Writing a Shell

Shell Shocked

This month Steven Goodwin will explain how to write a shell, and overcome the obstacles involved. After all, it's shell programming without a shell!

BY STEVEN GOODWIN

Esther Keller, visipix.com

To the outside observer, Linux is one process. To everybody else, it is an orchestrated blend of kernel, modules and tools. Each one dedicated to an individual task. No more. No less. This approach extends to the login procedure. The moderately friendly login prompt (requesting a user name and password) is provided by a different program from the moderately friendly command prompt (that lets you execute programs). This prompt is called a shell. And because it is just another program, we can change it, add to it, or replace it completely with our own. Which is what we're going to do!

Windmills of Your Mind

The shell is a command interpreter that allows you (as the user) to enter commands that Linux will, in turn, execute. Conceptually, it is very simple. You type a command. The shell executes it. The underlying operating system takes care of the complexities such as checking file permissions, loading the file, updating the process table and handling the SUID bit. In return, the shell takes responsibility for redirection, wildcard parameters, controlling pipes and much, much more. For a brief run-down, see Box 1: Some Shell Features.

Box 1: Some Shell Features

The maintenance of environment variables like PS1 and PATH. The latter being an ordered list of directories, which the shell searches to find programs.

Internal commands. Some commands, like *cd*, *export* and *history*, are implemented by the shell code itself. This improves speed since there is no disk access.

Wildcard expansion. To convert *rm *.jpg* into *rm file1.jpg file2.jpg file3.jpg*, for example.

Inline editing. This includes tab completion, and key shortcuts (like Ctrl+A or Ctrl+E) to jump to the beginning, or end, of the current line, respectively.

Redirection and pipes. Any of the three basic streams, input, output and error, can be redirected from, or to, files and through pipes.

Since the shell is replaceable, many programmers have sought to replace it with their own version. Like the multitude of Linux distributions, each shell attempts to fill a particular role. Some (like ash) are very small, with a minimal set of features, and intended for rescue disks. Others, like csh or ksh, provide very powerful scripting capabilities for more complex work.

The most popular Linux shell is called bash, and stands for Bourne Again SHell. It was originally written by the GNU project, and is a free (and updated version) of the Bourne Shell, written by Stephen Bourne. It has become the de-facto standard in Linux due to its comprehensive feature set, powerful editing options and, no doubt, its role in the kernel build process. We hope to fill the multimedia control niche. So let us start writing *mmshell*!

If I had a hammer

mmshell will provide basic multimedia functionality, such as MP3 playback, volume control, and CD access, using the external tools outlined in Table 1: Tools. With such a restricted set of features we shall not need complex inline editing, or a command history buffer. Instead, we shall adopt a simple menu-style interface so the user doesn't need to type any commands, or remember their specific command line switches. *mmshell* will be demonstrated with a specific multimedia user called *music*.

We shall begin by developing a traditional menu system. Then, we will study the ways in which normal programs differ from shell programs, and how to overcome them, finishing up with the correct way to install and use our new shell.

Joyride

Most programmers have, at one time in their lives, written a menu system. They're not big, and they're not clever. Literally. A good menu system should be small and stupid, as to provide the lowest bar to entrance. Our stand-alone menu program might appear as in Listing 1.

From a programming perspective, this is very simple. You may need to change the device switches for the *cdcd* and *aumix* programs, but even they need very little explanation. I probably don't even need to include the compiler arguments. But I will!

```
gcc mmshell.c -o mmshell
```

However, since this program is being used as a shell, there are other considerations.

Table 1: Tools

Tool	Functionality
mplayer	An impressive audio and video playback tool which supports multiple codecs, including several Windows ones. If codec support isn't important, mpg123 may be more suitable, as mplayer can cause output problems requiring a terminal reset.
cdcd	Command line-driven CD player. Very useful. Also supports cddb and CD index. If no configuration file (.cdcdrc) is found in the users home directory, the program enters interactive mode to retrieve these settings
aumix	Lightweight mixer with basic functionality.

Orinoco Flow

Let us first consider the basic structure, as this is very similar regardless of whether it is run as a shell or stand-alone program. After all, the shell is just a program, so we'd expect nothing less. The program starts with our *main* function as normal, including the executable's file name as argv[0] (but prefixed with a minus sign, when run as a shell), and ends (logging the user out in the process) when the *main* function completes. In order for the bash shell to reach this point we have to type the internal command 'exit', which does the equivalent of our 'Quit' menu option.

