Chapter 20 – Case Study: Linux

Outline
20.1 Introduction
20.2 History
20.3 Linux Overview
20.3.1 Development and Community
20.3.2 Distributions
20.3.3 User Interface
20.3.4 Standards
20.4 Kernel Architecture
20.4.1 Hardware Platforms
20.4.2 Loadable Kernel Modules
20.5 Process Management
20.5.1 Process and Thread Organization
20.5.2 Process Scheduling
20.6 Memory Management
20.6.1 Memory Organization
20.6.2 Physical Memory Allocation and Deallocation
20.6.3 Page Replacement
20.6.4 Swapping

Chapter 20 – Case Study: Linux

Outline (continued)
20.7 File Systems
20.7.1 Virtual File System
20.7.2 Virtual File System Caches
20.7.3 Second Extended File System (ext2fs)
20.7.4 Proc File System
20.8 Input/Output Management
20.8.1 Device Drivers
20.8.2 Character Device I/O
20.8.3 Block Device I/O
20.8.4 Network Device I/O
20.8.5 Unified Device Model
20.8.6 Interrupts
20.9 Kernel Synchronization
20.9.1 Spin Locks
20.9.2 Reader/Writer Locks
20.9.3 Seqlocks
20.9.4 Kernel Semaphores
20.10 Interprocess Communication

Chapter 20 – Case Study: Linux

Outline (continued)

20.10.1 Signals
20.10.2 Pipes
20.10.3 Sockets
20.10.4 Message Queues
20.10.5 Shared Memory
20.10.6 System V Semaphores
20.11 Networking
20.11.1 Packet Processing
20.11.2 Netfilter Framework and Hooks
20.12 Scalability
20.12.1 Symmetric Multiprocessing (SMP)
20.12.2 Nonuniform Memory Access (NUMA)
20.12.3 Other Scalability Features
20.12.4 Embedded Linux
20.13 Security
20.13.1 Authentication
20.13.2 Access Control Methods
20.13.3 Cryptography

Objectives

After reading this chapter, you should understand:

- Linux kernel architecture.
- the Linux implementation of operating system components such as process, memory and file management.
- the software layers that compose the Linux kernel.
- how Linux organizes and manages system devices.
- how Linux manages I/O operations.
- interprocess communication and synchronization mechanisms in Linux.
- how Linux scales to multiprocessor and embedded systems.
- Linux security features.

20.1 Introduction

- Linux kernel version 2.6
  - Core of the most popular open-source, freely distributed, full-featured operating system
  - Linux source code is available to the public for examination and modification and is free to download and install
- Popular in high-end servers, desktop computers and embedded systems
- Supports many advanced features
  - Symmetric multiprocessing (SMP),
  - Nonuniform memory access (NUMA),
  - Access to multiple file systems
  - Support for broad spectrum of hardware architectures

20.2 History

- Created in 1991 by Linus Torvalds a student at the University of Helsinki, Finland
  - The name Linux is derived from "Linus" and "UNIX"
- The Minix source code served as a starting point
- Torvalds sought advice from the community
- Developers continued to support the concept of a new, freely available operating system

20.2 History

- Distribution
  - Enables users unfamiliar with Linux details to install and use Linux
  - Includes software such as
    - The Linux kernel
    - System applications (e.g., user account management, network management and security tools)
    - User applications (e.g., GUIs, Web browsers, text editors, e-mail applications, databases, and games)
    - Tools to simplify the installation process

20.2 History

- Major version number
  - Incremented at Torvalds's discretion for each kernel release that contains a feature set significantly different from that of the previous version
  - Minor version number (the digit directly following the first decimal point
    - Even numbers are considered to be stable releases
    - Odd minor version number, such as 2.1.6, indicates a development version
  - Digit following the second decimal point is incremented for each minor update to the kernel

20.3 Linux Overview

- Linux systems include user interfaces and applications in addition to the kernel
- Borrows from the UNIX layered system approach
- System contains kernel threads to perform services
  - Implemented as daemons, which sleep until awakened by a kernel component
- Multiuser system
  - Restricts access to important operations to users with superuser (also called root) privileges

20.3.1 Development and Community

- Torvalds controls all modifications to the kernel
- Relies on a group of about 20 “lieutenants” to manage kernel enhancements
- As a development kernel nears completion:
  - Feature freeze: no new features are added to the kernel
  - Code freeze: only code that fixes important bugs are accepted
- Many corporations support Linux development
- Linux is distributed under the GNU Public License (GPL)
- Linux is free, copyrighted software

20.3.2 Distributions

- Over 300 distributions available
- Typically organized into packages, each containing a single service or application
- Popular distributions include:
  - Debian
  - Mandrake
  - Red Hat
  - SuSE
  - Slackware

20.3.3 User Interface

- Can be accessed via the command-line via shells such as *bash*, *csh* and *esh*
- Most Linux GUIs are layered
  - X Window System
    - Lowest level
    - Provides to higher GUI layers mechanisms to create and manipulate graphical components
  - Window manager
    - Builds on mechanisms in the X Window System interface to control the placement, appearance, size and other window attributes
  - Desktop environment (e.g., KDE, GNOME)
    - Provide user applications and services

20.3.4 Standards

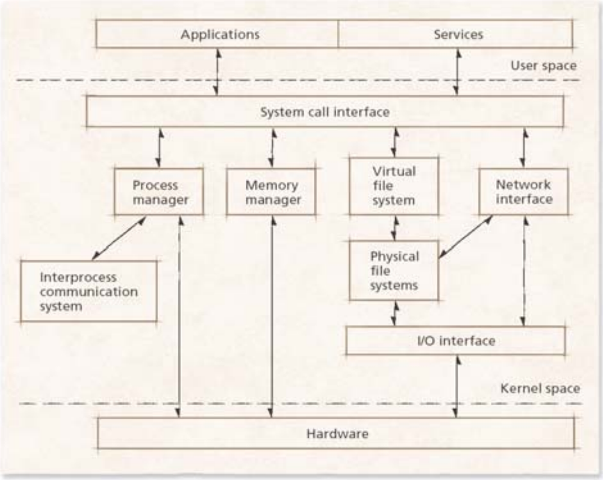
- Linux increasingly conforms to popular standards such as POSIX
- The Single UNIX Specification (SUS)
  - Suite of standards that define user and application programming interfaces for UNIX operating systems, shells and utilities
  - Version 3 of the SUS combines several standards (including POSIX, ISO standards and previous versions of the SUS)
- Linux Standards Base (LSB)
  - Project that aims to standardize Linux so that applications written for one LSB-compliant distribution will compile and behave exactly the same on any other LSB-compliant distribution

20.4 Kernel Architecture

- Monolithic kernel
  - Contains modular components, however
- UNIX-like or UNIX-based operating system
- Six primary subsystems:
  - Process management
  - Interprocess communication
  - Memory management
  - File system management
    - VFS: provides a single interface to multiple file systems
  - I/O management
  - Networking

20.4 Kernel Architecture

Figure 20.1 Linux architecture.



20.4.1 Hardware Platforms

- Supports a large number of platforms, including
  - x86 (including Intel IA-32), HP/Compaq Alpha AXP, Sun SPARC, Sun UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390 and zSeries, MIPS, HP PA-RISC, Intel IA-64,AMD x86-64,H8/300,V850 and CRIS.
- Architecture-specific code
  - Performs operations implemented differently across platforms
- Porting
  - Modifying the kernel to support a new platform
- Source tree
  - Loosely organizes kernel into separate components by directory
- User-mode Linux (UML)
  - Important tool for kernel development

20.4.2 Loadable Kernel Modules

- Loadable kernel modules
  - Contains object code that, when loaded, is dynamically linked to a running kernel
  - Enables code to be loaded on demand
    - Reduces the kernel’s memory footprint
  - Modules written for versions of the kernel other than the current one may not work properly
  - *Kmod*: a kernel subsystem that manages modules without user intervention
    - Determines module dependencies and loads them on demand

20.5 Process Management

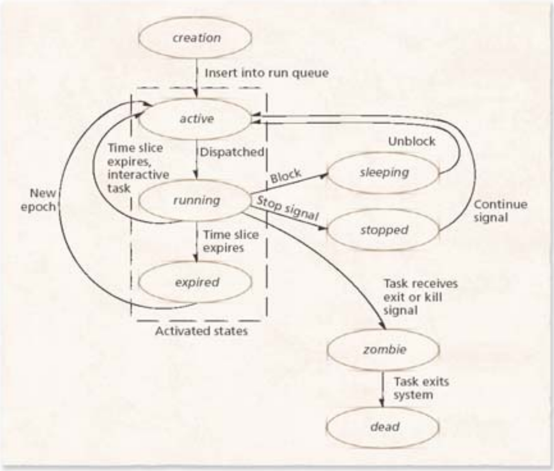
- Process manager
  - Responsible primarily for allocating processors to processes
  - Also delivers signals, loads kernel modules and receives interrupts

20.5.1 Process and Thread Organization

- Tasks
  - Processes and thread are internally represented as tasks using the task_struct structure
- Process manager maintains references to processes using a circular, doubly linked list and a hash table
- Task states:
  - *Running*
  - *Sleeping*
  - *Zombie*
  - *Dead*
  - *Stopped*
  - *Active and expired* (not stored by state)

20.5.1 Process and Thread Organization

Figure 20.2 Task state-transition diagram.



20.5.1 Process and Thread Organization

- *init*
  - Uses the kernel to create all other tasks
    - The clone system call creates new tasks
    - The fork system call creates tasks that initially share their parent's address space using copy-on-write (analogous to processes)
    - When a process issues a clone system call, it can specify which data structures to share with its parent
      - If address space is shared, clone creates a traditional thread
      - If clone is called from a kernel process, resulting thread, called a kernel thread, share's the kernel's address space
    - Although less portable than Pthreads, Linux threads can facilitate programming and lead to more efficient applications
      - Native POSIX Thread Library (NPTL) conforms to POSIX

20.5.2 Process Scheduling

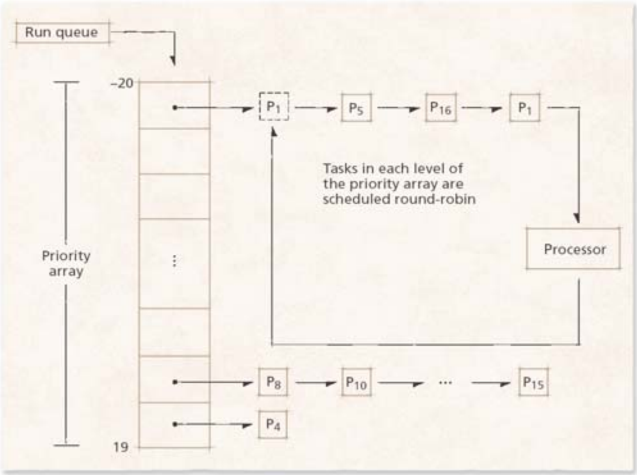
- Scheduler goals:
  - Run all tasks within a reasonable amount of time
  - Respecting task priorities
  - Maintaining high resource utilization
  - High throughput
  - Reducing the overhead of scheduling operations
  - Scale to high-end systems
- All scheduling operations execute in constant time
  - Improve scalability because execution time independed of number of tasks in the system

20.5.2 Process Scheduling

- Preemptive scheduler
  - Each task runs until its quantum, or time slice, expires, a higher priority process becomes runnable or the process blocks
  - Tasks placed in run queues (similar to multilevel feedback queues)
  - Priority array maintains pointer to each level of the run queue
    - Task of priority *i* is placed in the *i*th entry of the priority array for a run queue
  - Scheduler dispatches the task at the front of the list in the highest level of the priority array
    - If more than one task exists in a level of the priority array, tasks are dispatched from the priority array round-robin
    - When a task enters the *blocked* or *sleeping* (i.e., *waiting*) state, or is otherwise unable to execute, that task is removed from its run queue

20.5.2 Process Scheduling

Figure 20.3 Scheduler priority array.


