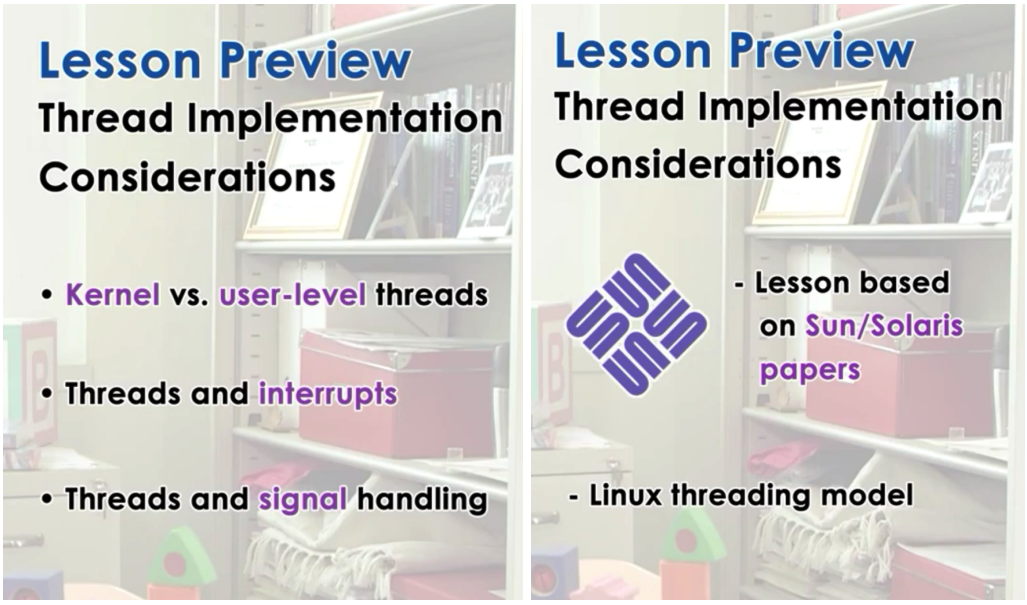
P2L4 - Thread Design Considerations

01 - Lesson Preview

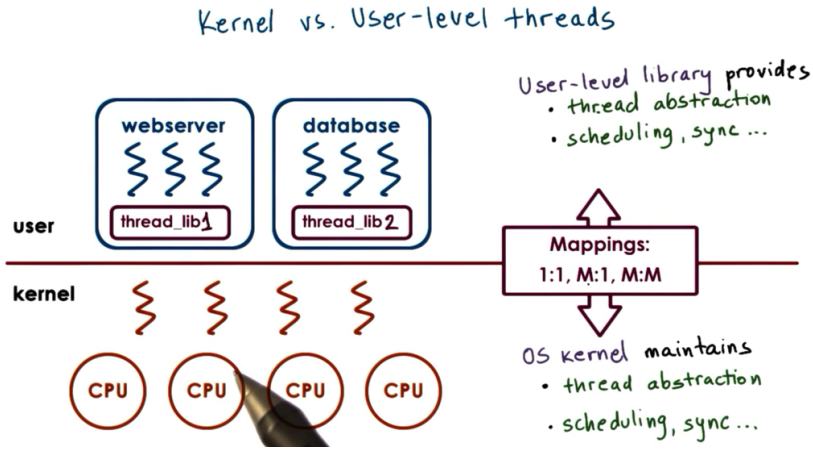


Instructor Notes

Sun/Solaris Papers

- "Beyond Multiprocessing: Multi-threading the Sun OS Kernel" by Eykholt et. al.
- "Implementing Lightweight Threads" by Stein and Shah

02 - Kernel vs User Level Threads

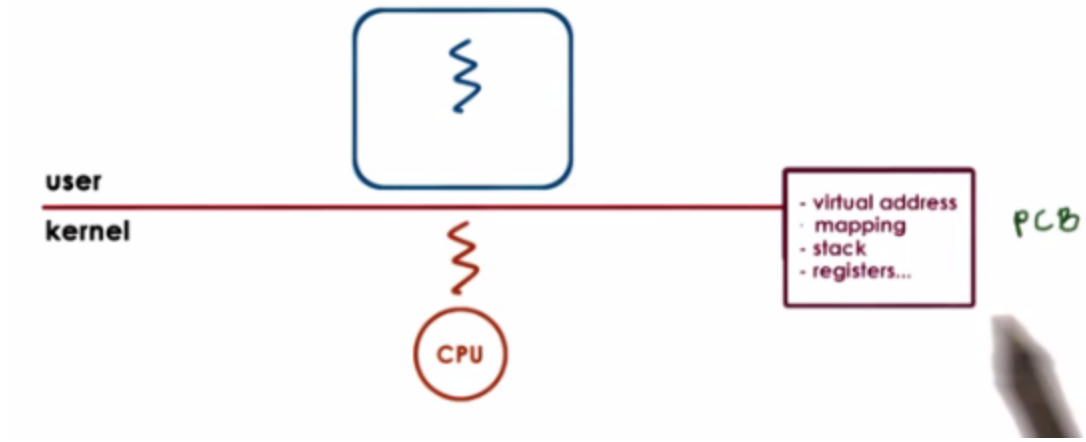


Let's start by revisiting the illustration we used in the threads and concurrency lecture. There we explained that threads can be supported at user level, at kernel level, or both. Supporting threads at the kernel level means, that the OS kernel itself is multithreaded. To do this, the OS kernel maintains some abstraction, for our threads of data structure to represent threads, and it performs all of the

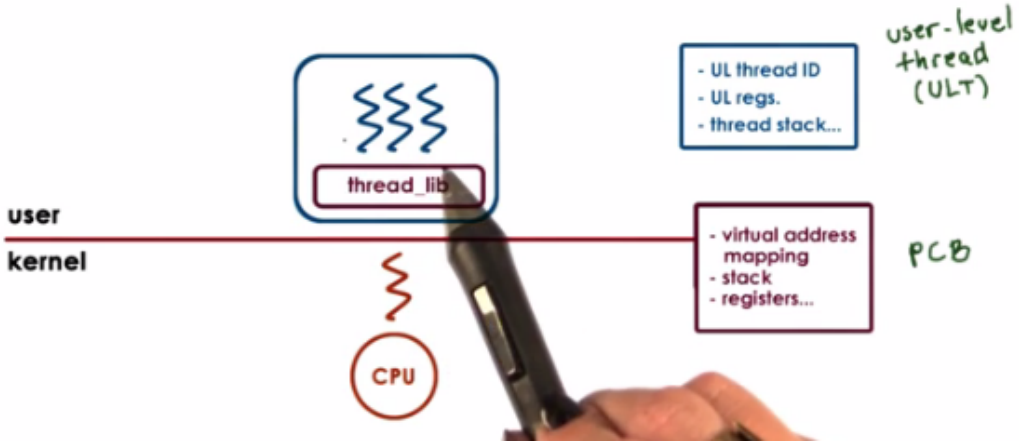
operations like synchronizations, scheduling, et cetera, in order to allow these threads to share the physical resources. Supporting threads at the user level means that there is a user-level library, that is linked with the application, and this library provides all of the management in the runtime 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.

03 - Thread Data Structures Single CPU

Single threaded process. All of the information for this process is contained in the PCB. Whenever the process makes a system call, it traps it into the kernel. Assuming 1 CPU.

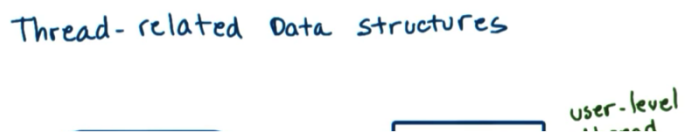


Many to One: Here the PCB contains the virtual address mappings as well as the stack and register information for the kernel level thread.



Many to Many: We don't want to recreate all of the information in PCB for the multiple kernel level threads so we split the PCB. The PCB will still contain the virtual mappings, while another data structure will keep information about the stack and register pointers of the kernel level threads.

To the user level threads, each thread looks like a CPU. The user level thread library has to make decisions such as scheduling on to the underlying kernel level threads.



04 - Thread Data Structures At Scale

If there are multiple processes, then we need these data structures for each process. There must be relationships kept amongst the different data structures. The user level thread maintains a pointer to the PCB and vice versa, while the kernel level thread also keeps a mapping to the PCB and vice versa.

If there are multiple CPUs, then the KLT maintain pointers to which CPU it is running on via a shared CPU data structure.

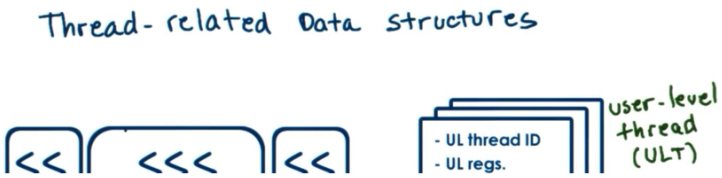
When the kernel itself is multithreaded, multiple KLTs support single UL process. When the kernel needs to context switch amongst kernel level threads that belong to different processes, it can quickly determine they point to different PCBs (i.e. different virtual address mappings), and therefore it can easily decide it needs to completely invalidate the existing address mapping and restore new ones. In the process, it will save the entire PCB of the first KLT, and if it context switching to another one, it will restore the PCB of the other one.

