File Systems Basics

Nima Honarmand

File and "inode"

- **File**: user-level abstraction of storage (and other) devices
  - Sequence of bytes
- **inode**: internal OS data structure representing a file
  - *inode* stands for ***index node***, historical name used in Unix
- Each inode is identified by its index-number (***inumber***)
  - Similar to processes being identified by their PID
- Each file is represented by <u>exactly one </u>inode in kernel
- We store both inode as well as file data on disk

File Data vs. Metadata

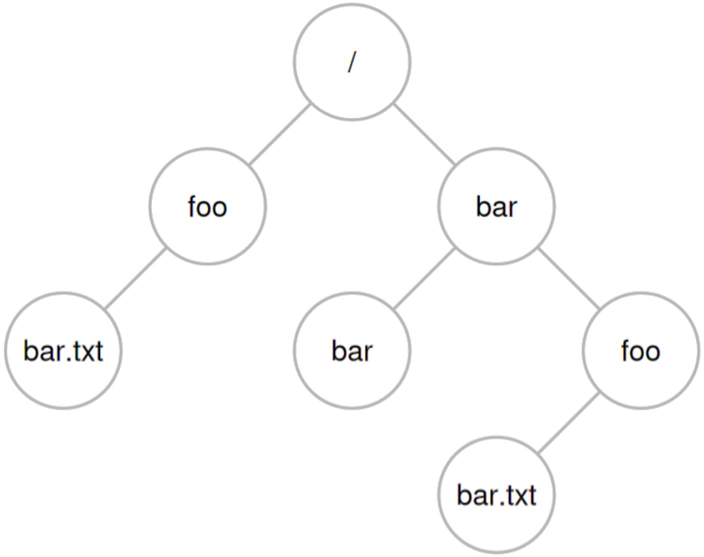
- **File Data**: sequence of bytes comprising file content
- **File Metadata**: other interesting things OS keeps track of for each file
  - Size
  - Owner user and group
  - Time stamps: creation, last modification, last access
  - Security and access permission: who can do what with this file
- inode stores metadata and provides pointers to disk blocks containing file data

Directory and "dentry"

- **Directory**: special file used to organize other files into a hierarchical structure
  - Each directory is a file in its own right, so it has a corresponding inode
- Logically, directory is a list of **<file-name, inumber>** pairs
  - Internal format determined by the FS implementation
- File name is not the same thing as the file, it's just a string of characters we use to refer to the file
  - inode is actual the file
- **Directory entry**: each <file-name, inumber> pair
  - Called a ***dentry*** in Linux; we'll use this name

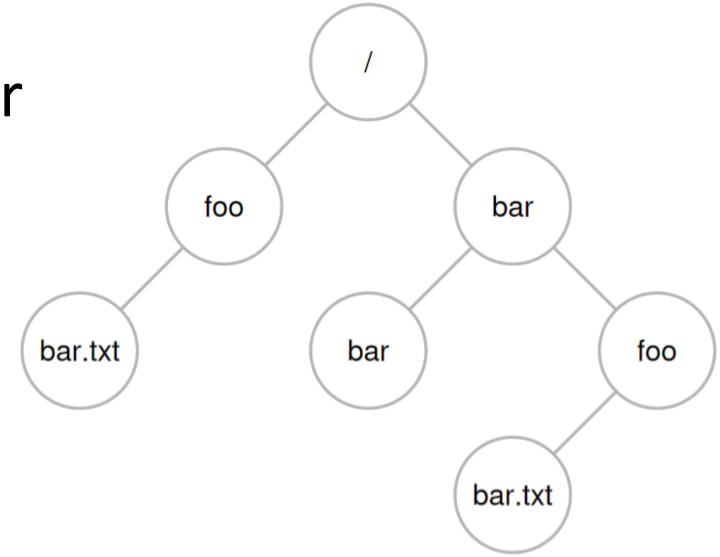
Directory Hierarchy

- Each dentry can point to a normal file or a another directory.
- This allows hierarchical (tree-like) organization of files in a file system.
- In this tree, all internal nodes are directories and leaves are ordinary files.



