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

Abstract

Dynamic update is a mechanism that allows software updates and patches to be applied to a running system without loss of service or down-time. Operating systems would benefit from dynamic update, but place unique demands on any implementation of such features. These demands stem from the event-driven nature of operating systems, from their restricted run-time execution environment, and from their role in simultaneously servicing multiple clients.

We have implemented a dynamic update mechanism in the K42 research operating system, and tested it using previous modifications of the system by kernel developers. Our system supports updates changing both kernel code and data structures. In this paper we identify requirements needed to provide a dynamically updatable operating system, describe our implementation, and present our experiences in designing and using the dynamic update mechanism. We also discuss its applicability to other operating systems.

1 Introduction

As computing infrastructure becomes more widespread, there has been an increasing number of patches for functionality, performance, and security reasons. To take effect, these patches traditionally require either restarting system services, or often rebooting the machine. This results in downtime. Sometimes this downtime can be scheduled, if for example the patch adds a feature, improves performance, etc. However, in some situations, such as applying a security patch, delaying the update is not desirable. Users and system administrators are forced to trade off the increased vulnerability of a security flaw against the cost of unplanned downtime.

First published in *Proceedings of USENIX '05: General Track*, Anaheim, CA, USA, April 2005.

Dynamic update [26] is used to avoid such downtime. It involves on-the-fly application of software updates to a running system without loss of service. The increased unplanned down-time of computing infrastructure to apply updates, combined with the demand for continuous availability, provides strong motivation to investigate dynamic update techniques for operating systems.

In addition to the above mentioned impact on availability, dynamically updatable systems have other benefits. Such systems provide a good prototyping environment. They allow, for example, a new page replacement, file system, or network policy to be tested without rebooting. Further, in more mature systems such as mainframes, some user constraints prevent the system from ever being shutdown. In such an environment, users can only get new functionality into the system by performing a dynamic update.

An operating system is a unique environment with special constraints, and additional challenges must be solved to provide dynamic update functionality. We have addressed these challenges in the implementation of a dynamic update mechanism for K42, an object-oriented research operating system supporting hot-swapping. The focus of this paper is on the implementation and mechanisms needed to provide dynamic update. This work builds on previously reported work [6, 28], and on other K42 features. Some of the requisite characteristics we identify for dynamic update exist in other systems or have recently been incorporated [22], while others require additional support. Where appropriate, we point out the generality of our techniques to other operating systems, as well as what infrastructure would be required to take full advantage of dynamic update techniques. In addition to describing our implementation, we describe our experiences applying dynamic update in K42, using three motivating examples taken from changes made by K42 kernel developers.

The rest of this paper is organised as follows: Section 2 discusses the system requirements for supporting

dynamic update, Section 3 describes our implementation of dynamic update in K42, and Section 4 discusses how the same functionality might be implemented in other operating systems. Next, Section 5 describes our experiences applying dynamic update to K42 using three motivating examples, Section 6 discusses the limitations of our implementation and our plans for future work, Section 7 compares related work, and Section 8 concludes.

2 Requirements for dynamic update

There are several fundamental requirements in providing a dynamic update capability. Here we identify them, in Section 3.2 we describe how we satisfy them in K42, and then in Section 4 we generalise to other operating systems.

2.1 Classification of updates

At a minimum, dynamic update needs to support changes to the code of a system, however there are varying levels of support possible for updates which also affect data. We classify dynamic updates in this way:

1. Updates that only affect code, where any data structures remain unchanged across the update. This is easier to implement, but imposes significant limitations on what updates may be applied dynamically.
2. Updates that affect both code and global, single-instance, data. Examples of this might include changes to the Linux kernel's unified page cache structure, or to K42's kernel memory allocator.
3. Updates that affect multiple-instance data structures, such as the data associated with an open socket in Linux, or an open file.

2.2 Requirements

Having classified the possible updates, we now introduce a set of fundamental requirements for dynamic update.

Updatable unit: In order to update a system, it is necessary to be able to define an updatable unit. Depending on the class of update supported, and the implementation of the system, a unit may consist of a code module, or of both code and encapsulated data. In both cases, there must be a clearly defined interface to the unit. Furthermore, external code should invoke the unit in a well-defined manner, and should not arbitrarily access code or data of that unit.

While creating updatable units is easier with support from languages such as C++, it is still possible without such support. Primarily, providing updatable units means designing with good modularity and obeying module boundaries. The structure of the system dictates what is feasible.

Safe point: Dynamic updates should not occur while any affected code or data is being accessed. Doing so could cause undefined behaviour. It is therefore important to determine when an update may safely be applied. In general however, this is undecidable [15]. Thus, system support is required to achieve and detect a safe point. Potential solutions involve requiring the system to be programmed with explicit update points, or blocking accesses to a unit, and detecting when it becomes idle, or *quiescent*.

