
Korn Shell Programming Techniques

Tried and True Programming Techniques to Get You Started Quickly
by Ken Gottry

Table of Contents

Background	2
Writing a Script – Some Basics.....	3
Writing a Simple Script	4
Run the UNIX Command Interactively	4
Create a Shell Script	4
Make the Shell Script Executable	4
Test the Shell Script	5
Launching the Script.....	6
Extending the Simple Script.....	8
Redirecting stdout.....	8
Executing sub-commands within the script	8
Generating Unique File Names.....	9
Structured Programming Techniques	9
Writing a “for-loop” Script.....	11
Writing a “while-loop” Script	12
Quick Reference Card.....	14

Background

The *command shell* is the layer that interacts with the user and communicates with the operating system. When using MS-DOS most people use the `command.com` shell; however a different shell can be specified via the COMSPEC environment variable.

Similarly, each UNIX® user must select a command shell to use to communicate to UNIX. When a UNIX account is established the sysadmin selects the user's default shell. Normal options are Bourne Shell (`/bin/sh`), C-Shell (`/bin/csh`), Korn Shell (`/bin/ksh`) and Bourne-Again Shell (`/bin/bash`). While many developers use C-Shell because of its C-like syntax, This is a subjective selection and this article uses the Korn shell exclusively. All of the scripts discussed in this document are written using Korn shell. The syntax will not necessarily work under any other shell.

When you execute a shell script from the command line, your default shell is used. If your default shell is Korn, then your scripts execute without syntax errors. But what if you want others to execute your script. You can't rely on the user's default shell to ensure your scripts are always run using the Korn shell. So, you use a UNIX feature whereby the first line of a shell script indicates under what shell the script is to be executed. The syntax in Figure 1 forces a script to be run using the Korn shell regardless of what shell the current user is executing.

```
#!/bin/ksh
# your script goes here. All lines starting with # are treated as comments
```

Figure 1 - Force a script to be executed by the Korn shell

Some documentation uses a different command prompt symbol to indicate the current shell as shown in Figure 2. Since this author's favorite shell is the Korn shell, all of the examples in this article use the `$`-prompt. Since you cannot ensure that your scripts will always be executed using the Korn shell, put **`#!/bin/ksh`** as the first line in each scripts. (The `$`-prompt in this article just indicates that a command is being entered at the command line.)

Prompt	Shell
\$	Bourne or Korn
%	C-shell
#	Root login

Figure 2 - UNIX prompt symbols

Writing a Script – Some Basics

A UNIX script file is similar to a DOS BAT file. All of the programming do's and don'ts from the DOS world still apply in UNIX.

Writing any script involves these steps:

1. Run the UNIX command interactively at a shell prompt
2. Create the shell script containing the UNIX command
3. Make the shell script executable
4. Test the script
5. Launch the script
 - a. Interactively
 - b. Once, at a future date and time
 - c. Repeatedly on a fixed schedule
 - d. Using an HTML form

Writing a Simple Script

Assume that you want to write a script to capture *vmstat* information. You want to run *vmstat* on 2-second intervals for one minute. Use the five steps described above to achieve your goal.

Run the UNIX Command Interactively

First, look up the documentation for *vmstat* using *man vmstat*. Next, run the command interactively to be sure you understand the syntax and the expected output. Figure 3 shows the syntax to run *vmstat* 30 times at 2-second intervals.

```
$ vmstat 2 30
```

Figure 3 - Run the vmstat command interactively 30 times at 2 second intervals

Create a Shell Script

Next, create a script file containing the command. You should establish standards describing script location and script names. Store all things of a specific category, for instance a company, in a subdirectory under `/usr/local`. For this example, assume the company is Acme Products so the directory is `/usr/local/acme`. Within this directory create one subdirectory called **scripts** and another called **logs**. Other subdirectories may be necessary for other purposes.

Next, use a text editor such as *vi* to create a script file called **capture_vmstat.sh**. File extensions are meaningless in UNIX unlike DOS where EXE, COM and BAT indicate executable files. You could use `“.sh”` as an extension to denote shell script files, but that doesn't make the script executable. This naming convention for files makes it easier to quickly identify files. Also, you can use the *find* command to locate all files of a particular type if the file names adhere to a standard.

