

# Introduction to the Linux Operating System

*Balbir Singh ([balbir@in.ibm.com](mailto:balbir@in.ibm.com))*

*Adapted from the slides by Schilberschatz, Galvin and Gagne*

*Copyright 2005*

Adapted from Silberschatz, Galvin and  
Gagne ©2005

# The Linux Operating System

- Process Management
- Scheduling
- Memory Management
- File Systems
- Input and Output
- Interprocess Communication

# Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - ▶ Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device.
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

Adapted from Silberschatz, Galvin and Gagne ©2005

# Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont):
  - Communications – Processes may exchange information, on the same computer or between computers over a network
    - ▶ Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to be constantly aware of possible errors
    - ▶ May occur in the CPU and memory hardware, in I/O devices, in user program
    - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing

# Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - ▶ Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **Accounting** - To keep track of which users use how much and what kinds of computer resources

# Operating System Services (Cont.)

- ▶ **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
  - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

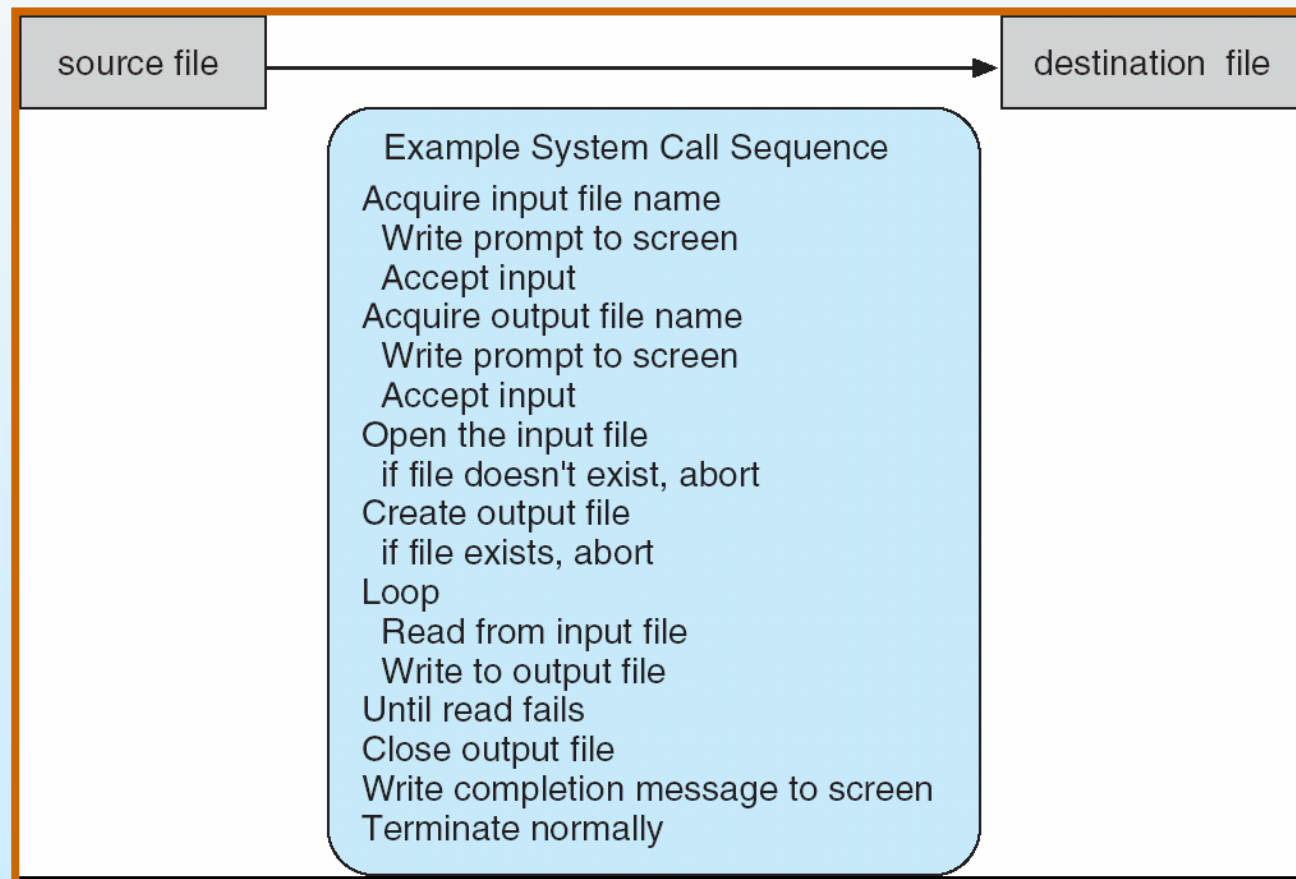
Adapted from Silberschatz, Galvin and  
Gagne ©2005

