Chapter 11: File Systems

11.1 Attributes

Metadata – attributes associated with a file (e.g. name, access rights, ... etc.)

- The space that metadata takes up is called *overhead*

List of attributes

1. Name

   > We often use a hierarchical system like the one shown below

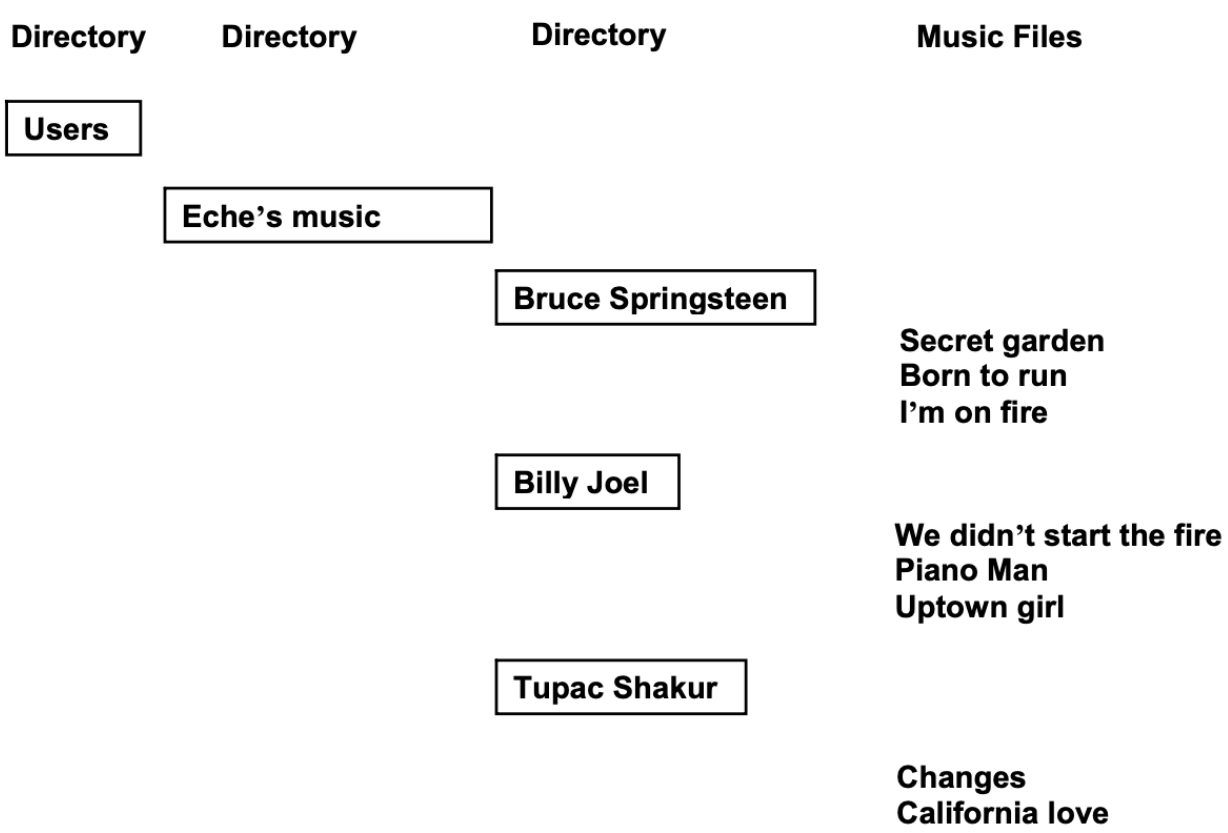


Figure 11.1: Multi-level directory

Some OS require files to have an extension (e.g. `.txt` or `.docx`). Other systems such as UNIX and Windows just use the extension to guess how to render and open the file.

An *alias* allows the user to create a "nickname" for a file

- `ln foo bar` creates an alias bar for the file foo
  - This is called a *hard link*
  - If we delete the file `foo`, we can still access it through `bar`
  - These two have the same i-node
  - `bar` will point to the same file as `foo`, so think of it as another reference to the same object. Deleting just `foo` only deletes one reference
  - **i-node access rights hard links size creation time name**

    ```
    3193357 -rw-------  2 rama 80 Jan 23 18:30 bar
    3193357 -rw-------  2 rama 80 Jan 23 18:30 foo
    ```
- `ln -s foo bar` creates an alias as well, but it is a `soft link`
  - This is called name equivalence and `bar` only points to `foo` instead of the actual file

- **i-node access rights hard links size creation time name**

```
3193495 lrwxrwxrwx   1 rama  3 Jan 23 18:52 bar -> foo
3193357 -rw-------   1 rama 80 Jan 23 18:30 foo
```

- Note how the size of `bar` is only 3 bytes, since it just contains the pointer to foo
- If you delete `bar` now, the file still exists. However, if you delete `foo`, trying to access the file through `bar` will result in an error

Soft links improve usability . However, it has to resolve the alias by traversing its internal data structures.

Hard links improve performance as there is no time traversing internal data structures , but it can lead to circular lists (which makes it really hard to delete directories). For this reason, Unix disallows creating hard links to directories

2. Access Rights

   This attribute specifies *who* has access to a particular file and *what* kind of privilege each allowed user enjoys.

Chmod

```sh
chmod u+x virus.sh
```

Allows for the changing the permission mode of a file. In the above command, we are giving execute privileges to the user

- `r` – read privileges
- `w` – write privileges
- `x` – execute privileges

Chown

```sh
chown richard virus.sh
```

Allows for changing ownership of a file.

Ideally, we may want to give individual access rights to each user in the system to each file in the system. Unix divides this into three groups: user, group, and all

- User is a specific user of the system
- Group is the set of authorized users
- All refers to all authorized users

Chgrp

```sh
chgrp cs2200 virus.sh
```

Makes `cs2200` the group owner of `virus.sh`

Example metadata of a file

```sh
rwxrw-r--   1 rama     fac   2364 Apr 18 19:13 foo
```

- The first three bits encode read, write, and execute privileges for rama
- The next three bits encode read and write privileges for group fac (no execute)
- The final three bits encode read privilege only for all users

The “1” after the access rights states the number of hard links that exists to this file

3. Creation Time
   > Time when the file was created first

4. Last Write Time
   > Time when the file was last written. In most file systems the creation time attribute is the same as the last write time attribute

- Note: moving a file doesn't change last write time

5. Size
   > Total space occupied on the file system

11.2 Design Choices

One of the fundamental design issues in file system is the physical representation of a file on the disk

- Sequential access
  - Commands like `cat`, `more`, `less` that read the contents sequentially
- Random access
  - Searching through the file
  - Commands like `tail`, `grep`

We also need to consider what happens when the file grows in size and how to allocate the space on the disk efficiently

Therefore the *figures of merit* for a file system design are:

- Fast sequential access
- Fast random access
- Ability to grow the file
- Efficient allocation of storage
- Efficiency of space utilization on the disk

We are going to use a 4-tuple for the disk address `(cylinder#, surface#, sector#, size of disk block)`

