Operating Systems

Complutense University of Madrid

2020-2021

Unit 2: File systems

Juan Carlos Sáez

Contents

Contents

What is a file?

Definition

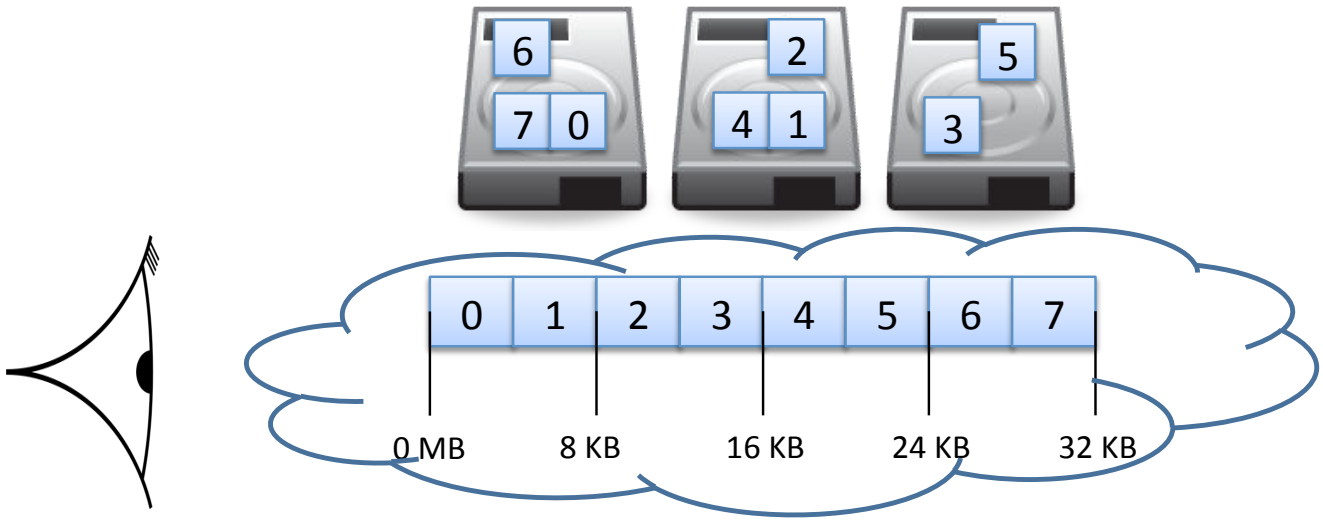
- A file is a non-volatile information unit that stores related data under a well-defined name
- Files are used to:
  - Store the machine code and source code of our programs
  - Feed programs with input data
  - Store information generated by applications

The OS File Management System takes care of ...

1. Offering services to user programs to perform operations on files
2. Handling file permissions
3. Guaranteeing the integrity of file contents and metadata
4. Dealing with low-level issues to support file systems on a wide range of storage devices

Logical view vs. Physical view

- Logical view:
  - Files
  - Directories
  - File systems and partitions
- Physical view:
  - Blocks/Bytes scattered across one or several storage devices



Physical view (top) and logical view (bottom) of a file.

Features of files and file systems

- Persistent information storage
  - Files and their content do not disappear upon system reboot
- A file may store related information in a way that makes sense to a specific application
- Logical and structured names
- Not tied to a specific application's life cycle
- User programs may access files via system calls or library functions
- The file abstraction hides the underlying mechanism to store information on physical devices

Files: User view

- A contiguous logical address space used to store data

File classifications

- By kind of stored data:
  - Text files
  - Binary files
- Programs:
  - Source files, scripts
  - Object files, executable files
- Documents (application specific)

File attributes

- **Name**: the only attribute that makes sense to humans
- **File IDs**: file descriptor ID, owner ID, group ID
- **File type (from the OS perspective)**: Required on systems that support different file types (regular and special)
  - For more information: `man 2 stat`
- **File size**: size in bytes, size on disk, max size, ...
- **Protection**: Access control bits (`rwx` for user, group and everyone else)
- **Time information**: creation, last access, last modification, ...
- ...

File types in UNIX

- UNIX and UNIX-like OSes (Linux, Mac OS X, Solaris, ...) expose several OS abstractions to the user as files
  - Not every file that users can see is an *ordinary* file

File types in UNIX

- Regular files (ordinary files)
- Directories
- Character device files
- Block device files
- Symbolic links
- Named pipes (FIFOs)
- ...

File descriptors

- Every object/entity handled by the OS has a name or unique descriptor
- A file features two unique descriptors:
  1. **Logical descriptor**: name of the file
  2. **Physical descriptor**: internal data structure that the OS manipulates to support operations on the file
- Directories enable the OS to associate a file's logical descriptor with its physical descriptor

More on logical descriptors (names)

Naming rules and conventions are OS specific

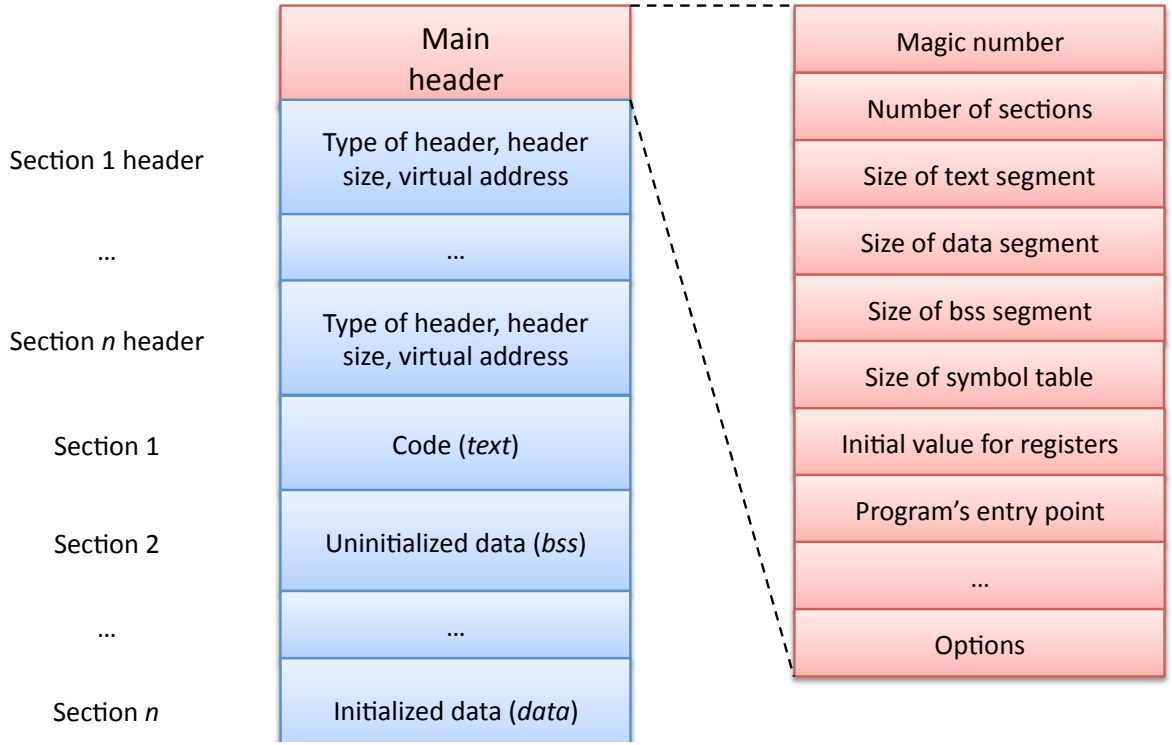
- **Maximum character length**: 8 (MS-DOS), 4096 (UNIX)
- **Case sensitive/insensitive**: `MyFile` and `myfile` represent the same file on Windows but different files on Linux
- **File extension**
  - Mandatory/optional
  - Simple/Multiple
  - Fixed for a particular document type
  - Generally, extensions are only meaningful to applications (pdf, doc, html, c, cpp, ...)

More on physical descriptors

- Upon a file's creation, the OS allocates an internal data structure for the file (physical descriptor)
  - Stored on disk (*persistent*)
  - Cached in memory when the file is being accessed by the OS on behalf of a user program
- Some file systems associate numerical IDs to physical descriptors
  - Used as a pointer to the physical descriptor
  - Example: inode number (UNIX)

Contents of a file for the OS

- From the OS perspective, most files are just a bunch of bytes spread all over one or several storage devices
- The OS recognizes just a few file formats:
  - Executable files
  - Text files (e.g. scripts)



File systems

Low-level access to storage devices is ...

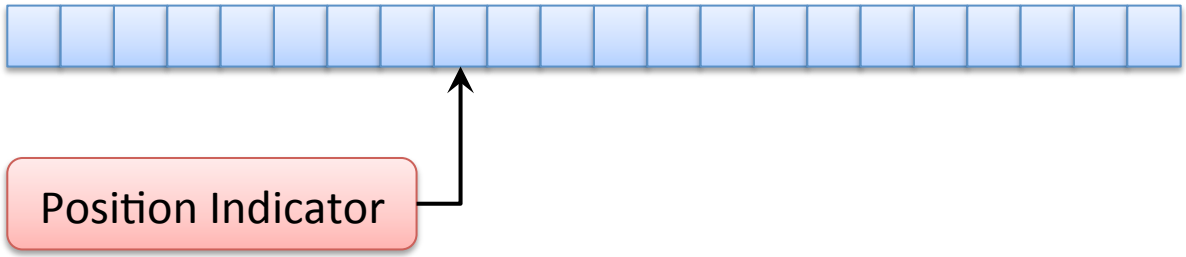
- Complex:
  - Explicit knowledge of physical features of a particular device
  - Accesses in terms of physical addresses
- Unsafe:
  - Users with access to the physical level have no restrictions

The file management system ...

- Exposes a *logical view* of storage devices to users
- Implements a set of convenient operations (create, delete, read, write, ...) that hide the physical details to user programs
- Implements protection mechanisms
- Fast and efficient access to files

Logical and physical view of a file

- User: Logical representation (Byte array)



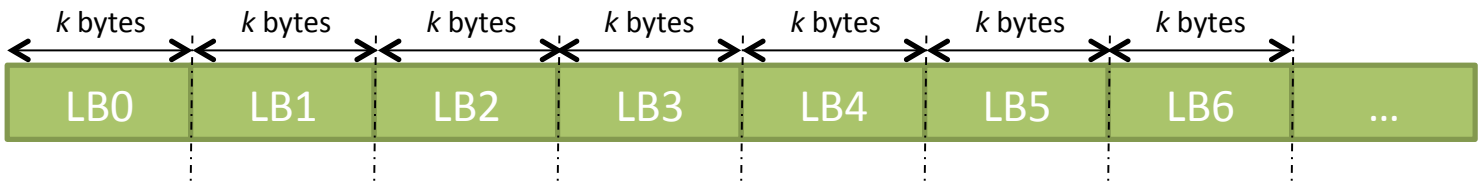
- OS: Physical representation tied to a specific device (Set of blocks)

File A	
Blocks:	13
	20
	1
	8
	3
	16
	19

1	2	**3**	4	5	6
7	**8**	9	10	11	12
13	14	15	**16**	17	18
19	**20**	21	22	23	24
25	26	27	28	29	30

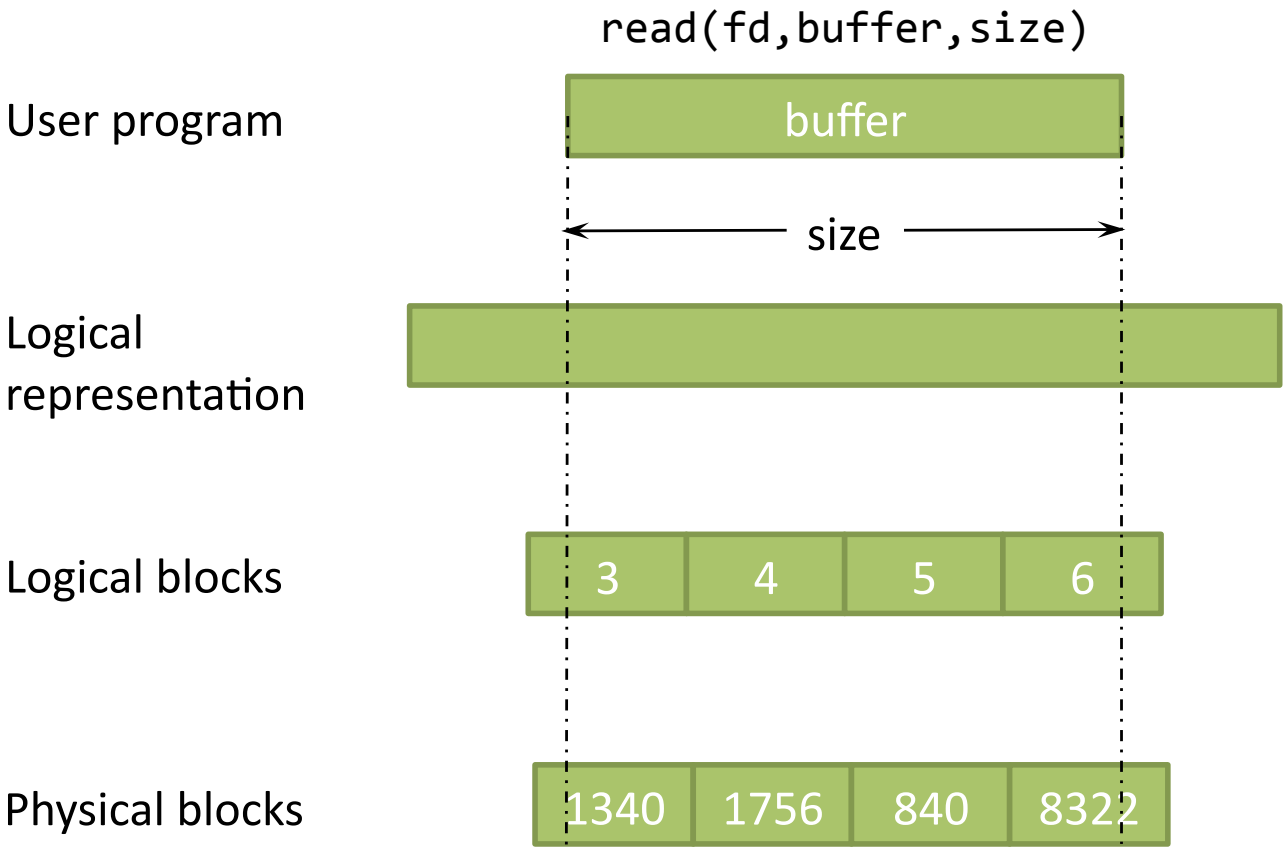
Logical Block vs. Physical Block

- The array of bytes in the logical representation can be divided into *logical blocks*
  - If the block size is k bytes, a logical block is a set of k contiguous bytes in the file



