Using segmentation to build a capability-based single address space operating system

Wuyang Chung

wy-chung@outlook.com

Abstract

This paper describes several old computer science technologies, i.e. single address space operating system, capability-based addressing and segmentation. These technologies combined together can create a system that is simple, secure and fast. Although segmentation is in x86 architecture, it is not suited for building a capability-based addressing system. This paper describes a segmentation architecture that is suited for building a capability-based single address space operating system. It also illustrates several other benefits that we can get from segmentation, such as software-managed TLB, performance improvement for big memory servers, IOMMU simplification, quasi-statically linked shared-library, cross-domain call, etc.

Introduction

Segmentation has been in x86 architecture for a long time but it is seldom used. Currently it is used for the per CPU data and TLS (thread local storage) data. But there are actually lots of benefits we can get from segmentation. It can be used to implement single address space operating system. It can also be used to implement capability-based addressing system. So we can get the benefits of both single address space operating system and capability-based addressing system by using segmentation. Besides that there are other benefits we can get from segmentation. With physical segment, we can implement software-managed TLB, improve the performance for a big memory sever and simplify the design of IOMMU. With I/O segment, we don't need memory-mapped I/O, so there will be no memory holes in the memory address space. For shared library, since it has its own code and data segment, there is no need for PIC, GOT and PLT and it can be quasi-statically linked during link time.
In the following section I will briefly introduce single address space operating system followed by capability-based addressing system. After that I will describe a segmentation architecture that is suited for building a capability-based addressing system. I will also describe other miscellaneous stuff related with segmentation.

Single Address Space Operating System

Most OS today use virtual address space to protect processes from each other, i.e. each process has its own private virtual address space. This multiple virtual address space model has some drawbacks. First, pointers can not be used in shared memory buffer. When multiple processes in a multiple virtual address space system want to share a memory buffer, the memory buffer must be mapped into each process's virtual address space. Most of the time it is not possible to map the memory buffer at the same virtual address for all the processes, so pointers can not be used in the memory buffer. This makes it harder to have a complicated data structure in the shared memory buffer. Other techniques must be used to represent pointers in the shared buffer. The second drawback is that TLB must be flushed during context switch if the CPU has no ASID. For CPUs with ASID, there will be duplication of translation information in TLB for the shared memory buffer. So system will spend more time on TLB miss handling.

Single address space operating system can solve both problems by running all processes on a single address space. But there is one big problem with single address space operating system, that is protection. How can the processes be protected from each other on a single address space. There are several ways to create multiple protection domains on a single address space, i.e. domain page model, page group model and capability-based addressing. Among which capability-based addressing is the most promising one.

Capability-based Addressing

In a capability-based addressing system, a process can access a memory segment only when it has the capability to that segment. Since a process can not access any memory in the system, processes can be protected from each other when running on a single address space. The most important thing in a capability system is to protect capability from forgery. There are two ways to protect capability from forgery, i.e. tagged approach and partitioned approach. The tagged approach uses extra tag bit for each memory “word” to protect capability. The partitioned approach limits the place where capabilities can be stored to protect capability. In this paper I will focus on partitioned approach because partitioned approach can be implemented by segmentation.

In partitioned approach each process has a C-list that stores a list of capabilities for a process. A process can only access the segments pointed to by the capabilities in its C-list. Figure 1 shows a capability-based addressing system using C-list.

