Advanced Operating Systems
Project 4: File Systems
Design Document

The questions in this design document should reflect the design of the code you wrote for the project. **Your grade will reflect both the quality of your answer in this document and the quality of the design implementation in your code.** You may receive partial credit for answering questions for parts of the project that you did not get to implement, but you must indicate in your answer that there is no corresponding implementation, or you will not receive any credit.

For each question, **you should include both the name of the file(s), function name(s), and the line numbers where the relevant code may be found** – both the code that answers the question directly and any function that you refer to in your answer.

These design documents will be completed and submitted as a group. Please use this document as a guide for design and discuss the questions and their potential answers prior to beginning implementation. **All text in *italics* is intended to be replaced with your answer**. When you have completed your design document, submit it to the Canvas assignment Project 1 (Threads): Design.

Team Information Fill out the following information for each member of your team.

Name: *Yunseok Kang*
EID: *yk9784*
Email: *yunseok.kang@utexas.edu*
Unique Number: *51275*

Name: *Dohyun Kwon*
EID: *dk29493*
Email: *gundo0109@utexas.edu*
Unique Number: *51275*

Name: *Stefanus Adrian*
EID: *sja2673*
Email: *stefanus.adrian@utexas.edu*
Unique Number: *51275*

Preliminaries Please provide any preliminary comments and cite consulted sources.

If you have any preliminary comments on your submission or notes for the TAs, please give them here.

Your answer here.

Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Your answer here.

INDEXED AND EXTENSIBLE FILES (10 points)

The 10 total points are divided equally among the following sub-questions.

A1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary for your indexed and extensible file implementation. Identify the purpose of each in 25 words or less.

```
#define INODE_NUM_BLOCKS 124  // total number of data blocks (122+1+1)
#define INODE_NUM_DIRECT 122  // number of direct blocks
#define INODE_IND_IDX INODE_NUM_DIRECT // indirect block index
#define INODE_DUB_IND_IDX INODE_NUM_BLOCKS - 1 // doubly indirect block index
#define INODE_NUM_IN_IND_BLOCK 128 // total numbef of blocks


struct inode_indirect_sector
  {
    block_sector_t block_idxs[INODE_NUM_IN_IND_BLOCK];
  };

struct inode_disk
  {
    // Array of direct block indices where actual file data is stored.
    block_sector_t block_idxs [INODE_NUM_BLOCKS];

    bool is_dir;
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    bool is_symlink;
  };

struct inode
{
  // Mutex for protecting from  concurrent modifications.
  struct lock lock;

  // for synchronizing read operations that extend beyond the end of the file.
  struct lock eof_lock;

  // Used to wait for inode data to be fully loaded on open.
  struct condition data_loaded_cond;

  // indicating whether the inode data has been successfully loaded and is ready to use.
  bool data_loaded;

  // Mutex for directory operations, providing synchronization during modifications.
  struct lock dir_lock;

  // List element for including this inode in global lists.
```

```
  struct list_elem elem;

  // Disk sector number where this inode's data is stored.
  block_sector_t sector;

  // indicating if this inode represents a directory  or a file.
  bool is_dir;

  // Size of the file or directory in bytes.
  off_t length;

  // Counter tracking how many processes have this inode open.
  int open_cnt;

  // Boolean indicating if the inode has been marked for deletion.
  bool removed;

  // Counter to manage write denial states. When greater than 0, writing is denied.
  int deny_write_cnt;

  // Physical size of the inode content
  // may differ from logical size due to things like sparse allocation
  int physicalsize;

  // indicating if this inode is a symbolic link.
  bool is_symlink;
};
```

A2. What is the maximum size of a file supported by your inode structure? Show your work.

The inode structure allows a maximum file size of 62,464 + 65,536 + 8,388,608 = 8,516,608 bytes, which is approximately 8.1 megabytes.

1. Direct Blocks : 122 direct blocks×512 bytes/block=
   62,464 bytes
2. Single Indirect Block: 128 pointers/block×512 bytes/pointer=
   65,536 bytes
3. Double Indirect Block: 128×128 pointers/block×512 bytes/pointer=8,388,608 bytes

A3. Explain how your code avoids a race if two processes attempt to extend a file at the same time.

When a file needs to be extended, the `eof_lock` is used to ensure that only one process can perform the extension at a time. If data is written beyond the current file size, the lock is rechecked, and extension proceeds only if necessary. This approach guarantees data consistency when multiple processes write to a file simultaneously, minimizes the use of `eof_lock` to reduce performance degradation, and allows general inode operations to proceed concurrently with extension tasks.

A4. Suppose processes A and B both have file F open, both positioned at end-of-file. If A reads and B writes F at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B writes nonzero data, A is not allowed to see all zeros. Explain how your code avoids this race.

In the PintOS file system, race conditions involving concurrent reads and writes at the end of a file are managed using specific locks (`eof_lock`) and condition variables (`data_loaded_cond`) integrated into the inode structure. When a process attempts to write beyond the end-of-file, it acquires `eof_lock` to serialize file extensions and ensure data integrity. Concurrent reads during this write operation either wait for the write to complete using condition variables or read the fully written data, ensuring they do not read uninitialized or inconsistent data. This synchronization mechanism effectively prevents race conditions and maintains the consistency and reliability of file operations.

A5. Explain how your synchronization design provides "fairness". File access is "fair" if readers cannot indefinitely block writers or vice versa. That is, one or many processes reading from a file cannot forever prevent another process from writing to the file, and one or many processes writing to a file cannot forever prevent another process from reading the file.

