Sistemas Operativos: File Systems

Disk Data Structures

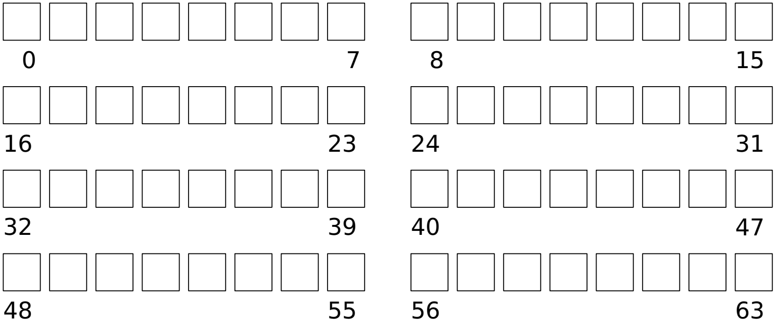
Pedro F. Souto (`pfs@fe.up.pt`)

May 29, 2020

File System Implementation

Given An array of disk blocks

Challenge Store the contents of the files and directories of a file system



File System Implementation: Goals and Constraints

Goals

- **Performance** Disks are much slower than CPU or evem DRAM
- **Capacity Utilization** 1 TB capacity appeared around 2010
- **Reliability** Disks are relatively fragile. Users expect data on disk to persist

Constraints

- **Technology** HDD vs. SSD
- **Usage Pattern**
  - Most files have only a few KB
  - Very large files take up a significant amount of a disk capacity
  - A significant number of accesses is to very large files
  - Some files are accessed sequentially whereas others are acessed randomly

Allocation Strategies

Different alternatives

- Contiguous
- Extent-based
- Linked
- File-allocation Tables
- Indexed
- Multi-level Indexed

Issues

- Amount of fragmentation (internal and external)
  - Free space that cannot be used
- Ability to grow file over time
- Performance of sequential access
- Performance of random access
- Meta-data space overhead
  - Meta-data must be stored persistently

Contiguous Allocation

Idea Allocate each file to contiguous sectors on disk

Meta-data First block and file size

Allocation Need to find sufficient free space

- Must predict future size of file

Example IBM OS/360 (mid 60s)

			A	A	A		B	B	B	B	C	C	C		

Evaluation

Fragmentation	- Horrible: needs periodic compaction
Ability to grow over time	- May require moving
Sequential access (seek time)	+ Excellent performance
Random access (speed to caclulate)	+ Simple
Metadata overnead	+ Little overhead

Fixed Number of Extents

Idea Allocate multiple contiguous regions (**extents**) per file

- **Meta-data** Small array (<10) for each file
  - Each entry: first block and size
- **Allocation** Need to find sufficient free space for extent

			A	A	A		B	B	B	B	C	C	C		

D	D	D	A	A	A	D	B	B	B	B	C	C	C	B	B

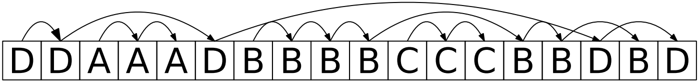
Evaluation

Fragmentation	- Less fragmentation than contiguous
Ability to grow over time	- Can grow (until run out of extents)
Sequential access (seek time)	+ Still good performance (generally)
Random access (speed to caclulate)	+ Still simple
Metadata overnead	+ Still small little overhead

Linked Allocation

Idea Link possibly scattered disk blocks with file contents

- **Meta-data** Location of first block. In addition:
  - Each block contains pointer to the next block
- **Example** Alto (first PC, mid-70s)



Evaluation

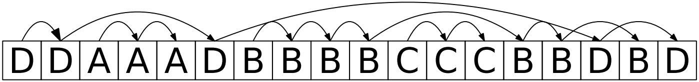
Fragmentation	+ No external frag.; internal?
Ability to grow over time	+ Can grow easily
Sequential access (seek time)	+/- Depends on data layout
Random access (speed to caclulate)	- Horrible
Metadata overnead	- One pointer per block

Trade-off Block size (does not need to equal sector size)

File-Allocation Table (FAT)

Idea Keep linked-list information for all files in on-disk table (FAT)

- **Meta-data** Location of first block. In addition:
  - FAT table itself (1 entry per block)
- **Example** DOS (but from the late 70s)



Show Draw FAT

Evaluation Comparison with Linked Allocation

- **Advantage** Easier and faster calculation for random access
- **Disadvantage** One extra read (FAT) for each data read
- **Optimization** Cache FAT in main memory
  - **Advantage** Improves both advantage and disadvantage
  - **Issue** Large file systems. Cache FAT partially?

