Operating Systems

CSE 231

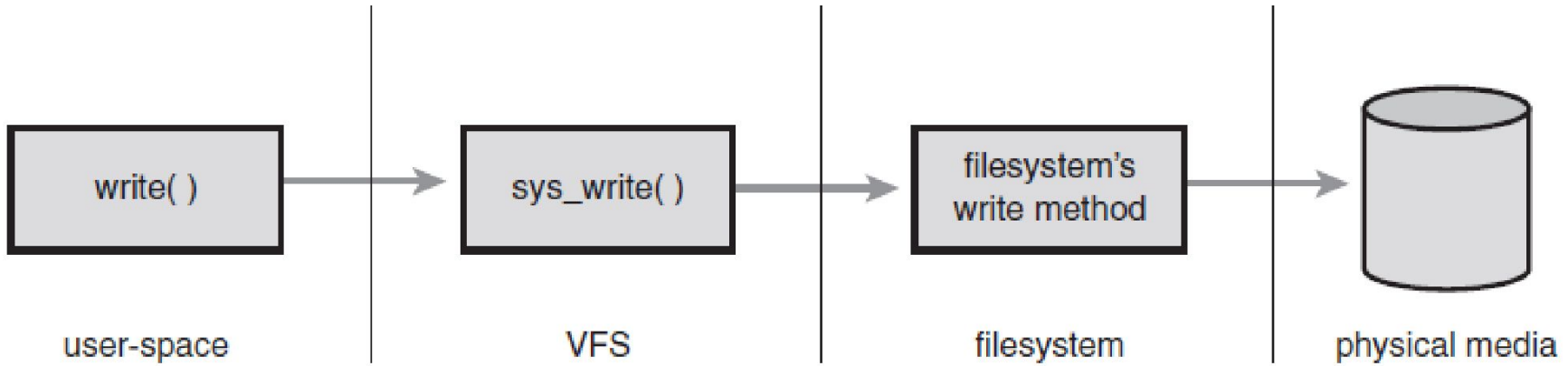
Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 11: Nov. 30 – Dec. 3

Linux/Unix Virtual File System (aka Virtual File Switch aka VFS)

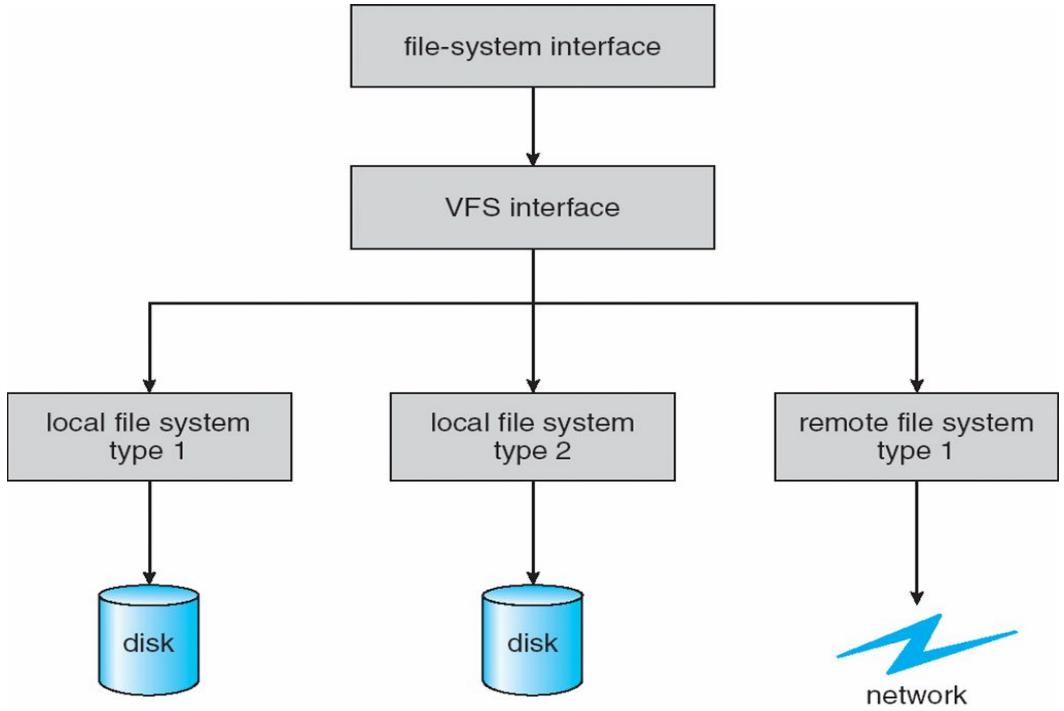
- Every file system entity is either a file, directory, inode or a mount point.
- VFS provides the unified user-level view of all entities irrespective of the underlying filesystem and media

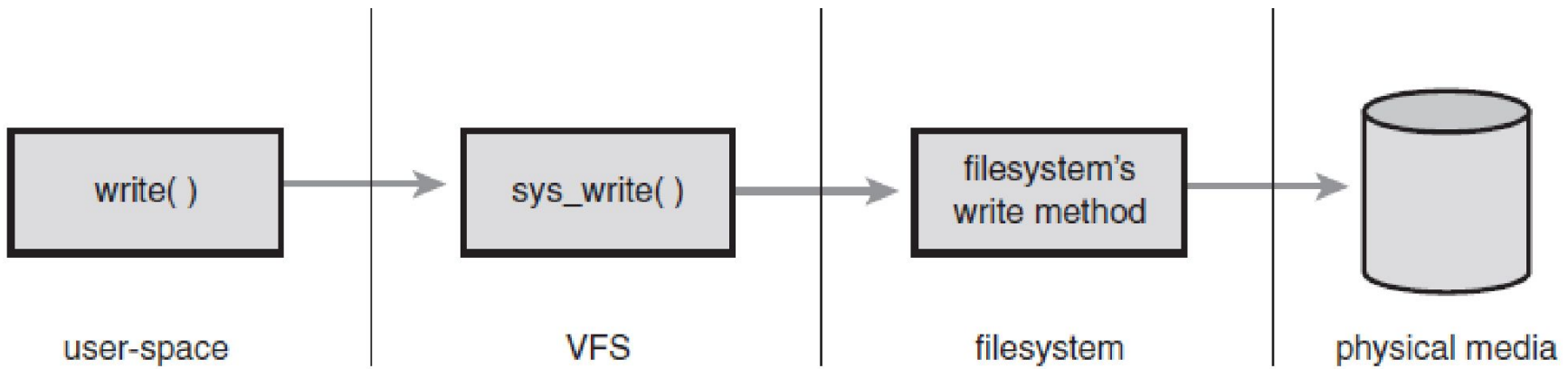


Virtual File Systems

- **Virtual File Systems** (**VFS**) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines
- The API is to the VFS interface, rather than any specific type of file system

VFS Basics


file-system interface
VFS interface
local file system type 1
local file system type 2
remote file system type 1
disk
disk
network


write( )
sys_write( )
filesystem's write method
user-space
VFS
filesystem
physical media

Unix/Linux Inode

- Unix systems separate the concept of a file from any associated information about it, such as access permissions, size, owner, creation time, and so on (aka *metadata).*
- Metadata stored in a file called an *inode*.

