Scheduler Activations

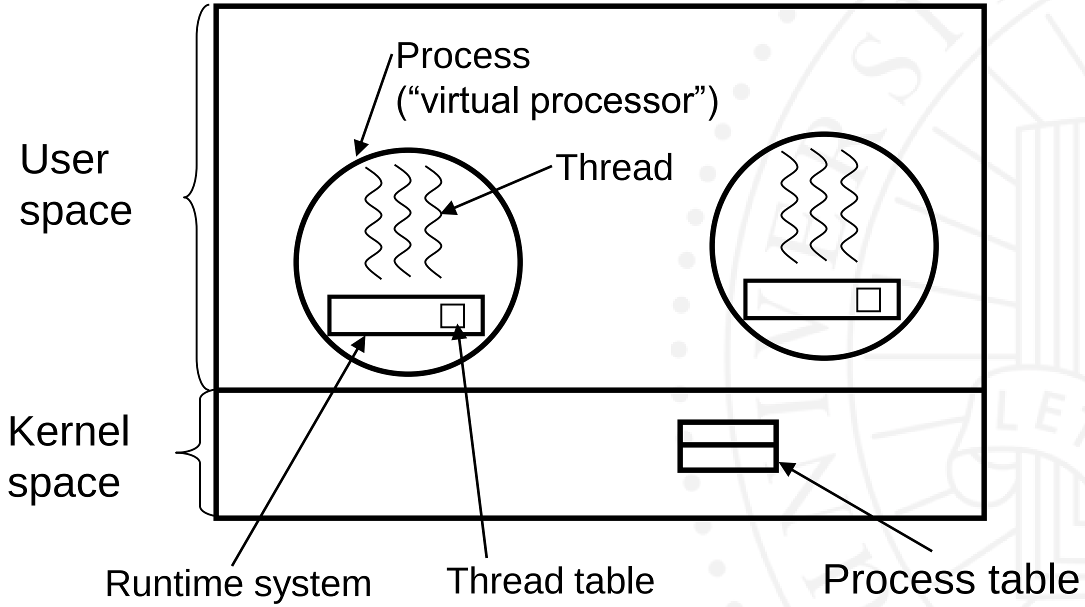
Adopted some slides from www.cs.pdx.edu/~walpole/class/cs533/winter2007/slides/92.ppt

Managing Concurrency Using Threads

- User-level library
  - Management in application’s address space
  - High performance and very flexible
  - Lack functionality
- Operating system kernel
  - Poor performance (when compared to user-level threads)
  - Poor flexibility
  - High functionality
- New system: kernel interface combined with user-level thread package
  - Same functionality as kernel threads
  - Performance and flexibility of user-level threads

