Programming Paradigms and Kernels

s0840449

November 25, 2010

If one was to examine the source code for any popular operating system kernel - be it Linux, Unix, Windows NT, Mach[1], or even Google's brand new Chrome OS - one would discover two things that they all have in common. Firstly, they are all written in a combination of C and some assembly code[2]. Secondly, without a single exception they are all written imperatively. (While some kernels have an object-oriented design, they are still written in a modular, imperative style.) It is surprising that imperative programming is the preferred choice for all of these operating systems, especially given that there are many other common paradigms available for usage. Different paradigms can be beneficial to development, by providing features such as type safety or functionality separation, and can also provide a better experience for the end-user, with features such as increased reliability and the capability for more complex kernel configuration. This essay attempts to explore this issue, starting by examining the advantages and disadvantages of alternative paradigms and languages for kernel implementation, and also looking at why imperative programming is still overwhelmingly the paradigm of choice.

Although the scope of programming paradigms is large and quite vague, four main sections can be identified[3]. These are agent-oriented programming, declarative programming, imperative programming and structured programming. Declarative programming can be divided into constraint-based, functional and logic programming. Likewise, structured programming can be separated into object-oriented and aspect-oriented programming[4]. Therefore, we now have seven paradigms - agent-oriented, func-

[1] The kernel used by Mac OS X[1].

[2] Mac OS X, Windows NT and Chrome OS all also contain some C++ code[1, 2], but it is unclear whether it would be found in their kernel code or just the general operating system code.

[3] Most other paradigms are either too vague to discuss, or can be defined as belonging to one or a combination of these sections.

[4] Although a sub-section of object-oriented programming, the aims of the aspect-oriented paradigm are sufficiently different to consider it as a separate paradigm when examining kernel design.

tional, object-oriented, aspect-oriented, constraint, logical, and imperative. I focus only on the first four of these paradigms as they provide the best alternatives to imperative programming for kernel design. Constraint and logical programming are not included as applying their approach for solving problems to kernels would be such a complex task that no evidence could be found indicating that it has ever been attempted either academically or in a professional setting.

Arguably one of the most well-known and used paradigms in computing today, object oriented programming developed naturally from the concepts of procedural and modular programming. As is evident from the name, object oriented programming revolves around the idea that everything is an 'object' - an item with a set of variables (what it 'knows' or 'has') and methods (what it can do). The main features of object oriented programming are classes, methods, message-passing, encapsulation (for both data and methods), and inheritance[3]. If applied to kernel design, these features would likely provide beneficial to both the programmer and user, and so it is not a surprise that much work has been done in the area of object oriented kernels, both professionally and academically[4-13]. Indeed, as mentioned in the introduction, most operating systems are object oriented at some level, even if not in the kernel.

The most obvious benefits for the programmer are the same ones that object oriented programming bring to any software project - type safety, complexity reduction, modularity, extensibility, maintainability and re-usability. These same benefits can also extend to the kernel itself. In their 2002 paper, Golm et al. introduce the JX operating system, which highlights many of the advantages object oriented code can bring to a kernel[4]. Apart from a small microkernel written in C - which is used for extremely low-level tasks such as system boot up and CPU state restoration - the entirety of the operating system is written in the object oriented language Java. One of the main benefits that the JX kernels makes use of is the principle of modularity. The entire system is split into 'domains', which are individual "unit(s) of protection and resource management". By making use of this modularity JX is able to have an extremely configurable system - for example, each domain is able to have its own garbage collector, scheduler[5], and view of memory; all without much effort being required by the developers. JX's memory management also makes use of the object oriented paradigm - chunks of memory are represented as 'memory objects'. These specialised objects can be seen from the callers point of view as normal, everyday objects which can be interacted with via methods, but when translated by the Java Virtual Machine (JVM) they are replaced by machine instructions

[5] In JX, domains are scheduled to run by the main scheduler, and they then use their own internal scheduler to determine which thread within them to run.

for direct memory access. This allows memory access to be fast, and also protects memory from unauthorised access through the use of flag variables within the memory object. The object oriented principle of inheritance is also used within memory management, allowing the memory object class to be an abstract class which is inherited by multiple different classes representing different types of memory - ReadOnlyMemory, DeviceMemory, etc. The modularity that object oriented design brings to the JX kernel makes it a very reliable system, as bugs in one object should not cause others to crash.

