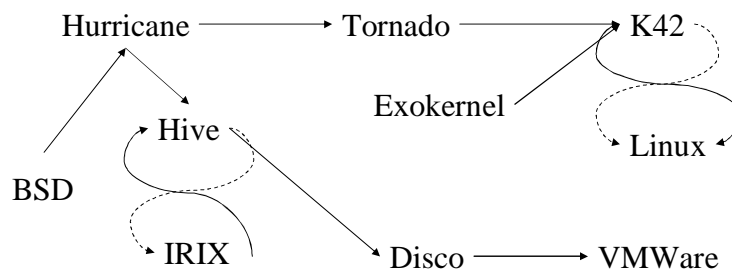


The K42 Research Operating System, with a focus on scalability

Orran Krieger for K42 group
<http://www.research.ibm.com/K42>

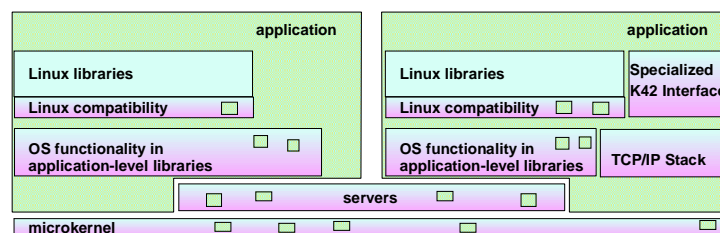
History



Goals

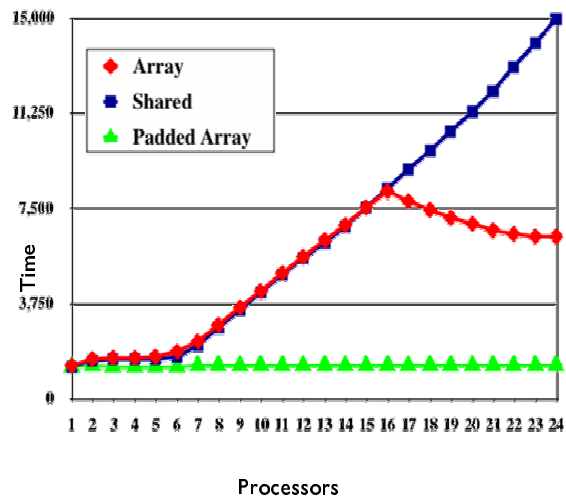
- scalability:
 - up to large MP and large applications
 - down for small-scale MP and small apps on large-scale MP
- flexibility/customizability:
 - policies/implementations of every physical/virtual resource instance can be customized to application needs
 - system can adapt to security and performance faults without penalizing common case performance
- maintainability/extensibility:
 - highly module structure
 - re-enable the OS research community
- full functionality and Linux compatibility:
 - support huge numbers of Linux apps, libraries and drivers without modification
 - exploit infrastructure as testbed for policies and implementations that can be transferred to Linux

