COMP3301 2025 Assignment 2 - PCI device driver

- Due: Week 9
- $Revision: 526 $

1 Academic Integrity

All assessments are **individual**. You should feel free to discuss aspects of C programming and assessment specifications with fellow students and discuss the related APIs in general terms. You should not actively help (or seek help from) other students with the actual coding of your assessment. It is cheating to look at another student's code, and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion (outside of the base code given to everyone), formal misconduct proceedings will be initiated against you. If you're having trouble, seek help from a teaching staff member. Do not be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school website: https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct.

Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository. You must follow the following code usage and referencing rules for all code committed to your Git repository (not just the version that you submit), in Table 1.

Table 1: Code Origin and Uses/References

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you by COMP3301 teaching staff or posted on the discussion forum by teaching staff.	**Permitted** May be used freely without reference. (You must be able to point to the source if queried about it - so you may find it easier to reference the code.)
Code you wrote this semester for this course Code you have personally written this semester for COMP3301.	**Permitted** It may be used freely without reference, provided you have not shared or published it.
Unpublished code you wrote earlier Code you have personally written in a previous enrollment in this course or in another UQ course or for other reasons and where that code has not been shared with any other the person or published in any way or submitted for assessment.	**Conditions apply; requires references** May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code. If such code is used without appropriate referencing, then this will be considered misconduct.

Code Origin	Usage/Referencing
C Code from AI tools Whilst students may use AI and/or MT technologies, successful completion of the assessment in this course will require students to critically engage in specific contexts and tasks for which artificial intelligence will provide only limited support and guidance.	**Conditions apply; requires references** A failure to reference generative AI or MT use may constitute student misconduct under the Student Code of Conduct. To pass this assessment, students will be required to demonstrate detailed comprehension of their submission, independent of AI and MT tools.
Code copied from other sources Code, in any programming language: - copied from any website or forum (including Stack Overflow and CSDN); - copied from any public or private repositories; - copied from textbooks, publications, videos, apps; - copied from code provided by teaching staff only in a previous offering of this course; - written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); - written by an AI tool that you did not personally and solely interact with; - written by you and available to other students; or - from any other source besides those mentioned in earlier table rows above.	**Prohibited** May not be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken. This prohibition includes code written in other programming languages that has been converted to C.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions.	**Conditions Apply, references required** May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning is referenced in a comment adjacent to that code. If such code is used without appropriate referencing then this will be considered misconduct.

2 OpenBSD `duct(4)` Network Interface Driver

The task in this assignment is to develop an OpenBSD kernel driver to support a simplified network interface device.

The purpose of this assignment is to demonstrate your ability to read and comprehend technical specifications and apply low level C programming skills to an operating system kernel environment.

You will be provided with a specification for the Ductnet network device, the emulated device itself on the COMP3301 virtual machines, and userland code to test your driver functionality.

This document should be read in conjunction with the *COMP3301 A2 Device Interface Specification*, which contains the in-depth technical details of the hardware interface your driver must use.

3 Background

One of the most common types of device driver in a modern operating system is a driver for a Network Interface Card (NIC). These allow the system to communicate across various kinds of computer networks, including Ethernet, InfiniBand, Fibre Channel etc.

Drivers for real network interface cards are often quite complex in order to meet performance and scalability goals, and have to deal with hardware interfaces that are often very flexible and programmable. Their interface to the rest of the operating system is also quite complex, due to integration with the wider network stack within the kernel.

In this assignment you will be writing a driver for a very simple PCI network interface, connected to a basic network called Ductnet (held together by the proverbial duct tape). In order to focus on the core elements of device driver development and PCI hardware, we have greatly simplified both the network and the hardware interface. You will not have to deal with TCP/IP or the wider networking stack, but you will get a bit of a taste of the way that NIC drivers work and how they operate.

