CS 5204 Operating Systems Lecture 5
Godmar Back

Announcements
• Project/survey paper handout on Wednesday

Motivation
• Q: Is Lauer/Needham relevant to current systems?
• Which model should we pick for which application – both are available on current systems
• Must understand implementation trade-offs on contemporary systems
  – In addition to programming model trade-offs

Implementing Threads
• Issues:
  – Who maintains thread state/stack space?
  – How are threads mapped onto CPUs?
  – How is coordination/synchronization implemented?
  – How do threads interact with I/O?
  – How do threads interact with existing APIs such as signals?
  – How do threads interact with language runtimes (e.g., GCs)?
  – How do terminate threads safely?

Managing Stack Space
• Stacks require continuous virtual address space
  – virtual address space fragmentation (example 32-bit Linux)
• What size should stack have?
  – How to detect stack overflow?
  – Ignore vs. software vs. hardware
• Related: how to implement
  – Get local thread id “pthread_self()”
  – Thread-local Storage (TLS)

Nonpreemptive Threads
• Aka Coroutines
  – CPU switches at well-defined points (“yield”, or synchronization points: “lock”, “wait”)
  – Low context-switch cost - similar to procedure call
• Advantages
  – Can make integrating garbage collection easier
  – Can allow for very fine-grained resource control (Capriccio)
  – Can be implemented w/o kernel support
Nonpreemptive Threads (cont’d)
• Disadvantages:
  – Can increase latency
  – Hard to extend to multiprocessor machines
  – Makes termination of uncooperative threads hard (why?)
• Note: using nonpreemptive threads does not negate need for locks – why?

Preemptive Threads
• CPU can switch at any time
  – Higher context switch cost: more state to save
• Advantages
  – Allows for quasi-parallelism (latency benefits)
• Disadvantages
  – Requires kernel support
  – Can make scheduling & GC control harder

Example: x86
• Nonpreemptive = C calling conventions:
  – Caller-saved: eax, ecx, edx + floating point
  – Callee-saved: ebx, esi, edi, esp
    • ebp, eip for a jmpbuf size of 6*4 = 24 bytes
• Preemptive = save entire state
  – All registers + 108 bytes for floating point context
• Note: context switch cost = save/restore state cost + scheduling overhead + lost locality cost

On Termination
• If you terminate a thread, how will you clean up if you have to terminate it?
• Strategies:
  – Avoid shared state where possible
  – Disable termination
  – Use cleanup handlers
    tryfinally,
    pthread_cleanup

User-level Threads (aka 1:N)
• Kernel sees one thread per process
• Scheduling + synchronization done in user-space
  – Potentially fast context switches
  – fast locks (if nonpreemptive!)
• Threads are lightweight

User-level Threads (cont’d)
• Drawbacks:
  – I/O blocks entire process
  – Often nonpreemptive (although can be advantage)
  – Both of these can be remedied with signals
    • Virtual timers for preemption
    • Asynchronous I/O signals
    • This is expensive and often fragile
  – Not multiprocessor capable
Kernel-level Threads (aka 1:1)

- Kernel manages threads
- I/O blocks only current thread
- Context switch requires kernel trap
  - Synchronization may require kernel traps as well
- OS timer interrupts provides preemption, kernel scheduler schedules threads
  - Allows for use of SMPs

M:N Model

- Implemented in Solaris
  - Implementation for Linux in NGTL
- Idea:
  - Create small number M of LWPs ("light-weight processes") onto which (larger number) N of user-level threads are being scheduled
  - Only create LWP if all LWPs are currently blocked; time out unused LWPs

Lightweight Processes (M:N)

Source: Multithreading in the Solaris Operating Environment, Sun 2002

M:N Model

- Solaris discarded LWPs
  - Linux never introduced them (NPTL won over NGTL)
- Why:
  - Automatic concurrency control hard
  - How many LWPs should be allocated?
  - Experience showed limited gain from faster context switches in user mode
  - _schedlock contention
  - Signal implementation difficult
    - Needed manager thread for asynchronous signals

Drawbacks of M:N model

Linux NPTL

- Recent 1:1 model
- Enabled by kernel changes:
  - Introduction of task groups in kernel (e.g. exit_group)
  - scalable scheduling facilities O(1) scheduler
- Fast-path synchronization in user-mode
  - Futex (Fast Userspace Mutex) – avoids need to enter kernel for common case
  - FUTEX_WAIT/FUTEX_WAKE

Outlook

- 1996 talk by John Ousterhout: "Why threads are a bad idea"
  - Threads are hard to program (synchronization, deadlock)
  - Threads break abstractions
  - Threads are hard to make fast
  - Threads aren't well-supported
- Conclusion: use threads only when their power is needed, for true CPU concurrency – else use single-threaded event-based model
- SEDA & Capriccio (and others) followed
Summary

• Implementation issues:
  – Stack management
  – Preemptive vs. nonpreemptive
    • “cooperative multitasking”
  – User-level vs Kernel-level models
  – I/O management & signal implementation