Run queue
–20
P1
P5
P16
P1
Tasks in each level of the priority array are scheduled round-robin
Processor
Priority array
⋮
P8
P10
...
P15
P4
19

20.5.2 Process Scheduling

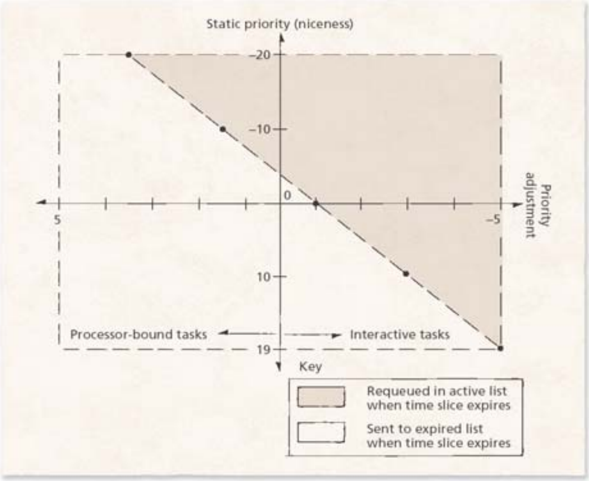
- To prevent indefinite postponement, each task executes once per epoch
  - Epoch defined by starvation limit, derived empirically
  - Scheduler organizes tasks into active and expired lists
    - Only tasks in the active list can be dispatched
    - When starvation limit is reached, every task is placed in the expired list after its quantum expires
      - Guarantees that low-priority tasks will run eventually

20.5.2 Process Scheduling

- Priority
  - Each task is assigned a static priority (also called nice value)
  - Tasks are scheduled according to their effective priority
    - I/O-bound tasks receive boost
    - Processor-bound tasks are penalized with lower priority
  - A high-priority task can be rescheduled after their time slice expires

20.5.2 Process Scheduling

Figure 20.4 Priority values and their corresponding levels of interactivity.



20.5.2 Process Scheduling

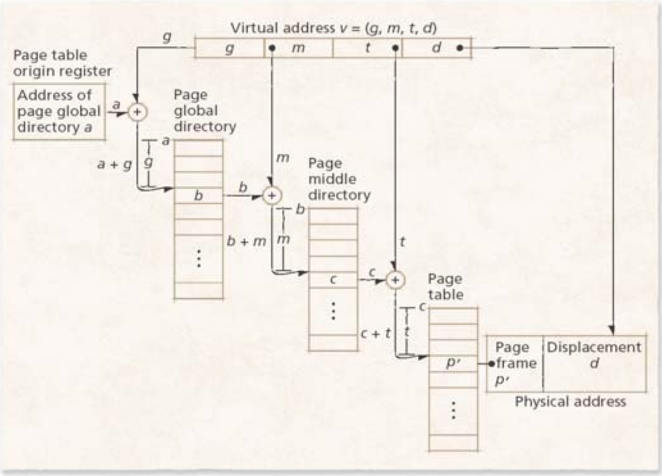
- Multiprocessor scheduling
  - Scheduler performs dynamic load balancing
  - Attempts to reduce load imbalance, not perfectly blance run queues
  - Attempts to migrate only cache-cold tasks
- Real-time scheduling
  - Soft real-time scheduler
  - RT tasks can specify round-robin, FIFO or default scheduling policy
  - RT tasks are always rescheduled after quantum expiration
  - RT tasks can be created only by users with root privileges

20.6 Memory Management

- Memory manager supports 32- and 64-bit addresses
- Also supports NUMA

20.6.1 Memory Organization

Figure 20.5 Page table organization.



20.6.1 Memory Organization

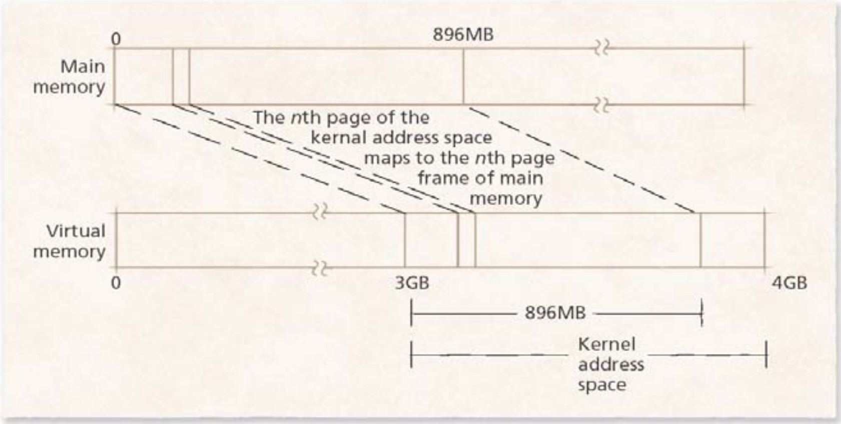
- Linux uses paging exclusively
  - Often implemented using a single page size
  - On 32-bit systems, kernel can address 4GB of data
    - On 64-bit systems, the kernel supports up to 2 petabytes of data
  - Three levels of page tables
    - Page global directory
    - Page middle directory
    - Page tables
  - On systems that support only two levels of page tables, page middle directory contains exactly one entry
- Virtual address space organized into virtual memory areas to group information with same permissions (similar to segments)

20.6.1 Memory Organization

- Linux on the IA-32 architecture
  - Kernel attempts to reduce overhead due to TLB flushing on context switch
  - Divides each 4GB address space into a 3GB region for process data and instructions and a 1GB address space for kernel data and instructions
  - Most of the kernel’s address space is directly mapped to main memory so that it can access information belonging to any process

20.6.1 Memory Organization

Figure 20.6 Kernel virtual address space mapping.

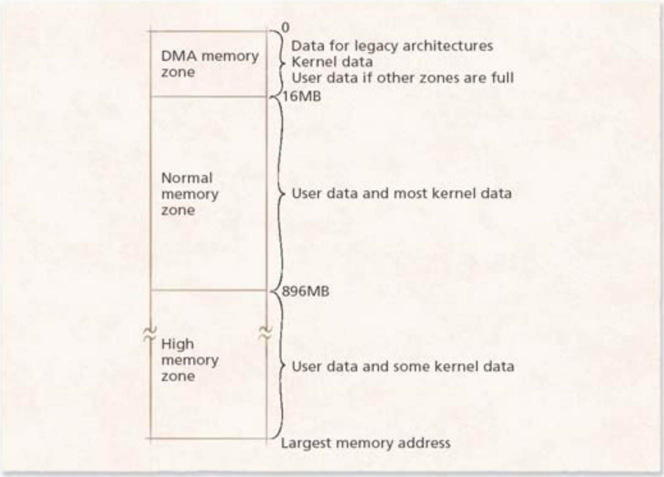

0
896MB
Main memory
The *n*th page of the kernal address space maps to the *n*th page frame of main memory
Virtual memory
0
3GB
4GB
896MB
Kernel address space

20.6.1 Memory Organization

- Memory zones
  - DMA memory: first 16MB of main memory
    - Kernel attempts to make memory available in this region for legacy hardware
  - Normal memory: between 16MB and 896MB on the IA-32 architecture
    - Stores user data and most kernel data
  - High memory: > 896MB on the IA-32 architecture
    - Contains memory that the kernel does not permanently map to its address space
- Bounce buffer
  - Allocates low memory temporarily for I/O
  - Data is "bounced" to high memory after I/O completes

20.6.1 Memory Organization

Figure 20.7 Physical memory zones on the IA-32 Intel architecture.

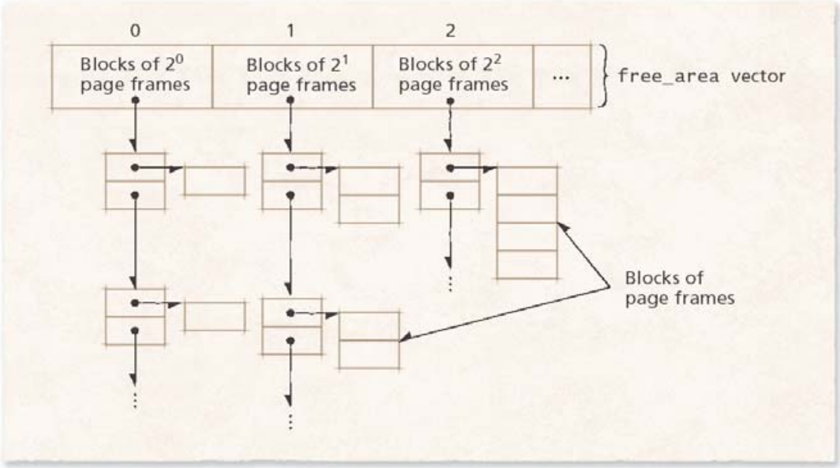

0
DMA memory zone
Data for legacy architectures
Kernel data
User data if other zones are full
16MB
Normal memory zone
User data and most kernel data
896MB
High memory zone
User data and some kernel data
Largest memory address

20.6.2 Physical Memory Allocation and Deallocation

- Zone allocator
  - Allocates frames to processes from high memory, if available
    - Otherwise, allocates from normal memory, if available
    - Allocates from low memory if no other memory is available
  - Uses the binary buddy algorithm to find blocks of contiguous page frames of appropriate size for the process
- Slab allocator
  - Allocates memory for structures smaller than a page
- Memory pool
  - Region of memory that the kernel guarantees will be available to a kernel thread or device driver regardless of memory load

20.6.2 Physical Memory Allocation and Deallocation

Figure 20.8 `free_area` vector.


