ext3 modes

disks usually have write caches and re-order writes, for performance

- sometimes hard to turn off (the disk lies)
- people often leave re-ordering enabled for speed, out of ignorance

bad news if disk writes commit block before preceding stuff

- then recovery replays "descriptors" with random block #s! and writes them with random content!
- Solution: have a checksum in the commit record. Ext3 does not support this though.

ordered vs journaled

- journaling file content is slow, every data block written twice
- perhaps not needed to keep FS internally consistent
- can we just lazily write file content blocks?
- no: if metadata updated first, crash may leave dangling pointers from file inodes
- ext3 ordered mode:
  - write content block to disk before commiting inode w/ new block # thus won't see stale data if there's a crash
- most people use ext3 ordered mode

ext3 very successful

- but: no checksum -- ext4

What are the persistence guarantees for the user?

None, except that her data will be persistent after 30 seconds. But can use `fsync()` to ensure that the data has become persistent. How will fsync() be implemented in ext3? enough to close commit the current transaction? or do we need to apply the log to the FS tree also?

Databases provide persistence guarantees, and so make extensive use of fsync() if implemented in userspace. Makes filesystem performance much worse. Better to expose raw disk to database than layering it on top of a file system.

Protection and Security

Protection involves mechanisms that prevent accidental or intentional misuse of the system. There are three aspects to a protection mechanism:

- Authentication: identify a responsible party (*principal*) behind each action.
- Authorization: determine which principals are allowed to perform which actions.
- Access enforcement: Combine authentication and authorization to control access.

A tiny flaw in any of these areas can compromise the entire protection mechanism.

Authentication

- Typically done with passwords:
  - A secret piece of information used to establish user identity
  - Must not be stored in directly-readable form: one-way transformations
  - Passwords should be relatively long and obscure, to prevent brute-force attacks.
- Alternate form of authentication: badge or key
  - Does not have to be kept secret
  - Should not be forgable or copyable
  - Can be stolen, but owner will know if it is
  - Paradox: key must be cheap to make, hard to duplicate
- Once authentication is complete, the identity of the principal must be protected from tampering
- Once you log in, your user ID is associated with every process executed under that login: each process inherits user ID of its parent

Authorization

Goal: determine which principals can perform which operations on which objects

- Logically, authorization information represented as an *access matrix*
  - One row per principal
  - One column per object
  - Each entry indicates what that principal can do to that object
- In practice, a full access matrix would be too bulky, so it gets stored in one of two compressed ways: access control lists and capabilities.

- *Access Control Lists* (ACLs): organize by columns.
    - With each object, store information about which users are allowed to perform which operations
    - Most general form: list of (user, privilege) pairs
    - For simplicity, users can be organized into groups, with a single ACL entry for an entire group
    - ACLs can be very general (Windows) or simplified (Unix)
    - UNIX: 9 bits per file:
        - user/owner, group, others
        - read, write, execute for each of the above
        - "root" has all permissions for everything
        - additional setuid bit to allow any user to execute file with owner privileges, i.e., you can execute "os-submit-lab" with your instructor's privileges to allow copying of your files to his directory
- *Capabilities*: organize by rows.
    - With each user, indicate which objects may be accessed, and in what ways.
    - Store a list of (object, privilege) pairs with each user. This is called a capability list.
    - Typically, capabilities also act as names for objects: can't even name objects not referred to in your capability list.
    - As if there was no root directory in UNIX and no "..". Another example is file descriptors or page tables for processes (here processes are the principals).
- Systems based on ACLs encourage visibility of objects: shared public namespace
- Capability systems discourage visibility; namespaces are private by default
- Capabilities have been used in experimental systems attempting to be very secure. However they have proven to be clumsy to use (painful to share things), so they have mostly fallen out of favour for managing objects such as files.

Access Enforcement

- Some part of the system must be responsible for enforcing access controls and protecting authentication and authorization info
- This portion of the system has total power, so it should be as simple and small as possible.
- Examples: OS kernel, tamper-proof code to check cryptographic keys, etc.

Common attacks involve *Trojan Horses*, where a useful program is tricked or modified into doing something that it is not supposed to do, but has the privilege of doing. Example: if there is a bug in the "submit-lab" script, you can trick it to execute anything with the instructor privileges.