VFS Objects and Their Data Structures

- VFS is objected-oriented.
  - Superblock: Represents a mounted filesystem/partition.
  - Inode: Represents metadata a specific file.
  - Dentry: Represents a directory entry.
  - File: Represents an open file as associated with a process.

VFS operations

- The super_operations object, which contains the methods that the kernel can invoke on a specific filesystem, such as write_inode() and sync_fs().
- The inode_operations object, which contains the methods that the kernel can invoke on a specific file, such as create() and link().
- The dentry_operations object, which contains the methods that the kernel can invoke on a specific directory entry, such as d_compare() and d_delete().
- The file_operations object, which contains the methods that a process can invoke on an open file, such as read() and write().

Superblock Object

The superblock object is implemented by each filesystem and is used to store information describing that specific filesystem.

```
struct super_block {
        struct list_head        s_list;         /* list of all superblocks */
        dev_t                   s_dev;          /* identifier */
        unsigned long           s_blocksize;    /* block size in bytes */
        unsigned char           s_blocksize_bits; /* block size in bits */
        unsigned char           s_dirt;         /* dirty flag */
        unsigned long long      s_maxbytes;     /* max file size */
        struct file_system_type s_type;         /* filesystem type */
        struct super_operations s_op;           /* superblock methods */
        struct dquot_operations *dq_op;         /* quota methods */
        struct quotactl_ops     *s_qcop;        /* quota control methods */
        struct export_operations *s_export_op;  /* export methods */
        unsigned long           s_flags;        /* mount flags */
        unsigned long           s_magic;        /* filesystem's magic number */
        struct dentry           *s_root;        /* directory mount point */
        struct rw_semaphore     s_umount;       /* unmount semaphore */
        struct semaphore        s_lock;         /* superblock semaphore */
        int                     s_count;        /* superblock ref count */
        int                     s_need_sync;    /* not-yet-synced flag */
        atomic_t                s_active;       /* active reference count */
        void                    *s_security;    /* security module */
        struct list_head        s_inodes;       /* list of inodes */
        struct list_head        s_dirty;        /* list of dirty inodes */
        struct list_head        s_io;           /* list of writebacks */
        struct list_head        s_more_io;      /* list of more writeback */
        struct hlist_head       s_anon;         /* anonymous dentries */
        struct list_head        s_files;        /* list of assigned files */
        struct list_head        s_dentry_lru;   /* list of unused dentries */
        int                     s_nr_dentry_unused; /* number of dentries on list */
        struct block_device     *s_bdev;        /* associated block device */
        struct mtd_info         *s_mtd;         /* memory disk information */
        struct list_head        s_instances;    /* instances of this fs */
        struct quota_info       s_dquot;        /* quota-specific options */
        int                     s_frozen;       /* frozen status */
        wait_queue_head_t       s_wait_unfrozen; /* wait queue on freeze */
        char                    s_id[32];       /* text name */
        void                    *s_fs_info;     /* filesystem-specific info */
        fmode_t                 s_mode;         /* mount permissions */
        struct semaphore        s_vfs_rename_sem; /* rename semaphore */
        u32                     s_time_gran;    /* granularity of timestamps */
        char                    *s_subtype;     /* subtype name */
        char                    *s_options;     /* saved mount options */
};
```

Superblock Operations

s_op superblock object which points to table of following function:

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *);
  int (*write_inode) (struct inode *, int);
  void (*drop_inode) (struct inode *);
  void (*delete_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  void (*write_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);
  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*clear_inode) (struct inode *);
  void (*umount_begin) (struct super_block *);
  int (*show_options)(struct seq_file *, struct vfsmount *);
  int (*show_stats)(struct seq_file *, struct vfsmount *);
  ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
  ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
  int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

Inode Object

- The inode object represents all the information needed by the kernel to manipulate a file or directory.
- For Unix-like filesystems, this information is simply read from the on-disk inode.
- If a filesystem does not have inodes, however, the filesystem must obtain the information from wherever it is stored on the disk.
- For FS without inodes the metadata is constructed in memory in whatever manner applicable to the system (with help from the FS driver).

Inode Object

```
struct inode {
        struct hlist_node       i_hash;         /* hash list */
        struct list_head        i_list;         /* list of inodes */
        struct list_head        i_sb_list;      /* list of superblocks */
        struct list_head        i_dentry;       /* list of dentries */
        unsigned long           i_ino;          /* inode number */
        atomic_t                i_count;        /* reference counter */
        unsigned int            i_nlink;        /* number of hard links */
        uid_t                   i_uid;          /* user id of owner */
        gid_t                   i_gid;          /* group id of owner */
        kdev_t                  i_rdev;         /* real device node */
        u64                     i_version;      /* versioning number */
        loff_t                  i_size;         /* file size in bytes */
        seqcount_t              i_size_seqcount;/* serializer for i_size */
        struct timespec         i_atime;        /* last access time */
        struct timespec         i_mtime;        /* last modify time */
        struct timespec         i_ctime;        /* last change time */
        unsigned int            i_blkbits;      /* block size in bits */
        blkcnt_t                i_blocks;       /* file size in blocks */
        unsigned short          i_bytes;        /* bytes consumed */
        umode_t                 i_mode;         /* access permissions */
        spinlock_t              i_lock;         /* spinlock */
        struct rw_semaphore     i_alloc_sem;    /* nests inside of i_sem */
        struct semaphore        i_sem;          /* inode semaphore */
        struct inode_operations *i_op;          /* inode ops table */
        struct file_operations  *i_fop;         /* default inode ops */
        struct super_block      *i_sb;          /* associated superblock */
        struct file_lock        *i_flock;       /* file lock list */
        struct address_space    *i_mapping;     /* associated mapping */
        struct address_space    i_data;         /* mapping for device */
        struct dquot            *i_dquot[MAXQUOTAS]; /* disk quotas for inode */
        struct list_head        i_devices;      /* list of block devices */
        union {
            struct pipe_inode_info  *i_pipe;    /* pipe information */
            struct block_device     *i_bdev;    /* block device driver */
            struct cdev             *i_cdev;    /* character device driver */
        };
        unsigned long           i_dnotify_mask; /* directory notify mask */
        struct dnotify_struct   *i_dnotify;     /* dnotify */
        struct list_head        inotify_watches; /* inotify watches */
        struct mutex            inotify_mutex;  /* protects inotify_watches */
        unsigned long           i_state;        /* state flags */
        unsigned long           dirtied_when;   /* first dirtying time */
        unsigned int            i_flags;        /* filesystem flags */
        atomic_t                i_writecount;   /* count of writers */
        void                    *i_security;    /* security module */
        void                    *i_private;     /* fs private pointer */
};
```