0
1
2
Blocks of 2^0 page frames
Blocks of 2^1 page frames
Blocks of 2^2 page frames
...
free_area vector
Blocks of page frames

20.6.3 Page Replacement

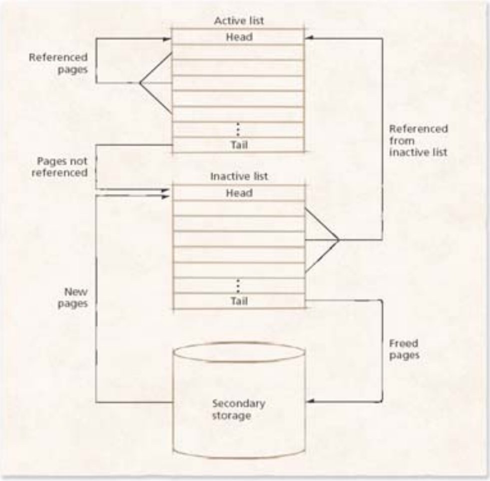
- General characteristics
  - Only user pages can be replaced
  - As pages are read into memory, the kernel inserts them into the page cache
    - Dirty pages flushed to disk using write-back caching
    - System swap file
      - Stores program data and procedure pages on secondary storage

20.6.3 Page Replacement

- Page replacement is performed independently for each page zone
  - Algorithm is a variant of the clock page-replacement algorithm
    - Two linked lists per zone
      - Active list contains pages that have been referenced recently
      - Inactive list contains pages that have been used less recently
    - Page enters system at the head of the inactive list, referenced bit set
    - If the page is active or inactive and its referenced bit is off, the bit is turned on
      - Ensures that recently referenced pages are not selected for replacement
    - If page is inactive and is being referenced for the second time (referenced bit is on), page is moved to head of the active list, referenced bit is cleared
      - Allows the kernel to distinguish between referenced pages that have been accessed once and those that have been accessed more than once recently
      - The latter are placed in the active list so they are not selected for replacement

20.6.3 Page Replacement

Figure 20.9 Page-replacement system overview.



20.6.4 Swapping

- *kswapd* (the kernel swap daemon)
  - Periodically frees page frames by flushing dirty pages to disk
  - Swaps pages from the tail of the inactive list
    - First determines if the page has a valid entry in the swap cache
      - Enables clean pages to be freed immediately
    - Cannot free a page frame if
      - Page is shared
        - *kswapd* must unmap multiple references to the page
        - Reverse mapping improves efficiency
      - Page is dirty
        - *kswapd* must flush it to disk
        - Performed asynchronously by *pdflush*
      - Page is locked (e.g., currently under I/O)
        - *kswapd* must wait until page is unlocked

20.7 File Systems

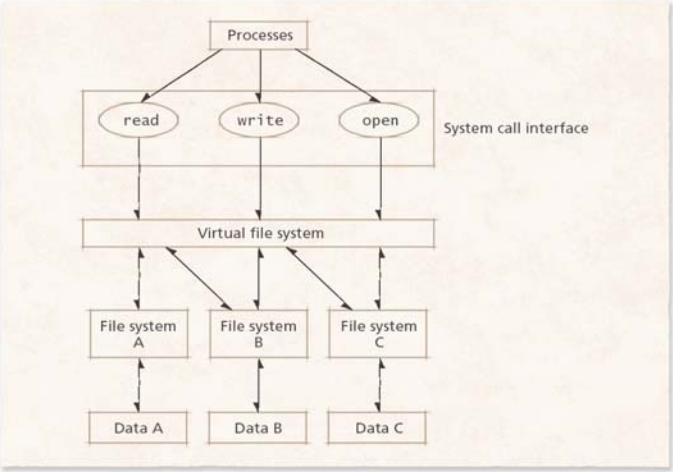
- Each particular file system determines how to store and access its data
- A file refers to more than bits on secondary storage
  - Access points to data, which can be found on a local disk, across a network, or even generated by the kernel itself
  - Enables the kernel to access hardware devices, interprocess communication mechanisms, data stored on disk and a variety of other data sources using a single generic file system interface
- Kernel supports more than 40 file systems

20.7.1 Virtual File System

- VFS
  - Abstracts the details of file access, allowing users to view all the files and directories in the system under a single directory tree
  - All file-related requests are initially sent to the VFS layer, which provides an interface to access file data on any available file system
  - Processes issue system calls such as read, write and open, which are passed to the virtual file system.
    - VFS determines the file system to which the request corresponds and calls the corresponding routines in the file system driver, which perform the requested operations

20.7.1 Virtual File System

Figure 20.10 Relationship between the VFS, file systems and data.

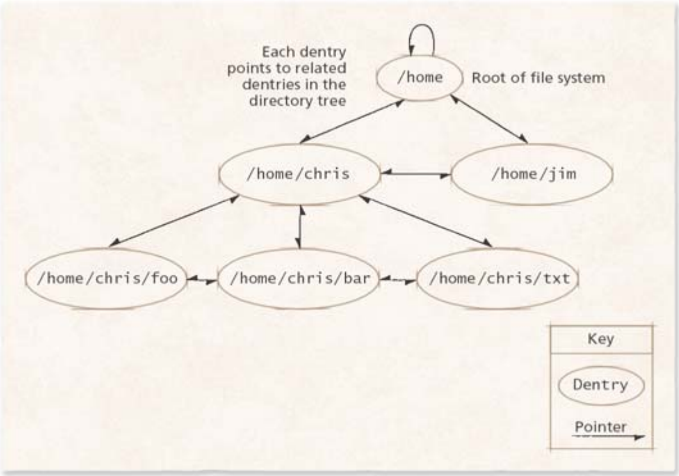


20.7.1 Virtual File System

- VFS inode
  - Describes the location of each file, directory or link within every available file system
  - Reference each file by an inode number and file system number
- File descriptor
  - Contains:
    - Information about the inode being accessed
    - Information about the position in the file being accessed
    - Flags describing how the data is being accessed (e.g. read/write, append-only)
- Dentry (directory entry)
  - Maps file descriptors to inodes
  - Contains the name of the file or directory an inode represents

20.7.1 Virtual File System

Figure 20.11 Dentry organization for a particular /home directory.

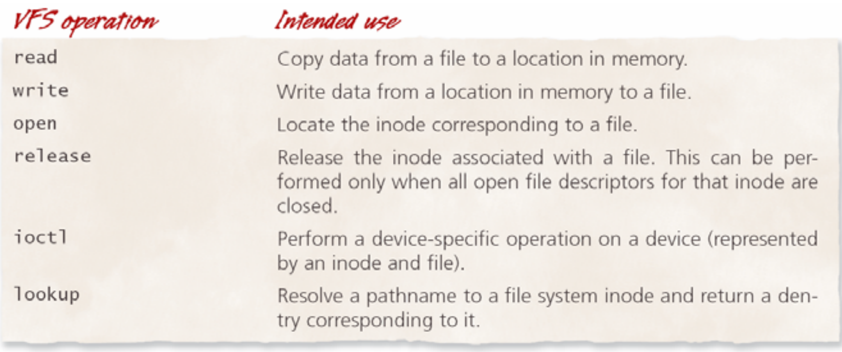


20.7.1 Virtual File System

- VFS superblock
  - Contains information about a mounted file system, such as
    - The type of file system
    - Its root inode's location on disk
    - Housekeeping information that protects the integrity of the file system
  - Stored exclusively in main memory, created when FS is mounted
- The VFS defines generic file system operations
  - Requires that each file system provide an implementation for each operation it supports
  - For example, the VFS defines a read function, but does not implement it

20.7.1 Virtual File System

Figure 20.12 VFS file and inode operations.



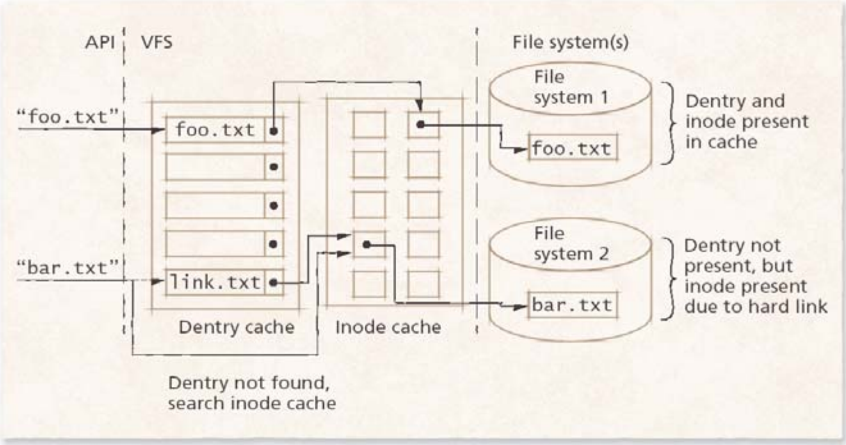
VFS operation	*Intended use*
`read`	Copy data from a file to a location in memory.
`write`	Write data from a location in memory to a file.
`open`	Locate the inode corresponding to a file.
`release`	Release the inode associated with a file. This can be performed only when all open file descriptors for that inode are closed.
`ioctl`	Perform a device-specific operation on a device (represented by an inode and file).
`lookup`	Resolve a pathname to a file system inode and return a dentry corresponding to it.

20.7.2 Virtual File System Caches

- dcache (directory entry cache)
  - Contains dentries corresponding to directories that have recently been accessed
  - Allows the kernel to quickly perform a pathname-to-inode translation if the file specified by the pathname is located in main memory
- Inode cache
  - Contains inodes corresponding to dentries in the dcache
- To perform pathname-to-inode conversions quickly, dcache is searched first, then inode cache

20.7.2 Virtual File System Caches

Figure 20.13 Dentry and inode caches.


