SESUG Paper 122-2019

UNIX X Command Tips and Tricks

David B. Horvath, MS, CCP

ABSTRACT

SAS® provides the ability to execute operating system level commands from within your SAS code – generically known as the "X Command". This session explores the various commands, the advantages and disadvantages of each, and their alternatives. The focus is on UNIX/Linux but much of the same applies to Windows as well. Under SAS EG, any issued commands execute on the SAS engine, not necessarily on the PC.

- X
- %sysexec
- Call system
- Systask command
- Filename pipe
- &SYSRC
- Waitfor

Alternatives will also be addressed – how to handle when NOXCMD is the default for your installation, saving results, and error checking.

INTRODUCTION

In this paper I will be covering some of the basics of the functionality within SAS that allows you to execute operating system commands from within your program. There are multiple ways you can do so – external to data steps, within data steps, and within macros. All of these, along with error checking, will be covered.

RELEVANT OPTIONS

Execution of any of the SAS System command execution commands depends on one option's setting:

```
XCMD              Enables the X command in SAS.
```

Which can only be set at startup:

```
         options xcmd;
                 ____
                 30
WARNING 30-12: SAS option XCMD is valid only at startup of the SAS System.
The SAS option is ignored.
```

Unfortunately, ff NOXCMD is set at startup time, you're out of luck. Sorry! You might want to have a conversation with your system administrators to determine why and if you can get it changed. It is interesting to note that SAS University Edition is NOXCMD – no way to execute UNIX commands within the VM.

X

The basic X Command executes independent of data steps and macros; it can be interspersed inside and outside of those constructs; the command executes when it is found, not within any control structure. The SAS Engine actually interprets some commands rather than passing them off to the command shell supplied with the operating system. As a result, it does not spawn off sub-processes which is important – in UNIX, information does not persist between sub-process invocations. A “change directory” in one sub-process does not persist (does not apply) when a separate sub-process issues a similar command. But with SAS, it actually does persist (which is fairly handy).

Note that the log handling is a bit annoying.

The best way to understand this is with some examples. Starting with the `pwd` and `cd` commands (print working directory and change directory) under UNIX:

```
NOTE: Current working directory is '/this/is/the/sas/install/directory'.
26         x "pwd"  ; /* works within SAS */
27         x "cd /my/directory/is/here"
27       !                  ; /* works within SAS */
NOTE: Current working directory is '/my/directory/is/here'.
28         x "pwd"  ; /* works within SAS */
```

The echo command and combining commands with the “;” behave a bit differently than normal under UNIX:

```
29         x "echo $HOME"
29       !               ; /* works but no output */
30         x "pwd; cd $HOME; pwd"
30       !                       ; /* no output, works? */
```

I can get multiple commands on one line to execute together and send output to a file using the “>” operator (known as redirection of STDOUT):

```
31         x 'echo start; echo mid; echo end>temp2.txt'
31       !                ; /* output to file, works */
```

I know this works because I can look at the output file (temp2.txt):

```
end
```

In this case, "start" and "mid" did not appear because of the way I wrote the statement – I only redirected the output of the end echo statement. I can redirect the output of multiple commands into one file or combine the files into one file as show below using the “( )” parenthesis operator:

```
37         x '(echo start; echo mid; echo end)>temp1.txt'
37       !   /* output to file, all 3 statements to file */
```

I confirm this behavior by looking at the output file (temp1.txt):

```
start
mid
end
```

Again, the command did exactly what I instructed it to. In this case I wanted the output of all three sent to the same file.

X AND %SYSRC

As you can see, I am getting very little information within the log – I only know the `echo` commands worked because I sent the output to a file. Some commands, like `cd` , do provide output in the log because SAS itself is performing the work rather than passing the command to the shell. But I can learn more about execution through the use of %SYSRC which will tell me the UNIX error code:

```
37         x '(echo start; echo mid; echo end)>temp1.txt'
37       !  ; /* output to file, all 3 statements to file */
SYMBOLGEN:  Macro variable SYSRC resolves to 0
38         %put &SYSRC;
0
```

In this case, execution was successful (we knew that before by looking at temp1.txt). We know it here because zero is success in UNIX. If the submitted command does not exist, we get the return value of 127:

```
39         x 'this_command_doesnot_exist'
39       !          ; /* non-zero RC */
SYMBOLGEN:  Macro variable SYSRC resolves to 127
40         %put &SYSRC;
127
```

