

MATHEMATICAL MODELING FOR COMPUTATIONAL SCIENCE

AM215 - LECTURE 1 APPENDIX: Advanced Linux Topics

Friday, September 5th, 2025

■ Appendix: Advanced Linux Topics

This appendix contains detailed information on advanced Linux topics that complement the main lecture. These slides provide deeper coverage for students who want to explore topics further.

■ Advanced File Finding with Actions

`find` becomes even more powerful when you can act on the files you discover:

```
find . -name "*.txt" -exec wc -l {} \;
```

[finished]

```
1 ./Documents/sample.txt
1 ./temp_file.txt
10 ./sample_data.txt
```

This finds all `.txt` files and counts the lines in each one.

Understanding the syntax:

- `-exec command {} \;`: Execute command on each found file
- `{}`: Placeholder for the current file
- `\;`: Terminates the command (semicolon must be escaped)

More useful examples:

```
# Find and delete old temporary files
find /tmp -name "*.tmp" -mtime +7 -exec rm {} \;

# Find large files and show their sizes
find . -size +10M -exec ls -lh {} \;

# Find Python files and check their syntax
find . -name "*.py" -exec python3 -m py_compile {} \;
```

Safety tip: Use `-exec echo {}` first to see what files would be affected before running destructive commands!

■ File Links: Creating Shortcuts and References

Linux supports two types of file links:

```
# Create a symbolic link (shortcut)
ln -s my_project/README.md readme_link

# Create a hard link (another name for the same file)
ln my_project/README.md readme_hardlink
```

Try this in a new terminal: Link creation modifies the filesystem.

Symbolic links (soft links):

- Point to another file by path
- Can link to files on different filesystems
- Broken if original file is deleted
- Show as `l` in `ls -l` output

Hard links:

- Multiple names for the same file data
- Share the same inode (file system identifier)
- Original and link are indistinguishable
- File data persists until all hard links are deleted

When to use each: Symbolic links for shortcuts and cross-file system references, hard links for backup copies that share storage.

■ File Synchronization with `rsync`

For copying and synchronizing files, especially over networks, `rsync` is your friend:

```
# Basic file copy (like cp but smarter)
rsync -av my_project/ backup_project/

# Sync to remote server (if you have SSH access)
rsync -av my_project/ user@server:/path/to/backup/

# Show what would be transferred without doing it
rsync -av --dry-run my_project/ backup_project/
```

Try this in a new terminal: `rsync` modifies files and directories.

Why use `rsync` over `cp`?

- **Incremental:** Only copies changed files
- **Network aware:** Efficient over slow connections
- **Preserves attributes:** Permissions, timestamps, ownership
- **Progress display:** Shows transfer progress
- **Dry run:** Test before actual transfer

Common `rsync` options:

- `-a`: Archive mode (preserves everything)
- `-v`: Verbose output
- `-z`: Compress during transfer
- `--delete`: Remove files from destination that don't exist in source
- `--progress`: Show transfer progress

■ Understanding File Creation Defaults: `umask`

When you create new files, Linux uses `umask` to set default permissions:

```
umask
```

```
[finished]
```

```
0022
```

Shows your current `umask` value (usually `0022`)

How `umask` works:

- Default file permissions: `666` (read/write for all)
- Default directory permissions: `777` (read/write/execute for all)
- `umask 022` removes write permission for group and others
- Result: files get `644`, directories get `755`

Common `umask` values:

- `022`: Files `644`, directories `755` (standard)
- `002`: Files `664`, directories `775` (group-writable)
- `077`: Files `600`, directories `700` (private)

Changing `umask`:

```
# More restrictive (private files)
umask 077

# More permissive (group collaboration)
umask 002
```

Make permanent: Add `umask 022` to your `~/.bashrc`

Here documents for multi-line text:

```
cat << EOF
This is a multi-line message.
It can contain variables like $USER.
And spans multiple lines.
EOF

# Write to a file
cat << 'EOF' > config.txt
# Configuration file
debug=true
log_level=info
EOF
```

Arrays for storing lists:

```
# Create an array
files=("file1.txt" "file2.txt" "file3.txt")

# Access elements
echo "First file: ${files[0]}"
echo "All files: ${files[@]}"
echo "Number of files: ${#files[@]}"

# Loop over array
for file in "${files[@]"; do
    echo "Processing: $file"
done
```

Essential practices for reliable scripts:

```
#!/usr/bin/env bash

# Enable strict mode
set -euo pipefail

# Use meaningful variable names
input_file="$1"
output_dir="/tmp/processed"
timestamp=$(date +%Y%m%d_%H%M%S)

# Always quote variables
if [ -f "$input_file" ]; then
    cp "$input_file" "$output_dir/backup_${timestamp}"
fi

# Use functions for repeated code
log_message() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"
}

# Provide usage information
usage() {
    echo "Usage: $0 <input_file>"
    echo "Process input file and create backup"
    exit 1
}

# Check arguments
[ $# -eq 1 ] || usage
```