- A physical block represents a disk location where a logical block can be stored

Operations on files



Translation from logical into physical address

- User processes access file data using a *logical address* (in bytes)
  - The *logical address* must be translated into the associated *physical address*
- Translation steps
  1. Logical address → Logical block address = (Num. logical block, Offset)
     - Num. logical block = $\left\lfloor \frac{\text{LogicalAddr(bytes)}}{\text{BlockSize(bytes)}} \right\rfloor$
     - Offset = LogicalAddr(bytes) **mod** BlockSize(bytes)
  2. Logical block address → Physical address = (Num. physical block, Offset)
     - Filesystem-specific data structures are used to maintain the correspondence between logical and physical blocks

Internal organization of the file system

How do we associate the logical blocks of a file with the actual location on disk (physical blocks)?

How do we allocate free blocks for a file when it needs more space?

Data structures (I)

- The OS needs various data structures to manipulate files and file systems

Data structures on disk

- File system must be self-contained, non-volatile and independent from a specific OS
- Metadata must be also stored on disk
  - Data structures to account for free/used blocks
  - Data structures enabling the translation between logical and physical addresses
  - ...

Data structures (II)

- Data must be also in memory when the OS needs to work with it

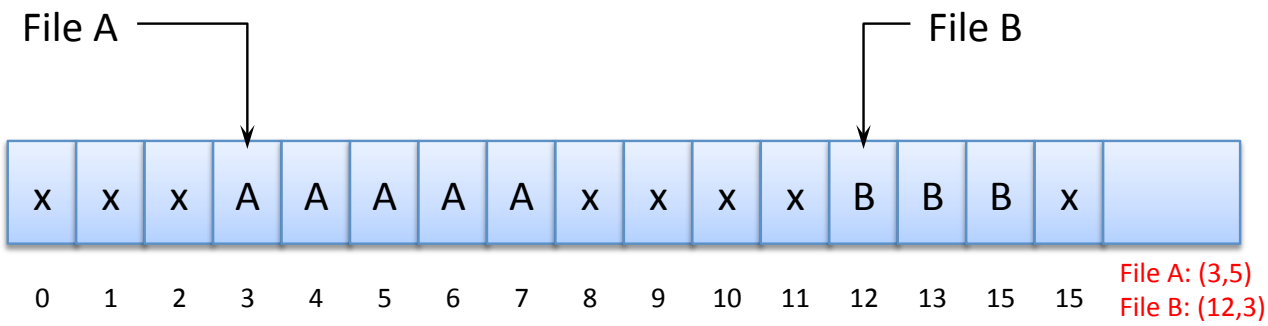
Data structures in memory

- The OS maintains different data structures in memory for efficient manipulation of data and metadata
- Not all data on disk is maintained in memory as well

File allocation schemes

- Contiguous allocation
  - Used in CD-ROMs and tapes
- Linked allocation
  - Example: FAT (File Allocation Table) file systems
- Indexed allocation
  - UNIX-SV, FFF (Fast File System), ext2, etc.
- File allocation based on balanced trees
  - NTFS, JFS, Reiser, XFS, etc.

Contiguous file allocation



CONTIGUOUS ALLOCATION

Advantages

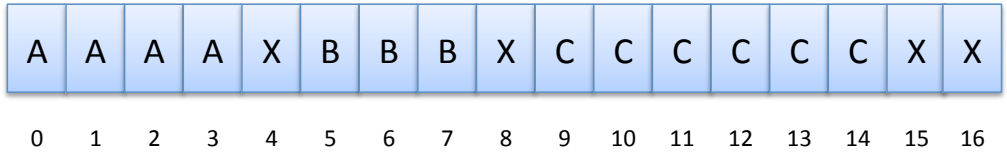
- Optimal sequential accesses
- Enable data prefetching (read)
- Easy implementation of random accesses

Disadvantages

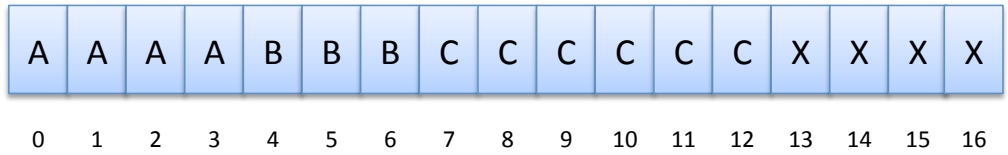
- Resizing files may be costly
- External fragmentation, predeclaring file size
- Require compaction every so often

External fragmentation (example)

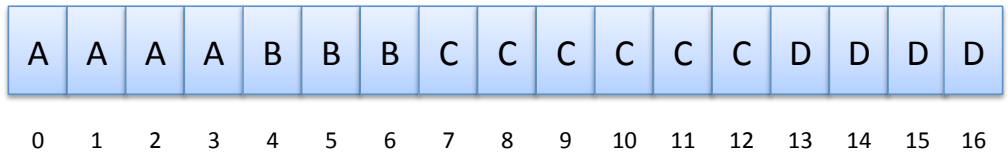
- Create a 4-block file (D) in a FS with contiguous allocation



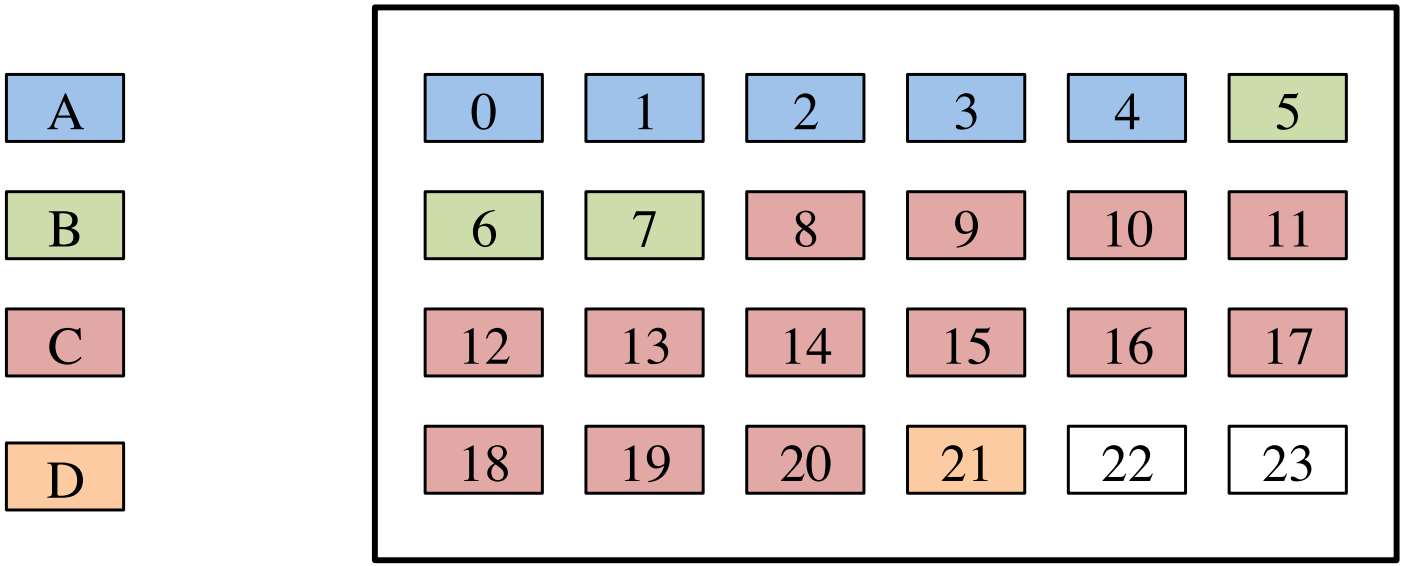
- 3 existing files using 13 out of 17 blocks
  - It is not possible to create the file with the initial organization
- Apply compaction



- Create the file

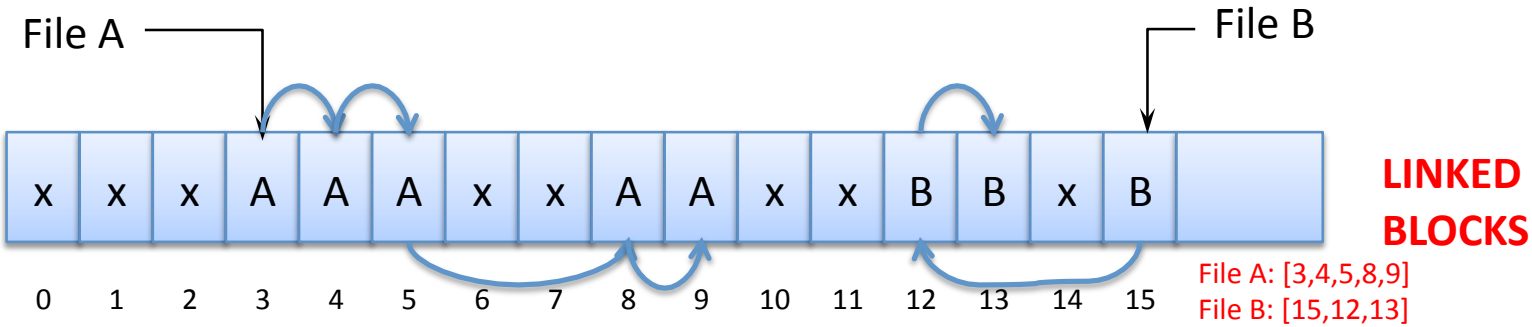


Contiguous allocation: CD-ROM (ISO-9660)



Linked file allocation

- A linked list of blocks is used to keep track of the various blocks in a file
- Two approaches
  - linked list of individual blocks
  - linked list of contiguous blocks



Linked file allocation (II)

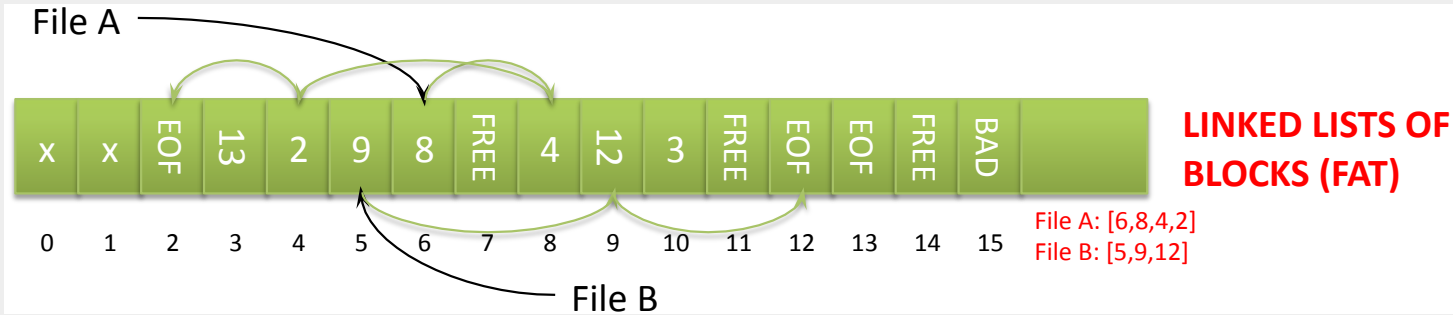
Where are the list pointers stored?

