Linux System Administration

PM-TRTNB319

Zsolt Schäffer, PTE-MIK

Legal note

These slides form an outline of the curriculum and contain multiple references to the official Red Hat training materials (codes RH124, RH134 and RH254), since students of this subject are also eligible for the named RH learning materials. Some diagrams and pictures originating from the RH courses will show up in this outline as there is an agreement in place which allows for our Faculty to teach this subject based on the Red Hat curriculum.

All other pictures and diagrams sourced from third parties are explicitly marked with a reference to the origin.

Be aware that the mentioned parts and also this series of slides as a whole are property of their respective owners and are subject to copyright law.

Legal note

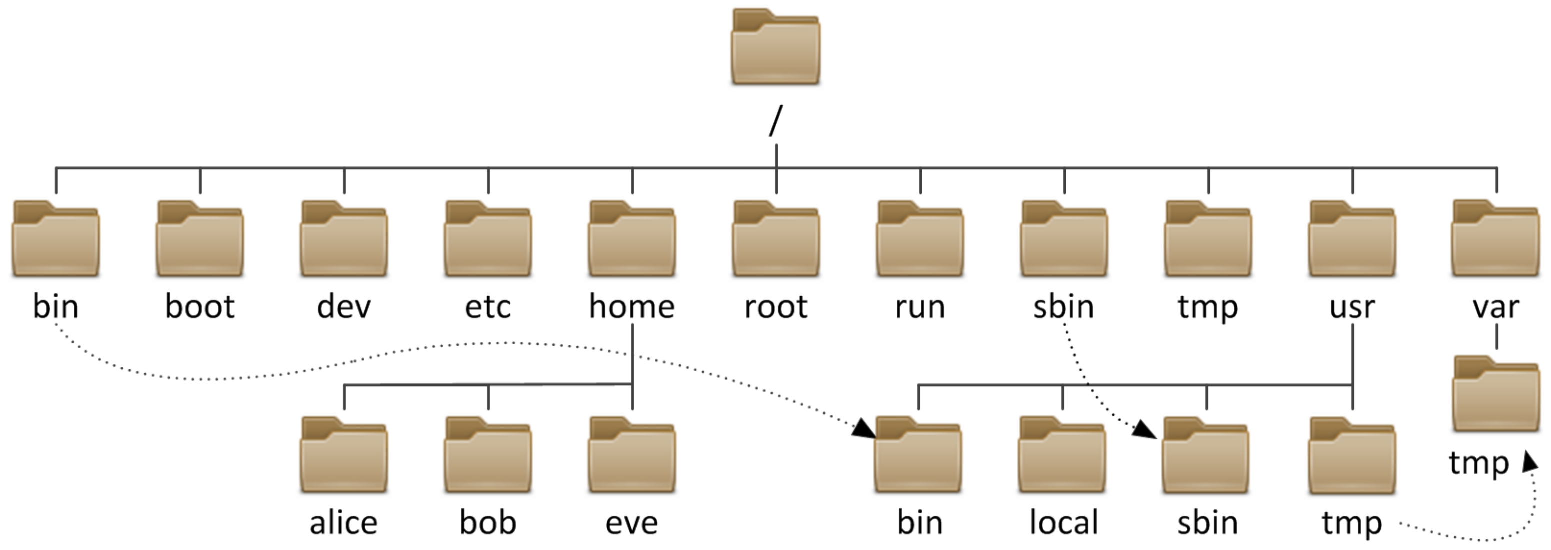
Managing files

Lecture 2

Chapter 1

Filesystem hierarchy

- Linux systems have only one filesystem structure, with one single root point, denoted with a slash (/) called root directory.
- Additional filesystems may be attached to a (sub)directory of the structure. This called mounting.
- The standard filesystem structure is shown on the figure. The arrows represent symlinks. More on filesystems in lecture 5.



