Comprehensive Operating Systems Lab Manual

Instructors: Muhammad Hamza & Muhammad Abdullah Orakzai

Spring 2022 & Fall 2021

Contents

1 Lab #01: Booting and Linux Basics **4**
1.1 How a PC Turns On 4
1.2 Types of Firmware 4
1.3 Introduction to Linux 4
1.4 Practical Orientation 4
1.5 Exercise 4

2 Lab #02: Basics of Ubuntu **5**
2.1 Ubuntu's File System Structure 5
2.2 Terminal and the Command-Line Interface (CLI) 5
2.2.1 Why Use the Terminal? 5
2.3 Basic Commands 6
2.4 Exercises 6

3 Lab #03: Commands and Syntax **6**
3.1 Overview 6
3.2 Command Syntax 7
3.3 The Asterisk (*) Wildcard 7
3.4 Case Sensitivity 7
3.5 Auto-Completion 7
3.6 Practicing Common Commands 7
3.6.1 Working with Directories and Files 8
3.7 Redirection 8
3.8 Exercise 1 8

4 Lab #04: Advanced Linux Features **9**
4.1 NANO Editor 9
4.2 Links 9
4.2.1 Hard Links 9
4.2.2 Soft Links (Symbolic Links) 10
4.3 More and Less Commands 10
4.4 Searching Files 10
4.4.1 find Command 10
4.4.2 locate Command 10
4.5 File Permissions 11

4.5.1 Changing File Permissions with chmod 11
4.5.1.1 Absolute (Numeric) Mode 11
4.5.1.2 Symbolic Mode 11
4.6 Directory Permissions 12
4.7 File or Directory Ownership 12
4.8 GNU Compiler Collection (GCC) 12
4.9 Executing Linux Commands from a C Program 12
4.10 Exercise 13

5 Lab #06: Process Management **13**
5.1 Introduction to Processes 13
5.2 Viewing Processes with `ps` 13
5.2.1 Process States (STAT Codes) 14
5.3 Key System Calls 14
5.4 Forking and Process Hierarchy 15
5.5 Process Creation, States, and Lifecycle 15
5.6 Replacing Process Image with exec() 16
5.7 Delaying and Terminating Processes 16
5.8 Exercises 16
5.9 Summary 17

6 Lab #8: Input/Output **17**
6.1 Standard I/O Streams and File Descriptors 17
6.2 Opening and Closing Files 18
6.3 Reading and Writing Data 18
6.4 Examples 18
6.5 Exercise 19

7 Lab #09: Pipes for IPC **19**
7.1 Introduction to Pipes 19
7.2 Creating Pipes in C 19
7.3 Fork and Pipes 20
7.4 Using dup2() for Redirection 20
7.5 Exercise 20

8 Lab #10: IPC - Signals **21**
8.1 Introduction to Signals 21
8.2 Using the `kill` Command 21
8.3 Using the `kill()` System Call 21
8.4 Signal Handling with `signal()` 22
8.5 Exercises 22

9 Lab #11: Shell Scripting **22**
9.1 Introduction 22
9.2 Shell Basics 22
9.3 Wildcards and Redirection 23
9.4 Examples of Arithmetic and Loops 23
9.5 Control Structures 24
9.6 Functions in Shell Scripts 24

9.7 Exercise 24
9.8 Exercise 25

10 Lab #12: POSIX Threads 25
10.1 Introduction 25
10.2 Thread Creation 25
10.3 Passing Multiple Arguments 26
10.4 Thread Termination 26
10.5 Synchronization with Mutex 26
10.6 Exercise 27

11 Lab #13: Semaphores 27
11.1 Introduction 27
11.2 Basic Operations 28
11.3 Example 1: Binary Semaphore (One at a Time) 28
11.4 Example 2: Counting Semaphore (Multiple at a Time) 29
11.5 Example 3: Binary Semaphore as a Signal 30
11.6 Exercise 30
11.7 Summary 31

12 References 31

1 Lab #01: Booting and Linux Basics

1.1 How a PC Turns On

When you power on a PC:

1. Firmware (BIOS or UEFI) initializes hardware and tests memory.
2. It locates and loads the bootloader from the Master Boot Record (MBR) or EFI System Partition.
3. The bootloader (e.g., GRUB) then loads the operating system kernel.
4. The kernel initializes devices, mounts the root filesystem, and starts system services.

1.2 Types of Firmware

- **BIOS (Legacy Mode)**: Older firmware interface, uses MBR.
- **UEFI (Modern Firmware)**: Newer, more flexible, with GUI and larger boot partitions.

