CSS 262: Linux Administration

Process Management & Systemd

Lecture 3: Mastering Processes & Services

📋 Today's Agenda

Part 1: Process Management

- What are processes?
- Process lifecycle & states
- Process monitoring commands
- Managing processes (signals, jobs)
- Process priorities & nice values
- `/proc` filesystem

Part 2: Systemd

- Introduction to systemd
- Unit files & service management
- systemctl commands
- Creating custom services
- Boot process & targets
- Logs with journalctl

🎯 **Learning Objective:** Master process control and modern service management for production systems.

🔄 Quick Recap: Week 2

What We Covered

- User and group management
- File ownership and permissions (rwx)
- chmod, chown, chgrp commands
- Special permissions (SUID, SGID, sticky bit)
- Access Control Lists (ACLs)

Key Takeaway

Proper permission management is the foundation of Linux security - principle of least privilege.

✅ **Assumption:** You can now create users, manage groups, and set appropriate file permissions.

Part 1: Process Management

Understanding how Linux manages running programs

What is a Process?

Definition

A **process** is an instance of a running program. Every command you run, every application you open, creates at least one process.

Key Characteristics

- **PID:** Unique Process ID
- **PPID:** Parent Process ID
- **Owner:** User running the process
- **Memory:** Allocated RAM
- **State:** Running, sleeping, stopped, etc.

Process vs Program

- **Program:** Static file on disk
- **Process:** Program in execution
- One program can have multiple processes
    - Example: Multiple Firefox windows

💡 **Key Concept:** Everything in Linux runs as a process - from your shell to system services.

Process Hierarchy

Linux Process Tree

All processes descend from **init** (PID 1) - the first process started by the kernel.

```
systemd (PID 1)
├── sshd (PID 856)
│   └── sshd (PID 2341) - user connection
│       └── bash (PID 2342)
│           └── vim (PID 2450)
├── apache2 (PID 1024)
│   ├── apache2 (PID 1025) - worker
│   └── apache2 (PID 1026) - worker
└── cron (PID 789)
```

Orphan & Zombie Processes

- **Orphan:** Parent dies → adopted by init
- **Zombie:** Child exits but parent hasn't read status → `<defunct>`

Viewing Processes: ps

The Classic Process Snapshot Command

```
# Show your processes
ps

# Show all processes (BSD style)
ps aux

# Show process tree
ps auxf

# Show processes for specific user
ps -u john
```

Understanding ps aux Output

```
USER  PID  %CPU %MEM    VSZ   RSS TTY  STAT START TIME COMMAND
john 1234  2.5  1.3 123456 8192 pts/0 S+ 10:00 0:02 python script.py
```

Key Columns: USER (Owner), PID (Process ID), %CPU/%MEM (Usage), STAT (State), COMMAND (Program)

Process States

STAT Column in ps aux

Code	State	Description
R	Running	Executing on CPU
S	Sleeping	Waiting for event
D	Disk Sleep	Waiting for I/O
T	Stopped	Suspended (Ctrl+Z)
Z	Zombie	Terminated but not reaped
< / N	Priority	Nice < 0 / Nice > 0
+	Foreground	In foreground group

Examples: `S+` , `R<` , `Ss`

Dynamic Process Monitoring: top

Interactive Process Viewer

```
top                 # Launch top
top -u john         # Show only john's processes
top -p 1234,5678    # Monitor specific PIDs
```

Top Display (First Few Lines)

```
top - 10:23:45 up 5 days, load average: 0.52, 0.48, 0.45
Tasks: 245 total, 1 running, 244 sleeping
%Cpu(s): 12.5 us, 3.2 sy, 83.8 id, 0.3 wa
MiB Mem: 15924 total, 2341 free, 8456 used
MiB Swap: 2048 total, 2048 free, 0 used
```

Interactive Commands Inside top

- `k` - Kill process | `r` - Renice | `u` - Filter by user
- `M` - Sort by memory | `P` - Sort by CPU | `T` - Sort by time
- `1` - Show individual CPUs | `q` - Quit

💡 **Pro Tip:** Use `htop` for better UX

Understanding Load Average

The Three Numbers

```
load average: 0.52, 0.48, 0.45
              │     │     │
              │     │     └─ 15-minute average
              │     └─ 5-minute average
              └─ 1-minute average
```

What Does It Mean?

