The Linux System

Chapter 21

- The slides do not contain all the information. So cannot be treated as a study material for Operating System. Please refer the text book for exams.

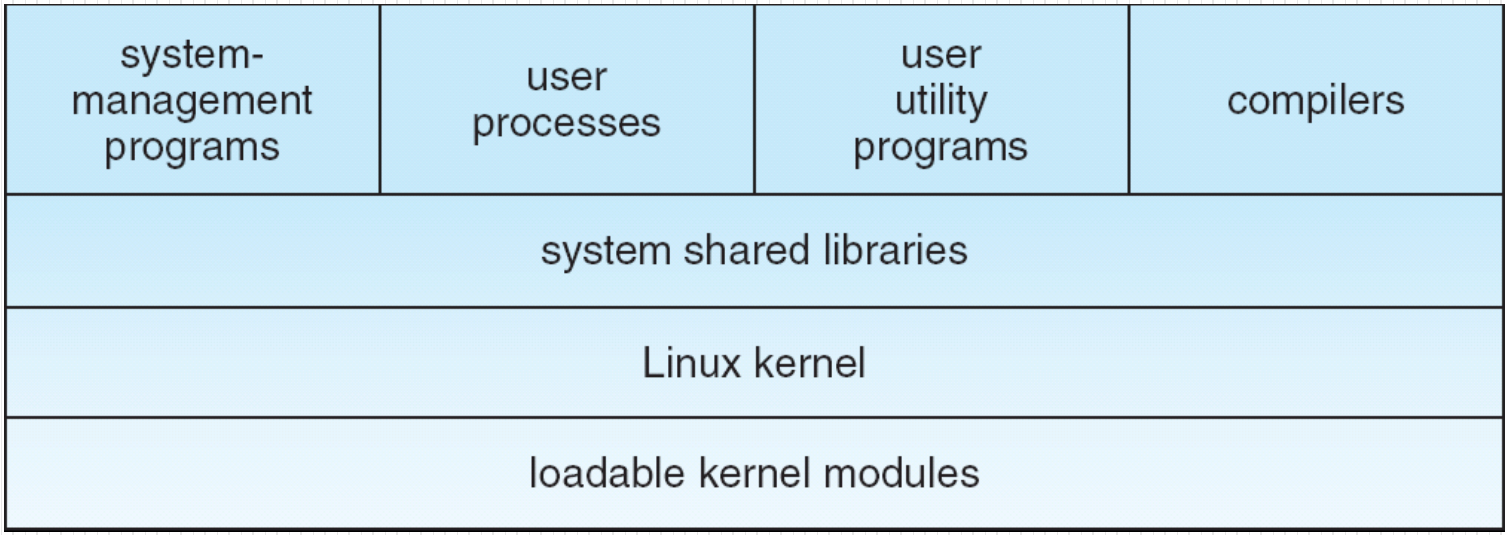
Topics

- Design Principles
- Kernel Modules
- Process Management
- Scheduling
- Memory Management
- File Systems
- Interprocess Communication
- Security

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
- The Linux programming interface adheres to the UNIX semantics, rather than to BSD behavior

Components of a Linux System



Components of a Linux System (Cont.)

- 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. Some utilities may be invoked just once to initialize and configure some aspect of the system; others known as daemons may run permanently – check for incoming network connections

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.

Kernel Modules

- Three components to Linux module support:
  - Module management
  - Driver registration
  - Conflict resolution

Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Reference made to kernel symbols are point to correct location in kernel's address space
- 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

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 – character devices(printer, terminals) , block devices(disk drives)
  - File systems - implement a format for storing files on disk
  - Network protocols – module may implement a packet filtering rules
  - Binary format – specifies a way of recognizing and loading a new type of executable file

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* – device driver probes that auto detect device configuration from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware

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`
- This model is advantage of simplicity as we don't have to specify every detail of the environment of a new program in a system call that runs the program
- Process properties fall into three groups:
  - the process's identity
  - Environment
  - 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

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.
- Identity and environment are set when the process is created and not changed until the process exits
- Process context includes the following

1. 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.
2. 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.
3. 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.

Process Context (Cont.)

4. 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 are stored here.
5. The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive.
6. 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
- Using `clone` gives an application fine-grained control over exactly what is shared between two threads.

