Operating Systems

FileSystem

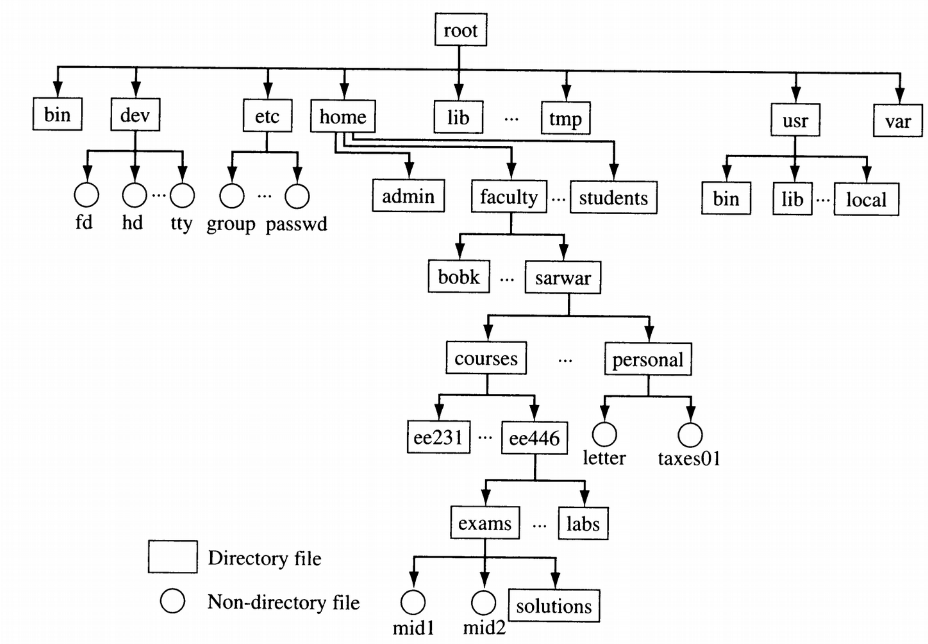
Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

File System

File system resides on secondary storage (disks)

- Provided user interface to storage, mapping logical to physical
- Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily



Concepts:

- File: stored data chunks
- Directory: collection of files or other directories
- Mount Point: directory that holds the file system of a device
- Link: pointer to a file in the file system
  - logical: deleting link does not affect original file
  - physical: when number of links is 0, file deleted

File Systems

"Root" file system (/) usually has several folders and some of them store additional file systems

- other disks: e.g. a pen drive
- file images: e.g. the ubuntu image
- remote directories: e.g. remote home
- logical file systems, populated by the OS, do not necessarily correspond to physical bytes on the disk
  - `/proc`: information about running processes
  - `/sys`: information about system (cpu/buses)
  - `/dev`: direct view of phyical devices

Command line control through the **mount/umount** shell programs:

- `mount -t <type> <device> <mount point>`
- `umount <device>|<dir>`

Mounting can be done automatically by a daemon listening on the USB to add removable devices to your file system

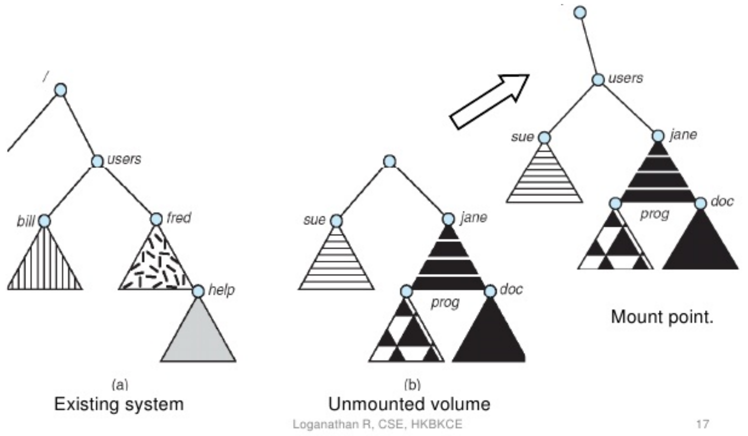
- `mount -t <type> <device> <dir>`
- `umount <device>|<dir>`

Mount

Maps a "volume" with its file system on a folder of the current file system.