Key principles:

- Use strict mode (`set -euo pipefail`)
- Quote all variables
- Provide clear error messages
- Include usage information
- Comment your code

Techniques for finding and fixing script problems:

```
# Run with debugging output
bash -x myscript.sh

# Add debugging to script
#!/usr/bin/env bash
set -x # Print each command before executing

# Conditional debugging
DEBUG=${DEBUG:-false}
if [ "$DEBUG" = "true" ]; then
    set -x
fi

# Add debug messages
debug() {
    if [ "$DEBUG" = "true" ]; then
        echo "DEBUG: $" >&2
    fi
}

debug "Processing file: $filename"
```

Common debugging approaches:

- `bash -x script.sh`: Show all executed commands
- `set -x` in script: Enable command tracing
- Add `echo` statements to show variable values
- Use `DEBUG` environment variable for conditional debugging

Run with debugging: `DEBUG=true ./myscript.sh`

What is POSIX?

- **P**ortable **O**perating **S**ystem **I**nterface
- A standard that defines how Unix-like systems should behave
- Ensures scripts work across different Unix/Linux systems

POSIX-compliant scripting:

```
#!/bin/sh # Use /bin/sh for maximum compatibility

# POSIX-compliant test syntax
if [ "$var" = "value" ]; then
    echo "POSIX compliant"
fi

# Avoid bash-specific features like:
# [[ ]] (use [ ] instead)
# $((arithmetic)) in some contexts
# Arrays (not in POSIX)
```

When to care about POSIX:

- Writing scripts for multiple systems
- Working in enterprise environments
- Contributing to open source projects
- Maximum portability is required

When bash-specific features are okay:

- Personal scripts on your own system
- When you know the target environment
- When you need advanced features like arrays

More powerful screen techniques:

```
# Start named session
screen -S myproject

# List all sessions
screen -ls

# Reattach to specific session
screen -r myproject

# Force reattach (if stuck)
screen -D -r myproject
```

Try this in a new terminal: Named sessions help organize different projects.

Screen workflow:

1. Start screen session for each project
2. Create windows for different tasks (editing, testing, monitoring)
3. Detach when switching projects
4. Reattach when returning to work

Essential screen commands:

- **Ctrl+A, A**: Go to beginning of line (useful in shell)
- **Ctrl+A, K**: Kill current window
- **Ctrl+A, "**: List all windows
- **Ctrl+A, S**: Split screen horizontally

■ Introduction to `tmux`: Modern Terminal Multiplexing

`tmux` is a more modern alternative to `screen`:

```
# Start new tmux session
tmux

# Basic tmux commands (prefix: Ctrl+B)
# Ctrl+B, C - Create new window
# Ctrl+B, N - Next window
# Ctrl+B, P - Previous window
# Ctrl+B, D - Detach session
# Ctrl+B, % - Split vertically
# Ctrl+B, " - Split horizontally

# Reattach to session
tmux attach
```

Try this in a new terminal: `tmux` has better defaults and more features than `screen`.

`tmux` advantages:

- Better pane management (split windows)
- More intuitive commands
- Better status bar
- Easier configuration
- Active development

Essential keyboard shortcuts for terminal efficiency:

Process control:

- **Ctrl+C**: Interrupt current process
- **Ctrl+Z**: Suspend current process
- **Ctrl+D**: End input (logout if at prompt)

Command line editing:

- **Ctrl+A**: Beginning of line
- **Ctrl+E**: End of line
- **Ctrl+U**: Clear line before cursor
- **Ctrl+K**: Clear line after cursor
- **Ctrl+W**: Delete word before cursor

History navigation:

- **Ctrl+R**: Search command history
- **Ctrl+P** / **↑**: Previous command
- **Ctrl+N** / **↓**: Next command

Practice these shortcuts - they'll save you hours of typing!

■ Command History: Never Retype Again

Your shell remembers everything you type:

```
history | tail -10
```

_____ [finished] _____

Shows your recent command history.

History shortcuts:

- **!!**: Repeat last command
- **!n**: Repeat command number n
- **!string**: Repeat last command starting with "string"
- **^old^new**: Replace "old" with "new" in last command

History search:

- **Ctrl+R**: Interactive search
- **history | grep pattern**: Search all history

Customize history:

```
# In ~/.bashrc
export HISTSIZE=10000      # Commands in memory
export HISTFILESIZE=20000  # Commands in file
export HISTCONTROL=ignoredups # Ignore duplicates
```

■ File Compression: Advanced Options

Beyond basic compression:

Individual compression tools:

```
# Compress a single file
gzip large_file.txt          # Creates large_file.txt.gz
gunzip large_file.txt.gz     # Restores large_file.txt

# Better compression (slower)
bzip2 large_file.txt        # Creates
                             large_file.txt.bz2
bunzip2 large_file.txt.bz2  # Restores large_file.txt
```

