Operating Systems, fall 2002

Local File Systems in UNIX

Lior Amar,
David Breitgand
(recitation)
www.cs.huji.ac.il/~os

I/O: UNIX approach

- The basic model of the UNIX I/O system is a sequence of bytes that can be accessed either *randomly* or *sequentially*
- Applications may need various level of structure for their data, but the kernel imposes no structure on I/O
- **Example:** ASCII text editors process documents consisting of *lines* of characters where each line is terminated by ASCII *line-feed* character. Kernel knows nothing about this convention

I/O stream

- The UNIX kernel uses a single data model, *byte stream*, to serve all applications;
- As a result I/O stream from one program can be fed as input to any other program;
- Pipelines can be formed;
- **This is a characteristic UNIX tool-based approach;**

Descriptors

- Unix processes use *descriptors* to reference I/O streams;
- Descriptors are small unsigned integers;
- Descriptors are obtained from system calls *open()*, *socket()*, *pipe();*
- System calls *read()* and write() are applied to descriptors to transfer data;
- System call *lseek()* is used to specify position in the stream referred by desriptor;
- System call *close()* is used to de-allocate descriptors and the objects they refer to.

What’s behind the descriptor?

- Descriptors represent *objects* supported by the kernel:
- **file**
- **pipe**
- **socket**

File

- A linear array of bytes with at least one name;
- A file exists until all its names are explicitly deleted, and no process holds a descriptor for it;
- In UNIX, I/O devices are accessed as files. These are called *special device files;*
- There is nothing special about them for the user processes, though;
- Terminals, printers, tapes are all accessed as if they were streams of bytes;
- They have names in the file system and are referred to through their descriptors.

Special Files

- The kernel can determine to what hardware device a special file refers and uses a resident module called *device driver* to communicate with the device;
- Device special files are created by the *mknode()* system call (by the super-user only)
- To manipulate device parameters *ioctl()* system call is used;
- Different devices allow different operations through *ioctl()*
- Devices are divided into two groups:
  - **Block devices (structured)**
  - **Character devices (unstructured)**

Devices are not created equal

- **Block devices**:
  - Random (anywhere in the stream) access devices;
- Internal implementation is based on the notion of ***block***, a minimal group of bytes that can be transferred in one operation to and from the device;
- A number of blocks can be transferred in one operation (this is, usually, more efficient), but less then block bytes of data is not transferred;
- **To user application, the block structure of the device is made transparent through internal buffering being done in kernel. User process may read/write a single byte because it works with I/O stream abstraction ☺**
- tapes, magnetic disks, drums, cd-roms, zip disks, floppy disks, *etc.*

Devices are not created equal

- **Character devices**:
  - Sequential access devices;
  - Internal implementation often supports the notion of ***block transfer***,
  - Moreover, in many cases the blocks supported by character devices are very large due to efficiency considerations (e.g., communication interfaces)
- Then why they are called character?
  - Because the first such devices were terminals
- Mouse, keyboard, display, network interface, printer, *etc*.

Devices are not created equal

File systems, organized, collections of files, are **always** created on the block devices, and **never** on the character devices

Block devices can (and usually do) support character device Interface. But the opposite is not true.

Single physical block device can be partitioned into a number of logical devices. Each such logical device can have its own file system. Each such logical device is represented by its own special device file. //take a look at **/dev** directory to see them

So far, it's enough with the special files. But we'll get back to them later on :)

pipe

- They are linear array of bytes as files, but they are unidirectional sequential communication links between the related processes (father/son);
- They are transient objects;
- They get their file names in the /tmp directory automatically, but *open()* cannot be used for them;
- Descriptors obtained from *pipe()* system call.
- Data written to a pipe can be read only once from it, and only in the order it was written (FIFO);
- Have limited size.

FIFO

- There is a special kind of pipes, called *named pipes*;
- They are identical to unnamed pipes, except they have normal names, as any other file, and descriptors for them can be obtained through *open()* system call;
- Processes that wish to communicate through them in both directions should open one FIFO for every direction.

Socket

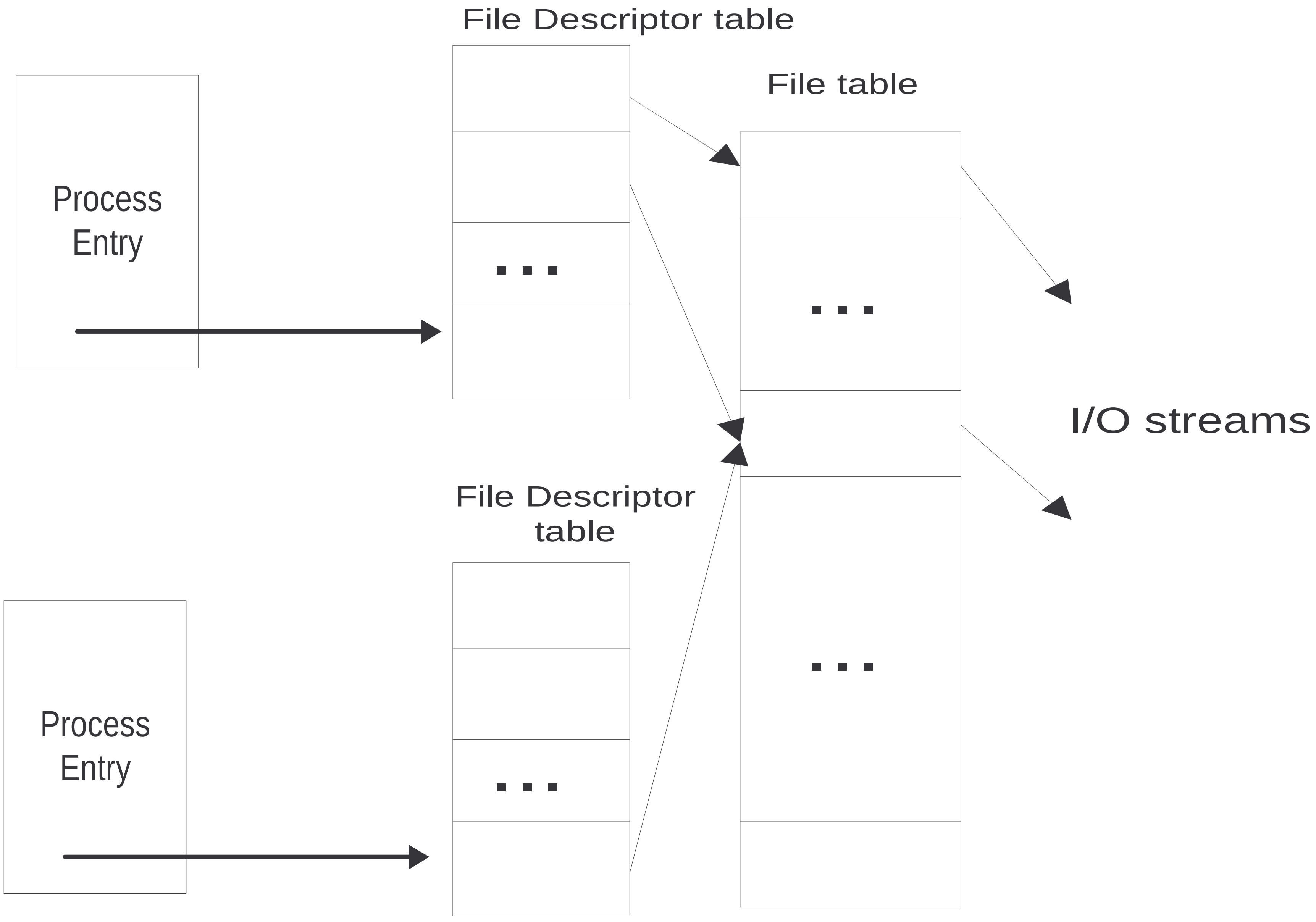
- Socket is a transient object that is used for inter-process communication;
- It exists only as long as some process holds a descriptor on it;
- Descriptor is created through the *socket()* system call;
- Sequential access; similar to pipes;
- Different types of sockets exist:
  - Local IPC;
  - Remote IPC;
  - Reliable/unreliable *etc.*