Mounting volumes might require privileges (e.g. read write on the raw device)

Unmounting does the opposite



Command line control through the **mount/umount** shell programs:

- `mount -t <type> <device> <mount point>`
- `umount <device>|<dir>`

Mounting can be done automatically by a daemon listening on the USB to add removable devices to your file system

- `mount -t <type> <device> <dir>`
- `umount <device>|<dir>`

File

A "regular" file is

- a permanent storage with contiguous address space
- Contents definded by the creator.
  - In some systems also by the name extension (.exe, .txt)
  - On *nix the extension is optional, but some user programs still rely on it.

Identifying the file type

The shell command

- `file <filename>`

reads the first bytes of a file trying to identify its type

File Attributes

- For the user:
  - name (string)
  - identifiler
  - directory
  - size
  - permissions
  - ownership
  - times
    - creation
    - last modified
  - ...
- For the system
  - identifier (unique in system)
  - location (where on device/fs)
  - ...

Ownership and Permission

A file belongs to an owner and to a group.

Users belong to one or more of these classes:

- user
- group
- others (if none of the above apply)

Each class can be granted granted the following

Permissions:

- read,
- write,
- execute (in case of directories, execute means listing)

Permissions can be changed either by the user ot by root

changing permissions

```
chmod <ugo>[+|-]<rwx> <file>
```

Owenership can be changed by root

changing owner

```
chown <user> <group> <file>
```

user can read and write

group can read and write

others can read

Example on my shell

```
giorgio@frisbi2:~/teaching/sistemi_operativi/slides_v2/odp$ ls -l
-rw-rw-r-- 1 giorgio giorgio 1216426 Sep 19 16:39 os_09_cpu_scheduling.odp
```

user: giorgio

group giorgio

Directory

Is a file of file entries, than can be:

- hard links: are pointers to a file. They are equivalent to each other. deleting all hard links to a file, deletes the file.
- soft links: are soft pointers, deleting all symbolic links does not delete a file.
- other directories.

can host another file system if it is "mounted" in the directory

- In this case the files in the directory are not accessible until the hosted filesystem is "unmounted"
- the directory will show the "root" of the mounted file system

Commands:

- listing:
  - `ls`: lists a directory
- creating a link
  - `ln -s <old_name> <new>`

  if -s is omitted, a hard link is created
- removing a file/link
  - `rm <filename>`

  use `-rf` option to remove recursively all
- removing a directory
  - `rmdir <dirname>`

Consequences of the links:

- The structure in a Unix file system is a generic graph.
- Sometimes the OS prevents loops

File Operations

File is an abstract data type (class)

Operations:

- create
- delete
- truncate
- open
- close
- seek
- read
- write
- wait for a change (select)

`<stdio.h>`

- a FILE is an opaque pointer
- offers a high level interface to the files
  - `fopen`
  - `fclose`
  - `fprintf/fscanf`
  - `fread/fwrite`
  - `fseek`

These operations are mapped to syscalls of the specific OS, and are implemented to "behave" coherently among different OSes

File Descriptor, Opening

Descriptor: integer that characterizes a file within a process

Obtained by returned upon opening/creating the file.

The same file can be opened multiple times. Each open action returns a different descriptor.

