CS 330 - Operating Systems

File Systems

27-10-2025

File Metadata

- File system maintains information about each stored file
- `stat/fstat` system call
  - Pathname to file or file descriptor to fill stat structure

```
struct stat {
  dev_t      st_dev;      // ID of device containing file
  ino_t      st_ino;      // inode number
  mode_t     st_mode;     // protection
  nlink_t    st_nlink;    // number of hard links
  uid_t      st_uid;      // user ID of owner
  gid_t      st_gid;      // group ID of owner
  dev_t      st_rdev;     // device ID (if special file)
  off_t      st_size;     // total size, in bytes
  blksize_t  st_blksize;  // blocksize for filesystem I/O
  blkcnt_t   st_blocks;   // number of blocks allocated
  time_t     st_atime;    // time of last access
  time_t     st_mtime;    // time of last modification
  time_t     st_ctime;    // time of last status change
};
```

Building a File System

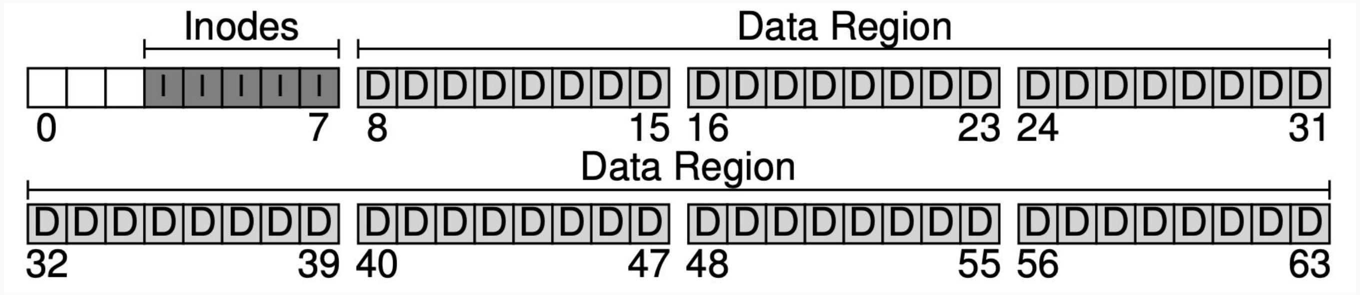
- Simple File System (SFS)
  - Basic data structures
    - Types of on-disk structures used to organize data and metadata
    - Range from arrays to trees
  - Access methods
    - Mapping system calls to its structures
    - Which to read and write to in an efficient manner?

Data Structures in SFS

- Disk divided into blocks
- Simple file systems use one block size (e.g., 4KB block)
    - Series of indexed 4KB blocks
    - (Assume 64 blocks in the examples)
- Storage in blocks
    - User data
    - Information about each of the files
        - Which data blocks are in the file, size, access policies …
        - Use **inodes**; block space for them

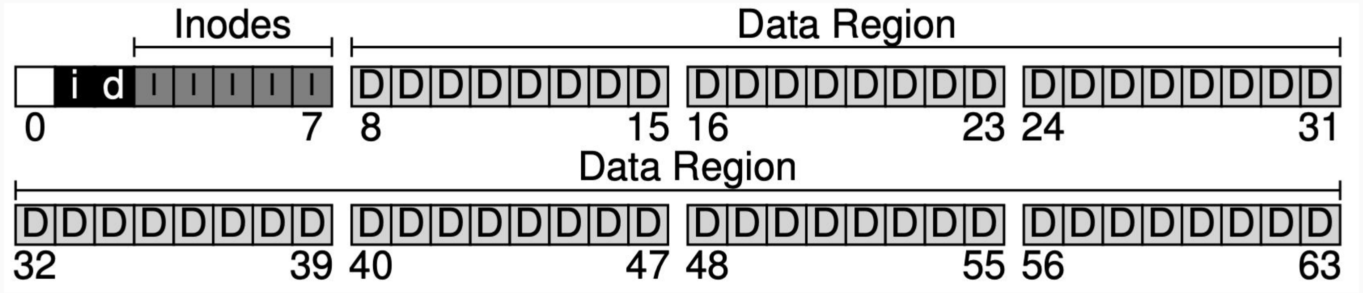
Data Structures in SFS

- Storage in blocks
  - User data
  - Information about each of the files
    - Which data blocks are in the file, size, access policies ...
    - Use **inodes**; block space for them



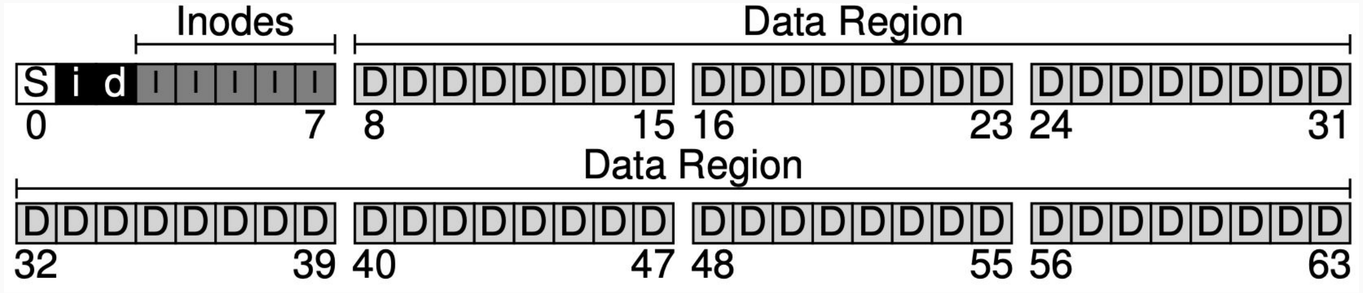
Data Structures in SFS

- Storage in blocks
  - User data
  - Information about each of the files
  - Allocation structures
    - Track free blocks
    - Free list, bitmap etc.