11.2.1 Continuous Allocation

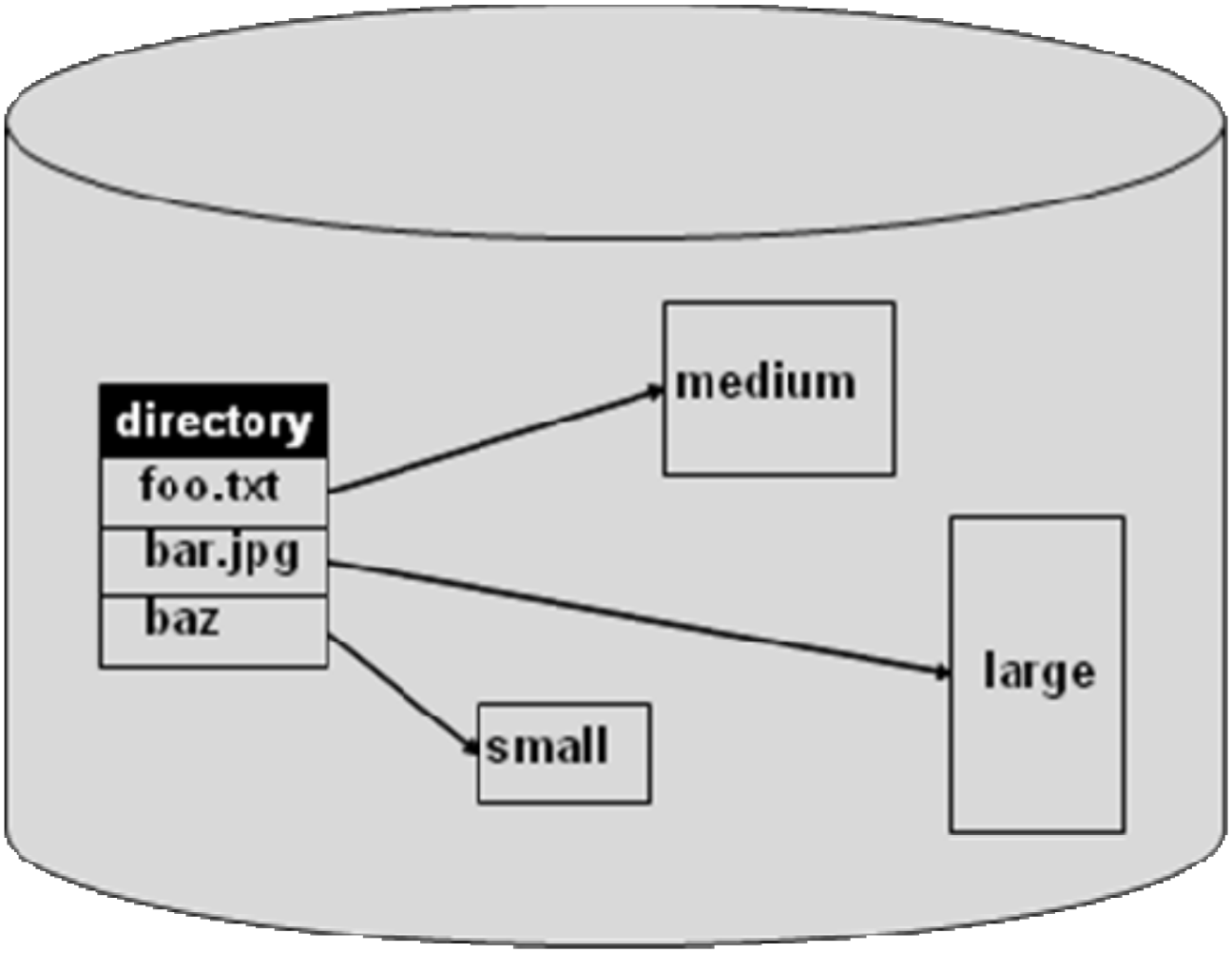


Figure 11.2: Directory that maps file name to a disk block address for contiguous allocation

At file creation time, the file system pre-allocates a fixed amount of space to the file

- Amount of space is determined by file type (e.g. `.txt` or `.wav`)

The file system keeps a free list of available disk blocks

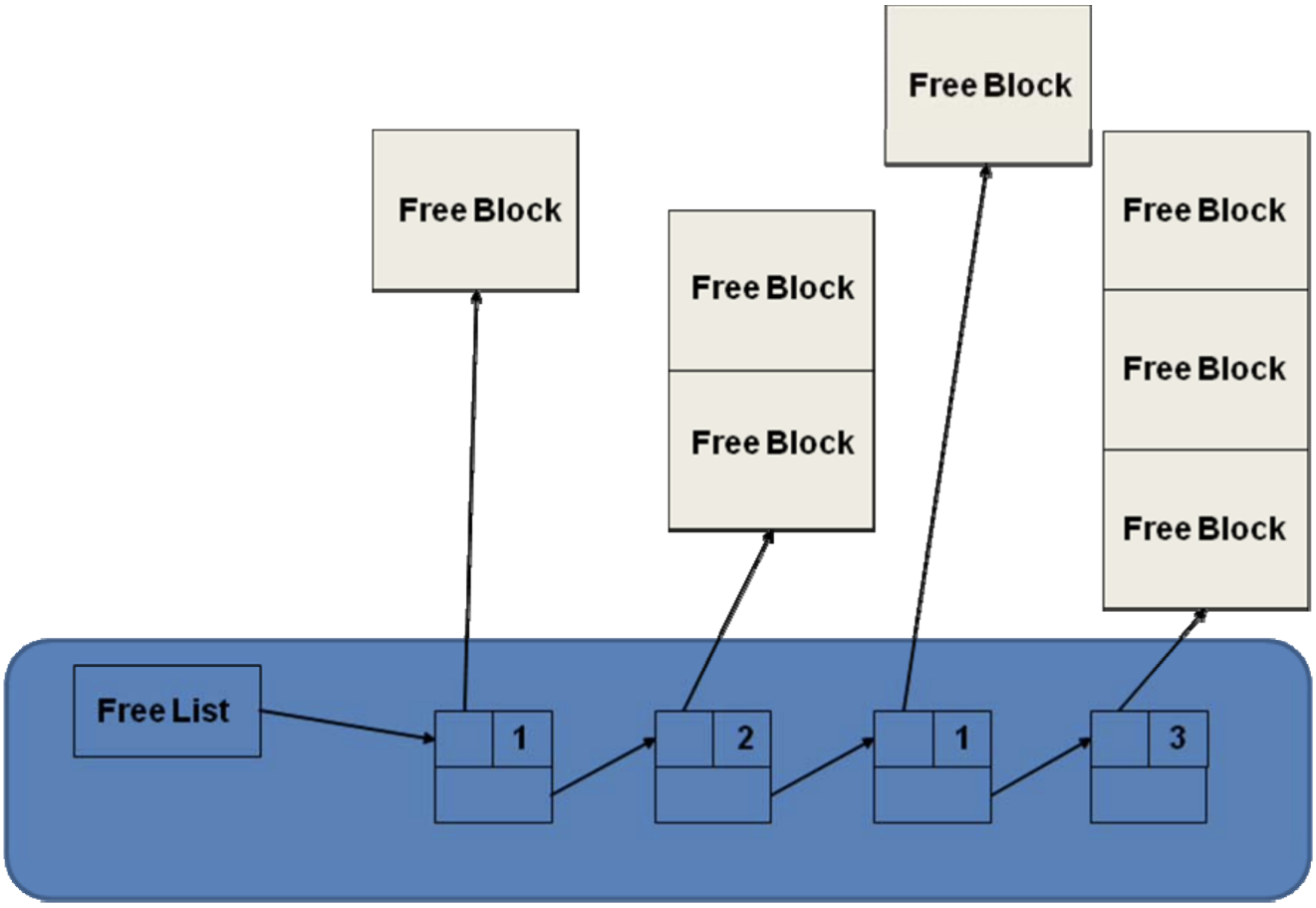


Figure 11.3: Free list with blocks pointing to disk addresses for contiguous allocation

Allocation of disk blocks to a new file may follow one of first fit or best fit policy. Deletion returns the blocks to the free list

- Compaction is not very efficient due to the overhead
- Suffers from external and internal fragmentation

The file system keeps this data structure (queue) both in persistent memory (the disk itself) and also caches it in memory for faster lookup

Problems

1. File cannot grow in size beyond the allocation
2. Significant internal and external fragmentation

Example

Given the following

```
Number of cylinders on the disk = 10000
Number of platters = 10
Number of surfaces per platter = 2
Number of sectors per track = 128
Number of bytes per sector = 256
Disk allocation policy = contiguous cylinders
```

a). How many cylinders are required to store a file of size 3 Mbytes?

$$\text{Number of tracks in a cylinder} = \text{Num platters} \cdot \text{Surfaces/Platter} = 10 \cdot 2 = 20$$

$$\text{Size of track} = \text{Sectors/Track} \cdot \text{Bytes per sector} = 128 \cdot 256 = 2^{15} \text{ bytes}$$

$$\text{Size of cylinder} = \text{Tracks/Cylinder} \cdot \text{Size of track} = 20 \cdot 2^{15} \text{ bytes}$$

$$\textbf{Num Cylinders} = \left\lceil \frac{3 \cdot 1024 \cdot 1024}{20 \cdot 2^{15}} \right\rceil = 5 \text{ cylinders}$$

b). How much internal fragmentation is there?

