Karlsruhe Institute of Technology

Betriebssysteme

16. File Systems

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP

File Systems

- Motivation, Introduction
- File Management
- Directory Management
- Objectives:
  - To explain the function of file systems
  - To describe the interfaces to file systems
  - To discuss file-system design tradeoffs
    - access-methods
    - file sharing
    - file locking

Motivation (1)

OS Abstraction	HW Resource
Processes, Threads	CPU
Address Space	Main Memory (RAM)
Files	Disk, CD, . . .

- Files are the third major OS-provided abstraction over HW resources
- *Do we still need files and a classical file system or better a database with an object store?*

Motivation (2)

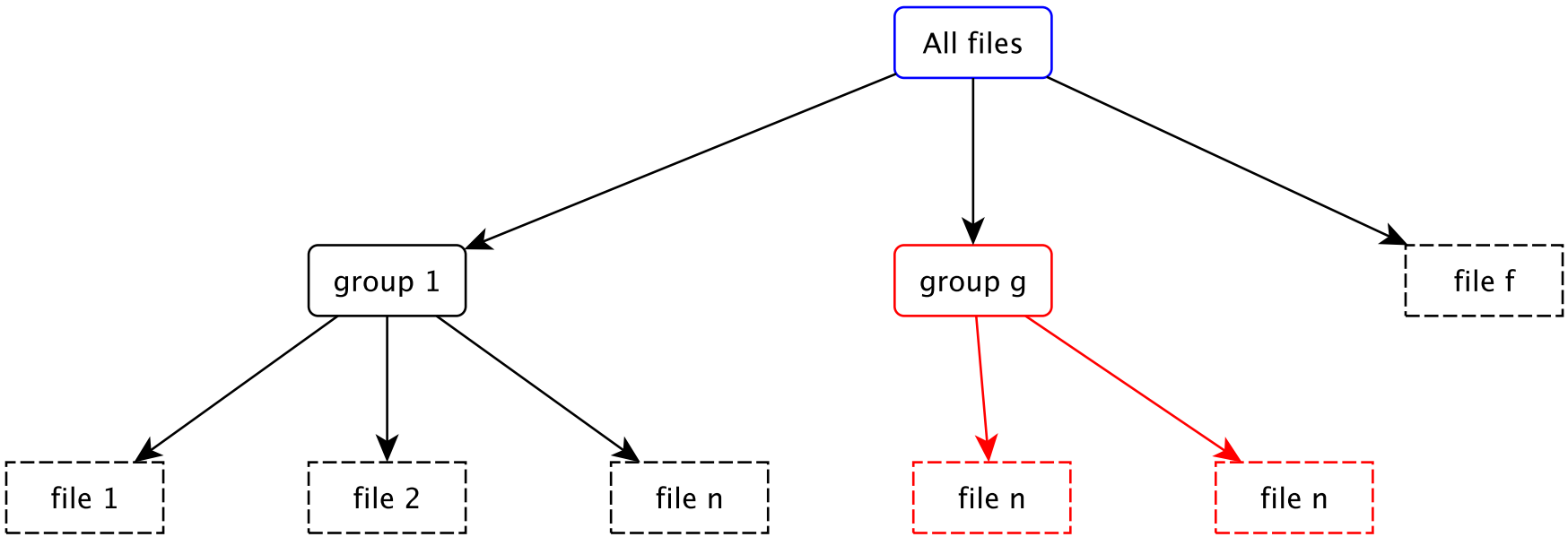
- Enable storing of large amount of data
  - File - contiguous logical address space
- File types:
  - data
    - numeric
    - character
    - binary
  - Program
- Store data/program consistently & persistently
- Look-up easily previously stored data/program

File Systems

- Most files are still located on disks which are really messy physical devices:
  - Errors, bad blocks, redundant arrays of disks (RAID), . . .
- Job of an OS is to hide this mess from higher level software
  - Low-level device control (initiate a disk read, etc.)
  - High-level abstractions (read file)
- OS might provide different levels of disk access to different clients (applications)
  - Physical disk (surface, cylinder, sector)
  - Logical disk = partition (disk block#)
  - Logical volume = multiple partitions (volume block#)
  - Logical file (file block, record, byte#)

Overview File System

- OS may support multiple file systems
  - Instances of the same FS type
  - Different FS types, e.g. ext2 & Reiser
- All file systems are typically bound into a single **namespace**
  - Often hierarchical as a rooted tree
    - Internal node = directory (mount point)



File

- Collection of related information
  - Executable program
  - Text files
  - Compressed binary images
  - Structured document
  - . . .
- A file has a set of **attributes**, i.e. its **meta data**
- Attributes differ between OSes and FSs, e.g.
  - **Name, identifier**
  - **Type**
  - **Location** (physical address of a file on device)
  - **Size** (# bytes or # blocks)
  - **Protection** (who can access and how)

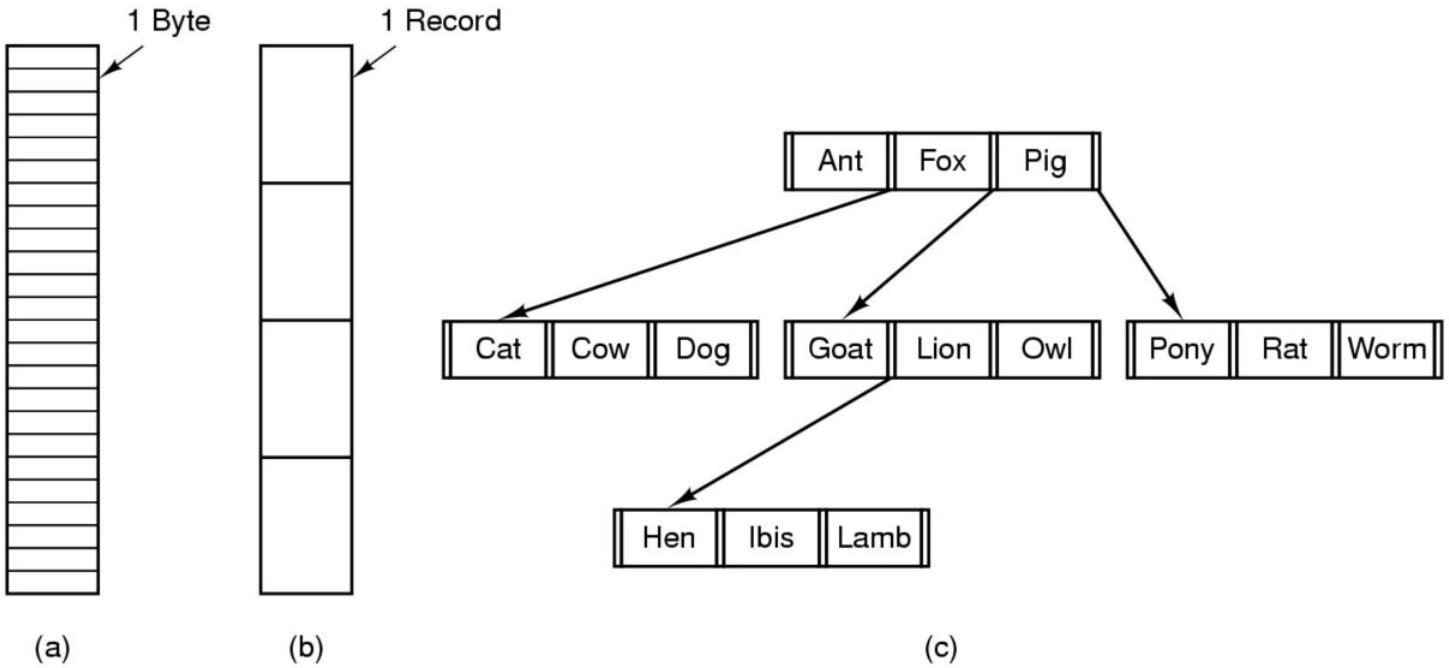
Typical File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current Size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File Structure (1)

