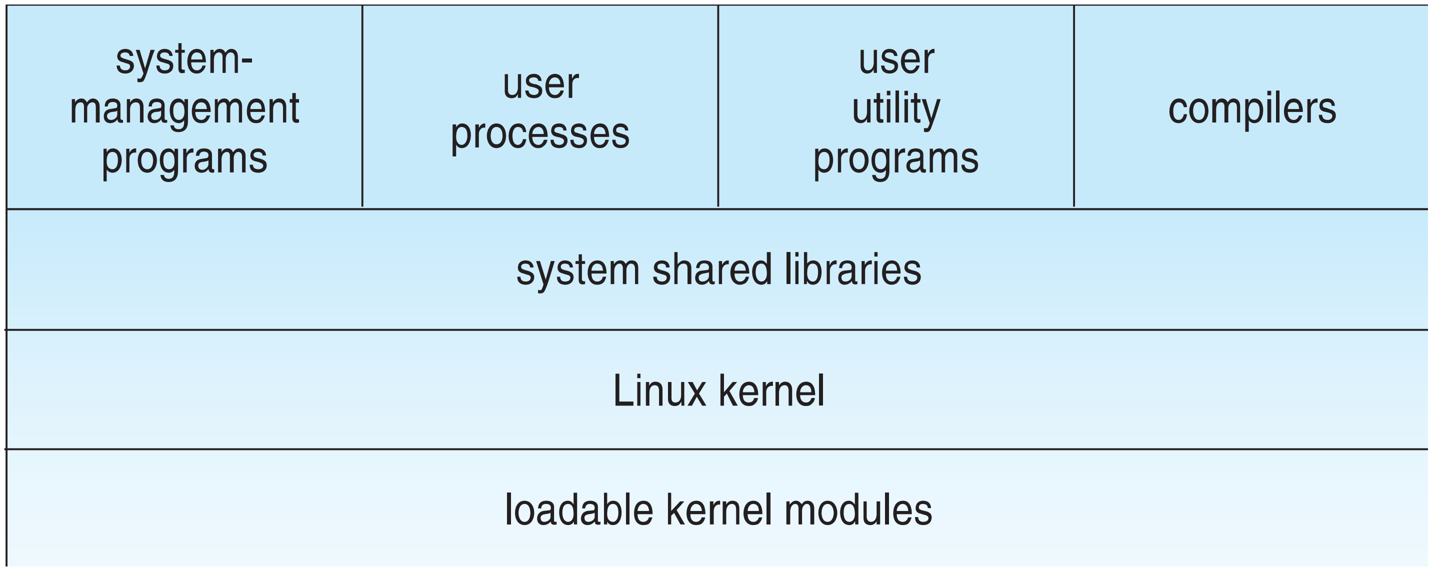
Chapter 18:
The Linux System

Ms.S.Lakshmi Priya
AP/CSE

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



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
- Kernel mode
  - Privileged mode – all kernel code runs in this mode
- User mode
  - User applications and
  - all OS support code that does not need the intervention of kernel are kept in system libraries runs in this mode

Components of a Linux System (Cont.)

- Linux retains UNIX's historical model: the kernel is created as a single, monolithic binary
  - all kernel code and data structures are kept in a single address space, no unnecessary context switches
  - Kernel subsystems communicate using simple function calls rather than IPC (message passing)
- But still room for modularity – some components are kept as loadable modules to load/unload dynamically at runtime

Components of a Linux System (Cont.)

- The system libraries define
  - a standard set of special functions through which applications interact with the kernel, and (eg: System calls)
  - which implement much of the operating-system functionality that does not need the full privileges of kernel code
- The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.
- The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines.

Components of a Linux System (Cont.)

- The Linux system includes a wide variety of user-mode programs—bothsystem utilities and user utilities.
- The **system utilities** perform individual specialized management tasks
  - Some system utilities are invoked just once to initialize and configure some aspect of the system.
  - Others—known as **daemons** in UNIX terminology—run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files
- User utilities are also necessary to the basic operation of the system but do not require elevated privileges to run. They include simple file-management utilities such as those to copy files, create directories, and edit text files

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
  - i.e., it sets the execution domain – how each system call can behave
- 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
  - Processes and their children can have different namespaces.