Our only extra consideration here is to trap the SIGINT signal. This occurs when the user hits Control + C. When running as a normal program, our menu would just close down and we'd have to start it again. But since this is now a shell, it would also log us out. So, before that happens we must close any open files and stop any music playing. The reason for closing files becomes clear when using lock files, as mentioned in Box 2: Aces High. The trap must be set up with a suitable callback function, like so:

```
01 /* Added to main function */
02 signal(SIGINT, CloseHandler);
03
04 /* The parameter allows this➥
   handler to be re-used to ➥
   handle different signals */
05 void CloseHandler(int Signal)
06 {
07    puts("Exiting...");
08    /* Stop all music here */
09    exit(0);
10 }
```

Instead of handling the Control + C signal, we could ignore it completely. This effectively disables it, for which we use:

```
signal(SIGINT, SIG_IGN); /* ide➥
ntical to sigignore(SIGINT); */
```

It is very likely that you'll want to catch other signals, too. If we were writing a mathematical shell, for example, the floating-point exception (SIGFPE) would need to be handled. A complete list of signals can be found in */usr/include/bits/signum.h*, although not all of these can be caught (for example, SIGKILL and SIGSTOP).

All Woman

A much bigger implication of shell programming concerns the *system* function, which, believe it or not, loads its own shell to execute the command given! The shell in question is *sh*, and usually a symlink to bash. While this is not necessarily an issue in itself (save the extra time to load another shell), it can become problematic if you're writing a shell for a confined environment, as you will also need a copy of bash on the disk.

Instead of *system*, we need to use a combination of *vfork* and *execvp*. The first of these functions splits the current program into two identical processes. The first half of the fork (called the parent) continues to run as the normal shell, while the second half (called the child) executes our external program using the second function, *execvp*. A fork is needed because *execvp* replaces the existing process, making it impossible to continue running our shell. Splitting our existing process into two, means one half can live when the other half becomes expendable. See Listing 2.

This also gives us the bonus feature of knowing the process ID of the spawned program. We can then use this ID to kill the process (and thus silence the music), should the user hit Control + C.

```
kill(process_id, SIGKILL);
```

execvp is one of the many exec*() function variations that exist. All are documented in *man exec*, which is a recommended read.

Listing 1: Stand-alone menu

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 void PrintMenu(void)
05 {
06  puts("Menu Options\n");
07  puts("1. Play CD");
08  puts("2. Stop CD");
09  puts("3. Play MP3s");
10  puts("4. Mute");
11  puts("5. Unmute");
12  puts("9. Quit");
13 }
14
15 int GetUserOption(void)
16 {
17 char opt[16];
18
19  printf("Option: ");
20  fgets(opt, sizeof(opt), stdin);
21  opt[sizeof(opt)-1] = '\0';
22  return atoi(opt);
23 }
24
25 void ProcessMenu(int iOption)
26 {
27  switch(iOption)
28  {
29  case    1:
30    system("cdcd -d /dev/cdrom
   play");
31    break;
32  case    2:
33    system("cdcd -d /dev/cdrom
   stop");
34    break;
35  case    3:
36    system("mplayer mp3s/*.mp3
   &>/dev/null");
37    break;
38  case    4:
39    system("aumix -d /dev/mixer
   -v0");
40    break;
41  case    5:
42    system("aumix -d /dev/mixer
   -v99");
43    break;
44  }
45 }
46
47 int main(int argc, char
   **argv)
48 {
49 int iOpt;
50
51  puts("Multimedia Shell -
   v0.0\n");
52  do
53  {
54    PrintMenu();
55    iOpt = GetUserOption();
56    ProcessMenu(iOpt);
57  }
58  while(iOpt != 9);
59
60  return 0;
61 }
```

Box 2: Aces High

Although the file permissions will prevent non-authorized users from accessing the CD and soundcard, we might want to limit use of the *mmshell* to a single instance. This would prevent two different users issuing conflicting requests, and could be implemented by the shell refusing to run for any user, other than *music*.

We could also create a simple *lock* file in the */home/music* directory. This file would only need to exist to indicate that a user was logged in, and prevent *mmshell* accepting new commands or allowing additional logins from that user. The lock file would be removed when the user logged out.

It is here that the necessity for handling signals (like SIGINT) becomes apparent. If, for example, the user hit Control+C while in *mmshell* and the signal *wasn't* handled, the lock file would never be removed, and no one could log in as *music* again, until *root* deleted it manually.

We can call our custom spawn function like so:

```
char *args[] = { "cdcd", "play"
, NULL };
    ForkAndExec("cdcd", args);
```

Here we must give the program name in two places: the executable filename, and the first argument. This is a convention, whereby argument zero is always the program name. It doesn't have to be the programs real name (since some might be symlinks), but there is no harm in doing so. Being C programmers, we're used to the program name being passed to us as argv[0].

The filename doesn't need a full path, since the *evecvp* version will search the */bin* and */usr/bin* directories automatically, if a path is not included.

Without Me

When we removed the *system* call from our shell, we lost more than a wasted call to bash. We lost a friend! Everything that the shell provides (refresh your memory with Box 1: Some Shell Features, if necessary) has been lost. For instance, the redirection (that hides *mplayer*s excessive output), the environment variables (especially PATH), and the wildcard expansion. The latter omission means we can no longer use *.mp3 for our playlist. If we tried, there would simply be an error saying, 'File *.mp3' not found. Solving this can either be done correctly (by enumerating each file in the directory, using the opendir-readdir-closedir trio), or resorting to the *system* command. Space prohibits the full solution to be listed here, unfortunately, but the source is available on the Linux Magazine web site at *http://www.linux-magazine.com/issue/40/mmshell2.c*

Another shell feature that is sorely missed is the environment. Every variable we've come to depend on when programming within a shell, is suddenly absent when programming without one. Naturally enough, the *libc* system designers had a solution to this. In fact, they had two.

The first solution involves changing our *execvp* function to *execle*. This function takes the usual program arguments, albeit in a different format, and postfixes them with a NULL-terminated array of environment variables.

```
execle("/usr/bin/cdcd", "cdcd",
 "play", NULL, ppEnviroVars);
/* Note: full path required */
```

The second solution relies on a global variable! It is called *environ*, and has the same format as *ppEnviroVars* above. It is used on all the other, non-*execle*, forms of the exec*() function.

The only other omission we shall mention here is the search path. We've already seen how *execvp* will check the two most common directories for executables. *execplp* does the same. However, none of the other variations do. This is not normally an issue however, because it is very sensible to include complete paths for any executable we spawn. This is not an ecumenical matter, but a security one.

Captain Swing

Depending on your viewpoint, security is either a fantastic challenge and playground of endless possibilities. Or it is the biggest administration hassle of your day. Whatever your opinion, however, it is certainly the most necessary. Even programmers are not immune from having to consider security. After all, it is the programmers that put the security bugs in the software in the first place! But what constitutes a security bug?

A brief glance at any of the security advisories (such as BugTraq) or Linux Magazine's own Insecurity News, will highlight a number of software issues. By far the most common bug is one that fits the template:

Version X of program Y has a buffer overrun exploit, enabling rogue input to execute arbitrary commands as the current user.

While this is obviously a very bad state of affairs for servers that run as root, or have access to the wheel group, it might appear unimportant to us. After all, it is just a simple shell. If we're using it, we must be authorized to use this machine anyway, where we could do

Listing 2: Splitting in two

```
01 void ForkAndExec(const char
   *pName, const char **ppArgs)
02 {
03 pid_t process_id;
04
05  /* We use vfork instead of
   fork because it's quicker */
06  /* when you only intend the
   child to call execvp */
07  process_id = vfork();
08
09  switch(process_id)
10  {
11  case    -1:
12    printf("ERROR! Could not
   spawn %s...", pName);
13    return;
14
15  case    0:
16    /* The child is born to
   die, after been replaced */
17    /* by another program. */
18    execvp(pName, ppArgs);
19    exit(0);
20
21  default:
22    /* The parent process */
23    printf("Spawned PID %d\n",
   process_id);
24  }
25
26 return process_id;
27 }
```

worse damage with bash and the *rm* command! Right? Wrong! Every security problem is a problem. If we're using *mmshell* in a restricted environment (perhaps to operate a kiosk machine), users won't have the authority to change the shell. A security hole might allow this, giving crackers a foothold allowing them to climb higher.

Security inside the shell can be achieved by solid programming techniques: compiling with every pedantic warning option your compiler supports, using tools such as Splint [1] and Flawfinder [2], and exhaustive testing. You need to check your bounds (easier if you avoid functions like *sprintf* and *strcpy*), validate any user input (checking for, and removing, escape sequences) and avoiding any software you don't trust. The latter point is difficult to ensure once you execute another program, since control has now left the shell. Creating secure kiosk systems becomes difficult because a lot of external software (like email clients and editors) allow you to 'shell out' to bash.

Once you have found, or written, a secure external program, you can safely execute it. One simple step to maintain this level of security is to execute it using its full path. This gives no room for misinterpreting the location of 'mplayer', for example, and executing a rogue, Trojan, program by mistake or intent. Naturally, the executable in question should also be in a secure directory that can only be written to by root. To confirm the path of an executable in use, try typing:

```
$ which mplayer
/usr/local/bin/mplayer
```

Secure programming is a vastly complex topic. And many good words have been written about it. Some them are available at [3] and [4].

Sleeping in my car

Having completed our development, we must now test it. To do this, we will create a separate user called *music*, and add it to the *audio* group.

```
# adduser --ingroup audio music
```

By making sure the *audio* group has the appropriate permissions to control the CD player, and access the soundcard, we will have a fully functional test user. Since the shell is very restrictive, it is not wise to use it on our own account, lest we err and need to log in as root to fix it.

Testing is a two-stage process. First, we need to log on as the user *music* with the bash shell, and use the program as a normal menu system. This will confirm we have the right permissions set up, and our signals are working. Then we need to configure the *music* user to load and run our new shell after login.

Although Linux permits normal (i.e. non-root) users to change their shell, it does not permit them to add their own shells to the system. The only permitted shells are those explicitly listed in the */etc/shells* file. This is to maintain security and stability of the system. Users trust their shell to be discrete, and not log every key press. Even if the user had logged into her computer using a secure shell (such as ssh), a Trojan horse-like shell could still monitor those keypresses once ssh had passed control over to the user's default shell. Fortunately, only root can change the permitted shells.

Having made a careful audit of the new shell code, we can copy it into the */usr/bin* directory, and add its name to the list of acceptable shells. You can do this as any user, so long as it's root!

```
# cp mmshell /usr/bin
# echo /usr/bin/mmshell >>/etc➥
/shells
```

We then need to tell the login prompt that we wish to use this shell, instead of bash. Although this can be done by changing the /etc/passwd file directly, this is not recommended. Better instead to use the *chsh* program. This works as any user, although only root can change the shell of other users.

She Sells C-Shells, on the C-Shore

Of course, not everyone is a C programmer. It is perfectly feasible to write your own shell in Python, Perl or Ruby as the programming rules, and method of installation, are exactly the same. In reality, of course, shells are rarely written in any language other than C. But that doesn't normally stop anybody from doing so, as can be seen at [5].

```
$ chsh -s /usr/bin/mmshell
```

This will take effect the next time we log in. So, type 'exit', and log back in. If everything is working properly (and there's no reason why it shouldn't – although you may need to amend the paths of individual programs like *aumix* or *mplayer*) you will see:

```
mymachine login: music
Password:
Linux tori 2.4.19 #44 SMP ➥
Sun Dec 28 19:07:54 GMT ➥
2003 i686 unknown
Last login: Sun Jan 4 ➥
14:32:26 2004 from mymachine

Multimedia Shell - v0.1

Menu Options

1. Play CD
2. Stop CD
3. Play MP3s
4. Mute
5. Unmute
9. Quit
```

And there you have it! With this basic functionality in place, you could easily upgrade *mmshell* to include new, or different features according to your desires. Or you could write a completely new shell. Perhaps a kiosk system for email and web browsing, or an interactive teletext system using *alevt* and a TV card. My original idea (and the basis for this article) was *sleepshell* that enabled me to choose which type of relaxation music to play at bedtime: forest, streams or ambient effects. You may have better ideas. In fact. I'd count on it! ■

INFO

[1] Splint: *http://www.splint.org/download.html*

[2] Flawfinder: *http://www.dwheeler.com/flawfinder/*

[3] Secure Programming: *http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html*

[4] Quality coding: *http://www.linux-magazine.com/issue/27/CodeTesting.pdf*

[5] Perl shell: *http://sourceforge.net/projects/psh/*