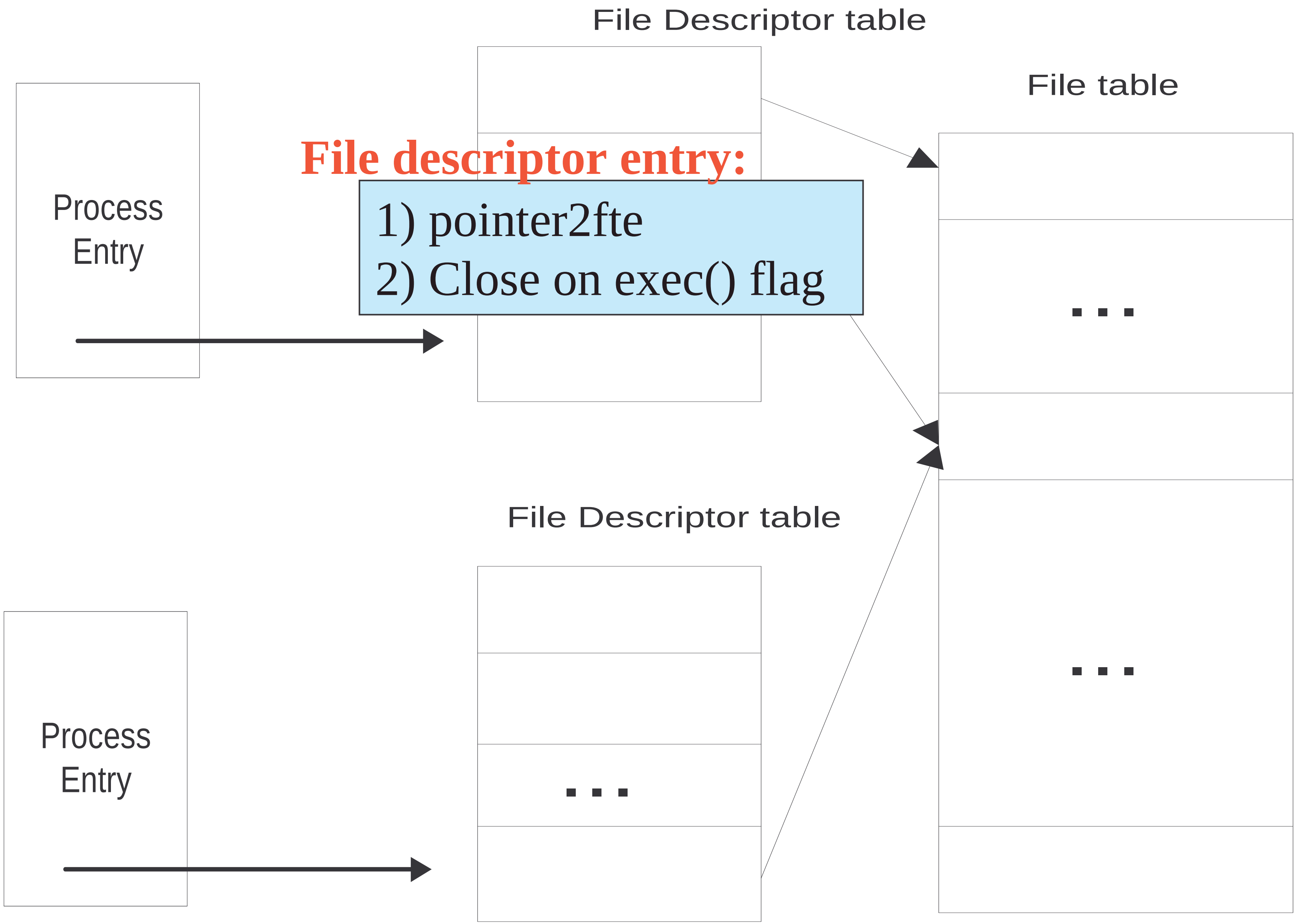
To summarize, so far

- Descriptor refers to some kind of I/O stream
- But all I/O streams have the same interface:
  - **file**

Where descriptors are?

- The kernel maintains a per-process *descriptor table* that kernel uses to translate the external representation of I/O stream into internal representation;
- Descriptor is simply an index into this table;
- Consequently, descriptors have only local meaning;
- Different descriptors in different processes can refer to the same I/O stream;
- Descriptor table is inherited upon *fork()*;
- Descriptor table is preserved upon *exec();*
- ***When a process terminates the kernel reclaims all descriptors that were in use by this process***


File Descriptor table
File table
Process
Entry
...
...
I/O streams
File Descriptor
table
...
Process
Entry
...



File Descriptor table

File table

Process Entry

File descriptor entry:

1) pointer2fte
2) Close on exec() flag

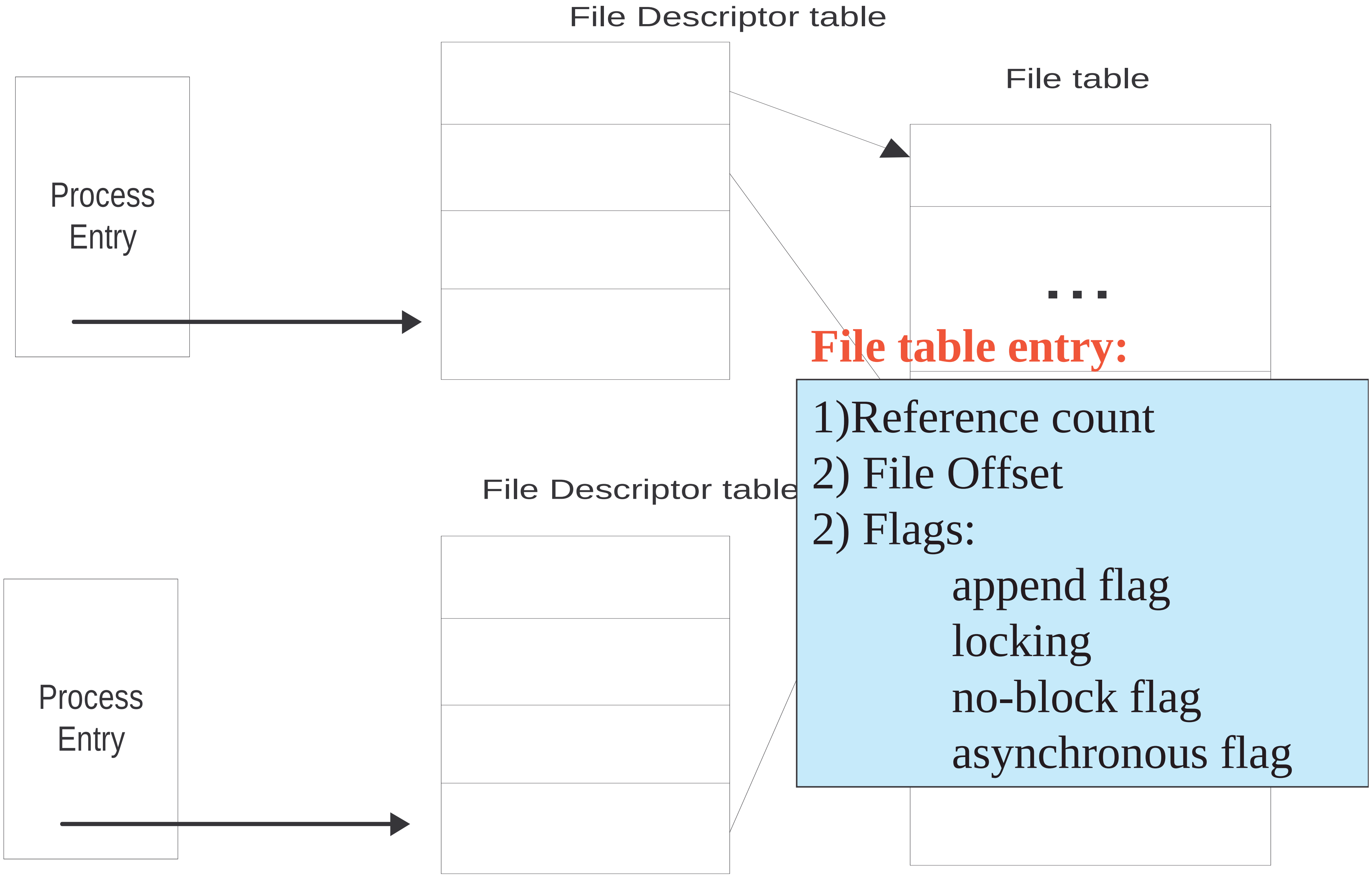
. . .

