CSS 262: Linux Administration

Bash Scripting & Automation

Lecture 5: Automate All the Things

📋 Today's Agenda

Part 1: Scripting Fundamentals

- What is a shell script?
- Shebang, execution, and permissions
- Variables & data types
- User input & output

Part 2: Control Structures & Functions

- Conditionals (`if` , `case`)
- Loops (`for` , `while` , `until`)
- Functions & scope
- Arrays & string operations

Part 3: Text Processing & Pipelines

- `grep` , `sed` , `awk` , `cut`
- Regular expressions
- Pipelines & redirection

Part 4: Automation & Best Practices

- Cron jobs & scheduling
- Real-world scripts
- Error handling & debugging
- Security considerations

🎯 **Learning Objective:** Develop automated solutions for system maintenance and log processing using Bash scripting.

🔄 Quick Recap: Week 4

Storage, Filesystems & LVM

- **Storage Hierarchy**: Physical disks → Partitions → Filesystems
- **Partition Tables**: MBR (legacy) vs GPT (modern standard)
- **Filesystems**: ext4 (general), XFS (performance), Btrfs (snapshots)
- **Mounting**: Manual `mount` and persistent `/etc/fstab`
- **LVM**: Flexible storage with PV → VG → LV architecture
- **Snapshots**: Point-in-time copies for consistent backups

✅ You should now be comfortable managing storage, filesystems, and LVM volumes!

Part 1: Scripting Fundamentals

Your First Bash Scripts

🐚 What is a Shell Script?

A **shell script** is a text file containing a sequence of commands that the shell executes.

Why Script?

- **Repeatability**: Run the same tasks consistently
- **Automation**: Schedule tasks to run unattended
- **Efficiency**: Combine many commands into one action
- **Documentation**: Scripts document your procedures
- **Error Reduction**: Eliminate human mistakes

The Bash Shell

- **Bash** = Bourne Again Shell (default on most Linux distros)
- Superset of the original Bourne shell (`sh`)
- Available at `/bin/bash`
- Check your shell: `echo $SHELL`

📜 Anatomy of a Script

```
1   #!/bin/bash
2   # backup.sh - Simple backup script
3   # Author: sysadmin | Date: 2025-01-15
4
5   set -euo pipefail
6
7   BACKUP_DIR="/backup"
8   SOURCE_DIR="/var/www"
9   DATE=$(date +%Y%m%d_%H%M%S)
10  ARCHIVE="${BACKUP_DIR}/www_${DATE}.tar.gz"
11
12  mkdir -p "$BACKUP_DIR"
13  tar czf "$ARCHIVE" "$SOURCE_DIR"
14
15  echo "Backup complete: $ARCHIVE"
```

Line 1: Shebang (`#!/bin/bash`) – tells the kernel which interpreter to use

Line 5: `set -euo pipefail` – strict mode for safer scripts

Lines 7-9: Variables store configuration – easy to modify

▶️ Running Scripts

Method 1: Make it Executable

```
chmod +x backup.sh
./backup.sh
```

Method 2: Invoke the Interpreter

```
bash backup.sh
```

Method 3: Source (runs in current shell)

```
source backup.sh
# or
. backup.sh
```

⚠️ **Key Difference:** `./script.sh` runs in a subshell (isolated), while `source script.sh` runs in the current shell (can modify your environment).

📦 Variables

Assigning Variables (no spaces around `=`)

```
NAME="Linux"              # String
COUNT=42                  # Integer (still stored as string)
FILES=$(ls /tmp)          # Command substitution
TODAY=$(date +%F)         # Command substitution
```

Using Variables

```
echo "Welcome to $NAME"
echo "File count: ${COUNT}"       # Braces for clarity
echo "Config: ${NAME}_config"     # Braces to delimit name
```

Environment vs Local Variables

```
MY_VAR="local only"           # Local to current shell
export MY_VAR="shared"        # Available to child processes
env                           # Show all environment variables
```

🔢 Special Variables

Variable	Description
`$0`	Script name
`$1` - `$9`	Positional arguments
`$#`	Number of arguments
`$@`	All args (preserves quoting)
`$*`	All args (single string)
`$?`	Exit status of last command
`$$`	Current process PID
`$!`	PID of last background process

