COMS20001 lab. worksheet: week #16

- 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-4_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-4_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]. In comparison to previous worksheets, this question tasks you with a more active development role. It provides a starting point which demonstrates how to configure and handle interrupts from one of the SP804 timers, then concludes with a challenge: the idea is to extend the kernel presented in week #15 (which depended on each process cooperatively invoking the `yield` system call) so it supports pre-emptive multi-tasking and thus *enforces* periodic context switches.

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`: the former is identical to, and the latter similar those provided in the worksheet for week #14. The difference is that *these* have no support for supervisor call interrupts: only IRQ interrupts are handled, so the previous `handler_swi` function is missing, as is the associated entry in the interrupt vector table.

`kernel.[ch]` Figure 5 and Figure 7 illustrate the header file `kernel.h` and source code `kernel.c`: the former is identical to, and the latter similar those provided in the worksheet for week #14. As above, the difference stems mainly from the fact this kernel handles IRQ interrupts only. However, it is also true that both `kernel_handler_rst` and `kernel_handler_irq` differ because the *type* of IRQ interrupts is different: previously we were interested in interrupts from a UART, whereas now we focus on a timer. This means

- Lines #13 to #22, which configure the interrupt handling mechanism, are different. Note that a) most obviously the `SP805_t` instance `TIMER0` is first configured st. an interrupt is raised every 2^{20} timer ticks,

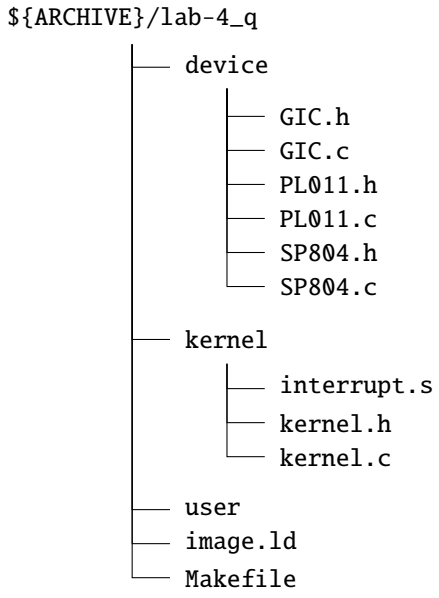

Figure 1: *A diagrammatic description of the material in* `lab-4_q.tar.gz`.

then b) the GIC configuration is largely similar, bar the fact a different source (i.e., #32 for the timer vs. #44 for the UART) is enabled. As previously, the final configuration step is then to enable IRQ interrupts, i.e., turn off the mask preventing them being handled by the processor.

- Lines #36 to #38, which represent the main interrupt handling step, are different. We now test whether the timer raised the interrupt, and, if so, handle it; once complete, we clear the timer interrupt and thus reset it st. a subsequent interrupt is generated 2^{20} timer ticks later.

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. You should observe a sequence of 'T' characters written periodically to the emulation terminal: this demonstrates an IRQ interrupt was raised by the timer, and susequently handled by `handler_irq` and `kernel_handler_irq`.

Recall that the worksheet from week #14 *also* wrote a sequence of 'T' characters to the emulation terminal. However, keep in mind a subtle but important difference. Previously the characters were being written via a (synchronous) system call; this implies the processor was *actively* executing the associated `svc` instructions, so unable to do anything else. The situation here is different because the timer operates concurrently with the processor, so the latter will now be *inactive* between IRQ interrupts: although we are not making it do so here, it *could* be doing something else before that something is interrupted and control passes to the kernel.

Q1–§4 Next steps

The behaviour described above is vital wrt. realisation of pre-emptive multi-tasking, the central concept in which is that a timer interrupts the execution of a user process after some period *rather* than it needing to invoke a `yield` system call. As such, the obvious next step is one that also acts as a direct step towards the coursework assignment. The worksheet from week #15 already provided a simple kernel with cooperative multi-tasking: take that, and upgrade it (using the material from this worksheet) to support pre-emptive multi-tasking. Note that having doing so, the `yield` system call is no longer required; you could retain it or remove it, but each user program need not invoke it (and *should* not, when testing the new kernel at least).