Load average = number of processes waiting for CPU time.

Rule of Thumb (for a 4-core system):

- `< 4.0` - System has capacity
- `= 4.0` - System fully utilized
- `> 4.0` - System overloaded

Example: N-core system at Load N.0 = 100% utilized

Modern Alternative: htop

Enhanced Process Viewer

```
sudo apt install htop      # Install
htop                       # Launch
```

Advantages over top

- ✅ Color-coded output
- ✅ Mouse support
- ✅ Visual CPU/memory bars
- ✅ Tree view built-in
- ✅ Easier to kill processes
- ✅ Scroll horizontally/vertically

Keyboard Shortcuts

- `F9` - Kill process
- `F7/F8` - Adjust nice value
- `F5` - Tree view
- `F6` - Sort by column
- `F4` - Filter processes
- `/` - Search
- `Space` - Tag multiple

🎯 **Recommendation:** Use htop for interactive monitoring, ps/top for scripting.

Process Signals

Signals are Messages Sent to Processes

```
kill -TERM 1234  # Send SIGTERM
kill -9 1234     # Force kill (SIGKILL)
killall firefox  # Kill by name
```

Common Signals: SIGHUP (1), SIGINT (2), SIGKILL (9), SIGTERM (15), SIGSTOP (19)

Killing Processes Properly

The Right Way to Stop Processes

```
# Step 1: Try graceful shutdown
kill 1234
kill -TERM 1234      # Same as above (TERM is default)

# Wait 5-10 seconds...

# Step 2: If still alive, force kill
kill -9 1234
kill -KILL 1234      # Same as above
```

Good Practice ✅

1. Try SIGTERM first
2. Wait for graceful shutdown
3. Use SIGKILL as last resort

Avoid ❌

- Starting with `kill -9`
- Killing init (PID 1)
- Killing critical system processes

Foreground vs Background Jobs

Job Control in the Shell

```bash
# Run in background
./script.sh &

# Suspend running process (Ctrl+Z), then resume
bg          # Resume in background
fg          # Resume in foreground

# Job control
jobs        # List jobs
fg %2       # Bring job 2 to foreground
bg %1       # Send job 1 to background
```

Process Management Examples

Real-World Scenarios

```
# Find and kill all Python processes
pkill -f python

# Find process using specific port
lsof -i :8080

# Kill process using port 8080
kill $(lsof -t -i :8080)

# See what files a process has open
lsof -p 1234
```

Process Priority: nice & renice

Controlling Process CPU Priority

Nice values: -20 (highest priority) to 19 (lowest priority) • **Default:** 0

```
# Start with low priority (nice)
nice -n 10 ./cpu-intensive-task.sh

# Start with high priority (requires root)
nice -n -10 ./important-task.sh

# Change priority of running process
renice -n 5 -p 1234           # Set nice to 5
renice -n 10 -u john          # All john's processes

# View nice values
ps -eo pid,ni,cmd             # Show PID, nice, command
```

💡 **Use Case:** Run backups with `nice 19`

The /proc Filesystem

Virtual filesystem exposing kernel and process data.

Examples:

- `/proc/1234/` - Process-specific info
- `/proc/cpuinfo` - System-wide info
- `/proc/meminfo` - Memory details

Useful Process Commands

Command	Purpose
`ps` / `top` / `htop`	View/monitor processes
`pgrep` / `kill`	Find/terminate PID
`nice` / `renice`	Adjust priority

Part 2: Systemd

🔧

Modern Linux service management and init system

What is Systemd?

