About Operating Systems, Abstractions and APIs

Adrian Kosmaczewski

2007-12-15

Charles Petzold, in its book "Code", states the following:

> In theory, application programs are supposed to access the hardware of the computer only through the interfaces provided by the operating system. But many application programmers who dealt with small computer operating systems of the 1970s and early 1980s often bypassed the operating system, particularly in dealing with the video display. Programs that directly wrote bytes into video display memory ran faster than programs that didn't. Indeed, for some applications - such as those that needed to display graphics on the video display - the operating system was totally inadequate. What many programmers liked most about MS-DOS was that it 'stayed out of the way' and let programmers write programs as fast as the hardware allowed.

(Charles Petzold, "Code", pages 332 & 333)

This paragraph shows the state of things during the MS-DOS & early Windows versions timeframe (from late 1970s until 2000 approximately). During this time, programmers could directly access computer memory, bypassing the APIs offered by the operating system, and thus having total control of the hardware.

This shows two different trends in computer programming, one that respects the functionality offered by the operating system, and another that bypasses it. There are advantages and disadvantages to each approach, and the following paragraphs shows some of them.

The Conflict

The Apple Macintosh (1984) and the NeXT computer (1989) were among the first systems to introduce a complete API (Application Programming Interface) that completely shielded application developers from directly accessing the hardware on which the application ultimately runs. In the case of the Apple Macintosh, this API could be programmed in Pascal, while for the NeXT it was using

the Objective-C language.

The tradeoff between the MS-DOS approach and the API one can be resumed in three areas: performance, portability & maintenance, and security. The first one, considered critical in the 80s, has been one of the major factors of the success of the "compatible IBM PC" + MS-DOS platform; similar applications could ran faster than in Macintosh environments; also, a bigger number of games (which heavily use graphics) was available for that platform, and this also led to a majority of people to choose it.

Direct access to the hardware brought a first problem, of maintenance and portability; indeed, software written this way was too hard to port to different operating systems or processor architectures, since it relied heavily in the availability of certain hardware interruptions and circuitry. While, on the other hand, Apple Macintosh software created for the first version of the Mac OS (1984) could run seamlessly, without recompilation (just copy the executable and double-click on it), until Mac OS 9 (1999).

Microsoft Windows

Regarding Microsoft Windows, the situation is slightly more complicated. Windows began its life as a GUI around MS-DOS in 1985, and from its version 95 it gradually became a more independent system, but never truly becoming a multi-threaded, multi-tasking operating system. This system used to support old MS-DOS software, allowing it to run natively:

> I first heard about this from one of the developers of the hit game SimCity, who told me that there was a critical bug in his application: it used memory right after freeing it, a major no-no that happened to work OK on DOS but would not work under Windows where memory that is freed is likely to be snatched up by another running application right away. (…) They reported this to the Windows developers, who disassembled SimCity, stepped through it in a debugger, found the bug, and added special code that checked if SimCity was running, and if it did, ran the memory allocator in a special mode in which you could still use memory after freeing it.

This was not an unusual case. The Windows testing team is huge and one of their most important responsibilities is guaranteeing that everyone can safely upgrade their operating system, no matter what applications they have installed, and those applications will continue to run, even if those applications do bad things or use undocumented functions or rely on buggy behavior that happens to be buggy in Windows n but is no longer buggy in Windows n+1. In fact if you poke around in the AppCompatibility section of your registry you'll see a whole list of applications that Windows treats specially, emulating various old bugs and quirky behaviors so they'll continue to work.

(Joel Spolsky, 2004)

As you can see, direct hardware access is not only a problem for applications developers… it was one for Microsoft as well.

Simultaneously, Windows NT was started in 1988 by another team, largely composed of engineers that worked in the Digital Equipment Corporation OpenVMS system. The NT kernel features, among several distinctive characteristics, the HAL (Hardware Abstraction Layer):

> A hardware abstraction layer (HAL) is an abstraction layer between the physical hardware of a computer and the software that runs on that computer. Its function is to hide differences in hardware and therefore provide a consistent platform to run applications on.

(Wikipedia, 2006)

The NT's HAL abstracts the complete hardware beneath, and the only way to access hardware functionality is through the API. The NT team approach was radically different to that of the "classic" Windows team; they would not fix compatibility issues application per application, but would rather define an API upfront and publish it to the developers.

The NT kernel has since replaced the older 9x kernel, from version XP onwards. This shift broke many old software packages, that are not able to run properly (if at all) in the new versions of Windows. Only those programs that use the Windows API exclusively are able to run properly (I have a copy of Lotus Improv 3.1, bought in 1992, that I used to run under Windows 3.1, and that runs perfectly well under Windows XP…!)

Present Situation

Nowadays the "performance" characteristic named above has been replaced by the security concern; code that can access directly the computer hardware without permission checks (particularly in times of the Internet) can be potentially extremely dangerous: chapter 4 of the book "Hacking Exposed" (ISBN 0-0721-2127-0), by Joel Scambray, Stuart McClure and George Kurtz exposes tens of different vulnerabilities caused by direct hardware access known to Windows 95, 98 and ME - that is, the non-NT kernel. In other terms, such systems connected directly to the Internet are simply too vulnerable to be safely usable.

The conflict between calling an API or accessing the hardware directly today has been won by the API approach; consumer operating systems, at least, have taken this road and there is no turning back. The advantages are evident; the same source code base can be used to build the same application in several different platforms, lowering maintenance and support costs; platform vendors can improve performance and security reducing the impact in existing software packages; application developers can share knowledge, tips and tricks; security is built from the ground up.

On the other side, the growth in capacity of operating systems make limited

resources to appear as unlimited: memory (using paging on disk), printers (using print queues) or even screen desktop space (using multiple desktops such as the GNOME approach, or on-screen gadgets such as Expose on the Mac).

Conclusion

Currently there are APIs for graphic manipulation, such as OpenGL, that allow software such as Google Earth to run in different hardware architectures using the same code base. This is possible thanks to the higher power of today's hardware, and to the advances in operating system design, that make the overhead of API calls a non-significant portion of the overall CPU time needed to execute programs. Thus, the conflict has largely been resolved, in my opinion.

References

Charles Petzold, "Code - The Hidden Language of Computer Hardware and Software", 2000, Microsoft Press, ISBN 0-7356-1131-9 (website)

Joel Spolsky, "How Microsoft Lost the API War", June 13, 2004 [Internet], http://www.joelonsoftware.com/articles/APIWar.html, (Accessed December 14th, 2007)

Wikipedia, "Hardware abstraction layer" [Internet], http://en.wikipedia.org/wiki/Hardware_abstraction_layer (Accessed December 14th, 2007)