Linux tools for text processing

Robert Sinkovits, PhD
Director of Education
San Diego Supercomputer Center

https://github.com/sinkovit/COMPLECS

Table of contents

This session is intended for anyone who …

… needs to post-process text files to extract run times, results or information about the hardware used for the calculations. The skills taught here will also be useful when constructing workflows, preparing high-throughput computing workloads or pre-processing data to get it into the correct format.

The Linux environment provides a wide range of tools for sorting, splitting and manipulating files along with tools for extracting content based on location (line number) or value.

In this sessions we'll cover the tools split, head, tail, awk, sed, grep and paste. Some of these tools have many options, so we'll just look at the basics to keep things more manageable.

Can’t I just do this all by hand? Do I really need to learn yet another tool (or tools)?

While these text manipulation tasks *could* be done by hand, the process can be time-consuming, tedious and, worst of all, error prone. This is especially true if they need to be done many times, such as extracting a value from the potentially thousands of files generated during a set of parameter-sweep calculations.

An obvious solution is to automate the process. Sometimes this requires the development of relatively complex parsers that are beyond the ability of non-programmers to write, but often simple Linux utilities are sufficient.

Hands on examples

A subdirectory is provided for each of the Linux tools covered in this tutorial. We recommend that you execute the commands as we go along and, either during the breaks in the presentation or later on your own, experiment with the examples.

The best way to master these tools is to try new things

- Deliberately introduce errors (e.g., leave out quotes, commas or semicolons) to see what happens
- Extend the examples to introduce new features
- Chain the tools together to create more complex workflows
- Apply the tools to your own use cases

Table of contents

head/tail – output first/last part of file

By default, head/tail output the first/last 10 lines of file

```
$ head file.txt
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
```

```
$ tail file.txt
Line 11
Line 12
Line 13
Line 14
Line 15
Line 16
Line 17
Line 18
Line 19
Line 20
```

head/tail – output first/last part of file

As expected, the user can override the default and specify number of output lines

```
$ head -n 7 file.txt
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
```

```
$ tail -n 7 file.txt
Line 14
Line 15
Line 16
Line 17
Line 18
Line 19
Line 20
```

head/tail – output first/last part of file

Can also specify that all but the last NUM lines be output (head -n -NUM) or all lines from NUM onward are output (tail -n +NUM)

```
$ head -n -15 file.txt
Line 1
Line 2
Line 3
Line 4
Line 5
```

```
$ tail -n +16 file.txt
Line 16
Line 17
Line 18
Line 19
Line 20
```

Table of contents

paste – merge lines of files

The paste command combines two or more files line-by-line. By default, tab is used as delimiter, but this can be overridden using -d option. Much more convenient than cutting and pasting into a spreadsheet.

```
$ cat fruits.txt
apple
banana
grape

$ cat colors.txt
red
yellow
purple

$ paste fruits.txt colors.txt
apple  red
banana yellow
grape  purple
```

```
$ cat fruits.txt
apple
banana
grape

$ cat colors.txt
red
yellow
purple

$ paste -d ',' fruits.txt colors.txt
apple,red
banana,yellow
grape,purple
```

Table of contents

sort – sort lines of text files

Without arguments, sort is done by lexicographical order. Be aware that the handling of upper and lower case letters will depend on the LC_ALL environment variable

```
$ cat unsorted1.txt
Banana
apple
Grape
pear
apple
peach
Banana
Grape
orange
```

```
$ sort unsorted1.txt
apple
apple
Banana
Banana
Grape
Grape
orange
peach
pear
```

```
$ export LC_ALL=C
$ sort unsorted1.txt
Banana
Banana
Grape
Grape
apple
apple
orange
peach
pear
```

sort – sort lines of text files

Can also choose the field to be used for sorting using the -k option, with the -n option used to sort by numeric value. In the second example, sorting numerically is probably the desired behavior.

```
$ cat unsorted2.txt
banana 17  taco
apple   6  pizza
grape  23  hotdog
pear   35  pretzel
peach  12  popcorn

$ sort -k3 unsorted2.txt
grape  23  hotdog
apple   6  pizza
peach  12  popcorn
pear   35  pretzel
banana 17  taco
```

```
$ sort -k2 unsorted2.txt
peach  12  popcorn
banana 17  taco
grape  23  hotdog
pear   35  pretzel
apple   6  pizza

$ sort -k2 -n unsorted2.txt
apple   6  pizza
peach  12  popcorn
banana 17  taco
grape  23  hotdog
pear   35  pretzel
```

