DCS216 Operating Systems

Lecture 27
File Systems (3)

Jun 17th, 2024

Instructor: Xiaoxi Zhang
Sun Yat-sen University

Content

Content

- File-System Structure
  - File-System Layers
  - On-Storage Structures
  - In-Memory Structures
  - Partitions and Mounting
  - Virtual File Systems
- The Very Simple File System (VSFS)
- Directory Implementation
- Allocation Methods
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation
- Free-Space Management
- Efficiency and Performance
- Recovery

Directory Implementation

- As mentioned earlier, a directory is simply a **logical** data structure that contains mappings of <`filename`, `file_number`>.
  - The directory itself is also a special type of **file**. It has an **inode** and the data content associated with the **inode**.
- But how is a directory actually implemented? In other words, how is the mappings of <`filename`, `inode_num`> stored in a directory?

```
$ ls -ali .
total 76
8567717 drwxrwxr-x  2 ubuntu ubuntu  4096 Jun  4 19:59 .
7236895 drwxrwxr-x 23 ubuntu ubuntu  4096 Jun  4 11:03 ..
8567720 -rw-rw-r--  1 ubuntu ubuntu     6 Jun  4 12:00 bar
8567875 -rw-rw-r--  1 ubuntu ubuntu  1781 Jun  4 19:17 file_io_syscall.c
8567901 -rw-rw-r--  1 ubuntu ubuntu     6 Jun  4 17:18 foo.txt
8567719 -rw-rw-r--  1 ubuntu ubuntu    80 Jun  4 11:33 hello.c
8567878 -rwxrwxr-x  1 ubuntu ubuntu 18600 Jun  4 14:13 lowio_creat
8567718 -rw-rw-r--  1 ubuntu ubuntu   257 Jun  4 14:11 lowio_creat.c
8567879 -rwxrwxr-x  1 ubuntu ubuntu 17632 Jun  4 16:25 lowio_lseek
8567721 -rw-rw-r--  1 ubuntu ubuntu   467 Jun  4 16:24 lowio_lseek.c
8567877 -rw-rw-r--  1 ubuntu ubuntu   153 Jun  4 14:11 Makefile
```

inode

file_name

Directory Implementation

Linear List

- The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
- For example

```
reclen | inum | strlen | name
  12       5       2     .
  12       2       3     ..
  12      19       4     foo
  12     255       4     bar
  36     258      28     foobar_is_a_pretty_longname
```

Directory Implementation

Linear List

- The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
- For example

```
reclen | inum | strlen | name
  12        5        2     .
  12        2        3     ..
  12       19        4     foo
  12      255        4     bar
  36      258       28     foobar_is_a_pretty_longname
```

12	5	0	2	.	\0	?	?	?	?	?	?

12	2	0	3	.	.	\0	?	?	?	?	?

12	19	0	4	f	o	o	\0	?	?	?	?

12	FF	0	4	b	a	r	\0	?	?	?	?

36	2	1	28	f	o	o	b	a	r	_	i

s	_	a	_	p	r	e	t	t	y	_	l

o	n	g	n	a	m	e	\0	?	?	?	?

Directory Implementation

Linear List

- The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
- For example

```
reclen | inum | strlen | name
  12        5       2      .
  12        2       3      ..
  12       19       4      foo
  12      255       4      bar
  36      258      28      foobar_is_a_pretty_longname
```

12	5	0	2	.	\0	?	?	?	?	?	?

12	2	0	3	.	.	\0	?	?	?	?	?

12	19	0	4	f	o	o	\0	?	?	?	?

12	FF	0	4	b	a	r	\0	?	?	?	?

36	2	1	28	f	o	o	b	a	r	_	i
s	_	a	_	p	r	e	t	t	y	_	l
o	n	g	n	a	m	e	\0	?	?	?	?

For simplicity, default record length is 12 bytes for normal files (file name length less than 8); the max **inum** is 65535 (addressable with 2 bytes); variable length filename is supported by specifying larger reclen.

Directory Implementation

- **Linear List**
  - The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
  - **Advantages**: Simple to implement.

Directory Implementation

- **Linear List**
  - The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
  - **Advantages**: Simple to implement.
  - To **create** a new file in the directory:
    - Search the directory to be sure that no existing file has the same name.
    - Add a new entry `<filename, inode_num>` at the end of the directory.
  - To **delete** an existing file from the directory:
    - Search the directory for the specific file.
    - Release the space allocated to that file.

Directory Implementation

Linear List

- The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
- **Advantages:** Simple to implement.
- **Disadvantages:** Time-consuming to operate (linear search time)
  - For example, in a directory with many many number of files (e.g., 60,000 files), the system will have to iterate through all of the files in order to locate the file that we wish to `open()`.

```
reclen | inum | strlen | name
  12        5        2      .
  12        2        3      ..
  12       19        4      foo
  12      255        4      bar
  36      258       28      foobar_is_a_pretty_longname
 ...
  12    65535        7      lastone
```

Directory Implementation

Linear List

- The simplest method of implementing a directory is to use a linear list of **file names** with pointers (**inode number**) to the data blocks.
- **Advantages**: Simple to implement.
- **Disadvantages**: Time-consuming to operate (linear search time)
  - **Workaround**: we could keep the **linear list ordered** (e.g., **sorted** by filename), to reduce the search time from $O(N)$ to $O(logN)$ with **Binary Search**. However, the insertion time is increased from $O(1)$ to $O(N)$.

```
reclen | inum | strlen | name
  12        5        2     .
  12        2        3     ..
  12      255        4     bar
  12       19        4     foo
  36      258       28     foobar_is_a_pretty_longname
  12    65535        7     lastone
 ...
```

Directory Implementation

- **Hash Table**
  - Linear list with **hash** data structure.
    - The hash table takes a value computed from the filename and returns a pointer to the filename in the linear list.
  - Decrease directory search time to $O(1)$.

Directory Implementation

Hash Table

- Linear list with **hash** data structure.

```
Hash function: h(key) = key % 16
```

```
filename
.
..
foo
bar
foobar_is_a_pretty_longname
lastone
```

```
key
46
46 + 46 = 92
'f' + 'o' + 'o' = 324
'b' + 'a' + 'r' = 309
2859
758
```

	key	filename
00		
01		
02		
03		
04	324	foo
05	309	bar
06	758	lastone
07		
08		
09		
10		
11	2859	foobar_is_a_pretty_longname
12	92	..
13		
14	46	.
15		

Directory Implementation

- **Hash Table**
  - Linear list with **hash** data structure.
    - The hash table takes a value computed from the filename and returns a pointer to the filename in the linear list.
  - Decrease directory search time to $O(1)$.
  - **Problems:**
    - **Collisions** – situations where two filenames hash to the same location.
    - Hash tables are generally fixed size and the hash functions depends on that size.
      - Only good if entries are fixed size, or use chained-overflow method.

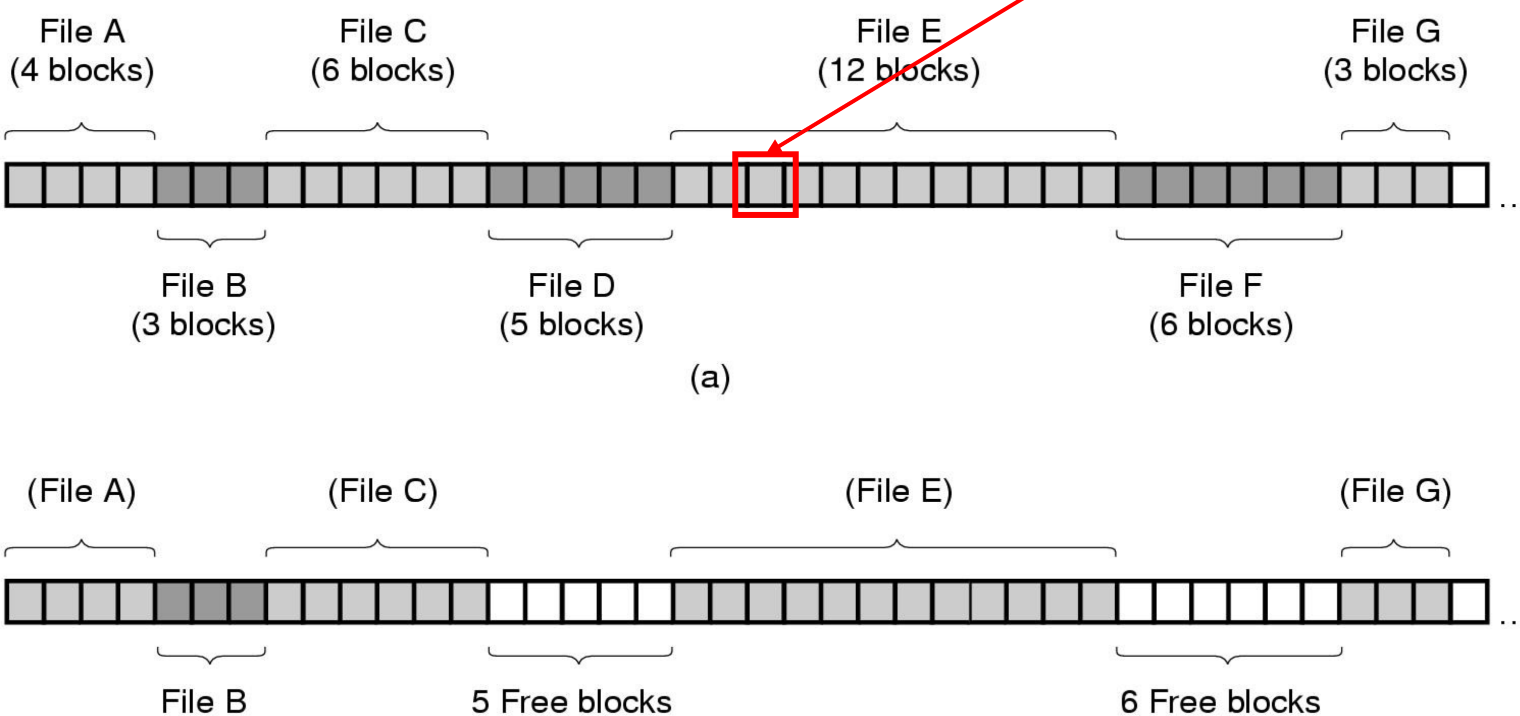
Allocation Methods

- An allocation method refers to how **disk blocks** are **allocated** for files.
  - **Contiguous** Allocation (**连续**分配)
  - **Linked** Allocation (**链接**分配)
  - **Indexed** Allocation (**索引**分配)

