Dynamic updates for object-oriented operating-system kernels

Tobias Langer
FAU Erlangen-Nuremberg
Tobias.Langer@fau.de

ABSTRACT

Modern operating systems have high requirements for service availability, making maintenance downtime very expensive or even impossible. However the threat of unfixed security vulnerabilities on a running system is too high to refrain from installing security updates or scheduling them for later. And also other non-security relevant updates might be needed for the system.

Dynamic update is an approach to counter the trade-off between expensive service downtime and the need for updates. Updates are applied at system runtime and become effective after installing the update completed. An object oriented operating system design has shown to be a good foundation for the implementation of dynamic update systems.

This paper presents techniques for the application of dynamic update in object oriented designed operating systems. Dynamic update will be introduced in general. The concepts then are extended and applied to object oriented operating systems. Additionally an implementation of dynamic update in the K42 operating system kernel will be discussed.

Keywords

Object Oriented Operating Systems, Dynamic Update Systems

1. INTRODUCTION

With the wide spread use of computer systems, software updates have become very common nowadays. Among these updates are fixes for bugs, security updates, performance upgrades and updates introducing new features. The installation of these updates always requires maintenance time and requires a reboot of the service, resulting in downtime. With the high availability requirements to modern operating systems, downtime becomes more and more expensive and is avoided where possible.

System administrators therefore have to weigh the importance of an update in respect to the costs resulting from the downtime. Often maintenance downtimes are scheduled for the service to install update sets. Especially the installation of security fixes is usually very time critical. System administrators can not afford to schedule these updates for a maintenance shut down in the following month. The recent discovery of the "Heartbleed" vulnerability is a great example for the importance of such upgrades.

With the high availability of modern IT-systems, also scheduled shutdowns for non-security relevant updates are costly and should be avoided where possible. Any reboot or restart of a system results in a period of unavailability of the process and state loss of the system.

One solution for the requirement of patches under the constraint of high system availability is *dynamic update*. Dynamic update refers to a set of technologies which allow systems to be upgraded at runtime with no need for reboot. The applied updates become effective after the updating process has completed. The term originally stems from the field of application software but was recently adopted to operating systems [3, 6, 5, 4, 2]. Operating systems yet have even higher requirements than service applications. As operating systems provide their own runtime system, which should be dynamically updatable itself, the requirements for such a system differ fundamentally from dynamic update systems for application software. The operating system itself must provide manners to track the usage of updateable units.

The dynamic update system on the other side should work on a level as fine grained as possible and should support as many changes as possible to be useful for most of the required updates. Furthermore a dynamic update system is only usable when it does not affect the overall performance of the application.

2. DYNAMIC UPDATE

Dynamic update is a term for rebootless, state conserving update method for software. It was first introduced by Lee with the *DYMOS* system [8] for programs written in a Modula derivative.

Today there are several approaches to dynamic update.
Among these are specialized compilers which insert special *update points* or redirections into the compiled programs, making them dynamically updatable [10]. Others use *binary rewriting* to update a running program [2]. With binary rewriting the existing code segment or function is rewritten with new code in the text segment of the application. Notable is the attempt made with *UpStare* [9], in this dynamic update system no explicit update points are needed. Instead the system does a complete stack reconstruction, where the current stack layout is mapped to a new stack layout. This also allows more complex scenarios, e.g. nested function calls and recursion.

Dynamic update has also been successfully applied to operating system kernels: The *Ksplice* tool [2] implements a dynamic update system for the Linux kernel. To update the kernel, Ksplice stops the execution of the computer, except for the Ksplice application, applies the patch for the corresponding functions with binary rewriting, updates affected data, and resumes computation afterwards. Even though the state of the operating system is restored, this approach requires a complete stop of the whole operating system which may lead to brief periods of unavailability of the whole system.

2.1 Principles

In order to develop a common foundation of theoretical essentials and requirements for dynamic update systems, it is worth taking a look at the various types of updates which exist for software. One way to classify updates is to do so by looking at the extent of change they introduce into the updated program. In general a more flexible update system allows more powerful modifications to the underlying program, they tend to become more complex to implement though.

