CSE536 Event ordering in Distributed Environment

Nagarjuna Myla
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85287
Email: nmyla@asu.edu

***Abstract*—In this project we worked and did experiments on the Linux kernel modules. This project is divided into 3 projects where in project1 subdivided into project1a, project1b, project 1c. Project1a includes preparing the required software setup. Project1b includes setting up the linux kernel build to the latest version so that can be worked on. Project1c includes creating a new character device module cse5361 experimenting on remote debugging with ddd. This project gives an overall idea on how to build a kernel and debug the kernel using remote machine with tools ddd, gdb, minicom, open ssh, filezilla ftp. Project2 extends the capability of the character device which we created in project1 by talking to cse5361 module on a remote machine by completely burying the IPv4. Builds a communication channel that uses normal read/write system calls to communicate over IP. Project3 extends the capability of the project2 deliverable by impose an ordering on messages using a logical clock over the distributed systems.**

***Keywords*—Linux, kernel, module, character device, network, logical, clock, event, ordering, distributed, environment,ddd, build,protocol, ipv4, communication, vmware, ubuntu,gdb, minicom, debug, configure, filezilla, ssh.**

I. Introduction

A. Terminology

The following terms were used below in explaining the porject work:
IP- Internet Protocol
OS- Operating system
TLDS: top level directory structure
TM: target machine (which is server)
DM: debug machine (which is client)
RecordId - ACK=0 Event=1
Final Clock: To store the global time across multiple systems to impose ordering
Orig Clock: Local clock to the single system to store its events
Source: IP address of source machine
Destination: IP address of destination machine

B. Goal Description

1) Project1A - Installing Ubuntu Linux: In the project we will be installing VMware and installing Ubuntu as a guest OS. We have to install 64 bit VM enviroment and install 64 bit compatible Ubuntu.

2) Project1B - Download and Compile the Linux Kernel: In preparation for compiling applications and kernel objects to run in the Ubuntu development environment we need to have the kernel source code and associated utilities available on our copy of Ubuntu

3) Project1C Build and Debug a .ko and an App: To build kernel and debug it using remote machine. Writing device drivers and accessing data through them. Establishing a linkage between .ko a device driver and Makefile and Kconfig.

4) Project2 Build a communication channel that uses normal read/write system calls to communicate over IP: Our mission is to replace (or at least bury) the standard socket interface used by most network programs and allow cse5361app to communicate in full duplex with another cse5361app at an arbitrary network end location via ipv4 using standard read/write system calls. Ipv4 is not to be altered. A write on system A should fill a read buffer on system B and visa-versa. The calls can be blocking or non-blocking.

5) Project 3 Use a logical clock to impose an ordering on messages: When sending a message from user i to user j over distributed environment we want to have an ordering to determine among several messages which came first. The problems here include
1. Where to store the clock
2. How to handle multiple apps on the same host
3. How to make the communication reliable and not duplicated
4. Transaction format Store IP numbers in standard format (Big Endian in_aton() at kernel level, inet_aton() at the user level)

C. Assumptions

cse5361 Character device buffer can hold a max of 256 characters.

II. Proposed Solution

A. Project1A - Installing Ubuntu Linux

Downloaded the VMware player from the website https://my.vmware.com/web/vmware/downloads and installed. Downloaded Ubuntu from http://www.ubuntu.com/download/desktop and installed on the VM ware. Allocated required RAM of 1GB. Did the required installation process of ubuntu on vmware player.

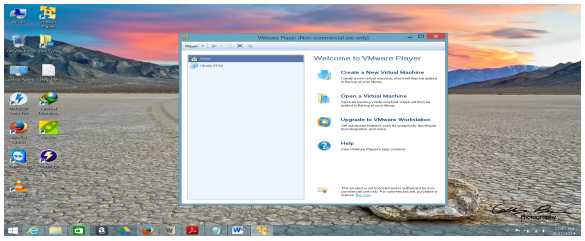

Fig. 1: Project1a: Installing VMware

Fig. 2: Project1a: Installed UBUNTU

B. Project1B - Download and Compile the Linux Kernel

Following steps when executed in a terminal will download the kernel source code and produce an installation image that will allow you to upgrade to the latest kernel and subsequently make modifications to it.