Structure

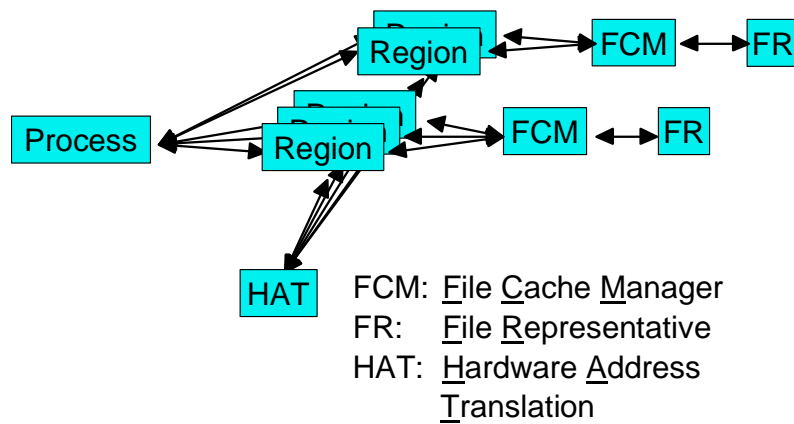


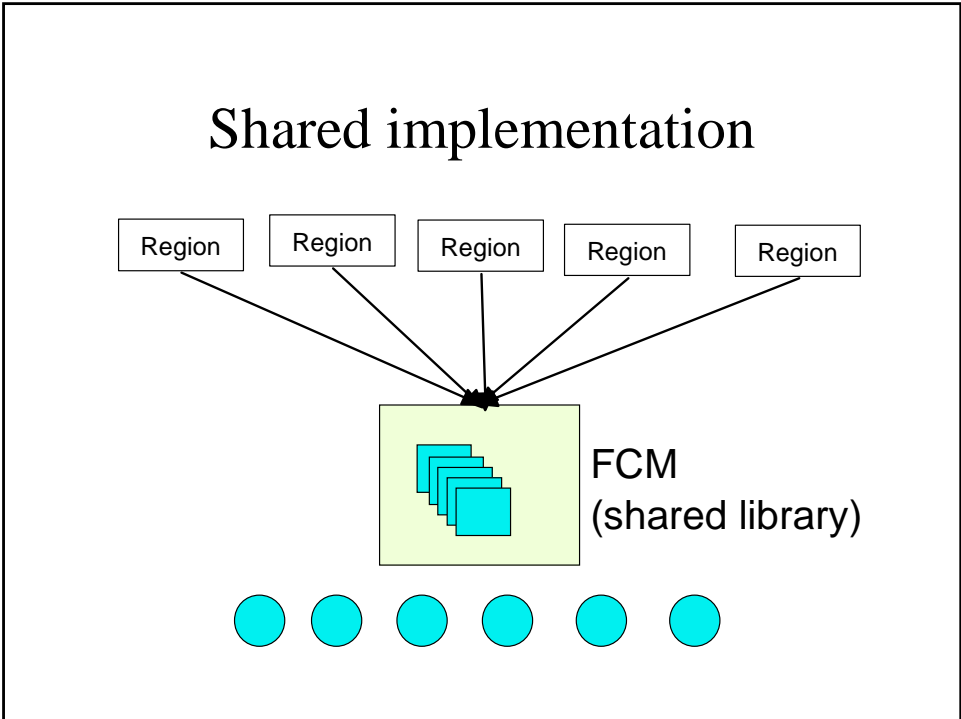
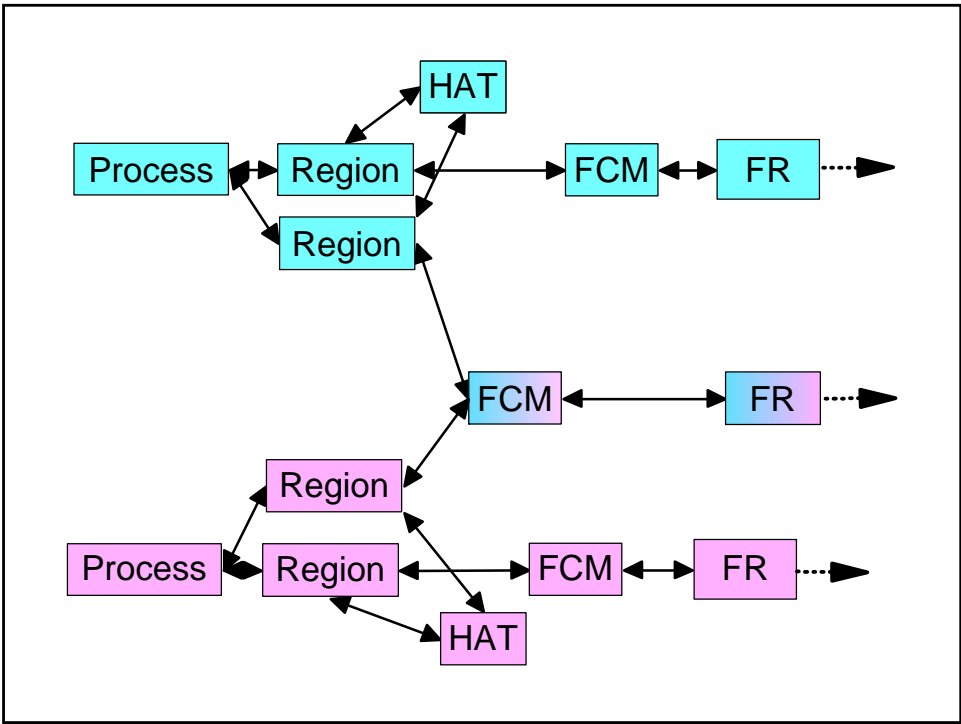
- Most OS functionality implemented in user-level library
 - examples: timers, thread library, TCP stack...
 - allows OS services to be customized for applications with specialized needs
 - also avoids interactions with kernel/servers and reduces space/time overhead in kernel/servers
- Object-oriented design at all levels

MP performance is hard!

