**Secure File Shredding in Go: Design and Implementation**

**Problem Description**

Shred tool in Go

Implement a Shred(path) function that will overwrite the given file (e.g. “randomfile”) 3 times with random data and delete the file afterwards. Note that the file may contain any type of data.

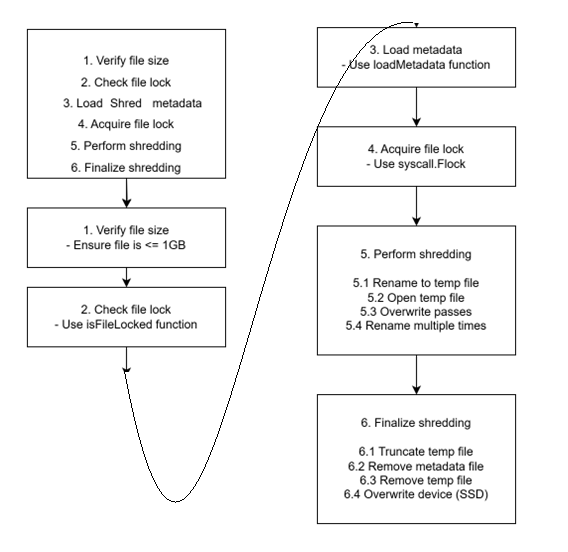
 You are expected to give information about the possible test cases for your Shred function, including the ones that you don’t implement, and implementing the full test coverage is a bonus :)

 In a few lines briefly discuss the possible use cases for such a helper function as well as advantages and drawbacks of addressing them with this approach.

**Project Assumption**

* Overwrites the file content with random data for a specified number of passes to ensure that the original data is difficult to recover.
* Set a file size threshold above which special file handling is required (e.g., 1GB or 10GB).
* The shred function is invoked for a few files dispersed throughout the system, possibly at different times (even if the file system is extensive). In scenarios where the shred function is used intensively with high frequency in a large, busy and vary file systems, the implementation must be adjusted to include advanced batch operations across the file system rather than individual block or sector operations.
* The file resides on a common and standardized file system where system file access functions can be successfully called, and OS is preemptive multitasking operating system.
* Logical to Physical Mapping (Wear-Leveling Techniques): The function does not account for wear-leveling techniques used in SSDs. These techniques, which involve the Flash Translation Layer (FTL) managing the mapping between logical block addresses (LBAs) and physical flash memory locations, distribute write and erase cycles evenly across all memory cells. This can complicate data destruction.
* Hardware Aspects (Residual Magnetic Traces in HDDs): The function does not address hardware-specific issues such as residual magnetic traces in HDDs. These traces can potentially be read using advanced forensic techniques.

**System Design**

****

**Focus on Data Sanitization (Irrecoverability) : Overwrite content and metadata.**

* Overwrite not only file content but also file name (file nod entry or file system metadata) with random data multiple times to make the original data irrecoverable. The file name is renamed and truncated to zero bytes and then deleted it, ensuring that no traces are left behind.
* Ensure file system metadata (file name and directory entry) is also irrecoverable so no residual information is left behind.
* Use crypto/rand to generate cryptographically secure random data**.**

**Transactional Safety : Use Temporary file.**

* Uses a temporary file to ensure atomicity and allow for a consistent state during the shredding process. If an error occurs, you can revert to the original file without data corruption.
* isolates the shredding operation from the original file. This helps manage concurrent access issues, no other process can modify the file while it is being shredded.
* The Shred function uses a temporary file to perform the overwrite operations.
* If the shredding process is interrupted, we can consider process resumption.

**Process Resumption: Save progress for interrupted processes.**

* Save progress in a metadata file, including the current pass and temporary file path, allowing the process resume from the last saved state, ensuring complete shredding without data miss.

**Treat Concurrency: use locks.**

* Prevent concurrent race conditions aspects during the shredding process

**Error Handling:**

* Handle errors gracefully and allow for recovery. Maintain a consistent state even if the process is interrupted.

**Future Design Considerations and Edge Cases**

**I.** **Performance in Data Sanitization (Irrecoverability)**

1. **Improved Obscurity of Shredded File**

Obscure the exact location of the original physical data blocks file by crating

a fake appearance.

This makes it difficult for any recovery attempt to determine the specific blocks that contained the removed file by blurring the boundaries between the file's data blocks and the adjacent data blocks.

