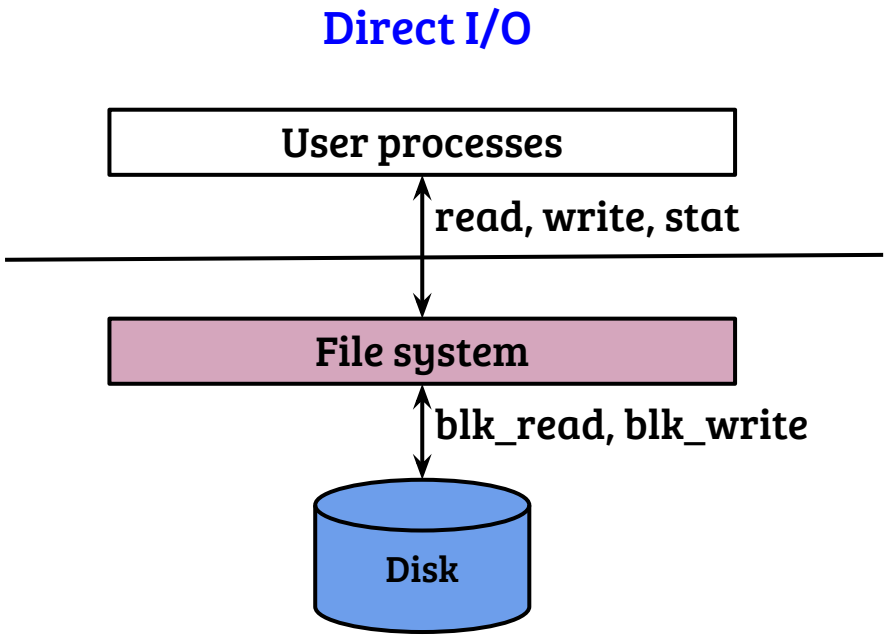
Operating Systems

Filesystem: caching and consistency

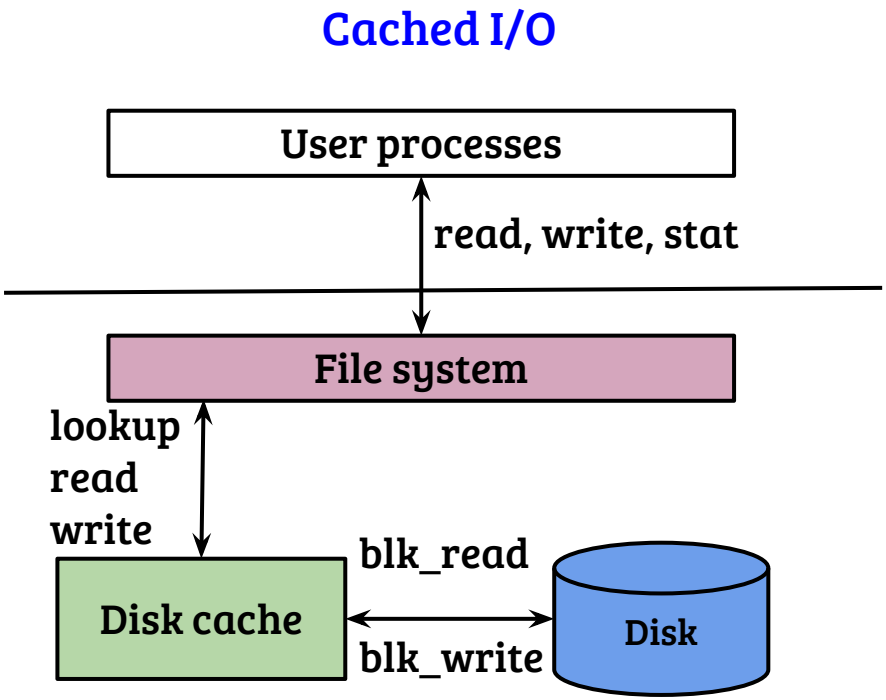
Debadatta Mishra, CSE, IITK

Direct I/O



- In direct I/O, operations carried out directly on the device
- I/O is performed directly to/from user buffer (at block boundary)
- Performance can be degraded severely
- O_DIRECT and O_SYNC flags in Linux/Unix open( ) system call
- Can be useful for applications like databases

