1 UNIX Commands

One of the most attractive features of UNIX is its extensibility. Even a novice UNIX user can add commands to the operating system, and those commands will be immediately accessible, and can be accessed in the same way system commands are accessed, namely by typing the name of the command. This is because when the UNIX shell encounters a command name, it searches through a set of directories known as the search path until it encounters a file with the name of the command it encountered. If that file has execute permissions appropriately set, then it is executed. (Recall that to set the execute permission for a file, the appropriate UNIX command is `chmod +x filename`).

Two types of files can serve as commands on UNIX systems. First, a text file which has appropriate execute permissions is interpreted as a collection of commands. By default, these commands are executed by the Bourne shell, `/bin/sh`, allowing . A curious mechanism is used to inform the operating system of an alternative interpreter for commands within an executable file; if the first line of the file is of the form

```
#! program_name
```

then `program_name` will be used to interpret the commands instead of the Bourne shell. Since that line would have to be ignored under other circumstances, this technique will work for any program which uses the pound sign (`#`) as a comment delimiter. Such programs include all the UNIX shells (`/bin/sh`, `/bin/csh`, `/bin/ksh`, etc.) as well as the perl (usually `/usr/local/bin/perl`) text processing language.

The second type of file which is executable is a binary file created as the result of compiling a program in some low-level language like C or Fortran. The operating system recognizes such files by the so-called "magic number" which is a bit pattern inserted at the beginning of the file when it is first created, and which is different for each different type of file which is supported on the system. (See the man page for the `file` command or the `magic` file format for more information about magic numbers.) Since the compiler or loader deals with this detail, you will rarely have to worry about it. Executable text and binary files can be freely mixed in directories without causing any confusion.

If you write a command of your own, and you want it to automatically execute when you type its name, it must be found in one of the directories included in your search path. Usually your search path will be set in the start-up file for your shell (either `.cshrc` for the C shell or `.profile` for the Bourne or Korn shells). Under the Bourne and Korn shells, the search path is stored in the shell variable `PATH`, while in the C shell it is stored in the shell variable `path`. You can see your search path by typing the UNIX command `echo $PATH`. The `bin` subdirectory of your home directory is usually on your search path, and is a good choice for storing commands which you write. Remember that there are some built-in commands which the shell will always find before it goes to the search list; one common example which sometimes causes problems is the `test` command. You would be well advised to not name your command files `test`, since the shell will generally find the built-in command first. Alternatively, you can always refer to a UNIX command by its full pathname; in particular, to execute a file called `commands` in the current directory, you can type

```
./commands
```

at the UNIX prompt, and there will be no possibility of confusion as to which file you want to execute. One word of caution if you use the C shell; the C shell doesn't rescan commands in its search path after it is first started unless you type the command `rehash`. So if you are using the C shell, and you add a command to a directory in your search path, you need to type `rehash` before the C shell will recognize it in an already running shell.

The simplest command files consist of a sequence of commands exactly like you would type them in at the command line. When the command is executed, the individual commands will be executed, one after the other. This can be a good way to execute commands which need to be done in a specific order, or to execute multiple runs of another command, because it is inefficient to run several invocations of a program at once, as opposed to running them serially.

2 Command Line Arguments

Although creating a command file to execute a series of commands can be useful, the real utility of UNIX commands is that you can pass information to them easily through the command line. UNIX commands generally follow a common format for the command line. While some UNIX commands require their arguments in a specified order (like the commands `cp` or `mv`), most UNIX commands offer more flexiblity by observing the convention that options are single letters, preceded by hyphens (`-`), and may or may not be followed by optional arguments. Even common commands, like `ls`, which lists the files in a directory, may have many options, without interfering with their simple default behavior. If you're writing your own UNIX commands, it's easy to access the information on the command line in a way similar to most standard commands. The rest of this article will provide ways of reading the command line arguments for shell scripts, perl programs and C and Fortran programs.

For the sake of example, suppose we have a program which will extract one or more lines from a file. We wish to accept a starting line number, an ending line number, a flag indicating whether to print the line number along with the line, and finally a file name. Thus there are two options with required arguments, which we'll call `-s` and `-f` for start and finish, respectively, and an option which simply serves as a flag, which we'll call `-n` for numbered.