File Descriptor table

Process Entry

. . .

. . .



File Descriptor table

File table

Process Entry

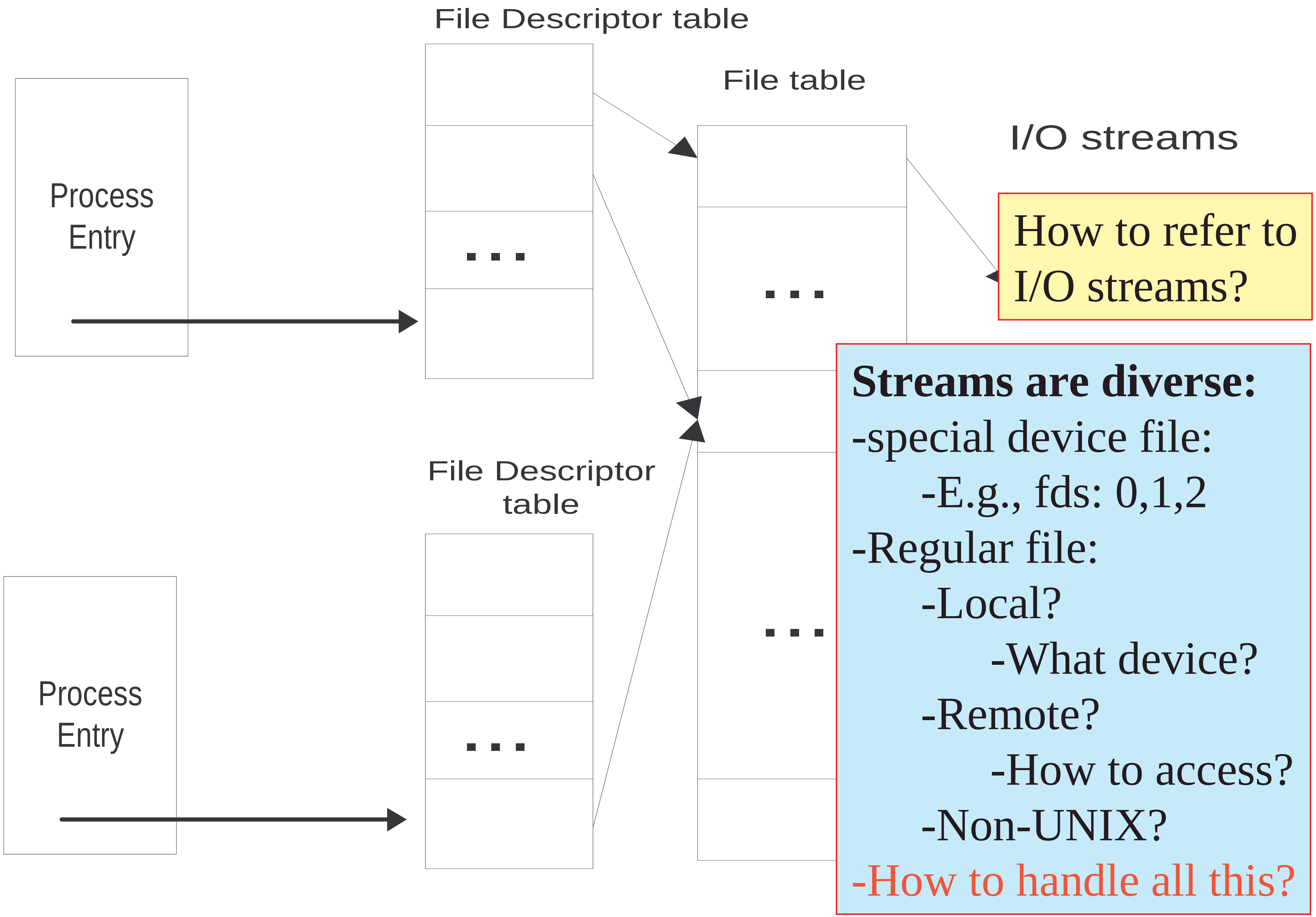
…

File table entry:

1)Reference count
2) File Offset
2) Flags:
append flag
locking
no-block flag
asynchronous flag

File Descriptor table

Process Entry

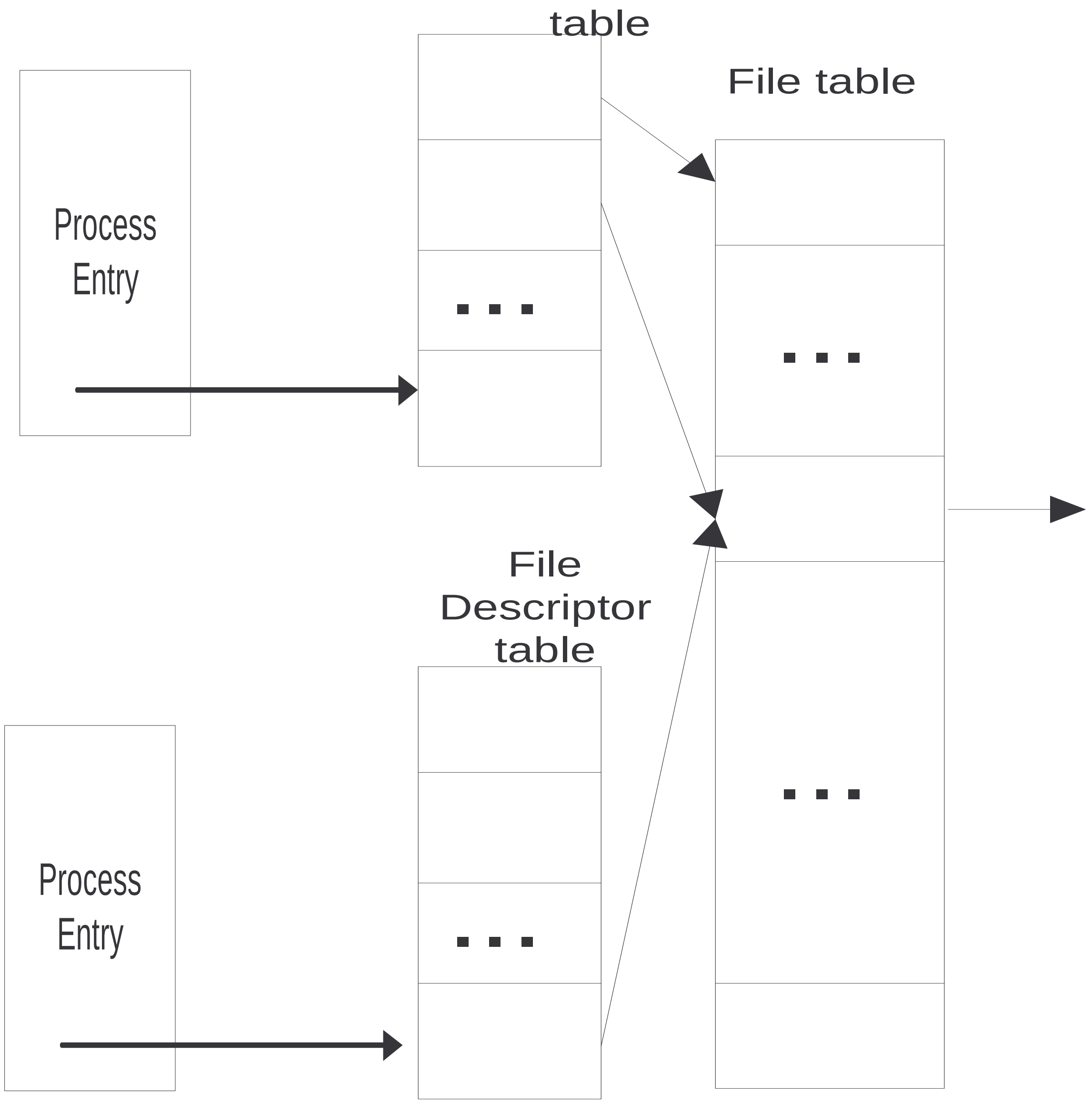


How to refer to I/O streams?