3.1 Packet-switched networks

In general in computer networking, networks are designed to be packet-switched (as opposed to circuit-switched designs which were often used for analog communications). This means that information is exchanged in discrete “packets” of data.

Packets on a network may be of a fixed length, or variable length, or a mix (e.g. variable length with some specific limits on both minimum and maximum size). In general they contain three parts: a *header* (which is metadata about the packet, usually including its length, which goes at the start), the *data* itself, and then sometimes a *trailer* (usually including a checksum).

Since networks usually involve more than just two entities communicating, the header of a packet usually incorporates some kind of *address*, which indicates which computer on the network the packet is being sent to or from.

In the simplest type of network, called a *bus* or *hub* network, there is only one single shared medium of communication which all the computers (*stations*) share. Every station can observe packets intended for any other station, but generally will only process packets where the packet header includes an address that they are “interested in”.

Some networks only support packets being addressed to a specific other station on the network: this is often referred to as *unicast* networking. Others support forms of *broadcast* or *multicast* traffic as well, where packets can be addressed to all stations on the bus, or some subset of interested stations.

In more complex networks, the packet *data* field often contains another header for a different type of packet (or *frame* or *segment* or other terminology), forming a “stack” of layers. This is usually done so that multiple different services or types of services can share one lower-layer address on the network, and also to allow the creation of more complex protocols on top of the lower-level transport (e.g. TCP sits on top of IP in order to provide reliably in-order message delivery even across lossy networks).

The Ductnet network used in this assignment is a packet-switched bus network with both unicast and multicast addressing. There is only a single layer of packet headers used on the network.

3.2 The PCI bus

The most commonly used and reliable means by which network interface cards connect to a host CPU is via the PCI bus.

PCI (the Peripheral Component Interconnect) originated on the Intel x86 PC platform, but has also become used on other CPU architectures. It specifies physical connectors and electrical signalling of add-in cards and components, as well as the logical data link traffic layer on top of it, and most aspects of the host software interface used by operating systems.

PCI focuses primarily on peripherals which can be operated via memory-mapped registers, and most modern PCI devices also have DMA capabilities. It allows for device auto-discovery by the OS and system firmware, as well as configuration of the location and size of memory-mapped register regions, and provides some basic control to e.g. enable and disable DMA for each device.

The interface between the Ductnet interface hardware and your host VM will come in the form of an emulated PCI device.

In this course, the tutorial content and practicals 4 and 6 provide a lot of information and practical experience with implementing drivers for PCI devices on OpenBSD. You will be using this experience and content for this assignment.

3.3 Descriptor rings

The device driver interface to memory-mapped devices has undergone several stages of evolution over the life of the PCI bus, as demands for performance and capacity for memory bandwidth have both increased.

Many early devices focused entirely on memory-mapped I/O, where drivers would read and write all traffic via the device BAR. This was also prudent in an era where DMA support was sometimes patchy and unreliable across different hardware.

In the modern era, DMA use is ubiquitous and devices focus on ways to improve performance by making the best out of this capability.

One of the most common ways modern devices make use of DMA is via descriptor rings. These are regions of memory describing commands sent to the device, or responses from it, which are operated as a circular buffer. Host device drivers write new commands into the circular buffer, and the device reads them and carries them out. This enables the host to produce a lot of commands in a short period, and have the device carry them out in order without having to wait for previous commands to finish.

For network interface cards, there are typically multiple rings, with some rings set aside for transmitting packets from the host onto the network (*TX rings* or *transmit rings*) and some set aside for receiving packets from the network (*RX* rings or *receive rings*).

The host will generally keep the NIC's TX rings empty, only adding packets to them when there is something to transmit, and keep the NIC's RX rings full, with pre-prepared buffers for the NIC to write received packets into at any time when they arrive.

As packets arrive, the NIC driver will handle the packet and then eventually return the buffer to the RX ring to be re-used for another packet.

Many NICs also have separate rings for *control commands*, which usually have a different structure to the RX and TX rings.