When to use what:

- **tar + gzip**: General purpose, good speed/compression balance
- **tar + bzip2**: Better compression for archival storage
- **zip**: Cross-platform compatibility (Windows, macOS)
- Individual **gzip**/**bzip2**: Single files, streaming compression

Advanced tar options:

- **--exclude**: Skip certain files or patterns
- **--update**: Only add files newer than archive versions
- **--verify**: Verify archive after creation

Basic networking commands for everyday use:

```
ping -c 3 google.com
```

————— [finished] —————

```
PING google.com (142.250.64.78) 56(84) bytes of data.  
64 bytes from lga34s30-in-f14.1e100.net (142.250.64.78):  
icmp_seq=1 ttl=115 time=24.9 ms  
64 bytes from lga34s30-in-f14.1e100.net (142.250.64.78):  
icmp_seq=2 ttl=115 time=24.1 ms  
64 bytes from lga34s30-in-f14.1e100.net (142.250.64.78):  
icmp_seq=3 ttl=115 time=22.5 ms  
  
--- google.com ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time  
2002ms  
rtt min/avg/max/mdev = 22.488/23.822/24.873/0.994 ms
```

Test network connectivity (sends 3 packets).

Essential network commands:

```
# Check network interfaces  
ip addr show  
  
# Test DNS resolution  
nslookup google.com  
  
# Show network connections  
netstat -tuln  
  
# Download files  
wget https://example.com/file.txt  
curl -O https://example.com/file.txt
```

Try these in a new terminal: Network commands work better in separate terminals.

When to use:

System logs contain valuable troubleshooting information:

```
ls /var/log/
```

————— [finished] —————

```
btmptmp
gdm
journal
lastlog
libvirt
lightdm
private
wtmp
X.0.log
```

Common log locations (may vary by system).

Important log files:

- `/var/log/syslog`: General system messages
- `/var/log/auth.log`: Authentication attempts
- `/var/log/kern.log`: Kernel messages
- `~/.bash_history`: Your command history

Viewing logs safely:

```
# View recent entries
tail -20 /var/log/syslog

# Follow log in real-time
tail -f /var/log/syslog

# Search for errors
grep -i error /var/log/syslog
```

Note: Log access may require sudo privileges on some systems.

■ Troubleshooting Configuration Issues

What to do when your shell configuration breaks:

If your terminal won't start:

```
# Use a different shell temporarily
/bin/bash --norc

# Restore your backup
cp ~/.bashrc.backup ~/.bashrc
```

If commands don't work:

```
# Check for syntax errors
bash -n ~/.bashrc

# Start with minimal config
mv ~/.bashrc ~/.bashrc.broken
cp /etc/skel/.bashrc ~/.bashrc
```

Best practices:

- Always backup before editing
- Add changes gradually
- Test in a new terminal
- Comment your customizations
- Keep a working backup

Remember: You can always start fresh with the default configuration if needed!

Fish prioritizes user experience and discoverability:

```
# Check if fish is available
which fish

# Switch to fish temporarily
fish

# Fish features to try:
# - Syntax highlighting as you type
# - Auto-suggestions from history
# - Web-based configuration
```

Try this in a new terminal: Fish has a very different syntax from bash.

Fish advantages:

- **Syntax highlighting:** See errors as you type
- **Auto-suggestions:** Ghost text from history
- **No configuration needed:** Works great out of the box
- **Web interface:** Configure through browser
- **Intuitive scripting:** More readable than bash

Trade-off: Not POSIX compliant - bash scripts won't work directly

Even in GUI environments, you'll use terminal emulators:

Popular terminal emulators:

- **GNOME Terminal:** Default on many Linux distributions
- **Konsole:** KDE's feature-rich terminal
- **Alacritty:** GPU-accelerated, very fast
- **Kitty:** Modern with advanced features
- **iTerm2:** Popular on macOS
- **Windows Terminal:** Microsoft's modern terminal

Features to look for:

- Tabs and split panes
- Color themes and customization
- Font rendering quality
- Performance with large outputs
- Integration with shell features

A typical progression for new Linux users:

Stage 1: GUI Comfort

- Use desktop environment like Windows/macOS
- Occasionally open terminal for tutorials
- Copy-paste commands without understanding

Stage 2: Terminal Curiosity

- Start using terminal for file operations
- Learn basic commands (`ls`, `cd`, `cp`, `mv`)
- Begin to see the power of command combinations

Stage 3: CLI Proficiency

- Comfortable with pipes and redirection
- Write simple shell scripts
- Prefer terminal for many tasks

Stage 4: Power User

- Mix CLI and GUI based on task efficiency
- Extensive shell customization
- Help others learn the command line

Remember: There's no "right" level - use what works for your needs!