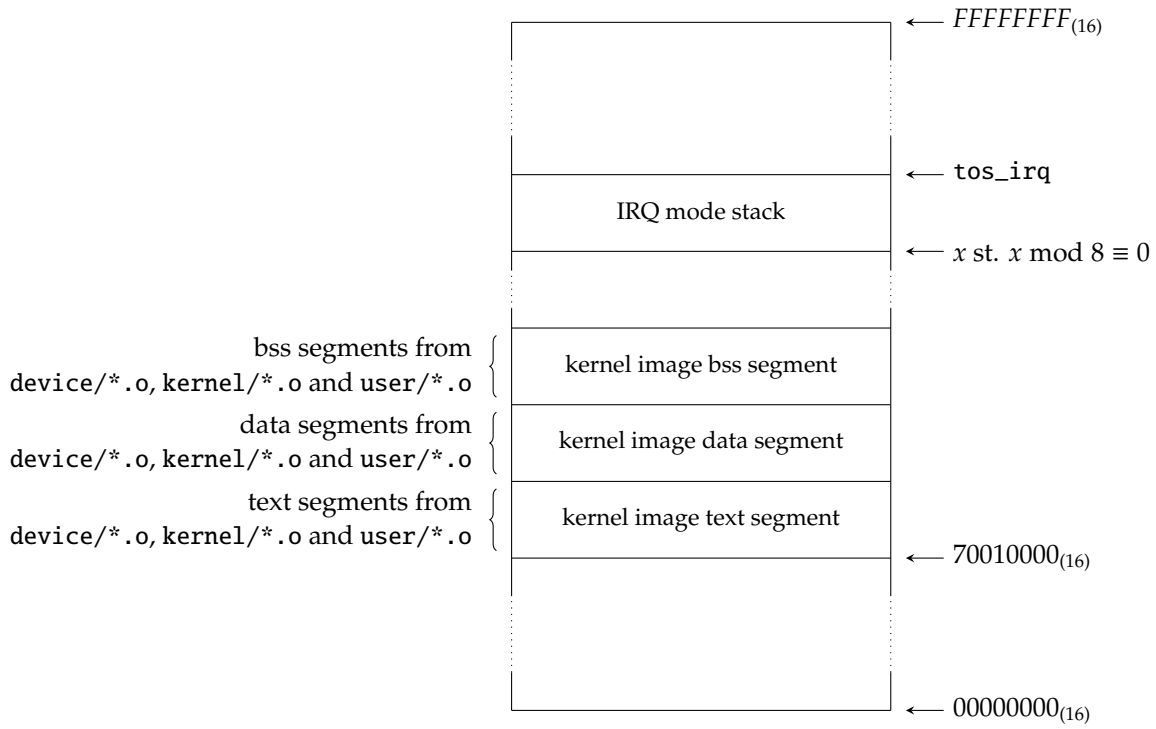

Figure 2: *A diagrammatic description of the memory layout realised by* `image.ld`.

```
1  SECTIONS {
2    /* assign address (per  QEMU)  */
3    .       =     0x70010000;
4    /* place text segment(s)       */
5    .text : { kernel/interrupt.o(.text) *(.text .rodata) }
6    /* place data segment(s)       */
7    .data : {                         *(.data        ) }
8    /* place bss  segment(s)       */
9    .bss  : {                         *(.bss         ) }
10   /* align  address (per AAPCS)  */
11   .       = ALIGN(8);
12   /* allocate stack for irq mode */
13   .       = . + 0x00001000;
14   tos_irq = .;
15 }
```

Figure 3: *image.ld*

```
1  #ifndef __INTERRUPT_H
2  #define __INTERRUPT_H
3
4  //  enable IRQ interrupts
5  extern void irq_enable();
6  // disable IRQ interrupts
7  extern void irq_unable();
8
9  #endif
```

Figure 4: *kernel/interrupt.h*

```
1  #ifndef __KERNEL_H
2  #define __KERNEL_H
3
4  #include <stddef.h>
5  #include <stdint.h>
6
7  #include   "GIC.h"
8  #include "PL011.h"
9  #include "SP804.h"
10
11 #include "interrupt.h"
12
13 #endif
```

