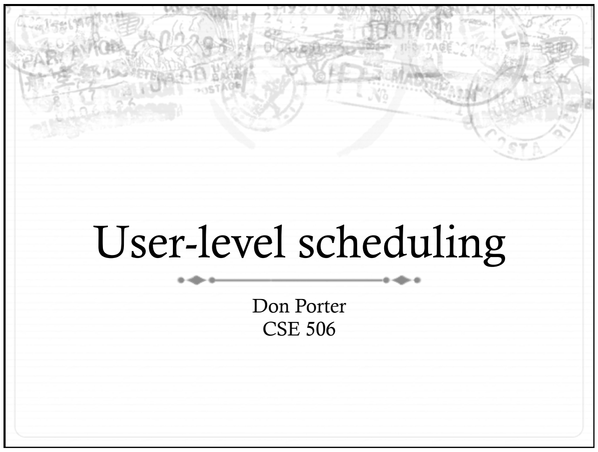
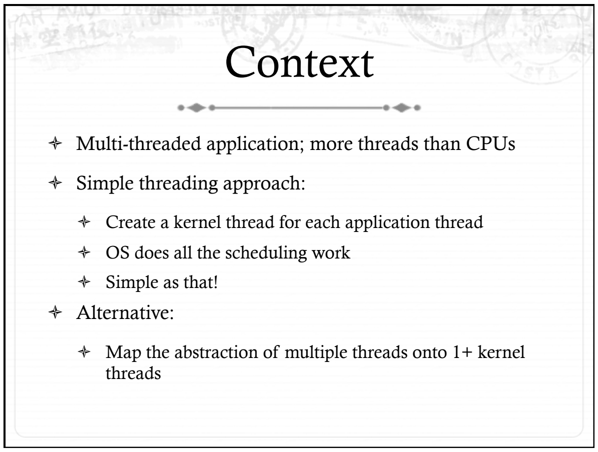


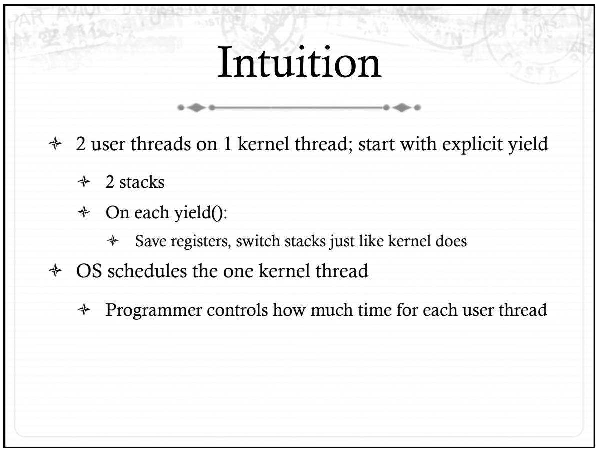
User-level scheduling

Don Porter
CSE 506



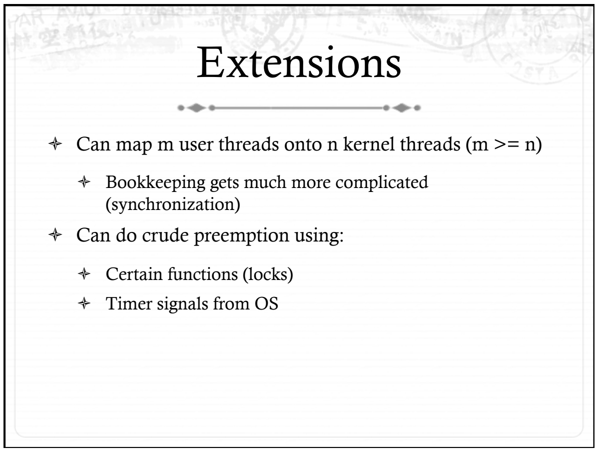
Context

- Multi-threaded application; more threads than CPUs
- Simple threading approach:
  - Create a kernel thread for each application thread
  - OS does all the scheduling work
  - Simple as that!
- Alternative:
  - Map the abstraction of multiple threads onto 1+ kernel threads



