A PThread-based Network Driver

Due at 11:59pm, on Tuesday, 29 May 2018

1 General Overview

A piece of code is required which will function as a (de)multiplexor for a simple network device. Calls will be made to this software component to pass packets of data onto the network. Packets arriving from the network will be demultiplexed and passed to appropriate waiting callers. The emphasis is on the concurrency, not on networking issues.

The core of your solution will be one or more threads, which move packets of data between queues or buffers (amongst other things). You will be programming in C and using the PThread libraries. PThreads enable you to dynamically create threads and provides support for both mutual exclusion locks and for condition variables.

A tutorial on Pthreads, together with sample code for a generic BoundedBuffer, is available on Canvas. If you have not already thoroughly read and understood the Pthreads tutorial, you should do so immediately.

2 Detailed Problem

You are to write a piece of code which provides the central functionality of a network device driver. The ADT you are producing will be called NetworkDriver, and you will make calls onto an instance of the NetworkDevice ADT (provided by the test harness). Calls will be made to your code from the application level, as well.

Your code will be passing around pointers to PacketDescriptors; this is an ADT about which you need to know almost nothing. One of the components of the system is a FreePacketDescriptorStore; when you create the store, you hand it a region of memory (a void * pointer and a length); this memory block is divided up into pieces that are the right size for a PacketDescriptor and the resulting PacketDescriptors are stored in the FreePacketDescriptorStore. The FreePacketDescriptorStore is an ***unbounded*** container, which will be used by your code and the test harness as a place from which to acquire PacketDescriptors. When they are no longer used, the PacketDescriptors must be returned to the FreePacketDescriptorStore.

You are provided an existing piece of code for the FreePacketDescriptorStore, along with other components of the test harness. Similarly, you may use the BoundedBuffer described above if that is helpful within your solution. *All other code must be your own work; plagiarism/collusion detection will be undertaken to check this is the case.* The Makefile in the starting archive assumes that all you are providing is networkdriver.c, an implementation of the networkdriver.h interface; it also assumes the c89 standard, and uses “-W -Wall -pedantic” as compiler flags.

The test harness includes fake applications, which will acquire PacketDescriptors from the store, initialize their contents and then pass them to your code for delivery onto the network.

The test harness includes a fake, full-duplex NetworkDevice; it can simultaneously transmit a packet onto the network and receive a packet from the network. For sending packets to the network, you can pass PacketDescriptors to the NetworkDevice for dispatch. The complication is that the NetworkDevice can only process one packet at a time, and is quite slow in doing so;

therefore, you may have to queue up other requests. An application thread should not have to wait for its packet to be placed on the network. Once it has handed it to your code, it should be allowed to return and engage in other activity. This means you are likely to need dedicated threads within your code to pass packets to the network device.

Once a packet has been sent onto the network, your code should pass the descriptor back to the free packet descriptor store.

The `NetworkDevice` will also pass packets from the network to your code. This will entail you having a `PacketDescriptor` ready and waiting and a thread blocked ready to handle the pseudo-interrupt saying a packet has arrived. You will need to set-up a new empty packet descriptor to await the next packet and block as soon as possible waiting for it. This means that any processing of a recently arrived packet must be minimal, handing it over to another (buffered?) part of your code for dispatch by a different thread.

Packets will need to be passed to the fake application layer when they are requested, and eventually the packet descriptors will be returned to the free packet descriptor store by the fake applications.

Because this code is supposed to live inside the OS, you are required to use bounded sized data structures within the parts of your code that handle the passing of packet descriptors between the applications and the network device – i.e. you cannot use `malloc()`. It's possible that application threads will block, either because they are awaiting incoming packets, because the buffers for outgoing packets are full (as those data structures are supposed to be bounded), or because there are no free packet descriptors available.

The thread that handles incoming packets must not block, except to await an incoming packet. So, either: the thread that processes the received packets must also not block; or, if it does and the buffer for received packets subsequently becomes full, the packet receiver thread must discard packets rather than blocking on a full buffer.

In summary, you have applications that run at their own pace, and a network device that runs at a very different pace and for which the send and receive requests must be serialized. Your network driver must bridge between the two, despite the speed mismatch; in particular, it must always be ready to receive a message from the network. You are to use Pthreads and bounded data structures to provide this bridging function to implement the spec in `networkdriver.h`.

The starting files for this exercise are available from Canvas.

You should start the design of your solution now, sketching out what you are trying to achieve, and the rough code needed to do it. Your sketch will most likely take the form of an interaction or collaboration diagram[1]. Your GTFs and I will be happy to review your design after you have completed it.

