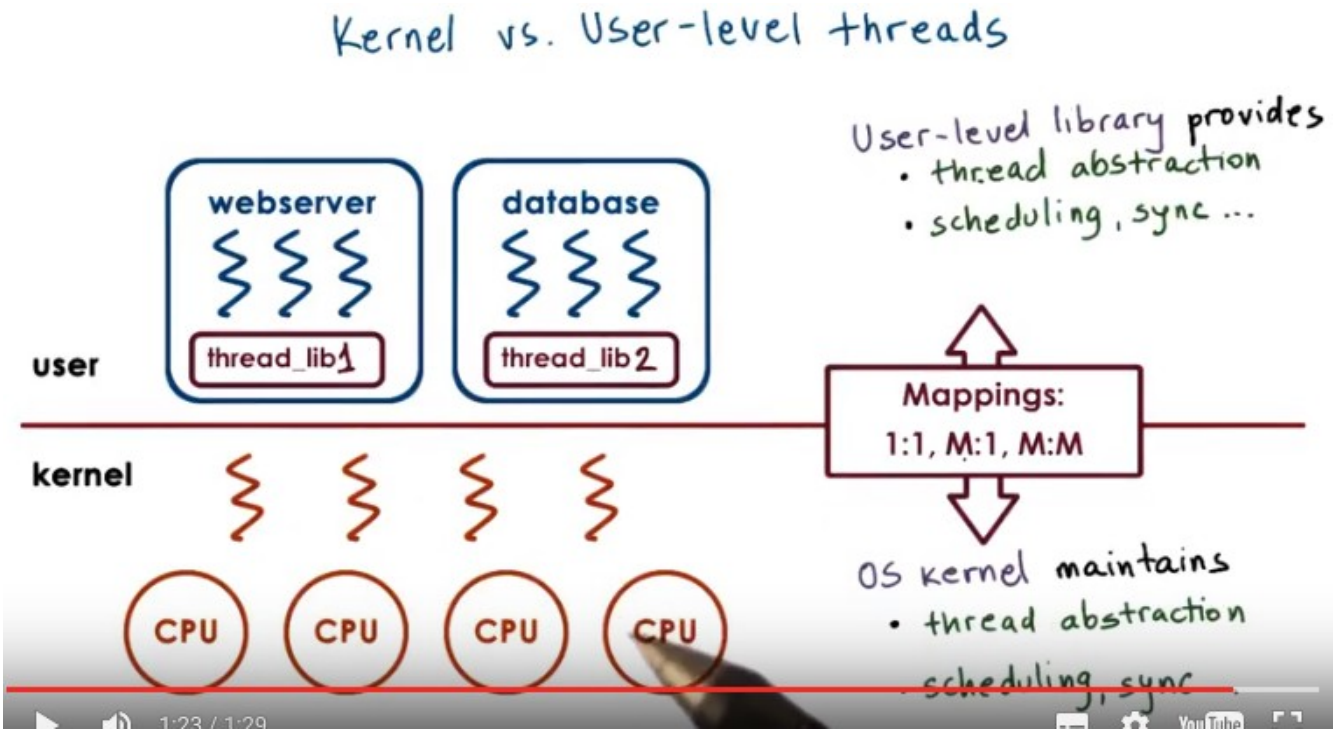
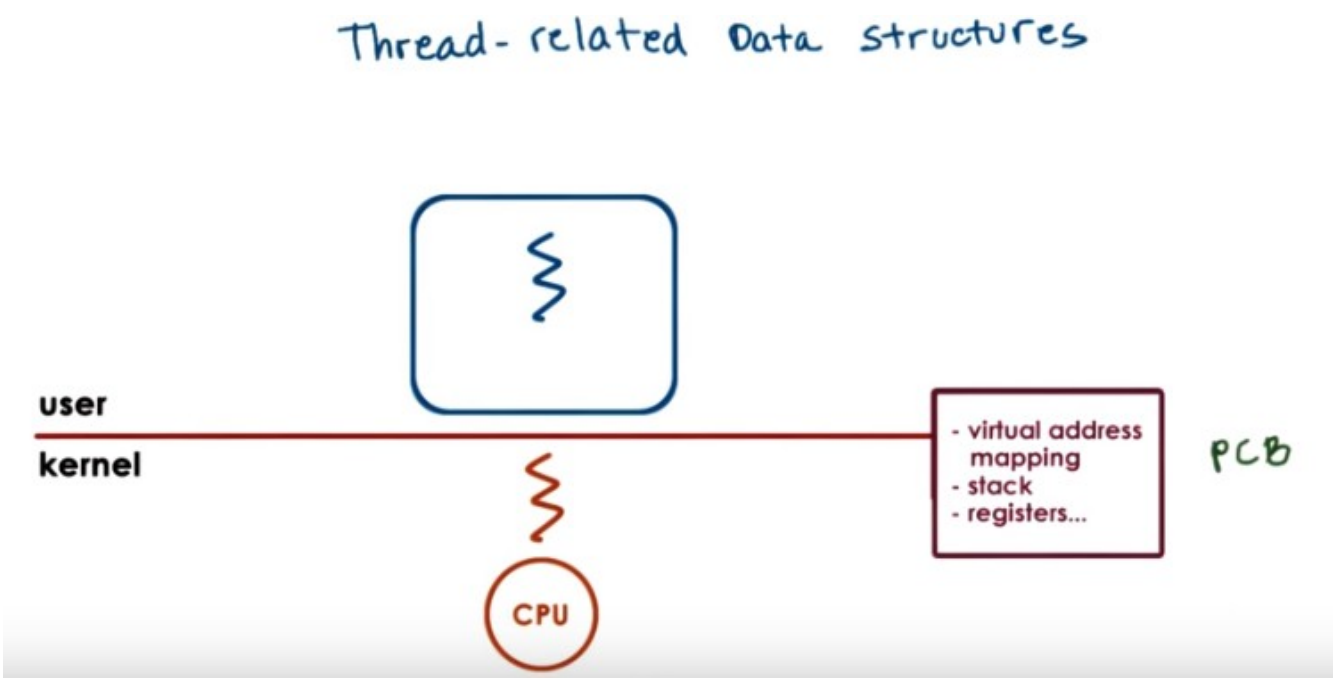
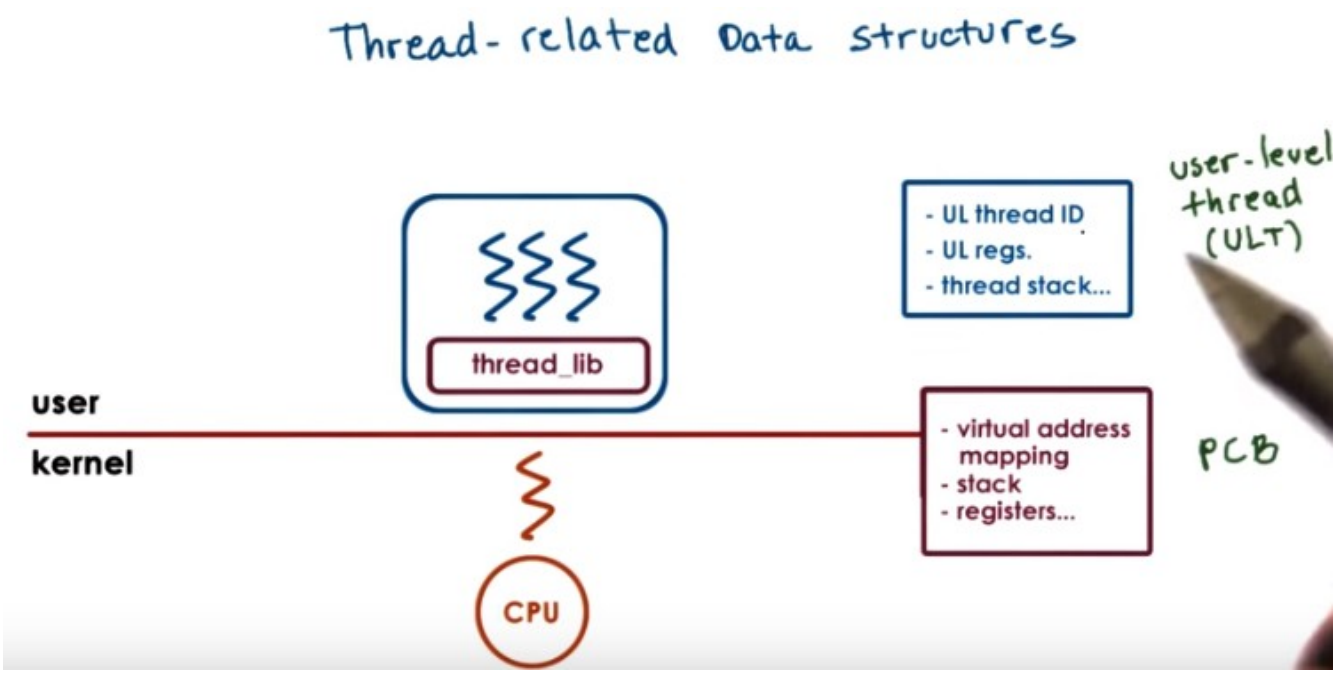
1. In our introductory lecture on threads and concurrency, we talked about the fact that threads can be implemented at the kernel level, at the user level, or both. In this lesson we will re-visit that statement and see what it is that is necessary in terms of data structures and mechanisms from the operating system. In order for us to be able to implement threads at both the kernel and the user level. During this discussion, we will also look at two notification mechanisms that are supported by OSs. And that includes interrupts and signals. To make the discussion in this lecture more concrete, we will use two older papers, the Eykholt paper Beyond Multiprocessing: Multithreading the Sun OS Kernel and the Stein and Shah paper on implementing lightweight threads. These are older papers, but they provide us with some historic information on how threading systems evolved over time. There are links to both of these papers in the Instructor's Notes. We will end this lecture with a brief summary of the current threading model in the Linux operating system.



2. Let's start by revisiting the illustration we used in the threads and concurrency lecture. There we explained that threads can be supported at user level, at kernel level, or both. Supporting threads at the kernel level means, that the OS kernel itself is multithreaded. To do this, the OS kernel maintains some abstraction, for our threads of data structure to represent threads, and it performs all of the operations like synchronizations, scheduling, et cetera, in order to allow these threads to share the physical resources. Supporting threads at the user level means that there is a user-level library, that is linked with the application, and this library provides all of the management in the runtime support for threads. It will support a data structure that's needed to implement the thread abstraction and provide all the scheduling synchronization and other mechanisms, that are needed to make resource management decisions for these threads. In fact, different processes may use entirely different user-level libraries that have different ways to represent threads that support the different scheduling mechanisms, et cetera. We also discussed several mechanisms, how user-level threads can be mapped onto the underlying kernel level-threads, and we said these include a one-to-one, many-to-one, and a many-to-many mapping, and briefly touched upon some of the pros and cons of each approach. Now, we'll take a more detailed look at about, what exactly is needed to describe kernel versus user-level threads, and to support all of these types of models.

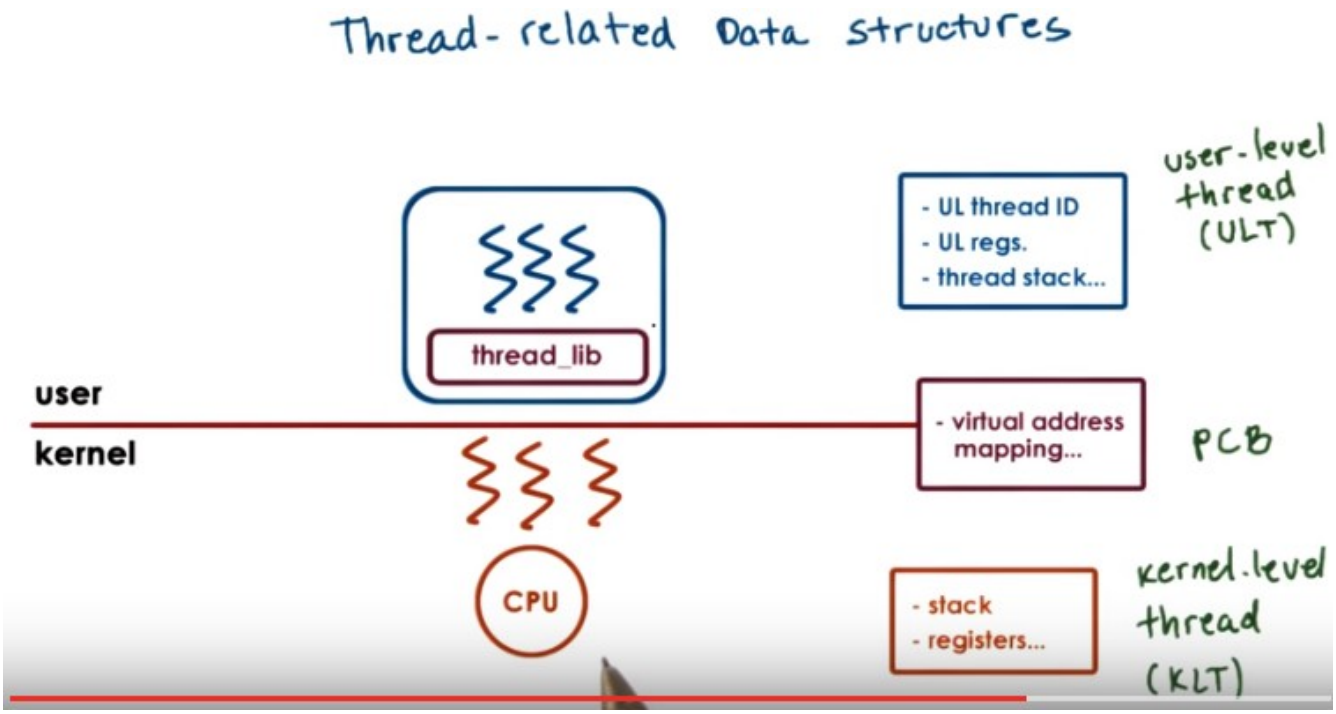


3. Let's start by looking at what happens in a single threaded process. The process is described with all of the process-related state, that includes the address space or the virtual to physical address mappings, its stack, its register since it has a single thread. Whenever this process makes a system call, it tracks into the kernel, executes in the context of a kernel thread. All of the information about the state of this process we said is contained in its process control block. And let's, for now, assume that we're dealing with just one CPU.

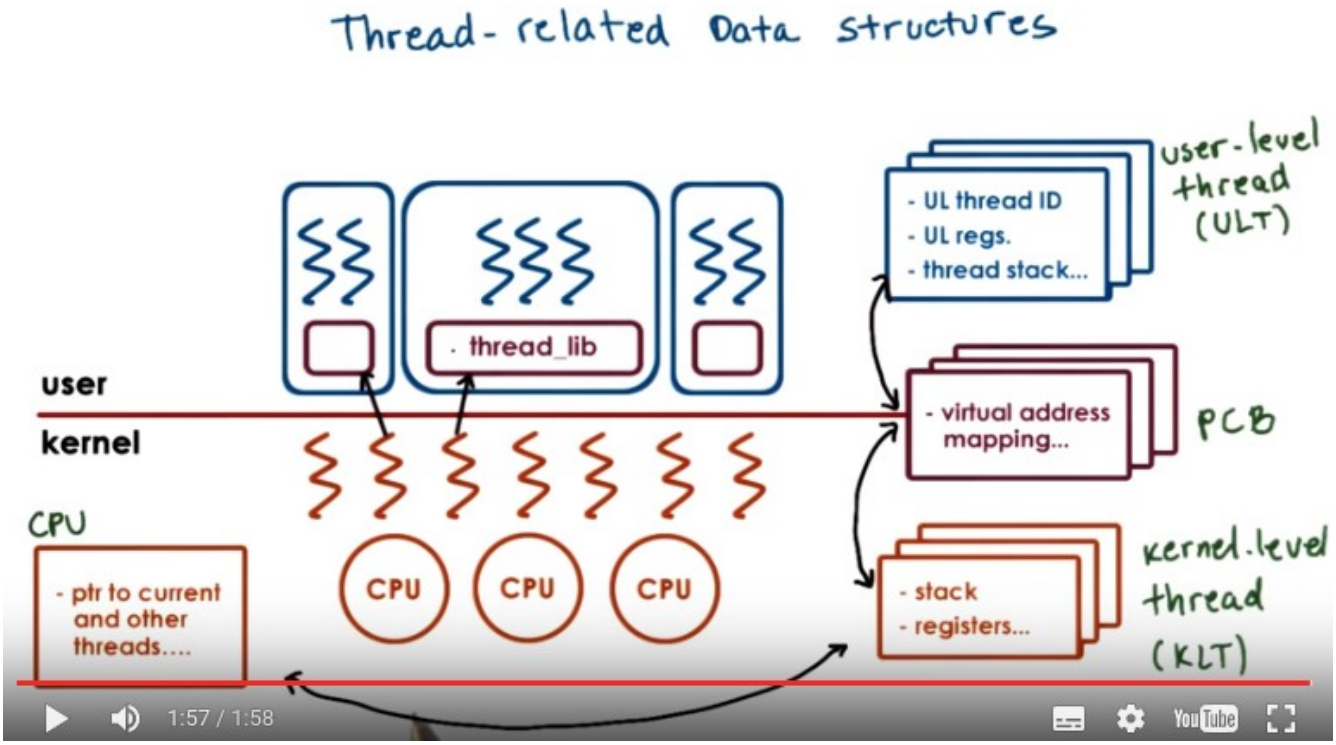


Let's now make the process multithreaded, so this will look like our many-to-one model, where many user-level threads are supported by one kernel level thread, and that there is a user level threading library that manages these threads. This user-level library will need some way to represent threads so

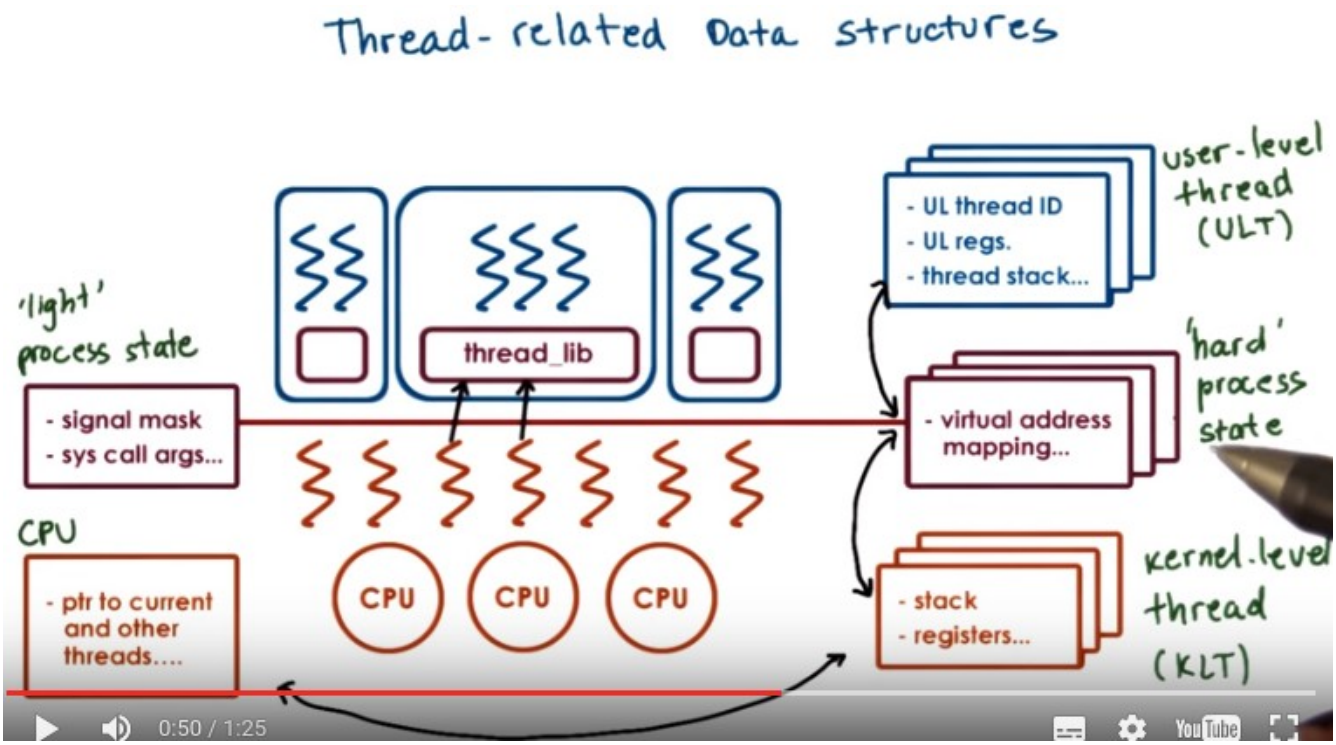
that it can track their resource use and make decisions regarding scheduling and synchronization. So it will have some user-level thread data structure.



If we wanted it to be multiple kernel level threads associated with this process, we see that currently we have the process control block by containing the, all the virtual address mappings, as well as the execution state of the process thread. We don't want to have to replicate this entire data structure with all this information, just so as to represent different stack and register values for the kernel-level entities. So we will start splitting up this process control block structure to separate the information that's specifically useful to represent the execution state of the kernel-level threads, so their stack and register pointer. And this will still have, the process control block will still have the virtual address mappings, as well as some additional information that's relevant for the entire process, so for all of the kernel-level threads. Note that from the perspective of the user-level threading library, the underlying kernel-level threads look sort of like virtual CPUs. So the threading library looks at the user-level threads and decides which one of the user-level threads will be scheduled onto the underlying kernel-level threads. Unix-based systems, for instance, have certain operations called set jump and long jump which are useful when we need to save and restore the context of the user-level thread.

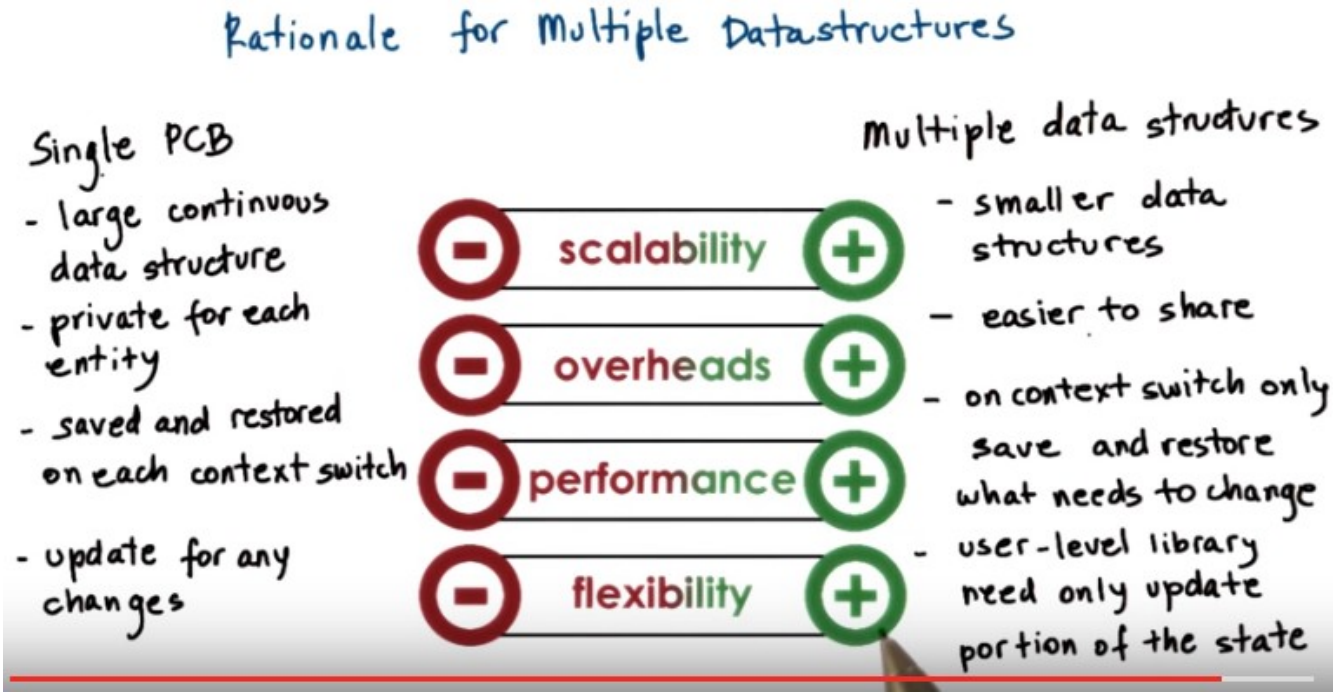


4 (沒完全理解, 先看第 6 節的圖 is helpful). Now, let's say we have multiple such processes. Clearly we will need copies of these user-level thread structures, process control blocks, and kernel-level thread structures to represent every single aspect of every one of these processes. So, we'll need to start maintaining certain relationships among them. With respect to the user-level threads(某一個 process 的 threads) within the threading library, it keeps track of all of the user-level threads that represent the single process. So, there is a relationship between them and the process control block that represents that address space. For each process we need to keep track of what are the kernel-level threads that execute on behalf of this process, and vice versa, for each kernel-level thread we have to make sure we know what is the address space within which that thread executes. If the system has multiple CPUs, we need to have a data structure to represent the CPU. And then we need to maintain a relationship between the kernel-level threads and the CPU. So what is the CPU that a kernel level thread has affinity to, last strand it was scheduled on, and then for a CPU, a pointer to its current thread or a pointer to the threads that typically run there, and similar information. When the kernel itself is multithreaded, we said we can have multiple kernel-level threads supporting a single user-level process. When the kernel needs to schedule, or context switch, among kernel-level threads that belong to different processes, it can quickly determine that they point to a different process control block. So they will have different virtual address mappings, and therefore can easily decide that it needs to completely invalidate the existing address mappings and restore new ones. In the process, it will save the entire process control block structure of the first kernel-level thread, and then if it's context switching to the second one, it will restore the entire process control block structure of the second one.



