David Butterworth
Operating System Device Driver
MSci Hons Computer Science
(with Industrial Experience)
21st March 2014
I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the Department's use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date:

Signed:
Abstract

With conventional Operating Systems being designed at a time where a single processor was the norm, they have decisions within the core of the system that are no longer true. While these systems now support multiple core machines it is worth working from first principles to see what a modern operating system could look like. In this report a new operating system is examined that utilizes an alternative approach and a network driver is designed to fit with this system.
Chapter 1

Introduction

The primary aim of this project is to provide a design for an ethernet driver for the experimental node-based distributed modular exokernel Operating System currently being developed at Lancaster University. In addition to defining the design for an ethernet driver this report will also look at the overall device driver interface, PCI bus access, and a Serializer. An example driver will be provided for the Intel 82574 Gigabit Ethernet Card. This driver will be capable of sending and receiving ethernet packets from a network, detecting and configuring the network card.

Working Documents are available at http://www.lancaster.ac.uk/ug/butterwd/
Chapter 2

Background

This section will briefly define a standard computer system, and then go into more detail about some of the differences between this and the computer system that the project is addressing. An overview of the expected structure of the entire system.

2.1 Standard View of a Computer System

A standard computer system can be defined on many levels, from a personal computer all the way up to a large hadoop cluster and beyond. One of the things that these often have in common is shared memory, if not shared disk. This means that memory can be passed simply by handing a pointer around. While when on a single machine this is efficient, when scaled out to multiple machines the process is limited by the maximum bandwidth between the machines.

2.2 View of a Computer System used within this Project

The system that this project is based on tackles two key concepts. Firstly the concept of Inter Process Communication (IPC) as entirely separate to network communication needed to be reconsidered. In many ways there is no reason for there to be separate protocols with current technology. This reduces the processing overhead needed to handle the conversion from internal IPC into the various external network protocols.

In this system memory is also not shared between processes, instead everything is IPC. While this may be wasteful when only on a single machine it is possible to speed this up substantially as demonstrated in [3] where it is shown that it is not always necessary for passed data to incur the time penalty of using memory and can sometimes be carried out purely using registers. The benefit is evident when using more than one machine, as memory does not need to be replicated, in addition to this most files are small so over the fast networks that are available today they can be passed across the network in very little time, in some cases faster than the user would notice.
The overall structure of the system is best described in two ways. Firstly in a tree structure, as any given node within the system can have zero or more children and up to one parent. This would extend in both directions until a single top node, and nodes which have no children are reached.

This leads to the second way to describe the system, in that at any given level the system can be treated in exactly the same way. This is due to any node upon receiving a job that it must execute to either accept the job if it or its children have the ability to accept more work, or reject it so that its parent can reassign it to another node. Upon accepting a given job then that node is responsible for either executing the job itself, or passing the job down to one of its children. That child would then go through the same process of deciding if it can accept the job or not.

If a node and its children are not able to execute a job then it must pass it upwards to its parent if it has one, or signal the job dispatcher that the execution has failed due a lack of resources. As passing jobs around the system would itself use resources, at each stage in this process the chances of the job being accepted needs to be weighted against the cost of actually making the transfer.

At this stage it becomes essential to define exactly what is meant by a node within the system. An individual node must meet three criteria, have some form of processor, memory, and have the ability to communicate with the wider system.

The model does not restrict the form that these must take thus meaning that the range of devices that can be classified as a node include everything from microchips to Field Programmable Gate Arrays (FPGA), intelligent network cards, and conventional computers. While building a system from components that only satisfy the minimal requirements would be possible, it is very unlikely that this in fact would be the case. To cover this all additional functionality of a given node is described within a feature list. This feature list would include items such as a keyboard or mouse being attached, a display being available for use, the speed, number of cores, and bit-width of the processor.

The system itself is highly modular, as such all functionality is broken down into logical sections, with particular care taken in reducing code duplication. A common example is that many types of operation require a buffer, however rather that requiring each module implements its own version, extracting this code into an independent module results in a greater chance of code reuse and can in some cases improve readability of the module itself.

The system utilizes a similar model to android where an enable and disable call is provided to let the module perform any clean up required and save state if applicable, this method will also be used in migration of modules between nodes. The returned value from enable and disable will contain information about if a module is able to be disabled, migrated, or if there are special requirements for the module to be able to run.

