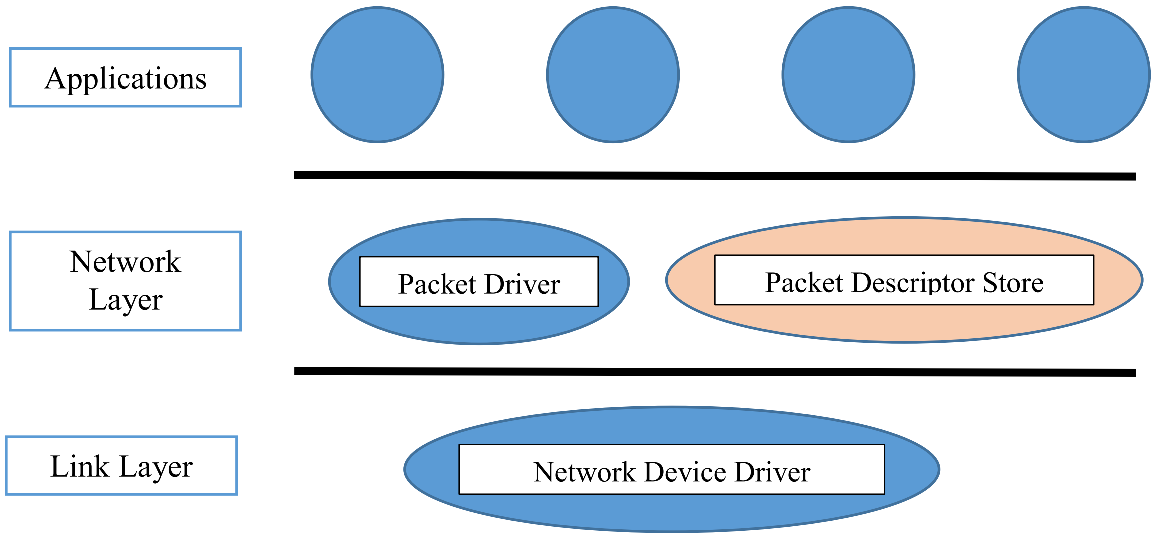
A PThread-based Packet Driver

Due at 11:59pm, on Monday, 9 May 2022

1 General Overview

You are part of a team constructing a new, lightweight operating system. The relevant architectural aspects of the OS are as follows …



Applications want to send packets to and receive packets from other applications over a network. We have a network device driver, a combination of software and hardware, already written for the chosen network medium. Your job is to write a packet driver that sends packets on the network and receives packets from the network. While this sounds like a straightforward task, there are many constraints on your design.

1.1 Applications

For this project, the applications are provided in an object file named `testharness.o`. Each application is distinguished by its `PID`. The following is from `pid.h`:

```
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. */
```

1.2 Packet Descriptors

The communications enabled by the OS require that an application wishing to send a packet must allocate a `PacketDescriptor` and load it up with the data to send over the network; it then hands the `PacketDescriptor` to your Packet Driver. When the Network Device Driver receives data from the network medium, it places the data in a PacketDescriptor that your Packet Driver has supplied to it; it is then your Packet Driver's responsibility to deliver the descriptor to the application to which the data is destined.

When your Packet Driver is initialized by the operating system bootstrap, it must create a PacketDescriptorStore using some memory provided to the driver as an argument to its initialization routine. Both applications and your Packet Driver will allocate PacketDescriptors and return PacketDescriptors at appropriate times during their processing.

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

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

After allocating a PacketDescriptor, you must reset the descriptor to be empty before registering it with the NetworkDevice.

```
    void initPD(PacketDescriptor *pd);
```

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.

1.3 The Network Device and its Interface

The network device is full-duplex, in that your packet driver can be sending a single packet and receiving a single packet simultaneously. This means that you are expected to have at least two PThreads active in your driver, one sending packets and the other receiving packets.