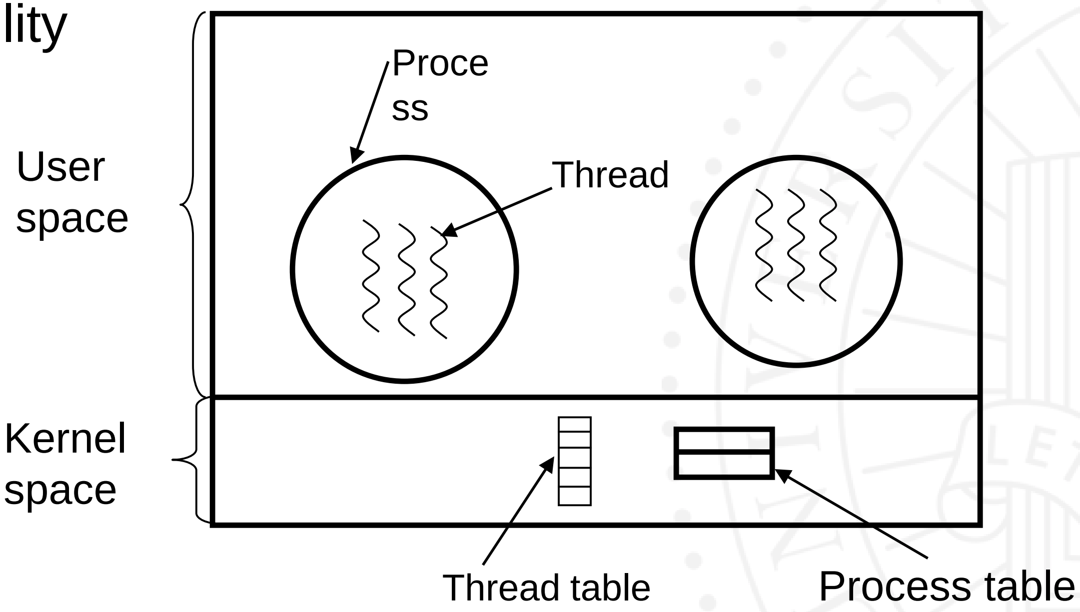
User-level Threads

- Thread management routines linked into application
- No kernel intervention == high performance
- Supports customized scheduling algorithms == flexible
- (Virtual) processor blocked during system services == lack of functionality
  - I/O, page faults, and multiprogramming cause entire process to block



Kernel Threads

- No system integration problems (system calls can be blocking calls) == high functionality
- Extra kernel trap and copy and check of all parameters on all thread operations == poor performance
- Kernel schedules thread from same or other address space (process)
- Single, general purpose scheduling algorithm == lack of flexibility



Kernel Threads Supporting User-level Threads

- **Question: Can we accomplish system integration by implementing user-level threads on top of kernel threads?**
- Typically one kernel thread per processor (virtual processor)
  - User-level thread blocks, so does kernel thread: processor idle
  - More kernel threads implicitly results in kernel scheduling of user-level threads
  - Increasing communication between kernel and user-level will negate performance and flexibility advantages of using user-level threads
- **Answer: No**
- **Also:**
  - No dynamic reallocation of processors among address spaces
  - Cannot ensure logical correctness of user-level thread system built on top of kernel threads

Goals (from paper)

- Functionality
  - No processor idles when there are ready threads
  - No priority inversion (high priority thread waiting for low priority one) when its ready
  - When a thread blocks, the processor can be used by another thread
- Performance
  - Closer to user threads than kernel threads
- Flexibility
  - Allow application level customization or even a completely different concurrency model

Problems

- User thread does a blocking call?
  - Application loses a processor!
- Scheduling decisions at user and kernel not coordinated
  - Kernel may de-schedule a thread at a bad time (e.g., while holding a lock)
  - Application may need more or less computing
- Solution?
  - Allow coordination between user and kernel schedulers

Scheduler activations

- Allow user level threads to act like kernel level threads/virtual processors
- Notify user level scheduler of relevant kernel events
    - Like what?
- Provide space in kernel to save context of user thread when kernel stops it
    - E.g., for I/O or to run another application

