CSE 4/521 Introduction to Operating Systems

Lecture 28 – The Linux System

(Linux History, Design Principles, Kernel Modules, Process Management, Scheduling)

Summer 2018

Overview

Objective:

- To explore the history of the UNIX operating system from which Linux is derived and the principles upon which Linux's design is based.
- To examine the Linux process model and illustrate how Linux schedules processes .

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling

Overview

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling

Linux History

- Linux is a modern, free operating system based on UNIX standards
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility, released as open source
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- Designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Linux system has many, varying Linux distributions including the kernel, applications, and management tools

Linux History: The Linux Kernel

- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system
- Linux 1.0 (March 1994) included these new features:
  - Support for UNIX's standard TCP/IP networking protocols
  - BSD-compatible socket interface for networking programming
  - Device-driver support for running IP over an Ethernet
  - Enhanced file system
  - Support for a range of SCSI controllers for high-performance disk access
  - Extra hardware support
- Version 1.2 (March 1995) was the final PC-only Linux kernel
- Kernels with odd version numbers are development kernels, those with even numbers are production kernels

Linux History: Linux 2.0

- Released in June 1996, 2.0 added two major new capabilities:
  - Support for multiple architectures, including a fully 64-bit native Alpha port
  - Support for multiprocessor architectures
- Other new features included:
  - Improved memory-management code
  - Improved TCP/IP performance
  - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand
  - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems
- 3.0 released in 2011, 20th anniversary of Linux, improved virtualization support, new page write-back facility, improved memory management, new Completely Fair Scheduler

Linux History: Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project
- The main system libraries were started by the GNU project, with improvements provided by the Linux community
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories
- File System Hierarchy Standard document maintained by the Linux community to ensure compatibility across the various system components

Linux History: Linux Distributions

- Standard, precompiled sets of packages, or distributions, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools
- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management
- Early distributions included SLS and Slackware
  - Red Hat and Debian are popular distributions from commercial and noncommercial sources, respectively.
  - The RPM Package file format permits compatibility among the various Linux distributions

Linux History: Linux Licensing

- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation
- Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product
    - Can sell distributions, but must offer the source code too

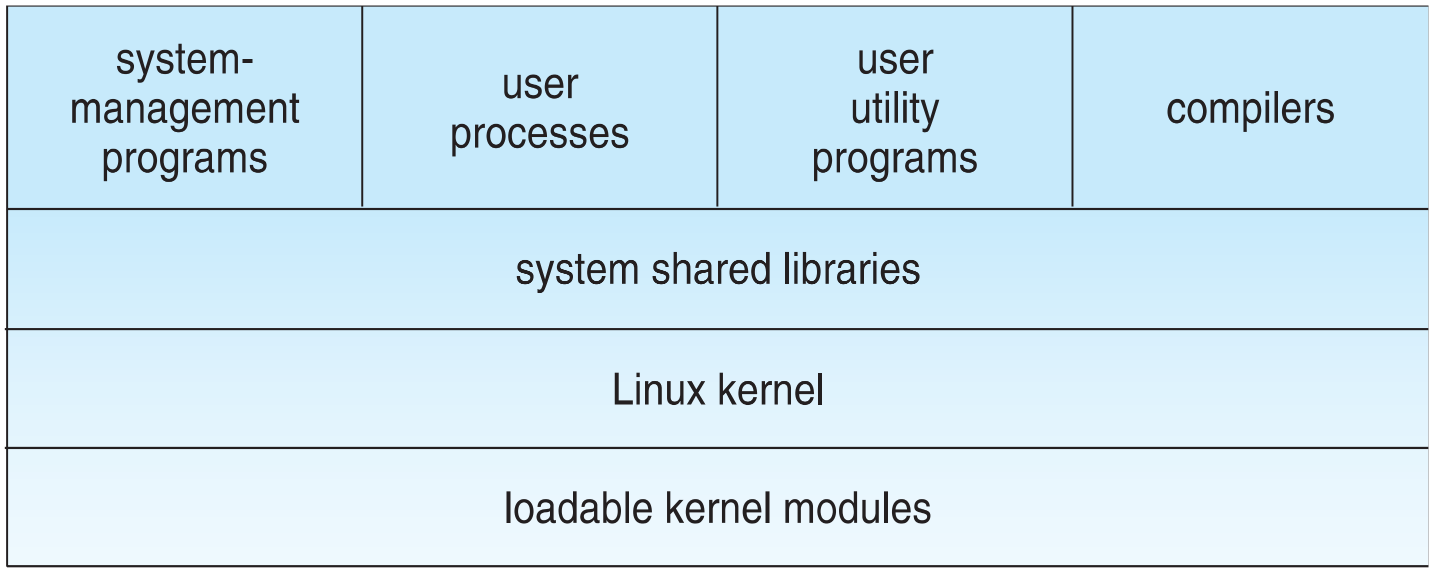
Overview

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
    - Supports Pthreads and a subset of POSIX real-time process control
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Design Principles: Components of a Linux System



Design Principles: Components of a Linux System

- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction 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

Design Principles: Components of a 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
- User-mode programs rich and varied, including multiple shells like the bourne-again (bash)

