SESUG Paper 127-2019

Leveraging "UNIX Tools" (GNU) for Data Analysis

David B. Horvath, MS, CCP

ABSTRACT

Life would be so much easier if everything was in a database or readily processed within SAS®. But that is not the case. All too often we get data files (or have to send them) in various formats. This session discusses some of the tools available to help you figure out what the file looks like so you can pull it apart using those tools or your SAS. While the GNU version of these tools will be the focus, the skills learned apply to many different platforms (Microsoft's Bash under Windows 10, Cygwin under Microsoft Windows, MAC OSX, the Linux core of Android, commercial Linux — like Red Hat Enterprise, and commercial UNIX — like IBM's AIX or Sun/Oracle's Solaris).

Of particular interest are 'head', 'tail', 'wc', 'awk', 'dd conv', and shells.

A few of the differences between UNIX/Linux and Windows will also be discussed in case you ever have to deal with those environments in our heterogeneous environments. This knowledge also comes in handy if you need to migrate code from an existing UNIX/Linux-based application.

INTRODUCTION

SAS provides the ability to process many kinds of data but there are times that it does not meet your data analysis needs – at least until you understand the data to be able to write the code to process it. Even when using EG, there are difficulties dealing with certain files. For instance, EG is only available on your PC – what if the data is on a server somewhere? How long will it take to download to view a large file (like 4 Gb or larger)? How about very wide files – like record lengths of almost 88,000 characters? Or files that contain control characters or binary values? Those without record separators (aka "fixed length records") or those with non-standard record separators (like LF in Windows or CR/LF in UNIX)?

How would you be able to view and understand that data?

This really comes down to using the right tool in the right place.

SOME TERMINOLOGY

It would be best to start with some terminology. Some of it is confusing because they sound similar but have different meanings.

GNU – GNU is Not UNIX. This is a command line interface (CLI) environment that mimics that under UNIX. The GNU project was started in 1983 by Richard M. Stallman at MIT.

FOSS – Free/Open Source Software. "FOSS programs are those that have licenses that allow users to freely run the program for any purpose, modify the program as they want, and also to freely distribute copies of either the original version or their own modified version. " – http://www.webopedia.com/TERM/F/FOSS.html

Open System. In the generic sense, this is any system that follows a public set of standards. But in this case: A system that meets the POSIX/UNIX standards

UNIX is an operating system and a registered trademark of The Open Group. It is often used as generic term as in "UNIX-like". It is considered to be an Open System but not FOSS – a mistake my students often make. That is because it follows the standards but you have to pay for it – and aren't allowed to modify/share it. The different versions are referred to as "flavors" (i.e., Solaris, AIX, Tru64, etc.).

Linux is an operating system that consists of a kernel plus the GNU command environment. It is an Open System and FOSS as it follows the standards and you can get it for free, modify, and share freely. The different versions are referred to as "distros" or distributions (i.e., Red Hat, Ubuntu, Knoppix).

Cygwin is an Open Source collection of GNU tools that runs under Windows. Windows handles all the kernel activities like files and memory. The commands look like you are using UNIX but there are some subtle differences.

RIGHT TOOL, RIGHT PLACE

I'm a strong proponent of using the right tool on the right system. While SAS is a great tool, it isn't always the right one, especially as you are trying to perform initial analysis on a physical file. The location of the file may help determine the tools – you could process it on the source, (rather than waiting for the file to be transmitted), or you could load it to another system. Another consideration is that there may be better tools than you are using now – you don't always have to write code and learning skills that work on multiple systems can be good for your career.

You have a few alternatives. You can run under Windows, Microsoft Subsystem for UNIX-based Applications (SUA), Windows Subsystem for UNIX (Beta under Windows 10), commercial UNIX (whether spelled Linux, AIX, HPUX, Solaris, etc., etc., etc.), Cygwin under Windows, or even Proprietary operating systems like IBM's Z/OS. Those options end up being segregated by cost and availability. At low cost you have supported Linux (from Oracle, Red Hat, or even IBM) and supported Cygwin under Windows (also Red Hat). Essentially free are unsupported distros of Linux (which can be run directly on hardware or under a virtual window or virtual machine), Cygwin under Windows, Microsoft Subsystem for UNIX-based Applications (SUA) – now depreciated, and Windows Subsystem for UNIX (Beta under Windows 10). At the high cost level, you have vendor specific UNIX (like Solaris on Sun/Oracle hardware) and other proprietary operating systems.

