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

Wuyang Chung

wy-chung@outlook.com

Abstract

There are two approaches to build a capability system, tagged approach and partitioned approach. CHERI is using the tagged approach. This paper will talk about another way to build a capability system, that is the partitioned approach. The partitioned approach can actually be implemented by using segmentation. Although segmentation exists in x86 for a long time, unfortunately it is not well-suited for building a capability system. This paper proposes a new segmentation hardware that can be used to build a capability system. Besides that we can also get a lot of benefits from using segmentation, such as simplification of TLB miss handling in guest virtual machine, simplification of IOMMU implementation, simplification of shared library, simplification of cross-domain call, etc. This paper will also explain why we need a single address space operating system and what is its relationship with capability system. At the end I will also talk about how the system can maintain backward compatibility with old multi-address-space programs.

Introduction

This paper will first briefly describe single address space operating system and the benefits of SASOS. It then describes that capability can actually be used to implement a SASOS. It then goes on to describe how the capability system can be implemented by segmentation.

Although segmentation has been in x86 architecture for a long time, unfortunately it is not well-suited to implement a capability system. Currently it is only used for the per CPU data when in kernel mode and TLS (thread local storage) data when in user mode. But there are actually lots of benefits we can get from segmentation. With virtual segment, it can simplify CPU's TLB miss walk hardware. 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 partially linked during link time.

In the following sections I will first briefly introduce single address space operating system followed by capability-based addressing system. After that I will describe the new segmentation architecture that is suited for building a capability-based addressing system. Then I will describe many of the benefits we can get form this new capability/segmentation architecture.

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. It cannot be quanranteed that the memory buffer will be mapped 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 address space switch if the CPU does not support ASID. For CPUs do support ASID, there will be duplication of translation information in TLB. For example if you are running 10 ls programs at the same time, you will have 10 copies of the the same page mappings in TLB. This will make the TLB less efficient, 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. In this paper capability-based addressing is proposed for creating multiple protection domains on a single address space.

Capability-based Addressing

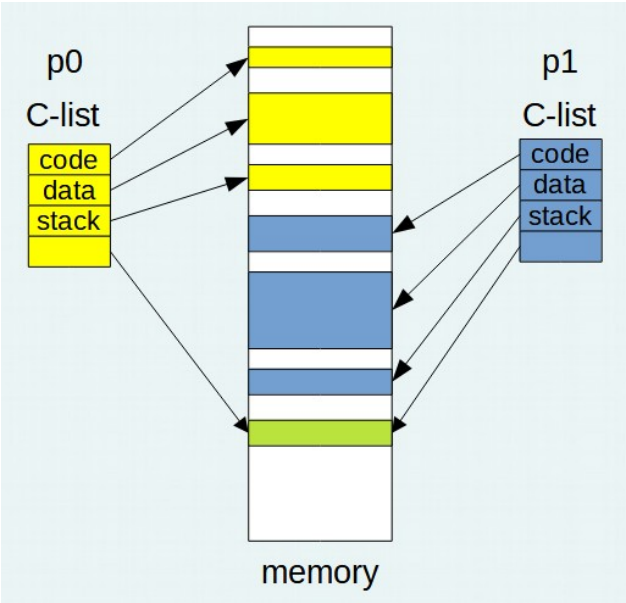

Figure 1: capability-based addressing using partitioned approach

In a capability-based addressing system, a process can access a memory segment only when it has a capability to that segment. Since a process can not access any memory in the system, processes can be protected from each other even 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. CHERI is using the tagged approach. The partitioned approach limits the place where capabilities can be stored to protect capability. In this paper I will focus on partitioned approach since partitioned approach can be implemented by using 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 partitioned approach. As can be seen processes p0 and p1 are protected from each other. This picture actually looks like a segmentation architecture so the next section will describe about segmentation.

