FS Facilities

Naming, APIs, and Caching

OS Lecture 17

UdS/TUKL WS 2015

Naming Files

Recall: `inodes`

What is an `inode`?

- the data structure of a filesystem representing a *byte stream* (= a file) on stable storage

How are `inodes` addressed?

- by index in a filesystem-specific table
- low-level implementation fact
- We need to map **human-readable names** to `inodes`.

Mapping Names to Files

`/home/bbb/notes.txt → [inode A]`

`../etc/my-server.conf → [inode X]`

`/srv/production/etc/my-server.conf → [inode B]`

`/srv/testing/etc/my-server.conf → [inode C]`

Historic Developments

Mapping: *human-readable name* → *inode*

The beginning: a single, flat table
→ one lookup table for the whole system

Towards directories: per-user lookup tables
→ separate, flat namespace for each user

Proper directories: Multics directory tree
→ popularized by UNIX

Practical Challenges

1. running multiple instances of the same application
   → absolute and relative filenames
2. multiple names for the same file
   → hardlinks and symlinks
3. multiple disks
   → mount points
4. multiple filesystem types
   → virtual file system (VFS) layer

Absolute vs. Relative Names

Absolute name: e.g., `/home/bbb/notes.txt`

- unambiguously identifies a file
- start name resolution at **filesystem root**
  → '/' is the root directory, traditionally inode 2

Relative name: e.g., `../etc/my-server.conf`

- identifies a file in **context** of calling process
- start name resolution at *current working directory*
  → `..` means *parent directory* (= go up one level)

Current Working Directory (CWD)

- used to resolve relative filenames
- POSIX: one CWD per process (*not* per thread)
- inherited from parent at fork
    - `cd` in shell = “change directory” (= set CWD)
    - processes launched from shell “start running in the current directory”

`chroot()`

Change root — change the meaning of `/`.

- Can be used to restrict a process to a subtree of the filesystem.
- Files that are not children of the new root become effectively “invisible”.
- **Example**: `chroot(“/tmp/sandbox”)`
  - ensures that the call `open(“/foo/bar”, …)` is effectively interpreted as `open(“/tmp/sandbox/foo/bar”, …)`
- Note: by itself, this is *not* a security feature.

Implementation (UFS)

How are directories stored on disk?

- Just as regular files!
- A directory is just a file that contains a table of *name* → *inode* mappings.
- Each “directory file” consists of *chunks*, where each chunk is small enough (512 bytes) to be written with a single I/O operation (→ *atomicity*).
- Each chunk contains variable-size file records.

Unix FS Directory Contents Records

```
#define MAXNAMLEN   255
struct  direct {
    u_int32_t d_ino;        /* inode number of entry */
    u_int16_t d_reclen;     /* length of this record */
    u_int8_t  d_type;       /* file type */
    u_int8_t  d_namlen;     /* length of name */
    char      d_name[MAXNAMLEN + 1];
};
```

On disk, `d_name` is not actually 256 bytes long, but variably sized to a multiple of 4 bytes to hold the name plus any trailing free space.

Record and Chunk Invariants

1. The sum of all the lengths of all `struct direct` records in a chunk always adds up to the chunk’s size.
   - Any trailing free space after a record is added to the record’s `d_reclen`.
2. No `struct direct` record crosses any chunk boundary (→ *atomicity*).
3. At most one chunk is modified as part of a single operation (→ *atomicity*).

Name Lookup

Lookup is a very common operation and must be fast.

- *Sequentially scan* all chunks. For each record,
  - first compare length of name (`d_namelen`),
  - then byte-wise compare `d_name` field.
- Important **optimization**: start next search where last finished. Why? (Hint: think of `ls -l`)
- What about directories with large numbers of entries?

To delete a directory entry

1. *Sequentially scan* all chunks to find a `struct direct` record with matching name (error if not found)
   - let `to_del` denote the to-be-deleted record
2. If `to_del` is *not* the first in the chunk, add the length of `to_del` to the predecessor
   - let `pred` denote the predecessor of `to_del`:

     ```
     pred->d_reclen += to_del->d_reclen;
     ```
3. Otherwise, set `to_del->d_ino` to 0 (i.e., a special value indicating "invalid record").
4. Write chunk containing `to_del` to disk.