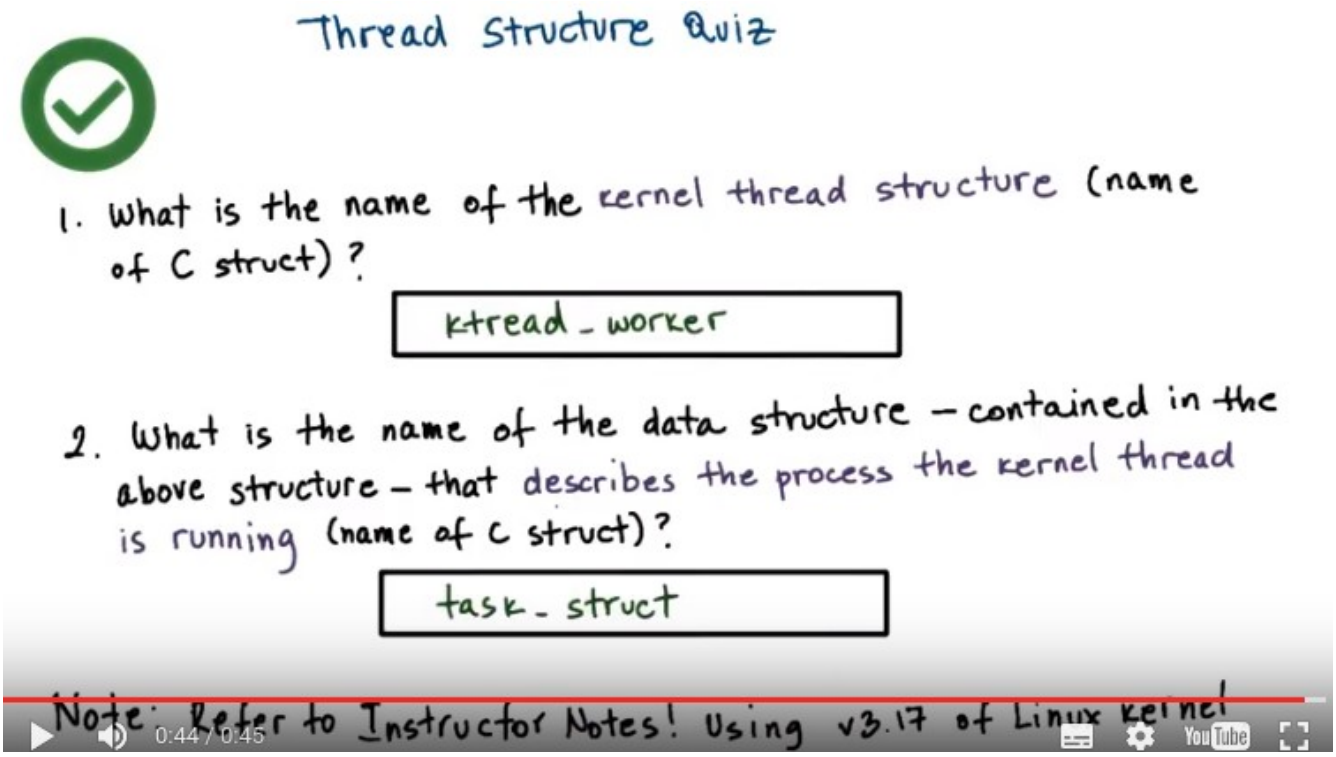
注意上圖中 'light' process state 和 'hard' process state 的對比

5 (沒完全理解，先看第 6 節的圖 is helpful). Now when two kernel level threads belong to the same address space there's actually information in the process control blog that is relevant for the entire process. However there is information here that is specific to just one kernel level thread. For instance this includes information about signals or system call arguments. So when we're contact switching among these two kernel level threads, there is a portion of this process control block information that we want to preserve, like all of the virtual address mappings. But then there is portion of it, that's really specific to the particular kernel level thread and it depends on what is the user level thread that's currently executing. So it's something that the threading library directly impacts. So we will split up the information that was contained originally in the process control block, and we will separate it into the hard process state that's relevant for all of the user level threads that execute within that process. And then, the more light weight process state that is only relevant for a subset of the user level threads that are currently associated with a particular kernel-level thread. So we started off with one large contiguous process control block structure, and then we divided the information that was contained in it across a number of different data structures.



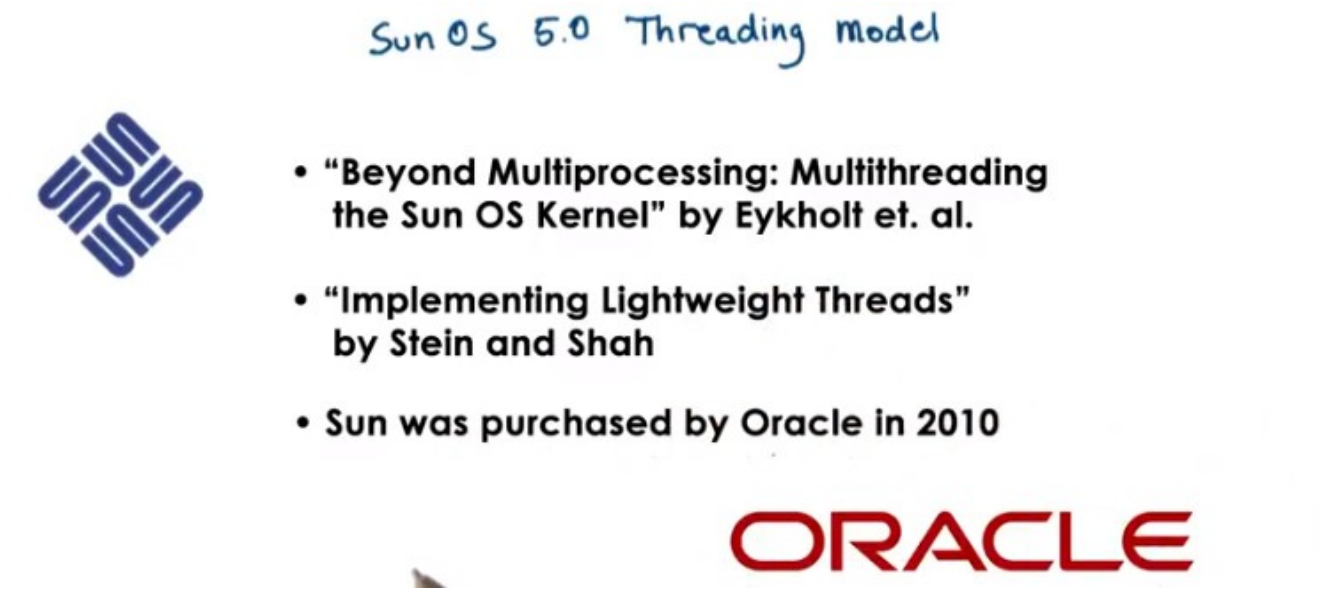
6. When we had just one single process control block, it was a large contiguous data structure. We have to maintain separate copies for every single thread, even though they may share some information. Whenever we need to context switch we need to save and restore this entire data structure, and it is large we said. And finally, it's just this one data structure that's used for so many different operations. For scheduling, for memory management, for synchronization. If we want to customize any aspect of that, we are potentially going to affect multiple OS services, and so it makes updates a little bit challenges. So all in all, there are multiple limitations to this approach of using a single process control block structure to represent all aspects of the execution state of a process. Scalability is limited due to the size. Overheads are limited because they need to have private copies. Performance is affected because everything has to be saved and restored. And then flexibility is affected by the fact that updates are a little bit more difficult. In contrast, when we have multiple data structures, we actually end up with mu, multiple small data structures. The information that was contained in the original process control block, is now maintained via pointers by pointing to much smaller data elements. Then it becomes easy to share portions of that information. We will point to the same data structure for those components of the state, which are identical across threads or processes. And we will create new elements when we need to have different information. On a context switch, only that portion of the state that actually needs to change will be saved and restored. And then both, any kinds of modifications will impact only subset of the data elements. And then the interactions between the user-level library and the system will also be carried out through a much smaller more, more confined interfaces. All in all, this trend to use multiple data structure leads to improvements across the board. We gain on scalability, on overheads because we don't have to have separate copies for, for everyone. We have improvements in performance because context which time can be reduced, and we have more flexibility. As a result, operating systems today typically adapt this type of approach for organizing information about their execution contexts.

7. Now that we have discussed how thread structures are separated let's take a look at an actual Linux kernel implementation in this quiz. For each of the questions in this quiz we will be referencing version 3.17 of the Linux kernel. The first one is, what is the name of the kernel thread structure that's used in Linux? We're looking for the name of a C structure, basically. The second question is, what is the name of the data structure, that's actually contained in the above data structure, that describes the process that the kernel thread is running? Again, we're looking for a name of a C structure. Provide your answers in these text boxes and refer to the instructor notes that reference the 3.17 version of the Linux kernel.

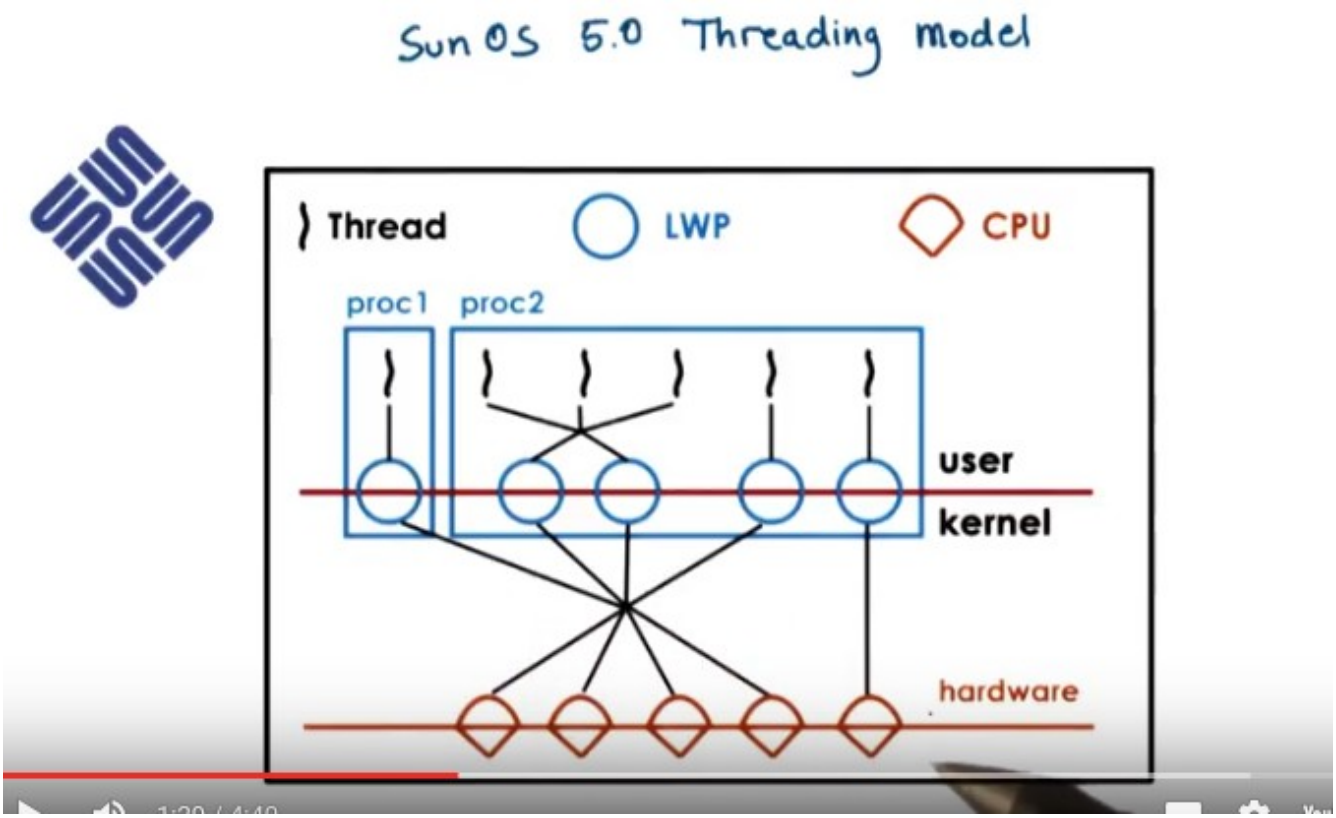


以上的 ktread_worker 是 typo, 應為 kthread_worker. task_struct 在本 note 最後的 Linux 中會講.

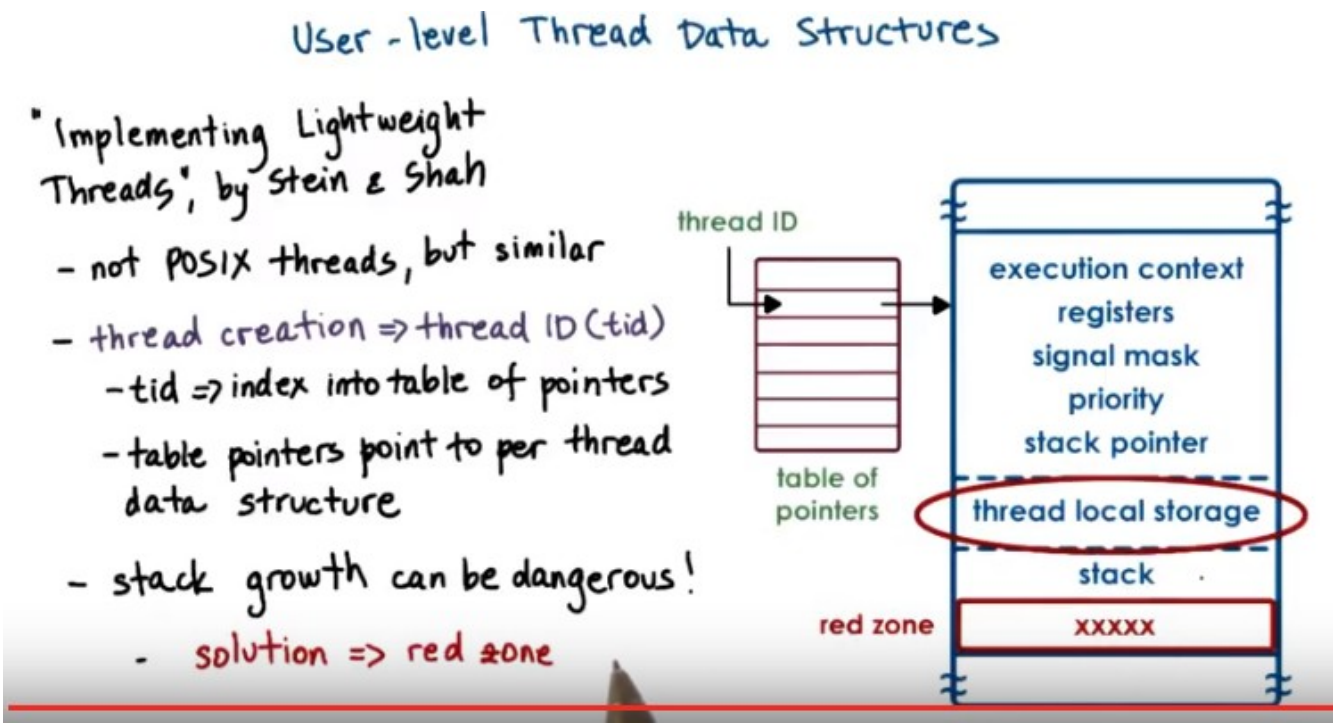
8. If you browse the kthread.H header file, you will see in line 66 that there is a structure ktread_worker. This data structure, as well as the various functions that are defined in this file, provide a simple interface for creating and stopping kernel threads. You can see that within the kthread_worker data structure, there are four members. The stem lock data structure is definitely not the one that's used to describe a process, nor is the list head that points to a list of kthread_workers. If you click on the next one, task_struct, you will see that it's a holding place for tons of important information regarding a process. So our answer now is at task_struct.



9. Let's look now at the data structures that are described in the two reference papers of this lesson. The two papers describe the kernel and user-level implementations of threads in the SunOS 5.0 kernel of Solaris 2.0. So Solaris is the operating system. Sun, where this work was done, no longer exists; it was bought by Oracle in 2010. But it was very well known for the quality and stability of its UNIX distributions. 別人說的 Unix 內核是不是就是這裡說的這個 kernel? It was also one of the leader in introducing new and revolutionary features into the kernel. And this is why we are looking at its threading model.



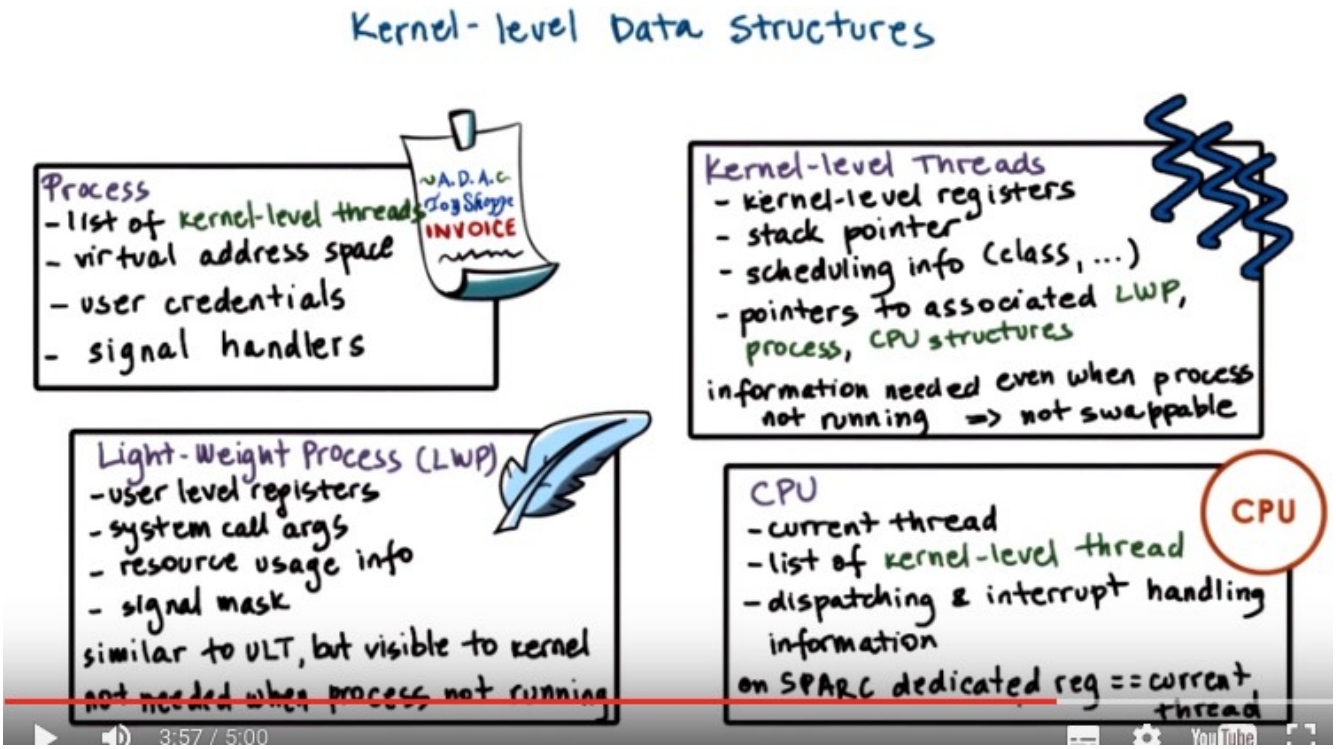
This is a diagram from figure one in the Stein and Shah paper, Implementing Lightweight Threads. And it illustrates quickly the threading model supported in the operating system. Going from the bottom up, the OS is intended for multi-processor systems, with multiple CPUs and the kernel itself is multi-threaded. There are multiple kernel-level threads. At user level, the processes can be single or multithreaded. Both many-to-many as well as one-to-one mappings are supported. Each kernel-level thread that's executing a user-level thread, has a lightweight process data structure associated with it. From the user-level libraries perspective, these lightweight processes represent the virtual CPUs onto which it's going to be scheduling the user-level threads. At the kernel level, there will be a kernel-level scheduler that will be managing the kernel-level threads and scheduling them onto the physical CPUs.



上圖最右邊的大表 即 user-level thread data structure, 它左邊的小表即 table of pointers, 其中每個 pointer 都指向一個 user-level thread data structure, 而 thread ID 就是 table of pointers 這個數組中的指標

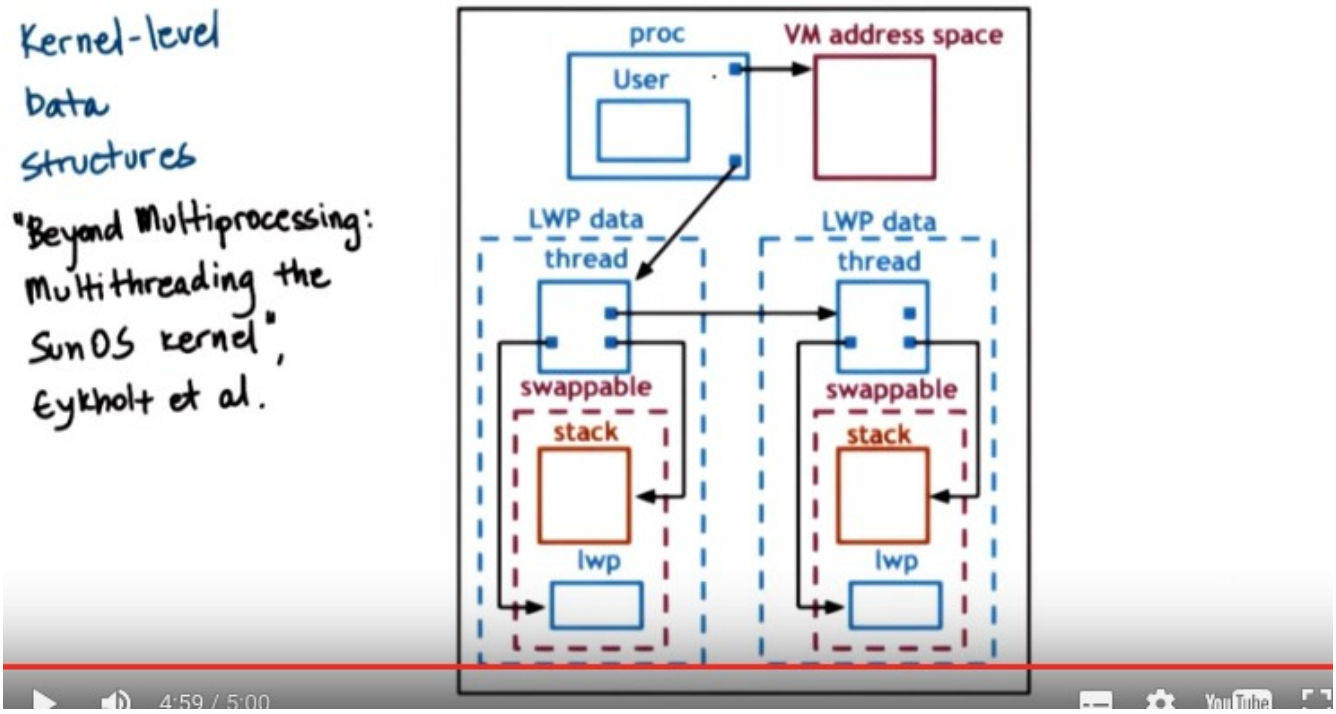
We will now look a little more closely at the user-level thread data structures. They are described in the implementing lightweight threads paper by Stein & Shah. This does not describe pthreads, the POSIX threads, but it's a similar type of user-level threading library. When a thread is created, the library returns a thread ID. And this is not a direct pointer to the actual thread data structure like we've implied before. Instead, it's an index in a table of pointers. It is the table pointers that in turn point to the actual thread data structure. The nice thing about this is that if there is a problem with the thread, if the thread ID were a pointer, then that pointer would just point to some corrupt memory. And we can't really figure out what's going on. Whereas here, by having the thread ID index into a table entry, we can encode some information into the table entry that can provide some meaningful feedback or an error message. The thread data structure, we said, contains a number of fields, registers, signal mask, priority. There's also the stack pointer, of course, that points to the stack, and then there is the thread local storage area. This area, this includes the, variables that are defined in the thread functions that are known at compile time, so the compiler can allocate private storage on a per-thread basis for each of them. The stack itself, its size, it may be defined based on some library defaults or the user can provide a stack. But basically the size of a lot of this information, is known up front at compile time, so we can create these thread data structures and sort of layer them in a continuous way, and that can help us achieve locality. It can make it easy for the scheduler to find the next thread. It just has to basically multiply the thread integs with the size of the data structure. The problem however is that the threading library doesn't really control the stack growth, so it doesn't in, inject itself between any kind of update and what gets written on the stack. And then the operating system itself, it doesn't know that there are multiple user-level threads. So it's possible that as the stack is growing, that one thread will end up overwriting the data structure of another thread. If this happens, the tricky part is that the error, the problem will be detected when that other thread gets to run. However, the cause of the problem is a

completely different thread. So, so this makes debugging a little bit tricky. The solution that was introduced in this paper was to separate the information about different threads with a so-called red zone. This really refers to a portion of the virtual address space that's not allocated. So if a thread, it's running, and its stack is increasing, if it tries to write to an address that basically falls in this red zone region, then the operating system will cause a fault. Now it's however much easier to reason about what happened because the fault, the problem, was directly caused by the thread that was executing. So it's easier to do root cause analysis and to fix the problem.



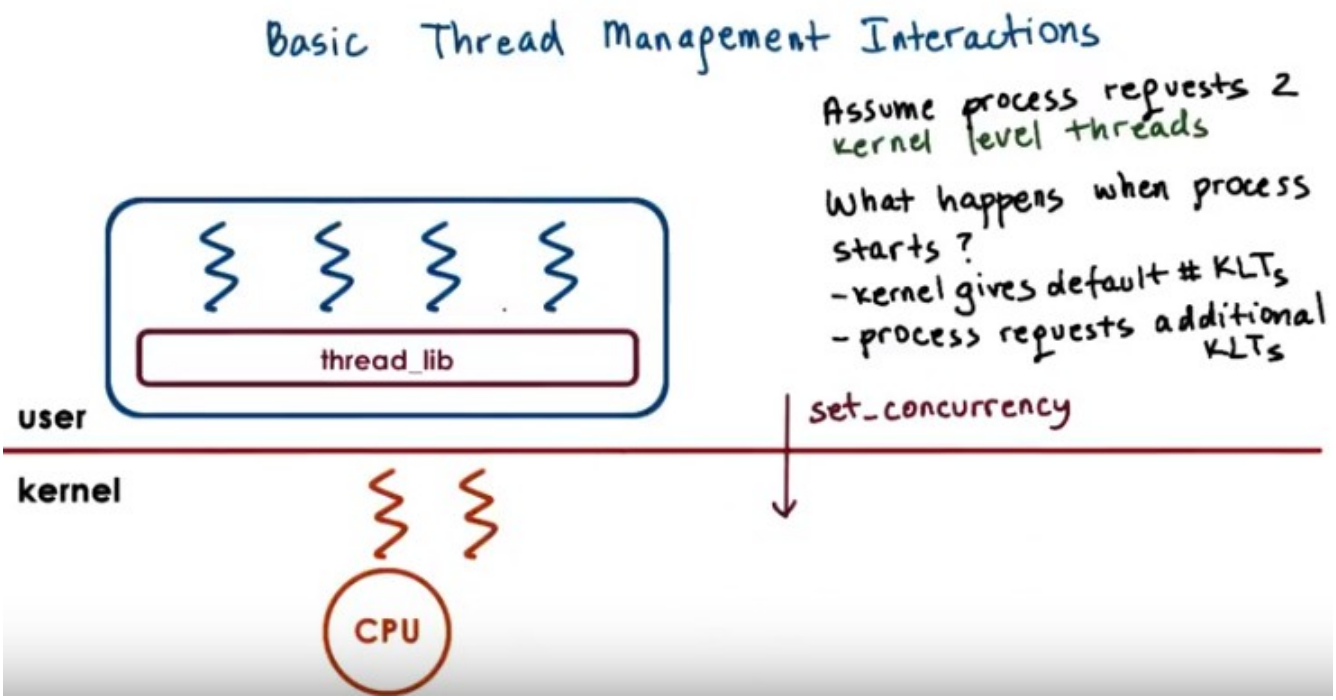
10. Let's move now to the kernel level data structures. First for each process we maintain information about that process. What are all the kernel level threads that execute within that process address space? So what are the mappings that are valid between the virtual and physical memory? What are the user credentials? For instance, if this process is trying to access a file, we have to make sure that that particular user has access to that file. And then, information like, what are the signal handlers that are valid for this process. We'll talk about this a little bit later, but for now, know that this is information about how to respond to certain events that can occur in the operating system. Next, we have the lightweight process data structure and this contains information for a sub subset of the process. For instance, it can have information that's relevant to one or more of the user level threads that are executing in the context of the process. And keep track of their user-level registers and the system call arguments. The information that's maintained in a light-weight process data structure is in some ways similar to what we maintain at the user level in the user-level thread data structure. But this is what's visible to the kernel, so when the OS-level schedulers need to make scheduling decisions they can see this information and act upon it. Also note that we track resource usage information in this data structure. At the operating system level, the kernel tracks resource uses on a per kernel thread basis. And this is maintained in the data structure for the lightweight process that corresponds to that kernel level thread. So if we want to find out the aggregate resource usage for the entire process, we need to basically walk through all of the lightweight processes that are associated with it. The kernel level data structure includes the kernel level information, like registers, stack pointers, scheduling class. And it also has pointers to the various data structures that are associated with this kernel. So what is the

lightweight process? What is the actual address space? What is the CPU where this is running? One thing to note about these two data structures is that the kernel-level thread structure, it has information about a kernel-level thread, about an execution context, that is always needed. They're operating system level services that need to access some information even when a thread is not active. Like, for instance, scheduling information if they need to decide whether they need to activate that thread. So this information is basically not swappable. It always has to be present in memory. Whereas in contrast the light weight process data structure, the information that it maintains does not always have to be present in memory so. If we're running under memory pressure, it is possible to swap out this content. This also potentially allows the system to support larger number of threads in a smaller memory footprint than what would've been the case if everything needed to be constantly memory. Next is the CPU data structure. It has information like the current thread that's currently scheduled, list of the other kernel level threads that ran there. Some information how to actually execute the procedure for dispatching a thread, or how to respond to various interrupts on the referral devices. Note that if we have information about the CP and a given CPU once we know the current thread through it we can find that information about all of the different data structures that are needed to rebuild the entire process state. On the SPARC architecture that is used in the Solaris papers, there are extra registers, so there are lots of registers. And the implementation is such that there is one dedicated register that is used to point to the current thread at any given point of time. So you're in context which this register is updated. But what it implies is that it's easy to just access that register and then immediately be able to start tracking through these pointers to find the right information. That's in contrast to perhaps having to access memory to read the CPU structure to then read the current thread information, et cetera.

