Project 7: Hypervisory (4%)

ENEE 447: Operating Systems — Spring 2016

Assigned: Monday, Mar 28; **Due:** Friday, Apr 8

Purpose

In this project you will figure out how to use three levels of privileges via three different levels of virtual memory. We still will not run separate application and kernel/ukernel binaries (we need SD card access for that), but we can run each piece of code in its own separate domain, translated through its own page table. This is how datacenters operate, for example: the "guest" operating system runs within a virtual memory session, unaware that it is in fact running in virtual memory. This allows the hypervisor -- the operating system for operating systems — to protect itself and keep the system secure. So, for example, a hypervisor could juggle several different instances of Windows, plus several instances of Linux, plus instances of MacOSX as well, all on the same machine, all at the same time, all completely unaware of each other.

Your Task

Your code will build on the last project's code: you will use both TTBR0 and TTBR1 to translate references, so that the ukernel's references are translated whenever the machine is in privileged mode (e.g., whenever the ukernel on core1 is handling interrupts), and the "user" code's references are also translated. The following page of ARM documentation shows what needs to happen:

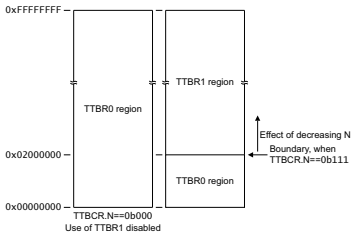


Figure B3-6 How TTBCR.N controls the boundary between the TTBRs, Short-descriptor format

In the selected TTBR. the following bits define the memory region attributes for the translation table walk:
- the RGN, S and C bits, in an implementation that does not include the Multiprocessing Extensions
- the RGN, S, and IRGN[1:0] bits, in an implementation that includes the Multiprocessing Extensions.

For more information, see *TTBCR, Translation Table Base Control Register, VMSA* on page B4-1724, *TTBR0, Translation Table Base Register 0, VMSA* on page B4-1729 and *TTBR1, Translation Table Base Register 1, VMSA* on page B4-1733.

Translation table walks, when using the Short-descriptor translation table format describes the translation.

B3.5.5 Translation table walks, when using the Short-descriptor translation table format

When using the Short-descriptor translation table format, and a memory access requires a translation table walk:
- a section-mapped access only requires a read of the first-level translation table
- a page-mapped access also requires a read of the second-level translation table.

Reading a first-level translation table describes how either TTBR1 or TTBR0 is used, with the accessed VA, to determine the address of the first-level descriptor.

Reading a first-level translation table shows the output address as A[39:0]:

- On an implementation that includes the Virtualization Extensions, for a Non-secure PL1&0 stage 1 translation, this is the IPA of the required descriptor. A Non-secure PL1&0 stage 2 translation of this address is performed to obtain the PA of the descriptor.
- Otherwise, this address is the PA of the required descriptor.

The full translation flow for Sections, Supersections, Small pages and Large pages on page B3-1332 then shows the complete translation flow for each valid memory access.

Reading a first-level translation table

When performing a fetch based on TTBR0:
- the address bits taken from TTBR0 vary between bits[31:14] and bits[31:7]
- the address bits taken from the VA, that is the input address for the translation, vary between bits[31:20] and bits[24:20].

The width of the TTBR0 and VA fields depend on the value of TTBCR.N, as Figure B3-7 on page B3-1332 shows.

There are two page tables: one for the bottom portion of the address space (where the ukernel will live), and one for the top portion of the address space. What this documentation says is that the smallest bottom portion ends at 0x02000000 … meaning a 32MB space.

The ARM architecture defines three levels of access privilege:

A3 Application Level Memory Model
A3.6 Access rights

A3.6 Access rights

ARMv7 defines additional memory region attributes, that define access permissions that can:

- Restrict data accesses, based on the privilege level of the access. See *Privilege level access controls for data accesses* on page A3-143.
- Restrict instruction fetches, based on the privilege level of the process or thread making the fetch. See *Privilege level access controls for instruction accesses* on page A3-143.
- On a system that implements the Security Extensions, restrict accesses so that only memory accesses with the Secure memory attribute are permitted. See *Memory region security status* on page A3-144.