3 Arguments in Shell Scripts

The examples provided here are for Bourne or Korn shell scripts; similar techniques are available for C shell scripts. Individual arguments to shell scripts are denoted as `$0` (the name of the command), and `$1`,...,`$9`, and the entire set of arguments is denoted `$*`. The simplest way to process arguments in a shell script is to loop over the set of arguments and use a `case` statement along with the `shift` statement, which removes arguments from the `$*` variable. The shell variables `start`, `finish`, `num` and `file` will be used to hold the extracted information.

```
#!/bin/sh

start=1
finish=1
num=0
file=""
```

```
for i in $*; do
  case $i in
  -s ) start=$2
       shift
       shift ;;
  -f ) finish=$2
       shift
       shift ;;
  -n ) num=1
       shift ;;
   * ) filename=$1 ;;
  esac
done

if test "$filename" = ""
then
   echo "No filename given.  Exiting ..."
   exit
fi
```

The two semicolons ;; end each clause of the case statement. For options with required arguments, the value is extracted from the next argument on the list ($2), and the argument list is shifted twice. For options serving as flags, only one shift is needed. In the example above, only the filename was checked; it general, it's a good idea to check to make sure all the arguments were presented properly.

4 Arguments in Perl Programs

There are several perl subroutines which are available to parse command line options. One convenient one is `&Getopts`, found in the file `getopts.pl`, which should be part of your distribution. You pass `&Getopts` a string which contains the letters you wish to recognize as valid options, following those which have required arguments with a colon (:). `&Getopts` removes the options from the argument list (which is stored inside perl as the `@ARGV` array), and sets variables of the form `$opt_letter` to appropriate values when the flag represented by `letter` is found on the command line. If a particular option is not given on the command line, the corresponding `$opt_letter` value is undefined. Thus, the previous example could be written as follows in perl:

```
#!/usr/local/bin/perl

require 'getopts.pl';

&Getopts("s:f:n");

$start = $opt_s ? $opt_s : 1;
$finish = $opt_f ? $opt_f : 1;
$num = $opt_n ? 1 : 0;

die "No filename given" if (! $ARGV[0]);
$filename = $ARGV[0];
```

Since `&Getopts` removes arguments from the `@ARGV` array, the filename will be the first element (index 0) of that array after the arguments are processed.

5 Arguments in C Programs

Command line arguments are passed to C programs through two optional arguments to the `main()` function; `argc` is an integer giving the number of arguments (using the convention that the 0th argument is the command name), and `argv` is an array of character strings containing the arguments themselves. The memory allocated to the strings in `argv` will not be reused, so their addresses can be safely copied into local variables. Here is one example of how to process the command line arguments of the previous example:

```
#include <stdio.h>

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

```
int i,start=1,finish=1,num=0;
char *filename=NULL;

i = 1;
while(i < argc){
      if(argv[i][0] == '-')goto doopt;
      filename = argv[i];
      i++;
      continue;

doopt:
      if(argv[i][1] == 's')
              sscanf(argv[++i],"%d",&start);
      else if(argv[i][1] == 'f')
              sscanf(argv[++i],"%d",&finish);
      else if(argv[i][1] == 'n')
              num = 1;
      else fprintf(stderr,"(%s)Unrecognized option, %s\n",
                          argv[0], argv[i]);
      i++;
      }

if(filename == NULL){
      fprintf(stderr,"(%s)No filename given\n");
      exit(1);
      }
                  . . .
```

6 Arguments in Fortran Programs

As a final example, the fortran subroutine `getarg` and function `iargc` will be used to read command line arguments inside a fortran program. The fortran `read` statement is used to convert the character string read by `getarg` to a number where needed.

```
      character*20 filename
```

```
      character*20 rdarg
      character brkarg(20)
      equivalence (rdarg,brkarg)
      integer start,finish,num

      start = 1
      finish = 1
      num = 0
      narg = iargc()
      do while (i.le.narg)
          call getarg(i,rdarg)
          if(brkarg(1) .ne. "-")goto 5
          if(brkarg(2) .eq. "s")then
                call getarg(i + 1,rdarg)
                read(rdarg,*)start
                i = i + 2
                goto 10
                end if
          if(brkarg(2) .eq. "f")then
                call getarg(i + 1,rdarg)
                read(rdarg,*)finish
                i = i + 2
                goto 10
                end if
          if(brkarg(2) .eq. "n")then
                num = 1
                i = i + 1
                goto 10
                end if
 5        filename = rdarg
          i = i + 1
10        end do
                   . . .
```