Systems for the Real World

Antti Kantee
Helsinki University of Technology
pooka@cs.hut.fi

1. Introduction

Programming is hard and systems programming doubly so. Anyone who disagrees is either a genius or has not done any serious systems programming for a production quality environment. Implementing an algorithm correctly is possible with some effort. Doing the same in a highly concurrent environment with tight resource constraints and dealing properly with errors in all situations is extremely demanding.

As a result of the inherent difficulties of systems programming, the operating systems available today are buggy, unstable and nobody fully understands why they work. The understanding of current systems is supposed to form the basis of ”tomorrow's” computer systems, which are envisioned to be distributed, heterogeneous and even more complex than the current systems.

This work targets the problems of existing systems and tries to develop methods and subsystems which make the life of the systems programmer easier. The focal point is especially in file systems, but the results are targeted for application also to other parts of the system. The reasons for selecting an existing system and addressing concrete problems are multiple.

- It is easy to find motivation for solving problems which one finds oneself and others struggling with on a daily basis.
- The inertia of existing systems is huge.
- The amount of leg work required for making a new operating system usable in the real world is enormous.
- As a corollary of the previous, to be able to observe real world problems in a new type of system takes very long and a huge amount of effort.

The work described here is being developed in the main source tree of the NetBSD Operating System [1]. NetBSD is a real-world deployed BSD-derived open source operating system which is used in everything from desktop machines to embedded systems.

The development model of NetBSD has two advantages for this type of work. First, the inclusion or exclusion of some features is not decided by an arbitrary committee, but rather developers are equal in the sense that everyone has right to make changes to the source tree. Second, there is a single definitive development branch to which developers can make commits. This means that the current development version is always available for consumers to use, review and examine.

2. The Problem

If systems research has become irrelevant [4], existing systems should be stable and well-understood. They are not so. Systems continue to grow without really addressing the problems that make developing and maintaining them hard.

2.1. Complexity

Systems are complex. This is a fact of life. Dividing complexity into parts can be done, but they will still have complex interactions.

2.2. Copypaste

A major part of the problem is how systems software is developed: more often than not through copypaste and modification. After a few iterations it becomes very hard to determine why the newest version of the code does some things or why it even works.

In any real world general purpose operating system there are literally tens of copies of the same code around. Fixing a bug in one copy means as a practical process going over the entire source tree and fixing all the copies - some of which may have been slightly modified. This can lead to more bugs.

The core problem is that the penalties of copypaste can be observed only much later when it typically is already someone else's problem.

2.3. Non-modular Implementation

Existing systems frequently operate purely on ”internal” interfaces. This means that they communicate through C function calls and data structures. This makes it impossible for components not in the same domain to communicate.

3. Solutions and Implications

3.1. Microkernelism and Message Passing

Microkernels are an interesting approach, but they are only part of the solution. The fundamental problem does not go away because of isolation: programming is still hard and crashed servers still do not work. There is a certain degree of merit in moving non-central components out of the kernel, but overdoing it will buy no additional benefit.

File systems are an extremely central part of an operating system. Their management of the directory namespace wholly defines how we perceive the operating system.

Unfortunately, implementing a file system is one of the most difficult tasks in a system. The virtual file system interface is extremely complex because it tries to cope with all possible cases. This leads to it being fully understood by very few people with the typical implementation strategy being ”copypaste and modify until it does not crash and burn”. This in turn leads to systems which work because of luck rather than because they are correct.

Apart from file systems involving central mass media storage, for which the best place is the kernel, most file systems are better off being implemented in a safer environment. To investigate this, the *puffs* [2] userspace file systems framework was implemented for NetBSD. While work continues, the apparent lesson has been that file systems are complex no matter where they are implemented, but at least implementing them outside of the kernel is more pleasant.

3.2. Synergy

Existing OS abstractions should be respected to a certain point. However, if they are found to be suboptimal or dysfunctional, they should be changed accordingly. This is different from a typical research approach where changes tend to be isolated from the main system.

This leads to unifying interfaces. For example, kernel and user interfaces which do the same thing (e.g. locking) are frequently different for no good reason. Fewer interfaces means better understanding.

3.3. Testing and Development

Open source software is too frequently tested by trying out the most common code path. The reason is simple: testing is not as fun as coding.

As the frequent code paths are covered by just checking if code runs as in the above example, tests should be more interested in what happens in the presence of errors. A simple way to combine unit testing and integration testing is to run the implementation under test in an environment where it is easy to inject faults into various backend routines.

The Runnable Userspace Meta Program (rump) [3]environment is a start at providing an environment for easy kernel development and fault injection. It provides the ability to run kernel code in userspace, although it is currently limited to file systems. The difference to classic test harnesses is the option to seamlessly integrate the implementation under test with a running system and therefore test real workloads with real applications instead of artificial ones.

3.4. Security Considerations

Exporting kernel interfaces outside the kernel brings consequences to how operating system security models work. From the implementation perspective the most interesting challenge is ”bulletproofing” the bottom of an existing kernel in a similar way to how the system call layer protects all incoming requests - especially file systems can exhibit complex interactions between return values from different operations. This may call for semi-formal methods to achieve certainty of correct operation.

4. Impact and Conclusions

The work described in this document is above all a proposal for systems programmers to be able to sleep better and longer. The general approach, as to opposed to designing a new system which meets the new demands is to take an existing system and start modifying it while solving existing problems, and keeping real world usability closely in mind.

References

[1] The NetBSD Project. *http://www.NetBSD.org*.

[2] A. Kantee. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon 2007*, pages 29–42.

[3] A. Kantee. Runnable userspace meta programs, 2007. http://www.NetBSD.org/docs/puffs/rump.html.

[4] R. Pike. Systems software research is irrelevant. August 2000. http://www.cs.bell-labs.com/cm/cs/who/rob/utah2000.ps.