Example

```
#!/bin/bash
echo "Script: $0"
echo "First arg: $1"
echo "All args ($#): $@"
echo "Last exit code: $?"
```

💬 Input & Output

Reading User Input

```
read -p "Enter your name: " USERNAME
echo "Hello, $USERNAME!"
read -sp "Password: " PASS       # -s = silent (no echo)
echo                              # newline after hidden input
read -t 10 -p "Quick! " ANSWER   # -t = timeout (seconds)
```

Output Commands

```
echo "Simple output"
echo -e "Tabs:\tand\nnewlines"    # -e enables escape sequences

printf "%-20s %5d\n" "Users:" 42  # Formatted output
printf "%-20s %5d\n" "Groups:" 7
```

Redirecting Output

```
echo "log entry" >> /var/log/app.log    # Append
echo "fresh start" > /var/log/app.log   # Overwrite
command 2>/dev/null                      # Discard errors
```

🧮 Arithmetic

Arithmetic Expansion `$(( ))`

```
A=10
B=3
echo "Sum: $((A + B))"          # 13
echo "Diff: $((A - B))"         # 7
echo "Product: $((A * B))"      # 30
echo "Division: $((A / B))"     # 3 (integer only!)
echo "Modulo: $((A % B))"       # 1
echo "Power: $((A ** B))"       # 1000
```

Increment / Decrement

```
((COUNT++))                     # Increment
((COUNT--))                     # Decrement
((COUNT += 5))                  # Add 5
```

Floating Point (use `bc`)

```
RESULT=$(echo "scale=2; 10 / 3" | bc)
echo "$RESULT"                  # 3.33
```

Part 2: Control Structures & Functions

Making Decisions and Repeating Tasks

🔀 Conditionals: `if` / `elif` / `else`

Basic Syntax

```
if [[ condition ]]; then
    # commands
elif [[ condition ]]; then
    # commands
else
    # commands
fi
```

Example: Check Disk Usage

```
USAGE=$(df / --output=pcent | tail -1 | tr -d '% ')

if [[ "$USAGE" -gt 90 ]]; then
    echo "CRITICAL: Disk usage at ${USAGE}%!"
elif [[ "$USAGE" -gt 75 ]]; then
    echo "WARNING: Disk usage at ${USAGE}%"
else
    echo "OK: Disk usage at ${USAGE}%"
fi
```

🧪 Test Operators

Numeric Comparisons

Operator	Meaning
`-eq`	Equal
`-ne`	Not equal
`-gt`	Greater than
`-ge`	Greater or equal
`-lt`	Less than
`-le`	Less or equal

String Comparisons

Operator	Meaning
`==`	Equal
`!=`	Not equal
`-z`	Empty string
`-n`	Non-empty string
`<` / `>`	Lexicographic

Logical Operators

Operator	Meaning
`&&`	AND
`||`	OR
`!`	NOT

🧪 Test Operators: File Tests

File Tests

Operator	Meaning
`-e`	File exists
`-f`	Is regular file
`-d`	Is directory
`-r`	Is readable
`-w`	Is writable
`-x`	Is executable
`-s`	File is non-empty
`-L`	Is symbolic link

Example

```
if [[ -f "/etc/passwd" ]]; then
    echo "File exists"
fi

if [[ -d "/tmp" && -w "/tmp" ]]; then
    echo "Dir is writable"
fi
```

🔄 The `case` Statement

Syntax

```
case "$variable" in
  pattern1)  commands ;;
  pattern2)  commands ;;
  *)         default commands ;;
esac
```

Example: Service Control

```
#!/bin/bash
case "$1" in
  start)   systemctl start myapp ;;
  stop)    systemctl stop myapp ;;
  restart) systemctl restart myapp ;;
  status)  systemctl status myapp ;;
  *)       echo "Usage: $0 start|stop|restart|status"; exit 1 ;;
esac
```

🔁 Loops: `for`

C-Style For Loop

```
for ((i = 1; i ⩽ 5; i++)); do
    echo "Iteration $i"
done
```