File Path

- File path is the human-readable string of characters we use to refer to a node in directory tree



- For example:
  - /
  - /foo
  - /bar/foo/bar.txt

- Each valid path corresponds to exactly one dentry
  - <u>And dentry points to exactly one inode</u>

- Multiple dentries can point to the same inode

  → **Multiple paths might map to the same file**

Hard Links

- An inode uniquely identifies a file for its lifespan
  - Does not change when renamed
- Each dentry that points to an inode is a ***hard link*** to that file
  - We'll talk about soft links later
- inode keeps track of these links to the file
  - Count "1" for every such link
- When link count is zero, file becomes inaccessible and can be garbage collected
  - There is no 'delete' system call, only '***unlink***'

Demo: link count in output of `ls -l`

File Operations: `open()`

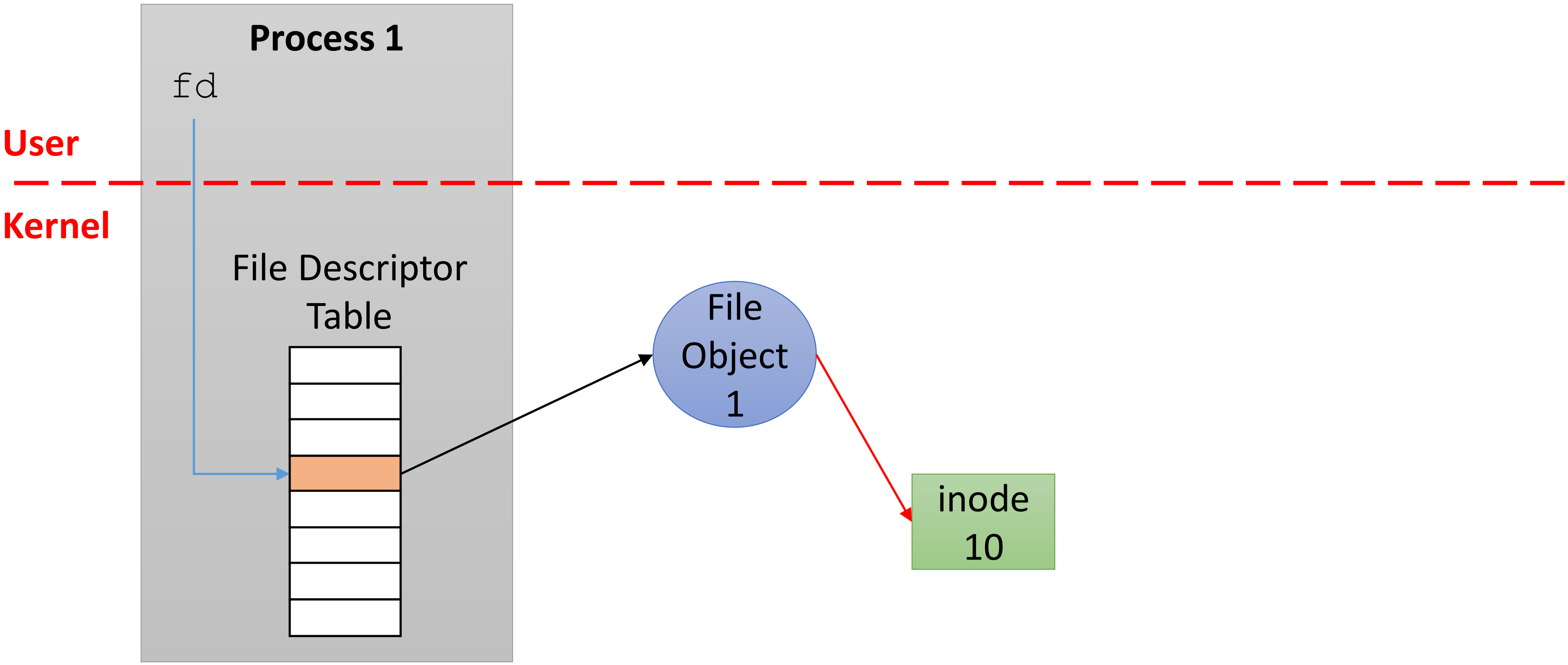
```
int open(char *path, int flags, int mode);
```

- Traverses the directory tree to find the dentry corresponding to `path`
- Checks/does a lot of things according to `flags`
- Examples of `flags`:
  - O_RDONLY, O_WRONLY, O_RDWR: requested type of access to file
  - O_CREAT: create if not existing
  - O_TRUNC: truncate the file upon opening
  - And many others; see the man page
- `mode` is used to set the file permissions if a new file is created

