COMS20001 lab. worksheet: week #15

- 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-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]Execute the command `tar xvfz lab-3_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 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, and, in particular, a `yield` system call 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 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; the only difference is some additional files within the (previously empty) `user` directory which demand explanation.

Normally, the kernel would be totally separate from any given user program: the idea is the kernel can (dynamically) retrieve the program image from some storage medium (e.g., a disk), then use it to populate the address space (e.g., the text segment) of a process that captures the state of execution. However, doing so is complex; there must actually be a storage medium to start with, for example, and even then the kernel must implement components such as a device driver and file system. Such demands are *too* complex at this stage, so we make two fairly significant simplifications: we

- assume the whole system (i.e., the kernel *plus* any user programs) is (statically) compiled to form a single kernel image, *and*
- assume each such user program is executed once and once only, and never terminates: this means we opt for some special- vs. general-purpose mechanisms, e.g., by pre-allocating some of the resources (e.g., PCBs, stack space) each process requires.

In support of both, the additional files are

```
${ARCHIVE}/lab-3_q
 ├── device
 │    ├── GIC.h
 │    ├── GIC.c
 │    ├── PL011.h
 │    ├── PL011.c
 │    ├── SP804.h
 │    └── SP804.c
 ├── kernel
 │    ├── interrupt.h
 │    ├── interrupt.s
 │    ├── kernel.h
 │    └── kernel.c
 ├── user
 │    ├── libc.c
 │    ├── libc.h
 │    ├── P0.h
 │    ├── P0.c
 │    ├── P1.h
 │    └── P1.c
 ├── image.ld
 └── Makefile
```

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

- `P0.[ch]` and `P1.[ch]`, which are definitions of the two user programs (statically) compiled into the kernel image, and
- `libc.[ch]`, which represents a (very limited) standard library against which the programs are linked: the role of this library is similar to the (full) C standard library, in the sense it abstracts interaction with the kernel via two system call wrappers.

Based on the above, the kernel itself has a fairly limited remit: it needs to a) include an appropriate process table that captures the state of execution (for each executed user program), and b) implement a context switching mechanism and scheduling algorithm so said processes can be suspended and resumed as need be.

Q1–§2 Understand the archive content

`image.ld` Figure 4 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 5 and Figure 7 illustrate the header file `interrupt.h` and source code `interrupt.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:

- `handler_svc` can be viewed as three parts:
  - a 5-instruction prologue,
  - invocation of `kernel_handler_svc`, the high-level handler in `kernel.c`, and
  - a 5-instruction epilogue.

