Systems and Networking – Unit I

B.Sc. in Applied Computer Science and Artificial Intelligence

2021-2022

Gabriele Tolomei

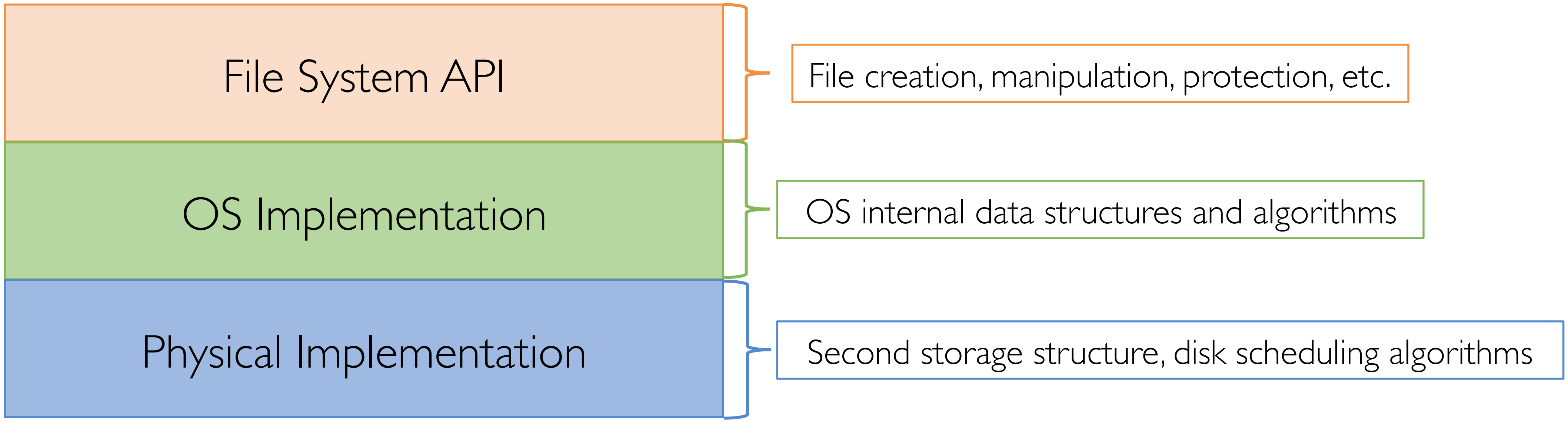
Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA UNIVERSITÀ DI ROMA

File System's Logical View



File System's Logical View

File System API	File creation, manipulation, protection, etc.
OS Implementation	OS internal data structures and algorithms
Physical Implementation	Second storage structure, disk scheduling algorithms

