentially, is a double act of breaking a link that used to exist here, make it point to me, and get the coordinates of this guy, so that I can fix him up. And remember that this is happening simultaneously. Perhaps with other guys trying to do the same thing, joining this queue. And therefore, this operation of breaking the queue and getting the coordinate of my predecessor has to be done atomically. And in order to facilitate that, we will propose having a primitive operation called **fetch and store**, **an atomic operation, and the semantics of this fetch and store operation is that when you make this call and give it two arguments, L and Me. What this fetch and store are going to do is, it's going to return to me what used to be contained in L**, so what used to be contained in L is my predecessor. So I'll get that, and I'll get the coordinates of this guy. **And at the same time, it's also storing into L a new node that is the pointer to the new node that is me.** And so that is what is being accomplished by this. The double act that I mentioned of getting my predecessors coordinates and setting this guy to point to me is accomplished using this fetch-and-store operation. It's an atomic operation. **And clearly, the architecture is not having this fetch and store instruction you have to simulate that with a test and set instruction.**

19. Link Based Queueing Lock (cont)



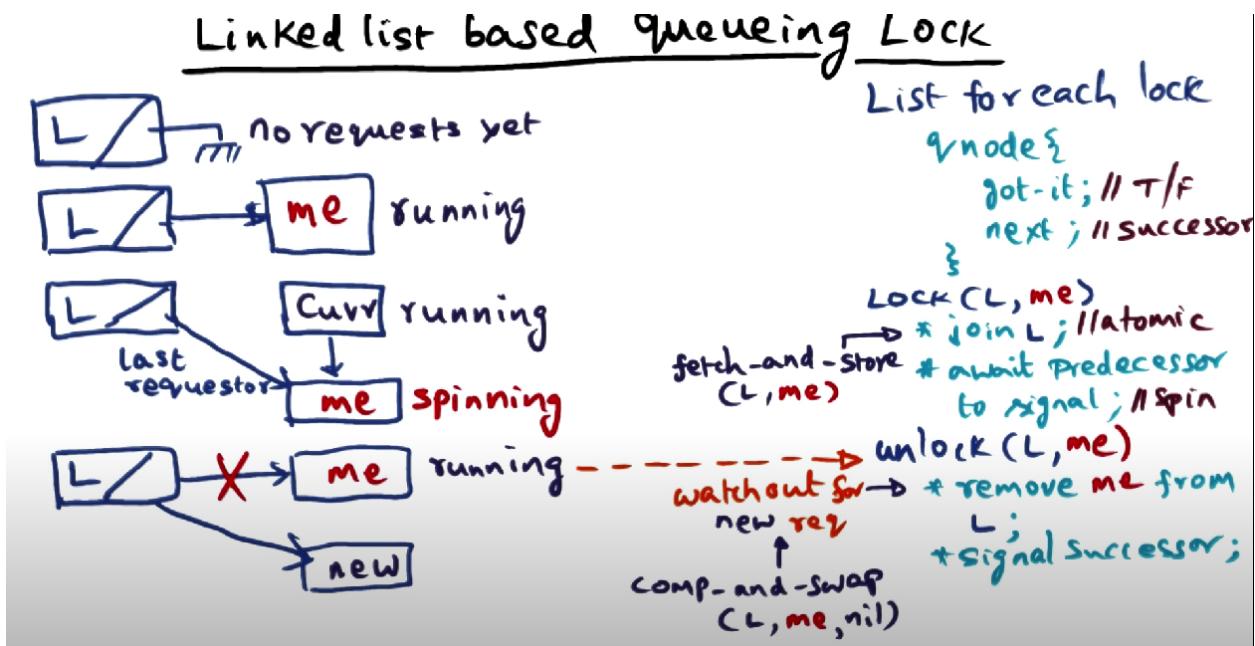
So once I've done the double act, then I can set up the current node's next pointer to point to me. And then I'll be done with joining the cube and then I can await the predecessor to signal me. So, I'm spinning on the got-it variable. And how will I know that I've got the lock? Well, my predecessor who is currently using the lock will eventually come around and call this **unlocked function**. And the unlocked function is basically taking, again, **two arguments**. **One argument being the name of the lock, and, and the other argument is the guy that's making the unlock call**, in this case, the current node that's making the unlock call. And what it does is to remove current from. On the list and it is going to signal the successor. And the way the successor is going to be signalled is because the current node has an x pointer and the x pointer says he's the next guy waiting in line for getting this particular lock. And he's pinning on the got it variable. So he's just going to signal the successor. By setting the guarded variable for the successor to be true, and that will get me out of my spin loop, and I'll have the lock. And I'm now running inside the critical section having obtained the lock that's protecting the data structure associated with that critical section.

Linked list based queuing LOCK



So now I'm in my critical section. And eventually I'll get done with my critical section. When I get done with my critical section I have to unlock and I call the unlock function. Normally the unlock function involves me removing myself from this link list and then signaling the successor. So these are the two things I have to do. Remove myself from the list, and signal any successor. The special case occurs. When there is no successor to me. The special case when that occurs what I have to do is I have to set the headnode, the dummy node, of the link, link list, namely L to null to indicate that there is no request... Waiting for this lock. So that's a special case. And so if I look at this picture here, what I have to do is I have to set this L to null, and then I'll be done. I don't have a successor signal. But wait, there could be a new request that is forming. And if a new request is forming, now this guy what you would have done is To do a fetch and store. And, and if you did a fetch and store on this linked list, what would have happened is that he would've gotten my coordinates, and you'd have set the list to point to him. So the new request is forming, but it will not form completely yet. In other words, the next pointer in me is not pointing to this new request yet. **And this is the classic race condition that can occur in parallel programs, and in this particular case, the race condition is between the unlocker, that is me, and the new requester that is coming to put himself on the queue.** And such race conditions are the bane of parallel programs. And one has to be very, very watchful for such ? conditions. And being an operating system designer, you have to be ultra careful to ensure that your synchronization algorithm is implemented correctly. You don't want to give the user the experience of the blue screen of death. You have to think through any corner case that can happen In this kind of scenario and design the software in such a way, operating system in particular, to make sure that all sets of these conditions are completely avoided. Now, let's return to this particular case and see how we can take care of this situation.

20. Link Based Queueing Lock (cont)

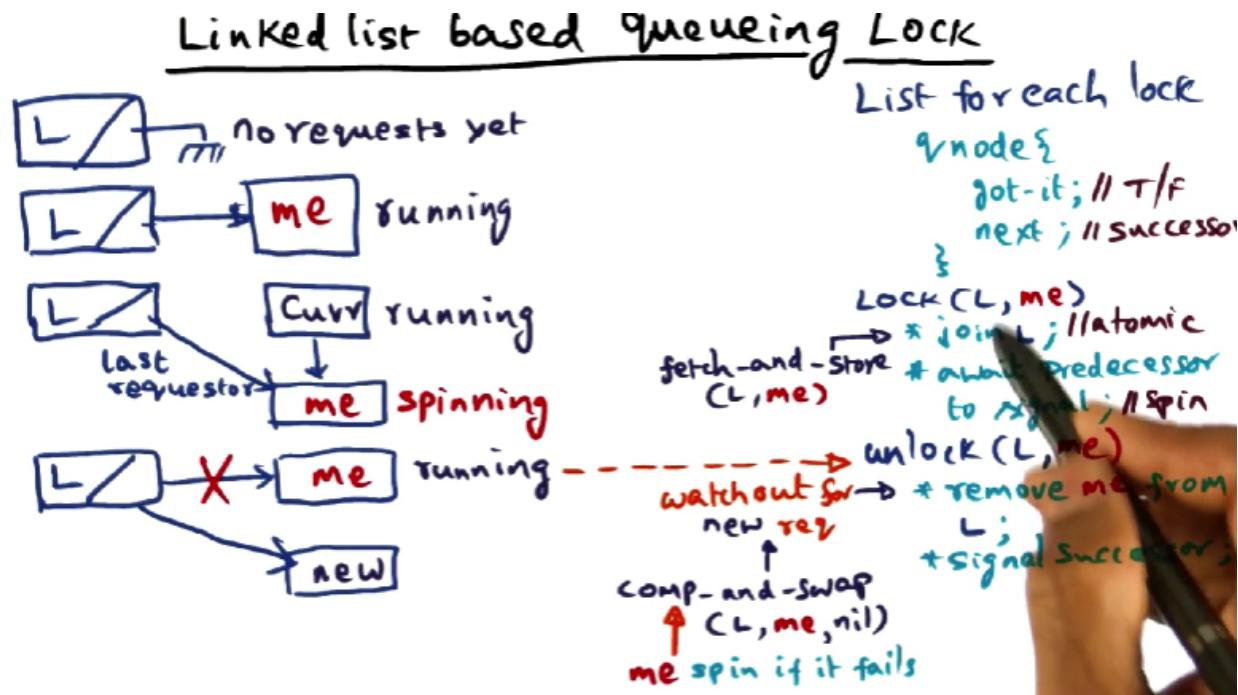


So if there was a new request that is forming, you know that the new request would have called the lock algorithm. And if you call this lock algorithm, and it actually executed this fetch and store operation, then you know that this link is no longer going to be pointing to me. But is going to be pointing to him, right? And that's what this fetch and store would have done. It is to give this new guy my coordinates, and it'll also set the linked list to point to him as the last requester. So that would have been accomplished through this fetch-and-store.

So what I have to do, when I come in and try to unlock, that is, removing me from the queue. Even though my next pointer is nil, I cannot trust it entirely because it could be a successor that is forming, it's just that it's not that the formation of the list is not complete yet. So what should I do? Well, remember when I told you if I was the only guy, what I wanted to do was to set this guy to nil to indicate that there's no requesters after me. the, the list is empty. But before I do that, I have to double check if there is a request that is in the information. And, in other words, I want to have an atomic way of setting this guy to nil if in fact he's pointing to me. And the invariant in this case, is that. **If he's pointing to me, I can set him to nil. If he's not pointing to me, I cannot set him to nil** because he's pointing to somebody else. That's the invariant that I should be looking for, so I need an atomic way of checking for the that invariant. **And the invariant is in the form of a conditional store operation.** The conditional store being. Do this store only if some condition is satisfied. Now in this particular case, I'm going to tell you a primitive that will be useful for this purpose. **And that primitive is what is called compare and swap. It takes three arguments.** The first two arguments is saying, here is L and this is me. Check if these two are the same. If these two are the same, then you set L to the third argument. The third argument is what L has to be set to if these two are the same. That's where it's called compare and swap. You are comparing the first two arguments, and if the first two arguments happen to be equal, then we are saying set the first argument to be equal to the third argument. So that's the idea behind compare and swap. So, essentially when I execute the

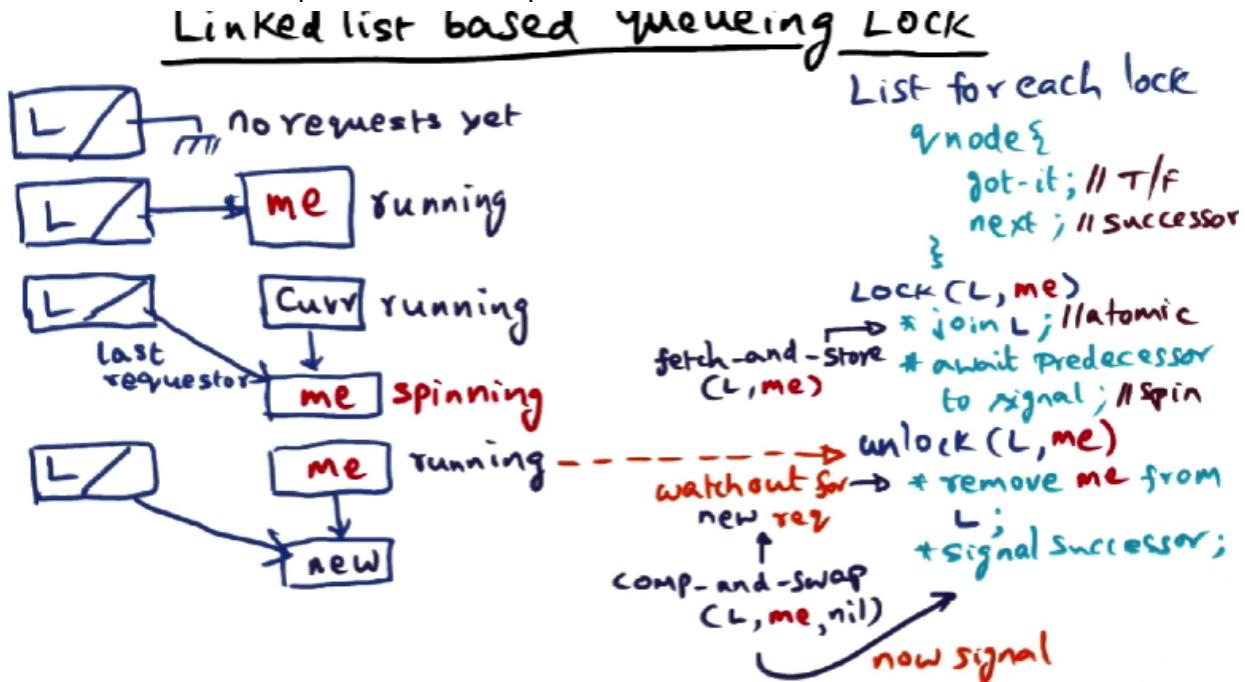
compare and swap operation, on L, me, and nil. What I'm telling is to, to set this guy to nil if he's pointing to me. If he is not pointing to me, don't do that. So that's the idea behind compare and swap.

21. Link Based Queueing Lock (cont)



So this compare and swap instruction is going to return true if it found that L and me, that first two arguments, are the same and therefore it set L to the third argument, in that case, it's a success and success is indicated by a true being returned by the operation. But on the other hand, **if the comparison failed, it won't do the swap. It'll simply return false**. So it won't do the swap, but it'll return false. So that's the semantic of this particular instruction. Again, this is an atomic instruction. And this atomic instruction maybe available in the architecture. But if it isn't, then you have to simulate it using test and set instruction. So in this particular example that I am showing you, when I try to do this unlock operation because this new guy has come in and he's executing, he's halfway through executing his lock algorithm. So he has done the fetch and store and, and he's going to set up the list so that my next pointer will point to him. So that's the process that he's in right now. So at that point, I'm coming in, I'm saying, well, I want to do the unlock operation, and that's when I found that my next pointer is nil. And so what I have to do is, do this compare and swap, and at the compare and swap, now it's going to return to me false, indicating that this particular operation failed. So once I know that this operation has failed, then I'm going to spin. And **so the semantic of the unlock call is, I come in, remove myself from L. And in order to do that, I'm going to do this compare and swap on the linked list. And if I find that the compare and swap instruction fails, I'm going to spin**. Now what am I spinning on? When will it become not nil? **So basically what I'm going to do is I'm going to spin on my next pointer being not nil. So right now it's nil**. That's the reason that I think that there's nobody after me. I was going to set this guy to nil. But I know that compare and swap fail

and therefore I know that there's a request information and I'm going to spin waiting for my next pointer to become not nil. Now when will my next pointer become not nil? Remember that this guy the new guy that is doing this lock operations doing exactly what I did earlier. Right? And, and what he's doing is he's gotten my coordinates and he is in the process of setting it up, so that my next pointer's going to point to him. So, eventually, he'll complete that operation. So my spinning is on this becoming not nil and it'll become not nil because of this new guy completing what he needs to do as part of this, lock operation.



And, so, eventually the next pointer in, in my note will point to him and at that point I can come out of my spin loop. Now, I'm ready to signal the successor that hey, you got the lock. So, that's how I can make sure that when we unlock the corner case that occurs during unlock and that is there is no requesters after me, I can take care of that by doing this atomic and ensuring that there's no race condition between me the unlocker and a new requester that is in the process of forming through this lock call. So once this lock data structure has been fixed up nicely by this new requester, so far as I'm concerned, everything is good. I can, the list is good, and therefore I can go ahead and signal the next guy that he's got the lock and be done with it.