# System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use

# Example of System Calls

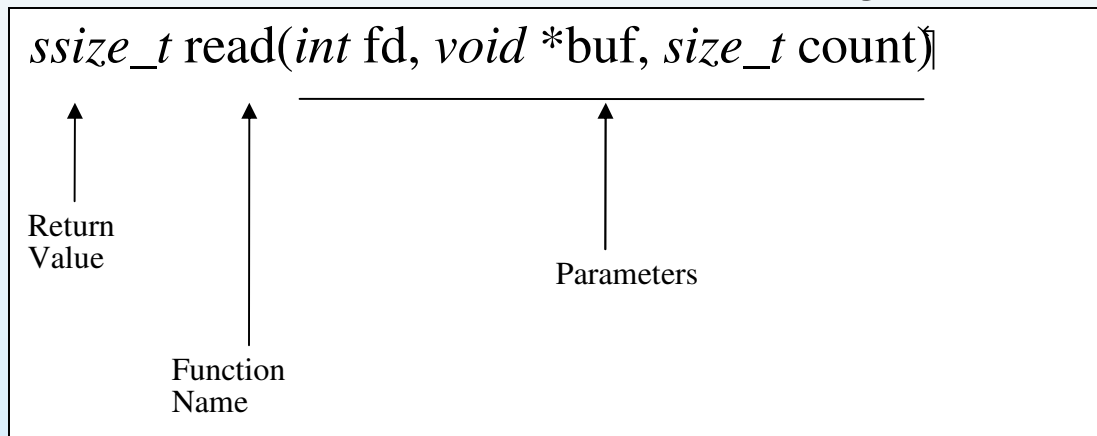
- System call sequence to copy the contents of one file to another file



Adapted from Silberschatz, Galvin and  
Gagne ©2005

# Example of Standard API

- Consider the read() function in the
- POSIX API—a function for reading from a file

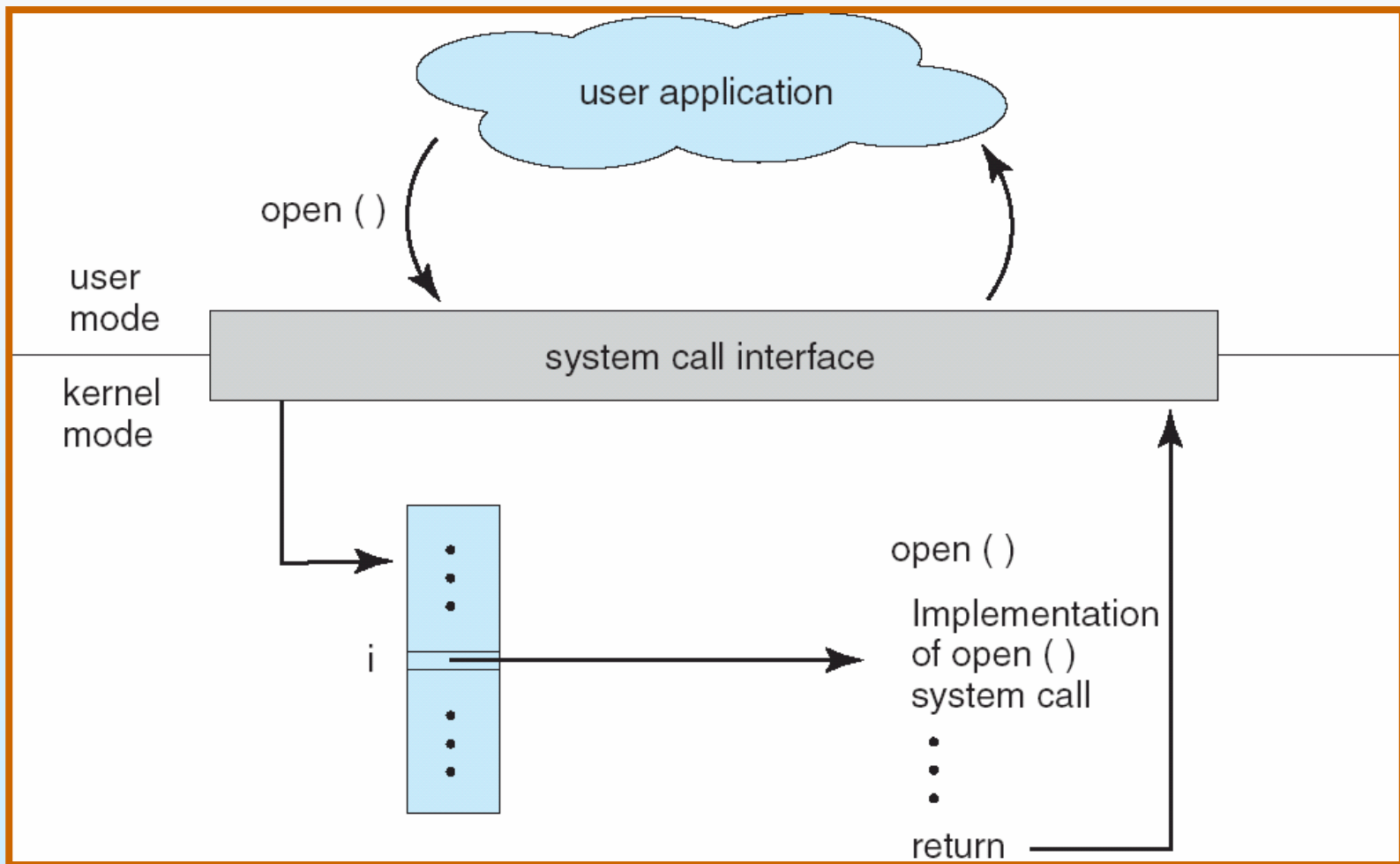


A description of the parameters passed to read(2)

- `int fd`—Handle of the file to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the number of bytes to be read into the buffer