Allocation Methods

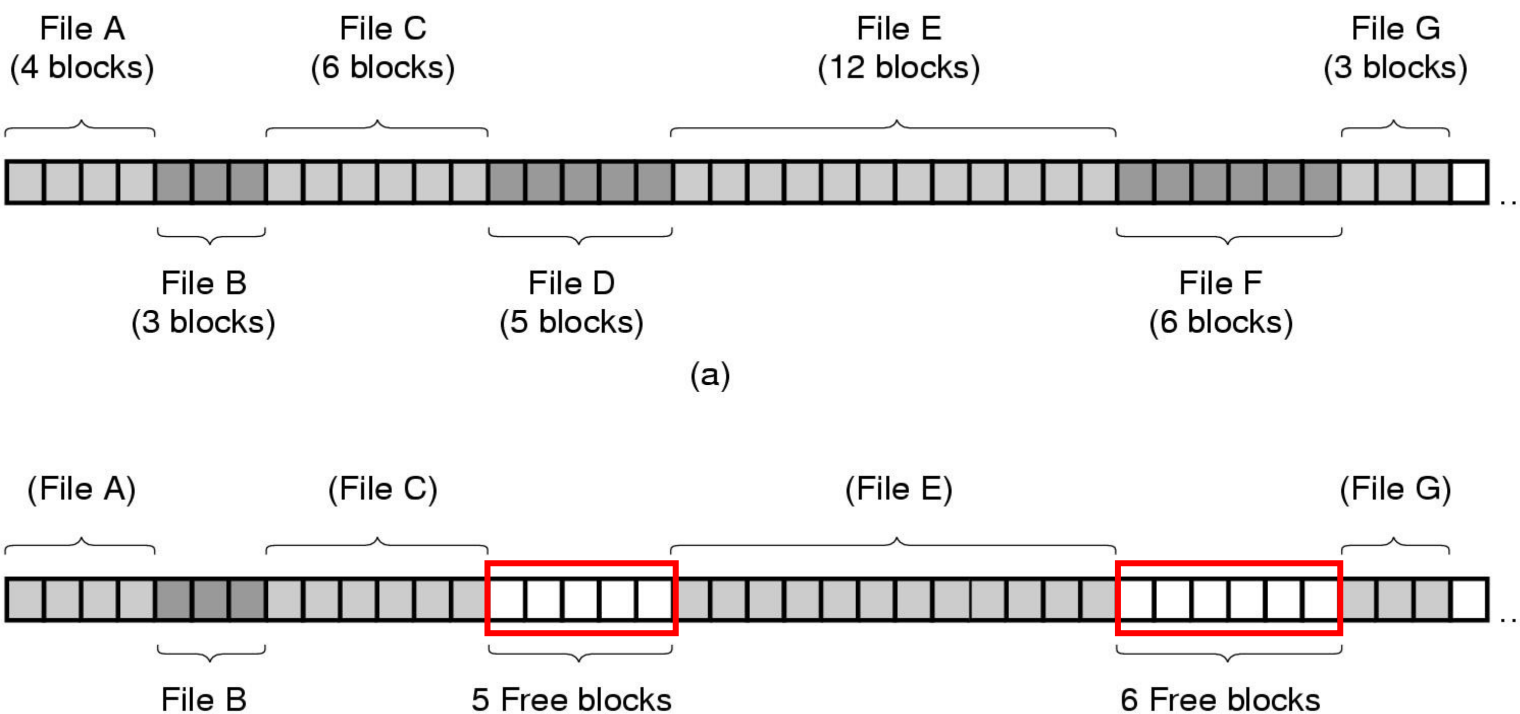
Contiguous Allocation

- Each file occupies a set of **contiguous** blocks on the disk.
  - **Simple**: only starting location (block #) and length (# of blocks) required.
  - Enables **random access**.
    - E.g., can access the 3rd **block** of **File E** directly via `(&E + 2)`.
  - Best **performance** *in most cases*.



Contiguous Allocation

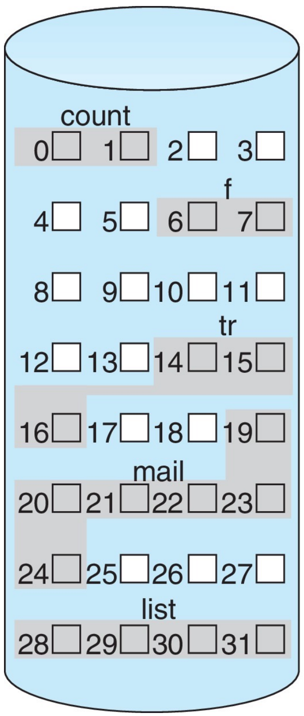
- **Problems** with **contiguous allocation**:
  - Finding space on the disk for a new file.
  - Knowing file size.
  - **External Fragmentation**
    - need for **compaction** **off-line** (downtime) or **on-line** (file corruption).



Allocation Methods

Contiguous Allocation

- Mapping from **Logical Address** to **Physical Address**
  - Assume block size == 512 bytes
  - `LA / 512 = Q`
    `LA % 512 = R`
  - Block number to be accessed:
    `starting_addr + Q`
  - Displacement (`offset`) into block:
    `R`



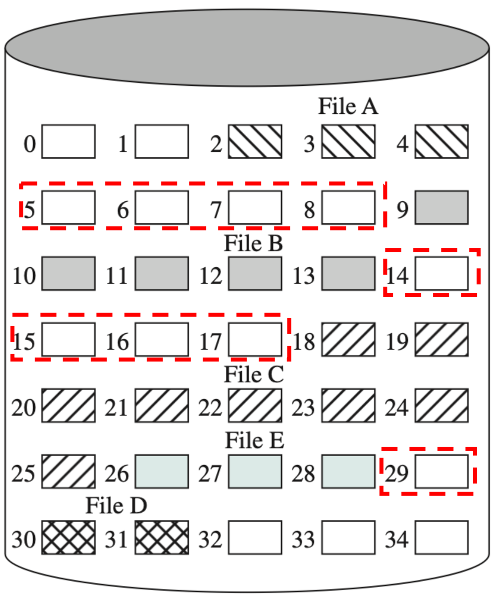
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation Methods

Contiguous Allocation

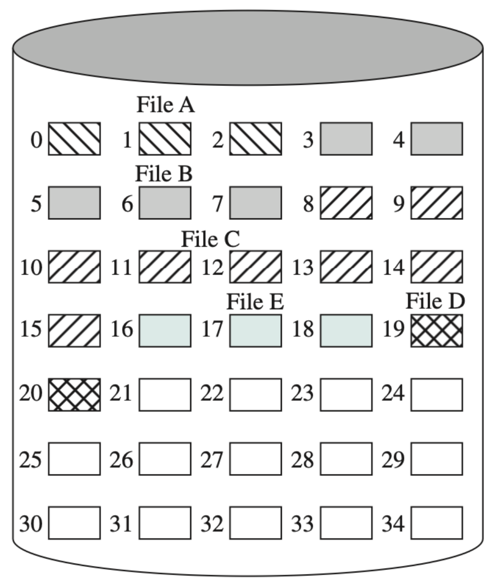
- The need for **compaction (碎片合并/压缩)**.
  - As files are allocated and deleted, **external fragmentation** is inevitable.
  - **Holes** (fragments) between allocated files may be unusable, leading to significant loss of storage space.
  - One strategy to prevent or reduce such loss is to regularly **compact** all free spaces into one contiguous space.



File allocation table

File name	Start block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.9 Contiguous File Allocation



File allocation table

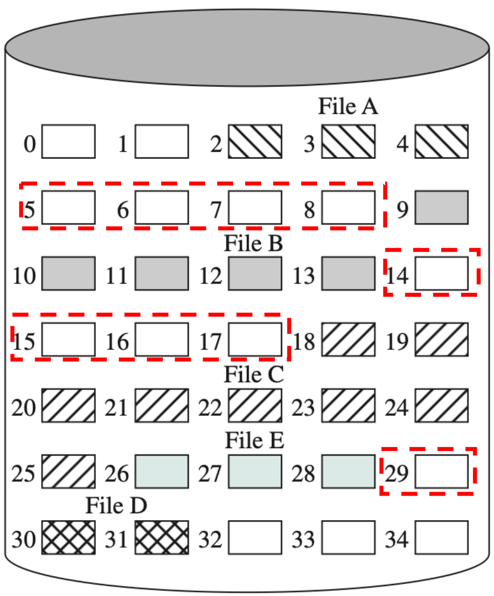
File name	Start block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.10 Contiguous File Allocation (After Compaction)

Allocation Methods

Contiguous Allocation

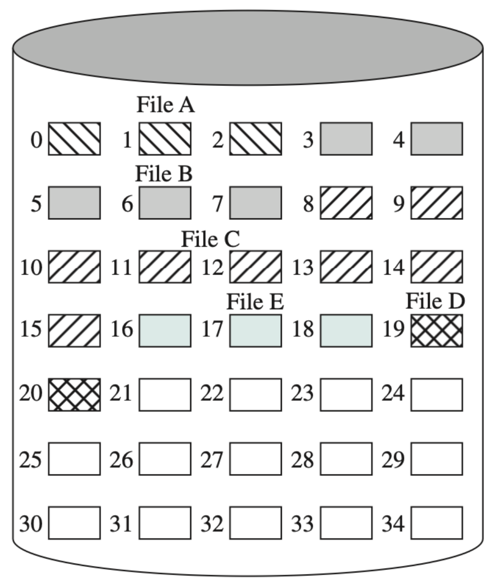
- The need for **compaction (碎片合并/压缩)**.
  - Compaction can be done **off-line** or **on-line**.
    - **off-line (停机期间)**: the file system must be unmounted (卸载).
    - **on-line (在线)**: special care must be taken when **compacting** (**defragmentation**) on-line, otherwise data corruption may occur.