An operating system is fundamentally event-driven, responding to application requests and hardware events, unlike most applications, which are structured as one or more threads of execution. As discussed later, this event-based model can be used to detect when an updatable unit of the system has reached a safe point. Additional techniques can be employed to handle blocking I/O events or long running daemon threads.

State tracking: For a dynamic update system to support changes to data structures, it must be able to locate and convert all such structures. This requires identifying and managing all instances of state maintained by a unit in a uniform fashion, functionality usually provided in software systems using the factory design pattern [12]. Note that the first two classes of update, dynamic update to code and dynamic update to single-instance data, are still possible without factories, but it is not possible to support dynamic update affecting multiple-instance data without some kind of state tracking mechanism.

State transfer: When an update is applied affecting data structures, or when an updated unit maintains internal state, the state must be transferred, so that the updated unit can continue transparently from the unit it replaced. The state transfer mechanism performs this task, and is how changes to data structures can be supported.

Redirection of invocations: After the update occurs, all future requests affecting the old unit should be redirected. This includes invocations of code in the unit. Furthermore, in a system supporting multiple-instance data structures, creation of new data structures of the affected type should produce the updated data structure.

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. Furthermore, if updates are from multiple sources, the versioning may not be linear, causing the interdependencies between updates to become complex and difficult to track.

The level of support required for version management is affected by the complexity of update interdependencies, but at a minimum it should be possible to track a version number for each update present in the system, and for these version numbers to be checked before an update is applied.

3 Dynamic update in K42

We now describe our implementation of dynamic update in K42. As noted previously, some of the techniques used in the implementation are specific to K42, but other operating systems are becoming more amenable to dynamic update, as discussed in the next section.

3.1 K42

The K42 project is developing a new scalable open-source research operating system incorporating innovative mechanisms and policies, and modern programming technologies. It runs on 64-bit cache-coherent PowerPC systems, and supports the Linux API and ABI. It uses a modular object-oriented design to achieve multiprocessor scalability, enhance customisability, and enable rapid prototyping of experimental features (such as dynamic update).

Object-oriented technology has been used throughout the system. Each resource (for example, virtual memory region, network connection, open file, or process) is managed by a different set of object instances [5]. 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. The object-oriented nature enables adaptability, because different resources can be managed by different implementations. For example, each running process in the system is represented by an in-kernel instance of the *Process* object (analogous to the process control block structure present in other operating systems). Presently two implementations of the *Process* interface exist, *ProcessReplicated*, the default, and *ProcessShared*, which is optimised for the case when a process exists on only a single CPU [2]. The K42 kernel defaults to creating replicated processes, but allows for a combination of replicated and shared processes.

K42 uses clustered objects [4], a mechanism that enables a given object to control its own distribution across processors. Using the object translation table facility provided by clustered objects, hot-swapping [4, 28] was implemented in K42. Hot-swapping allows an object instance to be transparently switched to another implementation while the system is running, and forms the basis of our dynamic update implementation.

3.2 Support for dynamic update

Requirements

In Section 2.2, we identified several requirements for dynamic update of an operating system. In K42, these requirements are addressed by our implementation of the dynamic update mechanism, as follows:

Updatable unit: A good choice for the dynamically updatable unit in K42 is the same as for hot-swapping, namely the object instance. K42 is structured as a set of objects, and the coding style used enforces encapsulation of data within objects. Each object's interface is declared in a virtual base class, allowing clients of an object to use any implementation, and for the implementation to be changed transparently by hot-swapping.

Safe point: K42 detects quiescent states using a mechanism similar to read copy update (RCU) in Linux [22, 23]. This technique 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.

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 [13].

The implementation blocks new invocations of an object being updated, and then uses the generation-count mechanism to detect quiescence [28].

State tracking: state-tracking is provided by factory objects, which are described in detail in Section 3.3.

State transfer: Once the object being swapped is quiescent, the update framework invokes a state transfer mechanism which transfers state from the old object to the new object, using a *transfer negotiation protocol* to allow the negotiation of a common intermediate format that both objects support [28]. Object developers must implement data conversion functions to and from common intermediate formats.

This generalised technique was developed to support hot-swaps between arbitrary implementations of an object. In the case of dynamic update, usually the replacement object is merely a slightly modified version of the original object, with similar state information, so the conversion functions perform either a direct copy, or a copy with slight modifications.

In cases where a lot of state is maintained, or when many object instances must be updated, a copy is an unnecessary expense, because the updated object is deleted immediately afterwards. For example, the process object maintains a series of structures which describe the process' address space layout. To avoid the cost of deep-copying and then discarding these structures, the data transfer functions involved simply copy the pointer to the structure and set a flag in the old object. When the object is destroyed it checks this flag and, if it is set, does not attempt destruction of the transferred data structures. Effectively ownership of the structure is transferred to the new object instance. This only works in cases where the new object uses the same internal data format as the old object. This is true in many dynamic update situations. In cases where this is not true, the negotiation protocol ensures that a different transfer function is used.

