Developing a Linux command-line utility

Best practices and thoughtful coding make for solid command-line tools

Vasudev Ram (vasudevram@yahoo.com)
Independent software consultant

01 June 2002

Learn how to write Linux command-line utilities that are foolproof enough even for end users. Starting with an overview of solid command-line best practices and finishing with a comprehensive tour of a working page-selection tool, this article gives you the background you need to begin writing your own utilities.

This article illustrates how to write a Linux command-line utility, similar to standard commands such as cat, ls, pr, mv, and so on. I've chosen a utility called selpg, which stands for SELect PaGes. Selpg allows the user to specify a range of pages to be extracted from the input text, which could come from a file or another process. It is modeled on de facto conventions for creating commands in Linux, including:

- Working standalone
- Working as a component in a pipeline of commands (by reading either standard input or a filename argument, and writing to standard output and standard error)
- Taking command line options that modify its behavior

I developed selpg for a customer some time ago. I subsequently posted it on a UNIX mailing list, and many of the members told me they found it to be a useful tool.

The utility reads text input, either from the standard input or from a filename given as a command line argument. It allows the user to specify a range of pages from this input that are then to be output. For example, if the input contains 100 pages, the user can specify printing only pages 35 to 65. This has a practical benefit as it avoids wasting paper when the selected output is to be printed on a printer. Another example is when the original file is very large, and was printed earlier, but certain pages did not print correctly because the printer jammed or had some other problem. This tool can be used in such cases to print only the needed pages.

Apart from containing a real-life example of a Linux utility, this article has the following features:

- It illustrates the power of the Linux environment for software development.

- It shows appropriate use of some system calls and library functions from C, including fopen, fclose, access, setvbuf, perror, strerror, and popen.
- It implements thorough error checking of the sort that should exist in a utility meant for general use, rather than as a one-off program.
- It gives warnings about potential problems, such as buffer overflows that can occur when programming in C, and gives advice about how to prevent them.
- It shows how to do hand-coded command-line argument parsing.
- It shows how to use the tool in pipelines and with redirection of input, output, and error streams.

Command-line guidelines

A writer of a general-purpose Linux utility should follow certain guidelines in the code. These are recommendations that have evolved over time and help ensure that the utility can be used in a more flexible way, particularly with regard to cooperation with other commands (whether built-in or user-written) and with the shell -- this cooperation being one of the keys to the power of Linux as a development environment. The selpg utility illustrates all of the guidelines and features below. (Note: In the examples that follow, the "$" sign represents the shell prompt and is not to be typed.)

Guideline 1. Input

Input should be allowed to come from either:

- A filename specified on the command line. For example:
  ```
  $ command input_file
  ```
  In this case the command should read the input_file.
- Standard input (stdin), which is by default the terminal (the user's keyboard). For example:
  ```
  $ command
  ```
  Here, whatever the user types until a Control-D (end of file indicator) becomes the command's input.

But stdin can also be redirected to come from a file, using the "<" (redirect standard input) operator of the shell, like so:

```
$ command < input_file
```

Here, the command reads its standard input, but the shell/kernel has redirected it so that it comes from input_file.

Stdin can also come from the standard output of another program, using the "|" (pipe) operator of the shell, as follows:

```
$ other_command | command
```

Here the standard output (stdout) of other_command is passed transparently (by the shell/kernel) to the stdin of command.

Guideline 2. Output

Output should be written to standard output (stdout), which is again, by default, the terminal (the user's screen):

```
$ command
```

In this case, the output of command goes to the screen.

Again, it can be redirected to a file, using the ">" (redirect standard output) operator of the shell:

```
$ command > output_file
```

Here, command still writes to its stdout, but this is redirected by the shell/kernel so that the output goes to output_file.

Or, the command's output can become the standard input for another program, again using the "|" operator like this:

```
$ command | other_command
```

In this case, the shell/kernel arranges for the output of command to become the input for other_command.

Guideline 3. Error output

Error output should be written to standard error (stderr), which is by default again the terminal (the user's screen):

```
$ command
```

Here, any error messages that occur when running command will be written to the screen.

But, using redirection of stderr, errors can also be redirected to go to a file. For example:

```
$ command 2>error_file
```

In this case, the normal output of command goes to the screen, whereas any error messages are written to error_file.

It is possible to redirect both stdout and stdin to different files, like so:

```
$ command >output_file 2>error_file
```

Here, stdout goes to output_file while anything written to stderr goes to error_file.

It is also possible to redirect stderr to the same place to which stdout has already been redirected. For example:

```
$ command 2>&1
```

In this case, the notation "2>&1" means "send stderr to wherever stdout is directed to," so both error and normal messages will go to the screen. Of course, this is redundant, since the simple invocation

```
$ command
```

will do the same. But this feature is useful when stdout has already been redirected to another source, on the same command line, and you want stderr to go to the same destination. For example:

```
$ command >output_file 2>&1
```

In this case, stdout has first been redirected to output_file; hence the "2>&1" will cause stderr to also be redirected to output_file.