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable executable object

File Structure (2) (OS's Point of View)



Three kinds of files:

(a) byte sequence (provides maximal flexibility)

(b) record sequence (often with fixed sized records)

(c) Tree (sometimes with variable sized records)

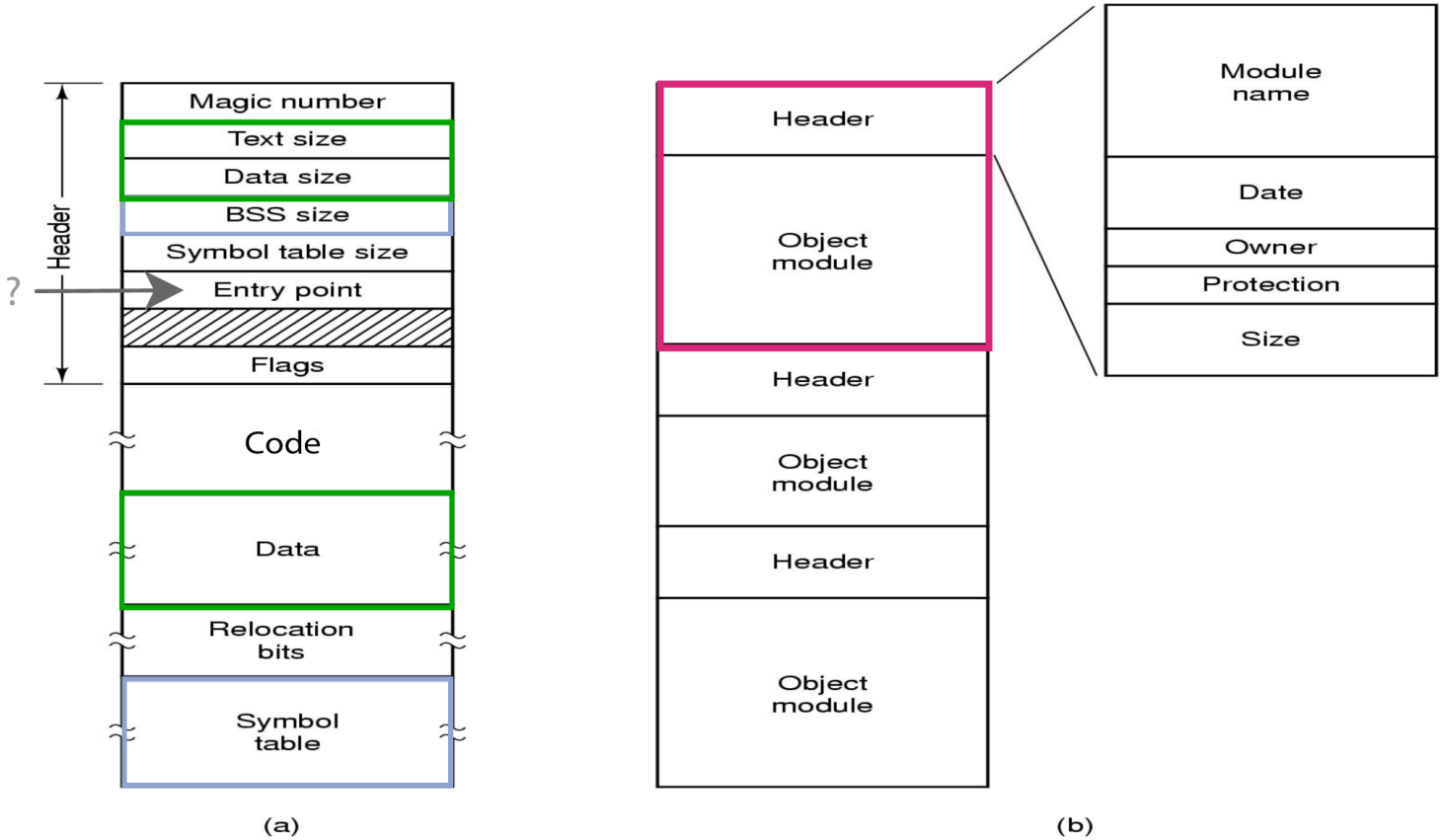
File Types

- Regular files
  - executable, dll, object, source, text, . . .
- Special files
  - Directory, device (character, block), links
- A file's type can be encoded (see `man 1 file`) in
  - its FS internal data structure (e.g. Unix)
    - Inode
  - its name (e.g. file extensions in Windows)
    - .com, .exe, .bat, .dll, .jpg, . . .
  - its content (e.g. Unix)
    - magic number or an initial character (e.g. `#!` for shell scripts)

Regular File Types (1)

file type	usual extensions	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files, grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Regular File Types (2)



(a) Executable file (e.g. ELF)

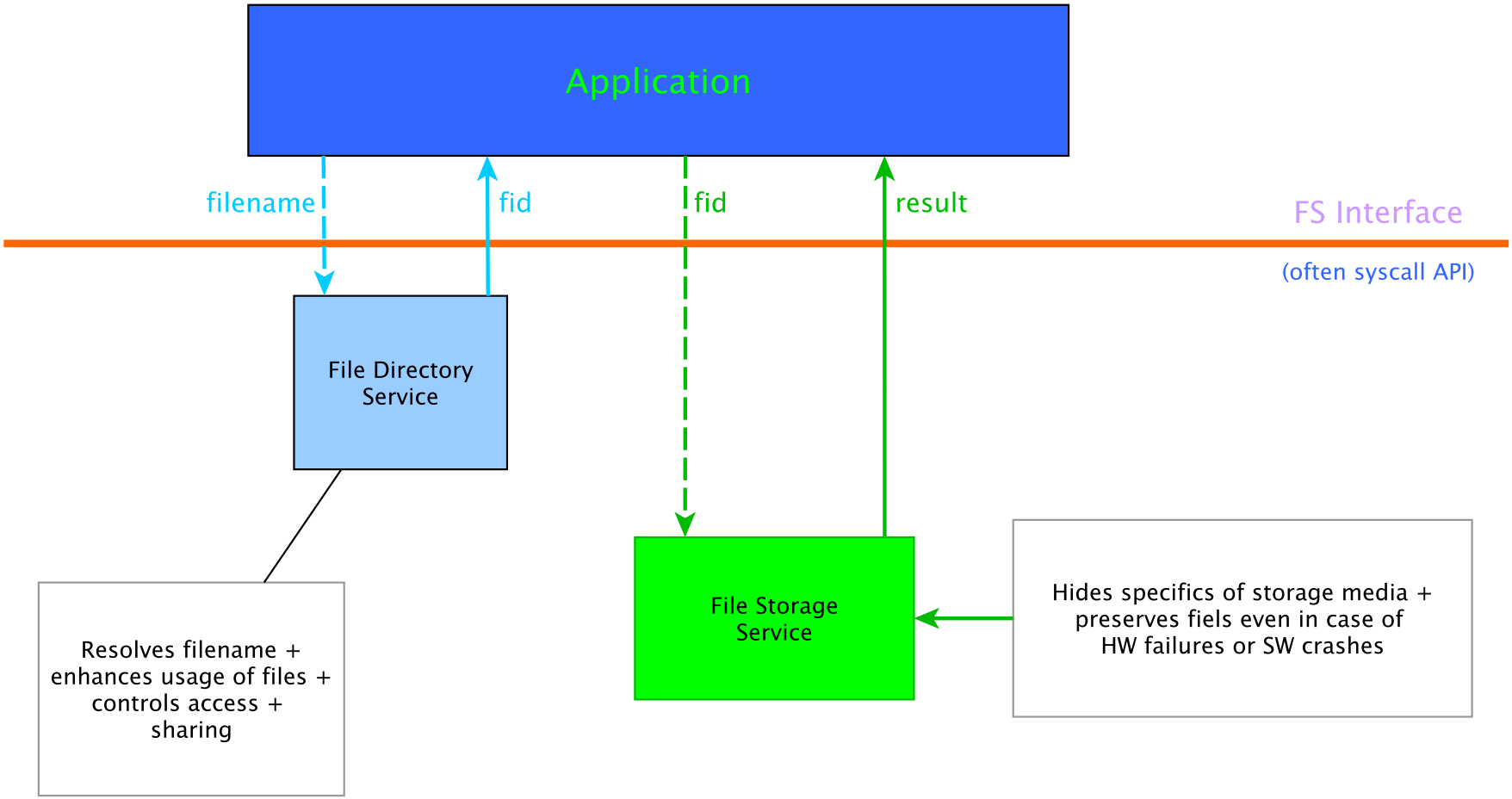
(b) Archive file (e.g. tar)

