Redirecting Output and Using Pipelines in the Linux Shell

Learning Goals

- Capture command output and errors into files
- Route command output through other commands using pipes

Understanding Standard I/O Streams

When a program runs in a Linux shell, it interacts with three fundamental input/output streams:

- Standard Input (stdin) – usually the keyboard (file descriptor 0)
- Standard Output (stdout) – typically the terminal screen (file descriptor 1)
- Standard Error (stderr) – also the terminal, but for error messages (file descriptor 2)

Each of these is managed using a system of file descriptors, which are numeric identifiers that track where data comes from or goes. Here's a summary:

Descriptor	Name	Default Source/Destination	Purpose
0	stdin	Keyboard	Input
1	stdout	Terminal	Normal Output
2	stderr	Terminal	Error Output
3+	Custom	User-specified files/devices	Input/Output

Changing I/O Behavior with Redirection

Redirection allows you to change where input comes from or where output goes. Instead of printing everything to the screen, you can:

- Save it to a file
- Suppress it entirely
- Chain it into another command

Redirection works as follows:

Common Redirection Operators

Operator	Function
`> file`	Redirect stdout to overwrite a file
`>> file`	Redirect stdout to append to a file
`2> file`	Redirect stderr to overwrite a file
`2>/dev/null`	Discard stderr completely
`> file 2>&1` or `&> file`	Redirect both stdout and stderr to the same file (overwrite)
`>> file 2>&1` or `&>> file`	Redirect both stdout and stderr to the same file (append)

⚠Redirection order matters. For example:

```
> output.log 2>&1   # Both stdout and stderr go to output.log
2>&1 > output.log   # Only stdout goes to the file; stderr still appears in terminal
```

Because of this, many prefer using &> and &>> in Bash 4+ for clarity. However, these are not portable across all shell types (e.g., sh, dash), so for maximum compatibility, stick to > file 2>&1.

Practical Redirection Examples

Store the current date and time in a file:

```
date > /tmp/saved-timestamp
```

Save the last 100 lines of a secure log:

```
tail -n 100 /var/log/secure > /tmp/last-100-log-secure
```

Combine multiple files into one:

```
cat step1.sh step2.log step3 step4 > /tmp/all-four-steps-in-one
```

List all files (including hidden ones) in your home directory:

```
ls -a > my-file-names
```

Append a new line to an existing file:

```
echo "new line of information" >> /tmp/many-lines-of-information
```

Redirect only error messages to a file:

```
find /etc -name passwd 2> /tmp/errors
```

Redirect output and errors to separate files:

```
find /etc -name passwd > /tmp/output 2> /tmp/errors
```

Redirect output to a file, discard errors:

```
find /etc -name passwd > /tmp/output 2> /dev/null
```

Send both output and errors to the same file:
find /etc -name passwd &> /tmp/all-message-output

Append both output and errors to an existing file:
find /etc -name passwd >> /tmp/all-message-output 2>&1

Using Pipes to Chain Commands

A pipe (|) connects the output of one command directly into the input of another. Think of it as a conveyor belt for data processing.

Examples of Pipelining:

Display long listings page-by-page:
ls -l /usr/bin | less

Count the number of files in a directory:
ls | wc -l

List the 10 most recently changed files and save to a file:
ls -t | head -n 10 > /tmp/first-ten-changed-files

Combining Redirection with Pipes

Normally, when mixing redirection and pipes, the shell evaluates the whole pipeline first, then applies redirection. This can cause unexpected behavior if you're not careful.

Bad Example – Output goes to a file, and less gets nothing:
ls > /tmp/saved-output | less

Solution: Use tee

The tee command allows output to be sent to a file and passed through the pipeline at the same time. It's like a "T-junction" in a plumbing system.
ls -l | tee /tmp/saved-output | less

Append to file while still viewing output:
ls -t | head -n 10 | tee -a /tmp/append-output

Redirecting Both Output and Errors Through a Pipe

To include stderr in a pipeline, redirect it first:
`find / -name passwd 2>&1 | less`

Note: Do not use &> or &>> in this context—they won't pipe stderr.