Kernel upcalls

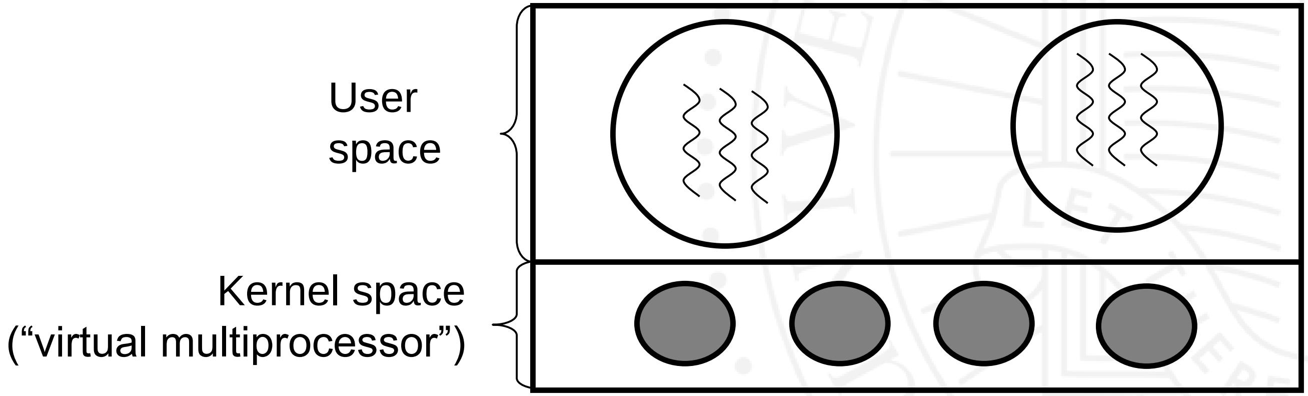
- New processor available
  - Reaction? Run time picks user thread to use it
- Activation blocked (e.g., for page fault)
  - Reaction? Runtime runs a different thread on the activation
- Activation unblocked
  - Activation now has two contexts
  - Running activation is preempted – why?
- Activation lost processor
  - Context remapped to another activation
- What do these accomplish?

Runtime->Kernel

- Informs kernel when it needs more resources, or when it is giving up some
- Could involve the kernel to preempt low priority threads
  - Only kernel can preempt
- Almost everything else is user level!
  - Performance of user-level, with the advantages of kernel threads!

Virtual Multiprocessor

- Application knows how many and which processors allocated to it by kernel.
- Application has complete control over which threads are running on processors.
- Kernel notifies thread scheduler of events affecting address space.
- Thread scheduler notifies kernel regarding processor allocation.



Scheduler Activations

- Vessels for running user-level threads
- One scheduler activation per processor assigned to address space.
- Also created by kernel to perform upcall into application's address space
  - "Scheduler activation has blocked"
  - "Scheduler activation has unblocked"
  - "Add this processor"
  - "Processor has been preempted"
- **Result:** Scheduling decisions made at user-level and application is free to build any concurrency model on top of scheduler activations.

Scheduler activations (2)

