- The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. 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: you can contact them, to report then resolve said problem, using details collated at

  https://www.bristol.ac.uk/it-services/contacts

- The worksheet is written *assuming* you work in the lab. using UoB-managed equipment. If, however, you prefer to use your own equipment, *unsupported*[a] alternatives *do* exist: you could a) manually install any software dependencies yourself, *or* b) use the unit-specific Vagrant[b] box by following instructions at

  https://www.github.com/danpage/COMS20001/blob/COMS20001_2019/vagrant/README.md

- We intend the 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 C (for core questions, that *should* be attempted within the lab. slot), A (for additional questions, that *could* be attempted within the lab. slot), or R (for revision questions, that support preparation for any viva and/or exam). Keep in mind that we only *expect* you to attempt 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).

[a] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[b] https://www.vagrantup.com

COMS20001 lab. worksheet #4

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

http://tinyurl.com/y2fxztqv/csdsp/os/sheet/lab-4_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[C]. 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 worksheet #3 (which depended on each process cooperatively invoking the `yield` system call) so it supports pre-emptive multi-tasking and thus *enforces* periodic context switches.

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 #2.

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` are essentially identical to worksheet #2 (bar the specialisation to interrupt types handled), so we omit any discussion of them.

`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 former is identical to, and the latter similar those provided in worksheet #2 (bar the specialisation to interrupt types handled). The difference is the lack of support for supervisor call interrupts. More specifically, only IRQ interrupts are handled; as a result, the `lolevel_handler_svc` interrupt handler is missing (as is the associated entry in the interrupt vector table).

`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 former is identical to, and the latter similar those provided in worksheet #2. As above, the main difference stems from the fact this handles IRQ interrupts only. However, it is also true that both `hilevel_handler_rst` and `hilevel_handler_irq` differ due to the *type* of IRQ interrupts: previously we handled interrupts requested by a UART, whereas now we focus on a timer.

- `hilevel_handler_rst` is invoked by `lolevel_handler_rst` every time a reset interrupt is requested and needs to be handled. Lines #20 to #31, which configure the interrupt handling mechanism, are different: note that a) most obviously the `SP804_t` instance `TIMER0` is first configured st. interrupts are requested every 2^{20} timer ticks, then b) the GIC configuration is largely similar, bar the fact a different source (i.e., #36 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.
- `hilevel_handler_irq` is invoked by `lolevel_handler_irq` every time a IRQ interrupt is requested and needs to be handled. Lines #43 to #45, which represent the main interrupt handling step, are different. We now test whether the timer requested 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.

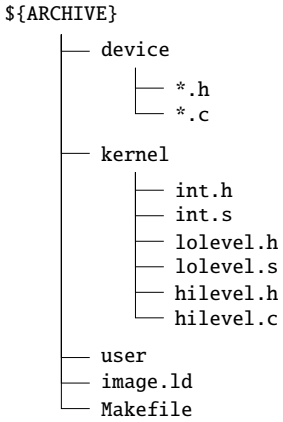

Figure 1: *A diagrammatic description of the material in* `lab-4_q.tar.gz`.

```
8  SECTIONS {
9    /* assign load address (per  QEMU) */
10   .       =     0x70010000;
11   /* place text segment(s)           */
12   .text : { kernel/lolevel.o(.text) *(.text .rodata) }
13   /* place data segment(s)           */
14   .data : {                         *(.data        ) }
15   /* place bss  segment(s)           */
16   .bss  : {                         *(.bss         ) }
17   /* align       address (per AAPCS) */
18   .       = ALIGN( 8 );
19   /* allocate stack for irq mode     */
20   .       = . + 0x00001000;
21   tos_irq = .;
22 }
```

