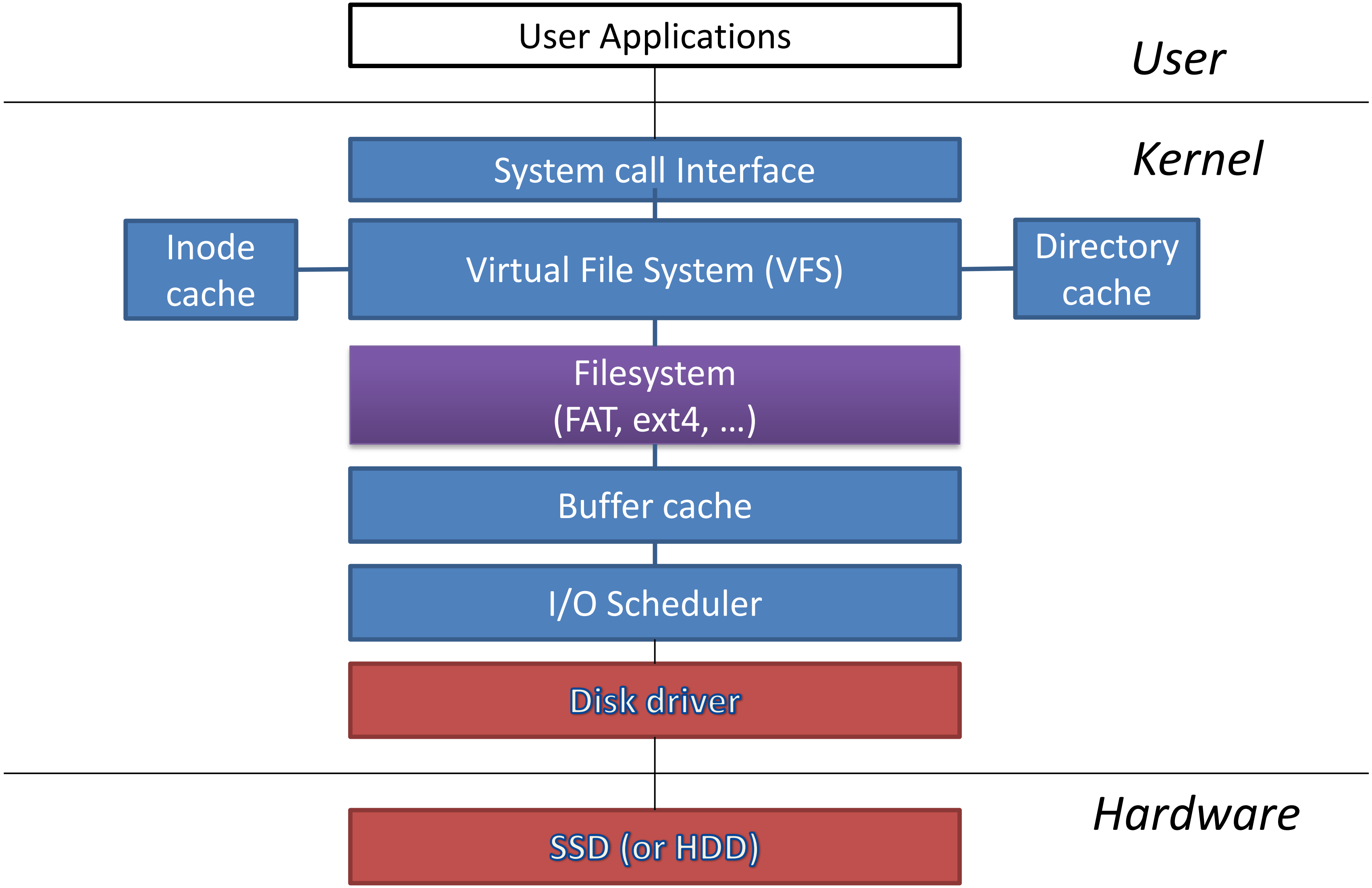
Filesystem

Disclaimer: some slides are adopted from book authors' slides with permission

Storage Subsystem in Linux OS



Filesystem

- Definition
  - An OS layer that provides **file** and **directory** abstractions on disks

- File
  - User's view: a collection of **bytes** (non-volatile)
  - OS's view: a collection of **blocks**
    - A block is a logical transfer unit of the kernel (typically block size >= sector size)

