Karlsruhe Institute of Technology

Operating Systems

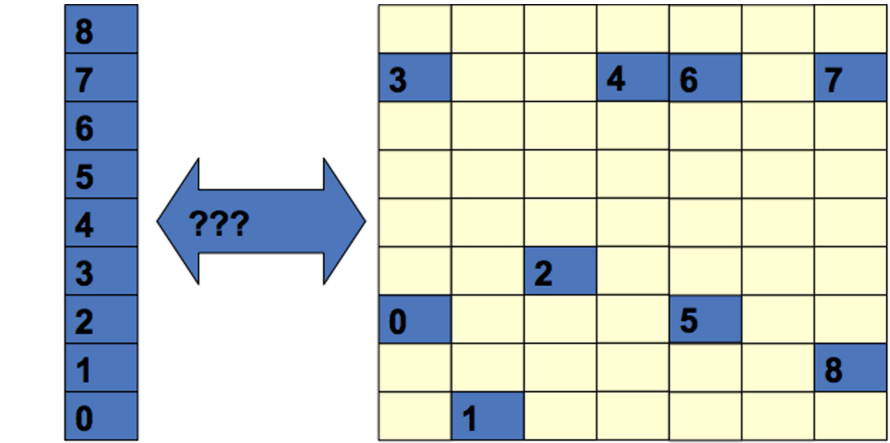
18. Implementing File Systems (Overview and File Allocation)

Prof. Dr. Frank Bellosa | WT 2021/2022

Implementing File Systems

- File Implementation
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation
- Directory Implementation
- File-System Structures
- Examples
  - UNIX
  - FFS/Ext2
  - Journaling
  - Log-structured File Systems

Implementing Files



File with a set of logical file blocks (records)

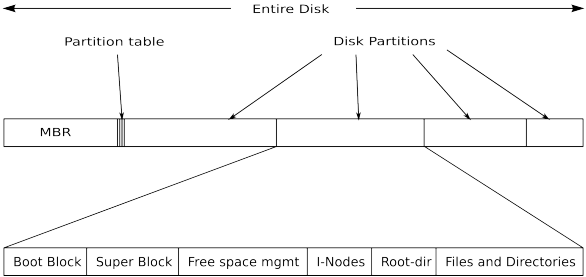
Disk with allocated and free physical disk blocks

Disk Structure

- Disk can be subdivided into partitions
- Disks partitions[1] can be used raw – without a file system – or formatted with a file system (FS)
- Entity containing a FS is known as a volume
- Each volume containing a FS also tracks that FS's info in the device directory or the volume table of contents
- Besides general-purpose FSs there are many special purpose FSs, frequently all within the same operating system or computer

[1] Partitions also known as minidisks, slices

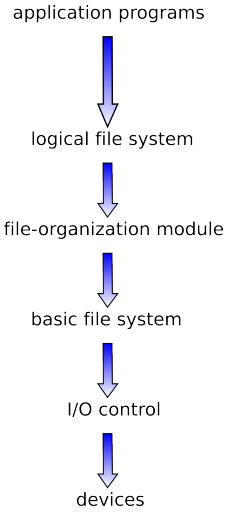
Implementing a FS on Disk



- Possible FS layout per partition
- Sector 0 of disk = MBR
  - Boot info (if PC is booting, BIOS reads and executes MBR)
  - Disk partition info
- Sector 0 of partition is volume boot record

[TB15]

Layered File System [SGG12]



A Typical File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Implementing Files

- FS must keep track of meta data
  - *Which logical block belongs to which file?*
  - *In what order are the blocks that form the file?*
  - *Which blocks are free for the next allocation?*
- Given a logical region of a file, the FS must identify the corresponding block(s) on disk
  - Needed meta data stored in
    - File allocation table (FAT)
    - Directory
    - Inode
- Creating (and updating) files might imply allocating new blocks (and modifying old blocks) on the disk

Allocation Policies

- Preallocation:
  - Need to know maximum size of a file at creation time
    (in some cases no problem, e.g. file copy)
  - Difficult to reliably estimate maximum size of a file
  - Users tend to overestimate file size, just to avoid running out of space
- Dynamic allocation:
  - Allocate in pieces as needed

Fragment Size [2]