It really does depend on your organization. If you need support (most firms) you have one set of choices. Some highly regulated industries cannot handle FOSS – but do allow supported Linux. Some other considerations include existing IT controls and systems as well as the general political structure.

I am going to focus on Cygwin in examples for several reasons including the availability of both free and supported versions, even stodgy heavily regulated organizations allow it, Windows Subsystem for UNIX is still Beta, Microsoft Subsystem for UNIX-based Applications (SUA) is depreciated.

I consider Cygwin the gateway drug of choice for learning UNIX.

We're going to answer a few basic questions:

- What does the file look like at the beginning?
- Need to sample a few rows
- Need to perform special processing on first row (header)
- What does the file look at the end?
- Need to see what it looks like at the end
- Need to perform special processing on last row (trailer)
- Have a bad record too far in file to view in editor
- Get a file in another character set (like EBCDIC)
- Get a file that contains binary data
- Get a file that breaks your tool (XML Engine, ETL Tool)
- Need to understand the layout
- Need to perform basic sanity checks
- Do you really trust your data vendor?

WHAT DOES THE FILE LOOK LIKE?

When you're given a new file, you need to determine what it looks like. You don't want to write code until you understand the layout (even when provided a record layout). There are multiple ways of doing this in UNIX.

YOU NEED TO SEE A FEW SCREENS WORTH

When you want to look at the first few screens and don't want to deal with an editor – especially for large files that will take a long time to load (or when the file isn't very viewable in the editor). In this case, you can use the following commands (availability varies by flavor or distro):

```
more FILENAME
```

```
less FILENAME
```

```
pg FILENAME
```

Any of these will display the file a screen full at a time based on the file contents and size of available screen.

YOU NEED TO SEE THE BEGINNING OF A FILE

When you want to look at the first N records of a file, you can use the head command. The default is 10 lines but you can select any number. This also comes in handy if you want to strip off just the first record for special processing (handling a header record for instance). The following is the syntax involved to get the first 10 lines:

```
head FILENAME
```

When you want a specific number of lines you can use either of the following forms:

```
head -2 FILENAME
```

```
head –n 2 FILENAME
```

Which will produce the same result:

```
LINE 1
```

```
LINE 2
```

YOU NEED TO SEE THE END OF A FILE

When you want to look at the last N records of a file, you can use the tail command. The default is 10 lines but you can select any number. This also comes in handy if you want to strip off just the last record for special processing (handling a trailer record for instance). The following is the syntax involved to get the last 10 lines:

```
tail FILENAME
```

When you want a specific number of lines you can use either of the following forms:

```
tail -2 FILENAME
```

```
tail –n 2 FILENAME
```

Which will produce the same result:

```
LINE 99
```

```
LAST LINE
```

YOU NEED TO SEE A RECORD SOMEWHERE IN THE MIDDLE OF A FILE

A common occurrence is trying to look for a specific record somewhere in the middle of a file – for instance, when you are told that there is bad data at a particular record number. By combining commands, we can strip out specific records. These are generally quicker than starting up an editor. The following example shows you how to get records around #50:

```
head -51 FILENAME | tail -3
```

Sends the first 51 records to the tail command which selects the last three resulting in the following output:

```
LINE 49
LINE 50
LINE 51
```

To get just record number 50:

```
head -50 FILENAME | tail -1
```

Will send the first 50 records to the tail command which selects the last record resulting in the following.

```
LINE 50
```

You could also reverse the process to feed the last N records (tail) into head to get the first record.

YOU NEED TO SEE A RECORD BASED ON A KEY

Another common occurrence is trying to look for a specific record based on some key value but I don't know the record number. This works best when you have a unique key but it also works when there are multiple hits that are a small percentage of the file. If necessary, you can search for additional keys within that subset. If you have any experience with DOS, you may remember the find command. The following examples find results based on a specific key:

```
grep –i "line 50" FILENAME
```

```
grep  "LINE 50" FILENAME
```

Both of these commands will produce the same result:

```
LINE 50
```

The difference is that the first is performing a case insensitive search – "line" will match to "line", "LINE", "Line", "LiNe", etc.