API
VFS
File system(s)
"foo.txt"
foo.txt
File system 1
foo.txt
Dentry and inode present in cache
"bar.txt"
link.txt
File system 2
bar.txt
Dentry not present, but inode present due to hard link
Dentry cache
Inode cache
Dentry not found, search inode cache

20.7.3 Second Extended File System (ext2fs)

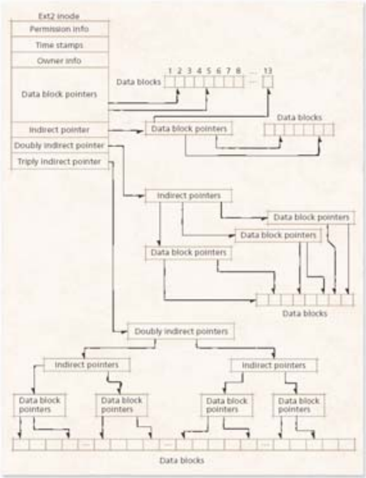
- Ext2 characteristics
  - Goal: high-performance, robust file system with support for advanced features
  - Typical block sizes are 1,024, 2,048, 4,096 or 8,192 bytes
  - By default, five percent of the blocks are reserved exclusively for users with root privileges when the disk is formatted
    - Safety mechanism provided to allow root processes to continue to run if a malicious or errant user process consumes all other available blocks in the file system

20.7.3 Second Extended File System (ext2fs)

- ext2 inode
  - Represents files and directories in an ext2 file system
  - Stores information relevant to a single file or directory, such as time stamps, permissions, the identity of the file's owner and pointers to data blocks
    - First 12 pointers directly locate the first 12 data blocks
    - 13th pointer is an indirect pointer
      - locates a block that contains pointers to data blocks
    - 14th pointer is a doubly indirect pointer
      - locates a block of indirect pointers.
    - 15th pointer is a triply indirect pointer
      - locates a block of doubly indirect pointers
  - Provides fast access to small files, while supporting larger files

20.7.3 Second Extended File System (ext2fs)

Figure 20.14 Ext2 inode contents.

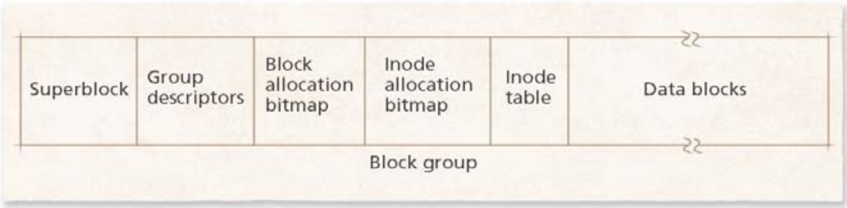


20.7.3 Second Extended File System (ext2fs)

- Block groups
  - Clusters of contiguous blocks
  - File system attempts to store related data in the same block group
  - Reduces the seek time for accessing large groups of related data
  - Contains
    - The superblock
      - Critical information about the entire file system, not just a particular block group.
        - Includes the total number of blocks and inodes in the file system, the size of the block groups, the time at which the file system was mounted and other housekeeping data
        - Redundant copy of the superblock is maintained in some block groups
    - Inode table
      - Contains an entry for each inode in the block group
    - Inode allocation bitmap
      - Tracks inode use within a block group

20.7.3 Second Extended File System (ext2fs)

Figure 20.15 Block group.

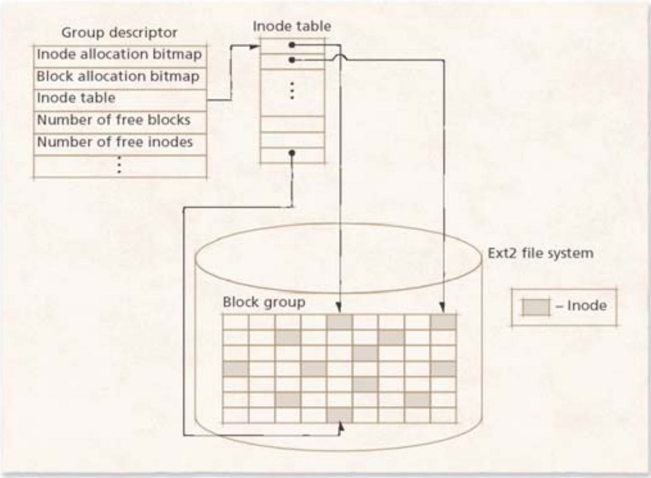


20.7.3 Second Extended File System (ext2fs)

- Block groups (cont.)
  - Contains
    - Block allocation bitmaps
      - Track each group’s block usage
    - Group descriptor
      - Contains the block numbers corresponding to the location of the inode allocation bitmap, block allocation bitmap and inode table, accounting information

20.7.3 Second Extended File System (ext2fs)

Figure 20.16 Group descriptor.

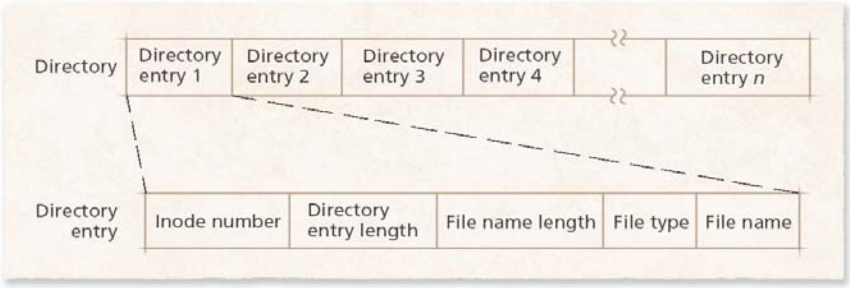


20.7.3 Second Extended File System (ext2fs)

- Block groups (cont.)
  - Contains
    - Remaining blocks in each block group store file/directory data
      - Directory information stored in directory entries
        - Each directory entry is composed of an inode number, directory entry length, file name length, file type and file name
- File security
  - File permissions
    - Specify read, write and execute privileges for three categories of users
      - Owner, group, other
  - File attributes
    - Control how file data can be modified
    - For example, append-only

20.7.3 Second Extended File System (ext2fs)

Figure 20.17 Directory structure.



Directory

Directory entry 1

Directory entry 2

Directory entry 3

Directory entry 4

Directory entry *n*

Directory entry

Inode number

Directory entry length

File name length

File type

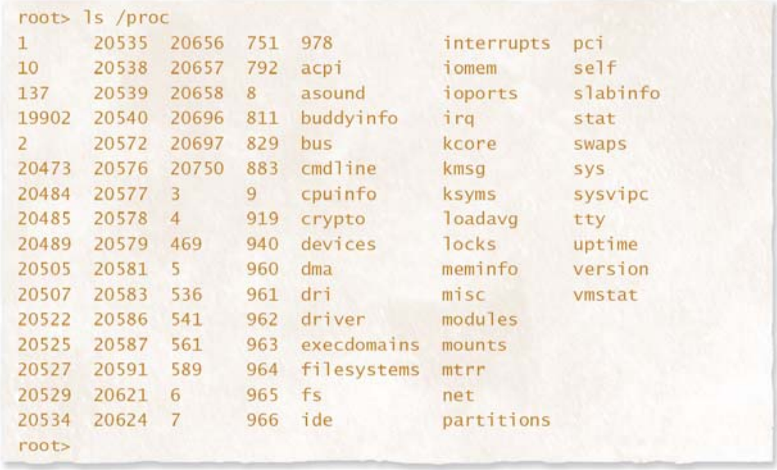
File name

20.7.4 Proc File System

- Procfs
  - Created to provide real-time information about the status of the kernel and processes in a system
  - Enables users to obtain detailed information describing the system, from hardware status information to data describing network traffic
  - Exists only in main memory
    - Proc file data is created on demand
    - Proc read and write calls can access kernel data
      - Enable users to send data to the kernel

20.7.4 Proc File System

Figure 20.18 Sample contents of the `/proc` directory.



```
root> ls /proc
1      20535  20656  751  978          interrupts  pci
10     20538  20657  792  acpi         iomem       self
137    20539  20658  8    asound       ioports     slabinfo
19902  20540  20696  811  buddyinfo    irq         stat
2      20572  20697  829  bus          kcore       swaps
20473  20576  20750  883  cmdline      kmsg        sys
20484  20577  3      9    cpuinfo      ksyms       sysvipc
20485  20578  4      919  crypto       loadavg     tty
20489  20579  469    940  devices      locks       uptime
20505  20581  5      960  dma          meminfo     version
20507  20583  536    961  dri          misc        vmstat
20522  20586  541    962  driver       modules
20525  20587  561    963  execdomains  mounts
20527  20591  589    964  filesystems  mtrr
20529  20621  6      965  fs           net
20534  20624  7      966  ide          partitions
root>
```

20.8 Input/Output Management

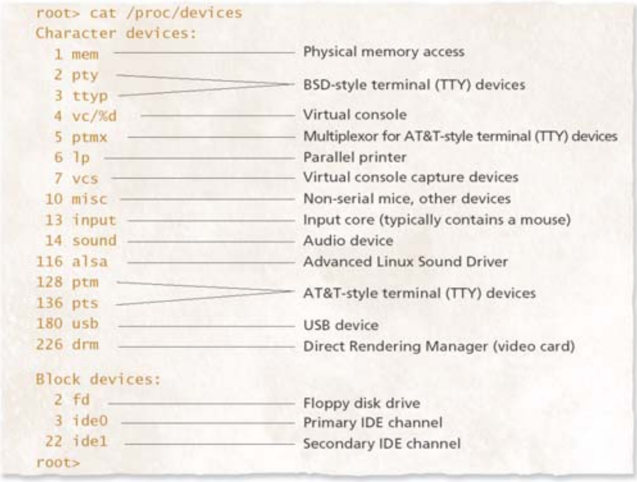
- Kernel provides common interface for I/O system calls
- Devices are grouped into classes
  - Members of each device class perform similar functions
  - Allows the kernel to address the performance needs of certain devices (or classes of devices) individually

20.8.1 Device Drivers

- Device driver: software interface between system calls and a hardware device
  - Most have been written by independent developers
  - Typically implemented as loadable kernel modules
- Device special files
  - Most devices are represented by device special files
  - Entries in the /dev directory that provide access to devices
  - List of devices in the system can be obtained by reading the contents of /proc/devices

20.8.1 Device Drivers

Figure 20.19 `/proc/devices` file contents.



```
root> cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 vc/%d
  5 ptmx
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
116 alsa
128 ptm
136 pts
180 usb
226 drm

Block devices:
  2 fd
  3 ide0
 22 ide1
root>
```

Physical memory access
BSD-style terminal (TTY) devices
Virtual console
Multiplexor for AT&T-style terminal (TTY) devices
Parallel printer
Virtual console capture devices
Non-serial mice, other devices
Input core (typically contains a mouse)
Audio device
Advanced Linux Sound Driver
AT&T-style terminal (TTY) devices
USB device
Direct Rendering Manager (video card)
Floppy disk drive
Primary IDE channel
Secondary IDE channel

20.8.1 Device Drivers

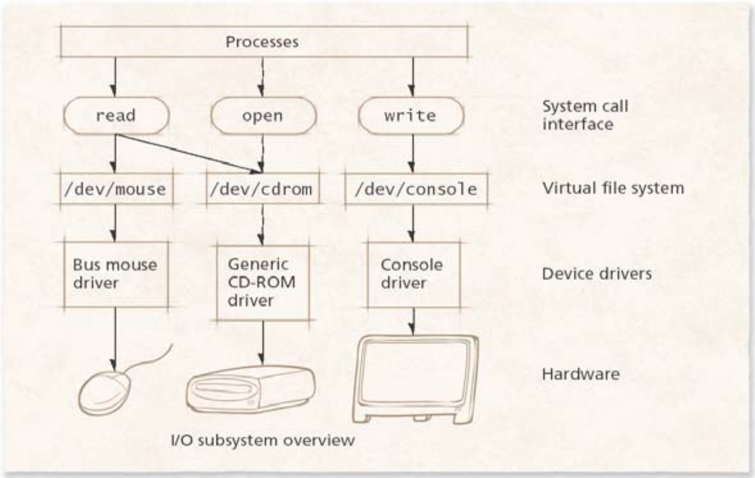
- Device classes
  - Groups devices that perform similar functions
- Major and minor identification numbers
  - Used by device drivers to identify their devices
  - Devices that are assigned the same major identification number are controlled by the same driver
  - Minor identification numbers enable the system to distinguish between devices of the same class

20.8.1 Device Drivers

- Device special files are accessed via the virtual file system
  - System calls pass to the VFS, which in turn issues calls to device drivers
  - Most drivers implement common file operations such as read, write and seek
  - To support tasks such as ejecting a CD-ROM tray or retrieving status information from a printer, Linux provides the ioctl system call

20.8.1 Device Drivers

Figure 20.20 I/O interface layers.

