User-Level vs. Kernel-Level Threads

Hello, and Welcome to this lesson on user-level vs. kernel-level thread management.

We will start by refreshing our memory on how kernel-level threads are used by user programs.

We do the same for user-level threads. We will focus on how simple it is to implement a user-level thread library.

We will then discuss the pros and cons of user-level thread management in comparison to kernel-level threads.

This will set the stage for a brief analysis how the kernel could help the management of user-level threads.

We will briefly explore how this is done in "Scheduler Activations", an influential system developed in the early nineties.

At the end of this lesson you will understand the limitations of user-level libraries for high-performance systems, and you will be able to trade-off the benefits of using kernel vs. user-level thread management in your applications.

Let's begin.

Kernel-Level Threads

Let's start with kernel-level thread management.

In such a system, threads are a kernel-level entity. The thread creates threads on the user's behalf. In this example, we have a process with a single thread.

When the user wants to create a new thread, it asks the kernel to do so.

The kernel now creates a new kernel level thread, which the user can utilize as a new flow of control in its application.

The same happens when a third thread is created.

As we will see later, kernel-level threads make it easy, compared to user-level threads, to take advantage of system resources, including processors.

Their disadvantage, however, is that they are very expensive, as we will see.

User-Level Threads

User-level thread management, on the other hand, is done by a user-level library, and the kernel is not involved.

When the user program creates a thread, a new user-level thread is created inside the library.

The same goes for subsequent threads.

In this way the user can run multithreaded applications without the cost of kernel intervention.

User-Level Threads using Standard C Library

User-level thread libraries are very easy to implement, and the standard C library provides a great support to do so in form of simple context management functions. . Here we see the function **getcontext**, which loads the current thread's execution context into a structure of type "ucontext".

The current context contains at least 4 fields.

- The first field contains a copy of the CPU registers, as it has been created by the call to **getcontext**.
- The second field describes the location and size of the stack of the current thread.
- The third field lists the signals that are blocked in the context of the thread,
- and the last field is a pointer to the next context descriptor, which is the context that is to be loaded when this thread "returns from its thread function".

This function **getcontext**, needs a few more functions to enable the implementation of a comprehensive thread management library.

The function **setcontext** make a previously saved context the current context. The execution continues with the context specified by the argument to **setcontext**. If we think in terms of user level threads, **setcontext** causes the context switch from one thread to the other.

The function **makecontext** creates a new context. It does this by taking a context that has been initialized by a previous call to **getcontext** and modifying it so that it will continue execution by calling a function, the function **func**, which is passed as argument.

Using the functions **getcontext**, **setcontext**, and **makecontext** it is then very easy to set up a number of threads and switch between them.

User-Level Threads: in Practice

In reality, user level threads of a process are not required to be mapped to a single kernel-level thread.

This would make little sense, in particular in systems with multiple processors.

Rather, user-level thread libraries allow their threads to be bound to more than one kernel-level thread.

The System scheduler then schedules those threads on the available processors, such that in a well-balanced system we ideally have one processor allocated to every non-blocked user-level thread.

User-Level Threads

User-level threads have a number of advantages.

- First, thread operations are very light-weight because they incur only function invocation overheads.
- Second, since it is easy to implement user-level threads, for example with the functions **get/set** and **make** context, the user level thread library designer can flexibly tailor the library to their needs.

For general-purpose multithreaded computing a pthread model may be appropriate, or a Java Thread model, or a more exotic Actors model, or others. Since scheduling is done in the user level library as well, the behavior of the user level threads can be easily tailored to the application.

- Finally, the implementation of the user level thread library requires no changes to the OS, which makes both the library and the highly portable.

Unfortunately, all forms of user-level threads have a number of limitations that makes it difficult for applications to make efficient use of system resources, including processors.

In the following we will analyze the pros and cons of user-level threads more in detail.

User-Level Threads: Advantages

Let's start with the advantages in comparison to kernel-level threads.

In comparison, kernel-level threads are costly. The thread management operations all require system calls, which requires several context switches, even for operations within the same process. This is very expensive.

This table compares the relative cost of creation (null fork) for user-level threads, kernel-level threads, and kernel-level processes. We see that a kernel level thread is about 30 times more expensive to create than a user-level thread. Of course the numbers are much worse for processes.

Similarly, the context switch cost (signal-wait) is an order more expensive for kernel-level threads than for user-level threads. Again, the switch between processes if of course again higher. For comparison, we see the cost of a procedure call and that of an exception.

Kernel-level threads are also more difficult to tune to the particular application. While the application designer can develop a customized user-level library for their application, they would have to use the system provided, general, kernel-level thread API, which may be overly limiting.

User-Level Threads: Limitations

Now that we know the advantages of user-level threads, what are their limitations?

Generally, some designers argue that kernels are difficult to use and difficult to integrate with system services. What does this mean?

Kernel events, for example when a running thread is blocked, or when a thread resumes after being blocked, or when the system simply switches from one thread to the other, the application is not made aware. This can lead to all kind of complications, as we will see.