File allocation table

File name	Start block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.9 Contiguous File Allocation



File allocation table

File name	Start block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.10 Contiguous File Allocation (After Compaction)

Contiguous Allocation

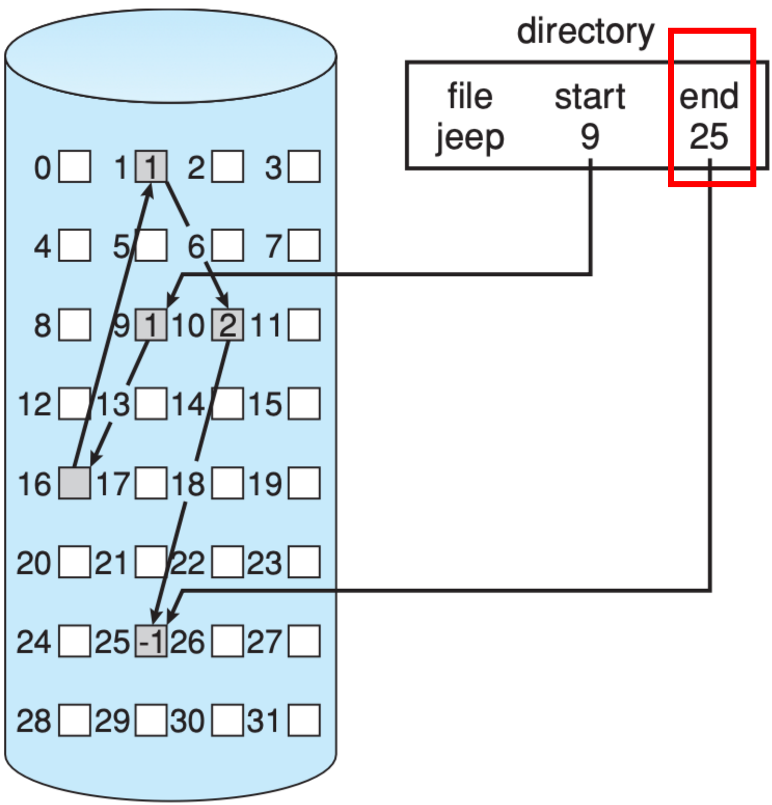
- **Extent**-Based File Systems
  - Many newer file systems (SPARC Veritas File System, Linux 2.6.19 ext4, etc.) use a modified version of contiguous allocation scheme.
  - In an extent-based file system, a contiguous chunk of space is allocated **initially** for file allocation.
  - Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added.
    - The location of a file's blocks is then recorded as a location and a block count, **plus** a link to the first block of the next **extent**.
    - A file may consist of **one or more** **extents**.

Linked Allocation

- **Linked Allocation** (also called **chained allocation**) solves all problems of contiguous allocation.
  - Each file is a **linked list** of storage blocks.
  - These blocks may be scattered anywhere on the disk.
  - Each block contains a pointer to the next block.
  - **End-of-File** could be achieved by:
    - Ending with a **NULL** pointer.

Linked Allocation

- **Linked Allocation** (also called **chained allocation**) solves all problems of contiguous allocation.
  - Each file is a **linked list** of storage blocks.
  - These blocks may be scattered anywhere on the disk.
  - Each block contains a pointer to the next block.
  - **End-of-File** could be achieved by:
    - Ending with a **NULL** pointer.
    - Explicitly specify an **END** block.

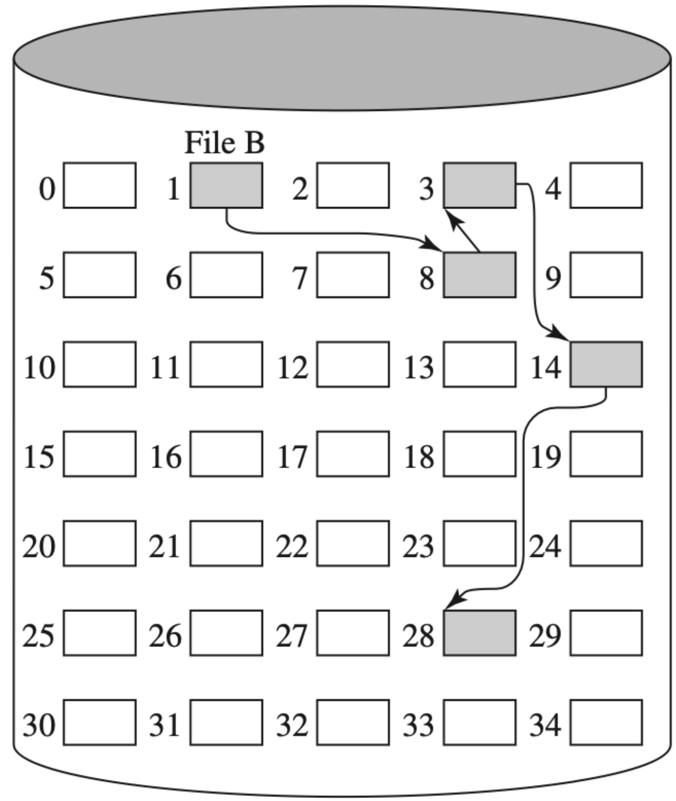


Allocation Methods

- **Linked Allocation**
  - **Linked Allocation** (also called **chained allocation**) solves all problems of contiguous allocation.
    - Each file is a **linked list** of storage blocks.
    - These blocks may be scattered anywhere on the disk.
    - Each block contains a pointer to the next block.
    - **End-of-File** could be achieved by:
      - Ending with a **NULL** pointer.
      - Explicitly specify an **END** block.
      - or specify the **length** of the file.

File allocation table

File name	Start block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •



Allocation Methods

- **Linked Allocation**
  - Advantages:
    - **Simple:**
      - To **create** a new file, simply create a new entry in the directory. Each entry has a pointer to the first block of the file.
        - Initially, an empty file $\Rightarrow$ NULL pointer in the directory entry.
      - To **read** a file, simply follow the linked list *from block to block*.
      - To **write** (**append**) to the file $\Rightarrow$ new block allocated and append to the end of the file. (last pointer **points** to the new block).
        - The size of a file need not be declared when first created; a file can continue to **grow** as long as free blocks are available.
    - No **external fragmentation** $\Rightarrow$ No need for **compaction**.
      - Free space management: no waste of storage space.

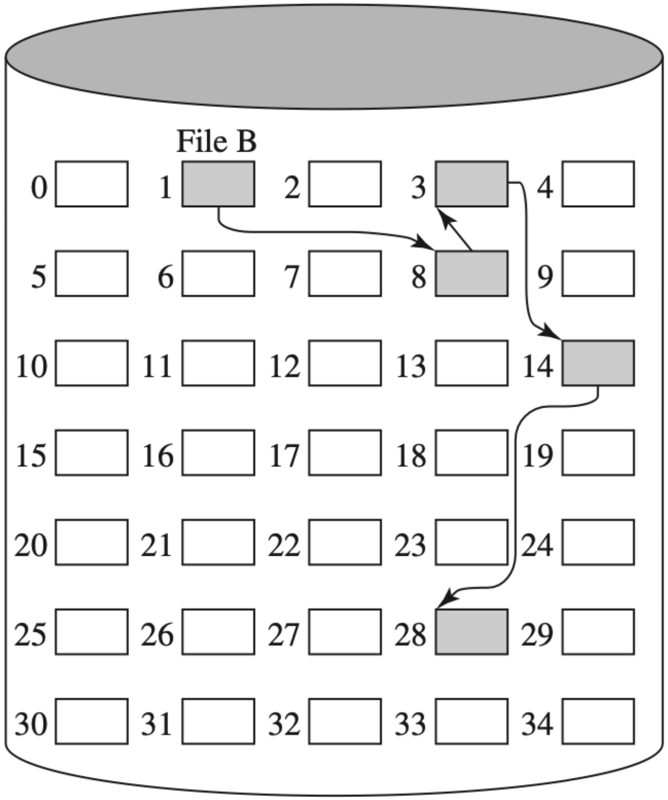
Allocation Methods

Linked Allocation

- Disadvantages:
  - **Sequential access** only. No **random access** within a file.
    - To locate the i^{th} block of a file, we must start from the beginning of that file and follow the pointers until we get to the i^{th} block.

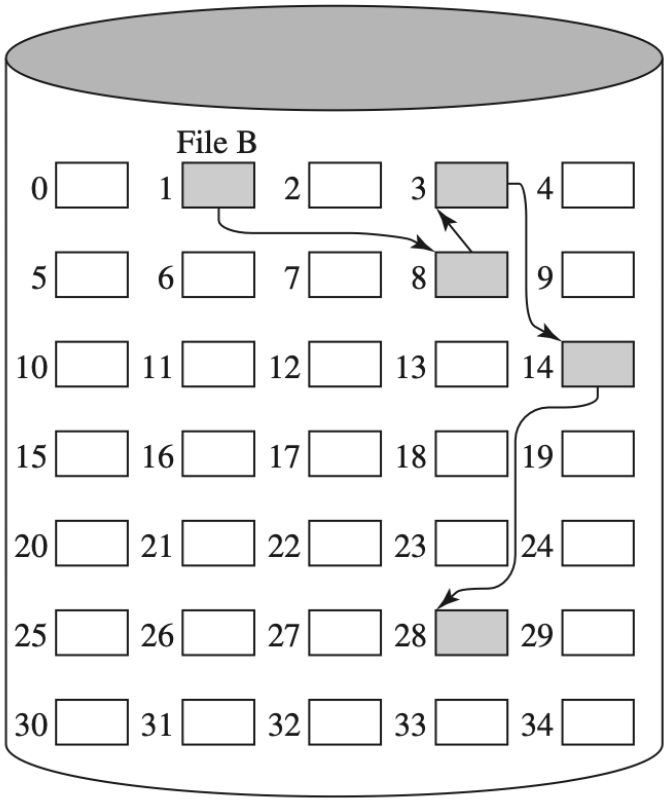
Linked Allocation

- Disadvantages:
  - **Sequential access** only. No **random access** within a file.
    - To locate the i^{th} block of a file, we must start from the beginning of that file and follow the pointers until we get to the i^{th} block.
  - **Difficult** to make use of **Locality** (**局部性**).
    - The file may be scattered around the disk, i.e., the next block (that the pointer points to) may be located far away from the current block, making it difficult to leverage **Locality** (very low **cache hit ratio**).