Directory structure

- For an explanation of each directory on the figure refer to **RH124 CH 2.1.** As a supplement to the RH learning material some further directories are described here.
  - /proc contains information about all the running processes. It is a virtual filesystem, it's contents is determined by the running state of the kernel, it is not stored on disk.
  - /sys provides information about devices, buses, drivers. Also provides an API, most of the files are writable in this directory. Changing files in this directory affects how the kernel or certain kernel modules behave or operate.
  - /dev nowadays isn't stored on disk. It is a tmpfs managed by the udev daemon. It contains the device files for hardware items that are present on the system.
  - /lib contains shared libraries (.so files) and kernel modules (.ko files), along with some accounting information related to libraries.
  - /opt is where optional software is installed. Custom installers outside of the built-in package manager are encouraged to write into this directory instead of /bin or /sbin
  - /mnt is the default directory for mounting additional (e.g. removable or network) drives in several distros. RHEL uses /run/media for this purpose.

Navigating paths

- A file or directory is explicitly specified by giving its full name and path (location inside the directory structure).
- Filesystem object names are case sensitive in Linux.
- If the location specifier starts with a / (slash), it is an absolute path. E.g. /home/jane/Documents/cv.pdf
- Otherwise its a relative path, which is meant to be interpreted relative to the current working directory. E.g. Pictures/photo1.jpg if current directory is /home/jane → this equals to the absolute path of /home/jane/Pictures/photo1.jpg
- There is shortcut (substitution) for the currently logged in users home directory: ~ (tilde) For e.g. ~/Documents
- The shortcut to a specific users home directory looks like this: ~adam/some.file

Navigating paths

- The current working directory can be printed with the **pwd** command.
- The current working directory can be changed with the **cd** command.
- The current working directory is referenced with a . (single period), the parent directory with .. (double period)
- A directory can be listed with the **ls** command. Also take note of the -l -a -R -Z -h switches.
- Refer to **RH124 CH2.2.** for further details.

- You can use the **file** command to get information about a file's type, content and format. (Very useful!)

Managing files and directories

create and remove

- You can create empty files with the **touch** command and empty directories with the **mkdir** command. (note -p switch)
- You can remove a file using the **rm** command, to remove a directory with everything inside use **rm -r** command. (note the -f switch)
- The -r switch usually means recursive. Everything in a directory and its sub-directories and their sub-directories, etc… is considered to be the subject of the operation if this is used. E.g. ls -r; cp -r; rm -r
- The -f switch means force. Don’t ask, just do the operation. Use careful with rm!

Managing files and directories

copy and move

- You can copy files or directories with the **cp** command.
- You can rename/move files or directories with **mv**.
- Consider the source and destination types and number of arguments described in **RH124 CH2.3.**
- You can expect different behavior depending whether object2 is an existing file, an existing directory or a non-existing object.

Shell command line

```
# mv object1 object2
# cp object1 object2
# cp -r dir1 object2
# mv object1 object2 object3 dir1
```

Practice

- Complete lab practice in **RH124 CH 2.4.**
- Now do the same the smart way!

Shell command line

```
# mkdir Music Videos Pictures
# touch {Music/song{1..6}.mp3,Pictures/snap{1.
.6}.jpg,Videos/film{1..6}.avi}
# mkdir friends family work
# cp **/*[12].{mp3,jpg,avi} friends
# cp **/*[34].{mp3,jpg,avi} family
# cp **/*[56].{mp3,jpg,avi} work
```

