Adding Networking Support to CertiKOS

Kevin Abbott
5/11/2016

Abstract

This project was to add basic networking support to the CertiKOS kernel. It involved implementing a driver for the e1000 network adapter, porting the functionality of lwIP lightweight TCP/IP stack, and creating a simple test for receiving and sending UDP packets between the kernel and an arbitrary host. The framework is now in place to bring start bringing the whole network stack online.

1 Introduction

The FLINT research group at Yale is currently working on CertiKOS, which is a small, formally verified operating system kernel built in order to give provable guarantees on functional correctness and safety. The kernel is designed by breaking up a normally complex kernel system into many abstraction layers that can each be certified independently. In its current state, CertiKOS provides memory and process management and is able to support multiple programs running independently and securely. Since CertiKOS requires a provably correct specification, it is inherently limited in its size and scope of features. As a result, the goal of CertiKOS is to use it as a hypervisor and have traditional operating systems running on top of it. In this setup, users get the benefit of both provable safety and the full feature set of commercial operating systems.

One necessary feature currently missing from CertiKOS, though, is a full network stack. Once it is running a network stack, the kernel will be able to support a much broader range of applications and can function as a full-fledged hypervisor. The goal of this project was to begin the process of adding networking support to the CertiKOS kernel. This was achieved by implementing a slimmed-down network adapter driver, porting a lightweight TCP/IP stack, and writing a simple test application that showed the working functionality of the network stack. In its current state, the kernel is able to send and receive UDP packets from arbitrary hosts. While useful, there is still a lot of work left to be done to give CertiKOS full networking support, but this project built the initial functionality and framework needed to bring a full network stack online and began the steps of abstracting layers of the network hardware interface in order to be verified in the future.

The following sections go over the work done in this project to add networking support. Section 2 discusses the e1000 network adapter, how it is initialized, and how sending and receiving packets works. Section 3 outlines how networking is done within QEMU and how the provided debugging functionality is used. Section 4 goes over the major system calls added in order to support a user-level TCP/IP stack. Section 5 describes lwIP, the lightweight TCP/IP stack that was ported and used for this project. Section 6 outlines the implementation and functionality of the test application. Lastly, section 7 overviews the current state of the project and what work needs to be done in order to give CertiKOS fully functional, verified networking support.

This project was built using the resources of, and porting code from, the MIT 6.828 JOS course and available solutions as well as previous CPSC 422 courses[1][2][3][4].

2 E1000 Driver

The first part of the this project involved implementing a driver for a network device, specifically the e1000 network adapter. The e1000 NIC is a standard family of Intel network cards. While drivers for other network cards will need to be implemented in the future, this is a good starting point given the widespread usage of the e1000[5].

2.1 PCI Initialization

The first step to initializing the e1000 network adapter is to start up the device and setup a MMIO (memory mapped IO) region in order to read and write to the network adapter’s registers. This is handled in kern/dev/pci.c.
At initialization, the PCI Bus is scanned for devices to be mounted. Each device is matched to a specific device class in order for the memory region to be setup and then matched against a vendor and device id for the specific driver to be called. In the case of the e1000 NIC, it is matched as a bridge PCI in order for the function pci_bridge_attach to be called, which initiates the MMIO region. Next the vendor id and device id are matched to that of the e1000 device. Now the function e1000_attach in kern/dev/e1000.c is able to be called in order to fully initialize the network adapter.

### 2.2 E1000 Initialization

Now that the kernel has access to the e1000’s register via MMIO it is possible to initialize the device for transmitting and receiving packets. This MMIO address region is defined in the corresponding pci_func struct, which is created for every PCI device. In this struct, the reg_base[0] field defines the beginning of the MMIO region. In kern/dev/e1000.c, the value of e1000 is set as a pointer to the beginning of this region, which essentially makes it a pointer to the device’s registers.

Since the e1000 uses DMA (direct memory access) to get packets from the system to transmit and to pass received packets to the system, the next step is to initialize the transmit and receive buffers.