sort – sort lines of text files

Sort can be restricted to listing unique values by specifying -u option

```
$ cat unsorted1.txt
Banana
apple
Grape
pear
apple
peach
Banana
Grape
orange
```

```
$ sort unsorted1.txt
apple
apple
Banana
Banana
Grape
Grape
orange
peach
pear
```

```
$ sort -u unsorted1.txt
apple
Banana
Grape
orange
peach
pear
```

Table of contents

split – split a file into pieces

By default, split breaks a file into chunks of 1000 lines, using naming convention xaa, xab, xac, … Options available to specify different chunk sizes, prefixes and numbering

```
$ split genome.fasta
$ wc -l *
   9991 genome.fasta
   1000 xaa
   1000 xab
   1000 xac
   1000 xad
   1000 xae
   1000 xaf
   1000 xag
   1000 xah
   1000 xai
    991 xaj
 19982 total
```

```
$ $ split -l 2000 genome.fasta genome_
$ wc -l *
 2000 genome_aa
   2000 genome_ab
   2000 genome_ac
   2000 genome_ad
   1991 genome_ae
   9991 genome.fasta
  19982 total
```

split – split a file into pieces

For files that have a well-defined structure, such as FASTA files used for genomic data, the default record separator (newline character) may lead to undesired behavior. In this case, we want to keep the annotation and sequence together

```
$ head -1 genome_aa
>sp|Q61151|2A5E_MOUSE Serine/threonine-protein phosphatase 2A 56 ...

$ head -1 genome_ab
SYTTYMKEEVDRYRITIGNKTCVFEKENDPSVMRSPSAGKLIQYIVEDGGHVFAGQCYAE

$ head -1 genome_ac
PLGGGREVWFGFHQSVRPAMWNMMLNIDVSATAFYRAQPIIEFMCEVLDIQNINEQTKPL
```

split – split a file into pieces

Fortunately, we can specify a different record separator using the -t option to control how the file is split. Be careful that delimiter only occurs where expected and note that it is not included in output (we'll show later how to fix this)

```
$ split -l 200 -t '>' genome.fasta genome_

$ head -1 genome_aa
>sp|Q61151|2A5E_MOUSE Serine/threonine-protein phosphatase 2A 56 ...

$ head -1 genome_ab
sp|O35674|ADA19_MOUSE Disintegrin and metalloproteinase domain ...

$ head -1 genome_ac
sp|Q91WF3|ADCY4_MOUSE Adenylate cyclase type 4 OS=Mus musculus ...
```

Table of contents

grep – print lines that match a pattern

At its simplest, grep returns all lines in a file containing the search string. Upper and lower case letters are different characters and you'll need the -i flag if you want your matches to be case insensitive

```
$ cat file1.txt
apple banana pear
cherry banana peach
orange lemon lime
apple Banana pear
cherry Banana peach
pear kunquat grape
grape guava lime
pear peach kiwi
```

```
$ grep banana file1.txt
apple banana pear
cherry banana peach

$ grep Banana file1.txt
apple Banana pear
cherry Banana peach

$ grep –i banana file1.txt
apple banana pear
cherry banana peach
apple Banana pear
cherry Banana peach
```

grep – print lines that match a pattern

grep provides options for listing lines that occur before or after the matching line. This is extremely useful when the desired content can vary, but occurs in a known location relative to matching pattern.

```
$ grep -B 3 kumquat file1.txt # print three lines before matching line
orange lemon lime
apple Banana pear
cherry Banana peach
pear kumquat grape

$ grep -A 2 kumquat file1.txt # print two lines after matching line
pear kumquat grape
grape guava lime
pear peach kiwi
```

