UNIVERSITY OF BRISTOL
DEPARTMENT OF COMPUTER SCIENCE
http://www.cs.bris.ac.uk

Assessed coursework
Concurrent Computing (COMS20001)

CW2

Note that:

1. **The deadline for this courswork is** 22/04/16**, with standard regulations enforced wrt. late submission.**
2. **This courswork represents** 25 **percent of marks available for COMS20001, and is assessed on an individual basis: make sure you are aware of and adhere to the various regulations**[a] **which govern this mode of assessment.**
3. **There are numerous support resources available, for example:**
   - **the unit forum hosted via**

     http://www.ole.bris.ac.uk

     where lecturers, lab. demonstrators and students all regularly post questions and answers,
   - **the lecturer responsible for this courswork, namely Daniel Page, who is available within stated office hours, by appointment, or via email.**

[a] http://www.bristol.ac.uk/academic-quality/assessment/codeonline.html

1 Introduction

Any sufficiently large program eventually becomes an operating system.

– M. Might (http://matt.might.net/articles/what-cs-majors-should-know/)

This coursework focuses on development of an operating system kernel. The practical nature of this task is important from an educational perspective: it should offer a) deeper understand of various topics covered in theory alone, *and* transferable experience applicable when you either b) develop software whose effectiveness and efficiency depend on detailed understanding how it interacts with hardware and/or a kernel, or c) develop software for a platform that lacks a kernel yet still requires run-time support of some kind (that *you* must therefore offer). The constituent stages offer a variety of options, in attempt to cater for differing levels of interest in the topic as a whole. In particular, note that they offer an initially low barrier to entry for what is obviously a challenging task when considered as a whole.

2 Terms and conditions

- This assignment is intended to help you learn something; where there is some debate about the right approach, the assignment demands that *you* make an informed decision *yourself*. This should be based on a reasonable argument formed via *your own* background research (rather than reliance on the teaching material alone).
- The coursework design includes two heavily supported, closed initial stages which reflect a lower mark, and one totally unsupported, open final stage which reflects a higher mark. This suggests the marking scale is non-linear: it is clearly easier to obtain X marks in the initial stages than in the final stage.

  The terms open and closed should be read as meaning flexibility wrt. options *for* work, *not* open-endedness wrt. workload. For example, even though each option in the later stage is potentially open-ended it has clear success criteria at which point you can (and should) stop.
- The assignment description may refer to `marksheet.txt`. Download this ASCII text file from

  http://www.ole.bris.ac.uk/bbcswebdav/courses/COMS20001_2015/csdsp/os/cw/CW2/marksheet.txt

  then complete and include it in your submission: this is important, and failure to do so may result in a loss of marks.
- Implementations produced as part of the assignment are marked using a platform equivalent to the CS lab. (MVB-2.11). As a result, they *must* compile, execute and be thoroughly tested against default operating system and development tool-chain versions available.
- You should submit your work via the SAFE submission system at

  http://wwwa.fen.bris.ac.uk/COMS20001/

  including all source code, written solutions and any auxiliary files you think are important (e.g., any example input or output).
- Your submission will be assessed in a 20min viva held in week #23 or #24. By the stated submission deadline, select a viva slot online via

  http://doodle.com/poll/ymy3rup8gvvrvp4n

  to suit your schedule. Note that:

  – The viva will be based on your submission via SAFE: you will need to know your candidate number so this can be downloaded.
  – The discussion will focus on demonstration and explanation of your solution wrt. the stated success criteria. Keeping this in mind, it is *essential* you have a simple, clear way to execute and demonstrate your work. Ideally, you will be able to a) use one (or very few) command(s) to build a kernel image (e.g., using a script of `Makefile`), then b) demonstrate that a given success criteria has been met by discussing appropriate diagnostic output (e.g., via the emulator terminal).
  – Immediate personal feedback will be offered verbally, with a general, written marking report disseminated some time later.

3 Material

Download and unarchive the file

http://www.ole.bris.ac.uk/bbcswebdav/courses/COMS20001_2015/csdsp/os/cw/CW2/question.tar.gz

somewhere secure within your file system (e.g., in the `Private` sub-directory in your home directory). The content *and* structure of this archive, as illustrated in Figure 1, should be familiar: it closely matches that used by the lab. worksheets, and thus represents a skeleton starting point for your submission.

