COMS20001 lab. worksheet #5

- All of the below assumes you are working on the workstations in MVB-2.11. If/when need be, you *could* alternatively make use of a dedicated (albeit experimental) Codio box: the unit web-page includes a per-worksheet link to enable direct access from Blackboard.
- Both the hardware and software in MVB-2.11 is managed by the IT Services Zone E team. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: either talk to them directly in room MVB-3.41, or submit a service request online via

  `http://servicedesk.bristol.ac.uk`

- We intend this worksheet to be attempted, at least partially, in the associated lab. session. Your attendance is important, since this session represents the primary source of formative feedback and help for COMS20001. Perhaps more so than in units from earlier years, *you* need to actively ask questions of and seek help from either the lectures and/or lab. demonstrators present.
- The questions are roughly classified as either L (for coursework related questions that should be completed in the lab. session), or A (for additional questions that are entirely optional). Keep in mind that we only *expect* you to complete the first class of questions: the additional content has been provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring it (since it is not directly assessed).

Before you start work, download (and, if need be, unarchive[a]) the file

`http://tinyurl.com/y72dnlcy/csdsp/os/sheet/lab-5_q.tar.gz`

somewhere secure[b] in your file system: it is intended to act as a starting point for your own work, and will be referred to in what follows.

[a] Use the `gz` and `tar` commands within a BASH shell (e.g., in a terminal window), *or* the archive manager GUI (available either via the menu `Applications`→`Accessories`→`Archive Manager` or by directly executing `file-roller`) if you prefer.

[b] For example, the `Private` sub-directory in your home directory.

Q1[A]. By this point, we assume you will prefer to focus on the (assessed) coursework assignment rather than the (non-assessed) lab. worksheets, so this question is entirely optional. Put another way, it focuses on a topic related to the final stage of the assignment and is thus more challenging than previous worksheets: unless you aim to submit a solution for the associated option, it makes sense to ignore this question and spend your time working on the rest of the assignment instead.

In the following Sections, the goal is to offer, via another example kernel image, an introduction to the Memory Management Unit (MMU) and hence a route toward implementation (and/or more thorough understanding) of virtual memory.

Background

The PB-A8 platform, or, more specifically, the Cortex-A8 processor, houses an MMU based on the ARM Virtual Memory System Architecture (VMSA): it represents a flexible, and hence complex aspect of the architecture, documented in [1, Chapters B3+B4]. This documentation is not for the faint hearted, and at ∼ 400 pages it would be tempting to disregard it as totally inaccessible.

However, it *is* reasonable to make use of the MMU *if* we consider a carefully limited focus. In fact, we already did this to some extent in the lecture(s), where we used VMSA as an example of paged memory more generally. Within that example, the MMU was described as performing two fundamental tasks: wlog. considering a load from address x, it a) checks whether use of x should be allowed wrt. whatever access permissions have been set in relation to it, then, iff. the check passes, b) translates the virtual address x into a physical address used to identify an element in physical memory. It performs both tasks in a largely autonomous manner, with little or no intervention from the processor. This is important: if it were *not* the case, it would imply a significant performance overhead. However, it is vital the processor (or, more precisely the kernel executing on the processor) configures the MMU st. it can operate as intended. This, in a nutshell, is our focus in the following: we will aim to demonstrate, in a practical sense, 1) how the VMSA-based MMU is configured via the ARM co-processor interface, and 2) that the resulting behaviour matches our previously theoretical understanding of paged memory.

A recap on the theory of paged memory In the lecture(s), we described translation of virtual address x into a physical address as

$$x \mapsto b[\mathsf{MSB}_t(x)] \cdot l + \mathsf{LSB}_{w-t}(x)$$

where $l = \rho$ is the page size: the look-up

$$b[\mathsf{MSB}_t(x)]$$

captures the idea that the MMU will access a page table based on the virtual address. That is, the physical address is constructed using a) the base address of a physical page frame, looked-up using the page number (i.e., the t MSBs of x), plus b) the page offset (i.e., the $w - t$ LSBs of x). More concretely, if $w = 32$ and $t = 12$, for example, we could say that because

$$x = \underbrace{x_{31,\ldots,20}}_{\text{page number}} \parallel \underbrace{x_{19,\ldots,0}}_{\text{page offset}}$$

we get

$$x \mapsto b[x_{31,\ldots,20}] \parallel x_{19,\ldots,0}$$

where, again, $b[x_{31,\ldots,20}]$ represents look-up[1] of the base address.