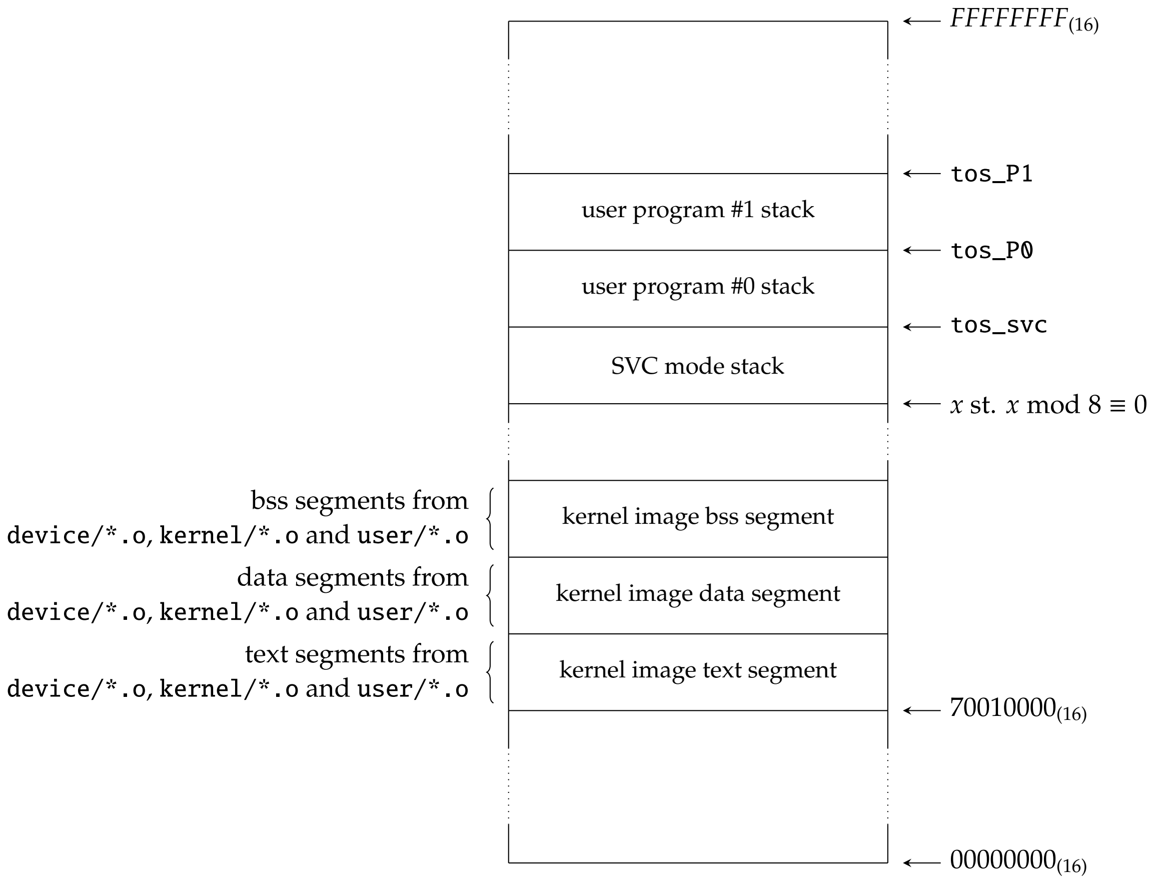

FFFFFFFF(16)
tos_P1
user program #1 stack
tos_P0
user program #0 stack
tos_svc
SVC mode stack
x st. x mod 8 ≡ 0
bss segments from device/*.o, kernel/*.o and user/*.o
kernel image bss segment
data segments from device/*.o, kernel/*.o and user/*.o
kernel image data segment
text segments from device/*.o, kernel/*.o and user/*.o
kernel image text segment
70010000(16)
00000000(16)

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

To understand how this works, imagine the processor is executing a user mode process (in USR mode) and executes an `svc` instruction. This raises a supervisor call interrupt, and causes the interrupt handling mechanism to invoke `handler_svc` (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 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:

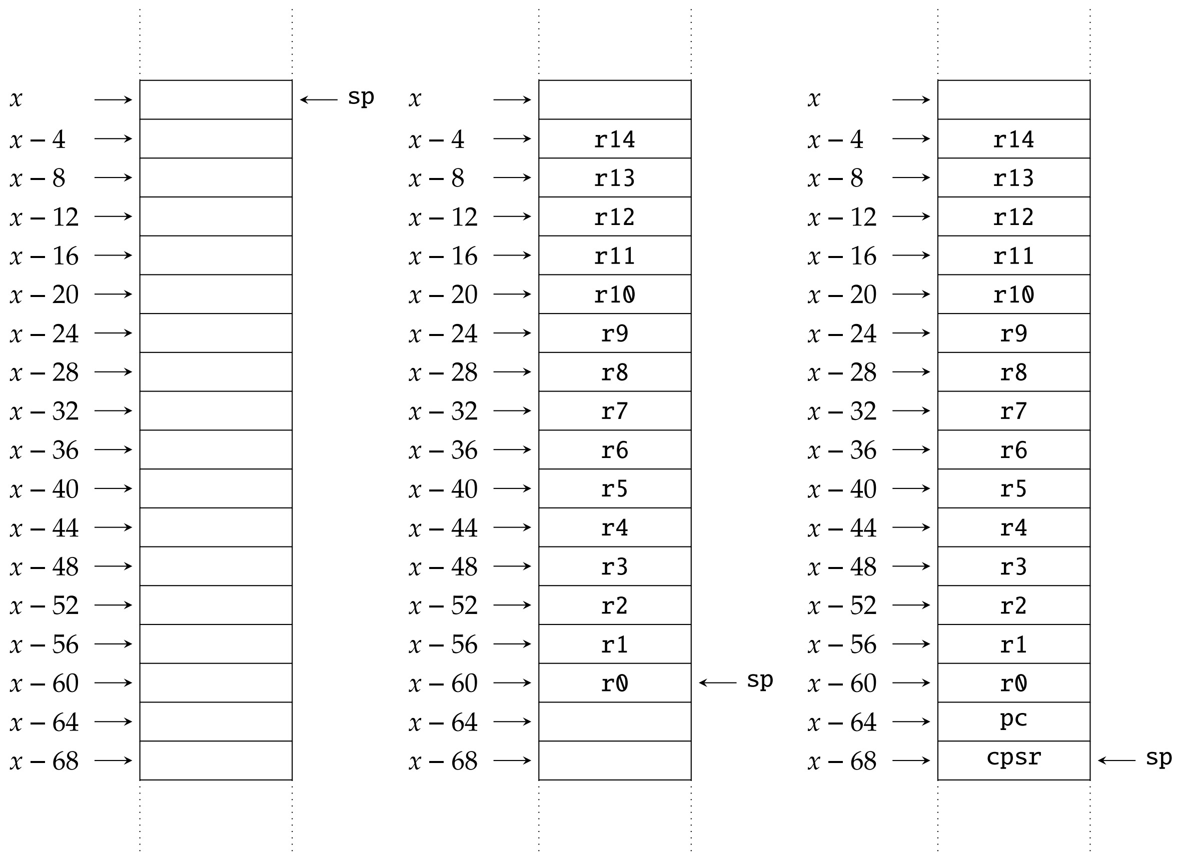
- Line #23 allocates space on the SVC mode stack for the execution context.
- Line #24 preserve USR mode `r0` through to `r12`, plus `sp` (i.e., `r13`) and `lr` (i.e., `r14`).
- Lines #25 and #26 preserve USR mode CPSR (which was copied into SVC mode SPSR by the interrupt), plus USR mode `sp` (i.e., `r13`, which was copied into SVC mode `lr` by the interrupt).

Figure 3 illustrates the stack content at each step in this process. So, with the USR mode execution context preserved on the SVC mode stack, `kernel_handler_svc` is invoked:

- Line #28 set argument #0 equal to the SVC mode `sp`.
- Line #29 and #30 set argument #1 equal to the 24-bit immediate value encoded in the `svc` instruction that raised the interrupt now being handled.
- Line #31 invokes `kernel_handler_svc` with the arguments as specified above.

Since argument #0 is effectively a pointer to the preserved execution context, `kernel_handler_svc` can use/update this as it sees fit. However, it must consider one fact, namely that whatever the preserved execution context contains when the function returns will be restored by the `handler_svc` epilogue:

- Lines #33 and #34 restore USR mode CPSR into SVC mode SPSR, plus USR mode `sp` into SVC mode `lr`.
- Line #35 restore USR mode `r0` through to `r12`, plus `sp` (i.e., `r13`) and `lr` (i.e., `r14`).
- Line #36 deallocates space on the SVC mode stack for the execution context.



(a) *Stack layout at function entry point (i.e., line #27).* **(b)** *Stack layout after* `sub` *and* `stmia` *(lines #28 and #29).* **(c)** *Stack layout after* `stmdb` *(line #31), capturing the preserved USR mode state.*

Figure 3: *An example showing how* `handler_svc` *preserves (lines #28 to #31) the USR mode state on the SVC mode stack ready for use by* `kernel_handler_svc` *and then subsequent restoration (lines #38 to #41).*

  - Line #37 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.

- `handler_rst` is a lot easier to explain:

  - As with the worksheet from week #14, lines # 6 to #9 initialise the interrupt vector table and SVC mode stack.
  - Lines #16 and #20 are identical to the prologue in `handler_svc`: they restore an execution context from the SVC mode stack once `kernel_handler_rst` returns.
  - Unlike `handler_svc`, however, there is nothing sane to preserve: this interrupt relates to initialisation after reset, so there can be no currently executing user process. As a result, line #11 simply allocates the right amount of space on the SVC mode stack *for* an execution context, but does not fill it with anything.
  - Lines #13 and #14 invoke `kernel_handler_rst`, with `sp` as an argument so the function can update the execution context: it *must* do so, otherwise the restoration steps will result in "junk" being written into all the registers.