Steps for that:

1. Identify Adjacent physical Block close to file's allocated blocks.
2. Read the data from these adjacent blocks to understand the current data patterns. That is crucial to ensure that the patterns used for overwriting blend seamlessly with the existing data.
3. Create an overwrite with pattern that mimics or matches the data found in the adjacent blocks. The goal is to generate a pattern that, when written, appears as if the data from the adjacent blocks extends into the file's blocks and in that way effectively hide the real boundaries (hide real block of removed data) of the original file.
4. After each pass, verify that the overwritten pattern extends correctly into the adjacent blocks without leaving discernible borders.
5. **Cacheability and Bufferability in Secure File Shredding**

When designing a secure file shredding function, considering the cacheability and bufferability of the system can significantly impact the effectiveness, performance, security, and safety of the shredding process.

1. **Cryptographic randomness**
2. Cryptographic Randomness: Ensure that the random data used for overwriting is cryptographically secure.
3. Different Patterns: Use different patterns of data (e.g., alternating bit patterns) to further ensure the original data is unrecoverable.
4. **Hardware-level secure erase**

Exploit hardware-level secure erase commands if available benefit of more safe erasing proc.

**II. Performance in Speed Optimization**

1. **Distributed File Systems and Networked FS:**

Treat differentiated based on file system type (e.g., NTFS, ext4, FAT32, Networked Filesystems: NFS, SMB) covering Cross-File-System Compatibility.

1. **Batch Processing:**

Implement batch processing for overwriting large number of files to improve performance. Aggregate multiple shred operations to minimize the performance impact.

1. **Parallelism:**
2. Optimize the scanning and pattern generation process to handle multiple files concurrently.
3. Utilize parallel processing where possible (e.g. multithreads to write data) to speed up the shredding process.
4. **Scalability**:
   1. Large Files: Ensure the function can handle very large files efficiently.
   2. Chunked Processing**:** For very large files, process the file in chunks in a single operation to avoid excessive memory usage and to manage I/O more efficiently.
   3. Enhance the shredding process using low-level IO file operations or block/sector-level access.

**III. Operational Performance**

1. **Concurrency Handling:**

Implement mechanisms to handle situations (alerts / or defined rules apply for specific case) where concurrent access is detected after starting the shredding process. For example: cancel destruction and rollback at original if someone tries to access the file during processing. – someone try to lock the already locked file!

1. **Resilient Error handing and Recovery:**
   1. Retry Mechanisms: Implement retry mechanisms for transient errors (e.g., temporary file rights access or writes error issues) or because file is already in use or locked by another process, providing appropriate error messages and options to retry.
   2. Logging: Add detailed logging to help diagnose issues that occur during the shredding process.
2. **Security Enhancements:**
   1. Secure Metadata Storage: Ensure that metadata is stored securely to prevent tampering or unauthorized access.
   2. Integrity Checks: Implement integrity checks to verify that the file content and metadata have not been tampered with during the shredding process.
3. **User Configurability:**
   1. Customizable Parameters: Allow users to customize parameters
   2. User Feedback: Provide feedback to the user during the shredding process, such as progress indicators or logs.
4. **Compliance with Standards:**
   1. Data Protection Regulations: Ensure that the shredding process complies with relevant data protection and privacy regulations (e.g., GDPR, HIPAA) and industry standards (e.g., NIST guidelines).
   2. Overwriting Patterns: Some standards recommend specific patterns for overwriting data (e.g., alternating binary patterns) to ensure more thorough erasure.
5. **Filesystem corruption safeguards:**

Implement safeguards to handle potential file system corruption scenarios, ensuring data integrity and the ability to resume or restart the shredding process.

1. **Permission Issues handling:**

Handle situations where the process lacks sufficient permissions to overwrite or delete/rename the file, providing clear error messages and guidance for resolving permission issues.

1. **Resilience to external unexpected:**
   1. System: Enhance the resilience of the shredding process to interruptions such as system crashes or forced terminations, ensuring that the process can be reliably recovered.
   2. Disk Full Situations: Handle scenarios where the disk is full and additional writes may fail.
   3. Power Loss: Consider the impact of power loss during the shredding process and implement mechanisms to resume or restart securely.
   4. Access restriction changes: Manage access restriction or context changes while processing is ongoing.
   5. Security against Hacking: Ensure that unauthorized users cannot access or tamper with the secure shredding process.