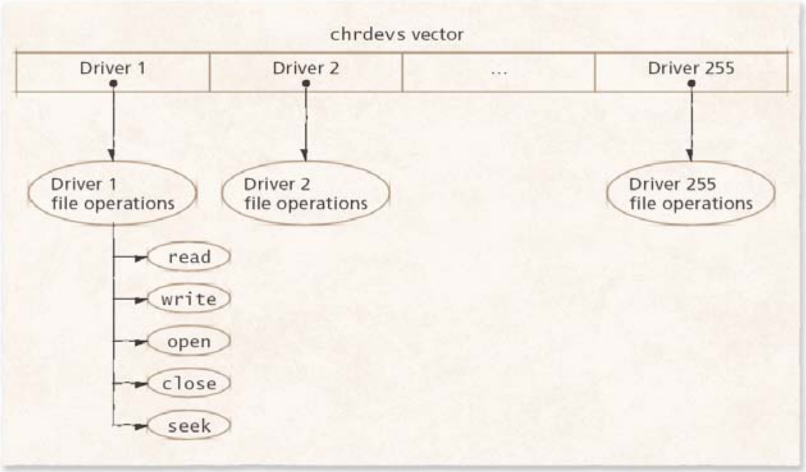


20.8.2 Character Device I/O

- Character device
  - Transmits data as a stream of bytes
  - Represented by a device_struct structure that contains the driver name and a pointer to the driver's file_operations structure
    - Maintains the operations supported by the device driver
  - All registered drivers are referenced by the chrdevs vector
- The file_operations structure
  - Stores functions called by the VFS when a system call accesses a device special file

20.8.2 Character Device I/O

Figure 20.21 `chrdevs` vector.

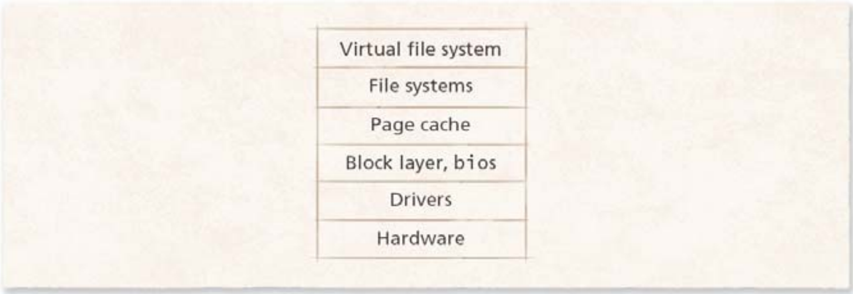


20.8.3 Block Device I/O

- Block I/O subsystem
  - Block devices are identified by major and minor numbers
  - The kernel's block I/O subsystem contains a number of layers to modularize block I/O operations by placing common code in each layer.
  - To minimize the amount of time spent accessing block devices, the kernel uses two primary strategies
    - Caching data
    - Clustering I/O operations

20.8.3 Block Device I/O

Figure 20.22 Block I/O subsystem layers.



20.8.3 Block Device I/O

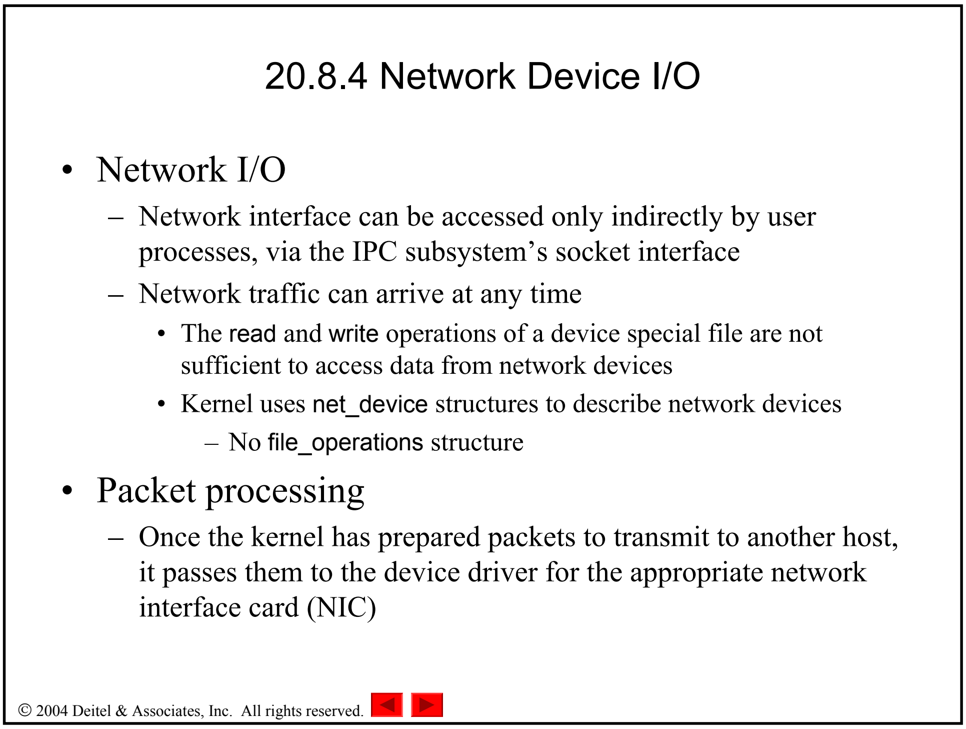
- When data from a block device is requested, kernel first searches page cache
  - If found, data is copied to the process's address space
  - Otherwise, typically added to a request queue
  - Direct I/O
    - Enables driver to bypass kernel caches when accessing devices
    - Important for databases and other applications where kernel caching is inappropriate and may reduce performance/consistency

20.8.3 Block Device I/O

- Disk scheduling
  - Performed by placing I/O requests in a request list
  - One request list for each device in the system
  - bio structure: maps to a number of page frames corresponding to a request
  - Block device drivers define a request operation that is called by the kernel
    - Kernel passes an ordered request list and the driver must perform all operations in the list
    - Device drivers do not define read and write operations
  - Some devices drivers, such as ones for RAID order their own requests and therefore bypass the kernel for request list ordering

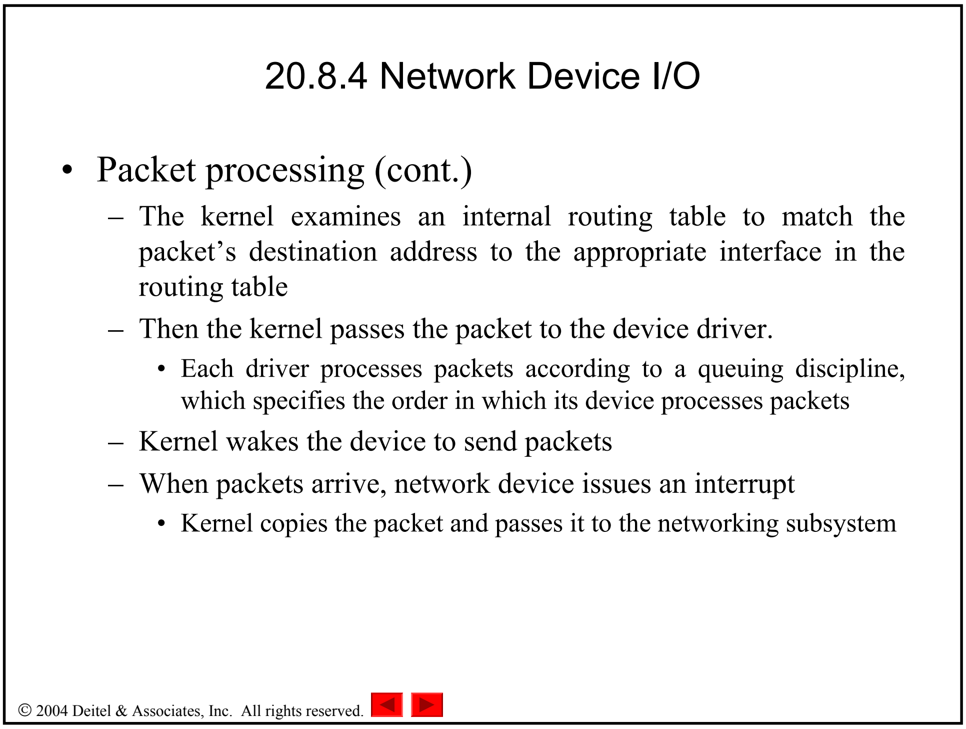
20.8.3 Block Device I/O

- Disk scheduling algorithms
  - Default is the LOOK algorithm
    - Can lead to synchronous read request starvation during concurrent writes
  - Kernel attempts to merge requests to adjacent blocks
  - Deadline scheduler
    - Eliminates read request starvation by ensuring all read operations are performed by a certain deadline
  - Anticipatory scheduler
    - Attempts to eliminate read request starvation and reduce excessive seeking behavior by delaying after a read request completes
    - The idea is that the process will issue another synchronous read operation before its quantum expires
    - Can lead to reduced throughput if process does not issue another read request to a nearby location



20.8.4 Network Device I/O

- Network I/O
  - Network interface can be accessed only indirectly by user processes, via the IPC subsystem's socket interface
  - Network traffic can arrive at any time
    - The read and write operations of a device special file are not sufficient to access data from network devices
    - Kernel uses net_device structures to describe network devices
      - No file_operations structure
- Packet processing
  - Once the kernel has prepared packets to transmit to another host, it passes them to the device driver for the appropriate network interface card (NIC)



