Chapter 4. Implementation of PIN/XINU

In order to make the courseware, we develop the system by porting the source code of TCP/IP in Xinu [10] to a user-level program which runs on Linux. In order to achieving above aim, we use message passing mechanism to simulate the behavior of packet transmission in network. Besides, we also need to modify the system calls which are used in TCP/IP protocol of Xinu [10]. Furthermore, we design a network topology simulator, which could construct a virtual network topology environment to simulate a real-world of network topology. The followings we briefly describe the parts which we had to modify and illustrate how to modify it, and explain the network topology simulator.

4.1. The Process Concept and Synchronization

Xinu provides several system calls that are needed for implementing TCP/IP protocols. The process concept and synchronization is one kind of system calls. The followings we describe the process concept and synchronization part which we modified.

4.1.1. Process Creation and Termination

Xinu provides the system call, *create*, to create a new process, the following code can create a new process.

procid = create(arguments); /* create a new process */

To terminate an existing process, a program calls the system call, kill, to terminate a process. The following code can terminate the process which identifier is the same as the argument.

kill(argument); /* terminate a process */

In PIN/XINU, we need rewrite process creation and termination. We used thread to replace process, Linux provides a thread library, called *Pthreads*, which is used for creating and managing threads.

4.1.2. Process Synchronization

Because each process in Xinu [10] runs independently, process synchronization must be considered. Xinu [10] provides three mechanisms to achieve process synchronization: *semaphores*, *ports*, and *message passing*. The following briefly illustrate the three mechanisms.

4.1.2.1.Semaphore

Xinu provides a system call, *screate,* which is invoked to create a semaphore. The following code can create a semaphore.

semid = screate(3); /* create a semaphore, specifying count three */

screate may return a semaphore identifier, when processes want to manipulate the semaphore count, processes can call the system calls, wait and signal. When a process calls the system call, *wait*, Xinu [10] decrement the semaphore's count by 1. If the count becomes negative, Xinu [10] blocks the process. To the contrary, when a process calls the system call, signal, Xinu [10] increase the semaphore's count by 1. If there are some processes which are blocked, Xinu [10] unblocks one of these processes. The following is an example to create a mutual exclusion semaphore and to call wait and signal to protected critical code:

```
s = screate(1);          /* create a mutual exclusion semaphore */
…
wait(s);                 /* call wait to protect the critical code */
… critical code …
signal(s);               /* call signal after finishing the critical code*/
```

In addition to providing mutual exclusion, courting semaphore is also used to provide synchronization for queue access in TCP/IP protocol. For example, IP datagrams are created by allocating a queue with finite size of block. When the queue is full, if a program wants to insert an item to the queue's block, it must be blocked until the queue has a free block. To the contrary, when the queue is empty, if a program wants to extract an item from the queue, it also must be blocked until an item existing in the queue.

In Linux, there are some system calls to create semaphores and control it, we use these system to replace above functions. In PIN/XINU, we use the system call, *semget*, to create semaphores, and use the system calls, semop and semctl, to manage and control the semaphores.

4.1.2.2.Ports

Another synchronization mechanism is ports. Ports provide an area which processes can pass data. The use of a port is like a finite queue of messages plus two semaphores, process creates a port by calling the function, pcreate, the following is an example code.

```
portid = pcreate(count);      /* create a port with space for up to $count
```

messages */

processes can call the system calls, *psend* and *preceive*, to store or remove items, such as the following code.

```
psend(portid, message);        /* send a message to a port */
message = preceive(port);    /* extract next message from port */
```

By the way, if the port is full, using *psend* will block the calling process until another process calls *preceive*, and if the port is empty, using *preceive* will block the calling process until another process calls *psend*.

In order to allowing processes to determine whether the port is full, Xinu [10] provide a system call, *pcount*. Firstly, *pcount* check the identifier of a port, and then retrun the current count of items in the port if the identifier of a port is exist. The following is the sample of *pcount*.

```
n = pcount(portid);                /* check current count of items in the port */
```

4.1.2.3.Message Passing

The third synchronization mechanism is *message passing*, that allows one process to send a message directly to another. When process want send a message to another process, it invoke the system call, *send*, to send a message directly. The following is the code of send.

```
send(message, pid);        /* send integer message to process pid */
```

A process can invoke the system call, *receive*, to wait for a message to arrive.

```
message = receive();      /* wait for message to arrive
```

In Xinu [10], *send* system call always proceeds, but *receive* system call will blocks the caller until a message arrives. In order to avoiding the infinite or long time waiting, Xinu [10] provide another system call, *recvtim*, which allow the caller to specify a waiting time for receive a message, if no message arrives within the specified time, *recvtim* return a special value, TIMEOUT, which mean fail to receive a message.

message = recvtim(50); /* wait 5 seconds (50 tenths of a second) for a message to arrive */

4.2. Network Interface Device Driver

In Xinu [10], when a packet arrived in network interface, the corresponding network interface interrupt is awakened, that causes the CPU suspend the running process, and then, CPU jump to the device driver code and execute it. In addition, the device driver also invokes protocol software to process the packet.

Due to the device driver is hidden under the I/O interface, processes can easily call device driver to get or send packets. The following is an example of sending a packet to an Ethernet interface.

write(eth0, buffer, length); /*write one packet to Ethernet 0 */

where *eth0* is a device descriptor which identifies a particular Ethernet interface device, *buffer* gives the address of a buffer that contains the frame to be sent, and *length* is the length of the frame measured in octects.

