As we saw in the video the idea of inheritance locks is to have a prevention mechanism against deadlock and starvation.

**You** might wonder why bother? Well, even big companies like IBM and Oracle occasionally find bugs in their software which cause deadlock and, what is worse, even in safety critical systems deadlocks have been detected. So it is really crucial to address this problem. So I will first talk a bit about deadlock and starvation, which should hopefully look familiar to most of you, just to give a bit of background information.

**So** there are four basic properties which must all hold if a deadlock occurs no matter how complicated the system is. In other words, if one of these four properties does not hold, no deadlock can occur. The first property is mutual exclusive access, that means if I have a resource I will not give it to anyone else and I won't share it. The second property is that a wait-and-hold condition must exist. That means I already have something, but I am waiting to get something else as well.

*The* third property: Resources cannot be pre-empted, that means there is no god or superpower that can take anything away from me. And the last property is a circular wait, that means I wait for John, John waits for James and James waits for me for example. So in a nutshell, if I write a deadlock prevention protocol, I simply need to make sure that it is impossible for the four properties to be true AT THE SAME TIME.

**And** that's exactly what people did; in particular Havender explored many deadlock prevention protocols. Let's look at one particular approach Havender suggested. The protocol goes along the lines: "If a locking-request cannot be satisfied right away (because the requested resource is locked by another process), all resources held by the requesting process are pre-empted (that is, made available to be locked by any other process) and the requesting process can only resume once the requested and the previously locked resources are available."

**2:15 [TAP PHONE]**

**So** let's look at an example: Here Process 1 holds Resource 3 and 4. Now Process 1 also wants Resource 2.

**Unfortunately** Resource 2 is already assigned to Process 3. That means Process 1 has to wait and therefore all the Resources held by Process 1 are pre-empted. That means, while Process 1 is waiting, Resources 3 and 4 can be used by a different process.

**Once** Resource 2 becomes available, Process 1 reacquires Resources 3 and 4, acquires Resource 1 and moves on. So far so good, but what if Process 1 cannot reacquire, say Resource 4, because someone else, for example Process 2, is using it now?

**Well** in that case Process 1 has to wait for Resource 4 and cannot acquire Resources 2 and 3 until it becomes available. That's annoying; because by the time Resource 4 becomes available someone else might have acquired Resource 3 or even Resource 2 again. In fact, Process 1 can only continue if ALL THREE resources are available at the same time and there's just no guarantee that this will ever be the case. Such 'a piggy in the middle' situation is what is called starvation.

**Now**, what's interesting about starvation is the fact that only some processes will suffer under starvation and typically also just for a limited amount of time even though this is not guaranteed. This makes starvation very difficult to detect and there is no common approach how to deal with starvation.

**3:45**

*During* my project I came up with this rule of thumb here which essentially reflects the two antagonisms: "NO*,* I won't give you my resources under any circumstances and if it comes to deadlock" vs. "Sure have all my resources as long as you like, I'm waiting here anyway". This makes it pretty obvious that the sweet spot lies somewhere in between. This is exactly where inheritance locks aim to be.

*So* here comes the key idea of the entire project: Do not allow anyone to take one of your resources, unless you are waiting for the requester either directly or indirectly.

**Let's** look at an example to understand what that means in practice. Here we have a bunch of process and a bunch of resources and a couple of those resources are assigned to processes. So what happens if Process 1 wants to acquire Resource 1?

**Not** much, Process 2 is not waiting for anyone so Process 1 has to wait for Resource 1. Different than in Havender's approach though, Process 1 keeps all its resources. Ok, so what happens when Process 3 wants to acquire Resource 2.

**Again** not much, because Process 1, who is holding Resource 2 is waiting for Resource 1 which is held by Process 2 which is not waiting for anyone. Let's see what happens if Process 2 tries to acquire Resource 3?

**That's** an interesting one, because Resource 3 is held by Process 3 who is actually indirectly waiting for Process 2. So Process 2 can actually acquire Resource 3. In fact in this example it's pretty easy to see that the system would otherwise deadlock. Unfortunately, there's one small complication namely, Process 2 has to release Resource 3 before it can release Resource 1, because otherwise it wouldn't be permitted to hold Resource 3 any longer. There are more interesting graph theory aspects here but sadly time does not permit to dive into this. So this idea was the basis for the project and I then moved on to implement it.

**5:50 [TAP PHONE]**

**In** particular, I wrote a C library which is based on the POSIX Standard, in other words on pthreads. An important aspect was to keep the API really simple and straightforward to use and I think I did a decent job there. In essence there are four function calls in the API, namely, create a lock, acquire a lock, release a lock and delete a lock. That means all the logic is happening behind the scenes and it's all nicely hidden away.

**It** was difficult to find benchmark tests for testing locking protocols, so I ended up writing my own benchmark tests. I tested mostly for speed and whether the protocol is prone to deadlock or starvation. The results show that inheritance locks are a bit slower than pthread and C++ mutexes, however, they can be faster than transactional memory as you can see here. Moreover, they do not suffer from deadlock or starvation.

**Finally**, I also tested the locking protocol with the SPIN model checker against deadlock and starvation. I am aware that most of you probably have not used SPIN before so just briefly: SPIN takes a model of a concurrent system and checks a number of definable properties against it. SPIN checks all potential routes of the system.

*So* in principle this is ideal for checking a locking protocol, but as you can imagine the workload of checking all the different execution routes of such a concurrent system becomes intractable. Therefore we decided to run the verification on a bunch of lab machines over the Christmas Holidays. Unfortunately, it all took a bit longer than anticipated and required a lot of memory. So after running the verification for over 10 days I received this email from lab support.

**So** that didn't go down too well and I had to abort the verification at that point. The good news is, until then no errors were found which is at least a pretty promising result.

**Thanks** for listening, Questions?