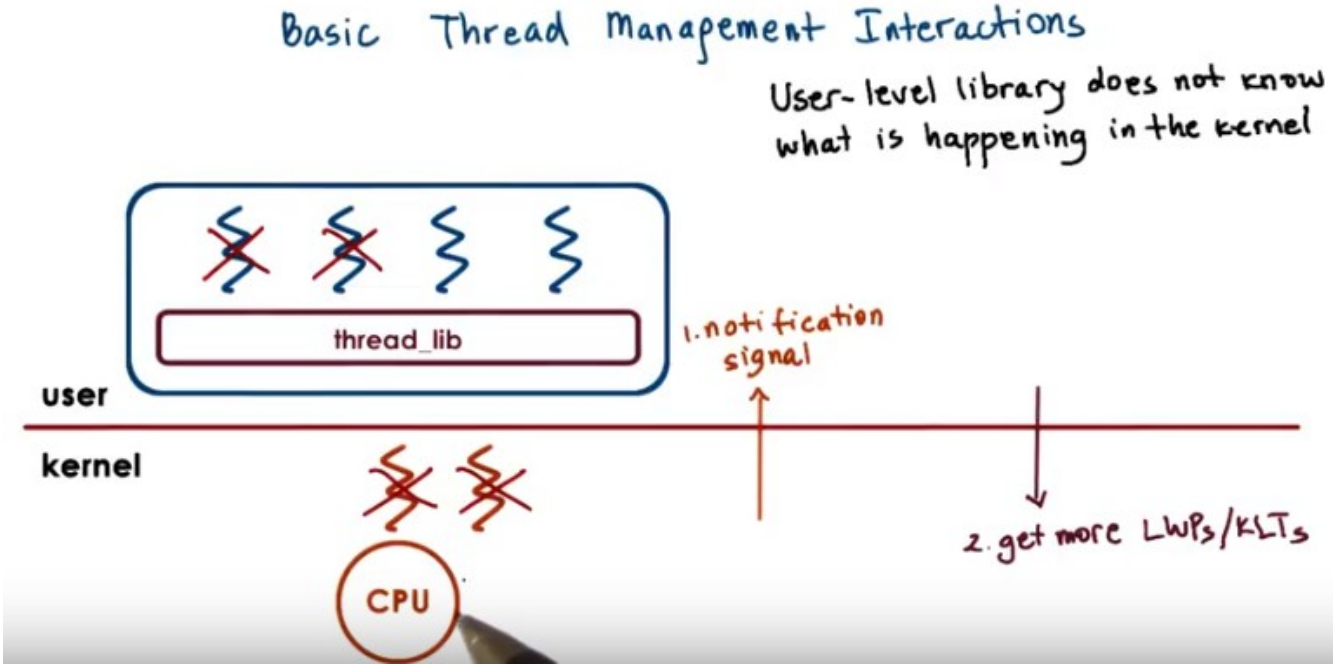


Here's how the Eykholt paper on multithreading the SunOS kernel describes the relationship between all of these data structures. This is figure two in this paper. A process data structure has information about the user, for instance the address space, and then points to a list of kernel-level thread structures. Each of the kernel-level thread structures points to the likely process that it corresponds to, to it's stack, and to other information. The lightweight processing stack. This portion of the state is actually

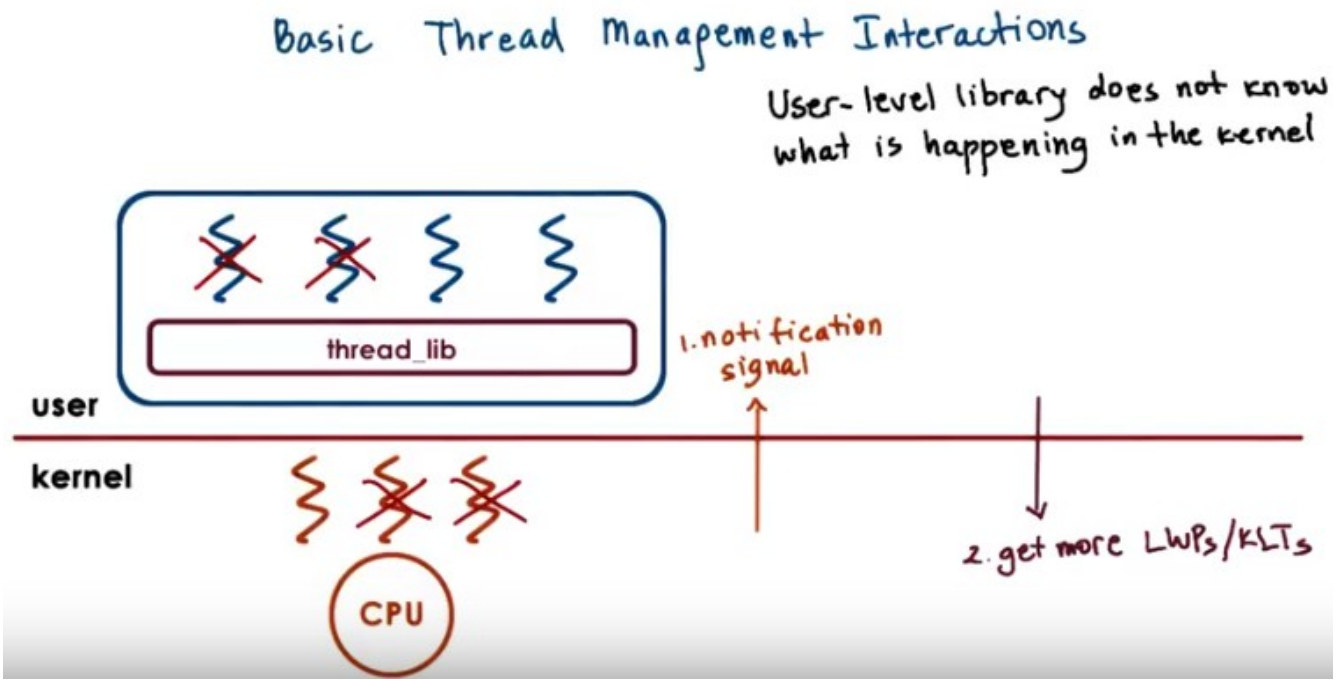
swappable. What's not shown in this figure that was showed in the previous image is any information about the CPU. And there is some other information, some other pointers that are not shown here so as not to clutter everything, like from the thread going back to the process, et cetera.



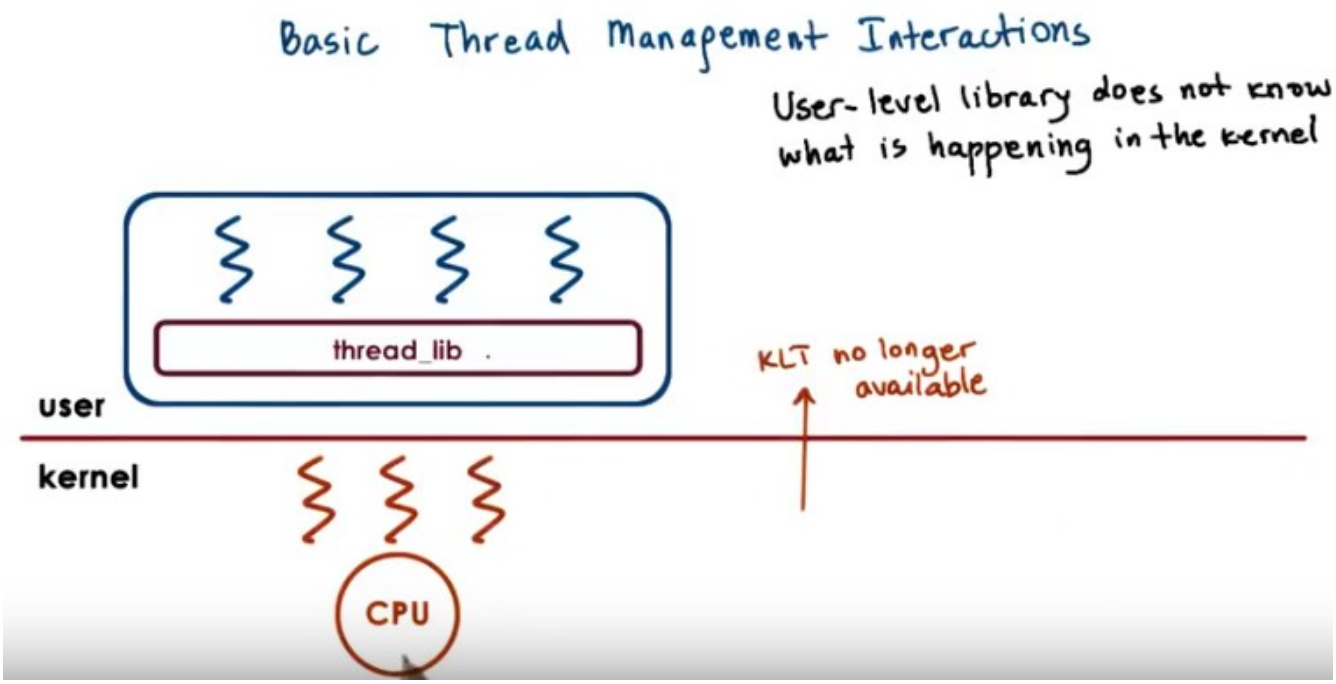
11. So we have threads at the user level, we have threads at the kernel level. We will now see what are some of the interactions that are necessary in order to efficiently manage threads. Consider we have a multithreaded process. And let's say that process has four user-level threads. However, the process is such that, at any given point of time, the actual level of concurrency is just two. Basically, if you look at the process, it always happens that two of its user-level threads are waiting on I/O, and then some other two are actually executing. So, if our operating system has a limit on the number of kernel threads that it can support, it would be nice if the user-level process actually said, I just really need two threads. So when the process starts, the kernel will first give it, let's say, a default number of kernel-level threads and the accompanying lightweight threads. And let's say that is one. Then the process will request additional kernel-level threads, and the way it's done is that the kernel now supports a system call called set_concurrency. In response to this system call the kernel will create additional threads and it will allocate those to this process.



Now lets consider this scenario in which the two user-level threads that were mapped on the underlying kernel-level threads block. They needed to perform some I/O operation and then they were basically moved on the wait queue that's associated with that particular I/O event. So the kernel level threads are blocked as well (因為那兩個被 block 的 user-level threads 是 map 到這兩個 kernel-level threads 的). Now let's say we have a situation in which the two user-level threads that were running on kernel level threads issued an I/O request, and now have to wait for that to complete. So it's a blocking I/O. What that means is that the kernel-level threads themselves, they're also blocked on that I/O operation. They're waiting in a queue somewhere in the kernel for that I/O event to occur. Now we have a situation where the process as a whole is blocked, because it only had two kernel-level threads, both of them are blocked, and there are user-level threads (另兩個沒被 block 的 user-level threads) that are ready to run and make progress. The reason why this is happening is because the user-level library doesn't know what is happening in the kernel, it doesn't know that the kernel threads are about to block. What would have really been useful is if the kernel had notified the user-level library before it blocks the kernel-level threads. And then the user-level library can look at its run queue, it can see that it has multiple runnable user-level threads, and, in response, can let the kernel know, so, call a system call to request more kernel-level threads or lightweight processes.

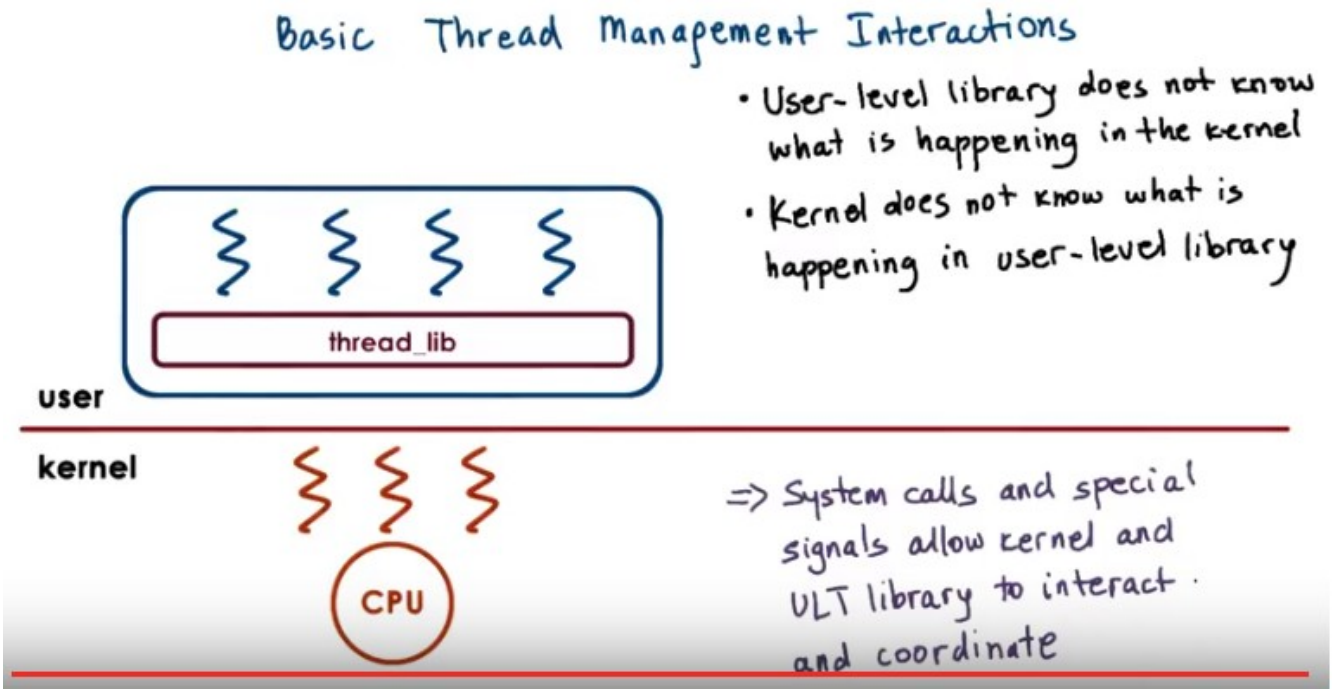


Now in response to this call, the kernel can allocate an extra kernel-level thread, and the library can start scheduling the remaining user-level threads onto the associated lightweight process.



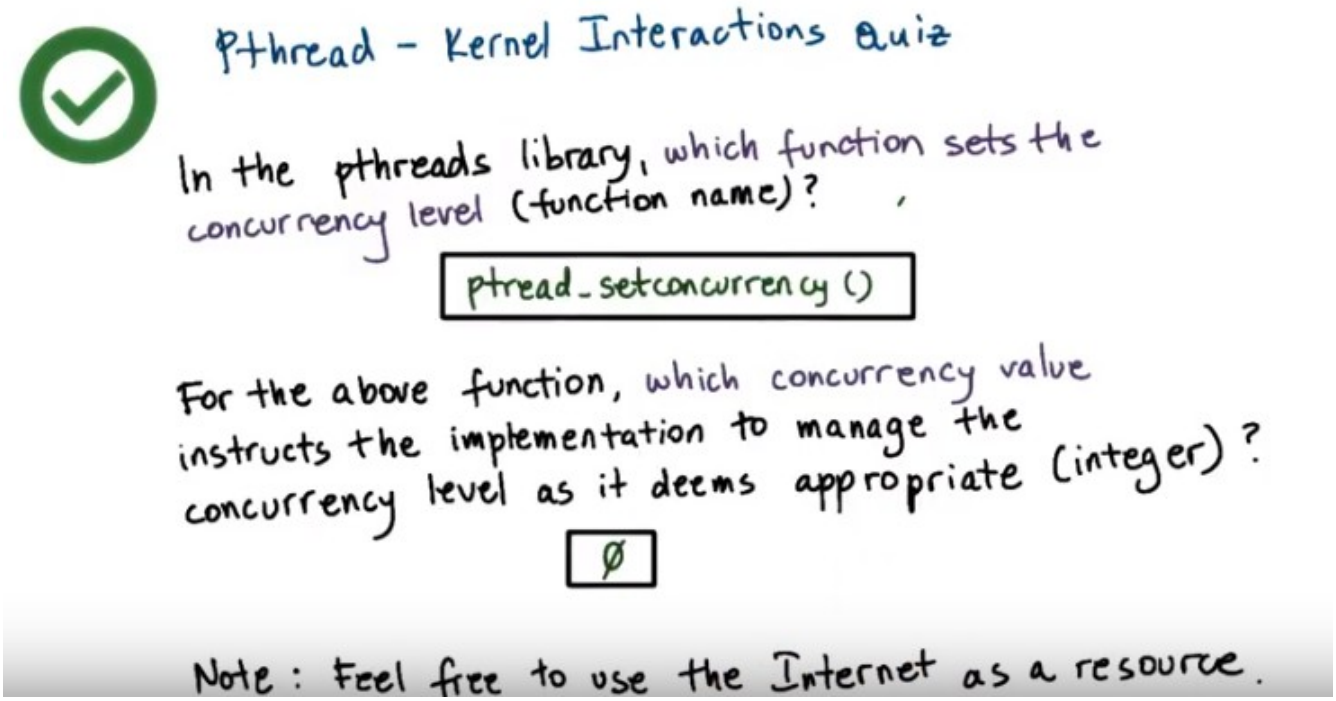
At a later time when the I/O operation completes, at some point the kernel will notice that one of the kernel-level threads is pretty much constantly idle (即不再需要這個 kernel-level thread 了), because we

said that that's the natural state of this particular application. So maybe the kernel can tell the kernel-level library that, you no longer have access to this kernel-level thread, so you can't schedule on it.



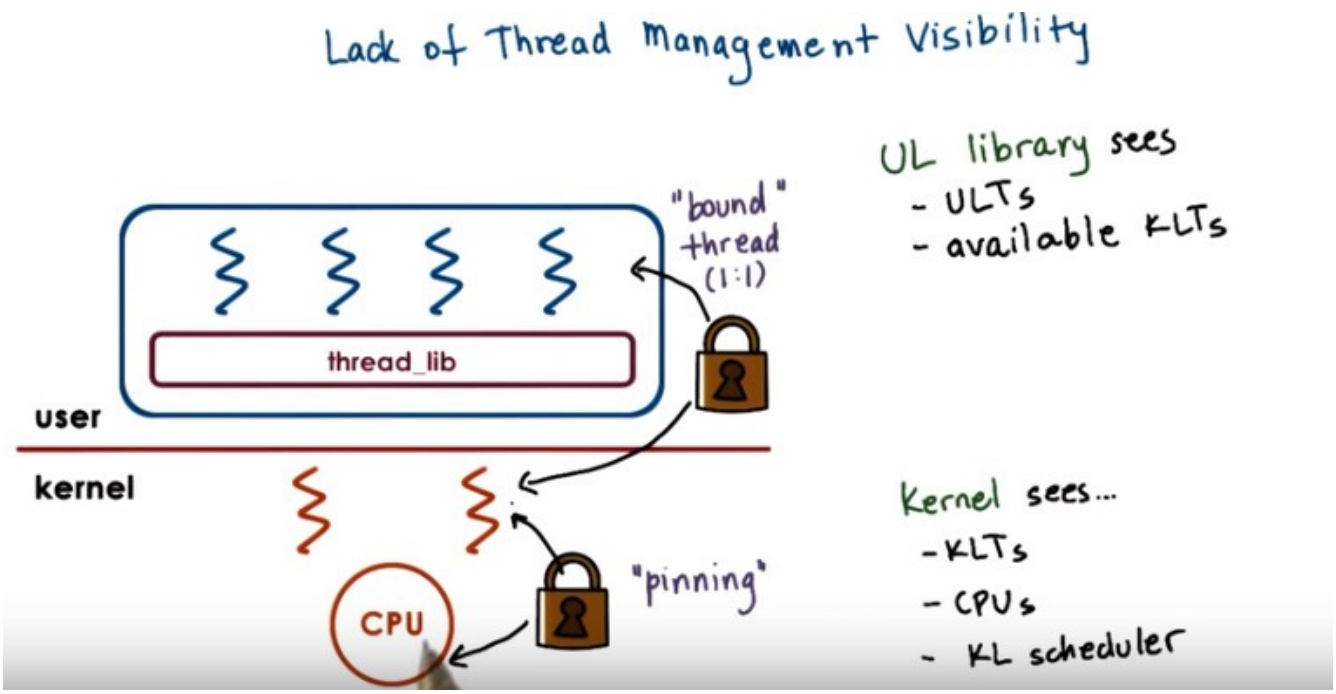
By going through these examples you realize that both the user-level library doesn't know what's happening in the kernel, but also the kernel doesn't know what's happening at the user level. Both of these fact cause for some problems. To correct for these issues, we saw how in the Solaris threading implementation, they introduced certain system calls and special signals that can be used to pass or request certain things among these two layers. And basically this is how the kernel-level and the user-level thread management interact and coordinate.

12. Let's take a quiz and look at an example of how the pthreads threading library can interact with a kernel to manage the level of concurrency that a process gets. The first question is, in the pthreads library, which function sets the concurrency level? We're looking for a function name here. For the second question, given the above function, what is the concurrency value that instructs the underlying implementation to manage concurrency as it finds appropriate? And we're looking for an integer value here. And please feel free to use the Internet as a resource to understand the answer to this question.

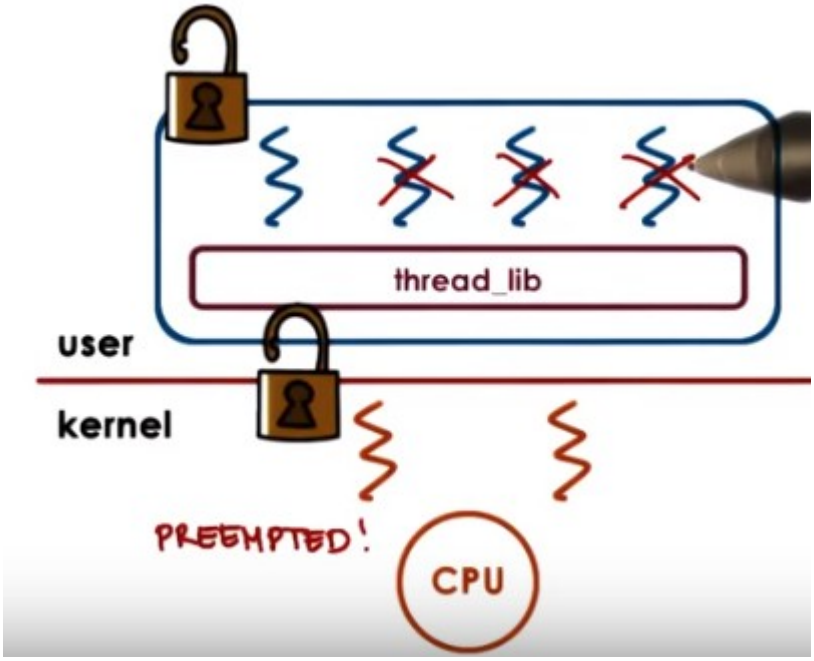


上圖中的 ptread_setconcurrency()應為 pthread_setconcurrency().

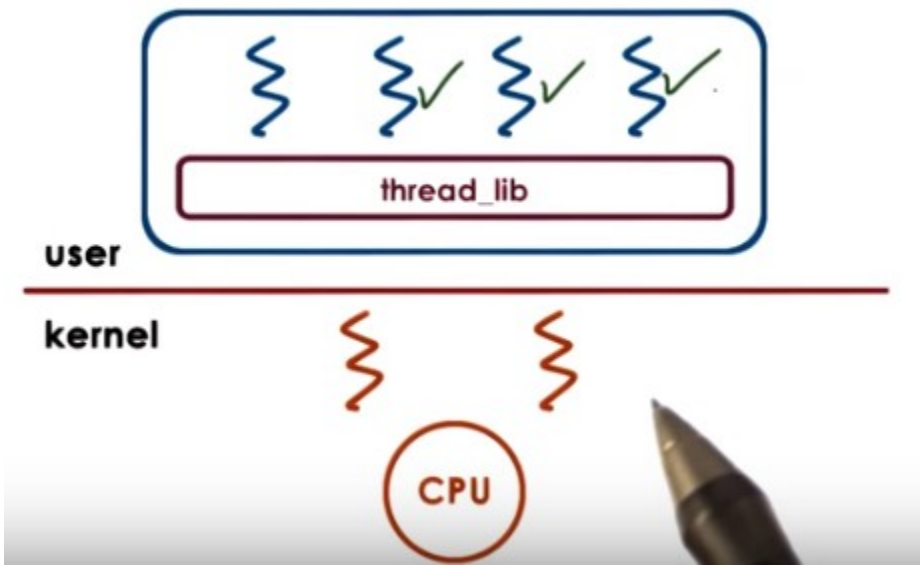
13. The answer to the first question is a very straightforward pthread_setconcurrency function. You can see that you can specify an exact value or you can pass a 0 which will mean that the underlying manager should decide how to manage the concurrency level for the particular process.



