



"Argument list too long": Beyond Arguments and Limitations

By Alessandro Naro

Created 2002-05-09 01:00

Four approaches to getting around argument length limitations on the command line.

At some point during your career as a Linux user, you may have come across the following error:

```
[user@localhost directory]$ mv * ../directory2
bash: /bin/mv: Argument list too long
```

The "Argument list too long" error, which occurs anytime a user feeds too many arguments to a single command, leaves the user to fend for oneself, since all regular system commands (`ls *`, `cp *`, `rm *`, etc...) are subject to the same limitation. This article will focus on identifying four different workaround solutions to this problem, each method using varying degrees of complexity to solve different potential problems. The solutions are presented below in order of simplicity, following the logical principle of Occam's Razor: If you have two equally likely solutions to a problem, pick the simplest.

Method #1: Manually split the command line arguments into smaller bunches.

Example 1

```
[user@localhost directory]$ mv [a-l]* ../directory2
[user@localhost directory]$ mv [m-z]* ../directory2
```

This method is the most basic of the four: it simply involves resubmitting the original command with fewer arguments, in the hope that this will solve the

problem. Although this method may work as a quick fix, it is far from being the ideal solution. It works best if you have a list of files whose names are evenly distributed across the alphabet. This allows you to establish consistent divisions, making the chore slightly easier to complete. However, this method is a poor choice for handling very large quantities of files, since it involves resubmitting many commands and a good deal of guesswork.

Method #2: Use the find command.

Example 2

```
[user@localhost directory]$ find $directory -type f -name
'*' -exec mv
{} $directory2/. \;
```

Method #2 involves filtering the list of files through the find command, instructing it to properly handle each file based on a specified set of command-line parameters. Due to the built-in flexibility of the find command, this workaround is easy to use, successful and quite popular. It allows you to selectively work with subsets of files based on their name patterns, date stamps, permissions and even inode numbers. In addition, and perhaps most importantly, you can complete the entire task with a single command.

The main drawback to this method is the length of time required to complete the process. Unlike Method #1, where groups of files get processed as a unit, this procedure actually inspects the individual properties of each file before performing the designated operation. The overhead involved can be quite significant, and moving lots of files individually may take a long time.

Method #3: Create a function. *

Example 3a

```
function large_mv ()
{
    while read line1; do
        mv directory/$line1 ../directory2
    done
}
ls -l directory/ | large_mv
```

Although writing a shell function does involve a certain level of complexity, I find that this method allows for a greater degree of flexibility and control than either

Method #1 or #2. The short function given in Example 3a simply mimics the functionality of the find command given in Example 2: it deals with each file individually, processing them one by one. However, by writing a function you also gain the ability to perform an unlimited number of actions per file still using a single command:

Example 3b

```
function larger_mv ()
{
    while read line1; do
        md5sum directory/$line1 >> ~/md5sums
        ls -l directory/$line1 >> ~/backup_list
        mv directory/$line1 ../directory2
    done
}
ls -l directory/ | larger_mv
```

Example 3b demonstrates how you easily can get an md5sum and a backup listing of each file before moving it.

Unfortunately, since this method also requires that each file be dealt with individually, it will involve a delay similar to that of Method #2. From experience I have found that Method #2 is a little faster than the function given in Example 3a, so Method #3 should be used only in cases where the extra functionality is required.

Method #4: Recompile the Linux kernel. **

This last method requires a word of caution, as it is by far the most aggressive solution to the problem. It is presented here for the sake of thoroughness, since it is a valid method of getting around the problem. However, please be advised that due to the advanced nature of the solution, only experienced Linux users should attempt this hack. In addition, make sure to thoroughly test the final result in your environment before implementing it permanently.

One of the advantages of using an open-source kernel is that you are able to examine exactly what it is configured to do and modify its parameters to suit the individual needs of your system. Method #4 involves manually increasing the number of pages that are allocated within the kernel for command-line arguments. If you look at the include/linux/binfmts.h file, you will find the following near the top:

```
/*
 * MAX_ARG_PAGES defines the number of pages allocated
for arguments
 * and envelope for the new program. 32 should suffice,
this gives
 * a maximum env+arg of 128kB w/4KB pages!
 */
#define MAX_ARG_PAGES 32
```

In order to increase the amount of memory dedicated to the command-line arguments, you simply need to provide the MAX_ARG_PAGES value with a higher number. Once this edit is saved, simply recompile, install and reboot into the new kernel as you would do normally.

On my own test system I managed to solve all my problems by raising this value to 64. After extensive testing, I have not experienced a single problem since the switch. This is entirely expected since even with MAX_ARG_PAGES set to 64, the longest possible command line I could produce would only occupy 256KB of system memory--not very much by today's system hardware standards.

The advantages of Method #4 are clear. You are now able to simply run the command as you would normally, and it completes successfully. The disadvantages are equally clear. If you raise the amount of memory available to the command line beyond the amount of available system memory, you can create a D.O.S. attack on your own system and cause it to crash. On multiuser systems in particular, even a small increase can have a significant impact because every user is then allocated the additional memory. Therefore always test extensively in your own environment, as this is the safest way to determine if Method #4 is a viable option for you.

Conclusion

While writing this article, I came across many explanations for the "Argument list too long" error. Since the error message starts with "bash:", many people placed the blame on the bash shell. Similarly, seeing the application name included in the error caused a few people to blame the application itself. Instead, as I hope to have conclusively demonstrated in Method #4, the kernel itself is to "blame" for the limitation. In spite of the enthusiastic endorsement given by the original binfmts.h author, many of us have since found that 128KB of dedicated memory for the command line is simply not enough. Hopefully, by using one of the methods above, we can all forget about this one and get back to work.

Notes:

* All functions were written using the bash shell.

** The material presented in Method #4 was gathered from a discussion on the linux-kernel mailing list in March 2000. See the "Argument List too Long" thread in the linux-kernel archives for the full discussion.

Links

[1] <https://www.ssc.com/lj/subs/NewUSA.html>

Source URL: <http://www.linuxjournal.com/article/6060>