- Extremes
  - Fragment size = length of file
  - Fragment size = smallest disk block size (sector size)
- Tradeoffs:
  - Contiguity $\Rightarrow$ speedup for sequential accesses
  - Many small fragments $\Rightarrow$ larger tables needed to manage free storage management as well as to support access to files
  - Larger fragments help to improve data transfer
  - Fixed-size fragments simplify reallocation of space
  - Variable-size fragments minimize internal fragmentation, but can lead to external fragmentation

[2] see page size discussion

Implementing Files

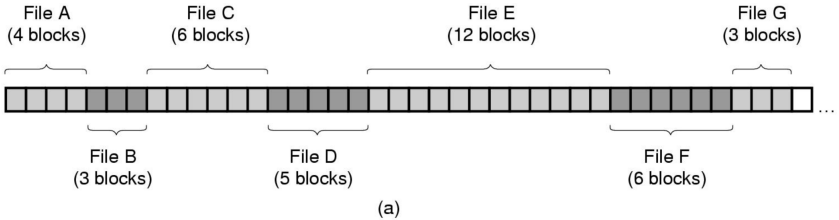
- 3 ways of allocating space for files:
  - contiguous
  - chained
  - indexed
    - fixed block fragments
    - variable block fragments

Contiguous Allocation

- Array of N contiguous logical blocks reserved per file (to be created)
- Minimum meta data per entry in FAT/directory
  - Starting block address
  - N
- *What is a good value for N?*
- *What to do with an application that needs more than N blocks?*
- Discussion similar to ideal page size
  - Internal Fragmentation
  - External Fragmentation

$\Rightarrow$ scattered disk

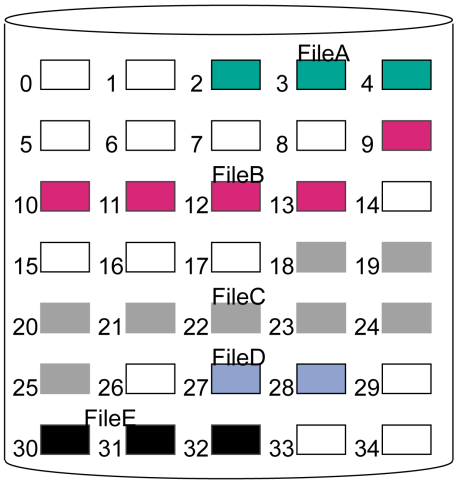
Scattered Disk [TB15]



(a) Contiguous allocation of disk space for 7 files

(b) State of the disk after files D and F have been removed

Contiguous File Allocation [Sta17]

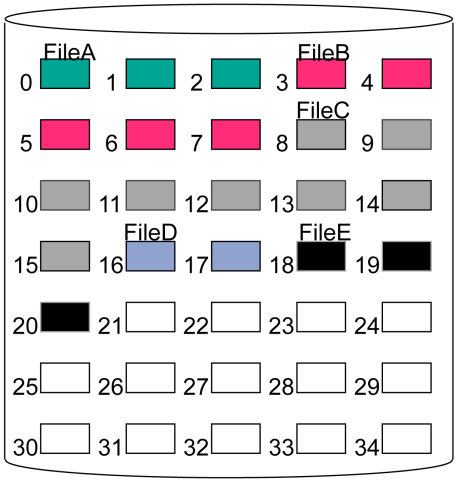


File Allocation Table

File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	27	2
FileE	30	3

Remark: To overcome external fragmentation $\Rightarrow$ periodic compaction

Contiguous File Allocation after Compaction)[Sta17]



File Allocation Table

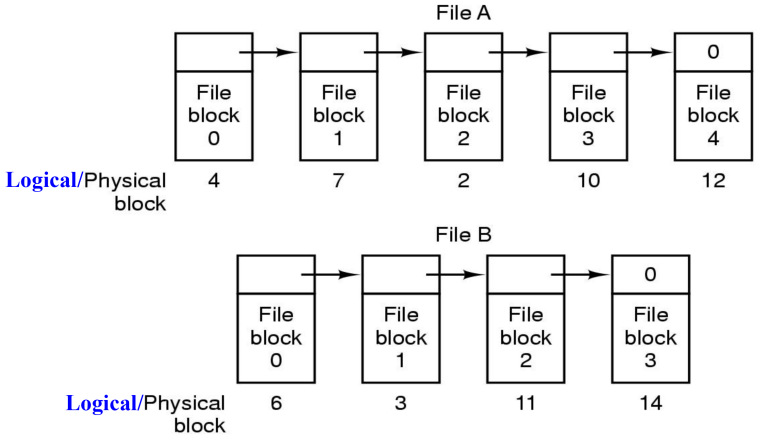
File Name	Start Block	Length
FileA	**0**	3
FileB	**3**	5
FileC	**8**	8
FileD	**16**	2
FileE	**18**	3