References

- info bash (GNU Bash Reference Manual)
- info coreutils 'tee invocation'
- man pages: bash, cat, head, less, tee, wc, tty

Working with Text Files from the Command Line

Objective

Learn how to create and edit text files directly from the terminal using the powerful vim editor.

Why Text Files Matter in Linux

Linux stores most system and application configurations in plain text files. These files can vary in format—key-value pairs, INI styles, YAML, XML, or custom structures. The benefit? They can be opened and modified using any basic text editor, making Linux systems incredibly transparent and flexible to manage.

Meet Vim: Your Go-To Terminal Editor

Vim (Vi Improved) is an enhanced version of the classic vi editor that's bundled with virtually every Unix and Linux system. It adds modern conveniences such as syntax highlighting, split windows, color themes, and efficient navigation tools—all within a text-only interface.

Installing and Launching Vim

On Red Hat-based systems, Vim can be installed and used in two ways:

- **Minimal version:**
  Lightweight, includes the vi command only.
  ```
  # dnf install vim-minimal
  $ vi filename
  ```

- **Enhanced version:**
  Full-featured editor with built-in help and training tools.
  ```
  # dnf install vim-enhanced
  $ vim filename
  ```

When the vim-enhanced package is present, regular users invoking vi might be redirected to vim via a shell alias—except for root and system users (UIDs below 200).

To verify which version you're using:

```
vi --version
vim --version
```

Modes of Operation in Vim

Vim operates through different modes, each serving a specific purpose:

Mode	Access Key	Description
Normal Mode	Default on launch	Navigate, delete, and manipulate text.
Insert Mode	`i`	Enter text; press `Esc` to return to Normal mode.
Visual Mode	`v` / `V` / `Ctrl+v`	Select text by character, line, or block.
Command Mode	`:`	Run commands like save (`:w`) or quit (`:q`).

Not sure what mode you're in? Press Esc a few times to safely return to Normal mode.

Essential Vim Commands for Beginners

Here's a shortlist of commands to get started confidently:

- `u` – Undo last change
- `x` – Delete a single character
- `:w` – Save (write) the file
- `:wq` – Save and exit
- `:q!` – Exit without saving

These basics cover the most frequent actions you'll perform when editing text on a Linux system.

Copying and Moving Text in Vim

To rearrange text:

1. Move the cursor to the start of the text block
2. Enter Visual Mode:
    - Character-wise: v
    - Line-wise: Shift+v
    - Block-wise: Ctrl+v
3. Highlight the desired content
4. Press y to yank (copy)
5. Navigate to the destination
6. Press p to paste

Visual Block mode is especially useful for column editing in configuration or data files.

Customizing Vim with Configuration Files

Vim's behavior can be tailored using config files:

- System-wide: /etc/vimrc
- User-specific: ~/.vimrc

For example, here's a .vimrc that improves editing YAML files:

```
autocmd FileType yaml setlocal ts=2
set number
```

This sets the tab width to 2 spaces in YAML files and enables line numbering globally.

Learn Vim Interactively

Run this built-in tutorial to practice Vim safely:

```
vimtutor
```

It comes with the vim-enhanced package and walks you through real usage scenarios interactively.

References

- man vim
- Inside Vim: :help
- Vim Online Manual:https://vimhelp.org/options.txt.html#options.txt

Customizing the Shell Environment

Objective

Learn how to define shell variables, configure Bash startup scripts, and control how your shell and its spawned processes behave.

What Are Shell Variables?

In the Bash shell, variables are temporary placeholders that help simplify commands, store data, or tweak the behavior of your shell environment. You can define a variable in a session, and optionally export it, making it accessible to any subprocess or command you launch from that shell.

Each open terminal window or SSH session runs its own isolated shell environment, meaning variables you set in one are not visible in another.

Setting Shell Variables

To define a shell variable, use the format:

```
VARIABLENAME=value
```

Valid variable names may include letters, numbers, and underscores, but must not begin with a number.