Adapted from Silberschatz, Galvin and Gagne ©2005

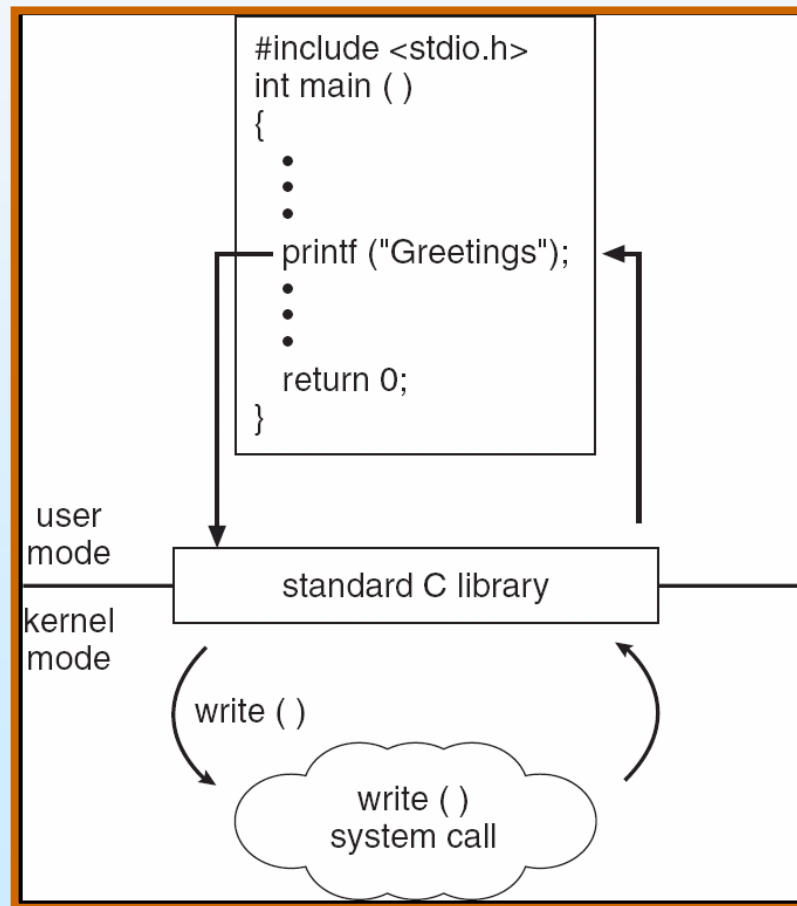
# API – System Call – OS Relationship



Adapted from Silberschatz, Galvin and Gagne ©2005

# Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Adapted from Silberschatz, Galvin and Gagne ©2005

# Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Adapted from Silberschatz, Galvin and Gagne ©2005

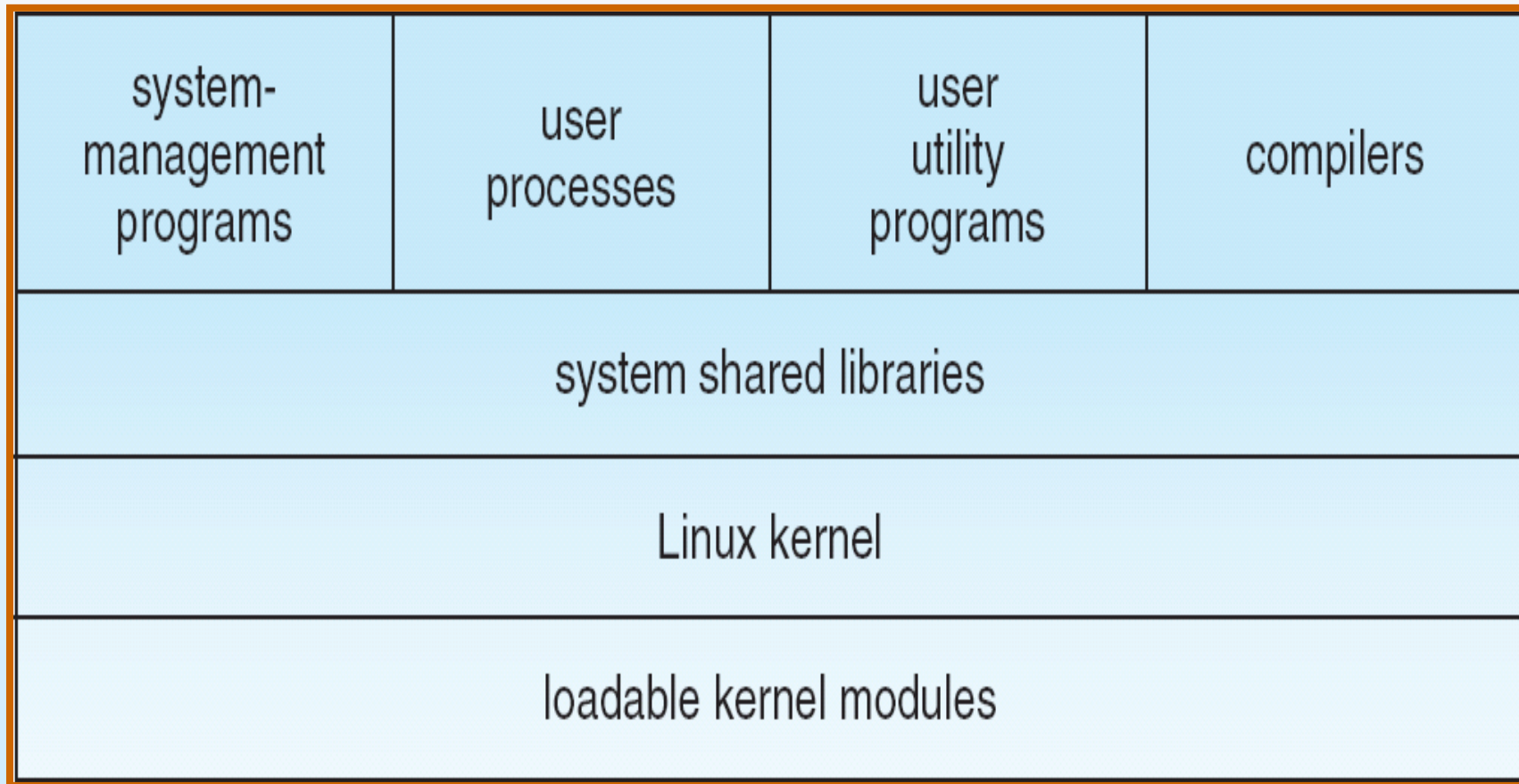
# Operating System Design and Implementation (Cont.)

