Developing on Linux

An Introduction to Development on Linux

Nathan Thomas

Red Hat, Inc

nthomas@redhat.com

1. Introduction

Switching to Linux for your development projects can seem like a daunting task at first; but given a little direction, you will find that thedevelopment environment is both powerful and easy to use. This paper is designed as a quick introduction to development under Linux, and will help you get your feet wet with the tools that you will need for a large scale development project. For now, the scope of this tutorial is writing a program in C or C++, and includes information on text editing, compiling, debugging, and version control.

There are an endless array of options available to Linux developers when it comes to Integrated Development Environments, debuggers, version control systems, and all other associated development tools. For simplicity's sake, this paper only covers the lowest common denominator among these tools, focusing on the development environment that comes standard with the Red Hat Linux distribution. After understanding the basic tools you are encouraged to go out and look at the options available to you so that you can build your own development environment based on your own preferences. There are a lot of good programs on the Linux Applications CD shipped with the Red Hat box set, and www.freshmeat.net is a great repository for all sorts of Linux development tools.

The thing that Windows developers miss most when they begin working on Linux is a nice graphical IDE. There are such tools available; just as with other operating systems, they are not all included with our standard operating system distribution. They are available from a number of commercial vendors, and provide a similar level of functionality as those available on Windows. As previously stated, this paper is designed to speak to the lowest common denominator of development tools, and as such does not cover these IDEs. It should also be noted that these IDEs are generally wrappers for the tools that we are going to discuss here, and getting to know the way these underlying tools work will help even those who plan to work with a commercial IDE.

2. About this Tutorial

For this tutorial we be using the following tools:

- text editor: vi
- c compiler: gcc
- c++ compiler: g++

- project control: make
- version control: cvs
- shell: bash

It is assumed that you know the basics of how to navigate through a Unix-like operating system. If not, it is recommended that you play around with your shell account and get some introductory reference material to help you find your way around. The best piece of advice there is on this front is use the 'man' command as much as possible. As an example, type 'man bash' at the command prompt. If you see any commands in the text of this tutorial that you don't fully understand, try using'man' to learn more.

Not to belabor the points made earlier about all the options available to you as a developer, but it should be noted that our selections here are actually not even the only choices available in the standard operating system. Again, these tools are the lowest common denominator among what is available on most Unix systems, and as such will serve our purposes well.

There is one more caveat to mention here: it is assumed you have set up the development environment on your Linux system; you had the option to install development packages when you went through the operating system installation procedure. If you haven't done so, you will need to add the necessary packages into your system now. How do I add the development libraries to my system now?

So, without further ado, let us start our first Linux development project.

3. Setting Up Your Shell

First, we need to make sure that your shell is set to the bash. To do this, use the echo command:

```
[nthomas@daisy nthomas]$ echo $SHELL
/bin/bash
```

If it comes back as anything else, use the chsh to change it to bash

```
[nthomas@daisy nthomas]$ chsh -s /bin/bash
```

This will change your shell to bash for the time being; if you want to make bash your shell permanently you will need to edit your login scripts, which is outside of the scope of this paper.

Now that your shell is the same as the one used in the illustrations in the book, your command line syntax will match up to what you see here.

4. Developing a Simple C Program with `vi` and `gcc`

For our first project, we will be creating the classic "Hello world!" program in C.

To make this tutorial easier, it is recommended that you do your file editing in one window and your compiling in another window (or different virtual consoles, if you aren't using X-Windows; to switch between virtual consoles, use the ALT-Function keys)- you can open this tutorial text in yet another window if you're reading it electronically.

Before we begin editing the program we're going to write, let's decide where we want to store the files. For now, we aren't concerned with version control, so we can just store the file under our home directory. So, go to your home directory:

```
[nthomas@daisy nthomas]$ cd ~/
```

Now create a subdirectory called 'src'

```
[nthomas@daisy nthomas]$ mkdir src
```

Now go into that directory and create a directory called 'hello':

```
[nthomas@daisy nthomas]$ cd src
[nthomas@daisy src]$ mkdir hello
[nthomas@daisy src]$ cd hello
```

Now, we can begin editing our first file, called hw1.c

In the window or virtual console that you want to use for editing, open the file for editing with vi:

```
[nthomas@daisy hello]$ vi hw1.c
```

This will put you into the vi editor. Vi is a very low level editor, and is generally available on any Unix system you use. It is a modal editor, and expects the user to tell it what they want to do (edit/delete/change/etc) before the user does it. For our purposes, there are only a few commands you need to know. First is how to insert text. To do this, you type "i". This tells the editor we will be adding text now. Of course, if you make a mistake and need to delete some text, you need to take it out of INSERT mode. To do this, you can hit the ESC key. Then, to delete the code you mistyped, go to that text and hit the 'x' key to delete one character at a time. Once you have the code finished, you can save it by escaping out of INSERT mode and typing ':w'. To quit the editor, escape out of INSERT mode and type ':q'. If you want to quit without saving, or overwrite an existing file vi might complain about doing these commands, but if you add an '!' to them, it forces vi to do what you want: ':q!' or ':w!'. You can also do both at once: 'wq!' will save and quit.

Vi has its detractors, but it is a very powerful editor, and once you have mastered its syntax you can blaze through text editing more efficiently than with just about any editor out there. It has a

good bit of functionality with regards to code evaluation, and has been the standard Unix editing tool for generations (quite li erally at this point). If you are unhappy with vi, you can look into emacs or pico, which are other command line editors with their own foibles. To learn more about vi, type 'man vi'; there are also a number of books covering vi use available.