Streams are diverse:

- -special device file:
  - -E.g., fds: 0,1,2
- -Regular file:
  - -Local?
    - -What device?
  - -Remote?
    - -How to access?
  - -Non-UNIX?
- -How to handle all this?

Solution: virtual file system layer



V-node structure:

- object oriented interface between the generic file representation and its internal implementation
- Function table;
- References the actual implementation:
  - Local file I-node;
  - Device driver;
  - NFS, etc.

V-node layer

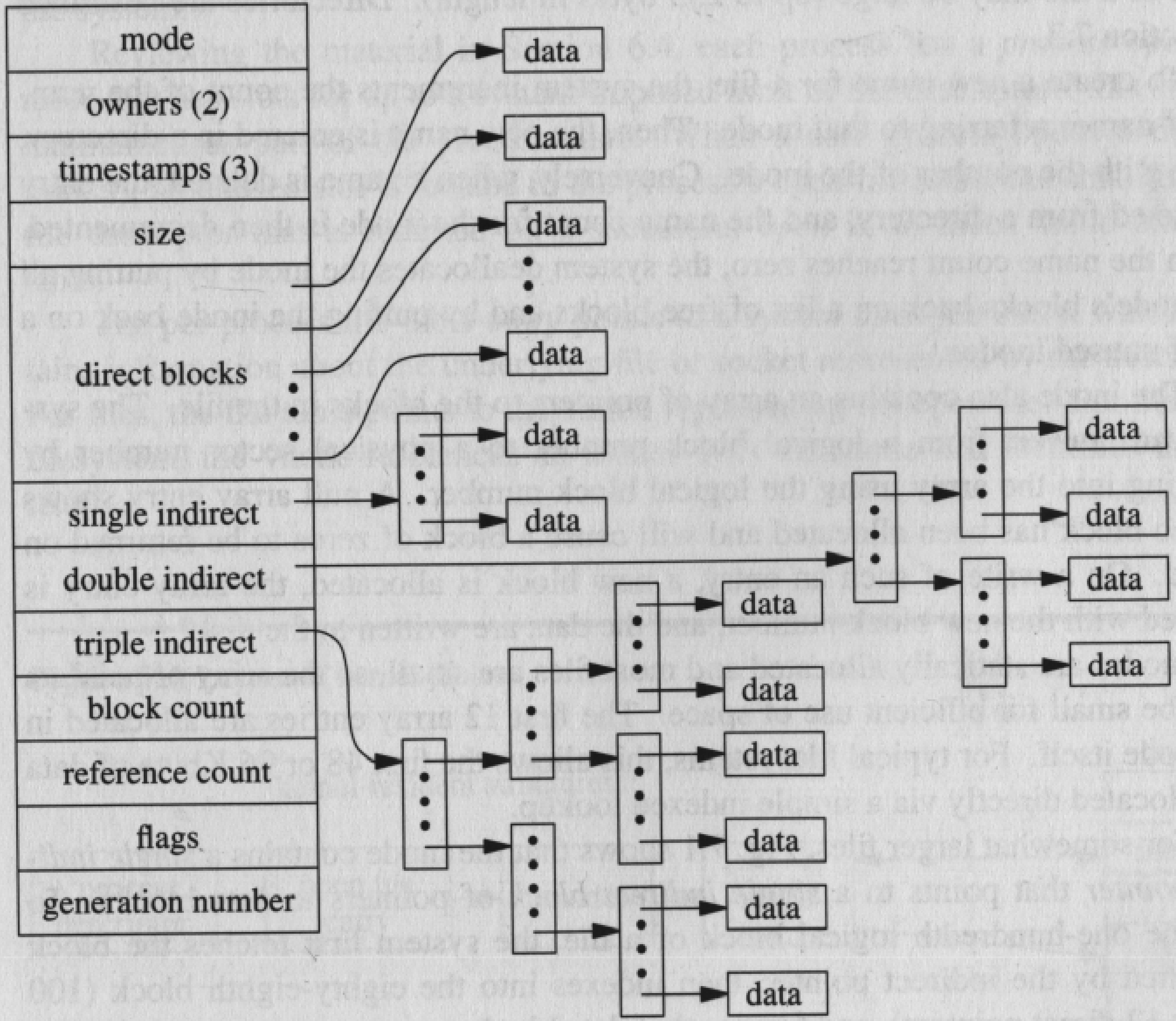
V-node interface functions consist of:

- File system independent functions dealing with:
  - Hierarchical naming;
  - Locking;
  - Quotas;
  - Attribute management and protection.
- Object (file) creation and deletion, read and write, changes in space allocation:
  - These functions refer to file-store internals specific to the file system:
  - Physical organization of data on device;
  - For local data files, these functions refer to v-node refers to UNIX-specific structure called i-node (*index node*) that has all necessary information to access the actual data store.

Regular Local Files and I-nodes

- Information about each regular local file is contained in the structure called I-node;
- There is 1-to-1 mapping between the I-node and a file.
- I-node structures are stored on the file system block device (e.g., disk) in a predefined location;
- Where it is exactly is file system implementation specific;
- To work with a file (through the descriptor interface) the I-node of the file should be brought into the main memory (in-core I-node) ->

Figure 7.1 The structure of an inode.



In-core I-nodes

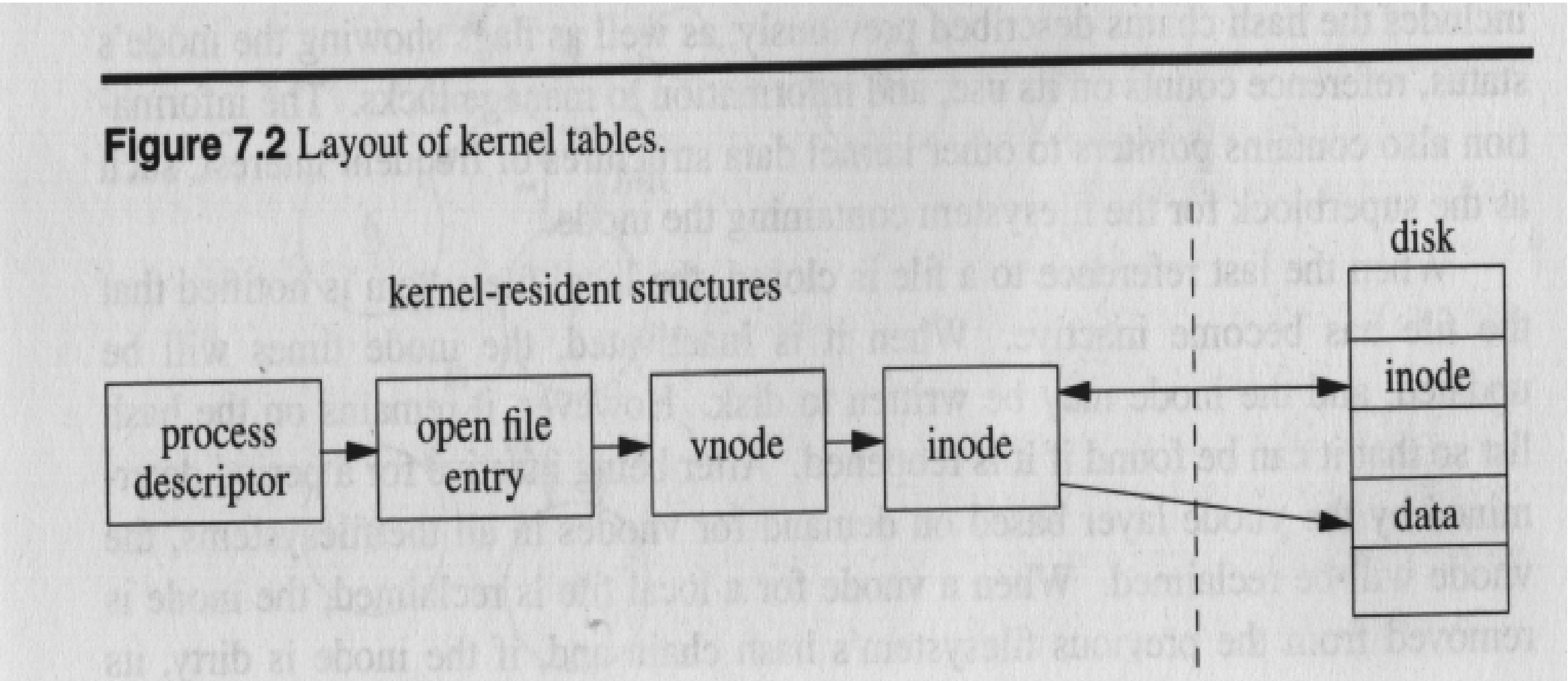


Figure 7.2 Layout of kernel tables.

