File systems

Mythili Vutukuru

CSE, IIT Bombay

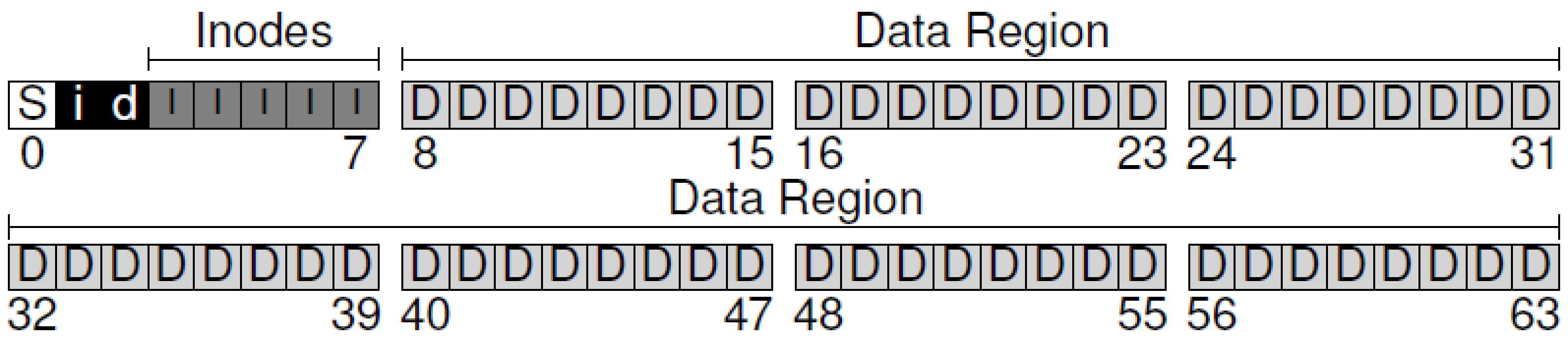
File System

- An organization of files and directories on disk
- An OS can implement one or more file systems
- Two main aspects of file systems
  - Data structures to organize data and metadata on disk
  - Implementation of system calls like open, read, write using the data structures
- Disks expose a set of blocks (usually 512 bytes)
- File system organizes files onto blocks
  - System calls translated into reads and writes on blocks

Index node (inode)

- Files are variable sized, split into fixed size blocks, stored non-contiguously
  - Much like how memory image of process is split into fixed size pages
  - Fixed size blocks avoids external fragmentation of disk storage
- For every file, index node (inode) keeps track of all the block numbers (locations on disk) where the file data is stored
  - Equivalent to a page table which keeps track of physical frame numbers
- Inode of a file is also stored in disk blocks
  - Much like how page table is stored in one or more pages hierarchically
- Inode stores all metadata about file (size, permissions, time of last access/modification, disk block numbers of file data, ..)

