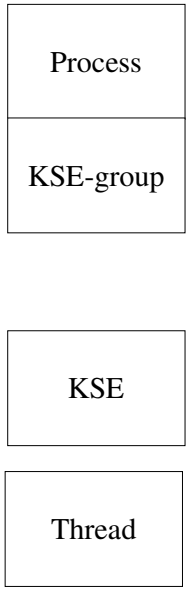
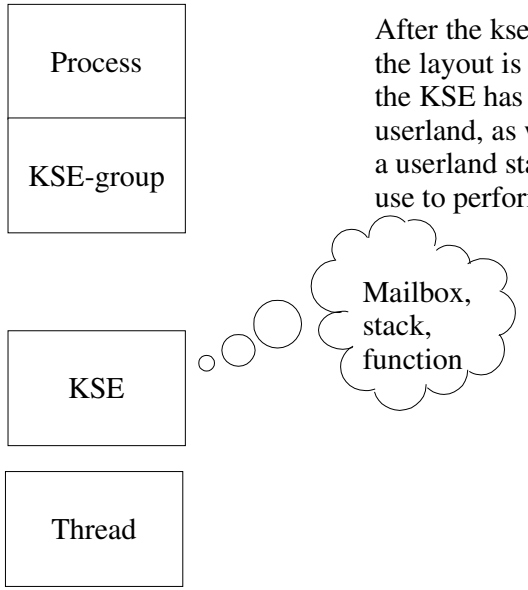
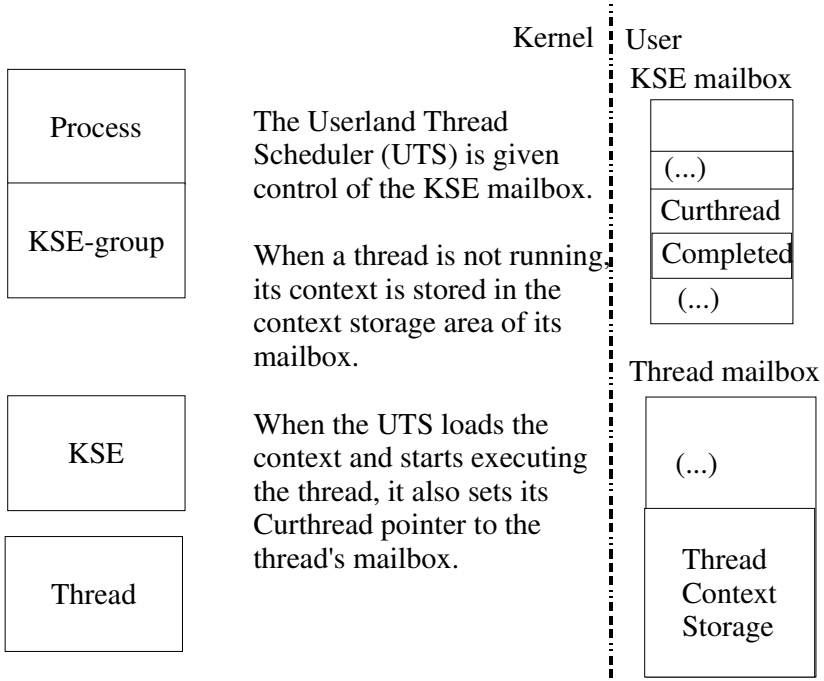


In the beginning there is a single process, with a single KSEG, a single KSE and a single thread.

Neither the thread nor the KSE have a mailbox in userland, so there is no way to make async upcalls or completion notifications. Therefore all operations in the kernel are synchronous and when a thread blocks, the entire process is blocked.