- *Option 1*: Store the pointers at the end of each block
  - Poor choice for random accesses
    - Access logical block $n \rightarrow$ Read n blocks
  - Space for data in a block $\neq 2^k$
- *Option 2*: Devote a separate region on disk to store linked lists
  - Used in FAT file systems

Linked file allocation (III)

File allocation table (FAT)

- Table stored in a special region on disk
  - As many entries as the number of addressable (physical) blocks
- Each *i-th* entry stores one of the following values:
  - Number: Pointer to the next block in a file's linked list of blocks
  - EOF/NULL: Marker to indicate the end of the linked list
  - FREE: The *i-th* block is unused
  - BAD: The *i-th* block is faulty
- Number of initial block of every file stored in the file's physical descriptor (not in the FAT)



Linked file allocation (IV)

Advantages

- Not subject to external fragmentation
- Resizing a file is a rather straightforward task
  - New blocks can be inserted at the end of the linked list
- Easy implementation of sequential accesses

Disadvantages

- Random accesses require traversing the linked list of blocks
- Locality is not taken into account
  - Blocks in a file may be spread all over the disk
- Periodic compaction is advisable

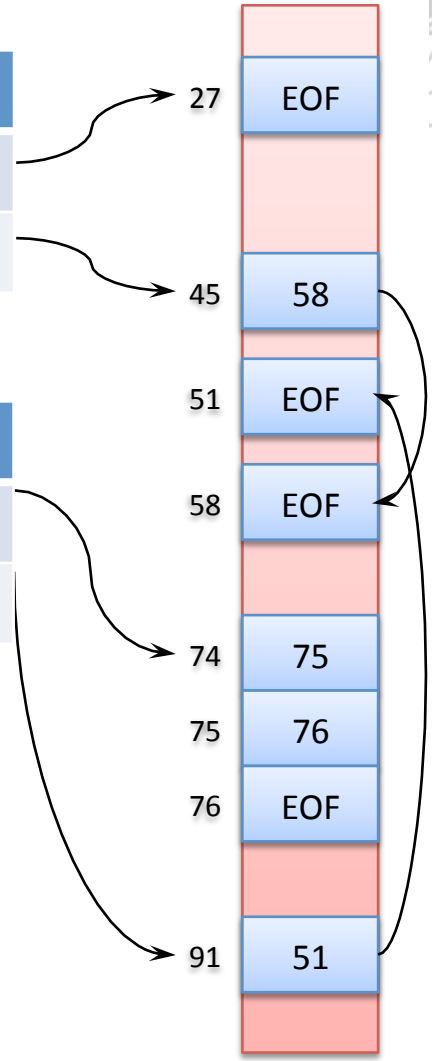
Linked file allocation: MS-DOS (FAT)

Root directory

Name	Attrib.	KB	Cluster
joe_dir	dir	5	27
file1.txt		12	45

Directory "joe_dir"

Name	Attrib.	KB	Cluster
card1.wp	R	24	74
rest.zip		16	91



FAT file systems use clusters rather than blocks.
A cluster is a set of contiguous blocks.
Every file occupies at least one cluster on disk.

Indexed file allocation

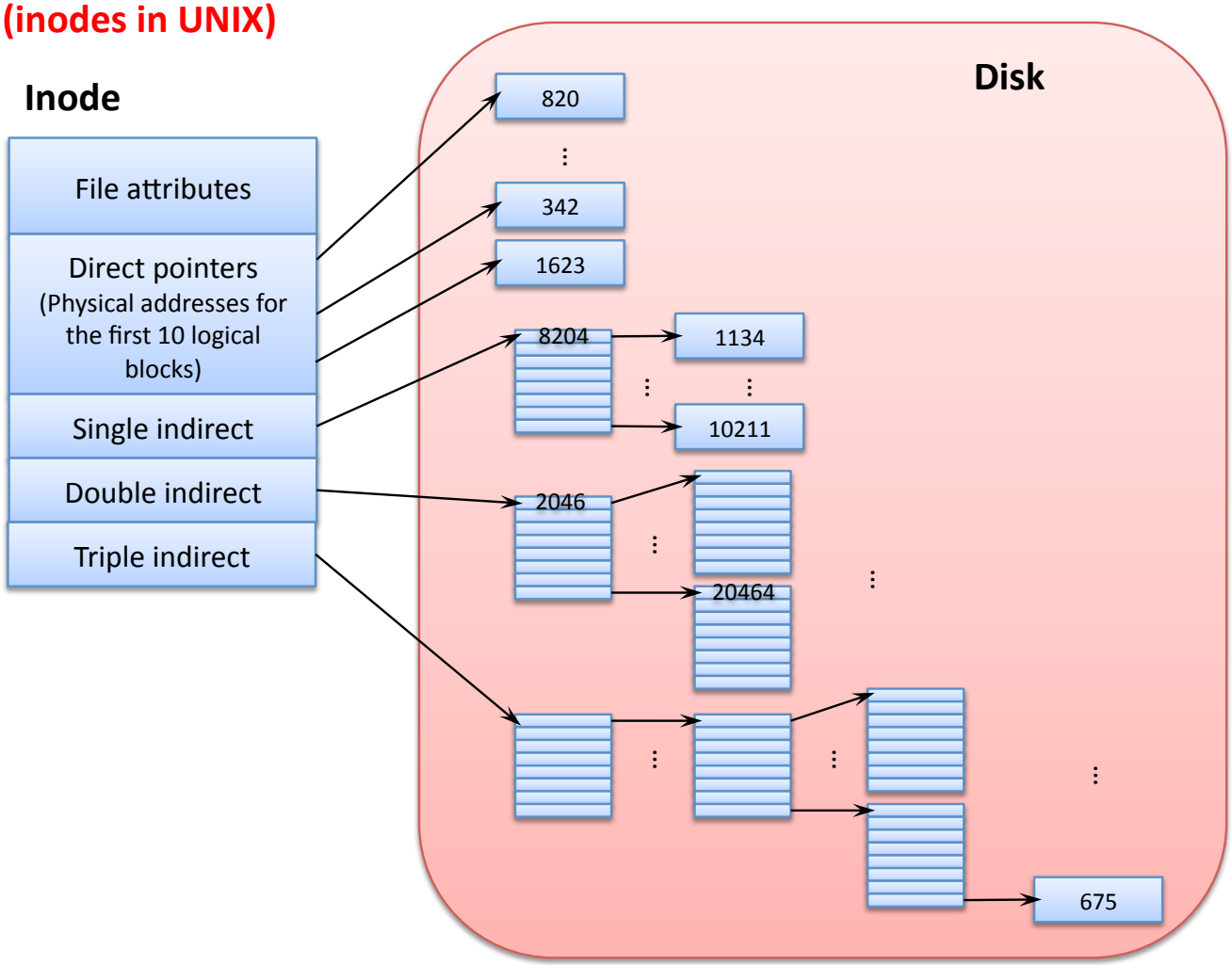
- In the indexed allocation a file's physical descriptor features a set of indexes (pointers) to access data
- Two types of indexes:
  1. **Direct**: points to a *data block*
  2. **Indirect**: points to an *index block*
     - Each index block stores k pointers to data blocks or index blocks

$$k = \left\lfloor \frac{\text{BlockSize}(\text{bytes})}{\text{IndexSize}(\text{bytes})} \right\rfloor$$

     - Different types of index blocks: simple, double, triple, ...

Indexed file allocation: Inodes (UNIX)

INDEXED ALLOCATION
(inodes in UNIX)



Indexed file allocation: Inodes (UNIX)

Advantages

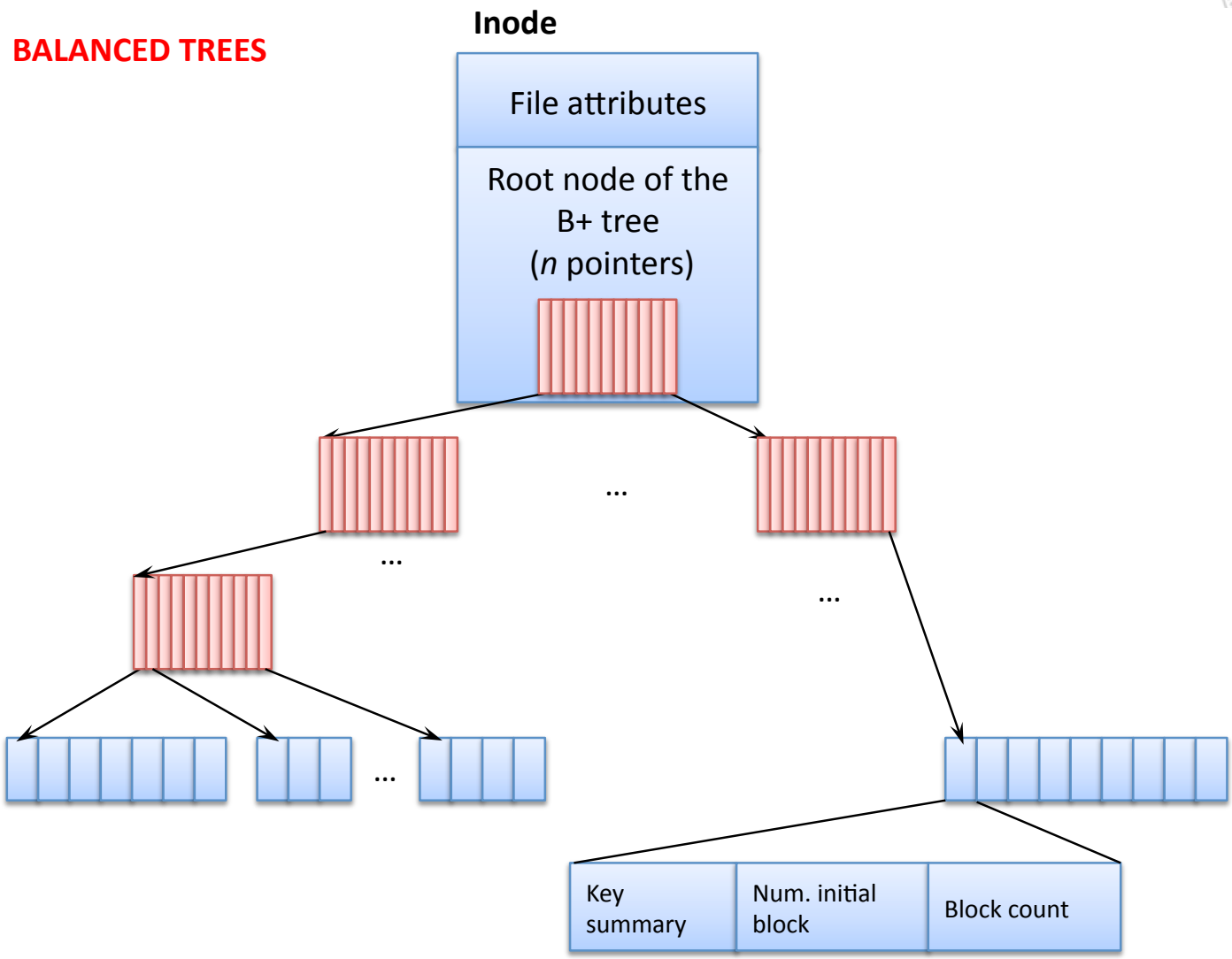
- Efficient access to initial d logical blocks in a file thanks to direct pointers
- Easy implementation of sequential and random accesses
  - There is no need to traverse linked lists of blocks

Disadvantages

- Inode must be in memory to ensure efficient disk accesses
- Random accesses have variable access time
  - Cost depends on the level of indexes associated with the logical block being accessed
- Maximum size of a file limited by indexed organization:
  - BS: block size; d: number of direct indexes; n: index size (bytes)
  - MaxFileSize$=\left(d+(BS/n)+(BS/n)^2+(BS/n)^3\right)\cdot BS$

File allocation based on balanced trees

BALANCED TREES



Managing free space

- The OS needs to know which blocks and physical file descriptors (e.g., inodes) are free
- Available choices:
  1. Bitmap (e.g., 1100000010010101010101001...)
     - Size of bitmap (bytes) = disk_size(bytes) / (8 * block_size(bytes))
     - Example: 16 GB / (8 * 1KB) = 2 MB
  2. Linked list of free blocks
     - Implemented as a stack or queue possibly *cached* in memory
     - Variant: linked list of contiguous free blocks: pairs (initial block, block count)
  3. Indexed allocation
     - Free space modeled as a file with indexed allocation

Contents

Concept of directory

- Object enabling to associate a file name (logical descriptor) with its internal or physical descriptor
  - Typically represented as a table (set of directory entries)
  - In some file systems, such as FAT, each directory entry is a physical descriptor itself
- Directories also make it possible for users to keep their information organized in a structured way
  - A directory is a node in hierarchical file systems