The PCI device in this assignment uses an interface based around descriptor rings. It includes a single TX ring, a single RX ring, and a single control command ring.

3.4 Non-Blocking I/O

Traditionally, and by default, when a program performs I/O on a file descriptor the kernel may wait or sleep in the syscall handler until the related operation makes progress. For example, if a program is fetching data off a web server using a network socket it will perform a series of `read()` syscalls against the socket file descriptor. If the server has not yet sent data, the kernel will de-schedule the calling program (put it to sleep) while it waits for data from the server. Only when data has been received by the network socket will the kernel wake the waiting program up and finish handling the read syscall. From the perspective of the program, this read syscall appears to "block" further execution of the program until data has been received from the network.

If the program has other work it could do while waiting for data from the network server, it can configure the file descriptor for non-blocking operation. In this situation, if the program attempts to `read()` from the socket file descriptor when no data is available, the kernel will return immediately with an error, setting `errno(2)` to `EAGAIN`. The program can detect this and move on to perform other work. When data does become available later, a subsequent read will proceed as normal.

This relies on the kernel being able to receive data on behalf of the file descriptor outside the context of the read syscall handler. Generally, this means the file descriptor has an associated buffer that can be filled in when the data becomes available (from the network, or a USB device, etc). When the program then tries to read from the file descriptor, it uses the buffered data.

Similarly, writes from a program can also block. A `write()` to a TCP network socket may block until the remote end of the connection has received and acknowledged the the data. A non-blocking write may fill a buffer and return immediately, with subsequent writes failing with `EAGAIN` until space in the buffer becomes available again. The kernel is then responsible for transmitting data out of the buffer and emptying it when the remote end has acknowledged the data.

For non-blocking read and write operations, the kernel is performing work associated with the file descriptor on behalf of the program while the program is doing other work. When this associated work has completed (e.g., a write buffer has space again, or a read buffer has data in it) the program likely also wants to be notified, so that it can perform further writes or reads.

The standard UNIX facilities for handling this notification are the `select(2)` and `poll(2)` syscalls. In OpenBSD, the `select(2)` system call is implemented as a wrapper around `poll(2)`, and drivers such as `duct(4)` only have to implement a handler for poll.

These allow programs to check a large set of file descriptors at once for which (if any) are currently available for `read()` or `write()`. They can also be configured to sleep (or block) until at least one of the set is available, or a specified amount of time has elapsed.

When read buffer has been or is filled by the kernel, it reports that fact to the poll facility, which will (if necessary) wake up the program and report which file descriptor has become readable.

This facility is often used by programs to enable event-driven I/O multiplexing across a large number of file descriptors (e.g. a network server handling many clients at once, without needing a separate thread per client). However, `select(2)` and `poll(2)` struggle to scale beyond a certain number of file descriptors or events due to their requirement to read in the entire set of FDs every time they are called. Different operating systems provide alternative (but non-standard) facilities to address this. OpenBSD, as a member of the BSD family of operating systems, uses the kqueue event notification facility.

In OpenBSD 7.1 and later, the `select(2)` and `poll(2)` system calls are internally implemented on top of the kernel facilities for `kqueue(2)`. This means that drivers only have to implement their side of `kqueue(2)` and then support for `select(2)` and `poll(2)` will be available "for free".

In this assignment, for full marks, your device driver will need to implement support for the `kqueue(2)` event notification mechanism.

4 Specifications

You will implement a driver called `duct(4)` that will attach to the PCI device specified in the Device Specification document, and provide access to the Ductnet network via a character device special file under `/dev`.

You may assume that userland tools which use your driver will directly open the device and use the `read`/`write` and `ioctl` interface specified below. You do not need to implement any additional userland components to go with it.

