

# UNIX Fundamentals: Level 2

## UNIX Concepts

SYS-ED/Computer Education Techniques, Inc.

Ch 1: 1

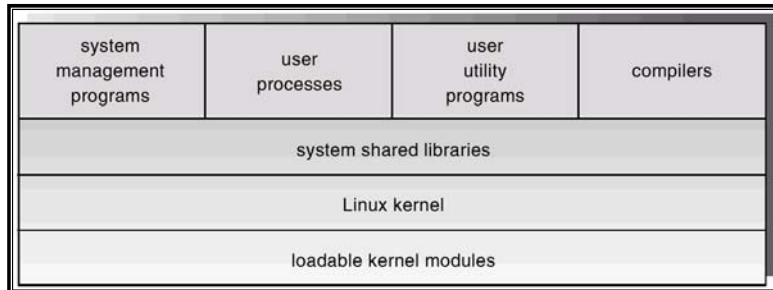
## Components of a UNIX/Linux System

- UNIX/Linux is composed of three main bodies of code.
- It is important to distinguish between the kernel and all other components.
- The kernel is responsible for maintaining the important abstractions of the operating system.
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer.
  - All kernel code and data structures are kept in the same single address space.

SYS-ED/Computer Education Techniques, Inc.

Ch 1: 2

## Components of a Unix/Linux System



SYS-ED/Computer Education Techniques, Inc.

Ch 1: 3

## Components of a UNIX/Linux System

- The system libraries define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code.
- The system utilities perform individual specialized management tasks.

SYS-ED/Computer Education Techniques, Inc.

Ch 1: 4

# Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The fork system call creates a new process.
  - A new program is run after a call to execve.
- A process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program.
- Process properties fall into three groups: the process's identity, environment, and context.

# Process Identity

- **PID: Process ID** - The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process.
- **Credentials** - Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files.

## Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
  - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.

## Process Environment

- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the user-mode system software.
- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

## Process Context

- The (constantly changing) state of a running program at any point in time.
- The scheduling context is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.

## Process Context

- The kernel maintains accounting information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.
- The file table is an array of pointers to kernel file structures. When making file I/O system calls, processes refer to files by their index into this table.

## Process Context

- Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
- The signal-handler table defines the routine in the process's address space to be called when specific signals arrive.
- The virtual-memory context of a process describes the full contents of the its private address space.

## Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.
- A distinction is only made when a new thread is created by the clone system call.
  - fork creates a new process with its own entirely new process context
  - clone creates a new process with its own identity, but that is allowed to share the data structures of its parent
- Using clone gives an application fine-grained control over exactly what is shared between two threads.

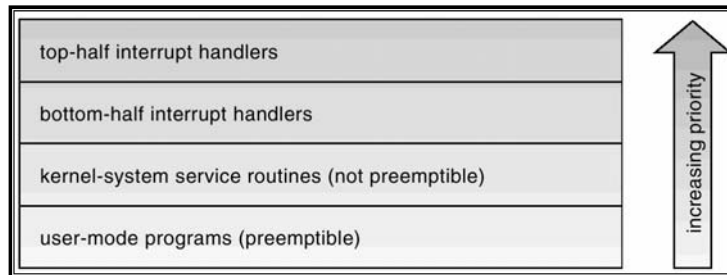
## Scheduling

- The job of allocating CPU time to different tasks within an operating system.
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks.
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

## Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs.
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section.

## Interrupt Protection Levels



- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level.
- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

SYS-ED/Computer Education Techniques, Inc.

Ch 1: 15

## Process Scheduling

- Linux uses two process-scheduling algorithms:
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real-time algorithm for tasks where absolute priorities are more important than fairness

SYS-ED/Computer Education Techniques, Inc.

Ch 1: 16

## Process Scheduling

- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class.
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the longest-waiting one
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robin processes of equal priority automatically time-share between themselves.

## Symmetric Multiprocessing

- Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors.
- To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.

## Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory.
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes.

## Virtual Memory

- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required.
- The VM manager maintains two separate views of a process's address space:
  - A logical view describing instructions concerning the layout of the address space.  
The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space.
  - A physical view of each address space which is stored in the hardware page tables for the process.

# Virtual Memory

- Virtual memory regions are characterized by:
  - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
  - The region's reaction to writes (page sharing or copy-on-write).
- The kernel creates a new virtual address space
  1. When a process runs a new program with the **exec** system call
  2. Upon creation of a new process by the **fork** system call

# Virtual Memory

- On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions.
- Creating a new process with **fork** involves creating a complete copy of the existing process's virtual address space.
  - The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child.
  - The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented.
  - After the **fork**, the parent and child share the same physical pages of memory in their address spaces.

## Static and Dynamic Linking

- A program whose necessary library functions are embedded directly in the program's executable binary file is statically linked to its libraries.
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions.
- Dynamic linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once.

## File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics.
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the VFS - virtual file system.
- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - The *inode-object* and the *file-object* structures represent individual files
    - the *file system object* represents an entire file system
  - A layer of software to manipulate those objects.

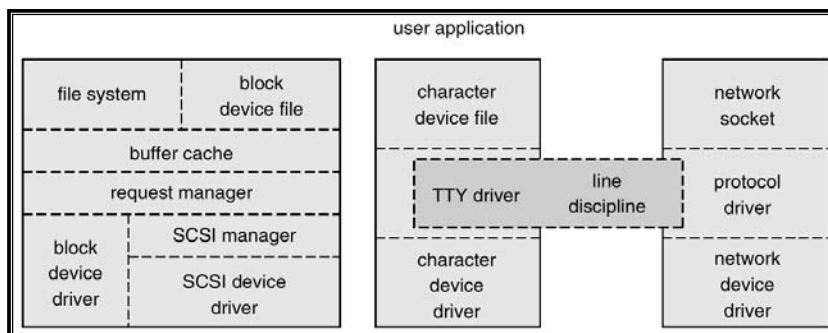
# Input and Output

- The Unix/Linux device-oriented file system accesses disk storage through two caches:
  - Data is cached in the page cache, which is unified with the virtual memory system.
  - Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.
- Linux splits all devices into three classes:
  - Block devices allow random access to completely independent, fixed size blocks of data.
  - Character devices include most other devices; they don't need to support the functionality of regular files.
  - Network devices are interfaced via the kernel's networking subsystem.

SYS-ED/Computer Education Techniques, Inc.

Ch 1: 25

# Device-driver Block Structure



SYS-ED/Computer Education Techniques, Inc.

Ch 1: 26

## Block Devices

- Provide the main interface to all disk devices in a system.
- The block buffer cache serves two main purposes:
  - It acts as a pool of buffers for active I/O.
  - It serves as a cache for completed I/O.
- The request manager manages the reading and writing of buffer contents to and from a block device driver.

## Character Devices

- A device driver which does not offer random access to fixed blocks of data.
- A character device driver must register a set of functions which implement the driver's various file I/O operations.
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device.
- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface.

## Passing Data Between Processes

- The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read at the other.
- Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space.
- To obtain synchronization, however, shared memory must be used in conjunction with another Interprocess-communication mechanism.

## Shared Memory Object

- The shared-memory object acts as a backing store for shared-memory regions in the same way as a file can act as backing store for a memory-mapped memory region.
- Shared-memory mappings direct page faults to map in pages from a persistent shared-memory object.
- Shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.