Three descriptors are "allocated" for you

- 0: `stdout`
- 1: `stdin`
- 2: `stderr`

All actions on a file are done through its descriptor.

```
//flags: bitmap specifying how to open
//mode: read/write/both
int open(const char *pathname,
         int flags,
         mode_t mode);

//mode: read/write/both
int creat(const char *pathname,
          mode_t mode);

int close(int fd);

//else clone descriptors, no effect on file system
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);
```

Reading/Writing

- Since a file is a sequence of bytes it can be seen as an array that can be written in "chunks"
- Each file descriptor is assiciated to a "pointer" to a file position inside the OS
- read/write operations occur at the current pointer position, and side effect on it
  - reading n bytes from location x will move the pointer to n+x
- lseek moves the pointer inside the file
- Process can also be paused waiting for a file to change through the `select()`. Analogous to sockets.
- Useful for interactive input.

```
ssize_t read(int fd,
             void *buf,
             size_t count);

ssize_t write(int fd,
              const void *buf, size_t count);

off_t lseek(int fd,
            off_t offset,
            int whence);

int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
    struct timeval *timeout);

//Manipulating sets of file descriptors
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

stats, dir, link, mount

Also file attributes (size, user etc) can be read/set (`fstat`)

Directory can be read in blocks

Links can be created - removed. Deletion of a file done by unlinking.

Special functions to read the directories.

Mount might take lots of options (`void* blob`).

```
int stat(const char *pathname, struct stat *buf);

int fstat(int fd, struct stat *buf);

int getdents(unsigned int fd,
             dtruct linux_dirent *dirp,
             unsigned int count);

int getdents64(unsigned int fd,
               struct linux_dirent64 *dirp,
               unsigned int count);

int link(const char *oldpath,
         const char *newpath);

int unlink(const char *path);

int mount(const char *source,
          const char *target,
          const char *filesystemtype,
          unsigned long mountflags,
          const void *data);
```

Devices

- The /dev filesystem provides hooks for physical devices in the system
  - `/dev/sda`: disk
  - `/dev/sda1`: 1st partition of 1st disk
  - `/dev/ttyACM0`: FTDI (serial)
  - `/dev/video0`: webcam
- In Unix devices are accessible as files. Device files are seen as two types:
  - blocks
  - characters

Block devices support seek, and bock reads

Character devices are streams

Example:

Writing something on the serial

- open `/dev/ttyACM0`
- write on the `fd`

Can also copy a file on the serial using the shell:

```
$> cat file > /dev/ttyACM0
```

Or viceversa, read from serial:

```
$> cat /dev/ttyACM0 > file
```

ioctl and termios

- The file interface is a unified way unix allow to access devices
- All is a file
- Special functions of a device can be accessed through `ioctl()` system call
- `ioctl()` is a hook to the driver, and works similar to a "nested" syscall. It takes a "Request" parameter that is an `int` specifying the action to be taken
- Each driver installs its own "request" and "request handler" when loaded.
- character devices can be configured through the unified `termios` interface
- can disable the echo, configure baud rate etc.
- lots of documentation. mostly unreadable.

mmap

We can map a file to memory for faster accesses.

When mmapping a file, the caches are handled transparently, and the file loading/write back is handled by the memory manager

When reading or writing back a "page" the correct driver is called.

Abstraction provided by the OS.

`mmap` can be used to read also special devices (e.g. a camera), without going through the disk.

```
void *mmap(void *addr,
           size_t length,
           int prot, int flags,
          int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Again, lots of options. Look at the documentation and at the examples.

Everything is a File

In Unix, everything is a file

- devices
- disk files

When something is not a file it has a file-like interface

- descriptors
- opening
- closing
- referencing

e.g. shared memory, semaphores etc.

This offers an elegant and coherent interface to the user

At hardware level, things are obviously different

- disks
  - scsi, ide, sata
- buses
  - PCIe, usb
- ports
  - serial, parallel
- graphic cards
- network cards
- other devices
  - keyboards, mouses, joystick cameras...

Physical Devices

- The operating system should cope with the specific devices
- Each device is different
- The implementation should maximize the coherence of the interface and force the driver to follow a specific interface
- A driver is a "virtual class" that should override standard methods defined by the interface
- storage devices obey the "block interface"
- Sometimes the driver of a specific storage device uses other devices on a system
  - e.g. a flask disk driver, on usb
    - exports a block storage interface
    - uses the usb-hub driver to access the hardware by using the usb protocol

This results in a layered architecture for the drivers.

