Useful Bash scripts

This past week I spent some time digging through old shelves and boxes and found a few of my early UNIX textbooks. Flipping through them I found it interesting to see that while our technology continues to race forward, a lot of the basics have stayed the same. Much of the security practices suggested twenty years ago are still valid today. Some of the backup scripts provided in these books (one of which talks about this new thing coming out called Netscape Navigator) are still helpful. We've talked about the command line here before a time or two and this week I'm feeling nostalgic. So I'd like to dig into the scripting toy box and share a few items that I have found useful over the years.

This first script is fairly simple. It checks to see how much room is currently available on the /home partition. If less than 10 GB of space remains free, then a warning is displayed. In place of printing a warning, we can substitute sending an e-mail or other notification.

```
#!/bin/bash
space=$(df | grep /home | awk '{print $4}')
if [ $space -lt 10485760 ] ; then
  echo "Running out of room"
fi
```

It works by running the df command, which displays disk usage information. The grep command then looks for the information related to our /home partition. And the awk command extracts the data in the forth column of df's output -- the forth column telling us how much space is still available. Do you have a room-mate who won't stop filling your hard drive with media files? Get cron to run this script and have it shutdown his or her favourite P2P client by running "killall" in place for the "echo" command.

Our next script was put together to get around a problem I kept creating for myself. Every so often I find myself downloading a program or performing an update only to discover that it's taking longer than expected and I'm waiting for it to finish so I can run another command. Just as an example, imagine I'm downloading nmap and, once the install is complete, I want to run nmap to scan my machine. Rather than wait around, I can use this script:

```
#!/bin/bash
if [ $# -lt 2 ] ; then
  echo "Usage: $0 PID command"
  exit 1
fi
while [ -d /proc/$1 ] ; do

  sleep 1
done
shift
$@
```

The script takes two or more arguments. The first is the process ID number of the program we're waiting for. Everything specified on the command line after that is what we want to run. Since each process ID has an entry in the /proc directory, we can see if a program is still running by checking for the existence of that directory. If we called the above

script "waitfor" and our slow process-in-progress is number "24658" we could use it like this:

```
waitfor 24658 nmap localhost
```

Once program number 24658 has completed its task, the waitfor script will run nmap to scan the local machine. The "shift" command at the bottom of the script says we're done with the first argument (the PID) and it can be discarded. The following line with the "$@" means we want to run the remaining arguments (nmap localhost) as if they were typed on the command line.

Have you ever been working from a terminal and wondered if a word you were using was spelled correctly? This next script can check that for you using the "spell" command available on most distributions. What this does is take a given word and run it through the system's spell-checker. If the word is spelled correctly it gives us confirmation. On the other hand, if the spell-checker can't find a match, it will try to display a list of possible correct spellings.

```
#!/bin/bash
# Check spelling of a word passed on the command line
if [ $# -lt 1 ] ; then
  echo "Usage: $0 word"
  exit 1
fi

# Check to see if the word is spelled correctly
if [ -z $(echo $1 | spell) ] ; then
  echo "$1 is spelled correctly."
  exit 0
fi

# Search for recommendation
echo "Could not find match. Try one of these."
grep ^$(echo $1 | cut -c -4) /usr/share/dict/words | head -n 20
```

I've broken the script into three chunks. The first block checks to make sure we gave the script a word to check. The second part hands the word over to the spell-checker and lets us know if the spell-checker gave us the "OK". If the script makes it to the third part, it assumes the word was not spelled correctly. It then takes the first four letters of the word we gave it (using the cut command) and uses grep to find matches to those first four letters in the system's dictionary. The "head" command at the end of the line limits the number of recommendations we get back to 20 or less, to avoid flooding our terminal screen.

This next script was born from two accidents I witnessed early in my IT days where computers caught fire. Both cases were due to hardware failure, in one case a CPU fan stopped working and the server kept going. Since it's often convenient to leave a machine running unattended, I came up with a small script that could be run from the terminal or from a cron job. Machines these days have more safety guards, but I think it's still worth looking at the script. This script checks the current temperature of the CPU. If the temperature reaches 50 degrees Celsius, a warning is displayed. Should the temperature reach 80 degrees, the machine is shut down.

```
#!/bin/bash
warning=50
killpoint=80
mysensor=temp1

current=$(sensors | grep $mysensor | awk '{print $2}' | cut -c 2-3)
if [ $current -gt $warning ] ; then
  echo "Warning, temperature at mid-range."
fi

if [ $current -gt $killpoint ] ; then
  echo "Too hot, shutting down."
  shutdown -P now
fi
```

The first line runs the sensors command which will display temperature information from the machine's internal sensors. The grep command will then, in this case, grab the data associated with sensor "temp1". Different systems may have different sensor names. The awk and cut commands then grab the temperature data, weeding out extra symbols, such as the leading "+" sign and trailing "C". The temperature is saved in a variable called "current", which we then compare to our warning and cutoff points.

Each of these scripts as-is are small and perform simple tasks. But in them are the building blocks for more powerful tools. As an example, the waitfor script could try to terminate the process it is waiting on after ten minutes. Or the script which checks for available disk space could be modified to hunt down large files that have not been accessed for a long time and list them. The temperature checking script could try to kill the process using the most CPU in an effort to cool things down. There are an amazing number of possibilities when using shell scripts.

If you are interested in learning about the command line and shell scripting, I recommend picking up a copy of UNIX: The Textbook. It does a great job of slowly introducing new concepts and gradually building upon existing material. I found it to be a helpful introduction to the UNIX family of operating systems. Brave souls may also want to read Advanced Bash-Scripting Guide, a text that jumps into the more interesting end of the scripting pool.