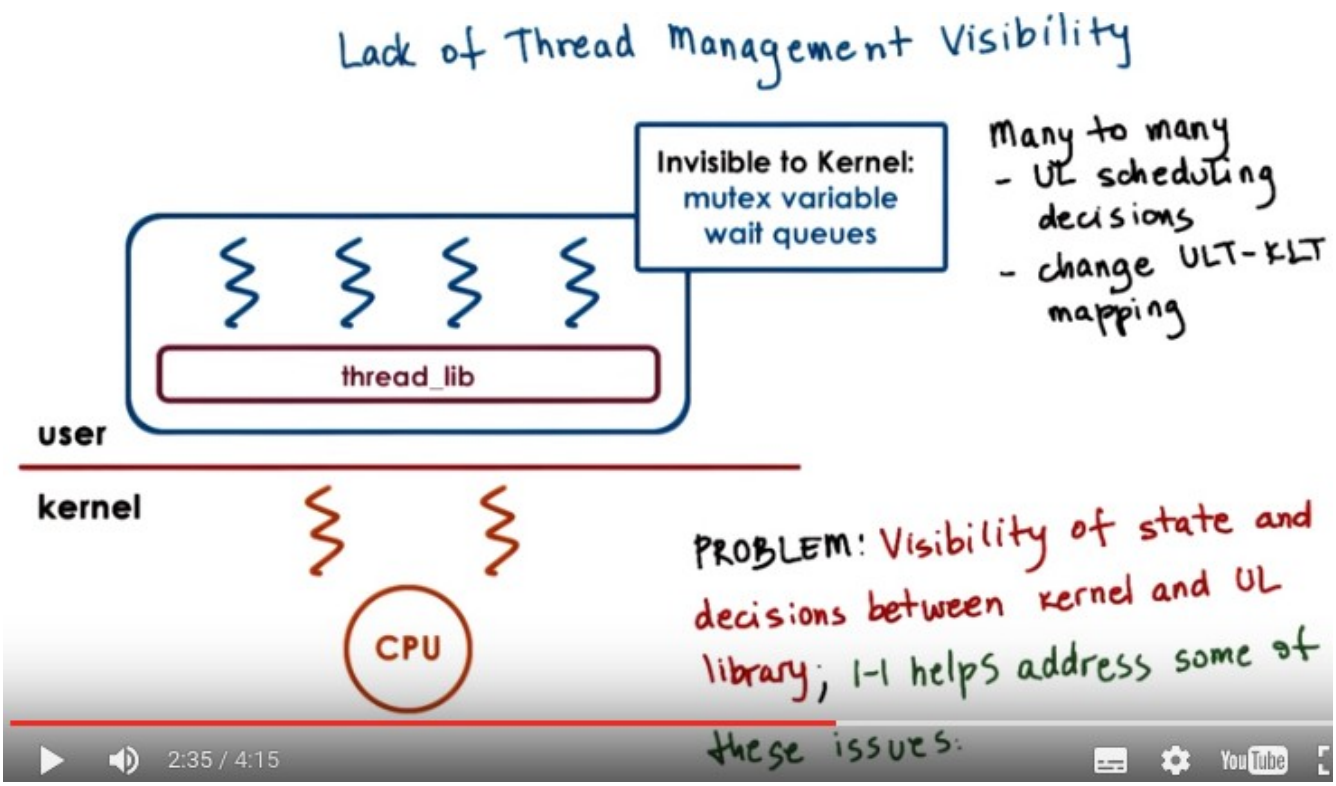
14. In the previous morsel, we talked about the fact that the kernel and the user-level library, don't have insight into each other's activities, and let's talk about that a little bit more now. In the kernel-level, the kernel sees all of the kernel-level threads, the CPUs, and, the kernel-level scheduler is the one that's making decisions. At the user-level, the user-level library sees the user-level threads that are part of that process, and the kernel-level threads that are assigned to that process. If the user-level threads and the kernel-level threads are using the one-to-one model, then every user-level thread will have a kernel-level thread associated with it, so, the user-level library will also essentially see as many, kernel-level threads, but it will be the kernel that will actually manage those. Even if it's not a one-to-one model, the user-level library can request that one of its, user-level threads be bound to a kernel-level thread. This is similar of what we would want to perhaps to in a multi-CPU system, if a particular kernel-level thread, is to be permanently associated with a CPU, except in that case we call it thread pinning, and the term that was introduced with the Solaris threads was that a user-level thread is bound to a kernel-level thread. And clearly, in a one-to-one model, every user-level thread is bound to a kernel-level thread.



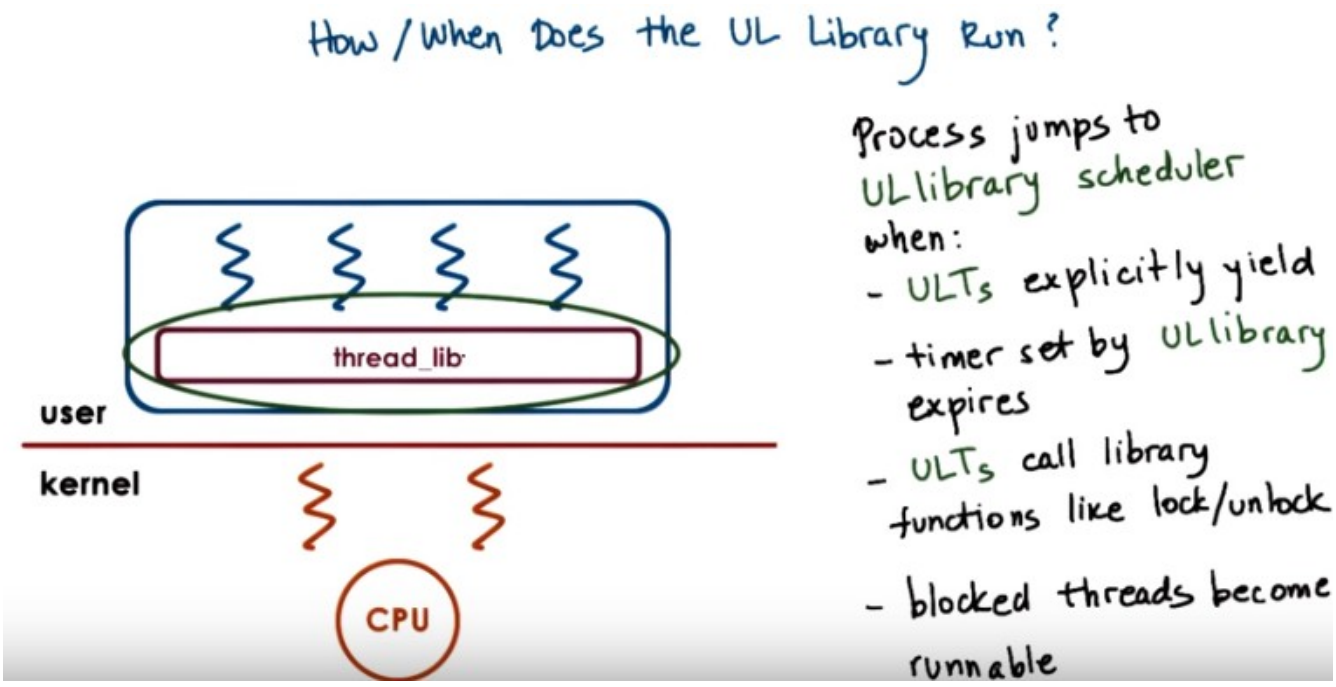
Now let's have the situation in which, one of the user-level threads has a lock, and so that, basically the kernel-level thread is now supporting the execution of that critical section code. Now let's say, the kernel preempted this kernel-level thread from the CPU to schedule the other one, so that means that the execution of this user-level thread, the execution of this critical section cannot continue.



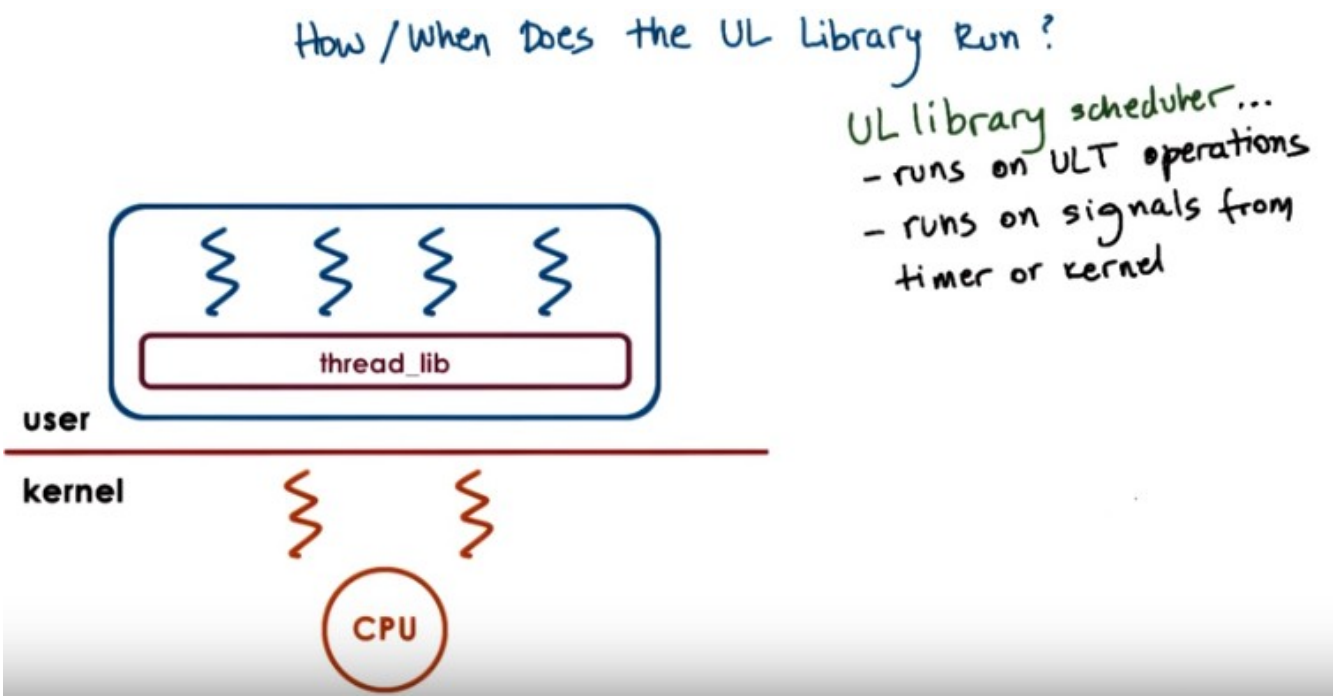
As the user-level library scheduler cycles through the rest of the user-level threads, if they need the lock, none of them will be able to continue. So only after the kernel-level, the schedule of this thread again, will the critical section complete, the lock will be released, and so subsequently the rest of the user-level threads will be able to execute.



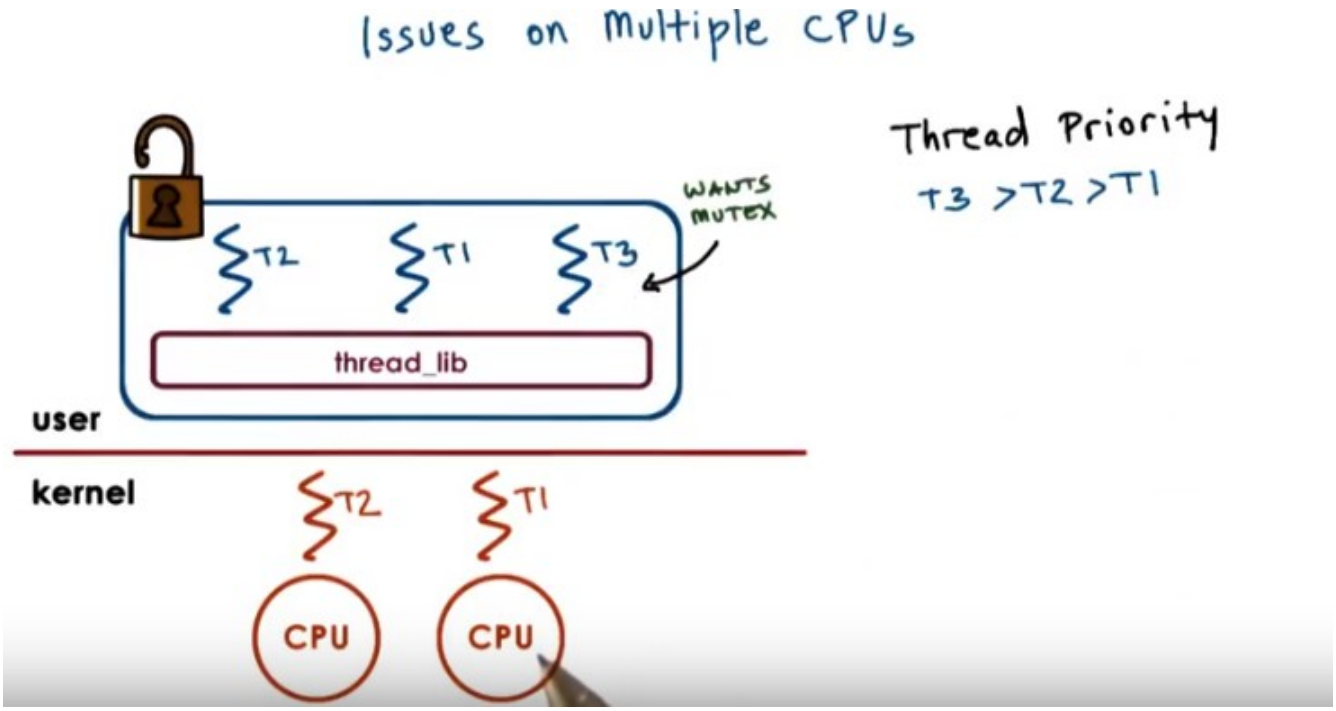
So to reiterate, this problem that there is lack of visibility between the kernel and the user level-thread management, is because at the user-level, the library, makes scheduling decisions that the kernel is not aware of, and that will change the user to kernel-level mappings. And also data structures, like mutexes and wait queues, that's also invisible to the kernel. So the fact that this lack of visibility causes situations such as the one that we described really leads us to the conclusion that we should look at one-to-one models, to address some of these issues.



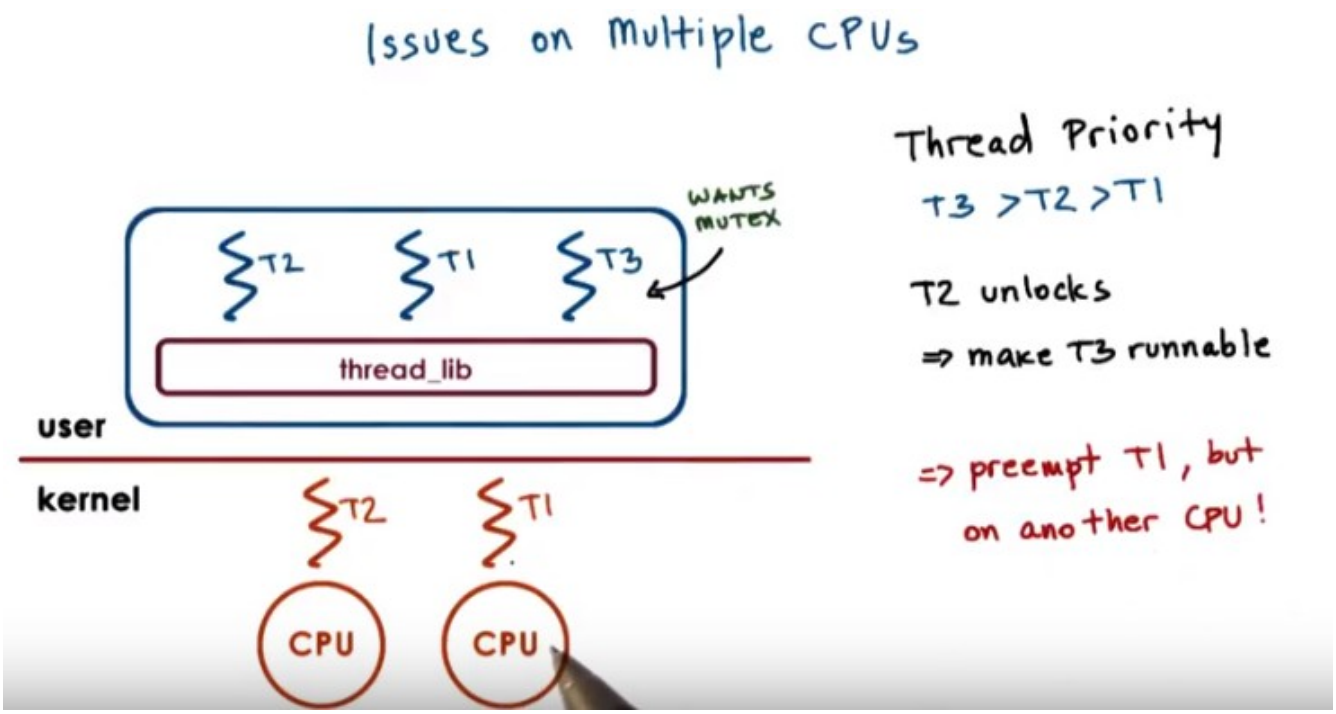
Since the user-level library plays such an important role in how the user-level threads are managed, we need to understand exactly, when does it get involved in the execution loop. The user-level library is part of user process, part of its address space, and occasionally the execution basically jumps to the appropriate program counter into this address space. There are multiple reasons why the control should be passed to the user-level library scheduler, a user-level thread may explicitly yield, a timer that's set by the user-level threading library may expire, also we jump into the user-level library scheduler whenever some kind of synchronization operation takes place, like, when we call a lock, clearly that thread may not be able to run if it needs to be blocked, when we call an unlock operation, then we need to evaluate what is then new runnable thread that the scheduler should allocate on the CPU. And in general, whenever we have a situation where a blocking user-level thread becomes runnable, we jump into the scheduler code, this is part of the library implementation, this is not something that you will explicitly see.



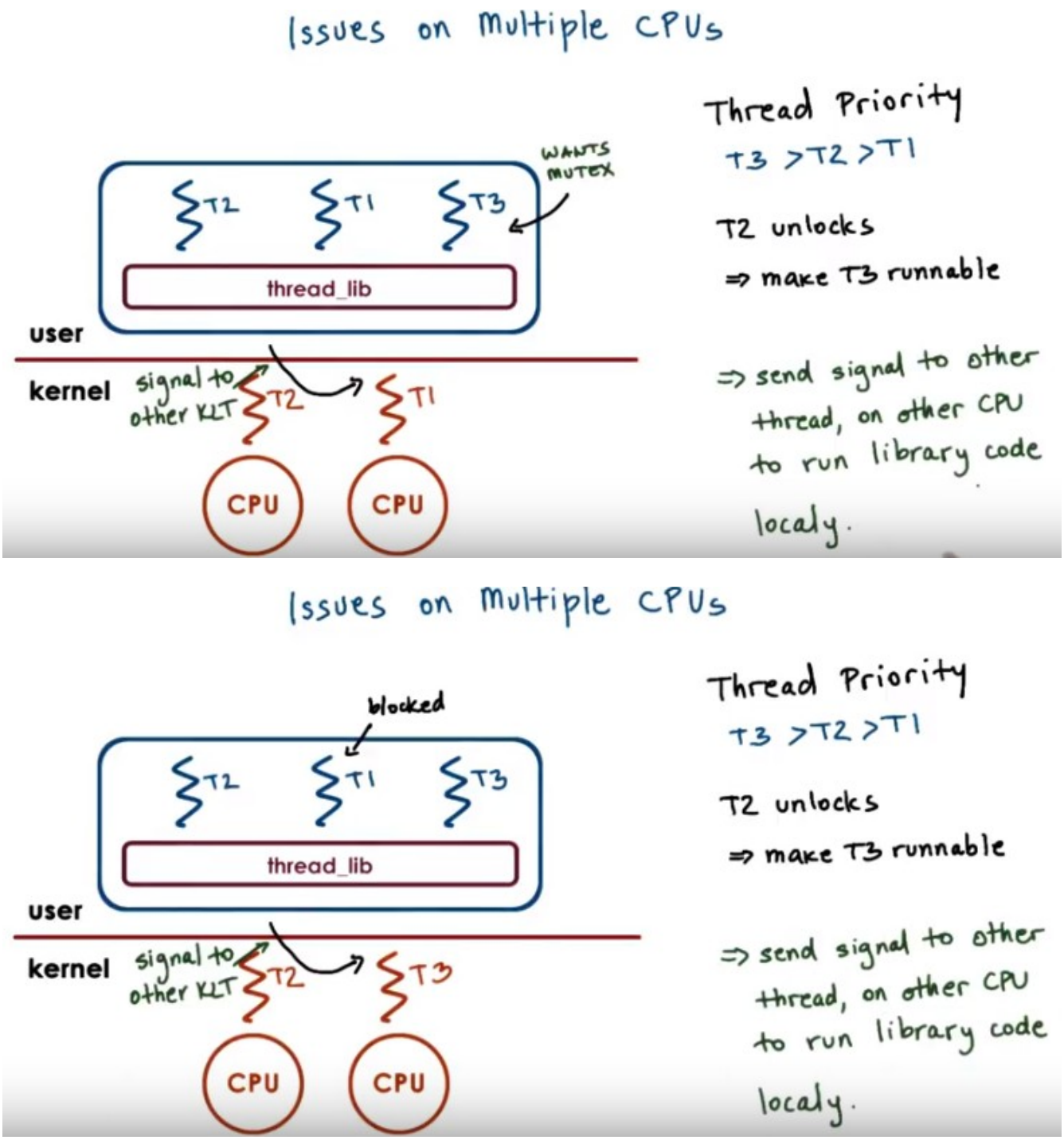
In addition in being invoked on certain operations that are triggered by the user-level threads, the library scheduler is also triggered in response on certain events, certain signals that come either from timer or directly from the kernel. The next morsel should give you an illustration of these interactions.



15. Other interesting management interactions between the user level threading library and the kernel level thread management occur when we have a situation where we have multiple CPUs. In all of the previous cases we've discussed, we only had a single CPU. So all of the user level threads ran on top of that CPU, and then whatever changes, in terms of which of the user-level threads will be scheduled, were made made by user-level threading library were immediately reflect on that particular CPU. In a multi-CPU system, the kernel level threads that support a single process may be running on multiple CPUs, even concurrently. So we may have a situation when the user-level library that's operating in the context of one thread on one CPU needs to somehow impact what is running on another thread on another CPU. Let's consider the following situation. Let's say we have three user-level threads that are running, T1, T2, and T3. And their priorities are such, so that T3 has the highest priority followed by T2, and then T1 has the lowest priority. Let's say the situation is such that T2 is running in the context of one of the kernel-level threads and currently holds a mutex. T3, the highest priority thread, is waiting on that mutex, and so it's blocked. It's not executing. And therefore, the other user-level thread, T1, is the one that's running on the other kernel-level thread on the other CPU (即右邊那個 CPU).

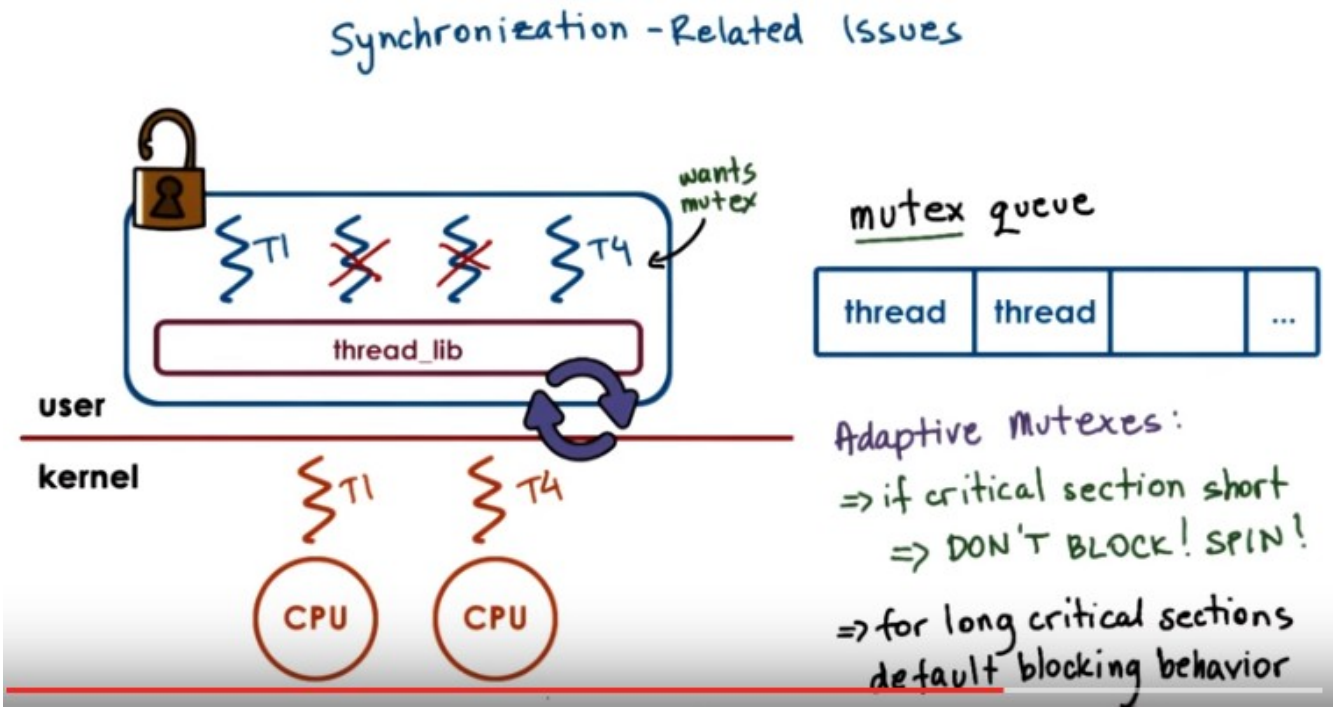


Now, at some later point, T2 releases that mutex, it unlocks it. And as a result of that ,T3 becomes runnable. Now, in all three threads, T1, T2, and T3, are runnable, and so we have to make sure that the ones with highest priority are the ones that actually get to execute. What needs to happen is T1 needs to be preempted, since that's the one with the lowest priority among the three. And we're making this realization while running in the context of the T2 thread. When T2 performed the unlock operation, that's when we invoked the user-level threading library. And that's when we determined that we need to schedule T3 on top of the other context. We need to context switch T1. However, T1 is running on another CPU (即右邊那個 CPU), and so we somehow need to notify this other CPU (即右邊那個 CPU) to do something to update its registers and its program counters. We cannot directly modify registers of one CPU (即右邊那個 CPU) when executing on another CPU (即左邊那個 CPU, 因為是 T2 unlock 的, 所以 T2 有責任將 unlock 的消息告訴告訴右邊那個 CPU).



What we need to do, instead, is to send some kind of signal, some kind of interrupt from the context of one thread and one CPU (即左邊那個 CPU) to the other thread on the other CPU (即右邊那個 CPU). ← 這就是為甚麼要弄 signal 和 interrupt. And to basically tell this other CPU to go ahead and execute the library code locally because the library needs to make some kind of scheduling decision and change who's executing. Once that signal happens, the user-level library on the second CPU will determine that it needs to schedule the highest priority user-level thread, T3. And thread T1, which has lowest priority, will be the one that's blocked. So basically, once we start adding multiple CPUs and have multiple kernel and user-level threads in the process, the interactions between the management and the kernel and the user-level becomes a little bit more complex than what the situation is when there's only

one CPU.

