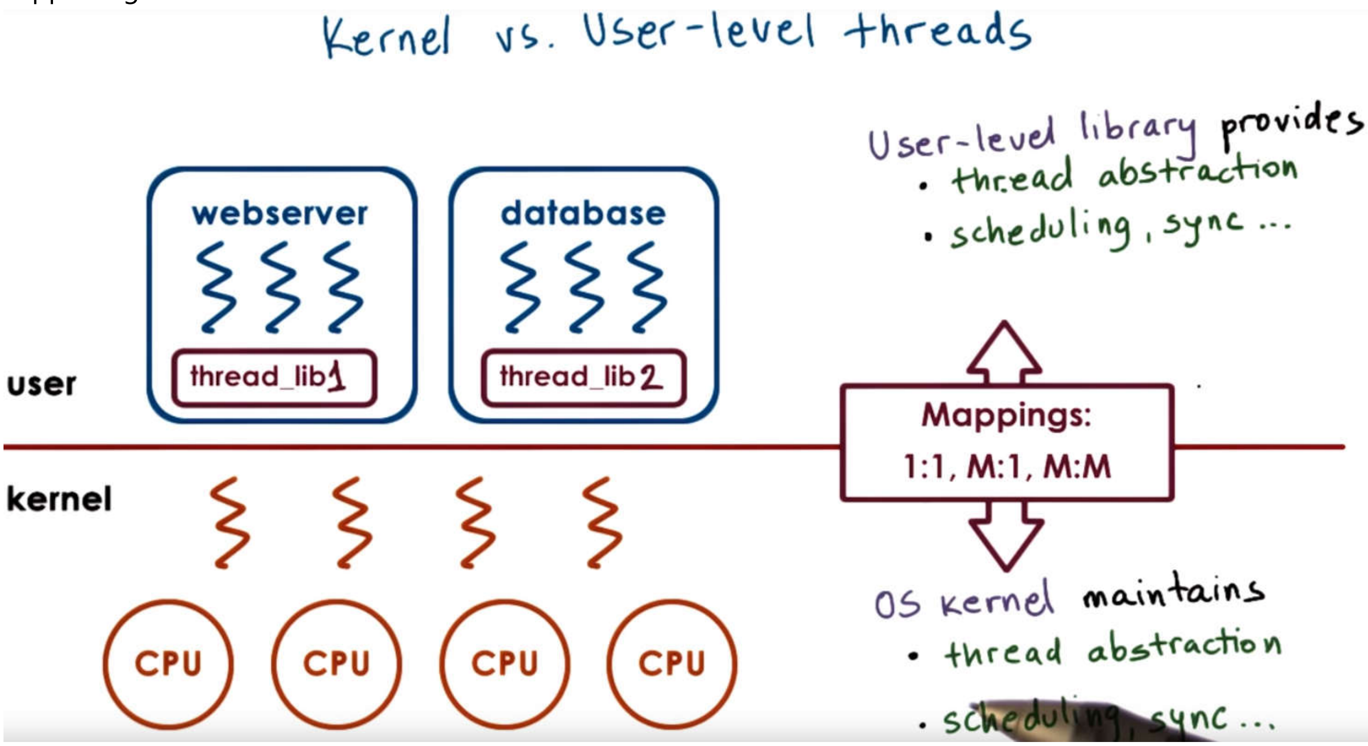
P2L4 Thread Design Considerations

Goal

- Data structures and mechanisms of kernel vs. user-level threads
- Two notification mechanisms supported by OSs:
  - Threads and interrupts
  - Threads and signal handling
- How threading systems evolve over time

1. Kernel vs. User-level threads

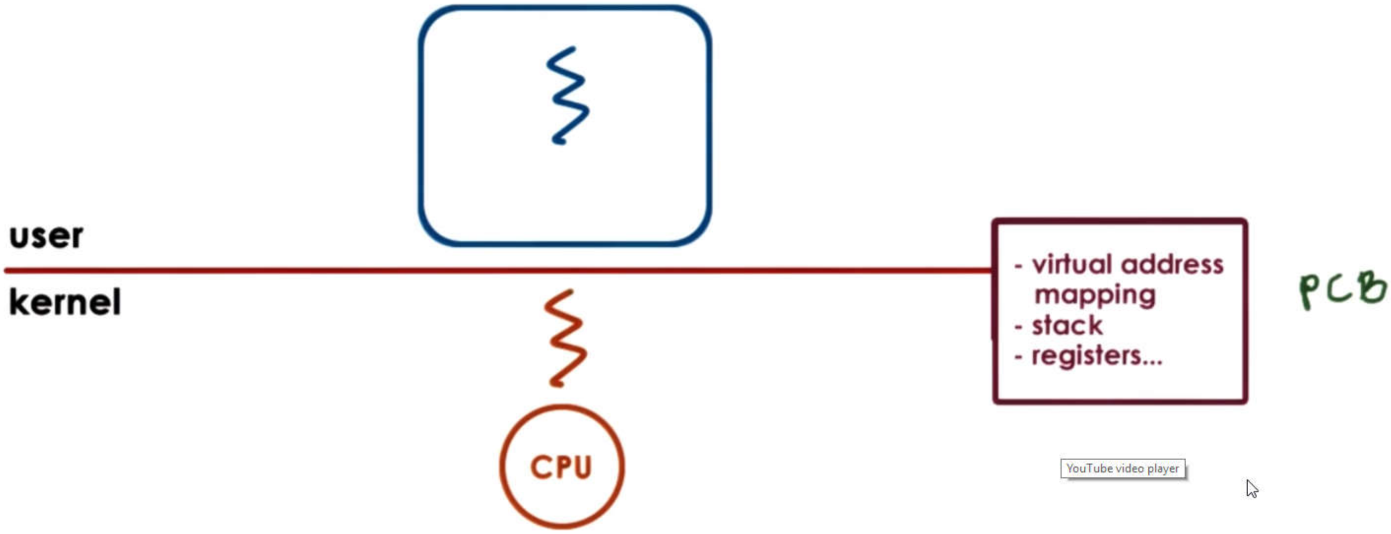
- supporting thread at the OS level means that the OS kernel itself is multithreaded



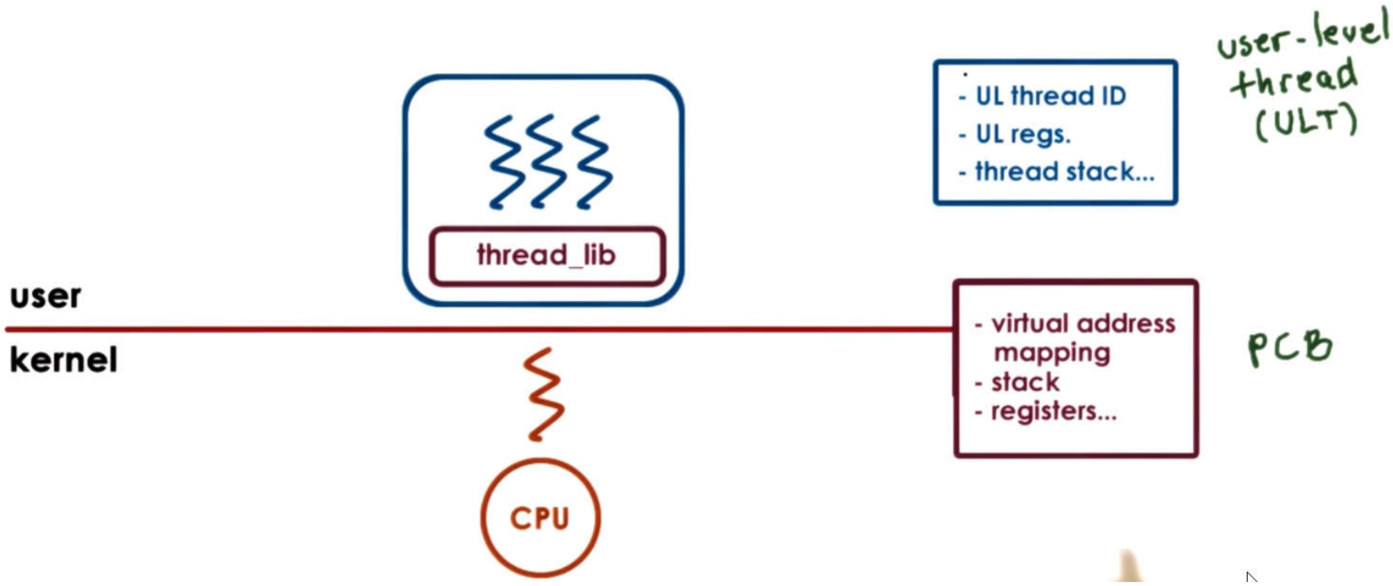
- user level threads
  - thread libs support data structures that's needed to implement the thread abstarction
  - provide all scheduling synchronization and other mechanisms
  - different threads can use entirely different thread libs.
- kernel level threads
  - OS kernel maintains thread abstraction, scheduling sync.
  - Support mapping between user and kernel threads.

1.1 Thread-related Data Structures

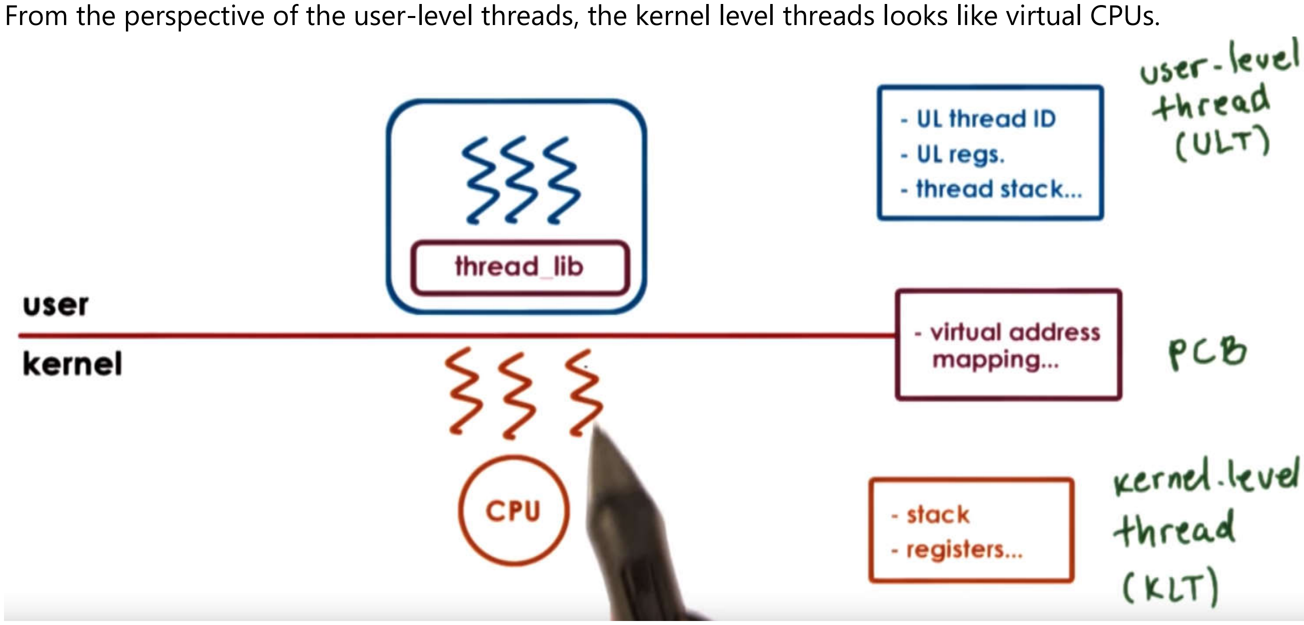
- Single thread on single process



- Multi user-threads on single kernel-thread process

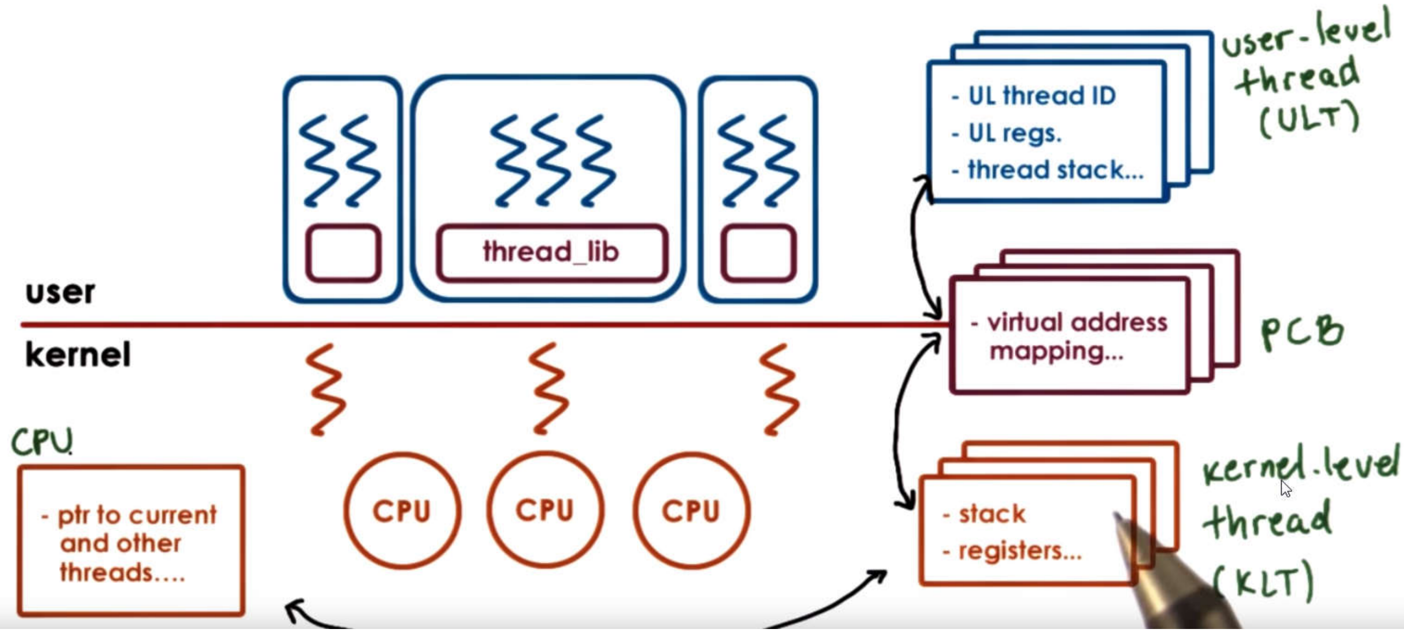


- Mulitple kernel-level threads associated with the process with multiple user-thread
    - From the perspective of the user-level threads, the kernel level threads looks like virtual CPUs.



Relationships among ULT, PCB and KTL

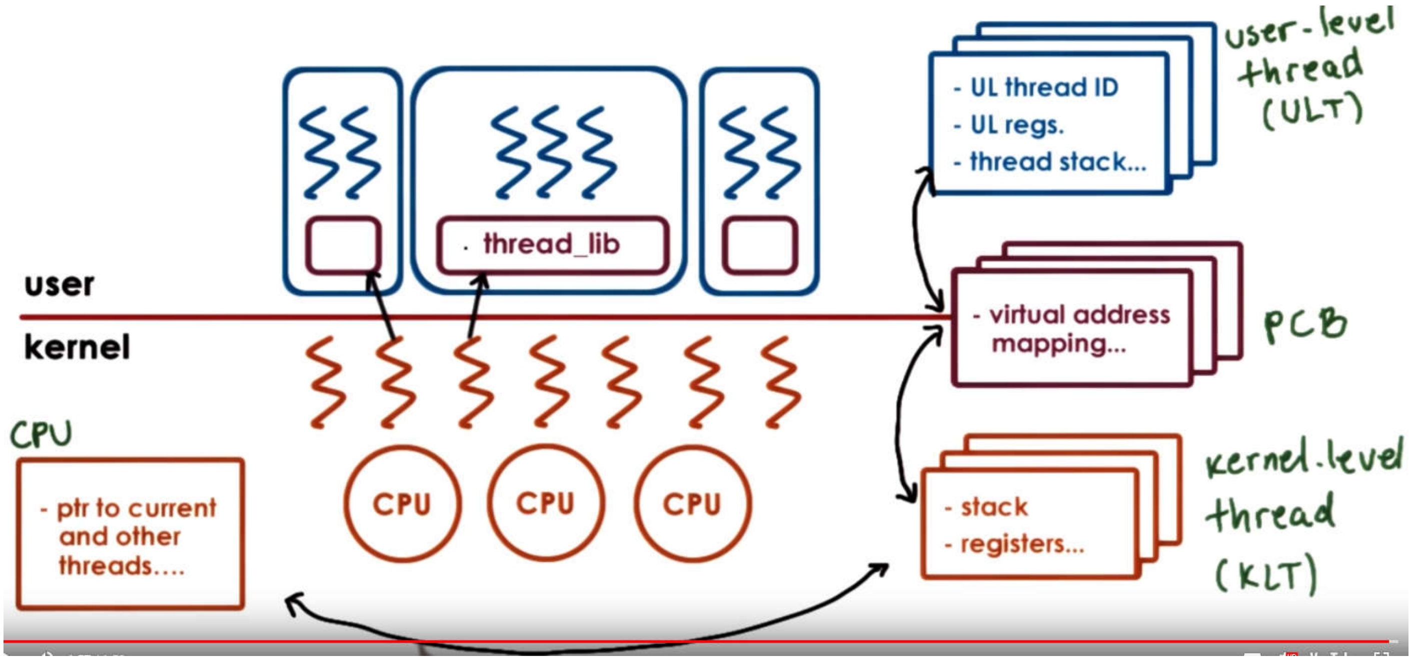
- Both the ULT and KLT has to know what is the address space within which that thread executes
- If there are multiple CPUs we have to maintain a data structure to represent the CPU
- CPU data structure has a relationship with KLTs