I can also save error output using the 2> operator (redirecting STDERR) as shown below:

```
41         x 'this_command_doesnot_exist 2>test5.txt'
41       !                   ; /* non-zero RC */
SYMBOLGEN:  Macro variable SYSRC resolves to 127
42         %put &SYSRC;
127
```

Looking at test5.txt, we see the actual error message from the command shell:

```
/bin/bash: this_command_doesnot_exist: command not found
```

There are a wide variety of error statuses returned by the various commands within UNIX. The best way to determine the possible values is to look at the manual entry for that command. For instance, the `ls` command provides directory listings; `man ls` will tell me that if the file listed does not exist, the return code will be set to 2 (0 for success) as shown below:

```
43         x 'ls -al no_such_file'
43       !                        ; /* non-zero RC */
SYMBOLGEN:  Macro variable SYSRC resolves to 2
44         %put &SYSRC;
2
```

I know that In this case, "no_such_file" does not exist, so the %SYSRC value of 2 is no surprise. If I execute `man ls` at the command line, I will see the following towards the bottom of the document:

```
Exit status:
      0      if OK,
      1      if minor problems (e.g., cannot access subdirectory),
      2      if serious trouble (e.g., cannot access command-line argument).
```

SYSTASK

The systask command also executes independent of data steps and macros. It does provide additional option and capabilities compared to the basic X command. With the shell modifier, the SAS engine does not interpret the commands – they get passed directly to the command shell. You can also select the shell you wish to use (Korn, Bourne, bash, etc.) – or you can let it default to the one assigned to your user ID. Without the shell modifier, it behaves much like basic X – some commands will be interpreted within SAS and some will be sent to the command shell.

Another modifier is wait. When used, execution within SAS will not continue until the command itself has finished executing. This is important if subsequent code will be using the output of the current command (of affect the files involved). If the command is truly independent, you can leave wait off and let your program execute a bit faster.

With systask, %SYSRC, like X, is set with the UNIX error code:

```
54         systask command 'echo BOL $HOME EOL' wait;
NOTE: LOG/Output from task "task62"
> BOL $HOME EOL
NOTE: End of LOG/Output from task "task62"
54       !              /* $HOME not interpreted */
55         systask command "echo BOL $HOME EOL" wait shell;
NOTE: LOG/Output from task "task63"
> BOL /my/directory/is/here EOL
NOTE: End of LOG/Output from task "task63"
55       !             /* $HOME interpreted */
```

In line 54, the command shell variable $HOME was not interpreted because SAS interpreted the command and performed the action. In 55, with the shell modifier, let the command shell interpret the command which caused a directory name to be substituted in.

Other SYSTASK Options

Systask supports a number of modifiers and commands in addition to those shown above. The important modifiers are as follows:

- Wait: wait for this command to execute before starting next
- Cleanup: wait for this command and any nowait before starting next
- Shell: Can also specify the shell to use: shell="/usr/bin/ksh"
- Status: Can specify a status variable to check later (rather than &SYSRC)
- Taskname: Can specify task name for later use in Waitfor. If none is provided, “TASK” followed by a sequential number will be assigned.

`Systask list` will provide a status report on any systasks executed with the nowait modifier. In the following example, there is only one task that met that criteria (the 150th that I had run):

```
"task150" --------------
     Type: Task
     State: COMPLETE
     Status Macro Variable: Unspecified
73
74         systask list;
```

SYSTASK and Waitfor

The waitfor command is used in conjunction with systask/nowait to allow a series of commands to execute in parallel. This is used when you want to wait for a series of commands to complete before continuing. You have a choice – you can wait for any of the commands to complete (_ANY_ option) or all of them (_ALL_). Essentially you can implement an OR or AND condition. You can also set a timeout period where execution will resume after the specified time has expired independent of tasks actually completing. The general form of the command is

```
waitfor _ALL_ task1 task2 task3;
```

%SYSEXEC – MACROS

We can also execute system commands within a macro using %SYSEXEC. A simple example executes both the list directory and an invalid command:

```
72          %macro commands;
73           %sysexec %str(ls -al > test6.txt);
74           %put &SYSRC;
75           %sysexec %str(command_doesnot_exist 2> test7.txt);
76           %put &SYSRC;
77          %mend commands;
```

Of course we can use %SYSRC to determine if these commands executed successfully. While my examples merely output the return code in %SYSRC, you certainly can apply logic based on that result.

