

File Systems – Allocation,
Performance and Security
Strategies
CS 111
Summer 2025
Operating System Principles
Peter Reiher

Outline

- Allocating and managing file system free space
- Other performance improvement strategies
- File system security

Free Space and Allocation Issues

- How do I keep track of a file system's free space?
- How do I allocate new disk blocks when needed?
 - And how do I handle deallocation?

The Allocation/Deallocation Problem

- File systems usually aren't static
- You create and destroy files
- You change the contents of files
 - Sometimes extending their length in the process
- Such changes convert unused disk blocks to used blocks (or visa versa)
- Need correct, efficient ways to do that
- Typically implies a need to maintain a free list of unused disk blocks

Remember Free Lists?

- We talked about them in the context of memory allocation
 - Primarily for variable sized partitions
- These aren't variable sized partitions
 - The free elements are fixed size blocks
 - Might use bit maps or similar structures
- But there are other issues
 - For hard disks, locality matters
 - For flash, issues of erasure and load leveling
- These issues may affect free list organization

Creating a New File

- Allocate a free file control block
 - For UNIX
 - Search the free I-node list/bitmap
 - Take the first free I-node (or a "good" one)
 - For DOS
 - Search the parent directory for an unused directory entry
- Initialize the new file control block
 - With file type, protection, ownership, ...
- Give new file a name
 - Writing it into a directory

Extending a File

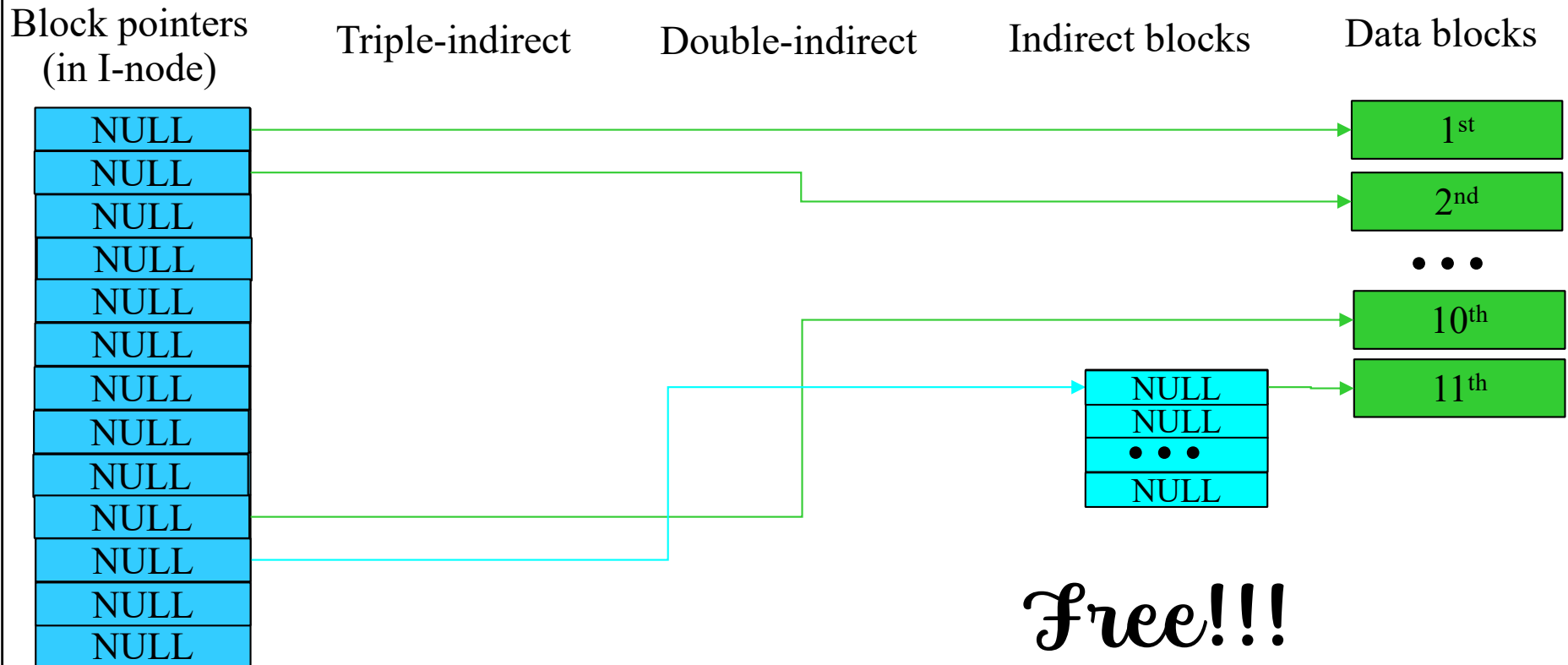
- Application requests new data be assigned to a file
 - May be an explicit allocation/extension request
 - May be implicit (e.g., replacing n bytes with $n+m$ bytes)
- Find a free chunk of space
 - Traverse the free list/bitmap to find an appropriate chunk
 - Remove the chosen chunk from the free list/bitmap
- Associate it with the appropriate address in the file
 - Go to appropriate place in the file or extent descriptor
 - Update it to point to the newly allocated chunk
- Generally don't need to update directory entry

