COMS20001 lab. worksheet #2

- 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 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 lab. slot. Your attendance is important, because the lab. slot represents a primary source of formative feedback and help. Note that, perhaps more so than in units from earlier years, *you* need to actively ask questions of and/or seek help from 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/ycgk8pce/csdsp/os/sheet/lab-2_q.tar.gz

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] Use the `gz` and `tar` commands within a BASH shell (or prompt, e.g., a terminal window) or similar, *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 within your home directory (which, by default, cannot be read by third-parties).

Q1[L]. This question acts as a practical exploration of interrupts, interrupt handling, and system calls. Doing so demands that you understand the hardware/software interface, meaning the content of this worksheet a) has more low-level detail than some others, and b) is important because it act as a basis on which most higher-level functionality is built.

Note that although the lab. worksheet does conclude with a set of hands-on tasks and challenges, at this point they are intentionally biased towards reading and understanding the material (vs. more active alternatives, e.g., programming). It is *crucial* not to view this as optional effort: carefully working through the admittedly detailed content should allow you to more easily and rapidly engage with longer-term challenges (e.g., those relating to a given coursework assignment).

Q1–§1 Explore the archive content

As Figure 1 illustrates, the content and structure of the archived material provided matches worksheet #1. The only difference is some additional files within the `kernel` directory, which are explained below.

Q1–§2 Understand the archive content

`image.ld`: the linker script Figure 2 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 3.

`int.[sh]`: low-level support functionality The header file `int.h` and source code `int.s` act as support functionality for the kernel, specifically relating to control of the interrupt handling mechanism. The latter is basically divided into two parts:

- The top part relates to initialisation of the interrupt handling mechanism. It exists to solve a particular problem. Recall that in order to function correctly, the interrupt vector table must be at address $00000000_{(16)}$ because this is what the processor expects. However, we know QEMU always loads the kernel image at

```
${ARCHIVE}
├── device
│    ├── *.h
│    └── *.c
├── kernel
│    ├── int.h
│    ├── int.s
│    ├── lolevel.h
│    ├── lolevel.s
│    ├── hilevel.h
│    └── hilevel.c
├── user
├── image.ld
└── Makefile
```