Examples:

```
COUNT=40
first_name=John
file1=/tmp/abc
_ID=RH123
```

These are temporary and only live within the session where they're created.

To view a list of current shell variables and functions:

```
$ set | less
```

Accessing Variable Values: Expansion

To reference a variable, prefix its name with a dollar sign:

```
echo $COUNT
```

This prints the value assigned to COUNT.

If you forget the $, you'll just see the variable name as-is:

```
echo COUNT    # Outputs: COUNT
```

Use curly braces when appending characters to a variable to avoid ambiguity:

```
echo ${COUNT}x    # Outputs: 40x
```

Persistent Configuration: Bash Shell Variables

Bash supports several built-in variables that affect its behavior. Examples include:

- HISTFILE – Sets the file that stores your command history (~/.bash_history by default)
- HISTFILESIZE – Controls how many commands are saved in the history file
- HISTTIMEFORMAT – Defines the timestamp format for each history entry

```
HISTTIMEFORMAT="%F %T "
history
```

You'll now see timestamps next to each history entry.

Another powerful variable is PS1, which controls the command prompt's appearance:

```
PS1="[\u@\h \W]\$ "
```

This sets the prompt to display your username, hostname, and working directory. Always wrap prompt values in quotes and end with a space for clarity and consistency.

Environment Variables vs Shell Variables

Shell variables are used by Bash internally. Environment variables, on the other hand, are inherited by any commands or programs launched from that shell.

To promote a shell variable into the environment:

```
EDITOR=vim
export EDITOR
```

Or combine both steps:

```
export EDITOR=vim
```

Programs can then read these values to configure themselves.

Common Environment Variables

- HOME: Your home directory
- LANG: Determines the locale (language, number formatting, etc.)
  ```
  export LANG=fr_FR.UTF-8
  date   # Now outputs in French
  ```

- PATH: A colon-separated list of directories that the shell searches for executable commands.
  ```
  echo $PATH
  export PATH=${PATH}:/home/user/sbin
  ```

To view all environment variables:
```
$ env
```

Setting the Default Editor

To specify which editor should be used by default:
```
export EDITOR=nano
```

This tells programs like crontab or git to use nano unless told otherwise.

Case Sensitivity and Naming Conventions

By tradition:

- Use ALL UPPERCASE for system/environment variables.
- Use lowercase or mixed case for personal/custom shell variables to avoid conflicts.

Automating Variable Setup with Bash Startup Scripts

When Bash launches, it runs initialization scripts depending on the context:

Shell Type	Configuration Files
Login shell	`/etc/profile`, `~/.bash_profile`
Interactive non-login	`/etc/bashrc`, `~/.bashrc`
Non-interactive (scripts)	File specified by `$BASH_ENV` (if defined)

To make your variables persist in every session:

- Put them in ~/.bashrc for interactive shells.
- Use ~/.bash_profile for login-specific setups (e.g., remote SSH).

Example ~/.bash_profile snippet:

```
# ~/.bash_profile
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi
export EDITOR=nano
```

For system-wide changes, create a ".sh" file inside /etc/profile.d/ (as root) with your export statements.

Creating Bash Aliases

Aliases simplify frequent or complex commands:

```
alias hello='echo "Hello, this is a long string."'
```

Now calling hello will produce:

```
Hello, this is a long string.
```

Add aliases to ~/.bashrc to make them permanent for your user.

Removing Variables and Aliases

To remove a shell variable:

```
unset file1
```

To stop exporting a variable:

```
export -n PS1
```

To delete an alias:

```
unalias hello
```

References

- man bash – Official Bash manual
- man env – For environment variables
- man builtins – Lists built-in shell commands
- :help in Vim – For text editor customization

Handling Compressed Archives with tar

Goal

Learn how to bundle files and directories into compressed .tar archives and extract their contents using the tar command.

What Is an Archive?

An archive is a single file that packages multiple files and directories. This can be a standard file or a device (like a tape or USB drive). Archiving is often used to simplify backups, transfers, or grouping related files.