Example: a simple file system

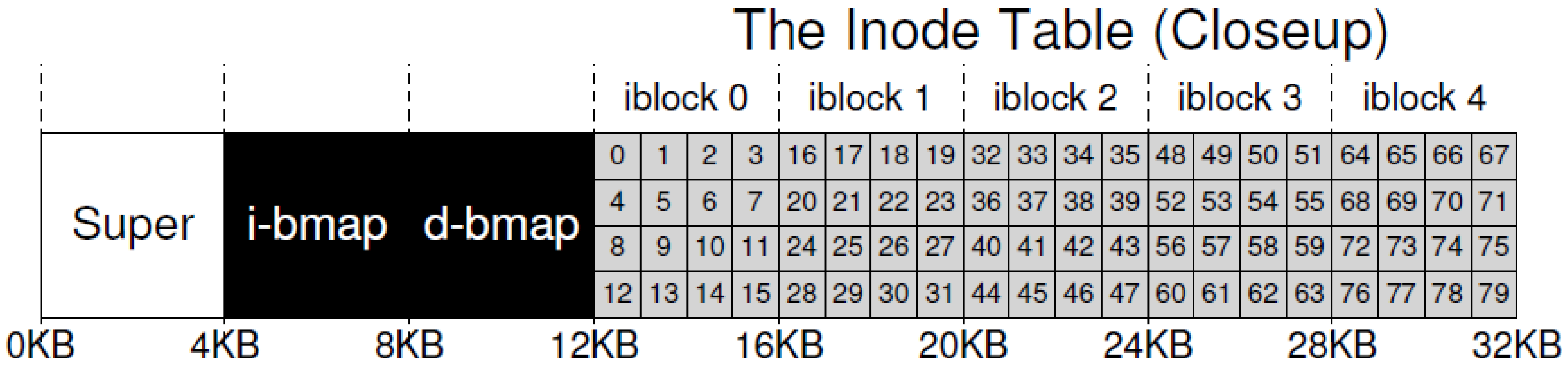


- Data blocks: file data stored in one or more blocks
- Metadata about every file stored in inode
  - Location of data blocks of a file, permissions etc.
- Inode blocks: each block has one or more inodes
- Bitmaps (or free lists): indicate which inodes/data blocks are free
- Superblock: holds master plan of all other blocks (which are inodes, which are data blocks etc.)
- Structure imposed when hard disk is “formatted” with a file system

Image credit: OSTEP

Inode on disk

- Usually, inodes (index nodes) stored in array contiguously on disk
  - Inode number of a file is index into this array, uniquely identifies file



- What does inode store?
  - File metadata: permissions, access time, etc.
  - Pointers (disk block numbers) of file data

Image credit: OSTEP

Structure of inode

- Inode stores block numbers hierarchically
  - Inode contains the block numbers of first few blocks of a file (direct blocks)
  - If direct blocks are full, inode contains block number of single indirect block, which contains block numbers of next few blocks of a file
  - If single indirect block is full, inode contains block number of double indirect block, which contains block numbers of more single indirect blocks
  - Triple indirect block can also be used for large files
- Not a symmetric hierarchical structure like page table
  - Most files are small, so first few block numbers of a file are made available easily without accessing multiple levels of inode
- Accessing a file from disk may require multiple disk accesses for inode

Limitations on file size

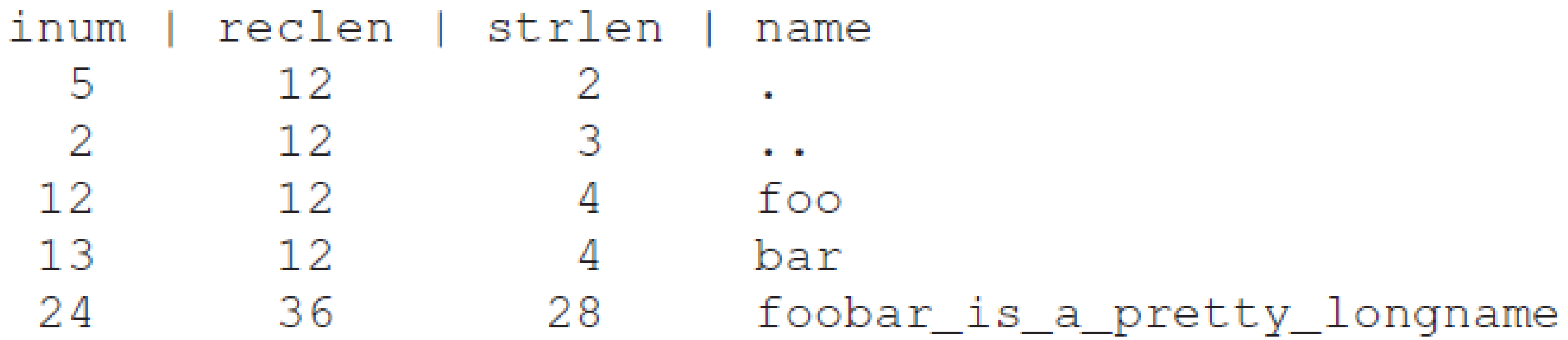
- Filesystem metadata imposes limits on maximum size of file that can be stored on filesystem, maximum number of files, maximum disk size that can be managed, and so on..
- Example: limit on file size imposed by inode structure
  - Suppose inode can store K direct blocks, one single, double, triple indirect block each
  - Suppose single indirect block can store N block numbers, double indirect block can store block numbers of N single indirect blocks, triple indirect block can store block numbers of N double indirect blocks
  - Maximum file size = K + N + N^2 + N^3 blocks
- Different file systems differ in these limits

Directories

- Directory is also a special kind of file in Linux-like operating systems
- File type in inode identifies if regular file or “directory” file
- Directory is a “file” which contains special data: names of files or sub-directories located within it, and their inode numbers
  - Data blocks of directory store these mappings between file names and inode numbers of the file
  - Inode of directory keeps track of the data blocks of directory
- How are filename→inode number mappings stored in directory data blocks?

