The Flux OS Toolkit: Reusable Components for OS Implementation

Bryan Ford Kevin Van Maren Jay Lepreau Stephen Clawson Bart Robinson Jeff Turner[†]

Department of Computer Science, University of Utah
Salt Lake City, UT 84112

oskit@cs.utah.edu http://www.cs.utah.edu/projects/flux/

Abstract

To an unappreciated degree, research both in operating systems and their programming languages has been severely hampered by the lack of cleanly reusable code providing mundane low-level OS infrastructure such as bootstrap code and device drivers. The Flux OS Toolkit solves this problem by providing a set of clean, well-documented components. These components can be used as basic building blocks both for operating systems and for booting language run-time systems directly on the hardware. The toolkit's implementation itself embodies reuse techniques by incorporating components such as device drivers, file systems, and networking code, unchanged, from other sources. We believe the kit also makes feasible the production of highly assured embedded and operating systems: by enabling reuse of low-level code, the high cost of detailed verification of that code can be amortized over many systems for critical environments. The OS toolkit is already heavily used in several different OS and programming language projects, and has already catalyzed research and development that would otherwise never have been attempted.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

[†] U.S. Department of Defense

1 Introduction

As operating system functionality continues to expand and diversify, it is increasingly impractical for a small group to implement even a basic useful OS core—e.g., the functionality traditionally found in the Unix kernel—entirely from scratch. Furthermore, in most research domains, only a few specific areas provide fodder for interesting research topics. For example, from reading an OS conference proceedings, one might be given the impression that building an OS "only" involves writing a virtual memory system, an IPC system, a file system, a scheduler, some fast local-area network code, and a profiler to produce nice bar charts. However, as any experienced OS builder knows, many of the problems involved in building an OS have already been solved countless times, and just aren't interesting to researchers. For example, any realistic OS, in order to be useful even for research, typically includes boot loader code, kernel startup code, various device drivers, kernel `printf` and `malloc` code, a kernel debugger, etc. A research project starting a new OS completely from scratch would invariably spend at least the first six months simply writing such infrastructure "grunge" before even starting on the interesting aspects of the OS.

1.1 Related work

Most OS researchers have realized this problem of high startup cost, and have resorted to cannibalizing BSD, Mach, or other freely available OSes rather than reinventing the wheel. Mach used BSD, Linux [13], and vendors' device drivers; SPIN [3] uses device drivers from FreeBSD; and VINO [17] takes its device drivers, bootstrap code, and low-level support for virtual memory from NetBSD.

While this approach saves time, the developer must manually examine and dissect the old OS; it would save even more time if the developer could simply obtain a set of clearly-documented components. It would also enable research by those whose primary expertise is in areas other than operating systems, e.g., programming language researchers who wish to explore the effects of higher-level languages running directly on the hardware. This is the purpose of the Flux OS Toolkit.

Recent research projects such as Exokernel [6], SPIN [3], and VINO [17], focus on creating *extensible systems* which allow applications to modify the behavior of the core OS to suit their particular needs. However, these systems still define a particular, fixed set of "core" functionality and a set of policies by which the core can be used and extended. The OS Toolkit, in contrast, makes no attempt to be a useful OS *in itself* and does not define any particular set of "core" functionality, but merely provides a suite of components from which real OSes can be built.

Other approaches involved creating an operating system built from a complex object-oriented framework, such as in the Choices[5] or Taligent [15] work. Although such efforts have been influential in other OS projects, such as Spring, they do not appear to have been widely used. In contrast, the OS Kit exhibits a less ambitious, but more pragmatic, and we believe more effective, approach to software design and re-use. Gabriel distinguished two approaches to software design and implementation, sardonically labeling them "The Right Thing" and "Worse is Better" [12]. The former is characterized by interface perfection at the cost of implementation complexity (e.g., Lisp with CLOS), whereas the latter sacrifices interface elegance and completeness for simplicity of implementation (e.g., Unix and C). Gabriel makes a strong case that "Worse is Better" is the more sucessful approach, and we believe that the OS Kit exemplifies this lesson.

1.2 Historical genesis of the OS Toolkit

We followed the cannibalization approach in our own OS research for some time. However, starting in 1995, that approach gradually evolved, resulting in what became the Flux OS Toolkit, or "OS Kit". Because we were finding our version of Mach [11] too constraining a vehicle in which to prototype new ideas, we developed a series of experimental kernels to try out ideas before designing our Fluke kernel [10]. In doing so, we gradually modularized and formalized the libraries of support code we developed, prototyping the OS Kit along the way. These experimental kernels embodied radical changes to fundamental aspects of OS structure which would have been impossible to explore in an existing operating system. One of these kernels explored implementations of high performance kernel-mediated capabilities and IPC paths, and took about 2 weeks to develop from scratch; the other explored interruptibility of kernel operations at arbitrary points (finding a more final expression in Fluke's atomic operations [18]), which took only a month. We have found this ability to prototype radical designs in a "real" kernel to be crucial to choosing designs that are worth fully developing.