grep – print lines that match a pattern

grep can do an inverted match to print lines that do not match the pattern

```
$ grep lime * # contain lime
file1.txt:orange lemon lime
file1.txt:grape guava lime
file2.txt:orange lemon lime
file2.txt:pear lime grape
file2.txt:grape guava lime
```

```
$ grep -v lime * # do not contain lime
file1.txt:apple banana pear
file1.txt:cherry banana peach
file1.txt:apple Banana pear
file1.txt:cherry Banana peach
file1.txt:pear kumquat grape
file1.txt:pear peach kiwi
file2.txt:apple banana pear
file2.txt:cherry banana peach
file2.txt:apple Banana pear
file2.txt:cherry Banana peach
file2.txt:pear peach kiwi
```

grep – print lines that match a pattern

Other options include printing the names of files that contain or don't contain the match and listing counts of matching lines

```
$ grep lime *
file1.txt:orange lemon lime
file1.txt:grape guava lime
file2.txt:orange lemon lime
file2.txt:pear lime grape
file2.txt:grape guava lime

$ grep -c lime *
file1.txt:2
file2.txt:3
```

```
$ grep -l kumquat * # contain match
file1.txt

$ grep -L kumquat * # don't contain match
file2.txt
```

grep – print lines that match a pattern

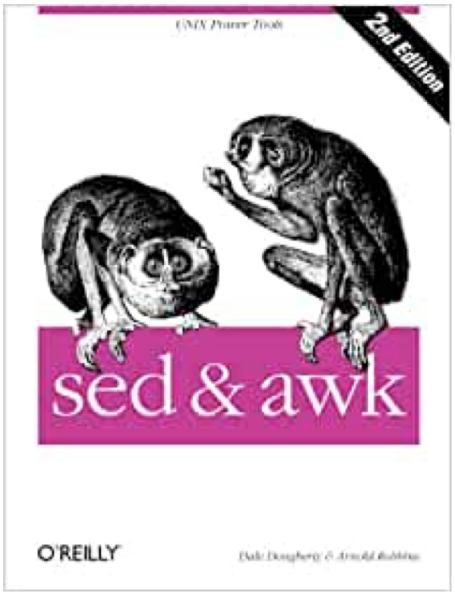
Up to this point, our pattern has been a string literal (e.g., lime or banana), but grep can recognize more complex patterns that use character classes, anchors and even regular expressions

```
$ grep pear file1.txt     # All lines containing pear
apple banana pear
apple Banana pear
pear kumquat grape
pear peach kiwi

$ grep '^pear' file1.txt  # Lines starting with pear (uses ^ anchor)
pear kumquat grape
pear peach kiwi

$ grep 'pear$' file1.txt  # Lines ending with pear (uses $ anchor)
apple banana pear
apple Banana pear
```

Table of contents

sed – stream editor for filtering and transforming text

sed is an extremely powerful editor that operates on a stream of data. We could easily spend days exploring sed and entire books have been written about it, but we'll just look at simple examples of two commonly used features.



http://sed.sourceforge.net/

http://sed.sourceforge.net/grabbag/tutorials/

sed – stream editor for filtering and transforming text

sed can be used to apply a substitution to each line of a file. While *some* of these operations can easily be done using a global replace in a traditional editor (vi, emacs), sed has the advantage that it can be scripted. In addition, since it works on a stream of data, the entire file does not need to fit into memory.

```
sed 's/replace-first-occurrence-of-this/with-this/'  [input-file]
sed 's/replace-every-occurrence-of-this/with-this/g' [input-file]
```

sed – stream editor for filtering and transforming text

```
# First instance in each line
$ sed s'/pear/XXXX/' file1.txt
XXXX apple banana grape
apple banana grape XXXX
XXXX apple pear grape
apple XXXX banana pear
XXXX apple banana pear

# All instances in each line
$ sed s'/pear/XXXX/g' file1.txt
XXXX apple banana grape
apple banana grape XXXX
XXXX apple XXXX grape
apple XXXX banana XXXX
XXXX apple banana XXXX
```

```
# Instances at start of each line
$ sed s'/^pear/XXXX/' file1.txt
XXXX apple banana grape
apple banana grape pear
XXXX apple pear grape
apple pear banana pear
XXXX apple banana pear

# Instances at end of each line
$ sed s'/pear$/XXXX/' file1.txt
pear apple banana grape
apple banana grape XXXX
pear apple pear grape
apple pear banana XXXX
pear apple banana XXXX
```

