Computer Operating System Experiment

Laboratory 1

Linux Shell Commands、C programming and debugger tools

Objective:

1) To familiar with Linux Operating system.
2) To practice the usage of Linux shell commands.
3) To practice with the creation of a Makefile.
4) To get reacquainted with C programming.
5) To practice your code reading skills.

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Answer all the questions in this lab sheet.

1. Introduction

What is a shell?

The shell is the command line interface to Linux. It is similar to the Microsoft DOS prompt as known to Windows users, but is much more powerful with its scripting languages. Currently the standard setup of Linux has included two shells, bash and tcsh.

1) Top

top is a text user interface for viewing the most active processes on your system. Press q to quit. Press c to show full commands. Press < or > to change the sorted column. Here are some of the most useful columns of top:

- PID { The process ID
- PR, NR { The priority value and nice value
- VIRT { The amount of virtual memory used by the process (KiB)
- RES { The amount of resident memory (actually located in RAM) used by the

process (KiB)
• SHR { The amount of memory that is sharable (shared libraries, mapped files, etc) (KiB)
• S { The state of the process (S for sleeping, R for running, T for stopped, Z for zombie)
• %CPU, %MEM { The percentage of CPU and RAM used by the process
• TIME+ { The amount of \CPU time" used by the process. The unit for this is seconds. If a process is idle, it is using 0 CPU time.
• COMMAND { The name of the process

2) ps

ps prints out a list of running processes. By default, ps only prints the processes that are part of your own console session, which probably only includes bash and ps itself. You can use \ps -eLf" or \ps auxww" to get way more detail. (There's a long history about why one of these commands requires a dash.)

The ps command contains mostly the same information as top, but it is easier to filter and search, because it only appears once and shows all the processes. The PPID column of \ps -eLf" tells you the parent process ID, which is the process ID of a process's parent. The NLWP column shows the number of light-weight processes (also known as threads) that belong to a particular process.

3) pidof and pgrep

pidof <process name> gets you the process ID of a single running process. For example, pidof vim will get me the process ID of my vim process. If there are multiple matches, then all the process IDs are printed. It's useful as a argument to another program. For example, I could attach gdb to my httpserver process by running \gdb -p $(pidof httpserver)". You can also use pgrep <process name>, which doesn't require the full process name, but only a part of it. For example, \pgrep a" would give me the PID of all processes that contain 'a' in their name.

4) proc

The /proc directory on Linux is a gold mine of information. You can run ls /proc/1 to get information about the process with PID = 1. Many of the tools mentioned earlier are implemented by reading the /proc directory, so if you need more detail about a process, look there! The /proc/self directory is a shortcut to the current process. (If you run ls /proc/self, then you will get information about the ls process!) Some useful parts of proc:

cmdline { The command used to start the process
• cwd { The current working directory of the process
• environ { The environment variables of the process
• exe { The program that is running in the process

• fd/ { A list of the file descriptors of the process
• fdinfo/ { More information about the file descriptors
• maps { The memory map of the process
• net { Details about network configuration
• status { Status of the process

5) lsof

lsof gets you information about the files, processes, and network connections on your system. Each line of output from lsof corresponds to an \open file", which can correspond to a process's current directory, an actual file that is in use by a process, a shared library, a network connection, or a chunk of shared memory. Here are a few useful ways to use it:

• lsof /lib/x86_64-linux-gnu/libc.so.6 { List instances of programs using the C standard library
• lsof -p 1234 { List open files for the process with process ID = 1234
• lsof -i tcp:80 { List uses of TCP port 80 (http)

6) netstat and ip

netstat and ip are two commands that help you debug the network connections on your system. Here are a few useful ways to use them:

• You can run \ip addr" or \ifconfig" to get the IP addresses and network interfaces of the system.
• You can run \ip route" or \netstat -nr" to get the routing table.
• You can run \netstat -nt" to get the list of current TCP connections on the system.
• You can run \netstat -ntl" to get the TCP ports that are listening for new connections.

Several homework assignments in this course will require the use of network connections in your programs. We will introduce more tools for testing network connections when those homework assignments are released.

7) kill, pkill, and killall

To kill a process, you can use \kill <PID>". You can also attach a signal to the command, like

\kill -TERM <PID>" or \kill -QUIT <PID>".

pkill will kill all processes that match a pattern.

For example, \pkill v" will kill all processes with 'v' in the name. killall does the same thing as pkill, but requires an exact match.