- Important principle to separate
  - Policy:** What will be done?
  - Mechanism:** How to do it?
- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

# 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

# Components of a Linux System



Adapted from Silberschatz, Galvin and  
Gagne ©2005

# Components of a Linux System (Cont.)

- Like most UNIX implementations, Linux is composed of user level and kernel level code
- 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

# Process Management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The **fork** system call creates a new process
  - A new program is run after a call to **execve**
- Under UNIX, a process encompasses all the information that the operating system must maintain 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

Adapted from Silberschatz, Galvin and Gagne ©2005

# 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

# Process Environment

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

# Process 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

## 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
- A distinction is only made when a new thread is created by the **clone** system call
  - **fork** creates a new process with its own entirely new process context
  - **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent

Adapted from Silberschatz, Galvin and Gagne ©2005

# Scheduling

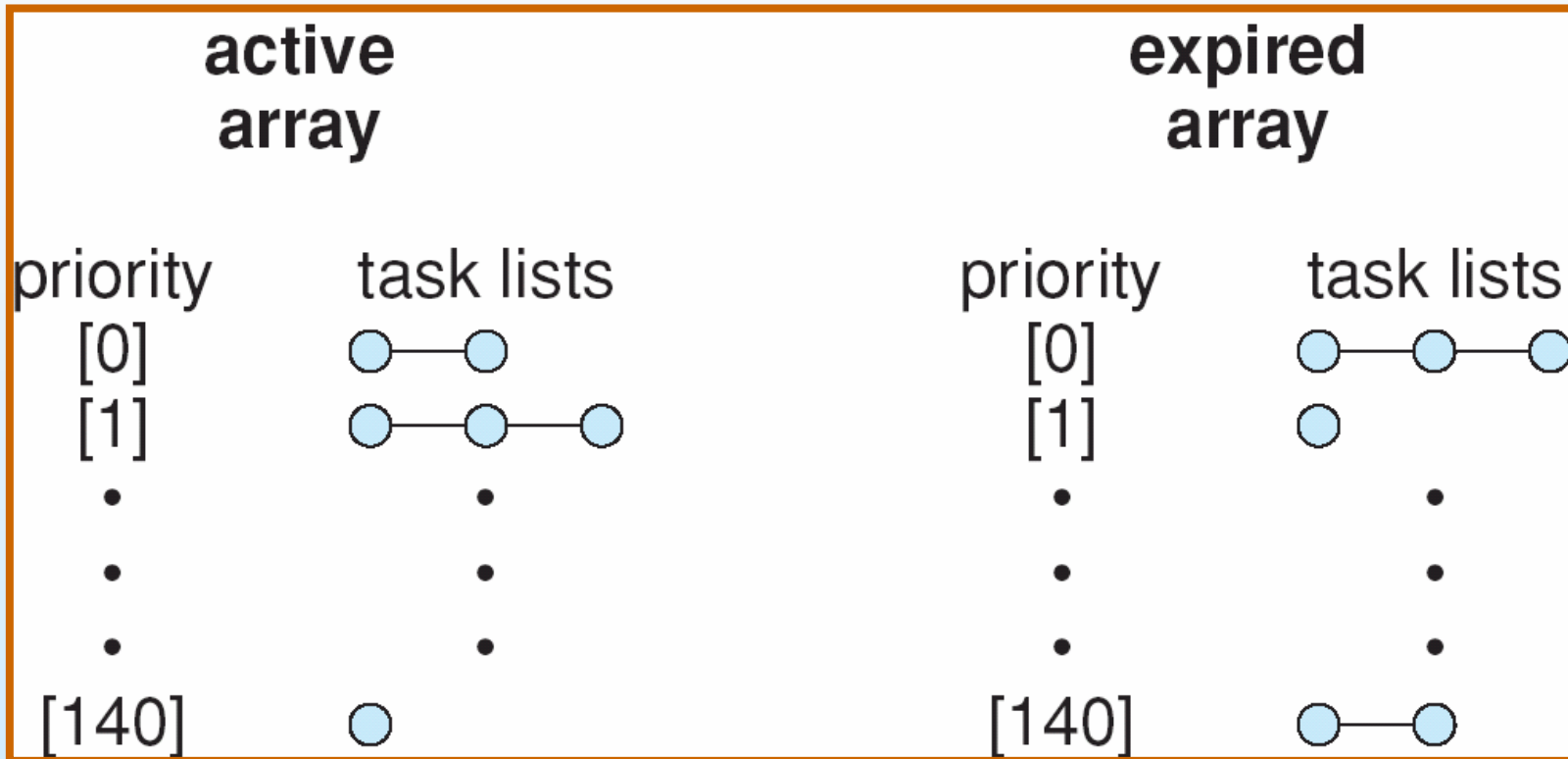
- The job of allocating CPU time to different tasks within an operating system
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested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