Part VI: File System

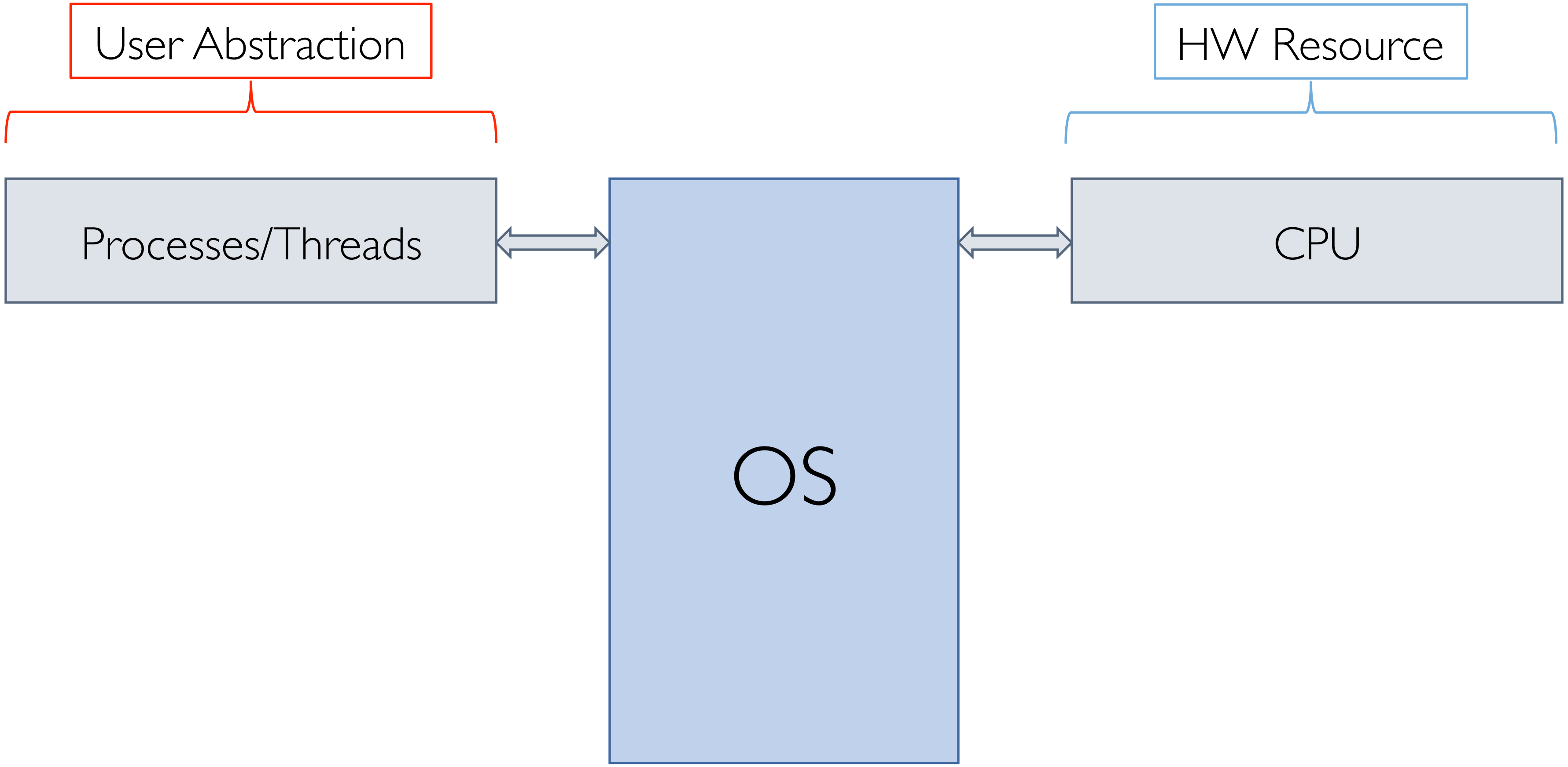
Abstraction

User Abstraction

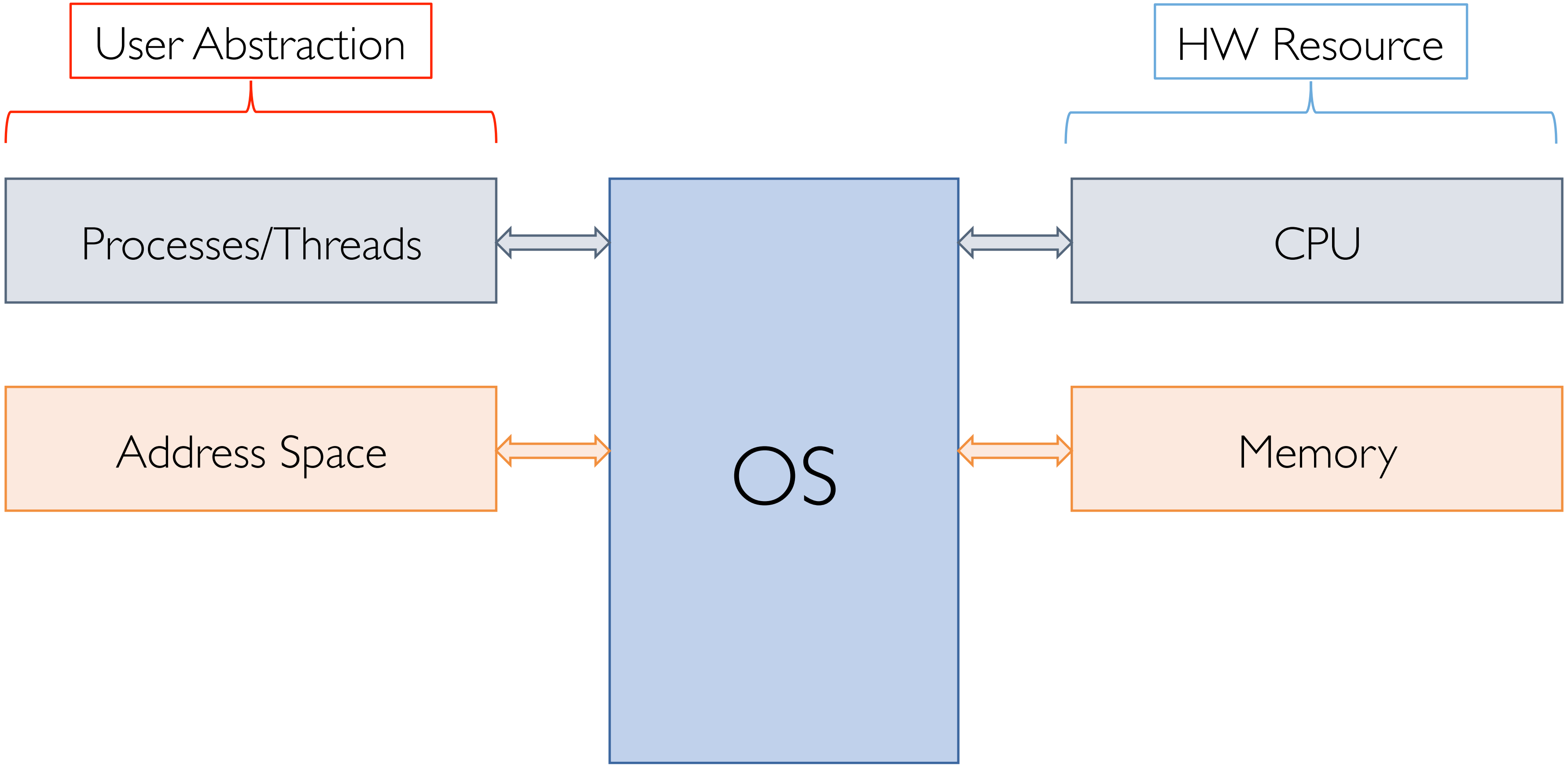
HW Resource

OS

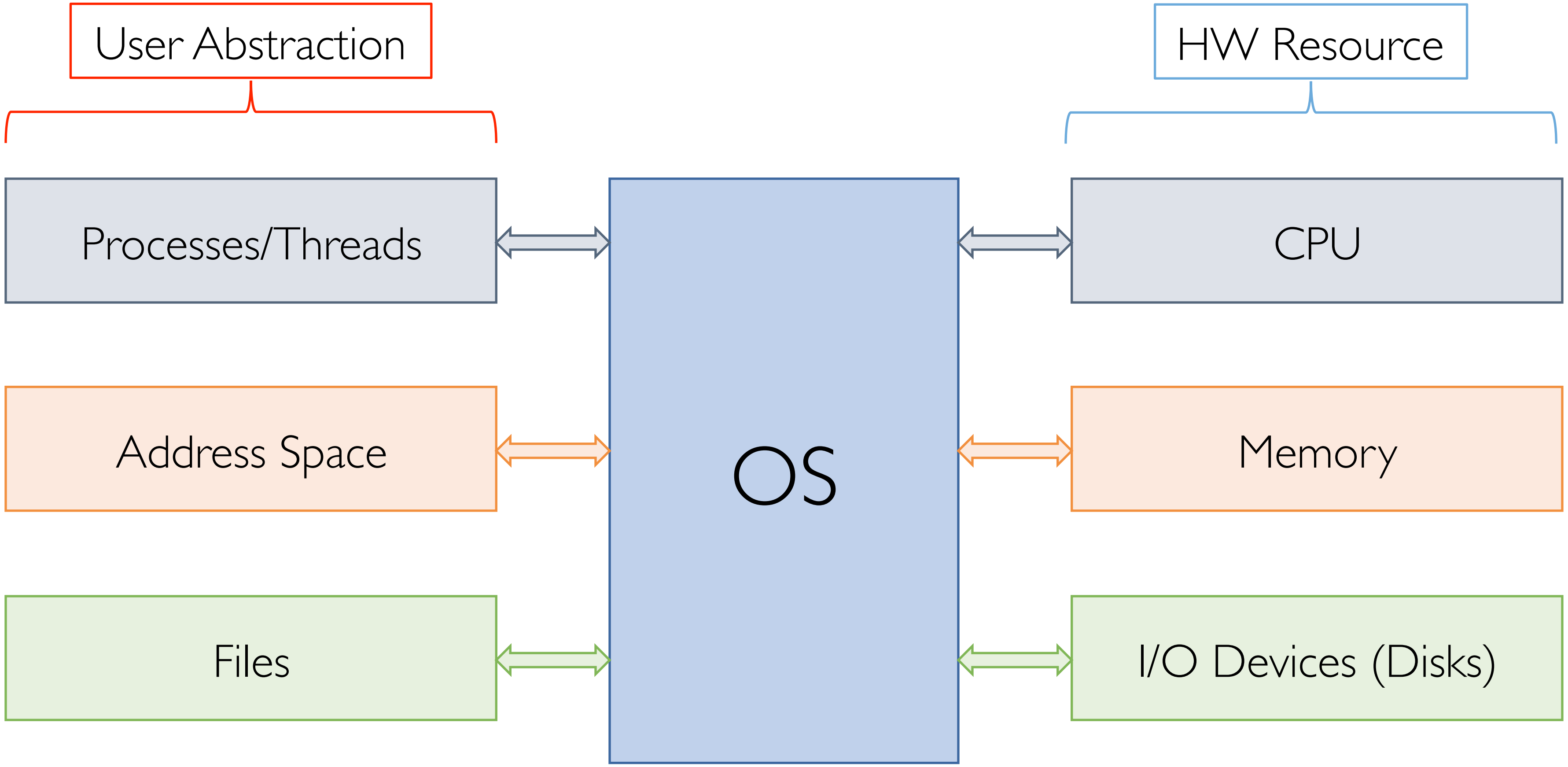
Abstraction



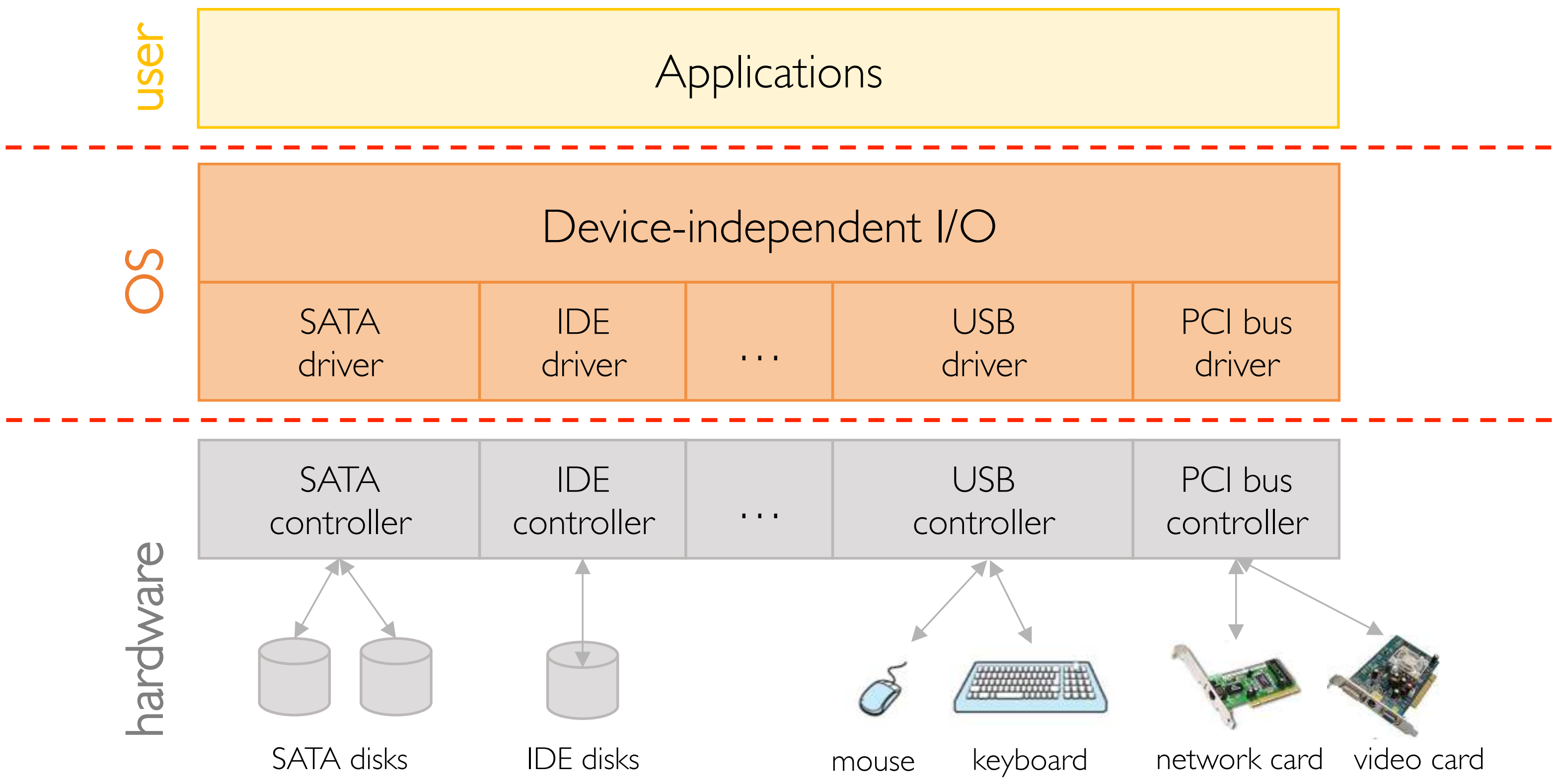
Abstraction



Abstraction



File System Abstraction



User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly
- Size → May want to store huge amounts of data

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly
- Size → May want to store huge amounts of data
- Sharing/Protection → Data can be shared where appropriate

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly
- Size → May want to store huge amounts of data
- Sharing/Protection → Data can be shared where appropriate
- Ease of Use → Data should be easily found, examined, modified, etc.

