Getting Started

Linux system programming
- About this course

Objectives and contents

1. Programming in Linux environment by using C (assume that you already have some C experience).

2. Linux(Unix) system facilities e.g.
   System calls, Process and signal handler, Inter- process communication, development tools, etc.

3. Linux hierarchy e.g.
   Linux file systems (e.g. VFS, ext2, etc), Linux kernel structure, device drivers.

Course Schedule (week 36 - week 50)

week 36 Getting started
Course introduction, Linux system, shell programming

week 37 Working with file
Linux file system, file descriptor and data structure, system calls

week 38 Working with file(continue)

week 39 Development tools
Make-file(program your own command), Linux C/C++ compilers, Debugging

week 40 Process and signals
Create process and terminate, multi-processing, handle signals

week 41 POSIX thread
: POSIX standard, Create threads and terminate, multi-threading

week 42 Case study

week 43 Autum Vaccation :o)

week 44 Inter-process communication(IPC)
: System V IPC facilities: Semaphores, Message queues, etc.

week 45 IPC facilities(Continue)

week 46 Pipes
: Process pipes, Pipe call, Named pipe:FIFOs

week 47 Sockets
: Create and close a socket, Sockets communication, Multi-clients

week 48 Programming GNOME using GTK+ Toolkit GTK+ widgets: buttons, menus, containers, etc.

week 49 Summary

week 50 Exam

Literatures and useful links

1. Beginning Linux Programming, 2nd Ed. Richard Stones, Neil Matthew, WROX press,1999.

2. http://www.dsl.org/cookbook (search for commands)

3. http://www.tldp.org/LDP/tlk/tlk-toc.html (Linux opt.sys.)

4. http://www.erlenstar.demon.co.uk/unix (Unix programming)

5. http://www.cs.cf.ac.uk/Dave/C/CE.html (C tutorial)

6. http://www.freshmeat.net (download free-wares)

Assessment

1. attendance and assignment(small exercises)

2. small project

3. final exam

Linux is not Unix

Unix was developed at Bell laboratory, it becomes a popular multiuser, multitasking operating system for a wide variety of different hardware platforms.

Linux is a freely distributed implementation of a Unix-like kernel, the low level core of an operating system.

Linux contains no Unix code, it takes Unix as a model, it is a rewrite based on POSIX standard.

GNU project

Linux community supports the concepts of free software, i.e. software that is free from restrictions, subject to the GNU General Public License(GPL).

Richard stallman, author of GNU Emacs, founder of Free Software Foundation, on of the the best known editors for Unix and other operating systems.

Locating system programs and development resources

- system programs

  /usr/bin programs supplied by the system for general use, including program development are found here.

  usr/local/bin programs added by administrators for a specific host computer or local network are found here.

usr/local is a good point to compile programs, to access required files, to upgrade the operating system.

usr/X11 is the X window system, which is commonly installed under this directory.

- Make sure the C compiler is installed in your system

```
type in this program, save it as hello.

#include<stdio.h>

int main()

{
printf(‘‘Hello World\n’’);
exit(0);
}
```

```
type in the two lines of commands, to see the
```

```
gcc -o hello hello.c

./hello
```

- Header files

  1. Header files are normally located in /usr/include. Depends on the particular form of Unix or Linux, they can run in /usr/include/sys, /usr/include/linux.

  2. Use header files in non standard place.

     For example, you would like compiler to look in the specific directory, as well as the standard places for header files, which are included in fred.c.

```
gcc -I/usr/openwin/include fred.c
```

3. Search for header file by using “grep” command.

   For example, you don’t remember a function call for exit status

   ```
   cd /usr/include

   grep EXIT_ *.h
   ```

   On the command line, it appears as following.

   ```
   stdlib.h:#define EXIT_FAILURE 1

   stdlib.h:#define EXIT_SUCCESS  0
   ```

Libraries

1. Standard libraries are usually stored in /lib and /usr/lib. The C compiler has to be told explicitly which libraries to search, as by default, it searches only the standard C library.

   A simple example:

   ```
   cc -o fred fred.c /usr/lib/libm.a

   or

   cc -o fred fred.c -lm

   Search for a specific library

   cc -o x11fred -L/usr/opendwin/lib
                   x11fred.c -lX11

   where -L indicates the location of libX11.
   ```

2. Static libraries and Shared libraries

Library is a collection of object files kept together in a ready-to-use form.

