COMS20001 lab. worksheet: week #14

- 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 in room MVB-3.41, submit a service request online via

  http://servicedesk.bristol.ac.uk

  or talk to the dedicated CS Teaching Technologist, Richard Grafton, in room MVB-2.07.
- We intend this worksheet to be attempted, at least partially, in the associated lab. session. Your attendance is important, since this session represents a central form of 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: passively expecting them to provide solutions is less ideal.
- 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 unarchive[a] the file

http://www.ole.bris.ac.uk/bbcswebdav/courses/COMS20001_2015/csdsp/os/sheet/lab/lab-2_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] Execute the command `tar xvfz lab-2_q.tar.gz` from a BASH shell (e.g., in a terminal window), or use the archive manager GUI (available by using the menu `Applications`→`Accessories`→`Archive Manager` or directly executing `file-roller`) if you prefer.

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

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

Although the worksheet concludes with a set of challenges relating to experimental exploration, the fundamental goal throughout is improved understanding of the material. Put another way, the work you do will be biased toward reading and understanding rather than programming at this point. It is crucial *not* to view this as optional effort: carefully working through what is, admittedly, a detailed worksheet will allow you to more easily and rapidly engage with the intellectual vs. engineering challenges involved.

Q1–§1 Explore the archive content

As shown by Figure 1, the content and structure of the archived material provided matches the worksheet from week #13.

Q1–§2 Understand the archive content

`image.ld` 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 2.

`interrupt.[sh]` Figure 4 and Figure 6 illustrate the header file `interrupt.h` and source code `interrupt.s`. Noting the former now has some content, unlike the worksheet from week #13, the latter can be considered as two parts:

- The bottom part supports initialisation of the interrupt handling mechanism, i.e., how interrupt handlers will be invoked. The existence of this part is motivated by a problem: in order to function correctly, we know the interrupt vector table needs to be placed at address $00000000_{(16)}$. *However*, due to the way QEMU works (i.e., the fact it will load the kernel image at address $70010000_{(16)}$) it cannot be loaded *directly* at this address. The solution is to load it somewhere *else* instead, then copy it into place.

  - Lines #42 to #49 define the 8-entry interrupt vector table, with one entry per interrupt type.
  - Lines #55 to #65 implement a function `table_copy` that copies the interrupt vector table to address $00000000_{(16)}$.
  - Lines #74 to #84 implement two functions `irq_enable` and `irq_unable` which enable and disable IRQ interrupts respectively; since these have global scope, and have a matching prototype in `interrupt.h`, all of the high-level functions in `kernel.c` can use this low-level functionality *without* resorting to inline assembly language.

- The top part captures three low-level interrupt handler implementations, one for each type that could be invoked using the interrupt handling mechanism described above.

  - Lines # 6 to #14 implement `handler_rst`: this function is similar to the one found in the worksheet from week #13, except that it now a) starts by calling `table_copy`, *then* b) initialise both the SVC *and* IRQ mode stacks, before finally c) invoking `kernel_handler_rst`.
  - Lines #16 to #22 implement `handler_irq`, while lines #24 to #30 implement `handler_svc`. These functions are similar in form: they handle the two remaining types, namely IRQ and supervisor call interrupts. Using `handler_irq` as an example, it a) corrects the return address per the ARM documentation, b) pushes all caller-save registers to the IRQ mode stack, c) invokes `kernel_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.

`kernel.[ch]` Figure 5 and Figure 7 illustrate the header file `kernel.h` and source code `kernel.c`. The latter implements three high-level interrupt handler functions: in this example the goal is to demonstrate when such interrupts are raised and handled (rather than perform some meaningful behaviour), so each function is fairly simple.

- `kernel_handler_rst` is invoked by `handler_rst` every time a reset interrupt is raised and needs to be handled.

  - Lines #13 and #14 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 raise an interrupt each time it receives a byte.
  - Lines #16 to #19 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 signal.
  - Line #21 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 #31 to #37 are *somewhat* similar way to the worksheet from week #13, 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 raise a supervisor call interrupt, i.e., perform an system call.