Redirection of invocations: K42 uses a per-address-space *object translation table*. Each object has an entry in the table, and all object invocations are made through this reference. In the process of performing a dynamic update, the translation table entries for an object are updated to point to the new instance, which causes future calls from clients to transparently invoke the new code. The object translation table was originally introduced into K42 to support the clustered object multiprocessor scalability mechanism [13], and we have been able to utilise it to implement hot-swapping and thus dynamic update.

When an object that has multiple instances is updated, we must also redirect creations of that type. This redirection is provided by the factory mechanism, described in Section 3.3.

Version management: We have implemented a simple versioning scheme for dynamic updates in K42. Each factory object carries a version number, and before an update proceeds these version numbers are checked. Further details follow in Section 3.3.

Hot-swapping

Because hot-swapping forms the basis of dynamic update, we outline its implementation here. Further details are available in previous papers [4, 28].

As we have mentioned, the object translation table adds an extra level of indirection on all object invocations. This indirection enables an interposition mechanism whereby an object's entry in the object translation table is modified, causing all accesses to that object to transparently invoke a different interposer object. The interposer can then choose to pass the call along to the original object. This mechanism is used by the hot swapping and dynamic update implementations.

Hot-swapping operates by interposing a mediator object in front of the object to be hot-swapped. The mediator passes through several phases, first tracking incoming calls until it knows (through the generation-count mechanism) that all calls are being tracked, then suspending further calls until the existing tracked calls complete. At this point the object is quiescent. The mediator then performs state transfer format negotiation, followed by the state transfer between the old and the new object instances. Finally, it updates the object translation table reference to the new object, and forwards the blocked calls.

3.3 Dynamic update implementation

Module loader

To perform updates, the code for the updated object must be present. The normal process for adding an object to K42 was to recompile the kernel, incorporating the new object, and then reboot the system. This is insufficient for dynamic update, so we have developed a *kernel module loader* that is able to load the necessary code for an updated object into a running kernel or system server.

A K42 kernel module is a relocatable ELF file with unresolved references to standard kernel symbols and library routines (such as *err_printf*, the console output routine). Our module loader consists of a special object in the kernel that allocates pinned memory in the kernel's text area, and a trusted user-space program that has access to the kernel's symbol table. This program uses that symbol table to resolve the undefined symbols in the module, and load it into the special region of memory provided by the kernel object. It then instructs the kernel to execute the module's initialisation code.

Our module loader operates similarly to that used in Linux [8], but is simpler. Linux must maintain a dynamic symbol table and support interdependencies between modules, we avoid this because all objects are invoked indirectly through the object translation tables. A module can (and to be useful should) contain code that is called by the existing kernel without requiring its symbols to be visible. Its initialisation code simply instantiates replacement objects and performs hot-swap operations to invoke the code in those object instances. Our

module loader performs the relocations and symbol table management at user-level, leaving only the space allocator object in the kernel.

Factory mechanism

Hot-swapping allows us to update the code and data of a single specific object instance. However, K42 is structured such that each instance of a resource is managed by a different instance of an object. To dynamically update a kernel object, the infrastructure must be able to both locate and hot-swap all instances of that object, and cause any new instantiations to use the updated object code. Note that, as we have mentioned, this problem is not unique to K42; to support dynamic updates affecting data structures requires a mechanism to track all instances of those data structures and update them.

Previously in K42, object instances were tracked in a class-specific manner, and objects were usually created through calls to statically-bound methods. For example, to create an instance of the *ProcessReplicated* object (the implementation used by default for *Process* objects), the call used was:

```
ProcessReplicated::Create(
  ProcessRef &out, HATRef h, PMRef pm,
  ProcessRef creator, const char *name);
```

This leads to problems for dynamic update, because the *Create* call is bound at compile-time, and cannot easily be redirected to an updated implementation of the *ProcessReplicated* object, and also because we rely on the caller of this method to track the newly created instance.

To track object instances and control object instantiations, we used the factory design pattern [12]. In this design pattern, the factory method is an abstraction for creating object instances. In K42, factories also track instances that they have created, and are themselves objects. Each factory object provides an interface for creating and destroying objects of one particular class, and maintains the set of objects that it has created.

The majority of the factory implementation is factored out using inheritance and preprocessor macros, so that adding factory support to a class is relatively simple. Using our previous example, after adding the factory, the creation call changed to:

```
DREF_FACTORY_DEFAULT(ProcessReplicated)
  ->create(...);
```

where `(...)` represents the arguments as before. The macro above hides some implementation details, whereby the default factory for a class is referenced using a static member; it expands to the following:

```
(*ProcessReplicated::Factory::factoryRef)
  ->create(...);
```

Using a factory reference allows us to hot-swap the factory itself, which is used in our implementation of dynamic update.

