COMS20001 lab. worksheet #3

- 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-3_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[L]. This question develops a simple yet functioning operating system kernel, which is based on the concept of cooperative multi-tasking. The goal is to provide a concrete introduction to representation of processes, and, crucially, realisation of a basic context switching mechanism. Doing so requires support for system calls including `yield`, that allows a process to cooperatively yield control of the processor.

Although the worksheet concludes with a set of challenges relating to experimental exploration, the more fundamental goal is to gain a solid 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 later challenges.

Explore the archive content

Normally, the kernel would be totally separate from any given user program: the idea is the kernel can dynamically retrieve a given program image from a storage medium (e.g., a disk), then use it to populate the address space (e.g., the text segment) of a user process to capture the state of execution. However, doing so is complex[1] in general and certainly *too* complex to consider at this stage. We therefore make some significant simplifications: throughout, we will

a assume the entire system (i.e., the kernel *plus* any user programs) is statically compiled to form a single kernel image,

b assume each such user program is executed exactly once (by the kernel, so automatically) and never terminates,

c restrict the form of the user programs executed so they cannot make use of static or global[2] variables.

[1] Beyond the fact a storage medium must actually exist in the first place, note that the kernel would have to include components such as a) a file system, supported by an device driver, and b) a loader rich enough to understand appropriate executable and/or object file formats.

[2] A technical justification for this in fact stems from the first simplification above. Consider a given user program; a static or global variable defined in said program will be located in the single data or bss segment in memory (depending whether it is initialised or

This implies a fairly limited remit for the kernel we will consider: it really only needs to a) include an appropriate process table that captures the state of execution (for each executed user program), and b) implement a context switch mechanism and scheduling algorithm so said processes can be suspended and resumed as need be.
As Figure 1 illustrates, the content and structure of the archived material provided matches worksheet #2. The only difference is some additional files within the (previously empty) `user` directory: in line with the above, these are

- `P1.[ch]` and `P2.[ch]`, which are definitions of user programs statically compiled into the kernel image, and
- `libc.[ch]`, which represents a (very limited) library against which the user programs are linked: it acts as a layer of abstraction that hides low-level interaction with the kernel (i.e., the detail of system calls) via two high-level, C wrapper functions.

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 latter is subtly, but *significantly* more complex than similar files in previous worksheets. This complexity stems from the fact that when an interrupt occurs, we now expect a *user* mode process to be executing. As a result, the execution context must be a) preserved at the interrupt handler entry point, and b) restored at the interrupt handler exit point. These differences are limited to the top part, and so the two functions outlined below:

- `lolevel_handler_svc` can be viewed as three parts:
  - a 5-instruction prologue,
  - invocation of `hilevel_handler_svc`, the high-level, C interrupt handler defined in `hilevel.c`, and
  - a 5-instruction epilogue.

  To understand how this works, imagine the processor is executing a user mode process (in USR mode) and executes an `svc` instruction. This requests a supervisor call interrupt, and causes `lolevel_handler_svc` to be invoked (with the processor now in SVC mode). The kernel must avoid corrupting the execution context associated with the user process, because otherwise would be impossible to resume (i.e., recommence execution of) the user process later. To see how a lack of care *could* cause problems, note that only certain registers are banked: the SVC mode `r0` register holds a value "owned" by USR mode, for example. So, it preserves said context on the SVC mode stack:

  - Line #32 corrects the return address,
  - Line #33 allocates space on the SVC mode stack for the execution context.
  - Line #34 preserves USR mode `r0` through to `r12`, plus `sp` (i.e., `r13`) and `lr` (i.e., `r14`).
  - Lines #35 and #36 preserve USR mode CPSR (which was copied into SVC mode SPSR by the interrupt), plus USR mode `pc` (which was copied into SVC mode `lr` by the interrupt).

  Figure 4 illustrates the stack content[3] at each step. With the USR mode execution context now preserved on the SVC mode stack, `hilevel_handler_svc` is invoked:

uninitialised). Now imagine the user program is executed twice, yielding two independent processes. Each of the processes is executing the same user program, so access to the static or global variable will conflict: there is only one such variable, rather than one per process. Note that the analogous problem is easier to resolve with local variables, since they reside on the stack.