HW vs. OS Capabilities

- HW provides:
  - Persistence: Disks are non-volatile storage devices
  - Speed (somewhat): Disks enable direct/random access
  - Size: Disks keep getting bigger (order of TBs on today's laptop)

HW vs. OS Capabilities

- HW provides:
  - Persistence: Disks are non-volatile storage devices
  - Speed (somewhat): Disks enable direct/random access
  - Size: Disks keep getting bigger (order of TBs on today's laptop)
- OS provides:
  - Persistence: Redundancy mechanisms
  - Sharing/Protection: Permissions (e.g., UNIX **`rwx`** privileges)
  - Ease of Use: named files, directories, search tools (e.g., Spotlight in macOS)

What's a File?

- The abstraction used by the OS to refer to the logical unit of data on a storage device
  - Named collection of related information (bytes!) stored on secondary memory

What's a File?

- The abstraction used by the OS to refer to the logical unit of data on a storage device
  - Named collection of related information (bytes!) stored on secondary memory
- Files are mapped by the OS onto physical storage devices (e.g., disks)
  - Such devices are non-volatile (their content persist across reboots)

What's a File?

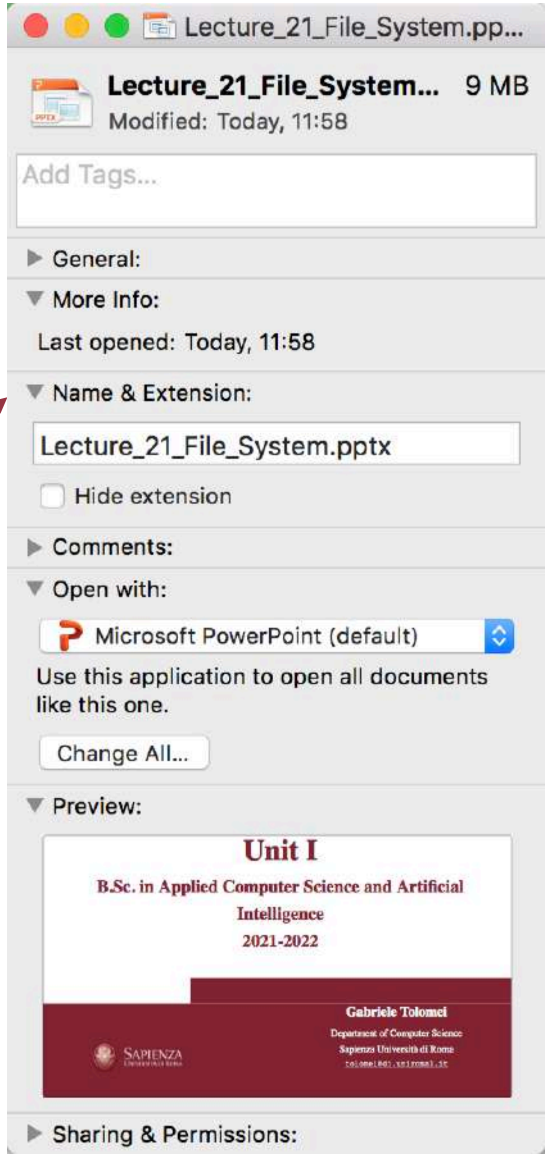
- The abstraction used by the OS to refer to the logical unit of data on a storage device
  - Named collection of related information (bytes!) stored on secondary memory
- Files are mapped by the OS onto physical storage devices (e.g., disks)
  - Such devices are non-volatile (their content persist across reboots)
- Files can contain programs (source, binary) or data
  - Examples: `main.cpp`, `test.exe`, `doc.txt`

Files: Attributes (Metadata)

- Different OSs keep track of different file attributes
- Examples:
  - Name: human-friendly identifier
  - Identifier: how the OS actually identifies the file (e.g., **`inode`** number)
  - Type: text, executable, other binary, etc.
  - Location (on the hard drive)
  - Size
  - Protection
  - Time & Date
  - User ID

Files: Attributes (Example)

All the information displayed are metadata associated with *this* file



Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata

Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata
- Data operations:
  - **`create()`**, **`open()`**, **`read()`**, **`write()`**, **`seek()`**, **`close()`**, **`delete()`**

Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata
- Data operations:
  - **`create()`**, **`open()`**, **`read()`**, **`write()`**, **`seek()`**, **`close()`**, **`delete()`**
- Metadata operations:
  - change owner/permissions (**`chown`**/**`chmod`**)
  - make symbolic links (**`ln`**)
  - etc.

Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata
- Data operations:
  - **`create()`**, **`open()`**, **`read()`**, **`write()`**, **`seek()`**, **`close()`**, **`delete()`**
- Metadata operations:
  - change owner/permissions (**`chown`**/**`chmod`**)
  - make symbolic links (**`ln`**)
  - etc.

Those are all system calls typically wrapped within a user library

OS (Kernel) File Data Structures

Global Open File Table

- shared by all the processes with an open file
- one entry for each open file
- multiple processes may have the same file open (counter)
- file attributes (ownership, protection, etc.)
- location of each file on disk
- pointers to location of each file on disk

OS (Kernel) File Data Structures

Global Open File Table

- shared by all the processes with an open file
- one entry for each open file
- multiple processes may have the same file open (counter)
- file attributes (ownership, protection, etc.)
- location of each file on disk
- pointers to location of each file on disk

Local Per-Process File Table

- one table for each process
- for each open file of this process:
  - pointer to the entry in the global table
  - current position in the file (offset)
  - open mode (r, w, r/w)

Files Operations: `create(filename)`

- Allocate disk space, also checking disk quotas and permissions

Files Operations: `create(filename)`

- Allocate disk space, also checking disk quotas and permissions
- Create a file descriptor for the file including:
  - **`filename`**
  - location on disk
  - other attributes

Files Operations: `create(filename)`

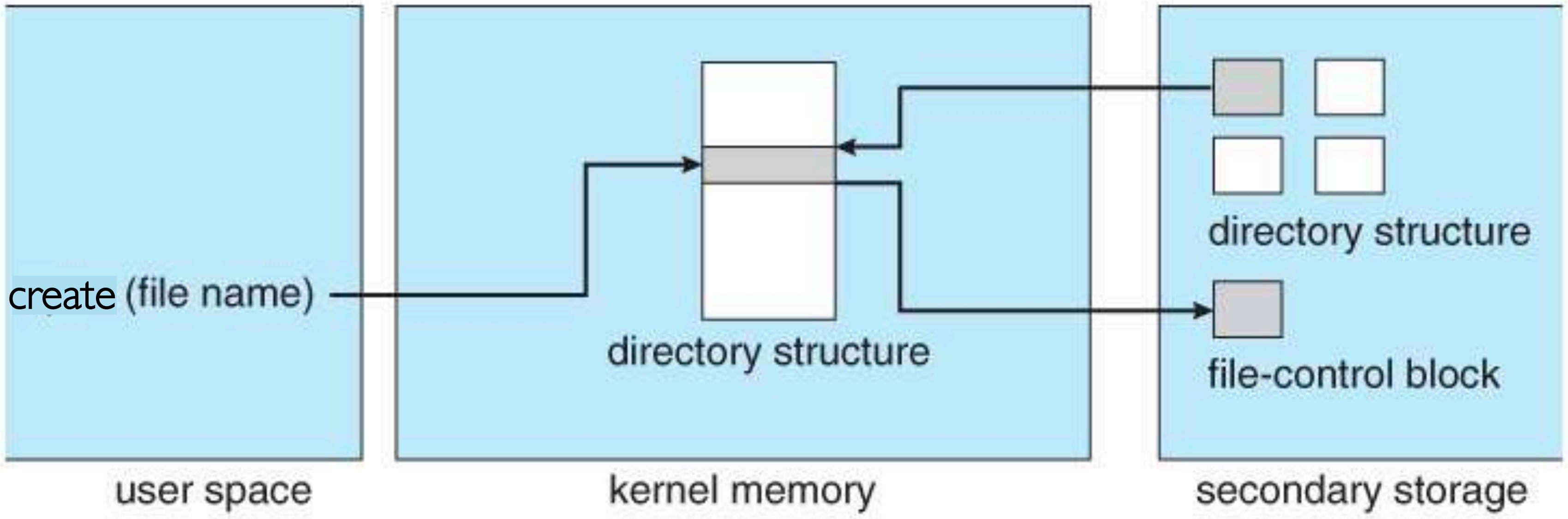
- Allocate disk space, also checking disk quotas and permissions
- Create a file descriptor for the file including:
  - `filename`
  - location on disk
  - other attributes
- Add the file descriptor to the directory that contains the file

Files Operations: `create(filename)`

- Allocate disk space, also checking disk quotas and permissions
- Create a file descriptor for the file including:
  - **`filename`**
  - location on disk
  - other attributes
- Add the file descriptor to the directory that contains the file