1.3 Introduction to Linux

Linux is a powerful, open-source OS with various distributions (distros):

- **Ubuntu**: User-friendly, good for beginners.
- **Linux Mint**: Similar to Windows-like interface.

1.4 Practical Orientation

1. **Check BIOS/UEFI**: On boot, press keys like `F2` or `Delete`.
2. **Booting Live USB**: Create a bootable USB with Ubuntu. Restart and select it from BIOS/UEFI.

1.5 Exercise

1. Identify if your machine uses BIOS or UEFI. 2. Boot from a Linux Live USB and explore basic commands like `ls`, `pwd`, and `cd`.

2 Lab #02: Basics of Ubuntu

2.1 Ubuntu's File System Structure

Ubuntu uses the Linux file system, which is structured as a hierarchy of directories under the root directory `/`. These directories contain important system files that generally cannot be modified unless you are the root user or use `sudo` for security and stability reasons.

- `/bin`, `/sbin`: Essential system applications (somewhat like `C:\Windows` in Windows).
- `/etc`: System-wide configuration files.
- `/home`: Contains personal directories for each user (e.g., `/home/yourusername`), similar to `C:\Users` in Windows.
- `/lib`: Library files, similar to `.dll` files on Windows.
- `/media`: Mount point for removable media (CD-ROMs, USB drives).
- `/root`: Home directory for the root user (not the same as the root directory `/`).
- `/usr`: Contains most user-installed program files (comparable to `C:\Program Files` in Windows).
- `/var/log`: Contains log files generated by various applications.

Figure 1 (conceptual): Ubuntu File System Structure At the top is `/`, the root of the hierarchy. All other directories and files branch off from it.

2.2 Terminal and the Command-Line Interface (CLI)

In Ubuntu, you have both a Graphical User Interface (GUI) and a Command-Line Interface (CLI) provided by the terminal. While most daily activities can be done via the GUI, the terminal offers powerful tools for troubleshooting, automation, and system administration tasks.

2.2.1 Why Use the Terminal?

- Some troubleshooting tasks require terminal commands.
- Batch operations on many files are often easier from the CLI.
- Learning the CLI is essential for advanced tasks, development, and administration.

When you type a command in the terminal, the shell interprets it. Commands often follow the pattern:

```
command [options] [arguments]
```

You can use `-help` with most commands to learn more about their usage and options.

2.3 Basic Commands

1. `ls`: Lists directory contents.

   ```
   ls               # Lists files in current directory
   ls -l /etc       # Lists files in /etc with details
   ```

2. `cd`: Changes the current directory.

   ```
   cd /home/user/Documents
   ```

3. `pwd`: Prints the current working directory.

   ```
   pwd
   ```

4. `adduser`: Adds a new user to the system.

   ```
   sudo adduser newusername
   ```

5. `passwd`: Changes the password of a user.

   ```
   sudo passwd newusername
   ```

6. `sudo`: Executes a command as a superuser (root).

   ```
   sudo apt update
   ```

7. `ifconfig`: Shows network configuration.
8. `iwconfig`: Shows wireless configuration.

2.4 Exercises

1. Use `pwd` to determine your current directory. 2. Use `ls` to list the contents of `/usr/bin`. Try adding different parameters like `-l` or `-a`. 3. Create a new user with `adduser` (requires superuser privileges), then use `passwd` to set their password. 4. Run `ifconfig` or `iwconfig` to view your network configuration. 5. Explore `cd` and `ls` commands to navigate through various directories and understand the file system structure.

3 Lab #03: Commands and Syntax

3.1 Overview

In Linux, commands are requests to the operating system to perform a particular function. Understanding command syntax, wildcards, case sensitivity, and auto-completion will enhance your efficiency and accuracy in using the terminal.

3.2 Command Syntax

The general syntax for entering commands in Linux is:

```
command -option(s) argument(s)
```

- **command**: The action to perform (e.g., `ls`).
- **option(s)**: Modify the command's behavior. For example, `ls -r` lists directory contents in reverse alphabetical order.
- **argument(s)**: The target of the command, such as files, directories, or devices. For example, `ls a*` lists all files starting with `a`.

Always remember to put spaces between the command, options, and arguments.

3.3 The Asterisk (*) Wildcard

The `*` (asterisk) is a wildcard that can match any set of characters:

- `ls *`: List all files in the current directory.
- `ls ab*`: List all files/folders starting with `ab`.

3.4 Case Sensitivity

Linux commands are case-sensitive, and standard commands are usually in lowercase. For example, `ls` works, but `LS` does not.

3.5 Auto-Completion

Pressing the `TAB` key after typing a partial command or filename attempts to auto-complete it:

- Type `f` + `TAB` to list all commands starting with `f`.
- Type `cd /home/a` + `TAB` to auto-complete a long directory name if unique.