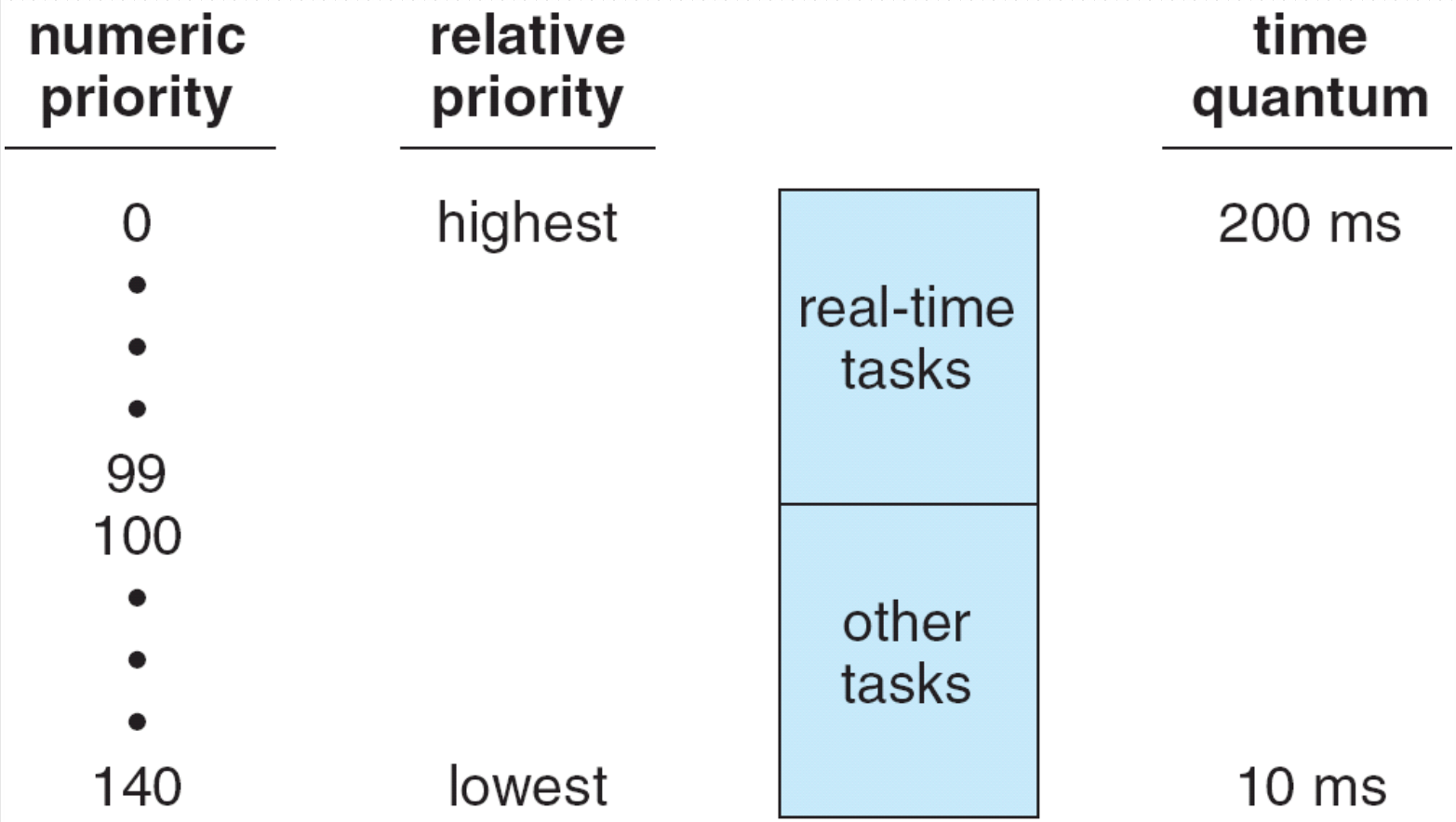
Processes and Threads

- When clone is invoked it is passed with a set of flags that determine how much sharing is to take place between the parent and child tasks
- Some of the flags are
- CLONE_FS – share file system info such as pwd
- CLONE_VM – share memory space
- CLONE_SIGHAND – same signal handlers
- CLONE_FILES – same set of files
- Lack of distinction between process & threads is as linux does not hold a process's entire context within the main process data structure – rather within independent subcontexts

Scheduling

- The job of allocating CPU time to different tasks within an operating system.
- Linux has 2 separate process scheduling
  - Time sharing algorithm for fair preemptive scheduling among multiple processes
  - Real time tasks absolute priorities are more important than fairness
- As of 2.5, new scheduling algorithm – preemptive, priority-based
  - Real-time range
  - nice value