The block devices in

/dev

should implement the block device interface

- query
- seek
- write block
- read block
- .....

block device			video device	
usb_flash	sata	ide	usb cam	1394

Disks

Storage usually organized in partitions

At the beginning

- partition table: data structure that describes the layout of the disk

The disk is accessed through a disk controller, that understands operations like

- read block xx
- write block yy

The low level part of the disk driver offers the high level part a unified interface to these operations.

DMA is typically used

memory management should ensure some physical contiguous pages to be available for the device/process

Disk:

- scheduling/caching are used by the higher part of the driver, depending on the media characteristics
- at OS level:
  - read block xx
  - change to another process, disk deals with the requests and writes in memory
  - on interrupt: identify the disk, resume the process, and copy the data in process space (optional)

With this interface we can just read/write blocks on the disk (or in a partition).

The files/folders structure should be implemented by adding bookkeeping structures to the bare storage

File System Implementation

- Think to the disk as a large binary file that can be accessed in chunks of 512/4096 bytes (blocks)
- On this file, we have to "map" a filesystem, with its folders, subfolders and so on.
- It’s a task of designing data structures
- Many ways to do that

- The OS should be aware of the type of filesystem in a partition to correctly deal with it
- On a linuxbox, serveral filesystems or disks
  - VFAT
  - ext2/3/4
  - ntfs
  - ...

  may cohexist.
- Mount takes care of that

File System Driver

- Is the component of the OS that deals with the specific filesystem
- exposes a uniform interface
  - open
  - close
  - read
  - ....
- uses the uniform interface of a block device

- Again: abstract interface, virtual class

- Example:
  - a filesystem can live on a file (image)

Data Structures

In Kernel

- For each open file, an instance of a structure: of type "OpenFileInfo" is created.
- For each descriptor opened by a process, appropriate structs "OpenFileRef" are put in a list accessible by browsing the PCB.
- Each OpenFileRef points to the corresponding unique OpenFileInfo, and stores an independent file pointer for handling seek/read/write