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 done 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).

kernel.[ch] Figure 6 and Figure 8 illustrate the header file `kernel.h` and source code `kernel.c`. When compared to previous worksheets, the former has some additional content which needs explanation:

- Lines #31 to #33 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 3 st. elements of said content. So if we have pointer `ctx` of type `ctx_t*` equal to `sp`, we can access values of the preserved execution context via fields in `ctx`.

- Line #35 define a type `pid_t` that captures an integer Process IDentifier (PID),

- Lines #37 to #40 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, such entry simply includes a PID and execution context.

These type definitions are used by the former, which implements the kernel itself:

- Line #12 defines a fixed-size array of `pcb_t` instances for use as the process table: there is one entry for each user process, with a pointer into this table maintained so it is clear which PCB is active (i.e., which process is currently executing).

- Lines #14 to #25 implement the function `scheduler`, which is the scheduler algorithm. We know there will always be two processes, so the implementation is (very) special-purpose: it simply checks which process is active, and performs a context switch to suspend it and resume the other. You could think of this as round-robin scheduling, but with two processes clearly there is only one choice! `memcpy` is used to copy the associated execution contexts into place, before updating `current` to reflect the (newly) active PCB.

- Line #27 to #55 implement the high-level interrupt handler `kernel_handler_rst`. First it 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. In short, these steps reflect (automatic) execution of the two user programs and hence formation of two associated user processes. With the process table initialised, the function then selects a PCB entry and activates it: `memcpy` is used to copy the execution context to addresses pointed at by `ctx` (as allocated by `handler_rst`).

  Note `handler_rst` will restore the execution context, and resume execution once `kernel_handler_rst` returns: the initialised value of CPSR has a subtle but important role. More specifically, we the (preserved) CPSR value is initialised so it reflects the processor in USR mode with IRQ interrupts enabled. By doing so, when the preserved execution context is restored, the processor is automatically switched to USR mode and IRQ interrupts enabled: there is no need to perform either explicitly, as there is in `kernel_handler_rst` say.

- Lines #57 to #86 implement the high-level interrupt handler `kernel_handler_svc`: recall this is invoked by `handler_svc` with the two arguments formed as

  - 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 raised the interrupt being handled: this is used as the system call identifier.

  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 #67 to #70 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 #71 to #82 Identifier $01_{(16)}$ is 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`.

`libc.[ch]` Figure 13 and Figure 14 illustrate the header file `libc.h` and source code `libc.c`. The idea here here is to capture the user-facing side of the system call API. Put another way, the *kernel*-facing side (i.e., the *implementation*) is described above; `libc.h` and `libc.c` support use of that implementation, offering a more appropriate abstraction by wrapping low-level `svc` instructions in high(er)-level functions. These wrappers abstract both the fact assembly language is required, *and* the actual calling convention used. Various choices exist: here we opt to use the `svc` instruction itself to communicate a system call identifier. However, we could equally have defined the API so this is communicated via a register in the same way as any arguments.

`P[01].[ch]` Figure 9 and Figure 10 illustrate the header file `P0.h` and source code `P0.c`, which describe user program #0; the files associated with user program #1 are essentially the same, so we focus on the former alone as an example.

- Line #4 defines a 20-character string `x` user to identify the program (this is the only difference in user program #1).
- Lines #6 to #8 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 `x` to the file descriptor whose identifier is 0: per the above, this is interpreted as transmitting it via the `PL011_t` instance `UART0`, and
  - a call to `yield` that instructs the kernel to invoke the scheduler: per the above this means a context switch occurs.

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 the two strings "hello world, I'm P0" and "hello world, I'm P1" being written 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.