3 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. */
```

[1] See https://en.wikipedia.org/wiki/Unified_Modeling_Language and links therein.

The network device ADT 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 ADT is defined as follows – your code may invoke the defined methods:

```
typedef struct network_device NetworkDevice;

struct network_device {
   /* instance-specific data */
   void *self;

   /* 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 */
   int (*sendPacket)(NetworkDevice *nd, PacketDescriptor *pd);

   /* tell the network device to use the indicated PacketDescriptor
    * for next incoming data packet; once a descriptor is used it won't
    * be reused for a further incoming data packet; you must register
    * another PacketDescriptor before the next packet arrives */
   void (*registerPD)(NetworkDevice *nd, PacketDescriptor *pd);

   /* The thread blocks until the registered PacketDescriptor has been
    * filled with an incoming data packet. The PID field of the packet
    * indicates the local application process which should be given it.
    * This should be called as soon as possible after the previous
    * call to registerPD() to wait for a packet
    * Only 1 thread may be waiting. */
   void (*awaitIncomingPacket)(NetworkDevice *nd);
};
```

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

```
void blocking_send_packet(PacketDescriptor *pd);
int nonblocking_send_packet(PacketDescriptor *pd);
/* 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, 0=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**pd, PID);
int nonblocking_get_packet(PacketDescriptor**pd, 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 identifier 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 you 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);
/* 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: You hand back a FreePacketDescriptorStore into
 *      which PacketDescriptors built from the memory
 *      described in args 2 & 3 have been put */

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

You will need to use the FreePacketDescriptorStore ADT facilities. The ADT is defined as:

```
typedef struct free_packet_descriptor_store FreePacketDescriptorStore;

FreePacketDescriptorStore *FreePacketDescriptorStore_create(
                    void *mem_start, unsigned long mem_length);
void FreePacketDescriptorStore_destroy(FreePacketDescriptorStore *fpds);

struct free_packet_descriptor_store {
   void *self;
   void (*blockingGet)(FreePacketDescriptorStore *fpds, PacketDescriptor **pd);
   int (*nonblockingGet)(FreePacketDescriptorStore *fpds, PacketDescriptor **pd);
   void (*blockingPut)(FreePacketDescriptorStore *fpds, PacketDescriptor *pd);
   int (*nonblockingPut)(FreePacketDescriptorStore *fpds, PacketDescriptor *pd);
   unsigned long (*size)(FreePacketDescriptorStore *fpds);
};

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

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

```
PID getPID(PacketDescriptor *pd);
```

4 Pseudocode for Your Driver

```
/* any global variables required for use by your threads and your driver routines */

/* definition[s] of function[s] required for your thread[s] */

void init_network_driver(NetworkDevice *nd, void *mem_start,
                         unsigned long mem_length, FreePacketDescriptorStore **fpds) {
   /* create Free Packet Descriptor Store using mem_start and mem_length */
   /* create any buffers required by your thread[s] */
   /* create any threads you require for your implementation */
   /* return the FPDS to the code that called you */
}

void blocking_send_packet(PacketDescriptor *pd) {
   /* queue up packet descriptor for sending */
   /* do not return until it has been successfully queued */
}

int nonblocking_send_packet(PacketDescriptor *pd) {
   /* if you are able to queue up packet descriptor immediately, do so and return 1 */
   /* otherwise, return 0 */
}

void blocking_get_packet(PacketDescriptor **pd, PID pid) {
   /* wait until there is a packet for `pid' */
   /* return that packet descriptor to the calling application */
}

int nonblocking_get_packet(PacketDescriptor **pd, PID pid) {
   /* if there is currently a waiting packet for `pid', return that packet */
   /* to the calling application and return 1 for the value of the function */
   /* otherwise, return 0 for the value of the function */
}
```

5 Developing Your Code

5.1 Implementation ground rules

Any helper functions that you require in your network driver must be defined as static functions in networkdriver.c.

Your driver operates in limited context – it is called by the OS bootstrap and the fake applications; it may only invoke methods defined in the other .h files provided in the starting archive.

Since it is inside the OS, your driver must use bounded sized data structures; as a result, you may ***not*** use malloc()/free().

5.2 Using git and Bitbucket

You must develop your code in Linux running inside the virtual machine image provided to you; all the object files for the testharness supplied to you have been compiled in this way. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state.

You should use your Bitbucket GIT repositories for keeping track of you programming work. As a reference, you can perform the command line steps below to create a new project directory and upload it to your uoregon-cis415 repository.