LINE 50

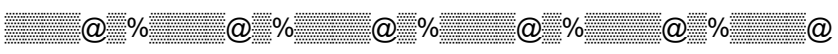
SAVING RESULTS

While it is handy to use these commands to send output to the screen, that isn't very useful over time. There are many times you want to save the results of these commands. There are multiple methods for this that comes under the generic term "redirection". You may remember some of these from the old DOS days. UNIX did not inherit these from DOS, it was the other way around!

- > sends output ("STDOUT") to a new file
- >> appends output ("STDOUT") to an existing file or creates a new file if it doesn't already exist
- 2> and 2>> sends error messages ("STDERR") to a file
- | sends output ("STDOUT") to the input of another command – shown with the head and tail command
- < gets "keyboard" input ("STDIN") from a file
- << gets "keyboard" input ("STDIN") from the current script

DEALING WITH FILES IN ANOTHER CHARACTER SET (LIKE EBCDIC)

Receiving a file in a "foreign" character set (like receiving an EBCDIC file when you work in ASCII). If you look at the file in an editor or even head or more command, you'll see garbage:



UNIX tools include a means of getting more information out of that file via the od command. In the following example, I am using the –c option to see printed characters along with -x to see output in hexadecimal:

```
od -c -x ebFILE
```

Which will show me the following:

```
0000000   ▒   ▒   ▒   ▒   @   ▒   %   ▒   ▒   ▒   ▒   @   ▒   %   ▒   ▒
            d3c9    d5c5    40f1    25d3    c9d5    c540    f225    d3c9
```

along with the remainder of the file.

If I recognize that this file is in ASCII or just guess, I can convert the entire file to ASCII using the dd command as shown::

```
dd if=ebFILE conv=ascii
```

Provides me with output that I can actually work with:

```
LINE 1
LINE 2
LINE 3
```

Along with the rest of the file.

This ends up being an imperfect solution if there is binary, zoned decimal, or packed decimal fields within the file. But this provides you some understanding of the content.

A similar process is used when receiving an ASCII file on an IBM mainframe – you can still view it in hexadecimal. With the dd command, you would use conv=ebcdic instead.

DEALING WITH FIXED LENGTH RECORD FILES

I often receive files that do not contain record separators (Carriage Return/Line Feed under Windows or just Line Feed under UNIX) known as “blocked” or “fixed record length”. When I look at the data, I see something like the following:

Looks like:

```
LINE 01LINE 02LINE 03LINE 04LINE 05LINE 06LINE 07
```

But we know each data record is 7 bytes long (because the file creator told us or we counted). I can read it in using SAS and fixed record lengths or I can convert it into "variable length" with separators:

```
dd conv=unblock cbs=7 if=INFILE
```

The conv option sets the action (unblock) and cbs sets the size resulting in this output:

```
LINE 01
LINE 02
LINE 03
LINE 04
LINE 05
LINE 06
LINE 07
```

More Options for dd

There are a few options for the dd command that can make your life easier:

- count=n number of blocks to copy
- ibs=n bytes per record
- skip=n number of blocks to skip first

- cbs=n unblock bytes per record (as shown in the prior example)

You can specify the input and output filename if you choose:

- if=INPUT_FILE
- of=OUTPUT_FILE

You can also combine options – for instance, converting to ASCII and unblocking in the same command:

```
dd conv=unblock,ascii
```

Just remember what I said about binary data in your file...

WORKING WITH BINARY FILES

You have a few options when receiving files that contain binary data. COBOL formatted records often contain binary data (but those are not the only source. The biggest problem is that the binary data itself is not exactly "human readable" like text. You'll likely need to write code to actually read the data but sometimes you just need to look at the file.

Of course, the Object Dump (od) command comes in handy as described previously.

One approach that has worked for me when dealing with binary data in EBCDIC files within an ASCII environment is a hybrid approach:

1. Use dd perform the conversion on the entire file
2. Read both the original file and the converted file
3. Use the original file for binary, packed, zoned decimal fields
4. Use the converted file for text fields

Of course, the other approach is to code the conversion of text within your program and process the rest in their native format.

WORKING WITH FILES THAT BREAK YOUR TOOL

BREAKING THE SAS XML ENGINE

We received a new XML file from an external vendor that included a tag that broke the SAS XML Engine:

```
<?xml version="1.0"?>
```

Every time we tried to process one of their files, the XML engine reported an error. We contacted the vendor who said the file was fine. The XML engine handled the file without the tag. So I wrote a bit of utility code that removed the tag from the input file:

```
awk '{gsub("\\<\\?xml version=\"1\.0\"\\?>", "");print $0;}'  FNAME > XMLNAME
```

This is a very fancy way of saying "convert every appearance (global substitute) in a record of that string to nothing and output the resulting record."