Since the network device can only send one packet at a time onto the communications medium, and the medium is not particularly high-bandwidth, it means that your packet driver will likely have a queue of outgoing requests from which to work. An application process should not have to wait for its packet to be placed onto the medium; once your code has accepted it for transmission, the application should be enabled to return and engage in other activity. Once a packet has been sent onto the network, your code should pass the PacketDescriptor 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.

After an application has finished using a received PacketDescriptor, it is that application's responsibility to return the packet descriptor to the free packet descriptor store.

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);
};
```

1.4 The Packet Driver

As you have likely surmised, your code is sandwiched between the applications, which run at their own speed, and the network device, which runs at a very different speed. Networking folks would tell you that your packet driver functions as

1) a multiplexor of packets from different applications onto the single network device; and

2) a demultiplexor that takes the stream of packets from the device and delivers them to the appropriate waiting callers.

We are not particularly interested here on networking issues - rather, the emphasis is using PThreads, mutexes, condition variables, and constrained resources to maximize concurrency in handling the speed mismatch between the applications and the network.

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, 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**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. */
```

The OS, when it boots up, will invoke the following initialization routine that you must implement in packetdriver.c, as well.

```
void init_packet_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 */
```

1.5 Summary

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 packetdriver.h.

2 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_packet_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 */
}
```

3 Developing Your Code

3.1 Implementation ground rules

Any helper functions that you require in your packet driver must be defined as static functions in packetdriver.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().

3.2 Using git and Bitbucket

You must develop your code in Linux running inside the virtual machine image provided to you; testharness.o and libTH.a in the starting archive have been compiled for DebianLinux on your host system architecture. Running in the VM 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.

3.3 Debugging Aids

3.3.1 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.

3.3.2 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
  ```

3.3.3 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.

4 Contents of P2start<arch>.tgz

This archive holds text files, and <arch>-specific executables, object files, and static libraries, that you will need to build the testharness with your packet driver. The contents are as follows:

file	description
BoundedBuffer.h	The header file for a thread-safe bounded buffer ADT; you may use instances for your bounded data structures in your packet driver.
demo	An <arch>-specific Debian executable that you can run to see how the model solution works.
destination.h	A header file that you may need to #include.
diagnostics.h	A header file that you may need to #include.
diags	A text file that describes diagnostic messages that you SHOULD NOT SEE when your mydemo program is running; they are indicative of coding errors.
fakeapplications.h	A header file that you may need to #include.
freepacketdescriptorstore__full.h	A header file that you may need to #include.
freepacketdescriptorstore.h	A header file that you may need to #include.
libTH.a	An <arch>-specific Debian library that contains all of the other object files needed to build mydemo.
Makefile	A makefile that builds mydemo from testharness.o, packetdriver.o, and libTH.a.
networkdevice__full.h	A header file that you may need to #include.
networkdevice.h	A header file that you may need to #include.
packetdriver.h	The header file that defines the global routines that you must implement.
packetdescriptorcreator.h	A header file that you may need to #include.
packetdescriptor.h	A header file that you may need to #include.
pid.h	A header file that you may need to #include.
queue.h	A header file that you may need to #include.
testharness.o	An <arch>-specific Debian object file that defines main().

5 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 GE, LAs, 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.

6 Submission[1]

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 "packetdriver.c" and a document named "report.txt", describing the state of your solution, and documenting anything of which we should be aware when marking your submission; you must also include a file named "design.jpg" or "design.pdf" containing the design figure for your solution.

Within the archive, these files should **not** be contained in a folder. Thus, if I upload "jsventek-project2.tgz", then I should see something like the following when I execute the following command:

```
% tar -ztvf jsventek-project2.tgz
-rw-rw-r-- joe/None      5125 2015-03-30 16:37 packetdriver.c
-rw-rw-r-- joe/None    629454 2015-03-30 16:30 design.pdf
-rw-rw-r-- joe/None      1234 2015-03-30 16:02 report.txt
```

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.

[1] 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)