On Disk

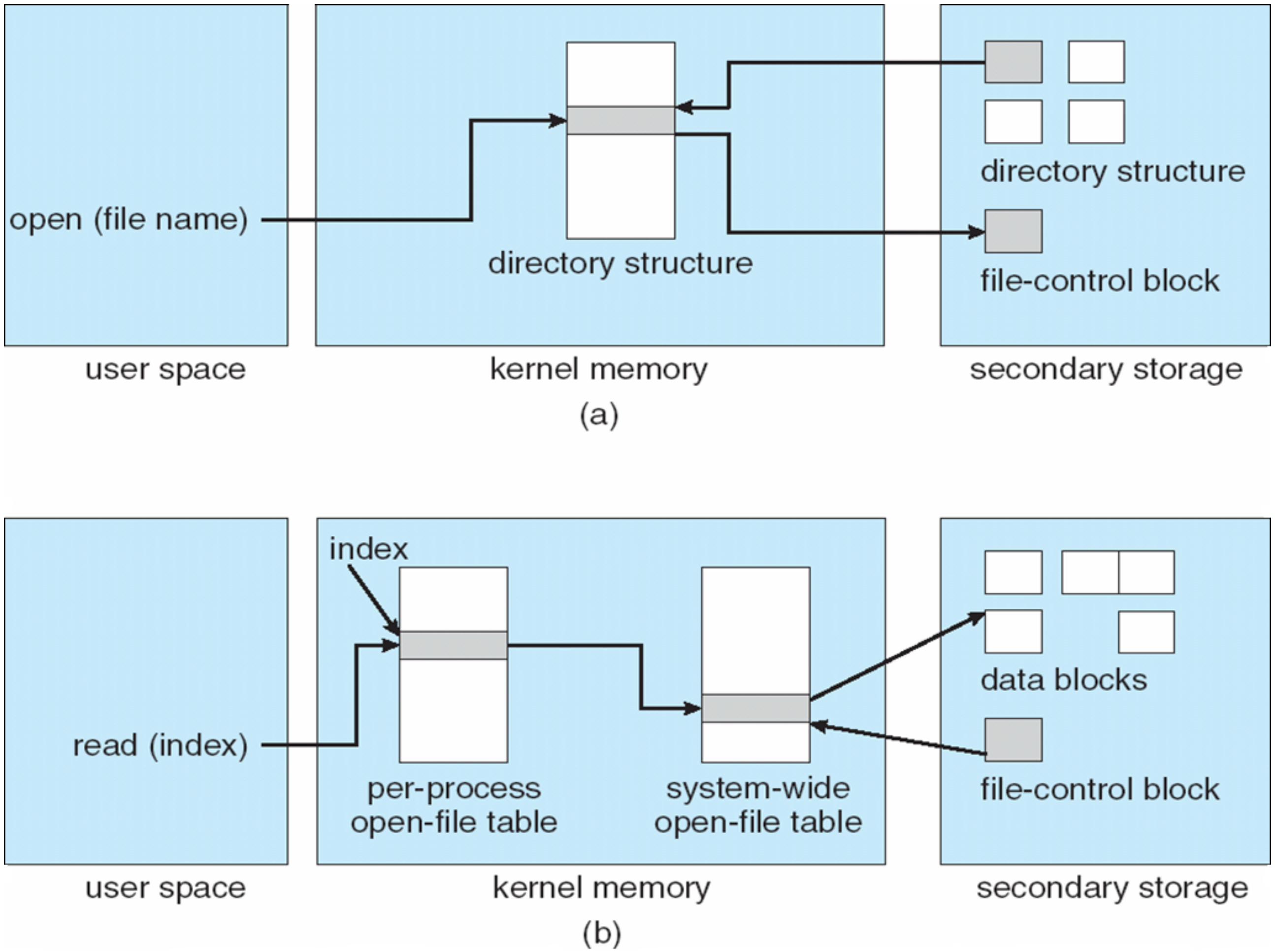
- each file is characterized by a struct (on disk),

  named FileControlBlock (FCB), holding specific file information

- each directory has an header (struct) that extends the file control block
- has a "sequence" of records storing information for each subdirectory/file

Both directories and files can span multiple blocks. This is done using pointers to blocks.

Data Structures



Questions

Implementing a File System

- How to organize a file?
- How to organize a directory?
- How to organize the space on the disk?

Performances

- Continuous write/read
- Random write/read

Disk are mechanical devices,

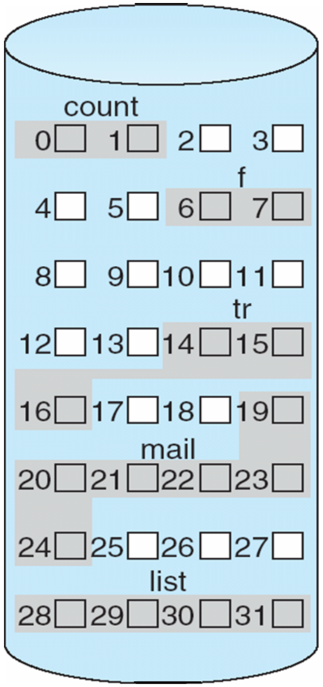
- accessing contiguous blocks is much faster than random blocks

File Allocation: Contiguous

Allocation: How a file is organized in the blocks of the disk?