An inode represents each file on a filesystem, but the inode object is constructed in memory only as files are accessed. This includes special files, such as device files or pipes.

Inode Ops

```
struct inode_operations {
   int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
   struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
   int (*link) (struct dentry *,struct inode *,struct dentry *);
   int (*unlink) (struct inode *,struct dentry *);
   int (*symlink) (struct inode *,struct dentry *,const char *);
   int (*mkdir) (struct inode *,struct dentry *,int);
   int (*rmdir) (struct inode *,struct dentry *);
   int (*mknod) (struct inode *,struct dentry *,int,dev_t);
   int (*rename) (struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
   int (*readlink) (struct dentry *, char __user *,int);
   void * (*follow_link) (struct dentry *, struct nameidata *);
   void (*put_link) (struct dentry *, struct nameidata *, void *);
   void (*truncate) (struct inode *);
   int (*permission) (struct inode *, int);
   int (*setattr) (struct dentry *, struct iattr *);
   int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
   int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
   ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
   ssize_t (*listxattr) (struct dentry *, char *, size_t);
   int (*removexattr) (struct dentry *, const char *);
   void (*truncate_range)(struct inode *, loff_t, loff_t);
   long (*fallocate)(struct inode *inode, int mode, loff_t offset,
                     loff_t len);
   int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                 u64 len);
};
```

Dentry Objects

- To facilitate path lookups, ensuring validity, directory path traversal VFS employs dentry objects.

```
struct dentry {
        atomic_t                d_count;      /* usage count */
        unsigned int            d_flags;      /* dentry flags */
        spinlock_t              d_lock;       /* per-dentry lock */
        int                     d_mounted;    /* is this a mount point? */
        struct inode            *d_inode;     /* associated inode */
        struct hlist_node       d_hash;       /* list of hash table entries */
        struct dentry           *d_parent;    /* dentry object of parent */
        struct qstr             d_name;       /* dentry name */
        struct list_head        d_lru;        /* unused list */
        union {
            struct list_head    d_child;      /* list of dentries within */
            struct rcu_head     d_rcu;        /* RCU locking */
        } d_u;
        struct list_head        d_subdirs;    /* subdirectories */
        struct list_head        d_alias;      /* list of alias inodes */
        unsigned long           d_time;       /* revalidate time */
        struct dentry_operations *d_op;       /* dentry operations table */
        struct super_block      *d_sb;        /* superblock of file */
        void                    *d_fsdata;    /* filesystem-specific data */
        unsigned char           d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
};
```

Dentry ops

```
struct dentry_operations {
        int (*d_revalidate) (struct dentry *, struct nameidata *);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete) (struct dentry *);
        void (*d_release) (struct dentry *);
        void (*d_iput) (struct dentry *, struct inode *);
        char *(*d_dname) (struct dentry *, char *, int);
};
```

File Object

- Used to represent a file opened by a process.
- The file object is the in-memory representation of an open file.

```
struct file {
        union {
            struct list_head    fu_list;        /* list of file objects */
            struct rcu_head     fu_rcuhead;     /* RCU list after freeing */
        } f_u;
        struct path             f_path;         /* contains the dentry */
        struct file_operations  *f_op;          /* file operations table */
        spinlock_t              f_lock;         /* per-file struct lock */
        atomic_t                f_count;        /* file object's usage count */  };
        unsigned int            f_flags;        /* flags specified on open */
        mode_t                  f_mode;         /* file access mode */
        loff_t                  f_pos;          /* file offset (file pointer) */
        struct fown_struct      f_owner;        /* owner data for signals */
        const struct cred       *f_cred;        /* file credentials */
        struct file_ra_state    f_ra;           /* read-ahead state */
        u64                     f_version;      /* version number */
        void                    *f_security;    /* security module */
        void                    *private_data; /* tty driver hook */
```

```
struct list_head        f_ep_links;     /* list of epoll links */
spinlock_t              f_ep_lock;      /* epoll lock */
struct address_space    *f_mapping;     /* page cache mapping */
unsigned long           f_mnt_write_state; /* debugging state */
```

File Operations

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                             unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                              unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int,
                      unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *,
                             int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area) (struct file *,
                                            unsigned long,
                                                                    unsigned long,
                                                                    unsigned long,
                                                                    unsigned long);
        int (*check_flags) (int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write) (struct pipe_inode_info *,
                                 struct file *,
                                 loff_t *,
                                 size_t,
                                 unsigned int);
        ssize_t (*splice_read) (struct file *,
                                loff_t *,
                                struct pipe_inode_info *,
                                size_t,
                                unsigned int);
        int (*setlease) (struct file *, long, struct file_lock **);
};
```

Data Structures Associated with Filesystems

Because Linux supports so many different filesystems, the kernel must have a special structure for describing the capabilities and behavior of each filesystem

```
struct file_system_type {
        const char              *name;     /* filesystem's name */
        int                     fs_flags;  /* filesystem type flags */


        /* the following is used to read the superblock off the disk */
        struct super_block      *(*get_sb) (struct file_system_type *, int,
                                            char *, void *);

        /* the following is used to terminate access to the superblock */
        void                    (*kill_sb) (struct super_block *);

        struct module           *owner;    /* module owning the filesystem */
        struct file_system_type *next;     /* next file_system_type in list */
        struct list_head        fs_supers; /* list of superblock objects */

        /* the remaining fields are used for runtime lock validation */
        struct lock_class_key   s_lock_key;
        struct lock_class_key   s_umount_key;
        struct lock_class_key   i_lock_key;
        struct lock_class_key   i_mutex_key;
        struct lock_class_key   i_mutex_dir_key;
        struct lock_class_key   i_alloc_sem_key;
};
```

Data Structures Associated with Filesystems

- get_sb() function is used to populate the file_system_type{} structures.

- There is only one file_system_type per filesystem, regardless of how many instances of the filesystem are mounted on the system, or whether the filesystem is even mounted at all.

struct vfsmount

- Represents the mount point of the filesystem.