So, back in vi, put yourself in INSERT mode (by pressing 'i') and you can begin to write your program. Following is the text for a simple helloworld program in c:

```
#include <stdio.h>

main() {
   printf("Hello World!\n");
   return(0);
}
```

Simply type this into the vi editor, and then save the file by escaping out of INSERT mode and typing ':w'.

Now, in the window or virtual console you want to use for compiling, you are ready to compile the hw1.c file and see if it works.

To do this, we use the Linux c compiler, gcc. Gcc is the Gnu C Compiler, and is an open source compiler maintained by the Free Software Foundation. Gcc uses the glibc c libraries as its definition of the c language. It is now completely ANSI compliant. The standard include path for gcc is /usr/include and this is where it expects to find header files included in angle brackets- <stdio.h> for example. For our program we have included stdio.h, which is the standard input/output header for c. This file's complete path is /usr/include/stdio.h and you can go look at it there if

you want to see how it works. If you wanted to include a file that wasn't in the standard header path of /usr/include you would put it in quotes instead of angle brackets, and if it was anywhere other than the directory where compilation was occurring, it would need a full path specified- like this: "/tmp/foo.h" .

If you want to find out more information about the way that gcc works, you can read the GCC-HOWTO file

To invoke gcc, you simply type 'gcc <filename.c>' and it will compile foryou.

```
[nthomas@daisy hello]$ gcc hw1.c
```

If you have no errors, gcc will return nothing. If you have errors, it will give them to you:

```
[nthomas@daisy hello]$ gcc hw1.c
hw1.c:4: unterminated string or character constant
hw1.c:4: possible real start of unterminated constant
```

In this case, there was no closing quote on the "Hello World!" string constant given to the printf statement. Just as on any other system, there are certain compile time error messages that will become familiar to you as indicative of a particular problem; some of the most common will be presented throughout the text of this paper.

Once the code is fixed, it compiles fine. If you now do an ls in the ~/src/hello/ directory, you will see that a file named a.out is there. This is the output file from the compiler, and its permissions are set to be executable

You can now test your program by invoking it from the command line of the window or virtual console where you compiled it. If you have added ’./’ to your path, you can just type ’a.out’; if you haven’t added ’./’ to your path you will need to type ’./a.out’ to tell your shell that the file a.out is to be found in the directory you are in.

Assuming you have written your C code correctly you should see this:

```
[nthomas@daisy hello]$./a.out
Hello World!
```

So, we now have a simple c program complete. Obviously though, this has little resemblence to the sort of programming most developers do. Now, we are going to transform this hello world program into a serious development project, with multiple headers, multiple object files, worldwide source contribution, and integrated debugging symbols. All before lunch.

To do all of this, we need to decide where we are going with hello world. It is not the goal here to actually create a complex program, but rather a program that utilizes all of the standard development tools. To do this, we will produce an expanded hello world program that ouputs "Hello World!" (or whatever we want) to every terminal that the user calling the program has open. This will wind up being very similar to the program ’wall’ which is part of the sysvinit suite and outputs messages to all users on a system. If you want to look at the sources for wall, you will need to unpack the sources from the sysvinit SRPM.

For now we will continue to edit the file in your home directory. We won’t really need the original hello world file from this point on, so open a new file called hw2.c

```
[nthomas@daisy hello]$ vi hw2.c
```

This version of the program will create a function called massivehello() that outputs a text string of our choosing to every console owned by the user calling the program .

It should be noted that this may not work for terminals being run under X-Windows because the process ownership may not match up.

The code (with comments) is as follows:

```
/*************************************************************************
Copyright (c) 1999 Red Hat, Inc.  All rights reserved.  This software is
licensed pursuant to the GNU General Public License version 2 or later
versions [or GNU Lesser General Public License], a copy of which may be viewed
at www.gnu.org/copyleft/gpl.html.
*************************************************************************/

#include <string.h>        /*For the strcmp call*/
#include <unistd.h>        /*symbolic constants- O_WRONLY and O_NDELAY here*/
#include <stdio.h>         /*Standard input and output for our writing out*/
#include <utmp.h>          /*UTMP functionality- try 'man utmp' for more
info*/
#include <fcntl.h>         /*file control for handling file descriptors*/
#include <pwd.h>           /*for access to password/user information*/

/*This is our primary function- it takes in the string to output*/
int massivehello(char *message)
{
   int uid;                /*current user's id number*/
   static int fd;          /*file descriptor id*/
   FILE *fp;               /*file pointer*/
   char *ttydata, *user;   /*buffers for terminal info, and username*/
   struct passwd *pwd;     /*password data structure*/
   struct utmp *utmp;      /*process information structure*/

   uid = getuid();         /*get the user's id number*/
   pwd = getpwuid(uid);    /*get the current process id (this program!)*/

   setutent();             /*go to the beginning of the process list*/

   user = pwd ? pwd->pw_name : "root";  /*figure out who is running this*/


   /*go through the process list and for each entry our 'user' owns, output*/
   /*our message*/
```

```
   while ((utmp = getutent()) != NULL)
   {
      if(utmp->ut_type != USER_PROCESS || /*make sure it's a user process*/
         utmp->ut_user[0] == 0 ||         /*and there is a valid user*/
         (strcmp(utmp->ut_user, user)))   /*and that user is our 'user'*/
      {
         continue;          /*skip this entry if any of that stuff failed*/
      }
      sprintf(ttydata, "/dev/%s", utmp->ut_line);

      /*ttydata is the name of the terminal to write to*/

      fd = open(ttydata, O_WRONLY | O_NDELAY);
         if ((fp = fdopen(fd, "w")) != NULL)
         {
            fputs(message, fp);
            fclose(fp);
         } else
         {
            close(fd);
         }
   }
   endutent();                      /*close the utmp file*/
   return(0);                       /*exit with no errors*/
}

int main ()
{
   int retval=0;
   retval = massivehello("Hello World!\n");  /*call our function!*/
   return(retval);
}

/*I recommend you look at the 'wall' code to see a more robust version
of these operations; this version has very little error checking or
versatility.*/
```

If there are any of the header files that you want to find out more about, you can either look through the source code (in /usr/include/) or you can type 'man <filename>' (ie., 'man stdio.h') to read a description. This should work for most of the standard header files in /usr/include.