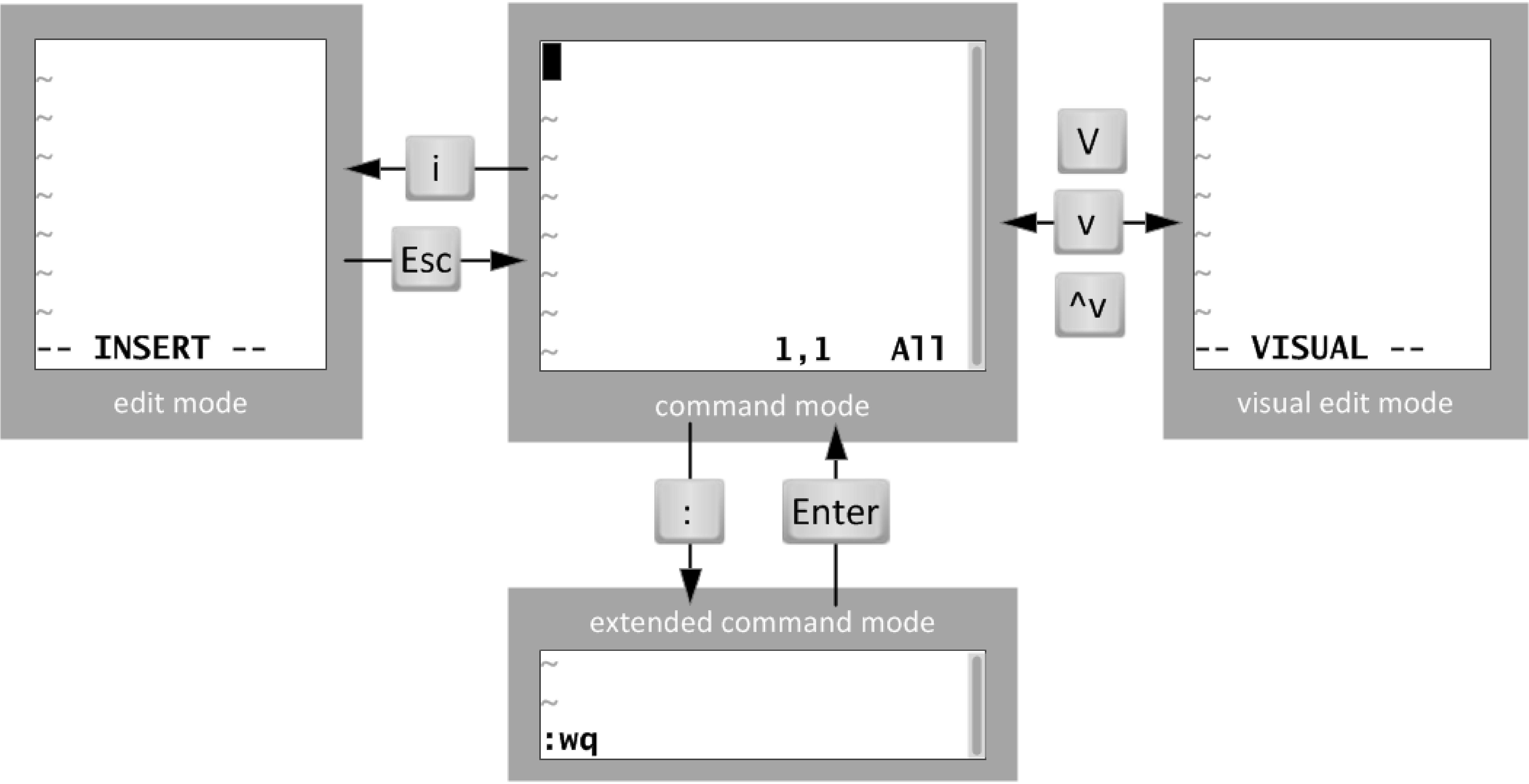
- More on filename globbing in **RH124 CH 2.5.** and lab practice **RH124 CH 2.6.**

Viewing files

- The most simple way to show a file’s contents is to use the **cat** command. This prints a file to the standard output.
- This command is usually applied when using redirects. (More on redirect later in this lecture)
- When reading on screen you can use the **more** or **less** commands to produce a scroll-able output.
- Remember that less is more :)
- Less produces a sideways and backwards scrollable output. You can use the arrow keys.
- More can only step forward with the output: one line (enter) or one page at a time (space).
- Quitting is possible with the q key.

Editing files

- The primary text editor present in all Unix systems is vi.
- vi has four operation modes: the command mode the edit mode, the visual edit mode, and the extended command mode.
- In command mode (this is the default) key presses are interpreted as instructions for the editor program. Extended commands start by typing a : (colon).
- In editor mode key presses are directed into file contents (actual typing).
- You can always return to command mode with Esc key.
- Visual mode is started with v or V or ctrl+v and terminates with the same key-press. You can do text selection in visual mode.


