# XOmB: an Exokernel for Modern 64-bit, Multicore Hardware

**James Larkby-Lahet[1], Brian A. Madden[2]*, Dave Wilkinson[1], Daniel Mosse[1]**

[1] Department of Computer Science – University of Pittsburgh
Pittsburgh, PA – USA

[2]Department of Computer Science – University of California, Santa Cruz
Santa Cruz, CA – USA

jamesll@cs.pitt.edu, madden@soe.ucsc.edu, {dwilk,mosse}@cs.pitt.edu

***Abstract.*** *We describe XOmB, an exokernel being developed to exploit the features of modern hardware. Specifically we discuss the use of large 64-bit virtual address spaces to provide a hybrid of segmentation and paging. This abstraction allows us to expose files and memory-mapped devices using the same mechanism. We also describe current work to develop a multicore scheduling mechanism that balances efficency and userspace flexibility.*

## 1. Introduction

Major operating system architectures have remained relatively unchanged for the past few decades. While there have been a number of research operating systems designed and implemented in the academic sector, their use and contributions have stayed largely in the academia [Ousterhout et al. 1988, Kaashoek et al. 1997]. Although the major monolithic systems such as Linux have seen redesigns of many of the subsystems, these re-designs are still rooted in the initial monolithic architecture and are hindered by this choice. There are a number of ways that the monolithic architecture stifles potential innovation: first, a monolithic operating system means that subsystems must exist within the kernel. This places two requirements on the subsystem code; it must both be trusted well written – the stability and security of the system depend on it. The second way that the monolithic kernel stifles potential innovation is in the difficulty of modification. Monolithic kernels have a steep learning curve as many of the subsystems are tightly coupled.

Besides just the difficulties presented by the monolithic architecture there are still a number of hurdles to overcome when trying to implement new subsystems – documentation. Unfortunately many of the existing systems have poor or non-existent documentation. In these cases it becomes a mess of reverse engineering and code study to understand the necessary components to make modifications and changes. It is because of these difficulties that we created the XOmB exokernel, an x86-64 operating system designed to be small, clean, well documented, and easy to extend and modify. XOmB is an exokernel architecture focusing strictly on security. This focus keeps the in-kernel abstractions to a minimum, leaving the kernel lightweight, easy to understand, and extensible. The majority of the subsystem code is pushed to userland where it is able to be swapped with any functionally equivalent code that a user may wish. The subsystems, otherwise known as library OSes, or libOSes, reside in userland rather than the kernel and allow for rapid development of new, potentially experimental libOSes without worry that they'll crash the

*participated in the project while an undergraduate at the University of Pittsburgh

system or make it insecure. Moreover, removing libOSes from the kernel allows multiple, possibly different, libOSes to run in parallel and provide the same service on the same system. For example, a web server application may be linked to a libOS that provides thread scheduling, while a database server on the same system may run a separate, different thread scheduler. This has the added benefit of allowing per-application optimizations and tuning.

Another boon of the exokernel design is the variety of architecture choices that can be designed for. For example, if a programmer wanted to design a new subsystem for a monolithic architecture they could, in theory, write a single, monolithic libOS to test their design. On the other hand, if a user wants to try a new system that depends on a looser architecture this is possible as well.

The last benefit of the exokernel architecture is that it provides clean divisions of code and keeps the kernel simple; this makes documentation easier and renders the code easier to work with and extend. It is our hope that XOmB can be used as a research tool for systems developers to rapidly prototype and implement new systems designs as well as act as an educational tool for systems students. Keeping the system small makes it perfect for teaching students about operating system design and implementation. The ability to add and remove libOSes at will make it a great project or hobby OS to learn on. For example, a systems professor could stub out sections of the code and provide the modified XOmB system to students as homework assignments. This would isolate the code that the students would have to work on and ensure that underlying code base is simple enough understand.

The rest of our paper is structured as follows: Section 2 presents some of our new architectural design decisions, Section 3 discusses current work and questions, Section 4 highlights future directions, and Section 5 concludes.

