Scheduler Activations

Robert Grimm
New York University

The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?

Threads

- Provide a "natural" abstraction for concurrent tasks
  - Sequential stream of operations
- Separate computation from address space, other process state
  - One element of a traditional Unix process
- But also pose non-trivial challenges
  - Can be hard to program (think: race conditions, deadlocks)
    - [Savage et al. SOSP '97], [Engler & Ashcraft SOSP '03]
  - Are hard to implement the right way
    - User-level vs. kernel-level

User-Level Threads

- Advantages
  - Common operations can be implemented efficiently
  - Interface can be tailored to application needs
- Issues
  - A blocking system call blocks *all* user-level threads
    - Asynchronous system calls can provide partial work-around
  - A page fault blocks all user-level threads
  - Matching threads to CPUs in a multiprocessor is hard
    - No knowledge about # of CPUs available to address space
    - No knowledge when a thread blocks

Kernel-Level Threads

- Primary advantage
  - Blocking system calls and page faults handled correctly
- Issues
  - Cost of performing thread operations
    - Create, exit, lock, signal, wait all require user/kernel crossings
      - On Pentium III, `getpid`: 365 cycles vs. procedure call: 7 cycles
  - Cost of generality
    - Kernel-threads must accommodate all "reasonable" needs
    - Kernel-threads prevent application-specific optimizations
      - FIFO instead of priority scheduling for parallel applications

So, How About...

- Running user threads on kernel threads?
  - Core issues persist
    - Just like processes, kernel threads do not notify user level
      - Block, resume, preempted without warning/control
    - Kernel threads are scheduled without regard for user-level thread state
      - Priority, critical sections, locks → danger of priority inversion
  - Some problems get worse
    - Matching kernel threads with CPUs
      - Neither kernel nor user knows number of runnable threads
    - Making sure that user-level threads make progress

Enter Scheduler Activations

- Let applications schedule threads
  - Best of user-level threads
- Run same number of threads as there are CPUs
  - Best of kernel-level threads
- Minimize number of user/kernel crossings
  - Make it practical