Incidentally, Xinu [10] provides a device abstraction to correspond to physical hardware. A process obeys the open-read-write-close paradigm to access all physical devices, including network interface. The followings briefly illustrate the details of device abstraction.

4.2.1. Device Abstraction

Xinu merges services and files into devices and uses a device paradigm for all input and output operations, including communication between an application program and protocol software. Device paradigm provides a *device abstraction*, which defines a set of devices to correspond to peripheral I/O hardware devices. In Xinu [10], if an application program uses an abstract device that corresponds to physical hardware or to a service, it follows the *open-read-write-close* paradigm to use it. The *open* function has three arguments:

d = open (device, name, other)

The first argument is a device identifier which specifies the device we want to use, and the second argument specifies the name of an object associated with that device. The meaning of the third argument depends on the device being opened. *Open* returns a device descriptor to be used for accessing the specified object.

When an application program wants to transfer data from or to the device which has been created, it uses the *read* and *write* functions. The following is an example of read a device.

len = read(device, buffer, buflen);

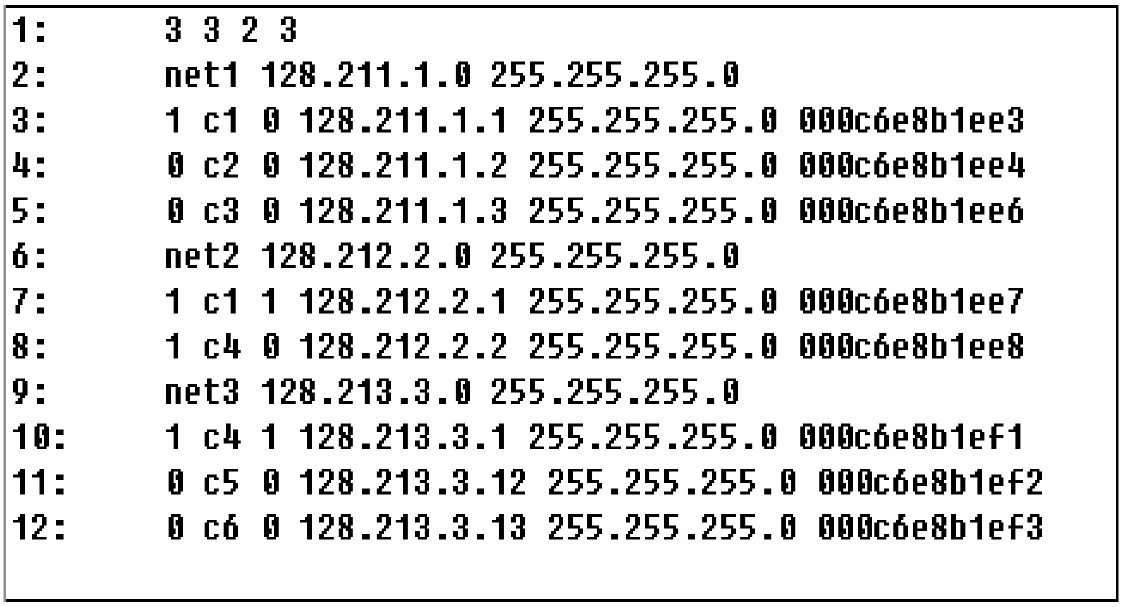
The following is an example of write a device.

status = write(device, buffer, buflen);

Both two functions take three argument, the first argument is a device identifier for the object, and the second argument point the address of a buffer, and the third argument specifies the length of the buffer in octects.

4.3. Network Topology Simulator

Besides modify modules given above, we also design a network topology simulator to simulate network topology. Network topology simulator builds a virtual network topology environment according the *network topology description* file. The network topology description file defines a network topology format, students can easily write the description file to build a virtual network topology for running TCP/IP protocol according the format. Figure 4.1 shows an example of network topology description file.



```
1:      3 3 2 3
2:      net1 128.211.1.0 255.255.255.0
3:      1 c1 0 128.211.1.1 255.255.255.0 000c6e8b1ee3
4:      0 c2 0 128.211.1.2 255.255.255.0 000c6e8b1ee4
5:      0 c3 0 128.211.1.3 255.255.255.0 000c6e8b1ee6
6:      net2 128.212.2.0 255.255.255.0
7:      1 c1 1 128.212.2.1 255.255.255.0 000c6e8b1ee7
8:      1 c4 0 128.212.2.2 255.255.255.0 000c6e8b1ee8
9:      net3 128.213.3.0 255.255.255.0
10:     1 c4 1 128.213.3.1 255.255.255.0 000c6e8b1ef1
11:     0 c5 0 128.213.3.12 255.255.255.0 000c6e8b1ef2
12:     0 c6 0 128.213.3.13 255.255.255.0 000c6e8b1ef3
```

Figure 4.1 An example of network topology description file

Network topology description file is divide into two part, the first part defines how many nets in the network, and the second part defines how many network hosts in each nets and illustrate each network interfaces. In Figure 4.1, the description file has twelve lines. In line 1, there are four numbers, the first number describes there are three nets in the network, from the second number to the fourth number are describes there are three, two, three hosts in net1, net2 and net3. From line 2 to line 5 are illustrate the hosts in the net1, line 2 describes the network IP address and the mask, from line 3 to line 5 are describing the information of each hosts on net1, it is divided into six columns. First column shows if the host is gateway, second column is the host name, third column is the identify number of the network interface of this host, fourth column is the IP address of the network interface, fifth column is the mask of the network interface, sixth column is the MAC address of the network interface.