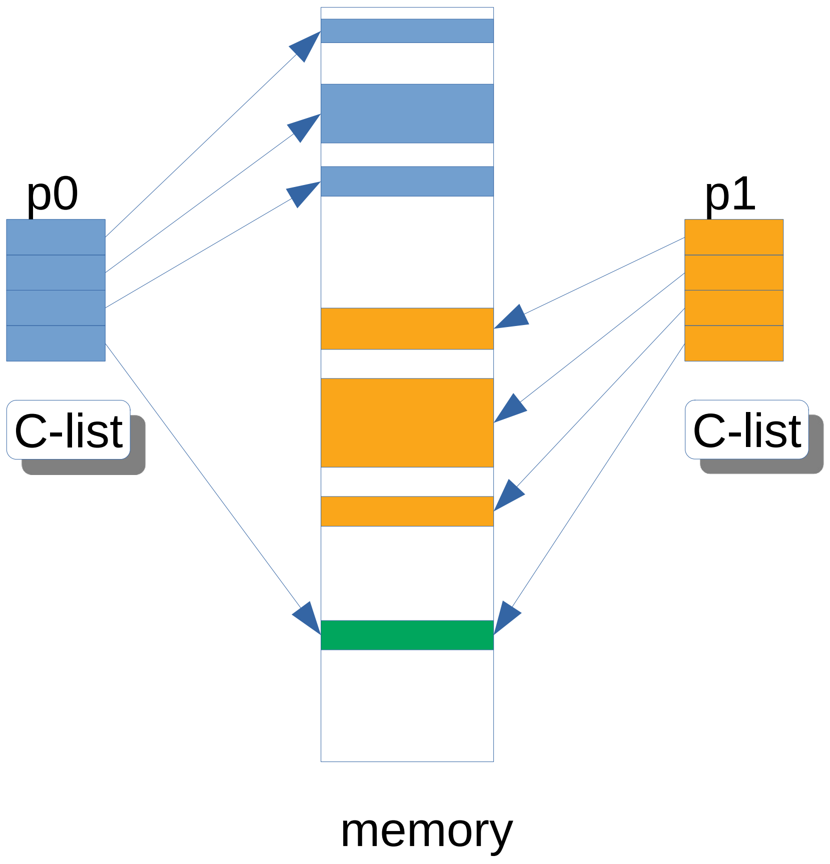

Figure 1: capability-based addressing

As can be seen processes p0 and p1 are protected from each other. Besides that they also share a segment. This picture actually looks like a segmentation architecture so the next section will describe about segmentation.

Segmentation

Figure 2 shows a proposed segmentation hardware that's suited for implementing a capability-based addressing system. By definition a capability is a permission plus an object ID. Since we are using capability to control the access to memory segment so the object ID here is a segment ID. Since the system has a segment descriptor table that describes all the segments in the system, the segment ID can be the index of the segment in segment descriptor table. Segment descriptor contains primarily a base address and a limit that describes the range of this segment. For a segmentation architecture that supports the implementation of capability-based addressing, the system must have a capability table (C-list) for each process. The threads of a process can only access the segments in the process' capability table. In Both the capability and segment descriptor, there is a 'key' field. This field is used for capability revocation. The capability is valid only when the key value in the capability and segment descriptor are equal. To revoke all the capabilities to a segment, we simply set a new random number in the key field of the segment descriptor. A thread have a default code capability (c5), data capability (c0) and stack capability (c4). There are 3 capability registers (c1 to c3) that are free for program use. Below I will describe some of the changes needed for segmentation hardware.