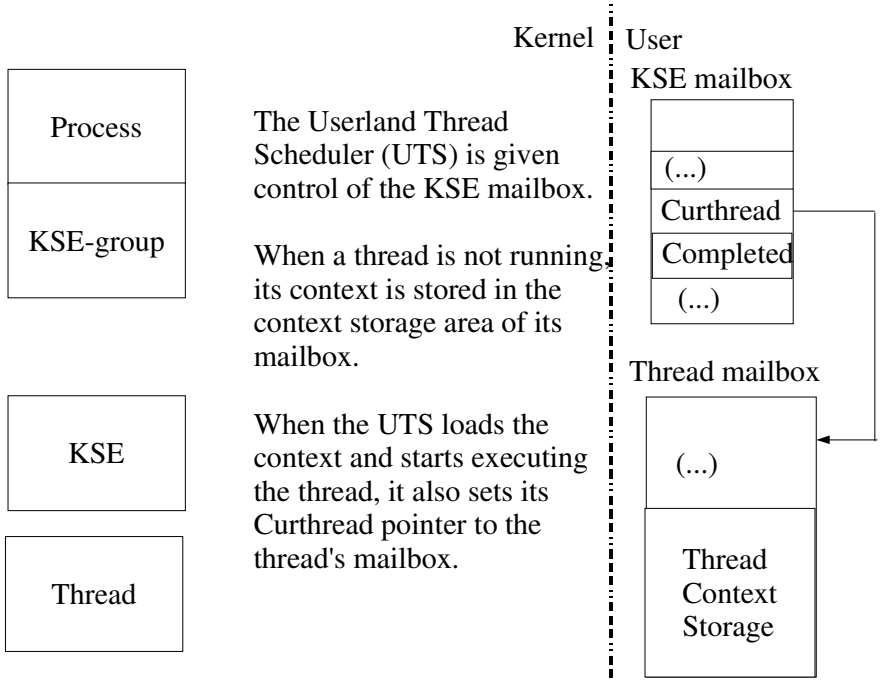
After the kse_create() syscall is called, the layout is the same except now the KSE has been given a mailbox in userland, as well as the addresses of a userland stack and function, that it can use to perform upcalls should it need to.



The Userland Thread Scheduler (UTS) is given control of the KSE mailbox.

When a thread is not running, its context is stored in the context storage area of its mailbox.

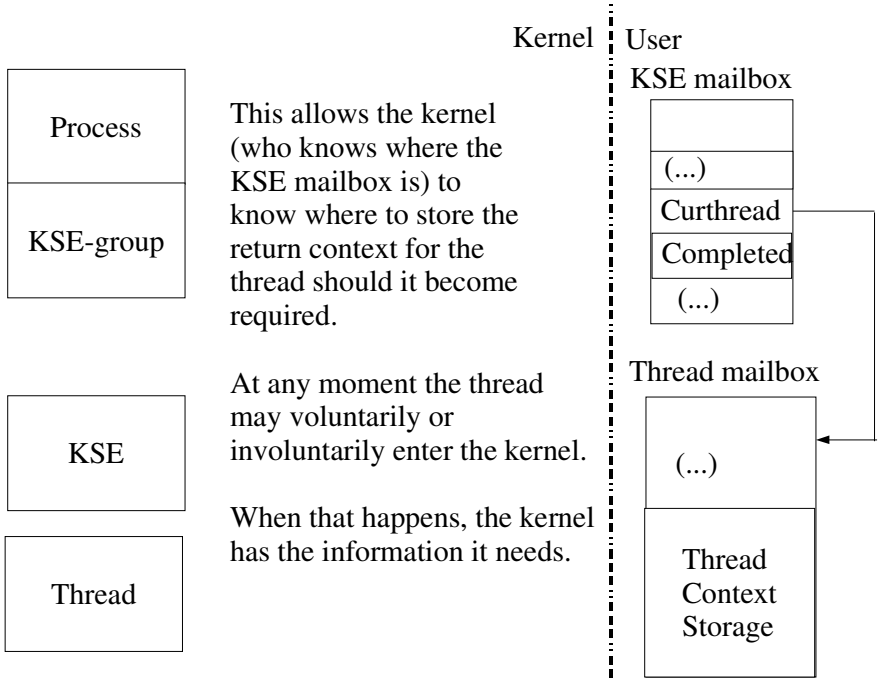
When the UTS loads the context and starts executing the thread, it also sets its Curthread pointer to the thread's mailbox.