Data Structures in SFS

- Storage in blocks
  - User data
  - Information about each of the files
  - Allocation structures
  - Superblock
    - Contains details about the file system like inode table, no. of inodes, blocks, etc.

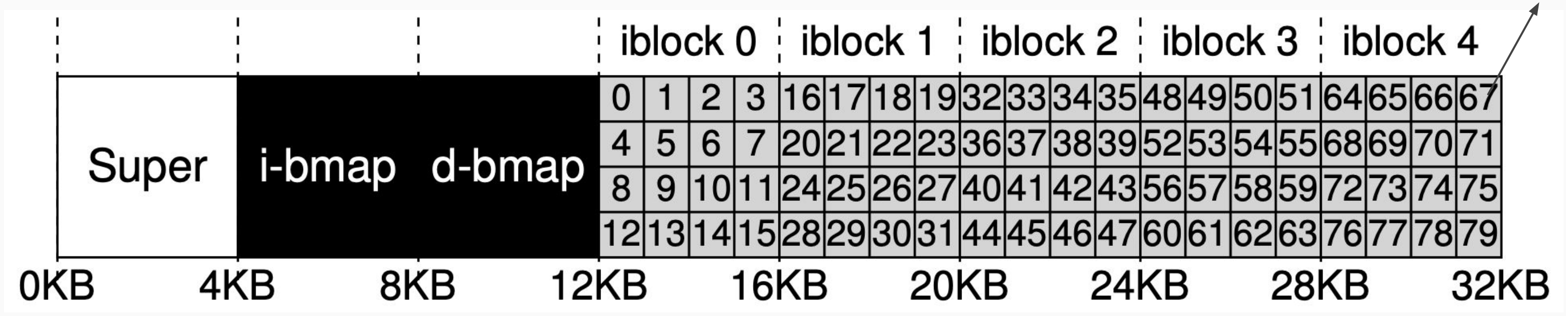


Inode

- Holds metadata for file
  - Length, permissions etc.
- Referred using a number
- Find inode using i-number

Read inode 32?

