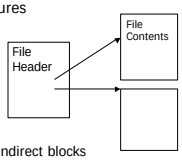
File System Reliability

Arvind Krishnamurthy
Spring 2004

File Systems

- File systems have lots of data structures
  - File headers (or i-nodes)
  - File headers point to file data blocks



  - For large files, file headers point to Indirect blocks
  - Directories are special files: a table mapping names to i-node numbers
  - Bitmap for free blocks
  - Bitmap for i-nodes (to remember which i-nodes have been allocated)
- Question: what kind of inconsistencies could occur in file systems?

File System Reliability

- For performance, all must be cached!
  This is OK for reads but what about writes?
- Options for writing data:
  Write-through: write change immediately to disk
  Problem: slow! Have to wait for write to complete before you go on

  Write-back: delay writing modified data back to disk (for example, until replaced)
  Problem: can lose data on a crash!

Multiple updates

- If multiple updates needed to perform some operations, crash can occur between them!
  - Moving a file between directories:
    - Delete file from old directory
    - Add file to new directory
  - Create new file
    - Allocate space on disk for header, data
    - Write new header to disk
    - Add the new file to directory

What if there is a crash in the middle? Even with write-through it can still have problems

Unix approach (ad-hoc)

- Meta-data: needed to keep file system logically consistent:
  - File headers, Directories, Indirect blocks
  - Bitmap, I-node bitmap
- Data: user bytes
- Meta-data: synchronous writes, data: asynchronous writes
  - If multiple updates to meta-data is needed, Unix does them in specific order
  - If it crashes, run the special program "fsck" which scans the entire disk for internal consistency to check for "in progress" operations and then fixes up anything in progress.

Ordering of operations

Let's say you want to extend the file by one block. Operations are:

- find a free block
- write block bit-map
- write i-node with pointer to free block and new file size
- write data

In what order should you perform the above operations?

Ordering of operations (contd.)

- Let's say you want to create an empty file in a directory:
  - find a free i-node
  - write i-node map
  - write i-node
  - write directory

What order should the above operations be performed?

User data consistency

- For user data, Unix uses "write back" --- forced to disk every 30 seconds (or user can call "sync" to force to disk immediately).

No guarantee blocks are written to disk in any order.

Sometimes meta-data consistency is good enough

How should vi save changes to a file to disk?

Wrong: delete old version, create new version

Correct: Write new version in temp file
Move old version to another temp file
Move new version into real file
Unlink old version

If crash, look at temp area; if any files out there, send email to user that there might be a problem.

Transaction concept

- **Transactions:** group actions together so that they are
  - **Atomic**: either happens or it does not (no partial operations)
  - **Serializable**: transactions appear to happen one after the other
  - **Durable**: once it happens, stays happened

Critical sections are atomic and serializable, but not durable

Need two more terms to describe transactions:
Commit --- when transaction is done (durable)
Rollback --- if failure during a transaction (means it didn't happen at all)

- Do a set of operations tentatively. If you get to commit, ok. Otherwise, roll back the operations as if the transaction never happened.

Transaction implementation

- Key idea: fix problem of how you make multiple updates to disk, by turning multiple updates into a single disk write
- Example: money transfer from account x to account y:

```
Begin transaction
  x = x + 1
  y = y – 1
Commit
```

- Keep log on disk of all changes in transaction.
  - A log is like a journal, never erased, record of everything you've done
  - Once both changes are on log, transaction is committed.
  - Then can "write behind" changes to disk --- if crash after commit, replay log to make sure updates get to disk

Transaction implementation (cont'd)

Memory cache

X: 0
Y: 2

Disk

Sequence of steps to execute transaction:
1. Write new value of X to log
2. Write new value of Y to log
3. Write commit
4. Write x to disk
5. Write y to disk
6. Reclaim space on log

X=1	Y=1	commit

write-ahead log (on disk or tape or non-volatile RAM)

Transaction implementation (cont'd)

X=1	Y=1	commit

1. Write new value of X to log
2. Write new value of Y to log
3. Write commit
4. Write x to disk
5. Write y to disk
6. Reclaim space on log

- **What if we crash after 1?**
  - No commit, nothing on disk, so just ignore changes
- **What if we crash after 2?** Ditto
- **What if we crash after 3 before 4 or 5?**
  - Commit written to log, so replay those changes back to disk
- **What if we crash while we are writing "commit" ?**
  - As with concurrency, we need some primitive atomic operation or else can't build anything. (e.g., writing a single sector on disk is atomic!)

Transaction implementation (cont'd)

- Can we write X back to disk before commit ?
- Yes: Keep an "undo log"
  - Save old value along with new value
  - If transaction does not commit, "undo change"

Transactions under multiple threads

- What if two threads run same transaction at same time? Use locks.

```
Begin transaction
  Lock x, y
    x = x + 1
    y = y – 1
  Unlock x, y
Commit
```

- Is the above approach correct?

Two-phase locking

- Don't allow "unlock" before commit.
- First phase: only allowed to acquire locks (this avoids deadlock concerns).
- Second phase: all unlocks happen at commit
- Thread B can't see any of A's changes, until A commits and releases locks. This provides serializability.

Transactions in file systems

- Write-ahead logging
  - Almost all file systems built after 1985 (NT, Solaris) uses "write ahead logging"
  - Write all changes in a transaction to log (update directory, allocate block, etc.) before sending any changes to disk.
  - Example transactions: "Create file", "Delete file", "Move file"

  This eliminates any need for file system check (fsck) after crash

  If crash, read log:
  - If log is not complete, no change!
  - If log is completely written, apply all changes to disk
  - If log is zero, then all updates have gotten to disk.
- Pros: reliability Cons: all data written twice

Announcements

- Remaining topics for the semester:
  - Networks
  - Security and authentication
  - Distributed file systems