Threading

Nima Honarmand
(Based on slides by Don Porter and Mike Ferdman)
Threading Review

• Multiple threads of execution in one address space
  – Why?
    • Exploits multiple processors
    • Separate execution stream from address spaces, I/O descriptors, etc.
    • Improve responsiveness of UI (and similar applications)

• x86 hardware:
  – One CR3 register and set of page tables
    • Shared by 2+ different contexts (each has RIP, RSP, etc.)

• Linux:
  – One mm_struct shared by several task_structs
Threading Libraries

• Kernel provides basic functionality
  – e.g.: create new thread

• Threading library (e.g., libpthread) provides nice API
  – Thread management (join, cleanup, etc.)
  – Synchronization (mutex, condition variables, etc.)
  – Thread-local storage

• Part of design is division of labor
  – Between kernel and library
User vs. Kernel Threading

• Kernel threading
  – Every application-level thread is kernel-visible
    • Has its own `task_struct`
  – Called **1:1**

• User threading
  – Multiple application-level threads (**m**)  
    • multiplexed on **n** kernel-visible threads (**m** \(>= n\))
  – Context switching can be done in user space
    • Just a matter of saving/restoring all registers (including RSP!)
  – Called **m:n**
    • Special case: **m:1** (no kernel support)
User Threading Implementation

• User scheduler creates:
  – Analog of `task_struct` for each thread
    • Stores register state when switching
  – 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.
Tradeoffs of Threading Approaches

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

- Takes a few hundred cycles to get in/out of kernel
  - Plus cost of saving/restoring registers
  - Time in the scheduler counts against your timeslice

- Forking a thread halves your time slice
  - At least in some schedulers

- 2 threads, 1 CPU
  - Run the context switch code locally
    - Avoiding trap overheads, etc.
    - Get more time from the kernel
Finer-Grained Scheduling Control

• Thread 1 has lock, Thread 2 waiting for lock
  – Thread 1’s quantum expired
  – Thread 2 spinning until its quantum expires
  – Can donate Thread 2’s quantum to Thread 1?
    • Both threads will make faster progress!

• Many examples (producer/consumer, barriers, etc.)

• Deeper problem:
  – Application’s data and synchronization unknown to kernel
    • Kernel makes blind decisions
Blocking I/O

• I/O requires going to the kernel

• When one user thread does I/O
  – All other user threads in same kernel thread wait
  – Solvable with async I/O
    • Much more complicated to program
User Threading Complexity

• Lots of libc/libpthread changes
  – Working around “unfriendly” kernel API

• Bookkeeping gets much more complicated
  – Second scheduler
  – Synchronization different

• Can do crude preemption using:
  – Certain functions (locks)
  – Timer signals from OS
Scheduler Activations

• **Reading assignment for next week**

• **Observations:**
  – Kernel ctxt switch more expensive than user ctxt switch
  – Kernel can’t infer application goals as well as programmer
    • nice() helps, but clumsy

• **Highly tuned multithreading should be done in app**
  – Better kernel interfaces needed
Scheduler Activations

• Better API for user-level threading
  – Not available on Linux

• On any blocking operation, kernel *upcalls* back to user scheduler
  – Eliminates most libc changes
  – Easier notification of blocking events

• User scheduler keeps kernel notified of how many runnable tasks it has (via system call)
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 efficiency of OS thread create and switch

• Linux 2.4 – Threading was slow
  – User-level thread packages were hot (e.g., LinuxThreads)
    • Code is really complicated
      • Hard to maintain
      • Hard to tune

• Linux 2.6 – Substantial effort into tuning kernel threads
  – Native POSIX Thread Library (NPTL)
  – Most JVMs abandoned user threads
    • Tolerable performance at low complexity
Other Problems Solved by NPTL

• Signaling
  – Correctness
  – Performance (Synchronization)

• Read the NPTL paper for more
  – Manager thread
  – List of all threads
  – etc.
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

- 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*
Issue 2: Performance

• Solved by adoption of **futex**
  – Essentially 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