As may be expected, object orientation does also have disadvantages when applied to kernel design. The two places it suffers most are in memory management and general running time. Although the type safety and modularity of object oriented programming can be very beneficial to the management of memory, the use of a shared address space (which is almost universal amongst object oriented kernels) means that often there is some memory that is allocated to 'something' (in JX, a domain) but which is unused. In a traditional imperative kernel this memory could be paged out, but this is not possible in an object oriented kernel. In order to address this problem, Grenhall et al. used a special bitmap allocator to handle these situations, which resulted in 0.01 percent of memory being wasted at any one time - roughly half a megabyte of wasted memory in a 4GB system. Additionally, since no use is made of hardware support for memory management, some security checking (such as buffer overflow detection and null pointer detection) has to be implemented in software as well, which can be both complex and potentially a security hazard if bugs are present in the checking code. The other main disadvantage to using an object oriented approach to kernel design is speed. Most object oriented languages, including Java, have a small overhead involved with their running. This overhead means that by developing a kernel with object oriented programming we end up sacrificing speed for reliability and configurability. In terms of the JX operating system, Grenhall et al. observed that their system ran at about 40 to 100% of the speed of the Linux operating system when performing file management tasks.

Aspect oriented programming is a relatively young paradigm[14] that attempts to separate out the logic in a system from the code. The programmer writes many small programs (aspects) that 'hook' on to a component piece (or multiple pieces) of code and provide functionality. When the code is compiled a special program 'weaves' the aspects an component code together to create the final code. In this way, you can change the logic of a system quite quickly and easily without digging through mountains of irrelevant code. The paradigm defines the situation where a logical decision appears buried within other code as a 'cross-cutting' point. In the complex world of an operating system kernel there are many of these cross-cuts -

for example synchronisation, isolation, error handling, constraint checking, etc[15].

Given the number of cross-cuts that appear with a typical operating system kernel, it is not surprising that aspect oriented programming has been seen as a potential solution to the complexity of modern kernels[15, 16, 17]. In their 2001 paper, Coady et al. attempt to apply the aspect oriented paradigm to the task of prefetching mapped files in the FreeBSD kernel[16] - a feature for which the implementation code is normally contained within multiple different files across the kernel. Their main aim was to show if aspect oriented programming could improve the modularity of the system. They successfully demonstrated this fact, replacing the original pre-fetching code with a set of aspects that clearly showed and encapsulated the locations where the logic cross-cut the code. An important point to note, however, is that their implementation language (AspectC, based on AspectJ[18]) was purely theoretical - they compiled their code into actual C by hand.

Another potential application of aspect oriented code that may have a wider reaching impact than merely increasing the modularity of the code is aspect-oriented embedded systems. Due to the small size and complexity of embedded systems, code reuse within this area is rare - generally developers are required to write a completely new kernel for each device. This is of course inefficient and time-consuming. In their 2001 paper Gal et al. apply the aspect oriented paradigm to implement a reusable and highly configurable operating system kernel for embedded devices[17]. They define a set of 'shell' classes which contain the base component code required for each class. These classes can then be combined and expanded using aspects, in order to create a system that is both reusable and efficient. Although they do not actually show the implementation of a full kernel in this style, Gal et al. do cover a high-level overview of how a fully configurable memory management system implemented in such a style would function. They show that the code size and running time of each solution scales directly with the complexity of the memory management system required. Only the code that is definitely needed within a configuration is compiled into the solution. Expanding this system to be able to define an entire kernel could be of great benefit to embedded system design.

The disadvantages of aspect orientated programming mostly come from its limited scope. Although useful, it does not bring many benefits outside of its reduction in complexity and its potential for creating reusable kernels. Any further benefits it brings are dependant on the paradigm that it is based on - if laid over an object oriented system, for example, one would of course get all of the advantages and disadvantages of object orientated code. A secondary, and lesser disadvantage is the learning curve for understanding the paradigm. Being quite different from any traditional style, it may be that developers would struggle to adapt to thinking in an aspect oriented

manner, which can be confusing at first.

