Essential Linux Command Reference Guide

Command 1: uname

(1) What is the uname command? (History, invention, author)

The uname command is a terminal utility program used in Unix/Linux to display detailed information about the system including kernel name, version, hardware platform, and operating system details.

History & Origin

- The uname command first appeared in **PWB/UNIX (Programmer's Workbench UNIX)**
- Both the system call and command are specified by **POSIX (Portable Operating System Interface)** standards
- **Author**: David MacKenzie wrote and developed the GNU version of uname
- The GNU version is included in the **"coreutils"** or **"sh-utils"** packages
- **Current Status**: uname is not available as a standalone program; it comes bundled with coreutils
- **Available across platforms**: Linux, macOS, Windows (via GnuWin32 project and UnxUtils)

Purpose

To allow system administrators and users to:

- Identify the operating system and kernel details
- Check system architecture and processor type
- Display hardware platform information
- Gather system metadata for scripting and automation
- Verify system compatibility before running specific software

(2) Why is uname useful for System Administrators?

System administrators frequently work with:

- Multiple Linux systems with different configurations
- Automated deployment and configuration scripts
- Software compatibility verification
- Cross-platform system environments
- Automated infrastructure management

Reasons for Use

- **Quick System Identification**: Shows OS name, kernel version, and hardware details instantly without complex commands
- **Scripting Support**: Output can be parsed in shell scripts for conditional execution based on system type

- **Compatibility Checking**: Helps verify if a system meets software requirements before installation
- **Multi-system Management**: Essential for managing heterogeneous environments (Linux, macOS, BSD)

(3) Basic Syntax and Examples

Basic Syntax:

```bash
uname [OPTIONS]
```

Common Options:

Option	Description
(none)	Prints kernel name (default)
-a	All information (kernel, hostname, kernel version, processor, OS)
-s	Kernel name
-n	Network node hostname
-r	Kernel release
-v	Kernel version
-m	Machine hardware name
-p	Processor type
-i	Hardware platform
-o	Operating system

Table 1: uname command options

Example 1: Display all system information

```bash
$ uname -a
Linux localhost.localdomain 5.10.0-14-generic #1 SMP Fri Oct 22 09:03:12 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

Example 2: Check kernel release for compatibility

```bash
$ uname -r
5.10.0-14-generic
```

Example 3: Display processor architecture

```bash
$ uname -m
x86_64
```

Example 4: Scripting with uname

```bash
#!/bin/bash
if [[ $(uname -s) == "Linux" ]]; then
echo "Running on Linux system"
KERNEL=$(uname -r)
echo "Kernel version: $KERNEL"
else
echo "Not running on Linux"
fi
```

(4) Practical Use Cases

- **Verify system before software installation**: Check architecture compatibility
- **Monitor fleet of servers**: Quickly identify OS versions across infrastructure
- **Automate system detection**: Write portable shell scripts that adapt to the running OS
- **Troubleshooting**: Identify kernel or hardware issues

Command 2: pgrep

(1) What is the pgrep command? (History, invention, author)

The pgrep command is a terminal utility program used in Unix/Linux to search for running processes based on specified criteria (name, user ID, terminal, etc.) and display their process IDs (PIDs) to standard output.

History & Origin

- pgrep is part of the **procps-ng package** (successor to procps package)
- Also available in **util-linux** and **procutil** packages depending on Linux distribution
- Originally developed as a more efficient alternative to parsing ps output with grep
- First appeared in **BSD Unix**, later adopted by Linux distributions worldwide
- Primary maintainers: Linux community and BSD developers
- Highly optimized for process searching without loading entire process list
- Available on Linux, BSD, macOS, and other Unix-like systems

Purpose

To allow users and administrators to:

- Find process IDs by process name without complex piping commands
- Search processes by user, terminal, or other attributes efficiently
- Count running instances of a specific process
- Kill specific process instances more safely than using wildcards
- Automate process monitoring and management in scripts

- Create condition-based scripts that check for process existence

(2) Why is pgrep useful for System Administrators?

System administrators frequently work with:

- Running services and application monitoring
- Process checking in automated cron jobs
- Service status verification for critical applications
- Process termination and cleanup tasks
- Application debugging and troubleshooting
- Resource allocation based on running processes

Reasons for Use

- **Simpler Alternative to ps | grep**: Much cleaner syntax compared to complex pipe operations
- **Avoids Matching grep Itself**: Prevents false matches that occur when using grep to search for processes
- **Direct PID Output**: Returns only process IDs, not entire process lines

(3) Basic Syntax and Examples

Basic Syntax:

