Scheduler Activations

CS380L: Mike Dahlin
September 24, 2003

1 Preliminaries
1.1 Review
1.2 Outline
• Scheduler activations
1.3 Preview

2 Principle: Expose revocation
• Exokernel theme: what is minimal abstraction needed for high performance implementations (e.g., to expose resource to application but let application control resource scheduling if it wants to do so).
• “Traditional” abstractions do a little too much
  • Networks: AM says “must be async, no buffering” (→ add synchronization and buffering at user level if you need it)
  • Network security?
  • File system metadata
  • Shared kernel cache buffer
  • Paging
  • Today: the traditional concurrency abstraction (threads) is too much
    • QUESTION what is the traditional concurrency abstraction?
    • SA approach
      • don’t provide illusion of infinite processors (anti-THE!)
      • don’t keep ready list or choose threads to run in kernel.

3 Scheduler activations
3.1 Basics
Traditional threads:

• Kernel (1) gives “activations” to user level and (2) informs user-level when “activations” are revoked (and provides the saved processor state).
• But user-level decides which threads run on which activations

- Schedule
- Running
- Ready
- De-schedule
- Unlock/signal
- Lock/wait
- Blocked

• But people tend to adopt 2-level model: N user-level threads on M kernel threads
• Why?
• Problems with kernel-thread-only approach
• Problems with user-thread-only approach
• But, problem with combined approach:
How many kthreads?
Lose control of user-level scheduling (what if thread holding UL lock is preempted or high-priority UL thread is preempted?)

+ Blumofe result

SA:

Basic interface:
- add_processor() – an idle processor is now available
- has_blocked() – an idle processor is now available
- has_unblocked(stateA, [stateB]) – thread A unblocked (put it on ready queue); BTW to tell you this I also blocked B (put it on ready queue too, and schedule someone to run with this activation)

- has BEEN preempted(stateA, [stateB]) – thread A got preempted (put it on ready queue); BTW to tell you this I also blocked B (put it on ready queue too, and schedule someone to run with this activation)

Notice: all schedule activation calls by kernel to user-level handlers provide an activation (that is, a scheduled thread). In order to keep number of threads allocated to a process from growing without bound, processor typically suspends a running activation in order to tell you “hasBeenPreempted” or “hasUnblocked”

3.2 Bells and whistles
- Advisory interface to tell kernel how many threads a process can profitably run with
  - Simple interface (could imagine a more complex one...is it worth it?)
  - “Want one more” (I have more ready threads than activations)
  - “Give one back” (I have fewer ready threads than activations)
- In exokernel: before revoking, kernel warns process “I’m about to take a processor away...perhaps you should give one of your choice up voluntarily instead”
  - Would that be useful here?

3.3 Details
Interaction of user-level critical section and kernel suspensions
- Problem: What if thread is suspended while holding lock on user-level scheduler?
  - Possible deadlock: activation call tries to grab lock to move suspended thread to ready list
- Example of general problem we’ve seen several times:
  - Mesa device drivers: kernel monitors v. hardware
  - Active messages: enqueue incoming message w/o grabbing a lock (handler cannot block)
  - This really is same problem: “interrupt handlers cannot block”
- Proposed approaches
Prevent: if thread is holding THE lock (not all locks), tell kernel to let it keep running

- More generally: tell kernel the priority of the running thread?
- DA: overhead to tell kernel this whenever lock is grabbed
- DA: need to “pin” pages
  - (Moral equivalent to AM: run with interrupts turned off...)

Recover: Set flag when grabbing “THE” lock; activation handler checks flag, if flag set, set YIELD flag; run current thread (without touching normal scheduling data structures); when flag releases THE lock, check YIELD flag and yield() if set

- DA: slows down common case for rare case
- Cute fix: 2 versions of code
- Other alternative: wait-free synchronization (e.g., Blumofe Cilk)

4 User-level threads v. kernel-level threads v. events

5 Subsequent systems

- Exokernel
  - How differ from Anderson?

- K42

- Solaris


- Solaris: user-level threads and “LWPs” (kernel-level threads)
- Limits of “old” Solaris (pre 2.6 - no scheduler activations)
  - No correlation between priority of user thread and priority of underlying LWP
  - User-level threads prone to priority inversion (fixed in Solaris for kernel threads, but not for user threads)

- No inheritance (e.g., inherit priority of parent) at user threads level
- Difficult to implement adaptive locks (b/c kernel state not available to threads library)
- Keeping sufficient pool of kernel threads s.t. runnable threads can run was not easily solved
  - workaround in 2.6: “SIGWAITING” signalled when last runnable kernel thread blocks; handler can create new kernel thread
- Kernel threads only block on condition variables. The kernel CV wait() code calls schedctl_check(SC_BLOCK) ("are all kernel threads for this process blocked?")
- If all kernel threads are now blocked
  - currentThread.khandoff = Get a new kernel thread from the process’s pool of inactive threads ()
  - After returning from schedctl_check(), the thread calls switch() which notices that khandoff is non-null, so it passes control to the specified kernel thread
  - The specified kernel thread wakes up (in user mode), calls the scheduler library, which hands it a user-level thread to run.

- Questions:
  - Why do they only kick off a new activation/kernel thread if the active number reaches 0? Would the implementation have to change if they did something more general (like in original scheduler activations – call any time a user-thread blocks?)
  - How else do they differ? What are pros and cons?

6 Project idea

Project 1: Perfect threads

I think you can build scheduler activations without modifying the kernel. (Idea is: use /proc). (Possible exception is: multiprogramming descheduling, but this should not be common for demanding apps. Actually, can still do it at user level, but probably need to be root...)
frames). Use scheduler activations to keep 1 kernel thread per processor active. →
programming model is equivalent to one-kernel-thread-per-request. Space over-
head is similar to events. Other overheads similar to having the minimum number
of kernel threads needed.

<table>
<thead>
<tr>
<th></th>
<th>kernel threads</th>
<th>user threads</th>
<th>evs</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem overhead</td>
<td>lpage</td>
<td>lpage</td>
<td>NA</td>
</tr>
<tr>
<td>ctx-switch overhead</td>
<td>ctxsw+copy-all-reg</td>
<td>copy-all-reg</td>
<td>ct</td>
</tr>
<tr>
<td>wait/signal/lock</td>
<td>ctx-sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>programming convenience</td>
<td>yes</td>
<td>yes</td>
<td>nc</td>
</tr>
<tr>
<td>tolerate blocking</td>
<td>yes</td>
<td>no</td>
<td>nc</td>
</tr>
</tbody>
</table>

SA makes user-threads have the good k-thread properties
heap allocates frames make
have the go

Project 2: Add scheduler activations to vin et al’s hierarchical CPU scheduler
(e.g., in QLinux)?

7 Admin

- Exam – 10/21
- Project checkpoint 10/30 – briefly list status and plan (e.g., orig 4 milestones.
  what has changed. New schedule)
- Lecture series –