22. Link Based Queueing Lock (cont)

I strongly advise you to look through the paper and understand both the link list version as well as the previous Anderson's array based lock version of the queuing locks. Because there are lots of subtleties in implementing these kinds of algorithms in the kernel and in the parallel operating system kernel. And therefore, it is important that you understand the subtleties by looking at the code. I've given you, of course, a description at a semantic level of what happens, but looking at the code will actually make it very clear what is going on in terms of writing a synchronization algorithm on a multiprocessor. And one of the things that I mentioned is that both the the link list based queuing lock as well as the earlier array based queuing lock required

fancier re-modified write instruction. So for instance, in this case, we need a fetch and store, and in this case and also a compare and swap to fancier re-modified write instruct, instructions. And similarly the array based queuing log required a fetch and increment. Now it is possible that the architecture doesn't have that. If that is the case then you have to simulate these fancier read modify write instructions using a simpler test and sentence structure.

23. Link Based Queueing Lock (cont)

So now let's talk about the virtues of this link list based queuing lock. Some of virtues are exactly similar to the Anderson's queuing lock, and that is it is fair. And so Anderson's lock was also fair, ticket lock was also fair. **The linked list queuing lock is also fair.** And again, the spin location is unique for every spinner, right? **Every spinner has a unique spin location to wait on** and so that is similar to the Anderson's queue lock as well. And that's good because you're not causing contention on the network when the lock is released. When one guy releases the lock, others if they're waiting, they don't, they don't get bothered by by the, by the signal. And **exactly one processor gets signaled when the lock is released.** That's also good. And usually, there's only one atomic operation per critical section. And the only thing that happens is this corner case. In order to implement this **corner case**, you have to **use a second atomic operation.** But if the link list has several members in this, in these examples. I'm just showing only two requesters at a time. But if the link list has a number of requesters, then if I am middle of the gang, have, using the lock, I simply signal the successor. I don't have to do anything fancy in terms of compare and swap. So this is something that needs to be done only for the corner case, not as a, a routine for doing the unlock operation. And the other good thing that we already mentioned is that **the space complexity of this data structure is proportional to the number of requesters to the lock at any point of time. So it is dynamic.** It's not statically defined as in the array-based queueing lock. And so that's one of the biggest virtues of this particular algorithm that the space complexity is bound by the number of dynamic requests to a particular lock, and not the size of the multi-processor itself.

Now the **downside** to this link list based queuing lock of course is the fact that there is **link list maintenance overhead that is associated with making a lock request or unlock request.** And Anderson's array-based queue lock because it is in a irregular structure can be slightly faster than this, link list based algorithm. And one of the things that I should mention to that is that both **Anderson's array-based queue lock as well as the MCS link list based, queue lock may result in poorer performance** if the architecture does not support fancy instructions like this, because they have to be simulated using test and set, so that can be a little detriment to to this particular algorithm as well.

We have discussed different algorithms for implementing locks in a shared memory multi processor. If the processor has some form of affection free operation, then the two flavors of queue based locks, both due to Anderson and MCS, they are good bet for scalability. If on the other hand, the processor only has test and set, then an exponential backoff algorithm would be a good bet for scalability.

24. Algorithm Grading

Question

Grade the algorithms by filling in this table

Algorithm	Latency (low/med/high)	Contention (low/med/high)	Fair (Y/N)	Spin (pvt/sh)	RMW ops per CS (low/med/high)	Space ovhd (low/med/high)	Signal only one on lock release (Y/N)
Spin on T&S							
Spin on read							
Spin w/delay							
Ticket lock							
Anderson							
MCS							



Latency: time spent by a thread to acquire a lock

waiting time: how long do I wait to obtain a busy lock

Contention: when a lock is freed, how long does it take in the presence of contention for a winner to emerge with a lock?

Fair: First come first serve

Spin: whether the spin is on private variable or shared variable

RMW ops per CS: how many RMW are required for a lock. This really depends on the amount of contention except for Anderson and MCS, which has two fixed numbers

Question

Grade the algorithms by filling in this table

Solution

Algorithm	Latency (low/med/high)	Contention (low/med/high)	Fair (Y/N)	Spin (pvt/sh)	RMW ops per CS (low/med/high)	Space ovhd (low/med/high)	Signal only one on lock release (Y/N)
Spin on T&S	low	high	N	S	high*	low	N
Spin on read	low	med	N	S	med*	low	N
Spin w/delay ✓	low++	low+	N	S	low+	low	N
Ticket lock	low	low++	Y	S	low++	low+	N
Anderson ✓	low+	low	Y	P	1	high	Y
MCS ✓	low+	low	Y	P	1(max 2)	med	Y

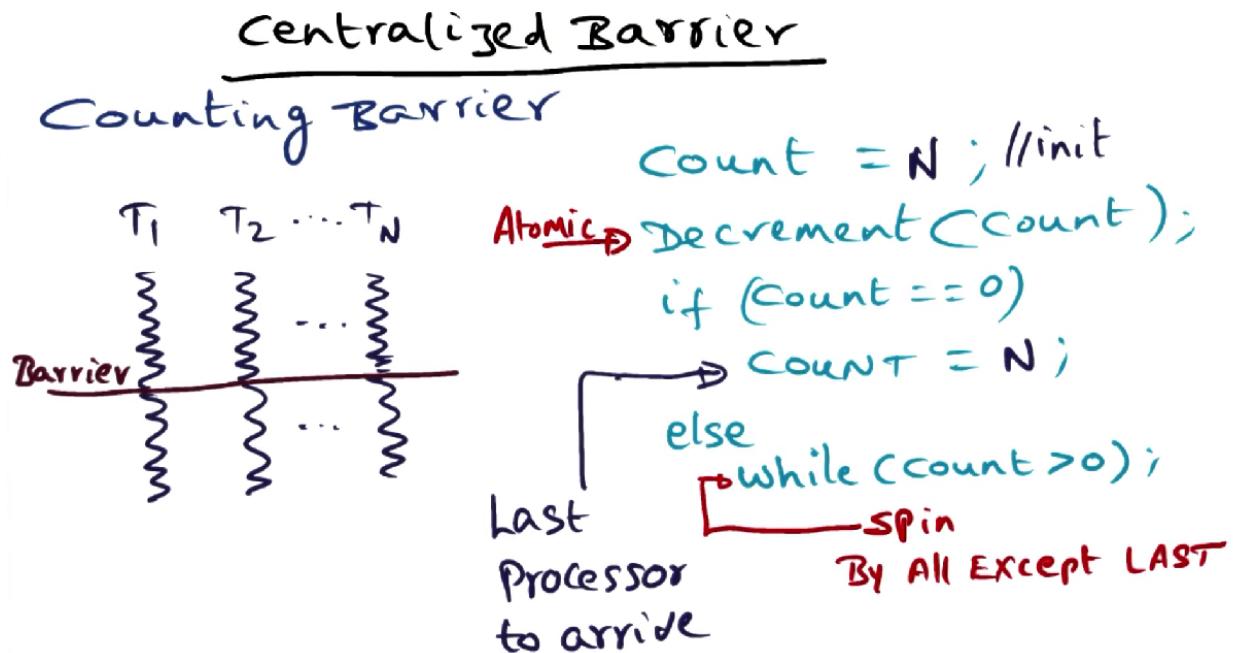
* Proportional to contention for lock

So I'm going to give you the solution for this particular question by filling out this table. And as I said, take your time thinking about it. And, and verifying your own intuition against what I'm presenting to you here. Now what you'll find is that MCS Link-based queue lock and Anderson's array-based queue lock are the two things, two algorithms that done, do quite well on most of the different categories of attributes that I have mentioned to you. But I should tell you that if you, if you have fancy instructions. Fancy read, modify, write instructions. Then Anderson's and MCS lock give you the best performance on all these attributes. But on the other hand, if the architecture does not support fancy read modified op operations and it only has testing set operation available, then some sort of a delay base is a in a exponential delay base or. Starting delay based, spin lock algorithm, may turn out to be the best performer. And in fact, when the amount of contention for lock is, fairly low, it's best to use a spin lock with exponential delay, start out a small delay and keep increasing it. On the other hand, if it is a highly contended lock Then it is good to use a Spin Lock that has categorically assigned various spots for every processor. And one of the things that I also want you to notice is that the number of re modify right operations that, you need to do for the different lock algorithms really depends on the amount of contention that is there for the lock in the case of spin algorithms. In the case of Anderson's and MCS the number of Atomic operation is always one, regardless of how much contention there is. And of course, in MCS, this is the quanta keys that you have to worry about during, during unlocking that might result in an extra remodified item operation. But in the case of the Spin algorithms the amount of contention is really dependent on the number of re modified item operations that you have to perform per critical section. Really depends on the, mode of, mode of contention that is there for the lock.

L04c: Communication

1. Barrier Synchronization

In the previous lesson, we looked at the efficient implementation of mutual exclusion lock algorithms. In this lesson, we're going to look at barrier synchronization & how to implement that efficiently in the operating system. And just to refresh your memory about the barrier, the barrier synchronization works like this, you have a bunch of processors and they all need to know where they are with respect to each other. Where they want to reach a barrier. And they want to wait here until everybody has arrived at this barrier. So if T1 arrives at the barrier, it's going to wait until everybody else has come. So one of the guys, maybe a straggler is going to come a little later, and in that case, everybody has to wait until all the threads that are part of this application have arrived at the barrier, then they can move on. And I mentioned to you that this kind of synchronization is very popular in scientific applications and they go through these phases where they execute code for a while, reach a barrier, and then execute code for a while, reach another barrier, execute four codes for a while, reach a barrier and so on. And, and I mentioned also that in real life this happens quite often. When we go to dinner with a bunch of our friends and some of us show up early and others come late. The usher is going to hold us all. "Wait 'til everyone is here. Until then I cannot seat you". So that same sort of this that's happening, with the barrier that all of the threads have to arrive at the barrier, only then they can proceed on. So that's semantic of the Barrier Synchronization. And I'm going to describe to you a very simple implementation of this barrier.



The first algorithm I'm going to describe to you is what is called a centralized barrier or also sometimes called a counting barrier. So **centralized barrier/counting barrier**, that's a name that, that's given to this. The idea is very simple. You have a counter, that's why it's called a counting barrier. You have a counter. And the counter is initialized to N, where N is the number of threads that need to synchronize at the barrier. And what is going to happen is that, when a thread arrives at the barrier, it's going to atomically decrement the count. A key thing is it has to be done atomically. So once is it atomically decremented and the count then, it's going to wait for the count to become zero. So long as the count is not zero, it's going to wait. So if the count is zero, we're going to do something else, but if the count is not zero that means that, I've arrived at the barrier, but I don't know where the others are yet. So I'm going to wait. So they're going to spin and the spin is saying while the count is greater than zero, spin. And all the processors except the last ones are going to be doing this spinning on count becoming zero. Now the last processor, the straggler may be the T2's straggler. And the straggler arrives eventually. And when he arrives, then what he's going to do is he's going to decrement also. And when he decrements the count, he'll see that the count has become zero. And so what he will do is he'll reset the count back up to N. And that is an indication that everybody, so, all of these guys are waiting on count is greater than zero. So as soon as the count becomes zero, then they can be released from the barrier. And the last processor to arrive is going to reset the count to N to indicate that when these guys go off before they come to the next barrier, the count has to be N. So that's the idea behind that. So very simple algorithm. Decrement the count atomically when you come to the barrier. If the count is greater than zero, then you know that everybody has not arrived, spin. And everybody except the last guy will do the spin. And the last guy that comes around decrements the counter for, and the counter becomes zero. And once the counter becomes zero, all the guys that are stuck here, they're going to be released. **And then the last processor will reset this count to N so that you know all these guys are now on their way to the next barrier.** So, it is resetting it to N so that the barrier can be executed again when all these guys get to the next barrier. And that's the idea behind the centralized barrier.

2. Problems With Algorithm

Question

```
Decrement(count) Last  
if (Count == 0) Processor  
    COUNT = N; ← to arride  
else while (Count > 0); ← All other processors
```

Do you see any problem with this algorithm?

[your Answer]

Now, I'm going to ask you a question. Given this very simple implementation of the barrier decrementing count and count becoming zero resetting it to N by the last processor and all the other guys waiting on the count not being not yet being zero, do you see any problem with this algorithm? And this is an open-ended question. So I want you to think about it and see, could this lead to any raise condition. And, and I mentioned to you when we talked about mutual exclusion algorithm itself that raise conditions are the bane of parallel programming. So, when you're implementing synchronization algorithms you better be absolutely certain that there are no race conditions.

Question

```
Decrement(count) Last  
if (Count == 0) Processor  
    COUNT = N; ← to arride  
else while (Count > 0); ← All other processors
```

Do you see any problem with this algorithm?

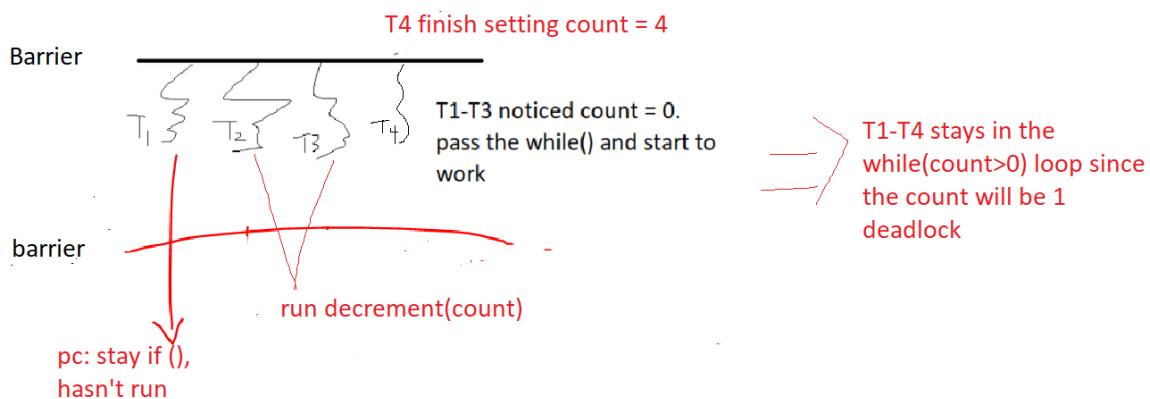
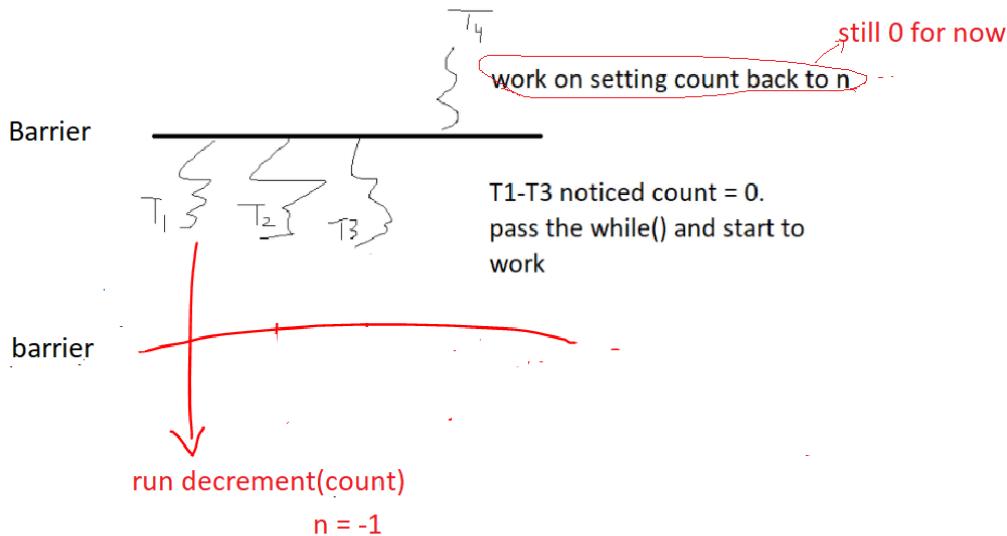
Solution

Before last processor sets count to N, other processors may race to the next barrier and go through!