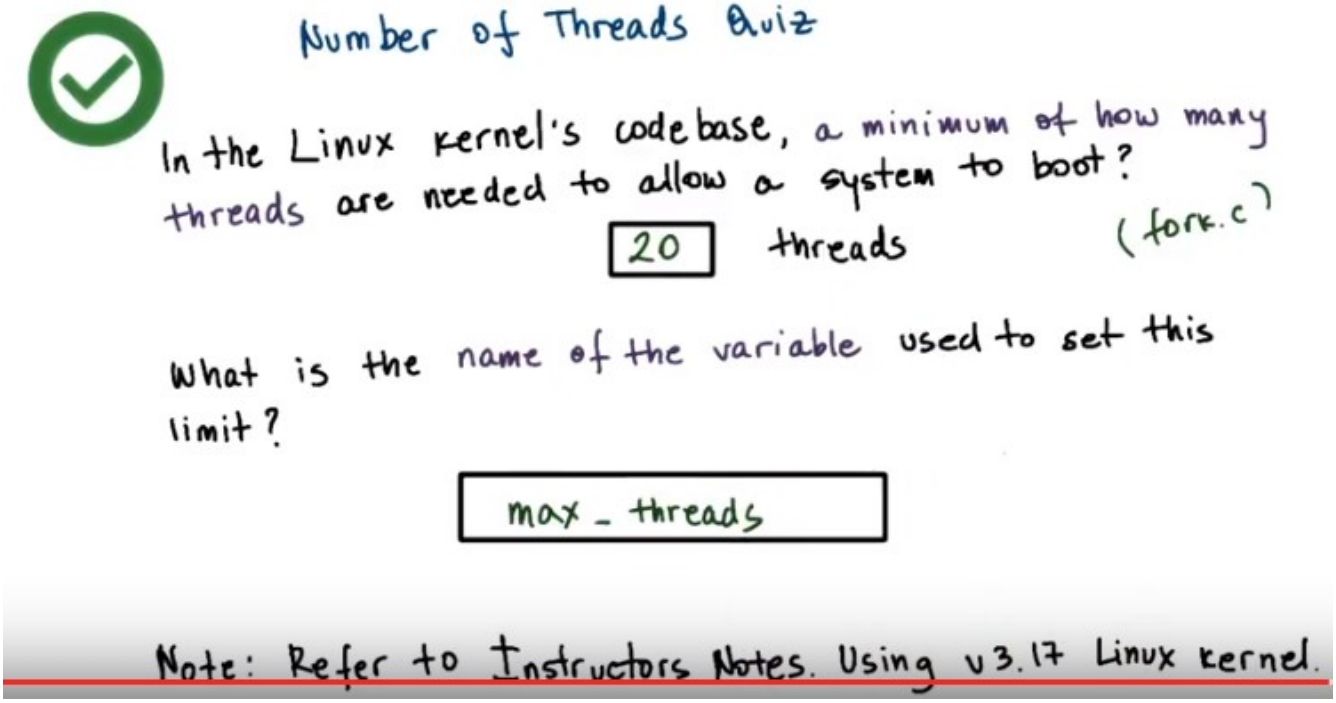


16. Another interesting case when we have multi CPU systems and threading support at the user and the kernel level is related to synchronization. Consider the following situation. We have one user level thread T1 running on top of one kernel level thread on one CPU. And this thread currently has a mutex. A number of user level threads may be blocked, but then on another CPU, currently a user level thread T4 is scheduled. Let's say this thread T4 actually needs to lock the same mutex that's currently held by T1. Now the normal behavior would be to place T4 in that case on the queue that's associated with this mutex. That's what we saw during our earlier discussion about threads and concurrency. However, on a multi-CPU system, it's possible to have this situation. The owner of the mutex, the one that's currently executing the critical section, is running on one CPU. And when we request that same mutex from the other CPU, it is possible that by the time we take this thread, T4, and context switch it and place it on the queue that's associated with this mutex. In that amount of cycles maybe the critical section here is very short and T1 will actually complete its execution. If that is the case if the critical section is short then we are better off if the thread that needs the mutex. Actually (T4) just ends up spinning on this CPU (即右邊那個 CPU). Just burning a few cycles, waiting a little bit until T1 actually releases the mutex. If it takes less time for T1 to release the mutex. Were better off spinning than actually picking a thread, context switching it, and queueing it up on a mutex queue. Super short critical sections don't block spin. For long critical sections we will have the default behavior where a thread is actually properly blocked placed on a queue that's associated with a mutex until the mutex is freed. We call these kinds of mutexes which sometimes result in the thread to spin and other times it result in the thread to block adaptive mutexes. Clearly these only make sense on multi-CPU systems, since whether or not we spin is going to depend on whether the owner of the mutex, like in this case, is actually running on the other CPU. In a single CPU system, that definitely won't be the case, so then it doesn't make sence to consider the use of adaptive mutexes. Early on, when we first introduced mutexes, we said that it is useful to maintain some information about the owner of the mutex. These adaptive mutexes are one example of how such information can be useful. When we try to lock a mutex. If the mutex is currently busy, we can look quickly who the owner of the mutex is, and then

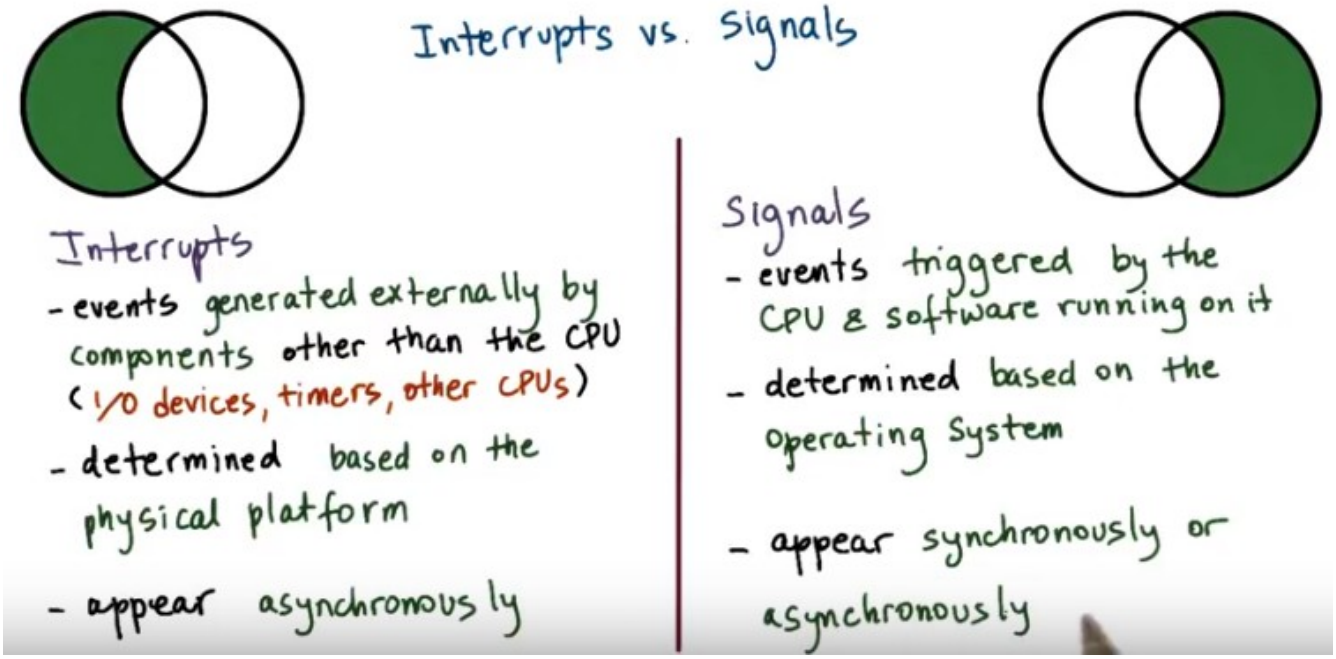
verify whether that other thread is running on another cpu. That will tell us whether or not we should spin or block. Clearly we'll also need to have some idea about the kinds of critical sections that are used with this mutex, so as to determine whether it's likely that the owner of the mutex will release it quickly so we can spin, or not, and in that case we need to block.

And at the end, I want to make some final points about destroying threads. Once a thread is no longer needed, so once it actually exits, it should be destroyed and its data structure, stack, etc., should be freed. However, since thread creation takes some time, like data structures need to be created and initialized, it makes sense to reuse these data structures, essentially as if we're reusing the actual threads. The way this is done is when a thread exits it's not immediately destroyed, the data structures are not immediately freed. Instead the thread is marked as it's on a death row. And periodically a special reaper thread will perform garbage collection which means that it will actually go ahead and free up all of the data structures that are associated with the threads on the death row. If a request for a thread comes in before the thread has been properly destroyed from the death row then its data structure and stack can be reused. And this will lead to performance gains since we don't have to wait for all the allocations.

17. As we saw so far, the interactions between the kernel and the user-level library involve requesting, allocating, and scheduling threads. And this you may assume there's some number of threads allocated at startup to get the operating system to boot. So as a quick quiz, answer the following questions. First, in the Linux kernel's codebase, what is the minimum number of threads that are needed to allow a system to boot? Second, what is the name of the variable that's used to set this limit? Each of these questions can be answered by examining the source code of the Linux kernel. And please refer to the Instructors Notes for some useful pointers.

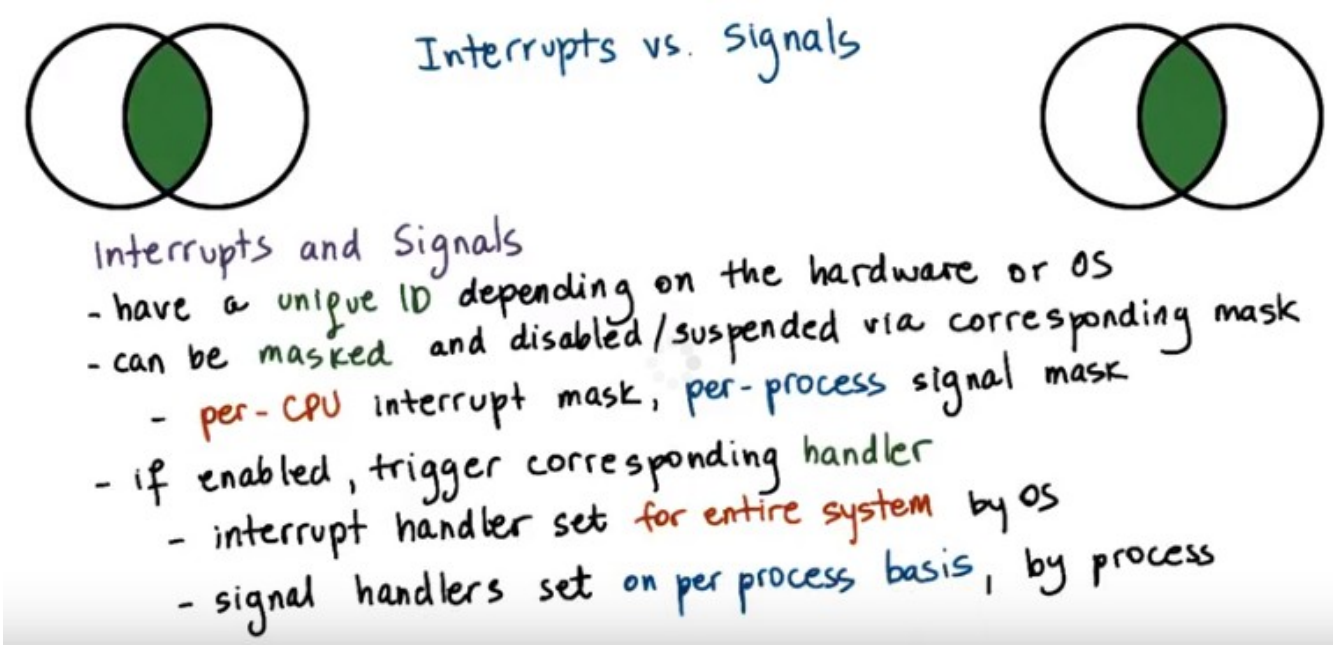


18. The answer to the first question is 20 threads. If you look through the source code for fork.c, you will see that in the, in it fork function, there is a place among lines 278 and 282 that ensures that at least 20 threads are going to be created to get the system to boot. And if you found the answer to this question, then you will know that the variable that holds this value is referred to as max_threads.

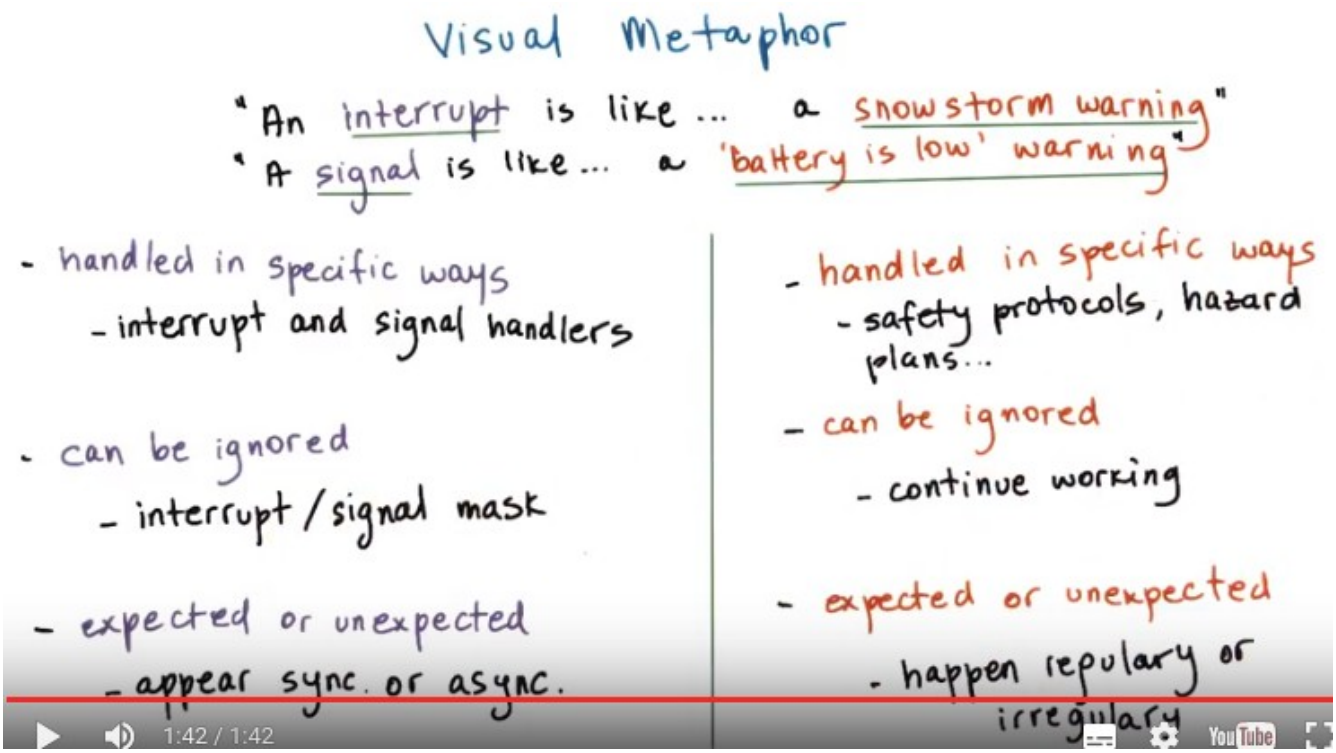


19. 本段是講 interrupt 和 signal 的不同點. In the earlier description of data structures, we mentioned two terms that we have not yet talked about, interrupts and signals. Let's take a moment now to explain these concepts in a little more detail. Interrupts are events that are generated externally to a CPU by components that are other than the CPU where the interrupt is delivered. Interrupts represent, basically,

some type of notification to the CPU that some external event has occurred. This can be from I/O devices like a network device delivering an interrupt that a network packet arrived or from timers notifying the CPU that a timeout has occurred or from other CPUs. Which particular interrupts can occur on a given platform depends on the specific configuration of the platform, like the types of devices that it has, for instance. Or the details about the hardware architecture and similar features. Another important characteristic about interrupts is they appear asynchronously. That's to say that they're not in the direct response to some specific action that's taking place on the CPU. Signals, on the other hand, are events that are triggered basically by the software that's running on the CPU. They're either for real generated by software, sort of like software interrupt, or the CPU hardware itself triggers certain events that are basically interpreted as signals. Which signals can occur on a given platform depends very much on the operating system. So two identical platforms will have the same interrupts, but if they're running a different operating system they will have different signals. Unlike hardware interrupts, signals can appear both synchronously and asynchronously. By synchronous here we mean that they occur in response to a specific action that took place on the CPU, and in response to that action, a synchronous signal is generated. For instance if a process is trying to touch memory that has not been allocated to it, then this will result in a synchronous signal.

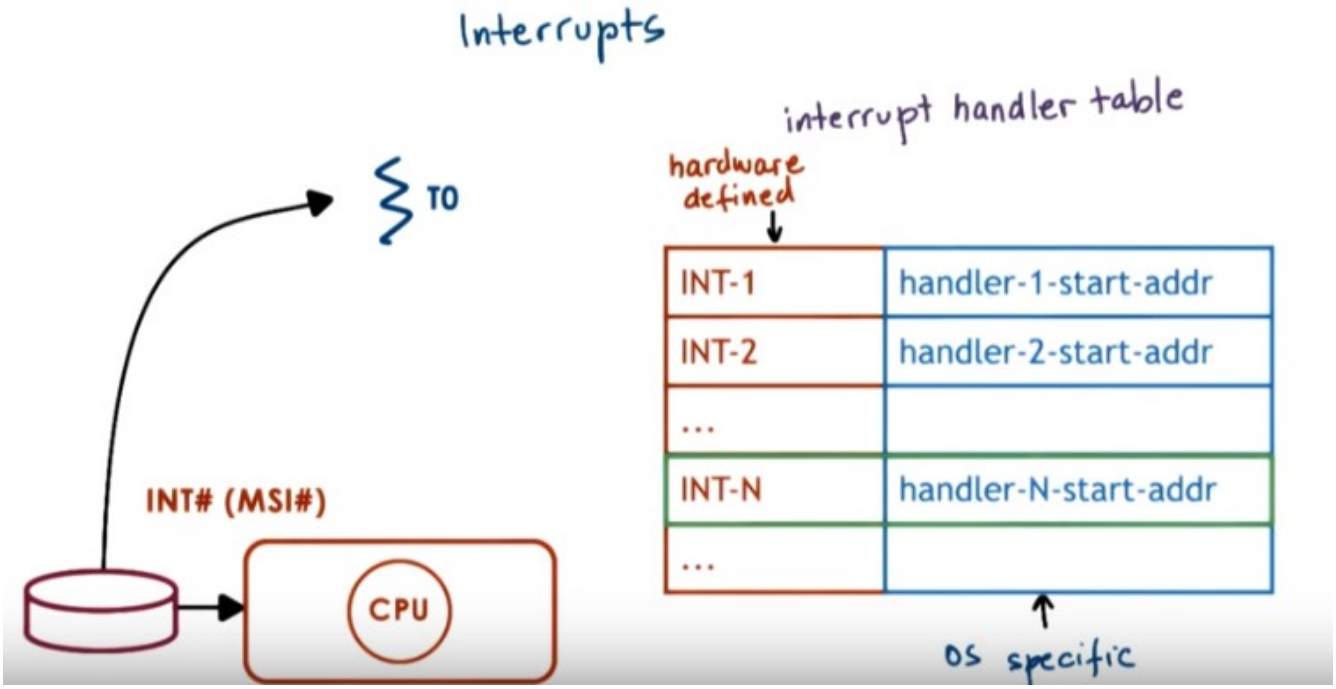


本段是講 interrupt 和 signal 的相同點. There's some aspects of interrupts and signals that are similar. Both interrupts and signals have a unique identifier. And its value will depend either on the hardware in the case of interrupts. Or on the operating system in the case of signals. Both interrupts and signals can be masked. For this, we use either a per CPU mask for the interrupt. Or a per process mask for the signals to disable or to suspend the notification that an interrupt or a signal is delivering. The interrupt mask is associated with a CPU because interrupts are delivered to the CPU as a whole. Whereas the signal mask is associated with a process, because signals are delivered to individual processes. If the mask indicates that the signal, or the interrupt, is enabled, then that will result in invoking the corresponding handler. The interrupt handlers are specified for the entire system by the operating system. For the signal handlers however, the operating system allows processes to specify their per process handling operations

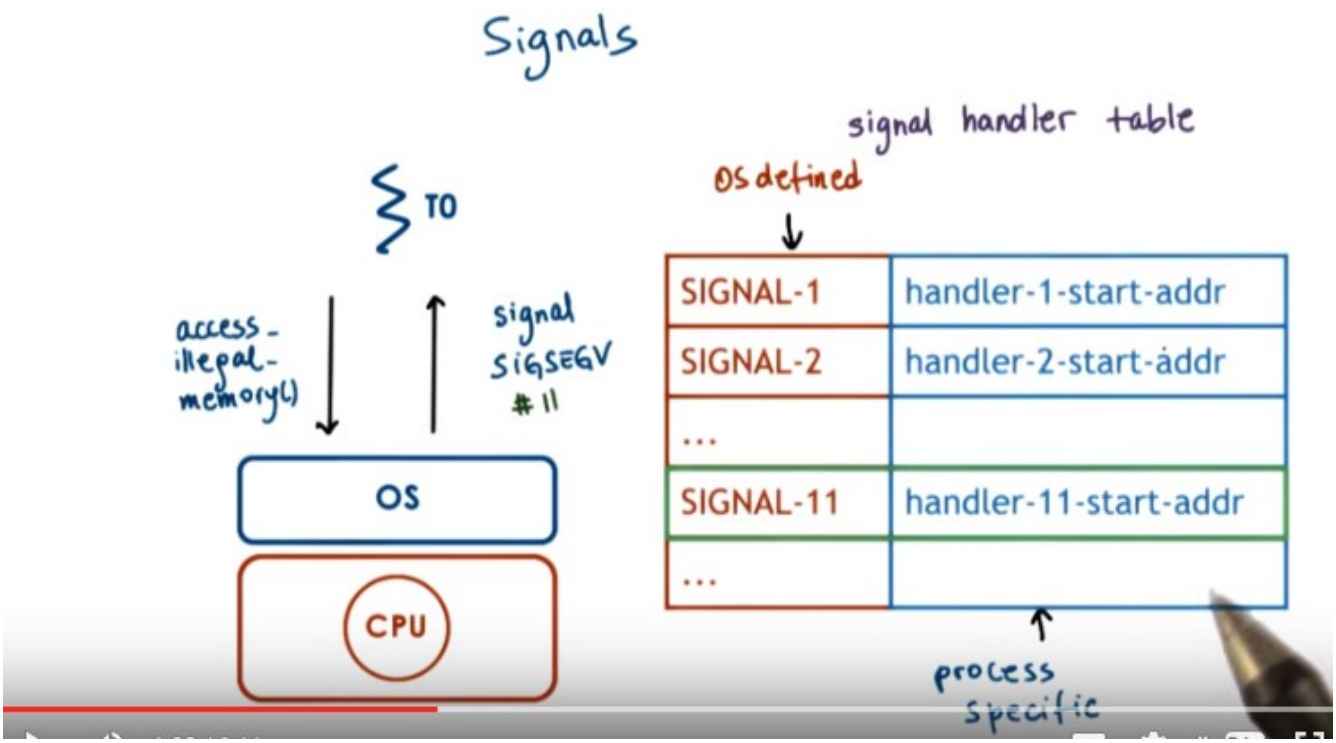


handler 到底是在 interrupted thread 上跑的, 還是在另一個專門的 thread 上跑的? 答: 兩種都可以, 但第一種 (23 段)可能會引起 deadlock (故要 mask), 而第二種(29 段中部)不會. 但由 29 段尾部知, 實際上 dynamically create threads 是很 expensive 的, 所以最好的是若 interrupted thread 有 lock, handler 就在另一個 thread 中跑; interrupted thread 沒有 lock, handler 就在 interrupted thread 中跑. 下面的幾段就是詳細討論這幾點.

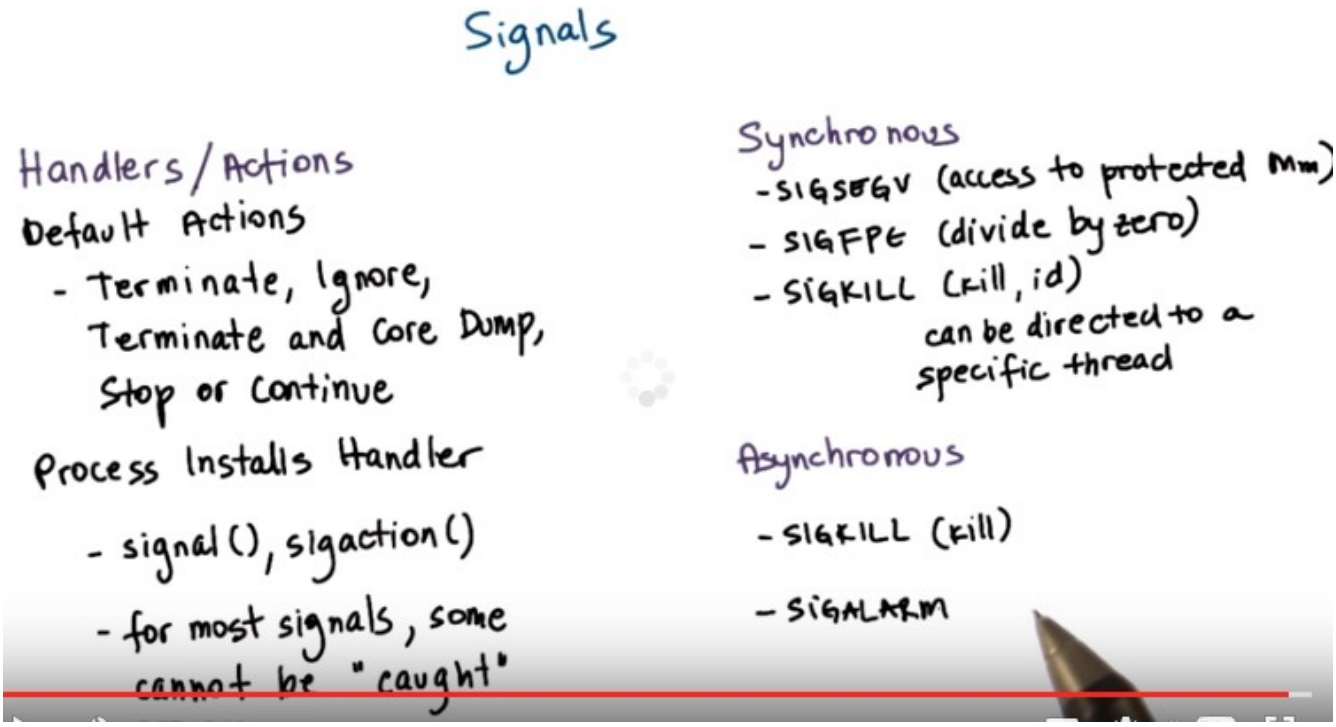
20. Now that we have compared and contrasted interrupts and signals, let's see how we can visualize these concepts. We'll use again an illustration within a toy shop, where we will try to make an analogy between an interrupt and a snowstorm warning, and a signal and a battery is low warning. The reason for these two choices is to make it a little bit more similar with the interrupt being generated by an event that's external to the CPU. So, an event that's external to the toy shop. Whereas the signal is more generated from within, so the battery is low is directly caused by the toy shop worker fixing a toy. First, each of these types of warnings need to be handled in specific ways. Second, both of them can be ignored. And last, we can think about both of them as being expected or unexpected. In a toy shop, handling these types of events may be specified via safety protocols or certain hazard plans. This is not uncommon. There may be, however, situations in which it's appropriate to just continue working. And finally, situations like the fact that the battery died are pretty frequent. They happen regularly, so they're expected. Whether or not it is expected for a snowstorm to occur, that will really depend on where the toy shop actually is. If we think about interrupts or signals, well, both of them are handled in a specific way and that's defined by the signal handler. Next, both interrupts and signals can be masked, as we said. And in that way, we can ignore them. And finally, as we previously discussed, these types of events can appear synchronously or asynchronously. So we have some analogy between these two contexts again.