Additional information:

1. how many file entries refer to I-node;
2. Locking status;

How I-nodes are identified?

- Each I-node is identified through its *number*
- Non-negative integer;
- This number serves as an index into the I-node list implemented in each file system;
- And there is a file system per device, remember?
- **Thus, I-node numbers have only local meaning**
- How to efficiently refer to the in-core I-nodes then?

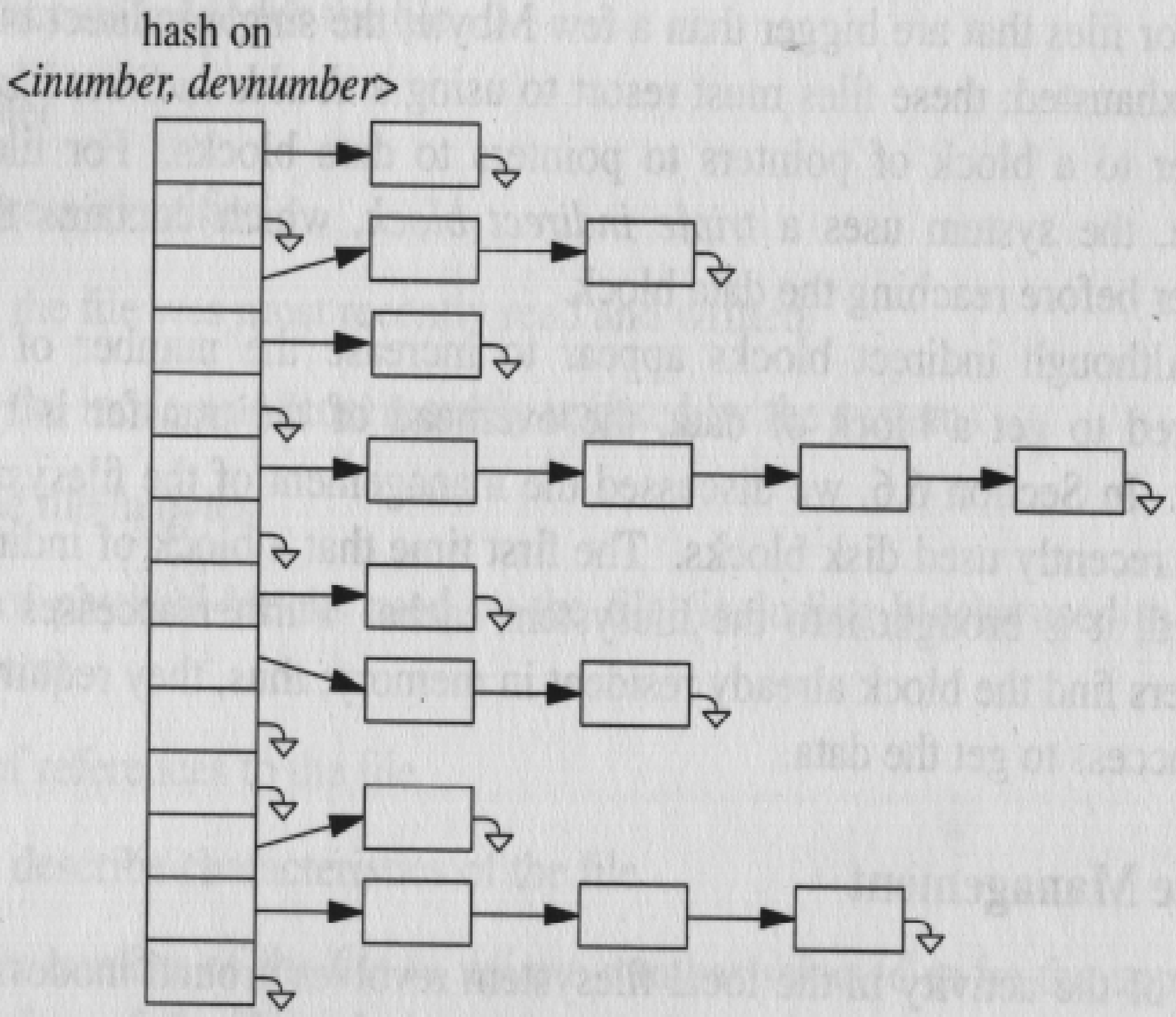


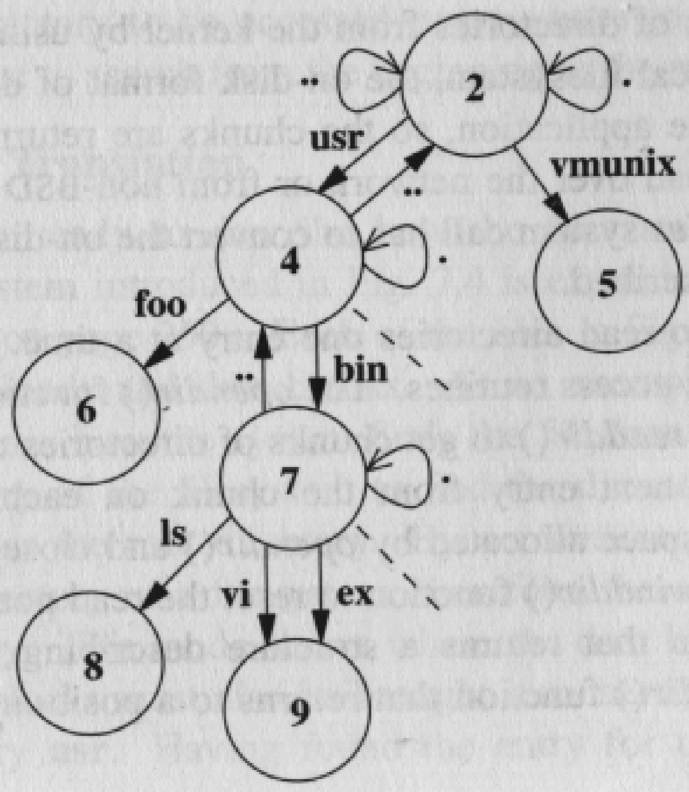
Figure 7.3 Structure of the inode table.

Issues with I-nodes

- Since in-core I-node should be created for each open file, we need a mechanism how to map the *filename* into the I-node number;
- Since this is an often used operation, I-nodes lookup and management should be efficient in time and space;
- Since each file should be allocated an I-node structure, we need to know what is in use, and what is free;

Logical View of the File System (generic)

Figure 7.4 A small filesystem tree.



Directories

- In UNIX there are special files (don’t mix with special device files) called *directories;*
- Directory is a file containing information about other files;
- As a file directory has an I-node structure;
- Flag in the structure indicates its type;
- **In contrast to other files, the kernel imposes a structure on directories**
- Directory is a collection of *directory entries* of variable length where each entry contains mapping:
- **<name, inode #>**

Directories

- Directories allocated in *chunks*
- Each chunk can be read/written in a single I/O operation;
  - – Why?