[3] Although not specifically important here, it still makes sense to explain two subtleties wrt. preservation of the USR mode execution context. First, notice that we first decrement the stack pointer then store the USR mode registers in *ascending* order; you might ask why not omit the first step, storing them in descending order instead. The reason for this choice is that when `ctx->gpr` is used as an array, elements with a larger index are expected to have a higher address: `ctx->gpr[ 1 ]` should, for example, be at an address 4B higher than `ctx->gpr[ 0 ]`. The approach used satisfies precisely this requirement. Second, notice that we treat `r11` as a general-purpose register; this assumes it is not being used as a frame pointer, which, per worksheet #1, we guaranteed by invoking GCC with the `-fomit-frame-pointer` flag. Within the coursework assignment this becomes important when implementing `fork`: when creating a new process the kernel has to initialise the stack pointer, but if the frame pointer is disabled then it be ignored (meaning the task of implementing `fork` is simpler than if the frame pointer were enabled).

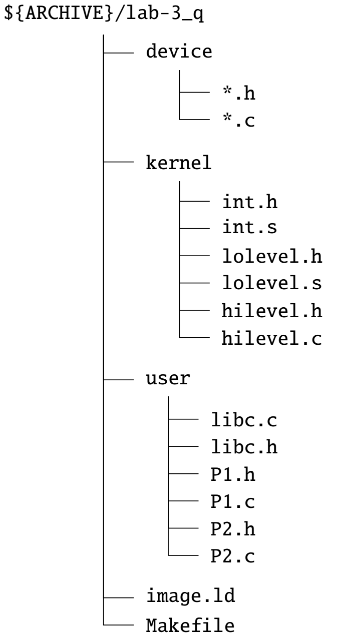


Figure 1: *A diagrammatic description of the material in* `lab-3_q.tar.gz`*.*

```
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 P1           */
  .        = . + 0x00001000;
  tos_P1  = .;
  /* allocate stack for P2           */
  .        = . + 0x00001000;
  tos_P2  = .;
}
```