256 bytes



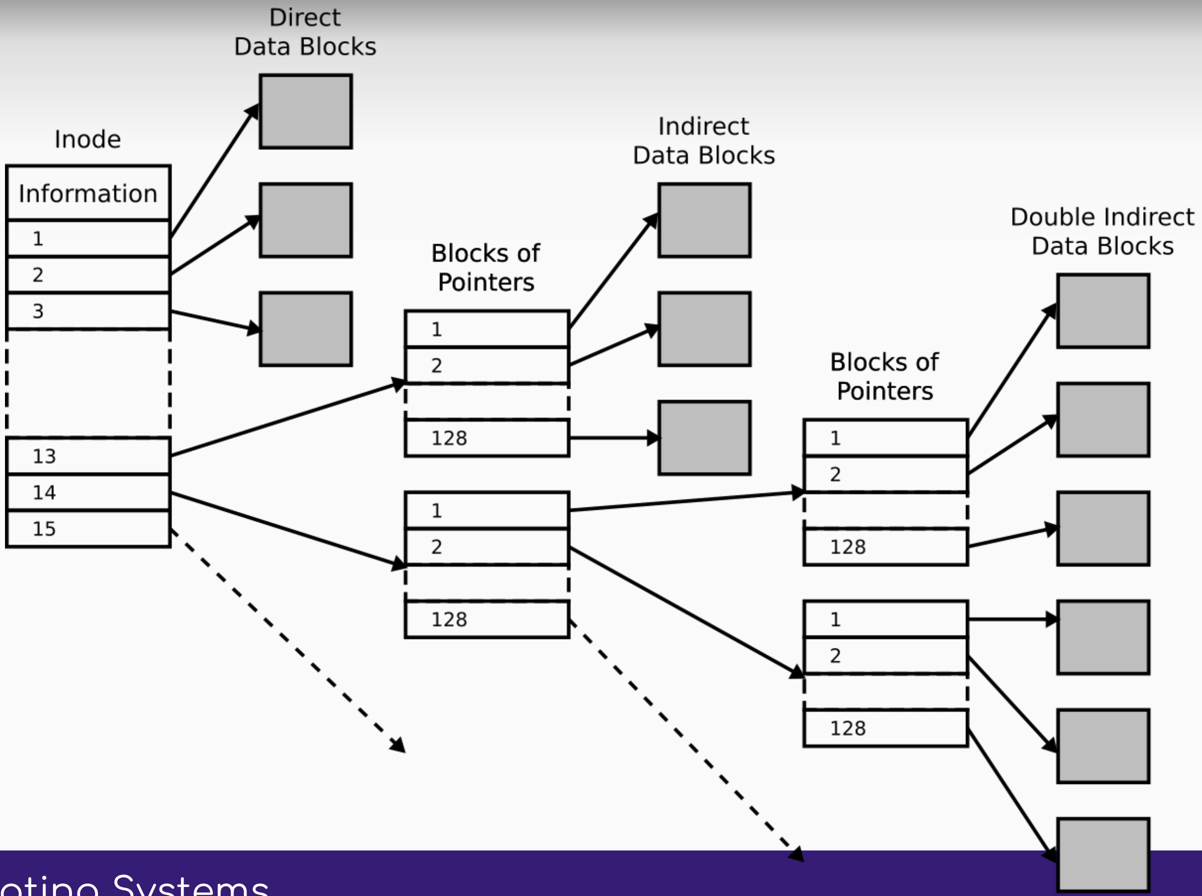
Inode

- Holds metadata for file
  - Length, permissions etc.
- Referred using a number
- Find inode using i-number

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

- How to find data blocks?
  - Direct pointers?

Multi-level Index



Multi-level Index

- For indexing bigger files
- Special indirect pointer
  - Points to block that contains pointers to the file blocks
- Inode may contain both direct pointers and indirect pointer
  - Large files use indirect pointer
- Much larger files will have multiple indirect pointers
  - Pointers to indirect blocks, last of which points to data blocks

Extents

- Extent is pointer plus length
    - Use extents instead of pointers
- Store extent to specify on-disk location
- May have multiple extents based on available space

- Pointer-based approaches are flexible but store lot of metadata per file
- Extent-based approaches are less flexible but more compact

Directories in SFS

- Special types of files
- Has an inode number itself
- List of filename-inode pairs
  - Can also be trees
- Contains two special entries
  - .
  - ..
- Deleted files in the directory are marked with inode number 0

Directories in SFS

- Contents

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

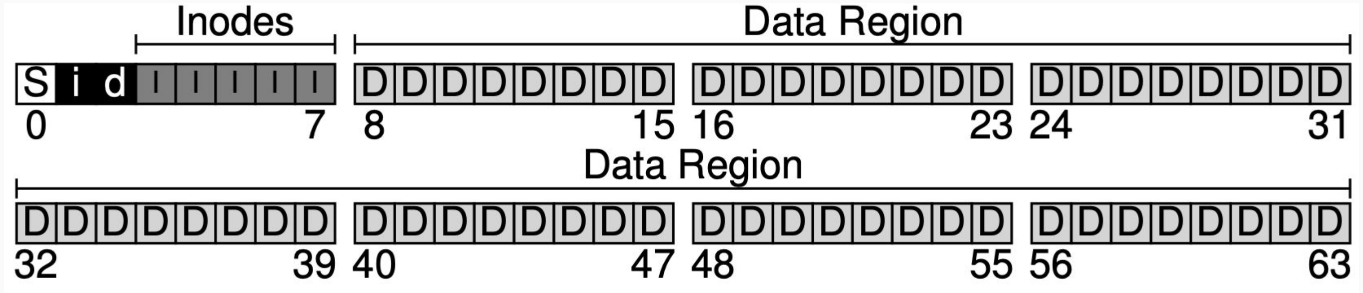
Free Space Management

- Track inodes and blocks for free space
- Used for allocating new files
- SFS uses two bitmaps
  - Inode bitmap tells which inode (number) is available
  - Data block bitmap tells which blocks are available

- Pre-allocation policy
  - Heuristics when allocating blocks
  - Sometimes the file system looks for multiple free blocks to allocate contiguous disk space to files