# Relationship Between Priorities and Time-slice Length

| <u>numeric priority</u> | <u>relative priority</u> |                 | <u>time quantum</u> |
|-------------------------|--------------------------|-----------------|---------------------|
| 0                       | highest                  | real-time tasks | 200 ms              |
| •                       |                          |                 |                     |
| •                       |                          |                 |                     |
| •                       |                          |                 |                     |
| 99                      |                          |                 |                     |
| 100                     |                          | other tasks     |                     |
| •                       |                          |                 |                     |
| •                       |                          |                 |                     |
| •                       |                          |                 |                     |
| 140                     | lowest                   |                 | 10 ms               |

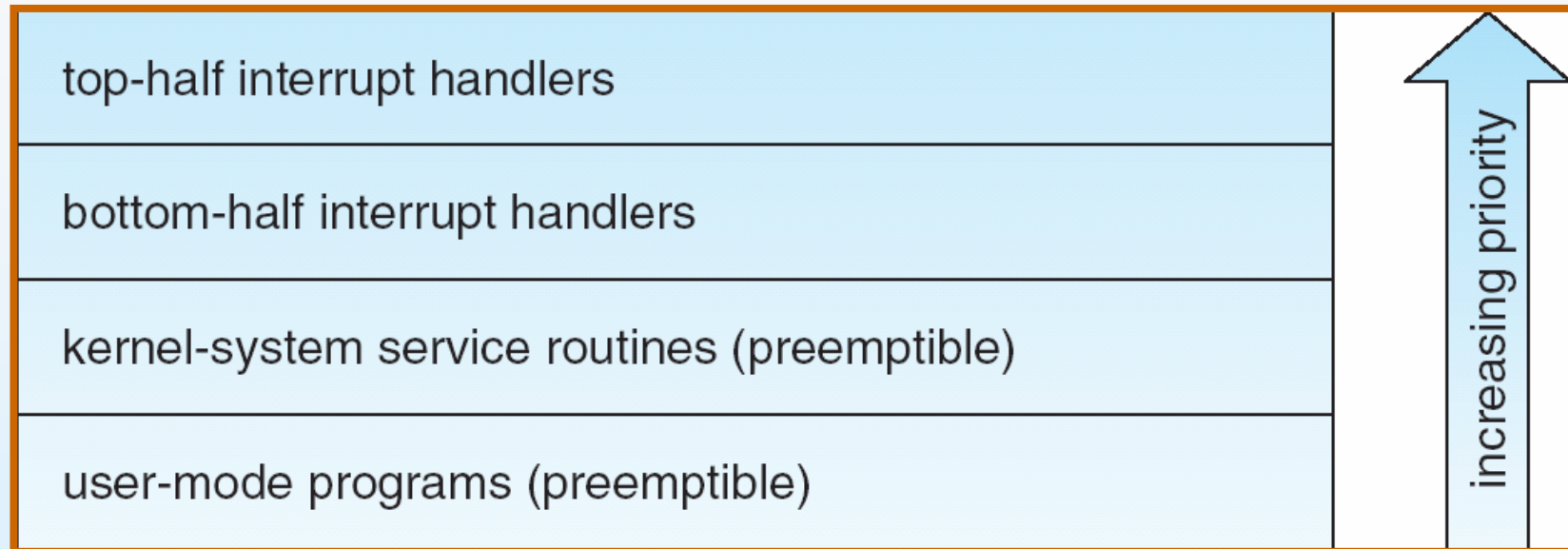
Adapted from Silberschatz, Galvin and Gagne ©2005

# List of Tasks Indexed by Priority



Adapted from Silberschatz, Galvin and Gagne ©2005

# 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

Adapted from Silberschatz, Galvin and Gagne ©2005

# Process Scheduling

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

Adapted from Silberschatz, Galvin and Gagne ©2005

# Memory Management

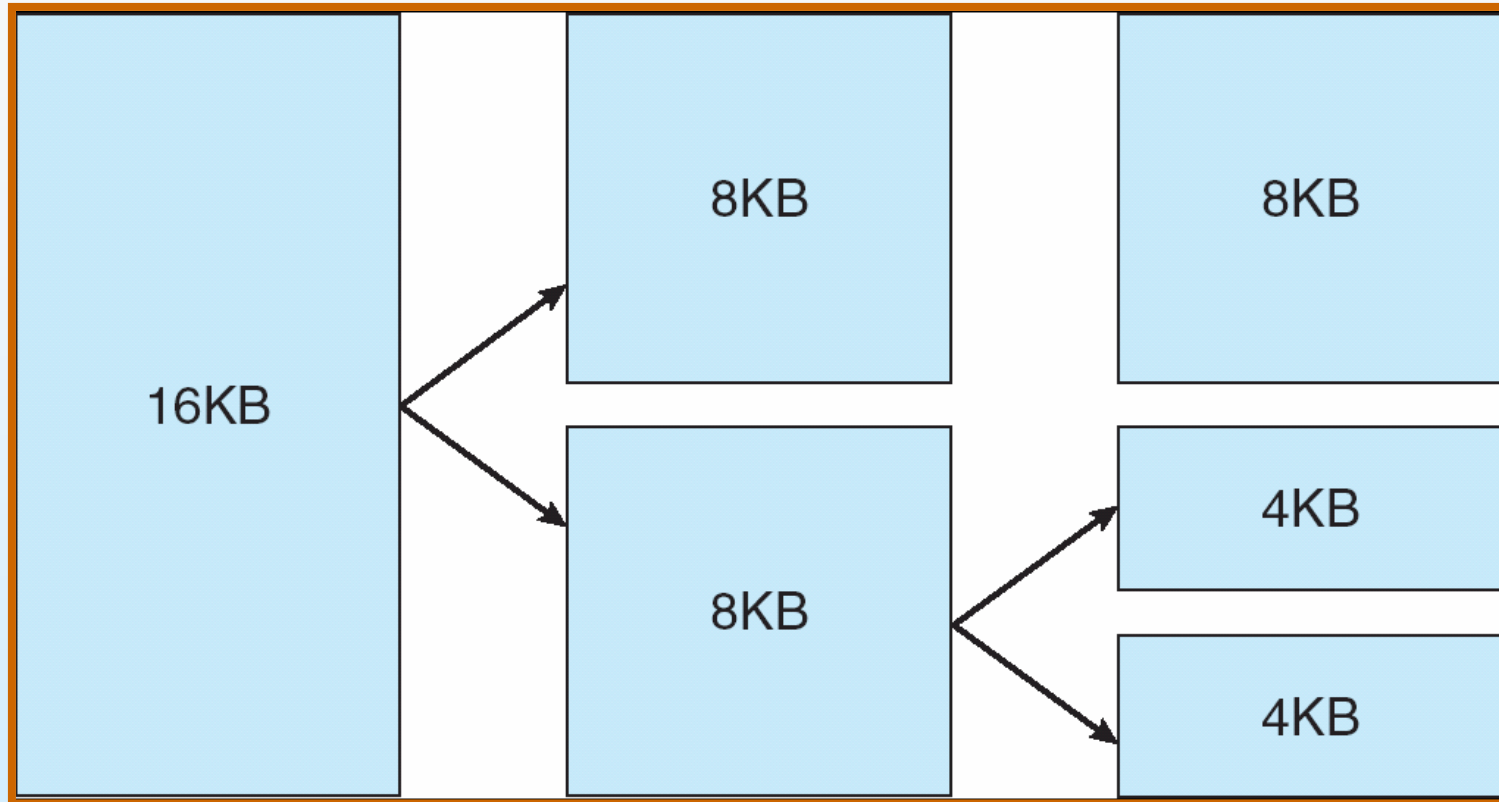
- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into 3 different **zones** due to hardware characteristics