$$\textbf{Internal fragmentation} = 5 \cdot (20 \cdot 2^{15}) - (3 \cdot 2^{20}) = 131,072 \text{ bytes}$$

11.2.2 Contiguous Allocation with Overflow Area

This strategy is the same as the previous one with the difference that the file system sets aside an *overflow* region that allows spillover of large files that do not fit within the fixed partition

- Overflow region consists of physically contiguous regions allocated to accommodate such spillover of large files
- File system needs *an additional* data structure to manage the overflow region

11.2.3 Linked Allocation

In this scheme, the file system deals with allocation at the level of individual disk blocks

- We also use a free list of available disk blocks
- File occupies as many file blocks as it needs
- Free list may actually be a linked list of the disk blocks with each block pointing to the next free block on the disk

Insertion

We cache the location of the head and dequeue from head of free list

Deletion

Add disk blocks to free list

Performance

Using an actual LinkedList is rather slow traversing, so we often use a bit vector (one bit for each disk block. The block is free if the corresponding bit is 0 and busy if it is 1)

Difference To Contiguous Allocation

- No guarantee the blocks allocated are contiguous

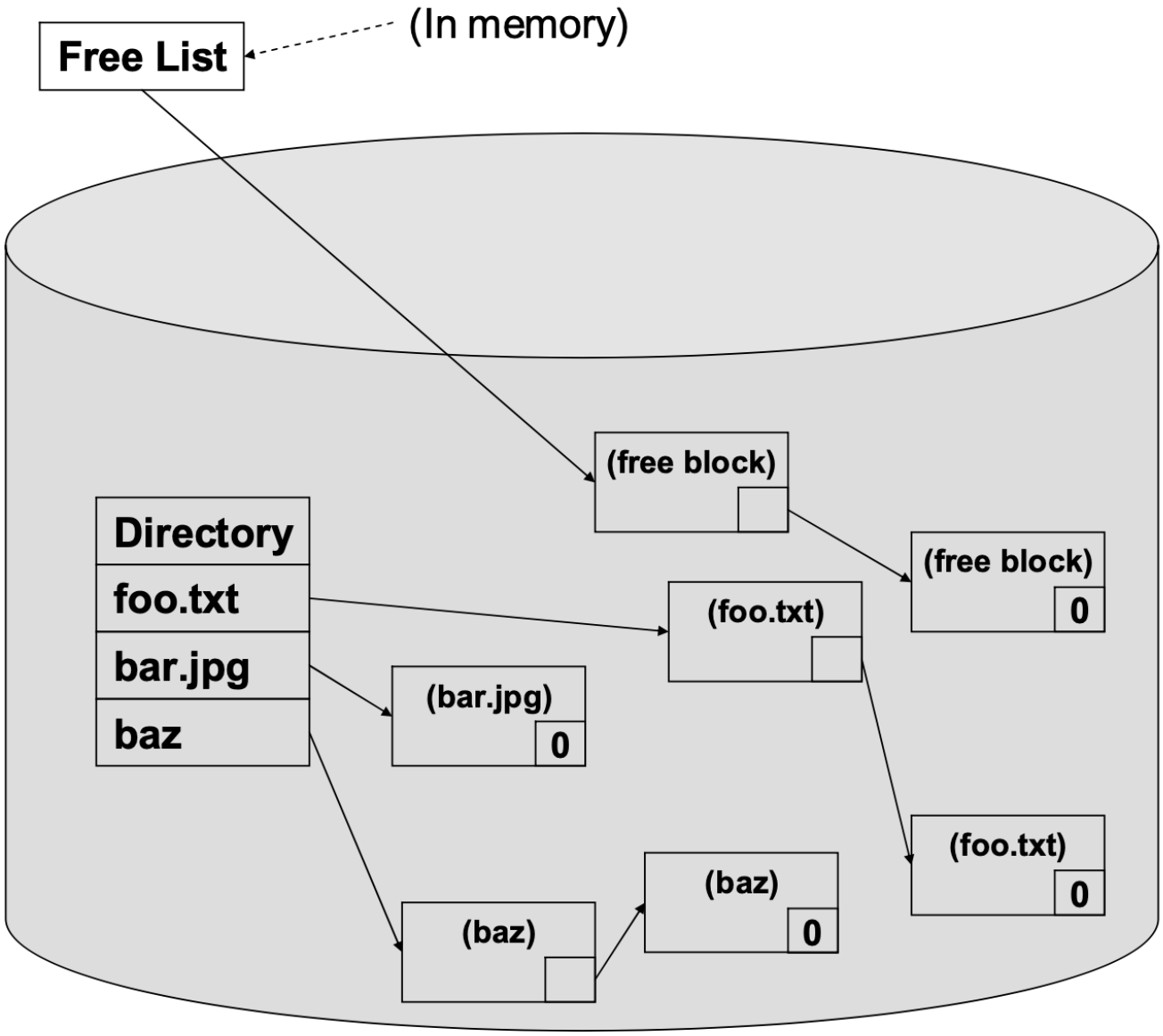


Figure 11.4: Linked Allocation

Advantages

- Fast allocation since it is one disk block at a time
- Accommodates easy growth of files
- No external fragmentation due to on-demand allocation
- No disk compaction

Disadvantages

- Slow random access (since disk blocks are not contiguous)
- Seek time is slow even for sequential (retrieving pointers takes time)
- Any bug in the list maintenance code leads to a complete breakdown of the file system

11.2.4 File Allocation Table (FAT)

Variation of Linked Allocation. A table on the disk, *File Allocation Table (FAT)*, contains the linked list of the files currently populating the disk

- Scheme divides disk into logical partitions
- Each partition has a FAT
- The *free/busy field* indicates the availability of that disk block (0 for free; 1 for busy)
- The *next field* gives the next disk block of the linked list that represents a file
- A *distinguished value* (`-1`) indicates this entry is the last disk block for that file
- A single directory for the entire partition contains the file name to FAT index mapping