20.8.4 Network Device I/O

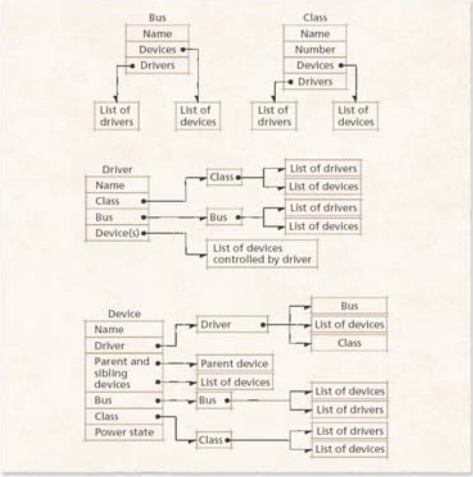
- Packet processing (cont.)
  - The kernel examines an internal routing table to match the packet's destination address to the appropriate interface in the routing table
  - Then the kernel passes the packet to the device driver.
    - Each driver processes packets according to a queuing discipline, which specifies the order in which its device processes packets
  - Kernel wakes the device to send packets
  - When packets arrive, network device issues an interrupt
    - Kernel copies the packet and passes it to the networking subsystem

20.8.5 Unified Device Model

- UDM
  - Attempts to simplify device management in the kernel
  - Relates device classes to system buses
    - Helps support hot-swappable devices
    - Power management
  - UDM defines structures to represent devices, device drivers, device classes, buses
  - Sysfs (system file system, located at /sys)
    - Provides interface to access information about devices in the UDM

20.8.5 Unified Device Model

Figure 20.23 Unified device model organization.



20.8.6 Interrupts

- Interrupt processing
  - When the kernel receives an interrupt from a particular device, the kernel passes control to its corresponding interrupt handler
  - Interrupt handlers do not belong to any single process context
    - Scheduler cannot place an interrupt handler in a run queue
    - Thus, interrupt handler cannot sleep, call the scheduler, be preempted or raise faults or exceptions
    - As a result, most interrupt handlers are designed to execute quickly

20.8.6 Interrupts

- Interrupts are divided into top and bottom halves
  - Top half runs as described in preceding slide
  - Bottom half is scheduled to be performed by a software interrupt handler later
    - Softirqs
      - Can be scheduled simultaneously on multiple processors
      - Must contain reentrant code
    - Tasklets
      - Serialized
      - More commonly implemented—few interrupt handlers can benefit from parallel processing
  - Software interrupts typically are handled in interrupt context or in a process’s context, executing with higher priority than other processes
    - If interrupt load is high, user processes could be indefinitely postponed
      - *ksoftirq* schedules tasklets and softirqs to be executed later

20.9 Kernel Synchronization

- Kernel control paths
  - Code that directly accesses kernel data, hardware, or other critical system resources
  - If two kernel control paths were to access the same data concurrently, a race condition could result
  - To prevent this, the kernel provides two basic mechanisms for providing mutually exclusive access to critical sections: locks and semaphores

20.9.1 Spin Locks

- Spin locks
  - Protect critical sections in kernel control paths on SMP-enabled systems
  - Once a spin lock is acquired, all subsequent requests to the spin lock cause busy waiting (spinning) until the lock is released
  - Unnecessary in uniprocessor systems
  - Several types of spin locks
    - Interrupt handler spin locks: disable interrupts on local processor
    - Bottom-half spin locks: disable software interrupt handlers
    - Others discussed in subsequent slides

20.9.1 Spin Locks

- Preemption lock counter
  - Prevents kernel control paths from being preempted while holding a spin lock
- Preventing deadlock and indefinite postponement with spin locks:
  - If a kernel control path has already acquired a spin lock, the kernel control path must not attempt to acquire the spin lock again before releasing it
  - Similarly, a kernel control path must not sleep while holding a spin lock. If the next task that is scheduled attempts to acquire the spin lock, deadlock will occur

20.9.2 Reader/Writer Locks

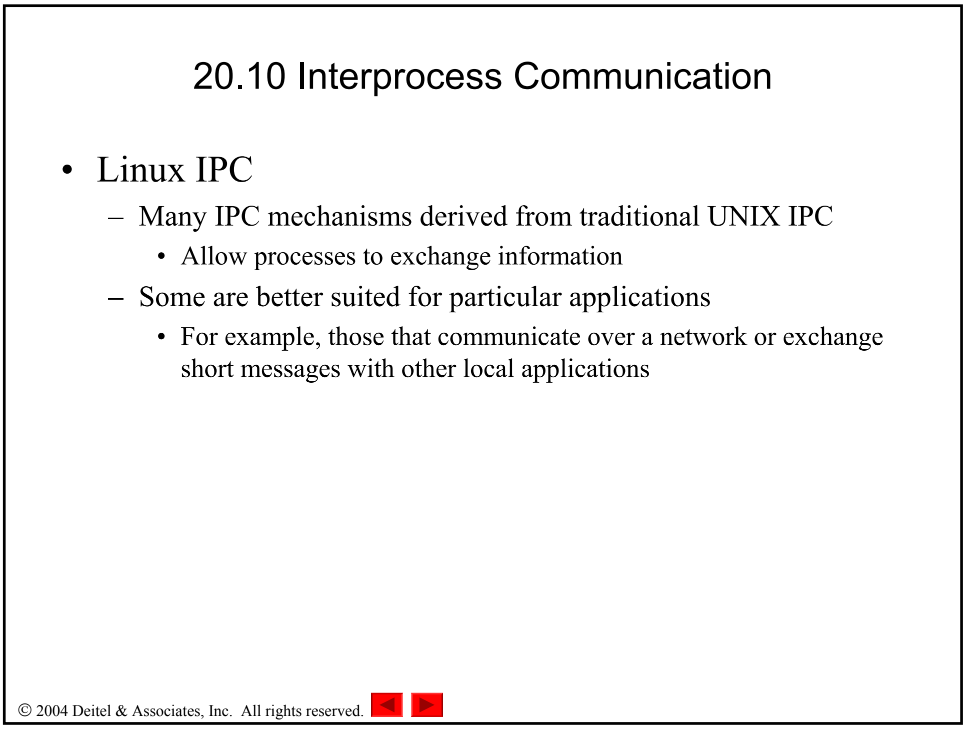
- Reader/writer locks
  - In some cases, multiple kernel control paths need only to read (not write) the data accessed inside a critical section
  - To optimize concurrency in such a situation, the kernel provides reader/writer locks
    - Allow multiple kernel control paths to hold a read lock, but permit only one kernel control path to hold a write lock with no concurrent readers.
    - A kernel control path that holds a read lock on a critical section must release its read lock and acquire a write lock if it wishes to modify data.
    - An attempt to acquire a write lock succeeds only if there are no other readers or writers concurrently executing inside their critical sections.
    - Can lead to improved performance
    - Also possible for writers to be indefinitely postponed.

20.9.3 Seqlocks

- Seqlocks
  - Allow writers to access data immediately without waiting for readers to release the lock
  - Combines spinlock with a sequence counter
  - Requires readers to detect if a writer has modified the value of the data protected by the seqlock by examining the value of the seqlock’s sequence counter
  - Appropriate for interrupt handling

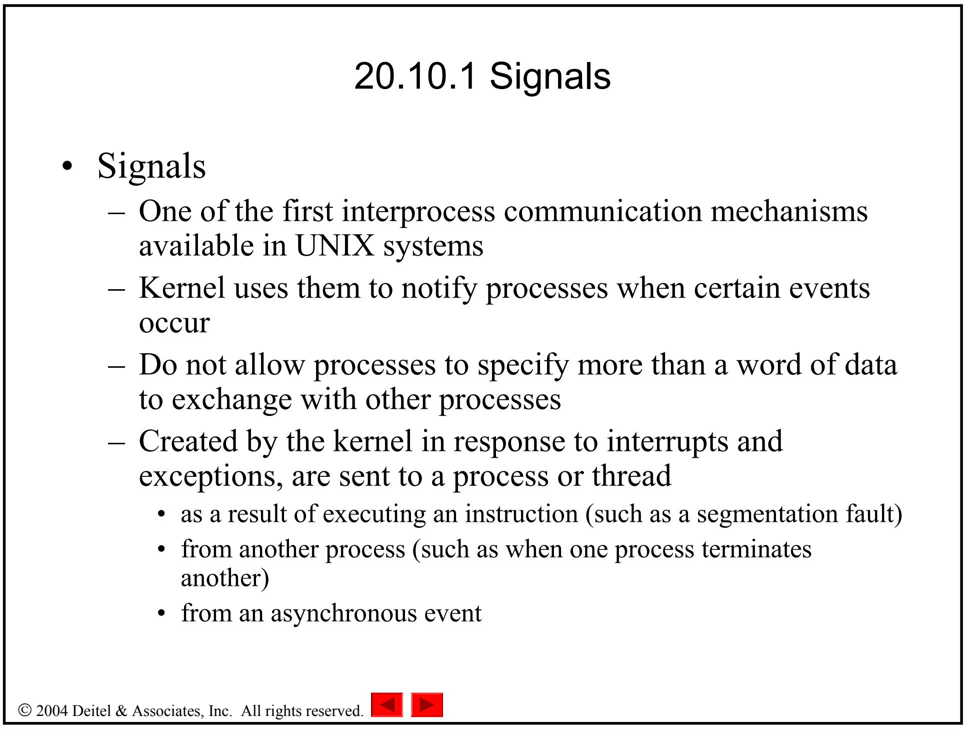
20.9.4 Kernel Semaphores

- Spin locks can be inefficient
  - Busy waiting can lead to inefficiency if kernel control paths must wait for long periods
- Kernel semaphores
  - Implemented as counting semaphores
  - Before entering critical section, must call function down
    - If the value of the counter is greater than 0, decrement the counter, allow the process to execute.
    - If the value of the counter is less than or equal to 0, down decrements the counter, and the process is added to the wait queue and enters the *sleeping* state.
      - Reduces the overhead due to busy waiting
  - When a process exits its critical section, must call function up
    - If the value of the counter is greater than or equal to 0, increments counter
    - If the value of the counter is less than 0, up increments the counter, and a process from the wait queue is awakened to execute its critical section



20.10 Interprocess Communication

- Linux IPC
  - Many IPC mechanisms derived from traditional UNIX IPC
    - Allow processes to exchange information
  - Some are better suited for particular applications
    - For example, those that communicate over a network or exchange short messages with other local applications

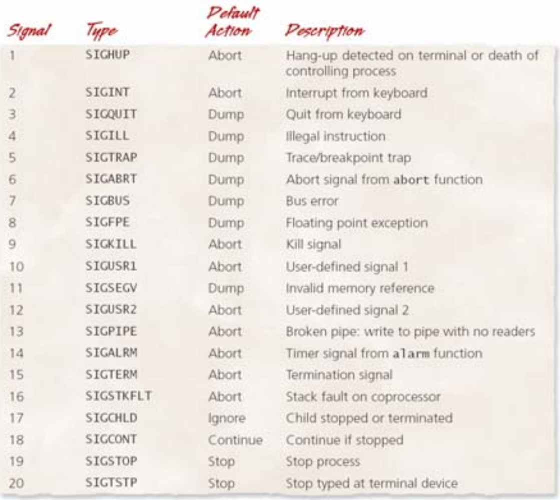


20.10.1 Signals

- Signals
  - One of the first interprocess communication mechanisms available in UNIX systems
  - Kernel uses them to notify processes when certain events occur
  - Do not allow processes to specify more than a word of data to exchange with other processes
  - Created by the kernel in response to interrupts and exceptions, are sent to a process or thread
    - as a result of executing an instruction (such as a segmentation fault)
    - from another process (such as when one process terminates another)
    - from an asynchronous event

20.10.1 Signals

Figure 20.24 POSIX signals.