The global tx_desc_array is an array of tx_desc structs which contain information on packets in the transmit queue. Each structs addr field contains the address of the corresponding tx_pkt packet, which is held in the tx_pkt_bufs global. A tx_pkt is simply an array of the packet data. The initialization process for the transmit queue involves setting the addr fields in tx_desc_array to the corresponding element of tx_pkt_buf and then setting the status flags of each tx_desc is set to E1000_TXD_STAT_DD, indicating that there is no packet needing to be sent.

The initialization process for the receive queue is very similar. The rcv_desc elements of rcv_desc_array are the receive corollary to tx_desc. Upon initialization, the address field of each rcv_desc is set to the corresponding rcv_pkt element of the rcv_pktbufs global.

The next step is to set the appropriate registers of the e1000 to tell it about the transmit and receive queues, thus initializing the MMIO region. The Transmit Descriptor Base Address register (e1000[E1000_TDBAL]) is set to beginning address of the transmit queue and the Transmit Descriptor Length register (e1000[E1000_TDLEN]) is set to the length in bytes of the transmit queue. The Transmit Descriptor Head and Tail registers (e1000[E1000_TDH] and e1000[E1000_TDT]) are each set to 0. These registers are used to indicate the current head and tail locations of the transmit queue. For the receive queue, the Receive Descriptor Base Address register (e1000[E1000_RDBAL]) is set to the address of the queue and the Receive Descriptor Length register (e1000[E1000_RDLEN]) is set to the length in bytes of the queue. The Receive Descriptor Head (e1000[E1000_RDH]) is set to 0 and the Receive Descriptor Tail (e1000[E1000_RDT]) is set to the number of rcv_desc elements in the queue. This indicates that the device has control of the entire queue and there are no received packets waiting to be processed by the kernel.

The final step to initializing the e1000 NIC is enabling the device to transmit and receive packets. Enabling transmission requires setting the Transmit Control register (e1000[E1000_TCTL]) with the flags to enable transmit, pad short packets, and handle collision threshold and distance with full duplex communication (E1000_TCTL_EN, E1000_TCTL_PSP, E1000_TCTL_CT_DEFAULT, and E1000_TCTL_COLD_FULLD respectively). Lastly the Transmit Inter Packet Gap register (e1000[E1000_TIPG]) is set according to the default IEEE 802.3 Ethernet value (E1000_TIPG_IEEE8023_DEFAULT) for the transmit timer.

Enabling reception is a bit more involved. First the device MAC address must be read from the card’s EEPROM. This is done with the read_mac function and involves reading the 6 byte mac address two bytes at a time over the PCI bus. Once the mac address is read, the lower 4 bytes are loaded into the Receive Address Low register (e1000[E1000_RAL]) and the upper two bytes are loaded into the Receive Address High register (e1000[E1000_RAL]). The MAC address need to be explicitly set, since some systems use the address from the boot PROM instead of from the adapter’s EEPROM when setting up the filter for receiving packets. Once the
MAC address is loaded, the Receive Control register (e1000[E1000_RCTL]) can be set with the flags to enable receiving packets. This register is set with the receive enable flag, the bit size of the receive buffer, the Buffer Extension Size flag to indicate the receive buffer is larger than 2048 bytes, and the Strip Ethernet CRC so that the device removes the packets Ethernet checksum before DMA-ing the received packet (E1000_RCTL_EN, E1000_RCTL_BSIZE, E1000_RCTL_BSEX, and E1000_RCTL_SECRC respectively).

Now the e1000 is ready to transmit and receive packets. When receiving a packet the device will verify that the MAC address in the ethernet header matches the device's filter, and then put it in the next available rcv_pkt_bufs slot whose associated rcv_desc doesn’t have the DD flag (E1000_RXD_STAT_DD) of the status field set. The DD bits indicate whether a received packet in the buffer has been processed by the system yet. For transmission, the device will periodically check the head of the transmit queue and see if there is packet waiting without the DD bits in the tx_desc status field set, which indicates that the system wants the packet to be sent.

