Section 11: File System Details and Reliability

William Liu

April 13, 2015

Contents

1 Warmup

1.1 Conceptual Questions

For each of the ACID properties, discuss the implications of a transaction system lacking that property.

Atomicity - data left in intermediate state
Consistency - data left in invalid state
Isolation - concurrent transactions may interfere with each other
Durability - crashes may destroy committed transactions

Suppose that we wish to create a reliable storage system. How can we ensure reliability even if part of the system temporarily crashes? How about if part of the system permanently crashes?

Use transactions and redundant systems.

Assume your Pintos file system needed to update a multiple sector long data structure on disk. Describe how you might arrange to detect if your system failed in the middle of writing the data structure.

A simple way to detect that the system failed in the middle of updating a multiple sector data structure would be to embedded a robust checksum of the data structure that is updated on every write. Should we detect that the recorded checksum doesnt match the current checksum of the data then the most likely cause is a crash during an update.

Describe how you might build your system to recover from such a failure

In order to recover from such a failure, you need some kind of replication. Simply have two copies of the data structure that you alternate between would work. After a crash you simply choose the newest copy with a correct checksum. It is also possible to use a log and write-ahead logging. By writing to the log first before writing to the data structure you can always roll forward after a crash.

Discuss briefly the different kinds of conflicts that may occur in concurrently executed transactions, in terms of reads and writes.

write-read – reading uncommitted data.
read-write – unrepeatable reads.
write-write – overwriting data.

2 Vocabulary

- **Unix-like File System** -

  Unix-like File System is a file system used by many Unix and Unix-like operating systems, although there may be some minor changes over different operating systems.

  Basically, disk space is divided into five parts

  Boot Sector

  Super Block

  Free Block Bitmap: This part indicates which block is reserved and which is not. Each bit represents one block. If a bit in the Free Block Bitmap is set to 1, it indicates the corresponding block is free; otherwise the corresponding block has been allocated. Free Block Bitmap is much more efficient than File Allocated Table since one bit is usedto tell whether a block is allocated or not. This feature allows Unix-like file system to scale up to large disk space without wasting too much memory. Assume each block has size of 8 KiB. A 1 TiB disk contains 2^{27} blocks. So the Free Block Bitmap only takes 16 MiB space, which is 1/64000 of total disk space. Meanwhile, with FAT, it would take 1 GiB to store File Allocation Table (1/1000 of disk space).

  Inode Table: Inode Table stores all the inodes.

  Data Blocks

- **Inodes** -

  Inode is the data structure that describes the meta data of files (regular files and directories). One inode represents one file or one directory. Inode is composed of several fields such as Ownership, Size, Last Modify Time, Mode, Link Count and Data Block pointers (which point to the data blocks that store the content of the file).

  Each inode has 12 direct block pointers (different file system may have different number of direct block pointers). Every direct block pointer directly points to a data block. The maximun size of a file using only direct block pointers can be as big as 80 KiB.

  To extend file size, inode introduces the concept of indirect block pointer. Indirect block pointer does not point to file content; instead, it points to the data blocks which store direct block pointers. An inode has one singly, one doubly indirect, and one triply indirect block pointer.

- **Transaction** - A transaction is an indivisible set of data operations that must either succeed or fail as a single unit, that takes data from one valid state to another valid state.

  Commonly, reliable transactions are said to satisfy the ACID properties.

- **ACID** - An acronym standing for the four key properties of a reliable transaction.

  Atomicity - the transaction must either occur in its entirety, or not at all.

  Consistency - transactions must take data from one consistent state to another, and cannot compromise data integrity or leave data in an intermediate state.

  Isolation - concurrent transactions should not interfere with each other; it should appear as if all transactions are serialized.

  Durability - the effect of a committed transaction should persist despite crashes.

- **Idempotent** - Idempotency is the property on operations that defines the ability of an operation to be repeated without effect beyond the first occurrence. If an operation can be repeated multiple times, but only the first such operation changes the outcome, the operation is said to be idempotent.