2 Toolkit design

The Flux OS Toolkit is a framework and set of modularized library code, together with extensive documentation [9] for the construction of operating system kernels, servers, and other core OS functionality. The goal is for developers to take the OS Kit and immediately have a starting point for investigating "real" OS issues such as scheduling, VM, IPC, file systems, or security. Researchers in programming languages for systems software benefit as well, as the toolkit makes it easy to run language systems on the bare hardware.

The intention of this toolkit is not to "write the OS for you"; we certainly want to leave the OS writing to the OS writer. The dividing line between the "OS" and the "OS Toolkit," as we see it, is basically the line between what OS writers *want* to write and what they would otherwise *have* to write but don't really want to. Naturally this will vary among different OS groups and developers. If you really want to write your own x86 protected-mode startup code, or have found a clever way to do it "better," you are perfectly free to do so and simply not use the corresponding code in our toolkit. However, the OS Kit is modular enough so that you can still easily use *other* parts of it to fill in other functional areas you don't want to have to deal with yourself (or areas that you just don't have time to do "yet.")

As such, the toolkit is designed to be usable either as a whole or in arbitrary subsets, as requirements dictate. It is useful not only for kernels but also for other OS-related programs, such as boot loaders or servers running on top of a microkernel.

While the OS Kit currently runs on x86 PCs, it is designed to be portable to other architectures, and most of the OS Kit's exported interfaces are architecture-neutral. Most of the heavily architecture-specific aspects of the OS Kit are isolated in a single component, the low-level kernel support library, whose purpose is to provide easy access to the raw privileged-mode hardware facilities without adding overhead or obscuring the underlying abstractions. For example, on the x86, the kernel support library includes functions to directly create and manipulate x86 page tables and segment registers. Other OS Kit components can, and often do, provide higher-level architecture-neutral facilities built on these low-level mechanisms; however, the architecture-specific interfaces always remain accessible in order to provide maximum flexibility.

3 Sample components

The toolkit currently contains fifteen major libraries, ranging from uniprocessor and multiprocessor bootstrap code, through memory management, to support for popular file systems and disk partitioning schemes.[1] In the following sections we briefly describe some of these components.

3.1 Kernel bootstrap support

One of the time-honored ways to waste time in a research project is to write a boot loader for a new OS: as an inviolate rule, each new OS must have its own boot loader, and that boot loader must be incompatible with those of all other operating systems. Furthermore, each OS often has *several* boot loaders: one to boot from the hard disk, one to boot from across the network (this "one" often multiplied by the number of distinct Ethernet cards to be supported), one to boot from an existing OS such as MS-DOS, etc.

While searching for a good bootstrap solution for our own OS research, we examined the bootstrap mechanisms of a number of existing systems, and found that the diversity of existing mechanisms was caused not by any fundamental difference in the bootstrap service required by the OS, but instead merely by the completely ad hoc way in which boot loaders are typically constructed. In other words, *because* boot loaders are so fundamentally uninteresting, OS developers felt compelled to produce a minimal quick-and-dirty design, which results in this boot loader being unsuitable for the *next* OS due to slight differences in design philosophy or requirements.

To solve this problem, we worked with key people in various other OS projects to produce the *MultiBoot standard* [8] for x86 PCs, which is a standard interface between a boot loader and an OS so that any compliant boot loader can load any compliant OS. This standard interface includes features needed by advanced systems but typically not cleanly supported by existing boot loaders, such as support for boot images of unlimited size and boot images consisting of multiple distinct files. We then incorporated all the necessary support code into the OS Kit to make it trivial to create MultiBoot-compliant OS kernels, and included a set of simple MultiBoot-compliant boot loaders. A more complete and powerful MultiBoot-compatible boot loader, GRUB [4], is also available as a separate package. The result is that, using the OS toolkit, writing a "Hello World" OS kernel that boots from standard boot loaders is as easy as writing an ordinary "Hello World" application in C.

The OS Kit also provides the necessary code to initialize and start multiple processors in a symmetric multiprocessing (SMP) system. Of course, it is still up to the OS writer to make the overall OS SMP-safe.