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.
  - Child usually inherits the environment of the parent
  - Alternatively, variants of exec can be used to set new environment – execle, execve
- 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
  - Saved copies of all process' registers
  - Info about scheduling priority, any outstanding signals waiting to be delivered to 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
  - i.e., amount of time spent in both user mode and kernel mode
- The **file table** is an array of pointers to kernel file structures
  - When making file I/O system calls, processes refer to files by their index into this table, the **file descriptor** (**fd**)

Process Context (Cont.)

- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files
  - Process' root directory, pwd and namespace are stored in this
  - default directories to be used for new file searches are also stored here
- The **signal-handler table** defines the action to take in response to a specific signal
  - Valid actions are ignoring the signal, **terminating the process (default)** and invoking a routine in process' address space.
- The **virtual-memory context** of a process describes the full contents of the its private address space
  - Text, data and stack section

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() duplicates a process without loading a new executable image
  - clone() behaves similar to fork except that it accepts as arguments a set of flags and dictate what resources are shared between the parent and the child
- 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.

Scheduling

- The job of allocating CPU time to different tasks within an operating system
- Linux supports preemptive multitasking
- Process scheduler decides which process runs and when
- 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 (tasks spawned by Linux's I/O subsystem)
- Linus uses two process scheduling algorithms for user processes:
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real time algorithm for tasks where absolute priorities are more important than fairness

CFS

- Linux Kernel version 2.6 introduced Completely Fair Scheduler (CFS)
  - A new scheduling algorithm – preemptive, priority-based algorithm with two separate priority ranges
  - Real time priorities range from 0 to 99
  - Nice value of process ranging from -20 to 19
    - Set using nice() system call
  - Smaller nice values indicate higher priorities – ie., not nice to other processes
- 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
- This allotment is adjusted by weighting each task by its nice value

CFS (Cont.)

- Smaller nice value -> higher weight (higher priority)
- Processes with default nice value have a weight of 1 – priority is unchanged
- Higher nice value → lower weight(lower priority)

- Then each task run with for time proportional to task's weight divided by total weight of all runnable tasks
- To calculate the actual length of a time a process runs
  - A configurable variable **target latency** is used
    - Is the interval of time during which every 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
    - If 1000 tasks, each runs for 1 microsecond
  - Switching costs are involved – scheduling processes for short lengths of time is inefficient

CFS (Cont.)

- Therefore, another configuration variable called the **minimum granularity** is used
  - It is the minimum amount of time any process is allotted the processor's time
  - Regardless of the latency, the process will run for at least the minimum granularity
- **Minimum granularity** ensures each run has reasonable amount of time
- Advantage: ensures that switching costs do not grow unacceptably large when the number of runnable processes becomes too large
- Disadv: actually violates fairness idea

Real time scheduling

- Two scheduling classes
  - FCFS and round robin
  - Along with the notion of priority
- Linux ensures that higher priority process will run first. Among processes with equal priority the one with longest waiting time is scheduled next.
- Difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue
- Round-robin processes of equal priority will automatically time-share among themselves.
- Linux's real-time scheduling is soft—rather than hard—real time.
- The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees on deadlines

Kernel Synchronization

- Kernel scheduling its own tasks is different from how is schedules user processes
- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt
- All these tasks may handle the same data structures
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section

Kernel Synchronization (Cont.)

- Linux uses two techniques to protect critical sections:
  - Normal kernel code is non preemptible (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
  - After 2.6 version, provides spin locks, semaphores, and reader-writer versions of both to achieve synchronization
    - Behavior modified if on single processor or multi:
    - Spinlocks are used when locks are held for short durations. For longer periods, semaphores are used.

