Lecture 24: Scheduler Activations and First-Class User-Level Threads

Mothy Roscoe and Joe Hellerstein

November 21, 2005

Background

We saw last week that whether you use events or threads, you always hit problems:

- Events don't solve the problem of long-running handlers, or dealing with multiple processors.
- Threads are hard to synchronize or schedule - the only way to control which thread is running is with locks and the like (which is not what they're for), or prodding the [kernel] scheduler.
- Neither events nor threads can deal effectively with split-phase operations. These are not just blocking calls (though they are a problem with many operating systems), but also reschedules (other things are running), and other blocking events such as page faults.

The problem here is not the OS API to other operations, but that the scheduler in the kernel is hidden from either threads or events.

Both papers are about making the scheduler visible. Interestingly, they appeared at the same time in the same conference.

It's clear you need threads to exploit multiprocessors where the model is fine-grained parallelism model discussed: threads in a shared address space (others are also possible).

Threads can be implemented in two different ways: kernel and user-space.

The case for user-space threads:

- Kernel threads are expensive: thread context switch involves crossing protection boundary to/from kernel.
- Inflexible: can't easily customize the scheduling policy, since it's in the kernel and there's typically one.

Fast user-level threads packages existed (CThreads, FastThreads, etc.):

- Create one kernel thread for each processor, just use these like an OS would use processors to run the user-level threads.
- Implement user-level threads entirely at the user-level in the runtime system: 1) Any user thread can run on any kernel thread. 2) Very fast, both for thread creation and context switch (no kernel calls in either case), and 3) Synchronization between user threads can be handled entirely at user-level. Can do things like spin-wait on locks.
- Result: much faster thread primitives can support much finer-grained parallelism.

Problem is that the user-level scheduler is oblivious to scheduling decisions being taken in the kernel, and vice versa

Solution: design a protocol for passing scheduling information back and forth between the kernel and the runtime system.

The way the interface works:

- The kernel allocates physical processors to address spaces, can do this any way it sees fit.
- Threads are implemented in user-space: an address space can run arbitrary threads on arbitrary processors allocated to it.
- Kernel *upcalls* the address space on scheduling events: processor allocation / deallocation, thread blocking / unblocking.
- User-space downcalls to the kernel to request change in the number of physical processors $\Rightarrow$ allows processors to be yielded to other address spaces.

Scheduler Activations

- Replaces the kernel notion of a thread.
- Created for each processor assigned to an address space.
- Provides space in the kernel for saving processor context of the currently running user thread when the thread is stopped by the kernel (e.g. for I/O or processor preemption to another application).

Kernel creates a new activation and does an upcall for one of the following reasons:

- New processor available. Runtime picks a user thread to run on it.
- Existing activation blocked (e.g. for I/O or page fault). Runtime picks another user thread to run on the new activation.
- Activation unblocked and is now runnable. New activation includes processor context for two old activations: the newly unblocked one and the one that was preempted in order to make this notification. Why was it necessary to preempt a second activation?
  - To obtain a processor to run on. See fig 1 in paper, where black spots represent processors.
- Activation lost its processor (to another application). Similar to unblocked activation case: new activation contains processor contexts for two old activations: the one whose processor was allocated to another application and the one whose processor is being used to run the new activation.

Runtime informs the kernel when the number of runnable threads = number of allocated processors +- 1.

- Tells the kernel about transitions from needing another processor to not needing another processor and vice-versa.
- Don't need to tell kernel about greater disparities between the two because that won't change the kernel's behavior.
- All other runtime thread operations are strictly user-level.

Result: get the performance of user-level threads with the consistent behavior of kernel threads.

Above this is built a user-level thread scheduler with the same interface as the existing Topaz threads package $\Rightarrow$ mechanism completely transparent to applications.

Some details:

- User-level priority scheduling: may need to pull a lower priority user thread off of another activation. This is done by having the runtime tell the kernel to preempt the processor running the low priority user thread (only the kernel can preempt a processor). The preempted processor is used to do an upcall back to the application.
- Dealing with preempted activations running in critical sections:

  - Runtime checks during an upcall whether the preempted/unblocked user thread was running in a critical section. Continues the user thread out of the critical section if so. Then puts the user thread on the appropriate queue.
  - Critical sections are detected by keeping a hash table of section begin/end addresses that are computed by placing special assembly instructions around critical sections in the object code and then post-processing the object code.

3 key features about this paper:

- Goal is to get user-level threads performance with the scheduling consistency provided by kernel-level threads in a multiprogramming environment.
- The problem to solve: coordinating two independent thread schedulers: the kernel and the application runtime.
- Scheduler activations used as a vessel to transmit information between the two as well as to provide virtual processors for running user-level threads.

Some flaws:

- Authors wave their hands regarding the 5x slower upcall than kernel thread performance.
- Only one application was tested. How would "ordinary" user-level threads perform relative to scheduler activations on other applications? Does the kernel's scheduling policy affect the relative performance in any interesting ways?

Also, while one might think scheduler activations replaces kernel threads, vestiges still remain - for example, the notion of a user-level thread "blocking" in the kernel and creating a new activation.

Psyche

So how is Psyche different from Scheduler Activations? Why?

- Psyche adopts arguably a more radical approach: write a new operating system to push the idea further.
- Also explore further implications of moving threads out of the kernel: what they called *multimodal thread programming*.

- Allow lots of different thread and synchronization models, which can still interoperate.

Implementation:

- shared-memory structures (read-only for kernel $rightarrow$ user-space, read/write for user $rightarrow$ kernel).
- virtual processor abstraction
  - software interrupts for predefined set of kernel events
  - dedicated stack for handling upcalls
  - upcalls also used for inter process communication (PPC)

Major difference:

- Scheduler activations treats the kernel and user-space thrads package as an integrated whole
- Psyche is much more concerned about general purpose interfaces:
  - between the kernel and the user-level thread scheduler
  - between hetergeneous ULS's in different address spaces

Q. Is the inter-ULS interface general enough for synchronisation between applications?

A note on blocking system calls: reading between the lines in the Psyche paper, it's clear that a "blocking system call" is used in a different sense to that in systems like Unix. It refers to anything that make take some time (such as reading a disk block). If that kind of functionality is built into the kernel, it's inevitable that such split-phase operations occur.

Afterword: uniprocessors

Interestingly, this way of implementing threads was not limited to multiprocessor systems. Nemesis adopted a uniprocessor variant of this. Why?

- Remove all blocking from the kernel - even on page faults or interrupts. In fact, remove all threads from the kernel.
- Applications given explicit feedback on both their progress and their CPU application (for adaptive multimedia).

- Policy for multiplexing all resources (in this case, CPU) moved into the application and out of the kernel.

On a uniprocessor, you want something different:

- Don't upcall on deschedules, since you don't have a spare processor.
- Single upcall stack: kernel knows if an upcall is in progress and simply resumes the domain
- The ULS sets pointers to the "context slot" to save the processor state when the domain is descheduled. These slots act as a cache for thread state.
- From personal experience, this so dramatically simplifies the implementation of a threads package that it justifies itself.
- Achilles heel: need to (1) atomically resume a thread context, and (2) clear the "in activation" bit
  - On an Alpha: PAL call costing 1 pipeline drain :-)
  - On an iA32: System call costing the earth :-(

A Lesson

Expose physical resources to applications as much as possible without sacrificing kernel-enforced isolation and protection. Hide the complexity this creates by user-level abstractions rather than kernel-level ones. Avoid the "semantic bottleneck" of the kernel interface.