Scheduler Activations

- Motivation
- Design
- Implementation
- Performance

Where do we use threads?

- Blocking I/O calls
- Client/Server interactions
- Parallelism

- Why can't we use processes for those jobs?
  - shared memory, for example, DB shared buffer pool
  - overhead for fork

Thread Abstractions

- User-level thread
  - state: thread id, saved registers, stacks, local storage (for library calls)
  - creation: 40 lines of C code
  - exit: 6 lines
  - context switching user level threads:
    - save registers: 5-6 instructions on i386
    - switch stack pointers: 2 instructions
    - restore registers: 4-5 on i386

Thread Abstractions

- Kernel level threads
  - kernel level stack
  - saved registers
  - local storage
  - signal handling info
  - virtual time alarms
  - user time, CPU usage
  - pointers to scheduling queues;
- creation, exit, switching: at least a system call

Latency Comparison

- procedure call: 8 ns clock -> 24ns
- user-level thread switching:
- system call: 2 us
- kernel-level thread switching: >> 4us

How can we reduce kernel level thread overhead?

- kernel needs to protect itself
  - checking parameters
  - priority scheduling
  - preemptive priority scheduling
  - priority inheritance...

Problems

- User level doesn't know about any blocking or preemption
  - solution 1: create more kernel level threads than available processor
  - problem: spin-waiting situations, idle kernel threads, priority problems, deadlock introduced by blocked kernel threads
- Kernel doesn't know which threads to take away

A Two-Level Solution

- What would be an ideal solution?
  - who has what role?
  - what is being communicated?
  - what's impact on application programmer?

Scheduler Activations

- user-level stack and kernel-level stack
  - the user-level stack is different from thread's stack
- kernel: save thread states
- user level thread manager: which thread is on the activation

=> similar to kernel threads

Upcall Points

- Add this processor (processor #)
- processor has been preempted(preemted activation # and machine state)
- scheduler activation has blocks (activation #)
- schedule activation is unblocked (activation #)

Three Examples

- Starting up
- Blocking I/O
- Processor Reallocation

Details

- User-level priority scheduling
- preemption during upcalls
- when is upcall made exactly

- why don't we worry about user level code executing too long during an upcall?

Passing Information from User Level to Kernel Level

Upcalls during Critical Sections

- Design
- Implementation

Implemented Prototype

- Processor Allocation Policy
- Cached scheduler activation

Performance

Pilot: an OS for PCs

- A special OS: written in one language (Mesa), supports only one language (Mesa)
- Pilot as a Mesa program that provides an *interface* through an *implementation*
  - types
  - procedures
  - constants
  - error signals

Files

- simple primitives
- uids
- file attribute: permanence
- immutable files

Virtual Memory

- One address space (protection?)
- Interface: Space
- Procedure: Space.Create
- Procedure: Space.Map
- Space as a swapping entity:
  - Space.Activate (pre-swapping)
  - Space.Deactivate (free and/or writeback)
  - Space.Kill (delete)

Mapping versus Read/Write

- File mapping:
  - pros
  - cons
- File read/write interface:
  - pros
  - cons

Streams and I/O devices

- transducer: export a sequential stream interface for a device

Communications

- network addresses using processor ID
- "sockets" as an I/O device with Streams interface (datagram service)
- Network Streams (similar to TCP)

Implementation Features

- Storage system managers implemented on top of the swapper and filer "kernels"
  - the "kernels" are small, powerful and handle common cases