Cached I/O

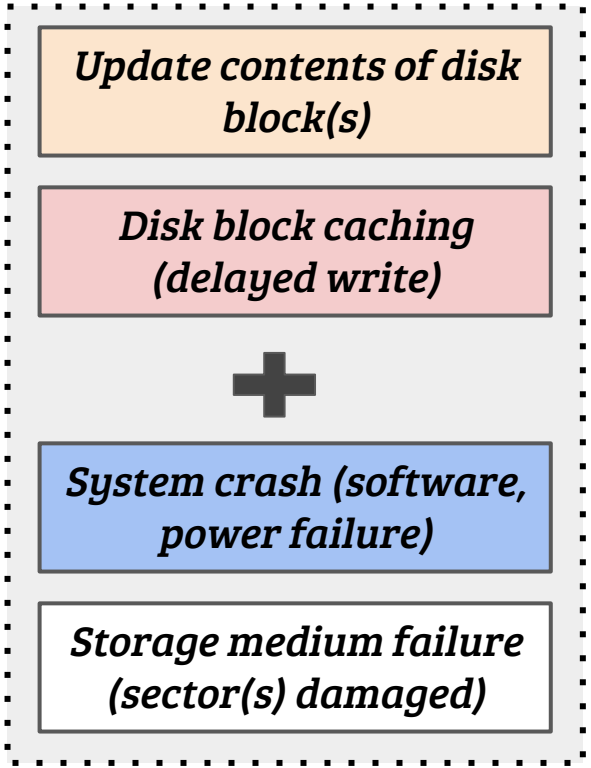


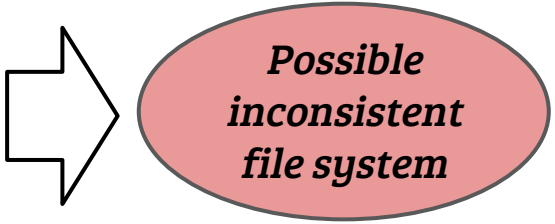
- Some disk blocks are cached in memory, can be used to serve subsequent requests
- Disk cache size << Disk size, How to manage?
- Cache write policy: write through, write back
- What is the lookup key?
- Hiding miss latency: prefetching

Consistency: different perspectives

- What is a consistent file system?
- Consistent w.r.t. system call level guarantees
  - Example-1: If a write( ) system call is successful then, . . .
  - Example-2: If a file creation is successful then, . . .
  - Difficult to achieve with asynchronous I/O, why?
- Consistent w.r.t. file system invariants
  - Example-1: If a block is pointed to by an inode data pointers then, …
  - Example-2: If a directory entry contains an inode then, . . .
  - Possible, require special techniques

File system inconsistency: root causes



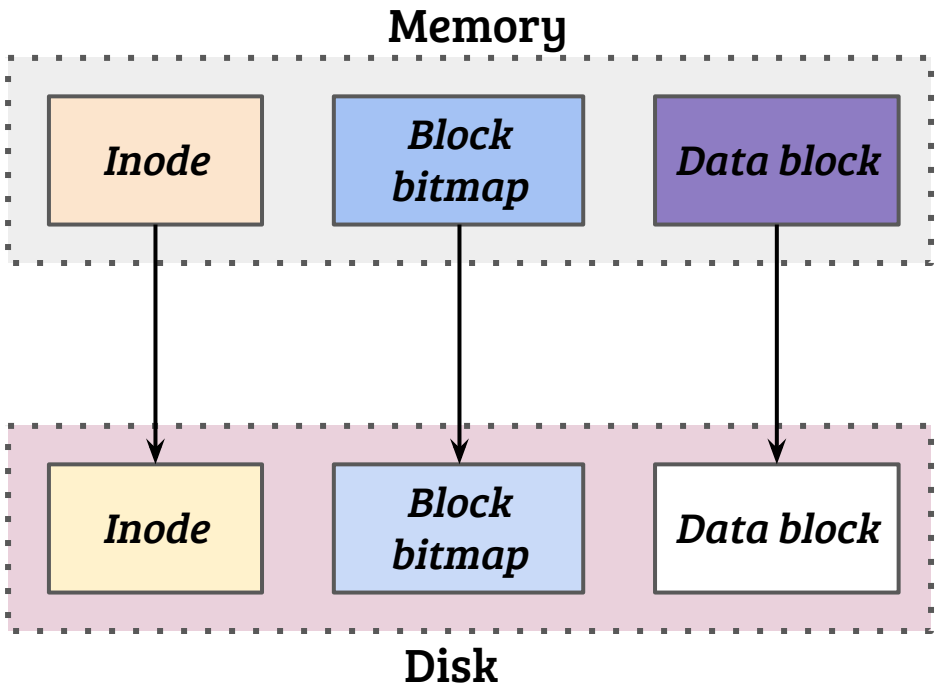


- No consistency issues if user operation translates to read-only operations on the disk blocks

- Only one block is written, can it result in an inconsistent state?