-- INSERT --
edit mode
i
Esc
1,1 All
command mode
V
v
^v
-- VISUAL --
visual edit mode
:
Enter
extended command mode
:wq

Most important vi commands

- In command mode:
  - press i for insert mode.
  - press a for append mode
  - type dd to delete (cut) whole line
  - type yy to copy whole line
  - type p for paste after cursor, P for pasting before cursor
- Advanced commands:
  - :w to save file
  - :q to quit
  - :q! to quit without saving file
  - :wq to save and quit

Selecting ranges in vim

- Position the cursor over the starting position.
- Activate char, line or block selection with v, V or ctrl+v
- Position the cursor to the ending position.
- Now you can cut with d, copy with y, and paste with p or P.
- You can insert with shift+I, append with shift+A. Input is applied to every selected line.
- Return to command mode by pressing the same key you entered with.
- More on vi in **RH124 CH 4.2-4.3. and RH134 CH3**.
- Complete lab practice **RH124 CH 4.6.**

Finding files

- An easy to use and fast utility to search for files is **locate**. It is speedy because it doesn't search the filesystem itself, rather it looks in a database.
- Mentioned database has to be updated regularly, using the **updatedb** command, to maintain consistency.

Shell command line

```
# locate *dir2*
/home/user/Download/glibc-2.21/dirent/tst-fopendir2.c
/home/user/Download/glibc-2.21/manual/exapmles/dir2.c
```

- The universal primordial standard tool is **find**.
- Find can be used with several filtering options, not just filename. Filter options include for e.g.: file type (refer to lecture 1, slide 32.), date, permissions, owner, size, etc.

Finding files

- More on find in **RH124 CH4.1.** Also consider the following examples:
- To find all socket type files under /var directory with it's name containing sql and not owned by adam, type the following.

Shell command line

```
# find /var -type s -name "*sql*" ! -user adam
```

- To find and delete all large and old files owned by bob in external drive (with size larger than 100MB and accessed more then a month ago) type the following.

Shell command line

```
# find /mnt/external -type f -user bob -size
+100M -atime +31 -exec rm {} ;
```

Introduction to BASH shell

Lecture 2

Chapter 2

Shell

- As discussed earlier in lecture 1 a shell provides an execution environment with variables, starts other processes, interprets user commands, provides advanced command editing features (like history and TAB completion) and much more.
- A shell can also be pre-programmed to execute a series of commands based on decisions. This is called shell scripting.
- Note the difference between interpreted (scripted) and compiled binary programs (ELF vs. text). There is a wide range of scripting languages available in *nix environments. Note that text files can also be executed in Linux, not just binaries. (specify interpreter with #!)
- There is plethora of shells for Linux. Further on in this course only BASH (Bourne-again shell) will be discussed, which is the most popular one to date.

Shell basics

- One of the most handy features is the TAB completion. It can complete filenames, commands and with some tweaks even command switches and arguments in order to reduce the time needed for typing.
- Pressing TAB once completes an already typed fragment of a filename or command to the point where it is not ambiguous.
- Pressing TAB twice quickly will display all the possible endings for the currently typed chunk.
- In case output would not fit the screen it is automati-cally piped through less.
- You can roll back and edit the previously entered commands with the arrow keys. This is called command history. A search in history can be initiated with ctrl+r.

Shell variables

- Bash handles only text variables. Numbers are also stored as a string of characters. Variables don’t have to be declared (exceptions!), they can be used simply by giving them a value. E.g. CARBRAND=”Toyota”
- Later on you can refer to the value of the variable with a $ sign. E.g. $CARBRAND
- Variables are evaluated through a mechanism called substitution. The shell, when parsing any input will look for $ signs and know it has to do something before executing the command. Basically it will replace one string with another, hence called substitution. Works like the macro pre-processor in a C compiler.

