Improving Dynamic Update for Operating Systems

Andrew Baumann
University of New South Wales
& National ICT Australia
andrewb@cse.unsw.edu.au

Jonathan Appavoo
IBM T.J. Watson Research Center
jappavoo@us.ibm.com

1. INTRODUCTION

Modern operating systems are subject to a constant stream of patches and updates: to fix bugs, improve performance, or add features. Dynamic update offers significantly increased availability for operating systems, and enables administrators to avoid a difficult choice between the cost of down time and the risk of remaining unpatched. However, an operating system kernel is a unique environment for dynamic update; it is generally event-driven, multithreaded, and involves a high degree of concurrency and asynchrony. It also provides a very restricted runtime environment. Existing dynamic update mechanisms are generally unsuited for use with operating-system code, either because they do not support concurrency [11, 13], require the system to be implemented in a specific language [1, 7, 9], or rely on a higher level of runtime support than is feasible within a traditional OS [5, 6].

This work aims at developing a model supporting dynamic update to operating systems code.

2. DYNAMIC UPDATE FOR OPERATING SYSTEMS

In recent work [3], we have developed a dynamic update model and prototype implementation. This model relies on the modular nature of modern operating systems to define an updatable unit, and requires a mechanism for achieving a safe point for applying an update within such a unit, a system for tracking state maintained by the unit and then transferring that state to the form required by the updated unit, a mechanism to redirect all invocations from the old unit to its update, and a form of version management to track update dependencies.

A prototype of the model has been implemented in the K42 experimental operating system. This prototype makes use of K42's object-oriented design, as well as its hot-swapping feature, which provides the mechanisms for achieving a safe point and redirecting invocations. The implementation adds *factory objects* for tracking state information associated with the updated object classes, and for coordinating the update process. It also makes use of a kernel module loader, and the *hot-swapping* feature [12] of K42. As described in the previous work [3], a number of actual developer changes to the K42 kernel have successfully been applied as dynamic updates to a running system. These include adding new kernel interfaces, fixing a race condition in the kernel memory allocator, and a performance optimisation in part of the memory management code.

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.

3. CURRENT AND FUTURE WORK

3.1 Changes to interfaces

The current implementation does not support any updates which change the interfaces of kernel objects. This includes adding and removing methods, or changing their parameters, and has been the biggest limitation to the applicability of our update system—in selecting example updates to demonstrate the system, we found that many had to be discarded because of this restriction. Interface changes are problematic, because if an interface were to change, all code using that interface would potentially need to be changed at the same time. However in a running OS kernel, changing many different objects at once leads to two potential problems. First, the system could become unresponsive while many affected objects are changed, because an object must be kept quiescent while its data is converted. Second, deadlock could occur, if there are any interdependencies between the updates.

To avoid these problems, an interface-changing update should be staged, so that code which uses the old interface can continue to function with the new interface before it is updated. By attaching version information to the references used to invoke objects, it becomes possible to detect when an older version of the interface is expected. The call can then either be adjusted to conform to the new interface (for example, by changing the method number, or adding a default parameter). This should enable support for a large proportion of all interface-changing updates.

This scheme supports dynamic updates which change interfaces in such a way that old invocations can be rewritten to conform to the new interface. For example, adding, renaming or reordering methods, adding parameters with default values, or reordering parameters. The obvious updates which could not easily be supported by this model are the removal of a method with no replacement available, or the addition of a parameter with no default or easily-computed value to a method. A recent survey of several large software systems [10] has found that changes to and deletions of function prototypes and type definitions, are relatively infrequent. This suggests that support for a restricted form of interface changes may have value, however we need to confirm this by performing a similar study using the K42 revision history.

We are also planning to perform experiments to investigate the feasibility of supporting arbitrary interface-changing updates, which would require achieving system-wide quiescence, and changing multiple affected objects at once. These experiments will involve

examining previous changes to determine which objects are affected, simulating the system under various workloads to count how many of those object instances would need to be concurrently transformed, and thus estimating the total time for which the system would be required to be blocked during an update.