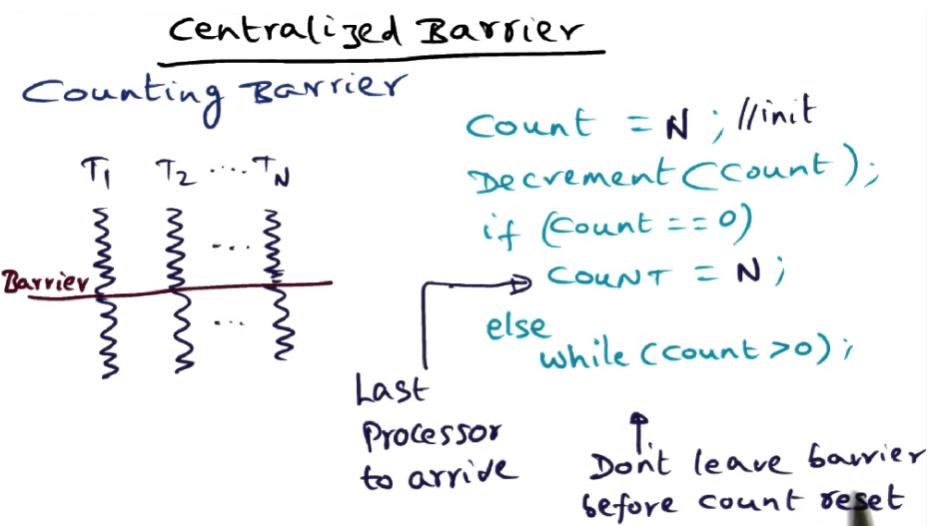
The answer is yes. There is a problem. And the problem is that before the last processor, the last processor guy comes and sets the counter back up to N. And remember what the last processor is doing, decrementing the count. And if the count is zero, as soon as there is a

decrement of the count and the count is bigger than zero the other guys are sitting here. They're going to go off on their merry way, executing code towards the next barrier. And the last processor is, in the meanwhile, fitting the count back up to N. But before the last processor sets the count back up to N, the other processors may race to the next barrier. And they may go through, because they may find that this count has not been set to N, yet. And they will find that the count is zero, and then they'll fall through. And that can be another happy situation. Right?

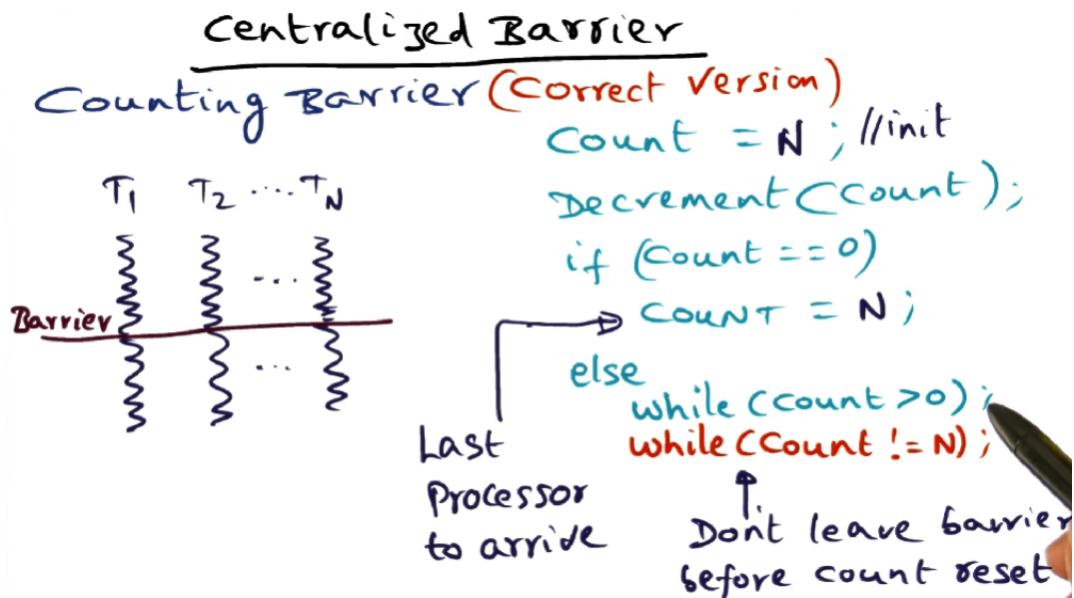
(So basically it is not safe to the case taht we need to re-enter the barrier. It can generate a deadlock problem. Like below)



3. Counting Barrier



So there is a **problem** with the centralized barrier. That is **when the count has become 0 if these guys immediately are allowed to go on executing before the count has been reset to N, then they can all reach the next barrier and then they fall through. And that is a problem.** So the key thing to do to avoid this problem, or to overcome this problem, is to make sure that the threads that are waiting here, don't leave the barrier before the count has been reset to N. Right? So they're all waiting here for the count to become zero, and once the count has become zero they are ready to go, but, we don't want to let them go yet. We want to let them go only after the count has been reset to N.



```

void spin_on_read(){

    // Change the slide code as below.
    // The indent confused me again!
    Decrement(count);

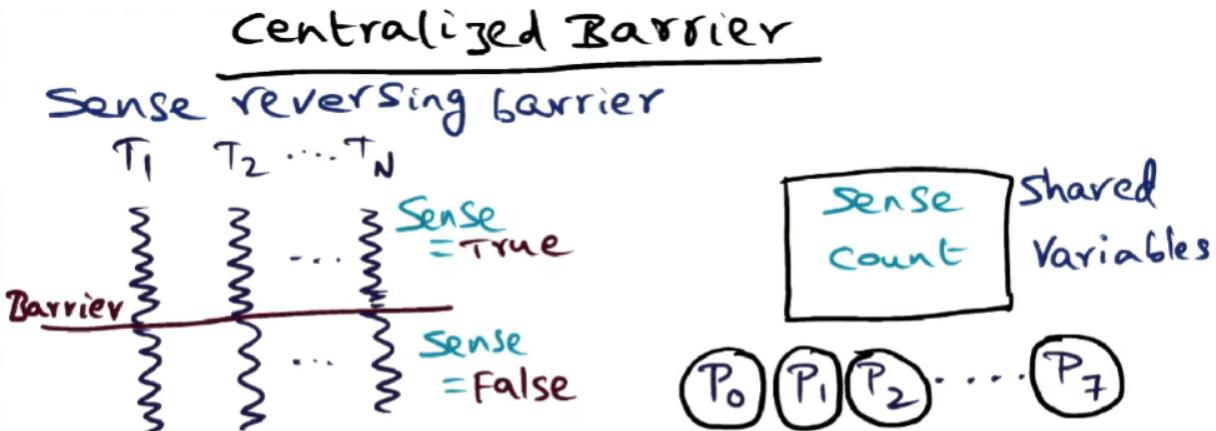
    if(count == 0){
        count = N;
    }
    else{
        while(count>0);
    }

    // So need to check whether the last thread finishes its work
    while(count != N);
}

```

So what we're going to do is, we're going to add another spin loop here. And that is **after they recognize that the count has become 0, they're going to wait till the count is not N yet.**

And so this ordering of these two statements is very important, obviously. So, we want to wait till the count has become 0. At that point, we know that the value is over, but we want to make sure that the counter has been reset to N by the last guy, and once that has been done, then we are ready to go on executing the code that we need to execute til we get to the next barrier.



So we solve the problem with the first version of the centralized barrier, and that is the counting barrier. By having a second spindle. That's the problem, right? There are two spin loops for every barrier in the counting algorithm, and **ideally, we would like to have a single spindle**. And that's the reason that we have this particular algorithm, which is called the **sense reversing barrier**.

If you recall in the counting barrier, we needed two spinning episodes. The first spinning episode was when you arrive at the barrier, decrement the count, and wait for the count to become 0. That's the first spinning episode. And the second spinning episode to leave the barrier, what you need to do was to make sure that the count has become N, right? Those were the two spinning episodes that were there in the counting barrier.

And in the sense reversal barrier, we're going to get rid of one of those spinning episodes. The arrival one, we'll get rid of it. So we don't have to spin on count becoming zero. And we'll see how that is done. So what you notice is that in addition to the count, there is a sense variable, in the shared variables that we have, we included a new variable called **sense variable** that's also shared by all the processes that want to accomplish a barrier synchronization. **The idea behind the sense variable is that the sense variable is going to be true for one barrier episode, and it's going to be false for the next barrier.** So because we at most have one barrier at a time, and therefore, if you call this barrier the true barrier, the next barrier is going to be the false barrier. So that's the way we can identify which barrier we are in at any particular point of time so far as a given thread is concerned about looking at the sense variable.

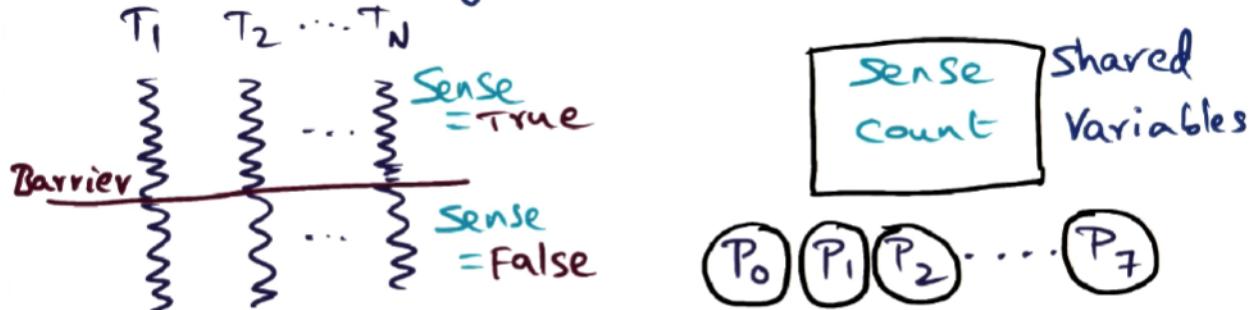
4. Sense Reversing Barrier

To better understand this concept, check -

<http://15418.courses.cs.cmu.edu/spring2013/article/43>

Centralized Barrier

Sense reversing barrier



All except last :

- Decrement count
- Spin on Sense reversal

Last:

- Reset count to N ; reverse sense

So the barrier algorithm is going to work like this. **When a thread arrives at a barrier, what it is going to do is decrements the count exactly like in the counting barrier.** It's going to decrement the count. **But after its decrements the count, what it is going to do is, it's going to spin on Sense reversal.** Remember that, you know the sense flag is going to be True for this barrier and once everybody has progressed to the next barrier, the sense flag will become false. And therefore, let's say that **we are executing the true barrier**.

In other words, all the threads are executing some right here. The sense flag is true, and so if T_1 comes along it decrements the count and it's not going to worry about whether the count has become zero or not. All that it is going to read and wait for the sense to reverse. So it's saying "well my sense is we are on the true value here, I'll stay here until the sense becomes false. I'll know then that, that we've moved on to the next value point."

That's the idea behind what all the processors will do except the last one. What did the last one do? Well, you guessed it. **The last one, in addition to resetting the count to N, which was happening in the counting barrier, was also going to reverse the sense flag.** So, the last processor comes along and finds that the count has become zero, it'll reset it to N. No problem with that. And then it is going to reverse the sense flag. It used to be True here, and it's going to reset it to False. And all the other guys are waiting on the sense reversal. So decrementing the count itself by chaining the count value, doesn't do anything to these threads. Only when the sense flag is reversed, all these guys come out of the spindle and they can go on. So you can see now that **we have only one spinning episode per critical section or one spinning episode per Barrier**. What we're doing is we decrement the count and spin on sense reversal, the last guy decrements the count. When the count goes to zero, resets it to N. And then it is going to reverse the sense. And that is the signal for all the reading processes to say well we

can now go on to the next phase of the computation. So we've gotten rid of one of the spinning episodes that used to be there in the pure counting version of the centralized barrier. One of the centralized barrier is simple and intuitive as to what's going on and of course with the sense reversing barrier we got rid of two spinning episodes and got it down to one. All of these are good things.

But the problem is, that you have a shared variable for all the processors. And so if you have a large scale multi-processor. And if you're running large-scale scientific applications with lots of parallel threads and they have to do a barrier, causes a lot of contention on the interconnection network. Because of this hot spot for this shared variable. And remember what our good friend Chuck Thacker said, less sharing means the multi-processor is more scalable. And that is something that we want to carry forward in thinking about how to get rid of this sharing that is happening among a large number of processes in order to build a more scalable version of various synchronization algorithms.

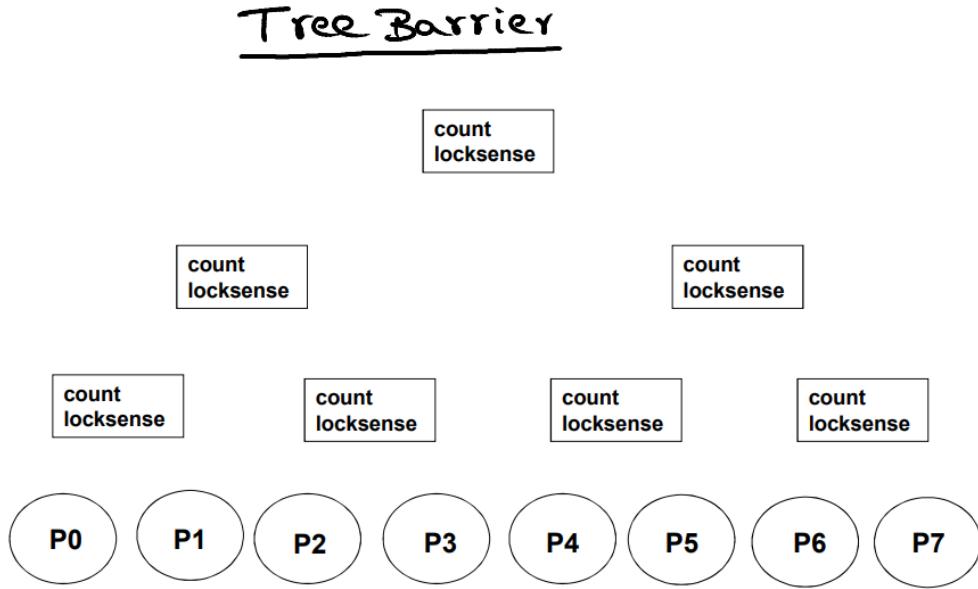
```
struct Bar_t {
    int counter;    // initialize to 0
    int flag;       // initialize to 0
    LOCK lock;
};

int local_sense = 0; // private per processor

// barrier for p processors
void Barrier(Bar_t* b, int p) {
    local_sense = (local_sense == 0) ? 1 : 0;
    lock(b->lock);
    int arrived = ++(b->counter);
    if (b->counter == p) { // last arriver sets flag
        unlock(b->lock);
        b->counter = 0;
        b->flag = local_sense;
    }
    else {
        unlock(b->lock);
        while (b.flag != local_sense); // wait for flag
    }
}
```

Sense Reversal Barrier. Credit: (Nkindberg 2013)

5. Tree Barrier

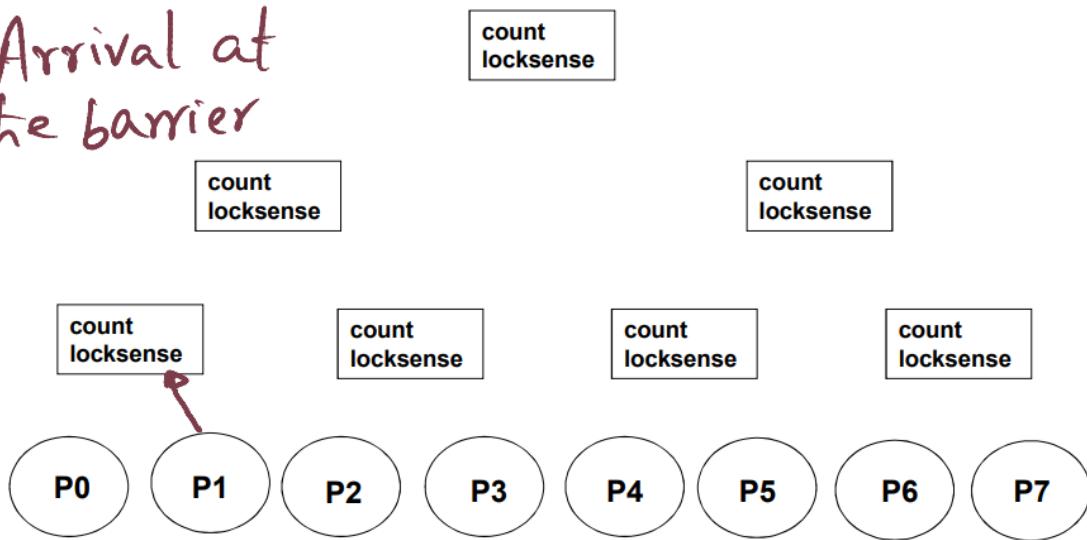


So I'm going to first describe to you a more scalable version of the sense reversal algorithm. And the basic idea is to use **divide and conquer**. I have a hierarchical solution. That is, **limit the amount of sharing to a small number of processes**. Let's say a small number K of processes and in this example, K is equal to 2.