```
struct vfsmount {
        struct list_head   mnt_hash;        /* hash table list */
        struct vfsmount    *mnt_parent;     /* parent filesystem */
        struct dentry      *mnt_mountpoint; /* dentry of this mount point */
        struct dentry      *mnt_root;       /* dentry of root of this fs */
        struct super_block *mnt_sb;         /* superblock of this filesystem */
        struct list_head   mnt_mounts;      /* list of children */
        struct list_head   mnt_child;       /* list of children */
        int                mnt_flags;       /* mount flags */
        char               *mnt_devname;    /* device file name */
        struct list_head   mnt_list;        /* list of descriptors */
        struct list_head   mnt_expire;      /* entry in expiry list */
        struct list_head   mnt_share;       /* entry in shared mounts list */
        struct list_head   mnt_slave_list;  /* list of slave mounts */
        struct list_head   mnt_slave;       /* entry in slave list */
        struct vfsmount    *mnt_master;     /* slave's master */
        struct mnt_namespace *mnt_namespace; /* associated namespace */
        int                mnt_id;          /* mount identifier */
        int                mnt_group_id;    /* peer group identifier */
        atomic_t           mnt_count;       /* usage count */
        int                mnt_expiry_mark; /* is marked for expiration */
        int                mnt_pinned;      /* pinned count */
        int                mnt_ghosts;      /* ghosts count */
        atomic_t           __mnt_writers;   /* writers count */
};
```

Vfsmount mount flags

.

Flag	Description
`MNT_NOSUID`	Forbids `setuid` and `setgid` flags on binaries on this filesystem
`MNT_NODEV`	Forbids access to device files on this filesystem
`MNT_NOEXEC`	Forbids execution of binaries on this filesystem