Shell command line

```
# echo "I drive a $CARBRAND”
I drive a Toyota
```

Shell variables

- The * character discussed in file globbing is actually also a substitution. Same as ~ for home directories. The * (star) character is substituted with a list of all files, than the command is executed with a long list of arguments. E.g. mv /home/david/old/* /mnt/archive
- You can specify multi word strings or strings with special characters by enclosing them in " (double quotes) E.g. FANCY_VAR="There are many *s on the sky". This makes bash disregard any substitution other then $VARIABLE likes.
- You can make bash disregard even variable substitution when enclosing something in ' (single quotes). E.g. echo 'I have got 10 $ in my pocket.'
- You can escape single characters with \ (backslash). E.g. "I've got 10 \$ in my pocket"

Shell variables

- Substitution works inside any string (except when using single quotes). E.g. COUNT=3 VAR="There are $COUNT *s on the sky"; echo $VAR
- The scope (lifespan) of variables can be one single line, one whole script, one user session or inheritable to all sub processes (exporting). Practice with the following:

Shell command line

```
# NAME=ALICE echo "Hello $NAME"
Hello
# NAME=ALICE bash -c echo "Hello $NAME"
Hello Alice
# NAME=Alice
# echo "Hello $NAME"
Hello Alice
# export NAME [then check in subshell]
```

Shell variables

- Substitution even works for executing whole commands and taking their output. This is done with a subshell. E.g. DAYOFWEEK=$(date '+%A'); echo "It's $DAYOFWEEK today". Enclosing a command in ` (backtick) is equivalent to $()
- You can print all the variables with the **env** command. You can use the **unset** command to delete a variable.
- Besides string variables and indexed arrays Bash 4 also has associative arrays. Refer to **man declare** for more.
- There are some special environment variables directly affecting the shell. E.g.: $TERM, $PS1, $PATH, $LANG

Shell command line

```
# export PS1="\u@\h \w> "
adam@lab12 /etc/mail>
```

Interactive shells

- There are two types of shells. The interactive shell is what the user is interfacing with, when typing commands. This is also called a login shell. This is what we have been using so far.
- The non-interactive or non-login shell is also a userspace process, it is started when executing a shell script. It reads the script, interpreting it line by line and performs accordingly. It doesn’t interact with a user directly, it doesn’t have a prompt.
- When a bash process is starting up it automatically executes the /etc/bashrc and ~/.bashrc scripts.
- A login shell in addition to the former files also executes /etc/profile and ~/.bash_profile scripts.
- This is where the environment variables such as $PATH are initialized to a sane value.

Shell scripting basics

- Use your newly acquired text editor skills to create the following file:

/home/student/script1.sh

```
#!/bin/bash
SAVE_IFS=$IFS
IFS=" "
echo "I got $# arguments. I'll print them back one
at a time, until I run out or reach \"stop\"";echo
for argument in $*;do
   echo $argument
   if [ $argument = "stop" ]; then
      echo "I reached stop"; break
   fi
done
IFS=$SAVE_IFS
echo; echo "Goodbye"
```

- Now make it executable with **chmod a+x script1.sh**

Shell scripting basics

- Some special variables:

  $IFS – internal field separator for processing lists

  $? – the return value of the last command (numeric)

  $# – the number of arguments

  $0 – the zero-th argument (what was the script called?)

  $1 $2 $3 … $9 – the first, second,… argument

  $* – all the arguments except $0

- Bash scripts can have functions. Function can also have arguments.

```
get_list_backups() {
    evaluate $1…
}
```

Shell scripting basics

test condition

- if test "$V1" = "$V2"
- if ["$V1" = "$V2"]
  Both are equal, test and [are userspace programs returning either zero or non-zero depending on expression that follows. Note that spaces are required.