Relationship Between Priorities and Time-slice Length



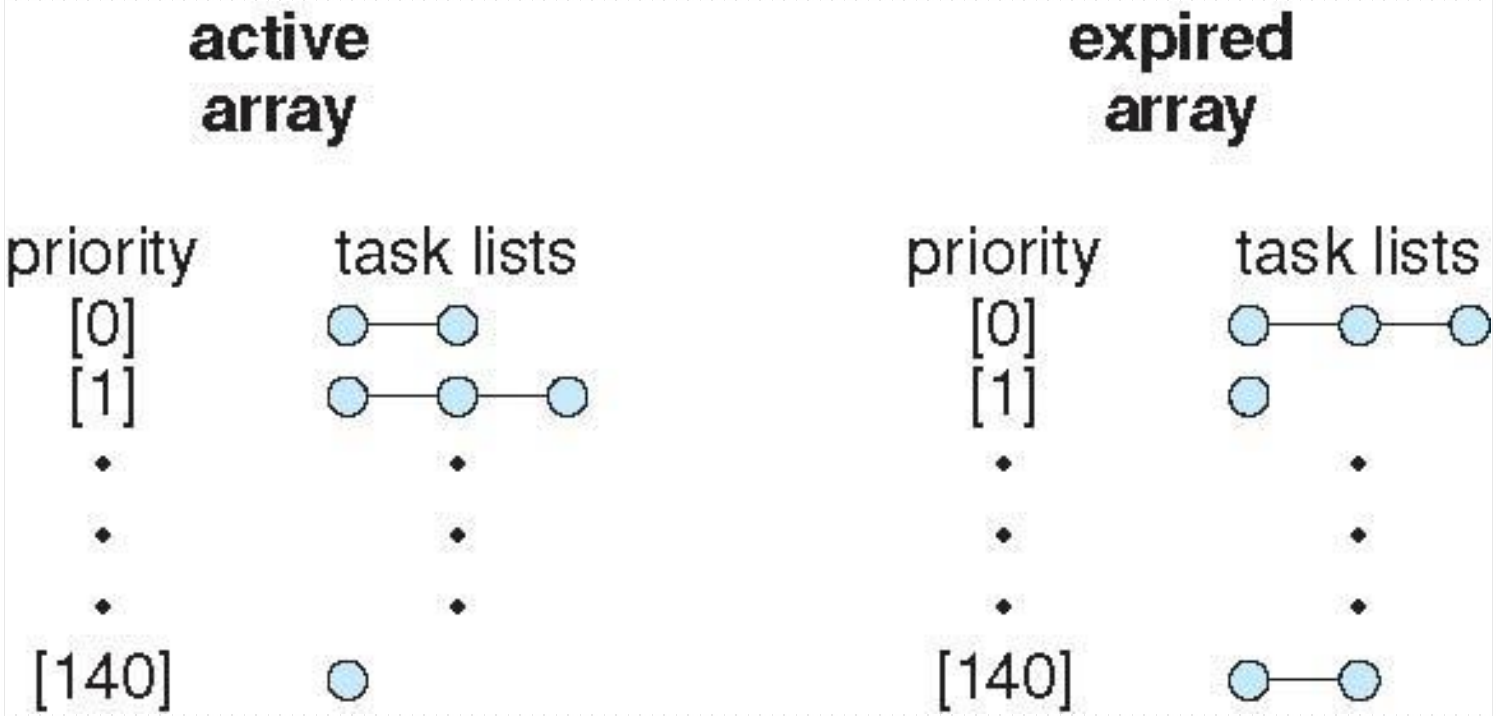
A real time range value from 0 to 99 and nice value ranging from 100 to 140

List of Tasks Indexed by Priority

A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice.

When a task has exhausted its time slice it is considered **expired** and not eligible for execution again until all other tasks have exhausted their quanta.

Once all tasks finish their time quanta both the arrays are exchanged



Process Scheduling

- Tasks are assigned dynamic properties that are based on the nice value plus or minus a value up to the value 5
- Whether the value is added or subtracted from a task's nice value depends on the interactivity of the task
- Tasks that are more interactive have more sleep time and have nice value – 5
- Task's dynamic property is recalculated when the task has exhausted its time quantum and is moved to expired array
- When arrays are exchanged all tasks in the new active array have new priorities and corresponding time slices

Process Scheduling

- Real time scheduling
- Linux implements two real time scheduling tasks
  - FCFS
  - Round Robin
- The scheduler always runs process with the highest priority
- Among the process of equal priority it runs the process that has been waiting the longest
- Real time tasks are assigned static priorities
- Real time scheduling is soft rather than hard real time – scheduler guarantees about relative priorities of real time process, but the kernel does not offer any guarantees about how quickly real time process will become runnable

Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt

- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section.

Kernel Synchronization (Cont.)

- Prior to version 2.6 was a non preemptive kernel – a process running in kernel mode could not be preempted even if higher priority process became available to run
- Linux kernel provides spinlocks and semaphores for locking the kernel
- On a SMP machines the fundamental locking mechanism is spinlock

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

- Two system calls preempt_disable() and preempt_enable() – not preemptable if kernel is holding locks – preempt_count for a task

Kernel Synchronization (Cont.)

