# CS202 (003): Operating Systems
# File System I

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

# Last Time

# What does file system do?

Provide persistence

Provide a way to “name” a set of bytes on the disk (files)

Provide a way to map from human-friendly-names to “names” (directories)

# Where are file systems implemented?

Disk, over network, in memory, in NVRAM, on tape, with paper…

We are going to focus on the disk and generalize later

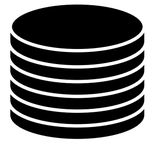
**Important properties of disks:**
(a) information don't go away
(b) we can modify most of the information (except BIOS ROM, …)

**Therefore:**
(a) we are going to put all our important state on the disk
(b) we have to live with what we put on the disk

# What is a file?

a bunch of named bytes on the disk

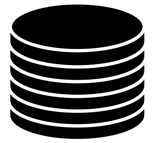
collection of disk blocks



# What FS does…

a bunch of named bytes on the disk

collection of disk blocks



Map name and offset to disk blocks

Operations

create (file), delete (file), read, write

Goal: operations have as few disk access as possible and minimal space overhead

Why care about space overhead when disks are huge?

while disk space might be plentiful, efficient space usage is crucial for performance because it affects both cache utilization and I/O efficiency.

# Address translations

**virtual address -> page table -> physical address**

**offset -> inode -> disk block address**

The inode contains the mapping between file offsets and disk block addresses

**file name -> directory -> file # (inode number in Unix)**

Directories provide the mapping from human-readable names to inode numbers

# Implementing files

Goal: operations have as few disk access as possible and minimal space overhead

*for now we're going to assume that the file's metadata is known to the system*

**access pattern we could support**

## Sequential

File data processed in sequential order

*By far the most common mode*
*Example: editor writes out new file, compiler reads in file, etc*

## Random

Address any block in file directly without passing through the rest of the blocks

*Examples: large data sets, demand paging, databases*

## Keyed

Search for block with particular values

*Examples: associative database, index*
*This thing is everywhere in the field of databases, search engine, but not usually provided by a FS in a OS*

**Some observations**

- All blocks in file tend to be used together, sequentially
- All files in directory tend to be used together
- All **names** in directory tend to be used together
- Most files are small
- Much of the disk is allocated to large files
- Many of the I/O operations are made to large files
- Want good sequential and good random access

# Candidate Designs

## Contiguous *"extend based"*

User must declare the file size upfront before creation
The entire space for the file is allocated at once
File metadata tracks: starting location, file size

`[<free> a1 a2 a3 <5 free> b1 b2 <free>]`

*what if a file c needs 7 sectors?!*

**Advantages:** simple implementation, fast file access (both sequential and random)

**Disadvantages:** fragmentation

## Linked Files

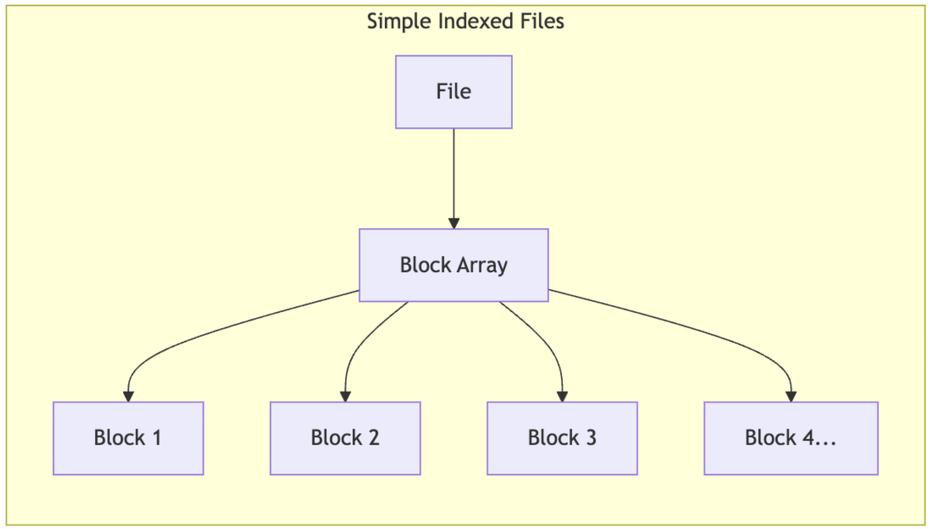

Keep a linked list of free blocks
Each block holds pointer to the next one
Metadata: pointer to file's first block

**Advantages:** no more fragmentation, easy sequential access

**Disadvantages:** bad random access, pointers take up room in blocks (mess up data alignment)

# Candidate Designs - Indexed Files

## Basic Indexed Files

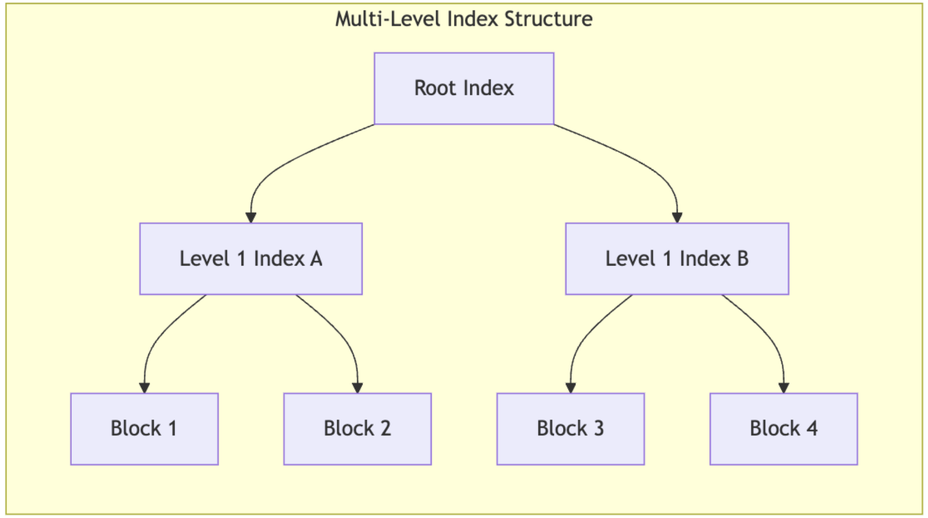



Each file has an array containing pointers to all its data blocks
The array is allocated when the file is created
Blocks are allocated dynamically using a free list system