Abstract File Operations

A file is an abstract data type/object offering

- `create()`
- `write()`
- `read()`
- `reposition()` (within a file)
- `delete()`
- `truncate()`
- `open(`F_i`)` - search the directory structure on disk for entry F_i, and move its meta data to memory
- `close(`F_i`)` - move cached meta data of entry F_i in memory to directory structure on disk

Interaction with a FS



Goals of File Management

- Provide a convenient naming scheme for files
- Provide uniform I/O support for a variety of storage device types
- Provide standardized set of I/O interface functions
- Minimize/eliminate loss or corruption of data
- Provide I/O support and access control for multiple users
- Enhance system administration (e.g., backup)
- Provide acceptable performance

File Names

- FS with a convenient *naming scheme*, e.g.
  - Textual names
  - Restricted alphabet, i.e.
    - Only certain characters (e.g. no ? or /)
    - Limited length
    - only certain formats, e.g.
      DOS: 8 character string.**xyz** character suffix
      XP: 255 character string.**xyz** character suffix
    - Case (in)sensitive
    - Names must fulfill certain convenient, extension `xyz.c` (or `xyz.C`) if C(++)-Compiler should run

Open Files

- Several meta data are needed to manage open files:
  - **file pointer:** pointer to last read/write location, per process that has the file opened
  - **access rights:** per-process/task access mode information
  - **file-open count:** counter of number of times a file is opened - to allow removal of data from open-file table when last process closes
  - **disk location:** cache of data access information

File Access

- Strictly sequential access (early system)
  - read all bytes/records from the beginning
  - cannot jump around, could only rewind
  - sufficient as long as storage was a tape
- Random access (current systems)
  - bytes/records read in any order
  - essential for database systems

File Organization and Access

- Possible access patterns:
  - Read the whole file
  - Read individual blocks of a file
  - Read blocks preceding/following the current one
  - Retrieve a subset of records
  - Write/update a complete file sequentially
  - Insert/delete/update one record in a file
  - Update blocks in a file

Access Methods

- Sequential Access

```
read next
write next
rewind
no read after last write
append
```

- Direct access

```
read n
write n
position to n
read next
write next
```

```
n = relative position number
```

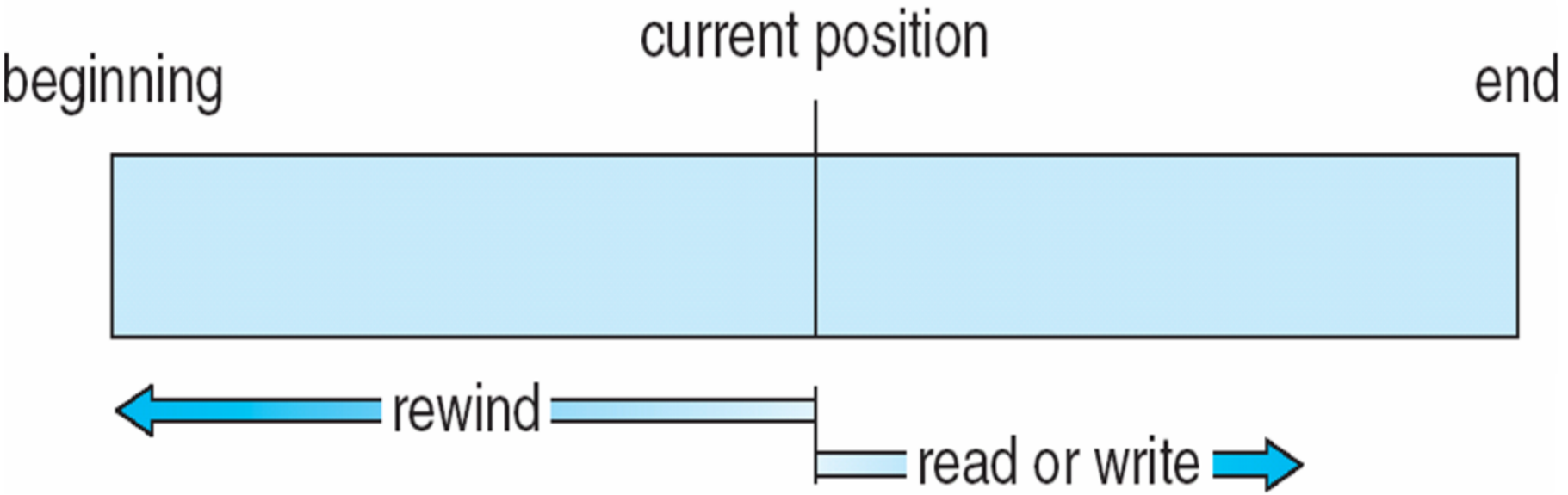
File Access Methods(2)

- Plain (unstructured) file (generic file)
  - Entity: **byte** (sometimes: **block**)
  - If an application wants to structure a persistent data container it has to implement its internal structure
- Structured file
  - Entity: **record** (or user type objects. . .)

Remark:

Since Unix, many OSes only offer plain files, applications and libraries can implement specific structured file types on top of this.