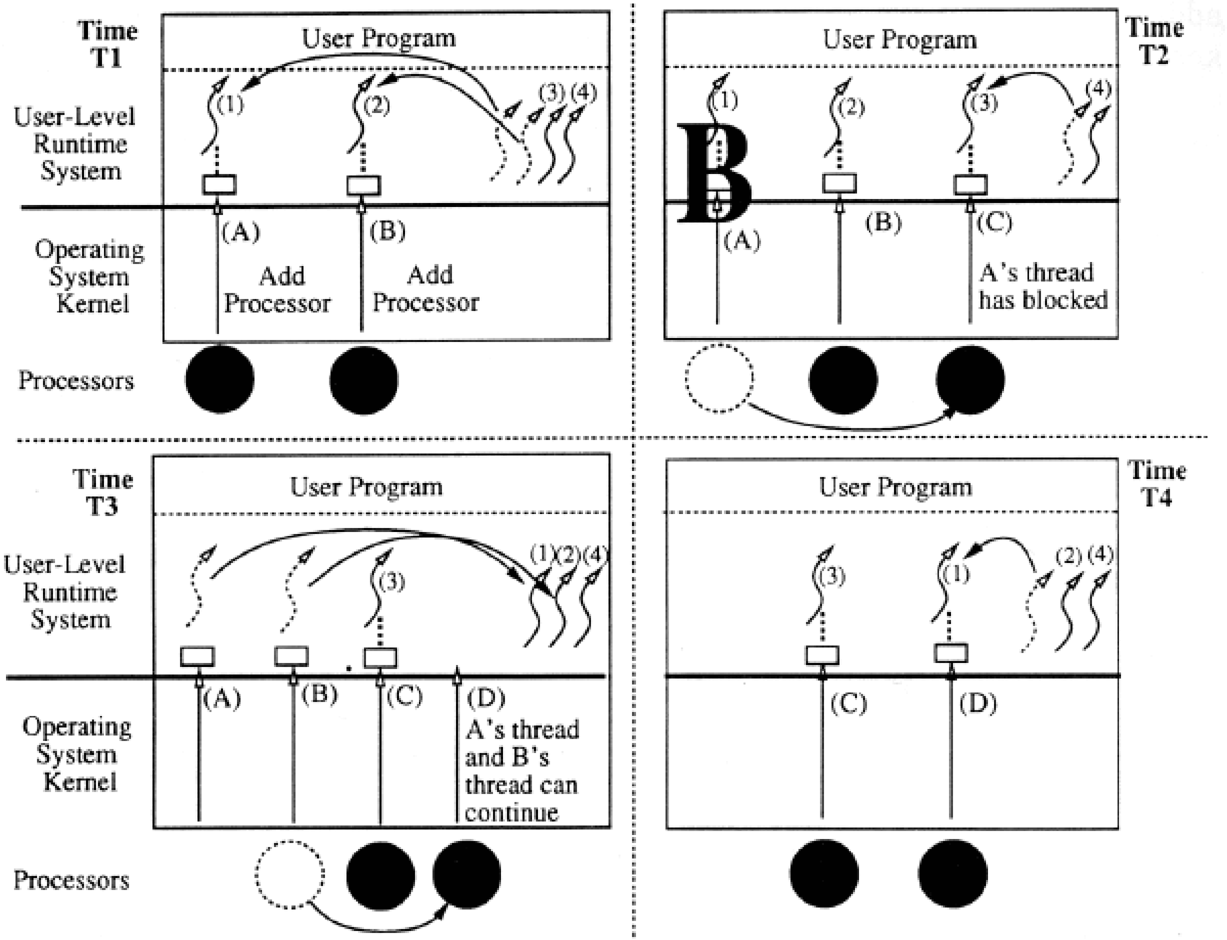


Fig. 1. Example: I/O request/completion.

Preemptions in critical sections

- Runtime checks during upcall whether preempted user thread was running in a critical section
  - Continues the user thread using a user level context switch in this case
    - Once lock is released, it switches back to original thread
    - Keep track of critical sections using a hash table of section begin/end addresses

Implementation

- Scheduler activations added to Topaz kernel thread management.
  - Performs upcalls instead of own scheduling.
  - Explicit processor allocation to address spaces.
- Modifications to FastThreads user-level thread package
  - Processing of upcalls.
  - Resume interrupted critical sections.
  - Pass processor allocation information to Topaz.

Performance

- Thread performance without kernel involvement similar to FastThreads before changes.
- Upcall performance significantly worse than Topaz threads.
  - Untuned implementation.
  - Topaz in assembler, this system in Modula-2+.
- Application performance
  - Negligible I/O: As quick as original FastThreads.
  - With I/O: Performs better than either FastThreads or Topaz threads.

Application Performance (negligible I/O)