Deleting a File

- Release all the space that is allocated to the file
 - For UNIX, return each block to the free block list
 - DOS does not free space
 - It uses garbage collection
 - So it will search out deallocated blocks and add them to the free list at some future time
- Deallocate the file control lock
 - For UNIX, zero the inode and return it to free list
 - For DOS, zero the first byte of the name in the parent directory
 - Indicating that the directory entry is no longer in use
- For UNIX-style systems, delete the directory entry

Freeing Space From a Unix File

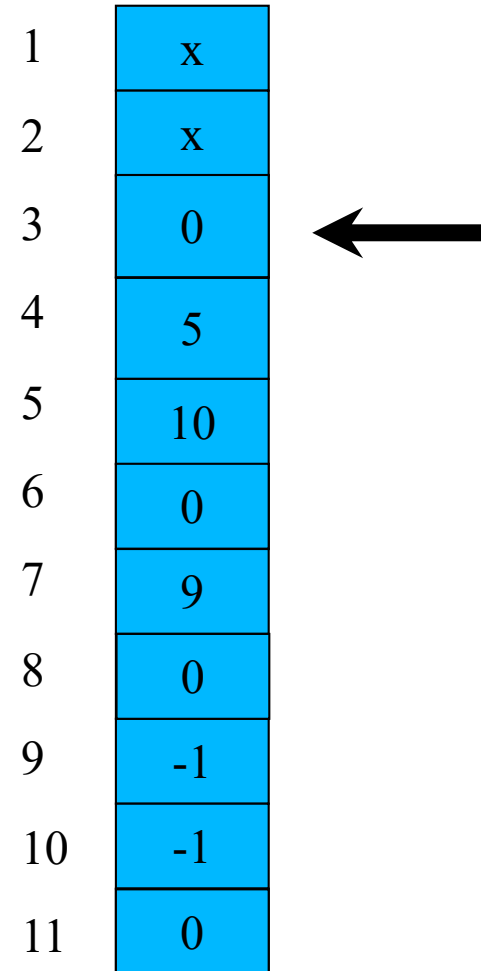
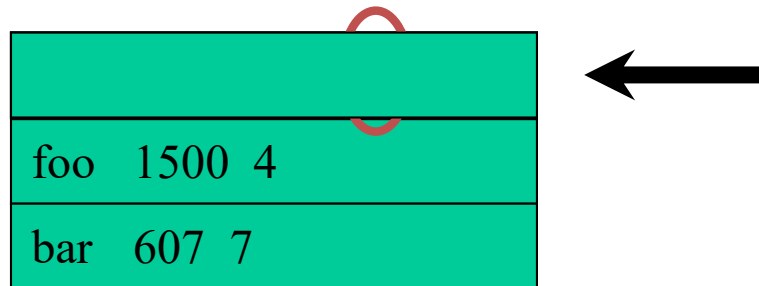
Free!!!



Let's delete this file!