Overview

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling

Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
- Four components to Linux module support:
  - module-management system
  - module loader and unloader
  - driver-registration system
  - conflict-resolution mechanism

Kernel Modules: Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Module loading is split into two separate sections:
  - Managing sections of module code in kernel memory
  - Handling symbols that modules are allowed to reference
- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed

Kernel Modules: Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time
- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

Kernel Modules: Conflict Resolution

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.
- The conflict resolution module aims to:
  - Prevent modules from clashing over access to hardware resources
  - Prevent autoprobes from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware:
    1. Kernel maintains list of allocated HW resources
    2. Driver reserves resources with kernel database first
    3. Reservation request rejected if resource not available

Overview

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling

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 `exec()`
- Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups: the process's identity, environment, and context

Process Management: Process Identity

- Process ID (PID) - 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
- Personality - Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls
- Namespace – Specific view of file system hierarchy
  - Most processes share common namespace and operate on a shared file-system hierarchy
  - But each can have unique file-system hierarchy with its own root directory and set of mounted file systems

Process Management: 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.
- 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 Management: 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
- 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, the file descriptor (fd)

Process Management: 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

Process Management: 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
    - Both are called tasks by Linux
- A distinction is only made when a new thread is created by the `clone()` system call
    - `fork()` creates a new task with its own entirely new task context
    - `clone()` creates a new task 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

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Overview

- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling

Scheduling

- The job of allocating CPU time to different tasks within an operating system
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver
- As of 2.5, new scheduling algorithm – preemptive, priority-based, known as *O*(1)
  - Real-time range
  - nice value
  - Had challenges with interactive performance
- 2.6 introduced Completely Fair Scheduler (CFS)

Scheduling: Completely Fair Scheduler (CFS)

- Eliminates traditional, common idea of time slice
- Instead all tasks allocated portion of processor's time
- CFS calculates how long a process should run as a function of total number of tasks
- *N* runnable tasks means each gets 1/*N* of processor's time
- Then weights each task with its nice value
  - Smaller nice value -> higher weight (higher priority)

Scheduling: Completely Fair Scheduler (CFS)

- Then each task run with for time proportional to task's weight divided by total weight of all runnable tasks
- Configurable variable target latency is desired interval during which each task should run at least once
  - Consider simple case of 2 runnable tasks with equal weight and target latency of 10ms – each then runs for 5ms
    - If 10 runnable tasks, each runs for 1ms
    - Minimum granularity ensures each run has reasonable amount of time (which actually violates fairness idea)

Scheduling: 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

Scheduling: Kernel Synchronization

- Linux uses two techniques to protect critical sections:

1. Normal kernel code is nonpreemptible (until 2.6)
   – when a time interrupt is received while a process is executing a kernel system service routine, the kernel's need_resched flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode
2. By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures

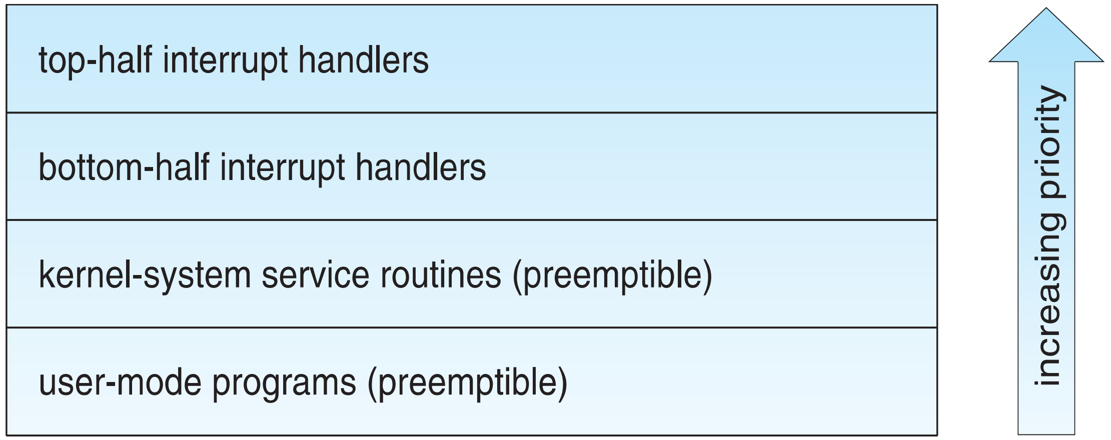
- Provides spin locks, semaphores, and reader-writer versions of both
  - Behavior modified if on single processor or multi:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Scheduling: Kernel Synchronization

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration
- Interrupt service routines are separated into a *top half* and a *bottom half*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

Scheduling: Interrupt Protection Levels


top-half interrupt handlers
bottom-half interrupt handlers
kernel-system service routines (preemptible)
user-mode programs (preemptible)
increasing priority

- 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

Scheduling: 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
- Until version 2.2, 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
- Later releases implement more scalability by splitting single spinlock into multiple locks, each protecting a small subset of kernel data structures
- Version 3.0 adds even more fine-grained locking, processor affinity, and load-balancing

Credits for slides

Silberschatz, Galvin and Gagne