The script file has two lines in it. The first line leaves nothing to chance, stating that the Korn shell should execute the commands inside this script. The second line is the UNIX command itself. Figure 4 is a complete listing of *capture_vmstat.sh* script.

```
#!/bin/ksh
vmstat 2 30
```

Figure 4 - capture_vmstat.sh script to run vmstat 30 times at 2-second intervals

Make the Shell Script Executable

Unlike DOS that uses the file extension to determine if a file is executable or not, UNIX relies on file permissions. The *chmod* command is used to mark the file executable.

The simplest way to turn on the execute-bit is using `chmod +x capture_vmstat.sh`. In a production environment, on an exposed server you must also consider owner, group and world permissions to control complete access to the script. (The topic of file permissions is beyond the scope of this document.) See *man chmod* for more information.

Test the Shell Script

Now the script is ready to test. Unlike DOS, UNIX does not automatically look in the current directory for a file to execute. UNIX provides the PATH environment variable. UNIX will only search for executables in directories identified in the PATH variable. Since most people don't include the current directory in the PATH (a dot indicates the current directory), just typing the command in Figure 5 will not work because `/usr/local/acme/scripts` is not in the PATH.

```
$ cd /usr/local/acme/scripts
$ capture_vmstat.sh
```

Figure 5 - This will NOT execute the script unless “dot” is in the PATH

You must explicitly specify the full file name of the script, including path. Do not rely on the PATH variable because it could get changed in the future and one of two things could go wrong. First, the directory where your scripts reside could be inadvertently removed from the PATH and UNIX would no longer be able to locate your scripts. Worse yet, UNIX might find and execute a script by the same name in a different directory, one listed in the new PATH. Therefore, to be safe, you should always execute your scripts by specifying the full file name as shown in Figure 6.

```
$ /usr/local/acme/scripts/capture_vmstat.sh
```

Figure 6 - Specifying the full file name to ensure UNIX finds the correct script

Maybe you don't like typing so a shortcut relies on the fact that “.” (dot) refers to the current directory. First, change to the script directory and then execute the script by prepending “./” (dot-slash) to the script name as shown in Figure 7. This doesn't save much typing if you are only executing one script; however, if you are going to execute several scripts from your script directory, then you only have to type the directory name once.

```
$ cd /usr/local/acme/scripts
$ ./capture_vmstat.sh
```

Figure 7 - Executing the script using the dot-slash notation

Regardless of how you invoke the `capture_vmstat.sh` script, the output should be identical to what you get when you run `vmstat` interactively.

Launching the Script

Now you have the script and you know it works. There are four ways to run the script.

1. *Interactively*

Document the script and let others (perhaps the Help Desk staff) run the script file. The folks who run the script don't need to know UNIX commands or syntax in much the same way that DOS users don't need to understand DOS commands or syntax in order to use a BAT file created for them.

2. *Using the at Command*

Use the *at* command to execute a script once at some time in the future. Check *man at* for details. Some UNIX systems cancel running "at-jobs" when a user logs out. Check system documentation carefully.

3. *Using the cron Utility*

Use the *crontab* file to execute a script repeatedly on a fixed schedule. Check *man crontab* for details. Figure 8 shows a simple *crontab* entry to run your script once an hour from 8AM-5PM at 10 minutes past the hour every Monday, Wednesday and Friday:

```
10 8-17 * * 1,3,5 /usr/local/acme/scripts/capture_vmstat.sh
```

Figure 8 - crontab entry to run the capture_vmstat.sh script

Before moving on to the fourth method for launching your script you need to understand two problems with running scripts via *crontab*. First, since you are not logged in when the script is executed, you can't rely on Korn shell being the default shell. Therefore, you must be sure to use "#!/bin/ksh" as the first line of your script as explained in Figure 1. Second, the current version of your script sends its output to the terminal. When *cron* launches the script there is no terminal so *cron* must redirect the stdout somewhere. The normal place is to the email inbasket of the user whose crontab launched the script. While this may be acceptable, other (better) solutions are described below when you expand your basic script.

4. *Using an HTML Form*

Launch your script using an HTML form and POST'ing your script via CGI (common gateway interface). The output of the command will be sent back to the browser so the <PRE> and </PRE> HTML tags should be used to preserve formatting.

There is a bit more to this HTML form method than described here, and there are numerous security risks with using FORM's and CGI. However, this method has proven very successful for use by in-house Help Desk staff or other level-one support personnel.

Extending the Simple Script

The previous script was the shell-script version of “hello, world”, the standard first program written when learning a new programming language. Now you can add a few more basic features to it.

Redirecting stdout

First, the script sends its output to stdout, which is normally the terminal. You can extend the script to redirect the output to a log file as shown in Figure 9.

```
#!/bin/ksh
vmstat 2 30 > /usr/local/acme/logs/vmstat.log
```

Figure 9 - Redirecting stdout to a file

But this introduces a couple of new problems. First, every time you run the script it overwrites the contents of the last log file. To correct that, append the new output to the end of the existing log file. But then you need to know when each output in the log was created, since the date-time stamp on the file only indicates when the last one was written.

Executing sub-commands within the script

Write the current date and time to the file preceding each execution of the script. Use “>>” to append the output to the end of the file rather than overwriting the existing file. In Figure 10, a uniquely identifiable character is put in column one to make it easy to scan the file using *find* and *find-next*. You can also write the current date and time to the log file. In Figure 10 `$(date)` instructs the Korn shell to execute the *date* command and place the output into the *echo* command line. Whenever you want to execute a UNIX command and use the output, type a `$` and enclose the command within parentheses.

```
#!/bin/ksh
echo "#--- $(date) >> /usr/local/acme/logs/vmstat.log
vmstat 2 30 >> /usr/local/acme/logs/vmstat.log
```

Figure 10 - Appending stdout to a log file

In Figure 11, the Korn shell is instructed to run the *netstat* command, *grep* for ESTABLISH, and use *wc* to count the number of lines by enclosing these commands in `$(xxx)`. Further the Korn shell is instructed to store the output of these commands in environment variable `CTR_ESTAB`. Then in the *echo* command, the Korn shell is instructed to use the value stored in that environment variable. To use a value that is stored in an environment variable, put a `$` in front of the variable name, e.g. `$CTR_ESTAB`. To improve readability and to avoid ambiguities, use the Korn shell option of enclosing the variable name inside curly braces, e.g. `${CTR_ESTAB}`.

```
# store current date as YYYYMMDD in variable DATE for later use
export DATE=$(date +%Y%m%d)

# count number of established socket connections and write to log
export CTR_ESTAB=$(netstat -na -P tcp | grep ESTABLISH | wc -l)
export CTR_CLOSE_WAIT=$(netstat -na -P tcp | grep CLOSE_WAIT | wc -l)
echo "${DATE} ${CTR_ESTAB} ${CTR_CLOSE_WAIT} >> ${LOG_FILE}
```

Figure 11 - Using \$(xxx) to execute a command within a Korn shell script

Generating Unique File Names

What happens if multiple users run the script concurrently? The output from each script would be interleaved in the output file since each instance of the script would be writing to the same output file. You can create a unique output file name by placing the PID number (represented by \$\$) in the file name, as shown in Figure 12.

```
#!/bin/ksh
echo "#--- $(date)" >> /usr/local/acme/logs/vmstat.$$log
vmstat 2 30 >> /usr/local/acme/logs/vmstat.$$log
```

Figure 12 - Using \$\$ to generate unique file names using current PID

When the next user runs the script, a different PID will be assigned to the script's execution thus causing a separate log file to be created each time instead of appending to the existing log file. Maybe that's not a bad thing, but it's not what you want to achieve.

Another possibility, instead of using an environment variable whose value is changed each time the script is executed, is to use an environment variable that is set once, outside the script, prior to the execution of the script. UNIX automatically sets the LOGNAME environment variable whenever a user logs in. In Figure 13, this value is imbedded into the log file name so that each user can have a log file:

```
#!/bin/ksh
echo "#--- $(date)" >> /usr/local/acme/logs/vmstat.${LOGNAME}.log
vmstat 2 30 >> /usr/local/acme/logs/vmstat.${LOGNAME}.log
```

Figure 13 - Generating a file name using an environment variable whose value is externally set

Structured Programming Techniques

Two final touch-ups and you're done with your basic Korn shell script. First, what if you want to change the frequency or duration of the *vmstat* command? You can accept those values via command line arguments rather than hard-code the interval and duration in the *vmstat* command. These arguments can be stored in environment variables from where the *vmstat* command can access them. Of course, your script must provide default values in case the user doesn't provide values via the command line.

