

CS 236: Operating Systems Lab

Spring 2025-26

Lab 1: Hello OS!

Instructions

- This file is part of the **lab1.tar.gz** archive, which contains multiple directories with files associated with the exercise questions listed below.
- Most questions in this lab are tutorial-style and aim to provide an introduction to tools and files related to system and process information and control.

Part 1: The OS view

(a) Tools

The following are some basic Linux tools. The first step of this lab is to get familiar with the usage and capabilities of these tools.

To know more about them, use: `man <tool>`. Start with `man man`.

- **top**

`top` provides a continuous collective view of the system and operating system state. For example, a list of all processes, the resource consumption of each process, system-level CPU usage, and so on. The system summary information displayed, order, etc., has several configurable knobs.

`top` also allows you to send signals to processes (change priority, stop, etc.).

Sample tasks: (Hint: `man top`)

- Display processes of a specific user (e.g., `labuser`)
- Add **ppid** information to the displayed information (What is **pid**? What is **ppid**?)

- **ps**

The `ps` command is used to view the processes running on a system. It provides a snapshot of the processes, along with detailed per-process information, such as process ID, CPU usage, memory usage, and command name. The `ps` command has several flags to display different types of process information. For example, executing `ps` without arguments will not show all processes on the system, but a combination of flags as input parameters will.

Sample tasks: (Hint: `man ps`)

- List all the processes (of all users) in the system
- List all processes whose **ppid** is 2
- Which process has **pid** 2?

- **iostat**

`iostat` is a command useful for monitoring and reporting CPU and device usage statistics. For example, the command reports total activity and rate of activities (read/write) to each disk/partition and can be configured to monitor continuously (after every specified interval).

Sample tasks: (Hint: **man iostat**)

- Display the average CPU utilization of your system
- Display average disk and CPU utilization every 3 seconds for 10 times
- Display the average disk utilization of a specific disk every 1 second

- **strace**

`strace` is a diagnostic and debugging tool used to monitor the interactions between processes and the operating system (Linux). The tool traces the set of functions (system calls/calls of the Application Binary Interface) and signals (events) used by a program to communicate with the operating system.

Sample task: (Hint: **man strace**)

- Display all system calls and signals made by any command (for example, display the system calls made by the **ls** command).
- Display a summary of total time taken by each system call, time taken per system call during a program execution, and the number of times a system call was invoked.

- **lsof**

`lsof` is a tool used to list open files. The tool lists details of the file and the users and processes that are using it.

Sample task: (Hint: **man lsof**)

- Display all opened files of a specific user.
- Display all open files used by your shell and their sizes in a human-readable format.

- **lsblk**

`lsblk` is a tool used to list information about all available block devices, such as hard disk drives (HDDs), solid-state drives (SSDs), flash drives, CD-ROMs, etc.

Sample task: (Hint: **man lsblk**)

- Display all device permissions (read, write, execute) and owners.

- Also, look up the following commands/tools:

pstree, lshw, lspci, lscpu, dig, netstat, df, du, watch.

(b) The `proc` file system

The `proc` file system is a mechanism provided by Linux for communication between userspace and the kernel (operating system) using the file system interface. Files in the `/proc` directory report values of several OS parameters and can also be used for configuration and (re)initialization. The `proc` file system is very well-documented in the man pages, specifically in the '`man proc`' entry.

Understand system-wide `proc` files, such as `meminfo` and `cpuinfo`, as well as process-related files, including `status`, `stat`, `limits`, and `maps`. System-related `proc` files are available in the directory `/proc`, and process-related `proc` files are available at `/proc/<process-id>/`

Exercises

1. Collect the following basic information about your machine using the `proc` file system and the tools listed above, and answer the following questions. Also, mention the tool and file you used to get the answers.
 - a. Find the Architecture, Byte Order, and Address Sizes of your CPU.
 - b. How many CPU sockets, cores, and CPU threads does the machine have?
 - c. Find the sizes of L1, L2, and L3 caches.
 - d. What is the total main memory and secondary memory of your machine, and how much of it is free?
 - e. Find the number of total, running, sleeping, stopped, and zombie processes. *[A zombie process is a stopped/terminated process that is still waiting to be cleaned up.]*
 - f. How many context switches has the system performed since bootup? A context switch is the process of storing the state of a process or thread so that it can be restored and resume execution at a later point, and then restoring a different, previously saved, state. This allows multiple processes to share a single CPU and is an essential feature of a multitasking operating system.
2. Run all programs in the subdirectory named `memory` and identify the memory usage of each program. Compare the memory usage of these programs in terms of `VmSize` & `VmRSS`. Are they the same? Should they be the same? Why or why not?
Hint: `/proc/[pid]/status` `/proc/[pid]/statm`
[Note that VmSize and VmRSS needs to be checked at multiple points of execution for a program under consideration.]
3. Run the executable `subprocesses` provided in the sub-directory `subprocess` and provide the **last three digits** of your roll number as a command-line argument. Find the number of subprocesses created by this program. Describe how you obtained the answer.
4. Run `strace` along with the binary program `empty` (file located in subdirectory `strace`). What do you think the output of `strace` indicates in this case? How many different system calls can you identify?

Next, use `strace` along with the binary program `hello` (located in the same directory).