Indexed Allocation

Idea Use fixed-length array of entries pointing to blocks **per file**

- **Meta-data** Fixed-sized array of block pointers
  - Allocate array at file creation file

D	D	A	A	A	D	B	B	B	B	C	C	C	B	B	D	B	D

Evaluation

Fragmentation	+ No external frag.; internal?
Ability to grow over time	+/- Can grow easily up to max file size
Sequential access (seek time)	+/- Depends on data layout
Random access (speed to caclulate)	+ Easy
Metadata overnead	- Large overhead for meta-data Wastes space for unused pointers

Trade-off Block size (does not need to equal sector size)

Multi-Level Indexing

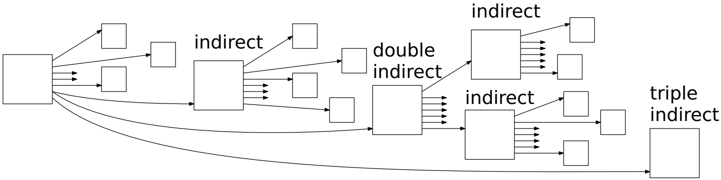
Idea Similar to multi-level page tables

- Dynamically allocate hierarchy of pointers to blocks

Meta-data Small number of pointers allocated statically

- Additional pointers to blocks of pointers

Example Unix FFS-based file systems (mid-80s), ext2, ext3



Evaluation Comparison with indexed allocation

Advantage Does not waste space for unused pointers

- Still fast access for small files

Disadvantage Extra disk reads to access indirect blocks

- Keep indirect blocks cached in main memory

Variable Number of Extents

Idea Dynamically allocate extents

Meta-data Use a multi-level tree structure

- Each leaf onde: first block and extent length

Example NTFS (mid 90s)

Evaluation

Fragmentation	+ Both reasonable
Ability to grow over time	+ Can grow easily up to max file size
Sequential access (seek time)	+ Still good performance
Random access (speed to caclulate)	+/- Depends on the size
Metadata overnead	Relatively small overhead

Multi-Level Indexed Implementation

On-disk Data Structures

Data block

Inode table

Indirect block

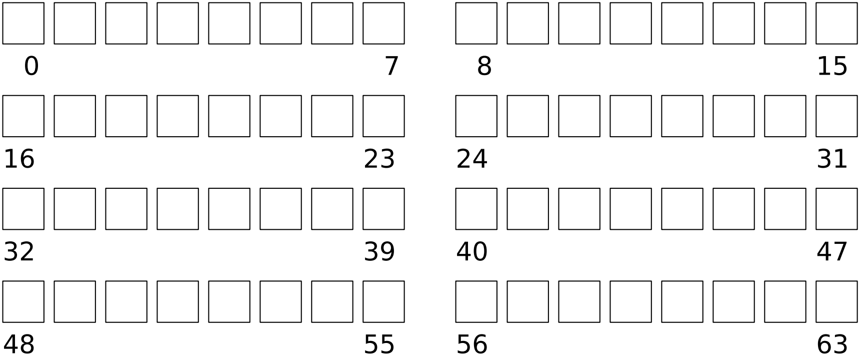
Directories

Data bitmap

Inode bitmap

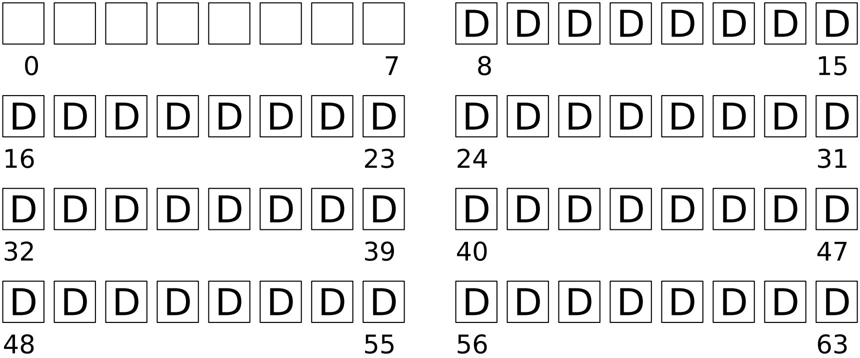
Superblock

FS Structures: Empty disk



- Assume each block is 4 KB

FS Structures: Data Blocks



IMP. Actual layout may be different (see next lecture)