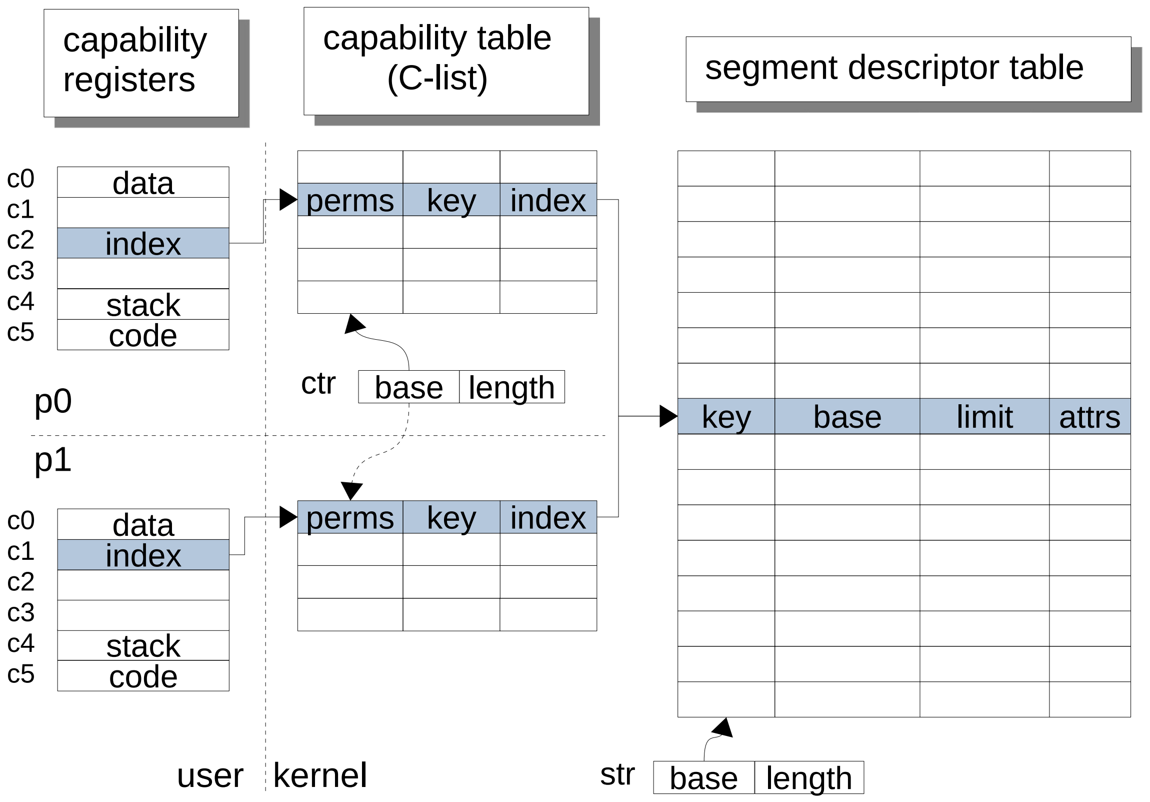

Figure 2: segmentation hardware

In order to speed up the segment translation process, two caches are needed in the CPU. First is segment TLB, the original TLB in the CPU should be called page TLB in segmentation hardware. Second is capability cache. Each time a capability handle (index to the capability table) is loaded into a capability register, it's corresponding capability is also loaded into its capability cache.

Most CPUs today have a stack that grows towards 0. This is because in the old days there can be only one thread in a process. It is flexible to have stack and heap grow in opposite direction. But with today's multi-threaded process, there is no benefit for a stack to grow towards 0 and It is even harmful in segmentation. Figure 3 shows that if the stack grows towards 0, there is no way to expand the stack when the stack pointer reaches 0. But if the stack grows towards ∞, it is usually possible to expand the stack when the stack pointer reaches the segment limit by increasing the segment limit.

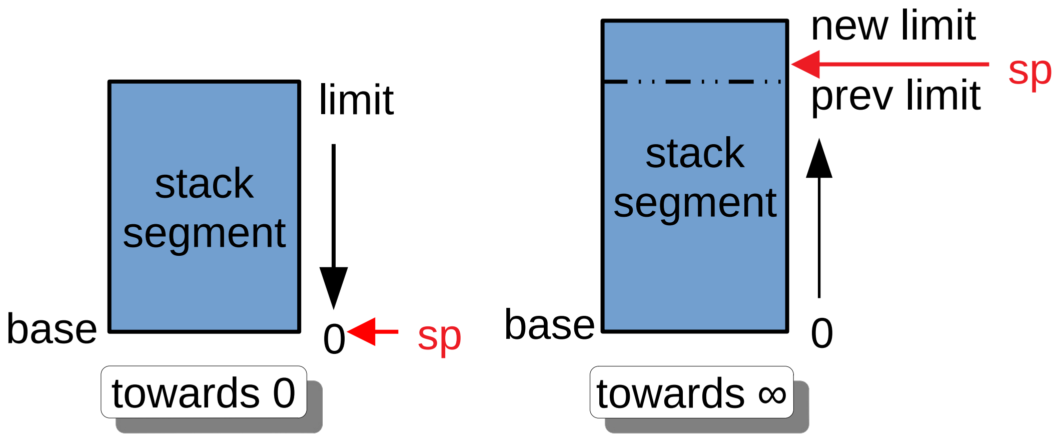

Figure 3: stack growth direction

Before the following topic, the term *far pointer* needs to be defined. In a segmentation hardware, CPU always needs to provide two information to MMU in order to access memory, i.e. segment and offset within that segment. Most memory accesses have a default segment, so program only needs to provide the offset within that default segment. But if the program wants to access data that is not in default segment, it will need to use far pointer to access that data. So a far pointer contains both segment and offset within that segment.

In a program, a data pointer might point to either global variable or local variable. i.e. point to either data segment or stack segment. In order for the program to access the data correctly, all pointers will need to be converted to far pointers. But this will make the program larger and inefficient. Since a data pointer can point to either data or stack segment, data and stack segment should share one offset space. Figure 4 shows this scheme. In the figure when the offset is bigger than 4G – 256M, the program actually wants to access the stack segment. The hardware should automatically switch to use default stack segment to access the data.