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

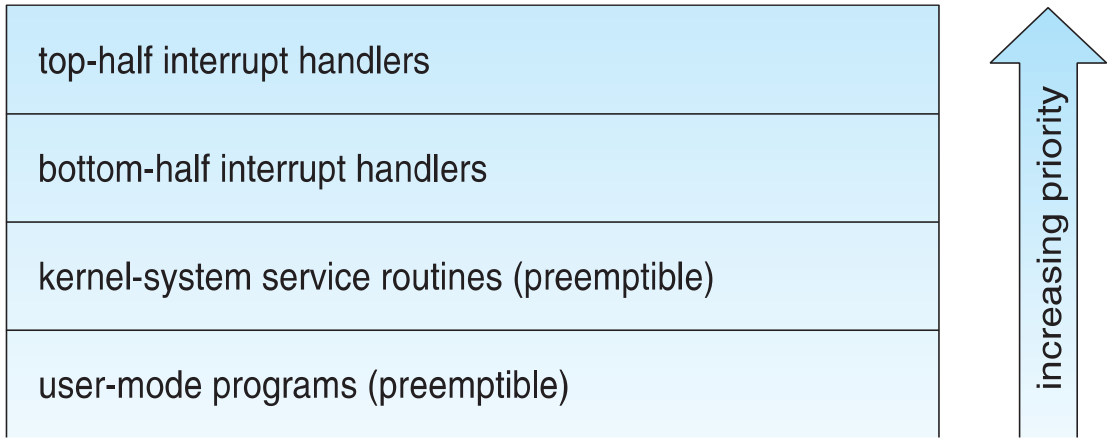
Kernel Synchronization (Contd)

- The second technique applies to critical sections that occur in an interrupt service routines
  - By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.
  - However, there is a penalty for this because,
    - In most hardware architectures, enabling/disabling interrupts instructions are expensive
    - Disabling interrupts means suspending all I/O – performance degradation
- To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled.
  - This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

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. Interrupts of the same number (or line) are disabled, but other interrupts mayrun.
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

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

Memory Management

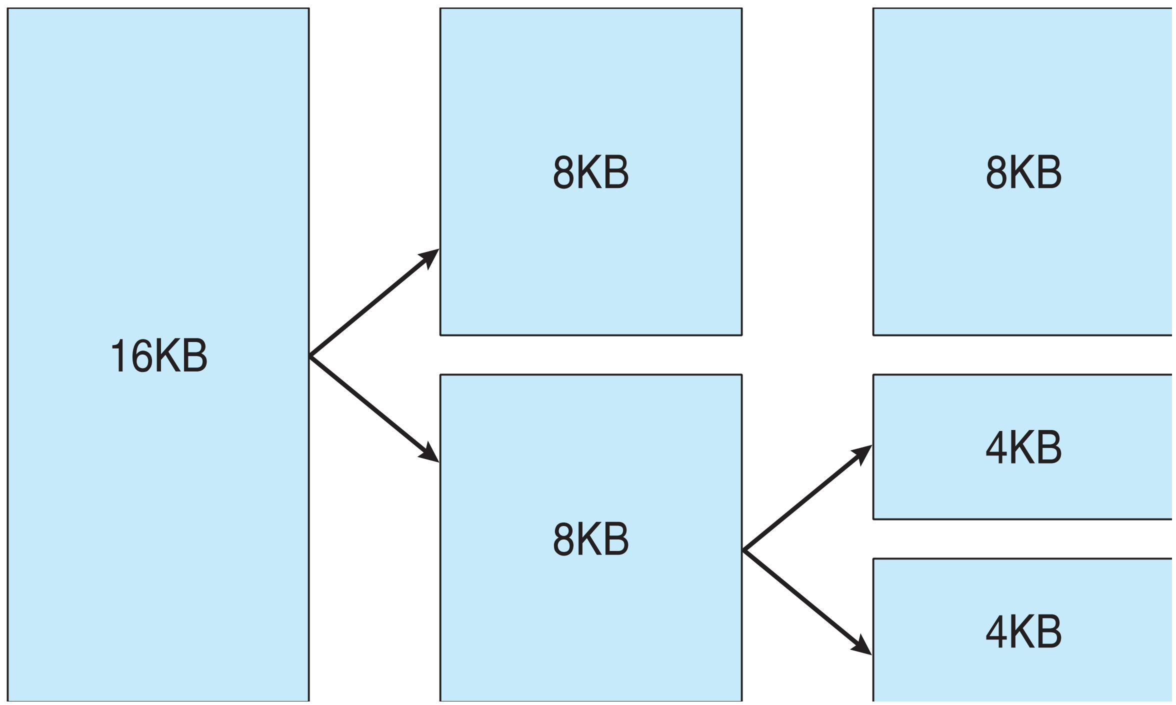
- Linux's physical memory-management system deals with
  - Handling physical memory - allocating and freeing pages, groups of pages, and small blocks of memory
  - handling virtual memory - memory mapped into the address space of running processes
- Linux is available for a wide range of architectures – there needs to be an architecture independent way of describing memory
- Splits memory into four different zones due to hardware characteristics
  - 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