- `kernel_handler_irq` is invoked by `handler_irq` every time a IRQ interrupt is raised and needs to be handled. Three of the steps in [1, 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:

  - Line #50 receives some input via the `PL011_t` instance `UART0` by invoking `PL011_getc`: since we *know* the interrupt must have been raised due to a byte being received, we *assume* this is the case. That is, we omit any checking as why the UART raised the interrupt, which might be required in a general context.
  - Lines #52 to #55 transmit some output via the `PL011_t` instance `UART0` by invoking `PL011_putc`, thus demonstrating the interrupt was handled.
  - Line #57 is fairly crucial: it basically signals to the interrupt source, in this case the UART device, that the interrupt it has raised 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.

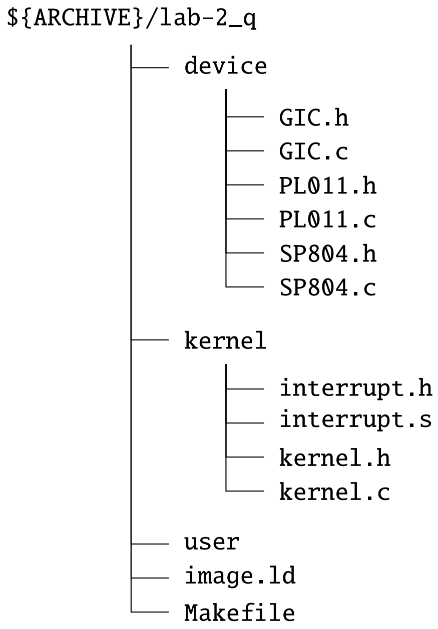

Figure 1: *A diagrammatic description of the material in* `lab-2_q.tar.gz`*.*

- `kernel_handler_svc` every time a supervisor call interrupt is raised and needs to be handled: this occurs whenever a `svc` instruction is executed. Line #71 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 in the worksheet from week #13, first launch QEMU then `gdb`. Issue the

```
continue
```

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

- periodically, without any user interaction, a 'T' character is written to the emulation terminal: this demonstrates a supervisor call interrupt was raised and then handled first by `handler_svc` and then by `kernel_handler_svc`, and also

- whenever you press a key, a 'K' character plus the key pressed are written to the emulation terminal: this demonstrates an IRQ interrupt was raised (i.e., by the UART) and then handled first by `handler_irq` and then by `kernel_handler_irq`.

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).

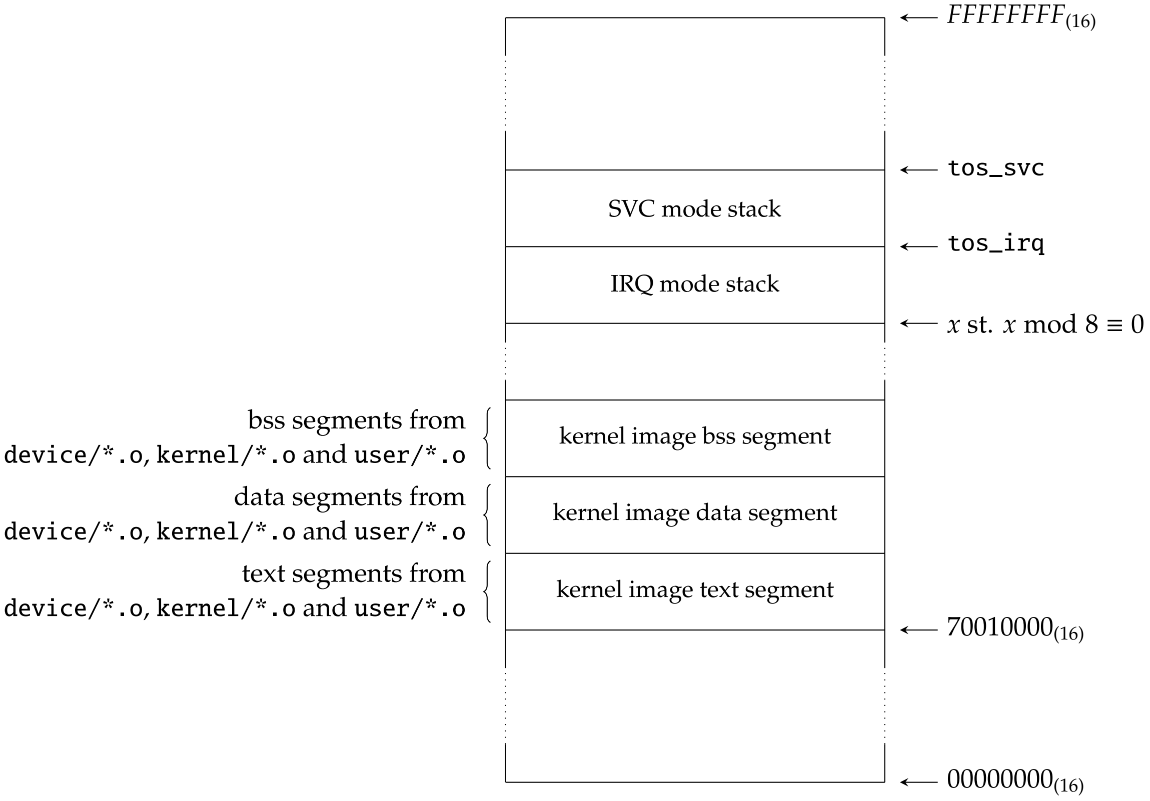
FFFFFFFF(16)
tos_svc
SVC mode stack
tos_irq
IRQ mode stack
x st. x mod 8 ≡ 0
bss segments from device/*.o, kernel/*.o and user/*.o
kernel image bss segment
data segments from device/*.o, kernel/*.o and user/*.o
kernel image data segment
text segments from device/*.o, kernel/*.o and user/*.o
kernel image text segment
70010000(16)
00000000(16)