Segmentation

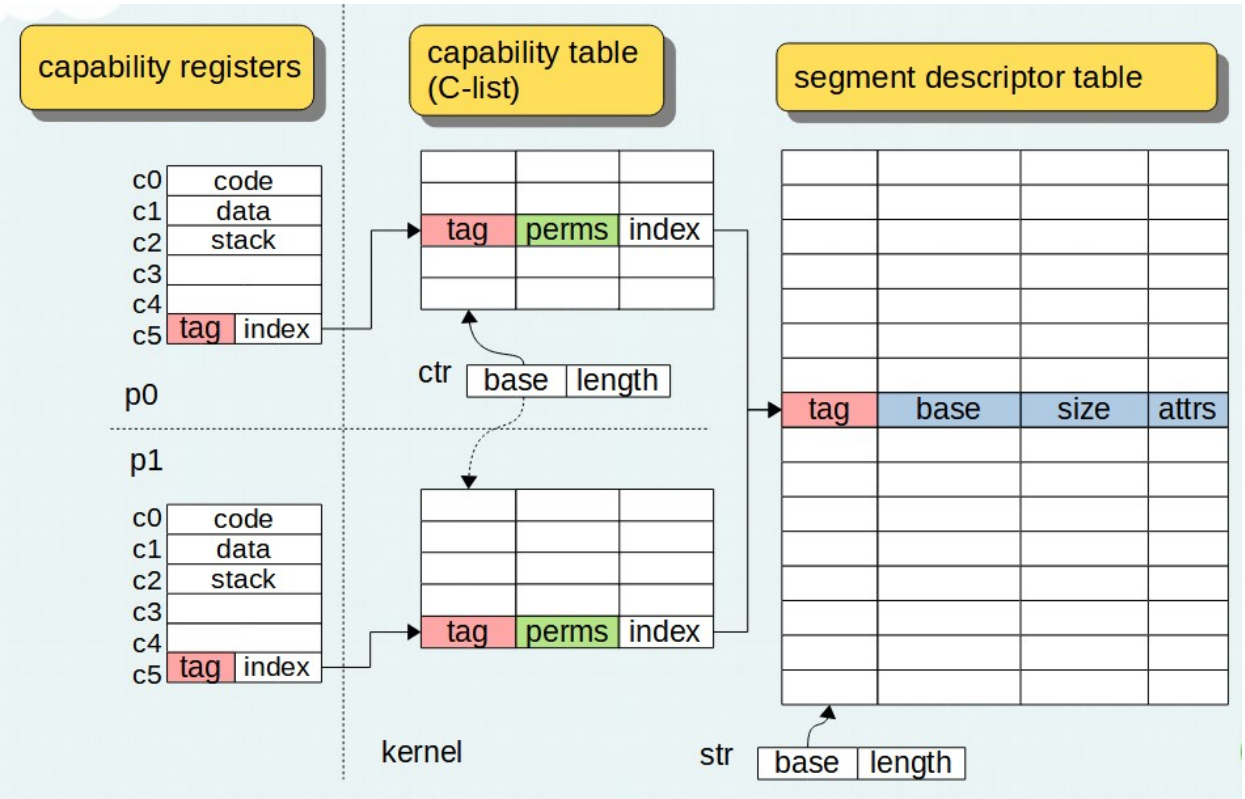

Figure 2: segmentation hardware

Figure 2 shows a proposed segmentation hardware that's suited for implementing a capability-based addressing system. It comprises a segemnt descriptor table, a capability table (i.e. C-list) for each process and a new register file called capability registers. Segment descriptor table records all the segments in the system. Each segment descriptor contains the base address of the segment, the size of the segment and some attributes. There is also a tag field. I will talk about it latter.

Capability table records a list of capabilities a process owns. By definition a capability is a permission plus an unique segment ID. Since the index to the segment descriptor table can be seen as an unique ID for a segment, the unique segment ID in the capability is actually an index to the segment descriptor table. The tag field is used for capability revocation. A capability is valid only when the tag field in the capability is equal to the tag field of the corresponding segment descriptor. When the OS wants to revoke all the capabilities to a segment, it can simply change the tag field in that segment descriptor then all the capabilities pointing to that segment are revoked.

The capability register is used to store a capability handle. A capability handle is similar to a file handle. A process can access a segment only when it has a capability handle to a capability that in term points to that segment. The tag field is used to verify if a capability handle is valid or not. The handle is valid only when the tag field in the capability handle is equal to the tag field of the correspoiding capability. A thread has a default code capability (c0), data capability (c1) and stack capability (c2). It points to the default code, data and stack segment. When a program accesses its code, data and stack, the CPU will automatically use its corresponging default capability to access the code or data. There are 3 capability registers (c3 to c5) that are free for program use.