Executing this macro will result in the following log output:

```
79          %commands;
MLOGIC(COMMANDS):  Beginning execution.
MLOGIC(COMMANDS):  %SYSEXEC  ls al  test6.txt
MLOGIC(COMMANDS):  %PUT &SYSRC
SYMBOLGEN:  Macro variable SYSRC resolves to 0
0
MLOGIC(COMMANDS):  %SYSEXEC  command_doesnot_exist 2 test7.txt
MLOGIC(COMMANDS):  %PUT &SYSRC
SYMBOLGEN:  Macro variable SYSRC resolves to 127
127
MLOGIC(COMMANDS):  Ending execution.
```

Note that the list directory command was successful (%SYSRC of 0) and the invalid command was not (returned 127). You may have wondered about the differences between the macro and the log output. This was not a copy/paste issue – it is yet another log oddity.

CALL SYSTEM– DATA STEP

While you can execute X or SYSTASK within a data step, those will execute only once – are not included within the implied read loop. In order to execute system commands within the loop, you would use CALL SYSTEM. These commands are passed to the system call shell – not interpreted by the SAS engine as is the case with X and SYSTASK (without shell modifier). %SYSRC is set for each call. In the following example both systask and call system are demonstrated:

```
85          data dir;
86           filename commands PIPE "ls | head -2";
```

```
87          infile commands truncover;
88          input result $char60.;
89
90          string="echo " || result || " >> test4.txt";
91          call system(string); /* no output - but it executes multiple
times  */
92          system_rc=symget("SYSRC");
93          call system("this_command_doesnot_exist");
94          system_rc2=symget("SYSRC");
95
96          systask command "pwd" wait shell;  /* runs once */
NOTE: LOG/Output from task "task59"
> /my/directory/is/here
NOTE: End of LOG/Output from task "task59"
97          output;
98         run;
```

Note that the pwd command is only executed once as shown in the log as task59. I will talk about the filename statement on line 86 shortly. The log results:

```
NOTE: The infile COMMANDS is:
      Pipe command="ls | head -2"

NOTE: 2 records were read from the infile COMMANDS.
      The minimum record length was 22.
      The maximum record length was 24.
NOTE: The data set WORK.DIR has 2 observations and 4 variables.
NOTE: Compressing data set WORK.DIR increased size by 100.00 percent.
      Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.00 seconds
```

We can look at the output two ways – proc print of the output dataset which shows that the command to be executed, the first call succeeding, and the second call failing because the command does not exist:

Obs	result	string	system_rc	system_rc2
1	FILE1	echo FILE1 >> test4.txt	0	127
2	FILE2	echo FILE2 >> test4.txt	0	127

We can also verify this by looking at the text file created within the call itself (test4.tx)t:

```
FILE1
FILE2
```

In simple terms, two records were read from infile, two records were written to work.dir, and two records were written to test4.txt via the echo command.

FILENAME PIPE

Filename Pipe is another means of executing system commands within your SAS program. In general, it behaves like any other filename statement – it accepts data written from SAS (FILE/PUT) or provides data for SAS to read (INFILE/INPUT). What it does differently is that the data comes from/is sent to a UNIX command rather than working with a simple file. I've found three main uses for this capability – when I am looking for output of a system command (like getting a directory listing in the example above), when I want something to happen in parallel for efficiency reasons (like reading from a compressed file), and when I want to perform an action that SAS is unable.

An example that covers both situations is when I want my output to be stored in a compressed form. I can do this with a regular FILENAME statement and then issue a gzip command from the command line – taking more time.