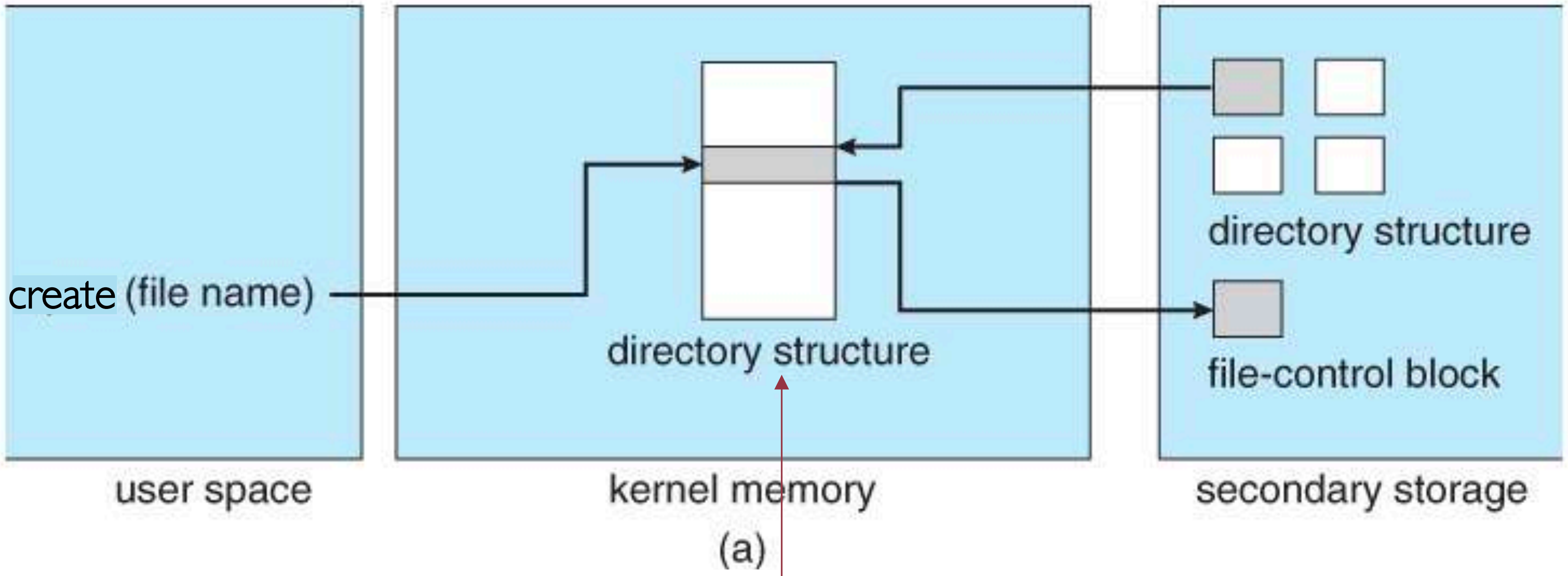
We will talk about file descriptors and directories in a few slides…

Files Operations: `create(filename)`



(a)

Files Operations: `create(filename)`



Cached from on-disk directory structure

Files Operations: `create(filename)`

- Optional file attribute: file type (MS Word, executable, etc.):
  - better error detection
  - specialized default operations (e.g., double-click triggers the right application)
  - storage layout optimization
  - more complex filesystem and OS
  - less flexibility (what if we want to change the file type)
- In UNIX no file type, Windows and Mac opt for user-friendliness

Files Operations: `delete(filename)`

- Find the directory containing the file
- Free the disk blocks used by the file
- Remove the file descriptor from the directory
- Behavior dependent on hard links (more on this later)

Files Operations: `open(filename,mode)`

- Returns the **`fileID`** the OS associated with that **`filename`**

Files Operations: `open(filename,mode)`

- Returns the `fileID` the OS associated with that `filename`
- Check the global open file table if the file is already open by another process, if not:
  - Find the file and copy the file descriptor into the global open file table

Files Operations: `open(filename,mode)`

- Returns the **`fileID`** the OS associated with that **`filename`**
- Check the global open file table if the file is already open by another process, if not:
  - Find the file and copy the file descriptor into the global open file table
- Check protection of the file against the mode, if not ok abort

Files Operations: `open(filename,mode)`

- Returns the **`fileID`** the OS associated with that **`filename`**
- Check the global open file table if the file is already open by another process, if not:
  - Find the file and copy the file descriptor into the global open file table
- Check protection of the file against the mode, if not ok abort
- Increment the open count

Files Operations: `open(filename,mode)`

- Returns the **`fileID`** the OS associated with that **`filename`**
- Check the global open file table if the file is already open by another process, if not:
  - Find the file and copy the file descriptor into the global open file table
- Check protection of the file against the mode, if not ok abort
- Increment the open count
- Create an entry in the process' file table pointing to the entry of the global table, and initialize the file pointer to the beginning of the file

Files Operations: `close(fileID)`

- Remove the entry for the file in the process' file table

Files Operations: `close(fileID)`

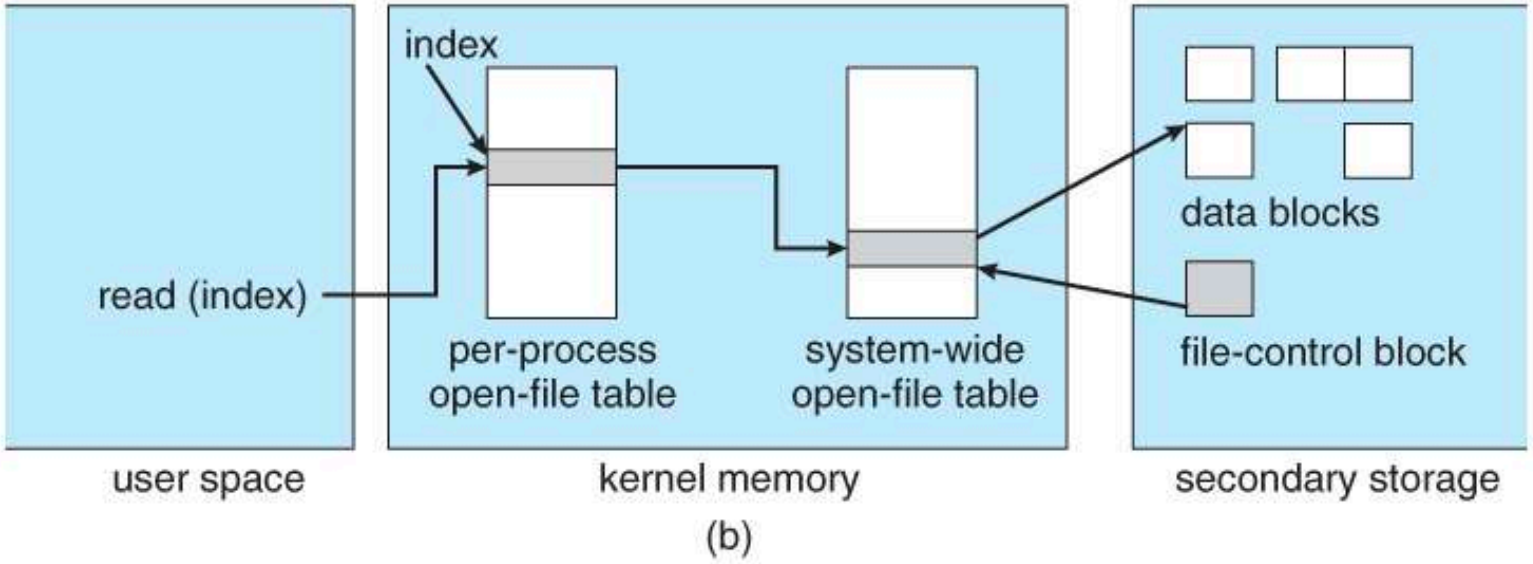
- Remove the entry for the file in the process' file table
- Decrement the open count of this file on the global file table

Files Operations: `close(fileID)`

- Remove the entry for the file in the process' file table
- Decrement the open count of this file on the global file table
- If the open count gets to 0 → no process has this file open
  - The corresponding entry in the global table can be safely removed

Files Operations: `read(fileID)`

- Read a file given the index (file descriptor) returned by the **`open`** call
- In order for a file to be read, it must therefore be open!