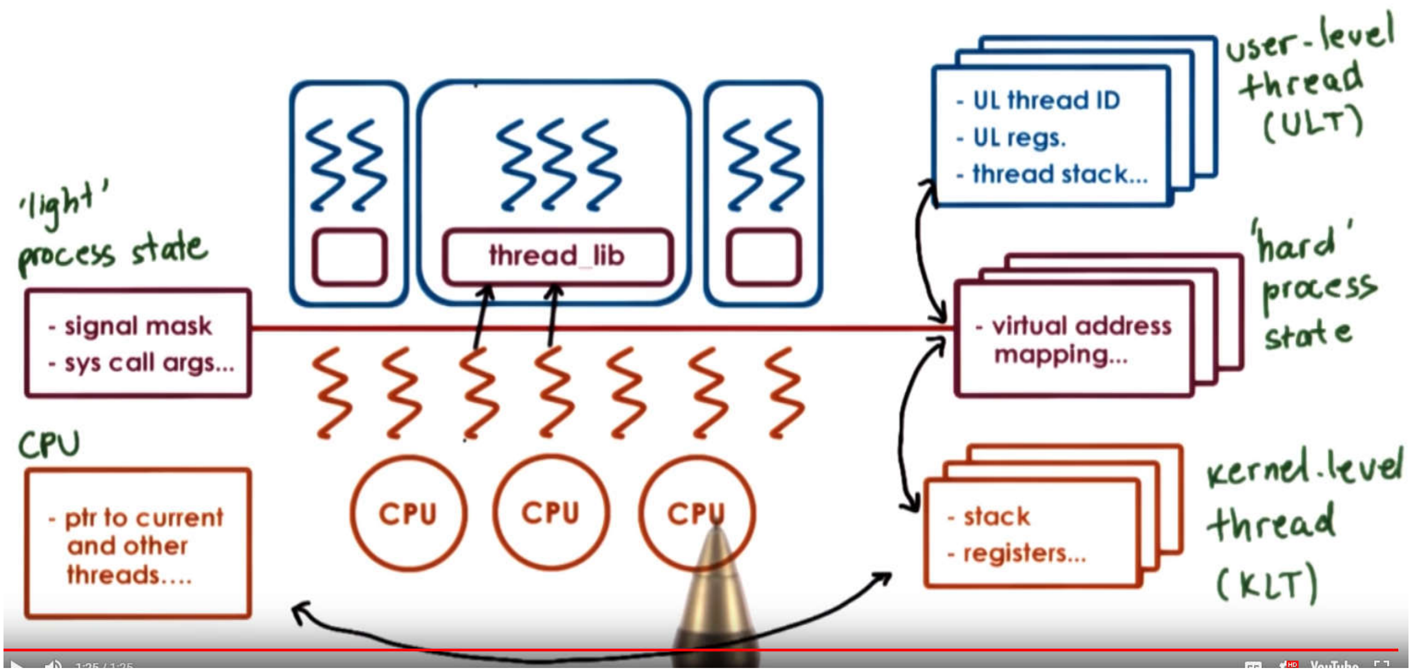
When the kernel is multi-threaded we can have multiple kernel-level threads supporting a single user-level process.

When the kernel needs to schedule/ context switch among kernel-level threads that belong to different processes, it can quicly determine that they point to a different process control block, hence different virtual address mappings.

So the kernel needs to completely invalidate the existing address mappings and restore new ones.
The kernel will save the entire PCB of the thread to be switched off and restore the PCB of the thread to be loaded to run.



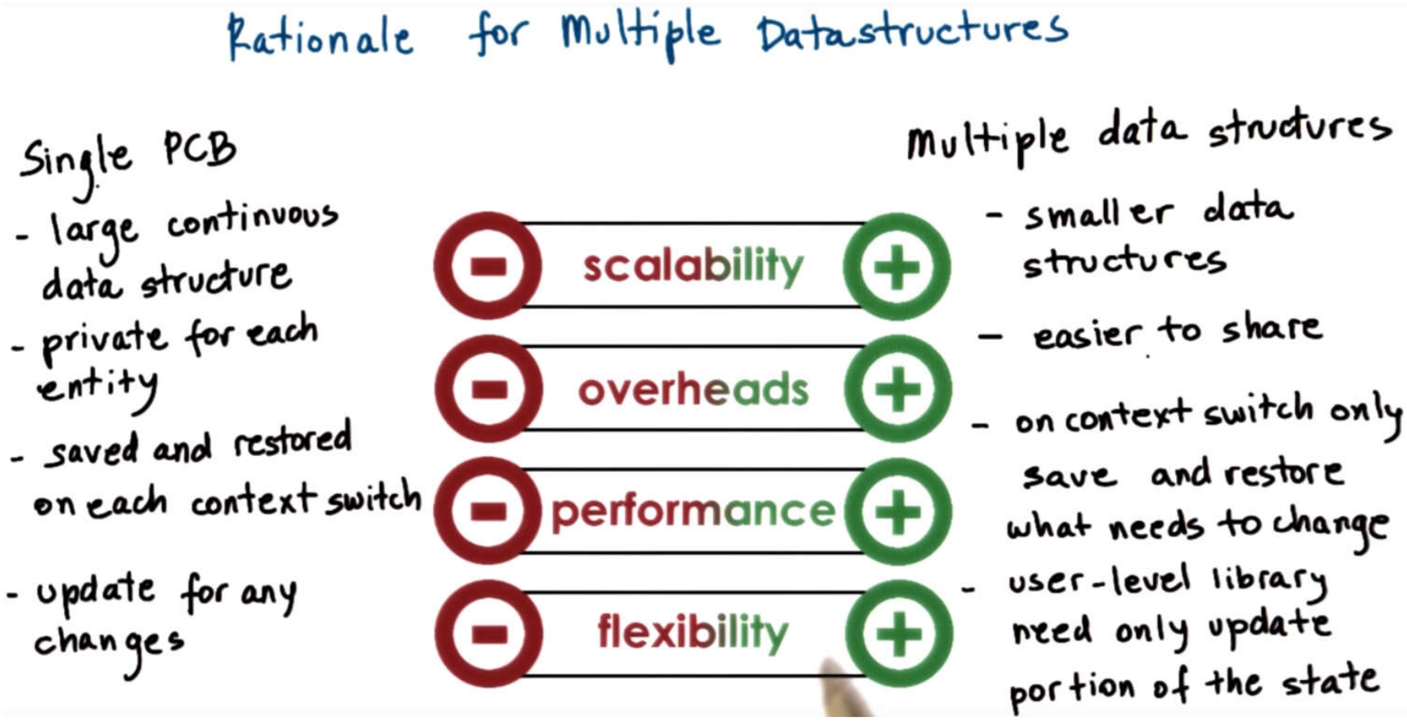
1.2 Hard and Light Process State

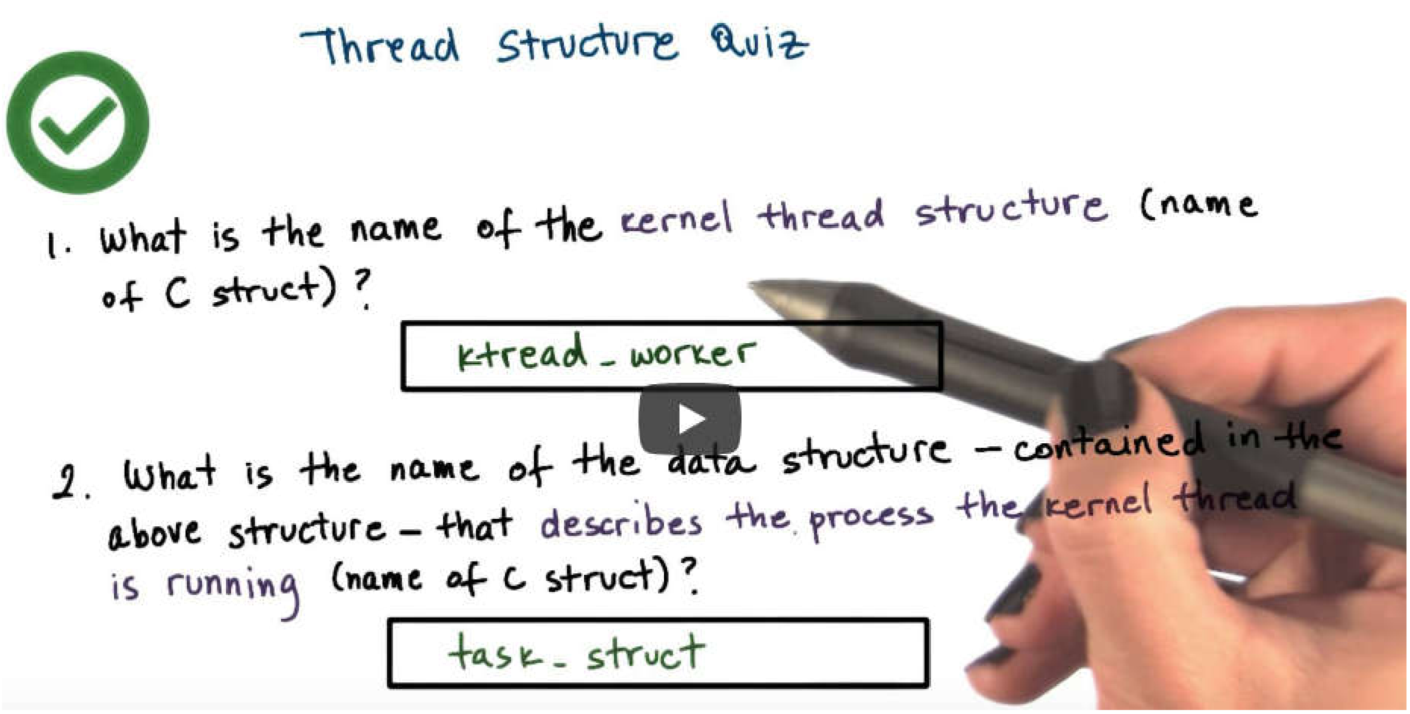


When there are multiple kernel-level threads supporting one process, hence they belong to the same virtual address mapping scheme and the kernel is doing a context switch between them, there are portions of the PCB that they share the same data and part of the PCB that are specific to each kernel-level threads (e.g. signal masks, sys call args).

- hard process state: Information in PCB that the kernel-level threads in the process share— virtual address mapping etc.
- light process state: Information in PCB that are only relavant to a particular kernel-level thread in the process, and the user-level threads that are mapped to the kernel-level thread.

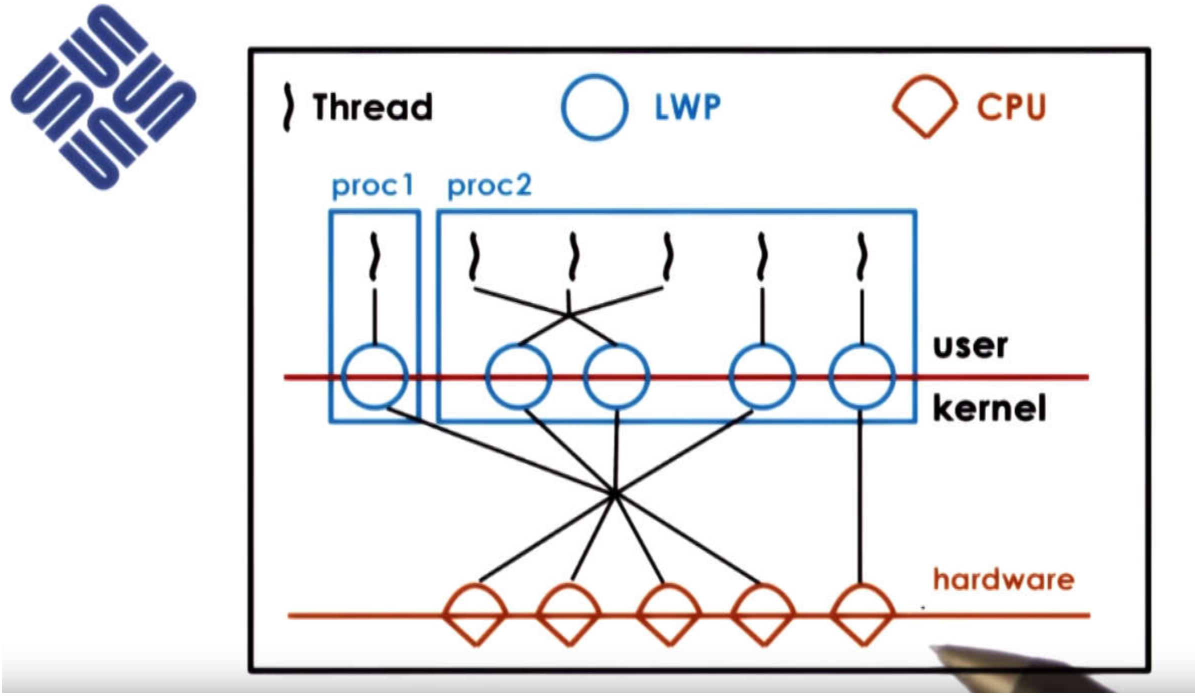
Rationale for splitting single PCB into multiple smaller data structures



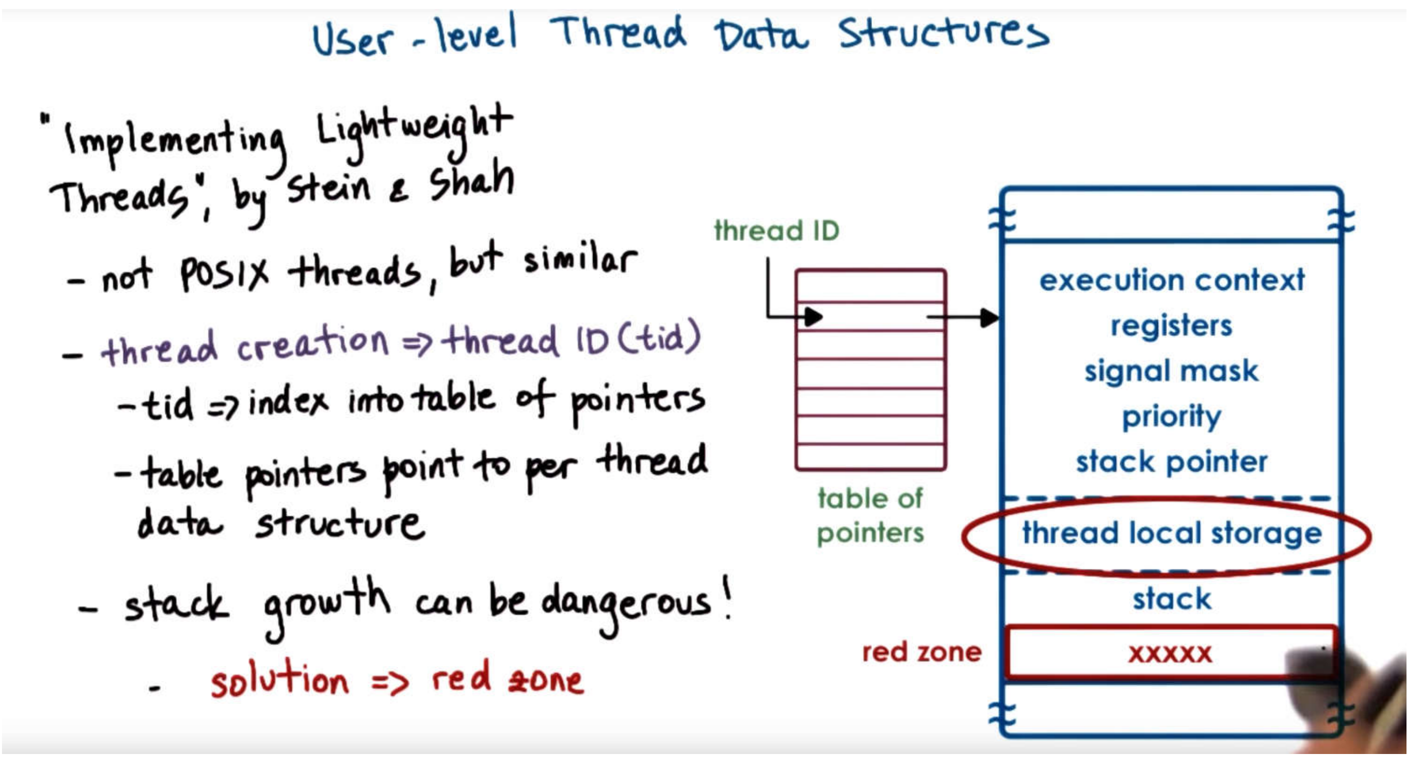


1.3 Sun OS 5.0 Threading Model

- Multi-kernel threads
- Both single and multi user-level threads
- 1 to 1 and many to many user-kernel level thread mapping
- Each kernel-level threads have a light weight process data structure representing the virtual CPUs onto which it's going to be scheduling the user-level threads
- kernel level scheduler managing the kernel level threads