A Sequential Access to a File



Operations on Unstructured Files

```
CreateFile(pathname)
DestroyFile(pathname)
OpenFile(pathname, read/write)
ReadFile(FID, byte-range, memory location)
WriteFile(FID, byte-range, memory location)
CloseFile(FID)
PositionPointer(FID,position for pointer)
```

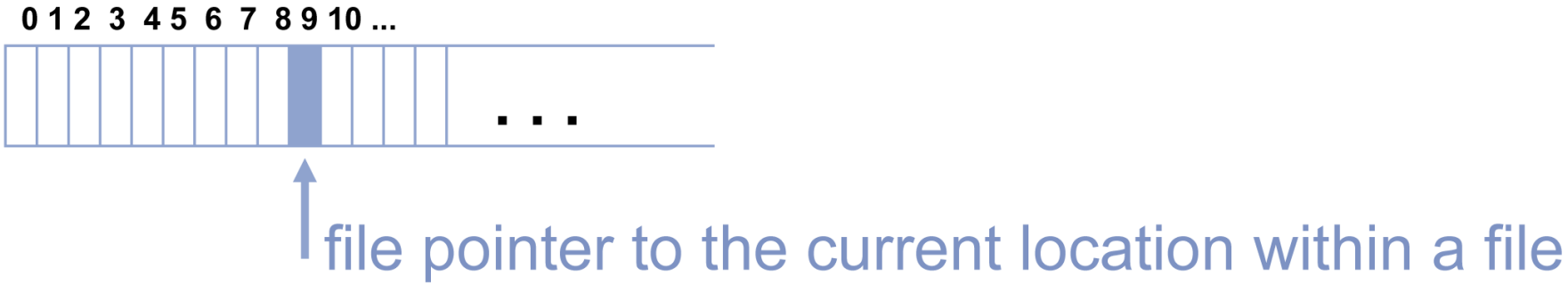
Remark:

`memory location` is the data area within AS of the calling process (e.g. within heap or stack)

Plain File

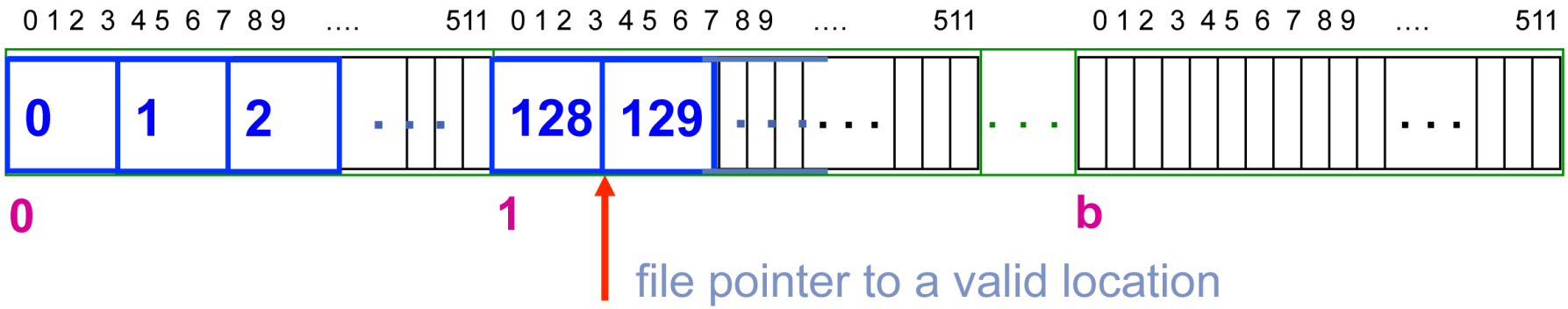
Definition:

A plain file is a sequence of bytes (gaps are possible). Typically located on a disk



- Characteristic: You can randomly access any byte within an unstructured file if you have positioned its file pointer appropriately.
- Problem: Disks cannot access bytes; only blocks.
- Solution: Buffer file blocks (classical method) or entire files (memory mapped files) within main memory

Structured File



Records = logical entities tightly coupled to a specific application, e.g. record of an employee

Employee-file might contain all relevant information, e.g.

- employee number, family name, . . . ,
- employee position, department number,
- passport number, birth date, salary, etc.

Records of equal size or not (then additional length field is needed)

Records with a special key field ($\Rightarrow$ some ordering within a file)

Example: File Operation I

Usage of the following program: `$ copyfile abc xyz`

```
#include <sys/types.h> /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char *argv[]);  /* ANSI prototype */

#define BUF_SIZE 4096 /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700 /* protection bits for output file */

int main(int argc, char *argv[])
{
  int in_fd, out_fd, rd_count, wt_count;
  char buffer[BUF_SIZE];

  if (argc !=3) exit(1); /* syntax error if argc is not 3 */

  /* Open the input file and create the output file */
  in_fd = open(argv[1], O_RDONLY); /* open the source file */
  if (in_fd < 0) exit(2); /* if it cannot be opened, exit */
```

Example: File Operation II

```
  out_fd = create(argv[2],OUTPUT_MODE); /* create the destination file */
  if (out_fd < 0) exit (3);  /* if it cannot be created, exit */

  /* Copy loop */
  while (TRUE) {
    rd_count = read(in_fd, buffer, BUF_SIZE); /*read a block of data */
    if (rd_count <= 0) break;  /* if end of file or error, exit loop */
    wt_count = write(out_fd,buffer,rd_count); /* write data */
    if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
  }

  /* Close the files */
  close(in_fd);
  close(out_fd);
  if (rd_count == 0) /* no error on last read */
    exit(0);
  else
    exit(5); /* error on last read */
}
```

Goal of Directories

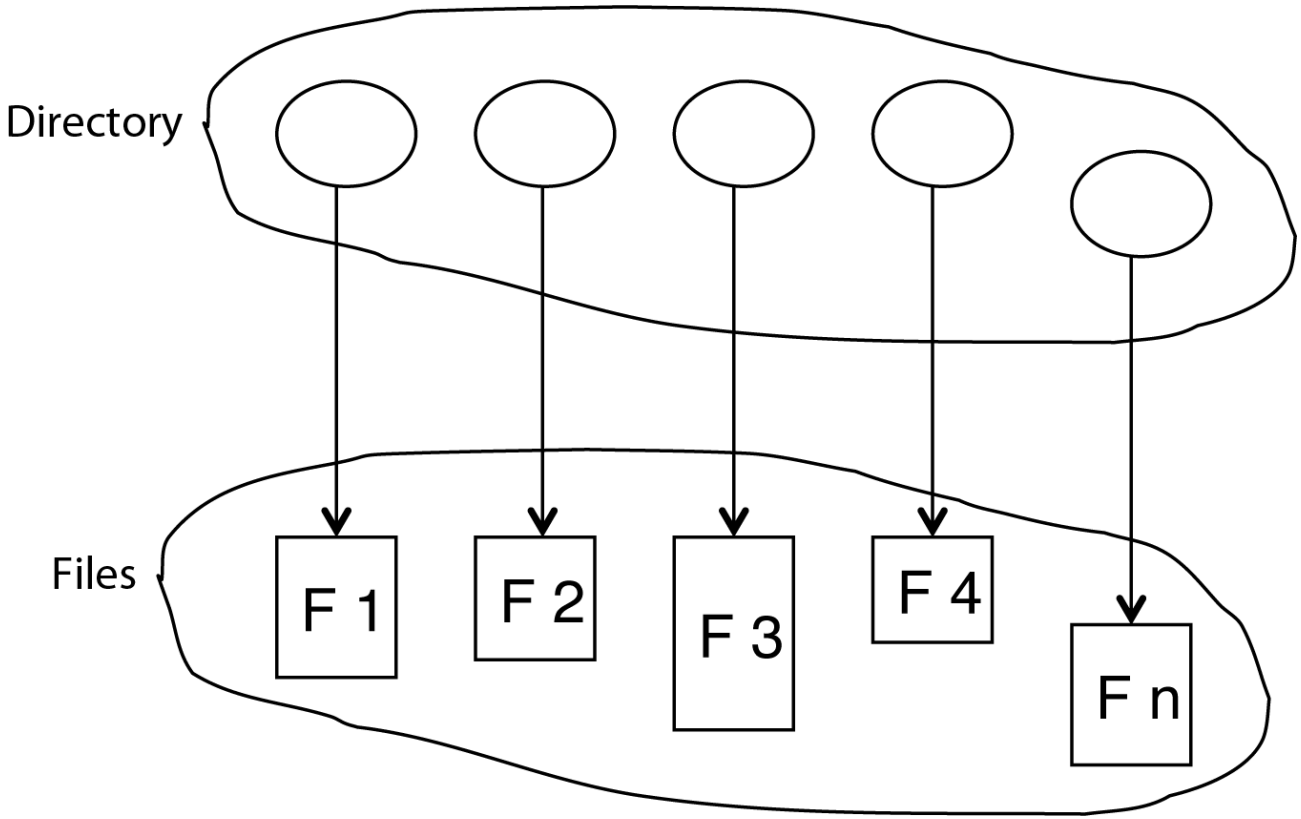
- **Naming:** convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- **Grouping:** logical grouping of files by properties
  - all Java programs
  - all games
  - all programs of a project
  - . . .
- **Efficiency:** fast operations

