Module Hot-Swapping for Dynamic Update and Reconfiguration in K42

Andrew Baumann
University of NSW & National ICT Australia

Jeremy Kerr
IBM Linux Technology Center

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

Abstract

K42 is an open-source research OS for 64-bit multiprocessor systems, focusing on the PowerPC® architecture. It uses an object-oriented design to achieve good performance, scalability, customisability, and maintainability. K42 supports the Linux® API and ABI allowing it to use unmodified Linux applications and libraries, and can be deployed by running on an existing installation of Linux. The development process of K42 is intended to share concepts with Linux; many ideas have been transferred between the two projects.

K42 implements each system resource (such as an open file or process) as a set of unique object instances, and supports hot-swapping, which allows these objects to be changed on-the-fly. This enables dynamic reconfiguration and adaptability of the system, and when combined with a kernel module loader, supports dynamic update and hot patching of the OS. Examples of this might include dynamically adapting to file access patterns, or replacing insecure code in the network stack without downtime.

In this paper we will introduce K42, discuss its hot-swapping capability, and suggest possible ways in which this technology could benefit Linux.

1 Introduction

In a kernel supporting loadable modules, such as Linux, kernel code can be changed without a reboot, but only in restricted circumstances. Code can be added, but it can be removed only when the entire kernel is inactive, and only when no other parts of the kernel hold references to that code. For example, to update a file system module requires unmounting all file systems using that module, which in turn means killing all processes that have files open on those file systems. If the root file system happens to be an *ext3* file system, and a bug needs to be fixed in the *ext3* code, a kernel with loadable modules is of no use, because the module can't be unloaded.

Hot-swapping and dynamic update features avoid such problems, and offer many exciting benefits for an operating system. Kernel code can be changed on the fly, adapting to user behaviour and access patterns. Patches can be applied dynamically, without the need to reboot or even affect service to applications.

K42 is a research operating system supporting hot-swapping, and in recent work [5] we have developed dynamic update features using hot-swapping. Although the implementation of these features depends partly on K42's unique object-oriented structure, we believe they could be applied to a commodity operating system such as Linux, with acceptable levels of performance.

In the rest of this paper, Section 2 introduces the K42 operating system, Section 3 covers its hot-swapping feature, Section 4 details our implementation of dynamic update in K42, Section 5 discusses the application of these features to Linux, Section 6 covers some related work, and Section 7 concludes.

2 K42 background

K42 [8] is an open-source general-purpose operating system being developed at IBM® Research for cache-coherent multiprocessors. It is designed to be highly

Appears in *Proceedings of Linux.conf.au*, Canberra, Australia, April 2005.

scalable, from small SMP machines that we expect to become ubiquitous, to large-scale NUMA systems. It presently runs only on 64-bit PowerPC hardware (such as IBM pSeries® and Apple® G5), but is designed to be portable; in the past it has also been used on MIPS systems. K42 supports the Linux API and ABI for user-level applications [1], and uses Linux code in the kernel for its device drivers and network stack [10].

Although it implements the Linux API, K42 is structured very differently from a standard Linux system. K42 implements much of what is traditionally considered kernel functionality at user-level, in libraries and server processes. For example, K42's fundamental IO model is event-driven; all blocking operations are emulated in library code [1]. Threads and the scheduler are also implemented at user level [9].

2.1 Object model

K42 is implemented in C++, and uses a modular object-oriented design to achieve multiprocessor scalability, enhance customisability, and support novel features such as hot-swapping and dynamic update. Unlike modules in Linux, K42 objects are fine grained, and encapsulate all their data behind function interfaces.

Each resource (for example, virtual memory region, network connection, open file, or process) is managed by a different set of object instances [3]. Each object encapsulates the meta-data necessary to manage the resource as well as the locks necessary to manipulate the meta-data, thus avoiding global locks, data structures, and policies. This also enables adaptability, because different resources can be managed by different implementations.

K42 uses clustered objects [2], a mechanism that enables a given object to control its own distribution across processors. Each clustered object is invoked using indirection through an object translation table (OTT). The OTT is stored in processor-specific memory, so the same object reference can transparently invoke different object *representatives* on different CPUs. The same OTT mechanism is used to implement K42's hot-swapping feature, as discussed in the next section.

