CS 162
Project 3: File Systems
Design Document

GSI: Andrew Chen
Group Number: 28
Chuchu Zhang chuchuzoe@berkeley.edu
Chuhan Zhang chz.cookie@berkeley.edu
Shu Li shuli1995@berkeley.edu
Yichao Shen mark.shen@berkeley.edu

Preliminary Questions
==========

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

2) 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.

http://courses.cs.vt.edu/~cs3204/spring2007/pintos/Project4SessionSpring2007.pdf

Buffer Cache
==========

3) 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.

create a new file filesys/cache.c

```
struct cache_block {
  char data[256];             /* storage for actual block in disk */
  block_sector_t file_start;   /* disk inode position on disk.*/
  block_sector_t block_index;  /* identify block's location on disk */
  bool dirty;                 /* for write-back check */
  struct list_elem elem;       /* list_elem in LRU list */
  int ref_cnt;                /* number of threads accessing */
  struct lock cache_lock;      /* to synchronize dirty and ref_cnt */

}

/* Overall number of cache_blocks created */
```

```
static int cache_block_count;

/* Max number of cache_blocks */
const int max_cache_blocks = 64;

/* LRU list containing all cache_block */
static struct list LRU;

/* Lock enforcing consistency of LRU list */
struct lock LRU_lock;
```

4) Describe how your cache replacement algorithm chooses a cache block to evict.

Traverse reversely the LRU list. For each element visited, it acquires the cache_lock and check the value of ref_cnt. If ref_cnt is non-zero, release the lock and looks for the next element in the list. When a element is found with ref_cnt 0, set it to -1 and release cache_lock. Then acquires LRU_lock and remove it from the list LRU, perform write back (if necessary) and recycle the memory.

5) An optional part of this project is making your buffer cache periodically flush dirty blocks to disk. Describe a possible implementation strategy for this feature.

Similar to the replacement policy, traverse the LRU list and for each element, acquires the cache_lock to check ref_cnt. If ref_cnt is 0, the data is not being accessed at the moment, so perform the write back to disk and clear the dirty state, finally release the cache_lock.

6) An optional part of this project is implementing read-ahead caching for files in your buffer cache. Describe a possible implementation strategy for this feature.

Add a const u_int8 prefetch_blocks, struct condition prefetch_cond, both are file scope under cache.c.

Use the consumer-producer model, the producer (prefetcher), created in an independent thread, grabs the LRU_lock and sleeps on prefetch_cond (releases the lock in cond_wait) while the length of list is max_cache_blocks, and wakes up (reacquires the lock) to prefetch prefetch_blocks blocks next to block_index of the first cache_block in the LRU list, since according to the LRU algorithm, the first cache_block in the list is the most recently accessed one. Then it releases the lock. (The process of adding cache_blocks to a full list will trigger replacement algorithm.) The consumer sleeps on prefetch_cond if the list is empty (very unlikely except just after system starts up). After waking up, it searches for the requested block. If found, it acquires cache_lock and increment the ref_cnt and releases the lock; if not found, consumer sends request of a specific block_index to cache_get_block() and sleep for IO as normal.

Note that the replacement algorithm is still compatible with this scheme. It runs whenever producer fetches new blocks and the buffer cache is out of space.

It is be difficult to determine the amount of blocks to read ahead, too many of them would increase the cost of a cache miss, and too few would not effectively improve the IO performance. In addition, read ahead is mostly beneficial during a sequential read; during random accesses it would actually become useless and hurt the performance.

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

The replacement algorithm has to check that ref_cnt of the particular cache_block is 0 (with cache_lock acquired) before deciding to evict that cache_block.

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

Any accesses of the buffer cache has to first acquire cache_lock and increment ref_cnt. Add a procedure that if ref_cnt is -1, access is denied, and lock is released.

Indexed and Extensible Files
==========

9) 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.

In inode.c

```
struct inode_disk
  {
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    block_sector_t direct[50];          /* Direct blocks. */
    block_sector_t single_indirect[75]; /* Singly indirect blocks. */
    block_sector_t double_indirect[1];  /* Doubly indirect blocks. */
  };

struct inode
  {
    struct list_elem elem;     /* Element in inode list. */
    block_sector_t sector;     /* Sector number of disk location. */
    int open_cnt;              /* Number of openers. */
    bool removed;              /* True if deleted, false otherwise. */
    int deny_write_cnt;        /* 0: writes ok, >0: deny writes. */
    struct lock size_lock;     /* This lock is for file extension. */
    int type;                  /* TYPE_FILE: directory TYPE_Dir:file. */
```

};

10) What is the maximum size of a file supported by your inode structure? Show your work.
50 * 512 + 75 * 512 * 128 + 1 * 512 * 128 * 128 = 13329408 bytes = 12.7 MB

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

If one process wants to write at the end of a file, it needs to acquire the size_lock in that file's inode struct. This process will write to the file first and then update the metadata of the inode. Then the thread will release the size_lock so the other process can extend the file and write to the file.