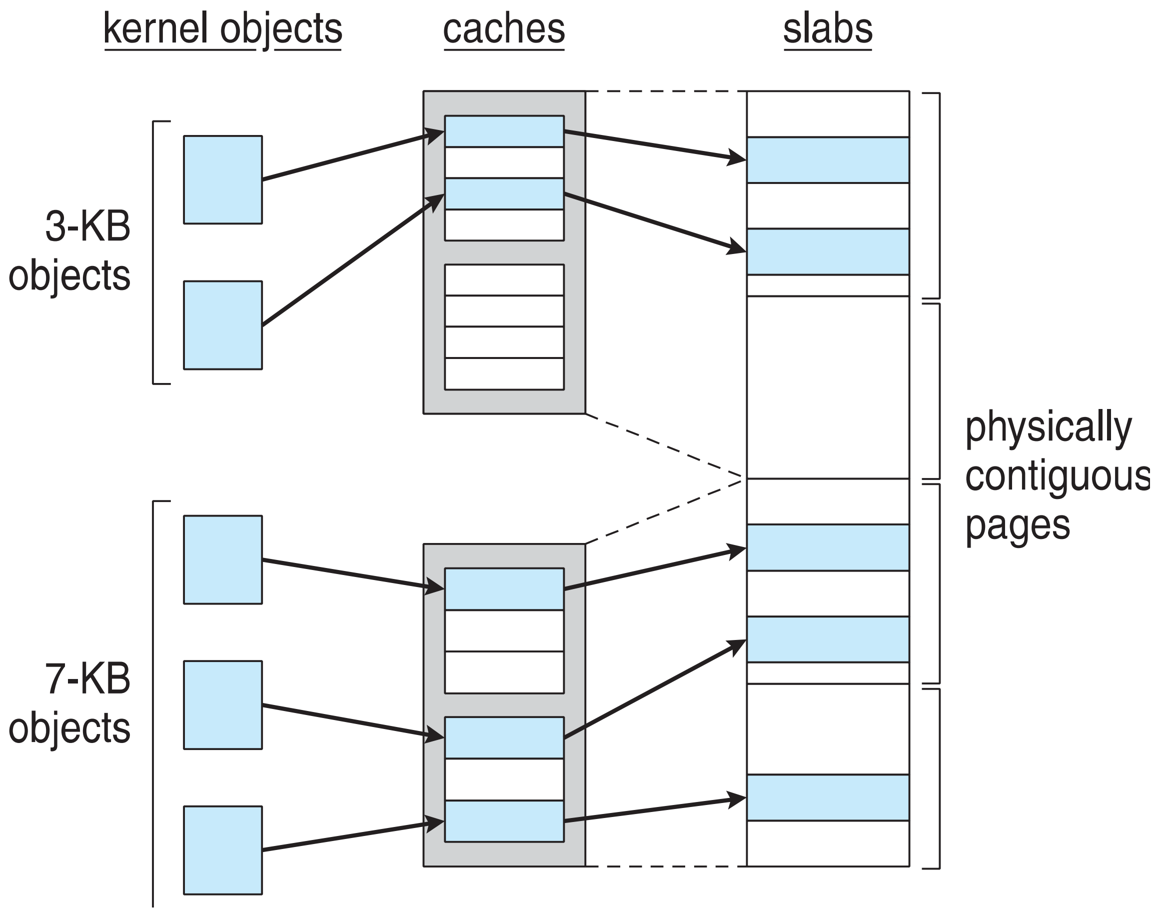
Splitting of Memory in a Buddy Heap



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
  - Kernel data structures – process descriptors, semaphores, file objects
  - Slabs – caches – objects
  - Each cache represents a unique data structure
  - Caches are filled with instances of objects along with its state
  - Eg: process descriptors – 1.7KB
  - Slab states:
    - Free, Full, Partial
  - Allocation starts in partial if available,then free . If no free slabs, create a new slab and add it to cache