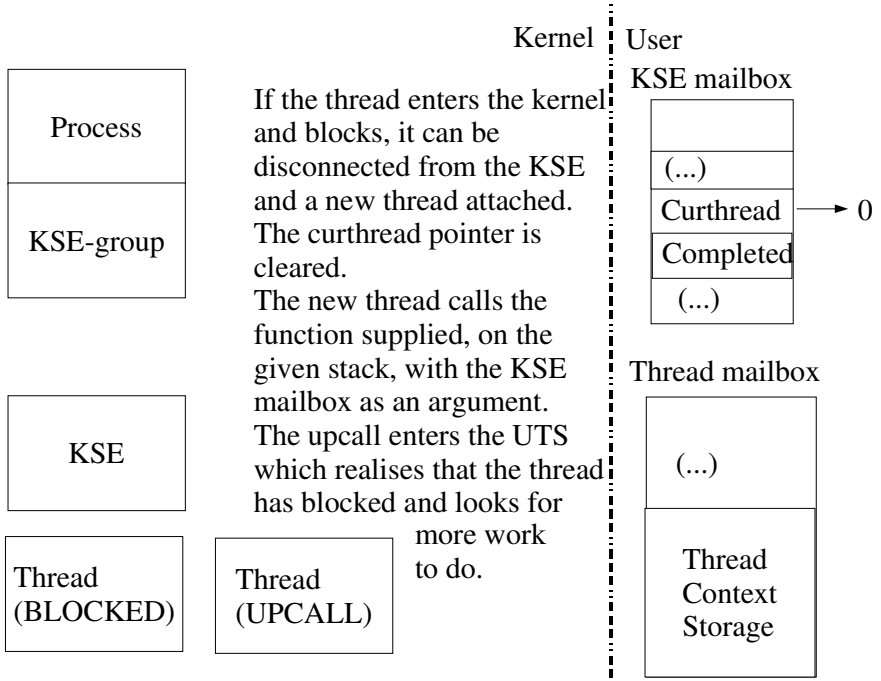


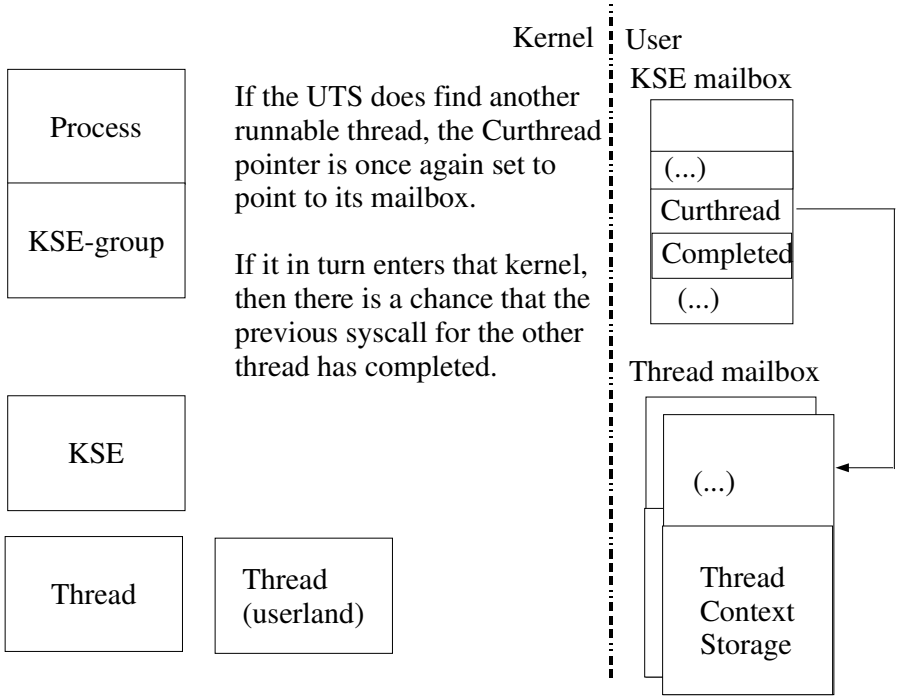
The Userland Thread Scheduler (UTS) is given control of the KSE mailbox.