2. Tools

Tools are important for every programmer. If you spend time learning to use your

tools, you will save even more time when you are writing and debugging code. This section will introduce the most important tools for this course.

1) Make

GNU Make is program that is commonly used to build other programs. When you run make, GNU Make looks in your current directory for a file named Makefile and executes the commands inside, according to the makefile language.

```
my_first_makefile_rule:

    echo "Hello world"
```

The building block of GNU Make is a rule. We just created a rule, whose target is my_first_makefile_rule and recipe is echo "Hello world". When we run make my_first_makefile_rule, GNU Make will execute the steps in the recipe and print \Hello world". Rules can also contain a list of dependencies, which are other targets that must be executed before the rule. In this example, the task_two rule has a single dependency: task_one. If we run \make task_two", then GNU Make will run task_one and then task_two.

2) GDB: The GNU Debugger

GDB is a debugger that supports C, C++, and other languages. You will not be able to debug your projects effectively without advanced knowledge of GDB, so make sure to familiarize yourself with GDB as soon as possible.

Commands to know

- **run, r**

  Start program execution from the beginning of the program. Also allows argument passing and basic I/O redirection.

- **quit, q**

  Exit GDB

- **kill**

  Stop program execution.

- **break, break x if condition, clear**

  Suspend program at specified function (e.g. \break strcpy") or line number (e.g. \break file.c:80").

  The \clear" command will remove the current breakpoint.

- **step, s**

  If the current line of code contains a function call, GDB will step into the body of the called function. Otherwise, GDB will execute the current line of code and stop at the next line.

- **next, n**

  Execute the current line of code and stop at the next line.

• **continue, c**

Continue execution (until the next breakpoint).

• **finish**

Continue to end of the current function.

• **print, p**

Print value stored in variable.

• **call**

Execute arbitrary code and print the result.

• **watch, rwatch, awatch**

Suspend program when condition is met. i.e. x > 5.

• **backtrace, bt, bt full**

Show stack trace of the current state of the program.

3. Practice:

1) Practice and answer following question.

(1) How do you find your home directory quickly? And change the directory to your home.

(2) Type these commands "**cd /usr/lib/**", "**cd ./bin/**", "**cd ..**", and "pw**d**". Give the current directory which you are located. What are the meanings of . and .. ?

(3) Type "cd" to go back to your home directory. Type "**cat > testcat.txt**" in the command line. After pressing return, type the following line of text "This is a test of cat.", and then press **crtl-d**. Type "**cat testcat.txt**" again. What do you see?

(4) Type "**find . -name testcat.txt –print > list.lst**" in the command line. You will find a file "**list.lst**" in your current directory. Use **cat** commands to show its contents. What is the result? What is the meaning of **> list.lst**?

(5) Go to "/" directory and type the command to find the bin directory.

(6) Open two terminals on Linux. In each terminal, type the command "ps". Give the process number (PID) of the bash process in both terminals. Why they are different?

(7) Start the Fire-Fox web browser by typing "**firefox &**" at the command line and type "ps" in the terminal. Can you see the process of Fire-Fox web browser? Find the parent process and child process.

(8) Use the command "kill" to kill the processes of Fire-Fox. Give the command you have used. What did you see after killing their processes?

2) **Now you are ready to write a program that reveals its own executing structure. The first.c provides a rather complete skeleton. You will need to modify it to get the addresses that you are looking for.**

```
#include <stdio.h>
#include <stdlib.h>

/* A statically allocated variable */
int var1;

int my_func(int i);

/* A statically allocated, pre-initialized variable */
volatile int stuff = 7;

int main(int argc, char *argv[]) {
        /* A stack allocated variable */
        volatile int i = 0;

        /* Dynamically allocate some stuff */
        volatile char *buf1 = malloc(10);
        /* ... and some more stuff */
        volatile char *buf2 = malloc(10);

        my_func (3);
        return 0;
}
```

(1) output the variables and their addresses belongs to stack, heap and data segment. The output of the solution looks like the following (the addresses may be different).

static data xxx, address: 0x501012
Heap: data xxx, address: 0x571203
…

(2) While you are looking through gdb to debug first.c, think about the following questions and put your answers in the file.

• What is the value of argv? (hint: print argv)
• What is pointed to by argv? (hint: print argv[0])

What is the address of the function main?

• Try info stack. Explain what you see.

• Try info registers. Which registers are holding aspects of the program that you recognize?

3) **Write a shell script to capture Process ID by the process name and kill it if it exists.**