Kernel designs explained

Thom Holwerda
XXXXXXX
English writing course, March 2007

Introduction

Fifteen years ago, in 1992, two heavyweights in the field of operating system design were entangled in what would become a classic discussion on the fundamentals of kernel architecture (Abrahamsen (no date)). The heavyweights in question were Andrew S. Tanenbaum, the author of *"Operating systems design and implementation"* (Tanenbaum & Woodhull, 2006) and Linus Torvalds, the then-young and upcoming writer of what would become one of the most successful operating system kernels in computer history: the Linux kernel. Like most of his colleagues at that time, Tanenbaum was a proponent of the microkernel architecture, while Torvalds, more of a pragmatist than Tanenbaum, argued that the monolithic design made more sense. The discussion gained an extra dimension because of the fact that Torvalds had studied Tanenbaum's book in great detail before he started writing the first version of the Linux kernel.

In May 2006, Tanenbaum accidentally reignited the debate by publishing an article titled *"Can we make operating systems reliable and secure?"* (Tanenbaum et al., 2006). In this article, Tanenbaum, who had just released the third major revision of his microkernel-based operating system MINIX, argues that microkernels may be making a comeback. Torvalds replied on a public internet forum (Torvalds, 2006), putting forward the same arguments used 14 years earlier.

In this article, I will try to make the 'microkernel vs. monolithic kernel' debate more accessible and understandable for laymen. I will explain the purpose of a kernel, after which I will detail the differences between the two competing designs[1]. Finally, I will introduce the hybrid design, which aims to combine the two. In the conclusion, I will argue that this hybrid design is the most common kernel type in the world of personal computers[2] today.

What is a kernel?

Every operating system has a kernel. The task of an operating system's kernel is to take care of the most basic of tasks a computer operating system must perform: assign hardware resources to software applications in order for them to complete the tasks the users want them to do. For instance, when you browse through the world wide web, your browser needs processor time to properly display the web pages, while also needing space on your hard drive to store commonly accessed information, such as login credentials or downloaded files. While it is the task of the operating system to properly spread the computer's resources across running applications, it is the kernel that performs the actual act of assigning.

You can compare it to a chef cooking a dish in a modern kitchen. The various ingredients (the computer applications) need to be prepared using kitchen appliances (the system resources) in order to form the dish (the operating system) after which this dish can be served to the people attending the dinner (the users). In this analogy, the chef is the kernel because he decides when the ingredients are put into the kitchen appliances, while the dish is the operating system because it depends on the dish which ingredients and kitchen appliances are needed. This analogy also stresses the symbiotic relationship between kernel and operating system: they are useless without each other. Without a recipe, a cook cannot prepare a dinner; similarly, a recipe without a cook will not magically prepare itself.

[1] There are kernel designs other than the monolithic kernel and microkernel (such as nano and exokernels) but they are beyond the scope of this article.

[2] In the context of this article, a personal computer is defined as a desktop or laptop/tablet computer intended for home or office use.

Differences between kernel types

An important aspect in operating system design is the distinction between 'kernelspace' and 'userspace'. Processes (each computer program is a collection of processes) run in either kernelspace or userspace. A process running in kernelspace has direct access to hardware resources, while one running in user space needs to make a 'system call' in order to gain access to hardware (Cesati & Bovet, 2003). For instance, when you want to save a document in a word processor, the program makes a system call to that part of the kernel which manages hard drive access, after which this access is granted or denied (in other words, the document is stored on the hard drive or not). Because hardware can in fact be damaged by software, access to it is restricted in the above manner.

In a monolithic design, every part of the kernel runs in kernelspace in the same address space. The definition of address space is beyond the scope of this article, but one consequence of all parts of the kernel running in the same address space is that if there is an error ('bug') somewhere in the kernel, it will have an effect on the entire address space; in other words, a bug in the subsystem that takes care of networking might crash the kernel as a whole, resulting in the user needing to reboot his system.

There are two ways to solve this problem. The first of the two is to 'simply' try to keep the amount of bugs to a minimum. In fact, proponents of the monolithic design often argue that the design itself forces programmers to write cleaner code because the consequences of bugs can be devastating. The major problem to this approach is that writing bug-free code is considered to be impossible, and the 6 million lines of code in for example the monolithic Linux kernel allow for a large number of possible bugs.

Microkernels approach the problem in a different manner in that they try to limit the amount of damage a bug can cause. They do this by moving parts of the kernel away from the dangerous kernelspace into userspace, where the parts run in isolated processes (so-called 'servers') which cannot communicate with each other without specific permission to do so; as a consequence, they do not influence each other's functioning. The bug in the networking subsystem which crashed a monolithic kernel (in the above example) will have far less severe results in a microkernel design: the subsystem in question will crash, but all other subsystems will continue to function. In fact, many microkernel operating systems have a system in place which will automatically reload crashed servers.

While this seems to be a very elegant design, it has two major downsides compared to monolithic kernels: added complexity and performance penalties.