The Modern Init System

Systemd is the init system (PID 1) used by most modern Linux distributions. It manages system boot, services, and system state.

Replaced

- **SysVinit** (traditional init scripts)
- **Upstart** (Ubuntu's previous init)

Key Features

- Parallel service startup
- On-demand activation
- Service dependencies
- Resource control (cgroups)

Systemd Components

- `systemctl` - Service manager
- `journalctl` - Log viewer
- `systemd-analyze` - Boot analysis
- `hostnamectl` - Hostname management
- `timedatectl` - Time/date settings

Systemd Units

Everything in Systemd is a Unit

Unit Type	Extension	Purpose
Service	`.service`	System services
Socket	`.socket`	IPC sockets
Target	`.target`	Group of units
Timer	`.timer`	Scheduled tasks

Unit File Locations:

1. `/etc/systemd/system/` - Admin units
2. `/lib/systemd/system/` - Package units

Basic Service Management: systemctl

Essential Commands

```
# Service Status
systemctl status nginx             # Detailed status
systemctl is-active nginx          # Check if running
systemctl is-enabled nginx         # Check if starts at boot

# Start/Stop/Restart
systemctl start nginx              # Start service
systemctl stop nginx               # Stop service
systemctl restart nginx            # Stop, then start
systemctl reload nginx             # Reload config (if supported)
systemctl reload-or-restart nginx  # Reload if possible, else restart

# Enable/Disable (boot time)
systemctl enable nginx             # Start at boot
systemctl disable nginx            # Don't start at boot
systemctl enable --now nginx       # Enable and start immediately

# View Configuration
systemctl cat nginx                # Show unit file
systemctl show nginx               # Show all properties
systemctl list-dependencies nginx  # Show dependencies
```

Listing Services

Finding and Filtering Units

```
# List services
systemctl list-units --type=service
systemctl list-units --type=service --all    # Including inactive

# List failed/enabled
systemctl --failed
systemctl list-unit-files --state=enabled

# Search
systemctl list-units | grep ssh
```

Pro Tip: Use `systemctl --failed` to find problems

Understanding Service Status

Reading systemctl status Output

```
systemctl status nginx
```

```
● nginx.service - A high performance web server
     Loaded: loaded (/lib/systemd/system/nginx.service; enabled)
     Active: active (running) since Mon 2026-02-10 10:30:15 UTC
   Main PID: 1234 (nginx)
      Tasks: 5 (limit: 4620)
     Memory: 12.5M
```

Key Information

- **Loaded:** Unit file location and boot status
- **Active:** Current state (active, inactive, failed)
- **Main PID:** Primary process ID
- **Memory/CPU:** Resource usage

Service Unit File Structure

Anatomy of a .service File

```
[Unit]
Description=My Custom Application
After=network.target                    # Start after network
Requires=postgresql.service             # Dependency required

[Service]
Type=simple                             # Service type
User=appuser                            # Run as user
WorkingDirectory=/opt/myapp
Environment="NODE_ENV=production"
ExecStart=/usr/bin/node server.js
ExecReload=/bin/kill -HUP $MAINPID     # Reload command
Restart=on-failure                      # Auto-restart on crash

[Install]
WantedBy=multi-user.target              # Enable in this target
```

Service Types

Type= Directive Options

Type	Description	Use Case
simple	Main process is ExecStart	Foreground services
forking	Process forks, parent exits	Daemons (nginx, apache)
oneshot	Process exits after starting	Setup scripts

Examples:

- `Type=simple` - Process stays in foreground
- `Type=forking` - Traditional daemon with PIDFile
- `Type=oneshot` - Run once and exit

Creating a Custom Service

Example: Python Web Application

1. Create service file: `/etc/systemd/system/myapp.service`

```
[Unit]
Description=My Python Web Application
After=network.target

[Service]
Type=simple
User=www-data
WorkingDirectory=/opt/myapp
ExecStart=/opt/myapp/venv/bin/python app.py
Restart=always

[Install]
WantedBy=multi-user.target
```

2. Activate:

```
sudo systemctl daemon-reload
sudo systemctl enable --now myapp.service
```

Systemd Targets

Targets = Runlevels in Systemd

Targets define system states (similar to SysV runlevels).

