Providing Dynamic Update in an Operating System

Andrew Baumann, Gernot Heiser
University of New South Wales & National ICT Australia

Jonathan Appavoo, Dilma Da Silva,
Orran Krieger, Robert W. Wisniewski
IBM T.J. Watson Research Center

Jeremy Kerr
IBM Linux Technology Center

TEASER

Our system can apply updates to the running kernel

- ✓ Negligible performance impact
- ✓ Applicable to a mainstream OS

Overview:

① Problem & background
② Design
③ Implementation in K42
④ Experiences & results
⑤ Open issues
⑥ Related work
⑦ Conclusion

Problem: System Availability / Patches

Down time is increasingly expensive, we want the OS to be highly available

OSes require frequent updates, patches, etc.

- ✗ Requires restarting services, often a reboot
- ✗ Trade off cost of down-time *vs.* risk of remaining unpatched

PROBLEM: SYSTEM AVAILABILITY / PATCHES

Down time is increasingly expensive, we want the OS to be highly available

OSes require frequent updates, patches, etc.

- ✗ Requires restarting services, often a reboot
- ✗ Trade off cost of down-time *vs.* risk of remaining unpatched

How can we improve availability without sacrificing the ability to fix security holes, improve performance, etc?

PROBLEM: SYSTEM AVAILABILITY / PATCHES

Down time is increasingly expensive, we want the OS to be highly available

OSes require frequent updates, patches, etc.

- ✗ Requires restarting services, often a reboot
- ✗ Trade off cost of down-time *vs.* risk of remaining unpatched

How can we improve availability without sacrificing the ability to fix security holes, improve performance, etc?

Dynamic Update

DYNAMIC UPDATE

An established software technique (and field of research)

- ➜ Allows software updates to a running system while continuing to provide access to the resource managed by that software
- ➜ Previous work focused on update to applications and services
- ✓ This work focuses on dynamic update for operating systems

Issues in OS Dynamic Update

Classification of updates:

① code

② single-instance (global) data

③ multiple instance data structures

Issues in OS Dynamic Update

Classification of updates:

① code

② single-instance (global) data

③ multiple instance data structures

Potential limitations:

➜ What parts of the OS can be changed?
  ➜ e.g. low-level exception code

➜ How critical is the timeliness?
  ➜ May need guarantees for security updates

➜ Are updates to system libraries or middleware supported?

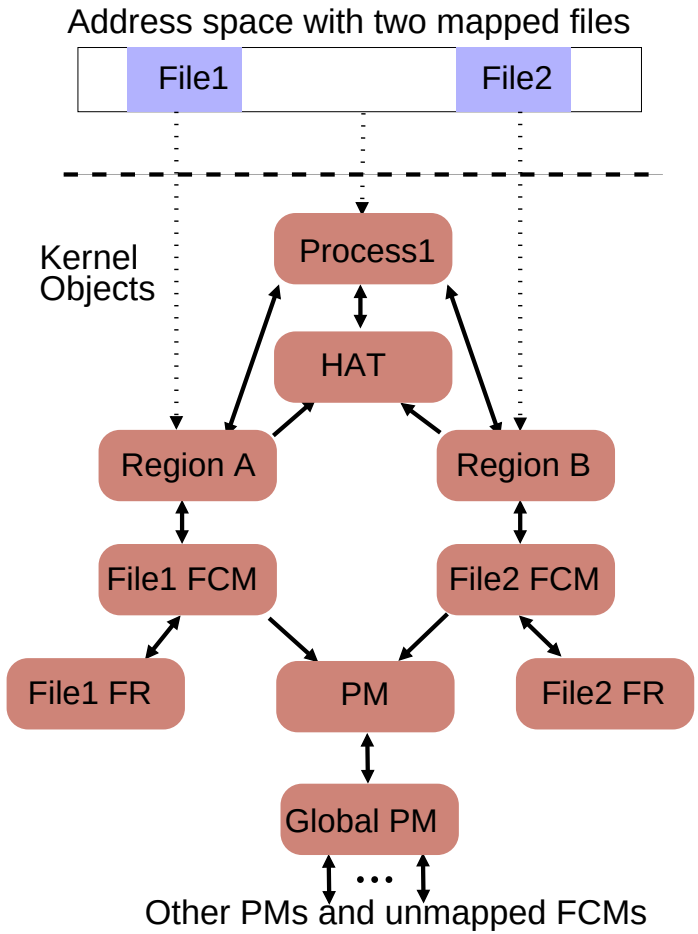
REQUIREMENTS

① Updatable unit
② Mechanism for detecting a safe point
③ State tracking
④ State transfer
⑤ Ability to redirect invocations
⑥ Version management

These could be provided in mainstream operating systems, such as Linux (further details in Linux.conf.au 2005).