Agent-oriented programming is related to the idea of object-oriented programming. In this paradigm, every object is actually an 'agent' - a standalone item which is defined by its beliefs, decisions, responsibilities and obligations[19]. An agent-oriented system works by having a large set of agents, each of which have their own goals to accomplish. The agents communicate and negotiate with each other as necessary to achieve their goals. This kind of system seems plausible for implementing an operating system kernel, as each of the major tasks - scheduling, memory management, I/O, etc - could be handled by a different agent. However, because this paradigm is quite new[6], few efforts have been made so far to extend the idea towards implementing an operating system kernel. Current research shows promise for this method, though. In 2009, Palanca et al. presented an outline of an theoretical agent based operating system[20]. They suggest using agents as the basic unit of execution in the system, replacing the process model that most operating systems use. In order to manage these units they introduced the idea of a kernel 'organisation', which is a group of privileged agents responsible for the well-being of the system. This organisation would have both the ability to punish agents for 'misbehaving' - being too resource intensive, attempting to access protected memory, etc - and also reward agents who behave well by, for example, favouring them when scheduling.

By using agent oriented programming, the proposed system would gain a number of positive features. It would be very reliable, as the loss of any one agent would not immediately crash the kernel (although if it was the scheduler agent or the memory management agent a full system crash may not be far behind!) It would also be very flexible, for much the same reason - agents could be activated and deactivated as needed. Other potential benefits include good networking integration, as the model of an agent is very suited to network communications, and a unified view of all system interaction - everything is either a service provider, or a service consumer (or often both.) Palanca et al. (2009) also claim that such a system would provide a performance increase over a normal operating system by cutting out the copious amounts of 'middle-ware' present in modern operating systems. However, they are quite vague on the subject and given the theoretical nature of the suggestion I feel that such a bold claim is overoptimistic.

There are also downsides to an agent oriented operating system - mainly in terms of security. Since any agent is allowed access to the processor - and quite possibly to a shared memory space - this could pose a significant security risk for the system. Although the special kernel organisation would exist to monitor and deal with 'bad' processes, it remains to be seen how effective this would be in practice. Overall, an agent oriented kernel offers

[6] Shoham first presented agent-oriented programming in 1990.

many potential advantages, but also some quite serious downsides. It would also be complex to implement, and the lack of maturity in the paradigm means that there is little prior work to rely on.

The last paradigm I will be discussing is functional programming. This paradigm first appeared in the multi-paradigm language Lisp which was invented in 1958[21] - although its strong ties to lambda calculus cause some advocates of functional programming to claim that it predates computers by many years! The main feature of functional programming is the referential transparency of the paradigm - it is a stateless paradigm, where for the same input a function will always give exactly the same output. This contrasts with the traditional imperative style which often concentrates on changing the state of a function (usually by assigning values to variables) when it is called[22]. The main advantage of this paradigm is the simplification it provides - the outcome of calling a function is guaranteed to be based purely on the inputs and nothing else. Due to the potential benefits offered by the use of functional programming, many attempts have been made at creating functional-based operating systems. Notable examples are the Nebula[23], Kent Applicative[24], Hello[25] and House[26] operating systems.

Type and memory safety are two of the basic advantages of using functional programming to implement an operating system kernel. The former is especially useful as it helps both when writing the kernel and when attempting to prevent cross-contamination (malicious or otherwise) of kernel functions. In their 2005 paper, Hallgren et al. present further advantages and disadvantages for the use of the paradigm[27] in operating system design. Since the kernel must be able to interact directly with the low level hardware in a system, they construct their own monad[7] which handles virtual memory management, process execution protection and low level I/O operations. They also use the mathematical nature of functional programming in order to reason about the kernel they are designing, using a programming logic named P-Logic. The advantages of their application of functional programming to kernel design are numerous. Memory safety is accomplished with little effort, by distinctly typing different kinds of memory - physical memory, virtual memory page maps, programmed I/O ports and memory mapped I/O locations are all defined as different types. Concurrency is similarly dealt with, by using the inbuilt multi-threading properties provided by the compiler[8].

Although the implementation by Hallgren et al. covers many of the necessities for an operating system kernel, it does contain flaws. The safety of the system, in terms of malicious user processes, is constrained to protecting

[7] No description of monads will be given here. Readers are advised to refer to Wadler's *Monads for Functional Programming*[28] for an overview.