```bash
pgrep [OPTIONS] PATTERN
```

Common Options:

Option	Description
-l	Show process name along with PID
-u USER	Search only for processes owned by USER
-U USER	Search only for processes NOT owned by USER
-t TERMINAL	Search only on specified terminal
-c	Count matching processes
-f	Match against full process command line
-g PID	Match process group ID
-P PID	Match parent process ID
-x	Match exact process name
-n	Return newest matching process

Table 2: pgrep command options

Example 1: Find all processes named 'apache2'

```bash
$ pgrep apache2
1234
1235
1236
```

Example 2: Find processes with names and PIDs

```bash
$ pgrep -l apache2
1234 apache2
1235 apache2
1236 apache2
```

Example 3: Count running processes of a specific user

```bash
$ pgrep -u pranav -c
23
```

Example 4: Find newest instance of a process

```bash
$ pgrep -n python3
5678
```

Example 5: Kill all processes matching a pattern

```bash
$ pkill -f "old_script.py"
```

(4) Practical Use Cases

- **Monitor service availability**: Check if critical processes are running
- **Automated startup scripts**: Prevent duplicate processes from starting
- **System cleanup**: Terminate zombie or hung processes efficiently
- **Log monitoring**: Send alerts if important services stop running

Command 3: ps (Process Status)

What is ps?

The **ps** command displays information about active processes currently running on the system. It provides detailed snapshots of process status including memory usage, CPU usage, and execution status.

Syntax:

```bash
ps [OPTIONS]
```

Common Use Cases:

- View running processes with detailed information
- Monitor resource consumption (CPU, memory)
- Track process hierarchy and parent-child relationships
- Identify zombie processes
- Track user-specific processes

Examples:

```bash
# Show all processes
$ ps aux

# Show process tree
$ ps -ef

# Show processes with memory usage sorted
$ ps aux --sort=%mem

# Show specific user's processes
$ ps -u username

# Show process hierarchy
$ ps --forest -ef
```

Command 4: top (Real-time Process Monitor)

What is top?

The **top** command provides real-time monitoring of system processes, showing CPU and memory usage, load averages, and process details in an interactive interface.

Key Features:

- Real-time process monitoring
- Interactive control (pause, resume, sort)
- System load and memory statistics

- Customizable display and sorting
- Process priority adjustment

Interactive Commands:

- **q**: Quit top
- **k**: Kill a process (specify PID)
- **r**: Change process priority (renice)
- **f**: Add/remove columns
- **P**: Sort by CPU usage
- **M**: Sort by memory usage

Example Usage:

```bash
# Launch top monitor
$ top

# Launch top and exit after 2 iterations
$ top -n 2

# Monitor specific process by PID
$ top -p 1234

# Batch mode output
$ top -b -n 1
```

Command 5: kill and pkill (Process Termination)

Terminating Processes Safely

The **kill** command sends signals to processes. The most common signals are:

Signal	Name	Effect
1	SIGHUP	Hang up / restart
2	SIGINT	Interrupt (Ctrl+C)
9	SIGKILL	Force kill (cannot be caught)
15	SIGTERM	Terminate gracefully (default)
19	SIGSTOP	Stop process (cannot be caught)

Table 3: Common kill signals

Safe Termination Process:

1. First attempt graceful termination (SIGTERM, signal 15)
2. Wait a few seconds for process cleanup
3. If necessary, force termination (SIGKILL, signal 9)

Examples:

```bash
# Graceful termination
$ kill -15 1234

# Force termination
$ kill -9 1234

# Kill by process name
$ pkill -f "python script.py"

# Kill all processes of a user
$ pkill -u username
```

Command 6: systemctl (System Service Management)

Managing System Services

The **systemctl** command controls the systemd system and service manager, essential for modern Linux distributions (Ubuntu, CentOS, Fedora, etc.).

Common Operations:

Command	Purpose
systemctl start SERVICE	Start a service
systemctl stop SERVICE	Stop a service
systemctl restart SERVICE	Restart a service
systemctl reload SERVICE	Reload configuration
systemctl enable SERVICE	Enable on boot
systemctl disable SERVICE	Disable on boot
systemctl status SERVICE	Check service status
systemctl list-units	List all services

Table 4: systemctl commands

Examples:

```bash
# Check if Apache is running
$ sudo systemctl status apache2

# Start the web server
$ sudo systemctl start apache2

# Enable Apache to start on boot
$ sudo systemctl enable apache2

# Restart PostgreSQL database
$ sudo systemctl restart postgresql

# View all running services
$ systemctl list-units --type=service --state=running
```

Real-World Scenarios and Scripts

Scenario 1: Monitoring Web Server Health

```bash
#!/bin/bash

SERVICE="apache2"
ADMIN_EMAIL="admin@example.com"

# Check if service is running
if pgrep -x "$SERVICE" > /dev/null; then
    echo "$SERVICE is running"
else
    echo "$SERVICE is not running - attempting restart"
    sudo systemctl restart $SERVICE

    # Check again
    if pgrep -x "\$SERVICE" > /dev/null; then
        echo "\$SERVICE restarted successfully"
    else
        echo "ALERT: \$SERVICE failed to start"
        # Send alert email
    fi
fi
```

Scenario 2: System Resource Audit

```bash
#!/bin/bash

echo "=== System Information ==="
uname -a
echo ""

echo "=== Top Memory Consumers ==="
ps aux --sort=-%mem | head -6
echo ""

echo "=== Top CPU Consumers ==="
ps aux --sort=-%cpu | head -6
echo ""

echo "=== Process Count by User ==="
pgrep -u . -c | sort -rn | head -5
```

```

### Scenario 3: Automated Cleanup Script

```bash
#!/bin/bash

# Kill processes older than 7 days without activity

OLD_PROCESSES=$(pgrep -f "inactive_task")

for PID in $OLD_PROCESSES; do
    echo "Terminating old process: $PID"
    kill -15 $PID

    # Force kill if not terminated
    sleep 2
    if kill -0 \$PID 2>/dev/null; then
        kill -9 \$PID
    fi
done
```

Best Practices and Tips

- **Always use graceful termination first**: Use SIGTERM (signal 15) before SIGKILL (signal 9)
- **Verify processes before killing**: Double-check the PID and process name
- **Use pgrep for scripts**: More reliable than ps | grep for automation
- **Monitor systemd services**: Use systemctl status regularly
- **Document critical processes**: Know which services are essential for your system
- **Test in non-production**: Always test process management scripts in development first
- **Keep logs**: Monitor syslog for process start/stop events
- **Use process monitoring tools**: Consider tools like monit, nagios for production systems

---

Conclusion

Understanding these essential Linux commands—**uname**, **pgrep**, **ps**, **top**, **kill**, and **systemctl**—is crucial for effective system administration. Each command serves a specific purpose in process management and system monitoring. Combined, they provide administrators with comprehensive tools to manage, monitor, and troubleshoot system processes efficiently.
```

Mastering these commands enables you to:

- Quickly diagnose system issues
- Automate routine administrative tasks
- Maintain system stability and performance
- Respond promptly to service failures
- Optimize resource utilization

Regular practice with these tools will improve your Linux administration skills and system management capabilities.

References

[1] Linux man-pages online documentation: https://man7.org/linux/man-pages/

[2] GNU Coreutils - uname documentation: https://www.gnu.org/software/coreutils/manual/

[3] procps-ng package documentation: https://gitlab.com/procps-ng/procps

[4] systemd system and service manager: https://systemd.io/

[5] Advanced Bash-Scripting Guide: Process Management chapter: http://tldp.org/LDP/abs/html/process-sub.html

[6] Red Hat System Administration Guide: Managing Processes: https://access.redhat.com/documentation/