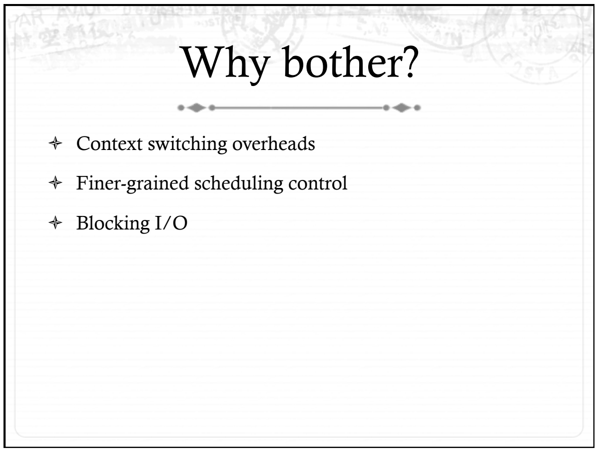
Intuition

- 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



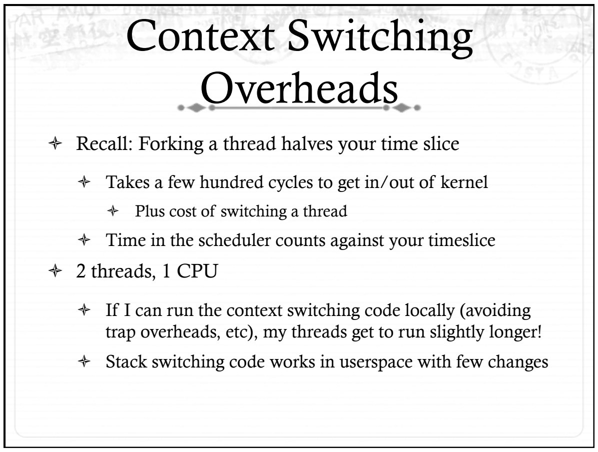
Extensions

- 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



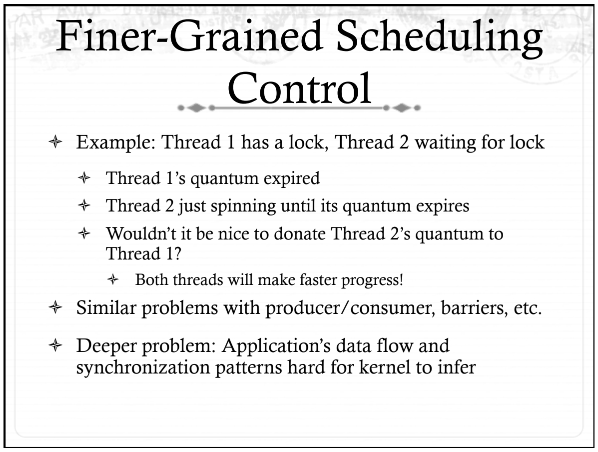
Why bother?

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



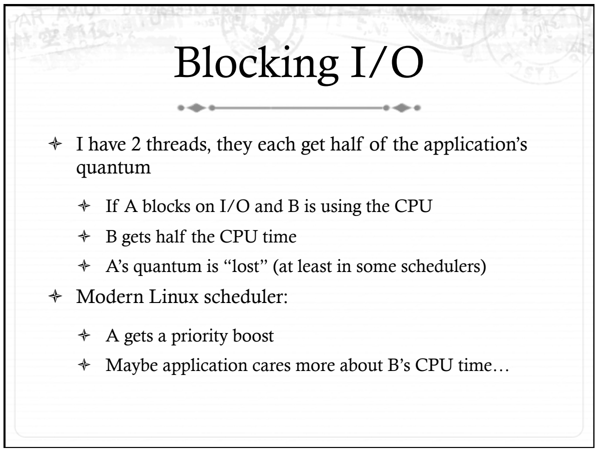
Context Switching Overheads

- 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…

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 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)

Process Start

- Rather than jump to main, kernel upcalls to scheduler
  - New timeslice
- Scheduler initially selects first thread and starts in "main"

New Thread

- When a new thread is created:
  - Scheduler issues a system call, indicating it could use another CPU
  - If a CPU is free, kernel creates a new SA
  - Upcalls to "New timeslice"
  - Scheduler selects new thread to run; loads register state

Preemption

- Suppose I have 4 threads running (T 0-3), in SAs A-D
- T0 gets preempted, CPU taken away (SA A dead)
- Kernel selects another SA to terminate (say B)
  - Creates a SA E that gets rest of B's timeslice
  - Calls "Timeslice expired upcall" to communicate:
    - A is expired, T0's register state
    - B is also expired now, T1's register state
- User scheduler decides which one to resume in E

Blocking System Call

- Suppose Thread 1 in SA A calls a blocking system call
  - E.g., read from a network socket, no data available
- Kernel creates a new SA B and upcalls to "Blocked SA"
  - Indicates that SA A is blocked
  - B gets rest of A's timeslice
- User scheduler figures out that T1 was running on SA A
  - Updates bookkeeping
  - Selects another thread to run, or yields the CPU with a syscall

Un-blocking a thread

- Suppose the network read gets data, T1 is unblocked
  - Kernel finishes system call
- Kernel creates a new SA, upcalls to "unblocked thread"
  - Communicates register state of T1
  - Perhaps including return code in an updated register
  - Just loading these registers is enough to resume execution
    - No iret needed!
- T1 goes back on the runnable list---maybe selected

Downsides

- 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

User Timeslicing?

- Suppose I have 8 threads and the system has 4 CPUs:
    - I will only ever get 4 SAs
- Suppose I am the only thing running and I get to keep them all forever
    - How do I context switch to the other threads?
    - No upcall for a timer interrupt
    - Guess: use a timer signal (delivered on a system call boundary; pray a thread issues a system call periodically)

Preemption in the scheduler?

- Edge case: A SA is preempted in the scheduler itself
    - Holding a scheduler lock
- Uh-oh: Can't even service its own upcall!
- Solution: Set a flag in a thread that has a lock
    - If a preemption upcall comes through while a lock is held, immediately reschedule the thread long enough to release the lock and clear the flag
    - Thread must then jump back to the upcall for proper scheduling

Scheduler Activation Discussion

- Scheduler activations have not been widely adopted
    - An anomaly for this course
    - Still an important paper to read:
        - Think creatively about "right" abstractions
        - Clear explanation of user-level threading issues
- People build user threads on kernel threads, but more challenging without SAs
    - Hard to detect preemption of another thread and yield
    - Switch out blocking calls for non-blocking versions; reschedule on waiting---limited in practice

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

Summary

- User-level threading is about performance, either:
  - Avoiding high kernel threading overheads, or
  - Hand-optimizing scheduling behavior for an unusual application
- User-threading is challenging to implement on traditional OS abstractions
- Scheduler activations: the right abstraction?
  - Explicit representation of CPU time slices
  - Upcalls to user scheduler to context switch
  - Communicate preempted register state