Second, what if you change your mind about the log file naming convention? This is not something you want the user to have to provide each time via a command line argument. However, if you have hard-coded the log file name in multiple lines within the script, then when you decide to use a different naming convention you will have to search every line of the script to see where the name was specified.

Instead, store the log file name in an environment variable and modify each command to append output to the file name contained in the variable. Then, when you change the log file naming convention, all you need to do is modify the one line where the environment variable is set.

```
#!/bin/ksh
# -----
# capture_vmstat.sh <INTERVAL> <COUNT>
# <INTERVAL> vmstat interval
# <COUNT> vmstat count
# run vmstat and capture output to a log file
#-----

# indicate defaults for how often and for how long to run vmstat
export INTERVAL=2 # every 2 seconds
export COUNT=30 # do it 30 times

# obtain command line arguments, if present
if [ "${1}" != "" ]
then
    INTERVAL=${1}
    # if there is one command line argument, maybe there's two
    if [ "${2}" != "" ]
    then
        COUNT=${2}
    fi
fi

# directories where scripts and logs are stored
export PROGDIR=/usr/local/acme/scripts
export LOGDIR=/usr/local/acme/logs

# define logfile name and location
export LOG_FILE=${LOGDIR}/capture_vmstat.${LOGNAME}.log

# write current date/time to log file
echo "#--- $(date) >> ${LOG_FILE}"
vmstat ${INTERVAL} ${COUNT} >> ${LOG_FILE}

# say goodnight, Gracie
exit 0
```

Figure 14 - A more robust version of the capture_vmstat.sh script

Writing a “for-loop” Script

Sometimes you want to execute a single command against a list of objects. For example, you may want to use the *rsh* command to remotely execute the same command against multiple servers (see *man rsh* for details ... and for security risks when using *r-commands*).

One technique is to store the list of objects in an environment variable, perhaps called LIST. Then you can use the “for” loop to execute the rsh command repeatedly, each loop having the next value in the LIST. Figure 15 shows a sample of a for-loop script.

```
#!/bin/ksh

export LIST="bvapp1 bvapp2 bvapp3"

export LOG=/usr/local/acme/logs/throw_away.log

for SERVER in ${LIST}
do
    # each loop has a different value for ${SERVER}
    echo "#----- values from ${SERVER}" >> ${LOG}
    rsh ${SERVER} "ps -f -u bv -o pid,pmem,pcpu,rss,vsz" >> ${LOG}
done

# say goodnight, Gracie
exit 0
```

Figure 15 - A simple for-loop script

Writing a “while-loop” Script

Sometimes you may want to execute a single command, wait a while, and then execute the command again. Sometimes you want this loop to continue indefinitely while other times you want the loop to execute a finite number of times and then terminate.

Say you want to monitor processes running under user “bv”. You want to monitor “bv” every 10 seconds for 2 hours. First, using the code in Figure 16 you test the command interactively (see *man ps* for details):

```
ps -f -u bv -o pid,pcpu,pmem,rss,vsz,comm
```

Figure 16 - Interactive ps command using the -o argument

Now you need to write a script file that executes this in a loop. The loop should pause for 10 seconds between executions of the ps command. The loop should execute 720 times [every 10 seconds means 6 times per minute or 360 times per hour (60 mins/hr * 6/min) for two hours]. Figure 17 shows a simple while-loop script.

```
#!/bin/ksh

export INTERVAL=10
export COUNT=720

export LOG=/usr/local/acme/logs/while_loop_test.log

export CTR=0
while [ true ]
do
    if [ ${CTR} -ge ${COUNT} ]
    then
        exit
    fi
    echo "#----- $(date +%Y%m%d-%H%M%S)" >> ${LOG}
    ps -f -u bv -o pid,pcpu,pmem,rss,vsz,comm >> ${LOG}
    CTR=$(expr ${CTR} + 1)
    sleep ${INTERVAL}
done
```