The message passing method used within this system is based on two bidirectional channels that can send uint32_ts. These channels are denoted as data and signal. While no actual restrictions are placed on what can be sent down each of the channels conceptually they are used for different classes of information. The signal channel is for control messages and configuration data, where the data channel is for the information required for the
operation of the specific module. For example a graphics card would be likely to take a representation of what should be displayed on the screen. The two channels are referred to collectively as a Pipe.

System service architecture (search service) mention that it currently does not exist but is relevant as it will be a central part of the completed system.

A potentially key problem with this system is that detection of services, however as this problem has been solved in multiple different ways already it is considered out of scope. Within this document the method of detection of services will be referred to as the System Service architecture.
Chapter 3

Design

This section will address a proposed design for the PCI interface, the Serializer, the Ethernet Driver interface, the Network driver, and a high level design for the Network Stack. The design decisions made here are assuming a completed system around the sections that are discussed.

3.1 PCI

The design for PCI is largely already described by the specification.[5] As such the only real design decision was to decide on the access method that will be provided. Two approaches can be taken with this, firstly have PCI access controlled by an independent module, or wrap it within a system service that contains other similar modules. If there will only be a single instance of the PCI module on a given node then memory mapped I/O is likely to be more efficient, however if multiple instances are attempting to use the same memory it would result in an increased chance of memory corruption occurring. In addition to this as there are several operations that require locking it would likely cause undefined behaviour if conflicting operations occur at the same time.

In addition to this it is likely to be more efficient if when the PCI module is instantiated if it walked the entire PCI address space and built an internal representation of the state of the PCI bus. In order to support hotplugging of devices either PCI Express hotplug or Advanced Configuration and Power Interface (ACPI) hoplug would need to be implemented.[5, p 439]

3.2 Serializer

The Serializer is intended to send any datastructure through the existing pipe infrastructure, due to the nature of the system memory can not be reliably passed by simply using a pointer as in a large portion of existing systems, instead the memory needs to be manually sent to anything that needs to use it. This simple serializer is intended to handle a simple
datastructure that contains no variable length fields or pointers. If those are required then either a separate serializer or one that wraps this and provides additional functionality would be needed.

The underlying pipe that is being used must not be used for other types of data while the serializer is operational. This prevents any corruption from occurring within the data, if at a later point a type of pipe is available that can handle locking then this will be readdressed.

The API used to access the serializer is intentionally similar to the regular pipe interface simply to maintain consistency. The key difference being that the write call to both channels now takes an additional parameter of size, which is defined as a number of bytes and takes a pointer to the data to be sent rather than a uint32_t, and the read takes a void * buffer and returns a uint32_t to represent the size of the datastructure that has just been written into the buffers provided.

By handling the buffers in this way it allows the modules using the serializer to handle the allocation and release of memory using an internal policy system rather than this being enforced on them by the serializer.

In normal operation of the serializer it should receive a pointer to a memory area, and information stating how large the item to be sent is. At this point the serializer should send a framing start message over the internal pipe, followed by the size of what is being sent, and send the data across followed by a checksum and a framing end message.

```cpp
class Serializer {
    public:
        Serializer(Pipe * pipe);
        uint32_t read(void * buffer);
        void write(void * data, uint16_t size);
        uint32_t ready();
        void signal(void * data, uint16_t size);
        uint32_t_signal(void * buffer);
        uint32_t hasSignal();
};
```

Listing 3.1: Proposed Serializer API
### 3.3 Device Driver API

A large portion of the consideration for the Driver API was simply deciding what was required across all classes of device. If a feature was required for a single class of device but for no others then to fit with the minimal nature of the rest of the system, and to not add any additional overhead for all other devices then it would not be included within the DeviceDriver API.

This is vitally important as the same API would be used to communicate with a serial port, and to a graphics card. Utilizing the existing Pipes infrastructure to actually do any communication with the driver using a specific protocol that can be different according to the requirements of the devices results in a very simple API actually required, as a driver is still a module an enable and disable call needs to be included, and the constructor needs to take a Pipe. Other than that there is very little actually required as the independent data and signal channels within a Pipe can handle everything from configuring the device over the signal channel, to actually using the device with the data channel.

The format of the data channel in particular is undefined and is not required to be consistent for the different devices.

In the case if a serial driver, it would be able to send all the data within a single message over the data pipe, as all other relevant information would already have been configured over the signal pipe.