Contiguous allocation – each file occupies set of contiguous blocks

- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include
  - finding space for file,
  - external fragmentation,
  - need for **compaction off-line** (**downtime**) or **on-line**



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

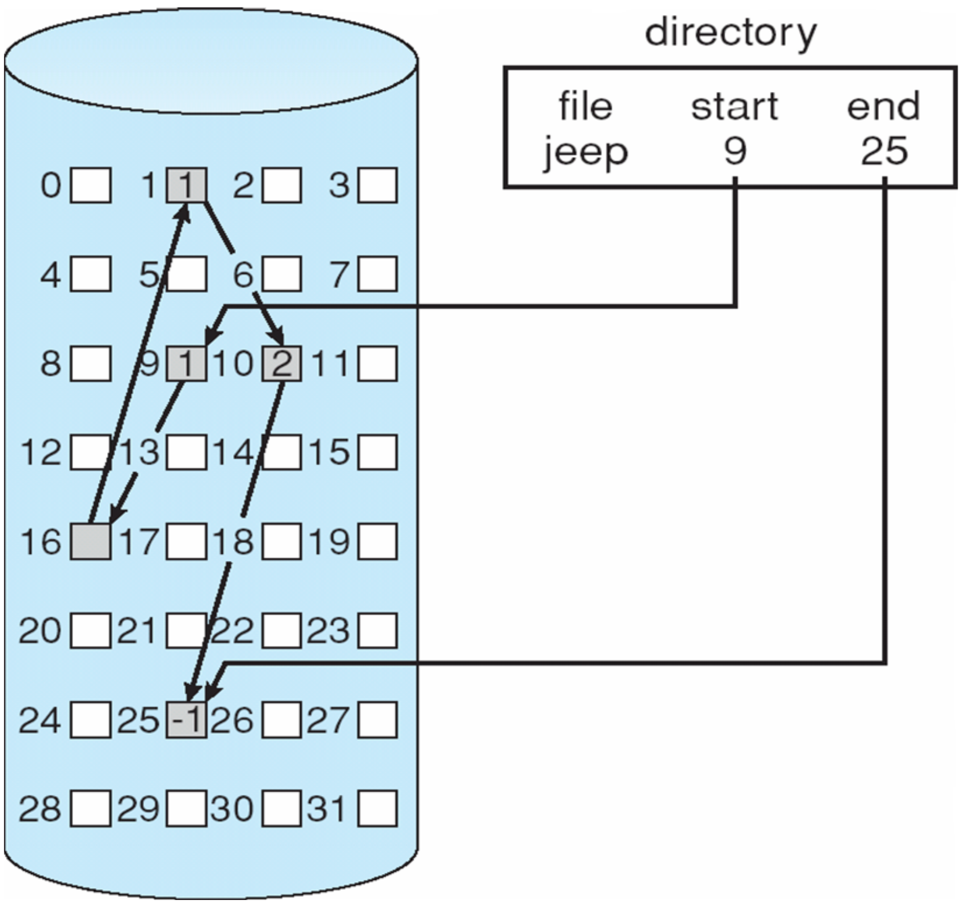
File Allocation: Linked

The (first/last) bytes of a block in the file contain the block index of the next

If care is not taken so that subsequent blocks are contiguous, sequential read is inefficient

- the "head" of the disk moves forth and back.