**Advantages:** sequential and random access are both easy

**Disadvantages:** need to store the array somewhere

## Multi-level Indexed Files

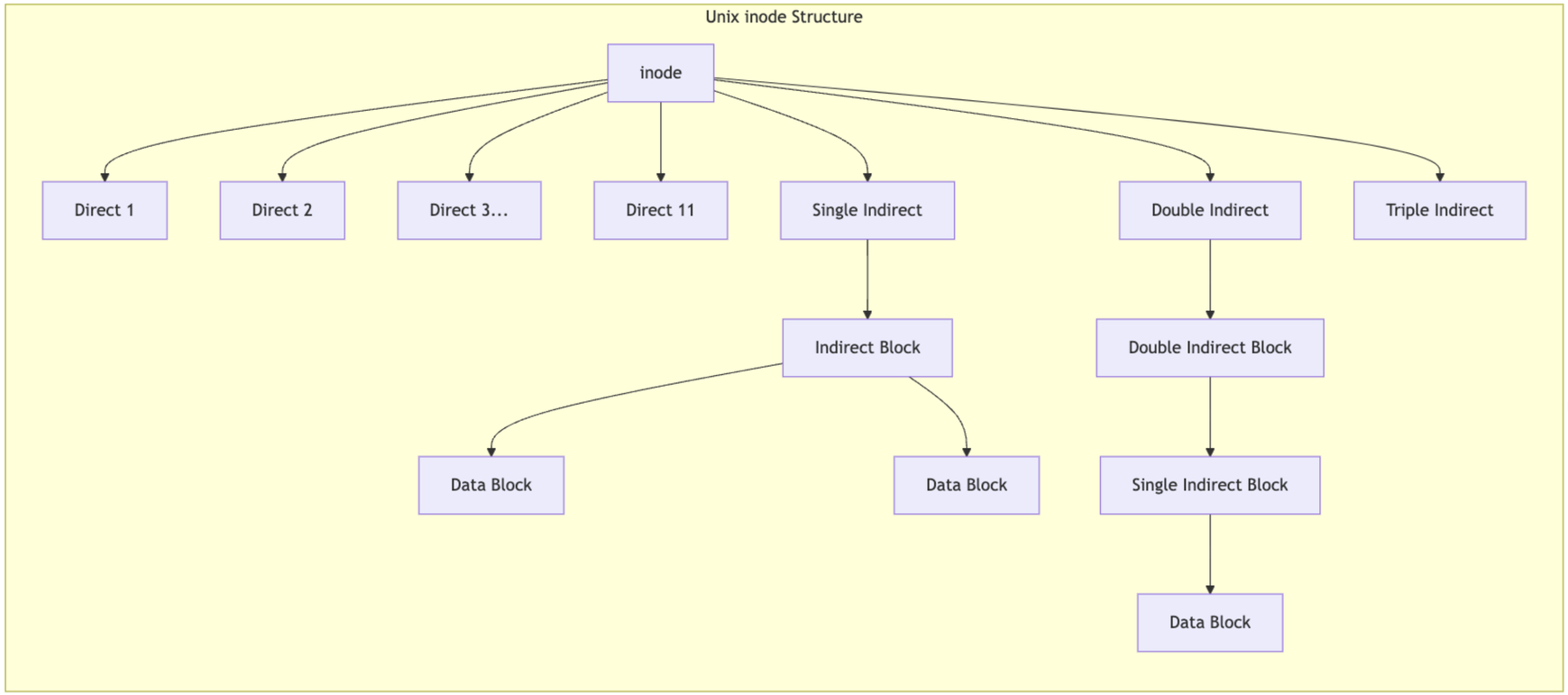



Address the limitation of basic indexed files

**Advantages:** more efficient space utilization, flexible allocation

**Disadvantages:** multiple disk accesses required to reach data blocks, performance penalty for each level traversed

# Candidate Designs - Unix Inode Structure




inode contains:
permissions
times for file access, file modification, and inode-change
link count (# directories containing file)
…… (ptr to data blocks)

*"Why is thistle intentionally imbalanced?"*

optimize for short files. each level of this tree requires a disk seek

**Advantages:** simple, easy to build; fast access to small files; maximum file length can be enormous

**Disadvantages:** worst case # of accesses pretty bad, worst case overhead pretty bad, because you allocate blocks by taking them off unordered freelist, metadata and data get strewn across disk

# Some notes about inode

Fixed-size array storage

Fixed array size during initialization

Multiple inodes per disk block

Lives in known location, originally at one side of disk,
now lives in pieces across disk (helps keep metadata close to data)

The index of an inode in the inode array is called an
**i-number** (OS refers to files by i-numbers)

When a file is opened, the inode brought in memory (written
back to disk when modified and file closed or time elapses)

HW 10 is released today!

HW 9 is due tomorrow!