PROGRAM ASSIGNMENT 5:

Basic C Signals and Sockets; ICMP/PING, and Traceroute

Overview

The goal of this lab is to familiarize yourself with network level principals and basic socket programming by re-implementing the ping and traceroute programs.

The programming in this assignment will be in C requiring raw sockets. On most systems, accessing raw sockets requires administrative privilege (i.e., you will require root privilege). You may need to create a virtual machine using VMWare player and install the latest version of Ubuntu.

Deliverables

Your submission should minimally include the following programs for each part of the lab:

• PART 1: MYsignal.c – Makefile – README

• PART 2: ping.c – rtt graph.png – Makefile – README

Additionally, if there are any short answer questions in this lab write-up, you should provide well marked answers in the README.

PART 1: Signals and Timing in C

You will be required to set up basic signals and signal handlers to complete your ping and traceroute program. Here you will write a sequence of simple C program that will teach you the basics of signal handling. A signal is a operating system mechanism that enable programs to be “signaled” to take specific actions. You are probably already familiar with some UNIX signals; for example, by pressing CTRL-C on the terminal, you are directing the operating system (OS) to deliver a SIGINT signal to the running program, which usually has the effect of terminating the program.

You will employ the SIGALRM signal to take periodic actions, such as ping remote host. To help you get started, I’ve provided a basic “Hello World” program mySignal.c:

```
Program: mySignal.c
1 /* mySignal.c */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <unistd.h>
7 void handler(int signum) {  //signal handler
8 printf("Hello World!\n");
9 exit(1); //exit after printing
10 }
11 int main(int argc, char * argv[]){
12 signal(SIGALRM,handler); //register handler to handle SIGALRM
13 alarm(1); //Schedule a SIGALRM for 1 second
14 while(1); //busy wait for signal to be delivered
15 return 0; //never reached
16 }
```

There are two key function calls in hello signal.c: signal() and alarm(). The alarm() system call instructs the operating system to deliver a SIGALRM signal after n seconds, n being 1 in this example. The signal() system call instructs the operating system to execute the function handler when the SIGALRM signal is delivered. Now, it is clear that the "Hello World" program sets up a signal handler for SIGALRM, line 12; a timing for the delivery of the SIGALRM, line 13; busy waits for the signal to be delivered, line 14; and once the signal is delivered, the signal handler is invoked, printing "Hello World" and exiting, line 7-9. At this point you should compile and execute hello signal.c and observe the timing of the output — it is delayed by 1 second.

This style of programming with signal is based on the principals of preemptive execution. That is, the execution of the handler function preempts the main execution of the program. Once the signal is delivered (during the busy wait), program execution jumps to the handler function, and once the handler returns, execution jumps back to the point where the main execution was preempted (or would have, if there was not an exit() call). This is a very powerful (and often confusing) programming paradigm.

Signal Handling Programming Problems

Program solutions to the following problems by extending MySignal.c:

1. Change MySignal.c such that after the handler is invoked, an additional printf("Turing was right!\n") occurs in main() before exiting. You will probably need to use a global variable and change the condition on the while loop.

2. Change hello signal.c such that every second, first "Hello World!" prints from the signal handler followed by "Turing was right!" in main(), over and over again indefinitely. The output should look like:

```
Hello World!
Turing was right!
Hello World!
Turing was Right!
 ...
```

3. Program a new program timer.c that after exiting (via CTRL-C), will print out the total time the program was executing in seconds. To accomplish this task, you will need to register a second signal handler for the SIGINT signal, the signal that is delivered when CTRL-C is pressed. Conceptually, your program will request a SIGALRM signal to occur every second, tracking the number of alarms delivered, and when the program exits via CTRL-C, it will print how many alarms occurred, or the number of seconds it was executed.

PART 2: ICMP Ping Program

Complete a ping program. This will require the use of ICMP (Internet Control Message Protocol) packets and the use of raw sockets. Your program will require root privileges, and so you should develop your code on the virtual machine provided or on your local machine.