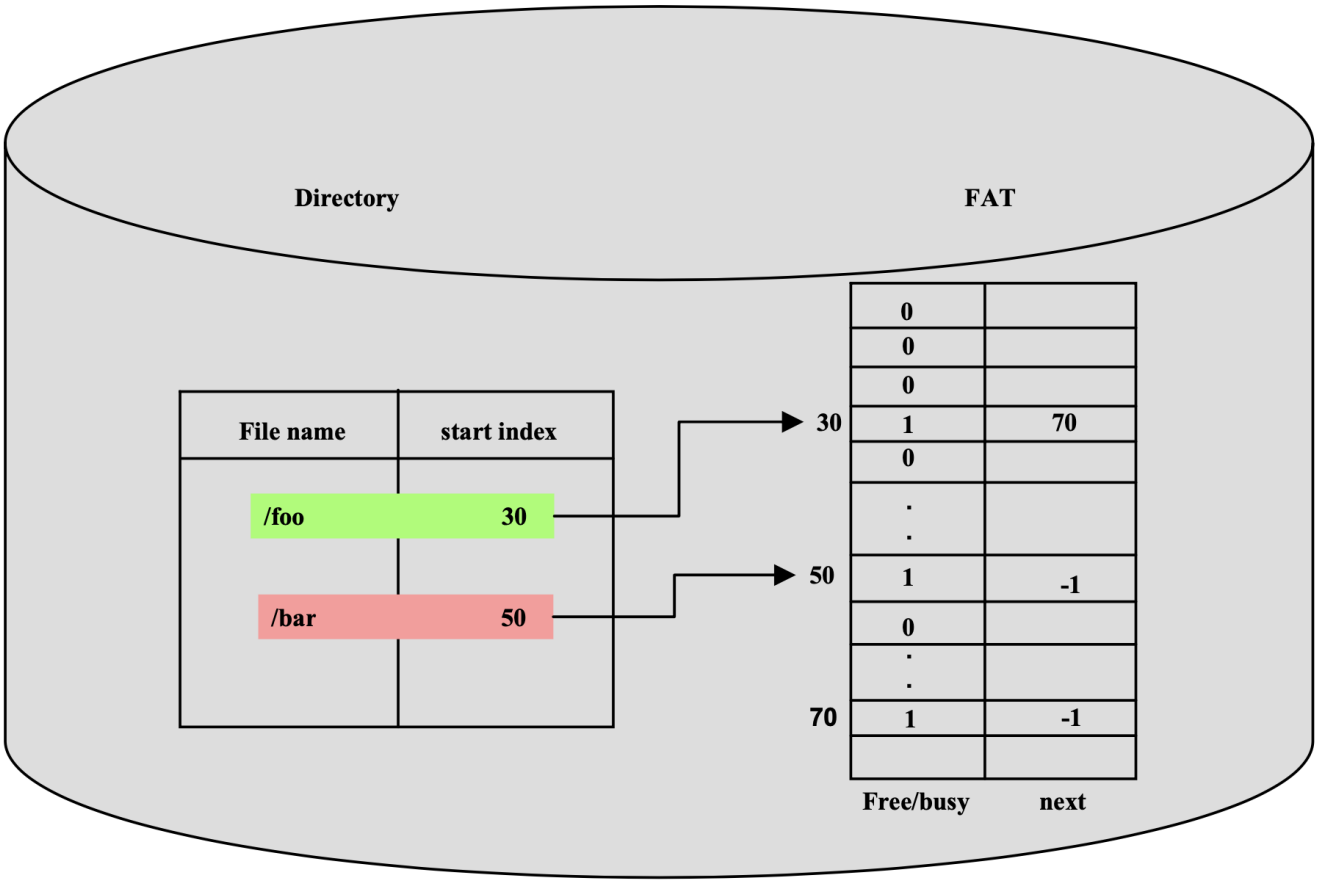


Figure 11.5: File Allocation Table (FAT)

For example, /foo occupies two disk blocks: 30 and 70. The next field of entry 30 contains the value 70, the address of the next disk block. The next field of entry 70 contains -1 indicating this is the last block for /foo. Similarly, /bar occupies one disk block (50). If /foo or /bar were to grow, the scheme will allocate a free disk block and fix up the FAT accordingly.

Advantages

- Less chance of errors compared to the linked allocation
- By caching FAT in memory, the scheme leads to efficient allocation times compared to linked allocation
- Performs similar to linked allocation for sequential file access (*arguably bad too*)
- Performs better than linked allocation for random access since the FAT contains the next block pointers for a given file

Disadvantages

- Logical partitioning leads to a level of management of the space on the disk for the end user that is not pleasant
- Creates an artificial scarcity of space on the disk even when there is physical space

---

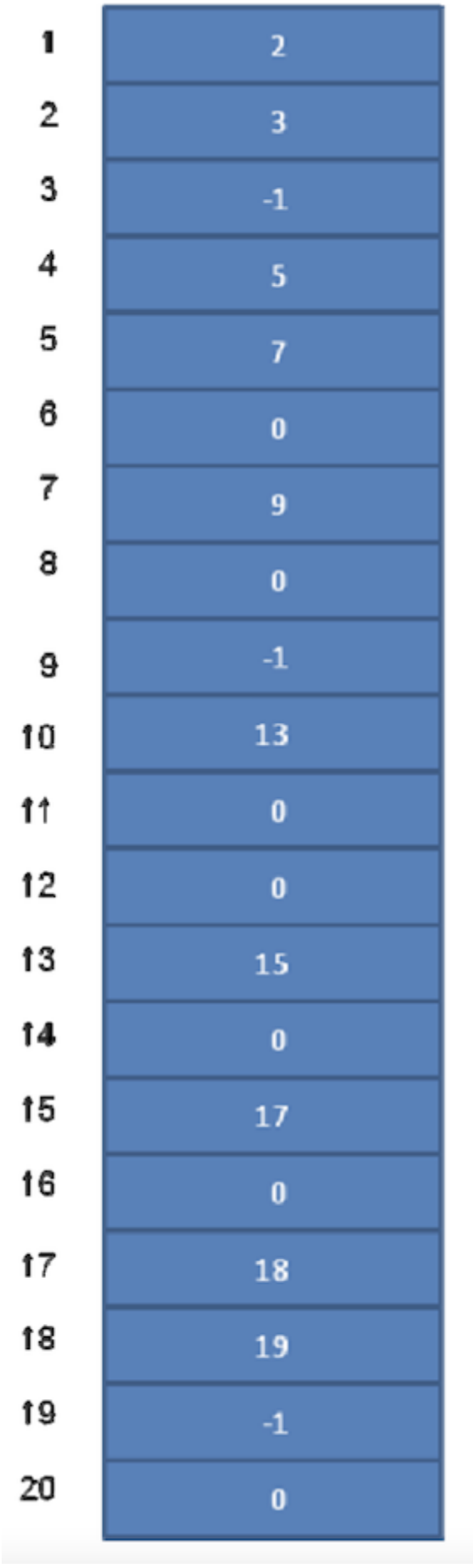
Example

`foo` occupies disk blocks 1, 2 and 3

`bar` occupies disk blocks 10, 13, 15, 17, 18 and 19

`gag` occupies disk blocks 4, 5, 7 and 9

Show the contents of the FAT (show the free blocks and allocated blocks per convention used in this section)


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
2
3
-1
5
7
0
9
0
-1
13
0
0
15
0
17
0
18
19
-1
0

11.2.5 Indexed Allocation

This scheme allocates an *index disk block* for each file

- Index block is a fixed-size data structure that contains addresses for data blocks that are part of that file
    - Essentially, this scheme aggregates data block pointers for a file scattered all over the FAT data structure into one table per file
- This table called *index node* or *i-node*, occupies a disk block
- The *directory* contains the the *file name* to *index node* mapping for each file
- Maintain a free list as a bit vector of blocks