From theory toward practice: configuration of the VMSA-based MMU The MMU itself is exposed to the processor via the ARM co-processor interface, which is a general mechanism for adding functionality to the processor: the idea is to have a general-purpose processor, but allow additional tightly-coupled (e.g., on-chip, with a shared clock and dedicated communication busses) *co*-processors to support a range of special-purpose, domain-specific operations. Although they are more capable, one could consider interaction with a co-processor via two cases:

a the processor can transfer data to and from the co-processor using the `mcr` and `mrc` instructions,

b the processor can off-load computation to the co-processor using the `cpd` instruction.

The transfer of data could either represent input to or output from subsequent computation, *or* configuration of the co-processor (implying the register accessed is some form of control register). The latter is precisely how the MMU interface works: the processor, for example, transfers data into a specific co-processor register in order to tell the MMU where the page table base address is. The instruction syntax is described [1, Section A4.10] as

```
co-processor  →     processor  ⇒  mrc <id>, <opc1>, <Rd>, <CRn>, <CRm>, <opc2>
   processor  →  co-processor  ⇒  mcr <id>, <opc1>, <Rs>, <CRn>, <CRm>, <opc2>
```

so, for example, the instruction

```
mrc p15, 0, r0, c1, c0, 0
```

moves the content of register $\mathsf{GPR}[0]$ to co-processor #15: the co-processor registers $\mathsf{CPR}[15, 1]$ and $\mathsf{CPR}[15, 0]$ are cited as used (e.g., as the destination for the move), as well as the two (immediate) operands $0_{(10)}$ and $0_{(10)}$ (which will, presumably, control what the transfer "means" to this co-processor).

As the above hints, the MMU is configured via co-processor #15. [1, Section B3.17] offers a detailed overview of the co-processor registers, and fields within them: configuration of the MMU therefore reduces to storing concrete values into those registers via suitable use of `mcr` and `mrc` instructions.

From theory toward practice: a concrete VMSA configuration VMSA supports a range of functionality, and parameters which control that functionality: this allows configuration of a single design and implementation to suit many use-cases. As already outlined, we have limited focus in terms of functionality and hence consider a single parameterisation outlined point-by-point in the following:

- Ignoring the potential to expand the latter via LPAE, the MMU translates between a 32-bit virtual physical addresses and a 32-bit physical physical addresses. In relation to notation used in the lecture(s), this means $w = 32$.
- We set the page size to $\rho = 1\text{MiB}$: in ARM terminology, each such page is described as a section.
- This means $2^{32}/2^{20} = 2^{12} = 4096$ pages (resp. page frames) cover the virtual (resp. physical) address space, each of which can be identified via a 12-bit (section) base address. In relation to notation used in the lecture(s), this means $t = 12$.

[1] Note that the multiplication by l (i.e., ρ is not required, because we are concatenating the LHS and RHS together: the LHS is implicitly being multiplied, because it becomes the $t = 12$ MSBs.

