System Programming - Linux
Week 7 - Introduction to Processes

Lecturer: Dr. Aggrey Obbo (PhD)

May 30, 2025

Contents

1 Introduction **1**

2 The Concept and Fundamentals of Linux Process **2**

3 Monitoring Running Processes **4**

4 Signals(Controlling Processes) **5**

5 Foreground and Background Processes: **5**

6 Threads and Context Switching **5**

7 Conclusion **6**

8 Appendix: Some Basic Process Commands **7**

1 Introduction

A Linux process is a dynamic execution acting on behalf of a process which utilizes hardware such as CPU and memory. It is a running entity with a finite lifetime. The Linux file system is a complex data structure for permanent storage of data on devices. It is a file and directory structure and a way for processes to store and manipulate data over time.

It is crucial for a Linux System programmer to be able to monitor/control processes, deal with InterProcess communication(IPC) and also to optimize the system resources. It allows them to write efficient and robust applications that communicate properly with the operating system, handle system calls properly, and debug issues with process behavior and resource competition. These notes are to help the reader be able to do just that.

Objectives

- Describe Concepts and Fundamentals of Process concepts
- Utilize command-line tools to monitor processes
- Perform basic process control
- Manage foreground and background processes
- Understand the Linux process lifecycle

2 The Concept and Fundamentals of Linux Process

In Linux, a process refers to an active instance of a program and is defined by several important elements. When a program is run, the kernel loads it into memory, allocates the necessary resources, and initiates a process.

The key elements of a process are as follows:

Elements of a Process

[1][2] Every process is distinctly recognized by a Process ID (PID), which is a positive integer assigned sequentially by the kernel. The parent process is the process that created the current process, and its PID is referred to as the Parent Process ID (PPID). While the initial process, started by the kernel during boot, is called init process and is a PID of 1.

A process possesses its own address space, a virtual memory area that comprises the program's executable code, its data (including global and static variables), the stack (for function calls and local variables), and the heap (for dynamic memory usage). Additionally, the process keeps track of its execution context, such as the program counter (which shows the next instruction to execute) and the values in various registers.

Moreover, a process is linked to resources like open files, network connections, and allocated memory blocks. It also has credentials, which include the user and group IDs that dictate its access rights. The kernel maintains a process control block (PCB) for each active process, which holds all this critical information, enabling the operating system to manage and schedule processes effectively.

Environment variables function as adjustable configuration parameters that affect the behavior of processes in Linux. When a new process is initiated, it inherits a collection of environment variables from its parent process, typically the shell. These variables can provide important details such as the paths to executable files (PATH), the user's home directory (HOME), the system language settings (LANG), and configurations specific to applications.

Process Lifecycle

[1][2][3] The various states of processes are essential for understanding how the Linux kernel manages multiple concurrent programs. They denote different phases in a process's lifecycle, reflecting its ongoing actions and resource needs.

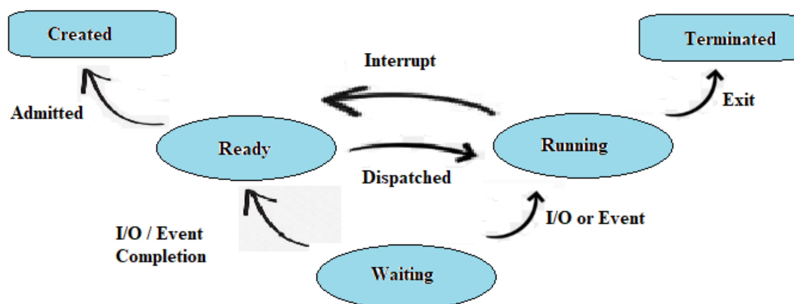

Figure 1: Process states

Process Creation: fork() creates a new child process that is a copy of the parent process. The new process will have the same code, data, and resources as the parent process, but it still has its own unique process ID. Both the parent and child processes execute the next instruction after the fork() call.

exec() replaces the current process code and data with the code and data of a new program. exec() terminates the current process and a new program is loaded and executed. The process ID remains the same.

Running State: When a process is in the running state, it is currently being processed by the CPU. In managing resources, the kernel frequently interrupts running processes to give other processes a chance to execute, resulting in a number of short periods in a ready-to-run state. This state have still been categorized as running state.

Sleeping State: This state signifies that the process is awaiting the occurrence of an event, such as completion of an I/O operation, a notification from another process, or even expiration of a timer.

Interruptible sleep can be awakened by signals, whereas uninterruptible sleep typically waits for crucial kernel tasks to complete and cannot be interrupted, not even by a SIGKILL.

Stopped state: In the stopped state, a process's execution has been momentarily paused, often because of a signal like SIGSTOP initiated by the user or another process. These processes can be restarted later with a SIGCONT signal.

Terminated: Normal exit when the program finishes termination due to received signals.

