Written by: Dr. Farkhana Muchtar

LAB 3: BASIC LINUX SYSTEM CALLS

Introduction of System Calls

1 Overview

In this lab tutorial, we will explore the basics of Linux system calls using C and Python code. System calls are a fundamental aspect of operating systems, as they provide an interface between the user programs and the kernel. Understanding how to use system calls will help you gain a better understanding of how operating systems function.

What is a system call?
A system call is a mechanism that allows user programs to request services from the kernel. These services include creating processes, managing files, and communicating with devices.

How do system calls work?
System calls work by invoking a specific function in the kernel through a software interrupt. This interrupt triggers a context switch from user mode to kernel mode, where the kernel can safely execute the requested operation.

2 Installation

In this lab exercise, we will explore Linux system calls using two different programming languages: C and Python. Students have the flexibility to choose either language for their lab assessments and projects.

Task 1: Update and Upgrade Ubuntu Packages.

- Updating package/software status

  ```
  sudo apt update
  ```

- List down upgradable package/software

  ```
  apt list --upgradable
  ```

- Upgrade those packages/software

  ```
  sudo apt upgrade
  ```

Task 2: Install Build-Essential Package

The **build-essential** package in Ubuntu is a metapackage that contains essential tools and libraries required for compiling and building software from source code. Installing build-essential ensures that you have the necessary tools for basic software development and for compiling software on your system. This `build-essential` metapackage is prerequisite for next exercise, which is writing, compiling and running C.

To install `build-essential` metapackage, run this command:

```
sudo apt install build-essential
```

Task 3: Install Python3

By default, Ubuntu comes with Python3 pre-installed. However, to ensure that you have the latest version, run the following command:

```
sudo apt install python3
```

To verify the installation and check the version of Python3, run:

```
python3 --version
```

Task 4: Set Python3 as the Default Python

To set Python3 as the default Python interpreter, you can install the `python-is-python3` package, which will create a symlink from `/usr/bin/python` to `/usr/bin/python3`:

```
sudo apt install python-is-python3
```

Alternatively, you can manually set up the alternatives system for Python. Run the following commands to add both python2 (if installed) and python3 to the alternatives system:

```
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 1
```

Now, when you run `python --version`, it should show the Python3 version installed on your system.

Task 5: Install pip for Python3

`pip` is the package installer for Python. To install pip for Python3, run the following command:

```
sudo apt install python3-pip
```

To verify the installation and check the version of `pip`, run:

```
pip --version
```

With these steps, you have successfully installed Python3, set it as the default Python interpreter, and installed pip for Python3 on your Ubuntu system. You can now start using Python3 and pip to install and manage Python packages.

Written by: Dr. Farkhana Muchtar

3 Writing System Calls Programs

In this lab tutorial, we will explore Linux system calls related to

1. Process control
2. File manipulation
3. Device manipulation
4. Information maintenance
5. Communication
6. Protection

Each exercise will include both C and Python code examples, along with an explanation of the concepts being covered. It is advisable to implement all the exercises inside `/home/<username>/workspace` directory.

3.1 Exercise 1: Process Control - fork()

Objective: Learn how to create a new process using the `fork()` system call.

Explanation: In this exercise, we use the `fork()` system call to create a new child process. The child process prints its ***process ID (PID)***, while the parent process waits for the child process to complete.

3.1.1 fork() in C Code

Step 1: Create a new text file named test.txt with some content.

```
echo “Insert any sentence here.” > test.txt
```

Step 2: Write a new C code called `fork.c`, using Nano.

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

int main() {
    int fd;
    char buffer[32];

    // Open a file for reading
    fd = open("test.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Read from the file
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read == -1) {
        perror("read");
        close(fd);
        return 1;
```

```
    }

    buffer[bytes_read] = '\0';
    printf("Content of test.txt: %s\n", buffer);

    // Close the file
    close(fd);

    return 0;
}
```

Step 3: Compile the code using gcc

```
gcc fork.c -o fork
```

Step 4: Run the compiled program

```
./fork
```

3.1.2 fork() in Python Code

Step 1: Write a new Python code, named `fork.py`, using Nano.

```
import os

pid = os.fork()

if pid == 0:
    print(f"Child process with PID {os.getpid()}")
elif pid > 0:
    print(f"Parent process with PID {os.getpid()}")
    os.wait()
else:
    print("Fork failed")
    exit(1)
```

Step 2: Run a Python script using python interpreter

```
python fork.py
```

Written by: Dr. Farkhana Muchtar

3.2 Exercise 2: File Manipulation - open(), read(), and write()

Objective: Learn how to use the `open()`, `read()`, and `write()` system calls to manipulate files.

Explanation:
In this exercise, we use the `open()` system call to open a file for reading. We then read the contents of the file using the `read()` system call and display the contents. Finally, we close the file using the `close()` system call.

3.2.1 open(), read() and close() in C Code

Step 1: Create a new text file, named `test.txt` with some content.