4.1 Device functionality

All the basic features of the device must be implemented by your driver. These include:

- Allocating and configuring descriptor rings
- Sending control commands to start packet processing and add a unicast address filter
- Transmitting packets via the TX ring
- Receiving packets via the RX ring
- Adding and removing multicast address filters based on `ioctl` requests
- Detecting hardware errors in the `FLAGS` register

You should also implement the ability to stop packet processing and release any buffers on the RX ring when no userland program has the device file open.

You may, if you wish, also implement resetting the device following a fatal error. However, there will be no additional marks awarded for doing so.

Your device driver must be able to handle transmitted and received packets with data payloads up to 16384 bytes in total length.

You may choose the strategy your driver uses for checking the `FLAGS` register yourself. Upon detecting an error in the `FLAGS` register, your driver should print a brief description of the error to the system console and `dmesg` (e.g. using the kernel `printf(9)` function).

Error conditions detected in the `EVFLAGS` register (RX drops or jumbos) should also be detected and a brief description of the error should be printed to the system console and `dmesg` in the same manner as for fatal errors.

If a command is in progress at the time a fatal occurs, then your driver must also return an error to userland as described below (e.g. a `read` in progress must not "hang", it must immediately return `EIO` to userland).

4.1.1 Concurrency

For full marks (see the criteria sheet for further details) your device must support concurrent use of the device special file interface by multiple userland processes and/or threads at once. This includes both multiple single-threaded processes, and multiple threads within the same process, all of which may be using the same device major/minor node at once.

When multiple threads or processes enter the `read()` function at the same time, it is permissible for arriving packets to be given to them in any order. If some number of threads block in `read()` and then that number of packets arrive without any further threads entering `read()`, all of the `read()` calls should return. However, we will not require a guarantee of "fairness" between multiple threads calling `read()` repeatedly, so having some situations where one of the readers may "starve" (never return any data) if multiple threads are making repeated racey calls is acceptable.

If multiple threads or processes enter the `write()` function or an `ioctl()` at the same time, all of their operations are required to complete just as if they were entered sequentially, but the order the packets end up on the wire across thread or processes does not need to be well-defined.

4.2 Device special file interface

The `duct(4)` device special file should use major number 102 on the amd64 architecture for a character device special file.

The device minor number maps 1:1 with the `duct(4)` unit number.

The naming scheme for the special files in `/dev` must be of the form `/dev/ductX`, where `X` is the unit number as a numeric string.

Examples:

Device	Major	Minor	(Unit)
`/dev/duct0`	102	0	0
`/dev/duct1`	102	1	1
`/dev/duct2`	102	2	2

Your driver may, if you wish, return additional `errno(2)` values as well as those specified below, in situations that are not covered by the `errno` values and descriptions listed.

4.2.1 open() entry point

The `open()` handler should ensure that the minor number has an associated kernel driver instance with the same unit number attached.

Then it should set up any resources needed for the device to begin operation. This is a good place to allocate any expensive resources that should only be consumed if the device is actively being used.

Access control is provided by permissions on the filesystem (your driver code does not need to perform extra permissions checks).

Whenever a file descriptor for the device is open in userland, the driver must place the device in a state where it is able to receive unicast packets (even if no thread or process has called `read()` yet).

If the device has been opened, and `read()` has not been called, at least 64 packets must be able to be queued up waiting for userland to collect when it calls `read()`. If more than 64 packets arrive before `read()` is called, then it is permissible to drop the remaining packets (either in driver code or in hardware).

The open() handler should return the following errors for the reasons listed. On success it should return 0.

ENXIO no hardware driver is attached with the relevant unit number
ENOMEM the kernel was unable to allocate necessary resources

4.2.2 close()

Can be used to release any resources that were allocated by `open()`. It is permissible for some driver resources allocated at `open()` to not be released at `close()`, if these will be re-used by a subsequent `open()` or are still in use by hardware at the time.

If an I/O operation is currently in-flight (i.e. a `read()` is currently being processed in another thread), then `close()` must block until that call has completed.