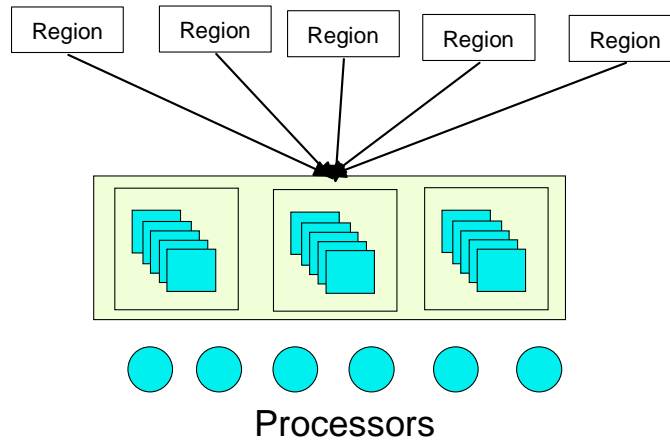


Object-oriented design

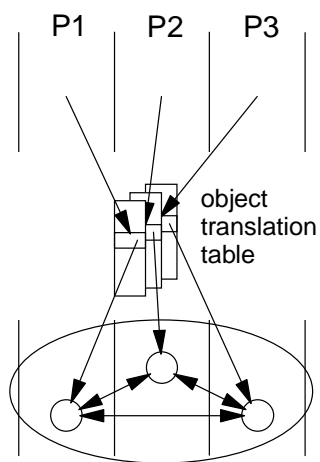




Distributed implementation



Clustered-object implementation



- Object-translation table: level of indirection, processor specific memory.
 - Overhead: a single cached pointer dereference
 - Details:
 - Fixed number of physical page frames per processor.
 - Synchronization through root object.
 - Reprs created on demand.
 - RCU/garbage collection to free.
- Real complexity is in libraries to simplify usage.

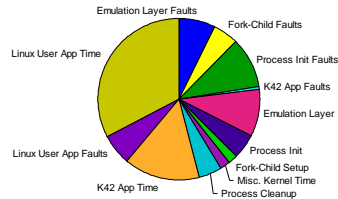
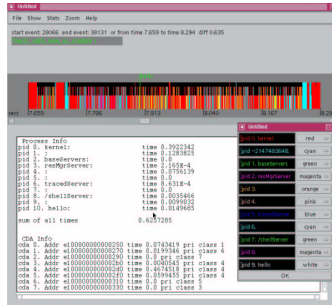
Hot-swapping

- change one system component/type for another without bringing system down
- potential uses
 - scalability
 - system availability
 - monitoring
 - flexibility, maintainability
 - extensibility
 - testing
 - performance

Hot-Swapping

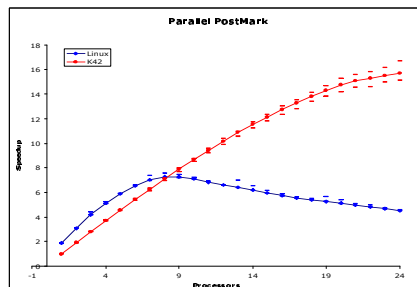
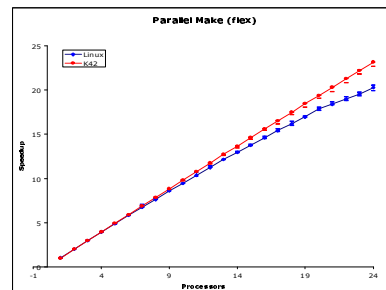
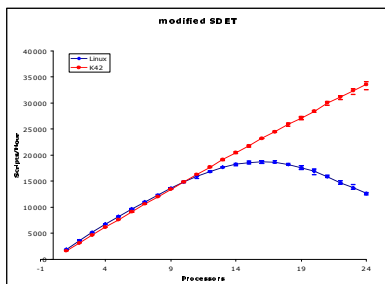
- Depends on:
 - object-oriented structure of system
 - technology to establish a quiescent state
 - also used for non-blocking synchronization, and avoiding locking hierarchies (RCU++)
 - level of indirection
 - also used for clustered objects
- Implementation has no additional overhead
 - when not swapping an object
 - for objects not being swapped
- Limitations:
 - is not instantaneous
 - initial prototype: coordinated swapping, factory, external policies...