In order to speed up the segment translation process, two caches are needed in the CPU. First is segment TLB. It is used to cache the recently used segment descriptors. The original TLB in the

CPU should be called page TLB since it is used to cache recently used page table entries. Second is capability cache. Each capability register has a cache. Each time a capability handle is loaded into a capability register, it's corresponding capability and segment descriptor 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. 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 maximum segment size by enlarging the segment size.

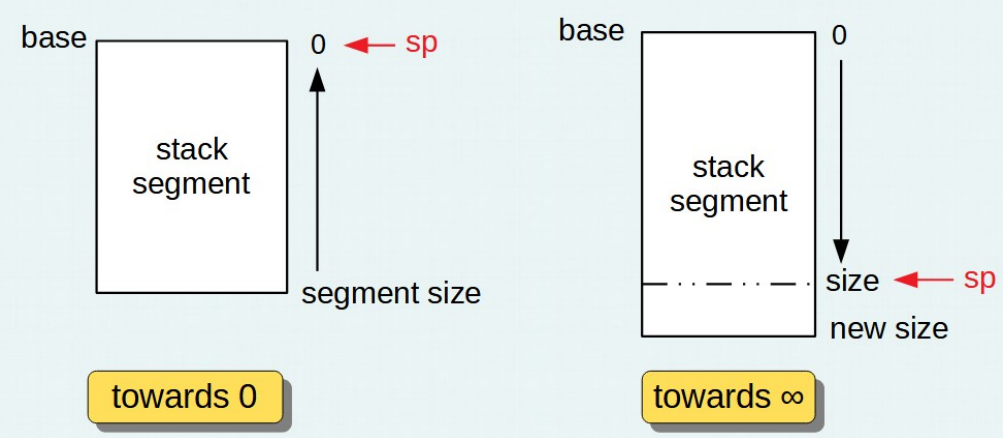

Figure 3: stack growth direction

A capability is basically a pointer to a segment, so capability and segment are almost equivalent. In the following sections I will use capability and segment interchanably in order to make the idea more easy to understand. Before continuing the following sections, the term *far pointer* needs to be defined. In a segmentation hardware, CPU always needs to present 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 segmentation hardware, a logical address (segment + offset) will be translated into a linear address by the hardware. 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 a physical address. For I/O segment, it is a I/O address. That is, a virtual segment will translate a logical address to a virtual address, a physical segment will translate it to a physical address and a I/O segment will translate it to a I/O address. If the segment size is 64-bit in the segment descriptor, it is a long segment. The next section will talk about the benefits of using segmentation.

Benefits of Virtual Segment

There are a couple of benefits we can get from virtual segment. The first is that segment move for virtual segment is very fast. It is because when moving a virtual segment we only need to copy the memory mappings instead of copying the data itself. Also we don't need to fix any pointers in the segment after the segment is moved. The second benefit of virtual segment is that it can make the TLB miss walk in a virtual machine a lot simpler. Curently on a CPU that supports virtulization,

when a TLB miss happens in the virtual machine, the TLB miss walk will need to do two virtual to physical address translations. It has to first translate the guest virtual address to guest physical address. Since a guest physical address is actually a system virtual address, the hardware will then need to translate again from system virtual address to system physical address. With virtual segment the CPU may only need to do one translation. How is it possible? By having guest's capability point to system's virtual segment, the segmentation hardware can then translate the guest logical address directly to system virtual address. Of course this can only happen on para-virtualization. When the guest OS wants to allocate a virtual segment, it will send a segment create request to hypervisor. Hypervisor will then create a virtual segment and return a capability to this segment to guest OS. Since the capability points to a system's virtual segment, the segmentation hardware will translate a guest logical address directly to system's virtual address. So when TLB miss happens on guest machine, the CPU only needs to walk the page table hierarchy once to translate system virtual address to system physical address.

Benefits of Physical Segment

There are a couple of interesting applications for physical segment. We can implement a system with software-managed TLB with physical segment. To run TLB miss handling routine in OS, it must be quaranteed that another TLB miss will not happen when that routine runs. By putting TLB miss handling routine in physical segment, it can be guaranteed. In the paper [1] it describes that for a program with a big memory data, the CPU spends 51% of cycle time servicing TLB misses. They proposed a idea called direct segment which will bypass TLB when doing virtual to physical address translation. By using direct segment, the performance of the program can be improved dramatically. The idea of direct segment is that when CPU accesses a virtual address that falls within a specified range, CPU will use simple addition and subtraction operations to translate virtual address to physical address. This is exactly what physical segment do. By putting the big data in physical segment, it will have the same result. Physical segment can also make the design of IOMMU simple. Since it translates directly 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 or malfunctioned device. A hardware device must have a valid capability to a physical segment in order to access system memory.