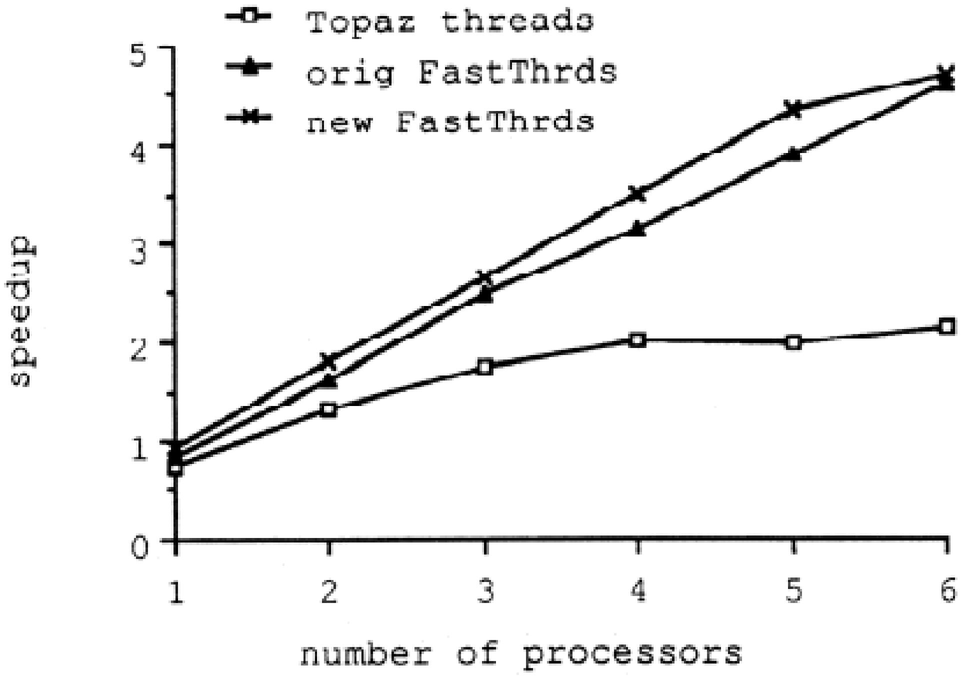


Fig. 2. Speedup of N-Body application versus number of processors, 100% of memory available.

Application Performance (with I/O)

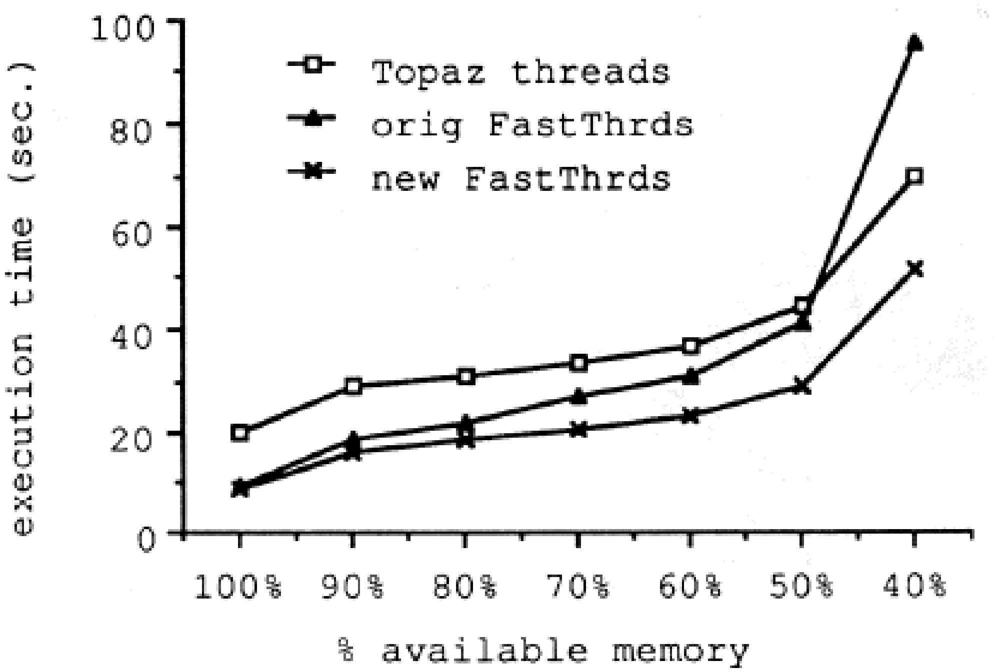


Fig. 3. Execution time of N-Body application versus amount of available memory, 6 processors.

Discussion

- Summary:
  - Get user level thread performance but with scheduling abilities of kernel level threads
  - Main idea: coordinating user level and kernel level scheduling through scheduler activations
- Limitations
  - Upcall performance (5x slowdown)
  - Performance analysis limited
- Connections to exo-kernel/spin/microkernels?