Linked Allocation

- Disadvantages:
  - **Sequential access** only. No **random access** within a file.
    - To locate the i^{th} block of a file, we must start from the beginning of that file and follow the pointers until we get to the i^{th} block.
  - **Difficult** to make use of **Locality** (**局部性**).
    - The file may be scattered around the disk, i.e., the next block (that the pointer points to) may be located far away from the current block, making it difficult to leverage **Locality** (very low **cache hit ratio**).
  - Storage Space **Utilization**.
    - extra space required for the pointers.
    - If a pointer requires 4 bytes out of a 512-byte block, the 0.78% of disk is used for pointers (**overhead**).



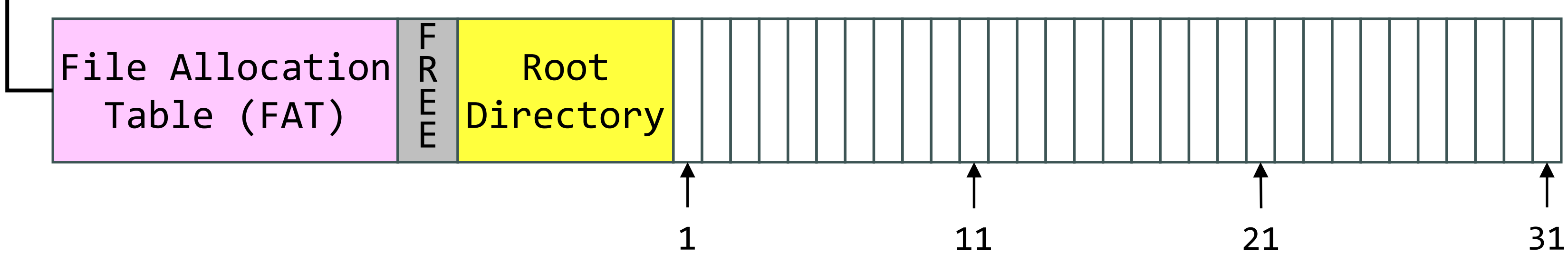
- **File Allocation Table (FAT)**
  - An important variation of Linked Allocation is the use of a **File Allocation Table** (**FAT**, **文件分配表**).
    - a combination of **continuous** and **linked** allocation (*most of the time*).
  - A section of storage at the beginning of the volume is set aside to contain the **table**.
    - The table has **one entry for each block** and is **indexed by block number**.
    - Each entry contains a pointer to the next block in the file.
      - a NULL pointer (`0`) indicates End-Of-File.
      - an *implicit* **linked list** via indexed table.

Allocation Methods

File Allocation Table (FAT)

- An important variation of Linked Allocation is the use of a **File Allocation Table** (**FAT**, **文件分配表**).
  - a combination of **continuous** and **linked** allocation (*most of the time*).
- A section of storage at the beginning of the volume is set aside to contain the **table**.
  - The table has **one entry for each block** and is **indexed by block number**.
  - Each entry contains a pointer to the next block in the file.
    - a NULL pointer (`0`) indicates End-Of-File.
    - an *implicit* **linked list** via indexed table.

Block #	1	2	···	6	7	8	···	18	19	20	···	25	26	27	28	29	30	31
Next #	2	3	···	0	8	9	···	26	20	21	···	0	27	0	29	30	31	0

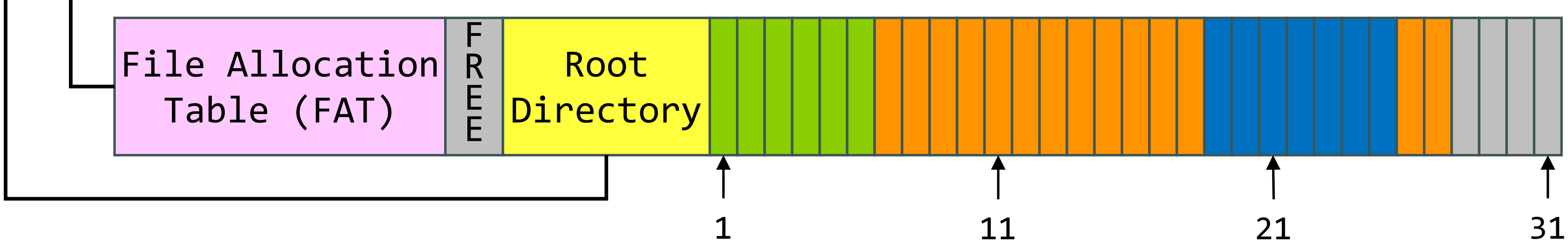


File Allocation Table (FAT)

- A section of storage at the beginning of the volume is set aside to contain the **table**.
  - The table has **one entry for each block** and is **indexed by block number**.
  - Each entry contains a pointer to the next block in the file.
    - a NULL pointer (`0`) indicates End-Of-File.
    - an *implicit* **linked list** via indexed table.

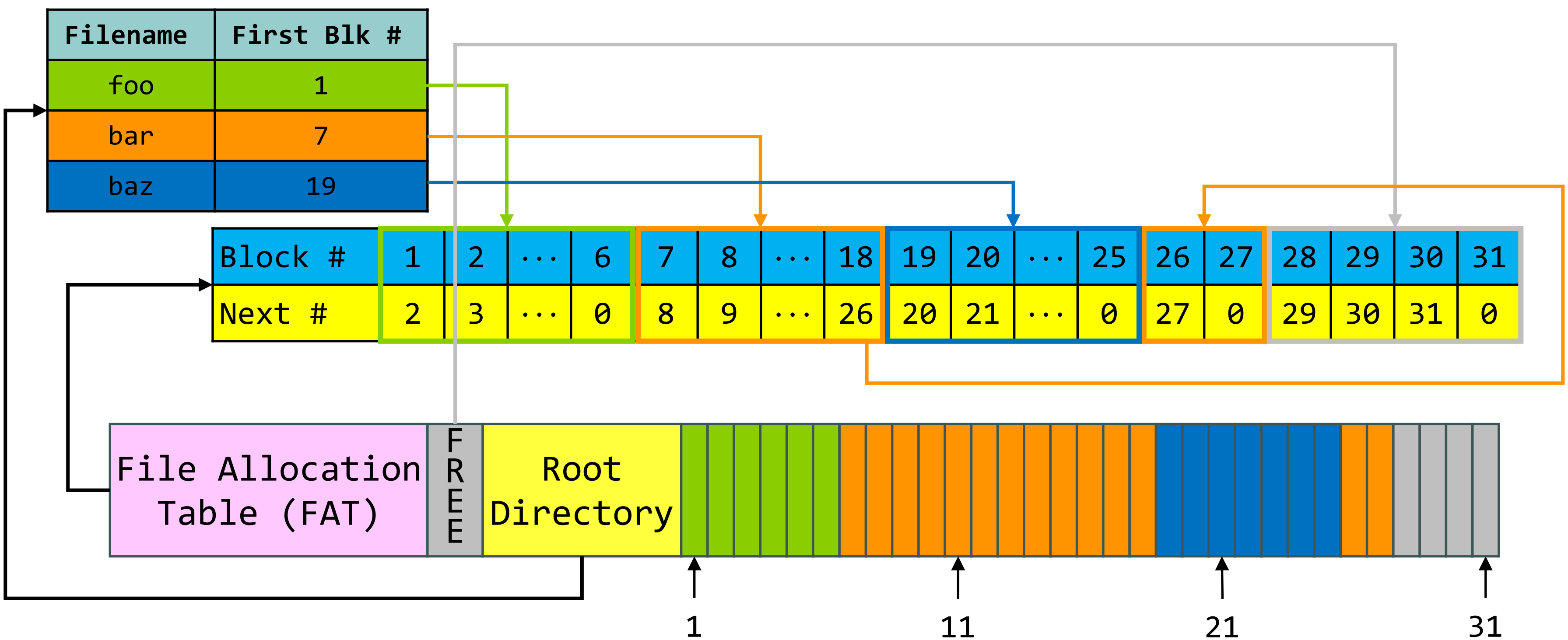
Filename	First Blk #
foo	1
bar	7
baz	19

Block #	1	2	···	6	7	8	···	18	19	20	···	25	26	27	28	29	30	31
Next #	2	3	···	0	8	9	···	26	20	21	···	0	27	0	29	30	31	0



File Allocation Table (FAT)

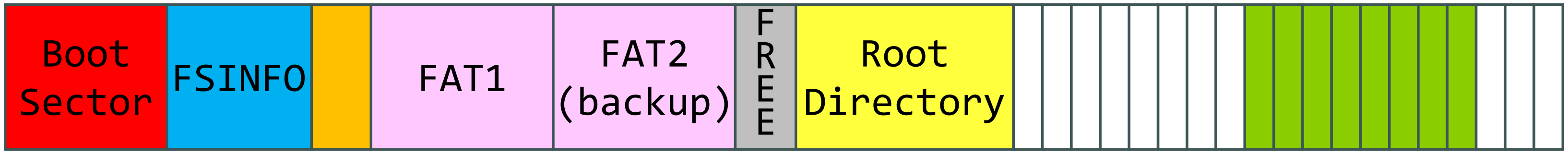
- **FAT** is basically an *implicit* **linked list**.
  - Instead of **allocate** (at the end of each block) a pointer to the next block, we delegate the linked list structure to the **FAT**.
    - The **FAT** can then be **cached** in the **main memory**, which **speeds up** walking through the **linked list** structure of a file.
      - enables **random-access**.