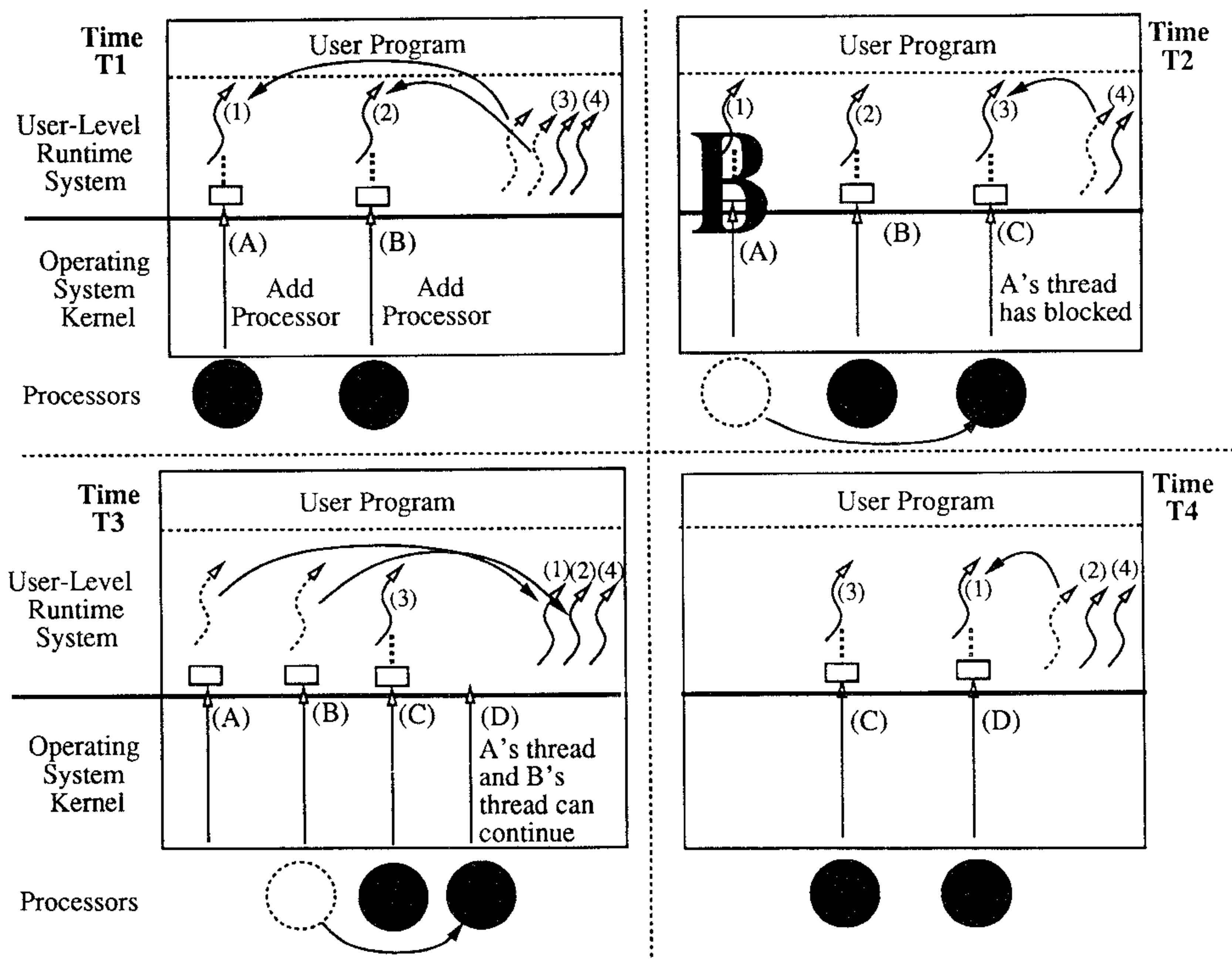
Activations as Virtual CPUs

- Execution always begins in user-level scheduler
  - Scheduler keeps activation to run thread
- Execution may be preempted by kernel
  but never resumed directly (see previous point)
  - Crucial difference from kernel threads — why?
- Number of activations
  - One for each on-going execution (i.e., actual CPU)
  - One for each blocked thread — why?

"Ups and Downs"

- Upcalls
  - New processor available
  - Processor has been preempted
  - Thread has blocked
  - Thread has unblocked
- Downcalls
  - Need more CPUs
  - CPU is idle
  - Preempt a lower priority thread
  - Return unused activation(s)
    - After extracting user-level thread state

An Example



- I/O request and completion

Number of Crossings

- For creating, exiting, locking, signaling, waiting?
- For full preemption (say only CPU), I/O?
- For partial preemption (say 1 CPU)?

Preemption

- Where is the thread's state?
  - Stack, control block at user-level
  - Registers at kernel-level → return with upcall
- What about preempting threads in critical sections?
  - Poor performance if thread holding spinlock
  - Deadlock if thread holding lock for scheduler data structures
- How to prevent these problems?
  - Detect thread in critical section
  - Finish critical section on next upcall
    - Copy of critical section returns to scheduler

Preemption (cont.)

- What if thread in critical section is blocked on a page fault?
  - We have to take the performance hit
- What if the scheduler causes a page fault?
  - We cannot create an arbitrarily large number of scheduler activations!
  - Rather, kernel detects this special case and delays activation

