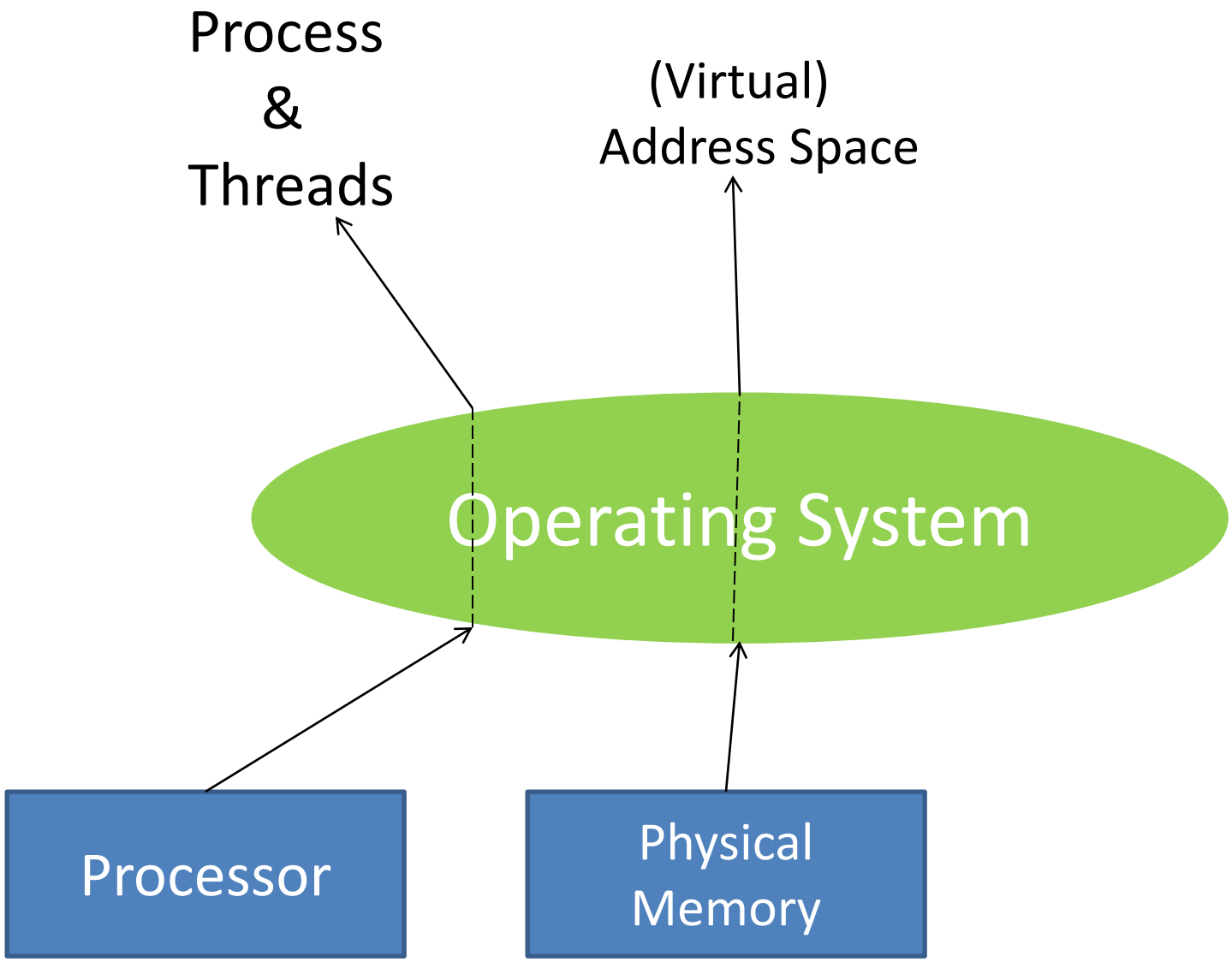
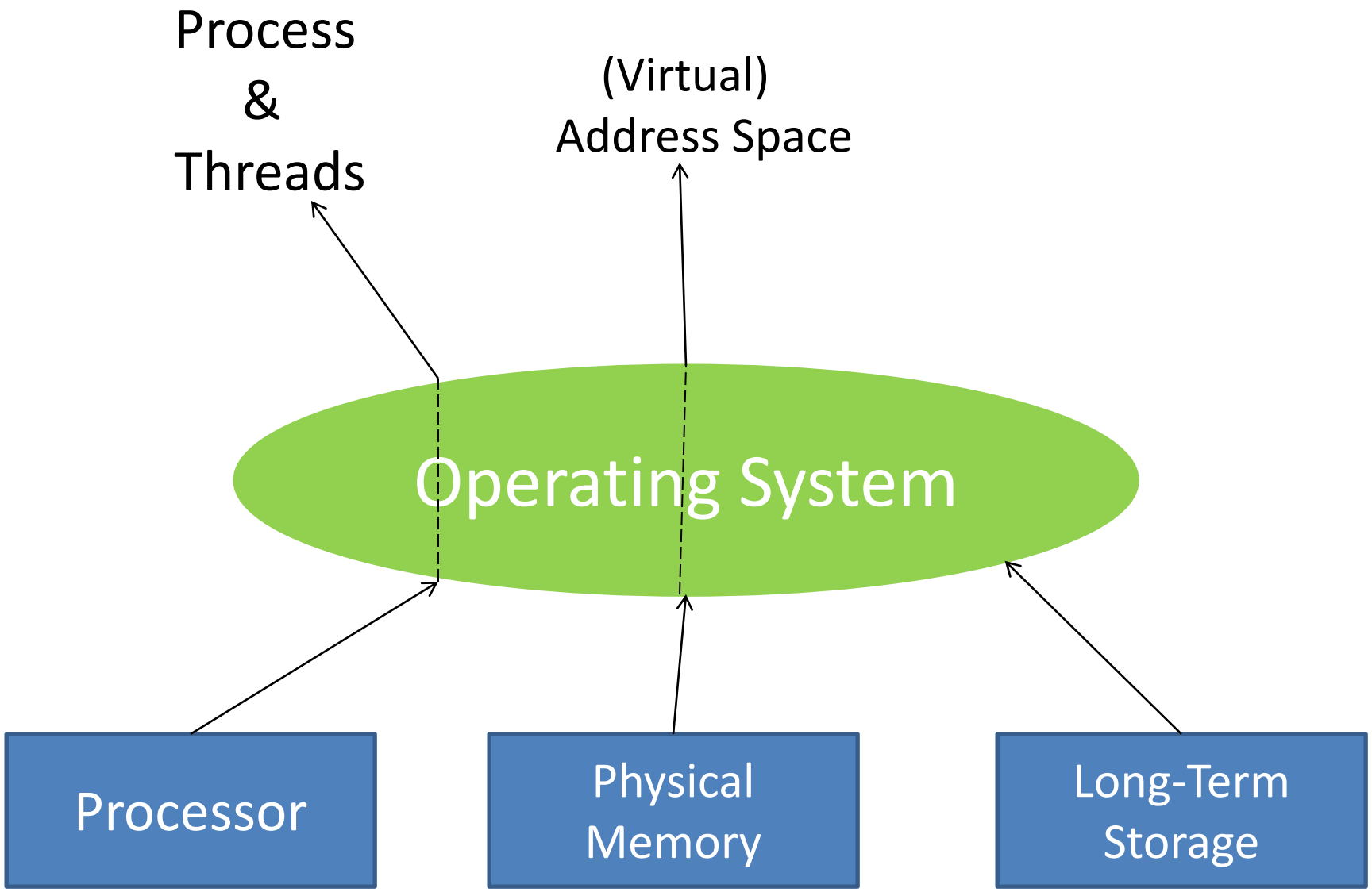
NYU
New York University