- **Logging file system** - A logging file system is a file system in which all modifications are done via transactions. Instead of modifying the disk directly, intended changes are written to an append-only log. Once the transaction is committed, it becomes safe to copy those changes onto disk, since in the event of a crash, transactions committed to the log can safely be re-copied onto disk.
- **2PL** - Two phase locking is a transaction protocol which is used to ensure atomic and isolated commitment of concurrent transactions. Each transaction must acquire a shared lock for any data it wishes to read, and an exclusive lock for any data it wishes to write. Each transaction is broken into two distinct phases: the growing phase, in which it acquires the lock as it needs them, and the shrinking phase, in which it begins releasing its lock. A transaction may not acquire any locks after releasing a single lock.

  One key variant of 2PL is strict 2PL, in which all locks must be released at the very end of the transaction.

  2PL guarantees that the dependency graph of transactions is acyclic, and that conflicting transactions will be serialized in some order.

3 Problems

3.1 Unix-like File System Details

3.1.1 inodes

For the following questions, assume a FFS inode with 12 direct block pointers, one singly indirect block pointer, one doubly indirect block pointer, and one triply indirect block pointer. Additionally, assume a 4 KiB block size, and 32-bit block pointers.

Consider the `inode_disk` struct in Pintos, which only allows a file to be allocated contiguously on the disk. Create one or more structs to represent an on-disk FFS inode, which allows for direct block pointers as well as indirect block pointers, and does not require a file to have its data allocated contiguously.

```
typedef uint32_t block_sector_t;


/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
```

```
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    block_sector_t direct[12];
    block_sector_t indirect[3];
    unsigned magic;                     /* Magic number. */
    uint32_t unused[110];               /* Not used. */
  };
```

Consider a random access to the 1200th data block of a file. Supposing that the inode is already in memory, how many disk accesses does it take to read the contents of that data block using Pintos's inode structure? What about using FFS's inode structure?

Each indirect block can hold up to 1024 block pointers, so the 1200th block is doubly indirect, and requires three accesses.

Under Pintos' inode system, file allocation is contiguous, so only one access is required.

Fill out the following struct to represent an in-memory indirect block, then fill out the function in order to return the `block_sector_t` corresponding to the `n`th byte inside that indirect block. Suppose that you have a function `read_indirect_block(block_pointer_t block)` which reads an indirect block from the disk into an `indirect_block_t`. Assume that the file is large enough to contain the desired byte of the file, and ignore triply indirect pointers for this problem. (Bonus: Figure out why triply indirect pointers need to be ignored in this problem.)

```
typedef struct indirect_block {
    block_sector_t block; // block_sector_t of this indirect block
    uint32_t indirection_level;

    _________________ block_pointers[________];
} indirect_block_t

block_sector_t sector_of_byte(indirect_block_t* iblock, uint32_t byte_number) {

    if (iblock->indirection_level > 1) {
        int index = byte_number >> (__________________________________);
        indirect_block_t next_iblock = read_indirect_block(iblock->block_pointers[index]);
        return sector_of_byte(_______________________________);
    }
    return iblock->block_pointers[byte_number >> 12];
}
```

```
typedef struct indirect_block {
    block_sector_t block; // block_sector_t of this indirect block
    uint32_t indirection_level;

    block_sector_t block_pointers[1024];
} indirect_block_t

block_sector_t sector_of_byte(indirect_block_t* iblock, uint32_t byte_number) {
```

```
    if (iblock->indirection_level > 1) {
        index = byte_number >> (10 * iblock->indirection_level + 2);
        next_iblock = read_indirect_block(iblock->block_pointers[index]);
        return sector_of_byte(&next_iblock, byte_number % (1 << (10*iblock->indirection_level + 2));
    }
    return iblock->block_pointers[byte_number >> 12];
}
```

3.1.2 Free Block Bitmaps

Consider a free block bitmap given as an integer array. Fill in the blanks of `is_set` to check if the bit corresponding to the `i`th block is set, assuming that the input is valid, in-bounds, and 4 bytes.

```
int bitmap[1024];

int is_set(int index) {
    int offset = index % 32;
    return (bitmap[index / 32] ____________________________);
}
```

```
int bitmap[1024];

int is_set(int index) {
    int offset = index % 32;
    return (bitmap[index / 32] << (31 - offset)) >> 31;
}
```

3.2 Reliability

Consider the following approach to a transactional key-value store which only stores integers, and each transaction consists of a single arithmetic operation which updates the value associated with a single key. Each transaction consists of the following steps.

```
Read T[i];
Write "T[i] = T[i] + 5" to log
Write "commit T[i]" to log
Commit
Copy changes to disk
Clear log
```

To recover from any potential crashes, the system iterates through the entire log and replicates any committed transactions that are still in the log. Identity a key problem with this system.