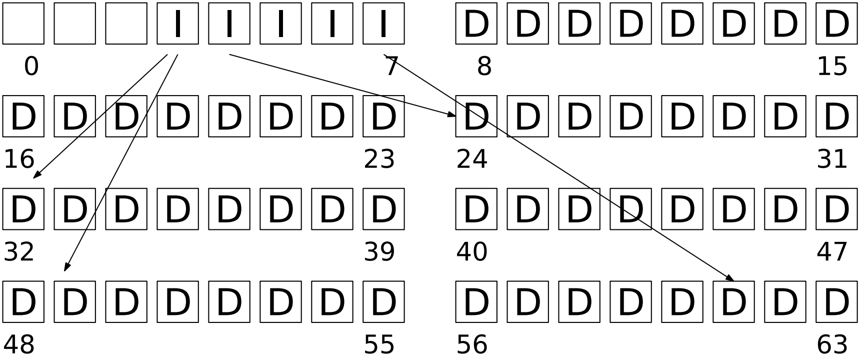
FS Structures: Inode

Inode Likely "index-node"

- Data structure with file metadata kept on disk

```
type (file or dir)
uid (owner)
rwx (permissions)
size (in bytes)
num blocks
time (access)
ctime (create)
links_counts (#paths)
addrs[N ] (N data blocks)
```

FS Structures: Inode Blocks



IMP. Actual layout is different (see next lecture)

FS Structures: Inode Block

- Inode size: 256 bytes (may be 128 bytes)
  - 4KiB disk block size
  - 16 inodes per block

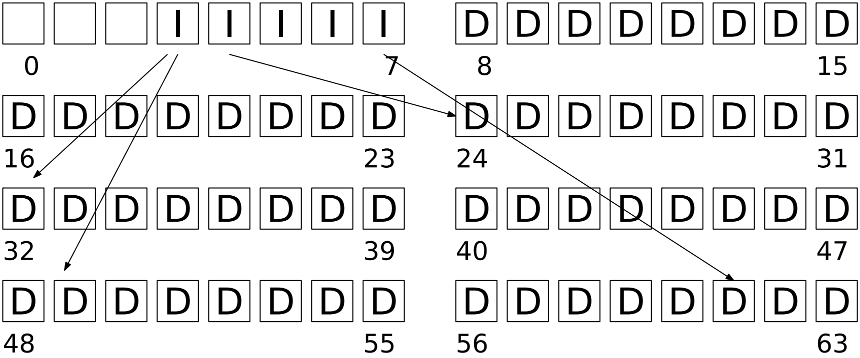
Inode 16
Inode 17
Inode 18
Inode 19
Inode 20
Inode 21
Inode 22
Inode 23
Inode 24
Inode 25
Inode 26
Inode 27
Inode 28
Inode 29
Inode 30
Inode 31

Question How to find an inode on disk, given its number?

FS Structures: Inode Block Location (1/2)

Assumption 16 inodes/block

Question What is the location for inode with number 0?



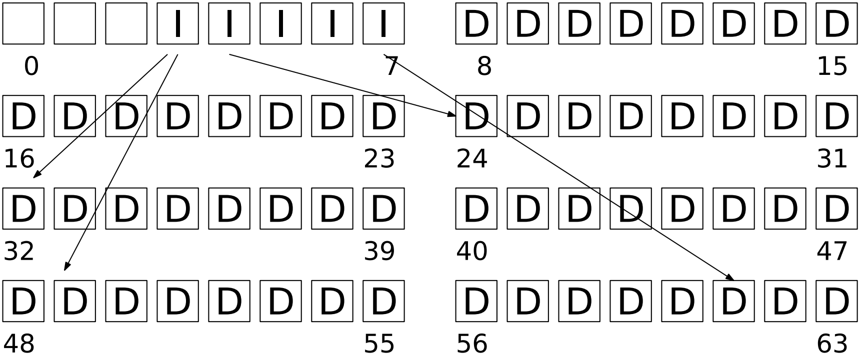
Block first inode + 0/16 = 3 + 0 = 3

Offset within block 0%16 x 256 = 0

FS Structures: Inode Block Location (2/2)

Assumption 16 inodes/block

Question What is location for inode with number 47?

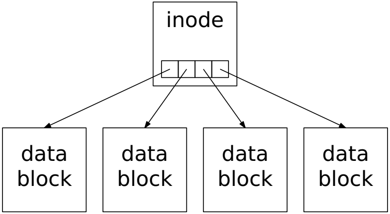


Block first inode + 47/16 = 5 + 0 = 5

Offset within block 47%16 x 256 = 15x256 = 0xF00

FS Structures: Single Level Pointer Table

Assumption Single level inode, i.e. only pointers to data blocks



Question What is maximum file size?