Target	Description
`poweroff.target`	Shutdown
`rescue.target`	Single-user mode
`multi-user.target`	Multi-user, no GUI
`graphical.target`	Multi-user with GUI
`reboot.target`	Reboot

Commands:

```
systemctl get-default
sudo systemctl set-default multi-user.target
```

Systemd Timers

Modern alternative to cron for scheduled tasks.

Files needed:

- `backup.service` - Service definition with `ExecStart`
- `backup.timer` - Timer with `OnCalendar=*-*-* 02:00:00`

Activate: `systemctl enable --now backup.timer`

Viewing Logs: journalctl

Systemd's unified logging system.

View logs:

- `journalctl` - All logs
- `journalctl -f` - Follow

Filter:

- `journalctl -u SERVICE` - By service
- `journalctl --since "1 hour ago"` - By time

Advanced journalctl

More Powerful Log Analysis

```
# Kernel messages
journalctl -k

# Multiple services
journalctl -u nginx -u apache2 -f

# Limit output
journalctl -n 50              # Last 50 lines

# Disk management
journalctl --disk-usage
```

Boot Analysis: systemd-analyze

Analyze System Boot Performance

```
# Overall boot time
systemd-analyze
# Output: Startup finished in 2.5s (kernel) + 8.3s (userspace) = 10.8s

# Service breakdown & critical path
systemd-analyze blame
systemd-analyze critical-chain

# Generate boot chart & verify units
systemd-analyze plot > boot-chart.svg
systemd-analyze verify /etc/systemd/system/myapp.service
```

💡 Use `systemd-analyze blame` to find slow services

Resource Control with Systemd

Limiting Service Resources

```
[Service]
# CPU limits
CPUQuota=50%                    # Max 50% of one CPU

# Memory limits
MemoryMax=512M                  # Hard limit

# Task limits
TasksMax=100                    # Max processes/threads

# I/O limits
IOWeight=100                    # I/O priority
```

Apply temporarily:

```
systemctl set-property nginx.service MemoryMax=1G
```

Dependency Management

Controlling Service Relationships

```
[Unit]
# Ordering: Start after these units
After=network.target postgresql.service

# Ordering: Start before these units
Before=nginx.service

# Requirements: Won't start without these (hard dependency)
Requires=postgresql.service

# Wants: Prefer these but can start without (soft dependency)
Wants=redis.service

# Conflicts: Can't run together
Conflicts=apache2.service

# Conditions: Only start if condition met
ConditionPathExists=/opt/myapp/config.yml
ConditionFileNotEmpty=/etc/myapp/config
```

Debugging Failed Services

Troubleshooting Checklist

```bash
# Check status and logs
systemctl status myapp.service
journalctl -u myapp.service -n 50

# Verify unit file
systemd-analyze verify /etc/systemd/system/myapp.service

# Test manually
sudo -u www-data /path/to/binary

# Check dependencies and permissions
systemctl list-dependencies myapp.service
ls -la /opt/myapp/
```

Common Issues: Wrong user/group, missing dependencies, incorrect paths, permissions

Systemd Best Practices

DO ✅

- Use `Type=simple` for foreground apps
- Set appropriate `User=` and `Group=`
- Define clear dependencies
- Use `Restart=on-failure` for critical services
- Document with `Description=` and `Documentation=`
- Set resource limits
- Use `After=network.target` for network services
- Test unit files with `systemd-analyze verify`
- Use timers instead of cron
- Keep unit files in `/etc/systemd/system/`

DON’T ❌

- Run services as root unless necessary
- Use `Type=forking` for simple services
- Ignore failed dependencies
- Set `Restart=always` on oneshot services
- Edit files in `/lib/systemd/system/`
- Forget `daemon-reload` after changes
- Use absolute paths in `ExecStart` without testing
- Mix systemd and init.d scripts
- Ignore log messages
- Leave broken services enabled

