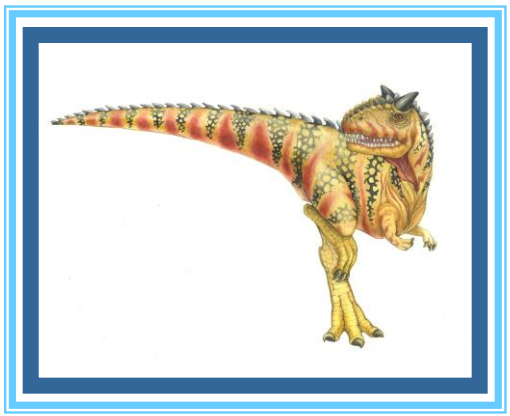
Chapter 20: The Linux System

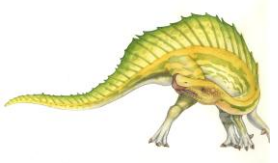


Silberschatz, Galvin and Gagne ©2018



Outline

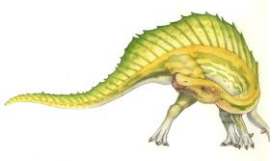
- Linux History
- Design Principles
- Kernel Modules
- Process Management
- Scheduling
- Memory Management
- File Systems
- Input and Output
- Interprocess Communication
- Network Structure
- Security





Objectives

- 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 and provides interprocess communication
- To look at memory management in Linux
- To explore how Linux implements file systems and manages I/O devices

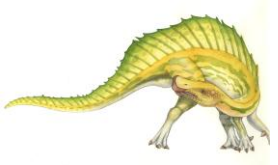


Troy University Resources



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
- It has been 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





The Linux Kernel

- Version 0.01 (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
- Historically, odd minor versions marked **development kernels** and even minor versi**ons marked stable kernels. Modern releases use a different scheme**

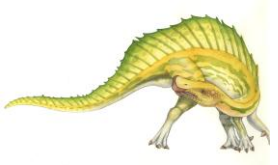


Troy University Resources



Linux 2.0

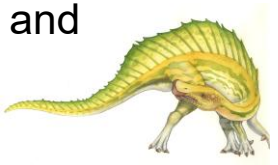
- Released in June 1996, 2.0 added two major new capabilities:
  - Support for multiple architectures, including a fully 64-bit native Alpha port
  - Support for multiprocessor architectures
- Other new features included:
  - Improved memory-management code
  - Improved TCP/IP performance
  - Support 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
- 2.4 and 2.6 increased SMP support, added journaling file system, preemptive kernel, 64-bit memory support
- Linux 3.0 was released in 2011, the twentieth anniversary. CFS had already been introduced in Linux 2.6.23 (2007)





The 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
  - Specifies overall layout of a standard Linux file system, determines under which directory names configuration files, libraries, system binaries, and run-time data files should be stored



Troy University Resources



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, others include **Canonical** and **SuSE**
- RPM provides a package format for RPM-based distributions. Package compatibility also depends on dependencies and the target system





Linux Licensing

- The Linux kernel is licensed under GNU GPL version 2 only, with an explicit system-call exception
  - Not **public domain**, in that not all rights are waived
- Private use does not require publication. Distributing a modified kernel must satisfy GPLv2, including its corresponding-source requirements
  - Can sell distributions, but must offer the source code too