Desirable properties of directories

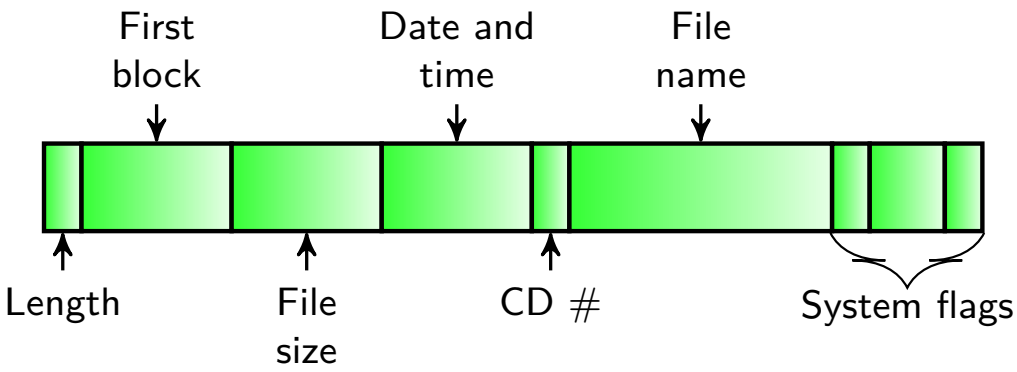
Properties

- **Efficiency**: find files in a directory quickly
- **Simplicity**: each directory entry must be as simple as possible
- **Naming**: Convenient and simple to users
  - Two different users may choose the same name to refer to different files
  - Several names may be used to refer to the same file
- **Organization**: Users may store files in separate directories based on their types (e.g. C programs, games, text documents, ...)
- **Structured view**: The OS must expose a clear API enabling user processes to traverse directories hiding the filesystem-specific implementation

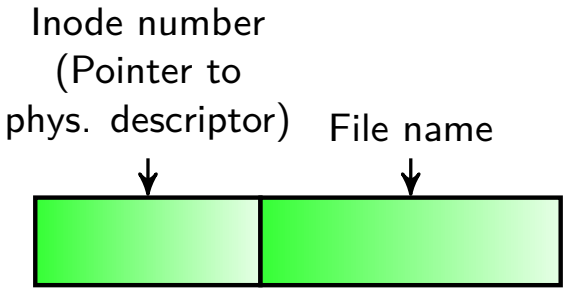
Directory organizations

- Both files and directories are stored on disk
- Directories are special files implemented using a format known by the OS
- Two approaches to implement directories have been used:
  1. The directory entry for a file is also its physical descriptor
     - File attributes are stored in the directory entry
     - Example: CD-ROM, FAT
  2. A directory is merely a table of pairs *(file_name,ID)*
     - ID is a unique number ("pointer") to refer to the file's physical descriptor
     - Example: UNIX file system

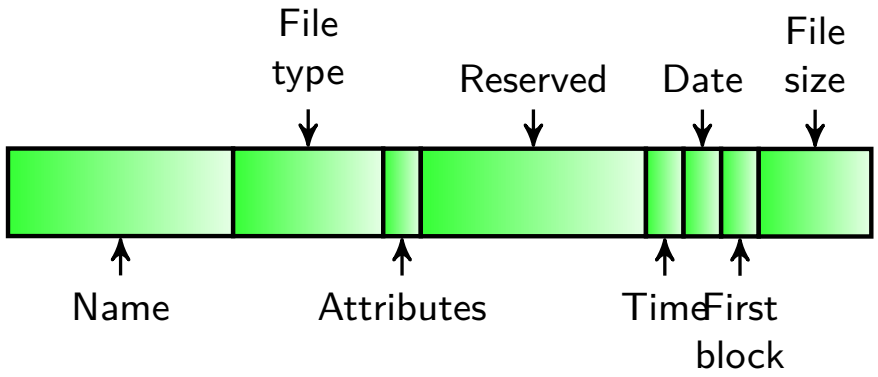
Contents of a directory entry



Directory entry of ISO-9660 (CD-ROM)



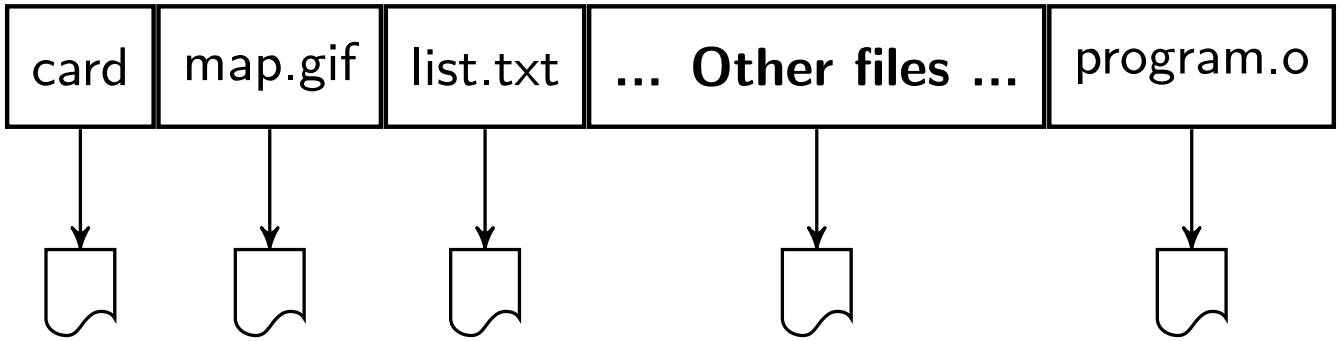
Directory entry de Unix SV



Directory entry of FAT (MS-DOS)

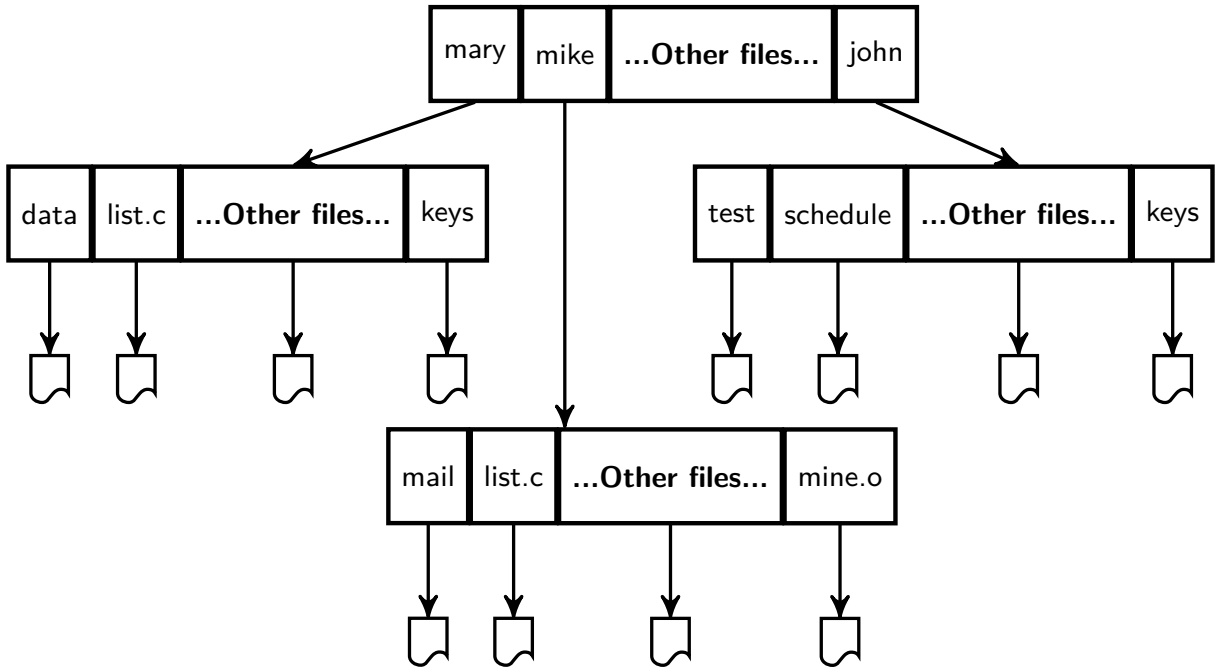
Directory hierarchies: Single level

- A single directory exists for the entire system
  - Shared by all users
- Issues related to naming, organization and efficiency



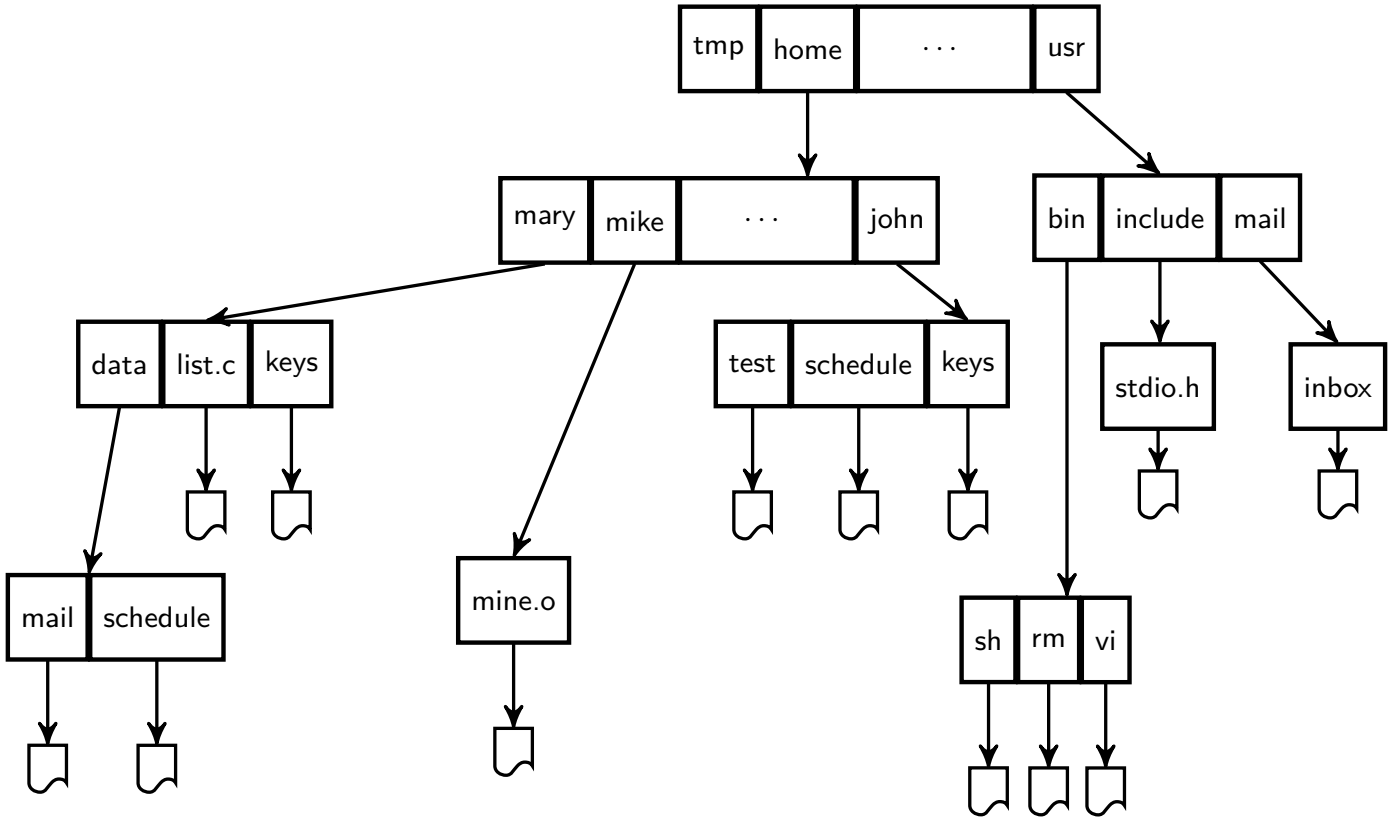
Directory hierarchies: Two levels

- Each user owns a private directory
- File paths can be explicit (*absolute*) or implicit (relative to the user's directory)
- Two different users may choose the same name to refer to different files
- Efficient search but poor organization properties



Directory hierarchies: Tree based

- Efficient search and good organization properties
- More complex paths are needed to refer to files



Paths in tree-based directory hierarchies (I)

Special directories

- **Working directory '.'**
  - Example: `cp /users/mike/keys .`
  - On BASH:
    - It can be changed with `cd`
    - The full path can be obtained with `pwd`
- **Parent directory '..'**
  - Example: `ls ..`
- **Home directory**: The user's *base* directory
  - `cd`
  - `cd $HOME`

OS	Default location of HOME
Windows 7 and later	`C:\Users\<username>`
GNU/Linux	`/home/<username>`
Solaris	`/export/home/<username>`
Mac OS X	`/Users/<username>`