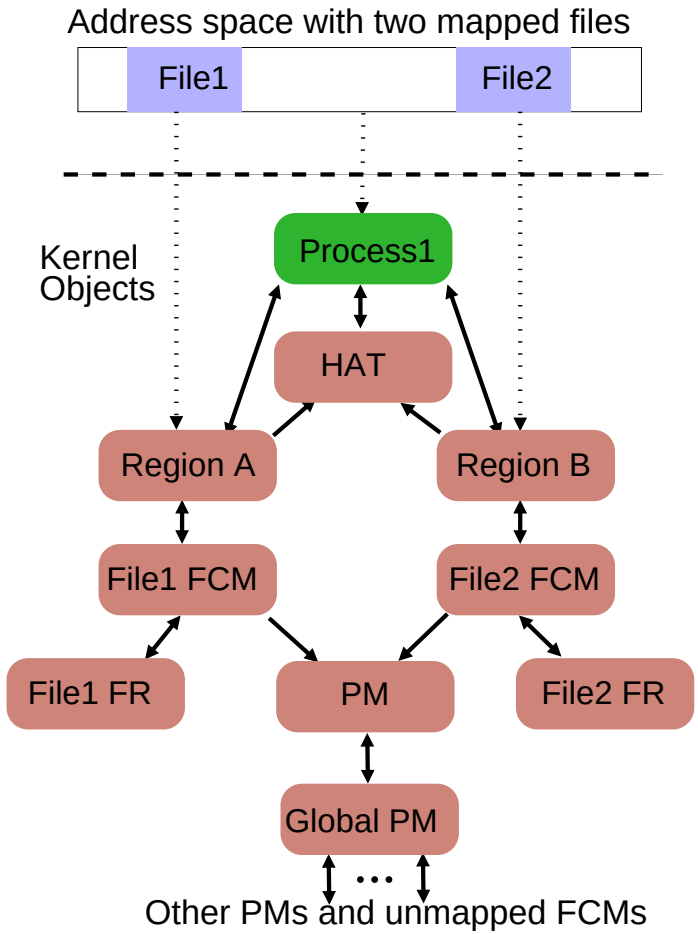
THE K42 OPERATING SYSTEM

- Scalable research OS:
  - Open source
  - Supports Linux API/ABI
  - Good prototyping environment
- Object oriented:
  - Each resource managed by set of object instances
  - Helps achieve scalability
- Supports object hot-swapping



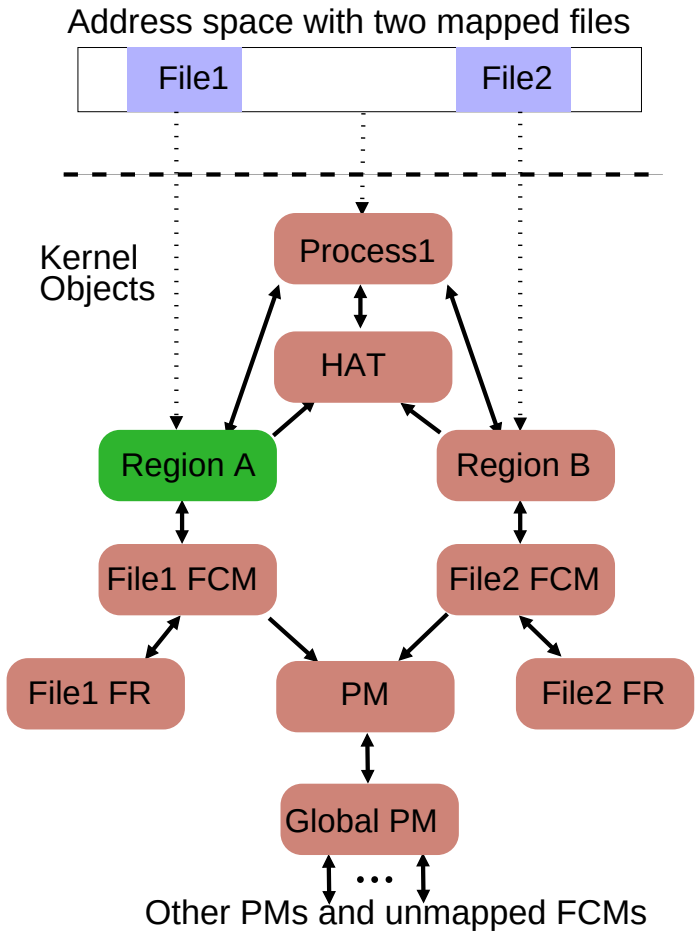
THE K42 OPERATING SYSTEM

- Scalable research OS:
  - Open source
  - Supports Linux API/ABI
  - Good prototyping environment
- Object oriented:
  - Each resource managed by set of object instances
  - Helps achieve scalability
- Supports object hot-swapping



