Project 8: Virtual Memory (4%)

ENEE 447: Operating Systems — Spring 2019

Assigned: Tuesday, Apr 23; **Due:** Sunday, May 5

Purpose

In this project you will figure out how to turn on the ARM's virtual memory system and run at least two different threads in two different virtual spaces that are the "same" addresses but map to completely different physical locations. Virtual memory underlies many of computing's most important facilities, including process protection, shared memory, multitasking, the kernel's privileged mode, the familiar virtual-machine programming model, and more. It is essential to most operating systems, especially general-purpose operating systems. Your implementation will be very simple but will have all of the essentials, including shared pages (two different virtual pages mapping to the same physical page), different mapping characteristics for different pages, etc. This is as real as it gets. With this, you will have built all of the primary functions one finds in a modern operating system.

You will read in application binaries from the SD card to start threads, and you will do this both as the startup thread (the shell) as well as in response to "RUN" commands executed in the shell, which will start up either or both of the "app1" and "app2" binaries. The difference between this project and the previous one is that, whereas, in the previous project each of the applications were hard-coded at build time to run in predefined memory locations (something that is not really practical in a general-purpose machine), in this project, each application has its code and data start at location 0x00100000, and its stack start at location 0x7FFFFFF0. Thus, to run two different user-level threads, you need to have separate page tables for each process and to figure out how to tell the ARM processor about two different ASIDs.

Working Example

You have been given a working binary file to experiment with. The following is its boot sequence.

```
[c0|00:01.957] ...
[c0|00:01.959] System is booting, kernel cpuid = 00000000
[c0|00:01.964] Kernel version [p8-solution, Mon Apr 22 20:45:29 EDT 2019]
[c0|00:01.971] Initializing SD Card ...
[c0|00:01.975] EMMC: reset card.
[c0|00:01.978] EMMC: setting clock speed to  00061A80
[c0|00:01.983] GO_IDLE_STATE 00000000
[c0|00:01.986] SEND_IF_COND 000001AA
[c0|00:01.989] APP_CMD 00000000
[c0|00:01.992] SD_SENDOPCOND 50FF8000
[c0|00:02.396] APP_CMD 00000000
[c0|00:02.399] SD_SENDOPCOND 50FF8000
[c0|00:02.403] ALL_SEND_CID 00000000
[c0|00:02.406] SEND_REL_ADDR 00000000
[c0|00:02.409] SEND_CSD AAAA0000
[c0|00:02.412] EMMC: setting clock speed to  017D7840
[c0|00:02.417] CARD_SELECT AAAA0000
[c0|00:02.420] APP_CMD AAAA0000
[c0|00:02.423] SEND_SCR 00000000
[c0|00:02.429] SET_BLOCKLEN 00000200
 sdTransferBlocks read blk 00000000 len 00000001 addr 0002BD80
[c0|00:02.437] READ_SINGLE 00000000
 sdTransferBlocks read blk 00002000 len 00000001 addr 0002BD80
[c0|00:02.450] READ_SINGLE 00002000
[c0|00:02.464]   ... SD Card working.
[c0|00:02.467] Starting virtual memory ...
[c0|00:02.471] TTBCR before = 00000000
[c0|00:02.475] Initialize DACR
[c0|00:02.478] Initialize SCTLR.AFE
[c0|00:02.481] SCTLR before AFE = 00C51838
[c0|00:02.485] Setting page table to 00030000
[c0|00:02.489] PTE[0] = 00026C0A
```

```
[c0|00:02.492] PTE[1] = 00126C0A
[c0|00:02.495] SCTLR before = 00C51838
[c0|00:02.498] SCTLR after = 00C5183D
[c0|00:02.502]   ... VM up and running
[c0|00:02.505] Calling create_thread
[c0|00:02.509] NULL thread 00000000
[c0|00:02.512]  tcb   = 00013DF4
[c0|00:02.515]  stack = 0001FFFC
[c0|00:02.518]  start = 00000040
[c0|00:02.521]  ttbr0 = 0003004A
[c0|00:02.524]  asid  = 00000000
[c0|00:02.527]  PTE[0] = 00026C0A
[c0|00:02.530]  PTE[1] = 00126C0A
[c0|00:02.533]  PTE[2] = 00226C0A
[c0|00:02.536] Calling create_thread
 sdTransferBlocks read blk 00003DCA len 00000001 addr 000077A8
[c0|00:02.545] READ_SINGLE 00003DCA
 LocateFATEntry: [shell.bin]
 sdTransferBlocks read blk 00003DCB len 00000001 addr 000077A8
[c0|00:02.559] READ_SINGLE 00003DCB
 sdTransferBlocks read blk 00027A0A len 00000001 addr 000077A8
[c0|00:02.570] READ_SINGLE 00027A0A
[c0|00:02.576] create success 00000001
 sdTransferBlocks read blk 00027A0B len 00000001 addr 000077A8
[c0|00:02.585] READ_SINGLE 00027A0B
 sdTransferBlocks read blk 00027A0C len 00000001 addr 000077A8
[c0|00:02.596] READ_SINGLE 00027A0C
 sdTransferBlocks read blk 00027A0D len 00000001 addr 000077A8
[c0|00:02.608] READ_SINGLE 00027A0D
 sdTransferBlocks read blk 00027A0E len 00000001 addr 000077A8
[c0|00:02.619] READ_SINGLE 00027A0E
 sdTransferBlocks read blk 00027A0F len 00000001 addr 000077A8
[c0|00:02.631] READ_SINGLE 00027A0F
[c0|00:02.637] create_thread - successful file read into 00200000
[c0|00:02.642] new thread from disk:
[c0|00:02.646] shell.bin 00200000
[c0|00:02.649] shell 00000001
[c0|00:02.651]  tcb   = 00013E5C
[c0|00:02.654]  stack = 7FFFFFF0
[c0|00:02.657]  start = 00100000
[c0|00:02.660]  ttbr0 = 0003404A
[c0|00:02.663]  asid  = 00000001
[c0|00:02.666]  PTE[0] = 00000000
[c0|00:02.669]  PTE[1] = 00226C0A
[c0|00:02.673]  PTE[2] = 00000000
[c0|00:02.676] ...
[c0|00:02.677] Init complete. Please hit any key to continue.
```

<hit enter>

```
Running the eggshell on core 0.
Available commands:
 RUN  = 004E5552
 PS   = 00005350
 TIME = 454D4954
 LED  = 0044454C
 LOG  = 00474F4C
 EXIT = 54495845
 DUMP = 504D5544

Please enter a command.
 c0> PS

CMD_PS
[c0|00:29.279] Active processes ...
[c0|00:29.282] Dumping TCB for thread 00000001
[c0|00:29.286] shell 00000001
[c0|00:29.289]  tcb @ 00013E5C
[c0|00:29.291]   r0   00000001
[c0|00:29.294]   r1   0000000A
[c0|00:29.297]   r2   00005350
[c0|00:29.300]   r3   00005350
[c0|00:29.303]   r4   7FFFFFB8
[c0|00:29.305]   r5   00000000
[c0|00:29.308]   r6   00000000
[c0|00:29.311]   r7   00000009
[c0|00:29.314]   r8   00000000
[c0|00:29.317]   r9   00100BA8
[c0|00:29.319]   r10  00000000
[c0|00:29.322]   r11  504D5544
```

```
[c0|00:29.325]   r12  7FFFFFB2
[c0|00:29.328]   sp   7FFFFF94
[c0|00:29.331]   lr   001008E8
[c0|00:29.333]   pc   00100270
[c0|00:29.336]   spsr 60000150
[c0|00:29.339]   ttbr 0003404A
[c0|00:29.342]   asid 00000001

Please enter a command.
 c0>
```