In a microkernel design, only a small subset of the tasks a monolithic kernel performs reside in kernelspace, while all other tasks live in userspace. Generally, the part residing in kernelspace (the actual 'microkernel') takes care of the communication between the servers running in userspace; this is called 'inter-process communication (IPC)'[3]. These servers provide functionality such as sound, display, disk access, networking, and so on.

This scheme adds a lot of complexity to the overall system. A good analogy (*Microkernels: augmented criticism* (no date)) is to take a piece of beef (the monolithic kernel), chop it into small parts (the servers), put each of those parts into hygienic plastic bags (the isolation), and then link the individual bags to one another with strings (the IPC). The total weight of the end result will be that of the original beef, plus that of the plastic bags and string. Therefore, while a microkernel may appear simple on a very local level, at a global level it will be much more complex than a similar monolithic kernel.

This complexity also creates performance issues (Chen & Bershad, 1994). Simply put, the communication between the servers of a microkernel takes time. In a monolithic design, this communication is not needed as all the servers are tied into one big piece of computer code, instead of several different pieces. The result is that a monolithic kernel will generally out perform a microkernel (provided they are similar feature-wise). This explains why Torvalds chose to write Linux in a monolithic fashion; in the early '90s, computer resources were much more limited than they are today, and hence anything that could increase performance was a welcome addition.

[3] IPC serves more purposes than just the communication between servers in a microkernel. However, for the sake of clarity, this article assumes that it is in fact the only task of IPC.

As an answer to these concerns, a new type of kernel design was devised. This design combines the monolithic and microkernel design in that it has characteristics of both. It keeps some subsystems in kernelspace to increase performance, while keeping others out of kernelspace to improve stability. That part of a hybrid kernel running in kernelspace is in fact structured as if it were a microkernel; as a consequence, parts which run in kernelspace can actually be 'moved out' of it to userspace relatively easily. Microsoft Corp. has recently demonstrated this flexibility by moving large parts of its audio subsystem in the Windows operating system from kernelspace to userspace (Torre, 2005).

The hybrid design has been heavily criticised. Torvalds (2006) and Rao (2006) said the term hybrid was devised only for marketing reasons, while Mikov (2006) argues that the fact that hybrid kernels have large parts running in kernelspace outweighs the fact that it is structured as a microkernel.

I disagree with these criticisms on the basis that if system C combines aspects of both systems A and B, it is a hybrid of those two systems. As an analogy, consider the mule (the offspring of a female horse and a male ass). The mule carries characteristics of both an ass as well as a horse, and hence it is classified as a 'hybrid'.

Conclusion

While the microkernel design provides better overall stability and local simplicity than a monolithic design, these advantages do come at a price: reduced performance and increased global complexity. Monolithic kernel designs will inevitably perform better than microkernel designs (when similarly featured), while also reducing global complexity. As a compromise between the two designs, a hybrid was devised which combines aspects of both the micro and monolithic kernel.

The Windows NT kernel (used in Windows NT, 2000, XP, and Vista), which powers most personal computers today, is a hybrid kernel; the same applies to the XNU kernel used in Apple's Mac OS X. Even a traditional monolithic kernel such as Linux gained microkernel-like functionality through concepts such as loadable kernel modules[4] and FUSE ('Filesystem in userspace')[5]. Therefore, despite criticism, the hybrid design is now the most commonly used type of kernel in personal computers.

[4] While loadable kernel modules provide microkernel server-like functionality in that they can be inserted into running kernels, they do not provide any of the security and stability advantages of microkernel userspace servers.
[5] FUSE allows filesystem drivers to be run in userspace.

References

Abrahamsen, P. (no date). *Linus vs. Tanenbaum*. Retrieved on 18th March 2007, http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html

Cesati, M. & Bovet, D. P. (2003). *Understanding the Linux kernel* (pp. 303-371). Sebastopol: O'Reilly.

Microkernels: augmented criticism (no date). Retrieved on 18th March 2007, http://tunes.org/wiki/Microkernel#Argumented_Criticism

Chen, J. B. & Bershad, B. N. (1994). The impact of operating system structure on memory system performance. *ACM Symposium on Operating Systems Principles, 14*, 120-133.

Mikov, T. (2006). *Hybrid (micro)kernels*. Retrieved on 18th March 2007, http://www.realworldtech.com/forums/index.cfm?action=detail&id=66598&threadid=66595&roomid=2

Rao, S. (2006). *Hybrid (micro)kernels*. Retrieved on 18th March 2007, http://www.realworldtech.com/forums/index.cfm?action=detail&id=66596&threadid=66595&roomid=2

Tanenbaum, A. S., Herder J. N., Bos, H. (2006). Can we make operating systems reliable and secure? *Computer, 39* (5), 44-51.

Tanenbaum, A. S. & Woodhull, A. S. (2006). *Operating systems design and implementation, third edition*. Upper Saddle River: Prentice Hall.

Torre, C., (2005). *Going deep: Vista audio stack and API* [video interview]. http://channel9.msdn.com/Showpost.aspx?postid=145665

Torvalds, L. (2006). *Hybrid kernel, not NT*. Retrieved on 18th March 2007, http://www.realworldtech.com/forums/index.cfm?action=detail&id=66630&threadid=66595&roomid=2