OS Fundamentalism: Using XOmB for fundamental OS research

James Larkby-Lahet* Dave Wilkinson II* Daniel Mossé* Ahmed Amer†
jamesll@cs.pitt.edu dwilk@cs.pitt.edu mosse@cs.pitt.edu a.amer@acm.org

Introduction: XOmB is a novel exokernel-inspired operating system (OS) project aimed at developing the most flexible OS architecture possible as a means of building more efficient systems and building systems more efficiently. In this poster we intend to illustrate the suitability of XOmB as the basis for research on fundamental OS redesign, and as a means of making core systems software development and experimentation more accessible to a wider audience.

Our immediate aim is to build a research and education platform, but by doing so we contend XOmB offers the opportunity to develop more streamlined OS implementations for general purpose and server use. The flexibility we aim to achieve should allow the same kernel to be deployed in many scenarios, adapting the userspace if needed.

Philosophy: The design of the XOmB kernel does not impose any unnecessary abstractions, and thus avoids hiding state from the applications or incurring additional overheads (e.g. implementing a database on top of a filesystem introduces numerous inefficiencies [6]). Due to minimizing abstractions imposed by the OS, XOmB will be able to provide a software base sufficient to build new interfaces without having to tear down or re-engineer the kernel. The exokernel [2] strived to create such a flexible system, by "decoupling authorization from the actual use of a resource" and incorporating ideas from extensible systems to use downloadable code for customization of kernel behavior. The exokernel shared kernel state (for example, read-only information regarding the buffer pool) with userpace in order to remove kernel abstractions and thus the kernel information hiding.

We diverge significantly from prior efforts in our treatment of state with the XOmB exokernel. We relegate even more kernel code to userspace libraries, which are untrusted and can be customized (completely modified or replaced) on a per-application basis. We do so by extracting two key components of the exokernel: device drivers and system state. By taking advantage of device virtualization technology normally intended for hypervisors (particularly an I/O-MMU and self-virtualizing devices), our design places driver implementations in untrusted userspace, with equivalent flexibility to libOSes. The I/O-MMU is not a fundamental requirement for the XOmB kernel, only for untrusted userspace drivers. Similarly self-virtualizing devices are only needed to provide application customization of drivers.

The EROS microkernel also demonstrated the feasibility of a stateless OS kernel, but sought extreme reliability, through a novel persistent virtual memory system that forced persistence onto applications through periodic checkpointing [5]. XOmB embraces EROS's approach of removing all state from kernel, but does not force system-wide persistence and capabilities, which are against the notion of OS flexibility (any imposed abstraction reduces flexibility).

Note that, in the EROS example above, the applications' semantic knowledge can allow it to more efficiently checkpoint itself. That is, in addition to flexibility, performance is also enhanced. If an abstraction is stateful, then implementing it in the kernel (and thus imposing it on all processes) interferes with efficient flexibility by (a) adding an overhead for processes which do not use it, and (b) forcing processes that desire a different abstraction to around the existing abstraction. Therefore, **stateful abstractions should be implemented in userspace libraries**, allowing greater choice, easier, and more efficient (re)implementation.

Design: In order to minimize kernel abstractions, we build the kernel with a minimal interface that uses a single abstraction. In particular, given the power of addressing memory, our interface uses an existing mechanism, namely *virtual memory* and specifically *page tables*, as our interface with the kernel. Resources are allocated by mapping pages into the address space of a process; that is, at the request of a process the kernel updates the page tables for the process, including the permission bits of the page table entries. Once this mapping has occurred, no further kernel interaction is necessary for use of the resource (the page tables are used automatically through the hardware memory management unit–MMU). The permission bits in the page tables can be thought of as an implicit capability, as a

*Dept. of Computer Science, University of Pittsburgh
†Dept. of Computer Engineering, Santa Clara University

process/thread cannot address/use/access a resource that it does not have authorization to use (the MMU will check if the permission bits are set for that type of access and only allows access after validation). This contrasts with the explicit capability model of EROS, where a capability must be provided and checked for every access.

We show that the virtual memory interface is sufficient to manage **any** OS-managed resource. Specifically, resources can be memory (DRAM, SCM, etc), shared memory, files, devices, and other processes/threads. Clearly, regions of memory can be managed as is traditional through page tables and MMUs; this is true for both volatile and non-volatile memories. Files can be viewed as a continuous region of virtual memory, and managed likewise. Since modern devices utilize a memory-mapped IO space, we can also use virtual memory to allocate devices directly to applications for manipulation.

Prior work on Scheduler Activations and CPU Inheritance Scheduling has demonstrated that it is feasible to manage CPU scheduling and interrupts in userspace [1, 3]. The root scheduler thread is placed in the *init* process. We aim to extend this work to memory allocation as well.

Experience: The idea of this minimal stateless kernel design evolved during several years of effort, predominately done by full-time undergraduate students in their spare time for self-education. Currently, we have a downloadable 64-bit x86 kernel (`www.xomb.org`) built to make use of the latest multicore hardware from Intel and AMD. Our XOmB kernel follows our design described above, minimizing the abstractions imposed on the users [4]. As such, the XOmB kernel has **only five system calls**, which create and share memory regions and create and switch address spaces. In comparison, Linux 3.0.3 contains a minimum of 270 system calls with hundreds of additional system calls that may be enabled based upon configuration.

In addition to running some utilities we have developed ourselves, we have successfully ported the *mcf* benchmark from SPEC CPU2006 and *binutils*. Our *gcc* port is running but not yet functional. The kernel is still under development, but current efforts are aimed at eliminating what little state remains (console and keyboard state), and moving page allocation entirely into userspace. To highlight our argument regarding the usefulness of XOmB's inherent flexibility, we point to cases like that of an undergraduate student who, with zero prior knowledge of XOmB, was able to implement rebootless kernel upgrades, which swap the kernel code and then return to the running applications. This single student was able to go from no knowledge of XOmB to completion in 2.5 months.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," in *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, (New York, NY, USA), pp. 95–109, ACM, 1991.

[2] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, (New York, NY, USA), pp. 251–266, ACM, 1995.

[3] B. Ford and S. Susarla, "CPU inheritance scheduling," in *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, (New York, NY, USA), pp. 91–105, ACM, 1996.

[4] J. Larkby-Lahet, B. A. Madden, D. Wilkinson, and D. Mossé, "XOmB: an exokernel for modern 64-bit, multicore hardware," in *WSO - VII Workshop de Sistemas Operacionais*, July 2010.

[5] J. Shapiro and N. Hardy, "EROS: a principle-driven operating system from the ground up," *Software, IEEE*, vol. 19, pp. 26 –33, jan/feb 2002.

[6] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, pp. 412–418, July 1981.