SCC.231 (Week 2) - Introduction to Linux processes

Week 2: OS processes

Learning Objectives

This lab assignment introduces you to the concept of processes using the Linux OS. We will use Python to explore the process API of the Linux kernel. You will learn how to create, manage, and terminate processes using system calls and signals. By the end of this assignment, you will have a solid understanding of how processes work and how to manipulate them programmatically.

- ☐ Understand the concept of processes in an operating system.
- ☐ Use CLI tools to view and manage processes.
- ☐ Create and manage processes using Python.
- ☐ Handle signals in Python.
- ☐ Study the performance of parallel processing using multiple child processes.

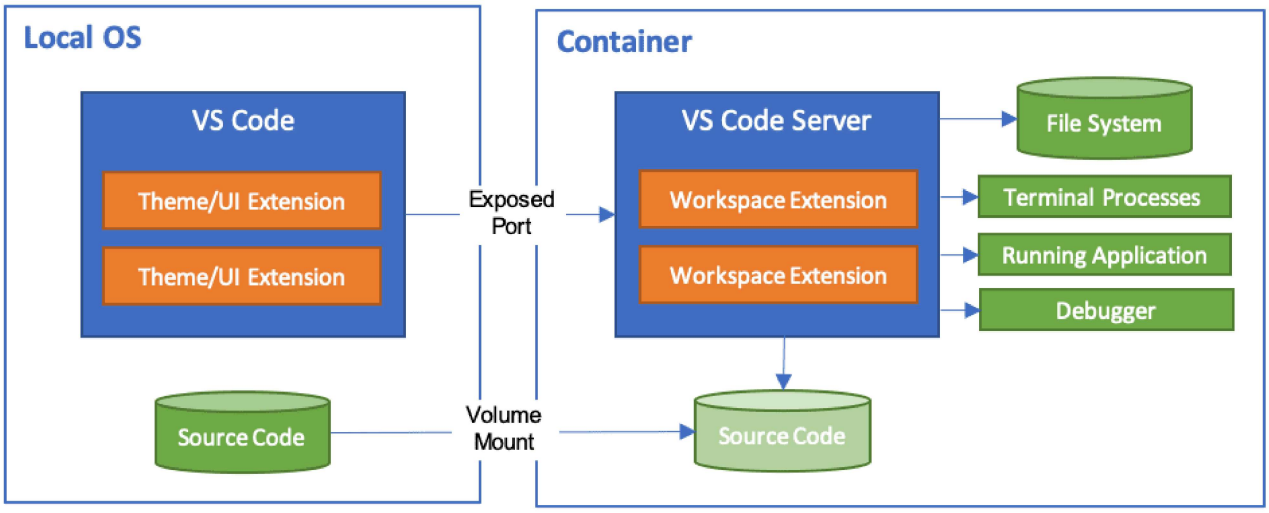
> **Important Note**: This assignment is designed to familiarize you with OS processes. Please avoid simply copying and pasting code from the assignment and executing commands without understanding them. Take the time to read through the instructions and understand the concepts being presented; aim to answer the questions at the end of each task.

Task 0: Introduction to VSCode devcontainers

In order to test you code, we use this year the VS.code devcontainer capabilitity. Devcontainers exploit an OS mechanism called namespaces to run isolated instances of the software that you will use for your lab activities and coursework. A container is a lightweight, standalone, and executable package hat includes everything needed to run a piece of software, including the code, runtime, libraries, and dependencies. They look like Virtual Machines, but they are much more lightweight. The most popular container technology is called Docker (https://www.docker.com/).

Devcontainers are used in development environments to ensure consistency across different development setups by providing a standardized environment. This helps developers avoid issues related to differences in local development environments and ensures that the application behaves the same way regardless of where it is run.