[8] Hallgren et al. used the GHC compiler[29].

the kernel heap from random writing that would cause corruption - and even this protection can be bypassed when using I/O devices that are DMA enabled. There are no security measures proposed against malicious user processes not targeting the kernel. Using the basic form of concurrency offered by the compiler is also flawed, as Haskell code can only be paused at 'safe points', which means that interrupts cannot be instantaneous. An alternative form of concurrency control is presented in the paper which would fix this flaw. Finally, they did have to rely on a small amount of imperative C code to capture some of the low level operations and data structures - context switching relies heavily on this imperative code to store the necessary contextual data. However, despite these flaws, the application of functional programming to kernel implementation shows great promise. Recently the House operating system was extended to include a lightweight concurrency system and a priority-based scheduling system[30], which solves some of the problems that Hallgren et al. described in their paper.

So far I have examined many programming paradigms and shown that although they all have disadvantages[9], most show at least some promise for implementing an operating system kernel. This of course now begs the question - if such alternative paradigms are suitable, then why are they not used in any major operating system kernel? It is hard to give a definitive answer to this question, as it is likely that each set of developers has their own reasons, but I believe that there are two main reasons behind this lack of alternative paradigms. These are the speed of operating system development in the "early days", and the sheer cost (in terms of both money and effort) of developing a new modern operating system kernel from scratch.

Operating system kernels were originally written imperatively because when they were first created imperative programming was the most advanced paradigm in common use. Due to both hardware and user demands, operating systems expanded in functionality at an impressive rate. This fast growth rate meant that there was little time for any kernel developer to go back and rewrite their kernel using any of the new paradigms that were becoming commonplace in software development. Therefore, all new functionality was also written in an imperative style, to fit into these now decades old kernels. Although this growth did eventually slow, operating system kernels are now so large that developing a kernel that could match the functionality of a modern day operating system would be a very difficult task. As an example, one can examine either the Windows NT kernel or the Linux kernel. In 2001, the estimated size of the Windows XP operating system was 40 million lines of code[31, 32], of which a significant portion was likely the kernel code - a figure which has probably only grown since then. In terms of the Linux kernel, in 2008 a study found that it contained

[9] As do all approaches to kernel design, including imperative.

over 6.7 million lines of code[33], which is an estimated 7500 man-years of effort (using the COCOMO model[34]), or $1.3 billion[10].

This immense size and cost means that it would be difficult for either an open source project or a software company to develop a brand new kernel using any of the alternative paradigms examined in this essay. The former would struggle to find the manpower necessary to develop the kernel, and to keep up with hardware changes and user demands. The latter might find man power easier to obtain, but then would be faced with the cost of developing an operating system kernel - not to mention the difficulty of competing in a (commercial) market dominated by Microsoft's Windows NT and Apple's Mac OS. There have, however, been some commercial inroads into developing a non-traditional operating system kernel - Microsoft have been working for some time on developing a highly-dependable operating system named Singularity[13] which uses an object-oriented kernel. Although Microsoft have stated that Singularity is not meant to replace their popular (and imperative) Windows NT family of operating systems[36], work has begun recently on a top secret project named Midori, which is known to be an offshoot of the Singularity project and may be a serious attempt to update the Windows operating system kernel for the modern world of software development.

To summarise, in this essay we have seen that there are many viable paradigms for operating system kernel develop apart from the traditional imperative style. These paradigms each bring their own advantages and disadvantages to kernel design, but all show promise. The major obstacle to using them for kernel development is practicality. Modern kernels are so large and invested in an imperative design that it makes little sense commercially for any company to attempt to use a completely new paradigm, and the open source community does not have the man power to create an entirely new kernel. However, projects such such as Microsoft's new Singularity object oriented operating system and their secret Midori offshoot project show that at least one of the major players in the operating system market might be open to exploring new approaches to kernel development. Should these projects be successful, hopefully we will see more developers taking advantage of the many benefits offered by a non-traditional approach to kernel development.

[10] As an interesting aside, the costs of developing the entirety of both the Red Hat[35] and Fedora[33] Linux distributions have been estimated at $1.2 billion and $13.7 billion respectively (figures adjusted for year 2010 US dollars.)