Directory structure

- Directory stores records mapping filename to inode number



inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

- Fixed size records in arrays, linked list of records, or more complex structures (hash tables, binary search trees etc.)
- How to lookup a file inode number in a directory?
  - Fetch inode of directory, locate its data blocks, read data blocks
  - Search for filename in data blocks of directory (traverse array/linked list/binary search tree) and retrieve inode number of file

Image credit: OSTEP

Pathnames

- File identified in filesystem by its pathname: series of directories, starting at root dir, leading to a file in the root filesystem
- When we want to open/read/write file, we need to find its inode number (from which we can retrieve file data) using pathname
- Given a pathname of file, how to locate its inode number?
  - Start with root directory inode (well known)
  - For every element (directory) in pathname, read directory data blocks, lookup next element filename in directory, retrieve inode number of next element
  - Repeat above process recursively, until entire path name is traversed and we find inode number of the desired file in its parent directory

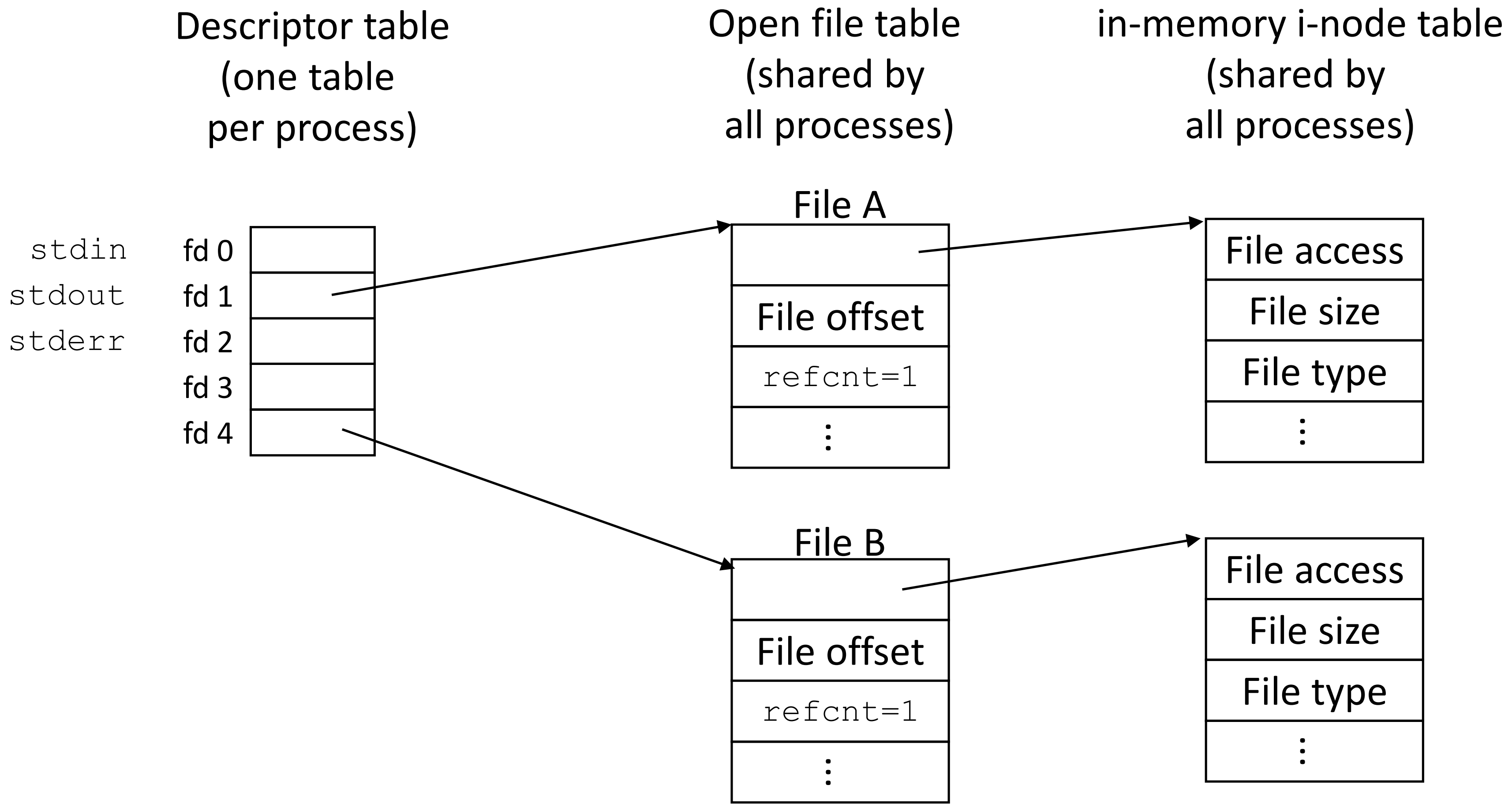
Opening a file

- Why open a file before using it? To have the inode readily available (in memory) for future operations on file
    - Open returns file descriptor which points to in-memory inode
    - Reads and writes can access file data from inode
- What happens during open?
    - The pathname of the file is traversed, starting at root
    - Inode of root is known, to bootstrap the traversal
    - Recursively do: fetch inode of parent directory, read its data blocks, get inode number of child, fetch inode of child. Repeat until end of path
    - If new file, new inode and data blocks will have to be allocated using bitmap, and directory entry updated

In-memory data structures

- When a file is opened, in-memory inode is cached from on-disk copy
  - Quick access of file data block numbers as long as file is in use
- Open file table: data structure used to keep track of open files
  - Shared across all processes in the system
  - One open file table entry created for every open system call
  - Contains pointer to in-memory inode and other information about open file (e.g., offset at which the file is being read/written)
  - Entries created on opening sockets, pipes also (point to socket/pipe info)
- File descriptor array: per-process array of open files
  - Part of the PCB of a process, file descriptor number returned is index into this array
  - Contains pointer to open file table entry
  - Every process has three files (standard in/out/err) open by default (fd 0, 1, 2)

In-memory data structures



```
fd = open("/home/foo/a.txt", flags)
read(fd, ..)
write(fd, ..)
close(fd)
```