So essentially, you know, what we are saying is, **if you have n processors that the condition, break them up into small groups of k processors**. And so we build the hierarchical solution and the hierarchical solution obviously leads to a tree solution. And so, since we have K processors competing and accomplishing a variable among themselves. If you have N processors, then you have a $\log N$ of the base K as a number of levels in the tree, in order to achieve the value. And in this case, what we have done is K is equal to 2. And so, the number of levels and with the eight processors, The number of levels in the tree is going to be three.

Tree Barrier

Arrival at
the barrier



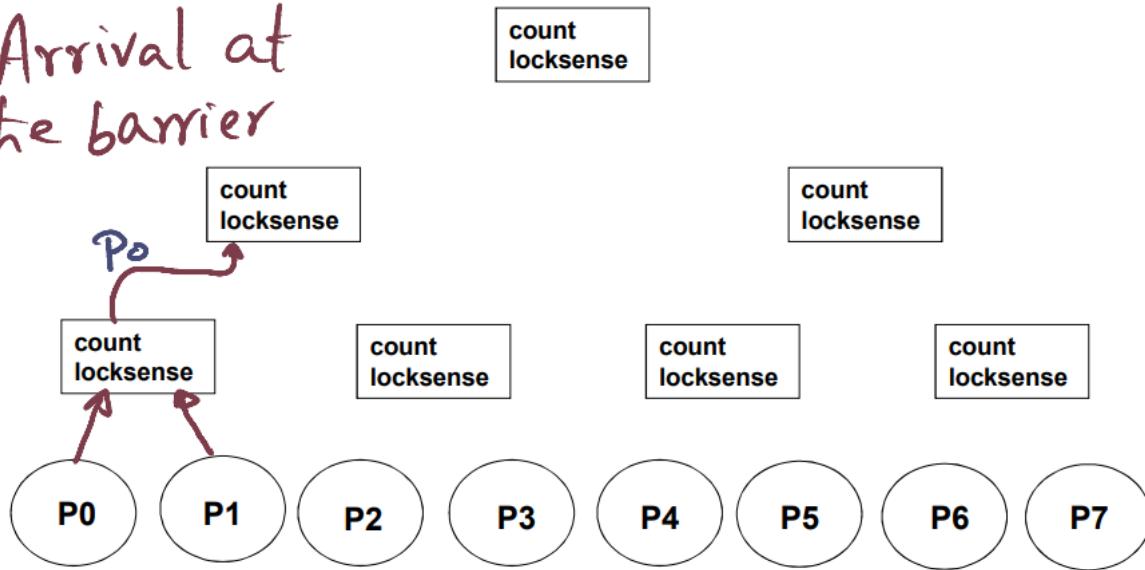
So let's talk about what happens when we arrive at the barrier. So, a micro-level algorithm works exactly like a sense reversing algorithm. And that is, these two processes if they're sharing this data structure at count variable and a locksense variable and you see that for every k processes and in this case k being two, every two processes you have issued two shared variables: a count variable and a locksense variable. Count variable locksense variable, count and locksense.

So what's going to happen and you'll see that you have this count and locksense variable replicated in every level of the tree, and we'll talk about how these going to, variables are going to be used in the progression of this algorithm.

So let's first talk about arriving at a barrier. So let's say that P1 has arrived at the barrier. What it is going to do is, it's going to go and decrement this counter. Now, what is this counter going to be set to? Well, This counter is just for the key processes that value syncing here and keeping two this counter is going to be two. And so, this guy is going to decrement the count, and if the count is not zero it's going to basically wait for the sense to reverse. Just like the sense reversal of algorithms. The same thing is going to happen that P1 comes here decrements the count and it waits for the sense to reverse by spinning on this flag.

Tree Barrier

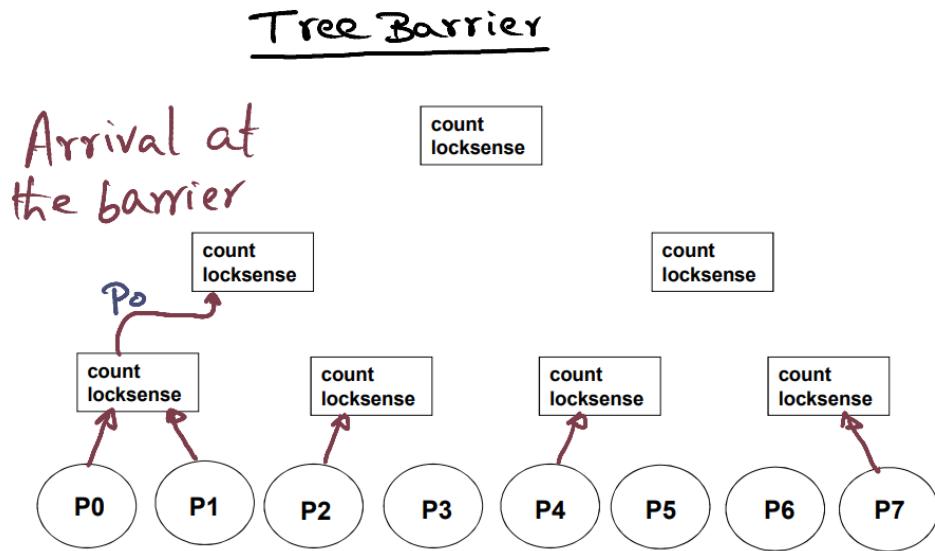
Arrival at
the barrier



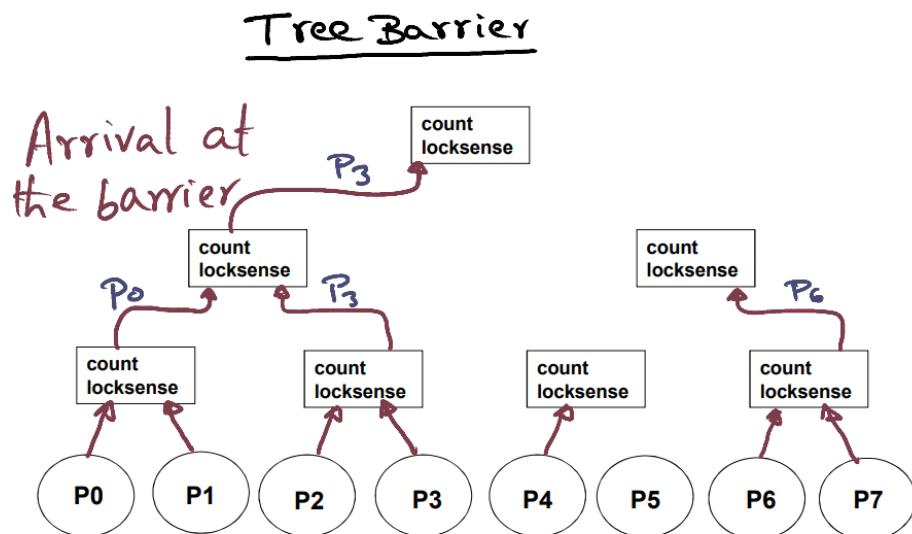
Sometime later, P0 comes to the barrier and it decrements the count, count goes to zero, but you're not done with the barrier yet, because the barrier is for all of the processes. So what P0 is going to say is "okay, between the two of us I know that we both have reached the value because the count is zero. But I have to go up, and go to the next level up and here I'm going to decrement the count here, to indicate that I've arrived at the value".

So P0's the one that arrive up the tree, P1 is stuck here waiting for sense to diverse, P0 moves up. So remember that even though P0's come here decremented the count and made it zero, that doesn't flip the sense flag yet. Right? Because the value will be done only when everybody has arrived, and therefore all that P0 is going to do now is decrement the count, see that it is 0, then it is going to move up in the tree and go to the next level of the tree. And this data structure, which is now shared among this half of the tree this half of the tree is sharing this data structure, so P0 decrements this count. And what'll this count be resized to? Again, 2, right? Because at every level, you have k processors, k being 2 in this case, arriving at a barrier. So P0 arrives here, decrements the count, count is not 0 yet, and so it waits. So P0 is going to wait on locksense to reverse here. P1 is waiting on locksense deliveries here P0 is not waiting on locksense deliveries here because it has arrived at the barrier but his partners are still stragglers, they have not arrived at the barrier yet.

6. Tree Barrier (cont)

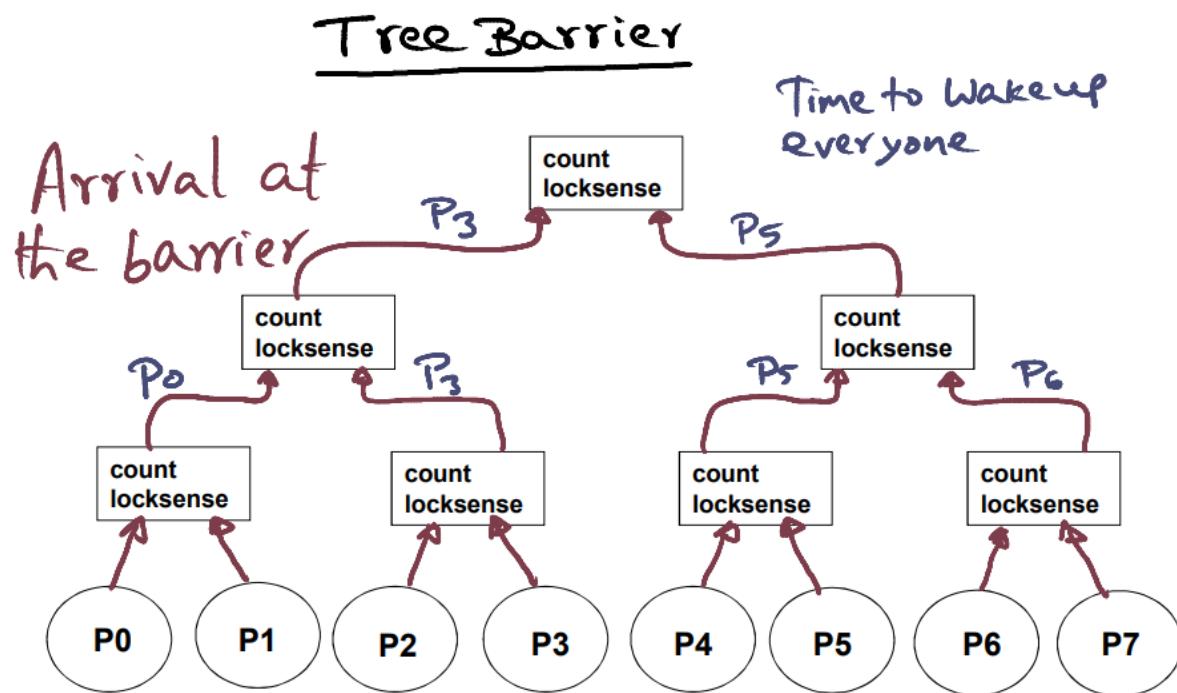


Of course, multiple processors can arrive at the barrier at the same time and all of them are going to work with their local data structure. So, like, this guy will work with this local data structure. This guy with this local data structure. With this local data structure. And each of them is waiting for his partner to arrive so that he can move up the tree. So that's what going on.



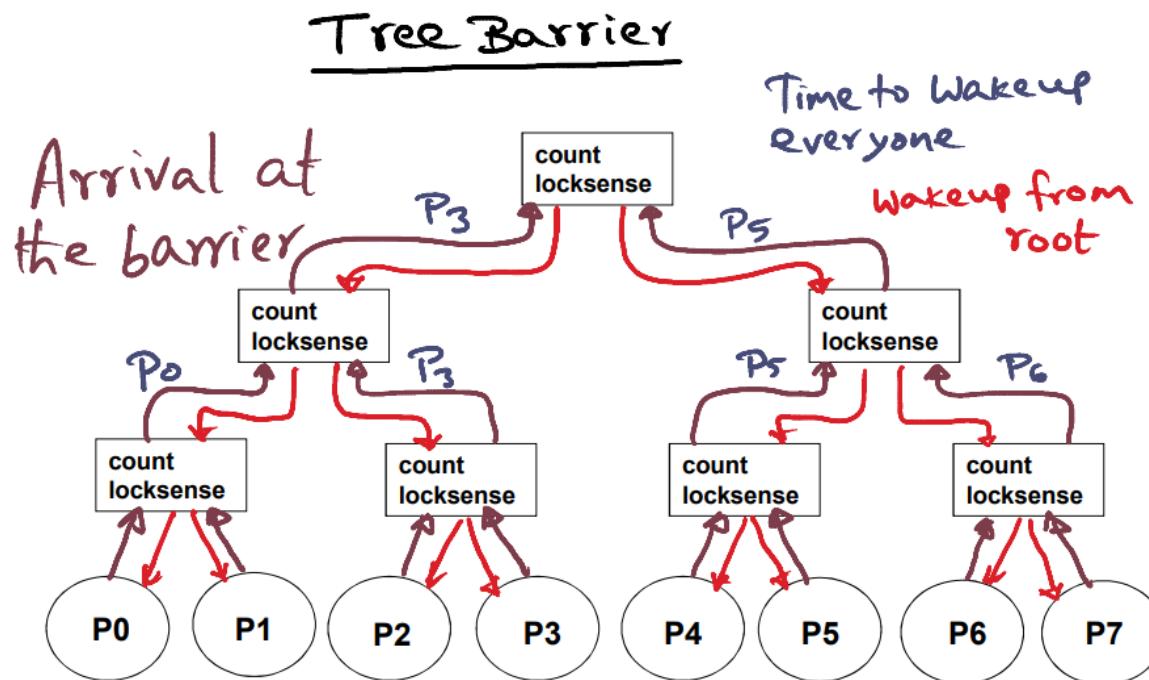
so eventually, P3 is going to arrive and so when P3 arrives, he decrements the count, sees it as zero so he can move up the tree. When it comes here it says, oh the count is already one so I

decrement it and the count becomes zero and remember P0 decremented the count and it is waiting on locksense. So P3, when it comes here, finds that the count is one, decrements it, becomes zero and it moves up the tree because the barrier is still not done until we know that everybody has arrived at the barrier. So in the meanwhile, on this half of the tree, what's going on is that P4 has arrived, P5 is not there yet, P6 and P7 have arrived. And it turns out that P6 was the last guy to come to the barrier here, and therefore, he is the guy that has moved up. And he has decremented count. And he's waiting for this half of the tree to arrive at the barrier. And you can guess which one is going to come up, right? Because P4 has already arrived here, and so if P4 has already arrived here, he's decremented the count, and he's waiting on locksense to flip. So the straggler in this whole seam, scheme of things, is this guy right here. He's the guy who is, is still not arrived, but eventually, he'll also arrive. When he arrives, he will decrement the count, find that the count has become zero, move up the tree, and he'll find that this count is already decremented also, and when he comes up here, he will decrement it to zero, and then he'll say, oh, if we're all done, so we can move up here. So, that's what is going to happen. So we come here, P5 comes here and goes all the way up. And then when it comes up here, it sees that P3 has already decremented the count to one. And so when he comes up, he decrements it, and it becomes zero. And at this point, everybody has arrived at the barrier.