3.6 Practicing Common Commands

Some commonly used commands:

ls List directory contents.

cd Change directory.

mkdir Create a new directory.

rmdir Remove an empty directory.

cat View or create text files, or combine files.

cp Copy files and directories.

mv Move or rename files and directories.

rm Remove files or directories.

Examples:

```
ls              # Lists files in current directory
cd /            # Change to the root directory
mkdir images    # Create a directory named images
rmdir images    # Remove the images directory (if empty)
cat file.txt    # View the contents of file.txt
```

3.6.1 Working with Directories and Files

`cd /`, `cd ..`, and `cd` are used to navigate to the root, parent directory, and home directory respectively.

3.7 Redirection

Redirection captures output from a command and sends it to a file or input of another command:

- > Redirect output to a file (overwrite).
- » Append output to a file.
- < Take input from a file instead of the keyboard.

Examples:

```
ls > newfile          # Redirect output of ls to newfile
cat < newfile         # Use newfile as input to cat, displaying its content
cat > hello.txt       # Create hello.txt and write to it, end with Ctrl+D
cat >> hello.txt      # Append additional text to hello.txt
cat file1 file2 > combined.txt  # Combine file1 and file2 into combined.t
```

For directories:

```
mkdir images          # Create directory images
mkdir -p images/pics  # Create directory structure, creating parents if n
rmdir images          # Remove images directory if empty
rm -r images          # Remove images directory and its contents recursiv
```

3.8 Exercise 1

Implement the directory tree (as shown in class or lab instructions) using the commands discussed. Use `mkdir`, `cd`, and `ls` to navigate, create directories, and verify their structure. Experiment with `cat`, `rm`, `cp`, and `mv` to manipulate files. Finally, use redirection to merge file contents and explore how input/output redirection works in different scenarios.

Good luck!

4 Lab #04: Advanced Linux Features

This lab covers various advanced Linux features including using a text editor (nano), creating links, viewing large outputs with `more` and `less`, searching for files, managing file and directory permissions, handling file ownership, and compiling programs using the GNU Compiler Collection (GCC).

4.1 NANO Editor

Nano is a user-friendly text editor for Linux systems. To start nano:

```
nano myfile
```

If `myfile` does not exist, it will be created.

Common shortcuts in nano:

- `Ctrl+X`: Exit
- `Ctrl+O`: Save
- `Ctrl+W`: Search
- `Ctrl+K`: Cut
- `Ctrl+U`: Paste
- `Ctrl+C`: Show cursor position

4.2 Links

A link in Linux is a pointer to a file or directory. Links allow multiple filenames to reference the same file or directory. There are two types of links:

4.2.1 Hard Links

- Have the same inode as the original file.
- Persist even if the original file is moved or deleted.
- Cannot be created across different filesystems.
- Cannot be created for directories.

To create a hard link:

```
ln original.txt hardlink.txt
```

4.2.2 Soft Links (Symbolic Links)

- Similar to shortcuts in Windows.
- Store the path to the target file rather than the file's contents.
- Can cross filesystem boundaries.
- Become invalid ("dangling") if the original file is deleted or moved.
- Can be created for directories.

To create a soft link:

```
ln -s original.txt softlink.txt
```

4.3 More and Less Commands

When output is too large for one screen, use `more` or `less` to view it page-by-page or line-by-line.

```
ls /etc -lh | more
```

`more`: Scroll forward with `Space`, quit with `q`.

```
ls /etc -lh | less
```

`less`: Scroll up/down with arrow keys, quit with `q`.

4.4 Searching Files

To search for files:

4.4.1 find Command

```
find / -name "*.pdf"      # Find all pdf files in root directory
find . -name "prog.c"     # Find prog.c in current directory
```

4.4.2 locate Command

`locate` uses a database to quickly find files by name:

```
locate "*.pdf"
locate test4
```

Ensure the database is updated using:

```
sudo updatedb
```

4.5 File Permissions

Every file and directory has permissions for three categories: Owner (u), Group (g), and Others (o). Each category has three types of permissions: read (r), write (w), and execute (x).

Permissions are often represented numerically:

- r = 4
- w = 2
- x = 1

Summing these gives a digit for each category. For example, `chmod 761 file` means:

- Owner: rwx = 7
- Group: rw- = 6
- Others: –x = 1

View file permissions with:

```
ls -l
```

4.5.1 Changing File Permissions with chmod

`chmod` changes the file mode bits.