FAT32 Layout

```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
```

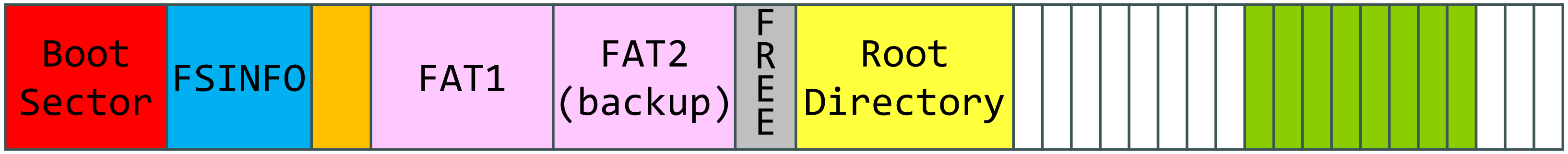
Format the disk, "-F32" means FAT32



FAT32 Layout

```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
```

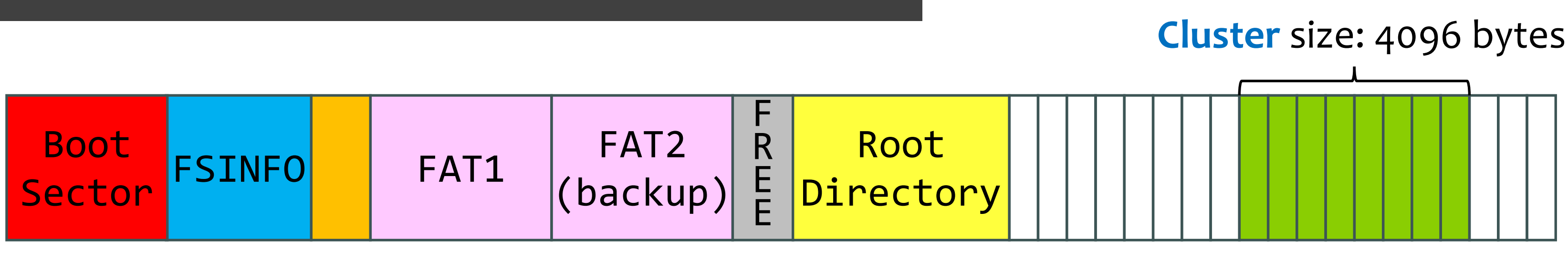
Read the info stored in the boot sector.



FAT32 Layout

```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
...
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
      4096 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    512000 bytes per FAT (= 1000 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1040384 (sector 2032)
    127738 data clusters (523214848 bytes)
...
```

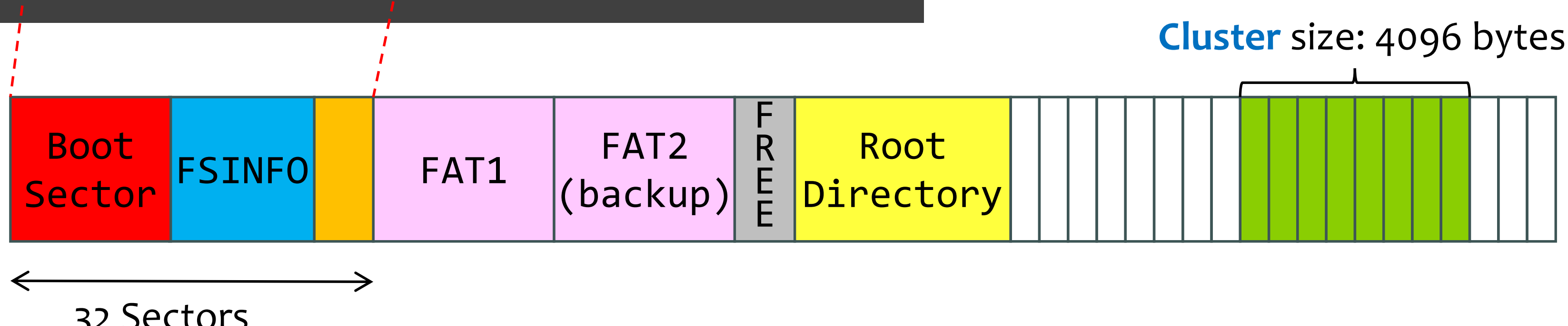
A **cluster** (**block**) is made of 8 **sectors.**



Allocation Methods

FAT32 Layout

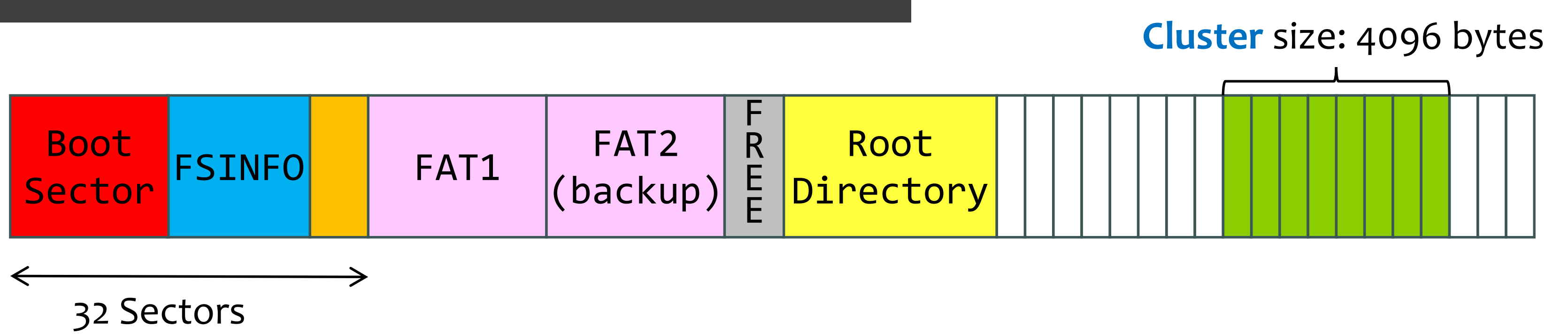
```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
...
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
      4096 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    512000 bytes per FAT (= 1000 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1040384 (sector 2032)
    127738 data clusters (523214848 bytes)
...
```



FAT32 Layout

```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
...
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
      4096 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    512000 bytes per FAT (= 1000 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1040384 (sector 2032)
    127738 data clusters (523214848 bytes)
...
```

2 FATs:
- FAT1 is the primary.
- FAT2 serves as the backup.

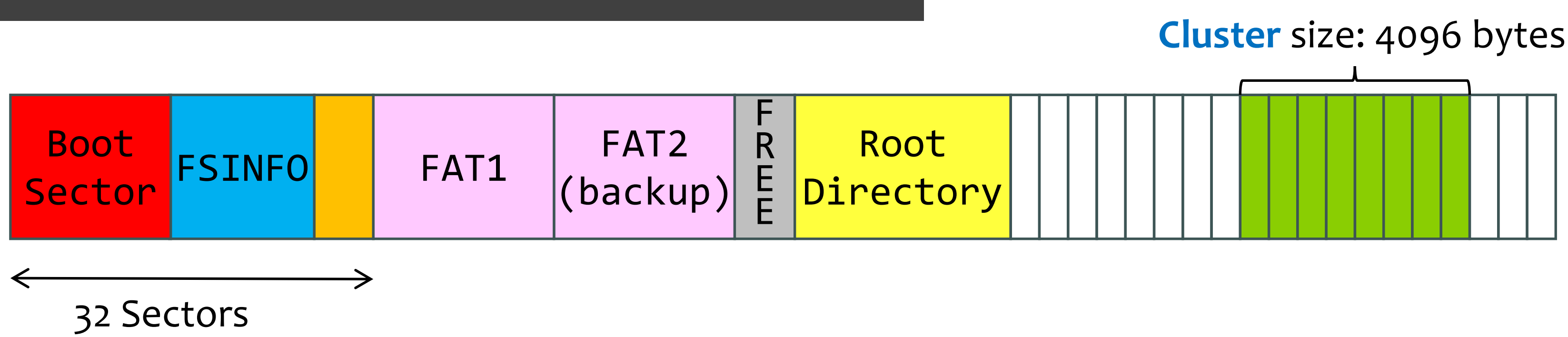


Allocation Methods

FAT32 Layout

```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
...
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
      4096 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    512000 bytes per FAT (= 1000 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1040384 (sector 2032)
    127738 data clusters (523214848 bytes)
...
```

Each FAT is of size 512000 bytes, with a 4-byte (32-bit, hence FAT32) address, it can index up to $\frac{512000}{4} =$ 128000 entries (clusters), just enough for the 127738 data clusters in the Data Area.

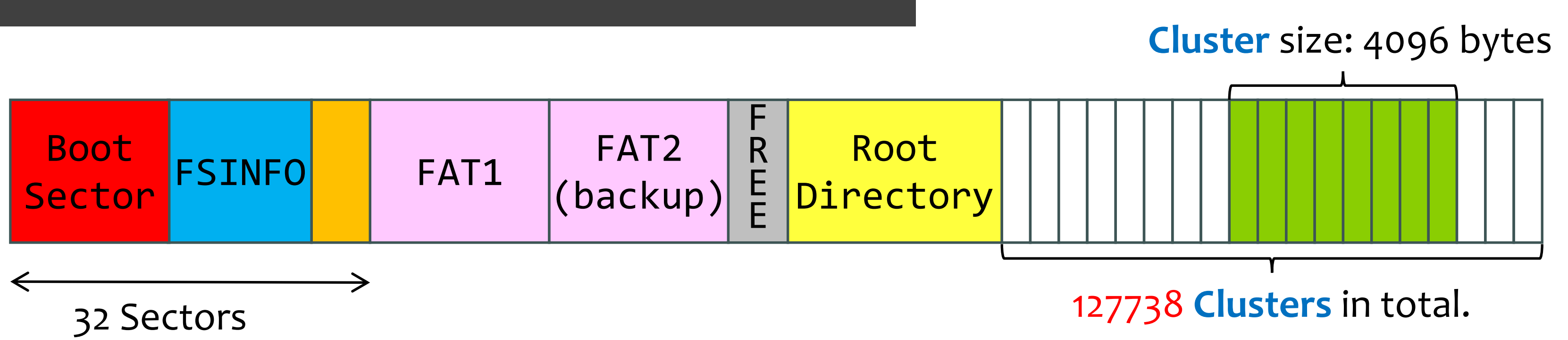


Allocation Methods

FAT32 Layout

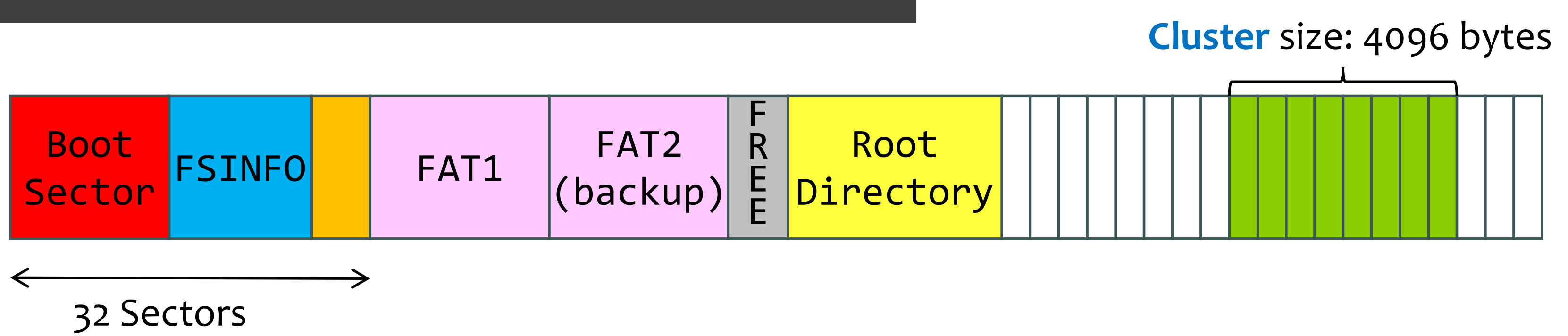
```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
...
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
      4096 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    512000 bytes per FAT (= 1000 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1040384 (sector 2032)
    127738 data clusters (523214848 bytes)
...
```