When examining the kind of changes which can come with updates, one can distinguish between three types of updates, which have increasing complexity for the realisation within a dynamic update system, but also allow more in depth changes of the modified system itself:

- *Code changing updates:* These updates exclusively change the code of the underlying system. Code changing updates are in general the easiest to realize. A code changing update can be implemented by binary-rewriting of the corresponding location in the text segment with updated code. Another possibility is adjusting invocations so they point to the updated code instead of to the old code. When doing this, it must be ensured that all references to old code are replaced completely.
- *Code and data structure changing updates:* These updates do not only change existing code, but also the layout and content of data structures. This is e.g. the case, when an additional element counter is introduced into a list data type. When updating, all existing instances of the data structure must be transformed into the new data structure. Furthermore it must be ensured that future instances are by default created according to the new scheme. Any code changing updates can be resolved as for simple code updates.
- *Interface changing updates:* In contrast to the previous two update types, interface changing updates introduce or remove functionality into or from the updated code and modules, which effectively leads to an altered module interface. This is the case when new classes are introduced to or removed from the software architecture, when classes get new methods, or if classes are merged or split. Interface changes also appear when the signature of methods is altered, meaning when parameters are added or removed or when the return value is changed. In principle these updates can also be resolved by changes of references, however additional attention must be paid to backwards compatibility. Additionally one must take care to make the new interface known to the system. Another implementation for interface changing updates would be interposing objects which act as Adapters to the original object. Using the Adapter pattern allows to modify the external interface, without a need for changes inside objects.

Baumann et al. present a similar classification in [5], whereas they lay focus on the necessary effort in relation to the limitations of the dynamic update system. For this they rule out interface changing updates completely and differ between the amount of data structures which have to be changed for the update instead. This distinction between different types of data structure changing updates comes from the consideration that there will be substantially less effort in replacing one single global data structure, e.g. the page management, in contrast to tracking and changing structures with a high number of instances, such as process control blocks.

The different update types can be summarized as code changing and structure changing updates. For the latter ones simple statical replacing is not sufficient. As there might be alive instances of the structures or processes which wish to access modules whose interface changes when applying the update, one must take care of the *state* of the running program as well. At design time of the dynamic update system one can already take account for the supported update type. The feasibility of different update types depends on the chosen minimal updatable unit, whereas this can range from code snippets, to whole functions, objects or modules.

In [7], Gupta, Jalote et al. present a theoretical framework to describe dynamic update systems in general. Any running system can be described by a tuple (s, Π), with the running code Π and its internal state s. Applying an update effectively maps this tuple to the tuple (s', Π'). The mapping of the program code can be solved by the above listed manners. For the state transfer a special function must be given. This function is called the *state transformer function* and must guarantee that the original state is converted in such a way that the updated code can handle the updated state.

The main problem for dynamic update systems is the update of resources, code and state alike, when they are currently in use, and avoiding erroneous behavior. Altering running code of a function while the processor is executing this code is almost certain to lead to unexpected behavior. One way to avoid this problem is for the dynamic update system to wait for so called *safe points*[1]. At these safe points, the update system is guaranteed to be able to update code and the state without the hazard of running in erroneous states.

Summarizing any dynamic update system has to follow a general flow when applying an update to a system. In [3] Baumann et al. describe this flow as follows:

[1]In some literature the safe points are also called *update points*. The also often noted *quiescence points* mean points where no thread exists which involves the to be updated resources.

- *Apply updates at safe points* A change to the running system may only occur when one can assure that the update does not result in unexpected behavior, as e.g. the system state becomes invalid.
- *Transfer of state information* Whenever the updated unit carries a state, the update must ensure that state is valid for the updated program code as well.
- *Invocation redirection* Any invocations to the updated unit must lead to the new version of the code, rather than to the old one.

2.2 Dynamic update for operating systems

In contrast to application software, operating systems impose further demands to dynamic update systems. They do not have a designated runtime system but rather provide their own runtime system. This means that the operating system must itself implement the means for a dynamic update system.