4.5.1.1 Absolute (Numeric) Mode

Use numeric codes:

```
chmod 764 file.txt
```

4.5.1.2 Symbolic Mode

Use characters for owners and permissions:

- u = user/owner
- g = group
- o = others
- a = all

```
chmod u+r file.txt      # Add read permission for user
chmod g-w file.txt      # Remove write permission for group
chmod o+x file.txt      # Add execute permission for others
chmod a+rwx file.txt    # Give all permissions to everyone
```

4.6 Directory Permissions

Directory permissions work similarly. Execute permission (x) allows entering the directory, read (r) allows listing its contents, and write (w) allows creating/removing files in it.

```
chmod u+x directory     # User can now enter the directory
chmod g+wrx directory   # Group gets all permissions
chmod o-rwx directory   # Others lose all permissions
```

4.7 File or Directory Ownership

`chown` changes the owner of a file/directory:

```
sudo chown username file.txt
sudo chown username:group file.txt
```

`chgrp` changes the group ownership:

```
sudo chgrp groupname file.txt
```

4.8 GNU Compiler Collection (GCC)

Use `gcc` for C and `g++` for C++:

```
gcc prog.c -o prog
./prog

g++ progcpp.cpp -o progcpp
./progcpp
```

4.9 Executing Linux Commands from a C Program

Use `system()` from `stdlib.h` to run shell commands in C:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("Running ls -l:\n");
    system("ls -l");
    return 0;
}
```

Compile and run:

```
gcc command.c -o command
./command
```

4.10 Exercise

1. Practice creating and editing files with nano. 2. Create hard and soft links and verify their behavior after renaming or deleting the original files. 3. Use `more` and `less` to view the contents of large directories like `/etc`. 4. Search for specific file types using `find` and `locate`. 5. Change file permissions using both numeric and symbolic modes. 6. Modify ownership of files and directories using `chown` and `chgrp`. 7. Compile and run a simple C and C++ program using `gcc` and `g++`. 8. Write a C program that executes a Linux command using `system()`.

Good Luck!

5 Lab #06: Process Management

5.1 Introduction to Processes

A process is an instance of a program running on the operating system. Each process has:

- A unique Process ID (PID)
- A Parent Process ID (PPID)
- Associated states and resources

When a program is loaded for execution, the operating system creates a process structure. The original ancestor process in a Linux system is the `init` process (PID 1). All other processes are descendants of this process, forming a hierarchy. You can explore this hierarchy using:

```
pstree
```

5.2 Viewing Processes with `ps`

The `ps` command displays information about running processes. For example:

```
ps au
```

Common columns to note:

- **USER**: The owner of the process
- **PID**: The process ID
- **CPU & MEM**: CPU and memory usage
- **VSZ**: Virtual memory size
- **RSS**: Resident set size (physical memory)
- **TTY**: The controlling terminal associated with the process
- **STAT**: The current state of the process

- **START**: When the process started
- **TIME**: The amount of CPU time used
- **COMMAND**: The command or program executed by the process

5.2.1 Process States (STAT Codes)

Some common process states:

- **D**: Uninterruptible sleep
- **R**: Running or runnable (on run queue)
- **S**: Interruptible sleep (waiting for an event)
- **T**: Stopped (e.g., by a signal)
- **Z**: Zombie (terminated but not reaped by parent)
- **<**: High-priority
- **N**: Low-priority
- **s**: Session leader (process with child processes)
- **l**: Multi-threaded process
- **+**: Process is in the foreground group

5.3 Key System Calls

- `getpid()`: Returns the PID of the current process.
- `getppid()`: Returns the PPID (the parent's PID).
- `fork()`: Creates a new process (child), returning 0 to the child and child's PID to the parent.
- `exec()`: Replaces the current process image with a new program (there are multiple variants like `execl`, `execv`, `execvp`, etc.).
- `exit()`: Terminates the calling process and returns a status to its parent.
- `abort()`: Abnormally terminates the process by sending a SIGABRT signal.

5.4 Forking and Process Hierarchy

The `fork()` system call splits one process into two: a parent and a child. After a fork:

- The child has a unique PID but inherits most attributes from the parent.
- The return value of `fork()` in the parent is the child's PID, while in the child it is 0.
- Both parent and child continue execution from the point after `fork()`.

Example:

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        printf("Child PID: %d, PPID: %d\n", getpid(), getppid());
    } else {
        // Parent process
        printf("Parent PID: %d, Child PID: %d\n", getpid(), pid);
    }
    return 0;
}
```

5.5 Process Creation, States, and Lifecycle

Each process goes through creation, running, possible waiting (non-running), and eventual termination:

1. **Creation**: Occurs via `fork()`.
2. **Running**: The process is executing instructions.
3. **Non-Running**: Process is waiting (sleeping) for I/O or event.
4. **Termination**: Via `exit()`, `abort()`, or receiving a terminating signal.