```
% cd /path/to/your/uoregon-cis415
% mkdir project2
% echo "This is a test file." >project2/testFile.txt
% git add project2
% git commit –m "Initial commit of project2"
% git push –u origin master
```

Any subsequent changes or additions can be saved via the add, commit, and push commands.

5.3 Debugging Aids

5.3.1 valgrind

You are ***not*** required to use valgrind on mydemo to check for memory leaks; there has been no effort in the testharness to free allocated storage when the time limit has expired, so running your program under valgrind will always show memory leaks; since you are not permitted to use malloc()/free(), those leaks are my responsibility.

5.3.2 Checking for leaking packet descriptors

Your driver must not leak PacketDescriptors. mydemo can provide you with information regarding the current state of the FreePacketDescriptorStore as follows:

- define an environment variable TESTHARNESS_LOG_DESCRIPTORS as

  ```
  export TESTHARNESS_LOG_DESCRIPTORS=yes
  ```

- when you execute mydemo (or demo, for that matter) with this environment variable defined in this way, every 10 seconds it will print out log lines on stdout of the form

  ```
  FPDS> FreePacketDescriptors: out|free|total = xx|yy|zz
  ```

- If you wish to check that there are no leaks in your network driver, and do not want to see all the other logging information, you can execute the following command:

  ```
  ./mydemo | grep FPDS
  ```

5.3.3 Changing the amount of time that the test harness runs

When you invoke `demo` or `mydemo`, it will run for 120 seconds/2 minutes. If you specify a numeric argument to either program, as in `./mydemo 240`, that is the number of seconds that you wish the demo to run. Specifying an absurdly small number for the argument (e.g., 5) will not be an effective test of your code. Specifying a large number for the argument (e.g., 3600) means that it will run for a long time, so you can be sure that you do not leak packet descriptors.

6 Helping your Classmate

This is an individual assignment. You should be reading the manuals, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project's end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. If you cannot obtain help from the GTFs or the lecturer, it is possible that a classmate can be of assistance.

In your status report on the project, you should provide the names of classmates that you have assisted, with an indication of the type of help you provided. You should also indicate the names of classmates from whom you have received help, and the nature of that assistance.

Note that this is not a license to collude. We will be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

7 Submission[2]

You will submit your solutions electronically by uploading a gzipped tar archive via Canvas.

Your TGZ archive should be named “<duckid>-project2.tgz”, where “<duckid>” is your duckid. It should contain your “networkdriver.c” and a document named “report.pdf” or “report.txt”, describing the state of your solution, and documenting anything of which we should be aware when marking your submission; if you have not included your design figure in “report.pdf” (such as you are turning in “report.txt”), you must also include a file named “design.jpg” or “design.pdf” containing the design figure for your solution.

These files should be contained in a folder named “<duckid>”. Thus, if you upload “jsventek-project2.tgz”, then we should see the following when we execute the following command:

```
% tar –ztvf jsventek-project2.tgz
-rw-rw-r-- joe/None      5125 2015-03-30 16:37 jsventek/networkdriver.c
-rw-rw-r-- joe/None    629454 2015-03-30 16:30 jsventek/report.pdf
```

Your source file must start with an “authorship statement”, contained in C comments, as follows:

- state your name, your login, and the title of the assignment (CIS 415 Project 2)
- state either “This is my own work.” or “This is my own work except that …”, as appropriate.

[2] Note that if you do not structure your submission as defined in this section, your submission will not be graded, and you will receive a 0 for the project. If you do not remember how to create a submission with this structure, please review Section 6 of the handout for Project 0.

Marking Scheme for CIS 415 Project 2

Design and report (40)

- honest statement of the state of the solution (10)
- design diagram showing dataflows and interactions in your system (12)
- appropriate sizing of bounded buffers/other data structures (4)
- appropriate behaviour when there is a shortage of PacketDescriptor's for receiving packets (4)
- explanation of blocking behaviour matching the requirements (8)
- novel design features (2)

Implementation (60)

- ***workable*** implementation (30)
    - appropriate synchronization (6)
    - appropriate number and initialization of threads (4)
    - appropriate return of PacketDescriptor's (4)
    - low complexity search for next PacketDescriptor associated with the redeeming thread (6)
    - appropriate initialization of data structures (6)
    - sufficient commentary in the code (4)
- ***working*** implementation (10)
- No packet descriptor leakage (20)

TOTAL (100)