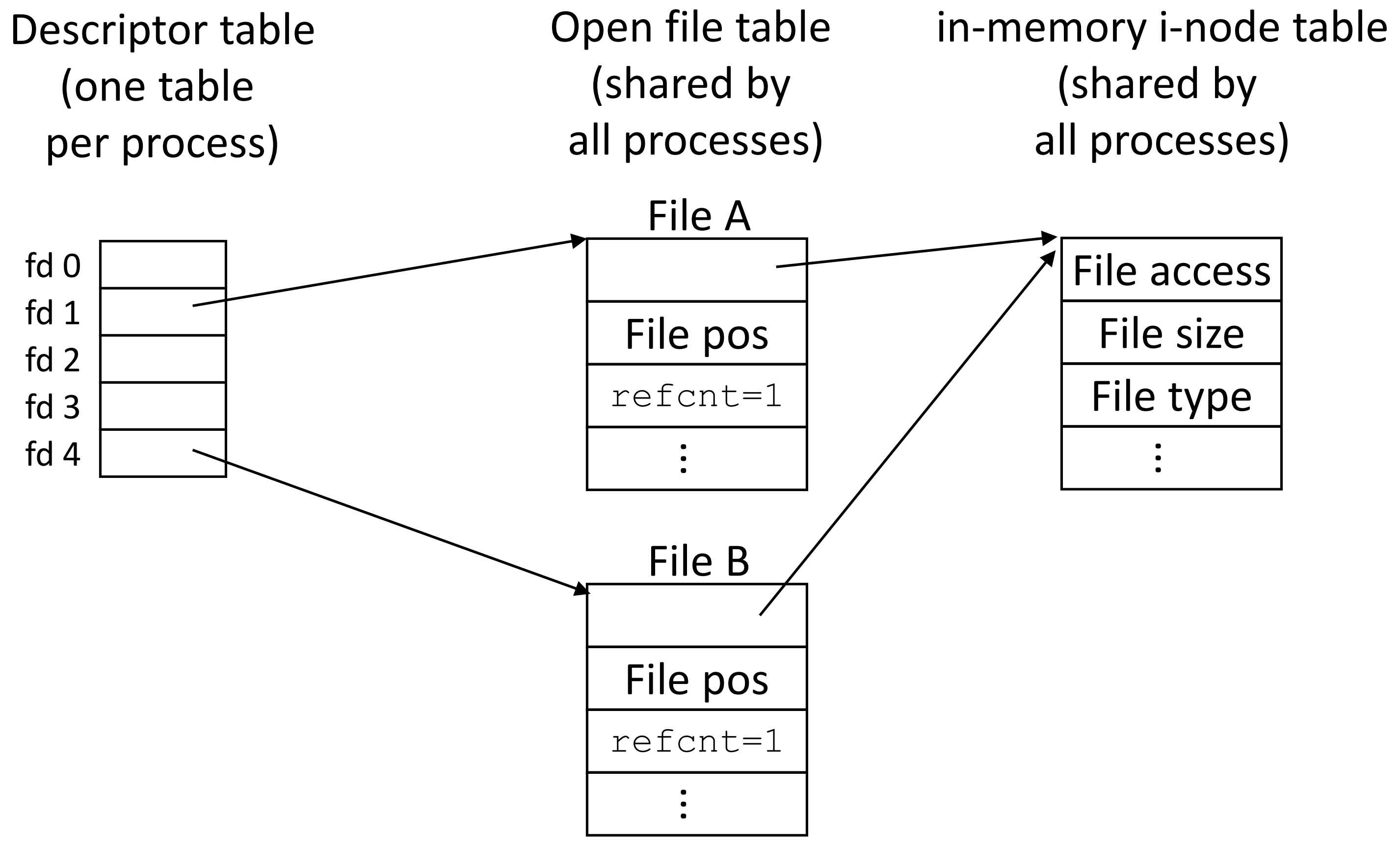
Open system call

- Takes file pathname and other flags as input, returns file descriptor of file
    - Traverse pathname in directory tree, find inode number of file
    - Create a new file if one doesn't exist (depending on flags given to open) → allocate new inode, add mapping from filename to inode number in parent directory
    - Copy inode of file into memory from disk (if not already present in memory)
    - Create new open file table entry, with pointer to in-memory inode
    - Allocate free entry in file descriptor array, store pointer to open file table entry
    - Return index of newly allocated file descriptor array entry
- Every process has 3 files open by default, subsequent open files will get next free entries in file descriptor array
- Close system call deletes file descriptor and open file table entries