A few things to note from this. The following lines show that the bottom two bits of the kernel's PTEs are 0b10, which indicates that the pages are mapped at a "section" level, meaning 1MB pages (this simplifies the mapping scheme tremendously). They also indicate that the kernel's mappings are global (the bit at 0x00020000 is bit 17, set to 1, which is the "not-global" bit, meaning that the mappings are shared across all code).

```
[c0|00:02.489] PTE[0] = 00026C0A
[c0|00:02.492] PTE[1] = 00126C0A
```

The following line shows that the data is read into physical page 0x002 (address 0x00200000):

```
[c0|00:02.637] create_thread - successful file read into 00200000
```

The kernel uses *de facto* physical addresses, because the ARM's virtual memory mechanism does not have any easy way to allow the kernel to use physical addresses while user applications use virtual ones. When the MMU is turned on, all addresses will be translated, so we have the kernel do a 1:1 mapping.

You will also notice that, in the earlier section it is shown that the start address of the newly created thread, the shell, is 0x00100000, and its stack address is 0x7FFFFFF0. Later, when the PS command is run, the shell has been executing for a short while, and its PC and SP registers indicate that it does, indeed, execute starting at 0x00100000, and its stack does indeed start just below 0x80000000 and work its way downward.

One of the difficult aspects of moving data back and forth between the user code and the kernel code is the transfer of data through pointers. Character-based I/O is relatively simple (e.g., reading and writing to the console), but more complex data requires bulk transfer through pointers. The problem is that pointers do not work across address spaces, as we have discussed in class. The solution that most operating systems adopt is to use physical addresses, or *de facto* physical addresses as mentioned above, to "copy in" or "copy out" data between the kernel space and the user's space. This requires a manual translation between the user's virtual address (what is sent in through a system call), and its physical location. An example of this in action is the transfer of a character string from user space to kernel space in the LOG system call:

```
Please enter a command.
 c0> LOG "FOO BAR"

CMD_LOG [FOO BAR]
[c0|01:05.075] FOO BAR

Please enter a command.
 c0>
```

The string "FOO BAR" is read in a character at a time from the console, and then it is sent as a string to the kernel-log device. If the translation is not done correctly, this will either produce garbage, or it will cause a non-recoverable address fault, at which point the OS comes to a grinding halt.

Transferring strings is also used to start up applications. Note that the trap handler recognizes both file names and the simple integers "1" and "2" as input (as indicating "app1.bin" and "app2.bin" respectively). This will allow you to test your code even if the string-transfer is not working correctly.

```
Please enter a command.
 c0> RUN BLK "APP1.BIN"
```



```
CMD_RUN [BLK, 00100BB1]
[c0|01:27.345] SYSCALL_START_THREAD name = 004B4C42
[c0|01:27.350] SYSCALL_START_THREAD file = 00100BB1
[c0|01:27.354] BLK
[c0|01:27.356] Calling create_thread
 sdTransferBlocks read blk 00003DCA len 00000001 addr 000077A8
[c0|01:27.365] READ_SINGLE 00003DCA
 LocateFATEntry: [APP1.BIN]
 sdTransferBlocks read blk 00003DCB len 00000001 addr 000077A8
[c0|01:27.379] READ_SINGLE 00003DCB
 sdTransferBlocks read blk 00027A4A len 00000001 addr 000077A8
[c0|01:27.390] READ_SINGLE 00027A4A
[c0|01:27.396] create success 00000001
 sdTransferBlocks read blk 00027A4B len 00000001 addr 000077A8
[c0|01:27.405] READ_SINGLE 00027A4B
[c0|01:27.411] create_thread - successful file read into 00400000
[c0|01:27.416] new thread from disk:
[c0|01:27.420] APP1.BIN 00400000
[c0|01:27.423] BLK 00000002
[c0|01:27.425]  tcb   = 00013EC4
[c0|01:27.428]  stack = 7FFFFFF0
[c0|01:27.431]  start = 00100000
[c0|01:27.434]  ttbr0 = 0003804A
[c0|01:27.437]  asid  = 00000002
[c0|01:27.440]  PTE[0] = 00000000
[c0|01:27.443]  PTE[1] = 00426C0A
[c0|01:27.446]  PTE[2] = 00000000

Please enter a command.
 c0>
```

At this point, the LED starts blinking in a 1/2/3/4/1/2/3 … pattern, and the shell is responsive.

A few things to note from the output above. First, the string transfer, as described above. Second, the data is copied into physical page 0x004 (physical address 0x00400000), like the previous application binary went into page 0x002. Every application starts out with two 1MB pages: one to hold code & data, the other to hold the stack.

If the PS command were run at this point, we would see those values changing over time as the code executes and moves up and down the stack:

```
Please enter a command.
 c0> PS

CMD_PS
[c0|01:39.441] Active processes ...
[c0|01:39.444] Dumping TCB for thread 00000001
[c0|01:39.448] shell 00000001
[c0|01:39.451]  tcb @ 00013E5C
[c0|01:39.454]   r0   00000001
[c0|01:39.457]   r1   0000000A
[c0|01:39.460]   r2   00005350
[c0|01:39.462]   r3   00005350
[c0|01:39.465]   r4   7FFFFFB8
[c0|01:39.468]   r5   00000000
[c0|01:39.471]   r6   00000000
[c0|01:39.474]   r7   00000009
[c0|01:39.476]   r8   00000000
[c0|01:39.479]   r9   00100BA8
[c0|01:39.482]   r10  00000000
[c0|01:39.485]   r11  504D5544
[c0|01:39.488]   r12  7FFFFFB2
[c0|01:39.491]   sp   7FFFFF94
[c0|01:39.493]   lr   001008E8
[c0|01:39.496]   pc   00100270
[c0|01:39.499]   spsr 60000150
[c0|01:39.502]   ttbr 0003404A
[c0|01:39.505]   asid 00000001
[c0|01:39.507] Dumping TCB for thread 00000002
[c0|01:39.512] BLK 00000002
[c0|01:39.514]  tcb @ 00013EC4
[c0|01:39.517]   r0   00000003
[c0|01:39.520]   r1   7FFFFFD0
[c0|01:39.523]   r2   00000008
[c0|01:39.525]   r3   00000000
[c0|01:39.528]   r4   00000000
[c0|01:39.531]   r5   000AAE60
```



```
[c0|01:39.534]   r6   05EE2A63
[c0|01:39.537]   r7   00000004
[c0|01:39.539]   r8   00000000
[c0|01:39.542]   r9   00000000
[c0|01:39.545]   r10  00000000
[c0|01:39.548]   r11  00000000
[c0|01:39.551]   r12  00000000
[c0|01:39.553]   sp   7FFFFFCC
[c0|01:39.556]   lr   00100220
[c0|01:39.559]   pc   00100050
[c0|01:39.562]   spsr 80000150
[c0|01:39.565]   ttbr 0003804A
[c0|01:39.568]   asid 00000002

Please enter a command.
 c0>
```

As said before, this represents all of the main points of an operating system: we have multiple threads running in user space, each using the same virtual address (which simplifies the job of the compiler and linker), but each is operating out of a different physical space. This is what virtual memory is all about, and with this project, you have encountered the heart of the OS.