### 2.3 Transmitting Packets

With the e1000 fully initialized, it is fairly straightforward for the kernel to send packets. This is handled by the e1000_transmit function. Since the device itself handles incrementing the Transmit Descriptors and updating the status field with the DD flag, all the kernel has to do is check if the status field of the tx_desc at the tail of the transmit queue is set with the DD flag, indicating there is room on the queue for another packet to transmit. If the flag is set, the kernel moves the packet data into the corresponding tx_pkt, sets the packet length in the tx_desc, removes the DD flag of the status field, and sets the RS and EOP flags of the command field. The RS flag (E1000_TXD_CMD_RS) indicates that the device should report the status of the packet transmission. The EOP flag (E1000_TXD_CMD_EOP) indicates that this is the last tx_desc making up the packet. This is required since multiple descriptors can be used to make up a single packet. Lastly, the Transmit Descriptor Tail register is incremented.

### 2.4 Receiving Packets

The process for the kernel to receive is similarly straightforward. The kernel reads the index of the next rcv_desc and forces the corresponding descriptor to be loaded from memory. If the DD flag of the status field is set, then the descriptor points to a packet ready to be read, else there is no packet waiting to be read. The kernel then moves the packet data into a buffer and removes the DD and EOP flags from the status field, thus allowing the descriptor to be reused. Lastly, the Receive Descriptor Tail register is incremented.

### 3 Networking in QEMU

With the e1000 driver implemented, it is able to be tested. The CertiKOS kernel used for this project runs within the QEMU hypervisor, which provides a virtualized e1000 NIC [7]. This virtualized adapter is brought up simply by adding the following flag to the QEMU command in the main Makefile.

```
-net nic,model=e1000, macaddr=52:54:00:AA:13:37
```

This is used in conjunction with the -net user flag which allows for user level networking. While this limits the feature set, it allows the guest (CertiKOS) to do networking without host root privileges.

On top of providing a virtualized NIC, QEMU creates a private network between the guest and the host network. The gateway to the host network is at 10.0.2.2, the local DNS is at 10.0.2.3, and guest devices start at 10.0.2.15. Packets arriving at the host machine destined for our kernel go through a network address translation via QEMU. In order for this to occur, the ports required by the kernel must be mapped to corresponding host ports. This is done with the following command.

```
-redir [TCP/UDP]:Host_Port::Guest_Port
```

Here the port redirection is defined for either TCP or UDP packets and is a mapping from the Host_Port to the Guest_Port. In the current setup, there is a UDP mapping from a host port to port 7, and TCP mappings from host ports to port 7 and port 80.

Since the e1000 device in QEMU is fully virtualized, it is simple to log the network traffic. Adding the -net dump,file=qemu.pcap flag to the QEMU command prints all the packet communication in qemu.pcap. This can then be viewed using the following command.

```
TCPdump -XXnr qemu.pcap
```

Lastly, by adding the a number of flags to the make qemu command, it is possible to see debugging output of the e1000. The useful flags are TX to log packet transmissions, TXERR to log transmit errors,
RX to log received packets, RXFILTER to log packets matching a MAC filter, and RXERR to log receive errors. They are used as follows.

make qemu E1000_DEBUG=TX,TXERR,RX,

4 System Calls

For this project, the majority of the network stack sits in user space. As a result, before any program can utilize the e1000 to send and receive packets, a number of system calls needed to be added. There were also a number of small changes made to the existing system calls, especially to the IPC calls in order to change their blocking/non-blocking nature.

4.1 Timer

The simplest system call to add was a function to get the current time in milliseconds, which can be used for creating timeouts in the network stack. In the kernel, this functionality utilizes the Lapic timer to keep track of the current time. Whenever there is a timer interrupt, a counter in kern/dec/time.c is incremented. Thus when a user program makes a call to SYS_time_msec, the value of this counter times the number of milliseconds a Lapic interrupt represents is returned.