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

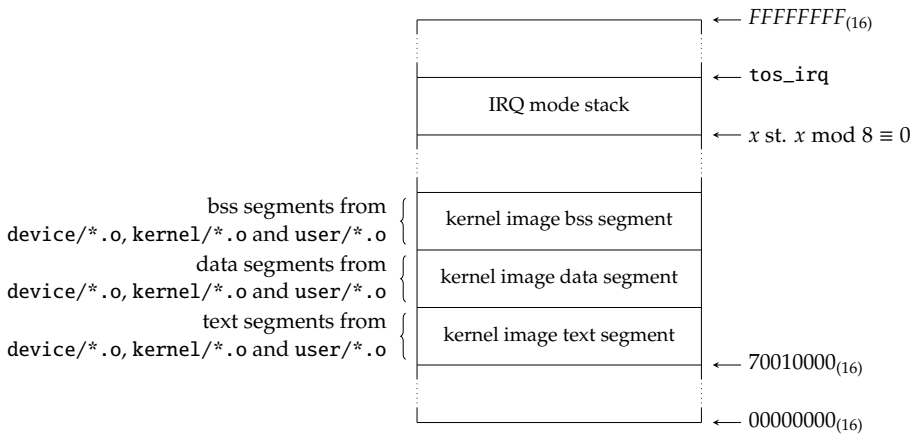

Figure 3: *A diagrammatic description of the memory layout realised by* `image.ld`.

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

Recall the kernel image provided with worksheet #2 *also* wrote a sequence of 'T' characters to the emulation terminal. However, it is important to keep in mind a subtle difference. Those characters were previously 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 is different here, because the timer operates concurrently with the processor. As a result, the processor will now be *inactive* between IRQ interrupts: although we are not capitalising on this fact, 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 execution of a user process after some period (*rather* than it needing to invoke a `yield` system call). As such, the obvious next step lso acts as a direct step towards the coursework assignment.

Worksheet #3 already provided a simple kernel image that supports 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 will no longer be 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).

Q2[A]. As outlined in worksheet #1, the build system provided will link your implementation with newlib: this is a limited implementation of the full C standard library, ideal for embedded platforms. Understanding the set of functionality newlib supports, and how to make use of it, will become more attractive as the software you write for the PB-A8 increases in complexity. For example, you might want to start dynamically allocating memory using `malloc`, or producing more involved output via `sprintf`. Although both are supported by newlib, making use of them is a challenge. One on hand, you can gain a lot of insight by investigating this challenge yourself; one *could* argue that doing so forms part of the coursework assignment. On the other hand, however, it can be enormously frustrating if you just want to use `sprintf`.

As such, this question, which is really more of a guided overview akin to worksheet #1, is a compromise. The focus is on the former[1] case above: how can we use newlib to support bare-metal programs which use `malloc` to dynamically allocate memory. It provides a ready-made solution for this case, but also attempts to explain newlib in enough detail that you can address other, similar cases yourself.

Q2–§5 Understanding the problem

Triggering the error A sensible first question to ask is what happens when we *try* to use `malloc`. Take the material provided for this worksheet, and

- add the line

  ```
  #include <stdlib.h>
  ```

  to `hilevel.h`, then
- add the line

  ```
  uint8_t* x = ( uint8_t* )( malloc( 10 ) );
  ```

  to `hilevel.c`, e.g., in `hilevel_handler_rst`.

Compiling the result *should* now produce an error message something like "undefined reference to `end`". This is basically the same situation as when you use (i.e., reference) a variable but forget to define it (or at least it is not in scope).

[1] One motivation for this choice is that it actually solves *both* cases above: it turns out that `sprintf` etc. use `malloc`, so if *you* use `sprintf` then you basically get the same problem and the same solution works.

Locating the cause The next question to ask is what uses `end`, and where is that? You might intuitively guess that it must be `malloc`, because we get the error only when invocation of that function is added: to confirm this we therefore need to look at the implementation of `malloc`. The browsable git repository for newlib at

http://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git

makes this easier:

- The first step is to find `malloc`: the (short) definition is in the file

  http://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=blob;f=newlib/libc/stdlib/malloc.c

  and just invokes `_malloc_r`.

- It turns out that `_malloc_r` is defined in

  http://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=blob;f=newlib/libc/stdlib/mallocr.c

  but is reasonably hard to locate. Searching for `_malloc_r` shows it is used in a pre-processor definition, st. `mALLOc` will be translated into `_malloc_r`. Further down the file, the function `mALLOc` is defined, which, after applying the pre-processor instead defines `_malloc_r`: this is what we are interested in.