Virtual Memory and the ARM/Raspberry Pi

Address translation is the mechanism through which the operating system provides virtual address spaces to user-level applications. The operating system maintains a set of mappings that translate references within the per-process virtual spaces to the system's physical space. Addresses are usually mapped at a *page* granularity—typically several kilobytes. The mappings are organized in a *page table*, and for performance reasons most hardware systems provide a *translation lookaside buffer (TLB)* that caches those PTEs (page-table entries; i.e. mappings) that have been needed recently. When a process performs a load or store to a virtual address, the hardware translates this to a physical address using the mapping information in the TLB. If the mapping is not found in the TLB, it must be retrieved from the page table and loaded into the TLB before processing can continue. The ARM has a TLB, and its hardware can automatically walk the page tables and load the TLB with the required information, when it find it in the page table.

The ARM's page table looks like this:

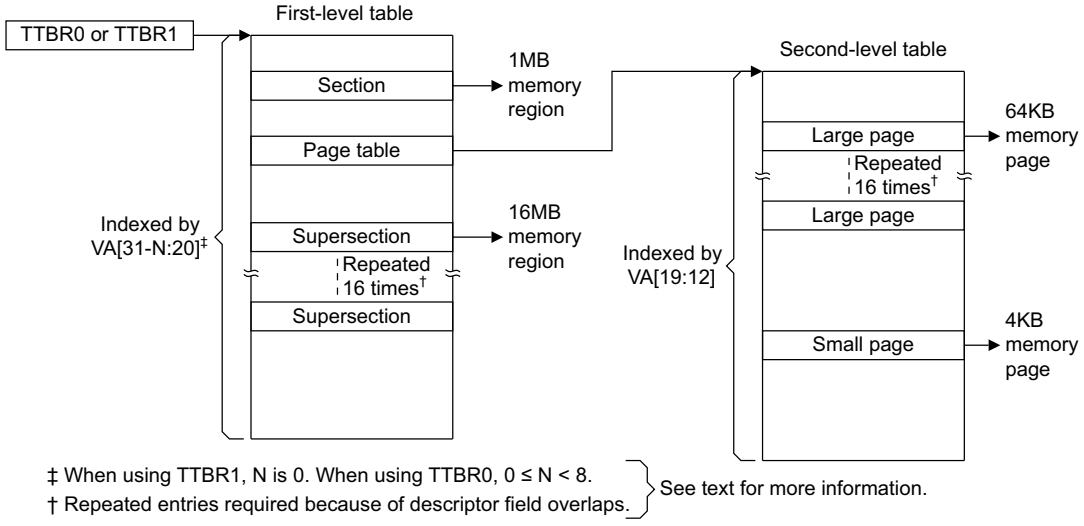


Figure B3-3 General view of address translation using Short-descriptor format translation tables

Note that there is one 4096-entry page in the first-level table and potentially thousands of pages making up the second-level table. However, if the PTE at the first level indicates that it maps a large area, like a 1MB "section" or a 16MB "supersection," then there need be no second-level table at all. That is what we

will do: have one simple 4096-entry table per process (and one for the kernel as well), with each entry mapping a 1MB "section" of memory.

The format of the ARM PTE (page-table entry) looks like this:

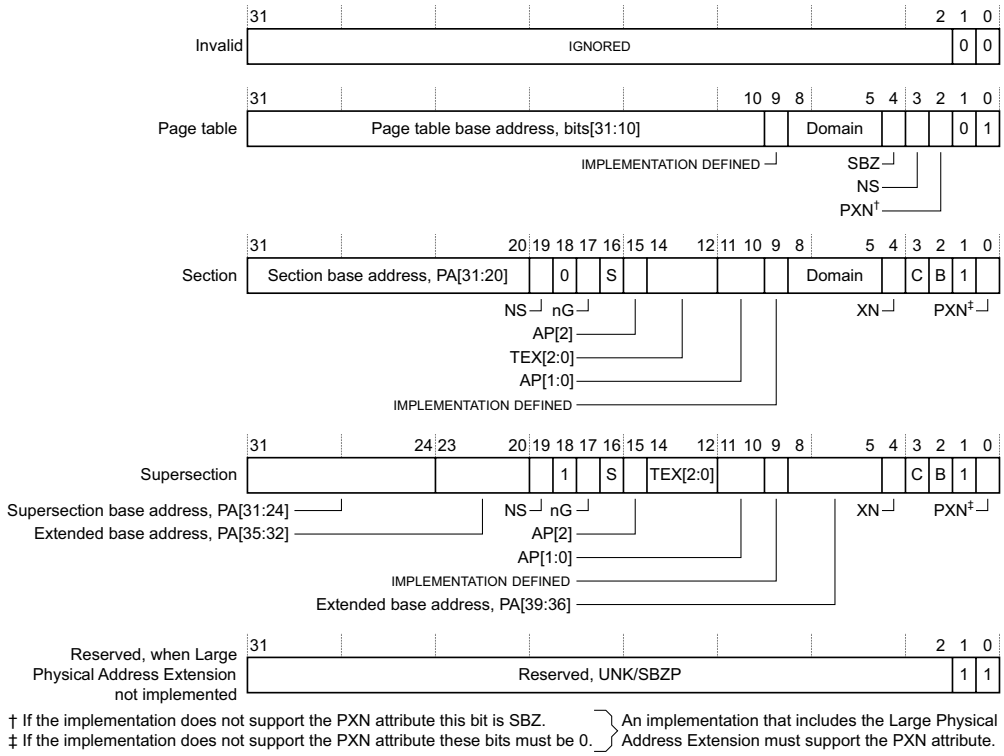


Figure B3-4 Short-descriptor first-level descriptor formats

Putting 0b10 in the bottom two bits indicates that the PTE is for a 1MB section. That is what we will do. Go to the ARM documentation for the details on the various fields in the entry: each topic shown you in this write-up will constitute anywhere from a few pages to a dozen pages in the ARM documentation, so it is a bit much to copy every page into this write-up.

Your First-Ever VM Implementation

We will implement the simplest of facilities: a single level page table (just an array, really) of page-table entries (PTEs) indexed by the virtual page number. Our page sizes will be the 1MB sections, so the page table need only hold 4K entries to map the entire 4GB space. Using large pages allows the table to be relatively small: 16KB per page table.

Note that, if a page size is 1MB, then the bottom 20 bits are page-offset bits, and the topmost 12 bits create the virtual page number. Thus an address looks like the following in hex:

```
0xVVVOOOOO
```

Where the "V" bits make up the virtual page number, and the "O" bits make up the page offset.