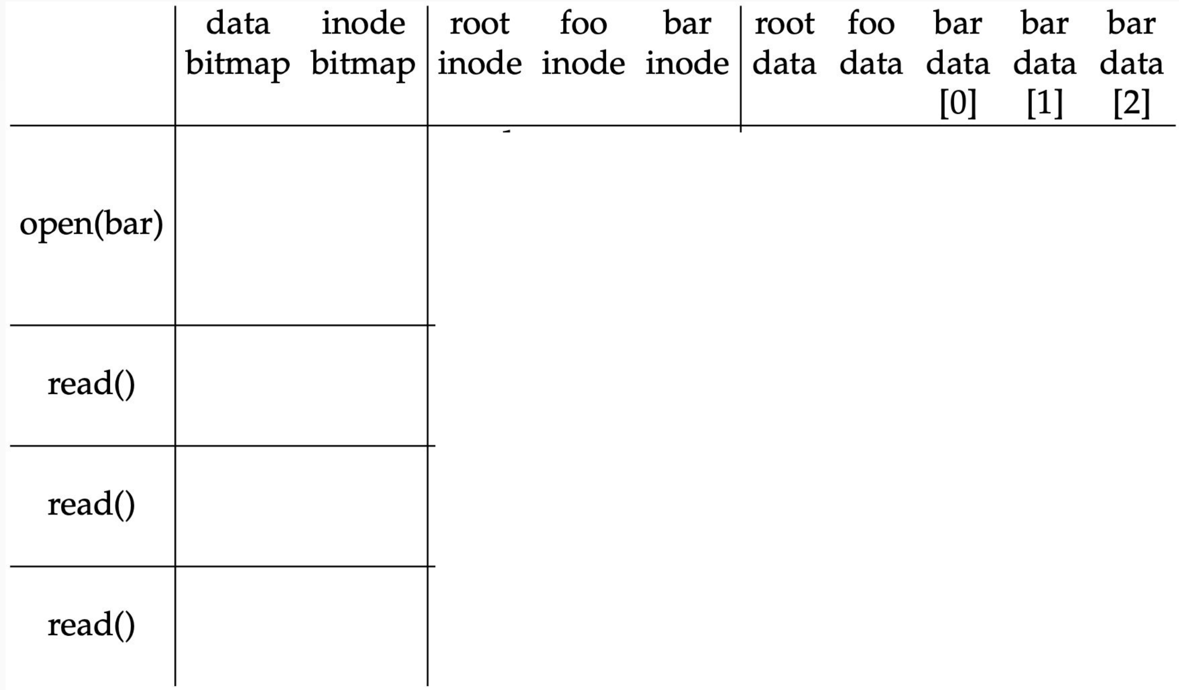
Data Structures in SFS

- Storage in blocks
  - User data
  - Information about each of the files
  - Allocation structures
  - Superblock
    - Contains details about the file system like inode table, no. of inodes, blocks, etc.



Reading from Disk

```
1.  open ("/foo/bar", O_RDONLY)
2.  read ()
3.  close ()
```

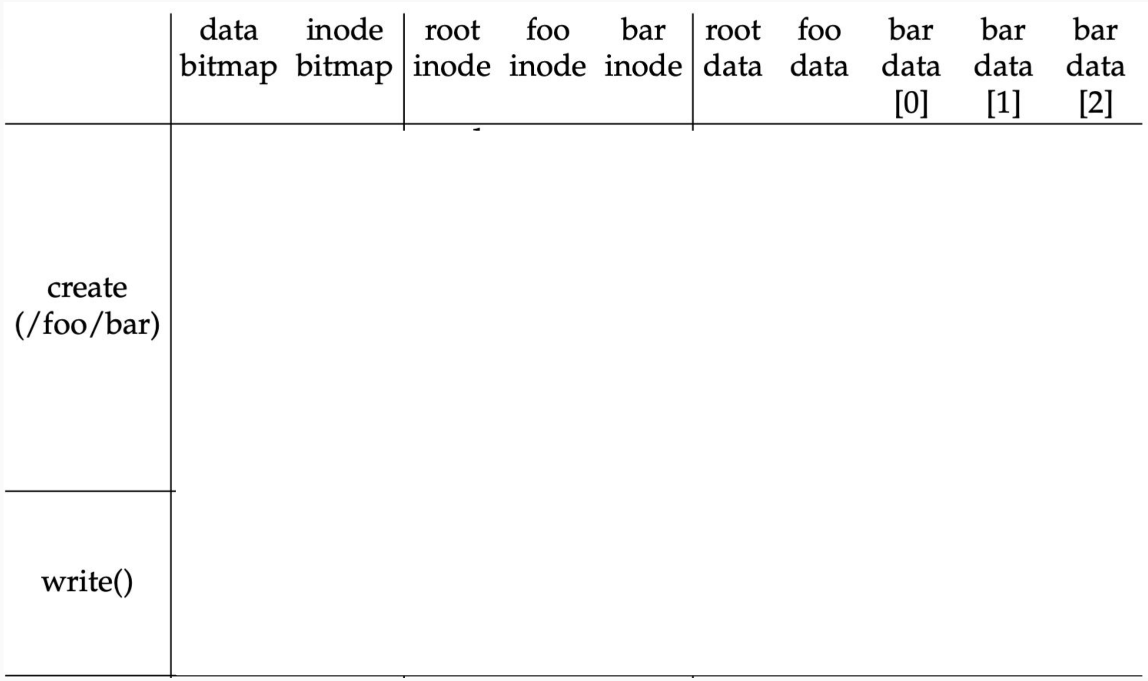


	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)										
read()										
read()										
read()										

Writing to Disk

```
1.  open ("/foo/bar", O_WRONLY)
2.  write ()
3.  close ()
```

- May allocate blocks



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)										
write()										

Caching and Buffering

- Reads and writes to disks are expensive
  - Many I/Os slow the process
- Cache in system memory (DRAM)
  - Fixed-size cache holds blocks
  - Can use similar strategies like LRU
- Partitioning the DRAM
  - Static
    - At boot time (say, 10% of total memory)
  - Dynamic
    - As need be