Each FAT is of size 512000 bytes, with a 4-byte (32-bit, hence FAT32) address, it can index up to $\frac{512000}{4} =$ 128000 entries (clusters), just enough for the 127738 data clusters in the Data Area.



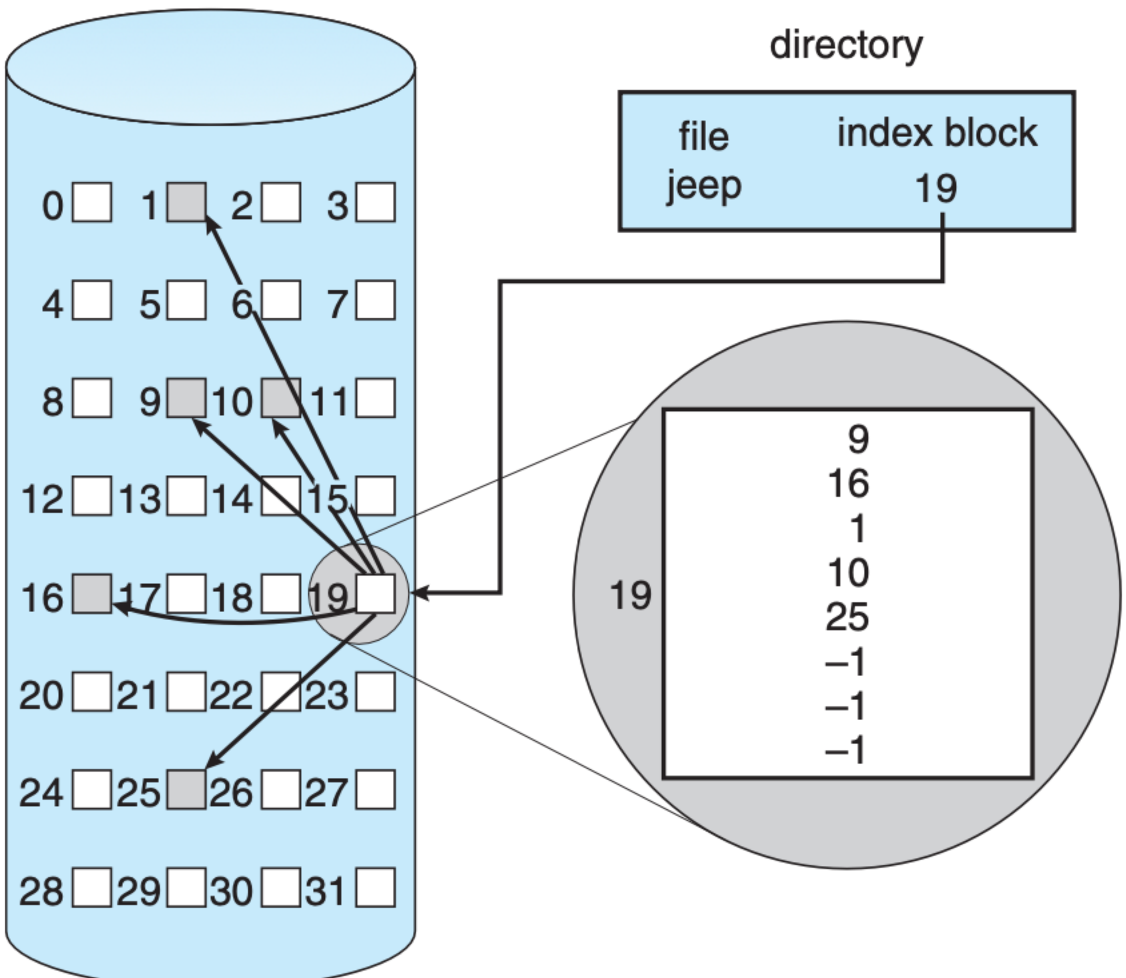
FAT32 Layout

```
$ sudo modprobe brd rd_nr=1 rd_size=512000
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 4.2 (2021-01-31)
$ sudo dosfsck -v /dev/ram0
...
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
      4096 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
    512000 bytes per FAT (= 1000 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1040384 (sector 2032)
    127738 data clusters (523214848 bytes)
...
```



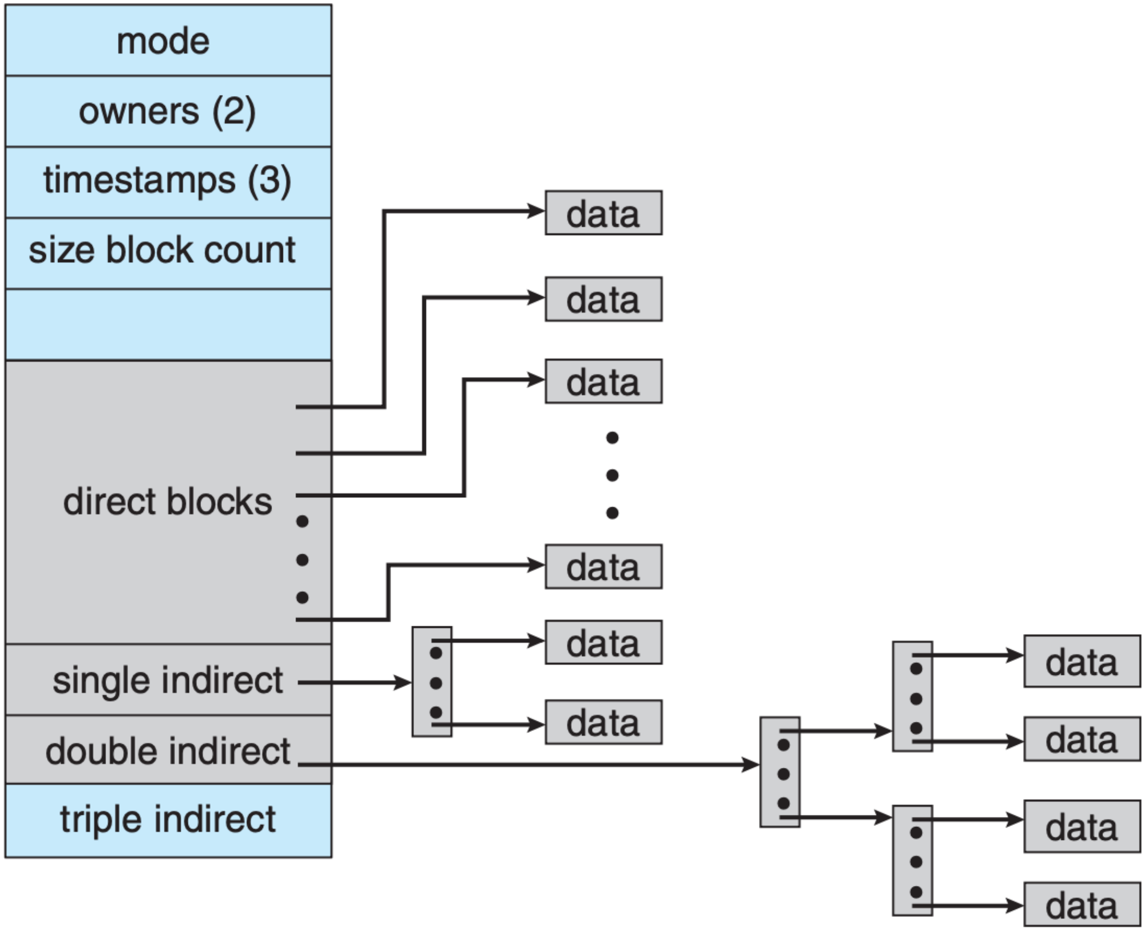
Indexed Allocation

- As mentioned earlier in **VSFS** (**Very Simple File System**), the UNIX File System, and ext2/ext3/ext4 in Linux, etc.