The kernel code on **core0** at the outset initializes the user page tables to 0s … in other words, all PTEs are invalid at startup. Thus, the **enable_vm**() routine needs only to set a handful of PTEs and then turn the correct switches to get the TLB operational. There are only a handful of distinct pages being used by your code at the moment the **enable_vm**() function is called:

- 0x3F0xxxxx — GPIO addresses

- 0x3F1xxxxx — GPIO addresses
- 0x3F2xxxxx — GPIO addresses
- 0x3F3xxxxx — GPIO addresses
- 0x400xxxxx — timer/clock device-register addresses
- 0x000xxxxx — where nearly all your code and data lies

You will also want to use the following for user code, data, and stack data:

- 0x001xxxxx–0x010xxxxx — for thread code, data, stacks (can be as big a region as you want)

You will want to create a mapping for each. The general code and data should be mapped as normal data, but the I/O addresses (0x3Fxxxxxx and 0x40xxxxxx) should be marked as non-cacheable so that they are handled correctly. This is controlled by the TEX field starting at bit 12 in the PTE.

ARM Documentation

You will find the *ARM Architecture Reference Manual* to be invaluable. I will point out some of the most important pages, but you need to explore this document yourself, because the information that you need is spread out all over the document. This is one of those (perhaps many) instances in which you curse ARM, because they really are a misnomer: ARM stands for Acorn RISC Machines, and RISC means Reduced Instruction-Set Computer … any computer architecture that requires tens of thousands of pages of documentation cannot possibly—in any way, shape, or form—be considered "reduced" …

B3 Virtual Memory System Architecture (VMSA)
B3.5 Short-descriptor translation table format

Short-descriptor translation table first-level descriptor formats

Each entry in the first-level table describes the mapping of the associated 1MB MVA range.

Figure B3-4 shows the possible first-level descriptor formats.

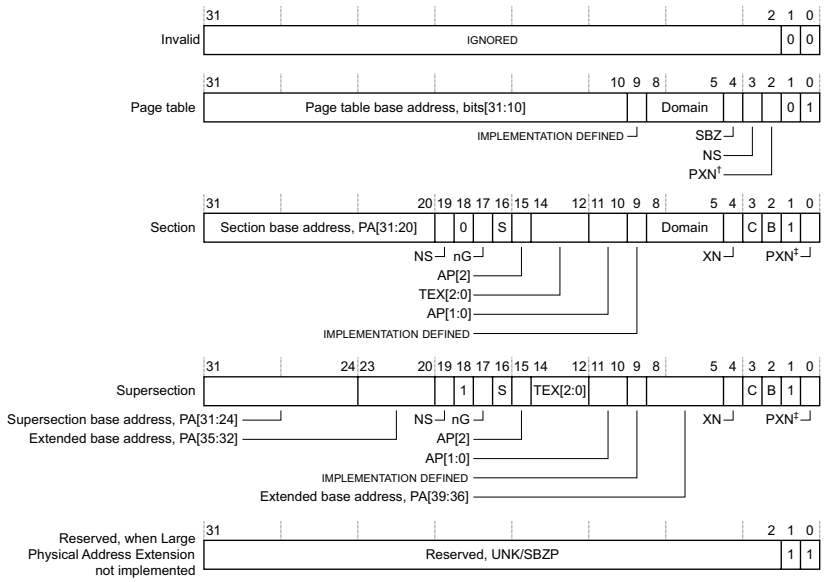


Figure B3-4 Short-descriptor first-level descriptor formats

Inclusion of the PXN attribute in the Short-descriptor translation table formats is:

- OPTIONAL in an implementation that does not include the Large Physical Address Extension
- required in an implementation includes the Large Physical Address Extension.

Descriptor bits[1:0] identify the descriptor type. On an implementation that supports the PXN attribute, for the Section and Supersection entries, bit[0] also defines the PXN value. The encoding of these bits is:

`0b00`**, Invalid**

The associated VA is unmapped, and any attempt to access it generates a Translation fault.

Software can use bits[31:2] of the descriptor for its own purposes, because the hardware ignores these bits.

`0b01`**, Page table**

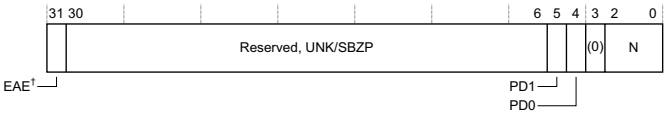
The descriptor gives the address of a second-level translation table, that specifies the mapping of the associated 1MByte VA range.

Above is a picture the format of the PTE … each of the bits has meaning, and the pages appearing after this one in the *Architectural Reference Manual* go into detail (and some are described *much* later in the document). Pay close attention to the bits involved in how the memory behaves (e.g., caching), because some of the settings are specifically for I/O addresses.

Note: in this project we are re-routing I/O addresses through the TLB. I suspect this is unusual, except for hypervisor/guest-operating-system configurations, because the OS on other architectures often runs in physical mode and is the only one allowed to touch the devices.

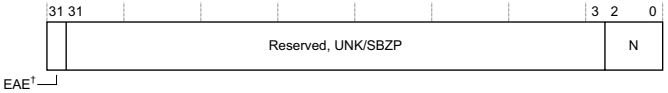
TTBCR format when using the Short-descriptor translation table format

In an implementation that includes the Security Extensions and is using the Short-descriptor translation table format, the TTBCR bit assignments are:



† Reserved, UNK/SBZP, if the implementation does not include the Large Physical Address Extension.

In an implementation that does not include the Security Extensions, and is using the Short-descriptor translation table format, the TTBCR bit assignments are:



† Reserved, UNK/SBZP, if the implementation does not include the Large Physical Address Extension.

EAE, bit[31], if implementation includes the Large Physical Address Extension

Extended Address Enable. The meanings of the possible values of this bit are:

0 Use the 32-bit translation system, with the Short-descriptor translation table format. In this case, the format of the TTBCR is as described in this section.

1 Use the 40-bit translation system, with the Long-descriptor translation table format. In this case, the format of the TTBCR is as described in *TTBCR format when using the Long-descriptor translation table format* on page B4-1726.

This bit resets to 0, in both the Secure and the Non-secure copies of the TTBCR.

Bit[31], if implementation does not include the Large Physical Address Extension

Reserved, UNK/SBZP.

Bits[30:6, 3] Reserved, UNK/SBZP.

PD1, bit[5], in an implementation that includes the Security Extensions

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1. The encoding of this bit is:

0 Perform translation table walks using TTBR1.

1 A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

PD0, bit[4], in an implementation that includes the Security Extensions

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss for an address that is translated using TTBR0. The meanings of the possible values of this bit are equivalent to those for the PD1 bit.

Bits[5:4], in an implementation that does not include the Security Extensions

Reserved, UNK/SBZP.

Shown above is the TTBCR, the register that determines how big the page size is, and whether there is one page-table or two, via the N bits. We will set it to use just one: the TTBR0 table, and we will disable the TTBR1 table, through the setting of the N bits in the TTBCR register.

B4.1.154 TTBR0, Translation Table Base Register 0, VMSA

The TTBR0 characteristics are:

Purpose TTBR0 holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

This register is part of the Virtual memory control registers functional group.

Usage constraints Only accessible from PL1 or higher.

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

Configurations The Multiprocessing Extensions change the TTBR0 32-bit register format.

The Large Physical Address Extension extends TTBR0 to a 64-bit register. In an implementation that includes the Large Physical Address Extension, TTBCR.EAE determines which TTBR0 format is used:

EAE==0 32-bit format is used. TTBR0[63:32] are ignored.

EAE==1 64-bit format is used.