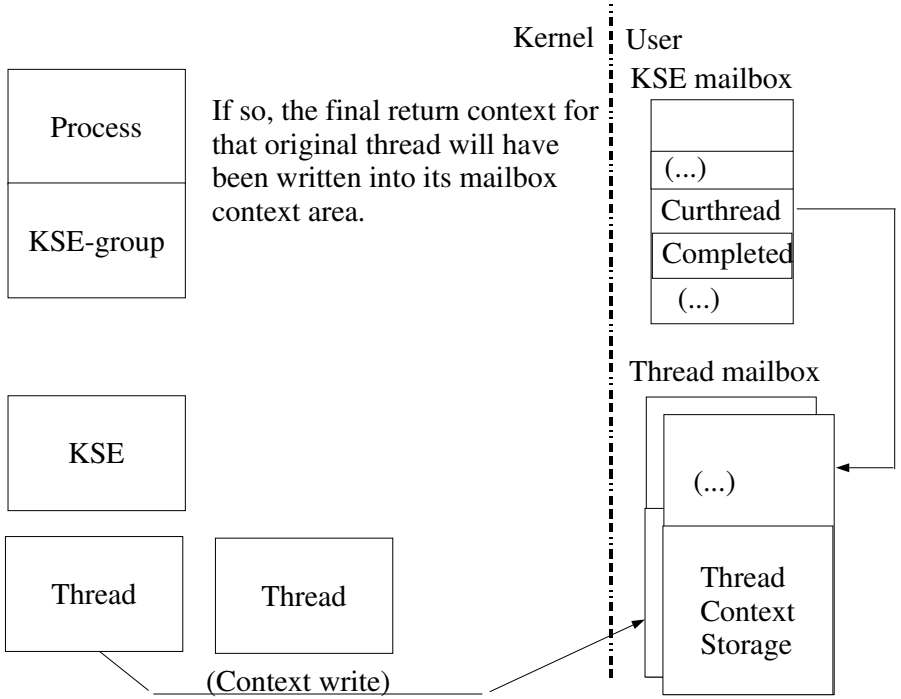
When a thread is not running, its context is stored in the context storage area of its mailbox.

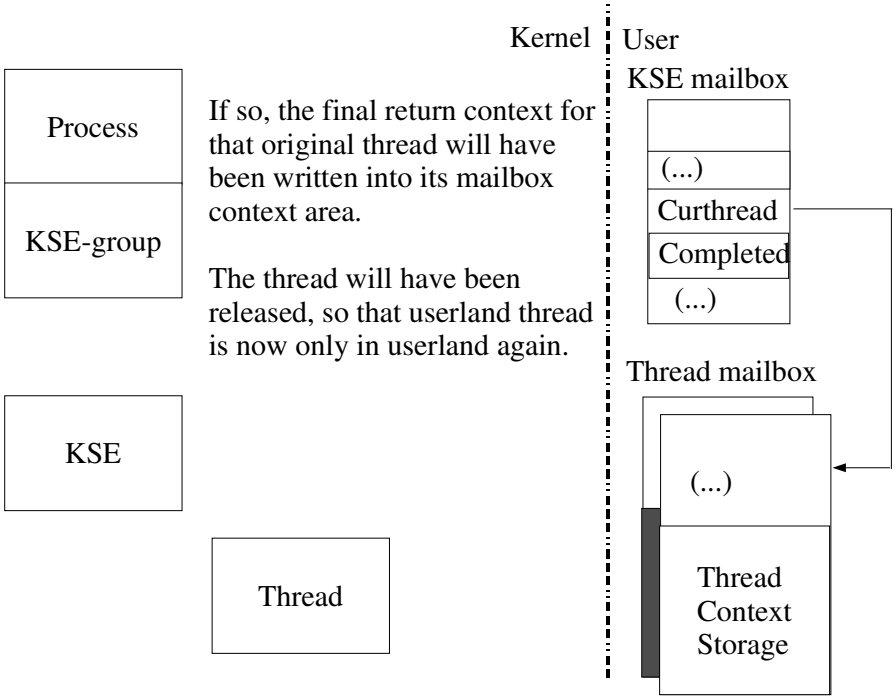
When the UTS loads the context and starts executing the thread, it also sets its Curthread pointer to the thread's mailbox.