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

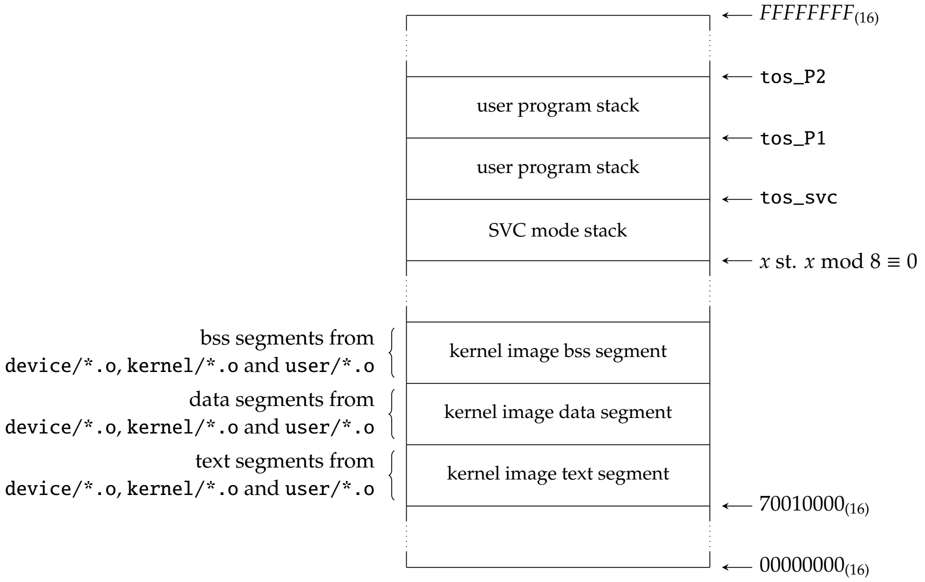
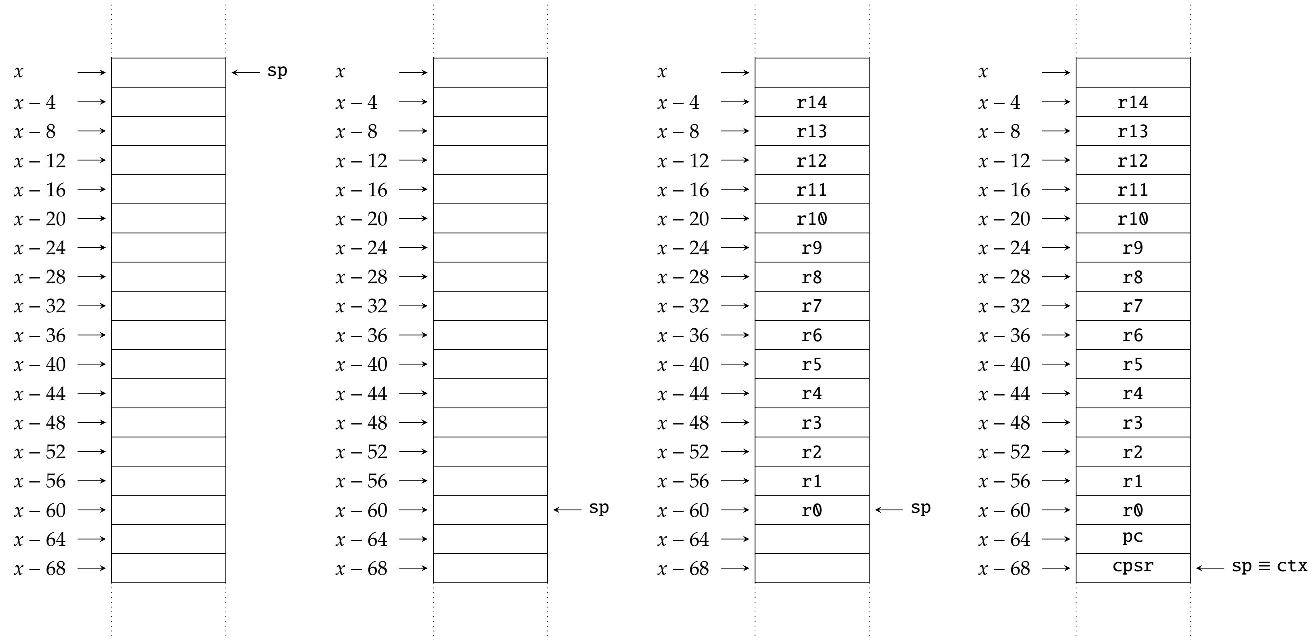

FFFFFFFF(16)
tos_P2
user program stack
tos_P1
user program stack
tos_svc
SVC mode stack
x st. x mod 8 ≡ 0
bss segments from device/*.o, kernel/*.o and user/*.o
kernel image bss segment
data segments from device/*.o, kernel/*.o and user/*.o
kernel image data segment
text segments from device/*.o, kernel/*.o and user/*.o
kernel image text segment
70010000(16)
00000000(16)

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



(a) *Stack layout at function entry point (i.e., Line #32).* **(b)** *Stack layout after* `sub` *(i.e., Line #33).* **(c)** *Stack layout after* `stmia` *(i.e., Line #34).* **(d)** *Stack layout after* `stmdb` *(i.e., Line #36).*

Figure 4: *An example showing how* `lolevel_handler_svc` *preserves the USR mode state on the SVC mode stack (Lines #33 to #36) ready for use by* `hilevel_handler_svc` *and then subsequent restoration (Lines #43 to #46). Note that the eventual value of* `sp` *is passed to the high-level interrupt handler* `hilevel_handler_svc` *as* `ctx`*: this allows it to manipulate the execution context preserved on the stack.*

- Line #38 sets argument #0 equal to the SVC mode `sp`.
- Lines #39 and #40 set argument #1 equal to the 24-bit immediate value encoded in the svc instruction that requested the interrupt now being handled.
- Line #41 invokes `hilevel_handler_svc` with the arguments as specified above.