Also, the kernel schedules kernel-level threads largely obliviously to the applications. At best the application can define a few scheduling parameters, such as priority of a thread, but other than that, it has no control over how its threads are being scheduled.

Let's illustrate the first of these two points with a scenario:

When a user-level thread blocks, maybe because it is issuing a blocking I/O operation, then the kernel-level thread blocks as well.

If this happens, the application just lost a processor. Let's further illustrate this scenario in pictures.

The user application is bound to one kernel level thread, and therefore has one user level thread running.

After a user-level context switch, which of course does not involve the kernel, another thread runs.

After a second context switch a third thread executes.

At some point this thread makes a system call, and switches to kernel mode.

While it executes the system call, it yields the CPU, and blocks.

We see that the operating system has no way to switch to any of the other ready user level threads, because these threads are not visible to the OS.

As a result, the application blocks looses its processor. The entire application blocks.

How can we deal with this?

One naïve way to avoid this situation is to simply create more kernel threads for the application. If a kernel thread blocks, there are other kernel threads that can take over.

This works fine until the thread unblocks. At that point we have too many kernel threads, and if we have more kernel level threads than processors, the OS, because it schedules the kernel threads, starts deciding on behalf of the application, which user-level threads get to run next. As a result, user-level threads the need service may be delayed.

One way to solve this problem could be to schedule the kernel-level threads of an application in some sort of a round robin fashion.

This has also all kinds of difficulties.

For example, the user-level thread library may implement locking with spin locks, which busy loop while they wait for the lock. This is very easy to implement and handle efficiently in the user-level thread library.

If a thread that holds a spin lock is preempted, then any user level thread that wants the same lock will start spinning as well, until the lock holder thread gets to use the CPU again.

These are some of the difficulties when using user-level threads for high-performance applications.

User-Level or Kernel-Level Threads

Given these pros and cons of user and kernel-level threads, the system designer faces a dilemma.

Should they employ kernel-level threads, which provide seamless access to system resources, including processors, but perform poorly, or should they employ user-level threads on top of kernel-level threads, which perform well, but have the all the limitations that we just mentioned?

How about Kernel-Level Support for ULTs?

What if we could add the appropriate kernel-level support for user-level threads such that the designer could build their user-level library in a way that all these problems would go away?

What if we could build user level threads that mimic the behavior of kernel level threads?

By this we mean that no processors would idle when user-level threads are ready.

And as a result, one could do effective multiprogramming within the process and across processes.

But at the same time keep the performance overhead low, preferably similar to that of user-level threads?

One way to enable this would be to build the kind of support into the kernel, that would allow, just as now, allow the kernel to allocate physical processors to processes.

But it would allow the ULT library to control which of the threads get to run on the processor. Within the limitations that we discussed, this is possible in ULT libraries today.

However, the UTL management would know about when kernel-level threads are suspended, and when they are resumed, in the kernel.

Similarly, the UTL system can request and release processors on demand. This would be done by requesting/releasing kernel-level threads.

One Solution: “Scheduler Activations”

Anderson and others proposes “scheduler activations” to do just that.

A simplified way to think about scheduler activations is as kernel-level threads that provide an additional API to the ULT library.

For example, the ULT library can make system calls to request additional processors in form of kernel-level threads, and to release a kernel-level thread, basically indicating that there are not sufficient threads active, and that a processor would become idle.

The kernel would provide information to the ULT, on the other hand by providing a number of upcalls. These would indicate to the application that a kernel thread has been blocked, or unblocked, that a processor has been preempted, or that a processor has been added.

Handling Blocked Threads using Scheduler Activations

Let's see how a ULT library can take advantage of this.

The application currently runs on a single kernel level thread, which means that there is one processor allocated to this application.

One thread is active and is executing.

At some point it makes a system call, and after a bit it gets blocked inside the system call.

A normal ULT library would not know about this, and would not be able to switch in another thread, for example. Even if it knew, there is no mechanism to do so.

In scheduler activations this is a bit different. When the OS detects that the first thread is blocked, it creates a second kernel-level thread, and lets the ULT library in the application know about it.

The application now can resume another thread to run on the new kernel level thread.

Let's see what happens when the original thread gets unblocked.

Resuming Blocked Threads using Scheduler Activations

The thread unblocks and resumes execution. The kernel detects this, and preempts the second thread, that now one thread too many.

The application is informed about this by an upcall, and it preempts the second running user thread.

Now we are back to where we were before the first thread got blocked.

Conclusion: User-Level vs. Kernel-Level Thread

We have come to the conclusion of this lesson on user-level vs. kernel-level threads.

We have refreshed our memory about KLTs and ULTs, and we have done a brief comparative evaluation.

We have pointed out some of the difficulties that application designers face when trading-off ULTs vs. KLTs. and we have discussed how additional kernel level support would help.

We have looked at Scheduler Activations, which are one well-known example of such kernel-level support.

We hope that you enjoyed this lesson on ULTs vs. KLTs and that you learned about the capabilities of both approaches and when to use each.

Thank you for watching.