Write Buffering

- Writes need to go to disk for persistence
- Buffer writes and perform them together
    - Performance benefits
- Delay writes and batch some updates into set of I/Os
- Avoid unnecessary writes
    - Create file followed by delete need not modify disk
- Modern file systems buffer writes
    - 5 to 30 seconds
    - System crashes before writes are propagated to disk result in lost updates
- Databases avoid this by calling fsync

fsync

- `write()` writes data to persistent storage at some time in the future
    - Buffer writes in memory
- Many times, we need to force disk writes
- Use `fsync` to force writes to disk
    - Force all *dirty* data to be written

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
                S_IRUSR|S_IWUSR);
int rc = write(fd, buffer, size);
rc = fsync(fd);
```

Reading/Writing to Files

- Normally, programs use system calls to do file I/O
    - E.g., read, write
- Works on a ***copy*** of the data
    - Open a file; read blocks of data into program’s address space
    - Use/modify these blocks w/o affecting the actual file
    - Invoke write system call to copy the changes back to the file
    - Efficient for small files
- Alternatively, map file contents directly into program’s address space

Memory-mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
    - A file is initially read using demand paging
    - A page-sized portion of the file is read from the file system into a physical page
    - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls

Memory-mapped Files

- Allows several processes to map the same file allowing the pages in memory to be shared
- Treat memory as a write-back cache for disk.
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when we scan for dirty pages

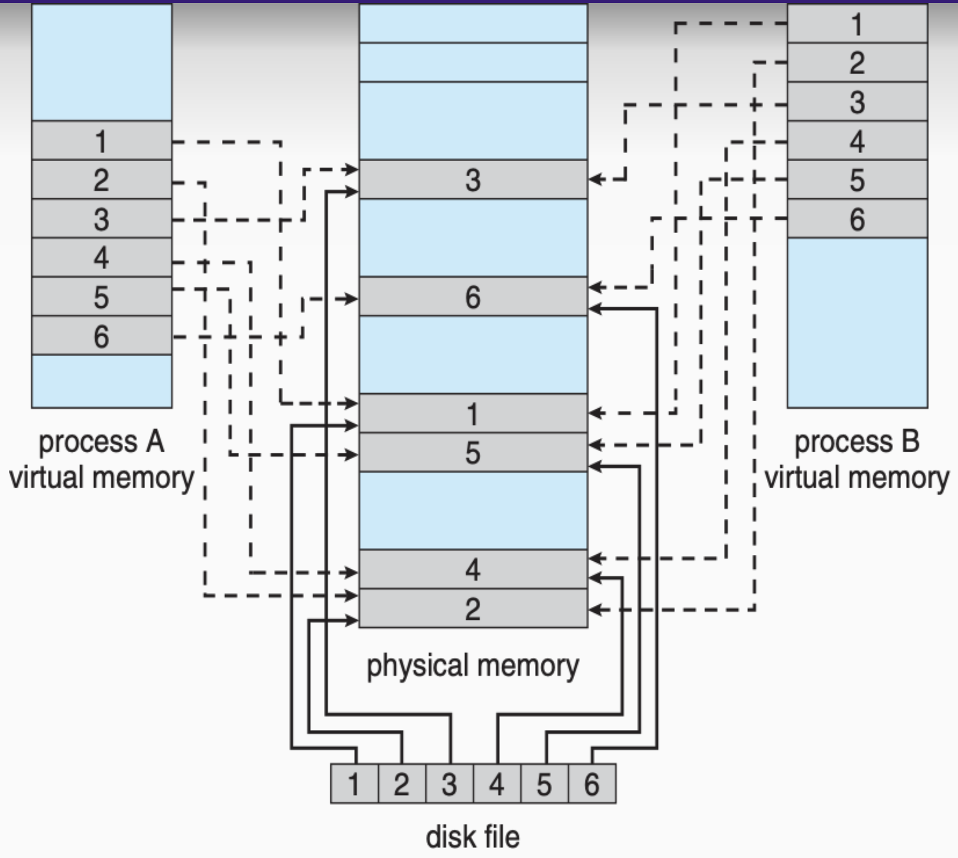
Memory-mapped Files

- Advantages
  - Transparency
    - Can operate on the bytes in the file as if they are part of memory
