File Systems: Free Space & Naming

- Learning Objective
  - Evaluate trade-offs between different free space representations and management techniques.
  - Explain some alternative ways to implement a directory (naming) structure in a file system.
  - Critique different designs, evaluating their trade-offs.
- Topics:
  - Finish up evaluation of free space management strategies.
  - Naming exercise.
  - In-depth study of directory implementation

Exercise 2: Free Space Management

- Assume you allocate in fixed size blocks:
  - How do you keep track of free space?
  - How do you select which blocks to allocate to a particular file?

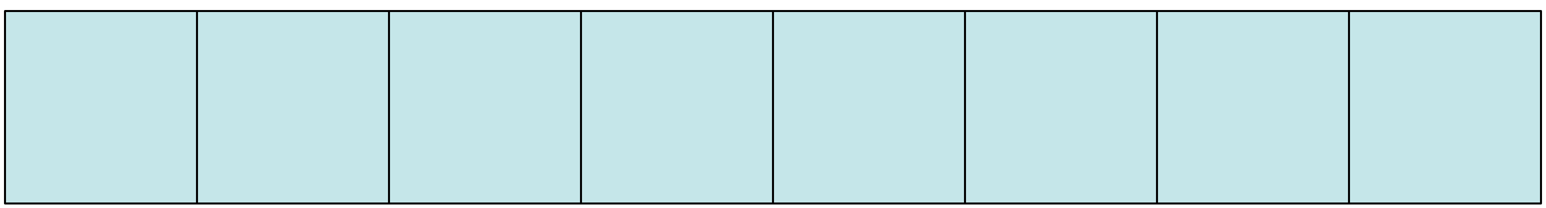
- Assume that you allocate variable size extents:
  - How do you select the extent size?
  - How do you manage free space?
  - Where do you allocate extents?

Free Space Management (1)

- There is often a tradeoff between the amount of (allocation) meta data you keep and the quality of allocation.
- Fixed size blocks:
  - Free list: link all the free pages together in a list (placing the pointer on the actual page).
    - Metadata: One pointer (excellent).
    - Ease of allocation: Pull first block off the list (excellent).
    - Ability to produce good (e.g., contiguous) allocations? Poor.
  - Bitmaps
    - Metadata: One bit per block (good)
    - Ease of allocation: Find a free bit (good)
    - Ability to produce good allocations? (good)
- How do these apply to a small number of block sizes?

Buddy Allocation

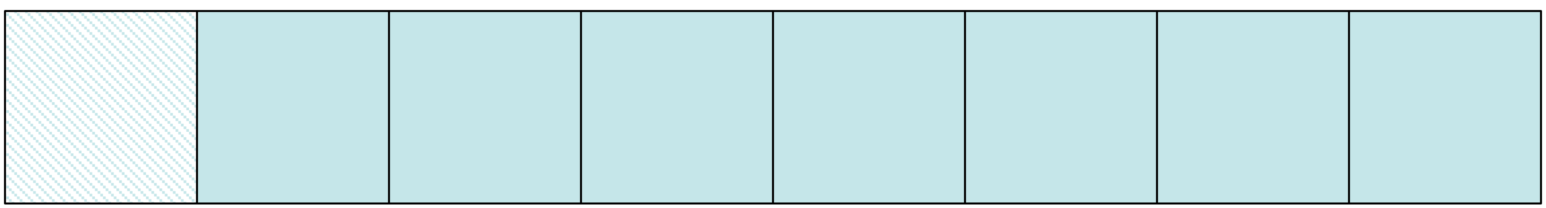
- One way to support multiple block sizes is to make all the sizes be a power-of-two multiple of a basic block size.
- Rather than assign disk blocks to different sized file system blocks haphazardly, create blocks of size 2^N by splitting a block of size 2^{N+1}



1. Disk is collection of maximum size blocks

Buddy Allocation

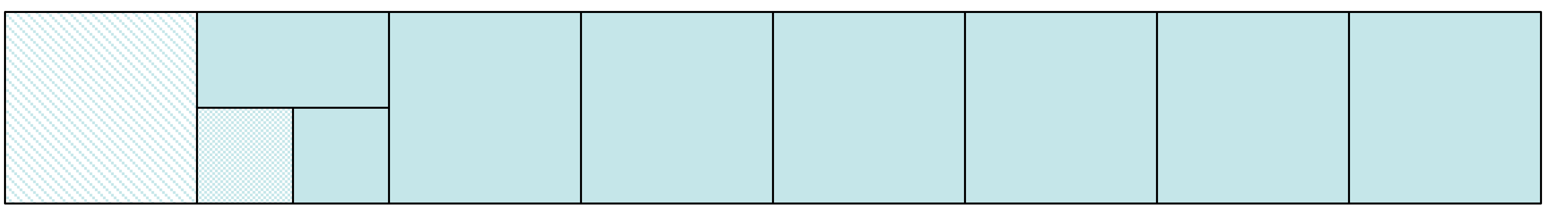
- One way to support multiple block sizes is to make all the sizes be a power-of-two multiple of a basic block size.
- Rather than assign disk blocks to different sized file system blocks haphazardly, create blocks of size 2^N by splitting a block of size 2^{N+1}



