RISC-V Segmentation Extension Proposal

Wuyang Chung

wuyang.chung1@gmail.com

Introduction

Traditional operating systems use virtual memory to create protection domains for processes. Each process has its own private virtual address space (protection domain) so they can be protected from each other. The drawbacks of this multiple address space model are that context switch overhead is high and data sharing is complex and difficult. In order to avoid these drawbacks, single address space operating system (SASOS) has been proposed. The biggest problem with SASOS is protection. There are several ways to create multiple protection domains on a single address space and capability system is one of the them. There are also several ways to implement capability system and segmentation is one of them. Besides being used as a mechanism to build a SASOS, there are several other benefits of segmentation. It can be used to point to I/O address space so a user level device driver can access its hardware device directly without kernel intervention. It can translate a segment address directly to physical address so TLB pollution from segment with very low or no locality of reference can be avoided. It can be helpful to implement software-managed TLB. It can also be used as a way to do upcall from kernel to user space.

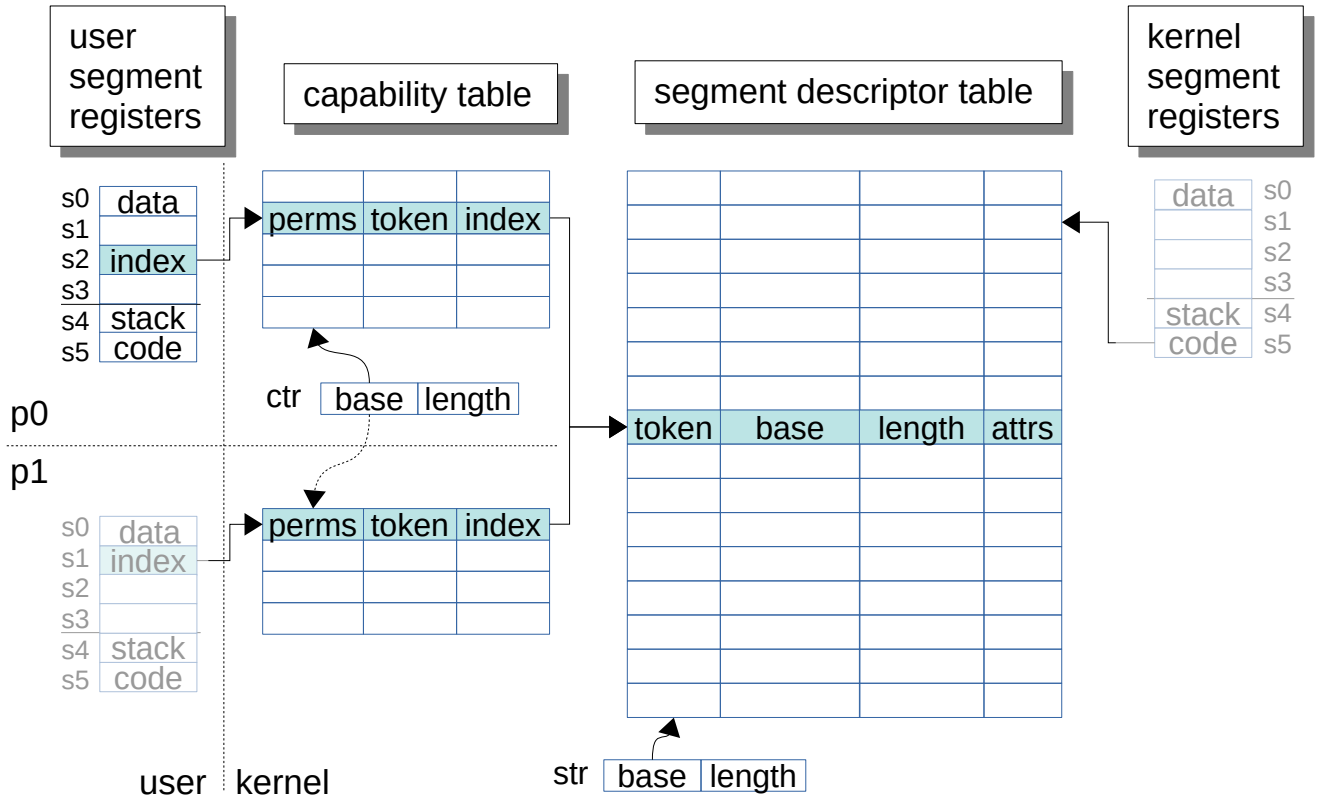

Figure 1: The relationship between segment registers, capability table and segment descriptor table.