3.2 Multiple-address-space updates

To date, we have only supported updates to kernel code. However, some updates to operating systems affect code that is part of the system libraries and runs in an application's address space. These include changes to the system-call interface, as well as pure library updates. To support such updates, a model is required for coordinating updates across all protection domains, rather than just within the kernel. As the first step in providing this support, we plan to extend the factory model to cover multiple protection domains.

3.3 Transfer to Linux

The K42 operating system offers a number of features for dynamic update, primarily the object-oriented nature of the system, with all data encapsulated behind method interfaces, as well as the support for hot-swapping [12], enabling run-time changes to object implementations. However, the dynamic update model should support all operating systems with a suitably modular structure. To support claims regarding the model's generality, initial investigations have been made into the requirements for a Linux implementation [4].

As with many other modern operating systems, Linux supports the use of loadable modules for some parts of the kernel, such as file systems and device drivers. Code within these modules is invoked via pointer indirection, such as in the virtual file system layer. The modularity, combined with the function pointer indirection, and the availability of read-copy-update mechanisms to help detect quiescence, provide similar functionality to the basic mechanisms used to implement dynamic update in K42. The main outstanding conceptual problem is handling kernel threads blocked inside a module that is being updated. In some cases, such as an interruptible system call, these threads can be aborted. In the remaining cases, we may have to resort to a wait and retry mechanism.

4. RELATED WORK

Although many dynamic update systems have been designed, and some of these systems already include features such as support for interface changes, to our knowledge, no other work has focused on dynamic update in the context of an operating system. This has led to limitations in the applicability of existing dynamic update models to operating systems code.

Some systems are language-based, or require specific runtime language support. These include DYMOS [9], the work by Hicks [7] which depends on a type-safe variant of C, and several other examples [1, 2] which make use of features available only in higher-level languages. Dynamic C++ classes [8] could be used to support dynamic update, but do not address the important problem of converting existing data. Other dynamic update systems, including PODUS [11] and Proteus [13] require programs to be single-threaded. Finally, some systems are domain specific, and rely upon support which could not be provided within an operating system environment, such as migrating message channels [6], or transaction features of databases [5].

5. CONCLUSION

Full dynamic update support for mainstream operating systems would offer many benefits, but is currently not available. This work is heading towards that goal, by developing a dynamic update model that has been shown to work for K42, improving it to support interface changes and multiple-address-space updates, and experimenting with similar dynamic update features in Linux.

Acknowledgements

This work is partially supported by DARPA under contract NBCH30390004. 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.

6. REFERENCES

[1] J. R. Andersen, L. Bak, S. Grarup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation, and evaluation of the Resilient Smalltalk embedded platform. In *12th ESUG*, Köthen, Germany, Sep 2004.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*, chapter 9, pages 121–123. Prentice Hall, 2nd edition, 1996.

[3] 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 *2005 USENIX Techn. Conf.*, pages 279–291, Anaheim, CA, USA, Apr 2005.

[4] 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*, Canberra, Australia, Apr 2005.

[5] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, pages 403–417, Anaheim, CA, USA, Oct 2003.

[6] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *3rd Int. Conf. Configurable Distr. Syst.*, pages 70–80, Annapolis, MD, USA, May 1996. Comp. Soc. Press.

[7] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Aug 2001.

[8] G. Hjálmtýsson and R. Gray. Dynamic C++ classes—a lightweight mechanism to update code in a running program. In *1998 USENIX Techn. Conf.*, pages 65–76, Jun 1998.

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

[10] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax syntax tree matching. In *2nd Int. WS Mining Softw. Repositories*, pages 2–6, May 2005.

[11] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *Softw.*, 10(2):53–65, Mar 1993.

[12] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *2003 USENIX Techn. Conf.*, pages 141–154, San Antonio, TX, USA, 2003.

[13] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *32nd POPL*, Long Beach, CA, USA, Jan 2005.