Signal	Type	Default Action	Description
1	SIGHUP	Abort	Hang-up detected on terminal or death of controlling process
2	SIGINT	Abort	Interrupt from keyboard
3	SIGQUIT	Dump	Quit from keyboard
4	SIGILL	Dump	Illegal instruction
5	SIGTRAP	Dump	Trace/breakpoint trap
6	SIGABRT	Dump	Abort signal from `abort` function
7	SIGBUS	Dump	Bus error
8	SIGFPE	Dump	Floating point exception
9	SIGKILL	Abort	Kill signal
10	SIGUSR1	Abort	User-defined signal 1
11	SIGSEGV	Dump	Invalid memory reference
12	SIGUSR2	Abort	User-defined signal 2
13	SIGPIPE	Abort	Broken pipe: write to pipe with no readers
14	SIGALRM	Abort	Timer signal from `alarm` function
15	SIGTERM	Abort	Termination signal
16	SIGSTKFLT	Abort	Stack fault on coprocessor
17	SIGCHLD	Ignore	Child stopped or terminated
18	SIGCONT	Continue	Continue if stopped
19	SIGSTOP	Stop	Stop process
20	SIGTSTP	Stop	Stop typed at terminal device

20.10.1 Signals

- A process/thread can handle a signal by
  1. Ignore the signal—processes can ignore all but the SIGSTOP and SIGKILL signals.
  2. Catch the signal—when a process catches a signal, it invokes its signal handler to respond to the signal.
  3. Execute the default action that the kernel defines for that signal
- Default actions
  - Abort: terminate immediately
  - Memory dump: Copies execution context before exiting
  - Ignore
  - Stop (i.e., suspend)
  - Continue (i.e., resume)

20.10.1 Signals

- Signal blocking
  - A process or thread can block a signal
    - Signal is not delivered until process/thread stops blocking it
  - While a signal handler is running, signals of that type are blocked by default
    - Still possible to receive signals of a different type
  - Common signals are not queued
    - Real-time signals provide signal queuing

20.10.2 Pipes

- Pipes
  - Producer process writes data to the pipe, after which the consumer process reads data from the pipe in first-in-first-out order
  - When pipe is created, an inode that points to pipe buffer (page of data) is created
  - Access to pipes is controlled by file descriptors
    - Can be passed between related processes (e.g., parent and child)
  - Named pipes (FIFOs)
    - Can be accessed via the directory tree
  - Limitation: Fixed-size buffer

20.10.3 Sockets

- Sockets
  - Allows pairs of processes to exchange data by establishing direct bidirectional communication channels
  - Primarily used for bidirectional communication between multiple processes on different systems, but can be used for processes on the same system
  - Stored internally as files
  - File name used as socket's address, accessed via the VFS

20.10.3 Sockets

- Stream sockets
  - Implement the traditional client/server model
  - Data is transferred as a stream of bytes
  - Use TCP to communicate, so they are more appropriate for reliable communication
- Datagram sockets
  - Faster, but less reliable communication
  - Data is transferred using datagram packets
- Socketpairs
  - Pair of connected, unnamed sockets
  - Limited to use by processes that share file descriptors

20.10.4 Message Queues

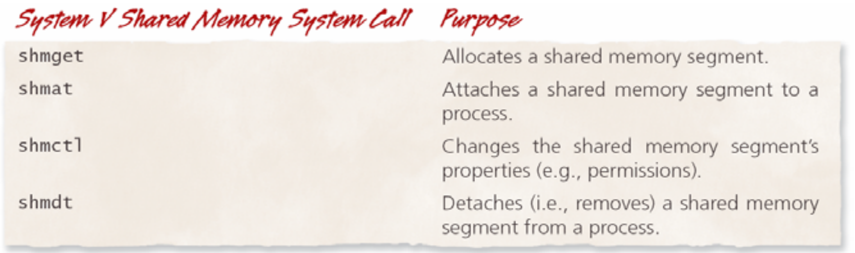
- Message queues
  - Allow processes to transmit information that is composed of a message type and a variable-length data area
    - Stored in message queues, remain until a process is ready to receive them
    - Related processes can search for a message queue identifier in a global array of message queue descriptors
      - Message queue descriptor contains
        - Queue of pending messages
        - Queue of processes waiting for messages
        - Queue of processes waiting to send messages
        - Data describing the size and contents of the message queue

20.10.5 Shared Memory

- Shared memory
  - Advantages
    - Improves performance for processes that frequently access shared data
    - Processes can share as much data as they can address
  - Standard interfaces
    - System V shared memory
    - POSIX shared memory
      - Does not allow processes to change privileges for a segment of shared memory

20.10.5 Shared Memory

Figure 20.25 System V shared memory system calls.



System V Shared Memory System Call	Purpose
`shmget`	Allocates a shared memory segment.
`shmat`	Attaches a shared memory segment to a process.
`shmctl`	Changes the shared memory segment's properties (e.g., permissions).
`shmdt`	Detaches (i.e., removes) a shared memory segment from a process.

20.10.5 Shared Memory

- Shared memory implementation
  - Treats region of shared memory as a file
  - Shared memory page frames are freed when file is deleted
  - Tmpfs (temporary file system) stores such files
    - Tmpfs pages are swappable
    - Permissions can be set
    - File system does not require formatting

20.10.6 System V Semaphores

- System V semaphores
  - Designed for user processes to access via the system call interface
- Semaphore arrays
  - Protect a group of related resources
  - Before a process can access resources protected by a semaphore array, the kernel requires that there be sufficient available resources to satisfy the process's request
  - Otherwise, kernel blocks requesting process until resources become available
- Preventing deadlock
  - When a process exits, the kernel reverses all the semaphore operations it performed to allocate its resources

20.11 Networking

- Networking subsystem
  - Performs operations on network packets
- Packets
  - Stored in a contiguous physical memory area described by an sk_buff structure
  - As a packet traverses layers of the network subsystem, network protocols add and remove headers and trailers containing protocol-specific information

20.11.1 Packet Processing

- Path taken by network packets
  - NIC receives a packet
    - Generates an interrupt
  - Interrupt handler calls the network device’s driver routine
    - Allocates an sk_buff for the packet
    - Copies the packet from the network interface into the sk_buff
    - Adds the packet to a queue of packets pending processing
      - One queue per processor
    - Raises softirq for asynchronous processing

20.11.1 Packet Processing

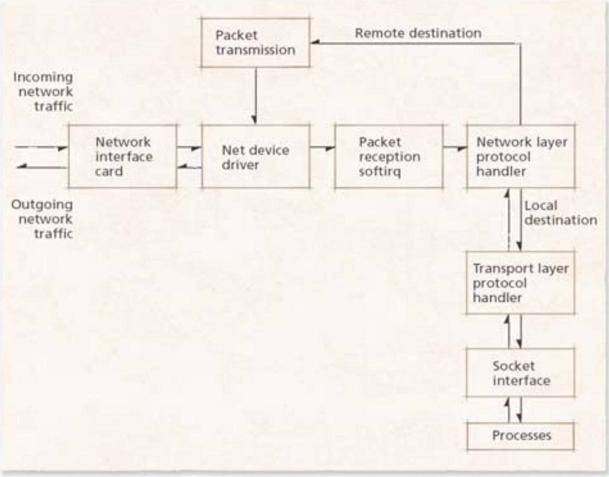
- Path taken by network packets (cont.)
  - One softirq processes all packets in the per-processor queue
    - Can execute simultaneously on multiple processors
    - Processes packets in the processor’s queue until
      - The queue is empty
      - A predefined maximum number of packets are processed
      - A time limit is reached
    - If the latter two occur, the softirq calls the scheduler for rescheduling
    - Softirq processes packet by removing it from the queue and passing it to the IP protocol handler

20.11.1 Packet Processing

- Path taken by network packets (cont.)
  - IP handler
    - Determines destination address
      - If another host, packet is forwarded
      - If local, IP handler strips the IP header from the sk_buff and passes it to the transport layer handler
  - Transport layer handler
    - Can be TCP, UDP or ICMP
    - Determines port number and delivers packet data to the socket that is bound to that port, which passes the data to its associated process

20.11.1 Packet Processing

Figure 20.26 Path followed by packets received by the networking subsystem.

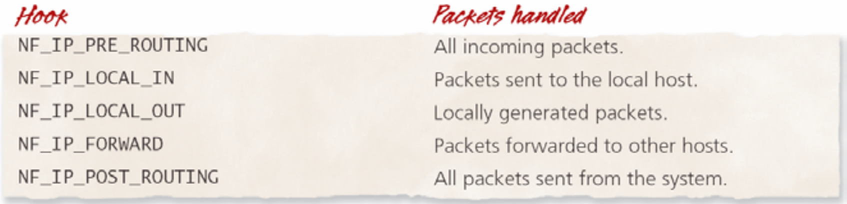


20.11.2 Netfilter Framework and Hooks

- Netfilter framework
  - Mechanism designed to allow kernel modules to directly inspect and modify packets
- Hooks
  - Enable modules to register to examine, alter and/ or discard packets
- Netfilter hooks are placed at various stages of the IP protocol handler

20.11.2 Netfilter Framework and Hooks

Figure 20.27 Netfilter hooks.



Hook	Packets handled
NF_IP_PRE_ROUTING	All incoming packets.
NF_IP_LOCAL_IN	Packets sent to the local host.
NF_IP_LOCAL_OUT	Locally generated packets.
NF_IP_FORWARD	Packets forwarded to other hosts.
NF_IP_POST_ROUTING	All packets sent from the system.