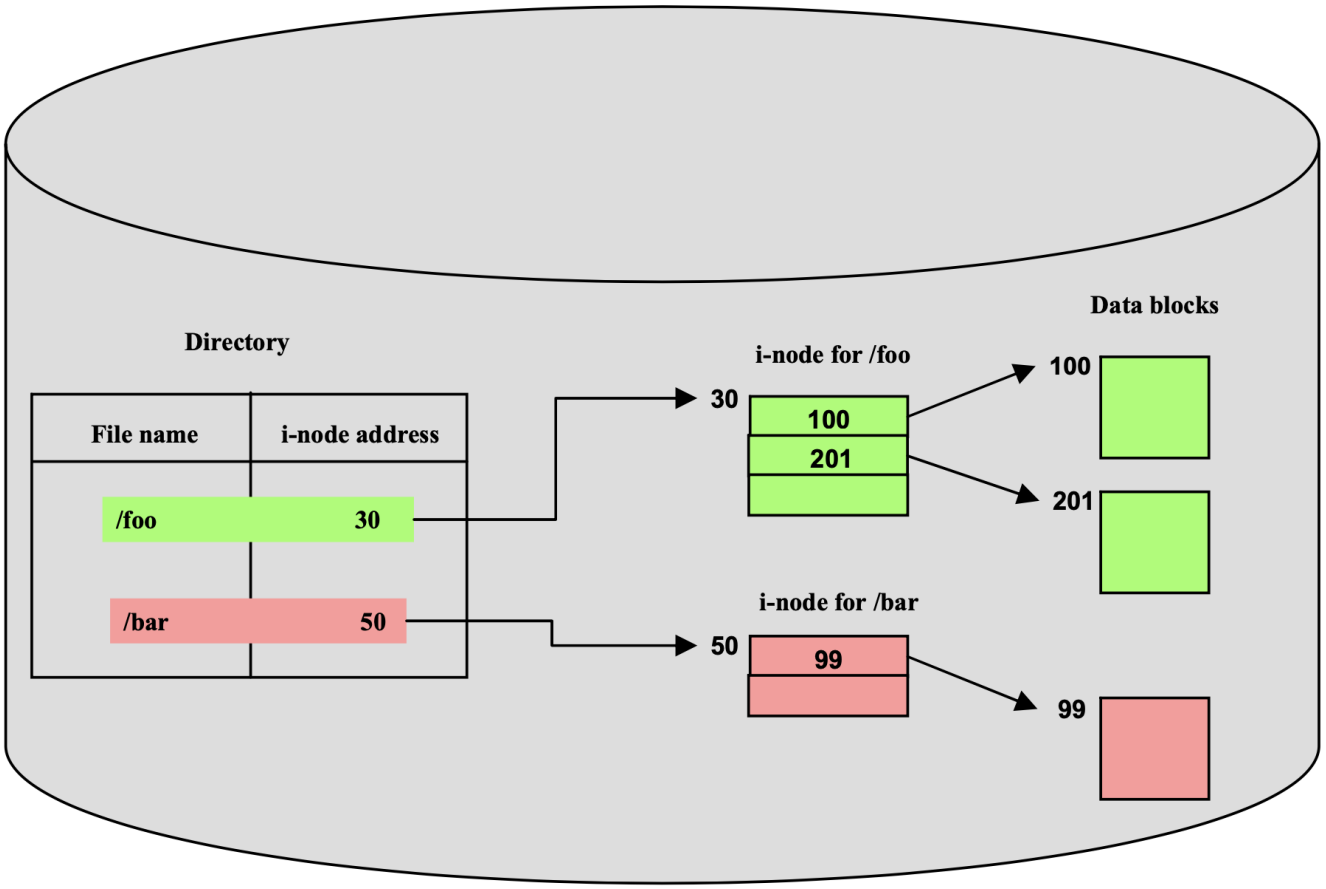


Figure 11.6: Indexed Allocation

Advantages

- Better random access time than FAT

Disadvantages

- Limit to max file size (i-node is a fixed size data structure per file with direct pointers to data blocks)
    - **Max file size = number of data block points in an i-node**

Calculations

Size of i-node = size of sector

Minimum amount of space for a file = size of i-node + size of data block

Number of data block pointers in an i-node = size of i-node / size of data block pointer

Maximum size of file = Number of data block pointers in i-node * size of data block

11.2.6 Multilevel Indexed Allocation

This scheme fixes the limitation in the indexed allocation by making the i-node for a file an indirection table

- With one level indirection, each i-node entry points to a first level table, which in turn points to the data blocks
- The scheme can be extended to make an i-node a two (or even higher) level indirection table depending on the size of the files that need to be supported by the file system

Disadvantages

- Small files that may fit in a few data blocks have to go through extra levels of indirection

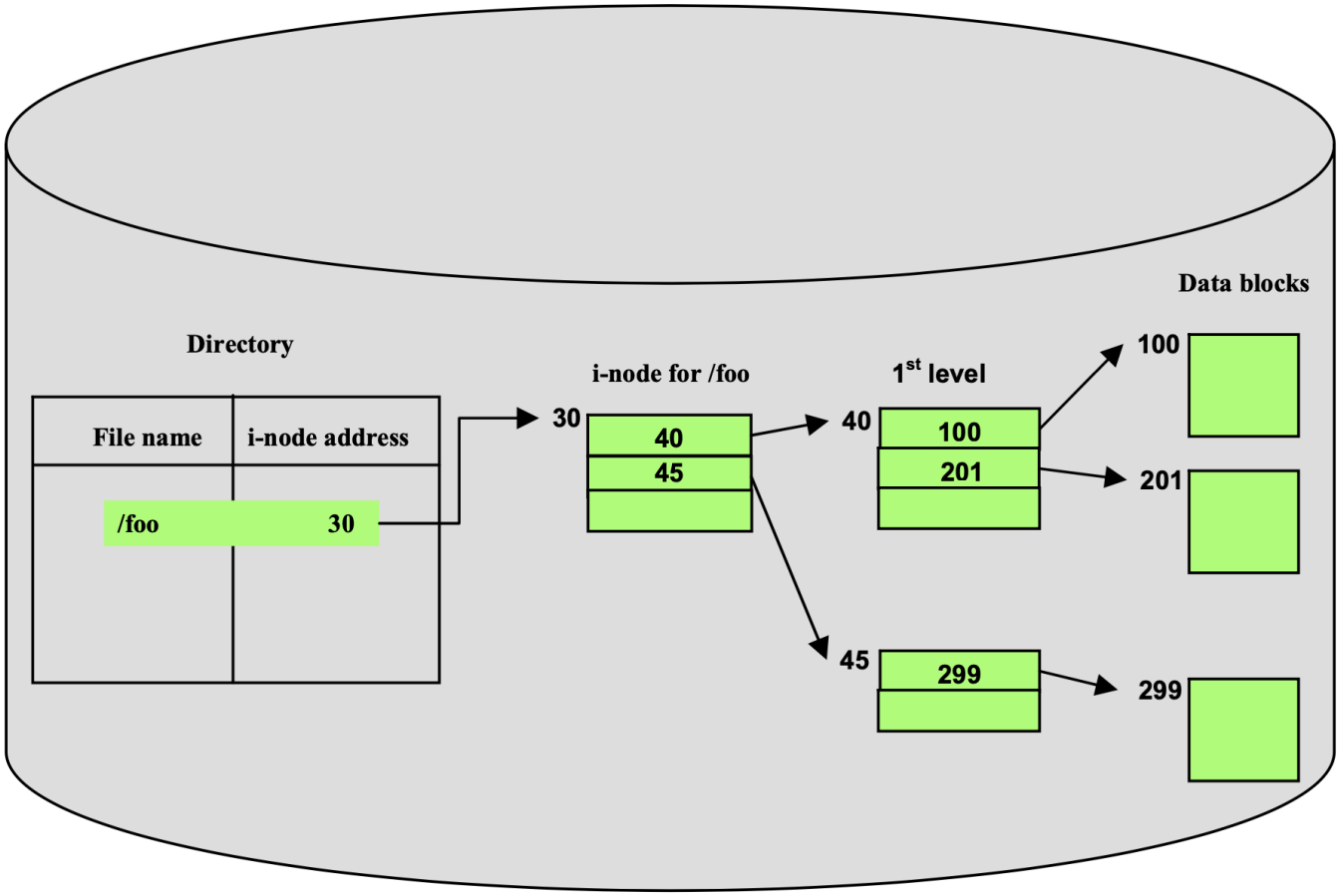


Figure 11.7: Multilevel Indexed Allocation (one level of indirection)

11.2.7 Hybrid Indexed Allocation

Combines the ideas from the previous two schemes

- Every file has an i-node
- Accommodates all the data blocks for small file with direct pointers
- If the file size exceeds the capacity of direct blocks, then the scheme uses single or more levels of indirection for the additional data blocks