In the more complex case of a graphics card doing OpenCL calculations there is likely to be a much larger amount of data being transferred than the uint32_t that a single write to the underlying Pipe can handle without further abstraction layers. A graphics card can of course also be used for its original purpose of graphics processing, but this again would be spoken to via the same methods, but with a different configuration such that the result of the operation is displayed on screen rather than returned to main memory.

Each of these use cases are already covered by the internal Pipe architecture, as such there is no reason to have a complex interface for a standard device driver and instead a very simple design is proposed containing a constructor and two functions.
class DeviceDriver
{
    public:
        DeviceDriver( Pipe *pipe );
        int enable();
        int disable();
}

Listing 3.2: Proposed Device Driver API

The enable and disable calls are required so that the system itself can treat the Driver in the same way as a regular Module so that no special case is required.

In general it is not relevant to migrate a device driver due to it being locked to a piece of hardware so the disable call would return with a value informing the calling process of this. A driver will still need to be able to be completely shutdown then a it needs to be able to be forced, or be initialised with a call over the signal channel in which case it should perform any cleanup that is required and then return with a value signifying success.

3.4 Ethernet Driver for a PCI Network Card

The function of the network driver itself should be minimal, providing only the ability to initialise, configure and reset the card, and send and receive a packet from the network. It should also contain methods to access any information specific to the network card itself, for example the statistical counters.

The primary reason for this is to help reduce the amount of additional processing that the driver needs to do, increasing the proportion of the time that the driver is doing useful tasks when running. The secondary reason being to reduce the number of unnecessary features that are implemented within a driver, fitting with the modular nature of the rest of the system.

As one of the requirements for the driver is that it should handle all input from the network card and pass it along to the next layer of the network stack as soon as possible. While desirable from a purely performance standpoint this is also important when consideration is given to the potential loops that are created due to the nature of the overall system. It is possible if not likely that a configuration command for the card will come from an external node. This results in the command needing to travel over the network link, and then be handled at each level of the network stack before it can be passed into the driver over the signal channel.

Building on system features already described, the driver will be created as a module that is locked to an individual node within the system. As part of the constructor a pipe is handed to the driver so that it can communicate to the rest of the system. Once the connection via pipe is available then the rest of the configuration can occur purely using the signal channel.

On enabling the driver it should utilize the system service architecture to find the appropriate network card. Depending on the type of search carried out then the driver should
validate that it has been handed the correct device. Depending on the outcome of this it should either continue with the initialisation, reattempt the search or signal failure to the system. The initialisation sequence should preform all operations required to place the network card into a know state and to be ready for data transfer between the host system and the wider network. This initialisation will generally consist of resetting the card, configuring the network link, setting up the receive and transmit queues in addition to any network card specific configuration. Any interrupt vectors required should also be mapped during this stage.

The card should fire an interrupt when a state is reached that matches that of the interrupt mask configured during the initialisation. When the interrupt reaches the driver it should acknowledge the interrupt to prevent it from firing again immediately and depending on the cause of the interrupt carry out the required operation.

For the send operation of the network driver the data to actually be sent should be sent over the data channel and passed through a serializer so that the driver can just handle ethernet packets. There is an implementation decision at this point to be made to decide if it is worthwhile validating that the data to be sent onto the network has a valid ethernet header.

When a packet is passed to the driver to be sent the driver is required to place it into a transmit queue that the network card can access, updating any headers stored within this structure as required. Once this has been completed then if the card is idle then it should be notified that there is more data that can be handled. The card should update this descriptor list showing when the packet has been transferred from host memory, and the memory are can be reused. Also depending on the interrupt mask set and the specific card used, an interrupt may be fired when the card has read a descriptor in the transmit queue.

The receive call is very similar to the send other than the data transfer is in the opposing direction. When a packet is received by the card from the network it should transfer the data into host memory. Using the interrupt mechanisms already described the driver should recognise that there has been data transferred into the receive queue. The driver should then move the data from the receive queue and send it over the data pipe to the network stack using the serializer. Once this has completed then the receive descriptor should be marked to show that it is available for reuse.

As a further note some network cards support offloading checksum calculations into hardware freeing up cpu cycles on the host machine. Utilizing this would depend on support from the network stack to function correctly, as such the design will not stipulate that this must be used when available.
Chapter 4

Example Implementation

This section will cover some of the problems that were faced when writing the example implementations for PCI bus access and a Network Driver. Due to the system being part of an ongoing research project the system is not completes. As such some of the decisions that were made during the course of this project may not hold true in the future and would likely require partially reimplementing.