For convenience, some parts of the OS Kit, such as its default console I/O and debugging support, are designed to be automatically SMP-safe; other parts of the OS Kit can easily be made SMP-safe but require the OS writer to provide the appropriate synchronization mechanisms at the individual component level. Since the components do not generally contain fine-grained synchronization internally, higher-level components, such as the file system and the networking code, will work, but not exhibit optimal parallel performance. However, the low-level components are small enough to provide an appropriate level of granularity in typical situations.

3.2 Memory management

Another aspect of OS implementation that often involves a large amount of uninteresting work is physical memory management. Many research operating systems supporting virtual memory start out simply by keeping free physical pages on a list; systems that *don't* support virtual memory typically use a simple `malloc`-like allocator of some kind. Unfortunately, in practice, all hopes of using such clean, simple solutions are quickly dashed as soon as the unsuspecting OS attempts to support real hardware, which invariably proves to be painfully picky. For example, devices often require the use of contiguous physical memory blocks larger than a page in size, requiring the VM system to scrounge through page lists for contiguous pages. Even less well-behaved devices are extremely common: many DMA devices on PCs require contiguous buffers in the lowest 16MB of physical memory. In general, operating systems must efficiently manage address spaces of all types, such as virtual address spaces, paging spaces, block or page maps, etc.; these are precisely the kinds of grimy issues that OS researchers don't have time to worry about, but must be solved if the OS is ever to become "real" in any sense.

To address these memory management issues, the OS Kit includes a set of simple, but extremely flexible, memory management support libraries. The *list-based memory manager*, or LMM, provides powerful and efficient primitives for managing allocation of either physical or virtual memory, in kernel or user-level code, and includes support for managing multiple "types" of memory in a pool, and for allocations with various type, size, and alignment constraints. The *address map manager*, or AMM, is designed to manage address spaces that don't necessarily map directly

[1] The OS Kit currently includes the following libraries: low-level kernel bootstrapping and support, multiprocessor support, a list-based memory manager, an address map manager, a minimal C library, memory allocation debugging, disk partitioning, file system reading, program loading, a math library, device drivers, the NetBSD Fast File System, and the FreeBSD and *x*-kernel network protocol stacks.

to physical or virtual memory; it provides similar support for other aspects of OS implementation such as the management of processes' address spaces, paging partitions, free block maps, or IPC namespaces.

3.3 Minimal C library

Mature OS kernels typically contain a considerable amount of code that merely reimplements basic C library functionality such as `printf` and `malloc`. This is done because the "real" C library implementations of these functions make too many assumptions about the surrounding environment and are not flexible enough to work in a kernel environment. For example, the standard C library's `printf` includes a mass of complicated buffering code, which uses many different system calls, terminal-related `ioctl`s, and dynamic memory allocation, when all that the kernel really needs is simple formatted console output. Similarly, standard `malloc` implementations make fundamental assumptions about the layout of a process's address space, e.g., that the heap is arbitrarily growable using `sbrk`, and will always be contiguous and monotonically increasing.

The OS Kit includes a minimal C library that provides common C library routines without all the unnecessary frills and unwanted assumptions in standard C libraries. For example, locales and floating-point are not supported, and the standard I/O calls don't do any buffering, relying instead on the underlying read and write operations provided by the OS. The C library routines are highly modularized and well-separated, preventing the entire library from being linked in when one function is called. Dependencies between functions are minimized, and those dependencies that do exist are well-documented, so that individual functions can be replaced as necessary in order to adapt the minimal C library to arbitrary environments. For example, `printf` relies on the OS only to provide a `putchar` implementation.

3.4 Device drivers

One of the most expensive and boring tasks in OS development and maintenance is supporting all of the different kinds of device hardware available. Devices are tricky and their glitches often undocumented; sometimes only binary versions of drivers are available. Recognizing the impracticality of providing our own device support from scratch, and the advantages of reusing others' code, we made a key extension to the approach taken in the rest of the OS Kit.

We generalized the approach taken by Goel at Columbia and Utah, which allowed unchanged Linux device drivers to be used by the Mach 3.0 kernel [13]. As illustrated in Figure 1, the OS Kit's device driver support is composed of two cleanly-separated pieces: a large base of code imported di-