Operations Performed on a Directory

- Create a file
- Delete a file
- Rename a file
- Traverse the file system
- List a directory
- search for a file

Directory (Folder) (1)

- Directory is a node in a FS owned by an (authorized) subject (e.g. **root**) containing information about (some or all) files of the FS



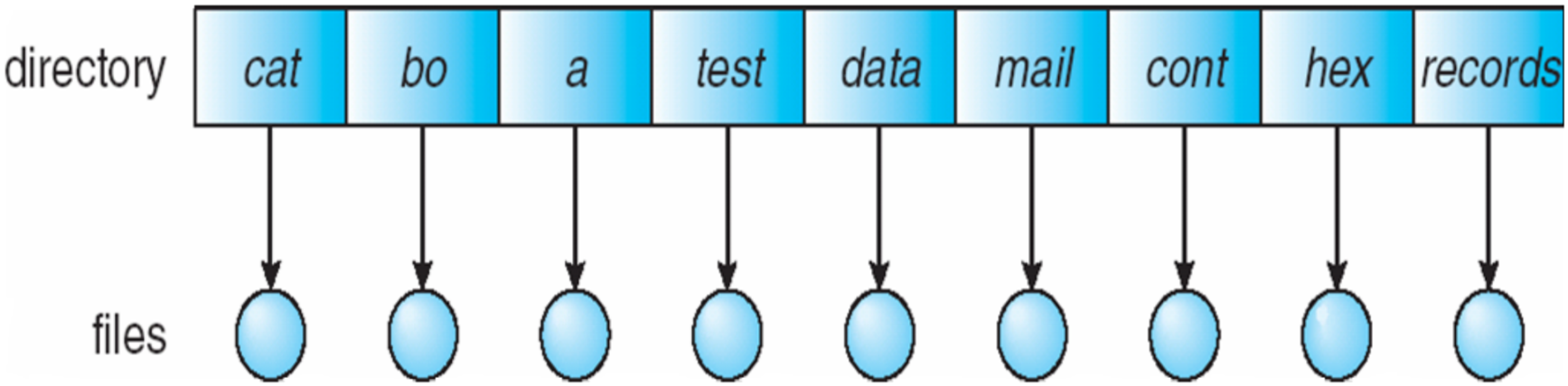
- Both directories and files reside on disk or . . .
- Backups of these both objects are kept on tapes etc.

Directory (Folder) (2)

- The collection of directories and files establish a (hierarchical) FS structure
- In `Linux` there are some special directories e.g.
  - **root**
  - **home**
  - **working**
- Principle structure of a modern FS is a rooted tree
  - Pathnames help to unambiguously identify files
  - Provides mapping between file names $\rightarrow$ files
- Process of file retrieval = navigation

