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 05/04/19, 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. Before you start work, 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, 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 assignment 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 understanding 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

- The assignment description may refer to `marksheet.txt`. Download this ASCII text file from

  http://tinyurl.com/ycgk8pce/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.

- The assignment design includes two heavily supported, closed initial stages which reflect a lower mark, and one mostly 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: each stage has clear success criteria that should limit the functionality you implement, meaning you can (and should) stop work once they have been satisfied.

  The stages are intentionally designed and ordered so as to be a) compatible (meaning that a solution for some stage i can co-exist with that for another stage j), and b) cumulative (meaning that meeting the success criteria for some stage i strongly implies meeting it for another, previous stage $i - 1$). The latter fact implies you do not (necessarily) need to maintain a "history" of solutions to demonstrate.

- Perhaps more so than other assignments, this one has a large design space of possible approaches for each stage. Part of the challenge, therefore, is to think about and understand which approach to take: this is not purely a programming exercise st. implementing *an* approach (elsewhere perhaps even prescribed in the description) is enough. Such decisions should ideally be based on a reasoned argument formed via your *own* background research (rather than reliance on the teaching material alone).

- You should submit your solution via the SAFE submission system at

  http://tinyurl.com/y8jqeyrc

  including, if/when relevant, any a) source code files, b) text or PDF files, (e.g., documentation) and c) auxiliary files (e.g., example output) *you* feel are of value.

- Your submission will be assessed in a 20 minute viva (meaning oral exam). By the stated submission deadline, select a viva slot online via

  http://doodle.com/poll/xu8naniz4g7gnrtv

  to suit your schedule. Note that:

  – During the sign-up process it will ask for your name; ideally, you should use the format "full name (UoB user name)" (e.g., "Daniel Page (csdsp)".

  – The location of the viva is room MVB-3.42a, *not* the CS lab. (i.e., *not* MVB-2.11).

  – Your submission will be marked using a platform equivalent to the CS lab. (MVB-2.11). As a result, it *must* compile, execute, and be thoroughly tested against the default operating system and development tool-chain versions available.

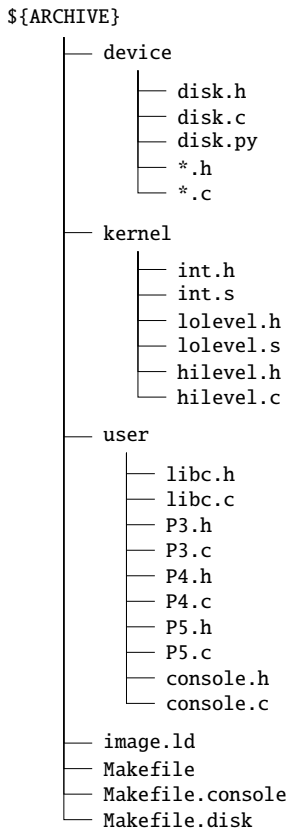


Figure 1: *A diagrammatic description of the material in* `question.tar.gz`.

- The viva will be based on your submission via SAFE: you will need to know your candidate number[1] 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 or `Makefile`), then b) demonstrate that a given success criteria has been met by discussing appropriate diagnostic output. Any significant editing and/or recompilation of the solution during the viva is strongly discouraged, as are multi-part solutions (e.g., use of separately compiled source code for each stage).
- Immediate personal feedback will be offered verbally, with a general, written marking report released at the same time as the marks.

3 Material

Download and unarchive the file

`http://tinyurl.com/ycgk8pce/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.

- Two extra (sub-)`Makefile` are provided: these relate to Appendix A and Appendix B, automating associated commands, and are introduced at the appropriate points below.
- The extra files `disk.[ch]` and `disk.py` also relate to Appendix B, but are only relevant for one option within the final stage.
- In combination, `image.ld`, `lolevel.[sh]` and `int.[sh]` implement a interrupt handling skeleton for the reset, SVC and IRQ interrupts: this is analogous to worksheet #2, with each low-level interrupt handler function invoking an associated high-level, empty placeholder in `hilevel.[ch]`.

[1] This is the 5-digit number you also use in exams, and can be looked-up in SAFE, for example; it is *not* the same as the one on your UCard, nor related to your user name.

- All the user programs provided, namely `console.[ch]` plus `P[345].[ch]`, relate specifically to success criteria in one or more of the stages: they should *not* be altered. If, say, you need to demonstrate or debug functionality in your kernel, then a better approach would be to include *additional* user programs of your own design; you can of course base them on those provided, e.g., embellishing `console.[ch]` to support additional commands.