This will require a bit more work

DOS Garbage Collection



Remove myfile.txt

Now let's garbage collect!

A deleted file!

Not deleted

Not deleted

Free Space Maintenance

- File system manager manages the free space
- Getting/releasing blocks should be fast operations
 - They are extremely frequent
 - We'd like to avoid doing I/O as much as possible
- Unlike memory, it matters which block we choose
 - Can't write fully-written flash blocks
 - May want to do wear-levelling and keep data contiguous
 - Other issues for hard disk drives
- Free-list organization must address both concerns
 - Speed of allocation and deallocation
 - Ability to allocate preferred device space

Free Lists vs. Bitmaps

- Free lists are lists of pointers to free memory elements
- Bitmaps are structures with a single bit indicating if a piece of memory is free
- Free lists are searched by following pointers
- Bitmaps are searched by using logical operations like shifting and ORing
 - Which are quicker than following pointers
- Most (but not all) file systems use bitmaps

Performance Improvement Strategies

- Transfer size
- Caching

Allocation/Transfer Size

- Per operation overheads are high
 - DMA startup, interrupts, device-specific costs
- Larger transfer units are more efficient
 - Amortize fixed per-op costs over more bytes/op
 - Multi-megabyte transfers are very good
- What unit do we use to allocate storage space?
 - Small chunks reduce efficiency
 - Large fixed size chunks -> internal fragmentation
 - Variable sized chunks -> external fragmentation
 - Tradeoff between fragmentation and efficiency

Flash Drive Issues

- Flash is becoming the dominant technology
 - Sales overtook HDD in 2021
- Special flash characteristics:
 - Faster than hard disks, slower than RAM
 - Any location equally fast to access
 - But write-once/read-many access
 - Until you erase
 - You can only erase very large areas of memory
- Think about this as we discuss other file system issues

Caching

- Caching for reads
- Caching for writes

Read Caching

- Persistent storage I/O takes a long time
 - Deep queues, large transfers improve efficiency
 - They do not make it significantly faster
- We must eliminate much of our persistent storage I/O
 - Maintain an in-memory cache
 - Depend on locality, reuse of the same blocks
 - Check cache before scheduling I/O

Read-Ahead

- Request blocks from the device before any process asked for them
- Reduces process wait time
- When does it make sense?
 - When client specifically requests sequential access
 - When client seems to be reading sequentially
- What are the risks?
 - May waste device access time reading unwanted blocks
 - May waste buffer space on unneeded blocks

Write Caching

- Most device writes go to a write-back cache
 - They will be flushed out to the device later
- Aggregates small writes into large writes
 - If application does less than full block writes
- Eliminates writes that don't matter
 - If application subsequently rewrites the same data
 - If application subsequently deletes the file
- Accumulates large batches of writes
 - A deeper queue to enable better disk scheduling

Common Types of Disk Caching

- General block caching
 - Popular files that are read frequently
 - Files that are written and then promptly re-read
 - Provides buffers for read-ahead and deferred write
- Special purpose caches
 - Directory caches speed up searches of same dirs
 - Inode caches speed up re-uses of same file
- Special purpose caches are more complex
 - But they often work much better by matching cache granularities to actual needs

Pinning File Data in Caches

- Caching usually controlled by LRU-ish strategy
- But some file data is *pinned* in memory
 - Not subject to cache replacement, temporarily
- Inodes of files that processes have open are one example
 - To ensure quick access to a structure that will probably be used again
- Contents of current working directories may be pinned

File System Security

- An OS's file system is a shared resource
- All processes can request access to files
- In many systems, multiple users share the computer
- Typically, not all users should be allowed to access all files
 - Even with access, maybe not in all ways at all times

How OS Provides File System Security

- All process accesses to files use system calls
- The OS can examine each such system call for security purposes
- Allow access if security permits
- Deny access if it doesn't
- How does the OS decide?

Deciding on File Access

- Many possible approaches
- Most common is based on per-user access control for each file
- A given user is allowed to access a given file in particular ways
 - E.g., read accesses are allowed, write accesses are not
- On each system call, determine if that user should access that file in that way

How?