- Point #6 of the documentation hints at the next step: a macro `MORECORE`, normally defined as `sbrk`, will be used by `mALLOc` to extend the amount of memory allocated for the heap. So the next question is, where is `sbrk` defined? It appears in

  http://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=blob;f=newlib/libc/syscalls/syssbrk.c

  but just invokes `_sbrk_r`, which is, in turn, defined in

  http://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=blob;f=newlib/libc/reent/sbrkr.c

  and, again in turn, invokes `_sbrk`. At this point, you can be forgiven for cursing the naming scheme used for these functions! But assuming you still have the motivation, this is the final step: where is `_sbrk` defined?

This is where the trail ends. The reason is that newlib is actually split into *two* parts: newlib contains the actual functionality, i.e., the C standard library libc, whereas *another* library, libgloss, contains sets of support functionality for a specific platforms (e.g., the PB-A8, or ARM-based platforms more generally). This support functionality includes replacement functions for non-existent system calls on platforms that lack a kernel. In particular,

http://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=blob;f=libgloss/arm/syscalls.c

includes the definition for `_sbrk` we were looking for. Finally, the problem is clear: this function needs the linker to defined the symbol `end`, i.e., *this* is where the error is coming from. There are a couple of things worth pointing out about this implementation:

- We could have saved ourselves some work by reading over the newlib documentation for libc

  http://www.sourceware.org/newlib/libc.html

  relating to system calls: although the `sbrk` implementation shown there is an example, it may have already given us a hint as to the cause.

- The function definition includes

  ```
  __attribute__((weak))
  ```

  which may be unfamiliar. Attributes are a mechanism to communicate information from the programmer to compiler, but which fall outside the remit of the C language itself. The attributes GCC understands wrt. functions can be found at

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

  In this case, the function is marked as being weak, which, in short, means it can be overridden by another function definition (with the same identifier). This is a useful concept within the context of a library: the library can provide a default implementation, but also the program linked against it can replace this with an alternative if need be.

Q2–§6 Implementing a solution

After all the analysis above, the solution is fortunately very simple: we just need to instruct the linker to provide the missing symbol. Add the following to `image.ld`

```
.heap : {
        end    = .;
  _heap_start = .;
  .            = . + 0x00001000;
  _heap_end    = .;
}
```

between the directives for the bss segment and the stack segment(s), so *after* the line including `.bss` and *before* the line including `ALIGN( 8 )`. This instructs the linker to a) allocate a region (in this case 4KiB) of memory, b) define the missing symbols required by the newlib `malloc` st. that region can be managed as a heap; recompilation using this updated linker script completes with no error.

As an aside, note that an alternative solution would be to capitalise on the definition of `sbrk` as a weak function by overriding it with our *own* implementation. This is certainly involves more work, but, equally, it allows more control over how the allocated region is managed.

Q2–§7 Next steps

Now we can *compile* the program, but of course that does not mean it will (necessarily) *work*! To gain some confidence that it will, and *how* it will work, it makes sense to explore how the newly equipped `malloc` behaves:

a Execute[2] the existing example (from above): by using the debugger to step through the functions invoked, can you reproduce the analysis above (i.e., show the sequence identified by hand is *actually* the one executed)? Inspect the pointer `x`. Can you think of a way to reason whether or not this is sane, i.e., (roughly) what you expect?

b Develop a more complicated example, which mixes invocations of both `malloc` (for various sized allocations) and `free`. If you again inspect the pointers returned by `malloc`, do they make sense? Can you show that `malloc` will reuse a previously deallocated region?

c 4KiB is not *that* large a region; you could easily imagine this being exhausted. Develop an example that does so intentionally. How can you check for this error condition?

d Various parameters in the newlib implementation of `malloc` will have been selected at compile-time. One example is the way `malloc` deals with alignment, i.e., at what boundaries it will guarantee each allocation align with. Develop an example that tests what this parameter is set to.

[2] There is one minor complication. As is, GCC will probably optimise the example by applying dead code elimination: put simply, it spots `x` is unused so when we debug the program it appears as if the variable is not there. There are various possible solutions, one of which is to add an artificial use of `x`, e.g., add an assignment `x[ 0 ] = 123;`, after it is initialised.