High performance dynamically updateable software architecture

Author: Jim J. Moore

Research Disclosure Database Number 481010

Published in May 2004
(Electronic Publication Date : 14 Apr 2004 13:47)

The Research Disclosure Journal is normally published and distributed on the 10th of every month unless that date coincides with a weekend or public holiday, when it is published directly afterwards. In these cases it is always published by the 12th of every month. Every disclosure is also placed on the RD Electronic database as soon as it is received and it may be published on the database prior to being published in the next edition of Journal.

Research Disclosure is the unique international defensive publication service that allows the world's intellectual property community to establish prior art, and provides an alternative to obtaining a patent at a fraction of the cost and the time taken. It is the world's longest running, independent, industry standard prior art disclosure service.

Kenneth Mason Publications Ltd give consent for this disclosure to be printed out providing it is for personal use, or for the personal or internal use of patent examiners or specific clients only. Photocopies may be made providing it is for personal use, or for the personal or internal use of patent examiners or specific clients and not for resale and the copier pays the usual photocopying fee/s to the relevant Copyright Clearance Centre. This consent does not extend to abstracting for general distribution for advertising, or promotional purposes, for creating new collective works or for resale. This consent also does not extend to other kinds of scanning, printing or copying, such as printing, scanning or copying for general distribution for advertising, or promotional purposes, for creating new collective works or for resale. Document delivery services are expressly forbidden from scanning, printing or copying any Research Disclosure content for re-sale unless specifically licensed to do so by the publishers.

Research Disclosure Journal, ISSN 0374-4353. © Kenneth Mason Publications Ltd
The Book Barn, Westbourne, Hants. PO10 8RS. UK
Tel: +44 (0)1243-377977 Fax: +44 (0)1243-379136
e-mail : info@ResearchDisclosure.com

High performance dynamically updateable software architecture

Background

A computer system typically consists of a combination of hardware which is controlled and managed by operating system software that is installed and executing on that system. If the computer system is being used for a mission critical purpose, then a fault in that system could cause a critical impact to the business and/or operations of the user.

If this fault occurs in the currently executing software, then there are traditionally only two ways to handle the fault. One is to “fail-over” the service provided by that computer system to a secondary system capable of assuming control. The other is to endure a period of downtime while remedial action is taken. This is frequently in addition to the original period of downtime caused by the first occurrence of the fault because the system may have automatically reset and resumed operations. Therefore it needs to be taken out of production again so that a new version of the software, complete with a fix for the identified fault, can be installed.

In circumstances like these, particularly where a secondary system is not available to take over the provision of service, then it would clearly be advantageous if the executing software can have a fix applied without needing to take the computer system out of production. In other words, the requirement is the capability to update a currently executing software image on a computer system. This is widely referred to as “dynamic software updates” or more commonly “hot patching”.

However, the ability to implement this feature into an existing operating system is largely constrained by the architecture of the software. If the ability to dynamically update the operating system software is designed in from the start, then aside from issues around the performance, logical state and safety of the mechanism, it is not too difficult to implement. It is, however, far harder to retrospectively add this facility to an operating system that did not have this capability designed into it.

This paper discloses a method for the design of a software architecture that both enables the dynamic update capability while also delivering very high levels of performance, safety and logical integrity.

Summary

The method proposed in this paper takes the layered page table concept found in many implementations of memory management units and applies it to the problem described above. By implementing a software execution architecture using this method, it becomes possible to quickly and safely update executing software while at the same time ensuring

high performance from the software by extending the use of existing hardware and software technology that understands the layered table principle.

Detailed Description

In order to fully explain the method, it is necessary to briefly explain the concept of layered tables and how they are currently applied to memory management, so that the reader can see how these ideas are innovatively applied to a software execution architecture.

A computer system may have some amount of physical memory installed in it (RAM) which may or may not be contiguously addressable at the hardware level. It is very common for operating systems to virtualize the memory configuration by providing a contiguously linear logical address space, so that host software running on the system does not need to worry about how much physical memory there is or about the hardware configuration of that memory. Software then reads and writes to memory via this virtual address range and the hardware and operating system software cooperate to translate the supplied virtual address into a physical address.

This virtual to physical address translation is often achieved via the use of layered tables as shown in figure 1 below.

