

File Systems: Consistency Issues

1

File Systems: Consistency Issues

- ◆ File systems maintain many data structures
 - Free list/bit vector
 - Directories
 - File headers and inode structures
 - Data blocks
- ◆ All data structures are cached for better performance
 - Works great for read operations
 - ... but what about writes?
 - ◊ If modified data is in cache, and the system crashes → all modified data can be lost
 - ◊ If data is written in wrong order, data structure invariants might be violated (this is very bad, as data or file system might not be consistent)
 - Solutions:
 - ◊ Write-through caches: Write changes synchronously → consistency at the expense of poor performance
 - ◊ Write-back caches: Delayed writes → higher performance but the risk of losing data

2

What about Multiple Updates?

- ◆ Several file system operations update multiple data structures
- ◆ Examples:
 - Move a file between directories
 - ◊ Delete file from old directory
 - ◊ Add file to new directory
 - Create a new file
 - ◊ Allocate space on disk for file header and data
 - ◊ Write new header to disk
 - ◊ Add new file to a directory
- ◆ What if the system crashes in the middle?
 - Even with write-through, we have a problem!
- ◆ The consistency problem: The state of memory+disk might not be the same as just disk. Worse, just disk (without memory) might be inconsistent.

3

Which is a metadata consistency problem?

- ◆ A. Null double indirect pointer
- ◆ B. File created before a crash is missing
- ◆ C. Free block bitmap contains a file data block that is pointed to by an inode
- ◆ D. Directory contains corrupt file name

4

Consistency: Unix Approach

- ◆ Meta-data consistency
 - Synchronous write-through for meta-data
 - Multiple updates are performed in a specific order
 - When crash occurs:
 - ◊ Run "fsck" to scan entire disk for consistency
 - ◊ Check for "in progress" operations and fix up problems
 - ◊ Example: file created but not in any directory → delete file; block allocated but not reflected in the bit map → update bit map
 - Issues:
 - ◊ Poor performance (due to synchronous writes)
 - ◊ Slow recovery from crashes

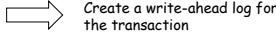
5

Consistency: Unix Approach (Cont'd.)

- ◆ Data consistency
 - Asynchronous write-back for user data
 - ◊ Write-back forced after fixed time intervals (e.g., 30 sec.)
 - ◊ Can lose data written within time interval
 - Maintain new version of data in temporary files; replace older version only when user commits
- ◆ What if we want multiple file operations to occur as a unit?
 - Example: Transfer money from one account to another → need to update two account files as a unit
 - Solution: Transactions

6

Transactions	
<ul style="list-style-type: none"> ◆ Group actions together such that they are <ul style="list-style-type: none"> ➢ Atomic: either happens or does not ➢ Consistent: maintain system invariants ➢ Isolated (or serializable): transactions appear to happen one after another. Don't see another tx in progress. ➢ Durable: once completed, effects are persistent ◆ Critical sections are atomic, consistent and isolated, but not durable ◆ Two more concepts: <ul style="list-style-type: none"> ➢ Commit: when transaction is completed ➢ Rollback: recover from an uncommitted transaction 	7

Implementing Transactions	
<ul style="list-style-type: none"> ◆ Key idea: <ul style="list-style-type: none"> ➢ Turn multiple disk updates into a single disk write! ◆ Example: <p>Begin Transaction $x = x + 1$ $y = y - 1$ Commit</p> <div style="display: inline-block; vertical-align: middle; text-align: center;">  </div> ◆ Sequence of steps: <ul style="list-style-type: none"> ➢ Write an entry in the write-ahead log containing old and new values of x and y, transaction ID, and commit ➢ Write x to disk ➢ Write y to disk ➢ Reclaim space on the log ◆ In the event of a crash, either "undo" or "redo" transaction 	8

Transactions in File Systems	
<ul style="list-style-type: none"> ◆ Write-ahead logging → journaling file system <ul style="list-style-type: none"> ➢ Write all file system changes (e.g., update directory, allocate blocks, etc.) in a transaction log ➢ "Create file", "Delete file", "Move file" --- are transactions ◆ Eliminates the need to "fsck" after a crash ◆ In the event of a crash <ul style="list-style-type: none"> ➢ Read log ➢ If log is not committed, ignore the log ➢ If log is committed, apply all changes to disk ◆ Advantages: <ul style="list-style-type: none"> ➢ Reliability ➢ Group commit for write-back, also written as log ◆ Disadvantage: <ul style="list-style-type: none"> ➢ All data is written twice!! (often, only log meta-data) 	9

<p>Where on the disk would you put the journal for a journaling file system?</p> <ol style="list-style-type: none"> 1. Anywhere 2. Outer rim 3. Inner rim 4. Middle 5. Wherever the inodes are 	10

Transactions in File Systems: A more complete way	
<ul style="list-style-type: none"> ◆ Log-structured file systems <ul style="list-style-type: none"> ➢ Write data only once by having the log be the only copy of data and meta-data on disk ◆ Challenge: <ul style="list-style-type: none"> ➢ How do we find data and meta-data in log? <ul style="list-style-type: none"> ◦ Data blocks → no problem due to index blocks ◦ Meta-data blocks → need to maintain an index of meta-data blocks also! This should fit in memory. ◆ Benefits: <ul style="list-style-type: none"> ➢ All writes are sequential; improvement in write performance is important (why?) ◆ Disadvantage: <ul style="list-style-type: none"> ➢ Requires garbage collection from logs (segment cleaning) 	11

File System: Putting it All Together	
<ul style="list-style-type: none"> ◆ Kernel data structures: file open table <ul style="list-style-type: none"> ➢ Open("path") → put a pointer to the file in FD table; return index ➢ Close(fd) → drop the entry from the FD table ➢ Read(fd, buffer, length) and Write(fd, buffer, length) → refer to the open files using the file descriptor ◆ What do you need to support read/write? <ul style="list-style-type: none"> ➢ Inode number (i.e., a pointer to the file header) ➢ Per-open-file data (e.g., file position, ...) 	12

Putting It All Together (Cont' d.)

- ◆ Read with caching:

```
ReadDiskCache(blocknum, buffer) {
    ptr = cache.get(blocknum) // see if the block is in cache
    if (ptr)
        Copy blksize bytes from the ptr to user buffer
    else {
        newOSBuf = malloc(blksize);
        ReadDisk(blocknum, newOSBuf);
        cache.insert(blockNum, newOSBuf);
        Copy blksize bytes from the newOSBuf to user buffer
    }
}
```
- ◆ Simple but require block copy on every read
- ◆ Eliminate copy overhead with mmap.
 - Map open file into a region of the virtual address space of a process
 - Access file content using load/store
 - If content not in memory, page fault

11

Putting It All Together (Cont' d.)

- ◆ Eliminate copy overhead with mmap.
 - `mmap(ptr, size, protection, flags, file descriptor, offset)`
 - `munmap(ptr, length)`

Virtual address space



Refers to contents of mapped file

- ◆ `void* ptr = mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0);`
- ◆ `int foo = *(int*)ptr;`
 - ◆ `foo` contains first 4 bytes of the file referred to by file descriptor 3.

14