When a process terminates but the parent has not yet read its exit status, the terminated process becomes a **Zombie** (Z state). Once the parent calls `wait()` or `waitpid()`, the zombie is cleaned up.

If a parent dies before its child, the child becomes an orphan and is adopted by `init` (PID 1).

5.6 Replacing Process Image with exec()

`exec()` family of calls load and run a new program, replacing the current process image. After a successful `exec()`, the original code of the process is no longer running:

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    char *args[] = {"/usr/bin/ls", "-l", NULL};
    execv(args[0], args);
    // If execv is successful, code below won't execute.
    printf("This will not print if exec succeeds.\n");
    return 0;
}
```

5.7 Delaying and Terminating Processes

- `sleep(seconds)`: Suspends execution for given seconds.
- `exit(status)`: Exits the process with the given status code.
- `abort()`: Immediately terminates the process with SIGABRT.

5.8 Exercises

Exercise 1: Basic PID and PPID
Using a simple program:

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Process PID: %d\tPPID: %d\n", getpid(), getppid());
    return 0;
}
```

1. Compile and run this program. 2. Use `ps au` and `top` commands to view the running process and note the PID, PPID, and other fields.
Exercise 2: Fork and Process Hierarchy
Modify the code:

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int i;
    printf("Process PID %d \t PPID %d \n", getpid(), getppid());
    for (i = 0; i < 1; ++i) {
        if (fork() == 0)
            printf("Process PID %d \t PPID %d \n", getpid(), getppid());
    }
    return 0;
}
```

1. How many processes are created for `i<1`? 2. Increase the loop count to `i<2`, then `i<3`. Count the processes using `pstree` or `ps`. 3. Draw the process hierarchy tree for each case. 4. Explain why calling `fork()` multiple times results in an exponential number of processes.

Exercise 3: Using exec()
Write a program that forks a child process. In the child, call `execv()` to run `ls`. The parent should wait and then print a message after the child finishes.

Exercise 4: Process States and Waiting
Use `sleep()` in a forked process:

1. Create a parent and child process.
2. Let the parent `sleep(120)` and the child immediately `exit(0)`.
3. In another terminal, use `ps` and `pstree` to observe the states of the processes. Note any *Zombie* processes.

Exercise 5: Counting Forked Processes
Write a code that calls `fork()` a certain number of times to achieve exactly 6 processes in total (including the original). Verify using `pstree`.

Exercise 6: Experiment with abort and exit
Write a program that uses `atexit()` to register a few functions. Call `exit(0)` at the end and observe the order of execution of registered functions. Then try calling `abort()` inside one of the registered functions and observe what happens.

5.9 Summary

In this lab, you learned:

- How to view and understand processes, their IDs, and their hierarchy.
- The various process states and how to use `ps`, `top`, and `pstree`.
- System calls like `fork()`, `exec()`, `getpid()`, `getppid()`, `exit()`, and `abort()`.
- How processes terminate and what happens when a parent or child dies first.

Explore these concepts further to fully understand process management in Linux.

6 Lab #8: Input/Output

6.1 Standard I/O Streams and File Descriptors

In Linux, each process has three standard file streams open at all times:

- `stdin` (0) – standard input (usually keyboard)
- `stdout` (1) – standard output (usually terminal screen)

- `stderr` (2) – standard error (usually terminal screen)

Any additional opened files receive higher file descriptors (3, 4, ...). Input/Output (I/O) in Linux can be done at different levels. Shell commands like `echo` can redirect output easily:

```
echo "Hello" > hello.txt
```

This writes "Hello" into `hello.txt`. You can even write directly to a process's file descriptor by using:

```
echo "Hello" > /proc/X/fd/1
```

(replace X with the process ID obtained from `ps`), which sends "Hello" to that process's standard output.

6.2 Opening and Closing Files

To work directly with files at a lower system call level, use:

- `open(path, flags, mode)`: Opens/creates a file. Examples:
  - `O_WRONLY` for write-only
  - `O_CREAT` to create if it doesn't exist
  - `0666` sets the file's permissions (read/write for everyone).
- `close(fd)`: Closes the file descriptor `fd`.

6.3 Reading and Writing Data

- `write(fd, buffer, size)`: Writes `size` bytes from `buffer` to `fd`.
- `read(fd, buffer, size)`: Reads up to `size` bytes into `buffer` from `fd`.

6.4 Examples

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main() {
    int fd = open("file.txt", O_WRONLY | O_CREAT, 0666);
    if (fd == -1) {
        perror("open");
        return 1;
    }
    write(fd, "Hello World\n", 12);
    close(fd);
    return 0;
}
```

Reading from a file:

```
int fd = open("file.txt", O_RDONLY);
char buffer[200];
ssize_t bytesRead = read(fd, buffer, sizeof(buffer)-1);
if (bytesRead > 0) {
    buffer[bytesRead] = '\0';
    printf("Contents: %s\n", buffer);
}
close(fd);
```

6.5 Exercise

1. Use `echo "Hello"` to write into `hello.txt`, then open and read it back with a C program. 2. Create a file using `open()` and write user input to it using `write()`. Check the file size and contents. 3. Experiment with `O_APPEND` and different permissions (e.g., `0666`) to see their effects.