## Relationship of Zones and Physical Addresses on 80x86

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

Adapted from Silberschatz, Galvin and Gagne ©2005

# Splitting of Memory in a Buddy Heap



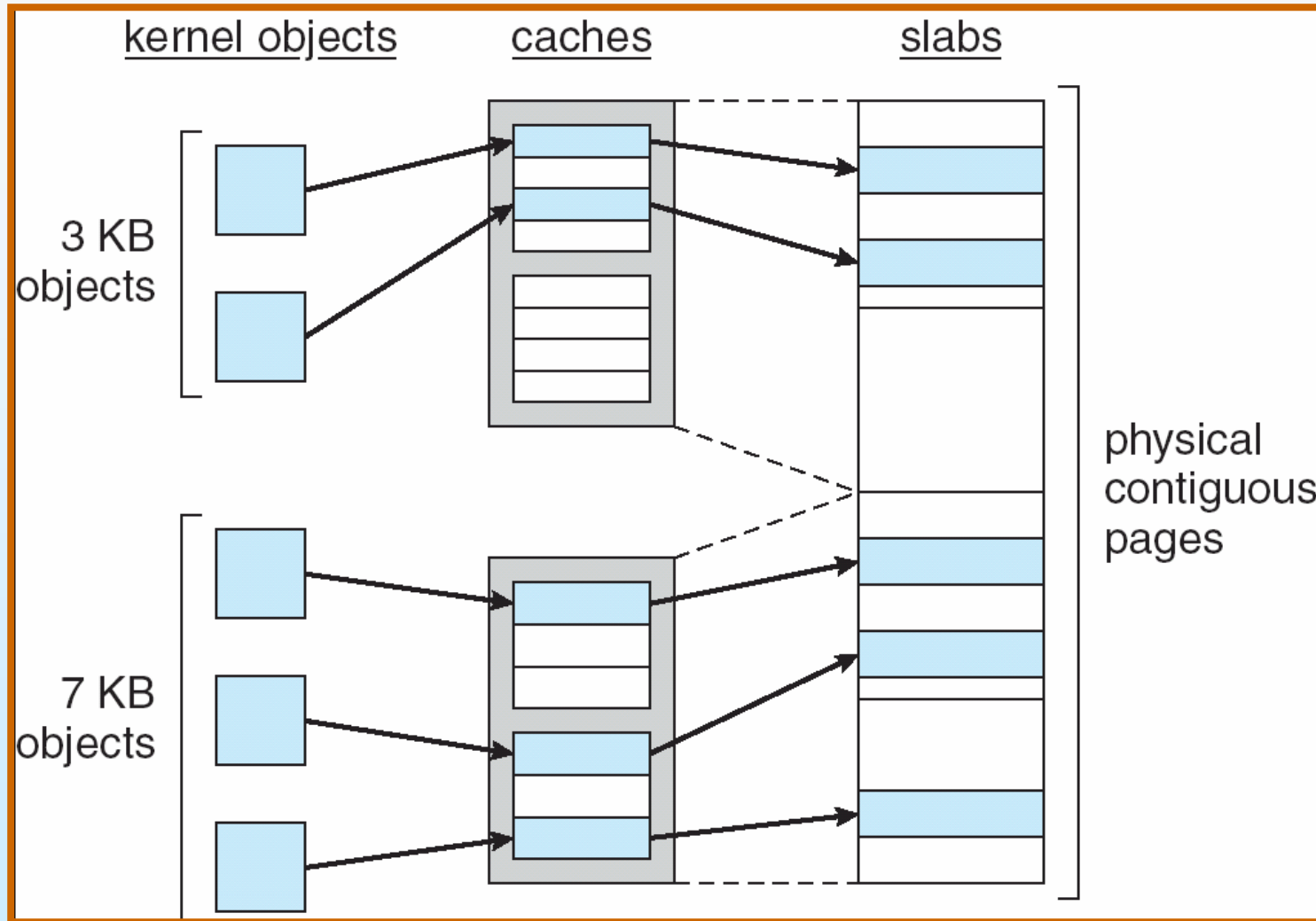
Adapted from Silberschatz, Galvin and Gagne ©2005

# 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

Adapted from Silberschatz, Galvin and Gagne ©2005

# Slab Allocation



Adapted from Silberschatz, Galvin and Gagne ©2005

# 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
  - A physical view of each address space which is stored in the hardware page tables for the process

Adapted from Silberschatz, Galvin and Gagne ©2005

# 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

Adapted from Silberschatz, Galvin and Gagne ©2005

## 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

# Virtual Memory (Cont.)

- 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

# 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

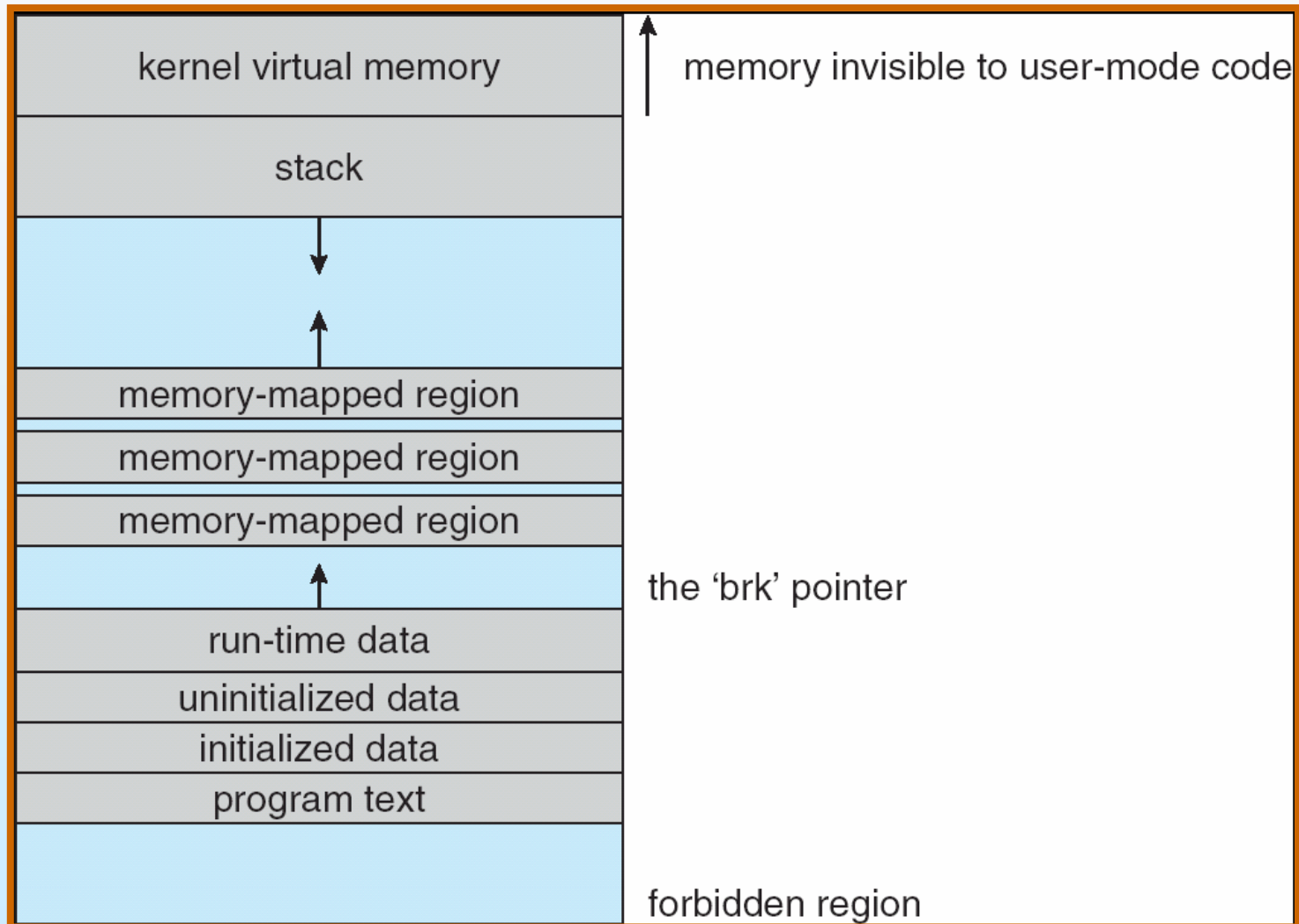
Adapted from Silberschatz, Galvin and Gagne ©2005