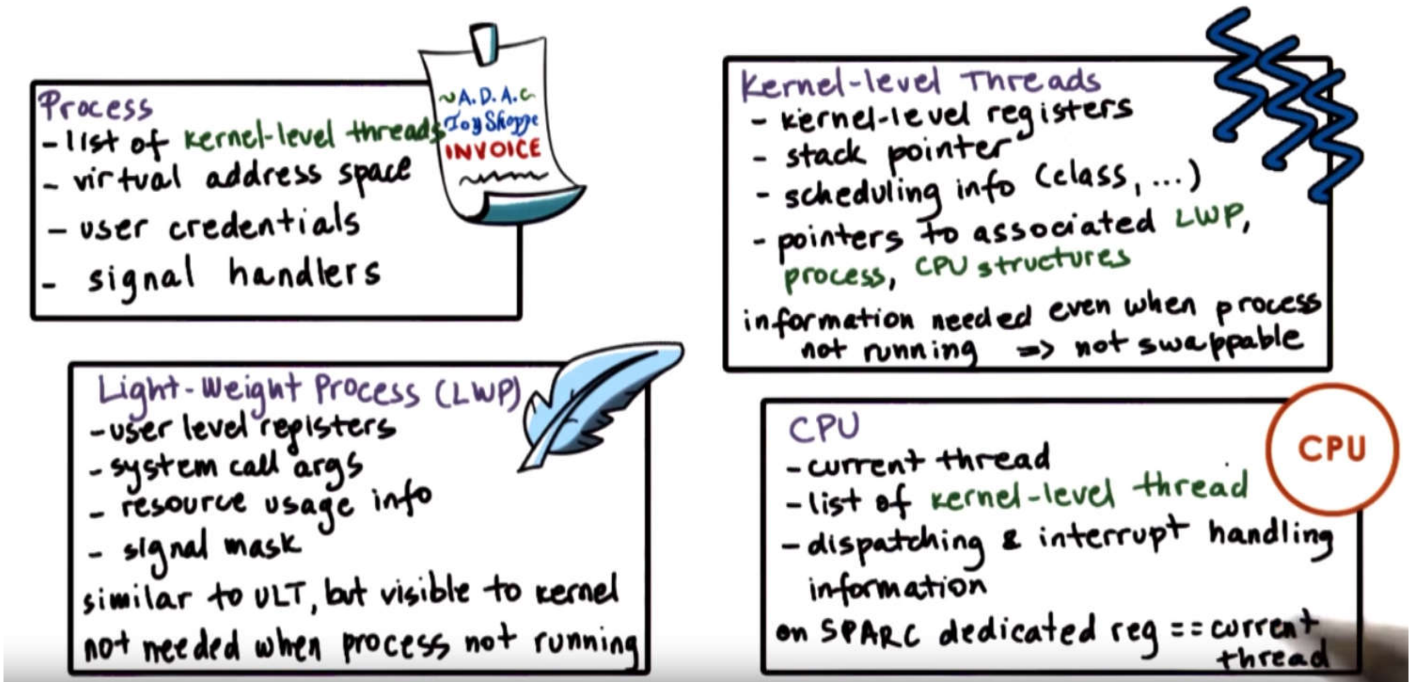
1.4 User-level thread data structures

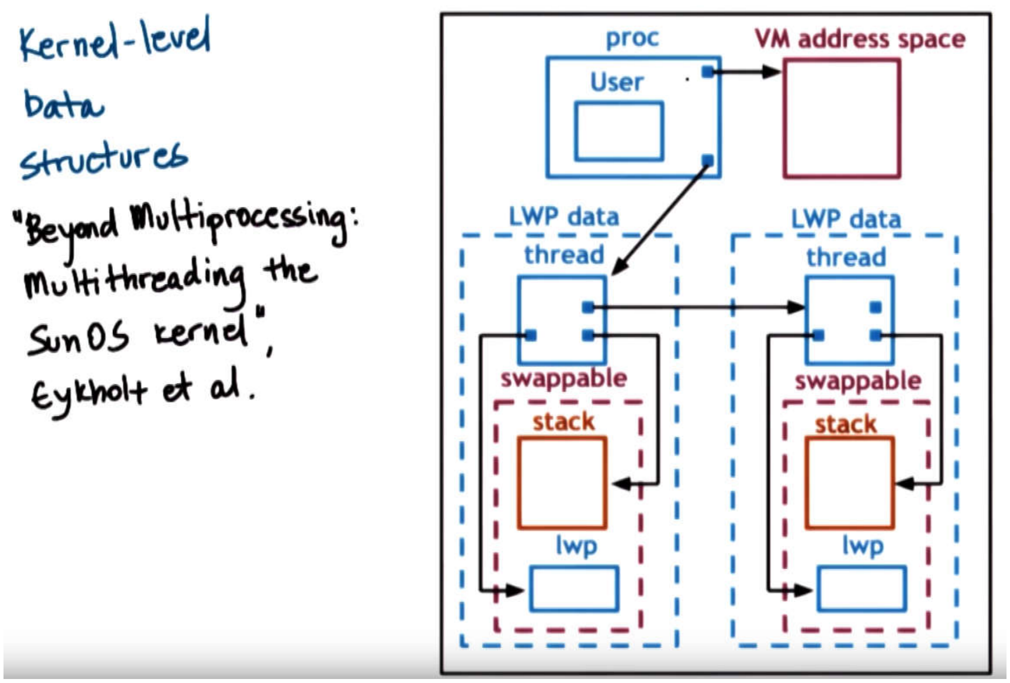


- Having thread ID point to a table entry, we can store some info about the thread in the table entry. This could help us avoid dereferencing a thread id pointer just to find it points to corrupted memeory.
- Thread local storage: include the variables defined in thread functions that are known in the compile time so the compiler can allocate private storage on a per-thread basis for each of them.
- Avoid stack overflow, wracking other data structures by separating them with the non-dereferencable red zone.

1.5 Kernel-level data structures

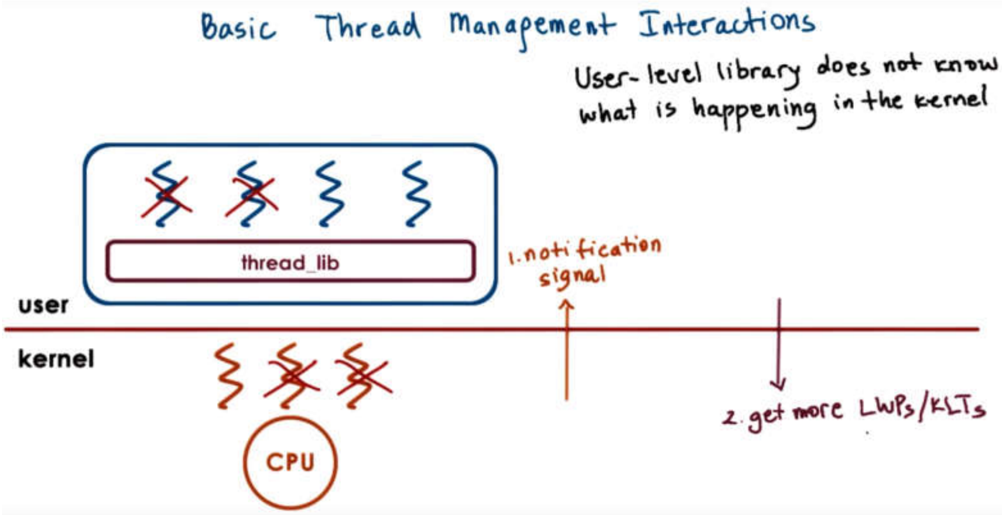
- At OS level, the kernel tracks resource uses on a per kernel thread basis, maintained in the lightweight process that corresponds to taht kernel level thread.



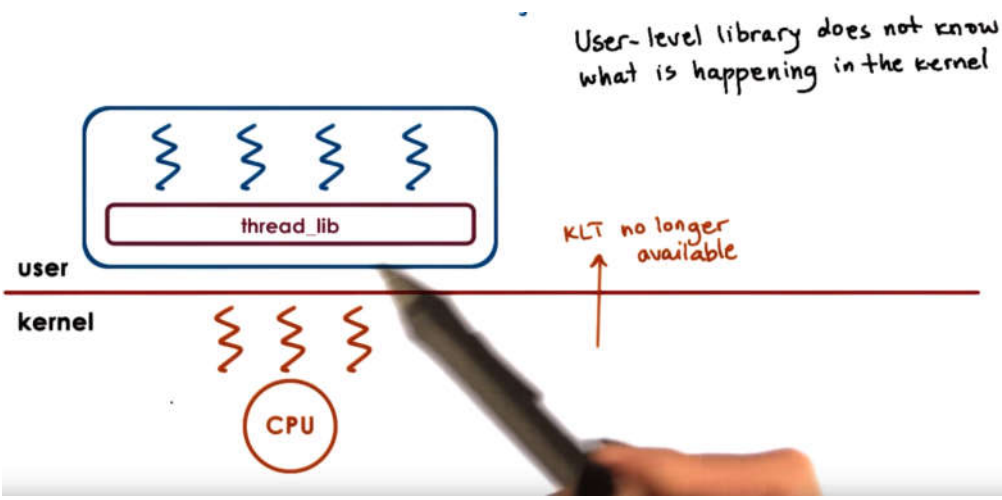


2. Basic Thread Management Interactions

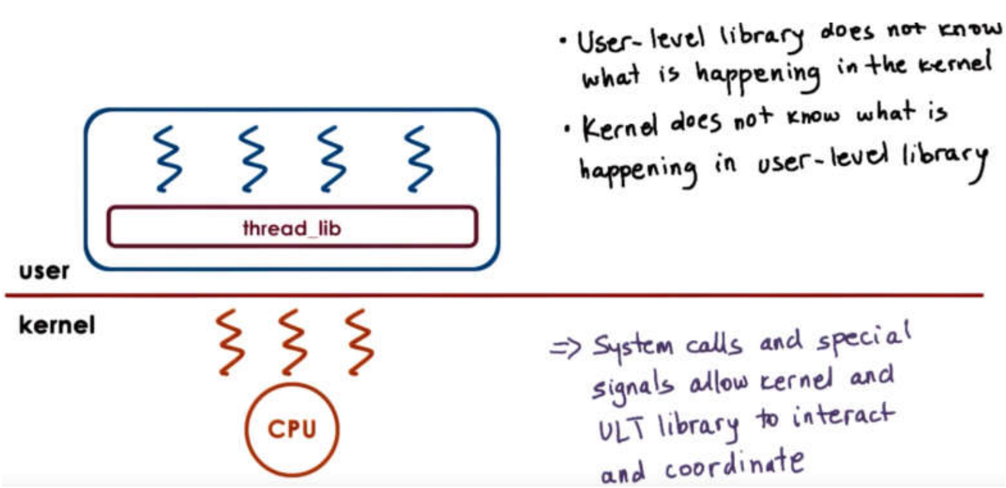
- Example Case: The process originally has two KLTs, but both of them are blocked due to an I/O operation. The two other ULTs are unnable, so the kernel can send a signal to ULT and then give it an extra KLT to run the runnable threads.



- Then when the blocking is done, the kernel will tell the ULTs that the extra KLT is no longer available.

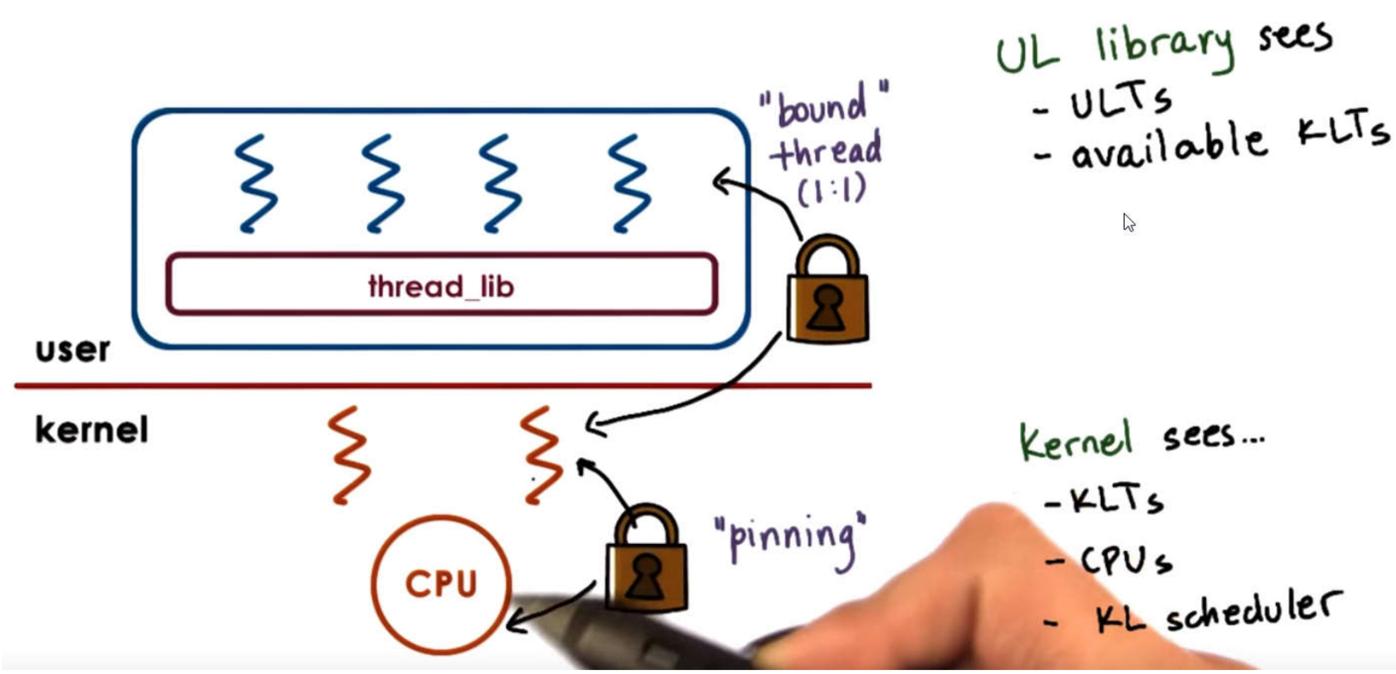


-Summary:

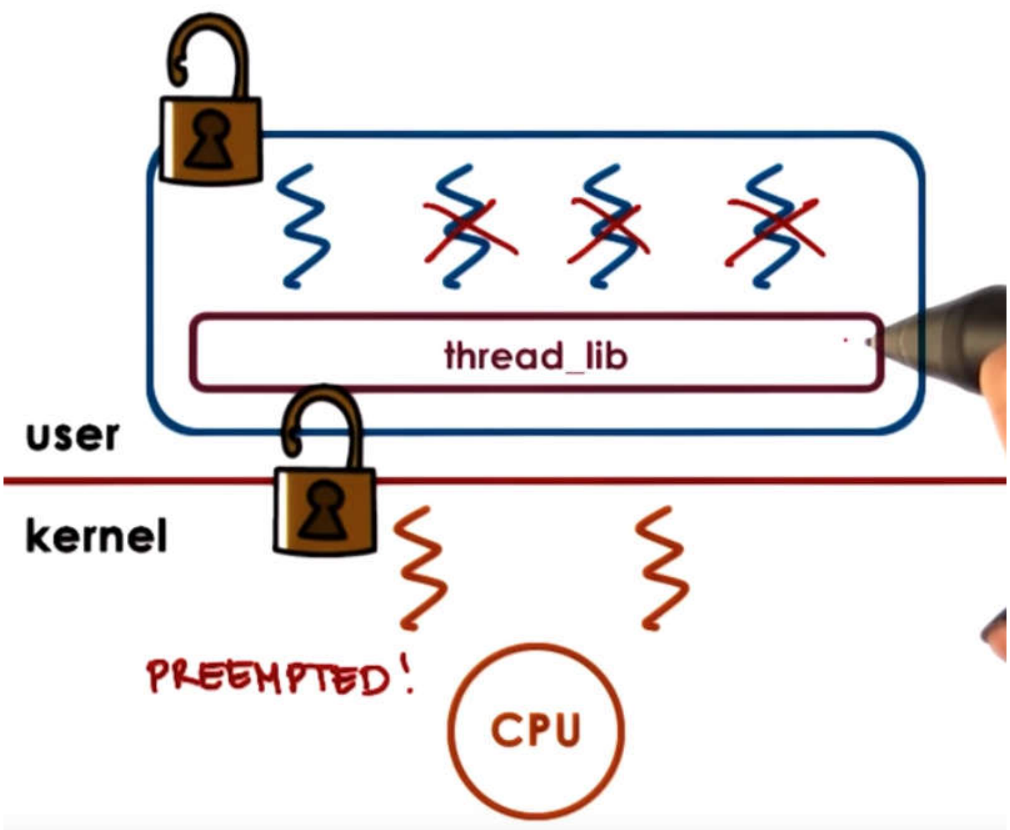


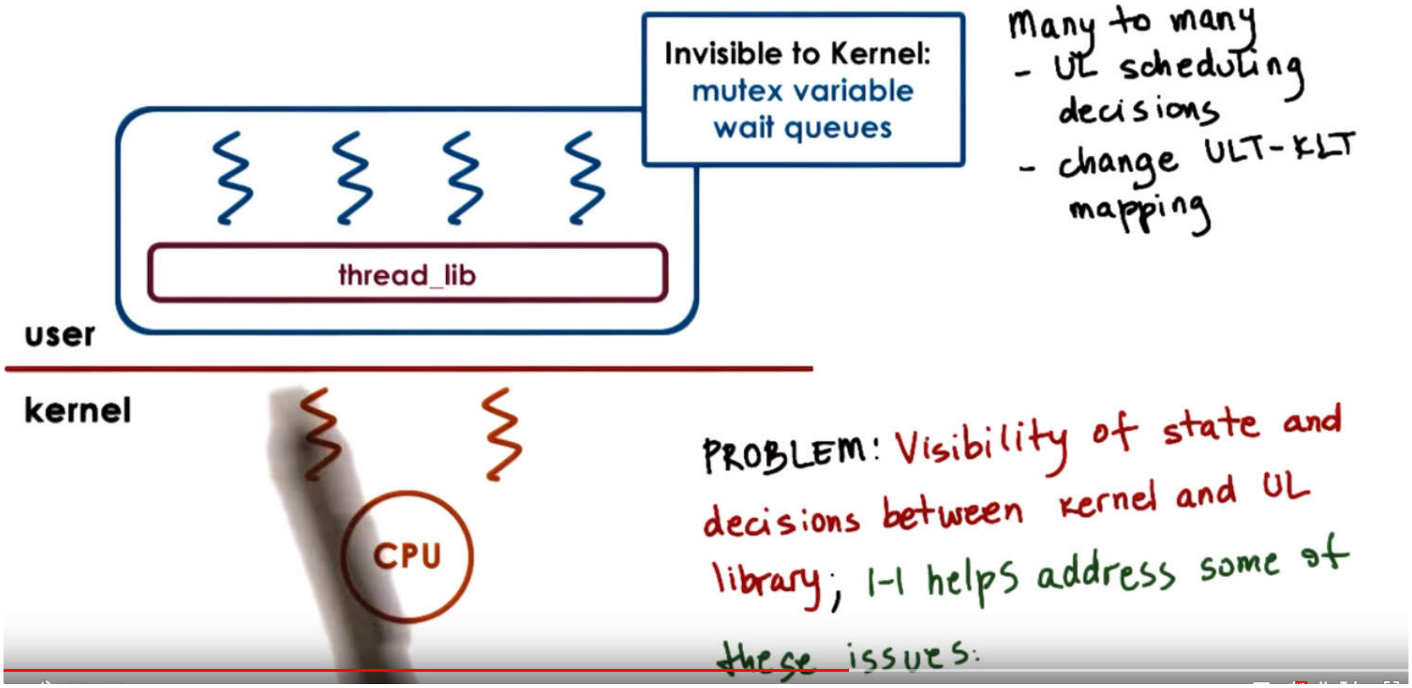
2.1 Thread Managment Visibility

- Bound ULT to KLT

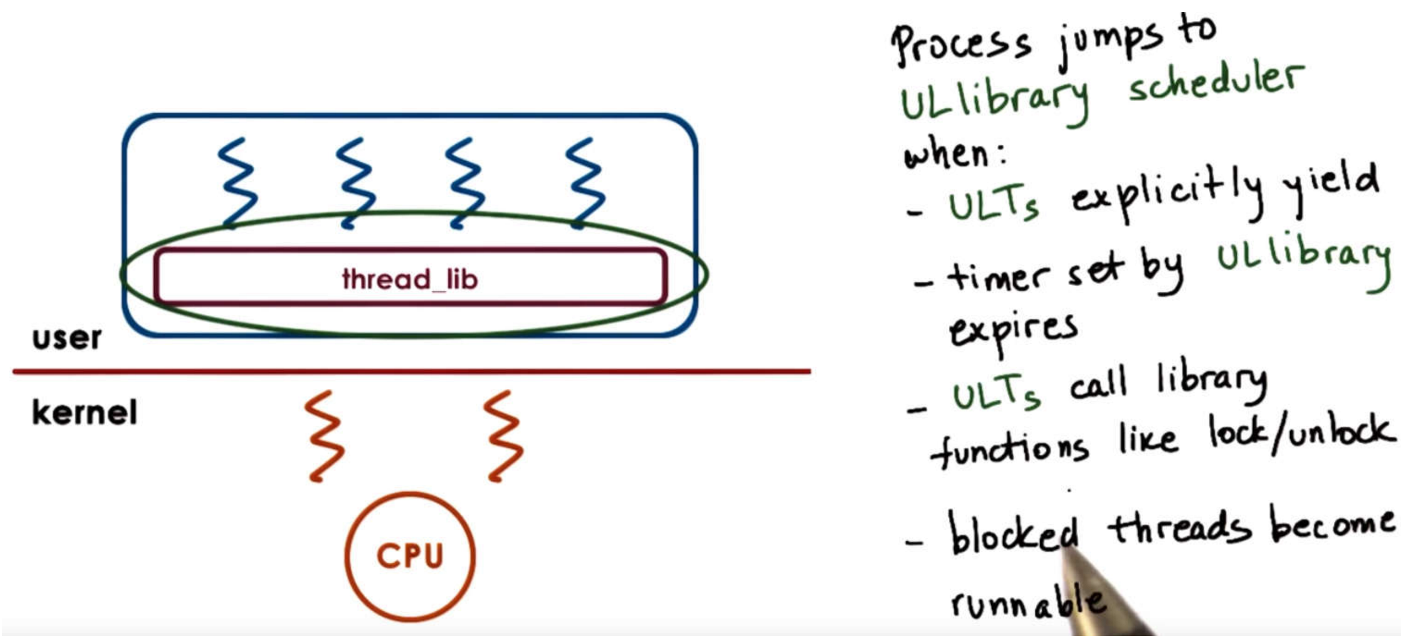


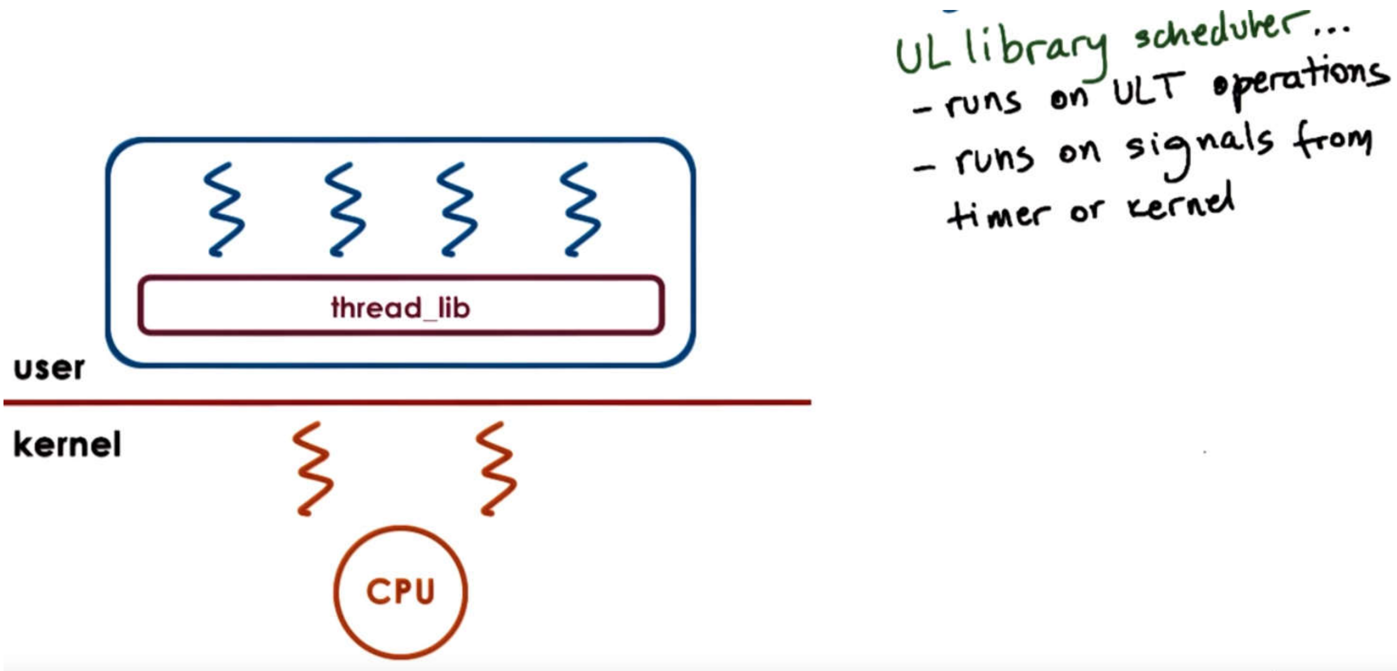
- Problem Case:
  When the kernel cannot see ULT scheduling decisions, it might switch a thread that has the mutex lock off the CPU and then iterate throught other threads just to find that they are blocked at the mutex lock acquisition.





How does the UL Lib Run?



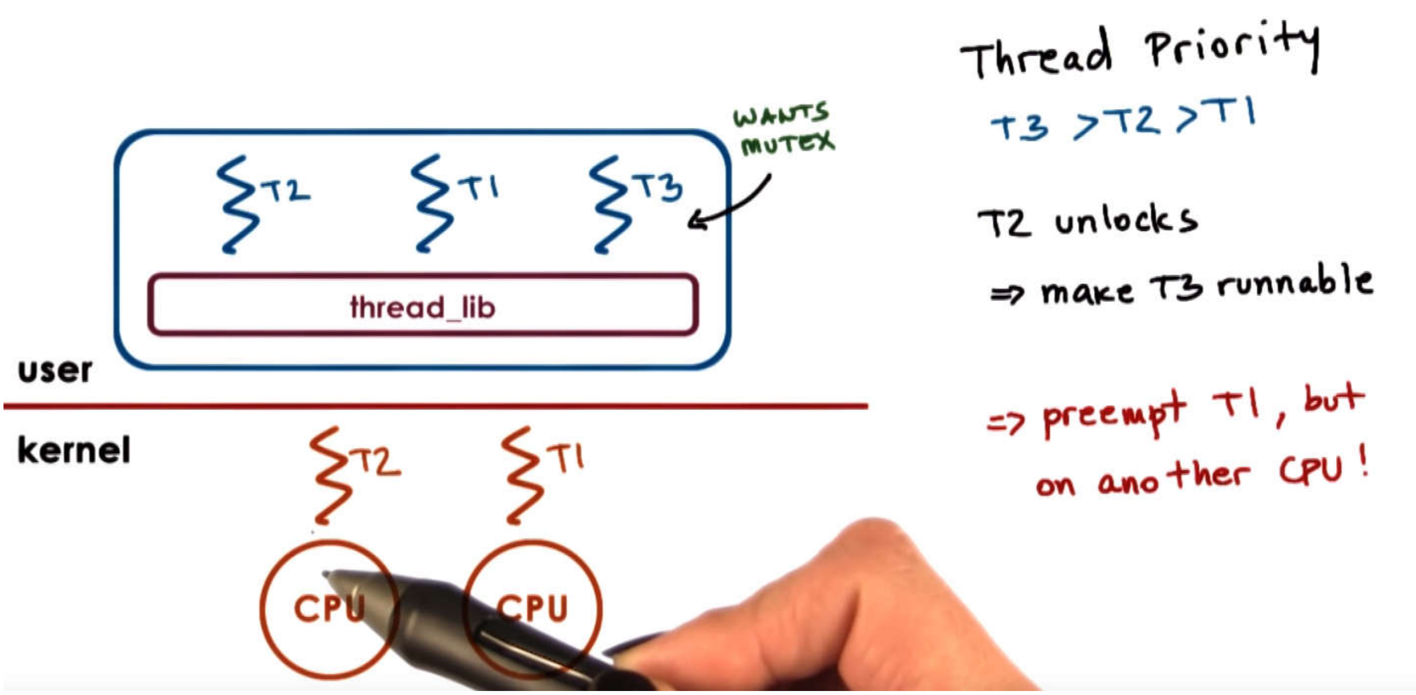


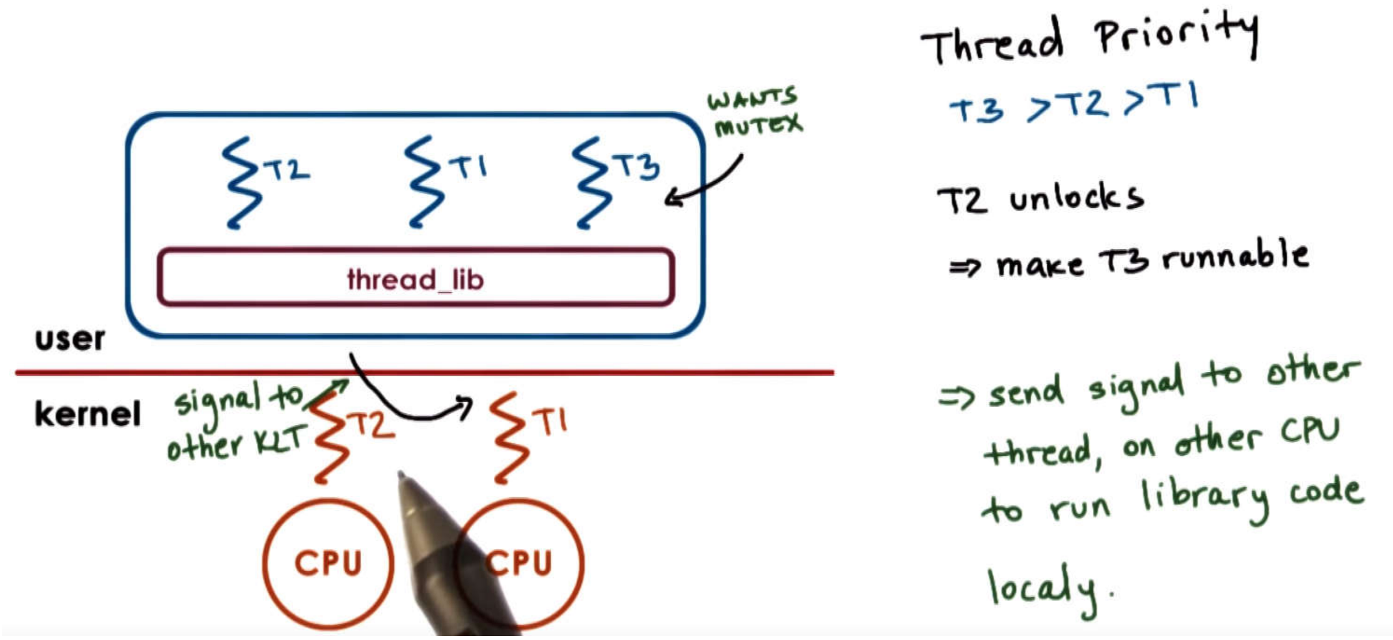
2.2 Issues on Multiple CPUs

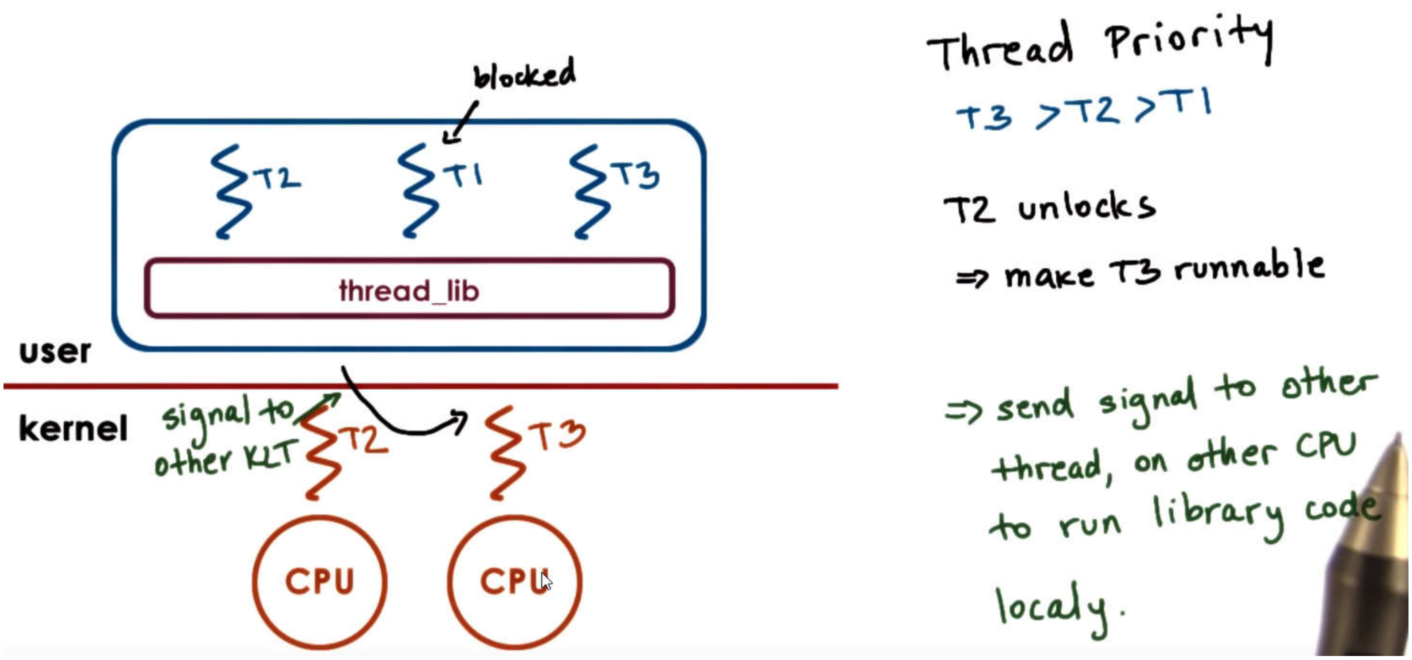
Question: Why we cannot directly modify register of one CPU when executing on another CPU? --P2L4 Lesson 13

Question: What does preempt mean?

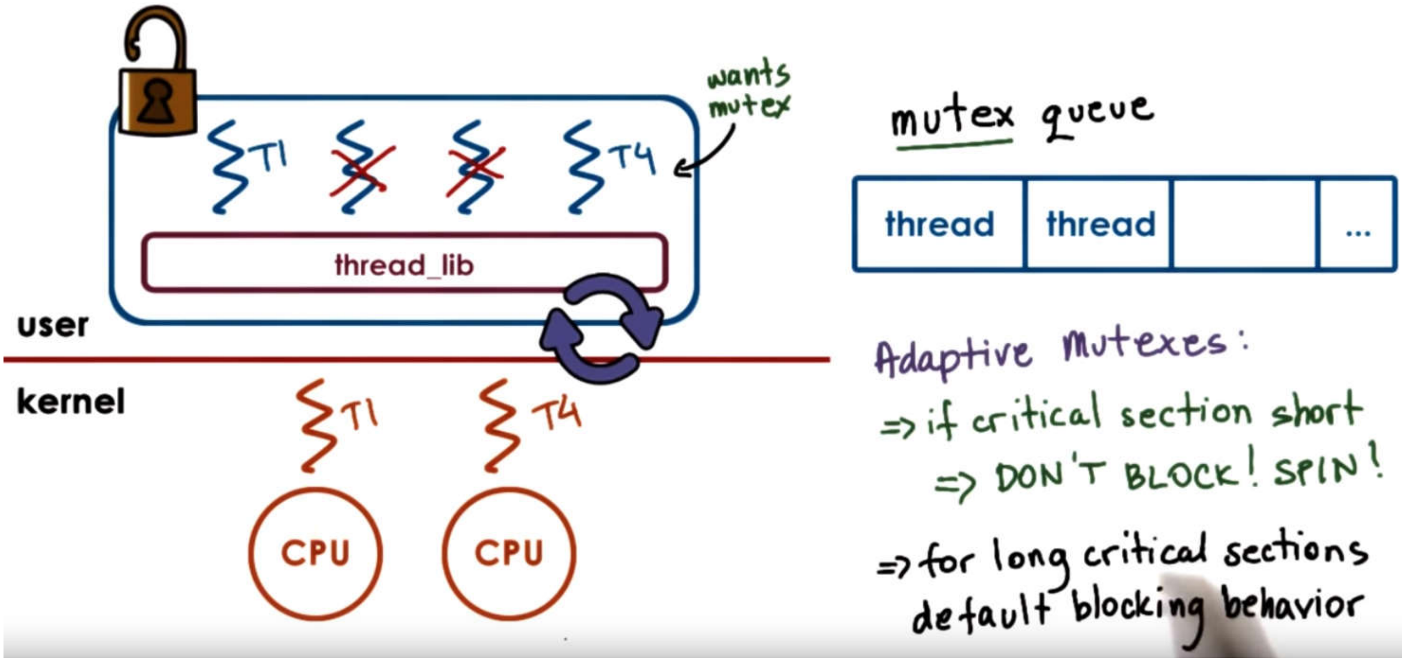
When there are multiple CPUs, we need interaction between KLTs running on different CPUs.