In a capability system, each process has a capability table that stores the list of capabilities (to segments) a process can access. Since a process can only access the segments listed in its capability table, it forms a protection domain on a single address space for the process. Each capability contains an index to segment descriptor table. Each segment descriptor describes a segment in the system. It contains the base address, length and other attributes of the segment. The system have 5

segment registers, s0 to s4. Figure 1 shows the relationship between segment registers, capability table and segment descriptor table. When in user mode each segment register contains an index to capability table. s2 to s4 have designated usage. s4 is for code segment. s2 is for data segment. s3 is for stack segment. Operating system will initialize s2 to s4 with proper values when program starts and hardware will use the designated segment register when access code, data and stack segment. Thus it makes segmentation transparent for user programs that do not use it. The segment version of the load/store instructions can only specify segment s0 to s1 so one bit is needed to specify a segment. When in kernel mode, segment register contains an index to segment descriptor table since kernel program have privilege to access all segments in the system. So segment register in kernel mode points directly to segment descriptor table.

Load and Store with Segment Override Instructions

For each load/store instruction an optional segment override can be used to designate a segment register. The override can be s0 to s4. Segment register s0 is dedicated to be used as data segment so when an override is omitted it actually means s0. Below is the sample code for segment override.

```
LW x1, s1:imme[x2]
```

We need to squeeze two bits from the instruction for segment override. For embedded extension, the fourth bit of each register field can be used since embedded extension has only 16 general purpose registers. The instruction format is shown in figure 2.

load

imm[11:0]	o_1	rs1	func3	o_0	rd	opcode

store

imm[11:5]	o_0	rs2	o_1	rs1	func3	imm[0:4]	opcode

o_1	o_0	segment
0	0	data
0	1	s1
1	0	s2
1	1	s3

Figure 2: The load and store with segment override instructions

Load and Store Segment Register

There are two instructions for loading and storing segment register. The load segment register instruction (LSR) will load a value in memory to a s register. It has a similar format to LW instruction except that “rd” specifies a segment register. The store segment register (SSR) will store the value in a s register to memory. It has a similar format to SW instruction except that “rs2” specifies a segment register. The system should also have instructions to move data between s register and x register.

Function Call and Return to Segment

Since shared libraries have its own code segment, the JAL and JALR should also have segment version, called SJAL and SJALR. In addition to saving (pc+4), SJAL and SJALR also save the current code segment index. The operand also needs to provide the segment index of the target segment.

Segment Descriptor

Segment descriptor describes a segment in the system. It has the following fields.

Gen	Generation number. It is used for quickly revoking all the capabilities to this segment. There is a corresponding gen field in capability. If the gen field in capability is not equal to the gen field in segment descriptor, that capability is invalid.
Base	The base address of the segment.
Length	The length of the segment.
Attribute.v	The valid bit. If set, this segment descriptor is valid.
Attribute.cow	The copy-on-write bit. A write access to a COW segment will trigger a COW fault. OS can then move the segment to another place and clear the COW bit.
Attribute.io	I/O segment. If set, the segment address is translated to I/O address instead of virtual address. When a load/store with segment override on a segment with this bit set, the CPU will then load/store data from the I/O space instead of the memory space.
Attribute.identity	Identity-mapped. If set, the segment address is translated to physical address instead of virtual address.
Attribute.granularity	If set, the length field is in units of page instead of byte.
Software_attribute .owner	The owner of this segment. Only the owner can revoke a capability by setting a new gen number.
Software_attribute max permission	The max permission of this segment. Set by the owner of this segment.

Capability

Capability has the following fields.

Index	An index to the segment descriptor table. It means that a process has capability to access this segment.
Gen	Generation number. See the description above for segment descriptor.
Permission.read	Process can read this segment.
Permission.write	Process can write this segment.
Permission.execute	Process can execute this segment.
Attribute.v	The valid bit. If set, this capability is valid.

Segment Shadow Register

Each segment register has a corresponding shadow register. This shadow register is used to cache the capability. When an index is loaded into a segment register, its corresponding capability is cached in its shadow register.

Segment TLB

TLB is a cache for translation. In addition to the page TLB, the hardware will also have a segment TLB to cache segment descriptor.

Software-managed TLB

If the OS have kernel's code, data and stack segment identity-mapped and have kernel's code, data and stack segment descriptors wired down in segment TLB, then page fault and segment TLB miss will not happen when executing kernel code. So it is possible to have an implementation with software-managed TLB (both page and segment TLB).