2.2 Quiescence detection

K42 detects quiescent states using a mechanism similar to read copy update (RCU) in Linux [11]. K42's quiescence detection mechanism makes use of the fact that each system request is serviced by a new kernel thread, and that all kernel threads are short-lived and non-blocking. This differs from RCU, which operates only around critical sections such as locks, because Linux has long-lived kernel threads.

Each thread in K42 belongs to a certain epoch, or *generation*, which was the active generation when it was created. A count is maintained of the number of live threads in each generation, and by advancing the generation and waiting for the previous generations' counters to reach zero, it is possible to determine when all threads that existed on a processor at a specific instance in time have terminated [7].

The generation count mechanism is used to support deferred object deletion, and enable hot-swapping and dynamic update, as detailed in the next section.

3 Hot-swapping

Using the object translation table, hot-swapping [2, 13] was implemented in K42. Hot-swapping allows an object instance to be transparently switched to another implementation while the system is running.

A hot-swap operation, as shown in Figure 1, consists of the following six phases:

1. Prior to an update, threads are invoking methods in an object via the OTT. In this case the system incurs no performance overhead, besides the cost of pointer indirection through the OTT.

2. A *mediator* object is interposed, by setting the OTT entry for the object to point to it. The mediator then tracks all incoming calls, forwarding them to the old object, and advances the thread generation, waiting for the previous thread generation counters to all reach zero.

3. Once previous thread generation counters are zero, the mediator is guaranteed that all threads executing in the object are being tracked by it. It now begins blocking calls by new threads into the object, while waiting for all the calls it has tracked to complete; to avoid deadlock, recursive calls are not blocked.

4. Once all forwarded calls complete, the object is *quiescent*, and can be safely swapped. State transfer functions are used to import its state into the new object.

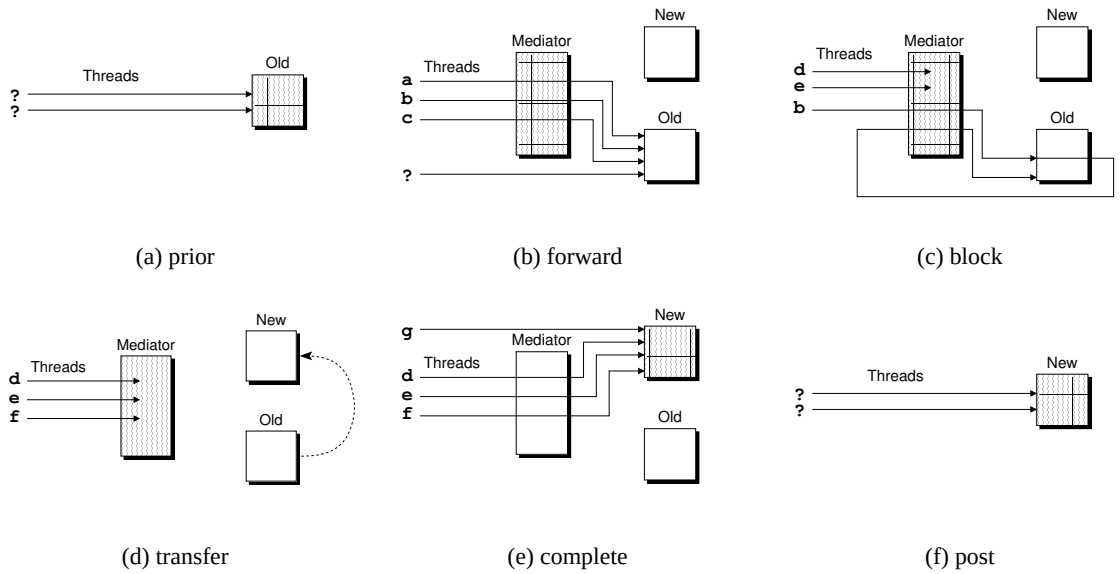

Figure 1: Phases in an object hot-swap: (a) prior to the swap threads invoke an object directly; (b) a mediator object is interposed, forwarding new calls to the object; (c) once all calls are tracked, the mediator blocks incoming calls and waits for the existing ones to finish; (d) when the object is quiescent, state is transferred to its replacement; (e) the mediator forwards blocked calls to the new object; (f) the old object and mediator are destroyed.

5. The mediator sets the OTT entry to point to the new object (allowing new calls to directly invoke it), and forwards the calls it had previously blocked to the new object.