There are two types of table; a "page directory" and a "page table". A page table contains an array of page addresses, each of which is the physical address of a "page" of physical

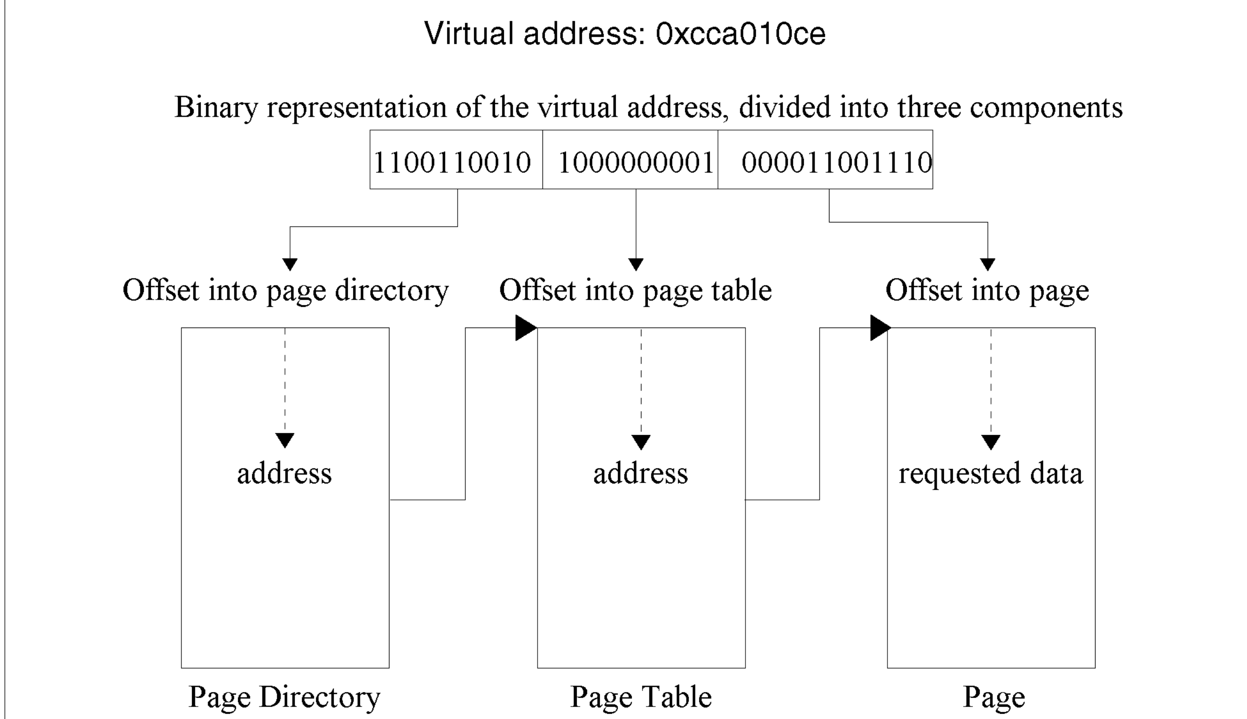

Figure 1 – Address translation using layered page tables

memory. The page directory is an array of page table addresses, each of which is the physical address of a page table. In this way, a huge range of virtual memory can be translated. The virtual address is broken down into three parts. The first part is used to offset into the page directory, to retrieve the address of a page table. The second part is used to offset into that page table to identify a physical page of memory and the final third part is simply the offset into that page where the required data is to be read from or written to. A deep understanding of memory management concepts and the elegance of the page layering model is not required to understand this disclosure, it is merely sufficient that the reader understands that existing hardware, in cooperation with the operating software, already uses this mechanism for memory address translation.

It is possible to apply this architecture to the design of operating system software. Consider an operating system to be a collection of facilities (e.g. memory), each of which may have a number of implementations (e.g. physical and virtual memory) and each implementation may have a number of associated methods (e.g. to allocate or deallocate memory).

With this concept in mind, it is now possible to describe a layered table approach to the execution of the software. Any function call can be described by an encoding of the facility, implementation and method that the function represents. The same approach used to translate a virtual address can be used to translate an encoded facility, implementation and method into an address for the function associated with that method. This is shown in figure 2 below.