4.2 Memory Allocation

The network stack also requires memory allocation at a more fine-grained level than a page. As a result, a simple malloc/free functionality was added. While this could have been implemented in user space and have merely relied on a system call to allocate pages, having it in the kernel allows for added functionality, especially with regards to shared memory between programs.

4.2.1 Malloc

The system call SYS_malloc, implemented in kern/lib/malloc.c, takes as a parameter the size in bytes of the amount of memory to be allocated and returns the virtual address of the malloced space.

Internally, the kernel scans through the user-level program’s virtual address space looking for an empty block large enough for the desired malloc. For every process, the kernel keeps track of the current location of where to look. If this location is the start of a page, which it is at start up, the system simply scans through the address space by page size looking for an available page. If it finds one, the kernel allocates the page to the process. Since there can now be multiple references to a page, the system keeps track of the number of references in the last 4 bytes of a page. When a page is first allocated, this reference is set to 2 to account for the reference used by the mallocing process and the current reference within the malloc function. This reference counter ensures that when free is called, the page is only deallocated if there are no remaining references. On subsequent calls to malloc, if the process’s current malloc pointer is in the middle of a page, the kernel checks whether the desired malloc size would fit within the rest of the page. If it would, the system increments the reference counter and returns the respective address. If it wouldn’t, the kernel frees its pointer to the current page and starts looking for a new free page.

One last functionality of malloc is that it can allocate memory blocks larger than the page size. To do this, it utilizes the PTE_CONTINUED permission flag in the page table entry, which indicates that the malloc continues on to the next logical page.

4.2.2 free

The system call SYS_free takes an address of a malloced block to be freed. When free is called on an address, the system first checks if the malloced block spans multiple pages. If it does, it continues to free pages until it finds one without the PTE_CONTINUED flag. When it finds this last page, the system decrements the reference counter at the end of the page. If this reference counter drop to zero, the page no longer has any active reference and can safely be deallocated.

4.3 Net Send and Receive

The most important system calls added were those to utilize the transmit and receive functionality of the e1000 driver. These are the SYS_net_transmit and SYS_netReceive calls, which transmit and receive packets respectively.

When a program calls SYS_net_transmit, which takes a memory address and length, the kernel copies the data into its buffer and then makes the call to e1000_transmit. There is no direct memory access between the e1000 and memory allocated to user level. The kernel then returns the error status of the packet transmission.

When a program calls SYS_net_receive, which takes an address of an allocated page, the kernel calls e1000_receive and, if there is a new packet, copies the data into the user program’s buffer.
5 lwIP

With a functioning e1000 driver and implementations of the necessary system calls, it is now possible to bring up a complete network stack. This project utilizes lwIP, which is a lightweight TCP/IP implementation[6]. It is the main network stack used in the JOS assignments.

While lwIP provides a complete implementation of TCP/IP and related protocols, it requires a lot of changes to get it working within the CertiKOS framework. The service was built with a lot of assumptions about shared memory and the types of functions user-level programs can utilize. Many of these assumptions violate the CertiKOS requirements. As a result, a significant portion of this project involved appropriately porting lwIP and making it work within CertiKOS.

At this point, most of the lwIP functionality exists in CertiKOS and can be found in user/net/. The next step is to bit by bit bring the full stack online. This needs to be done slowly and carefully in order to verify the functionality of each piece. While lwIP is a relatively lightweight TCP/IP implementation, it is still very complex and has a lot of places for bugs.

6 User Application

The final part of this project involved building a simple network application in order to test the e1000 driver and the small subset of lwIP that was fully brought online. The application built for this project, user/EchoTest/EchoTest.c, simply receives a UDP packet from an external source and echoes it back. The implementation of this functionality, though, is fairly involved.

In order for CertiKOS to make any connection with the host network it has to inform the QEMU gateway that it is online and of its IP address, while also learning the gateway’s MAC address. It does this through an initial ARP, address resolution protocol, announcement at startup. This announcement is a broadcast on the QEMU private network that the the guest (CertiKOS) at a specified IP address and MAC Address is looking for the gateway (10.0.2.2). When the gateway receives this packet it knows the guest is online and therefore can start forwarding packets as they are received. It responds with a message indicating which MAC address the gateway IP is at. From this point on the kernel never has to worry about how packets are transferred to and from the host network.