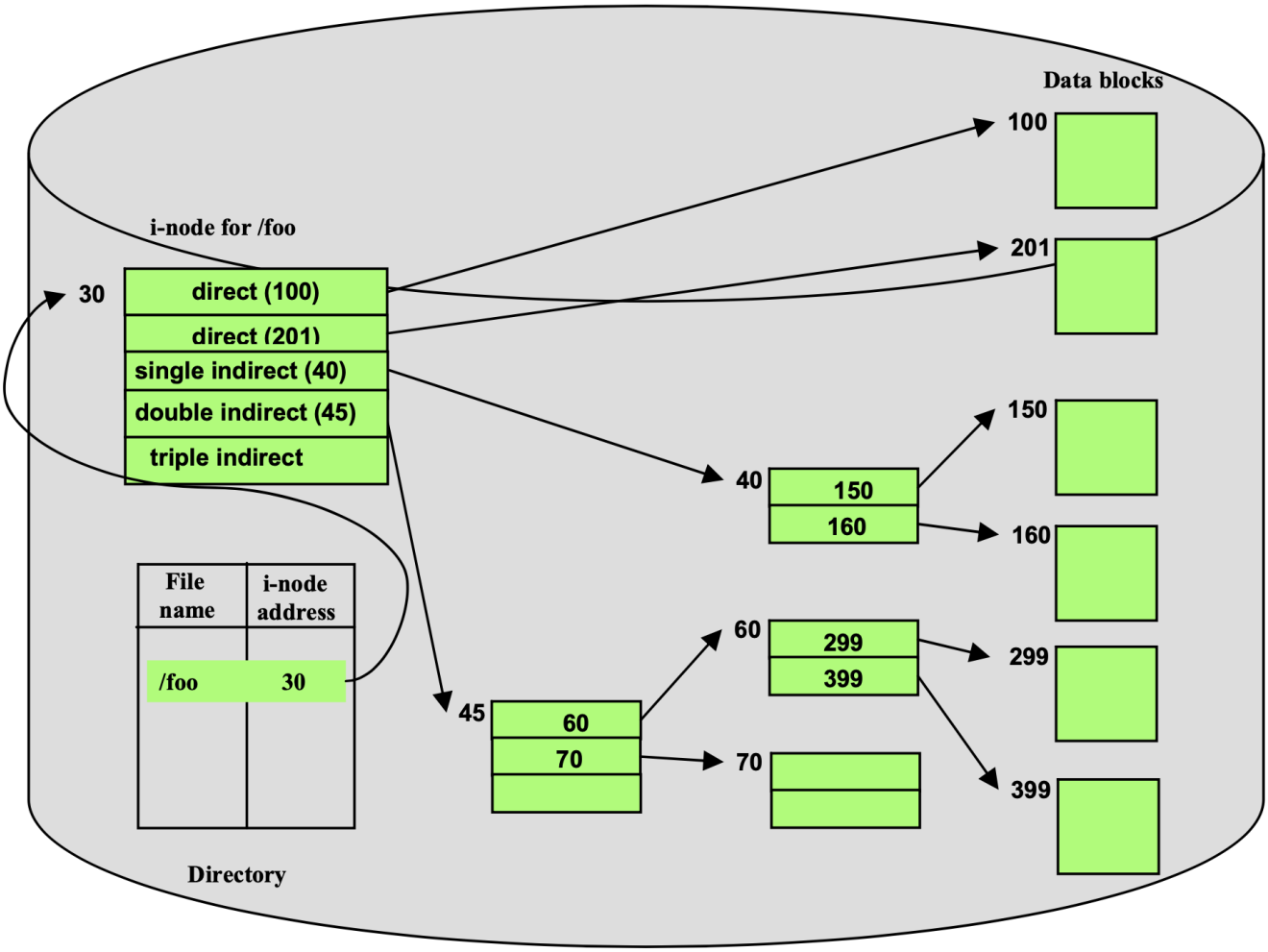


Figure 11.8: Hybrid Indexed Allocation

11.3 Putting It Together

UNIX uses a hybrid allocation approach with hierarchical naming

- To allow for aliasing, the i-node also includes a reference count field

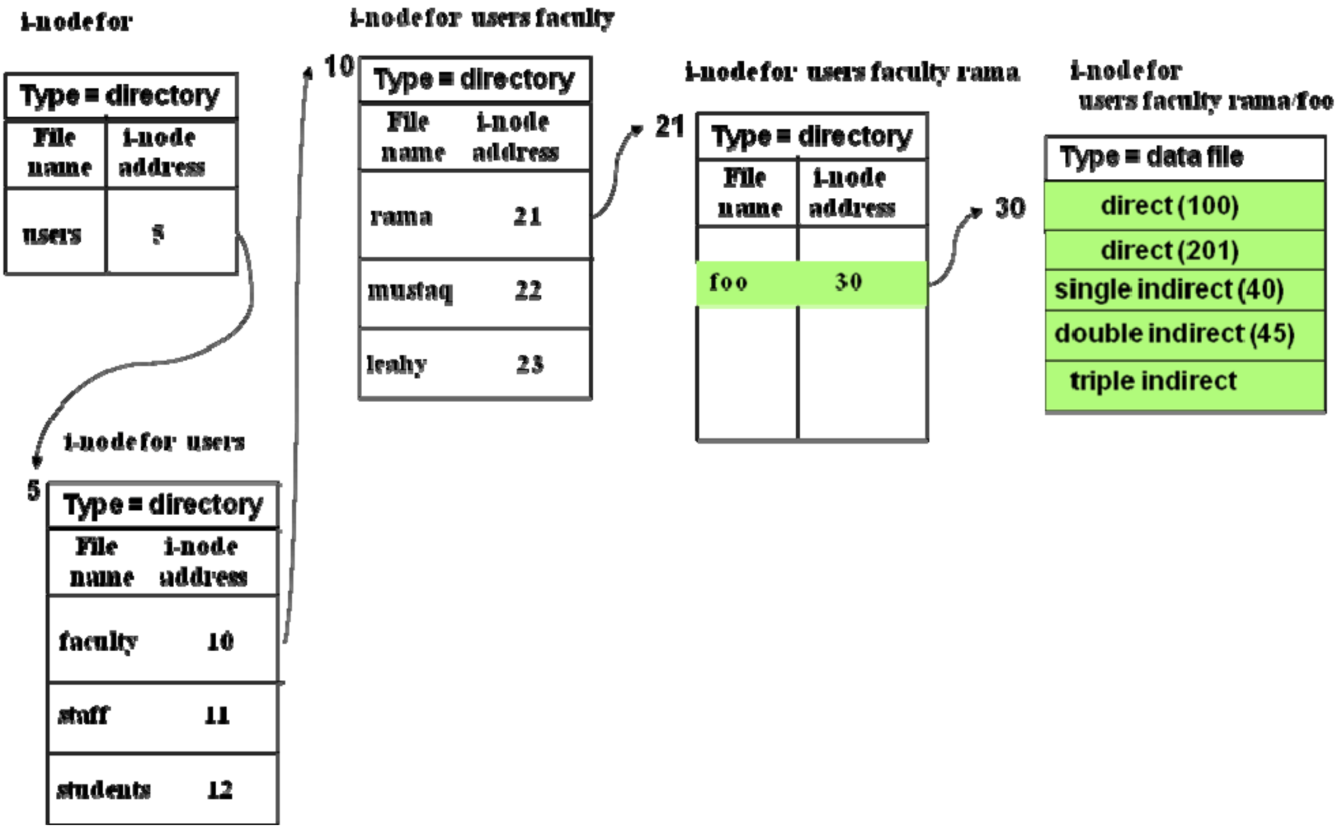


Figure 11.9: A simplified i-node structure for a hierarchical name in UNIX

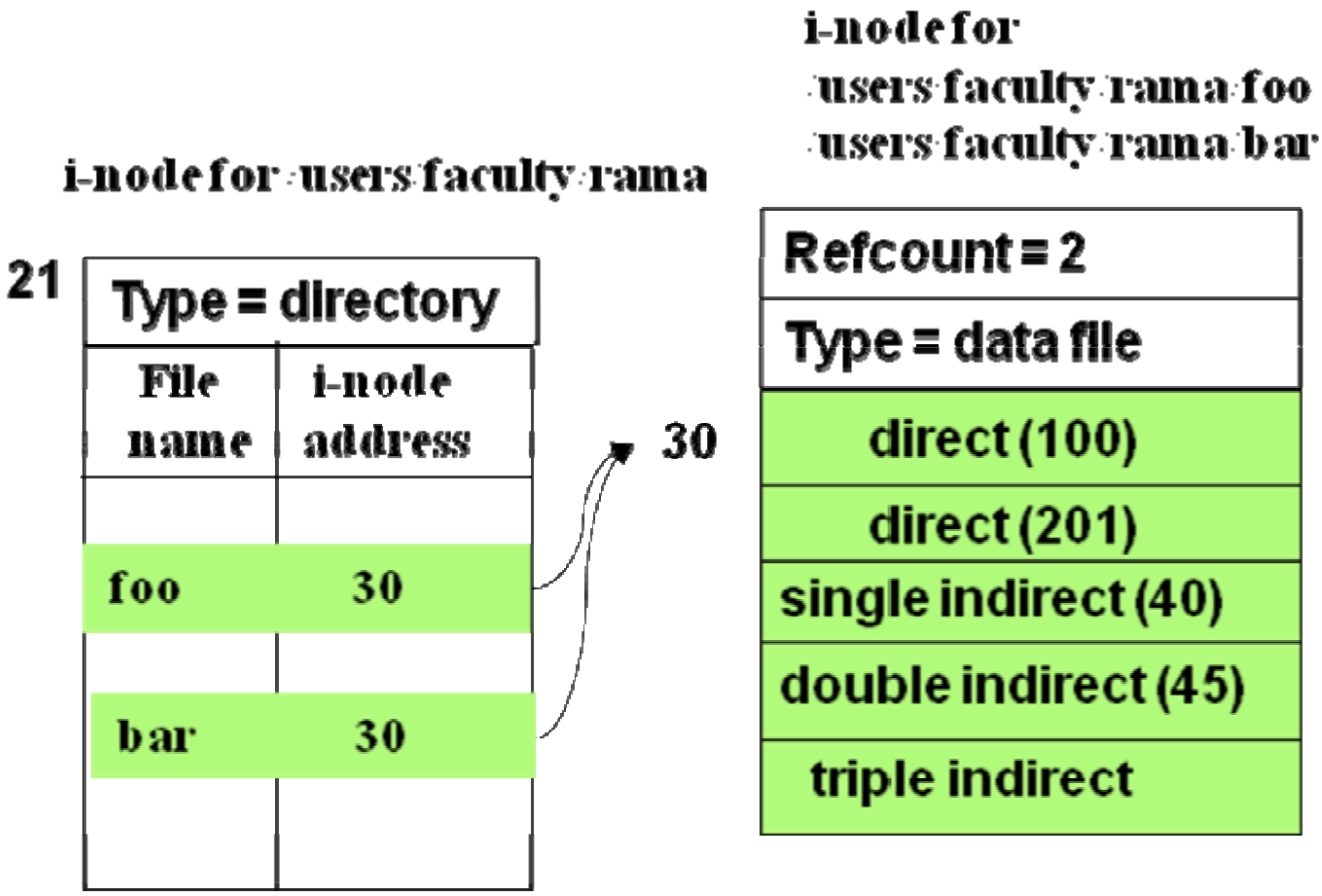


Figure 11.10: Two files foo and bar share an i-node since they are hard links to each other

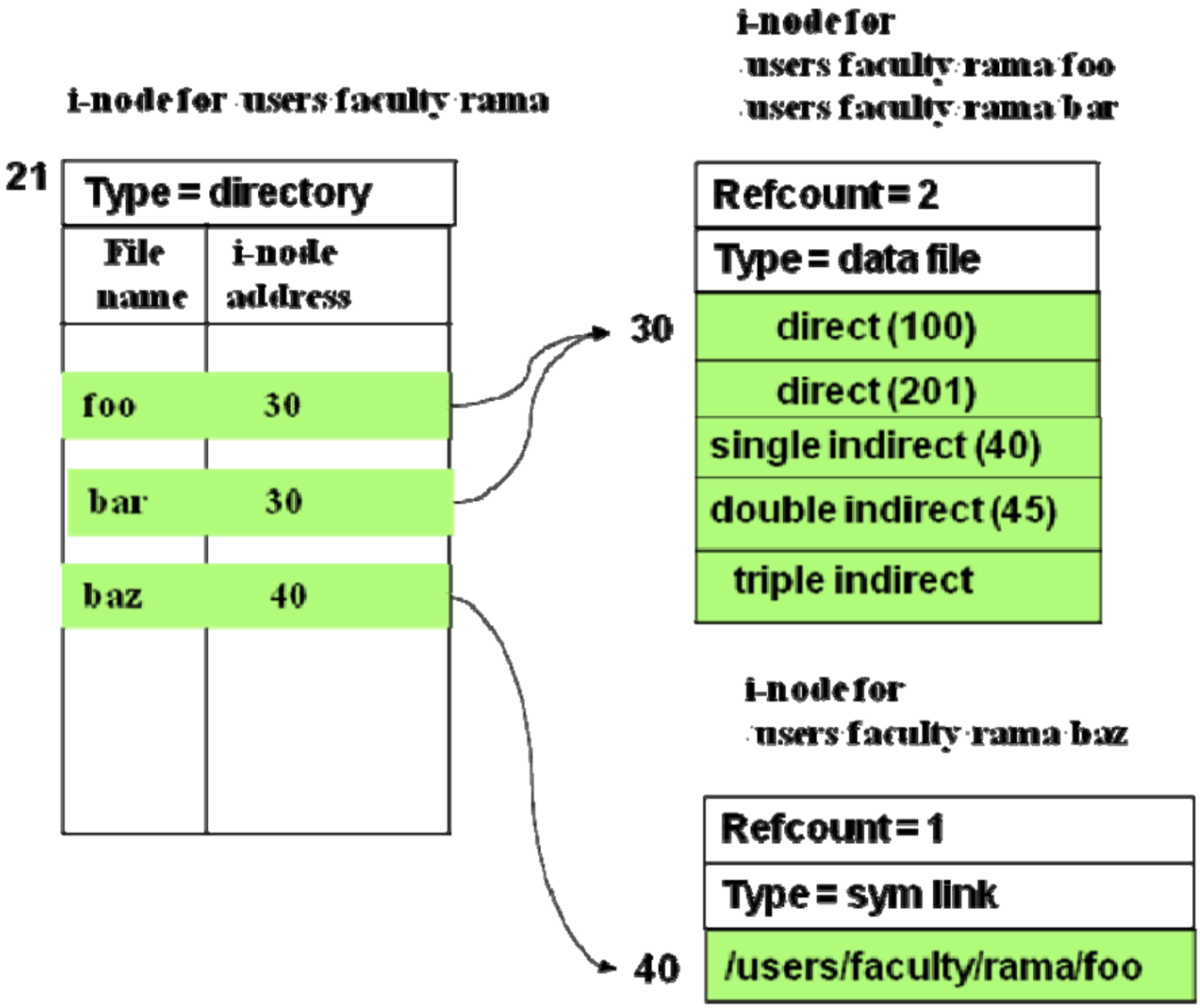


Figure 11.11: baz is a symbolic (soft) link to foo

11.4 Components of the File System

Media Independent Layer

- This layer consists of the user interface, i.e. the Application Program Interface (API) provided to the users
- Consists of the name resolver

Media Specific Storage Space Allocation Layer

- Embodies the space allocation (on file creation), space reclamation (on file deletion), free-list maintenance, and other functions associated with managing the space on the physical device

Device Driver

- Deals with communicating the command to the device and effecting the data transfer between the device and the operating system buffers

Media Specific Requests Scheduling Layer

- Scheduling requests from OS commensurate with the physical properties of the device