  - Zero copy I/O
    - Do not need to copy file data from kernel buffers into user memory
    - Changes the program's page table entry to point to the physical page frame containing that portion of the file
    - Kernel is responsible for copying data back and forth to disk
  - Pipelining
    - Operate on the data in the file once the page tables are set up
    - Does not need to wait for the entire file to be read into memory
    - With multiple threads, a program can use explicit read/write calls to pipeline disk I/O, but it needs to manage the pipeline itself.

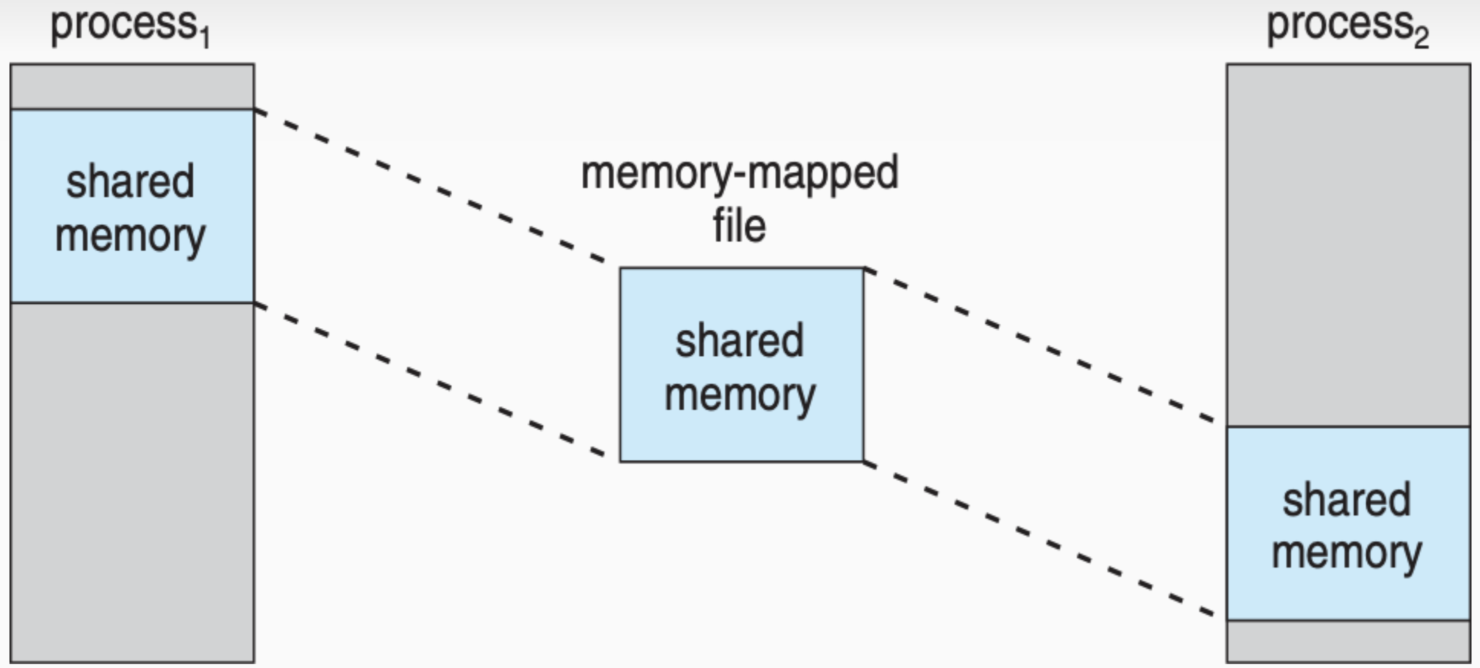
Memory-mapped Files

- Advantages
    - Interprocess communication
        - Two or more processes can share information instantaneously
    - Large files
        - Managing large files is easy as the bookkeeping is done by the operating system

Memory-mapped Files



Memory-mapped Files



File System - Consistency

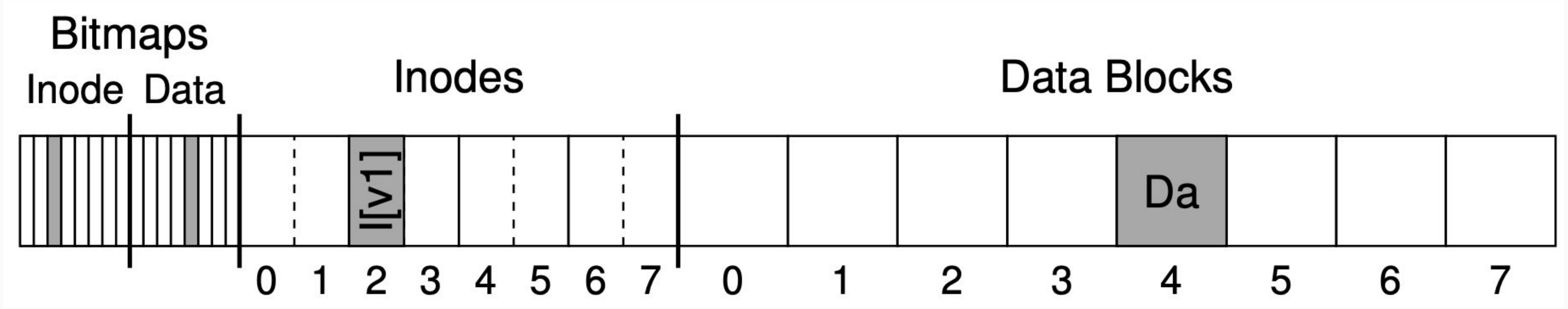
- Implements the abstractions: files, directories, etc.
- Data is stored persistently
    - Stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs)
- How does it store/update the persistent data in case of a power loss or system crash?
    - What if we are in the middle of writing the data to the disk?
    - Example: Suppose we want to perform A & B to complete an op
        - What if A completes and B does not?
        - Or vice versa?
    - May leave the system in inconsistent state

Surviving Crashes

- File system data structures must persist despite power loss or system crash
  - Crashes make updating persistent data structures difficult
- FSCK (file system checker)
  - traditional
- Journaling
  - Quicker recovery

Example

- Append single data block to existing file
  - Open the file
  - `lseek` to move the file offset to end of file
  - Write block and close

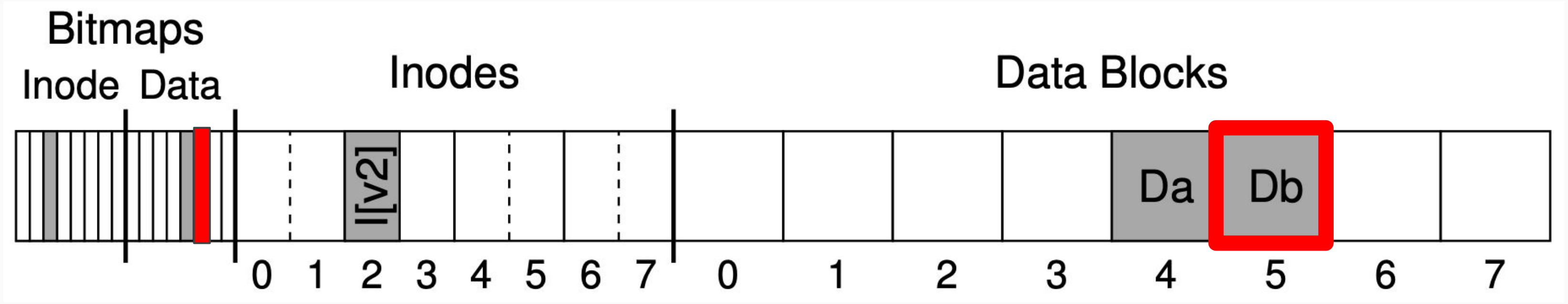