File Operations: `open()`

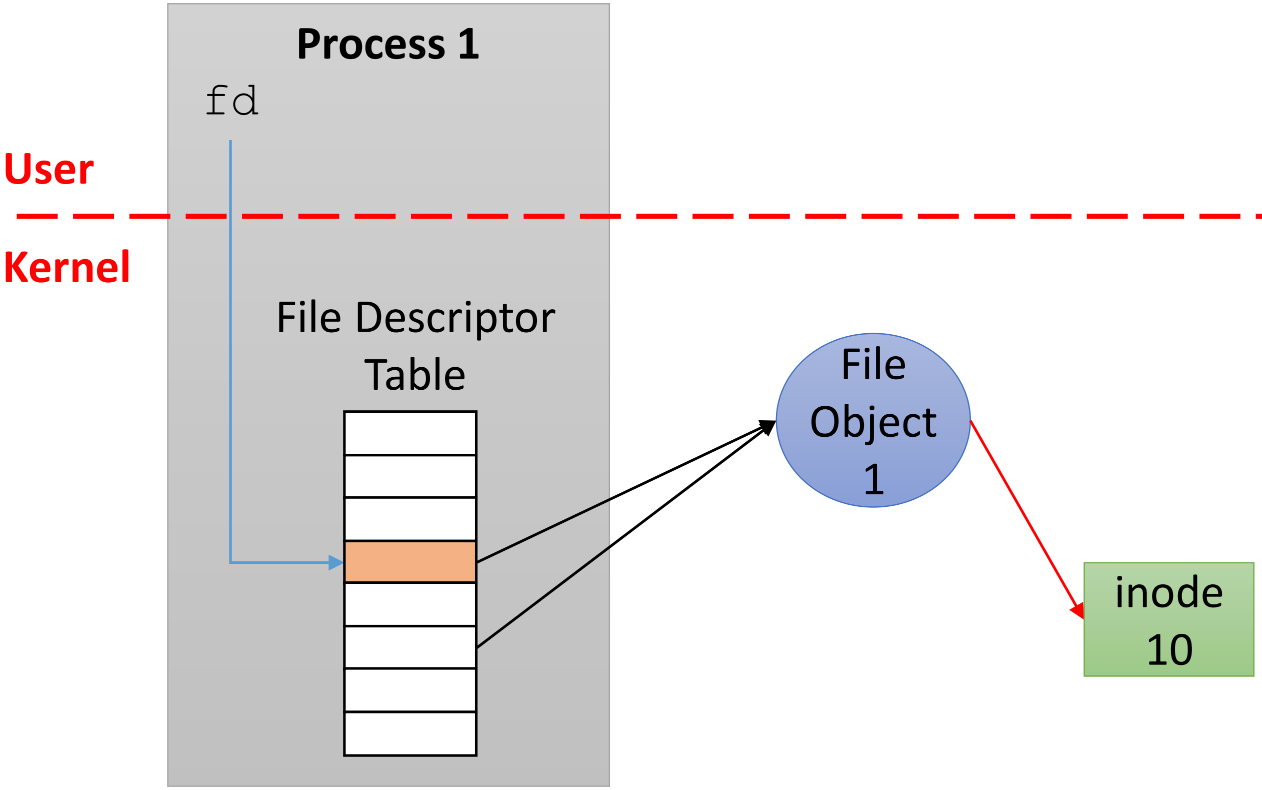
- If path is valid and requested access is permitted, `open()` returns a **file descriptor**
- File descriptor is an index into the per-process **File Descriptor Table**
  - FDT is a kernel data structure; user program only has a index into it
- Each entry in file descriptor table is a pointer to a **File Object**
  - File object represents <u>an instance if an opened file</u>
- File object then points to the inode (either directly or through dentry)