sed – stream editor for filtering and transforming text

sed can also print out specific lines. There are a lot of options and we'll just focus on a few things that aren't easy to do using other tools

```
$ sed -n '5'p file2.txt   # print fifth line
This is line 3

$ sed -n '3,5'p file2.txt # print lines 3-5
This is line 3
This is line 4
This is line 5

$ sed -n '2~3'p file2.txt # print every third line starting at line 2
This is line 2
This is line 5
This is line 8
```

sed – stream editor for filtering and transforming text

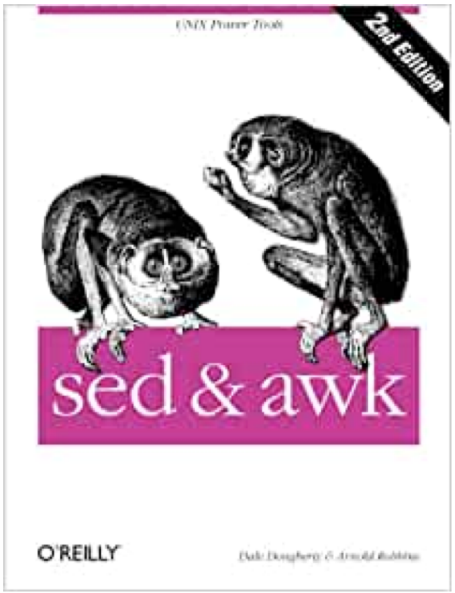
Like all other commands that write to stdout, sed output can be directed to a file. There's just one gotcha – since sed streams through the file rather than storing in memory, redirecting to original file doesn't work. Write to temp file, or even better use the -i option to edit in place

```
$ head -2 file1.txt > file3.txt
$ sed s'/pear/XXXX/' file3.txt > file3.txt
$ cat file3.txt
[no output]

$ head -2 file1.txt > file3.txt
$ sed -i s'/pear/XXXX/' file3.txt
$ cat file3.txt
XXXX apple banana grape
apple banana grape XXXX
```

Table of contents

awk – pattern scanning and processing language

awk is a powerful text processing tool named after its three creators Alfred Aho, Peter Weinberger, and Brian Kernighan. There are a lot of capabilities and we'll limit ourselves to a few of the simple and commonly used ones.



https://www.gnu.org/software/gawk/manual/gawk.html

https://en.wikibooks.org/wiki/An_Awk_Primer/Awk_Command-Line_Examples

awk – pattern scanning and processing language

awk operates on a file, one line at a time, splitting into fields referenced by position ($1, $2, $3, …). The last field can also be accessed as $NF, where NF is the number of fields. Most common use of awk is to print selected fields from each line.

```
$ cat people.txt
Bob dog meatloaf banana blue
Cindy hamster pizza kiwi purple
Mary cat sushi apple green
Susan fish pizza grape yellow

$ awk '{print $1, $3, $NF}' people.txt
Bob meatloaf blue
Cindy pizza purple
Mary sushi green
Susan pizza yellow
```

awk – pattern scanning and processing language

awk can also take a search pattern and include other text in the output

```
$ awk '{print $1 " favorite food is " $3}' people.txt
Bob favorite food is meatloaf
Cindy favorite food is pizza
Mary favorite food is sushi
Susan favorite food is pizza

$ awk '/pizza/ {print $1, $3}' people.txt # limit to lines matching pizza
Cindy pizza
Susan pizza
```

awk – pattern scanning and processing language

By default, awk uses whitespace as the delimiter, but this can be changed using the -F option. This is useful when working with comma separated value (CSV) files

```
$ cat states.txt
California,Sacramento,Valley Quail,Poppy
Texas,Austin,Mockingbird,Bluebonnet
Florida,Tallahassee,Mockingbird,Orange Blossom
New York,Albany,Eastern Bluebird,Rose
Pennsylvania,Harrisburgh,Ruffed Grouse,Mountain Laurel

$ awk -F ',' '{print $1 " state bird is " $3}' states.txt
California state bird is Valley Quail
Texas state bird is Mockingbird
Florida state bird is Mockingbird
New York state bird is Eastern Bluebird
Pennsylvania state bird is Ruffed Grouse
```