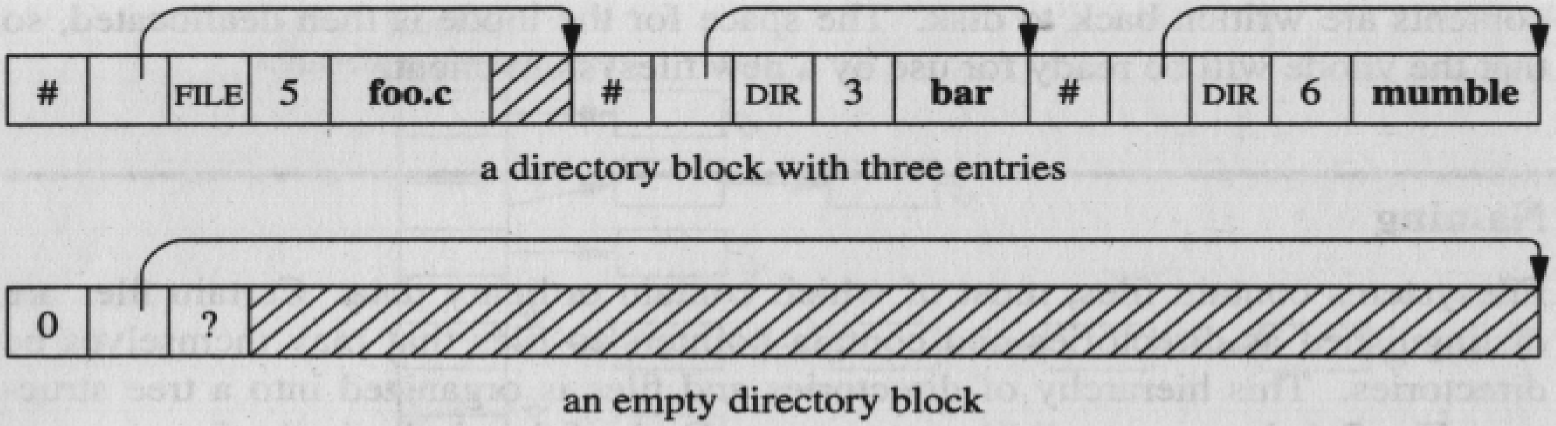


Figure 7.5 Format of directory chunks.

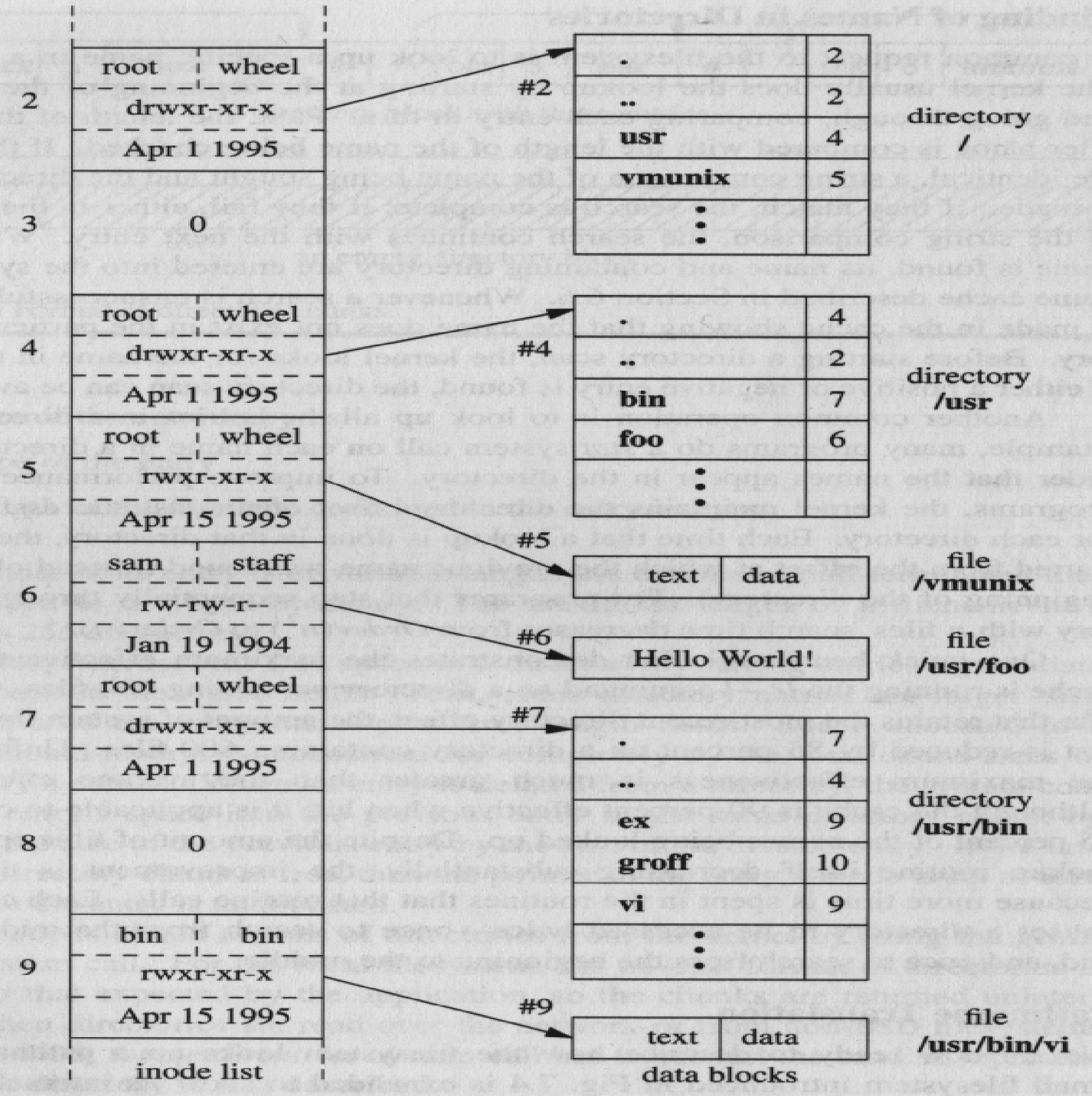
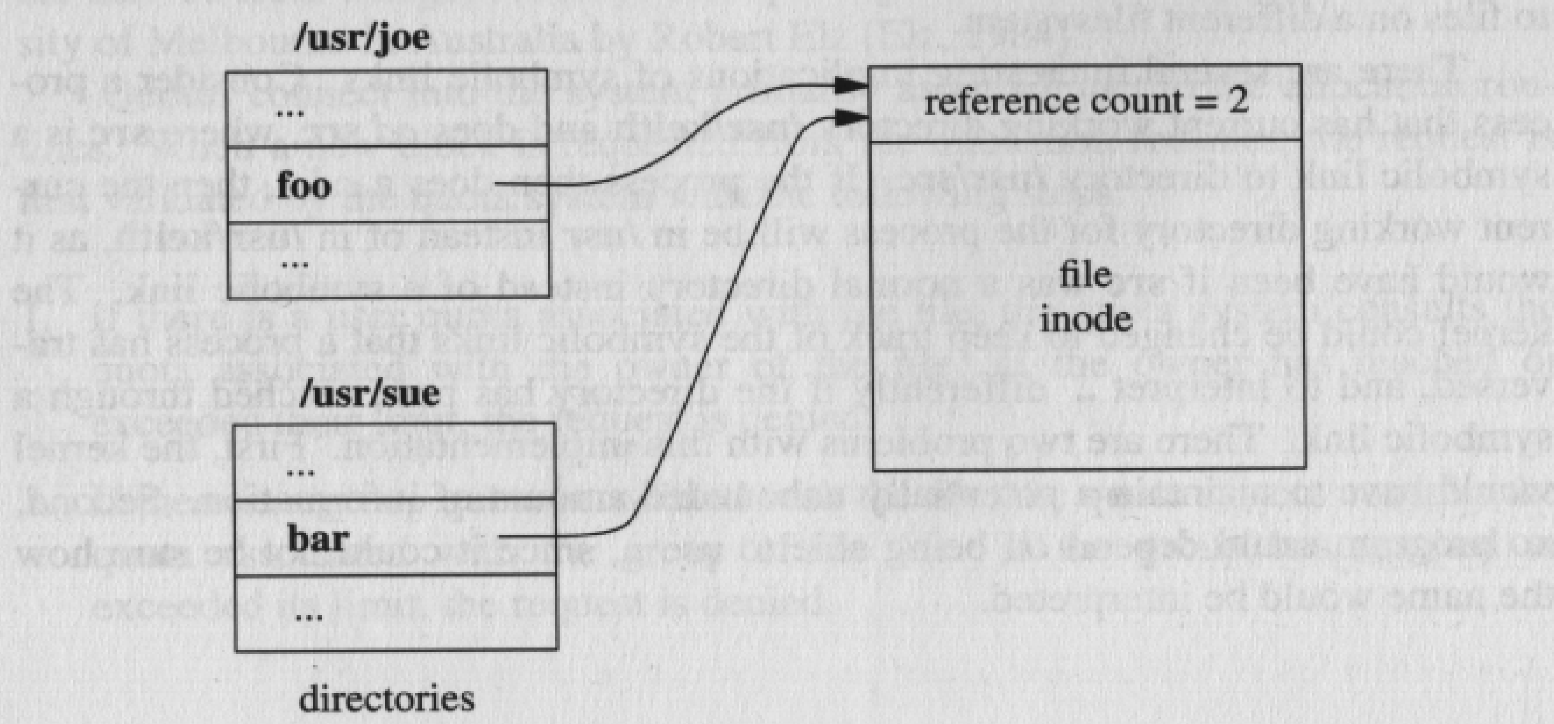


Figure 7.6 Internal structure of a small filesystem.