Benefits of I/O Segment

The purpose of I/O segment is to make memory address space for memory only and move all the other stuff to I/O address space, such as CPU's CSR, PCI configuration space, video frame buffer, boot ROM, etc. With I/O segment, there will be no memory holes in memory address space. The same load/store instructions can be used to load from or store to either I/O or memory address space depending on the segment type. Also for a device driver, even if it runs on user level, as long as it has a capability to its I/O segment, it can access its I/O segment in user level without kernel intervention. Another benefit of I/O segment is that it is more fine-grained that paging. A segement can be as small as one byte while for paging the smallest size is 4K bytes.

Miscellaneous Benefits

Segmentation can make shared-library simpler 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 has its own code and data segment so we can know the offsets of all the global variables and functions at compile time. It means that the

offset part of global variables and functions can be statically linked at compile time. We only need to link the segment part of the address at program load time. So shared libraries can be partially linked at compile time on segmentation hardware.

The other benefit of segmentation is that it can make cross-domain call simpler to implement. Cross-domain call has many names in the literature. It is also known as migrating thread model, protected control transfer or passive object model. Currently a thread is confined in two protection domains, the process that created it and the kernel. It is not allowed for a thread running on one user level process to call into another user level process. If a client thread needs a service from a server, it will send a message to the server and wait for the reply. On the server side, a server thread that is waited on a message queue, receives the message, executes the request in the message and sends a reply message back to the client. This is complex and inefficient. By using cross-domain call, the client thread can call into the server. Traditionally to implement cross-domain call, the OS needs to switch current stack to a stack in the server domain when the thread is migrating to the server and switch back to its original stack in the client domain when the thread returns back to client. This is because OS wants to protect client stack from server and protect server stack from client. If we use the same stack for both the client and server, then we need to figure out a a way to protect the client part of the stack from the server and protect the server part of the stack from the client. This is pretty simple on a capability/segmentation hardware. By moving the stack capability from client domain to server domain when doing cross-domain call, the server part of the stack is protected from the client. We still need to figure out a way to protect the client part of the stack from the server. This can be accomplished by introducing a privileged register called stack domain boundary (sdb) register which is set by OS when the thread is doing a cross-domain call. This register restricts a thread from accessing the part of the stack that is below the address specified in sdb register so the client part of the stack can also be protected from the server.

Compatibility with Old Programs

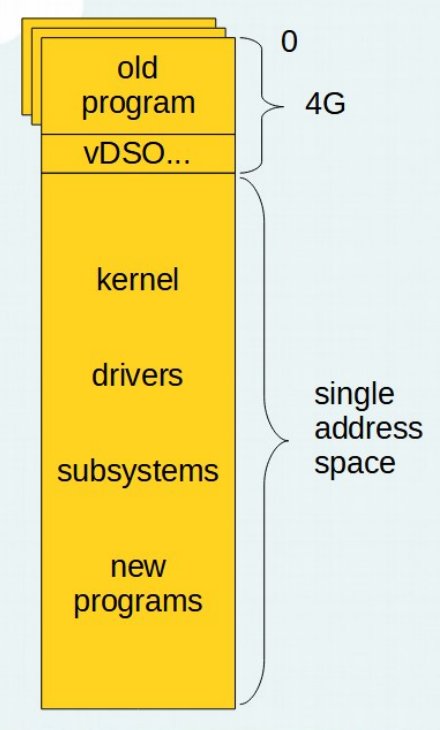

Figure 4: compatibility with old multi-address-space programs

In order for this new architecture to be adopted, it is needed to maintan backward compatibility with old multi-address-space programs. Figure 4 shows a way to maintain backward compatibility with

old programs. The platform runs 32-bit programs on 64-bit virtual address space. The first 4G address space is reserved for old multi-address-space programs. As you can see there are multiple virtual address spaces in this range. The rest of the address space is for the kernel and new single-address-space programs.

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. The hardware must support capability and segmentation. The compiler tools must support far pointer, far function call and a new way to implement shared-library. And the OS must support capability and 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.

[8] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), June 2014.