On Linux systems, the tar command is the standard utility for creating and extracting such archives. While other tools like zip exist (using the legacy PKZIP algorithm), tar is more flexible and supports multiple compression methods.

Archiving can be done with or without compression. Compression reduces file size and is helpful when saving space or transferring data over a network.

The tar Command: Core Functions

The tar tool supports a range of actions and options. Here are the most essential:

Primary Actions:

- -c or --create → Create a new archive
- -t or --list → View the archive contents
- -x or --extract → Extract files from an archive

Common Options:

- -f → Specify the archive file
- -v → Show progress while processing files
- -p → Preserve file permissions when extracting
- --xattrs → Store extended attributes
- --selinux → Include SELinux contexts

Compression Methods:

- -a → Auto-detect compression from file extension
- -z → Use gzip (.tar.gz)
- -j → Use bzip2 (.tar.bz2)
- -J → Use xz (.tar.xz)

⚠ Legacy tar syntax (without dashes, e.g. tar cvf) is still supported but discouraged in modern scripts.

Creating a Basic Archive

To package files into an archive, use tar -cf. Example:

```
$ tar -cf mybackup.tar myapp1.log myapp2.log myapp3.log
```

This creates mybackup.tar containing the specified log files.

By default, tar strips leading / from absolute paths. This ensures safer extraction because paths are treated as relative, preventing accidental overwrites of system files.

Root Example:

```
# tar -cf /root/etc-backup.tar /etc
```

You'll see a message like Removing leading '/' from member names.

- Only readable files are included in the archive. If you're not root, files with restricted permissions will be skipped.

Viewing Archive Contents

To preview what's inside an archive without extracting:

```
$ tar -tf archive-name.tar
```

Example:

```
$ tar -tf /root/etc-backup.tar
```

Extracting an Archive

It's best to extract into an empty folder to avoid overwriting.
Example:

```
mkdir /root/etcbackup
cd /root/etcbackup
tar -xf /root/etc-backup.tar
```

To preserve original file permissions:

```
$ tar -xpf /path/to/archive.tar
```

Superusers (root) automatically retain original ownership; regular users become the new owners of extracted files.

Creating Compressed Archives

The tar command supports several compression formats. Use one of the flags below when creating your archive:

Flag	Compression Type	File Extension
`-z`	gzip	`.tar.gz`
`-j`	bzip2	`.tar.bz2`
`-J`	xz	`.tar.xz`

Examples:

Gzip Compression:

```
tar -czf /root/etcbackup.tar.gz /etc
```

Bzip2 Compression:

```
tar -cjf /root/logbackup.tar.bz2 /var/log
```

XZ Compression:

```
tar -cJf /root/sshconfig.tar.xz /etc/ssh
```

You can list the contents of a compressed archive with just tar -tf, and tar auto-detects the compression method:

```
tar -tf /root/etcbackup.tar.gz
```

Extracting Compressed Archives

You don't need to specify the compression type when extracting—tar figures it out automatically:

```
tar -xf archive.tar.gz
```

If the wrong compression flag is used (e.g. -z with an .xz file), tar will throw an error:

```
tar -xzf file.tar.xz
# gzip: stdin: not in gzip format
```

Verifying Archive Size Before Extraction

Want to check how much space an archive will occupy after extraction? Use:

```
gzip -l file.tar.gz
xz -l file.xz
```

Example Output:

```
gzip -l mybackup.tar.gz
compressed   uncompressed  ratio    name
221603125    303841280     27.1%    mybackup.tar

xz -l file.xz
Strms  Blocks  Compressed  Uncompressed  Ratio  Check  Filename
1      1       195.7 MiB   289.8 MiB     0.675  CRC64  file.xz
```

Using Compression Tools Standalone

If you want to compress a single file without creating an archive:

- gzip file
- bzip2 file
- xz file

And to decompress:

- gunzip file.gz
- bunzip2 file.bz2
- unxz file.xz

Note: These tools work on one file at a time and don't bundle multiple files like tar.