To create a new directory entry

1. *Sequentially scan* all chunks to see if name is already taken (return error if so)
2. Keep track of **total free space** in each chunk. Note: free space may be *fragmented*.
3. Find first chunk into which new `struct direct` will fit (or append a new chunk).
4. If necessary, rewrite chunk to **coalesce free space**.
5. Write new entry into free space (setting `d_reclen` to occupy the free space) and write chunk to disk.

Path resolution & lookup

How to resolve a path such as `/a/b/c`?

1. Load root directory (`/`) from disk.
2. Lookup directory named “`a`” in root directory to find inode of `a`.
3. Load `a` directory from disk.
4. Lookup directory named “`b`” in `a` directory to find inode of `b`.
5. Load `b` directory from disk.
6. Lookup entry named “`c`” in `b` directory to find inode of `c`.
7. Return `c`.

Path resolution & lookup

General approach:

1. Split pathname into list of *path components*
2. set `cursor` to *root directory* if first component is `/`; otherwise start at *CWD*.
3. While list of path components is not empty:
   - remove `head` (= first element) from list
   - `cursor` ← lookup `head` in directory represented by `cursor`
   - if not found return error
4. return `cursor`

Names ≠ Files!

- A directory entry *links* a name to an inode
- The directory entry itself is *not* the file, just *a name* of the file. Rather, inodes represent files (i.e, *are* files).
- Multiple directory entries can link to the *same* file.
  → A single file can have many names.
- The (single) inode contains all relevant per-file metadata (permission bits, access times, creation times, etc.)
- inodes are **reference-counted**: the number of times it is referred to in any directory
- A file is "deleted" when the reference count drops to zero.

Hard Links

- A *hard link* is just a directory entry as discussed so far: association of a name with an inode.
- A hard link prevents a file from being deleted (i.e., it counts towards the inode's reference count).
- Regular files may have multiple incoming links (many names for the same byte stream).
- Directories may not have multiple incoming hard links. *Why*?

Hard Links — Example (1/2)

```
$ echo -n “Hello” > a.txt
$ ln a.txt b.txt # creating a hard link
$ cp a.txt c.txt # create a **copy**
```

Observe: `a.txt` and `b.txt` refer to the same inode, but `c.txt` does not.

```
$ ls -i a.txt b.txt c.txt # print inode
9239376 a.txt   9239376 b.txt   9240275 c.txt
```

Hard Links — Example (2/2)

Observe: `a.txt` and `b.txt` are equivalent.

```
$ echo “ World” >> b.txt
$ cat a.txt
Hello World
$ rm a.txt
$ cat b.txt
Hello World
$ cat c.txt
Hello
```

Soft (or Symbolic) Links, aka Symlinks

- A *soft link* is a file that *redirects* to another filename: an association of two names.
- In contrast to a hard link, a *soft link* does *not* affect the reference count of the target.
- In fact, target may not even exist.
- The target may reside on another filesystem and may be a directory.

Lookup with Symlinks

- On disk, symlinks are simply short files that contain a pathname.
- At each step during pathname resolution, check if `cursor` points to a symlink.
  - If so, read symlink and *prepend* contents to list of path components.
    → *What about cycles*?
- To deal with potential cycles, a finite number of symlinks is traversed by the lookup code before returning `ELOOP` error. → *Why not do the same for hard links*?

Symlink Example (1/2)

```
$ mkdir -p a/b/c/d/e/f/g/h/i/j/k/l/m/n
$ mkdir -p x/y/z
# Create a symlink named “shortcut” in x/y/z to “n”
$ (cd x/y/z; ln -s ../../../a/b/c/d/e/f/g/h/i/j/k/l/m/n shortcut)
$ echo “Hello” > a/b/c/d/e/f/g/h/i/j/k/l/m/n/msg.txt
$ cat x/y/z/shortcut/msg.txt
Hello
$ echo “there.” >> x/y/z/shortcut/msg.txt
$ cat a/b/c/d/e/f/g/h/i/j/k/l/m/n/msg.txt
Hello
there.
```

Observe: appears to work just like a hard link, but `x/y/z/shortcut/` points to a directory (impossible with hard links).

Symlink Example (2/2)

```
$ rm a/b/c/d/e/f/g/h/i/j/k/l/m/n/msg.txt
$ cat x/y/z/shortcut/msg.txt
cat: x/y/z/shortcut/msg.txt: No such file or directory
$ ls -l x/y/z
total 8
lrwxr-xr-x  1 bbb  wheel  36 Jan  2 22:10 shortcut
    -> ../../../a/b/c/d/e/f/g/h/i/j/k/l/m/n
```

Observe: symlink still exists, but now points to a non-existent target (unlike hard links).