6. The mediator destroys itself and the old object, and the hot-swap is complete.

4 Dynamic update

Hot-swapping changes a single specific object instance, and was designed to enable adaptability. More recently, we have extended hot-swapping to support dynamic update [4, 5], enabling operating system updates (such as security fixes or performance improvements) to be applied on-the-fly.

In designing dynamic update, we identified the following key requirements for a dynamically updatable operating system [5]:

Updatable unit: The system must have a unit of update, with a well-defined and respected interface. This enables the unit to be changed during an update operation without affecting the rest of the system. Many systems are designed in a modular fashion, leading to natural updatable units.

In K42 the updatable unit is the object instance. K42's coding style enforces encapsulation of data within objects, allowing data structures to be changed by a dynamic update.

Safe point: Dynamic updates should not occur while any affected code or data is being accessed, so support is required to achieve and detect a safe point. During a hot-swap, K42 blocks new invocations of an object being updated, and then uses the generation-count mechanism to detect quiescence.

State tracking: For a dynamic update system to support changes to data structures, it must be able to locate and convert all such structures. In K42, we have implemented factory objects [6] to perform this task. Factories are responsible for creating, tracking, and destroying every object of a specific class, and they form the basis of our dynamic update implementation.

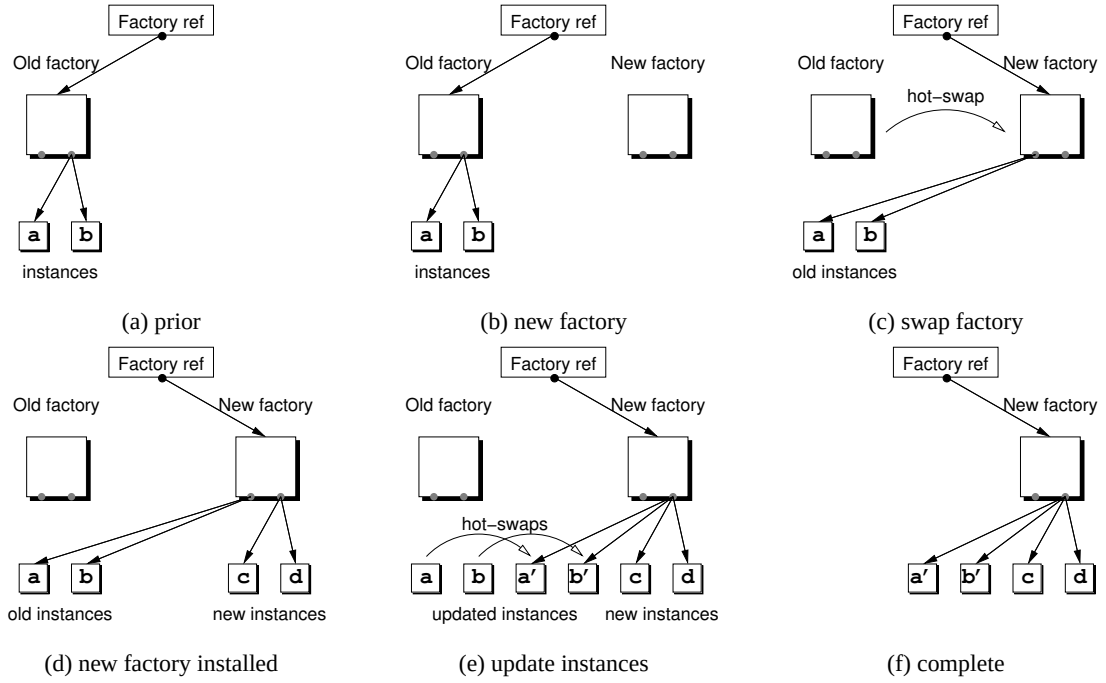

Figure 2: Phases in the dynamic update of a multiple-instance object: (a) prior to update the old factory is maintaining instances of a class; (b) instantiate a new factory for the updated class; (c) hot-swap the factory with its replacement (transferring the set of managed instances); (d) after the hot-swap, new instantiations are handled by the updated factory code (thus creating objects of the new type); (e) update old instances by hot-swapping each to an updated replacement; (f) the update is complete.

State transfer: When an update alters data structures, or when an updated unit maintains internal state, the state must be transferred to the new unit. In K42, a *transfer negotiation protocol* is used to allow selection of a common intermediate format. Then, state export and import functions implemented by the object developer are invoked to perform the conversion.

