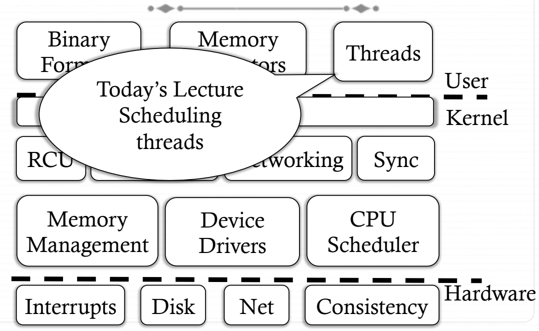
Native POSIX Thread Library (NPTL)

CSE 506
Don Porter

Logical Diagram



Today's reading

- Design challenges and trade-offs in a threading library
- Nice practical tricks and system details
- And some historical perspective on Linux evolution

Threading review

- What is threading?
  - Multiple threads of execution in one address space
  - x86 hardware:
    - One cr3 register and set of page tables shared by 2+ different register contexts otherwise (rip, rsp/stack, etc.)
  - Linux:
    - One mm_struct shared by several task_structs
  - Does JOS support threading?

Ok, but what is a thread library?

- Kernel provides basic functionality: e.g., create a new task with a shared address space, set my gs register
- In Linux, libpthread provides several abstractions for programmer convenience. Examples?
  - Thread management (join, cleanup, etc)
  - Synchronization (mutex, condition variables, etc)
  - Thread-local storage
- Part of the design is a division of labor between kernel and libraries!

User vs. Kernel Threading

- Kernel threading: Every application-level thread is implemented by a kernel-visible thread (task struct)
  - Called 1:1 in the paper
- User threading: Multiple application-level threads (m) multiplexed on n kernel-visible threads (m >= n)
  - Called m:n in the paper
  - Insight: Context switching involves saving/restoring registers (including stack).
    - This can be done in user space too!

Intuition

- 2 user threads on 1 kernel thread; start with explicit yield
  - 2 stacks
  - On each yield():
    - Save registers, switch stacks just like kernel does
- OS schedules the one kernel thread
  - Programmer controls how much time for each user thread

Extensions

- Can map m user threads onto n kernel threads (m >= n)
  - Bookkeeping gets much more complicated (synchronization)
- Can do crude preemption using:
  - Certain functions (locks)
  - Timer signals from OS

Why bother?

- Context switching overheads
- Finer-grained scheduling control
- Blocking I/O

Context Switching Overheads

- Recall: Forking a thread halves your time slice
  - Takes a few hundred cycles to get in/out of kernel
    - Plus cost of switching a thread
  - Time in the scheduler counts against your timeslice
- 2 threads, 1 CPU
  - If I can run the context switching code locally (avoiding trap overheads, etc), my threads get to run slightly longer!
  - Stack switching code works in userspace with few changes

Finer-Grained Scheduling Control

- Example: Thread 1 has a lock, Thread 2 waiting for lock
  - Thread 1's quantum expired
  - Thread 2 just spinning until its quantum expires
  - Wouldn't it be nice to donate Thread 2's quantum to Thread 1?
    - Both threads will make faster progress!
- Similar problems with producer/consumer, barriers, etc.
- Deeper problem: Application's data flow and synchronization patterns hard for kernel to infer

Blocking I/O

- I have 2 threads, they each get half of the application's quantum
  - If A blocks on I/O and B is using the CPU
  - B gets half the CPU time
  - A's quantum is "lost" (at least in some schedulers)
- Modern Linux scheduler:
  - A gets a priority boost
  - Maybe application cares more about B's CPU time…

Blocking I/O and Events

- Events are an abstraction for dealing with blocking I/O
- Layered over a user-level scheduler
- Lots of literature on this topic if you are interested…

Scheduler Activations

- Observations:
  - Kernel context switching substantially more expensive than user context switching
  - Kernel can't infer application goals as well as programmer
    - nice() helps, but clumsy
- Thesis: Highly tuned multithreading should be done in the application
  - Better kernel interfaces needed

What is a scheduler activation?

- Like a kernel thread: a kernel stack and a user-mode stack
  - Represents the allocation of a CPU time slice
- Not like a kernel thread:
  - Does not automatically resume a user thread
  - Goes to one of a few well-defined "upcalls"
    - New timeslice, Timeslice expired, Blocked SA, Unblocked SA
    - Upcalls must be reentrant (called on many CPUs at same time)
  - User scheduler decides what to run

User-level threading

- Independent of SA's, user scheduler creates:
  - Analog of task struct for each thread
    - Stores register state when preempted
  - Stack for each thread
  - Some sort of run queue
    - Simple list in the (optional) paper
    - Application free to use O(1), CFS, round-robin, etc.
- User scheduler keeps kernel notified of how many runnable tasks it has (via system call)

Downsides of scheduler activations

- A random user thread gets preempted on every scheduling-related event
  - Not free!
  - User scheduling must do better than kernel by a big enough margin to offset these overheads
- Moreover, the most important thread may be the one to get preempted, slowing down critical path
  - Potential optimization: communicate to kernel a preference for which activation gets preempted to notify of an event

Back to NPTL

- Ultimately, a 1:1 model was adopted by Linux.
- Why?
  - Higher context switching overhead (lots of register copying and upcalls)
  - Difference of opinion between research and kernel communities about how inefficient kernel-level schedulers are. (claims about O(1) scheduling)
  - Way more complicated to maintain the code for m:n model. Much to be said for encapsulating kernel from thread library!