If the implementation includes the Security Extensions, this register:

- is Banked
- has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

Attributes A 32-bit or 64-bit RW register with a reset value that depends on the register implementation. For more information see the register bit descriptions. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

The following subsections describe the TTBR0 formats:

- *32-bit TTBR0 format*
- *64-bit TTBR0 and TTBR1 format* on page B4-1731.

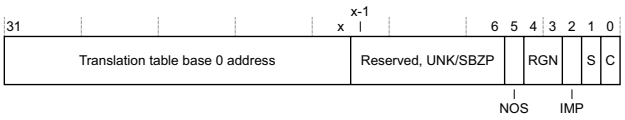
See *TTBCR, Translation Table Base Control Register, VMSA* on page B4-1724 for more information about using this register.

Note

See *TTBCR, Translation Table Base Control Register, VMSA* on page B4-1724 for a summary of the registers that define the translation tables for other address translations.

32-bit TTBR0 format

In an implementation that does not include the Multiprocessing Extensions, the 32-bit TTBR0 bit assignments are:



Shown above is the TTBR0 register. This contains the address of the page table for the currently executing process. When you context switch to another running *process* (which has a different address space, as opposed to switching to another *thread*, which doesn't), you need to give the hardware the pointer to the new process's address space.

Figure B3-3 gives a general view of address translation when using the Short-descriptor translation table format.

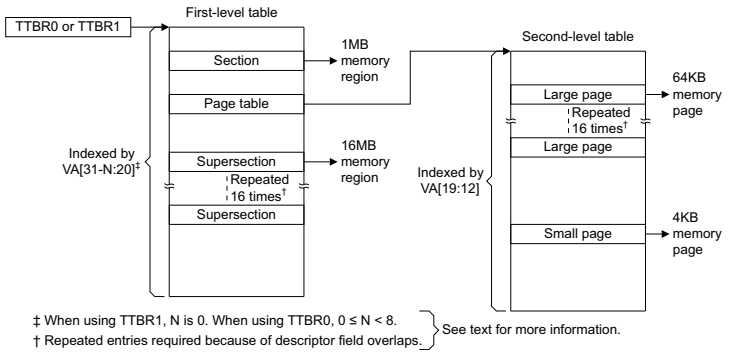


Figure B3-3 General view of address translation using Short-descriptor format translation tables

Additional requirements for Short-descriptor format translation tables on page B3-1328 describes why, when using the Short-descriptor format, Supersection and Large page entries must be repeated 16 times, as shown in Figure B3-3.

Short-descriptor translation table format descriptors, *Memory attributes in the Short-descriptor translation table format descriptors* on page B3-1328, and *Control of Secure or Non-secure memory access, Short-descriptor format* on page B3-1330 describe the format of the descriptors in the Short-descriptor format translation tables.

The following sections then describe the use of this translation table format:

- *Selecting between TTBR0 and TTBR1, Short-descriptor translation table format* on page B3-1330
- *Translation table walks, when using the Short-descriptor translation table format* on page B3-1331.

B3.5.1 Short-descriptor translation table format descriptors

The following sections describe the formats of the entries in the Short-descriptor translation tables:

- *Short-descriptor translation table first-level descriptor formats* on page B3-1326
- *Short-descriptor translation table second-level descriptor formats* on page B3-1327.

For more information about second-level translation tables see *Additional requirements for Short-descriptor format translation tables* on page B3-1328.

Note

Previous versions of the *ARM Architecture Reference Manual*, and some other documentation, describes the AP[2] bit in the translation table entries as the APX bit.

Information returned by a translation table lookup on page B3-1320 describes the classification of the non-address fields in the descriptors as address map control, access control, or attribute fields.

Shown above is the page-table organization, again (this is reproduced to give you the page number). The first level entries point to second-level entries, which point to the actual page data. When the first-level entries identify themselves as “sections” they instead point directly to page data.

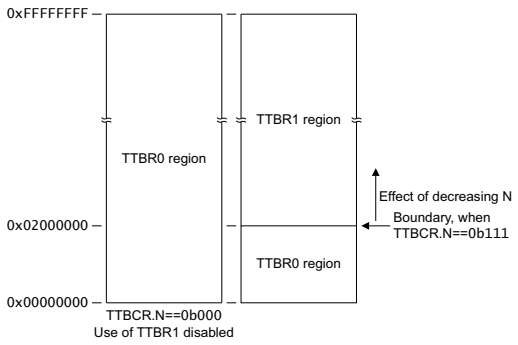


Figure B3-6 How TTBCR.N controls the boundary between the TTBRs, Short-descriptor format

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.

The discussion in the page above (and pages following it in the documentation) indicates how the system behaves wrt multiple multiple simultaneous mappings (e.g. split between two different guest operating systems). One is mapped through the TTBR0 page table, and the other is mapped through the TTBR1 page table, and the amount of memory assigned to each is variable. We will only use the TTBR0 page table and register.

B3.5.3 Control of Secure or Non-secure memory access, Short-descriptor format

Access to the Secure or Non-secure physical address map on page B3-1321 describes how the NS bit in the translation table entries:

- for accesses from Secure state, determines whether the access is to Secure or Non-secure memory
- is ignored by accesses from Non-secure state.

In the Short-descriptor translation table format, the NS bit is defined only in the first-level translation tables. This means that, in a first-level Page table descriptor, the NS bit defines the physical address space, Secure or Non-secure, for all of the Large pages and Small pages of memory described by that table.

The NS bit of a first-level Page table descriptor has no effect on the physical address space in which that translation table is held. As stated in *Secure and Non-secure address spaces* on page B3-1323, the physical address of that translation table is in:

- the Secure address space if the translation table walk is in Secure state
- the Non-secure address space if the translation table walk is in Non-secure state.

This means the granularity of the Secure and Non-secure memory spaces is 1MB. However, in these memory spaces, table entries can define physical memory regions with a granularity of 4KB.

B3.5.4 Selecting between TTBR0 and TTBR1, Short-descriptor translation table format

As described in *Determining the translation table base address* on page B3-1320, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N == 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- if N > 0 then:
  - if bits[31:32-N] of the input VA are all zero then use TTBR0
  - otherwise use TTBR1.

Table B3-1 shows how the value of N determines the lowest address translated using TTBR1, and the size of the first-level translation table addressed by TTBR0.

Table B3-1 Effect of TTBCR.N on address translation, Short-descriptor format

TTBCR.N	First address translated with TTBR1	TTBR0 table	
		Size	Index range
0b000	TTBR1 not used	16KB	VA[31:20]
0b001	0x80000000	8KB	VA[30:20]
0b010	0x40000000	4KB	VA[29:20]
0b011	0x20000000	2KB	VA[28:20]
0b100	0x10000000	1KB	VA[27:20]
0b101	0x08000000	512 bytes	VA[26:20]
0b110	0x04000000	256 bytes	VA[25:20]
0b111	0x02000000	128 bytes	VA[24:20]

Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

Figure B3-6 on page B3-1331 shows how the value of TTBCR.N controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

Shown above are the values that indicate how much space goes to the TTBR0 address space, and how much goes to the TTBR1 address space.

Banked system control registers

In an implementation that includes the Security Extensions, some system control registers are Banked. Banked system control registers have two copies, one Secure and one Non-secure. The SCR.NS bit selects the Secure or Non-secure copy of the register. Table B3-33 shows which CP15 registers are Banked in this way, and the permitted access to each register. No CP14 registers are Banked.