To provide rudimentary support for configuration management, factories carry a version number identifying the specific implementation of the factory and its type. The factories in the base system all carry version zero, and updated factories have unique non-zero version numbers. We assume a strictly linear model of update, when an update occurs the current version number of the factory is compared to the version number of the update, and if the update is not the immediately succeeding version number, the update is aborted. To support reverting updates in this scheme, we could reapply the previous version with an increased version number.

Performance and scalability influenced our implementation of the factories. For example, object instances are tracked for dynamic update in a distributed fashion using per-CPU instance lists. Moreover, we found that adding factories to K42 was a natural extension of the object model, and led to other advantages besides dynamic update. As an example, in order to choose between *ProcessReplicated* and *ProcessShared*, K42 had been using a configuration flag that was consulted by the code that creates process objects to determine which implementation to use. Using the factory model, we could remove this flag and allow the scheme to support an arbitrary number of implementations, by changing the default process factory reference to the appropriate factory object.

Steps in a dynamic update

We use factories to implement dynamic update in K42. To perform a dynamic update of a class, the code for the update is compiled along with some initialisation code into a loadable module. When the module is loaded, its initialisation code is executed. This code performs the following steps (illustrated in Figure 1):

1. A factory for the updated class is instantiated. At this point the version number of the updated factory is checked against the version number of the existing factory, if it is incorrect the update is aborted.

2. The old factory object is located using its statically bound reference, and hot-swapped to the new factory object; during this process the new factory receives the set of instances that was being maintained by the old factory.

3. Once the factory hot-swap has completed, all new object instantiations are being handled by the new updated factory, and therefore go to the updated

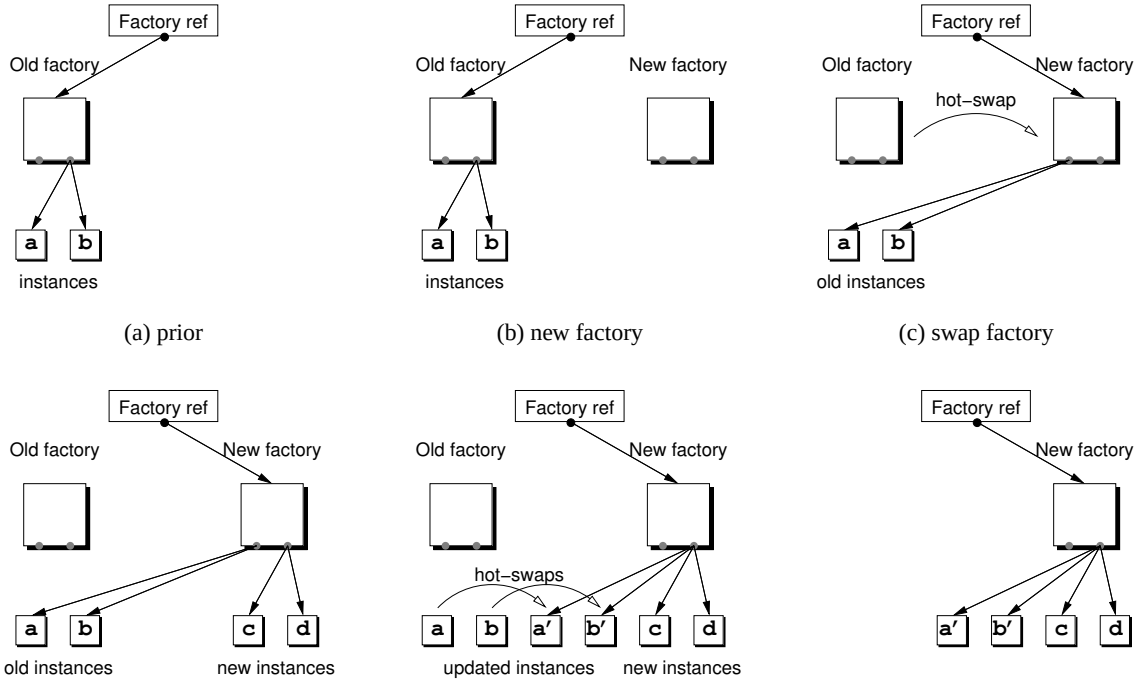

Figure 1: Phases in the dynamic update of a multiple-instance object using a factory: (a) prior to update the old factory is maintaining instances of a class; (b) instantiate a new factory for the updated class; (c) hot-swap the old factory with its new replacement (transferring the list of managed instances); (d) after the hot-swap completes, new instantiations are handled by the updated factory code (thus creating objects of the new type); (e) update old instances by traversing the list and hot-swapping each instance to an updated replacement (occurs in parallel on each CPU); (f) the update is complete.

class. However, any old instances of the object have not yet been updated.

4. To update the old instances, the new factory traverses the set of instances it received from the old factory. For each old instance it creates an instance of the updated object, and initiates a hot-swap between the old and the new instances.