## 2. Exploiting 64-bit Addressing

Most modern architectures are now supporting large 64-bit virtual address spaces in order to support mapping a larger physical address space for main memory and IO. These large address spaces are controlled by the simple adaptation of paging from 32 bit systems consisting of multilevel pagetables. For instance, for 64-bit Intel and AMD x86 architectures, four level pagetables are used [AMD 2007]. Each level consists of a page sized block of pointers to the next level and eventually to the physical address of the mapped page. The nature of the tree structure is to allow for a sufficient means to map the entire 64-bit space yet allow sparsity where pages are not mapped.

With the jump from 32-bit addressing, the address space has grown exponentially from 4 gigabytes to potentially 16 exobytes. With this extra room, one can make a tradeoff of using large continuous sections of virtual space in order to more conveniently represent regions of the physical address space. In XOmB, we use theses sections of virtual memory as the abstraction for address spaces to represent files, IO spaces for devices, and shared data structures. These sections serve a similar purpose as traditional memory segments.

## 2.1. A Case for Segmentation

Segmentation is an alternative to paging that protects access to ranges of memory. A segment is simply a block of memory given by a pointer, a length, and set of permissions. These permissions will, among other things, denote whether or not a user application can access them and whether or not it is read-only.

With sufficiently large address spaces, it becomes possible to split these spaces into segments. With even the 48-bits of virtual address space given by current 64-bit x86 architectures and segments of 1 gigabyte in size, a process can have 256K segments mapped in at one time. To define a segment within virtual memory, each segment is represented by a pointer to a single page table that provides a root to a tree and may contain one or more levels. That is, segments are smaller subsections of the overall tree that maps virtual addresses to physical addresses. To allow a process access to a segment, it only needs to map in this root table, given by a single pointer, somewhere within its own overall page table. If the segment is three levels, then it would need to map the segment in a single level deep in the process page table. After it is mapped, the process has a virtual address that it can use to reference this segment.

With this scheme, the benefits of both paging and segmentation can be used simultaneously. A benefit of segmentation is that one set of permissions can define an entire range of memory and can typically be applied per process. With this scheme, since permission bits such as the read-only bit are applied to all page tables involved in a translation and override any weaker condition, one can simply mark in the page table tree of the process this permission bit on the entry that links to the root of the segment. This would mean that the segment is entirely read-only with the setting of a single bit.

The segment can be shared with this in mind. One process can have a segment mapped into its virtual address space read-only, and another can have it mapped read-write. This can happen since the permission bit to do this is outside of the segment page tables and only marked in the individual process page tables. With shared memory architectures that enforce write coherency, an added benefit is that a write by one process can be seen by the other.

A benefit of virtual memory is the flexibility of the mapping due to the small granularity of memory it can map. Typically a page is small. For x86 architectures, it is 4 kilobytes [AMD 2007]. With segments being defined by page tables, one can still mark the permission bits at the page granularity and map noncontinuous regions of physical memory. By setting the permission bits at the page granularity within the segment pagetable, regardless of how the segment is mapped in to a process, this particular page will force some behavior. For example, a segment might have a page marked read-only. Even if the system maps in this segment read-write, this page will overrule and will still be read-only. By allowing the mapping of noncontinuous regions of physical memory, a system can map arbitrary pages alongside memory mapped regions and also still employ traditional swap mechanisms at a small granularity as opposed to swapping an entire segment upon heavy memory load.

## 2.2. Byte-Addressable File System

An application of this segmented virtual memory system is a file system. It is reasonable to assume that byte-addressable persistent storage could be placed near the main proces-

sor much like DRAM is today [Baek et al. 2009]. In this case, storage would be accessible through common loads and stores of the processor. For these technologies, virtual memory would be available to map virtual addresses to pages on the storage device just as it would for DRAM.

Traditional file systems use indirect pointers to access files. These pointers are typically contained in a single disk block. Even current research into byte-addressable file systems adapt the same traditional model [Condit et al. 2009]. However, these blocks are generally pulled from disk as needed, copied into a buffer, and read before accessing the next block which would involve overhead due to the copy. Fortunately, these traditional indirect blocks serve the same purpose as a page table. Considering the storage is byte-addressable, there is no need for two models of indirection. By representing the layout of a file using page tables, the file can simply be mapped into the address space of the process which opens it where hardware will simply use as page tables the indirect blocks that have been stored alongside the data.