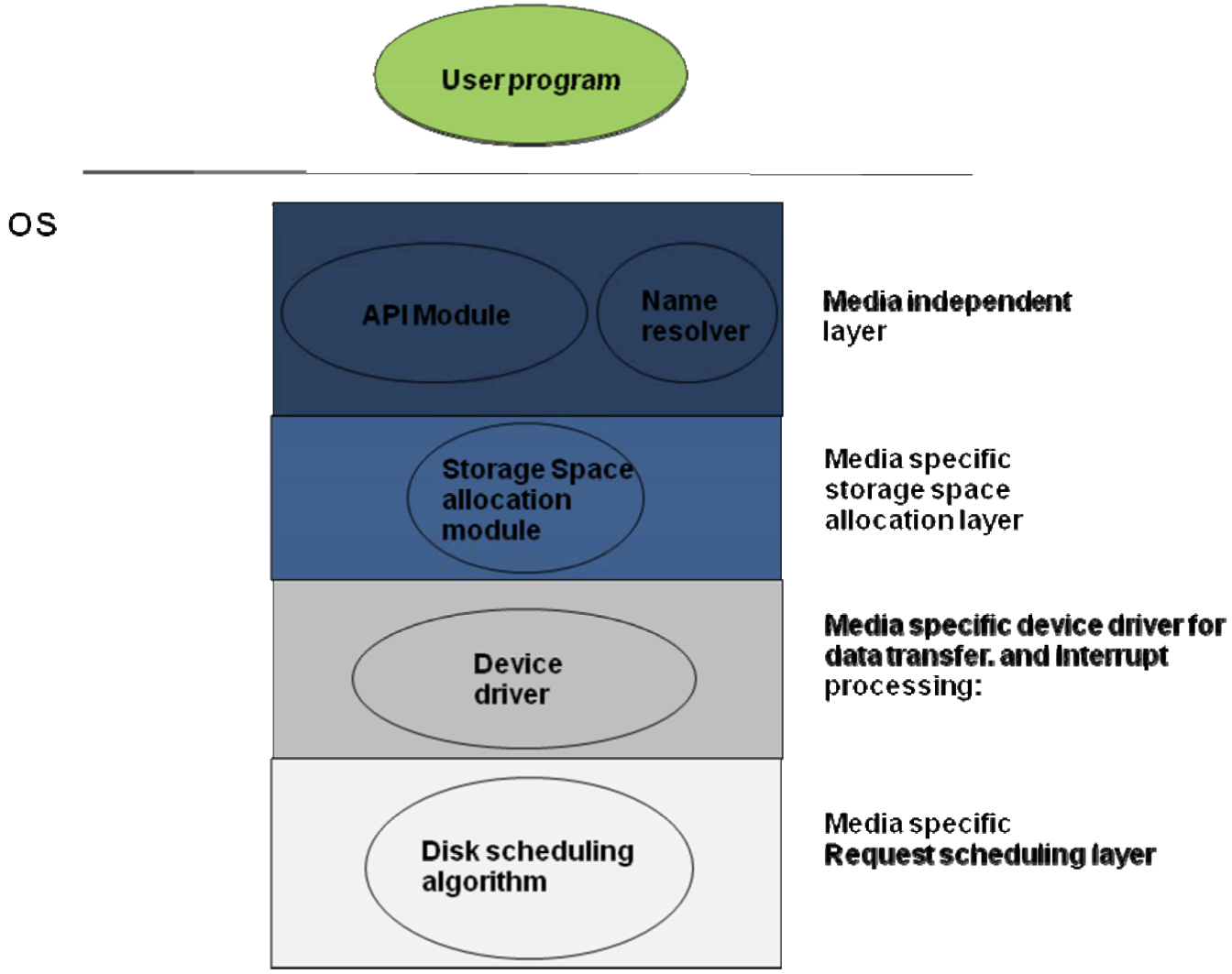


Figure 11.12: Layers of a disk-specific file system manager

Anatomy of Creating a File

1. The API routine for creating a file calls validates the call by checking the permissions, access rights, and other related information for the call. After such validation, it calls the name resolver.

2. The **name resolver** contacts the storage allocation module to allocate an i-node for the new file
3. The **storage allocation module** gets a disk block from the free list and returns it to the name resolver. The storage allocation module will fill in the i-node commensurate with the allocation scheme (see Section 11.2). Let us assume the allocation in effect is the hybrid scheme (Section 11.2.7). Since the file has been created without any data as yet, no data blocks would have been allocated to the file
4. The **name resolver** creates a directory entry and records the name to i-node mapping information for the new file in the directory

Notice that these steps do not involve actually making a trip to the device since the data structures accessed by the file system (directory and free list) are all in memory

Anatomy of Writing To A File

1. As before the API routine for file write will do its part in terms of validating the request.
2. The name resolver passes the memory buffer to the storage allocation module along with the i-node information for the file.
3. The storage allocation module allocates data blocks from the free list commensurate with the size of the file write. It then creates a request for disk write and hands the request to the device driver.
4. The device driver adds the request to its request queue. In concert with the disk-scheduling algorithm, the device driver completes the write of the file to the disk.
5. Upon completion of the file write, the device driver gets an interrupt from the disk controller that is passed back up to the file system, which in turn communicates with the CPU scheduler to continue the execution of your program from the point of file write

As far as the OS is concerned, the file write call is complete **as soon as the request is handed to the device driver**

- Success/failure will be interrupt

11.5 Interaction among the various subsystems

We got too many subsystems in an OS running (e.g. VM Manager, Scheduler, ... etc.). We just upcall stuff until it works.

11.6 Layout of the file system on the physical media

How does BIOS know where to even find the OS? What about which OS to load?

- Master Boot Record (MBR) knows what to do
- BIOS just finds MBR and executes it

If you have multiple OSes (like me), you have to partition

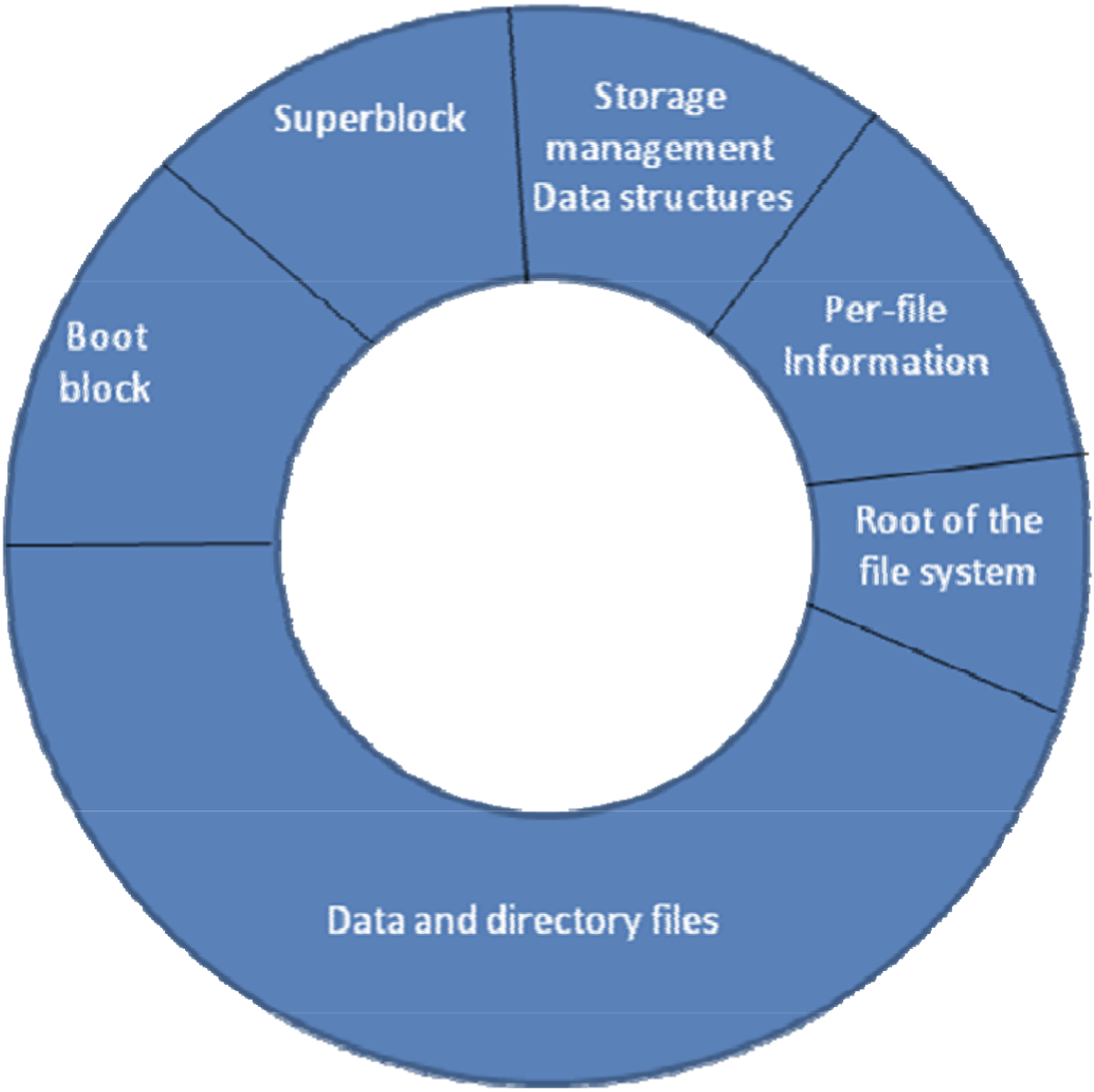


Figure 11.13: Layout of each partition

Superblock contains all the information pertinent to the file system contained in this partition

- The boot program (in addition to loading the operating system or in lieu thereof) reads in the superblock into memory
- Typically contains a code usually referred to as the magic number that identifies the type of file system housed in this partition, the number of disk blocks, and other administrative information pertaining to the file system
- If superblock is corrupted, you're fucked

I/O Issues

Users typically will create a bunch of intermediary files, so it's not optimal to always write to I/O devices. Thus, the OS will wait and then flush the files to update the file system (the reason why you have "it's not safe to remove the device")

11.7 Dealing With System Crashes

In UNIX, the operating system automatically runs *file system consistency check (fsck)* on boot up

- Makes sure in-memory data structures and on-disk versions are the same

There's also more stuff like Virtual File Systems, NTFS, and Unix