If there are sections of this code that still do not make sense to you, try reading the footnotes that have been made to this point, as they explain a lot of the underlying nature of the Unix operating system structure (and hence Linux). If you are still having difficulty, it is recommended that you do some reading up on Linux sytem programming. Standard reference books on this subject include "The Unix Programming Environment" by Kernighan and Pike, "The Standard C Library" by Plauger, and "The C Programming Language" by Kernighan and Ritchie.

5. Using gdb to Debug a C Program

Now that we have begun to deal with pointers and memory issues there is a greater chance that we will decide we want to use a debugger to check on the referenced value of some of the variables. To do this, we will move onto a brief introduction to our next tool, gdb.

Gdb is the Gnu DeBugger, and is a simple command line debugger that is most useful when used to evalute a c binary with debugging symbols compiled in.

To compile debugging symbols into our code, we have to give gcc a -g argument:

```
[nthomas@daisy hello]$ gcc -g hw2.c
```

Now, call gdb with a.out as your argument:

```
[nthomas@daisy hello]$ gdb a.out
(gdb)
```

There are a tremendous number of ways to run gdb, and it is recommended to learn more about it that you read the man page for gdb (there is a reference in the man page to additional resources). Typing 'help' in gdb will also give a concise list of your options. Here, we will cover the basic gdb operations of setting breakpoints and stepping through the running code.

In gbd, type 'run' to get the program to proceed.

```
(gdb) run
Starting program: /home/support/nthomas/src/hello/a.out

Program received signal SIGSEGV, Segmentation fault.
0x40000031 in ?? () from /lib/ld-linux.so.2
(gdb)
```

Obviously, there is something in the code that is causing a segmentation fault. This is good, as it gives us a chance to explore how we use gdb to solve a problem in our executable (this fault is included in the source given above).

It is possible because of the nature of this particular problem that your code will not exit with the same failure, or may not even fail at all. Many of the memory values that are mentioned in the upcoming text will vary from machine to machine, and you should not be concerned by this.

In this case, the failure occured while running through code in ld-linux.so.2, and gdb doesn't tell us what portion of the hw2.c code was executing at the time. So, we will need to step through the code to see where the failure occurs.

You can set break points by typing 'break' and the line number of where the breakpoint is in the original source code. With a larger piece of source code it would be necessary to narrow down where in the code the failure might be occurring by setting breakpoints around the major functions; here it is a little bit unnecessary, but we will go ahead and walk through the steps.

First, we set breakpoints around our major function calls- massivehello() in this case. To do this, use the break command, and give it the line number of the function call in the original source file. massivehello() is called at line 70, so set a break at 70 and 71 (71 will automatically drop down to 72, which is where the next action occurs, at the 'return' call):

```
(gdb) break 70
Breakpoint 1 at 0x8048791: file hw2.c, line 70.
(gdb) break 71
Breakpoint 2 at 0x80487a3: file hw2.c, line 71.
(gdb)
```

Now, run the program and hit the first breakpoint:

```
(gdb) run
Starting program: /home/support/nthomas/src/hello/a.out

Breakpoint 1, main () at hw2.c:70
70       retval = massivehello("Hello World!\n");  /*call our
function!*/
```

Now, continue through the breakpoint by typing 'c':

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
```

```
0x40000031 in ?? () from /lib/ld-linux.so.2
(gdb)
```

So, before we reached the second breakpoint there was a segmentation fault, so the problem is in the massivehello() code. Now, we need to step through the massivehello() code and see where the failure occurs. To do this, we use the 'step' command to step into the massivehello() function and then the 'next' command to run through each line inside:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/support/nthomas/src/hello/a.out

Breakpoint 1, main () at hw2.c:70
70       retval = massivehello("Hello World!\n");  /*call our
function!*/
```

We hit our first breakpoint, and now we begin to 'step' through it.

```
(gdb) step
massivehello (message=0x804881f "Hello World!\n") at hw2.c:25
25       uid = getuid();         /*get the user's id number*/
```

Now that we've stepped into the function, we look at the 'next' statment until we see a failure or return from the function (using 'c' for continue will drop us through the function to its return again if we want- additionally, 'step' will send us into any function that is called by massivehello() just as it sent us into massivehello()).

```
(gdb) next
26       pwd = getpwuid(uid);    /*get the current process id (this
program!)*/
(gdb) next
28       setutent();             /*go to the beginning of the process list*/
(gdb) next
30       user = pwd ? pwd->pw_name : "root";  /*figure out who is running
this*/
(gdb) next
36       while ((utmp = getutent()) != NULL)
(gdb) next
```

```
38          if(utmp->ut_type != USER_PROCESS || /*make sure it's a user
process*/
(gdb) next
42             continue;          /*skip this entry if any of that stuff
failed*/
(gdb) next
38          if(utmp->ut_type != USER_PROCESS || /*make sure it's a user
process*/
(gdb) next
42             continue;          /*skip this entry if any of that stuff
failed*/
```

The debugger is now going through the while loop that checks each process to see if it a viable candidate for outputting the message to; there are quite a few processes running, so all of this output won't be included here. These first several are failing on the check to make sure they are user processes; eventually we get to one that is a user process.

```
(gdb) next
38          if(utmp->ut_type != USER_PROCESS || /*make sure it's a user
process*/
(gdb) next
44          sprintf(ttydata, "/dev/%s", utmp->ut_line);
```