With this definition, a file is defined by a tree of page tables, it is represented by a pointer to the first page table, it has a length, and finally a set of permissions. Therefore, a file fits our definition of a segment. With files treated as segments, they can be mapped into processes by simply linking the process page table to the page table of the file, as mentioned previously. Writes to the file can be performed by using normal processor store instructions.

In XOmB, a working RAM file system has been implemented as described with files as segments. Every segment has a constant page level depth of 2, with a total file size of 1 gigabyte. Each directory is a file in and of itself, but can only be opened read-only. To do a file traversal, the library OS opens each directory on its own. The process is responsible for caching directory traversals. A simple shell has been written that can change the current directory and list directories. Currently, the kernel must know the directory structure in order to know a process has access to a file, but a userland process that acts as a file system server, much like a microkernel design, would be possible as well.

### 2.3. Memory-Mapped IO

One interesting consequence of segments is that they may be created to wrap memory-mapped IO for devices. This presents a clean abstraction around many devices. The kernel itself will only care to determine that a process which requests a mapping has permission to do so. After mapping the address space of the device to the process, the library OS that is linked to the application can manage the device in userland. This fits with the exokernel model of minimal abstractions, allowing userland processes as much control of the bare hardware without the kernel specifically knowing anything in particular about how to drive a device.

XOmB uses this type of segment to give the user process access to the video buffer. In this case, our file system maps ”/dev/console” to a segment which contains a 4K page of metadata allocated in DRAM that describe the dimensions of the screen and position of the cursor on a character buffer along with a mapping to the memory-mapped IO space of the video card. A userland library OS opens this file which will map in the address space of the video card into the virtual address space of the process. The library then

implements calls for printing character strings to the screen which involve simple writes using pointers to virtual memory.

## 3. Designing for Multicore Support

Our focus in designing XOmB's multicore processing support has been high-throughput processes. We plan for processes with a large number of threads of execution competing for concurrent use of a smaller number of CPUs. This M:N thread scheduling is not well addressed by userspace or kernelspace threading alone, but instead requires explicit communication between the kernel and userspace thread library, as addressed by Scheduler Activations [Anderson et al. 1991]. This work also informed some of the design decisions in the single-core Xok exokernel [Kaashoek et al. 1997]. The key point is that CPU revocation is explicit, and processes have the chance to context switch themselves at the end of a quantum.

In order to support efficient multithreading, when a system call is issued that would block in a typical kernel design, the userspace thread scheduler should have the opportunity to schedule another thread, rather than having control revoked. This has led us to an asynchronous, non-blocking system call interface. For calls that would otherwise block, we will use callbacks (often called upcalls when made from kernel to userspace) to reenter the thread scheduler and inform it to unblock a given thread when the response is ready. To further enhance efficiency, we will support batching of syscalls, an extension proposed for Xok but never implemented [Kaashoek et al. 1997]. Batching can be applied to upcalls as well.

The design decisions we have discussed are specifically intended to accommodate throughput-oriented workloads, but we must also plan to accommodate latency-sensitive and even soft-realtime processes. How will global scheduling decisions be made?

Let us first discuss an example of why global scheduling is difficult. A commonly discussed issue in multicore scheduling decisions, which can have quite an affect on low-latency and realtime processes, is CPU affinity. Affinity schedulers attempt to improve cache hit rates by rescheduling processes on the same CPU(s) as they ran previously [Torrellas et al. 1995]. However, affinity scheduling for competing parallel processes, given perfect information *a priori*, is a bin-packing problem and is therefore NP-hard [Nightingale et al. 2009].

Nontheless, scheduling decisions must be made quickly, on the fly. For general purpose systems schedules cannot be pre-computed. In order to efficiently resolve multicore scheduling complexities, and to give processes a level of control befitting an exokernel, we intend to develop a market-based CPU scheduler. Economic-inspired approaches have been proposed for a variety of resource management problems, particularly in the context of distributed computing [Yeo and Buyya 2006].