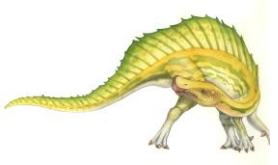


Troy University Resources



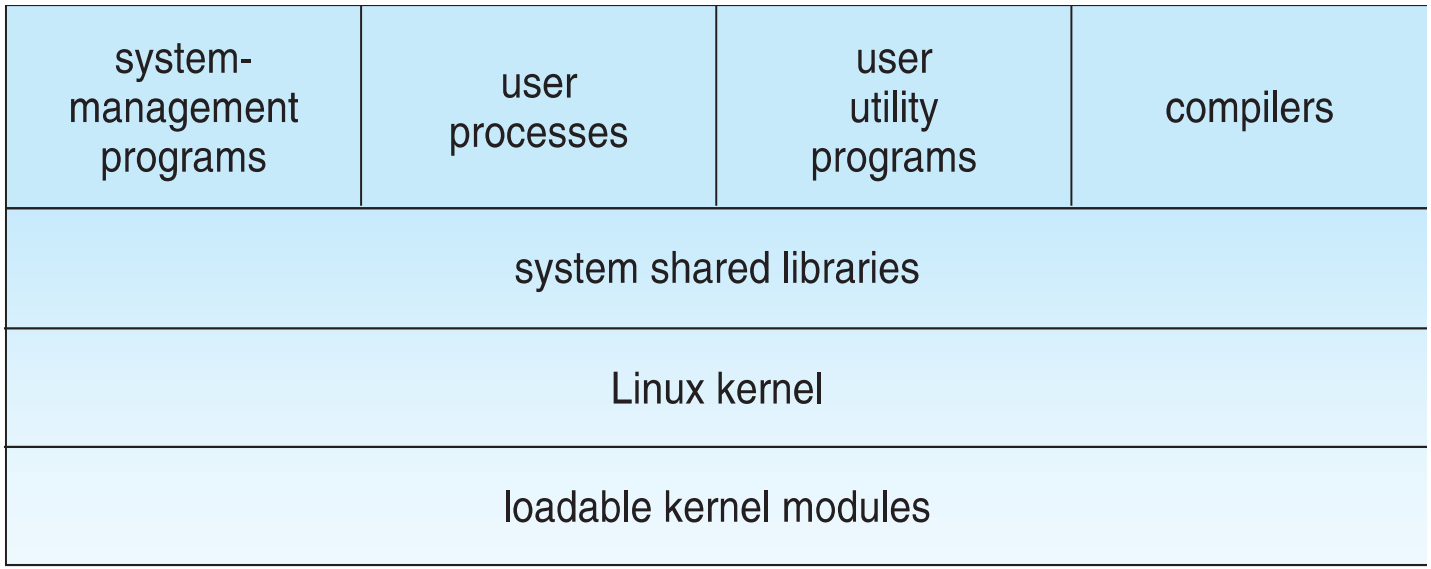
Design Principles

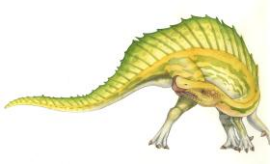
- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with 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





Components of a Linux System



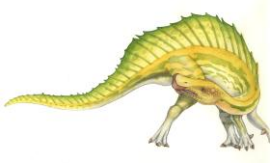


Troy University Resources



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





Components of a Linux System (Cont.)

- 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**)

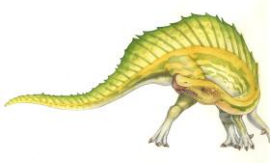


Troy University Resources



Kernel Modules

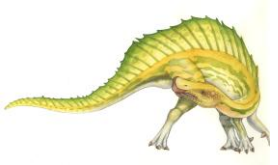
- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol
- Kernel modules must comply with applicable licensing and exported-symbol rules
- Kernel modules 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**





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
- User-space tools load requested modules and their dependencies on demand. Unloading is possible only when references and kernel policy permit it

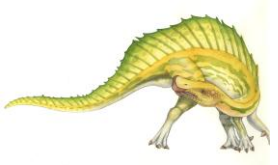


Troy University Resources



Driver Registration

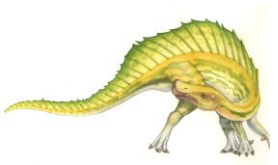
- Allows modules to tell the rest of the kernel that a new driver has become available
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time
- Registration tables include the following items: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





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

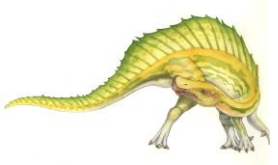


Troy University Resources



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 Identity

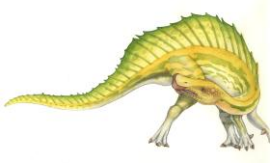
- **Process ID (PID)** - The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process
- **Credentials** - Each process must have an associated user ID and one or more group IDs that determine 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
  - Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX
- **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 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 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**)

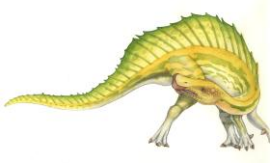


Troy University Resources



Process Context (Cont.)

- 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
  - 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 child task with a separate virtual address space, initially sharing private pages through copy-on-write
  - `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.

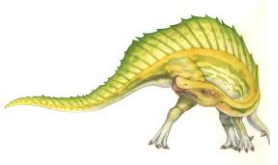


Troy University Resources



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
- As of 2.5, new scheduling algorithm – preemptive, priority-based, known as *O*(1)
  - Real-time range
  - nice value
  - Had challenges with interactive performance
- CFS arrived in 2.6.23. Linux 6.6 began the transition **to EEVDF**





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)

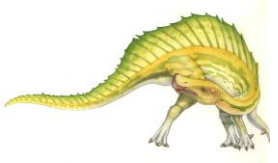


Troy University Resources



CFS (Cont.)

- 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)





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 hardware device can raise an interrupt, causing the CPU to execute the kernel's interrupt handler
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section



Troy University Resources

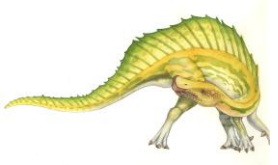


Kernel Synchronization (Cont.)

- Historically, Linux used these techniques for kernel 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. The second technique applies to critical sections that occur in an interrupt service routines

  Disabling local interrupts prevents local interrupt handlers from running. On SMP, locks are also needed to exclude other CPUs
  - 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.





Kernel Synchronization (Cont.)