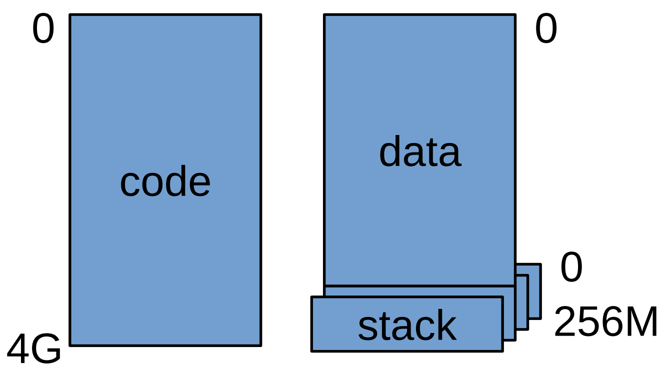

Figure 4: offset sharing for data and stack segment

In a segmentation hardware, a segment address (segment + offset) will be translated into linear address. Depending on the type of the segment, it will translate to different linear address. For a virtual segment, the linear address is a virtual address. For physical segment, it is physical address. For I/O segment, it is I/O address. If the segment has 64-bit limit, it is a long segment. There are a couple of interesting applications for physical segment. We can implement a system with software-managed TLB with physical segment. By putting TLB miss handling routine in physical segment, we can guarantee that a TLB miss will not happen during TLB miss handling. [1] describes that for a program with a big memory data with very low locality of reference, the CPU spends 51% of cycle time servicing TLB misses. We can dramatically increase the performance of that program by putting the big memory data in physical segment (called direct segment in the paper). Physical segment can also make the design of IOMMU simple. Since it maps to physical memory, there is no need for paging in IOMMU. The capability to the physical segment can also be used to protect the system memory from being accessed by malicious of malfunctioned device. A hardware device must have a valid capability to a physical segment in order to access system memory.

With I/O segment we no longer need memory-mapped I/O. All the I/O related stuff can move to I/O space, such as the device control registers, PCI configuration space, video frame buffer, boot ROM, etc. There are several advantages for I/O segment. First, there will be no memory holes in memory address space. Second, segment is more fine-grained then page. Third, one set of load/store instructions can be used to load from or store to either memory or I/O. Forth, device can only be access by a device driver that has the right capability to the device's I/O segment.

Segmentation can make shared-library simple to implement. There is no need for PIC (position independent code), GOT (global offset table) and PLT (procedure linkage table) when shared library is implemented on a segmentation hardware. Shared library will has its own code and data segment so we can know exactly the offset of all the global variables and functions in the shared library. It means that the offset part of global variables and functions can be linked in link time and the segment part of the address is linked during the program load time. So shared libraries can be quasi-statically linked on segmentation hardware.

Current operating systems confine a thread in one protection domain. It is because that it is hard to call into another domain in multiple virtual address space operating system. In single address space operating system it is easier so thread no longer need to be confined in one protection domain. This concept of thread traveling among protection domain is called cross-domain call. It can encourage modularity and improve security. It can also make resource accounting more accurate. With segmentation it can make cross-domain call easier by simply moving the stack capability from the caller domain to the callee domain.

Conclusion

Segmentation is a way to implement capability-based addressing system and capability-based addressing is a way to implement single address space operating system. So they are actually all related to each other. By using segmentation we can get not only the benefits of segmentation itself but also the benefits of single address space operating system and capability-based addressing system. In order for this idea to work, there are lots of changes needed in a computer system. First the hardware must support segmentation. Second the compiler tools must support far data, far function call, far pointer and a new way to implement shared-library. Third the OS virtual memory system must also support segmentation. For application programs, segmentation can be made transparent by operating system and compiler tools. Still it is sometimes necessary for application programs to use segmentation in order to improve performance.

References

[1] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, Michael M. Swift. Efficient virtual memory for big memory servers. In Proc. ISCA, 2013.

[2] Eric J. Koldinger, Jeffrey S. Chase, Susan J. Eggers. Architectural support for single address space operating systems. In ASPLOS-V, 1992.

[3] Gernot Heiser, Fondy Lam and Stephen Russell. Resource Management in the Mungi Single-Address-Space Operating System. Proceedings of Australasian Computer Science Conference, Perth Australia, Feb. 1998, Springer-Verlag, Singapore, 1998.

[4] Henry M. Levy. Capability-based Computer Systems. Digital Press, 1983.

[5] J. Bradley Chen, Brian N. Bershad. The impact of operating system structure on memory system performance. In 14th ACM Symposium on Operating System Principles (SOSP '93), Asheville, NC, pp. 120–133.

[6] J. Liedtke. On micro-kernel construction. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, December 1995.

[7] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single Address Space Operating System. ACM Transactions on Computer Systems, 12(4), November 1994.