Slab Allocator in Linux



Eg: 12KB slab (made up of three 4KB pages) can hold 6 2KB objects

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

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 the fork, the parent and child share the same physical pages of memory in their address spaces
    - Except for pages containing private region of parent. They are marked as read-only and copy-on-write when it is copied to child. If either of the process modifies it, a copy is created for their use.

Swapping and Paging

- The VM paging system relocates pages (unlike UNIX which swaps the entire process) 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
    - Modified version of second chance algorithm + LFU
    - Multiple pass clock – i.e.,for each pass of the clock, "age" of the process is incremented for frequently used pages and decremented for others)
  - The **paging mechanism** actually carries out the transfer, and pages data back into physical memory as needed

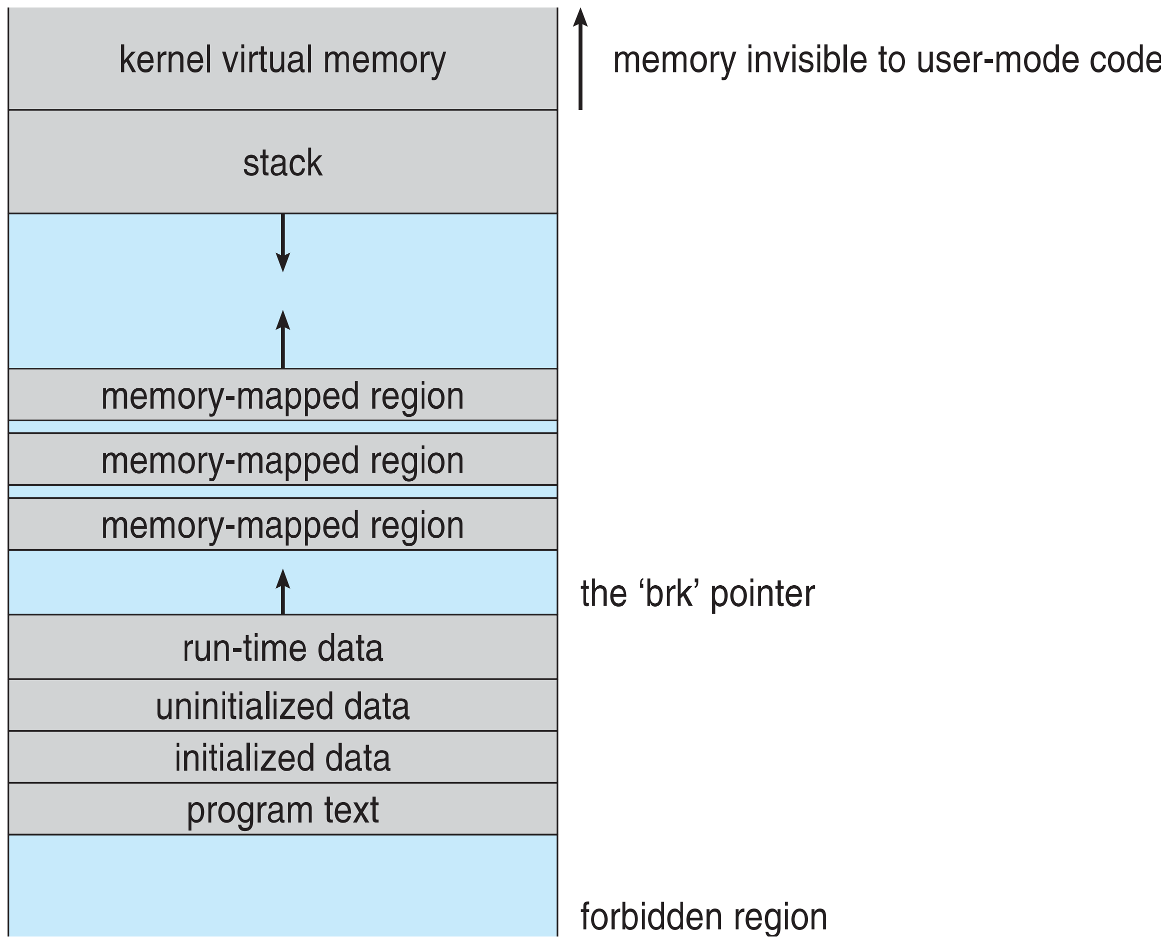
Kernel Virtual Memory

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use
- This region is invisible for that process when running in user mode
- This kernel virtual-memory area in the process contains two regions:
  - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code
  - 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

Executing and Loading User Programs

- Exec() – used to load a new program
- No single loader function in Linux
- Instead, 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 (Executable and Linkable format) and a.out binary formats
  - ELF is flexible – new sections can be added in to the binary without the loader being confused
- 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-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



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
  - Adv: statically linked executables can commence running as soon as they are loaded.
  - Disadv: 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
  - Every dynamically linked program contains small statically linked function called when process starts
  - Maps the link library into memory
  - Link library determines dynamic libraries required by process and names of variables and functions needed
  - Maps libraries into middle of virtual memory and resolves references to symbols contained in the libraries
  - It does not matter exactly where in memory these shared libraries are mapped: Shared libraries compiled to be position-independent code (PIC) so can be loaded anywhere