I can also use FILENAME ZIP. If I need a file in GZIP format, I can use the GZIP modifier – but only if I have version 9.4M5 or newer. If I have an earlier version, I have to use a FILENAME PIPE:

```
filename gzipit pipe 'gzip -c > /my/output/directory/file.txt.gz'; * run the
UNIX gzip;
```

FILENAME PIPE AND &SYSRC

Unfortunately, &SYSRC is not set by Filename Pipe. In the following example, we attempt to execute three commands that don't actually exist:

```
113         data baddir;
114          filename commands PIPE "lx ; lx ; lx";
115          system_rc=symget("SYSRC");
116
117          infile commands truncover;
118          input result $char60.;
119          output;
120         run;

NOTE: The infile COMMANDS is:
      Pipe command="lx ; lx ; lx"

NOTE: 3 records were read from the infile COMMANDS.
      The minimum record length was 32.
      The maximum record length was 32.
```

Three invalid commands are executed and their output stored in the BADDIR dataset. We know the commands are invalid because the output message tells us so. As shown earlier in this paper, %SYSRC should have been set to 127 as a result. But as we can see below, it was not:

Obs	system_rc	result
1	0	/bin/bash: lx: command not found
2	0	/bin/bash: lx: command not found
3	0	/bin/bash: lx: command not found

SHELL SCRIPTS

When the NOXCMD option is set, all of the commands that execute system commands are disabled: X, SYSTASK COMMAND, CALL SYSTEM, %SYSEXEC, and FILENAME PIPE. In that case, if you need to

execute commands within the UNIX command environment you have two choices (three if you count convincing your administrator to allow XCMD): manually type each command every time or a shell script.

A shell script is a set of commands in a file with the shell name as the first line (#!/bin/ksh in the example below) that performs the required steps. In the following example, the directory is changed, the return code checked to be sure, a directory listing is created in a file (to be read by your_sas_part_1.sas), the current directory is printed, your second program is run, and then the output is compressed with gzip.

```
#!/bin/ksh
cd /desired/directory
# You can check return codes here with the $?
if [[ $? -gt 0 ]]; then
        echo "cd failed"
        exit 3
fi
# if we get here 'cd' succeeded
ls -al | head -2 > temp.txt
sas your_sas_part_1.sas
pwd
sas your_sas_part_2.sas
gzip /my/output/directory/file.txt
```

It is a good practice to check the return code after each command – no reason to run the next command if the prior failed.

CONCLUSION

SAS provides many features allowing you to execute system commands from within your program. But it really is about choices:

- Sometimes it is better to execute UNIX commands in your program
- Sometimes not

It really is up to you to make the decision.

REFERENCES

x, sysexc, system:

- http://support.sas.com/documentation/cdl/en/hostunx/61879/HTML/default/viewer.htm#xcomm.htm
- http://support.sas.com/documentation/cdl/en/hostunx/63053/HTML/default/viewer.htm#p0w085btd5r0a4n1km4bcdpgqibt.htm

%sysexec:

- http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#n08ecabbpebv2xn13ieu8uylrohm.htm

filename pipe:

- http://support.sas.com/documentation/cdl/en/hostunx/63053/HTML/default/viewer.htm#n1ceb0xedanuj3n19l3g73awk1wf.htm
- https://documentation.sas.com/?docsetId=hostunx&docsetTarget=n1ceb0xedanuj3n19l3g73awk1wf.htm&docsetVersion=9.4&locale=en

bash scripts:

- https://www.taniarascia.com/how-to-create-and-use-bash-scripts/

systask:

- http://support.sas.com/documentation/cdl/en/hostunx/63053/HTML/default/viewer.htm#p0lzxl2mwndagun1dtxbst9s4jea.htm

xsync:

- https://documentation.sas.com/?docsetId=hostunx&docsetTarget=p0is4umpbeej5pn1xwc8mwgt9qz5.htm&docsetVersion=9.4&locale=en

xcmd:

- http://support.sas.com/documentation/cdl/en/hostwin/69955/HTML/default/viewer.htm#p0xtd57b40ehdfn1jyk8yxemfrtv.htm

x command (x windows) options:

- https://documentation.sas.com/?docsetId=hostunx&docsetTarget=n1nx651zrt6pdcn171xjr3xwfxhq.htm&docsetVersion=9.4&locale=en

waitfor:

- https://documentation.sas.com/?docsetId=hostunx&docsetTarget=p0w8zwo1dyssdfn1mjm11dt2v7e2.htm&docsetVersion=9.4&locale=en

ACKNOWLEDGMENTS

I want to thank the organizers of this great conference, my employer for their willingness to allow me to expand my horizons through events like this, and, last but certainly not least, my spouse Mary who doesn't complain when I spend so much time at the keyboard working on documents like this.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David B. Horvath, CCP
+1-610-859-8826
dhorvath@cobs.com
http://www.cobs.com