Figure 2: *A diagrammatic description of the memory layout realised by* `image.ld`*.*

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
- perform 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[1] "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`).

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

```
SECTIONS {
  /* assign address (per  QEMU)  */
  .        =     0x70010000;
  /* place text segment(s)       */
  .text : { kernel/interrupt.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 3: *image.ld*

```
#ifndef __INTERRUPT_H
#define __INTERRUPT_H

//  enable IRQ interrupts
extern void irq_enable();
// disable IRQ interrupts
extern void irq_unable();

#endif
```

Figure 4: *kernel/interrupt.h*

```
#ifndef __KERNEL_H
#define __KERNEL_H

#include <stddef.h>
#include <stdint.h>

#include   "GIC.h"
#include "PL011.h"
#include "SP804.h"

#include "interrupt.h"

#endif
```

Figure 5: *kernel/kernel.h*

```
/* Each of the following is a low-level interrupt handler: each one is
 * tasked with handling a different interrupt type, and acts as a sort
 * of wrapper around a high-level, C-based handler.
 */

handler_rst: bl    table_copy             @ initialise interrupt vector table

             msr   cpsr, #0xD2            @ enter IRQ mode with no interrupts
             ldr   sp, =tos_irq           @ initialise IRQ mode stack
             msr   cpsr, #0xD3            @ enter SVC mode with no interrupts
             ldr   sp, =tos_svc           @ initialise SVC mode stack

             bl    kernel_handler_rst     @ invoke C function
             b     .                      @ halt

handler_irq: sub   lr, lr, #4             @ correct return address
             stmfd sp!, { r0-r3, ip, lr } @ save    caller-save registers

             bl    kernel_handler_irq     @ invoke C function

             ldmfd sp!, { r0-r3, ip, lr } @ restore caller-save registers
             movs  pc, lr                 @ return from interrupt

handler_svc: sub   lr, lr, #0             @ correct return address
             stmfd sp!, { r0-r3, ip, lr } @ save    caller-save registers

             bl    kernel_handler_svc     @ invoke C function

             ldmfd sp!, { r0-r3, ip, lr } @ restore caller-save registers
             movs  pc, lr                 @ return from interrupt