Step 1 Make a directory for kernel development. This is done in a local home directory.
mkdir cse536
cd cse536

Step 2 Download and install the utilities needed to perform the compile and packaging of the kernel. We start by first updating the package list. After that we can download all of the needed packages. Some will have already been installed so this is a shotgun approach but should capture any missing utilities.
sudo apt-get update
sudo apt-get install dpkg-dev kernel-package gcc libc6-dev binutils make bin86 module-init-tools gawk gzip grep libncurses5-dev

Step 3 The next instruction downloads and decompresses the source code for the latest kernel from the Ubuntu source tree. Do not use root for this.
apt-get source linux-image-$(uname -r)

Step 4 This step will take the longest. We must first a) change the working directory to be the top level source directory for the kernel. We can then b) configure the kernel. For now we will retain the default configuration so just exit after the configurator starts. c) The last preparatory step is to clean the build environment. This will cause everything to be rebuilt. d) The last command will take several hours to complete so be sure your computer is plugged in and is not set to sleep. For some computers it may take all night. Do not use any root privileges in this step.
cd linux-3.5.0
make menuconfig
make-kpkg clean
fakeroot make-kpkg –initrd –revision=1.0.cse536 kernel_image

Step 5 This step will show you your current revision of the OS (Note: Software Updater also updated the kernel) and then update the kernel to match your source code. You must reboot (shutdown command) to use the new kernel. You can verify the update by displaying the version again.
cat /proc/version
sudo dpkg -i ../linux-image-3.5.7.2_1.0.cse536_i386.deb
sudo shutdown -r now
cat /proc/version

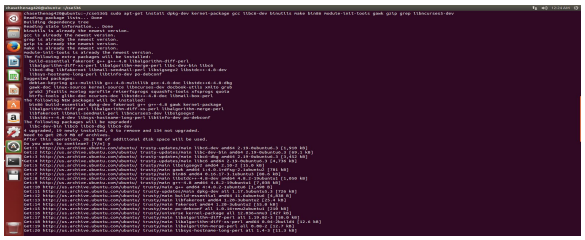

Fig. 3: Project1b: Buliding Kernel

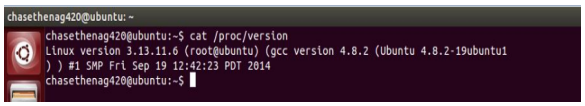

Fig. 4: Project1b: Version of Kernel after build

C. Project1C Build and Debug a .ko and an App

The code required for this can be found in zip file.
Step 1 Compile the program which is is the zip package cse536app.c in your development directory (CSE536) on Ubuntu
Step 2 Patch the kconfig and Makefile files in the driver/char directory and compile a .ko
Step 3 Install and use your .ko
Step 4 Debug the kernel with ddd

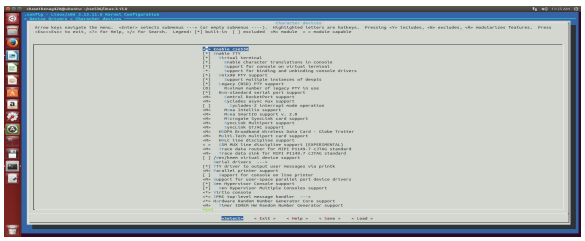

Fig. 5: Project1c: Loading CSE5361 character device

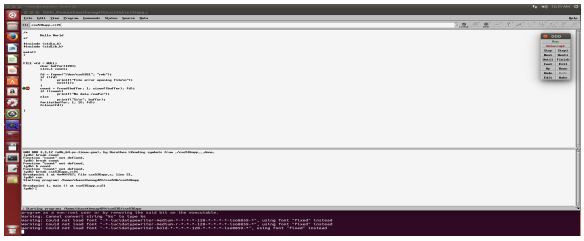

Fig. 6: Project1c: Using ddd to debug

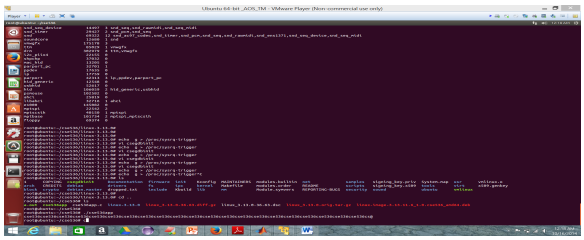

Fig. 7: Project1a: Reading and Writing from device

D. Project2 Build a communication channel that uses normal read/write system calls to communicate over IP

In theory and in practice any Linux kernel module can interface with and use any other Linux kernel module as long as the interfacing module is designed to meet all of the existing module's requirements. In this case our cse5361 module will be modified to communicate with cse5361 modules on other computers by interfacing with standard IPv4. Internet Protocol (IP) normally operates in the standard internet network stack and is able to interface with a variety of transport layer modules as well as link layer modules. Typical transport layer modules include TCP and UDP. A typical link layer module would be Ethernet. In the current world of network technology IPv4 is the internet glue and the lower layer modules interface IPv4 with local hardware while the upper layer modules provide application programs access to IPv4.