Single-Level Directory

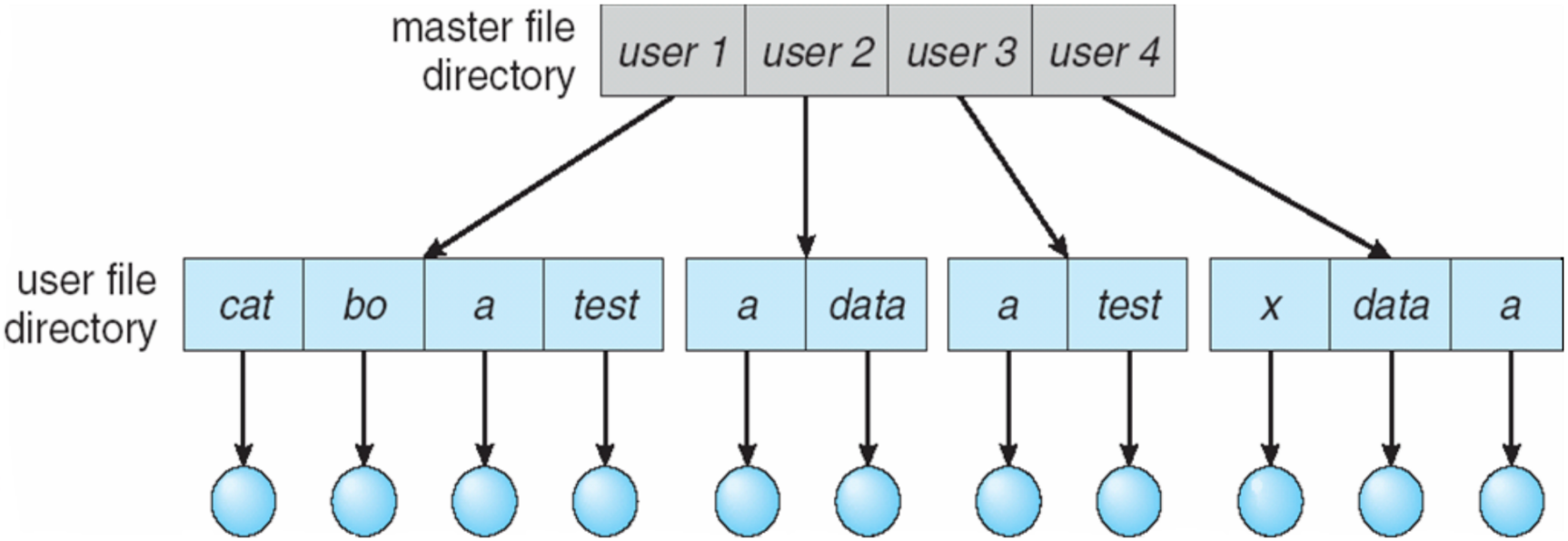
- A single directory for all users



- Naming problem
- Grouping problem

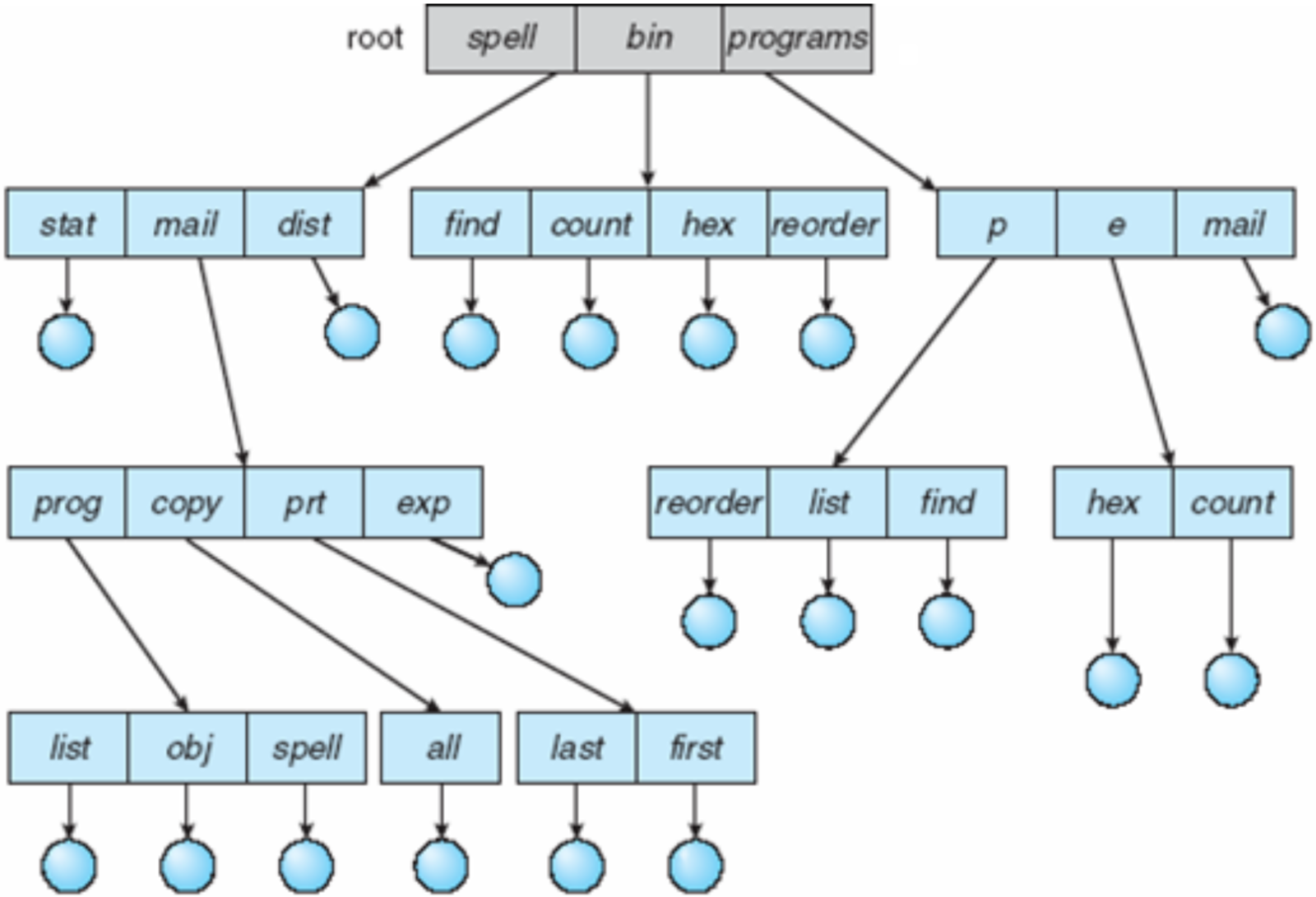
Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



Role of Working Directory

- Absolute pathnames can be tedious, especially when FS is deep
- Idea of a (current or) working directory `cwd`
  - File is referenced via a (hopefully shorter) relative pathname
  - `cwd` belongs to a (process') task's execution environment
  - The initial `cwd` is often called home
- Example:

```
cwd = /home/lief/secret/examinations/SA
lpr ./solution_exam
```

Relative VS. Absolute Pathnames

- Absolute pathname
  - Path from root of FS to file, e.g. `/home/lief/secret/examinations/SA`
- Relative pathname
  - Path from current working directory to file

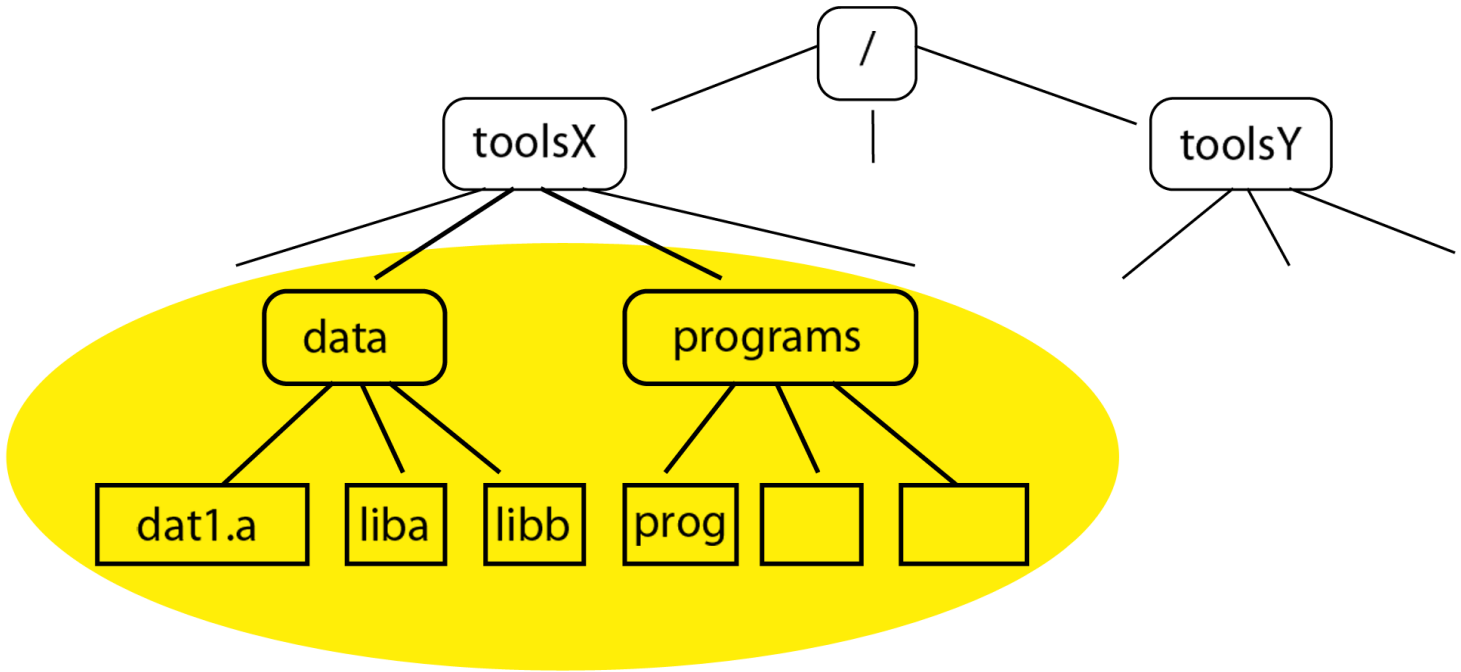
Note:

- `.` refers to current directory
- `..` refers to parent directory

Benefit of Relative Pathname

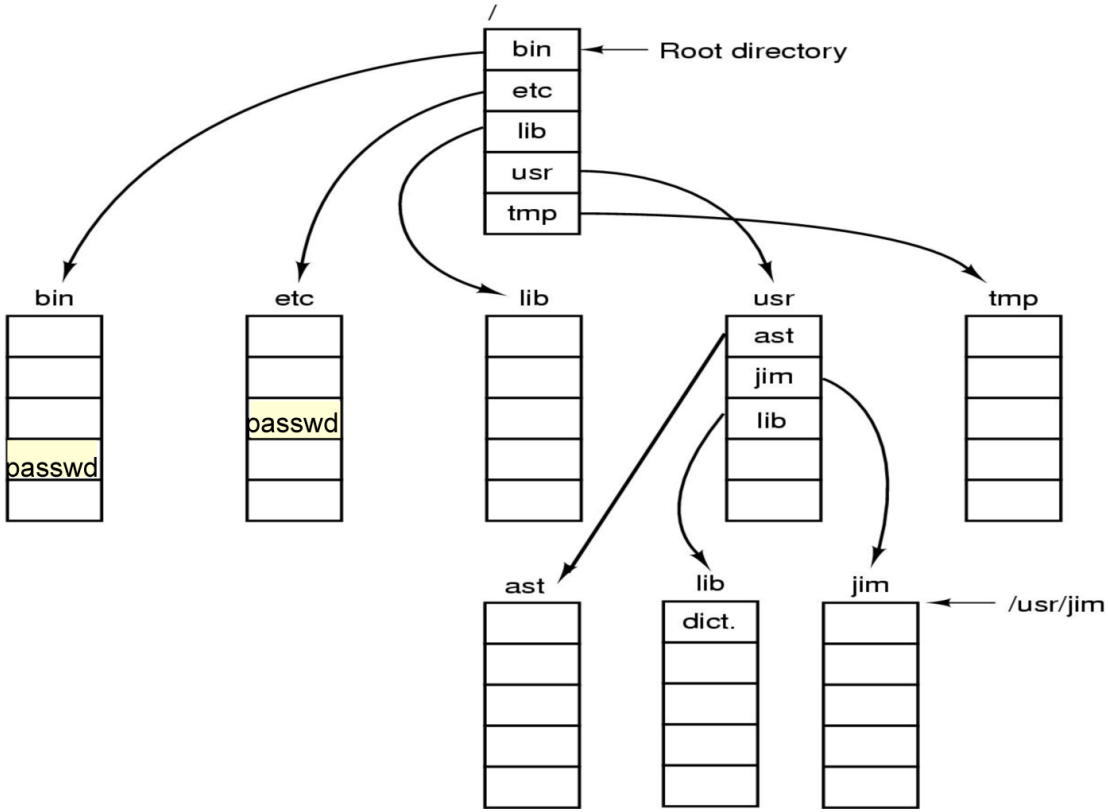
- Improved portability

Example: A program system



If you move the complete program system you must change all absolute pathnames whereas relative pathnames can survive

Hierarchical FS (a la UNIX)



Unambiguous *file names* via **pathnames**, e.g.

`/bin/passwd` $\neq$ `/etc/passwd`

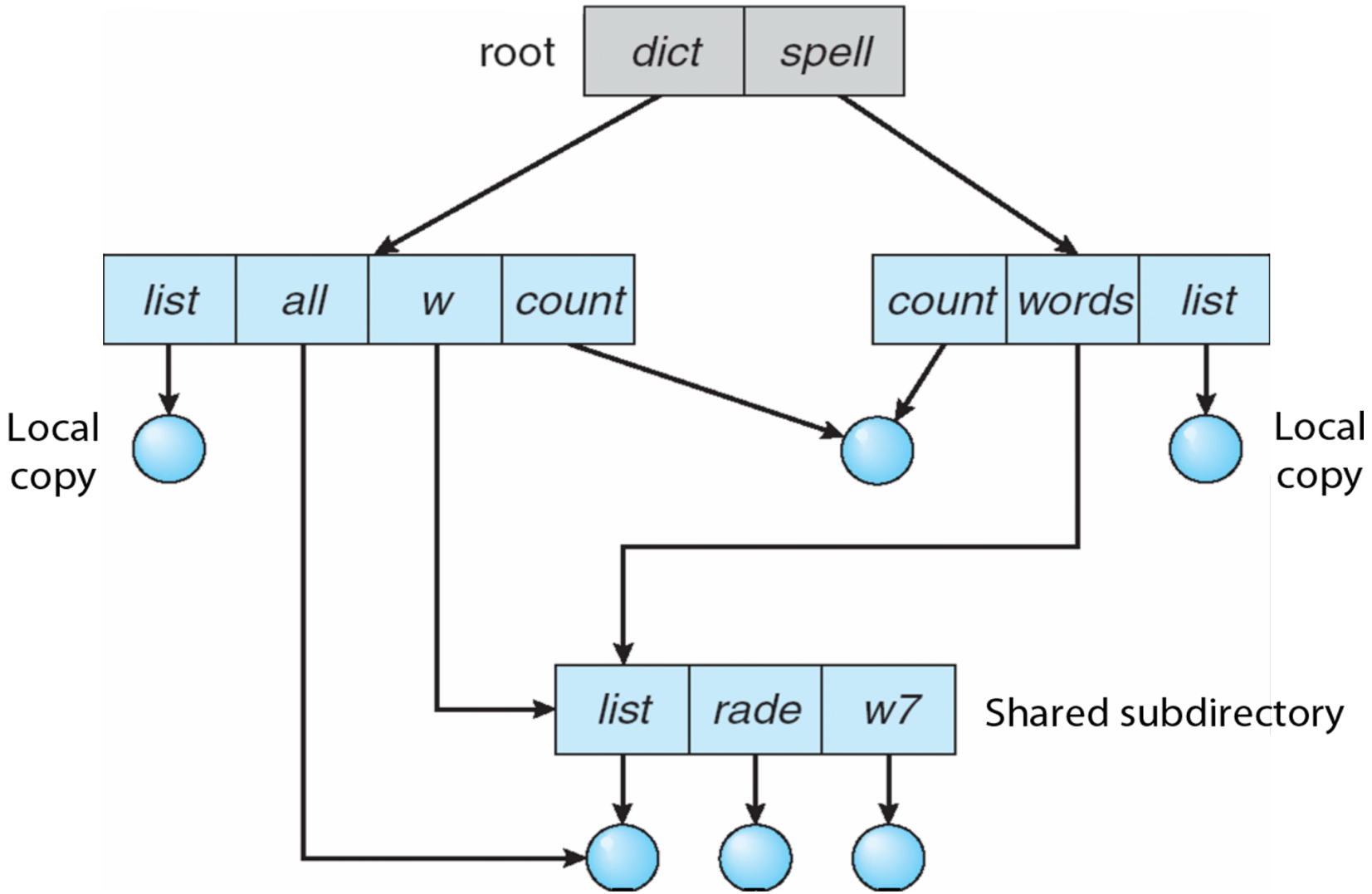
UNIX Directory Operations

- `opendir`
- `closedir`
- `readdir`
- `mkdir`
- `rmdir`

UNIX Link

- Direct access to a file without navigation
- UNIX hard link: `ln filename linkname`
  (another name to the same file = same inode, file is only deleted if last hardlink has been deleted, i.e. if refcount in inode = 0); invalid links are not possible
- Symbolic link: `ln -s filename linkname`
  (a new file `linkname` with a link to a file with name `filename`, whose file might be currently not mounted or not even exist.)