4.1 PCI

The PCI access chosen at this stage was a result of dynamic module loading not being fully implemented and therefore need hard coding in most situations. As such at this stage in the systems development, and knowing that all devices would need to be plugged in on boot to be correctly and mapped by the BIOS it was decided that only the functionality required for the operation of the network driver would be implemented.

There are two main methods that are used when accessing the PCI bus, the I/O ports interface provided by the x86 processor and memory mapped access. Due to only a single module currently using the bus and there being better documentation carrying out the interactions using the I/O ports on both the OSDev wiki[1] and in the Indespensible PC Hardware Book[4] this approach was taken.

The PCI search behaves in a similar way to the standard library function strtok where it takes an intial parameter on what to search for, and returns the first item that matches, successive calls to the function return the next item matching the search. Once no further items are found then it returns a values that is defined to signify that. This does cause a race condition, but in a more complete system this function call would be wrapped within the system service layer which would handle concurrency.

The search itself was implemented using a series of nested loops, returning if a device was found.
4.2 Network Driver

The network driver had more problems, the initial network card that was selected to write the example driver for was later found to be unavailable for purchase, as such a newer card was selected that could be purchased for testing on a physical device. This resulted in a reasonable amount of work becoming redundant due to the switch between a card that could only handle Fast Ethernet, to one that could handle Gigabit Ethernet, resulting in redesign of the driver.

The functionality that was achieved was the detection of the card, initialisation and access to the statistical registers. This was handled before sending and receiving were as access to the statistics assisted in some debugging of memory issues that were discovered. As the card requires aligned continuous and contiguous memory for both the receive and send buffers and there was no memory manager implemented that could handle that the decision was made to handle it outside the driver as the wider system should provide that capability.
Chapter 5

Testing

This section will describe the test equipment and carry out unit tests on features of the example implementations.

5.1 Hardware Specifications

The testbed consisted of four MSI C847MS-E33 Micro-ATX Motherboard with Intel Celeron NM70 CPU and 4GB DDR3 RAM each. In each of these a Intel EXPI9301CTBLK Gigabit Pro PCI Express Gigabit Network Card[2] was plugged into a known PCI port. These were booted using PXE Boot using the onboard network card. Each node could be remotely rebooted over ssh and a serial connection was accessible through telnet.

The Intel network cards were all attached to a quiet network using a Netgear ProSafe 8-port Gigabit Desktop Switch 10/100/1000 Mbps (GS108). This switch is rated as follows.

Bandwidth: 16Gbps (non-blocking)
Forwarding rate
10 Mbs - 14,880 packets/sec
100 Mbps - 148,800
1000 Mbps - 1,488,000 packets
Latency (1500 byte packets)

<table>
<thead>
<tr>
<th>Network Connection</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Mbs</td>
<td>30 us (max)</td>
</tr>
<tr>
<td>100 Mbs</td>
<td>6 us (max)</td>
</tr>
<tr>
<td>1000 Mbs</td>
<td>4 us (max)</td>
</tr>
</tbody>
</table>

Mac Address database 4K

5.2 PCI

Detect a device on the PCI bus.
Search for an Ethernet Device
Search for a second Ethernet Device
Search for another type of device (PCI Bridge)

Paging enabled! (439 active pages)

Heap frames:
Pooling 524288B of memory as a heap. Headers are 16B
Size: 524288B Memory Map: -524288B ->
Stack base: @0000167010