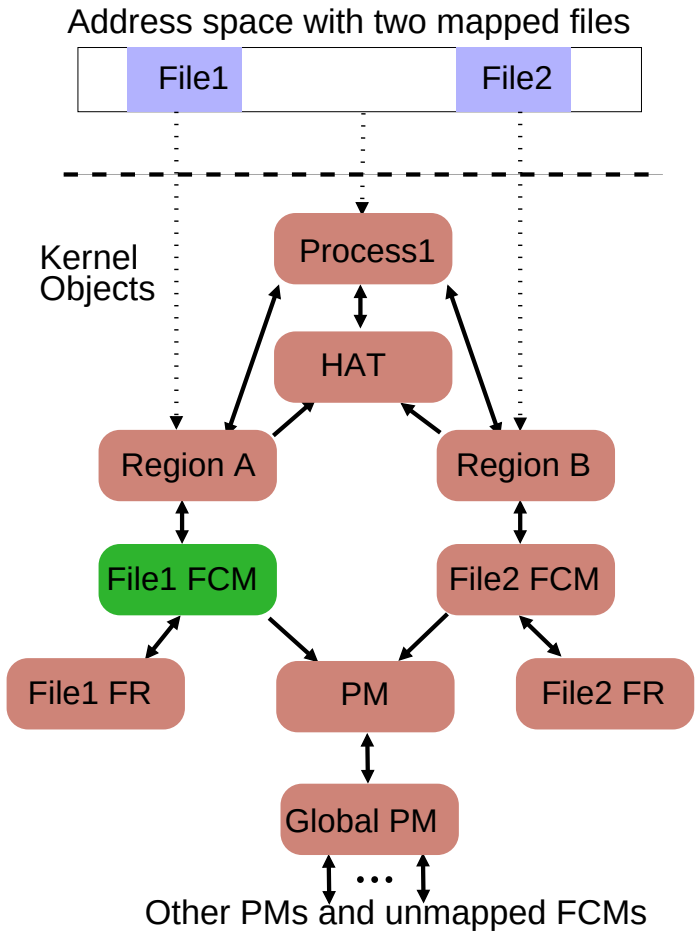
THE K42 OPERATING SYSTEM

- Scalable research OS:
  - Open source
  - Supports Linux API/ABI
  - Good prototyping environment
- Object oriented:
  - Each resource managed by set of object instances
  - Helps achieve scalability
- Supports object hot-swapping



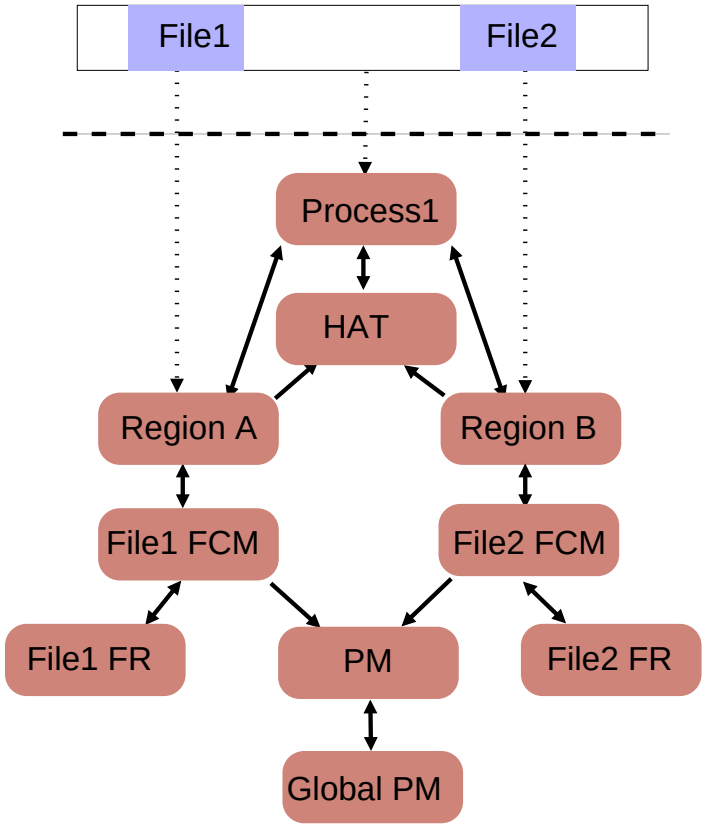
THE K42 OPERATING SYSTEM

- Scalable research OS:
  - Open source
  - Supports Linux API/ABI
  - Good prototyping environment
- Object oriented:
  - Each resource managed by set of object instances
  - Helps achieve scalability
- Supports object hot-swapping