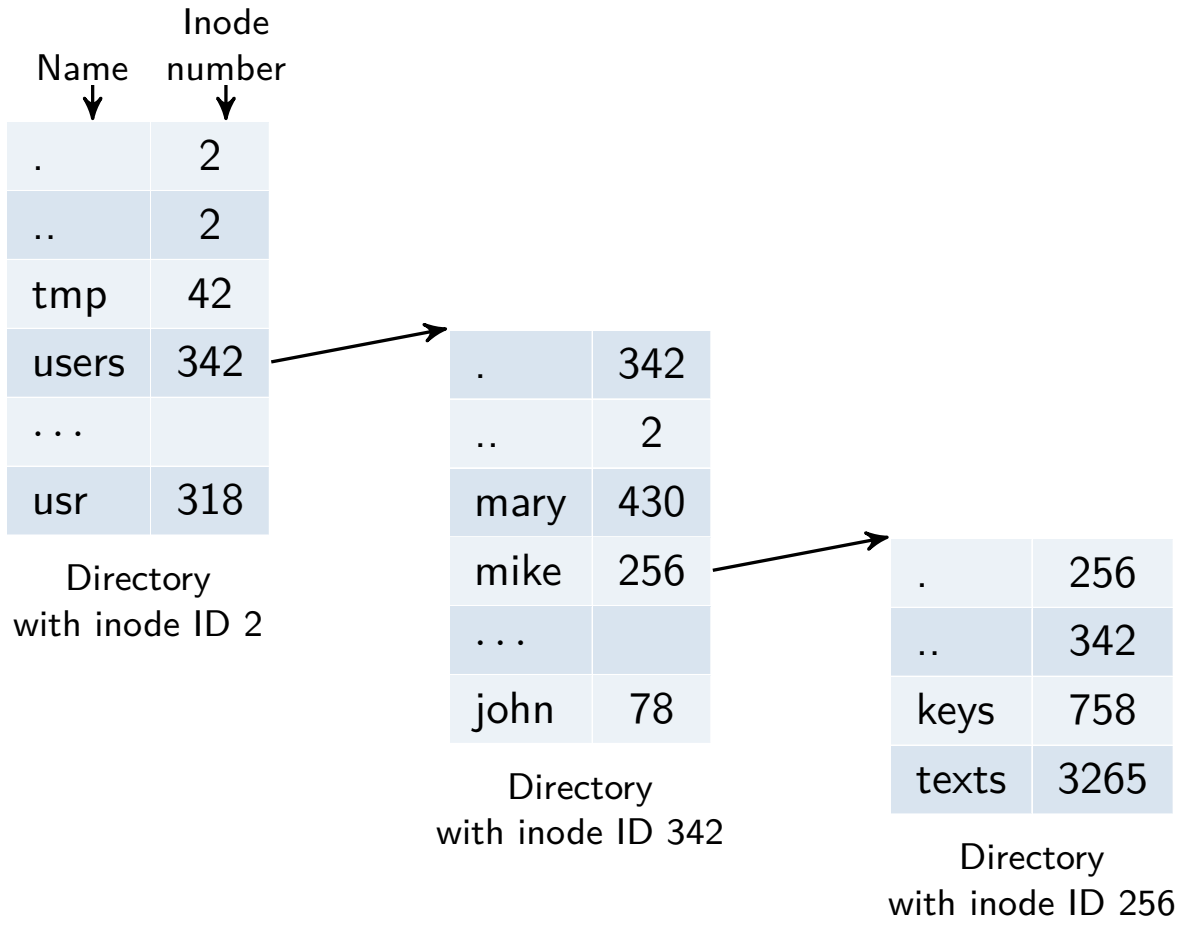
Paths in tree-based directory hierarchies (II)

Two types of paths

- **Absolute/full path**: File name specification with respect to the root directory
  - Root is "/" on UNIX/Linux and "\" on Windows
  - Example: `/users/mike/keys`
- **Relative path**: File name specification with respect to the process's current working directory
  - Examples: (Assume current working directory is `/users`)
    - `mike/keys`
    - `../users/mike/keys`
    - `./mike/keys`

Processing pathnames on Linux (I)

- Goal: Retrieve a file's physical descriptor from a pathname
- Example for `/users/mike/keys`



Processing pathnames on Linux (II)

Processing steps for `/users/mike/keys`

1. Load into memory the blocks from directory with inode-id=2
2. Search for `users` in the directory: inode-id 342 is found
3. Load inode 342 into memory
4. Load data blocks associated with inode 342 into memory
5. Search for `mike` in the directory: inode-id 256 is found
6. Load inode 256 into memory
7. Load data blocks associated with inode 256 into memory
8. Search for `keys` in the directory: inode-id 758 is found
9. Load inode 758 into memory

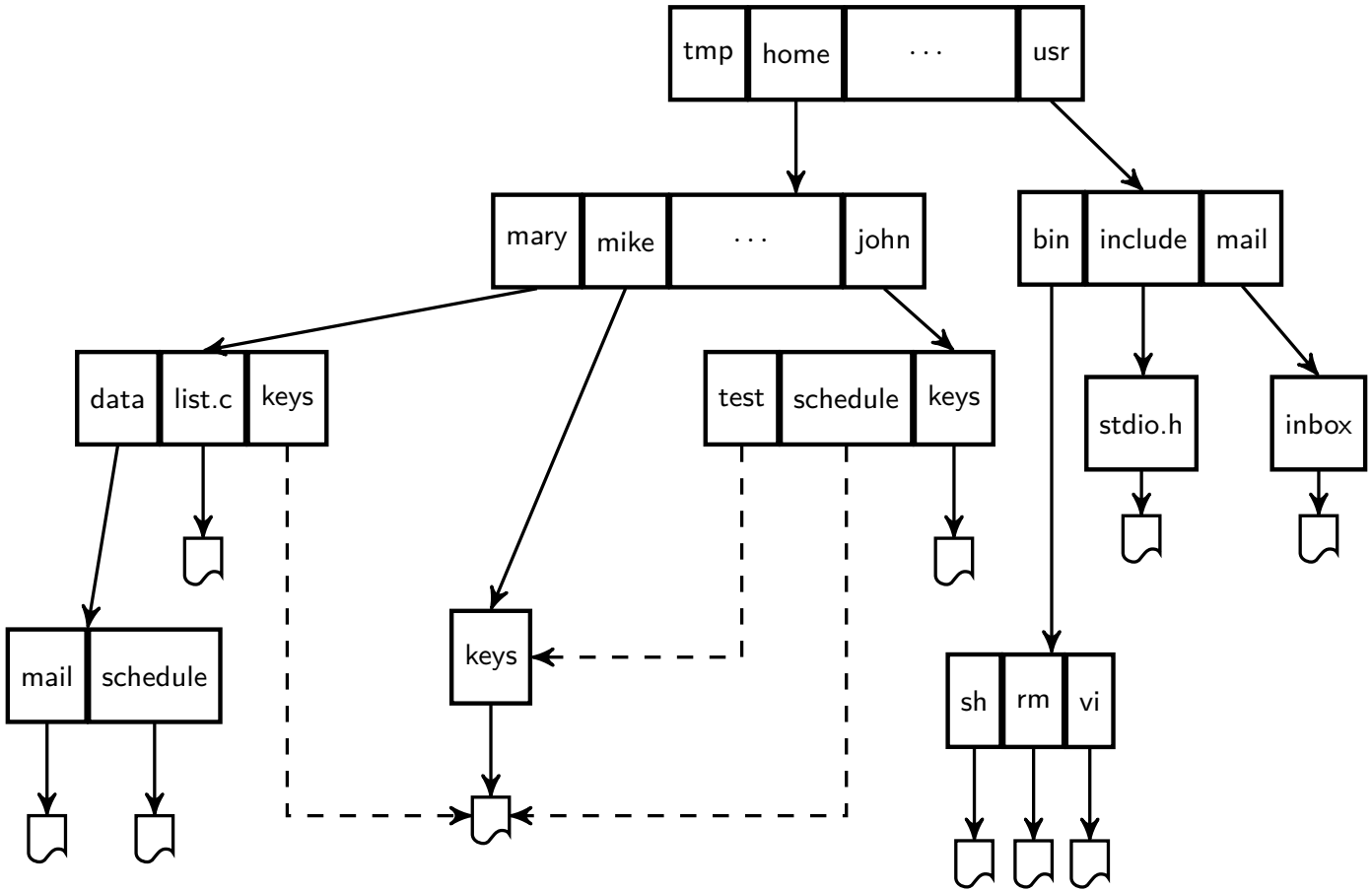
Processing pathnames on Linux (III)

When does the iterative algorithm stop?

1. The file was found
2. A token of the path being processed is not found in the current directory
3. An intermediate token of the path being processed is found in the current directory but it is a file rather than a directory

Directory hierarchies: Acyclic graph

- Two non-equivalent paths may lead to the same file
- The file system must support file name *aliases*



Aliases in acyclic graph hierarchies: links (I)

- **Link**: one of the possible names to refer to a file
  - **Physical/Hard link**: Every file descriptor is equipped with a link counter
    - Number of aliases for the file in the file system
  - **Symbolic/Soft link**: A special file that stores the pathname of another file
    - A symlink is said to be broken if the pathname does not exist or cannot be recognized
- Possible link removal policies:
  1. (hard links) Decrease link counter; if 0 $\rightarrow$ delete the file
  2. Search for all links to the file and remove them all
  3. (symlinks) Delete just the requested link, and keep the rest

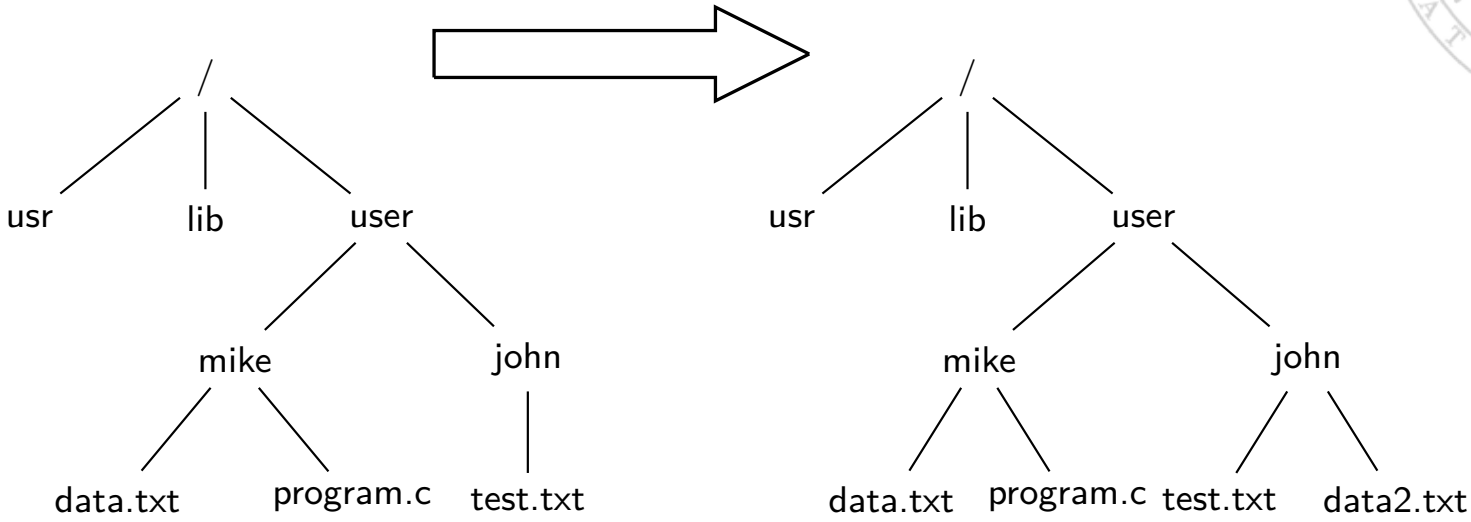
Aliases in acyclic graph hierarchies: links (II)

Issues and limitations

- Creating links may lead to cycles in the graph. Two solutions:
  1. Allow creating links to regular files, but not to directories
  2. The OS invokes an algorithm to detect potential cycles every time the user aims to create a link
- On UNIX, hard links are allowed only within the same file system
  - Symlinks may be used to point to files in the same or a different file system