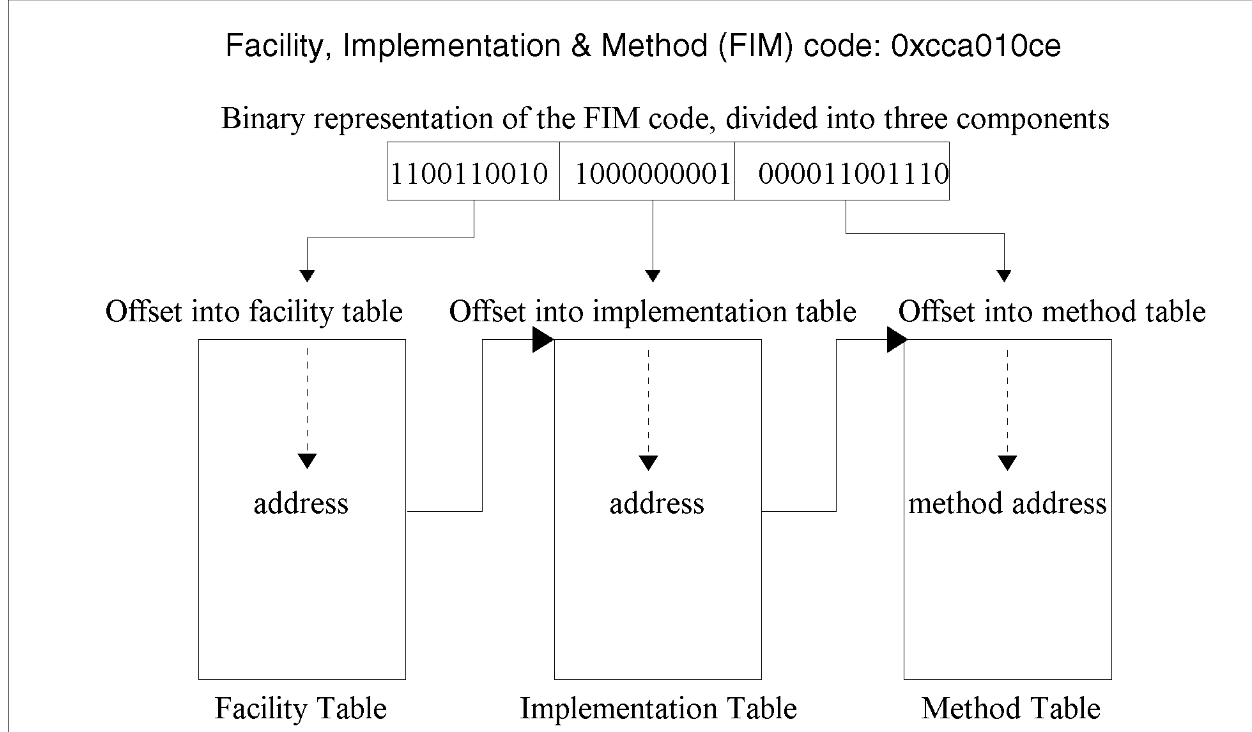

Figure 2 – Translating a FIM code into a callable address using layered tables

This architecture provides a powerful mechanism for dynamically updating an executing software image because all function calls (method calls) are made via translations. This means that if the need arises to repair or update a method, all we need to do is update the method table entry so that it points to the replacement function. Furthermore, it is possible to update an entire implementation simply be rewriting an implementation table pointer. For example, if we have a UFS implementation of the filesystem facility and we need to make sweeping changes to that version of UFS, then we can simply update the implementation table entry corresponding to UFS rather than update each existing method table entry for that implementation. Hence, we have a very high performance method for dynamically updating an executing software image.

This architecture has the additional benefit of the least significant bits of the facility and implementation table entries being free (because the address of the next layered table can be page aligned and so no address values need to be represented by these bits).

Again, analagous to the implementation of MMU page tables, these lower bits can be used to contain state information. This can be useful for dynamic software updates, as it provides a location for us to keep state information about the software, e.g. whether it is safe, whether an update is pending, whether a thread is within that particular area of software and so on.

One of the more significant disadvantages of indirect function or method calls is the additional processing time it takes to make the call. From a pure speed perspective, it is obviously quicker to call a function directly than go through some level of abstraction. However, the layered translation table approach to calling a method provides one of the highest performing mechanisms for abstracted calls. This performance can be improved still further by enhancing processor design to implement the translation in hardware, in the same way that we do for memory address translation. This should be fairly trivial to achieve, as the hardware technology already exists for traversing layered page tables in an MMU and this method simply extends that same principle.

Disclosed by Jim J. Moore