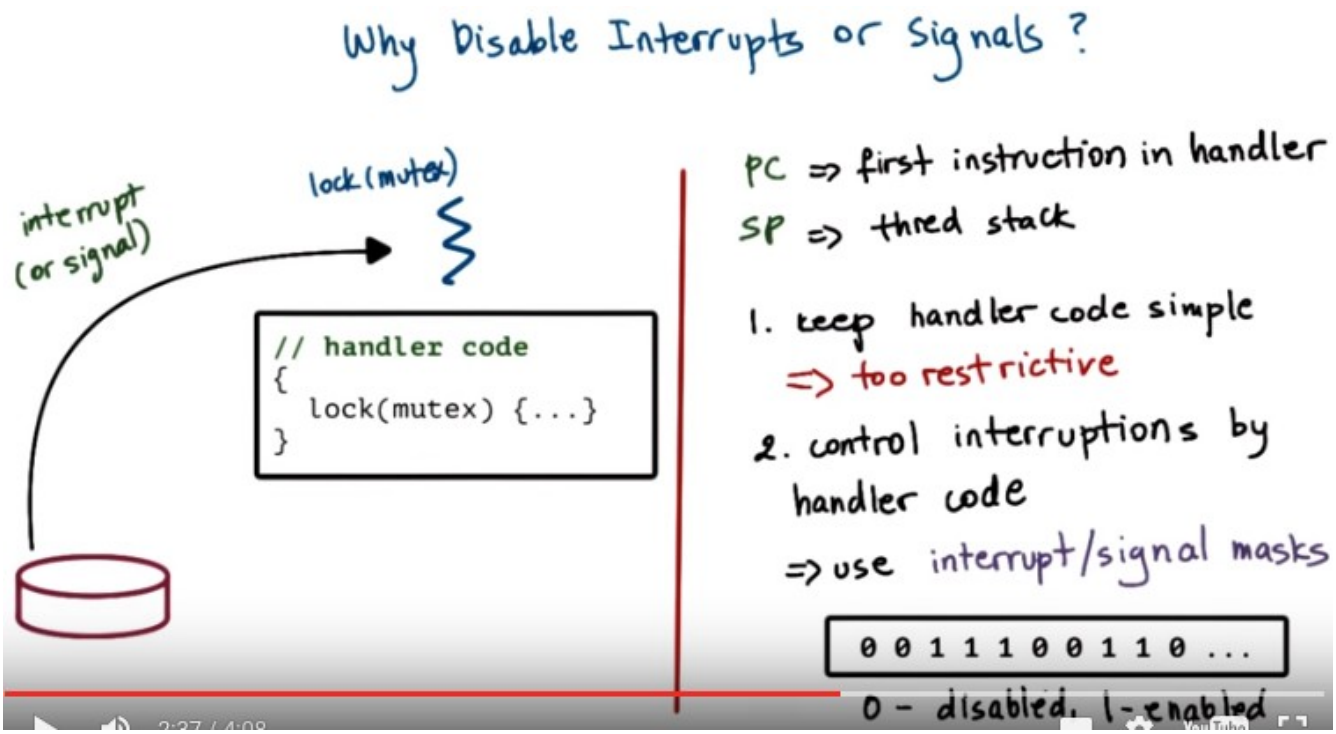
21. Now lets talk a little bit more in depth about interruption signal handling. Lets start with interrupts. When a device like disk for instance wants to send the notification to the CPU it sends an interrupt by basically sending a signal through the interconnect that connects the device to the CPU complex. In the past for this we used dedicated wires, but most modern there's a special message called a message signal interrupter, MSI, that can be carried on the same interconnect that connects the devices to the CPU complex. So PC Express, for instance. Based on the pins where the interrupt occurs or based on the MSI message. The interrupt can be uniquely identified so we know, based on this information, exactly which one of the devices generated the interrupt. Okay, so now the interrupt interrupts the execution of the thread that was executing on top of the CPU. And now what? Now, if the interrupt is enabled only, based on the interrupt number a table is referenced. For all the interrupt supported in this system, this table specifies what is the starting address of the interrupt handling routines. So, this is the interrupt handler table. Based on the interrupt number, for instance, interrupt-N in this case. We look up the starting address of the handler co. And then the program counter is set to that starting address. And the execution of interrupt handling link routine starts. All of this happens in the context of the thread, which was interrupted. Remember again that which exact interrupts can occur on a platform depends on the hardware and how they're handled is specified by the operating system.



22. The situations with signals differs because signals are not generated by an external entity. For instance, if this thread (T0) is trying to access a memory location that hasn't been allocated to it, so it's basically performing an illegal memory access. That will result in the signal being generated, that's called SIGSEGV. So once the OS generates this fault, then the rest of the processing is similar to what was happening in interrupts. The OS maintains a signal handler for every process in the system. For each signal in the system, this table will specify the starting address of a handling routine. So the signal would discuss SIGSEGV. That's number 11 in Linux, the access illegal memory. And for that signal, there will be a handling routine whose starting address will be specified in this table. And as a reminder, again, the signals that can occur on a particular platform are really defined by the operating system that executes there. And how they're handled can be specified by the process.

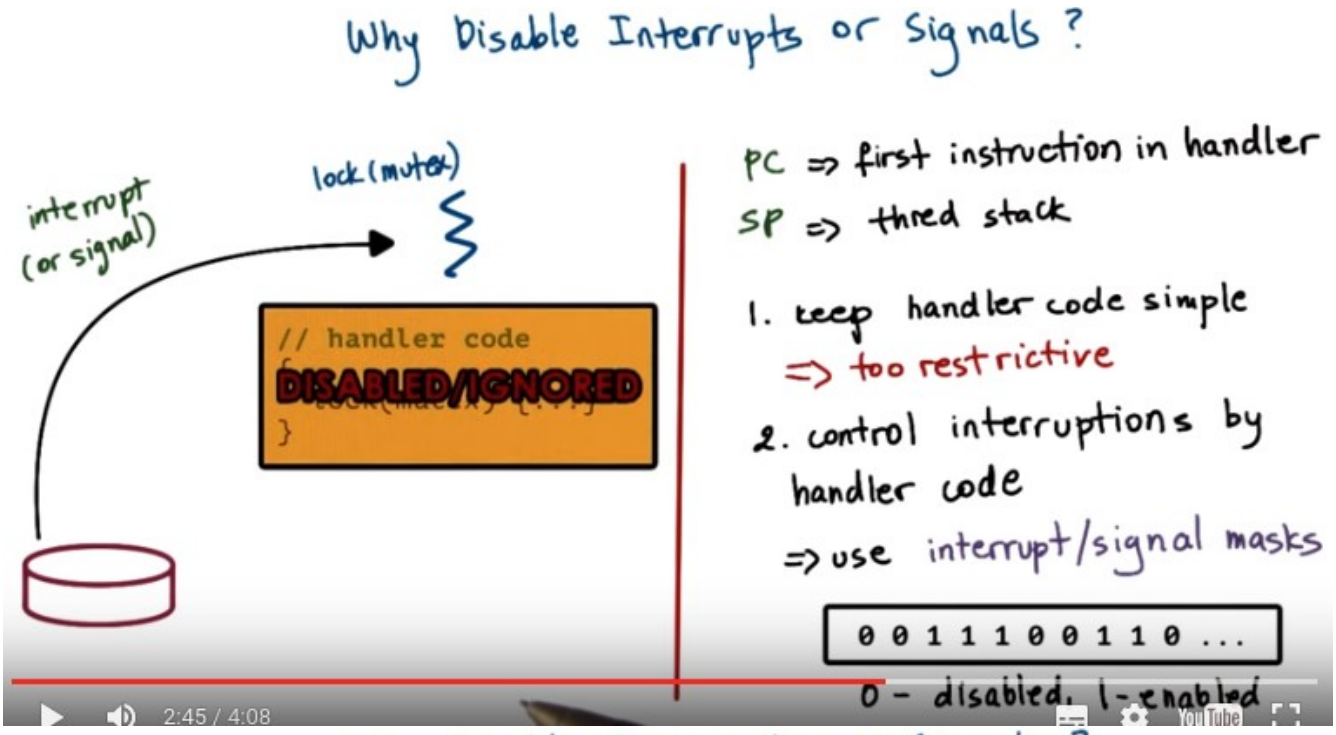


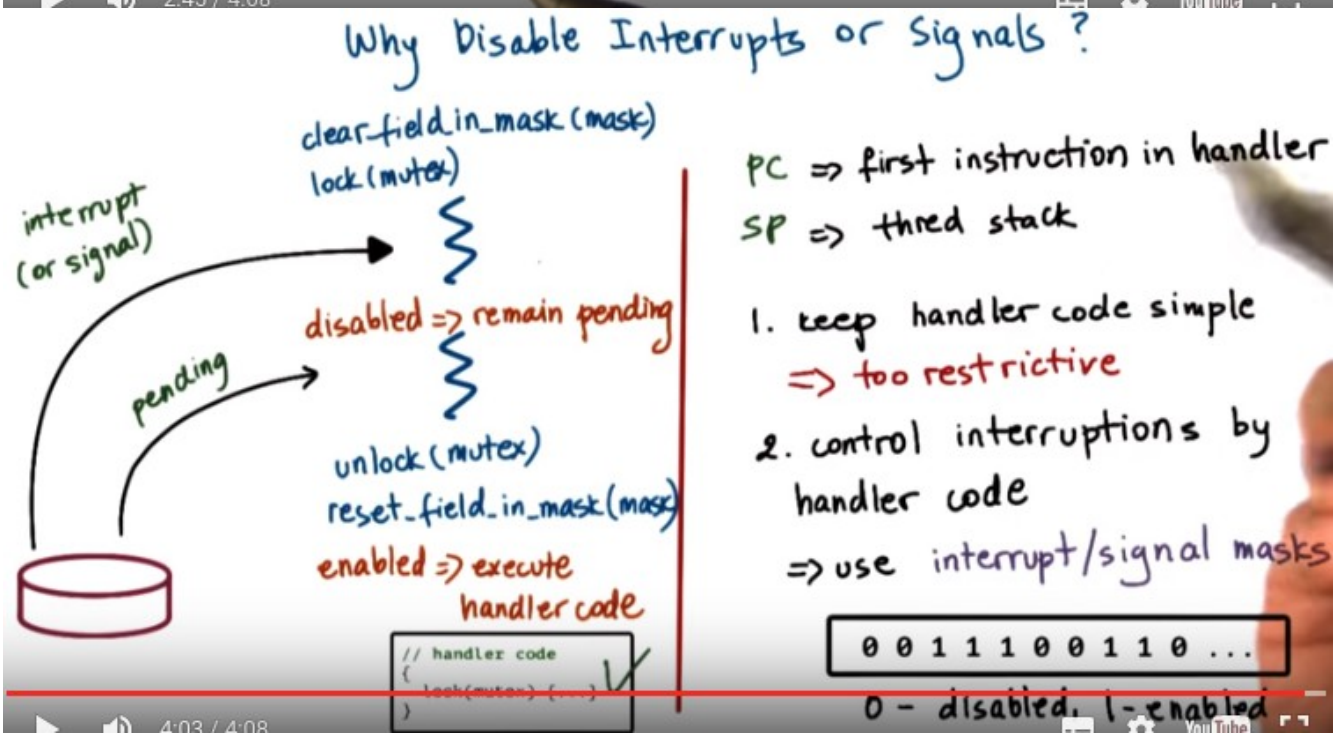
A little more on signals now. The reason we said that a process may specify how a signal should be handled. Is because the operating system actually specifies some default actions for handling signals. For instance, a default action for a signal could be that when that signal occurs a process should be terminated. Or maybe that the signal should simply be ignored. An example of what could happen when the SIGSEGV signal occurs is to terminate at and core dump. So that one can inspect the core dump and determine the reason for the crash of the process. Other common default actions in UNIX like systems include to stop a process or to continue a stopped process. For most signals however, a process is also allowed to install its own custom handling routine. And there are system calls or library calls that allow a process to do this. There are certain signals which are exception to this. These refer to them as signals that cannot be caught. For instance, that would always kill the process. Here are a few examples of synchronous signals. For instance as a result of an attempt at access to illegal memory location to protect that memory location. This signal SIGSEGV would occur. Or that we have a signal that occurs synchronously as the result of an attempt to divide by 0. An example of a synchronous signal is also the one that can be directed from one process to another. So there is an API how to send a directed signal to a specific thread. And this is really asynchronous event. There are also asynchronous signals. For instance, this same command kill that's here used to send a directed signal can also be used to cause a process to terminate. And from the process perspective, this is generated asynchronously. Similarly, a timeout that's generated as a result of a time expiring, is another example of an asynchronous signal.



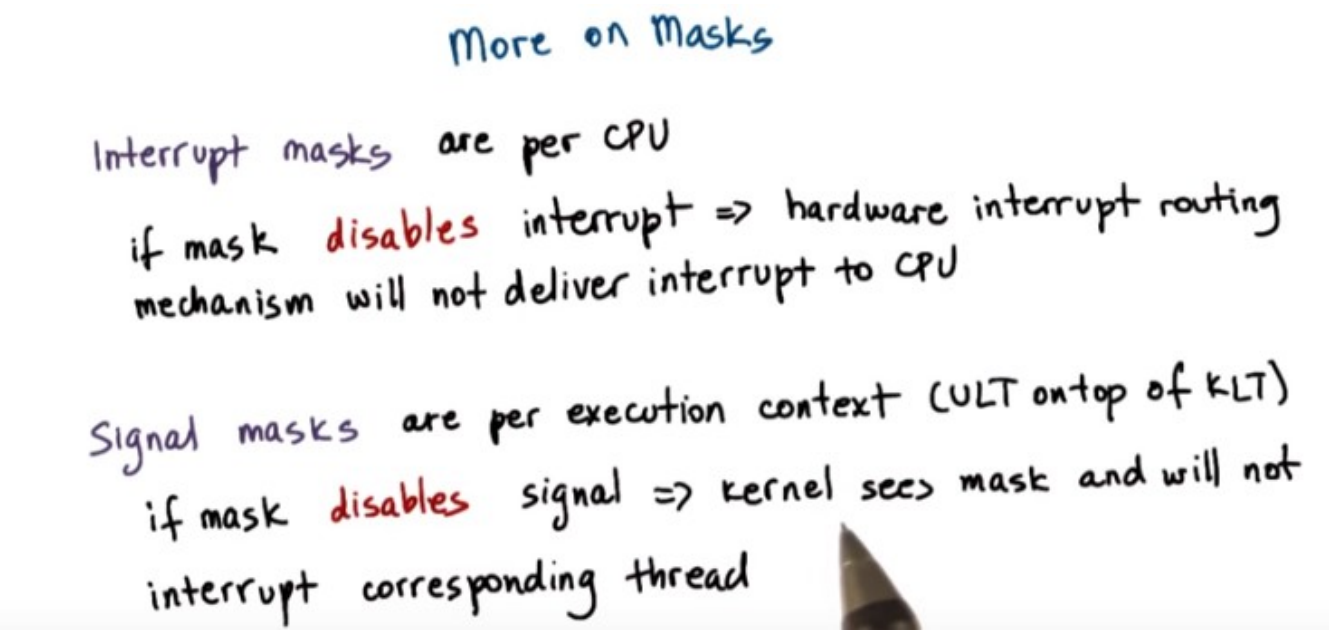
23. There is a problem with both interrupts and signals, in that they're executed in the context of the thread that was interrupted. This means that they're handled on the thread stack (即此種情況下, interrupt 和 sigal 是在 interrupted thread 上跑的) and can cause certain issues that will lead us to the answer of why we should sometimes disable interrupts and signals. To demonstrate this problem, let's assume we have some arbitrary thread that's executing, and this is its program counter and its stack pointer. At some point in the execution, an interrupt occurs, or a signal, and as a result of that, the program counter will change, and it will start pointing to the first instruction of the handler. However, notice that the stack pointer will remain the same. And in fact, this can be nested. There may be multiple interrupts or multiple signals. And in a nested fashion, they will keep executing on the stack of the thread, which was interrupted. If the handling code, the handling routine, if it needs to access some state that perhaps other threads in the system would be accessing, then we have to use mutexes. However, if the thread which was interrupted already had that exact same mutex that's needed in the handling routine, we have a deadlock situation. The interrupted thread will not release the mutex until the handling routine completes the execution on its stack and returns. And we know that that clearly wont hap'pen because this one is locked on this mutex. To prevent from these issues, one possibility we have is to keep the handler code simple. What this means in this context is we can prohibit the handling code to use mutexes. Even if there is no possibility for the handler code to lock on some mutex operation, then the deadlock will not occur. The problem with this is that it's too restrictive. It limits what a handler can do. So instead of enforcing that a handler has to be simple and avoid the use of mutexes, we introduce masks. These masks allow us to dynamically enable or disable whether the

handling code can interrupt the executing mutex. We call these interrupt or signal masks. The mask is a sequence of bits where each bit corresponds to a specific interrupt or signal, and the value of the bit, zero or one, will indicate whether the specific interrupter signal is disabled or enabled. When an event occurs, first, the mask is checked, and if the event is enabled, then we proceed with the actual handler invocation and interrupt or signal handling. If the event is disabled, then the signal or the interrupt remains standing, and it will be handled at a later time when the mask value changes.

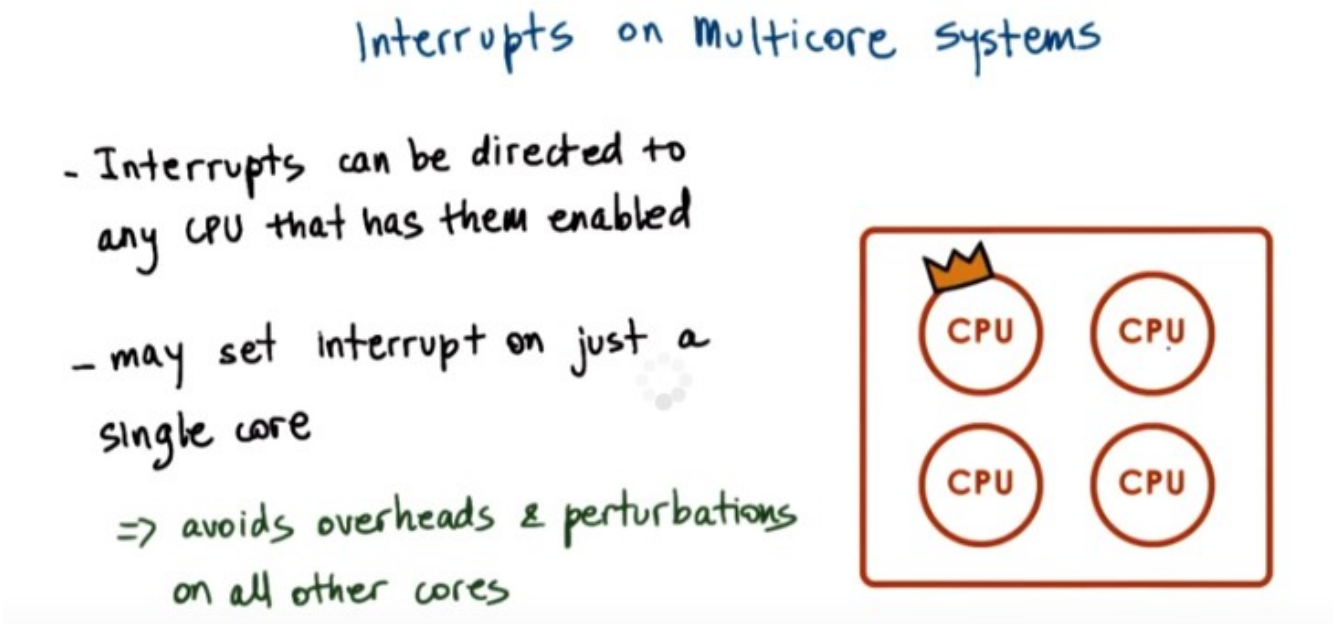




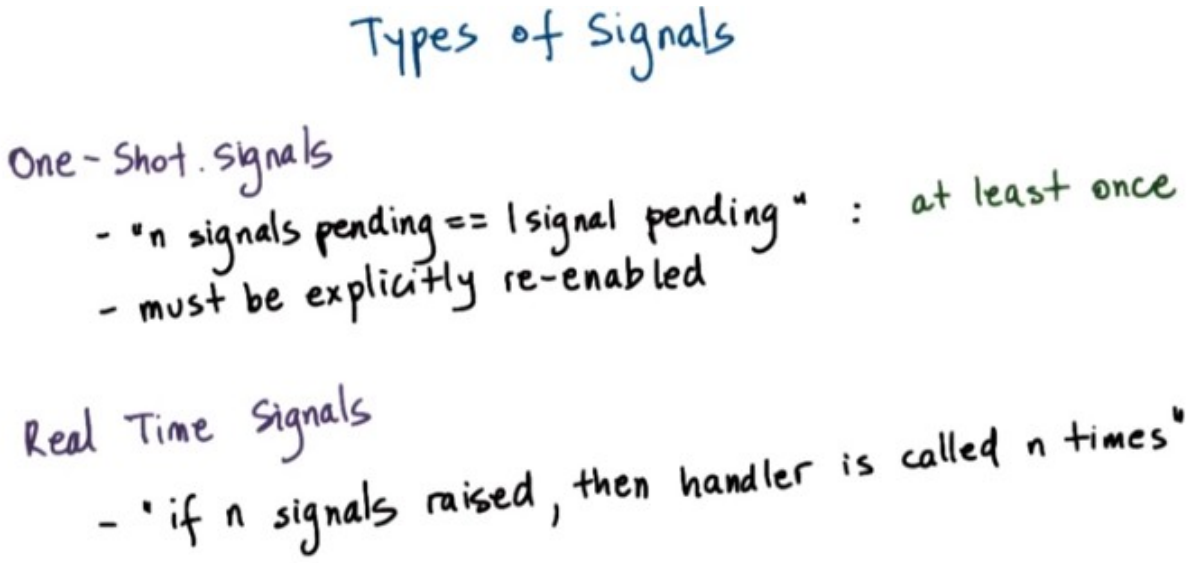
To solve the deadlock situation that we described, the thread, prior to acquiring the mutex, it would have disabled the interrupt. So then even if the interrupt occurs, it will be disabled, and it will not interrupt the execution of the threads. It will not interrupt this critical section. If the mask indicates that an interrupt is disabled, then it will remain pending until a later time. Once the lock is freed, once we perform an unlock operation on the mutex, the thread will then reset the appropriate field in the mask. As a result, the interrupt becomes enabled. And at that point, the operating system will allow the execution of the handler code. Know that at this point, it is okay to execute this code because we no longer hold the mutex (因為 handler 和那個 thread 是同一個 thread). So the thread that would be interrupted when the handler code is called doesn't hold this mutex. As a result, the deadlock will be avoided. We should point out that while an interrupt or a signal is pending, then other instances may occur, and they will remain pending as well. Once the event is enabled over here, the handling routine will typically be executed only once, so if we want to ensure that a signal handling routine is executed more then once, it is not just sufficient to generate the signal more then once.



24. Here are a few more things that you should know about masks. Interrupt masks are maintained on per CPU basis. What this means is that if the interrupt mask disables a particular interrupt, the hardware support for routing interrupts will just not deliver that interrupt to the CPU. The signal mask, however, that depends on what exactly is the user-level process, for instance, the user-level thread, doing at a particular moment, so we say that the signal masks are per execution context. If a signal mask is disabled, the kernel sees that, and in that case, it will not interrupt the corresponding thread. So it will not interrupt this execution context.

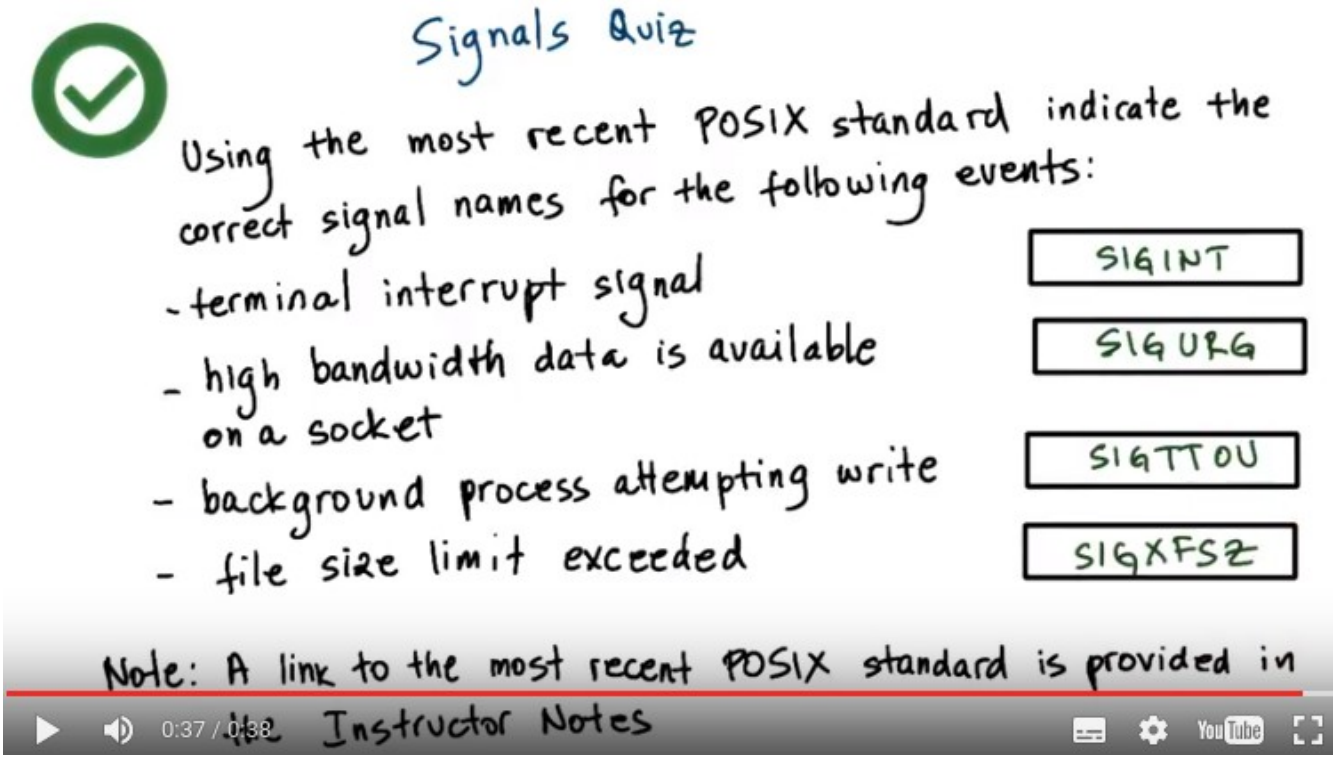


25. There are many details related to interrupt handling that we'll not discuss in this class. I would like to make some final notes about interrupts. And specifically, interrupts in the presence of multi-core Systems. Actually, this applies not just to multi-core Systems, but to multi-CPU systems in general. On the multi-CPU systems, the interrupt routing logic will direct the interrupt to any one of the CPUs. That, at a particular point of time, has that Interrupt enabled. The reason we put this crown here is, what we can do in these multi-CPU systems. We can specify that only one of the CPUs, only one of the cores, is designated for handling the interrupts. That one will be the only CPU that has the interrupts enabled. And so, what that will allow us to do is we'll be able to avoid any overheads or perturbations related to interrupt handling from any of the other cores. The net effect will be in proof performance.

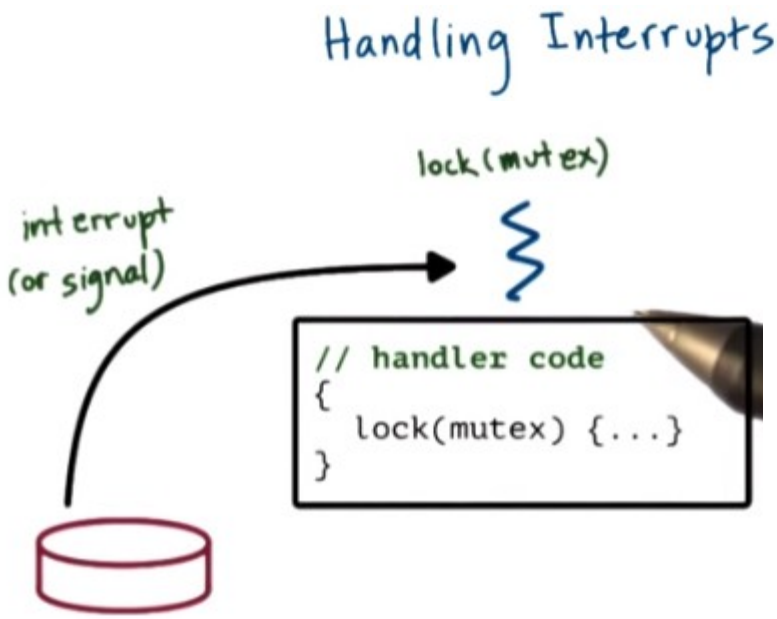


26. And finally, one more point regarding signal handling. We have two types of signals. The first type are the so-called one-shot signals. One property of these signals is that we know that if they're multiple instances of the same signal that will occur, they will be handled at least once. So it is possible that if we have a situation in which only one signal of that kind occurred versus n signals of that same kind occurred, that only one execution of the actual signal handler is performed. The other thing about the one-shot signals is that the handling routine must be re-enabled every single time. So, if the process wants to install some custom handler for a particular signal, then invoking the operation will mean that once when the signal occurs, the process specific handling routine will be invoked. However, any future instances of that signal will be handled by the default operating system action. Or, if the operating system chooses to ignore such signals then they will be lost. Another type of signals are so-called real time signals that are supported in an operating system like Linux, for instance. And their behavior is such that if a signal is raised n times, then the handler is guaranteed to be called n times as well. So, they have sort of a queuing behavior as opposed to an overriding behavior, as is the case with the one-shot signals.