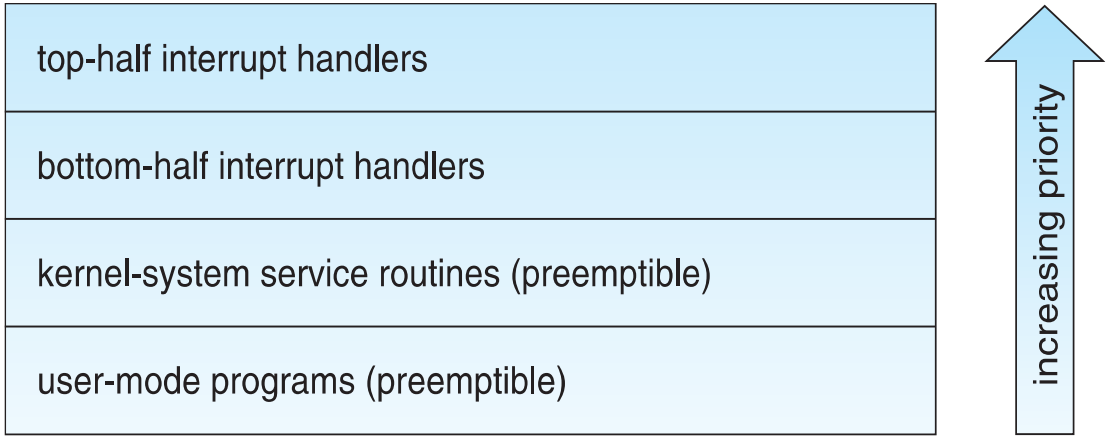
- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration
- Interrupt service routines are separated into a *top half* and a *bottom half*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
  - Deferred work runs through mechanisms such as softirqs or workqueues. Concurrency rules depend on the mechanism
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code



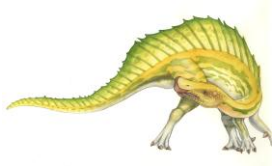
Troy University Resources



Interrupt Protection Levels



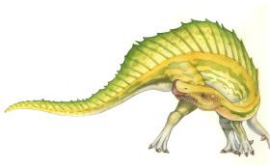
- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level
- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs





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



Troy University Resources



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
- Memory is divided into **zones** according to architecture and configuration
  - Architecture specific, for example on x86:

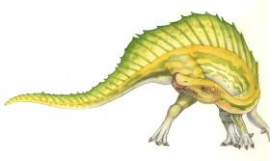
zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB





Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - Whenever two allocated partner regions are both freed up they are combined to form a larger region
  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request

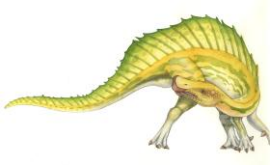


Troy University Resources



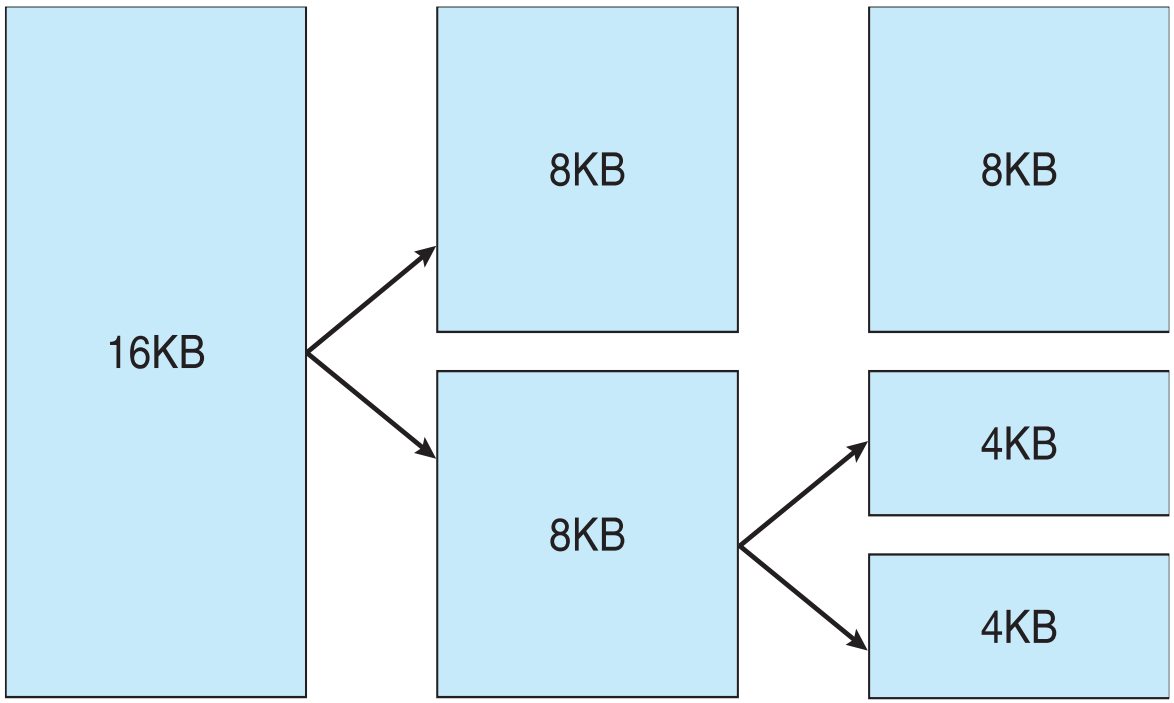
Managing Physical Memory (Cont.)

