Introduction to C Programming in Linux Environment

Ibrahim Korpeoglu
Department of Computer Engineering
Bilkent University
Spring 2004

This document includes information about the followings:

- Logging to Linux and setting up the environment.
- Writing your first program
- How to compile your program?

A. Logging into Linux and Setting Up the Environment

On the graphical interface that is provided to you, just type your user name and password to enter the system. You can create one or more windows by scrolling the menu bars and finding out an option for a terminal window. For example, in Mandrake Linux we have KTerm, XTerm, Konsole, as the different X terminal programs. Just start one of them, say Xterm.

When you select Xterm it will open and give you a window to type in your commands. You can open multiple Xterm windows.

On an Xterm window, what you have is a command prompt where you can type in your commands. The prompt is provided to you by a process called a shell. A shell is a process that is responsible from taking commands from a user and interpreting them (execute them).

There is not a single shell program in Linux that is available for your usage. There are different shells available. You can use any one of them. The following are the names of some of the shell programs that you can find in your system:

`bash` - Bourn Shell
`csh` - C Shell
`tcsh` - C Shell with filename completion and command line editing.
`ksh` - Korn Shell.

By default, Linux may use bash. But you can change to a different shell. I suggest bash or tcsh, which provide command name completion and browsing the history of commands that you have just typed by using the arrow keys.

When a shell starts (when you created an Xterminal window), it reads a configuration file

and sets some environment variables. The name of the configuration file depends on the shell that is used. The csh and tcsh, for example, are using a file named ".cshrc". The bash shell is using a file named ".bashrc".

You can edit those shell configuration files to configure the environment. The configuration may include:

- Setting up the PATH environment variable (i.e. specifying a list of directories where a given command or program will be searched for by the shell when you type the name of the command or program to run it) .
- Setting up the appearance of the command prompt.
- Setting aliases (short names) for the programs that you want to execute by typing shorter names or by given some options to the program which you do not want to type always.
- Setting up the values for some environment variables for the shell. Those environment variables are used by the programs that are started up from that shell. PATH, for example, is such an environment variable.

You can see the current settings for some of the environment variables by typing `"set"` at the bash command line (assuming your shell is bash).

You can learn more information about a shell by using the `"man"` tool (from manual) of Linux (and Unix).

If you type "`man bash`" or "`man tcsh`" at the command prompt, you will see the manual pages for those shells, bash and tcsh. These manual pages include all information you may need about the shell and about how to write shell scriptsusing the commands that are available in the shell. This includes the syntax of the scripts specific to that shell.

You should have the "`.bashrc`" file in the top level directory of your home directory (the directory that you log into by default). For example, the home directory of a username "ahmet" may be: /home/ahmet/. Then the user Ahmet should have the `.bashrc` file in this directory (not in a sub-directory of this directory).

A "`.bashrc`" file may include the following lines to set the PATH environment variable and to set some aliases.

```
PATH="PATH:/sbin:/usr/sbin:/bin:/usr/gnu/bin:/usr/local/bin:/usr/ucb:/b
in:/usr/b
in:/usr/local/sbin:/usr/X11R6/bin:."

# User specific aliases and functions
alias emacs="emacs -fn 9x15"
alias mail="mozilla -mail"
alias web="mozilla"
alias moz="mozilla"
```

The line for setting PATH include the directories like "/sbin", "/usr/bin", etc., to search for the command or programs whose names are typed-in at the command prompt of the bash shell. The last directory in the PATH specification is a special directory name "." (a dot). This corresponds to the current directory where you type the commands. This is useful to force the shell to search also the current directory (whatever it is) for the commands/programs whose names are typed by the shell user.

You can copy this `.bashrc` file from one of your friends if he/she has already set it up. Then, you may modify it according to your needs.

Examples of programs and their locations in the directory hierarchy where you can find them are shown below:

`gcc` (GNU C compiler): /usr/bin
`g++` (GNU C++ compiler): /usr/bin
`tcsh`: /bin/tcsh
`xxgdb` (GNU debugger with graphical interface): /usr/X11R6/bin/xxgdb
`gdb` (GNU debugger with command line interface): /usr/bin
`ifconfig` (command to see the network configuration): /sbin
`netstat` (command to see some network status information): /bin
`emacs` (a text editor): /usr/bin
`vi` (a text editor): /bin
`make` (a utility to compile big projects that include many source files): /usr/bin

You can get more information about the commands and programs listed above again by using the `man` utility. For example, if you type "`man gcc`", you will get a man page that will give you a very detailed information about the gcc compiler, the parameters it may take, etc.

The `man` tool does not only give information about the commands or installed programs. It may also give information about the system calls or C library functions that available to use. For example, if you type the following, it will give you detailed information about the C library function `printf` and its usage.

```
man printf
```

The manual pages available in a Unix system is categorized into sections. For example, man pages for commands are in section 1, man pages for system calls are in section 2, and man pages for C library functions are in section 3, etc. If somebody gives you a command or function name and the section number, and if you want to find out the corresponding man page, then you have to invoke the man tool with `-S` (section-number) option. (Note that it is capital S). For example,

```
man -S 2 fork
```

```
man -S 3 print
man -S 1 ls
```

Here, the `fork` system call is searched in section 2, `printf` library function is searched in section 3, and `ls` command is searched in section 1.