This system is not idempotent, since if the system crashes after committing changes to disk, but before clearing the log, it will repeat some arithmetic operation, with the final result being the

output of the composition of two identical arithmetic operations.

Consider the following approach to file creation in FFS. For each of the ACID properties, discuss whether the given approach satisfies that property. Assuming that we cannot afford to scan the whole disk to check for inconsistencies, which of the ACID properties does using a log provide?

```
Find free block
Find free inode
Write used bit to bitmask (in log)
Write inode to inode table in log
Write data to log
Write directory entry for inode to log
Commit to disk
Copy log changes to disk
```

Remember that atomicity is define as follows: if an action is interrupted part way through, its as if it never happened.

Atomicity - By recording all redo and undo information, and making sure to write to the log before the corresponding data page gets to disk, we ensure atomicity of transactions.

Consistency - Cannot solve with just logging.

Isolation - Need some separate synchronization mechanism in order to ensure isolation. This is vulnerable to race conditions without the use of a global lock for finding free resources.

Durability - We write all log records before commits, so if we assume that writes to the log are small and crashing won't interrupt a write to the log, we eliminate distinction between memory and disk, as if we crash, we can always retrace our steps. Since log entry operations are idempotent, executing them again will guarantee that we reach a consistent state prior to the failure.

The logging system provides for atomicity and consistency – it is now possible for a file creation to partially complete, leaving data in an inconsistent state.

For the following schedule of transactions operating on data objects A and B, identify any pairwise conflicts between operations from different transactions.

This information could be used to construct a graph where each transaction is represented by a node, with edges representing conflicting operations between transactions on a particular data object.

T1	R(A)		W(A)					
T2		R(A)						
T3					W(A)		W(B)	
T4				W(A)		W(A)		W(B)

Any write-write, read-write, or write-read conflict on the same data object.

3.3 mmap

Recall the concept of memory mapped files, which allows a block of virtual memory to have a direct correlation with some portion of a file on disk.

Suppose that it takes twice as long to read a 4000 bytes than 2000 bytes from a memory mapped file, but it takes ten times longer to read 8000 bytes. What are some likely causes for the non-linear read time?

Page fault

Discuss the advantages and drawbacks of memory mapped file accesses compared to traditional disk accesses for small random file reads and writes to many files of varying size.

Memory mapped files allow for in-place updates without the need to seek into the file to the appropriate location.

However, since this access pattern accesses data in many different files, it will incur overhead for paging the files in and out. This may or may not be worse than the overhead of traditional I/O, which involves overhead in the syscall framework and copying memory.

Various operating systems have offered memory-mapped access to files, either as the fundamental way to access files or as an additional feature. Consider the Unix mmap system call that has the form (somewhat simplified):

void *mmap(void *addr, size_t len, int prot, int flags, int filedes, off_t offset);

where data is being mapped from the currently open file filedes starting at the position in the file described by offset; len is the size of the part of the file that is being mapped into the process' address space; prot describes whether the mapped part of the is readable or writable. The return value, addr, is the address in the process' address space of the mapped file region.

When the process subsequently references a memory location that has been mapped from the file for the first time, what operations happen in the operating system? What happens upon subsequent accesses?

First access leads to Page Fault. OS commits the page, i.e maps the page to a physical page. Subsequent accesses, the page of the file is there in memory.

Assume that you have a program that will read sequentially through a very large file, computing some summary operation on all the bytes in the file. Compare the efficiency of performing this task when you are using conventional read system calls versus using mmap.

Sequential reading with conventional read system calls may be worse because it may read the file in larger number of chunks, and incurs multiple copy overheads for all reads.

With mmap:

1. there's less copying. In using conventional read system calls, often OS and library level routines pass data through one or more buffers before it reaches an application-specified buffer, the applicaton then dynamically allocates storage, then copies from the I/O buffer to that storage so the data's usable after the file reading completes.

2.memory mapping can simplify the application's parsing job by letting the application treat the entire file content as accessible, rather than worrying about when to read another buffer full

3.Also using memory maping you typically use less system calls. This could result in significant speedup of random access reads (i.e. lseek before every read).