Table B3-33 Banked CP15 registers

CRn[a]	Banked register	Permitted accesses[b]
c0	CSSELR, Cache Size Selection Register	Read/write only at PL1 or higher
c1	SCTLR, System Control Register[c]	Read/write only at PL1 or higher
	ACTLR, Auxiliary Control Register[d]	Read/write only at PL1 or higher
c2	TTBR0, Translation Table Base 0	Read/write only at PL1 or higher
	TTBR1, Translation Table Base 1	Read/write only at PL1 or higher
	TTBCR, Translation Table Base Control	Read/write only at PL1 or higher
c3	DACR, Domain Access Control Register	Read/write only at PL1 or higher
c5	DFSR, Data Fault Status Register	Read/write only at PL1 or higher
	IFSR, Instruction Fault Status Register	Read/write only at PL1 or higher
	ADFSR, Auxiliary Data Fault Status Register[d]	Read/write only at PL1 or higher
	AIFSR, Auxiliary Instruction Fault Status Register[d]	Read/write only at PL1 or higher
c6	DFAR, Data Fault Address Register	Read/write only at PL1 or higher
	IFAR, Instruction Fault Address Register	Read/write only at PL1 or higher
c7	PAR, Physical Address Register	Read/write only at PL1 or higher
c10	PRRR, Primary Region Remap Register	Read/write only at PL1 or higher
	NMRR, Normal Memory Remap Register	Read/write only at PL1 or higher
c12	VBAR, Vector Base Address Register	Read/write only at PL1 or higher
c13	FCSEIDR, FCSE PID Register[e]	Read/write only at PL1 or higher
	CONTEXTIDR, Context ID Register	Read/write only at PL1 or higher
	TPIDRURW, User Read/Write Thread ID	Read/write at all privilege levels, including PL0
	TPIDRURO, User Read-only Thread ID	Read-only at PL0 Read/write at PL1 or higher
	TPIDRPRW, PL1 only Thread ID	Read/write only at PL1 or higher

a. For accesses to 32-bit registers. More correctly, this is the primary coprocessor register.
b. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
c. Some bits are common to the Secure and the Non-secure copies of the register, see *SCTLR, System Control Register, VMSA* on page B4-1707.
d. See *ADFSR and AIFSR, Auxiliary Data and Instruction Fault Status Registers, VMSA* on page B4-1523. Register is IMPLEMENTATION DEFINED.
e. Banked only in an implementation that includes the FCSE. The FCSE PID Register is RAZ/WI if the FCSE is not implemented.

Shown above is a (partial) list of the various control registers that you have to deal with. Nice to have it in one place. The ***mmu.s*** file has a bunch of functions that read and write many of these registers.

B4.1.130 SCTLR, System Control Register, VMSA

The SCTLR characteristics are:

Purpose The SCTLR provides the top level control of the system, including its memory system.

This register is part of the Virtual memory control registers functional group.

Usage constraints Only accessible from PL1 or higher.

Control bits in the SCTLR that are not applicable to a VMSA implementation read as the value that most closely reflects that implementation, and ignore writes.

In ARMv7, some bits in the register are read-only. These bits relate to non-configurable features of an ARMv7 implementation, and are provided for compatibility with previous versions of the architecture.

Configurations In an implementation that includes the Security Extensions, the SCTLR:

- is Banked, with some bits common to the Secure and Non-secure copies of the register
- has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

For more information, see *Classification of system control registers* on page B3-1451.

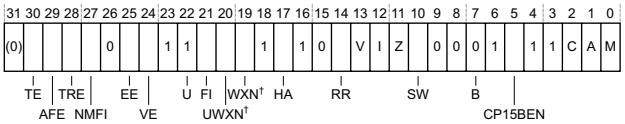
Attributes A 32-bit RW register with an IMPLEMENTATION DEFINED reset value, see *Reset value of the SCTLR* on page B4-1713. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Note

In an implementation that includes the Virtualization Extensions, some reset requirements apply to the Non-secure copy of SCTLR.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

In a VMSAv7 implementation, the SCTLR bit assignments are:



† Reserved before the introduction of the Virtualization Extensions, see text for more information.

Bit[31] Reserved, UNK/SBZP.

TE, bit[30] Thumb Exception enable. This bit controls whether exceptions are taken in ARM or Thumb state. The possible values of this bit are:

0 Exceptions, including reset, taken in ARM state.

1 Exceptions, including reset, taken in Thumb state.

In an implementation that includes the Security Extensions, this bit is Banked between the Secure and Non-secure copies of the register.

An implementation can include a configuration input signal that determines the reset value of the TE bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0 in an ARMv7-A implementation.

For more information about the use of this bit, see *Instruction set state on exception entry* on page B1-1182.

Shown above is the System Control Register, which has the all-important M bit in it, which turns on/off the MMU (i.e., virtual memory).

B4.1.36 CONTEXTIDR, Context ID Register, VMSA

The CONTEXTIDR characteristics are:

Purpose CONTEXTIDR identifies the current *Process Identifier* (PROCID) and, when using the Short-descriptor translation table format, the *Address Space Identifier* (ASID).

This register is part of the Virtual memory control registers functional group.

Usage constraints Only accessible from PL1 or higher.

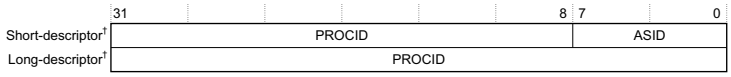
Configurations The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

In an implementation that includes the Security Extensions, this register is Banked.

Attributes A 32-bit RW register with an UNKNOWN reset value. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

In a VMSA implementation, the CONTEXTIDR bit assignments are:



† Current translation table format

PROCID, bits[31:0], when using the Long-descriptor translation table format

PROCID, bits[31:8], when using the Short-descriptor translation table format

Process Identifier. This field must be programmed with a unique value that identifies the current process. See also *Using the CONTEXTIDR*.

ASID, bits[7:0], when using the Short-descriptor translation table format

Address Space Identifier. This field is programmed with the value of the current ASID.

Note

When using the Long-descriptor translation table format, either TTBR0 or TTBR1 holds the current ASID.

Using the CONTEXTIDR

The value of the whole of this register is called the *Context ID* and is used by:

- the debug logic, for Linked and Unlinked Context ID matching, see *Breakpoint debug events* on page C3-2041 and *Watchpoint debug events* on page C3-2059
- the trace logic, to identify the current process.

The ASID field value is an identifier for a particular process. In the translation tables it identifies entries associated with a process, and distinguishes them from global entries. This means many cache and TLB maintenance operations take an ASID argument.

For information about the synchronization of changes to the CONTEXTIDR see *Synchronization of changes to system control registers* on page B3-1461. There are particular synchronization requirements when changing the ASID and Translation Table Base Registers, see *Synchronization of changes of ASID and TTBR* on page B3-1386.

When threads from multiple address spaces run, the hardware needs to be able to distinguish them. Shown above is the register that does so. It tells the hardware "any PTE you load while running, attach this ASID to it when you put it into the TLB." That way, when that process is swapped out and then is swapped back in later, it can still use its old mappings if they are still in the TLB.