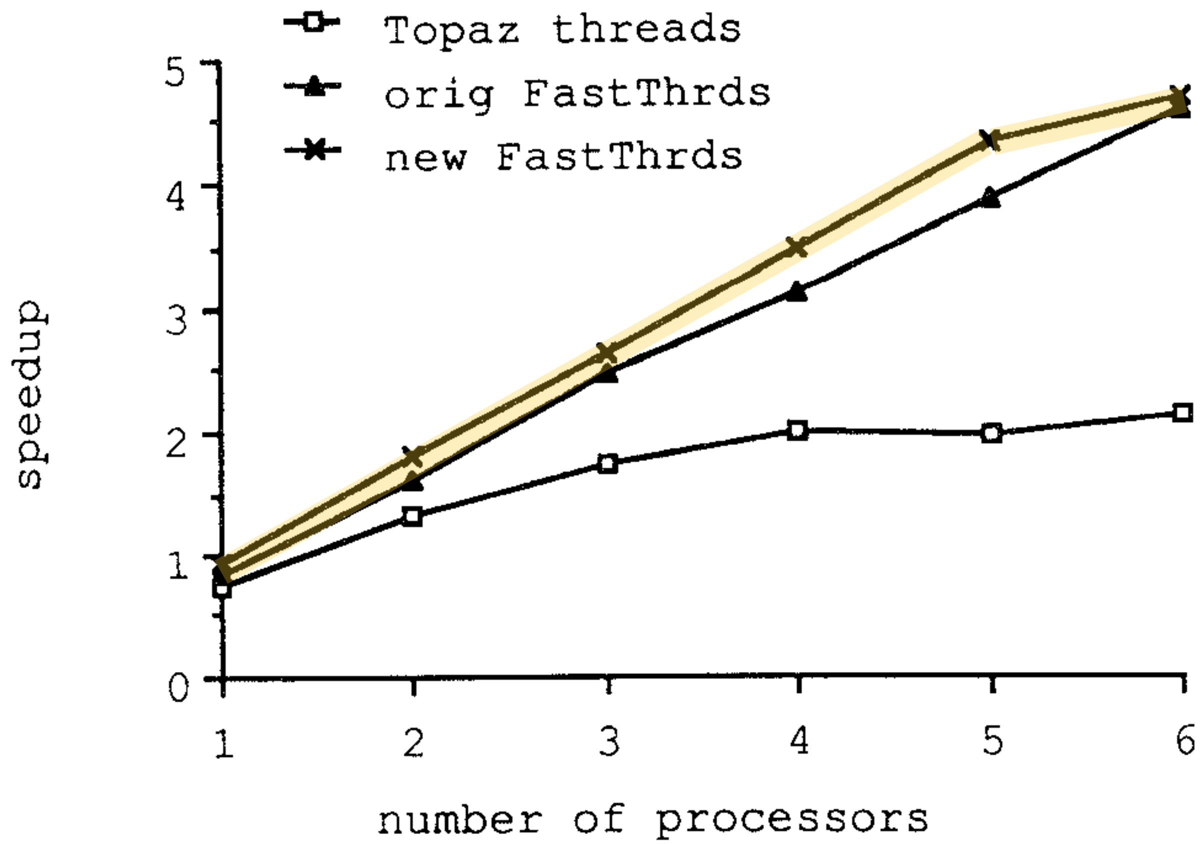
Evaluation

Micro-Benchmarks

Operation	FastThreads on Topaz threads	FastThreads on Scheduler Activations	Topaz threads	Ultrix processes
Null Fork	34	37	948	11300
Signal-Wait	37	42	441	1840

- What is the cost of thread operations? See above
- What is the cost of upcalls?
  - Cost of blocking/preemption should be similar to kernel-threads to make activations practical for uniprocessors
  - But, implementation is five times slower
    - Written in Modula-2+, rest of thread system in assembly
      - [Schroeder & Burrows TOCS '90] shows how to tune — really?

Application Performance

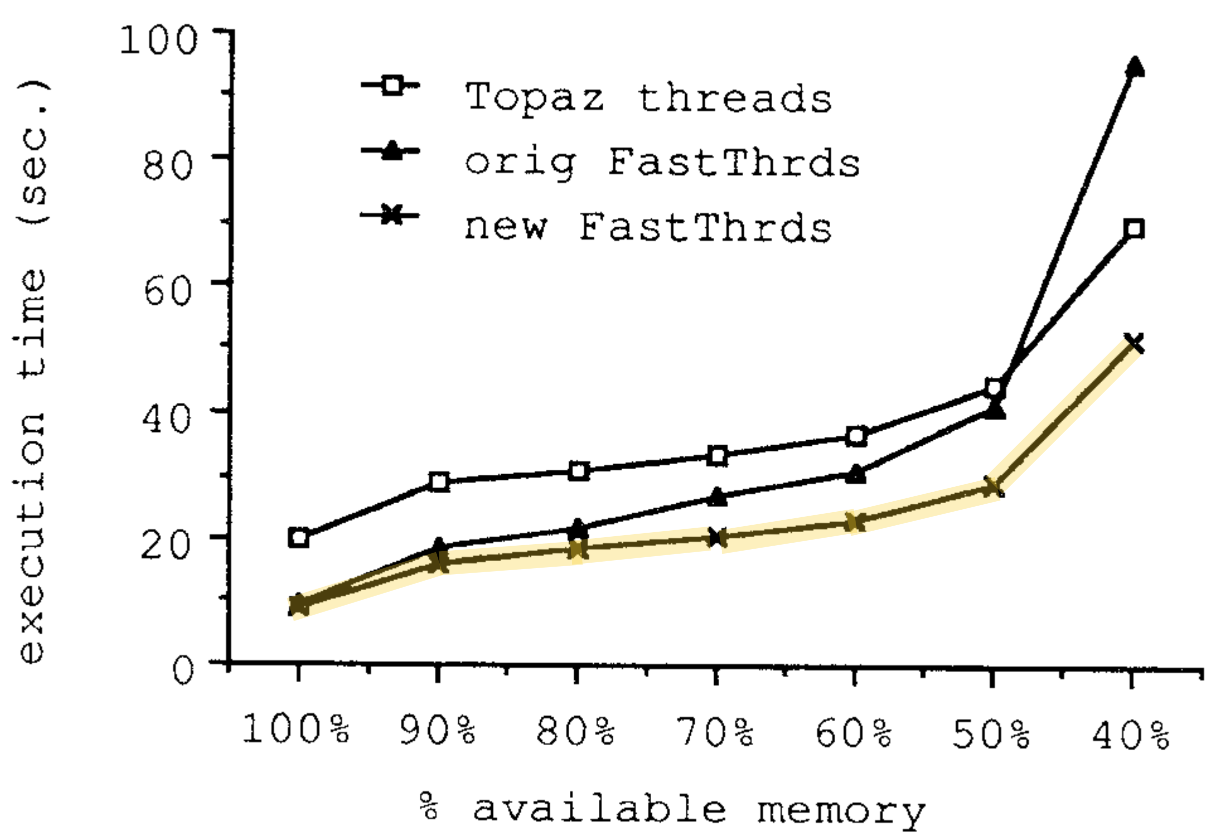


Speedup with all memory available

Topaz threads	Original FastThreads	New FastThreads
1.29	1.26	2.45

Speedup with 2 apps

Execution time with limited memory



What Do You Think?