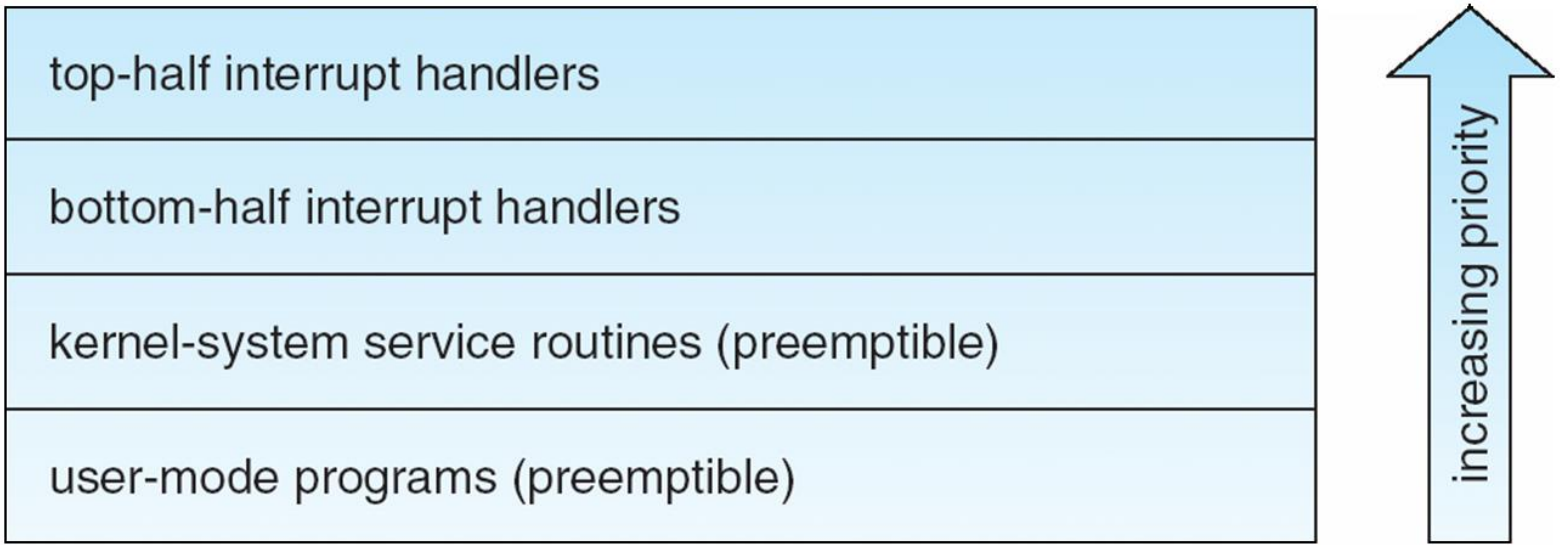
- Second technique used in Interrupt service routine
- The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.
- On most hardware architectures, interrupt enable and disable instructions are expensive.
- As long as interrupts remain disabled, all I/0 is suspended, and any device waiting for servicing will have *to* wait until interrupts are reenabled; so performance degrades.

Kernel Synchronization (Cont.)

- 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.

- Interrupt service routines are separated into a *top half* and a *bottom half*.
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

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.
- This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself

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.
- 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.
- In Version 2.2 of the kernel, a single kernel spinlock (sometimes termed BKL for "**big kernel lock**") was created to allow multiple processes running on different processors) to be active in the kernel concurrently.
- Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel's data structures.

Memory Management

- Memory management under Linux has two components
- First deals with allocating and freeing pages, groups of pages, and small blocks of memory.
- Second handles virtual memory, memory mapped into the address space of running processes.
- Linux Splits memory into 3 different **zones** due to hardware characteristics of Intel's 8086