Iterate Over a List

```
for USER in alice bob charlie; do
    echo "Creating user: $USER"
    useradd "$USER"
done
```

Iterate Over Files

```
for FILE in /var/log/*.log; do
    SIZE=$(du -sh "$FILE" | cut -f1)
    echo "$FILE: $SIZE"
done
```

Range

```
for i in {1..10}; do
    echo "Server-$i"
done
```

🔁 Loops: `while` & `until`

`while` – Loop While Condition is True

```
COUNT=0
while [[ $COUNT -lt 5 ]]; do
    echo "Count: $COUNT"
    ((COUNT++))
done
```

Reading a File Line by Line

```
while IFS= read -r LINE; do
    echo "Processing: $LINE"
done < /etc/passwd
```

`until` – Loop Until Condition is True

```
until ping -c1 -W2 google.com &>/dev/null; do
    echo "Waiting for network..."
    sleep 5
done
echo "Network is up!"
```

🧩 Functions

Defining Functions

```bash
greet() {
    local NAME="$1"
    echo "Hello, $NAME!"
}
greet "World"
```

Return Values

```bash
is_root() {
    [[ $(id -u) -eq 0 ]]
}
if is_root; then
    echo "Running as root"
else
    echo "Not root"; exit 1
fi
```

Returning Data (stdout capture)

```bash
get_ip() {
    hostname -I | awk '{print $1}'
}
MY_IP=$(get_ip)
echo "Server IP: $MY_IP"
```

💡 Always use `local` for function variables to avoid polluting the global scope.

📚 Arrays

Indexed Arrays

```
SERVERS=("web01" "web02" "db01")

echo "${SERVERS[0]}"        # web01
echo "${SERVERS[@]}"        # All elements
echo "${#SERVERS[@]}"       # Length: 3

SERVERS+=("monitor01")      # Append
unset SERVERS[2]            # Remove index 2
```

Associative Arrays (Bash 4+)

```
declare -A PORTS
PORTS[http]=80
PORTS[https]=443
PORTS[ssh]=22

for SERVICE in "${!PORTS[@]}"; do
    echo "$SERVICE → ${PORTS[$SERVICE]}"
done
```

Looping Over Arrays

```
for SERVER in "${SERVERS[@]}"; do
    ping -c1 -W2 "$SERVER" &>/dev/null \
      && echo "$SERVER: UP" \
      || echo "$SERVER: DOWN"
done
```

Part 3: Text Processing & Pipelines

Parsing Logs and Transforming Data

🔍 `grep` – Search Text

Basic Usage

```
grep "error" /var/log/syslog            # Find lines with "error"
grep -i "error" /var/log/syslog         # Case-insensitive
grep -n "error" /var/log/syslog         # Show line numbers
grep -c "error" /var/log/syslog         # Count matches
grep -r "TODO" /home/dev/project/       # Recursive search
```

Regular Expressions

```
grep -E "^root:" /etc/passwd            # Lines starting with "root:"
grep -E "failed|error" /var/log/auth.log # Match either pattern
grep -E "[0-9]{1,3}\.[0-9]{1,3}" access.log  # IP-like patterns
grep -v "^#" /etc/ssh/sshd_config       # Exclude comments
```

Practical: Failed SSH Logins

```
grep "Failed password" /var/log/auth.log | tail -20
grep "Failed password" /var/log/auth.log | \
  grep -oE "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+" | sort | uniq -c | sort -rn
```

✂️ `cut`, `sort`, `uniq` – Slice & Dice

`cut` – Extract Fields

```
cut -d: -f1 /etc/passwd              # Usernames (field 1, : delimited)
cut -d: -f1,3 /etc/passwd            # Username and UID
```

`sort` – Order Lines

```
sort /etc/passwd                     # Alphabetical sort
sort -t: -k3 -n /etc/passwd          # Sort by UID (numeric)
du -sh /var/log/* | sort -rh         # Sort by human-readable sizes
```

`uniq` – Remove Duplicates (requires sorted input)

```
sort access.log | uniq -c            # Count occurrences
sort access.log | uniq -c | sort -rn # Top occurrences
```

Combined Pipeline

```
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -10
```

🔧 `sed` – Stream Editor

Stream editor: transform text line-by-line (search/replace, print, delete).