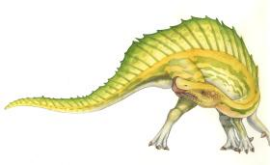
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator)
- Also uses **slab allocator** for kernel memory
- **Page cache** and virtual memory system also manage physical memory
  - Page cache is kernel's main cache for files and main mechanism for I/O to block devices
  - Page cache stores entire pages of file contents for local and network file I/O





Splitting of Memory in a Buddy Heap

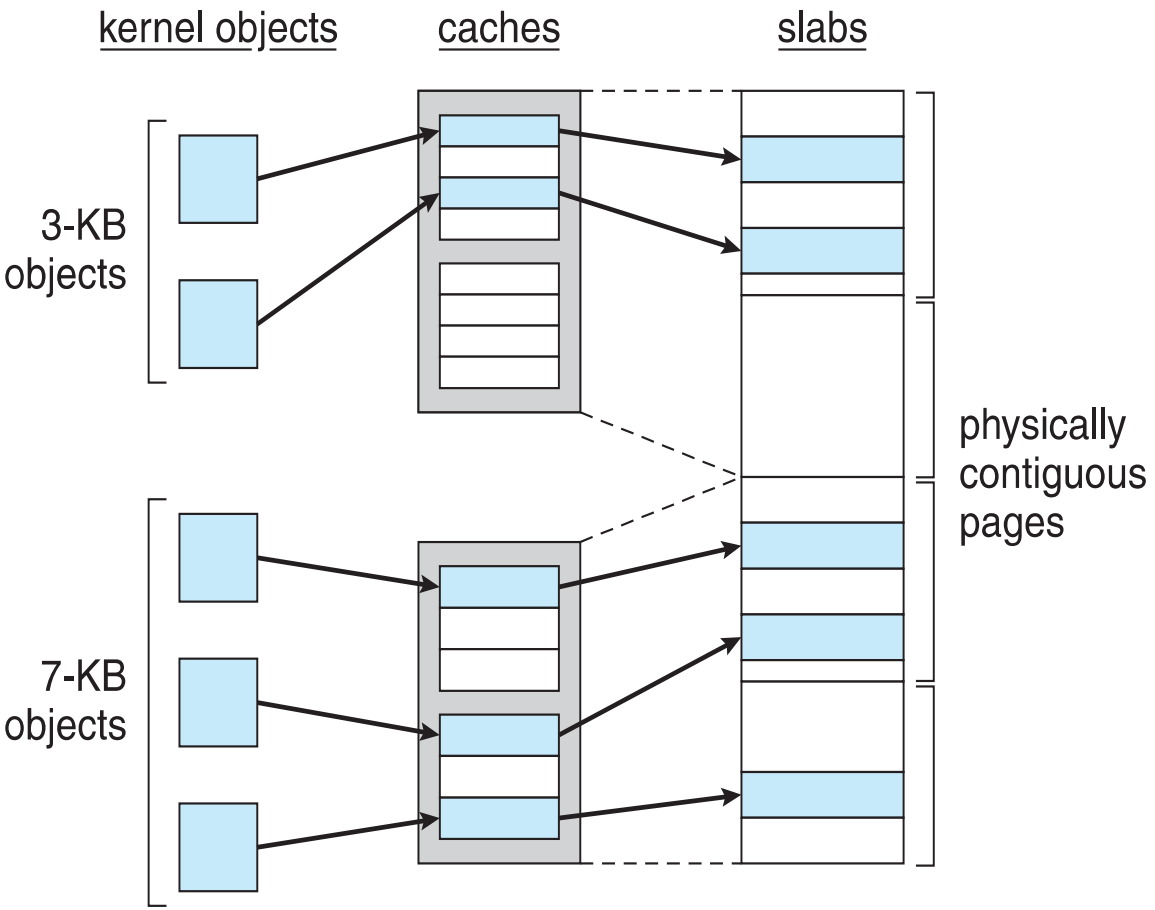


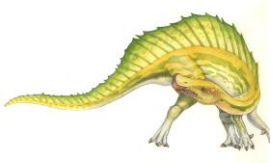


Troy University Resources



Slab Allocator in Linux

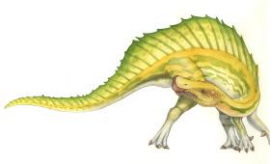






Virtual Memory

- 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 non-overlapping 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

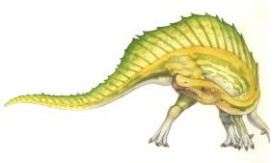


Troy University Resources



Virtual Memory (Cont.)

- 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 (Cont.)