4 Description

4.1 Context

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 [1] emulated by QEMU[2]. The capabilities of said platform suggest a remit for the kernel, or, equivalently, a motivating context: if and when it makes sense to do so, imagine the platform and kernel form an embedded, consumer electronics product (e.g., set-top box[3] or media center).

4.2 Stages

Stage 1. This stage involves the implementation of a baseline kernel, which acts as a starting point then improved in subsequent stages. It is important to note that each sub-stage is supported directly by the lab. worksheet(s) and lecture slides(s); it would be sensible to complete or recap the associated background material before starting.

(a) The kernel developed in lab. worksheet #3 was based on the concept of cooperative multi-tasking: driven by regular invocation of the `yield` system call, it context switched between and so concurrently executed a fixed number of user processes (that stemmed from statically linked user programs). The task presented in lab. worksheet #4 was to enhance this starting point by supporting pre-emptive multi-tasking instead: complete this task.

Success criteria. Initialise the kernel so the user programs `P3` and `P4` are automatically executed (noting that neither program invokes `yield`), and thus demonstrate their concurrent execution.

(b) The kernel developed in lab. worksheet #3 used a special-purpose scheduling algorithm: it could do so as the result of assuming a fixed number of user processes exist. Improve on this by a) generalising the implementation so it deals with any number of processes (i.e., *any* n, vs. *only* $n = 2$), and b) capitalising on the concept of priorities somehow, using a different scheduling algorithm of your choice.

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

Stage 2. This stage involves the design and implementation of various improvements to the baseline kernel, which, in combination, allow it to support richer and so more useful forms of user program. Each sub-stage is less directly supported, meaning more emphasis on *you* designing then implementing associated solutions.

As a first step, we shift the baseline kernel into a more realistic setting. Initialise the kernel so *only* the `console` user program (see Appendix A) is executed automatically: this will allow you to interact with the kernel via a command-line shell[4], and thus, with appropriate alterations, control each (sub-)stage.

(a) The kernel developed in lab. worksheet #3 assumed a fixed number of user processes exist. Improve on this by supporting dynamic[5] creation and termination of processes via `fork`, `exec`, 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` can be much simpler than in POSIX [2, Page 881], for example.

Success criteria. Altering the provided user programs where appropriate, demonstrate the dynamic creation and termination of some user processes (i.e., correct behaviour of the underlying `fork`, `exec` and `exit` system calls) via appropriate use of the console.

[2] http://www.qemu.org
[3] http://en.wikipedia.org/wiki/Set-top_box
[4] http://en.wikipedia.org/wiki/Shell_(computing)
[5] Note that dynamic process creation does not *imply* a need to use `malloc` etc. for dynamic memory allocation: it is enough to assume some maximum number of processes, allocate a fixed set of resources (e.g., PCBs) to match, then allocate resources for any active processes from that set.

(b) 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. Develop a new user program which uses your IPC mechanism to solve the dining philosophers[6] problem: upon execution, it should first use `fork` to spawn 16 new "philosopher child processes" which then interact with each other via IPC. Demonstrate execution of this new program from the console, and explain how the solution a) ensures mutual exclusion, and b) prevents starvation.

Stage 3. This stage includes several diverse, more challenging options which you can select between. Keep in mind that a) you should only attempt this stage having first completed each previous (sub-)stage, and b) per `marksheet.txt`, you select and submit *one* option only: *if* you submit more, only the option with the highest mark will be considered wrt. assessment.

(a) As shown in worksheet #1, the PB-A8 represents a complete computer system. As such, an ambitious but realistic goal is to investigate various devices *not* utilised thus far. For example, either:

i. **Success criteria.** Demonstrate use of the MMU to a) prevent access by one process into an address space allocated to the kernel or another process, and b) offer some degree of memory virtualisation (i.e., a uniform address space per process).

ii. **Success criteria.** Demonstrate a) management of the LCD and PS/2 controllers within an appropriate device driver framework, and b) implementation of an improved UI vs. interaction via the QEMU terminal and hence command-line shell: this could be achieved either with a user- or kernel-space window manager, for example, however simple.

(b) In contrast to investigating one of the various real, albeit emulated devices supported by the PB-A8, we could consider a compromise: for certain cases we could consider a simplified device instead, and therefore focus on higher-level use rather than low-level detail of the device itself.

Appendix B 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, allowing some form of directory hierarchy, and b) support a suite of system calls such as `open`, `close`, `read` and `write`, with semantics of your own design, which, in turn, demand management of file descriptors.

Success criteria. Demonstrate either a) two new user programs which model the `cat` and `wc` tools (i.e., the ability to write data into a new file, or append to an existing file, then count the lines etc. in it), 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).