27. In the previous morsel we mentioned several signals. For this quiz, I will ask you to look at the most recent POSIX standard, and then indicate the correct signal names for the following events. The events are terminal interrupt signal, second, high bandwidth data is available on a socket. Next background process attempting to write. And the last event to look at is file size limit exceeded. Note that a link to the most recent POSIX standard is provided in the instructor notes.

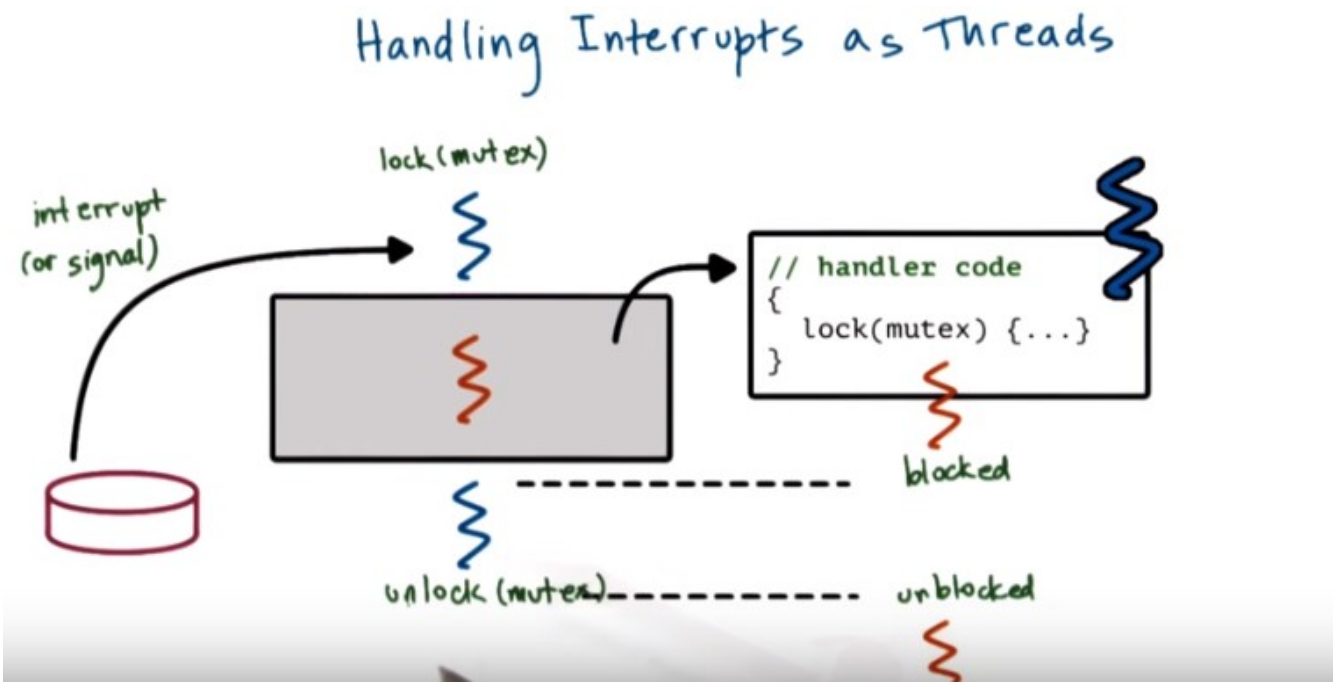


28. So hopefully you found the link for the signals.h header file. On that reference page, you will the table describing the signals and their default actions and descriptions. If you did not find that page, try searching for signal.h online. Using that information as a reference, you can see that the terminal interrupt signal is sickened. For high bandwidth data is available on a socket, SIGURG is used. For background process attempting write, SIGTTOU. And for file size limit exceeded, SIGXFSZ. So, now next time you need to see a signal reference, you will know where to look.



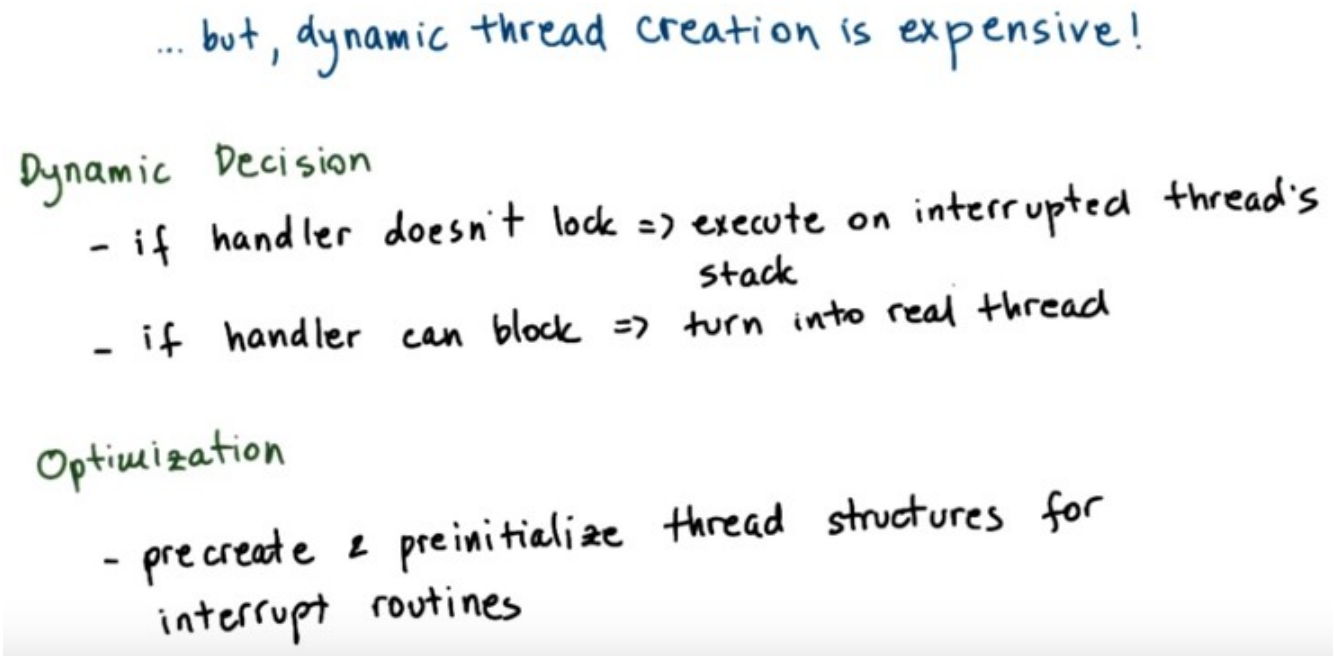
29. Now that we have a basic understanding of how interrupts are typically handled, let's look at the relationship between interrupts and threads. Recall from the previous example that when an interrupt

occurred there was a possibility of a deadlock. And this was happening because the interrupt handling routine was waiting on something, was trying to lock a mutex that was already held by the thread that was interrupted by that interrupt routine. A similar situation could have happened for the signal handling routine. So how can we solve this?

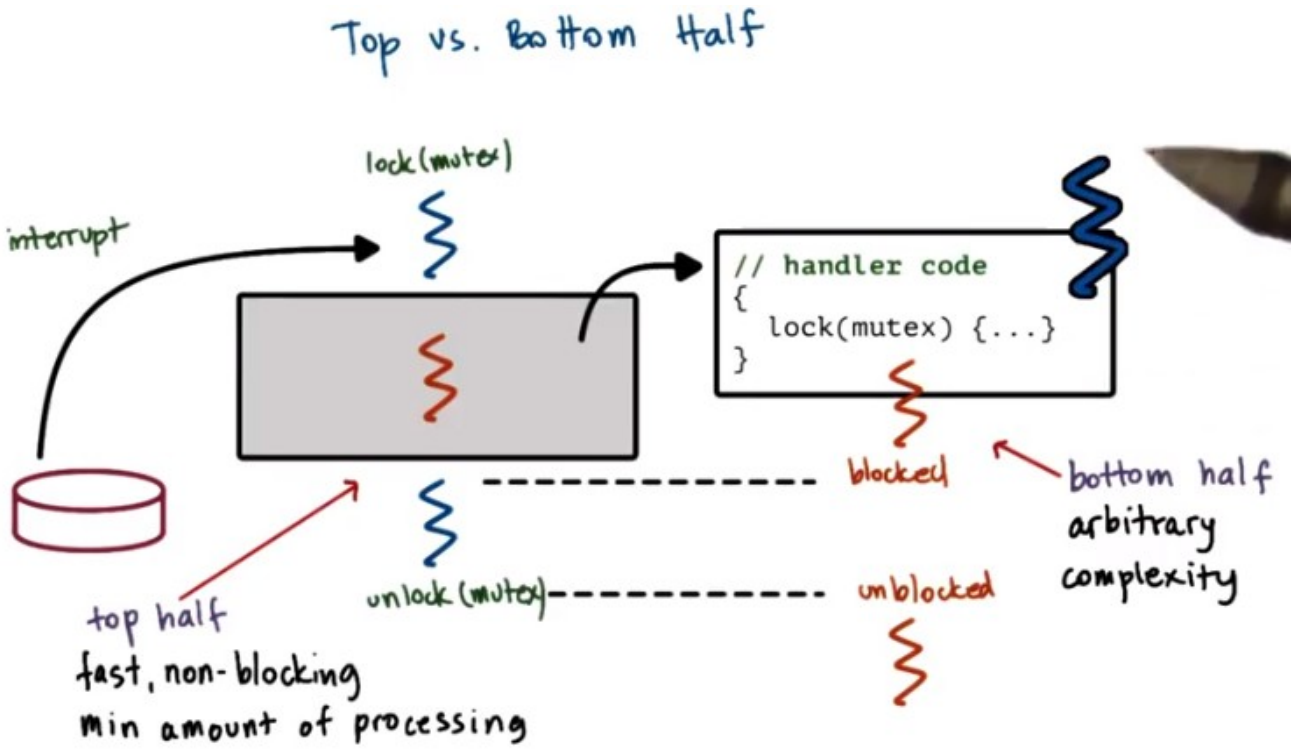


注意上圖最後一行是 unlock (mutex) ----- unblocked (不是 unlocked)

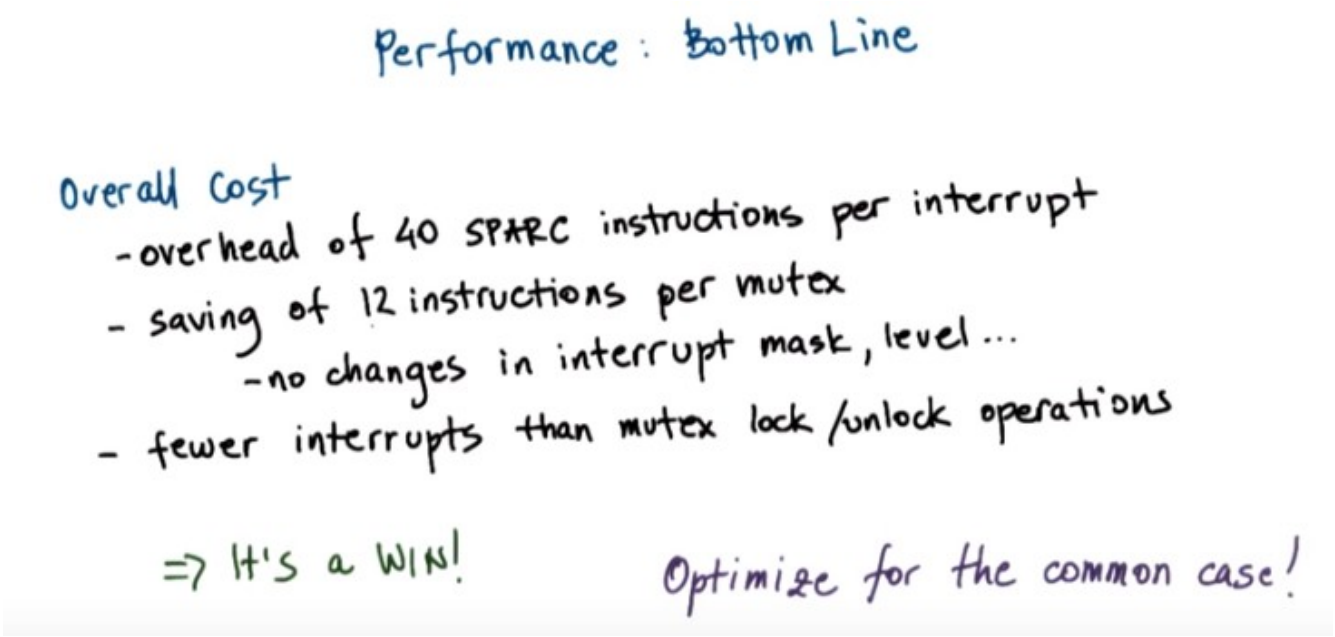
One way that's illustrated in the SunOS paper is to allow interrupts to become full-fledged threads. And that this should be happening every time they're potentially performing blocking operations. In this case, although the interrupt handler is blocked at this particular point, it has its own context, its own stack, and therefore it can remain blocked. So at that point, the thread scheduler can schedule the original thread back on the CPU. And that one will continue executing. Eventually the original thread will unlock the mutex and at that point, the thread that corresponds to the interrupt handling routine will be free to actually execute. The way this happens looks as follows. Whenever an interrupt or a signal occurs, it interrupts the execution of a thread. And by default, that handling routine should start executing in the context of the interrupted thread using its stack and its registers. If the handling routine is going to be performing synchronization operations, in that case, that handler code will execute in the context of a separate thread (這是關鍵) . When the locking operation is reached, if it turns out that this one (即 handler) blocks, then the handler code and its thread will be placed in a wait queue associated with the mutex, and instead the original thread will be scheduled. When the unlock (original thread 的) operation happens, we go back and we unschedule, we de-queue the handler code from the queue that's associated with the mutex and the handling routine can complete.



This sounds like it makes sense. However, one concern is that the dynamic thread creation is expensive. The decision that needs to be made dynamically is whether or not the handler should be handled on the stack of the interrupted thread or as a real thread. The rule that's described in the SunOS paper that's used in the Solaris system is that if the handler routine doesn't include locks, then it's definitely not going to block and so it's safe to execute it on the stack of the interrupted thread. However if there is a possibility of the handler to block because it tries to lock mutexes, then we turn it into a real thread. In order to eliminate the need to dynamically create threads, whenever it's determined that a handler can potentially lock, the kernel precreates and preinitializes a number of threads for the various interrupt routines that it can support. What this means is that the kernel will precreate the number of threads and their associated thread data structures. It will initialize those data structures too so that they point to the appropriate place in the interrupt handling routine, so that any interrupt internal data is appropriately allocated, and similar types of activities. As a result, the creation of a thread is removed from the fast path of the interrupt processing. So, we don't pay that cost when an interrupt actually occurs, and therefore the interrupt handling time can be significantly sped up.



30. Furthermore, when an interrupt first occurs and we're in this initial, in this top part of the interrupt handler, it may be necessary to disable certain interrupts (即 interrupted thread 忽略一些 interrupt). We said that's one way to prevent the deadlock situation. But then, when the interrupt handling is passed to a separate thread, then we can enable any interrupts that we had disabled originally. Because now this is a separate thread, so interrupts occurring can be handled in the same way as it would for any other thread in the system. So there isn't any danger of some additional deadlock situations, because we are executing in an interrupt-handling routine. So basically, this much safer in terms of having external interrupts occur when we are executing in this bottom part of the handling code. I intentionally chose the words top and bottom to describe what's happening in this situation. Because this description of how Solaris uses threads to handle interrupt, is a very common technique how we allow the interrupt-handling routine to potentially have arbitrary complexity, and not be worried about deadlocks. In Linux, these two parts of the interrupt processing are referred to as the top half and the bottom half. So, this is really what we illustrate with this portion of the lesson. The top half will perform a minimum amount of processing, and it's required to be non-blocking. It will be fast basically. The bottom half is pretty much allowed to perform arbitrary types of processing operations. The top half executes immediately when an interrupt occurs. Whereas the bottom half, like any other thread, can be scheduled for a later time. It can block. And so, other than perhaps because of certain timeouts that are associated with the device, we're not really restricted when it actually gets to execute. The paper goes into further detail to describe a specific policy as to how to interpret the priority levels that are associated with the threads when they're being interrupted. Also priority levels associated with the devices. And then use these priority levels in deciding when and how a thread should be used to handle the particular interrupt. And we will skip that discussion. But the takeaway is that if you want to permit arbitrary functionality to be incorporated in the interrupt-handling operations. Then you really need to make sure that, that handling routine is executed by another thread that you can potentially synchronize with. And that thread potentially is allowed to block

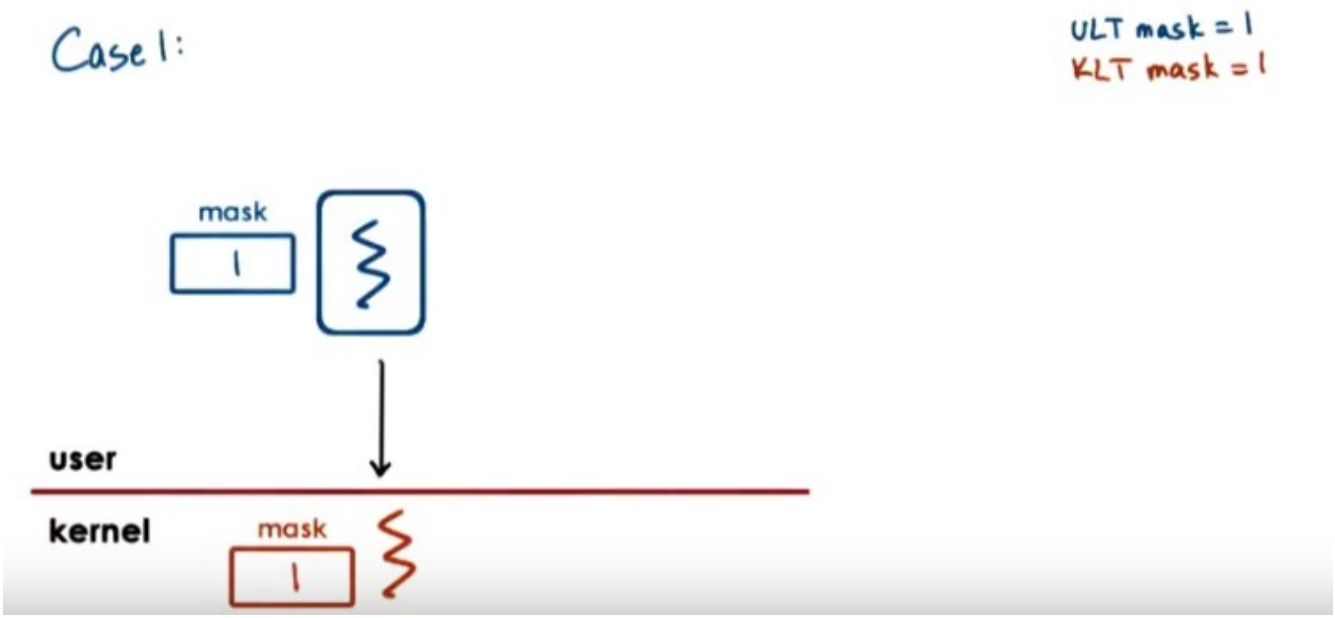


31. Now the reason that this paper described this exercise of creating threads to handle interrupts was really motivated by performance. The operations that are necessary to perform the appropriate checks and if necessary, create a thread to handle an interrupt, add about 40 instructions to each interrupt handling operation. However, as a result of that, it is not necessary to repeatedly change the interrupt mask whenever a mutex is locked, and then, switch it back again whenever the mutex is unlocked. This saves about 12 instructions for every mutex operation. Now because there are way fewer interrupts in the system than mutex lock and unlock operations, then clearly this ends up being a winning situation. We end up saving much more than the actual cost that we end up paying on each interrupt. This observation is also one of the most important lessons in system design, and that is, optimize for the common case. The common case here where the mutex lock/unlock operations. And so, we wanted to make those as efficient as possible. We saved 12 instructions there. Yes, we end up paying somewhere else. We can not sacrifice the safety and the correctness of the system. So we have to make sure we use some other technique to compensate for the fact that we added this optimization. But as long as the net effect is a positive one, this is a very good practice.

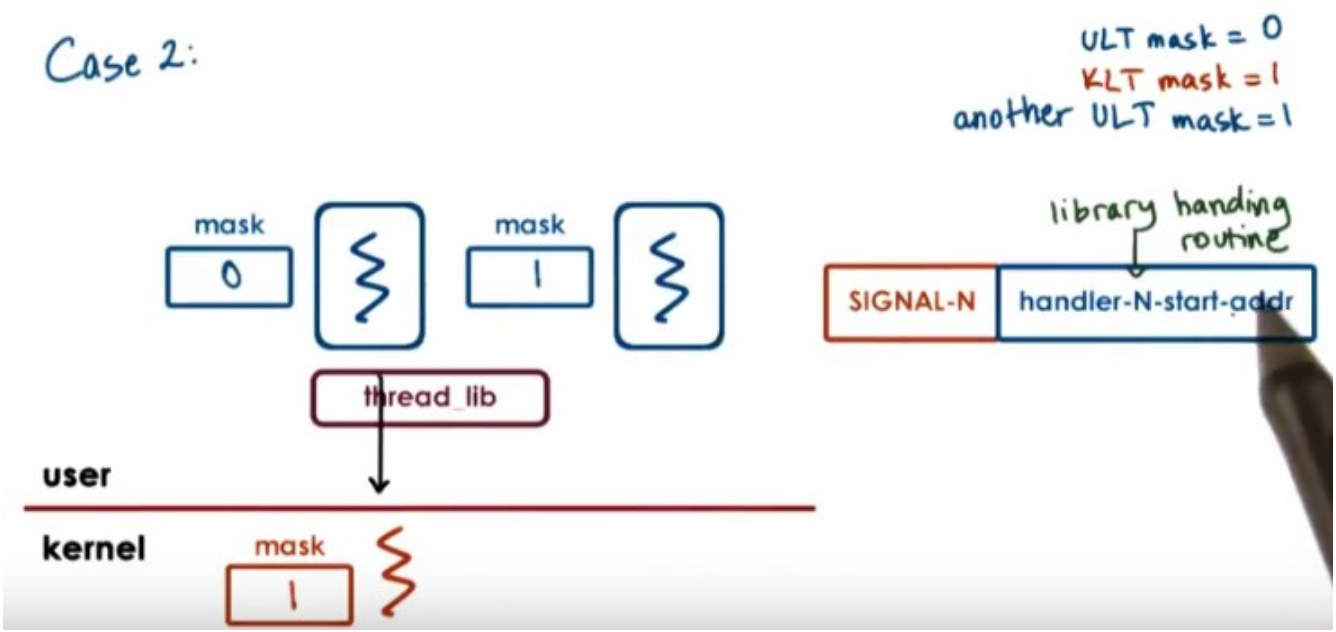
Threads and Signal Handling

disable => clear signal mask
signal occurs – what to do with the signal?

32. Lets now look at some of the interplay between threads and the way signals need to be handled. In the Solaris threads implementation as described in the papers there's a signal mask that's associated with each user level done. And that's part of the user level process. It's visible at the user level library level. There is also signal mask that's associated with the kernel level thread or rather the likely process that it's attached to and that kernel level mask is only visible at the kernel level. Now in when a user level thread wants to disable a signal it clears the appropriate bit in the signal mask (弄成 0) and this is happening at user level. This mask is not visible to the kernel. Now when a signal occurs, the kernel needs to know what should it do with the signal. It is possible that the kernel visible signal mask has that bit still set at one, so the kernel thinks that the signal is enabled as far as this particular process, this particular thread is concerned. If we don't want to have to make a system call and cross from user into kernel level each time a user-level thread modifies the signal mask, then we need to come up with some kind of policy. The SunOS paper that describes the lightweight user-level threading library proposes a solution of how to handle this situation.

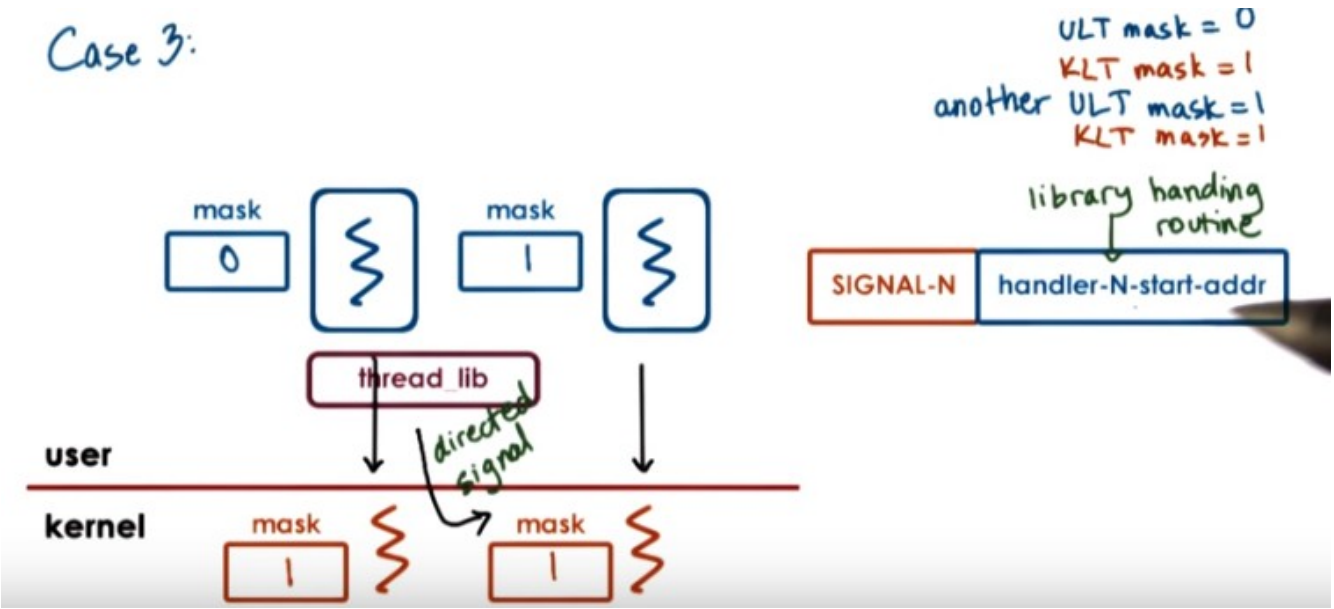


33. To explain what's happening, let's consider a sequence of different situations. The first case we'll look at, both the user-level signal mask and the kernel-level signal mask had the signal enabled. Let's say this is the user-level thread that's currently actually executing on top of this kernel-level thread. Now when the signal occurs, no problem. The kernel sees that the signal is enabled, so it will interrupt the currently running user-level thread that's running on top of this kernel-level thread. And there's absolutely no problem because that user-level thread had the signal enabled as well, so the processing will be safe.