Hard link: example

```
$ ln /user/mike/data.txt /user/john/data2.txt
```



mike

.	23
..	100
data.txt	28
program.c	400

john

.	80
..	100
test.txt	60

mike

.	23
..	100
data.txt	28
program.c	400

john

.	80
..	100
test.txt	60
data2.txt	28

inode 28

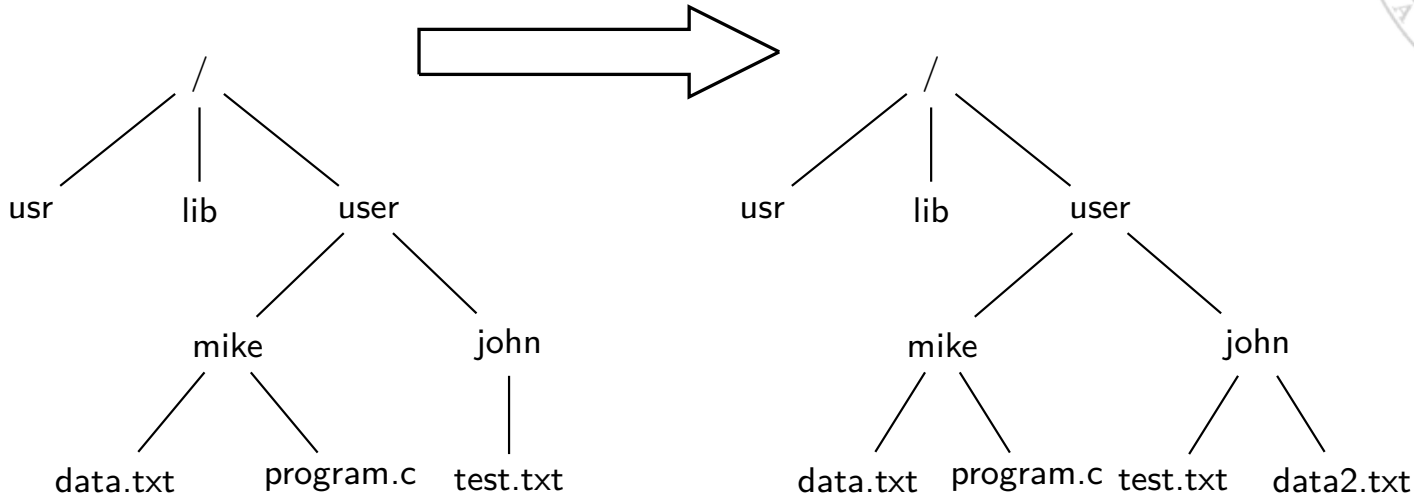
links=1
file attributes
and additional data

inode 28

links=2
file attributes
and additional data

Symbolic link: example

```
$ ln -s /user/mike/data.txt /user/john/data2.txt
```



mike

.	23
..	100
data.txt	28
program.c	400

john

.	80
..	100
test.txt	60

mike

.	23
..	100
data.txt	28
program.c	400

john

.	80
..	100
test.txt	60
data2.txt	130

inode 28

links=1
file attributes
and additional data

inode 28

links=1
file attributes
and additional data

inode 130

links=1
file attributes
and additional data

↓

"/user/mike/data.txt"

Hard links vs. symlinks

Terminal

```
jcsaez@debian:~/OS/links$ ls
info.pdf
jcsaez@debian:~/OS/links$ ln info.pdf alias1
jcsaez@debian:~/OS/links$ ln -s info.pdf alias2
jcsaez@debian:~/OS/links$ stat info.pdf
  File: ‘info.pdf’
  Size: 68883       Blocks: 136        IO Block: 1048576 regular file
Device: 24h/36d Inode: 26802399    Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1058/  jcsaez)   Gid: ( 1060/  jcsaez)
Access: 2015-08-31 14:36:43.564966921 +0200
Modify: 2015-08-31 14:36:43.585657807 +0200
Change: 2015-08-31 14:37:00.532906762 +0200
 Birth: -
```

- Two links to `info.pdf` are created
  - `alias1` (hard link)
  - `alias2` (symbolic link)
- The `stat` command can be used to retrieve file attributes, the inode ID and the current value for the link counter

Hard links vs. symlinks

Terminal

```
jcsaez@debian:~/OS/links$ stat alias1
  File: ‘alias1’
  Size: 68883       Blocks: 136        IO Block: 1048576 regular file
Device: 24h/36d Inode: 26802399    Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1058/  jcsaez)   Gid: ( 1060/  jcsaez)
Access: 2015-08-31 14:36:43.564966921 +0200
Modify: 2015-08-31 14:36:43.585657807 +0200
Change: 2015-08-31 14:37:00.532906762 +0200
 Birth: -
jcsaez@debian:~/OS/links$ stat alias2
  File: ‘alias2’ -> ‘info.pdf’
  Size: 8           Blocks: 1          IO Block: 1048576 symbolic link
Device: 24h/36d Inode: 26802400    Links: 1
Access: (0777/lrwxrwxrwx)  Uid: ( 1058/  jcsaez)   Gid: ( 1060/  jcsaez)
Access: 2015-08-31 14:37:06.702552995 +0200
Modify: 2015-08-31 14:37:06.702552995 +0200
Change: 2015-08-31 14:37:06.702552995 +0200
 Birth: -
```

- `info.pdf` and `alias1` share the same inode
- The `alias2` file is not a regular file and uses a different inode

File systems

- A file system (FS) enables to keep information organized in a storage device in a format known by the OS
- Prior to the installation of a file system, disks must be physically or logically divided into partitions or volumes
  - A partition is a disk share that the OS can manipulate independently from the other partitions
  - Common partitioning schemes:
    - Master Boot Record (MBR)
    - GUID/GPT - Mac OS X (x86)
    - Apple Partition Map (APM) - Mac OS (PowerPC)
- Once a partition has been created, the OS must create the specific data structures for the FS and store them in the partition
  - The `format` or `mkfs` commands make it possible to perform these operations on Windows and UNIX systems respectively

Filesystem metadata

- The FS installed on a partition must be self-contained
  - Both data and metadata must be stored on disk

Types of FS metadata

1. Global FS parameters
   - Partition size
   - Block size
   - Total block and physical descriptor counts
2. Critical metadata
   - Root directory
   - FATs, array of physical descriptors (UNIX)
3. Data structures to manage free space
   - Free/used blocks
   - Free/used physical descriptors (e.g. inodes)

File systems and partition structures

- Partition or volume: coherent set of data and metadata
- Examples:

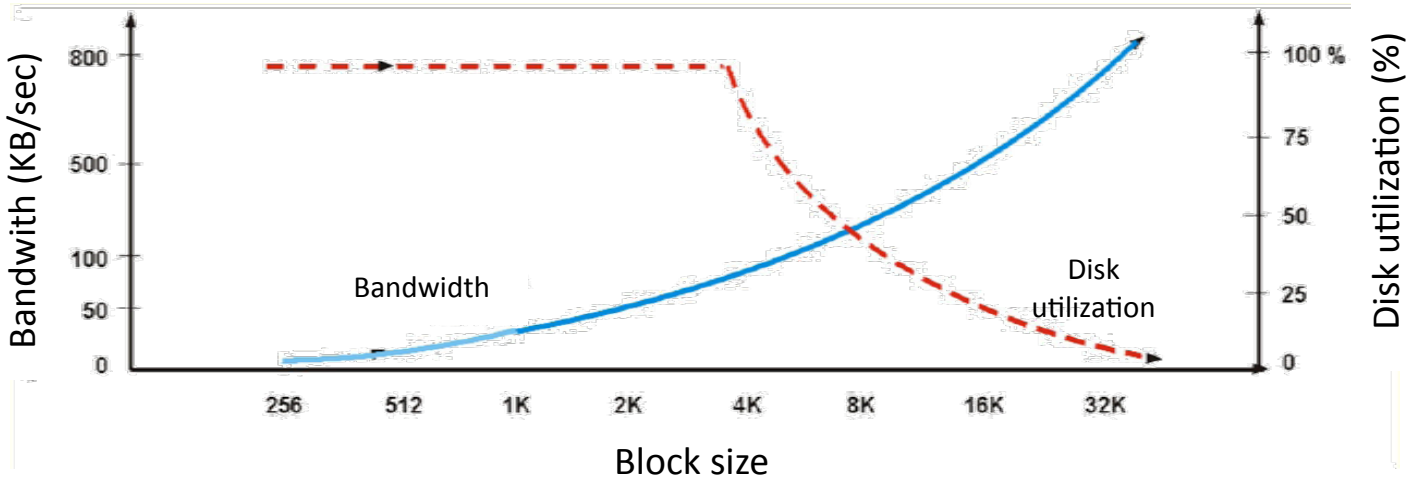
CD-ROM

FAT (MS-DOS)

UNIX

Setting the block size

- **Block**: Logical set of disk sectors and minimal transfer unit used by the OS
  - Every file occupies at least one block on disk
  - The choice of the block size has an important impact on I/O performance
  - Most OSes define a default block size
  - Users can specify which block size to use when installing the FS



Internal fragmentation

Example

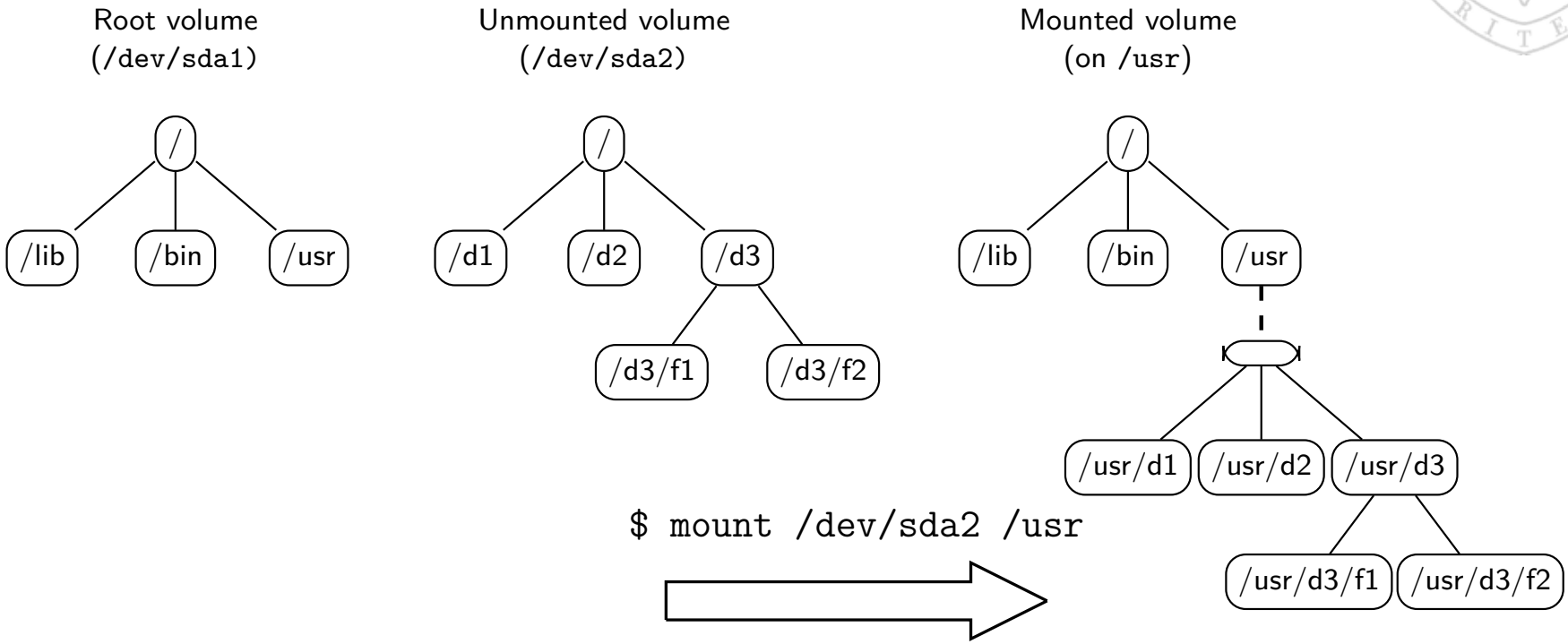
- File system where the volume is completely full with 7KB files
  - Block size = 2KB
- Calculate the degree of internal fragmentation (fraction of disk space wasted) assuming that the space occupied by metadata is negligible
  - How much space on disk is needed to store a 7KB file?
    - $\text{Blocks}_{7KB-\text{file}} = \left\lceil \frac{\text{FileSize(bytes)}}{\text{BlockSize(bytes)}} \right\rceil = \left\lceil \frac{7KB}{2KB} \right\rceil = 4 blocks\ (8KB)$

    - Percentage of disk space wasted $= \frac{1KB}{8KB} \cdot 100 = 12.5\%$
- For a 1TB disk $\rightarrow$ 125GB would be wasted!!