/* The following captures the interrupt vector table, plus a function
 * to copy it into place (which is called on reset): note that
 *
 * - for interrupts we don't handle an infinite loop is realised (to
 *   to approximate halting the processor), and
 * - we copy the table itself, *plus* the associated addresses stored
 *   as static data: this preserves the relative offset between each
 *   ldr instruction and wherever it loads from.
 */

table_data:  ldr   pc, address_rst        @ reset                 vector -> SVC mode
             b     .                      @ undefined instruction vector -> UND mode
             ldr   pc, address_svc        @ supervisor call       vector -> SVC mode
             b     .                      @ abort (prefetch)      vector -> ABT mode
             b     .                      @ abort     (data)      vector -> ABT mode
             b     .                      @ reserved
             ldr   pc, address_irq        @ IRQ                   vector -> IRQ mode
             b     .                      @ FIQ                   vector -> FIQ mode

address_rst: .word handler_rst
address_svc: .word handler_svc
address_irq: .word handler_irq

table_copy:  mov   r0, #0                 @ set destination address
             ldr   r1, =table_data        @ set source      address
             ldr   r2, =table_copy        @ set source      limit

table_loop:  ldr   r3, [ r1 ], #4         @ load  word, inc. source      address
             str   r3, [ r0 ], #4         @ store word, inc. destination address

             cmp   r1, r2
             bne   table_loop             @ loop if address != limit

             mov   pc, lr                 @ return

/* These function enable and disable IRQ interrupts respectively, by
 * toggling the 7-th bit of CPSR to either 0 or 1.
 */

.global irq_enable
.global irq_unable

irq_enable:  mrs   r0,   cpsr             @  enable IRQ interrupts
             bic   r0, r0, #0x80
             msr   cpsr_c, r0

             mov   pc, lr

irq_unable:  mrs   r0,   cpsr             @ disable IRQ interrupts
             orr   r0, r0, #0x80
             msr   cpsr_c, r0

             mov   pc, lr
```

Figure 6: *kernel/interrupt.s*

```
#include "kernel.h"

void kernel_handler_rst() {
  /* Configure the mechanism for interrupt handling by
   *
   * - configuring UART st. an interrupt is raised every time a byte is
   *   received,
   * - configuring GIC st. the selected interrupts are forwarded to the
   *   processor via the IRQ interrupt signal, then
   * - enabling IRQ interrupts.
   */

  UART0->IMSC           |= 0x00000010; // enable UART    (Rx) interrupt
  UART0->CR              = 0x00000301; // enable UART (Tx+Rx)

  GICC0->PMR             = 0x000000F0; // unmask all          interrupts
  GICD0->ISENABLER[ 1 ] |= 0x00001000; // enable UART    (Rx) interrupt
  GICC0->CTLR            = 0x00000001; // enable GIC interface
  GICD0->CTLR            = 0x00000001; // enable GIC distributor

  irq_enable();

  /* Force execution into an infinite loop, each iteration of which will
   *
   * - delay for some period of time, which is realised by executing a
   *   large, fixed number of nop instructions in an inner loop, then
   * - execute a supervisor call (i.e., svc) instruction, thus raising
   *   a software-interrupt (i.e., a trap or system call).
  */

  while( 1 ) {
    for( int i = 0; i < 0x20000000; i++ ) {
      asm volatile( "nop" );
    }

    asm volatile( "svc 0" );
  }

  return;
}

void kernel_handler_irq() {
  // Step 2: read  the interrupt identifier so we know the source.

  uint32_t id = GICC0->IAR;

  // Step 4: handle the interrupt, then clear (or reset) the source.

  if( id == GIC_SOURCE_UART0 ) {
    uint8_t x = PL011_getc( UART0 );

    PL011_putc( UART0, 'K' );
    PL011_putc( UART0, '<' );
    PL011_putc( UART0,  x  );
    PL011_putc( UART0, '>' );

    UART0->ICR = 0x10;
  }

  // Step 5: write the interrupt identifier to signal we're done.

  GICC0->EOIR = id;
}

void kernel_handler_svc() {
  /* Each time execution reaches this point, we are tasked with handling
   * a supervisor call (aka. software interrupt, i.e., a trap or system
   * call).
   */

  PL011_putc( UART0, 'T' );
}
```

Figure 7: *kernel/kernel.c*

- 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 from COMS10002: can you optimise your solution somehow based on what strace shows about how it behaves?

References

[1] ARM Limited. RealView Platform Baseboard for Cortex-A8. Technical Report HBI-0178, 2011. http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html.