This utmp entry is a user process; to take a look at what sort of process it is and who owns it you can use the 'print' function in gdb:

```
(gdb) print utmp->ut_user
$4 = "nthomas", '\000' <repeats 24 times>
(gdb) print utmp->ut_type
$5 = 7
(gdb) print utmp->ut_line
$6 = "tty1", '\000' <repeats 27 times>
```

This is a tty being used by the user nthomas. Since I know that nthomas is the user who is running the gdb session (because that's me!), this utmp entry should be a viable candidate to write out our message to. So, we'll continue to step through the code with 'next':

```
(gdb) next
48          fd = open(ttydata, O_WRONLY | O_NDELAY);
(gdb) next
```

```
Program received signal SIGSEGV, Segmentation fault.
0x40000031 in ?? () from /lib/ld-linux.so.2
(gdb)
```

Whoops! So, the segmentation fault occurs while trying to open the file ttydata. Because the context has switched to ld-linux.so.2 (which is where the open command is) the value of ttydata is no longer immediately obtainable (gdb provides ways to do this; but we won't be covering them here); to find out what it was, we will need to step through the program again and check out the value before it is passed to the open call. The optimal location to start checking is at the 'sprintf' command when ttydata is assigned, as it will only be tripped when the process is a viable user process, saving us a lot of time. That is line 44 of the code. So, remove the existing break points with the 'clear' command:

```
(gdb) clear 70
Deleted breakpoint 1
(gdb) clear 71
Deleted breakpoint 2
```

And add another at line 44:

```
(gdb) break 44
Breakpoint 3 at 0x80486f4: file hw2.c, line 44.
```

Now, run through the program again to the breakpoint:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/support/nthomas/src/hello/a.out

Breakpoint 3, massivehello (message=0x804881f "Hello World!\n") at hw2.c:44
44          sprintf(ttydata, "/dev/%s", utmp->ut_line);
```

Now, go ahead and let this command execute with the 'next' command and

evalute the result afterwards:

```
(gdb) print ttydata
$7 = 0x8049850 "/dev/tty1"
```

This value looks fine, so to find out what is occurring, you will need to step into the open command and see how the failure occurs:

```
(gdb) step
Cannot insert breakpoint 0:
Temporarily disabling shared library breakpoints:
0

Program received signal SIGSEGV, Segmentation fault.
0x40000031 in ?? () from /lib/ld-linux.so.2
```

Well, not much luck there; the failure occurs in a shared library and gdb can't let us know what is going on in there as things stand right now. We are able to link the symbol table from a shared object in with gdb, but there will be no debugging symbols, and we will simply have to step through each function manually. This library could be recompiled with debugging symbols and linked directly, but that would take a large amount of time and effort; so maybe it is possible to take the data we've gotten from the debugger so far and think about what it tells us might be happening.

It certainly appears that the value of ttydata is correct, so it is very odd that it would be causing a segmentation fault. Segmentation faults rarely have anything to do with what the actual data in a memory address is though; they have more to do with overwriting or deleting memory that is already assigned to another object in a program. Given that ttydata has just been written to by a sprintf command, and that ttydata is the variable we are passing into the open command that fails, it seems possible that there is an issue with the address space it is using to store its value in. So, to look at what is going on with the space for tty before it is written to by sprintf, run the program again in the debugger, with the same break point at line 44 and print out ttydata prior to its being assigned:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/support/nthomas/src/hello/a.out

Breakpoint 1, massivehello (message=0x804881f "Hello World!\n") at hw2.c:44
44          sprintf(ttydata, "/dev/%s", utmp->ut_line);
(gdb) print ttydata
$2 = 0x8049850 "\234\230\004\bh3\001@'\236"
```

That memory region looks like maybe it is being used by something else. We can use the 'x' command to look at what it is that the is at the address where this points - (0x8049850):

```
(gdb) x  0x8049850
0x8049850 <_GLOBAL_OFFSET_TABLE_>:  0x0804989c
(gdb) x 0x0804989c
0x804989c <_DYNAMIC>:   0x00000001
```

That certainly doesn't look like an empty character string, like we'd expect. It looks likely at this point that ttydata is using someone else's address space to try and store its data. Why could that be? Well, looking back at the source code, it may be obvious by now that the mistake in the coding was that the char pointer 'ttydata' was never initialized or assigned memory, so when data is written to it that data may well write over previously assigned memory. Because of the transient nature of these sorts of memory failures, as previously mentioned, you may not have experienced the failure in the same location, or even at all. However, this sort of memory-related bug is fairly common, and often requires the use of gdb to resolve. Had gdb not pointed us towards where the failure actually occurred in the code, it might have taken significantly longer to locate the mistake.

So, to rectify this we need to allocate space for ttydata, which stores a device name; most of the devices that are available under Linux have names that are 7 characters or less, but to be on the safe side (since this is only a single variable, and therefore doesn't account for much space), a size of 50 characters will be fine. So, add the following lines to the program above to allocate space and then free it for ttydata (the mistakes are not coded into the versions of the program shown at the end of this text- see the next section for more on this):

At line 25 (just under the variable declarations in massivehello()):

```
ttydata = (char*)malloc(50*sizeof(char));
```

At line 65 (just before the return() in massivehello()):

```
free(ttydata);
```

6. Compiling Multiple .c and .h Files

So, the original hello world program has expanded into a somewhat larger tool. Now, it needs to be broken up into separate header and c files for modularity (realistically, this might be unnecessary, but, again, it is beneficial for showcasing the development tools). Once it is broken

into pieces it will be more difficult to compile the program as a whole, and to manage the project effectively you will need to use the ’make’ utility.

