Advanced Operating Systems

Rajesh Surana
School of Computing, Informatics and Decision Systems Engineering
Arizona State University
Tempe, US
rsurana@asu.edu

***Abstract*—This project covers everything right from installing of Linux to implementing a new networking protocol for later use in clock synchronization in distributed systems.**

***Index Terms*—Linux, Networking, module, compile, synchronization, distributed.**

I. INTRODUCTION

We choose Linux for our project because it is open source as well as free. In addition, it is easily customizable because we have source code, compiling tools as well as online forums and blogs for help. In all, in terms of cost, support and legality of use, Linux is the best choice. For this project we used Ubuntu 14.04.1 LTS. The project itself is divided in 3 parts.

Before using any tool or software, the basic requirement is that we should be able to install it on our system. That's why first step of part-1 is installation of Ubuntu. Goal is to make ourselves comfortable with the GUI of the Ubuntu and its functionalities such as terminal, text editor or compiler. It is important to understand how to update the software to get the latest security mechanisms as well as new utilities. Same analogy applies to kernel also. In fact, update of kernel is not just meant to be better functionality but an environment which is more secure and bug free. That's why our second step of part-1 is to understand how to update a kernel to its latest revision. Perk of part-1 is at last where we get to know the typical structure of kernel module. We even devise a way to debug a kernel module on top of which debugger run.

In part-2, we build a communication channel that uses normal read/write system calls to communicate over IP. Here, we replace the standard socket interface used by most network programs and allow one application to communicate in full duplex via ipv4. Ipv4 is not to be altered. A write on system A will fill a read buffer on system B and visa-versa.

Part-3 uses part-2 to communicate between machines to implement a logical clock. When sending a message from system A to system B we want to have an ordering to determine among several messages which came first.

II. SOLUTION

This section is divided into 3 parts-

A. Installation, Compilation and Debugging

In part-1 of the project, we decide to use virtual machine instead of directly installing Linux-Ubuntu OS on the machine. Installed Guest OS can be easily recovered if corrupted when installed as virtual machine. Hypervisor used here is VMware Player. In addition, guest OS can be easily migrated as a virtual machine from one system to another. In second step

1) Download and install the utilities needed to perform the compile and packaging of the kernel.
2) Download and decompress the source code for the latest kernel from the Ubuntu source tree.
3) Clean and built the environment.
4) Update the kernel to match your source code.

In the third step of the part-1, we created the application 'cse536app' and character device driver which is used by this application. Then we learnt how to debug the kernel module. Procedure for this step-

1) Compile the program cse536app.c in development directory (CSE536) on Ubuntu.
2) Patch the kconfig and Makefile files in the driver/char directory and compile a .ko.
3) Install and use .ko.
4) Debug the kernel with ddd.

But, the concept of debugging is to give step by step execution of the application by stopping it for each applicable statement. If debugger stops kernel for debugging through every instruction, debugger which run on the kernel will also stop. To avoid this deadlock we used two virtual machines and connected them via named pipes. One could be debugging machine and other is target machine whose kernel can be debugged.

B. Communication Protocol over IP

In part-2, I achieved Registration and deregistration of new IP Protocol using inet_add_protocol() and inet_del_protocol(). New protocol with protocol number 234 is registered to IPv4 using net_protocol datastructure. To access device we used file_operations structure to declare functions. Most tricky part was to send data to IPv4 through sk_buff datastructure. My logic for this is

1) First create and set up sk_buff.
2) Reserve some space for protocol headers using skb_reserve().
3) Add some user data to sk_buff.
4) Put protocol headers and sent the packet to IPv4 using ip_local_out().

C. Logical Clock for Ordering Events

In part-3, most challenging part was to design a logic for logical clock in a distributed system in the absence of regular socket interface responsible for reliable communication. To solve this I followed below steps-

On sender side,

1) We create a data structure of 256 bytes as described in the problem statement.

```
struct packetformat {
uint32_t record_id;
uint32_t final_clock;
uint32_t original_clock;
__be32 source_ip;
__be32 destination_ip;
uint8_t data[236];
} ;
```

2) Then, we set destination address and monitor IP address.
3) Next, we take input from user from our menu W option and assign it to 'data' field of packet.
4) Initialize other members such as record ID =1, final and original clock to 0.
5) In the cse5361 kernel module, we assign clock value to original clock and send the event to destination. At the same time Pass the event to cse536app again and send the copy to CSE536monitor using the udpclient code.
6) When acknowledgement is not received in 5 seconds, resend the message.
7) Increment the clock after each event sent.

On receiver side,

8) Check the record ID, if it is 1 then change it to 0 and assign clock value to final clock value field. Resend the message to sender with rest of the field unchanged.
9) Also, this is the time to update clock. If packet's original clock is greater than local clock then
   clock = received original clock +1;
   final clock = clock;

On sender side again,

10) When acknowledgement is received (record ID is 0) read it in cse536app.
11) If acknowledgement matches with event then send the copy to CSE536monitor using the udpclient code.

III. My contributions

In part-1 of the project, I found out that to install a guest OS of 64-bits on a host machine of 32 or 64 bits a virtualization technology is needed. This setting can be turned on in the BIOS set up which is off by default. Latest versions of Ubuntu needs at least 4 GB RAM and processor power of that equivalent to Intel's i5 processor for host system to run virtual machine smoothly on it. Compilation of kernel in virtual machine could take from 2 hours to 3 days depending on host system RAM and processor power. In part-2 of the project, I created a 'sk_buff' structure which was different from that created by other transport layer protocols which are registered with IPv4 Network layer protocol. 'sk_buff' is a structure used to communicate between different layers of network protocol stack. In part-3, I devised a logic to impose an ordering over messages to decide which came first. I also implemented a logic to copy data from kernel space to user space in case user space data structure can only be updated with the values in kernel space.

IV. Result

I have a fully functional Ubuntu OS running as a virtual machine, a '.ko' module which application can use as a character device driver to access character device, a new networking protocol which uses IPv4 protocol to establish communication between two systems and a synchronization mechanism in distributed systems for maintaining order between messages which came first.

V. New skills, techniques, or knowledge acquired

I got to know the importance of hypervisors and how they could save us from operating system crashes and onerous recovery process when we are experimenting with actual kernel code. I learnt how to write and compile a '.ko' module as well as updating the kernel to latest revision. The major experience of this project was to understand how to compile the source code of the kernel and making an installation image that is used to upgrade to the latest kernel. I understood the working of networking protocol in Linux and have successfully written a networking protocol which uses IPv4 for establishing communication between two systems while burying usual socket interface. As there is no socket interface we have unreliability in the communication. We overcome this by implementing a logical clock to work in an unreliable distributed environment to assign ordering between messages. This gave me a real time experience of how to make distributed environment reliable when there is no centralized clock or memory for achieving synchronization.

VI. Team Members

This was an individual effort project.

References

Some useful references which are not used but include information related to this project-

[1] Jonathan Corbet, Alessandro Rubini, "Linux Device Drivers", O'Reilly Media, 2nd edition, June 2001.

[2] Christian Benvenut, "Understanding Linux Network Internals", 1st Edition, December 2005.

[3] Paul Krzyzanowski, "Clock Synchronization", Lectures on distributed systems-Rutgers University, 2000.