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

Memory Management

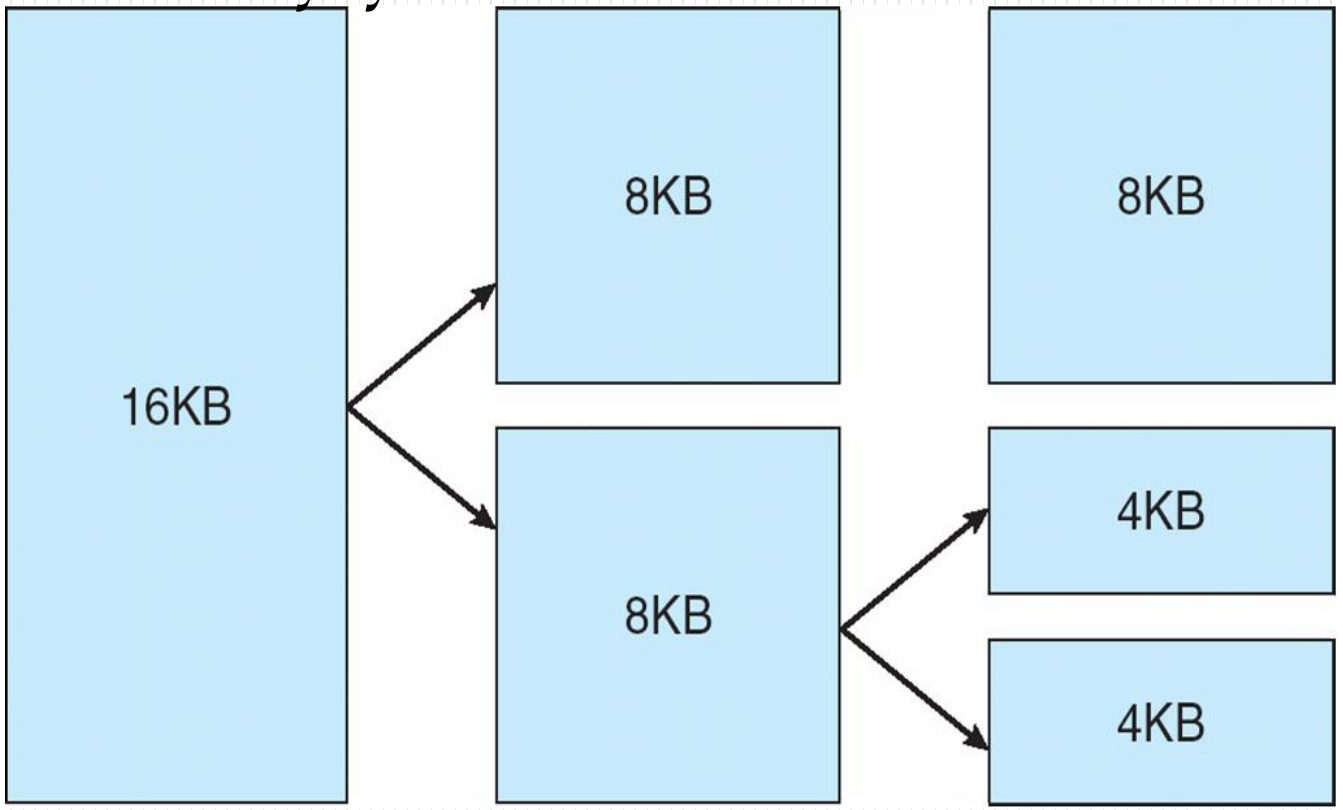
- On these systems, the first 16 MB of physical memory comprise ZONE_DMA.
- ZONE_NORMAL identifies physical memory that is mapped to the CPU's address space – is used for most routine memory requests.
- Finally, ZONE_HIGHMEM (for "high memory") refers to physical memory that is not mapped into the kernel address space.
- For example, on the Intel's 32-bit Intel architecture (where 2^{32} provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as high memory and is allocated from ZONE_HIGHMEM.

Splitting of Memory in a Buddy Heap

The primary physical manager is **page allocator**

Each zone has its own allocator who is responsible for allocating and freeing all physical pages for the zone