Files Operations: Read

- 2 possible ways of reading a file:
  - random/direct access
  - sequential access

Files Operations: Read

- 2 possible ways of reading a file:
  - random/direct access
  - sequential access
- random/direct access → hard drives (or main memory)
  - Can access to a specific disk block (memory address)

Files Operations: Read

- 2 possible ways of reading a file:
  - random/direct access
  - sequential access
- random/direct access → hard drives (or main memory)
  - Can access to a specific disk block (memory address)
- sequential access → devices which do not support direct access (e.g., tape drives)
  - Need to go all the way through the desired position

Files Operations: Read (Random Access)

- **`read(fileID, from, size, bufAddress)`**
  - OS reads **`size`** bytes from file position **`from`** into **`bufAddress`**

Files Operations: Read (Random Access)

- **`read(fileID, from, size, bufAddress)`**
  - OS reads **`size`** bytes from file position **`from`** into **`bufAddress`**

```
for (i = from; i < from + size; ++i) {
    bufAddress[i - from] = fileID[i];
}
```

Files Operations: Read (Sequential Access)

- **`read(fileID, size, bufAddress)`**
  - OS reads **`size`** bytes from file current position (**`fp`**) into **`bufAddress`**, and updates the file position accordingly

Files Operations: Read (Sequential Access)

- **`read(fileID, size, bufAddress)`**
  - OS reads **`size`** bytes from file current position (**`fp`**) into **`bufAddress`**, and updates the file position accordingly

```
for (i = 0; i < size; ++i) {

    bufAddress[i] = fileID[fp + i];

}
fp += size;
```

Files Operations: Other Operations

- **`write`** → similar to **`read`** but copies from buffer to the file

Files Operations: Other Operations

- **`write`** → similar to **`read`** but copies from buffer to the file
- **`seek`** → just updates the file position (no need to actual I/O)

Files Operations: Other Operations

- `write` → similar to `read` but copies from buffer to the file
- `seek` → just updates the file position (no need to actual I/O)
- `mmap` → Memory mapping a file
  - Map (a part of) the virtual address space to a file
  - Read from/write to that portion of memory implies OS reads from/writes to the corresponding location in the file (stored on disk)
  - File accesses are greatly simplified (no read/write system calls are necessary)
  - No need to copy from/to the buffer in kernel space at each operation

File Access Methods: Programmer's Perspective

- Sequential → Data is accessed in order, one byte/record at a time
  - Example: compiler reading source file

File Access Methods: Programmer's Perspective

- Sequential → Data is accessed in order, one byte/record at a time
  - Example: compiler reading source file
- Direct/Random → Data is accessed at a specific position
  - Example: text editor "goto line" feature

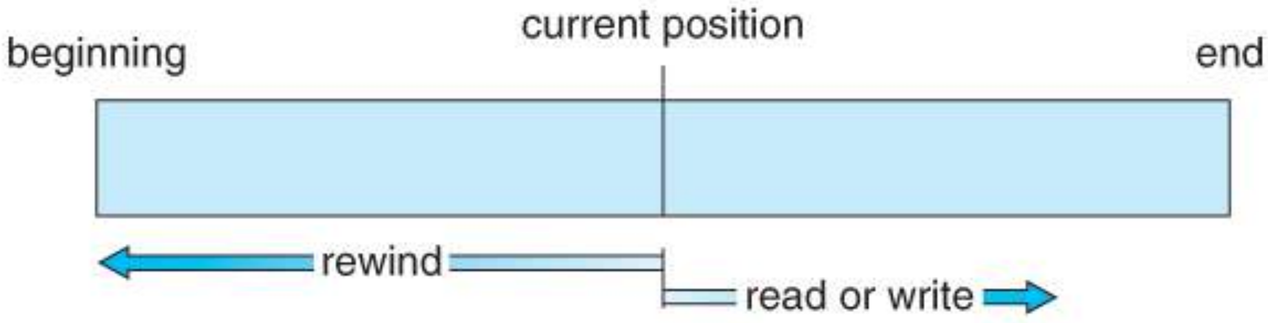
File Access Methods: Programmer's Perspective

- Sequential → Data is accessed in order, one byte/record at a time
  - Example: compiler reading source file
- Direct/Random → Data is accessed at a specific position
  - Example: text editor "goto line" feature
- Keyed/Indexed → Data is accessed based on a key
  - Example: database search

File Access Methods: OS's Perspective

Sequential

Keep a pointer to the next byte in the file, and update the pointer on each read/write operation



File Access Methods: OS's Perspective

Direct/Random

Address any block of data directly given its offset within the file

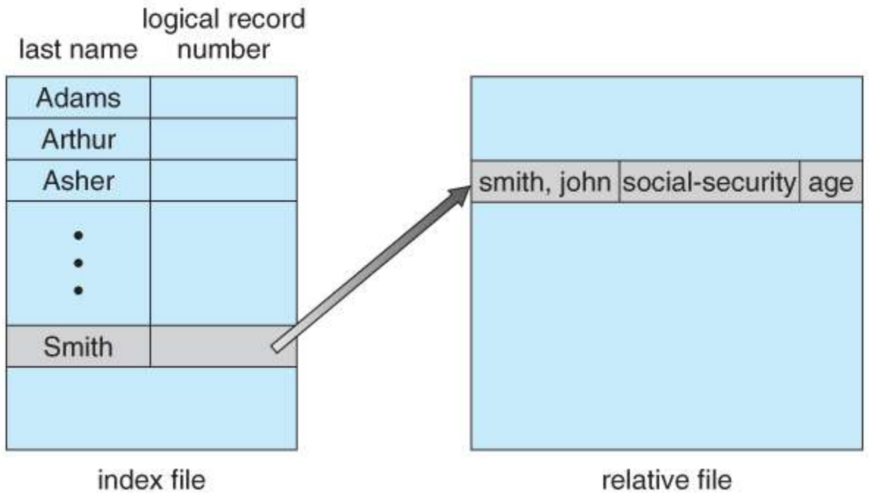
sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

simulating sequential access using direct access

File Access Methods: OS's Perspective

<u>Keyed/Indexed</u>

Address any block of data directly given a key



implemented on top of direct access

Naming and Directories

- Need a method to getting back files that are located on disk
- OS uses unique numbers to identify files
- Users would rather use human-friendly names to refer to files
- Directory → OS data structure which maps file names to descriptors

Naming and Directories

- Need a method to getting back files that are located on disk
- OS uses unique numbers to identify files
- Users would rather use human-friendly names to refer to files
- Directory → OS data structure which maps file names to descriptors

Stored on disk and cached in OS kernel memory

Directory: Overview

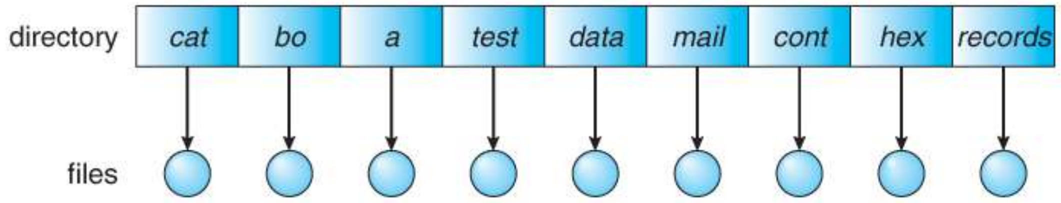
- Directory operations to be supported include:
  - Search for a file
  - Create a file (add it to the directory)
  - Delete a file (erase it from the directory)
  - List a directory (possibly ordered in different ways)
  - Rename a file (may change sorting order)
  - Traverse the file system

Directory: Naming Strategies

- Single-Level Directory
  - One name space for the entire disk
  - Every filename must be unique
  - Use a special area of disk to hold the directory
  - Directory contains (name, index) pairs
  - If one user uses a name, no one else can
  - Used by early personal computers because their disks were very small

Directory: Naming Strategies

- Single-Level Directory
  - One name space for the entire disk
  - Every filename must be unique
  - Use a special area of disk to hold the directory
  - Directory contains (name, index) pairs
  - If one user uses a name, no one else can
  - Used by early personal computers because their disks were very small

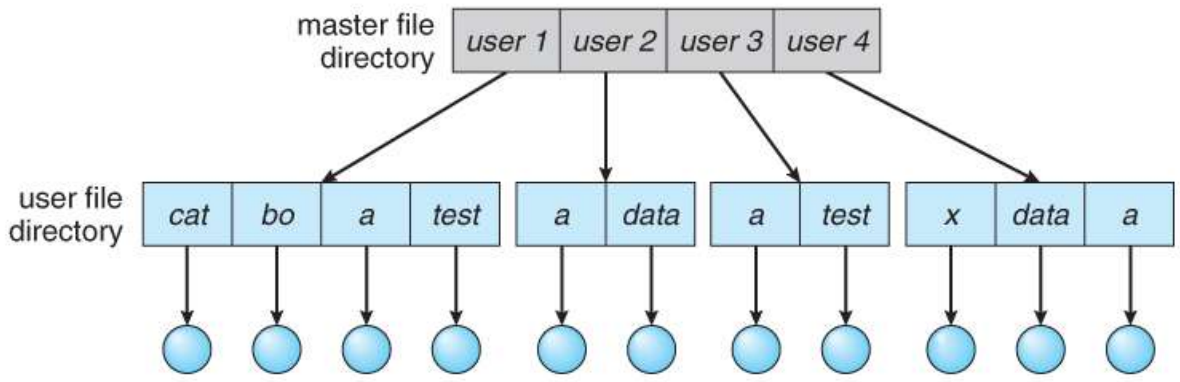


Directory: Naming Strategies

- Two-Level Directory
  - Each user gets their own directory space
  - File names only need to be unique within a given user's directory
  - A master file directory is used to keep track of each users directory
  - A separate directory is generally needed for system (executable) files