When all extant file descriptors for a given `duct(4)` device have been closed, any multicast filters which have been added with the `DUCTIOC_ADD_MCAST` ioctl must be removed from the device.

`close()` must not return an error. If any I/O operations during `close()` fail, or the device returns an error, then a message should be written to the system console and `dmesg`.

4.2.3 read()

`read()` receives a packet from the Ductnet network and places it in a userland-provided buffer.

A single `read()` call will always receive only a single Ductnet packet. If userland does not provide sufficient space for the packet in its buffer to the kernel, then the packet will be truncated and the remaining bytes dropped.

Packets are written into the userland buffer with their entire Ductnet header intact at the beginning of the buffer (`SOURCE`, `DESTINATION`, `LENGTH` and the reserved word – see the Device Specification document for details). Userland may use `readv()` to provide separate buffer space for the header if desired (the driver should use the UIO framework to handle this as normal).

The `read()` call will block by default until a packet is available. If non-blocking I/O is requested for the file descriptor and no packet is ready to write out to userland immediately, it should return `EAGAIN`.

A blocking `read()` must block in an interruptible fashion, such that userland processes can receive signals such as `SIGINT` (e.g. resulting from `Ctrl+C`) and immediately exit the system call.

EAGAIN non-blocking I/O was requested, but the operation would block (no packet is currently waiting)
EIO a hardware error prevented receiving any packets

4.2.4 write()

`write()` transmits a packet to the Ductnet network.

A single `write()` call will always transmit only a single Ductnet packet. If userland does not provide a full Ductnet packet in the buffer given to the kernel, then an error is returned and no packet is transmitted.

Packets should be given to the kernel with their entire Ductnet header at the beginning of the buffer (`SOURCE`, `DESTINATION`, `LENGTH` and the reserved word). The kernel or the hardware device will overwrite the contents of the `SOURCE` field and its value will be ignored, but it still must be present in the buffer. Userland may use `writev()` to provide a separate buffer for the header if desired (and the driver should use the UIO framework to handle this as per normal). The `LENGTH` field must be validated against the amount of data provided by userland by the driver, and `EINVAL` returned if it does not match up.

The `write()` call will block by default until the packet has been transmitted. If non-blocking I/O is requested for the file descriptor, and space is available on the TX ring, then the call should make a copy of the userland data for transmission, add it to the ring, and return immediately. If no space is available on the hardware device's TX ring, a non-blocking `write()` should immediately return `EAGAIN`.

A blocking `write()` must block in an interruptible fashion, such that userland processes can receive signals such as `SIGINT` (e.g. resulting from `Ctrl+C`) and immediately exit the system call.

EINVAL the packet header was invalid, or data truncated
EAGAIN non-blocking I/O was requested, but the operation would block (no space on the TX ring available)
EIO a hardware error prevented receiving any packets

4.2.5 kqfilter()

The `kqfilter()` function entry point is used by the kqueue subsystem to request information about the usability of the open device instance, for read and write operations.

The following table summarises the possible events which can be observed using `kqueue(2)` on a `duct(4)` device:

Event	Reason
W	ready to write a new packet (space available on the TX ring)
R	a new packet is ready to copy into userland

4.2.6 ioctl()

4.2.6.1 DUCTIOC_GET_INFO

Returns the device version and local address information, in a `struct duct_info_arg` (defined in `<dev/ductvar.h>`):

```
struct duct_info_arg {
        uint32_t        duct_major;
        uint32_t        duct_minor;
        uint32_t        duct_hwaddr;
};
```

ENXIO no hardware driver is attached with the relevant unit number

4.2.6.2 DUCTIOC_ADD_MCAST

Adds a multicast filter to the Ductnet device. Takes a `struct duct_mcast_arg` (defined in `<dev/ductvar.h>`) as input.

```
struct duct_mcast_arg {
        uint32_t        duct_mask;
        uint32_t        duct_addr;
};
```

The `DUCTIOC_ADD_MCAST` ioctl blocks until the command is completely finished.