Summary

Task	Command Example
Create basic archive	`tar -cf archive.tar file1 file2`
Create gzip-compressed tar	`tar -czf archive.tar.gz dir/`
Create bzip2 archive	`tar -cjf archive.tar.bz2 dir/`
Create xz archive	`tar -cJf archive.tar.xz dir/`
List archive contents	`tar -tf archive.tar`
Extract archive	`tar -xf archive.tar`
Preserve permissions	`tar -xpf archive.tar`
Add extended attributes	`--xattrs --selinux --acls`
Check compressed size	`gzip -l file.gz` or `xz -l file.xz`

References

- man tar
- man gzip, gunzip
- man bzip2, bunzip2
- man xz, unxz

Secure File Transfers Between Systems

Objective

Learn how to transfer files safely between local and remote systems using SSH-based tools, with a focus on the sftp utility.

Secure File Transfers with SFTP

The OpenSSH suite includes several tools for secure remote interactions. One of them is the Secure File Transfer Protocol (SFTP), which provides an encrypted channel for transferring files between machines. It's interactive, user-friendly, and leverages the same authentication and encryption mechanisms as ssh.

You can start an SFTP session using this format:
sftp [user@]remotehost

If the username is omitted, the command defaults to the current local user.

Example:

```
sftp remoteuser@remotehost

remoteuser@remotehost's password: ******
Connected to remotehost.
sftp>
```

This opens an interactive SFTP prompt (sftp>) where you can issue a variety of commands to manage files on the remote system.

SFTP Interactive Commands

SFTP supports many familiar file management commands, including:

- ls, cd, mkdir, rmdir – navigate and manage remote directories
- put – upload a file
- get – download a file
- exit or bye – close the session
- help – list all available commands

To see the full command list:

```
sftp> help
```

Some SFTP commands can also operate on the local system by prefixing the command with l. For example:

```
sftp> pwd     # remote working directory
sftp> lpwd    # local working directory
```

Uploading Files

To upload a file from your local system to a directory on the remote host:

```
sftp> mkdir hostbackup
sftp> cd hostbackup
sftp> put /etc/hosts
```

Output:

```
Uploading /etc/hosts to /home/remoteuser/hostbackup/hosts
/etc/hosts                          100% 227     0.2KB/s 00:00
```

To upload an entire directory recursively:

```
sftp> put -r directory
```

Example Output:

```
Uploading directory/ to /home/remoteuser/directory
file1                                          100%
file2                                          100%
```

Downloading Files

To retrieve a file from the remote system:

```
sftp> get /etc/yum.conf
```

Output:

```
Fetching /etc/yum.conf to yum.conf
/etc/yum.conf                       100% 813     0.8KB/s   00:00
```

Exit the session:

```
sftp> exit
```

One-Line File Downloads

To download a file non-interactively in a single command:

```
sftp remoteuser@remotehost:/path/to/file
```

Example:

```
sftp remoteuser@remotehost:/home/remoteuser/remotefile
```

This connects, fetches the file, and exits—all in one step.
Note: Uploading files with put is not supported via one-line SFTP syntax.

A Note About scp

The scp (Secure Copy) command is a classic tool for transferring files over SSH, but it's based on the outdated rcp protocol. Security flaws—particularly CVE-2020-15778—mean that using scp can expose systems to command injection vulnerabilities.

Red Hat and security experts strongly recommend avoiding scp in favor of:

- sftp – for interactive or scripted file transfers
- rsync – for more advanced or automated transfers with synchronization

Although some patches were applied, scp still carries risks due to compatibility constraints and its legacy design.

Summary: Common SFTP Workflow

Task	Command or Syntax
Start SFTP session	`sftp user@host`
View remote working directory	`pwd`
View local working directory	`lpwd`
Upload a file	`put filename`
Upload a directory recursively	`put -r directory`
Download a file	`get remote/file/path`
Exit SFTP	`exit` or `bye`
Run local commands	Prefix with `l` (e.g., `lls`, `lcd`)

References

- man sftp – Full command manual
- Red Hat Security: CVE-2020-15778
- OpenSSH documentation: https://www.openssh.com/manual.html

Securely Sync Files Across Systems

Objective

Use rsync to efficiently and securely synchronize files or directories between a local system and a remote server.