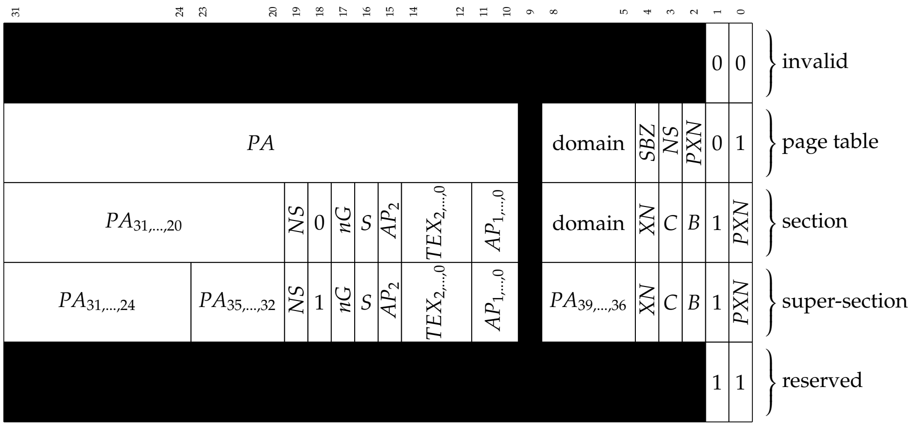

Figure 1: *VMSA short descriptor formats, per [1, Figure B3-4].*

- The page size implies use of 32-bit page table entries as described in Figure 1; in ARM terminology, these are so-called short descriptors. The resulting $\lambda = 1$ level, 4096 entry page table is $4096 \cdot 4\text{B} = 16\text{KiB}$ in size.
- Figure 1 shows numerous fields in (all) page table entry types. However, since our focus is limited we can also focus on a (very) limited set of fields. If E denotes some i-th entry for $0 \leq i < 4096$, then:
  - The two LSBs, i.e., $E_{1,\ldots,0}$, are set to $10_{(2)}$ to specify the entry type.
  - The 12-bit field $E[PA]$, i.e., $E_{31,\ldots,20}$, controls the translation process by specifying the base address of a (physical) page frame the i-th (virtual) page should map to.
  - The 4-bit field $E[\text{domain}]$, i.e., $E_{8,\ldots,5}$, controls the domain for this page: essentially this is a way to manage the access permission mechanism for a collection of pages, i.e., a domain. Each of 16 possible domains is configured via a field in DACR [1, Section B4.1.43]. In particular,

    $$\mathsf{DACR}_{2\cdot j+1,\ldots,2\cdot j} \left\{ \begin{array}{lclcl} 01_{(2)} & \Rightarrow & \text{client domain} & \Rightarrow & \text{access checking} \\ 11_{(2)} & \Rightarrow & \text{manager domain} & \Rightarrow & \text{no access checking} \end{array} \right.$$

    st. for a page configured in a manager domain, the value of $E[AP]$ is irrelevant: accesses will not even be checked, so can never fail due to the permissions set.
  - The 3-bit field $E[AP]$, which is specified in two parts, i.e., E_{15} and $E_{11,\ldots,10}$, controls the access permissions for the i-th page; in particular,

    $$E[AP] = \left\{ \begin{array}{lcr} 000_{(2)} & \Rightarrow & \text{no access in } \textit{any} \text{ processor mode} \\ 011_{(2)} & \Rightarrow & \text{access in } \textit{any} \text{ processor mode} \end{array} \right.$$

    per [1, Table B3-8].
  - All other bits are set to zero.

Putting all this together, consider an example where the 0-th entry in the page table is

$$E = 12300C02_{(16)}$$

st.

$$\begin{array}{lcr} E[PA] & = & 123_{(16)} \\ E[\text{domain}] & = & 0000_{(2)} \\ E[AP] & = & 011_{(2)} \end{array}$$

That is, (virtual) page $000_{(16)}$

- is mapped to (physical) page frame $123_{(16)}$, st. mapping of the associated 2^{20} address region can be described by

  $$\begin{array}{rcl} 00000000_{(16)} & \mapsto & 12300000_{(16)} \\ 00000001_{(16)} & \mapsto & 12300001_{(16)} \\ & \vdots & \\ 000FFFFF_{(16)} & \mapsto & 123FFFFF_{(16)} \end{array}$$

- is in domain 0: if this is configured as a client domain then the access permissions will be checked, but
- has access permissions that allow all loads and/or stores, i.e., full or unrestricted access, in any processor mode (including unprivileged, user mode).

Explore the archive content

As Figure 2 illustrates, the content and structure of the archived material provided matches worksheet #4.

Understand the archive content

`image.ld`: the linker script Figure 3 illustrates the linker script `image.ld`. It controls how `ld` produces the kernel image from object files, which, in turn, stem from compilation of the source code files; the resulting layout in memory is illustrated by Figure 4. Notice that the layout is roughly annotated, on the right-hand side, with an association between address regions and page numbers. For example, the kernel image itself is captured within the 1MiB region spanning addresses $70000000_{(16)}$ to $700FFFFF_{(16)}$ inclusive, and hence page number $700_{(16)}$.

`int.[sh]`: low-level support functionality The header file `int.h` and source code `int.s` are essentially identical to worksheet #4 (bar the specialisation to interrupt types handled), so we omit any discussion of them.

`lolevel.[sh]`: low-level kernel functionality The header file `lolevel.h` and source code `lolevel.s` are essentially identical to worksheet #4 (bar the specialisation to interrupt types handled), so we omit any discussion of them.