awk – pattern scanning and processing language

awk provides basic math capabilities, including square root, log, exp, trig functions and random number generation. Note that operations and commands are separated by semicolons

```
$ awk '{sum=$1+$2; prod=$1*$2; print $1, $2, sum, prod}' numbers.txt
1.1 2.2 3.3 2.42
3.3 4.4 7.7 14.52

$ awk '{sin1=sin($1); cos2=cos($2); print $1, $2, sin1, cos2}' numbers.txt
1.1 2.2 0.891207 -0.588501
3.3 4.4 -0.157746 -0.307333

$ awk '{log1=log($1); sqrt2=sqrt($2); print $1, $2, log1, sqrt2}' numbers.txt
1.1 2.2 0.0953102 1.48324
3.3 4.4 1.19392 2.09762
```

awk – pattern scanning and processing language

1. By default, awk applies operations to each line. BEGIN and END allow you to execute additional code just once at the start or end of the file processing
2. Commands can be broken across multiple lines
3. C-style formatted printing available using printf, explicitly need newline (\n)

```
$ $ awk 'BEGIN {PI=3.14159; print "  x    y   sin(x)  cos(y)"}
> {sin1=sin($1*PI); cos2=cos($2*PI);
> printf "%4.1f %4.1f %7.4f %7.4f\n",
> $1, $2, sin1, cos2}' numbers.txt
  x    y   sin(x)  cos(y)
 1.1  2.2 -0.3090  0.8090
 3.3  4.4 -0.8090  0.3090
```

Table of contents

Case studies

Most of our examples have been simple and maybe even a little contrived, involving things like lists of fruits. In the remainder of this tutorial, we give some more complex real-life examples illustrating how multiple tools can be used together.

This would be a good time to think about your own workloads and how you could use these tools to simplify, automate or streamline your research

Case study – High Performance LINPACK (HPL)

When deploying a new supercomputer, we run thousands of HPL jobs to ensure that all nodes are functioning correctly and delivering expected performance. This requires extracting node name, run time and completion code from the output files.

```
vgr-10-01
...
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR12R2R4       207800   384     6     8            4386.47              1.3638e+03
HPL_pdgesv() start time Wed Apr 13 21:55:29 2022

HPL_pdgesv() end time   Wed Apr 13 23:08:35 2022
...
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=   1.00164594e-03 ...... PASSED
================================================================================
...
```

Case study – High Performance LINPACK (HPL)

Bash script illustrating iteration over files, backticks to capture command output and use of multiple utilities (head, grep, awk, wc) to extract required info from output files

```
echo "node      time    P F"                    # Print header
for file in `ls output.hpl*`                    # Iterate over output files
do
  node=`head -1 $file`                          # Node name from first line
  t=`grep WR12R2R4 $file | awk '{print $6}'`    # Time from 6th field of WR12R2R4 line
  npass=`grep PASSED $file| wc -l`              # Number of passed tests
  nfail=`grep FAILED $file| wc -l`              # Number of failed tests
  echo $node $t $nfail $npass
done
```

HPL output taken from real machine testing, with failure conditions added for illustrative purposes only.

Case study – High Performance LINPACK (HPL)

Bash script illustrating iteration over files, backticks to capture command output and use of multiple utilities (head, grep, awk, wc) to extract required info from output files

```
echo 'node      time    P F'
for file in `ls output.hpl*`
do
  node=`head -1 $file`
  t=`grep WR12R2R4 $file | awk '{print $6}'`
  npass=`grep PASSED $file| wc -l`
  nfail=`grep FAILED $file| wc -l`
  echo $node $t $nfail $npass
done
```

```
$ ./process-hpl.sh
node      time    P F
vgr-10-30 4729.98 0 1
vgr-10-24 4751.87 0 1
vgr-10-11 4824.54 0 1
vgr-10-11 4808.07 1 0
vgr-10-11 4804.13 0 1
vgr-10-08 4883.66 0 1
vgr-10-37 4851.29 0 1
vgr-10-07 4697.88 1 0
```