The awk language was written by Aho, Weinberger, and Kernighan – early developers of UNIX and the C programming language. It should be no surprise that the language is very similar to C.

In awk, $0 is the entire record as a string and there is an implied read loop (much like a SAS set). In this case, input comes from FNAME while output is written to XMLNAME.

To process a 4,720,941,011 byte file takes about 4 minutes using about 1/3 CPU core – 19 MB/second

BREAKING OUR ETL TOOL

Our ETL tool was faulting on random records that contained NUL characters in the feed from our transaction processing system (TPS). Of course this happened at 2 AM!

The TPS received the data from the web interface. When we tried to resolve the issue, both of those teams disclaimed responsibility as they had showed no errors in their processing. A little research showed that the ETL Vendor purposely included that error as a feature; they didn't think a text field should contain a NUL character.

The solution was to remove any NUL characters from the input file:

```
sed 's/\x0//g' < INFILE > OUTFILE
```

The sed, or Stream editor, command takes in INFILE (via redirection), substitutes nothing for the NUL character (written as \x0) every place seen, and writes the output to OUTFILE (again via redirection).

The sed command is also rather fast – to process a 62,021,760 byte file takes about 10 seconds (CPU not noted) – 31 MB/second

I later learned that I could've also used the translate command instead:

```
tr '\000' '' < INFILE > OUTFILE
```

```
tr -d '\000' < INFILE > OUTFILE
```

The first example translate the NUL (written as \000) to nothing while the second deletes (-d) any NUL.

Tool Choices (awk vs sed vs tr)

In the first situation I used awk and sed in the second. While it seems like I could've used either (both are standard tools). But there are differences in capabilities – sed has a limited record length (around 4,000 bytes) while awk has a record length limit matching the C long size (or total available real/virtual memory). sed uses less memory and is a bit faster with a limited "programming" language known as regular expressions. awk has a full feature programming language – C with Strings. The tr command would have done the job but, in all honesty, I didn't think about it.

While researching this paper, I learned that sed will automatically drop NUL characters; the following tells sed to write INFILE to OUTFILE:

```
sed ":w" < INFILE > OUTFILE
```

There is actually a real reason why I did not use sed. The average record length was almost 88,000 characters wide which exceeded sed's limit. It also exceeded SAS's string limit of 32,767 characters so I could process the records as strings.

BREAKING OUR INFILE/PROC IMPORT CODE

We regularly received a CSV file onto our UNIX server from the source system that loaded just fine every month until one particular example. When read into the program, no records were found. When looking at the file under Windows in Notepad, it looked fine. It even loaded into Excel just fine. Looking at the file in those tools I saw:

```
key,val1,val2
123,1,2
123,3,4
125,5,6
```

So I looked at the file using Object Dump ('od') which showed:

```
0000000   k   e   y   ,   v   a   l   1   ,   v   a   l   2  \n   1   2
           656b    2c79    6176    316c    762c    6c61    0a32    3231
0000020   3   ,   1   ,   2  \n   1   2   3   ,   3   ,   4  \n   1   2
           2c33    2c31    0a32    3231    2c33    2c33    0a34    3231
0000040   5   ,   5   ,   6  \n
           2c35    2c35    0a36
```

Each record was separated with a “newline” (line feed) character. However, the code included the termstr=crlf option which expected a file terminated with both carriage return and linefeed:

```
0000000   k   e   y   ,   v   a   l   1   ,   v   a   l   2  \r  \n   1
           656b    2c79    6176    316c    762c    6c61    0d32    310a
0000020   2   3   ,   1   ,   2  \r  \n   1   2   3   ,   3   ,   4  \r
           3332    312c    322c    0a0d    3231    2c33    2c33    0d34
0000040  \n   1   2   5   ,   5   ,   6  \r  \n
           310a    3532    352c    362c    0a0d
```

Somewhere along the way, the carriage return was lost. The fix was to remove the termstr parameter. This was difficult to debug using Windows tools and even EG because it looked totally normal in notepad, Excel, and EG.