```
owner            : A                pointer    : 4
permissions      : read-write       pointer    : null
size             : 1                pointer    : null
```

Example

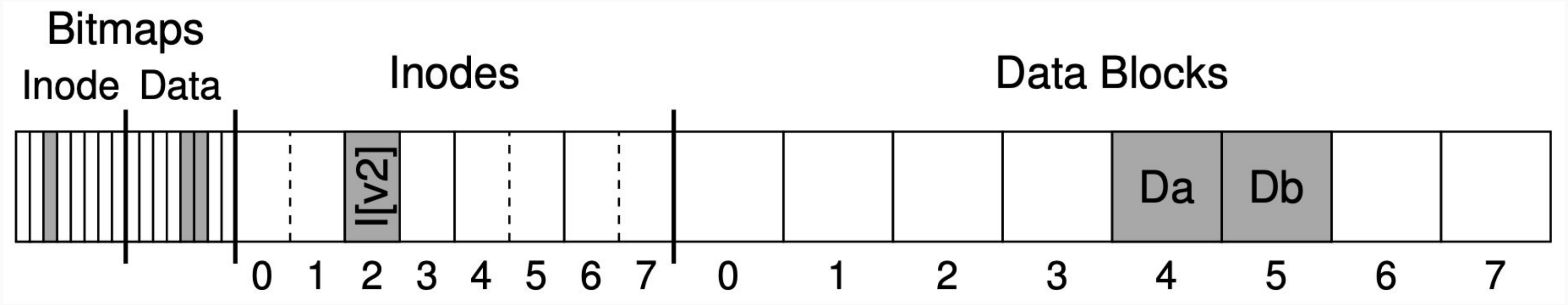
- Need three writes
    - One for inode
    - One for bitmap
    - One for data block