05 - Hard and Light Process State

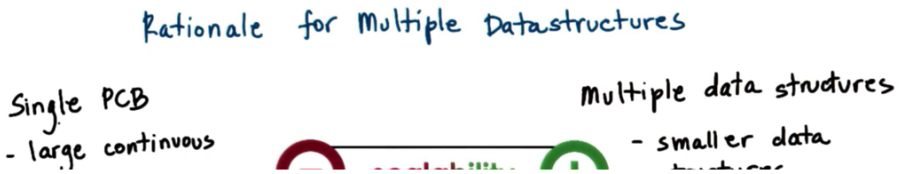
If two kernel level threads point to the same address space, there is certain information in PCB that is applicable to all threads in that process, but there is also information in PCB specific to that KLT (i.e. signals/sys call args). It depends on which user level thread is currently running, so the thread_library directly impacts this.

We split the PCB into two PCB state structures.

- Hard process state: information that applies to all threads within a process (virtual address mappings)
- Light process state: information specific to a certain thread (signal mask/sys call args)



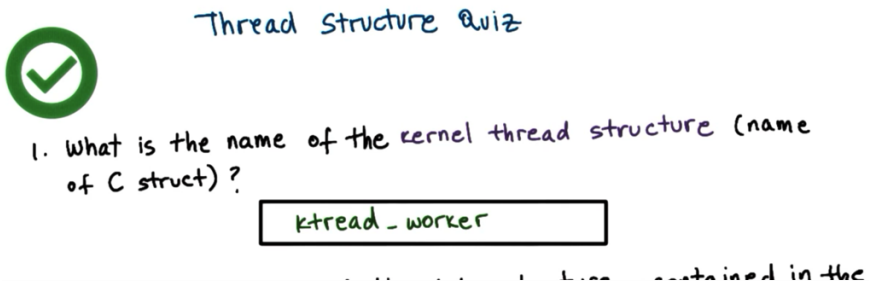
06 - Rationale for Data Structures



07 - Thread Structures Quiz

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.

08 - Thread Structures Quiz



If you browse the kthread.h header file, you will see in line 66 that there is a structure kthread_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.

Links

- Free Electrons Linux Cross Reference
    - Use the Identifier Search and find the source files in which structures are **defined**
- Interactive Linux Kernel Map

Feedback:

- The structure we are looking for has the word **kthread** in it; try looking in the kthread.h header file
- It is likely you will find the answer to Question 2 before Question 1
- For help with Question 1, try searching **"task_struct kthreads"**
- For help with Question 2, try searching **"Linux structure for processes"**

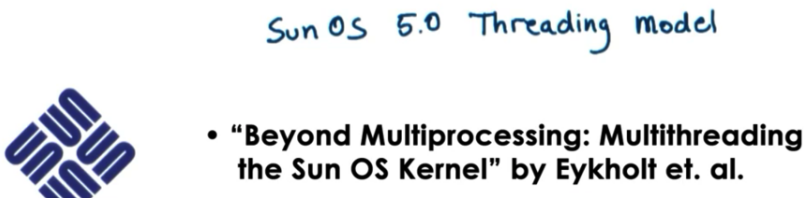
Hints:

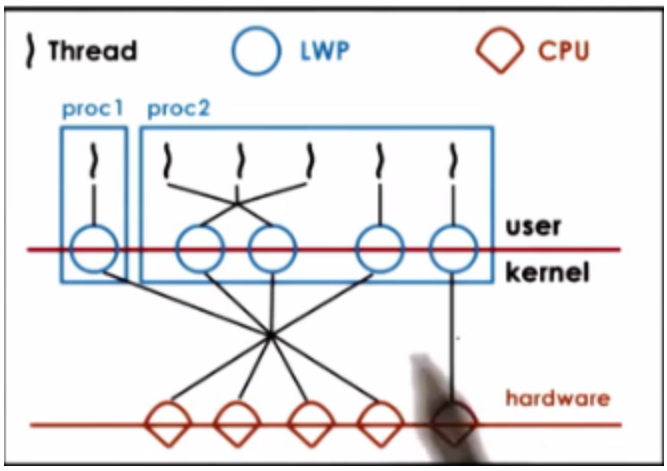
- Before consulting a Linux Cross Reference, try searching for **"Linux kernel thread structures"**
- See the Instructor Notes for help!

Errata

ktread_worker is misspelled; it should read **kthread_worker**. You can find this struct defined in Linux Cross References such as free-electrons.com.

09 - User Level Structures in Solaris 20





Supports many-to-many and one-to-one relationships. OS intended for multiple processor systems. The kernel level is multi-threaded. User level, processes can be single or multithreaded.

Each kernel level thread that's executing a ULT has a lightweight process (LWP) data structure associated with it. To the user level library, the lightweight data structures represent the virtual CPU on which it will be scheduling its threads on. Kernel level scheduler manages kernel level threads and schedules them on physical CPU.

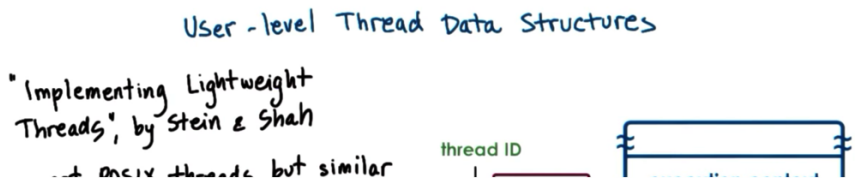
ULT: Stein and Shah
In this model, when a thread is created a thread ID is returned. This thread ID is not a pointer to the data structure of the thread but rather an index within a table. The index of that table serves as the pointer to the actual thread data structure. This is useful because if the thread data structure were corrupt, we would not be able to retrieve any information by use of the pointer.

The thread data structure contains a number of fields. In particular, the thread local storage contains the variables that are defined in the thread functions which are known at compile time. The compiler can allocate private storage on a per thread basis. Stack size might be defined by library defaults or user provided.

The point is, the size of the ULT data structure is known at compile time therefore the thread data structures can be created such that it is contiguous in memory. This would improve locality and memory access (faster speed!).

However, stack growth is not controlled by threading library! It does not interfere with what is written on the stack. Likewise, OS doesn't know the existence of multiple ULTs.

Stack growth is dangerous because there exists no management of the size of the stack. As the stack grows larger, it can actually overwrite the memory of another data structure. This makes debugging difficult. A solution to this is the red zone. If a thread begins attempts to write in the red zone, an error will be raised. Since the thread ID is part of thread data structure, the root cause can be identified which helps in debugging.



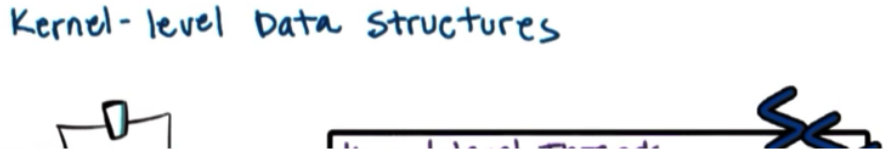
Sun/Solaris Papers

- "Beyond Multiprocessing: Multithreading the Sun OS Kernel" by Eykholt et. al.
- "Implementing Lightweight Threads" by Stein and Shah

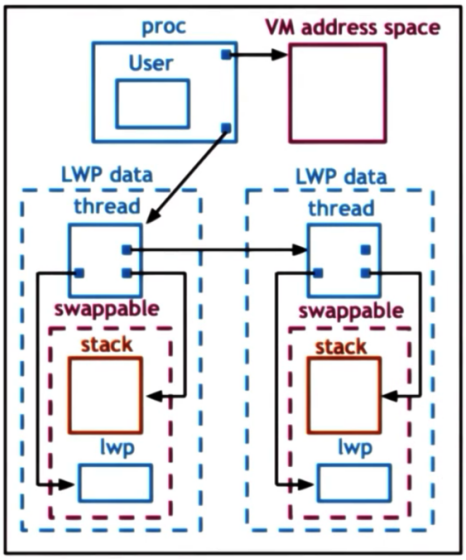
Sun/Solaris Figures

- Stein and Shah Paper, Figure 1
- Eykholt Paper, Figure 2

10 - Kernel Level Structures in Solaris 20



Kernel-level
Data
Structures
"Beyond Multiprocessing:
Multithreading the
SunOS kernel",
Eykholt et al.



- *signal handlers = information about how to respond to certain events that occur in OS
- LWP has information regarding some subset of the process: i.e. info regarding one or more ULTs in the process.
- *NOTE TO OTHER USERS* this section can be improved greatly

11 - Basic Thread Management Interaction

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.

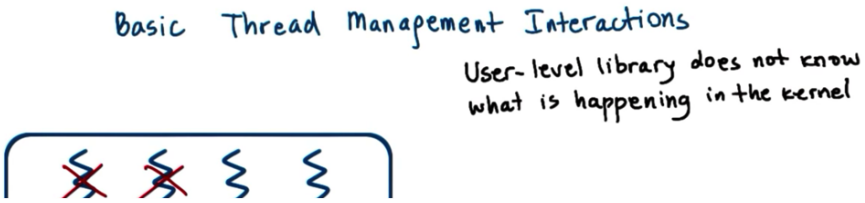
Basically, if you look at the process, it always happens that two of its user-level threads are waiting on I/O, and then some other two are actually executing. So, if our operating system has a limit on the number of kernel threads that it can support, it would be nice if the user-level process actually 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.