Note that handling the various registers is extremely difficult to do, and so the changeover at process-switch time has been done for you. Otherwise, you would easily spend weeks trying to get it right. Remember, the important thing you are to learn in this project is the concept of mapping … learning the low-level details of how to interact with the ARM hardware is not the main goal. Thus, the interrupt vectors have been provided … the IRQ vector is shown below (the SVC vector is very similar):

```
irq_handler:

    // hard-coded return to kernel VM
    mov     sp,#0
    mcr     p15, 0, sp, c13, c0, 1      @ Write Rt to CONTEXTIDR
    isb
    mov     sp,#0x30000
    orr     sp,sp,#0x4a
    mcr     p15, 0, sp, c2, c0, 0       @ Write r0 to 32-bit TTBR0
    isb

    ldr     sp, tcb_address_runningthread      @ load the now-destroyed r13 w TCB pointer
    stmia   sp,{r0-lr}^     @ Save all user registers r0-lr
                    @ (the ^ means user registers)

    str     lr,[sp,#60]     @ store saved PC to TCB
    str     lr, save_lr_irq           @ save the SVC lr
    mrs     lr, SPSR        @ load SPSR (assume ip not a swi arg)
    str     lr,[sp,#64]     @ store to TCB
    ldr     lr, save_lr_irq           @ save the SVC lr

    @ Call the C version of the handler
    mov     sp, #SVCSTACK0
    bl      clear_timer_interrupt
    bl      periodic_timer
    bl      set_timer

    ldr     sp, tcb_address_runningthread      @ load the now-destroyed r13 w TCB pointer
    ldr     r0,[sp,#64]               @ retrieve saved CPSR
    msr     SPSR_cxsf, r0             @ move it into place

    ldr     lr,[sp,#60]               @ restore address to return to

    @ Restore saved values.  The ^ means to restore the userspace registers
    ldmia   sp, {r0-lr}^

    // no longer need the local-mode sp - use it to switch to user VM
    ldr     sp,[sp,#72]               @ retrieve saved ASID
    mcr     p15, 0, sp, c13, c0, 1      @ Write Rt to CONTEXTIDR
    isb
    ldr     sp, tcb_address_runningthread      @ load the now-destroyed r13 w TCB pointer
    ldr     sp,[sp,#68]               @ retrieve saved TTBR
    mcr     p15, 0, sp, c2, c0, 0       @ Write r0 to 32-bit TTBR0
    isb

    subs    pc, lr, #4                @ return from exception
```

There is a lot going on here. The following puts the machine back to kernel mode, using the thread ID 0, and a hard-coded pointer to the thread-0 page table:

```
    // hard-coded return to kernel VM
    mov     sp,#0
    mcr     p15, 0, sp, c13, c0, 1      @ Write Rt to CONTEXTIDR
    isb
    mov     sp,#0x30000
    orr     sp,sp,#0x4a
    mcr     p15, 0, sp, c2, c0, 0       @ Write r0 to 32-bit TTBR0
    isb
```

The first thing it does is move “0” into the ASID register, and then it moves 0x0003004A into the TTBR0 register. The 0x00030000 value is a pointer to the page table. The 0x4A is cacheable/sharable information, and I am not sure that it is necessary.

The next thing that happens is storing of the currently-running thread’s information to its TCB:

```
    ldr     sp, tcb_address_runningthread      @ load the now-destroyed r13 w TCB pointer
    stmia   sp,{r0-lr}^     @ Save all user registers r0-lr
                    @ (the ^ means user registers)

    str     lr,[sp,#60]     @ store saved PC to TCB
    str     lr, save_lr_irq           @ save the SVC lr
    mrs     lr, SPSR        @ load SPSR (assume ip not a swi arg)
    str     lr,[sp,#64]     @ store to TCB
    ldr     lr, save_lr_irq           @ save the SVC lr
```


This looks just like the previous project. A this point the code is free to do the handling. In this case (it is the IRQ vector, which handles the periodic timer interrupt), the call is to the ***periodic_timer()*** function, and also clearing and re-setting the timer:

```
@ Call the C version of the handler
mov     sp, #SVCSTACK0
bl      clear_timer_interrupt
bl      periodic_timer
bl      set_timer
```

Next, the register-file state is restored from the TCB. The ***periodic_timer()*** function may schedule a new task, so the new TCB may not be the same as the old TCB.

```
ldr     sp, tcb_address_runningthread      @ load the now-destroyed r13 w TCB pointer
ldr     r0,[sp,#64]                 @ retrieve saved CPSR
msr     SPSR_cxsf, r0               @ move it into place

ldr     lr,[sp,#60]                 @ restore address to return to

@ Restore saved values.  The ^ means to restore the userspace registers
ldmia   sp, {r0-lr}^
```

At this point, we cannot touch any of the registers that might affect the thread about to be run. That includes **r0–r14**, and the **IRQ–lr** register (not the same as the **USR–lr** register). The **IRQ–lr** register is used to get back to the user program, and the **USR–lr** register is the user thread's most recent function return point. The only register no longer needed is the **IRQ–sp** register. Therefore, we use this to set up the next thread's virtual memory configuration:

```
// no longer need the local-mode sp - use it to switch to user VM
ldr     sp,[sp,#72]                 @ retrieve saved ASID
mcr     p15, 0, sp, c13, c0, 1      @ Write Rt to CONTEXTIDR
isb
ldr     sp, tcb_address_runningthread      @ load the now-destroyed r13 w TCB pointer
ldr     sp,[sp,#68]                 @ retrieve saved TTBR
mcr     p15, 0, sp, c2, c0, 0       @ Write r0 to 32-bit TTBR0
isb
```