First, here is the new structure of the hello world program:

- hw3.c: mainhello() calls to massivehello();
- hw3.h: massivehello() definition;
- ui1.c: main() calls to inputparser();
- ui1.h: inputparser() definition; error_handle() definition; spew_usage() definition; input_length() definition;

The sources for these files are displayed at the end of this paper (under names without the numbers included in them).

gcc needs to know the name of all the .c files it needs to compile together to form a single executable, so that they can share their symbols. To do this, simply add both of the .c files listed above to the gcc invocation:

```
[nthomas@daisy hello]$ gcc ui1.c hw3.c
```

This builds a file named a.out, overwriting the previous executables you have built in this directory. To avoid that you can give a name to gcc where it should write the file out to instead of a.out. This is done with the -o argument:

```
[nthomas@daisy hello]$ gcc -o miniwall ui1.c hw3.c
```

This command will create a new executable file named ’miniwall’.

There is another way to accomplish the same thing, however, which is by using gcc to create the object files for each source component of the final exectuable. To tell gcc to compile/assemble without linking to create the final executable, give it the -c argument:

```
[nthomas@daisy hello]$ gcc -c ui1.c
```

This creates an object file called ’ui1.o’.

```
[nthomas@daisy hello]$ gcc -c hw3.c
```

Again, this creates an object file, called ’hw3.o’

Now, to link the two together, give them as arguments to gcc. Here, gcc is also told to write the resulting executable out to the file ’miniwall’ by the -o argument.

```
[nthomas@daisy hello]$ gcc -o miniwall ui1.o hw3.o
```

7. Using `make` to Simplify the Build Process

It isn't a big problem in this case, but if you were dealing with a large development project and had hundreds of source and header files to work on, it would be very time consuming to have to rebuild them all every time you changed any one of them. The 'make' utility is designed to resolve this issue by keeping up with what files were changed when and only recompiling the files that have been changed or have dependencies that have been changed/recompiled. To use 'make' you must create a 'makefile' for it to read the dependencies from. This file is most commonly called 'makefile' or 'Makefile', but can be anything you want, as long as you tell 'make' where it is (make will find a makefile named 'makefile' or 'Makefile' be default). The syntax for the makefile isn't all that difficult...but as you string more and more dependencies along and begin using some of the more advanced features of the utility it can become fairly complex. If you are unable to get your makefile to build your project correctly, there are some debugging options available; for this and more info on make, read the man page, which covers all of the command line

arguments, but not the makefile syntax (check out www.gnu.org for more info). Following is a fairly simple makefile for miniwall:

```
miniwall : ui1.o hw3.o
        gcc -o miniwall ui1.o hw3.o

ui1.o : ui1.c
        gcc -c -w ui1.c

hw3.o : hw3.c
        gcc -c -w hw3.c
```

This does precisely the same thing as the few command line arguments to gcc shown above. By default, make tries to build the first package listed in the makefile. In this case, that is 'miniwall'. So, simply calling make in this directory will build miniwall. To tell it to build a particular component of the makefile, one must specify the specific target to build in the makefile (miniwall, ui1.o, or hw3.o in this case).

It is important to get the syntax in the makefile exactly correct. The first line of each package is the name of the file to build then ':' and then the list of dependencies. Then, there needs to be a carriage return to the next line. On the second line, you must tab over once and enter the command line syntax for the command used to build that file. If a tab isn't used, it will not work. Whatever command is put on the second line will be executed, even if it doesn't create the

file. If more commands than will fit on one line are needed, the line can be continued with '\' rather than a carriage return.

Now, use the make command to build the miniwall executable:

```
[nthomas@daisy hello]$ make
make: 'miniwall' is up to date.
```

Well, if you ran the gcc -c commands to build the .o files a moment ago and haven't edited the files since then make doesn't see any reason to rebuild them. So, to get make to rebuild miniwall, simply use the 'touch' command to update the access time on those files:

```
[nthomas@daisy hello]$ touch ui1.c hw3.c
[nthomas@daisy hello]$ make
gcc -c -w ui1.c
gcc -c -w hw3.c
gcc -o miniwall ui1.o hw3.o
```

When make went to build miniwall it saw that the sources for the two dependencies had been updated more recently than their associated object files and rebuilt them; once this was done, it saw that the object files were newer than the latest copy of miniwall and it decided to rebuild that as well. It is possible to 'trick' make into not realizing it should rebuild, by updating all of the files in question, so they still appear to be matched up:

```
[nthomas@daisy hello]$ touch ui1.c hw3.c miniwall ui1.o hw3.o
[nthomas@daisy hello]$ make
make: 'miniwall' is up to date.
```

In a case like this, it would be handy to have a 'make clean' option, where make would get rid of all of the old object files and start afresh on the build. This is also useful when cleaning up the build directory. Make does support such a function- it is done by adding the following lines to the end of the makefile:

```
.PHONY : clean

clean :
        rm *.o
```

You could just say:

```
clean :
	rm *.o
```

but as previously discussed, the name to the left of the colon is the name of the file that is checked for dependencies. While no file named 'clean' will be created by the rm command, what would happen if a file named clean already existed? Let's try it using a modified makefile:

```
[nthomas@daisy hello]$ cp makefile makefilebad
[nthomas@daisy hello]$ vi makefilebad
```

Here the file is edited to drop the 'PHONY' lines

```
[nthomas@daisy hello]$ make -f makefilebad clean
rm *.o
```

This worked fine and deleted all of the object files

Now, create a file named clean:

```
[nthomas@daisy hello]$ touch clean
```

Now remake the object and executable files:

```
[nthomas@daisy hello]$ make -f makefilebad
gcc -c -w ui1.c
gcc -c -w hw3.c
gcc -o miniwall ui1.o hw3.o
```

Now try and use the clean option:

```
[nthomas@daisy hello]$ make -f makefilebad clean
make: 'clean' is up to date.
```

So, make sees that the file clean is up-to-date (it has no dependencies) and doesn't execute the command associated with it and fails to remove the object files.