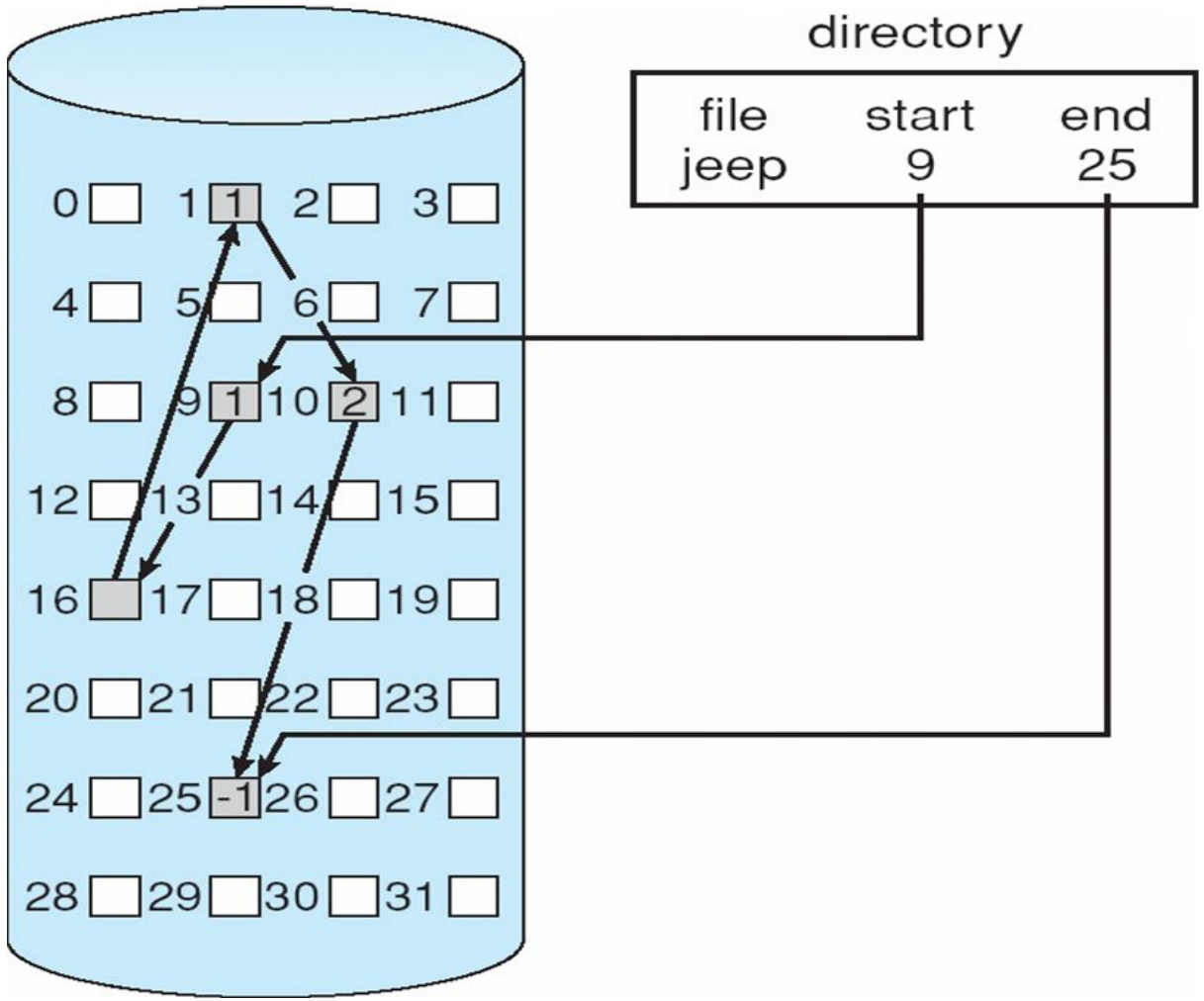
Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

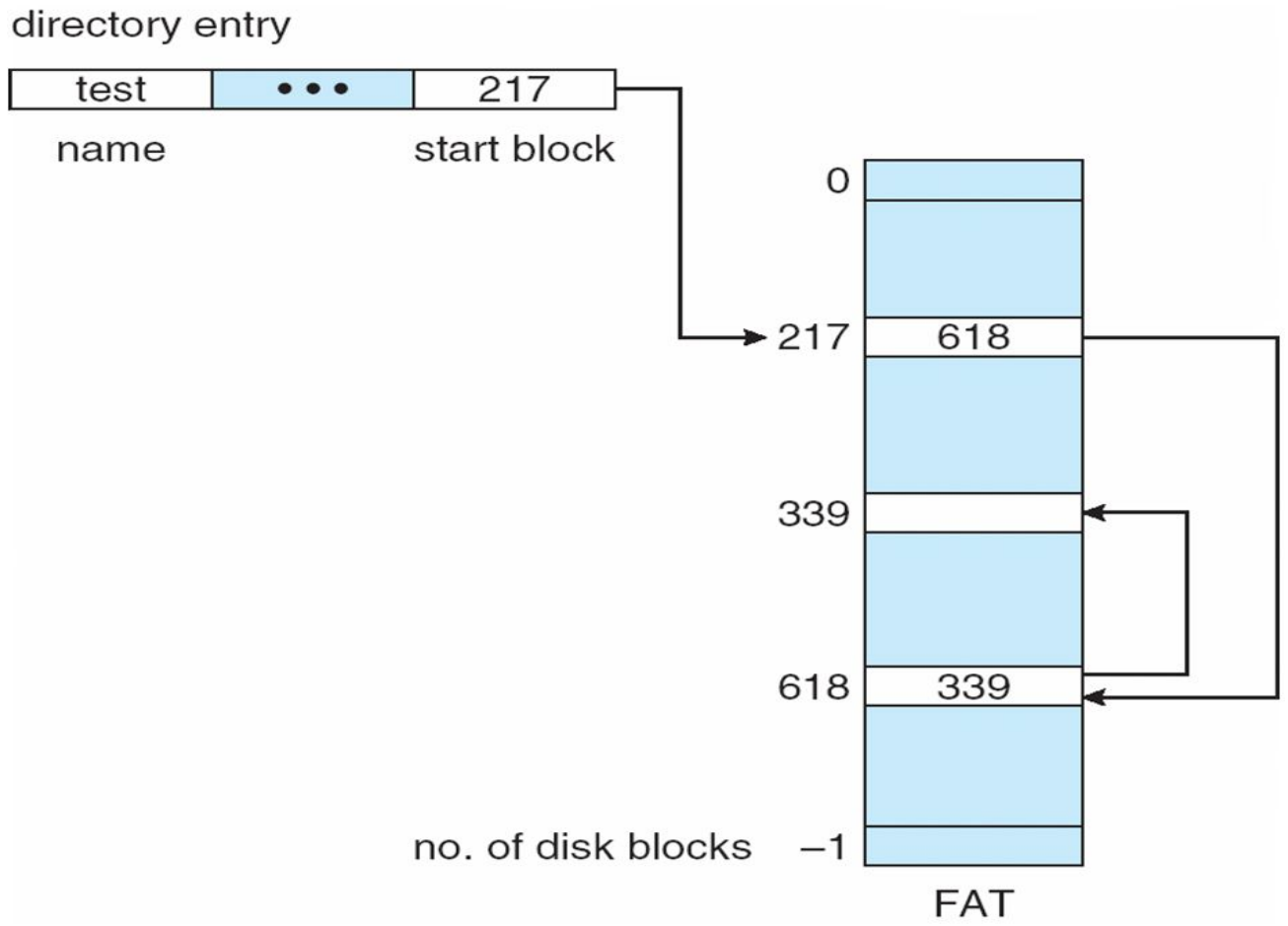
Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks
- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

Linked Allocation

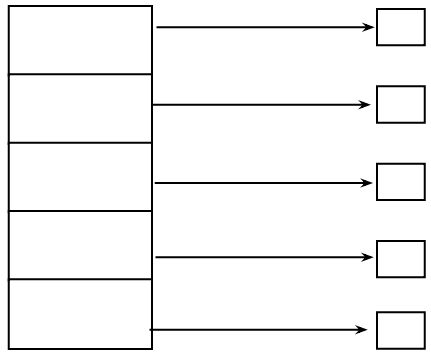


File-Allocation Table



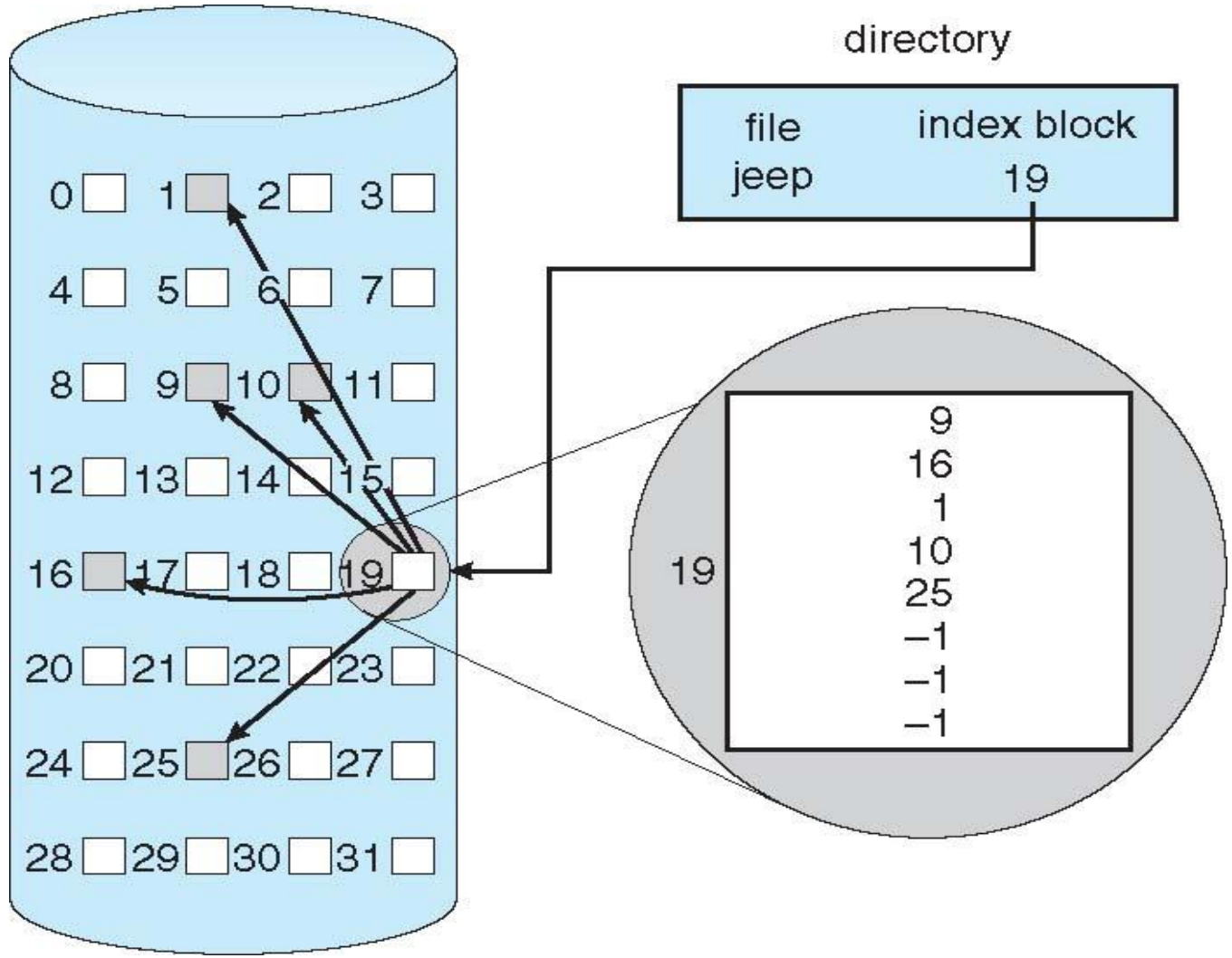
Allocation Methods - Indexed

- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks
- Logical view



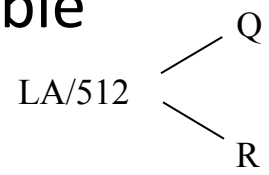
index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

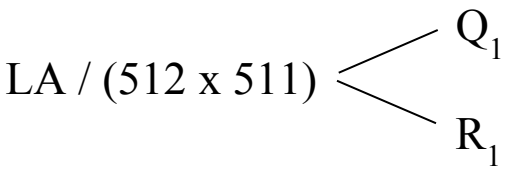
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table



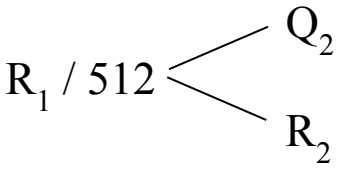
Q = displacement into index table
R = displacement into block

Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- Linked scheme – Link blocks of index table (no limit on size)



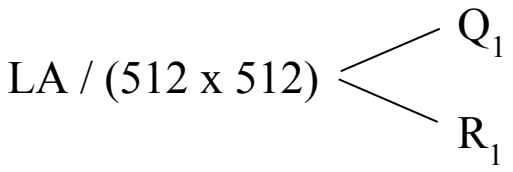