Symlink: ELOOP Example

```
$ mkdir x
$ mkdir y
$ ln -s '../y/foo' x/foo
$ ln -s '../x/foo' y/foo
$ ls -l x/foo y/foo
[…] x/foo -> ../y/foo
[…] y/foo -> ../x/foo
$ cat x/foo
cat: x/foo: Too many levels of symbolic links
```

Observe: the mutually recursive symlinks exist in the filesystem as intended, but `open()` returns ELOOP error.

Multiple Disks (1/3)

Can filesystems span multiple disks?

- Traditionally, physical disks are managed independently: filesystems such as FFS/UFS, Ext2, XFS, JFS, HFS+ do not span across disks at the implementation level.
- Instead, a *merged view* of filesystems on multiple disks is provided by the kernel by ***mounting*** them as *subtrees* of the root filesystem.
- In Microsoft OSs, separate filesystems traditionally have separate roots (`C:`, `D:`, …).

Multiple Disks (2/3)

*With **logical volume mangement** (LVM), multiple smaller block devices can be made to appear as one large **virtual** device.*

- LVM inserts a *layer of indirection* between the filesystem and the actual physical block devices.
- LVM manages multiple disks, hiding them from the rest of the system. Instead, it presents a large, idealized, contiguous *volume* (= virtual disk) to the filesystem.
- This allows classic filesystems such as FFS/UFS, ext2, etc. to be used across multiple disks.
- LVM can also provide redundancy (RAID), on-the-fly encryption, etc.

Multiple Disks (3/3)

ZFS comes with its own LVM.

- ZFS is targeted at truly large, highly available filesystems, which inherently requires the use of multiple block devices.
- Instead of relying on an underlying LVM layer, ZFS itself has its own notion of *storage pools* that bundle multiple block devices into a single *virtual device*.
- ZFS also (optionally) provides compression, encryption, block de-duplication, replication, …

Using Multiple Filesystems

Despite LVM, OSs often use multiple separate filesystems (aka *partitions*, *slices*, or *volumes*). Why?

- simple way to use **multiple block devices**
- **isolation**: don't allow user directories (`/home`), log files (`/var/log`), or temporary files (`/tmp`) to fill up system partition (`/`); can also isolate **I/O bandwidth**.
- **specialization**: use an FS that's good for large files for archive directory (e.g., XFS), but FS that's good for many small files for user directories (e.g., ReiserFS).

Mount Points

The kernel provides a single hierarchical namespace despite separate physical filesystems.

- *Mounting* a filesystem means making it available as a *subdirectory* of another, already mounted filesystem.
  - Example: `mount /dev/sdb0 /mnt/usbstick`
  - makes filesystem on device `/dev/sdb0` (a device file) available as a subtree starting at `/mnt/usbstick`
- Any pre-existing files of the original FS below the *mount point* (= directory where a filesystem is mounted) are hidden by the newly mounted FS.

VFS: inodes vs. vnodes

The kernel must transparently deal with multiple filesystem types.

- Original UNIX filesystem implementation (e.g., name lookup) dealt directly with *inodes*.
- But inodes are low-level, highly FS-specific detail.
- Should each filesystem re-implement name lookup? No!
  → *Need a single, higher-level implementation: the VFS.*
- The *virtual filesystem* (VFS) layer operates on *vnodes*, an abstract interface that encapsulates FS-specific inodes. Lookup works as before, except that actual parsing of on-disk data is delegated FS-specific methods.

Everything is a file (1/3)

Almost anything can be exposed via the VFS…

- `procfs` — list processes and their properties as files under `/proc`
- `devfs` — represent physical devices as device files under `/dev`
- `fdesc` — represent open file descriptors of calling process as files (e.g., `/dev/fd/0` is an alias for `STDIN`. Try `cat /dev/fd/0`.)