Xok, in its single and multiprocessor variants, exposed scheduling to userspace processes as a vector of quanta, which could be reserved just like other resources [Kaashoek et al. 1997, Chen 2000]. We intend to extend this approach by having processes bid on quanta, rather than reserve them. Bids will be made with a periodically refreshed virtual currency. Altering the amount of virtual currency allocated to specific processes provides a mechanism for users to express the relative priority of processes.

Bidding allows low-latency and realtime processes to bid high amounts on specific quanta which are particularly desirable. Throughput focused processes, on the other hand, can bid lower amounts on a larger number of quanta, allowing them to scavenge more time, on average, as a long as they aren't picky about when they run. By exposing the quanta for particular CPUs, processes can enforce their own cache affinity policies, if they desire, by offering higher bids to run on the same CPU in the future.

If we further use the same virtual currency in exchange for other resources, such as DRAM, this can be a powerful technique for providing feedback to the process about the scarcity of resources and could even allow algorithms to dynamically make time-space tradeoffs to improve overall performance in the face of competition for resources.

The final piece of the multicore puzzle is synchronization. To minimize the impact of critical sections, our thread scheduler uses lockfree synchronization [Massalin and Pu 1992]. We intend to use lock-free data structures, employing the x86-64 compare-and-swap assembly instruction, to reduce contention for all frequently shared data [AMD 2009].

## 4. Future Work

While we feel XOmB has tremendous potential as an educational vehicle and a convenient research platform, we believe the long term viability of the project will be greatly enhanced if becomes something more than simply another UNIX clone. We present two, potentially complimentary, future directions for the project. These extensions will exploit the advantages of the exokernel and can serve to guide our design so as to maximize userspace flexibility.

### 4.1. Virtualization Alternative

Virtual machine hypervisors and exokernel are very similar in their goals of securely multiplexing raw hardware. However, while virtualization technologies, even para-virtualization, express performance results in term of slowdown relative to the original system, exokernels have demonstrated 2-4x speedup for unmodified applications [Barham et al. 2003, Kaashoek et al. 1997]. Exokernels may be able to provide greater consolidation efficiency than virtualization while maintaining strong isolation.

There is also great potential in the use of hardware virtualization support to securely multiplex hardware allowing fully userspace drivers. This would provide the ultimate realization of exokernel principles.

### 4.2. Distributed Computing

Once we have implemented an economic allocation system within the XOmB operating system, we plan to extend this model to allow provisioning of resources between machines. Virtual currency can provide a means of resolving scarcity but also a means of accounting and billing for the use of these resources.

## 5. Conclusion

In this paper we have presented the XOmB exokernel, a small, extensible operating system focused on providing security rather than abstractions. Instead userland libOSes are

employed to fill the roles traditionally performed by the kernel. By utilizing the notion of segmentation in conjunction with virtual memory we have created a new useful abstraction that will benefit every aspect of the system from processes to filesystems.

As we continue to work on XOmB we plan to implement a number of other policies that will help us make the best use of current, and future hardware.

## References


AMD (2007). AMD64 architecture programmer's manual volume 2: System programming. Publication number 24593.

AMD (2009). AMD64 architecture programmer's manual volume 3: General-purpose and system instructions. Publication number 24594.

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

Baek, S., Sun, K., Kim, E., Choi, J., Lee, D., and Noh, S. H. (2009). Taking advantage of storage class memory technology through system software support. In *5th Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2009) co-located with ISCA 2009*.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA. ACM.

Chen, B. (2000). Multiprocessing with the exokernel operating system. Master's thesis, Massachusetts Institute of Technology.

Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. (2009). Better I/O through byte-addressable, persistent memory. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, New York, NY, USA. ACM.

Kaashoek, M. F., Engler, D. R., Ganger, G. R., Brice no, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. (1997). Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 52–65, New York, NY, USA. ACM.

Massalin, H. and Pu, C. (1992). A lock-free multiprocessor OS kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):8.

Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. (2009). Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA. ACM.

Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. (1988). The sprite network operating system. *Computer*, 21(2):23–36.

Torrellas, J., Tucker, A., and Gupta, A. (1995). Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 24(2):139–151.

Yeo, C. S. and Buyya, R. (2006). A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw. Pract. Exper.*, 36(13):1381–1419.