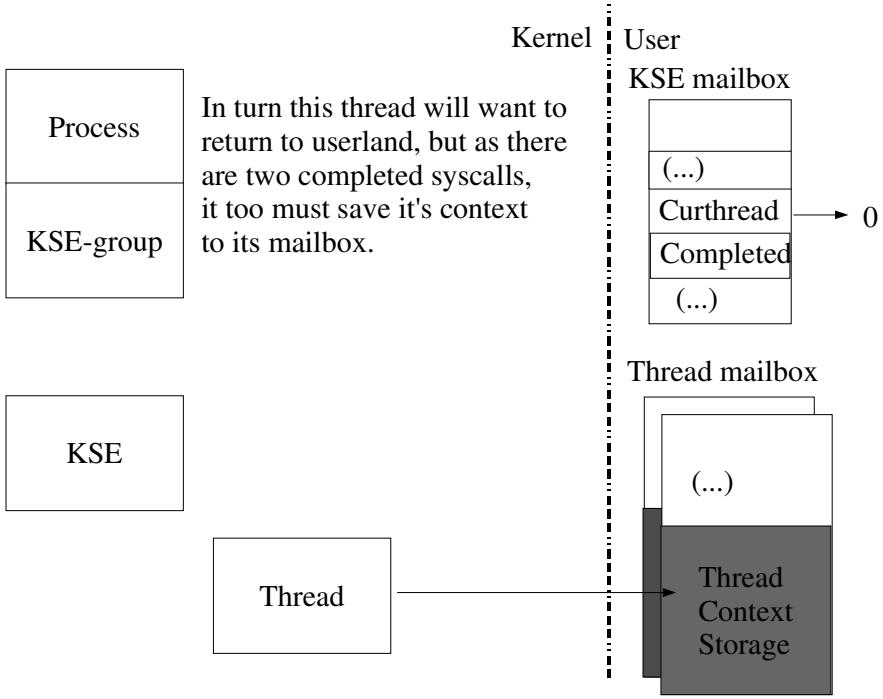

Kernel
User
KSE mailbox
(...)
Curthread
Completed
(...)
Thread mailbox
(...)
Thread Context Storage
Process
KSE-group
KSE
Thread
This allows the kernel (who knows where the KSE mailbox is) to know where to store the return context for the thread should it become required.
At any moment the thread may voluntarily or involuntarily enter the kernel.
When that happens, the kernel has the information it needs.