These attributes are defined:

- In a VMSA implementation, in the MMU, see *Memory access control* on page B3-1356, *Memory region attributes* on page B3-1366, and *The effects of disabling MMUs on VMSA behavior* on page B3-1314.
- In a PMSA implementation, in the MPU, see *Memory access control* on page B5-1761 and *Memory region attributes* on page B5-1762.

A3.6.1 Processor privilege levels, execution privilege, and access privilege

As introduced in *About the Application level programmers' model* on page A2-38, within a security state, the ARMv7 architecture defines different levels of *execution privilege*:

- in Secure state, the privilege levels are PL1 and PL0
- in Non-secure state, the privilege levels are PL2, PL1, and PL0.

PL0 indicates unprivileged execution in the current security state.

The current processor mode determines the execution privilege level, and therefore the execution privilege level can be described as the *processor privilege* level.

Every memory access has an *access privilege*, that is either unprivileged or privileged.

The characteristics of the privilege levels are:

PL0 The privilege level of application software, that executes in User mode. Therefore, software executed in User mode is described as unprivileged software. This software cannot access some features of the architecture. In particular, it cannot change many of the configuration settings.

Software executing at PL0 makes only unprivileged memory accesses.

PL1 Software execution in all modes other than User mode and Hyp mode is at PL1. Normally, operating system software executes at PL1. Software executing at PL1 can access all features of the architecture, and can change the configuration settings for those features, except for some features added by the Virtualization Extensions that are only accessible at PL2.

Note

In many implementation models, system software is unaware of the PL2 level of privilege, and of whether the implementation includes the Virtualization Extensions.

The *PL1 modes* refers to all the modes other than User mode and Hyp mode.

Software executing at PL1 makes privileged memory accesses by default, but can also make unprivileged accesses.

PL2 Software executing in Hyp mode executes at PL2.

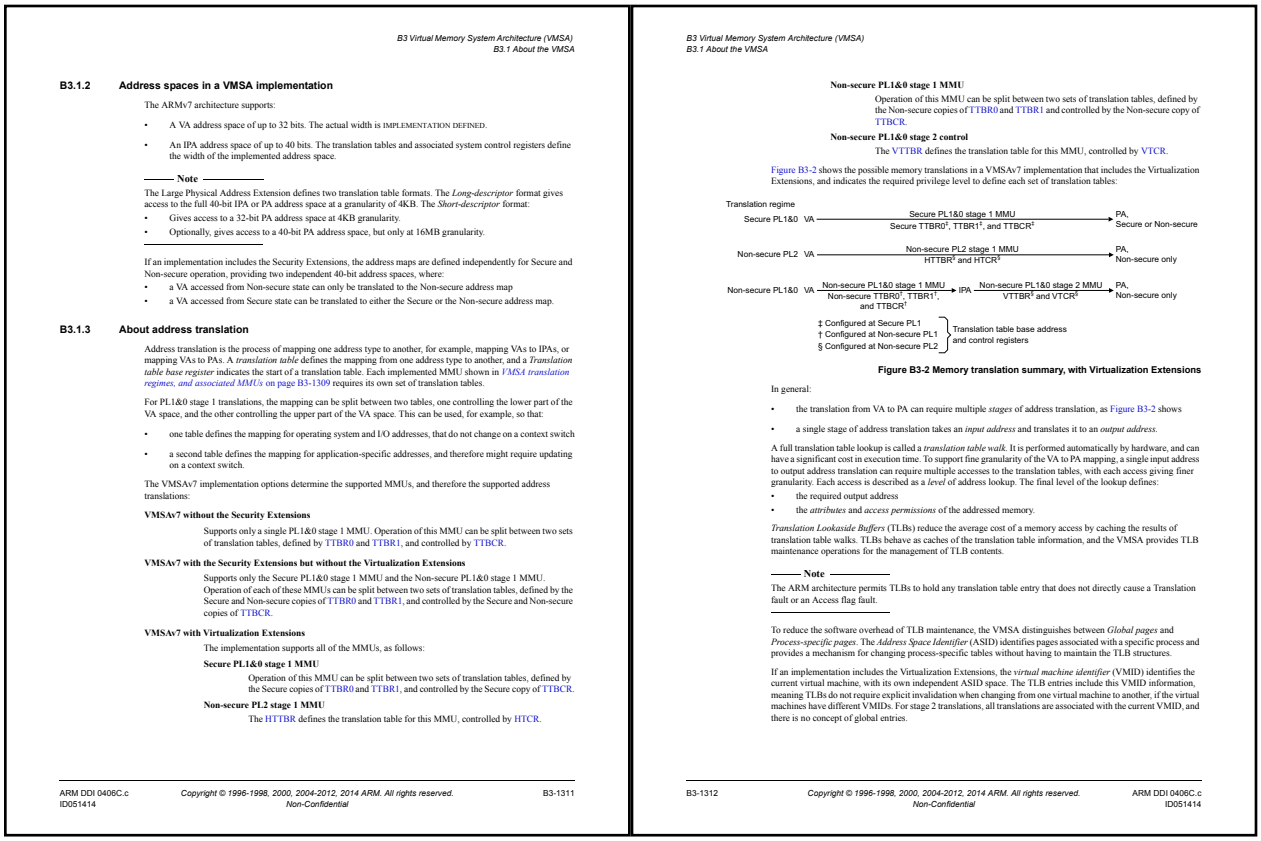
Software executing at PL2 can perform all of the operations accessible at PL1, and can access some additional functionality.