Since argument #0 is effectively a pointer to the preserved execution context, `hilevel_handler_svc` can use and/or update it as appropriate. However, in doing so it must consider one fact: whatever the preserved execution context contains once the function returns will always be restored by the epilogue in `lolevel_handler_svc`:

- Lines #43 and #44 restore USR mode CPSR into SVC mode SPSR, plus USR mode `pc` into SVC mode `lr`.
- Line #45 restores USR mode `r0` through to `r12`, plus `sp` (i.e., `r13`) and `lr` (i.e., `r14`).
- Line #46 deallocates space on the SVC mode stack for the execution context.
- Line #47 returns control at the now restored program counter value held in SVC mode `lr`; note that simultaneously, SVC mode SPSR is copied into USR mode CPSR which is what invokes the processor mode switch.

- `lolevel_handler_rst` is a lot easier to explain:
  - As with worksheet #2, Lines #16 to #19 initialise the interrupt vector table and SVC mode stack.
  - Lines #26 to #30 match the epilogue in `lolevel_handler_svc`, so restore an execution context from the SVC mode stack once `hilevel_handler_rst` returns.
  - Unlike `lolevel_handler_svc`, however, there is nothing sane to preserve because this interrupt relates to initialisation after reset: there can never be a currently executing user process at that point! As a result, Line #21 allocates the right amount of space on the SVC mode stack *for* an execution context, but does not fill it with anything.
  - Lines #23 and #24 invoke `hilevel_handler_rst`, with `sp` as an argument so the function can update the execution context: it *must* do so, otherwise the restoration steps in the epilogue will essentially just corrupt the register content.

It is vital to remember this implementation represents a trade-off in favour of simplicity, but at the expense of efficiency. For example, preserving the *entire* execution context on *every* interrupt allows for a clean and uniform separation between low- and high-level interrupt handlers, *but* could be inefficient if/when the same execution context is restored as preserved: in this case, we may have avoided the overhead of the former (and could do so, more easily at least, by writing more of the interrupt handler at a low-level where control over use of registers is easier and more explicit).

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`. When compared to previous worksheets, the former has some additional content which needs explanation:

- Line #42 define a type `pid_t` that captures an integer Process IDentifier (PID).
- Lines #44 to #50 define a type `status_t` that captures the status of a process, including, for example, whether it is currently executing.
- Lines #52 to #54 define a type `ctx_t` that captures an execution context (i.e., the processor state). In addition to noting that each pertinent component is present, keep in mind that `ctx_t` uses a specific order for the components: reading the content from `sp` upward, it matches Figure 4 wrt. elements of said content. So if we have a pointer `ctx` of type `ctx_t*` equal to `sp`, we can access the preserved execution context using (high-level) field in `ctx` (rather than via an unstructured, low-level offset from `sp`).
- Lines #56 to #60 define a type `pcb_t` that captures a Process Control Block (PCB), instances of which form entries in the process table: given the limited remit here, each such entry includes only a PID and execution context.

These type definitions are used by the latter, which implements the kernel itself in what can be viewed as two parts. The top part implements some support functionality:

- Line #19 defines a fixed-size array of `pcb_t` instances for use as the process table: there is one entry for each user process, with an integer index into this table maintained so it is clear which PCB is active (i.e., which process is currently executing).
- Lines #21 to #38 implement the function `scheduler`, i.e., the scheduler algorithm. We know there will always be two processes, so the implementation is (very) special-purpose: it just a) uses `index` to determine which process is currently executing, b) preserves the execution context of that process by invoking `memcpy` to copy `ctx` into the associated PCB, c) restores the execution context of the only other process by invoking `memcpy` to copy the associated PCB into `ctx`, d) updates `index` to reflect the currently executing process.