Redirection of invocations: After the update occurs, all future requests affecting the old unit should be redirected. In K42, indirections are redirected after a hot-swap by changing the object pointer in the OTT.

Version management: In order to package and apply an update, and in order to debug and understand the running system, it it necessary to know what code is actually executing. If an update depends on another update having previously been applied, then support is required to be able to verify this.

We currently use only a very simple version scheme in K42. Each factory object carries a version number, updated factories have increasing version numbers, and before any update proceeds the version numbers are checked for consistency.

A dynamic update in K42, as shown in Figure 2, consists of the following phases:

1. A factory for the updated class is instantiated.
2. The old factory object is hot-swapped with the new factory object. During the state transfer phase of the hot-swap, the new factory imports the old factory's set of live object instances.
3. After the hot-swap, all new object instantiations go to the updated class.
4. The new factory traverses the set of instances it received. For each, it creates an instance of the updated object, and initiates a hot-swap between the old and the new instances.
5. Finally, the update is complete and the old factory is destroyed.

Dynamic update works, and has been used to apply actual modifications by K42 developers to a running kernel [5], it also has very low performance impact.

5 Applying these ideas to Linux

At present, replacing elements of the Linux kernel is only possible if those components have been compiled as a module. Updating a Linux kernel module requires full removal and re-insertion, so any resources that the module provides will be unavailable during the update. Furthermore, releasing these resources may not be possible (for example, a file system servicing currently open files). We aim to investigate the feasibility of implementing a K42-style dynamic update mechanism in Linux to avoid these problems.

This section will address the current suitability of dynamic update in Linux, and present some ideas for future work to enable us to proceed.

5.1 Approach

Although Linux supports loadable kernel modules, it is not structured in the same highly modular fashion as K42. Modules may reach into each other, and are not required to encapsulate their state information. Furthermore, unlike K42's OTT, there is no standard abstraction for invoking code in a module; once it has a function pointer to a module's code, another module may call that function directly. Rather than attempt the mammoth task of rewriting Linux to have clearer module boundaries with a standard module invocation mechanism, we will concentrate on adding dynamic update features to a specific part of the kernel. If it proves successful, future work may extend dynamic update to other areas, or attempt such a restructure.

We have chosen to start with a file system driver module, for the following reasons:

- All of the module's symbols are statically defined, and not directly accessible from outside the module itself.
- Code in the module is always invoked through a table of function pointers, such as the *file_operations* or *inode_operations* structures.
- File systems are commonly used as loadable modules, and offer a compelling example.

In the case of a file system, the structures of function pointers can be used in the same way as K42's OTT. By overwriting the function pointers, we can interpose or redirect calls to the module. Fortunately, many other classes of modules use tables of function pointers in the same way, allowing our design to extend beyond file systems.

5.2 Requirements

Our initial design addressed the previously outlined requirements for dynamic update as follows:

Updatable unit: As discussed, the updatable unit is a file system driver module, although we hope to extend this to other module classes.

Safe point: In order to update a module we need to ensure that there are no kernel threads that are executing within any of the the module's functions. In current Linux kernel code, each module has a reference count, incremented when the module is used by another part of the kernel (for example, when a file is opened, the file system's usage count is incremented). However, we have no indication of when the module's code is executed—it is possible for a module to have a non-zero reference count, but not have any kernel threads executing within the module. If we only allow dynamic updates to be performed when the reference count is zero, then we fall back to the existing case of requiring a module to be unused before updating it.

Using the table of function pointers, we can interpose a mediator as in K42; however, RCU in Linux does not provide the same thread-based quiescence detection mechanism, so we require an alternative. Two possibilities include checking the stack of running kernel threads to determine if any are executing inside the module, or adding atomic counters at function entry and exit points.

Checking the stack of kernel threads would involve walking through the stack frames of every kernel thread in the system to determine which, if any, were executing within the affected module. The problem with this approach is that, although it does not affect the base case performance, it would significantly delay the critical phase of an update in which new calls are blocked. It would also scale poorly as more kernel threads are added.

We believe that adding a module use count, that is incremented and decremented on entry to and exit from

all externally-callable functions, is the best option. This will add a small cost to module invocations, but in return will enable us to efficiently detect quiescence by waiting for the counter to reach zero.

State tracking: State tracking is required when data structures with multiple instances, such as a file system's inode structures, are to be changed during a dynamic update. To track such state a module would be required to incorporate a factory; for example, by adding each allocated inode to a list, if it doesn't already do so.

