Scheduler Activations

Two comparisons:
  1) threads vs. process
  2) kernel-level vs. user-level threads

1) kernel-level threads vs. process
  * what is a process?
    1) set of instructions
    2) some state (memory, file descriptors, signal handlers, etc.)
    * In Linux: clone(SIGCHLD, 0);
  * what is a thread?
    * really just a process that shares state with another process
      * shares: memory, file descriptors, signal handlers
    * In Linux: clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

2) kernel-level vs. user-level threads
  * pretty simple:
    * kernel threads are implemented by OS
      - use system calls to create, manage, destroy, etc.
    * user-level threads are implemented by some user code
      - just function calls to create, maange, destroy, etc.
      - paper mentions that the compiler can help here with register allocation.
        how is that? A: we don't have to save every register, just the ones
        actually being used, if the compiler tells us which they are.
  * key difference:
    - OS can make _fully informed decisions_ when scheduling kernel-level
      threads since it has a global view of processor resources.
    - The user-level threading library obviously cannot. At least not without
      scheduler activations!

3) What can go wrong when user-level threads on top of kernel-level threads?
  1. Recall the lock problem.
    a. setup:
      - processors: p1, p2
      - user-level threads: u1, u2
      - kernel-level threads: t1, t2, t3 (some other proc)
      - u1 and u2 share lock at some point
    b. user-level schedules u1 on t1, u2 on t2
    c. kernel schedules t1, t3
    d. u1 takes lock
    e. kernel deschedules t1, schedule t2
    f. u2 tries to take lock; can't. waits and waits.
    - problem: kernel has no clue about critical sections.

1:1 systems vs. N:1, vs. M:N systems
  * 1:1 uses one kernel thread per thread
  * N:1 puts N user threads on one thread, and uses just one thread
    - example: event-like things: Node.JS
  * M:N puts M user threads on N kernel threads
    - common user-level threads, go-routines

Scheduler Activations
Effective Kernel Support for the User-Level Management of Parallelism
Thomas Anderson, et. al
SOSP 1991

Grand Idea
  - Kernel processes are:
    - great because the kernel knows about them
        - so can do a relatively good job of scheduling them.
    - not great because they tend to be slow (i.e, context switching)

```
  - Kernel-level threads are:
    - good for the same reasons as kernel processes
    - still pretty slow (10x slower than user-level threads, according to paper)
  - User-level threads are:
    - great because they tend be fast (i.e., context switch via function call)
    - not so great because the kernel has no idea they exist
      - many user-level threads on one process
      - kernel may reschedule an important user-level thread, etc.
  - So, give user-level threads an OS API to make them better
    - somehow have the kernel know about what's going on with threads
    - somehow have the user-level thread tell the kernel what's going on
    - the mechanism to do both is called a "scheduler activation"

Effective Kernel Support for User-Level Management of Parallelism (3)
  - kernel provides user-level library with its own virtual multiprocessor
  - here are the key aspects to this abstraction:
    - kernel may change number of processors in that multiprocessor
    - user-level library controls which threads to run on those processors
    - kernel notifies user-level library when:
      - an event it occurs: the user-level thread does the actual handling
      1) it changes number of processors
      2) user-level thread blocks or wakes up in the kernel
    - U-L library notifies kernel when it wants more/needs fewer processors
      - U-L library only notifies kernel of things that affect processor alloc.

Kernel->User Communication (3.1)
  - mechanism for this U-L to K-L communication is called "scheduler activation"
    - name chosen because K-L events activate U-L library's scheduler
    - scheduler activations serve three roles: they
      1) are the execution context for running user-level threads
        - just like a kernel-level thread: many U-L threads per activation
          - it seems.
      2) notify the user-level thread system of a kernel event
      3) provides space in kernel for saving U-L thread context when blocking
  - scheduler activations look pretty similar to traditional kernel threads
    - contains two execution stacks: one for kernel, one for app
    - U-L thread scheduler runs on the activation's user-level stack
    - each U-L thread is allocated its own stack when it starts
  - when a program starts:
    - kernel creates a scheduler activation, assigns it to processor
    - then, upcalls into a fixed entry point
    - U-L library initializes intself and runs the main app. thread
  - main thread may request more processors
    - kernel creates an additional scheduler activation for each processor
    - upcalls into the U-L to tell it that new processor is ready
  - when kernel needs to notify U-L of events
    - create new scheduler activation, assigns it to processor
    - jumps (upcalls_ into some entry point
    - then, app can do whatever it wants, just like if it was in a k-thread
  - crucial distinction is:
    - kernel never resumes stopped (i.e, because of blocking) U-L threads
    - instead, new scheduler activation is created
      - notifies U-L of stopped U-L thread
    - U-L decides what to do by:
      - saving state and "removing" old thread for old activation
      - tells kernel old activation can be reused
      - decies which thread to run on the processor
  - invariant: # RUNNING scheduler activations == # virtual processors
    - when new processor is added, new scheduler activation
    - because when U-L blocks, launch new scheduler activation
  - the following the scheduler activation upcall points:
    1) new processor added
    2) processor has been preempted
    3) scheduler activation has blocked
    3) scheduler activation has unblocked
```