The bottom part implements two high-level interrupt handler functions, which use the support functionality outlined above:

- `hilevel_handler_rst` is invoked by `lolevel_handler_rst` every time a reset interrupt is requested and needs to be handled.
  - Lines #54 to #59 and Lines #61 to #66 initialises the process table by copying information into two PCBs, one for each user program; in each case the PCB is first zero'ed by `memset`, and then components such as the program counter value are initialised (to the address of the entry point, e.g., `main_P1`). In essence, these steps reflect (automatic) execution of the two user programs, and hence formation of two associated user processes.
  - Lines #72 to #74 then select a PCB entry (in this case the 0-th entry), and activate it. Note that the specific value[4] used to initialise CPSR means each process will execute in USR mode, with IRQ interrupts enabled. As such, there is *no* explicit call to `int_enable_irq` (as *was* required in worksheet #2 for example).
- `hilevel_handler_svc` is invoked by `lolevel_handler_svc` every time a reset interrupt is requested and needs to be handled. Recall that it is passed two arguments, namely
  - a pointer to the preserved execution context (on the SVC mode stack, i.e., the value of `sp`), and
  - the immediate operand of the `svc` instruction that requested the interrupt being handled: this is used as the system call identifier.

[4] Using the literal $50_{(16)}$ implies that $\mathsf{CPSR}[M] = 10000_{(2)}$, $\mathsf{CPSR}[F] = 1_{(2)}$, and $\mathsf{CPSR}[I] = 0_{(2)}$, i.e., USR mode, with FIQ interrupts disabled and IRQ interrupts enabled.

The function uses the system call identifier to decide how the interrupt should be handled, and so what the kernel-facing side of the system call API should do:

- Lines #89 to #92 deal with identifier $00_{(16)}$, i.e., the `yield` system call: when one of the user processes invokes `yield` as defined in `libc.c`, the resulting supervisor call interrupt is eventually handled by this case. The semantics of this system call are that the process scheduler should be invoked, i.e., that the currently executing user process has yielded control of the processor (thus permitting the other to execute).
- Lines #94 to #105 deal with identifier $01_{(16)}$, i.e., the `write` system call: when one of the user processes invokes `write` as defined in `libc.c`, the resulting supervisor call interrupt is eventually handled by this case. The semantics of this system call are that some `n`-element sequence of bytes pointed to by `x` are written to file descriptor `fd`; the kernel ignores `fd`, and simply writes those bytes to the `PL011_t` instance `UART0`.

Note that not *all* potential system calls are handled by this kernel, but adding additional cases is simply a matter of following the same approach as the above.

`libc.[ch]`: the user mode library The header file `libc.h` and source code `libc.c`. capture the user-facing side of the system call API. Put another way, we described the *kernel*-facing side (i.e., *implementation*) above; `libc.h` and `libc.c` support use of that implementation, providing a more appropriate level of abstraction by wrapping raw, low-level `svc` instructions in high(er)-level functions. These wrappers abstract (at least) two details:

- To specifically execute an `svc` instruction, we need to use assembly language: the C language has no native support for such low-level concepts. To do so, as hinted in worksheet #1, we harness inline assembly language to combine C and assembly language (in particular, variables in the former can be used in the latter). Resources such as

  http://wiki.osdev.org/Inline_Assembly

  provide an introduction, but, briefly, you can consider each function as being

  - a top part which is an assembly language template (e.g., with place-holders such as `%0` akin to those used in `printf`) "filled in" by the compiler using the stated inputs and outputs,
  - a (potentially empty) list of outputs referring to C variables,
  - a (potentially empty) list of inputs, referring to C variables (or constants),
  - a (potentially empty) list of registers that are "clobbered" (or changed) within the template; the rationale for this list, at a high level, is to communicate any hidden impact of the template to the compiler (e.g., a change to some register is would otherwise *assume* remains the same after as before the template).

- The user-facing side of the system call API must adhear to the calling convention. Various choices exist: here we opt to use the `svc` instruction itself to communicate a system call identifier, with arguments and return values communicated via the first four general-purpose registers (much like a function call).