Everything is a file (2/3)

- `tmpfs` (Linux) — RAM-based, temporary filesystem
- `debugfs` (Linux) — exposes kernel data structures for debugging
- `sysfs` (Linux) — exposes kernel configuration data and settings, and hardware details
- Plan 9 exposes networking via FS namespace → `/net/tcp/` and `/net/udp/`

Everything is a file (3/3)

Can delegate parts of the filesystem namespace to user-level processes (i.e., drivers in userspace)

- FUSE (**f**ile system in **use**rspace) provides easy way to implement custom filesystems
  → e.g., in Python!
  → sshfs, GMailFS, mysqlfs, WikipediaFS, …
- Similar functionality originally offered by *portals* in 4.4BSD.
- Standard way of implementing *any* filesystem on top of microkernels.

File system APIs

Read/Write Primitives

How to read from and write to files?

- **Explicitly**: `read()` and `write()` system calls, the classic, simple approach
- **Implicitly**: via memory-mapped files with `mmap()`, which can be a more efficient approach

`read()` Overview (1/2)

- Process opens file handle with `open()`:
  → VFS layer performs name lookup, associates *file descriptor* (in the process file descriptor table) with a *vnode*, which in turn abstracts a filesystem-specific *inode*
- Process allocates buffer space (in user space)
- Process issues `read()` system call with pointer to buffer
- VFS layer triggers *vnode*'s `read()` method (if not cached)

`read()` Overview (2/2)

- actual FS allocates buffer for block I/O (in kernel space)
- actual FS uses inode info to request block read from disk
- when I/O operation completes (interrupt), VFS copies data from block I/O buffer to process-provided buffer in user space
- `read()` system call returns
- `write()` works like `read()`, just in the reverse direction

`seek()` vs. absolute offset

- UNIX `read()`/`write()` work at implicit position of file
- makes sequential access easy, but “jumps” to other offset require explicit `seek()` call
- alternative: pass offset explicitly as argument to `pread()`/`pwrite()`, which do not modify implicit file pointer

Vectored I/O

`pwritev()` and `preadv()`

- useful if data is to be read to / to be written from *many small buffers* that are scattered throughout the user address space
- to reduce system call overhead, the process provides vector of `(buffer ptr, length)` descriptors
- the VFS layer fills/writes the buffers in sequence

Discussion

Explicit I/O is straightforward, but has some downsides.

- **Copy overhead**: data is explicitly copied between kernel buffer and user buffer.
- **System call overhead**: can become significant for frequent short reads/writes
- **Double paging**: user buffer may be paged out → two redundant copies of data on disk.

Memory-Mapped Files

In UNIX/POSIX, a file's contents can be directly mapped into user space with `mmap()`.

- After mapping, any page fault in mapped address range will be handled by reading **corresponding page** from file.
- To read from the file, the process simply loads from the mapped addresses.
- To write to the file, the process simply stores to the mapped addresses.
- Kernel will (eventually) flush dirtied pages back to disk (unless explicitly prevented from doing so).

Advantages of `mmap()`

- no double-paging
- no copying
- no system call overhead (after initial setup)
- if two or more processes map the same file
  → ***shared** memory segment*

Controlling Sharing

Sometimes, modifications should not be written to disk.

- `MAP_PRIVATE` — do *not* write modifications back to the file (copy-on-write semantics)
- `MAP_SHARED` — modifications are immediately visible to other processes
  - even if they use `read()`

Other `mmap()` Variants and Options

`mmap()` and `madvise()` allow fine-grained control

- `MAP_ANONYMOUS` (Linux) or `MAP_ANON` (BSD) — not backed by file, initialized to zero (e.g., used to implement `malloc()`)
- `MAP_HUGETLB` (Linux) — use large pages (fewer TLB entries)
- `MAP_LOCKED` (Linux) — do not page out
- `MAP_GROWSDOWN` (Linux) — used for stacks
- `MAP_POPULATE` (Linux) — pre-page (don’t wait for page faults)
- `MAP_NOSYNC` (FreeBSD) — don’t regularly write dirty pages to disk
- `VM_FLAGS_PURGABLE` (Mach, OS X) — volatile cache

File Locking

Concurrent access to shared files poses the risk of race conditions.

Example: one process updating a configuration file or system database, while another process is reading it.
How to synchronize?

- In case of `mmap()`, can place a semaphore in the file (= shared memory segment) itself. *Limitations*?
  (→ `MAP_HASSEMAPHORE` on *BSD)

- Ad-hoc solution: recall that creating a hard link (= name creation) is atomic: to "lock", create an empty *lock file*; to "unlock", unlink the lock file. *Why is this not a great idea*?