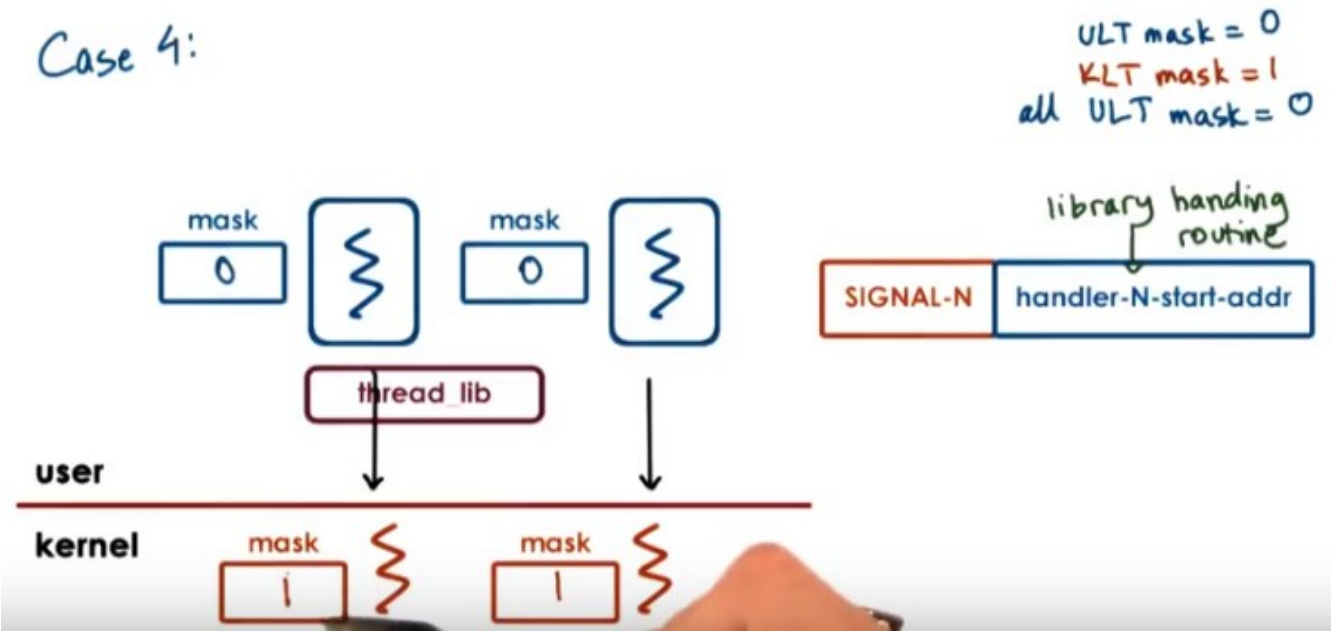
注意上圖中的右邊那個 ULT 沒有在運行, 而是在 in the run queue

34. Now let's consider a second case. Here, the kernel level mask is one, so the kernel thinks that the process overall can handle the signal. However, the user level thread that's currently running on top of the kernel level thread has the signal disabled. So its mask has the bit zero in the signal, in the appropriate place for the signal. But there is another user level thread that's currently in the run queue. So it's not executing. It's runnable but it's not executing at this particular point of time. That one has the mask enabled. The threading library, that manages both of these user level threads, will know about this thread. So now when a signal occurs at the kernel level, the kernel sees that the process overall knows how to handle this particular signal. So it has the bit set as one. But, it should be appropriate for it to interrupt this user level thread, because this particular user level thread, the one that's currently active, has a signal disabled bed. We should figure out a way how to get to the threading library because that's the one that knows about this other runnable thread that would be capable of handling a signal. (以下幾句不是講 Solution) We said that the way signals are handled is that when they interrupt the process or either a thread that's running in the process. The handling routine that needs to be executed is specified in the signal handler stable. So then one easy thing that we can do is for all the signals in the system, we can have a special library routine that will basically wrap the signal handling routine. So when a signal occurs, we start executing the library provided handler. That library provided handler can see all of the masks of the user level threads. (Solution 開始) If we have a situation like here, where the currently scheduled user level thread cannot handle the signal. But there is another runnable user level thread that can, the library handling routine can invoke the library scheduler and can now make this user level thread (指右邊那個 mask 為 1 的) running on the kernel level thread. So then the signal can be handled.

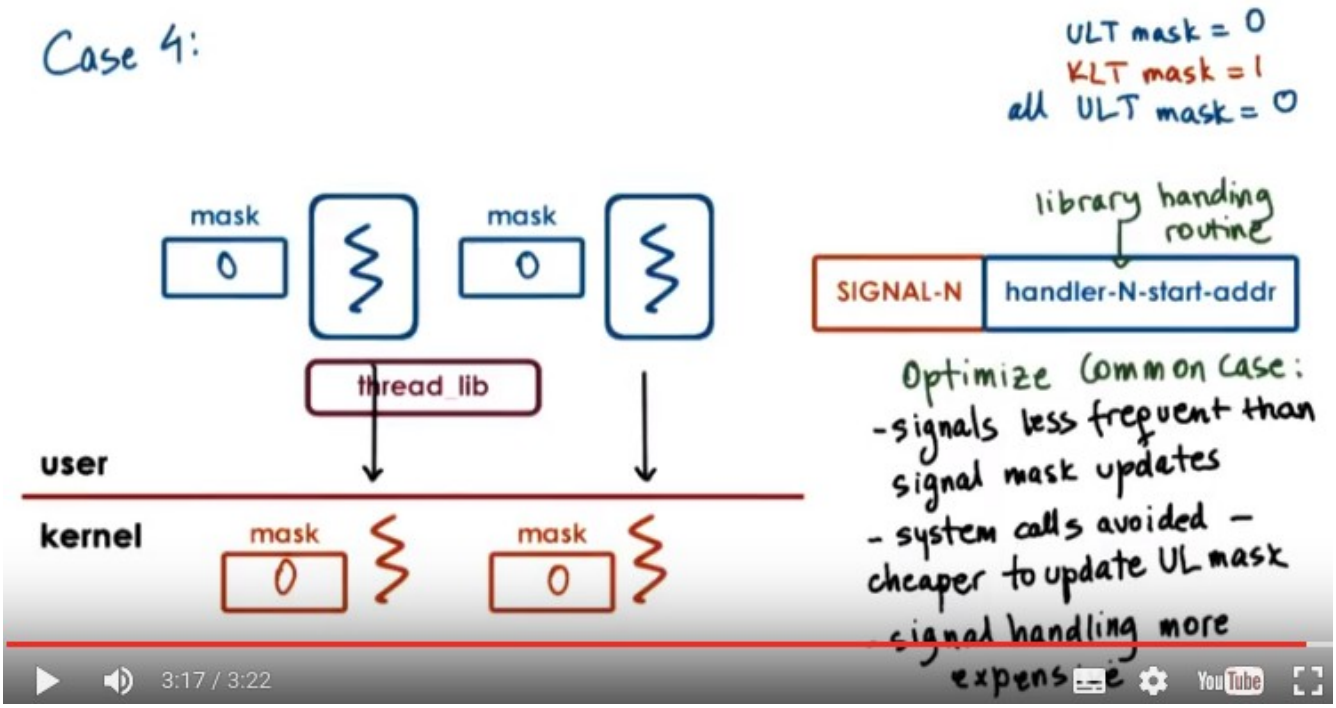


35. Now let's look at yet another case. This is case three now. So here we have a similar thing in that the user level thread that's currently executing on top the kernel level thread where the signal actually occurs, this user level thread has the signal disabled. In the process overall, there is another user level thread that has the signal enabled. And unlike in the previous case when this user level thread (指右邊那個 mask 為 1 的) was just on the run queue, now in this case this user level thread is currently running on another kernel level thread on another CPU. So when the signal is delivered in the context of this kernel level thread (左邊那個), the library handling routine will kick in. The library handling routine knows that there is a user level thread in the system that can handle this particular signal. And it sees that this user level thread is currently associated, it's executing on top of a kernel level thread, or rather

a lightweight process that's managed by the threading library. Since the library knows this, the library will then generate a directed signal to the other kernel level thread to the lightweight process where the user level thread currently is executing. When the OS delivers the signal to this particular kernel level thread (右邊那個), it sees signal mask enabled. Great. And it moves up. Now technically it will still go into the library handling routine first, right? Because that's a wrapper for all of the signal handlers. Here the library handling routine will say, great, the current user level thread that's running in the context of this kernel level thread can handle the particular signal of the signal and so it finally will allow the execution of the signal handler.



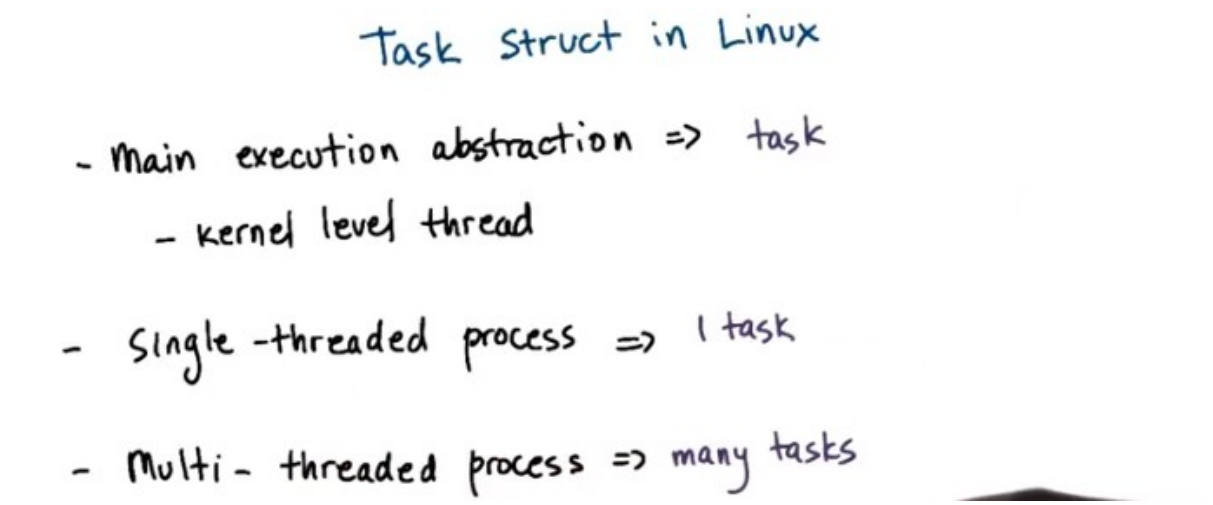
36. And now let's consider one final case in which every single one of the user-level threads has that particular signal disabled. So all of the user-level thread masks are zero. The kernel-level masks are still one, the kernel still thinks that the process can handle this particular signal. When the signal occurs here (左邊那個 KLT), the kernel sees that the signal mask is one, and so it will interrupt whoever is executing in the context of this kernel-level thread. The library handling routine kicks in. It sees that this particular thread has the mask zero. And it sees that it doesn't have any other user-level threads that can handle that particular signal.



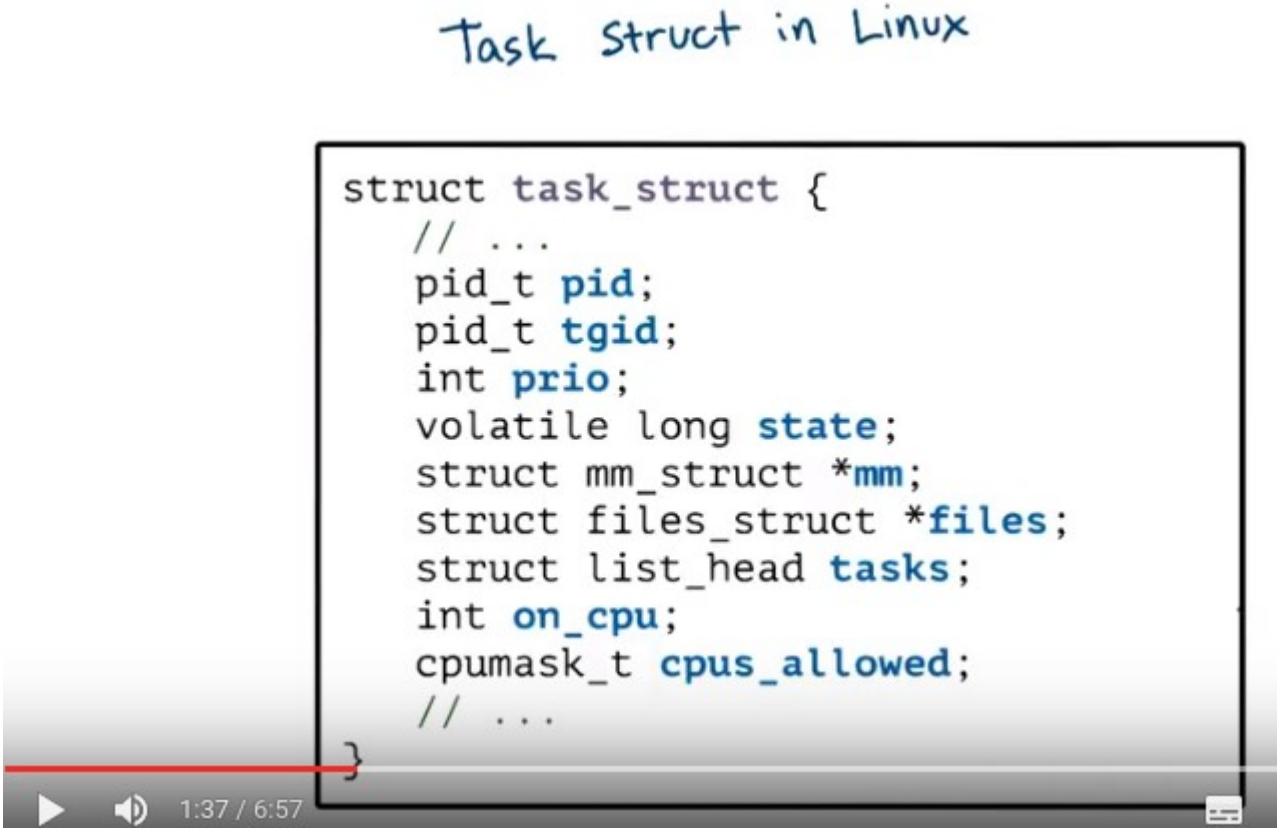
注意上圖還是 case 4, 不是一個新的 case.
上圖中左邊那個 KLT 的 mask 變為 0 後就沒再變了, 右邊那個 KLT 的 mask 中途在 1 和 0 之間變 了幾次.

Now what will happen is the threading library, at that point, will perform a system call, and it will request that the signal mask of the underlying kernel-level thread be changed. So this signal mask (左邊那個 KLT 的) will become zero. Now, from the execution of one thread, we can go ahead and affect the state of the kernel-level masks that are associated with other threads: they may be executing on other CPUs. So here, we can only change the mask that's associated with this kernel-level thread (左邊那個) , and then what the threading library will do, it will basically reissue the signal for the entire process again. The OS will now find another thread in that process. 後面幾句點混亂, 沒弄清楚 . So in this case, this other kernel-level thread. It had originally the mask. The OS will, in that case, find another kernel-level thread. In this case, this other kernel-level thread that has the mask enable. So this one has its zero. And we'll try to do the same thing. We'll try to deliver the signal in the context of this kernel-level thread. It will interrupt this user-level thread via the library handler. As a result, this particular kernel-level mask will be changed as well via system call, and the process will continue until all of the kernel-level signal masks don't show that the signal is disabled for this process. Now another possibility (好像是在說另一回事了, 主要是想強調對 mask 的 update) is that one of the user-level threads finishes whatever they were doing and are ready to enable the signal mask for that signal again. Because the threading library knows that it has already disabled all of the kernel-level signal masks. It will, at that point, have to perform a system call to go into the kernel and update one of the signal masks, so it appropriately reflects that now the process is capable of handling the signal. The solution for how signal handling is managed and what kind of interactions happen between then kernel and the user-level library is in the same spirit of optimizing the common case. Signals, actual signals, occur much less frequently than the need to be safe and update the signal mask. So whenever we would have a certain critical portion of the code, we would first disable and then again enable the signal. And in most of those cases, a signal doesn't really occur. So then we tried to make the common case cheap, so the updates of the signal mask, we just apply them to the user-level signal mask, and the actual system call that's necessary to reflect that change in the kernel is avoided. As a result of that, we have to make

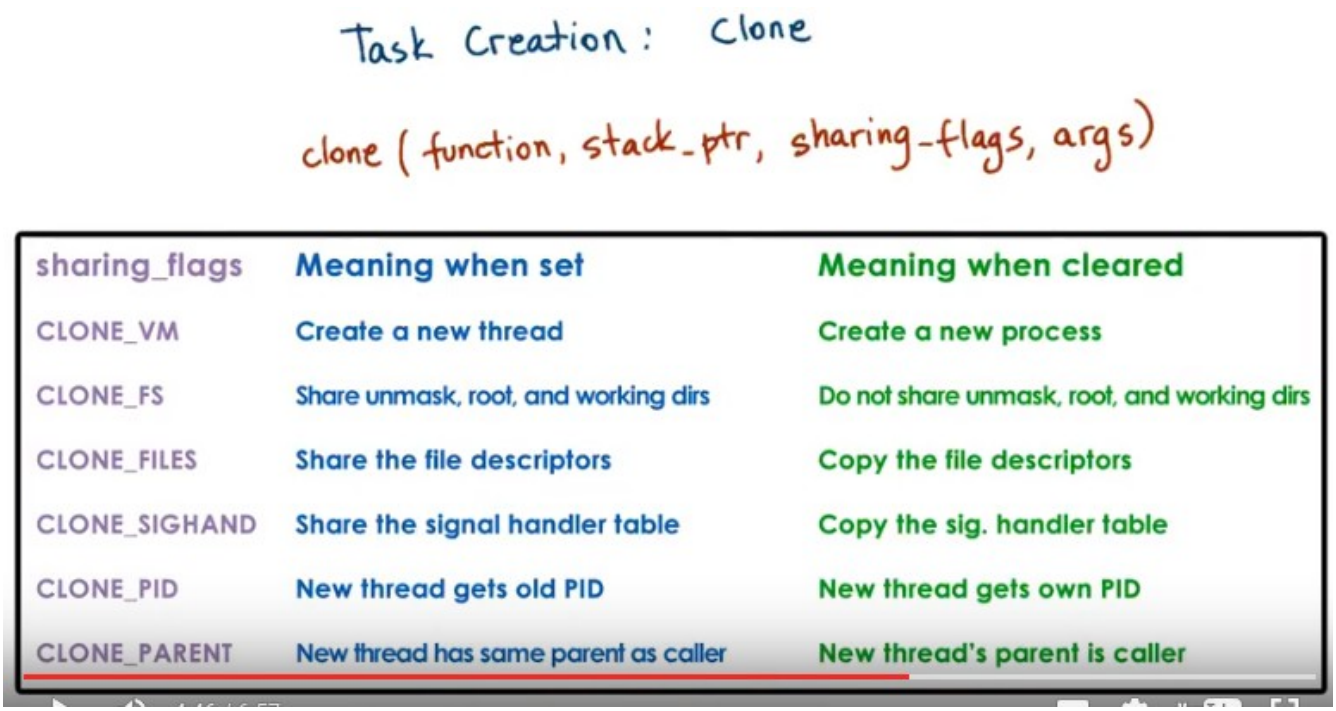
the actual signal handling a little bit more complex. But hey, that's the less frequent of the two, and we want to optimize the common case.



37. Finally, let's look at some aspects of the threading support in Linux as well. Note that the current threading support in Linux has a lot of lessons learned based in large part on earlier experiences with threads. Such as, the experiences that are presented in the two Solaris papers were described. Like all operating systems, Linux has an abstraction to represent processes (即也有 process). However, the main abstraction that it uses to represent an execution context is called a task (即 thread 叫 task). And it's represented via corresponding task structure. A task is essentially the execution context of a kernel level thread. A single-threaded process will have one task, and a multi-threaded process will have many tasks. One per thread.

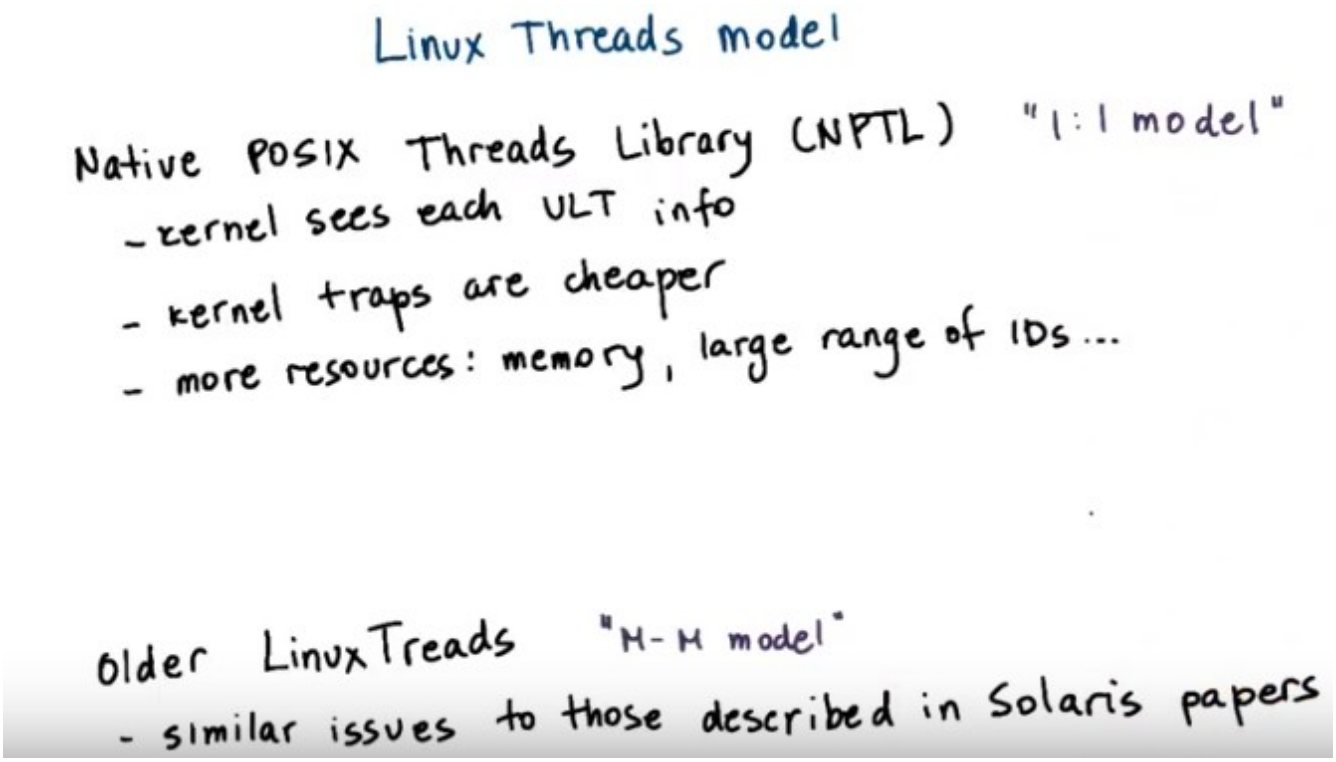


Some of the key elements in a task structure are shown here. Each task is identified by a task identifier, however for historic reasons we call this a pid like a process ID. It's a little bit misleading. What this means is that, if we have a single thread of process that has one task, then basically the task ID and the process ID are the same. If we have a multi-threaded process, then we will have multiple tasks. Each will be identified by its own identifier for the task, and that will be held in the process ID. Now the process as a whole, basically the entire group of tasks, will be identified by the process ID of the very first task that was created when the process was first created. This information is also stored in the task group ID field (tgid). In addition a task structure maintains a list of tasks (即那個 struct list_head tasks). So this basically links all of the tasks that are part of a single process it's all of threads of the process. And so one can figure out what the process ID for a group of tasks is also by walking through this list. Having learned from implementation efforts like the Solaris threads implementation Linux never had one contiguous process control block like what we described at the start of this course. Instead the process state was always represented through a collection of references to data structures, like the memory management (* mm), file management (* files). These are all referenced via pointers. So this makes it easy for tasks in a single process to share some portions of the address space, like the virtual address mappings or files. And in that case these pointers would simply point to the same memory-management structure or file-structure. There are a number of other fields in the task structure. It's a data structure that's approximately 1.7 kilobytes large so there's quite a lot of information in it.



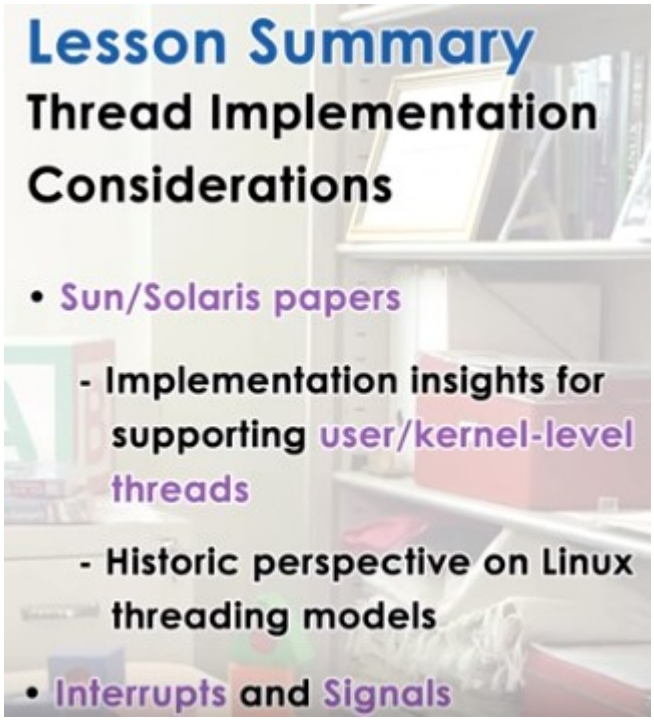
To create a new task, Linux supports an operation clone, and this is similar to this. It takes a function pointer and arguments similar to like what we saw when we were creating a new thread. But it also has this one argument that's called sharing_flags. The flags parameter is a bit map that specifies which portion of the state of a task will be shared between the parent and the child task. As you can see the values of these flags can have different effects when they're set versus when they're cleared. For instance, when all of the flag bits are set, then we're really creating a new thread that shares everything, the entire address space and everything else with the parent thread. They're part of the same address space. If all of the sharing flags are clear, then we're really not sharing anything between the child and the parent. And this is more similar to what we saw happens when we're forking a new process. And

in some cases various combinations make sense. For instance you may want to share the files or something else between the parent task and the child task. Speaking of fork you should know a couple of things. First of all fork in Linux is internally implemented via clone by basically having these flags cleared. And also fork has a very different semantics in Linux and compliant OS's in general for multithreaded processes versus single threaded processes. So for a single threaded process when we're forking we're really expecting that the created, that the child process will be a full replica of the parent process. Where as with mult-threaded processes the child will be a single threaded process. So we're really going to create a replica of the portion of the address space that's visible from the parent thread. >From the parent task in the process that called the fork. This has a lot of implications on some issues related to synchronization, to what happens with mutexs. It's beyond the scope o f the class, but I should just make sure that, that you're aware of this. So that's why I'm bringing it up at this point.



The current implementation of Linux Threads is called Native POSIX Threads Library, NPTL. And this is a one to one model where there is a kernel level task for each user-level thread. This implementation replaced an earlier implementation, Linux threads, which was very similar to the many to many model that was described in the Solaris papers. And it suffered from the same kind of complexity regarding signal management, etc. In NPTL because of the one to one model the kernel sees every user level thread. Sees all of its information, whether it's block synchronization, what is its signal mask, everything. This is made possible for two reasons. First, the kernel traps have become much cheaper. So the user to kernel level crossing that we've been trying to avoid in part with this many to many model has become much faster and we can afford to go through the kernel and update the kernel level signal map. Also modern platforms have more memory so there really isn't some constraint to keep the number of kernel level threads as small as possible. So we can create as many kernel level threads and, as the process needs. There aren't restrictions on the range of IDs that are too stressing for most of the common processes. So these sorts of things eliminate some of the main reasons for going to the many to many model. Still, however, when we start thinking about extremely large number of threads, and as a community this is something that comes up in the context of exascale

computing. Or when we are thinking about thread management in platforms that are really complex, that maybe have different kinds of processors, heterogeneity, et cetera. Then it makes sense to start thinking again about user-level library support, about providing more custom policies for how threads are managed, how threads, how many threads are there going to be in the system, or similar issues. But for most practical purposes, the one to one model that's supported by the current Linux threading model is completely sufficient.



38. In this lesson, we reviewed two older papers that gave us some historic perspective and some insights into the challenges related to supporting threads. A major takeaway from these papers is that now we have a better understanding as to why current operating systems like Linux have their present day threading model. In addition, in this lesson, we also introduced interrupt and signals to important notification mechanisms supported in operating systems today.

39. As the final quiz, please tell us what you learned in this lesson. Also, we'd love to hear your feedback on how we might improve this lesson in the future.