4 Description

The overarching goal of this assignment is to develop an initially simple but then increasingly capable operating system kernel. It should execute and thus manage resources on a specific ARM-based target platform, namely a RealView Platform Baseboard for Cortex-A8 [2] emulated by QEMU[1].

Stage 1. This stage is designed to match the task in the lab. worksheet from week #16: you should implement a simple, functioning operating system kernel that supports pre-emptive multi-tasking. All similar simplifications made there can also be made here. In particular, note that at this point you can and should assume a) each user program is statically compiled into the kernel image, and b) there is no need to deal with dynamic process creation or termination.

Success criteria. Demonstrate concurrent execution of all the provided user programs (i.e., those in the `user` directory) under control of the kernel. Note that ideally, the programs would be updated so as to provide suitable output: each produce a sequence of integers step-by-step, so writing those integers (plus some form of process identifier) to the emulator terminal should help demonstrate their progress.

Stage 2. This stage involves the design and implementation of some standard improvements to the kernel.

(a) The kernel developed in the lab. worksheet(s) assumed two user processes exist at all times (once executed). Improve on this by supporting dynamic creation and termination of processes via `fork` and `exit` system calls. Since you design the semantics of these system calls, any reasoned simplifications are allowed provided they support the desired functionality: `fork` need not replicate POSIX [1, Page 881] for example!

Success criteria. Demonstrate a command-line shell[2] (however simple) that allows the user to enter commands and thus interactively control program execution (i.e., process creation and termination); in UNIX parlance, this interaction is often termed job control[3].

(b) The kernel developed in the lab. worksheet(s) used a special-purpose scheduling algorithm: it could do so as the result of the assumption about two user processes existing at all times. Improve on this by implementing an alternative; you can select or design an algorithm, but it must be able to capitalise on the concept of priorities somehow.

Success criteria. Demonstrate the differing behaviour of your implementation vs. round-robin scheduling (as implemented in the same kernel), plus explain when and why this represents an improvement.

(c) The kernel developed in the lab. worksheet(s) lacked any mechanism for Inter-Process Communication (IPC). The first half of the unit introduced various ways to support the concept of IPC: implement one of them in the kernel.

Success criteria. Consider a problem from first half of the unit that motivated study of IPC (e.g., dining philosophers), and demonstrate a a solution for it using your implemented mechanism.

Stage 3. This stage has several diverse options, each representing a more ambitious improvement to the kernel. The marking scheme clearly indicates that *one* option should be submitted; there is nothing to prevent you attempting and submitting *more* than one, but only the best mark among those you submit will count.

(a) An alternative to considering one of the various real, albeit emulated devices supported by the PB-A8 is a compromise: for certain cases we could consider a simplified device instead, and therefore (legitimately) focus on higher-level use rather than the device itself.

[1] http://www.qemu.org
[2] http://en.wikipedia.org/wiki/Shell_(computing)
[3] http://en.wikipedia.org/wiki/Job_control_(Unix)

```
${ARCHIVE}/question
├── device
│   ├── disk.h
│   ├── disk.c
│   ├── disk.py
│   ├── GIC.h
│   ├── GIC.c
│   ├── PL011.h
│   ├── PL011.c
│   ├── SP804.h
│   └── SP804.c
├── kernel
│   ├── interrupt.h
│   ├── interrupt.s
│   ├── kernel.h
│   └── kernel.c
├── user
│   ├── libc.h
│   ├── libc.c
│   ├── P0.h
│   ├── P0.c
│   ├── P1.h
│   ├── P1.c
│   ├── P2.h
│   └── P2.c
├── image.ld
└── Makefile
```

Figure 1: *A diagrammatic description of the material in* `question.tar.gz`.