Guideline 4. Execution

It should be possible to run the program either standalone, or as part of a pipeline, as shown in the examples above. This can be restated as follows: the program should operate the same, irrespective of its source of input (a file, a pipe, or the terminal) and of its output destination. This allows for maximum flexibility in how it is used.

Guideline 5. Command-line arguments

If the program is such that it can behave differently based on its input or on user preferences, then it should be written to accept command line arguments known as *options,* which allow the user to specify which behavior is to be used on this invocation.

Command-line arguments that are options are indicated by a prefixed "-" (hyphen). The other kind of arguments are those that are not options, in other words, they do not really modify the behavior of the program, but are more like data names. Typically they stand for filenames on which the program will operate, though this need not be the case; the arguments could stand for something else like a printer destination or job-id (see "man cancel" for an example).

Non-option arguments (those not prefixed by a hyphen), which may represent filenames or anything else, should come at the end of the command, if present at all.

Typically, if a filename argument is specified, the program takes that as its input. Otherwise it reads from stdin.

All options should begin with a "-" (hyphen). An option can have an argument attached to it.

A syntax diagram for a Linux utility would look like this:

```
$ command mandatory_opts [ optional_opts ] [ other_args ]
```

where:

- `command` is the name of the command itself

- `mandatory_opts` is a list of options that must be present for the command to work at all
- `optional_opts` is a list of options that may or may not be specified, as per the user's choice; however, it could be that some of them are mutually exclusive, as in the case of the "-f" and "-l" options of selpg (for which see below).
- `other_args` is a list of other arguments to be processed by the command; this can be anything at all, not just filenames.

In the above definitions, the term "list of options" means a series of options separated by spaces, tabs, or a combination of both.

Parts of the above syntax shown in square brackets can be left out (in which case you have to leave the brackets out too).

Individual options can look like this:

```
-f (an option alone)
-s20 (an option with an attached argument)
-e30 (an option with an attached argument)
-l66 (an option with an attached argument)
```

Some utilities use a slightly different format for options with arguments in which the argument is separated from its option by white space -- for example, "-s 20" -- but I've chosen not to do this as it complicates the coding; the only benefit is that it makes the command a bit more readable.

The above are actual options that selpg supports.

Selpg program logic

As stated earlier, selpg is a utility to select a range of pages from text input. This input can come either from a file specified as the last command line argument, or from standard input if no file name argument is given.

Selpg first processes all of its command line arguments. After scanning all option args (that is, those prefixed by a hyphen), if it finds one more arg, it takes it to be the name of the input file and tries to open it for reading. If there is not another arg, selpg assumes input is from stdin.

Argument processing

"-sNumber" and "-eNumber" mandatory options:
Selpg requires the user to specify the start and end pages of the range to be extracted, using two command line arguments, "-sNumber" (for example, "-s10," indicating start at page 10), and "-eNumber," (for example, "-e20," indicating end at page 20). It does sanity checking on the page numbers given; in other words, it checks that both numbers are valid positive integers and that the end page is not less than the start page. These two options, "-sNumber" and "-eNumber" are mandatory and have to be the first two arguments on the command line after the command name selpg:

```
$ selpg -s10 -e20 ...
```

(... is the remaining part of the command, as described below).

"-lNumber" and "-f" optional options:
Selpg can operate on two kinds of input text:

Type 1: Pages of this type of text have a fixed number of lines. This is the default type, so no option has to be given to indicate it. That is, if neither a "-lNumber" nor a "-f" option is given, it is understood that the pages of are fixed length, 72 lines each.

This was chosen as the default as it is a very common page length on line printers. The idea is to make the most common usage of the command the default, so the user doesn't have to type more options than needed. This default can be overridden by the "-lNumber" option, like this:

```
$ selpg -s10 -e20 -l66 ...
```

which indicates that pages are of fixed length, 66 lines each.

Type 2: Pages of this type of text are delimited by the ASCII form-feed character -- value 12 in decimal -- denoted by "\f" in C. The advantage of this format over the fixed-number-of-lines-per-page format is that it saves space on disk if there is a lot of variation in the number of lines per page and there are many pages in the file. After the lines containing text, Type 2 pages only need one character -- a form feed -- to indicate the end of the page. Printers recognize the form-feed character and automatically advance the print head by the required number of lines necessary to begin the next line of data on a new page.