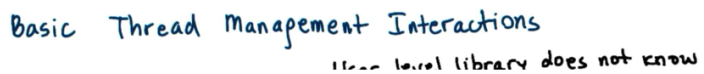
Then the process will request additional kernel-level threads, and the way it's done is that the kernel now supports a system call called **set_concurrency**. In response to this system call the kernel will create additional threads and it will allocate those to this process.

Now let’s 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. 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 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, 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 - PThread Concurrency Quiz

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.

13 - PThread Concurrency Quiz

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.

Feedback:

- If you are having trouble getting started, try searching **"sets concurrency level"**

Hint:

- For Question 1 (the function name), make sure to only write the name of the function -- do not include any arguments, etc.

Errata

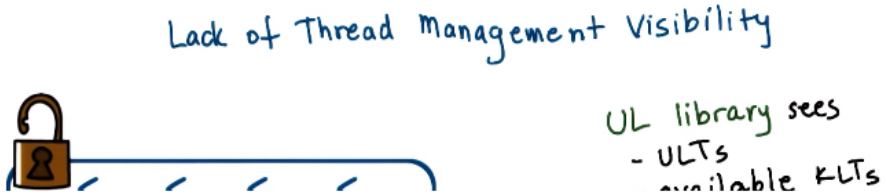
ptread_setconcurrency() is misspelled; it should read **pthread_setconcurrency()**. Read more about the function on the setconcurrency() man page.

14 - Thread Management Visibility and Design

In the previous slide we talked about the fact that the kernel and the user level library don't have visibility into each other's activities. The kernel sees all the kernel level threads, the CPUs, and the kernel level scheduler (which is the one 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 user level library will

essentially see many kernel level threads. But it will be the kernel that will actually manage those. Even if it's not one to one model, the user level library can request that one of its user level threads can be bound to a kernel level thread. This is similar if you in a multi CPU system. If a particular kernel level thread is to be permanently associated with the CPU, except in that case we call it thread pinning. The term introduced with the Solaris threads was that a user level thread is bound to a kernel level thread. Clearly in one-to-one model every user level thread is bound to a kernel level thread.

Now in one situation, one of the user level threads has a lock, so basically the kernel level thread now is supporting the execution of that critical section code (this is shown by the two locks in the image below).

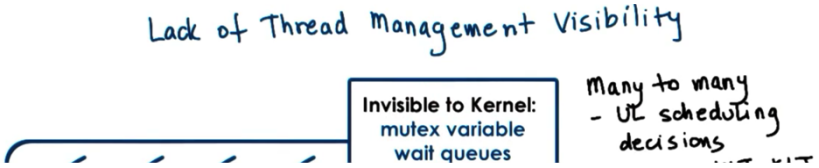


Consider when the kernel preempts a particular kernel level thread to schedule another one on the CPU.

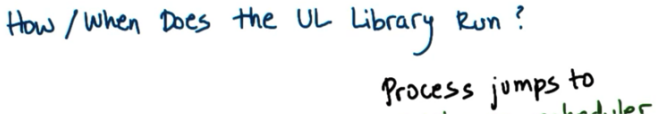
So the execution of this critical section cannot continue. As the user level library scheduler cycles through the rest of the user level threads, if they need the lock, none of them will be able to continue.

So only after at the kernel level schedules this thread again, the critical section will be completed, the lock will be released. And so subsequently the rest of the user level threads will be able to execute.

There is a visibility issue between the kernel and user level thread management. This is because at the user level the library make scheduling decisions that the kernel is not aware of and that will change the user to kernel level mapping. Data structures like mutex and wait queues are also invisible to the kernel.**The lack of visibility caused the situation such that the one that we described really leads us to the conclusion that we should look at one-to-one model to address some of these issues.**

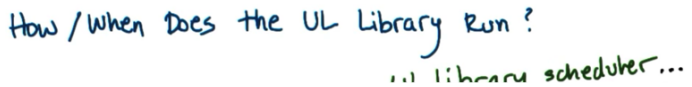


Since the user level library plays such an important role in how the user level threads are managed, **we need to understand exactly when it get involved in the execution loop.**



The user level library is part of the user process part of its address space. And occasionally the execution 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. The user level thread may explicitly yield a timer set by user level. The thread 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 the new runnable thread that the scheduler should to allocate on the CPU. 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 not something you will explicitly see.

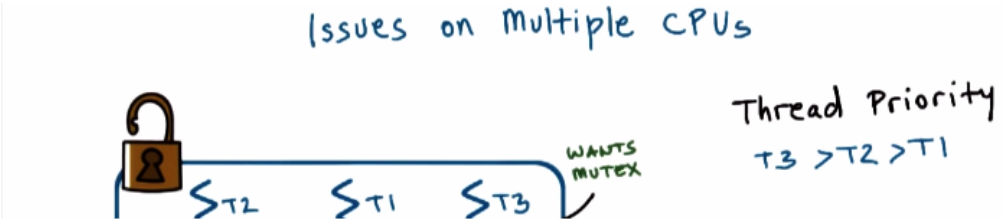
In addition of being invoked on some operations that are triggered by user level threads, the user level library scheduler is also triggered in response on certain events/signals that come either from timer or directly from the kernel.



15 - Issues on Multiple CPUs

In a multi CPU system, the kernel level thread that support a single process may be running in multiple CPUs, even concurrently. So we may have a situation where the user level library that is operating on one thread of one CPU needs to impact what is running on another thread of another CPU.

For example, we have three threads, T3, T2, and T1. The thread priority is that T3>T2>T1. T2 is currently running in the context of one thread on one CPU and currently holds a mutex. T3 with the highest priority is waiting on the mutex and is blocked. T1 is running on the other CPU. At a later point, T2 releases the mutex (but is still running on the CPU), and as a result T3 becomes runnable.



All three threads are runnable. We have to make sure the ones with the highest priority are the ones that get executed. This means T1 need to be preempted. It is the lowest priority among the three. After T2 performs the unlock operation, we will involve the user level thread library so that it schedule T3 on the CPU T1 is currently using. We need to context switch T1. However, T1 is running on a CPU, so we need to notify the CPU to update its registers and program counter. We cannot directly modify the register of one CPU when executing on another CPU.

What we need to do instead is send some kind of signal/interrupt from the context of one KLT on one CPU to the KLT on the other CPU and tell the other CPU to execute the library code locally because the library needs to make a scheduling decision and change who it is executing. Once the signal happens, the user level library on the second CPU will determine that it needs to schedule the highest priority thread T3 by blocking the lower priority thread T1.

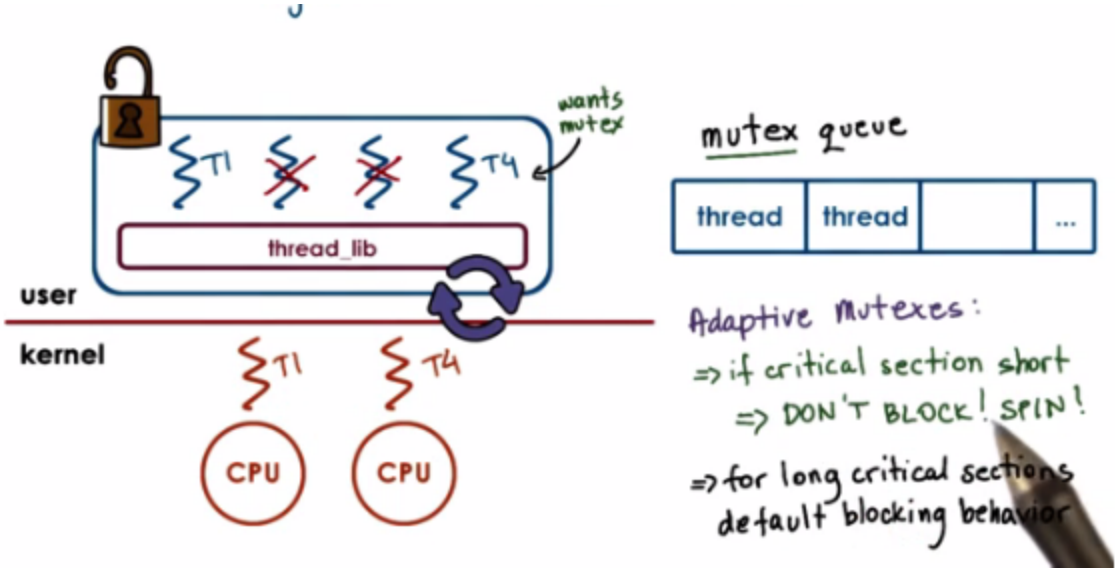
So when we have multi CPU, multi kernel and multi user level thread, interaction between the management of user level and kernel level becomes more complicated than the situation when there is only one CPU.