Q_1 = block of index table
R_1 is used as follows:



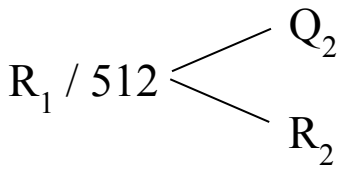
Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)

- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)

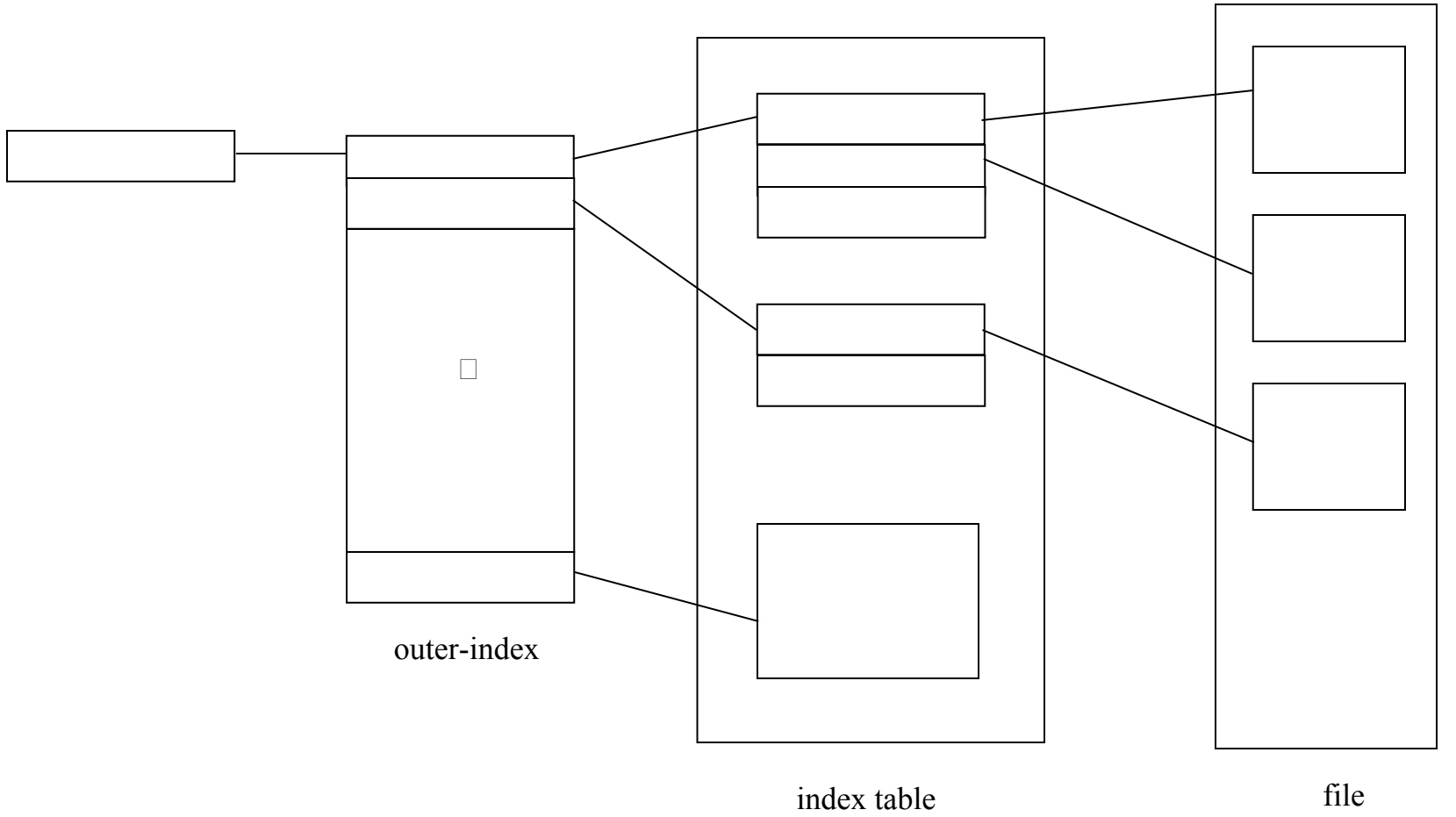


Q_1 = displacement into outer-index
R_1 is used as follows:

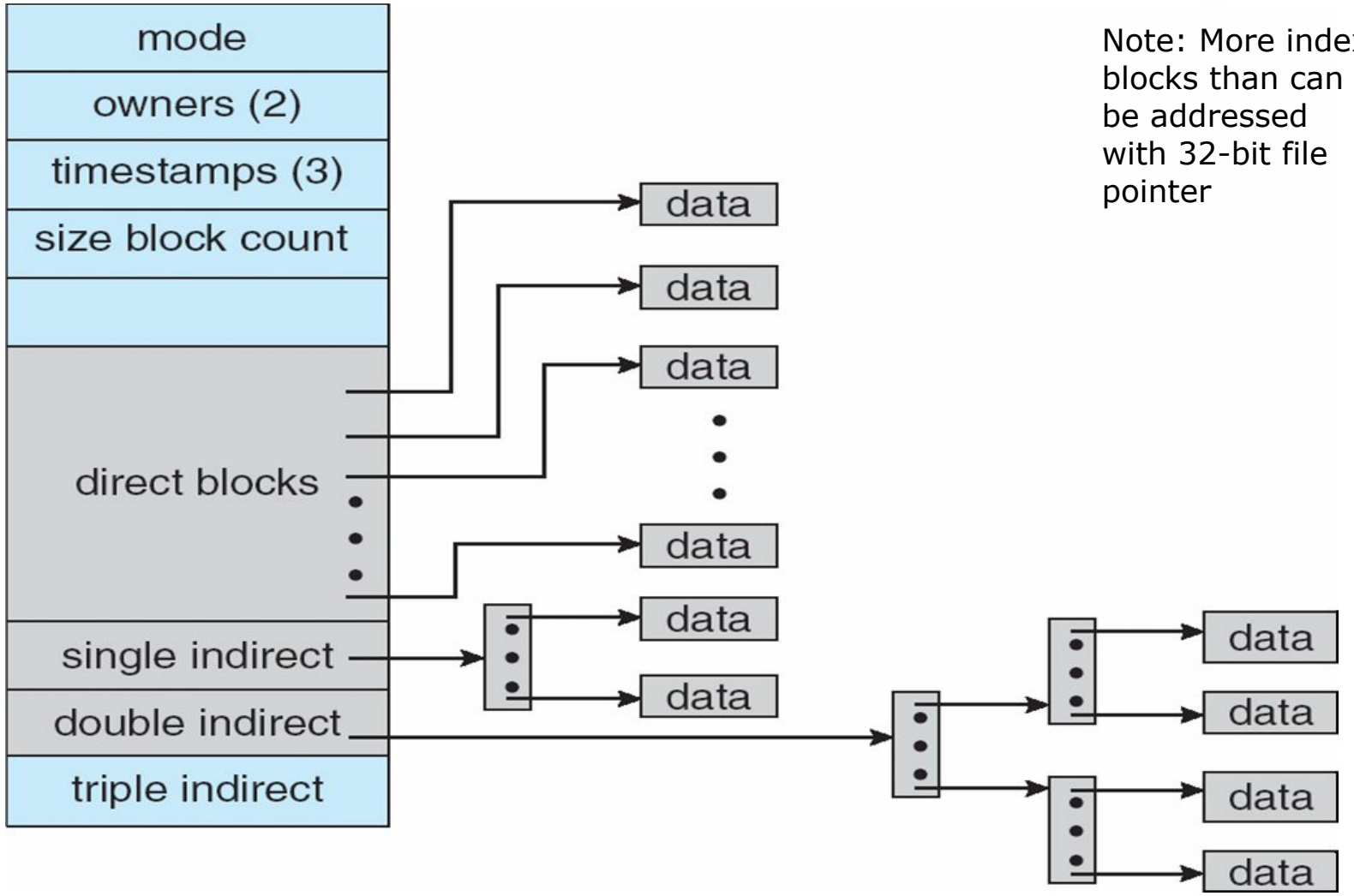


Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation – Mapping (Cont.)



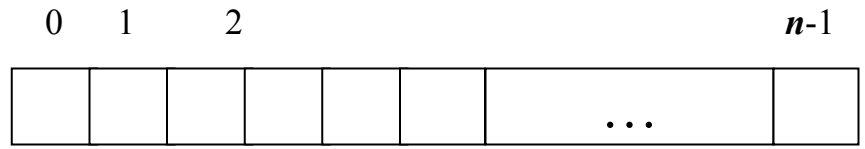
Combined Scheme: UNIX UFS
(4K bytes per block, 32-bit addresses)



Note: More index blocks than can be addressed with 32-bit file pointer

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term "block" for simplicity)
- **Bit vector** or **bit map** (***n*** blocks)



bit[*i*] = 1 ⇒ block[*i*] free

0 ⇒ block[*i*] occupied

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

CPUs have instructions to return offset within word of first "1" bit