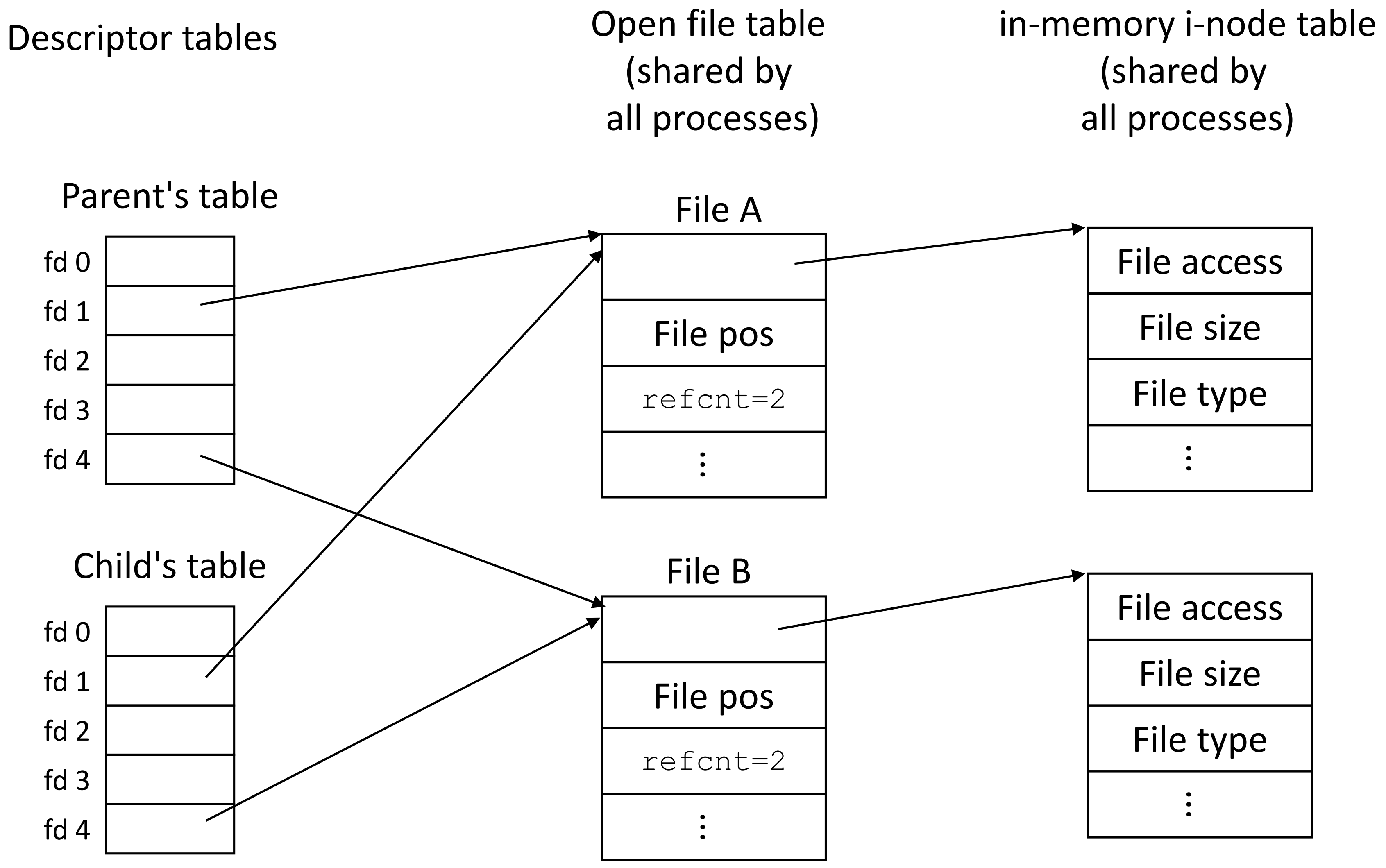
More on open file table

- Every open system call creates new entry in open file table and file descriptor array
- Suppose same file opened by two separate “open” system calls
  - Will result in separate entries in open file table, and file descriptor array, because offset of reading/writing can be different
  - Multiple open file table entries store pointer to same inode of the file
- When parent forks child process, file descriptor array of parent is duplicated in child process
  - Parent and child file descriptor arrays point to same open file table entries
  - Offset of file reading/writing are shared between parent and child
  - Usually one of them should close the file for correct operation
- Reference count used to track multiple pointers to same entry