Search & Replace

```
sed 's/old/new/' file.txt        # First match per line
sed 's/old/new/g' file.txt       # All matches (global)
sed -i 's/old/new/g' file.txt    # In-place edit
sed -i.bak 's/old/new/g' file    # In-place with .bak backup
```

Line Operations

```
sed -n '5,10p' file.txt          # Print lines 5–10 only
sed '3d' file.txt                # Delete line 3
sed '/^#/d' config.txt           # Delete lines starting with #
sed '/^$/d' file.txt             # Delete empty lines
```

Practical: Config Editing

```
sed -i 's/^#Port 22/Port 2222/' /etc/ssh/sshd_config
sed '/^#/d; /^$/d' /etc/ssh/sshd_config   # Strip comments and blanks
```

`awk` – Pattern Processing

Basic Structure: `awk 'pattern { action }' file`

```
awk '{print $1}' access.log                    # Print first field
awk '{print $1, $7}' access.log                # Print IP and URL
awk -F: '{print $1, $3}' /etc/passwd           # Custom delimiter
```

Filtering

```
awk '$3 > 1000' /etc/passwd                    # UID > 1000
awk '/error/ {print $0}' /var/log/syslog       # Lines matching "error"
awk 'NR ⩾ 10 && NR ⩽ 20' file.txt              # Lines 10-20
```

Built-in Variables & Computation

```
# Count lines
awk 'END {print NR}' file.txt

# Sum a column (e.g., bytes transferred)
awk '{sum += $10} END {print sum}' access.log

# Average response time
awk '{sum += $NF; n++} END {print sum/n}' access.log
```

🔗 Pipelines & Redirection

The Pipe (|)

```
grep "error" /var/log/syslog | wc -l
```

File Descriptors

FD	Name	Default
0	stdin	Keyboard
1	stdout	Terminal
2	stderr	Terminal

Redirection

```
command > file        # stdout (overwrite)
command >> file       # stdout (append)
command 2> file       # stderr to file
command &> file       # Both stdout & stderr
command 2>&1          # stderr to stdout
command < file        # File as stdin
```

Process Substitution

```
diff <(ls dir1) <(ls dir2)
```

📝 Practical: Log Analysis Pipeline

Analyze Apache/Nginx Access Log

```bash
#!/bin/bash
# log_report.sh - Generate access log report
LOG="${1:-/var/log/nginx/access.log}"

echo "═══ Access Log Report ═══"
echo "Generated: $(date)"

echo "--- Total Requests ---"
wc -l < "$LOG"

echo "--- Top 10 IP Addresses ---"
awk '{print $1}' "$LOG" | sort | uniq -c | sort -rn | head -10

echo "--- HTTP Status Code Summary ---"
awk '{print $9}' "$LOG" | sort | uniq -c | sort -rn

echo "--- Top 10 Requested URLs ---"
awk '{print $7}' "$LOG" | sort | uniq -c | sort -rn | head -10

echo "--- Requests Per Hour ---"
awk '{print $4}' "$LOG" | cut -d: -f2 | sort | uniq -c
```

Part 4: Automation & Best Practices

From Scripts to Production

⏰ Cron — Scheduling Tasks

Crontab Format

```
┌──────── minute (0-59)
│ ┌────── hour (0-23)
│ │ ┌──── day of month (1-31)
│ │ │ ┌── month (1-12)
│ │ │ │ ┌ day of week (0-7)
│ │ │ │ │
* * * * * command
```

Managing Crontab

```
crontab -e                # Edit crontab
crontab -l                # List entries
sudo crontab -u root -e   # Edit root's
```

Examples

```
# Every day at 2:30 AM
30 2 * * * /opt/scripts/backup.sh

# Every 15 minutes
*/15 * * * * /opt/scripts/health_check.sh

# Monday-Friday at 9 AM
0 9 * * 1-5 /opt/scripts/daily_report.sh

# First day of every month
0 0 1 * * /opt/scripts/monthly_cleanup.sh
```