Hyp mode is normally used by a hypervisor, that controls, and can switch between, Guest OSs, that execute at PL1.

User-level code runs in PL0, which your boot code invokes as USR mode; guest operating system code (our "ukernel") runs in PL1, typically in SVC, SYS, IRQ, and FIQ modes; the main kernel runs in HYP mode which is PL2.

Note that ARM doesn't actually restrict the ability of code running in PL1 to take over the machine. So their differentiation between PL1 and PL2 is kind of a fake security thing. But whatever. Presumably

whatever real machine you use out in industry to build your systems will have multiple priority levels that actually work. We will fake it by translating the middle-level OS's references through the TLB. The following pages describe a bit about this:



B3 Virtual Memory System Architecture (VMSA)
B3.1 About the VMSA

B3.1.2 Address spaces in a VMSA implementation

The ARMv7 architecture supports:

- A VA address space of up to 32 bits. The actual width is IMPLEMENTATION DEFINED.
- An IPA address space of up to 40 bits. The translation tables and associated system control registers define the width of the implemented address space.

Note

The Large Physical Address Extension defines two translation table formats. The *Long-descriptor* format gives access to the full 40-bit IPA or PA address space at a granularity of 4KB. The *Short-descriptor* format:

- Gives access to a 32-bit PA address space at 4KB granularity.
- Optionally, gives access to a 40-bit PA address space, but only at 16MB granularity.

If an implementation includes the Security Extensions, the address maps are defined independently for Secure and Non-secure operation, providing two independent 40-bit address spaces, where:

- a VA accessed from Non-secure state can only be translated to the Non-secure address map
- a VA accessed from Secure state can be translated to either the Secure or the Non-secure address map.

B3.1.3 About address translation

Address translation is the process of mapping one address type to another, for example, mapping VAs to IPAs, or mapping VAs to PAs. A *translation table* defines the mapping from one address type to another, and a *Translation table base register* indicates the start of a translation table. Each implemented MMU shown in *VMSA translation regimes, and associated MMUs* on page B3-1309 requires its own set of translation tables.

For PL1&0 stage 1 translations, the mapping can be split between two tables, one controlling the lower part of the VA space, and the other controlling the upper part of the VA space. This can be used, for example, so that:

- one table defines the mapping for operating system and I/O addresses, that do not change on a context switch
- a second table defines the mapping for application-specific addresses, and therefore might require updating on a context switch.