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 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 non-cooperative (i.e., does not invoke `yield`) or semi-cooperative (i.e., does invoke `yield`, just less often than the other two user programs) 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 this design to match.

```
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 svc mode */
  .       = . + 0x00001000;
  tos_svc = .;
  /* allocate stack for P0       */
  .       = . + 0x00001000;
  tos_P0  = .;
  /* allocate stack for P1       */
  .       = . + 0x00001000;
  tos_P1  = .;
}
```

Figure 4: *image.ld*

```
#ifndef __INTERRUPT_H
#define __INTERRUPT_H

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

#endif
```

Figure 5: *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"

// Include functionality from newlib, the embedded standard C library.

#include <string.h>

// Include definitions relating to the 2 user programs.

#include "P0.h"
#include "P1.h"

/* The kernel source code is made simpler by three type definitions:
 *
 * - a type that captures each component of an execution context (i.e.,
 *   processor state) in a compatible order wrt. the low-level handler
     preservation and restoration prologue and epilogue,
 * - a type that captures a process identifier, which is basically an
 *   integer, and
 * - a type that captures a process PCB.
 */

typedef struct {
  uint32_t cpsr, pc, gpr[ 13 ], sp, lr;
} ctx_t;

typedef int pid_t;

typedef struct {
  pid_t pid;
  ctx_t ctx;
} pcb_t;

#endif
```

Figure 6: *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, #0xD3             @ enter SVC mode with no interrupts
             ldr   sp, =tos_svc            @ initialise SVC mode stack

             sub   sp, sp, #68             @ initialise dummy context

             mov   r0, sp                  @ set    C function arg. = SP
             bl    kernel_handler_rst      @ invoke C function

             ldmia sp!, { r0, lr }         @ load   USR mode PC and CPSR
             msr   spsr, r0                @ set    USR mode        CPSR
             ldmia sp, { r0-r12, sp, lr }^ @ load   USR mode registers
             add   sp, sp, #60             @ update SVC mode SP
             movs  pc, lr                  @ return from interrupt

handler_svc: sub   lr, lr, #0              @ correct return address
             sub   sp, sp, #60             @ update SVC mode stack
             stmia sp, { r0-r12, sp, lr }^ @ store  USR registers
             mrs   r0, spsr                @ get    USR        CPSR
             stmdb sp!, { r0, lr }         @ store  USR PC and CPSR

             mov   r0, sp                  @ set    C function arg. = SP
             ldr   r1, [ lr, #-4 ]         @ load                 svc instruction
             bic   r1, r1, #0xFF000000     @ set    C function arg. = svc immediate
             bl    kernel_handler_svc      @ invoke C function

             ldmia sp!, { r0, lr }         @ load   USR mode PC and CPSR
             msr   spsr, r0                @ set    USR mode        CPSR
             ldmia sp, { r0-r12, sp, lr }^ @ load   USR mode registers
             add   sp, sp, #60             @ update SVC mode SP
             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
             b     .                       @ IRQ                   vector -> IRQ mode
             b     .                       @ FIQ                   vector -> FIQ mode

address_rst: .word handler_rst
address_svc: .word handler_svc

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 7: *kernel/interrupt.s*

```c
#include "kernel.h"

/* Since we *know* there will be 2 processes, stemming from the 2 user
 * programs, we can
 *
 * - allocate a fixed-size process table (of PCBs), and use a pointer
 *   to keep track of which entry is currently executing, and
 * - employ a fixed-case of round-robin scheduling: no more processes
 *   can be created, and neither is able to complete.
 */

pcb_t pcb[ 2 ], *current = NULL;

void scheduler( ctx_t* ctx ) {
  if      ( current == &pcb[ 0 ] ) {
    memcpy( &pcb[ 0 ].ctx, ctx, sizeof( ctx_t ) );
    memcpy( ctx, &pcb[ 1 ].ctx, sizeof( ctx_t ) );
    current = &pcb[ 1 ];
  }
  else if ( current == &pcb[ 1 ] ) {
    memcpy( &pcb[ 1 ].ctx, ctx, sizeof( ctx_t ) );
    memcpy( ctx, &pcb[ 0 ].ctx, sizeof( ctx_t ) );
    current = &pcb[ 0 ];
  }
}