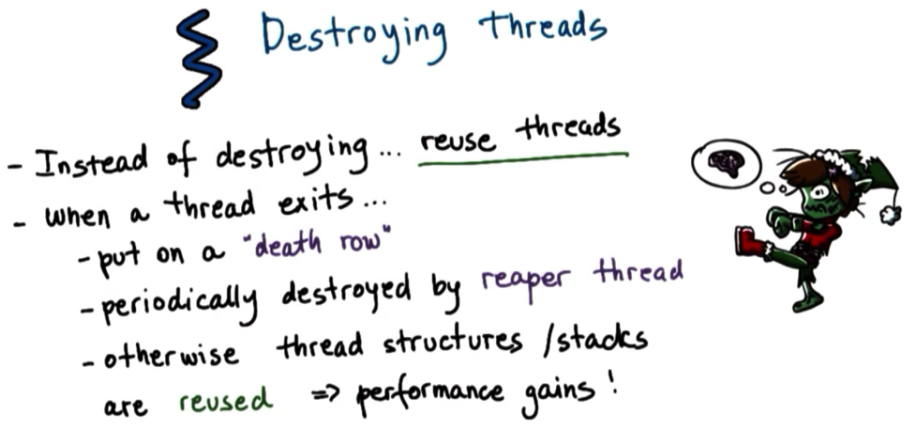
16 - Synchronization-Related Issues

Adaptive Mutexes -> spin or block (if extra CPU available and first CPU/Thread has short critical section)

Applicable only to multi-cpu systems. Consider the scenario where ULT1 is running on KLT1 and currently owns a mutex. ULT4, running on KLT4, wants the mutex. If the amount of time that T1 will take to finish is less than the amount of time T4 will take to be placed on the queue and context switch, then it makes more sense to just have T4 "spin, burn some cycles" and wait for the mutex.

Mutex data structures should contain information about the current owner. With that information, a thread can determine if the owner of a mutex is running on a separate CPU. If another CPU is available and there is a short critical section for T1, then T4 should spin instead of blocking.





17 - Number of Threads Quiz

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 Instructor's Notes for some useful pointers.

Links

- Free Electrons Linux Cross Reference
    - Use the Identifier Search and find the source files in which structures are defined
- Interactive Linux Kernel Map

18 - Number of Threads Quiz

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, init 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.

Feedback:

- If you are having trouble getting started, try searching **"Linux kernel fork"** on Google or **"fork_init"** from the Free Electrons Identifier Search

Hint:

- Once you find the correct source file, look at the **__init fork_init** function

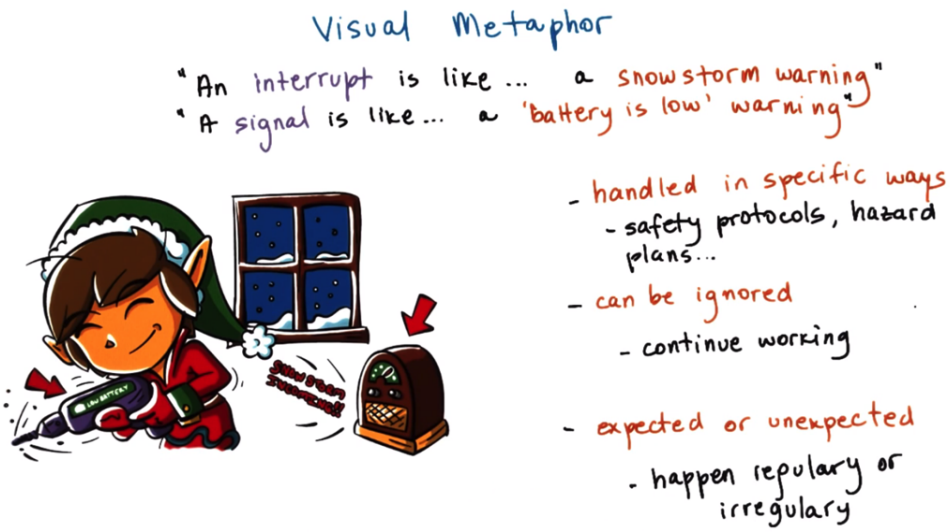
19 - Interrupts and Signals Intro

In the earlier description of data structures, we mentioned two terms that we have not yet talked about, interrupts and signals. 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 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.

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.

20 - Visual Metaphor



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.

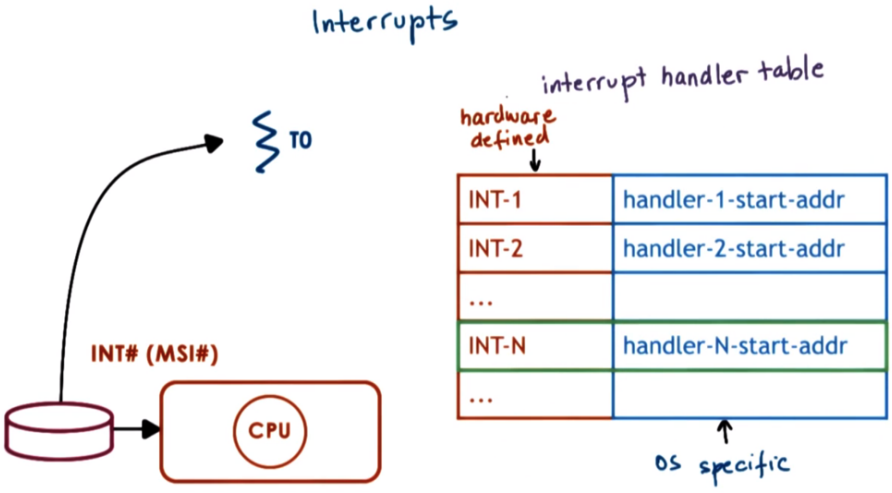
In a toy shop, handling these types of events may be specified via safety protocols or certain hazard plans. This is not uncommon. There may be, however, situations in which it's appropriate to just continue working. And finally, situations like the fact that the battery died are pretty frequent. They 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 - Interrupt Handling

Device interrupts the CPU via the wires that connect it. It sends a unique message, i.e. INT-N. The hardware defined message is then looked up in a table and the corresponding handler is called. The program counter is moved to that address of the handler code and the handler routine is then executed.



22 - Signal Handling

Basically the same as interrupt discussion above except signal is not generated by external entity. Also, the OS defines the possible signals, i.e. SIGNAL-11, instead of the hardware. There are default actions defined by the OS in response to signals, but each process can have their own defined signal handlers which respond in a user defined way in response to the OS signal.

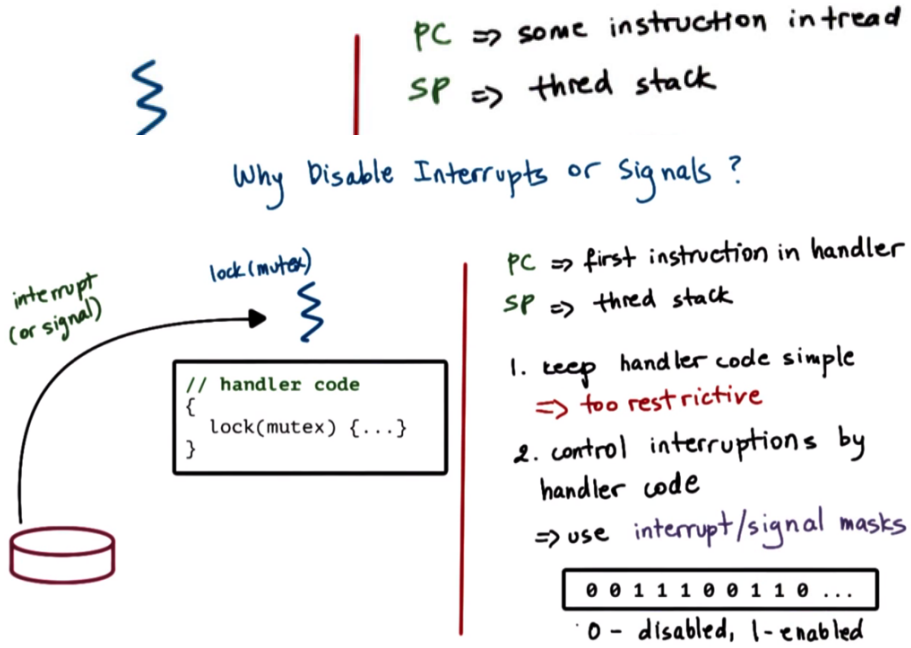
23 - Why Disable Interrupts or Signals

1 problem with interrupts/signals -> executed in context of thread that was interrupted -> handled on thread stack which leads to discussion on why it should sometimes be disabled.