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

    block size = 4KB = 2^{12} bytes

    disk size = 2^{40} bytes (1 terabyte)

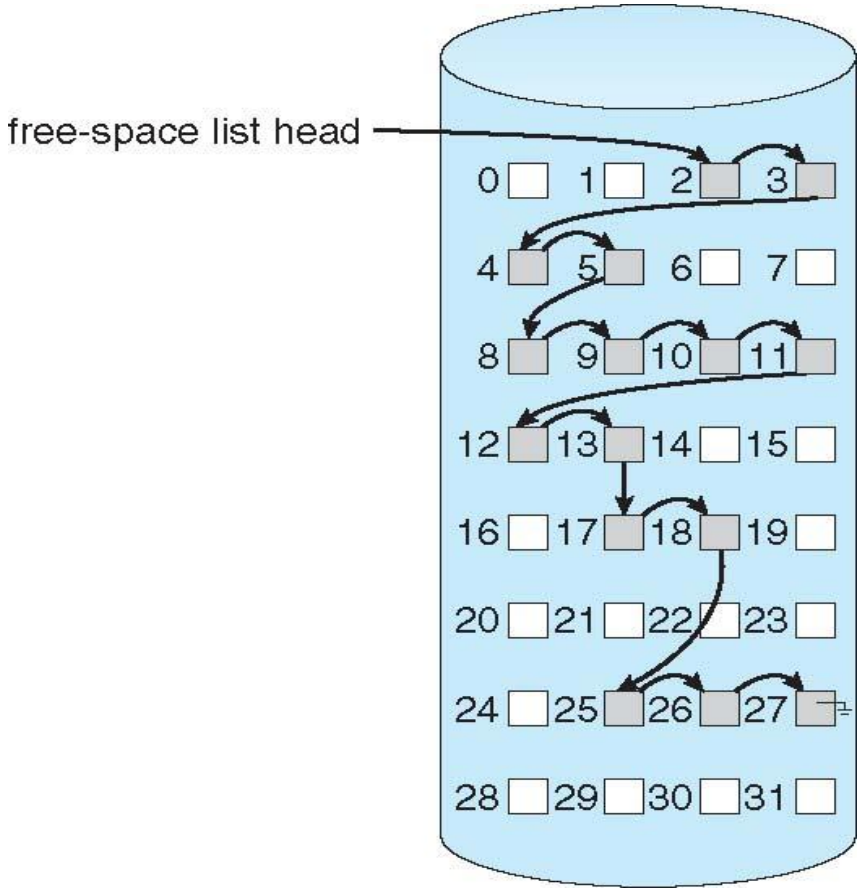
    $\boldsymbol{n} = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)



Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)

- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet

NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

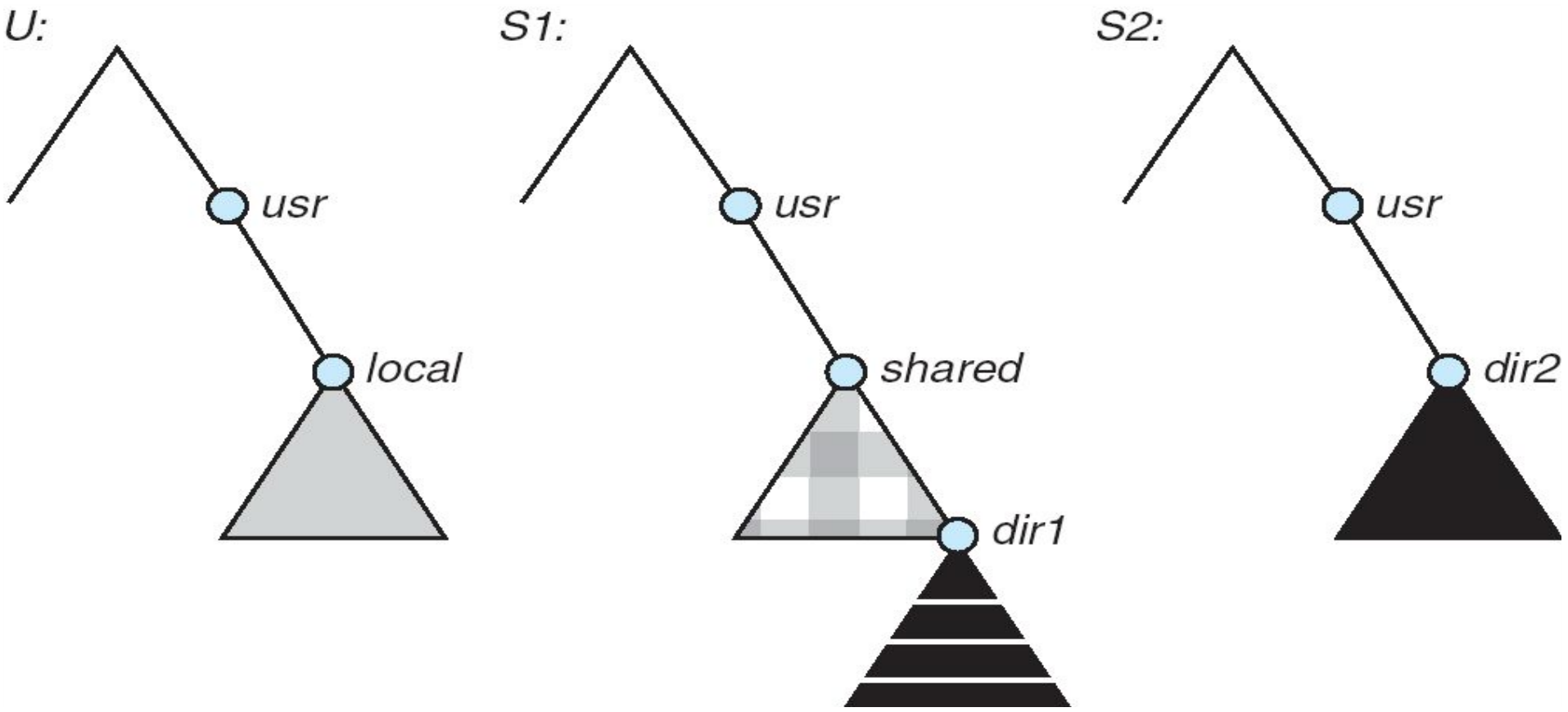
NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media

- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces

- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

Three Independent File Systems



NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is just coming available – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)

- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests

- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture