

W4118: file systems



Instructor: Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

Outline

□ File system concepts

- What is a file?
- What operations can be performed on files?
- What is a directory and how is it organized?

□ File implementation

- How to allocate disk space to files?

What is a file

- User view
 - Named byte array
 - Types defined by user
 - Persistent across reboots and power failures

- OS view
 - Map bytes as collection of blocks on physical storage
 - Stored on nonvolatile storage device
 - Magnetic Disks

Role of file system

- Naming
 - How to "name" files
 - Translate "name" + offset → logical block #
- Reliability
 - Must not lose file data
- Protection
 - Must mediate file access from different users
- Disk management
 - Fair, efficient use of disk space
 - Fast access to files

File metadata

- ❑ **Name** - only information kept in human-readable form
- ❑ **Identifier** - unique tag (number) identifies file within file system (**inode number** in UNIX)
- ❑ **Location** - pointer to file location on device
- ❑ **Size** - current file size
- ❑ **Protection** - controls who can do reading, writing, executing
- ❑ **Time, date, and user identification** - data for protection, security, and usage monitoring
- ❑ How is metadata stored? (**inode** in UNIX)

File operations

- ❑ `int creat(const char* pathname, mode_t mode)`
- ❑ `int unlink(const char* pathname)`
- ❑ `int rename(const char* oldpath, const char* newpath)`
- ❑ `int open(const char* pathname, int flags, mode_t mode)`
- ❑ `int read(int fd, void* buf, size_t count);`
- ❑ `int write(int fd, const void* buf, size_t count)`
- ❑ `int lseek(int fd, offset_t offset, int whence)`
- ❑ `int truncate(const char* pathname, offset_t len)`
- ❑ ...

Open files

- ❑ Problem: **expensive** to resolve name to identifier on each access
- ❑ Solution: open file before access
 - **Name resolution**: search directories for file name and check permission
 - Read relevant file metadata into **open file table** in memory
 - Return index in open file table (**file descriptor**)
 - Application pass index to OS for subsequent access
- ❑ **System-wide open file table** shared across processes
- ❑ **Per-process open file table** stores current pointer position and index to system-wide open file table

Directories

- Organization technique
 - Map file name to location on disk
 - Also stored on disk

- Single-Level directory
 - Single directory for entire disk
 - Each file must have unique name
 - Not very usable

- Two-level directory
 - Directory for each user
 - Still not very usable

Tree-structured directory

- Directory stored on disk just like files
 - Data consists of <name, index> pairs
 - Name can be another directory
 - Designated by special bit in meta-data
 - Reference by separating names with slashes
 - Operations
 - User programs can read (`readdir()`)
 - Only special system calls can write

- Special directories
 - Root (`/`): fixed index for metadata
 - `.` : this directory
 - `..` : parent directory

Acyclic-graph directories

- ❑ Directories can share files
- ❑ Create links from one file
- ❑ Two types of links
 - **Hard link**
 - Multiple directory entries point to same file
 - Store reference count in file metadata
 - Cannot refer to directories; why?
 - **Symbolic link**
 - Special file, designated by bit in meta-data
 - File data is name to another file

Path names

- Absolute path name (full path name)
 - Start at root directory
 - E.g. /home/junfeng/teaching

- Relative path name
 - Full path is lengthy and inflexible
 - Give each process **current working directory**
 - Assume file in current directory

Directories as files

- Directory as special files that store pointers to the contained files
 - File data is interpreted by FS code
- Separate functionality in two levels
 - Lowest: storage management
 - Highest: naming, directory
- Advantage: simplifies design and implementation

Protection

- Type of access
 - Read, write, execute, append, delete, list ...

- Access control list
 - Associate lists of users with access rights for every file
 - Advantage: complete control
 - Disadvantage
 - Tedious to construct list (may not know in advance for all users)
 - Require variable-size information

- Classify users
 - user, group, other
 - Advantage: easier to implement
 - Disadvantage: no fine grained control

Outline

□ File system concepts

- What is a file?
- What operations can be performed on files?
- What is a directory and how is it organized?

□ File implementation

- How to allocate disk space to files?

Typical file access patterns

□ Sequential Access

- Data read or written in order
 - Most common access pattern
 - E.g., copy files, compiler read and write files,
- Can be made very fast (peak transfer rate from disk)

□ Random Access

- Randomly address any block
 - E.g., update records in a database file
- Difficult to make fast (**seek time and rotational delay**)

Disk management

- Need to track where file data is on disk
 - How should we map logical sector # to surface #, track #, and sector #?
 - Order disk sectors to minimize seek time for sequential access

- Need to track where file metadata is on disk

- Need to track free versus allocated areas of disk
 - E.g., block allocation bitmap (Unix)
 - Array of bits, one per block
 - Usually keep entire bitmap in memory

Allocation strategies

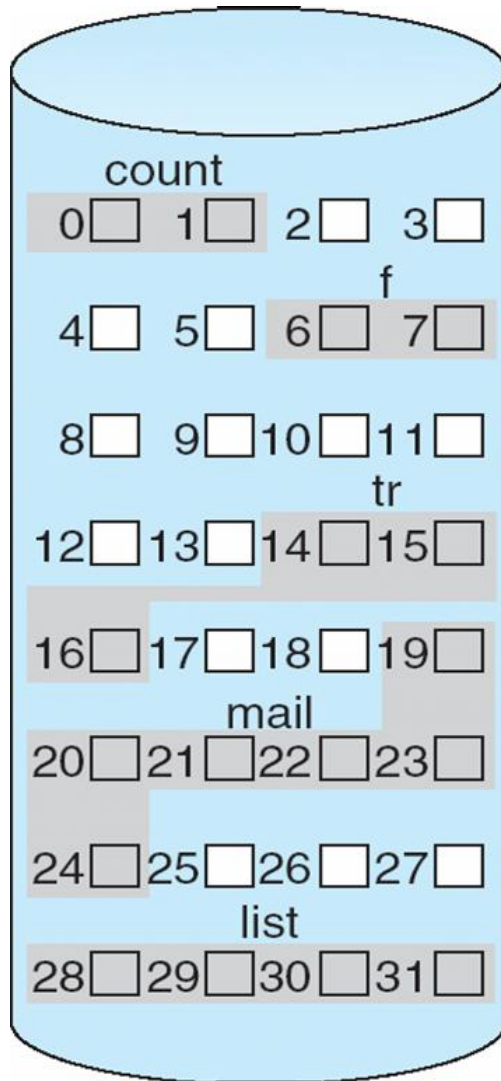
- Various approaches (similar to memory allocation)
 - Contiguous
 - Extent-based
 - Linked
 - FAT tables
 - Indexed
 - Multi-Level Indexed

- **Key metrics**
 - Fragmentation (internal & external)?
 - Grow file over time after initial creation?
 - Fast to find data for sequential and random access?
 - Easy to implement?
 - Storage overhead?

Contiguous allocation

- Allocate files like **continuous memory allocation** (base & limit)
 - User specifies length, file system allocates space all at once
 - Can find disk space by examining bitmap
 - Metadata: contains starting location and size of file

Contiguous allocation example



directory

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

Pros and cons

□ Pros

- **Easy** to implement
- **Low** storage overhead (two variables to specify disk area for file)
- **Fast sequential** access since data stored in continuous blocks
- **Fast** to compute data location for **random** addresses. Just an array index

□ Cons

- **Large external fragmentation**
- **Difficult to grow file**

Extent-based allocation

- Multiple contiguous regions per file (like segmentation)
 - Each region is an **extent**
 - Metadata: contains small array of entries designating extents
 - Each entry: start and size of extent

Pros and cons

□ Pros

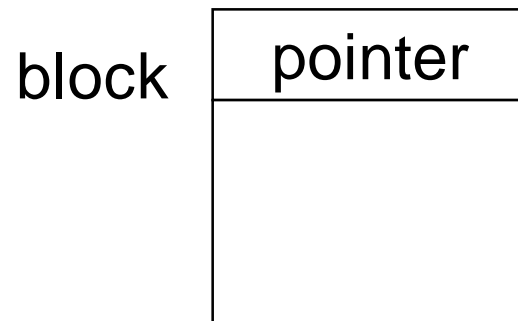
- **Easy** to implement
- **Low** storage overhead (a few entries to specify file blocks)
- File **can grow** overtime (until run out of extents)
- **Fast sequential** access
- **Simple** to calculate **random** addresses

□ Cons

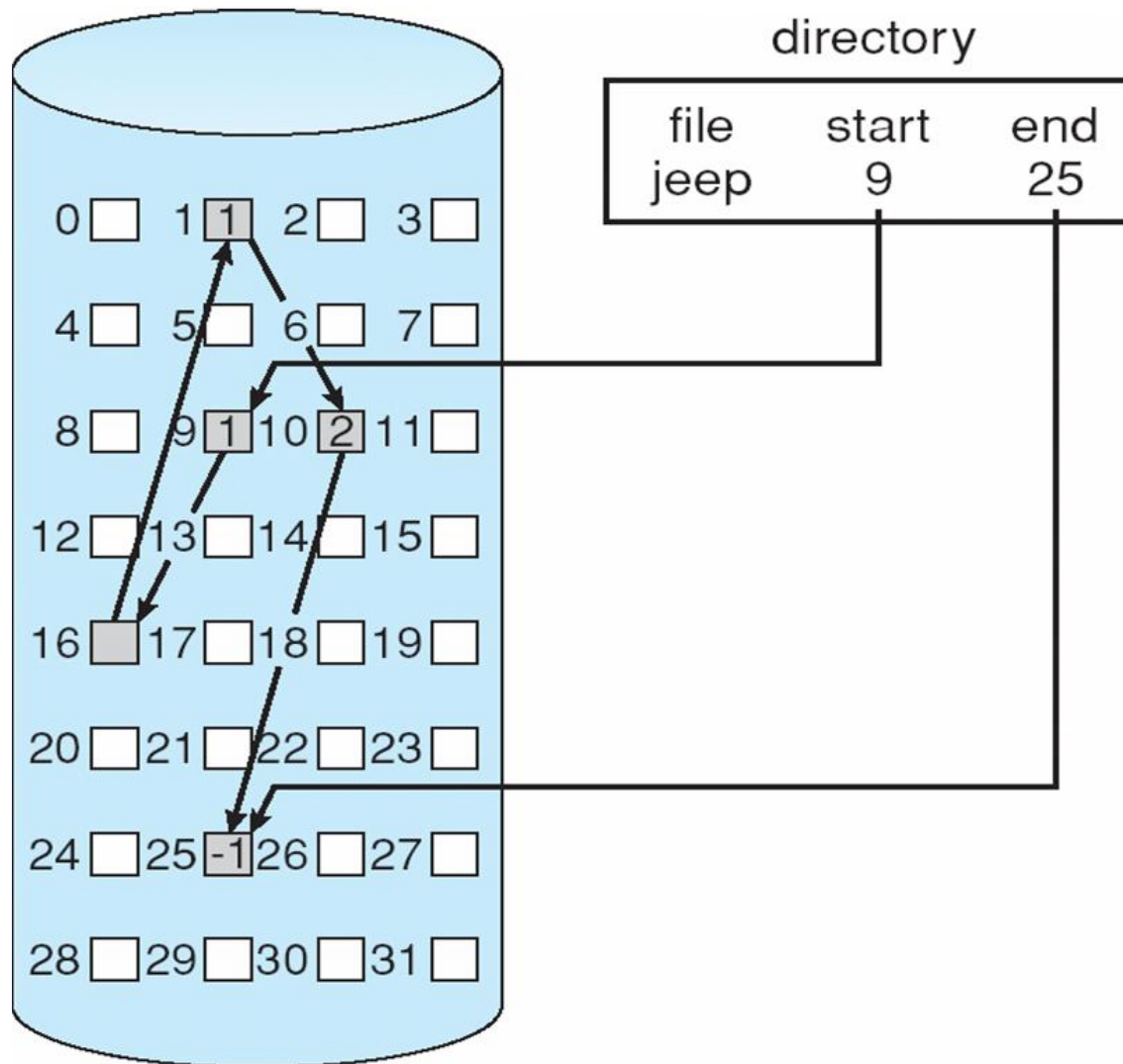
- Help with **external fragmentation**, but still a problem

Linked allocation

- All blocks (fixed-size) of a file on linked list
 - Each block has a pointer to next
 - Metadata: pointer to the first block



Linked allocation example



Pros and cons

□ Pros

- **No** external fragmentation
- Files **can be easily grown** with no limit
- Also easy to implement, though awkward to spare space for disk pointer per block

□ Cons

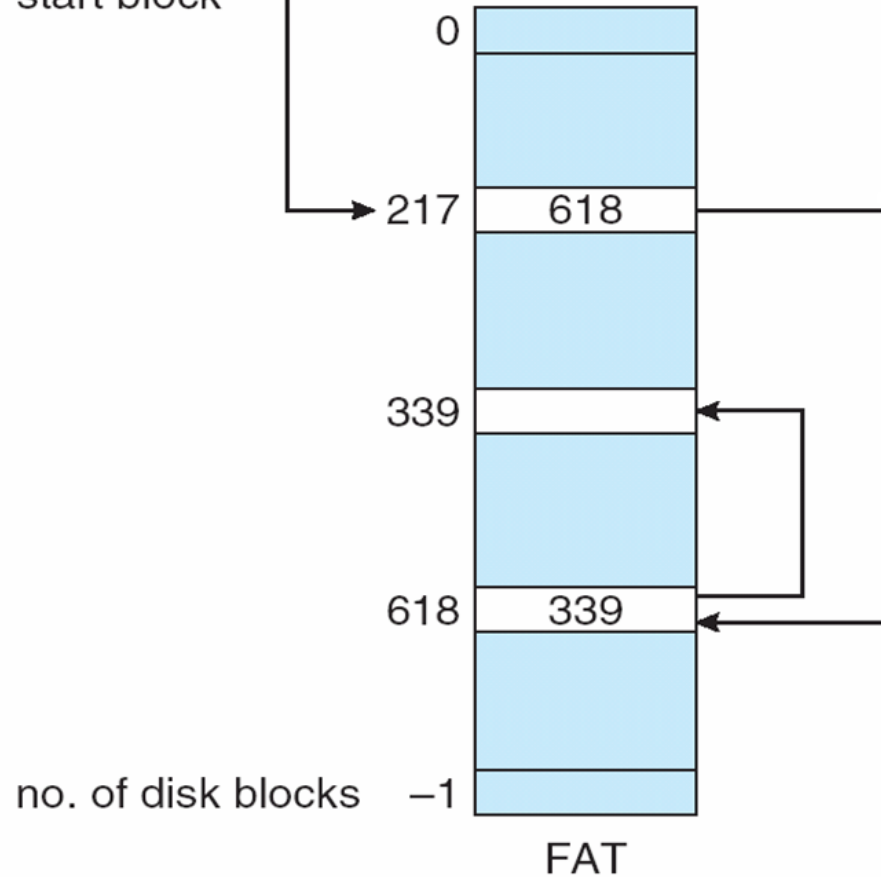
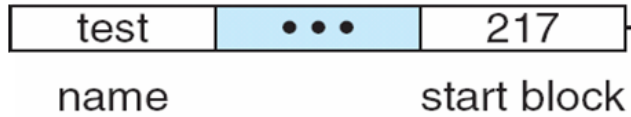
- **Moderate** storage overhead (one pointer per block)
- **Potentially slow** sequential access
- **Difficult** to compute **random** addresses

Variation: FAT table

- Store linked-list pointers outside block in **File-Allocation Table**
 - One entry for each block
 - Linked-list of entries for each file
- Used in MSDOS and Windows operating systems

FAT example

directory entry



Pros and cons

□ Pros

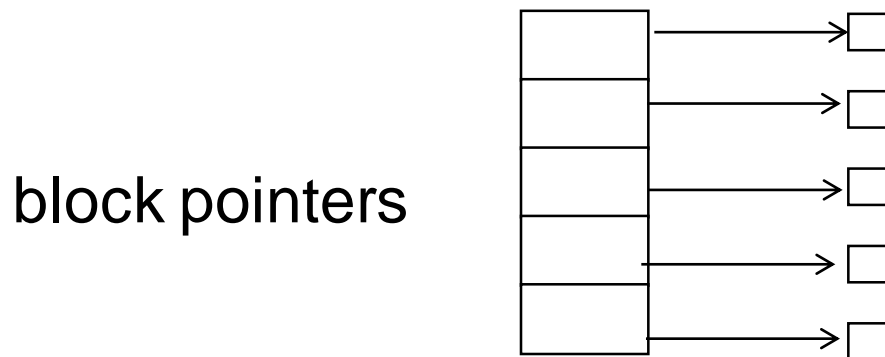
- **Fast random** access. Only search cached FAT

□ Cons

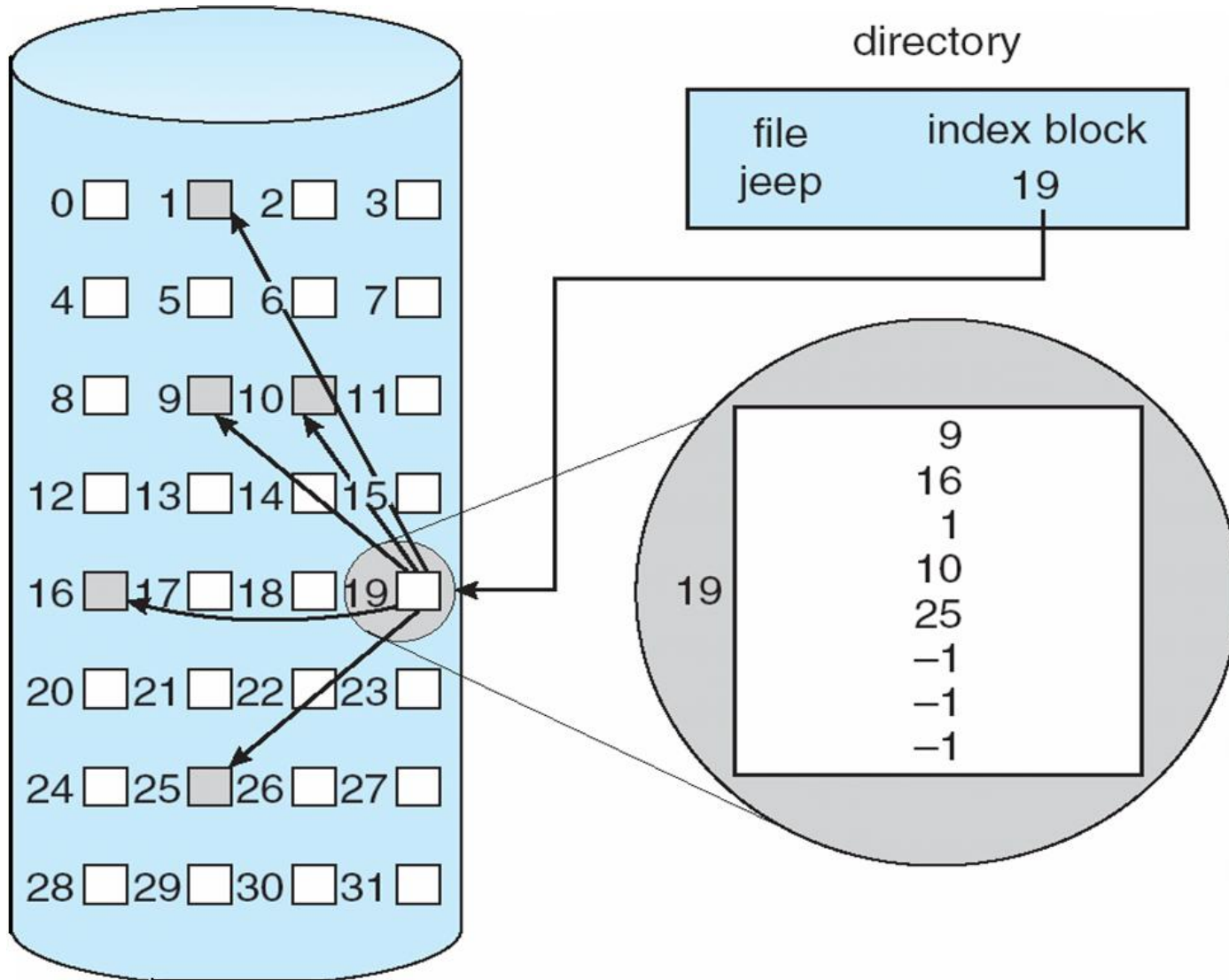
- **Moderate storage** overhead for FAT table
- **Potentially slow** sequential access

Indexed allocation

- File has array of pointers (**index**) to block
 - **Allocate block pointers contiguously in metadata**
 - Must set max length when file created
 - Allocate pointers at creation, allocate blocks on demand
 - Cons:
 - **Maintain multiple lists of block pointers**
 - Last entry points to next block of pointers
 - Cons:



Indexed allocation example



Pros and cons

□ Pros

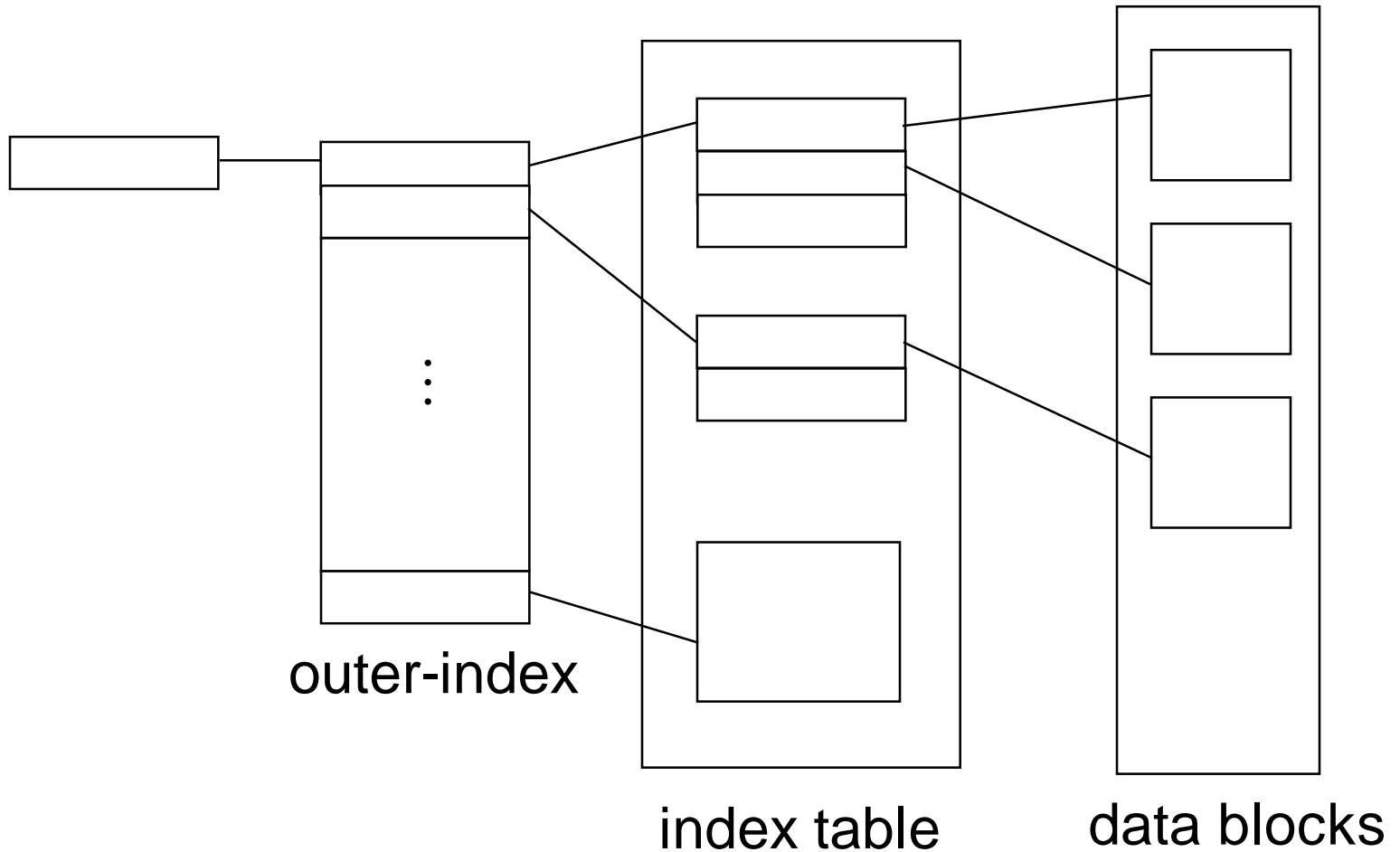
- Easy to implement
- No external fragmentation
- Files can be easily grown with the limit of the array size
- Fast random access. Use index

□ Cons

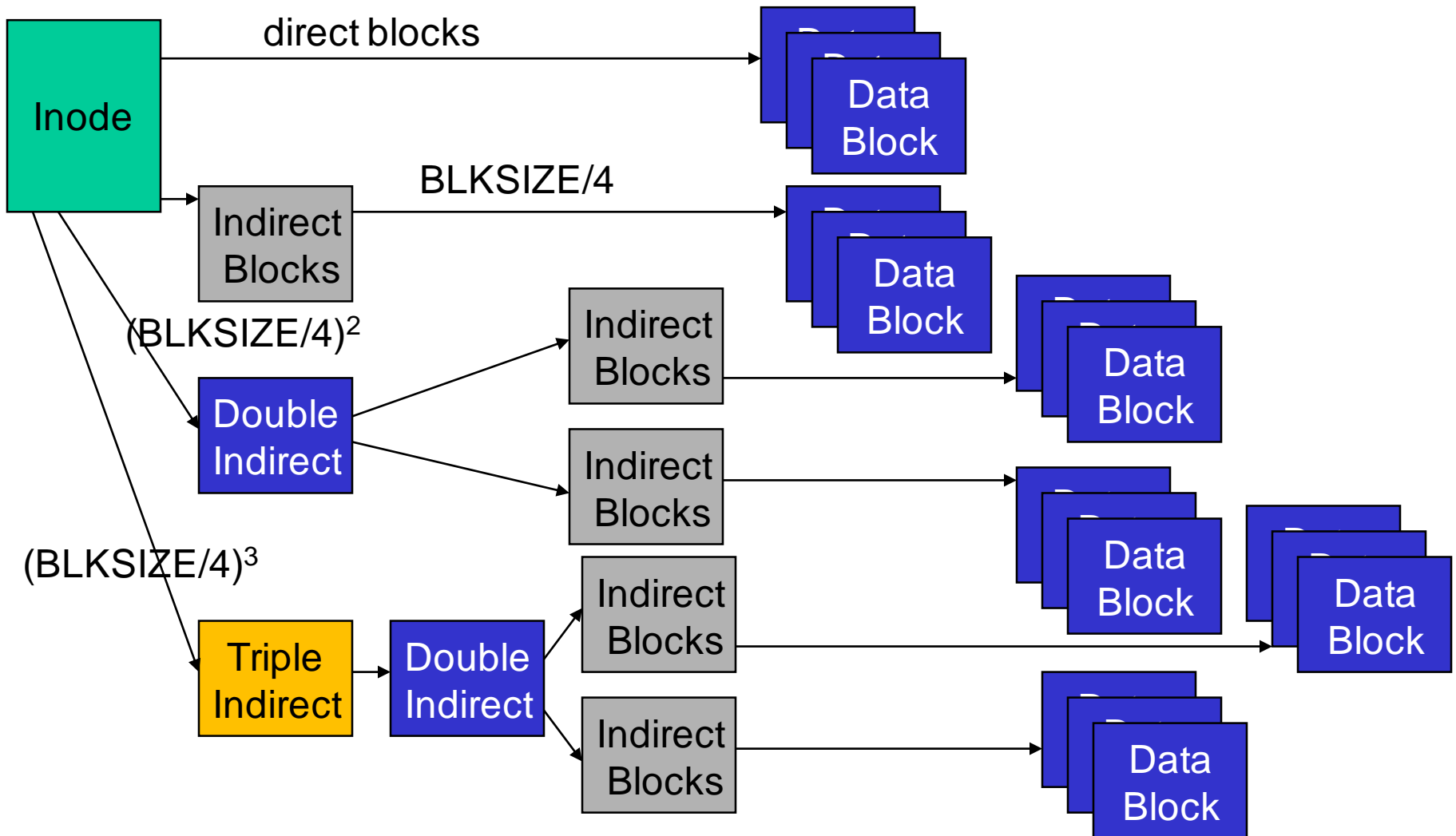
- Large space overhead (index)
- Sequential access may be slow.
 - Must allocate contiguous block for fast access

Multi-level indexed files

- Block index has multiple levels



Multi-level indexed allocation example (xv6, UNIX FFS, and Linux ext2/ext3)



Pros and cons

□ Pros

- No external fragmentation
- Files can be easily grown with much larger limit compared to one-level index
- Fast random access. Use index

□ Cons

- Large space overhead (index)
- Sequential access may be slow.
 - Must allocate contiguous block for fast access
- Implementation can be complex