At this point, the user application simply waits for a packet to be received. When it does, it casts the packet as a header followed by the message data, and then uses the header information to form and send a return echo packet. A packet header includes an Ethernet header, an IP header, and either a TCP or UDP header. Since the current application only supports UDP, if it receives a TCP packet it simply ignores it. Creating the reply packet is relatively straightforward.

Any Ethernet header, which is just a source and destination MAC address, processed by the kernel can only take one of two forms. This is because the QEMU gateway is always the next hop for the system. All incoming packets have the gateway MAC as the source and the e1000 MAC as the destination. All outgoing packets need to have the e1000 MAC as the source and the gateway MAC as the destination.

The IP and UDP headers are similarly straightforward. In the IP header, the source and destination address need to be flipped; and in the UDP header, the source and destination ports need to be flipped. In addition both of the checksums in the IP and UDP headers are recalculated. Since these checksums are a result of a bunch of bitwise addition, the echo application doesn’t actually need new checksums as they will just be the same. Having this functionality in place, though, makes it trivial to extend this test application.

7 Future Work

In its current state, this project has added a fully working e1000 driver and basic UDP packet support to CertiKOS. While it is now possible to do some basic networking, a lot of functionality still needs to be added to create a fully working network stack.

The majority of this work involves bringing more of the TCP/IP stack online as well as moving more of it into kernel space. Since the protocols of the network stack are inherently designed as layers of abstraction, it is relatively straightforward how to iterate through and add functionality to the kernel. As of now, only the driver sits in kernel space and the rest sits in user space. The next step is to bring the link layer of the network stack into the kernel. This layer would be responsible for adding and removing the Ethernet header from packets as well as multiplexing/demultiplexing packets between IP layer instances, and potentially even between different networking devices. The next step up would be to bring the TCP/IP stack into the kernel. This stack includes layers for handling the IP, TCP, and
UDP headers; creating the socket interface; and multiplexing/demultiplexing packets between the network and the user application using the socket interface. Besides this core functionality, there is also the need for adding full ARP support, so the kernel can do networking outside QEMU; as well as adding other protocols like DHCP and DNS. CertiKOS can do networking without these, but they are necessary protocols to make it a fully functioning network stack.

Beyond just adding necessary network stack functionality, a number of other functions need to be added to make the current driver fully usable and efficient. This includes adding support for interrupts, so that the kernel can learn when a new packet is received or when a packet is transmitted and there is room in the transmit queue.

7.1 Verification

Before bringing the rest of the network stack into the kernel and adding a lot of complex functionality, the first step is to verify the e1000 driver. In its current state, the e1000 device is abstracted as a small component with a simple send/receive interface. By having it this simple, it should be possible to do formal verification. The driver can even further be abstracted into PCI initialization, e1000 initialization, and packet transmit and packet receive layers. As more portions of the network stack are brought online, it will be possible to extend the functionality of the formally verified components.

7.2 CPSC 422 Assignment

The last major requirement for this project is the consideration for how to add it as an assignment for CPSC 422. There are a number of directions this can go. One aspect could be to implement a portion of the e1000 driver. Overall, the e1000 device is very complex and implementing the driver required a lot of research and debugging. As a result, making the whole driver an assignment would probably not be very productive to students; but at least a portion of it should be worked with, since learning about device DMA and MMIO would be useful. Another aspect of a networking assignment could be to implement a test program similar to EchoTest. Given the functionality now available in lwIP and the various system calls, it is straightforward to write an echo application, or even a more complex application once TCP is working. There is a potential for creating an open ended portion of the assignment as the network stack could be chained with other components of CertiKOS to do things like an HTTP server. Any assignment, though, should be focused on understanding how network application and the TCP/IP layers interact with hardware and other parts of the kernel.

References