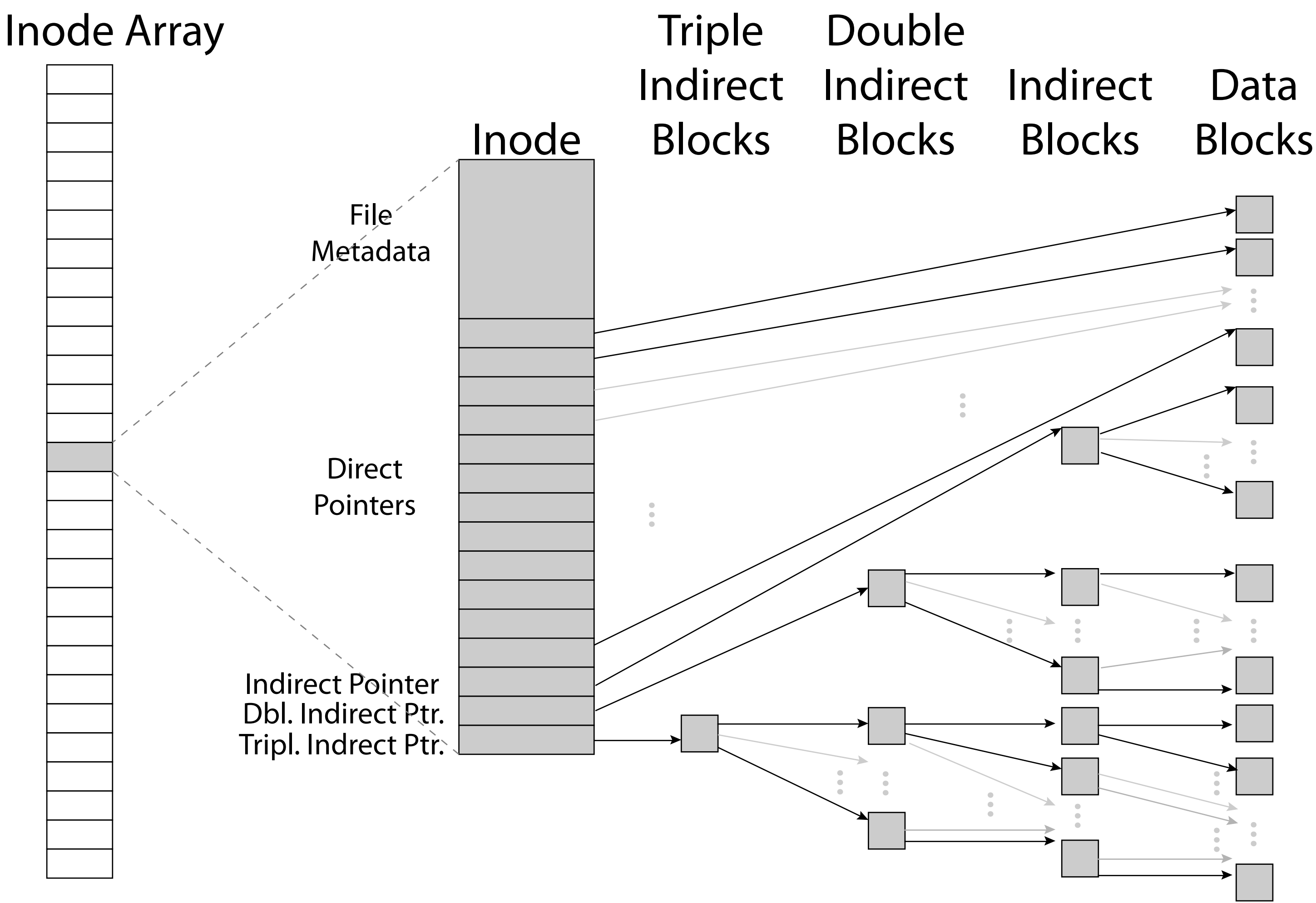
Indexed Allocation

- As mentioned earlier in **VSFS** (**Very Simple File System**), the UNIX File System, and ext2/ext3/ext4 in Linux, etc.



Indexed Allocation

- As mentioned earlier in **VSFS** (**Very Simple File System**), the UNIX File System, and ext2/ext3/ext4 in Linux, etc.



Allocation Methods

Comparison

- Comparison of different Allocation Methods.

	Contiguous	Linked	Indexed	
Pre-allocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Free Space Management

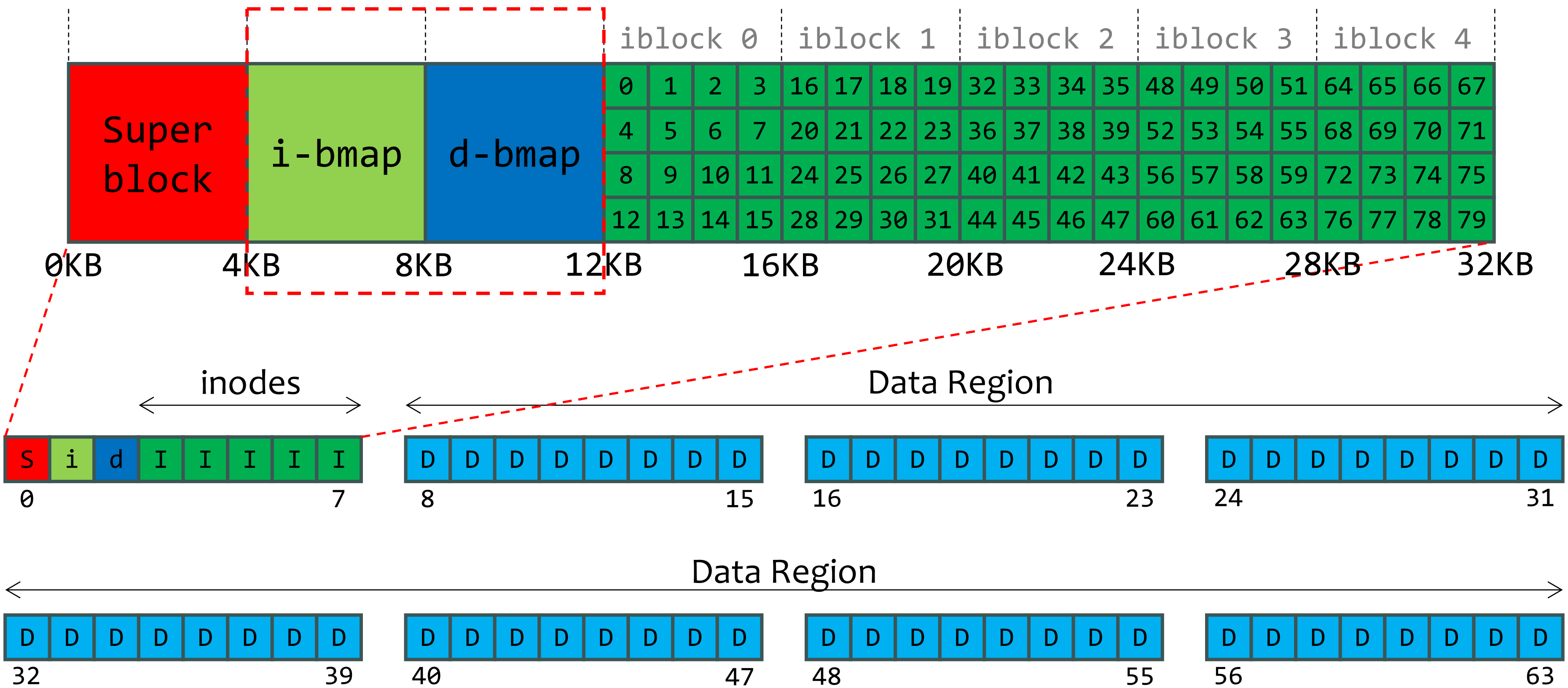
- The system maintains a **free-space list** to keep track of free disk space, or available blocks/clusters on disk.
- There are many different schemes to implement a **free-space list**:
  - **Bit Vector** (Bitmap)
  - **Linked List**
  - **Grouping**
  - **Counting**
  - **Space Maps**

Free Space Management

- **Bit Vector**
  - Most frequently, the **free-space list** is implemented as a **bitmap** or bit vector.
    - Each block is represented by one bit.
      - If the block is free, the bit is set to `1`.
      - If the block is allocated, the bit is set to `0`.
    - Bitmap requires extra space. For example, assume:
      - block size = **4KB** = 2^{12} bytes
      - disk size = **1TB** = 2^{40} bytes
      - # of blocks: $2^{40} / 2^{12} = 2^{28}$
    - Then, space requirement for bitmap: 2^{28} bits = **32MB**

Free Space Management

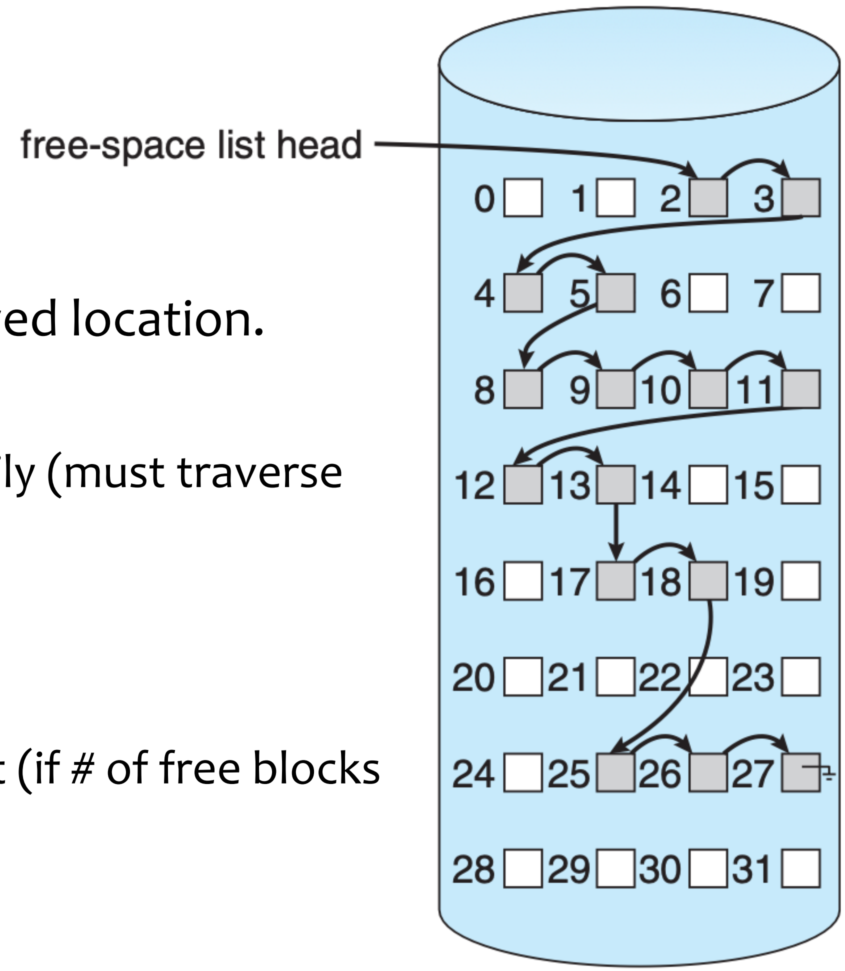
- **Bit Vector**
  - Example: **VSFS** or **ext2/ext3/ext4**



Free Space Management

Linked List

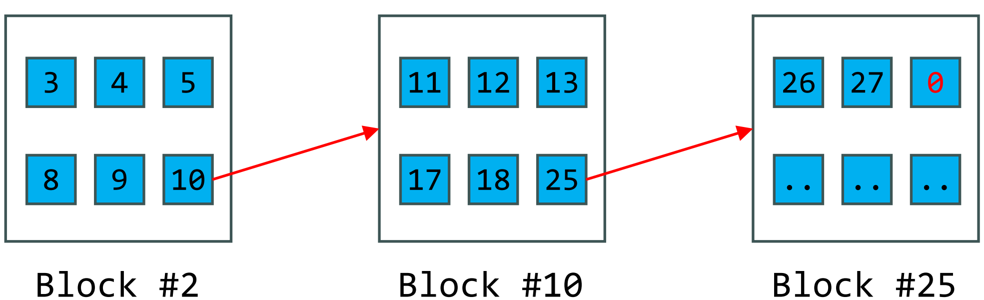
- Another approach is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory.
- For example: blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27 are free. The number of the first free block (#2) is stored in a reserved location.