Example: same file opened multiple times

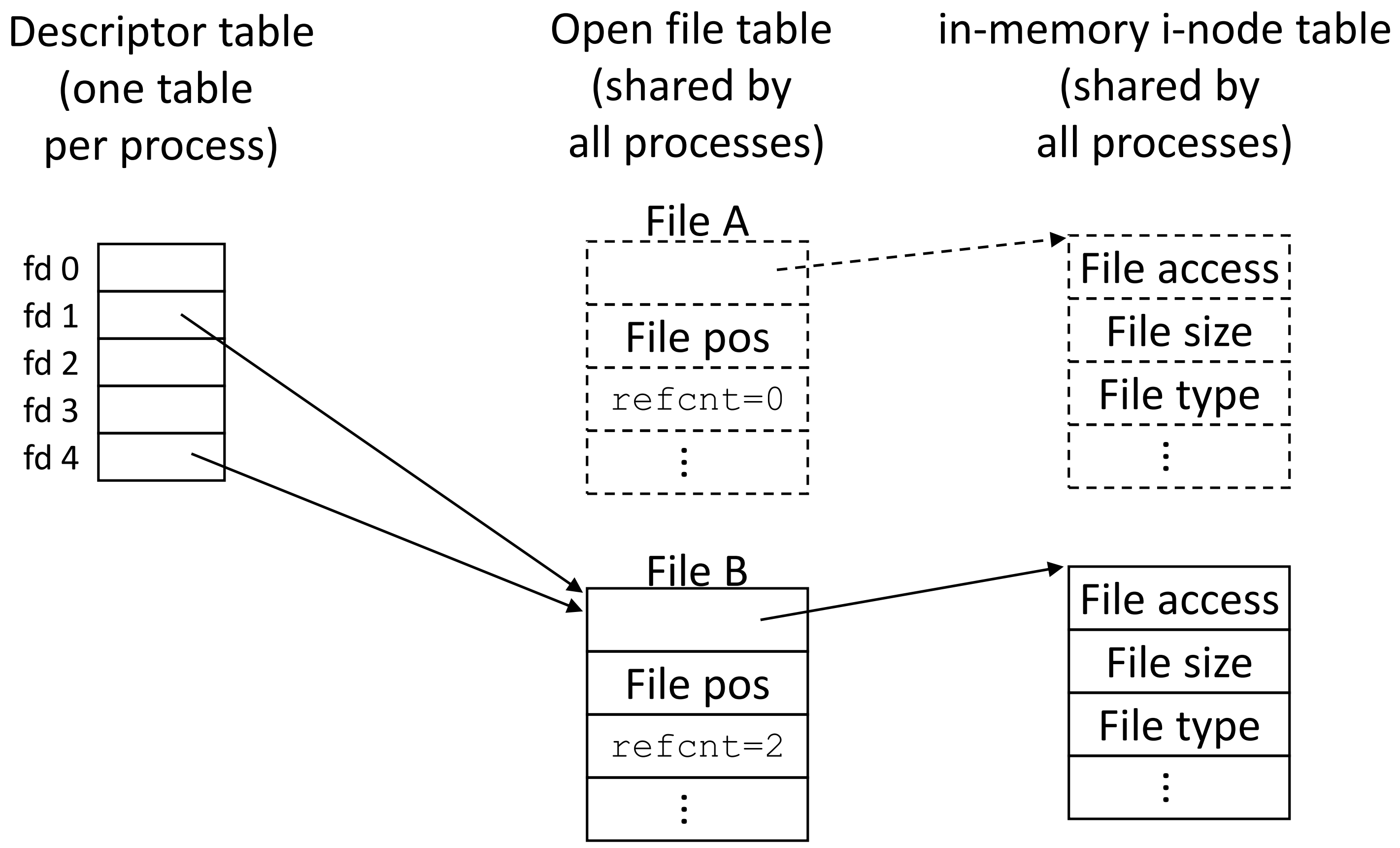


Example: file system data structures after fork



Dup2 and redirection

- System call dup2 used to duplicate file descriptors
- One fd entry made as duplicate of another entry
- Used for I/O redirection by making stdin or stdout as duplicate of a file's file descriptor


Descriptor table (one table per process)
Open file table (shared by all processes)
in-memory i-node table (shared by all processes)
fd 0
fd 1
fd 2
fd 3
fd 4
File A
File pos
refcnt=0
⋮
File access
File size
File type
⋮
File B
File pos
refcnt=2
⋮
File access
File size
File type
⋮

Disk buffer cache

- File data that is read from hard disk is retained in memory for some time in the disk buffer cache = memory pages that cache recently read disk data
- Any changes to disk data is made in the cached copy of disk buffer cache first, then written to disk later
  - Write-through cache: changes written to disk immediately (synchronous writes)
  - Write-back cache: changes written to disk after some delay (asynchronous writes)
  - Write-back cache has better performance, but can lose data in case of power failure
- Benefits of disk buffer cache
  - Improved performance due to fewer disk accesses
  - Merge changes when multiple processes modify same file data
- Most OS allocate unused physical memory to disk buffer cache
  - Some applications doing their own optimizations can bypass cache

Read system call

```
fd = open("/home/foo/a.txt")
char buf[64]
n = read(fd, buf, 64)
```

- Input is file descriptor, user memory to read into, number of bytes to read
  - Use file descriptor array index, access open file table entry, then inode
  - Based on offset, identify which data block(s) of file to read using inode information