Stack growth direction

The stack growth direction in segmentation architecture should be upward. If the stack grows downward and the stack pointer reaches zero, there is no way to expend the stack segment. On the other hand if the growth direction is upward, there is still a good chance to expend the stack when the stack pointer reaches the stack length. OS can simply move the stack segment to a new location with big enough free space. The segment move operation is chip. No page copying are needed, only the page table entries are copied. Stack growth downward is only beneficial for single-threaded program. For multi-threaded program and SASOS, there is no benefit that we can get from stack growth downward.

Segment Offset Space Sharing for Data and Stack Segment

For a function with data pointer parameter that points to some object, it might point to an object in either data segment or stack segment. In flat memory model, this is not a problem. But in segmentation model it is. In order to let the hardware know which segment the object resides, the segment offset space is split into two parts. The lower part is reserved for the data segment and the upper part is reserved for the stack segment. Say for example an offset space of 4 GiB and the upper 256 MiB is reserved for stack segment. Normally a load/store instruction will use data segment to access the data. However if a load/store instruction with a segment offset (effective address) greater than 4 GiB – 256 MiB, the hardware will use stack segment instead to access the data.

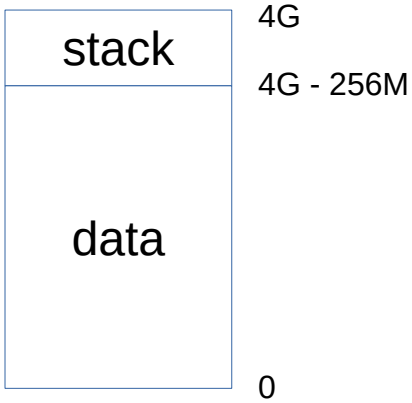

Figure 3: Segment offset is split into two parts. The lower part is for data segment and the upper part is for stack segment.

Kernel upcall

Kernel upcall can be used for the kernel to notify a user level device driver an interrupt event. The simplest way to do a kernel upcall is to migrate a kernel thread to user level device driver. Kernel thread migration can be implemented by migrating the corresponding thread stack to the user level device driver. In order to protect the kernel portion of the stack when the stack is migrated to user level, a new register is needed to protect the kernel portion of the stack. This register, called *stack protection base*, is used to prevent a user level program from accessing portion of the stack that is below the stack protection base. A new instruction is also needed to load/store value to this register.

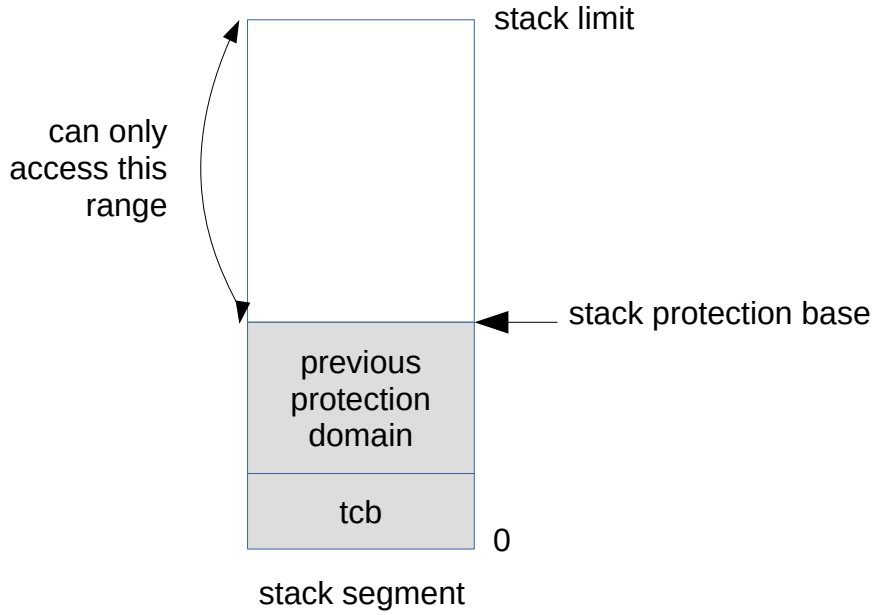

Figure 4: Stack protection base

Guard Null instead of Guard Page

Most operating systems use guard page to detect null pointer dereference. Using guard page to detect null pointer dereference not only wastes an entry in page table, it also wastes virtual address space. Hardware should support guard null for software to detect null pointer dereference. Guard null is segment offset 0 for every segment in the system. Every load/store or instruction fetch operations that reference segment offset 0 should trigger an exception so that operating system can terminate the offending process.