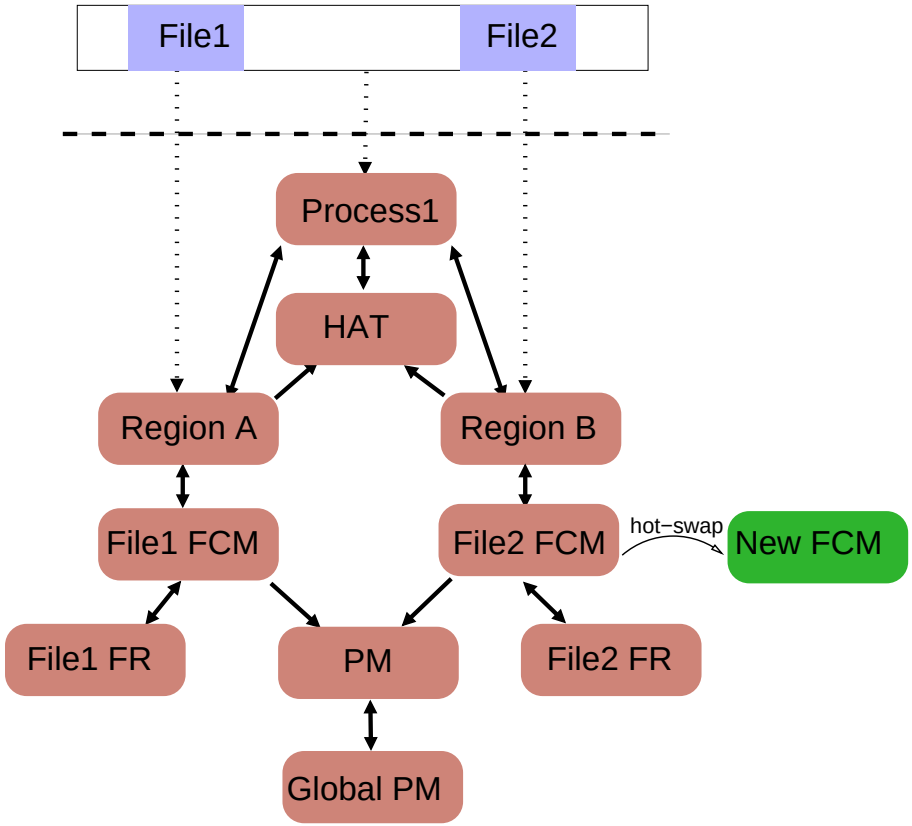
OBJECT HOT-SWAPPING

- ➜ Specific object instance substituted on-the-fly
- ➜ Designed for adaptation, customisability
  - ➜ e.g. change caching/prefetching policy
- ✓ Basis of dynamic update implementation