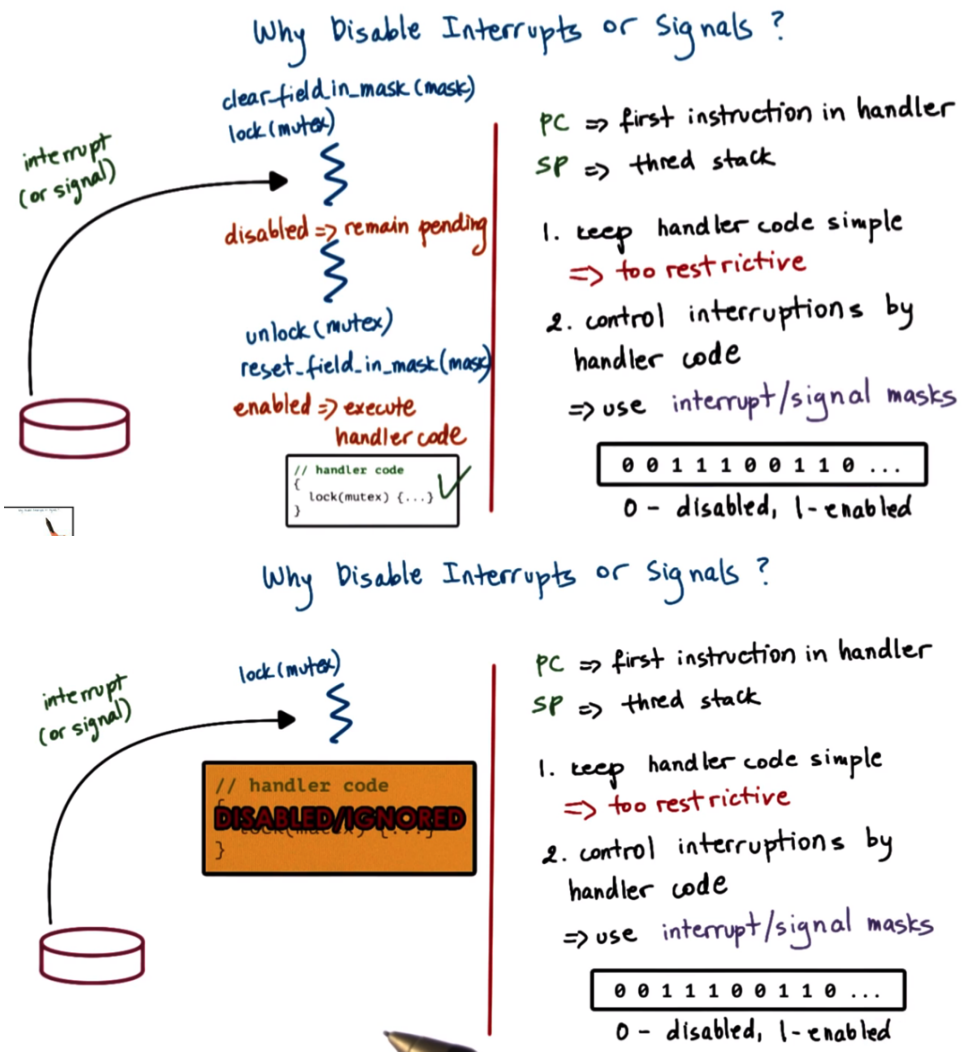
The basic logic is this: If the interrupted thread has a locked mutex, and the handler needs to also acquire the same mutex, then a deadlock will occur. This can be remedied by the thread first disabling the ability to be interrupted before it acquires the mutex. Likewise, it enables the ability to be interrupted after it releases the mutex.

The ability (or lack thereof) to be interrupted to all possible interrupts/signals is contained in an interrupt/signal mask. Note that if an interrupt/signal comes in while the ability to be interrupted is disabled, then the interrupt/signal is pending. Once the ability to be interrupted is enabled, the interrupt/signal proceeds as it should. However, if the same type of interrupt/signal occurs while the

first was pending (say N times), it is not guaranteed they will be handled (it might be that it is only handled once).

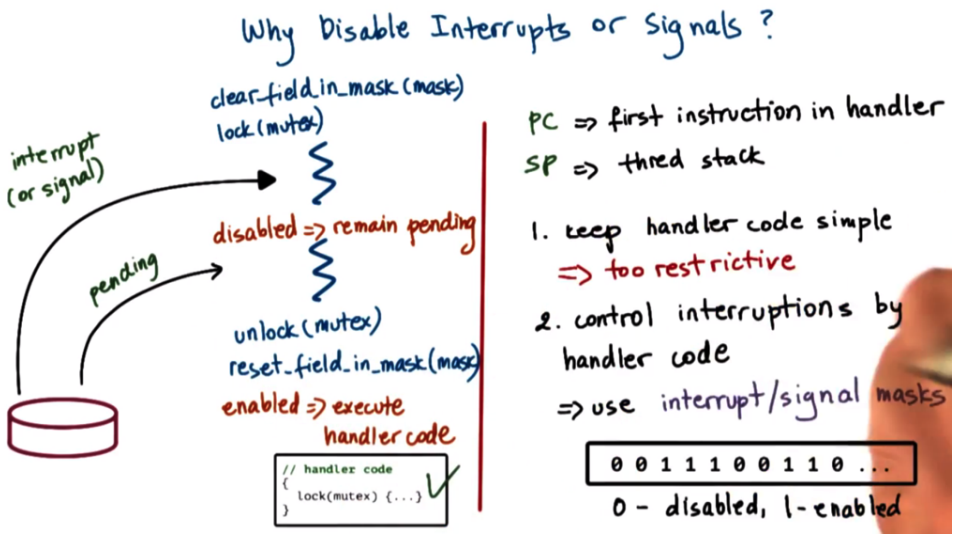


To explain why we need to disable interrupt, we provide an example. In the example, we have a thread and its PC and stack are shown in the right. In some point of the execution, an interrupt occurs. As a result, the program count will change and it will start point the first instruction in the handler. The stack pointer will remain the same. And this can be nested if there are multiple singles/interrupts. In a nested fashion, they will keep executing on the stack of the thread which was interrupted. the handling routine needs to access to some state perhaps some other thread is accessing, so we need to use mutex. However, if the thread was interrupted already has the mutex that was needed by the handler routine, we have **deadlock situation**. The interrupted thread will not release its mutex until the handler routine complete its execution on its stack. And the handler routine is blocked by the mutex. To prevent this situation, one solution is to keep the handler code simple which means we can prohibit the handler code to use mutex. If there is no possibility for the handler to be blocked on some mutex operation, then the deadlock situations will not occur. The problem with the method is too restrictive.



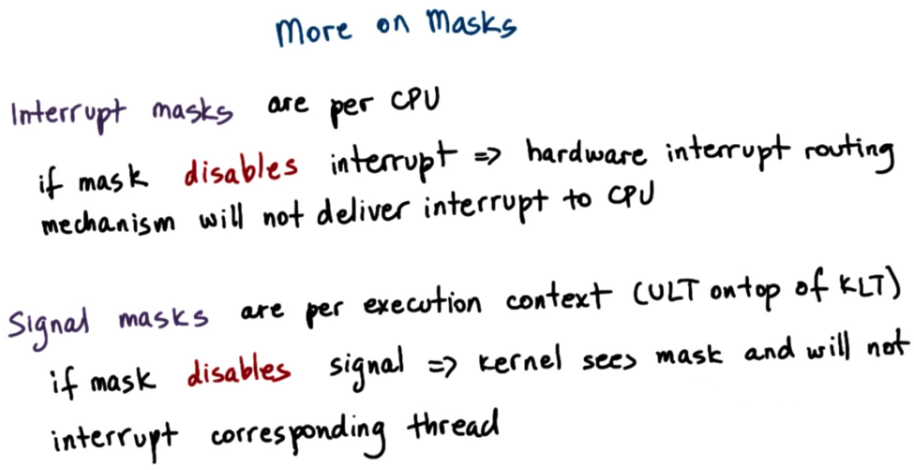
Instead of keeping the handler code simple, we introduce masks which will dynamically allow us to enable or disable whether the handling code will interrupt the mutex. We call these interrupt/signal masks. The mask is a sequence of bits where each bit corresponds to a specific interrupt or signal. And the value 0/1 indicates whether this specific interrupt or signal is disabled or enabled. When an event occurs, first the mask is checked. If the event is enabled, then we process with the handler procedure.

If the event is disabled, then the interrupt/signal will remain suspended until at a later time when the mask value changes.
To solve the deadlock situation, the thread before acquiring a mutex, it would disable the interrupt. So even when the interrupt occurs, it will be disabled and not interrupt the execution of the critical section. If the mask indicates the interruption is disabled, then it will remain pending until a later time. After the mutex is unlocked, the mask value is reset to its proper value. As a result the interrupt will become enabled. The operation will allow the execution of the handler code. Now at this time, it is ok to execute this code since we are no longer holding the mutex. The deadlock is avoided.



While an interrupt or signal is pending, other instance/s may occur. They will remain pending as well. Once the event is enabled, the handler routine will typically only execute once. So if we want to ensure the handler routine is exec'd more than once, it is not just sufficient to generate the signal more than once.