File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- UNIX files can be anything capable of handling the input or output of a stream of data.
  - Eg: Device drivers, IPC channels or network connections also look like files to the user.
- Internally, the kernel hides implementation details of the above file types 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
  - and a layer of software to manipulate the objects.

File Systems (Cont.)

- VFS defines four main object types:
  - 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.)

- For each of these four object types, the VFS defines a set of operations.
- Every object of one of these types contains a pointer to a function table that lists the addresses of the actual functions that implement the defined operations for that object.
- Eg: for the file object operations,
  - 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

Linux File system objects

Inode and File objects

- An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents (and also the owner, size and time of creation) and
- a file object represents a point of access to the data in an open file (where in the file the process is currently reading or writing) and its permissions
- A process cannot access an inode's contents without first obtaining a file object pointing to the inode.
- File objects typically belong to a single process, but inode objects do not. There is one file object for every instance of an open file, but always only a single inode object.
- Even when a file is no longer in use by any process, its inode object may still be cached by the VFS to improve performance for future requests

Linux File system objects (Contd.,)

Superblock object

- The superblock object represents a connected set of files that form a self-contained file system.
- OS maintains one superblock object for each disk device mounted as a file system and for each networked file system currently connected.
- The main responsibility of the superblock object is to provide access to inodes.
- The VFS identifies every inode by a unique file-system/inode number pair and gets this info from the superblock object

Dentry object

- A dentry object represents a directory entry, which may include the name of a directory in the path name of a file (such as /usr) or the actual file (such as stdio.h).
- Eg: the file /usr/include/stdio.h contains the directory entries (1) /, (2) usr, (3) include, and (4) stdio.h. Each of these values is represented by a separate dentry object.