At the core of your ping program, you will craft an ICMP ECHO request packet, send it to the specified destination, and wait for a reply. You will send a packet every second, and print output based on the replies. To familiarize yourself with ping, try executing the built-in ping program that comes standard on must OS installations. For example, here is some sample output of pinging google.com on ubuntu:

```
Sample ping output trace [aviv@myrtle] ˜ >ping google.com
PING google.com (74.125.228.7) 56(84) bytes of data.
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=1 ttl=53 time=9.41 ms 64 bytes from
iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=2 ttl=53 time=8.97 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=3 ttl=53 time=8.90 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=4 ttl=53 time=8.93 ms
64 bytes from iad23s05-in-f7.1e100.net (74.125.228.7): icmp_req=5 ttl=53 time=8.79 ms
 --- google.com ping statistics --- 5 packets transmitted, 5 received, 0% packet loss, time 4005ms rtt
min/avg/max/mdev = 8.796/9.003/9.411/0.236 ms
```

Your goal is to duplicate this output as best you can. Note that after pressing CTRL-C, the program prints out statistics of the round-trip times. To get you started, I've provided a sample program, icmp ex.c, that sends a single ICMP ECHO request and receives a reply. You will alter this program so that the sending and receiving is repeated, and round-trip time statistics are printed upon completion. Additionally, I have provided you with a program to calculate the Internet checkqsum. If the checksum is off, your ICMP echo may be dropped.

Hints

• Timing Packets: Check out the gettimeofday() function, and think about embedding information in the data field of the ECHO packet, it will be echoed back at you. • Signals are Asynchronous: Signals can occur at any time, and some function are not re-entrant, which means they will need to be restarted if interrupted.

• Read the Manuals: The man pages are your friend. In the terminal, type man of almost any function, and you should get some information. If not, consult the internet, but be careful not to copy code. You may find it useful to consult the manual pages for socket, getaddrinfo, gettimeofday, and netinet in particular.

• Read the Header Files: Many of the headers, particularly those containing structures that define a packet, are really useful to review. You can find them in usr/include.

• Perform Error Checking: C does not throw exceptions, so although your program is malfunctioning, it will still execute. Instead, take the time to perform error checking. Use perror() instead of manually checking the error code, but sometimes, you might want to check the error codes.

• Network Byte order and Sequence Numbers: Don't forget that you will need to swap the bytes where appropriate. Particularly check the sequence number. Lab Questions/Problems Once your program is finished, complete the following lab questions/problems:

1) Use your ping program it to record round trip time (RTT) statistics for the following destinations:

Use the average across at least 30 pings to each destination, but do not run multiple pings simultaneously. This data should be included in your submission in a space-separated file named RTT-stats.dat with location and RTT as the two columns.

• Philadelphia: planetlab1.cis.upenn.edu	• Colorado: planetlab2.cs.colorado.edu
• New Jersey: planetlab1.rutgers.edu	• Montana: pl1.cs.montana.edu
• New York: planetlab1.cs.columbia.edu	• Seattle: planetlab02.cs.washington.edu
• Boston: lefthand.eecs.harvard.edu	• Palo Alto: pllx1.parc.xerox.com
• DC: planetlab1.cs.georgetown.edu	• Japan: planetlab4.goto.info.waseda.ac.jp
• Chicago: planetlab5.cs.uiuc.edu	• Korea: netapp6.cs.kookmin.ac.kr
• Atlanta: planet1.cc.gt.atl.ga.us	• England: planetlab-1.imperial.ac.uk
• Houston: ricepl-4.cs.rice.edu	• Germany: planet1.zib.de
• Israel: planetlab2.tau.ac.il	• France: planetlab2.utt.fr

2. Do you notice any patterns in the round-trip times? Which destinations take the longest to reach, and which the shortest?

3. Graph the round-trip time versus distance from Swarthmore, and include the graph in your submission folder. Is it always the case that places near-by have faster round trip times?