References

[1] "Mac OS X Technology Overview", August 2009, (http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/OSX_Technology_Overview/OSX_Technology_Overview.pdf [PDF])

[2] "The Programming Languages Beacon", (http://www.lextrait.com/Vincent/implementations.html)

[3] Armstrong, D., J., "The Quarks of Object-Oriented Development", Communications of the ACM, February 2006

[4] Golm, M., Felser, M., Wawersich, C., and Kleinoder, J., "The JX Operating System", Proceedings of the 2002 USENIX Annual Technical Conference, June 2002

[5] The Genera Operating System, (http://www.symbolics-dks.com/Genera-1.htm)

[6] The Choices Operating System, (http://choices.cs.uiuc.edu/)

[7] The JNode Operating System, (http://www.jnode.org/)

[8] Doi, N., and Kodama, Y., "An Implementation of an Operating System Kernel using Concurrent Object Oriented Language ABCL/c+", ECOOP 88 European Conference on Object-Oriented Programming, 1988

[9] The Off+ Microkernel, (http://lsub.org/who/nemo/off.html)

[10] Kon, F., Carvalho, D., and Campbell, R., H., "Automatic Configuration in the 2K Operating System"

[11] Schoettner, M., Marquardt, O., Wende, M., and Schulthess, P., "Architecture of an Object-Oriented Cluster Operating System"

[12] Frohlich, A., A., and Schroder-Preikschat, W., "EPOS: An Object-Oriented Operating System", Proceedings of the 2nd ECOOP Workshop on Object-Orientation and Operating Systems, 1999

[13] The Singularity Operating System, (http://research.microsoft.com/en-us/projects/singularity/#publications)

[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., and Irwin, J., "Aspect Oriented Programming", ECOOP 97 European Conference on Object-Oriented Programming, 1997

[15] Lohmann, D., "Aspect-Oriented Operating System Design",

[16] Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., Ong, J. S., and Gudmundson, S., "Exploring an Aspect-Oriented Approach to OS Code", Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems,
2001

[17] Gal, A., Schroder-Preikschat, W., and Spinczyk, O., "On Minimal Overhead Operating Systems and Aspect-Oriented Programming", Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems,
2001

[18] The AspectJ programming language, (`http://www.eclipse.org/aspectj/`)

[19] Shoham, Y., "Agent-Oriented Programming",
Artificial Intelligence 60 (51-92), June 1991

[20] Palanca, J., Botti, V., and Garcia-Fornes, A., "Towards Organizational Aspect-Oriented Operating Systems", Proceedings of the 2009 ACM symposium on Applied Computing,

[21] McCarthy, J., "History Of Lisp", February 1979, (`http://www-formal.stanford.edu/jmc/history/lisp/lisp.html`)

[22] Hudak, P., "Conception, Evolution and Application of Functional Programming Languages", ACM Computing Survey, September 1989

[23] Karlsson, K., "Nebula: A Functional Operating System", Technical Report, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 1981

[24] Cupitt, J., "A Brief Walk Through KAOS", Technical Report 58, Computing Laboratory, University of Kent, February 1989

[25] Fu, G., "Design and Implementation of an Operating System in Standard ML", August 1999

[26] The House Operating System, (`http://programatica.cs.pdx.edu/House/`)

[27] Hallgren, T., Jones, M. P., Leslie, R., and Tolmach, A., "A Principled Approach to Operating System Construction in Haskell", ICFP 2005

[28] Wadler, P., "Monads for Functional Programming", Advanced Functional Programming, Proceedings of the Bastad Spring School, May 1995

[29] The Glasgow Haskell Compiler, (`http://www.haskell.org/ghc/`)

[30] Graunke, K., W., "Extensible Scheduling in a Haskell-based Operating System", 2010

[31] Maraia, V., "The Build Master: Microsoft's Software Configuration Management Best Practices"

[32] O'Brien, L., "How Many Lines of Code in Windows?", (http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/)

[33] McPherson, A., Proffitt, B., and Hale-Evans, R., "Estimating the Total Development Cost of a Linux Distribution", October 2008, (http://www.linuxfoundation.org/sites/main/files/publications/estimatinglinux.html)

[34] The COCOMO II Model, (http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html)

[35] Wheeler, D., A., "More Than a Gigabuck: Estimating GNU/Linux's Size", July 2002, (http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html)

[36] Stross, R., "Windows Could Use a Rush of Fresh Air", The New York Times, June 2008, (http://www.nytimes.com/2008/06/29/technology/29digi.html)

[37] Worthington, B., "Microsoft's plans for post-Windows OS revealed", Software Development Times, July 2008 (http://www.sdtimes.com/microsoft_s_plans_for_post_windows_os_revealed/32627)