The Exokernel Approach to Operating System Extensibility

Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr.
MIT Laboratory for Computer Science

To provide modularity and performance, operating systems kernels should have only minimal embedded functionality. The *exokernel* is a new operating system structure that achieves this minimalist goal by letting applications manage the underlying hardware resources directly instead of hiding the hardware functionality in layers of operating system software. Applications can safely extend exokernel interfaces. By having direct access to the physical hardware applications can implement policies that incorporate application-specific knowledge.

Today's operating systems are large, inefficient, and most importantly, inflexible. In our view, the bulk of operating system woes can be solved by simply pushing the OS interface lower. To achieve this goal, we have defined a new operating system structure that safely exports only those resources defined by the underlying hardware. Our goal in the exokernel is to put abstractions traditionally implemented by the kernel out into user-space, where it is the job of application-level libraries or servers to abstract the exposed hardware resources.

By moving management of resources out into user-space, an exokernel allows aggressive customization. We see three reasons for customization. The first is to tailor policies to given hardware configurations, allowing applications to advance with the hardware. The second is to tailor policies to given applications. For example, closer integration with virtual memory management can enable distributed shared memory systems and garbage collectors to run an order of magnitude faster. The third is to improve performance by reducing the overhead of interrupts, exceptions, context switches, and cache misses, which are the primary sources of overhead in today's operating systems.

The exokernel's sole function is providing protected access to physical resources. It is responsible for allocating, deallocating, and multiplexing these resources in a secure way. Resource access is controlled through the use of 64-bit capabilities. The resources exported by the exokernel include physical memory (divided into pages), the CPU (divided into time-slices), the TLB, context-identifiers, and disk memory (divided into blocks).

The exokernel performs two non-physical functions: exception forwarding and upcalls. The exokernel hands exceptions directly to applications, enabling very efficient user-level exception handling. Upcalls implement synchronous cross-domain transfer of control. Operationally, an upcall transfers the program counter in one domain to an agreed upon value in another and installs the called domain's exception context. This mechanism is extremely light-weight (our relatively "green" implementation costs 20 instructions on a MIPS R3000) and lets applications build their own IPC semantics. For example, a client that trusts a server can let the server save and restore registers that it uses.

The exokernel can be *programmed* by down-loading application code into the kernel proper. This code is made safe by bounding wild jumps and memory operations through sandboxing. Furthermore, execution time can be bounded through either code augmentation or interrupts. The ability to download code allows applications to perform operations not possible in user-space. For example, this code can access unmapped physical memory, which is important for handling TLB-miss exceptions. Additionally, by allowing user code and state to reside behind the supervisor barrier the frequency of user/supervisor kernel crossings can be reduced.

The exokernel's low-level interface allows very flexible user-level implementations of traditionally rigidly defined OS services. For instance, since the exokernel does not enforce a particular page-table structure applications can aggressively specialize them: Since the address space of a "protected object" may be comprised solely of two or three pages it can be mapped using only a small, linear page-table, while a single 64-bit address-space subsystem may use inverted page-tables.

To summarize, the advantages of an exokernel based OS structure: (1) implementors can build ambitious systems without fighting inappropriate OS abstractions, (2) abstractions can be specialized to a given domain, (3) implementors can experiment without affecting the rest of the system and (4) applications can achieve good performance since they can utilize specialized libraries. Furthermore, as these libraries reside in the same protection domain as the application, cross-domain calls are avoided.

No other operating system allows this degree of flexibility in the definition and implementation of fundamental system objects.[1]

[1] See http://www.pdos.lcs.mit.edu for more information.