7 Lab #09: Pipes for IPC

7.1 Introduction to Pipes

Pipes provide a unidirectional communication channel between two processes. They connect the standard output of one process directly to the standard input of another, enabling inter-process communication (IPC).

Conceptually:

- A pipe is a buffer in memory.
- One process writes to the pipe, while the other reads from it.
- The reader cannot read data before the writer writes it.

On the shell, the '|' character implements a pipe. For example:

```
pstree | less
```

Here, the output of `pstree` becomes the input of `less`, allowing you to scroll through it.

7.2 Creating Pipes in C

Use the `pipe()` system call:

```
int pfd[2];
pipe(pfd);
```

This creates a pipe with two file descriptors:

- `pfd[0]`: Read end
- `pfd[1]`: Write end

7.3 Fork and Pipes

After calling `fork()`, both parent and child processes have access to the pipe. Typically, one process closes the unused end and either only reads or only writes.

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main() {
    int pfd[2];
    char buffer[50];
    pipe(pfd);

    if (fork() == 0) {
        // Child: read only
        close(pfd[1]);
        read(pfd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        close(pfd[0]);
    } else {
        // Parent: write only
        close(pfd[0]);
        char *msg = "Hello from Parent!";
        write(pfd[1], msg, strlen(msg)+1);
        close(pfd[1]);
    }
    return 0;
}
```

7.4 Using dup2() for Redirection

By using `dup2()`, you can redirect standard input or output to the pipe ends, emulating the shell pipeline behavior programmatically:

```
dup2(pfd[0], 0); // Replace stdin with pipe read end
dup2(pfd[1], 1); // Replace stdout with pipe write end
```

7.5 Exercise

1. Modify the example to send multiple messages from the parent and read them in the child. 2. Use `dup2()` to connect `ls` and `wc` as if running ‘ls | wc‘.

This demonstrates how pipes facilitate IPC by connecting the output of one program directly into the input of another.

8 Lab #10: IPC - Signals

8.1 Introduction to Signals

Signals are an inter-process communication (IPC) mechanism used to notify a process about events such as:

- `SIGINT`: Interrupt (e.g. Ctrl+C)
- `SIGTERM`: Request process termination
- `SIGKILL`: Forceful termination (cannot be handled or ignored)
- `SIGUSR1`, `SIGUSR2`: User-defined signals for custom purposes

They are essentially asynchronous event notifications sent to a process.

8.2 Using the `kill` Command

The `kill` command sends signals to processes:

```
kill -s SIGTERM <PID>   # Send a SIGTERM
kill -9 <PID>           # Send SIGKILL (force terminate)
kill -l                 # List available signals
```

For example, find your shell's PID using `ps`, then:

```
kill -TERM <PID>
```

This requests your shell to terminate gracefully.

8.3 Using the `kill()` System Call

The `kill()` function sends signals programmatically:

```
#include <signal.h>
#include <unistd.h>

int main() {
    kill(getpid(), SIGTERM); // Send SIGTERM to itself
    return 0;
}
```

Parameters:

- First argument: Signal number (e.g., `SIGTERM`)
- Second argument: PID of the target process

8.4 Signal Handling with `signal()`

A process can customize how it responds to signals using `signal()`:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void sigHandler(int sigNum) {
    printf("Signal %d received!\n", sigNum);
}

int main() {
    // Handle Ctrl+C (SIGINT)
    signal(SIGINT, sigHandler);
    while(1) {
        printf("Running...\n");
        sleep(1);
    }
    return 0;
}
```

Instead of the default behavior, this program prints a message when it receives `SIGINT`.

8.5 Exercises

1. Run the signal handling program and press Ctrl+C. Observe the output. 2. Change the handler to `SIGUSR1` and send the signal using:

```
kill -SIGUSR1 <PID>
```

3. Create a parent process that sends `SIGTERM` to its child process after a delay. Verify termination using `ps`.

By experimenting with these examples, you will understand how signals serve as a lightweight IPC mechanism for event notification and process control.