In the described synchronization design for reader-writer fairness, reading operations check for data availability and lock the file only when necessary, such as when waiting for data to be loaded or written at the end of the file. This approach minimizes lock duration, allowing other operations like writing to proceed with minimal delay. Writing operations that extend the file size use the `eof_lock` to avoid inconsistencies during concurrent writes. This lock is released quickly after the write is complete, reducing wait times for subsequent operations.

A6. Is your file layout a multilevel indexed structure? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative file structure, and what advantages and disadvantages does your structure have, compared to a multilevel index? In your explanation, include at least two reasons to support your choices.

Yes, the file layout in the provided inode structure utilizes a multilevel indexed structure, incorporating direct, indirect, and doubly indirect blocks. The inode structure in the provided file layout uses a multilevel indexed structure with direct, indirect, and doubly indirect blocks, designed to balance performance, storage efficiency, and scalability. This setup allows for quick access to small files through direct blocks, while indirect and doubly indirect blocks efficiently handle larger files by scaling complexity as needed. This hierarchical arrangement optimizes for performance by minimizing disk reads for small to medium files and manages larger files without overwhelming the inode with too many pointers, thus supporting a wide range of file sizes efficiently and maintaining manageable inode sizes even as file demands grow.

SUBDIRECTORIES (10 points)

The 10 total points are divided equally among the following sub-questions.

B1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary for your subdirectory implementation. Identify the purpose of each in 25 words or less.

```
struct dir
  {
  ...
  struct lock *lock;                  /* Shared across dirs of inode. */
  ...
  };

struct inode_disk
{
  ...
  // Flag to indicate if this inode represents a directory.
  bool is_dir;
  ...
};

struct inode
{
   ...
  // Mutex for directory operations, providing synchronization during modifications.
  struct lock dir_lock;

  // indicating if this inode represents a directory  or a file.
  bool is_dir;
  ...
};

struct thread
  {
    ...
    // Current working directory of the process, initialized at filesystem setup
    void *cwd;
    ...
  };
```

B2. Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

The `dir_open_dirs()` function in the filesystem implementation supports both absolute and relative path traversals by starting at the root directory for absolute paths and the current working directory for relative paths. It processes the path by segmenting it using slashes("/"), handling each segment

iteratively, and opening the corresponding directory or file inode. The traversal method varies with absolute paths offering a consistent and unambiguous resource location starting from the filesystem's root, and relative paths providing flexibility but depending on the process's current directory context. This design allows for handling a range of use cases effectively, ensuring error checks for nonexistent paths or permissions issues, and accommodating the needs for both simplicity in resource location and flexibility in directory navigation.

B3. How do you prevent races on directory entries? For example, only one of two simultaneous attempts to remove a single file should succeed, as should only one of two simultaneous attempts to create a file with the same name, and so on.

Race conditions on directory entries are mitigated through several synchronization strategies. Each directory employs a synchronization lock, ensuring that modifications such as file creation or deletion are handled sequentially by only one process at a time. Operations on directories are transaction-like, allowing for atomic changes that ensure consistency and prevent partial updates. Inode reference management further supports safe deletions by keeping files accessible until all references are closed. Additionally, consistency checks prevent the creation or deletion of files with conflicting names. Finally, serialized access to critical shared resources like the free map guarantees that concurrent access does not lead to corruption or inconsistent filesystem states. These mechanisms collectively enhance the filesystem's stability and reliability, crucial for educational and experimental explorations of operating system concepts.

B4. Does your implementation allow a directory to be removed if it is open by a process or if it is in use as a process's current working directory? If so, what happens to that process's future file system operations? If that's not allowed, how do you prevent it?

Directories cannot be removed if they are open by a process or if they serve as a process's current working directory. This safeguard is enforced through several mechanisms: each directory inode tracks its open count, which must be zero before the directory can be removed; operations attempting to delete a directory must check if it's the current working directory of any process; and synchronization is managed with locks to prevent concurrent access during critical operations like deletion. If a process attempts to delete a directory in use, the operation fails, ensuring filesystem integrity and preventing errors in subsequent file operations on non-existent directories.

B5. What type of variable/structure did you use to represent the current directory of a process? Why did you choose this representation? Give at least two reasons.

The current directory of a process is represented using a structure called `struct dir`, which is a pointer stored within each process's thread structure. This approach was chosen for several reasons.

1. Direct Access to Directory Data: The `struct dir` holds a reference to the `struct inode` of the directory, allowing direct access to the underlying filesystem data. This is crucial for efficient management and traversal of directory entries, as it minimizes the need for repeated disk accesses to locate directory data, thus speeding up file operations within the current directory.

2. Synchronization and Safety: The `struct dir` includes synchronization primitives, such as locks, that help manage concurrent access to the directory. This is particularly important in a multi-threaded environment where multiple processes might attempt to read from or write to the directory

simultaneously. By encapsulating these mechanisms within the directory structure, it ensures that directory operations are performed safely and atomically, preventing data races and inconsistencies.

These reasons underscore the benefits of using a dedicated structure like `struct dir` to manage directory-related operations, providing both performance efficiencies and maintaining integrity in the management of directory data.

SURVEY QUESTIONS (optional) Answering these questions is optional, but it will help us improve the course in future semesters. Feel free to tell us anything you want – these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the semester.

In your opinion, was this assignment, or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?

Your answer here.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Your answer here.

Is there some particular fact or hint we should give students in future semesters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

Your answer here.

Do you have any suggestions for the TAs to more effectively assist students, either for future semesters or the remaining projects?

Your answer here.

Any other comments?

Your answer here.