In [5] Baumann et al. identify these means which dynamic update systems demand from an operating systems:

- *Updatable unit* The system must define an updatable unit, meaning that the exchangeable parts must have a fixed, well defined update interface and behavior, so any update can work uniformly. This requires the operating system to be designed with modularity.
- *Safe point* As stated above, the system must support certain points where the dynamic update system knows for sure that the updated data and code is not currently accessed or executed. This can be guaranteed by special safe points, by detecting the access to data and code and explicitly blocking the access to these resources, or by checking the system for idle time or *quiescence*. With operating systems such a quiescence point occurs when the event queue runs empty.
- *State tracking* As the dynamic update system not only has to update the code, but also transfer the old system state into a new one, the operating system must provide a mechanism to easily access all instances of state holding data.
- *State transfer* To allow the update system to transfer the current state of the operating system to the new state, the operating system must provide a mechanism for the replacement of state holding units.
- *Redirection of invocations* After an update all calls and references should point to the updated code. The operating system must thus enable easy redirection of references. Thinkable would be a global reference table which resolves all invocations.
- *Version management* The more updates are applied to the system, the higher is the chance for dependence between updates. To prevent invalid states or broken code, the system must implement means to check whether all premises for updates are given. The inter update dependencies thereby can become quite complex, appropriate mechanisms must be provided.

  In the case of interface changing updates, user space programs must also have a possibility to check the currently supported interface of the kernel.

3. DYNAMIC UPDATE WITH OBJECT ORIENTED METHODOLOGIES

Dynamic update support should ideally be designated at design time of the operating system. Especially object oriented designs offer the means for a well structured design with respect to extendability with dynamic update.

The above named requirements for an operating system with dynamic update support can be realized by object oriented means with little effort. In a fully object oriented operating system, the minimal modular unit is an object. Objects on the one hand "contain" executable code in their methods. On the other hand they carry a certain state. Thus they contain both things one wants to change with an update, code and state. It seems therefore natural to set objects as the minimal updatable unit for the dynamic update system. To enable as much flexibility as possible, as many function calls as possible should be not bound statically, but resolved at runtime. If now a module of the operating system is to be updated, one must simply exchange all objects involved.

Coming from the updatable unit, the methods for the realization of the other requirements have to be developed and are highly depending on the structure of the operating system itself. In the following some general methods for the update of objects will be discussed. A concrete realization of the other requirements will be discussed later on.

3.1 Interposition

Interposition describes means where a new object `C` is inserted between two existing objects `A` and `B`. All calls from `A` to `B` are then effectively redirected to `C`. This allows a series of modifications to the original control flow.

The interposed object has effectively the means to redirect or suppress any call to the original object. By reimplementing methods of the old object with updated code, fixes can be applied. Also prologues and epilogues for the original method call are thinkable.

Interposing also allows the implementation of interface changing updates by using the Adapter design pattern. To add new functionality to an existing object, the interposing object is implemented as an Adapter for the existing object and inserted between the caller and the updated object. From the view of the caller, the old object has a changed interface, as the former is unaware that there is an Adapter inbetween the two old objects. The original object itself remains unchanged. In Figure 1 an interposed object is shown, which adds further functionality to an existing object.

Using interposition however also leads to additional indirection, which results in the worst case in unnecessary overhead. Furthermore only a small amount of the possible updates can be covered with this technique, as the state of the underlying object can not be updated but only substituted or supplemented by the state of the Adapter object. Additionally the original object remains in memory. Updates

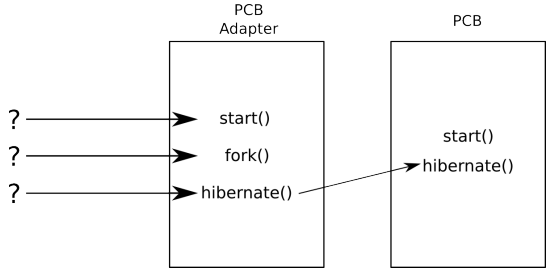

Figure 1: An interposing Adapter object which fulfills various tasks: The `start()` function call is completely shadowed by an updated version of the function. The `fork()`-function has been introduced as an interface changing update. The `hibernate()`-call is simply forwarded.

which do more restructuring in the applications design, such as splitting or merging classes, are tricky to be mapped to the Adapter pattern.