Chained Allocation (Linked List) (1)

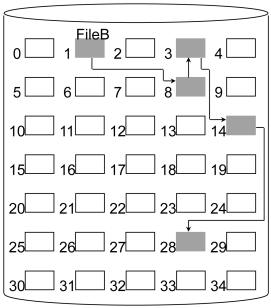
- Per file a linked list of logical file blocks, i.e.
  - Each file block contains a pointer to next file block, i.e. the amount of data space per block is no longer a power of two,
    $\Rightarrow$ Consequences?
  - Last block contains a NIL-pointer (e.g. -1)
- FAT or directory contains address of first file block
- No external fragmentation
  - Any free block can be added to the chain
- Only suitable for sequential files
- No accommodation of the principle of disk locality
  - File blocks will end up scattered across the disk
  - Run a defragmentation utility to improve situation

Chained Allocation (2)

- Storing a file as a linked list of disk blocks [TB15]



Chained Allocation (3) [Sta17]



File Allocation Table

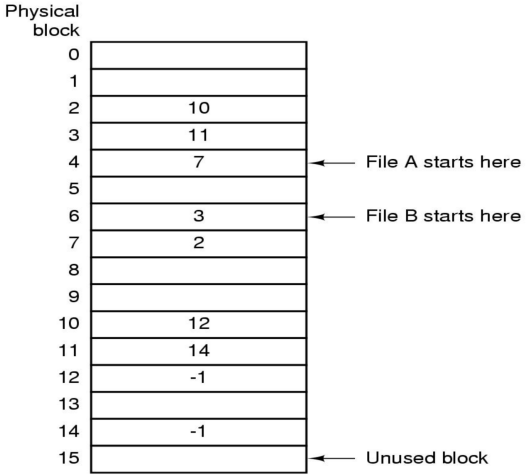
File Name	Start Block	Length
...	...	...
FileB	1	5
...	...	...

Remark: If you only access sequentially this implementation is acceptable (multiple requests with DMA?).
However requesting an individual record requires tracing through the chained blocks, which requires far too many disk accesses.

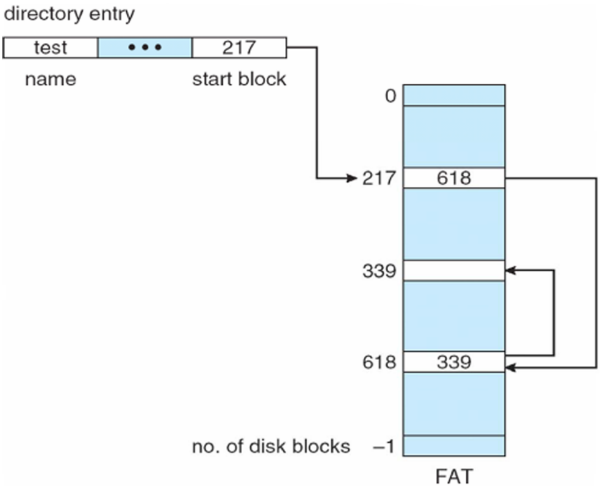
Linked List Allocation within RAM [TB15]

- Each file block only used for storing file data
- Linked list allocation with FAT in RAM
  - Avoids disk accesses when searching for a block
  - Entire block is available for data
  - Table gets far too large for modern disks
    - Can cache only, but still consumes significant RAM
    - Used in MS-Dos, OS/2
- Similar to an inverted page table, one entry per disk block

[TB15]



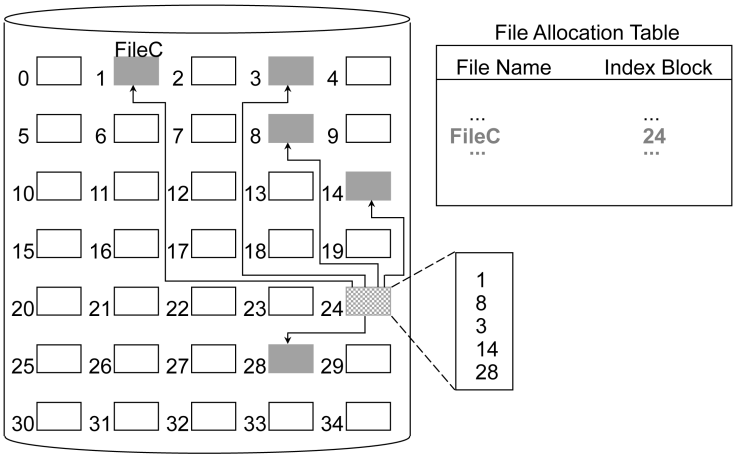
File Allocation Table [SGG12]