File Descriptors and File Objects



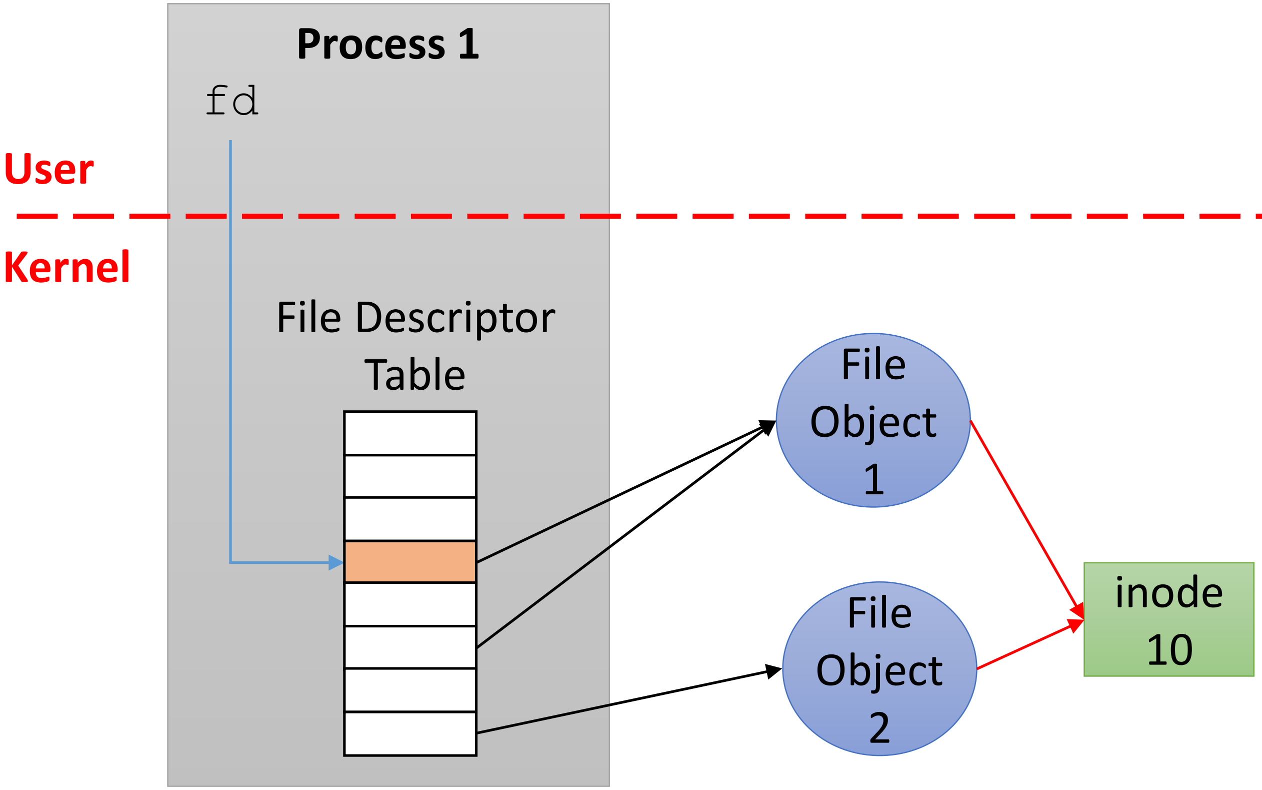
- `fd` indexes into FDT; FDT entry points to File Object
- File object points to corresponding inode

File Descriptors and File Objects



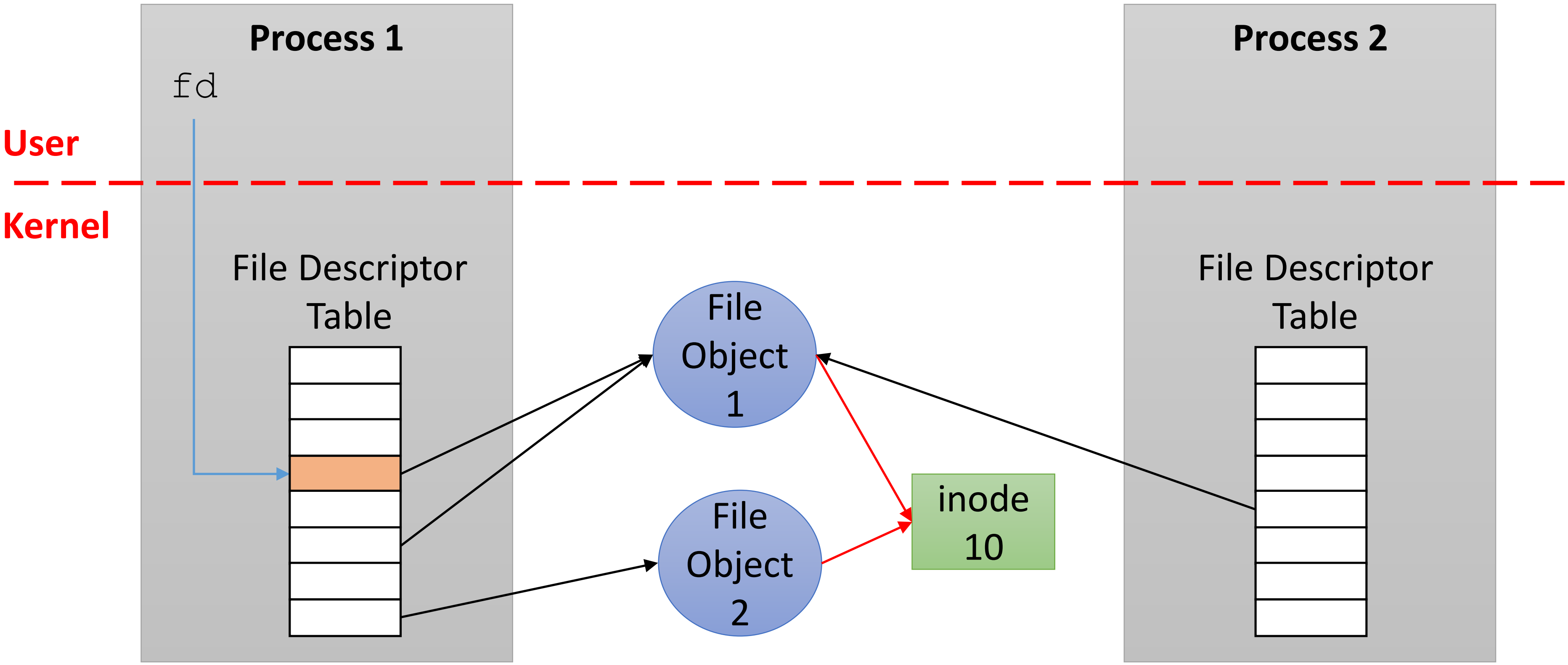
- Multiple entries in same FDT may point to same file object
  - E.g., after a `dup()` syscall