So let's understand what each processor does. When a processor arrives at a barrier it is going to decrement the count. If the count is not zero, it's going to spin on this locksense flag. If a processor arrives at a barrier, decrements the count, finds that the count is zero, then what it's going to do is one of two things. The first thing it's going to do is, he's going to say, "do I have a parent? If I have a parent, what I have to do is, I have to recurse". Do the same thing to the next level. **So the algorithm is, decrement the count and see if the count becomes zero. If the count has become zero, then you recurse. If the end of the parent is there, you recurse. If the count does not become zero, then spin on the local locksense flag. And you continue this.** So you continue this P0, that this came up here and informed this is another parent. So so this, you know, it, it is, it is, it is stuck here. But P3, when it came, later on, it moves up. And when it came up here, this is the last part. So there's no more recursing here. So when P5 finally arrives here, it finds that there is no more parent. This is the root of the tree. And since we reached the root of the tree, you know that if the count is zero now at the root of the tree, then everybody has arrived at the barrier. So count at the root of the tree becoming zero is indicative to the last arriving processor, P5 in this case, that everybody has arrived at the barrier, so it's time now to wake up everyone.

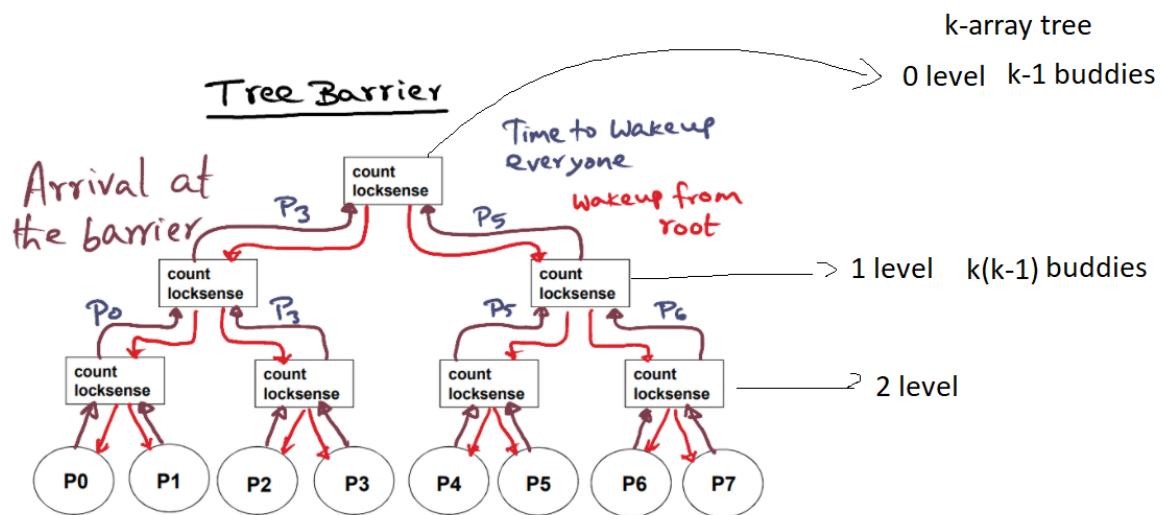
7. Tree Barrier (cont)



So the last processor to arrive at the root of the tree, in this case, P5. He's the guy who is going to start the waking up process for everyone, and the way the wake-up process works is that P5

having realized that he has reached the root of the tree and having realized that he's the last one to arrive because the count is already zero after you decremented it, he's going to flip this locksense flag.

So, when he flips this locksense flag, what's going to happen? Two things, one is this guy, P3, he's waiting on this locksense flipping. So he's going to be released from the spin he's on. Of course, P5 has reset the count back up to n to prepare for the next barrier and it has flipped the locksense. So freeing up P3 and it is now ready to go down the tree as well to tell his buddies that the barrier is done and wake up everyone along the way.

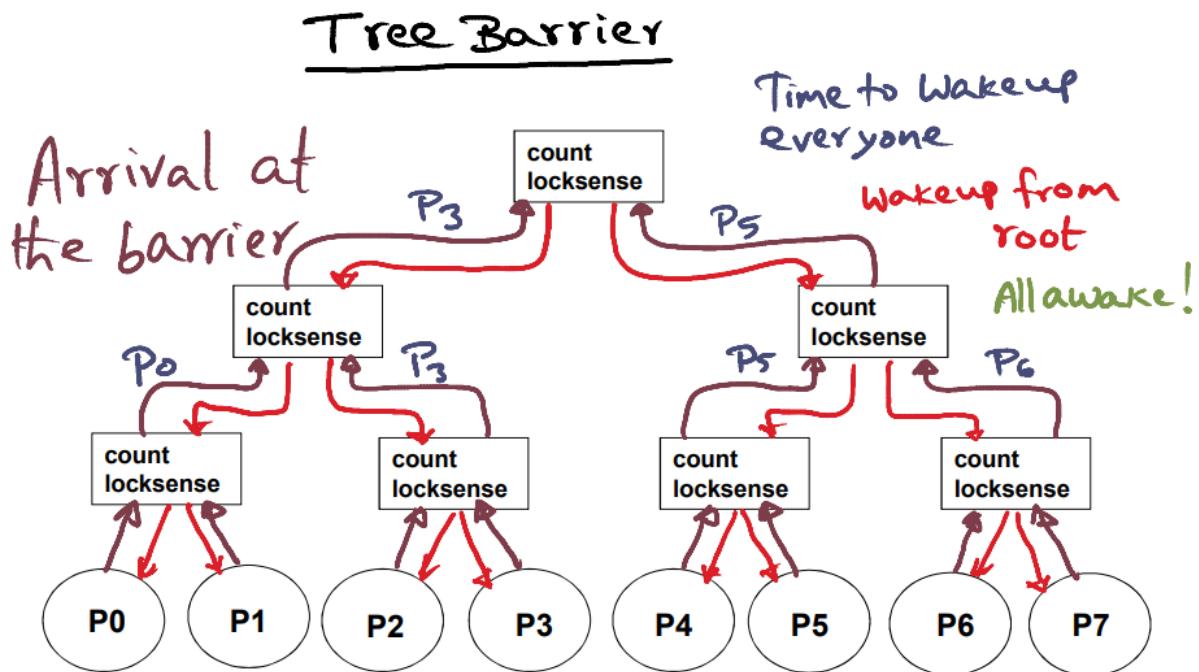


So the wakeup starts from the root. And, so in this case, P5 and P3 having been released from the root, they go come down to the next level. And they're going to wake up their buddies that are waiting at this level of the tree. Remember I told you that this can be a K-ary tree. K happens to be two in this case. But for any general K, basically, at every level of the tree, there's going to be on K minus 1 buddies waiting here, K minus 1 buddies waiting here. So what we're going to do is we're going to release that many prisoners from every level of the tree. So this is the zeroth level of the tree. There's the first level of the tree. There's the second level of the tree. At the zeroth level, there is k minus 1 buddies. At the first level, there are k times k minus 1 buddies waiting. And similarly as you go down the different levels of the tree, there're more and more buddies waiting to be released.

So for this simple example, with the K equal to two, when he comes up here, comes down to this level, P3 is going to release P0 and P5 is going to release P6. And so now we have more helpers, to go down the tree and wake up more people. So at this level, only P5 was there to

wake up P3, and at this level, both P3 and P5 are there to wake up the respective buddies, P0 in this case, and P6 in this case. So once P0 and P6 have been woken up, there are four of them now available that can go down to the next level of the tree. And they can go down to the next level of the tree. P0 can wake up, his buddies at this level of the tree, P3 his buddies at this level, P5, and P6. And so now all the others, so P1, in this case, P2 in this case, P4 in this case, and P7 in this case, we're all been waiting at this level of the tree, they will all get awakened because of these guys marching down from the root. And basically what each of these guys is doing on the way down is to flip this locksense flag. So the first thing that P5 did was to flip the locksense flag over here. That released this guy. And when, when P3 and P5 come to this level of the tree, each of them respectively flips the locksense flag that is associated with this data structure, and when they do that, P5 releases P6, P3 release P0, and now both P0, P3, P5 and P6 on this side. They all can go down to the next level. And P0 can flip locksense over here, P3 can flip locksense over here, P5 over here, P6 over here. That is going to release the rest of the buddies, P1, P2, P4, and P7. And everybody has now been released from the barrier, and that signals that the spin is done for all the barrier, the processes that I've been waiting for, and the barrier completion is complete.

8. Tree Barrier (cont)



So once, these locksense flags have been flipped, then all of the processors that have been waiting on these locksense as respective nodes, they're going to be released and everybody is now awake.

So the tree barrier is a fairly intuitive algorithm that builds on the simple centralized sense reversal barrier except that it breaks up this And processes into K-sized groups, so that they can all do spinning on a less contentious set of shared variables. So that's good. **It's a recursive algorithm that builds on the centralized sensor reverse algorithm, and allows scaling up to a large number of processes.** Because the amount of shading is limited to k, and so long as the k is small, like two or four, then the amount of contention for shared variables is limited to that number. So those are all good things about that, but there are lots of problems as well.

The first problem that I want you to notice is that **the spin location is not statically determined for each processor**. So for instance, if you take this particular execution that I've shown you in this picture, P0 happens to arrive later than P1. So P1 is the first to arrive here and so when P1 arrived here, it decremented count and it realized that "the count is not zero, I'm going to spend here". And P0 arrived later. And that's why it went up to the next level. And it is spinning on this locksense variable over here. So, in another execution of the same program, it is possible that P0 arrives first. If P0 arrives first, then it'll spin on its locksense variable that is in this data structure. And P1 will be the second guy to arrive, and therefore, he'll be the guy that will move up. And he'll be the guy that will be spinning on this locksense flag. So, the locksense

flag that a particular processor is going to spin on, is not statically determined. But **it is dynamically determined depending on the arrival pattern of all these processes at a barrier**. And the arrival pattern is going to be different for different runs of the program. Since it depends on the amount of code that is getting executed on each one of these processors. And other variables such as how busy the processor is and so on.

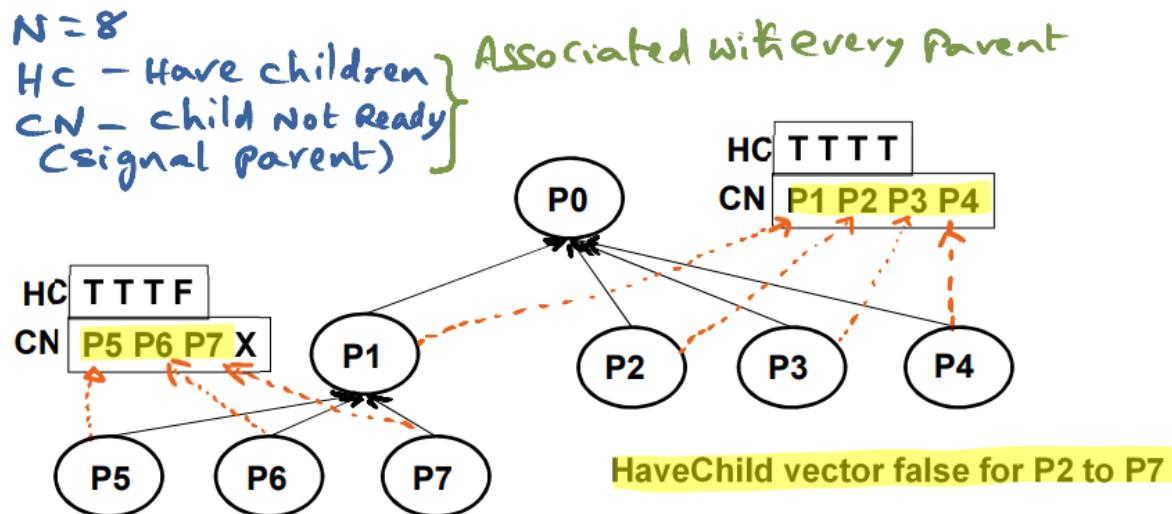
And the second source of the problem is that **the ariness of the tree determines the amount of contention for shared variables**. I've mentioned that you know, here it is showing, shown with two, two processors. But if you increase the ary of the tree to be key to be something more than two maybe four or eight or something like that. And if you have a large-scale multiprocessor with 1000 processors the array of the tree may be much more than 2, and in that case, the mode of contention for said data structures is going to be significant and that can result in more contention on the network as well.

The other issue with this Tree Barrier is that **it depends on whether our multiprocessor that we are executing this algorithm on is cache coherent or not cache coherent**. If it is cache coherent multiprocessor, then, you know, the spin, even though it's on a particular variable, it could be encashed in a private cache, and therefore, the cache coherent hardware will indicate when the spin variable changes value. But if it's a non cache coherent multiprocessor, the fact the spin variable that we have to associate with a particular processor is not static, but dynamic. Means that the spin may be happening for P0 on a remote memory. Remember I mentioned to you that one of the styles of architecture is a distributive shared memory architecture? Sometimes the distributive shared memory architecture is also called a **non-uniform memory access architecture, or NUMA**, And the reason it is called NUMA architecture is that the access to local memory for a particular processor is going to be faster than the processor's access to remote memory. And if you don't have cache coherence, then the spinning that has to be done has to be done on a remote memory, and that goes through the network. And so static association of the spin location of the processor is very crucial if it's a non-cache-coherent shared memory machine.

So the next algorithm that I'm going to describe to you is due to the authors of the paper that we are reviewing in this lesson, which is John Mellor-Crummey and Michael Scott, and for this reason, that algorithm is going to be called the MCS barrier. It's also a tree barrier but you'll see that in the MCS algorithm, the spin location is statically determined as opposed to the dynamic situation that you have in the hierarchy of the tree barrier here.

9.4 Ary Arrival

MCS Tree Barrier (4-ary arrival)



So the MCS tree barrier is also a tree barrier. It's a modified tree barrier, and what you'll notice, and once again, to make life simple, I'm showing you an arrangement of the MCS tree barrier with 8 nodes. And it's a 4 ary arrival tree. **The arrival tree and the wake-up tree are different in the MCS algorithm.** The arrival tree is a 4 ary tree, and I'm showing the arrangement for N equal to 8. There are 2 data structures that are associated with every parent, this one data structure is what is called have children, and the other data structure is what is called child not ready. And I'll describe to you what each one of these things is.

Having children is a data structure that is associated with every node. This data structure is going to have meaning only when a node is also a parent. So for example, if you look at this arrangement, node P0 has 4 children, P1, P2, P3 and P4. And if you look at node P1, it has 3 children. And so, P5, P6 and P7, has 3 children. And so we have a total of 8 processes, so we've got all 8 processes accounted for here. And therefore, these guys, P2, P3, P4, all the way up to P7, they're not as lucky as P0 and P1. They don't have children. So P2 through P7, they do not have children. And therefore, their HaveChild vector is false. So what you see here is a HaveChild vector and the HaveChild vector is true for P0 in all the big positions. And indicating that it has because it's a 4 ary tree, it can potentially have up to four children. And yes, P0 has 4 children. And the have child vector is true all the way, whereas, for P1, the have child vector is true for the first 3 children and false for the fourth because it has only 3 children. And these guys don't have any children. And similarly, these guys don't have any children. So, the HaveChild vector is completely false for P2 through P7.

