Processing
(threads, agents, formalism)

How can processing activity be structured on a single processor?

How can application-level information and system-level information be combined to provide efficient scheduling of processing activities?

Why is mobility of a processing activity desired and how can it be achieved?

How can the concepts of communication, processing, and mobility be represented in a formal model?
Support for concurrent and parallel programming

- conform to application semantics
- respect priorities of applications
- no unnecessary blocking
- fast context switch
- high processor utilization

Relative importance:

- Functional
- Performance
“Heavyweight” Process Model

- simple, uni-threaded model
- security provided by address space boundaries
- high cost for context switch
- coarse granularity limits degree of concurrency
“Lightweight” (User-level) Threads

- thread semantics defined by application
- fast context switch time (within an order of magnitude of procedure call time)
- system scheduler unaware of user thread priorities
- unnecessary blocking (I/O, page faults, etc.)
- processor under-utilization
Kernel-level Threads

- thread semantics defined by system
- overhead incurred due to overly general implementation and cost of kernel traps for thread operations
- context switch time better than process switch time by an order of magnitude, but an order of magnitude worse than user-level threads
- system scheduler unaware of user thread state (e.g., in a critical region) leading to blocking and lower processor utilization
Problem

• Application has knowledge of the user-level thread state but has little knowledge of or influence over critical kernel-level events (by design! to achieve the virtual machine abstraction)
• Kernel has inadequate knowledge of user-level thread state to make optimal scheduling decisions

Solution: a mechanism that facilitates exchange of information between user-level and kernel-level mechanisms.

A general system design problem: communicating information and control across layer boundaries while preserving the inherent advantages of layering, abstraction, and virtualization.
Scheduler Activations: Structure

- Change in processor requirements

Scheduler activations

Kernel support

User

Thread library

- Change in processor allocation
- Change in thread status
Communication via Upcalls

The kernel-level scheduler activation mechanism communicates with the user-level thread library by a set of upcalls:

Add this processor (processor #)
Processor has been preempted (preempted activation #, machine state)
Scheduler activation has blocked (blocked activation #)
Scheduler activation has unblocked (unblocked activation #, machine state)

The thread library must maintain the association between a thread’s identity and thread’s scheduler activation number.
Role of Scheduler Activations

Invariant: there is one running scheduler activation (SA) for each processor assigned to the user process.
Avoiding Effects of Blocking

Kernel threads

Scheduler Activations
Resuming Blocked Thread

1: unblock

2: preempt

3: upcall

4: preempt

5: resume

user

kernel
Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>FastThreads on Topaz Threads</th>
<th>FastThreads on Scheduler Activations</th>
<th>Topaz Threads</th>
<th>Ultrix process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null fork</td>
<td>34</td>
<td>37</td>
<td>948</td>
<td>11300</td>
</tr>
<tr>
<td>Signal-Wait</td>
<td>37</td>
<td>42</td>
<td>441</td>
<td>1840</td>
</tr>
</tbody>
</table>