Operating Systems Engineering
Assignment 3: Q2 & Q3

R00159222 - Zachary Dair
zachary.dair@mycit.ie

This document provides answers to the theory questions, **Q2** and **Q3**

Question 2:

The user is able to send and receive packets to/from the **E1000** device with system calls such as **read** and **write** in the **XV6** operating system. This is due to **Unix**, which **XV6** is based on, providing an almost universal representation of certain resources. Resources such as devices, pipes, files and sockets all of which are represented by a similar structure in the form of **files**. Therefore enabling the usage of standard file descriptors to access these resources.

When a user is attempting to send packets in XV6 using E1000, a **connect** function is used as a replacement for the typical **open** function when writing to a file. The **connect** function requires the parameters: destination address, source port and destination port and will return a file descriptor which can later be used when writing. The usage of the **connect** function can be seen in the user program **nettests**, specifically line 34 of **nettests.c**.

The returned file descriptor from the **connect** function can be passed as a parameter to the typical **write** function. A function which has previously been used to write the contents of a buffer to a file, such as in **cp.c**. In this case the file descriptor can be considered as a connection to the socket to which the user can conduct **read** and **write** operations. As seen on line 40 of **nettests.c** the user is able to call the **write** function in order to send data through the socket, similarly the user can receive data by using the **read** function as seen on line 47 both of which use their associated system calls.

When the connect function is used, it will result in the creation of both a file struct in file.h but as a socket type, and additionally a socket struct which is allocated using the sockalloc function found in sysnet.c. This socket struct will contain the same parameters as passed in the user level (destination address, source port, destination port), this new socket will then be stored into a queue of sockets to be processed.

The **write** function will result in it's associated system call being utilised and will lead to the OS running the **filewrite** function. Inside **filewrite** the type of our file struct is evaluated, if the file is a socket the **sockwrite** function is called. Inside the **sockwrite** function a new buffer is allocated and the data to send is added into the buffer, the associated header is then added to the top of the buffer. Once the header is added the ip layer information can be added to the buffer and finally the ethernet layer can be added. At this stage the packet is now ready to be sent which is accomplished using the **e1000_transmit** function.

Alternatively when the user is attempting to receive packets and the **read** function is called, the associated system call is utilised leading to the consequent running of the **fileread** function which then evaluates the type of file struct as did the **filewrite** function. If the type is a socket, the **sockread** function is called which checks the contents of the buffer, ensuring it’s not empty, if it is empty the process will sleep and wait for more data. However if there is a packet to read in the queue of packets it will be removed and returned to be utilised by the user level code.

Question 3:

The open source virtual machine monitoring technology **Firecracker** can be used to create and run microVMs. These microVMs differ from a standard VM as they are more lightweight and have a degree of non-essential functionality stripped out in order to provide more efficient isolated deployment environments.

The user is able to create a number of processes each containing one microVM, the number of processes and therefore microVMs is limited to the available hardware resources. Each process provides three threads, an **API**, **VMM** and **vCPUs**. The **API** provides communication capabilities between the Client and **Firecracker’s** API. The **VMM** thread handles the device emulation, such as the emulation of the **VirtIO** devices: **Net**, **Block** and **Vsock**. Additionally there is a **vCPU** thread which handles synchronous operations.

VM Sockets or more commonly **Vsock** is a method of communication between a host machine and guest virtual machine(s), in **Firecracker** the **Vsock** device is responsible for providing **VirtIO-Vsock** support. **VirtIO-Vsock** is a communications device that enables communication between the host and guest VM. This communication takes place through **AF_UNIX** and **AF_VSOCK** sockets, belonging to the host and guest respectively. Before communication between these sockets can take place a path to the **AF_UNIX** port must be defined for the **VirtIO-Vsock** device.

Firecracker’s VirtIO-Vsock implementation enables two forms of communication, the first where the connection is initiated by the host, the second where the guest initiates the connection.

In the case of a host initiated connection, the guest will create an **AF_VSOCK** socket and will begin listening on a specified port, subsequently the host will connect to the **AF_UNIX** socket and sends a **CONNECT** message, in pure text, which contains the port number for the connection. Upon receiving this **CONNECT** message the guest will accept the connection, therefore providing the desired communication channel. To finalise the connection an acknowledgement message is sent back to the host. Following this procedure **Firecracker** will attach the connection to the software on the guest side that required the connection in the first place.

Alternatively in the case of the guest initiated connection, the host will listen on the specified port using the **AF_UNIX** socket, subsequently the guest will create an **AF_VSOCK** socket and connect to the host content id of (**Host_CID**) using a specified port. The host will then accept the new connection, therefore establishing the communication channel.

Firecracker provides a number of benefits by enabling the user to use **VirtIO-Vsock** such as multiple clients connecting through one listen socket, no address configuration required for the guest, a reduced attack services as there is no ethernet or TCP/IP and additionally **VirtIO-Vsock** can provide communication channels to VMs with no network interfaces.