- 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 fork, private mappings initially share physical pages. A write triggers copy-on-write so parent and child retain separate private memory

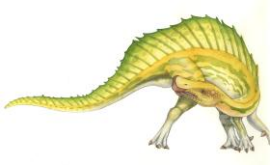


Troy University Resources



Swapping and Paging

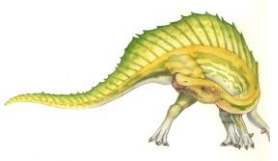
- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else
- The VM paging system can be divided into two sections:
  - The **pageout-policy** algorithm decides which pages to write out to disk, and when
  - The **paging mechanism** actually carries out the transfer, and pages data back into physical memory as needed
  - Can page out to either swap device or normal files
  - Bitmap used to track used blocks in swap space kept in physical memory
  - Allocator uses next-fit algorithm to try to write contiguous runs





Traditional Kernel Mapping

- Traditionally, part of each process's virtual address space maps kernel memory. With page-table isolation, user-mode page tables omit most kernel mappings
- This kernel virtual-memory area contains two regions:
  - A direct-map region maps eligible physical RAM into kernel virtual addresses. Its coverage depends on the architecture
  - The reminder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory

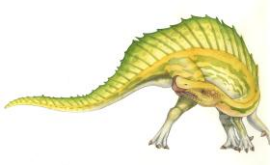


Troy University Resources



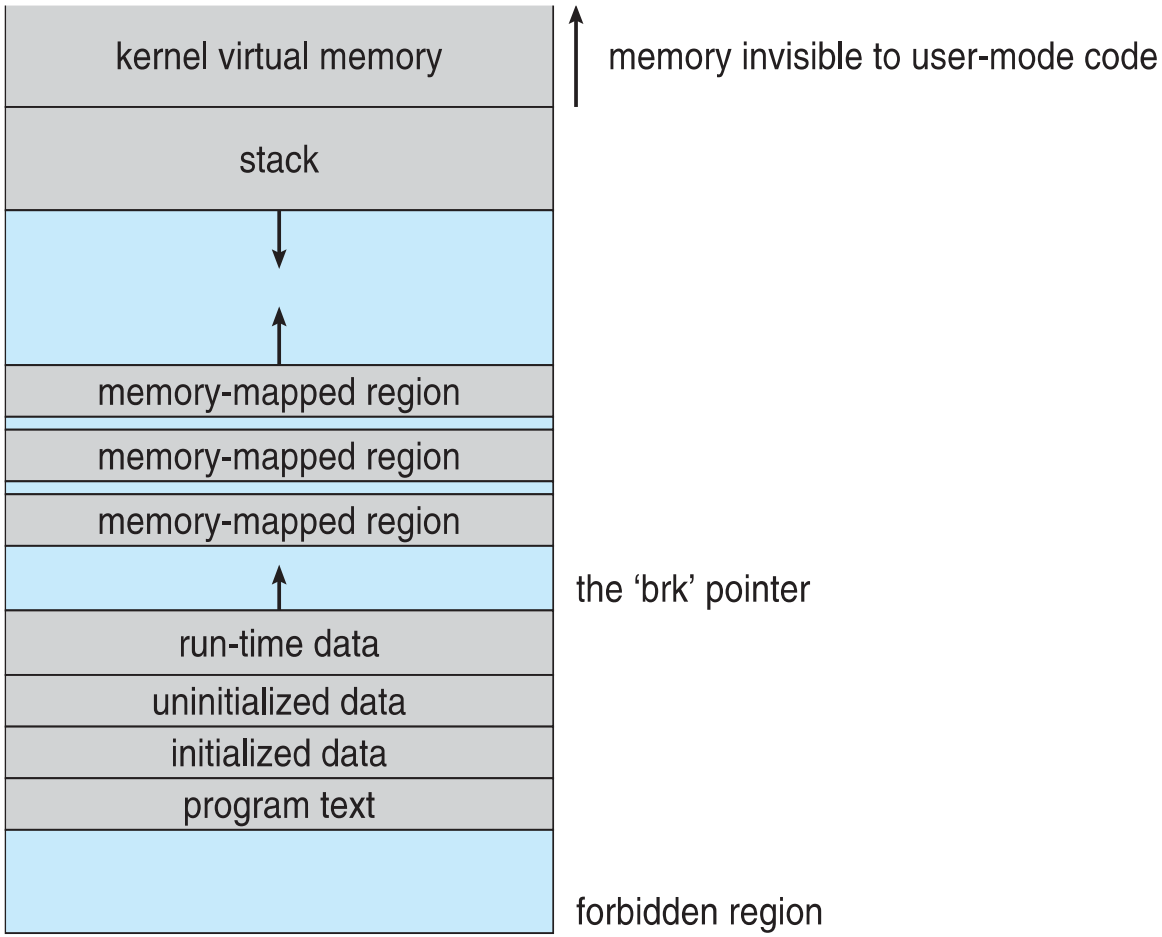
Executing and Loading User Programs

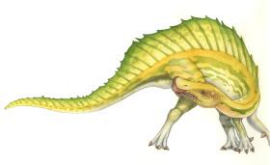
- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made
- The registration of multiple loader routines allows Linux to support both the **ELF** and **a.out** binary formats
- Initially, binary-file pages are mapped into virtual memory
  - Only when a program tries to access a given page will a page fault result in that page being loaded into physical memory
- An ELF executable has headers, loadable segments, and other structures
  - The ELF loader uses program headers to map loadable segments into virtual memory





Memory Layout for ELF Programs



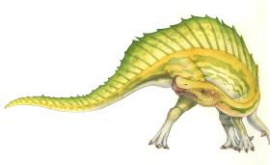


Troy University Resources



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





Static and Dynamic Linking (Cont.)