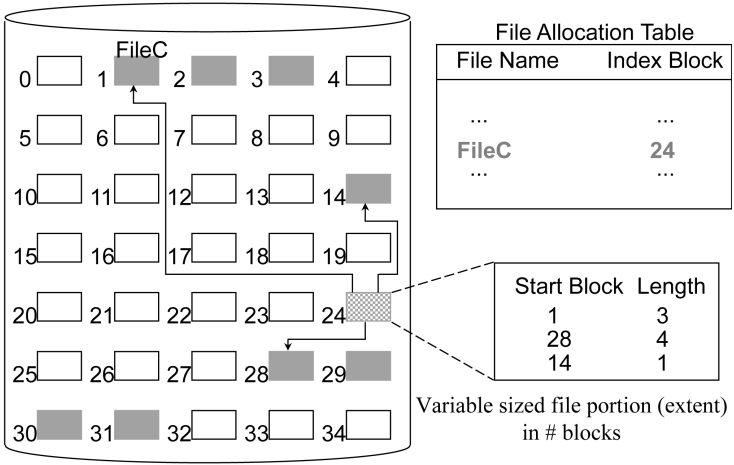
Indexed Allocation (1)

- FAT (or special inode table) contains a one-level index table per file
  - Fixed-size array of fragment pointers
    - Fixed block fragments
    - Variable block fragments
- Allocate space for pointers at file creation time

Indexed Allocation (2)[Sta17]



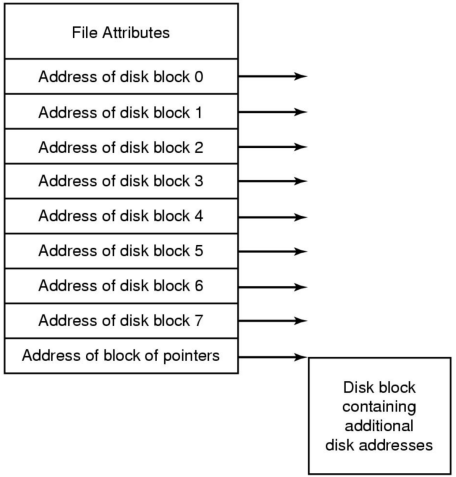
Indexed Allocation (3)[Sta17]



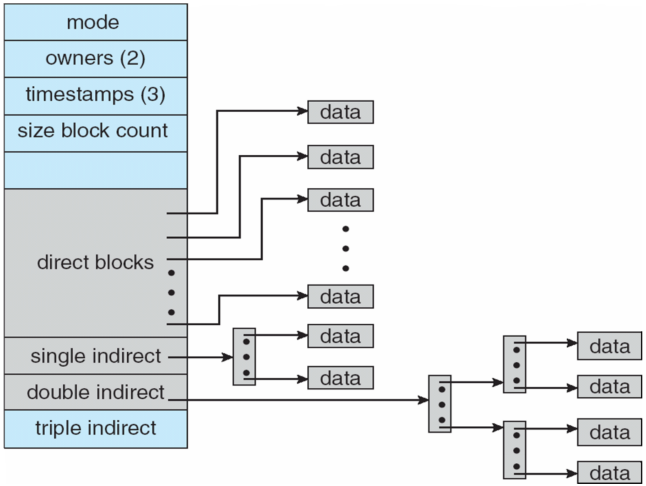
Analysis of Indexed Allocation

- Supports sequential and random access to a file
- Fragments
  - Block sized
    - Eliminates external fragmentation
    - Large overhead for unneeded pointers (most files are small!)
  - Variable sized
    - Improves contiguity
    - Reduces index size

Multi-Level-Indexing [TB15]



Example: UNIX (4k bytes per block)[SGG12]



Summary: File Allocation Methods

characteristic	contiguous	chained	indexed	
preallocation?	necessary	possible	possible	
fixed or variable size fragment?	variable	fixed	fixed	variable
fragment 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

References

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[Sta17] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, USA, 9th edition, 2017.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.