  - Check if file data block(s) present in disk buffer cache
    - If cache miss, device driver issues read command to hard disk, process is moved to blocked state, OS will context switch to another process
    - When read completes, device controller will DMA the block(s) into an empty buffer in disk buffer cache, raises interrupt
    - OS handles interrupt, marks process as ready to run, scheduler will switch to process in future
  - Copy requested number of bytes from data block(s) in disk buffer cache into user-provided memory buffer
  - User code resumes, system call returns number of bytes actually read, or error
    - Actual bytes may be less than requested, e.g., end of file

Write system call

- Input is file descriptor, user memory buffer containing data to be written, number of bytes to write
  - Using file descriptor and inode, identify which data block(s) of file to write into
  - If we are writing beyond end of file, file size expands, new blocks needed
    - Allocate new data blocks for file on disk (update free list or bitmap)
    - Add new data block numbers into file inode
  - Locate data block(s) present in disk buffer cache
    - If not, read data block(s) into buffer cache first
  - Copy requested number of bytes from user memory buffer into data block(s) cached in disk buffer cache, cached block is now marked "dirty"
    - Write-through cache: synchronously write to disk immediately
    - Write-back cache: asynchronously update disk copy later
  - User code resumes (after delay in case of sync write, immediately for async write), system call returns number of bytes actually written, or error

Memory mapping a file

```
fd = open("/home/foo/a.txt")
char *buf = mmap(fd, size, ..)
buf[0] = …
buf[1] = …
munmap(..)
```