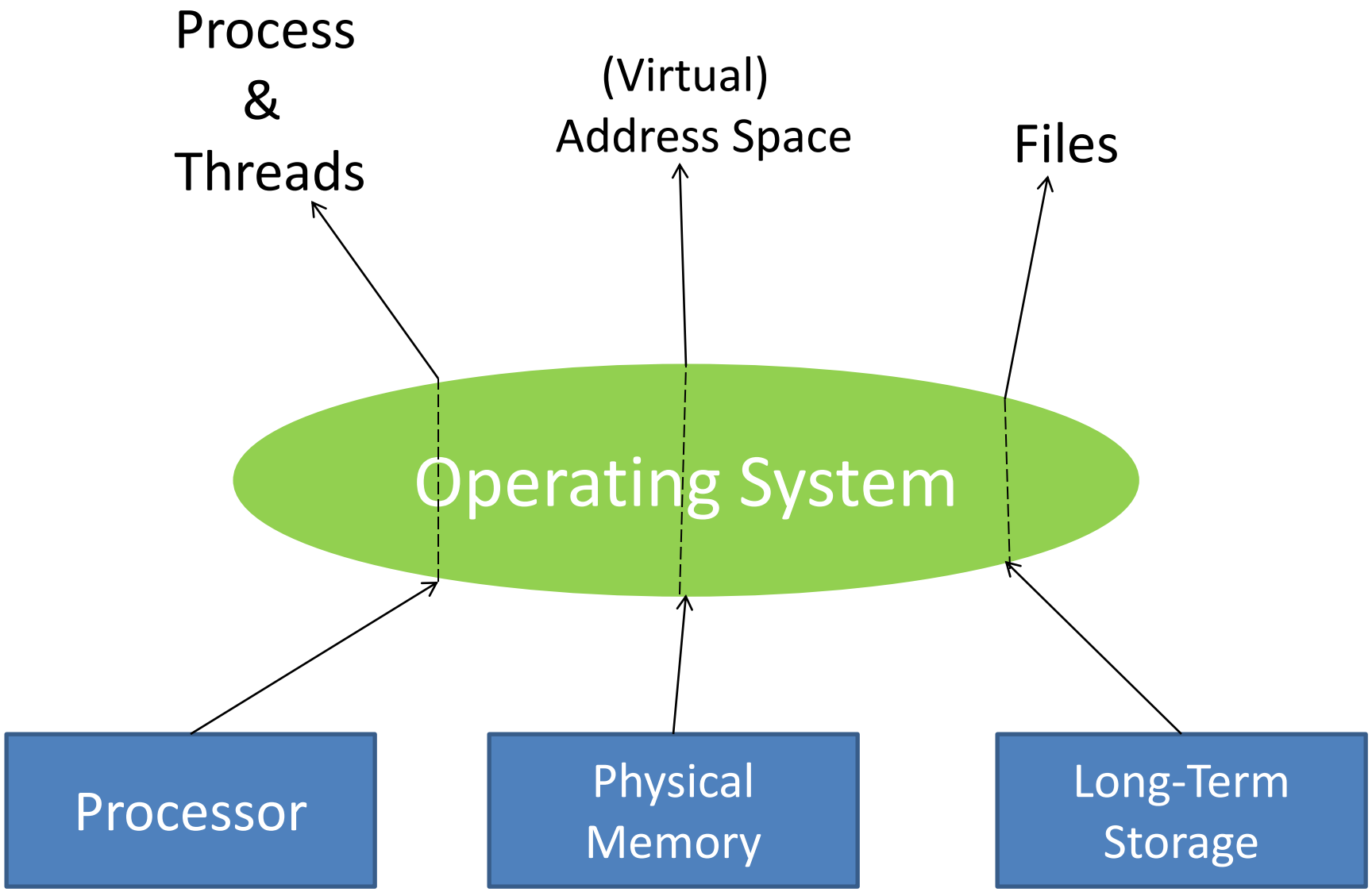
# Operating Systems

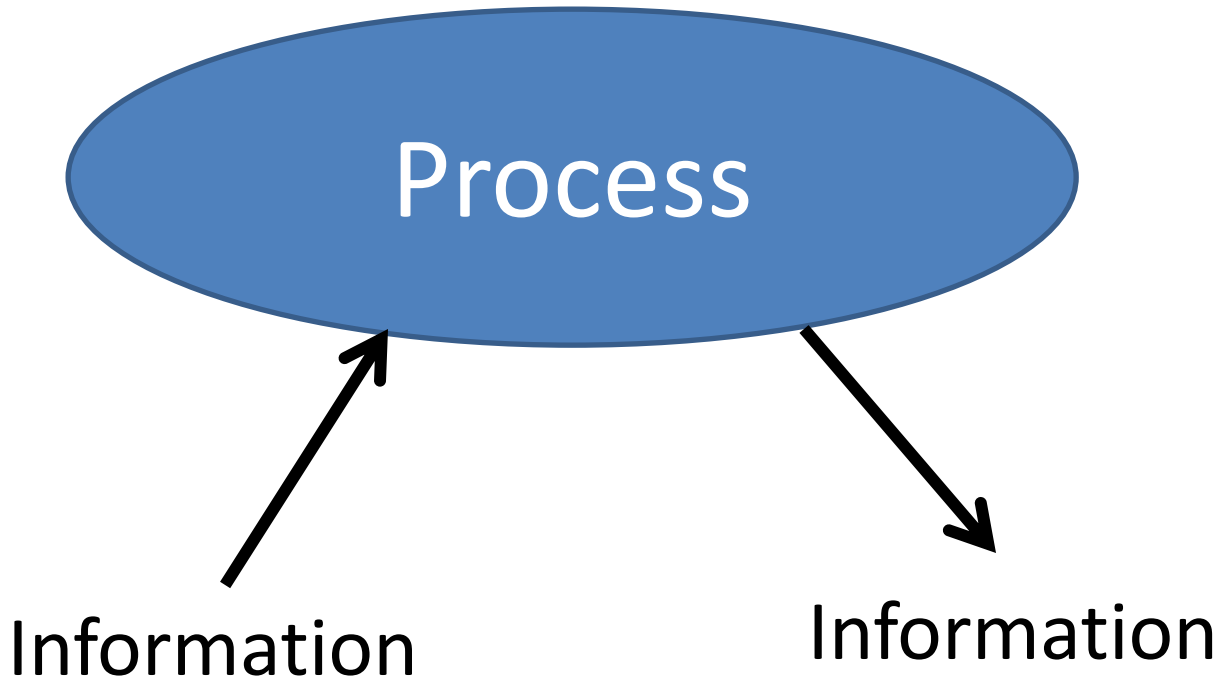
## File Systems

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com


Process
&
Threads
(Virtual)
Address Space
Operating System
Processor
Physical
Memory


Process
&
Threads
(Virtual)
Address Space
Operating System
Processor
Physical
Memory
Long-Term
Storage


Process
&
Threads
(Virtual)
Address Space
Files
Operating System
Processor
Physical
Memory
Long-Term
Storage




Is it OK to keep this information only in the process address space?

# Shortcomings of Process Address Space

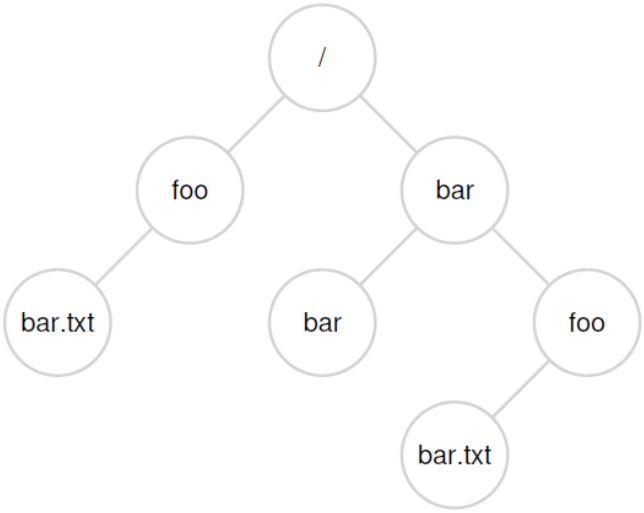
- Virtual address space may not be enough storage for all information.
- Information is lost when the process terminates, is killed, or the computer crashes.
- Multiple processes may need the information at the same time.

# Requirements for Long-term Information Storage

- Store very large amount of information.
- **Persistence**: Information must survive the termination of the process using it.
- Multiple processes must be able to access the information concurrently.

# Two Key Abstractions for Storage

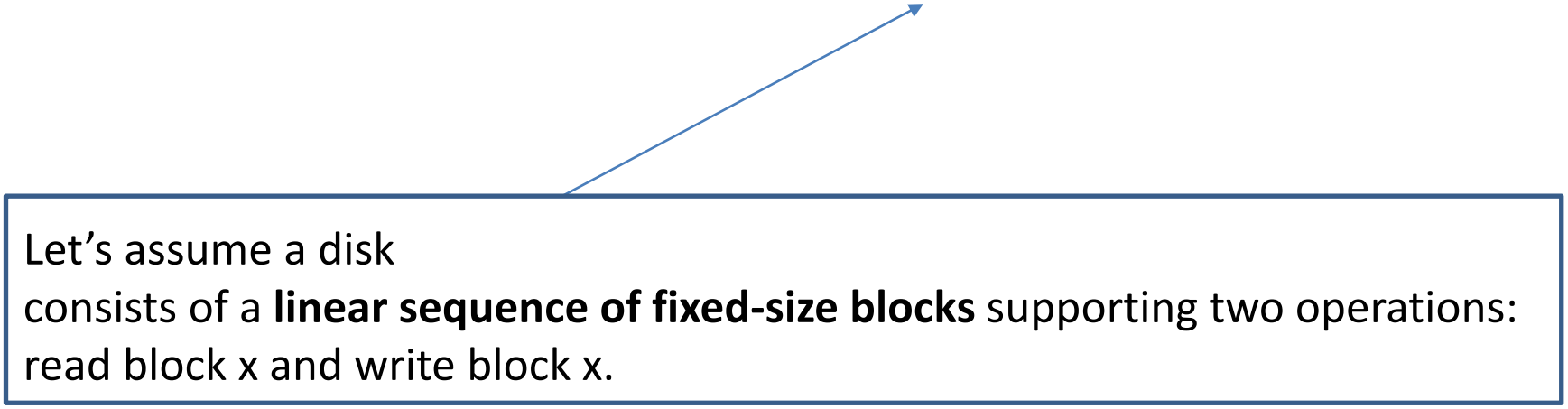
- Files
  - Simply a linear array of bytes
  - In most systems, the OS does not know much about the structure of the file.
- Directories




Directories and files have low-level names beside the name used by the user.

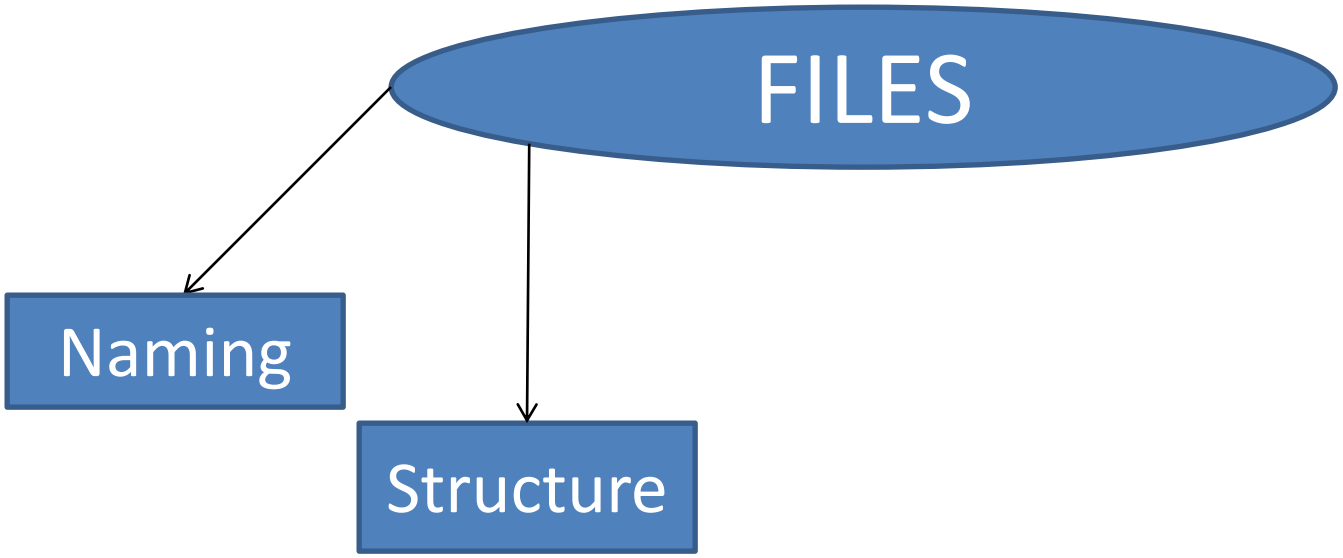
# Issues with long term storage

- How do you find information?
- How do you keep one user from reading another user's data?
- How do you know which blocks are free?

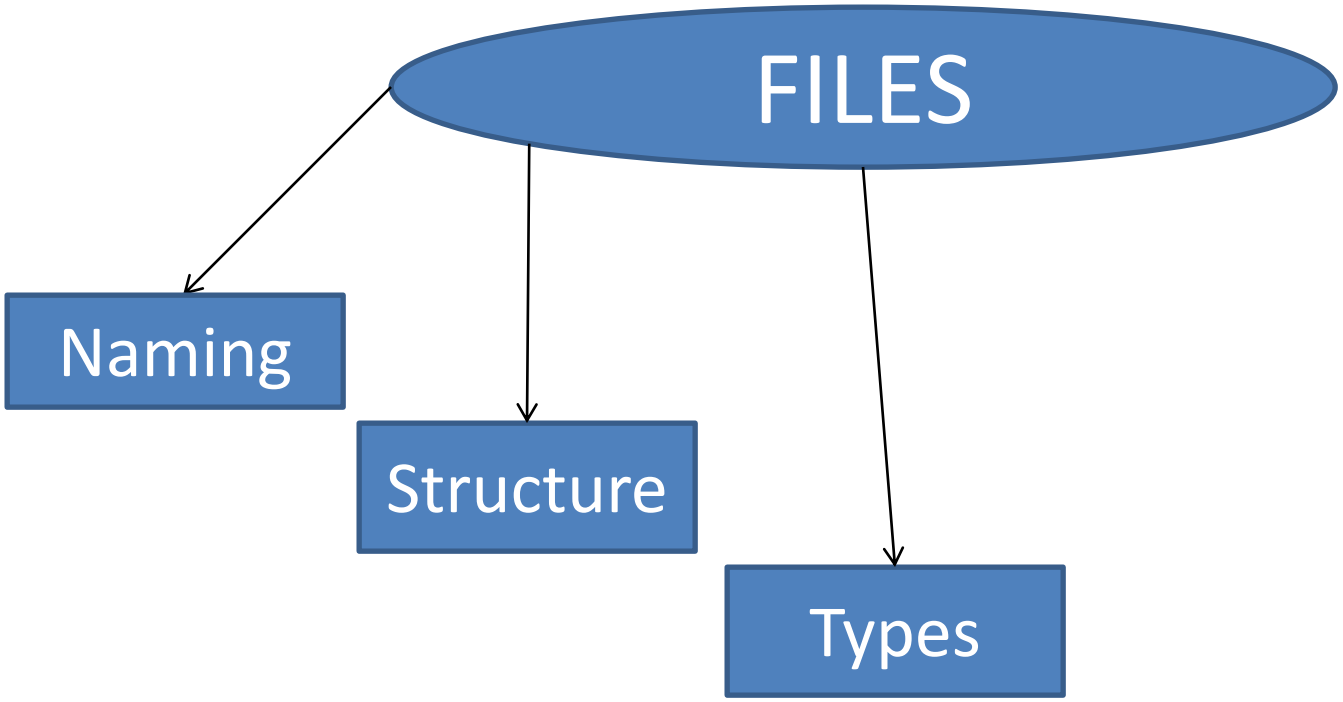



## Naming

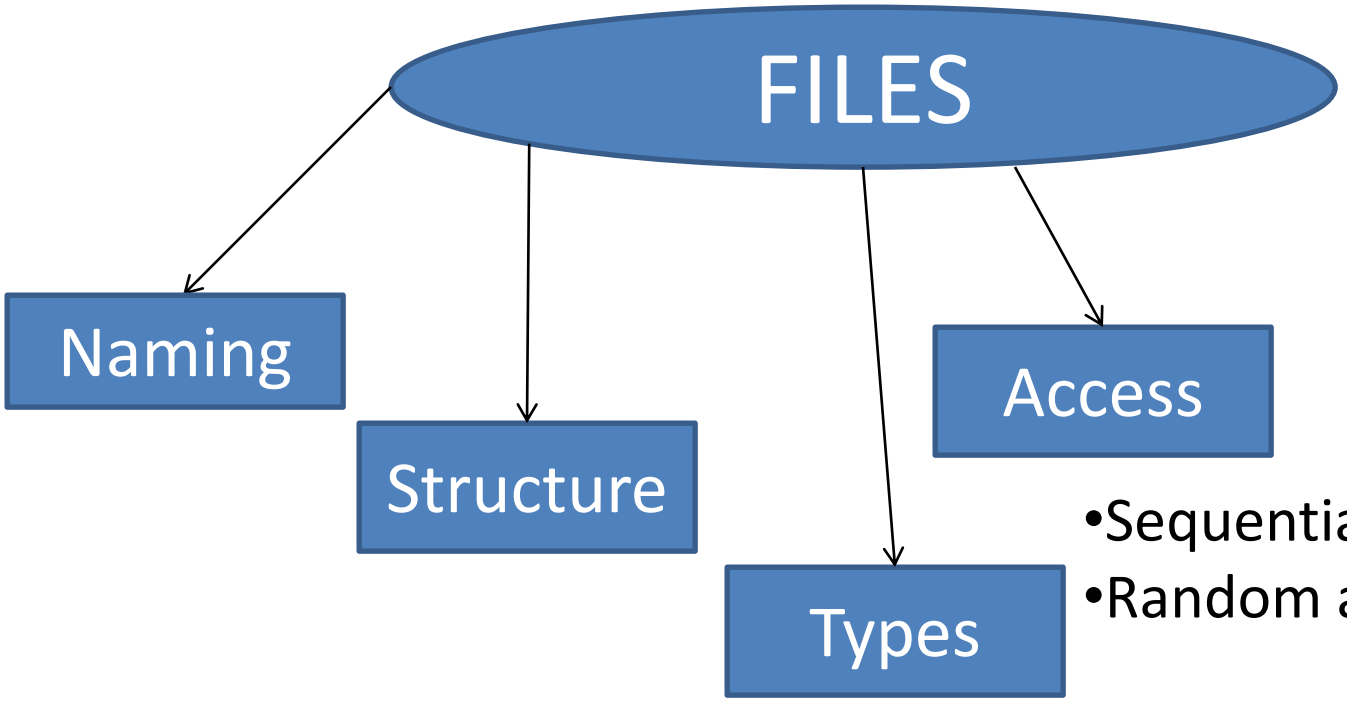
- Shields the user from details of file storage.
- In general, files continue to exist even after the process that creates them terminates.



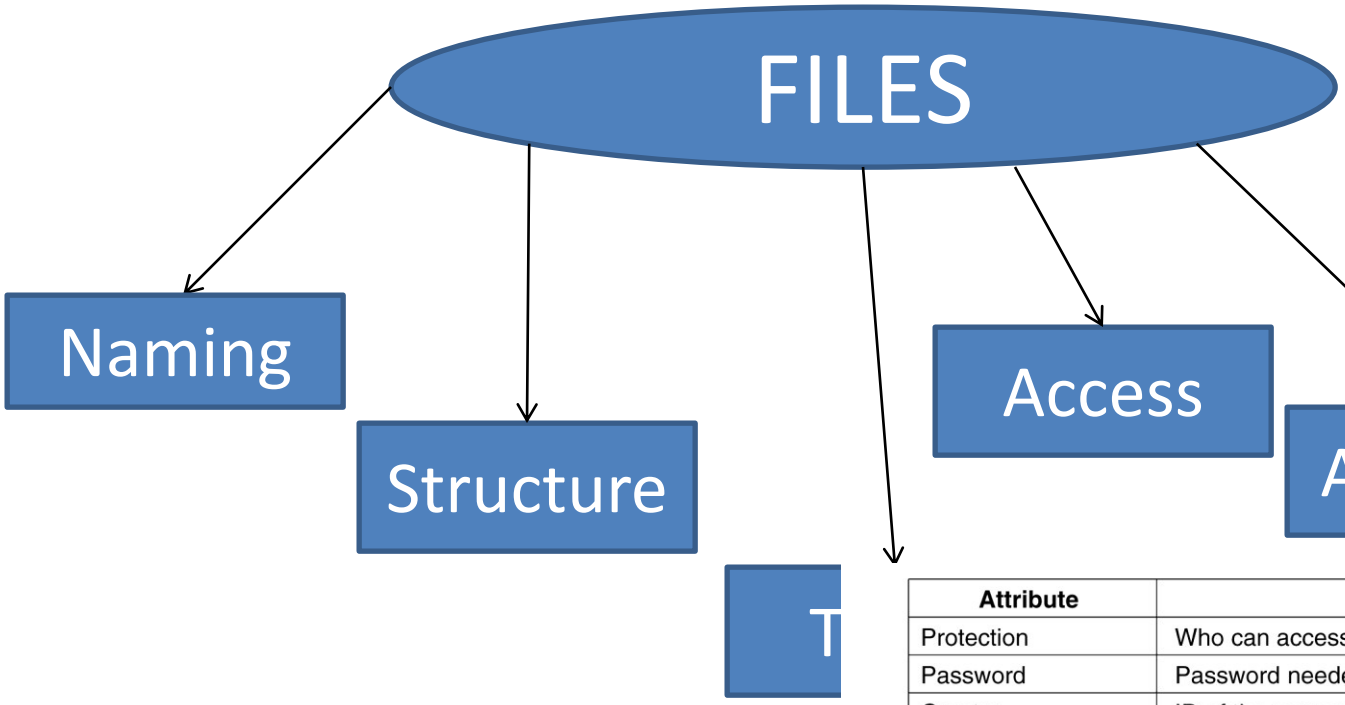

- OS sees it as a linear stream of bytes.
- Users can have different views (e.g. record, …)
- Programs can impose structures (e.g. jpg, mp3, …)



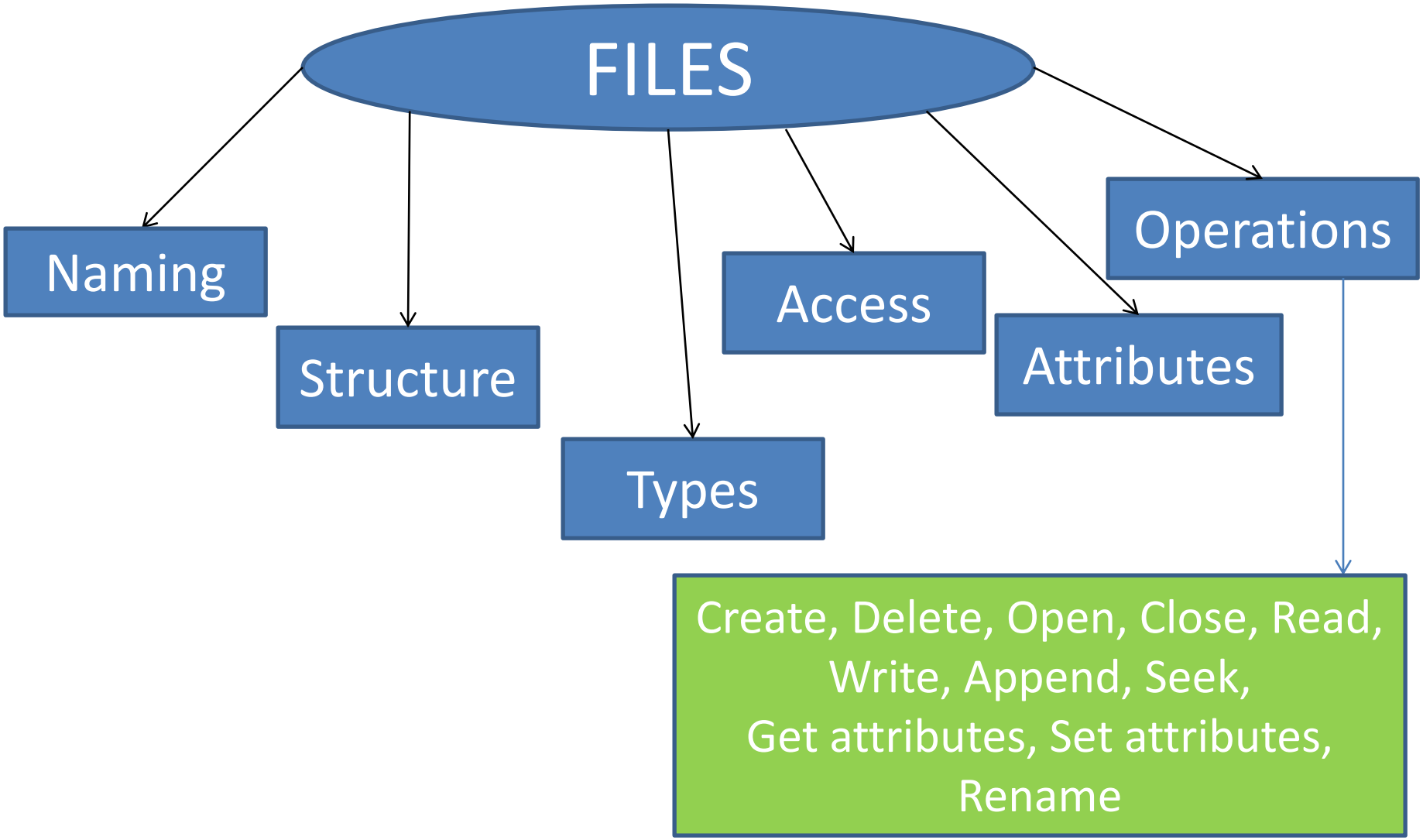

- Regular files
  - ASCII
  - Binary
- Character special
  - to model serial devices (printers, networks, …)
- Block special
  - to model disks




- Sequential
- Random access




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


FILES
Naming
Structure
Types
Access
Attributes
Operations
Create, Delete, Open, Close, Read,
Write, Append, Seek,
Get attributes, Set attributes,
Rename

# Dealing with Files in C (As an Example)

# First step

- Declaration:

```
FILE *fptr1, *fptr2 ;
```

# Opening Files

- The statement:

  fptr1 = **fopen** ( “filename", "r" ) ;

  would open the file filename for input (reading).

  - r: read
  - w: write
  - a: append
  - … there are some more

# Testing for Successful Open

- If the file was not able to be opened, then the value returned by the *fopen* routine is NULL.
- For example, let's assume that the file *mydata* does not exist. Then:

```
FILE *fptr1 ;
fptr1 = fopen ( "myfile", "r") ;
if (fptr1 == NULL)
{
        printf ("File 'mydata' did not open.\n") ;
}
```

# Reading From Files

- In the following segment of C language code:

```
int a, b ;
FILE *fptr1, *fptr2 ;
fptr1 = fopen ( "mydata", "r" ) ;
fscanf ( fptr1, "%d%d", &a, &b) ;
```

the *fscanf* function would read values from the file "pointed" to by *fptr1* and assign those values to *a* and *b*.

# End of File

- The end-of-file indicator informs the program when there are no more data (no more bytes) to be processed.
- There are a number of ways to test for the end-of-file condition. One is to use the *feof* function which returns a true or false condition:

```
fscanf (fptr1, "%d", &var) ;
if ( feof (fptr1) )
{
        printf ("End-of-file encountered.\n");
}
```

- Another (better) way of testing EOF:

```
while(fscanf(fp,"%d ", &current) == 1)
{

}
```

# Writing To Files

```
int a = 5, b = 30;
FILE  *fptr2 ;
fptr2 = fopen ( “filename", "w" ) ;
fprintf ( fptr2, "%d %d\n", a, b ) ;
```

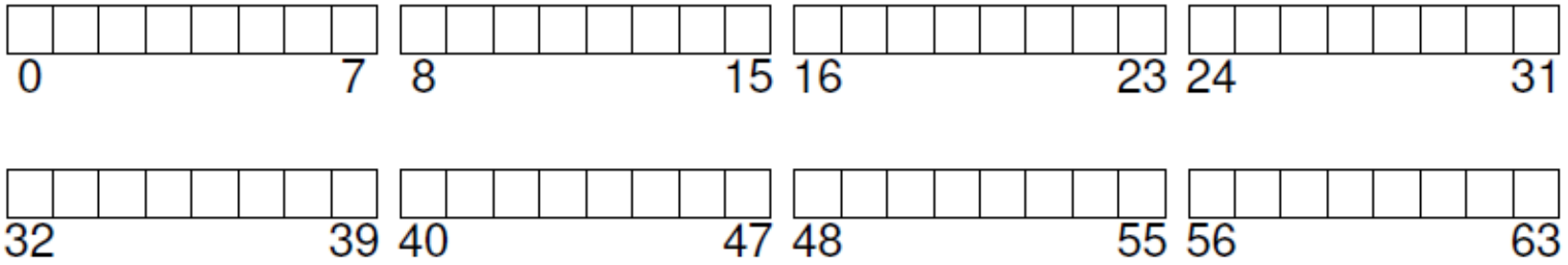
the fprintf functions would write the values stored in *a* and *b* to the file "pointed" to by *fptr2*.

# Closing Files

```
fclose ( fptr1 ) ;
```

Once the files are open, they stay open until you close them or end the program (which will close all files.)

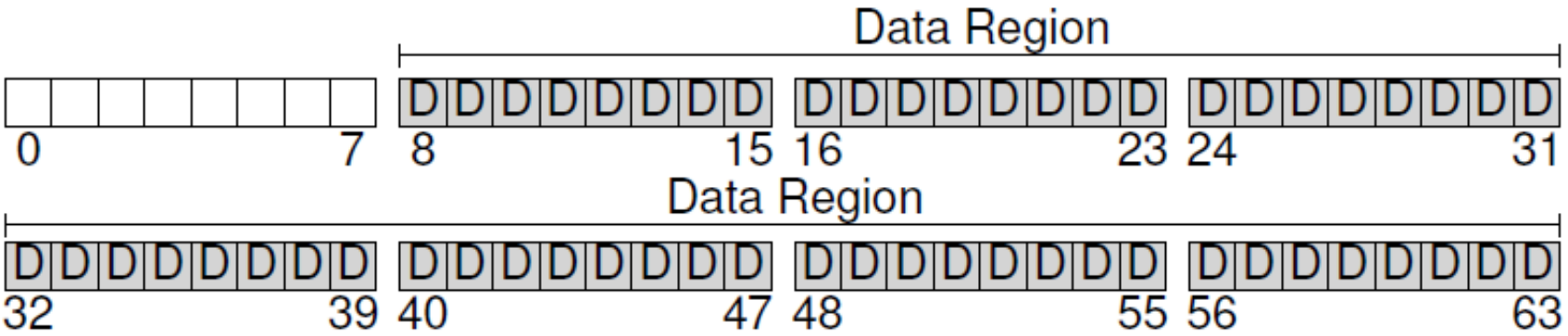
# Implementing Files (What happens at the low-level?)




Assumptions:

- A disk is a sequence of blocks.
- All blocks are of the same size.
- If the disk has N blocks, they are addressed 0 → N-1

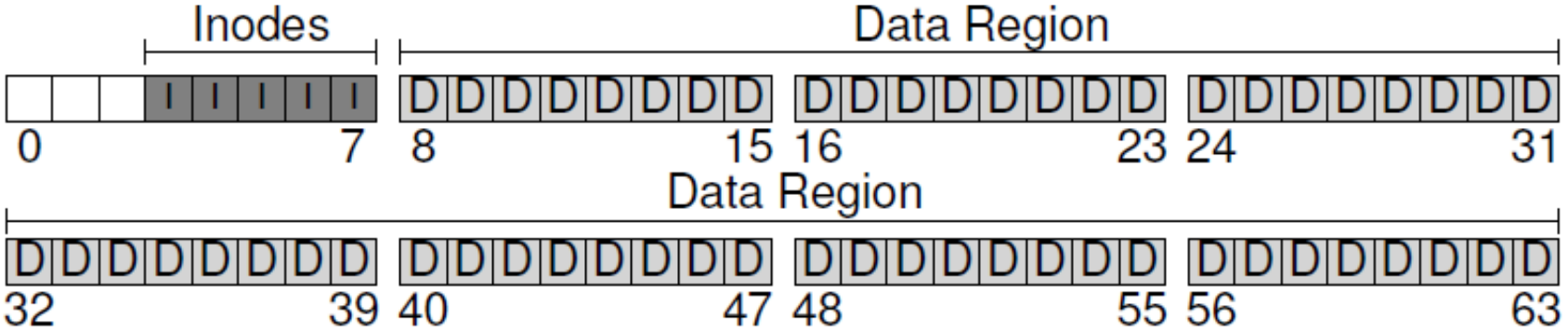
# Implementing Files (What happens at the low-level?)




- Most of the disk is reserved for the user data, shown as ‘D’ in the figure above.
- Each file consists of one or more blocks, not necessarily contiguous.

How to keep track which block(s) belong to which file?

# Implementing Files (What happens at the low-level?)

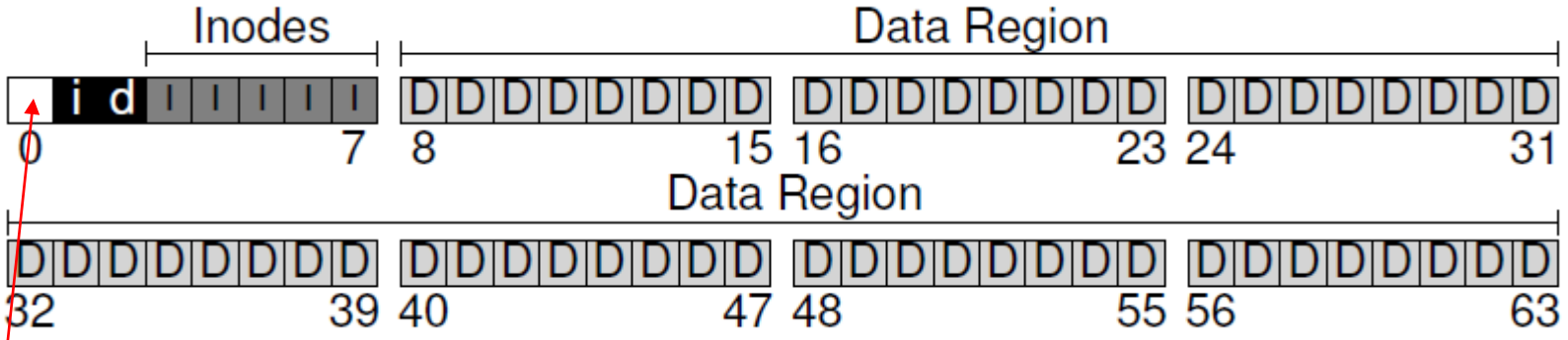



- Each file has an I-node, indicated by the "I" in the figure above.
- I-node stands for "Index-Nodes", partially for historical reasons.
- I-node contains all the meta-data of the file (security, pointer to first block, etc).
- Each block can hold many i-nodes because an i-node is small (e.g. 256 bytes) and a block is 4KB or more).

How to keep track:

- Which block holds i-nodes and which block holds user data?
- How to keep track of free blocks?

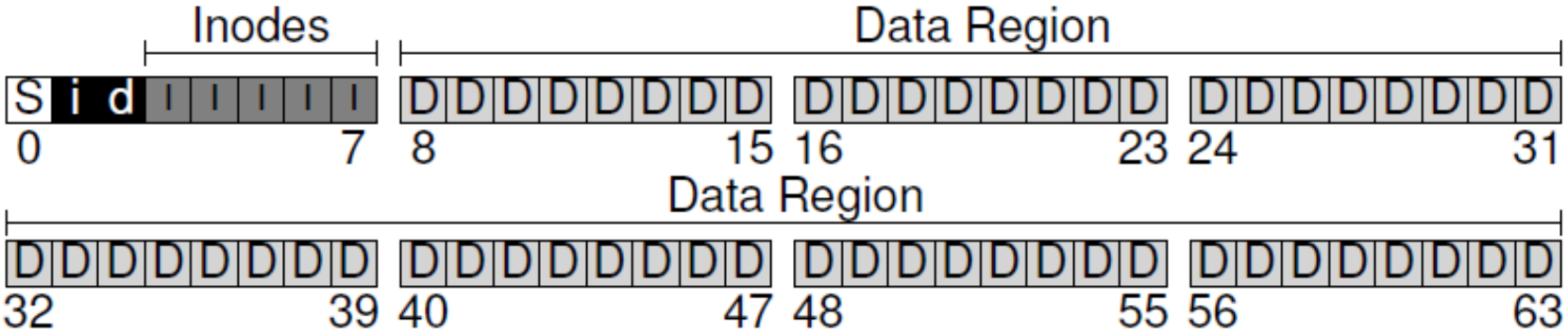
# Implementing Files (What happens at the low-level?)




- Both ‘i’ and ‘d’ blocks are bitmaps
- A bit can be 0 or 1
- 0 means block is empty and 1 means it is used.
- This way, we keep track of empty and used blocks in both in Inodes region and the data region.

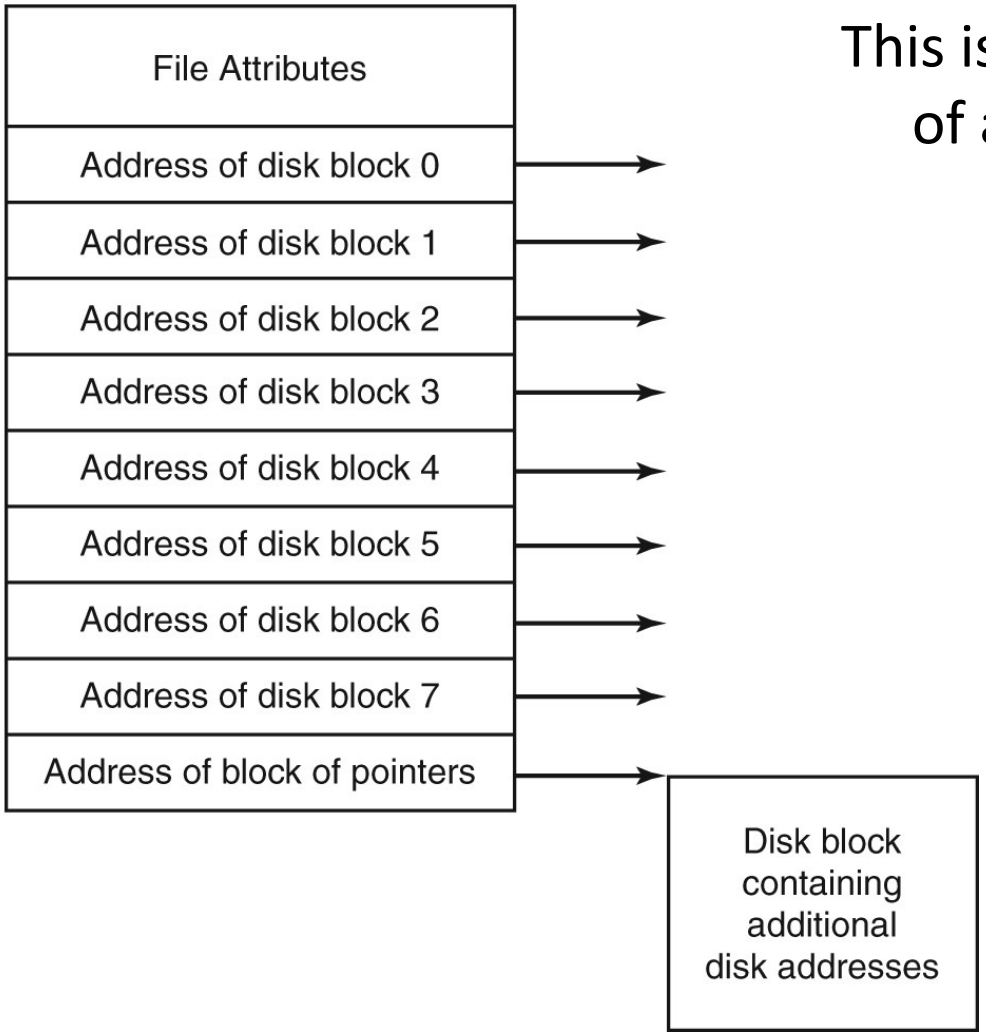
What is the job of that empty block at the beginning?

# Implementing Files (What happens at the low-level?)




- S stands for “superblock’.
- Contains information about this file-systems (e.g. type, number of i-nodes, etc).
- The OS reads the S block and attach this volume the file-system tree if
- there are other volumes.
- A volume can be a disk partition, another disk, etc.
- A directory is just a special type of files.

# How to deal with large files?




This is an example of an i-node.

# Summary of the Big Picture

- The programmer accesses the file using the API provided by the programming language used.
- This is compiled, and later executed, to a system call.
- Using the device independent I/O layer in the OS, the file access is translated to block(s) access request(s).
  - This is done using the file-system organization we saw in the several previous slides (i.e. I-nodes, etc).
- The device driver takes these requests and translate them to low-level access of the disk (e.g. read sector number a from track b on plate c).

# Conclusions

- Files and file system are major parts of an OS.
- We have just scratched the surface of file systems. There are more sophisticated way of implementing it but what we saw is the basics needed.
- Files and File system are the OS way of abstracting storage.