Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism

by Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy


Presented by Daniel Cristofani
• Threads can be implemented at the user level or the kernel level.

• Either approach has serious problems.

(picture from Andrew Tanenbaum's *Modern Operating Systems*, 3rd ed., p. 107)
Kernel Threads

- All thread operations are system calls.
- This means lots of context switches, so they're slower by at least an order of magnitude.
- Kernel threads are scheduled by the OS's scheduler.
  - It doesn't know anything about the app's structure or needs.
  - Or if it knows a little, giving it that information is more overhead.
Kernel Threads, cont'd

- Kernel threads are scheduled pre-emptively just like processes.
  - They have to be, or one process's thread could take way more than its share of CPU time.
- Kernel threads are overgeneralized to support all reasonable uses.
User Threads

• User threads and their scheduler run within a kernel thread or a process.
  – So the operating system can control them.

• Thread operations are calls to a user-level library.
  – Basically a regular function call, so they're fast.
User Threads, cont'd

- User-level thread schedulers can be modified freely for each application.
  - And they have fast access to all user-level state for making scheduling decisions.

- User threads are scheduled cooperatively
  - They tell the user-level scheduler when it can give control to another thread.
  - Threads in the same app have to trust each other.

- So what's wrong with them?
Consider when a kernel thread does a disk read:

- It does a system call which traps to the kernel.
- The kernel sends a request to the disk,
- then blocks that thread, because the response will take forever.
  - The kernel saves the thread's state, and
  - gives that processor to another thread.
  - This makes sense because there is nothing else that thread could do before it gets the information anyway.
Kernel thread disk read, cont'd

- Eventually the kernel gets the data from the disk.
- The kernel puts that thread back on the ready list.
- Sooner or later the scheduler gives it a processor.
- It returns from the kernel to user space and continues.
• When a user thread running within a kernel thread (or process) performs a disk read, the exact same sequence happens.

• But the kernel doesn't know anything about any user-level threads.

• When it blocks, it blocks the current kernel thread.

• The kernel thread is tied up forever waiting for the disk to spin.
  
  - No other user-level threads can use that kernel thread meanwhile, even if there are some waiting. The app's capacity is diminished without notification.
• The same thing happens when a user thread blocks at the kernel level for other reasons, e.g. page faults.

• The user-level scheduler can't do anything about it, and doesn't even know it has a reduced number of kernel threads to work with.
  – This can even lead to deadlock if too many threads are blocked.
    • The user-level threads that could make progress and unblock the others can't be run because there are no unblocked kernel threads left to run them on.
Another problem:

- By scheduling kernel threads, the kernel scheduler is (indirectly) scheduling user threads, obliviously.
  - It will pre-empt a kernel thread that is running a user thread it knows nothing about.
  - Murphy's Law says it will pick the worst possible.
  - It may pre-empt a thread holding a lock.
    - Or one that other user threads are otherwise waiting on.
    - Other user threads might have to block or spin their wheels for a long time.
  - It may pre-empt a high-priority user thread for a low-priority one, or an idle scheduler.
In short:

- The kernel is scheduling kernel threads, and thus user threads, without knowledge of the user thread state.
- The user-level scheduler is scheduling user threads within kernel threads without knowledge of the kernel thread state.
The proposed solution is essentially:

- The kernel allocates physical processors to apps.
- The user-level scheduler of each app allocates user threads to processors.
• The kernel will tell an app when it gives the app a new processor, or takes one away.

• The kernel will tell an app when a thread blocks or unblocks in the kernel.
  – While it's blocked, the app's scheduler can give that processor to another thread.

• The scheduler can rethink its allocation of threads to processors based on this information.