Contrast that with Type 1, in which the file has to contain PAGELEN - CURRENTPAGELEN newlines to advance the text to the next page, where PAGELEN is the fixed page size and CURRENTPAGELEN is the number of actual text lines on the current page. In this case, the printer actually has to print that many newlines in order to make the print head move to the top of the next page. This is inefficient not only in terms of disk space but also in terms of printer speed (although the actual difference might not be much).

The Type 2 format is indicated by the "-f" option, as follows:

```
$ selpg -s10 -e20 -f ...
```

This tells selpg to look for form feed characters in the input and treat them as the page delimiters.

Note: the "-lNumber" and "-f" options are mutually exclusive.

"-dDestination" optional option:
Selpg also allows the user to send the selected pages directly to a printer, using the "-dDestination" option. Here, "Destination" should be the name of a print destination as accepted by the "-d" option of the lp command (see "man lp"). The destination should exist -- selpg does not check for this. To check that this option has worked, after running the selpg command with the "-d" option, run the command "lpstat -t". This should show a print job added to the print queue for "Destination." If a printer is currently connected for that destination and enabled, the output should

appear on the printer. This feature is implemented using the `popen()` system call, which allows one process to open a pipe to another process, either for output or input. In this case we open a pipe to the command

```
$ lp -dDestination
```

for output, and write to that pipe instead of to stdout:

```
selpg -s10 -e20 -dlp1
```

This sends the selected pages as a print job to the lp1 print destination. You should see a message like "request id is lp1-6". This message is from the lp command; it shows the print job id. If you quickly run the command `lpstat -t | grep lp1` just after the selpg command, you should see the job in the queue for lp1. If you wait for some time before running the lpstat command, you might not see the job, since it disappears from the queue once it is printed.

Input processing

Once all the command line arguments are processed, the actual processing of the input starts, using the options and input and output sources and targets specified.

Selpg keeps track of the current page number as follows: If input is fixed-lines-per-page, it counts newlines until the page length is reached and increments a page counter. If input is form-feed-delimited, it counts form feeds instead. In either case, as long as the condition that the page counter value is between the start and end pages holds true, it outputs the text (by line or by char). When that condition is false, that is, the page counter is either less than the start page or greater than the end page, it does not write any output. Voila! you get the pages you wanted on the output.

Commentary on the code

It's time to take a detailed look at the source. At this point, you may want to download selpg.c if you haven't already (see Resources). I'll take you through the code a piece at a time.

Lines starting with the comment "==== includes ====="

These lines specify the header files needed.

stdio.h is the header for the C standard I/O library. It is needed for the file opening, closing, reading, and writing functions (`fopen()`, `fclose()`, `fgets()`, `getc()`, and so on), for the `printf()` family of functions, and for the `setvbuf()` function.

stdlib.h is needed for the `atoi()` function (ASCII to integer), which converts a string to an integer.

string.h is for the string functions such as `strcpy()` and `strcmp()`.

unistd.h is needed for the `access()` function.

limits.h is needed for the definition of INT_MAX, which specifies the highest possible value for an int on your compiler/OS/hardware platform. Using this instead of a hardcoded value improves portability of the code.

assert.h is for the `assert()` debugging macro.

errno.h is needed for the declaration of errno, the global system call error number variable (more on that below).

Lines starting with the comment "==== types ====="
There is only one type defined, the `selpg_args` structure. A pointer to a variable of this type is passed to the `process_args()` function, and on return, the variable contains values obtained from arg processing. A `typedef` is used to give the type a shorter name of sp_args.

We also define a macro INBUFSIZ, which is the size of a character array used as a buffer while reading the input. This is for better performance.

Lines starting with the comment "==== globals ======"
`Progname` is a global char* variable to save the name by which the command was invoked, for display in error messages. This way, even if you rename the selpg command to something else, the new name will be shown in messages; you won't have to modify the code.

Lines starting with the comment "==== prototypes ==="
These lines declare the function prototypes of all functions in the code, as per ANSI C conventions. This is standard practice nowadays and helps the compiler to detect type mismatches between function definition/declaration and function usage.

The main() function
This is simple: It declares the variables needed, sets the fields of struct `sa` to defalt values, then calls function `process_args()` with the variables `ac`, `av`, and `&sa`. `ac` stands for "argument count" and contains the number of command line arguments, *including* the command name itself; `av` stands for "argument vector" and is a pointer to pointer to char; it contains all the command line arguments as an array of character strings. `&sa` is a pointer to a structure of type `sp_args`. After `process_args()` returns, the parsed values of the arguments are in the structure `sa`; we pass this variable to the function `process_input()`, which does the work of selecting the needed pages and writing them to the specified destination.