To avoid this, we use the .PHONY package name to tell make that there isn't a file to be associated with the 'clean' package name; thus it won't check, and it won't matter if there is a file named clean in the directory or not. Most clean statements also contain the name of any targets that can be created by the makefile that don't have a .o ending (in this case, 'miniwall'.

Thus, the final line would read 'rm *.c miniwall' and would delete the executable when clean was run. During development I often don't add this command on the final executable.

As previously stated, you can include a remarkable amount of logic in a makefile, and wind up with a very robust and flexible way to build large scale projects. If you have a complex development environment with many different compilation derivatives and large numbers of source files, it is recommended you delve into make a little more, by checking out the gnu web pages at www.gnu.org . If you want to see how complex makefiles can become, check out the 400+ line makefile for the kernel in /usr/src/linux/ (if you installed the kernel development packages).

8. Using the Concurrent Versions System for Source Management

So, now our little hello world program is a much more complex development project with multiple sources and a makefile to put it all together consistently. As it gains popularity, it is sure to attract developers from around the world who want to work on it; and it is now time to start looking into version control.

The standard version control system for Linux is called cvs, which stands for Concurrent Versions System. The man page for cvs is about 1500 lines long and contains a remarkable amount of information. As the man page states however, there are only about 5 commands you will need to know for basic operation. Of course, here you will be setting the cvs up for our project so there will be a few more commands to know, but it still isn't all that difficult. However, if you make a mistake in cvs you could destroy months of work, so it is certainly recommended you learn as much about cvs as you can, and the man page is a good place to start. www.gnu.org also has more extensive documentation on the subject.

Creating the cvs source repository (where the safe copies of your source will live) is easy.

To start with, you will need to set your CVSROOT environment variable so that when you call cvs it will know where to look for your source repository. It is possible to avoid this and simply tell cvs where to look when you invoke it, but this value usually stays the same for a given development system, so it saves a lot of time and effort to just set it in your .bashrc file. To do this, add the following lines to you .bashrc file:

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
```

This will tell all cvs commands that the source repository will live in /usr/local/cvsroot - obviously you could change this, but it is recommended that you make the last directory 'cvsroot' wherever you put the repository.

Remember that you will need to run 'source ~/.bashrc' to get this environment variable set in your current shell (or logout and log back in).

Now that this is set we can call the 'cvs init' command to create the repository:

```
[nthomas@daisy hello]$ source ~/.bashrc
[nthomas@daisy src]$ cvs init
cvs [init aborted]: cannot make directory /usr/local/cvsroot: No such
file or directory
```

Well...this highlights the fact that you will need write permissions on the directory where you want to put the cvsroot. So, either su to a user who can write to that directory (root, presumably) or put the repository in a place you can write to (somewhere in your home directory). Putting the tree in a central location facilitates widespread accesses better, so if you can, try not to put it in your home directory (it is also important to put it somewhere it isn't likely to get destroyed, and a home directory isn't necessarily the safest place in that regard).

```
[nthomas@daisy src]$ su -
Password:
[root@daisy /root]# cvs init
cvs init: No CVSROOT specified!  Please use the '-d' option
cvs [init aborted]: or set the CVSROOT environment variable.
```

We didn't alter the .bashrc for root, so cvs doesn't know where the cvsroot should be. We add the -d argument along with the directory where the cvsroot should go:

```
[root@daisy /root]# cvs -d /usr/local/cvsroot init
```

Now, if we look in /usr/local/cvsroot we can see that CVSROOT has been created there. This is the administration file for cvs where you can set a number of variables. However, all of the source trees will live as subdirectories right off of /usr/local/cvsroot.

It is the idea of cvs that you will never need to concern yourself with looking in the /usr/local/cvsroot/ subdirectories to deal with the source trees manually. However, there are a couple of cases where this can be important. The most common one is to set the permissions properly so that a given Unix group can gain access to that tree.

Anyone who is going to even read the files must have write permissions for the directory as it keeps track of all cvs accesses and must write a logfile in that directory.

What we will do is simply tell cvs where our working directory is for the miniwall program we're working on and it will copy all of the files automatically and set up the history information on their initial checkin. However, it will also assign the source tree a revision number, thus obsoleting our rudimentary numbering scheme for our files. In addition to this, we have multiple versions of the various files in our working directory which we don't want included in the cvs. So, let's create a

subdirectory, cvshello (or whatever you want) and copy the needed files into it, and drop any numbering information we've assigned to them.

```
[nthomas@daisy hello]$ mkdir cvshello
[nthomas@daisy hello]$ cp *.c cvshello/
[nthomas@daisy hello]$ cp *.h cvshello/
[nthomas@daisy hello]$ cp makefile cvshello/
[nthomas@daisy hello]$ cd cvshello
[nthomas@daisy cvshello]$ ls
hw1.c  hw2.c  hw3.c  hw3.h  makefile  ui1.c  ui1.h
[nthomas@daisy cvshello]$ rm hw1.c
[nthomas@daisy cvshello]$ rm hw2.c
[nthomas@daisy cvshello]$ ls
hw3.c  hw3.h  makefile  ui1.c  ui1.h
[nthomas@daisy cvshello]$ mv hw3.c hw.c
[nthomas@daisy cvshello]$ mv hw3.h hw.h
[nthomas@daisy cvshello]$ mv ui1.h ui.h
[nthomas@daisy cvshello]$ mv ui1.c ui.c
```

Now, of course, we need to change the .c files to reflect the change in name of the .h files they include as well as the makefile to relfect all of the name changes.

Be sure to attempt to build miniwall when you are done.

Assuming it builds and runs cleanly you will want to run 'make clean' and make sure to delete the miniwall executable as well, and we'll be ready to su back to root and create import the sources to the repository.