File Descriptors and File Objects



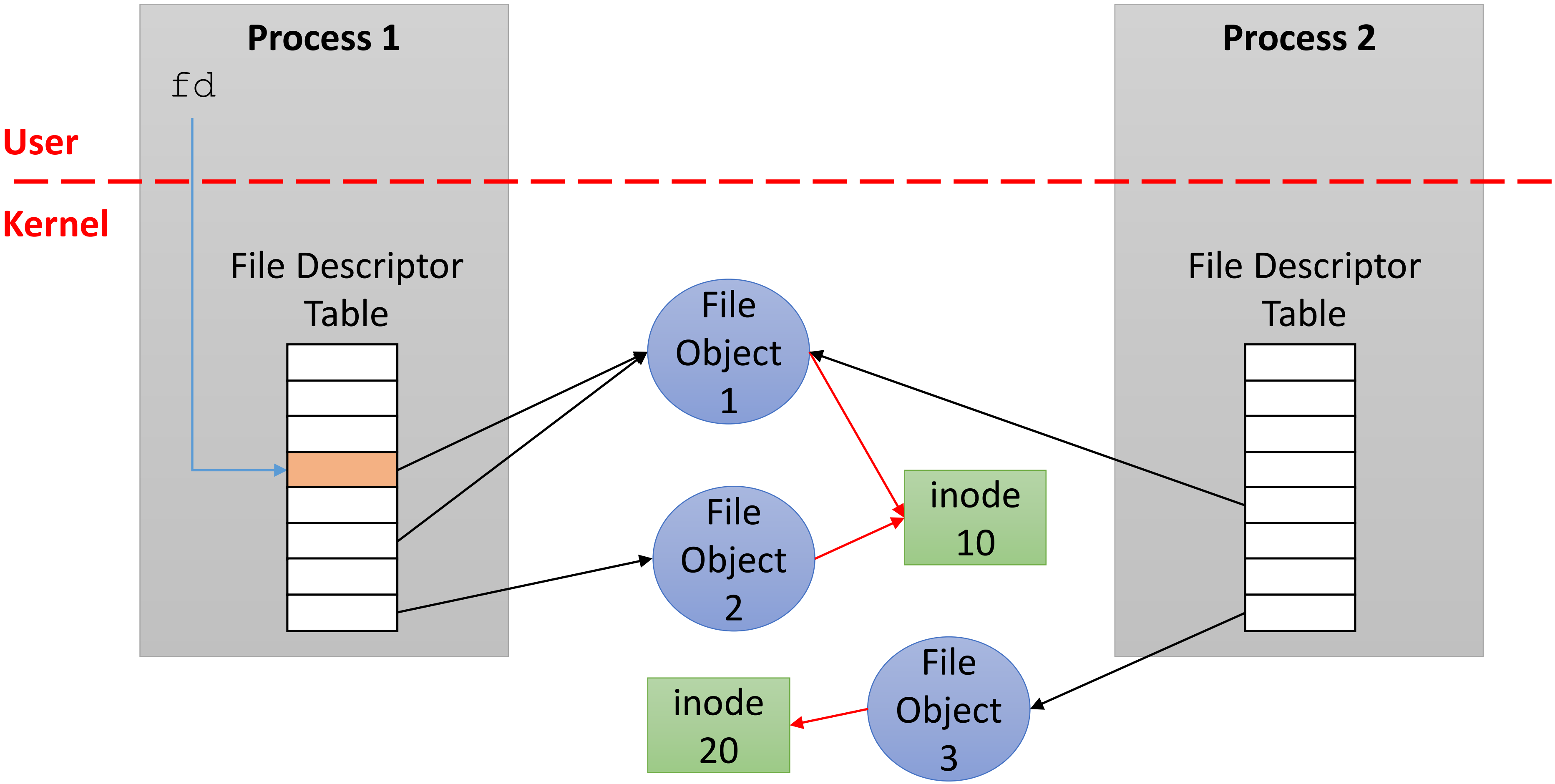
- Multiple file objects might point to the same inode
  - E.g., if the file has been opened multiple times
  - Either by the same process or a different one

File Descriptors and File Objects



- The same file object might be pointed to by FDTs of different processes
  - E.g., due to `fork()`. Remember? FDT gets copied at form time.