24 - More on Signal Masks

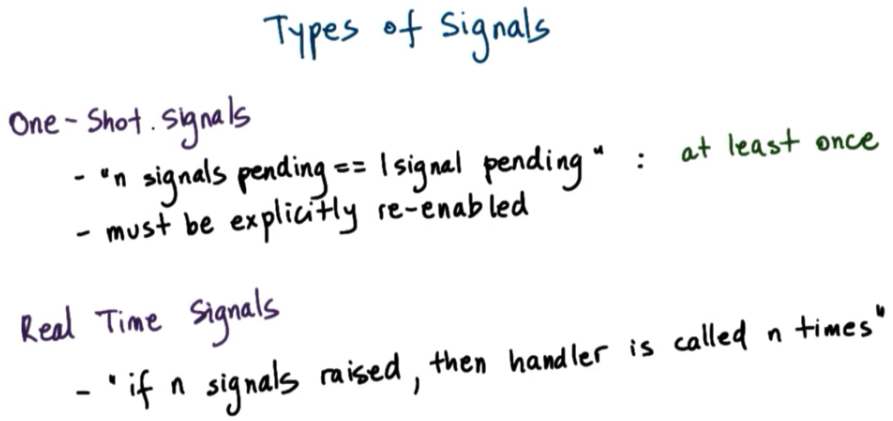


25 - Interrupts on Multicore Systems

Instead of all CPUs receiving an interrupt, let only one CPU receive it. This reduces total cost.

26 - Types of Signals

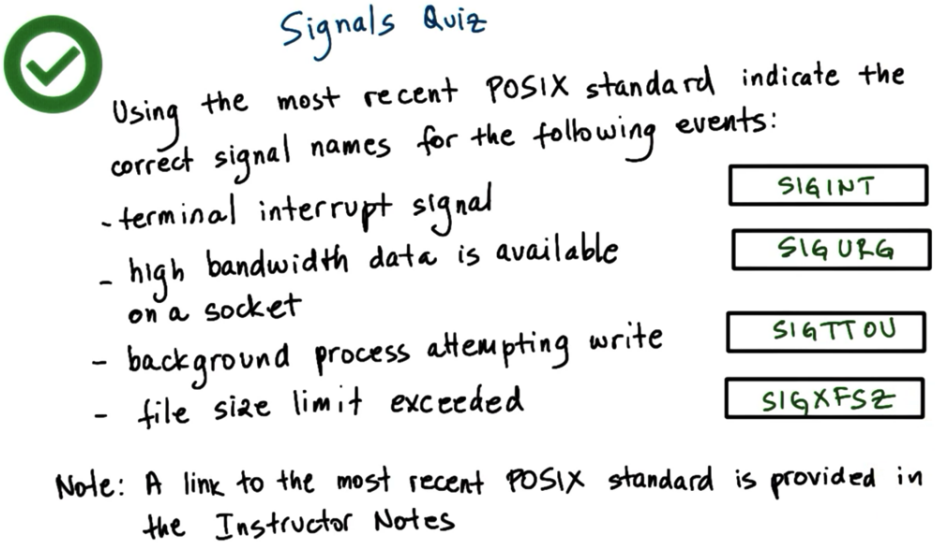
If a one-shot signal is not re-enabled, next time the particular signal comes, it will be handled be the default OS handler instead of the specific handler for that process.



27 - Signals Quiz

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 - Signals Quiz



Hints:

- For help, use the link provided in the **Instructor Notes**; once there, search for "signals"
- You may also try **searching for the specific event** (from the problem description)
- Most signal documentation includes a table for many, if not all, of the signals we are looking for

Link

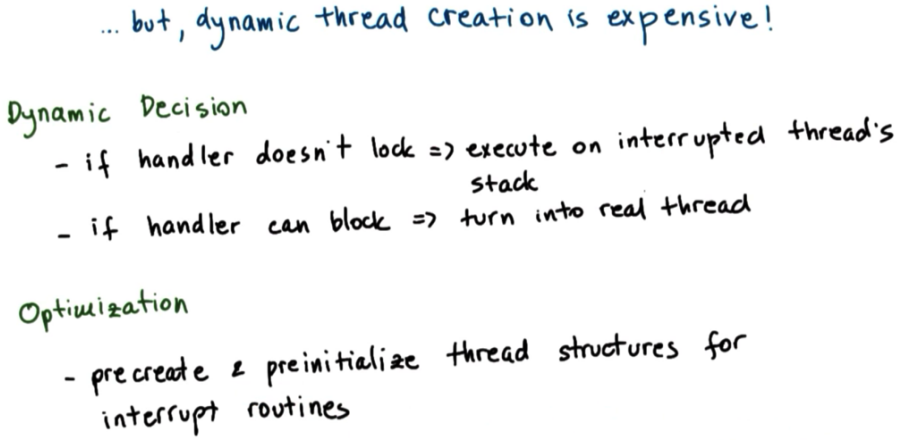
- POSIX Standard (POSIX.1-2008)

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 SIGINT. 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.

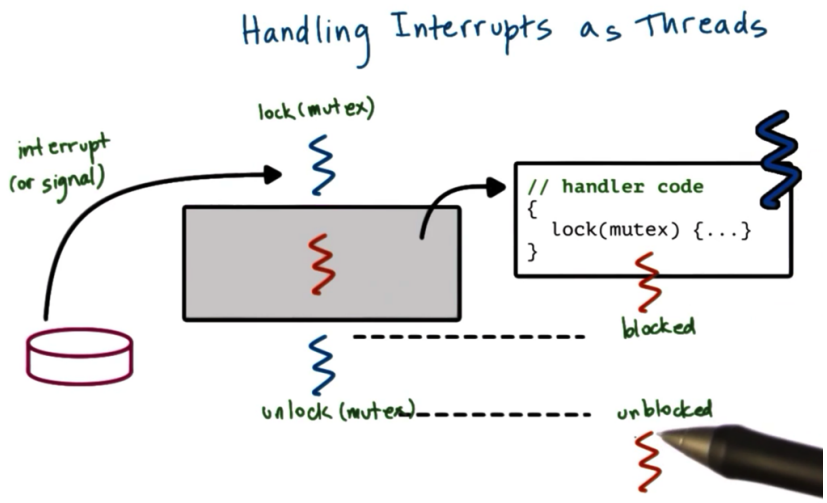
29 - Interrupts as Threads

Summary:
Any time a handler might block, i.e. it acquires mutex and executes code that needs to be synchronized, let it interrupt as a full fledged thread. This way it has it's own context and stack and can remain blocked. The interrupted thread then continues and unlocks when done. The handler then can execute its own code. The only concern with this approach is that creating threads dynamically is expensive, so have a pre-created/initialized pool of interrupt threads for the various interrupt routines the kernel supports. I.E. The thread data structure, that is created for every interrupt thread, points to the proper location of the interrupt handler.

Now let's look at the relationship between interrupts and threads. In previous example, when interrupt occur there is a possibility of deadlock. One way as illustrated in the Sun OS paper is to allow interrupts to become full-fledged threads. This should happen every time it is performing blocking execution. In this case the interrupt handler is blocked at this point. It has its own context, stack, thus it can remain blocked.



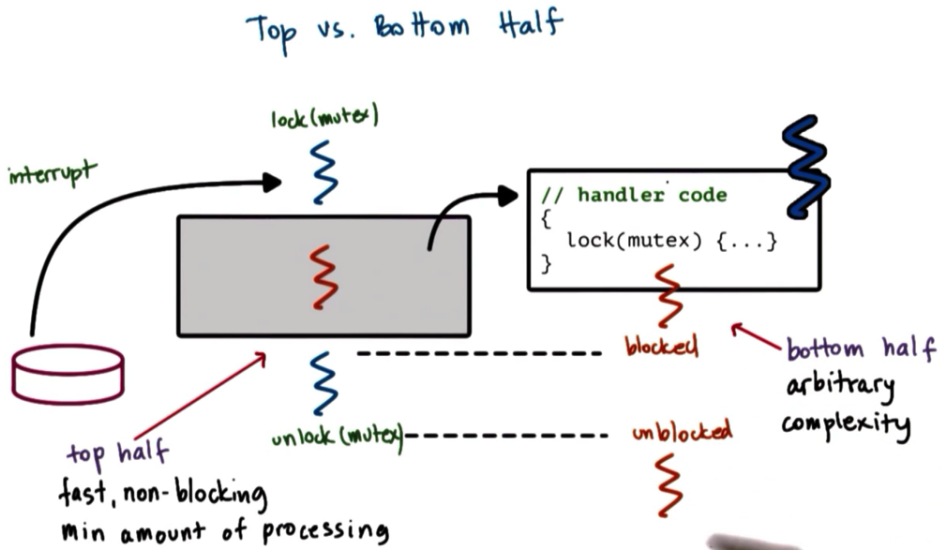
So the thread scheduler can schedule the original thread back on the CPU and that will continue executing. Eventually the original thread will unlock the mutex, and the thread corresponding to the interrupt handler routine will be executed. The way it happen is following. Whenever a signal/interrupt occurs, it interrupts a thread. By default the handling routine will start executing in the context of the interrupt thread using it stack, etc. If the handling routine is going to perform synchronizing routine, in this case, that handler code will execute on a separate thread. When the locking operation is reached, it will be placed in a wait queue associated with the mutex. The original thread will be scheduled. When the unlock operation happens, we will go back and dequeue the handler code from the mutex queue.



The problem with this operation is dynamic thread creation is expensive. The decision needs to make is whether the handler is be handed on the stack of the interrupted thread or as a real thread. The rule described in Sun OS paper is that if the handler does not include a lock, then no dynamic thread will be created. However, if there is a possibility for the thread to be blocked, then we turn it into real thread.
To limit the needs to dynamic create a thread, whenever determine there is a potential to block the thread, kernel pre-creates and pre-initialize a number of threads for the variable interrupt routine that it can support. This means that the kernel will pre-create a number of threads and their associated data structures and initialize those data structures so they point to the right point of the interrupt routine.