Dealing with multiple file systems

- How are multiple file systems exposed to the user?
  - On Windows a separate directory tree is exposed for each file system (`c:\users\mike\keys`, `j:\joe\tmp`)
  - On UNIX/Linux a unique tree is maintained for the whole system (`/users/mike/keys`, `/mnt/joe/tmp`, ...)
- On UNIX, administrative commands are needed to maintain the file system tree: `mount`, `umount`
  - `mount /dev/sda3 /users`
  - `umount /users`
- Having a unique tree offers a unified view of the files and hide the various device types to user processes
- Unfortunately, the unique tree also makes it harder processing pathnames and dealing with links

Mounting file systems or partitions



Contents

POSIX services for files (I)

- Logical view
  - A file is a byte array to user processes
  - The file has an associated position indicator that points to the next byte to be read/written
- Numeric file descriptor: integer ranging from 0 to 64K
  - Number used by the programmer to perform operations on an open file
  - The descriptor can refer to a regular or a special file
  - Warning: Numeric file descriptor (API abstraction) $\neq$ Physical descriptor $\neq$ Logical descriptor
- Pre-defined numeric file descriptors:
  - 0 $\rightarrow$ standard input
  - 1 $\rightarrow$ standard output
  - 2 $\rightarrow$ standard error output

POSIX services for files (II)

- Specific services are provided to read and modify file attributes
- Every file has protection information associated with it

```
owner    group    other
rwx      rwx      rwx
```

  - Three bits encode the r(ead), w(rite) and e(x)ecute permissions for each *agent*
  - The File API represents this information via octal digits
    - Example: 755 means `rwxr-xr-x`

Basic operations on files

- **creat**: Creates a file with a given path name and protection information. The function returns a numeric file descriptor
- **open**: Opens a file with a given path name to perform write/read operations on it.The function returns a numeric file descriptor
- **unlink**: Removes a file
- **read**: Reads data from an open file using the associated numeric descriptor. The retrieved data is copied onto a memory region belonging to the process
- **write**: Write data located in a memory region onto an open file using the associated numeric descriptor
- **close**: closes an open file
- **lseek**: relocate the position indicator of an open file
- **stat**: returns file attributes

Create a file

creat()

```
int creat(char *name, mode_t mode);
```

- Arguments:
  - `name`: path name of the file
  - `mode`: Permission bitmask
- Return value:
  - Returns an integer file descriptor or -1 on failure
- Examples:

```
fd=creat("data.txt", 0751);
fd=open("data.txt", O_WRONLY|O_CREAT|O_TRUNC,0751);
```

Open a file

open()

```
int open(char *name, int flag, ...);
```

- Arguments:
  - `name`: path name of the file
  - `flags`: Option bitmask
    - `O_RDONLY` Read only
    - `O_WRONLY` Write only
    - `O_RDWR` Read and write
    - `O_APPEND` The file offset gets positioned at the end of the file
    - `O_CREAT` If the file does not exists create it
    - `O_TRUNC` Truncate the file when opened for writing
- Return value:
  - Returns an integer file descriptor or -1 on failure

Remove a directory entry

unlink()

```
int unlink(const char* path);
```

- Arguments:
  - `path`: path name of the file
- Return value:
  - Returns 0 on success and -1 upon failure
- Description:
  - Removes the specified directory entry and decrements the link counter of the associated file
  - When the link counter reaches 0 and no other process is working with the file, the file is actually removed (freeing up the associated space on disk)

Read data from a file

`read()`

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

- Arguments:
  - `fd`: numeric file descriptor
  - `buf`: start address of the memory region where the read data will be copied onto
  - `n_bytes`: number of bytes to be read
- Return value:
  - On success, it returns the number of bytes actually read. On failure the function returns -1
- Description:
  - Transfers as many as `n_bytes` to the process address space
    - Less than `n_bytes` may be transferred (e.g., we reached the end of file)
  - `read()` implictly moves forward the file's position indicator by the amount of bytes actually read

Write data to a file

`write()`

```
ssize_t write(int fd, void *buf, size_t n_bytes);
```

- Arguments:
  - `fd`: numeric file descriptor
  - `buf`: start address of the memory region where the data to be written is stored
  - `n_bytes`: number of bytes to be written
- Return value:
  - On success, it returns the number of bytes actually written. On failure the function returns -1
- Description:
  - Writes as many as `n_bytes` to the file
    - Less than `n_bytes` may be transferred (e.g., if we exceed the maximum file size or the process is interrupted by a signal)
  - `write()` implicitly moves forward the file's position indicator by the amount of bytes actually written

Close a file

close()

```
int close(int fd);
```

- Arguments:
  - `fd`: numeric file descriptor
- Return value:
  - Returns 0 on success and -1 on failure
- Description:
  - The process loses the association with the file

Relocate the file's position indicator

lseek()

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

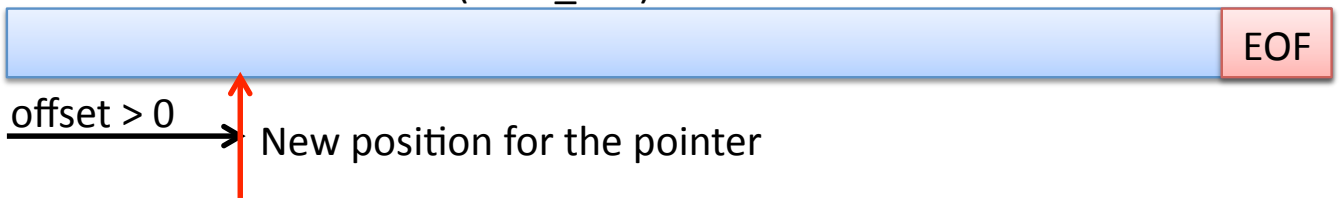
- Arguments:
  - `fd`: numeric file descriptor
  - `offset`: relative or absolute offset
  - `whence`: base for the pointer relocation
- Return value:
  - On success, it returns the new location of the position indicator. On failure the function returns -1
- Description:
  - Establishes a new location for the file's position indicator:
    - `whence == SEEK_SET` → position = offset
    - `whence == SEEK_CUR` → position = cur. position + offset
    - `whence == SEEK_END` → position = file size + offset

Relocate the file's position indicator

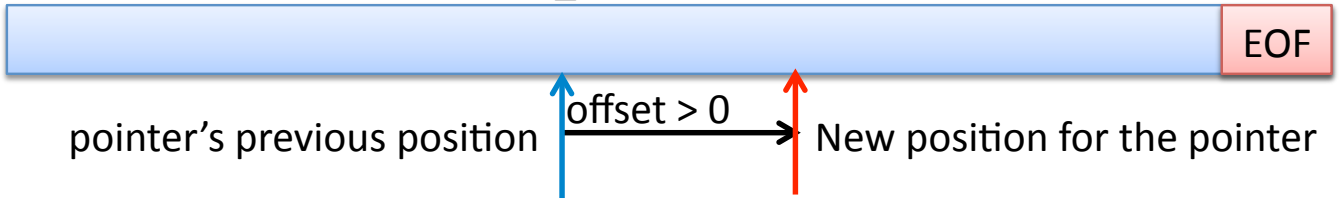
```
off_t lseek(int fd,
            off_t offset,
            int whence)
```

In UNIX-like file systems, it is simple to implement lseek() due to the indexed file allocation. Typically, just one disk operation is required for this task.

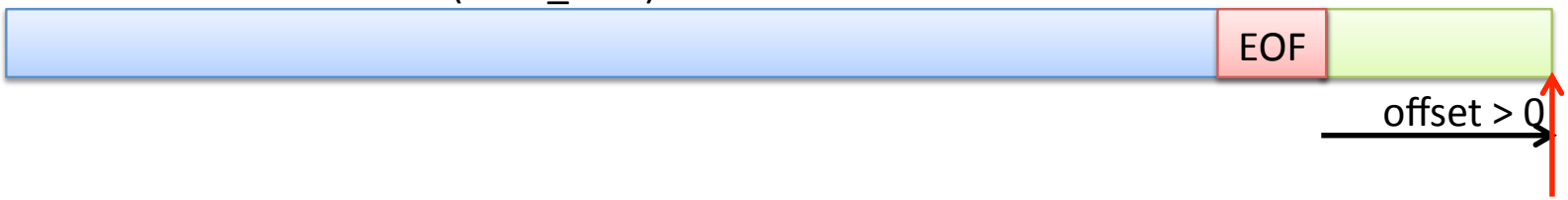
- Case #1: whence == 0 (SEEK_SET)



- Case #2: whence == 1 (SEEK_CUR)



- Case #3: whence == 2 (SEEK_END)



Retrieve file attributes

`stat()` and `fstat()`

```
int stat(char *name, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

- Arguments:
  - `name`: path name of the file
  - `fd`: numeric file descriptor
  - `buf`: pointer to a `struct stat` where the OS will store the value of the various attributes
    - Run "`man 2 stat`" to know the fields in `struct stat`
- Return value:
  - Returns 0 on success and -1 on failure

Example: create a copy of a file (1/2)

Usage

```
copy_file <input-file> <output-file>
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define BUFSIZE     512

void main(int argc, char **argv) {
    int fd_in, fd_out, n_read;
    char buffer[BUFSIZE];

    fd_in = open(argv[1], O_RDONLY); /* open the input file */
    if (fd_in < 0){
        perror("open");    exit(1);
    }

    fd_out = creat(argv[2], 0644);      /* create the output file */
    if (fd_out < 0){
        close(fd_in);
        perror("creat");    exit(1);
     }
```

Example: create a copy of a file (2/2)

```
C  /* main loop to transfer data between files */     while ((n_read =
read(fd_in, buffer, BUFSIZE)) > 0)  {             /* Transfer data from
the buffer onto the output file */         if (write(fd_out, buffer,
n_read) < n_read)  {             perror("write");
close(fd_in); close(fd_out);             exit(1);         }     }
if (n_read < 0)  {             perror("read");          close(fd_in);
close(fd_out);         exit(1);     }      close(fd_in);  close(fd_out);
exit(0); }
```

POSIX services for directories

- Logical view on UNIX: table of pairs *(name, inode number)*
- Difficult management:
  - File names have variable length
  - On UNIX and Linux, multiple file systems may "coexist" in a unique directory tree/graph
    - Different directory implementations
- POSIX services: operations to create, read, modify and traverse directories

Basic operations on directories

- **mkdir**: creates a directory
- **rmdir**: removes an empty directory
- **opendir**: opens a directory as a set of abstract entries and locates the directory's position indicator in the first entry
- **readdir**: reads the next directory entry
- **rewinddir**: places the directory's position indicator in the first entry
- **closedir**: closes a directory for reading
- **link/symlink**: Creates a new directory entry with a hard link or a symbolic link
- **unlink**: Removes a directory entry
- **rename**: Renames a directory entry
- **chdir**: Changes the process's current working directory
- **getcwd**: Returns the path of the current working directory

Create a directory

mkdir()

```
int mkdir(const char *name, mode_t mode);
```

- Arguments:
  - `name`: path name of the directory
  - `mode`: Permission bit mask
- Return value:
  - Returns 0 on success; -1 upon failure
- Description:
  - Creates a new directory with a given name
  - `UID_owner = UID_effective_user`
  - `GID_owner = GID_effective_user`

Remove a directory

rmdir()

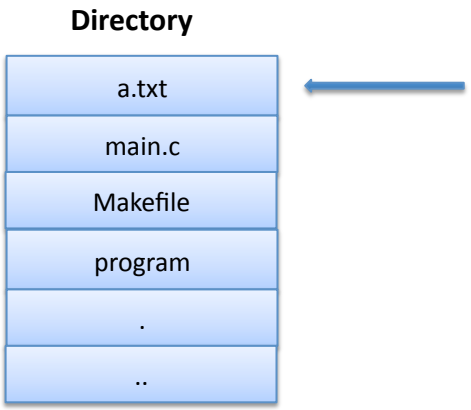
```
int rmdir(const char *name);
```

- Arguments:
  - `name`: path name of the directory
- Return value:
  - Returns 0 on success; -1 upon failure
- Description:
  - Removes the directory as long as it is empty
  - If the directory is not empty, `rmdir()` returns an error

POSIX services for directories (II)

- Directories are exposed to the programmer as a table (array of `struct dirent`) with an associated position indicator

```
struct dirent {
        /* file name */
        char *d_name;
        ...
}
```



- When opening the directory (`opendir()`) the position indicator points to the first entry
- `readdir()` returns the current entry and moves the position indicator to the next entry
  - If all entries have been already visited, `readdir()` returns NULL
- `rewinddir()` makes the position indicator point to the first entry

Open a directory

opendir()

```
DIR *opendir(char *dirname);
```

- Arguments:
  - `dirname`: path name of the directory
- Return value:
  - Returns a non-NULL directory descriptor on success, and NULL upon failure
- Description:
  - Opens a directory and exposes it to the user as a sequence of entries
  - The position indicator points to the first entry

Read directory entries

`readdir()`

```
struct dirent *readdir(DIR *dirp);
```

- Arguments:
  - `dirp`: descriptor returned by `opendir()`
- Return value:
  - Returns a non-NULL `struct dirent` entry on success, and NULL in the event no more directory entries exist or an error occurred.
- Description:
  - Returns the current entry and moves the position indicator to the next entry
  - Note that `struct dirent`'s is implementation-specific. Programs should only access the `d_name` field (`char*`).

Rewind the read pointer in a directory

`rewinddir()`

```
void rewinddir(DIR *dirp);
```

- Arguments:
  - `dirp`: descriptor returned by `opendir()`
- Description:
  - Rewinds the position indicator back to the first entry

Close a directory

closedir()

```
int closedir(DIR *dirp);
```

- Arguments:
  - dirp: descriptor returned by opendir()
- Return value:
  - Returns 0 on success; -1 upon failure
- Description:
  - The process loses the association with the directory

Create a directory entry

`link()` and `symlink()`

```
int link(const char *existing, const char *new);
int symlink(const char *existing, const char *new);
```

- Arguments:
  - `existing`: path name of a existing file
  - `new`: path name of a the new hard link or symbolic link
- Return value:
  - Returns 0 on success; -1 upon failure
- Description:
  - Creates a new hard/symbolic link to an existing file
  - The OS does not keep track of the existing file name
  - `existing` cannot be the path of a directory

Remove a directory entry

unlink()