Real-World Examples

Example 1: Node.js API Server

```
[Unit]
Description=Node.js API Server
After=network.target

[Service]
ExecStart=/usr/bin/node server.js
Restart=always

[Install]
WantedBy=multi-user.target
```

Example 2: Database Backup Timer

```
# backup-db.service
[Service]
Type=oneshot
ExecStart=/usr/local/bin/backup.sh

# backup-db.timer
[Timer]
OnCalendar=*-*-* 03:00:00
```

Quick Reference: systemctl

Command	Purpose
`systemctl start/stop/restart SERVICE`	Start, stop, or restart
`systemctl status SERVICE`	View status
`systemctl enable/disable SERVICE`	Start at boot or not
`systemctl is-active SERVICE`	Check if running
`systemctl list-units --type=service`	List services
`systemctl --failed`	List failed services
`systemctl daemon-reload`	Reload systemd config
`systemctl cat SERVICE`	Show unit file

Quick Reference: journalctl

Command	Purpose
`journalctl` / `journalctl -f`	View all / follow logs
`journalctl -u SERVICE`	Logs for service
`journalctl -b` / `-k`	Boot logs / kernel
`journalctl -p err`	Only errors
`journalctl --since "1 hour ago"`	Time filter
`journalctl -n 50`	Last 50 lines
`journalctl --vacuum-time=7d`	Clean old logs

Lab 3 Preview: Process & Service Management

What You'll Do

1. ⚙️ Monitor processes with ps, top, and htop
2. 🎯 Practice process control (signals, priorities)
3. 🔧 Create a custom systemd service
4. ⏰ Set up a systemd timer
5. 📊 Analyze boot performance
6. 📝 Troubleshoot failed services

Deliverables

Process monitoring report, custom service file, timer config, boot analysis

Time Estimate: 2-3 hours

Common Scenarios

Scenario 1: High CPU Usage

```
# Find and investigate
top                              # Press 'P' to sort by CPU
ps aux --sort=-%cpu | head -10   # Top 10 consumers
lsof -p PID                      # What files?

# Fix
renice -n 10 -p PID
kill PID
```

Scenario 2: Service Won’t Start

```
systemctl status myapp.service
journalctl -u myapp.service -n 50
systemd-analyze verify /etc/systemd/system/myapp.service
```

Scenario 3: Find Zombie Processes

```
ps aux | grep 'Z'
kill -9 PARENT_PID               # Kill parent to reap
```

Week 3 Action Items

✅ Before Next Lecture

1. Read **Chapter 10** (Processes)
2. Practice process monitoring
3. Experiment with systemctl on your VM
4. Create a simple custom service

✅ For Lab This Week

1. Complete **Lab 3: Process Management & Systemd**
2. Monitor processes, create services, set up timers
3. Debug service failures and optimize boot

📝 Practice

Find resource-hungry processes, create services, set up timers, debug configurations

Key Takeaways

Process Management

- Every program runs as a process with a unique PID
- Use `ps`, `top`, and `htop` to monitor processes
- Signals control process behavior - prefer SIGTERM over SIGKILL
- Nice values control CPU priority (-20 to 19)
- `/proc` filesystem exposes kernel information

Systemd

- Modern init system and service manager
- Unit files define service behavior
- `systemctl` manages services, `journalctl` views logs
- Timers replace cron for scheduled tasks
- Resource limits prevent runaway services
- Dependencies ensure proper startup order

🎯 **Mastery Goal:** Be comfortable troubleshooting process issues and managing services in production.

Questions?

❓

Processes and services are the heart of a running Linux system!

Next Week: Storage, Filesystems & LVM 💾

Thank You!

⚙️

Remember: Master the fundamentals - processes and services - to troubleshoot anything!

CSS 262 - Linux Administration & *nix Systems for Cybersecurity