As a result the creation of a thread from the fast path of the interrupt processing. So we don't pay the cost when the interrupt occurs.

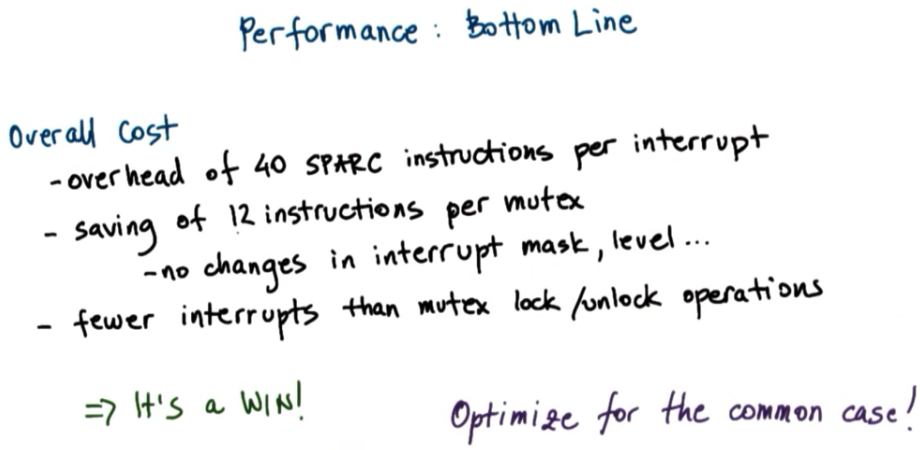
30 - Interrupts Top vs Bottom Half



When the interrupt first occurs, we are in the initial/top part of this handler. It may be necessary to disable certain interrupts to prevent deadlock situation. When the interrupt is passed to another thread, then we can enable any interrupt to its original state, because now the handler is in a different thread. So interrupt can be handled as any thread in the system.
The top will handle a minimum amount of processing. It is required to be non-blocking. The bottom is allowed to have any arbitrary/complex processing. The top half execute when the interrupt immediate occurs. The bottom half is like any other thread can be scheduled for a later time and can be blocked.

31 - Performance of Threads as Interrupts

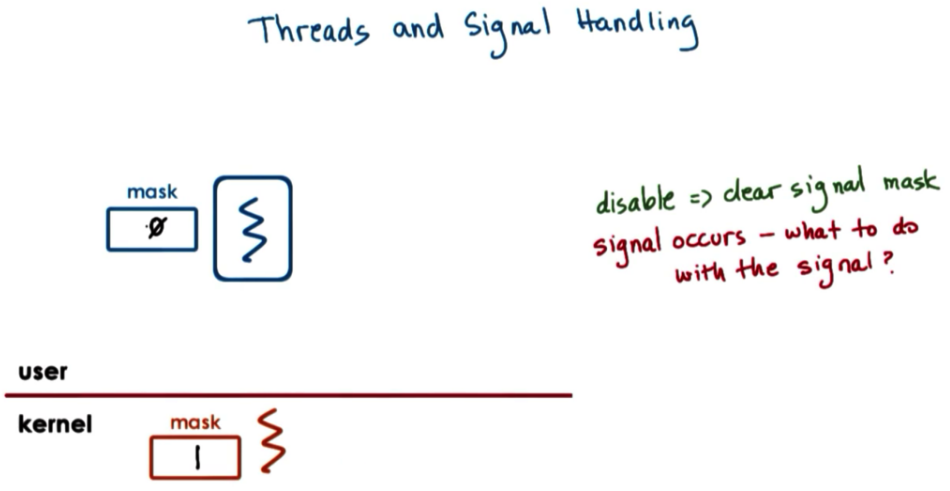
It is best to handle interrupts by using threads instead of doing the mask/unmask/check mutex/etc. ops.



*common case here = mutex lock/unlock ops, optimize here. It turns out that even though we optimized for that case, another scheme of creating a thread has better performance since we do not want to compromise the safety and operation of the system..

32 - Threads and Signal Handling

Signal masks associated with the ULT and the KLT (or attached LWP). When ULT wants to disable a mask, it changes the bit in the ULT mask to 0. Note that the ULT mask is not visible to the kernel (and vice versa). When signal arises, what does the kernel do with the signal? If the kernel visible signal mask is enabled (bit set to 1), it wants to respond to the signal. So either we need a method to cross the user/kernel barrier and set the kernel mask in this case to 0 (since user level mask is 0), **or we need to have a policy that deals with the situation where the user level and kernel level masks are different.** Next 4 morsels deal with examples of this policy.



Instructor Notes

Sun/Solaris Papers

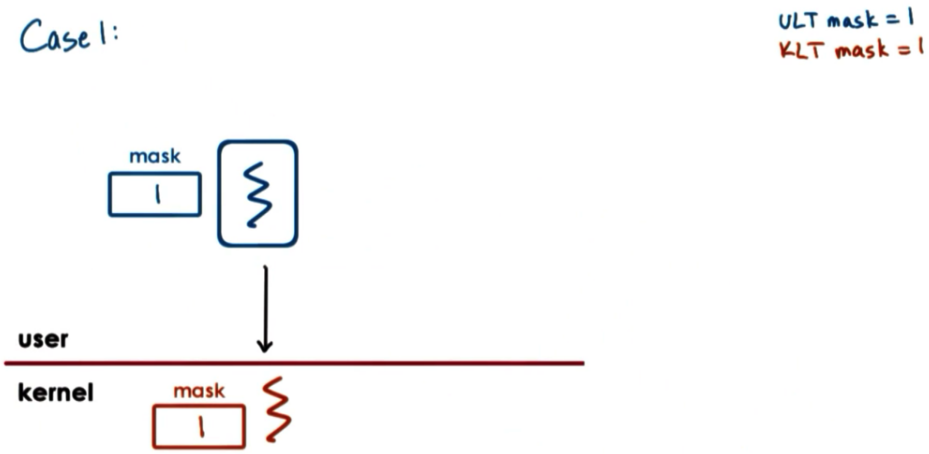
- "Beyond Multiprocessing: Multithreading the Sun OS Kernel" by Eykholt et. al.
- "Implementing Lightweight Threads" by Stein and Shah

Sun/Solaris Figures

- Stein and Shah Paper, Figure 1
- Eykholt Paper, Figure 2

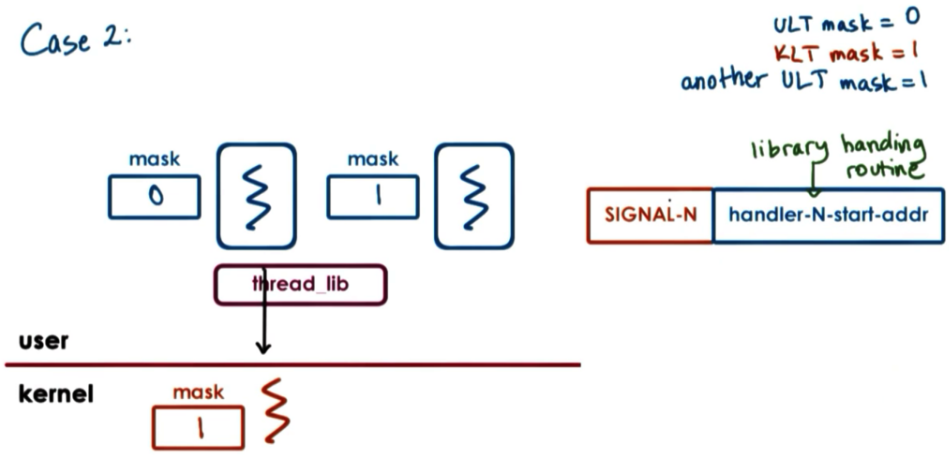
33 - Threads and Signal Handling Case 1

Absolutely no problem here. KLT will be interrupted when signal is received.



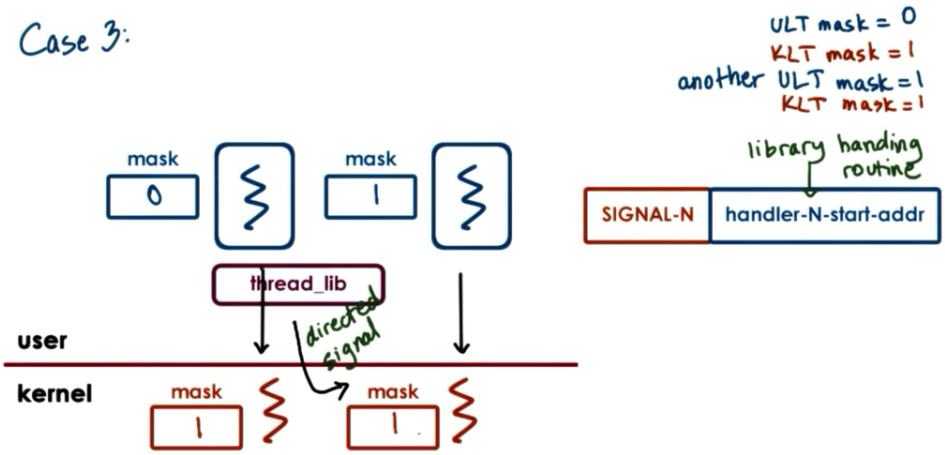
34 - Threads and Signal Handling Case 2

The ULT running on the KLT does not have the signal enabled, but another ULT does have the signal enabled. There exists a library handling routine that serves as a wrapper class for the signal handlers. When a signal occurs, the KLT runs the wrapper routine. The wrapper routine has access to all the threads running in the process so it will see that ULT2 has the mask enabled and will then switch ULT2 to be running on KLT.