void kernel_handler_rst( ctx_t* ctx              ) {
  /* Initialise PCBs representing processes stemming from execution of
   * the two user programs.  Note in each case that
   *
   * - the CPSR value of 0x50 means the processor is switched into USR
   *   mode, with IRQ interrupts enabled, and
   * - the PC and SP values matche the entry point and top of stack.
   */

  memset( &pcb[ 0 ], 0, sizeof( pcb_t ) );
  pcb[ 0 ].pid      = 0;
  pcb[ 0 ].ctx.cpsr = 0x50;
  pcb[ 0 ].ctx.pc   = ( uint32_t )( entry_P0 );
  pcb[ 0 ].ctx.sp   = ( uint32_t )(  &tos_P0 );

  memset( &pcb[ 1 ], 0, sizeof( pcb_t ) );
  pcb[ 1 ].pid      = 1;
  pcb[ 1 ].ctx.cpsr = 0x50;
  pcb[ 1 ].ctx.pc   = ( uint32_t )( entry_P1 );
  pcb[ 1 ].ctx.sp   = ( uint32_t )(  &tos_P1 );

  /* Once the PCBs are initialised, we (arbitrarily) select one to be
   * restored (i.e., executed) when the function then returns.
   */

  current = &pcb[ 0 ]; memcpy( ctx, &current->ctx, sizeof( ctx_t ) );

  return;
}

void kernel_handler_svc( ctx_t* ctx, uint32_t id ) {
  /* Based on the identified encoded as an immediate operand in the
   * instruction,
   *
   * - read  the arguments from preserved usr mode registers,
   * - perform whatever is appropriate for this system call,
   * - write any return value back to preserved usr mode registers.
   */

  switch( id ) {
    case 0x00 : { // yield()
      scheduler( ctx );
      break;
    }
    case 0x01 : { // write( fd, x, n )
      int   fd = ( int   )( ctx->gpr[ 0 ] );
      char*  x = ( char* )( ctx->gpr[ 1 ] );
      int    n = ( int   )( ctx->gpr[ 2 ] );

      for( int i = 0; i < n; i++ ) {
        PL011_putc( UART0, *x++ );
      }

      ctx->gpr[ 0 ] = n;
      break;
    }
    default   : { // unknown
      break;
    }
  }

  return;
}
```

Figure 8: *kernel/kernel.c*

```c
#ifndef __P0_H
#define __P0_H

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

#include "libc.h"

// define symbols for P0 entry point and top of stack
extern void (*entry_P0)();
extern uint32_t tos_P0;

#endif
```

Figure 9: *user/P0.h*

```c
#include "P0.h"

void P0() {
  char* x = "hello world, I'm P0\n";

  while( 1 ) {
    write( 0, x, 20 ); yield();
  }
}

void (*entry_P0)() = &P0;
```

Figure 10: *user/P0.c*

```c
#ifndef __P1_H
#define __P1_H

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

#include "libc.h"

// define symbols for P1 entry point and top of stack
extern void (*entry_P1)();
extern uint32_t tos_P1;

#endif
```

Figure 11: *user/P1.h*

```c
#include "P1.h"

void P1() {
  char* x = "hello world, I'm P1\n";

  while( 1 ) {
    write( 0, x, 20 ); yield();
  }
}

void (*entry_P1)() = &P1;
```

Figure 12: *user/P1.c*

```c
#ifndef __LIBC_H
#define __LIBC_H

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

// cooperatively yield control of processor, i.e., invoke the scheduler
void yield();

// write n bytes from x to the file descriptor fd
int write( int fd, void* x, size_t n );

#endif
```

Figure 13: *user/libc.h*

Q2[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 refining 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!

```c
#include "libc.h"

void yield() {
  asm volatile( "svc #0     \n"  );
}

int write( int fd, void* x, size_t n ) {
  int r;

  asm volatile( "mov r0, %1 \n"
                "mov r1, %2 \n"
                "mov r2, %3 \n"
                "svc #1     \n"
                "mov %0, r0 \n"
              : "=r" (r)
              : "r" (fd), "r" (x), "r" (n)
              : "r0", "r1", "r2" );

  return r;
}
```

Figure 14: *user/libc.c*