This `ioctl` command can only be used with multicast filters (not unicast).

It must block in an interruptible fashion, such that userland processes can receive signals such as `SIGINT` (e.g. resulting from `Ctrl+C`) and immediately exit the ioctl system call.

It is permissible for an interrupted command to still continue executing on the device and have its output discarded by the driver.

EINVAL the filter argument is not a multicast filter (i.e. `duct_addr` or `duct_mask` does not have the high bit set)
ENOSPC too many filters have been added for the hardware to support adding another
ENXIO no hardware driver is attached with the relevant unit number
EIO a hardware error prevented handling this command

4.2.6.3 DUCTIOC_RM_MCAST

Removes a multicast filter from the Ductnet device. Takes a `struct duct_mcast_arg` (defined in `<dev/ductvar.h>`) as input.

```
struct duct_mcast_arg {
        uint32_t        duct_mask;
        uint32_t        duct_addr;
};
```

The `DUCTIOC_RM_MCAST` ioctl blocks until the command is completely finished.

This `ioctl` command can only be used with multicast filters (not unicast).

It must block in an interruptible fashion, such that userland processes can receive signals such as `SIGINT` (e.g. resulting from `Ctrl+C`) and immediately exit the ioctl system call.

It is permissible for an interrupted command to still continue executing on the device and have its output discarded by the driver.

EINVAL the filter argument is not a multicast filter (i.e. `duct_addr` or `duct_mask` does not have the high bit set)
ENOENT no filter has been previously added which exactly matches the input
ENXIO no hardware driver is attached with the relevant unit number
EIO a hardware error prevented handling this command

4.3 Reflection

Reflect on your implementation by answering the following questions:

- What steps have you taken to complete this assignment? List the steps and describe the step that you found the most challenging.
- What bugs have you encountered? Describe one bug. If you did not encounter any bugs, describe something which you believe was tricky.
- How did you debug the bug which you have identified earlier? List the steps taken and tools used (e.g., `ddb`, `printf`). If you did not encounter any bugs, explain why it is a good idea to induce a kernel panic as soon as you realise something in the kernel has gone wrong.
- How did you fix the bug which you have identified and debugged earlier? If you did not encounter any bugs, explain why it is important to use memory barriers when dealing with MMIO (memory mapped I/O).

These questions are open-ended, there is no expected answer. Upload your reflection as a PDF file to the Blackboard A2 reflection submission. Page length is a maximum of 2 pages or less. **PDF name must be STUDENT_NUMBER_A2.pdf**. Note this is your `4XXXXXXX` ID number and not your `s4XXXXXX` login.

4.4 Testing

4.4.1 Network services

On the Ductnet network that your VM is connected to, there are several pre-provided services at fixed addresses.

- At address `0xEC60` is an "echo" server. When any packet is sent to this address, it will reply with a packet containing the exact same data.
- At address `0xBEEF` is a "quote of the day" server. When an empty packet is sent to this address, it will reply with a packet containing a randomly chosen quote as ASCII text.

There is also a provided client for a `talk`-like chat service, operating on multicast group `0x8000CAFE`. There will be a bot user present on this chat service which will reply to commands like `!botsnack` sent in the chat room.

4.4.2 Userland tools

In the base patch for this assignment, we provide several tools:

- `ductctl` is a basic command that opens a Ductnet device and retrieves information about it. It can also send and receive a single packet at a time (in blocking mode) if desired.
- `ductclient` is a simple client for the echo and QOTD services
- `ductchat` is a client for the `talk`-like multicast chat service. It uses non-blocking I/O.

4.4.3 Control interface

Your VM control interface will be extended to include control commands related to the A2 device, including the ability to introduce error conditions.

The commands for controlling this will be documented in the updated *COMP3301 VM Control Interface* document.

4.4.4 Other testing

You can also write your own additional tools for testing your device driver. You may find this the easiest way to trigger or examine certain kinds of bugs in your implementation.