   This step proceeds in parallel across all CPUs where the old factory was in use, and while the rest of the system is functioning. Because each object instance is hot-swapped individually, and because K42 encapsulates all data behind object instances, there is no requirement to block all accesses to all objects of the affected type while an update is in progress.

5. Finally, the update is complete and the old factory is destroyed.

In some special cases, for example when an update adds new objects to the system that do not replace any existing objects, or when an update affects an object with only a single instance, the full dynamic update implementation is not required. In these cases the initialisation code in the module is simpler.

4 Dynamic update in other systems

In this section we discuss the way in which a dynamic update mechanism might be provided in operating systems other than K42, focusing on Linux. Previously we identified several key requirements, these might be provided as follows:

Updatable unit: Modularity is already widely used in constructing operating systems, the virtual file system (VFS) layer [20] being one well-known example. Furthermore, modern operating systems are being constructed in an increasingly modular fashion to allow for better multiprocessor performance, and improved customisability. This trend toward modularity provides the necessary interfaces for creating updatable units. Even monolithic systems such as Linux now have support for loadable kernel modules for device drivers, file systems, networking functionality, etc. This support could be

leveraged to provide dynamic update capabilities in the same areas where kernel modules can be used.

Safe point: Linux has recently incorporated the quiescence detection mechanism known as RCU [22], which is similar to the generation count mechanism used in K42. We expect that other operating systems would also be able to add RCU-style quiescence detection, which offers other benefits, such as improved scalability.

State tracking: A state tracking mechanism, such as a factory, is needed when dynamic update supports changes to the format of multiple-instance data, to locate all instances of that data. The factory design pattern is a generally well-understood software construction technique, and we would expect that factories could be added to an existing system. For example, in the Linux kernel, modules already maintain reference count and dependency information to prevent them from being unloaded while in use [25]. If the addition of factories was required, the modules could also be made responsible for tracking instances of their own state.

State transfer: The implementation of state transfer is something fairly unique to hot-swapping and dynamic update. In a system with clearly defined updatable units, it should be straightforward to implement the equivalent of K42's transfer negotiation protocol and state transfer functions.

Redirection of invocations: Few systems include a uniform indirection layer equivalent to K42's object translation table. Many systems such as Linux do use indirection to implement device driver abstractions or the VFS layer, and these pointers could be used to implement dynamic update. However, the lack of a uniform indirection mechanism would limit the applicability of dynamic update to those specific areas of the system.

For dynamic update to multiple-instance data structures to be supported, it is desirable that each instance be individually updatable. For example, the VFS layer's use of one set of function pointers per node, rather than one set for the entire file system, allows the file system's data structures to be converted incrementally. The alternative would be to block all access to all file system nodes while they are updated, effectively halting the whole system.

Version management: Versioning is an open problem for dynamic update. If our simple model of factory version numbers proves sufficient, it can be implemented in other systems.

Beyond these requirements, the dynamic update implementation also relies on a module loader to make the code for the update available in the running system. Loadable modules are already widely used, most operating systems include a kernel module loader or equivalent functionality, so this is not a significant constraint.

5 Experiments

5.1 Performance measurements

We have performed a number of measurements to evaluate the performance penalty imposed by our dynamic update mechanism. All the experiments were conducted on an IBM pSeries 630 Model 6E4 system, with four 1.2GHz POWER4+ processors and 8GB of main memory.

Overhead of factory mechanism

We ran microbenchmarks to directly measure the code of adding the factory mechanism. Using a factory for an object implies extra cost when creating the object, because the creation must be recorded in the factory's data structures.

We measured the cost of creating three different objects using a factory and using a statically bound method, this includes allocating storage space for the object, instantiating it, and invoking any initialisation methods. Each test was repeated 10,000 times, and the total time measured using the processor's cycle counter. Our results are summarised in Table 1.

The first object, a dummy object, was created specifically for this test, and encapsulates a single integer value. This result represents the worst-case for adding a factory, with an overhead of 12% over the base creation cost of 2.22μs. The next object is an FCM (file cache manager), an instance of which is maintained by K42 for every open file in the system. Creating an FCM object is 5.6% slower with a factory, but this number overstates the true impact, because in practice FCM creation is usually followed by a page fault, which must be satisfied either by zero-filling a page, or by loading it from disk. Finally, we measured the cost of creating a process object, a relatively expensive operation because it involves initialising many other data structures, and found that in such a case, the additional cost imposed by the factory was very small, even before considering the additional costs involved in process creation such as context switches and initial page faults.

