Twizzer: The Design and Implementation of a NVM Aware OS

Matt Bryson, Daniel Bittman, Darrell Long, Ethan Miller
University of California, Santa Cruz

1 Introduction

With non-volatile byte addressable memory on the horizon, it is time to consider how an operating system must be designed to work with non-volatile memory (NVM) on the memory bus. With this change in the memory hierarchy in mind, we have created a design for a future NVM aware OS, Twizzler. This design is centered around the use of objects to store both data and code, and representing these objects with 128 bit global unique identifiers (GUIDs). Objects can reference each other easily using pointers with an `id:offset` model, allowing for pointers to refer to other data objects without the need to serialize or swizzle them when objects are written out.

Persistance drives the changes we have to make. Our design uses a 128 bit address space to ensure we are not limited by the creation of objects since this results in a 3.4×10^{38} addresses. It is important to have a large address space for two reasons; one, to be able to track objects through the different layers of storage that are likely to exist on this system, and secondly, to allow for objects to be unique across machines, allowing new opportunities in networking. Using the 128 bit GUIDs as pointers is impractical so we have chosen to give objects local IDs that are resolved to GUIDs using a *Foreign Object Table*. Local IDs will be much smaller, with both the object ID and the offset fitting in 64 bits. This process is described in further detail in Section 2.

We have decided to redesign the operating system from the ground up to not only support the data model, but to more truly embody the principles of the Opal, Multics, and exokernel projects [3, 2, 4]. Because the kernel will have minimal responsibilities, most operating system functionality will be implemented in user space libraries, thereby reducing the complexity of the kernel and "getting it out of the way" of applications as they operate on data.

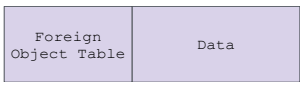

Figure 1: Object Layout

Local ID	Flags	GUID *or* Name	Resolver	Other data
Local ID	Flags	GUID *or* Name	Resolver	Other data
Local ID	Flags	GUID *or* Name	Resolver	Other data

Figure 2: Foreign Object Table

2 Data Model

Our 128 bit GUIDs are advantageous because they allows us to give each object a truly global, unique ID in a flat address space. A pointer comprised of a global object ID and an offset would be too large to fit inside a pointer without increasing pointer sizes to much larger than they are today, say a 128-bit or larger CPU. A 128 bit CPU would not be practical due to the number of design changes required and subsequent loss in performance. Instead, we choose to give objects local IDs such that a pointer is comprised of a local ID and offset, equaling 64-bits in total.

To translate local object IDs to GUIDs we use a Foreign Object Table (FOT). Objects consist of their data and an FOT, shown in Figure 1. The FOT stores the translation data of the object local IDs to GUIDs, and also stores flags, the GUID of the object, the resolution function (described in Section 3), and other data, such as permissions data. This is shown in Figure 2. This allows resolution to be flexible, and not to involve using the GUIDs as pointers directly. This data model allows for cross object pointers by resolving a GUID across several FOTs.

3 Design

We are breathing rare air. Rarely does the opportunity to change two paradigms – one in operating systems and one in networking – present itself. This opportunity comes from new non-volatile memory (NVM) technologies that

will be byte addressable. While we could make an incremental change to existing operating systems, we've decided that we have the opportunity for more. We have taken this opportunity to design a new operating system with NVM support in mind using core designs from systems such as Opal and Multics. Our design resolves around the core principle of all data being represented as objects, each associated with a 128 bit global unique ID (GUID). We choose the 128 bit identifier size to allow for objects to be truly global, as this provides an address space that is intractable to exhaust. Objects are (possibly large) blobs of data that could contain code, data, or both. This address space is large enough for frequent object creation with low probability of collision.

An *application* refers to a collection of objects operating in some manner to attain a goal (for example, a key-value store may be a collection of data objects and a code object which provides an interface). Objects may have pointers to data in any object (itself or others). Pointers are implemented as `entry:offset`, where `entry` refers to an entry in a per-object table called the *Foreign Object Table* and `offset` is the offset in the object for which the pointer is referring to. Entries in the Foreign Object Table have a GUID, options, a pointer to a resolver, and a pointer to additional data, shown in figure 2. The job of the resolver is to handle the situation where an object referred to by a Foreign Object Table entry is not present on the system. This design allows us to associate a significant amount of additional semantic data with each pointer in an object, providing extreme flexibility in data access. Because each pointer is associated with a function to resolve a non-local object, we can imagine different cross-network data access semantics being associated with each pointer, allowing applications to unify the methods with which they access data and push the details of data networked data access to an easily selectable resolution protocol, thus vastly simplifying network data access from the point of view of application developers.

4 Implementation, Questions, & Future Work

To best implement our proposed operating system, several changes should be made to the CPU. One of the most important changes would be to reintroduce segments to provide native hardware support for our addressing scheme. Additionally, increasing the virtual address space from a flat 48 bit address space to a 64 bit address space to allow for our local ID + offset pointer scheme to be implemented easier on current hardware. Hardware support for the FOT would allow for faster translation. We plan to implement two versions of this system - one that works around existing CPU limitations and one on RISC-V open architecture [1], allowing us to make hardware changes in either a physical or emulated environment.

With main memory persistent, rebooting, crashing, and restarting take on new meanings. What booting or rebooting even means in this context remains open - we see it primarily as hardware initialization since memory is no longer cleared. Additionally, returning to a safe state is more difficult when state is preserved between power cycles. It will be important to develop a process to return both the operating system and applications to a clean state in the event of failure.

5 Conclusion

There is still significant work to be done in implementing an NVM aware operating system, but we believe that we have created a roadmap for the future. We are working on a design that is tailored not only for NVM, but to what we believe the future of operating systems to be. Though our changes will work on existing hardware, we believe our suggestions for processor change to be useful to improve the utility of NVM and better implement our NVM-aware OS.

References

[1] K. Asanovi and D. A. Patterson. Instruction sets should be free: The case for risc-v. Technical report, Technical report, University of California at Berkeley, http://www. eecs. berkeley. edu/Pubs/TechRpts/2014/EECS-2014-146. pdf, 2014.

[2] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, Nov. 1994.

[3] F. J. Corbat and V. A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the November 30December 1, 1965, fall joint computer conference, part I*, pages 185–196. ACM, 1965.

[4] D. R. Engler, M. F. Kaashoek, and others. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.