The VMSAv7 implementation options determine the supported MMUs, and therefore the supported address translations:

VMSAv7 without the Security Extensions

Supports only a single PL1&0 stage 1 MMU. Operation of this MMU can be split between two sets of translation tables, defined by TTBR0 and TTBR1, and controlled by TTBCR.

VMSAv7 with the Security Extensions but without the Virtualization Extensions

Supports only the Secure PL1&0 stage 1 MMU and the Non-secure PL1&0 stage 1 MMU. Operation of each of these MMUs can be split between two sets of translation tables, defined by the Secure and Non-secure copies of TTBR0 and TTBR1, and controlled by the Secure and Non-secure copies of TTBCR.

VMSAv7 with Virtualization Extensions

The implementation supports all of the MMUs, as follows:

Secure PL1&0 stage 1 MMU

Operation of this MMU can be split between two sets of translation tables, defined by the Secure copies of TTBR0 and TTBR1, and controlled by the Secure copy of TTBCR.

Non-secure PL2 stage 1 MMU

The HTTBR defines the translation table for this MMU, controlled by HTCR.

ARM DDI 0406C.c ID051414 — *Copyright © 1996-1998, 2000, 2004-2012, 2014 ARM. All rights reserved. Non-Confidential* — B3-1311

B3 Virtual Memory System Architecture (VMSA)
B3.1 About the VMSA

Non-secure PL1&0 stage 1 MMU

Operation of this MMU can be split between two sets of translation tables, defined by the Non-secure copies of TTBR0 and TTBR1 and controlled by the Non-secure copy of TTBCR.

Non-secure PL1&0 stage 2 control

The VTTBR defines the translation table for this MMU, controlled by VTCR.

Figure B3-2 shows the possible memory translations in a VMSAv7 implementation that includes the Virtualization Extensions, and indicates the required privilege level to define each set of translation tables:

Figure B3-2 Memory translation summary, with Virtualization Extensions

In general:

- the translation from VA to PA can require multiple *stages* of address translation, as Figure B3-2 shows
- a single stage of address translation takes an *input address* and translates it to an *output address*.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single input address to output address translation can require multiple accesses to the translation tables, with each access giving finer granularity. Each access is described as a *level* of address lookup. The final level of the lookup defines:

- the required output address
- the *attributes* and *access permissions* of the addressed memory.

Translation Lookaside Buffers (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and the VMSA provides TLB maintenance operations for the management of TLB contents.

Note

The ARM architecture permits TLBs to hold any translation table entry that does not directly cause a Translation fault or an Access flag fault.

To reduce the software overhead of TLB maintenance, the VMSA distinguishes between *Global pages* and *Process-specific pages*. The *Address Space Identifier* (ASID) identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without having to maintain the TLB structures.

If an implementation includes the Virtualization Extensions, the *virtual machine identifier* (VMID) identifies the current virtual machine, with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another, if the virtual machines have different VMIDs. For stage 2 translations, all translations are associated with the current VMID, and there is no concept of global entries.

B3-1312 — *Copyright © 1996-1998, 2000, 2004-2012, 2014 ARM. All rights reserved. Non-Confidential* — ARM DDI 0406C.c ID051414

There is a new version of the "memmap" linker script, which looks like this:

```
MEMORY
{
        ram : ORIGIN = 0x0000, LENGTH = __SIZE__
}

SECTIONS
{
        .text : { *(.text*) } > ram
        .rodata : { *(.rodata*) } > ram
        .bss : { *(.bss*) } > ram
        .data : { *(.data*) } > ram

        .usercode  0x02000000 : { *(.usercode*      ) }

}
```

The "usercode" portion is new and tells the linker to put the code in that section way up into the region of (virtual) memory starting at 32MB, which happens to be covered by TTBR1.

So you will have two page tables: one to cover the ukernel, and another to cover user space. This will be enabled on cores other than core0, which will run the kernel in physical space.

Build It, Load It, Run It

Once you have it working, show us.