State transfer: We intend to use a similar scheme to the state transfer functions in K42. Each module supporting dynamic update will be required to implement state export and import functions—either marshalling state when a module is being replaced, or unmarshalling state when a module is replacing another.

In the case of a file system, data transfer could mean flushing as much state as possible (such as free blocks) to the disk cache, and placing the rest in a standard structure that can be imported by all implementations of the same file system.

Redirection of invocations: In order to redirect the module's function calls, we rely on the module exporting its interface in the form of structures containing function pointers. At update time, these structures are patched to refer first to the mediator object, and then later to the new module.

Because the structures of function pointers used differ between module classes, we will initially only support file system modules.

Version management: Although it is not required for an initial prototype implementation, we could use a similar version numbering scheme to the simple one developed for K42.

5.3 Problems

Although we hope to extend our design beyond file systems, not all kernel modules are well suited to the updatable unit structure. Possible difficulties include:

- Modules containing code that must run in interrupt context cannot be blocked by the interposing object for long periods of time.
- Modules that do not export a clean interface to perform the update, or have entry points that are not available to be patched later.

Some of these problems could be solved by making Linux more modular, and enforcing isolation between modules.

6 Related work

DKM [12] also supports dynamic kernel modification for Linux, but rather than using dynamic updates to enable adaptation and on-line upgrade, its goal is to enable faster development and debugging of the kernel, and as a result it has a different approach. DKM supports inserting tracepoints, nullifying functions, or replacing functions in the kernel. It does so by rewriting the first few instructions of the affected function to jump to modified code. DKM is more flexible than our system, in that it can potentially change any function in the kernel, but because it does not work at a module level, and does not enforce quiescence during a modification, it cannot support changes to data structures.

Nooks [14, 15] supports recoverable device drivers on Linux, and could be extended to support updatable drivers. In this system, shadow drivers monitor all calls made into and out of a device driver, and reconstruct a driver's state after a crash using the driver's public API. Only one shadow driver must be implemented for each device class (such as network, block, or sound devices), rather than for each driver. The main drawback of this approach is that there is a continual runtime performance cost imposed by the use of shadows, unlike conversion functions, which are only invoked at update time. Its advantage is that drivers don't need to be modified.

7 Conclusion

In this paper, we have outlined the implementation of the hot-swapping and dynamic update features in K42. We have also discussed how this technology could be transferred to Linux; we are presently working on a prototype implementation of these ideas within the Linux kernel.

Acknowledgements

We wish to thank the K42 team at IBM Research for their support of this effort, and Chris Yeoh for pestering us to submit this work to Linux.conf.au. Rusty Russell and Stephen Rothwell helped us with the Linux details.

This work was partially supported by DARPA under contract NBCH30390004, and by a hardware grant from IBM. National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

Availability

K42 is released as open source and is available from a public CVS repository; for details refer to the K42 web site at http://www.research.ibm.com/K42/.

Legal statement

References

[1] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the 2003 USENIX Technical Conference, FREENIX Track*, pages 323–336, San Antonio, TX, USA, June 2003.

[2] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems*, pages 3–8, Charleston, SC, USA, November 2002.

[3] Marc Auslander, Hubertus Franke, Ben Gamsa, Orran Krieger, and Michael Stumm. Customization lite. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[4] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving operating system availability with dynamic update. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, USA, October 2004.

[5] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Technical Conference*, Anaheim, CA, USA, April 2005.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[7] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, New Orleans, LA, USA, February 1999. USENIX.

[8] IBM K42 Team. *K42 Overview*, August 2002. Available from http://www.research.ibm.com/K42/.

[9] IBM K42 Team. *Scheduling in K42*, August 2002. Available from http://www.research.ibm.com/K42/.

[10] IBM K42 Team. *Utilizing Linux Kernel Components in K42*, August 2002. Available from http://www.research.ibm.com/K42/.

[11] Paul E. McKenney, Dipankar Sarma, Andrea Arcangelli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[12] Ronald G. Minnich. A dynamic kernel modifier for Linux. In *Proceedings of the LACSI Symposium*, September 2002.

[13] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, pages 141–154, San Antonio, TX, USA, 2003.

[14] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

[15] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on OS Principles*, Bolton Landing (Lake George), New York, USA, October 2003.