Allocator uses **buddy system**



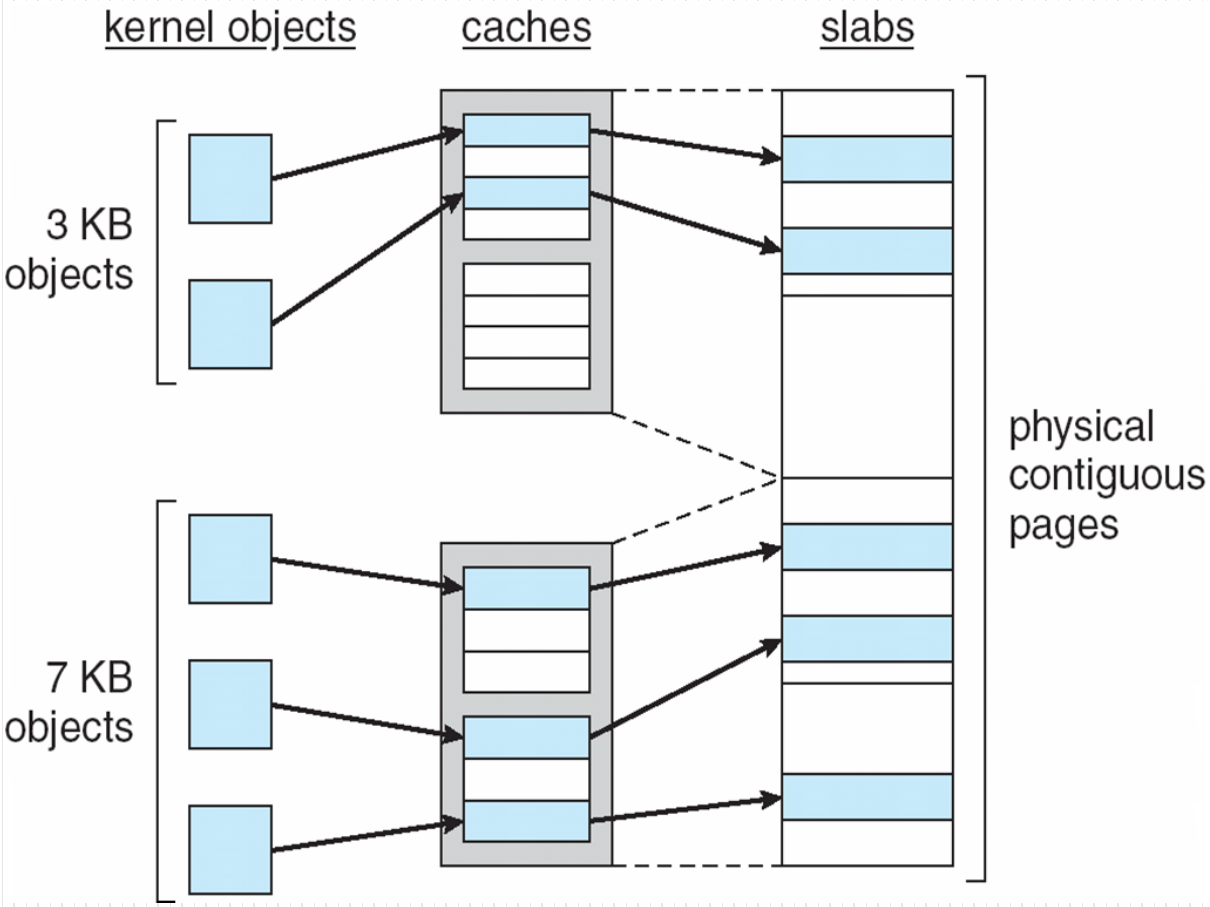
Managing Physical Memory

- 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.

Management of physical memory

- All memory allocation are made by
  - drivers that reserve a contiguous area of memory during system boot time
  - Dynamically by the page allocator
- Kernel functions dont have to use the basic allocator to reserve memory
- Several subsystems use the underlying page allocator to manage their own pools of memory. The most important are
  - Kmalloc() – variable length allocator – allocates entire page but splits them into smaller pieces
  - **slab allocator** allocating memory for kernel data structure
  - Page cache used for caching pages that belong to files

Slab Allocation



A slab may be in one of the 2 states

- **Full**
- **Empty**
- **Partial**

Management of physical memory

- Two other main subsystems in Linux do their own management of physical pages
  - Page cache
  - Virtual memory system
- **Page cache** is kernel's main cache for the block devices and the main mechanism through which I/O to these devices are performed
- Page cache stores entire pages of the file contents and is not limited to block devices – it can also cache network data
- **Virtual memory system** manages contents of each process's virtual address space .
- These two systems interact closely – reading a page of data into the page cache requires mapping of each process's virtual address space

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 nonoverlapping 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) - when a process tries to read a page in such a region it is given back a page of memory filled with zero's.
  - The region's reaction to writes– Mapping of the region into the process address space can be **private** or **shared**
  - If a process writes to a privately mapped region, then the pager detects a copy on write to keep changes local
  - Writes to shared region result in updating of the object mapped into that region, so that change is visible immediately

Virtual Memory (Cont.)

- **Lifetime of a virtual address space**
- 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
- 1st case – the process is given a new, empty virtual address space – It is upto the routines for loading the program to populate the address space
- 2nd case – 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 vm_area_struct descriptors, then creates a new set of page tables for the child.

Virtual Memory (Cont.)

- 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.

- A special case occurs when the copying operation reaches a virtual memory region that is privately mapped – copy on write to modify and the reference count is updated in the page table accordingly.
- This mechanism ensures that private data pages are shared between processes whenever possible ; copies are made only when absolutely necessary

Virtual Memory (Cont.)

- **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 **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
- Pageout policy uses modified version of second chance
- Multiple pass clock is used and every page has an age that is adjusted on each pass of the clock

Virtual Memory (Cont.)

- Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass
- This age valuing allows the pager to select pages to page out based on least frequently used policy
- Paging mechanism supports paging to swap devices and normal file – swapping to file is slow due to overhead of file system
- It uses next fit(first fit) to write to contiguous runs of disk block

Virtual Memory (Cont)

- **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.
- The page table entries that map to these kernel pages are marked protected so that pages are not modified in user mode
- This kernel virtual-memory area 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.