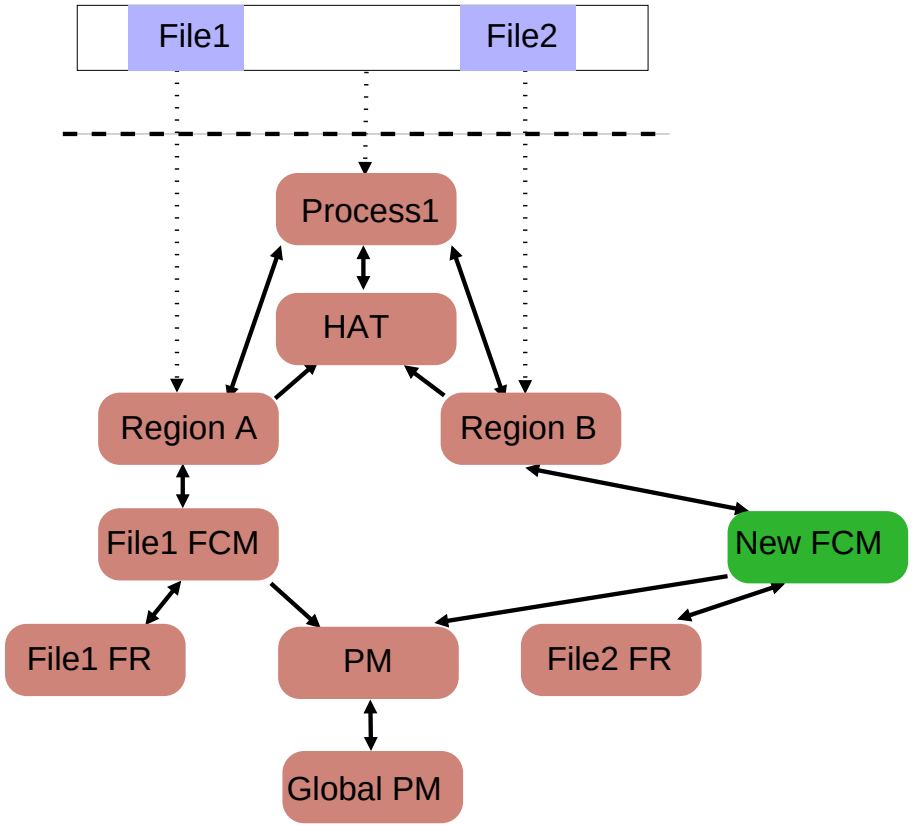
OBJECT HOT-SWAPPING

- ➜ Specific object instance substituted on-the-fly
- ➜ Designed for adaptation, customisability
  - ➜ e.g. change caching/prefetching policy
- ✓ Basis of dynamic update implementation



OBJECT HOT-SWAPPING

- ➜ Specific object instance substituted on-the-fly
- ➜ Designed for adaptation, customisability
  - ➜ e.g. change caching/prefetching policy
- ✓ Basis of dynamic update implementation



Dynamic Update in K42

Our prototype addresses the requirements:

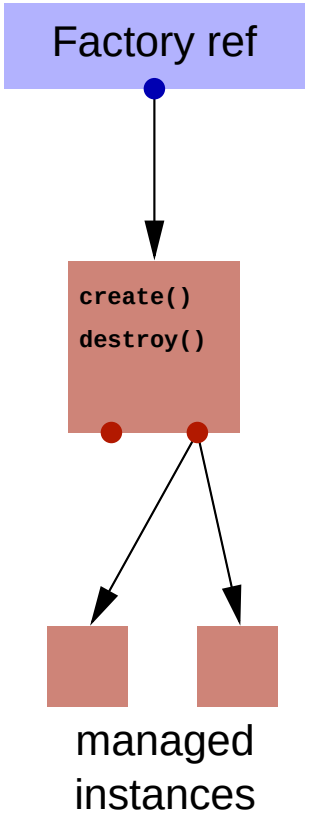
①	Updatable unit	Hot-swappable object instance
②	Safe point/quiescence	Read copy update (RCU) techniques
③	State tracking	Factory objects
④	State transfer	Transfer functions
⑤	Redirect invocations	Object translation table
⑥	Version management	Version numbers on factory objects

Factory Objects

- ➜ Use the factory design pattern
- ➜ Create and destroy objects of a specific class
- ➜ Track and manage object instances
- ✓ Improved model of object creation and management (over statically-bound calls)

Allow us to:

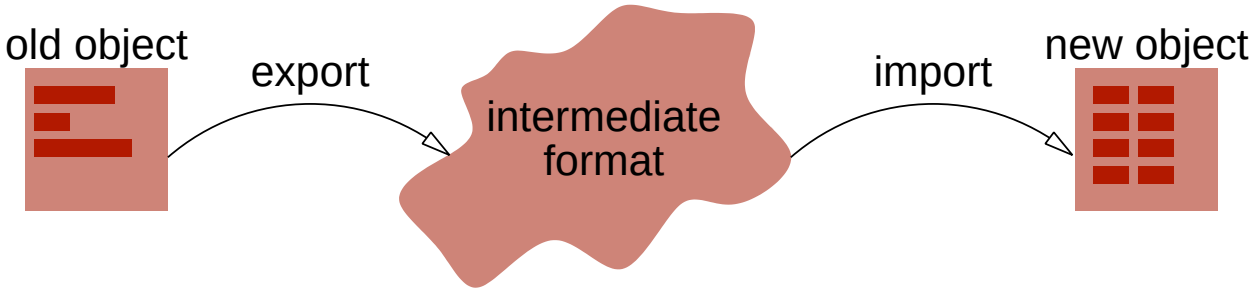
- ➜ Locate and update every instance
- ➜ Ensure future instantiations use the update



TRANSFER FUNCTIONS

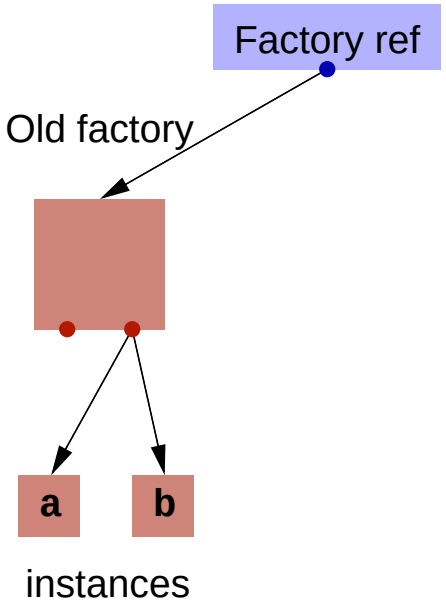
Support arbitrary changes in object representation