Why Use rsync?

The rsync utility is a robust and flexible command-line tool for syncing files and directories between systems. Unlike traditional file copy tools, rsync only transfers the differences between source and destination—making it highly efficient, especially after the initial sync.

Key Advantages:

- Minimizes data transfer by copying only changed blocks
- Supports secure transfer over SSH
- Preserves file attributes such as permissions, timestamps, and ownership
- Supports local-to-local, local-to-remote, and remote-to-local synchronization

Common rsync Options

When working with rsync, the following options are frequently used:

Option	Description
`-a` or `--archive`	Enables archive mode; preserves symbolic links, permissions, timestamps, groups, owners, and devices
`-v` or `--verbose`	Displays detailed output during transfer
`-n` or `--dry-run`	Simulates the command without making changes (safe for testing)
`-H`	Preserves hard links (not included in archive mode by default)
`-A`	Includes Access Control Lists (ACLs)
`-X`	Includes SELinux security contexts

Archive Mode (-a) Includes:

- -r: Recursively sync directories
- -l: Maintain symbolic links
- -p: Preserve file permissions
- -t: Preserve modification times
- -g: Preserve group ownership
- -o: Preserve file owner
- -D: Preserve device files

Dry Run First (Safety First)

Before running a real synchronization, always do a test with the -n (dry run) option:

```
rsync -avn source/ destination/
```

This will show you exactly what would be transferred without making any changes.

Remote Synchronization Syntax

rsync uses the format [user@]host:/path to reference remote systems. Either the source or the destination must be local.

To sync a local directory to a remote system:

```
rsync -av /path/to/local/dir user@remotehost:/remote/dir
```

To sync from a remote system to your local machine:

```
rsync -av user@remotehost:/remote/dir /local/dir
```

If you need to preserve file ownership on the destination, ensure you're running rsync as root.

Examples

1. Sync Local to Remote

   ```
   rsync -av /var/log root@hosta:/tmp
   ```

   Output:

   ```
   receiving incremental file list
   log/
   log/boot.log
   log/README
   ...
   total size is 11,585,690 speedup is 38.57
   ```

2. Sync Remote to Local

   ```
   rsync -av root@hosta:/var/log /tmp
   ```

   This copies logs from hosta into /tmp on the local machine.

3. Local-to-Local Sync

   ```
   rsync -av /var/log /tmp
   ```

   Efficiently duplicates the /var/log directory into /tmp.

Directory Trailing Slash: A Crucial Detail

The presence or absence of a trailing slash on the source path changes how rsync behaves:

- With /var/log/: Syncs the contents of the directory directly into the destination.
- Without /var/log: Syncs the entire log directory as a subdirectory of the destination.

Example:
```
rsync -av /var/log/ /tmp
```

Results in:
```
/tmp/boot.log
/tmp/README
```

But:
```
rsync -av /var/log /tmp
```

Results in:
```
/tmp/log/boot.log
/tmp/log/README
```

Most shells auto-complete directories with a trailing slash—be mindful when using tab-completion.

Performance Output Sample

```
sent 11,592,423 bytes  received 779 bytes  23,186,404.00 bytes/sec
total size is 11,586,755  speedup is 1.00
```

Summary

Task	Command Example
Dry run simulation	`rsync -avn source/ dest/`
Sync local dir to remote	`rsync -av /data user@host:/backup`
Sync remote dir to local	`rsync -av user@host:/backup /data`
Preserve ACLs and SELinux contexts	`rsync -avAX /source /dest`
Preserve hard links	`rsync -avH /source /dest`
Sync local directories	`rsync -av /var/log/ /tmp/`

References

- man rsync – Official manual for advanced usage
- https://rsync.samba.org – rsync homepage and documentation