Now, what about this Child Not Ready data structure? **The Child Not Ready data structure is a way by which each of these processes has a unique spot in the parent to signal when**

they are arriving at a barrier. So what I'm showing you here, the arrows here are showing you the specific spot in this data structure, the child not ready data structure associated with his parent, for each of the children, there is a unique spot for this guy to indicate that they've arrived at the barrier. And similarly, for this set of children, the parent is P0 and each child has a unique spot in the parent's child not ready vector to indicate that they've arrived at the barrier.

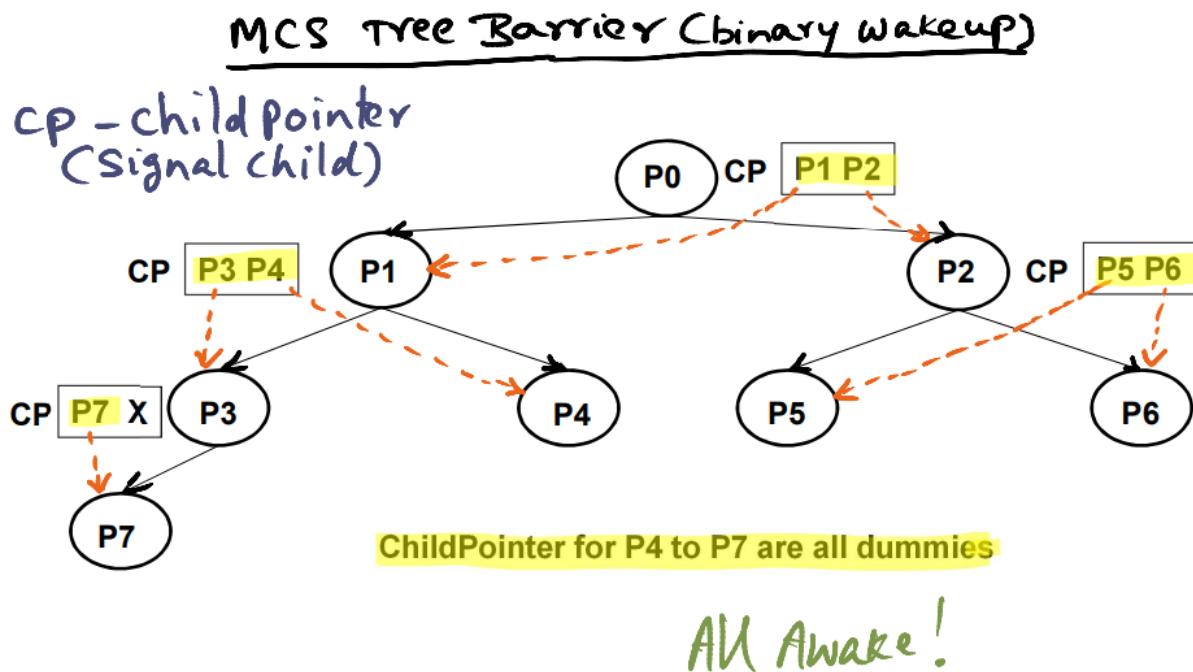
So the black arrows in this structure that I'm showing you are just showing the arrangement of the tree. And in terms of the parent-child relationship, for the 4-ary arrival tree. And the red arrows are the ones that are showing you the specific spot where a particular child is going to indicate to the parent that they have arrived at the barrier.

And as you can see that since P1 has 3 children, the fourth spot is empty indicating it has to wait only on 3 children to know that the value is completed on the tree and so it can move up. So, **the algorithm for barrier arrival is going to work like this: when each of these processors arrives at a barrier**, what they going to do is **they going to reach into the parent data structure on very specific spots statically determined**. That's important, right? So it's statically determined that this is a spot that P5 is going to indicate to the parent that it has arrived. This is the spot that P6 is going to indicate that it has arrived. P7, and similarly, **once all these guys have arrived at the barrier, P1 can check**, and the way P1 checks is, just sees **whether this CN vector has 1 in all these spots**. If there are ones in all these spots, it can spin on this, and **therefore, it knows that its children have arrived at the barrier. Once its children have arrived at the barrier, then it can move up the tree similar to what we saw in the vanilla tree barrier before. P1 is going to move up, and it's going to inform its parent**. And the way it does is by going to a specific spot in the parent's child not ready vector. And there is a specific spot assigned for P1. It's going to set this to indicate that it has arrived at the barrier. So what P0 is doing is waiting on everybody to arrive. If P0 is the first let's say to arrive at the barrier. It's waiting on everybody else to arrive at the barrier. Could be P0 is the first one or the last one, it doesn't matter. When P0 arrives at the barrier, it is going to wait on this child not ready all the bits being set by the children. And so, when each of these nodes arrives at a barrier, they know because of the arrangement of this data structure, they know their position in the data structure relative to other processes arriving at the barrier. And therefore P2, when it arrives at a barrier, It knows that all it has to do, given the structure, has to go to this part on the parent vector and set it to 1. P3 has to go to this part set it to 1 and so on, okay? And so once it is done, P0 will know that everybody has arrived at the barrier. So, that's the arrival at the barrier.

So once again, the recap. The arrival tree is a 4-ary tree. And the reason why they chose to use a 4 ary tree is that **there is a periodic result backing the use of 4 ary tree leading to the best performance**, and that's the reason that they chose this particular arrangement. And the second thing that I want you to notice is that each processor is assigned to a unique spot by construction, a unique spot in this 4 ary tree. And because of its unique spot, a particular process on may have children, or may not have children and in this case, I showed you that P0 and P1 have children, and the rest are not as lucky, because N is equal to 8. The other nice thing about this particular arrangement is that in a cash coherent multiprocessor, it is possible to

arrange so that all the specific spots that children have to signal the parent can be packed into one word of a processor and therefore, a parent has to simply spend on one memory location to know the arrival of everybody, so it doesn't have to individually spend on memory locations for different processes, they can all be packed into one word, and the cache coherence mechanism will ensure that P0 is alerted every time any of these guys modify this shared memory location.

10. Binary Wakeup



So the wakeup tree for the MCS barrier is a binary wakeup tree. Once again here, there's a theoretical result that backs this particular choice that the **shortest critical path** from the root to the last awakened child, is shortest when you have a binary wakeup tree, and that's the reason that they chose to have this construction. **Even though the arrival tree's a 4 Ary tree. The construction for the wakeup tree is a binary tree.**

And let me explain the construction of this binary wake-up tree. Every processor is assigned a unique spot again. So P0 the root and P1, P2 over here, P3, P4, P5, P6, and P7. So that completes the eight processes for this binary tree set-up for wakeup. And **the latest structure that is used in the wakeup tree is as a child pointer data structure**. And the ChildPointer data structure is essentially **a way by which a parent can reach down to the children and indicate that it is time to wake up**. So, that's the purpose of this ChildPointer data structure. And, once again, as you can see, depending on the particular location in this wakeup tree, they may have children, they may not have children. So, P0 has two children, P1 has two children,

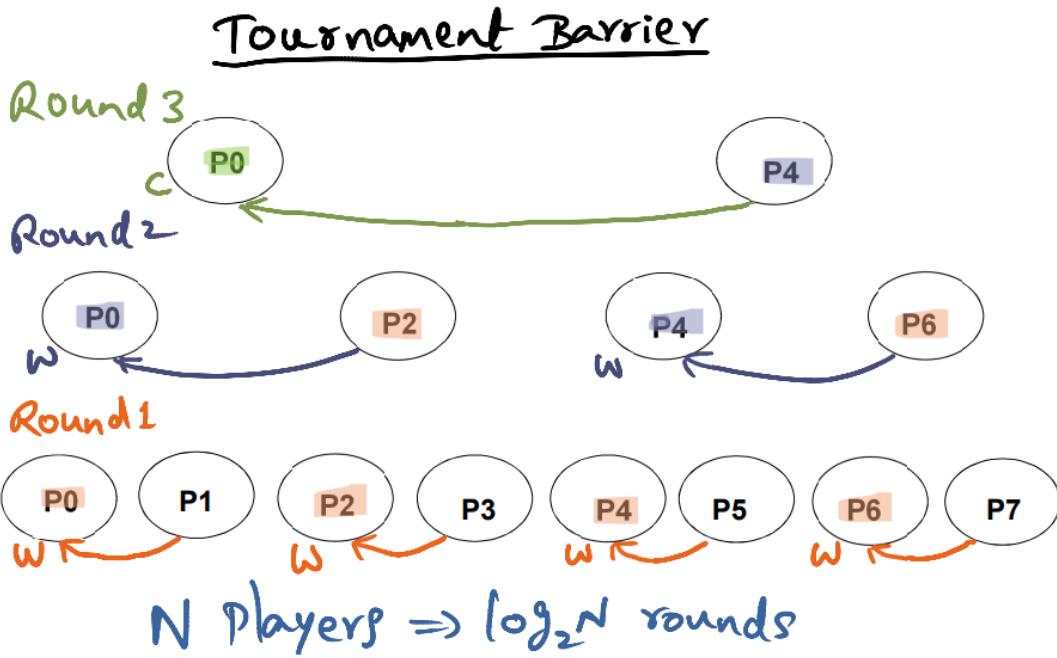
P3 and P4. P2 has two children, P5 and P6. P3 had one child, P7, and that is it. Because you have processors and these guys. Don't have any children P4, P5, and P6.

So in terms of waking up, what is going to happen is that when everybody arrives at the barrier P0 is going to be noticing it, and through the arrival tree. And so now it says "oh, it's time now to wake up everybody", and the way it does that, it has a specific pointer To reach into P1 and signal to P1 that it's time to wake up. And similarly, it has a specific pointer in P2 to wake up. So a particular memory location, which is a pointer to a location that this guy's waiting on to wake up. So it's going to do that. And so what is going on is that again, this is another important point that to know that it is time to wake up, each one of these processes is standing on a statically determined location. P2 is standing on a particular location here, and, and P1 is standing on a particular location here. And so when P0 signals P1 it is exactly sending a signal to P1 and it is not affecting any of the other processes. And similarly, when it signals P2 it signals exactly P2 using this pointer. And similarly, once P1 and P2 are woken up. They can march down the tree and signal P3 and P4, and signal P5 and P6 by using the statically assigned spots that the children are spinning on to indicate that it is time to wake up.

So, the key point I want to stress again is the fact that In this construction of the tree, by design, We make sure that we know a position in the tree and we know exactly the memory location that we have to spin on, in order to know that it is time to wake up. So these red arrows show the specific location that is associated with each one of these processors In the wakeup tree. So once the parents signal the children and they marched down and signal all the other children, then at that point, everybody's awake, and the barrier has been reached.

So the **key take away** point with the MCS tree barrier is that the wakeup tree is binary. The arrival tree is forwarding and the static locations associated with each processor, both in the arrival tree that we saw earlier and the wakeup tree. And through the specific statically assigned spot that each processor can spin on, we are making sure that **the amount of contention on the network is limited**. And also by packing the variables into a single data structure we can **make sure that the contention for shared locations is minimized** as neat as possible.

11. Tournament Barrier



Okay, the next value algorithm we're going to look at is what is called the **Tournament Barrier**. The barrier is organized in the form of the tournament with N players and since it's a tournament with N players and two players playing against each other. **In every match there are going to be $\log N$ rounds, $\log N$ with a base 2.**

So here is the setup for with 8, they're going to be, they're going to be three rounds corresponding to $\log(N)$. And being eight we get three rounds. The first round, second round and the third round. So in the first round, they're going to be four matches. P0 and P1 is one match. P2, P3. P4, P5. P6, P7. And the only catch is that we're going to rig this tournament. In other words what's going to happen is that **we're going to predetermine who is going to be the winner in this round**. And particularly, we're going to say P0 is the winner for this match, P2 for this one, P4 for this one, and P6 for this one. So in other words, the matches are rigged. In this day and age, when we hear about international scandals about match fixing. I guess this is not too far fetched.

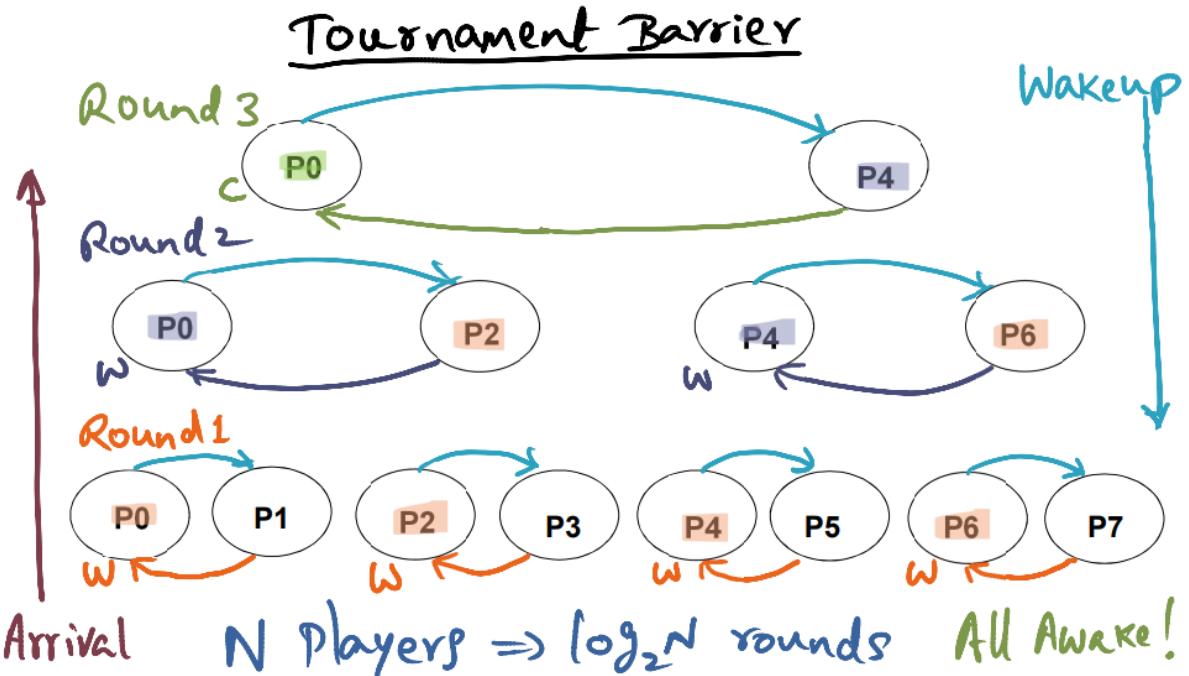
But what is the rational for match fixing? **The key rational is the fact that if the processes are executing on a shared memory machine**. Then the winner can basically sit on his bumper and wait for a process of P1 to come over and let him know that he has won the match, P2 can wait until P3 comes over and so on and so forth. And what that means **in a shared memory multiprocessor, is that the spin location** where P0 is waiting for P1 to come and inform him that he's lost the match **is fixed/static**. And so this is the idea behind match fixing, that the spin location for each of these processes, P0, P2, and P4, and P6, the winners in the first round, is predetermined. And **that is very useful, especially if you don't have a cache coherent**

multiprocessor. If you have NCC NUMA machine, in that case, it is possible to locate the spin location in the memory that is very close to P0 P2 P4 and P6 respectively. That's the idea behind this this match fixing.

So the result of matches, of course, P0 will advance to the next round. P2 will advance to the next round. P4 and P6. And once again, in the second drawing we're going to fix the matches. And the winner is going to be P0 for round 2. P4 for in this bracket for round 2. And so essentially what that means again, is that P0 and P4 can spin on a statically determined location in various processors and P2 and P6 respectively will come over and let the other guy know that when the match for this round. So that is the end of the second round. And of course, if you have you know, with N equal to eight, there are only three rounds but, if, for arbitrary N, we're going to have more levels in the tree and the and the every level. We're going to fix the the winners and, so it'll propagate up this tree in this fashion, in terms of determining statically, who are going to be the winners for each round of the tournament. And this will go on, all the way up to determining who the tournament champion is. So in this case, P0 is our luck guy, who wins the tournament and so he's the champion. And so P0's going to be waiting on a statically determined location, where P4 can come and signal that P0 has one determinant.

