2017 Spring CS330 Project 4: FILE SYSTEMS DESIGN DOCUMENT

---- TEAM ----

>> Team name.

Team 19

>> Fill in the names, email addresses and contributions of your team members.

Hyeonwoo Kang <kanglib@kaist.ac.kr> (70)

Young-geol Jo <rangewing@kaist.ac.kr> (30)

>> Specify how many tokens your team will use.

2

INDEXED AND EXTENSIBLE FILES

=====================

---- DATA STRUCTURES ----

A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

```
+enum file_type {
+  FILE_TYPE_REGULAR,
+  FILE_TYPE_DIR,
+};
```

enum file_type: Indicates whether a file is a regular file or a directory.

```
struct inode_disk
   {
-    disk_sector_t start;              /* First data sector. */
+    enum file_type type;              /* File type. */
     off_t length;                     /* File size in bytes. */
+    disk_sector_t parent;             /* Parent directory pointer. */
     unsigned magic;                   /* Magic number. */
-    uint32_t unused[125];             /* Not used. */
+    disk_sector_t pointers[14];       /* Block pointers. */
+    uint32_t unused[110];             /* Not used. */
   };

@@ -36,9 +40,12 @@ struct inode
     int open_cnt;                     /* Number of openers. */
     bool removed;                     /* True if deleted, false otherwise. */
     int deny_write_cnt;               /* 0: writes ok, >0: deny writes. */
-    struct inode_disk data;           /* Inode content. */
+    int pwd_cnt;                      /* 0: remove ok, >0: deny remove. */
+    struct lock mutex;                /* Synchronization. */
   };
```

type: Indicates whether this inode represents a regular file or a directory.

parent: Stores a pointer to the parent directory. (Directory only)

pointers: Stores direct (0-11), indirect (12) and doubly indirect (13) pointers to the data blocks.

pwd_cnt: Counts the number of processes which have this inode as a working directory.

mutex: Ensures mutual exclusion on this inode.

```
+static struct lock open_inodes_mutex;
```

open_inodes_mutex: Ensures mutual exclusion on open_inode.

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

Direct blocks: $12 \times 512 = 6\mathrm{K}$ bytes

Indirect blocks: $(512/4) \times 512 = 64\mathrm{K}$ bytes

Doubly indirect blocks: $(512/4)^\wedge 2 \times 512 = 8\mathrm{M}$ bytes

Total: $6\mathrm{K} + 64\mathrm{K} + 8\mathrm{M} = 8\mathrm{M} + 70\mathrm{K} \cong 8\mathrm{M}$ bytes

---- SYNCHRONIZATION ----

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

A process must hold the lock of the inode (mutex) to extend a file. As file extension is performed sequentially but not concurrently, there cannot be a race.

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.

A process must hold the lock of the inode (mutex) to read or write (and extend) a file. As file extension and read/write operations are performed sequentially but not concurrently, there cannot be that kind of race.

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

A process acquires and releases the lock of the inode (mutex) to access a file. The next reader/writer is decided by the waiters' priorities, therefore file access is fair.

---- RATIONALE ----

A6: Is your inode structure a multilevel index? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative inode structure, and what advantages and disadvantages does your structure have, compared to a multilevel index?

Yes; we chose this UFS-like design of inode structure for convenience of implementation.

SUBDIRECTORIES

===========

---- DATA STRUCTURES ----

B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

```
struct dir
   {
     struct inode *inode;                 /* Backing store. */
     off_t pos;                          /* Current position. */
+    disk_sector_t parent;                /* Parent directory. */
   };
```

parent: Stores a pointer to the parent directory.

```
struct file
   {
+    enum file_type type;        /* File type. */
     struct inode *inode;        /* File's inode. */
     off_t pos;                 /* Current position. */
     bool deny_write;            /* Has file_deny_write() been called? */
+    struct dir *dir;            /* Used for directories. */
   };
```

type: Indicates whether this file is a regular file or a directory.

dir: Stores a pointer to the directory structure. (Directory only)

```
@@ -154,6 +154,11 @@ struct thread
     mapid_t mmap_n;                      /* Maximum used mapids. */
 #endif

+#ifdef FILESYS
+    /* Owned by filesys/filesys.c. */
+    struct dir *pwd;
+#endif
+
     /* Owned by thread.c. */
     unsigned magic;                      /* Detects stack overflow. */
   };
```

pwd: Stores a pointer to the directory structure of process' current working directory.