- ➜ Objects must provide export and import methods



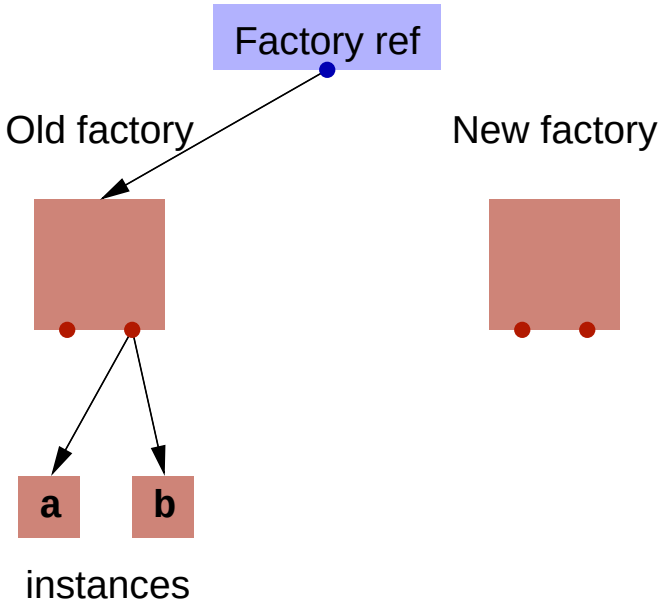
- ➜ Usually, intermediate format = old internal format
- ➜ Can avoid copying

Dynamic Update Implementation

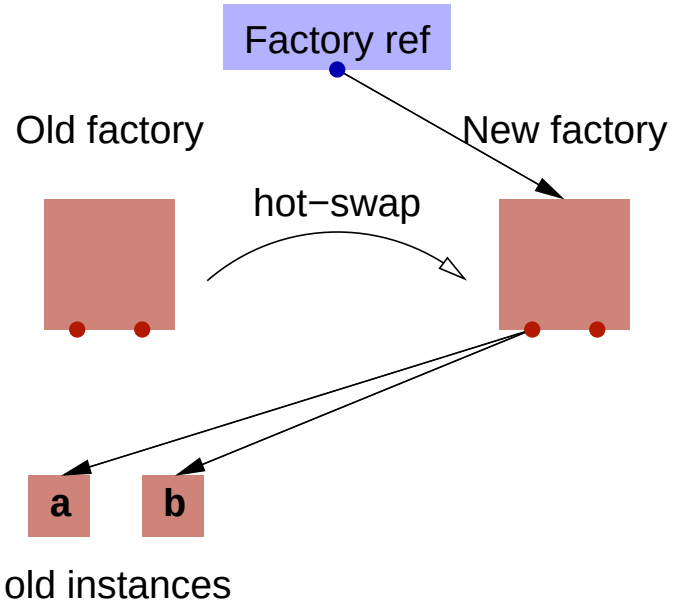


① Before update, the old factory tracks instances of a class

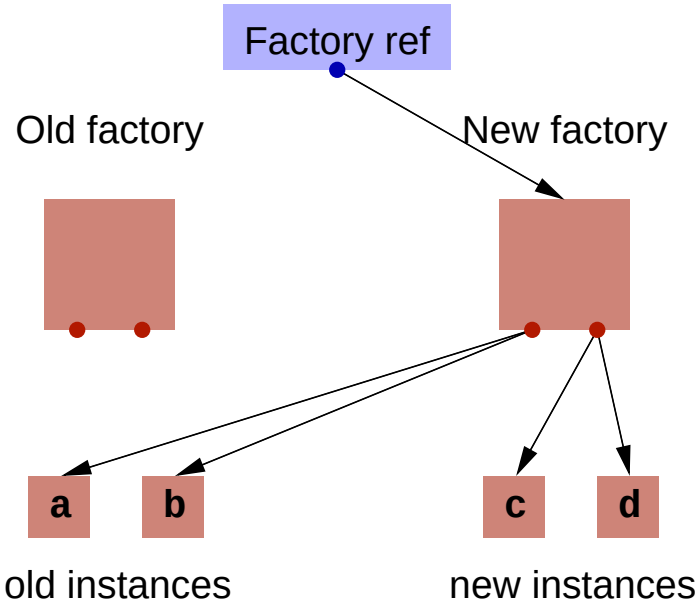
➜ Updated code compiled together with initialisation code as a loadable module