Compare the two `strace` outputs

- Which part of the output is common, and which part has to do with the specific program?
 - List all unique system calls for each program and look up the functionality of each.
5. Run the executable **openfiles** in the subdirectory files. List the files that are opened by this program, and describe how you obtained the answer.
 6. Locate all block devices on your system, including their mount points and the file systems present on them. A mount point is a file system directory entry that allows access to a disk. A file system describes how data is organized on a disk. Describe how you obtained the answer.

(c) Object Files

An object file is a file containing binary information (object code), a sequence of hardware instructions representing a program. Object files store information about data and code, as well as information about sections (such as text and data), and information used to relocate code and data in binary form. An object file is generated by a compiler or an assembler and represents an executable file or a shared library.

ELF (Executable and Linkable Format) is a universally used file format for object files on Unix-like machines. More about ELF here: <https://wiki.osdev.org/ELF>

• **objdump**

objdump is a command in Linux used to provide information about object/executable files

Sample task: (Hint: **man objdump**)

- Use **objdump** to inspect headers and assembler contents of executable sections of the **empty** and **hello** binaries provided for the strace exercise

Sample **objdump** commands:

- Display all sections' information in the object file.
`objdump -h <object-file>`
- Display the symbol table entries of the object file.
`objdump -t <object-file>`
- Display all header information.
`objdump -x <object-file>`
- Display all the assembler contents of the executable section.
`objdump -d <object-file>`
- Display the contents of all sections.
`objdump -s <object-file>`
- Display the content of a specific section in hex representation.
`objdump -s -j <section-name> <object-file>`

Exercise

Write a C program that performs matrix multiplication (or another complex program). Compile the program to generate an object file or executable, and then use the **objdump** tool to analyze the generated file.

Modern compilers support multiple levels of optimization, which can significantly impact the generated machine code. These optimizations can be enabled using the flags `-O1`, `-O2`, and `-O3` during compilation, with `-O3` applying the most aggressive optimizations.

Example: `gcc -O3 <C-file>`

Compile your program both with and without optimization, and analyze the resulting binaries using `objdump`. Compare the outputs. Identify and explain at least one interesting optimization effect that the compiler applies at higher optimization levels.

Part 2: Booting unraveled

The goal of this part of the lab is to learn about how a computer boots, and work with a dummy operating system! The question of interest here is, when a machine is powered on, how does it load the operating system and its components? Where is the kernel stored? Which files to read and execute? etc.

The answer to this lies with the idea of loading a portion of data from disk into memory and executing the corresponding contents. The road to world peace via operating systems starts here. If we can locate this special block of data on disk and verify that the disk's contents contain the codes for world peace, we are all set. In other words, this special block serves as the entry point for the operating system to take control of the hardware and perform its functions.

When a computer starts, a special program called the Basic Input/Output System (BIOS) is loaded from a chip into the main memory. The BIOS detects connected hardware devices, resets them, tests them, etc., and also looks for the special sector (the boot sector) on available disks to load the operating system.

The BIOS reads the first sector of each disk (one by one) and determines whether it is a boot disk (a disk with an operating system). A boot disk is detected via a magic number **0xaa55**, stored as the last two bytes of the boot sector of a disk.

1. The `boot_sector1.asm` file in the `myos` directory shows a sample assembly code that is supposed to perform a specific task. The idea is that this program produces machine instructions that would be copied to the boot sector when the computer is powered on.

Convert assembly (mnemonics) code to binary using the following,

```
$ nasm boot_sector1.asm -f bin -o boot_sector1.bin
```

If you want to see exactly what is inside the binary file, the following command will help you.

```
$ od -t x1 -A n boot_sector1.bin
```

The above binary can be used to set up (copy to) the first 512 bytes (the boot sector) of a disk. Instead of writing this boot sector to a physical hard disk, we can use an emulator. QEMU is a system emulator that provides a simple and convenient method for loading and executing the boot sector directly from a binary file.

```
$ qemu-system-i386 boot_sector1.bin
```

The above command emulates a system using the provided file as the attached disk (which,

in our case, contains the first 512 bytes of interest).

Compare the outputs of the booting process using the two programs, `boot_sector1.asm` and `boot_sector2.asm`, and justify your results. Submissions should include binary files and screenshots of QEMU, along with a clear explanation.

2. Let's do something slightly more interesting. Upon boot, our custom OS should display a message.

Write a program, `hello.asm`, that prints custom text (your name?) on the screen during boot-up, for example, "OS is awesome!".

To print a character on the screen, use the following code with appropriate repetitions and changes.

```
mov ah, 0x0e      ; set tele-type mode (output to screen)
mov al, 'B'       ; one ascii character hex code in register AL
int 0x10          ; send content of register to screen via an interrupt
```

Set up `hello.bin` as the input file for QEMU to use for booting and test output (capture screenshot and save it in a file named `hello.png`.)

Submission Guidelines

- All submissions via Moodle.
- Name your submissions as: `<rollno>_lab1.tar.gz`
For example, if your roll number is `25b0371`, then the submission will be `25b0371_lab1.tar.gz`
- The tar should contain the following files in the following directory structure:

```
<rollno>_lab1/
|__part1/
|    |__exercises_1_to_7.pdf
|__part2/
|    |__boot_sector1.bin
|    |__boot_sector1.png
|    |__boot_sector2.bin
|    |__boot_sector2.png
|    |__hello.asm
|    |__hello.bin
|    |__hello.png
```
- Deadline: **January 8, 2026, 5:00 p.m.**