If you want to give a keyword to `man` tool and see all the commands and functions related with that keyword, then you have to use the `-k` option of the `man`. (type "`man man`" to see the available options of `man`). For example, "`man -k print`" will give you all print related function/command names. You can then select one of them and see its manual page.

B. Writing your first program

Now, lets briefly describe how we write C programs in Linux. What I will describe is not the only way. There are integrated development environments in Linux which can be used to write, compile, debug, and run programs from inside the same development environment. I will not describe here their uses. I will describe each step separately: first writing the program, then compiling, then debugging, and then running.

You can write a C program using one of the editors emacs, vi, joe, or pico. There are also editors like KWrite. Hence, you are not restricted with the editors whose names are mentioned here.

Now, assume we start an editor, say `emacs`, and type in the following program, and save it with a name `linecount.c`. The program will count the number of lines in a text file and print out to the screen.

```
/************************************************************
                  CS 342 Operating Systems
                     Bilkent University
               Department of Computer Engineering
************************************************************/

#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <stdlib.h>
#include <ctype.h>
#include <dirent.h>

/*
   The First argument to the program should be the name of directory.
   The Second argument to the program should be the name of a file.
*/

int
```

```
main(int argc, char *argv[])
{

  int  cid;           /* child prcess id */
  char dirname[256];  /* name of directory */
  char filename[256]; /* name of file */
  char line[500];     /* buffer to store the line read from file */
  int linecount = 0;  /* the number of lines in the file */
   DIR  *  dirp;         /*  pointer  to  an  open   directory  -  directory
handle */
  FILE *fp;           /* pointer to an open file - file handle */

  if (argc != 3) {
    fprintf(stderr, "wrong number of arguments");
    exit(1);
  }

  strcpy(dirname,  argv[1]);
  strcpy(filename, argv[2]);

   /*  Here  we  are  testing  if  we  received  the  program  parameters
correctly */
  printf("directoryname = %s\n", dirname);
  printf("filename = %s\n", filename);


  /* First we change to the specified directory */
  if (chdir(dirname) < 0) {
    fprintf(stderr, "can not change into the directory \n");
    exit(1);
  }

  if ((dirp = opendir(dirname)) == NULL) {
    fprintf(stderr, "can not open the directory\n");
    exit(1);
  }
  /* The specified directory is is now the current directory */

  /*
     We create a child now that will read the given file. We will do
this
     just to demonstrate the use of fork().
  */
  cid = fork();
  if (cid == 0) {
    printf("this is child\n");
    fflush(stdout); /* this is to force the output to go to the screen
               immediatly */

    /* The child will open the file and count the number of lines */

    fp = fopen(filename, "r"); /* open the file for reading */

    while (!feof(fp)) {
      if (fgets(line, 500, fp) != NULL) /* assuming a line will have
```

```
less than 500 characters  */
    linecount++;
    }

    /* Now print out the result to the screen. */
    printf("the number of lines in this file = %d\n", linecount);

    printf("child says bye...\n");
    exit(0); /* exit normally */
  }
  else {
     /*  This  is  the  parent.  We  don't  need  to  do  anything.  We  just
terminate */
    printf ("parent says bye...\n");
    exit(0); /* exit normally */
  }
}
```

Now, your program is ready to compile, debug, run and test.

C. **How to compile?**

Since this is program written in C language, we will use a C compiler to compile it. The GNU compiler `gcc` is a widely used C compiler for Unix platforms.

In a window (hence at the command prompt of the shell), type the following to compile your program:

```
gcc -g -o linecount linecount.c
```

Here:

`gcc` is the name of the GNU compiler (which is a program stored in directory "/usr/bin").

`-g` option is an option of the compiler to force the compiler to generate debugging information which can be used by a debugger like `gdb` if you want to debug your program. If you don't put this option, you can not debug with `gdb` or `xxgdb`. Therefore, this option is important to have.

`-o` option is used to specify the name of the exutable file where the compiler output will be written into. Here linecount will be the name of the executable program that you have written. If you omit this option, then the executable file will have a default name `a.out`.

Then comes the name of the program source file (linecount.c). This is the input to the compiler.

After compiler compiles your program, it will produce an executable file named `linecount.` You can type this name to run your program. If the current directory (which is denoted by a dot "." in PATH environment variable) is not in the PATH, you

can either include the current directory (".") in the PATH and then run your program again, or your can give the path of your program relative to the current directory as follows: "`./linecount`".

But while invoking (starting running) your program, you should also specify the arguments that it expects: a directory name and a filename. For example, if we invoke the program using the following command syntax:

```
linecount . linecount.c
```

The program `linecount` will go into directory "`.`", which is *the current working directory* (i.e. the directory where the executable program `linecount` and its source code `linecount.c` are located). The program will open the file linecount.c and will count the number of lines inside the file and print that out. The following is the output that we can see on the screen:

```
directoryname = .
filename = linecount.c
parent says bye...
this is child
the number of lines in this file = 87
child says bye...
```

The output shows that there are 87 lines in the file `linecount.c`.