OPERATING SYSTEM (ENS351)

ASSIGNMENT - 1

on

Process Creation & Management Using Python OS

Submitted to:

Prof. Dr. Aman Jatain

Submitted by:

Harsh Kumar Jha

2301010467

B. Tech CSE

K. R. MANGALAM UNIVERSITY, GURUGRAM, HARYANA, INDIA

1. Introduction

In modern operating systems, efficient process management is fundamental to ensuring smooth multitasking and optimal resource utilization. This experiment provides a hands-on opportunity to explore and simulate core process management concepts found in Linux systems using Python programming.

The ability to create, control, and monitor processes is crucial not only for operating system developers but also for application developers who need to write efficient and responsive software. Through this assignment, we delve into how processes are created via the fork() system call, how a child process can execute different programs using exec(), and how processes communicate and synchronize with their parents.

Additionally, the experiment sheds light on important edge cases with processes such as zombie and orphan states, which can affect system performance if not managed properly. The /proc filesystem, a powerful feature of Linux, is used to inspect live process information, providing valuable insights into process states and resource usage.
Finally, by modifying process priorities with the nice system call, we observe firsthand how the operating system scheduler allocates CPU time to competing processes, influencing their execution order and overall system efficiency.

This practical approach transforms theoretical operating system concepts into tangible understanding and skills, preparing us for more advanced system programming and real-world scenarios.

2. Objectives

The primary goal of this experiment is to gain a thorough understanding of process management in Linux operating systems through practical simulation using Python. Specifically, the experiment aims to:

- Explore the lifecycle of processes, including creation, execution, and termination, by implementing the fork() and exec() system calls.
- Learn how to create child processes from a parent process and manage the parent-child relationship effectively.
- Identify and simulate special process states such as zombie and orphan processes to understand their causes and impact on system resources.
- Utilize the Linux /proc filesystem to inspect and extract details about running processes, such as their state, memory usage, executable paths, and open file descriptors.
- Demonstrate the role of process priority and the influence of the nice value on CPU scheduling, thereby observing how the operating system allocates resources among multiple competing processes.

Through achieving these objectives, the experiment aims to bridge the gap between theoretical concepts and practical system programming, laying a strong foundation for more advanced studies in operating system internals and application development.

3. Tools and Technologies Used

To successfully complete this experiment, we utilized the following tools and technologies:

- **Operating System:**
  The experiment was conducted on a Linux operating system (such as Ubuntu 20.04 or any compatible distribution). Linux provides a native environment to work with process management system calls (fork(), exec(), nice()) and direct access to the /proc filesystem, which is integral for process inspection.

- **Programming Language:**
  Python 3 was chosen for its simplicity and powerful standard libraries. The os module in Python allows interaction with the underlying operating system and facilitates process creation, execution, and management, closely mirroring the typical Linux system calls.

- **Python Modules Used:**
  - os: For system-level interactions like process forking, executing commands, reading process information from /proc, and setting nice values.
  - subprocess: To provide an alternative way to run system commands within child processes.
  - time: To incorporate time delays which help in observing process states such as zombies and orphans during the experiment.

- **Development Environment:**
  The code was developed and tested in a command-line terminal on the Linux system, which is ideal for direct interaction with system processes. Editors such as VSCode or any text editor compatible with Python that supports Linux file systems can be used.

4. Experiment Details and Code Explanation

This section walks through each task of the assignment, explaining the approach and highlighting key code segments to reinforce understanding.

Task 1: Process Creation Utility
This task involves creating multiple child processes from a single parent process using os.fork(). Each child process prints its own PID and its parent's PID, followed by a custom message. The parent process waits for all child processes to finish using os.wait(). This ensures proper synchronization and resource cleanup.
The critical function os.fork() splits the running program into two nearly identical processes, the parent and the newly created child, allowing us to observe how Linux manages multiple simultaneous processes effectively.

Task 2: Command Execution Using exec()
Building on Task 1, each child process replaces its own program's image with another program using os.execvp(). This system call is vital to execute different commands (like ls, date, ps) within the child without returning to the original Python script.
This mimics real-world scenarios where processes start other programs, highlighting one of the most fundamental functions of an operating system's process management.

Task 3: Zombie and Orphan Processes
This task simulates two special states in process management:

- **Zombie processes** occur when a child process finishes, but the parent hasn't yet called wait(). The child remains in the process table as a "defunct" process, which can waste system resources.
- **Orphan processes** are those whose parent exits before they do. The init system (PID 1) then adopts these children, ensuring they are handled correctly.

By deliberately skipping wait() and having the parent exit early, these states are demonstrated, showing their presence via system tools like ps. This deepens understanding of system resource management and process cleanup.

Task 4: Inspecting Process Info from /proc
Linux exposes detailed runtime information about processes through the /proc virtual filesystem. For this task, we input a process ID (PID) and read:

- **Process status** details including state and memory usage from /proc/[pid]/status.
- **Executable path** indicating the binary being run from /proc/[pid]/exe.
- **Open file descriptors** showing all files the process currently has open from /proc/[pid]/fd.

This introspection helps understand process properties and how Linux internally represents process metadata in real-time.

Task 5: Process Prioritization Using Nice Values
Finally, the experiment creates several CPU-bound child processes with different "nice" values. The nice value influences the process scheduling priority: lower values mean higher priority and thus more CPU time.
By observing the execution order and completion time of these child processes, we gain practical insight into how the Linux scheduler balances workload and allocates resources to ensure fairness and efficiency.

5. Results and Observations

The execution of the Python script yielded the following findings:

- Multiple child processes were successfully created and synchronized, demonstrating effective use of fork() and wait(). PIDs confirmed the parent-child relationships clearly.
- Execution of diverse Linux commands in child processes worked flawlessly, showcasing the effective use of execvp().
- Zombie processes appeared as expected when the parent did not call wait(), and orphan processes were adopted by the init system after the parent exited, verifying theory through practice.
- Live process details extracted from /proc showed comprehensive information like process state, memory status, executable location, and open files, emphasizing the power of Linux’s proc filesystem in OS monitoring.
- Allocation of different nice values led to CPU-intensive processes

finishing in varied order, with higher priority (lower nice value) processes completing sooner, confirming the scheduler's role in resource management.

These results validated the core concepts of Linux process management and reinforced the practical application of theoretical knowledge.

6. Challenges and Learning

During the experiment, a few challenges were encountered:

- Ensuring proper synchronization was crucial; missing wait() calls caused zombie states, which needed careful management to avoid misleading results.
- Accessing some /proc files required adequate permissions, reminding us of Linux's security considerations.
- Managing multiple child processes efficiently reinforced the complexity behind process scheduling and management that operating systems perform seamlessly.

Overall, this assignment provided invaluable hands-on experience, bridging abstract OS concepts with tangible outcomes.

7. Conclusion

This lab provided a practical exploration of Linux process management by implementing fundamental system calls and inspecting runtime process behavior using Python. It successfully demonstrated process creation, command execution within child processes, the phenomenon of zombies and orphans, process introspection via /proc, and the impact of process priority on scheduling.

The knowledge gained here lays a strong groundwork for the complexities of operating system design and real-world software development relying on process control.

Harsh Kumar Jha
Date: 15-September-2025