Figure 17 - Simple while-loop script

Figure 18 shows a snippet from the output log file.

```
#----- 19991203-123237
  PID %CPU %MEM  RSS  VSZ COMMAND
12007  0.2  0.8 13640 24280 cmsdb
11938  0.0  0.7 11536 20496 sched_poll_d
<snip>
#----- 19991203-123240
  PID %CPU %MEM  RSS  VSZ COMMAND
12007  0.2  0.8 13640 24280 cmsdb
11938  0.0  0.7 11536 20496 sched_poll_d
<snip>
#----- 19991203-123243
  PID %CPU %MEM  RSS  VSZ COMMAND
12007  0.3  0.8 13640 24280 cmsdb
11938  0.0  0.7 11536 20496 sched_poll_d
<snip>
#----- 19991203-123246
<and-so-on>
```

Figure 18- Output from the while-loop script

Quick Reference Card

The programming tips and techniques below are a quick reference to the programming style and methodology presented in this article where you will find more detail about each item listed below.

1. Always start scripts with a line that says

```
#!/bin/ksh
```

2. Always use uppercase when defining variables. Use underscores to separate words.

```
BIN_DIR=/opt/bv1to1/bin
```

3. Always export environment variables so that any sub-processes will have automatic access to the values:

```
export SUPPORT_IDS="userA@domain.com,userB@domain.com"
```

4. To execute a UNIX command and use the output elsewhere in a Korn shell script, type a \$, enclose the command within parentheses, and store the output in an environment variable.

```
export CTR_ESTAB=$(netstat -na | grep ESTABLISH | wc -l)
```

5. To use a value that is stored in an environment variable, put a \$ in front of the variable name. To improve readability and to avoid ambiguities, enclose the variable name inside curly braces.

```
echo "The number of ESTABLISHED connections is ${CTR_ESTAB}"
```

6. To ensure having a unique file name, use \$\$ to include the PID number in the file name. Insert the PID number into the file name just prior to the file extension:

```
export LOG_FILE=/tmp/capture_vmstat.$$ .log
```

7. Use **chmod +x filename** to make a script file executable.

```
chmod +x capture_vmstat.sh
```

8. Precede a script name with dot-slash when executing interactively so UNIX knows that the script is in the current directory.

```
./capture_vmstat.sh
```

9. Redirect stdout (>) to a log file or append stdout (>>) to a log file.

```
./capture_vmstat.sh >> ${LOG_FILE}
```

10. Redirect stderr, either to the same destination as stdout or to a unique file.

```
./capture_vmstat.sh >> ${LOG_FILE} 2>&1  
- or -  
./capture_vmstat.sh >> ${LOG_FILE} 2>>${ERR_LOG}
```

11. Use the for-loop to process a list of things.

```
export LIST=$(ls *sh)  
for FILE in ${LIST}  
do  
    echo "Processing ${FILE}"  
    cat ${FILE} | mailx -s "Here is ${FILE}" userA@domain.com  
done
```

12. Use the while-loop to process the same command repeatedly.

```
export INTERVAL=20  
export COUNT=180  
  
export CTR=0  
while [ true ]  
do  
if [ ${CTR} -ge ${COUNT} ]  
then  
    exit  
fi  
# --- do some command here ---  
sleep ${INTERVAL}  
done
```

References

Unix Shell Programming (Hayden Books Unix System Library)

by Stephen G. Kochan, Patrick H. Wood

Paperback - 490 pages

2nd Revised edition (January 1990)

Hayden Books; ISBN: 067248448X

Author Biography

Ken Gottry is a Sr. Infrastructure Architect with NerveWire, Inc. He has 30 years experience with systems ranging from mainframes, to minicomputers, to servers, to desktops. For the past 10 years his focus has been on designing, implementing and tuning distributed, multi-tier, and web-based systems. As a performance engineer consulting to numerous G2K firms, he has assessed and tuned web servers, app servers and JVM's running on Solaris.

Ken's articles have appeared on Sun's dot-com builder website (<http://dcb.sun.com>) and iPlanet's developer website (<http://developer.iplanet.com>). Also, Ken has recently published an article about Solaris performance tuning in SysAdmin magazine (<http://www.sysadminmag.com>).

Ken Gottry
506 Babcock Road
Tully, NY 13159
mobile: 315.383.5249
fax: 315.696.0065
ken@gottry.com
kgottry@nervewire.com