MORE MEANS OF LOOKING AT YOUR FILE

The Object Dump command allows you to see the values of characters in the file. Object Dump (the ‘od’ command) allows a few options that are useful; in the example above, I used -x -c:

-x – hexadecimal

-o – octal

-a – Characters with named control characters

-c – Characters with escaped control characters

This allows me to see special characters (the Carriage Return shown as ‘\r’ – normal UNIX practice) as well as the hexadecimal value which is particularly helpful when there is no easy representation like the Carriage Return. You can also pipe the output of ‘od’ into ‘grep’ to search for specific characters.

Another handy command if you’re looking for text is the ‘strings’ command which will show any collection of three or more printable characters. I’ve used that command to discover “hidden” options within programs.

You need to do a bit more work if you want to look at specific fields or record positions. We certainly could count positions in the output of ‘od’ but that’s a lot of work.

PULLING FIELDS

We received some new fields from our vendor. They provided a data dictionary as follows:

LATITUDE	2563	2570	8	N	The property location based upon the "latitude" component of latitude/longitude coordinates. The latitude is stored in decimal degrees to 6 decimal places (e.g., 12.123456) and will always be positive north of the North American continent.
LONGITUDE	2571	2579	9	N	The property location based upon the "longitude" component of latitude/longitude coordinates. The longitude is stored in decimal degrees to 6 decimal places (e.g., 123.123456) and will always be negative on the North American continent.

So we treated these fields as plain old SAS numeric fields. But when reading the file, SAS reported input errors. So I need to look at the actual data in the original file. Note that these start at position 2563 in the record; that would be a royal pain counting through the output of ‘od’ to find it. I have a few other choices:

```
cut -b 2563-2570,2571-2579 FILENAME
```

```
awk '{print " lat " substr($0, 2563, 8), " long " substr($0, 2571, 9);}'
FILENAME
```

The results of these two commands, respective, are:

```
4271275707387955R             lat 42712757  long 07387955R
4271341007387929K             lat 42712757  long 07387955R
4271020507387605N             lat 42710205  long 07387605N
```

As you can clearly see, these do not correspond to the descriptions. Contrary to the document, the format of Latitude is COBOL zoned decimal: 99v9(6), USAGE DISPLAY and Longitude is COBOL signed zoned decimal: S999v9(6), USAGE DISPLAY.

The cut command allows you to specify byte positions to display while awk allows you to write a simple program to print the information you want.

Now I know that I need to specify ZD format in my SAS program.

NEED TO PERFORM BASIC SANITY CHECKS

Unfortunately, I can't trust data providers – it doesn't matter if they are internal or vendors. Basic data controls can be implemented in SAS or with UNIX/Linux tools. You really need to protect yourself with Sanity Checks:

- Header/trailer processing
- Raw Record counts
- Key Record counts
- Record lengths
- Field counts
- Breaking up a file

HEADER/TRAILER PROCESSING

Header and trailer records let you know that you received a full file – providing information by their very existence and more as they grow in complexity with record counts and sums of specific fields.

The general process is as follows:

1. Read header record, save important values
2. Count data records and sum appropriate fields
3. Read trailer record, compare important values with your calculated
4. Abort on differences

Of course, it would make sense to perform this work within your SAS code but there are times that you have to perform this function where the tool is not available. Remember, sometimes all you have is a hammer. For instance, when the bank I was working for implemented direct ACH (Automated Clearing House) transaction processing rather than going through a third party, we needed a process to validate those files. Because of system/tool and staff availability, the ACH inbound/outbound file validator was written on a UNIX/Linux server requiring over 1300 lines of awk code. The code checked the following behavior within the file:

- File header/trailer – Credit & Debit sums, hash, record counts
- Batch header/trailer – Credit & Debit sums, hash, record counts
- Various parent/child record combinations (limited to 94 byte records)

- And check digits on various records/fields

While this certainly might run faster if it was written in SAS or a compiled language like C, the performance was good enough that there was no reason to recreate it. If performance was a problem, we could've used a conversion program to get C (awk2c) or Perl (awk2perl) code!

RAW RECORD COUNTS

There are plenty of situations where you need raw record counts – or the number of words or characters in the file. Your application could perform that work or can do so manually. As part of testing, since we are in a large team, confirm results by comparing SAS dataset counts, flat file counts within the log, and with counts of the physical file.