```
[root@daisy cvshello]# cvs -d /usr/local/cvsroot import -m "Imported
sources" miniwall RedHat start
N miniwall/hw.c
N miniwall/hw.h
N miniwall/ui.h
N miniwall/ui.c
N miniwall/makefile

No conflicts created by this import
```

You can see that we still had to use the '-d /usr/local/cvsroot' argument. Then, we told it to import, followed by '-m "Imported sources", which is simply the message we want the history files of the files we're importing to show (if we didn't specify this like this it would take us into

an editor automatically to type a message in). Then comes the name of the source tree, the name of the vendor, and the release tag (which you can take it on faith should be 'start' for this case).

To avoid making things too complicted, I'm just going to set the permissions to allow any user to have read/write access for the cvsroot and miniwall directory in /usr/local/ as well as the history file in /usr/local/cvsroot/CVSROOT - this would be a remarkably bad idea in practice.

Now, we'll drop back out to being our normal user persona and checkout the source code. When you check out the source tree it will create a directory by the name of 'miniwall' and put all of the sources into it. Once this is done, you really should remove your orignal working directory so that you don't get confused about having two sets of files around.

```
[nthomas@daisy src]$ cvs checkout miniwall
cvs checkout: Updating miniwall
U miniwall/hw.c
U miniwall/hw.h
U miniwall/makefile
U miniwall/ui.c
U miniwall/ui.h
```

Now CVS is aware that he have these files checked out and will make anyone else who wants to check these files out aware of this fact if they concurrently check the tree and out then try and submit changes after we have already done so. For example, we can log in as root and check the tree out as well:

```
[root@daisy /root]# cvs -d /usr/local/cvsroot checkout miniwall
cvs checkout: Updating miniwall
U miniwall/hw.c
U miniwall/hw.h
U miniwall/makefile
U miniwall/ui.c
U miniwall/ui.h
```

Now, say root decides to add 'miniwall' to the makefile clean target, as we mentioned earlier. Once the change is made, it can be committed with a 'cvs commit' command:

```
[root@daisy miniwall]# cvs -d /usr/local/cvsroot commit -m "Improved clean
target" makefile
Checking in makefile;
/usr/local/cvsroot/miniwall/makefile,v  <--  makefile
new revision: 1.2; previous revision: 1.1
```

```
done
```

Again, you see the -d argument and the -m followed by a message, and finally, the name of the file to commit.

Performing this operation has produced the first change in our version number. You can see from the output that we are now working on revision 1.2 instead of 1.1 as it was at the start.

Everything went smoothly here, because although nthomas has the tree checked out, no changes have been checked in. Now, nthomas will make a trivial change to the makefile and try to check it back in; this change will be to add '-f' to the 'rm' command in the clean target:

```
[nthomas@daisy miniwall]$ cvs commit -m "Improved clean target" makefile cvs
commit: Up-to-date check failed for ‘makefile’ cvs [commit aborted]: correct
above errors first!
```

This fails the 'Up-to-date' check, because the tree that nthomas has is now older than the tree in the repository and therefore should be updated with the newer repository version and then have the changes made to it and committed.

There are a number of mechanisms provided by cvs for notifying a user when a file is checked out and for doing reserved checkouts. These are not the default behavior though, and for now we will focus on more basic operation.

cvs does provide an 'update' command that will deal with concurrent checkouts and revisisions well by merging the new source in the cvs tree with your altered source (if you have not altered sources in your working directory this will behave just like a 'checkout' and simply put the latest cvs copies in your working directory. Here, though, it will help us fix our makefile discrepencies:

```
[nthomas@daisy miniwall]$ cvs update makefile
RCS file: /usr/local/cvsroot/miniwall/makefile,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into makefile
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in makefile
C makefile
```

It tried to do a merge, but it found conflicts. This basically means that when it went to merge it found that some of the sections that were modified in the latest cvs tree (committed by root) were the same sections where changes had been made in working directory. It can tell this by seeing that nthomas' working directory was built from the 1.1 source and that it had changed from that, but that it was also different from the 1.2 source.

So, since it couldn't merge the two sources, it copied the source in question to a backup file in the working directory name .#makefile.1.1.1.1 (which you can see with the 'ls -a' command).

The file 'makefile' will now include everything it did before plus the output of the diff generated by the merge:

```
miniwall : ui.o hw.o
        gcc -o miniwall ui.o hw.o

ui.o : ui.c
        gcc -c -w ui.c

hw.o : hw.c
        gcc -c -w hw.c

.PHONY : clean

clean :
<<<<<<< makefile
        rm -f *.o
=======
        rm *.o miniwall
>>>>>>> 1.2
```

The section beginning with '<<<<<<< makefile' is what was in my working directory. The section after the '=======' characters shows what's in the new cvs tree. It is left up to us what to do about this. Both of these changes could be nice, so we'll just edit the file to show

```
rm -f *.o miniwall
```

and commit it. (be sure to test it before you commit it!)