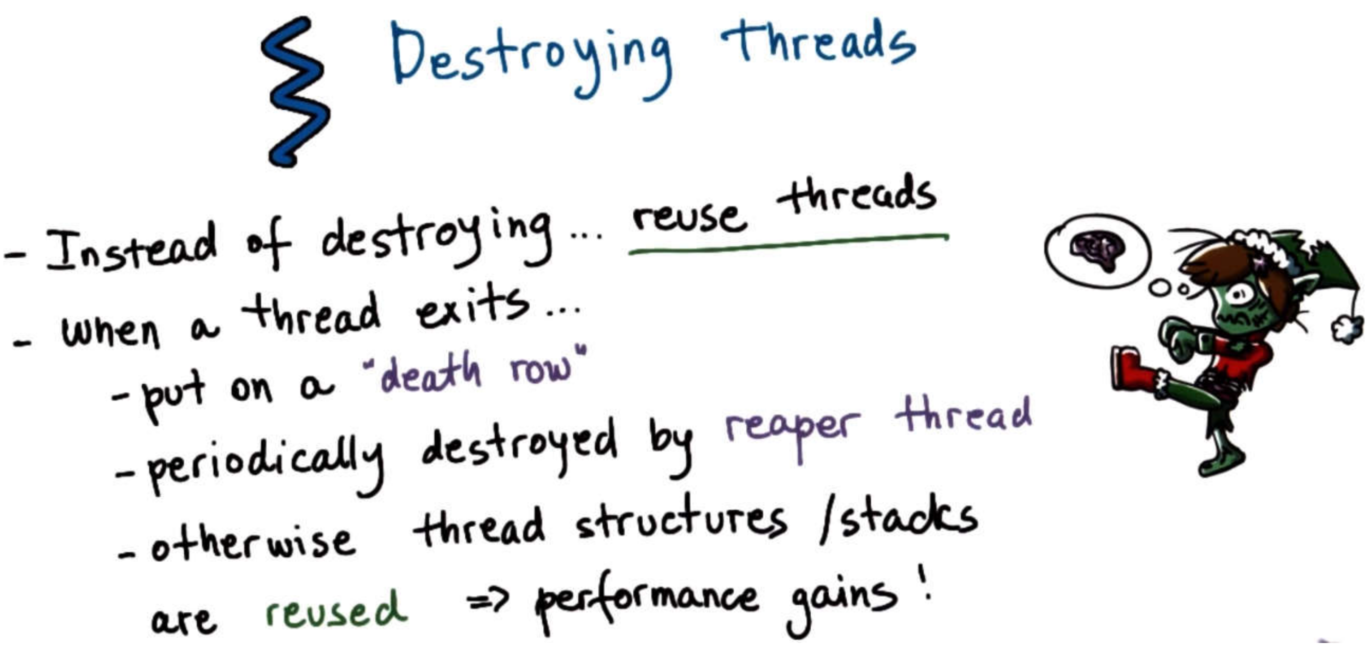
2.3 Synchronization Related Issues.



When critical section is short, T1 might release the mutex lock before the context switching is completed on T4. So it's better off leave T4 spinning on CPU 2 and burn a few cycles.

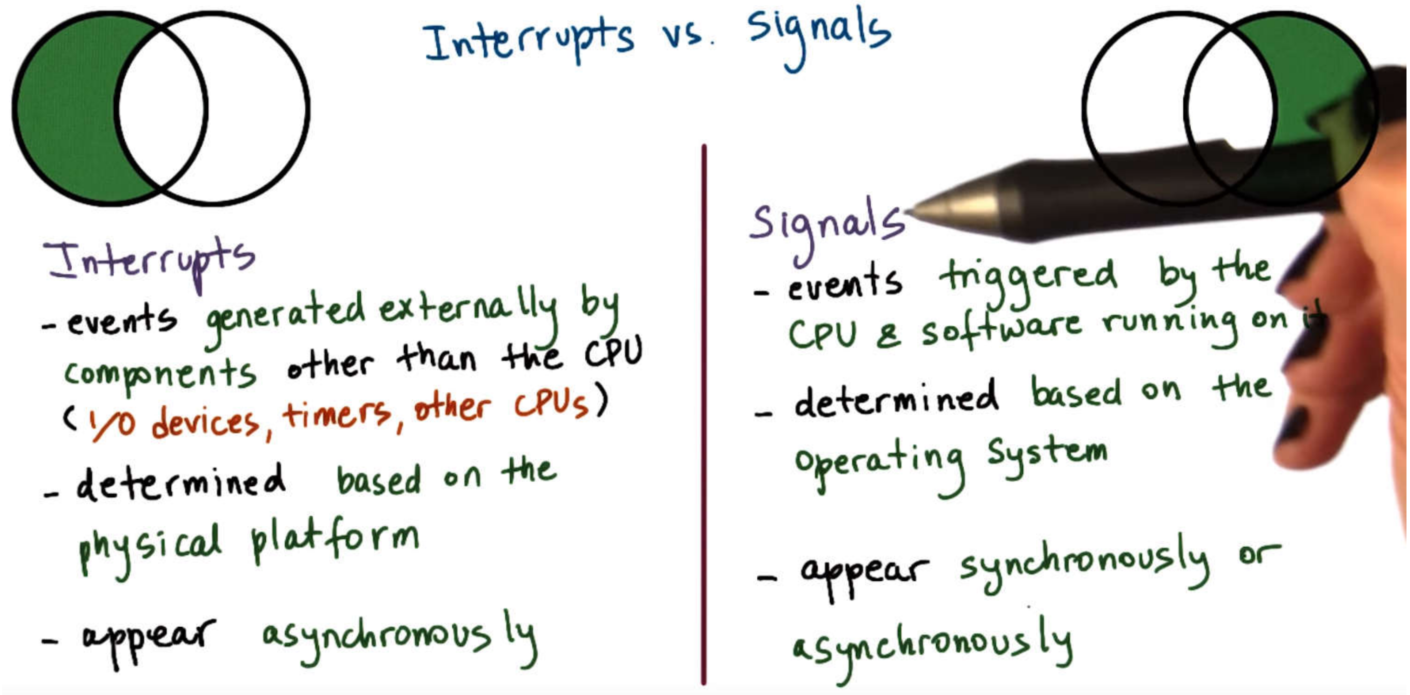
This is only possible on multi-CPU context.

2.4 Destroying Threads

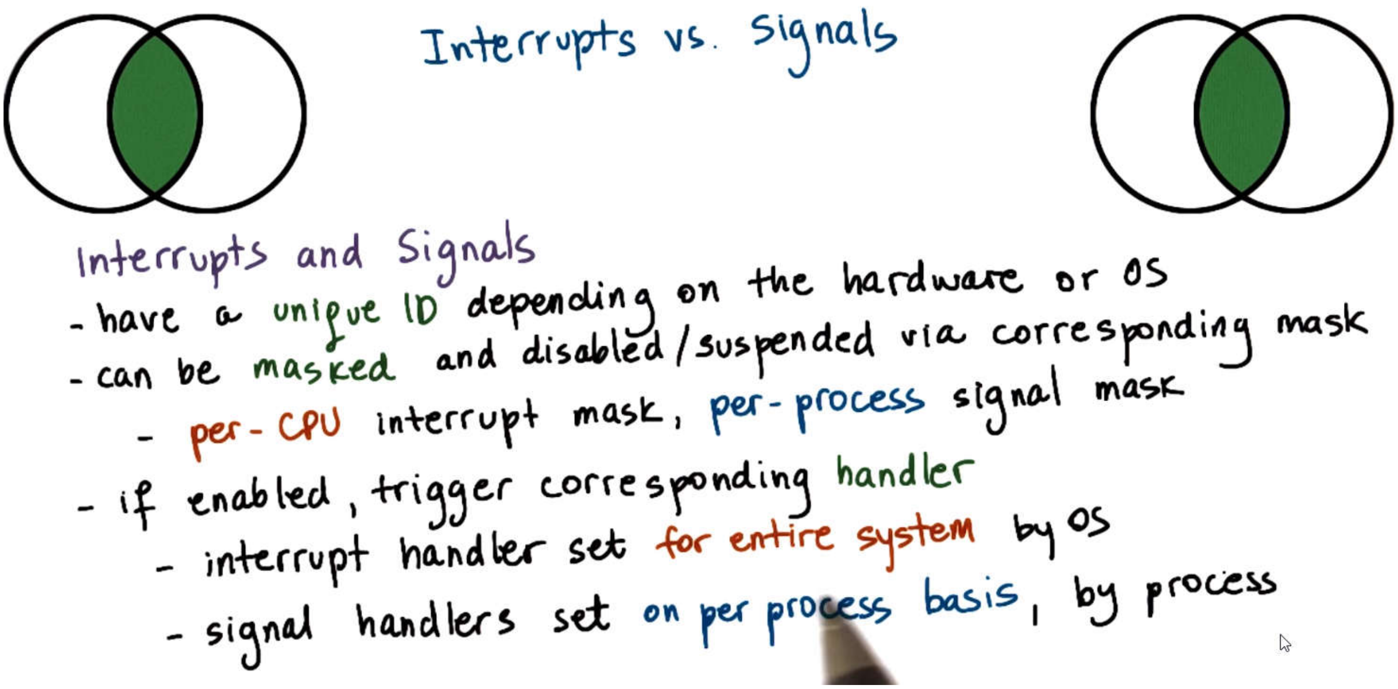


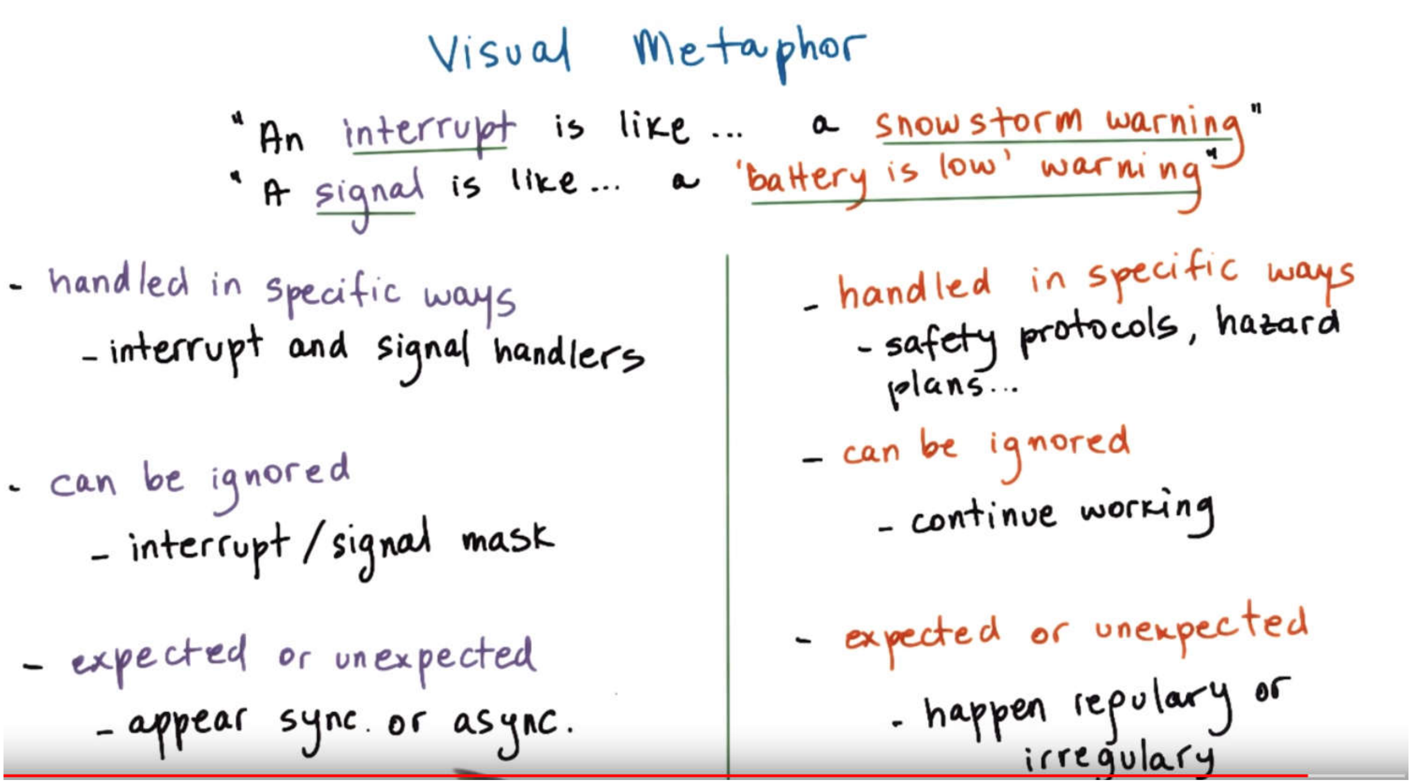
3. Communication Mechanisms

3.1 Interrupts vs. Signals

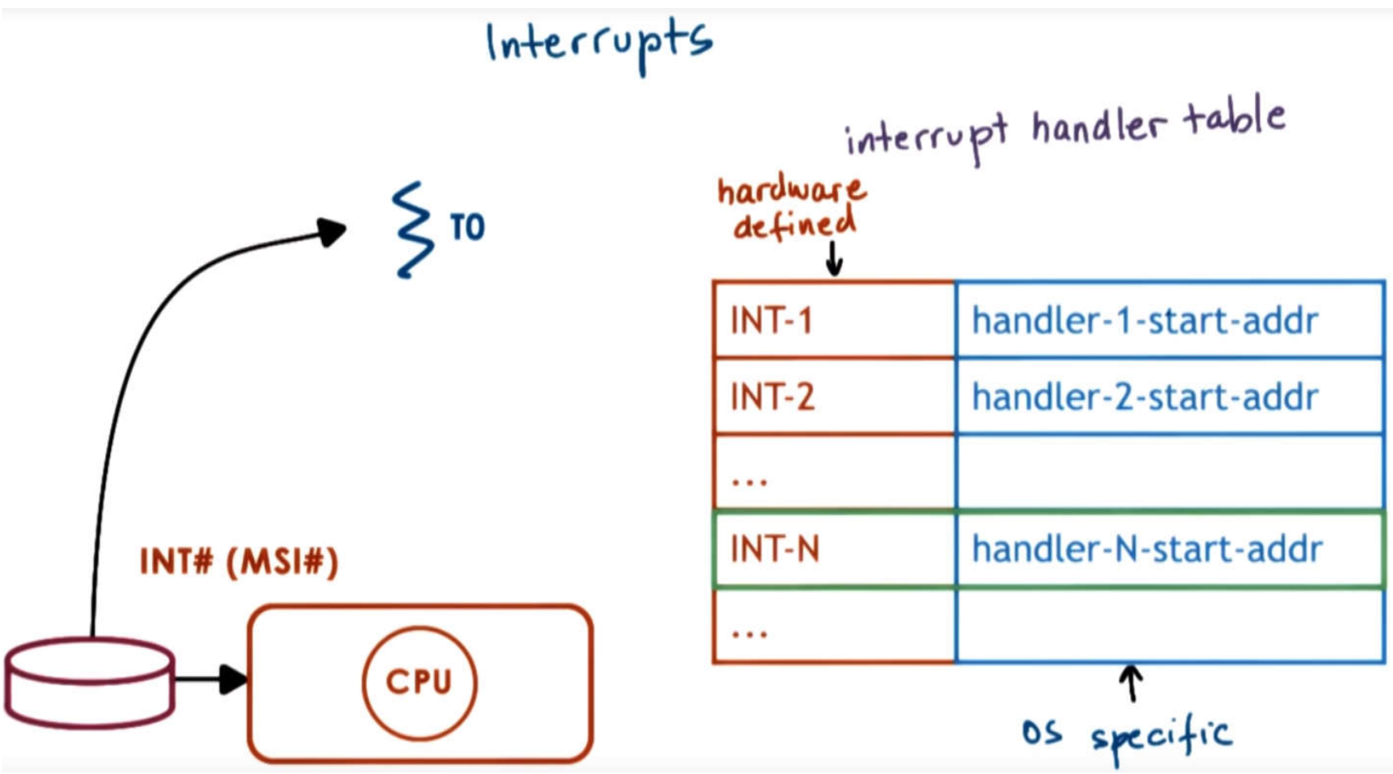


- **synchronously:** means it's triggered by some specific actions that took place on the CPU, a synchronous signal is generated in response to that action. (such as an attempt to access unallocated memory.)