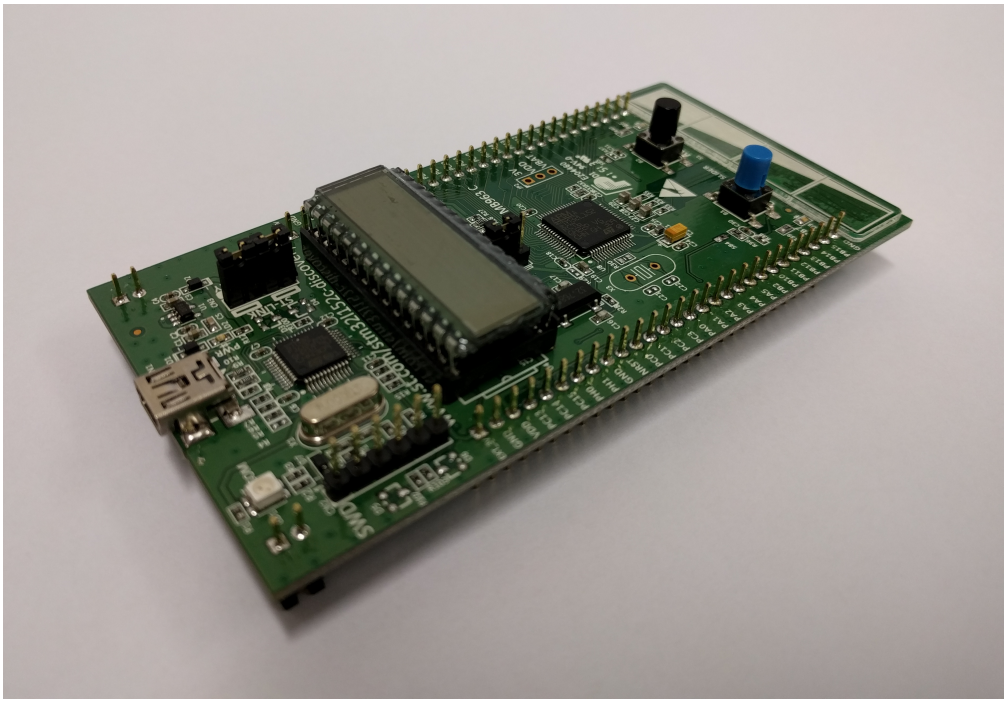


Figure 2: *A STM32L1 target platform.*

Appendix A outlines the source code provided in order to support such a case. The goal is to use a simplified disk, which offers block-based storage of data, to implement a file system: ideally this will a) implement a UNIX-like, inode-based data structure, and b) support a suite of system calls such as `creat`, `unlink`, `open`, `close`, `read` and `write`, with semantics of your own design, which, in turn, demand management of file descriptors.

Success criteria. Demonstrate either a) an example user program that reads and writes data to the disk in a non-trivial way, and/or b) the kernel dynamically loading a user program from the disk then executing it (vs. using one of the statically compiled user programs, as assumed above).

(b) We have a stock of Cortex-M3 based STM32L1 Discovery[4] development boards. Although Figure 2 at least hints the board is simpler than the PB-A8, there *are* two complicating factors wrt. using it: a) the Cortex-M3 processor has a somewhat different interrupt handling mechanism than the Cortex-A8, and *only* supports the Thumb instruction set, and b) support for these boards in the lab. is limited so, realistically, you will need to install the development tools[5] *yourself*. Provided you are willing to accept these challenges, the goal is to port your existing kernel and have it execute on this physical target platform. *Iff.* you are interested in doing so, you will need to contact the lecturer responsible for this coursework in order to reserve and collect a board.

Success criteria. Demonstrate the kernel executing on the physical board, and ideally utilising some of the board-specific devices (e.g., LCD screen) available. Given the challenges outlined above, demonstration using a pre-programmed board in the viva would be fine (alongside explanation of associated source code).

References

[1] Standard for information technology - portable operating system interface (POSIX). Institute of Electrical and Electronics Engineers (IEEE) 1003.1, 2008. http://standards.ieee.org/findstds/standard/1003.1-2008.html.

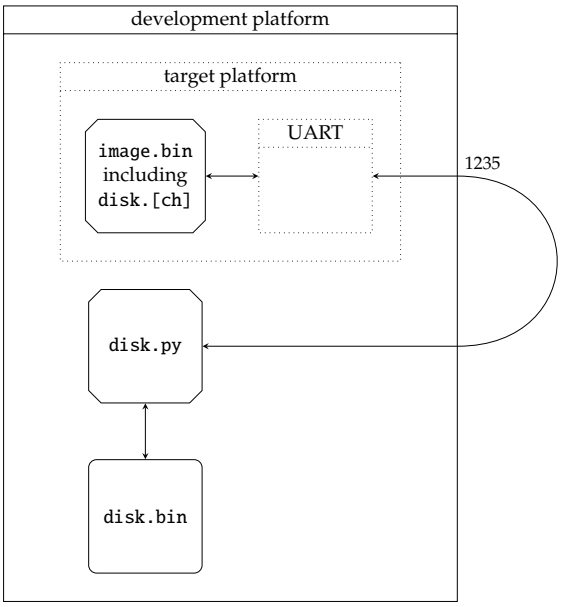
[2] ARM Limited. RealView Platform Baseboard for Cortex-A8. Technical Report HBI-0178, 2011. http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html.