Figure 5: *kernel/kernel.h*

```
1  /* Each of the following is a low-level interrupt handler: each one is
2   * tasked with handling a different interrupt type, and acts as a sort
3   * of wrapper around a high-level, C-based handler.
4   */
5
6  handler_rst: bl    table_copy              @ initialise interrupt vector table
7
8               msr   cpsr, #0xD2             @ enter IRQ mode with no interrupts
9               ldr   sp, =tos_irq            @ initialise IRQ mode stack
10
11              bl    kernel_handler_rst      @ invoke C function
12              b     .                       @ halt
13
14 handler_irq: sub   lr, lr, #4              @ correct return address
15              stmfd sp!, { r0-r3, ip, lr }  @ save    caller-save registers
16
17              bl    kernel_handler_irq      @ invoke C function
18
19              ldmfd sp!, { r0-r3, ip, lr }  @ restore caller-save registers
20              movs  pc, lr                  @ return from interrupt
21
22 /* The following captures the interrupt vector table, plus a function
23  * to copy it into place (which is called on reset): note that
24  *
25  * - for interrupts we don't handle an infinite loop is realised (to
26  *   to approximate halting the processor), and
27  * - we copy the table itself, *plus* the associated addresses stored
28  *   as static data: this preserves the relative offset between each
29  *   ldr instruction and wherever it loads from.
30  */
31
32 table_data:  ldr   pc, address_rst         @ reset                 vector -> SVC mode
33              b     .                       @ undefined instruction vector -> UND mode
34              b     .                       @ supervisor call       vector -> SVC mode
35              b     .                       @ abort (prefetch)      vector -> ABT mode
36              b     .                       @ abort     (data)      vector -> ABT mode
37              b     .                       @ reserved
38              ldr   pc, address_irq         @ IRQ                   vector -> IRQ mode
39              b     .                       @ FIQ                   vector -> FIQ mode
40
41 address_rst: .word handler_rst
42 address_irq: .word handler_irq
43
44 table_copy:  mov   r0, #0                  @ set destination address
45              ldr   r1, =table_data         @ set source      address
46              ldr   r2, =table_copy         @ set source      limit
47
48 table_loop:  ldr   r3, [ r1 ], #4          @ load  word, inc. source      address
49              str   r3, [ r0 ], #4          @ store word, inc. destination address
50
51              cmp   r1, r2
52              bne   table_loop              @ loop if address != limit
53
54              mov   pc, lr                  @ return
55
56 /* These function enable and disable IRQ interrupts respectively, by
57  * toggling the 7-th bit of CPSR to either 0 or 1.
58  */
59
60 .global irq_enable
61 .global irq_unable
62
63 irq_enable:  mrs   r0,   cpsr              @  enable IRQ interrupts
64              bic   r0, r0, #0x80
65              msr   cpsr_c, r0
66
67              mov   pc, lr
68
69 irq_unable:  mrs   r0,   cpsr              @ disable IRQ interrupts
70              orr   r0, r0, #0x80
71              msr   cpsr_c, r0
72
73              mov   pc, lr
```

Figure 6: *kernel/interrupt.s*

```
1  #include "kernel.h"
2
3  void kernel_handler_rst() {
4    /* Configure the mechanism for interrupt handling by
5     *
6     * - configuring timer st. it raises a (periodic) interrupt for each
7     *   timer tick,
8     * - configuring GIC st. the selected interrupts are forwarded to the
9     *   processor via the IRQ interrupt signal, then
10    * - enabling IRQ interrupts.
11    */
12
13   TIMER0->Timer1Load     = 0x00100000; // select period = 2^20 ticks ~= 1 sec
14   TIMER0->Timer1Ctrl     = 0x00000002; // select 32-bit   timer
15   TIMER0->Timer1Ctrl    |= 0x00000040; // select periodic timer
16   TIMER0->Timer1Ctrl    |= 0x00000020; // enable          timer interrupt
17   TIMER0->Timer1Ctrl    |= 0x00000080; // enable          timer
18
19   GICC0->PMR             = 0x000000F0; // unmask all            interrupts
20   GICD0->ISENABLER[ 1 ] |= 0x00000010; // enable timer          interrupt
21   GICC0->CTLR            = 0x00000001; // enable GIC interface
22   GICD0->CTLR            = 0x00000001; // enable GIC distributor
23
24   irq_enable();
25
26   return;
27 }
28
29 void kernel_handler_irq() {
30   // Step 2: read  the interrupt identifier so we know the source.
31
32   uint32_t id = GICC0->IAR;
33
34   // Step 4: handle the interrupt, then clear (or reset) the source.
35
36   if( id == GIC_SOURCE_TIMER0 ) {
37     PL011_putc( UART0, 'T' ); TIMER0->Timer1IntClr = 0x01;
38   }
39
40   // Step 5: write the interrupt identifier to signal we're done.
41
42   GICC0->EOIR = id;
43 }
```

Figure 7: *kernel/kernel.c*