`hilevel.[ch]`: high-level kernel functionality All high-level, kernel-specific functionality is captured by the header file `hilevel.h` and source code `hilevel.c`. Line #37 of `hilevel.h` defines an appropriate type of PTEs, allowing Line #14 of `hilevel.c` to define `T`, a 4069-element array of PTEs constituting the page table. Crucially, the array must be aligned to a multiple of 16KiB, which is achieved using a GCC varaiable attribute documented at

http://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html

Beyond this, the source code can be viewed as two parts. The top part implements three test functions which exercise the MMU: Lines #26 to #36 implement `test_0`, Lines #38 to #50 implement `test_1`, and Lines #52 to #65 implement `test_2`. The content of the functions is best explained experimentally, so is deferred until later. The bottom part, as usual, implements some high-level interrupt handler functions:

- `hilevel_handler_rst` is invoked by `lolevel_handler_rst` every time a reset interrupt is requested and needs to be handled. The majority of the function is dedicated to configuring the MMU, which is then exercised via the test functions as outlined above. Note that configuration makes use of low-level functionality in `device/MMU.[sh]`: although understanding this might be useful in the long term, it seems sensible to ignore it now.
  - Lines #70 to #72 initialise the page table, iterating through each element in `T` st. an identity mapping is described.
  - Line #76 initialises the page table pointer, i.e., provide the base address of the page table `T` to the MMU for use during the translation process.
  - Lines #78 and #79 configure domains 0 and 1 st. the former is a manager and latter a client domain: this means pages in domain 0 will have no access permission checks applied, whereas those in domain 1 will.
  - Line #81 enables the MMU.
  - Lines #85 to #87 invoke the test functions.
- `hilevel_handler_pab` is invoked by `lolevel_handler_pab` every time a pre-fetch abort interrupt is requested and needs to be handled; likewise, `lolevel_handler_dab` and `hilevel_handler_dab` handle data abort interrupts. Neither function actually does anything, so, in essence, they represent the opportunity to control execution (e.g., using a breakpoint) and observe such interrupts being requested.

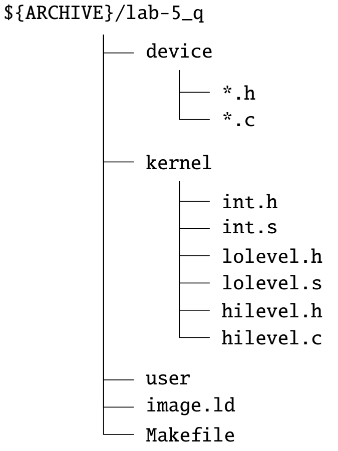

Figure 2: *A diagrammatic description of the material in* `lab-5_q.tar.gz`*.*

```
SECTIONS {
  /* assign load address (per  QEMU) */
  .        =      0x70010000;
  /* place text segment(s)            */
  .text : { kernel/lolevel.o(.text) *(.text .rodata) }
  /* place data segment(s)            */
  .data : {                         *(.data         ) }
  /* place bss  segment(s)            */
  .bss  : {                         *(.bss          ) }
  /* align        address (per AAPCS) */
  .        = ALIGN( 8 );
  /* allocate stack for svc mode      */
  .        = . + 0x00001000;
  tos_svc = .;
  /* allocate stack for abt mode      */
  .        = . + 0x00001000;
  tos_abt = .;
}
```