20.12 Scalability

- Scalability
  - Large computer companies have cooperated with independent developers to scale Linux to the high-end server market
  - Designers must decide how far to enable the standard Linux kernel to scale
    - Increasing its scalability might negatively affect its performance on desktop systems and low-end servers
  - Linux companies often provide separate distributions for high-end and desktop systems
  - Embedded-device manufacturers also use Linux to manage their systems

20.12.1 Symmetric Multiprocessing (SMP)

- SMP support
  - Version 2.0 was the first stable kernel release to support SMP systems
  - BKL (big kernel lock)
    - No process on any other processor could execute in kernel mode while BKL was held
    - The OS could not scale effectively to more than four processors
  - Fine-grained locking mechanisms
    - Improved scalability (up to 16 processors in 2.4)
    - Make the kernel more difficult to debug and develop

20.12.1 Symmetric Multiprocessing (SMP)

- Clustering
  - Alternative to SMP systems
  - Easier to develop and scale to large numbers of processors
  - Cheaper
  - Example: Beowulf clusters

20.12.2 Nonuniform Memory Access (NUMA)

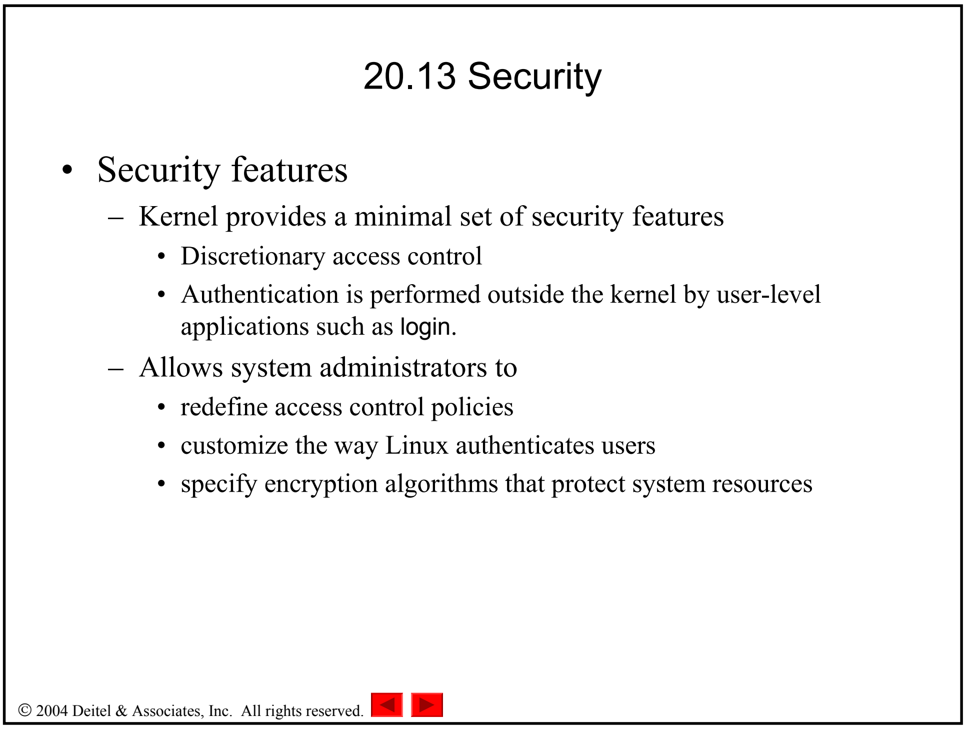
- NUMA support
  - Kernel modified to account for layout of nodes in the system
  - Attempts to allocate resources local to the node of the processor executing a process
  - Scheduler attempts to schedule a process on the same node in which it previous executed
  - Swap routines modified to free only local memory when available local memory is low
  - As of 2.6, still much work to be done
    - For example, no mechanism to migrate a process's pages to local memory if a process is migrated to a remote node

20.12.3 Other Scalability Features

- Increased field sizes
  - Maximum number of users and tasks increased
  - Size of time-tracking jiffies increased to reduce overflows
  - Disk block address size increased to 64 bits
  - Supports PAE for addressing large memories (up to 64GB) using a 32-bit processor
- Preemptible kernel
  - Kernel can be preempted by a high-priority user process
  - Preemption disabled while kernel executing inside a critical section
- Support for 64-bit processors (e.g., Itanium and Opteron)

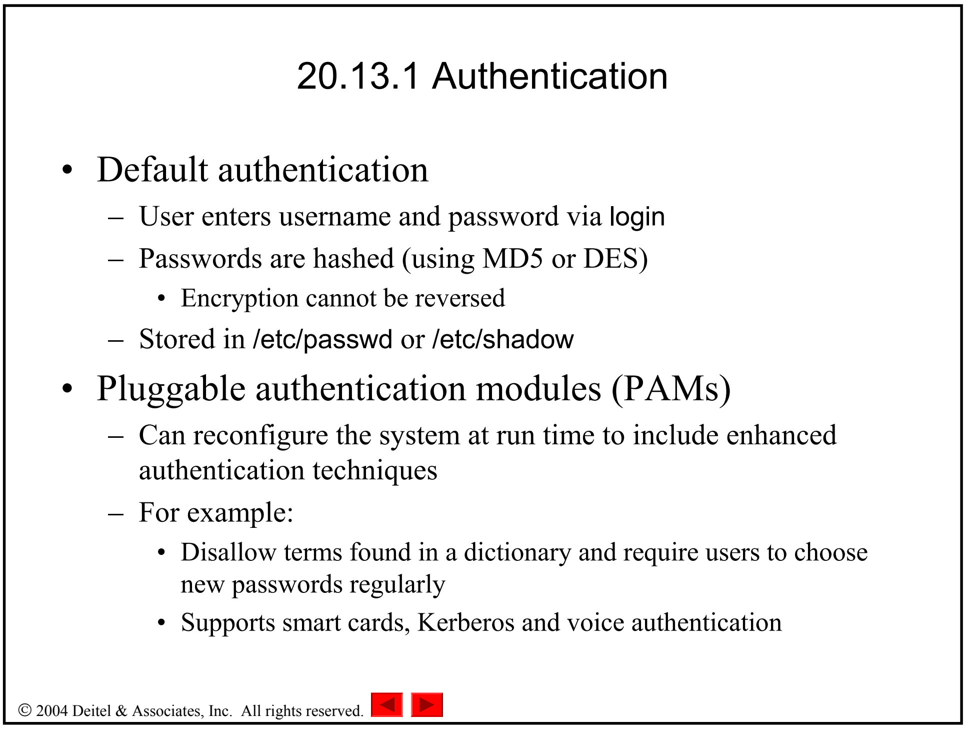
20.12.4 Embedded Linux

- Porting to embedded systems
  - Challenges: limited instruction sets, small memory and secondary storage sizes and devices that are not commonly found in desktops and workstations
  - Hard real-time process scheduling
    - Modify the scheduler to support additional priority levels, deadlines and lower scheduling latency
  - Modifying memory subsystem
    - Reduce the size of the kernel footprint.
    - Perform additional memory management operations (e.g., protection) in software for systems that do not support virtual memory



20.13 Security

- Security features
  - Kernel provides a minimal set of security features
    - Discretionary access control
    - Authentication is performed outside the kernel by user-level applications such as login.
  - Allows system administrators to
    - redefine access control policies
    - customize the way Linux authenticates users
    - specify encryption algorithms that protect system resources



20.13.1 Authentication

- Default authentication
  - User enters username and password via login
  - Passwords are hashed (using MD5 or DES)
    - Encryption cannot be reversed
  - Stored in /etc/passwd or /etc/shadow
- Pluggable authentication modules (PAMs)
  - Can reconfigure the system at run time to include enhanced authentication techniques
  - For example:
    - Disallow terms found in a dictionary and require users to choose new passwords regularly
    - Supports smart cards, Kerberos and voice authentication

20.13.2 Access Control Methods

- Access control attributes
  - Specify file permissions and file attributes
  - File permissions
    - Combination of read, write and/or execute permissions specified for three categories: *user*, *group* and *other*
  - File attributes
    - Additional security mechanism supported by some file systems
    - Allow users to specify constraints on file access beyond read, write and execute
    - Examples: append-only, immutable

20.13.2 Access Control Methods

- Linux security modules (LSM) framework
  - Allows a system administrator to customize the access control policy
  - Uses loadable kernel modules
  - Kernel uses hooks inside the access control verification code to allow an LSM to enforce its access control policy
  - Example: SELinux
    - Developed by NSA
    - Replaces Linux's default discretionary access control policy with a mandatory access control (MAC) policy

20.13.2 Access Control Methods

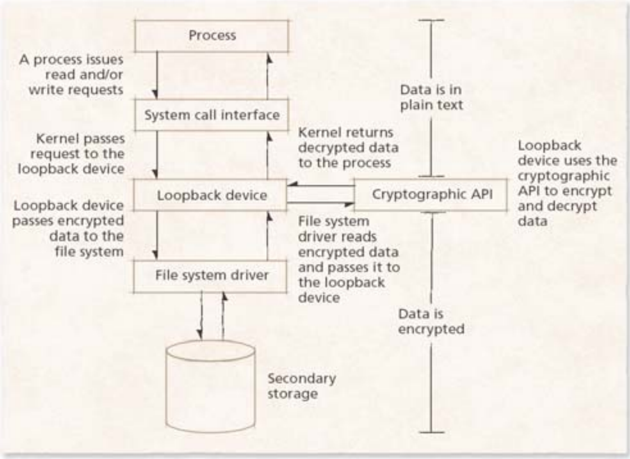
- Privilege inheritance
  - Normally a process executes with same privileges as the user who launched it
  - Some applications require process to execute with other user privileges
    - Example: passwd
  - setuid and setgid allow process to run with the privileges of the file owner
  - Improper use of setuid and setgid can lead to security breaches
  - LSM Capabilities allow administrator to assign privileges to applications as opposed to users to prevent this

20.13.3 Cryptography

- Cryptographic API
  - Enables users to access several forms of encryption to protect their data
  - Uses powerful algorithms such as DES, AES and MD5
  - Kernel uses Cryptographic API to implement IPSec
  - Enables users to create secure (encrypted) file systems
    - Loopback device
      - Layer between the virtual file system and the existing file system
      - Can be used to encrypt and decrypt data transferred between processes and the underlying file system

20.13.3 Cryptography

Figure 20.28 Loopback device providing an encrypted file system using the Cryptographic API.


Process
A process issues read and/or write requests
System call interface
Data is in plain text
Kernel passes request to the loopback device
Kernel returns decrypted data to the process
Loopback device uses the cryptographic API to encrypt and decrypt data
Loopback device
Cryptographic API
Loopback device passes encrypted data to the file system
File system driver reads encrypted data and passes it to the loopback device
File system driver
Data is encrypted
Secondary storage