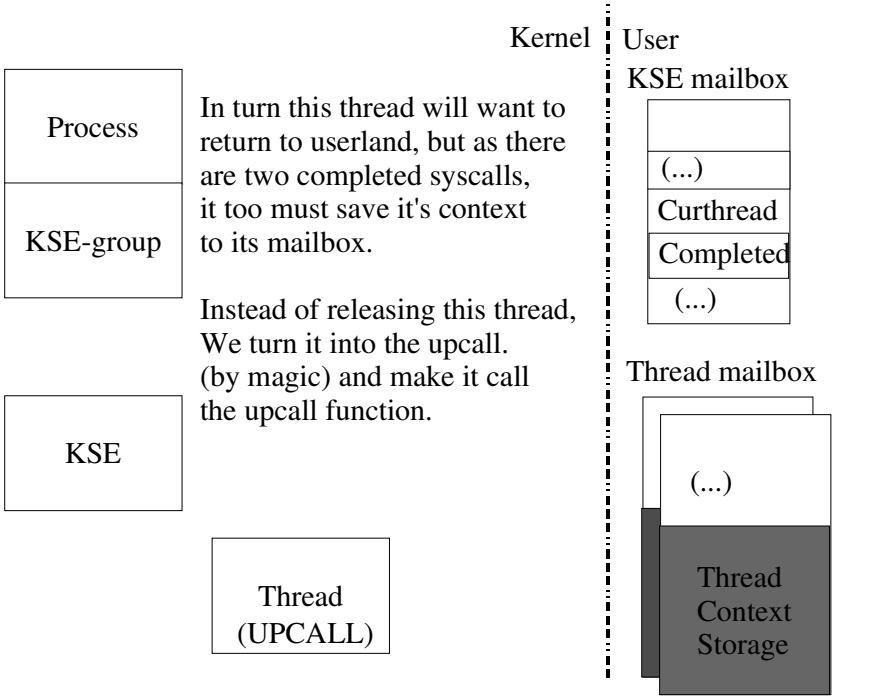

Kernel
User
KSE mailbox
(...)
Curthread
0
Completed
(...)
Thread mailbox
(...)
Thread Context Storage
Process
KSE-group
KSE
Thread (BLOCKED)
Thread (UPCALL)
If the thread enters the kernel and blocks, it can be disconnected from the KSE and a new thread attached.
The curthread pointer is cleared.
The new thread calls the function supplied, on the given stack, with the KSE mailbox as an argument.
The upcall enters the UTS which realises that the thread has blocked and looks for more work to do.

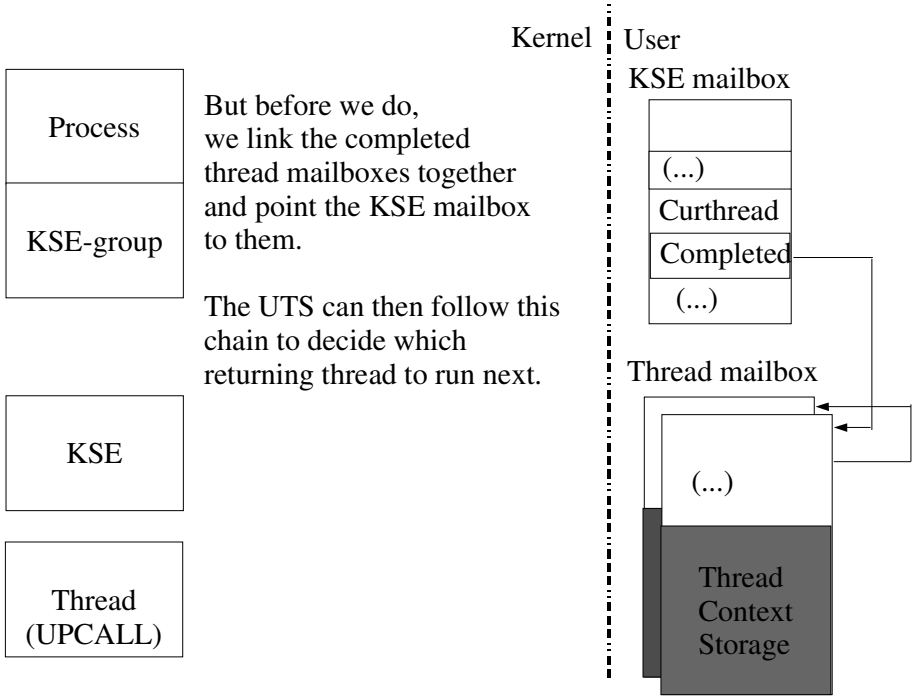

Kernel
User
KSE mailbox
(...)
Curthread
Completed
(...)
Thread mailbox
(...)
Thread Context Storage
Process
KSE-group
KSE
Thread
Thread (userland)
If the UTS does find another runnable thread, the Curthread pointer is once again set to point to its mailbox.
If it in turn enters that kernel, then there is a chance that the previous syscall for the other thread has completed.


Kernel
User
KSE mailbox
(...)
Curthread
Completed
(...)
Thread mailbox
(...)
Thread Context Storage
Process
KSE-group
KSE
Thread
Thread
(Context write)
If so, the final return context for that original thread will have been written into its mailbox context area.


