Outline

- ACM Symposium on OS Principles (SOSP) 1991
  - Using Continuations to Implement Thread Management and Communication in Operating Systems *Richard P. Draves, Brian N. Bershad, Richard F. Rashid, Randall W. Dean*
    - Reduce kernel thread stack space
  - Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Hank M. Levy*
    - Cooperative mechanism to enjoy the benefits of user level and kernel level threads

OS/Application threading…

- When a application requests service from the kernel, the user level process saves its state and makes a system call into kernel (crossing the supervisor protection boundary). The question is, what happens inside the kernel, do you have a kernel thread per user level threads? What happens if the kernel thread blocks on page fault (multiprocessor kernels allow kernels to be paged in)

Processing within kernel

- Process model
  - Multiple threads within kernel - one kernel stack per thread
  - Unique kernel stack - rescheduled and descheduled transparently (entire state is saved)
  - E.g. UNIX
  - Easy to use as blocking is transparent
  - Cannot optimize unwanted “stack” space (4KB per thread)
- Interrupt model
  - Single per-processor stack
  - Threads explicitly save state before blocking
  - E.g. Quicksilver, V

Possible solution

- User level thread - one kernel thread for many user level threads
  - At least need one kernel level thread
  - Blocked kernel threads still wasted resources
- Continuations
  - Provide code to save state
  - Behaves like process model
  - Performance of interrupt model
  - Allows further optimizations

Key idea

- Optimizing Mach 3.0
  - Application level representation of state while blocked
  - Application code to restore stack
  - Optimizations to reduce continuation operations: fast hand off for RPCs
    - Tradeoff stack space for complexity (application code)
- Is this relevant?
  - Current processors have lots of memory, so why bother?
  - Per processor kernel stack
    - reduce cache and TLB misses
- Software engineering concerns?
- Interrupt driven, co-routine style services
  - Much current work in MS Research and other places

User level, Kernel level threads?

- User level threads
  - Fast - the kernel does not need to know when there is a switch
  - Flexible - each process can use its own scheduler
  - Can block - Kernel does not know about the existence of user level threads
  - Question: How many kernel threads to use?
    - If you use too little, then you don’t fully use the system and blocked threads can be a problem
    - if you use too much (to protect against blocked threads) then the OS can schedule kernel threads that have a blocked/idle user level thread, threads that are in spinlock (while (condition is false);), priority inversion of user level threads
- Kernel level threads
  - Can block - kernel can schedule other kernel level threads
  - Slower - protection boundary crossed

Scheduler activation

- Cooperative mechanism
  - Kernel informs the user process of number of virtual “processors” as well as change in the number of processors
  - User threads use these processors without informing the kernel on the scheduling decisions
  - Scheduling decision by the user level library + Processor allocation, blocked thread processing etc by kernel scheduler activation mechanism
    - User thread can request and relinquish virtual processors
    - Upcalls from kernel to application
    - System calls from application to kernel