3.2 Hot-swapping

Hot-swapping describes methods of replacing code or whole modules in the running operating system. This also allows the transformation of the system's state.

The hot-swapping technique also involves interposition. At the beginning of an update a special mediator object is interposed between the caller and the object which should be updated. This mediator takes care that all existing calls to the object finish, while blocking any new incoming calls. When all running, unblocked calls have finished, it is guaranteed that no code currently depends on the state of the object. A quiescence state is reached, and the object can be updated without danger of reaching an invalid state. An update could be realized by creating a new object, which holds the altered state. After the update of the object has completed, the reference to the old state object is redirected to the new state object. Then the mediator object and the old state object can safely be destroyed. This process is shown in Figure 2.

While hot-swapping allows the exchange of modules and objects at runtime, it still requires to block the access to an object for at least a brief moment until quiescence is reached. When an update requires the exchange of many object instances, such as process control blocks, this might lead to performance losses.

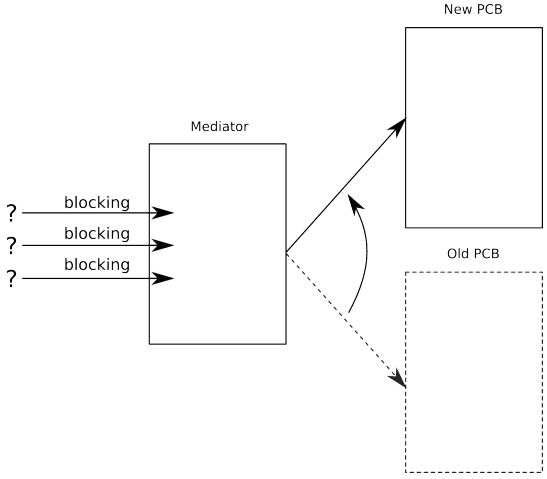

Figure 2: Procedure of a dynamic update of an object by hot-swapping it. The interposed mediator object blocks all incoming calls. When it ensured that there are no active threads interacting with the old object, the reference is redirected to the new object, all calls are unblocked.

3.3 Lazy update

As mentioned above, updates might require the transfer of more than one state carrying object. This is especially the case if the update affects units with multiple instances. For these cases updating all object instances at once takes much time while the system has to remain in a quiescence state. This might lead to heavy performance losses.

One consideration is that not all existing instances of the updated object have to be updated at time of update, as they might not be used immediately. Furthermore some of these objects might not be used anymore at all. With lazy update, object instances are updated at the time of their next use. This leads to a slight overhead at time of their first use after the update, however the overall performance loss is not as severe as for a state transformation at time of update. All calls following the first call which causes an update then are processed normally and thus cause no further overhead.

Lazy update again can be realized using interposition. For each object which should be updated, a lazy update object is interposed. Upon first invocation of the lazy update object, the actual update process is started for the state object. After the state transfer, the lazy object and the old object instance are deleted and the method call can run for the new instance.

4. DYNAMIC UPDATE IN K42

In [3, 6, 5, 4], Baumann et al. introduce a modification for the K42 kernel which adds a dynamic update mechanism. This dynamic update mechanism makes use of the kernel's hot-swapping mechanism to allow updates on object level.

4.1 The K42 kernel

The K42 kernel is a research micro kernel developed by IBM. It aims for scalability and compatibility to the Linux ABI

and API. The kernel was designed to be modular and object-oriented, as well as decentralized [1]. Furthermore it is implemented to be event-driven, with non-blocking, short-lived kernel threads.

The K42 kernel is heavily structured in an object-oriented manner, with each resource being managed by a set of object instances and all data being encapsulated. With the relevant data encapsulated that fine-grained, the kernel has good prerequisites for a dynamic update system on object level.