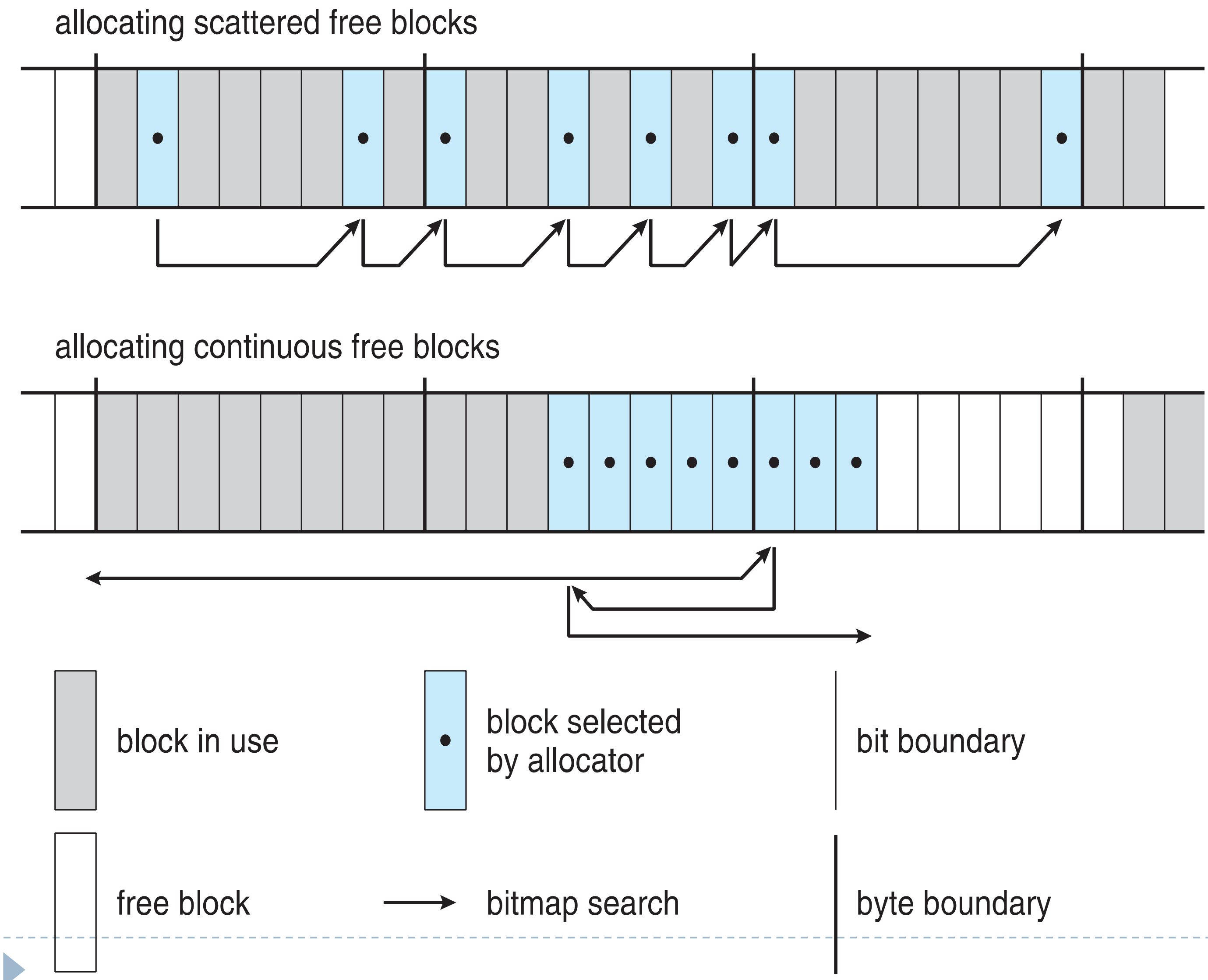
The Linux ext3 File System

- ext3 is standard on disk file system for Linux
  - Uses a mechanism similar to that of BSD Fast File System (FFS) for locating data blocks belonging to a specific file
  - Indirect blocks – multileveled
  - Supersedes older extfs, ext2 file systems
  - Work underway on ext4 adding features like extents

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
    - The default block size on ext3 varies as a function of total size of file system with support for 1, 2, 4 and 8 KB blocks
  - 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

Ext2fs Block-Allocation Policies



Journaling

- ext3 implements journaling, with file system updates first written to a log file in the form of transactions
  - Once in log file, considered committed
  - 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
  - Must be completed once system recovers
  - No other consistency checking is needed after a crash (much faster than older methods)
- Improves write performance on hard disks by turning random I/O on disk into sequential I/O in journal. Later, the journal entries are replayed asynchronously into the file system.

Thank You