- Some test examples:
  - if [-f path_to_file] – test if parameter is a file
  - if [-d path_to_file] – test if parameter is a dir
  - if [-e path_to_file] – test if parameter is existing
  - if [-z $VAR] – test if VAR is defined and not empty
  - if [$VAR1 -eq $VAR2] – test if numbers are equal
  - if [$VAR1 = $VAR2] – test if strings are the same

Shell scripting basics

test condition

- Also consider other test like -z -n -gt -ne -le -ge ... There are many.
- Conditions can be combined with logical operators

```
if [ -f "$file"] || [ -d "$file" ]
if [ "$file" = "aa" ] && [ -d "$file" ]
```

- Consider a substitution if $file variable is not set, there will be an invalid set of arguments.

```
if [ "$file" = "/home/some/file" ]
```

- Use this instead:

```
if [ X"$file" = "X/home/some/file" ]
```

Shell scripting basics

If construct and while construct

```
if [ condition ]
then
        operation1
else
        operation2
fi

while [ condition ]
do
        commands
done
```

Shell scripting basics

If construct and while construct

```
if [ condition ]; then
        operation1
else
        operation2
fi


while [ condition ]; do
        commands
done
```

Shell scripting basics

Case constuct

```
case $VAR in
-f)
        commands for option -f
        ;;
-d)
        commands for option -d
        ;;
*)
        echo “Unknown parameter” >&2
        exit 1
esac
```

Shell scripting basics

For constuct

```
for element in list; do
        do something with $element
done
```

- The commands do run for each element in the list or array. E.g.:

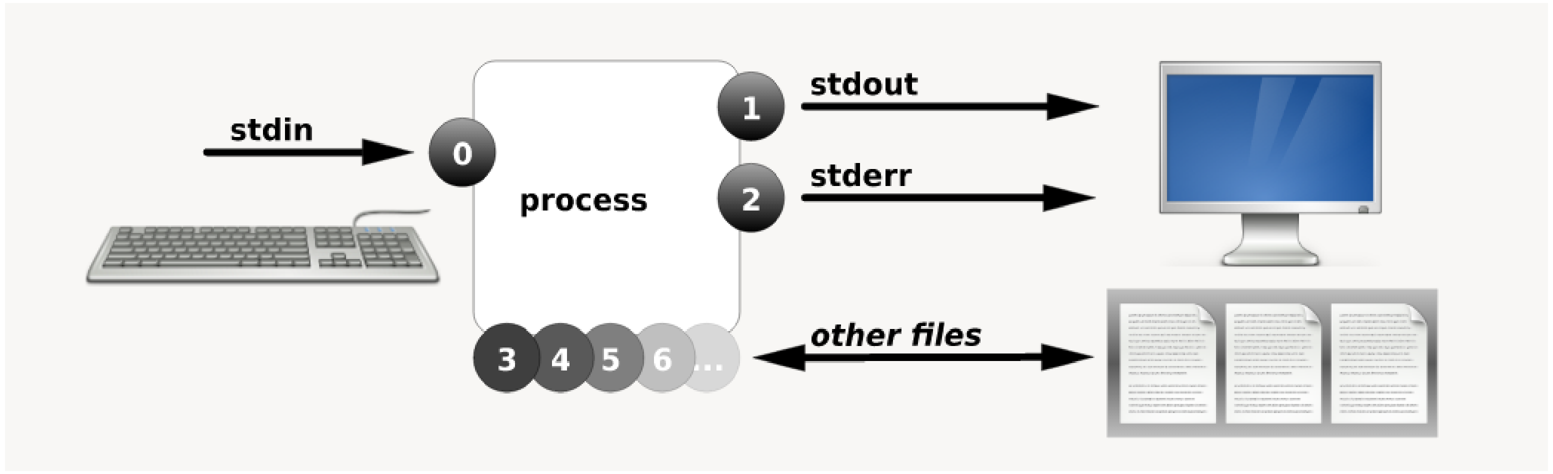
```
for i in $(ls *.xml); do
        echo $i
        mv $i $i.old
done
```

- To have C style counting for cycle use this example

```
for i in $(seq 8); do                    for i in $(seq 10 -2 4); do
    i (1 2 3 4 5 6 7 8)                      i (10 8 6 4)
```

Redirecting standard streams

- Remember lecture 1 slide 28-29 about file descriptors.