Filesystem

- File types
  - – Executables, DLLs, text, word, ….
  - – Filesystems mostly don’t care

- File attributes (metadata)
  - – Name, location, size, protection, …
- File operations
  - – Create, read, write, delete, seek, truncate, …

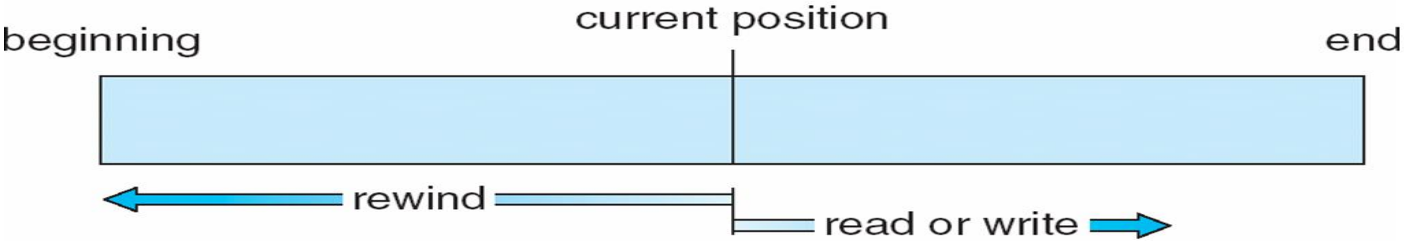
How to Design a Filesystem?

- What to do?
  - – Map disk blocks to each file
  - – Need to track free disk blocks
  - – Need to organize files into directories

- Requirements
  - – Should not waste space
  - – Should be fast

Access Pattern

- Sequential access
  - E.g.,) read next 1000 bytes



- Random access
  - E.g,) Read 10 bytes at the offset 300

- Remember that random access is especially slow in HDD.

File Usage Patterns

- Most files are small
  - .c, .h, .txt, .log, .ico, …
  - Also more frequently accessed
  - If the block size is too big, It wastes space (why?)

- Large files use most of the space
  - .avi, .mp3, .jpg,
  - If the block size is too small, mapping information can be huge (performance and space overhead)

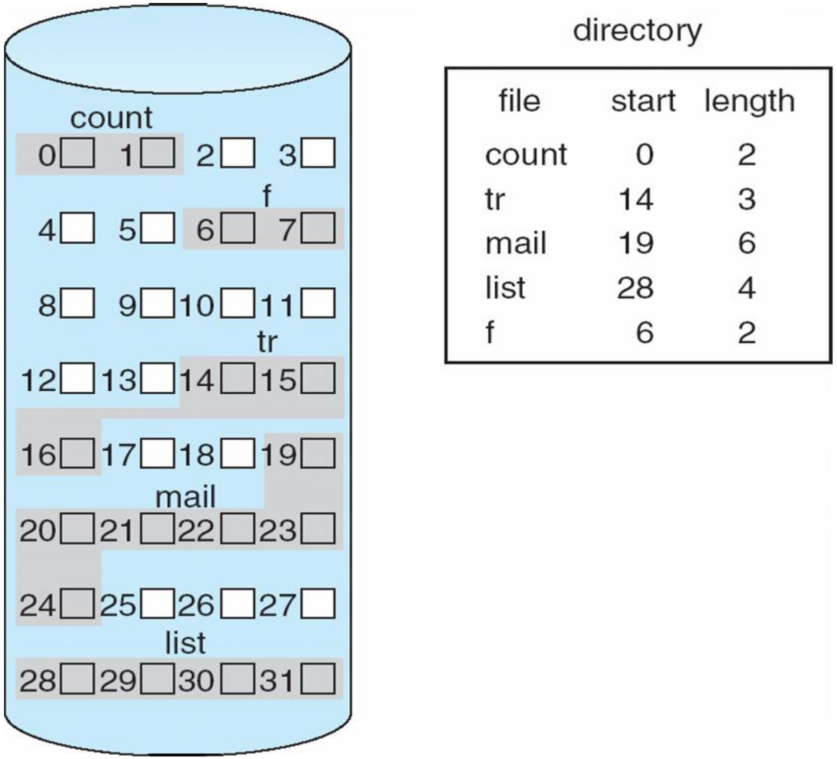
Disk Allocation

- How to map disk blocks to files?
  - Each file may have very different size
  - The size of a file may change over time (grow or shrink)

- Disk allocation methods
  - Continuous allocation
  - Linked allocation
  - Indexed allocation

Continuous Allocation

- Use continuous ranges of blocks
  - Users declare the size of a file in advance
  - File header: first block #, #of blocks
  - Similar to malloc()
- Pros
  - Fast sequential access
  - easy random access
- Cons
  - External fragmentation
  - difficult to increase

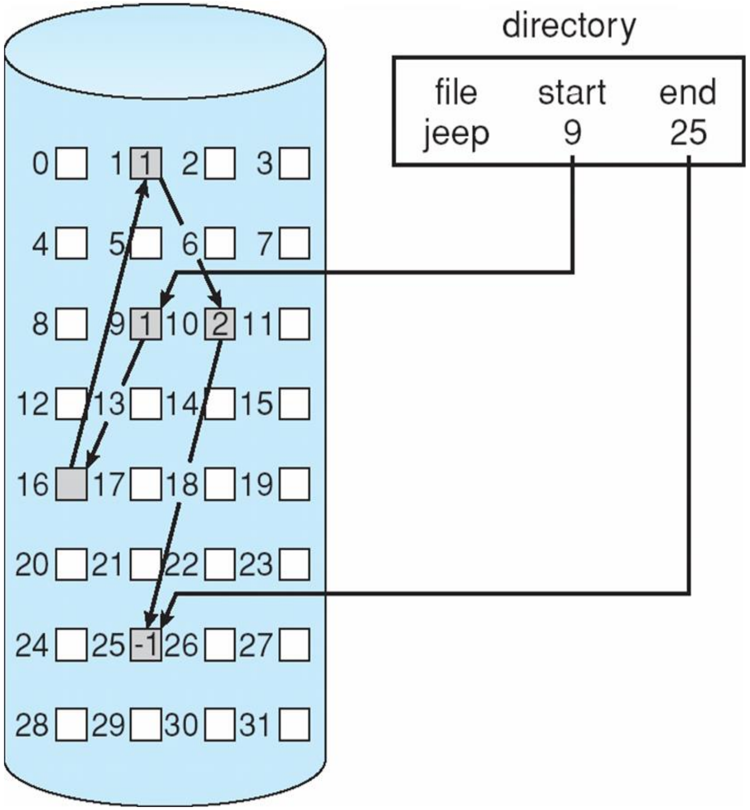


file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked-List Allocation

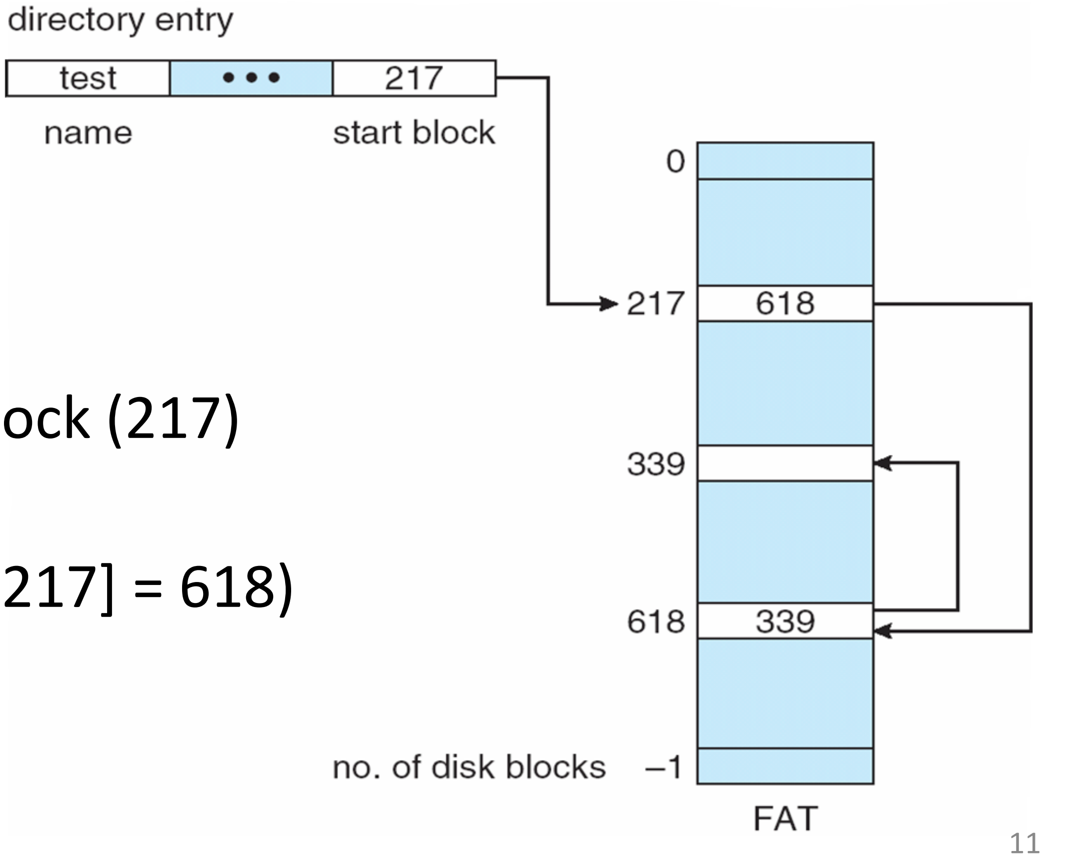
- Each block holds a pointer to the next block in the file

- Pros
  - Can grow easily
- Cons
  - Bad sequential access perf.
  - Unreliable (why?)



File Allocation Table (FAT)

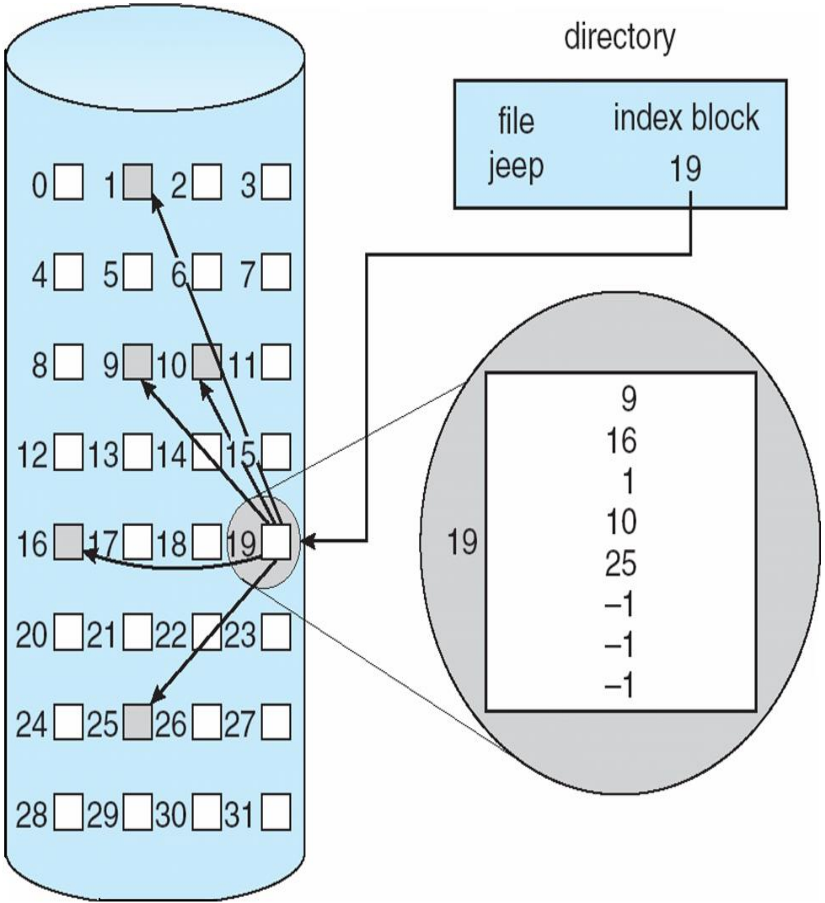
- A variation of linked allocation
  - Links are not stored in data blocks but in a separate table FAT[#of blocks]



  - Directory entry points to the first block (217)
  - FAT entry points to the next block (FAT[217] = 618)

Indexed Allocation

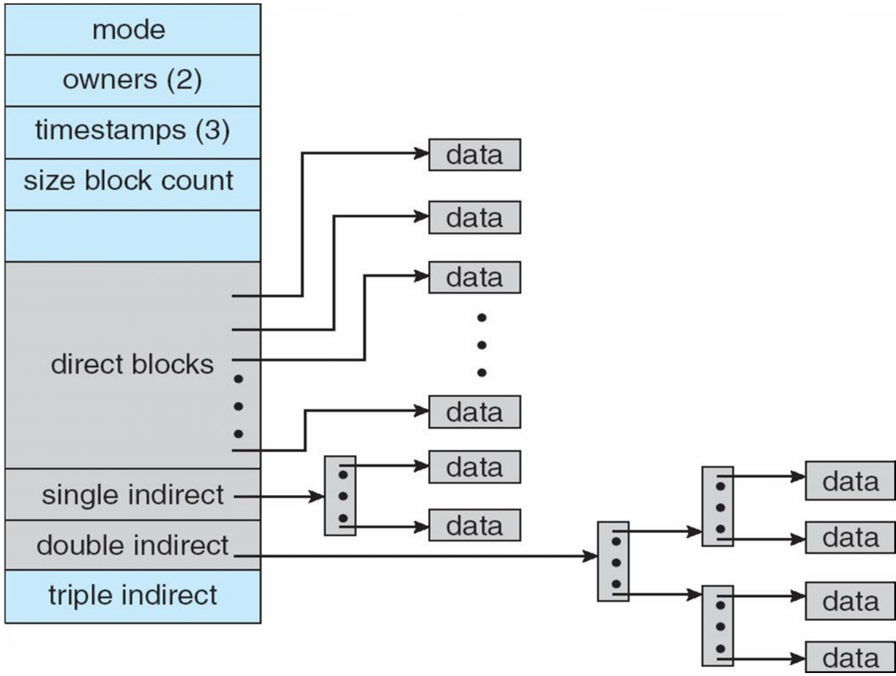
- Use per-file index block which holds block pointers for the file
  - Directory entry points to a index block (block 19)
  - The index block points to all blocks used by the file
- Pros
  - No external fragmentation
  - Fast random access
- Cons
  - Space overhead
  - File size limit (why?)



Multilevel Indexed Allocation

- Direct mapping for small files
- Indirect (2 or 3 level) mapping for large files

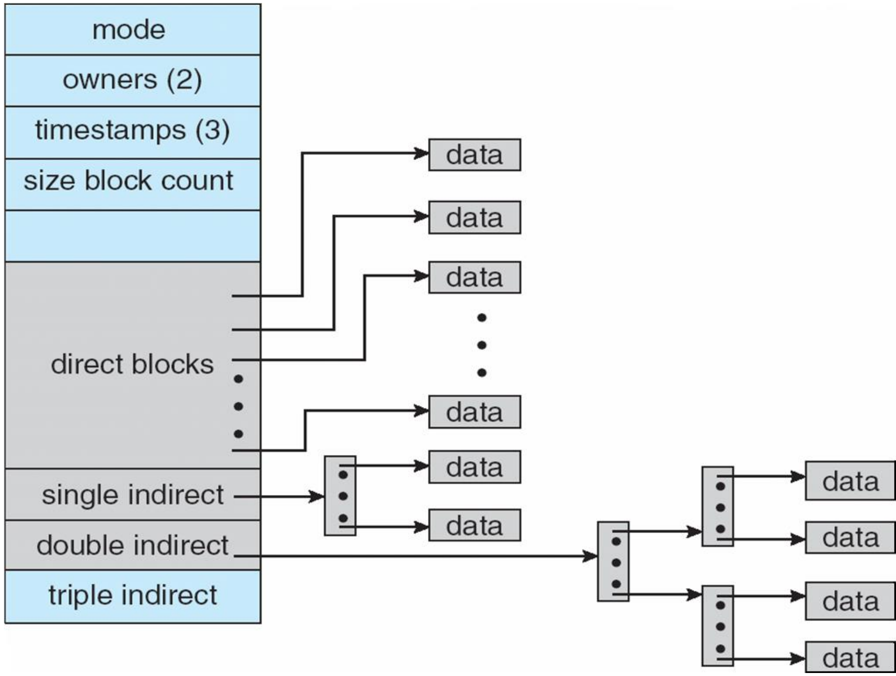
- 10 blocks are directly mapped
- 1 indirect pointer
  - 256 blocks
- 1 double indirect pointer
  - 64K blocks
- 1 triple indirect pointer
  - 16M blocks



Multilevel Indexed Allocation

- Direct mapping for small files
- Indirect (2 or 3 level) mapping for large files

- Pros
  - Easy to expand
  - Small files are fast (why?)
- Cons
  - Large files are costly (why?)
  - Still has size limit (e.g.,16GB)


mode
owners (2)
timestamps (3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data