`P[12].[ch]`: the user mode programs The header file `P1.h` and source code `P1.c` capture user program #1, and similarly for `P2.h` and `P2.c`. The purpose (and thus form) of both user programs is intentioanlly simple, which is evident by using `P1.c` as an example:

- Lines #11 to #13 implement an infinite `while` loop, which implies that the function never returns. Each iteration makes two system calls, namely

  - a call to `write` that instructs the kernel to write a string to the file descriptor `STDOUT_FILENO`; per the above, this is interpreted as transmitting it via the `PL011_t` instance `UART0` as a form of standard output stream, and
  - a call to `yield` that instructs the kernel to invoke the scheduler: per the above this means a context switch occurs.

- Line #15 performs an `exit` system call. Although the kernel has no associated implementation yet, it does so for completeness: a normal program[5] would call `exit` to terminate the associated process, although the while loop described above means control-flow never reaches *this* call.

[5] From the programmers perspective, the entry point for a normal C program is the `main` function. However, `main` is called from and returns to a wrapper called `_start` (or similar): although it *looks* like `main` lacks a call to `exit`, this is actually performed implicitly by `_start`.

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 the two strings "P1" and "P2" being written alternately to the emulation terminal. At an intuitive level, the behaviour of this kernel is easy to summarise. It should be clear that when the processor is reset, each of the two user programs are (automatically) executed and form associated processes. Over time, the processor is shared between said processes; this is achieved by the kernel switching between them when the process currently executing (cooperatively) relinquishes control via the `yield` system call. Each process does so, using the `write` system call to identify itself beforehand (so as to *demonstrate* it is executing), meaning the result is a sequence of identifier strings written to the emulator terminal as observed.

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 three context switches at most.

b Add an *additional*, third user program: initially this could be more or less identical to the other two, but the goal would be to explore questions like

  i what happens if it behaves in a semi-cooperative (i.e., does invoke `yield`, just less often than the other two user programs) or non-cooperative (i.e., does not invoke `yield`) manner, or

  ii whether/how the new user program can exploit the the lack of protection implemented by this kernel, and what behaviour might result when it does so.

c Add an *additional* system call by updating both to the user- and kernel-facing side of the system call API. A good candidate is `read`, since this acts as the natural counterpart to `write` by allowing processes to read bytes from the emulated UART.

d As outlined above, there are various possible choices wrt. design of the system call API: investigate how system call invocation is implemented in Linux, then alter the implementation used by this kernel to match. Having done so, what can you say about potential advantages or disadvantages of the different design choices?

Q2[A]. Consider this quote

> One process wrote A, the other wrote B, so I saw AAAA, BBBB, and so on [1].

This *should* sound somewhat familiar. Who said it? Linus Torvalds. What was the context? Early development of Linux. So, put another way, from the previous question you have a starting point *more or less* equivalent to a *very* early, prototypical Linux!

The history of Linux is interesting, and there are numerous sources that offer an overview; the Wired interview [1] from which the quote above was taken is one, early example. Absorbing this history is arguably time well spent, whether you do so in depth or not:

- lessons learned from how *it* developed *can* be applied in your own work (in this, and other units), and, perhaps more importantly,
- it should offer motivation: although *obviously* challenging for a variety of reasons, you can take stock after the first 3 weeks of the unit and be confident the outcome is already of *real* value and relevance.

Q3[A]. There is a fantastic short film at

http://www.youtube.com/watch?v=Q07PhW5sCEk

focusing on the Compatible Time-Sharing System (CTTS), developed at MIT in the early 1960s. CTTS was *highly* influential in terms of later technology: a range of subsequent systems such as Multics and UNIX refined concepts and approaches *it* pioneered. However, the film is arguably more remarkable because a) it offers such a brilliant explanation of what, at the time, was a revolutionary concept, and b) gives an insight into the challenges faced by computing in that era: it neatly illustrates the cost associated with maintaining one computer, for example, and so the value in maximising utilisation of it!

References

[1] G. Moody. *The Greatest OS That (N)ever Was*. http://www.wired.com/1997/08/linux-5. 1997 (see p. 7).