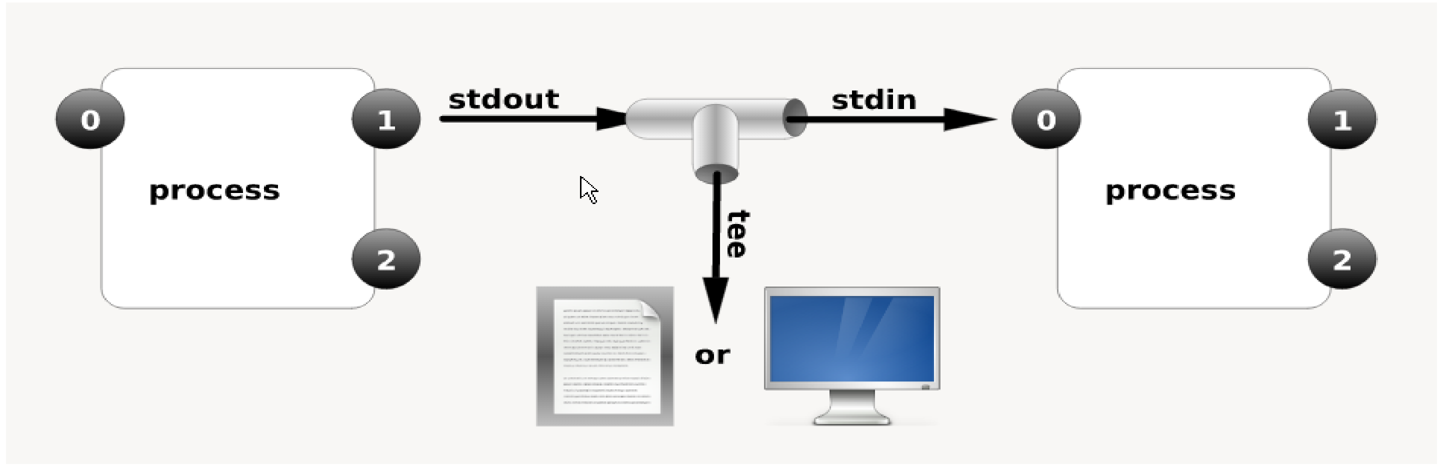
- When starting a process from shell, you can instruct it to override the default standard streams with a disk file, a named pipe or an anonymous pipe.
- This is useful for separating error messages to a log file, or automating input to a process in a script, but the most powerful benefit is the use of pipes. Pipes channel the output of a process to the input of another process. This is fast because no disk I/O or temp files are used.

Redirecting standard streams

- Examples on how to use redirects. Refer to **RH124 CH4.1.** for more.
  - some_program >some_file – redirect the stdout stream
  - some_program 2>some_file – redirect the std error stream
  - some_program 1>some_file – redirect the std out stream
  - some_program >file 2>&1 – redirect stderr to stdout and stdout (both streams) to file. Order is important!
  - >>file means append to file instead of overwriting from start
  - date -s < ~/date.sav – redirect standard input (sets the date to the string specified in date.sav file instead of requesting to type it from keyboard.)
  - cat <<EOF > ~/some_srcipt.sh – read multi-line text from input. Input can be terminated by typing EOF. Then save the output in some_sript.sh. This is called a 'here document'. (shell as text editor)

Pipes

- To connect standard streams of processes the | (pipe) symbol has to be used. This creates a temporary pipeline, which is discarded after the emitting process ends. This is called an anonymous pipe.
- You can create a named pipe as a filesystem object with **mkfifo** command. This is persistent, can be deleted with the **rm** command. A pipe can only be opened for write by exactly one process and can only be read by one single process. Once open it is unidirectional.