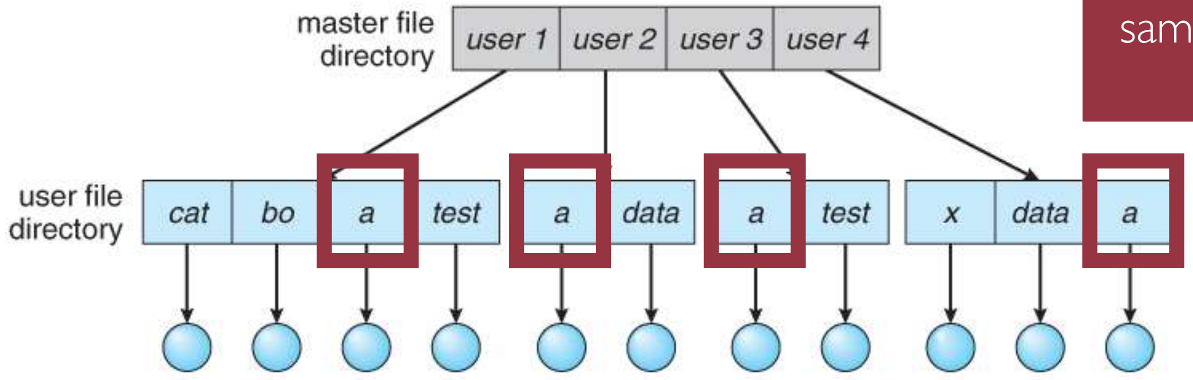
Directory: Naming Strategies

- Two-Level Directory
  - Each user gets their own directory space
  - File names only need to be unique within a given user's directory
  - A master file directory is used to keep track of each users directory
  - A separate directory is generally needed for system (executable) files



Directory: Naming Strategies

- Two-Level Directory
  - Each user gets their own directory space
  - File names only need to be unique within a given user's directory
  - A master file directory is used to keep track of each users directory
  - A separate directory is generally needed for system (executable) files

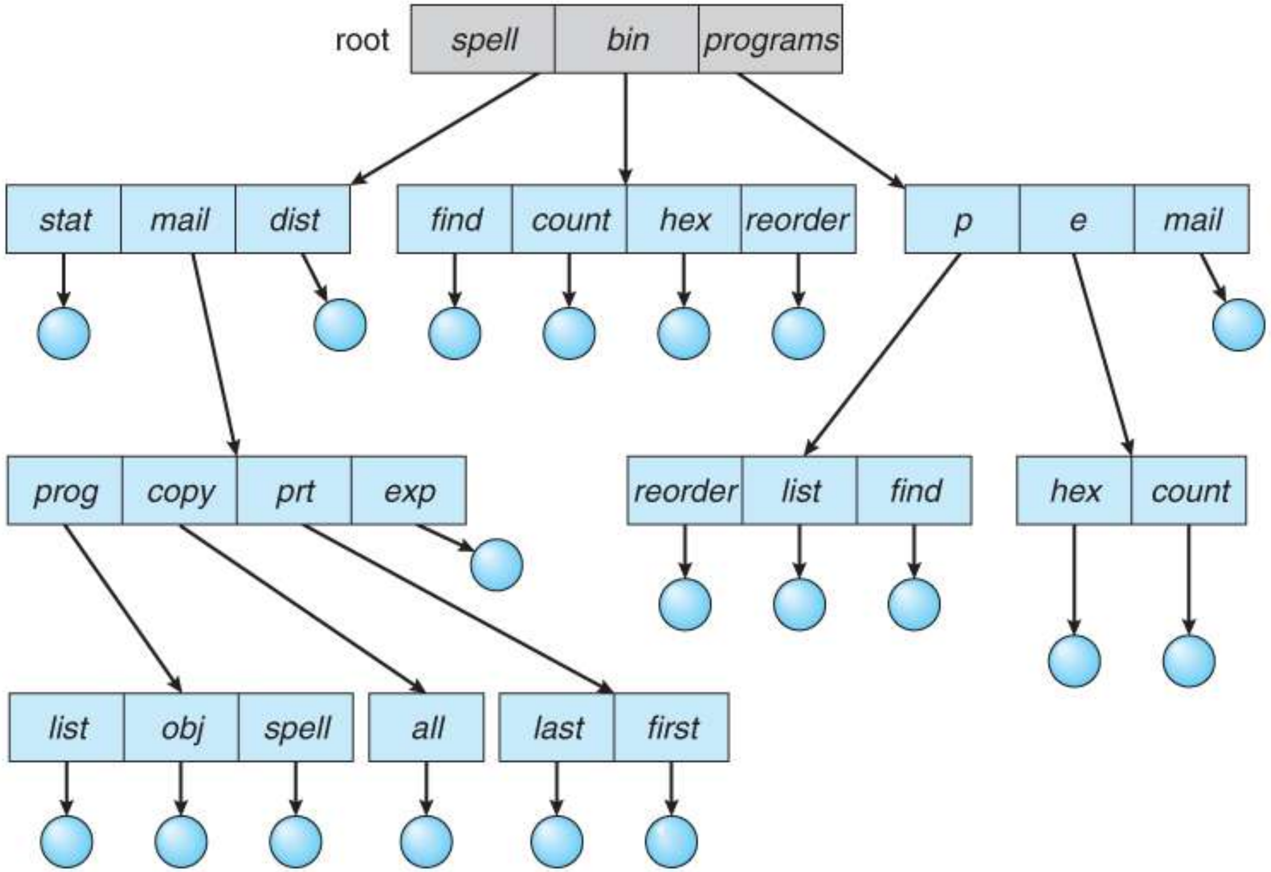


same user directories (possibly same filenames too!) yet underneath different master directories

Directory: Naming Strategies

- Multi-Level (Tree-based) Directory
  - An obvious extension to the two-tiered directory structure
  - Each user/process has the concept of a current directory from which all (relative) searches take place
  - Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory)
  - Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories
  - Used by most modern OSs (UNIX/Linux, Windows, and macOS)

Directory Tree



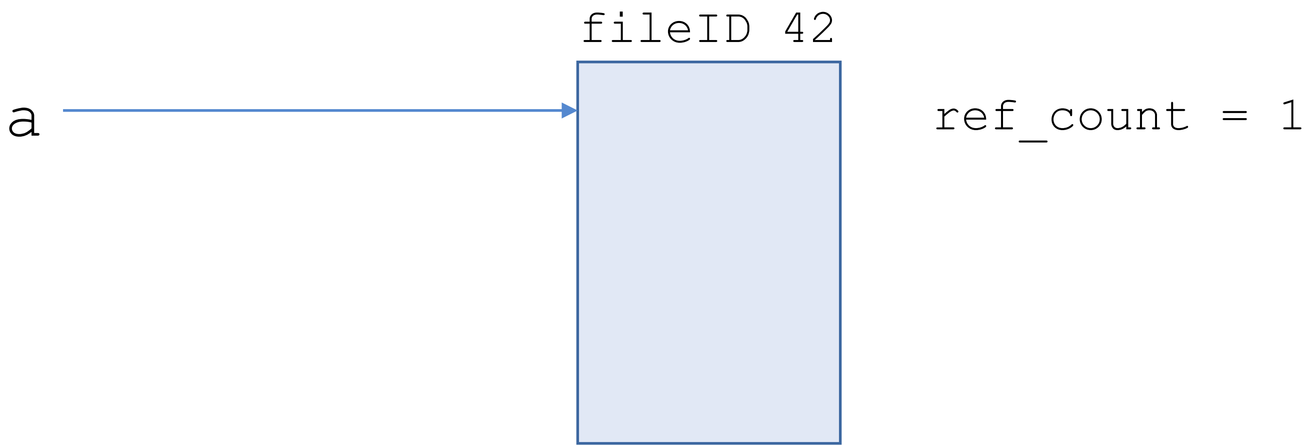
Referential Naming

- Sharing files between different user's directory trees may be complicated

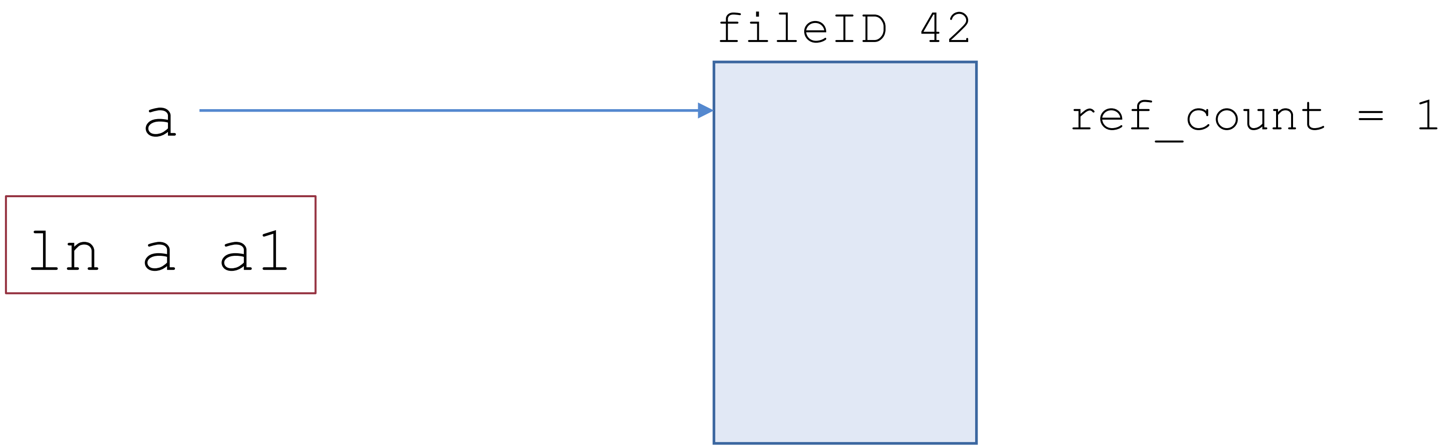
Referential Naming

- Sharing files between different user's directory trees may be complicated
- UNIX provides 2 types of links via the **`ln`** command:
  - hard link → multiple directory entries that refer to the same file
  - symbolic link → an alias to the linked file