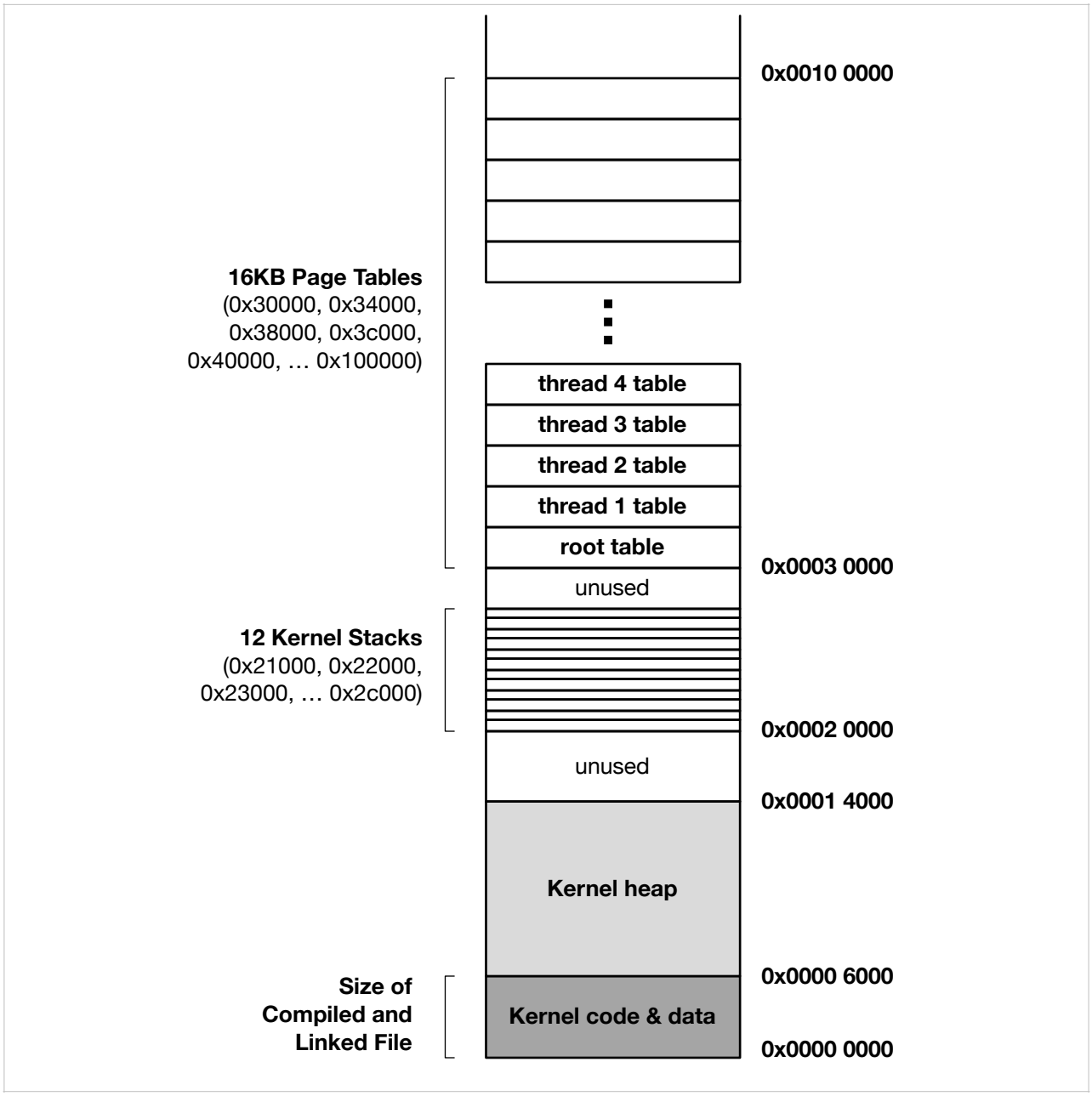
We grab the ASID register from the TCB and write it to the ASID control register. Then we sync (the "**isb**" instruction). Next, we grab the TTBR value (pointer to the user page table) from the TCB and write it to TTBR0, followed by another sync. Lastly, we return to user code via a *de facto* return-from-interrupt instruction, used widely in the ARM-32 architecture:

```
subs    pc, lr, #4                  @ return from exception
```

As mentioned above, the SVC handler is similar.

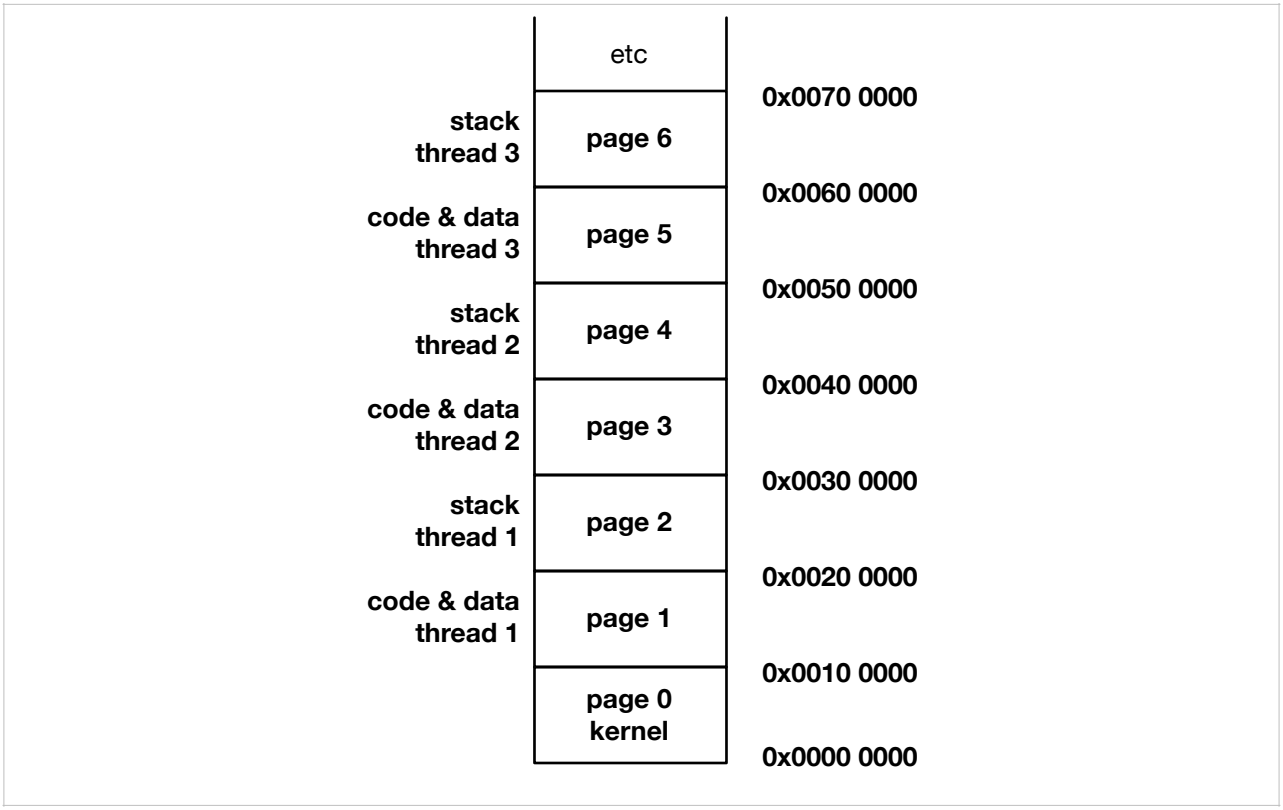
Where Things Go

As discussed in the previous project, we know how big the kernel is, and so we know where we can put things in physical memory. The following diagram indicates the major components for this project:



The main difference between this and the previous project is that the thread stacks have been moved elsewhere, since they are virtual pages and not physically assigned. Instead, starting at location 0x00030000 we have the page tables, indexed by the thread ID number. You only beed a handful of these, because you only need to run two threads (and we only have three application binaries at any rate …).

The physical page ends at the 1MB boundary: address 0x00100000. At that point we start using space for the application binaries. This is shown in the following figure:



Everything in the previous figure is in the "page 0 kernel" box at the bottom of the stack above. The system's physical memory is divided into 1MB chunks, called "sections" in the ARM documentation, and there are 4096 of them in the system, so we have 4096-entry page tables to map the space.

The easiest allocation scheme is to start at location 0x00100000 and increment it every time you create a new task: once for the code and data, and once for the stack.

The code and data starting at 0x00100000 is hard-coded into the linker files (**memmap** files) in the application directories.

Other Changes

Some other changes you might notice. To simplify things, the ***kernel.c*** module launches into the idle task first, and then it simply puts the shell on the ***runq***. The shell is started when the timer interrupt causes the IRQ interrupt handler to run, at which point it finds the shell on the ***runq*** and makes the thread active. Thus, there are only two places where user-thread contexts can be swapped (the two interrupt handlers), and there is only one place where a newly-created user thread can start running (the IRQ interrupt handler). The idle thread is actually a kernel thread.

Build It, Load It, Run It

Once you have it working, show us.