Pipes

- Processes can be chained using pipes infinitely.
- A pipe can be forked with the **tee** command, which works like a T junction. It forward its stdin to its stdout but also forwards a copy of the stream to the filename given as argument.
- The Linux concept of having a simple, specific, efficient utility (building block) for every task, combined with pipes is very powerful. Some examples:
  - `ls -t | head -n 10 | tee ~/ten-last-changed-files | mail student@desktop1.example.com`
  - `cut -d: -f7 /etc/passwd | uniq | sort`
  - `dd if=/dev/sda6 | pbzip2 -c -9 | nc hostB -p 2223`

    `netcat -p 2223 -l | pbzip2 -d | dd of=/dev/sda6`

Common utilities

- If you are not yet familiar with these utilities, please look up their man page, as they are quite useful with pipes.
  - less
  - head
  - tail
  - cut
  - tr
  - uniq
  - sort
  - grep
  - sed

Regular expressions

- Regular expressions are used to specify patterns that can be programmatically compared/matched against text.
- A utility called **grep** can be used to search files or stdin for pattern matches. Matched lines (or parts) are forwarded to stdout by grep.

Shell command line

```
# cat /etc/passwd | grep "John Smith"
jsmith:*:1003:1003:John Smith:/home/jsmith:/bin/sh
```

- Well known command line switches to grep are:
  - -v (invert match)
  - -i (case insensitive mode)
  - -c (count, print only number of matches, not the matched lines themselves)

Regular expressions

Common pattern match rules and symbols:

- . (dot) - a single character.
- ? - the preceding character matches 0 or 1 times only.
- * - the preceding character matches 0 or more times.
- + - the preceding character matches 1 or more times.
- {n} - the preceding character matches exactly n times.
- {n,m} - the preceding character matches at least n times and not more than m times.
- [agd] - the character is one of those included within the square brackets.
- [^agd] - the character is not one of those included within the square brackets.

Regular expressions

Common pattern match rules and symbols (continued):

- [c-f] - the dash within the square brackets operates as a range. In this case it means either the letters c, d, e or f.
- () - allows us to group several characters to behave as one.
- | (pipe symbol) - the logical OR operation.
- ^ - matches the beginning of the line.
- $ - matches the end of the line.

Refer to **RH134 CH2** on how to use grep and regular expressions.

Midnight Commander cheat sheet

Lecture 2

Chapter 3

Useful hotkeys

- F3 – view file
- F4 – start editor
- shift F4 – create new file in editor
- F5 – copy file
- F6 – move file
- shift + F6 – rename file
- F7 – make directory
- F8 – delete
- F9 – pulldown menu
- F10 – quit
- ctrl + o – Hide/show panels

Useful hotkeys

- ctrl + s – Jump to file
- insert – select item
- + – select items (filter)
- / – deselect item (filter)
- alt + a – insert pwd to command line
- alt + enter – insert selected files name to command line
- ctrl+x, t – insert selected names to command line.
- alt + shift + ? – find file panel
- ctrl + space – calculate directory size
- alt + h – command history
- ctrl+x, c – chmod for item
- ctrl+x, o – chown for item

Useful hotkeys in mcedit

- F2 – save file
- shift + F2 – save file as
- F3 – submit start of selection/ end of selection
- F4 – replace pattern match panel
- F5 – copy selection to current cursor position
- F6 – move selection to current cursor position
- F7 – search pattern panel
- F8 – delete selection or current line
- F9 – pulldown menu
- F10 – quit
- ctrl + u – undo

Recommended reading

Lecture 2

Chapter 4

Relevant chapters in RH:

- RH124 Chapter 2 (filesystem layout, file manipulation)
- RH124 Chapter 4 (redirect, cat, vim)
- RH134 Chapter 2 (regexp, grep)
- RH134 Chapter 3 (vim)

Read by next lecture:

- RH 124 CH5 (local users and groups)
- RH 124 CH6 (file access pemissions)
- RH 134 CH6 (ACLs)

Use your rhlearn.gilmore.ca login to access the learning materials.

Thank you for your attention!

Lecture 2, PM-TRTNB319, Linux System Administration

Zsolt Schäffer, PTE-MIK