35 - Threads and Signal Handling Case 3

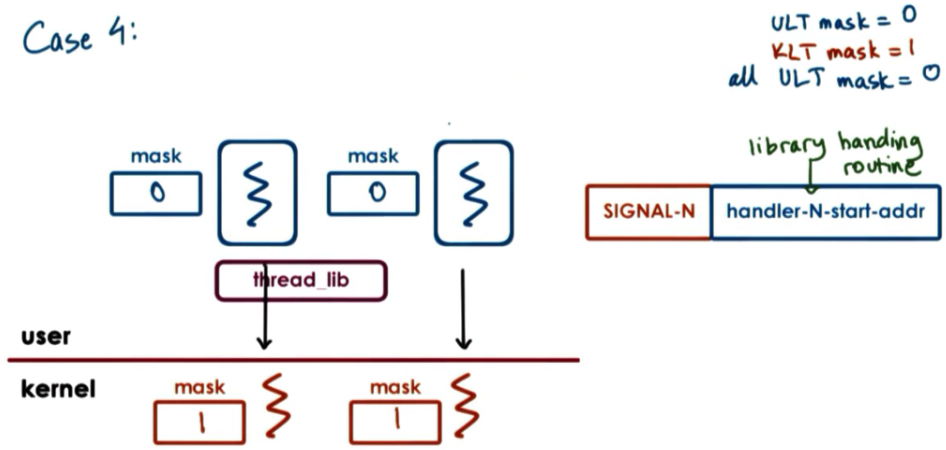
ULT1 (mask 0) is attached to KLT1 (mask 1) and ULT2 (mask 1) is attached to KLT2 (mask 1). Signal occurs on KLT1. The signal is delivered and the library handling routines kicks in and it knows that there is a ULT in the system that can handle the signal, i.e. ULT2, and it sees that ULT2 is attached to KLT2. Since the library knows this, it will generate a directed signal to the KLT2 (or associated LWP2). Since the mask of KLT2 is 1, it will send signal and the ULT library will call the library handling routine. The library handling routine will see that ULT2 has a non zero mask and then ULT2 will execute the handler.



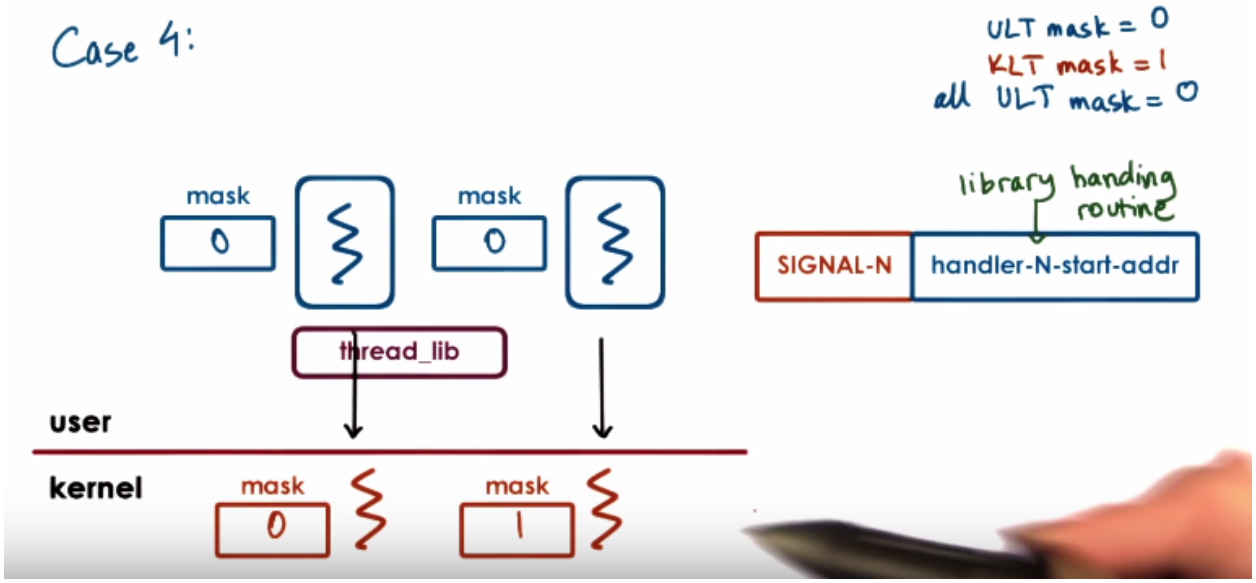
36 - Threads and Signal Handling Case 4

In this case, both ULT1 and ULT2 have masked disabled, but KLT1 and KLT2 have mask enabled. Net effect at the end of the day will be that KLT1 and KLT2 become disabled. However, once a ULT enables its mask, a KLT will also change its mask and the signal will be handled.

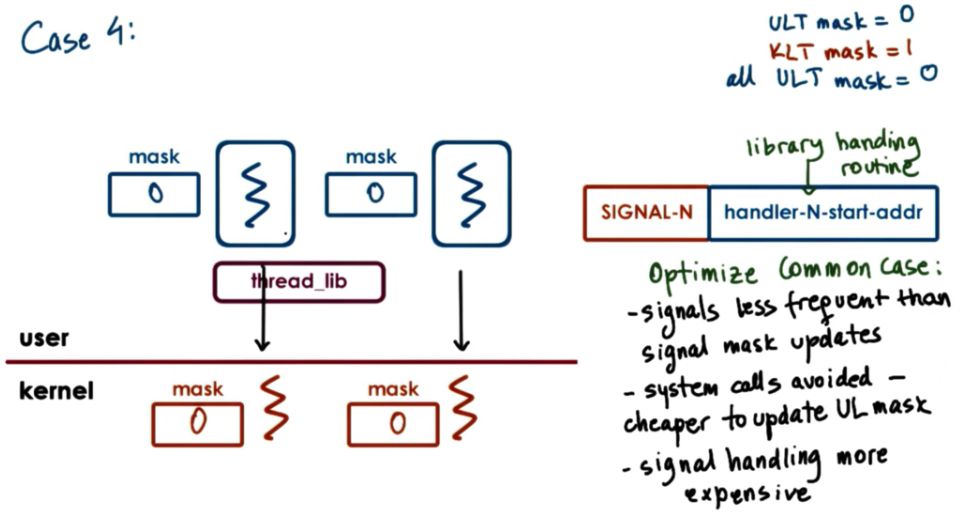
This happens as follows:
KLT1 receives signal, calls library handling routine which sees that no ULTs have an activated mask.



It then makes a system calls and changes KLT1 mask to 0.



The thread library handling routine is called again and checks the ULTs and see no one has an enabled mask. It then goes to KLT2 and disables its mask. The library handling routine then again checks for ULTs that have enabled masks.



Once a ULT is done with what it needs to do, it changes it mask status to 1. The library handling routine then would go to the KLTs and change the mask of the KLT attached the ULT (which just changed its mask setting to 1) to 1 as well. The library handling routine is then called again and it now sees the ULT with the enabled mask which it uses to handle the signal.

37 - Tasks in Linux

Task Struct in Linux

- Main execution abstraction => task
  - kernel level thread
- Single-threaded process => 1 task
- Multi-threaded process => many tasks

Task Struct in Linux

```
struct task_struct {
   // ...
   pid_t pid;
   pid_t tgid;
   int prio;
   volatile long state;
   struct mm_struct *mm;
   struct files_struct *files;
   struct list_head tasks;
   int on_cpu;
   cpumask_t cpus_allowed;
   // ...
}
```

Task Creation: Clone

clone (function, stack_ptr, sharing_flags, args)

sharing_flags	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share unmask, root, and working dirs	Do not share unmask, root, and working dirs
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the sig. handler table
CLONE_PID	New thread gets old PID	New thread gets own PID
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

Sharing flags have to do with the sharing of parent task_struct info with the child. If flags are set, then it is like creating a new thread since a new thread shares the parent's address space and only has its own stack and registers. When the flags are cleared, it is like creating a new process since a new process doesn't share the parent's information, it copies them.

Single threaded process forks a new process -> copy with same address space
Multi-threaded process forks a new process -> will only contain a portion of the parent's address space

Linux Threads model

Native POSIX Threads Library (NPTL) "1:1 model"
- kernel sees each ULT info
- kernel traps are cheaper
- more resources: memory, large range of IDs...

Older LinuxTreads "M-M model"
- similar issues to those described in Solaris papers

*more resources = refers to the fact that modern architecture has more resources so that a 1:1 model is affordable so that one doesn't have to deal with the problems of M:M model.

38 - Lesson Summary