Kernel
User
Process
KSE-group
KSE
Thread
If so, the final return context for that original thread will have been written into its mailbox context area.
The thread will have been released, so that userland thread is now only in userland again.
KSE mailbox
(...)
Curthread
Completed
(...)
Thread mailbox
(...)
Thread Context Storage


Kernel
User
Process
KSE-group
KSE
Thread
In turn this thread will want to return to userland, but as there are two completed syscalls, it too must save it's context to its mailbox.
KSE mailbox
(...)
Curthread
0
Completed
(...)
Thread mailbox
(...)
Thread Context Storage


Kernel
User
Process
KSE-group
KSE
Thread (UPCALL)
In turn this thread will want to return to userland, but as there are two completed syscalls, it too must save it's context to its mailbox.
Instead of releasing this thread, We turn it into the upcall. (by magic) and make it call the upcall function.
KSE mailbox
(...)
Curthread
Completed
(...)
Thread mailbox
(...)
Thread Context Storage


Kernel
User
Process
KSE-group
KSE
Thread (UPCALL)
But before we do, we link the completed thread mailboxes together and point the KSE mailbox to them.
The UTS can then follow this chain to decide which returning thread to run next.
KSE mailbox
(...)
Curthread
Completed
(...)
Thread mailbox
(...)
Thread Context Storage

So far we have seen a need for 2 structures, but we have 4. What are the others for?

If we have only one KSE, structure, coresponding to a single user (UTS) mailbox, then we can only run on one CPU at a time. In a multi CPU system we would want to run kse_create() once for each CPU. Then the UTS could schedule one thread for each KSE. I sometimes refer to KSEs as virtual CPUs, however while this is convenient at some moments it is not very accurate and I prefer to refer to their scheduler roles by calling them Kernel Schedulable Entities (KSE), by which name you have seen then refered to here (and in the code).

OK, so that's 3, but why 4?

Well,
There are reasons why we want to group threads. In some cases we may want some threads to run at different priorities to others, or we may want to in some way restrict some threads. It is even possible that we may want two threaded libraries to run their own threads without needing to know about each other or the main program knowing about them.

For this reason we have the last structure, KSE Groups. A GSE Group (KSEG) can be assigned a different set of scheduler parameters (e.g . “nice” values) and can schedule it's own threads and KSEs without any connection with anything happenning in other parts of the process (which owns all the files and memory etc.).

Threads in a KSEG with multiple KSEs (i.e. On an MP machine) can migrate around between the KSEs. This happens easily and sometimes between instructions. On the other hand threads in different KSEGs can not migrate between those KSEGs unless the UTS instances in those two KSEGs explicitly move them while they are in user space.

By the way, don't get user threads confused with the kernel “thread” structure. Each user thread is runn on top of a kernel thread, but while a user thread is not running, stored in it's context storage, there is no kernel thread associated with it. Kernel threads are only associated with user threads that are actually running, blocked in the kernel, or with the running UTS.

What's going on in the kernel?

The lifecycle of a thread coming in on a KSE enabled process might be as such.

A syscall occurs, and the thread (which has the kernel stack) is bound with the currently scheduled KSE.
The syscall proceeds, until it needs to block for a mutex.
At this time a mutex is considered a short term block and does not result in special treatment. The KSE and the thread are blocked until the mutex allows them to proceed. At some later time the thread decides to do a long term block, due to meeding to sleep on a condition variable or a device (whatever). Since this is clasified as along term blockage, the thread is unbound from the KSE and a new thread is bound in its place. The new thread is set to do an upcall.

At some later time in the future, the original thread is made runnable again. It is placed on the KSEGs run queue, and when a KSE in that group has the capacity, it is attached to it as a “guest” thread (it will already have a main “owner” thread but it will be sleeping or blocked in some way).

When the guest thread comppetes its kernel work,
it will write its userland return context to the thread mailbox and exit. This will return the KSE to the owner.

When the owner completes its work in the kernel, it will also save its return context and then it will schedule itself as an upcall. The upcall will be the next owner and will report back the two completed syscalls to the UTS.