The compiler and linker take care of combining the program code and the library into a single executable program. You must use the -l option to indicate which libraries.

Static Libraries : .a
Shared Libraries: .so

When a program uses a shared library, it is linked in such a way that it doesn't contain function code itself, but reference to shared code that will be made available at run time. When the program is executed, the function references are resolved and calls are made to the shared library, which will be loaded into memory if needed.

3. Try it out(build a library)

```
/* This is fred.c */
#iclude <stdio.h>
void fred(int arg)
{
printf(‘‘fred: you passed %d\n’’, arg);
}


/* This is bill.c*/
#include <stdio.h>
void bill(char *arg)
{
printf(‘‘bill: you passed %s\n’’, arg);
}


/* compile the functions */
cc -c bill.c fred.c
ls *.o
```

```
/* This is lib.h. It declares the function
 fred and bill*/
void bill(char *arg);
void fred(int);
```

```
/* This is program.c */
#include ``lib.h''
int main()
{
bill(``Hello World'');
exit(0);
}
```

```
/* compile the function separately, test it */
cc -c program.c
cc -o program program.o bill.o
./program
```

```
/* create our own library called libfoo.a */
ar crv libfoo.a bill.o fred.o
```

```
/* create a table contents for the library,
make the library ready to use */
ranlib libfoo.a


/* use this library to link to program.o */
cc -o program program.o libfoo.a
./program


/* a shorthand */
cc -o program program.o -L. -lfoo
```

Shell Programming

- What is shell?

1. Shell is a program that acts as the interface between you and Unix system, allowing you to enter commands for the operating system.

2. Shell is not suitable for time-critical and processor-intensive tasks.

   Why? because it is an interpreted programming language, execute a single line at once.

3. Shells family: sh(Bourne); csh (by Bill Joy); ksh (by David Korn); bash(Bourne Again Shell); rc (More C than csh, also from GNU project)

4. What shell you have? Let's check!

   ```
   /*check your system, what shell*/
   ```

```
$ ls -l /bin/sh
lrwxrwxrwx    1 root     root
         4 Sep 11  2001 /bin/sh -> bash

/*check the version if you have bash*/
$/bin/bash -version

GNU bash, version 2.04.21(1)-release
(i386-redhat-linux-gnu)
Copyright 1999 Free Software Foundation,Inc

or /* invoke your bash command prompt */
$/bin/bash
bash-2.04$  echo $BASH_VERSION
2.04.21(1)-release
bash-2.04$  exit
exit
$
```

So, I have a Bourne Again Shell! What is yours? Probably the same. It is the Linux staple, from the GNU project .bash, has advantage that the source code is free available.

- Redirection and Pipes

  1. Redirecting output

     ```
     $ ls -l > lsoutput.txt
     $ emacs lsoutput.txt &
     $
     ```

     You can output the result of a command into a file as you like.This is called redirecting, but be careful that your exiting file with the same name will be overwritten.

     ```
     $ ps >> lsoutput.txt
     $ emacs lsoutput.txt &
     $
     ```

     append the result of “ps” command into lsoutput.txt, check the file!

  2. Pipes

     ```
     /* try this on your machine */
     $ ps -x | sort | more
     $ ps -xo comm | sort | more
     ```

```
$ ps -xo comm | sort | uniq | more
```

uniq: only one copy extracted (if there are more than one copy of files)

- Interactive program

  1. A simple example

     ```
     $ /bin/bash
     bash-2.04$ for file in *
     > do
     > if grep -l POSIX $file
     > then more $file
     > fi
     > done
     ```

     you may not be patient to see what happened to your screen, so many files. So, type "q", back to bash prompt, then type "exit", back to shell command line. Please notice the structure of control flow is just same as we program in common languages, e.g. C.