The wc command will perform this important function for you:

```
wc –l FILE
```

```
wc FILE
```

Will produce the following output (just sample):

```
100 FILE


100     200     793 FILE
```

KEY RECORD COUNTS

We can get much more complex by counting records with a specific string or key. In this specific case, we needed to count number of records with same value in first field. This would be a royal pain to find records manually. So I wrote a little program instead:

```
sort INPUT_FILE | awk 'BEGIN{c=0;o="XXXXXXXXXXXXXXXXXX";} {if ($1 != o) {
print c,o; o=$1; c=1; } else c++;} END{print c,o,"end";}' | sort >
OUTPUT_FILE
```

Resulting output (just sample):

```
10 ONE

15 TWO

15 Three

16 Four
```

I will break each of these down.

1. Sort the input (to put all values for same first field together) and pipe the output to the next program:

   ```
   sort INPUT_FILE |
   ```

2. Execute awk to count records with same value (control break) and pipe the output to the next program:

   ```
   awk 'BEGIN{c=0;o="XXXXXXXXXXXXXXXXXX";} {if ($1 != o) { print c,o;
   o=$1; c=1; } else c++;} END{print c,o,"end";}' |
   ```

3. And finally, sort the output so the records are in count order:

   ```
   sort > OUTPUT_FILE
   ```

We should look at the awk program itself a bit more closely:

```
BEGIN{
   c=0;
```

```
   o="XXXXXXXXXXXXXXXXXX";}
{
   if ($1 != o)
   {
      print c,o;
      o=$1; c=1;
   }
   else c++;
}
END{print c,o,"end";}
```

BEGIN executes before the first record is read and sets a few variables. END executes after end-of-file is reached and prints off the last summary value. Within {} is the main part of the program, an implied read loop, that is executed for every record read. Awk automatically parses the read data into fields based on the field separator (a space by default). $0 is the entire input record, $1 is the first field, $2 the second, etc. So this program compares the first field to the saved value; if they differ, a control break took place – the resulting value printed and controls reset to the new value. If the values are the same, the count is incremented by one.

I have a few other awk examples.

RECORD LENGTH CHECKS

There is often a need to find or catch specific “bad” records to prevent bad data loads. We got the following error from our ETL (Extract, Transform, Load) tool, which wasn’t SAS:

```
Source file contains: 67099, destination table contains: 66979

FR_3016 Row [346]: Record length [18398] is longer than line sequential
buffer length [3005] for FILENAME.dat.  Record will be rejected.
```

We can’t just allow the process to skip the record – after all, it could be really important. And if this happens once in the load, there is a high probability there will be more. Rather than fixing one record, rerunning, and catching the next error, we want to find them all before rerunning. It would be a royal pain to find records manually, so a quick program will find them for us:

```
awk '{ if (length($0) > 3004) print NR, length($0);}' FILENAME.dat
```

Resulting output (just sample):

```
346 18398
10000 10000
10101 12357
13031 8192
```

The awk program compares the length of each record to a specific value and printed the record number (NR) and that length for each that meet the criteria.

FIELD COUNT CHECKS

Another common task is confirming the number of fields in the record of a delimited file. And you can choose the delimiter. You certainly could do this manually but that would require a lot of work. Instead, another awk program does the work for us:

```
awk 'BEGIN {FS=","; max=0;} {print "NF " NF; if (NF > max) max=NF;} END
{print "max " max;}' csv1.csv
```

Resulting output (just sample):

```
NF 9
NF 10
```

```
NF 9
NF 10
NF 9
max 10
```

In the awk program, FS sets the field separator (comma instead of a space). The program prints the number of fields for each record and saves the highest value. After the last record is read, it prints that highest value.

BREAKING UP A FILE TO MAKE IT MORE READABLE

We already had examples showing how to pull specific fields out of file using awk and cut. Another process would to be splitting records to a more readable size. For instance, if you have a XML file with 88,000 byte record length (like my earlier example), you could split the file into individual tags which would be more readable:

```
awk 'BEGIN {FS="<";} {for (i=1;i<=NF;i++) if (length($i) > 0) print "REC=" NR
" Field=" i " : <" $i;}' FILENAME
```

Resulting output (just sample):

```
REC=1 Field=12 : <wpt lat="40.224917" lon="-75.774154">
REC=1 Field=13 : <name>031201 CAR
REC=1 Field=14 : </name>
REC=1 Field=15 : <sym>Pin, Blue
```

The awk program sets the field separator and then prints every individual field showing the record number (NR), field number, and the actual text. Since “<” is the field separator, it is consumed in the process – not part of the field itself so has to be included in the print statement.