Assumptions

- **Inode size** 256 B
- **Block size** 4KiB (all can be used for pointers)
- **Block address** 4 B

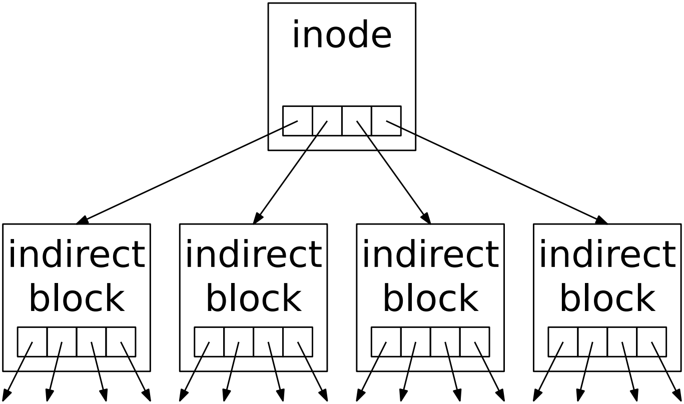
Answer

256 / 4 = 64 pointers per block

64 x 4 KiB = 64 KiB

Question How to support larger files?

FS Structures: Balanced Tree



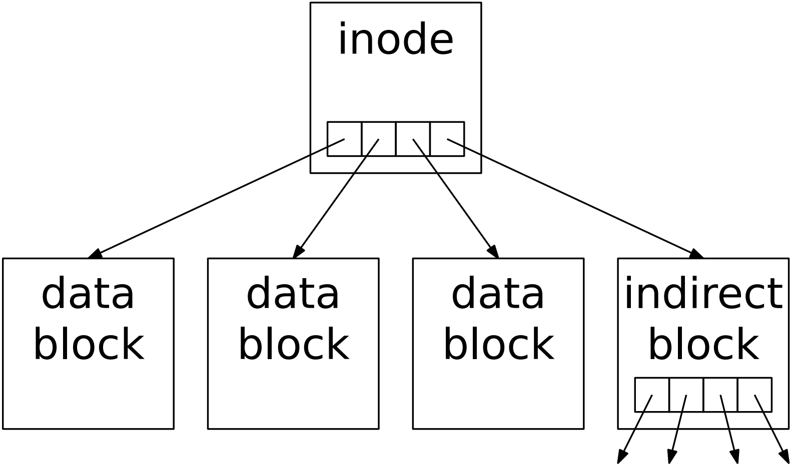
Note Indirect blocks are stored in data blocks

- Indirect blocks contain only pointers to files

Question How to optimize for small files?

FS Structures: Unbalanced Tree (FFS)

Answer Use an unbalanced tree.



Note FFS uses 2-level indirect blocks (i.e. an indirect block where each entry points to an indirect block) and 3-evel indirect blocks

Directories Implementation

Observation Depends on the file system

Common design:

- Use an inode per directory
  - A directory is a special type of file.
- Store directory entries in data blocks
- Large directories use multiple data blocks
- Use bit in inode to distinguish directories from files

Data structures for storing entries e.g.:

- Lists

valid	name	inode
1	.	124
1	..	35
1	foo	80
1	bar	23

- B-trees

Allocation

Issue How do we find free data blocks or free inodes?

Alternatives Among others:

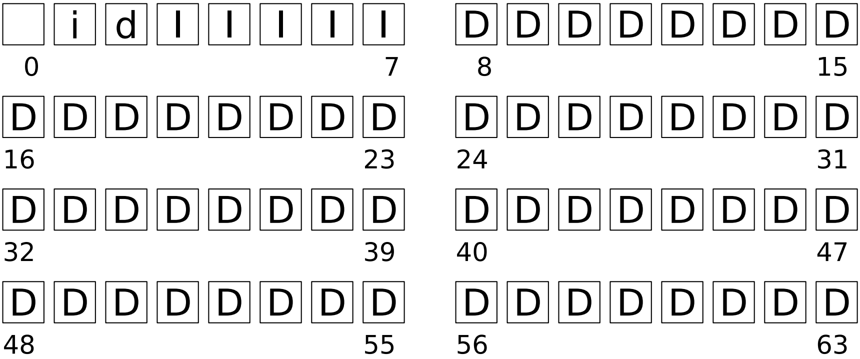
- Free list
- Bitmaps
- Tradeoffs in next lecture...