Links (hard)

Figure 7.7 Hard links to a file.



Hard links cannot span different file systems (local meaning only)

Links (soft)

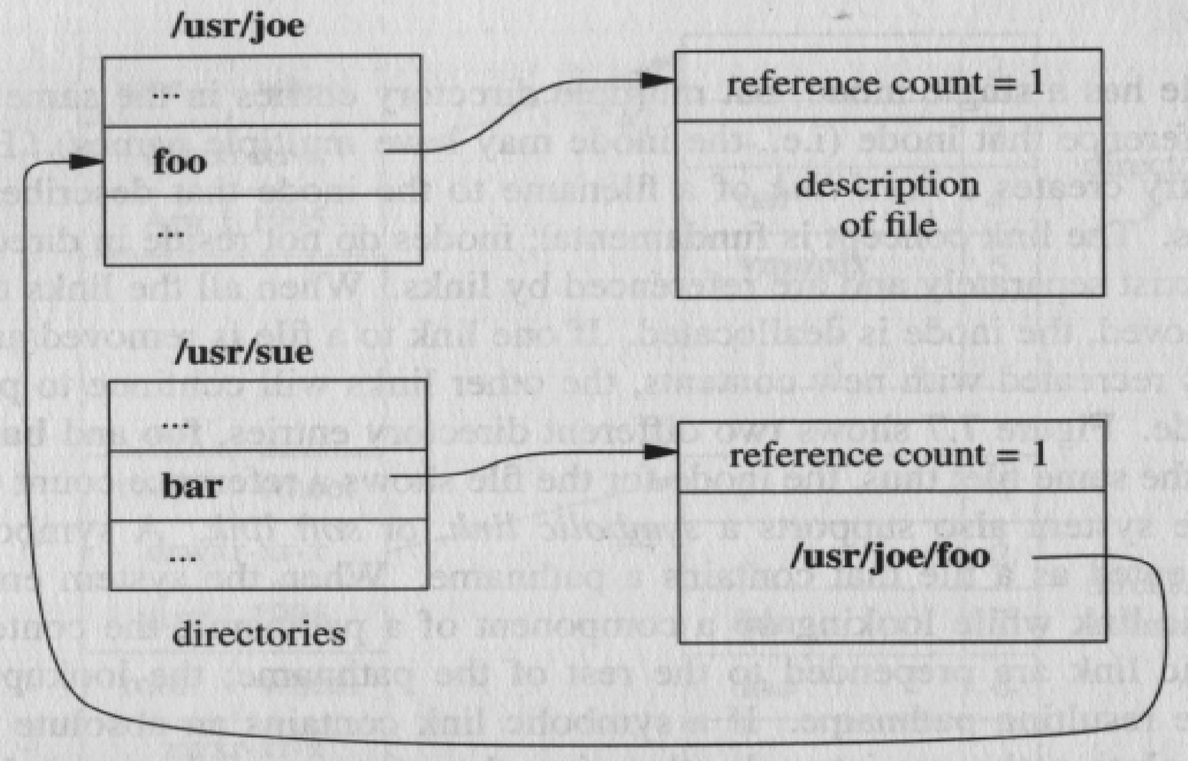


Figure 7.8 Symbolic link to a file.

Soft links can span the device boundaries

Local File System Organization

- Classical UNIX File System (old);
- Sequentially from a predefined disk addresses (cylinder 0, sector 0):
  - Boot block;
  - Superblock;
  - I-node hash-array;
  - Data blocks

Superblock

- Contains;
  - Size of the file system;
  - The number of free blocks in the file system;
  - Size of the logical file block;
  - A list of free blocks available for file allocation;
  - Index of the next free block on the list;
  - The size of I-node list;
  - The number of free I-nodes on the system;
  - The list of free I-nodes on the file system;
  - The index of the next free I-node on the list.

I-node allocation

- As long as there is a free I-node – allocate it;
- Otherwise scan the I-node list linearly, and enter into the super-block list of free I-nodes as many numbers as possible;
- Remember the highest free I-node number;
- Continue with step 1;
- Next time start scanning from the remembered I-node number; when at the end – go back to the beginning of the I-node list.