- This decision requires three things:
 1. Who is requesting the access?
 2. Which file are they trying to access?
 3. What are they asking to do to that file?

Determining *Who*

- Many possible ways, but one way is common
- Each process has a process descriptor
- Each process descriptor contains a field identifying a user who “owns” the process
- System calls are made by a particular process
 - Using a trap instruction while running their code
- OS consults that process’ descriptor to determine which user the process belongs to

Which File Is It?

- System calls accessing files require a parameter identifying the file
 - We'll discuss one typical approach to this in the next lecture
- The OS uses that parameter to identify the file being requested

What Are They Asking To Do?

- Two aspects to this question
 1. *What* is the process trying to do?
 - Determined by which system call is used or by a parameter in the call
 2. *What* is this user allowed to do to the file?
 - Determined by consulting per-file access information
 - Which is typically kept in the file descriptor

An Example

- Process X belonging to user Sue requests to open file foo for read
 - Using a suitable system call for that purpose
- The OS code to handle that system call inspects X's process descriptor
 - Determining it belongs to Sue
- The OS reads the file descriptor for foo
 - Probably from the flash drive
- And looks up information in the descriptor telling what Sue can do with file foo
- If read is allowed, foo is opened for read by X

Seems Pretty Obvious

- How else would you do it?
- But there are actually many different approaches that could be used
 - Identities other than a particular user
 - Methods of determining identity
 - Ways of controlling who can do what with each file
 - Granularities other than whole file accesses (such as accessing only a range of a file)
- Some systems do some of this differently

Then What?

- Presumably user Sue actually wants to read data from the file she opened
- Her process X will use a system call (like `read()`) to do so
- We could repeat the whole procedure of checking access
- But we typically don't
 - Process X successfully opened for read, so now it can read without further security checks

Another Issue

- Sue has read 1000 bytes of data from file foo
 - Putting them into a buffer in her process' data area
- Now the owner of file foo changes access permissions on the file
 - Taking away Sue's access
- What about that 1000 bytes of data Sue shouldn't be able to read any more?
- Tough luck, it's too late

An Implication

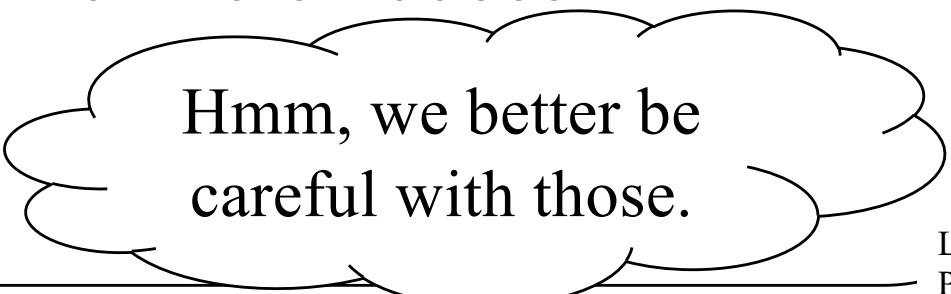
- Process access to a file is determined at open system call time
 - It's not rechecked later
- So if a process can read a file's data at time T_1
 - It's got a copy of that data in its memory pages
- Typically there is no way to “take the data back”
- Or prevent the process from sharing it
- Let someone read a file's secret and it's his secret now
- In turn implying the security depends on the process' own security

Another Security Issue

- Files and file systems are abstractions
 - Built on top of lower level software (e.g., device drivers)
 - And ultimately on hardware (e.g., a flash drive)
- There are often paths to access these lower level elements directly
 - Bypassing the access control we just discussed
- True file system security requires protecting those alternate access paths

For Example,

- Remember device special files from the device driver lecture?
- They allow block-level access to the disk
 - If the special file permissions allow
- Which allows an authorized user to look at the file blocks
- Without consulting the file's access permissions



Hmm, we better be careful with those.

Conclusion

- File systems require well designed methods of allocating, referencing, and freeing memory
- Memory allocation information for file systems must also be stored persistently
- Achieving good performance in file systems requires sophisticated caching strategies
- File system security is typically based on per-file, per-user access control