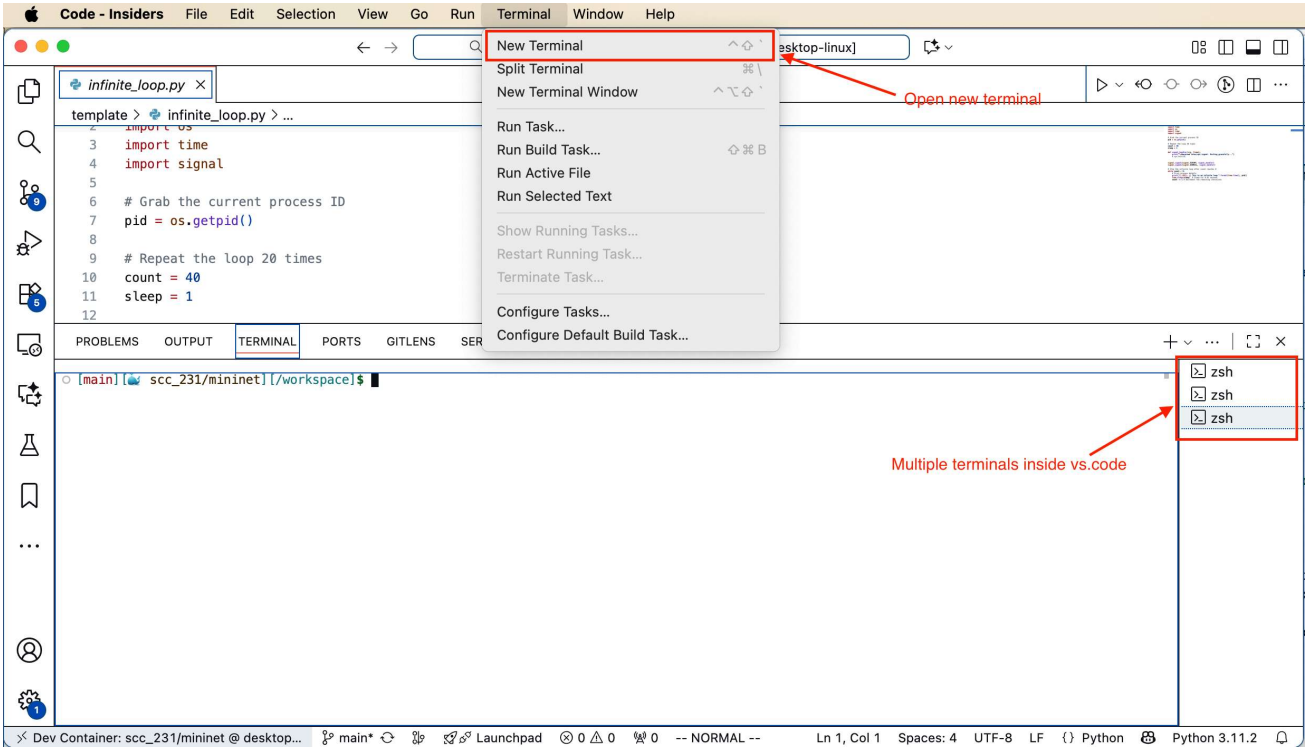
As part of this lab task, we provide a template project for VSCode, the automates the loading of the devcontainer. The project contains a `.devcontainer.json` file, that contains all the required configurations to load the container. You must unzip the template code files in a folder, and use the `File->Open Folder` menu option in VSCode to open the folder containing the unzipped files and load the decontainer environment. Be aware that in order to run commands in the devcontainer, you need to open a terminal in VSCode (Terminal -> New Terminal). You can then run the commands as you would in a regular terminal. **Running commands using the Ubuntu terminal application will not execute the commands inside the Devcontainer environment**. **You can open multiple terminal windows in VSCode, and each terminal will run commands inside the devcontainer**.



Devcontainer environment

You can run the sample code on your computer, by downloading the Docker Desktop installer from the official Docker website (https://www.docker.com/products/docker-desktop). For more detailed instructions, refer to the Docker Desktop documentation (https://docs.docker.com/desktop/).

The environment



Terminal in devcontainer

> [!WARNING] The environment setup within vs.code is not the same as your host system. You will be able to only access the files in the project folder you opened in vs.code. Also, in order to execture commands inside the container, you will need to open a terminal window in vs.code (Terminal -> New Terminal).

> The lab machines at Lancaster University give you access to “Docker” commands using Podman.
> Podman is a container engine that is compatible with Docker commands, so you can use the same commands you would use with Docker to manage containers and images. For the purpose of any 231 activities you can treat Docker and Podman as interchangeable.

Task 1: What is a Process?

A process is an instance of a program that is being executed. It contains the program code and the program state in memory. Each process has its own memory space, system resources, and execution context. Processes are managed by the operating system, which allocates resources and schedules their execution. Today we are going to explore the Linux process state and API using Python.