(c) Originally, emulation of the PB-A8 was motivated by a need to a) reduce the challenge of software development, and b) address the issue of scale when used in the context of the unit. That said, investigation of a *physical* alternative such the RaspberryPi[7] can be a rewarding exercise. So, provided you are willing to accept the associated and significant challenges, the goal is to port your existing kernel and have it execute on such a platform.

Success criteria. Demonstrate the kernel executing on whatever physical target platform you select, and, ideally, utilising board-specific devices available.

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

[7] Although the specifications differ per model, the RaspberryPi2 represents a good example because it houses a Cortex-A7 close in specification to the Cortex-A8 used by the PB-A8.

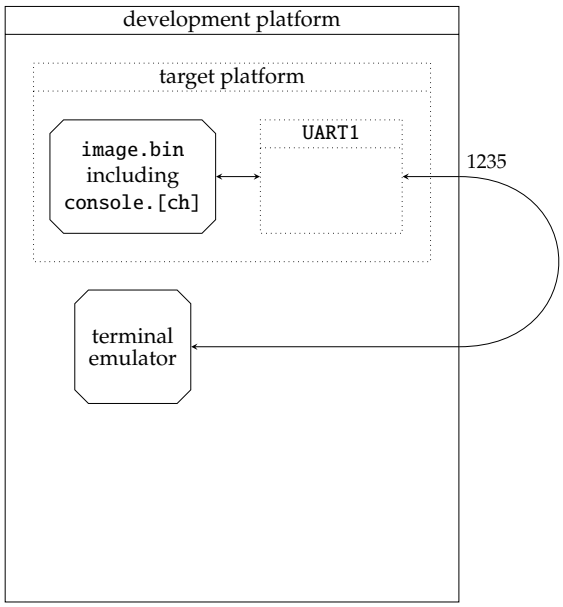
References

[1] *RealView Platform Baseboard for Cortex-A8*. Tech. rep. HBI-0178. ARM Ltd., 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html (see p. 4).

[2] *Standard for Information Technology - Portable Operating System Interface (POSIX)*. Institute of Electrical and Electronics Engineers (IEEE) 1003.1-2008. 2008. URL: http://standards.ieee.org (see p. 4).

A A simplified console: `console.[ch]`

Consider the diagram below:



Although the terms are often conflated, is is common to define a console as a (command-line) terminal specific to local, kernel-mode interaction; this contrasts with a more general terminal, where interaction a) can often be remote (e.g., over a network), and b) is related to user-mode login. The mechanism used to implement a console differs system-by-system, but in embedded contexts, use of a UART is not uncommon: a RaspberryPi, for example, has such a console by default. In fact, the lab. worksheet(s) already made extensive use of this model. QEMU was configured st. an emulated UART (namely the `PL011_t` instance `UART0`) is associated with the emulation terminal: this meant the kernel could read and write input and output, and hence interact with the user. However, QEMU is more flexible than this. It also supports association between an emulated UART and a network port; reading or writing bytes to or from the UART thus implies receiving or transmitting them across the network.

This is essentially what the diagram shows. A user program implementing the console (i.e., `console.[ch]`) executes under control of the kernel, using the `PL011_t` instance `UART1`. This is associated with a network port, allowing a connection from a terminal emulator executing on the development platform: the net result could be viewed as roughly analogous to a command-line terminal resulting from (remote) login to `snowy.cs.bris.ac.uk` using SSH.

A.1 Why is this a (reasonable) simplification?

The approach outlined above is not a radical simplification at all. In fact, the only significant difference vs. a real kernel is direct use of `UART1` by the console implementation: this would obviously need to be managed, by the kernel, via a device driver in reality. The major advantage of this minor simplification is the fact that console I/O can be segregated from I/O by other user programs. This is important, in so far as it offers a cleaner interactive interface with the kernel. Put simply, the alternative of interleaving *all* I/O can be confusing and therefore hinder progress wrt. the core ILOs.

A.2 Interacting with the console

Note that `Makefile.console` extends the vanilla `Makefile` provided with variables and build targets related to use of the console. We explain how to do so directly below, but keep in mind that, as a result, the same steps can and perhaps should be automated.

1. 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. Note that QEMU will initially wait for a connection to be made via said port, indicated by a message similar to

   ```
   QEMU waiting for connection on: disconnected:telnet:127.0.0.1:1235,server
   ```

2. Issue the command

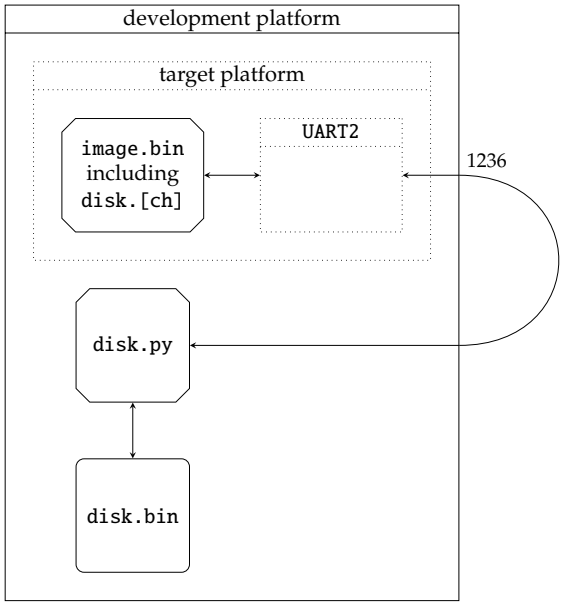
```
nc 127.0.0.1 1235
```

   in another terminal we refer to as the console terminal.

3. Once you execute the kernel (e.g., issue a `continue` command to `gdb`), the console terminal should show a command prompt (assuming the kernel has automatically executed the associated user process): by typing commands into the console terminal, you can interact with the console and hence the kernel.

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

Consider 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, stored on the file system available via the development platform.
- `disk.py` models the disk controller: it acts as an interface, accepting high-level commands that it satisfies at a low level using 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). As was the case in Appendix A, we capitalise on the ability to form an association between emulated UART and network port: by connecting the disk controller to the same port, we allow the kernel image and disk controller to communicate. `disk.[ch]` includes a set of high-level functions the kernel can call; you could view this source code as a primitive form of device driver, which acts as an abstraction of the communication protocol.

B.1 Why is this a (reasonable) simplification?

Hopefully it is obvious that the approach outlined above is not how a real disk would be connected to a real computer system like the PB-A8! Likewise, the communication protocol used has, at best, a limited relationship with a real analogy such as SCSI[8].

As a result, it is fair to question whether the approach represents a reasonable or useful simplification of reality. The answer is basically that it offers a compromise. That is, what it lacks wrt. realism, it gains in allowing a focus on high(er)-level ILOs: with a simplified low-level, block-based storage medium you can focus on high-level design and implementation of a file system without complicating detail that would exist otherwise. In particular, the approach allows a) inspection and manipulation of the disk state using more familiar and "trusted" tools on the development platform, and/or b) manual interaction with the disk controller in order to test and verify behaviour of a command (you can *type* a command, and then inspect the response). Used correctly, both can make the implementation challenge vastly easier.

B.2 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
```

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

we instruct `dd` to copy 1048576B 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:

- Although the use of `dd` *initialises* the disk image via 1048576 blocks each of 1B, the disk controller *using* said image must select a block length of more than 1B: not doing so basically means this is not a block device, and masks many inherent issues.
- Beyond this restriction, the total capacity should equal the product of whatever block count and length you opt for. For example, you *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.

B.3 Interacting with the disk

Note that `Makefile.disk` extends the vanilla `Makefile` provided with variables and build targets related to use of the disk. We explain how to do so directly below, but keep in mind that, as a result, the same steps can and perhaps should be automated. Also note `disk.py` has an optional `--debug` flag, which causes it to emit extra debugging information: this can be useful if/when use of it fails to behave as you expect.

B.3.1 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, and
- 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.,

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

     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 command

   ```
   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 data provided *must* match the block length expected by the controller: data which is too short (i.e., an incomplete block) or too long will result in an error, or even unexpected behaviour. The response can indicate

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

3. A command 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 command

   ```
   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.

B.3.2 Manual interaction

1. Issue the command

   ```
   nc -l 127.0.0.1 1236
   ```

   in one terminal.

2. Launch the disk controller using a command such as

   ```
   python disk.py --host=127.0.0.1 --port=1236 --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.

B.3.3 Programmatic interaction

1. In your kernel implementation, include `disk.h`: this includes declarations 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 `UART2`.

2. Ensure the line in `Makefile` that reads

   ```
   QEMU_UART += telnet:127.0.0.1:1236,server
   ```

   is uncommented, then launch QEMU as normal in one terminal: this instructs QEMU to associate the `PL011_t` instance `UART2` with the network port 1236. Note that QEMU will initially wait for a connection to be made via said port, indicated by a message similar to

   ```
   QEMU waiting for connection on: disconnected:telnet:127.0.0.1:1236,server
   ```

3. Launch the disk controller using a command such as

   ```
   python disk.py --host=127.0.0.1 --port=1236 --file=disk.bin --block-num=65536 --block-len=16
   ```

   at which point the connection QEMU was waiting for is made.

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.