The cse536app buffers used to exchange data should have a standard size of 257 bytes.
The first byte of each write buffer designates the type of the remaining buffer
Types include 1-destination IP number in record, Type 2-data. Others may be defined as required.

I have modified the cse536app.c which we created in the project1c to accept an argument which indicates the IP address of remote machine user want to communicate. This module is meant to be for duplex communication so both the communicating systems can talk to each other at same time by writing to the character device of remote machine and reading the character device of local machine. To make this work we have taken protocol number 234 as same as Major number 234 which we have used in the last project1c. When user starts the program by passing first argument as IP address it will be set in global variable in cse5361.ko module and will be used in subsequent communication with remote machine. I have modified the code in device driver cse536_write function which will read the first byte of data to decide whether it is ip address of remote machine or data to be sent. If first byte is 1 it is Ip address so will be stored in a global variable. When a send command issued by user from cse536app first byte is set as 2 so it is taken as data packet by cse5361.ko module and the data is send over IP to destination address which was stored before. To make it work I have set the IP header related parameters and protocol number 234 which we used. This protocol number will be used by remote machine to handover the packet to our cse5361.ko module. Once packet received by remote machine ipv4 will be calling the handler() function in our cse5361.ko module which we added in this project to store the data into buffer. Once the read command from cse536app is issued by user all the data stored in buffer linked list is read and displayed to user. I have modified the code in cse536_read function which we have created in last project1c to check if the buffer is initialized. If so data will be read and last read node memory is released so it will not be read again. Our cse536app will call fread() of cse5361 device in loop until it encounters zero bytes of data to be read from device. This way all the data sent to remote machine is read by remote machine and displayed to user once all data is read it will prompt for next action to read or send data to other machine.

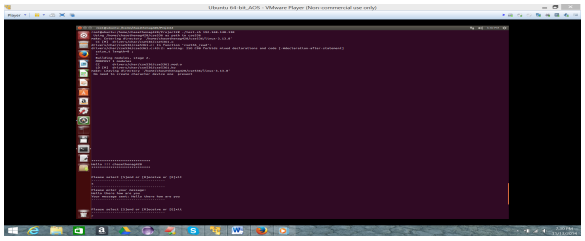

Fig. 8: Project2 AOS Debug Build

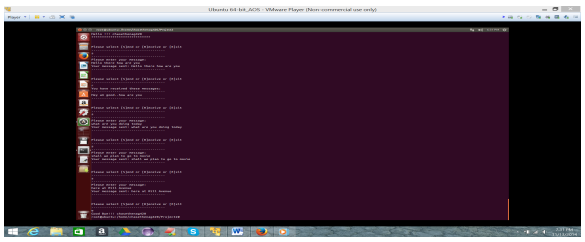

Fig. 9: Project2 AOS Debug Communication

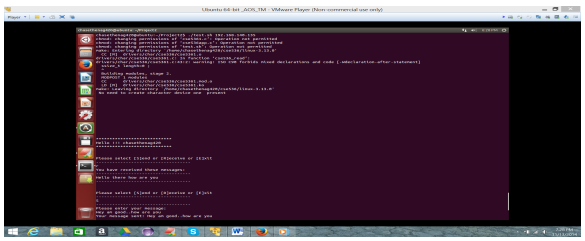

Fig. 10: Project2 AOS Target Build

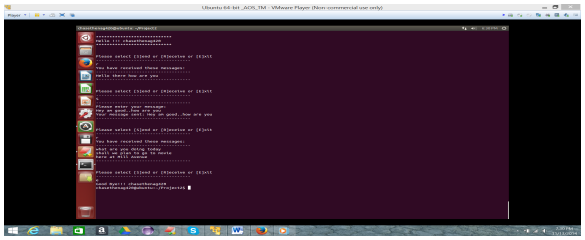

Fig. 11: Project2 AOS Target Communication

E. Project 3 Use a logical clock to impose an ordering on messages

We decided to store clock in the .ko. To handle multiple apps on the same host we decided to use synchronization to control access from multiple users. We have used copy data from the user to the kernel before allowing a tasklet thread to access the data. To make the communication reliable and not duplicated We decided to limit our reliability to waiting for an ack before retransmitting after 5 seconds. If a second timeout occurs discard the event as unobtainable. Keep no state on the receiver ignoring possible duplicates. We will use the following format to communicate over the network for Acknowledgements and Events.