File Descriptors and File Objects



Overall Picture

Why File Objects?

- Why don’t FDT entries directly point to inodes?
- Because each time you open a file, you might use different flags
  - E.g., Different permission requests
- Also, kernel tracks the “current offset” of each open file
  - Multiple open instances of the same file may be accessing the file at different offsets
- Again, use an extra-level of indirection to solve the problem!

Absolute vs. Relative Paths

- Each process has a ***working*** directory
  - Stored in its PCB
  - Specifically, it is a dentry pointer
- First character of path dictates whether to start search from root dentry (/) or current process's working directory dentry
  - An absolute path starts with the '/' character (e.g., /lib/libc.so)
  - Anything else is a relative path (e.g., foo/vfs.pptx)

File Path Lookup

- Execute in a loop looking for next piece
  - Treat ‘/’ character as component delimiter
  - Each iteration looks up part of the path
- Ex: ‘/home/myself/foo’ would look up…
  - ‘home’ in / → dentry ***A*** → inode ***X***
  - ‘myself’ in content of ***X*** → dentry ***B*** → inode ***Y***
  - ‘foo’ in content of ***Y*** → dentry ***C*** → inode ***Z***
- In every step, kernel should also check access permissions to see if user has been granted access

`open()` continued

- If inode found, create a new file object, find a free entry in FDT, and put the file object pointer there
- What if FDT is full?
  - Allocate a new table 2x the size and copies old one
- What if inode is not found?
  - `open()` fails unless O_CREAT flag was passed to create the file
- Why is create a part of open?
  - Avoid races in `if (!exist()) create(); open();`

File Operations: `read()` & `write()`

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- Read and write count number of bytes from file
  - But from where in the file?
- Kernel maintains a current location (sometimes called ***cursor***) for each open file
- Read and write start from that location, and advance the cursor by number of bytes read/written

File Operations: `read()` & `write()`

- Having a cursor serves sequential file accesses
- What if we need to access a random location in a file?

Two solutions:

1) Change the cursor before read/write
    - `off_t lseek(int fd, off_t offset, int whence);`
2) Use random-access versions of read/write:
    - `ssize_t pread(int fd, void *buf, size_t count, off_t offset);`
    - `ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);`

Demo: Using `strace` to see syscalls made by `cat`

File Operations: `link()`

```
int link(const char *oldpath, const char
*newpath);
```

- Creates a new hard link with path `newpath` to inode represented by `oldpath`
  - Creates a new name for the same inode
    - Opening either name opens the *same* file
  - This is *not* a copy
- This is the syscall used by Linux's `ln` command

Interlude: Symbolic Links

- Special file type that stores a string
  - String usually assumed to be a filename
  - Created with `symlink()` system call
- How different from a hard link?
  - Completely
  - <u>Doesn’t raise the link count of the file</u>
  - Can be “broken,” or point to a missing file (just a string)
- Sometimes abused to store short strings

```
[myself@newcastle ~/tmp]% ln -s "silly example" mydata
[myself@newcastle ~/tmp]% ls -l
lrwxrwxrwx 1 myself mygroup 23 Oct 24 02:42 mydata -> silly example
```

File Operations: `unlink()`

```
int unlink(const char *pathname);
```

- Removes the dentry corresponding to `pathname`
- Decreases link count of corresponding inode by 1
  - If inode link count reaches 0, FS can garbage collect it; Otherwise, leaves it be because there are other dentries pointing to it.
- This is the syscall used by Linux's `rm` command
  - There is no 'delete' system call, only `unlink()`

Interlude: Link Count & Ref Count

- inodes and dentries live in two worlds
  - ***On-disk*** copy
  - ***In-memory*** copy
- In-memory copies are caches of on-disk copies
  - E.g., inode cache keeps an in-memory copy of all on-disk inodes that may be used by some process
  - Similarly, for the dentry cache
- The kernel needs to know when it is safe to remove an on-disk copy or free an in-memory copy