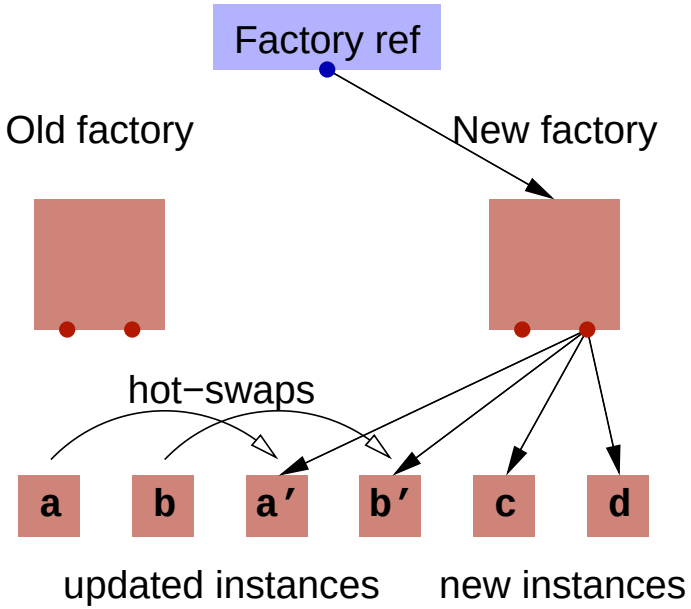
② Instantiate a new factory for the updated class



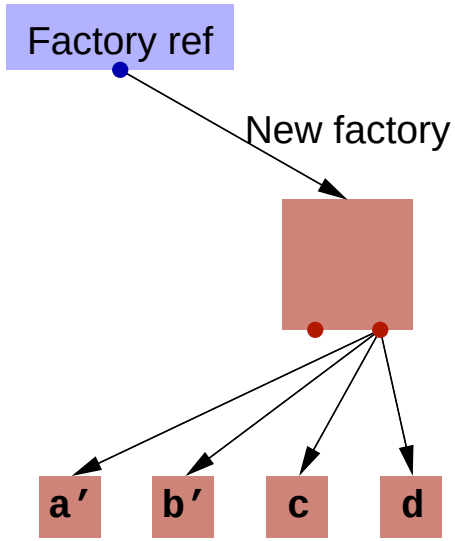
③ Hot-swap old factory with its replacement (transferring set of instances)



④ New instantiations handled by the updated factory



⑤ Hot-swap each old instance to updated replacement (in parallel on each CPU)



⑥ Destroy old factory

EXPERIENCES

Example updates applied (actual developer changes):

- ➜ New kernel API for partitioned memory region
  - ✓ Adds new functionality to kernel
- ➜ Fix for memory allocator race condition
  - ✓ Adds lock and locking code to crucial kernel object
- ➜ File cache manager optimisation for *unmapPage*
  - ✓ Affects each open file instance, demonstrating factory mechanism

RESULTS

Microbenchmarks for creation cost using factories:

object	*overhead*
dummy	12%
file	5.6%
process	0.73%

RESULTS

Microbenchmarks for creation cost using factories:

object	*overhead*
dummy	12%
file	5.6%
process	0.73%

Other results:

- ➜ No noticeable performance degradation on system throughput, as measured by a version of SPEC SDET

RESULTS

Microbenchmarks for creation cost using factories:

object	*overhead*
dummy	12%
file	5.6%
process	0.73%

Other results:

- ➜ No noticeable performance degradation on system throughput, as measured by a version of SPEC SDET
- ➜ Update time significant, 20ms to update 100 dummy objects
  - ➜ Implementation not yet optimised
  - ✓ System still responsive & usable during update

Open Issues & Future Work

- Performance: hot-swapping not optimised for concurrent swaps
- Updates affecting multiple address spaces: requires central service to track factories and coordinate updates
- Timeliness of updates: plan to add completion notification, and investigate lazy updating
- Changes to interfaces: would require mechanism to locate potential callers of an object, and atomic hot-swaps of multiple objects

Related Work

- ➜ Existing dynamic update systems for application software
  - ✗ rely on run-time language or system support
- ➜ Commercial OS features, Solaris Live Upgrade
  - ✓ Changes can be tested without affecting system
  - ✗ Requires reboot
- ➜ Restart services in component- or microkernel-based system
  - ✗ Still suffers downtime
- ➜ Modify extensible operating systems
  - ✗ Limited to those parts with hooks for extensibility
  - ✗ Not applicable to commodity operating systems
- ➜ Nooks supports recovery and restart of drivers
  - ✗ Not as applicable to other parts of OS
  - ✗ Shadows would have high performance overhead