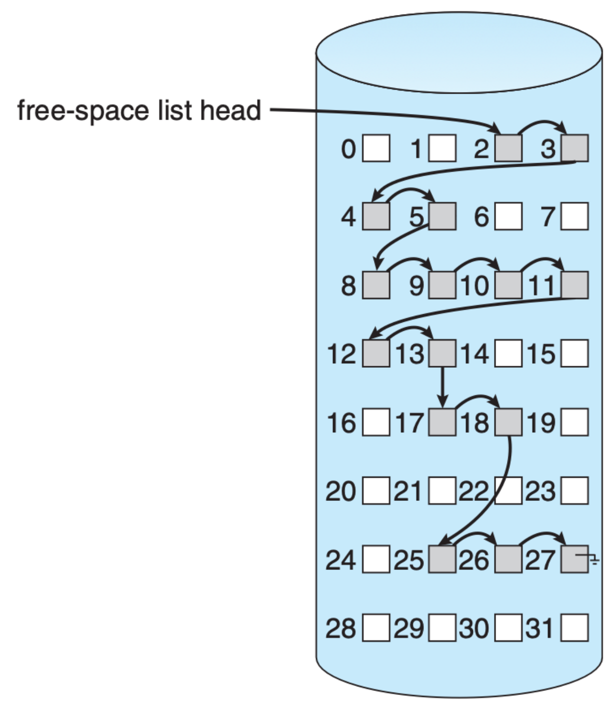


- **Disadvantages:**
  - Cannot get contiguous space easily (must traverse the list of free blocks).
- **Advantages:**
  - No waste of disk space
  - No need to traverse the entire list (if # of free blocks is also recorded). **How**?

Free Space Management

- **Grouping**
  - A modification of the **linked-list** approach.
  - It stores the addresses of n free blocks in the first free block.
    - The first $n-1$ of these blocks are actually free.
    - The last block contains the address of another n free blocks, and so on.
  - These addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard **linked-list** approach is used.
  - **Example:** Suppose $n = 6$.



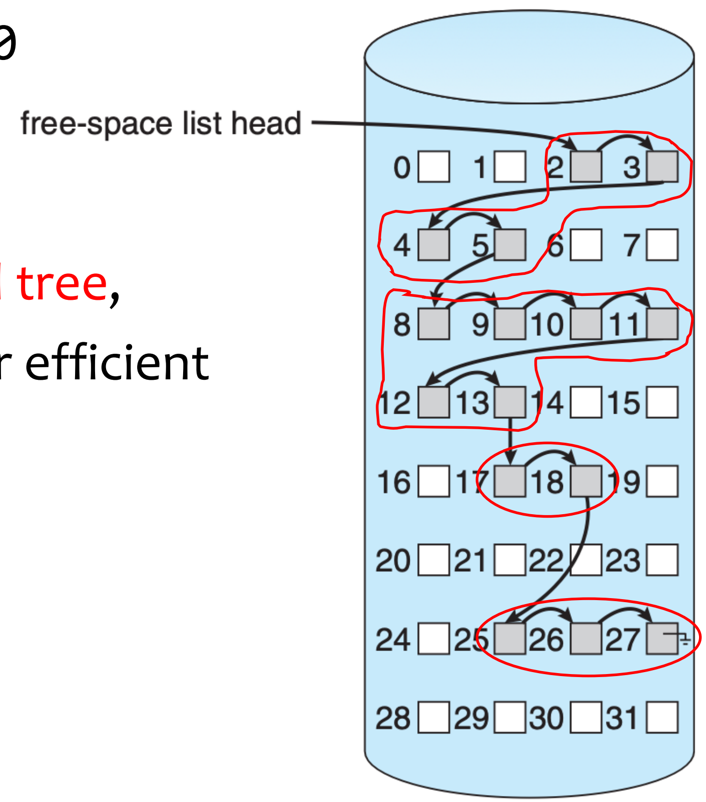


Free Space Management

- **Counting**
  - A modification of the **Bitmap** approach.
    - space is frequently contiguously allocated and freed.
    - Each entry contains the address of the next free block and the number of free blocks that contiguously follows it.
  - With simple **bitmap**, the bit vector table would be:

    `00111100111111000110000001110000`
  - With **counting**:

    `<2,4>|<8,6>|<17,2>|<25,3>`
  - These entries can be stored in a balanced tree, rather than a linear table or linked list, for efficient **lookup**, **insertion** and **deletion**.



Free Space Management

- **Space Maps**
  - ZFS (Zettabyte File System, Sun Solaris 10) was designed to encompass huge numbers of files, directories and even file systems.
    - On these scales, metadata I/O can have a huge performance impact.
    - Full data structures like bitmaps couldn't fit thousands of I/Os.
  - Divide device space into metaslab units and manage them.
    - Given volume can contain hundreds of metaslabs.
    - EAch metaslab has an associated space map.
      - Uses counting algorithm.
    - But records to log file rather than file system.
      - Log of all block activity, in time order, in counting format.
  - Metaslab activity is to load space map into memory in balanced-tree structure, indexed by offset.
    - Replay log into that structure.
    - Combine contiguous free blocks into single entry.

Efficiency and Performance

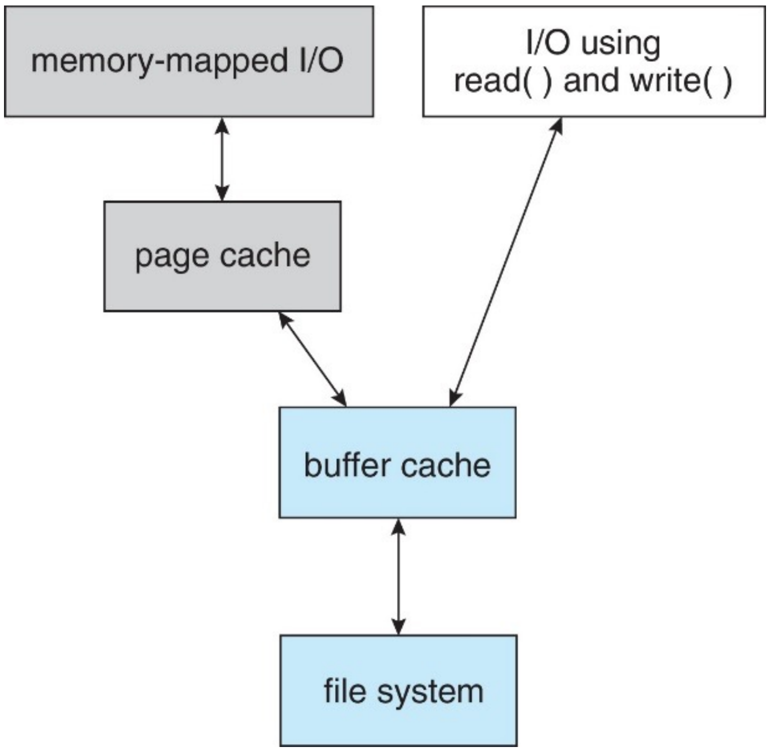
- Efficiency depends on
  - Disk allocation and directory algorithms.
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures.
- Performance
  - Disk cache
    - separate section/segment of main memory for frequently used blocks.
  - Free-behind and read-ahead
    - techniques to optimize sequential access.
  - Improve performance by dedicating section/segment of memory as virtual disk or RAM disk.

Efficiency and Performance

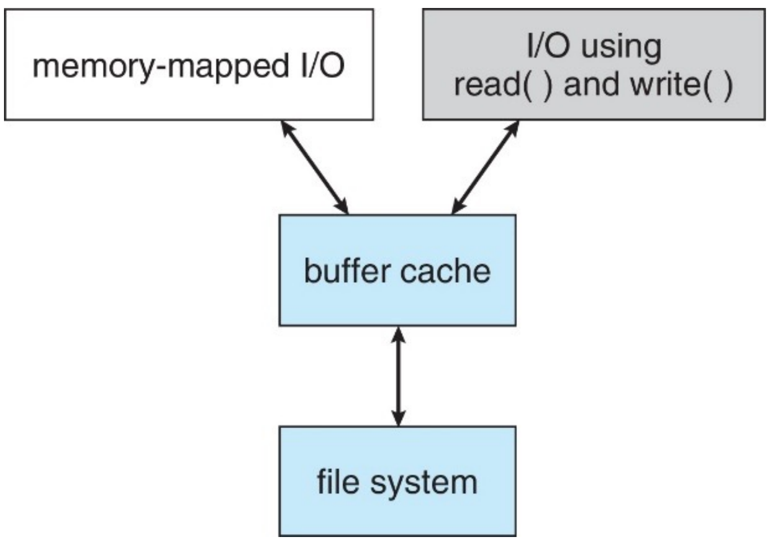
- Page Cache
  - A **page cache** caches pages rather than disk blocks using virtual memory techniques.
    - Memory-mapped I/O uses a **page cache**.
    - Routine I/O through the file system uses the buffer (disk)-cache.
- Unified Buffer Cache (UBC)
  - A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching.

Efficiency and Performance

- Unified Buffer Cache



I/O without a unified buffer cache



I/O using a unified buffer cache

Thank you!