To get a more complete picture of the overhead imposed by the presence of factories, we used the SPEC software development environment throughput (SDET) benchmark [11]. This benchmark executes one or more

object	*static create*	*factory create*	*overhead*
dummy	2.22μs	2.49μs	12%
file	4.37μs	4.61μs	5.6%
process	61.1μs	61.5μs	0.73%

Table 1: Cost of creating various kernel objects with and without a factory.

scripts of user commands designed to simulate a typical software-development environment (for example, editing, compiling, and various UNIX utilities). The scripts are generated from a predetermined mix of commands, and are all executed concurrently. It makes extensive use of the file system, process, and memory management subsystems.

We ran SDET benchmarks with four configurations of the system:

1. the base system with no factories in use,
2. a factory on FCM objects,
3. a factory on process objects,
4. factories on both FCM and process objects.

We found that in these cases the use of factories imposes no noticeable performance degradation on system throughput, as measured by SDET. We have not yet extended factories to other objects in the system, however we expect the performance impact to also be minor, since FCMs and processes constitute a significant portion of the kernel objects created.

Time to perform an update

The cost of performing a dynamic update itself is more significant, in some initial experiments we measured 20ms to update one hundred live instances of the dummy object. This is because the hot-swapping implementation is not yet optimised for the case where large numbers of objects are swapped concurrently. We plan to improve this, although since a dynamic update does not block the entire system while it is applied, the overall time taken for an update is less critical.

5.2 Experiences applying dynamic update

To demonstrate the effectiveness of dynamic update, we present three examples of system changes of increasing complexity that we have applied using dynamic update. These examples relate to the memory management code of the K42 kernel [3, 19].

New kernel interfaces for partitioned memory region

Benchmarking of a memory-intensive parallel application showed poor scalability during its initialisation phase. Analysis of the problem determined that a bottleneck occurred during the resizing of a shared hash table structure, and a new partitioned memory management object was developed that did not suffer from the problem. This object added a new interface to the kernel, allowing user programs to create a partitioned memory region if they specified extra parameters.

Adding a new piece of code to the kernel and making it available through a new interface is the simplest case of dynamic update, because we can avoid replacing old code or transferring state information. This update was implemented as a simple loadable module, consisting of the code for the new region object and some initialisation code to load it and make it available to user programs. This module could be shipped with programs requiring it, or could be loaded into the kernel on demand when a program requires the new interface, either way avoiding a reboot.

This scenario demonstrates the use of a module loader combined with an extensible kernel interface to add new functionality to a running kernel. There is nothing inherently new here, existing systems also allow modules to be loaded, for example to provide new file systems or device drivers. However, K42's modularity makes it possible to replace a portion of the page-fault path for a critical application with a new set of requirements, which could not be done on an existing system such as Linux.

Fix for memory allocator race condition

This scenario involves a bug fix to a kernel service, one of the key motivations for dynamic update. In the course of development, we discovered a race condition in our core kernel memory allocator that could result in a system crash when kernel memory was allocated concurrently on multiple CPUs.

Fixing this bug required adding a lock to guard the allocation of memory descriptors, a relatively simple code change. In fact, only two lines of code were added, one to declare the lock data structure, and another to acquire (and automatically release on function return) the lock. A recompile and reboot would have brought the fix into use. However, even with continual memory allocation and deallocation occurring, we were able to avoid the reboot and dynamically apply the update using our mechanism.

The replacement code was developed as a new class inheriting almost all of its implementation from the old buggy object, except for the declaration of the lock and a change to the function that acquired and released it. This caused the C++ compiler to include references to all the

unchanged parts of the old class in the replacement object code, avoiding programmer errors. Simple copying implementations of the state transfer functions were also provided to allow the object to be hot-swapped. The key parts of the replacement code are shown in Figure 2.

The new class was compiled into a loadable module, and combined with initialisation code that instantiated the new object and initiated a hot-swap operation to replace the old, buggy instance. Because this object was a special case object with only a single instance, it was not necessary to use the factory mechanism.

This scenario demonstrates the use of hot-swapping as part of our dynamic update mechanism, combined with a kernel module loader, to dynamically update live code in the system.

File cache manager optimisation

This scenario involves an optimisation to the implementation of file cache manager objects. An FCM is instantiated in the K42 kernel for each open file or memory object in the system.

We discovered that the *unmapPage* method did not check if the page in question was already unmapped before performing expensive synchronisation and IPC operations. These were unnecessary in some cases.

We developed a new version of the FCM object that performed the check before unmapping, and prepared it as a loadable module. Applying this update dynamically required the factory mechanism, because the running system had FCM instances present that needed to be updated, and because new instantiations of the FCM needed to use the code in the updated module.

This scenario demonstrates all the components of our dynamic update implementation, using a module loader to load the code into the system, a factory to track all instances of state information that are affected by an update, and hot-swapping to update each instance.

6 Open issues

The implementation we have accomplished thus far provides a basic framework for performing dynamic update. This is a rich area for investigation, and is becoming important for providing usable and maintainable systems. Here we discuss areas for future work.