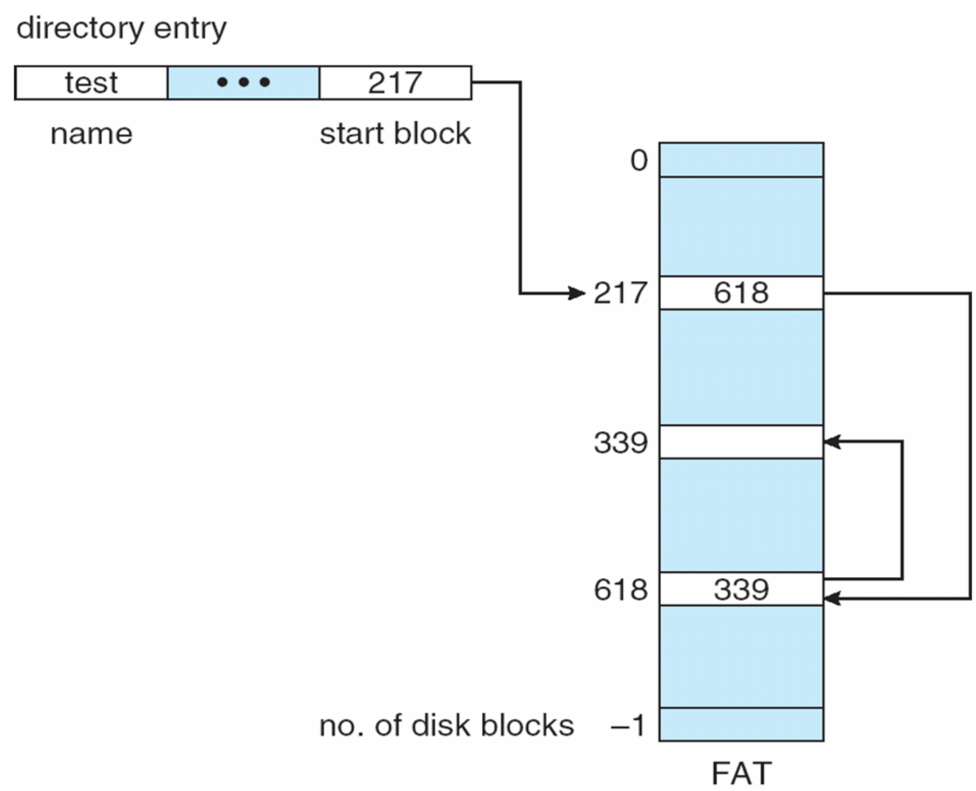
In practice, this is verified and sequential reads are efficient.



File Allocation: FAT

Stands for File Allocation Table

- At the beginning of the disk, put an array (FAT)
- The array is an array_list encoding the sequence of blocks
- The FAT itself is relatively small, and stays in RAM
- Implicitly encodes block structure (invalid blocks in the list)



