OS3: Assessed Coding Exercise

Sheet 2 (of 2)

This is an assessed exercise, which will count for 12% of the marks in this course.

Attached is a rough outline to guide your work on the exercise. The test harness consists of two main components: a simulated network device and a collection of fake application threads. There are a variety of other useful pieces available, such as a `BoundedBuffer` and the `FreePacketDescriptorStore`, which you may also use. You are to provide an implementation for the network driver, matching the spec in `networkdriver.h`.

The code for this exercise will be posted on Moodle in a few days.

Your Task

- You will download a code archive based on the checksum of your matric number, e.g. `OS3_CW_checksum_0.zip`. In this archive, you will find 10 files that come in 4 variants, and each contains a different variant of a function. Only one of these variants is correct. I suggest you use the `diff` command to see the differences between the files.
- You **should not modify** the source code in these files, but you **will have to add** a single line of code to each of the functions. In other words, the "correct" function is still incomplete, but will only require a single extra line of code, no changes to the given code.
- Once you've identified the correct functions, you should note their numbers and put them in a table in your report, as well as the code you've added.
  Example:

  For example, say you have identified the file `networkdriver_tx_VAR3.c` as the one with the correct function, and added a line

  ```
  int v = 0;
  ```

  to the function `sender_fn` in that file.

  Then in your report you would write:

Selected File	Added Code
networkdriver_tx_VAR3.c	`int v = 0;`

- Once you've identified all correct files, you should edit the Makefile and build the code using `make`.
  You should put the correct file names in the Makefile (i.e. the ones with _VAR), *not* rename the source files. You should ensure that your code compiles and runs correctly.
- Your report should contain a screenshot that shows the output of the make command (e.g. `make -f Makefile.linux`).
- Your report should contain a screenshot that shows the output of the application running for a few seconds.

Background information (taken from the `.h` files)

The test harness application threads are distinguished by their PID:

```
typedef unsigned int PID;
#define MAX_PID 10
/*
```

```
 * A PID is used to distinguish between the different applications
 * It is implemented as an unsigned int, but is guaranteed not to
 * exceed MAX_PID.
 */
```

The network device is "full duplex" – i.e., it can write a packet and read a packet at the same time. Writing a packet entails the calling thread simply invoking a method in the device, with that thread blocking until the packet has been successfully transmitted or until an error is detected. Reading a packet entails making two calls to the device, one to register a packet descriptor for receiving the next packet, and the second to block the calling thread until a packet has been received.

The network device supports the following calls – your code may invoke these:

```
int send_packet (NetworkDevice , PacketDescriptor);
/*
 * Returns 1 if successful, 0 if unsuccessful
 * May take a substantial time to return
 * If unsuccessful you can try again, but if you fail repeatedly give
 * up and just accept that this packet cannot be sent for some reason
 */

void register_receiving_packetdescriptor(NetworkDevice, PacketDescriptor*);
/*
 * tell the network device to use the indicated PacketDescriptor to
 * store the next incoming data packet; once a descriptor is used it
 * shouldn't be reused for a further incoming data packet as it will
 * contain data that is supposed to be delivered to an application;
 * you must therefore register a fresh PacketDescriptor before the
 * next packet arrives
 */

void await_incoming_packet(NetworkDevice);
/*
 * The calling thread blocks until the registered PacketDescriptor has
 * been filled with an incoming data packet. The PID field in the
 * incoming data packet will have been set to indicate the local
 * application process which is the intended recipient of the
 * PacketDescriptor.
 *
 * This should be called as soon as possible after the previous
 * call to wait for a packet returns. Of course, another PacketDescriptor
 * needs to have been registered before this call is reissued.
 * Only 1 thread may be waiting for an incoming packet.
 */
```

The fake application threads may make the following calls to your code – you must implement them.

```
void blocking_send_packet(PacketDescriptor);
int nonblocking_send_packet(PacketDescriptor);
/*
 * These calls hand in a PacketDescriptor for dispatching
 * The nonblocking call must return promptly, indicating whether or
 * not the indicated packet has been accepted by your code
 * (it might not be if your internal buffer is full) 1=OK, O=not OK
 * The blocking call will usually return promptly, but there may be
 * a delay while it waits for space in your buffers.
 * Neither call should delay until the packet is actually sent
 */

void blocking_get_packet(PacketDescriptor*, PID);
int nonblocking_get_packet(PacketDescriptor*, PID);
```

```
/*
 * These represent requests for packets by the application threads
 * The nonblocking call must return promptly, with the result 1 if
 * a packet was found (and the first argument set accordingly) or
 * 0 if no packet was waiting.
 * The blocking call only returns when a packet has been received
 * for the indicated process, and the first arg points at it.
 * Both calls indicate their process number and should only be
 * given appropriate packets. You may use a small bounded buffer
 * to hold packets that haven't yet been collected by a process,
 * but are also allowed to discard extra packets if at least one
 * is waiting uncollected for the same PID. i.e. applications must
 * collect their packets reasonably promptly, or risk packet loss.
 */
```

There are also some initialization calls, the first is for you to implement, the second is one you can use:

```
void init_network_driver( NetworkDevice nd, void * mem_start,
      unsigned long mem_length, FreePacketDescriptorStore * fpds_ptr);
/*
 * Called before any other methods, to allow you to initialize
 * data structures and start any internal threads.
 * Arguments:
 *    nd: the NetworkDevice that you must drive,
 *    mem_start, mem_length: some memory for PacketDescriptors
 *    fpds_ptr: You hand back a FreePacketDescriptorStore into
 *    which PacketDescriptors built from the memory
 *    described in args 2 & 3 have been put
 */

void init_packet_descriptor(PacketDescriptor *);
/*
 * Resets the packet descriptor to be empty.
 * Should be used before registering a descriptor
 * with the NetworkDevice.
 */
```

You will need to use the `FreePacketDescriptorStore` facilities. You can create a `FreePacketDescriptorStore` with:

```
FreePacketDescriptorStore create_fpds(void);
```

populate it with `PacketDescriptors` created from a memory range using:

```
void create_free_packet_descriptors(FreePacketDescriptorStore fpds,
                            void *mem_start, unsigned long mem_length);
```

and then acquire and release `PacketDescriptors` using:

```
void blocking_get_pd(FreePacketDescriptorStore, PacketDescriptor *);
int nonblocking_get_pd(FreePacketDescriptorStore, PacketDescriptor *);

void blocking_put_pd(FreePacketDescriptorStore, PacketDescriptor);
int nonblocking_put_pd(FreePacketDescriptorStore, PacketDescriptor);

/*
 * As usual, the blocking versions only return when they succeed.
 * The nonblocking versions return 1 if they worked, 0 otherwise.
 * The _get_ functions set their final arg if they succeed.
 */
```

Finally, given a `PacketDescriptor` you can access the embedded PID field using:

```
PID packet_descriptor_get_pid(PacketDescriptor *);
```

What to Hand-in

Everything is to be handed in online.

In Moodle, you upload a complete zip or tgz archive (other formats are **not** acceptable) containing

1. A report in PDF format that provides the following information:
    - A maximum of one paragraph indicating how well your solution matches the spec and how complete it is (partly coded?, compiled?, debugged? etc). Also indicate whether the buffers etc were your own code; if not you must state where you acquired them.
    - A table with the correct versions and the added code as described above
2. **All** source files and Makefiles required to build and test the code.

You must also complete an online declaration of originality as follows:

1. Go to https://webapps.dcs.gla.ac.uk/ETHICS/
2. Click on your year
3. Click on the piece of assessed coursework you are submitting

You are permitted to complete the declaration of originality within 24 hours of the deadline (to accommodate for the fact that it is available online only on the intranet, thus you have to be on a University machine to do so).

The following mark scheme will be used (/20):

- 5 marks for the design diagram showing dataflows and interactions in your system (Part 1)
- 15 marks for Part 2:
    - 10 marks for identifying and completing the functions (as described in your report)
    - 1 mark for each screenshot (build and run)
    - 1 mark if the code compiles correctly when tested by me
    - 2 marks if the code runs correctly when tested by me