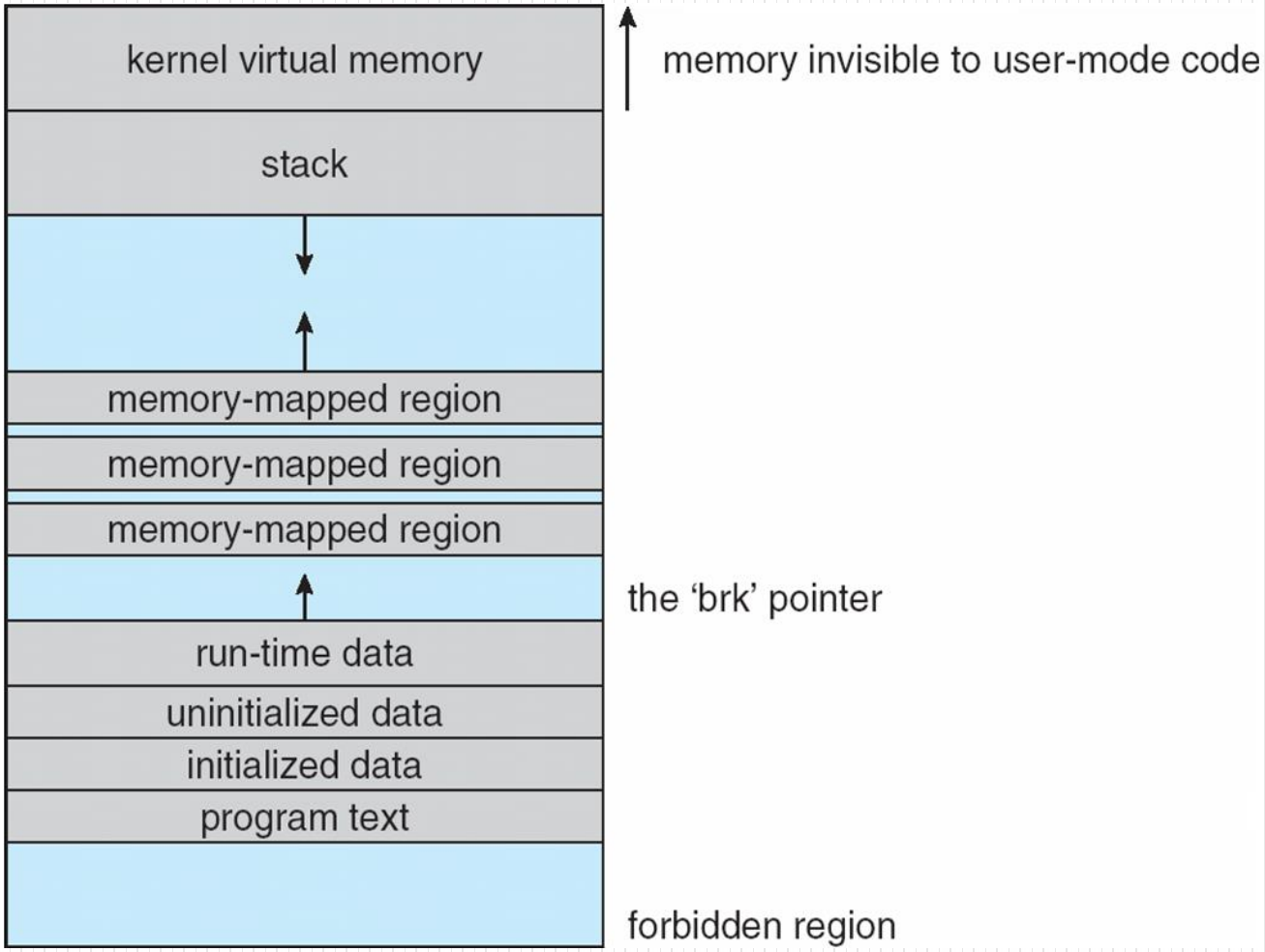
Executing and Loading User Programs

- **Execution and loading of 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 first job of the service is to check if the calling process has permission
- The task of the loader is to set up the mapping of program 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 – demand paging
- Older linux understood a.out for binary files
- New system use ELF format (Extensible Linkable Format)
- The registration of multiple loader routines allows Linux to support both the ELF and **a.out** binary formats.

Executing and Loading User Programs

- **Execution and loading of 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.
- 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



Mapping of programs into Memory

- ELF format has header followed by several page aligned sections
- In a reserved region at one end of the address space is the kernel
- Regions that need to be initialized is stack and program's text and data region
- Stack is created on top of user mode and it grows downwards towards lower numbered addresses
- At the end it has environment variables
- Section that contains program text is read only and write protected
- Writable initialized data are mapped next
- Any uninitialized data are mapped in as private demand zero region
- Variable sized region to hold data at run time. Each process had a pointer brk that points to the current extent of this data region

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.
- Linux implements with the help of special linker library – every DL program contains a statically linked function called when the program starts
- Static function maps the link library into memory – which finds which libraries are required and what variables etc - **Position independent code**

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
    - The *inode-object(individual file)* and the *file-object(open file)* structures represent individual files
    - the *superblock object* represents an entire file system
    - Dentry object – represents individual directory entry
  - A layer of software to manipulate those objects.