⏰ Systemd Timers (Modern Alternative)

Timer Unit: `backup.timer`

```
[Unit]
Description=Daily backup timer

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

Service Unit: `backup.service`

```
[Unit]
Description=Backup service

[Service]
Type=oneshot
ExecStart=/opt/scripts/backup.sh
```

Managing Timers

```
sudo systemctl enable --now backup.timer
systemctl list-timers --all
systemctl status backup.timer
journalctl -u backup.service
```

💡 Systemd timers offer logging, dependencies, and better error handling than cron.

Error Handling & Strict Mode

Strict Mode

```bash
#!/bin/bash
set -euo pipefail
# -e  Exit immediately on error
# -u  Treat unset variables as errors
# -o pipefail  Catch errors in pipelines
```

Custom Error Handling

```bash
die() {
    echo "ERROR: $1" >&2
    exit "${2:-1}"
}
[[ -f "$CONFIG" ]] || die "Config not found"
```

Trap – Run Cleanup on Exit

```bash
#!/bin/bash
set -euo pipefail

TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

cp important_data "$TMPDIR/"
process_data "$TMPDIR/"
# TMPDIR removed automatically on exit
```

🐛 Debugging Scripts

Debug Mode

```
bash -x script.sh     # Trace every command
bash -n script.sh     # Syntax check only
bash -v script.sh     # Print as read
```

Selective Debugging

```
#!/bin/bash
echo "Normal output"
set -x                # Turn on debugging
problematic_function
set +x                # Turn off debugging
echo "Back to normal"
```

Debugging Tips

```
echo "DEBUG: VAR=$VAR" >&2

[[ $# -ge 2 ]] || {
  echo "Usage: $0 <src> <dest>"
  exit 1
}

shellcheck myscript.sh
```

💡 **ShellCheck** (`shellcheck.net`) catches common bugs – always run it on your scripts!

🔒 Security Best Practices

Input Validation

```bash
# Sanitize user input
if [[ ! "$USERNAME" =~ ^[a-zA-Z0-9_]+$ ]]; then
    die "Invalid username"
fi

# Avoid eval with user data
# BAD:  eval "$user_input"
# GOOD: Use arrays for commands
cmd=("ls" "-la" "$dir")
"${cmd[@]}"
```

Quote Everything

```bash
# BAD:  rm -rf $DIR/*
# GOOD: rm -rf "${DIR:?}/"*
# ${DIR:?} fails if DIR is empty
```

File Permissions

```bash
# Restrict script permissions
chmod 700 admin_script.sh

# Secure temp files
TMPFILE=$(mktemp)
chmod 600 "$TMPFILE"
```

Avoid Common Pitfalls

```bash
# Use full paths in cron
PATH=/usr/local/bin:/usr/bin:/bin

# Don't store passwords in scripts
# Use: read -sp "Password: " PASS
# Or:  PASS=$(cat /etc/myapp/secret)

# Log actions for audit
logger "Backup started by $(whoami)"
```

📝 Real-World Script: System Health Check

Script checks CPU load, memory, and disk usage against thresholds; logs WARN or OK.

```
#!/bin/bash
set -euo pipefail
WARN_CPU=80
WARN_MEM=85
WARN_DISK=90
HOST=$(hostname)
DATE=$(date '+%F %T')
echo "═══ Health: $HOST ═══"
echo "Date: $DATE"

# CPU: load vs cores → WARN or OK
LOAD=$(cat /proc/loadavg | cut -d' ' -f1)
CORES=$(nproc)
CPU_PCT=$((LOAD * 100 / CORES))
if [[ "$CPU_PCT" -gt "$WARN_CPU" ]]; then echo "[WARN] CPU: $CPU_PCT%"; else echo "[ OK ] CPU: $CPU_PCT%"; fi

# Memory: usage % → WARN or OK
MEM_PCT=$(free | awk '/Mem:/ {print int($3/$2*100)}')
if [[ "$MEM_PCT" -gt "$WARN_MEM" ]]; then echo "[WARN] Memory: $MEM_PCT%"; else echo "[ OK ] Memory: $MEM_PCT%"; fi
```

📝 Real-World Script: Backup with Rotation