Performance monitoring

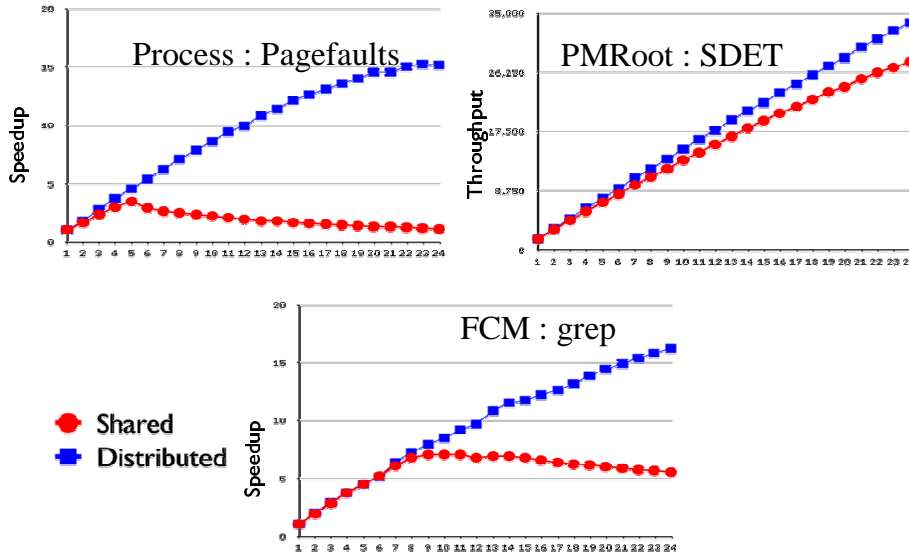


- unified cheap scalable non-blocking tracing infrastructure for correctness and performance debugging
- key parts of design recently transferred to LTT
- post processing tools easy to develop: lock contention, sampling, time breakdown, visualization, caching effects, ...
- in final numbers, disabled, but only % 1.6 difference UP and 24 way

Independent workloads

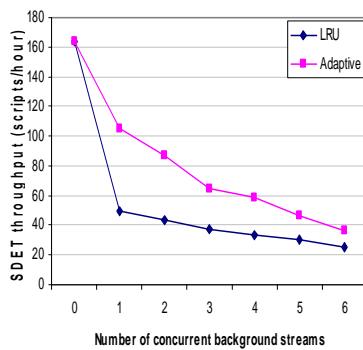


Clustered Objects and Hot-swapping

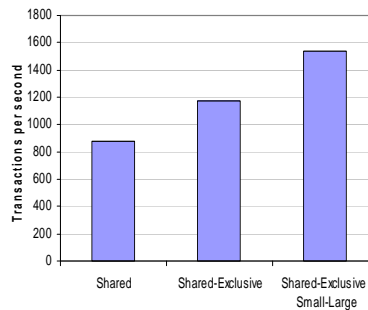


More hot-swapping

Adaptive paging



Adaptive file imp.



Status

- Performance is pretty good, scalability is pretty good, some core assumptions validated, real enough to run significant apps and benchmarks.
- Running unmodified Linux PPC 64-bit binaries.
- Availability under LGPL.
- Need to flesh out many toy implementations.
- Need to explore performance with larger subsystems and applications.
- Need to modify subsystems/applications to exploit K42 event mechanisms and customization capabilities.
- Need to build infrastructure and policies to drive hot-swapping.
- Focus has so far been on mechanisms and infrastructure, have hardly started the real research.

Discussion

- Object oriented design has advantages...
 - have found special casing easy
 - hot-swapping simpler than adaptive algorithm
 - clustered-objects relatively simple to do
 - local fixes, publish interface not structure
- General performance monitoring infrastructure key to identifying problems.
- Overheads of our approach:
 - advantages of Linux's hierarchical page tables: exception level traversal, identify PT entry for fast unmap and avoid segment unmapping, aggressive fork pre-mapping for anonymous memory
 - user-level implementation: cost initialization, page fault costs on fork
 - OO design: indirections, code replication, poor instruction cache locality, per-object data structures...
 - initialization costs of scalable implementations
 - We seem to be getting close UP via: lazy initialization, hot swapping/specialization... hope to avoid exception level pinned data structures

Discussion

- All policies based on local information.
- Dealing with multitude of implementations
 - How many will actually be valuable?
- Transferring technology back to vanilla Linux.
 - Much of the infrastructure (allocation, locks...) independent of our radical notions.
 - Even radical ideas, like RCU, can be eventually transferred if sufficiently compelling performance/complexity/functionality.
 - Our environment provides great way to prototype policies and implementations.

Impact on Linux

- IBM strategy
- PPC64
 - SVR4 ABI
 - Gcc and toolchain
 - Glibc
 - Gdb
- LTT
- RCU
- Object-based reverse mapping

What I didn't talk about

- Fully preemptive and pageable kernel
- Processor-specific memory and slew of other technologies/tricks for scalability
- Clustered-object implementation
- Facilities to simplify distributed implementation
- Hot-swapping in general
- Model for Linux compatibility, what we would like to see change in Linux, services we have/are re-implementing in Linux
- Thread model, scheduling model, event model...
- RCU
- PPC specific issues, x86-64, portability model...
- Interesting sub-system and scientific workload
- New interfaces and services for applications
- Exotic file systems...
- Implementation of different OS servers: memory management, IPC, process management...