Interlude: Link Count & Ref Count

- For in-memory copy, we use ***reference counts*** to in-memory objects
  - For every C pointer in kernel that points to an in-memory copy, increment ref count by 1
  - When someone releases the pointer, decrement ref count
  - When ref count reaches 0, it is safe to garbage-collect
- For on-disk copy, we use both hard-link count as well as ref count
  - E.g., it is only safe to garbage collect an on-disk inode when
    - There is no hard link pointing to it
    - There is no C-pointer to its in-memory cached copy

Example: Common Trick for Temp Files

- How to clean up temp file when program crashes?
- Use following syscalls to create the temp file
  - `open()` with O_CREAT (1 link, 1 ref)
  - `unlink()` (0 link, 1 ref)
- File gets cleaned up when program dies
  - Kernel removes last reference on exit
  - Happens regardless if exit is clean or not
  - Except if the kernel crashes / power is lost
    - Need something like fsck to “clean up” inodes without dentries
    - Dropped into `lost+found` directory

File Operations: `rename()`

```
int rename(const char *oldpath, const char
*newpath);
```

- Atomically renames a file, assuming `oldpath` is valid
- Deletes dentry corresponding to `oldpath`
- Creates a new dentry corresponding to `newpath`
- Note: `newpath` and `oldpath` might be in different directories

Example: How Editors Save Files

- Hint: don't want half-written file in case of crash
- General approach
  - Create a *temp* file (using `open`)
  - Copy *old* to *temp* (using `read` *old* / `write` *temp*)
  - Apply writes to *temp*
  - Close both *old* and *temp*
  - Do `rename(temp, old)` to atomically replace
- Drawback?
  - Hint 1: what if there was a second hard link to *old*?
  - Hint 2: what if *old* and *temp* have different permissions?

File Operations: `close()`

```
int close(int fd);
```

- Removes the entry from File Descriptor Table and decreases corresponding file object's ref count
- Can garbage-collect the file object if its ref count reaches 0, which in turn, decrements inode's ref count
- If inode's ref count reaches 0, can garbage-collect in-memory copy
  - If link count is also 0, can garbage-collect the on-disk copy
- FDs also closed when process exits
  - If not closed already

Other File Operations

- `dup()`, `dup2()` — Copy a file handle
  - Creates a second table entries for same file object
  - Obviously, increments file object's reference count
- `fstat()` — returns the file metadata stored in the inode
- `fcntl()` — Set flags on file object
  - E.g., `CLOSE_ON_EXEC` flag prevents inheritance on `exec()`
    - Can be set by `open()` or `fcntl()`

Directory Operations

- Creation
  - `int mkdir(const char *pathname, mode_t mode);`
- Removal
  - `int rmdir(const char *pathname);`
  - Only removes a directory if it is empty
    - i.e., no dentries other than `.` and `..`

Directory Operations: Traversal

- POSIX interface:
  - `DIR *opendir(const char *name);`
  - `struct dirent *readdir(DIR *dirp);`
  - `int closedir(DIR *dirp);`
- Linux kernel syscalls
  - `open()`: regular open syscall
  - `int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count);`
  - `close()`: regular close syscall

Syscalls vs. STDIO operations

- The operations we discussed so far are system calls, (typically) implemented by the kernel
  - They all use file descriptors to refer to files
  - They are often included from `<unistd.h>`
- In C, `<stdio.h>` adds another layer of abstraction on top of kernel files, called ***streams***
  - Streams are represented by `FILE` objects, which are user-mode (library) structures
- Stream operations that might be confused w/ syscalls often have a "f" prefixed to their names
  - E.g., `fopen()`, `fclose()`, `fread()`, `fwrite()`
- Other stream ops may or may not have an "f" prefix
  - E.g., `fputc()` and `putc()`