```
[nthomas@daisy miniwall]$ cvs commit makefile
Checking in makefile;
/usr/local/cvsroot/miniwall/makefile,v  <--  makefile
new revision: 1.3; previous revision: 1.2
done
```

You can see that I didn't use the -m "message" option, so I had to enter the message in a text editor when it was brought up. Upon exiting the editor (which is where it says 'done'), it finished the cvs commit.

Once you have done your commit, it is recommended that you release your working directory (this can be an issue with some cvs configruations). To do this, use the 'release' command:

```
[nthomas@daisy src]$ cvs release miniwall
You have [0] altered files in this repository.
Are you sure you want to release directory 'miniwall'
```

Believe it or not, those are all of the basic things you will need to know to set up and use a simple version control system. cvs is a very full featured system however, and to take advantage of what it has to offer it is suggested that you do quite a bit more research into it. One of the things you will likely need to know fairly quickly if you're doing an open-source project with cvs is how to do remote access; this is not terribly complicated, and is covered in depth in the www.gnu.org documentation on cvs.

9. That's It!

So, you should now know how to do all of the basic elements of development under Linux. There are a very large number of options out there, but the tools discussed here are by far the most common; knowing them will serve you well. As has been mentioned many times throughout this paper, there are lots of resources with information about these tools; with the basic knowledge you have now you should be able to read and understand any of them to expand your understanding. Read 'man' pages, check the 'HOWTO' files, and look things up on Linux-related web pages.

If you have any questions/comments/criticisms feel free to email

nthomas@redhat.com

Nathan Thomas

Technical Developer

Red Hat, Inc.

10. Source Listings

Following are the texts of the various source files discussed in throughout the text:

10.1. hw.c:

```
#include "hw.h"


int mainhello(char *message)
{
   int retval = 0;

   retval = massivehello(message);
   error_handle(retval);

   return(retval);

}
```

10.2. hw.h

```
#include <string.h>        /*For the strcmp call*/
#include <unistd.h>        /*symbolic constants- O_WRONLY and
O_NDELAY here*/
#include <stdio.h>         /*Standard input and output for our
writing out*/
#include <utmp.h>          /*UTMP functionality- try 'man utmp'
for more info*/
#include <fcntl.h>         /*file control for handling file
descriptors*/
#include <pwd.h>           /*for access to password/user
information*/


#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <setjmp.h>


/*This is our primary function- it takes in the string to
output*/
```

```
int massivehello(char *message)
{
   int uid;                /*current user's id number*/
   static int fd;          /*file descriptor id*/
   FILE *fp;               /*file pointer*/
   char *ttydata, *user;   /*buffers for terminal info, and
username*/
   struct passwd *pwd;     /*password data structure*/
   struct utmp *utmp;      /*process information structure*/


   ttydata = (char*)malloc(50*sizeof(char));


   uid = getuid();         /*get the user's id number*/
   pwd = getpwuid(uid);    /*get the current process id (this
program!)*/


   setutent();             /*go to the beginning of the process
list*/


   user = pwd ? pwd->pw_name : "root";  /*figure out who is
running this*/



   /*go through the process list and for each entry our 'user'
owns, output*/
   /*our message*/


   while ((utmp = getutent()) != NULL)
   {
      if(utmp->ut_type != USER_PROCESS || /*make sure it's a user
process*/
         utmp->ut_user[0] == 0 ||         /*and their is a valid
user*/
         (strcmp(utmp->ut_user, user)))   /*and that user is our
'user'*/
      {
         continue;          /*skip this entry if any of that
stuff failed*/
```

```
      }
      sprintf(ttydata, "/dev/%s", utmp->ut_line);

      /*ttydata is the name of the terminal to write to*/

      fd = open(ttydata, O_WRONLY | O_NDELAY);
         if ((fp = fdopen(fd, "w")) != NULL)
         {
            fputs(message, fp);
            fputs("\n", fp);
            fclose(fp);
         } else
         {
            close(fd);
         }
   }


   endutent();                         /*close the utmp file*/

   free(ttydata);

   return(0);                          /*exit with no errors*/
}
```

10.3. ui.c:

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

int main(int argc, char **argv)
{
   int retval = 0;
```

```
   int string_length = 0;
   char *message;

   string_length = input_length(argc, argv);

   message = ((char*)(malloc(string_length*(sizeof(char)))));

   retval = parser(argc, argv, message);
   error_handle(retval);

   retval = mainhello(message);
   error_handle(retval);

   free(message);

   return(retval);
}
```

10.4. ui.h:

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

void error_handle(int retval)
{
   if (retval)
   {
      fprintf(stderr, "Error: exiting\n");
      exit(1);
   }
   return;
}

int spew_usage(char* argv_0)
{
   fprintf(stderr, "Usage: %s [MESSAGE]\n", argv_0);
```

```
   fprintf(stderr, "Outputs MESSAGE to all terminals owned by
caller\n");
   fprintf(stderr, "With no MESSAGE \"Hello World!\" is
output\n");
}

int input_length(int argc, char **argv)
{
   int counter = 1;
   int string_length = 0;

   if (argc == 1)
   {
     return(strlen("Hello World!\n"));
   }

   while (counter < argc)
   {
      string_length += (strlen(argv[counter]));
      counter++;
   }

   string_length += argc;  /*to account for spaces*/

   return(string_length);
}



int parser(int argc, char **argv, char *message)
{
   int retval = 0;
   int counter = 2;

   if (argc == 1)
   {
```

```
      strcpy(message, "Hello World!\n");
      return(retval);

   }

   if (!(strcmp(argv[1], "--help")))
   {
      spew_usage(argv[0]);
      return(retval);
   }

   strcpy(message, argv[1]);
   strcat(message, " ");

   while (counter < argc)
   {
      strcat(message, argv[counter]);
      strcat(message, " ");
      counter++;

   }

   return(retval);
}
```

10.5. makefile:

```
miniwall : ui.o hw.o
        gcc -o miniwall ui.o hw.o

ui.o : ui.c
        gcc -c -w ui.c

hw.o : hw.c
```

```
        gcc -c -w hw.c

.PHONY : clean

clean :
        rm -f *.o miniwall
```