Changing object interfaces: Due to a limitation of the current hot-swapping implementation, and because there is no support for coordinated swapping of multiple interdependant objects, we cannot dynamically apply updates that change object interfaces. We could enable this by extending the hot-swapping mechanism to support simultaneous swaps of multiple objects, namely the object whose interface is to be changed and all objects possibly using that interface.

This is our next step in expanding the effectiveness of dynamic update. When considering various changes to K42 to use as example scenarios for this work, many had to be rejected because they involved interface changes. While such changes might be less common in a production system, rather than a rapidly evolving experimental system such as K42, the restriction on changing object interfaces is currently the most serious limitation of this work.

Updates to low-level exception code: Another open issue is what code can be dynamically upgraded. Currently our mechanism requires the extra level of indirection provided by the object translation table for tracking and redirection. Low-level exception handling code in the kernel is not accessed via this table, and as such can not currently be hot-swapped or dynamically updated. It may be possible to apply some dynamic updates through the indirection available in the exception vectors, or through careful application of binary rewriting (for example, changing the target of a branch instruction), but it is difficult to envision a general-purpose update mechanism for this code. There is an open issue for both K42 and other operating systems should we desire the ability to update such low-level code.

Updates affecting multiple address spaces: Our update system does not yet support updates to code outside the kernel, such as system libraries or middleware. At present, it is possible to perform an update in an application's address space, but there is no central service to apply an update to all processes which require it. We intend to develop operating system support for dynamically updating libraries in a coordinated fashion. As part of this work we may need to extend the factory concept to multiple address spaces. We also need to consider the possible implications of changing cross-address-space interfaces, such as IPC interactions or the system call layer.

Timeliness of security updates: For some updates, such as security fixes, it is important to know when an update has completed, and to be able to guarantee that an update will complete within a certain time frame. It may be possible to relax the timeliness requirement by applying updates lazily, marking objects to be updated and only updating them when they are actually invoked, as long as we can guarantee that the old code will not execute once an object has been marked for update.

State transfer functions: State transfer between the old and new versions of an object is performed by the

```
class PageAllocatorKernPinned_Update : public PageAllocatorKernPinned {
public:
    BLock nextMemDescLock;

    void pinnedInit(VPNum numaNode) {
        nextMemDescLock.init();
        PageAllocatorKernPinned::pinnedInit(numaNode);
    }

    virtual void* getNextForMDH(uval size) {
        AutoLock<BLock> al(&nextMemDescLock); // locks now, unlocks on return
        return PageAllocatorKernPinned::getNextForMDH(size);
    }

    DEFINE_GLOBALPADDED_NEW(PageAllocatorKernPinned_Update); // K42-ism
};
```

Figure 2: Update source code for second example scenario (memory allocator race condition).

hot-swap mechanism using state transfer methods: the old object provides a method to export its state in a standard format, which can be read by the new object's import method. This works well enough, but it requires the tedious implementation of the transfer code, even though most updates only make minor changes, if any, to the instance data (for example, adding a new data member). It may be possible to partially automate the creation of state transfer methods in such cases, as has been done in other dynamic update systems [17, 21].

An alternative approach to this problem is used by Nooks to support recoverable device drivers in Linux [31]. 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. A similar system could be used for dynamic update, instead of relying on objects to provide conversion functions, a shadow object could monitor calls into each updatable kernel object, reconstructing the object's state after it is updated. This approach suffers from several drawbacks. First, there is a continual runtime performance cost imposed by the use of shadows, unlike conversion functions which are only invoked at update time. Secondly, the use of shadow drivers is feasible because there is a small number of device interfaces relative to the number of device drivers, but this is generally not the case for arbitrary kernel objects, which implement many different interfaces.

Failure recovery: We do not currently handle all the possible failures that could occur during an update. While we cannot detect arbitrary bugs in update code, it is possible for an update to fail in a recoverable manner. For example, if the state transfer functions return an error code, the update should be aborted. Furthermore, resource constraints may prevent an update from being applied, because during an update both old and new versions of the affected object co-exist, consuming more memory. The system should either be able to check that an update can complete before attempting to apply it, or support transactions [27] to roll back a failed update.

Update preparation from source code: We need a mechanism to automate the preparation of updates from source code modifications. This could possibly be driven by make, using a rebuild of the system and a comparison of changed object files to determine what must be updated. However, it would be extremely difficult to build a completely generic update preparation tool, because changes to the source code of an operating system can have far-reaching and unpredictable consequences.

Configuration management: Our simple update versioning scheme assumes a linear update model, each update to a given class depends on all previous updates having been applied before it. Most dynamic update systems that have automated the update preparation and application process, have also assumed a linear model of update [17, 21]. This is most likely inadequate for real use, where updates may be issued by multiple sources, and may have complex interdependencies.

More complex versioning schemes exist, such as in the .NET framework, where each assembly carries a four-part version number, and multiple versions may coexist [24]. We will need to reconsider versioning issues once

we start automating the update preparation process, and more developers start using dynamic updates.