```
echo “Insert any sentence here.” > test.txt
```

Step 2: Run `ls -l` command to show `test.txt` file mode.

```
ls -l test.txt
```

Step 3: Write a new C code, name `file.c` using Nano.

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

int main() {
    int fd;
    char buffer[32];

    // Open a file for reading
    fd = open("test.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Read from the file
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read == -1) {
        perror("read");
        close(fd);
        return 1;
    }

    buffer[bytes_read] = '\0';
    printf("Content of test.txt: %s\n", buffer);

    // Close the file
    close(fd);

    return 0;
}
```

Step 4: Compile the code.

```
gcc file.c -o file
```

Step 5: Run the compiled program.

```
./file
```

Step 6: Run `ls -l` command again to compare `test.txt` file mode before running the compiled program.

3.2.2 open(), read() and close() in Python Code

Step 1: Remove `test.txt` from previous exercise.

```
sudo rm test.txt
```

Step 2: Create a new `test.txt` for this exercise.

```
echo “Insert any sentence here.” > test.txt
```

Step 3: Run `ls -l` command to show test.txt file mode.

```
ls -l test.txt
```

Step 4: Write a new Python code, named `file.py`.

```
import os

try:
    # Open a file for reading
    fd = os.open("test.txt", os.O_RDONLY)

    # Read from the file
    bytes_read = os.read(fd, 32)
    content = bytes_read.decode()

    print(f"Content of test.txt: {content}")

    # Close the file
    os.close(fd)
except OSError as e:
    print(f"Error: {e}")
```

Step 5: Run a python script using python interpreter

```
python file.py
```

Step 6: Run `ls -l` command again to compare test.txt file mode before running the compiled program.

Written by: Dr. Farkhana Muchtar

3.3 Exercise 3: Device Manipulation - ioctl()

Objective: Learn how to use the `ioctl()` system call to manipulate devices.

Explanation:
In this exercise, we use the `ioctl()` system call to perform a device-specific operation on a device file. We first open the device file for reading and writing and then call `ioctl()` with the appropriate command and data. Finally, we close the device file. In this exercise, we use the `ioctl` system call with the TIOCGWINSZ request to get the terminal window size.

3.3.1 ioctl() in C Code

Step 1: Create a new C code, named `ioctl.c`.

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <termios.h>

int main() {
    int fd_tty;
    struct winsize ws;

    // Open /dev/tty for reading
    fd_tty = open("/dev/tty", O_RDONLY);
    if (fd_tty == -1) {
        perror("open /dev/tty");
        return 1;
    }

    // Get the terminal window size
    if (ioctl(fd_tty, TIOCGWINSZ, &ws) == -1) {
        perror("ioctl TIOCGWINSZ");
        close(fd_tty);
        return 1;
    }

    // Print the terminal window size
    printf("Terminal size: %d rows, %d columns\n", ws.ws_row, ws.ws_col);

    // Close the device file
    close(fd_tty);

    return 0;
}
```

Step 2: Compile the code using gcc.

```
gcc ioctl.c -o ioctl
```

Step 3: Run a compiled program

```
./ioctl
```

Written by: Dr. Farkhana Muchtar

3.3.2 ioctl() in Python Code

Step 1: Write a new C code, named `ioctl.py` using Nano.

```
import os
import fcntl
import termios
import struct

try:
    # Open /dev/tty for reading
    fd_tty = os.open("/dev/tty", os.O_RDONLY)

    # Get the terminal window size
    buf = fcntl.ioctl(fd_tty, termios.TIOCGWINSZ, b"\x00" * 8)
    rows, cols, _, _ = struct.unpack("HHHH", buf)

    # Print the terminal window size
    print(f"Terminal size: {rows} rows, {cols} columns")

    # Close the device file
    os.close(fd_tty)

except OSError as e:
    print(f"Error: {e}")
```

Step 2: Run a Python script using Python interpreter

```
python ioctl.py
```

3.4 Exercise 4: Information Maintenance - stat()

Objective: Learn how to use the `stat()` system call to obtain file status information.

Explanation: In this exercise, we use the `stat()` system call to obtain information about a file, such as its size, user ID, and group ID. This information can be useful for checking file permissions and ownership, as well as determining the amount of disk space used by the file.

3.4.1 stat() in C Code

Step 1: Write a new C code, named `stat.c` using Nano.

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    struct stat file_info;

    if (stat("test.txt", &file_info) == -1) {
        perror("stat");
        return 1;
    }

    printf("Size of test.txt: %ld bytes\n", file_info.st_size);
    printf("User ID of test.txt: %d\n", file_info.st_uid);
    printf("Group ID of test.txt: %d\n", file_info.st_gid);

    return 0;
}
```