- Alternate ways to read/write a file is via memory mapping
  - mmap system call takes the file descriptor, size of data to memory map, and other arguments, returns the starting virtual address of mmap region
  - File data is read into one or more physical memory frames, which are mapped at free addresses in the process virtual address space (new page table entries)
  - Can access memory mapped file data like any other memory region
  - With demand paging, physical frames can be assigned on-demand only when mmap region accessed
- File can be memory mapped in private or shared mode
  - Shared mode: changes to file are written to disk immediately, seen by others
  - Private mode: changes to file are written to disk when memory unmapped

mmap vs. read/write syscalls

- mmap can be used for file-backed as well as anonymous pages
  - Physical frame mapped into address space can be empty frame or with file data
- Memory mapping a file is an easy way to read file data
  - Executable code, shared library code are memory mapped into virtual address space
- Memory mapping a file avoids extra data copies
  - Read system call reads data first into kernel memory, then copy into user buffer
  - Memory mapping a file copies file data into free physical frames, which are directly accessed by user using virtual addresses
- Memory mapping allows reading disk data in large page-sized chunks
  - Useful when reading/writing large amounts of data from file
  - Not very efficient when reading files in small chunks

Linking and unlinking

- Same file can be “linked” from different directories with different filenames using link system call
  - When file created, it is linked to its parent directory for first time
  - Subsequently, we can link to same file data from another directory also
- Hard linking: add entry in new directory, mapping new filename to old inode
  - If file deleted from old pathname, can still access it from new pathname
  - Link count of file in inode captures the number of such “links” to file inode
- Soft linking: add entry in new directory, mapping new filename to old filename
  - If file deleted from old pathname, soft link is “broken”
- Unlink system call: remove directory entry of a file from a particular directory
  - If this is last “link” to the file, the file is deleted from disk

Crash consistency

- Every system call updates multiple disk blocks
  - Example: when we append data to a file, we change data block, inode block, bitmap, ..
- All changes to disk blocks are first made in memory (disk buffer cache), then written to disk (synchronously or asynchronously)
  - Even metadata blocks (inode) are updated first in disk buffer cache
- If power failure happens in the middle of a system call, memory changes will be lost, disk can be only partially updated, may cause inconsistency in file data
  - Example: new data block written to disk, but not added to inode (written data is lost)
  - Example: new data block number added to inode, but data block contents not written (file contains garbage data)
- Crash consistency: how to ensure filesystem is consistent after a power failure?
  - Problem exists even with write-through disk buffer cache, but more prominent with write-back cache

File system checkers

- Programming tip for crash consistency: always update data blocks on disk first before updating metadata blocks
  - Better to write data block and not link from inode (lost data), rather than link from inode first and fail to update data block (garbage in file)
- Even with above tip, inconsistency can still occur, especially when multiple metadata blocks need to be updated
  - Example: bitmap updated to mark data block as used, inode updated to add pointer to data block, which metadata change to write to disk first?
- File system checking tools (e.g., fsck) check inconsistencies in metadata blocks after reboot and fix the blocks to make them consistent
  - Example: data block marked as used in bitmap, but not present in any inode, so mark as free
- What we want: atomicity (all changes pertaining to a system call happen all at once together or none happens at all)

Logging / journaling

- Logging/journaling: common technique for atomicity in systems
  - Can be applied to guarantee crash consistency in file systems also
- How to add logging to any file system?
  - All changes to be made to disk blocks are first written to a log on disk, original disk blocks are not touched
  - After all changes are logged to disk, special commit entry written to log
  - Next, changes are applied to the original disk blocks, log entries cleared
  - If crash happens before log is committed, then no changes are made to any disk block, it is as if system call never happened
  - If crash happens after log is committed, but before changes applied to original disk blocks, then log is replayed upon reboot and changes are completed

Virtual File System (VFS)

- Different file systems can have different implementations of system calls
  - A file system using logging/journaling may write to log first
  - A different directory implementation (fixed size records vs linked list) will lead to a different lookup function
- How to write filesystem code in a modular manner?
  - Should be easy to change system call implementations and switch filesystems
- Solution: Virtual File System (VFS)
  - Defines a set of objects (files, directories, inodes) and operations to be performed on these objects (open a file, lookup filename in directory, ..) for various system calls
  - A specific filesystem implements these functions on VFS objects, provides pointers to the functions to be invoked by OS
- OS filesystem code is built in layers for modularity: VFS, filesystem implementation, disk buffer cache, device driver