For this exercise, we will create a simple Python script that runs an infinite loop and prints a message every second. You can create a file named `infinite_loop.py` with the following content:

```
import time
import os
import sys


def main() -> None:
    # Repeat the loop 40 times
    count = 40
    sleep = 1    # Modify this value to change the sleep duration

    # Grab the current process ID
    pid = os.getpid()
    # Stop the infinite loop after count reaches 0
    while count > 0:
        # Print process details
        print("{:.03f}: {} This is an infinite loop.".format(time.time(),
         pid))
        time.sleep(sleep)  # Put the process to sleep for 'sleep' seconds
        count -= 1 # Decrement the remaining iterations

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

You can run this program using the following command:

```
python3 infinite_loop.py
```

You can view the program state using the `ps` command in another terminal window:

```
ps aux | grep infinite_loop.py
```

> Note: The `ps` command displays information about active processes. The `aux` options provide detailed information about all processes, and `grep infinite_loop.py` filters the output to show only the lines containing `infinite_loop.py`.

Task 1: Questions

1. What is the process ID (PID) of the infinite loop program? Does the PID change each time you run the program? Why or why not?
2. What happens if you run multiple instances of the infinite loop program (`python3 infinite_loop.py & python3 infinite_loop.py & python3 infinite_loop.py & python3 infinite_loop.py &`)? How can you differentiate between them using the `ps` command? Is the PID ordering in the messages sequential? Why or why not?
3. How accurate is the `time.sleep(1)` function in the infinite loop program? Does it guarantee that the program will print the message exactly every second? Why or why not? You can experiment by changing the sleep duration to a smaller value (e.g., `0.01` seconds) and observing the output. How does low sleep duration affect the accuracy of the timing?
4. Does smaller values of sleep duration lead to greater randomness in the order of the printed messages when running the command from question 2? You can experiment by changing the sleep duration to a smaller value (e.g., `0.01` seconds) and observing the output.

Task 2: Handling signals in Python

During the third lecture of the course, we learned about a special type of an exception, called a system call, which allows a process to request a service from the operating system's kernel. System calls provide an interface between user-space applications and the kernel, allowing programs to perform operations that require higher privileges or access to hardware resources. Examples of system calls include file operations (open, read, write, close), process management (fork, exec, wait), memory management (mmap, brk), and inter-process communication (pipe, shmget). System calls are typically invoked using a software interrupt or a special CPU instruction that switches the processor to kernel mode.

In order to support the inverse communication path, the OS uses the concept of *signals*. Signals are a limited form of inter-process communication used in Unix, Linux, and other POSIX-compliant operating systems. They are used to notify a process that a specific event has occurred. Signals can be sent by the OS or by other processes. When a signal is sent to a process, the OS interrupts the normal flow of execution and invokes a signal handler function, if one is defined. If no handler is defined, the default action for the signal is taken

(which may be to terminate the process). A good example of a signal is when you press `Ctrl + C` in the terminal where the program is running, which raises an **interrupt signal** (SIGINT) to the OS, in order to notify about the need to terminate a process.

The OS has a hardcoded list of supported *Signals* and system calls, each represented with a unique integer (Note: for the keen reader, you can check the relevant Linux header file). In this scenario, Ctrl+C triggers the OS to raise a SIGINT (Signal Interrupt) signal to the foreground process group, which has the value 2. If you re-run the command ps, you will see that the process is no longer running. Bellow you can find a list of common signals that the OS can handle:

Signal Name	Description	Value	Default Action
SIGHUP	Hangup detected on controlling terminal or death of controlling process		1
SIGINT	Interrupt from keyboard	2	Terminate
SIGQUIT	Quit from keyboard	3	Core dump
SIGKILL	Kill signal	9	Terminate
SIGTERM	Termination signal	15	Terminate
SIGCONT	Continue a process that was paused	19	Continue
SIGSEGV	Invalid memory reference	11	Core dump
SIGCHLD	Child process stopped or terminated	17	Ignore
SIGSYS	Bad system call	31	Core dump

You can send signals to processes from the command line using the `kill` command. For example, the command `kill -SIGTERM <PID>` will send the `SIGTERM` signal to the process with the specified PID, requesting it to terminate gracefully. You can also use the `kill` command to send other signals, such as `SIGKILL`, which forcefully terminates a process without allowing it to clean up resources. You can find the PID of the infinite loop program using the `ps` command as shown earlier.

By default operating systems implement a default action for each signal (fourth column in the table above). In Python, you can implement custom signal handlers using the `signal` module. You can define a custom signal handler function that will be called when a specific signal is received. For example, you can modify the `infinite_loop.py` script to handle the `SIGINT` signal gracefully. You firstly need to implement a signal handler function that will be called when the `SIGINT` signal is received. Here is an example of how to modify the `infinite_loop.py` script to handle the `SIGINT` signal:

```
def signal_handler(sig, frame):
    print("\nReceived interrupt signal. Ignoring signal...")
```

In the main part of the program, you need to import the `signal` (https://docs.python.org/3/library/signal.html) module and register the signal handler using the `signal.signal()` function `signal.signal(signal.SIGINT, signal_handler)`.The updated code now every time you run this modified script and press `Ctrl + C`, you should see the message "Received interrupt signal. Ignoring..." and will carry on execution. Do not despair, you can still terminate the process using the `kill` command and `SIGKILL`. You can experiment with handling other signals as well, such as `SIGTERM` or `SIGHUP`. You can raise these signals using the `kill` command from another terminal. The basic idea of custom signal handling is to allow the program to perform cleanup operations or other actions before terminating. We discuss a specific use-case of signal handling in subsequent tasks.

Task 2: Questions

1. What happens if you send a `SIGTERM` signal to the modified infinite loop program using the `kill` command? Does it exit gracefully or does it terminate immediately? Why?
2. What happens if you try to handle a signal that is not supported by the OS? For example, you can try to handle the `SIGKILL` signal, which cannot be caught or ignored. How does the program behave in this case?
3. A good way to terminate a program is to use the exit system call (*i.e.*, `sys.exit(0)`). What is the significance of the argument `0` (You can read about it in the python documentation (https://docs.python.org/3/library/sys.html#sys.exit)) or by using the help pages in Linux `man 3 kill`)? What happens if you pass a non-zero value to `sys.exit()`? You can experiment by changing the argument to a non-zero value. In

you terminal you can check the exit status of the last executed command using `echo $?`. What does the exit status indicate?

Task 3: Creating and managing child processes in Python

As dsiscussed during out lectures, Linux provides a set of system calls to manage the state of processes. Python provides a wrapper around these system calls using the `os` module. The most commonly used system calls for process management are `fork()`, `exec()`, `wait()`, and `exit()`. Here is a brief description of each system call:

System Call	Description
fork() (https://docs.python.org/3/library/os.html#os.fork)	Create a new process by duplicating the calling process. The new process is referred to as the child process.
exec() (https://docs.python.org/3/library/os.html#os.exec)	Replace the current process image with a new process image. This is typically used after a `fork()` to run a different program in the child process.
wait() (https://docs.python.org/3/library/os.html#os.wait)	Wait for a child process to terminate and retrieve its exit status. This is used by the parent process to synchronize with its child processes.
exit() (https://docs.python.org/3/library/os.html#os.exit)	Terminate the calling process and return an exit status to the parent process.

You can create child processes in Python using the `os.fork()` function. The `fork()` function creates a new process by duplicating the current process. The new process is called the child process, and the original process is called the parent process. Here is an example of how to create a child process using `os.fork()`:

```
import os
import time
import signal



child_finished = False

def sigchild_handler(signum, frame):
    global child_finished
    print("Child process terminated.")
    child_finished = True

# Create a child process. The parent process receives the child's PID,
        while the child process receives 0.
pid = os.fork()

if pid > 0:
    # This is the parent process
    print("Parent process PID: {}".format(os.getpid()))
    print("Child process PID from parent process: {}".format(pid))
    # Wait for the child process to finish
    # os.wait() can replace this block code  to block the parent process
    # until the child process terminates.
    signal.signal(signal.SIGCHLD, sigchild_handler)  # Prevent zombie
        processes
    while True:
        if child_finished:
            break
        time.sleep(1)
    print("Parent process has finished.")
else:
    # This is the child process
    print("Child process PID: {}".format(os.getpid()))
    for i in range(20):
        print("Child process is running... {}".format(i))
        time.sleep(1)
    print("Child process is exiting.")
```

In the code sample above we use a signal handler to catch the `SIGCHLD` signal, which is sent to the parent process when a child process terminates. This prevents the creation of zombie processes. The parent process waits for the child process to finish by checking the `child_finished` flag in a loop. You can use the `os.wait()` system call to block the parent process until the child process terminates, which is a simpler approach.

Task 3: Questions

1. What is the difference between the parent and child processes in terms of their PIDs? How can you identify which process is the parent and which is the child?

2. Below you can find a simple function to count all prime numbers between two integer values:

```
def count_primes(a, b):
    count = 0
    for num in range(a, b + 1):
        if num > 1:
            for i in range(2, int(num**0.5) + 1):
                if (num % i) == 0:
                    break
            else:
                count += 1
    return count
```

Can you implement a python program that estimates the number of prime numbers between 1 and 500000 using a varying number of child processes (parent and child) and the above prime counting function? Collect the execution time when using 1, 2, 4 ,and 5 process and plot the data. You can use the `time` command line tool to measure the execution time of your program (e.g., `time python3 prime.py`). You can use a spredsheet program for data plotting.

Task 4: Running external commands in Python

It is a common mechanism to run external commands from within another program. This is often done to leverage existing tools and utilities provided by the operating system. For example, instead of writting a custom implementation to list files in a directory, you can simply call the `ls` command. This is how terminal programs and psuedoterminals work. To achieve this functionality you can use the `os.fork()` and `os.execvp()` (https://docs.python.org/3/library/os.html#os.execvp) system calls. The `os.fork()` system call is used to create a new child process, while the `os.execvp()` system call is used to replace its memory space with a new program. Bellow, you can find a simple example that creates a child process to run the `ps -aux` command:

```
import os

def run_ps():
    pid = os.fork()  # create a child process

    if pid == 0:
        # Child process
        print(f"Child (PID={os.getpid()}): running 'ps -aux'")
        os.execvp("bash", ["bash", "-c", "ps -aux"])
        # execvp replaces the current process, so no code below this line
         runs in child
    else:
        # Parent process
        print(f"Parent (PID={os.getpid()}): waiting for child (PID=
         {pid})")
        os.wait()  # wait for the child process to finish
        print("Parent: child process finished")

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

When using the command line it is common to chain multiple commands together using pipes (`|`), to speed up execution. The GNU command line tools are . We used such a technique in Task 1 to process the output of the command `ps` and filter specific entries

based on a pattern match, using the `grep` command. Pipes are an inter-process communication mechanism, which we will explore in the next lab. For now we will use the fork() and exec() system calls to implement a simple command pipeline in Python. You can use intermediate files to store the output of one command and use it as input to another command. For this task, you should implement a process that realises a simple pipeline that counts all running processes using the `ps` command, filters the output to show only processes owned the user root using the `grep` command, and then counts the number of lines in the filtered output using the `wc -l` command. For example, the equivalent command line command would be:

```
bash -c 'ps aux > ps.txt';  bash -c 'grep root ps.txt > grep.txt'; wc -l
        grep.txt
```

The operator > in bash redirects the output of a command to a file, while the operator ';' separates multiple commands to be executed sequentially. To implement this command in Python, you will need to use `os.fork()` and `os.wait()`, to realise each stage of the pipeline.

Task 4: Questions

1. You can run external commands in Python using the `subprocess` module. The `subprocess.run()` function allows you to execute a command and wait for it to complete. Here is an example of how to run the `ls` command using the `subprocess` module:

```
import subprocess


def run_ps():
    result = subprocess.run(["ps", "aux"], capture_output=True,
         text=True)
    print("Output of 'ps aux':")
    print(result.stdout)


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

Check the document of the `subprocess` (https://docs.python.org/3/library/subprocess.html) command and inspect what options are available in the command. Can you implement the same functionality using the subprocess module instead of using `os.fork()` and `os.execvp()`?

Hacker Edition: Building a custom ps program

The Linux kernel is responsible for managing processes and maintaining the required state. This information is protected from user-space applications, but it is made available through the `/proc` filesystem. The `/proc` filesystem is a virtual filesystem that provides a hierarchical view of the system's processes and other kernel information. Each process has its own directory under the folder `/proc`, named after its PID. Inside each process directory, there are various files that contain information about the process, such as its command line, memory usage, open file descriptors, and more.

In this section, you will build a simplified version of the `ps` command using Python. This custom `ps` command will list all running processes along with their PIDs and command names. You will need to read the manpage for the proc filesystem using the command `man procfs`. You will to implement a Python program that reads the list of directories in the folder `/proc/`. You can read the documentation of the function `os.listdir()` (https://docs.python.org/3/library/os.html#os.listdir), to retrieve the list of folders. You will then need to read the documentation and discover how the to read the command line command that used to run the command.

> Note: You can ignore any directories that do not represent a process (i.e., directories that do not have a numeric name). You can read the documentation of the function `os.path.isdir()` to check if a path is a directory. In addition, some processes will have permission restrictions, i.e. they may not be accessible by a different user, like `root`. You can handle this by using a try-except block to catch any `PermissionError` exceptions that may occur when trying to read the command line of a process.