The Case for Thread Migration:
Predictable IPC in a
Customizable and Reliable OS
Published: OSPERT 2010

Gabriel Parmer
The George Washington University

Presented by Michael Peter
Component Based OS

- High degrees of system specialization
- Enhanced reliability
  - Scope of fault side-effects limited to component
- Decomposition into fine-grained components
  - Providing policy, abstraction, mechanism
  - Accessed by other components through interface
- IPC as key mechanism
  - Synchronous IPC
  - Thread migration
Synchronous IPC

Figure 1. Synchronous IPC between threads. Threads annotated with a $w$ are on a wait-queue associated with the server thread.
Thread Migration

**Figure 2.** Thread migration. Execution contexts are spread across components, but the same schedulable entity traces invocations.
Figure 3. COMPOSITE kernel data-structures involved in an invocation. A syscall specifies a capability (associated with $C_a$) that yields the component to invoke ($C_b$). A thread’s invocation stack saves the invoking component and enough state to return from the invocation (namely, the instruction and stack pointers).
TM in Composite (2)

- Capability controlled
- Linked through invocation stack
  - Return capability
- Execution stack retrieval
  - Restartable atomic operations
  - Block if no stack is available
- 4 word payload
- $0.7 \mu s$ for 2.4GHz P4, 1GHz Pentium M
  - Same order of magnitude as L4 systems
Allocation and Accounting

Figure 4. Invocations through components: (a) thread migration, (b) synchronous IPC.
Sync IPC Accounting and Execution

• Client execution accountability
  • Server thread is not tied to client thread
• Real-time task model
  • “An additional problem arises as the priority of a thread is often associated with its C (e.g. in rate-monotonic scheduling). It is not obvious how to assign priorities to threads throughout the system in the presence of pervasive thread dependencies.”

• Priority-inversion
  • Static structuring proposed
TM Accounting and Execution

- Priority inversion *within* a server component
- Composite: locking policies implemented in components
  - Details omitted
Sync IPC Implementation

- Lazy scheduling on IPC
  - Uncertainty as to the point of thread switch
  - Questioned if necessary at all
- Decoupling contexts (execution, scheduling)
  - Dependency graph traversal
Sync IPC Endpoint Contention

Figure 5. Invocations of and contention on various end-points. (a) All client threads invoke separate server threads. (b) Client threads invoke the same server thread, adding two to the server thread’s wait-queue. (c) Thread migration: execution contexts aren’t the target of invocation, thus cases similar to (b) are impossible.
Sync IPC Endpoint Contention (2)

- Wait queue placement if server thread is active
- Wakeup decision is proportional to length of queue
  - Bound number of calling threads
  - Has to be enforced for malicious clients
- One server thread for each client
  - Number of threads in a prot domain is often bounded
  - Resource consumption of threads (memory)
- Client addresses specific server thread
Figure 6. Retrieving execution contexts with thread migration. (a) Stacks are maintained on a freelist in the invoked component, or (b) in the kernel.
Migr Thread Endpoint Contention (2)

- Component is the invocation endpoint
  - Stacks are assigned as needed
- Lack of stacks results in blocking
  - Stack manager implements prio inheritance
- Differentiated services possible
  - Dedicated stacks for hard real-time
- Composite: stack allocation policy at user level
  - Kernel involvement conceivable (Pebble)
Sync IPC Enhancements

- Locating execution contexts
  - Attach thread pool to capability
  - Mimics thread migration with scheduling context handover
Component-based Scheduling

- Invocation path should not involve scheduling
  - No dependency on thread properties (prio)
  - No wakeup of more than one thread
  - Blocking and wakeup in scheduler component
- Pure sync IPC does not satisfy either
  - Receiver wakeup requires priority aware dispatch
  - Reply-wait may result in two runnable threads
Mutable Protection Domains

- MPD recognize significant overheads of fine-grained component-based systems
- Lift isolation between heavily interacting components
  - Erect or tear down boundaries as appropriate
  - Isolation maintained for most protdom boundaries
  - Significant performance improvements (40%)
- Intra-task thread switch is more expensive than intra-task component invocation
Thread Migration Limitations

- Execution context unavailability
  - Policies defined as components
    - Service class partitioning
    - Priority inheritance
- Fault recovery
  - IPC abort for sync IPC
  - Migrating threads spread across multiple protdoms
    - Should not be terminated upon fault
    - Forced return
Discussion

• What are the details of blocking?
  • Resource access is potentially an operation with global impact (PCP, SRP)

• What exactly is the scheduling scheme?
  • Periodic tasks?
  • Shared resources?
    – Access protocol?

• What is the trust relation between source and target domain?
  • Sender starts migration with almost depleted timeslice
  • Target does not return
Discussion (2)

• How are non-RT workloads accommodated?
  • Many RT scheduling schemes are not work preserving. Tasks with static priorities have a timeslice length attached. Utilization peaks out at appr. 69% for the general case.

• How are legacy workloads dealt with?
  • Is it possible to host a Linux kernel?

• How does memory management work?
  • Privileged component with pagetable manipulation rights does not fit local management approach.
  • Map is tightly integrated in IPC
    – Security concerns (receive window)