CONCLUSION

Our system can apply updates affecting code and data to a running kernel

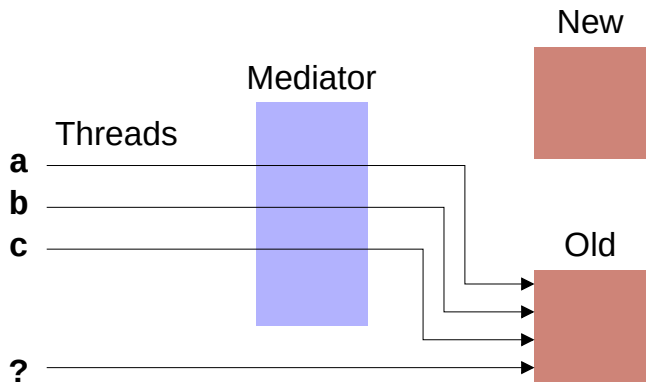
- ✓ Negligible performance impact
- ✓ Applicable to a mainstream OS

THE END

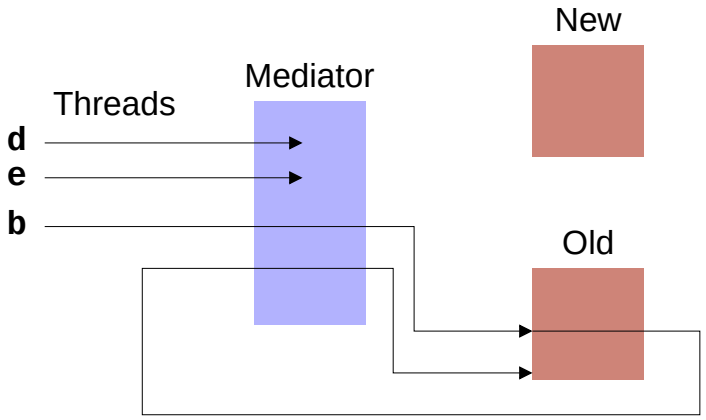
Questions?

HOT-SWAPPING IMPLEMENTATION

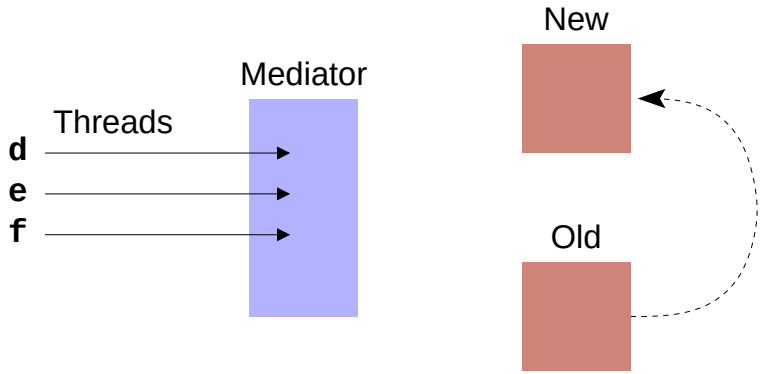
① Prior to hot-swap, an object is invoked by multiple threads



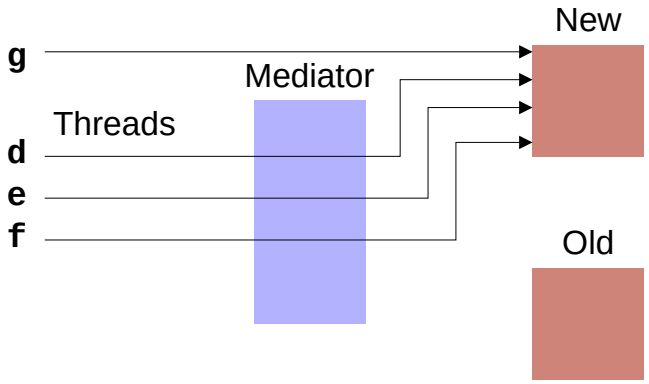
② A mediator object is interposed and tracks calls to the object (via object translation table)



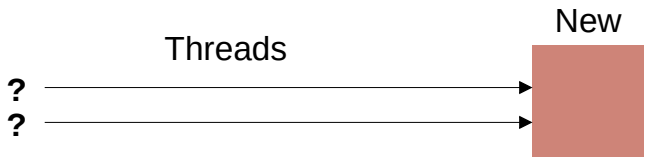
③ Once existing calls have completed, the mediator blocks new calls and waits for tracked calls to complete



④ Object is quiescent, state is transferred



⑤ Mediator updates the object reference and forwards blocked calls



⑥ Mediator and old object are destroyed