Case study – BERT natural language processing

In a recent deployment of specialized AI hardware, we ran BERT benchmarks to ensure that all eight of the cards in each node were working correctly and delivering consistent performance. Required extracting node name, card ID, timestamps, percentage correct answers and achieved rate from output files

```
vgr-10-02
...
[1] AIP (hl1) 0000:1b:00.0
...
TBEFORE 1646939242
...
Correct answers: 1466 out of 1724 = 85.03 %
Elapsed time (sec) including enqueue: total - 1.0239 per sample - 0.000593907
Achieved sentences/sec: 1683.77
...
TAFTER 1646940484
```

Case study – BERT natural language processing

Bash script illustrating iteration over files, backticks to capture command output and use of multiple utilities (head, tail, grep, awk, wc) to extract required info

```
for file in `ls -1 out*`
do
  node=`head -1 $file | awk '{print $1}'`
  card=`grep 'AIP (' $file | awk '{print $4}'`
  tstart=`grep TBEFORE $file | awk '{print $2}'`
  tend=`grep AFTER $file | awk '{print $2}'`
  corr=`grep 'Correct answers' $file | tail -1 | awk '{print $8}'`   # Last instance
  ach=`grep 'Achieved sentences' $file | tail -1 | awk '{print $3}'` # Last instance
  t=$(expr $tend - $tstart)                    # Do math to calculate tend - tstart
  echo $node $card $corr $ach $t
done
```

Case study – BERT natural language processing

Bash script illustrating iteration over files, backticks, shell arithmetic and use of multiple utilities (head, tail, grep, awk, wc) to extract required info

```
for file in `ls -1 out*`
do
  node=`head -1 $file | awk '{print $1}'`
  card=`grep 'AIP (' $file | awk '{print $4}'`
  tstart=`grep TBEFORE $file | awk '{print $2}'`
  tend=`grep AFTER $file | awk '{print $2}'`
  corr=`grep 'Correct answers' $file | tail -1 | awk '{print $8}'`   # Last instance
  ach=`grep 'Achieved sentences' $file | tail -1 | awk '{print $3}'` # Last instance
  t=$(expr $tend - $tstart)                    # Do math to calculate tend - tstart
  echo $node $card $corr $ach $t
done
```

```
$ ./process-bert.sh
vgr-10-02 0000:1b:00.0 85.03 1683.77 1242
vgr-10-01 0000:b1:00.0 85.03 1656.87 1342
vgr-10-01 0000:89:00.0 85.03 1687.34 1357
```

Case study – Splitting FASTA file

In the original example where we split a FASTA file, we noted that the delimiter was lost in the first annotation line. We can clean this up by using sed to modify lines that begin with 'sp|', which occurs only in annotations, to instead begin with '>sp|'

```
for file in genome_*
do
    sed 's/^sp|/>sp|/' $file > temp
    mv temp $file
done
```

```
$ head -1q genome_*
>sp|Q61151|2A5E_MOUSE Serine/threonine-protein phosphatase 2A 56 kDa ...
>sp|O35674|ADA19_MOUSE Disintegrin and metalloproteinase domain-containing ...
>sp|Q91WF3|ADCY4_MOUSE Adenylate cyclase type 4 OS=Mus musculus GN=Adcy4 PE=1 ...
>sp|Q8K209|AGRG1_MOUSE Adhesion G-protein coupled receptor G1 OS=Mus musculus ...
>sp|Q68G58|APEX2_MOUSE DNA-(apurinic or apyrimidinic site) lyase 2 OS=Mus ...
```

Table of contents

Conclusions

- Linux provides a powerful suite of tools for text manipulation. They can improve your productivity AND reduce the likelihood of making errors.
- Don’t be discouraged if you didn’t understand everything. We covered a lot of material in a pretty short time. Work through the examples at your own pace until you feel like you’ve truly mastered the content.
- We’ve barely scratched the surface, especially for the more complex tools such as awk and sed. While you can go much deeper, we probably covered most of what you need to know.