Step 2: Compile the C code

```
gcc stat.c -o stat
```

Step 3: Run a compiled program

```
./stat
```

Written by: Dr. Farkhana Muchtar

3.4.2 stat() in Python Code

Step 1: Write a new Python code, named `stat.py` using Nano.

```
import os
import stat

try:
    # Obtain file status information
    file_info = os.stat("test.txt")

    print(f"Size of test.txt: {file_info.st_size} bytes")
    print(f"User ID of test.txt: {file_info.st_uid}")
    print(f"Group ID of test.txt: {file_info.st_gid}")
except OSError as e:
    print(f"Error: {e}")
```

Step 2: Run a Python script using python interpreter

```
python stat.py
```

Written by: Dr. Farkhana Muchtar

3.5 Exercise 5: Communication - pipe()

Objective: Learn how to use the `pipe()` system call to create a communication channel between two processes.

Explanation: In this exercise, we use the `pipe()` system call to create a communication channel between a parent and a child process. The parent process writes a message to the pipe, and the child process reads the message from the pipe.

3.5.1 pipe() in C Code

Step 1: Write a new C code, named pipe.c using Nano.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    pid_t pid;
    char buf[128];

    // Create a pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }

    // Fork a new process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // Child process
        close(pipefd[0]); // Close unused read end

        // Write a message to the pipe
        const char *msg = "Hello from the child process!";
        write(pipefd[1], msg, strlen(msg));

        close(pipefd[1]); // Close the write end
        exit(EXIT_SUCCESS);
    } else {
        // Parent process
        close(pipefd[1]); // Close unused write end

        // Read the message from the pipe
```

```
        ssize_t n = read(pipefd[0], buf, sizeof(buf) - 1);
        if (n > 0) {
            buf[n] = '\0';
            printf("Received message: %s\n", buf);
        }

        close(pipefd[0]); // Close the read end
        wait(NULL); // Wait for the child process to exit
    }

    return 0;
}
```

Step 2: Compile the C code.

```
gcc pipe.c -o pipe
```

Step 3: Run a compiled program.

```
./pipe
```

3.5.2 pipe() in Python Code

Step 1: Write a Python Code, name `pipe.py` using Nano.

```
import os

def main():
    # Create a pipe
    read_fd, write_fd = os.pipe()

    # Fork a new process
    pid = os.fork()

    if pid == 0:
        # Child process
        os.close(read_fd)  # Close unused read end

        # Write a message to the pipe
        msg = b"Hello from the child process!"
        os.write(write_fd, msg)

        os.close(write_fd)  # Close the write end
        os._exit(0)
    else:
        # Parent process
        os.close(write_fd)  # Close unused write end

        # Read the message from the pipe
        buf = os.read(read_fd, 128)
        print(f"Received message: {buf.decode()}")

        os.close(read_fd)  # Close the read end
```

```
        os.wait()  # Wait for the child process to exit


if __name__ == "__main__":
    main()
```

Step 2: Run a Python script using Python interpreter

```
python pipe.py
```

Written by: Dr. Farkhana Muchtar

3.6 Exercise 6: Protection - chmod()

Objective: Learn how to use the `chmod()` system call to change the permissions of a file.

Explanation: In this exercise, we use the `chmod()` system call to change the permissions of a file. The example sets the file permissions to 644, which allows the owner to read and write the file, and others to read the file. This can be useful for managing access to files in a shared environment.

3.6.1 chmod() in C Code

Step 1: Create a new empty text file, named test.txt

```
touch test.txt
```

Step 2: Run command ls –l to see the file access mode.

```
ls -l test.txt
```

Step 3: Write a new C code, named chmod.c using Nano.

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    if (chmod("test.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1) {
        perror("chmod");
        return 1;
    }

    printf("Changed permissions of test.txt to 644 (rw-r--r--)\n");

    return 0;
}
```

Step 4: Compile the C code.

```
gcc chmod.c -o chmod
```

Step 5: Run a compiled program.

```
./chmod
```

Step 6: Run command ls –l again for a comparison.

```
ls -l test.txt
```

3.6.2 chmod() in Python Code

Step 1: Remove the existing test.txt

```
sudo rm test.txt
```

Step 2: Recreate the test.txt

```
touch test.txt
```

Step 3: Run command ls –l to see the file access mode.

```
ls -l test.txt
```

Step 4: Write a new Python code, named chmod.py using Nano.

```
import os

try:
    # Change the permissions of a file
    os.chmod("test.txt", 0o644)
    print("Changed permissions of test.txt to 644 (rw-r--r--)")
except OSError as e:
    print(f"Error: {e}")
```

Step 4: Run Python script using Python interpreter

```
python chmod.py
```

Step 5: Run a command ls –l again for comparison.

```
ls -l test.txt
```