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

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

CPSC 508 - Operating Systems

September 20, 2006
Presented by
▶ Alfred Pang

About the paper

I had help:
▶ http://web.cecs.pdx.edu/ walpole/class/cs533/winter2006/home.html
Kernel Threads

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

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska,
Kernel Threads

Advantages

▶ Comes with OS!

Disadvantages

▶ Heavy weight
▶ Inflexible one size fits all (scheduler)
User Threads

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

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska
User Threads

Advantages

▶ Provided as a library
▶ Low overhead - higher performance

Disadvantages

▶ Run to completion (not always a problem)
▶ Kernel unaware of user level.
▶ Problems comes from interactions with kernel thread.
User Threads
Right Tool for the Job

When to use kernel threads?

▶ I/O, high responsiveness

When to use user threads?

▶ Context switches of kernel threads too expensive.
Scheduler Activations - How it works

- User-level thread system notified of kernel events.
- Two levels of policies: processors allocation, thread scheduler
Kernel allocates processors

- User level notifies kernel when it needs more or fewer processors.
- Kernel allocates processors as appropriate, notifying user-level.
- Kernel preempts away processors as appropriate, notifying user-level.
User level system thread scheduler

- Each address space controls which threads to run on its allocated processors.
- Run as user-level threads (with light-weight advantages)...
- except when they get blocked in the kernel...
- scheduler activation is "freed".
- User level will be notified by kernel when it is ready to run; but up to user level scheduler to decide when it runs.
Characteristics

- Common case is when thread operations do not need kernel intervention.
- Proper behaviour (priority, handling blocks, wakes).
- Allow different scheduling policies.
- Transparent to programmer (library).
Fig. 1. Example: I/O request/completion.
Issues

- Dishonest resource requesting
- Handling Critical Sections - recovery by allowing lock to run until CS finished
Implementation

- Topaz (OS)
- DEC SRC Firefly multiprocessor workstation (6 processors)
- FastThreads (user-level)
- Reuse activation structures
- Special CS handling
- Debugging support
Performance

- Small degradation of Null Fork, Signal-Wait
- Upcall performance - "The signal-wait time is 2.4 milliseconds, a factor of five worse than Topaz threads."
- Application for benchmarking - O(N log N) solution to N-body problem
Speedup of N-Body application, 100% memory available

- Topaz threads
- orig FastThrds
- new FastThrds

![Graph showing speedup with varying number of processors for different thread configurations.](image)
Execution time of N-Body application vs available memory
Strengths

- The idea worked!
- Binary compatibility maintained in implementation.
- Correctly handles preemption in critical section.
Weakness

- Performance graphs are suspicious.
- Only one app - will this work with different workloads?
- Trying to explain the poor performance of the upcall (by saying that it performs just as poorly as another work by a similar factor).
- Increased overhead when compared to pure kernel threads or user threads
- Interactions of processor allocation policy and thread scheduling policy? Vulnerable to gaming.
Other implementations

Perhaps why this hasn’t taken off...

Also

- Is this all too complicated and fragile for improved performance in such a narrow scenario?
- Graceful system degradation under heavy loads?
- Can’t we all just use async I/O?
Questions

- What are the precise workload conditions that would make scheduler activations a superior solution?
- Are scheduler activations in use today? (Solaris LWP?)
- What happens to the performance under X conditions?
- Is it costly to do X?
- Can the X algorithm prevent gaming?
Your questions

- While context switches are fast in this model, will this performance increase always outweigh the overhead required to have communication between the OS and thread library schedulers? (MD)

- From a software engineering perspective, does it not seem fundamentally flawed to replicate scheduling logic in both the user-level threading library that is already present in OS? (MD)

- Is the assignation of virtual processors too coarse a mechanism for load balancing? (MD)
In the Solaris system, LWP (lightweight process) is provided as an interface between kernel threads and user-level threads. Comparing LWP with scheduler Activations, what is the similarities and differences? (HS)

The issue of preempting a user-level thread while it is in a critical section is dealt with through recovery rather than prevention. (Section 3, page 14-15). Does the proposed technique really avoid the issue that was raised for not using prevention – violating semantics of address space priority. How does ‘temporarily’ continuing using a context switch differ from holding off the preemption? (KL)
What is the main difference between upcalls and interrupts? (AE)

The Scheduler Activations combines features of user and kernel level threads. This ends up having a structure of n user level threads mapping to m virtual processor. How is this approach different from using a thread pool? (IS)
As an improvement to the scheduler activations design, the implementation allows temporary continuation of critical sections so that a user space thread inside a critical section (the critical sections are defined at compilation) can relinquish the processor only once it reaches a “safe” place. What happens if a thread needs to be preempted while it’s in a critical section that does not end (due to a bug for instance) (JSL)?
From Figure 2 Speedup of N-Body application versus number of processors, why is the performance of orig FastThrds and new FastThrds very closed? (GH)
Referring to the example in figure 1; when the priorities of the threads are also considered it’s stated that the user level will tell the kernel which processor should be preempted for upcall. How will the user level be aware of the I/O completion (and ask kernel to preempt specific processor) before the kernel makes an upcall?
How would the system deal with the (common) case where there are no ready threads? If the idle loop is in the user-level thread system, how would the HLT (x86) instruction be used, and would dynamic frequency-voltage scaling work? (At least in the x86, both are privileged)