So again, the important thing that I want you to get out of this this particular arrangement that I've mentioned is the fact that **the spin location for each of the processors that are waiting on the other guy are statically determined at every level.** So this the first round, the second round, and finally the championship, the championship round.

12. Tournament Barrier (cont)



So at this point, when p0 is declared the champion of the tournament, what we know is that everybody has arrived at the barrier. And this knowledge is available with p0 but not with anybody else. So everybody, everybody has arrived at the barrier, but P0 is the only one who knows because he's a champion, he knows that, that everybody has arrived at the barrier. So clearly, the next thing that has to happen is of course free up all the processors to indicate to them that you know, its time to move on to the next phase of your computation. So let's talk about the wake up. So what p0 is going to do is is going to tell p4 that it's time to wake up. And you know, if you want to use the tournament analogy again, in any tournament the winner walks over to the loser and shakes hands, right? So, you can sort of think of the same thing happening over here, P4 with the **???** face is waiting for P0 to come over, and let him know that okay, its a good match and shake hands with you. And so, P0 is going to come over and let him know, shake hands. So that's the first thing that happens. So in other words, at this point P0 is awake of course, and he is also waking up P4 saying that well barrier is done. And now one of these guys can go to the next level and do the honours at every level so, just as I said about P0 coming in and shaking hands with P4, what P0 is going to do is, go to the next round and shake hands with P2, P4 go to the next round and shake hands with P6 and, and so on. And of course, if you think about the analogy of a tournament, as soon as the match is over, the winner is going to shake hands with the loser. But in this case, the winner shakes hands with the loser after the tournament is all done. So at every level, we're going to have that. So, its essentially, P0 and P4 come down to the next level and they shake hands with the respective losers of that level. And as I said, if we have for some arbitrary N, where N is a binary power, you're going to have this kind of propagation of wake-up signals going from the winner to the loser at every round. And all of them wake-up and go to the next level. Because all of these guys are winners

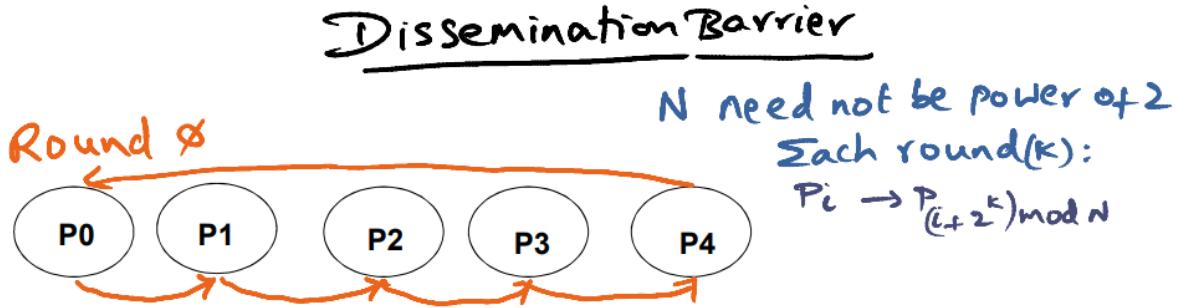
from the previous level. So, all of these winners will go down to the next level and wake up the losers at that level. So that's what is going to happen. Again, what that means from the point of view of a shared memory multiprocessor is that the spin location for P4, P2, and P6, it's all fixed, right? Statically determined. If P4 knows that P0 is going to come over and shake hands, and so that he can spin on a local variable that is close to its processor, and so again this is important for NCC NUMA machines in which there is no cache coherence and therefore it is convenient if P4 can be spinning on a memory location that is close to the processor. Same thing with P2 and P6 at the next level. So this process of waking up the losers at every level goes on till we reach round 1. And when at round 1, all the winners have congratulated. Well, not congratulated, but shook hands with the respective losers at the first round. At that point, the wake up is complete. Everybody's awake now. And, and, the, the barrier is done. So all are awake, and the barrier is done, and they can move on, the next phase of the computation. And once again, in order to make sure that there is sense reversal, everybody knows that this barrier is done, and they're going to go to the next phase of the computation where they will wait on the different sense of the barrier. . So, that's Tournament Barrier Algorithm. So the 2 things that I want you to take away is, the arrival moves up the tree like this, with match fixing. And all the respective winners at every round, waiting on a statically determined spin location. And similarly, when the wake up happens, the losers are all waiting on statically determined spin location in their respective processors and the winner comes over at every level at every round of the tournament, the winner comes over and tells the loser that it's time to wake up. So that's how this whole thing works. So now that we understand this tournament algorithm let's talk about the virtues of this algorithm.

13. Tournament Barrier (cont)

You will immediately notice that there's a lot of similarity between the Tournament algorithm and the sensor verse, interversing tree algorithm and also similarity to the MSC algorithm. So let's talk about the difference between the tree barrier and the tournament barrier first. So the, the main difference is that in the tournament barrier, the spin locations are statically determined, whereas in the tree barrier we saw that the spin locations are dynamically determined based on who arrives at a particular node in the barrier, in the tree in that algorithm. And what that means in the tournament barrier is that we can statically assign. The spin location for the processes at every round of the tournament. Another important difference between the tournament barrier and the tree barrier, is that there is no need for a fetch and free operation. Because all that's happening at every level. At every round of the tournament, there is spinning happening. And what is spinning? Basically reading. And there is the signaling happening, what is this? This is just writing. So, so long as we have atomic read and write operations in the multi-processor, that's all we need in order to implement the tournament barrier. Whereas uh, if you recall in the tree barrier we need fetch and free operation in order to atomically decrement the count variable. So that doesn't exist in the tournament barrier. That's, that's another good use. Now what about the total amount of communication that is needed. Well, it's exactly similar because of the tree arrangement. As you go up the tree the amount of communication that happens is going to decrease. Because the tree is getting pruned as you go

towards the root of the tree and so the amount of communication in the tournament barrier in terms of all the notation is exactly similar to the tree barrier it is older, login. That's the amount of communication that is needed. Now the other important thing that that I should mention is that at every round of the tournament you can see that there, there's quite a bit of communication happening. In the first round going up the tree, P1 is communicating with P0, P3 with P2 and so on. All of these red arrows. Are parallel communications that potentially take advantage of any inherent parallelism that exists in the interconnection network. So that's good news. That, that all of this communication can happen in parallel if the interconnection network allows that kind of parallelism. That can be exploited. And the, the other important point that I want you notice is that the tournament barrier works even if the processor is not a shared-memory machine. Because all that we're showing here is a message communication. So P1, P0 is waiting for a message from P1, and so on. So all of these arrows you can think of them as messages. And so even if the processor the multiprocessor is a cluster, well by a cluster what I mean is a set of processes in which the only way they can communicate with one another is through message passing. There is no shared memory, no physical shared memory. And even in that situation the, the tournament barrier will work perfectly fine to implement the barrier algorithm. Now let's make a comparison of tournament to to NCS. Now because this tournament is arranged as a tournament there are only two processes involved in this communication at any point of time in the parallel. So it means that it cannot exploit the spatial locality that may be there in the caches. If you recall, one of the virtues of the NCS algorithm is that it could exploit spatial locality. And that is, multiple spin variables could be located in the same cache line and, and the, the parent for instance could spin on a spin location to which multiple children are going to come and indicate that they are done. That's not possible in the, the tournament barrier because it is arranged as a tournament where there are two players playing against each other in every match. Similar to MCS, Tournament Barrier does not need a fetch and fee operation, so that's good. A common good property of both MCS and Tournament. The other important thing what tournament has an edge over MCS is the fact that tournament barrier works even if the processors are in a cluster. Meaning, it's not a shared memory machine and is only a cluster machine where only message passing is really good communicator to one another. Even in communicating that situation, you can implement the tournament barrier. So that's, another good thing. Now is a good time for me to mention to you. I've been using the word cluster what that means is that the set of nodes in the multiprocessor they don't physically share memory and the only way they can communicate with one another is through message passing. And is important for you to know this particular terminology cluster because clusters become the work horses for data intensive computing today. The data centers and content distribution networks we're going to see a lot of that when we talk about giant skin services later on in this course, and those environments, they all use this kind of a, a, a computation cluster. And these computation clusters employ on the order of thousands or 10000 nodes connected together through an interconnected network and they can operate as a parallel machine with only message passing as the vehicle for communication among the processes.

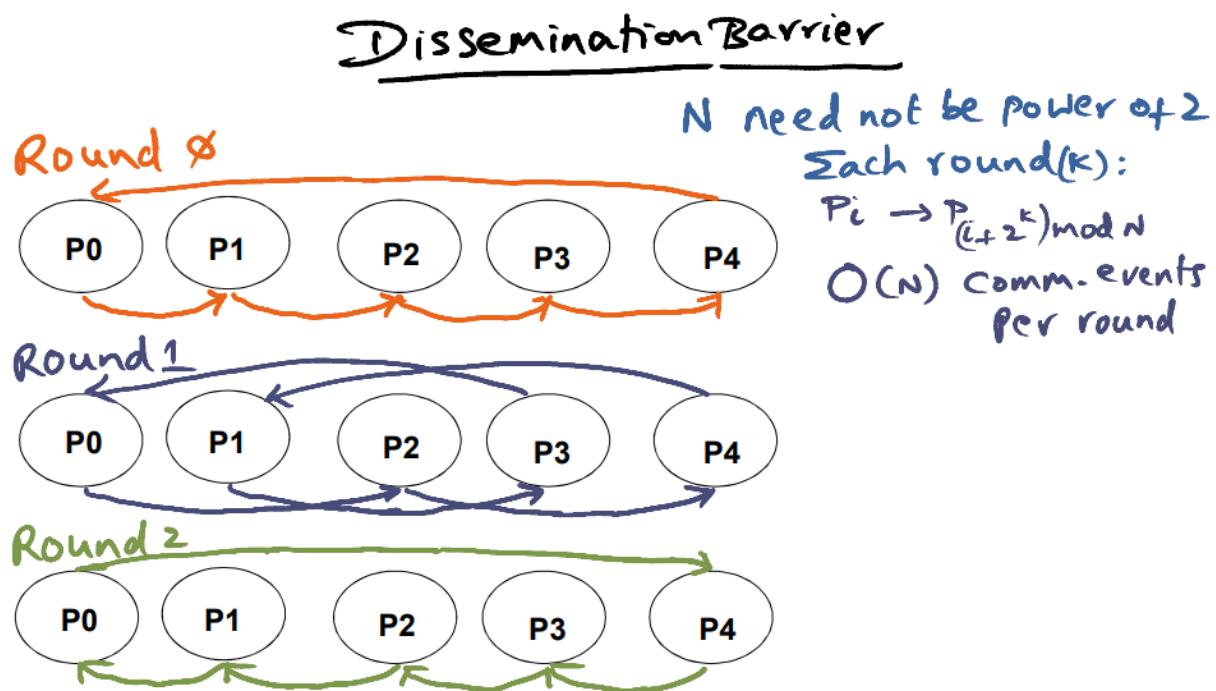
14. Dissemination Barrier



The last barrier algorithm I'm going to describe to you is what is called a Dissemination Barrier. And it works by information diffusion in an ordered manner among the set of participating processes. And, what you will see is that it is not pairwise communication as you saw in the tree barriers and the NCS barrier or the tournament barrier. But it is through information diffusion. The other nice thing about this particular barrier the dissemination barrier, is that it is since it is based on ordered communication among participating nodes. It's all like a, a well-orchestrated gossip protocol. And therefore, N need not be a power of 2. And you will see why this condition need not be satisfied as I start describing the algorithm to you. So what's going to happen is that, there's going to be information diffusion that's going to happen among these processors in several different routes. And in each round what's going to happen is that a processor is going to send a message to another ordained processor. And the particular processor that it's going to choose to send it is dependent on the round which we're in. So, the idea is that processor P_i will send a message to processor $P_i + 2^k \bmod N$. This is the peer to which P_i is going to send a message to. And of course you know an example is always more illustrative. So since we have five processors here, we can then figure what's going to happen in every round. And Round 0 k is going to be zero. So what's going to happen, is that since k is zero, Round 0, P_0 is going to be sending a message to $P_1 + 2^0 \bmod 5$, k being zero, is going to send it to P_1 . So, P_0 sends a message to P_1 . Similarly P_1 sends a message to P_2 , P_2 sends to P_3 and P_3 to P_4 . And the arrangement is that this is cyclically arranged. That if before the neighbor for him is going to be in the cyclic order whoever is the next neighbor. So, in this case since there is mod function that we using before is going to be sending its message to processor $P_5 \bmod 5$. N , N being, N , N being 5, it will be sending the message to P_0 . So this is Round 0 of the communication. So the key thing that I want you to get is that in every round, we're going to see more rounds in a minute in every round a processor is sending a message to an ordained processor based on their own number. But depending on their own numbers, their own number zero, P_0 sending to P_1 and so on and so forth. And that is what's going on. So this completes one round of gossip. And what you, what you want to see it that. All of these communications that I'm showing you are parallel communications. They're not waiting on each other. So P_1 , whenever it's ready to arrive at a barrier it's going to tell the next guy, P_2 is going to tell the next guy when he's ready and so on. Now how will these guys know that Round 0 is done well, if you take any particular process here let's say P_2 , as soon as it gets a message from P_1 and it has

sent a message to P3. It knows that Round 0 has done so far is as P2 is concerned, it can progress to the next level, or next round of the dissemination. So, each of these processes are independently making a decision that the round is over based on two things. One is, they have sent a message to the peer and they want to receive the message from the ordain neighbor that they're supposed to get it from. At the end of that, they can move on to the next round.