You can get just the tags if you don’t want the record/field number included:

```
awk 'BEGIN {FS="<";} {for (i=1;i<=NF;i++) if (length($i) > 0) print "<" $i;}'
FILENAME
```

SOURCES OF MORE INFORMATION

The best place for more information is the built in Manual:

```
man man
```

```
man -k transfer
```

You can check the manual for a specific command (in the example above, the man command itself) or search for a keyword (transfer in the example). While this may not be as good as a web search, it is more accurate because it is providing detail for the exact version you are using!

I spend a fair amount of time emphasizing the use of ‘man’ to my students. My first response when asked a question is “What does ‘man’ say?” Then I will help the students understand what the text means. I really want them to go there first.

There is a reason everyone in IT knows what "RTFM" means!

COMMAND LINE HELP

Most commands will also provide help if you provide an invalid option, use -?, or use - -help as parameters. For instance, with od:

```
od -?
```

Results in the following output under GNU:

```
od: unknown option -- ?
```

```
Try `od --help' for more information.
```

Or using - - help under GNU:

```
od --help
```

Results in:

```
Usage: od [OPTION]... [FILE]...
  or:  od [-abcdfilosx]... [FILE] [[+]OFFSET[.][b]]
  or:  od --traditional [OPTION]... [FILE] [[+]OFFSET[.][b] [+][LABEL][.][b]]

Write an unambiguous representation, octal bytes by default,
of FILE to standard output.  With more than one FILE argument,
concatenate them in the listed order to form the input.
With no FILE, or when FILE is -, read standard input.
```

Or od -? under Solaris:

```
usage: od [-bcCdDfFoOsSvxX] [-] [file] [offset_string]
       od [-bcCdDfFoOsSvxX] [-t type_string]... [-A address_base] [-j skip]
[-N count] [-] [file...]
```

Of course, there's always Google or Bing...

SOME OPERATING SYSTEM BASICS

DIRECTORY NAVIGATION

You will find UNIX/Linux directory structure very familiar because Windows shares many similarities include tree-metaphor structure. One big difference is that / is used instead of Windows' \. In addition, there are no drive letters and all disks are attached to the main directory structure ("mounted"). Back in the old days of DOS 3.1 – 6 there was a 'join' command that would allow you to attach a disk to the main tree structure. Cygwin and Windows 10 Bash are special cases because they are running under the Windows operating system.

Directory Navigation Commands:

- 'cd' – change directory
- 'cd' with no parameters – change to home directory
- 'cd -' – change to previous directory (where you were before)
- 'pwd' – display the current directory (Windows 'cd' does this)

SEARCHING DATA

I mentioned 'grep' before; this is the command to use when you want to search the contents of a file. The general form is:

```
grep OPTIONS "search string" file_name_wildcarded
```

With the following common OPTIONS:

- -v (invert match – show lines that don't match)
- -i (case insensitive search)
- -H (show file names)
- -n (show line number)

- -r (recurse through subdirectories)

The “search string” includes regular expressions (entire books have been written to explain that topic) but simple examples are:

- ^ (beginning of line)
- $ (end of line)
- . (any single character)
- [ABCabc] (single character that matches any of A, B, C, a, b, c)

DIRECTORY CONTENTS (LIST FILES)

The ‘ls’ command is the primary method for looking at listings of files; it allows many options including:

- -a – show all files; by default, files that begin with “.” are “hidden”
- -l – long display showing full information
- -1 – list in one column
- -t – sort by time
- -r – reversed (used with -t)
- -R – recurse through subdirectories

One big difference between Windows and UNIX/Linux is that commands under Windows have to expand wildcards themselves while the UNIX/Linux shell will expand it for you on the command line. As a result, you could end up with too many results to include on the command line itself. Watch out for the “Arg list too long” error which means that your wildcard list got too large.

OTHER USEFUL COMMANDS

Some of the other useful commands are:

- ‘cp’, ‘mv’, ‘rm’ – copy, move or rename, or delete (remove) a file
- ‘mkdir’, ‘rmdir’ – create (make) or delete (remove) a directory
- ‘exit’, ‘logout’ – leave the system
- ‘du’, ‘df’ – display storage usage or filesystem available
- Last, but certainly not least, ‘man’

CONCLUSION

While SAS includes many capabilities, there are times you should use a different tool. There is a large set of useful commands available under the base UNIX/Linux operating system to look at your data. It really is up to you to decide where the processing should take place..

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