- usually occur in combinations, so only one scheduler activation is created
- Example: a user-level thread blocks in the kernel
  1) kernel uses a fresh scheduler activation to notify U-L of event
    - so this means you have a runnable schedular activation (this one!)
    - U-L can run other U-L threads on this activation
  2) when U-L thread unblocks, kernel uses fresh activation to nofify U-L
    - remember: invariant of 1 scheduler activation per "processor"
    - if U-L has no "processors" (how?) kernel allocates new processor
      - notifies U-L of new processor AND of resumable blocked thread in one
    - if U-L has "processors", may have to preempt a "processor" to do upcall
      - first notifies of resumable blocked thread
      - then of preempted thread (which was on the preempted "processor")
      - U-L can decide which of the two (preempted or resumable) to execute
    - to resume a U-L thread, need that thread's state
      - most of the state is already in U-L: stack, control block
      - registers are saved by the kernel on blocking call
      - kernel passes registers to U-L when notifying of I/O completion
- Example: kernel needs to take a processor away from U-L A and give to B
  1) interrupts processor in A, stops existing activation
    - kernel doesn't need permission to steal processor
  2) "moves" processor to B, does upcall on this process in B w/new activation
  3) notifies A on existing some existing processor on a
    - starts new activation by preempting whatever's on that processor
    - notifies about two prremptions:
      1) moving processor away
      2) the old activation on this processor
  4) U-L A decides what to with the two preempted U-L threads
  - what happens when last processor is moved away?
    - delay notification until kernel eventually re-allocates it a processor
- U-L library might want to maintain priorities of U-L threads
  - so, when a higher priority thread gets preempted, U-L can
    - somehow ask kernel to preempt the processor with the lower priority
    - can do this because U-L knows where threads are
- scheduler activations can be used to implement things other than U-L threads
  - kernel doesn't need to know about the data structures used to represent parallelism at the user-level
- scheduler activations work even when there are no U-L threads
  - IE, when only the U-L thread manager is running
  - simple creates a new scheduler activation
  - reentrant U-L thread managers can then do what they need in there
- if a U-L thread needs to do work in the kernel after some I/O unblocks
  - kernel does the right thing and does the work, THEN upcalls U-L

User->Kernel Communication (3.2)

- the U-L thread system need not tell the kernel about every operation
  - key observation
- U-L system notifies the kernel when:
  1) it has more runnable threads than "processors"
    "add more processors (#)"
  2) it has more "processors" than runnable threads
    "this processor is idle"
- Two cases to look at:
  a) U-L system has more runnable threads than "processors"
    - if kernel doesn't assign new processors, then they MUST be busy
  b) U-L system has more "processors" than runnable threads
    - if kernel doesn't remove unused processors, then system must be idle
    - so, U-L can keep adding parallelism without noifying kernel
- these notifications are only hints
  - requesting a "processor" doesn't guarantee you a processor NOW
- requests are serialized
  - U-L requests processor
  - ... time passes ...
  - U-L finally gets processor, but doesn't need it
  - U-L must say it doesn't need it
    - Uh...so you have to trust them?

- Yeah, but not unique to this system.
- Yeah, kernel doesn't actually know about what's running.
- Can incentivize honest guys.

Dealing with preemption during critical sections (3.3)

- don't want bad stuff to happen when a U-L thread gets preempted
  - like scheduling a U-L thread that's just going to sit there and wait
  - or worse, dead-lock:
    - preempted thread holds lock on user-level thread ready list
    - newly scheduled thread marks prempted thread a RUNNABLE and tries to put preempted thread into list (this sounds like bad programming)
- two options:
  1) let the kernel know that you're entering a critical section
     - slow.
  2) have a way to recover when this happens
     - this is the approach they take
- U-L system wil receive an upcall when a thread is prempted
  - U-L can check if the preempted thread was in a critical section
    - if so, U-L system continues that thread until it's out of the criical section
  - once critical section is done, U-L can safely place preempted thread on ready list
  - Q: fun question: what if that U-L gets preempted while waiting?
    - then just do the same thing in the activation: stack them up

Implementation

- + 1200 lines to kernel (Topaz), + ~250 to user-level library (FastThreads)
  - most code was concerned with implementing processor allocation policy
- Processor allocation policy tries to be fair:
  - tries to never waste processors
  - distributes free processors evenly across spaces that need some
- Their implementation also has regular kernel threads
  - internel kernel implementation uses scheduler activations
    - so they actually do internal "downcalls", but, you know, just f-calls

Performance Enhancements (4.3)

- So, you want this to fast.
- most performance consideration pertain to dealing with critical sections
- somehow, U-L systems needs to know that a U-L thread is in a critical sect.
  - one way is to set a flag when it enters and clear when it leaves
    - apparently, this is not fast enough
      - adds ~10us, which is ~20% overhead (from 5.1)
    - ideally, don't want to do any work unless preemption occurs
- instead, use the compiler (lol)
  - mark each crticial section in code
    - so know you know the start and end PC of the crticial section
    - what about function calls in there?
      - they use a flag in this case
      - hopefully is rare. try to inline? make them small?
  - make copy each critical section
    - add a few lines to end to yield processor back to resumer
      - IE, the U-L scheduler thing
  - when U-L system gets preemption upcall
    - check PC to see if in crticial section or if the flag was set
    - if in critical section, jump to the copied code
    - copied code will jump back to the resumer
      - how? probably just does a 'ret', so need to do 'call' to copy
- scheduler activations are kinda expensive to create
  - so cache their structures for reuse (like a slab allocator, sounds like)
- discarded scheduler activations can be collected and returned in bulk
  - so batching