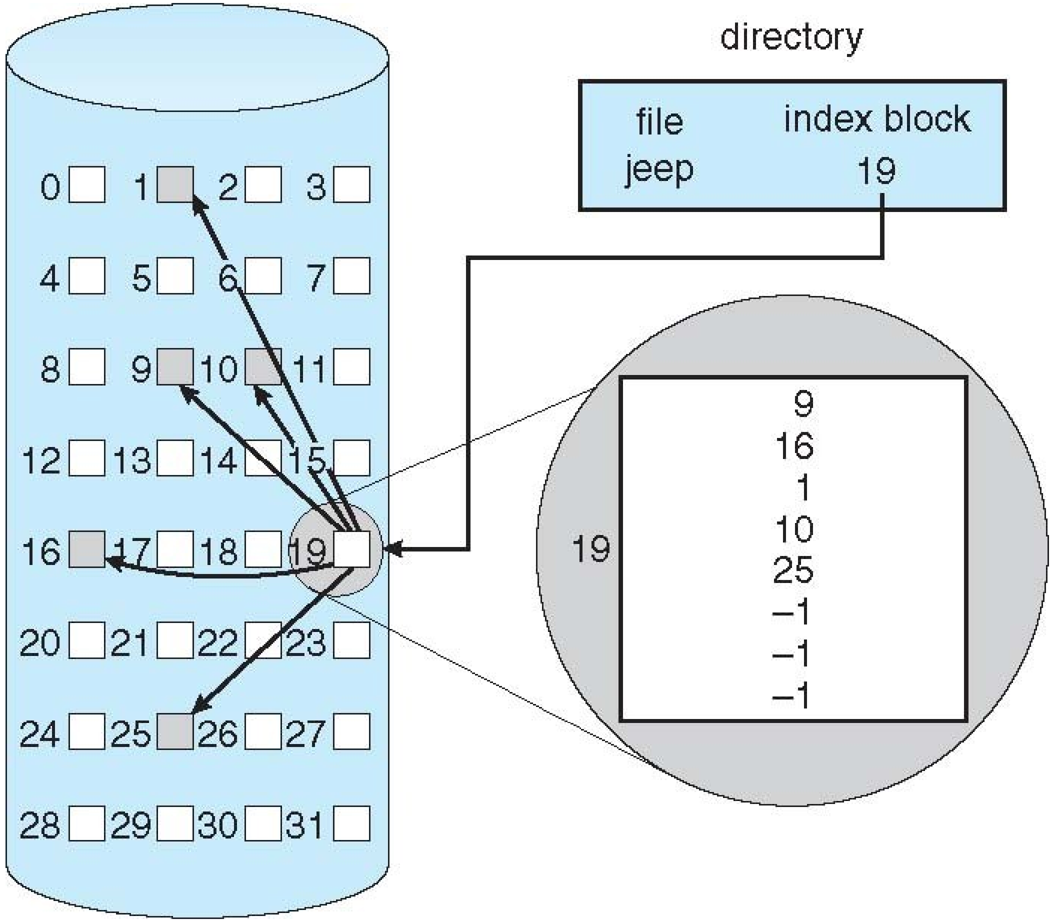
File Allocation: Indexed

- Blocks of the files of two types:
  - index
  - data

Index blocks contain an array of indices of contiguous data blocks.

Index blocks might be organized in a list.

Data blocks contain data.

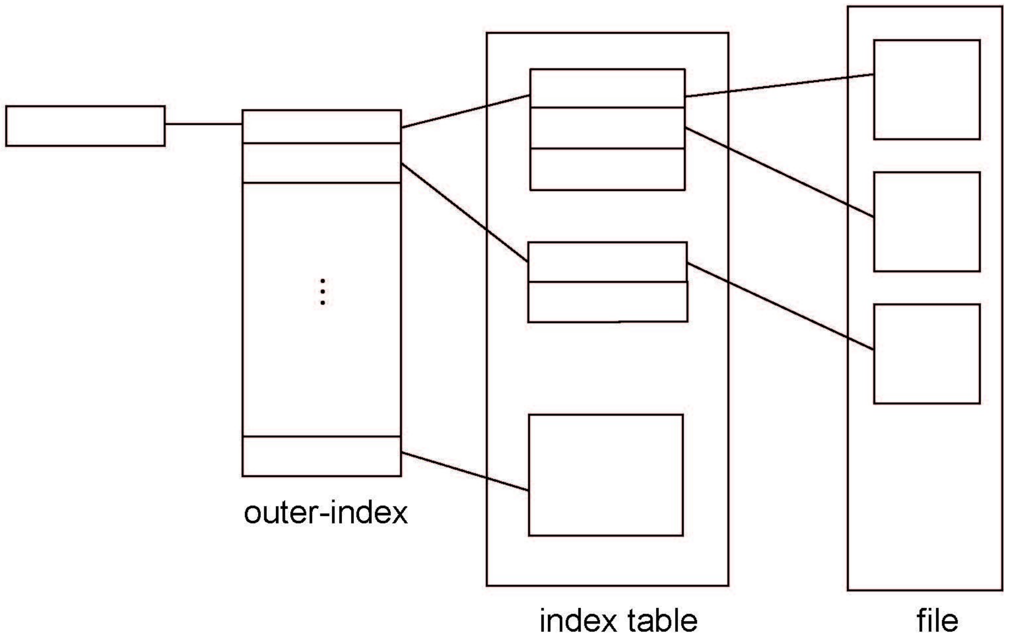


File Allocation: Indexed

- Indexed Allocation

  For very large files, the indices might be in multiple levels

- Level0
  - pointer of index blocks of level 1
  - Level1
    - pointer to data blocks



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

Managing Free Space

Bitmap:

- Keep a bitmap at the beginning of the disk, where each bit corresponds to a block

  When seeking for a free block, use the bitmap to find the closest one
- Slightly slower, but consiers the disk layout

Linked List

- Remember the SLAB allocator?
- link all free blocks on the disk.