Bitmaps? (1/2)

			I	I	I	I	I	D	D	D	D	D	D	D	D
0							7	8							15
D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
16							23	24							31
D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
32							39	40							47
D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
48							55	56							63

Question Where to store them?

Bitmaps? (2/2)



Issue Possibility for inconsistency

- Bitmaps may not be in agreement with block usage

Superblock

Issue Need to know basic FS configuration

- block size
- # of inodes
- # of data blocks

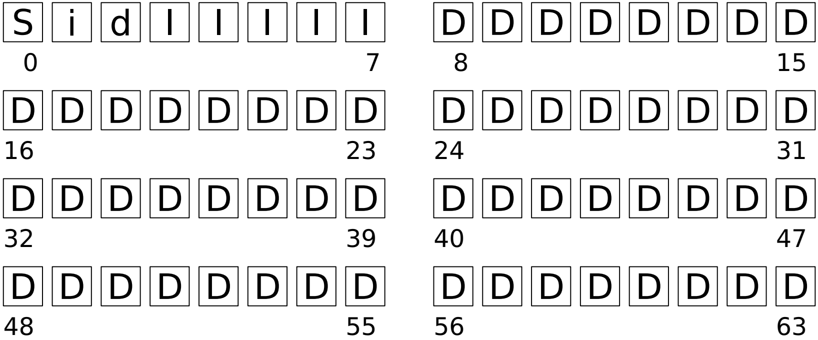
Solution ...

Superblock

Issue Need to know basic FS configuration

- block size
- # of inodes
- # of data blocks

Solution ... store this in superblock



On-disk Data Structures

- Superblock
- Inode bitmap
- Data(block) bitmap
- Inodes
- Data blocks also used for:
  - Directories
  - Indirect blocks i.e. blocks with pointers to other blocks

Operations: open `/foo/bar`

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	comment
		read					foo?
					read		foo?
			read				bar?
						read	bar?
				read			done

Operations: read `/foo/bar`

Assumption `/foo/bar` opened

bitmaps	root inode	foo inode	bar inode	root data	foo data	bar data	comment
			read				cache?
						read	data
			write				atime

Operations: write `/foo/bar`

Assumption `/foo/bar` opened

data bitmap	root inode	foo inode	bar inode	root data	foo data	bar data	com-ment
			read				cache?
read							if ...
write							if ...
						write	data
			write				

bar inode update:

- data pointers
- file size
- file timestamps

Operations: close `/foo/bar`

Assumption all data and metadata written directly to disk on other calls

inode bitmap	data bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

Thus nothing else to write

Operations: create `/foo/bar`

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	com-ment
		read					foo?
					read		foo?
			read				bar?
						read	bar?
	read						find
	write						set
						write	add
				read			??
				write			initial.
			write				atime

Question Why read bar inode before writing it?

How to reduce file system I/O costs?

Issue Simple file system system calls require an unsuspecting large number of disk accesses

`open()` requires at least two reads for each level in a pathname

1. For reading the inode of the directory.
2. For reading that directory's data block(s)

`create()` similar to open but it also requires:

- Read/write inode bitmap, to allocate inode for newly created file
- Writing to the parent directory's data block and inode

`read()` requires:

- Reading the file's inode (to locate the data block)
- Reading the file's data block
- Writing to the file's inode to update last access time

`write()` similar to write, but may also require

- Reading and writing the data bitmap, to allocate a new data block (if needed)

Solution: Use caching

Idea store frequently accessed disk blocks in main memory.

- Use LRU to manage the cache

Fixed-size caches

- Upon booting the kernel reserves a fixed number of pages, e.g. 10%, for storing disk blocks – **static partitioning**
- May waste main-memory space

Unified page cache

- Shared between the file system and virtual memory
- Allows **dynamic partitioning**
  - I.e. the amount of pages used by the file system may vary with time depending on the load

Performance improvements

Read buffering

- Opening a second file in the same directory as a previously opened file, may be done without any disk I/O
- A sufficiently large cache could reduce disk reads almost to zero

Write buffering may also reduce disk writes or reduce seek time.
By delaying writes, typically between 5 and 30 s, the OS can:

Batch multiple writes

Better schedule disk operations

Avoid disk writes altogether e.g. if a file is created and soon after deleted.

Issue If the system crashes data that was not written to disk will be lost

Trade-off performance vs. reliability

`fsync()` flushes to disk a file's data in the buffer cache