Meta-observation

- Much of 90s OS research focused on giving programmers more control over performance
  - E.g., microkernels, extensible OSes, etc.
- Argument: clumsy heuristics or awkward abstractions are keeping me from getting full performance of my hardware
- Some won the day, some didn't
  - High-performance databases generally get direct control over disk(s) rather than go through the file system

User-threading in practice

- Has come in and out of vogue
  - Correlated with how efficiently the OS creates and context switches threads
- Linux 2.4 – Threading was really slow
  - User-level thread packages were hot
- Linux 2.6 – Substantial effort went into tuning threads
  - E.g., Most JVMs abandoned user-threads

Other issues to cover

- Signaling
  - Correctness
  - Performance (Synchronization)
- Manager thread
- List of all threads
- Other miscellaneous optimizations

Brief digression: Signals

- Signals are like a user-level interrupt
  - Specify a signal handler (trap handler), different numbers have different meanings
  - Default actions for different signals (kill the process, ignore, etc).
- Delivered when returning from the kernel
  - E.g., after returning from a system call
- Can be sent by hand using the **`kill`** command
  - **`kill -HUP 10293 # send SIGHUP to proc. 10293`**

Signal masking

- Like interrupts, signals can be masked
  - See the **`sigprocmask`** system call on Linux
- Why?
  - User code may need to synchronize access to a data structure shared with a signal handler
  - Or multiple signal handlers may need to synchronize
  - See optional reading on signal races for an example

What was all the fuss about signals?

- 2 issues:
  1) The behavior of sending a signal to a multi-threaded process was not correct. And could never be implemented correctly with kernel-level tools (pre 2.6)
     - Correctness: Cannot implement POSIX standard
  2) Signals were also used to implement blocking synchronization. E.g., releasing a mutex meant sending a signal to the next blocked task to wake it up.
     - Performance: Ridiculously complicated and inefficient

Issue 1: Signal correctness w/ threads

- Mostly solved by kernel assigning same PID to each thread
  - 2.4 assigned different PID to each thread
  - Different TID to distinguish them
- Problem with different PID?
  - POSIX says I should be able to send a signal to a multi-threaded program and any unmasked thread will get the signal, *even if the first thread has exited*
- To deliver a signal kernel has to search each task in the process for an unmasked thread

Issue 2: Performance

- Solved by adoption of futexes
- Essentially just a shared wait queue in the kernel
- Idea:
  - Use an atomic instruction in user space to implement fast path for a lock (more in later lectures)
  - If task needs to block, ask the kernel to put you on a given futex wait queue
  - Task that releases the lock wakes up next task on the futex wait queue
- See optional reading on futexes for more details

Manager Thread

- A lot of coordination (using signals) had to go through a manager thread
  - E.g., cleaning up stacks of dead threads
  - Scalability bottleneck
- Mostly eliminated with tweaks to kernel that facilitate decentralization:
  - The kernel handled several termination edge cases for threads
  - Kernel would write to a given memory location to allow lazy cleanup of per-thread data

List of all threads

- A pain to maintain
- Mostly eliminated, but still needed to eliminate some leaks in fork
- Generation counter is a useful trick for lazy deletion
  - Used in many systems
  - Idea: Transparently replace key "Foo" with "Foo:0". Upon deletion, require next creation to rename "Foo" to "Foo:1". Eliminates accidental use of stale data.

Other misc. optimizations

- On super-computers, were hitting the 8k limit on segment descriptors
- Where does the 8k limit come from?
  - Bits in the segment descriptor. Hardware-level limit
- How solved?
  - Essentially, kernel scheduler swaps them out if needed
  - Is this the common case?
  - No, expect 8k to be enough

Optimizations

- Optimized exit performance for 100k threads from 15 minutes to 2 seconds!
- PID space increased to 2 billion threads
  - /proc file system able to handle more than 64k processes

Results

- Big speedups! Yay!

Summary

- Nice paper on the practical concerns and trade-offs in building a threading library
  - I enjoyed this reading very much
- Understand 1:1 vs. m:n model
  - User vs. kernel-level threading
- Understand other key implementation issues discussed in the paper