2. Allocate a large block.

Buddy Allocation

- One way to support multiple block sizes is to make all the sizes be a power-of-two multiple of a basic block size.
- Rather than assign disk blocks to different sized file system blocks haphazardly, create blocks of size 2^N by splitting a block of size 2^{N+1}



3. Allocate minimum-sized block.

Free Space Management (2)

- Extents
  - On-disk malloc (free list approach)
    - Keep free extents in lists, tagged with size
    - Or, like a slab allocator, have multiple lists with different-sized blocks
    - Metadata: one or a few pointers (excellent)
    - Ease of allocation: pretty good
    - Problems? Fragmentation (both internal and external)
  - Bitmap based: probably need to track in some primitive unit size
    - Metadata: one bit per primitive unit (good)
    - Ease of allocation: not great – need to search for contiguous chunks.

Exercise 3: Naming

- We will assume that you need to implement a hierarchical name space (i.e., directories & files).
  - How will you represent a directory?
  - How will you find the root directory ("/")?
  - How will you support traversing up a directory tree (cd ..)?
  - Be as specific as you can.

Naïve Naming

- One directory for the entire disk (file system).
- Small maximum name size.
- Set maximum number of files at creation time.
- Implementation:
  - Pre-allocate space for the directory when you create the file system.
  - The directory is essentially a big array of structures:
    - char name[max-file-name-size];
    - Either an actual file representation OR an id that easily maps to the file representation.

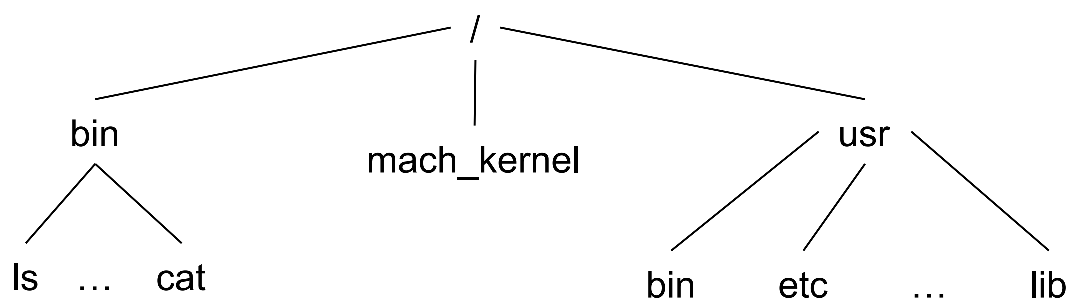
- Pros:
  - Really simple
- Cons:
  - Difficult to organize data
  - No two objects may have the same name.
  - On a multi-user system, users might have name collisions
  - Names are limited.

Hierarchical Naming

- Generalized tree structure
  - Directories are regular files with a special format.
  - A bit in the file meta-data indicates that a file is of type directory.
  - A directory entry is simply a mapping between names and a file index (a collection of name/value pairs).
    - User programs can read directories just like they read files.
    - Only the operating system can write directories (wouldn't want a user to corrupt the directory structure)

- Pros:

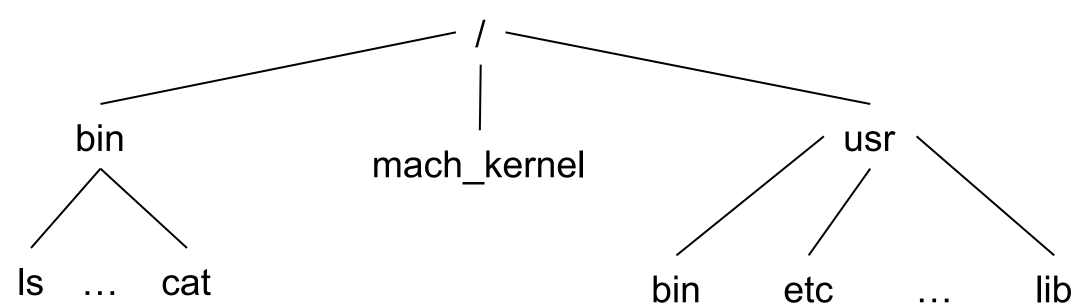
- Cons:



Hierarchical Naming

- Generalized tree structure
  - Directories are regular files with a special format.
  - A bit in the file meta-data indicates that a file is of type directory.
  - A directory entry is simply a mapping between names and a file index (a collection of name/value pairs).
    - User programs can read directories just like they read files.
    - Only the operating system can write directories (wouldn't want a user to corrupt the directory structure)

- Pros:
  - Much better organization
  - Reuses file implementation
- Cons:
  - Slightly more complicated file lookup.