To support online reconfiguration and adaption, the kernel supports hot swapping for modules and components. The hot swapping mechanism in K42 allows the replacement of objects, which makes it a good starting point for the implementation of dynamic update.

4.2 Implementation

The presented dynamic update system makes heavy use of the object oriented structure of the K42 kernel. In the following it will be explained how the modified kernel fulfills the above named requirements and which features the dynamic update features additionally implement. The dynamic updates are installed using a module loader.

K42 manages its objects via a central *object translation table* which is an additional indirection layer and allows a centralized unified access to object references. This table is the basis of the dynamic update system, the invocation redirection takes place by replacing entries in the object translation table. The kernel's hot-swapping mechanism makes use of this table for the replacement of objects. The kernel is designed to be event-driven and services these events with short lived threads. The hot-swapping mechanism waits for a state of quiescence for the to be changed object. When this state is reached, the reference in the object translation table is replaced. Due to the short life of the kernel threads, many of these quiescence states occur.

To check for quiescence states, the K42 kernel uses mediator objects. These mediator objects make use of the fact that kernel threads are short-lived and non-blocking. In K42 all kernel threads belong to a generation, which is determined by the generation the kernel is in when the thread is created. For each generation a count of all active threads is maintained. By advancing the generation and waiting for all threads of the previous generation to be completed, the mediator can make sure no currently running threads are holding references to the updated object. The K42 hot-swapping system uses this mechanism by explicitly starting a new generation when an object should be swapped.

The dynamic update system implemented by Baumann et al. takes object instances as the updatable units. An update of the object instances is done by hot-swapping. The hot-swapping mechanism originally was only capable of replacing special object instances. For the dynamic update system the kernel was modified in such a way that all object instances are generated not statically but by own factories. These factories cover multiple tasks, first they construct new objects, second, they take the role of the state tracking mechanism, by tracking all created object instances, and last they are

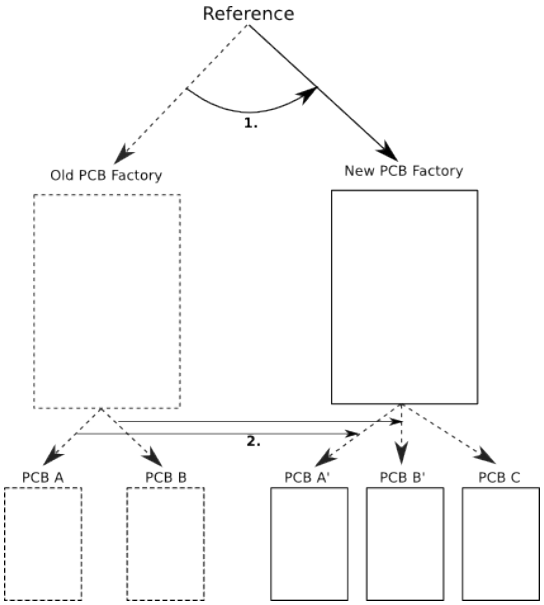

Figure 3: The steps of a dynamic update of factories in a K42 system. At (1) the factory object is replaced with a hot-swap. Using the bookkeeping mechanism of the factory, all existing old instances of the updated object are updated in (2). All calls to the new factory directly result in the instantiation of the new version of the object.

responsible for deleting object instances. When applying an update which requires the update of multiple object instances, the factory of the object type is queried for a list of all existing object instances. Thereby the hot-swapping of object instances can be applied lazily. In Figure 3 the process of an update of factories and instances is shown.

To ensure not only the currently existing instances of objects are replaced but also ones instantiated in the future, the corresponding factories of the updated objects are also updated using the hot-swapping mechanism.

Additionally the dynamic update system contains a simple version management system. Each factory object contains an internal version counter. Updates also have a version number for the updated object. The version numbers are assigned linearly. When applying an update, the internal version counter of the factory is compared to the update's version number. Only if the version number of the update directly follows the current number of the factory, the update can be applied and the internal counter of the factory is set to the version number of the update. This way update dependencies can be resolved easily and wrong update sequences can be prevented.