A Zombie process is a process that has completed its execution, but its entry continues to exist in the process table. This occurs because the parent process has not yet collected its exit status. Although zombiebiebie processes consume very few resources, a buildup of them may indicate an issue.

Orphaned processes are those whose parent has terminated but are adopted by the init process.

Understanding all these states allows us to troubleshoot performance problems and pinpoint processes that are behaving erratically on our Linux systems here in Kampala.

Process Control Block (PCB)

A process control block (PCB) or process descriptor is a data structure that the computer operating system use to hold relevant information regarding a process. When a process is initiated, the operating system generates a corresponding process control block. The PCB plays the role of process management by storing specific details about the process that are useful during its execution. An array of PCBs, which contains a PCB for all of the current processes in the system, is called the Process Table.

Pointer
Process State
Process Number
Process Counter
Register
Memory Limit
List of open files
. . .

Figure 2: Process control block

The exact contents of the PCB can vary slightly between perating systems, the following are however the common and essential parts of a PCB:

1. Pointers: Pointer to the Parent PCB: Play the role of linking and managing processes. The pointers:
   - Point to child PCBs: For the parent process to keep track of its children.
   - Pointer to the next PCB in the queue to enable organizing the PCBs into various data structures like linked lists or scheduling queues.
   - Pointers to memory segments define the location of the process's code, data, stack, and heap in memory
   - Pointers to open files maintain a list of resources the process is using

   These different types of pointers are used by the OS to maintain process relationships, and ensure efficient CPU utilization.

2. Process State: This indicates the current status of the process (New, Ready, running, waiting, terminated or suspended:

3. Process Identification or Process Number(PID): PID is a unique numerical identifier assigned to every created process. Parent Process ID (PPID) is the ID of the process that created the current process.

   User ID and Group ID Identify the user and user groups associated with a process. These are used during access control and resource management.

4. Program Counter (PC): PC is a register that holds the memory address of the next instruction to be executed. It plays an important role when resuming process execution after an interruption.

5. CPU Registers: This is a section that acts a high speed storage location of instructions for immediate processing, allowing for quick retrieval and manipulation of data. It is crucial for efficient program execution helps in context switching. These registers include; index registers, general-purpose registers, and status registers.

6. Memory Limit: The memory limit defines the details about memory allocated to the process. The information included depends on the memory management scheme used by the operating system for example base and limit Registers define the range of memory addresses accessible by the process, Page tables are used in paging systems to map virtual addresses to physical addresses, and segment tables used in segmentation systems to define the segments of memory allocated to the process.

7. Process Priority: This is a value indicating the relative importance of the process, and is used by the scheduler to determine which process in the queue should run next.

8. Accounting Information: The accounting information tracks resources consumed by the process. These can include; amount of CPU time used, time elapsed since the process started, account numbers and limits on resource utilization.

9. I/O Status Information: This keeps track of the I/O resources allocated to the process. For example list of open files and list of I/O devices currently used by the process.

10. Context Data: This encompasses all the information that needs to be saved when a process is switched out of the CPU so that may be necessary in aiding restoration of the process at a later time. This may include the contents of various registers, the program counter, and other process-specific data.

    The PCB is essential for multitasking operating systems. When the CPU switches between processes (context switching), the operating system saves the state of the current process in its PCB and loads the saved state of the next process from its PCB. This allows multiple processes to share the CPU and other system resources efficiently, creating the illusion of concurrent execution.

3 Monitoring Running Processes

By default, every started process runs in foreground and receives input from an input device such as a keyboard, and sends output to an output device normally the screen.

1. The ps Command: The ps command displays information about running processes. It shows details like process ID (PID), user, CPU and memory usage, and the command being executed for the current process. Output of the ps command can be filtered per user or group to provide specific information using the options such as:

   - -p <PID>: Getting information about a specific process using its ID.
   - grep <pattern>: Filtering the output to find specific processes (e.g. ps aux).
   - aux: Showing all processes running on the system, regardless of the user.
   - -u <username>: Listing processes owned by a specific user

2. Real-time Monitoring with top and htop: top is used to show all the running processes within the working environment of Linux. htop often provides a more user-friendly interface.

3. nice: Starts a new process and assigns it a priority value referred to as nice at the same time. The nice value ranges from -20 to 19, where -20 is of the highest priority.

   Syntax:

nice [-nice value]

The renice command is used to change the priority of an already running process and the syntax is as follows:

renice [-nice value] [process id]

4. df command is used to show the amount of available disk space being used by the file systems and the syntax is as follows:

   df

5. free shows the total amount of free and used physical and swap memory in the system. In addition, it also shows the buffers used by the kernel and the syntax is as follows:

   free

4 Signals(Controlling Processes)

[3][4] A signal is a notification to a process that something has happened. Messages sent to processes to tell them to do something. To interrupt, suspend or kill a process, a user can use terminal characters (Ctrl-C) or (Ctrl-Z) keyboard combinations.