Traditional Directory Implementation

- Directories are represented like files.
- Contents of directories are structured (`dirents`).
  - Name
  - Inode number
  - Type
- Directories grow in chunks of `dirents` that fit on a single disk block.
- Root directory has a designated inode.

The Root Directory

- This is the contents of the "/" directory on my machine.

Name	inumber	Name	inumber	Name	inumber
Applications	113	Desktop Folder	844727	Developer	844731
Documents	937803	Library	213	Marketocracy	937813
Network	84416	System	37	Updaters	937816
Users	38892	Volumes	26447	bin	24377
cdrom	937840	cores	84418	dev	296
etc	25116	home	5	mach_kernel	552433
net	3	opt	937844	private	214
sbin	4512	sw	1024168	tmp	25155
usr	40	var	25156	.	2
..	2				

Walking a Directory Path

- For historical reasons (because original versions of UNIX did this) we call:
  - File index structures: inodes
  - References to file index structures: inumbers
- Given a path /C1/C2/C3 …
  - Start at the root directory (a designated directory with a designated inumber).
    1. Let `inum` = root directory inumber; `current component` = C1
    2. Read the directory data for `inum`
    3. Find the entry with the name equal to the `current component`
    4. Fine the associated inumber
    5. Read the inode for that inumber
       – If it's not a directory, this is a bad pathname
       – If it is a directory, set inum to the inumber; set current component to next part of path and iterate back to step 2.

Directory Example

Assume:

- **Inode 2 is in disk block 100**
- **Inodes fit 8 to the block**
- **Block 100 contains inodes 0-7, 101 contains 8-15, etc.**
- **There are 100 blocks of inodes**

Exercise:
List all the blocks, in order that you need to read to open /usr/lib/libc.a

The number in these inodes is what is found in daddr[0]

	Disk block number	**Contents**							
inodes	100			200					
	101	202	203						
	102	204	205						
	…								

	Disk block number	**Contents**			
Data Blocks	200	., 2	.., 2	bin, 8	
		usr, 16	boot, 35	kadb, 27	
	201	., 11	.., 2	Some text	
		is in	this file		
	202	., 8	.., 2	ls, 91	
		csh, 105			
	203	., 9	.., 16	libc.a, 55	
		font, 77			
	204	., 16	.., 2	lib, 9	
		share, 52	ucb, 15	old, 66	

Directory Example

The number in these inodes is what is found in daddr[0]

	Disk block number	Contents							
inodes	100			200					
	101	202	203						
	102	204	205						
	..								
Data Blocks	200	., 2	.., 2	bin, 8					
		usr, 16	boot, 35	kadb, 27					
	201	., 11	.., 2	Some text					
		is in	this file						
	202	., 8	.., 2	ls, 91					
		csh, 105							
	203	., 9	.., 16	libc.a, 55					
		font, 77							
	204	., 16	.., 2	lib, 9					
		share, 52	ucb, 15	old, 66					

Assume:

- **Inode 2 is in disk block 100**
- **Inodes fit 8 to the block**
- **Block 100 contains inodes 0-7, 101 contains 8-15, etc.**
- **There are 100 blocks of inodes**

Exercise:
List all the blocks, in order that you need to read to open /usr/lib/libc.a

More Directory Fun

- In POSIX, every directory has two special entries “.” and “..”.
  - The “.” directory refers to the directory itself.
  - The “..” directory refers to the parent directory.
  - This is how the file system implements paths such as ../asst2.
- It is possible for more than one directory entry to refer to a single file.
  - *Hard link*: the same inumber appears in two different directories. The reference count for the inumber is incremented.
    - Could you create a hard link between two directories in different file systems?
    - When you remove (unlink) a file, you decrement its reference count and remove a name from a directory. When the reference count goes to zero, the file’s blocks are actually freed.
  - *Soft link* (symbolic link): file that contains the name of another file.
    - Files of this sort are identified by a bit in their file descriptor.
    - When the OS encounters a symbolic link, it continues pathname resolution using the pathname that appears in the file.
    - Can you create a soft link between two directories?
- What is the minimum link count for a directory?

Working Directory

- It is cumbersome (and inefficient for the OS) to use full pathnames every time you reference a file.
- POSIX maintains a single “current working directory” (cwd) for each process. The inumber of the cwd is stored in the user structure.
- When the OS wants to translate a name to an inumber, it looks at the first character in the path. If that character is “/”, the OS begins looking at the root. If it is not a path, the OS begins looking in the current directory.
- Some systems allow you to have more than one current working directory. The list of directories that are in the “current working directory set” are called a search path.