This dynamic update concept allows changes to many parts of the kernel. Unfortunately, the interface changing mechanism is only suitable for minor changes, such as added functionality. In some cases added or dropped parameters also can be supported. If the changes however are more extensive, Adapters are not suitable. Furthermore most of the code which is initially run to boot up the K42 kernel is not

written in an object oriented manner, which makes it improbable to be updated using this dynamic update system. Other methods like binary rewriting would be more suitable for these cases. Nevertheless if minimizing downtime is a goal, this code should not have to be executed while the system is up and running.

5. DISCUSSION

The presented ideas for the realization of dynamic update systems with methods of object oriented systems are highly depending on resolving object references at runtime. In the implementation of the discussed dynamic update system for the K42 kernel most of the statically bound references are converted to calls which are dynamically resolved. This hinders the application of especially inter-procedural optimization techniques such as code inlining, or devirtualization and branch prediction and thus leads to performance losses. Baumann et al. argue that this is the case for all modern operating systems with loadable module support [4] and do therefore not analyse the impact of their restructuring. However the proposed dynamic update system requires the operating system to completely refrain from many optimization techniques in order to function. Furthermore the dynamic update system works on object level, while loadable modules are much more coarse-grained than objects, which means that the introduced overhead for modules is not as severe as for objects. The performance difference might therefore be quite high whereas especially for systems with high performance requirements a performance analysis must be made to weigh the trade-off between probable performance and possible dynamic update support. Additionally the presented implementation introduces further indirection layers into the system to make hot-swapping applicable for dynamic update. The resulting overhead from the additional indirections however has proven to be negligible. This is explained with the partial storage of the object translation table in the level 2 cache. The indirection imposed by the object translation table leads to an additional instruction per virtual function call, however the necessary information can be looked up in the cache.

In general, existing operating system kernels indeed have a modular structure, nevertheless the strong object oriented design of the introduced K42 kernel is the exception rather than the rule, especially since modules are much more coarse-grained than objects. Rewriting these kernels especially for object orientation or explicitly for dynamic update is certainly no option. Nevertheless the modular structure of modern operating systems can be used to apply techniques like hot-swapping, to enable dynamic-update, at least at a module wide level when there is the possibility to check for quiescence states by tracking the usage of these modules. Furthermore future developments for these kernels can indeed especially have a focus explicitly on dynamic update and stronger modularization.

6. CONCLUSION

In this work the application and realization of dynamic update methods to operating systems was discussed. It was shown how an object oriented design of the operating system can be instrumented for a usable implementation of dynamic update support. Even though the introduced implementation for the K42 kernel had to cope with a system which was not primarily designed for dynamic update, it showed to be useful. With systems which were explicitly designed with dynamic update in mind, updates would be even more capable of making in depth changes in the system.

Nevertheless the introduced methods are not limited to object oriented operating systems. The introduced concepts of hot-swapping and interposing can easily be extended for dynamic update systems for regular applications. Central instances as a reference table could be built up with compiler support. On the other hand many modern operating systems have a very modular structure and even support dynamic loading of additional modules, e.g. the loadable kernel modules of Linux. When these methods are extended to other types of modules as well, it would be thinkable to also enable hot-swapping techniques on a modular level.

7. REFERENCES

[1] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42 overview.

[2] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[3] A. Baumann, J. Appavoo, D. Da Silva, O. Krieger, and R. 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*, pages 21–27, 2004.

[4] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, page 26. USENIX Association, 2007.

[5] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *USENIX Annual Technical Conference, General Track*, pages 279–291, 2005.

[6] A. Baumann, J. Kerr, J. Appavoo, D. Da Silva, O. Krieger, and R. W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in k42. In *6th Linux. Conf. Au*, 2005.

[7] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Software Engineering, IEEE Transactions on*, 22(2):120–131, 1996.

[8] I. Lee. *Dymos: a dynamic modification system.* PhD thesis, University of Wisconsin–Madison, 1983.

[9] K. Makris and R. A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction.

[10] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.