- Linux implements dynamic linking in user mode through special linker library
  - For a dynamically linked ELF program, the kernel loads the interpreter named by PT_INTERP
  - Execution begins in the dynamic linker, which prepares the program
  - Link library determines dynamic libraries required by process and names of variables and functions needed
  - The dynamic linker maps required libraries at suitable virtual addresses and resolves symbol references
  - Shared libraries compiled to be **position-independent code** (**PIC**) so can be loaded anywhere

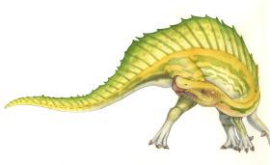


Troy University Resources



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 virtual file system (VFS)
- The Linux VFS is designed around object-oriented principles and is composed of four components:
  - A set of definitions that define what a file object is allowed to look like
    - The **inode object** structure represent an individual file
    - The **file object** represents an open file
    - The **superblock object** represents an entire file system
    - A **dentry object** represents an individual directory entry





File Systems (Cont.)

- 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 virtual file system (VFS)
- The Linux VFS is designed around object-oriented principles and layer of software to manipulate those objects with a set of operations on the objects
  - For example, file operations are defined by struct file_operations in the kernel source include/linux/fs.h:

    int open(. . .) — Open a file

    ssize_t read(. . .) — Read from a file

    ssize_t write(. . .) — Write to a file

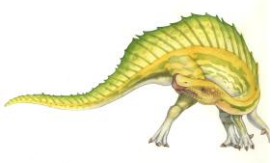
    int mmap(. . .) — Memory-map a file





The Linux ext3 File System

- **ext3** was a widely used Linux on-disk file system
  - Uses a mechanism similar to that of BSD Fast File System (FFS) for locating data blocks belonging to a specific file
  - Supersedes older **extfs**, **ext2** file systems
  - ext4 is its established successor, with features such as extents
  - Of course, many other file system choices with Linux distros





The Linux ext3 File System (Cont.)

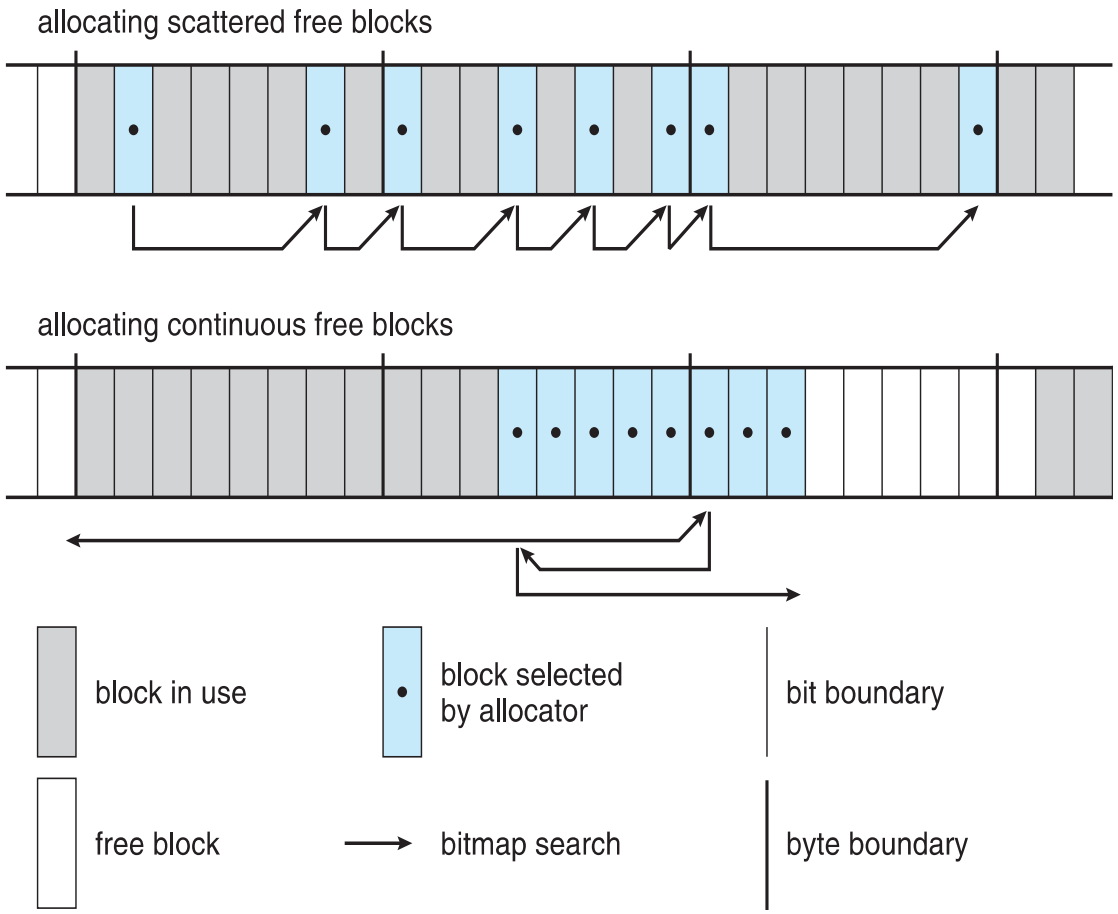
- The main differences between ext2fs and FFS concern their disk allocation policies
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
  - ext3 does not use fragments; it performs its allocations in smaller units
    - ext3 commonly uses 1, 2, or 4 KiB blocks, subject to architecture and format limits
  - ext3 uses cluster allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation on a **block group**
  - Maintains bit map of free blocks in a block group, searches for free byte to allocate at least 8 blocks at a time