• The scheduler can tell the kernel when it needs more or fewer processors.
  – The kernel will do what it can.
An Example:

- High-Priority App to Kernel:
- "I could really use another processor right about now."
- Kernel interrupts processors 2 and 3 from Low-Priority App and saves their state.
Example, cont'd:

- Kernel to scheduler of High-Priority App:
- "Here's processor 3.
- In fact I'm now using it to tell you it's yours now. Use it to run any thread you like."
Example, cont'd:

- Kernel to scheduler of Low-Priority App:
- "Hi. I just took processors 2 and 3 from you.
- Here's their register state.
- And you can have processor 2 back. Here it is."
• The kernel grabbed two processors precisely so it could tell LPA what just happened.
• If there weren't two available, LPA is frozen until a processor frees up, and then it gets told:
  – "Hi, I took your last processor a while ago. Here's its state and here's processor 6 for you to use now."
• This is okay because LPA is low-priority.
• The apps are told *which* processors they have so they can factor locality into their scheduling decisions.

• They can try to give each processor the same thread it had before, or a thread that is likely to want to use the same data.

• This makes it more likely that some of that data is still in cache.
The authors call their invention "scheduler activations", because the kernel transfers control to the scheduler at a designated entry point when it needs to give the scheduler information.

Each "activation" has some associated data at the kernel level and some at the user level.

- Kernel stack for saving the state of threads that are blocked in the kernel or pre-empted
- User stack for communicating with the user-level scheduler, and for it to run in
A scheduler activation is implemented similarly to a kernel thread in some ways, but it is not paused and resumed directly by the kernel.

Instead, the kernel gives an activation (basically an abstraction of a processor, plus some piece or pieces of status information) to the user-level scheduler. The scheduler uses that information (and other information it has) to decide what user thread to run on that processor.

Running activations are meant to correspond one-to-one with processors allocated to an app.
There are four things the kernel can tell the user-level scheduler.

1. Here, you can have this processor.
   - Every activation includes this one. (These four kinds of messages are often combined.)
   - The kernel makes that processor execute a jump into the user-space scheduler.
   - This is called an 'upcall' because the kernel is basically calling a routine that's in user space.

2. Processor X has blocked in the kernel. I have that thread's kernel state saved. Put that thread on your 'blocked' list.
3. Processor X has unblocked in the kernel.
   - Here's that thread's register state.
   - You can take its other user state out of the user stack of the "scheduler activation" it was running under, and put that thread back on your ready list.

4. Processor Y has been taken from you.
   - Here's that thread's register state.
   - You can get its other state and put that thread back on your ready list. (as above)
• Notice that these four messages correspond neatly with this graph.
  – It shows the different states a process or kernel thread can be in: running, ready to run, or blocked.
  – Whenever a kernel thread system would change the state of a thread "transparently", scheduler activations contact the user-level scheduler.
• It can update its information and rethink.

![State Transition Diagram]

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

**Figure 2-2.** A process can be in running, blocked, or ready state. Transitions between these states are as shown.

(picture from Andrew Tanenbaum's *Modern Operating Systems*, 3rd ed, p. 90)
There are two* things the user-level scheduler can tell the kernel.

1. I'd like X more processors if and when you have them available.

2. I'm not using this processor - you can have it back.

* (Actually there are at least three. Another one is "I should really be running thread Y instead of thread Z. Please pre-empt thread Z and give me its processor to run thread Y on." But that's somewhat obscure.)
Another Example.
The Big Picture, again:

- The kernel knows when an app wants more processors, or when it has more than it can use. That, plus the app's current priority, is all it really needs to know to allocate processors.

- More importantly, at any given time, each user-level scheduler knows exactly how many processors it has to work with, and which of its threads are in fact running, on what processors.
  - That is exactly what the scheduler needs to know in order to make efficient use of whatever processors the kernel can afford to give it.
  - This is a much cleaner division of labor.
  - The two don't trip over each other's feet anymore.
• Of course, the authors did a proof-of-concept implementation.

• It's pretty fast overall.
  – Almost as fast as user threads in the common case when the kernel doesn't need to be involved.
  – Five times as slow as kernel threads in the rarer case when the kernel does need to be involved.
    • Excuse: their prototype is untuned Modula-2+
  – Overall application performance appears to be better than either.
  – Their invention seems totally viable.