Multiple File Systems

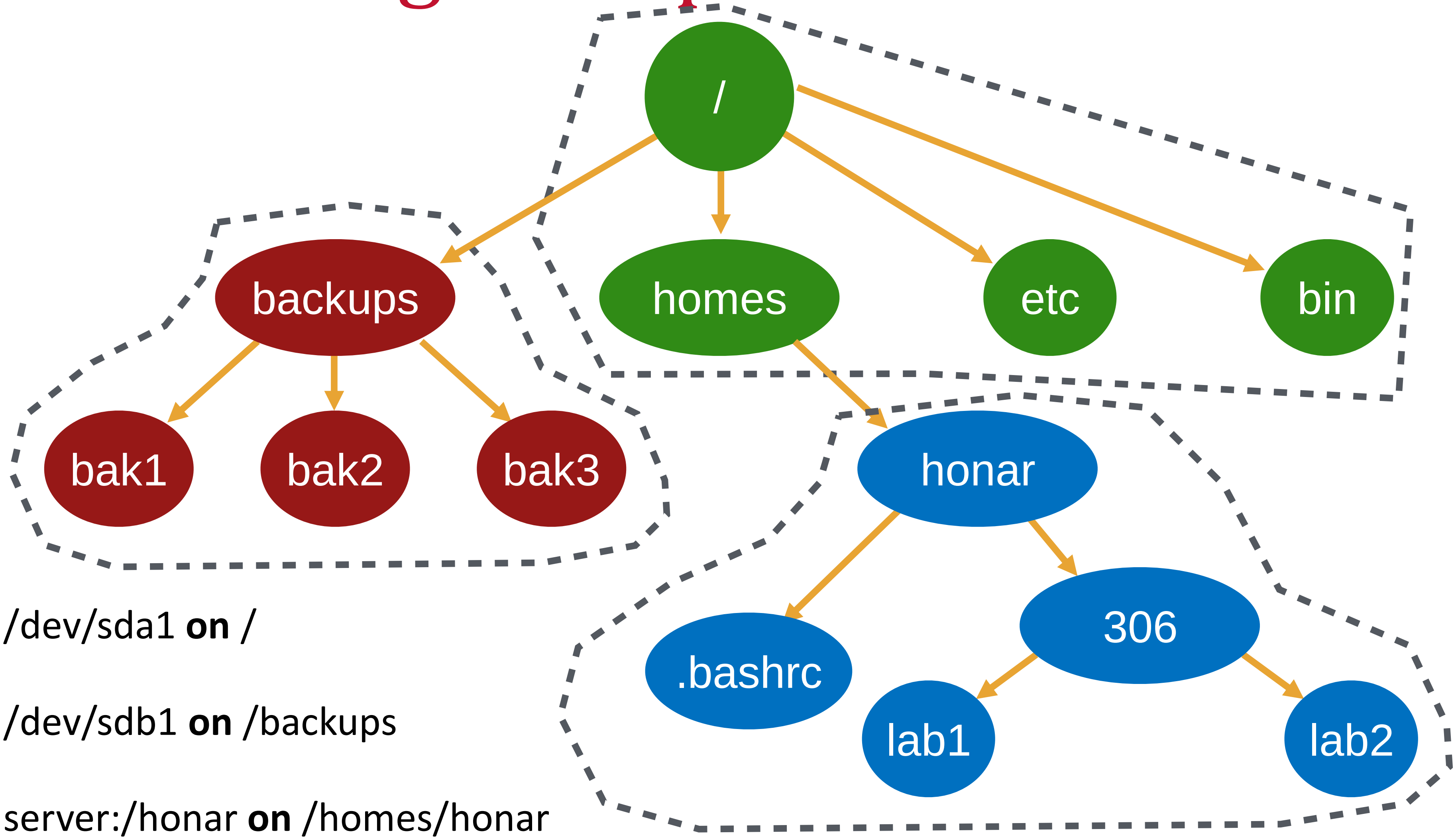
- Users often want to have multiple file systems
  - Multiple partitions per disk
  - Multiple disks
  - USB sticks
  - CD/DVD
  - Network file systems
  - etc.
- How to do this?
  - Windows approach: make each file system a separate Drive (C, D, etc.)
  - Unix approach: keep everything in one tree

Mounting Multiple FS

- Idea: stitch all the file systems together into a super file system!

```
~$ mount
/dev/sda1 on / type ext4 (rw)
/dev/sdb1 on /backups type ext4 (rw)
server:/honar on /homes/honar type nfs
```

Mounting Multiple FS



- /dev/sda1 **on** /
- /dev/sdb1 **on** /backups
- server:/honar **on** /homes/honar

Mounting Multiple FS

- Now that you know directory structure and FS objects, can you tell how it is done?
- The dentry corresponding to the mount location, points to the root inode of the mounted file system
  - E.g., dentry corresponding to /backups points to the inode corresponding to the root of the file system od /dev/sdb1.
- The actual implementation is a bit more complicated but this is the gist of it.

Core FS Objects (1)

- **inode** (index node): represent one file
  - Keeps metadata as well as pointers to data blocks
- **dentry** (directory entry): name-to-inode mapping
- **File object**: represents an opened file
  - Keeps pointer to inode (or dentry), access permissions, and file offset

Core FS Objects (2)

- **Superblock**: global metadata of a file system
  - E.g., a magic number to indicate FS type
  - E.g., allocation bitmaps to find free inodes and data blocks
  - Many file systems put this as first block of partition
- <u>Superblocks, inodes and dentries</u> are stored on-disk
  - and cached in main memory when accessed
- <u>File object</u> is only in memory