Relation between logical and physical views

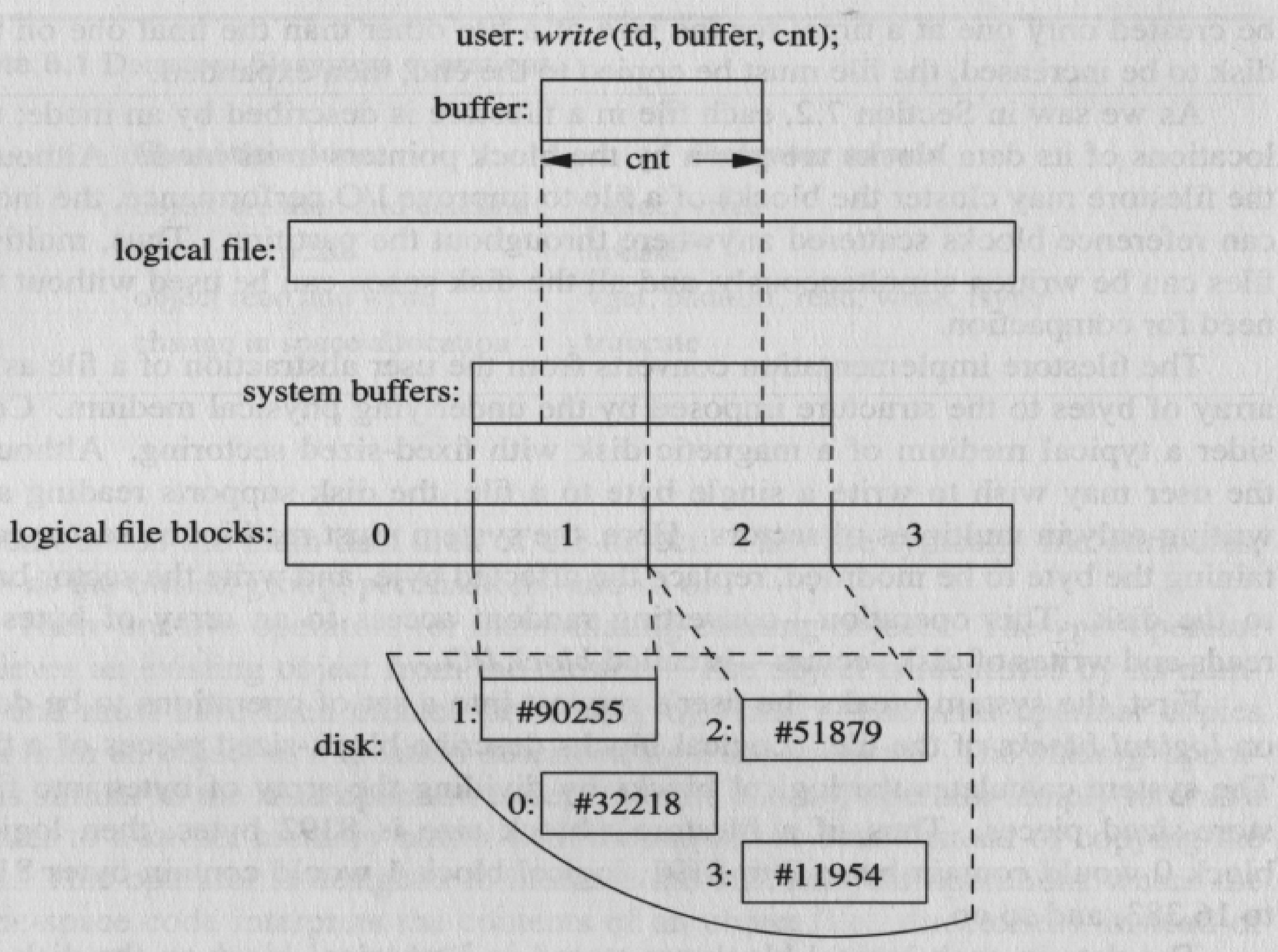
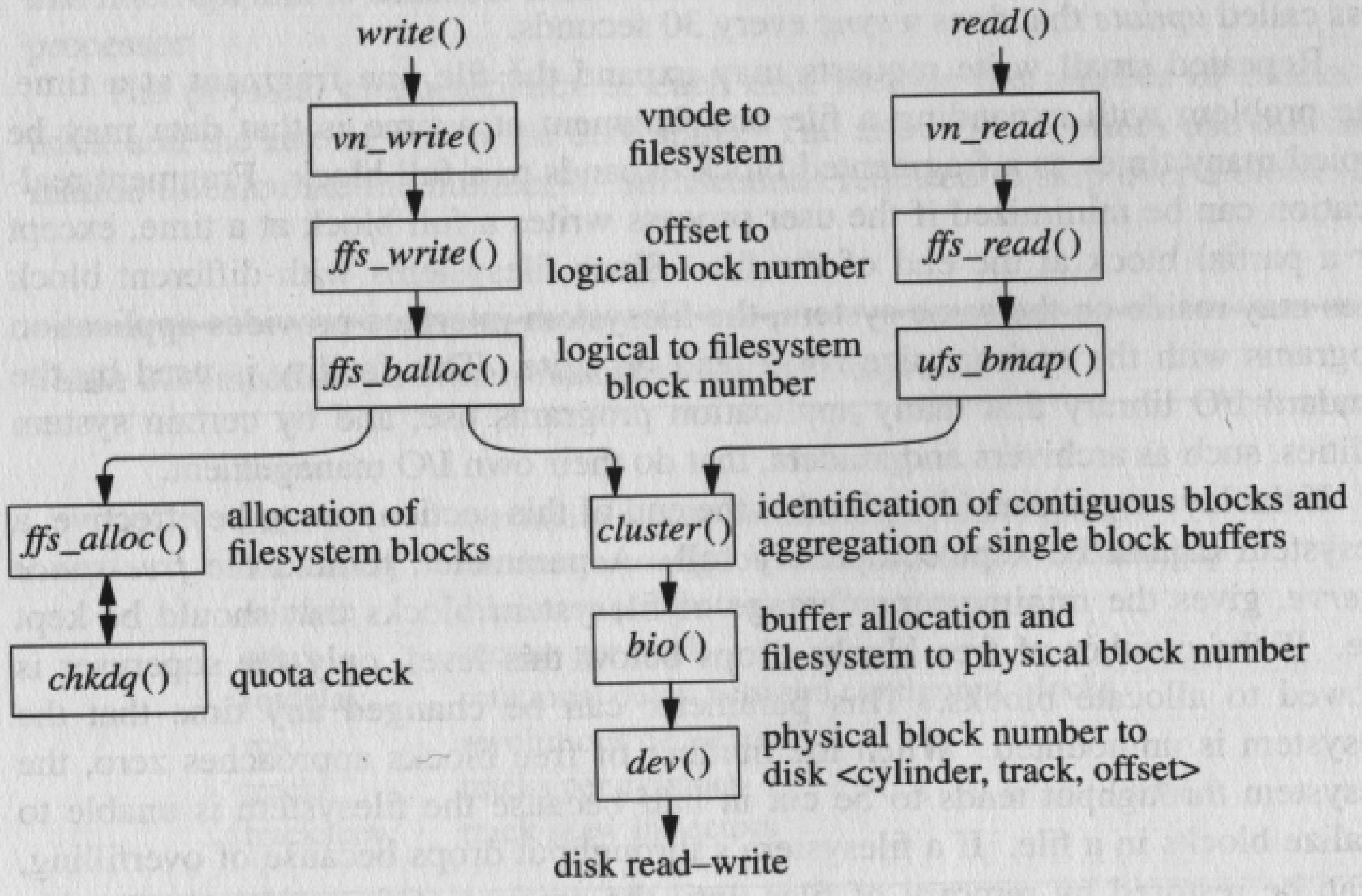


Figure 8.1 The block I/O system.

Figure 8.4 Procedural interface to reading and writing.



Example

- Block size is 1024;
- Offset 9000:

  (9000)mod(1024) = 808;

  (9000)div(1024) = 8;
- Offset 350,000:

  (350,000)div(1024) = 341:

  75-th direct block of the single indirect block of the 0-th double indirect block indirect block

  (350,000)mod(1024) = 816;

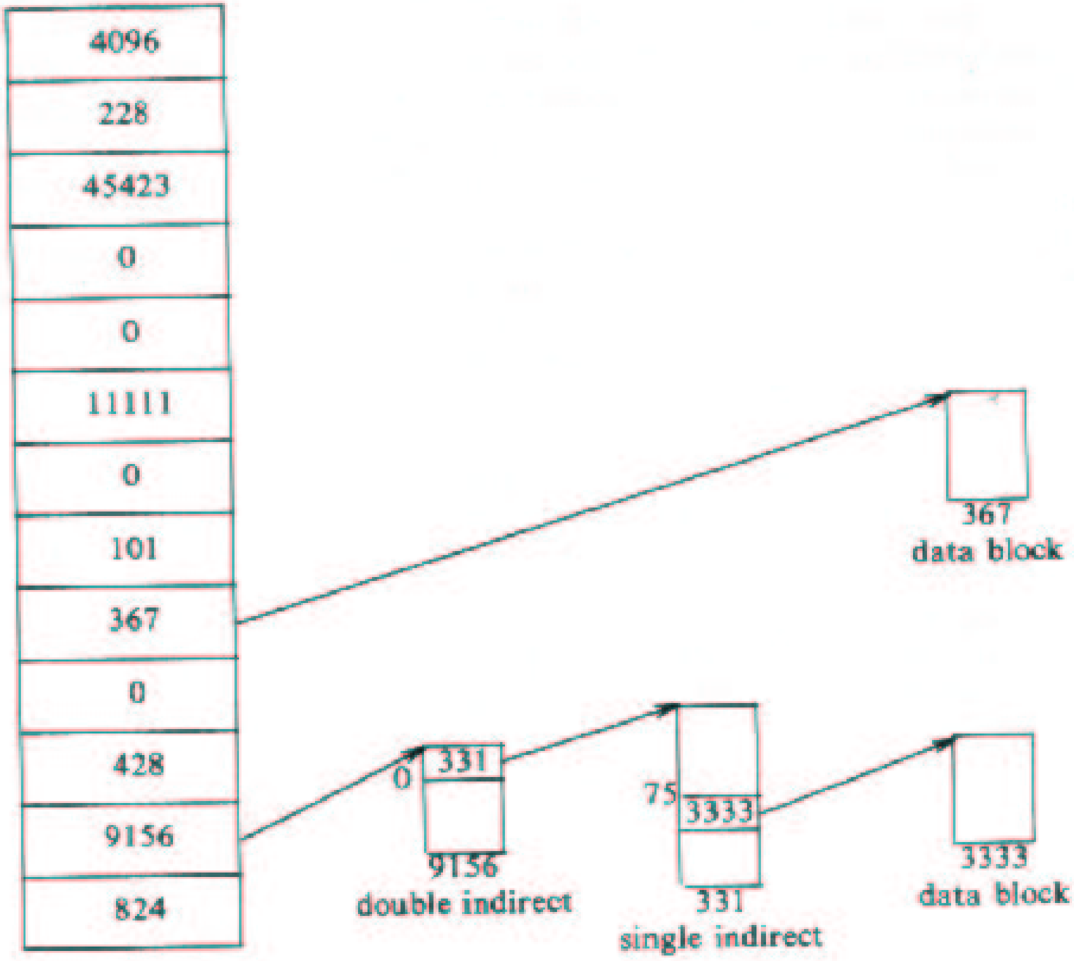


Figure 4.9. Block Layout of a Sample File and its Inode

What’s next?

- Issues with the old UNIX file system;
- BSD FFS (new fsystem)
- Log-based file system;
- NFS.