If at any point in the code, an error occurs such that processing cannot continue, we retrieve the system error message (if any), and display it along with our own message. Then we call the `exit()` function with an error code; for this utility, we've chosen to return different numbers for every different error condition. This is not a must, however; some utilities simply return 1 for any error condition and 0 for success, whereas others classify errors into categories, and return one of a smaller range of codes, say 1, 2, or 3, depending on the category. The only convention prescribed is that 0 should be returned for success and a nonzero value for failure. For more information on errors that can occur in system calls, see the section System call errors.

Since all errors cause an exit, if we return from the `process_input()` function, it means there was no error, so we return 0 from `main()`.

System call errors

When calling Linux system calls (or C library functions), as when calling your own functions, it is possible that errors will occur. Per the convention for C functions, in the case of success it's typical for the return value of the function to give some information about the result, and in the case of failure, the return value gives information about the reason for failure. There is a convention that is generally followed for this.

The meaning of the return value in case of a successful function or system call is routine-dependent; for example, `read()` returns the number of bytes actually read (as opposed to the number requested, which is one of the arguments to `read()`), whereas `fopen()` returns a pointer to a "struct FILE".

Here, we are concerned with the return value for unsuccessful calls.

Most calls return either -1, NULL, or EOF to indicate error. (As we've seen above, `fgets()` returns NULL while `getc()` returns EOF.) They also set a global int variable called errno. This is declared in errno.h as "extern int errno;". This int is an index into a table of system error messages, called `sys_errlist` (an array of strings), whose highest element has index `sys_nerr - 1`. Both `sys_errlist` and `sys_nerr` are also declared in errno.h.

When an error occurs in a system call, you can access and display a message corresponding to the error in at least one of two ways: by using either the `perror()` or the `strerror()` function. (Both of these are demonstrated in selpg.)

The `perror` function (see "man perror") accepts a single string argument, which is provided for customizing the message. It writes your string argument to stderr (standard error), followed by a colon and a space, then the system error message followed by a newline.

The `strerror()` function (see "man strerror") is slightly different. It allows you to access the system error message as a string. It requires an integer argument, `errno`, which should be the actual value of the global `errno` variable. You can access this variable by its actual name by including the header errno.h, which should be included in all C programs that use system calls (or library functions that invoke system calls). `Strerror()` returns the system error message as a string. There are conditions to be followed while using these functions. See their man pages for details.

Selpg makes extensive use of these functions to give proper messages to the user. I've also provided a small utility called showsyserr (see Resources), which allows you to see the system error messages. It can be run in one of two ways, either with a single command line argument, which should be an `errno` value, or without any arguments. If passed an argument, it will display only the message corresponding to that arg. If no arg is given, it displays all defined error messages.

By running this utility without arguments, you can use it as a tool to learn what the different error messages are. It can also be used in some cases as a useful debugging tool: Some Linux (or at least UNIX) utilities (and even daemons) do not display error messages, only the `errno` value. In this case, simply run showsyserr with that `errno` value and it will show you the corresponding error message. I've used this tool in troubleshooting UNIX problems in which there were a constant stream of messages on the console, messages such as "Panic - error 2". The source of the message was obviously a poorly-written module -- it should have given a more meaningful message -- but I was able to debug it by running showsyserr with the value 2 as the argument. It turned out that the file systems were full due to a rogue process writing continuously to the disk.

Once I knew it was a disk space problem, I was able to identify the process and kill it using other commands, such as ps and kill.

The process_args() function
Since this is a utility meant for general purpose use, not just by a small group of people, we do a lot of error-checking to make it as robust as possible.

Lines starting with the comment "==== check the command-line arguments ==="
We check whether the minimum number (three) of command line arguments has been passed. These three args are:

- The command name itself
- The `-sNumber` option
- The `-eNumber` option

If there are less than three args, we print a message and exit. Note the call to the `usage()` function; this lets the user know the correct way to call the utility. This is again a convention that should be followed when writing general purpose utilities.

Lines starting with the comment "handle 1st arg"
The comment "!!! PBO" is to indicate a Possible Buffer Overflow, a condition one should always check for in C programs, whether on Linux or any other platform. This can happen if the string copied into `s1` is longer than the allocated size of `s1`, which is BUFSIZ (a constant defined in stdio.h). The possibility of this is low, unless by accident or intention, but it pays to check. Buffer overflows are the cause of many a security bug. The correct way to do this would be to use the `strncpy()` function instead of `strcpy()`, and give the maximum number of bytes to copy as BUFSIZ - 1. Then, we would have to add a NULL character after the last char copied into s1.

The same procedure would have to be followed wherever the PBO comment occurs.

Here we copy the current arg, indexed by `argno` (which was initially set to 1, to point to the first actual arg after the command itself) to string var `s1`. Then we check that the first two chars of `s1` are "-s", to indicate the start page option. If not, do an error exit.

If it's okay, convert the remaining chars in `s1` into an integer. Check if it is a valid integer for this platform, using INT_MAX. (Remember, we have to use INT_MAX, not a hardcoded value, because the size of an int in C can vary by platform -- even among different compilers for the same platform).

If okay, save the start page in a field of the struct pointed to by `psa`.

Lines starting with the comment "handle 2nd arg"
Similar checks and actions are done as for the 1st arg. The only extra thing is that we check that the end page given is not less than the start page.

Lines starting with the comment "now handle optional args"
Here, we handle the remaining args, if any were given. Pay attention to the coding idiom used here, as it is typical of the style used in Linux. What we do is, in a loop, march down the argument list checking each one to see if it is an option. If so, the switch finds out which option it is. If valid, it sets a flag in the struct. We also set variables to remember the values of any data associated with the option, such as the page length in the case of the "-l" option and the print destination in the case of "-f". If the option is not valid, give an error message and exit.

Lines starting with the comment "there is one more arg"
After all the args prefixed with a "-" are processed, check if there are any remaining args. For selpg, there can be at most one such arg, which is taken to be the filename for input. We check if it exists and is readable using two calls to the `access()` function. This is actually redundant since the second call also checks for existence (a file cannot be readable if it does not exist), but I wanted to show that one can check for existence alone -- see "man access ()."

Lines starting with the comment "check some post-conditions"
Here we make use of C "assertions." This is a feature of the C language, not of Linux as such, but I show it because it is such a useful facility for producing reliable code.

See the section Design by contract for details on this.

The process_input() function
We first declare some variables.

Lines starting with the comment "set the input source"
If a filename arg was not given on the command line, we use stdin. Otherwise, we open the specified file for reading. If it cannot be opened, we do an error exit. Otherwise, we call the `setvbuf()` function to set a buffer of size INBUFSIZ (earlier #defined as 16KB) for reading from `fin`. This is for faster reading of the input. 16KB has been chosen as the size since, when I conducted experiments with different sizes ranging from 1KB up to 64KB, I found that the performance improved to some extent as I went in steps from a buffer size of 1KB to 16KB, but there was no significant improvement after that. Your mileage may vary, and you may want to experiment with changing the buffer size to find the best size for your Linux system. As an exercise, you can modify the code so that selpg accepts one more argument, say "-bNumber" - to indicate the buffer size. This would allow you to experiment without having to change the value of the #define macro and recompile the code each time. This could become one more optional option, and a default buffer size of whatever value you prefer could be used if this option is not specified, as I've done for the "-l" number option.

Design by contract

Assertions are very helpful during testing and debugging. The effect of the assert macro is to check if its Boolean argument is true (in other words, nonzero); if so, nothing is done; if it is false, the program terminates with a message saying that the assertion failed. The Boolean condition that caused the assertion to fail is also displayed. Assertions can be turned off by defining the NDEBUG macro, either by editing the source to add a line "#define NDEBUG" (no value for the #define need be given), or on the fly while compiling by including the command line switch "-DNDEBUG" on the cc or gcc compiler command line, like this:

```
$ cc -DNDEBUG -o selpg selpg.c
```

Assertions can be used anywhere in the code, but two good places to use them are at the entry and exit of functions to check for pre-conditions and post-conditions that you know should exist if the code is correct. That is, if a function has a certain set or range of valid values for its arguments, an assertion to this effect can be placed as the first line of code in the function. For example, if a function to calculate the square root can only handle non-negative numbers, you can have an assertion that the input argument is >= 0. This means that it is the responsibility of the calling function to ensure that only non-negative arguments are passed. If a negative value is given, the assertion will trigger and abort the program with a message that the condition "input argument >= 0" failed. Similarly, at the end of the function, if you know that there are certain post-conditions that should hold, write assertions corresponding to each. This ensures that you get to know if, due to a bug in the code in this function, the conditions that you expect to be true on exit, are not.

A good way to use assertions is to build two versions of your final app -- one with assertions disabled (call it the release version), and the other with assertions enabled (call it the debug version). Initially, the release version is given to users. If users are at other sites, the debug version can be given as well, but kept in a separate directory. When a problem is found in the app, tell them to temporarily replace the release version with the debug version and try to simulate the problem again. This time, the assertions will likely cause a message to appear that will help pinpoint the source of the bug.

Making debug and release versions can be automated by the make utility. See the accompanying makefile (in Resources later in this article) for a simple example of this. Make is a utility to automate the process of building an app from one or more files. It can handle dependencies, such as the rule that if a source file is modified, its corresponding object file must be regenerated by recompiling the source. It can also do a lot of other things.

Lines starting with the comment "set the output destination"

If no "-dDestination" option was given, we write to stdout. Otherwise, we try to open a pipe to the lp command, using the command string "lp -dDestination". This is done with the system call `popen()`, which can open to a pipe to another process, either for reading or writing. In this case, we open a pipe in write mode to lp. What this means is that all standard output from our selpg process will go to the standard input of the lp process. If `popen()` fails, do an error exit.

Lines starting with the comment "begin one of two main loops"

We execute one of two loops, depending on the type of input.

If the page type is fixed-lines-per-page, we read input line by line using the `fgets()` library function. (It returns NULL, defined in stdio.h, for either error or EOF [end-of-file]; after the loop, we check for which one occurred.) If not NULL, increment the line counter. Then check if the line count exceeds the page length. If so, bump the page counter up by 1 and reset the line counter to 0. Then check to see whether the page counter is within the requested range of pages, and if so, write the current line; otherwise do not write. Repeat the loop until `fgets()` returns NULL. Note: `fgets()` reads at most BUFSIZ - 1 chars from `fin` and stores them in the char string line. It appends a NULL character so that line contains a properly null-terminated C string.

The logic used for form-feed-delimited pages is roughly the same, but slightly simpler, as there is no need for a line counter: Read char by char using the `getc()` library function. This returns EOF on error or end-of-file. Check each char read to see whether it is a form feed; if so, increment the page counter. As in the fixed-lines-per-page case, write output only if the page is within the specified range. Repeat the loop until `getc()` returns EOF.

You may be wondering how the call to `setvbuf` helps, when we are reading either line by line or char by char. Well, here's how: Since we are using the stdio library, the lower level routines in the library read data from the disk a chunk at a time -- where the size of the chunk corresponds to the buffer size we've specified -- and put it into our buffer inbuf. Then, whenever we call either `fgets()` or `getc()`, the data is passed into our program variables (`line` or `c`). Since reading data from the disk in larger chunks is more efficient, and passing data from one location in memory (`inbuf`) to another (our program variables) is comparatively very fast, our I/O is faster -- at least in theory. In practice, buffering may occur at various other levels, such as on the hard disk itself, the controller, or the kernel disk drive device driver, so ours may not make much of a difference. Our use of `setvbuf()` may even make the code run slower. The moral of the story is that performance tuning is a tricky and complex business, and any changes should always be tested to see whether they make a difference.

Lines starting with the comment "end main loop"
We check whether either the start page or the end page number was greater than the total number of pages found and, if so, give an appropriate message. Then check whether any error occurred on the input stream and, if so, give a message. Finally, close the input stream, flush the output stream, and if it was a pipe, close it with the `pclose()` function. This will have the effect of sending an EOF to the lp process, which will in turn terminate.

If the output stream was stdout, and if it was not redirected to a file or a pipe, the selected pages will appear on the screen.

If the output stream was stdout, and if it was redirected to a file, the selected pages will be in that file.

If the output stream was stdout, and if it was piped to another process (at the command line level), the selected pages will become the input to that process. For example, you could pipe the output to the pager "less" so that you can view the output one page at a time and scroll back and forth.

If the output stream was piped to lp (from within the program, using the "-dDestination" option and `popen()`), the selected pages will be printed on the given print destination (assuming that it is enabled).

Using selpg

The following gives examples of workable `selpg` command strings, to show how the principles we've reviewed are put into practice by the end user:

1. `$ selpg -s1 -e1 input_file`
   This will write page 1 of "input_file" to stdout (the screen), since there is no redirection or piping.
2. `$ selpg -s1 -e1 < input_file`
   This does the same as example 1, but in this case, selpg is reading its stdin, which has been redirected by the shell/kernel to come from "input_file" instead of an explicitly named filename argument. Page 1 of the input is written to the screen.
3. `$ other_command | selpg -s10 -e20`
   The stdout of "other_command" is redirected by the shell/kernel to the stdin of selpg. Pages 10 through 20 are written to selpg's stdout, the screen.
4. `$ selpg -s10 -e20 input_file >output_file`
   Selpg writes pages 10 to 20 to its stdout; this is redirected by the shell/kernel to "output_file."
5. `$ selpg -s10 -e20 input_file 2>error_file`
   Selpg writes pages 10 to 20 to its stdout, the screen; any error messages are redirected by the shell/kernel to "error_file." Note that there must not be any white space between the "2" and the ">"; this is part of the shell's syntax (see "man bash" or "man sh").
6. `$ selpg -s10 -e20 input_file >output_file 2>error_file`
   Selpg writes pages 10 to 20 to its stdout, which is redirected to "output_file"; whatever selpg writes to stderr is redirected to "error_file." This invocation can be used when "input_file" is large; you don't want to sit around waiting for selpg to complete, and you want to save both the output and the errors.
7. `$ selpg -s10 -e20 input_file >output_file 2>/dev/null`
   Selpg writes pages 10 to 20 to its stdout, which is redirected to "output_file"; whatever selpg writes to stderr is redirected to /dev/null, the null device, which means that error messages are thrown away. The device /dev/null discards all output written to it, and when read from, returns an EOF immediately.
8. `$ selpg -s10 -e20 input_file >/dev/null`
   Selpg writes pages 10 to 20 to its stdout, which is thrown away; error messages appear on the screen. This could be used for the purpose of testing selpg, when you may only want (for some test case) to see the error messages, not the normal output.
9. `$ selpg -s10 -e20 input_file | other_command`
   The stdout of selpg is transparently redirected by the shell/kernel to become the stdin of "other_command," to which pages 10 thru 20 are written. An example of "other_command" could be lp, which would cause the output to be printed on the default printer for the system. It could also be wc, which would result in a display of the number of lines, words, and

characters contained in the selected range of pages. And it could be any other command that can read from its stdin. Error message still go to the screen.

10. `$ selpg -s10 -e20 input_file 2>error_file | other_command`
    Similar to example 9 above, with one difference: Error messages go to "error_file."

In any of the examples above involving redirection of stdout or stderr, replacing the ">" with a ">>" will cause the target file to be appended to (with the output or error data), rather than being overwritten, if it already exists. If it does not exist, it will be created and then the data will be written to it.

All the examples shown below can (with one exception) also be combined with redirection or piping as shown above. I'm simply not adding those features to the examples below, because I think there are enough occurrences of them in the examples above. The exception is that you cannot use output redirection or piping in any selpg invocation that contains the "-dDestination" option. Actually, you can still redirect or pipe stderr, but not stdout, since there will not be any standard output -- it's being internally piped to the lp process using the `popen()` function.

1. `$ selpg -s10 -e20 -l66 input_file`
   This sets the page length to 66 lines, so selpg will treat the input as though it is demarcated into pages of that length. Pages 10 thru 20 are written to selpg's stdout, the screen.
2. `$ selpg -s10 -e20 -f input_file`
   Pages are assumed to be delimited by form feeds. Pages 10 thru 20 are written to selpg's stdout, the screen.
3. `$ selpg -s10 -e20 -dlp1 input_file`
   Pages 10 through 20 are piped to the command "lp -dlp1" which will result in the output being printed on printer lp1.
4. One final example to show another feature of the Linux shell:
   `$ selpg -s10 -e20 input_file > output_file 2>error_file &`
   This makes use of a powerful feature of Linux, the ability to run a process in the "background." What happens in this case is that a "process id" (pid) such as 1234 will be displayed, and then your shell prompt will appear almost immediately, allowing you to type more commands to the shell. Meanwhile, the selpg process runs in the background, with stdout and stderr both redirected to files. The advantage is that you can continue doing other work while selpg runs.
   You can check to see whether it is still running or has completed by running the command ps (for Process Status). This will display a few lines, one for each process started from this shell session, including the shell itself. If selpg is still running, you'll see an entry for it, too. You can also kill the running selpg process by the command "kill -15 1234". If that doesn't work, try using "kill -9 1234". Warning: read "man kill" before trying this command on anything important.

System calls vs. library functions -- and a bit of history

What's the difference between a Linux system call and a C library function? Here are a few:

First, both are "functions" in the original C meaning of the word, which is: a separately defined piece of code that is defined once and can be called by name (any number of times from anywhere else in the code where it is in scope), can be passed arguments, and can return a value.

Interesting historical note: A large part of the Linux OS is written in C, and C is the "native" language of Linux, both for OS-level programming (as in writing the kernel and device drivers) and Linux system programming, which is about writing apps (such as an RDBMS, a network tool, or a custom business app) that make use of the system calls.

This is one of the reasons why Linux is such a powerful OS and development environment. As a developer, you get access to a great amount of OS functionality -- memory management, process management, file and directory management, device management, networking, threads, etc. -- for free and in a transparent way. All you need to do is include the header files that declare the calls you want to use and (in some cases) link your code with the libraries that implement them.

This happened partly as a historical accident, because UNIX and C were originally developed by almost the same group of people at Bell Labs. In fact, C was developed as a high-level language for system programming, specifically for writing operating systems. Most operating systems until then had been written in assembly language, whereas from the second (or so) version onwards, UNIX was mostly written in C, except for a very small percentage of hardware-dependent code that had to be written in the assembly language of the target platform.

The ability to write most of the OS in a high level language was one of the reasons why UNIX/Linux spread so much and became so successful. All other things being the same, studies have shown that the productivity of programmers, in terms of lines of code per day, is roughly the same irrespective of what level of language they code in, whether in assembly language or a high level language. On average, one line of high level language code translates into many lines of assembly language code. Hence, it takes less time to write the same amount of functionality in a high level language than in an assembly language, not to mention the fact that code written in higher level languages is much easier to debug and maintain.