[4] http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1295?sc=stm32l1

[5] On Window see http://www.coocox.org/, which offers a complete, packaged IDE and tool-chain; on Linux you need to work harder, but http://www.github.com/texane/stlink offers the programming and debugging tools to supplement an ARM tool-chain available, for example, via http://launchpad.net/gcc-arm-embedded.

A A simplified disk: `disk.py` and `disk.[ch]`

The concept is illustrated by the diagram below:



The lower part of the diagram illustrates components that exists on the development platform:

- `disk.bin` models the disk mechanism: it stores the disk content as a sequence of bytes as a so-called disk image. This is a standard file, in so far it is stored on the standard file system available via the development platform, so can be inspected and manipulated *independently* from any other component in the diagram.
- `disk.py` models the disk controller: it acts as an interface, accepting high-level commands that it satisfies at a low level via the underlying disk mechanism. When executed, the controller will connect to and communicates via a network port. Note it imposes structure on the disk content, so rather than a "flat" sequence of bytes it is interpreted as some number of blocks of a given length.

The upper part of the diagram illustrates components that exists on the target platform, which in this case is represented by QEMU (and so is, in turn, executing on the development platform):

- We have already capitalised on the fact an emulated UART can be associated with the standard streams of the QEMU process: thus allowed us to read and write input and output via the emulation terminal. It can *also* associate an emulated UART with a network port. This is shown in the diagram, where transmitting or receiving via the UART will be translated into an associated operation wrt. port 1235.
- The kernel image executed by QEMU can interact with the UART in the same way as before, and therefore transmit or receive bytes via port 1235. We capitalise on this by connecting the disk controller to the same port, this allowing the kernel image and disk controller to communicate. To abstract the protocol used in said communication, `disk.[ch]` provides a set of high-level functions the kernel can call; you could view this source code as a primitive form of device driver.

An obvious question is whether or why this concept models reality in any useful sense: if nothing else, it *should* be obvious that a physical PB-A8 would *not* be connected to a disk this way! The answer is then basically that it offers a compromise. That is, what it lacks wrt. realism, it gains in allowing focus on high-level ILOs: the simplification offers a non-volatile, block-based storage medium on top of which you can design and implement a file system without dealing with the low-level detail that *would* exist otherwise.

A.1 Creating a disk image

Creation of an initial, empty disk image is simple. For example, by issuing the command

```
dd of=disk.bin if=/dev/zero count=1048576 bs=1
```

we instruct `dd` to copy 1048576Bs from the input file `/dev/zero` to the output file `disk.bin`. Given that reading from `/dev/zero` will always produce a sequence of zero bytes (i.e., whose value is $00_{(16)}$), this will create a 1MiB file `disk.bin` that is *entirely* zero bytes. Note that:

- Clearly the total capacity should equal the product of whatever block count and length you opt for. For example, we *could* opt for more, shorter blocks st. $65536 \cdot 16\text{B} = 1\text{MiB}$ or for fewer, longer blocks st. $256 \cdot 4096\text{B} = 1\text{MiB}$ to suit: both result in the same capacity, but imply that the controller will interpret the underlying sequence of bytes in a different way.
- You can inspect the byte-by-byte file content using the command

  ```
  hexdump -C disk.bin
  ```

  Initially, however, it will produce somewhat limited output: the repeated zero bytes are printed in a "compressed" form, rather than in their entirety.

A.2 Interacting with the disk interface

A.2.1 The communication protocol

The communication protocol used by the disk controller is, by design, very simple: it receives a command then transmits a response where

- each command and response is 1-line of ASCII text terminated by an EOL character,
- each such line is comprised of some number of fields separated by space characters,
- each field is a sequence of bytes, represented in hexadecimal; the bytes are presented so when read left-to-right, the 0-th byte (resp. $(n-1)$-th byte) is toward the left (resp. right) of the field,
- the first field of each command or response is a 1-byte code which identifies the type (e.g., distinguishes between a write command vs. a read command, or success vs. failure response).