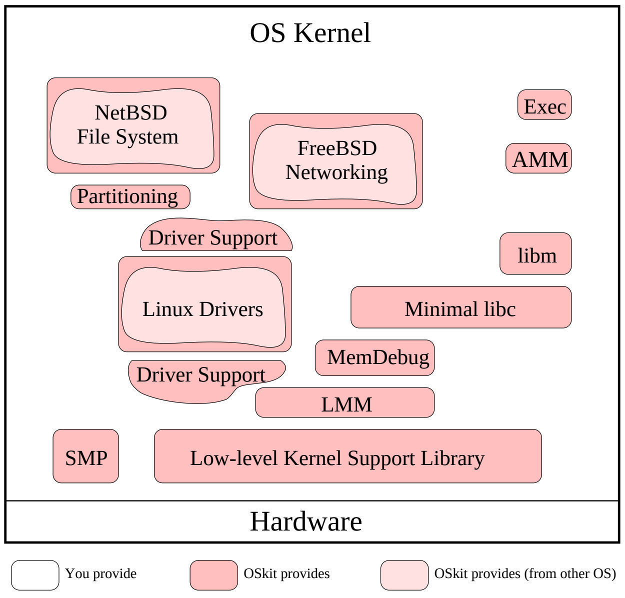

Figure 1. General OS Kit organization. Although the OS Kit's components work with each other easily, they are designed to be well separated from each other, allowing the OS to use them together or in isolation and to control how they interact with each other. Note that the relative size of the areas does not reflect the components' sizes.

rectly from an existing OS, and a small surrounding layer of "glue code" that mimics the execution environment of the donor OS. This design allows the device drivers to operate oblivious to their true surroundings in environments vastly different from those for which they were originally written, such as in preemptive or multiprocessor kernels, or even in user-mode processes. For example, in addition to providing basic functions and variables that the drivers reference, the Linux glue code also invisibly emulates Linux's current process abstraction so that the drivers can be run in environments in which the process abstraction is completely different or even nonexistent. The glue code surrounding the imported code hides the details of the original OS environment from the developer and in its place presents clean, simple, well-defined device interfaces. These device interfaces conform to a small subset of the Component Object Model [14], namely the interface querying and reference counting mechanisms, which allow them to be cleanly extended and updated over time and facilitate future binary-level compatibility.

This design required almost no modifications to the device drivers themselves, which vastly simplifies the task of keeping the drivers up-to-date with the newest versions of the donor operating system. Of course, the glue code still has to be updated to deal with changes in the drivers' overall native environment, but this is much simpler than updating all the device drivers manually. The use of binary versions of drivers, such as NetWare's ODI drivers, should also be possible, although we have not yet attempted this.

The OS Kit currently incorporates over 50 existing Ethernet and disk device drivers from Linux 2.0.29; fewer than 150 lines out of 80,000 were modified or added, mostly in header files. Some of these changes were to override macros for memory mappings and enabling/disabling interrupts; some fixed bugs that exist in Linux; and a few were added for debugging. The remainder were changes that added data structure elements required by the glue code, or defined different macros for operations; for example, we modified Linux's `skbuff` structure, which is used to store network packets.

Naturally, the flexibility provided by the framework sometimes imposes a performance cost, depending on the environment and the way the drivers are used. For instance, Linux uses contiguous buffers for network data, while many systems use more complex scatter-gather buffers, e.g., `mbufs`; thus using Linux drivers with BSD-derived networking code requires an extra copy on the send path.[2] However, even in projects intending to write custom device drivers specifically optimized for the OS in question, shrink-wrapped OS Kit device drivers are still very useful. Since OS Kit device drivers can coexist with custom drivers, they can be used as a base while custom drivers are being developed, and also to provide broader hardware coverage.

Note that although the OS Kit's device drivers would seem at first to be highly machine-dependent, several of the original Linux drivers it contains are already used on non-x86 platforms supported by Linux such as Alpha, MIPS, and PowerPC (although many others still have embedded x86 assembly). When NetBSD drivers are supported, we will gain more portability, since NetBSD has gone further in separating out machine-dependent code from the device drivers.

4 Reusable specification and verification

We are exploring one other aspect of the OS Kit. In addition to saving development time and money in OS design, the OS Kit also presents the possibility of reusable verification. Verification is an extremely expensive activity, and is usually carried out at the operating system API level. Even for an A1 security evaluation, the low-level code is not required to be formally verified, primarily due to the inordinate cost of the verification, and the fact that the infrastructure code was never or rarely reused. It is not just the security community that requires correct functioning: critical systems are being developed now to run on OSes that cannot support their safety, security, or reliability requirements. Also, many critical systems run as embedded systems whose infrastructure needs are a good match to the OS Kit's libraries. In embedded systems the OS Kit's code would constitute a higher fraction of the total code than in full-blown operating systems, which is further evidence of the value of pursuing reusable verification of OS Kit components.

[2] This important case is one reason we are integrating an alternate set of network drivers from NetBSD.

