Scheduler Activations

Jeff Chase
Threads in a Process

• Threads are useful at user-level
  – Parallelism, hide I/O latency, interactivity
• Option A (early Java): user-level library, within a single-threaded process
  – Library does thread context switch
  – Kernel time slices between processes, e.g., on system call I/O
• Option B (Linux, MacOS, Windows): use kernel threads
  – System calls for thread fork, join, exit (and lock, unlock,...)
  – Kernel does context switching
  – Simple, but a lot of transitions between user and kernel mode
• Option C (Windows): scheduler activations
  – Kernel allocates processors to user-level library
  – Thread library implements context switch
  – System call I/O that blocks triggers upcall
• Option D: Asynchronous I/O
Threads and the kernel

- Modern operating systems have multi-threaded processes.
- A program starts with one main thread, but once running it may create more threads.
- Threads may enter the kernel (e.g., syscall).
- Threads are known to the kernel and have separate kernel stacks, so they can block independently in the kernel.
  - Kernel has syscalls to create threads (e.g., Linux \texttt{clone}).
- Implementations vary.
  - This model applies to Linux, MacOS-X, Windows, Android, and pthreads or Java on those systems.
A thread

When a thread is blocked its TCB is placed on a sleep queue of threads waiting for a specific wakeup event.

This slide applies to the process abstraction too, or, more precisely, to the main thread of a process.
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

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Threads are the vehicle for concurrency in many approaches to parallel programming. Threads separate the notion of a sequential execution stream from the other aspects of traditional UNIX-like processes, such as address spaces and I/O descriptors. The objective of this separation is to make the expression and control of parallelism sufficiently cheap that the programmer or compiler can exploit even fine-grained parallelism with acceptable overhead.

Threads can be supported either by the operating system kernel or by user-level library code in the application address space, but neither approach has been fully satisfactory. This paper addresses this dilemma. First, we argue that the performance of kernel threads is inherently worse than that of user-level threads.

1 Introduction

The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within programs. Even a coarse-grained parallel program can exhibit poor performance if the cost of creating and managing parallelism is high. Even a fine-grained program can achieve good performance if the cost of creating and managing parallelism is low.

One way to construct a parallel program is to share memory between a collection of traditional UNIX-like processes, each consisting of a single address space and a single sequential execution stream within that address space. Unfortunately, because such processes were designed for multi-
The case for user-level threads

Inherent costs of kernel threads

- Cost of accessing thread management functions.
  - Kernel trap
  - Copying and checking parameters
  - Costly compared to user level.
- Cost of generality

Thread management is not inherently expensive.

- Can be within order of magnitude of procedure call.
- So overhead added by kernel is significant.
Scheduler Activations

- **Scheduler Activations** is an “interesting” kernel syscall interface. [Anderson et. al., SOSP 1991]
- It’s not an API! Not for application programmers. Not for the faint of heart.
- Suitable for implementing a thread API in a library.
- **What is the right division of function between the kernel and the library?**
  - Kernel manages only resource (processor) allocation and I/O.
  - User-level code (library) manages its resources, etc.
  - Interesting use of *upcalls*.
- Landmark paper for the concept of library OS.
Performance comparison

Comparing user-level threads, kernel threads, and processes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>FastThreads</th>
<th>Topaz Threads</th>
<th>Ultrix Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null fork</td>
<td>34</td>
<td>948</td>
<td>11300</td>
</tr>
<tr>
<td>Signal-wait</td>
<td>37</td>
<td>441</td>
<td>1840</td>
</tr>
</tbody>
</table>

Procedure call takes 7 microseconds.

Kernel trap takes 19 microseconds.

(Maybe kernel trap is not so significant.)
Goals of Scheduler Activations

When kernel intervention not needed, do as well as user-level threads.

But system maintains these invariants:

– No processor idles when a thread is ready.
– No higher-priority thread waits while a lower-priority thread runs.
– During any thread blocking, e.g., blocking syscall or page fault, other threads can run.
Sources of poor integration

Thesis: Kernel threads are the wrong abstraction for user-level threads.

- Kernel threads block, resume, and are preempted without notification to the user level.
- Kernel threads are scheduled obliviously w.r.t. the user-level state.