Acyclic-Graph FS Structure



File Sharing

- In multi-user systems, files can be shared among multiple users
- Three issues
  - *Efficiently access to the same file?*
  - *How to determine access rights?*
  - *Management of concurrent accesses?*

Access Rights (1)

- None
  - User might not know of existence of file
  - User is not allowed to read directory containing the file
- Knowledge
  - User can only determine the
    - file existence
    - file ownership

Access Rights (2)

- Execution
  - User can load and execute a program, but cannot copy it
- Reading
  - User can read the file for any purpose, including copying and execution
- Appending
  - User can only add data to a file, but cannot modify or delete any data in the file

Access Rights (3)

- Updating
  - User can modify, delete and add to file's data, including creating the file, rewriting it, removing all or some data from the file
- Changing protection
  - User can change access rights granted to other users
- Deletion
  - User can delete the file

Access Rights (4)

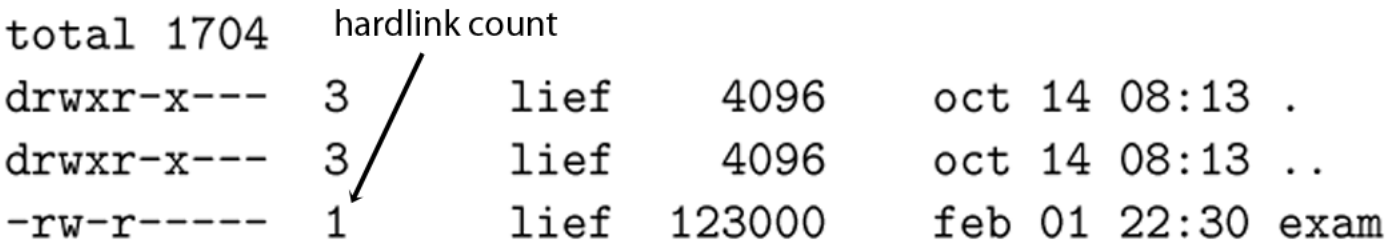
- Owner
  - Has all rights previously listed
  - May grant rights to other users using the following classes of users
    - Specific user
    - User groups
    - All (for public files)

Classical UNIX Access Rights (1)

```
total 1704
drwxr-x---  3      lief    4096     oct 14 08:13 .
drwxr-x---  3      lief    4096     oct 14 08:13 ..
-rw-r-----  1      lief  123000     feb 01 22:30 exam
```

- First letter: file type
  - `d` for directories
  - `-` for regular files
  - `k` for block files
  - ...
- Three user categories:
  - **u**ser, **g**roup and **o**thers

Classical UNIX Access Rights (2)



- Three access rights per category
  - read, write and execute
    - Execute permission for a directory = permission to access files in the directory
    - You must have the read permission to a directory if you want to list its content

Classical UNIX Access Rights (3)

- Shortcomings
  - Three user(subject) categories is not enough
  - In Windows you have a finer granularity concerning access rights per folder and per file, e.g. you can explicitly deny/allow access for a specific user
- UNIX has introduced the concept of ACLs
- An ACL is a list bound to a file `f`, containing all individual subjects & their individual permissions how to access this file `f`

UNIX ACLs (1)

If I want to view the content of the ACL of the file `exam` in my current directory, I can use the following command:

```
bellosa@i30s5:~> getfacl exam
# file: exam
# owner: bellosa
# group: i30staff
user::rwx
group::r--
other::---
```

UNIX ACLs (2)

If I wish to allow another person with an account on the same system to access file `exam`, I use the `setfacl` command, e.g.

```
setfacl -m user:name:permissions file
```

`name` is loginID of the person to which you want to assign access, `permissions` can be one or more of the following: `r, w, x`
`file` is the name of the file

Example:

I want to enable Philipp with an assumed loginID `pkupfer` to `read & modify`, but not to execute my file `exam`: I would use:

```
setfacl -u user:pkupfer:rw exam
```
[1]

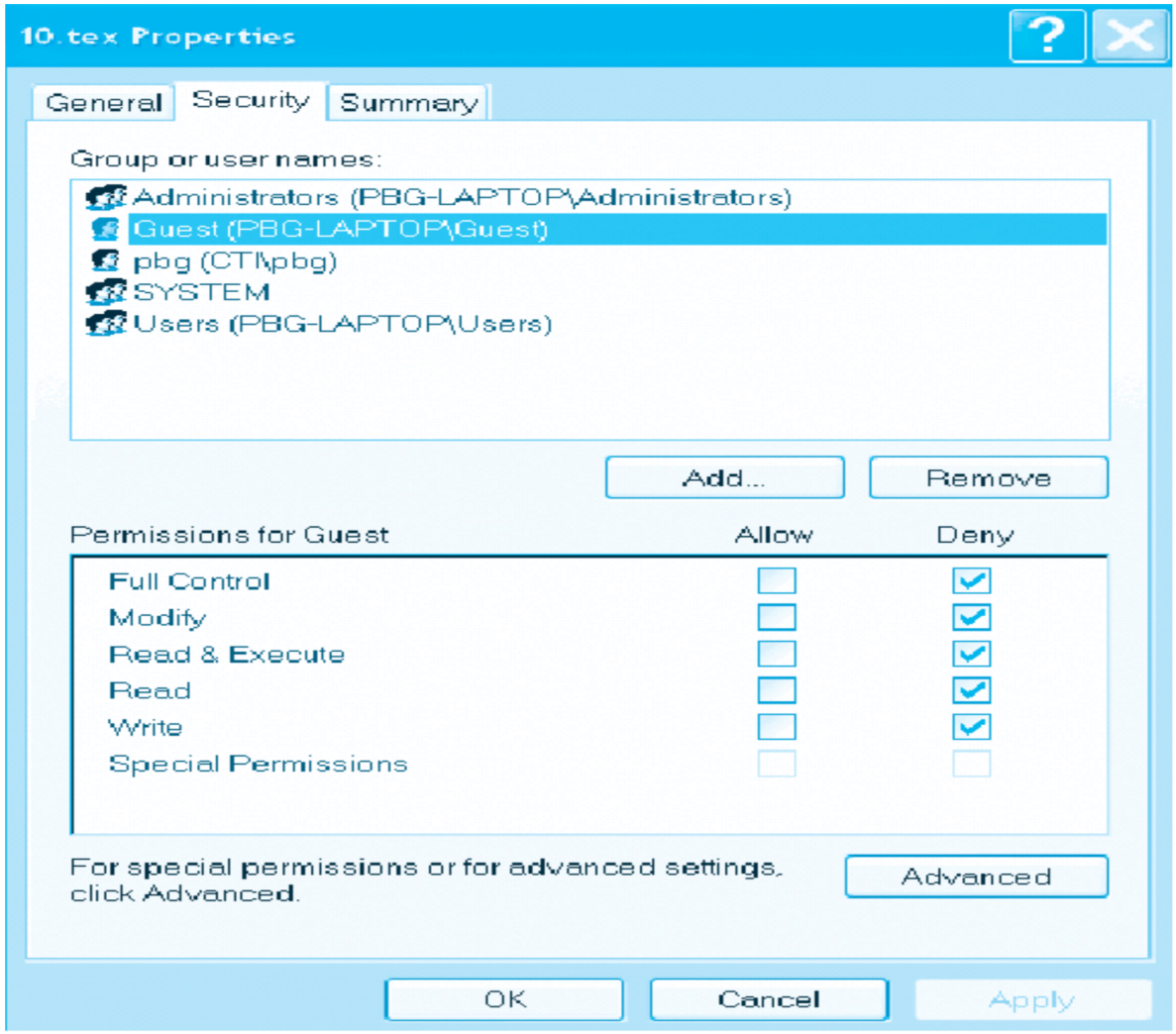
[1] Note: you always have to use the complete permission triple

UNIX ACLs (3)

Now when I type again `getacl exam`, the following information is displayed:

```
bellosa@i30s5:~> getfacl exam
# file: exam
# owner: bellosa
# group: i30staff
user::rwx
user:pkupfer:rw-
group::r--
mask::rw-
other::---
```

Windows Access-control List Management



Concurrent Access to Files

- Some OSes provide mechanisms for users to manage concurrent access to files
  - Examples: `flock()`, `fcntl()` system calls
- Applications can lock
  - entire file for updating file
  - individual records for updating
- Exclusive or shared:
  - **Exclusive** – Writer lock
  - **Shared** – Multiple readers allowed
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do