**CPE-2600 FALL 2024**

# Systems Programming

# Lab 9 System Calls

## Introduction

The purpose of this assignment is for you to familiarize yourself with commonly used system calls on a Portable Operating System Interface (POSIX) operating system.

Throughout the assignment, you will be asked to perform research and write some code. There is no specific deliverable for the research activities, but verbose comments will be expected in the code generated.

Work on the assignment is to be done individually. You are welcome to collaborate with others in the class, but the submitted assignment must be your work alone.

## Background and References

The Portable Operating System Interface (POSIX) is a standard maintained by IEEE and The Open Group to establish a set of interfaces to operating system services so that applications can be written and 'ported' from one operating system to another with minimal modifications. All Linux distributions and Windows (using the Windows Subsystem for Linux) as well as many other operating systems have been certified to be POSIX compliant.

There are hundreds of system calls within the POSIX standard. They can be divided into one of five (5) categories:

- Information Maintenance - Used to modify and retrieve information about the system as a whole
- Process Control - Used to create, destroy, and manipulate tasks/processes on the system
- File Management - Used to create, delete, and modify files in the file system
- Communication - Used for communication between tasks/processes and between other computers
- Device Management - Used for managing hardware devices

## Project Description

### Part 1: Information Maintenance System Calls

Information maintenance system calls provide data concerning the entire running system. This includes system name, number of CPUs, amount of memory, etc:

- clock_gettime
- uname
- gethostname
- get_nprocs
- sysconf
- getpagesize

Research each of the system calls above. Then write a program **(info.c / info)** that prints:

- The current time of day - in nanoseconds
- The system's network name
- The operating system name
- The operating system release and version
- The system's hardware type
- The number of CPUs on the system
- The total amount of physical memory ***IN BYTES***
- The total amount of free memory ***IN BYTES***

If any of these values are blank (i.e. NULL string or a NULL char *) do* **NOT*** print them out.

### Part 2: Process Control System Calls

Process control system calls manage tasks on the running system. Tasks/processes are essentially the representation of an executing program in a system. Processes might be in one of several states (active executing on the CPU, waiting for something like user input, suspended by a system administrator, etc.).

Each process is given an identifier by the operating system. Using this identifier it is possible to set and retrieve information about a process (e.g. priority, CPU affinity on a multiprocessor system, etc.) as well as manipulate processes (suspend them, wait for a time, end a process). More details on processes and process manipulation will be covered in a future assignment.

#### *Retrieving Process Information*

Linux organizes processes in the process table. To print out a list of process, Linux provides the ps command. Spend a few minutes experimenting with ps and common command line options.

There are lots of system calls available for retrieving information about processes in the system. Here are some:

- getpid
- getpriority
- sched_getscheduler
- getcpu
- getrusage
- getrlimit

Research each of the system calls and be able to describe what they are used for. Then write one program. The program **(pinfo.c / pinfo)** will: - Take a single command line parameter that is a process identifier - For that process identifier, print out - The process priority - The scheduling method - as a string of text ***NOT*** a number - If the process identifier does not exist, print an error message and exit - If the user does ***NOT*** specify a process identifier, print the information for the ***CURRENT*** executing process.

***NOTE:*** the SCHED_BATCH and SCHED_IDLE scheduling methods are Linux specific and are not supported on other operating systems. We'll return to these in a future lab. However, to get your program to compile you will need to include the following in your source file ***BEFORE*** you include any header files

```
#define _GNU_SOURCE
#include <stdio.h>
#include <sched.h>
.
.
.
```

## *Altering a Process*

There are several attributes that can be altered by a process in Linux. In this section, you'll learn two (2), the process priority and running state, specifically:

- nice
- nanosleep

Research each of the system calls above be able to describe what they are used for. Note the difference between `man nice` and `man 2 nice`. Then write a program **(pmod.c / pmod)** that:

- Modifies its own priority to ***REDUCE*** it by 10 - ***NOTE*** Make sure you specify the correct number to ***REDUCE*** the priority not increase.
- Sleeps for 1,837,272,638 nano seconds
- Prints a goodbye message and exits

While running the program, observe its behavior is top or htop. Specifically note the priority shown by these tools.

## Part 3: File Management System Calls

Using library functions provided by the C standard library (i.e. `stdio.h`) we've already worked with file input and output. These functions are all built using communication to the operating system through system calls, although abstracted through libc library functions. In this section, you'll research and write some programs that manipulate files using system calls instead of library functions.

First, locate and skim the man pages for file i/o via the system calls open, `read`, `write`, etc. Note that they operate with an integer "file descriptor" as opposed to a FILE * used by the libc counterparts. What other major differences do you observe?

We may explore file i/o with these methods later. For now we will experiment with accessing detailed file information not available through libc, namely file permissions. This information is accessible via system calls stat/fstat/lstat. Like other systems calls we have looked at, the three calls have minor differences. Select which one will work best for this program based on information in the man page.

Write a program **(finfo.c / finfo)** that displays the file information about a given file provided via the command line. The file name ***MUST*** be specified via the command line. Specifically, - The type of file - print this in a user readable way - The permissions set on the file - print these in a user readable way - The owner of the file - printing the user identifier (number) is enough - The size of the file ***IN BYTES*** - The date and time of last modification (format this for easy viewing, do ***NOT*** print out raw bytes or a large integer). - If at any point, there is an error print an error and exit.

***NOTE:*** You ***MUST*** be running within the file system created by your Linux installation ***NOT*** your Windows file system.

## Development Requirements

You are required to implement each section using it's own C source file.

- In summary, you will have the following source files and executables when finished:
  - info.c –> info
  - pinfo.c –> pinfo
  - pmod.c –> pmod
  - finfo.c –> finfo
- A repository will be created when you accept the assignment. It will have these four source files (mostly empty) and a make file.
- The assignment is available via GitHub Classroom. The link to the assignment is: https://classroom.github.com/a/_1o5PVOO

### Code Structure

Code must follow style guidelines as defined in the course material.

## Getting Started

The following files have been provided for you in your repository:

At the top of ***EACH SOURCE FILE*** include a comment block with your name, assignment name, and section number.

## Hints and Tips

Use the manual (`man`) pages for system calls to your advantage. They give a description of the system call functionality along with the required parameters.

## Testing and Debugging

### Debugging

Don't forget about using `gdb` and `valgrind` to help with debugging.

Your programs must be free of run-time errors and memory leaks.

## Deliverables

When you are ready to submit your assignment, prepare your repository:

- Make sure your name, assignment name, and section number are all files in your submission - in comment blocks of source file(s)
- Tag your repo with the tag `vFinal`
- Make sure all files and tags are committed and pushed to the main branch of your repository.

Finally, submit a URL to your repository to the associated Canvas assignment.