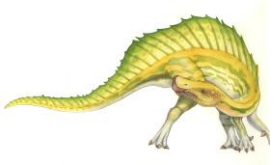


Troy University Resources



Ext2fs Block-Allocation Policies

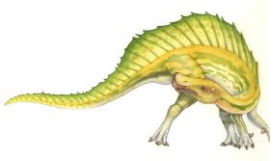






Journaling

- ext3 implements **journaling**, with file system updates first written to a log file in the form of **transactions**
  - A transaction is committed when its commit record is durably written
  - Over time, log file transactions replayed over file system to put changes in place
- On system crash, some transactions might be in journal but not yet placed into file system
  - Recovery replays committed transactions and ignores incomplete ones
  - Journal replay usually avoids a full scan after a crash. Other corruption can still require consistency checks
- Improves write performance on hard disks by turning random I/O into sequential I/O

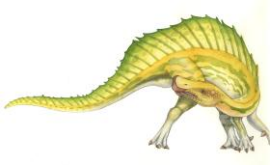


Troy University Resources



The Linux Proc File System

- The **proc file system** does not store data, rather, its contents are computed on demand according to user file I/O requests
- **proc** must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains
  - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode
  - When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer





Input and Output

- The 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

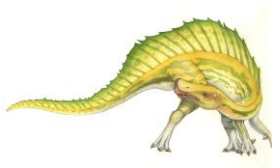


Troy University Resources



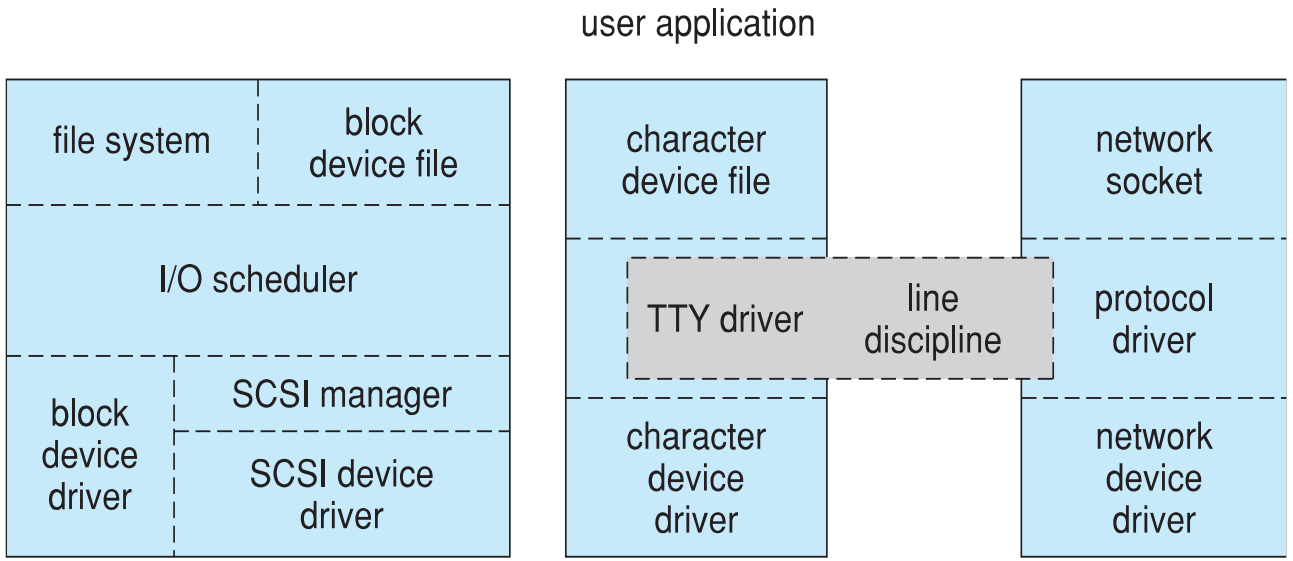
Block Devices

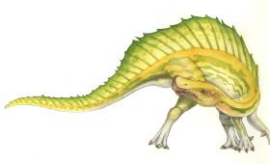
- Provide the main interface to all disk devices in a system
- The block buffer cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O
- The **request manager** manages the reading and writing of buffer contents to and from a block device driver
- Kernel 2.6 introduced **Completely Fair Queueing** (**CFQ**)
  - CFQ is historical. Modern block I/O schedulers depend on the kernel and device configuration
  - Fundamentally different from elevator algorithms
  - Maintains set of lists, one for each process by default
  - Uses C-SCAN algorithm, with round robin between all outstanding I/O from all processes
  - Four blocks from each process put on at once