9 Lab #11: Shell Scripting

9.1 Introduction

A shell script is a file containing a series of commands that the shell reads and executes. Unlike compiled programs, shell scripts are interpreted line by line. This makes them quick to write and test, perfect for automation of routine tasks.

9.2 Shell Basics

- **Shbang Line**: The first line `#!/bin/bash` specifies which shell will interpret the script.

- **Comments**: Start with `#` and extend to the end of the line.
- **Variables**: Assign with `name="value"` and reference with `$name`.
- **Shell Keywords**: Common keywords include `echo`, `read`, `if`, `fi`, `case`, `esac`, `for`, `while`, `do`, `done`, `until`, `break`, `continue`, `exit`.

9.3 Wildcards and Redirection

- **Wildcards**: Characters like `*`, `?`, and `[ ]` help match filenames. For example, `ls *.c` lists all C source files.
- **Redirection**: `>`, `<`, and `|` redirect input/output. For example, `echo "Hello" > out.txt` writes "Hello" to `out.txt`.

9.4 Examples of Arithmetic and Loops

Arithmetic Operations:

```
#!/bin/bash
echo "Enter a:"
read a
echo "Enter b:"
read b
echo "Sum:" $((a+b))
echo "Difference:" $((a-b))
echo "Product:" $((a*b))
echo "Quotient:" $((a/b))
```

For Loop: Even/Odd Check:

```
#!/bin/bash
nums="1 2 3 4 5 6 7 8"
for n in $nums
do
  q=$((n % 2))
  if [ $q -eq 0 ]; then
    echo "$n is even"
  else
    echo "$n is odd"
  fi
done
```

While Loop: Counting:

```
#!/bin/bash
a=1
while [ $a -lt 11 ]
do
  echo "$a"
  a=$((a+1))
done
```

9.5 Control Structures

If Statement:

```
#!/bin/bash
if [ $1 -gt 10 ]; then
  echo "Greater than 10"
else
  echo "10 or less"
fi
```

Case Statement:

```
#!/bin/bash
echo "Enter your choice:"
read choice
case $choice in
  1) echo "Option 1";;
  2) echo "Option 2";;
  *) echo "Invalid";;
esac
```

9.6 Functions in Shell Scripts

Functions group a set of commands:

```
#!/bin/bash
add() {
  c=$(( $1 + $2 ))
  echo "addition = $c"
}
add 5 10
```

9.7 Exercise

Task: Write a shell script that: 1. Prompts the user for a `.c` filename. 2. Copies content from another existing file (containing C code) into this `.c` file. 3. Presents the user with options: - (1) Compile the newly created `.c` file. - (2) Compile and run the `.c` file. - (3) Show the original file's contents. - (4) Show the contents of the current directory.

Use either an `if` statement or a `case` statement to implement the logic. Then, try implementing the same logic using a function and a case statement.

Hints: - Use `read` to get user input. - Use `cp` to copy files. - Use `gcc` for compilation and `./a.out` to run. - Use `cat` to show file contents. - Use `ls` to list directory contents.

By practicing these examples and the exercise, you will gain confidence in automating tasks and manipulating files, directories, and operations through shell scripting.

9.8 Exercise

Modify a Fibonacci program to compute terms in separate threads. Each thread calculates one Fibonacci call, and the main thread collects results.

10 Lab #12: POSIX Threads

10.1 Introduction

POSIX threads (pthreads) allow a single program to run multiple execution flows (threads) concurrently. Threads share:

- Process instructions (text segment)
- Global data (data segment)
- Open files (file descriptors)

Each thread has its own stack and registers, but all threads run within the same process space.

10.2 Thread Creation

Threads are created using `pthread_create()`:

- Splits one thread of execution into two threads.
- The `pthread_join()` call is used to wait for a thread to finish.

Example:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *print_message(void *ptr) {
    char *message = (char *)ptr;
    printf("%s\n", message);
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    pthread_create(&thread1, NULL, print_message, (void *)message1);
    pthread_create(&thread2, NULL, print_message, (void *)message2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

To compile:

```
gcc -pthread program.c -o program
```

10.3 Passing Multiple Arguments

If multiple arguments are needed, place them in a structure and pass a pointer to that structure. Example:

```
struct thread_data {
    int x, y, z;
};

void *print(void *threadArg) {
    struct thread_data *data = (struct thread_data *)threadArg;
    printf("X: %d, Y: %d, Z: %d\n", data->x, data->y, data->z);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    struct thread_data input = {1, 2, 3};
    pthread_create(&tid, NULL, print, (void *)&input);
    pthread_join(tid, NULL);
    return 0;
}
```

10.4 Thread Termination

A thread can terminate by:

- Returning from its start function.
- Calling `pthread_exit()`.
- The entire process terminating (causing all threads to end).