5 Current status

An early version of the OS Kit has been released publicly in beta-test form and is available from `http://-www.cs.utah.edu/projects/flux`. The OS Kit currently consists of about 3,500 public header file lines and 220,000 lines of code. Of these, 207,000 lines are reused virtually unmodified from existing sources, so the OS Kit's maintenance burden only consists of the remaining 13,000 lines of "native" OS Kit code, 23% of which is x86-specific. All line counts were taken after filtering out comments, blank lines, preprocessor directives, and punctuation-only lines (e.g., lines containing only a brace); the result typically runs $1/4$ to $1/2$ the size of the unfiltered code.

5.1 Existing uses of the OS Toolkit

Our Fluke microkernel [10] puts almost all of the OS Kit to use, and in fact over half of the Fluke kernel is OS Kit code. We used an early version of the OS Kit in MOSS [7], a DOS extender (a small OS kernel that runs on MS-DOS and creates a more complete process environment for 32-bit applications), which is being used in commercial products. Besides the experimental kernels mentioned earlier, we have used the OS Kit in smaller utilities, such as a specialized kernel to boot another kernel from the network.

Some of the OS Kit's external users have informed us of their efforts. At MIT, Olin Shivers et al. are investigating advanced-language operating systems and use the OS Kit to run SML/NJ on the bare hardware *as* the OS [16]. This is a goal the ML community has desired for years but until now the low-level aspects have presented too much of a barrier. The SR project at U.C. Davis [2] is exploring using the OS Kit to run SR directly on the hardware. Here at Utah, we have also ported two other languages to the OS Kit, Java and Smalltalk. The Systems and Communications Group at the University of Carlos III in Spain has employed the OS Kit in their distributed microkernel-based operating system project, named *Off* [1]. The OS Kit is also being used in the "bits and pieces microkernel" (bpmk) being developed in Finland.

6 Conclusion

The OS Kit has proven surprisingly powerful and popular, both at Utah and at external institutions, greatly aiding research and development in both operating systems and their implementation languages. The OS Kit's relatively mundane low-level components, and its provision of higher-level components through software reuse, fill crucial needs for wide classes of clients.

Acknowledgements

We thank Shantanu Goel for his important work, both at Columbia and at Utah, on the Linux device driver framework in Mach. Erich Boleyn co-authored the MultiBoot specification and developed a boot loader for it. We thank our clients for their feedback, in particular Olin Shivers, Albert Lin, and Greg Benson. Many others have made important contributions to the OS Kit, including Mike Hibler, Steve Smalley, Godmar Back, and Greg Benson. Mike Hibler and the anonymous reviewers provided helpful comments.

References

[1] F. J. Ballesteros and L. L. Fernandez. The Network Hardware is the Operating System. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1997. To appear.

[2] G. D. Benson and R. A. Olsson. A Portable Run-Time System for the SR Concurrent Programming Language. In *Proc. of the Workshop on Runtime Systems for Parallel Programming*, Geneva, Switzerland, April 1997. Held in conjuction with the 11th International Parallel Processing Symposium (IPPS'97).

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[4] E. Boleyn. GRUB – GRand Unified Bootloader. http://-www.uruk.org/grub/, 1996.

[5] R. Campbell, N. Islam, P. Madany, and D. Raila. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, Sept. 1993.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[7] B. Ford. MOSS: A DOS extender based on the Flux OS Toolkit. Available as http://www.cs.utah.edu/projects/flux/-moss/, 1996.

[8] B. Ford and E. S. Boleyn. MultiBoot Standard. Available as ftp://flux.cs.utah.edu/flux/multiboot, 1996.

[9] B. Ford and Flux Project Members. The Flux Operating System Toolkit. University of Utah. Postscript and HTML available under http://www.cs.utah.edu/projects/flux/oskit/html/, 1996.

[10] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[11] B. Ford and J. Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proc. of the Winter 1994 USENIX Conf.*, pages 97–114, Jan. 1994.

[12] R. P. Gabriel. Lisp: Good News, Bad News, How to Win Big. *AI Expert*, pages 31–39, June 1991.

[13] S. Goel and D. Duchamp. Linux Device Driver Emulation in Mach. In *Proc. of the Annual USENIX 1996 Technical Conf.*, pages 65–73, San Diego, CA, Jan. 1996.

[14] Microsoft Corporation and Digital Equipment Corporation. *Component Object Model Specification*, Oct. 1995. 274 pp.

[15] W. Myers. Taligent's CommonPoint: The Promise of Objects. *Computer*, 28(3):78–83, Mar. 1995.

[16] O. Shivers. Automatic Management of Operating System Resources. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, June 1997.

[17] C. Small and M. Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[18] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level Checkpointing Through Exportable Kernel State. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 85–88, Seattle, WA, Oct. 1996. IEEE.