Device-Driver Block Structure



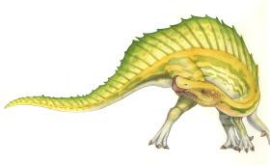


Troy University Resources



Character Devices

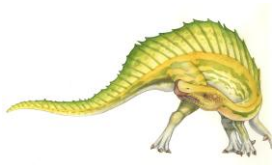
- A device driver which does not offer random access to fixed blocks of data
- A character device driver must register a set of functions which implement the driver's various file I/O operations
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply 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





Character Devices (Cont.)

- **Line discipline** is an interpreter for the information from the terminal device
  - The most common line discipline is tty discipline, which glues the terminal's data stream onto standard input and output streams of user's running processes, allowing processes to communicate directly with the user's terminal
  - Several processes may be running simultaneously, tty line discipline responsible for attaching and detaching terminal's input and output from various processes connected to it as processes are suspended or awakened by user
  - Other line disciplines also are implemented have nothing to do with I/O to user process – i.e., PPP and SLIP networking protocols

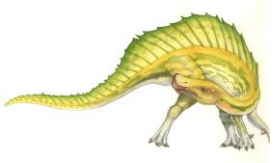


Troy University Resources



Interprocess Communication

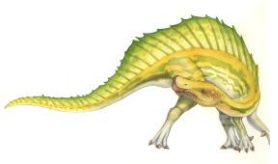
- Like UNIX, Linux informs processes that an event has occurred via **signals**
- Signals can include sender and status information. Real-time signals can queue and carry a value
- Signals may become pending during kernel execution and are normally delivered on return to user mode. Kernel waits also use task states and **`wait queue`**s
- Also implements System V Unix semaphores
  - Process can wait for a signal or a semaphore
  - Semaphores scale better
  - Operations on multiple semaphores can be atomic





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 a 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
- Shared-memory communication needs synchronization, for example process-shared mutexes, semaphores, or appropriate atomics

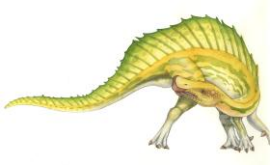


Troy University Resources



Network Structure

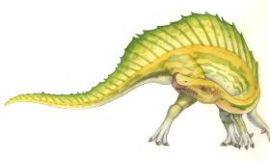
- Networking is a key area of functionality for Linux
  - It supports the standard Internet protocols for UNIX to UNIX communications
  - Older Linux releases also supported legacy protocols such as AppleTalk and IPX
- Internally, networking in the Linux kernel is implemented by three layers of software:
  - The socket interface
  - Protocol drivers
  - Network device drivers
- Most important set of protocols in the Linux networking system is the internet protocol suite
  - It implements routing between different hosts anywhere on the network
  - On top of the routing protocol are built the UDP, TCP and ICMP protocols
- Packets also pass to **firewall management** for filtering based on **firewall chains** of rules





Security

- The **pluggable authentication modules** *(***PAM***)* system is available under Linux
- PAM is based on a shared library that can be used by any system component that needs to authenticate users
- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**)
- Access control is performed by assigning objects a *protections mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access

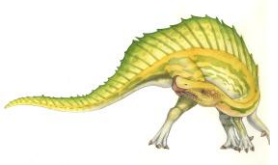


Troy University Resources

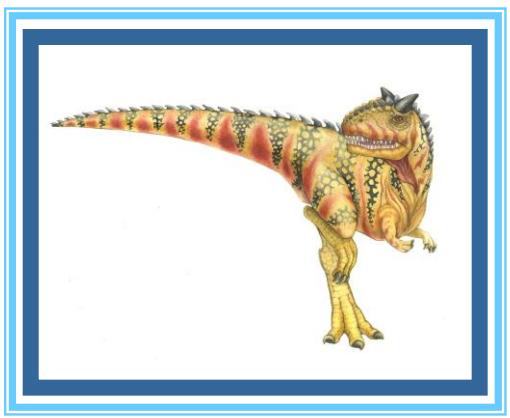


Security (Cont.)

- Linux augments the standard UNIX **setuid** mechanism in two ways:
  - It implements the POSIX specification's saved ***user-id*** mechanism, which allows a process to repeatedly drop and reacquire its effective uid
  - It has added a process characteristic that grants just a subset of the rights of the effective uid
- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process without granting it any other privileges



End of Chapter 20



Silberschatz, Galvin and Gagne ©2018