12) 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.

Two situations will happen when A reads and B writes F at the same time.

1. A reads after B begins to write. Since B first writes all data to the file and then update the metadata, A cannot access and read the data beyond the old EOF before B finishes writing. This is because when B writes, the length of the file is still the old length and A can read nothing.
2. A reads before B begins to write. A can read nothing of what B writes because B does not write anything yet.

13) Is your synchronization design "fair"? If so, explain how your synchronization design provides fairness. If not, explain how you could add fairness to your design. 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.

The synchronization for read/write without extending the file is fair because we do not have any locks.

The synchronization is not "fair" if two processes both want to extend the EOF. If both processes want to write, one process has to wait for size_lock. This situation could not be improved since extending a file and writing data into the new section must be atomic.

If one writes and one reads, the reading process has to wait for the writing process. This situation could be improved by updating file length byte by byte: every time the writing process successfully writes a byte to the file, update the length immediately.

14) 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?

We use multilevel index. We have 50 direct blocks, 75 singly indirect blocks and a doubly indirect block. We choose this combination because we want a moderate number of direct blocks. More direct blocks will facilitate small file access but too many direct blocks will lead to most data being stored in doubly indirect blocks instead of singly indirect blocks for large files, which will slow the looking up process.

Subdirectories
==========

15) 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.

To add in thread.h

```
struct dir * cwd;           /* Current working directory of thread */
```

To add in inode.h

```
enum file_type                      /* To identify if what type of file the
{                                           inode is used for */
        TYPE_FILE,
        TYPE_DIR
};
```

To add in struct inode in inode.c

```
struct inode
{
Add:
   struct lock dir_lock;     /* lock for the dir access*/
}
```

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

Check that if the given path starts with "/", then it's an absolute path; if not, it's a relative path and we need to append it to the cwd of the thread to get the full path.
Then we repetitively call get_next_part(char part[NAME_MAX + 1], const char **srcp) and put the full path as the second argument until we read the end of it. We'll get directories from highest level to lowest level and the file in the end. We dir_open_root(void) first and find the first file inode and continue dir_open(struct inode*inode) or get the final file data block.

17) 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.

Whenever modification to a directory entry is attempted, it requires the struct lock dir_lock in dir->inode first so that only one attempt succeed to remove or modify the file. After the job is done, the dir_lock gets released so others can modify the directory entries.

18) 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?

When the directory is opened by a process, which is indicated by open_cnt in dir->inode, we won't allow the removal of directory after we checked that open_cnt is bigger than 0. Also we check if thread's struct dir* cwd is the directory to be removed, if yes, we don't allow it to be removed.

19) Explain why you chose to represent the current directory of a process the way you did.

Because I store a struct dir* cwd in each thread(corresponds to a single process), when a new process is created, the cwd of the thread will either be set to root directory or be set as the same of its parent's. Whenever the user opens a file or directory by providing the relative path, relative path can be directly appended to struct dir* cwd. Also we ensure each process has separate and independent current working directory.

Student Testing
==========

20) Describe your testing plan for your buffer cache. For each of your two test cases, describe how your test works, describe the expected output, and if you need any new syscalls to support your test, describe them.

test1(test-buffer-effect):

Syscall to add:
syscall_clear (flushes and removes all cache_block from buffer cache LRU list)
syscall_hit_reset (set hit_cnt to 0)
syscall_get_hits (return hit_cnt)

Variable to add in cache.c: unsigned int hit_cnt.

To test buffer, call syscall_clear and syscall_hit_reset, then open and read a large file to user buffer. Close the file and call syscall_get_hits. Repeat the following without calling syscall_clear.

Compare the hits. If hits of the second pass is greater than that of the first pass, return true; false otherwise.

test2(test-sync-write):

Syscall to add: syscall_dump_buffer (copies all elements of LRU list to a pointer to user level buffer)

To test buffer, call syscall_clear defined above, create an empty file and create 20 threads writing the same string to the file. Then close the file and sleep for 5 seconds to make sure data is flushed to disk. Call syscall_dump_buffer and save the buffer. Traverse the list and save the block_index of current cache_block as it advances. Check that each cache_block has a unique block_index by comparing the current block_index with saved index, and that all cache_block has their bool dirty set to false. Return true if above conditions are satisfied; false otherwise.