Second, a Linux system call is basically a function that is a part of the operating system's code. It could be in the kernel core per se, or in a device driver that is linked into the kernel either statically or dynamically. So when you invoke a system call from your user code, say from function `foo()` in your program, you are actually invoking a routine in the kernel. This causes what is known as a context switch from "user space" to "kernel space," although that's just for you information. In terms of the way it is called, there is no difference between a system call and a library function. A C library function is not in itself a part of the OS code. It essentially runs in user space, but part of its implementation may run in kernel space. See the next point.

Third, a C library function may or may not have an associated underlying system call. Some functions need to use a system call in order to do their work, others do not.

An example of a library function that does not have an associated system call is the `strlen()` function declared in string.h. No OS functionality is needed; the implementation of this function simply walks down the string, incrementing a counter for each character, until it reaches the NULL character that terminates the string. Remember, in C, a string is nothing but an array of characters with a terminating NULL character (ASCII 0). It then returns the value of the counter. This is totally OS-independent code and works the same way on all C platforms, Linux or others.

An example of a library function that does have an associated system call is the `fopen()` function declared in stdio.h. This function is a high-level way of opening a file; but, since the only way to open a file in Linux is to use the `open()` system call, `fopen()` makes use of `open()` in its implementation. Similarly, functions such as `printf()` and `fprintf()` invoke the `write()` system call to do their work.

Conclusion

This article should give you enough of a jump start to begin writing Linux command-line utilities on your own, but if you want more information for further study, see the suggestions in the Resources below.

Resources

- Download the source code mentioned in this article:
  - selpg.c, the C code for selpg
  - showsyserr.c, the C code for the helper utility
  - makefile, a makefile to build selpg
- Read the man pages for `stdio`, and for all the system calls and functions used in this article. To learn more, be sure to read the other man pages described in the SEE ALSO section of each man page.
- Read the man page for `getopt()`, `man 3 getopt()`. The "3" here is the section of the man pages; section 3 is for C library functions. The "3" has to be given since there is also a getopt in section 1 (Commands) of the man pages, and if you don't specify the section, it shows the first, by default. I've used manual parsing of command-line arguments in this article, since it was relatively simple, in this case, and to demonstrate the technique. However, for more complex argument parsing, you may want to check out the `getopt()` function. It could save you some work.
- Read the man pages for `assert()`, `lp`, `cancel`, `lpstat`, `ps`, and `kill`.
- Read the man pages for `pipe()`, `dup()`, `open()`, `close()`, `exec()`, and `fork()`. `popen()` is implemented using `pipe()`, `fork()`, and `exec()`. Caution: using `pipe()` is complex compared to `popen()`! Consult a good book that shows the right way to use it, such as *Advanced UNIX Programming* by Marc J. Rochkind (Prentice-Hall, Inc., 1986).
- Read the man page for the shell you use: bash, csh, or whatever. Learn to use the powerful features of the shell at the command line, in scripts, and in interaction with your own programs. This will enhance your software development productivity a lot.
- Download and look at the source code for some Linux utilities besides the one presented here.
- Read the book *The UNIX Programming Environment* by Brian W. Kernighan and Rob Pike (Prentice-Hall, Inc., 1984). Though somewhat dated, its still one of the best introductions to UNIX and Linux for users as well as developers. It shows, like no other book I've come across, the power of this OS. It covers basic usage, commands, filters (another name for command line utilities), sed, awk, grep, shell programming, and C programming in UNIX. A masterpiece and a must-read.
- If you haven't already, read *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie(Prentice-Hall, Inc., 1988). It is the bible for the C language. If you find it tough going, try a book more oriented to beginners -- but be sure to read this one later. It will help you master the fluent use of C. It has a chapter or two on C/UNIX integration. It also has many examples of command-line utilities, argument processing, and so on. Another masterpiece and must-read.
- For a solid overview of Linux fundamentals, including coverage of a number of command-line utilities, read our series of LPI exam preparation tutorials.
- Need to get proficient with bash? Check out "Bash by example, Part 1, Part 2, and Part 3 (*developerWorks*, March -- April 2000).
- Find more Linux articles in the *developerWorks* Linux zone.

About the author

Vasudev Ram

Vasudev Ram is an independent software consultant with experience in project management, software development methodology, C, UNIX, Java, databases, and UNIX/Linux administration. He has worked at HCL Hewlett-Packard (now HCL Infosystems) as a systems engineer and consultant, and with Infosys Technologies Limited as a systems analyst and project manager. He is interested in comments from *developerWorks* readers. You can contact Vasudev at vasudevram@yahoo.com.