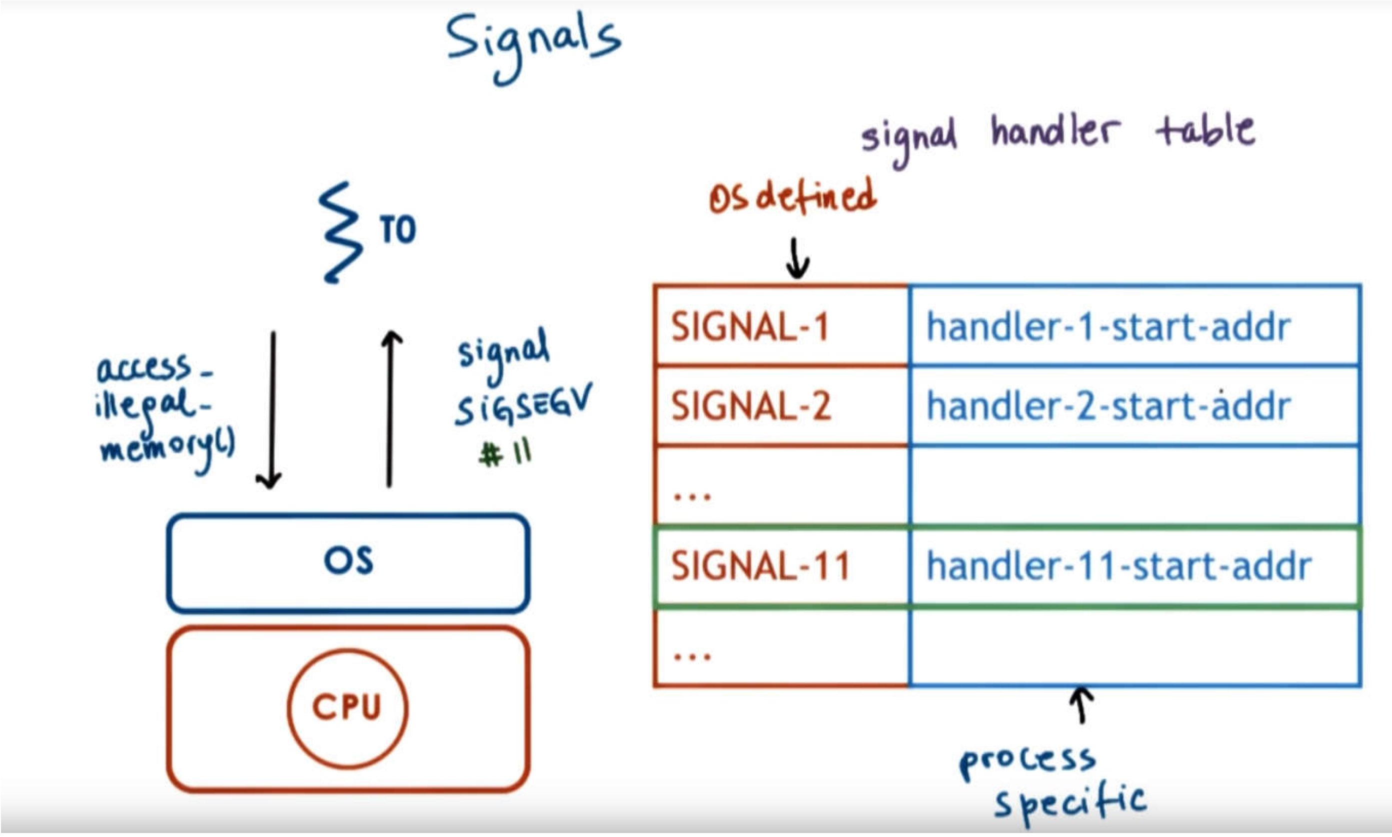


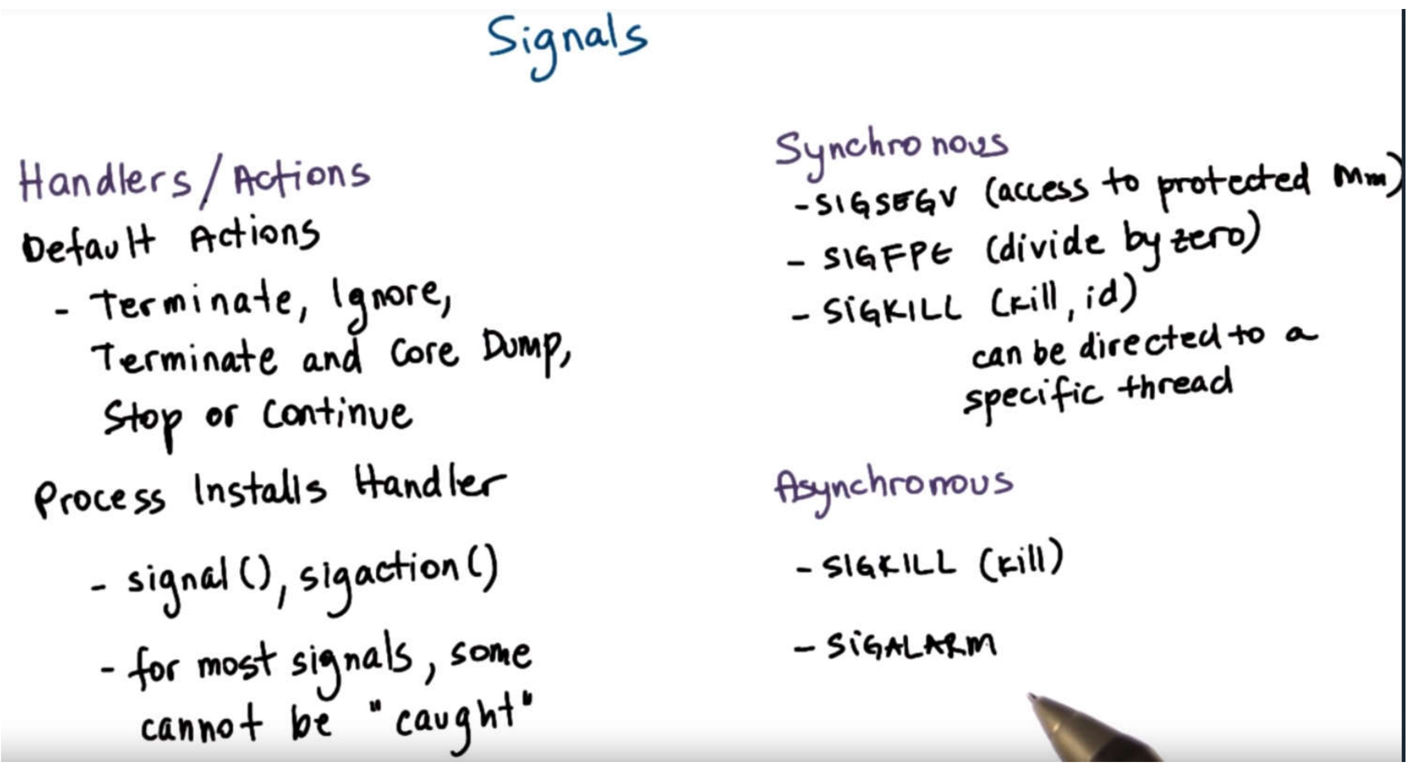


Interrupt Handling



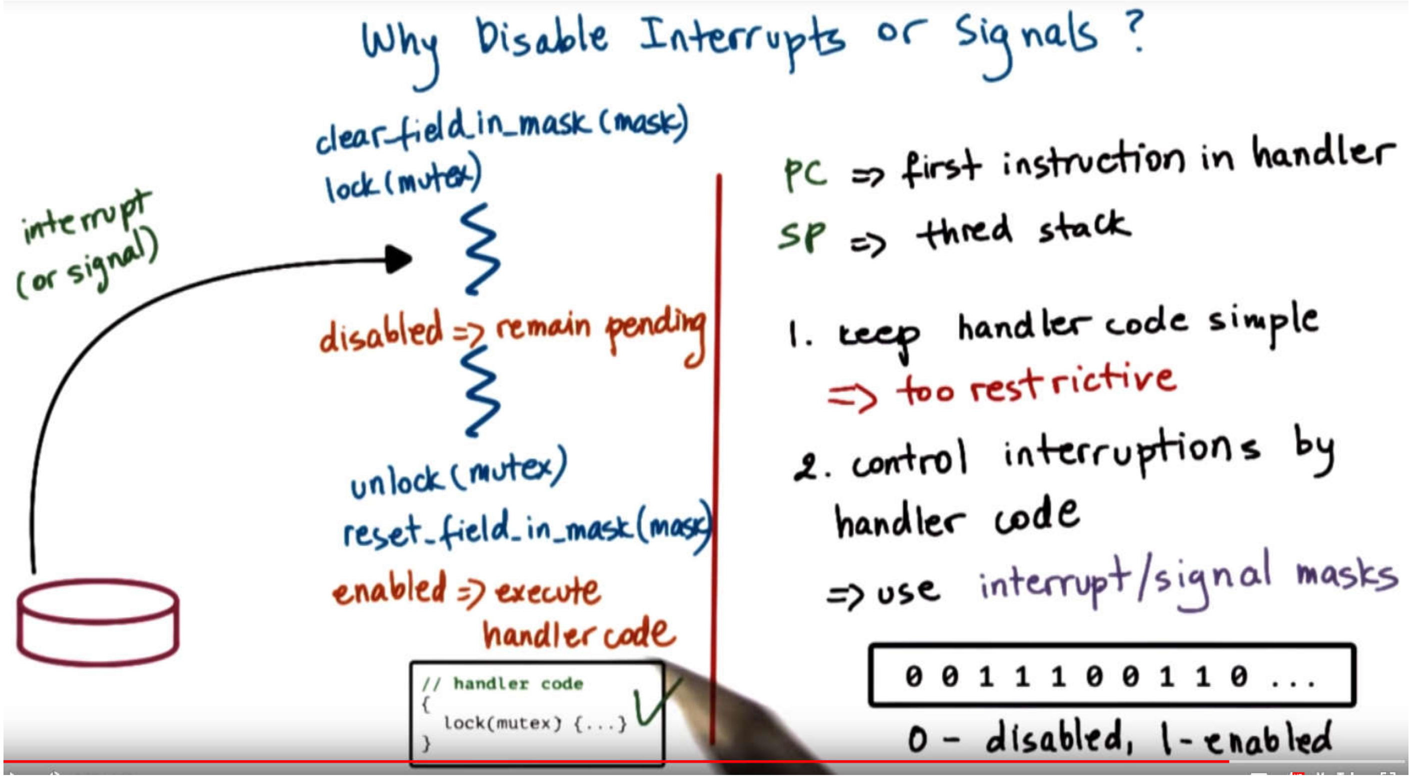
Signal Handling



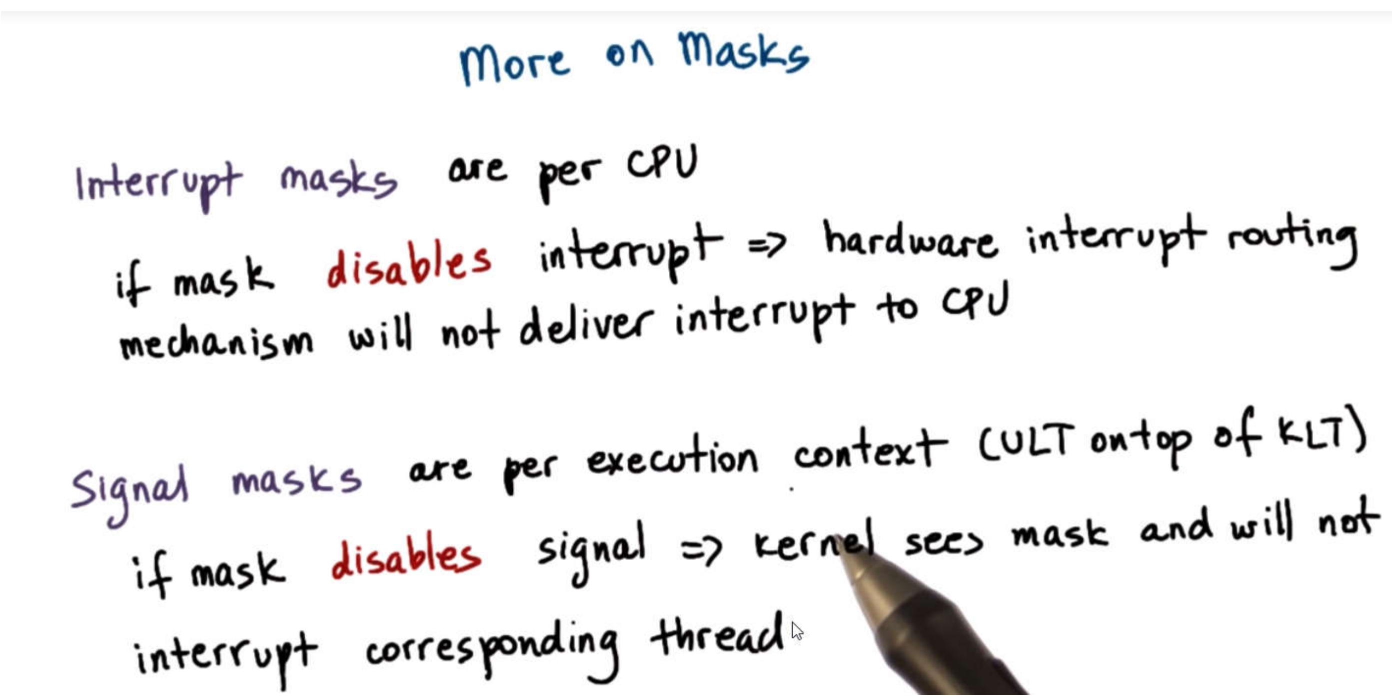


3.2 Disabling Interrupts or Signals

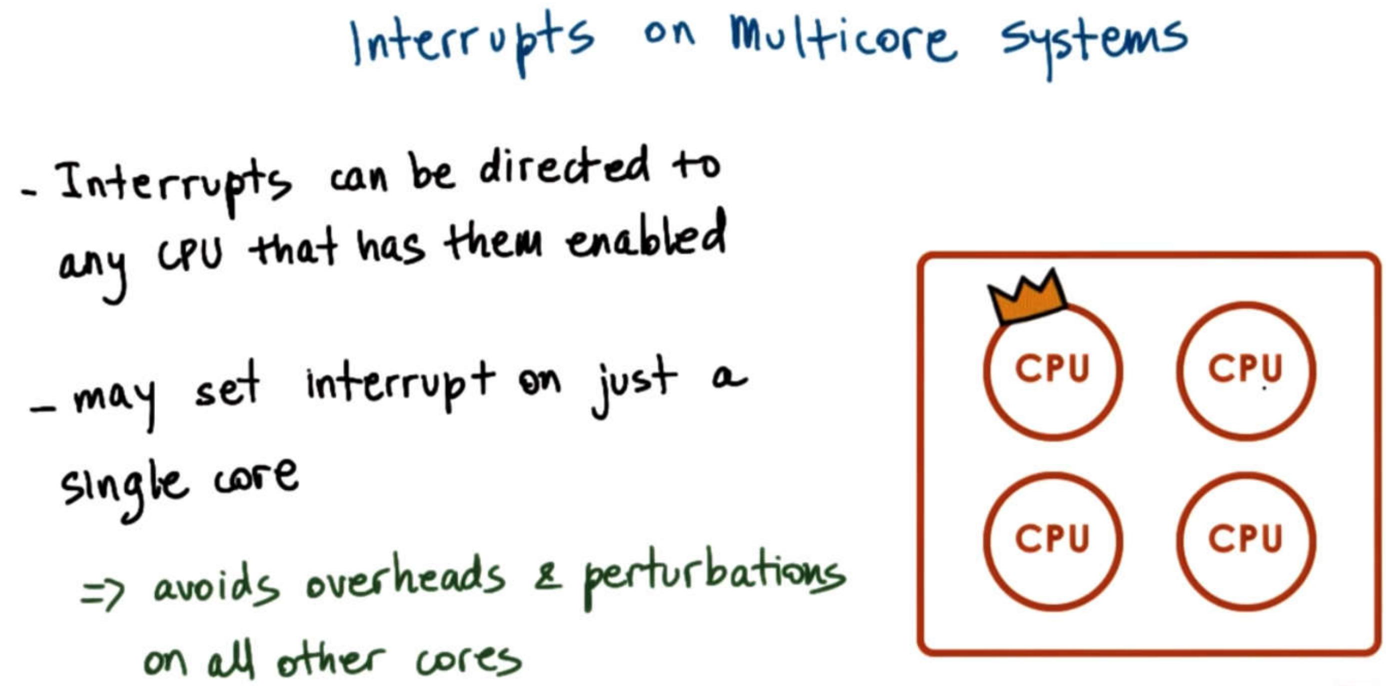
What happens when an interrupt/signal appears:



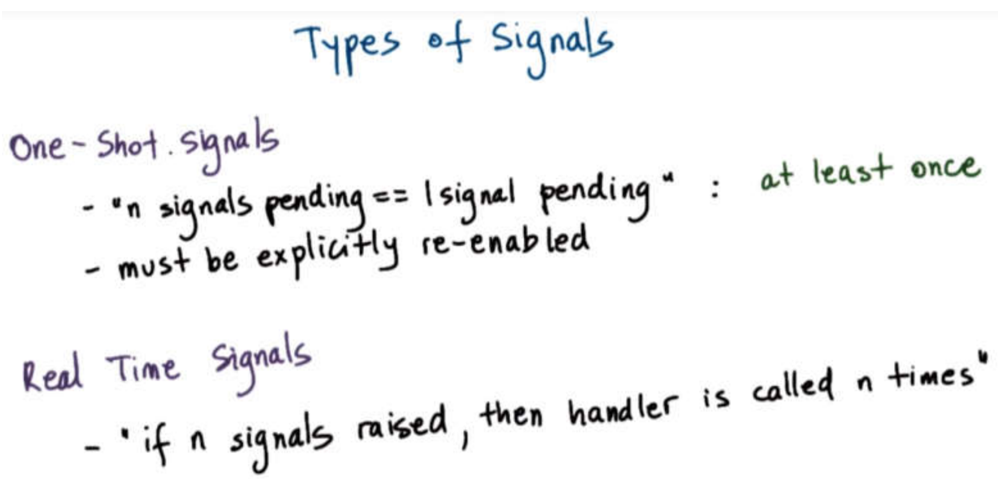
- When the signal/interrupt handler code needs to acquire a mutex to proceed, but the mutex is currently owned by the thread's code, there is a dead lock situation.
- Solution to the above problem: signal/interrupt masks, enabling us to enable/disable signal/interrupts.
- The mask is a sequence of bits that corresponds to specific signal or interrupts, 0 – disabled, 1-- enabled.
- In the thread code, when the thread needs to acquire the mutex, disable the possible signal/interrupts, after it's done with the critical secion, unmask the signals.
- While a signal/interrupt is pending, other signals/interrupts that occurred will be pending too, once the mask is reset, the handler routine will typically be executed only once.