15. Dissemination Barrier (cont)

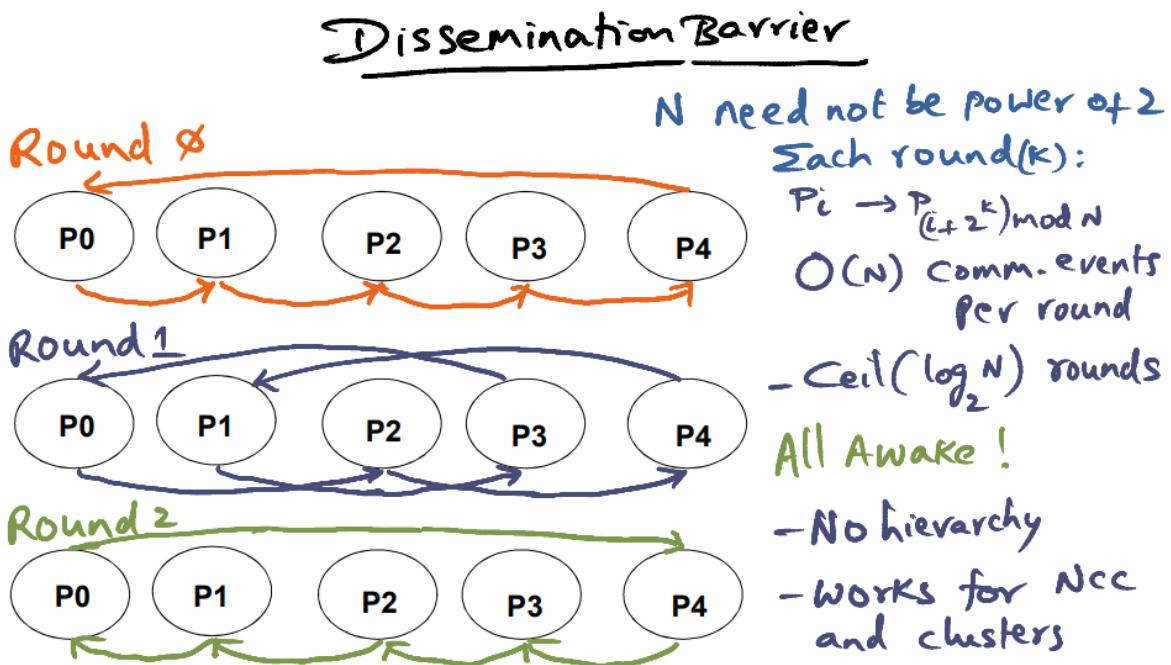


Now how many communication events are happening in every round? Well, it's order of N communication events per every round, because every processor is sending a message to another processor in, in every round. And therefore, the amount of communication that's happening is order of N , where N is the number of processors that are participating in this barrier. So now you can quickly see what's going to happen in the next round, and the next round, k is going to be equal to one and therefore. I'm going to choose, each processor is going to choose a neighbor to send the message to based on this formula that I've, that I have here. So in round zero, for instance, what we did was, P0 is sending a message a neighbor that is one distant from it because k was zero. And now. In round one case one and therefore P0 is going to be sending a message to a neighbor that is two distant from it. So P0 will send to P2 and similarly P1 two distant from it P3, P2 two distant P4. P4. Too distant from it, in cyclic order, is going to be P1. So it's sending a message to P1. So that's the round one of communication with k equal to one, round one of communication. Where once again, order of N messages are being exchanged among these processes to indicate that this round is complete. Just as I said

about Round zero, every processor will know that this round is complete when it receives a message from its ordained neighbor. So in this case, P2 is going to expect to receive a message from P0, and it has also sent its message to P4, its ordained neighbor to which it is supposed to send the message in this round. Once it is done, P2 knows that round one is over and it can progress to the next round. So the independent decision is being made by each one of these processes in terms of knowing that this particular round is over and they can progress to the next round of the dissemination barrier. And just as I mentioned in the previous round. All of these communications happen in parallel, so if the interconnection network has a redundant parallel path these parallel, these parallel paths can be exploited by the dissemination barrier. In order to do this communication very effectively. So the next round, meaning round number two. K is equal to two, and therefore, what we're going to do is, every one of these processors is going to be choosing a neighbor that is four distant from itself. So in other words, P0 is going to send a message to its neighbor that is four distant, that is, P4. P1 is going to send it to four distant. Which means if you wrap it around, it's going to be P0 and so on. So this is the communication that's happening in round two where every processor is sending a message to its neighbor who is four distant because gave with the two four distant from itself. So just sort of biz, belaboring the point the gossip in round two. Is over, so far as P3 is concerned, when it has received a gossip message from its four distant neighbor, which happens to be P4. And it has also sent a message to its four distant neighbor, P2 in this case. At this point, every one of these processes knows that round two of gossip is complete. And, similar to what I've been emphasizing all along, parallel communication pads in the interconnection network, can be fully exploited, by the dissemination barrier algorithm.

16. Barrier Completion

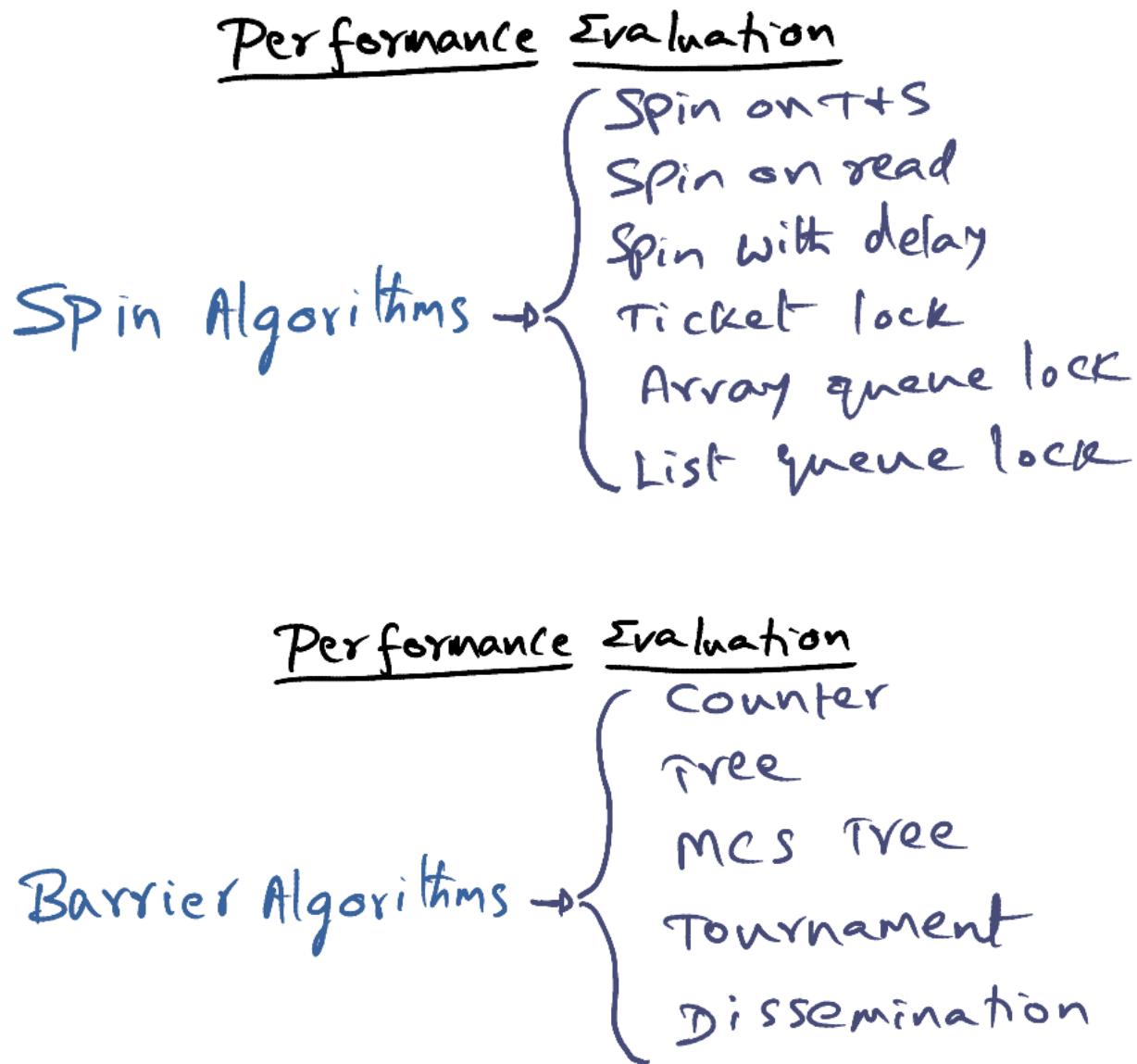
17. Dissemination Barrier (cont)

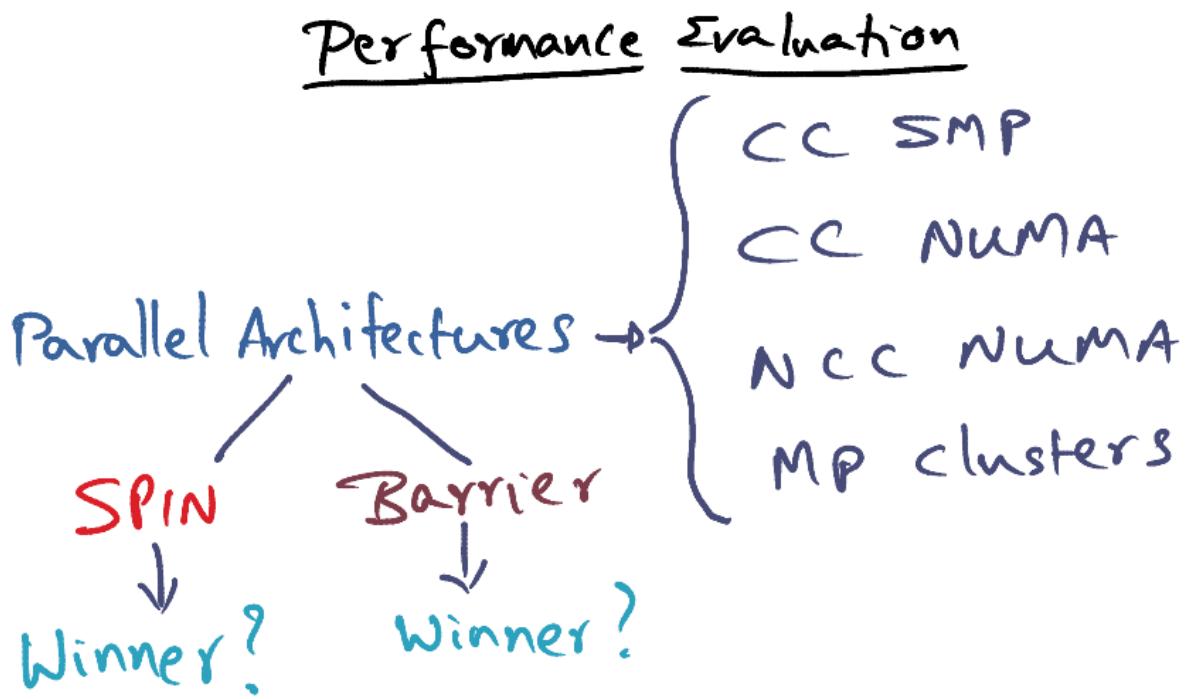


So, with any row of five, at the end of round two, every processor has heard from every other processor in the entire system. Right? So you can eyeball this figure and see that every processor has gotten the message from every other processor, and so it's common knowledge that for every processor that every one else has also arrived. Add the barrier. So, how many rounds does it take to know that everybody has arrived at the barrier? Well it's ceiling of $\log N$ to the base two. You add the ceiling because it's N need not be a part of two. So at this point, everybody is awake. So, barrier completion here there is no distinction between arrival and wake up as you, as you saw. In the case of the tree barrier or the MCS barrier or the tournament barrier. In the dissemination barrier, because it is happening by information diffusion, at the end of a ceiling of $\log n$ rounds, everybody has heard from everyone else in the entire system So everyone knows that that barrier has been breached. So in other words, in every zone of communication in the dissemination barrier, every processor, you eyeball any particular processor. Every processor is receiving exactly one message, exactly one message in every round of the dissemination barrier. So overall During the entire dissemination barrier information diffusion that's going on, every processor is receiving a total of a ceiling of $\log n$ to the base two messages. Every round one message, and so they are ceiling of $\log n$ rounds and so a ceiling of $\log n$ to the base two is a number of messages that any given processor is receiving and once. Every processor has received this $\lceil N \log_2 N \rceil$ number of messages. It knows that the barrier is complete. It can move on. And I've been using the word message in describing this dissemination barrier. It's convenient to do, to use that word because it is information diffusion but if you think about a shared memory machine, a message is basically a spin location. And, once again because I know an ordained processor is going to talk to me in

every round, the spin location for this guy is statically determined. Spin location, statically determined, and so on. So every round of the tournament, we can statically determine the spin location that a particular processor has to wait on, in order to receive a message. Which is really a signal from its ordained peer, for that particular round of the dissemination barrier. So, again static determination of spin location becomes extremely important if the multiprocessor happens to be an NCC NUMA machine. In that case, what you want to do is to locate the spin location. In the memory that is closest to the particular processor. So that becomes more efficient. And as always, in every one of these barried algorithms, you have to do sense reversal. That once this barrier is complete, everybody is going on to, to the next phase of the competition. And when they go to the next phase of the competition. They have to make sure that the barrier that they arrive is the next barrier. So you need sense reversal then, it needs to happen at the end of. Everybody algorithm. So now let's talk about some of the virtues of the dissemination barrier. The first thing that you'll notice is in the structure, there is no hierarchy. In the tree algorithms, the root of the tree automatically gives you a hierarchy in terms of The organization of the tree in the dissemination barrier, there's no such thing. And I already mentioned that this algorithm works for both NCC NUMA machine, as well as clusters. That's also a good thing. And there is no waiting on anybody else. So every processor is independently making a decision to send a message. As soon as it arrives at the barrier. Is ready to send a message to its peer for that particular round. And of course, every processor can move to the next round only after it has received a corresponding message from its peer for this particular round. So as soon as that happens, it can move on to the next round of the dissemination barrier. And all the processes will realize that the barrier is complete when they received Ceil ($\log N$) messages in the entire structure of this organism. So if you think about the total amount of communication, because the communication in every round is fixed, it's N messages in every round and since there are Ceil ($\log N$) rounds, the communication complexity. Of this algorithm is order and log in. Compare that to the communication complexity of the tournament, or the the Tree barrier. In both of those cases, the communication complexity was only log in, because of the hierarchy, as you go toward the root of the tree, the amount of communication shrinks, so. The amount of communication in those algorithms is only order of $\log N$. Where as, in this, this simulation down here, since there is no hierarchy, the total amount of communication in the, algorathim is order of analog N .

18. Performance Evaluation





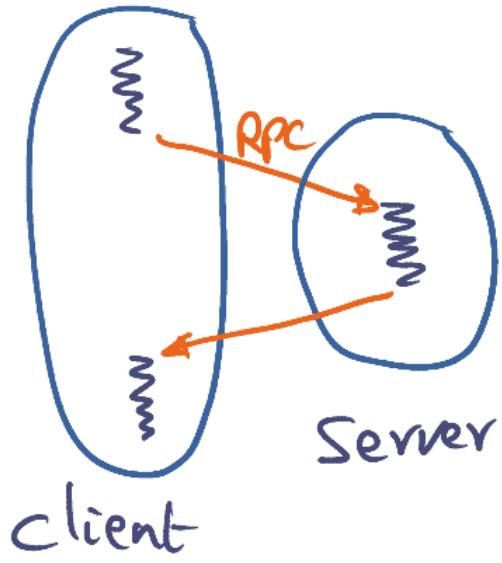
L04d: Lightweight RPC

1. RPC and Client-Server Systems

- Lesson Outline
- ✓ Machine model
 - ✓ Synchronization
 - Communication
 - Scheduling
 - Parallel OS case studies

RPC and client - server Systems

E.g. File System



RPC: usually remote
client - Server on
same machine?
— Performance
Vs.
Safety

- 2. RPC Vs Simple Procedure Call**
- 3. Kernel Copies**
- 4. Copying Overhead**
- 5. Making RPC Cheap**
- 6. Making RPC Cheap (Binding)**
- 7. Making RPC Cheap (Actual Calls)**
- 8. Making RPC Cheap (Actual Calls) cont**
- 9. Making RPC Cheap (Actual Calls) cont**
- 10. Making RPC Cheap Summary**
- 11. RPC on SMP**
- 12. RPC on SMP Summary**

L04e: Scheduling

- 1. Scheduling First Principles**
- 2. Scheduler**
- 3. Memory Hierarchy Refresher**
- 4. Cache Affinity Scheduling**
- 5. Scheduling Policies**
- 6. Minimum Intervening Policy**

7. Minimum Intervening Plus Queue Policy

8. Summarizing Scheduling Policies

9. Scheduling Policy

10. Implementation Issues

11. Performance

12. Performance (cont)

13. Cache Affinity and Multicore

14. Cache Aware Scheduling

15. Scheduling Conclusion

L04f: Shared Memory Multiprocessor OS

1. Shared Memory Multiprocessor OS Introduction

2. OS for Parallel Machines

3. Principles

4. Refresher on Page Fault Service

5. Parallel OS and Page Fault Service

6. Recipe for Scalable Structure in Parallel OS

7. Tornado's Secret Sauce

8. Traditional Structure

- 9. Objectization of Memory Management**
- 10. Objectized Structure of VM Manager**
- 11. Advantages of Clustered Object**
- 12. Implementation of Clustered Object**
- 13. Non Hierarchical Locking**
- 14. NonHierarchical Locking (cont)**
- 15. Dynamic Memory Allocation**
- 16. IPC**
- 17. Tornado Summary**
- 18. Summary of Ideas in Corey System**
- 19. Virtualization**
- 20. Virtualization to the Rescue**
- 21. Shared Memory Multiprocessor OS Conclusion**