Ack format
4 Record ID ack=0 or event =1
4 final clock
4 original clock
4 source IP
4 destination IP
236 string

Event format
4 Record ID ack=0 or event=1
4 final clock
4 original clock
4 source IP
4 destination IP
236 string

In order to use the monitor all transmissions will need to be made in a uniform manner. Using the record layout specified above app will submit a buffer to the kernel module using the write function. The app will initialize the buffer to zeros and then filled in the record ID with a 1 (for an event), the final and original clock with 0, the source IP with the local IP, and the destination IP with the destination address. The string can be any printable string terminated with a 0. When the write function returns, the original clock field in the buffer is updated by kernel module. The kernel module also have sends the event to its destination. Upon return the app will send a copy of the event, as returned by the module in the buffer, to the CSE536monitor using the udpclient code.
Upon receipt of an event by kernel module the clock is updated and the value stored in the final clock field and the event stored for the local app to read. The kernel module then change the record ID to 0 in a copy of the event and send the copy as an acknowledgement back to the source where it will be stored until the app reads it. When the app reads the acknowledgement it sends a copy to the CSE536monitor using the udpclient code.
Apps will be reading events from the module and acknowledgements for transmitted events as well. Only acknowledgements are sent to the CSE536monitor from the read function.

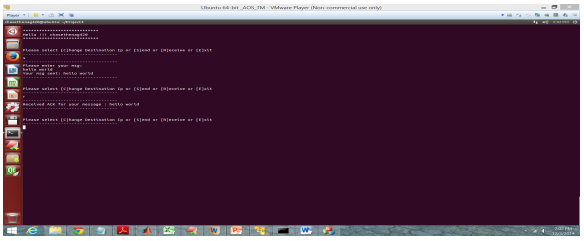

Fig. 12: Project3 AOS Send & Read Message

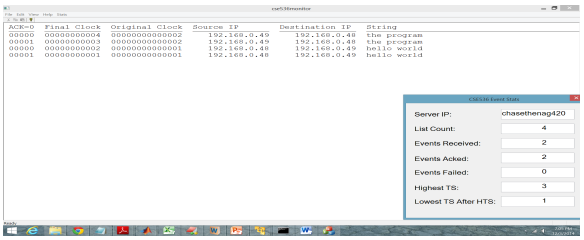

Fig. 13: Project3 AOS cse536monitor

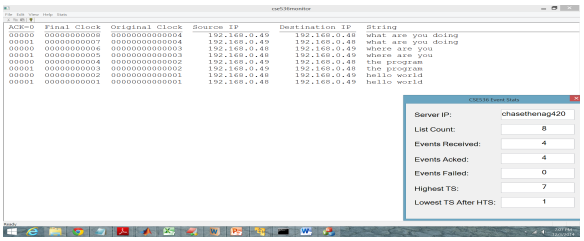

Fig. 14: Project3 AOS cse536monitor

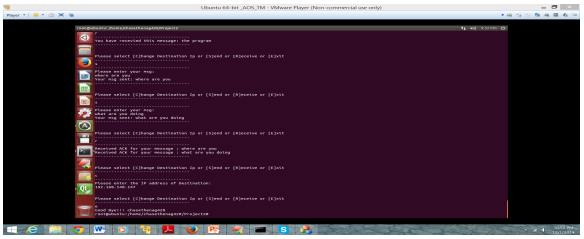

Fig. 15: Project3 AOS change destination ip & exit

III. Execution

To run project you need to complete all the steps in Project3, Project2 and Project1. Once you are ready execution of Project is as simple as running a shell script file. Extract the Project3.zip package and run test.sh by passing two parameters: 1. cse536monitor ip address and 2. remote machine IP address.

Usage: ./test.sh 'cse536monitor ip' 'remote machine ip'
Example: ./test.sh 192.168.0.14 192.168.0.15

Do the same process on the remote machine as well to connect to other machine.

Above script copies required files from Project3 directory to cse536 and linux-3.13.0/drivers/char/cse536 and builds the required modules and make your program to run which you can directly start communicating with other party once the other party is also ready with above steps. If you are running as normal user it will ask for root password.

IV. Conclusion

In this project we have worked and did experiment on setting up linux environment on virtual machine, building the linux kernel, writing a simple character device, writing a new protocol bury ipv4 and talks to other character device on remote machine, creating a logical clock to keep event ordering in the distributed environment while takingl using our character device with new protocol.

Acknowledgment

I would like to thank Dr. Alan Skousen for his support in each and every phase of this project.

References

[1] Network Buffers and Memory Management
http://www.linuxjournal.com/article/1312

[2] Protocol Numbers
http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml

[3] KernelDataStruct
http://www.linux-tutorial.info/modules.php?name=MContent&pageid=87

[4] sk_buff
http://vger.kernel.org/ davem/skb.html

[5] Lamport timestamps
http://en.wikipedia.org/wiki/Lamport_timestamps

[6] Happened-before
http://en.wikipedia.org/wiki/Happened-before