2. A Shell script

   Type this command in an editor, we save it as a file called “first.sh”.

```
#!/bin/sh
echo "Is it morning?Please answer yes/no"
read timeofday

if [ $timeofday = "yes" ]; then
echo ‘‘Good Morning’’
else
echo ‘‘Good afternoon’’
fi
exit 0
```

   Execute this file as following:

```
$./first.sh
Is it morning? Please answer yes or no
yes
Good Morning

$ ./first.sh
Is it morning? Please answer yes or no
```

```
no
Good afternoon
```

3. Function

Type this command in an editor, we save it as a file called “foo”.

```
#!/bin/sh
foo( ){
 echo "Is it morning? Please answer yes or r
 read timeofday

 if [ $timeofday = "yes" ]; then
 echo "Good Morning"
 else
 echo "Good afternoon"
 fi
}
echo "script starting"
foo
echo "script ended"
exit 0
```

Execute this file as following: (if it is executable :o)

```
$ ./foo
script starting
Is it morning? Please answer yes or no
yes
Good Morning
script ended

$ ./foo
script starting
Is it morning? Please answer yes or no
no
Good afternoon
script ended
```

What did we learn?

Unix Philosophy !!!

KISS (keep it small and simple)

Filter

Reusable-component

Flexibility

Review An Example

$\boxed{\text{bill.c}}$, $\boxed{\text{fred.c}}$

```
/*compile the functions */
cc -o bill.c fred.c
/usr/lib/../crt1.o: In function ‘_start’:
/usr/lib/../crt1.o: undefined reference to ‘main’
collect2: ld returned 1 exit status
```

error!!! the correct way: cc -c bill.c fred.c
$\boxed{\text{bill.o}}$, $\boxed{\text{fred.o}}$

/* Create a header file */
$\boxed{\text{lib.h}} = \boxed{\text{void bill(char *arg)}} ; \boxed{\text{void fred(int)}}$

/* Create an executable file(which has main() function) */
$\boxed{\text{program.c}}$= /* This is program .c */

```
#include "lib.h"
int main()
{
  bill("Hello World");
  fred(23);
  exit(0);
}

/* compile the program.c */
$ cc -o program program.c
/tmp/cc80IAYR.o: In function ‘main’:
/tmp/cc80IAYR.o: undefined reference to ‘bill’
/tmp/cc80IAYR.o: undefined reference to ‘fred’
collect2: ld returned 1 exit status
```

error!!! the correct way: cc -c program.c $\boxed{\text{program.o}}$

```
/* link the program.o to bill.o,
resolve the reference of bill() */
$ cc -o program program.o bill.o
program.o: In function ‘main’:
program.o: undefined reference to ‘fred’
collect2: ld returned 1 exit status
```

error!!! the correct way:
cc -o program program.o bill.o fred.o

```
/*execute the program */
./program
bill: you passed Hello World
fred: you passed 23
```

for this trivial program, you need to resolve the reference of each function call one by one, you may lose your mind if there are 100 function calls!

```
/* create a library to contain the reference of function calls */

ar crv libfoo.a bill.o fred.o

/* make a table of contents for this library.(not necessary for some system)*/

ranlib libfoo.a
```

now you can use this library to link to program.o directly

```
cc -o program program.o libfoo.a

/*execute the program */
./program
bill: you passed Hello World
fred: you passed 23
```

More about command line

Unix command line is not so mysterious. When you type a command, you actually run a program. Take a look at inside the program, you would be able to creat your own.

```
$ echo hello, world
$ hello, world
```

echo is a program with argument “hello, world”. And it can is implemented as followed,

```
#include <stdio.h>
main(int argc, char *argv[])
{
while(--argc > 0)
/* conditional expressions*/
printf((argc > 1) ? "%s" : "%s", *++argv);
printf("\n");
return 0;
}
```

Th main() function is called with two arguments. First, argc is the number of command-line arguments, *argv[] is a pointer to the beginning of the array of character strings which contain the arguments.

Conventionly, the index of argv[] starts from 0, as argv[0], argv[1], ..., argv[n]. e.g. in **echo hello, world**, argv[0]=echo, argv[1]=hello, argv[2]=world. And argc is at least 1(only with name of executing program).

In this program, argc is counted down, and when $argc > 1$, the argv is increased, starting with ++argv, makes it originally point at argv[1] instead of argv[0], because we don't want to echo back "echo"!

exercise

Type in this file, save it as echo.c, and compile it as before we learned(given a name "echo").

THE END