---- ALGORITHMS ----

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

If the given path ends with '/', the path should be resolved to a directory. If the path starts with '/', the path is absolute and the traversal starts from the root directory; otherwise the path is relative and the traversal starts from the working directory. Then the path is tokenized looked up sequentially. "." is skipped and ".." goes to the parent directory. At the end, we check the limit of the length of the real file name, and return the pointer the directory structure and the real file name.

---- SYNCHRONIZATION ----

B4: 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.

A process must hold the lock of the inode (mutex) to access a file in a directory. As file access is performed sequentially but not concurrently, there cannot be a race. The latter operation will peacefully fail.

B5: 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 not, how do you prevent it?

Our implementation maintains open and pwd (working directory) counts of a file to prevent a directory to be removed in the certain conditions.

---- RATIONALE ----

B6: Explain why you chose to represent the current directory of a process the way you did.

We could still implement the feature with the directory APIs instead of the inodes'. They are high-level and look good.

BUFFER CACHE

==========

---- DATA STRUCTURES ----

C1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

```
+struct line {
+  int flags;
+  disk_sector_t sector_idx;
+  uint8_t buffer[DISK_SECTOR_SIZE];
+};
```

struct line: Represents a cache line.

- flags: Stores several flags: present, accessed, dirty.
- sector_idx: Stores the sector number which the line stores.
- buffer: Caches the sector data.

```
+struct read_ahead_job {
+  disk_sector_t sector_idx;
+  struct list_elem elem;
+};
```

struct read_ahead_job: Represents a read-ahead job.

- sector_idx: Stores the sector number to be processed.
- elem: List element.

```
+static struct line cache[FILESYS_CACHE_MAX];
+static int cache_cursor;
+static struct lock cache_lock;
+static struct list read_ahead_queue;
```

cache: The buffer cache.

cache_cursor: A cursor used to implement the cache replacement algorithm.

cache_lock: Ensures mutual exclusion on this inode.

read_ahead_queue: The read-ahead job queue.

---- ALGORITHMS ----

C2: Describe how your cache replacement algorithm chooses a cache block to evict.

We implement clock algorithm as our cache replacement algorithm. It loops through the frame table infinitely to:

1. Check if the cache block is present.
2. Check the accessed flag of the cache line to find a victim.
3. Clear the accessed flag of the cache line.

When a victim is found, it exits the infinite loop immediately.

C3: Describe your implementation of write-behind.

A cache block is not written to the disk immediately after the content is changed. When a dirty cache block is evicted, it is written back to the disk. There is also a high-priority thread that writes dirty cache blocks back and clears dirty flags periodically.

C4: Describe your implementation of read-ahead.

When the file system requests a cache access, a read-ahead job is pushed to the read-ahead queue. The special thread processes the jobs to load the cache lines with the ahead sectors.

---- SYNCHRONIZATION ----

C5: When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?

A process must hold the cache lock to read/write/evict a buffer cache block. It ensures mutual exclusion.

C6: During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?

Same as above.

---- RATIONALE ----

C7: Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit from read-ahead and write-behind.

Frequent access to a single short file, especially from multiple processes, would benefit from buffer caching. A sequential access would benefit from read-ahead because of improved parallelism. A repeated access to a small range in a file would benefit from write-behind because of fast access time of RAM.

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the course in future quarters. 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 quarter.

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

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

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

Do you have any suggestions for the TAs to more effectively assist students in future quarters?

Any other comments?