Explicit File Locking API

To let processes synchronize ***efficiently*** *on files without resorting to busy-waiting or unconditional sleeps.*

- 4.2BSD, 4.3BSD: **whole-file** locking primitive
  - lock inherited across `fork()`
  - lock automatically released on (last) `close()`
- design choice: *mandatory* vs. *advisory* locks
  - mandatory locks are enforced by kernel;
    advisory locks can be ignored by userspace processes
  - BSD adopted advisory locks. *Why*?

POSIX Byte-Range Locks

In an attempt at improved flexibility & efficiency, POSIX adds ***advisory byte-range locks****.*

- Can lock arbitrary byte ranges: offset + length.
- Can acquire *shared* or *exclusive* locks (→ *reader/writer synchronization*).
  - No overlapping, exclusively locked byte ranges permitted.
- Questionable success: *rarely used in practice*, not powerful and fast enough for serious databases, but adds *substantial implementation complexity* in compliant kernels.

Filesystem Caches

Filesystem Caches

- **Name cache**: path lookup results
  - resolving a long path (e.g., `a/b/c/e/…/z`) requires loading the contents of many directories → many seeks
  - *locality principle*: often, the same name is reused many times (e.g., shell scripts, config files, `$PATH` search, etc.)
- **Buffer cache**: file contents
  - reads: avoid re-reading the same file (→ *locality*)
  - writes: combine many small writes to single disk write
- **Write cache**: memory-based cache on disk controller
  - should be transparent to OS, but can be buggy…

Name Cache

Essential for acceptable name resolution.

- When a translation succeeds, cache successful *name→vnode* lookup in name cache.
- When a translation *fails*, place **negative lookup result** in name cache. *Why is this important*?
- Obviously, much care must be taken to invalidate stale entries (based on either vnode or name).
- The name cache is complementary to directory hashing (or to storing directories as B-trees).

Buffer Cache

Cache file contents in memory.

- Classic UNIX: a separate, fixed-size memory pool created at boot time.
  - strictly separate from memory pool for VM
- **Modern approach**: unified I/O and VM pool
  - makes `MAP_SHARED` + `read()` a lot easier
  - `MAP_ANONYMOUS` vs. VM memory: little difference

Example: FreeBSD Buffer Cache Operations (1/2)

Acquiring and releasing buffers:

- `bread()`: given a *vnode*, an *offset* (in blocks), and a read *length*, return a locked buffer (filled with file contents) → uses FS-specific I/O method
- `brelse()`: release a *clean* buffer, wake any waiting threads
- `bqrelse()`: like `brelse()`, but don't yet reclaim, as reuse is expected

Example: FreeBSD Buffer Cache Operations (2/2)

Write back dirty buffers:

- `bdwrite()`: *delayed* write — buffer is queued for writing, but may be delayed by 20–30 seconds to accumulate later writes to same page(s)
- `bawrite()`: *asynchronous* write — called when a buffer is filled completely and no more writes expected
- `bwrite()`: *synchronous* write — caller must wait until write has completed (e.g., used for *fsync*)

FreeBSD Buffer Queues

All buffers are kept on one of four queues:

1. *dirty list*: changes must still be persisted. Maintained in LRU order: frequently accessed blocks are likely to stay at tail; buffer daemon writes back pages from beginning of list.
2. *clean list*: blocks not currently in use, but expected to be used soon (`bqrelse()`). Maintained in LRU order. When the clean list becomes empty, buffer daemon is triggered.
3. *empty list*: unused metadata without associated buffer memory; ready for reuse.
4. *locked list*: buffers that are currently being written.

Speculative Caching

Can the buffer cache help with files that are accessed only once?

- **Read-ahead**: when a process reads some blocks of a file, automatically queue additional I/O ops for subsequent blocks. → *Expectation: process is going to request them soon anyway.*
- **Write-behind**: don’t make application wait until its writes have actually been written to disk. → *Allows process to compute next writes while data is still being transferred.*

The Buffer Cache Problem

*Cache is stored in fast **volatile** memory → lost on power failure or OS crash.*

1. Write-through cache
   → all writes synchronously written to disk
   → cache helps only with reads
2. Write-through only FS metadata
   → maintains FS consistency, but risks losing (seconds of) user data
   → classic UNIX Approach; still slow for FS-intensive workloads
3. *write-ahead logging*: maintain log in fast, non-volatile memory (or on separate disk): widely used today
4. *soft updates*: carefully order updates such that version on disk is always consistent (FreeBSD)