```bash
#!/bin/bash
set -euo pipefail
# rotate_backup.sh

BACKUP_DIR="/backup"
SOURCE="/var/www"
RETENTION_DAYS=30
DATE=$(date +%Y%m%d_%H%M%S)
ARCHIVE="${BACKUP_DIR}/www_${DATE}.tar.gz"
LOGFILE="/var/log/backup.log"

log() {
  echo "$(date '+%F %T') $1" | tee -a "$LOGFILE"
}
```

```bash
log "Starting backup of $SOURCE"
mkdir -p "$BACKUP_DIR"

if tar czf "$ARCHIVE" \
    -C "$(dirname "$SOURCE")" \
    "$(basename "$SOURCE")"; then
    SIZE=$(du -sh "$ARCHIVE" | cut -f1)
    log "Backup complete: $ARCHIVE ($SIZE)"
else
    log "ERROR: Backup failed!"; exit 1
fi

# Rotate old backups
DELETED=$(find "$BACKUP_DIR" -name "*.tar.gz" \
  -mtime +$RETENTION_DAYS -delete -print | wc -l)
log "Cleaned up $DELETED old backup(s)"
```

📊 Summary: Bash Scripting & Automation

Key Concepts Covered

1. **Script Basics**: Shebang, execution, variables
2. **Special Variables**: `$1`, `$@`, `$?`, `$$`
3. **Control Structures**: `if`, `case`, `for`, `while`
4. **Functions**: `local` scope, return values
5. **Text Processing**: `grep`, `sed`, `awk`
6. **Pipelines**: `|`, redirection (`>`, `2>&1`)
7. **Automation**: Cron & systemd timers
8. **Best Practices**: Strict mode, security

The Scripting Mindset

- ✅ If you do it twice, script it
- ✅ Always use strict mode (`set -euo pipefail`)
- ✅ Quote your variables
- ✅ Validate inputs
- ✅ Log everything

🎯 Learning Objectives: Did We Achieve?

By now, you should be able to:

- ✅ Write and execute Bash scripts with proper structure
- ✅ Use variables, arguments, and arithmetic
- ✅ Implement conditionals and loops for control flow
- ✅ Define and use functions with local scope
- ✅ Process text with `grep`, `sed`, `awk`, and pipelines
- ✅ Schedule automated tasks with cron and systemd timers
- ✅ Apply error handling and strict mode
- ✅ Follow security best practices in scripts
- ✅ Build real-world system administration scripts

🎓 **Next Week:** Linux Networking Basics - Configure IP, DNS, and network interfaces!

🧪 Lab Practice: Bash Scripting

Exercise 1: User info script

- Write a script that takes a username as argument and prints: UID, GID, home dir, shell (use `getent` or parse `/etc/passwd`).
- Add a check: exit with an error message if no argument is given.

Exercise 2: Log summary

- In `/var/log` (or a copy), use `grep`, `cut`, `sort`, `uniq` to list the **top 5 most common** words in a log file (ignore case; skip very short words).
- Pipe the result into a small script that prints a one-line summary.

Exercise 3: Safe backup script

- Write a script that tars a given directory into `/tmp/backups` with a timestamp in the name.
- Use `set -euo pipefail`, check that the directory exists, and print the path of the created archive.

Exercise 4: `sed` config tweak

- Take a copy of a config file (e.g. `sshd_config` or any `.conf`). Use `sed` to comment out every line that contains a given keyword (e.g. `Port`), then show a diff.

🔗 Additional Resources

Documentation

- GNU Bash Manual - Official reference
- Bash Hackers Wiki - Community wiki
- ShellCheck - Online script linter

Books & Guides

- *"The Linux Command Line"* by William Shotts - Scripting chapters
- Advanced Bash-Scripting Guide - Comprehensive
- Google Shell Style Guide - Best practices

Practice

- Write a script to automate user account creation
- Build a log parser for Apache/Nginx access logs
- Create a backup script with rotation and email alerts

Questions?

Next: Linux Networking Basics