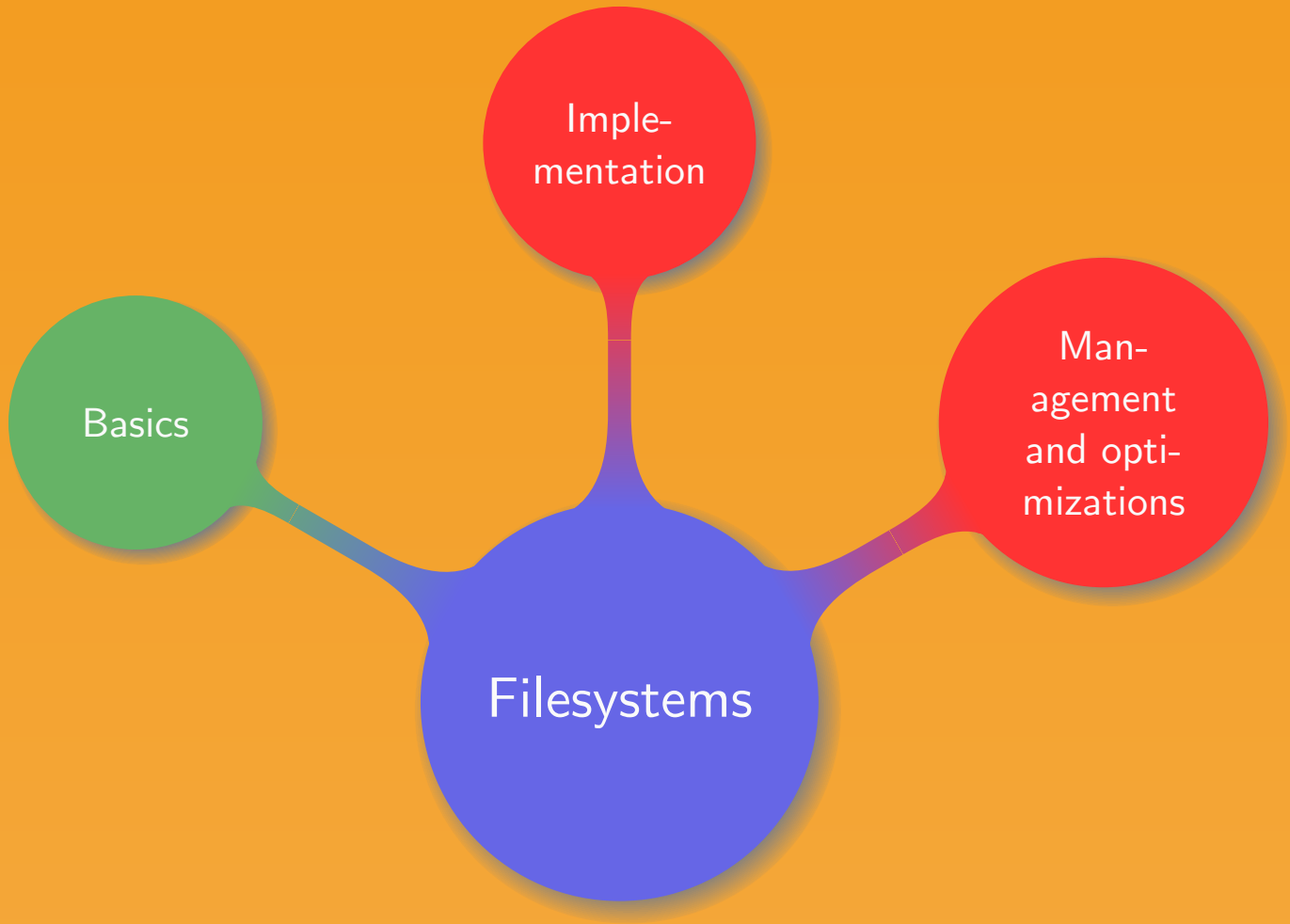
Introduction to Operating Systems

8. Filesystems

Manuel – Fall 2019

Chapter organisation



Goals

Limitations of virtual memory:

- Small
- Volatile
- Process dependent

Goals

Limitations of virtual memory:

- Small
- Volatile
- Process dependent

Goals that need to be achieved:

- Store large amount of data
- Long term storage
- Information shared among multiple processes

User's view

High level view of a file-system:

- Small part of the disk memory can be directly accessed using high level abstraction called a *file*
- File name can be case sensitive or insensitive
- File name is a string with (an optional) suffix
- Each file has some attributes containing special information

File operations

Common system calls related to files (Unix):

- Create
- Delete
- Rename
- Open
- Close
- Read
- Write
- Append
- Seek
- Set attributes
- Get attributes

File organisation

Structure content:

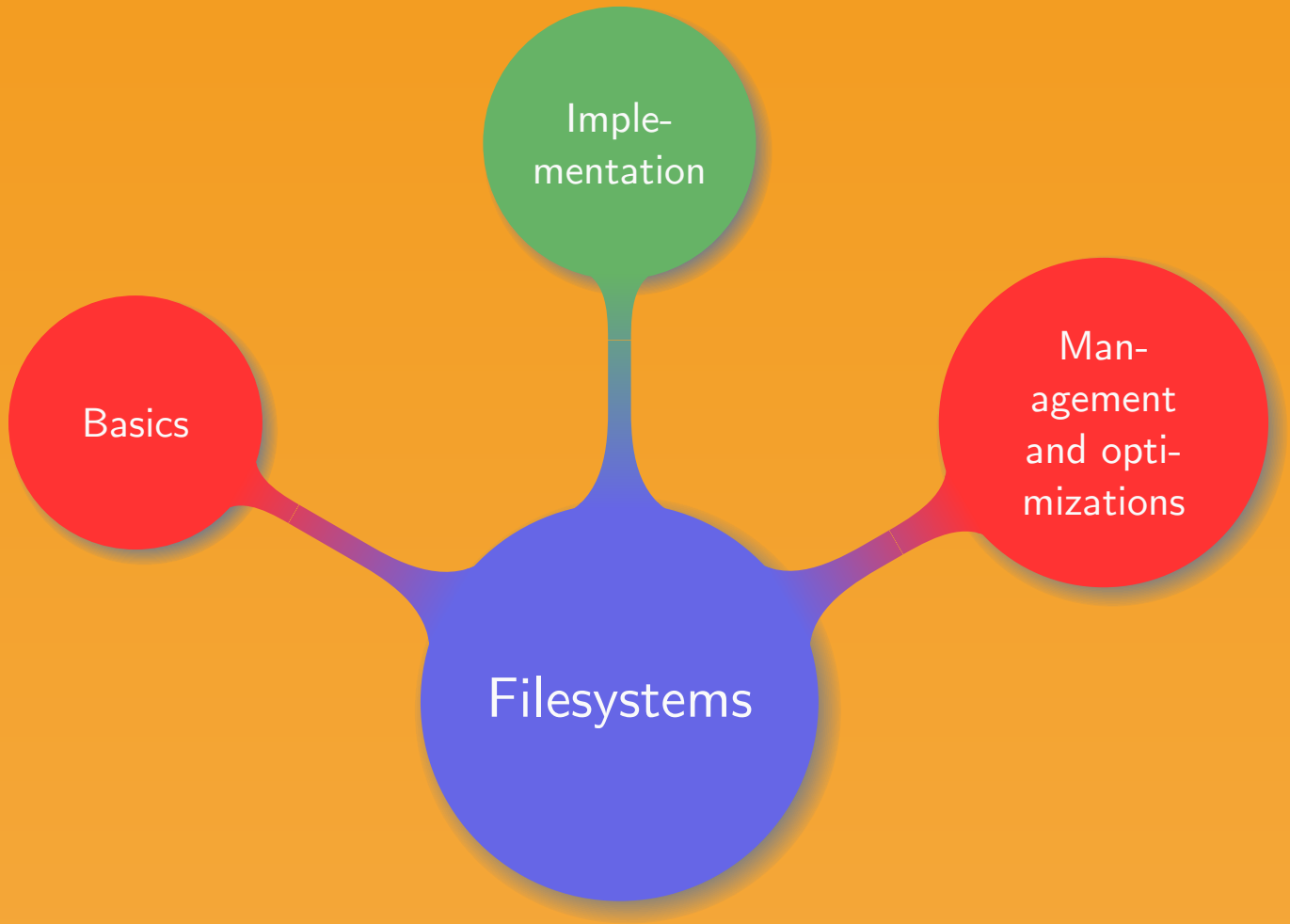
- Files are grouped inside a *directory*
- Directories are organised in a *tree*
- Each file has an *absolute path* from the root of the tree
- Each file has an *relative path* from the current location in the tree

Directory operations

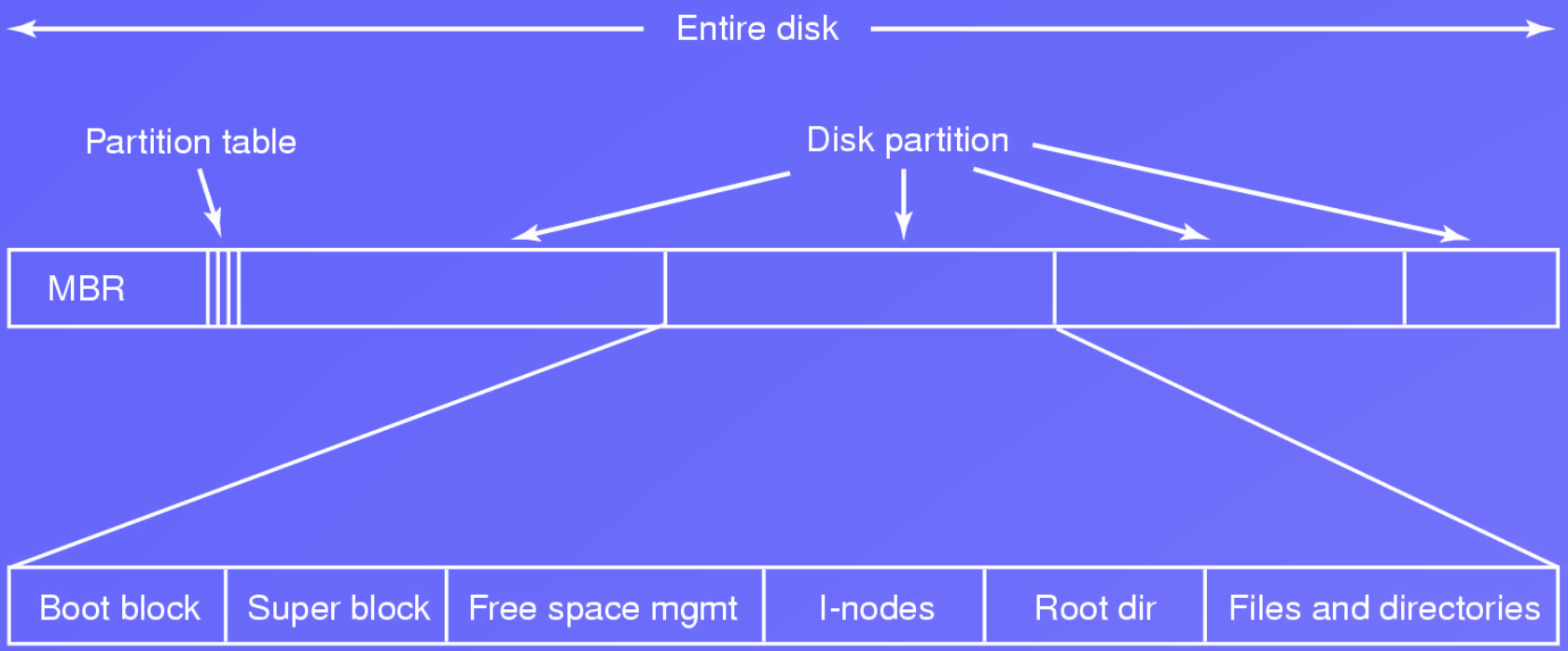
Common system calls related to directories (Unix):

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

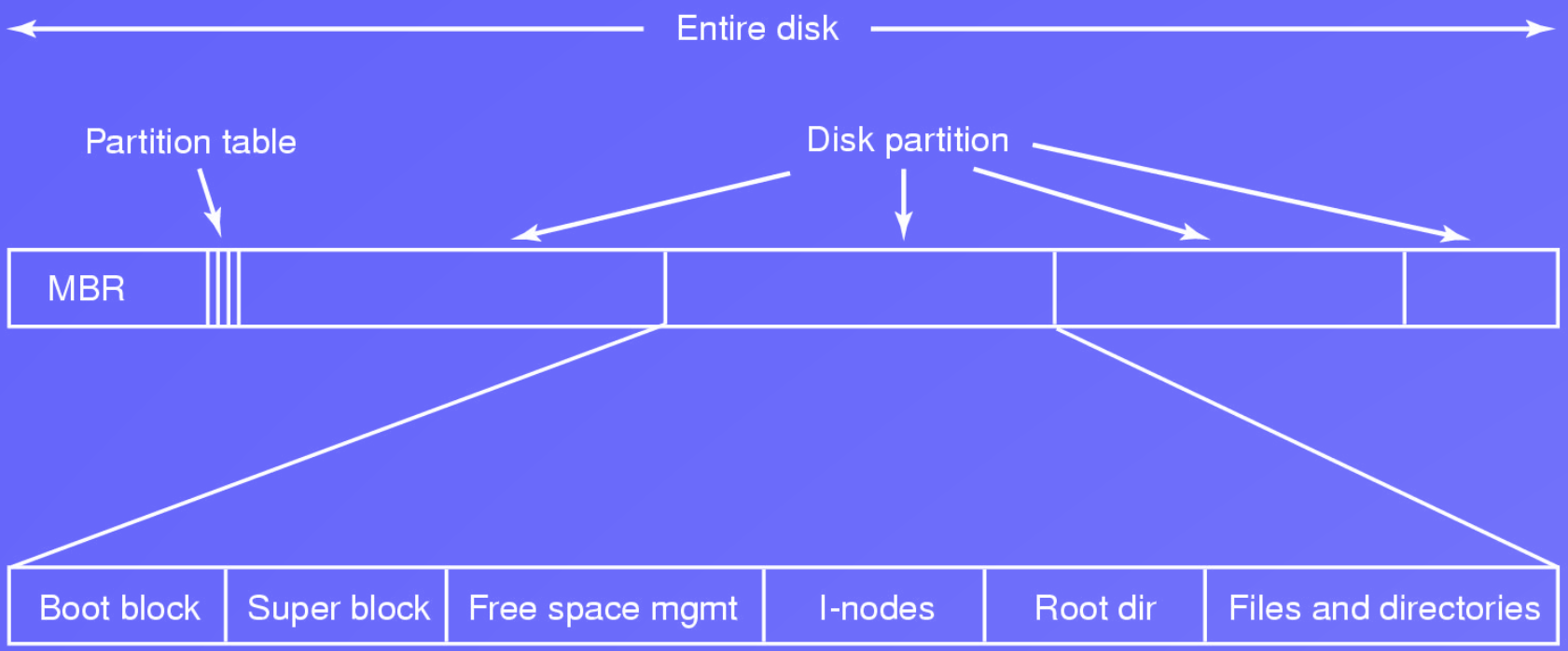
Chapter organisation



Basic disk layout

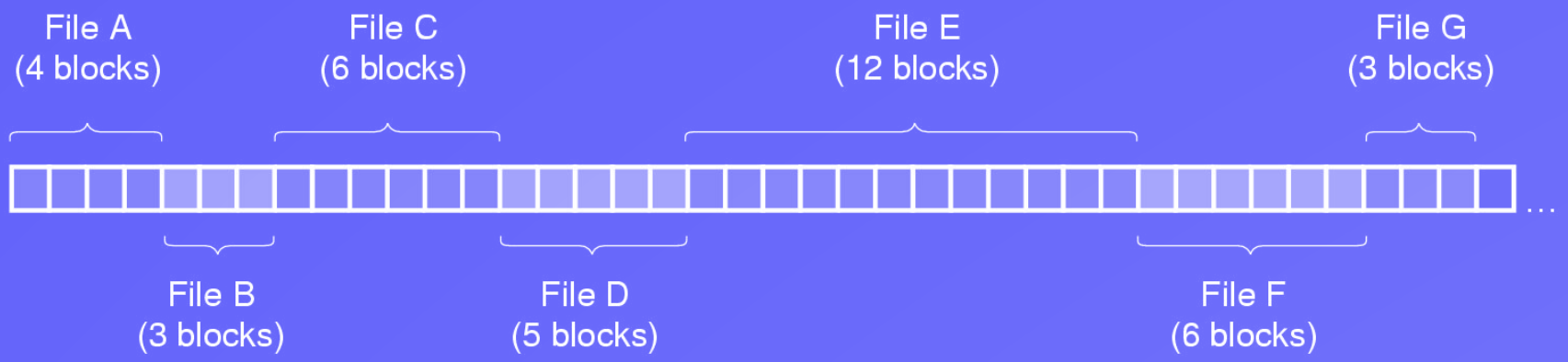


Basic disk layout



How to efficiently match disk blocks and files?

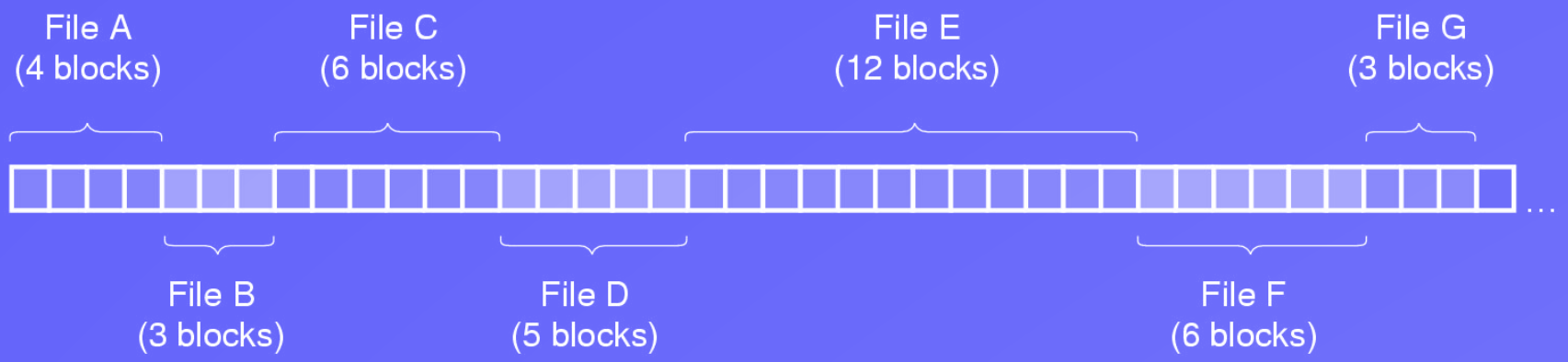
Contiguous allocation



Advantages:

- Simple to implement
- Fast: read a file using a single disk operation

Contiguous allocation

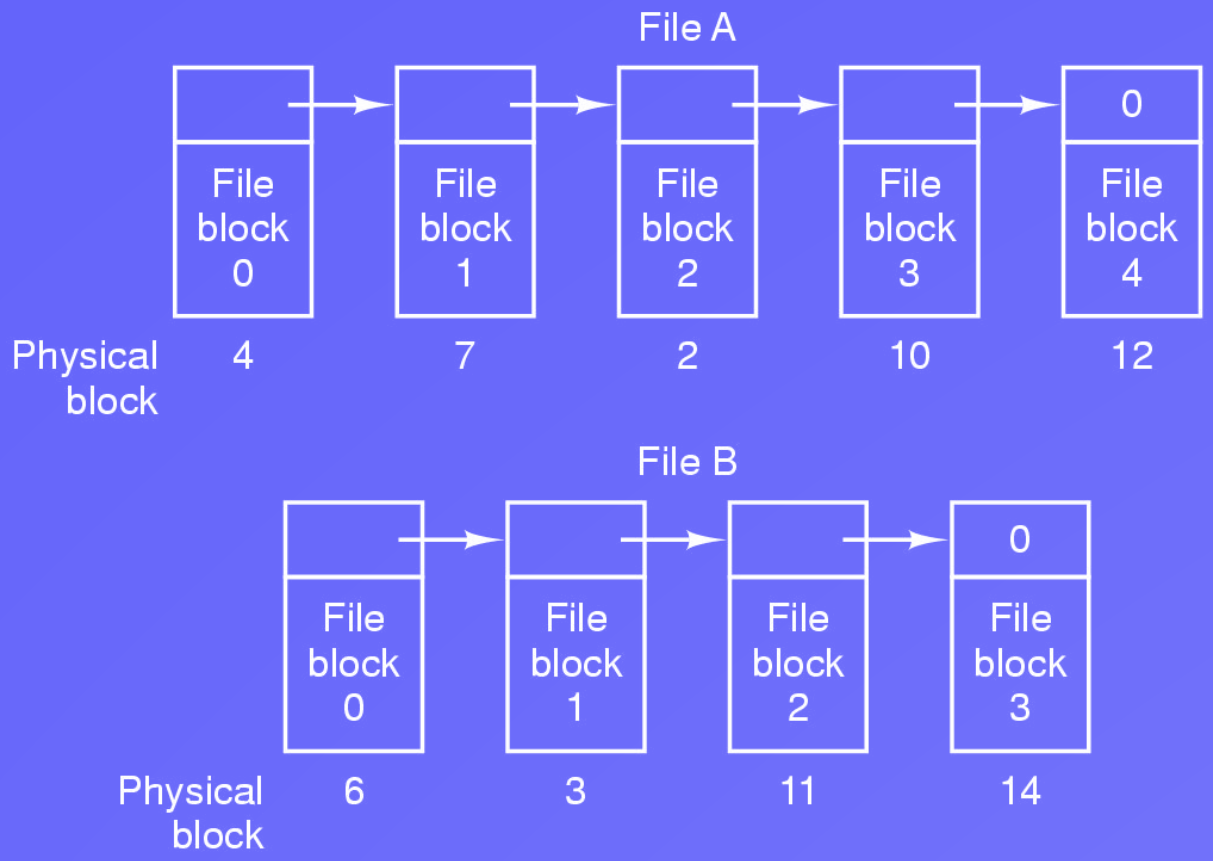


Advantages:

- Simple to implement
- Fast: read a file using a single disk operation

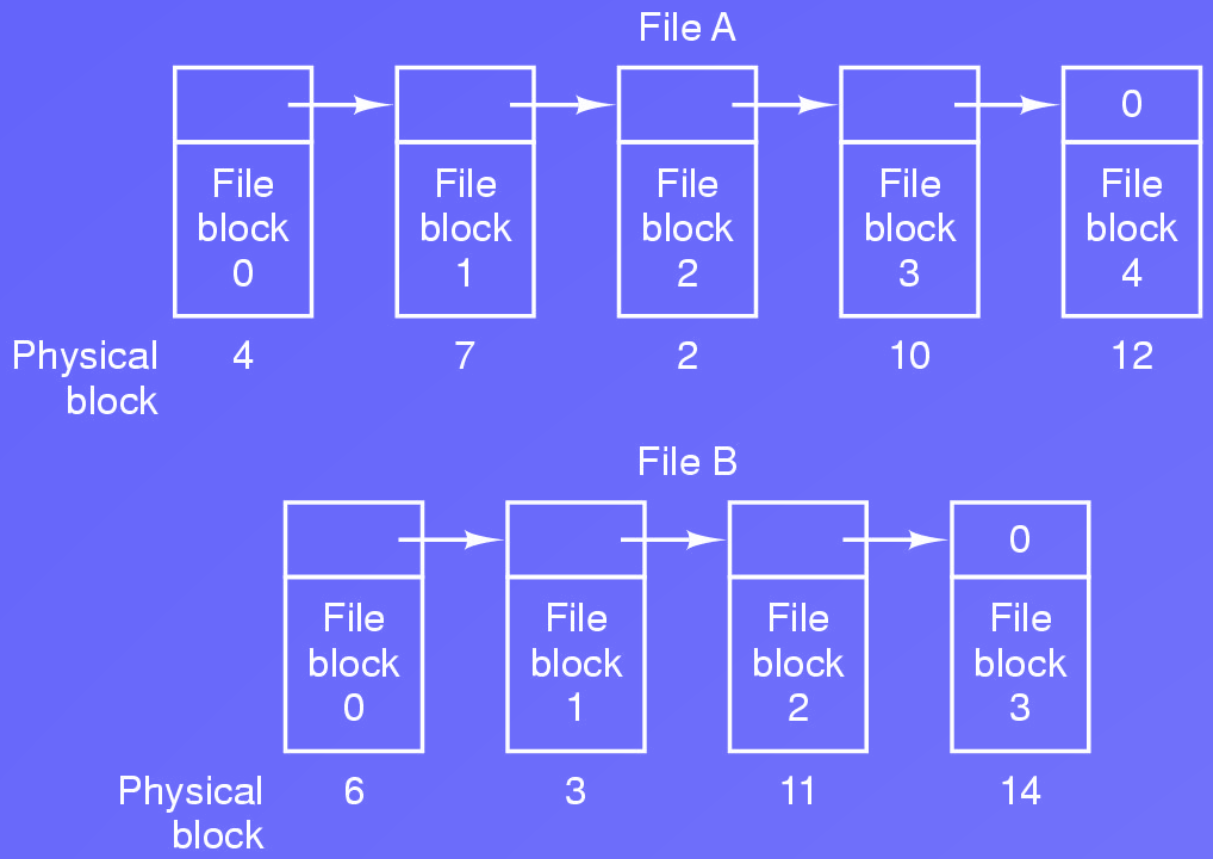
Drawback: what if files *D* and *F* are deleted?

Linked list



Advantage: no fragmentation

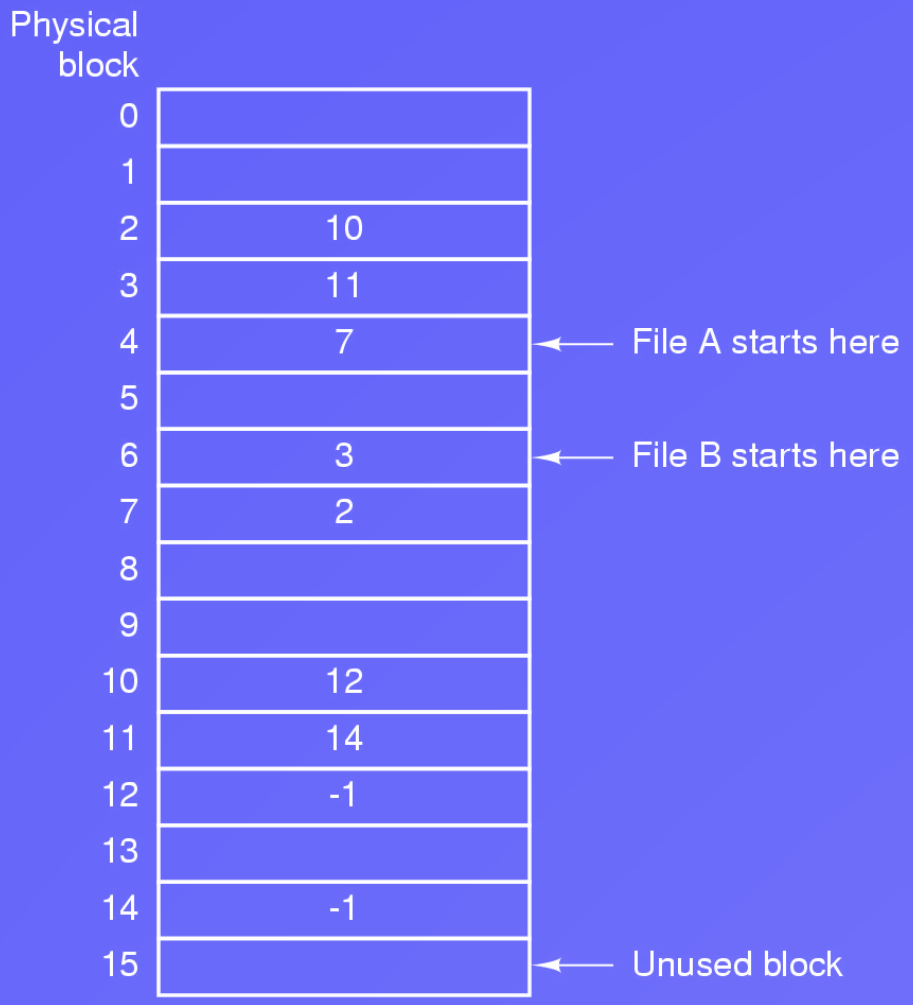
Linked list



Advantage: no fragmentation

Drawback: slow random access

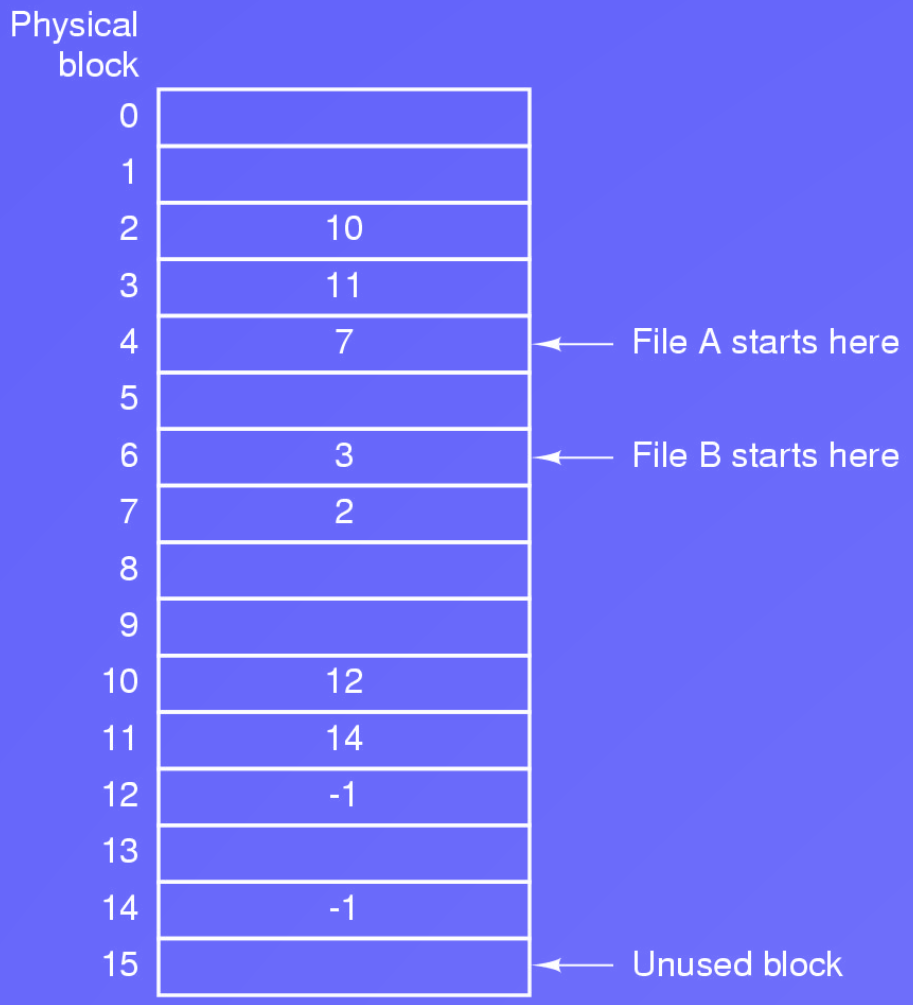
File Allocation Table



Idea: save the pointers on all the disk blocks inside a table in the main memory

Advantage: fast random access

File Allocation Table

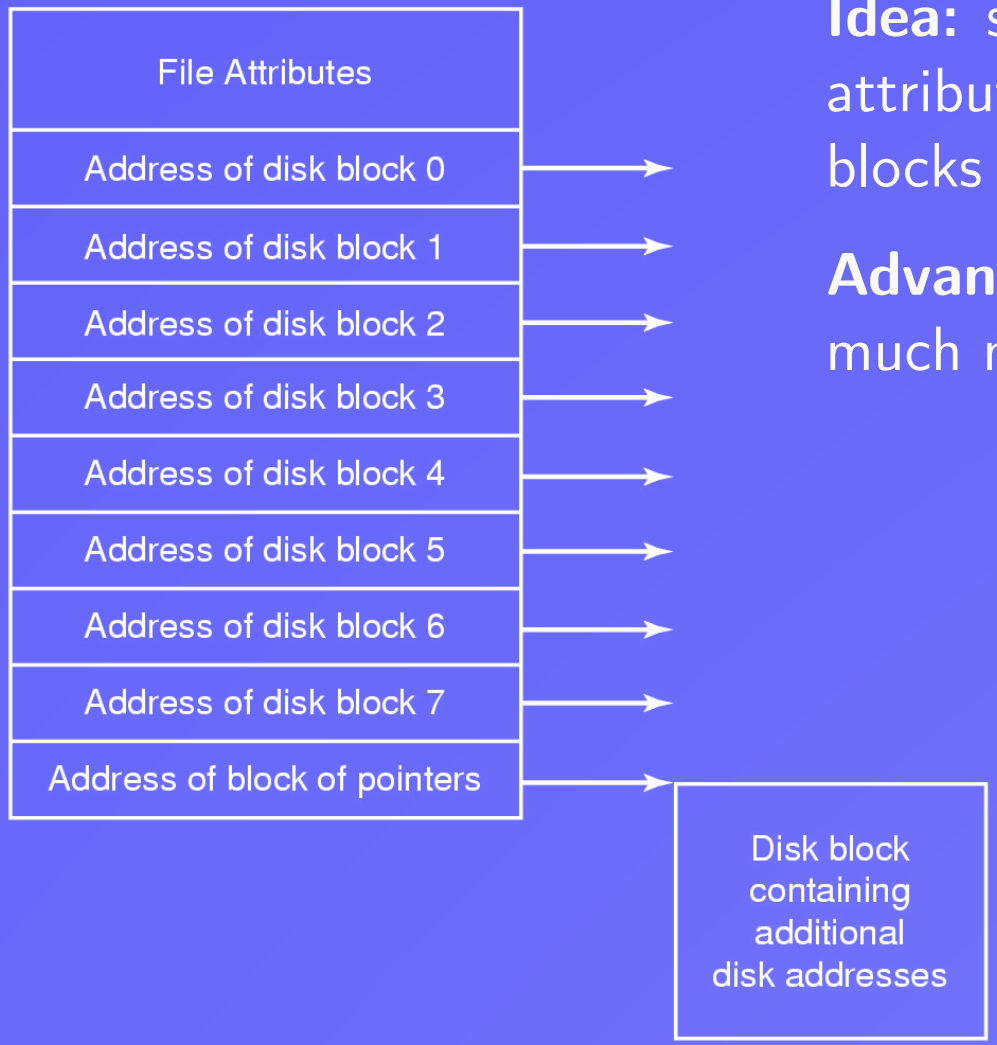


Idea: save the pointers on all the disk blocks inside a table in the main memory

Advantage: fast random access

Drawback: memory usage

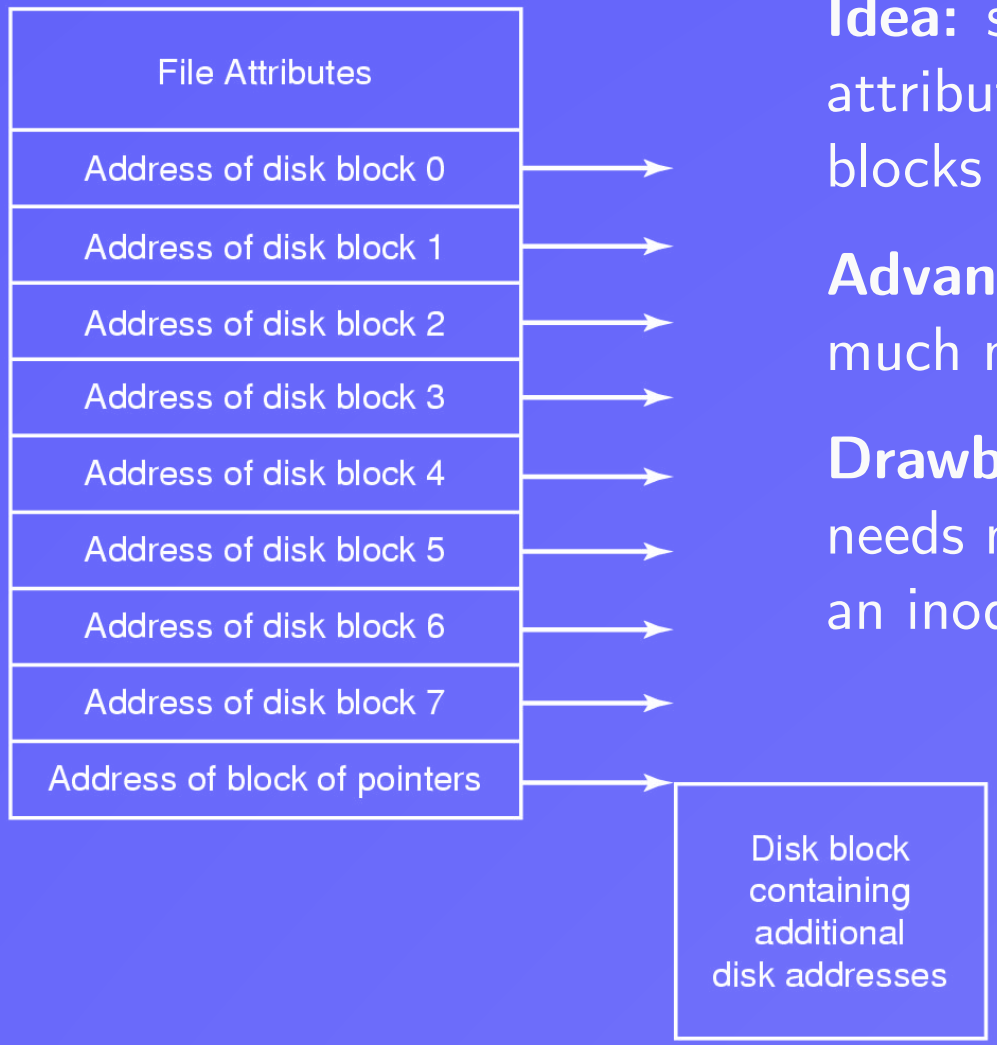
Index node



Idea: structure containing the file attributes and pointers on the blocks where the file is written

Advantage: fast, do not require much memory

Index node



Idea: structure containing the file attributes and pointers on the blocks where the file is written

Advantage: fast, do not require much memory

Drawback: what if a large file needs more blocks that can fit in an inode?

Directories

Simple design: fixed size entry (filename, attributes, disk address)

Directories

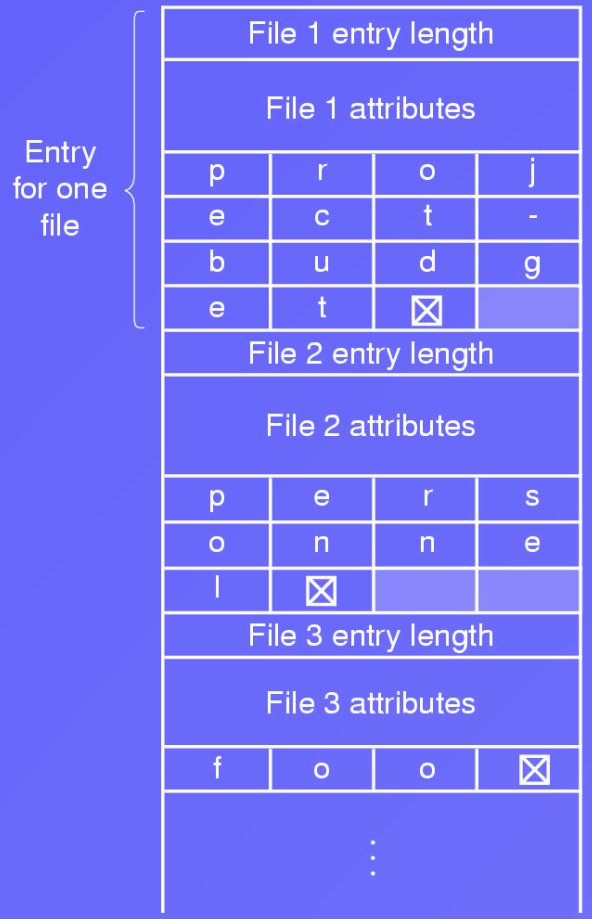
Simple design: fixed size entry (filename, attributes, disk address)

Drawback: how to handled long filenames?

Directories

Simple design: fixed size entry (filename, attributes, disk address)

Drawback: how to handled long filenames?



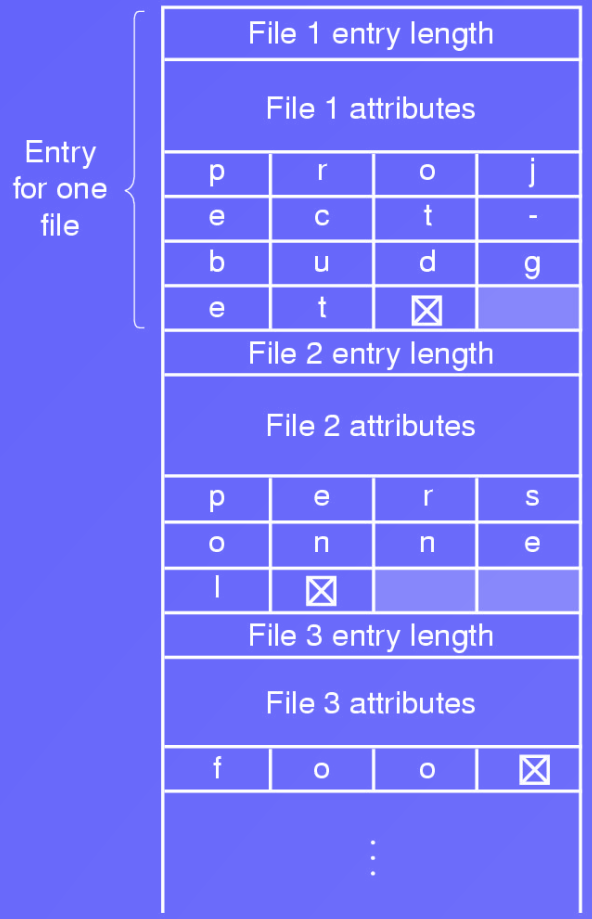
Idea: filename length not fixed

Advantage: can fit filename of arbitrary length

Directories

Simple design: fixed size entry (filename, attributes, disk address)

Drawback: how to handled long filenames?

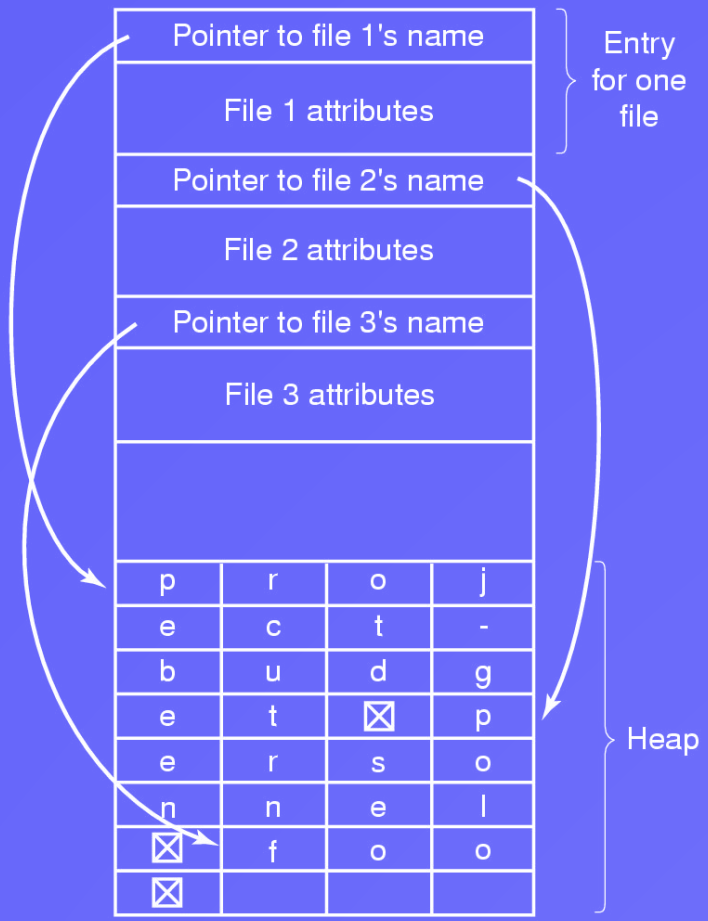


Idea: filename length not fixed

Advantage: can fit filename of arbitrary length

Drawback: space wasted, what if a directory entry spans multiple pages?

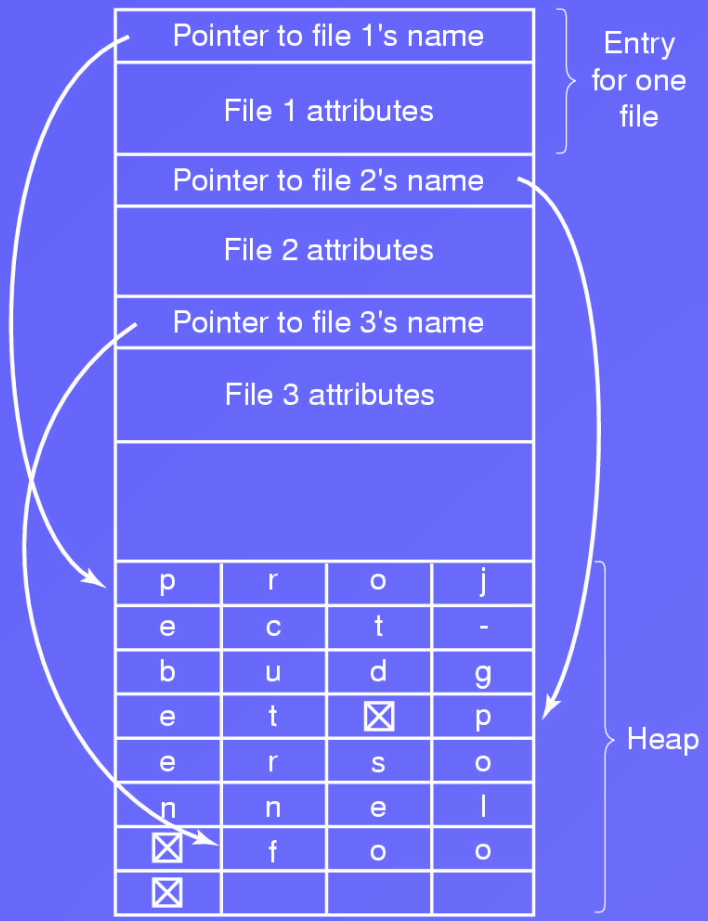
Directories



Idea: pointer to the filename

Advantage: no waste of space, space can be easily reused when a file is removed

Directories



Idea: pointer to the filename

Advantage: no waste of space, space can be easily reused when a file is removed

Drawback: as all the other strategies: slow on long directories

Journaling FS

Basic idea: log the operation to be performed, run it, and erase the log

Strategy: if a crash interrupts an operation, re-run it on next boot

Journaling FS

Basic idea: log the operation to be performed, run it, and erase the log

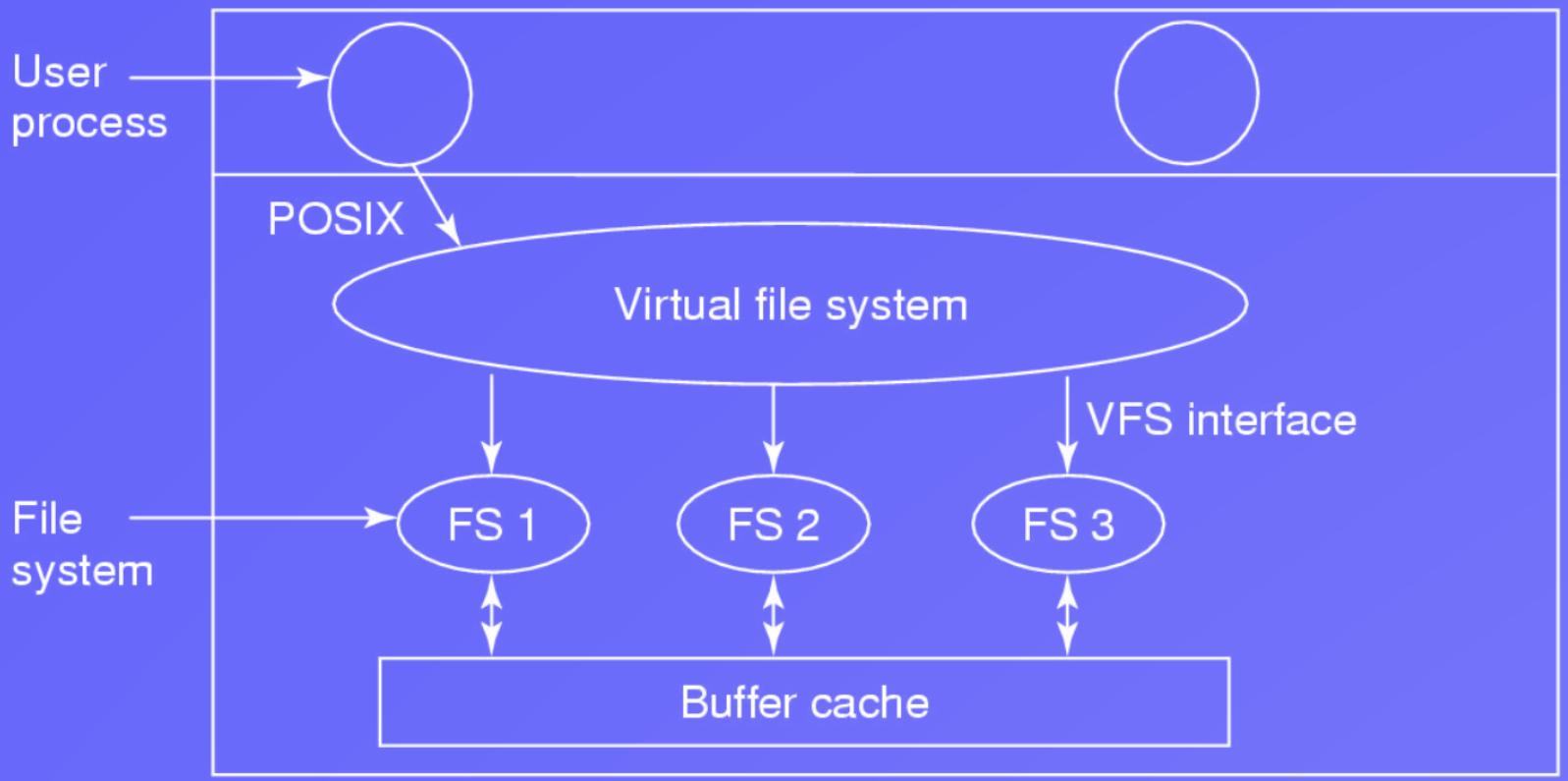
Strategy: if a crash interrupts an operation, re-run it on next boot

Problem: can any operation be applied more than once?

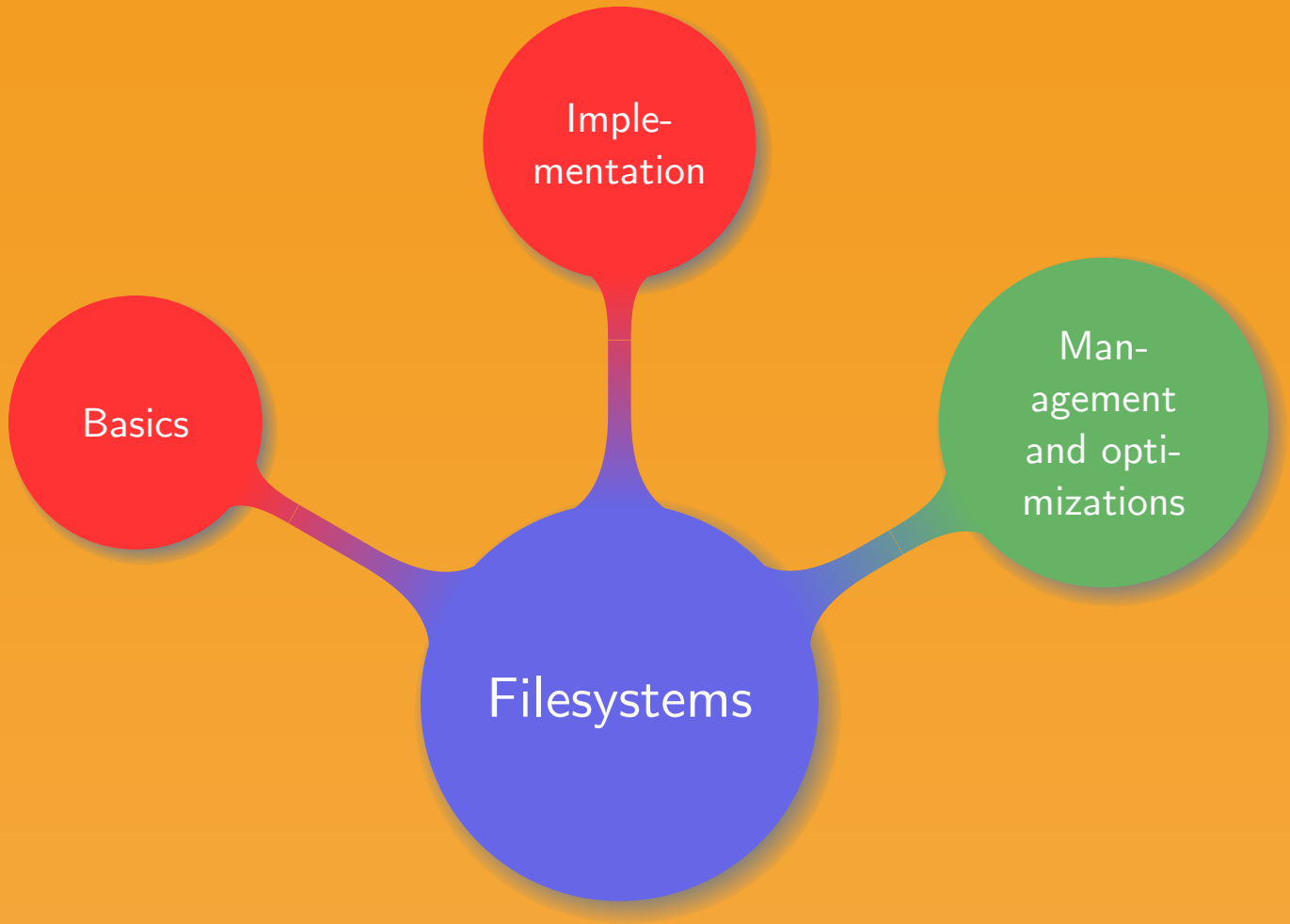
Example.
File deletion:

- (i) remove file from directory, (ii) release its i-node and (iii) add its disk blocks to the list of free blocks
- Operations (i) and (ii) can be repeated not (iii)

Virtual FS


User process
POSIX
Virtual file system
VFS interface
File system
FS 1
FS 2
FS 3
Buffer cache

Chapter organisation



Block size

Problem: how big should a block be?

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

Conclusion: time wasted

Block size

Problem: how big should a block be?

Using small blocks:

- Large files use many blocks
- Blocks are not contiguous

Conclusion: time wasted

Using large blocks:

- Small files do not fill up the blocks
- Many blocks partially empty

Conclusion: space wasted

Free blocks

Problem: how to keep track of free blocks?

Free blocks

Problem: how to keep track of free blocks?

- *Using a linked list:* free blocks addresses are stored in a block
  e.g. using 4KB blocks with 64 bits block address, how many free blocks addresses can be stored in a block?
- *Using a bitmap:* one bit corresponds to one free block
- *Using consecutive free blocks:* a starting block and the number of free block following it

Which strategy is best?

FS consistency

Checking the FS:

- Using the i-nodes, list in all the blocks used by all the files. Compare the complementary to the list of free blocks
- For every i-node in every directory increment a counter by 1. Compare those numbers with the counts stored in the i-nodes

FS consistency

Checking the FS:

- Using the i-nodes, list in all the blocks used by all the files. Compare the complementary to the list of free blocks
- For every i-node in every directory increment a counter by 1. Compare those numbers with the counts stored in the i-nodes

Common problems and solutions:

- Block related inconsistency:
  - List of free blocks is missing some blocks $\rightarrow$ add blocks to list
  - Free blocks appear more than once in list $\rightarrow$ remove duplicates
  - A block is present in more than one file $\rightarrow$ copy block and add it to the files
- File related inconsistency:
  - Count in i-node is higher $\rightarrow$ set link count to accurate value
  - Count in i-node is lower $\rightarrow$ set link count to accurate value

Caching

Idea: keep in memory some disk blocks using the LRU algorithm

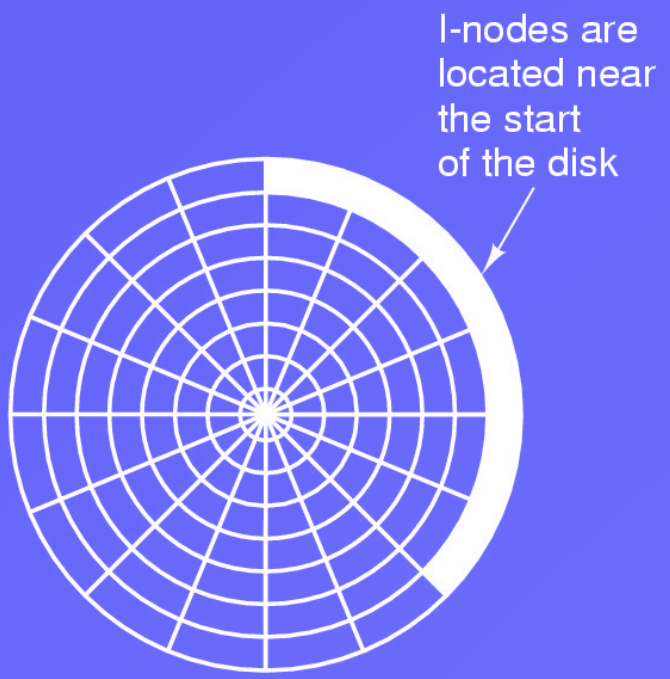
Questions:

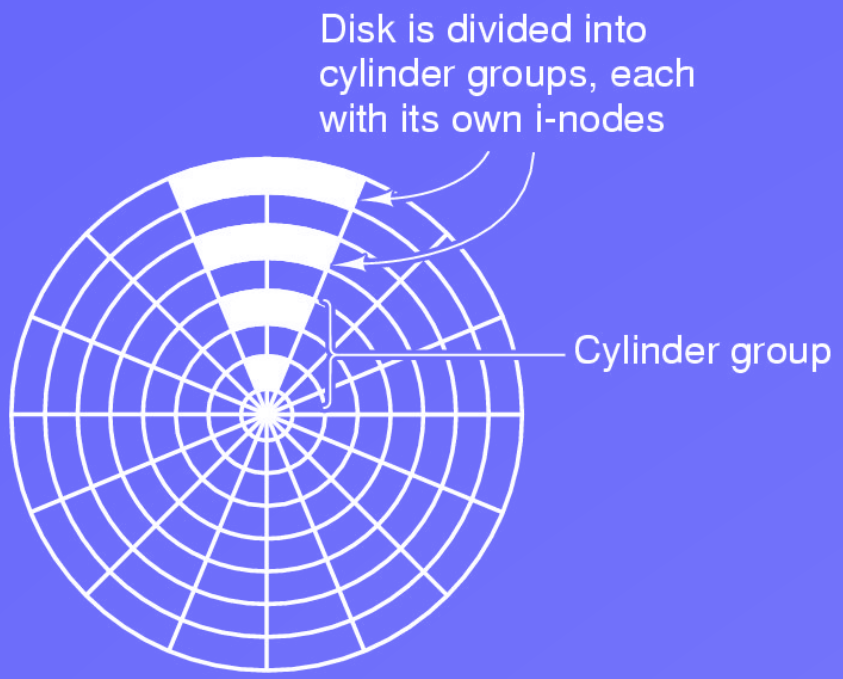
- Is a block likely to be reused soon?
- What happens on a crash?

Modified idea:

- Useless to cache i-node blocks
- Dangerous to cache blocks essential to file system consistency
- Cache partially full blocks that are being written

Arm motion


I-nodes are
located near
the start
of the disk


Disk is divided into
cylinder groups, each
with its own i-nodes
Cylinder group

Notes

A few extra remarks related to file systems:

- Quotas: assign disk quotas to users
- Fragmentation: how useful is it to defragment a file system?
- Block read ahead: when reading block k assume $k + 1$ will soon be needed and ensure its presence in the cache
- Logical volumes: file system over several disks
- Backups: how to efficiently backup a whole filesystem?
- RAID: Redundant Arrays of Inexpensive Disks

Key points

- What are the three main goals of a file system?
- Describe a basic disk layout
- Explain the structure of an i-node
- Mention three challenges in the design of a file system

Thank you!