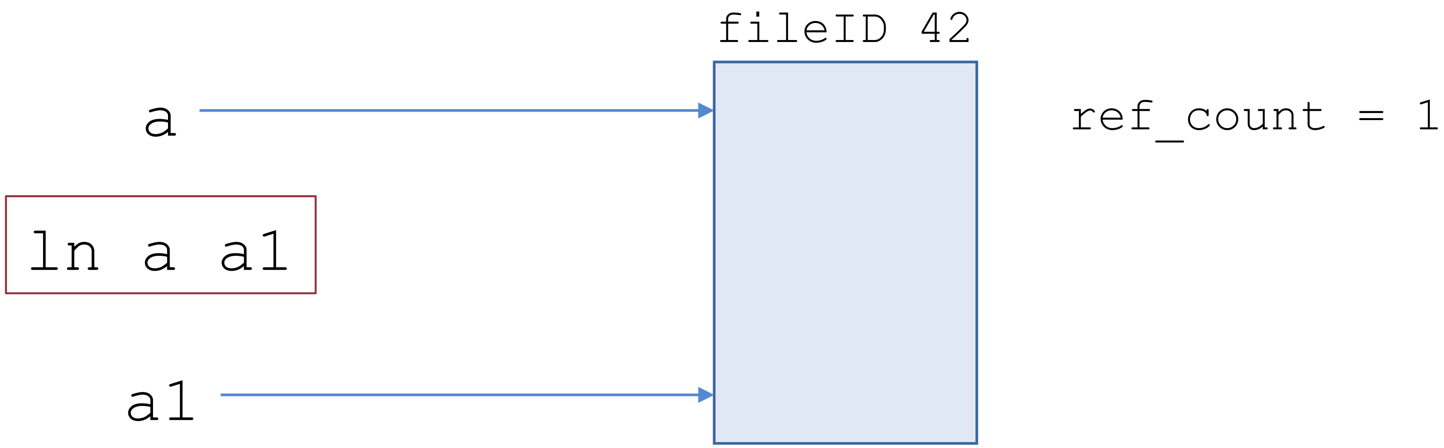
Referential Naming: Hard Links



Referential Naming: Hard Links

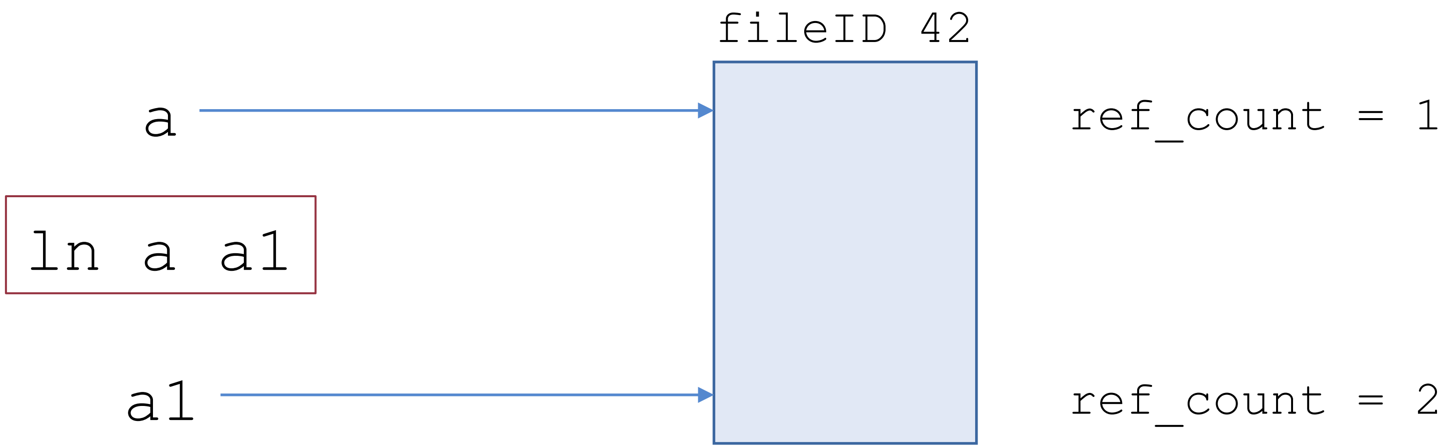


Referential Naming: Hard Links



Adds a second connection to a file

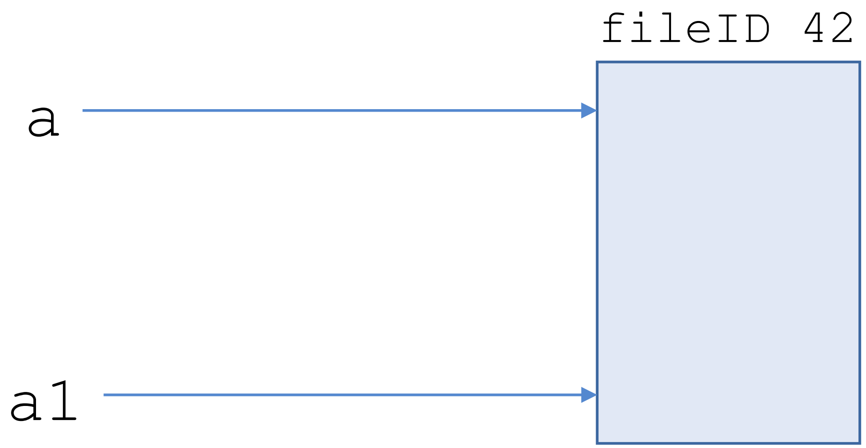
Referential Naming: Hard Links



Adds a second connection to a file

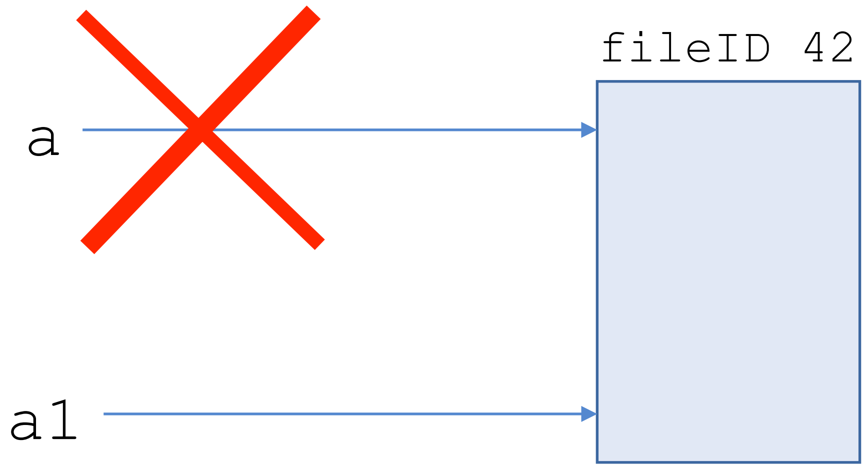
OS maintains reference counts, so it will delete a file only when the last hard link is deleted

Referential Naming: Hard Links



Change to the file using any of its hard links is reflected globally

Referential Naming: Hard Links



Removing a reference **does not** affect others!

as long as reference count > 0

Referential Naming: Hard Links

Problem

Hard links to directories may cause circular links which prevent the OS from claiming back disk space

Referential Naming: Hard Links

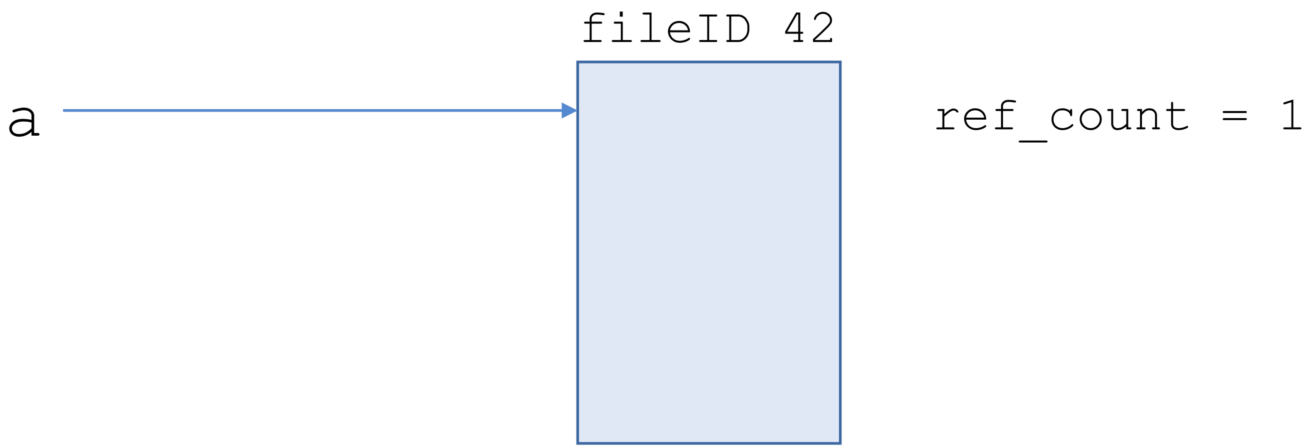
Problem

Hard links to directories may cause circular links which prevent the OS from claiming back disk space

