What Makes An Operating System "Portable"?

2000/07/09Hubert Feyrer

As an introduction the properties of a "hardware platform" are described, and it's showen that getting the same behaviour of software on different hardware platforms isn't "portability". After repeating the tasks of an operating system, it is explained what an operating system needs to provide in the lower layers to be portable. The article ends with a case study of the NetBSD operating system.

1. Introduction

A recent question on what makes some Open Source Operating System claim it's the "most portable operating system" made me think a bit about that term. The term "portable" is often used in the Operating Systems context, and I'd like to discuss this a bit.

2. Portable to what?

When speaking of a "portable" operating system, the ability to work on multiple hardware platforms is of interest. "Multi platform" is often used as a buzzword, but what exactly does it take for two platforms to be different enough to count as "different"? In the world of Microsoft Windows, differences between the Intel 80386, 80486 and Pentiun often weigh enough to count different platforms, as well as the differences between MS DOS, Windows 3.1, 3.11, 95, 98, NT, 2000, etc. do. Programmers have to care for enough differences in these operating system variants that each one can be considered a platform on it's own.

In the traditional Unix world, a platform is defined by the CPU it's using, such as the various Intel CPUs (ranging from i386 to Pentium or Sparc CPUs (from SPARC over MicroSPARC, SuperSPARC, HyperSPARC to (partly) UltraSPARC). Also important besides the CPU is the bus architecture used (such as VME, ISA, PCI, MCA, SBus, ...) as well as the word size of the CPU (usually 32 or 64bit). Differences there as well as in the general machine architecture can be a good reason for speaking of different platforms. E.g just think of the many machines the Motorola 680x0 was used in: machines using it include Commodore's Amiga as well as some Apple Macintosh models, Atari, NeXT, Sony's News workstations, various VME boards, etc. Although all these machines use the same CPU, they're still widely different through their overall system architecture, including IO and bus systems. Another example would be MIPS CPUs which are used in devices ranging from handheld PCs and PDAs to various workstations and servers from DEC and Silicon Graphics.

3. Platform vs. Implementation: Compatibility

Go and model some software after a certain specification, e.g. POSIX or the Single Unix Specification. Is that piece of software considered “portable”? Does it run on different “platforms”? Without knowing the properties of and differences between these platforms (see above), this can’t be told. Making two technical systems (software, hardware, ...) showing the same behaviour on defined input/output situations doesn’t make them portable - it just makes them “compatible”.

Each of the two systems may use it’s very own ways to conform to the specs, bearing differences in language, source layout, build system, compilers used, etc. Furthermore, nothing is said about the internal structure of the system, be it monolithic, modular, object oriented, etc.

4. Portability

It is important to know the details of the implementation when considering software portability. If source code for something can be used on several different platforms, and it still behaves the same on all platforms, the implementation can be considered “portable” to multiple platforms.

If you have, say, some software system named "Linux" run on two different platforms - let’s take i386 and a IBM 390 - do you know "Linux" is portable because the software system behaves the same on both platforms? No. But you can say the implementation on both platforms are compatible. You will have to look at the system’s source code to determine if one copy of the source code can be used to produce a working system on these platforms.

5. Operating System

In order for an operating system’s source code to produce a system behaving equally on several platforms, it has to know about the difference between the platforms’ architecture, and abstract these differences enough so that a higher software layer can use it, and provide a uniform interface regardless of the underlying architecture. Or, citing the classical definition of an operating system: to manage resources, and provide an interface for using them.

One remaining question is, which services are included for a "Unix" compatible system? Resource management for memory (RAM and virtual memory), disk and network input/output are needed for hardware abstraction, expected services provided by the operating system are APIs for manging data storage, file systems, network protocols, which in turn allow application programs like text editors, compilers and web-server using these software layers.

The question whether a graphical user interface (GUI) is considered part of the operating system or the application layer is only of interest here regarding the lowest level, i.e. how the X server accesses the gfx hardware. Everything built on top of the X server conforms to the X protocol, which can in turn serve as a platform for desktop systems like GNOME and KDE, and many other applications.

Putting all the above points together, one of the main requirements for a operating system to be portable to as many hardware platforms as possible is a proper abstraction of the underlying hardware not only to the applications using that interfaces, but also in the operating system's source code to work on a maximum number of different systems with as little code redundancy as possible. E.g. use abstract bus interfaces that help in accessing (say) a PCI bus, no matter if it's the system's main bus on a big or little endian system, or if the PCI bus in question is maybe attached behind some other PCI or TurboChannel bus via a bus bridge. And building on top of that abstraction, you can (e.g.) have network, SCSI, ... cards with the same controller chip (3COM, Adaptec, ...) on cards with wildly different bus interface (e.g. PCI, ISA, PCMCIA, CardBus, ...), but still write the core logic only once, and attach it via some bus glue to the various buses. That way if the driver itself gets updated, all cards benefit from it, e.g. if you rewrite a Tulip (ethernet) or Adaptec (scsi) chip's driver, it's immediately available on all cards and bus architectures, without a need to edit any of these. Also, a proper model for allowing direct memory access (DMA) from such components is useful to not transfer data via the CPU.

Besides proper bus and DMA abstraction, other areas that need to be written with portability in mind include :

- the machine's virtual memory system
- the file system code
- interrupt handling code
- trap and system call handling code

The VM system must be designed to handle machine dependent aspects of the different memory management units (MMUs) available, as well as dealing with cache issues. The file system's low layers must know about the physical disk layout that the machine's BIOS/ROM/firmware can deal with, so that it can write data in a way that the machine can boot from it. Interrupt handling code must be prepared to block various sources of interrupts against each other, while allowing others. A possible "big-lock" blocking out all interrupts or none is often not fine grained enough, e.g. when you need to manipulate timer code from a network interrupt. Not all systems have equal interrupt priorities, though, and this needs to be remembered when designing the interrupt system. Finally, each system has it's own way to pass data between userland and the kernel, and to notify the kernel of special events in general, usually in the form of traps or system calls. The way for passing data is either defined by the system architects, or it's modeled to follow the machine's "native" operating systems' calling concention, making it easy to emulate it's binaries.

6. Case study: NetBSD

The NetBSD operating system works on 29 different machine types.

Supported machines include:

Alpha CPU based machines from many vendors, Commodore Amiga (m68k and powerpc), ARC based MIPS machines, Acorn Archimedes, 32bit (Strong)ARM machines like DNARD Shark and CATS, Atari TT and Falcon, BeBox, Cobald Qube and RAQs, HP 300, MIPS based Handheld PCs, Intel Architecture PCs, m68k based Luna workstations, Macintosh (both m68k and powerpc based), m68k based VME boards, Sony NEWS (m68k and MIPS based), m68k poweres NeXT machines, Open Firmware PowerPCs, the pc532 experimental platform, MIPS based DecStations (pmax), PReP based PowerPC machines, MIPS based SGI machines, Hitachi's Super-H family, as found in many embedded systems, m68k, SPARC, HyperSPARC and UltraSPARC base Sun machines (and compatibles), DEC VAXen, and the Sharp X68030 series.

Supported CPUs include:

Alpha, Motorola m68k, (Strong)ARM, PowerPC, MIPS, Intel, National Semiconductors, Hitachi Super-H, SPARC, VAX.

Supported bus architectures include:

ATAPI, BIbus, CardBus, EISA, I2C, ISA, MII, MicroChannel, PCI, PCMCIA, QBus, SBus, TurboChannel, USB, VME, XMI.

All these systems are compiled from one single source tree.

Can you spell portability? NetBSD.