Scanning the PCI bus...
[00:00:00] Unrecognised device from 'Intel Corporation'
  Class: 6, Subclass: 0, Program interface: 0
  BAR->[ 0, 0, 0, 0 ]
[00:02:00] Unrecognised device from 'Intel Corporation'
  Class: 3, Subclass: 0, Program interface: 0
  BAR->[ 325504, 0, 0, 0 ]
[00:16:00] Unrecognised device from 'Intel Corporation'
  Class: 7, Subclass: 80, Program interface: 0
  BAR->[ -1610287136, 0, 0, 0 ]
[00:1a:00] Unrecognised device from 'Intel Corporation'
  Class: c, Subclass: 3, Program interface: 20
  BAR->[ -2147420192, 0, 0, 0 ]
[00:1b:00] Unrecognised device from 'Intel Corporation'
  Class: 4, Subclass: 3, Program interface: 0
  BAR->[ 325600, 0, 0, 0 ]
[00:1c:00] Unrecognised device from 'Intel Corporation'
  Class: 6, Subclass: 4, Program interface: 0
  BAR->[ 0, 0, 0, 0 ]
[00:1c:01] Unrecognised device from 'Intel Corporation'
  Class: 6, Subclass: 4, Program interface: 0
  BAR->[ 0, 0, 0, 0 ]
[00:1c:02] Unrecognised device from 'Intel Corporation'
  Class: 6, Subclass: 4, Program interface: 0
  BAR->[ 0, 0, 0, 0 ]
[00:1c:03] 6:4:1
  Intel - Hub Interface to PCI Bridge, rev. 32902 [0x2448:0x00c4]
  BAR->[ 0, 0, 0, 0 ]
[00:1d:00] Unrecognised device from 'Intel Corporation'
  Class: c, Subclass: 3, Program interface: 20
  BAR->[ 1879111648, 0, 0, 0 ]
[00:1f:00] Unrecognised device from 'Intel Corporation'
  Class: 6, Subclass: 1, Program interface: 0
  BAR->[ 0, 0, 0, 0 ]
[00:1f:02] Unrecognised device from 'Intel Corporation'
  Class: 1, Subclass: 6, Program interface: 1
  BAR->[ -256835584, -257884160, 0, 0 ]
[00:1f:03] Unrecognised device from 'Intel Corporation'
  Class: c, Subclass: 5, Program interface: 0
  BAR->[ 1342502880, 0, 0, 0 ]
[02:00:00] 2:0:0
Listing 5.1: Result of dumping the PCI devices on the bus and carrying out each test

The tests are successful. The both network cards detected are at the correct position on the PCI bus. Also finding the second type of device was also successful.

5.3 Network Driver
Chapter 6

Evaluation and Conclusions

The work that was carried out was only partially successful as the Network driver was not fully implemented. However pending further work there was no reason discovered that the designs proposed would not work.

Future work that needs to be carried out includes finishing the implementation of the network driver, looking into an implementation of a Serializer and adding capabilities for concurrency into the PCI module in particular.
Bibliography


Appendix A

Proposal: Operating System Device Driver
A.1 Abstract

Network drivers are vitally important for the modern operating system due to the interconnected nature of computing today. In addition to this the standardization of such communications is and will remain a major undertaking. This project aims to provide a working implementation of a driver for a network card, in addition to an implementation of at least the bottom two layers of the OSI standard networking model, with a subsection of the third layer provided time permitting. The project is split into 3 logical stages, the initial research and design stage, the implementation stage, and the testing and evaluation stage. These phases will be carried out in order, though there will be some overlap between them, especially in the implementation, and the testing and evaluation stage. By nature of the project being a practical based one the expected outcome is the driver and a portion of the networking stack, with performance testing carried out on the implementation and provided in a final report.

A.2 Introduction

This project's primary aim is to provide an implementation of a network driver for the i8255X chipset with an additional goal to provide an implementation of the network layer of the OSI model. The driver will be able to detect and initialise the network card and capable of sending and receiving data over the network. The network layer implementation will mainly be provided to simplify the testing process with ICMP used for a basic ‘ping’ implementation which is able to be used in a large portion of the testing. Additional features of the network layer will be implemented dependant on time constraints.

A.3 Background

The Operating System that this project is based on is able to scale through being distributed to additional nodes therefore bypassing the issues currently being experienced by more traditional operating systems, which are generally localised to one physical machine. This approach benefits more from having faster and faster processors rather than having multiple processing units to split the workload over. While development of these operating systems has resulted in them being able to utilize to a greater extent multiple processing cores they still suffer over a system that was designed from the beginning to run in this type of environment. With any computer system, it has become more common that connectivity to a network, generally the internet, is required. With a system that by its very nature is interconnected this requirement becomes more important. By using an implementation of IPC (Inter-Process Communication) it is possible to communicate between different processes on either the same or a remote machine. As such to avoid needing to reimplement the networking stack for each specific network card a device driver is instead used which must provide a common interface to the card to enable the rest of the system
to have no knowledge of any specific cards, but instead use the standard interface provided by the device drivers.

