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

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

Presenter: Quan (Cary) Zhang
Each stack of the user thread uses both for user thread when running and for saving the context when the running user thread enters kernel.

Every kernel thread which is created by XtCreateCustomThread will save the context of the running user thread into respective stack, and change the esp to the beginning address of user thread scheduler and change the esp to the scheduler stack.

Every time when the kernel thread returns to the user level, the user thread scheduler will get starting, and then decide which user thread will be chosen to run in this kernel thread by restoring the context in the stack. Use the mutual lock to ensure correctness.
• Parallel Computing is the elicitation of appearance of thread, but the essential factor is to achieve higher performance

• Parallel program can be realized by three ways:
  – View from performance
  – View from flexibility
Process (lose at the time delay of communication controlled by memory)

Kernel level Thread (lose at the time cost in the transmission between kernel and user space)

User level Thread needs nothing except the switch between different functions
Kernel Thread must compromise to the generality

Different kinds of user thread can comply with different kinds of requirement from outer user, without the necessity of kernel modification
• Q: What is the borderline between Kernel threads and User level thread?
• However, the user level thread is at the beck and call of kernel. In other word, the user level thread can never ever communicate with the kernel in order to arrange the resource initiative according to the factual situation, which can affect the performance of user thread, or even fallacy.
Q: Why the communication between the kernel and the user-level thread management system via scheduler activation has less overhead, comparing with possible ways to communicate between the kernel and the user-level for user-level threads built on the Traditional Kernel Interface?
Q: What kind of user-level thread operations that might affect processor allocation decisions?
• So, the design of Scheduler Activations is to let kernel not so arbitrary, but let it inform to user thread system, that is, to vector control from the kernel to the thread scheduler on a kernel event, letting the user thread scheduler take its effect, acting as a real scheduler.

• Q: Is the Scheduler activation lightweight? How is it implemented in there system?
• The Schedule Activation is activated if there exists a user thread when the state of kernel thread mapping to itself is changed.

• There is a user thread scheduler, communicating all the schedule activation, in order to have complete control over which of its threads are running on the allocating processor.
• **Supplement:**
  – The user thread scheduler can schedule each user thread preemptively under the help of kernel.
  – From the view to kernel, the kernel needs no knowledge to what is happening to the user level.
  – Some specific situations at the time of returning from kernel:
    • No user thread
    • Event handler
    • Interrupt that can cause continuing interrupt
• In order to prevent the situation that a kernel thread runs only user thread scheduler (nothing), the user thread scheduler also add two system call:
  – Add more processors:
  – This processor is idle:
• The critical section problem can cause poor performance or even deadlock.
• One solution is to use the free lists of thread control blocks (implemented in the Windows Kernel)
• Prevention mechanism break the semantics of the kernel
• Recovery, a mechanism that let the user thread continue its execution, is a more natural way to solve the problem
- Implementation Details:
  - Add an additional suit of thread-related system calls, rather than revise the original ones.
  - Copy the critical section code, eliminating the overhead on lock acquisition and release, simplifying the common case that the preemption does not occur on the thread, yet complicate the other case.
• Q: With regard to the mechanism mentioned in the paper for avoiding threads in the critical section, which is running a copy of that section, Where does this copy reside in the address space? (Is it in the same address space which the original code reside?) How does the switching happens to the new copy?

• I do not see much difference between their explanation of dealing with critical section and this approach. I would say this introduces an overhead.
While handling of pre-empted threads that are in a critical section, they state that a context switch is made to allow the thread to finish execution, or more accurately transfers control to the identical position in a copy of the critical section which then completes before handing control back. On which activation is this run? If it is run on the same one as the pre-empted thread, then it is effectively the same as waiting for the original thread to unlock before exiting, like Psyche and Simmunix. If it is on another activation in the thread, then what happens if all the threads are currently in a critical state? Or are they time spliced over an existing activations?
• Improvement of Performance (See paper)
  – Thread Performance
  – Upcall Performance
  – Application Performance
Discussion

• Since User-level thread system can notify the kernel the application needs more or fewer processors, so is the system robust to malicious user level behaviors who simply created a large amount of thread and thus apply for more processors in order to influence the kernel's allocation decision? What is the balance between this kind of malicious behavior and a task that really need a heavyweight?
• To make each program consume fair amount of resources, the authors proposed to favor address spaces that use fewer processors and penalize those that use more, utilizing this heuristics may work great in some cases, however, it is not good for computation intensive tasks, so are there some alternatives available? Can we measure the actual resource usage and try penalize those processes which use more than necessary amount of resources?
• As for the debugging consideration, the kernel assigns each scheduler activation being debugged a logical processor, so what will happen if the debugging system itself corrupts? Since no upcalls into the user level thread system will be made.
• Early in this paper the authors lay out the case against kernel-level thread management (1:1 threading). The authors chiefly complain about costs incurred (cost of crossing protection domains for thread management, cost of a 'general' scheduler). The authors state that these are inherent to a 1:1 model. Couldn't a proportion of these costs be artifacts of early 1:1 threading implementations?
• BSD seems to be replacing scheduler activations with the model of implementing user threads over kernel threads (1:1 threading). What are the particular reasons for reverting back to an older, and possibly slower scheduling mechanism? Was some of the lack of performance of kernel threads also a result of implementation issues?
- SA's have been implemented time and time again for various systems (Mach, FreeBSD, NetBSD, Linux), but in each case SAs were eventually dropped). Are SAs just hard to get right, or is the problem more fundamental?
• To deal with inopportune preemption, this paper proposed to continue execution of user thread before it quits the critical section, I think it will decrease the performance of concurrency, is there some performance study for this issue? And is there some other solutions?
While handling of pre-empted threads that are in a critical section, they state that a context switch is made to allow the thread to finish execution, or more accurately transfers control to the identical position in a copy of the critical section which then completes before handing control back. On which activation is this run? If it is run on the same one as the pre-empted thread, then it is effectively the same as waiting for the original thread to unlock before exiting, like Psyche and Simmunix. If it is on another activation in the thread, then what happens if all the threads are currently in a critical state? Or are they time spliced over an existing activations?
• The paper implies a one-to-one mapping of threads onto processors. In section 3.1, the example says that in normal operation, "threads can be created, run, and completed, all without kernel intervention". If the user level application creates a thread during its execution, doesn't it need to request another processor (because of the one-to-one mapping), meaning that any user level application that uses more than one thread does require kernel intervention? Because the kernel needs to assign a new scheduler activation for the new processor?
• The paper says that (in 3.2), when a user level application notifies the kernel that it has idle processors, if the processors are not needed, then the kernel leaves them assigned to the application. Wouldn't putting them back on the free list be more efficient, given a higher probability that a different user-level application, or a new user-level application will need those processors?