Some of the common signals include:

- SIGINT (2) (Ctrl+C): Interrupt triggered to stop a foreground process).
- SIGQUIT (3): Quit SIGKILL (9): Forcefully terminate a process. Used as a last resort!
- SIGTERM (15): **Gracefully** terminate a process This is the default kill signal.
- The kill Command: follows:
  kill <PID> (sends SIGTERM).
  This can be used with options as
  kill -<signal_number> <PID>or kill -<signal_name<PID>. or
  pkill and killall to kill processes by name. This is useful when you don't know the PID.
  pkill <process_name>: Sends a signal to all processes matching the name.
  killall <process_name>: Similar to pkill. (Exercise caution)

5 Foreground and Background Processes:

Programs running programs directly in the terminal are foreground, while background processes run in the background. The following commands are used with respect to background and foreground processes.

jobs: Listing currently running background jobs.
bg %<job_number<: Resuming a stopped job in the background.
fg %<job_number<: Bringing a background job back to the foreground.

6 Threads and Context Switching

Threads are small units of execution that operate within a process, utilizing the process's memory space, code, and data segments. This shared environment facilitates efficient communication and resource use compared to establishing entirely new processes. The main reason for employing threads is to realize concurrency within a single process, allowing tasks to run in parallel and enhancing responsiveness, particularly in applications that are I/O-bound.

To create a thread requires a system call such as pthread_create in POSIX environments such as Linux. This system call sets up a fresh thread control block (TCB) and a stack for the newly created thread. The TCB holds thread-related data, including the thread ID, program counter, register values, and scheduling priority. The new thread subsequently starts executing the designated function.

Context Switching in Linux: When multiple threads operate within a process (or across various processes), the operating system's scheduler oversees their execution. The mechanism known as context switching allows the kernel to halt the execution of one thread (or process) and then continue with another as illustrated in Figure 3 below. This process requires the saving of the complete state of the active thread (including registers, program counter, and stack pointer) into its Thread Control Block (TCB), followed by loading the saved state of the next thread to execute from its TCB. Generally, context switching between threads in the same process is quicker than switching between processes, as the memory space remains unchanged, which prevents Translation Lookaside Buffer (TLB) flushes. The Linux kernel uses a variety of scheduling algorithms to decide which thread should run next, striving for both fairness and efficiency.

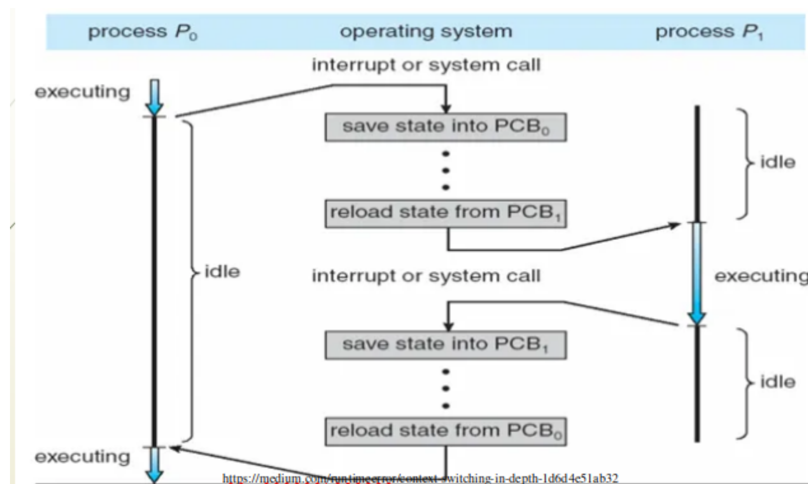

Figure 3: context switching

7 Conclusion

This lecture notes has shown the concepts of Linux processes and highlighted how that knowledge is vital for efficient system administration. Ideas such as the different states of processes, the relationships between parent and child processes, and the signaling mechanisms involved were presented. The role of tools such as ps, top, kill, nice, and renice for overseeing and managing active applications were explained. Processes management is especially important on shared systems to ensure equitable resource distribution and avoid system instability. Pursuing further knowledge into advanced subjects such as background processes (daemons) and the contemporary system initialization and management tool, systemd, will enhance any readers proficiency in the Linux environment.

List of References

1. Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Operating systems: Three easy pieces.

2. Mitchell, M., Oldham, J., & Samuel, A. (2001). Advanced linux programming (pp. 95-129). Berkeley: New riders.

3. William Jr, E. (2012). The Linux command line: a complete introduction. No Starch Press.

4. Love, R. (2013). Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.".

8 Appendix: Some Basic Process Commands

Running a command in the background
Add an ampersand (&) at the end rof the command to run it in the background. For example:
sleep 10 &
sleep 10 will run for 10 seconds. The & pushes it to the background.

Listing processes
ps
This will show processes associated with the current user and terminal.
To see all processes with a much detailed information, use;
ps aux

Finding a process by name (pgrep)

To output the PID of any running process named ”sleep”.
pgrep sleep