3.3 Interrupts on Multicore Systems

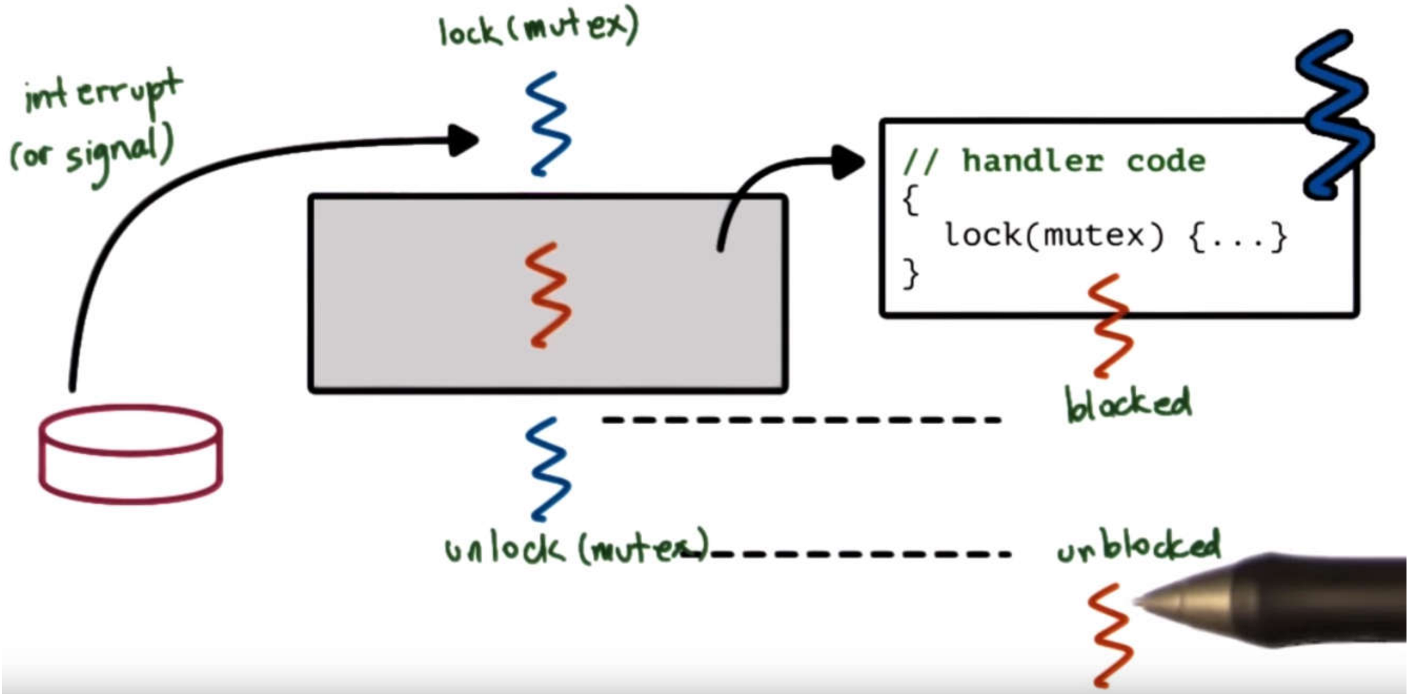


3.4 Types of Signals

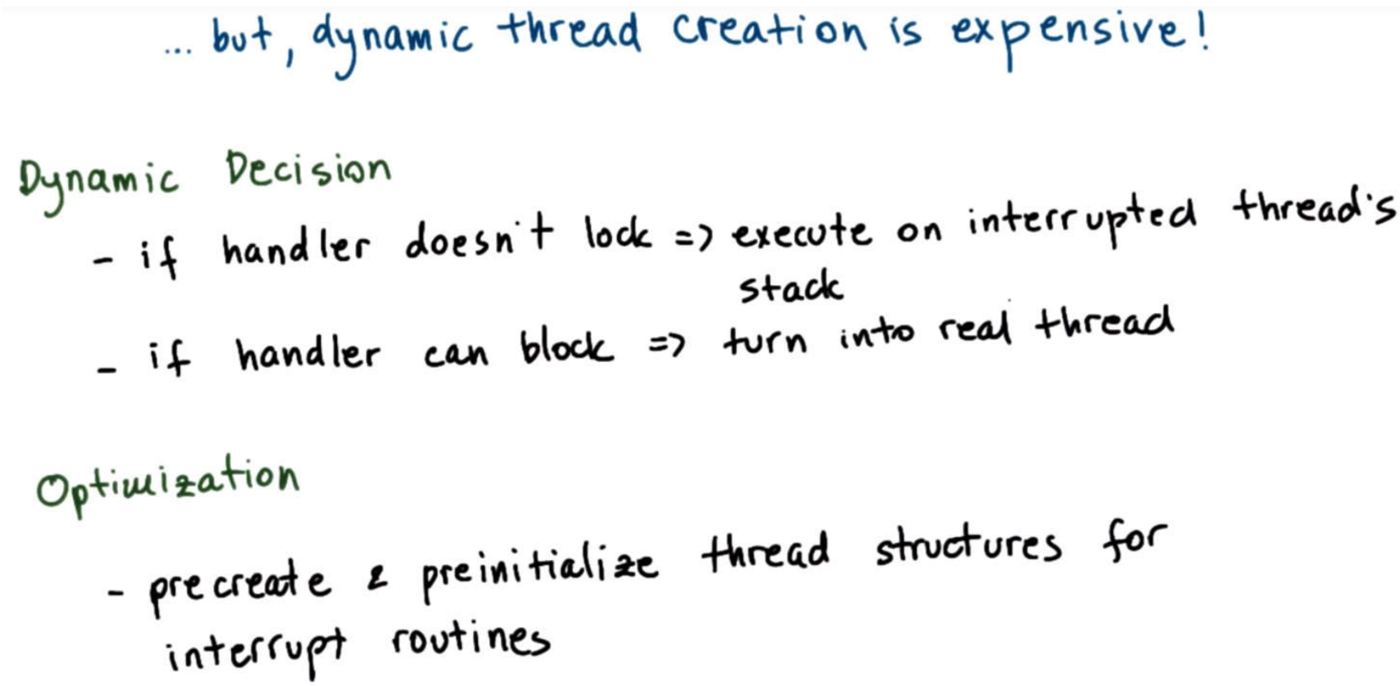


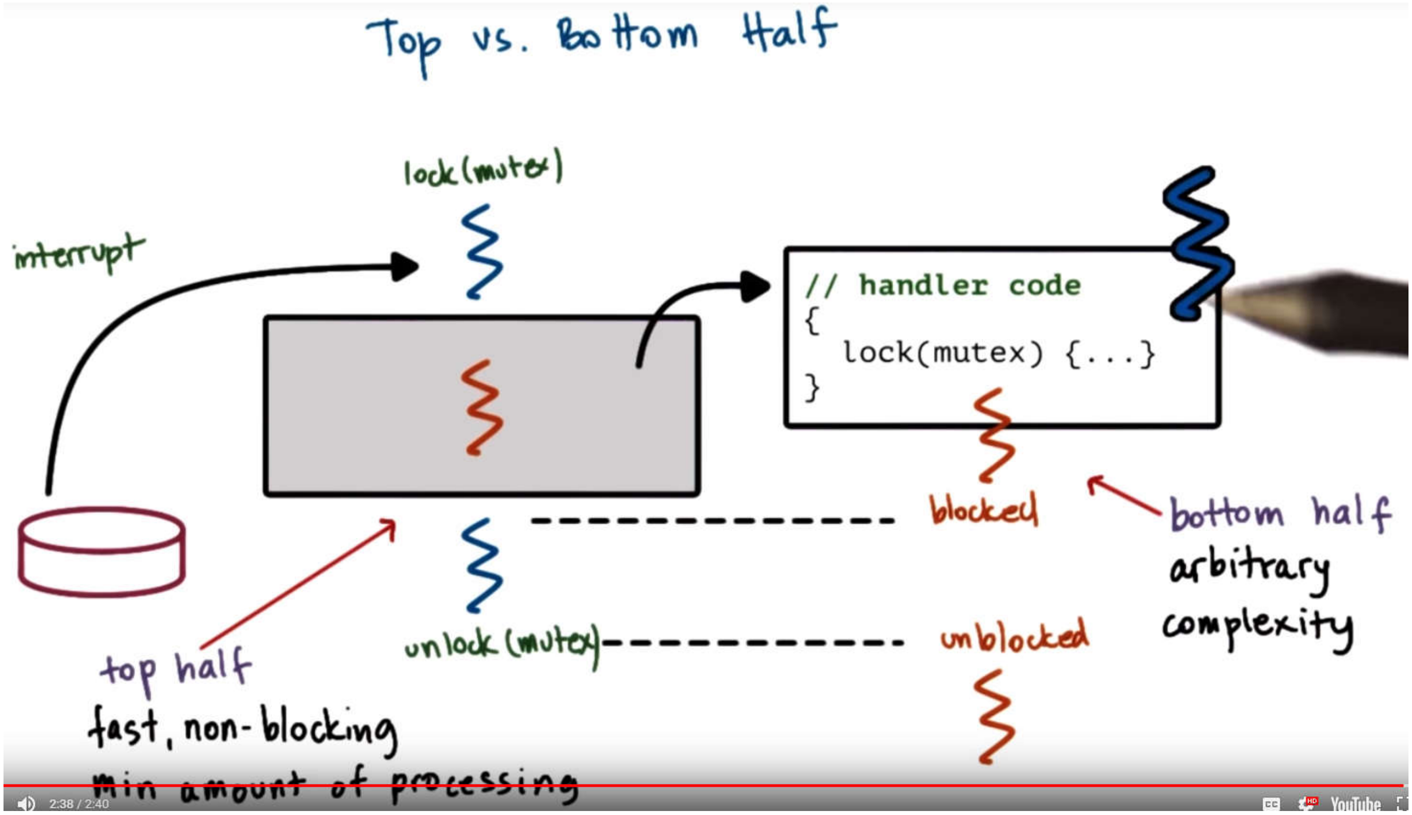
3.5 Handling Interrupts as Threads

- Problem: When an interrupt/signal is sent to a thread that has a mutex lock, also the interrupt handler needs to acquire the lock, there is a dead lock situation.
- One possible solution: Handle the interrupt as a thread, switch the original thread back till the mutex is released then unblock the interrupt handler “thread”.

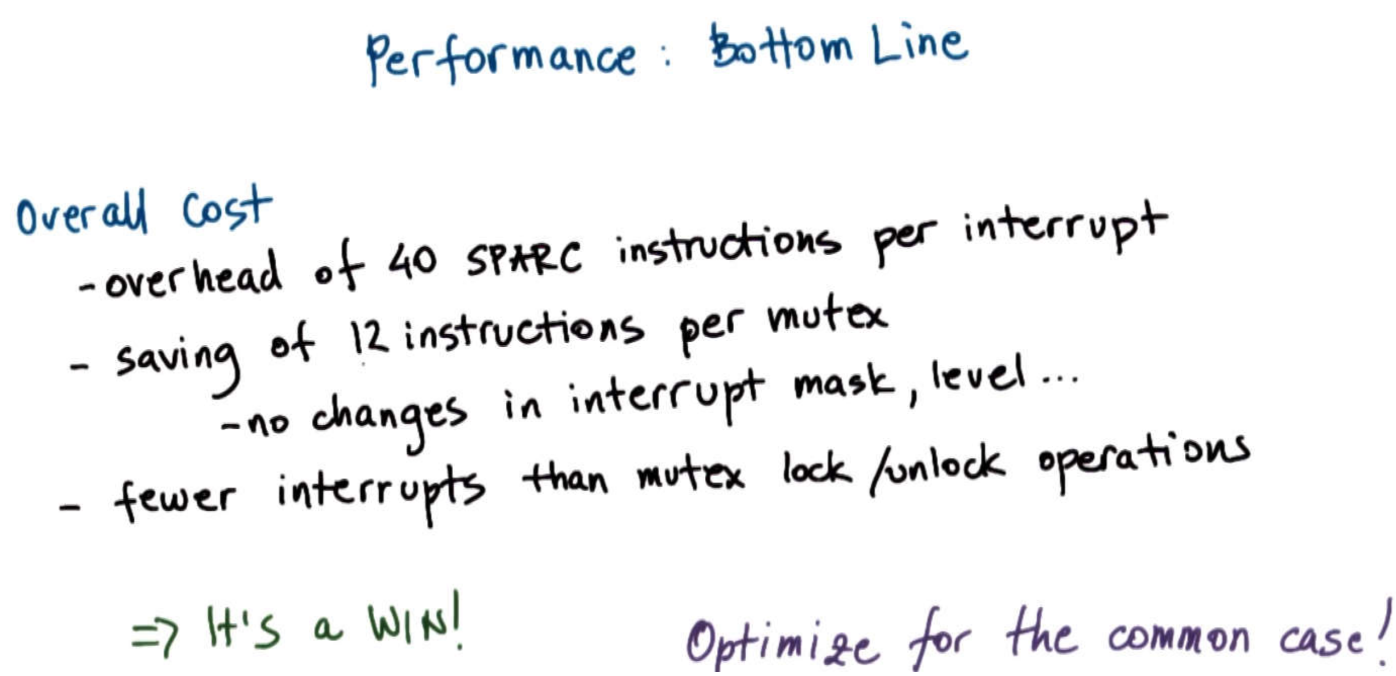


- However, dynamic thread creation is expensive:

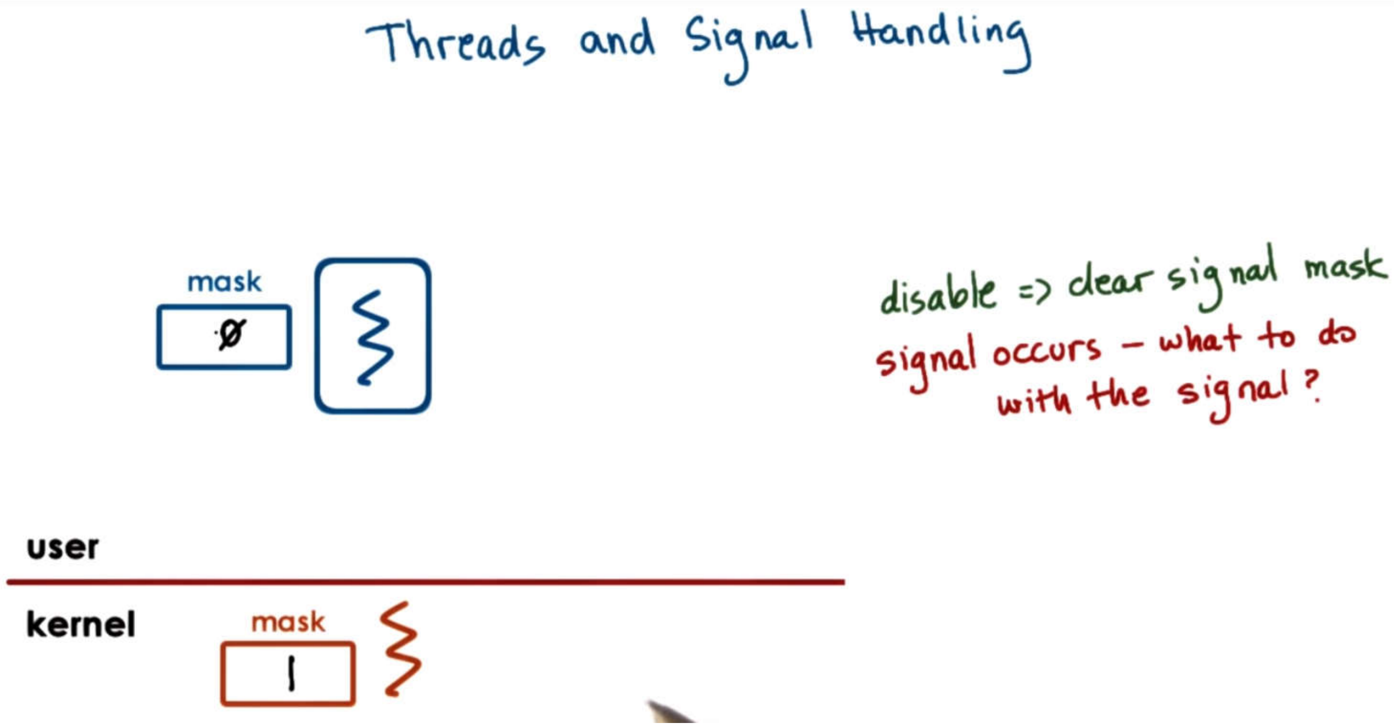




Rationale behind treating interrupt as threads:

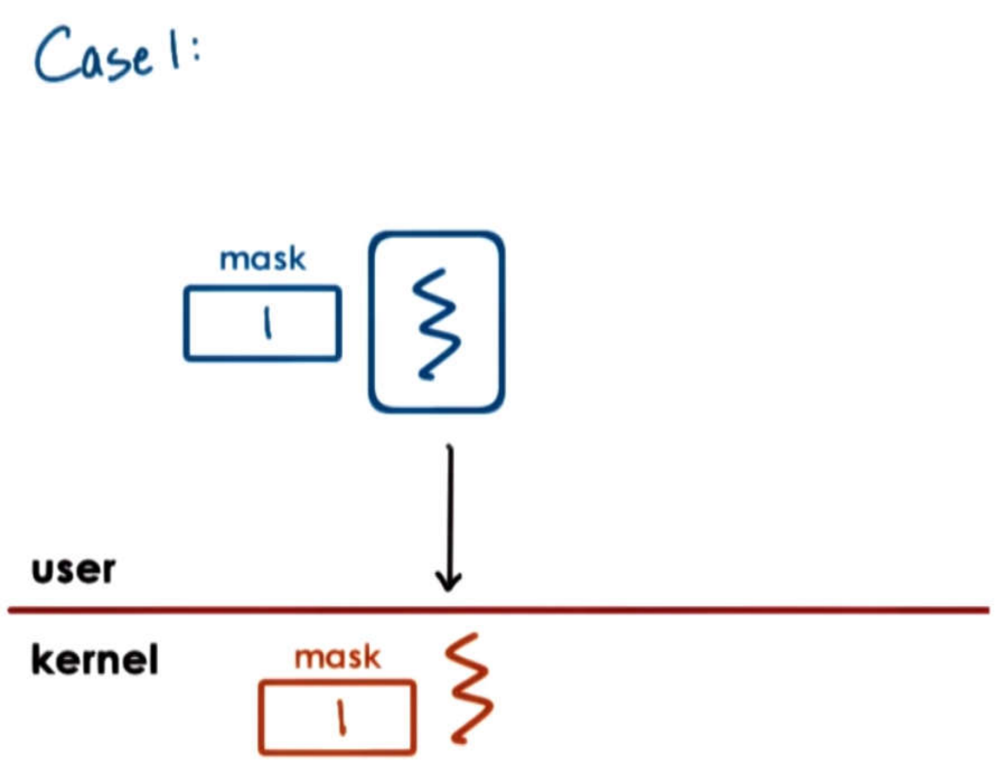


3.6 Threads and Signal Handling



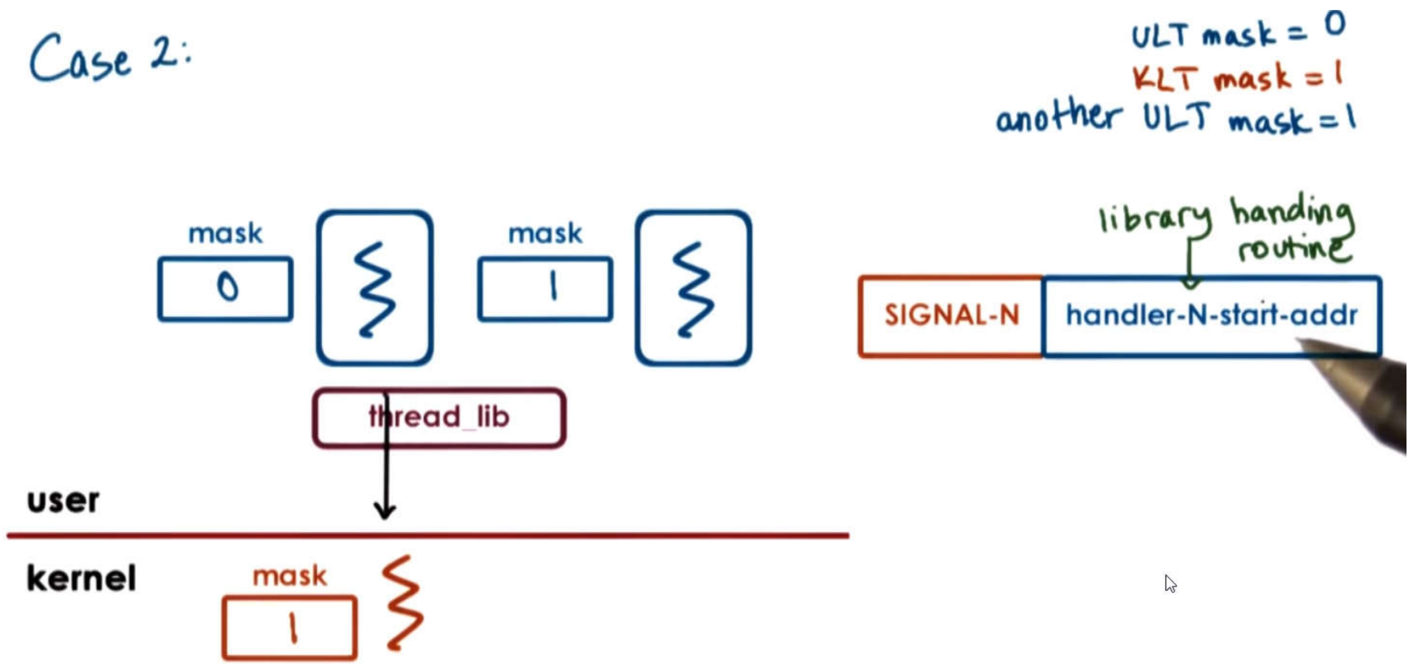
Problem: ULT and KLT has separate masks, when ULT disables a signal how to let the KLT that it mapps to know how to deal with the signal.

Case 1: ULT mask = KLT mask = 1



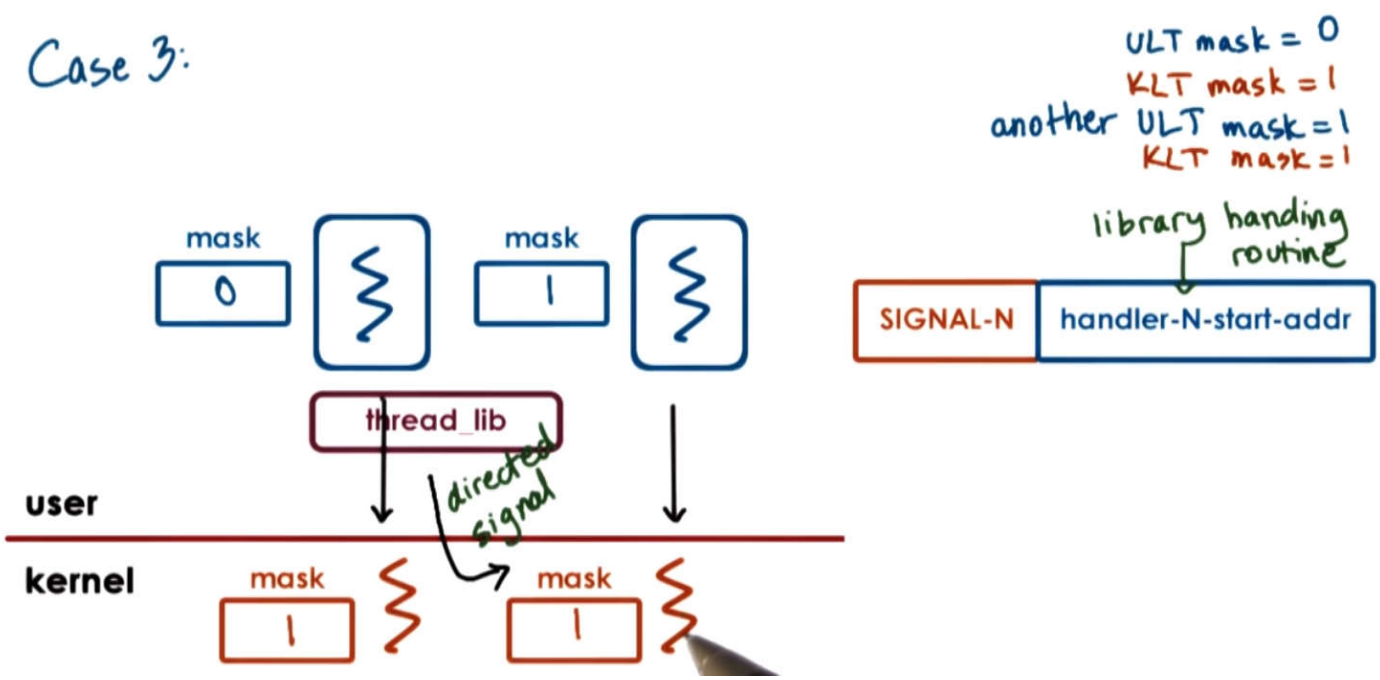
No problem here.

Case 2: ULT mask = 0, KLT mask = 1, another KLT mask = 1



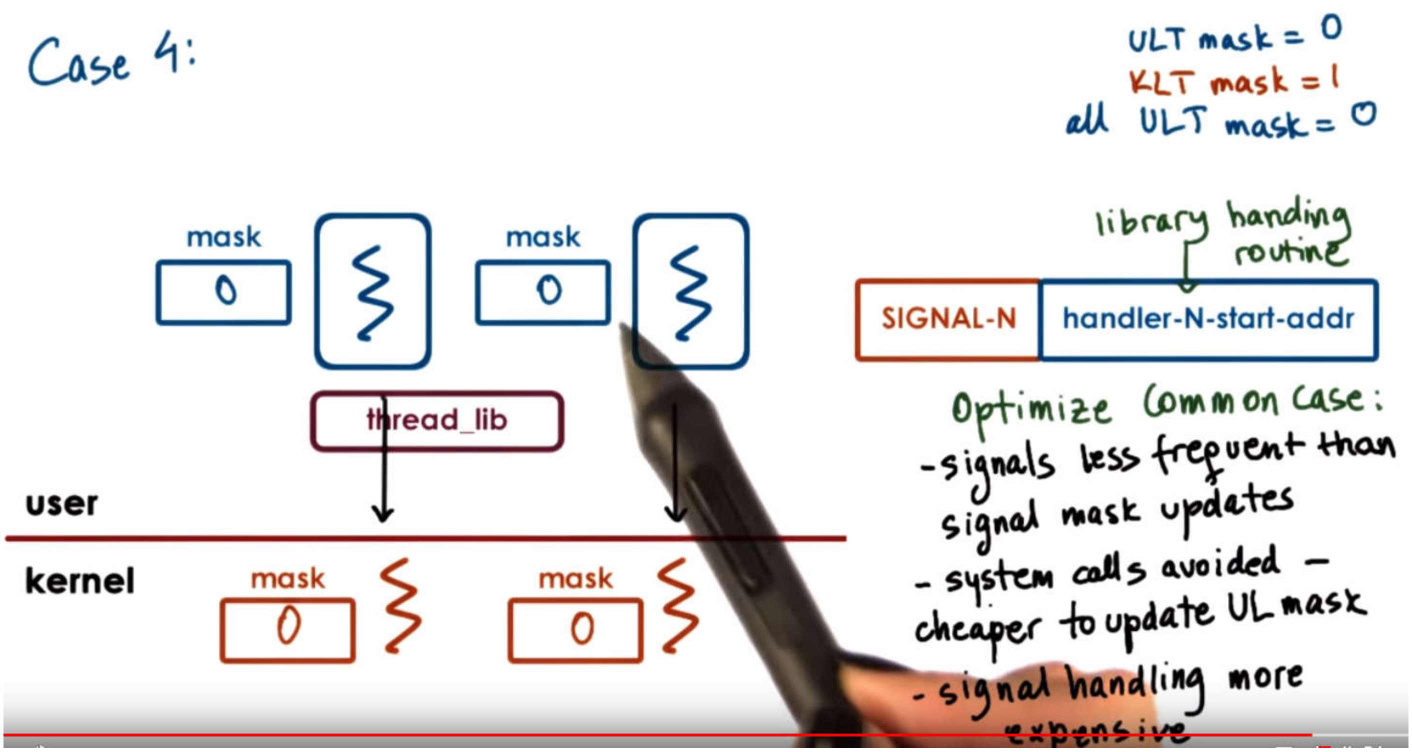
The user-level thread lib can see which ULT has the signal enabled.

Case 3: 2 KLTs mask = 1, 1 ULT mask = 1, 1 ULT mask = 0



The thread lib will direct the signal to the KLT mapped to the ULT that has mask enabled.

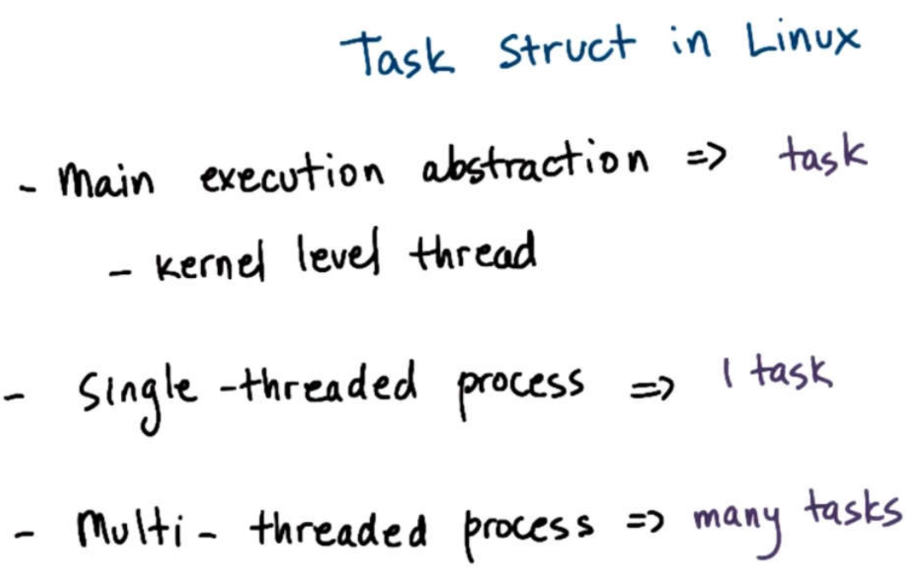
Case 4: KLT mask = 1, all ULT mask = 0



Thread lib will perform a system call to change underlying KLT mask to 0. Then the OS will reissue the signal to other threads in the process, if the thread also disables the signal, then it'll change the other thread's underlying KLT to 0.
If one of the ULT mask is updated to enable a signal, the thread lib will have to perform a system call to update the mask in KLT.

4. Threading Supports in Linux—Task

- Task: Main execution abstraction
  - execution context of a kernel level thread.



Task Struct in Linux

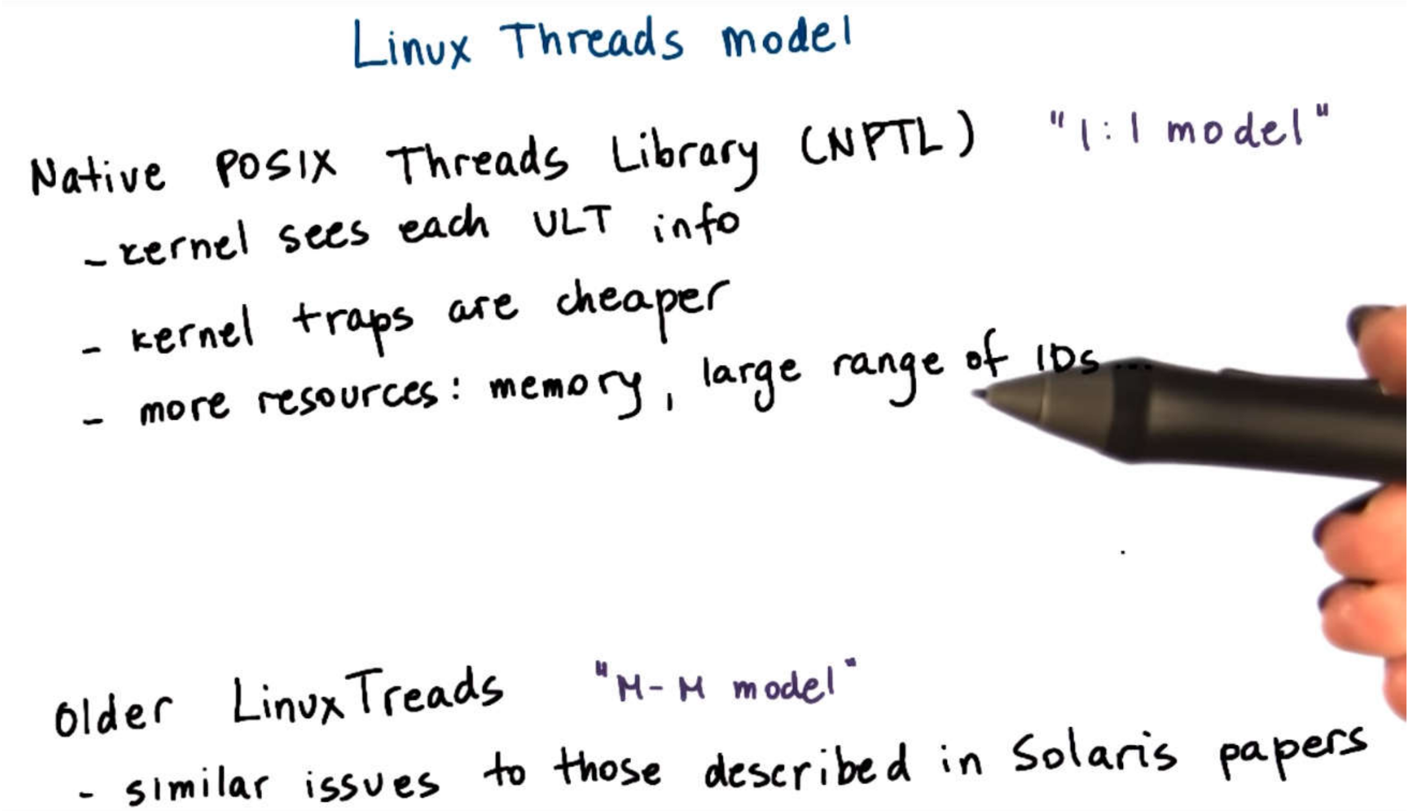
```
struct task_struct {
   // ...
   pid_t pid;
   pid_t tgid;
   int prio;
   volatile long state;
   struct mm_struct *mm;
   struct files_struct *files;
   struct list_head tasks;
   int on_cpu;
   cpumask_t cpus_allowed;
   // ...
}
```

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

- fork is implemented via clone



- kernel traps: user to kernel level crossing