There are only three commands, which are outlined briefly below:

1. A command of the form `00` invokes a query operation: it reports the block count and length of the disk. The response can indicate
   - failure, in which case the response is `01`, or
   - success, in which case the response is of the form `00 <data>` where the data field captures the number of blocks and their length: for example, the response

     ```
     00 00000100 10000000
     ```

     packs together two 32-bit integers, i.e.,

     $$\begin{array}{lcl} \texttt{00000100} & \mapsto & \langle 00_{(16)}, 00_{(16)}, 01_{(16)}, 00_{(16)} \rangle_{(2^8)} = 65536_{(10)} \\ \texttt{10000000} & \mapsto & \langle 10_{(16)}, 00_{(16)}, 00_{(16)}, 00_{(16)} \rangle_{(2^8)} = 16_{(10)} \end{array}$$

     st. there are 65536 blocks, each of length 16B.
2. A command of the form `01 <address> <data>` invokes a write operation: it writes the block of data to the disk at the given block address. For example, the request

   ```
   01 01230000 F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
   ```

   writes the 16-byte block, i.e., the sequence of bytes

   $$\langle F0_{(16)}, F1_{(16)}, F2_{(16)}, F3_{(16)}, F4_{(16)}, F5_{(16)}, F6_{(16)}, F7_{(16)}, F8_{(16)}, F9_{(16)}, FA_{(16)}, FB_{(16)}, FC_{(16)}, FD_{(16)}, FE_{(16)}, FF_{(16)} \rangle$$

   to the disk at the block address

   $$\langle 01_{(16)}, 23_{(16)}, 00_{(16)}, 00_{(16)} \rangle_{(2^8)} = 8961_{(10)}.$$

   The response can indicate
   - failure, in which case the response is `01`, or
   - success, in which case the response is `00`.

3. A request of the form

   ```
   02 <address>
   ```

   invokes a read operation: it reads a block of data from the disk at the given block address. For example, the request

   ```
   02 01230000
   ```

   reads a 16-byte block from the disk at the block address

   $$\langle 01_{(16)}, 23_{(16)}, 00_{(16)}, 00_{(16)} \rangle_{(2^8)} = 8961_{(10)}.$$

   The response can indicate

   - failure, in which case the response is `01`, or
   - success, in which case the response is of the form `00 <data>` where the data field captures the block read: for example, the response

     ```
     00 00112233445566778899AABBCCDDEEFF
     ```

     means the 16-byte block, i.e., the sequence of bytes

     $$\langle 00_{(16)}, 11_{(16)}, 22_{(16)}, 33_{(16)}, 44_{(16)}, 55_{(16)}, 66_{(16)}, 77_{(16)}, 88_{(16)}, 99_{(16)}, AA_{(16)}, BB_{(16)}, CC_{(16)}, DD_{(16)}, EE_{(16)}, FF_{(16)} \rangle$$

     was read.

This protocol obviously has a limited relationship with any analogy such as SCSI[6], and, as a result, is less than ideal in terms of realism. However, a range of advantages *also* stem from this compromise. For example, in combination, the protocol and controller allow a) manual interaction as a means of debugging commands otherwise managed by the device driver (you can actually *type* a command, and then inspect the response), and/or b) development of a file system on top of the disk *independently* from the kernel in the first instance, then integrating it only later. Both are clearly beneficial from a development perspective, aligning with the stated high-level remit.

A.2.2 Manual interaction

1. Issue the command

   ```
   nc -l 127.0.0.1 1235
   ```

   in one terminal.

2. Launch the disk controller using a command such as

   ```
   python disk.py --host=127.0.0.1 --port=1235 --file=disk.bin --block-num=65536 --block-len=16
   ```

   in another terminal.

3. You should be able to type commands into the first terminal, and, provided you adhere to the protocol, get a response from the disk controller.

A.2.3 Programmatic interaction

1. In your kernel implementation, include `disk.h`: this includes prototypes for functions that manage interaction with the disk controller. Note that the device driver *assumes* the disk controller will communicate via the `PL011_t` instance `UART1`.

2. Ensure the line in `Makefile` that reads

   ```
   QEMU_UART += telnet:127.0.0.1:1235,server
   ```

   is uncommented, then launch QEMU as normal in one terminal: this instructs QEMU to associate the `PL011_t` instance `UART1` with the network port 1235.

3. Launch the disk controller using a command such as

   ```
   python disk.py --host=127.0.0.1 --port=1235 --file=disk.bin --block-num=65536 --block-len=16
   ```

   in another terminal.

4. Once you execute the kernel (e.g., issue a `continue` command to `gdb`) the disk interface will respond to commands made via the device driver.

[6] http://en.wikipedia.org/wiki/SCSI