```
int unlink(const char* path);
```

- Arguments:
  - `path`: path name of the file
- Return value:
  - Returns 0 on success and -1 upon failure
- Description:
  - Removes the specified directory entry and decreases the link counter of the associated file
  - When the link counter reaches 0 and no other process is working with the file, the file is actually removed (freeing up the associated space on disk)

Rename a directory entry

`rename()`

```
int rename(char *old, char *new);
```

- Arguments:
  - `old`: path name of the file
  - `new`: new path name for the file
- Return value:
  - Returns 0 on success and -1 upon failure
- Description
  - Renames the file associated with the `old` path into the `new` path

Change the current working directory

chdir()

```
int chdir(char *name);
```

- Arguments:
  - `name`: path name of the directory
- Return value:
  - Returns 0 on success and -1 upon failure
- Description
  - Changes the process's current working directory to that specified by `name`

Get path of current working directory

getcwd()

```
char *getcwd(char *buf, size_t size);
```

- Arguments:
  - `buf`: pointer to a buffer where the libc stores the path of the current working directory
  - `size`: maximum amount of characters to store in buf
- Return value:
  - Returns a pointer to `buf` on success and NULL upon failure
- Description
  - Retrieves the full path of the process's current working directory

Example: listing entries in a directory (1/2)

Usage

```
list_dir <directory>
```

```c
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

#define MAX_BUF   256

void main(int argc, char **argv){
   DIR *dirp;
   struct dirent *dp;
   char buf[MAX_BUF];

   /* Prints the path of the current working directory */
   getcwd(buf, MAX_BUF);

   printf("Current working directory: %s\n", buf);

   ...
```

Example: listing entries in a directory (2/2)

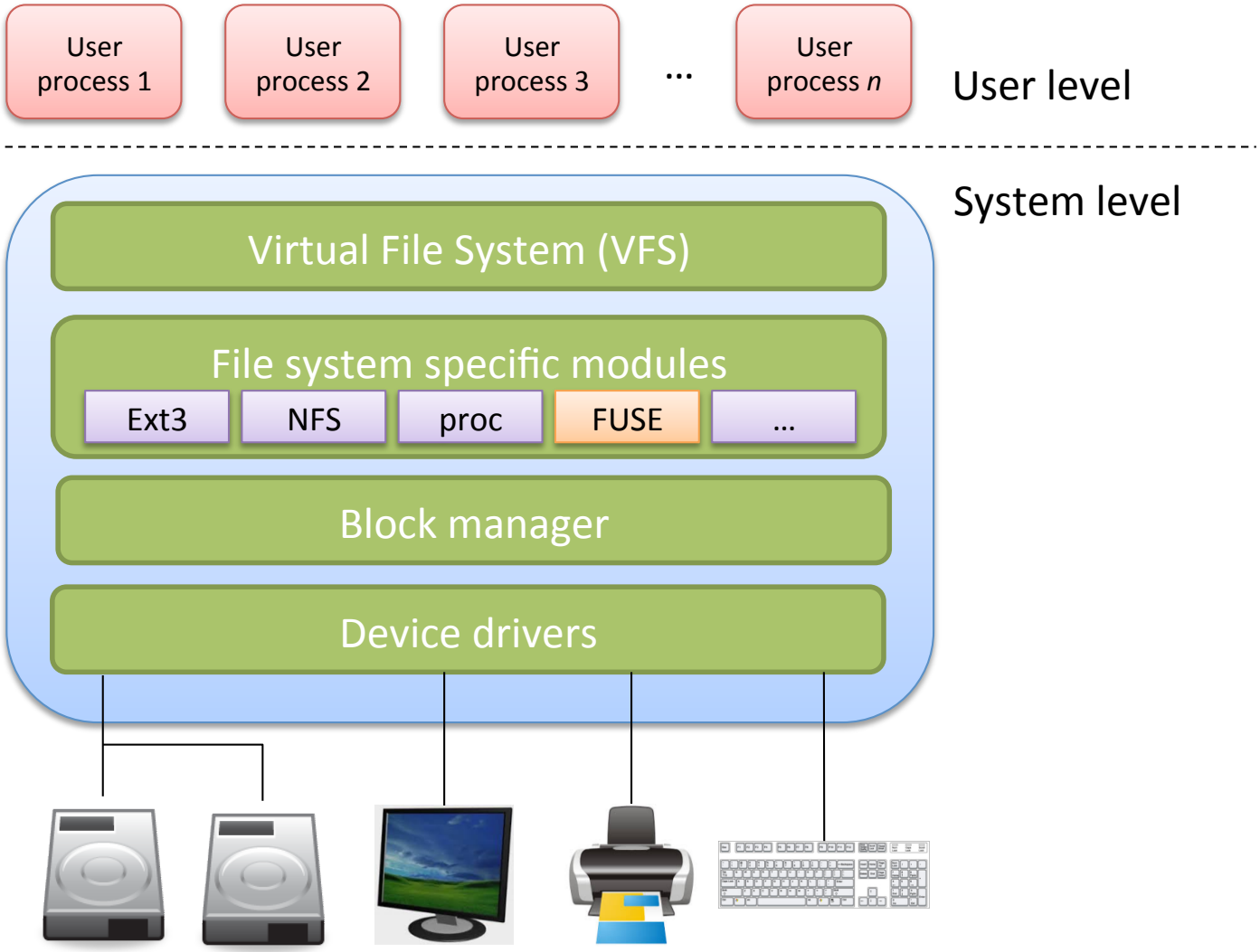
“‘C /* Open the directory passed as the first argument */ dirp = opendir(argv[1]);

```
if (dirp == NULL)  {
    fprintf(stderr,"Couldn't open %s\n", argv[1]);
} else {
    /* Iterate the set of directory entries */
    while ( (dp = readdir(dirp)) != NULL)
        printf("%s\n", dp->d_name);
  closedir(dirp);
```

} exit(0); } “‘

Contents

Architecture of the File Management System



Components of the File Management System

Virtual file system (VFS)

- Layer in charge of implementing file-related system calls
  - Create directory abstractions for user processes
  - Pathname processing algorithms
  - Basic file services (OS API)
  - File system agnostic abstractions:
    - Open files in the VFS represented by a *wrapper* of the physical descriptor (*vnode*: virtual node)

Components of the File Management System

Filesystem-specific modules:

- Layer in charge of implementing services for the various file systems supported by the OS
  - Features a module for each supported file system (UNIX, EXT3, NTFS, FAT, ...) or pseudo file systems (tmpfs, /proc, /sys, ...)
  - Each module is equipped with algorithms making it possible to translate logical addresses issued by user processes (byte level) into physical addresses (block level)
  - The various modules take care of managing free space in partitions, block allocation for files and actual manipulation of physical descriptors (such as inodes or directory entries)

Components of the File Management System

Block manager:

- Establishes the communication with the various device drivers via specific commands to read and write blocks from/to storage devices
  - Example: Read block 320 of FS#1 → `read_block 320 /dev/sda2`
- Implements mechanisms to optimize I/O performance, such as the *buffer cache*

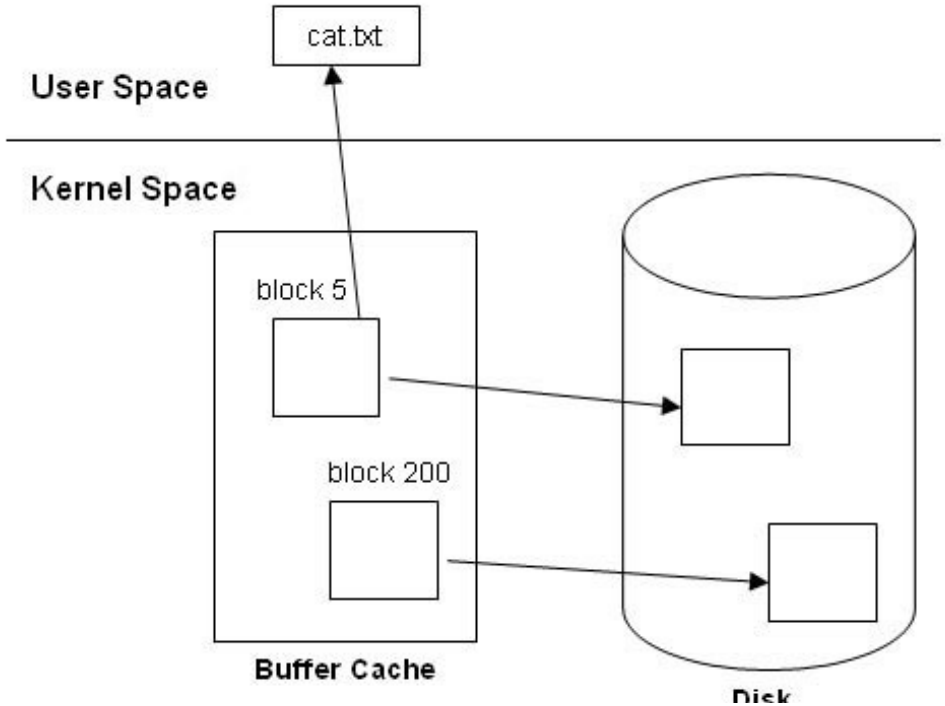
Components of the File Management System

Device drivers of storage devices:

- Translate high-level I/O request into the low-level format recognized by the HW controller of the target device
- A device driver exists for a set of devices of the same class or I/O protocol
  - The driver is in charge of *scheduling* multiple I/O requests to ensure acceptable latency and bandwidth

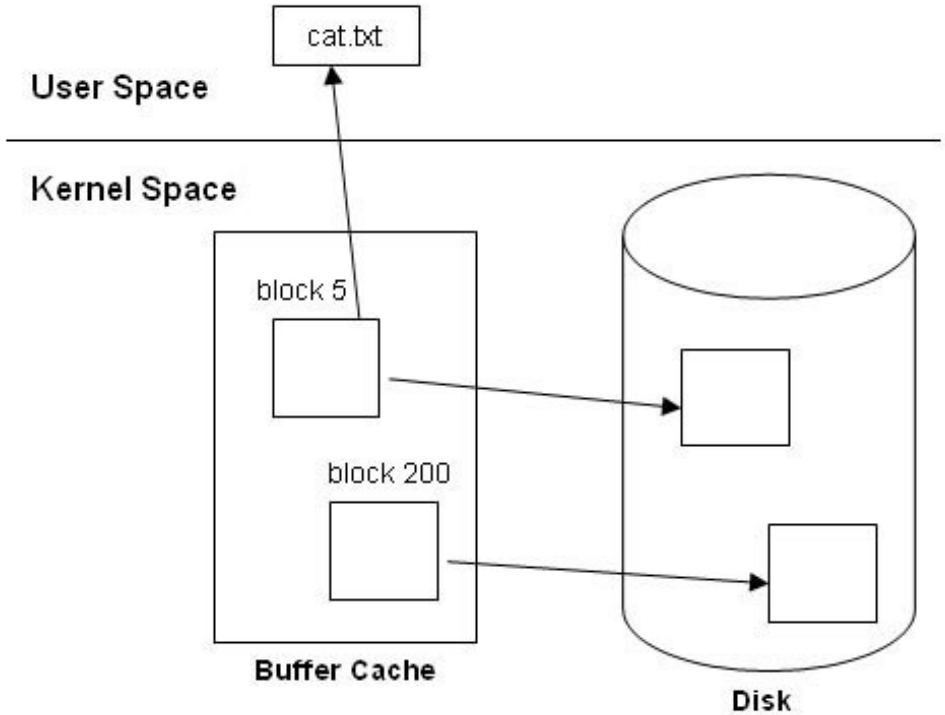
OS buffer cache

- Disk accesses are several orders of magnitude slower than accesses to main memory (RAM)
- **Buffer cache**: A memory region used as a software-controlled cache to keep blocks from storage devices
  - Leverages the principles of spatial locality and temporal locality



OS buffer cache

- Main issue: data consistency
  - Information may be updated in the buffer cache but not on disk
- The OS must implement a **replacement policy** and a **write policy**



Buffer cache replacement policies

Problem statement

- When the block manager (BM) receives a R/W request for a specific block it must look for it in the *buffer cache* first
  - If the block is not in the cache, the BM retrieves it from disk and stores it in the cache
  - Unfortunately, the buffer cache may be full: Another block must be evicted to make room for the new block

Replacement policies

- **FIFO** (First In First Out)
- **MRU** (Most Recently Used)
- **LRU** (Least Recently Used)
  - Blocks with a high reuse rate are kept in the cache
  - The most widely used policy

Buffer cache write policies (I)

- **Write-through**: Whenever a block is modified in cache, the block is updated on disk as well
  - No reliability issues, but poor system performance
- **Write-back**: A modified block is updated on disk only in the event of being evicted by the replacement policy
  - Optimal for performance, but subject to serious reliability issues

Buffer cache write policies (II)

- **Delayed-write**: The contents of modified blocks in cache are updated periodically on disk (e.g., every 30s. on UNIX)
  - Good performance/reliability trade-off
  - Better reliability than *write-back*
  - Special blocks (e.g., metadata) are updated immediately
  - A disk drive should not be removed from a computer without flushing the buffer cache first
- **Write-on-close**: Modified blocks from a file are updated on disk only when the process closes the file