This can include writing your own Ductnet network service if you wish. Communicating your VM's Ductnet address to other students is permissible (and even encouraged as a form of testing), as is exchanging packets with other students over the network.

Collaboration with other students *specifically on userland client tools* is permitted. However, you should be careful about conversations that may stray too far into sharing specific details of your driver implementation, which must be your work alone.

All Ductnet traffic will be logged and monitored by University staff.

4.5 Code Style

Your code is to be written according to OpenBSD's style guide, as per the `style(9)` man page.

An automatic tool for checking for style violations is available at https://stluc.manta.uqcloud.net/comp3301/public/cstyle.pl. This tool will be used for calculating your style marks for this assignment.

4.6 Compilation

Your code for this assignment is to be built on an `amd64` OpenBSD 7.7 system identical to your course-provided VM.

We will only be compiling and installing the kernel for this assignment; any modifications you make to userland testing tools for your own testing will be ignored when we mark your work.

Your kernel code must compile as a result of `make obj; make config; make` in `sys/arch/amd64/compile/GENERIC.MP`. relative to your repository root.

As a result of marking only your kernel, you may not modify the userland interface from what is shown above or the structures defined in `ductvar.h`.

4.7 Provided code

A patch will be provided that includes source for the userland client tools, and the `sys/dev/ductvar.h` header file containing the ioctls that the `duct(4)` driver must implement.

The provided code which forms the basis for this assignment can be downloaded as a single patch file at:

https://stluc.manta.uqcloud.net/comp3301/public/a2-base.patch

You should create a new `a2` branch in your repository based on the `openbsd-7.7` tag using `git checkout`, and then apply this base patch using the `git am` command:

```
$ git checkout -b a2 openbsd-7.7
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/a2-base.patch
$ git am < a2-base.patch
$ git push origin a2
```

The patch contains new headers, so you will need to run `make includes` once you have applied the patch:

```
$ cd /usr/src/include
$ doas make includes
```

To build the provided userland components, you will need to change into their relevant directories and compile and install them:

```
$ cd /usr/src/usr.bin/ductctl
$ make obj
$ make
$ doas make install

$ cd /usr/src/usr.bin/ductclient
$ make obj
$ make
$ doas make install

$ cd /usr/src/usr.bin/ductchat
$ make obj
$ make
$ doas make install
```

4.8 Recommendations

You should refer to the course practical exercises (especially pracs 4, 5 and 6) for the basics of creating a PCI device driver and attaching to a device.

The following are examples of some APIs you will need to use in this assignment:

- The `bus_space(9)` family of functions
- The `bus_dma(9)` family of functions
- `pci_intr_map_msix(9)` and `pci_intr_establish(9)`

5 Submission

Submission must be made electronically by committing to your Git repository on `source.eait.uq.edu.au`. In order to mark your assignment the markers will check out the `a2` branch from your repository. Code checked in to any other branch in your repository will not be marked.

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code to your repository, not binary artefacts, cscope databases, or patch files.

Remember to use `git status` before committing to examine the list of files being added.

Your `a2` branch should consist of:

- The openbsd-7.7 base commit
- The a2 base patch commit
- Commit(s) implementing the `duct(4)` driver

We recommend making regular commits and pushing them as you progress through your implementation. As this assignment involves a kernel driver, it is quite likely you will cause a kernel panic at some point, which may lead to filesystem corruption. Best practice is to commit and push before every reboot of your VM, just in case.

5.1 Marking

Your submission will be marked by course tutors and staff, during an in-person demo with you, at your weekly lab session.

You must attend your lab session in-person, otherwise your submission will not be marked. Online attendance (e.g. via Zoom) is not permitted.

6 Testing

We will be testing your code's functionality by compiling and running it in a virtual machine set up in the same way we detail in the Practical exercises.

Tests will be run against the same virtual agent device available in your personal virtual machine for the course.