Figure 1: *A diagrammatic description of the material in* `lab-2_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 irq mode     */
  .        = . + 0x00001000;
  tos_irq = .;
  /* allocate stack for svc mode     */
  .        = . + 0x00001000;
  tos_svc = .;
}
```

Figure 2: *The linker script* `image.ld`*.*

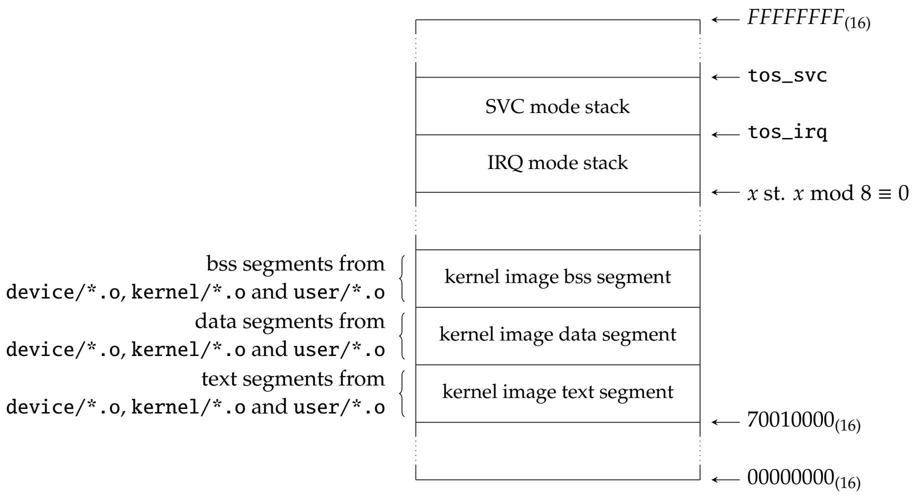

Figure 3: *A diagrammatic description of the memory layout realised by* `image.ld`*.*

address $70010000_{(16)}$, so cannot *directly* load the interrupt vector table where we want it. The solution is to load it somewhere else instead, then copy it into place:

  - Lines #18 to #29 define the 8-entry interrupt vector table, including a populated entry for each interrupt type that is handled.
  - Lines #33 to #43 implement a function `int_init` that copies the interrupt vector table to address $00000000_{(16)}$ and so initialises it ready for use.

- The bottom part relates to management of the IRQ and FIQ interrupt signals, or, more specifically, associated fields in CPSR which allow them to be enabled (or unmasked) and disabled (or masked):

  - Lines #54 to #64 implement two functions `int_enable_irq` and `int_unable_irq` which enable and disable IRQ interrupts respectively.
  - Lines #66 to #76 implement two functions `int_enable_fiq` and `int_unable_fiq` which enable and disable FIQ interrupts respectively.

`lolevel.[sh]`: low-level kernel functionality All low-level, kernel-specific functionality is captured by the header file `lolevel.h` and source code `lolevel.s`. The latter captures implementation of three low-level interrupt handler functions, one for each type that could be requested:

- Lines #17 to #25 implement `lolevel_handler_rst`: this function is similar to the one in worksheet #1, except it now a) invokes `int_init` to initialise the interrupt vector table, b) initialises *both* SVC *and* IRQ mode stacks, then, finally, c) invokes `hilevel_handler_rst`.

- Lines #27 to #33 and Lines #35 to #41 implement `lolevel_handler_irq` and `lolevel_handler_svc` respectively. These functions are similar in form: they handle the two remaining types, namely IRQ and supervisor call interrupts. Using `lolevel_handler_irq` as an example, it a) corrects[1] the return address, b) pushes all caller-save registers to the IRQ mode stack, c) invokes `hilevel_handler_irq`, then, after it returns, d) pops all caller-save registers from the IRQ mode stack, before finally e) returning to wherever execution was stopped in order to handle the interrupt.

`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`. The latter implements three high-level interrupt handler functions: in this example the goal is to demonstrate when interrupts are requested and handled (rather than perform any particular, meaningful behaviour), so each function is fairly simple.

- `hilevel_handler_rst` is invoked by `lolevel_handler_rst` every time a reset interrupt is requested and needs to be handled:

  - Lines #20 and #21 configure the emulated UART, namely the `PL011_t` instance `UART0`: the comments briefly describe each step, which each amount to setting various device registers appropriately. In short, the UART is configured st. it requests an interrupt each time it receives a byte.
  - Lines #23 to #26 configure the GIC, namely the `GICC_t` and `GICD_t` instances `GICC0` and `GICD0`: the comments briefly describe each step, which each amount to setting various device registers appropriately. In short, the GIC is configured st. the UART interrupt signal (i.e., #44) is distributed (or connected) to the processor IRQ interrupt.
  - Line #28 then enables IRQ interrupts wrt. the processor: this unmasks the IRQ interrupt signal, meaning any interrupt from the GIC is now "visible" to and so handled by the processor.
  - Lines #38 to #44 are *somewhat* similar way to worksheet #1, in the sense an infinite outer `while` loop implies the function never returns. Each iteration executes

    * a number of `nop` instructions intended to realise a delay for some fixed period, then
    * a `svc` instruction intended to request a supervisor call interrupt, i.e., perform a system call.

- `hilevel_handler_irq` is invoked by `lolevel_handler_irq` every time a IRQ interrupt is requested and needs to be handled. Three of the steps in [2, Section 4.11.3] are needed here, which are numbered to match. Although the steps numbered #2 and #5 are standard boiler-plate we need to include for *any* handler of this type, #4 deals specifically with interrupts from the UART: it first tests whether or not the interrupt stems from the UART, then, if so, takes some action to handle it. In this case, the action is captured as follows:

[1] [1, Table 2-12] dictates the offset used for correction. Note that `lolevel_handler_svc` uses an offset of 0, meaning that `sub lr, lr, #0` could in fact be omitted: it has no effect on `lr`. It is retained, however, so each interrupt handler has the same form, i.e., perform the same steps irrespective of interrupt type; doing so arguably makes explaining those steps easier.

  - Line #57 receives some input via the `PL011_t` instance `UART0` by invoking `PL011_getc`: since we *know* the interrupt must have been requested due to a byte being received, we assume this is the case. That is, we omit any checking as why the UART requested the interrupt, which may be required in a general context.
  - Lines #59 to #63 transmit some output via the `PL011_t` instance `UART0` by invoking `PL011_putc`, thus demonstrating the interrupt was handled.
  - Line #65 is fairly crucial: it basically signals to the interrupt source, in this case the UART device, that the interrupt it has requested was handled. Put another way, it clears (or cancels) the interrupt and so resets the interrupt generation logic in the device; this means it will then generate subsequent interrupts rather than mistakenly getting "stuck" with the current one.

- `hilevel_handler_svc` is invoked by `lolevel_handler_svc` every time a supervisor call interrupt is requested and needs to be handled: this occurs whenever a `svc` instruction is executed. Line #81 transmits some output via the `PL011_t` instance `UART0` by invoking `PL011_putc`, thus demonstrating the interrupt was handled.

Q1–§3 Experiment with the archive content

Following the same approach as worksheet #1, first launch QEMU. Next, launch `gdb` and issue the