10.5 Synchronization with Mutex

When threads share data, simultaneous access can cause race conditions. A mutex ensures mutual exclusion:

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t lock;
int counter = 0;

void *increment(void *arg) {
    pthread_mutex_lock(&lock);
    counter++;
    printf("Counter: %d\n", counter);
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

10.6 Exercise

Consider the Fibonacci function:

```
int fib(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Modify this to compute each `fib()` call in a separate thread. Each thread will:

- Take a number `n` as input.
- If `n` > 1, create two threads to compute `fib(n-1)` and `fib(n-2)`.
- Combine results.

This exercise helps understand how to break down a recursive function into multiple threads, and also highlights the need for synchronization to avoid race conditions.

By working through these examples, you'll gain familiarity with thread creation, joining, passing arguments, thread termination, and synchronization with mutexes in POSIX threads.

11 Lab #13: Semaphores

11.1 Introduction

A semaphore is a synchronization tool used to control access to shared resources. Think of it as a traffic light for threads:

- **Binary Semaphore**: Allows only one thread at a time into the critical section (like a single-lane bridge).
- **Counting Semaphore**: Allows a specified number of threads to access the resource simultaneously (like a toll booth with multiple open lanes).

Semaphores help:

- **Prevent race conditions:** Ensuring only one thread can modify shared data at a time.
- **Avoid deadlocks:** Proper signaling so threads don't wait indefinitely.
- **Manage ordering:** One thread can wait for another to signal before proceeding.

11.2 Basic Operations

Key semaphore functions in C:

- `sem_init(&sem, pshared, value)`: Initialize the semaphore with a given starting count.
- `sem_wait(&sem)`: Decrements the semaphore. If the count is zero, the calling thread blocks.
- `sem_post(&sem)`: Increments the semaphore, potentially unblocking a waiting thread.
- `sem_destroy(&sem)`: Cleans up the semaphore when done.

11.3 Example 1: Binary Semaphore (One at a Time)

Only one thread can enter the critical section at once:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t semaphore;

void* thread_function(void* arg) {
    sem_wait(&semaphore); // Locks (count down by 1)
    printf("Thread %ld is entering the critical section.\n", (long)arg);
    sleep(1);
    printf("Thread %ld is leaving the critical section.\n", (long)arg);
    sem_post(&semaphore); // Unlocks (count up by 1)
    return NULL;
}
```

```c
int main() {
    pthread_t threads[3];
    sem_init(&semaphore, 0, 1); // Initialize with 1 (binary)

    for (long i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)i);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);
    return 0;
}
```

11.4 Example 2: Counting Semaphore (Multiple at a Time)

Allows multiple threads (e.g., 3) to access the critical section at once:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t semaphore;

void* thread_function(void* arg) {
    sem_wait(&semaphore);
    printf("Thread %ld is accessing the resource.\n", (long)arg);
    sleep(2);
    printf("Thread %ld is done using the resource.\n", (long)arg);
    sem_post(&semaphore);
    return NULL;
}

int main() {
    pthread_t threads[5];
    sem_init(&semaphore, 0, 3); // Up to 3 threads at a time

    for (long i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, thread_function, (void*)i);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    sem_destroy(&semaphore);
    return 0;
}
```

11.5 Example 3: Binary Semaphore as a Signal

One thread waits until another signals it:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t semaphore;

void* thread1_function(void* arg) {
    printf("Thread 1: Waiting for signal...\n");
    sem_wait(&semaphore); // Wait until signaled
    printf("Thread 1: Received signal, proceeding.\n");
    return NULL;
}

void* thread2_function(void* arg) {
    sleep(1); // Simulate work
    printf("Thread 2: Sending signal...\n");
    sem_post(&semaphore); // Signal thread1
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    sem_init(&semaphore, 0, 0); // Start at 0, so thread1 waits

    pthread_create(&thread1, NULL, thread1_function, NULL);
    pthread_create(&thread2, NULL, thread2_function, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    sem_destroy(&semaphore);
    return 0;
}
```

11.6 Exercise

1. Modify the binary semaphore example to use a counting semaphore, allowing 2 threads at a time. 2. Run the program and observe that no more than 2 threads enter the critical section simultaneously.

11.7 Summary

Semaphores are essential for coordinating threads and managing access to shared resources. By using `sem_wait` and `sem_post`, you ensure threads don't interfere with each other, preventing race conditions and enabling efficient concurrency.

12 References

- `https://www.gnu.org/`
- `https://guru99.com`
- `https://ubuntu.com/tutorials`