File Systems

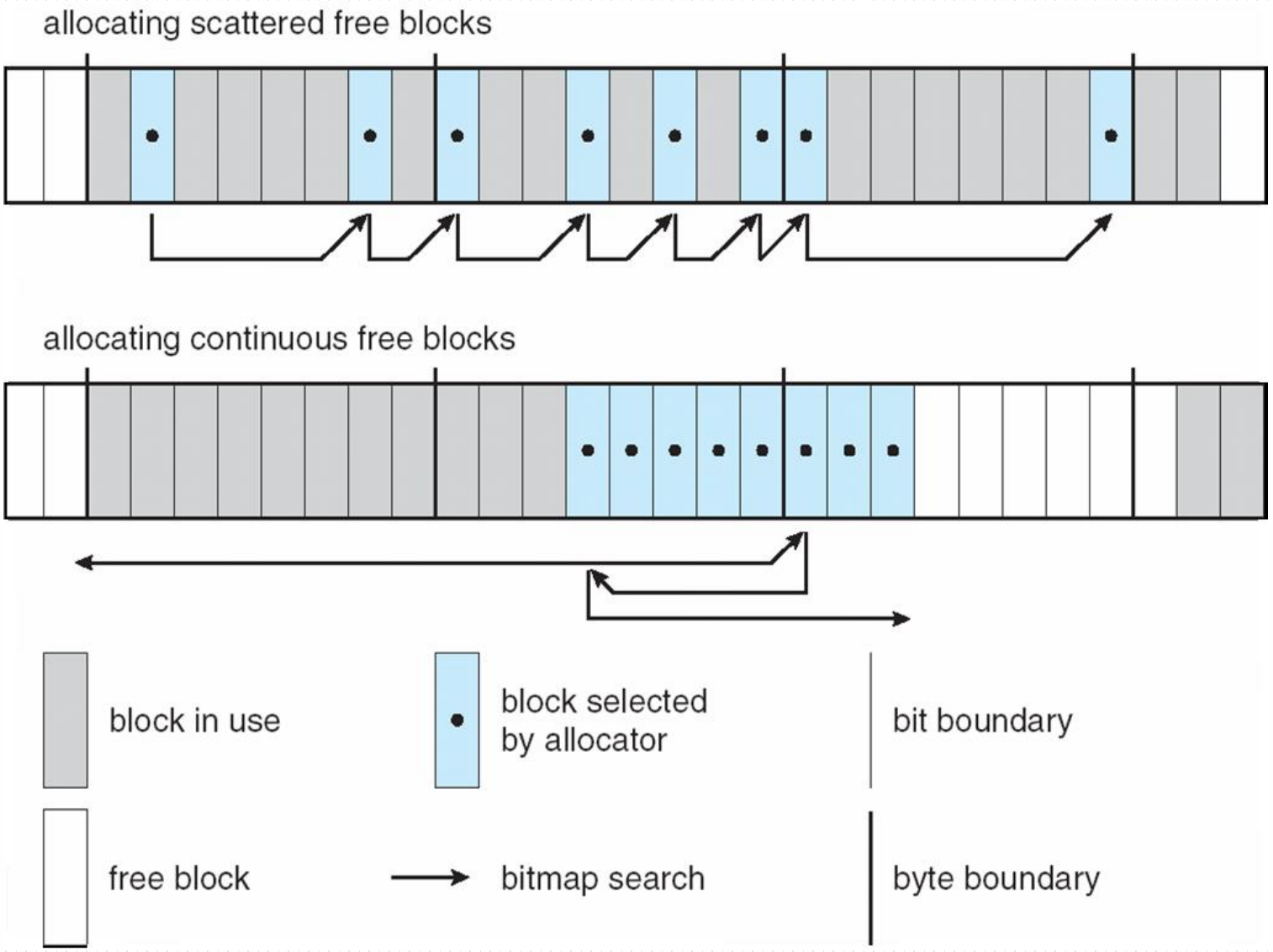
- The operations on these objects are
- Int open(..) – opens a file
- Read()
- Write()
- VFS software perform an operation on file system objects by calling the appropriate function from the object's function table
- Inode and file objects are mechanisms used to access files.
- Inode object is a data structure containing pointers to the disk blocks that contain the actual file contents
- File object represents point of access to the data in an open file.

The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file

- The main differences between ext2fs(second extended file system) and ffs (BSD fast file system) 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
  - Ext2fs does not use fragments; it performs its allocations in smaller units
    - The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported
  - Ext2fs uses 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

Ext2fs Block-Allocation Policies



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 with 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 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.
- To obtain synchronization we could use semaphores

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.