A.4 The Proposed Project

A.4.1 Aims and objectives

The primary aim of the project is to implement a device driver for the intel ethernet card based on the i8255X chipset and in particular the i82557 card. This specific card was selected as it is the base for the rest of the i8255X chipset family and as therefore allows more physical cards to be used with the same driver.

- Detect and initialize the card.
- Be able to send and receive packets of data from the network card.
- Have a working implementation of the Networking stack up to the MAC layer on the card
- Either implement or port an existing implementation of the IP layer of the networking stack
- Implement basic powersaving, ie an 'in use' and a 'powersave' state

A.4.2 Methodology

Implementation Plan

The expected process of implementation is to firstly be able to recognise and initialise the driver using the class codes for the chipset. At this point the card will be in a state that can be usable, and an interrupt vector defined for when data is recieved. At this point a basic version of sending and recieved packets through the card will be implemented. At this point interfaces that are required by the higher levels of the networking stack will be provided whilst implementing driver components as required by each interface. At this point the power management capabilities of the cards will be examined, with at least an 'in use' and a 'powersaving' mode implemented, others will be examined time permitting. Whist this is progressing the efficiency of the driver will be continuously measured with the intention to reduce the latency added by the driver in the network link.

Once the driver is functional to assist in the testing process an version of the IP layer of the networking stack will be provided. This implementation will either be a port of an existing version altered to fit in with the codebase of the Operating System and the network driver, or a reimplementation of features required for the testing process.
Testing Plan

The testing will be continuous, with more detailed testing of functionality after any major changes to the codebase to prevent any regressions. The basic testing will include tests to ensure that the cards can still be detected and initialised, data can be sent and received. Performance testing will be carried out upon reaching milestones, and on any major codebase changes. The performance testing will include simple round trip timing with separate packets when the card is under light or no load, and stress testing of the send and receive queues. The stress testing will be carried out in 'send only', 'receive only', and 'send and receive' to ensure that the interaction of sending and receiving don’t interfere with each other.

The latency testing is planned to be carried out using timestamps sending packets over the network, the actual timing will be both over the network and the time taken for the driver to process the data. The actual data will be recorded in a text file with identifiers for later processing to reduce the impact on the network. The tests will be carried out in two ways, firstly one packet every few seconds to simulate a quiet network link, and bursts of multiple packets at once to simulate a busy network link. The actual number of packets to be used will be determined when the testing is taking place and a range of values will be used to check what occurs when the FIFO queues on the card get full.

A.5 Programme of Work

Here follows the expected program of work, with an overview of what is expected to be contained within each section. A gantt chart with the anticipated timings of the project is contained in Figure 1.

1. **Setup the development environment** - Includes gaining access to the source code, configuring the build environment and doing some initial testing of what is already implemented in the research OS.

2. **Research the network card specifications** - Look into the capabilities of the card, the interfaces that are provided through the PCI link and look for any potential problems that could arise using the card.

3. **Research other implementations of the driver** - As there are other open source implementations of the network driver it is likely that any difficulties that arise when dealing with the network card have already been addressed, as such there is little need to develop new solutions to problems that have already been solved.

4. **Design the proposed solution** - Outline the proposed solution and required data structures.

5. **Development of driver** - The driver will be implemented using C++ and potentially some C or x86 Assembly.
6. **Development of a portion of the Networking Stack** - Examine other implementations of the IP stack and look to see if it is feasible to port existing implementations. Again implemented in mostly C++.

7. **Testing of each stage in the driver development** - Continuous testing of various areas of the driver code to prevent any regressions in functionality.

8. **Detailed testing of each driver component** - Final unit and performance testing to ensure that all objectives have been completed.

9. **Finish writing the Report**

### A.6 Resources Required

- Specification of the network card
- Qemu to use for an emulated environment to perform initial testing on.
- One or more physical network cards based on the correct chipset to use as part of a physical testbed. Required to a greater extent later on in the project.
- Physical hardware for use as a base for the testbed.

### A.7 References

- Intel Open Source Developer Manual for the i8255X chipset
  Revision 1.3 January 2006
- [http://wiki.osdev.org](http://wiki.osdev.org)
- The Indispensible PC Hardware Book - 4th Ed
  Section 3 - Internet Layer Protocols
Setup the development environment
Research the network card specifications
Research other implementations of the driver
Design the proposed solution
Development of driver
Development of a portion of the Networking Stack
Testing of each stage in the driver development
Detailed testing of each driver component
Finish writing the Report

Figure A.1: Gantt Chart