When user thread blocks, k-thread also blocks.

- If only one kernel thread \( \rightarrow \) stuck.
- If more kernel threads than cores, must time-slice.
  - Kernel thread could be preempted while in a critical section, increasing overhead (e.g., w/ spinlocks).
  - Priority issues
Effective kernel support for user-level threads

Each process supplied with a virtual multiprocessor.
- Kernel allocates processors to address spaces.
- User level threads system (ULTS) has complete control over scheduling.
- Kernel notifies ULTS whenever it changes the number of processors, or a user thread blocks or unblocks.
- ULTS notifies kernel when application needs more or fewer processors.
Approach

- Each app has complete knowledge of and control over the processors in its virtual MP.
- Kernel notifies the AS thread scheduler of all kernel events.
- Use upcalls to deliver notification.
- Upcall also delivers a processor to run the ULTS scheduler code: a **scheduler activation**.
Scheduler Activations

A scheduler activation is an allocated processor (core), delivered to the ULTS via upcall.

Three roles

- As a vessel, or execution context, for running user-level threads, like a kernel thread.
- As a notification to the ULTS of a kernel event.
- As a data structure for saving state, etc.

Two execution stacks

Note: UL threads are never directly resumed by the kernel.
Upcalls

Add this processor (processor #)
   - Execute a runnable user-level thread.

Processor has been preempted (preempted activation # and its machine state)
   - Return to the ready list the user-level thread that was executing in the context of the preempted scheduler activation.

Scheduler activation has blocked (blocked activation #)
   - The blocked scheduler activation is no longer using its processor.

Scheduler activation has unblocked (unblocked activation # and its machine state)
   - Return to the ready list the user-level thread that was executing in the context of the blocked scheduler activation.
At time T1, the kernel allocates the application two processors. On each processor, the kernel upcalls to user-level code that removes a thread from the ready list and starts running it.
At time T2, one of the user-level threads (thread 1) blocks in the kernel. To notify the user level of this event, the kernel takes the processor that had been running thread 1 and performs an upcall in the context of a fresh scheduler activation. The user-level thread scheduler can then use the processor to take another thread off the ready list and start running it.
At time T3, the I/O completes. Again, the kernel must notify the user-level thread system of the event, but this notification requires a processor. The kernel preempts one of the processors running in the address space and uses it to do the upcall. (If there are no processors assigned to the address space when the I/O completes, the upcall must wait until the kernel allocates one). This upcall notifies the user level of two things: the I/O completion and the preemption. The upcall invokes code in the user-level thread system that (1) puts the thread that had been blocked on the ready list and (2) puts the thread that was preempted on the ready list. At this point, scheduler activations A and B can be discarded.
Finally, at time T4, the upcall takes a thread off the ready list and starts running it.
Complications

Additional preemption required for correct priorities.

Preemption or upcall in thread manager.
- Thread manager needs to be reentrant.
- If preempted while idling, nothing needs to be done.
- Otherwise, need to do a UL CS to finish what it was doing.
- Also, if page fault in thread manager, need to handle this specially.
AS to Kernel Calls

Only needed whenever the relationship between number of processors and number of runnable threads changes.

Add more processors (additional # of processors needed)
  – Allocate more processors to this address space and start them running scheduler activations.

This processor is idle()
  – Preempt this processor if another address space needs it.
Critical Sections

When a upcall informs ULTS that a thread has been preempted, we do a UL CS to that thread to finish any critical sections.
Compute-Bound

No spin-waiting for Topaz so lots of overhead.

New FastThreads better for less than 6 processors due to poor scheduling of Topaz. Daemon thread causes preemption, even if processors avail.
I/O Bound

FastThreads degrades due to inability to reallocate processor when thread blocks. Topaz parallels scheduler activations, but poorer performance due to kernel involvement.
Multiprogramming

Ran two copies, ideal speedup is 3.
  – Topaz: 1.29
  – Original FastThreads: 1.26
  – Scheduler Activations: 2.45

Topaz:
  – Thread ops are more expensive.

FastThreads:
  – Time slicing causes waiting.