# Executing and Loading of Programs

(cont'd)

- An ELF-format binary file consists of a header followed by several page-aligned sections
  - The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory

# Memory Layout for ELF Programs



Adapted from Silberschatz, Galvin and Gagne ©2005

# 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

Adapted from Silberschatz, Galvin and  
Gagne ©2005

# 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 two components:
  - A set of definitions that define what a file object is allowed to look like
  - A layer of software to manipulate those objects

# 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

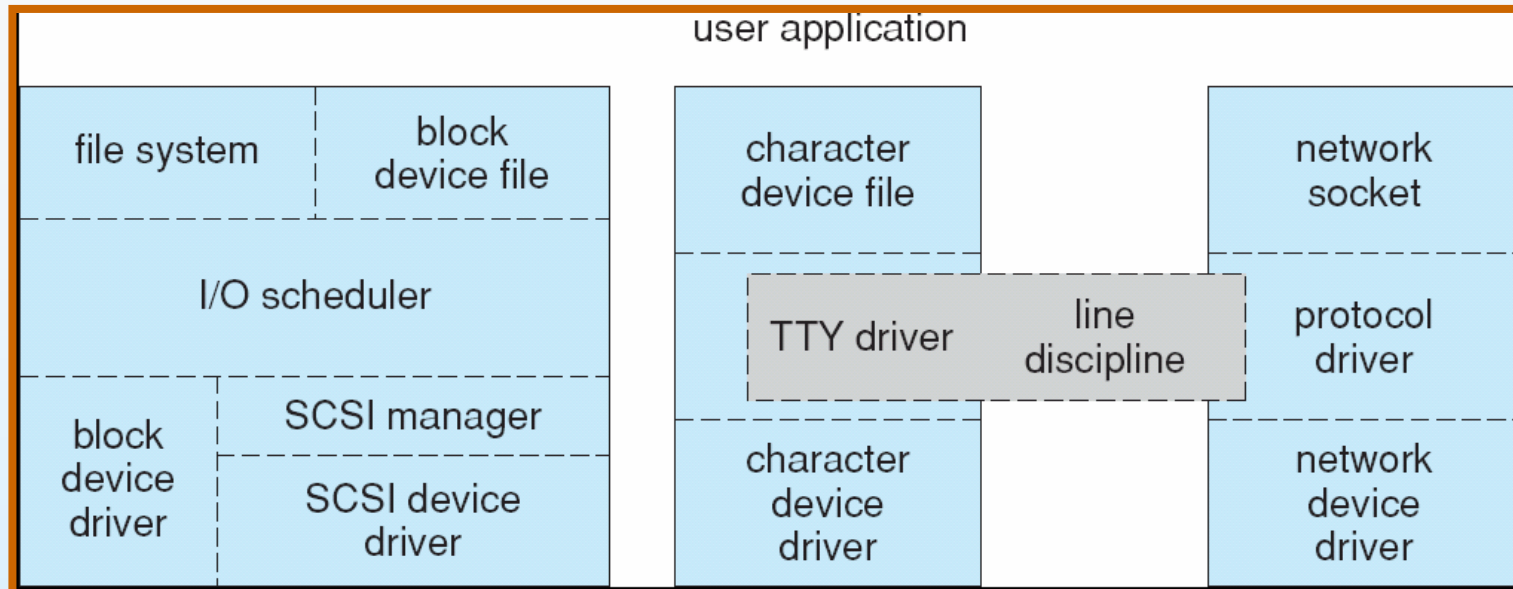
# Input and Output

- 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

# Input and Output (cont'd)

- 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

# Device-Driver Block Structure



Adapted from Silberschatz, Galvin and Gagne ©2005

# 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

# 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

# Interprocess Communication

- Like UNIX, Linux informs processes that an event has occurred via signals
- There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process
- The Linux kernel does not use signals to communicate with processes which are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and **wait queue** structures

# Passing Data Between Processes

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

# Shared Memory Object

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

# Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM.
- IBM is a registered trademark of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.
- Other company, product, and service names may be trademarks or service marks of others.

# 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

# 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

# Kernel Modules

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

Adapted from Silberschatz, Galvin and Gagne ©2005

# Module Management

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