Figure 3: *The linker script* `image.ld`*.*

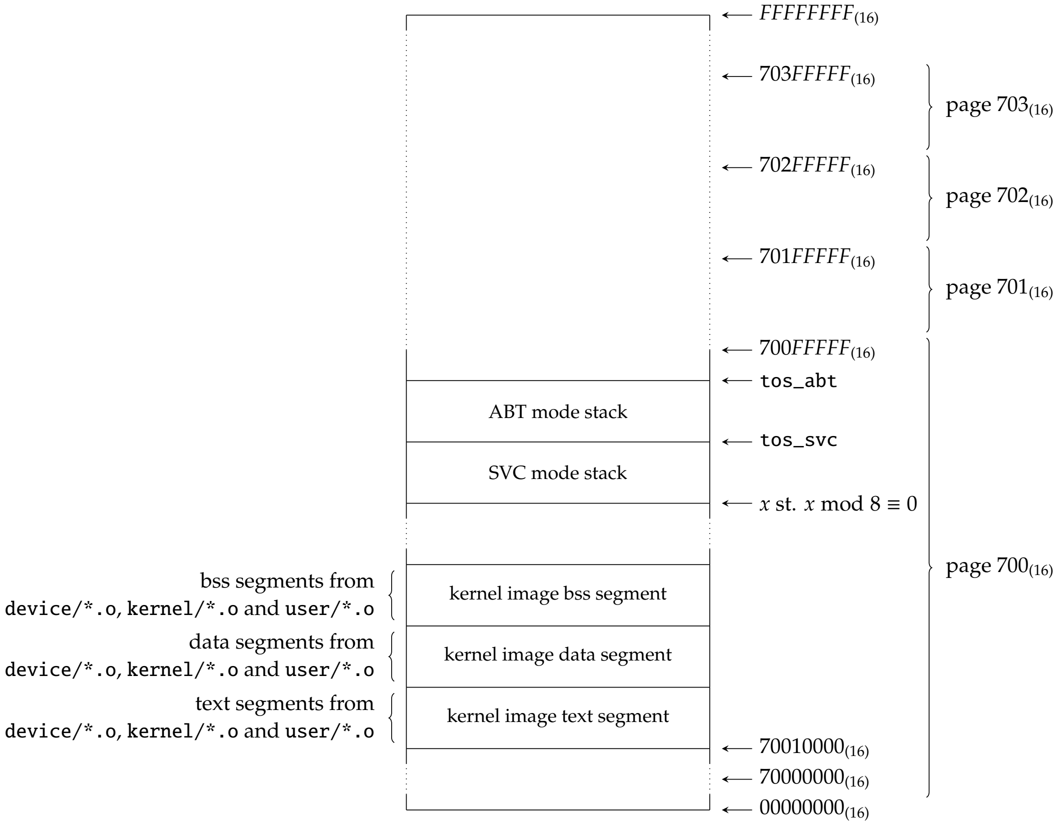

Figure 4: *A diagrammatic description of the memory layout realised by* `image.ld`*.*

Experiment with the archive content

Each case below has the same goal, namely to experiment with one of the three provided test functions. Following the same approach as worksheet #1, first launch QEMU. Next, launch `gdb`: use the debugging terminal to set a breakpoint st. execution halts on entry to the test function (e.g., via the command `break test_0`), and issue the

```
continue
```

command so the kernel image is executed.

- Consider test function `test_0`.
  - Assuming execution halted on entry to the function, the page table (i.e., content of `T`) will have been initialised by `hilevel_handler_rst`. To verify this, issue the command

    ```
    x/4096x T
    ```

    in the debugging terminal: it will print the page table content entry-by-entry, with each entry displayed as a 32-bit hexadecimal value.
  - Although picking through the output is challenging, we can inspect the same content more directly: issue the commands

    ```
    print/x T[ 0x701 ]
    print/x T[ 0x702 ]
    print/x T[ 0x703 ]
    ```

    to display the entries for pages $701_{(16)}$, $702_{(16)}$, and $703_{(16)}$. You should find, for example, that the entry for page $701_{(16)}$ is

    $$70100C02_{(16)},$$

    meaning it maps (virtual) page $701_{(16)}$ to (physical) page frame $701_{(16)}$. In fact, all entries are similar and thus implement an identity mapping (i.e., each i-th page is mapped to the i-th page frame).
  - Next, step through the invocations of `memset`. The easiest way to do so is to issue the command

    ```
    next 4
    ```

    in the debugging terminal. We expect said invocations to have initialised each byte in pages $701_{(16)}$, $702_{(16)}$, and $703_{(16)}$ to the constant values $01_{(16)}$, $02_{(16)}$, and $03_{(16)}$ respectively. We can verify this, either a) by stepping through the subsequent three assignments to `t0`, `t1` and `t2` (and so load from address zero of each associated page), or b) issuing the commands

    ```
    x/1x page_0x701
    x/1x page_0x702
    x/1x page_0x703
    ```

    in the debugging terminal to directly inspect the same content in memory.
- Consider test function `test_1`.
  - First, step through the three assignments which swap the entries relating to pages $701_{(16)}$ and $702_{(16)}$. By reinspecting them via

    ```
    print/x T[ 0x701 ]
    print/x T[ 0x702 ]
    print/x T[ 0x703 ]
    ```

    it should be clear this works as expected.
  - Next, step through invocation of `mmu_flush`. This invalidates all entries cached by the TLB, and is important because we *altered* the page table: the cached entries for pages $701_{(16)}$ and $702_{(16)}$ will therefore be incorrect until we invalidate them, at which point the TLB is forced to reload them from the page table. Strictly speaking we should perform an analagous flush of the data and instruction caches for the same reasons, but can omit this due to the simplicity of the example.
  - Now either a) step through the subsequent three assignments to `t0`, `t1` and `t2`, or b) issue the commands

    ```
    x/1x page_0x701
    x/1x page_0x702
    x/1x page_0x703
    ```

in the debugging terminal to directly inspect the same content in memory: you should see that the content has *seems* to have been swapped as well.