7 Related work

Previous work with K42 developed the hot-swapping feature [28], including the quiescence detection, state transfer, and object translation table mechanisms. Our work extends hot-swapping by adding the state tracking, module loader, and version management mechanisms, and combining them to implement dynamic update.

To our knowledge, no other work has focused on dynamic update in the context of an operating system. Many systems for dynamic updating have been designed, and a comprehensive overview of the field is given by Segal and Frieder [26]. These existing systems are generally either domain-specific [9, 10, 16], or rely on specialised programming languages [1,17,21], making them unsuitable for use in an operating system implemented in C or C++.

Proteus [29] is a formal model for dynamic update in C-like languages. Unlike our system for achieving quiescence on a module level, it uses pre-computed safe update points present in the code. Our system can also support explicit update points, however we have not found this necessary due to the event-driven programming used in K42.

Dynamic C++ classes [18] may be applicable to an updatable operating system. In this work, automatically-generated proxy classes are used to allow the update of code in a running system. However, when an update occurs it only affects new object instantiations, there is no support for updating existing object instances, which is important in situations such as security fixes. Our system also updates existing instances, using the hot-swapping mechanism to transfer their data to a new object.

Some commercial operating systems offer features similar to Solaris Live Upgrade [30], which allows changes to be made and tested without affecting the running system, but requires a reboot for changes to take effect.

Component- and microkernel-based operating systems, where services may be updated and restarted without a reboot, also offer improved availability. However, while a service is being restarted it is unavailable to clients, unlike our system where clients perceive no loss of service. Going a step further, DAS [14] supported dynamic update through special kernel primitives, although the kernel was itself not updatable.

Finally, extensible operating systems [7, 27] could potentially be modified to support dynamic update, although updates would be limited to those parts of the system with hooks for extensibility, and many of the same problems addressed here (such as state transfer) would be encountered.

8 Conclusion

We have presented a dynamic update mechanism that allows patches and updates to be applied to an operating system without loss of the service provided by the code being updated. We outlined the fundamental requirements for enabling dynamic update, presented our implementation of dynamic update, and described our experiences applying several example dynamic updates to objects taken from changes made by kernel developers. Our implementation also incurs negligible performance impact on the base system.

Although dynamic update imposes a set of requirements on operating systems, as described in this paper, those requirements are already being incrementally incorporated into systems such as Linux. This includes RCU to detect quiescent points, more modular encapsulated data structures, and the indirection needed to redirect invocations. We expect that dynamic update will become increasingly important for mainstream operating systems.

Acknowledgements

We wish to thank the K42 team at IBM Research for their input, in particular Marc Auslander and Michal Ostrowski, who contributed to the design of the kernel module loader, and Bryan Rosenburg, whose assistance in debugging K42 was invaluable. We also thank Raymond Fingas, Kevin Hui, and Craig Soules for their contributions to the underlying hot-swapping and interposition mechanisms, and finally our paper shepherd, Vivek Pai.

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: http://www.research.ibm.com/K42/.

References

[1] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius

Hansen, and Mads Torgersen. Design, implementation, and evaluation of the Resilient Smalltalk embedded platform. In *Proceedings of the 12th European Smalltalk User Group Conference*, Köthen, Germany, September 2004.

[2] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. IBM Research Report RC22863, IBM Research, July 2003.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995.

[8] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[9] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 403–417, Anaheim, CA, USA, October 2003.

[10] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd ICSE*, pages 470–476, San Francisco, CA, USA, 1976.

[11] Steven L. Gaede. Perspectives on the SPEC SDET benchmark. Available from http://www.specbench.org/osg/sdm91/sdet/, January 1999.

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

[13] 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.

[14] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. In *Proceedings of the 3rd ICSE*, pages 295–304, Atlanta, GA, USA, 1978.

[15] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[16] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, Annapolis, MD, USA, May 1996. IEEE Computer Society Press.

[17] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23. ACM, June 2001.

[18] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes—a lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Technical Conference*, pages 65–76, June 1998.

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

[20] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 238–247, Atlanta, GA, USA, June 1986.

[21] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin, Madison, 1983.

[22] 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.

[23] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, NV, USA, October 1998.

[24] Microsoft. *.NET Framework Developer's Guide*. Microsoft Press, 2005. Available from http://msdn.microsoft.com/library/.

[25] Paul Russell. Module refcount & stuff mini-FAQ. Post to Linux-kernel mailing list, November 2002. http://www.ussg.iu.edu/hypermail/linux/kernel/0211.2/0697.html.

[26] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, March 1993.

[27] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–228, November 1996.

[28] 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.

[29] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach, CA, USA, January 2005.

[30] Sun Microsystems Inc. *Solaris Live Upgrade 2.0 Guide*, October 2001. Available from http://wwws.sun.com/software/solaris/liveupgrade/.

[31] 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.