```
continue
```

command in the debugging terminal so the kernel image is executed. Assuming the emulation terminal has the UI focus, you should observe two different behaviours:

- whenever you press a key, a 'K' character plus the hexadecimal key-code of the key pressed are written to the emulation terminal: this demonstrates an IRQ interrupt was requested by the UART and then handled first by `lolevel_handler_irq` and then by `hilevel_handler_irq`, and
- periodically, without any user interaction, a 'T' character is written to the emulation terminal: this demonstrates a supervisor call interrupt was requested and then handled first by `lolevel_handler_svc` and then by `hilevel_handler_svc`.

Note that wrt. execution of instructions by the processor, pressing a key is an asynchronous event: it could occur at *any* time. However, a supervisor call interrupt is synchronous since it occurs as the result of executing a `svc` instruction. Put another way, the behaviour of this kernel could be summarised as follows. It is basically "stuck" in an infinite loop, which it enters when the processor is reset. The loop periodically executes a `svc` instruction to cause a supervisor call interrupt (and therefore write a 'T' character). However, when a key is pressed, it suspends this behaviour to handle the resulting IRQ interrupt (by writing a 'K' character); once complete, execution of the loop is resumed.

Q1–§4 Next steps

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

a An effective way to gain insight into an example of this type, in which the timing of events is crucial, is by drawing a time-line to illustrate said events. For example, by including a) when events (e.g., interrupts) occur, b) what the processor and memory (e.g., stack) state is, and c) what instructions (i.e., what function) is being executed at different periods of time, one can align the observed behaviour with the source code. Try to construct such a diagram for this example, limiting the duration to the first few seconds at most (but including at least one key press event).

b Within `hilevel_handler_rst` a `for` loop performs 2^{20} `nop` instructions to realise a delay of ~ 1s. However, the number required depends on the underlying hardware QEMU is executed on: if it can execute more (emulated) instructions per second, the loop needs to perform more iterations to compensate. Using a hard-coded number of iterations is therefore less than ideal: can you design and ideally implement a mechanism to discover (i.e., calibrate) the number of iterations needed for any underlying hardware?

Q2[A]. Consider the `printf` (for "print formatted") function, which is provided by the C standard library. In reality, each invocation of `printf` (and variants) results in a 2-step process: it

- parses the format string, populating it with the (variable length list of) arguments, then

- performs a system call into the kernel st. the formatted result is written to a file descriptor (e.g., typically `stdout` by default).

From the perspective of a caller, this approach offers a convenient API. For example, the traditional[2] "hello world" program is (by design) fairly trivial:

```
#include <stdio.h>

int main( int argc, char* argv[] ) {
  printf( "hello world\n" );

  return 0;
}
```

To achieve the required result, however, `printf` must interact with the kernel by performing system calls. So what if `printf` were unavailable? This question asks you to develop an alternative to the program above. The goal is to write the string "hello world" to `stdout` as before, but to do so by performing the system calls `printf` would *yourself*; the rationale is that doing this forces you to explore and hence understand the user-facing side of the system call interface, which complements the kernel-facing side studied via QEMU.

Your exact solution will depend on various concrete facts, such as the kernel (e.g., Windows vs. Linux) and processor type (e.g., ARM vs. x86). So, for the sake of concretness, imagine you develop your solution on a lab. workstation with a Linux 3.10.x kernel (viz. Centos 7) on an x86-64 (Core i7) processor. The ideal, step-by-step approach would be:

- Replace the implementation that uses `printf` with an equivalent that uses `write`; this is one step lower-level, since `write` is a wrapper around a system call of the same name.
- Replace the implementation that uses `write` with an equivalent that uses `syscall`; this is one step lower-level, since `syscall` is a generic way to perform any system call provided you know the system call identifier (i.e., which corresponds to `write`).
- Replace the implementation that uses `syscall` with an equivalent that uses inline assembly language; this is one step lower-level, since now you need to replicate what `syscall` itself does. This means using the system calling convention for the processor (e.g., x86) you intend to execute the result on.

If you *really* get stuck,

http://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

offers an introduction of sorts.

Q3[A]. `strace` (or "system trace") is a tool which can trace the systems calls made, signals received and resources used by some process (and any sub-processes). `strace` provides a vast range of functionality, but even a basic understanding of how to use it can help you understand and debug programs: it is particularly valuable when the associated source code is not available (so cannot be recompiled, and therefore potentially not easily debugged using `gdb`), or where an *executing* process needs to be debugged *without* (necessarily) interfering with or terminating it.

It *also* offers an excellent way to see how programs interact with the kernel, the details of which are normally (by design) abstracted by mechanisms such as the C standard library (per the question above). This question is vague and open ended: the goal is simply to point out `strace` exists. So, try it out: execute the command

```
strace ls
```

and marvel at how many (and which) system calls `ls` needs to produce a list of files in the current directory. Based on this, think about some previous coursework: can you optimise your solution somehow based on what `strace` shows about how it behaves?

References

[1] *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. ARM Ltd., 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html (see p. 3).

[2] *RealView Platform Baseboard for Cortex-A8*. Tech. rep. HBI-0178. ARM Ltd., 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html (see p. 3).

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