```
owner          : A             pointer    : 4
permissions    : read-write    pointer    : 5
size           : 2             pointer    : null
```



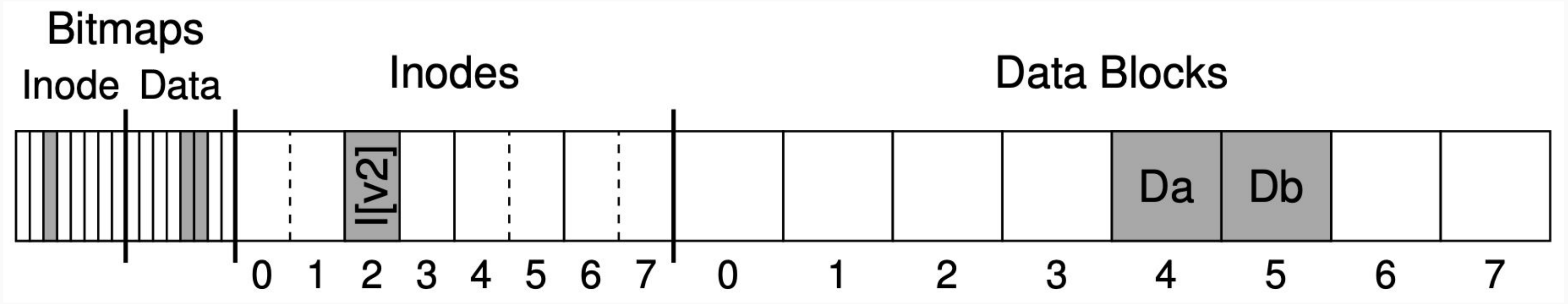
Example

- Assume single write succeeded
  - Data block Db written to disk (or)
  - Updated inode I[v2] written to disk (or)
  - Updated bitmap written to disk



Example

- Assume two writes succeeded
  - Updated inode and data block Db written to disk (or)
  - Updated inode and bitmap written to disk (or)
  - Updated bitmap and data block Db written to disk



Crash Consistency Problem

- Different crash scenarios possible
  - May cause inconsistency
  - May lead to space leaks
  - May return garbage data
- Ideally: move from one consistent state to another
  - Atomic updates
- Cannot guarantee as writes happen one after another

File System Checker

- Allow inconsistencies and fix when rebooting
- `fsck` tool
    - Find inconsistency and fix it
    - Make file system metadata consistent
    - Runs before file system mounted
    - Phases
        - Superblock check (sanity checks like file system size is more than number of allocated blocks)
        - Free blocks check
        - Inode state
        - Inode links
        - Duplicates and bad blocks
        - Directory checks

File System Checker

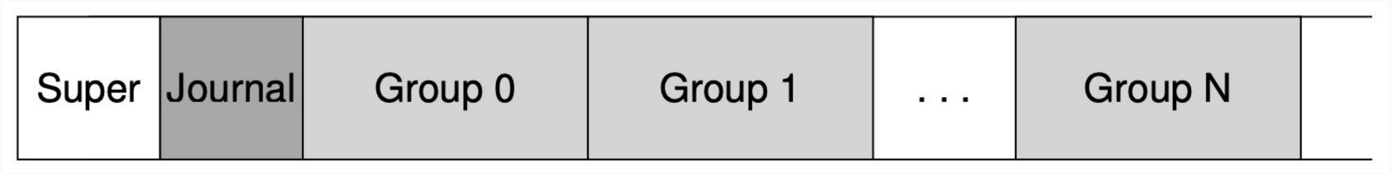
- Requires knowledge of file system
- Too slow
  - Prohibitive performance

Journaling

- Write-ahead logging
- Before updating the disk, update a log on disk
- In case of crash
  - Look up the log and try again
- More updates for faster recovery

Journaling

- Before updating disks, note the changes in log
- When system crashes lookup the log to fix
- Journal is placed in the same file system or separate device (persistently)



Data Journaling

- Log for the previous example
  - ...
  - Tx Begin (contains transaction ID)
  - I[v2]
  - B[v2]
  - Db
  - Tx End
  - ...
- Physical Logging - exact physical contents of update
- Logical Logging - logical representation of update

Data Journaling

- Checkpointing
  - Transaction is written to the log
  - Overwrite old structures on the file
  - Issue writes to I[v2], B[v2] and Db on disk

- Can write one block at a time : slow
- Can write the blocks -- TxB, I[v2], B[v2], Db, TxE -- all at once to the journal
  - Disk may internally write TxB, I[v2], B[v2] and TxE but not Db

Data Journaling

- Write all blocks except Tx End in the first write
  - TxB, I[v2], B[v2], Db
  - Issue another write for TxE (atomic)

- Updated Protocol
  - Journal write
    - Write contents to log
  - Journal commit
    - Write transaction commit block (TxE)
  - Checkpoint
    - Write contents to the disk

Recovery

- Crash before writing to the log
    - Skip the update
- Crash after committing to the log but before checkpoint
    - Scan the log
    - Look for committed transactions
    - Replay the transaction
    - Also known as **Redo logging**
    - May perform some updates again

Finite Log

- More and more transactions will fill the log
- Problems
  - Longer recovery for a large log
  - Cannot commit more transactions
- Use **circular log**
  - Circular data structure - reuse the log over and over
  - Free the space after a transaction is checkpointed
  - Maintain a journal superblock
    - Contains oldest and newest non-checkpointed transactions

Data Journaling Protocol

- Journal write
  - Write contents of the transaction to the log
- Journal commit
  - Write transaction commit block
- Checkpoint
  - Write the contents to their disk-locations
- Free
  - Mark transaction free by updating journal superblock

Metadata Journaling

- Data journaling - Journaling **all** data
  - Slow performance
- Ordered Journaling (metadata journaling)
  - Speed up performance
  - Do not write user data to the journal
    - TxB | I[v2] | B[v2] | TxE
  - Write data block directly to the disk (actual location)
  - When to write data block to the disk?

Metadata Journaling

- Write Db to disk after transaction completes?

Metadata Journaling

- Write Db to disk after transaction completes?
  - If I[v2] and B[v2] written to disk, but Db is not:
    file system is consistent but I[v2] may point to garbage data
  - Modify the protocol
    - Write Db first to the disk
  - Protocol
    - Data write
    - Journal metadata write
    - Journal commit
    - Checkpoint metadata
    - Free
- Metadata journaling is more commonly used journaling

Block Reuse

- Example
  - Suppose we are using metadata journaling
  - User is adding an entry to a directory (creating a file)
  - Contents are written to the journal (metadata)
    - `TxB | I[v] (ptr → 1000) | D (addr → 1000) | TxE`
  - Suppose user deletes the directory and frees up 1000 and creates a new file (reuses block 1000)
    - `TxB | I[v] | D | TxE | TxB | I[newfile] (ptr → 1000) | TxE`
    - Only inode of `newfile` is in journal and the block is at 1000
  - Assume a crash happens
- How to solve this?