More precisely, however, the same content is resident in the same physical addresses: physical address $70100000_{(16)}$ still contains $01_{(16)}$ for example. Once we updated the page table by swapping the entries, we are now *accessing* that content via different virtual addresses (cf. the identify mapping). In particular, *now* the (virtual) pages $701_{(16)}$ and $702_{(16)}$ are mapped to (physical) page frames $702_{(16)}$ and $701_{(16)}$ respectively: this means virtual address $70100000_{(16)}$ maps to physical address $70200000_{(16)}$, and thus, when we load from it, we get $02_{(16)}$ rather than $01_{(16)}$.

- Consider test function `test_2`.
  - Step through the four assignments: they update the entry for page $703_{(16)}$, say E, st.

$$\begin{array}{lcl} E[\text{domain}] & = & 0001_{(2)} \\ E[AP] & = & 000_{(2)} \end{array}$$

    That is, after the update the page is a) in domain 1 vs. 0, meaning a client domain with access checking enabled, and b) assigned access permissions that disallow all loads and/or stores.
  - Next, step through invocation of `mmu_flush`: as above, we must invalidate cached entries TLB because we altered the page table.
  - Now step through the subsequent three assignments to `t0`, `t1` and `t2`. The first two should execute as expected, but the third requests an data abort exception handled first by `lolevel_handler_dab` then `hilevel_handler_dab`. The reason should be obvious: in the above, we set the access permissions for the associated page so all loads are disallowed. So, when we then execute one from an address in that page, the MMU requests an exception and thereby allows the kernel to handler the situation; if the access was by a user process, for example, it might terminate that process.

Next steps

There are various things you could (optionally) do next: here are some ideas.

a The use of `mmu_flush` to flush the TLB is explained and motivated. What happens if you *remove* this: can you make sense of the resulting behaviour by explaining what the MMU is doing?

b A mechanism for handling pre-fetch and data interrupts has been provided. Investigate what these interrupts mean, e.g., clarify when and why they might be requested.

c Investigate the mechanism by which the cause of such interrupts can be determined; use this to write and experiment with a wider set of test functions that exercise specific cases.

d There are lots of ways you could extend the functionality within `MMU.[sh]`, and then make use of it. For example,

- At the moment only TTBR0 is used, although the MMU supports use of a second page table register TTBR1 (under control of TTBCR): can you make use of this somehow?
- In the above, a need to flush data and instruction caches is mentioned: can you write a function that supports doing so?

Q2[A]. In a nutshell, system programming[2] is the task of producing system software: unlike application software, it typically a) operates at a lower level of abstraction, often interacting more directly with the kernel and/or hardware devices say, b) is performance critical, and c) provides functionality to other software rather than (human) users. Various forms of documentation can be helpful, with UNIX-style manual pages[3] often representing a canonical source of information for system calls etc. However, these are often written in a fairly formal and technical style, so, although acting as an effective reference, they are normally less ideal as a learning resource. Fortunately, a range of accessible alternatives are available online. The set of guides at

http://beej.us/guide

[2] http://en.wikipedia.org/wiki/System_programming

[3] The `man` (or manual) command can be used to display said documentation via the command line. Two options are extremely useful: the `-k` option supports search for a keyword, and the `-s` option supports specification of a section within the manual (and hence disambiguation of keywords occurring in multiple entries). For example, `man -k send` will search for all entries including the keyword `send`, and `man -s 2 send` will display the specific entry within section 2.

represents a good example.

The various POSIX-based APIs for Inter-Process Communication (IPC) are an example of system-level interaction best explored using a hands-on, practical approach. The guide at

http://beej.us/guide/bgipc

supports such an approach, including detailed example source code.

References

[1] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html. 2014 (see pp. 1–3).