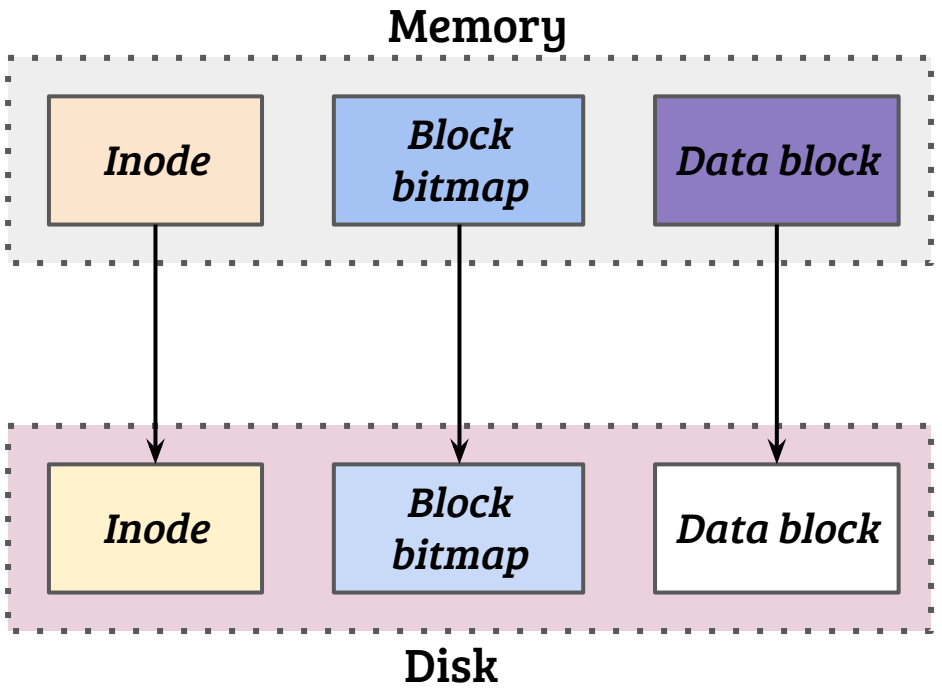
Example: Append to a file

- Assume operations ---(i) seek to the end of file, (ii) allocate a new block, (iii) write user data
- All structures read to memory to perform write
- Update inode → size and block pointers
- Update block bitmap → set used block bit
- Update data block
- Write back to the disk



Example: Append to a file

- Assume operations ---(i) seek to the end of file, (ii) allocate a new block, (iii) write user data
- All structures should be in memory to perform write
- Update inode → size and block pointers
- Update block bitmap → set used block bit
- Update data block
- Write back to disk



Three different write operations required, what if failure strikes before all of them complete?

Failure scenarios and implications

Written	Yet to be written	Implications
Data block	Inode, Block bitmap	File system is consistent (Lost data)
Inode	Block bitmap, Data block	File system is inconsistent (correctness issues)
Block bitmap	Inode, Data block	File system is inconsistent (space leak)

Failure scenarios and implications

Written	Yet to be written	Implications
Data block, Block bitmap	Inode	File system is inconsistent (space leakage)
Inode, Data block	Block bitmap	File system is inconsistent (correctness issues)
Inode, Block bitmap	Data block	File system is consistent (Incorrect data)

- Atomicity: Maintain a consistent file system state
- Not easy as disk commits one write at a time

File system consistency with *fsck*

- During FS mount, check if it had been cleanly unmounted when it was last used, How to know?
- Perform sanity checks at different levels
  - Superblocks
  - Free blocks
  - Inode checks
  - Directory content check
- May not solve all problems, example?

File system consistency with *journaling*

- Idea of journaling used in database systems extensively
- Idea: Before the actual operation, note down the operations (journal write), update the file system (checkpoint)
- Example journal entry: [Start] [Inode block] [Block bitmap] [Data block] [End]
- Recovery: journal entries inspected during mount

Issues:

- Journal write failure, block/device layer reordering
- Writes become very slow

Metadata journaling: performance-reliability tradeoff

- Idea: Data block is not part of the journal entry
- Practical with tolerable performance overheads
- Example journal entry: [Start] [Inode block] [Block bitmap] [End]

What should be the order?

- Data block write followed by journal write and checkpoint, why?