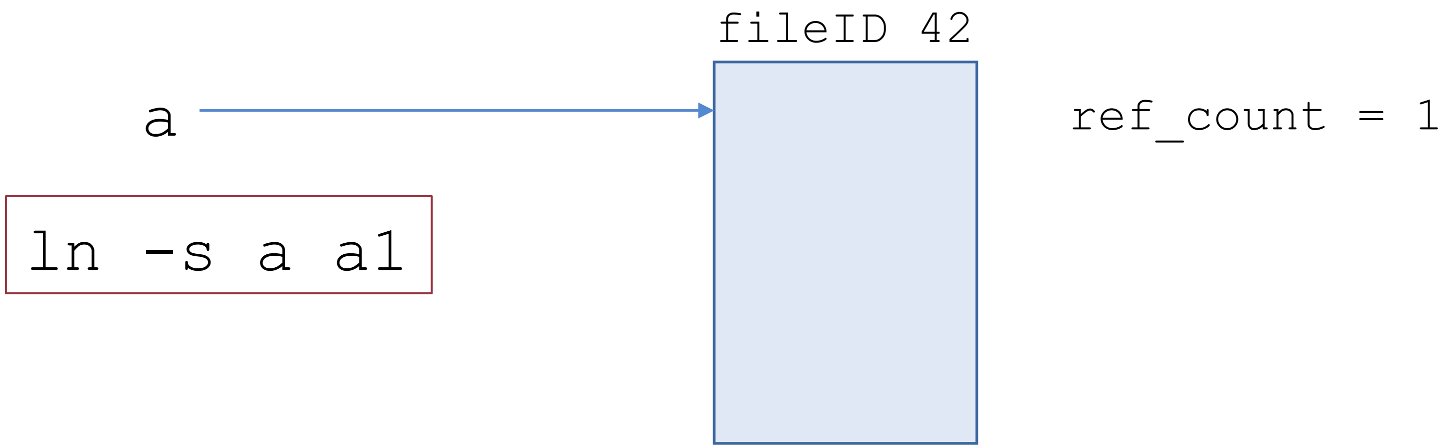
Solution

Do not allow hard links to directories at all!
Hard links to files are safe since files are leaves of the tree

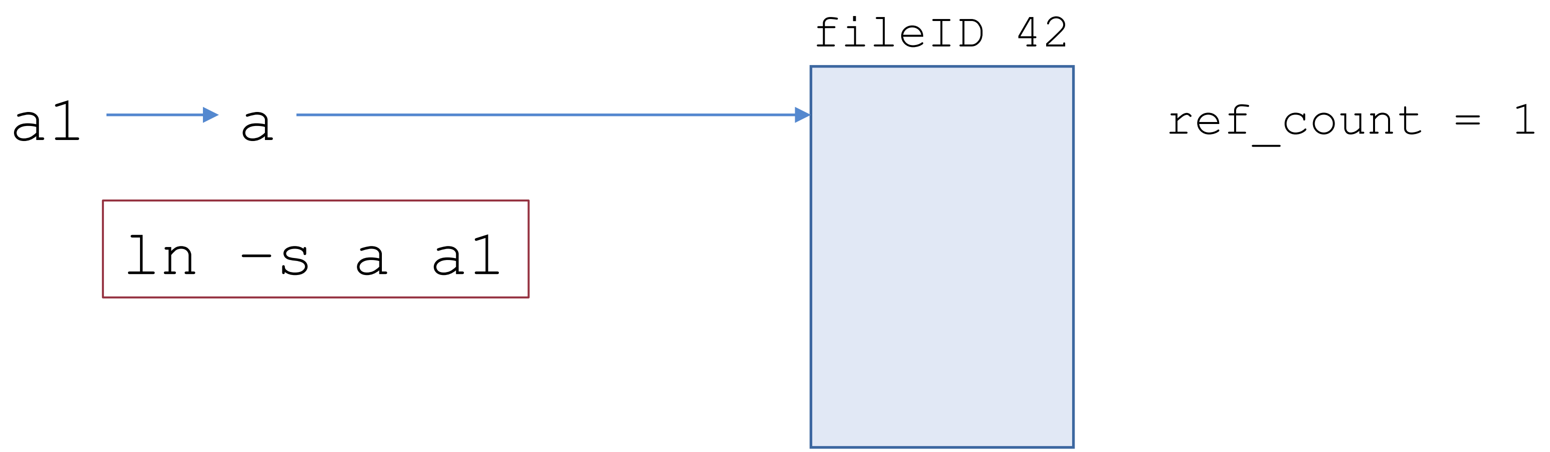
Referential Naming: Soft Links



Referential Naming: Soft Links

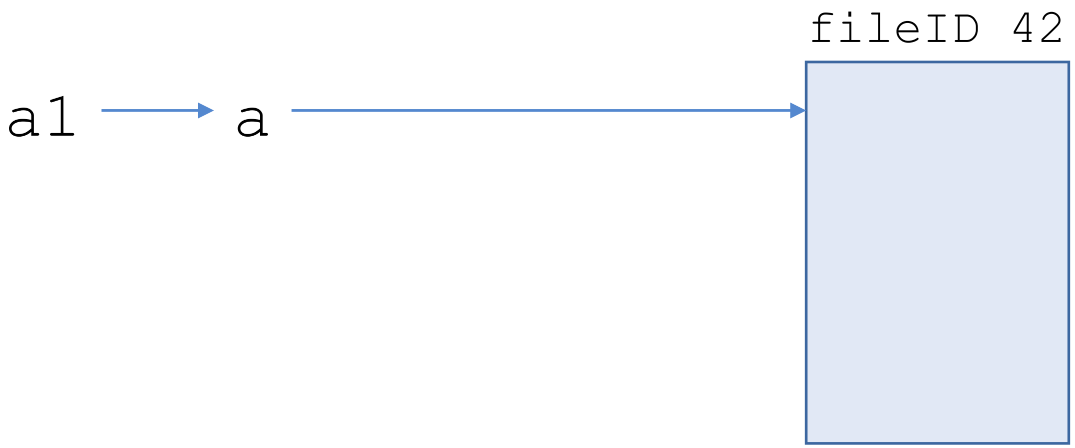


Referential Naming: Hard Links



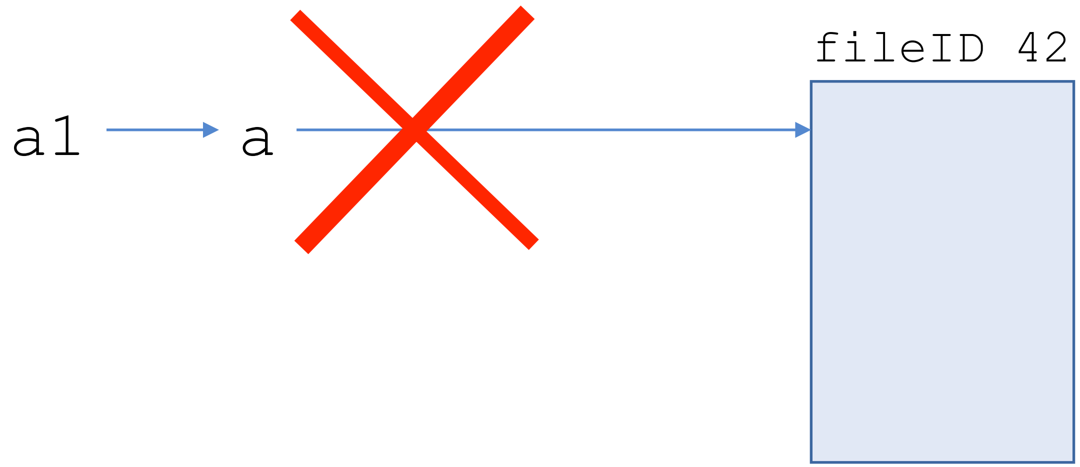
Adds a symbolic pointer to a file

Referential Naming: Soft Links



Change to the file using soft link is reflected globally

Referential Naming: Soft Links



Removing a reference affects all the symbolic links pointing to the file!

a1 remains in the directory but its content no longer exists (dangling pointer)

File Protection

- The OS must allow users to control sharing of their files

File Protection

- The OS must allow users to control sharing of their files
- Control access to files: grant or deny access to file operations depending on protection information

File Protection

- The OS must allow users to control sharing of their files
- Control access to files: grant or deny access to file operations depending on protection information
- 2 different approaches:
  - access lists and groups (Windows NT)
  - access control bits (UNIX/Linux)

File Protection: Access Lists

- Keep an access list for each file with user name and type of access
- PRO: Highly flexible solution
- CON: Lists can become large and tedious to maintain

File Protection: Access Control Bits

- 3 categories of users: (`owner`, `group`, `world`)
- 3 types of access privileges: (`read`, `write`, `execute`)
- Keep one bit for each privilege on each category

(`111101000` = `rwxr-x---`)

- PRO: Easy to implement and maintain
- CON: Less accurate

Summary

- The File System interface provides a convenient abstraction to users/applications that need to interact with I/O devices
- The OS is responsible to expose and implement such an interface hiding any specific details to users/applications
- File is the abstract data type used by the OS
- Operations on files are exposed through device-independent APIs